

emUSB-Device

USB Device stack

User Guide & Reference Manual

Document: UM09001
Software Version: 3.40.0
Revision: 0
Date: March 31, 2021



A product of SEGGER Microcontroller GmbH

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 (SEGGER) assumes no responsibility for any errors or omissions. SEGGER makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. SEGGER specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of SEGGER. 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.

© 2010-2021 SEGGER Microcontroller GmbH, Monheim am Rhein / Germany

Trademarks

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

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

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel. +49 2173-99312-0
Fax. +49 2173-99312-28
E-mail: ticket_emusb@segger.com*
Internet: www.segger.com

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at <https://www.segger.com/legal/privacy-policy/>.

Manual versions

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

As of version 3.00 the history has been reset. Older history entries can be found in older versions of this document.

Print date: March 31, 2021

Software	Revision	Date	By	Description
3.40.0	0	210331	RH	Add new chapter <i>Vendor Specific Class (VSC)</i> .
3.38.0	0	210129	RH	DFU class: <ul style="list-style-type: none"> Added function <code>USBD_DFU_AddAlternateInterface()</code>. Printer class: <ul style="list-style-type: none"> Added function <code>USB_PRINTER_SetClass()</code>.
3.36.3	0	201218	YR	Update to latest software version.
3.36.2	0	201215	RH	BULK class: <ul style="list-style-type: none"> Added function <code>USBD_BULK_PollForTX()</code>. CDC class: <ul style="list-style-type: none"> Added functions <code>USBD_CDC_PollForRX()</code>, <code>USBD_CDC_PollForTX()</code> and <code>USBD_CDC_ReceivePoll()</code>. Added new section <i>Timeout handling</i> Update to latest software version.
3.36.1	0	201030	YR	Update to latest software version.
3.36.0	0	201012	YR	Audio class: <ul style="list-style-type: none"> Updated API with new function and structure names. Update to latest software version.
3.34.3	0	200825	YR	Update to latest software version.
3.34.2	0	200723	YR	Update to latest software version.
3.34.1	0	200610	RH	Added section <i>Synopsys DWC2 driver (DMA mode)</i>
3.34.0	0	200320	RH	BULK class: <ul style="list-style-type: none"> Added function <code>USBD_BULK_AddAlternateInterface()</code>.
3.32.0	0	200226	RH	Added CCID chapter.
3.30.0	0	200127	YR	Added MIDI chapter.
3.28.1	0	191213	YR	Update to latest software version. HID class: Added <code>USBHID_Read()</code> and <code>USBHID_Write()</code> functions.
3.28.0	0	191129	YR	Printer class: <ul style="list-style-type: none"> Added function <code>USB_PRINTER_SetOnVendorRequest()</code>.
3.26	0	190830	RH	USB Core: <ul style="list-style-type: none"> Added function <code>USBD_GetVersion()</code>. Added function <code>USBD_SetCheckAddress()</code>. UVC class: <ul style="list-style-type: none"> Added function <code>USBD_UVC_WriteEx()</code>.
3.24	0	190701	RH	Added chapter <i>USB Device Firmware Upgrade (DFU)</i>
3.22	0	190617	RH	BULK class: <ul style="list-style-type: none"> Added function <code>USBD_BULK_PollForRX()</code>. Added function <code>USBD_BULK_ReceivePoll()</code>. USB Core: <ul style="list-style-type: none"> Added function <code>USBD_WaitForEndOfTransferEx()</code>. Added function <code>USBD_ReceivePoll()</code>. Added function <code>USBD_EnableSuperSpeed()</code>. Added function <code>USBD_SetWebUSBInfo()</code>. MSD class: <ul style="list-style-type: none"> Added function <code>USBD_MSD_Poll()</code>. Add section <i>XHCI driver</i> Add chapter <i>USB Video device Class (UVC)</i>
3.20	0	190515	YR	Update to latest software version. SmartMSD was renamed to VirtualMSD. Function prefixes changed from "SMSD" to "VMSD". HID class: Added <code>USBD_HID_ReadReport()</code> function.

Software	Revision	Date	By	Description
3.18b	0	190304	YR	Update to latest software version.
3.18a	0	190228	YR	Update to latest software version. Bulk Host API: <ul style="list-style-type: none"> Removed USBBULK_ResetPipe() Added USBBULK_ResetINPipe() Added USBBULK_ResetOUTPipe() USBBULK_DEV_INFO received an additional member - "Speed"
3.18	0	181126	RH	Update to latest software version.
3.16	0	181005	YR	Update to latest software version. MTP class: <ul style="list-style-type: none"> USBD_MTP_SendEvent() description updated.
3.14	0	180719	RH	Added section <i>Device driver specifics</i> on page 663 BULK class: <ul style="list-style-type: none"> Added functions USBD_BULK_ReadAsync() and USBD_BULK_WriteAsync(). CDC class: <ul style="list-style-type: none"> Added functions USBD_CDC_ReadAsync() and USBD_CDC_WriteAsync().
3.12	0	180504	RH	USB Core: <ul style="list-style-type: none"> Added function USBD_RemoveOnEvent(). Removed functions USBD_SetLogFunc() and USBD_SetWarnFunc(). Audio class: <ul style="list-style-type: none"> Added function USBD_AUDIO_Read_Task(). Added function USBD_AUDIO_Write_Task().
3.10	0	180322	RH	BULK class: <ul style="list-style-type: none"> Added function USBD_BULK_Add_Ex(). Update to latest software version. Added Audio chapter.
3.08	0	180212	RH	USB Core: <ul style="list-style-type: none"> Added function USBD_RegisterSCHook(). Added function USBD_AddePEX().
3.06e	0	180112	RH	Update to latest software version.
3.06d	0	171219	YR	Update to latest software version.
3.06c	0	171204	RH	USB Core: <ul style="list-style-type: none"> Added I/O functions.
3.06b	0	171013	YR	Update to latest software version. Corrected USBD_WriteEP0FromISR name (was USB_WriteEP0FromISR).
3.06	0	170915	RH	Printer class: <ul style="list-style-type: none"> Added function USB_PRINTER_ConfigIRQProcessing(). Added function USB_PRINTER_TaskEx(). USB Core: <ul style="list-style-type: none"> Added USBD_SetCacheConfig(). Chapter "Getting started" revised.
3.04	0	170724	YR	Update to latest software version. Added chapter "emUSB-Device-IP"
3.02q	0	170717	YR	Update to latest software version.
3.02p	0	170714	RH	Update to latest software version.
3.02o	1	170710	YR	Chapter Combining USB components: <ul style="list-style-type: none"> Added information on the MSD+MTP combination feature. Added Chapter "Profiling with SystemView".
3.02o	0	170701	RH	Major revision of the manual. <ul style="list-style-type: none"> Manual converted to text processor emDoc.
3.02n	0	170612	SR	Update to latest software version.
3.02m	0	170608	RH	Update to latest software version.
3.02l	0	170602	SR	Update to latest software version.
3.02k	0	170410	RH	Function USBD_AddEP(): Parameter 'Interval' changed.
3.02j	0	170301	RH	Update to latest software version.
3.02i	0	170126	SR	Update to latest software version.

Software	Revision	Date	By	Description
3.02h	0	170126	SR	Update to latest software version.
3.02g	0	170123	SR	Update to latest software version.
3.02f	0	170104	RH	Update to latest software version.
3.02e	0	161215	RH	Update to latest software version.
3.02d	0	161118	RH	Update to latest software version.
3.02c	0	161103	RH	Update to latest software version.
3.02b	0	161028	RH	Chapter VirtualMSD: <ul style="list-style-type: none"> Renamed functions according to the emUSB V3 conventions.
3.02a	10	161018	RH	Update to latest software version.
3.02	9	161007	SR	Chapter CDC: <ul style="list-style-type: none"> Updated Overview - Added Windows 10 Support. Updated Installing the driver. Section verification combined with Installing driver. Updated section The .inf file Added new section: Signing the package Testing communication to the USB device updated.
3.02	8	160930	RH	Chapter CDC Data structures: <ul style="list-style-type: none"> Removed CTS from <code>USB_CDC_SERIAL_STATE</code>. Chapter BULK communication: <ul style="list-style-type: none"> Removed description of the Segger USB driver(not necessary any more) BULK host API and sample applications support for Linux and MacOSX added. Added new function <code>USBD_BULK_SetMSDDescInfo()</code>. Added new function <code>USBBULK_GetDevInfoByIdx()</code>. Removed description of deprecated host API functions. Chapter VirtualMSD: <ul style="list-style-type: none"> Added memory usage calculation. Many minor corrections.
3.00g	7	160822	RH	Update to latest software version. Chapter Target OS Interface: <ul style="list-style-type: none"> New advanced OS layer interface. Chapter Mass Storage Device Class (MSD): <ul style="list-style-type: none"> Added new function <code>USBD_MSD_RequestRefresh()</code>
3.00f	6	160720	YR	Update to latest software version. Chapter VirtualMSD: <ul style="list-style-type: none"> Changed function prefix to VMMSD. Removed obsolete functions.
3.00e	5	160708	RH	Update to latest software version.
3.00d	4	160608	YR	Update to latest software version.
3.00c	3	160523	YR	Update to latest software version. Chapter Bulk communication: <ul style="list-style-type: none"> Added paragraph "Writing your own host driver".
3.00b	2	160427	YR	Update to latest software version.
3.00a	1	160415	SR	Chapter USB Core functions: <ul style="list-style-type: none"> Updated prototype for <code>USBD_SetMaxPower</code>. Chapter HID: <ul style="list-style-type: none"> Added new function for Setting a callback for <code>SET_REPORT</code>. Chapter Debugging: <ul style="list-style-type: none"> Changed all prototypes from <code>USB_*</code> to <code>USBD_*</code>.
3.00	0	160212	YR	Initial Version

About this document

Assumptions

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

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

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

How to use this manual

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

Typographic conventions for syntax

This manual uses the following typographic conventions:

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

Table of contents

1	Introduction	24
1.1	Overview	25
1.2	emUSB-Device features	26
1.3	emUSB-Device components	27
1.3.1	emUSB-Device-Bulk	27
1.3.2	emUSB-Device-MSD	27
1.3.2.1	Purpose of emUSB-Device-MSD	27
1.3.2.2	Typical applications	28
1.3.2.3	emUSB-Device-MSD features	28
1.3.2.4	How does it work?	28
1.3.3	emUSB-Device IP-over-USB	29
1.3.3.1	Typical applications	29
1.3.4	emUSB-Device-VirtualMSD	29
1.3.4.1	Typical applications	29
1.3.5	emUSB-Device-CDC	29
1.3.5.1	Typical applications	29
1.3.6	emUSB-Device-HID	29
1.3.6.1	Typical applications	29
1.3.7	emUSB-Device-MTP	30
1.3.7.1	Typical applications	30
1.3.8	emUSB-Device-Printer	30
1.3.8.1	Typical applications	30
1.3.9	emUSB-Device-RNDIS	30
1.3.9.1	Typical applications	30
1.3.10	emUSB-Device-CDC-ECM	31
1.3.10.1	Typical applications	31
1.4	Requirements	32
1.4.1	Target system	32
1.4.2	Development environment (compiler)	32
1.5	File structure	33
1.6	Multithreading	34
2	Background information	35
2.1	USB	36
2.1.1	Short Overview	36
2.1.2	Important USB Standard Versions	36
2.1.3	USB System Architecture	37
2.1.4	Transfer Types	39
2.1.5	Setup phase / Enumeration	39
2.1.6	Product / Vendor IDs	39

2.2	Predefined device classes	40
2.3	USB hardware analyzers	41
2.4	References	42
3	Getting started	43
3.1	How to setup your target system	44
3.1.1	Take a running project	44
3.1.2	Add emUSB-Device files	44
3.1.3	Configuring debugging output	44
3.1.4	Add hardware dependent configuration	45
3.1.5	Prepare and run the application	45
3.2	Updating emUSB-Device	47
3.3	emUSB-Device Configuration	48
3.3.1	USB_DEVICE_INFO	48
3.3.2	Additional required configuration for emUSB-MSD	49
3.3.3	Descriptors	49
3.3.4	Compile-time configuration	50
3.4	Host OS specifics	51
3.4.1	Windows registry	51
3.4.1.1	Cleaning the Windows registry	51
4	USB Core	52
4.1	Overview	53
4.2	Target API	54
4.2.1	USB basic functions	57
4.2.1.1	USB_D_GetState()	57
4.2.1.2	USB_D_GetSpeed()	58
4.2.1.3	USB_D_Init()	59
4.2.1.4	USB_D_IsConfigured()	60
4.2.1.5	USB_D_Start()	61
4.2.1.6	USB_D_Stop()	62
4.2.1.7	USB_D_DeInit()	63
4.2.1.8	USB_D_GetVersion()	64
4.2.2	USB configuration functions	65
4.2.2.1	USB_D_AddDriver()	65
4.2.2.2	USB_D_SetISREnableFunc()	66
4.2.2.3	USB_D_SetAttachFunc()	67
4.2.2.4	USB_D_AddEP()	68
4.2.2.5	USB_D_AddEPEX()	69
4.2.2.6	USB_D_SetDeviceInfo()	70
4.2.2.7	USB_D_SetClassRequestHook()	71
4.2.2.8	USB_D_SetVendorRequestHook()	72
4.2.2.9	USB_D_SetIsSelfPowered()	73
4.2.2.10	USB_D_SetMaxPower()	74
4.2.2.11	USB_D_SetOnEvent()	75
4.2.2.12	USB_D_RemoveOnEvent()	77
4.2.2.13	USB_D_SetOnRxEP0()	78
4.2.2.14	USB_D_SetOnRXHookEP()	79
4.2.2.15	USB_D_SetOnSetup()	80
4.2.2.16	USB_D_SetOnSetupHook()	81
4.2.2.17	USB_D_WriteEP0FromISR()	82
4.2.2.18	USB_D_EnableIAD()	83
4.2.2.19	USB_D_SetCacheConfig()	84
4.2.2.20	USB_D_RegisterSCHook()	85
4.2.2.21	USB_D_AssignMemory()	86
4.2.2.22	USB_D_UseV210()	87
4.2.2.23	USB_D_EnableSuperSpeed()	88
4.2.2.24	USB_D_SetWebUSBInfo()	89
4.2.2.25	USB_D_SetCheckAddress()	90

4.2.3	USB I/O functions	91
4.2.3.1	USB_D_Read()	91
4.2.3.2	USB_D_ReadOverlapped()	92
4.2.3.3	USB_D_Receive()	93
4.2.3.4	USB_D_ReceivePoll()	94
4.2.3.5	USB_D_ReadAsync()	95
4.2.3.6	USB_D_Write()	96
4.2.3.7	USB_D_WriteAsync()	97
4.2.3.8	USB_D_CancelIO()	98
4.2.3.9	USB_D_WaitForEndOfTransferEx()	99
4.2.3.10	USB_D_WaitForTXReady()	100
4.2.3.11	USB_D_GetNumBytesInBuffer()	101
4.2.3.12	USB_D_GetNumBytesRemToRead()	102
4.2.3.13	USB_D_GetNumBytesRemToWrite()	103
4.2.3.14	USB_D_StallEP()	104
4.2.4	USB Remote wakeup functions	105
4.2.4.1	USB_D_SetAllowRemoteWakeUp()	106
4.2.5	Data structures	107
4.2.5.1	USB_ADD_EP_INFO	107
4.2.5.2	USB_SETUP_PACKET	108
4.2.5.3	SEGGER_CACHE_CONFIG	109
4.2.5.4	USB_CHECK_ADDRESS_FUNC	110
4.2.5.5	USB_ASYNC_IO_CONTEXT	111
4.2.5.6	USB_WEBUSB_INFO	112
4.3	Timeout handling	113
5	Bulk communication	114
5.1	Generic bulk stack	115
5.2	Requirements for the Host (PC)	116
5.2.1	Windows	116
5.2.2	Linux	116
5.2.3	macOS	116
5.3	Example application	117
5.3.1	Running the example applications	118
5.3.2	Compiling the PC example application	119
5.3.2.1	Windows	119
5.3.2.2	Linux	119
5.3.2.3	macOS	119
5.4	Target API	120
5.4.1	Target interface function list	120
5.4.2	USB-Bulk functions	122
5.4.2.1	USB_D_BULK_Add()	122
5.4.2.2	USB_D_BULK_Add_Ex()	123
5.4.2.3	USB_D_BULK_AddAlternateInterface()	124
5.4.2.4	USB_D_BULK_SetMSDescInfo()	125
5.4.2.5	USB_D_BULK_CancelRead()	126
5.4.2.6	USB_D_BULK_CancelWrite()	127
5.4.2.7	USB_D_BULK_GetNumBytesInBuffer()	128
5.4.2.8	USB_D_BULK_GetNumBytesRemToRead()	129
5.4.2.9	USB_D_BULK_GetNumBytesRemToWrite()	130
5.4.2.10	USB_D_BULK_Read()	131
5.4.2.11	USB_D_BULK_ReadAsync()	132
5.4.2.12	USB_D_BULK_ReadOverlapped()	133
5.4.2.13	USB_D_BULK_Receive()	134
5.4.2.14	USB_D_BULK_ReceivePoll()	135
5.4.2.15	USB_D_BULK_SetContinuousReadMode()	136
5.4.2.16	USB_D_BULK_SetOnSetupRequest()	137
5.4.2.17	USB_D_BULK_SetOnRXEvent()	138
5.4.2.18	USB_D_BULK_SetOnTXEvent()	140

5.4.2.19	USBD_BULK_TxIsPending()	142
5.4.2.20	USBD_BULK_WaitForRX()	143
5.4.2.21	USBD_BULK_PollForRX()	144
5.4.2.22	USBD_BULK_WaitForTX()	145
5.4.2.23	USBD_BULK_PollForTX()	146
5.4.2.24	USBD_BULK_WaitForTXReady()	147
5.4.2.25	USBD_BULK_Write()	148
5.4.2.26	USBD_BULK_WriteAsync()	150
5.4.2.27	USBD_BULK_WriteEx()	151
5.4.3	Data structures	152
5.4.3.1	USB_BULK_INIT_DATA	152
5.4.3.2	USB_BULK_INIT_DATA_EX	153
5.5	Host API	154
5.5.1	Bulk Host API list	155
5.5.2	USB-Bulk basic functions	157
5.5.2.1	USBBULK_Init()	157
5.5.2.2	USBBULK_Exit()	158
5.5.2.3	USBBULK_AddAllowedDeviceItem()	159
5.5.2.4	USBBULK_GetNumAvailableDevices()	160
5.5.2.5	USBBULK_Open()	161
5.5.2.6	USBBULK_Close()	162
5.5.3	USB-Bulk direct input/output functions	163
5.5.3.1	USBBULK_Read()	163
5.5.3.2	USBBULK_ReadTimed()	164
5.5.3.3	USBBULK_Write()	165
5.5.3.4	USBBULK_WriteTimed()	166
5.5.3.5	USBBULK_CancelRead()	167
5.5.3.6	USBBULK_FlushRx()	168
5.5.4	USB-Bulk control functions	169
5.5.4.1	USBBULK_SetMode()	169
5.5.4.2	USBBULK_GetMode()	170
5.5.4.3	USBBULK_SetReadTimeout()	171
5.5.4.4	USBBULK_SetWriteTimeout()	172
5.5.4.5	USBBULK_ResetINPipe()	173
5.5.4.6	USBBULK_ResetOUTPipe()	174
5.5.4.7	USBBULK_ResetDevice()	175
5.5.5	USB-Bulk general GET functions	176
5.5.5.1	USBBULK_GetVersion()	176
5.5.5.2	USBBULK_GetDevInfo()	177
5.5.5.3	USBBULK_GetDevInfoByIdx()	178
5.5.5.4	USBBULK_GetUSBId()	179
5.5.5.5	USBBULK_GetProductName()	180
5.5.5.6	USBBULK_GetVendorName()	181
5.5.5.7	USBBULK_GetSN()	182
5.5.5.8	USBBULK_GetConfigDescriptor()	183
5.5.6	USB-Bulk data structures	184
5.5.6.1	USBBULK_DEV_INFO	184
6	Vendor Specific Class (VSC)	185
6.1	Vendor Specific Class	186
6.2	Requirements for the Host (PC)	187
6.2.1	Windows	187
6.2.2	Linux	187
6.2.3	macOS	187
6.3	Example application	188
6.3.1	Running the example applications	189
6.3.2	Compiling the PC example application	190
6.3.2.1	Windows	190
6.3.2.2	Linux	190

6.3.2.3	macOS	190
6.4	Target API	191
6.4.1	Target interface function list	191
6.4.2	USB-VSC functions	193
6.4.2.1	USB_D_VSC_Add()	193
6.4.2.2	USB_D_VSC_AddAlternateInterface()	194
6.4.2.3	USB_D_VSC_SetMSDescInfo()	195
6.4.2.4	USB_D_VSC_CancelIO()	196
6.4.2.5	USB_D_VSC_GetNumBytesInBuffer()	197
6.4.2.6	USB_D_VSC_GetNumBytesRemToRead()	198
6.4.2.7	USB_D_VSC_GetNumBytesRemToWrite()	199
6.4.2.8	USB_D_VSC_Read()	200
6.4.2.9	USB_D_VSC_ReadAsync()	202
6.4.2.10	USB_D_VSC_SetContinuousReadMode()	203
6.4.2.11	USB_D_VSC_SetOnSetupRequest()	204
6.4.2.12	USB_D_VSC_SetOnEPEvent()	205
6.4.2.13	USB_D_VSC_TxIsPending()	209
6.4.2.14	USB_D_VSC_WaitEP()	210
6.4.2.15	USB_D_VSC_PollEP()	211
6.4.2.16	USB_D_VSC_WaitForTXReady()	213
6.4.2.17	USB_D_VSC_Write()	214
6.4.2.18	USB_D_VSC_WriteAsync()	216
6.4.3	Data structures	217
6.4.3.1	USB_VSC_INIT_DATA	217
6.4.3.2	USB_VSC_MSOSDESC_INFO	218
6.4.3.3	USB_ON_ADD_FUNCTION_DESC	219
6.4.3.4	USB_VSC_ON_SET_INTERFACE	220
7	Mass Storage Device Class (MSD)	221
7.1	Overview	222
7.2	MSD Configuration	223
7.2.1	Initial configuration	223
7.2.2	Final configuration	223
7.2.3	MSD class specific configuration functions	223
7.2.4	Running the example application	223
7.2.4.1	MSD_Start_StorageRAM.c in detail	224
7.3	Target API	225
7.3.1	API functions	227
7.3.1.1	USB_D_MSD_Add()	227
7.3.1.2	USB_D_MSD_AddUnit()	228
7.3.1.3	USB_D_MSD_AddCDRom()	229
7.3.1.4	USB_D_MSD_SetPreventAllowRemovalHook()	230
7.3.1.5	USB_D_MSD_SetReadWriteHook()	231
7.3.1.6	USB_D_MSD_Task()	232
7.3.1.7	USB_D_MSD_Poll()	233
7.3.1.8	USB_D_MSD_SetStartStopUnitHook()	234
7.3.2	Extended API functions	235
7.3.2.1	USB_D_MSD_Connect()	235
7.3.2.2	USB_D_MSD_Disconnect()	236
7.3.2.3	USB_D_MSD_RequestDisconnect()	237
7.3.2.4	USB_D_MSD_RequestRefresh()	238
7.3.2.5	USB_D_MSD_UpdateWriteProtect()	239
7.3.2.6	USB_D_MSD_WaitForDisconnection()	240
7.3.3	Data structures	241
7.3.3.1	USB_MSD_INIT_DATA	241
7.3.3.2	USB_MSD_INFO	242
7.3.3.3	USB_MSD_INST_DATA	243
7.3.3.4	USB_MSD_LUN_INFO	244
7.3.3.5	PREVENT_ALLOW_REMOVAL_HOOK	245

9.2.5	Compile time configuration	290
9.3	Running the sample application	292
9.3.1	USB_MTP_Start.c in detail	292
9.4	Target API	294
9.4.1	API functions	295
9.4.1.1	USB_D_MTP_Add()	295
9.4.1.2	USB_D_MTP_AddStorage()	296
9.4.1.3	USB_D_MTP_Task()	297
9.4.1.4	USB_D_MTP_Poll()	298
9.4.1.5	USB_D_MTP_SendEvent()	299
9.4.2	Data structures	301
9.4.2.1	USB_MTP_FILE_INFO	301
9.4.2.2	USB_MTP_INIT_DATA	302
9.4.2.3	USB_MTP_INFO	303
9.4.2.4	USB_MTP_INST_DATA	304
9.4.2.5	USB_MTP_INST_DATA_DRIVER	305
9.4.2.6	USB_MTP_STORAGE_API	306
9.4.2.7	USB_MTP_STORAGE_INFO	308
9.4.3	Enums	309
9.4.3.1	USB_MTP_EVENT	309
9.5	MTP Storage Driver	311
9.5.1	General information	311
9.5.2	Interface function list	311
9.5.3	USB_MTP_STORAGE_API in detail	312
9.5.3.1	USB_MTP_STORAGE_INIT	312
9.5.3.2	USB_MTP_STORAGE_GET_INFO	313
9.5.3.3	USB_MTP_STORAGE_FIND_FIRST_FILE	314
9.5.3.4	USB_MTP_STORAGE_FIND_NEXT_FILE	315
9.5.3.5	USB_MTP_STORAGE_OPEN_FILE	316
9.5.3.6	USB_MTP_STORAGE_CREATE_FILE	317
9.5.3.7	USB_MTP_STORAGE_READ_FROM_FILE	318
9.5.3.8	USB_MTP_STORAGE_WRITE_TO_FILE	319
9.5.3.9	USB_MTP_STORAGE_CLOSE_FILE	320
9.5.3.10	USB_MTP_STORAGE_REMOVE_FILE	321
9.5.3.11	USB_MTP_STORAGE_CREATE_DIR	322
9.5.3.12	USB_MTP_STORAGE_REMOVE_DIR	323
9.5.3.13	USB_MTP_STORAGE_FORMAT	324
9.5.3.14	USB_MTP_STORAGE_RENAME_FILE	325
9.5.3.15	USB_MTP_STORAGE_DEINIT	326
9.5.3.16	USB_MTP_STORAGE_GET_FILE_ATTRIBUTES	327
9.5.3.17	USB_MTP_STORAGE_MODIFY_FILE_ATTRIBUTES	328
9.5.3.18	USB_MTP_STORAGE_GET_FILE_CREATION_TIME	329
9.5.3.19	USB_MTP_STORAGE_GET_FILELAST_WRITE_TIME	330
9.5.3.20	USB_MTP_STORAGE_GET_FILE_ID	331
9.5.3.21	USB_MTP_STORAGE_GET_FILE_SIZE	332
10	Communication Device Class (CDC)	333
10.1	Overview	334
10.1.1	Configuration	334
10.1.2	CDC-ACM issues on Windows 10	334
10.2	The example application	335
10.3	Installing the driver	336
10.3.1	The .inf file	340
10.3.2	Signing the package	341
10.3.3	Testing communication to the USB device	341
10.4	Target API	344
10.4.1	Interface function list	344
10.4.1.1	USB_D_CDC_Add()	346
10.4.1.2	USB_D_CDC_CancelRead()	347

10.4.1.3	USBDCancelWrite()	348
10.4.1.4	USBDCDCRead()	349
10.4.1.5	USBDCDCReadOverlapped()	350
10.4.1.6	USBDCDCReceive()	351
10.4.1.7	USBDCDCReceivePoll()	352
10.4.1.8	USBDCDCReadAsync()	353
10.4.1.9	USBDCDCSetOnBreak()	354
10.4.1.10	USBDCDCSetOnLineCoding()	355
10.4.1.11	USBDCDCSetOnControlLineState()	356
10.4.1.12	USBDCDCSetOnRXEvent()	357
10.4.1.13	USBDCDCSetOnTXEvent()	359
10.4.1.14	USBDCDCUpdateSerialState()	361
10.4.1.15	USBDCDCWrite()	362
10.4.1.16	USBDCDCWriteAsync()	363
10.4.1.17	USBDCDCWaitForRX()	364
10.4.1.18	USBDCDCPollForRX()	365
10.4.1.19	USBDCDCWaitForTX()	366
10.4.1.20	USBDCDCPollForTX()	367
10.4.1.21	USBDCDCWaitForTXReady()	368
10.4.1.22	USBDCDCWriteSerialState()	369
10.4.1.23	USBDCDCGetNumBytesRemToRead()	370
10.4.1.24	USBDCDCGetNumBytesRemToWrite()	371
10.4.1.25	USBDCDCGetNumBytesInBuffer()	372
10.4.2	Data structures	373
10.4.2.1	USB_CDC_INIT_DATA	373
10.4.2.2	USB_CDC_LINE_CODING	374
10.4.2.3	USB_CDC_SERIAL_STATE	375
10.4.2.4	USB_CDC_CONTROL_LINE_STATE	376
11	Human Interface Device Class (HID)	377
11.1	Overview	378
11.1.1	Further reading	378
11.1.2	Categories	378
11.1.2.1	True HIDs	378
11.1.2.2	Vendor specific HIDs	378
11.2	Background information	380
11.2.1	HID descriptors	380
11.2.1.1	HID descriptor	380
11.2.1.2	Report descriptor	380
11.2.1.3	Physical descriptor	381
11.3	Configuration	382
11.3.1	Initial configuration	382
11.3.2	Final configuration	382
11.4	Example application	383
11.4.1	USB_HID_Mouse.c	383
11.4.2	USB_HID_Echo1.c	383
11.4.2.1	Running the example	384
11.4.2.2	Compiling the PC example application	384
11.5	Target API	386
11.5.1	Target interface function list	387
11.5.2	HID Target API functions	388
11.5.2.1	USB_HID_Add()	388
11.5.2.2	USB_HID_GetNumBytesInBuffer()	389
11.5.2.3	USB_HID_GetNumBytesRemToRead()	390
11.5.2.4	USB_HID_GetNumBytesRemToWrite()	391
11.5.2.5	USB_HID_Read()	392
11.5.2.6	USB_HID_ReadOverlapped()	393
11.5.2.7	USB_HID_WaitForRX()	394
11.5.2.8	USB_HID_WaitForTX()	395

11.5.2.9	USBD_HID_Write()	396
11.5.2.10	USBD_HID_SetOnGetReportRequest()	397
11.5.2.11	USBD_HID_SetOnSetReportRequest()	398
11.5.2.12	USBD_HID_ReadReport()	399
11.5.3	Data structures	400
11.5.3.1	USB_HID_INIT_DATA	400
11.5.4	Type definitions	402
11.5.4.1	USB_HID_ON_GETREPORT_REQUEST_FUNC	402
11.5.4.2	USB_HID_ON_SETREPORT_REQUEST_FUNC	403
11.6	Host API	404
11.6.1	Host API function list	405
11.6.2	HID Host API functions	406
11.6.2.1	USBHID_Close()	406
11.6.2.2	USBHID_Open()	407
11.6.2.3	USBHID_Init()	408
11.6.2.4	USBHID_Exit()	409
11.6.2.5	USBHID_Read()	410
11.6.2.6	USBHID_Write()	411
11.6.2.7	USBHID_GetNumAvailableDevices()	412
11.6.2.8	USBHID_GetProductName()	413
11.6.2.9	USBHID_GetInputReportSize()	414
11.6.2.10	USBHID_GetOutputReportSize()	415
11.6.2.11	USBHID_GetProductId()	416
11.6.2.12	USBHID_GetVendorId()	417
11.6.2.13	USBHID_RefreshList()	418
11.6.2.14	USBHID_SetVendorPage()	419
12	Printer Class	420
12.1	Overview	421
12.1.1	Configuration	421
12.2	The example application	422
12.3	Target API	424
12.3.1	Interface function list	424
12.3.2	API functions	425
12.3.2.1	USB_PRINTER_Init()	425
12.3.2.2	USB_PRINTER_Task()	426
12.3.2.3	USB_PRINTER_TaskEx()	427
12.3.2.4	USB_PRINTER_ConfigIRQProcessing()	428
12.3.2.5	USB_PRINTER_Read()	429
12.3.2.6	USB_PRINTER_ReadTimed()	430
12.3.2.7	USB_PRINTER_Receive()	431
12.3.2.8	USB_PRINTER_ReceiveTimed()	432
12.3.2.9	USB_PRINTER_Write()	433
12.3.2.10	USB_PRINTER_WriteTimed()	434
12.3.2.11	USB_PRINTER_SetOnVendorRequest()	435
12.3.2.12	USB_PRINTER_SetClass()	436
12.3.2.13	USB_PRINTER_API	437
12.4	Printer API	438
12.4.1	General information	438
12.4.2	USB_PRINTER_API in detail	439
12.4.2.1	USB_PRINTER_GET_DEVICE_ID_STRING	439
12.4.2.2	USB_PRINTER_ON_DATA_RECEIVED	440
12.4.2.3	USB_PRINTER_GET_HAS_NO_ERROR	441
12.4.2.4	USB_PRINTER_GET_IS_SELECTED	442
12.4.2.5	USB_PRINTER_GET_IS_PAPER_EMPTY	443
12.4.2.6	USB_PRINTER_ON_RESET	444
13	IP-over-USB (IP)	445
13.1	Overview	446

13.2	Using only RNDIS or CDC-ECM	447
13.2.1	Working with emUSB-Device-IP	447
13.3	Configuration	449
13.3.1	Initial Configuration	449
13.3.2	Final configuration	449
13.3.3	Class specific configuration	449
13.4	Running the sample application	450
13.4.1	IP_Config_IP_over_USB.c in detail	450
13.5	emUSB-Device-IP + emNet as a "USB Webserver"	453
13.6	Target API	454
13.6.1	API functions	455
13.6.1.1	USB_IP_Add()	455
13.6.1.2	USB_IP_Task()	456
13.6.2	Data structures	457
13.6.2.1	USB_IP_INIT_DATA	457
14	Remote NDIS (RNDIS)	458
14.1	Overview	459
14.1.1	Working with RNDIS	459
14.1.2	Additional information	459
14.2	Configuration	460
14.2.1	Initial Configuration	460
14.2.2	Final configuration	460
14.2.3	Class specific configuration	460
14.3	Running the sample application	461
14.3.1	IP_Config_RNDIS.c in detail	461
14.4	RNDIS + emNet as a "USB Webserver"	463
14.5	Target API	464
14.5.1	API functions	465
14.5.1.1	USB_RNDIS_Add()	465
14.5.1.2	USB_RNDIS_Task()	466
14.5.1.3	USB_RNDIS_SetDeviceInfo()	467
14.5.2	Data structures	468
14.5.2.1	USB_RNDIS_INIT_DATA	468
14.5.2.2	USB_RNDIS_DEVICE_INFO	469
14.5.3	Driver interface	470
14.5.3.1	USB_IP_NI_DRIVER_API	470
14.5.3.2	USB_IP_NI_DRIVER_DATA	471
14.6	RNDIS IP Driver	472
14.6.1	General information	472
14.6.2	Interface function list	472
14.6.3	USB_IP_NI_DRIVER_API in detail	473
14.6.3.1	USB_IP_NI_INIT	473
14.6.3.2	USB_IP_NI_GET_PACKET_BUFFER	474
14.6.3.3	USB_IP_NI_WRITE_PACKET	475
14.6.3.4	USB_IP_NI_SET_PACKET_FILTER	476
14.6.3.5	USB_IP_NI_GET_LINK_STATUS	477
14.6.3.6	USB_IP_NI_GET_LINK_SPEED	478
14.6.3.7	USB_IP_NI_GET_HWADDR	479
14.6.3.8	USB_IP_NI_GET_STATS	480
14.6.3.9	USB_IP_NI_GET_MTU	481
14.6.3.10	USB_IP_NI_RESET	482
14.6.3.11	USB_IP_NI_SET_WRITE_PACKET_FUNC	483
14.6.3.12	USB_IP_NI_SET_REPORT_LINKSTATE_FUNC	484
15	CDC-ECM	485
15.1	Overview	486
15.1.1	Working with CDC-ECM	486
15.1.2	Additional information	487

15.2	Configuration	488
15.2.1	Initial configuration	488
15.2.2	Final configuration	488
15.3	Running the sample application	489
15.3.1	IP_Config_ECM.c in detail	489
15.4	Target API	491
15.4.1	API functions	492
15.4.1.1	USB_D_ECM_Add()	492
15.4.1.2	USB_D_ECM_Task()	493
15.4.2	Data structures	494
15.4.2.1	USB_ECM_INIT_DATA	494
15.4.3	Driver interface	495
15.4.3.1	USB_IP_NI_DRIVER_API	495
15.4.3.2	USB_IP_NI_DRIVER_DATA	496
15.5	CDC-ECM IP Driver	497
15.5.1	General information	497
15.5.2	Interface function list	497
15.5.3	USB_IP_NI_DRIVER_API in detail	498
15.5.3.1	USB_IP_NI_INIT	498
15.5.3.2	USB_IP_NI_GET_PACKET_BUFFER	499
15.5.3.3	USB_IP_NI_WRITE_PACKET	500
15.5.3.4	USB_IP_NI_SET_PACKET_FILTER	501
15.5.3.5	USB_IP_NI_GET_LINK_STATUS	502
15.5.3.6	USB_IP_NI_GET_LINK_SPEED	503
15.5.3.7	USB_IP_NI_GET_HWADDR	504
15.5.3.8	USB_IP_NI_GET_STATS	505
15.5.3.9	USB_IP_NI_GET_MTU	506
15.5.3.10	USB_IP_NI_RESET	507
15.5.3.11	USB_IP_NI_SET_WRITE_PACKET_FUNC	508
15.5.3.12	USB_IP_NI_SET_REPORT_LINKSTATE_FUNC	509
16	CDC-NCM	510
16.1	Overview	511
16.1.1	Working with CDC-NCM	511
16.1.2	Additional information	511
16.2	Configuration	512
16.2.1	Initial configuration	512
16.2.2	Final configuration	512
16.3	Running the sample application	513
16.3.1	IP_Config_NCM.c in detail	513
16.4	Target API	515
16.4.1	API functions	516
16.4.1.1	USB_D_NCM_Add()	516
16.4.2	Data structures	517
16.4.2.1	USB_NCM_INIT_DATA	517
16.4.3	Driver interface	518
16.4.3.1	USB_IP_NI_DRIVER_API	518
16.4.3.2	USB_IP_NI_DRIVER_DATA	519
16.5	CDC-NCM IP Driver	520
16.5.1	General information	520
16.5.2	Interface function list	520
16.5.3	USB_IP_NI_DRIVER_API in detail	521
16.5.3.1	USB_IP_NI_INIT	521
16.5.3.2	USB_IP_NI_GET_PACKET_BUFFER	522
16.5.3.3	USB_IP_NI_WRITE_PACKET	523
16.5.3.4	USB_IP_NI_SET_PACKET_FILTER	524
16.5.3.5	USB_IP_NI_GET_LINK_STATUS	525
16.5.3.6	USB_IP_NI_GET_LINK_SPEED	526
16.5.3.7	USB_IP_NI_GET_HWADDR	527

16.5.3.8	USB_IP_NI_GET_STATS	528
16.5.3.9	USB_IP_NI_GET_MTU	529
16.5.3.10	USB_IP_NI_RESET	530
16.5.3.11	USB_IP_NI_SET_WRITE_PACKET_FUNC	531
16.5.3.12	USB_IP_NI_SET_REPORT_LINKSTATE_FUNC	532
17	Audio	533
17.1	Overview	534
17.2	Introduction	535
17.3	Configuration	536
17.3.1	Initial configuration	536
17.3.2	Final configuration	536
17.3.3	Using the microphone interface	536
17.3.4	Using the speaker interface	537
17.4	Target API	538
17.4.1	API functions	539
17.4.1.1	USBBD_AUDIO_Add()	539
17.4.1.2	USBBD_AUDIO_Read_Task()	540
17.4.1.3	USBBD_AUDIO_Write_Task()	541
17.4.1.4	USBBD_AUDIO_Start_Play()	542
17.4.1.5	USBBD_AUDIO_Stop_Play()	543
17.4.1.6	USBBD_AUDIO_Start_Listen()	544
17.4.1.7	USBBD_AUDIO_Stop_Listen()	545
17.4.1.8	USBBD_AUDIO_Set_Timeouts()	546
17.4.2	Data structures	547
17.4.2.1	USBBD_AUDIO_INIT_DATA	547
17.4.2.2	USBBD_AUDIO_IF_CONF	548
17.4.2.3	USBBD_AUDIO_FORMAT	550
17.4.2.4	USBBD_AUDIO_UNITS	551
17.4.3	Function definitions	552
17.4.3.1	USBBD_AUDIO_TX_FUNC	552
17.4.3.2	USBBD_AUDIO_RX_FUNC	553
17.4.3.3	USBBD_AUDIO_CONTROL_FUNC	555
18	USB Video device Class (UVC)	558
18.1	Overview	559
18.2	Configuration	560
18.2.1	Initial configuration	560
18.2.1.1	Uncompressed video format	560
18.2.2	Final configuration	560
18.3	Target API	561
18.3.1	API functions	562
18.3.1.1	USBBD_UVC_Add()	562
18.3.1.2	USBBD_UVC_Write()	563
18.3.1.3	USBBD_UVC_WriteEx()	565
18.3.1.4	USBBD_UVC_SetOnResolutionChange()	566
18.3.2	Data structures	567
18.3.2.1	USBBD_UVC_INIT_DATA	567
18.3.2.2	USBBD_UVC_BUFFER	568
18.3.2.3	USBBD_UVC_DATA_BUFFER	569
18.3.2.4	USBBD_UVC_RESOLUTION	570
18.3.3	Function prototypes	571
18.3.3.1	USB_UVC_ON_RESOLUTION_CHANGE	571
19	Device Firmware Upgrade (DFU)	572
19.1	Overview	573
19.1.1	Using DFU on Windows	573
19.2	Configuration	574

19.2.1	Dual configuration mode	574
19.2.2	Single configuration	574
19.3	Target API	575
19.3.1	API functions	576
19.3.1.1	USB_DFU_Add()	576
19.3.1.2	USB_DFU_Add_RunTime()	577
19.3.1.3	USB_DFU_AddAlternateInterface()	578
19.3.1.4	USB_DFU_SetMSDescInfo()	579
19.3.1.5	USB_DFU_SetPollTimeout()	580
19.3.1.6	USB_DFU_Ack()	581
19.3.1.7	USB_DFU_SetError()	582
19.3.1.8	USB_DFU_ManifestComplt()	583
19.3.1.9	USB_DFU_GetStatusReqCnt()	584
19.3.1.10	USB_DFU_GetAlternateSetting()	585
19.3.2	Data structures	586
19.3.2.1	USB_DFU_INIT_DATA	586
19.3.3	Function prototypes	587
19.3.3.1	USB_DFU_DETACH_REQUEST	587
19.3.3.2	USB_DFU_DOWNLOAD	588
19.3.3.3	USB_DFU_UPLOAD	589
20	Musical Instrument Digital Interface (MIDI)	590
20.1	Overview	591
20.2	Introduction	592
20.3	Configuration	594
20.3.1	Initial configuration	594
20.3.2	Final configuration	594
20.3.3	Testing MIDI on different operating systems	594
20.4	Target API	595
20.4.1	API functions	596
20.4.1.1	USB_MIDI_Init()	596
20.4.1.2	USB_MIDI_Add()	597
20.4.1.3	USB_MIDI_ReceivePackets()	598
20.4.1.4	USB_MIDI_GetNumPacketsInBuffer()	599
20.4.1.5	USB_MIDI_ConvertPackets()	600
20.4.1.6	USB_MIDI_WritePackets()	601
20.4.1.7	USB_MIDI_WriteStream()	602
20.4.2	Data structures	603
20.4.2.1	USB_MIDI_INIT_DATA	603
20.4.2.2	USB_MIDI_JACK	604
20.4.2.3	USB_MIDI_PACKET	605
21	Smard Card Device Class (CCID)	606
21.1	Overview	607
21.2	Target API	608
21.2.1	API functions	609
21.2.1.1	USB_CCID_Init()	609
21.2.1.2	USB_CCID_Add()	610
21.2.1.3	USB_CCID_ReceiveCmd()	611
21.2.1.4	USB_CCID_SendStatus()	612
21.2.1.5	USB_CCID_SendDataBlock()	613
21.2.1.6	USB_CCID_SendEscape()	614
21.2.1.7	USB_CCID_SendParameters()	615
21.2.1.8	USB_CCID_SendDataRateAndClockFrequency()	616
21.2.1.9	USB_CCID_NotifySlotState()	617
21.2.1.10	USB_CCID_NotifyHwError()	618
21.2.2	Data structures	619
21.2.2.1	USB_CCID_INIT_DATA	619
21.2.2.2	USB_CCID_PROPERTIES	620

21.2.2.3	USB_CCID_CMD	622
21.2.2.4	USB_CCID_PROTOCOL_DATA_T0	624
21.2.2.5	USB_CCID_PROTOCOL_DATA_T1	625
21.2.3	Function prototypes	626
21.2.3.1	USBD_CCID_ABORT_CB	626
22	Combining USB components (Multi-Interface)	627
22.1	Overview	628
22.1.1	Single interface device classes	629
22.1.2	Multiple interface device classes	629
22.1.3	IAD class	629
22.2	Configuration	631
22.3	How to combine	632
22.4	emUSB-Device component specific modification	636
22.4.1	CDC component	636
22.4.1.1	Device side	636
22.4.1.2	Host side	636
22.5	MSD and MTP combination feature	638
22.5.1	Configuration	638
22.5.2	Limitations	639
22.5.3	Target API	640
22.5.3.1	USBD_MSD_MTP_Add()	641
22.5.3.2	USBD_MSD_MTP_GetMode()	642
22.5.3.3	USBD_MSD_MTP_Task()	643
22.6	Issues on Windows 7	644
22.6.1	Detailed description	644
23	Target OS Interface	645
23.1	General information	646
23.1.1	Operating system support supplied with this release	646
23.2	Interface function list	647
23.2.1	USB_OS_DeInit()	648
23.2.2	USB_OS_Delay()	649
23.2.3	USB_OS_DecRI()	650
23.2.4	USB_OS_GetTickCnt()	651
23.2.5	USB_OS_IncDI()	652
23.2.6	USB_OS_Init()	653
23.2.7	USB_OS_Signal()	654
23.2.8	USB_OS_Wait()	655
23.2.9	USB_OS_WaitTimed()	656
24	Target USB Driver	657
24.1	General information	658
24.1.1	Available USB drivers	658
24.2	Adding a driver to emUSB-Device	659
24.2.1	USBD_X_Config()	659
24.2.2	USBD_X_DisableInterrupt()	661
24.2.3	USBD_X_EnableInterrupt()	662
24.3	Device driver specifics	663
24.3.1	LPC54/55xxx full-speed driver	664
24.3.2	LPC54/55xxx high-speed driver	664
24.3.3	EHCI driver	665
24.3.4	nRF52xxx, nRF53xx driver	666
24.3.5	Synopsys DWC2 driver (slave mode)	667
24.3.6	Synopsys DWC2 driver (DMA mode)	667
24.3.7	XHCI driver	669
24.3.8	Renesas RX driver	670
24.3.9	AT91RM9200 driver	671

25	Support	672
25.1	Contacting support	673
26	Profiling with SystemView	674
26.1	Profiling overview	675
26.2	Additional files for profiling	676
26.2.1	Additional files on target side	676
26.2.2	Additional files on PC side	676
26.3	Enable profiling	677
26.4	Recording and analyzing profiling information	678
27	Debugging	679
27.1	Message output	680
27.2	API functions	681
27.2.1	USBD_AddLogFilter()	682
27.2.2	USBD_AddWarnFilter()	683
27.2.3	USBD_SetLogFilter()	684
27.2.4	USBD_SetWarnFilter()	685
27.2.5	USB_PANIC	686
27.2.6	USB_X_Log()	687
27.2.7	USB_X_Warn()	688
27.2.8	USB_OS_Panic()	689
27.3	Message types	690
28	Certification	692
28.1	What is the Windows Hardware Certification and why do I need it?	693
28.2	Certification offer	693
28.3	Vendor and Product ID	693
28.4	Certification without SEGGER Microcontroller	694
29	Performance & resource usage	695
29.1	Memory footprint	696
29.2	Performance	698
30	FAQ	699

Chapter 1

Introduction

This chapter will give a short introduction to emUSB-Device, including the supported USB classes and components. Host and target requirements are covered as well.

1.1 Overview

This guide describes how to install, configure and use emUSB-Device. It also explains the internal structure of emUSB-Device.

emUSB-Device has been designed to work on any embedded system with a 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/s) devices is approximately 1 MB/s. This data rate can indeed be achieved on fast systems, such as Cortex-M devices running at 48 MHz and above.

USB 2.0 high-speed mode (480 MBit/s) is also fully supported and is automatically handled. Using USB high-speed mode with an Cortex-M or faster could achieve values of approx. 44 MByte/s.

The USB standard defines four types of communication: Control, isochronous, interrupt, and bulk. Experience shows that for most embedded devices the bulk mode is the communication mode of choice because applications can utilize the full bandwidth of the Universal Serial Bus.

1.2 emUSB-Device features

Key features of emUSB-Device are:

- High performance
- 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-Device components

emUSB-Device consists of three layers: A driver for hardware access, the emUSB-Device 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-Device consists of a driver that fits to the used hardware, the emUSB-Device core and at least one of the USB class drivers.

Component	Description
USB protocol layer	
Bulk	emUSB-Device bulk component.
MSD	emUSB-Device Mass Storage Device class component.
IP-over-USB	emUSB-Device IP-over-USB component.
VirtualMSD	emUSB-Device VirtualMSD Component
CDC-ACM	emUSB-Device Communication Device Class component.
HID	emUSB-Device Human Interface Device Class component.
MTP	emUSB-Device Media Transfer Protocol component.
Printer	emUSB-Device Printer Class component.
RNDIS	emUSB-Device RNDIS component.
CDC-ECM	emUSB-Device CDC Ethernet Control Model component.
CDC-NCM	emUSB-Device CDC Network Control Model component.
UVC	emUSB-Device USB video class.
Audio	emUSB-Device USB audio class.
DFU	emUSB-Device Device Firmware Upgrade class.
MIDI	emUSB-Device Musical Instrument Digital Interface class.
Core layer	
emUSB-Device-Core	The emUSB-Device core is the intrinsic USB stack.
Hardware layer	
Driver	USB controller driver.

1.3.1 emUSB-Device-Bulk

emUSB-Device-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). This bidirectional channel, with built-in flow control, 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.

1.3.2 emUSB-Device-MSD

1.3.2.1 Purpose of emUSB-Device-MSD

Access the target device like an ordinary disk drive

emUSB-Device-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-Device-MSD, videos or photos taken by this camera can be conveniently accessed with the file system explorer of the used operating system when the camera is connected to the computer.

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-Device-MSD features

Key features of emUSB-Device-MSD are:

- Can be used with RAM, parallel flash, serial flash or mechanical drives
- Support for full-speed (12 Mbit/s) and high-speed (480 Mbit/s) 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-Device-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-Device 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.

Provide file system code on the target if necessary

There are basically two types of MSD devices, one is where the devices does not need to access the storage (e.g. USB stick, external HDD). The other type is where the device needs to write data onto the storage medium before it is accessed from a PC (e.g. data logger) or read data from it after it has been written onto the storage medium by a PC (e.g. a mp3 player or a device which reads configuration files from the storage). If you are using emUSB-Device-MSD you are most likely writing software for the former device type. emUSB-Device-MSD does not offer file-level access to the storage medium, you need a file system to access the storage. complex and time-consuming task and increases 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 is optimized for minimum memory consumption in RAM and ROM, high-speed and versatility. It is written in ANSI C and runs on any CPU and on any media. Refer to <https://www.segger.com/emfile> for more information about *emFile*.

1.3.3 emUSB-Device IP-over-USB

emUSB-Device IP-over-USB allows to run any IP-based protocol over USB. This component combines the advantages of RNDIS and CDC-ECM and allows plug-and-play on any major host operating system. Using the IP-over-USB technology in combination with a built in web server, the device can easily be accessed from any host (Windows, Linux, Mac) by simply typing the device name into the web browser.

1.3.3.1 Typical applications

Typical applications are:

- Headphones
- Printer
- Data logger
- Ethernet2USB adapter

1.3.4 emUSB-Device-VirtualMSD

The emUSB-Device-VirtualMSD component allows to easily stream files to and from USB devices. Once the USB device is connected to the host, files can be read or written to the application without the need for dedicated storage memory.

1.3.4.1 Typical applications

Typical applications are:

- Updating firmware (e.g. Handheld Terminal)
- Updating configuration files

1.3.5 emUSB-Device-CDC

emUSB-Device-CDC converts the target device into a serial communication device. A target device running emUSB-Device-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.

1.3.5.1 Typical applications

Typical applications are:

- Modem
- Telephone system
- Fax machine

1.3.6 emUSB-Device-HID

The Human Interface Device class (HID) is an abstract USB class protocol defined by the USB Implementers Forum. This protocol was defined for handling devices that humans use 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.

1.3.6.1 Typical applications

Typical applications are:

- Keyboard
- Mouse and similar pointing devices
- Gamepad
- Front-panel controls - for example, switches and buttons

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

1.3.7 emUSB-Device-MTP

The Media Transfer Protocol (MTP) is a USB class protocol which can be used to transfer files to and from storage devices. MTP is an alternative to MSD as it operates on a file level rather than on a storage sector level. The advantage of MTP is the ability to access the storage medium from the host PC and from the device at the same time. Because MTP works at the file level this also eliminates the risk of damaging the file system when the communication to the host has been canceled unexpectedly (e.g. the cable was removed). MTP is supported by most operating systems without the need to install third-party drivers.

1.3.7.1 Typical applications

Typical applications are:

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

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

1.3.8 emUSB-Device-Printer

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

1.3.8.1 Typical applications

Typical applications are:

- Laser/Inkjet printer
- CNC machine

1.3.9 emUSB-Device-RNDIS

emUSB-Device-RNDIS allows to create a virtual Ethernet adapter through which the host PC can communicate with the device using the Internet protocol suite (TCP, UDP, FTP, HTTP, Telnet). This allows the creation of USB based devices which can host a webserver or act as a telnet terminal or a FTP server. emUSB-Device-RNDIS offer a unique customer experience and allows to save development and hardware cost by e.g. using a website as a user interface instead of creating an application for every major OS and by eliminating the Ethernet hardware components from your device.

1.3.9.1 Typical applications

Typical applications are:

- USB-Webserver
- USB-Terminal (e.g. Telnet)
- USB-FTP-Server

1.3.10 emUSB-Device-CDC-ECM

emUSB-Device-CDC-ECM allows to create a virtual Ethernet adapter through which the host PC can communicate with the device using the Internet protocol suite (TCP, UDP, FTP, HTTP, Telnet). This allows the creation of USB based devices which can host a webserver or act as a telnet terminal or a FTP server. emUSB-Device-CDC-ECM offer a unique customer experience and allows to save development and hardware cost by e.g. using a website as a user interface instead of creating an application for every major OS and by eliminating the Ethernet hardware components from your device.

1.3.10.1 Typical applications

Typical applications are:

- USB-Webserver
- USB-Terminal (e.g. Telnet)
- USB-FTP-Server

1.4 Requirements

1.4.1 Target system

Hardware

The target system must have a USB controller. The memory requirements can be found in the chapter *Performance & resource usage* on page 695. In order to have the control when the device is enumerated by the host, a switchable attach is necessary. This is a switchable pull-up connected to the D+ Line of USB.

Software

emUSB-Device is optimized to be used with *embOS* but works with any other supported RTOS or without an RTOS in a superloop. For information regarding the OS integration refer to the chapter *Target OS Interface* on page 645.

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.

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-Device root directory:

Directory	Contents
Application	Contains the application programs. Depending on which stack is used, several files are available for each stack. Detailed information can be found in the corresponding chapter.
BSP	Contains example hardware-specific configurations for different eval boards.
Config	Contains configuration files (USB_Conf.h, USB_ConfigIO.c).
Doc	Contains the emUSB-Device documentation.
Inc	Contains include files.
Sample	Contains operating systems dependent files which allows to run emUSB-Device with different RTOS's.
SEGGER	Contains generic routines from SEGGER (e.g. memcpy).
USB	Contains the emUSB-Device source code.
Windows	Contains Windows specific applications which can be used in conjunction with the device application samples.

1.6 Multithreading

The emUSB target API is not generally thread safe. But it is allowed to handle different endpoints in different tasks in parallel. Examples are:

- A task that performs all reads of data from the host while another task sends data to the host.
- Operating on different interfaces (e.g. a BULK and a CDC interface) in independent tasks.

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 a bus architecture for connecting multiple 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 kB/s to hundreds of MB/s.
- 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, 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 Mbit/s for low-speed and 12 Mbit/s for full-speed devices. The maximum cable length between host and device is five meters. Up to 500 mA of electric current may be distributed to low power devices.

USB 2.0 (April 2000)

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

USB 3.0 (November 2008)

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

USB 3.1 (July 2013)

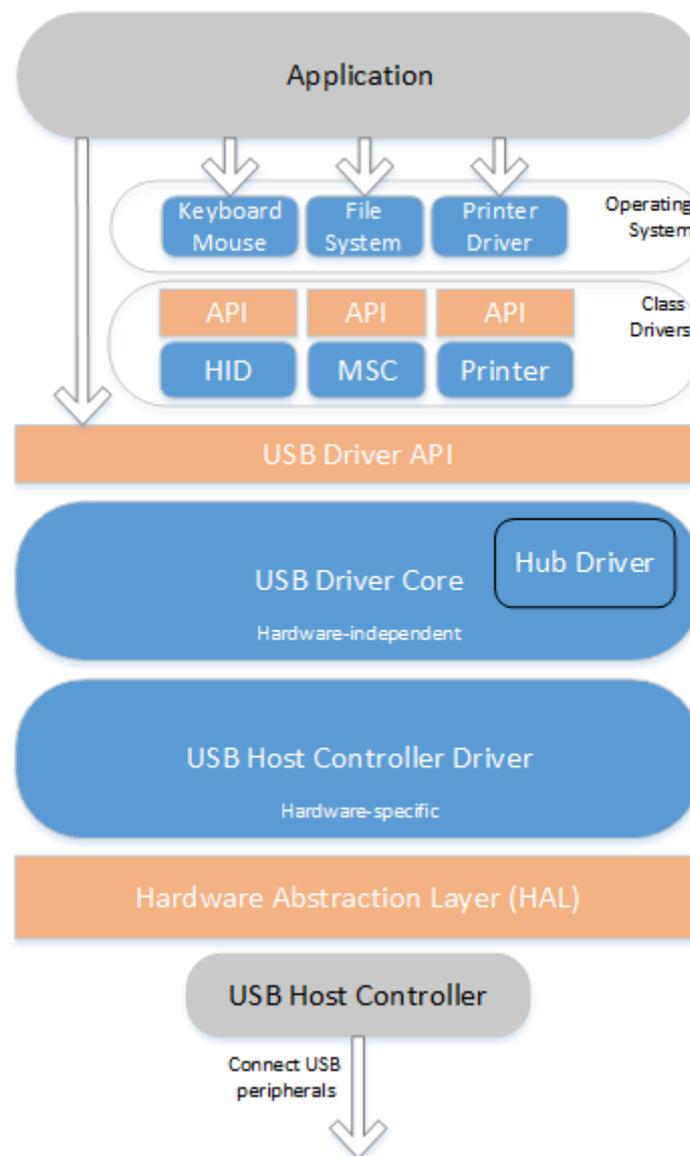
As all previous USB standards, USB 3.1 is fully forward and backward compatible. The new specification replaces the 3.0 standard and introduces new transfer speeds of up to 10 Gbit/s.

2.1.3 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.

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

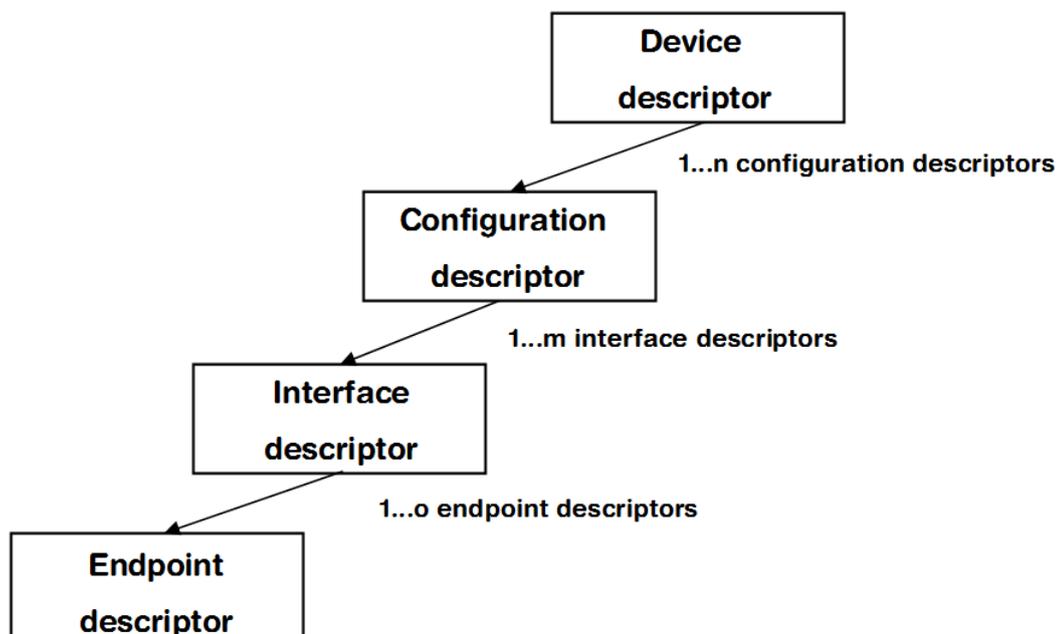
Two types of devices exist: hubs and functions. Hubs provide for additional USB attachment points. Functions provide capabilities to the host and are able to transmit or receive data or control information over the USB bus. Every peripheral USB device represents at least one function but may implement 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 describes its capabilities and resource requirements. A USB device must be configured by the host before its functions can be used. 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-Device-MSD, the embedded device will appear as a USB mass storage device, and the host OS provides the driver out of the box. In general, there is no need to develop a custom driver to communicate with target devices that use one of the USB class protocols.

Descriptors

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



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

2.1.4 Transfer Types

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

Control transfers

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

Interrupt transfers

Typically used by devices that need guaranteed quick responses (fixed 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.

Isochronous transfers

Typically used for applications which need guaranteed speed. Isochronous transfer offers a guaranteed bandwidth but with possible data loss. A typical use is for audio data which requires a constant data rate. Unlike bulk, control or interrupt transfers isochronous transfers do not receive an "ACK" from the other side, therefore the sender does not know whether the data was received by the other side correctly. For applications where constant data rate is more important than data integrity (audio, video) the potential data loss does not pose an issue.

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

The Product and Vendor IDs are necessary to identify the USB device. The Product ID describes a specific device type and does not need to be unique between different devices of the same type. USB host systems like Windows use the Product ID/Vendor ID combination to identify which drivers are needed.

For example: all our J-Link devices have the Vendor ID 0x1366 and Product ID 0x0105.

A Vendor and Product ID 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. Using the sample Vendor ID (0x8765) or the SEGGER Vendor ID in a finished product is not allowed.

Possible options to obtain a Vendor ID or Product ID are described in the chapter *Vendor and Product ID* on page 693.

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.

2.3 USB hardware analyzers

A variety of USB hardware analyzers are on the market with different capabilities. If you are developing an application using emUSB-Device it should not be necessary to have a USB analyzer, but we still recommend you do.

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-Device 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

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

Procedure to follow

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

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

3.1.1 Take a running project

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

3.1.2 Add emUSB-Device files

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

Add RTOS layer

Additionally add the RTOS interface layer to your project. Choose a file from the folder `Sample/USB/OS` that matches your RTOS. For embOS use `USB_OS_embOSv5.c`. There is also a file `USB_OS_None.c` containing a layer to be used for superloop applications without an RTOS.

Configuring the include path

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

- `Config`
- `Inc`
- `SEGGER`
- `USB`

3.1.3 Configuring debugging output

While developing and testing emUSB-Device, we recommend to use the DEBUG configuration of emUSB-Device. This is enabled by setting the preprocessor symbol `DEBUG` to 1 (or `USB_DEBUG_LEVEL` to 2). The DEBUG configuration contains many additional run-time checks and generate debug output messages which are very useful to identify problems that may occur during development. In case of a fatal problem (e.g. an invalid configuration) the program will end up in the function `USB_OS_Panic()` with a appropriate error message that describes the cause of the problem. Once the application is running correctly, `DEBUG` can be set to 0.

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

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

3.1.4 Add hardware dependent configuration

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

- Select an appropriate driver for the USB controller.
- Configure I/O pins of the MCU for USB.
- Configure PLL and clock divider necessary for USB operation.
- Install an interrupt service routine for USB.

Details can be found in *Target USB Driver* on page 657.

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

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

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

Add BSP file

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

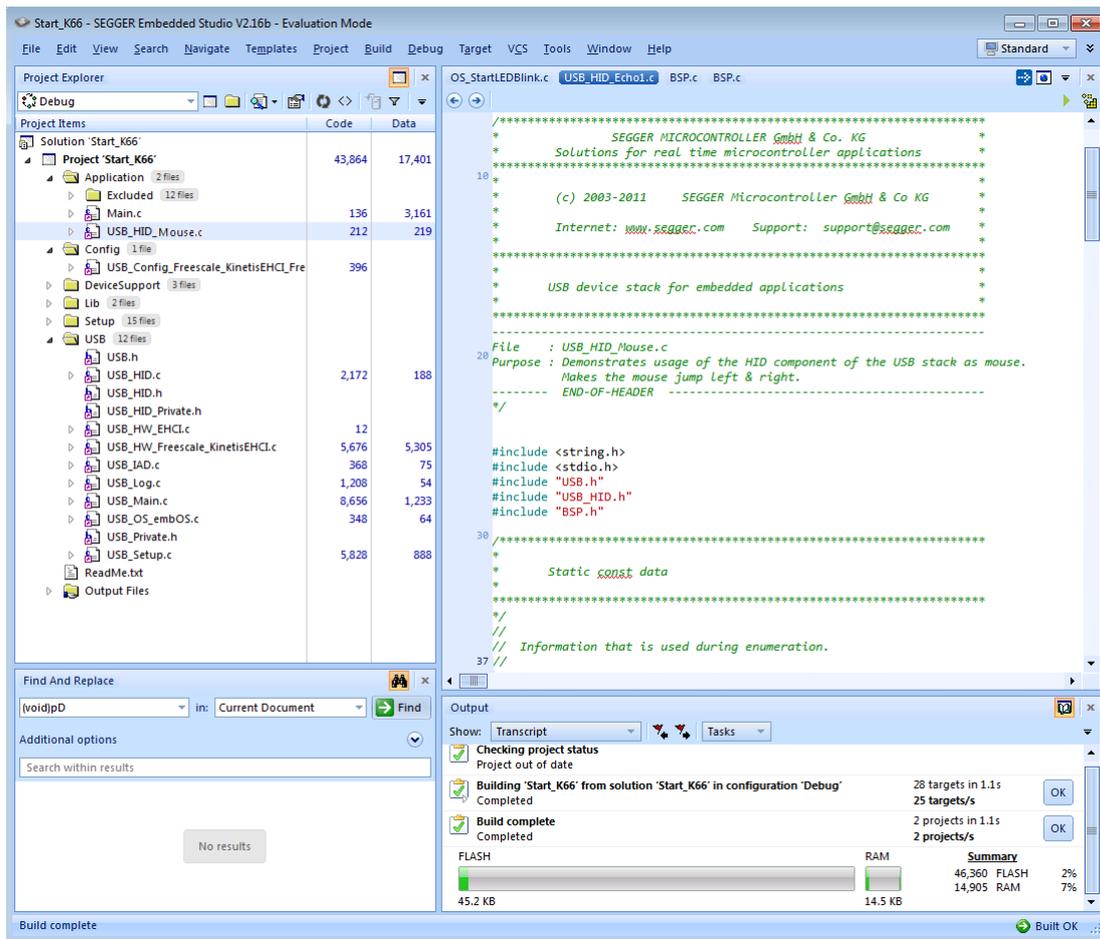
Add the appropriate `BSP_USB.c` file to your project. If there is no BSP file for your target hardware, take a file for a similar hardware and modify it if necessary.

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

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

3.1.5 Prepare and run the application

Choose a sample application from the folder `Application` and add it to your project. For example, add `USB_HID_Mouse.c` as your application to your project. Compile and run the application on the target hardware. After connecting the USB cable to the target device, the mouse pointer should hop from left to right.



3.2 Updating emUSB-Device

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

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

- USB
- Inc
- SEGGER
- Doc
- Sample/USB/OS

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

- `USB_Conf.h`
- `USB_ConfigIO.c`
- `BSP_USB.c`
- `USB_Config_<TargetName>.c`

3.3 emUSB-Device Configuration

An application using emUSB-Device must contain a `USB_DEVICE_INFO` structure containing the device identification information.

3.3.1 USB_DEVICE_INFO

Description

Device information that must be provided by the application via the function `USBD_SetDeviceInfo()` before the USB stack is started using `USBD_Start()`. Is used during enumeration of the device by the host.

Type definition

```
typedef struct {
    U16      VendorId;
    U16      ProductId;
    const char * sVendorName;
    const char * sProductName;
    const char * sSerialNumber;
} USB_DEVICE_INFO;
```

Structure members

Member	Description
<code>VendorId</code>	Vendor ID. Uniquely identifies the vendor on a USB device.
<code>ProductId</code>	Product ID. Uniquely identifies all USB devices of a vendor.
<code>sVendorName</code>	Vendor name. ASCII string of up to 126 characters.
<code>sProductName</code>	Description of the USB device. ASCII string of up to 126 characters.
<code>sSerialNumber</code>	Serial number of the USB device (ASCII string). May be <code>NULL</code> if no serial number should be provided.

Additional information

The Product ID in combination with the Vendor ID creates a worldwide unique identifier for the product model. The Vendor ID is assigned by the USB Implementers Forum (<https://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 to market without having been assigned your own Vendor ID. For emUSB-Device-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 340. Otherwise, the Windows host will be unable to locate the driver.

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

Note

The max string length cannot be more than 126 ANSI characters.

Note for MSD: In order to confirm to the USB bootability specification, the minimum string length of the serial number must be 12 characters where each character is a hexadecimal digit ('0' though '9' or 'A' through 'F').

Example

```
static const USB_DEVICE_INFO _DeviceInfo = {
    0x8765,          // VendorId
```

```

0x1234,      // ProductId
"Vendor",   // VendorName
"Bulk device", // ProductName
"13245678"  // SerialNumber
}
...
USBD_SetDeviceInfo(&_DeviceInfo);
...
USBD_Start();

```

This structure and functions are included in every example application and can be used without modifications in the development phase of your application, but you may not bring a product on the market without modifying the Vendor ID and Product ID.

Ids	Description
Default Vendor ID for all applications	
0x8765	Example Vendor ID for all examples. Do not use this in real products!
Used Product IDs	
0x1240	Example Product ID for all bulk samples.
0x1234	Example Product ID for deprecated bulk samples (using SEGGER Windows driver)
0x1200	Example Product ID for the MSD CD-ROM sample.
0x1000	Example Product ID for all MSD samples.
0x1088	Example Product ID for all UVC samples.
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.
0x1115	Example Product ID for HID keyboard sample.
0x1310	Example Product ID for the Audio Speaker sample.
0x1311	Example Product ID for the Audio Microphone sample.
0x1312	Example Product ID for the Audio Headset sample.
0x1350	Example Product ID for the MIDI sample.
0x2114	Example Product ID for the Printer class sample.
0x3000	Example Product ID for RNDIS sample.
0x3003	Example Product ID for ECM sample.
0x3004	Example Product ID for IP-over-USB sample.
0x3005	Example Product ID for NCM sample.

3.3.2 Additional required configuration for emUSB-MSD

Refer to *MSD Configuration* on page 223 for more information about the required additional configuration functions for emUSB-MSD.

3.3.3 Descriptors

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

Some optional descriptors may be enabled by calling the following functions:

- USBD_EnableIAD()
- USBD_UseV210()

- `USBD_EnableSuperSpeed()`
- `USBD_SetWebUSBInfo()`

3.3.4 Compile-time configuration

This chapter describes the optional defines which can be used in the file `USB_Conf.h` can be used to set configuration defines for emUSB-Device.

emUSB-Device can be used without changing any of the compile-time flags. All compile-time configuration flags are pre-configured with valid values which match the requirements of most applications.

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

Define	Default	Description
<code>USBD_OS_USE_USBD_X_INTERRUPT</code>	0	If set emUSB-Device will use the functions <code>USBD_X_EnableInterrupt()</code> / <code>USBD_X_DisableInterrupt()</code>
<code>USB_DEBUG_LEVEL</code>	2	Sets the debug level.
		0 - No checks and no logs.
		1 - Support "Panic" checks.
		2 - Support warn & log messages.
<code>USB_SUPPORT_TRANSFER_ISO</code>	0	Should isochronous transfers be supported? ISO is only required for audio and video classes.
<code>USB_MAX_POWER</code>	50	The maximum current consumption of the device in $x*2$ mA (e.g. 50 means 100 mA).
<code>USBD_SUPPORT_PROFILE</code>	0	Support profiling via SystemView?

3.4 Host OS specifics

3.4.1 Windows registry

The Windows registry is a database which stores settings for the operating system. The relevant aspect of the Windows registry in regard to USB development is the fact that Windows stores information about connected USB devices into the registry. Normally Windows stores the Vendor and Product ID pair together with the USB configuration of that particular device in the registry. During USB development this can have negative effects because, if you, the developer, change the USB configuration of a device Windows will still have the old USB configuration saved in the registry. While the USB device is functioning perfectly fine the old registry entry can result in the device not being properly recognized by Windows.

This issue is especially prevalent when developing an USB Audio device.

3.4.1.1 Cleaning the Windows registry

Easiest is to use a tool such as Uwe Sieber's "Device Cleanup Tool": https://www.uwe-sieber.de/misc_tools_e.html This tool allows any **not connected** devices to be removed from the registry.

Alternatively the registry can be cleaned by hand using the Windows registry editor.

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 emUSB-Device. These functions are required for all USB class drivers and the unclassified bulk communication component.

General information

To communicate with the host, the example applications include a USB-specific header `USB.h`. This file contains API functions to communicate with the USB host through the USB Core driver.

Every application using USB Core must perform the following steps:

1. Initialize the USB stack. To initialize the USB stack `USBD()` has to be called. `USBD_Init()` performs the low-level initialization of the USB stack and calls `USBD_X_Config()` 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-Device bulk component, refer to `USB_BULK_INIT_DATA` on page 122 for information about the initialization structure that is required when you want to add a bulk interface. For the emUSB-Device MSD component, refer to `USB_MSD_INIT_DATA` on page 241 and `USB_MSD_INST_DATA` on page 243 for information about the initialization structures that are required when you want to add an MSD interface. For the emUSB-Device CDC component, refer to `USB_CDC_INIT_DATA` on page 373 for information about the initialization structure that is required when you want to add a CDC interface. For the emUSB-Device HID component, refer to `USB_HID_INIT_DATA` on page 400 for information about the initialization structure that is required when you want to add a HID interface.
3. Provide device information using `USBD_SetDeviceInfo()`.
4. Start the USB stack. Call `USBD_Start()` to start the USB stack.

Example applications for every supported USB class and the unclassified bulk component are supplied. We recommend using one of these examples as a 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>USBD_Init()</code>	Initializes the USB device with its settings.
<code>USBD_Start()</code>	Starts the emUSB-Device Core.
<code>USBD_GetVersion()</code>	Returns the version of the stack.
<code>USBD_GetState()</code>	Returns the state of the USB device.
<code>USBD_IsConfigured()</code>	Checks if the USB device is initialized and ready.
<code>USBD_GetSpeed()</code>	Returns the current connection speed.
<code>USBD_Stop()</code>	Stops the USB communication.
<code>USBD_DeInit()</code>	De-initialize the complete USB stack.
USB configuration functions	
<code>USBD_AddDriver()</code>	Adds a USB device driver to the USB stack.
<code>USBD_SetISREnableFunc()</code>	Register function to enable USB interrupts.
<code>USBD_SetAttachFunc()</code>	Sets a function to perform hardware-specific actions to attach USB.
<code>USBD_AddEP()</code>	Returns an endpoint "handle" that can be used for the desired USB interface.
<code>USBD_AddEPEX()</code>	Returns an endpoint "handle" that can be used for the desired USB interface.
<code>USBD_SetDeviceInfo()</code>	Sets a all information used during device enumeration.
<code>USBD_SetClassRequestHook()</code>	Sets a callback function that is called when a setup class request is sent from the host to the specified interface index.
<code>USBD_SetVendorRequestHook()</code>	Sets a callback function that is called when a setup vendor request is sent from the host to the specified interface index.
<code>USBD_SetIsSelfPowered()</code>	Sets whether the device is self-powered or not.
<code>USBD_SetMaxPower()</code>	Sets the maximum power consumption reported to the host during enumeration.
<code>USBD_SetOnEvent()</code>	Sets a callback function for an endpoint that will be called on every RX or TX event for that endpoint.
<code>USBD_RemoveOnEvent()</code>	Removes a callback function which was added via <code>USBD_SetOnEvent</code> from the callback list.
<code>USBD_SetOnRxEP0()</code>	Sets a callback when data are received in the data stage of the setup request.
<code>USBD_SetOnRXHookEP()</code>	Sets a callback whenever data are received from a given endpoint handle.
<code>USBD_SetOnSetup()</code>	Sets a callback function that is called when any setup request is sent from the host.
<code>USBD_SetOnSetupHook()</code>	Obsolete, use <code>USBD_SetOnSetup()</code> .
<code>USBD_WriteEP0FromISR()</code>	Write data to EP0 (control endpoint).

Function	Description
<code>USBD_EnableIAD()</code>	Enables combination of multi-interface device classes with single-interface classes or other multi-interface classes.
<code>USBD_SetCacheConfig()</code>	Configures cache related functionality that might be required by the stack for cache handling in drivers.
<code>USBD_RegisterSCHook()</code>	Sets a callback function that will be called on every state change of the USB device.
<code>USBD_AssignMemory()</code>	Assigns an area of RAM to be used for the endpoint buffers and transfer descriptors by the USB driver.
<code>USBD_UseV210()</code>	Enable use of USB V2.10 specification revision.
<code>USBD_EnableSuperSpeed()</code>	Enable SuperSpeed in the USB stack.
<code>USBD_SetWebUSBInfo()</code>	For WebUSB capable USB devices this function may be called before <code>USBD_Start()</code> to enable WebUSB specific descriptors.
<code>USBD_SetCheckAddress()</code>	Installs a function that checks if an address can be used for DMA transfers.
USB I/O functions	
<code>USBD_Read()</code>	Reads data from the host.
<code>USBD_ReadOverlapped()</code>	Reads data from the host asynchronously.
<code>USBD_Receive()</code>	Reads data from host.
<code>USBD_ReceivePoll()</code>	Reads data from host.
<code>USBD_ReadAsync()</code>	Reads data from the host asynchronously.
<code>USBD_Write()</code>	Writes data to the host.
<code>USBD_WriteAsync()</code>	Sends data to the host asynchronously.
<code>USBD_CancelIO()</code>	Cancel any read or write operation.
<code>USBD_WaitForEndOfTransferEx()</code>	Wait until the current transfer on a particular EP has completed.
<code>USBD_WaitForTXReady()</code>	Waits (blocking) until the TX queue can accept another data packet.
<code>USBD_GetNumBytesInBuffer()</code>	Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer.
<code>USBD_GetNumBytesRemToRead()</code>	This function is to be used in combination with <code>USBD_ReadOverlapped()</code> .
<code>USBD_GetNumBytesRemToWrite()</code>	This function is to be used in combination with a non-blocking call to <code>USBD_Write()</code> .
<code>USBD_StallEP()</code>	Stalls an endpoint.
USB RemoteWakeUp functions	
<code>USBD_SetAllowRemoteWakeUp()</code>	Allows the device to publish that remote wake is available.
<code>USBD_DoRemoteWakeup()</code>	Performs a remote wakeup in order to wake up the host from the standby/suspend state.
Data structures	

Function	Description
USB_ASYNC_IO_CONTEXT	Contains information for asynchronous transfers.
USB_SETUP_PACKET	Structure containing a USB setup packet.
SEGGER_CACHE_CONFIG	Used to pass cache configuration and call-back function pointers to the stack.
USB_CHECK_ADDRESS_FUNC	Checks if an address can be used for DMA transfers.
USB_WEBUSB_INFO	Information that may be provided by the application for WebUSB capable USB devices.

4.2.1 USB basic functions

4.2.1.1 USBD_GetState()

Description

Returns the state of the USB device.

Prototype

```
unsigned USBD_GetState(void);
```

Return value

A bitwise combination of the USB state flags:

USB_STAT_ATTACHED	Device is attached. (Note 1)
USB_STAT_READY	Device is ready. (Note 2)
USB_STAT_ADDRESSED	Device is addressed. (Note 3)
USB_STAT_CONFIGURED	Device is configured. (Note 4)
USB_STAT_SUSPENDED	Device is suspended. (Note 5)

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.

Notes

- (1) Attached in a USB-specification sense of the word does not mean that the device is physically connected to the host via a USB cable, it only means that the pull-up resistor on the device side is connected. The status can be "attached" regardless of whether the device is connected to a host or not. This state can normally be ignored.
- (2) Ready denotes the USB controller state, the controller is "ready" after a bus reset. This state can normally be ignored.
- (3) A device is in an addressed state after it receives a valid (non-zero) USB address from the USB host. This state can normally be ignored.
- (4) When a device is "configured" the enumeration of the device has been successfully completed and the host can communicate with the device.
- (5) Suspend is set when the device is physically disconnected from the host or when the USB host suspends the connected device.

4.2.1.2 USBD_GetSpeed()

Description

Returns the current connection speed.

Prototype

```
int USBD_GetSpeed(void);
```

Return value

USB_SPEED_NONE	Unknown speed.
USB_SPEED_FS	Full-speed.
USB_SPEED_HS	High-speed.
USB_SPEED_SS	SuperSpeed.

4.2.1.3 USBD_Init()

Description

Initializes the USB device with its settings.

Prototype

```
void USBD_Init(void);
```

4.2.1.4 USBD_IsConfigured()

Description

Checks if the USB device is initialized and ready.

Prototype

```
char USBD_IsConfigured(void);
```

Return value

- 0 USB device is not configured.
- 1 USB device is configured.

4.2.1.5 USBD_Start()

Description

Starts the emUSB-Device Core.

Prototype

```
void USBD_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.

4.2.1.6 USBD_Stop()

Description

Stops the USB communication. This also makes sure that the device is detached from the HOST.

Prototype

```
void USBD_Stop(void);
```

4.2.1.7 USB_DelInit()

Description

De-initialize the complete USB stack.

Prototype

```
void USB_DelInit(void);
```

Additional information

This function also calls `USB_Stop()` internally.

4.2.1.8 USBD_GetVersion()

Description

Returns the version of the stack.

Prototype

```
U32 USBD_GetVersion(void);
```

Return value

Format: Mmmrr; e.g: 32401 is 3.24a

4.2.2 USB configuration functions

4.2.2.1 USBD_AddDriver()

Description

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

Prototype

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

Parameters

Parameter	Description
<code>pDriver</code>	Pointer to the driver API structure.

Additional information

To add the driver, use `USBD_AddDriver()` with the identifier of the compatible driver. Refer to the section "Available target USB drivers" in the `USB.h` header file for a list of supported devices and their valid identifiers.

Example

```

/*****
 *
 *      USB_D_X_Config
 */
void USB_D_X_Config(void) {
    BSP_USB_Init();
    USB_DRIVER_LPC17xx_ConfigAddr(0x2008C000); // USB controller of LPC1788
                                              // is located @ 0x2008C000

    USBD_AddDriver(&USB_Driver_NXPLPC17xx);
    USBD_SetISREnableFunc(_EnableISR, NULL, NULL);
}

```

4.2.2.2 USBD_SetISREnableFunc()

Description

Register function to enable USB interrupts.

Prototype

```
void USBD_SetISREnableFunc(USB_ENABLE_ISR_FUNC * pfEnableISR);
```

Parameters

Parameter	Description
<code>pfEnableISR</code>	Pointer to the function to install the interrupt handler and enable the USB interrupt.

Additional information

This function must be called within `USB_D_X_Config()` function. See *Adding a driver to emUSB-Device* on page 659. The functions pointer prototype is defined as follows:

```
typedef void USB_ENABLE_ISR_FUNC (USB_ISR_HANDLER * pfISRHandler);
```

Example

See `USB_D_AddDriver()`.

4.2.2.3 USBD_SetAttachFunc()

Description

Sets a function to perform hardware-specific actions to attach USB.

Prototype

```
void USBD_SetAttachFunc(USB_ATTACH_FUNC * pfAttach);
```

Parameters

Parameter	Description
<code>pfAttach</code>	Pointer to the attach function.

Additional information

This function must be called within `USB_D_X_Config()` function. See *Adding a driver to emUSB-Device* on page 659. The functions pointer prototypes are defined as follows:

```
typedef void USB_ATTACH_FUNC (void);
```

Example

See `USB_D_X_Config()` on page 659.

4.2.2.4 USBD_AddEP()

Description

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

Prototype

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

Parameters

Parameter	Description
<code>InDir</code>	Specifies the direction of the desired endpoint. <ul style="list-style-type: none"> • <code>USB_DIR_IN</code> • <code>USB_DIR_OUT</code>
<code>TransferType</code>	Specifies the transfer type of the endpoint. The following values are allowed: <ul style="list-style-type: none"> • <code>USB_TRANSFER_TYPE_BULK</code> • <code>USB_TRANSFER_TYPE_INT</code> ISO endpoints must be created using <code>USBD_AddEPEx()</code> .
<code>Interval</code>	Specifies the interval measured in units of 125us (micro frames). This value should be zero for a bulk endpoint.
<code>pBuffer</code>	Pointer to a buffer that is used for OUT-transactions. For IN-endpoints this parameter must be <code>NULL</code> .
<code>BufferSize</code>	Size of the buffer (OUT endpoints only). Must be a multiple of the maximum packet size.

Return value

> 0 A valid endpoint handle is returned.
= 0 Error.

Additional information

The `Interval` parameter specifies the frequency in which the endpoint should be polled for information by the host. It must be specified in units of 125 us.

Depending on the actual speed of the device during enumeration, the USB stack converts the interval to the correct value required for the endpoint descriptor according to the USB specification (into milliseconds for low/full-speed, into 125 us for high-speed).

For endpoints of type `USB_TRANSFER_TYPE_BULK` the value is ignored and should be set to 0.

This function must be called after `USBD_Init()` and before `USBD_Start()`.

4.2.2.5 USBD_AddEPEX()

Description

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

Prototype

```
unsigned USBD_AddEPEX(const USB_ADD_EP_INFO * pInfo,
                    U8 * pBuffer,
                    unsigned BufferSize);
```

Parameters

Parameter	Description
<code>pInfo</code>	Pointer to a structure of type <code>USB_ADD_EP_INFO</code> .
<code>pBuffer</code>	Pointer to an endpoint buffer that is used for OUT-transactions. For IN-endpoints or ISO endpoints this parameter should be <code>NULL</code> .
<code>BufferSize</code>	Size of the endpoint buffer (OUT endpoints only). Must be \geq the maximum packet size of the endpoint. For IN-endpoints or ISO endpoints this parameter should be 0.

Return value

> 0 A valid endpoint handle is returned.
 = 0 Error.

Additional information

This function must be called after `USBD_Init()` and before `USBD_Start()`.

4.2.2.6 USBD_SetDeviceInfo()

Description

Sets a all information used during device enumeration.

Prototype

```
void USBD_SetDeviceInfo(const USB_DEVICE_INFO * pDeviceInfo);
```

Parameters

Parameter	Description
<code>pDeviceInfo</code>	Pointer to a structure containing the device information. Must point to static data that is not changed while the stack is running.

Additional information

See `USB_DEVICE_INFO` on page 48 for a description of the structure.

Example

See `USB_DEVICE_INFO` on page 48.

4.2.2.7 USBD_SetClassRequestHook()

Description

Sets a callback function that is called when a setup class request is sent from the host to the specified interface index.

Prototype

```
void USBD_SetClassRequestHook(unsigned InterfaceNum,  
                               USB_ON_CLASS_REQUEST * pfOnClassRequest);
```

Parameters

Parameter	Description
InterfaceNum	Interface index that for setting the class request callback.
pfOnClassRequest	Pointer to the callback.

Additional information

Note that the callback will be called within an ISR, therefore it should never block. 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_CLASS_REQUEST(const USB_SETUP_PACKET * pSetupPacket);
```

4.2.2.8 USBD_SetVendorRequestHook()

Description

Sets a callback function that is called when a setup vendor request is sent from the host to the specified interface index.

Prototype

```
void USBD_SetVendorRequestHook(unsigned InterfaceNum,  
                               USB_ON_CLASS_REQUEST * pfOnVendorRequest);
```

Parameters

Parameter	Description
InterfaceNum	Interface index that for setting the class request callback.
pfOnVendorRequest	Pointer to the callback.

Additional information

Note that the callback will be called within an ISR, therefore it should never block. 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_CLASS_REQUEST(const USB_SETUP_PACKET * pSetupPacket);
```

4.2.2.9 USBD_SetIsSelfPowered()

Description

Sets whether the device is self-powered or not. Obsolete function, please use `USB_SetMaxPower()`.

Prototype

```
void USBD_SetIsSelfPowered(U8 IsSelfPowered);
```

Parameters

Parameter	Description
<code>IsSelfPowered</code>	<ul style="list-style-type: none">• 0 - Device is not self-powered.• 1 - Device is self-powered.

Additional information

This function has to 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).

4.2.2.10 USBD_SetMaxPower()

Description

Sets the maximum power consumption reported to the host during enumeration. This function also sets whether the device is self-powered (`MaxPower = 0`) or not.

Prototype

```
void USBD_SetMaxPower(unsigned MaxPower);
```

Parameters

Parameter	Description
<code>MaxPower</code>	Maximum power consumption of the device given in mA. <code>MaxPower</code> shall be in range between 0mA - 500mA, for SuperSpeed devices up to 900mA.

Additional information

This function shall be called before `USB_Start()`, as it will specify how much power the device will consume from the host. If this function is not called, a default value of 100 mA will be used.

4.2.2.11 USBD_SetOnEvent()

Description

Sets a callback function for an endpoint that will be called on every RX or TX event for that endpoint.

Prototype

```
void USBD_SetOnEvent(unsigned          EPIndex,
                    USB_EVENT_CALLBACK * pEventCb,
                    USB_EVENT_CALLBACK_FUNC * pfEventCb,
                    void                * pContext);
```

Parameters

Parameter	Description
<code>EPIndex</code>	Endpoint index returned by <code>USBD_AddEP()</code> .
<code>pEventCb</code>	Pointer to a <code>USB_EVENT_CALLBACK</code> structure (will be initialized by this function).
<code>pfEventCb</code>	Pointer to the callback routine that will be called on every event on the USB endpoint.
<code>pContext</code>	A pointer which is used as parameter for the callback function.

Additional information

The `USB_EVENT_CALLBACK` structure is private to the USB stack. It will be initialized by `USBD_SetOnEvent()`. The USB stack keeps track of all event callback functions using a linked list. The `USB_EVENT_CALLBACK` structure will be included into this linked list and must reside in static memory.

The callback function is called only, if a read or write operation was started for the endpoint using one of the `USBD_Read...()` or `USBD_Write...()` functions.

Additional information

The callback function has the following prototype:

```
typedef void USB_EVENT_CALLBACK_FUNC(unsigned Events, void *pContext);
```

Parameter	Description
<code>Events</code>	A bit mask indicating which events occurred on the endpoint.
<code>pContext</code>	The pointer which was provided to the <code>USB_SetOnEvent()</code> function.

Note that the callback function will be called within an ISR, therefore it should never block. The first parameter to the callback function will contain a bit mask for all events that triggered the call:

Event	Description
<code>USB_EVENT_DATA_READ</code>	Some data was received from the host on the endpoint.
<code>USB_EVENT_DATA_SEND</code>	Some data was sent to the host, so that (part of) the user write buffer may be reused by the application.
<code>USB_EVENT_DATA_ACKED</code>	Some data was acknowledged by the host.
<code>USB_EVENT_READ_COMPLETE</code>	The last read operation was completed.

Event	Description
<code>USB_EVENT_READ_ABORT</code>	A read transfer was aborted.
<code>USB_EVENT_WRITE_ABORT</code>	A write transfer was aborted.
<code>USB_EVENT_WRITE_COMPLETE</code>	All write operations were completed.

Example

```

// The callback function.
static void _OnEvent(unsigned Events, void *pContext) {
    if ((Events & USB_EVENT_DATA_SEND) != 0 &&
        // Check for last write transfer to be completed.
        USBD_GetNumBytesRemToWrite(EPIndex) == 0) {
        <.. prepare next data for writing..>
        // Send next packet of data.
        r = USBD_Write(EPIndex, &ac[0], 200, 0, -1);
        if (r < 0) {
            <.. error handling..>
        }
    }
}
// Main programm.
// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USB_D_SetOnEvent(EPIndex, &_usb_callback, _OnEvent, NULL);
// Send the first packet of data using an asynchronous write operation.
r = USBD_Write(EPIndex, &ac[0], 200, 0, -1);
if (r < 0) {
    <.. error handling..>
}
<.. do anything else here while the whole data is send..>

```

4.2.2.12 USBD_RemoveOnEvent()

Description

Removes a callback function which was added via `USBD_SetOnEvent` from the callback list.

Prototype

```
void USBD_RemoveOnEvent (      unsigned      EPIndex,  
                             const USB_EVENT_CALLBACK * pEventCb);
```

Parameters

Parameter	Description
<code>EPIndex</code>	Endpoint index returned by <code>USBD_AddeP()</code> .
<code>pEventCb</code>	Pointer to a <code>USB_EVENT_CALLBACK</code> structure which was used with <code>USBD_SetOnEvent</code> .

4.2.2.13 USBD_SetOnRxEP0()

Description

Sets a callback when data are received in the data stage of the setup request.

Prototype

```
void USBD_SetOnRxEP0(USB_ON_RX_FUNC * pOnRx);
```

Parameters

Parameter	Description
<code>pOnRx</code>	Pointer to a function that should be called when receiving data other than setup packets on EP0.

Additional information

Please note that this function can be called multiple times from different classes in order to check the data.

Note that the callback will be called within an ISR, therefore it should never block. 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);
```

4.2.2.14 USBD_SetOnRXHookEP()

Description

Sets a callback whenever data are received from a given endpoint handle. The callback function is called within the interrupt context and must not block.

Prototype

```
void USBD_SetOnRXHookEP(unsigned      EPIndex,  
                        USB_ON_RX_FUNC * pfOnRx);
```

Parameters

Parameter	Description
EPIndex	Any valid endpoint handle > 0.
pfOnRx	Pointer to the callback.

USB_ON_RX_FUNC is defined as follows:

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

4.2.2.15 USBD_SetOnSetup()

Description

Sets a callback function that is called when any setup request is sent from the host.

Prototype

```
void USBD_SetOnSetup(USB_SETUP_HOOK * pHook,
                    USB_ON_SETUP * pfOnSetup);
```

Parameters

Parameter	Description
pHook	Pointer to a USB_SETUP_HOOK structure (will be initialized by this function).
pfOnSetup	Pointer to the callback function.

Additional information

The USB_SETUP_HOOK structure is private to the USB stack. It will be initialized by USBD_SetOnSetup(). The USB stack keeps track of all setup callback functions using a linked list. The USB_SETUP_HOOK structure will be included into this linked list and must reside in static memory.

Note that the callback will be called within an ISR, therefore it should never block. If it is necessary to send data from the callback function through endpoint 0, use the function USBD_WriteEP0FromISR().

USB_ON_SETUP is defined as follows:

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

4.2.2.16 USBD_SetOnSetupHook()

Description

Obsolete, use `USB_SetOnSetup()`. Sets a callback function that is called when any setup request is sent from the host.

Prototype

```
void USBD_SetOnSetupHook(unsigned InterfaceNum,  
                          USB_ON_SETUP * pOnSetup);
```

Parameters

Parameter	Description
InterfaceNum	Interface index that for setting the setup request callback.
pOnSetup	Pointer to the callback function.

Additional information

Note that the callback will be called within an ISR, therefore it should never block. If it is necessary to send data from the callback function through endpoint 0, use the function `USB_WriteEP0FromISR()`.

4.2.2.17 USBD_WriteEP0FromISR()

Description

Write data to EP0 (control endpoint). This function may be called in an interrupt context.

Prototype

```
void USBD_WriteEP0FromISR(const void * pData,  
                          unsigned NumBytes,  
                          char Send0PacketIfRequired);
```

Parameters

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.

4.2.2.18 USBD_EnableIAD()

Description

Enables combination of multi-interface device classes with single-interface classes or other multi-interface classes.

Prototype

```
void USBD_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. Multi-interface device classes, such as CDC, Audio, MIDI use 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 a multi-interface class, but it may not be correctly recognized by the host, this is not standardized and depends on 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 a multi-interface component with any other component, please call `USBD_EnableIAD()` before adding the multi-interface component through `USBD_*_Add()`.

4.2.2.19 USBD_SetCacheConfig()

Description

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

Prototype

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

Parameters

Parameter	Description
<code>pConfig</code>	Pointer to an element of <code>SEGGER_CACHE_CONFIG</code> .
<code>ConfSize</code>	Size of the passed structure in case library and header size of the structure differs.

Additional information

This function has to called in `USBX_Config()`. This function replaces the legacy cache functions `BSP_CACHE_CleanRange` and `BSP_CACHE_InvalidateRange`. If you still want to use these routines please set `USBX_USE_LEGACY_CACHE_ROUTINES` to 1 in your `USB_Conf.h` file.

4.2.2.20 USBD_RegisterSCHook()

Description

Sets a callback function that will be called on every state change of the USB device.

Prototype

```
int USBD_RegisterSCHook(USB_HOOK          * pHook,
                       USB_STATE_CALLBACK_FUNC * pfStateCb,
                       void                * pContext);
```

Parameters

Parameter	Description
<code>pHook</code>	Pointer to a <code>USB_HOOK</code> structure (will be initialized by this function).
<code>pfStateCb</code>	Pointer to the callback routine that will be called on every state change.
<code>pContext</code>	A pointer which is used as parameter for the callback function.

Return value

- 0 OK.
- 1 Error, specified hook already exists.

Additional information

The `USB_HOOK` structure is private to the USB stack. It will be initialized by `USBD_RegisterSCHook()`. The USB stack keeps track of all state change callback functions using a linked list. The `USB_HOOK` structure will be included into this linked list and must reside in static memory.

Note that the callback function will be called within an ISR, therefore it should never block.

Example

```
// The callback function.
static void _OnStateChange(void *pContext, U8 NewState) {
    if ((NewState & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED)) == USB_STAT_CONFIGURED) {
        // Device is enumerated
    } else {
        // Device not enumerated
    }
}
// Main programm.
static USB_HOOK Hook;

USB_D_Init();
...
USBD_RegisterSCHook(&Hook, _OnStateChange, NULL);
...
USB_D_Start();
```

4.2.2.21 USBD_AssignMemory()

Description

Assigns an area of RAM to be used for the endpoint buffers and transfer descriptors by the USB driver. This function should be called from within the `USB_D_X_Config()` function. Not all drivers support this function.

If the driver uses DMA, the USB controller must have DMA access to this area. For some drivers, the memory should be aligned to a given boundary. If not aligned, the driver will increase the start address and reduce the size of the area to achieve proper alignment. This results in wasting of RAM and may cause the driver to run out of memory.

Prototype

```
void USBD_AssignMemory(void * pMem,
                      U32    MemSize);
```

Parameters

Parameter	Description
<code>pMem</code>	Pointer to the start of the RAM area to be used by the driver.
<code>MemSize</code>	Size of the RAM area in bytes.

Additional information

If the memory is not sufficient for the class and endpoint configuration, the USB driver will run into the `USB_OS_Panic()` function during initialization, if compiled for DEBUG mode (`USB_DEBUG_LEVEL > 0`). After successful initialization, the driver will usually issue a `USB_LOG()` message to report, how many bytes of the assigned memory are not used. The size of the memory area may then be adjusted.

Information how to calculate the size of the endpoint buffer memory and about any alignment requirements can be found in *Device driver specifics* on page 663.

4.2.2.22 USBD_UseV210()

Description

Enable use of USB V2.10 specification revision. This function should be called before after `USB_Start()`. Use only, if the device supports link power management (LPM).

Prototype

```
void USBD_UseV210(void);
```

4.2.2.23 USBD_EnableSuperSpeed()

Description

Enable SuperSpeed in the USB stack. Must be called in `USB_D_X_Config()`. If the USB driver or USB controller does not support SuperSpeed, calling this function has no effect.

Prototype

```
void USBD_EnableSuperSpeed(void);
```

4.2.2.24 USBD_SetWebUSBInfo()

Description

For WebUSB capable USB devices this function may be called before `USB_Start()` to enable WebUSB specific descriptors. This function can be used only, if the USB controller supports USB 2.1 compatibility, especially link power management (LPM).

Prototype

```
void USBD_SetWebUSBInfo(const USB_WEBUSB_INFO * pWebUSBInfo);
```

Parameters

Parameter	Description
<code>pWebUSBInfo</code>	Pointer to a structure containing the device information. Must point to static data that is not changed while the stack is running.

4.2.2.25 USBD_SetCheckAddress()

Description

Installs a function that checks if an address can be used for DMA transfers. Installed function must return 0, if DMA access is allowed for the given address, 1 otherwise.

Prototype

```
void USBD_SetCheckAddress(USB_CHECK_ADDRESS_FUNC * pfCheckValidDMAAddress);
```

Parameters

Parameter	Description
<code>pfCheckValidDMAAddress</code>	Pointer to the function.

Additional information

If the function reports a memory region not valid for DMA, the driver uses a temporary transfer buffer to copy data to and from this area.

4.2.3 USB I/O functions

4.2.3.1 USBD_Read()

Description

Reads data from the host.

Prototype

```
int USBD_Read(unsigned EPOut,
              void      * pData,
              unsigned  NumBytesReq,
              unsigned  Timeout);
```

Parameters

Parameter	Description
<code>EPOut</code>	Handle to an OUT endpoint returned by <code>USBD_AddEP()</code> .
<code>pData</code>	Pointer to a buffer where the received data will be stored.
<code>NumBytesReq</code>	Number of bytes to read.
<code>Timeout</code>	<code>Timeout</code> given in milliseconds. A zero value results in an infinite timeout.

Return value

= NumBytes Requested data was successfully read within the given timeout.
 ≥ 0 && < NumBytes `Timeout` has occurred (Number of bytes read before timeout).
 < 0 An error occurred.

Additional information

This function blocks the task until all data has been read or a timeout occurs. In case of a reset or a disconnect `USB_STATUS_ERROR` is returned.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via the `USBD_AddEP()` function. This data can be retrieved by a later call to `USBD_Receive()` or `USBD_Read()`. See also `USBD_GetNumBytesInBuffer()`.

In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 113).

4.2.3.2 USBD_ReadOverlapped()

Description

Reads data from the host asynchronously.

Prototype

```
int USBD_ReadOverlapped(unsigned EPOut,
                        void * pData,
                        unsigned NumBytesReq);
```

Parameters

Parameter	Description
EPOut	Handle to an OUT endpoint returned by <code>USBD_AddEP()</code> .
pData	Pointer to a buffer where the received data will be stored.
NumBytesReq	Number of bytes to read.

Return value

- ≥ 0 Number of bytes that have been read from the internal buffer (success).
- = 0 No data was found in the internal buffer, read transfer started (success).
- < 0 An error occurred.

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, `USBD_WaitForEndOfTransfer()` needs to be called.

Another synchronization method would be to periodically call `USBD_GetNumBytesRemToRead()` in order to see how many bytes still need to be received (this method is preferred when a non-blocking solution is necessary).

The read operation can be canceled using `USBD_CancelIO()`.

The buffer pointed to by [pData](#) must be valid until the read operation is terminated.

4.2.3.3 USBD_Receive()

Description

Reads data from host. The function blocks until any data have been received. In contrast to `USB_Read()` this function does not wait for all of `NumBytes` to be received, but returns after the first packet has been received or after the timeout occurs. In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 113).

Prototype

```
int USBD_Receive(unsigned EPOut,
                void * pData,
                unsigned NumBytesReq,
                int Timeout);
```

Parameters

Parameter	Description
<code>EPOut</code>	Handle to an OUT endpoint returned by <code>USBD_AddEP()</code> .
<code>pData</code>	Pointer to a buffer where the received data will be stored.
<code>NumBytesReq</code>	Number of bytes to read.
<code>Timeout</code>	<code>Timeout</code> given in milliseconds. A zero value results in an infinite timeout. If <code>Timeout</code> is -1, the function never blocks and only reads data from the internal endpoint buffer.

Return value

- > 0 Number of bytes that have been read within the given timeout.
- = 0 A timeout occurred (if `Timeout > 0`), zero packet received (not every controller supports this!), no data in buffer (if `Timeout < 0`) or the target was disconnected during the function call and no data was read so far.
- < 0 An error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling `USBD_Receive()` will return as much data as is currently available up to the size of the buffer specified within the specified timeout. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return the number of bytes read so far. If the target was disconnected before this function was called, it returns `USB_STATUS_ERROR`.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USBD_AddEP()`. This data can be retrieved by a later call to `USBD_Receive()` / `USBD_Read()`. See also `USBD_GetNumBytesInBuffer()`.

A call of `USBD_Receive(EPOut, NULL, 0, -1)` can be used to trigger an asynchronous read that stores the data into the internal buffer.

4.2.3.4 USBD_ReceivePoll()

Description

Reads data from host. The function blocks until any data have been received. In contrast to `USBD_Read()` this function does not wait for all of `NumBytes` to be received, but returns after the first packet has been received or after the timeout occurs. In contrast to `USBD_Receive()` this function will continue the read transfer asynchronously in case of a timeout.

Prototype

```
int USBD_ReceivePoll(unsigned EPOut,
                    void * pData,
                    unsigned NumBytesReq,
                    unsigned Timeout);
```

Parameters

Parameter	Description
<code>EPOut</code>	Handle to an OUT endpoint returned by <code>USBD_AddEP()</code> .
<code>pData</code>	Pointer to a buffer where the received data will be stored.
<code>NumBytesReq</code>	Number of bytes to read.
<code>Timeout</code>	<code>Timeout</code> given in milliseconds. A zero value results in an infinite timeout.

Return value

- > 0 Number of bytes that have been read within the given timeout.
- = 0 A timeout occurred (if `Timeout > 0`) or a zero packet received (not every controller supports this!), or the target was disconnected during the function call and no data was read so far.
- < 0 An error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling `USBD_ReceivePoll()` will return as much data as is currently available up to the size of the buffer specified within the specified timeout. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return the number of bytes read so far. If the target was disconnected before this function was called, it returns `USB_STATUS_ERROR`.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USBD_AddEP()`. This data can be retrieved by a later call to `USBD_Receive()` / `USBD_Read()`. See also `USBD_GetNumBytesInBuffer()`.

If a timeout occurs, the read transfer is not affected. Data send from the host after the timeout is stored into the internal buffer of the endpoint and can be read by later calls to `USBD_ReceivePoll()`.

If `Timeout = 0`, the function behaves like `USBD_Receive()`.

4.2.3.5 USBD_ReadAsync()

Description

Reads data from the host asynchronously. The function does not wait for the data to be received. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_ReadAsync(unsigned          EPIndex,
                    USB_ASYNC_IO_CONTEXT * pContext,
                    int                ShortRead);
```

Parameters

Parameter	Description
<code>EPIndex</code>	Handle to an OUT endpoint returned by <code>USBD_AddEP()</code> .
<code>pContext</code>	Pointer to an I/O context containing parameters and pointer to the callback function.
<code>ShortRead</code>	<ul style="list-style-type: none"> 0: The transfer is completed successfully after all bytes have been read. 1: The transfer is completed successfully after one packet has been read.

4.2.3.6 USBD_Write()

Description

Writes data to the host. Depending on the Timeout parameter, the function may block until `NumBytes` have been written or a timeout occurs.

Prototype

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

Parameters

Parameter	Description
<code>EPIndex</code>	Handle to an IN endpoint returned by <code>USBD_AddEP()</code> .
<code>pData</code>	Pointer to data that should be sent to the host.
<code>NumBytes</code>	Number of bytes to be written.
<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> .
<code>ms</code>	Timeout in milliseconds. 0 means infinite. If Timeout is -1, the function returns immediately and the transfer is processed asynchronously.

Return value

= 0 Successful started an asynchronous write transfer or a timeout has occurred and no data was written.
 > 0 && < `NumBytes` Number of bytes that have been written before a timeout occurred.
 = `NumBytes` Write transfer successful completed.
 < 0 An error occurred.

Additional information

This function also returns when the target is disconnected from host or when a USB reset occurred.

The USB stack is able to queue a small number of asynchronous write transfers (Timeout = -1). If a write transfer is still in progress when this function is called and the USB stack can not accept another write transfer request, the functions returns `USB_STATUS_EP_BUSY`. A synchronous write transfer (Timeout ≥ 0) will always block until the transfer (including all pending transfers) are finished.

In order to synchronize, `USBD_WaitForEndOfTransfer()` needs to be called. Another synchronization method would be to periodically call `USBD_GetNumBytesRemToWrite()` in order to see how many bytes still need to be written (this method is preferred when a non-blocking solution is necessary).

In case of a timeout, the write transfer is aborted (see *Timeout handling* on page 113).

The write operation can be canceled using `USBD_CancelIO()`.

If `pData` = NULL and `NumBytes` = 0, a zero-length packet is sent to the host.

The content of the buffer pointed to by `pData` must not be changed until the transfer has been completed.

4.2.3.7 USBD_WriteAsync()

Description

Sends data to the host asynchronously. The function does not wait for the data to be sent. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_WriteAsync(unsigned          EPIndex,
                    USB_ASYNC_IO_CONTEXT * pContext,
                    char                Send0PacketIfRequired);
```

Parameters

Parameter	Description
<code>EPIndex</code>	Handle to an IN endpoint returned by <code>USBD_AddEP()</code> .
<code>pContext</code>	Pointer to an I/O context containing parameters and pointer to the callback function.
<code>Send0PacketIfRequired</code>	Specifies that a zero-length packet shall be sent when the last data packet is a multiple of <code>MaxPacketSize</code> .

4.2.3.8 USBD_CancelIO()

Description

Cancel any read or write operation.

Prototype

```
void USBD_CancelIO(unsigned EPIndex);
```

Parameters

Parameter	Description
EPIndex	Handle to an endpoint returned by <code>USBD_AddEP()</code> .

4.2.3.9 USBD_WaitForEndOfTransferEx()

Description

Wait until the current transfer on a particular EP has completed. This function must be called from a task.

Prototype

```
int USBD_WaitForEndOfTransferEx(unsigned EPIndex,
                                unsigned Timeout,
                                int      AbortOnTimeout);
```

Parameters

Parameter	Description
<code>EPIndex</code>	Handle to the endpoint returned by <code>USBD_AddEP()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds, 0 means infinite wait.
<code>AbortOnTimeout</code>	If a timeout occurs, then the current transfer is terminated if <code>AbortOnTimeout</code> \neq 0. The current transfer is not affected in case of a timeout if <code>AbortOnTimeout</code> = 0. See also <i>Timeout handling</i> on page 113.

Return value

- 0 Transfer completed.
- 1 `Timeout` occurred.

4.2.3.10 USBD_WaitForTXReady()

Description

Waits (blocking) until the TX queue can accept another data packet. This function is used in combination with a non-blocking call to `USBD_Write()`, it waits until a new asynchronous write data transfer will be accepted by the USB stack.

Prototype

```
int USBD_WaitForTXReady(unsigned EPIndex,
                       int Timeout);
```

Parameters

Parameter	Description
<code>EPIndex</code>	Handle to an IN endpoint returned by <code>USBD_AddEP()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite. If <code>Timeout</code> is negative, the function will return immediately.

Return value

- = 0 A new asynchronous write data transfer will be accepted.
- = 1 The write queue is full, a call to `USBD_Write()` would return `USB_STATUS_EP_BUSY`.
- < 0 Error occurred.

Additional information

If `Timeout` is 0, the function never returns 1. If `Timeout` is -1, the function will not wait, but immediately return the current state.

4.2.3.11 USBD_GetNumBytesInBuffer()

Description

Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer. This functions does not start a read transfer.

Prototype

```
unsigned USBD_GetNumBytesInBuffer(unsigned EPIndex);
```

Parameters

Parameter	Description
EPIndex	Handle to an OUT endpoint returned by <code>USBD_AddEP()</code> .

Return value

Number of bytes which have been stored in the internal buffer.

Additional information

The number of bytes returned by this function can be read using `USBD_Read()` or `USBD_Receive()` without blocking.

4.2.3.12 USBD_GetNumBytesRemToRead()

Description

This function is to be used in combination with `USB_ReadOverlapped()`. It returns the number of bytes which still have to be read during the transaction.

Prototype

```
unsigned USBD_GetNumBytesRemToRead(unsigned EPIndex);
```

Parameters

Parameter	Description
<code>EPIndex</code>	Handle to an OUT endpoint returned by <code>USB_AddEP()</code> .

Return value

Number of bytes which still have to be read.

Additional information

Note that this function does not return the number of bytes that have been read, but the number of bytes which still have to be read. This function does not block.

4.2.3.13 USBD_GetNumBytesRemToWrite()

Description

This function is to be used in combination with a non-blocking call to `USB_Write()`. It returns the number of bytes which still have to be written during the transaction.

Prototype

```
unsigned USBD_GetNumBytesRemToWrite(unsigned EPIndex);
```

Parameters

Parameter	Description
<code>EPIndex</code>	Handle to an IN endpoint returned by <code>USBD_AddEP()</code> .

Return value

Number of bytes which still have to be written.

Additional information

Note that this function does not return the number of bytes that have been written, but the number of bytes which still have to be written. This function does not block.

4.2.3.14 USBD_StallEP()

Description

Stalls an endpoint.

Prototype

```
void USBD_StallEP(unsigned EPIndex);
```

Parameters

Parameter	Description
EPIndex	Handle to the endpoint handle returned by <code>USBD_AdddEP()</code> .

4.2.4 USB Remote wakeup functions

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

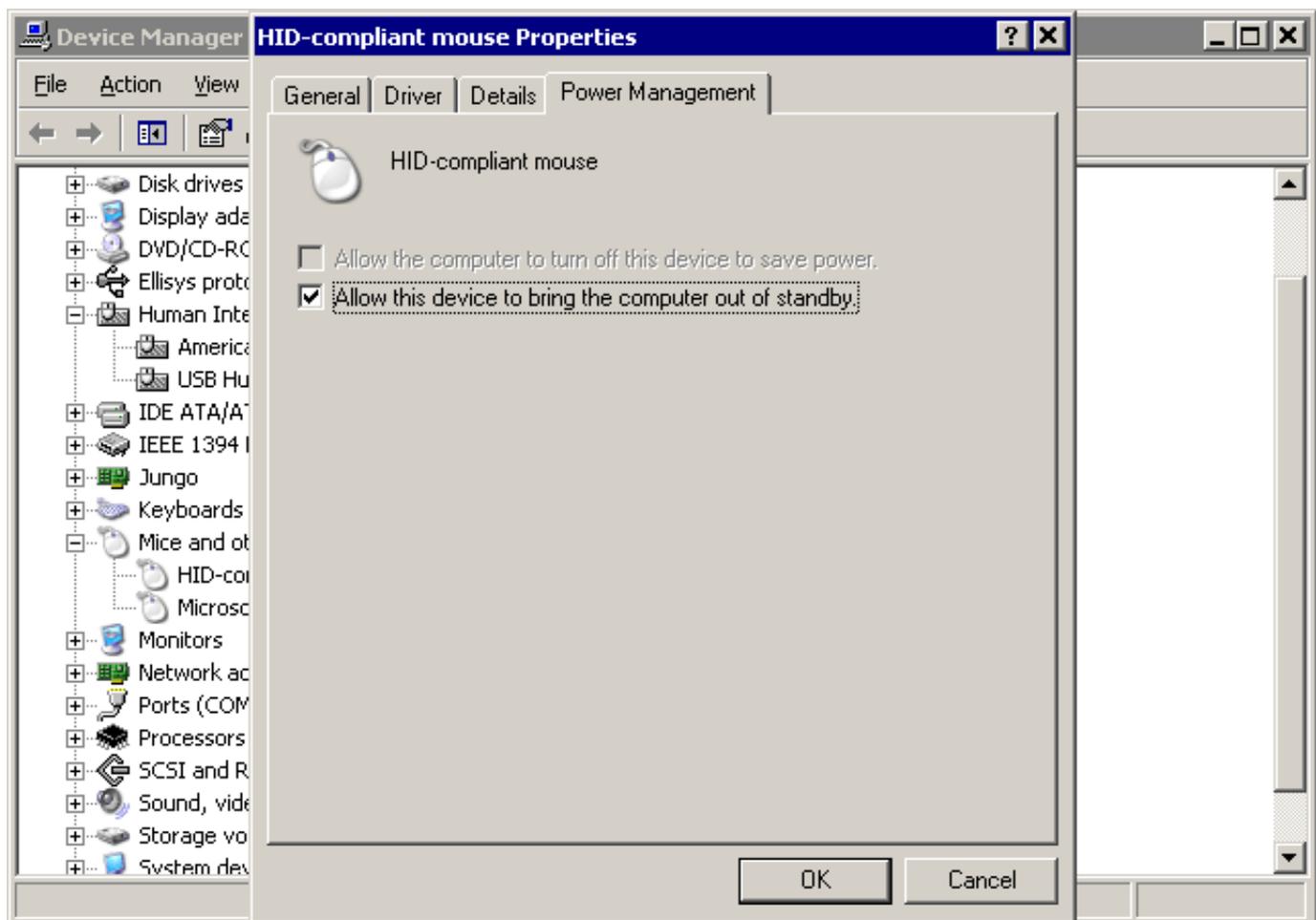
In order to do this a special resume signal is sent over the USB data lines.

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

Windows OS

Currently Windows OS only supports the wakeup feature on devices based on HID mouse/ keyboard, CDC Modem and RNDIS Ethernet class. Remote wakeup for MSD, generic bulk and CDC serial is not supported by Windows. So therefore a HID mouse class even as dummy interface within your USB configuration is currently mandatory.

Windows must also be told that the device shall wake the PC from the suspend state. This is done by setting the option "Allow this device to bring the computer out of standby".



macOS

macOS supports remote wakeup for all device classes.

4.2.4.1 USBD_SetAllowRemoteWakeUp()

Description

Allows the device to publish that remote wake is available.

Prototype

```
void USBD_SetAllowRemoteWakeUp(U8 AllowRemoteWakeup);
```

Parameters

Parameter	Description
AllowRemoteWakeup	<ul style="list-style-type: none">• 1 - Allows and publishes that remote wakeup is available.• 0 - Publish that remote wakeup is not available.

Additional information

This function must be called before the function `USBD_Start()` is called. This ensures that the Host is informed that USB remote wake up is available.

4.2.5 Data structures

4.2.5.1 USB_ADD_EP_INFO

Description

Structure used by `USBD_AddEPEx()` when adding an endpoint.

Type definition

```
typedef struct {
    unsigned MaxPacketSize;
    U16      Interval;
    U8       Flags;
    U8       InDir;
    U8       TransferType;
    U8       ISO_Type;
} USB_ADD_EP_INFO;
```

Structure members

Member	Description
MaxPacketSize	Maximum packet size for the endpoint.
Interval	Specifies the interval measured in units of 125us (microframes). This value should be zero for a bulk endpoint.
Flags	Specifies whether optional parameters are used. <ul style="list-style-type: none"> 0x00 - Ignore optional parameters. USB_ADD_EP_FLAG_USE_ISO_SYNC_TYPES - Use ISO_Type. If not set the endpoint will have the sync type <code>USB_ISO_SYNC_TYPE_NONE</code>.
InDir	Specifies the direction of the desired endpoint. <ul style="list-style-type: none"> USB_DIR_IN USB_DIR_OUT
TransferType	Specifies the transfer type of the endpoint. The following values are allowed: <ul style="list-style-type: none"> USB_TRANSFER_TYPE_BULK USB_TRANSFER_TYPE_ISO USB_TRANSFER_TYPE_INT
ISO_Type	Allows to set the synchronization type for isochronous endpoints. The following types are supported: <ul style="list-style-type: none"> USB_ISO_SYNC_TYPE_NONE (default) USB_ISO_SYNC_TYPE_ASYNCHRONOUS USB_ISO_SYNC_TYPE_ADAPTIVE USB_ISO_SYNC_TYPE_SYNCHRONOUS

Additional information

The [Interval](#) parameter specifies the frequency in which the endpoint should be polled for information by the host. It must be specified in units of 125 us. Depending on the actual speed of the device during enumeration, the USB stack converts the interval to the correct value required for the endpoint descriptor according to the USB specification (into milliseconds for low/full-speed, into 125 us for high-speed). For endpoints of type `USB_TRANSFER_TYPE_BULK` the value is ignored and should be set to 0.

The actual maximum packet size for bulk endpoints may be smaller than given in the [MaxPacketSize](#) field to meet the requirements of the actual USB speed.

For SuperSpeed bulk endpoints, [MaxPacketSize](#) can be $N * 1024$, where $N = 1...16$. Values of $N > 1$ enables the usage of burst transfers.

4.2.5.2 USB_SETUP_PACKET

Description

Structure containing a USB setup packet received from the host.

Type definition

```
typedef struct {
    U8  bmRequestType;
    U8  bRequest;
    U8  wValueLow;
    U8  wValueHigh;
    U8  wIndexLow;
    U8  wIndexHigh;
    U8  wLengthLow;
    U8  wLengthHigh;
} USB_SETUP_PACKET;
```

Structure members

Member	Description
bmRequestType	Setup request type.
bRequest	Setup request number.
wValueLow	Low byte of the value field.
wValueHigh	High byte of the value field.
wIndexLow	Low byte of the index field.
wIndexHigh	High byte of the index field.
wLengthLow	Low byte of the length field.
wLengthHigh	High byte of the length field.

4.2.5.3 SEGGER_CACHE_CONFIG

Description

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

Prototype

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

Member	Description
CacheLineSize	Cache line size of the CPU in bytes. Most Systems such as ARM9 use a 32 bytes cache line size.
pfDMB	Unused.
pfClean	Pointer to a callback function that executes a clean operation on cached memory. The parameter 'p' is always cache aligned. 'Num-Bytes' must be rounded up by the function to the next multiple of the cache line size, if necessary.
pfInvalidate	Pointer to a callback function that executes an invalidate operation on cached memory. The parameter 'p' is always cache aligned. 'Num-Bytes' must be rounded up by the function to the next multiple of the cache line size, if necessary.

Additional information

For further information about how this structure is used please refer to *USBD_SetCacheConfig* on page 84.

4.2.5.4 USB_CHECK_ADDRESS_FUNC

Description

Checks if an address can be used for DMA transfers. The function must return 0, if DMA access is allowed for the given address, 1 otherwise.

Type definition

```
typedef int USB_CHECK_ADDRESS_FUNC(const void * pMem);
```

Parameters

Parameter	Description
<code>pMem</code>	Pointer to the memory.

Return value

- 0 - Memory can be used for DMA access.
- 1 - DMA access not allowed for the given address.

4.2.5.5 USB_ASYNC_IO_CONTEXT

Description

Contains information for asynchronous transfers.

Type definition

```
typedef struct {
    unsigned          NumBytesToTransfer;
    void              * pData;
    USB_ASYNC_CALLBACK_FUNC * pfOnComplete;
    void              * pContext;
    int               Status;
    unsigned          NumBytesTransferred;
} USB_ASYNC_IO_CONTEXT;
```

Structure members

Member	Description
<code>NumBytesToTransfer</code>	Number of bytes to transfer. Must be set by the application.
<code>pData</code>	Pointer to the buffer for read operations, pointer to the data for write operations. Must be set by the application.
<code>pfOnComplete</code>	Pointer to the function called on completion of the transfer. Must be set by the application.
<code>pContext</code>	Pointer to a user context. Can be arbitrarily used by the application.
<code>Status</code>	Result status of the asynchronous transfer. Set by the USB stack before calling <code>pfOnComplete</code> .
<code>NumBytesTransferred</code>	Number of bytes transferred. Set by the USB stack before calling <code>pfOnComplete</code> .

4.2.5.6 USB_WEBUSB_INFO

Description

Information that may be provided by the application for WebUSB capable USB devices. Can be set via the function `USBD_SetWebUSBInfo()` before the USB stack is started using `USBD_Start()`. Is used during enumeration of the device by the host.

Type definition

```
typedef struct {
    U8          VendorCode;
    U8          DescIndex;
    U8          URLPrefix;
    const char * sURL;
} USB_WEBUSB_INFO;
```

Structure members

Member	Description
<code>VendorCode</code>	Vendor code used for the setup request.
<code>DescIndex</code>	Descriptor index of the descriptor containing the URL of the landing page.
<code>URLPrefix</code>	Prefix of the URL: 0 = "http://", 1 = "https://", 255 = none.
<code>sURL</code>	URL of the landing page. UTF-8 string.

4.3 Timeout handling

Many API functions have a timeout parameter that causes the functions to return, if the desired transaction can not be finished within the given time. Hardware USB controllers usually do not have a mechanism for timeouts. Therefore the USB stack has to handle timeouts as follows:

- Start a transaction.
- Wait for the transaction to complete or the timeout to expire.
- If the timeout has expired: **Abort** the current transaction.

Aborting a transaction is always a critical operation. The USB software is informed by the hardware only if a transaction has been completed. The software usually does not know, if a data transfer on the USB lines is still in progress. So if the USB stack decides to abort a transaction, this transaction may already be in progress at that time. In this case the abort of the transfer may cause the data currently transferred to be discarded without any notice to the software. Although the data packet was successfully transferred on the USB bus and acknowledged by the host, the data is lost from the target application's viewpoint.

Because this is usually not the behavior intended by the application, timeouts should be used to handle fatal errors only. Timeouts should not be used to repeatedly poll for data.

Bad example

NOT RECOMMENDED

```
for (;;) {
    // Try to read some data with 5 ms timeout
    NumBytesRead = USBD_Receive(EP, Buffer, 100, 5);
    if (NumBytesRead < 0) {
        <handle error>
        break;
    }
    if (NumBytesRead > 0) {
        <process the data>
        continue;
    }
    // NumBytesRead is 0 here, that means a timeout has occurred
    <execute other tasks>
    // Repeat the loop and retry to read data
}
```

In this example, data packets may be lost if they arrive exactly when the 5 ms timeout expires.

There are several options to avoid this problem:

- Using non-blocking API functions, like `USB_D_Receive()` whereas the Timeout value = -1 eg. `USB_D_Receive(EP, Buffer, 100, -1)`.
- Using asynchronous API functions, line `USB_D_ReadOverlapped()`, `USB_D_ReadAsync()`.
- Using blocking API functions with a timeout, that don't abort the transaction. They usually have a "Poll" in their name. The above example works well when using the function `USB_D_ReceivePoll()` instead of `USB_D_Receive()`.

The same applies when writing data to the host.

Chapter 5

Bulk communication

This chapter describes how to get emUSB-Device-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 Requirements for the Host (PC)

In order to communicate with a target (client) running emUSB-Device, the operating system running on the host must recognize the device connected to it.

5.2.1 Windows

Microsoft's Windows operating systems (Starting with XP Service Pack 2) contains a generic driver called WinUSB.sys that is used to handle all communication to a emUSB-Device running a BULK interface. If a emUSB bulk device is connected to a Windows 8, 8.1 and 10 PC for the first time, Windows will install the WinUSB driver automatically. For Windows versions less than Windows 8, Microsoft provides a driver for Windows Vista and Windows 7 but this needs to be installed manually. A driver installation tool including the mentioned driver is available in the `Windows\USB\Bulk\WinUSBInstall`. Windows XP user can use the driver package located under `Windows\USB\Bulk\WinUSB_USBBulk_XP`. In order to get emUSB BULK running with the WinUSB driver the following must be considered:

- The function `USBD_BULK_SetMSDescInfo()` must be called in the target application.
- The Product IDs 1234 and 1121 must not be used.

5.2.2 Linux

Linux can handle emUSB BULK devices out of the box.

By default a USB device can only be accessed by a process that is running with "root" rights. In order to use the USB bulk device from normal user programs an udev rule has to be configured for the device (refer to the linux udev documentation). The emUSB-Device release contains a sample configuration file `99-emUSBD.rules`, which may be modified and copied to `/etc/udev/rules.d` on the host machine.

5.2.3 macOS

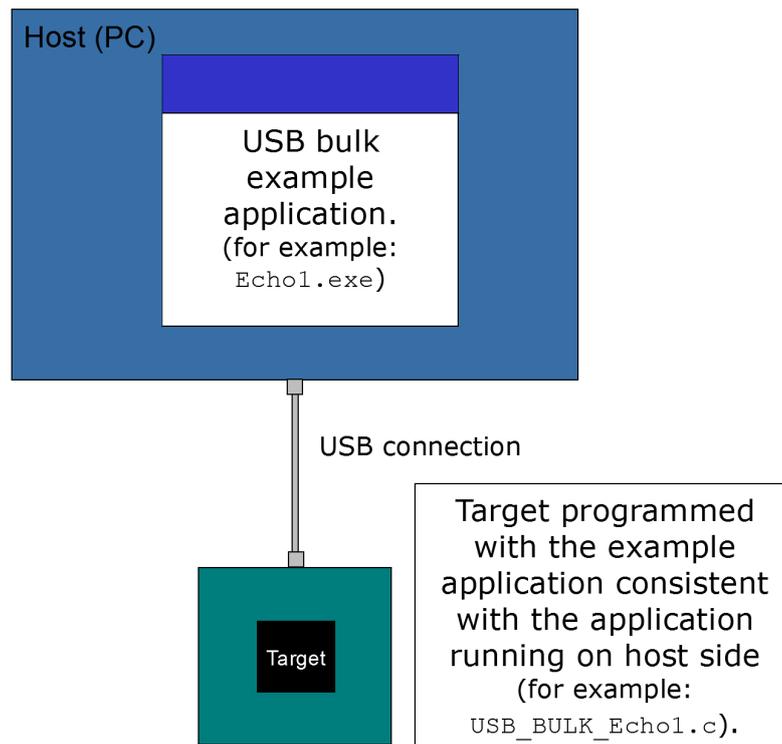
macOS can handle emUSB BULK devices out of the box.

5.3 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-Device.

The host sample applications can be used for Windows, Linux and MacOSX. Precompiled executables for Windows can be found in the subfolder `Windows/USB/Bulk/WindowsApplication/Exe`.

For example: The application `USB_BULK_Echo1.c` 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, 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 `USB_BULK_`, for example, if the host runs `Echo1.exe`, `USB_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-Device.

The following start application files are provided:

File	Description
<code>USB_BULK_Echo1.c</code>	This application was described in the upper text.
<code>USB_BULK_Test.c</code>	This application can be used to test emUSB-Device-Bulk with different packet sizes received from and sent to the PC host.
<code>USB_BULK_Performance.c</code>	Measures BULK data throughput.

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 device, use one of the following example applications:

File	Description
Echo1.exe	If the <code>USB_BULK_Echo1.c</code> sample application is running on the emUSB-Device-Bulk device, use this application.
Test.exe	If the <code>USB_BULK_Test.c</code> application is running on the emUSB-Device-Bulk device, use this application to test the emUSB-Device-Bulk stack.
Performance.exe	If the <code>USB_BULK_Performance.c</code> application is running on the emUSB-Device-Bulk device, use this application to measures BULK data throughput.

For information how to compile the host examples (especially for Linux and MacOSX) refer to *Compiling the PC example application* on page 119.

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

5.3.1 Running the example applications

To test the emUSB-Device-Bulk component, build and download the application of choice for the target-side.

To run one of the example applications, simply start the executable, for example by double clicking it.

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

Example output of Echo1.exe:

```

USB BULK Sample Echo1
USB BULK driver version: 2.70f, compiled: Oct 8 2014 16:40:43

Found 1 device
Found the following device 0:
  Vendor Name : Uendor
  Product Name: Bulk device
  Serial no.  : 13245678
To which device do you want to connect?
Please type in device number (e.g. '0' for the first device, q/a for abort):0
Starting Echo...
USB BULK driver version: 2.70f, compiled: Oct 8 2014 16:40:43
Enter the number of bytes to be send to the echo client: 100
.....
100 bytes successfully transferred.

```

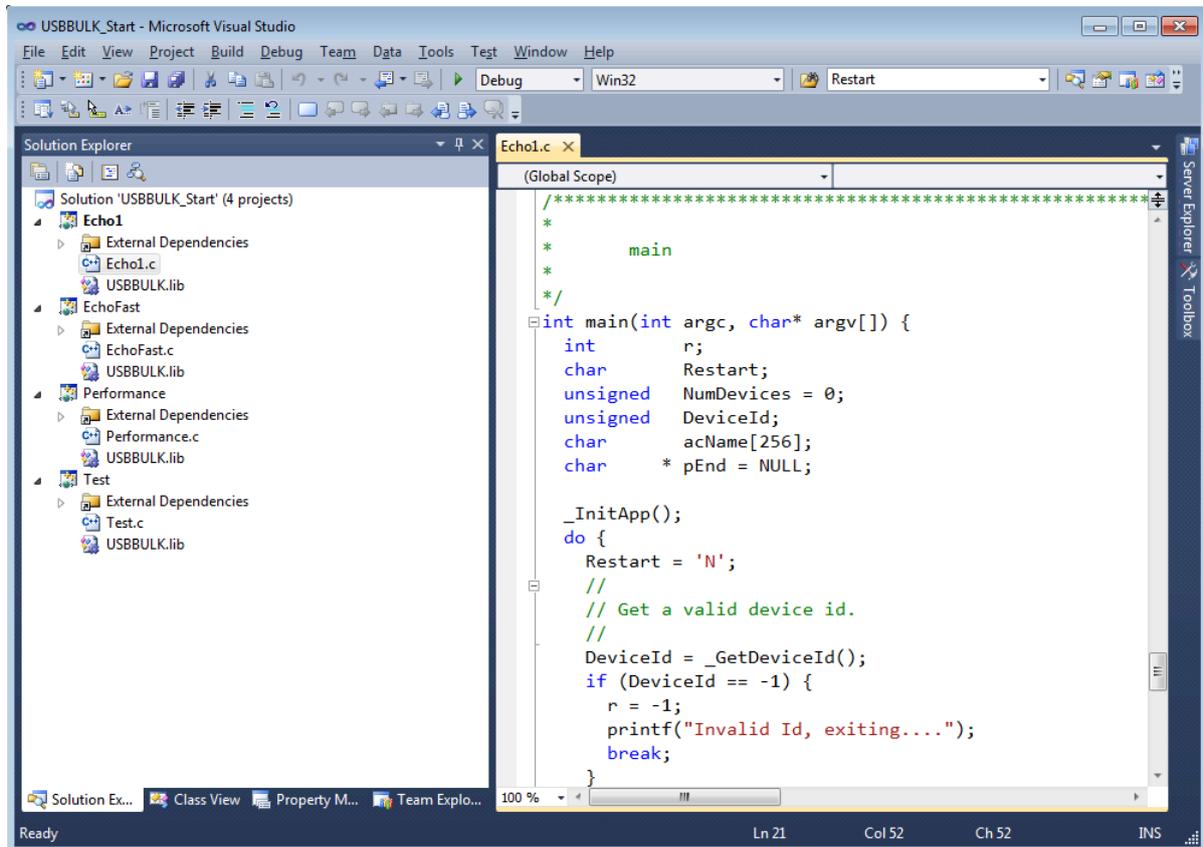
If the host example application can communicate with the emUSB-Device device, the example application will be in interactive mode for the Echo1 and the EchoFast application. In case of an error, a message box is displayed.

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.
Could not read from device (time-out)	The PC sample application was not able to read.
Wrong data read	The result of the target sample application is not correct.

5.3.2 Compiling the PC example application

5.3.2.1 Windows

For compiling the example application you need Visual C++ 2010 (or later).



The source code of the sample application is located in the subfolder `Windows\USB\Bulk\WindowsApplication`. Open the file `USBULK_Start.sln` and compile the source.

5.3.2.2 Linux

The subfolder `Windows\USB\Bulk\WindowsApplication` contains a Makefile for Linux. Change to this folder and execute "make".

5.3.2.3 macOS

The subfolder `Windows\USB\Bulk\WindowsApplication` contains a Makefile for macOS. Change to this folder and execute "make -f Makefile_MacOSX".

5.4 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_Bulk.h`). These files contain API functions to communicate with the USB host through the emUSB-Device 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 hosts kernel.

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

5.4.1 Target interface function list

Routine	Explanation
USB-Bulk functions	
<code>USBD_BULK_Add()</code>	Adds interface for USB-Bulk communication to emUSB-Device.
<code>USBD_BULK_Add_Ex()</code>	Adds interface for USB-Bulk communication to emUSB-Device.
<code>USBD_BULK_AddAlternateInterface()</code>	Adds an alternative interface for USB-Bulk interface.
<code>USBD_BULK_SetMSDescInfo()</code>	Enables use of Microsoft OS Descriptors.
<code>USBD_BULK_CancelRead()</code>	Cancels any non-blocking/blocking read operation that is pending.
<code>USBD_BULK_CancelWrite()</code>	Cancels any non-blocking/blocking write operation that is pending.
<code>USBD_BULK_GetNumBytesInBuffer()</code>	Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer.
<code>USBD_BULK_GetNumBytesRemToRead()</code>	Get the number of remaining bytes to read by an active read operation.
<code>USBD_BULK_GetNumBytesRemToWrite()</code>	After starting a non-blocking write operation this function can be used to periodically check how many bytes still have to be written.
<code>USBD_BULK_Read()</code>	Reads data from the host with a given timeout.
<code>USBD_BULK_ReadAsync()</code>	Reads data from the host asynchronously.
<code>USBD_BULK_ReadOverlapped()</code>	Reads data from the host asynchronously.
<code>USBD_BULK_Receive()</code>	Reads data from the host.
<code>USBD_BULK_ReceivePoll()</code>	Reads data from the host.
<code>USBD_BULK_SetContinuousReadMode()</code>	Enables continuous read mode for the RX endpoint.
<code>USBD_BULK_SetOnSetupRequest()</code>	Sets a callback function that is called when any setup request is sent from the host.

Routine	Explanation
<code>USBD_BULK_SetOnRXEvent()</code>	Sets a callback function for the OUT endpoint that will be called on every RX event for that endpoint.
<code>USBD_BULK_SetOnTXEvent()</code>	Sets a callback function for the IN endpoint that will be called on every TX event for that endpoint.
<code>USBD_BULK_TxIsPending()</code>	Checks whether the TX (IN endpoint) is currently pending.
<code>USBD_BULK_WaitForRX()</code>	Waits (blocking) until the triggered <code>USBD_BULK_ReadOverlapped()</code> has received the desired data.
<code>USBD_BULK_PollForRX()</code>	Waits (blocking) until the triggered <code>USBD_BULK_ReadOverlapped()</code> has received the desired data.
<code>USBD_BULK_WaitForTX()</code>	Waits (blocking) until a pending asynchronous <code>USBD_BULK_Write()</code> (Timeout parameter = -1) has sent the desired data.
<code>USBD_BULK_PollForTX()</code>	Waits (blocking) until a pending asynchronous <code>USBD_BULK_Write()</code> (Timeout parameter = -1) has sent the desired data.
<code>USBD_BULK_WaitForTXReady()</code>	Waits (blocking) until the TX queue can accept another data packet.
<code>USBD_BULK_Write()</code>	Sends data to the USB host.
<code>USBD_BULK_WriteAsync()</code>	Sends data to the host asynchronously.
<code>USBD_BULK_WriteEx()</code>	Send data to the USB host with <code>NULL</code> packet control.

5.4.2 USB-Bulk functions

5.4.2.1 USBD_BULK_Add()

Description

Adds interface for USB-Bulk communication to emUSB-Device.

Prototype

```
USB_BULK_HANDLE USBD_BULK_Add(const USB_BULK_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
<code>pInitData</code>	Pointer to USB_BULK_INIT_DATA structure.

Return value

Handle to a valid BULK instance. The handle of the first BULK instance is always 0.

Example

Example excerpt from BULK_Echo1.c:

```
static void _AddBULK(void) {
    static U8 _abOutBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
    USB_BULK_INIT_DATA Init;
    Init.EPIn = USBD_AddeP(USB_DIR_IN,
                          USB_TRANSFER_TYPE_BULK,
                          USB_HS_BULK_MAX_PACKET_SIZE,
                          NULL,
                          0);
    Init.EPOut = USBD_AddeP(USB_DIR_OUT,
                           USB_TRANSFER_TYPE_BULK,
                           USB_HS_BULK_MAX_PACKET_SIZE,
                           _abOutBuffer,
                           USB_HS_BULK_MAX_PACKET_SIZE);
    USBD_BULK_Add(&Init);
}
```

5.4.2.2 USBD_BULK_Add_Ex()

Description

Adds interface for USB-Bulk communication to emUSB-Device.

Prototype

```
USB_BULK_HANDLE USBD_BULK_Add_Ex(const USB_BULK_INIT_DATA_EX * pInitData);
```

Parameters

Parameter	Description
<code>pInitData</code>	Pointer to <code>USB_BULK_INIT_DATA_EX</code> structure.

Return value

Handle to a valid BULK instance. The handle of the first BULK instance is always 0.

Example

```
static void _AddBULK(void) {
    static U8 _abOutBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
    USB_BULK_INIT_DATA_EX Init;
    Init.Flags = 0;
    Init.EPIn = USBD_AddeP(USB_DIR_IN,
                          USB_TRANSFER_TYPE_BULK,
                          USB_HS_BULK_MAX_PACKET_SIZE,
                          NULL,
                          0);
    Init.EPOut = USBD_AddeP(USB_DIR_OUT,
                           USB_TRANSFER_TYPE_BULK,
                           USB_HS_BULK_MAX_PACKET_SIZE,
                           _abOutBuffer,
                           USB_HS_BULK_MAX_PACKET_SIZE);
    Init.pInterfaceName = "BULK Interface";
    USBD_BULK_Add_Ex(&Init);
}
```

5.4.2.3 USBD_BULK_AddAlternateInterface()

Description

Adds an alternative interface for USB-Bulk interface.

Prototype

```
void USBD_BULK_AddAlternateInterface(      USB_BULK_HANDLE      hInst,
                                           const USB_BULK_INIT_DATA_EX * pInitData,
                                           USB_ON_USER_SET_INTERFACE * pfOnUser);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_Add()</code> .
<code>pInitData</code>	Pointer to <code>USB_BULK_INIT_DATA_EX</code> structure.
<code>pfOnUser</code>	Callback function that is called, when the host changes the interface.

5.4.2.4 USBD_BULK_SetMSDescInfo()

Description

Enables use of Microsoft OS Descriptors. A USB bulk device providing these descriptors is detected by Windows to be handled by the generic WinUSB driver. For such devices no other driver needs to be installed.

Prototype

```
void USBD_BULK_SetMSDescInfo(USB_BULK_HANDLE hInst);
```

Parameters

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

Additional information

This function must be called after the call to the function `USB_BULK_Add()` and before `USB_Start()`.

5.4.2.5 USBD_BULK_CancelRead()

Description

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

Prototype

```
void USBD_BULK_CancelRead(USB_BULK_HANDLE hInst);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_Ad-</code> <code>d()</code> .

Additional information

This function shall 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.

5.4.2.6 USBD_BULK_CancelWrite()

Description

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

Prototype

```
void USBD_BULK_CancelWrite(USB_BULK_HANDLE hInst);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_Ad-</code> <code>d()</code> .

Additional information

This function shall be called when a pending asynchronous write 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.

5.4.2.7 USBD_BULK_GetNumBytesInBuffer()

Description

Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer. This functions does not start a read transfer.

Prototype

```
unsigned USBD_BULK_GetNumBytesInBuffer(USB_BULK_HANDLE hInst);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_Ad-</code> <code>d()</code> .

Return value

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

Additional information

If the host is sending more data than your target application has requested, the remaining data will be stored in an internal buffer. This function shows how many bytes are available in this buffer.

The number of bytes returned by this function can be read using `USB_BULK_Read()` without blocking.

Example

Your host application sends 50 bytes. Your target application only requests to receive 1 byte. In this case the target application will get 1 byte and the remaining 49 bytes are stored in an internal buffer. When your target application now calls `USB_BULK_GetNumBytesInBuffer()` it will return the number of bytes that are available in the internal buffer (49).

5.4.2.8 USBD_BULK_GetNumBytesRemToRead()

Description

Get the number of remaining bytes to read by an active read operation. This function is to be used in combination with `USBD_BULK_ReadOverlapped()`. After starting the read operation this function can be used to periodically check how many bytes still have to be read.

Prototype

```
unsigned USBD_BULK_GetNumBytesRemToRead(USB_BULK_HANDLE hInst);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USBD_BULK_Ad-</code> <code>d()</code> .

Return value

≥ 0 Number of bytes which have not yet been read.
 < 0 Error occurred.

Additional information

Alternatively the blocking function `USBD_BULK_WaitForRX()` can be used.

Example

```
NumBytesReceived = USBD_BULK_ReadOverlapped(hInst, &ac[0], 50);
if (NumBytesReceived < 0) {
    <.. error handling..>
}
if (NumBytesReceived > 0) {
    // Already had some data in the internal buffer.
    // The first 'NumBytesReceived' bytes may be processed here.
    <...>
} else {
    // Wait until we get all 50 bytes
    while (USBD_BULK_GetNumBytesRemToRead(hInst) > 0) {
        USB_OS_Delay(50);
    }
}
```

5.4.2.9 USBD_BULK_GetNumBytesRemToWrite()

Description

After starting a non-blocking write operation this function can be used to periodically check how many bytes still have to be written.

Prototype

```
unsigned USBD_BULK_GetNumBytesRemToWrite(USB_BULK_HANDLE hInst);
```

Parameters

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

Return value

Number of bytes which have not yet been written.

Additional information

Alternatively the blocking function `USB_BULK_WaitForTX()` can be used.

Example

```
r = USBD_BULK_Write(hInst, &ac[0], TRANSFER_SIZE, -1);
if (r < 0) {
    <.. error handling..>
}
// NumBytesToWrite shows how many bytes still have to be written.
while (USB_BULK_GetNumBytesRemToWrite(hInst) > 0) {
    USB_OS_Delay(50);
}
```

5.4.2.10 USBD_BULK_Read()

Description

Reads data from the host with a given timeout.

Prototype

```
int USBD_BULK_Read(USB_BULK_HANDLE hInst,
                  void * pData,
                  unsigned NumBytes,
                  unsigned Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_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>Timeout</code>	<code>Timeout</code> in milliseconds, 0 means infinite.

Return value

= `NumBytes` Requested data was successfully read within the given timeout.
 ≥ 0 && < `NumBytes` `Timeout` has occurred. Number of bytes that have been read within the given timeout.
 < 0 Error occurred.

Additional information

This function blocks a task until all data have been read or a timeout expires. This function also returns when the device is disconnected from host or when a USB reset occurs.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USB_BULK_AddEP()`. This data can be retrieved by a later call to `USB_BULK_Receive()` / `USB_BULK_Read()`. See also `USB_BULK_GetNumBytesInBuffer()`.

In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 113).

If a read transfer was still pending while the function is called, it returns `USB_STATUS_EP_BUSY`.

5.4.2.11 USBD_BULK_ReadAsync()

Description

Reads data from the host asynchronously. The function does not wait for the data to be received. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_BULK_ReadAsync(USB_BULK_HANDLE      hInst,
                        USB_ASYNC_IO_CONTEXT * pContext,
                        int                    ShortRead);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_Add()</code> .
<code>pContext</code>	Pointer to a structure of type <code>USB_ASYNC_IO_CONTEXT</code> containing parameters and a pointer to the callback function.
<code>ShortRead</code>	<ul style="list-style-type: none"> 0: The transfer is completed successfully after all bytes have been read. 1: The transfer is completed successfully after one packet has been read.

Example

```
static void _AsyncCb(USB_ASYNC_IO_CONTEXT * pIOContext) {
    U8 *p;

    p = (U8 *)pIOContext->pContext;
    *p = 1;
}

<...>

USB_ASYNC_IO_CONTEXT IOContext;
U8 AsyncComplete;

IOContext.NumBytesToTransfer = 5000;
IOContext.pData              = pBuffer;
IOContext.pfOnComplete       = _AsyncCb;
IOContext.pContext           = (void *)&AsyncComplete;
AsyncComplete = 0;
USB_BULK_ReadAsync(hInst, &IOContext, 0);
while (AsyncComplete == 0) {
    <.. Do other work. ..>
}
// Transaction is complete.
if (IOContext.Status < 0 || IOContext.NumBytesTransferred != 5000) {
    <.. error handling ..>
} else {
    <.. Process the data ..>
}
<...>
```

5.4.2.12 USBD_BULK_ReadOverlapped()

Description

Reads data from the host asynchronously.

Prototype

```
int USBD_BULK_ReadOverlapped(USB_BULK_HANDLE hInst,
                             void * pData,
                             unsigned NumBytes);
```

Parameters

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

Return value

- > 0 Number of bytes that have been read from the internal buffer (success).
- = 0 No data was found in the internal buffer, read transfer started (success).
- < 0 Error occurred.

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. Alternatively the function `USB_BULK_GetNumBytesRemToRead()` can be called periodically to check whether all bytes have been read or not. The read operation can be canceled using `USB_BULK_CancelRead()`. The buffer pointed to by `pData` must be valid until the read operation is terminated.

If a read transfer was still pending while the function is called, it returns `USB_STATUS_EP_BUSY`.

Example

See `USB_BULK_GetNumBytesRemToRead` on page 129.

5.4.2.13 USBD_BULK_Receive()

Description

Reads data from the host. The function blocks until any data has been received or a timeout occurs (if `Timeout` \geq 0). In contrast to `USBD_BULK_Read()` this function does not wait for all of `NumBytes` to be received, but returns after the first packet has been received. In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 113).

Prototype

```
int USBD_BULK_Receive(USB_BULK_HANDLE hInst,
                    void * pData,
                    unsigned NumBytes,
                    int Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USBD_BULK_Add()</code> .
<code>pData</code>	Pointer to a buffer where the received data will be stored.
<code>NumBytes</code>	Maximum number of bytes to read.
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite. If <code>Timeout</code> is -1, the function never blocks and only reads data from the internal endpoint buffer.

Return value

- > 0 Number of bytes that have been read.
- = 0 A timeout occurred (if `Timeout` > 0), zero packet received (not every controller supports this!), no data in buffer (if `Timeout` < 0) or the target was disconnected during the function call and no data was read so far.
- < 0 Error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling `USB_D_BULK_Receive()` will return as much data as is currently available -- up to the size of the buffer specified. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return `USB_STATUS_ERROR`.

If a read transfer was pending while the function is called, it returns `USB_STATUS_EP_BUSY`.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USBD_AddEP()`. This data can be retrieved by a later call to `USBD_BULK_Receive()` / `USBD_BULK_Read()`. See also `USBD_BULK_GetNumBytesInBuffer()`.

A call of `USBD_BULK_Receive(Inst, NULL, 0, -1)` can be used to trigger an asynchronous read that stores the data into the internal buffer.

5.4.2.14 USBD_BULK_ReceivePoll()

Description

Reads data from the host. The function blocks until any data has been received or a timeout occurs (if `Timeout` \geq 0). In contrast to `USBD_BULK_Read()` this function does not wait for all of `NumBytes` to be received, but returns after the first packet has been received. In contrast to `USBD_BULK_Receive()` this function will continue the read transfer asynchronously in case of a timeout.

Prototype

```
int USBD_BULK_ReceivePoll(USB_BULK_HANDLE hInst,
                          void * pData,
                          unsigned NumBytes,
                          unsigned Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USBD_BULK_Add()</code> .
<code>pData</code>	Pointer to a buffer where the received data will be stored.
<code>NumBytes</code>	Maximum number of bytes to read.
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite.

Return value

- > 0 Number of bytes that have been read.
- = 0 A timeout occurred (if `Timeout` > 0), zero packet received (not every controller supports this!) or the target was disconnected during the function call and no data was read so far.
- < 0 Error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling `USB_D_BULK_ReceivePoll()` will return as much data as is currently available -- up to the size of the buffer specified. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return `USB_STATUS_ERROR`.

If a read transfer was pending while the function is called, it returns `USB_STATUS_EP_BUSY`.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USBD_AddEP()`. This data can be retrieved by a later call to `USBD_BULK_Receive()` / `USBD_BULK_Read()`. See also `USBD_BULK_GetNumBytesInBuffer()`.

If a timeout occurs, the read transfer is not affected. Data send from the host after the timeout is stored into the internal buffer of the endpoint and can be read by later calls to `USBD_BULK_ReceivePoll()`.

If `Timeout` = 0, the function behaves like `USBD_BULK_Receive()`.

5.4.2.15 USBD_BULK_SetContinuousReadMode()

Description

Enables continuous read mode for the RX endpoint. In this mode every finished read transfer will automatically trigger another read transfer, as long as there is enough space in the internal buffer to receive another packet.

Prototype

```
void USBD_BULK_SetContinuousReadMode(USB_BULK_HANDLE hInst);
```

Parameters

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

Additional information

To check how many bytes have been read into the buffer, the function `USB_BULK_GetNumBytesInBuffer()` may be called. In order to read the data the function `USB_BULK_Receive()` needs to be called (non-blocking).

The USB stack will use the buffer that was provided by the application with `USB_AddEP()`. The transfer speed may be improved, if this buffer has a size of at least $2 * \text{MaxPacketSize}$. Normally `MaxPacketSize` for full-speed devices is 64 bytes and for high-speed devices 512 bytes.

Example

```
USB_BULK_SetContinuousReadMode(hInst);
<...>
for(;;) {
    //
    // Fetch data that was already read (non-blocking).
    //
    NumBytesReceived = USB_BULK_Receive(hInst, &ac[0], sizeof(ac), -1);
    if (NumBytesReceived > 0) {
        //
        // We got some data
        //
        <.. Process data..>
    } else {
        <.. Nothing received yet, do application processing..>
    }
}
```

5.4.2.16 USBD_BULK_SetOnSetupRequest()

Description

Sets a callback function that is called when any setup request is sent from the host.

Prototype

```
void USBD_BULK_SetOnSetupRequest(USB_BULK_HANDLE hInst,  
                                USB_ON_SETUP * pfOnSetupRequest);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by <code>USB_BULK_Add()</code> .
pfOnSetupRequest	Pointer to the callback function.

5.4.2.17 USBD_BULK_SetOnRXEvent()

Description

Sets a callback function for the OUT endpoint that will be called on every RX event for that endpoint.

Prototype

```
void USBD_BULK_SetOnRXEvent(USB_BULK_HANDLE      hInst,
                             USB_EVENT_CALLBACK   * pEventCb,
                             USB_EVENT_CALLBACK_FUNC * pfEventCb,
                             void                * pContext);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_Add()</code> .
<code>pEventCb</code>	Pointer to a <code>USB_EVENT_CALLBACK</code> structure. The structure is initialized by this function.
<code>pfEventCb</code>	Pointer to the callback routine that will be called on every event on the USB endpoint.
<code>pContext</code>	A pointer which is used as parameter for the callback function.

Additional information

The `USB_EVENT_CALLBACK` structure is private to the USB stack. It will be initialized by `USB_BULK_SetOnRXEvent()`. The USB stack keeps track of all event callback functions using a linked list. The `USB_EVENT_CALLBACK` structure will be included into this linked list and must reside in static memory.

The callback function is called only, if a read operation was started using one of the `USB_BULK_Read...()` functions.

The callback function has the following prototype:

```
typedef void USB_EVENT_CALLBACK_FUNC(unsigned Events, void *pContext);
```

Parameter	Description
<code>Events</code>	A bit mask indicating which events occurred on the endpoint.
<code>pContext</code>	The pointer which was provided to the <code>USB_SetOnEvent()</code> function.

Note that the callback function will be called within an ISR, therefore it should never block. The first parameter to the callback function will contain a bit mask for all events that triggered the call:

Event	Description
<code>USB_EVENT_DATA_READ</code>	Some data was received from the host on the endpoint.
<code>USB_EVENT_READ_COMPLETE</code>	The last read operation was completed.
<code>USB_EVENT_READ_ABORT</code>	A read transfer was aborted.

Example

```
// The callback function.
static void _OnEvent(unsigned Events, void *pContext) {
    unsigned NumBytes;

    if (Events & USB_EVENT_DATA_READ) {
        NumBytes = USBD_BULK_GetNumBytesInBuffer(hInst);
        if (NumBytes) {
            r = USBD_BULK_Receive(hInst, Buff, NumBytes, -1);
            if (r > 0) {
                <.. process data in Buff..>
            }
        }
    }
}

// Main program.
// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USB_D_BULK_SetOnRXEvent(hInst, &_usb_callback, _OnEvent, NULL);
USB_D_BULK_SetContinuousReadMode(hInst);
// Trigger first read
USB_D_BULK_Receive(Inst, NULL, 0, -1);
<.. do anything else here while the data is processed in the callback ..>
```

5.4.2.18 USBD_BULK_SetOnTXEvent()

Description

Sets a callback function for the IN endpoint that will be called on every TX event for that endpoint.

Prototype

```
void USBD_BULK_SetOnTXEvent(USB_BULK_HANDLE      hInst,
                           USB_EVENT_CALLBACK    * pEventCb,
                           USB_EVENT_CALLBACK_FUNC * pfEventCb,
                           void                  * pContext);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_Add()</code> .
<code>pEventCb</code>	Pointer to a <code>USB_EVENT_CALLBACK</code> structure. The structure is initialized by this function.
<code>pfEventCb</code>	Pointer to the callback routine that will be called on every event on the USB endpoint.
<code>pContext</code>	A pointer which is used as parameter for the callback function.

Additional information

The `USB_EVENT_CALLBACK` structure is private to the USB stack. It will be initialized by `USB_BULK_SetOnTXEvent()`. The USB stack keeps track of all event callback functions using a linked list. The `USB_EVENT_CALLBACK` structure will be included into this linked list and must reside in static memory.

The callback function is called only, if a write operation was started using one of the `USB_BULK_Write...()` functions.

The callback function has the following prototype:

```
typedef void USB_EVENT_CALLBACK_FUNC(unsigned Events, void *pContext);
```

Parameter	Description
<code>Events</code>	A bit mask indicating which events occurred on the endpoint.
<code>pContext</code>	The pointer which was provided to the <code>USB_SetOnEvent()</code> function.

Note that the callback function will be called within an ISR, therefore it should never block. The first parameter to the callback function will contain a bit mask for all events that triggered the call:

Event	Description
<code>USB_EVENT_DATA_SEND</code>	Some data was sent to the host, so that (part of) the user write buffer may be reused by the application.
<code>USB_EVENT_DATA_ACKED</code>	Some data was acknowledged by the host.
<code>USB_EVENT_WRITE_ABORT</code>	A write transfer was aborted.
<code>USB_EVENT_WRITE_COMPLETE</code>	All write operations were completed.

Example

```
// The callback function.
static void _OnEvent(unsigned Events, void *pContext) {
    if ((Events & USB_EVENT_DATA_SEND) != 0 &&
        // Check for last write transfer to be completed.
        USBD_BULK_GetNumBytesRemToWrite(_hInst) == 0) {
        <.. prepare next data for writing..>
        // Send next packet of data.
        r = USBD_BULK_Write(_hInst, &ac[0], 200, -1);
        if (r < 0) {
            <.. error handling..>
        }
    }
}

// Main program.
// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USBD_BULK_SetOnTXEvent(hInst, &_usb_callback, _OnEvent, NULL);
// Send the first packet of data using an asynchronous write operation.
r = USBD_BULK_Write(_hInst, &ac[0], 200, -1);
if (r < 0) {
    <.. error handling..>
}
<.. do anything else here while the whole data is send..>
```

5.4.2.19 USBD_BULK_TxIsPending()

Description

Checks whether the TX (IN endpoint) is currently pending. Can be called in any context.

Prototype

```
int USBD_BULK_TxIsPending(USB_BULK_HANDLE hInst);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_Ad-</code> <code>d()</code> .

Return value

- 1 We have queued data to be sent.
- 0 Queue is empty.

5.4.2.20 USBD_BULK_WaitForRX()

Description

Waits (blocking) until the triggered `USBD_BULK_ReadOverlapped()` has received the desired data.

Prototype

```
int USBD_BULK_WaitForRX(USB_BULK_HANDLE hInst,
                       unsigned          Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_Add()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite.

Return value

= 0 Transfer completed.
 = 1 `Timeout` occurred.
 < 0 An error occurred (e.g. target disconnected)

Additional information

In case of a timeout, the read transfer is aborted (see [Timeout handling](#) on page 113).

Example

```
if (USB_BULK_ReadOverlapped(hInst, &ac[0], 50) < 0) {
    <.. error handling..>
    return;
}
//
// USB_BULK_ReadOverlapped() will return immediately.
// Do something else while data may be transferred.
//
<...>
//
// Now wait until we get all 50 bytes.
// USB_BULK_WaitForRX() will block, until total of
// 50 bytes are read or timeout occurs.
//
if (USB_BULK_WaitForRX(hInst, timeout) != 0) {
    <.. timeout error handling..>
    return;
}
// Now we have 50 bytes of data.
// Process 50 bytes of data from ac[] here.
```

5.4.2.21 USBD_BULK_PollForRX()

Description

Waits (blocking) until the triggered `USBD_BULK_ReadOverlapped()` has received the desired data.

Prototype

```
int USBD_BULK_PollForRX(USB_BULK_HANDLE hInst,
                       unsigned          Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_Add()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite.

Return value

= 0 Transfer completed.
 = 1 `Timeout` occurred.
 < 0 An error occurred (e.g. target disconnected)

Additional information

In case of a timeout, the current transfer is not affected. The function may be called repeatedly until it does not report a timeout any more.

Example

```
if (USB_BULK_ReadOverlapped(hInst, &ac[0], 50) < 0) {
    <.. error handling..>
    return;
}
//
// USB_BULK_ReadOverlapped() will return immediately.
// While waiting for the data, we will blink a LED with 200 ms interval.
// USB_BULK_PollForRX() will return, if all data were read or 100 ms expired.
//
while ((r = USB_BULK_PollForRX(hInst, 100)) > 0) {
    ToggleLED();
}
if (r < 0) {
    <.. error handling..>
    return;
}
// Now we have 50 bytes of data.
// Process 50 bytes of data from ac[] here.
```

5.4.2.22 USBD_BULK_WaitForTX()

Description

Waits (blocking) until a pending asynchronous `USBD_BULK_Write()` (`Timeout` parameter = -1) has sent the desired data.

Prototype

```
int USBD_BULK_WaitForTX(USB_BULK_HANDLE hInst,
                        unsigned         Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USBD_BULK_Ad-</code> <code>d()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite.

Return value

= 0 Transfer completed.
 = 1 `Timeout` occurred.
 < 0 An error occurred (e.g. target disconnected)

Additional information

In case of a timeout, the write transfer is aborted (see *Timeout handling* on page 113).

5.4.2.23 USBD_BULK_PollForTX()

Description

Waits (blocking) until a pending asynchronous `USBD_BULK_Write()` (`Timeout` parameter = -1) has sent the desired data.

Prototype

```
int USBD_BULK_PollForTX(USB_BULK_HANDLE hInst,
                       unsigned         Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_Add()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite.

Return value

= 0 Transfer completed.
 = 1 `Timeout` occurred.
 < 0 An error occurred (e.g. target disconnected)

Additional information

In case of a timeout, the current transfer is not affected. The function may be called repeatedly until it does not report a timeout any more.

Example

```
if (USB_BULK_Write(hInst, &ac[0], 50, -1) < 0) {
    <.. error handling..>
    return;
}
//
// USB_BULK_Write() will return immediately.
// While waiting for the data to be transferred, we will blink a LED with
// 200 ms interval.
// USB_BULK_PollForTX() will return, if all data were send or 100 ms expired.
//
while ((r = USB_BULK_PollForTX(hInst, 100)) > 0) {
    ToggleLED();
}
if (r < 0) {
    <.. error handling..>
    return;
}
// Now all data have been send.
```

5.4.2.24 USBD_BULK_WaitForTXReady()

Description

Waits (blocking) until the TX queue can accept another data packet. This function is used in combination with a non-blocking call to `USBD_BULK_Write()`, it waits until a new asynchronous write data transfer will be accepted by the USB stack.

Prototype

```
int USBD_BULK_WaitForTXReady(USB_BULK_HANDLE hInst,
                             int Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_Add()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite. If <code>Timeout</code> is negative, the function will return immediately.

Return value

- = 0 A new asynchronous write data transfer will be accepted.
- = 1 The write queue is full, a call to `USB_BULK_Write()` would return `USB_STATUS_EP_BUSY`.
- < 0 Error occurred.

Additional information

If `Timeout` is 0, the function never returns 1.

If `Timeout` is -1, the function will not wait, but immediately return the current state.

Example

```
// Always keep the write queue full for maximum send speed.
for (;;) {
    pData = GetNextData(&NumBytes);
    // Wait until stack can accept a new write.
    USBD_BULK_WaitForTxReady(hInst, 0);
    // Issue write transfer.
    if (USB_BULK_Write(hInst, pData, NumBytes, -1) < 0) {
        <.. error handling..>
    }
}
```

5.4.2.25 USBD_BULK_Write()

Description

Sends data to the USB host. Depending on the `Timeout` parameter, the function blocks until `NumBytes` have been written or a timeout occurs.

Prototype

```
int USBD_BULK_Write(      USB_BULK_HANDLE  hInst,
                        const void        * pData,
                        unsigned          NumBytes,
                        int                Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_Add()</code> .
<code>pData</code>	Data that should be written.
<code>NumBytes</code>	Number of bytes to write.
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite. If <code>Timeout</code> is -1, the function returns immediately and the transfer is processed asynchronously.

Return value

= 0 Successful started an asynchronous write transfer or a timeout has occurred and no data was written.
> 0 && < `NumBytes` Number of bytes that have been written before a timeout occurred.
= `NumBytes` Write transfer successful completed.
< 0 Error occurred.

Additional information

This function also returns when the target is disconnected from host or when a USB reset occurred.

The USB stack is able to queue a small number of asynchronous write transfers (when using `Timeout = -1`). If a write transfer is still in progress when this function is called and the USB stack can not accept another write transfer request, the functions returns `USB_STATUS_EP_BUSY`. A synchronous write transfer (`Timeout ≥ 0`) will always block until the transfer (including all pending transfers) are finished or a timeout occurs.

In case of a timeout, the write transfer is aborted (see *Timeout handling* on page 113).

In order to synchronize, `USB_BULK_WaitForTX()` needs to be called. Another synchronization method would be to periodically call `USB_BULK_GetNumBytesRemToWrite()` in order to see how many bytes still need to be written (this method is preferred when a non-blocking solution is necessary). The write operation can be canceled using `USB_BULK_CancelWrite()`.

If `pData = NULL` and `NumBytes = 0`, a zero-length packet is sent to the host.

The content of the buffer pointed to by `pData` must not be changed until the transfer has been completed.

Example

```
NumBytesWritten = USBD_BULK_Write(hInst, &ac[0], DataSize, 500);
if (NumBytesWritten <= 0) {
    <.. error handling..>
}
```

```
if (NumBytesWritten < DataSize) {  
    <.. timeout occurred, data partially written within 500ms ..>  
} else {  
    <.. write completed successfully..>  
}
```

See also *USBD_BULK_GetNumBytesRemToWrite* on page 130.

5.4.2.26 USBD_BULK_WriteAsync()

Description

Sends data to the host asynchronously. The function does not block. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_BULK_WriteAsync(USB_BULK_HANDLE      hInst,
                          USB_ASYNC_IO_CONTEXT * pContext,
                          char                  Send0PacketIfRequired);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USB_BULK_Add()</code> .
<code>pContext</code>	Pointer to a structure of type <code>USB_ASYNC_IO_CONTEXT</code> containing parameters and a pointer to the callback function.
<code>Send0PacketIfRequired</code>	Specifies that a zero-length packet shall be sent when the last data packet is a multiple of <code>MaxPacketSize</code> .

Example

```
static void _AsyncCb(USB_ASYNC_IO_CONTEXT * pIOContext) {
    U8 *p;

    p = (U8 *)pIOContext->pContext;
    *p = 1;
}

<...>

USB_ASYNC_IO_CONTEXT IOContext;
U8 AsyncComplete;

IOContext.NumBytesToTransfer = 5000;
IOContext.pData              = pBuffer;
IOContext.pfOnComplete       = _AsyncCb;
IOContext.pContext           = (void *)&AsyncComplete;
AsyncComplete = 0;
USB_BULK_WriteAsync(hInst, &IOContext, 1);
while (AsyncComplete == 0) {
    <.. Do other work. ..>
}
// Transaction is complete.
if (IOContext.Status < 0 || IOContext.NumBytesTransferred != 5000) {
    <.. error handling ..>
} else {
    <.. data written successfully ..>
}
<...>
```

5.4.2.27 USBD_BULK_WriteEx()

Description

Send data to the USB host with `NULL` packet control. This function behaves exactly like `USBD_BULK_Write()`. Additionally sending of a zero length packet after sending the data can be suppressed by setting `Send0PacketIfRequired = 0`.

Prototype

```
int USBD_BULK_WriteEx(    USB_BULK_HANDLE  hInst,
                        const void      * pData,
                        unsigned         NumBytes,
                        char             Send0PacketIfRequired,
                        int               Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid BULK instance, returned by <code>USBD_BULK_Add()</code> .
<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 shall be sent when the last data packet is a multiple of <code>MaxPacketSize</code> . Normally <code>MaxPacketSize</code> for full-speed devices is 64 bytes. For high-speed devices the normal packet size is between 64 and 512 bytes.
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite. If <code>Timeout</code> is -1, the function returns immediately and the transfer is processed asynchronously.

Return value

= 0 Successful started an asynchronous write transfer or a timeout has occurred and no data was written.

> 0 && < `NumBytes` Number of bytes that have been written before a timeout occurred.

= `NumBytes` Write transfer successful completed.

< 0 Error occurred.

Additional information

Normally `USBD_BULK_Write()` is called to let the stack send the data 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 sent is `MaxPacketSize` bytes in size. When using this function, the zero-length packet handling can be controlled. This means the function can be called when sending data in multiple steps.

Example

```
// for high-speed devices
USBD_BULK_Write(hInst, _aBuffer1, 512, 0);
USBD_BULK_Write(hInst, _aBuffer2, 512, 0);
USBD_BULK_Write(hInst, _aBuffer3, 512, 0);
// this will send 6 packets to the host with sizes: 512, 0, 512, 0, 512, 0
USBD_BULK_WriteEx(hInst, _aBuffer1, 512, 0, 0);
USBD_BULK_WriteEx(hInst, _aBuffer2, 512, 0, 0);
USBD_BULK_WriteEx(hInst, _aBuffer3, 512, 1, 0);
// this will send 4 packets to the host with sizes: 512, 512, 512, 0
```

5.4.3 Data structures

5.4.3.1 USB_BULK_INIT_DATA

Description

Initialization structure that is needed when adding a BULK interface to emUSB-Device.

Type definition

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

Structure members

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

5.4.3.2 USB_BULK_INIT_DATA_EX

Description

Initialization structure that is needed when adding a BULK interface to emUSB-Device.

Type definition

```
typedef struct {
    U16      Flags;
    U8       EPIn;
    U8       EPOut;
    const char * pInterfaceName;
    U8       InterfaceClass;
    U8       InterfaceSubClass;
    U8       InterfaceProtocol;
} USB_BULK_INIT_DATA_EX;
```

Structure members

Member	Description
Flags	Various flags. Valid bits: <ul style="list-style-type: none"> • <code>USB_BULK_FLAG_USE_CUSTOM_CLASS_IDS</code> - Allows to set custom values for the <code>bInterfaceClass</code>, <code>bInterfaceSubClass</code> and <code>bInterfaceProtocol</code>.
EPIn	Endpoint for sending data to the host.
EPOut	Endpoint for receiving data from the host.
pInterfaceName	Name of the interface.
InterfaceClass	Only used when Flags has the <code>USB_BULK_FLAG_USE_CUSTOM_CLASS_IDS</code> bit set. Allows to set the USB Class ID to a different value (default is <code>0xFF</code>).
InterfaceSubClass	Only used when Flags has the <code>USB_BULK_FLAG_USE_CUSTOM_CLASS_IDS</code> bit set. Allows to set the USB SubClass ID to a different value (default is <code>0xFF</code>).
InterfaceProtocol	Only used when Flags has the <code>USB_BULK_FLAG_USE_CUSTOM_CLASS_IDS</code> bit set. Allows to set the USB Protocol ID to a different value (default is <code>0xFF</code>).

5.5 Host API

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

To communicate with the target USB-Bulk stack an API is provided that can be used on Windows, Linux and macOS systems.

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.

To use the API in an application the header file `USBBULK.h` must be included. Depending on the host operating system used the following components must be added to the host application:

- Windows: `USBBULK.lib` and `USBBULK.dll` (These files are provided for 32- and 64-Bit applications).
- Linux: `USBBULK_Linux.c`.
- macOS: `USBBULK_MacOSX.c`.

5.5.1 Bulk Host API list

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

Function	Description
USB-Bulk Basic functions	
<code>USBBULK_Init()</code>	This function needs to be called first.
<code>USBBULK_Exit()</code>	This is a cleanup function, it shall be called when exiting the application.
<code>USBBULK_AddAllowedDeviceItem()</code>	Adds the Vendor and Product ID to the list of devices the USBBULK API should look for.
<code>USBBULK_GetNumAvailableDevices()</code>	Returns the number of connected USB-Bulk devices.
<code>USBBULK_Open()</code>	Opens an existing device.
<code>USBBULK_Close()</code>	Closes an opened device.
USB-Bulk direct input/output functions	
<code>USBBULK_Read()</code>	Reads data from target device running emUSB-Device-Bulk.
<code>USBBULK_ReadTimed()</code>	Reads data from target device running emUSB-Device-Bulk within a given timeout.
<code>USBBULK_Write()</code>	Writes data to the device.
<code>USBBULK_WriteTimed()</code>	Writes data to the device within a given timeout.
<code>USBBULK_CancelRead()</code>	This cancels an initiated read.
<code>USBBULK_FlushRx()</code>	Flush the any received data.
USB-Bulk Control functions	
<code>USBBULK_SetMode()</code>	Sets the read and write mode for a specified device running emUSB-Device-Bulk.
<code>USBBULK_GetMode()</code>	Returns the current mode of the device.
<code>USBBULK_SetReadTimeout()</code>	Sets the default read timeout for an opened device.
<code>USBBULK_SetWriteTimeout()</code>	Sets a default write timeout for an opened device.
<code>USBBULK_ResetINPipe()</code>	Resets the IN pipe that is opened to the device.
<code>USBBULK_ResetOUTPipe()</code>	Resets the OUT pipe that is opened to the device.
<code>USBBULK_ResetDevice()</code>	Resets the device via a USB reset.
USB-Bulk general GET functions	
<code>USBBULK_GetVersion()</code>	Returns the version number of the USB-BULK API.
<code>USBBULK_GetDevInfo()</code>	Retrieves information about an opened USBBULK device.
<code>USBBULK_GetDevInfoByIdx()</code>	Retrieves information about a USB device.
<code>USBBULK_GetUSBId()</code>	Returns the Product and Vendor ID of an opened device.
<code>USBBULK_GetProductName()</code>	Retrieves the device/product name if available.

Function	Description
<code>USBBULK_GetVendorName()</code>	Retrieves the vendor name of an opened USBBULK device.
<code>USBBULK_GetSN()</code>	Retrieves the USB serial number as a string which was sent by the device during the enumeration.
<code>USBBULK_GetConfigDescriptor()</code>	Gets the received target USB configuration descriptor of a specified device.
Data structures	
<code>USBBULK_DEV_INFO</code>	

5.5.2 USB-Bulk basic functions

5.5.2.1 USBBULK_Init()

Description

This function needs to be called first. This makes sure to have all structures and thread have been initialized. It also sets a callback in order to be notified when a device is added or removed.

Prototype

```
void USBBULK_Init(USBBULK_NOTIFICATION_FUNC * pfNotification,
                 void * pContext);
```

Parameters

Parameter	Description
<code>pfNotification</code>	Pointer to the user callback.
<code>pContext</code>	Context data that shall be called with the callback function.

Example

```

/*****
 *
 *     _OnDevNotify
 *
 * Function description:
 *     Is called when a new device is found or an existing device is removed.
 *
 * Parameters:
 *     pContext - Pointer to a context given when USBBULK_Init is called
 *     Index    - Device Index that has been added or removed.
 *     Event    - Type of event, currently the following are available:
 *                 USBBULK_DEVICE_EVENT_ADD
 *                 USBBULK_DEVICE_EVENT_REMOVE
 *
 */
static void _OnDevNotify(void * pContext,
                        unsigned Index,
                        USBBULK_DEVICE_EVENT Event) {
    switch(Event) {
    case USBBULK_DEVICE_EVENT_ADD:
        printf("The following DevIndex has been added: %d", Index);
        NumDevices = USBBULK_GetNumAvailableDevices(&DeviceMask);
        break;
    case USBBULK_DEVICE_EVENT_REMOVE:
        printf("The following DevIndex has been removed: %d", Index);
        NumDevices = USBBULK_GetNumAvailableDevices(&DeviceMask);
        break;
    }
}
void MainTask(void) {
<...>
    USBBULK_Init(_OnDevNotify, NULL);
<...>
}

```

5.5.2.2 USBBULK_Exit()

Description

This is a cleanup function, it shall be called when exiting the application.

Prototype

```
void USBBULK_Exit(void);
```

Additional information

We recommend to call this function before exiting the application in order to remove all handles and resources that have been allocated.

5.5.2.3 USBULK_AddAllowedDeviceItem()

Description

Adds the Vendor and Product ID to the list of devices the USBULK API should look for.

Prototype

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

Parameters

Parameter	Description
<code>VendorId</code>	The desired Vendor ID mask that shall be used with the USBULK API.
<code>ProductId</code>	The desired Product ID mask that shall be used with the USBULK API.

Additional information

It is necessary to call this function first before calling `USBULK_GetNumAvailableDevices()` or opening any connection to a device.

The function can be called multiple times to handle more than one pair of Vendor and Product IDs with the API.

5.5.2.4 USBULK_GetNumAvailableDevices()

Description

Returns the number of connected USB-Bulk devices.

Prototype

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

Parameters

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

Return value

Number of available devices running emUSB-Device-Bulk.

Additional information

For each emUSB-Device 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`.

5.5.2.5 USBULK_Open()

Description

Opens an existing device. The ID of the device can be retrieved by the function `USB_BULK_GetNumAvailableDevices()` via the `pDeviceMask` parameter. Each bit set in the `DeviceMask` represents an available device. Currently 32 devices can be managed at once.

Prototype

```
USB_BULK_HANDLE USBULK_Open(unsigned Id);
```

Parameters

Parameter	Description
<code>Id</code>	Device ID to be opened (0..31).

Return value

≠ 0 Handle to the opened device.
= 0 Error occurred.

5.5.2.6 USBULK_Close()

Description

Closes an opened device.

Prototype

```
void USBULK_Close(USB_BULK_HANDLE hDevice);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the device that shall be closed.

5.5.3 USB-Bulk direct input/output functions

5.5.3.1 USBBULK_Read()

Description

Reads data from target device running emUSB-Device-Bulk.

Prototype

```
int USBBULK_Read(USB_BULK_HANDLE hDevice,
                void * pBuffer,
                int NumBytes);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.
<code>pBuffer</code>	Pointer to a buffer that shall receive the data.
<code>NumBytes</code>	Number of bytes to be read.

Return value

= `NumBytes` All bytes have been successfully read.
 > 0, < `NumBytes` Number of bytes that have been read. If short read transfers are not allowed (normal mode) this indicates a timeout.
 = 0 A timeout occurred, no data was read.
 < 0 Error occurred.

Additional information

If short read transfers are allowed (see `USBBULK_SetMode()`) the function returns as soon as data is available, even if just a single byte was read. Otherwise the function blocks until `NumBytes` were read. In both cases the function returns if a timeout occurs. The default timeout used can be set with `USBBULK_SetReadTimeout()`.

If `NumBytes` exceeds the maximum read size the driver can handle (the default value is 64 Kbytes), `USBBULK_Read()` will read the desired `NumBytes` in chunks of the maximum read size.

5.5.3.2 USBBULK_ReadTimed()

Description

Reads data from target device running emUSB-Device-Bulk within a given timeout.

Prototype

```
int USBBULK_ReadTimed(USB_BULK_HANDLE hDevice,
                      void * pBuffer,
                      int NumBytes,
                      unsigned ms);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.
<code>pBuffer</code>	Pointer to a buffer that shall receive the data.
<code>NumBytes</code>	Maximum number of bytes to be read.
<code>ms</code>	Timeout in milliseconds.

Return value

- > 0 Number of bytes that have been read.
- = 0 A timeout occurred during read.
- < 0 Error, cannot read from the device.

Additional information

The function returns as soon as data is available, even if just a single byte was read. If no data is available, the functions return after the given timeout was expired.

If `NumBytes` exceeds the maximum read size the driver can handle (the default value is 64 Kbytes), `USBBULK_ReadTimed()` will read the desired `NumBytes` in chunks of the maximum read size.

5.5.3.3 USBBULK_Write()

Description

Writes data to the device.

Prototype

```
int USBBULK_Write(      USB_BULK_HANDLE  hDevice,
                       const void      * pBuffer,
                       int              NumBytes);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.
<code>pBuffer</code>	Pointer to a buffer that contains the data.
<code>NumBytes</code>	Number of bytes to be written. If <code>NumBytes = 0</code> , a zero length packet is written to the device.

Return value

= `NumBytes` All bytes have been successfully written.
 > 0, < `NumBytes` Number of bytes that have been written.
 = 0 A timeout occurred, no data was written.
 < 0 Error, cannot write to the device.

Additional information

The function blocks until `NumBytes` were written or a timeout occurs. The default timeout used can be set with `USBBULK_SetWriteTimeout()`.

If `NumBytes` exceeds the maximum write size the driver can handle (the default value is 64 Kbytes), `USBBULK_Write()` will write the desired `NumBytes` in chunks of the maximum write size.

5.5.3.4 USBULK_WriteTimed()

Description

Writes data to the device within a given timeout.

Prototype

```
int USBULK_WriteTimed(    USB_BULK_HANDLE    hDevice,
                        const void          * pBuffer,
                        int                 NumBytes,
                        unsigned            ms);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.
<code>pBuffer</code>	Pointer to a buffer that contains the data.
<code>NumBytes</code>	Number of bytes to be written. If <code>NumBytes = 0</code> , a zero length packet is written to the device.
<code>ms</code>	Timeout in milliseconds.

Return value

= `NumBytes` All bytes have been successfully written.
 > 0, < `NumBytes` Number of bytes that have been written.
 = 0 A timeout occurred, no data was written.
 < 0 Error, cannot write to the device.

Additional information

The function blocks until `NumBytes` were written or a timeout occurs.

If `NumBytes` exceeds the maximum write size the driver can handle (the default value is 64 Kbytes), `USBULK_WriteTimed()` will write the desired `NumBytes` in chunks of the maximum write size.

5.5.3.5 USBBULK_CancelRead()

Description

This cancels an initiated read.

Prototype

```
void USBBULK_CancelRead(USB_BULK_HANDLE hDevice);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.

Additional information

Not supported on Linux and MacOSX.

5.5.3.6 USBULK_FlushRx()

Description

Flush the any received data.

Prototype

```
int USBULK_FlushRx(USB_BULK_HANDLE hDevice);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.

Return value

- = 0 Error, bad handle.
- ≠ 0 Success, flushing the RX buffer was successful.

5.5.4 USB-Bulk control functions

5.5.4.1 USBBULK_SetMode()

Description

Sets the read and write mode for a specified device running emUSB-Device-Bulk.

Prototype

```
unsigned USBBULK_SetMode(USB_BULK_HANDLE hDevice,
                        unsigned          Mode);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.
<code>Mode</code>	Read and write mode for the USB-Bulk driver. This is a combination of the following flags, combined by binary or: <ul style="list-style-type: none"> USBBULK_MODE_BIT_ALLOW_SHORT_READ USBBULK_MODE_BIT_ALLOW_SHORT_WRITE

Return value

= 0 Operation failed (invalid handle).
≠ 0 The operation was successful.

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_Read()` returns as soon as data is available, even if it is just a single byte.

USBBULK_MODE_BIT_ALLOW_SHORT_WRITE allows short write transfers. `USBBULK_Write()` and `USBBULK_WriteTimed()` return after writing the minimal amount of data (either `NumBytes` or the maximal write transfer size).

Example

```
static void _TestMode(USB_BULK_HANDLE hDevice) {
    unsigned Mode;
    char * pText;
    Mode = USBBULK_GetMode(hDevice);
    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, (int)hDevice);
    printf("Set mode to USBBULK_MODE_BIT_ALLOW_SHORT_READ\n");
    USBBULK_SetMode(hDevice, USBBULK_MODE_BIT_ALLOW_SHORT_READ);
    Mode = USBBULK_GetMode(hDevice);
    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, (int)hDevice);
}
```

5.5.4.2 USBBULK_GetMode()

Description

Returns the current mode of the device.

Prototype

```
unsigned USBBULK_GetMode(USB_BULK_HANDLE hDevice);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.

Return value

A combination of the following flags, combined by binary or:

- `USBBULK_MODE_BIT_ALLOW_SHORT_READ` - Short read mode is enabled.
- `USBBULK_MODE_BIT_ALLOW_SHORT_WRITE` - Short write mode is enabled.

5.5.4.3 USBULK_SetReadTimeout()

Description

Sets the default read timeout for an opened device.

Prototype

```
void USBULK_SetReadTimeout(USB_BULK_HANDLE hDevice,  
                           int Timeout);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.
<code>Timeout</code>	<code>Timeout</code> in milliseconds.

5.5.4.4 USBULK_SetWriteTimeout()

Description

Sets a default write timeout for an opened device.

Prototype

```
void USBULK_SetWriteTimeout(USB_BULK_HANDLE hDevice,  
                           int Timeout);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.
<code>Timeout</code>	<code>Timeout</code> in milliseconds.

5.5.4.5 USBULK_ResetINPipe()

Description

Resets the IN pipe that is opened to the device. It also flushes any data the USB bulk driver would cache.

Prototype

```
int USBULK_ResetINPipe(USB_BULK_HANDLE hDevice);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.

Return value

- ≠ 0 The operation was successful.
- = 0 Operation failed. Either an invalid handle was used or the pipe cannot be flushed.

5.5.4.6 USBULK_ResetOUTPipe()

Description

Resets the OUT pipe that is opened to the device.

Prototype

```
int USBULK_ResetOUTPipe(USB_BULK_HANDLE hDevice);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.

Return value

- ≠ 0 The operation was successful.
- = 0 Operation failed. Either an invalid handle was used or the pipe cannot be flushed.

5.5.4.7 USBULK_ResetDevice()

Description

Resets the device via a USB reset. This can be used when the device does not work properly and may be reactivated via USB reset. This will force a re-enumeration of the device.

Prototype

```
int USBULK_ResetDevice(USB_BULK_HANDLE hDevice);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.

Return value

- ≠ 0 The operation was successful.
- = 0 Operation failed. Either an invalid handle was used or the device cannot be re-set.

Additional information

After the device has been reset it is necessary to re-open the device as the current handle will become invalid.

5.5.5 USB-Bulk general GET functions

5.5.5.1 USBBULK_GetVersion()

Description

Returns the version number of the USBBULK API.

Prototype

```
unsigned USBBULK_GetVersion(void);
```

Return value

Version number, format:

< Major Version><Minor Version><Subversion> (Mmmrr, decimal).

Example: 30203 is 3.02c

5.5.5.2 USBBULK_GetDevInfo()

Description

Retrieves information about an opened USBBULK device.

Prototype

```
void USBBULK_GetDevInfo(USB_BULK_HANDLE    hDevice,  
                        USBBULK_DEV_INFO * pDevInfo);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.
<code>pDevInfo</code>	Pointer to a device info structure of type <code>USBBULK_DEV_INFO</code>

5.5.5.3 USBULK_GetDevInfoByIdx()

Description

Retrieves information about a USB device.

Prototype

```
int USBULK_GetDevInfoByIdx(unsigned Idx,  
                           USBULK_DEV_INFO * pDevInfo);
```

Parameters

Parameter	Description
<code>Idx</code>	Index of the device.
<code>pDevInfo</code>	Pointer to a device info structure of type <code>USBULK_DEV_INFO</code>

Return value

= 0; Error, bad device index.
≠ 0 Success

5.5.5.4 USBBULK_GetUSBId()

Description

Returns the Product and Vendor ID of an opened device.

Prototype

```
void USBBULK_GetUSBId(USB_BULK_HANDLE  hDevice,  
                      U16              * pVendorId,  
                      U16              * pProductId);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
pVendorId	Pointer to a variable that receives the Vendor ID.
pProductId	Pointer to a variable that receives the Product ID.

5.5.5.5 USBULK_GetProductName()

Description

Retrieves the device/product name if available.

Prototype

```
int USBULK_GetProductName(USB_BULK_HANDLE hDevice,  
                          char * sProductName,  
                          unsigned BufferSize);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
sProductName	Pointer to a buffer that should receive the string.
BufferSize	Size of the buffer, given in bytes.

Return value

- = 0 Error, product name not available or buffer too small.
- ≠ 0 Success, product name stored in buffer pointed by [sProductName](#) as 0-terminated string.

5.5.5.6 USBULK_GetVendorName()

Description

Retrieves the vendor name of an opened USBULK device.

Prototype

```
int USBULK_GetVendorName(USB_BULK_HANDLE hDevice,  
                        char * sVendorName,  
                        unsigned BufferSize);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
sVendorName	Pointer to a buffer that should receive the string.
BufferSize	Size of the buffer, given in bytes.

Return value

- = 0 Error, bad handle.
- ≠ 0 Success, vendor name stored in buffer pointed by [sVendorName](#) as 0-terminated string.

5.5.5.7 USBULK_GetSN()

Description

Retrieves the USB serial number as a string which was sent by the device during the enumeration.

Prototype

```
int USBULK_GetSN(USB_BULK_HANDLE hDevice,
                U8 * pBuffer,
                unsigned BuffSize);
```

Parameters

Parameter	Description
<code>hDevice</code>	Handle to the opened device.
<code>pBuffer</code>	Pointer to a buffer which shall receive the serial number of the device.
<code>BuffSize</code>	Size of the buffer in bytes.

Return value

- = 0 Operation failed. Either an invalid handle was used or the serial number is not available.
- ≠ 0 The operation was successful.

Additional information

If the function succeeds, the buffer pointed by `pBuffer` contains the serial number of the device as 0-terminated string. If `BuffSize` is too small, the serial number is truncated.

5.5.5.8 USBULK_GetConfigDescriptor()

Description

Gets the received target USB configuration descriptor of a specified device.

Prototype

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

Parameters

Parameter	Description
hDevice	Handle to the opened device.
pBuffer	Pointer to the buffer that shall store the descriptor.
Size	Size of the buffer, given in bytes.

Return value

- ≠ 0 [Size](#) of the returned USB configuration descriptor (Success).
- = 0 Operation failed. Either an invalid handle was used or the buffer that shall store the config descriptor is too small.

5.5.6 USB-Bulk data structures

5.5.6.1 USBBULK_DEV_INFO

Type definition

```
typedef struct {
    U16  VendorId;
    U16  ProductId;
    char acSN[];
    char acDevName[];
    U8   InterfaceNo;
    U8   Speed;
} USBBULK_DEV_INFO;
```

Structure members

Member	Description
VendorId	Vendor ID of the device.
ProductId	Product ID of the device.
acSN	0-terminated string which holds the serial number of the device.
acDevName	0-terminated string which holds the device name.
InterfaceNo	Interface number used by this device.
Speed	Device speed. One of the following: USBBULK_SPEED_UNKNOWN USBBULK_SPEED_LOW USBBULK_SPEED_FULL USBBULK_SPEED_HIGH USBBULK_SPEED_SUPER

Chapter 6

Vendor Specific Class (VSC)

This chapter describes how to get emUSB-Device-VSC up and running.



6.1 Vendor Specific Class

The Vendor Specific Class (VSC) 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.

6.2 Requirements for the Host (PC)

In order to communicate with a target (client) running emUSB-Device, the operating system running on the host must recognize the device connected to it.

6.2.1 Windows

Microsoft's Windows operating systems (Starting with XP Service Pack 2) contains a generic driver called WinUSB.sys that is used to handle all communication to a emUSB-Device running a VSC interface. If a emUSB device is connected to a Windows 8, 8.1 and 10 PC for the first time, Windows will install the WinUSB driver automatically. For Windows versions less than Windows 8, Microsoft provides a driver for Windows Vista and Windows 7 but this needs to be installed manually. A driver installation tool including the mentioned driver is available in the `Windows\USB\VSC\WinUSBInstall`. Windows XP user can use the driver package located under `Windows\USB\VSC\WinUSB_USBVSC_XP`. In order to get emUSB VSC running with the WinUSB driver the following must be considered:

- The function `USBD_VSC_SetMSDescInfo()` must be called in the target application.
- The Product IDs 1234 and 1121 must not be used.

6.2.2 Linux

Linux can handle emUSB VSC devices out of the box.

By default a USB device can only be accessed by a process that is running with "root" rights. In order to use the USB VSC device from normal user programs an udev rule has to be configured for the device (refer to the linux udev documentation). The emUSB-Device release contains a sample configuration file `99-emUSBD.rules`, which may be modified and copied to `/etc/udev/rules.d` on the host machine.

6.2.3 macOS

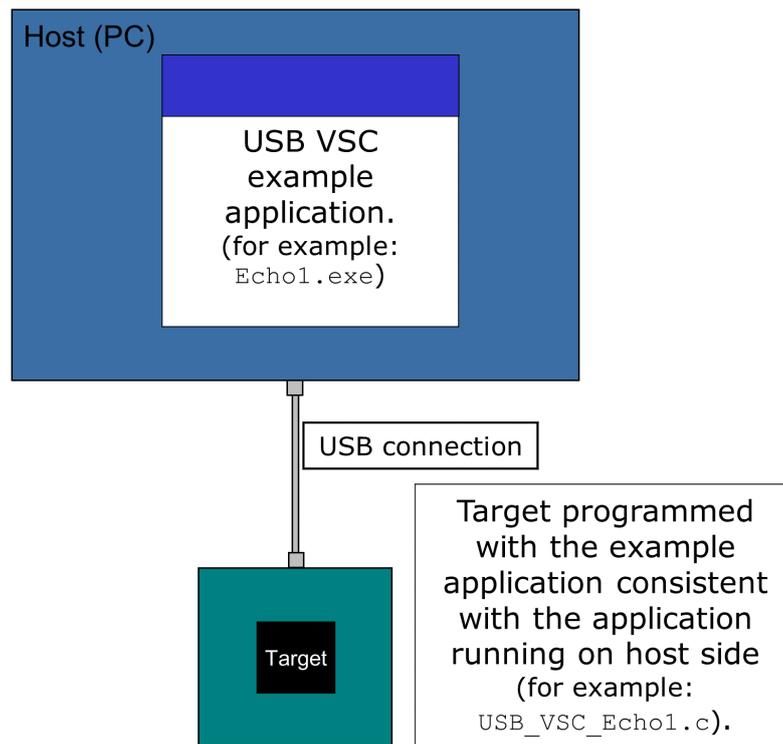
macOS can handle emUSB VSC devices out of the box.

6.3 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-Device.

The host sample applications can be used for Windows, Linux and MacOSX. Precompiled executables for Windows can be found in the subfolder `Windows/USB/VSC/WindowsApplication/Exe`.

For example: The application `USB_VSC_Echo1.c` 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, 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 `USB_VAPI_`, for example, if the host runs `Echo1.exe`, `USB_VSC_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-Device.

The following start application files are provided:

File	Description
<code>USB_VSC_Echo1.c</code>	This application was described in the upper text.
<code>USB_VSC_Test.c</code>	This application can be used to test emUSB-Device-VSC with different packet sizes received from and sent to the PC host.
<code>USB_VSC_Performance.c</code>	Measures VSC data throughput.

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 device, use one of the following example applications:

File	Description
Echo1.exe	If the <code>USB_VSC_Echo1.c</code> sample application is running on the emUSB-Device-VSC device, use this application.
Test.exe	If the <code>USB_VSC_Test.c</code> application is running on the emUSB-Device-VSC device, use this application to test the emUSB-Device-VSC stack.
Performance.exe	If the <code>USB_VSC_Performance.c</code> application is running on the emUSB-Device-VSC device, use this application to measures VSC data throughput.

For information how to compile the host examples (especially for Linux and MacOSX) refer to *Compiling the PC example application* on page 119.

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

6.3.1 Running the example applications

To test the emUSB-Device-VSC component, build and download the application of choice for the target-side.

To run one of the example applications, simply start the executable, for example by double clicking it.

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

Example output of `Echo1.exe`:

```

USB BULK Sample Echo1
USB BULK driver version: 2.70f, compiled: Oct 8 2014 16:40:43

Found 1 device
Found the following device 0:
  Vendor Name : Uendor
  Product Name: Bulk device
  Serial no.  : 13245678
To which device do you want to connect?
Please type in device number (e.g. '0' for the first device, q/a for abort):0
Starting Echo...
USB BULK driver version: 2.70f, compiled: Oct 8 2014 16:40:43
Enter the number of bytes to be send to the echo client: 100
.....
100 bytes successfully transferred.

```

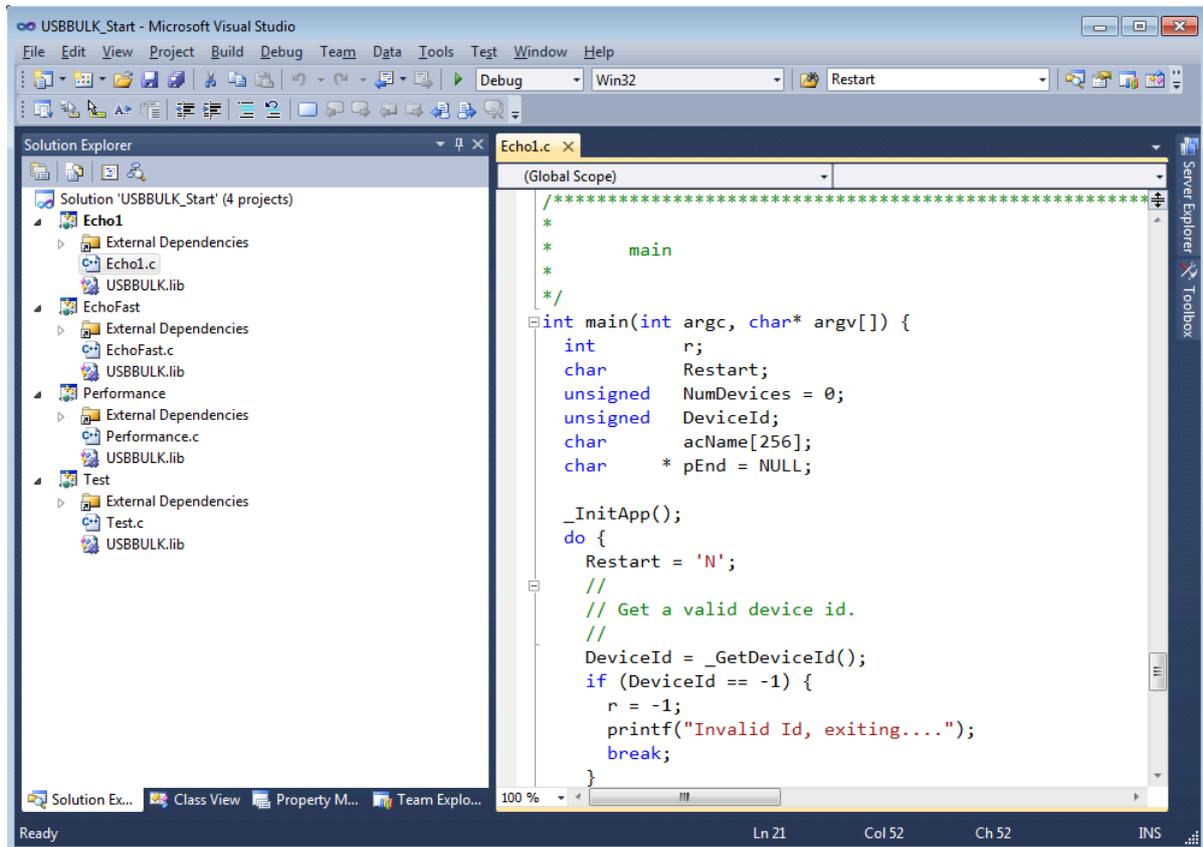
If the host example application can communicate with the emUSB-Device device, the example application will be in interactive mode for the `Echo1` and the `EchoFast` application. In case of an error, a message box is displayed.

Error Messages	Description
Unable to connect to USB VSC 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.
Could not read from device (time-out)	The PC sample application was not able to read.
Wrong data read	The result of the target sample application is not correct.

6.3.2 Compiling the PC example application

6.3.2.1 Windows

For compiling the example application you need Visual C++ 2010 (or later).



The source code of the sample application is located in the subfolder `Windows\USB\VSC\WindowsApplication`. Open the file `USBVSC_Start.sln` and compile the source.

6.3.2.2 Linux

The subfolder `Windows\USB\VSC\WindowsApplication` contains a Makefile for Linux. Change to this folder and execute "make".

6.3.2.3 macOS

The subfolder `Windows\USB\VSC\WindowsApplication` contains a Makefile for macOS. Change to this folder and execute "make -f Makefile_MacOSX".

6.4 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_VSC.c`, `USB_VSC.h`). These files contain API functions to communicate with the USB host through the emUSB-Device 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 hosts kernel.

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

6.4.1 Target interface function list

Routine	Explanation
USB VSC functions	
<code>USBD_VSC_Add()</code>	Adds a VSC interface to emUSB-Device.
<code>USBD_VSC_AddAlternateInterface()</code>	Adds an alternative interface for USB-VSC interface.
<code>USBD_VSC_SetMSDescInfo()</code>	Enables use of Microsoft OS Descriptors.
<code>USBD_VSC_CancelIO()</code>	Cancels any non-blocking/blocking data transfer operation that is pending.
<code>USBD_VSC_GetNumBytesInBuffer()</code>	Returns the number of bytes that are available in the internal BULK-OUT end-point buffer.
<code>USBD_VSC_GetNumBytesRemToRead()</code>	Get the number of remaining bytes to read by an active read operation.
<code>USBD_VSC_GetNumBytesRemToWrite()</code>	After starting a non-blocking write operation this function can be used to periodically check how many bytes still have to be written.
<code>USBD_VSC_Read()</code>	Reads data from the host with a given timeout.
<code>USBD_VSC_ReadAsync()</code>	Reads data from the host asynchronously.
<code>USBD_VSC_SetContinuousReadMode()</code>	Enables continuous read mode for the RX endpoint.
<code>USBD_VSC_SetOnSetupRequest()</code>	Sets a callback function that is called when any setup request is sent from the host.
<code>USBD_VSC_SetOnEPEvent()</code>	Sets a callback function for the IN end-point that will be called on every TX event for that endpoint.
<code>USBD_VSC_TxIsPending()</code>	Checks whether the TX (IN endpoint) is currently pending.
<code>USBD_VSC_WaitEP()</code>	Waits (blocking) until the triggered <code>USBD_VSC_ReadOverlapped()</code> has received the desired data.

Routine	Explanation
<code>USBD_VSC_PollEP()</code>	Waits (blocking) until the triggered <code>USB-D_VSC_ReadOverlapped()</code> has received the desired data.
<code>USBD_VSC_StallEP()</code>	Stall an EP.
<code>USBD_VSC_WaitForTXReady()</code>	Waits (blocking) until the TX queue can accept another data packet.
<code>USBD_VSC_Write()</code>	Sends data to the USB host.
<code>USBD_VSC_WriteAsync()</code>	Sends data to the host asynchronously.
<code>USBD_VSC_SetOnVendorRequest()</code>	Sets a callback function that is called when a setup vendor request is sent from the host to the specified interface index.
<code>USBD_VSC_SetOnSetupRequest()</code>	Sets a callback function that is called when any setup request is sent from the host.
<code>USBD_VSC_SetOnClassRequest()</code>	Sets a callback function that is called when a setup class request is sent from the host to the specified interface index.
<code>USBD_VSC_AddAlternateInterface()</code>	Adds an alternative interface for USB-VSC interface.
Data structures and callbacks	
<code>USB_VSC_INIT_DATA</code>	Initialization structure that is needed when adding a VSC interface to emUSB-Device.
<code>USB_VSC_MSOSDESC_INFO</code>	MS OS descriptor structure that contains for MS related OSes information how to deal with device with out having a driver store.
<code>USB_ON_ADD_FUNCTION_DESC</code>	Call back that is used to add an additional descriptor between the interface or one of its alternate setting descriptor and the endpoint descriptor(s).
<code>USB_VSC_ON_SET_INTERFACE</code>	Global callback function that is called whenever an alternate setting is set for a for an inteface that was added with <code>USB-D_VSC_Add()</code> .

6.4.2 USB-VSC functions

6.4.2.1 USBD_VSC_Add()

Description

Adds a VSC interface to emUSB-Device.

Prototype

```
USB_VSC_HANDLE USBD_VSC_Add(const USB_VSC_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
<code>pInitData</code>	Pointer to <code>USB_VSC_INIT_DATA</code> structure.

Return value

Handle to a valid VSC instance. The handle of the first VSC instance is always 0.

Example

Example excerpt from `USB_VSC_Echo1.c`:

Example

```
static void _AddVSC(void) {
    static U8 _abOutBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
    USB_VSC_INIT_DATA Init;
    Init.Flags = 0;
    Init.EPIn = USBD_AddeP(USB_DIR_IN,
                          USB_TRANSFER_TYPE_BULK,
                          USB_HS_BULK_MAX_PACKET_SIZE,
                          NULL,
                          0);
    Init.EPOut = USBD_AddeP(USB_DIR_OUT,
                           USB_TRANSFER_TYPE_BULK,
                           USB_HS_BULK_MAX_PACKET_SIZE,
                           _abOutBuffer,
                           USB_HS_BULK_MAX_PACKET_SIZE);
    Init.pInterfaceName = "VSC Interface";
    USBD_VSC_Add(&Init);
}
```

6.4.2.2 USBD_VSC_AddAlternateInterface()

Description

Adds an alternative interface for USB-VSC interface.

Prototype

```
void USBD_VSC_AddAlternateInterface(    USB_VSC_HANDLE    hInst,
                                       const USB_VSC_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid VSC instance, returned by <code>USB_D_VSC_Ad-</code> <code>d()</code> .
<code>pInitData</code>	Pointer to <code>USB_VSC_INIT_DATA</code> structure.

6.4.2.3 USBD_VSC_SetMSDescInfo()

Description

Enables use of Microsoft OS Descriptors. A USB bulk device providing these descriptors is detected by Windows to be handled by the generic WinUSB driver. For such devices no other driver needs to be installed.

Prototype

```
void USBD_VSC_SetMSDescInfo(      USB_VSC_HANDLE      hInst,
                                const USB_VSC_MSOSDESC_INFO * pMSDescInfo);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid VSC instance, returned by <code>USB_D_VSC_Add()</code> .
<code>pMSDescInfo</code>	Pointer to a MS OS descriptor structure.

Additional information

This function must be called after the call to the function `USB_D_VSC_Add()` and before `USB_D_Start()`.

6.4.2.4 USBD_VSC_CancelIO()

Description

Cancels any non-blocking/blocking data transfer operation that is pending.

Prototype

```
void USBD_VSC_CancelIO(U8 EPIndex);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to <code>USB_D_VSC_Add()</code> .

Additional information

This function shall be called when a pending asynchronous transfer 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.

6.4.2.5 USBD_VSC_GetNumBytesInBuffer()

Description

Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer. This functions does not start a read transfer.

Prototype

```
unsigned USBD_VSC_GetNumBytesInBuffer(U8 EPIndex);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to <code>USB_D_VSC_Add()</code> .

Return value

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

Additional information

If the host is sending more data than your target application has requested, the remaining data will be stored in an internal buffer. This function shows how many bytes are available in this buffer.

The number of bytes returned by this function can be read using `USB_D_VSC_Read()` without blocking.

Example

Your host application sends 50 bytes. Your target application only requests to receive 1 byte. In this case the target application will get 1 byte and the remaining 49 bytes are stored in an internal buffer. When your target application now calls `USB_D_VSC_GetNumBytesInBuffer()` it will return the number of bytes that are available in the internal buffer (49).

6.4.2.6 USBD_VSC_GetNumBytesRemToRead()

Description

Get the number of remaining bytes to read by an active read operation. This function is to be used in combination with `USBD_VSC_Read()` where `Timeout=-1`. After starting the read operation this function can be used to periodically check how many bytes still have to be read.

Prototype

```
unsigned USBD_VSC_GetNumBytesRemToRead(U8 EPIndex);
```

Parameters

Parameter	Description
<code>EPIndex</code>	A valid EP Index that was also passed to <code>USBD_VSC_Add()</code> .

Return value

≥ 0 Number of bytes which have not yet been read.
 < 0 Error occurred.

Additional information

Alternatively the blocking function `USBD_VSC_WaitForRX()` can be used.

Example

```
NumBytesReceived = USBD_VSC_Read(hInst, &ac[0], 50, -1, 0);
if (NumBytesReceived < 0) {
    <.. error handling..>
}
if (NumBytesReceived > 0) {
    // Already had some data in the internal buffer.
    // The first 'NumBytesReceived' bytes may be processed here.
    <...>
} else {
    // Wait until we get all 50 bytes
    while (USBD_VSC_GetNumBytesRemToRead(hInst) > 0) {
        USB_OS_Delay(50);
    }
}
```

6.4.2.7 USBD_VSC_GetNumBytesRemToWrite()

Description

After starting a non-blocking write operation this function can be used to periodically check how many bytes still have to be written.

Prototype

```
unsigned USBD_VSC_GetNumBytesRemToWrite(U8 EPIndex);
```

Parameters

Parameter	Description
<code>EPIndex</code>	A valid EP Index that was also passed to <code>USB_D_VSC_Add()</code> .

Return value

Number of bytes which have not yet been written.

Additional information

Alternatively the blocking function `USB_D_VSC_WaitForTX()` can be used.

Example

```
r = USBD_VSC_Write(hInst, &ac[0], TRANSFER_SIZE, -1);
if (r < 0) {
    <.. error handling..>
}
// NumBytesToWrite shows how many bytes still have to be written.
while (USB_D_VSC_GetNumBytesRemToWrite(hInst) > 0) {
    USB_OS_Delay(50);
}
```

6.4.2.8 USBD_VSC_Read()

Description

Reads data from the host with a given timeout.

Prototype

```
int USBD_VSC_Read(U8          EPIndex,
                  void        * pData,
                  unsigned    NumBytes,
                  int         Timeout,
                  unsigned    Flags);
```

Parameters

Parameter	Description
EPIndex	One of the EPIndex was used in <code>pInitData</code> when calling <code>USBD_VSC_Add()</code> .
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
Timeout	Timeout in milliseconds. 0 means infinite. If Timeout is -1, the function returns immediately and the transfer is processed asynchronously.
Flags	Various flags: <ul style="list-style-type: none"> • <code>USB_VSC_READ_FLAG_RECEIVE</code> - This turns the read function in a the receive mode. • <code>USB_VSC_READ_FLAG_POLL</code> - Can only be used with the <code>USB_VSC_READ_FLAG_RECEIVE</code>. This function will not abort the transfer in case of a timeout.

Return value

= [NumBytes](#) Requested data was successfully read within the given timeout.
 ≥ 0 && < [NumBytes](#) [Timeout](#) has occurred. Number of bytes that have been read within the given timeout.
 < 0 Error occurred.

Additional information

- Normal mode:

This function blocks a task until all data have been read or a timeout expires. This function also returns when the device is disconnected from host or when a USB reset occurs.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USBD_AddEP()`. This data can be retrieved by a later call to `USBD_VSC_Read()`. See also `USBD_VSC_GetNumBytesInBuffer()`.

When [Timeout](#) = -1 the read transfer will be initiated and the function returns immediately. In order to synchronize, `USBD_VSC_WaitForRX()` needs to be called. Alternatively the function `USBD_VSC_GetNumBytesRemToRead()` can be called periodically to check whether all bytes have been read or not. The read operation can be canceled using `USBD_VSC_Cancel()`. The buffer pointed to by [pData](#) must be valid until the read operation is terminated.

In case of a timeout, the read transfer is aborted (see [Timeout handling](#) on page 113).

If a read transfer was still pending while the function is called, it returns `USB_STATUS_EP_BUSY`.

- Receive mode:

If no error occurs, this function returns the number of bytes received. Calling `USBD_VSC_Receive()` will return as much data as is currently available -- up to the size of the buffer

specified. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return `USB_STATUS_ERROR`.

If a read transfer was pending while the function is called, it returns `USB_STATUS_EP_BUSY`.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USBD_AddEP()`. This data can be retrieved by a later call to `USBD_VSC_Read()`. See also `USBD_VSC_GetNumBytesInBuffer()`.

A call of `USBD_VSC_Read(EPIndex, NULL, 0, -1, USB_VSC_)` can be used to trigger an asynchronous read that stores the data into the internal buffer.

- Receive in polled mode:

In contrast to receive mode this function will continue the read transfer asynchronously in case of a timeout.

6.4.2.9 USBD_VSC_ReadAsync()

Description

Reads data from the host asynchronously. The function does not wait for the data to be received. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_VSC_ReadAsync(U8                EPIndex,
                       USB_ASYNC_IO_CONTEXT * pContext,
                       int                ShortRead);
```

Parameters

Parameter	Description
<code>EPIndex</code>	A valid OUT EP Index that was also passed to <code>USB_D_VSC_Add()</code> .
<code>pContext</code>	Pointer to a structure of type <code>USB_ASYNC_IO_CONTEXT</code> containing parameters and a pointer to the callback function.
<code>ShortRead</code>	<ul style="list-style-type: none"> 0: The transfer is completed successfully after all bytes have been read. 1: The transfer is completed successfully after one packet has been read.

Example

```
static void _AsyncCb(USB_ASYNC_IO_CONTEXT * pIOContext) {
    U8 *p;

    p = (U8 *)pIOContext->pContext;
    *p = 1;
}

<...>

USB_ASYNC_IO_CONTEXT IOContext;
U8 AsyncComplete;

IOContext.NumBytesToTransfer = 5000;
IOContext.pData              = pBuffer;
IOContext.pfOnComplete       = _AsyncCb;
IOContext.pContext           = (void *)&AsyncComplete;
AsyncComplete = 0;
USB_D_VSC_ReadAsync(hInst, &IOContext, 0);
while (AsyncComplete == 0) {
    <.. Do other work. ..>
}
// Transaction is complete.
if (IOContext.Status < 0 || IOContext.NumBytesTransferred != 5000) {
    <.. error handling ..>
} else {
    <.. Process the data ..>
}
<...>
```

6.4.2.10 USBD_VSC_SetContinuousReadMode()

Description

Enables continuous read mode for the RX endpoint. In this mode every finished read transfer will automatically trigger another read transfer, as long as there is enough space in the internal buffer to receive another packet.

Prototype

```
void USBD_VSC_SetContinuousReadMode(U8 EPIndex);
```

Parameters

Parameter	Description
<code>EPIndex</code>	A valid EP Index that was also passed to <code>USB_D_VSC_Add()</code> .

Additional information

To check how many bytes have been read into the buffer, the function `USB_D_VSC_GetNumBytesInBuffer()` may be called. In order to read the data the function `USB_D_VSC_Receive()` needs to be called (non-blocking).

The USB stack will use the buffer that was provided by the application with `USB_D_AddEP()`. The transfer speed may be improved, if this buffer has a size of at least $2 * \text{MaxPacketSize}$. Normally `MaxPacketSize` for full-speed devices is 64 bytes and for high-speed devices 512 bytes.

Example

```
USB_D_VSC_SetContinuousReadMode(hInst);
<...>
for(;;) {
    //
    // Fetch data that was already read (non-blocking).
    //
    NumBytesReceived = USB_D_VSC_Read(hInst, &ac[0], sizeof(ac), -1, USB_VSC_READ_FLAG_POLL);
    if (NumBytesReceived > 0) {
        //
        // We got some data
        //
        <.. Process data..>
    } else {
        <.. Nothing received yet, do application processing..>
    }
}
```

6.4.2.11 USBD_VSC_SetOnSetupRequest()

Description

Sets a callback function that is called when any setup request is sent from the host.

Prototype

```
void USBD_VSC_SetOnSetupRequest(USB_VSC_HANDLE hInst,  
                                USB_ON_SETUP * pfOnSetupRequest);
```

Parameters

Parameter	Description
hInst	Handle to a valid VSC instance, returned by <code>USB_D_VSC_Ad-</code> <code>d()</code> .
pfOnSetupRequest	Pointer to the callback function.

6.4.2.12 USBD_VSC_SetOnEPEvent()

Description

Sets a callback function for the IN endpoint that will be called on every TX event for that endpoint.

Prototype

```
void USBD_VSC_SetOnEPEvent(U8          EPIndex,
                           USB_EVENT_CALLBACK * pEventCb,
                           USB_EVENT_CALLBACK_FUNC * pfEventCb,
                           void          * pContext);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to USBD_VSC_Add().
pEventCb	Pointer to a USB_EVENT_CALLBACK structure. The structure is initialized by this function.
pfEventCb	Pointer to the callback routine that will be called on every event on the USB endpoint.
pContext	A pointer which is used as parameter for the callback function.

Additional information

The USB_EVENT_CALLBACK structure is private to the USB stack. It will be initialized by USBD_VSC_SetOnEPEvent(). The USB stack keeps track of all event callback functions using a linked list. The USB_EVENT_CALLBACK structure will be included into this linked list and must reside in static memory.

The callback function is called only, if a transfer operation was started using either USBD_VSC_Read() or USBD_VSC_Write() functions.

The callback function has the following prototype:

```
typedef void USB_EVENT_CALLBACK_FUNC(unsigned Events, void *pContext);
```

Parameter	Description
Events	A bit mask indicating which events occurred on the endpoint.
pContext	The pointer which was provided to the USBD_SetOnEvent() function.

Note that the callback function will be called within an ISR, therefore it should never block. The first parameter to the callback function will contain a bit mask for all events that triggered the call:

EP Direction	Event	Description
OUT	USB_EVENT_DATA_READ	Some data was received from the host on the endpoint.
OUT	USB_EVENT_READ_COMPLETE	The last read operation was completed.
OUT	USB_EVENT_READ_ABORT	A read transfer was aborted.
IN	USB_EVENT_DATA_SEND	Some data was sent to the host, so that (part of) the user write buffer may be reused by the application.

EP Direction	Event	Description
IN	USB_EVENT_DATA_ACKED	Some data was acknowledged by the host.
IN	USB_EVENT_WRITE_ABORT	A write transfer was aborted.
IN	USB_EVENT_WRITE_COMPLETE	All write operations were completed.

Example for an OUT EP

```
// The callback function.
static void _OnEvent(unsigned Events, void *pContext) {
    unsigned NumBytes;

    if (Events & USB_EVENT_DATA_READ) {
        NumBytes = USBD_VSC_GetNumBytesInBuffer(hInst);
        if (NumBytes) {
            r = USBD_VSC_Receive(hInst, Buff, NumBytes, -1);
            if (r > 0) {
                <.. process data in Buff..>
            }
        }
    }
}

// Main program.
// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USB_D_VSC_SetOnEPEvent(EPOut, &_usb_callback, _OnEvent, NULL);
USB_D_VSC_SetContinuousReadMode(EPOut);
// Trigger first read
USB_D_VSC_Read(EPOut, NULL, 0, -1, USB_VSC_READ_FLAG_POLL);
<.. do anything else here while the data is processed in the callback ..>
```

Example for an IN EP

```
// The callback function.
static void _OnEvent(unsigned Events, void *pContext) {
    if ((Events & USB_EVENT_DATA_SEND) != 0 &&
        // Check for last write transfer to be completed.
        USBD_VSC_GetNumBytesRemToWrite(_hInst) == 0) {
        <.. prepare next data for writing..>
        // Send next packet of data.
        r = USBD_VSC_Write(_hInst, &ac[0], 200, -1);
        if (r < 0) {
            <.. error handling..>
        }
    }
}

// Main program.
// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USB_D_VSC_SetOnEPEvent(hInst, &_usb_callback, _OnEvent, NULL);
// Send the first packet of data using an asynchronous write operation.
r = USBD_VSC_Write(_hInst, &ac[0], 200, -1, 0);
if (r < 0) {
    <.. error handling..>
}
<.. do anything else here while the whole data is send..>
```

6.4.2.13 USBD_VSC_TxIsPending()

Description

Checks whether the TX (IN endpoint) is currently pending. Can be called in any context.

Prototype

```
int USBD_VSC_TxIsPending(U8 EPIndex);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to <code>USB_D_VSC_Add()</code> .

Return value

- 1 We have queued data to be sent.
- 0 Queue is empty.

6.4.2.14 USBD_VSC_WaitEP()

Description

Waits (blocking) until the triggered `USB_D_VSC_ReadOverlapped()` has received the desired data.

Prototype

```
int USBD_VSC_WaitEP(U8          EPIndex,
                   unsigned Timeout);
```

Parameters

Parameter	Description
<code>EPIndex</code>	A valid EP Index that was also passed to <code>USB_D_VSC_Add()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite.

Return value

= 0 Transfer completed.
 = 1 `Timeout` occurred.
 < 0 An error occurred (e.g. target disconnected)

Additional information

In case of a timeout, the read transfer is aborted (see [Timeout handling](#) on page 113).

Example

```
if (USB_D_VSC_Read(hInst, &ac[0], 50, -1, 0) < 0) {
    <.. error handling..>
    return;
}
//
// USB_D_VSC_Read() with Timeout==-1 will return immediately.
// Do something else while data may be transferred.
//
<...>
//
// Now wait until we get all 50 bytes.
// USB_D_VSC_WaitEP() will block, until total of
// 50 bytes are read or timeout occurs.
//
if (USB_D_VSC_WaitEP(hInst, timeout) != 0) {
    <.. timeout error handling..>
    return;
}
// Now we have 50 bytes of data.
// Process 50 bytes of data from ac[] here.
```

6.4.2.15 USBD_VSC_PollEP()

Description

Waits (blocking) until the triggered `USB_D_VSC_ReadOverlapped()` has received the desired data.

Prototype

```
int USBD_VSC_PollEP(U8          EPIndex,
                   unsigned Timeout);
```

Parameters

Parameter	Description
<code>EPIndex</code>	A valid EP Index that was also passed to <code>USB_D_VSC_Add()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite.

Return value

= 0 Transfer completed.
 = 1 `Timeout` occurred.
 < 0 An error occurred (e.g. target disconnected)

Additional information

In case of a timeout, the current transfer is not affected. The function may be called repeatedly until it does not report a timeout any more.

Example for an IN EP

```
if (USB_D_VSC_Write(hInst, &ac[0], 50, -1, 0) < 0) {
    <.. error handling..>
    return;
}
//
// USB_D_VSC_Write() will return immediately.
// While waiting for the data to be transferred, we will blink a LED with
// 200 ms interval.
// USB_D_VSC_PollForTX() will return, if all data were send or 100 ms expired.
//
while ((r = USB_D_VSC_PollEP(hInst, 100)) > 0) {
    ToggleLED();
}
if (r < 0) {
    <.. error handling..>
    return;
}
// Now all data have been send.
```

Example for an OUT EP

```
if (USB_D_VSC_Read(hInst, &ac[0], 50, -1, 0) < 0) {
    <.. error handling..>
    return;
}
//
// USB_D_VSC_Read() with Timeout== -1 will return immediately.
// While waiting for the data, we will blink a LED with 200 ms interval.
// USB_D_VSC_PollForRX() will return, if all data were read or 100 ms expired.
//
while ((r = USB_D_VSC_PollEP(hInst, 100)) > 0) {
    ToggleLED();
}
```

```
if (r < 0) {  
    <.. error handling..>  
    return;  
}  
// Now we have 50 bytes of data.  
// Process 50 bytes of data from ac[] here.
```

6.4.2.16 USBD_VSC_WaitForTXReady()

Description

Waits (blocking) until the TX queue can accept another data packet. This function is used in combination with a non-blocking call to `USB_D_VSC_Write()`, it waits until a new asynchronous write data transfer will be accepted by the USB stack.

Prototype

```
int USBD_VSC_WaitForTXReady(U8 EPIndex,
                           int Timeout);
```

Parameters

Parameter	Description
<code>EPIndex</code>	A valid EP Index that was also passed to <code>USB_D_VSC_Add()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite. If <code>Timeout</code> is negative, the function will return immediately.

Return value

- = 0 A new asynchronous write data transfer will be accepted.
- = 1 The write queue is full, a call to `USB_D_VSC_Write()` would return `USB_STATUS_EP_BUSY`.
- < 0 Error occurred.

Additional information

If `Timeout` is 0, the function never returns 1.

If `Timeout` is -1, the function will not wait, but immediately return the current state.

Example

```
// Always keep the write queue full for maximum send speed.
for (;;) {
    pData = GetNextData(&NumBytes);
    // Wait until stack can accept a new write.
    USBD_VSC_WaitForTxReady(hInst, 0);
    // Issue write transfer.
    if (USB_D_VSC_Write(hInst, pData, NumBytes, -1) < 0) {
        <.. error handling..>
    }
}
```

6.4.2.17 USBD_VSC_Write()

Description

Sends data to the USB host. Depending on the `Timeout` parameter, the function blocks until `NumBytes` have been written or a timeout occurs.

Prototype

```
int USBD_VSC_Write(    U8          EPIndex,
                     const void * pData,
                     unsigned NumBytes,
                     int      Timeout,
                     unsigned  Flags);
```

Parameters

Parameter	Description
<code>EPIndex</code>	One of the <code>EPIndex</code> was used in <code>pInitData</code> when calling <code>USBD_VSC_Add()</code> .
<code>pData</code>	Data that should be written.
<code>NumBytes</code>	Number of bytes to write.
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite. If <code>Timeout</code> is -1, the function returns immediately and the transfer is processed asynchronously.
<code>Flags</code>	Various flags: Current flags: <ul style="list-style-type: none"> <code>USB_VSC_WRITE_FLAG_NO_NULL_PACKET</code> - Specifies that a zero-length packet shall be sent when the last data packet is a multiple of <code>MaxPacketSize</code>. Normally <code>MaxPacketSize</code> for full-speed devices is 64 bytes. For high-speed devices the normal packet size is between 64 and 512 bytes.

Return value

= 0 Successful started an asynchronous write transfer or a timeout has occurred and no data was written.

> 0 && < `NumBytes` Number of bytes that have been written before a timeout occurred.

= `NumBytes` Write transfer successful completed.

< 0 Error occurred.

Additional information

This function also returns when the target is disconnected from host or when a USB reset occurred.

The USB stack is able to queue a small number of asynchronous write transfers (when using `Timeout = -1`). If a write transfer is still in progress when this function is called and the USB stack can not accept another write transfer request, the functions returns `USB_STATUS_EP_BUSY`. A synchronous write transfer (`Timeout ≥ 0`) will always block until the transfer (including all pending transfers) are finished or a timeout occurs.

In case of a timeout, the write transfer is aborted (see *Timeout handling* on page 113).

In order to synchronize, `USBD_VSC_WaitForTX()` needs to be called. Another synchronization method would be to periodically call `USBD_VSC_GetNumBytesRemToWrite()` in order to see how many bytes still need to be written (this method is preferred when a non-blocking solution is necessary). The write operation can be canceled using `USBD_VSC_CancelWrite()`.

If `pData = NULL` and `NumBytes = 0`, a zero-length packet is sent to the host.

The content of the buffer pointed to by `pData` must not be changed until the transfer has been completed.

Example

```
NumBytesWritten = USBD_VSC_Write(hInst, &ac[0], DataSize, 500);
if (NumBytesWritten <= 0) {
    <.. error handling..>
}
if (NumBytesWritten < DataSize) {
    <.. timeout occurred, data partially written within 500ms ..>
} else {
    <.. write completed successfully..>
}
```

See also `USB_D_VSC_GetNumBytesRemToWrite` on page 199.

6.4.2.18 USBD_VSC_WriteAsync()

Description

Sends data to the host asynchronously. The function does not block. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_VSC_WriteAsync(U8          EPIndex,
                        USB_ASYNC_IO_CONTEXT * pContext,
                        char          Send0PacketIfRequired);
```

Parameters

Parameter	Description
<code>EPIndex</code>	A valid EP Index that was also passed to <code>USB_D_VSC_Add()</code> .
<code>pContext</code>	Pointer to a structure of type <code>USB_ASYNC_IO_CONTEXT</code> containing parameters and a pointer to the callback function.
<code>Send0PacketIfRequired</code>	Specifies that a zero-length packet shall be sent when the last data packet is a multiple of <code>MaxPacketSize</code> .

Example

```
static void _AsyncCb(USB_ASYNC_IO_CONTEXT * pIOContext) {
    U8 *p;

    p = (U8 *)pIOContext->pContext;
    *p = 1;
}

<...>

USB_ASYNC_IO_CONTEXT IOContext;
U8 AsyncComplete;

IOContext.NumBytesToTransfer = 5000;
IOContext.pData              = pBuffer;
IOContext.pfOnComplete      = _AsyncCb;
IOContext.pContext          = (void *)&AsyncComplete;
AsyncComplete = 0;
USB_D_VSC_WriteAsync(hInst, &IOContext, 1);
while (AsyncComplete == 0) {
    <.. Do other work. ..>
}
// Transaction is complete.
if (IOContext.Status < 0 || IOContext.NumBytesTransferred != 5000) {
    <.. error handling ..>
} else {
    <.. data written successfully ..>
}
<...>
```

6.4.3 Data structures

6.4.3.1 USB_VSC_INIT_DATA

Description

Initialization structure that is needed when adding a VSC interface to emUSB-Device.

Type definition

```
typedef struct {
    U16      Flags;
    U8       aEP[];
    U8       NumEPs;
    const char * pInterfaceName;
    U8       InterfaceClass;
    U8       InterfaceSubClass;
    U8       InterfaceProtocol;
} USB_VSC_INIT_DATA;
```

Structure members

Member	Description
Flags	Various flags. No flags so far This has to be initialized to 0.
aEP	Array of Endpoints Indices to be used. Each EPIndex needs to be allocated by <code>USBD_AddEP</code>
NumEPs	Number of EPIndex in array.
pInterfaceName	Name of the interface.
InterfaceClass	Sets the USB Class ID .
InterfaceSubClass	Sets the USB SubClass ID.
InterfaceProtocol	Sets the USB Protocol ID.

6.4.3.2 USB_VSC_MSOSDESC_INFO

Description

MS OS descriptor structure that contains for MS related OSes information how to deal with device with out having a driver store.

Type definition

```
typedef struct {
    const char    * sCompatibleID;
    const char    * sSubCompatibleID;
    U32           NumProperties;
    USB_MS_OS_EXT_PROP  aProperties[];
} USB_VSC_MSOSDESC_INFO;
```

Structure members

Member	Description
<code>sCompatibleID</code>	Strings that gives MS OS a hint to the driver that shall be loaded
<code>sSubCompatibleID</code>	[Optional] Gives a sub id string, in most cases this can be NULL.
<code>NumProperties</code>	NumBer of properties that are stored in <code>pProperties</code>
<code>aProperties</code>	Variable array of MS OS extended OS descriptors. Depending on the <code>sCompatibleID</code> , this can be some sub structure which will be stored in the Windows registry. Eg. with WinUSB this contains the GUID which is needed to identify your device among other WinUSB devices.

6.4.3.3 USB_ON_ADD_FUNCTION_DESC

Description

Call back that is used to add an additional descriptor between the interface or one of its alternate setting descriptor and the endpoint descriptor(s).

Type definition

```
typedef const U8 * (USB_ON_ADD_FUNCTION_DESC)(USB_VSC_HANDLE hInst,
                                             U8
                                             IFAAlternateSetting);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid VSC instance, returned by <code>USBD_VSC_Ad-</code> <code>d()</code> .
<code>IFAlternateSetting</code>	Data that should be written.

Return value

`= NULL` No additional descriptor shall be added to this interface or its alternate setting..
`≠ NULL` Pointer to a USB descriptor.

Additional information

USB Descriptor follow a specific format. The first byte is always the length. The second byte describes the descriptor type. Anything after these 2 bytes is descriptor dependent.

6.4.3.4 USB_VSC_ON_SET_INTERFACE

Description

Global callback function that is called whenever an alternate setting is set for a for an interface that was added with `USBD_VSC_Add()`.

Type definition

```
typedef void (USB_VSC_ON_SET_INTERFACE)(USB_VSC_HANDLE hInst,  
                                        U8  
                                        AlternateInterface);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid VSC instance, returned by <code>USBD_VSC_Add()</code> .
<code>AlternateInterface</code>	Alternate interface that was set by the host..

Additional information

Each interface has one alternate setting which is the default setting. This call back is called after the host has set the alternate setting. By default all data transfers of the previous Endpoint interface alternate setting are canceled.

Chapter 7

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.



7.1 Overview

The Mass Storage Device (MSD) is a USB class protocol defined by the USB Implementers Forum. The class itself is used to access one or more storage devices such as flash drives or memory sticks.

As the USB mass storage device class is well standardized, every major operating system such as Microsoft Windows (after Windows 2000), Apple OS X, Linux and many more support it. So therefore an installation of a custom host USB driver is normally not necessary.

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

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

7.2 MSD Configuration

7.2.1 Initial configuration

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

7.2.2 Final configuration

The configuration must only be modified, when emUSB-Device is deployed in your final product. Refer to *emUSB-Device Configuration* on page 48 for detailed information about the generic information functions which must be adapted.

In order to comply with the Mass Storage Device Bootability specification, the serial number provided by the function `USBD_SetDeviceInfo()` must be a string with at least 12 characters, where each character is a hexadecimal digit ('0' through '9' or 'A' through 'F').

7.2.3 MSD class specific configuration functions

Beside the generic emUSB-Device configuration functions (*emUSB-Device Configuration* on page 48), the following should be adapted before the emUSB-Device MSD component is used in a final product. Example implementations are supplied in the MSD example application `USB_MSD_FS_Start.c`, located in the `Application` directory of emUSB-Device.

Each logical unit (storage) which is added to the MSD component has its own set of name and id values which is supplied when the logical unit is first added through `USBD_MSD_AddUnit()`

Example

```
static const USB_MSD_LUN_INFO _Lun0Info = {
    "Vendor",      // MSD VendorName
    "MSD Volume", // MSD ProductName
    "1.00",       // MSD ProductVer
    "134657890"   // MSD SerialNo
};
...
InstData.pLunInfo = &_Lun0Info;
...
USB_MSD_AddUnit(&InstData);
```

7.2.4 Running the example application

The directory `Application` contains example applications that can be used with emUSB-Device and the MSD component. To test the emUSB-Device-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.

7.2.4.1 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) {
    USBD_Init();
    _AddMSD();
    USBD_Start();
    while (1) {
        while ((USBD_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED))
               != USB_STAT_CONFIGURED) {
            BSP_ToggleLED(0);
            USB_OS_Delay(50);
        }
        BSP_SetLED(0);
        USBD_MSD_Task();
    }
}

```

The first step is to initialize the USB core stack using `USBD_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_HS_BULK_MAX_PACKET_SIZE];
    USB_MSD_INIT_DATA InitData;
    USB_MSD_INST_DATA InstData;
    InitData.EPIn = USBD_AddEP(1, USB_TRANSFER_TYPE_BULK,
                               USB_HS_BULK_MAX_PACKET_SIZE, NULL, 0);
    InitData.EPOut = USBD_AddEP(0, USB_TRANSFER_TYPE_BULK,
                                 USB_HS_BULK_MAX_PACKET_SIZE,
                                 _abOutBuffer, sizeof(_abOutBuffer));

    USBD_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;
    InstData.pLunInfo = &_Lun0Info;
    USBD_MSD_AddUnit(&InstData);
}

```

The example application uses a RAM disk as storage medium.

The example RAM disk has a size of 23 kB (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` (in multiples of 512), but the size must be at least 23 kB 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

```

7.3 Target API

Function	Description
API functions	
<code>USBD_MSD_Add()</code>	Adds an MSD-class interface to the USB stack.
<code>USBD_MSD_AddUnit()</code>	Adds a mass storage device to emUSB-Device-MSD.
<code>USBD_MSD_AddCDRom()</code>	Adds a CD-ROM device to emUSB-Device-MSD.
<code>USBD_MSD_SetPreventAllowRemovalHook()</code>	Sets a callback function to prevent/allow removal of storage medium.
<code>USBD_MSD_SetReadWriteHook()</code>	Sets a callback function which gives information about the read and write block-wise operations to the storage medium.
<code>USBD_MSD_Task()</code>	Task that handles the MSD-specific protocol.
<code>USBD_MSD_Poll()</code>	Function which handles MSD commands.
<code>USBD_MSD_SetStartStopUnitHook()</code>	Sets a callback function which is called when the command StartStopUnit is called.
Extended API functions	
<code>USBD_MSD_Connect()</code>	Connects the storage medium to the MSD component.
<code>USBD_MSD_Disconnect()</code>	Disconnects the storage medium from the MSD.
<code>USBD_MSD_RequestDisconnect()</code>	Sets the DisconnectRequest flag.
<code>USBD_MSD_RequestRefresh()</code>	Performs a disconnect (optional), a detach and optionally a re-attach, to inform host that volume contents has changed.
<code>USBD_MSD_UpdateWriteProtect()</code>	This function updates the write protect status of the storage medium.
<code>USBD_MSD_WaitForDisconnection()</code>	Waits for disconnection while time out is not reached.
Data structures	
<code>USB_MSD_INIT_DATA</code>	emUSB-Device-MSD initialization structure that is required when adding an MSD interface.
<code>USB_MSD_INFO</code>	emUSB-Device-MSD storage interface.
<code>USB_MSD_INST_DATA</code>	USB-MSD initialization structure that is required when adding an MSD interface.
<code>PREVENT_ALLOW_REMOVAL_HOOK</code>	Callback function to prevent/allow removal of storage medium.
<code>READ_WRITE_HOOK</code>	Callback function which is called with every read/write access to the storage medium.

Function	Description
USB_MSD_INST_DATA_DRIVER	USB-MSD initialization structure that is required when adding an MSD interface.
USB_MSD_STORAGE_API	USB-MSD initialization structure that is required when adding an MSD interface.
START_STOP_UNIT_HOOK	Callback function which is called when a START STOP UNIT SCSI command is received.

7.3.1 API functions

7.3.1.1 USBD_MSD_Add()

Description

Adds an MSD-class interface to the USB stack.

Prototype

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

Parameters

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

Additional information

After the initialization of general emUSB-Device, this is the first function that needs to be called when an MSD interface is used with emUSB-Device. The structure `USB_MSD_INIT_DATA` must be initialized before `USBD_MSD_Add()` is called.

7.3.1.2 USBD_MSD_AddUnit()

Description

Adds a mass storage device to emUSB-Device-MSD.

Prototype

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

Parameters

Parameter	Description
<code>pInstData</code>	Pointer to a <code>USB_MSD_INST_DATA</code> structure containing the information of the added storage device.

Additional information

It is necessary to call this function immediately after `USBD_MSD_Add()`. It will then add an R/W storage device to emUSB-Device-MSD. The structure `USB_MSD_INST_DATA` must be initialized before calling `USBD_MSD_AddUnit()`.

7.3.1.3 USBD_MSD_AddCDRom()

Description

Adds a CD-ROM device to emUSB-Device-MSD.

Prototype

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

Parameters

Parameter	Description
<code>pInstData</code>	Pointer to a <code>USB_MSD_INST_DATA</code> structure containing the information of the added storage device.

Additional information

Similar to `USB_MSD_AddUnit()`, this function should be called after `USB_MSD_Add()`. The structure `USB_MSD_INST_DATA` must be initialized before `USB_MSD_AddCDRom()` is called.

7.3.1.4 USBD_MSD_SetPreventAllowRemovalHook()

Description

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

Prototype

```
void USBD_MSD_SetPreventAllowRemovalHook
    (U8 Lun,
     PREVENT_ALLOW_REMOVAL_HOOK * pfOnPreventAllowRemoval);
```

Parameters

Parameter	Description
Lun	Logical Unit Number. Using only one storage medium, this parameter is 0.
pfOnPreventAllowRemoval	Pointer to the callback function that shall be called.

Additional information

The callback is called within the MSD task context. The callback must not block.

7.3.1.5 USBD_MSD_SetReadWriteHook()

Description

Sets a callback function which gives information about the read and write block-wise operations to the storage medium.

Prototype

```
void USBD_MSD_SetReadWriteHook(U8 Lun,
                                READ_WRITE_HOOK * pfOnReadWrite);
```

Parameters

Parameter	Description
Lun	Zero-based index for the unit number. Using only one storage medium, this parameter is 0.
pfOnReadWrite	Pointer to the callback function that shall be called.

7.3.1.6 USBD_MSD_Task()

Description

Task that handles the MSD-specific protocol.

Prototype

```
void USBD_MSD_Task(void);
```

Additional information

After the USB device has been successfully enumerated and configured, the `USB-D_MSD_Task()` should be called. This function blocks until the device is detached or is suspended. After a detach or suspend `USB-D_MSD_Task()` will return.

Check `USB-D_MSD_Poll()` if you need a non-blocking version.

7.3.1.7 USBD_MSD_Poll()

Description

Function which handles MSD commands. Using this function is only necessary if you want to avoid using the blocking `USB_MSD_Task` function. This can be necessary if you are not using an RTOS.

Prototype

```
int USBD_MSD_Poll(void);
```

Return value

- 2 O.K. Command was processed, but a protocol error occurred.
- 1 O.K. Command was processed successfully.
- 0 O.K. Timeout occurred.
 - 1 An error occurred. (e.g. no cable connected).

Additional information

This function must be called periodically, otherwise the host can time out the device since it does not process commands. It normally blocks for `USB_MSD_POLL_TIMEOUT` milliseconds while waiting for a command from the host. Should a command arrive during the timeout it will be processed, which could potentially increase the block duration. The duration could also decrease because the function returns as soon as a command is finished.

In case of return value 2 the calling task should still call this function again, if possible recovery will be initiated.

7.3.1.8 USBD_MSD_SetStartStopUnitHook()

Description

Sets a callback function which is called when the command StartStopUnit is called.

Prototype

```
void USBD_MSD_SetStartStopUnitHook(U8 Lun,
START_STOP_UNIT_HOOK * pfOnStartStopUnit);
```

Parameters

Parameter	Description
Lun	Zero-based index for the unit number. Using only one storage medium, this parameter is 0.
pfOnStartStopUnit	Pointer to the callback function that shall be called. For detailed information about the function pointer, refer to START_STOP_UNIT_HOOK.

7.3.2 Extended API functions

7.3.2.1 USBD_MSD_Connect()

Description

Connects the storage medium to the MSD component.

Prototype

```
void USBD_MSD_Connect(U8 Lun);
```

Parameters

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

Additional information

The storage medium is initially always connected to the MSD component. This function is normally used after the storage medium was disconnected via `USB_MSD_Disconnect()` to carry out file system operations on the device application side. Because the device can not actively perform a connect operation this function sets an internal flag and the next time when the host requests the status of the storage medium the storage medium is connected back to the MSD component.

7.3.2.2 USBD_MSD_Disconnect()

Description

Disconnects the storage medium from the MSD.

Prototype

```
void USBD_MSD_Disconnect(U8 Lun);
```

Parameters

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

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 the device to the host, the function `USB_D_MSD_Connect()` shall be used. See `USB_D_MSD_RequestDisconnect()` and `USB_D_MSD_WaitForDisconnection()` for a graceful disconnection method.

7.3.2.3 USBD_MSD_RequestDisconnect()

Description

Sets the DisconnectRequest flag.

Prototype

```
void USBD_MSD_RequestDisconnect(U8 Lun);
```

Parameters

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

Additional information

This function sets the 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()` shall be called.

Notes

If the host tries to access the storage medium while this flag is set to 1, the status of the storage medium changes to disconnected.

7.3.2.4 USBD_MSD_RequestRefresh()

Description

Performs a disconnect (optional), a detach and optionally a re-attach, to inform host that volume contents has changed.

Prototype

```
void USBD_MSD_RequestRefresh(U8 Lun,
                             U32 Flags);
```

Parameters

Parameter	Description
Lun	Zero-based index for the unit number. Using only one storage medium, this parameter is 0.
Flags	Request flags, a bit-ored combination of the following flags: <ul style="list-style-type: none"> USB_MSD_TRY_DISCONNECT - Try a medium disconnect before doing a USB detach. USB_MSD_RE_ATTACH - Automatically re-attach after detach has been done.

Additional information

If the flag `USB_MSD_TRY_DISCONNECT` is given, the function sets the 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. If the host acknowledges the disconnect, the medium is reconnected and the function `USBD_MSD_Task()` will return.

If the flag `USB_MSD_TRY_DISCONNECT` is not set or the host ignores the disconnection of the medium, the USB device is detached from the host (using `USB_Stop()`).

If the flag `USB_MSD_RE_ATTACH` is set, the device is re-attached after some delay using `USB_Start()`. Then the function `USBD_MSD_Task()` will return. The function `USB_D_MSD_RequestRefresh()` returns immediately while the procedure is executed in the `USB_D_MSD_Task()`.

Returning of the function `USBD_MSD_Task()` allows the application to reinitialize the volume (or calling `USB_Start()`, if `USB_MSD_RE_ATTACH` was not set) before calling `USB_D_MSD_Task()` again.

Detaching the USB device not only affects the specified volume ([Lun](#)) but all volumes of the device and any other USB class interfaces.

7.3.2.5 USBD_MSD_UpdateWriteProtect()

Description

This function updates the write protect status of the storage medium.

Prototype

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

Parameters

Parameter	Description
<code>Lun</code>	Zero-based index for the unit number. Using only one storage medium, this parameter is 0.
<code>IsWriteProtected</code>	Set the write protect flag: 1 - Medium is write-protected. 0 - Medium is NOT write-protected.

Additional information

Please make sure that this function is called when the LUN is disconnected from the host, otherwise the change in the WriteProtected flag is normally not recognized.

7.3.2.6 USBD_MSD_WaitForDisconnection()

Description

Waits for disconnection while time out is not reached.

Prototype

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

Parameters

Parameter	Description
Lun	Zero-based index for the unit number. Using only one storage medium, this parameter is 0.
TimeOut	Timeout give in ms. How long should this function wait, until it stops waiting.

Return value

- 0 Error - Time out reached. Device not disconnected.
- 1 Success - Device disconnected.

Additional information

After triggering the disconnection via `USBD_MSD_RequestDisconnect()` the stack disconnects the storage medium as soon as the host requests the status of the storage medium. Win2k does not periodically check the status of a USB MSD. Therefore, the timeout is required to leave the loop. The return value can be used to decide if the disconnection should be forced. In this case, `USBD_MSD_Disconnect()` shall be called.

7.3.3 Data structures

7.3.3.1 USB_MSD_INIT_DATA

Description

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

Type definition

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

Structure members

Member	Description
EPIn	Bulk IN endpoint for sending data to the host.
EPOut	Bulk OUT endpoint for receiving data from the host.
InterfaceNum	Interface number. This member is used internally, set to 0.

Additional information

This structure holds the endpoints that should be used with the MSD interface. Refer to `USBD_AddEP()` for more information about how to add an endpoint.

7.3.3.2 USB_MSD_INFO

Description

emUSB-Device-MSD storage interface.

Type definition

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

Structure members

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

7.3.3.3 USB_MSD_INST_DATA

Description

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

Type definition

```
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;
    const USB_MSD_LUN_INFO    * pLunInfo;
} USB_MSD_INST_DATA;
```

Structure members

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 USB_MSD_INST_DATA_DRIVER for detailed information about how to initialize this structure.
DeviceType	Determines the type of the device: 0: Direct access block device 5: CD/DVD
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.
IsWriteProtected	Specifies whether the storage medium shall be write-protected.
pLunInfo	Pointer to a USB_MSD_LUN_INFO structure. Filling this structure is mandatory for each LUN.

Additional information

All non-optional members of this structure need to be initialized correctly, except [DeviceType](#) and [pfHandleCmd](#) because it is done by the functions [USBD_MSD_AddUnit\(\)](#) or [USBD_MSD_AddCDROM\(\)](#).

7.3.3.4 USB_MSD_LUN_INFO

Description

Structure that is used when adding a logical volume to emUSB-Device-MSD.

Type definition

```
typedef struct {
    const char * pVendorName;
    const char * pProductName;
    const char * pProductVer;
    const char * pSerialNo;
} USB_MSD_LUN_INFO;
```

Structure members

Member	Description
<code>pVendorName</code>	Vendor name of the mass storage device. The string should be no longer than 8 bytes.
<code>pProductName</code>	Product name of the mass storage device. The product name string should be no longer than 16 bytes.
<code>pProductVer</code>	Product version number of the mass storage device. The product version string should be no longer than 4 bytes.
<code>pSerialNo</code>	Product serial number of the mass storage device. The serial number string must be exactly 12 bytes, in order to satisfy the USB bootability specification requirements.

Additional information

The setting of these values is mandatory, if these values remain `NULL` at initialisation emUSB-Device will report a panic error in debug builds (`USB_PANIC`).

7.3.3.5 PREVENT_ALLOW_REMOVAL_HOOK

Description

Callback function to prevent/allow removal of storage medium. See `USBD_MSD_SetPreventAllowRemovalHook()`.

Type definition

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

Parameters

Parameter	Description
<code>PreventRemoval</code>	Show whether the device shall be locked or not. <ul style="list-style-type: none">• 0 - The device shall be removable.• 1 - The device shall be locked.

Additional information

Most OSes call the prevent/allow removal before any write operation. This callback will be called for all LUNs that are available on the host.

7.3.3.6 READ_WRITE_HOOK

Description

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

Type definition

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

Parameters

Parameter	Description
Lun	Specifies the logical unit number which was accessed through read or write.
IsRead	Specifies whether a read or a write access was used: <ul style="list-style-type: none"> • 1 : read • 0 : write
OnOff	States whether the read or write request has been initialized (1) or whether it is complete (0).
StartLBA	The first Logical Block Address accessed by the transfer.
NumBlocks	The number of blocks accessed by the transfer, starting from the StartLBA .

7.3.3.7 USB_MSD_INST_DATA_DRIVER

Description

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

Type definition

```
typedef struct {
    void * pStart;
    U32    StartSector;
    U32    NumSectors;
    U16    SectorSize;
    void * pSectorBuffer;
    unsigned NumBytes4Buffer;
    U8     NumBuffers;
} USB_MSD_INST_DATA_DRIVER;
```

Structure members

Member	Description
<code>pStart</code>	A pointer defining the start address
<code>StartSector</code>	The start sector that is used for the driver.
<code>NumSectors</code>	The available number of sectors available for the driver.
<code>SectorSize</code>	The sector size that should be used by the driver.
<code>pSectorBuffer</code>	Pointer to an application provided buffer to be used as temporary buffer for storing the sector data.
<code>NumBytes4Buffer</code>	Size of the application provided buffer.
<code>NumBuffers</code>	Number of buffer that are available. This is only used when using the MT storage layer.

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 members of `USB_MSD_INST_DATA_DRIVER` have the following meaning:

`USB_MSD_StorageRAM:`

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

`USB_MSD_StorageByName:`

Member	Description
<code>pStart</code>	Pointer to a string holding the name of the volumes that shall be used, for example "nand:" "mmc:1:"
<code>StartSector</code>	Specifies the start sector.
<code>NumSectors</code>	Number of sector that shall be used.
<code>SectorSize</code>	This member is ignored.
<code>pSectorBuffer</code>	Pointer to an application provided buffer to be used as temporary buffer for storing the sector data
<code>NumBytes4Buffer</code>	Size of the buffer provided by the application. Please make sure that the buffer can at least 3 sectors otherwise, <code>pSec-</code>

Member	Description
	<code>torBuffer</code> and <code>NumBytes4Buffer</code> are ignored and an internal sector buffer is used. This sector-buffer is then allocated by using the FS-Storage-Layer functions.

7.3.3.8 USB_MSD_STORAGE_API

Description

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

Type definition

```
typedef struct {
    USB_MSD_STORAGE_INIT          * pfInit;
    USB_MSD_STORAGE_GETINFO      * pfGetInfo;
    USB_MSD_STORAGE_GETREADBUFFER * pfGetReadBuffer;
    USB_MSD_STORAGE_READ         * pfRead;
    USB_MSD_STORAGE_GETWRITEBUFFER * pfGetWriteBuffer;
    USB_MSD_STORAGE_WRITE        * pfWrite;
    USB_MSD_STORAGE_MEDIUMISPRESENT * pfMediumIsPresent;
    USB_MSD_STORAGE_DEINIT       * pfDeInit;
} USB_MSD_STORAGE_API;
```

Structure members

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	De-initializes the storage medium.

Additional information

`USB_MSD_STORAGE_API` is used to retrieve information from the storage device driver or access data that needs to be read or written. Detailed information can be found in *MSD Storage Driver* on page 251.

7.3.3.9 START_STOP_UNIT_HOOK

Description

Callback function which is called when a START STOP UNIT SCSI command is received.

Type definition

```
typedef void (START_STOP_UNIT_HOOK)(U8 Lun,  
                                   U8 StartLoadEject);
```

Parameters

Parameter	Description
Lun	Specifies the logical unit number.
StartLoadEject	Specifies which operation is executed by the host: <ul style="list-style-type: none">• 0 : Stop disk• 1 : Start disk and make ready for access• 2 : Eject disk if permitted• 3 : Load, start and make disk ready.

7.4 MSD Storage Driver

7.4.1 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.

7.4.1.1 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.

7.4.1.2 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.

Note

If you are not using emFile or the RAM driver you will have to provide your own sector write/read routines for your storage medium.

7.4.2 Interface function list

As described above, access to a storage medium is realized through an API-function table (`USB_MSD_STORAGE_API`). The storage functions are declared in `USB_MSD.h`.

7.4.3 USB_MSD_STORAGE_API in detail

7.4.3.1 USB_MSD_STORAGE_INIT

Description

Initializes the storage medium.

Type definition

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

Parameters

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
pDriverData	Pointer to a USB_MSD_INST_DATA_DRIVER structure that contains all information that is necessary for the driver initialization. Refer to USB_MSD_INST_DATA_DRIVER structure for detailed information.

7.4.3.2 USB_MSD_STORAGE_GETINFO

Description

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

Type definition

```
typedef void (USB_MSD_STORAGE_GETINFO) (U8          Lun,  
                                         USB_MSD_INFO * pInfo);
```

Parameters

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
pInfo	Pointer to a USB_MSD_INFO structure. For detailed information about the USB_MSD_INFO structure, refer to USB_MSD_INFO.

7.4.3.3 USB_MSD_STORAGE_GETREADBUFFER

Description

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

Type definition

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

Parameters

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

Return value

Maximum number of consecutive sectors that can be read at once by the driver.

7.4.3.4 USB_MSD_STORAGE_READ

Description

Reads one or multiple consecutive sectors from the storage medium.

Type definition

```
typedef I8 (USB_MSD_STORAGE_READ)(U8      Lun,
                                   U32      SectorIndex,
                                   void *    pData,
                                   U32      NumSectors);
```

Parameters

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

Return value

= 0 Success.
 ≠ 0 Failed.

7.4.3.5 USB_MSD_STORAGE_GETWRITEBUFFER

Description

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

Type definition

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

Parameters

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

Return value

Maximum number of consecutive sectors that can be written into the buffer.

7.4.3.6 USB_MSD_STORAGE_WRITE

Description

Writes one or more consecutive sectors to the storage medium.

Type definition

```
typedef I8 (USB_MSD_STORAGE_WRITE)(
    U8      Lun,
    U32     SectorIndex,
    const void * pData,
    U32     NumSectors);
```

Parameters

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.

Return value

= 0 Success.
 ≠ 0 Failed.

7.4.3.7 USB_MSD_STORAGE_MEDIUMISPRESNT

Description

Checks if medium is present.

Type definition

```
typedef I8 (USB_MSD_STORAGE_MEDIUMISPRESNT)(U8 Lun);
```

Parameters

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

Return value

- 1 Medium is present.
- 0 Medium is not present.

7.4.3.8 USB_MSD_STORAGE_DEINIT

Description

De-initializes the storage medium.

Type definition

```
typedef void (USB_MSD_STORAGE_DEINIT)(U8 Lun);
```

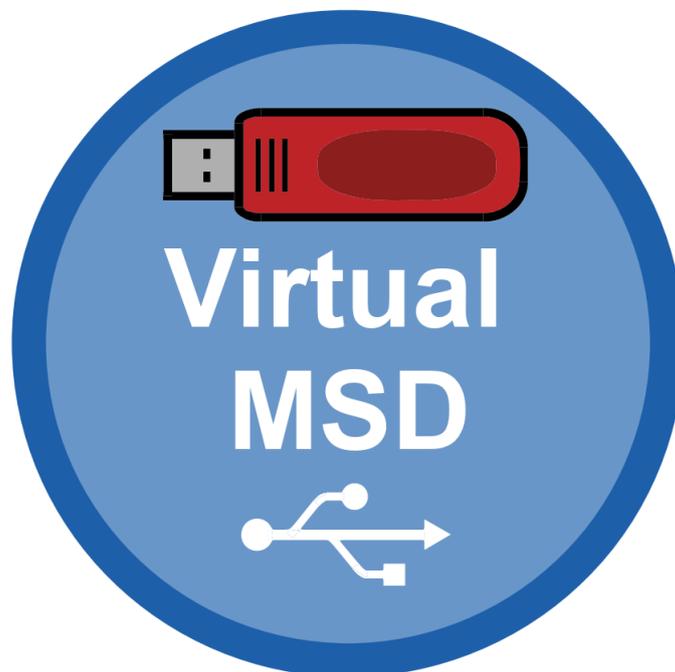
Parameters

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

Chapter 8

Virtual Mass Storage Component (VirtualMSD)

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



8.1 Overview

The VirtualMSD component allows to easily stream files to and from USB devices. Once the USB device is connected to the host, files can be read or written to the application without the need for dedicated storage memory.

This makes the software very flexible: it can be used for various types of applications and purposes, with no additional software or drivers necessary on the host side.

The VirtualMSD software analyzes what operation is performed by the host and passes this to the application layer of the embedded target, which then performs the appropriate action. A simple drag and drop is all it takes to initialize this process, which is supported by a unique active file technology.

Virtual MSD can access all data which has been created prior to the device being attached to the host, live data cannot be provided.

VirtualMSD allows to use the storage device in a virtual manner, which means data does not need to be stored on a physical medium.

The storage device will be shown on the host as a FAT formatted volume with a configurable size and a configurable file list.

With the help of that virtual function, the target device can be used for different applications by simply dragging and dropping files to and from the storage medium:

- Firmware update application.
- Configuration updater.
- File system firewall - protect the target's filesystem from being manipulated by the host.

The component itself is based on MSD class and thus can be used on virtually any OS such as any Windows, macOS or any Linux distribution (including Android) which supports MSD, without installing any third party tools.

8.2 Configuration

8.2.1 Initial configuration

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

8.2.2 Final configuration

The configuration must only be modified if emUSB-Device is deployed in your final product. Refer to *emUSB-Device Configuration* on page 48 for detailed information about the generic information functions which must be adapted.

8.2.3 Class specific configuration functions

For basic configuration please refer to the MSD chapter *MSD class specific configuration functions* on page 223. In addition to the MSD configuration functions described there the following VirtualMSD functions are available.

Function	Description
emUSB-Device-VirtualMSD configuration functions	
USB_VMSD_X_Config()	Configures the VirtualMSD component.

8.2.3.1 USB_VMSD_X_Config()

Description

Main user configuration function of the VirtualMSD component. This function is provided by the user.

Prototype

```
void USB_VMSD_X_Config(void);
```

Example

```
void USB_VMSD_X_Config(void) {
    //
    // String information used when inquiring the volume.
    //
    static const USB_MSD_LUN_INFO _LunInfo = {
        "Vendor",      // MSD VendorName
        "MSD Volume", // MSD ProductName
        "1.00",        // MSD ProductVer
        "134657890"    // MSD SerialNo
    };
    //
    // Global configuration
    //
    USBD_VMSD_AssignMemory(&_aMEMBuffer[0], sizeof(_aMEMBuffer));
    //
    // Setup LUN0
    //
    USBD_VMSD_SetNumSectors(0, 8000);
    USBD_VMSD_SetSectorsPerCluster(0, 32); // Anywhere from 1...128, needs to be
    2^x
    USBD_VMSD_SetNumRootDirSectors(0, 2);
    USBD_VMSD_SetUserAPI(0, &UserFuncAPI);
    USBD_VMSD_SetVolumeInfo(0, "Virt0.MSD", &_LunInfo); // Add volume ID
    //
    // Push const contents to the volume
    //
    USBD_VMSD_AddConstFiles(0, &_aConstFiles[0], COUNTOF(_aConstFiles));
}
```

Additional information

During the call of `USB_VMSD_Add()` this user function is called in order to configure the VirtualMSD module according to the user's preferences. In order to allow the user to configure the volume it is necessary to provide either a memory block or memory allocation/free callbacks to VirtualMSD component.

8.2.4 Running the example application

The directory `Application` contains example applications that can be used with `emUSB-Device` and the VirtualMSD component. To test the VirtualMSD 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.

8.2.5 Calculation of RAM memory usage for VirtualMSD

An application has to provide RAM memory in order to use VirtualMSD either via a call to the function `USB_VirtualMSD_AssignMemory()` or by setting callback functions for memory allocation. The amount of memory used can be calculated as follows:

For each volume:

Purpose	Bytes used	Minimum
Global volume information	128	128
Cluster info for predefined files added with <code>USB_VirtualMSD_AddConstFiles()</code>	2 (for each file)	0
I/O Buffer	512	512
Directory	$m * 512$	512
FAT	$n * 512$	512
Total	-	1664

The number of files that can be stored on the volume depends on the size of the directory which is configured using `USB_VirtualMSD_SetNumRootDirSectors()`:

Number of root directory sectors (m)	Used memory for directory (bytes)	max. number of files with short (8.3) file name
1	512	15
2	1024	31
3	1536	47
4	2048	63
5	2560	79
6	3072	95

Files with long file names may occupy multiple entries in the directory, depending on the actual length.

The number of FAT sectors (n) depends on the virtual size of the volume (configured using `USB_VirtualMSD_SetNumSectors()`) and the number of sectors per cluster:

Number of sectors	Sectors per cluster	Used memory for FAT (bytes)	approx. virtual volume size (MB)
10880	32	512	5.4
21792	32	1024	10.8
32704	32	1536	16.3
43616	32	2048	21.7
54528	32	2560	27.2
65440	32	3072	32.7
76352	32	3584	38.1
87264	32	4096	43.6
98176	32	4608	49.0
109088	32	5120	54.5
120000	32	5632	59.9
130720	32	6144	65.3
43520	128	512	21.3
87168	128	1024	43.5

Number of sectors	Sectors per cluster	Used memory for FAT (bytes)	approx. virtual volume size (MB)
130816	128	1536	65.3
174464	128	2048	87.1
218112	128	2560	108.9
261760	128	3072	130.8
305408	128	3584	152.6
349056	128	4096	174.4
392704	128	4608	196.2
436352	128	5120	218.1
480000	128	5632	239.9
522800	128	6144	261.3

There is no disadvantage of using the maximum possible number of sectors per cluster (128).

In most cases the minimal configuration (FAT = 512 and directory = 512) should be sufficient. It supports a small number of files with a total size of all files up to 21 MB. If more files or bigger files are needed, the required parameters can be looked up in the tables above.

8.3 Target API

Function	Description
API functions	
<code>USBD_VMSD_Add()</code>	Create VirtualMSD volumes and add MSD interface to the device.
User supplied functions	
<code>USB_VMSD_X_Config()</code>	User supplied function that configures all storages of the VMSD component.
Configuration functions	
<code>USBD_VMSD_AssignMemory()</code>	Assigns memory to the VirtualMSD module.
<code>USBD_VMSD_SetUserAPI()</code>	Sets the default user callbacks for the VirtualMSD component.
<code>USBD_VMSD_SetNumRootDirSectors()</code>	Sets the number of sectors which should be used for root directory entries.
<code>USBD_VMSD_SetVolumeInfo()</code>	Sets the volume name for a specified LUN.
<code>USBD_VMSD_AddConstFiles()</code>	Adds constant files to VirtualMSD.
<code>USBD_VMSD_SetNumSectors()</code>	Sets the number of sectors available on the volume.
<code>USBD_VMSD_SetSectorsPerCluster()</code>	Set number of sectors per cluster.
Data structures	
<code>USB_VMSD_CONST_FILE</code>	This structure contains information about a constant file which cannot be changed at run time and should be shown inside the VirtualMSD volume (e.g.
<code>USB_VMSD_USER_FUNC_API</code>	This structure contains the function pointers for user provided functions.
<code>USB_VMSD_FILE_INFO</code>	Structure used in the read and write callbacks.
<code>USB_VMSD_DIR_ENTRY_SHORT</code>	Structure used to describe an entry with a short file name.
Function definitions	
<code>USB_VMSD_ON_READ_FUNC</code>	Callback function prototype that is used when calling the <code>USBD_VMSD_SetUserAPI()</code> function.
<code>USB_VMSD_ON_WRITE_FUNC</code>	Callback function prototype that is used when calling the <code>USBD_VMSD_SetUserAPI()</code> function.
<code>USB_VMSD_MEM_ALLOC</code>	Function prototype that is used when memory is being allocated by the VirtualMSD module.
<code>USB_VMSD_MEM_FREE</code>	Function prototype that is used when memory is being freed by the VirtualMSD module.

8.3.1 API functions

8.3.1.1 USBD_VMSD_Add()

Description

Create VirtualMSD volumes and add MSD interface to the device.

Prototype

```
void USBD_VMSD_Add(void);
```

Additional information

After the initialization of emUSB-Device, this is the first function that needs to be called when the VirtualMSD component is used with emUSB-Device. During the call of the said function the user function `USB_VMSD_X_Config()` is called in order to configure the storage itself.

8.3.1.2 USB_VMSD_X_Config()

Description

User supplied function that configures all storages of the VMSD component.

Prototype

```
void USB_VMSD_X_Config(void);
```

Additional information

This function is called automatically by `USBD_VMSD_Add()` in order to allow to configure the storage volumes that VirtualMSD should show after configuration.

Only the following functions must be called in this context:

Allowed functions with <code>USB_X_VMSD_Config</code> :
<code>USBD_VMSD_AssignMemory()</code>
<code>USBD_VMSD_SetUserAPI()</code>
<code>USBD_VMSD_SetNumRootDirSectors()</code>
<code>USBD_VMSD_SetVolumeInfo()</code>
<code>USBD_VMSD_AddConstFiles()</code>
<code>USBD_VMSD_SetNumSectors()</code>
<code>USBD_VMSD_SetSectorsPerCluster()</code>

8.3.1.3 USBD_VMSD_AssignMemory()

Description

Assigns memory to the VirtualMSD module.

Prototype

```
void USBD_VMSD_AssignMemory(U32 * p,  
                             U32  NumBytes);
```

Parameters

Parameter	Description
p	Pointer to the memory which should be dedicated to VirtualMSD.
NumBytes	Size of the memory block in bytes.

Additional information

See *Calculation of RAM memory usage for VirtualMSD* on page 264.

8.3.1.4 USBD_VMSD_SetUserAPI()

Description

Sets the default user callbacks for the VirtualMSD component.

Prototype

```
void USBD_VMSD_SetUserAPI(const USB_VMSD_USER_FUNC_API * pUserFunc);
```

Parameters

Parameter	Description
<code>pUserFunc</code>	Pointer to a <code>USB_VMSD_USER_FUNC_API</code> structure which holds the default function pointers for multiple functions.

Notes

(1) Must only be called from `USB_VMSD_X_Config()` during initialization phase

8.3.1.5 USBD_VMSD_SetNumRootDirSectors()

Description

Sets the number of sectors which should be used for root directory entries.

Prototype

```
void USBD_VMSD_SetNumRootDirSectors(unsigned Lun,
                                     unsigned NumRootDirSectors);
```

Parameters

Parameter	Description
Lun	Specifies the logical unit number.
NumRootDirSectors	Number of sectors to be reserved for the root directory entries.

Additional information

The number of sectors reserved through this function is subtracted from the number of sectors configured by `USB_D_VMSD_SetNumSectors()`. These sectors hold the root directory entries for the specified LUN. A single sector contains 512 bytes, a short file name entry (also called 8.3 filenames) needs 32 bytes, therefore a single sector has enough space for 16 root directory entries. Please note that when using LFN (long file names) the number of entries required for a single file is dynamic (depending on the length of the file name).

Notes

(1) Must only be called from `USB_VMSD_X_Config()` during initialization phase

8.3.1.6 USBD_VMSD_SetVolumeInfo()

Description

Sets the volume name for a specified LUN.

Prototype

```
int USBD_VMSD_SetVolumeInfo(    unsigned        Lun,
                                const char        * sVolumeName,
                                const USB_MSD_LUN_INFO * pLunInfo);
```

Parameters

Parameter	Description
Lun	Specifies the logical unit number.
sVolumeName	Pointer to a string containing the name of the LUN.
pLunInfo	Pointer to USB_MSD_LUN_INFO structure contain all relevant MSD strings.

Return value

≥ 0 O.K.
 < 0 Error

Notes

(1) Must only be called from `USB_VMSD_X_Config()` during initialization phase

8.3.1.7 USBD_VMSD_AddConstFiles()

Description

Adds constant files to VirtualMSD. Allows to add multiple files which should be shown on a VirtualMSD volume as soon as it is connected. A common example would be a "Readme.txt" or a link to the company website.

Prototype

```
int USBD_VMSD_AddConstFiles(    unsigned          Lun,
                               const USB_VMSD_CONST_FILE * paConstFile,
                               unsigned          NumFiles);
```

Parameters

Parameter	Description
Lun	Specifies the logical unit number.
paConstFile	Pointer to an array of USB_VMSD_CONST_FILE structures.
NumFiles	The number of items in the paConstFile array.

Return value

≥ 0 O.K.
 < 0 Error

Additional information

For additional information please see USB_VMSD_CONST_FILE.

Notes

(1) Must only be called from USB_VMSD_X_Config() during initialization phase

Example

```
#define COUNTOF(a)      (sizeof(a)/sizeof(a)[0])
static const U8 _abFile_SeggerHTML[] = {0x3C, 0x68, 0x74, 0x6D, 0x6C, 0x3E, 0x3C,
0x68, 0x65, 0x61, 0x64, 0x3E, 0x3C, 0x6D, 0x65, 0x74, 0x61, 0x20, 0x68, 0x74,
0x70, 0x2D, 0x65, 0x71, 0x75, 0x69, 0x76, 0x3D, 0x22, 0x72, 0x65, 0x66, 0x72, 0x65,
0x73, 0x68, 0x22, 0x20, 0x63, 0x6F, 0x6E, 0x74, 0x65, 0x6E, 0x74, 0x3D, 0x22, 0x30,
0x3B, 0x20, 0x75, 0x72, 0x6C, 0x3D, 0x68, 0x74, 0x74, 0x70, 0x3A, 0x2F, 0x2F, 0x77,
0x77, 0x77, 0x2E, 0x73, 0x65, 0x67, 0x67, 0x65, 0x72, 0x2E, 0x63, 0x6F, 0x6D, 0x2F,
0x69, 0x6E, 0x64, 0x65, 0x78, 0x2E, 0x68, 0x74, 0x6D, 0x6C, 0x22, 0x2F, 0x3E, 0x3C,
0x74, 0x69, 0x74, 0x6C, 0x65, 0x3E, 0x53, 0x45, 0x47, 0x47, 0x45, 0x52, 0x20, 0x53,
0x68, 0x6F, 0x72, 0x74, 0x63, 0x75, 0x74, 0x3C, 0x2F, 0x74, 0x69, 0x74, 0x6C, 0x65,
0x3E, 0x3C, 0x2F, 0x68, 0x65, 0x61, 0x64, 0x3E, 0x3C, 0x62, 0x6F, 0x64, 0x79, 0x3E,
0x3C, 0x2F, 0x62, 0x6F, 0x64, 0x79, 0x3E, 0x3C, 0x2F, 0x68, 0x74, 0x6D, 0x6C, 0x3E};
static USB_VMSD_CONST_FILE _aConstFiles[] = {
// sName          pData          FileSize          Flags
{ "Segger.html",  _abFile_SeggerHTML,  sizeof(_abFile_SeggerHTML),  0, }
};
/*****
 *
 *      USB_VMSD_X_Config
 *
 *      Function description
 *      This function is called by the USB MSD Module during USB_VMSD_Init() and
 *      initializes the VirtualMSD volume.
 */
void USB_VMSD_X_Config(void) {
<...>
    USBD_VMSD_AddConstFiles(1, &_aConstFiles[0], COUNTOF(_aConstFiles));
<...>
}
```

8.3.1.8 USBD_VMSD_SetNumSectors()

Description

Sets the number of sectors available on the volume.

Prototype

```
void USBD_VMSD_SetNumSectors(unsigned Lun,  
                             unsigned NumSectors);
```

Parameters

Parameter	Description
Lun	Specifies the logical unit number.
NumSectors	Specifies the number of sectors for a LUN.

Notes

(1) Must only be called from `USB_VMSD_X_Config()` during initialization phase

8.3.1.9 USBD_VMSD_SetSectorsPerCluster()

Description

Set number of sectors per cluster.

Prototype

```
void USBD_VMSD_SetSectorsPerCluster(unsigned Lun,  
                                     unsigned SectorsPerCluster);
```

Parameters

Parameter	Description
Lun	Specifies the logical unit number.
SectorsPerCluster	Number of sectors per cluster for the LUN.

Additional information

[SectorsPerCluster](#) can be anywhere between 1 and 128, but needs to be a power of 2. Larger clusters save memory because the management overhead is lower, but the maximum number of files is limited by the number of available clusters.

Notes

(1) Must only be called from `USB_VMSD_X_Config()` during initialization phase

8.3.2 Data structures

8.3.2.1 USB_VMSD_CONST_FILE

Description

This structure contains information about a constant file which cannot be changed at run time and should be shown inside the VirtualMSD volume (e.g. Readme.txt). This structure is a parameter for the `USBD_VMSD_AddConstFiles()` function.

Type definition

```
typedef struct {
    const char * sName;
    const U8    * pData;
    unsigned    FileSize;
    U32         Flags;
} USB_VMSD_CONST_FILE;
```

Structure members

Member	Description
<code>sName</code>	Pointer to a zero-terminated string containing the filename.
<code>pData</code>	Pointer to the file data. Can be <code>NULL</code> .
<code>FileSize</code>	Size of the file. Normally the size of the data pointed to by <code>pData</code> .
<code>Flags</code>	Can be one of the following items: <ul style="list-style-type: none"> <code>USB_VMSD_FILE_WRITABLE</code>: The file is writable <code>USB_VMSD_FILE_AHEAD</code>: File is located at the start of the volume. Normally constant files are allocated at the end of the volume.

Additional information

If a file does not occupy complete sectors the remaining bytes of the last sector are automatically filled with 0s on read. If `pData` is `NULL` the file is not displayed in the volume. This is useful when the application has certain files which should only be displayed after certain events (e.g. the application displays a Fail.txt when the device is reconnected after an unsuccessful firmware update).

8.3.2.2 USB_VMSD_USER_FUNC_API

Description

This structure contains the function pointers for user provided functions. This structure is a parameter for the `USBD_VMSD_SetUserAPI()` function.

Type definition

```
typedef struct {
    USB_VMSD_ON_READ_FUNC * pfOnReadSector;
    USB_VMSD_ON_WRITE_FUNC * pfOnWriteSector;
    USB_VMSD_MEM_ALLOC * pfMemAlloc;
    USB_VMSD_MEM_FREE * pfMemFree;
} USB_VMSD_USER_FUNC_API;
```

Structure members

Member	Description
<code>pfOnReadSector</code>	Pointer to a callback function of type <code>USB_VMSD_ON_READ_FUNC</code> which is called when a sector is read from the host. This function is mandatory and can not be <code>NULL</code> .
<code>pfOnWriteSector</code>	Pointer to a callback function of type <code>USB_VMSD_ON_WRITE_FUNC</code> which is called when a sector is written from the host. This function is mandatory and can not be <code>NULL</code> .
<code>pfMemAlloc</code>	Pointer to a user provided alloc function of type <code>USB_VMSD_MEM_ALLOC</code> . If this pointer is <code>NULL</code> the internal alloc function is called. If no memory block is assigned <code>USB_PANIC()</code> is called.
<code>pfMemFree</code>	Pointer to a user provided free function of type <code>USB_VMSD_MEM_FREE</code> . If this pointer is <code>NULL</code> the internal free function is called.

8.3.2.3 USB_VMSD_FILE_INFO

Description

Structure used in the read and write callbacks.

Type definition

```
typedef struct {  
    const USB_VMSD_DIR_ENTRY_SHORT * pDirEntry;  
} USB_VMSD_FILE_INFO;
```

Structure members

Member	Description
pDirEntry	Pointer to a USB_VMSD_DIR_ENTRY_SHORT structure.

Additional information

Check `USB_VMSD_ON_READ_FUNC`, `USB_VMSD_ON_WRITE_FUNC` and `USB_VMSD_DIR_ENTRY_SHORT` for more information.

8.3.2.4 USB_VMSD_DIR_ENTRY_SHORT

Description

Structure used to describe an entry with a short file name. This structure is a member of `USB_VMSD_DIR_ENTRY`.

Type definition

```
typedef struct {
    U8  acFilename[];
    U8  acExt[];
    U8  DirAttr;
    U8  NTRes;
    U8  CrtTimeTenth;
    U16 CrtTime;
    U16 CrtDate;
    U16 LstAccDate;
    U16 FstClusHI;
    U16 WrtTime;
    U16 WrtDate;
    U16 FstClusLO;
    U32 FileSize;
} USB_VMSD_DIR_ENTRY_SHORT;
```

Structure members

Member	Description
<code>acFilename</code>	File name, limited to 8 characters (short file name), padded with spaces (0x20).
<code>acExt</code>	File extension, limited to 3 characters (short file name), padded with spaces (0x20).
<code>DirAttr</code>	File attributes. Available attributes are listed below.
<code>NTRes</code>	Reserved for use by Windows NT.
<code>CrtTimeTenth</code>	Millisecond stamp at file creation time. This field actually contains a count of tenths of a second.
<code>CrtTime</code>	Creation time.
<code>CrtDate</code>	Date file was created.
<code>LstAccDate</code>	Last access date. Note that there is no last access time, only a date. This is the date of last read or write.
<code>FstClusHI</code>	High word of this entry's first cluster number.
<code>WrtTime</code>	Time of last write.
<code>WrtDate</code>	Date of last write.
<code>FstClusLO</code>	Low word of this entry's first cluster number.
<code>FileSize</code>	File size in bytes.

Additional information

The following file attributes are available for short dir entries:

Attribute	Explanation
<code>USB_VMSD_ATTR_READ_ONLY</code>	The file is read-only.
<code>USB_VMSD_ATTR_HIDDEN</code>	The file is hidden.
<code>USB_VMSD_ATTR_SYSTEM</code>	The file is designated as a system file.
<code>USB_VMSD_ATTR_VOLUME_ID</code>	This entry is the volume ID (volume name).
<code>USB_VMSD_ATTR_DIRECTORY</code>	The file is a directory.

Attribute	Explanation
USB_VMSD_ATTR_ARCHIVE	The file has the archive attribute.
USB_VMSD_ATTR_LONG_NAME	The file has a long file name.

8.3.3 Function definitions

8.3.3.1 USB_VMSD_ON_READ_FUNC

Description

Callback function prototype that is used when calling the `USBD_VMSD_SetUserAPI()` function.

Type definition

```
typedef int (USB_VMSD_ON_READ_FUNC)(
    unsigned Lun,
    U8 * pData,
    U32 Off,
    U32 NumBytes,
    const USB_VMSD_FILE_INFO * pFile);
```

Parameters

Parameter	Description
<code>Lun</code>	Zero-based index for the unit number. Using only one virtual volume, this parameter is 0.
<code>pData</code>	Pointer to a buffer in which the data is stored.
<code>Off</code>	Offset in the file which is read by the host.
<code>NumBytes</code>	Amount of bytes requested by the host.
<code>pFile</code>	Pointer to a <code>USB_VMSD_FILE_INFO</code> structure describing the file.

Return value

= 0 Success.
 ≠ 0 An error occurred.

8.3.3.2 USB_VMSD_ON_WRITE_FUNC

Description

Callback function prototype that is used when calling the `USBD_VMSD_SetUserAPI()` function.

Type definition

```
typedef int (USB_VMSD_ON_WRITE_FUNC)(
    unsigned Lun,
    const U8 * pData,
    U32 Off,
    U32 NumBytes,
    const USB_VMSD_FILE_INFO * pFile);
```

Parameters

Parameter	Description
<code>Lun</code>	Zero-based index for the unit number. Using only one virtual volume, this parameter is 0.
<code>pData</code>	Pointer to the data to be written (received from the host).
<code>Off</code>	Offset in the file which the host writes.
<code>NumBytes</code>	Amount of bytes to write.
<code>pFile</code>	Pointer to a <code>USB_VMSD_FILE_INFO</code> structure describing the file.

Return value

= 0 Success.
 ≠ 0 An error occurred.

Additional information

Depending on the behavior of the host operating system it is possible that `pFile` is `NULL`. In this case we recommend to perform data analysis to recognize the file.

8.3.3.3 USB_VMSD_MEM_ALLOC

Description

Function prototype that is used when memory is being allocated by the VirtualMSD module.

Type definition

```
typedef void * (USB_VMSD_MEM_ALLOC)(U32 Size);
```

Parameters

Parameter	Description
<code>Size</code>	<code>Size</code> of the required memory in bytes.

Return value

Pointer to the allocated memory or NULL.

8.3.3.4 USB_VMSD_MEM_FREE

Description

Function prototype that is used when memory is being freed by the VirtualMSD module.

Type definition

```
typedef void (USB_VMSD_MEM_FREE)(void * p);
```

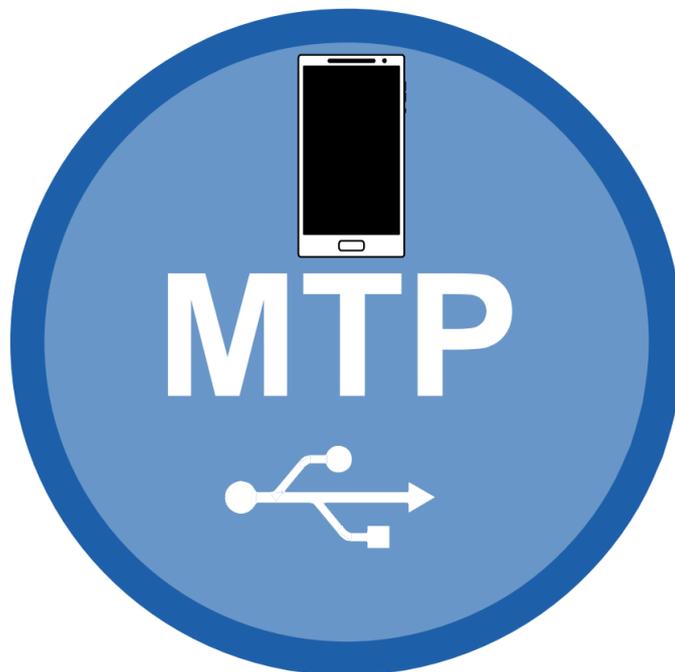
Parameters

Parameter	Description
P	Pointer to a memory block which was previously allocated by USB_VMSD_MEM_ALLOC.

Chapter 9

Media Transfer Protocol Class (MTP)

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



9.1 Overview

The Media Transfer Protocol (MTP) is a USB class protocol which can be used to transfer files to and from storage devices. MTP is an official extension of the Picture Transfer Protocol (PTP) designed to allow digital cameras to exchange image files with a computer. MTP extends this by adding support for arbitrary data types.

MTP is an alternative to Mass Storage Device (MSD) and in contrast to MSD which reads and writes sector data, it operates at the file level. This type of operation gives MTP some advantages over MSD:

- The cable can be safely removed during a data transfer without damaging the file system.
- The file system does not need to be FAT (can be the SEGGER emFile File System (EFS) or any other proprietary file system)
- The application has full control over which files are visible to the user. Selected files or directories can be hidden.
- Virtual files can be presented.
- Host and target can access storage simultaneously without conflicts.

MTP is supported by most operating systems out of the box and the installation of additional drivers is not required.

emUSB-Device-MTP supports the following capabilities:

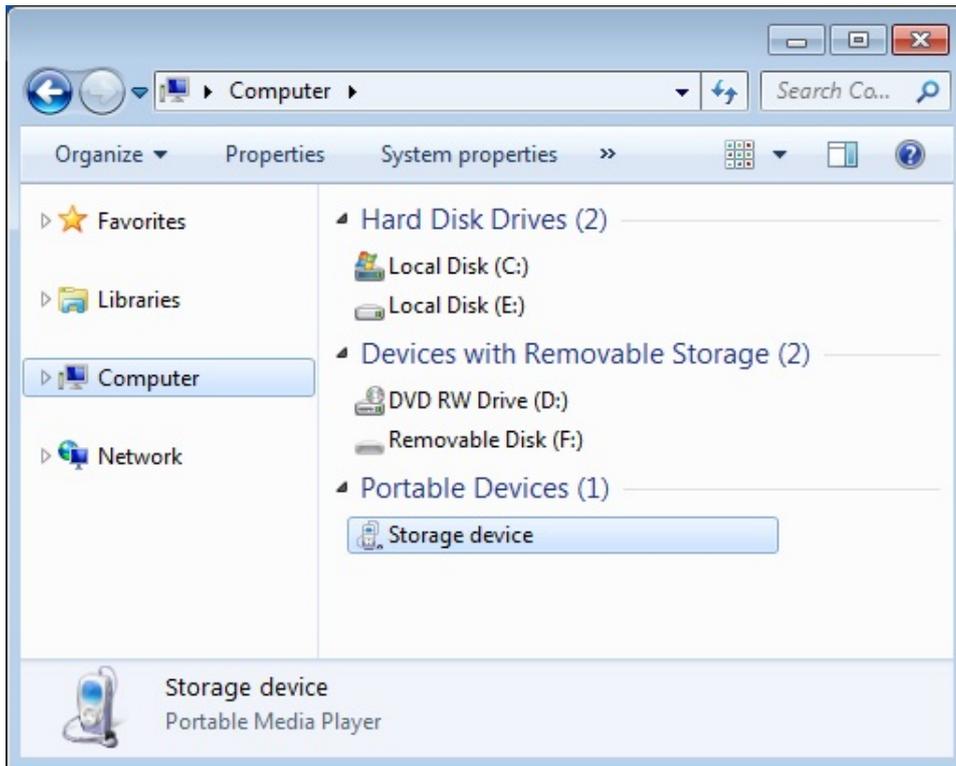
- File read
- File write
- Format
- File delete
- Directory create
- Directory delete

emUSB-Device-MTP comes as a complete package and contains the following:

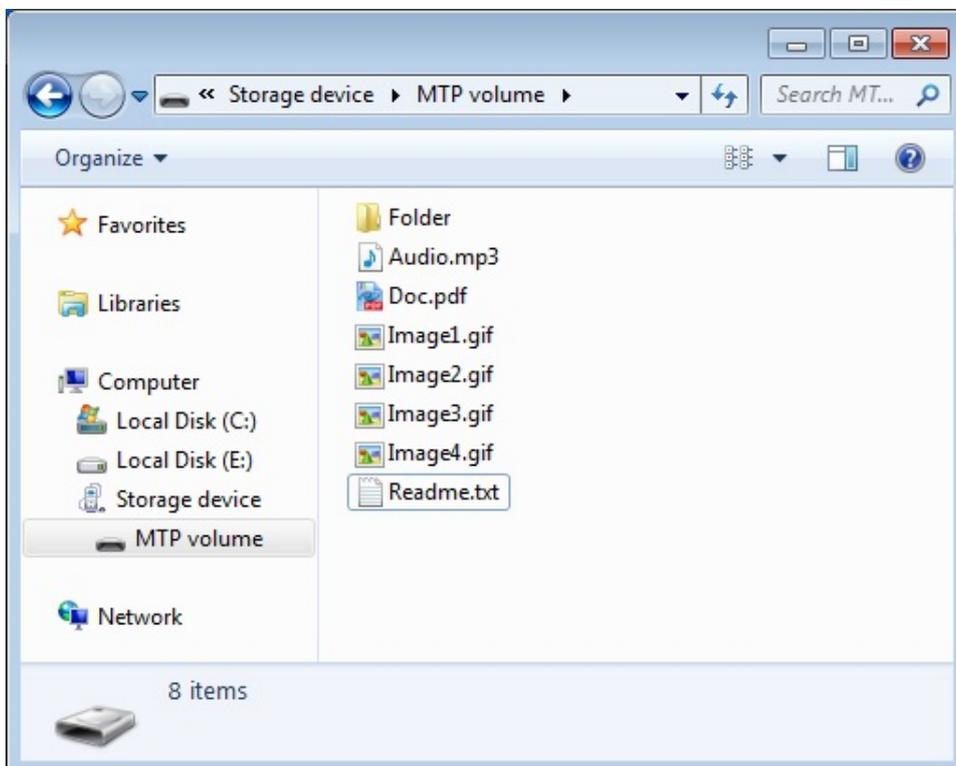
- Generic USB handling
- MTP device class implementation
- Storage driver which uses emFile
- Sample application showing how to work with MTP

9.1.1 Getting access to files

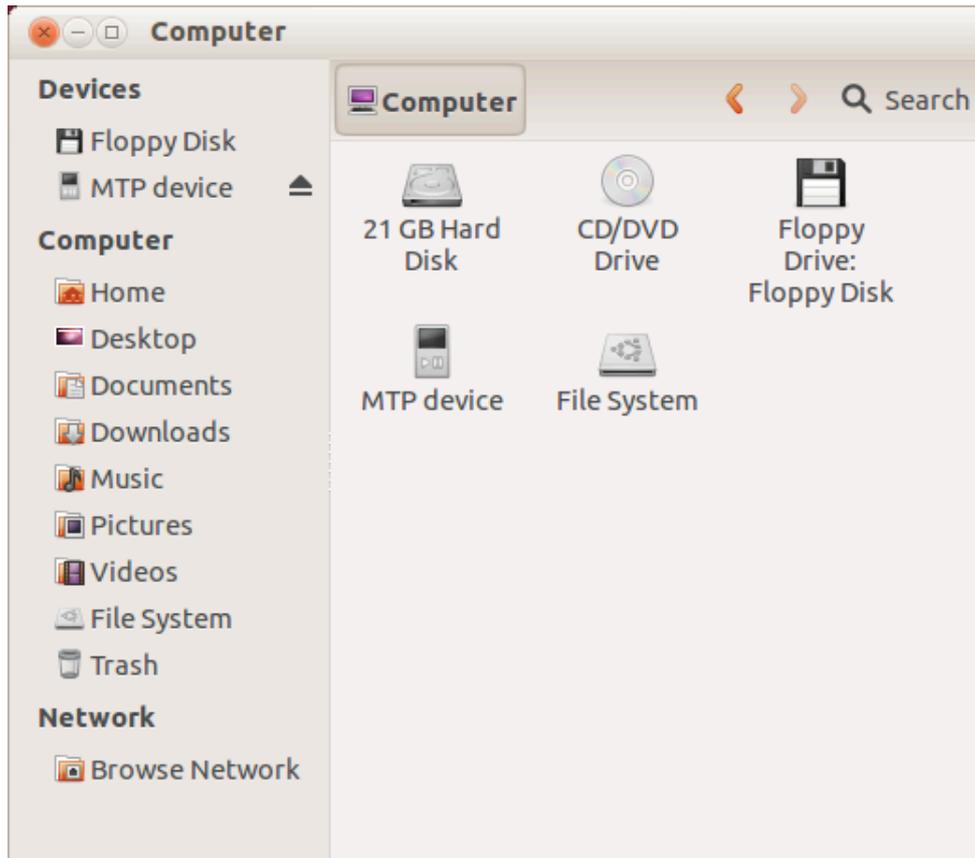
An MTP device will be displayed under the “Portable Devices” section in the “Computer” window when connected to a PC running the Microsoft Windows 7 operating system:



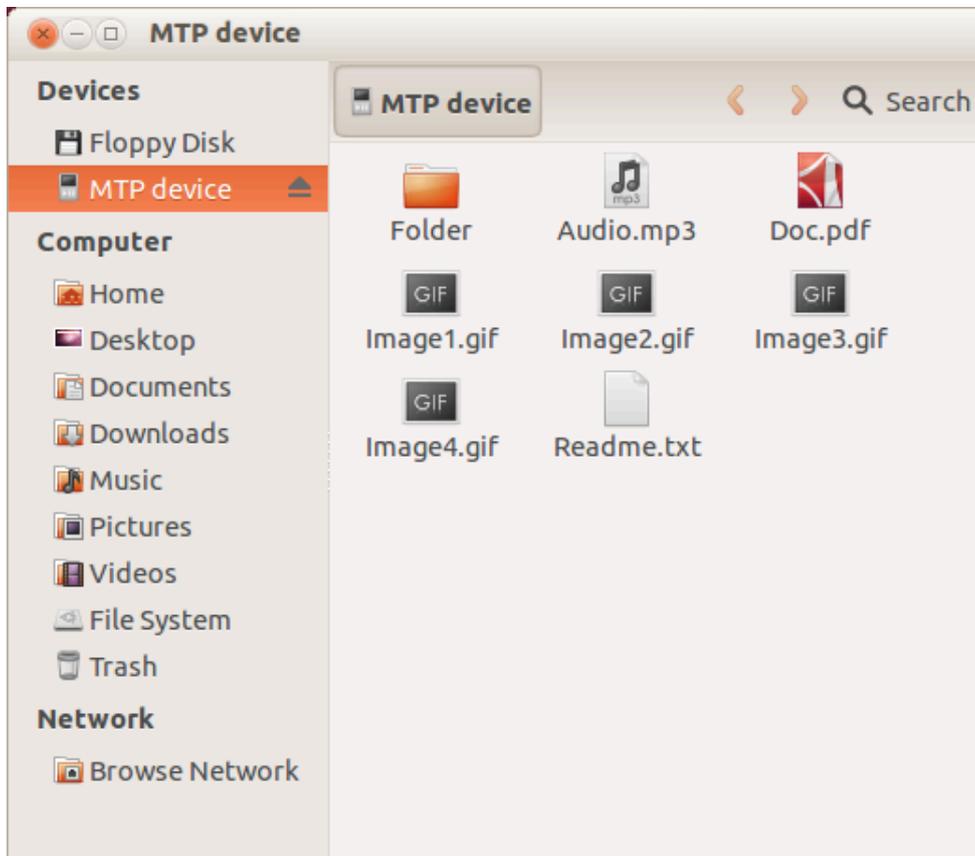
The file and directories stored on the device are accessed in the usual way using the Windows Explorer:



On the Ubuntu Linux operating system a connected MTP device is shown in the “Computer” window:



The files and directories present on the MTP device can be easily accessed via GUI:



On other operating systems the data stored on MTP devices can be accessed similarly.

9.1.2 Additional information

For more technical details about MTP and PTP follow these links:

[MTP specification](#)

[PTP specification](#)

9.2 Configuration

9.2.1 Initial configuration

To get emUSB-Device-MTP up and running as well as doing an initial test, the configuration as delivered with the sample application should not be modified.

9.2.2 Final configuration

The configuration must only be modified when emUSB-Device is integrated in your final product. Refer to section *emUSB-Device Configuration* on page 48 for detailed information about the generic information functions which have to be adapted.

9.2.3 emFile and MTP configuration for UTF8 characters

If you need to support non-ASCII characters you need to set the define `MTP_SUPPORT_UTF8` to 1 in your `USB_Conf.h` file. Furthermore you need to set the defines `FS_SUPPORT_FILE_NAME_ENCODING`, `FS_SUPPORT_EXT_ASCII` and `FS_SUPPORT_MBCS` to 1 in your `FS_Conf.h` file. Additionally you must make sure that LFN is active (`FS_FAT_SupportLFN()`) and that the following functions have been called: `FS_SetCharSetType(&FS_CHARSET_CP932)`, `FS_FAT_SetLFNConverter(&FS_UNICODE_CONV_UTF8)`. See the emFile documentation for details.

If you are not using emFile you must make sure that your filesystem is using UTF8.

9.2.4 Class specific configuration

Beside the generic emUSB-Device configuration functions (*emUSB-Device Configuration* on page 48), the following should be adapted before the emUSB-Device MTP component is used in a final product. Example implementations are supplied in the MSD example application `USB_MTP_Start.c`, located in the `Application` directory of emUSB-Device.

An MTP device is required to present an additional information set to the host. These values are added during the initial call to `USBD_MTP_Add()`.

Example

```
static const USB_MTP_INFO _MTPInfo = {
    "Vendor",           // MTP Manufacturer
    "Storage device", // MTP Model
    "1.0",             // MTP DeviceVersion
    "0123456789ABCDEF0123456789ABCDEF" // MTP SerialNumber.
                                   // It must be exactly 32 characters long.
};
...
InitData.pMTPInfo = &_MTPInfo;
...
USB_MTP_Add(&InitData);
```

9.2.5 Compile time configuration

The following macros can be added to `USB_Conf.h` file in order to configure the behavior of the MTP component.

The following types of configuration macros exist:

Binary switches "B"

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

Numerical values "N"

Numerical values are used somewhere in the code in place of a numerical constant.

Type	Macro	Default	Description
N	MTP_MAX_NUM_STORAGES	4	Maximum number of storage units the storage layer can handle. 4 additional bytes are allocated for each storage unit.
B	MTP_SAVE_FILE_INFO	0	Specifies if the object properties (file size, write protection, creation date, modification date and file id) should be stored in RAM for quick access to them. 50 additional bytes of RAM are required for each object when the switch is set to 1.
N	MTP_MAX_FILE_PATH	256	Maximum number of characters in the path to a file or directory.
B	MTP_SUPPORT_UTF8	1	Names of the files and directories which are exchanged between the MTP component and the file system are encoded in UTF-8 format.
B	MTP_SUPPORT_EVENTS	1	Support Events such as object removed/added, new storage added/removed.
B	USB_MTP_NAME_CASE_SENSITIVE	0	When checking file names and directory names the string compare will be case sensitive.

9.3 Running the sample application

The directory `Application` contains a sample application which can be used with emUSB-Device and the MTP component. To test the emUSB-Device-MTP component, the application should be built and then downloaded to target. Remove the USB connection and reconnect the target to the host. The target will enumerate and will be accessible via a file browser.

9.3.1 USB_MTP_Start.c in detail

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

```
// MainTask() - excerpt from USB_MTP_Start.c
void MainTask(void);
void MainTask(void) {
    USBD_Init();
    _AddMTP();
    USBD_Start();
    while (1) {
        while ((USBD_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED))
               != USB_STAT_CONFIGURED) {
            BSP_ToggleLED(0);
            USB_OS_Delay(50);
        }
        BSP_SetLED(0);
        USBD_MTP_Task();
    }
}
```

The first step is to initialize the USB core stack by calling `USBD_Init()`. The function `_AddMTP()` configures all required endpoints, adds the MTP component to emUSB-Device and assigns a storage medium to it. More than one storage medium can be added. The access to storage medium is done using a storage driver. emUSB-Device comes with a storage driver for the SEGGER *emFile* file system.

```
// _AddMTP() - excerpt from USB_MTP_Start.c
static void _AddMTP(void) {
    USB_MTP_INIT_DATA InitData;
    USB_MTP_INST_DATA InstData;
    //
    // Add the MTP component to USB stack.
    //
    InitData.EPIn = USBD_AddeP(1, USB_TRANSFER_TYPE_BULK,
                              USB_HS_BULK_MAX_PACKET_SIZE,
                              NULL, 0);

    InitData.EPOut = USBD_AddeP(0, USB_TRANSFER_TYPE_BULK,
                                USB_HS_BULK_MAX_PACKET_SIZE,
                                _acReceiveBuffer,
                                sizeof(_acReceiveBuffer));

    InitData.EPInt = USBD_AddeP(1, USB_TRANSFER_TYPE_INT,
                                10, NULL, 0);

    InitData.pObjectList = _aObjectList;
    InitData.NumBytesObjectList = sizeof(_aObjectList);
    InitData.pDataBuffer = _aDataBuffer;
    InitData.NumBytesDataBuffer = sizeof(_aDataBuffer);
    InitData.pMTPInfo = &_MTPInfo;
    USBD_MTP_Add(&InitData);
    //
    // Add a storage driver to MTP component.
    //
    InstData.pAPI = &USB_MTP_StorageFS;
    InstData.sDescription = "MTP volume";
    InstData.sVolumeId = "0123456789";
    InstData.DriverData.pRootDir = "";
    USBD_MTP_AddStorage(&InstData);
}
```

```
}
```

The size of `_acReceiveBuffer` and `_aDataBuffer` buffers must be a multiple of USB maximum packet size. The size of the buffer allocated for the object list, `_aObjectList` must be chosen according to the number of files on the storage medium. `emUSB-Device-MTP` assigns an internal object to each file or directory requested by the USB host. The USB host can request all the files and directories present at once or it can request files and directories as user browses them. An object requires a minimum of 54 bytes. The actual number of bytes allocated depends on the length of the full path to file/directory.

9.4 Target API

Function	Description
API functions	
<code>USBD_MTP_Add()</code>	Adds an MTP interface to the USB stack.
<code>USBD_MTP_AddStorage()</code>	Adds a storage device to emUSB-Device-MTP.
<code>USBD_MTP_Task()</code>	Main task function of MTP component which processes the commands from host.
<code>USBD_MTP_Poll()</code>	Function which handles MTP commands.
<code>USBD_MTP_SendEvent()</code>	Sends an event notification to the host.
Data structures	
<code>USB_MTP_FILE_INFO</code>	Structure which stores information about a file or directory.
<code>USB_MTP_INIT_DATA</code>	Structure which stores the parameters of the MTP interface.
<code>USB_MTP_INFO</code>	Structure that is used when initialising the MTP module.
<code>USB_MTP_INST_DATA</code>	Structure which stores the parameters of storage driver.
<code>USB_MTP_INST_DATA_DRIVER</code>	Structure which stores the parameters passed to the storage driver.
<code>USB_MTP_STORAGE_API</code>	Structure that contains callbacks to the storage driver.
<code>USB_MTP_STORAGE_INFO</code>	Structure which stores information about a storage.
Enums	
<code>USB_MTP_EVENT</code>	Enum containing the MTP event codes.

9.4.1 API functions

9.4.1.1 USBD_MTP_Add()

Description

Adds an MTP interface to the USB stack.

Prototype

```
int USBD_MTP_Add(const USB_MTP_INIT_DATA * pInitData);
```

Parameters

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

Return value

0 - Successfully added.

Additional information

After the initialization of USB core, this is the first function that needs to be called when an MTP interface is used with emUSB-Device. The structure `USB_MTP_INIT_DATA` has to be initialized before `USB_MTP_Add()` is called. Refer to `USB_MTP_INIT_DATA` for more information.

9.4.1.2 USBD_MTP_AddStorage()

Description

Adds a storage device to emUSB-Device-MTP.

Prototype

```
USB_MTP_STORAGE_HANDLE USBD_MTP_AddStorage(const USB_MTP_INST_DATA * pInstData);
```

Parameters

Parameter	Description
<code>pInstData</code>	Pointer to a <code>USB_MTP_INST_DATA</code> structure which contains the parameters of the added storage.

Return value

- = 0 Invalid handle, storage could not be added
- ≠ 0 A valid storage handle, this handle can be used with the `USB_MTP_SendEvent` to indicate an event to the host.

Additional information

It is necessary to call this function immediately after `USB_MTP_Add()`. This function adds a storage device such as a hard drive, MMC/SD card or NAND flash etc., to emUSB-Device-MTP, which will be used as source/destination of data exchange with the host. The structure `USB_MTP_INST_DATA` must be initialized before `USB_MTP_AddStorage()` is called. Refer to `USB_MTP_INST_DATA` for more information.

9.4.1.3 USBD_MTP_Task()

Description

Main task function of MTP component which processes the commands from host.

Prototype

```
void USBD_MTP_Task(void);
```

Additional information

The `USBD_MTP_Task()` should be called after the USB device has been successfully enumerated and configured. The function returns when the USB device is detached or suspended.

Check `USBD_MTP_Poll()` if you need a non-blocking version.

9.4.1.4 USBD_MTP_Poll()

Description

Function which handles MTP commands. Using this function is only necessary if you want to avoid using the blocking `USB_MTP_Task` function. This can be necessary if you are not using an RTOS.

Prototype

```
int USBD_MTP_Poll(void);
```

Return value

- 2 O.K. Command was processed, but a protocol error occurred.
- 1 O.K. Command was processed successfully.
- 0 O.K. Timeout occurred.
 - 1 An error occurred. (e.g. no cable connected).

Additional information

This function must be called periodically, otherwise the host can time out the device since it does not process commands. It normally blocks for `USB_MTP_POLL_TIMEOUT` milliseconds while waiting for a command from the host. Should a command arrive during the timeout it will be processed, which could potentially increase the block duration. The duration could also decrease because the function returns as soon as a command is finished.

In case of return value 2 the calling task should still call this function again, if possible recovery will be initiated.

9.4.1.5 USBD_MTP_SendEvent()

Description

Sends an event notification to the host.

Prototype

```
int USBD_MTP_SendEvent(USB_MTP_STORAGE_HANDLE hStorage,
                      USB_MTP_EVENT          Event,
                      void                    * pPara);
```

Parameters

Parameter	Description
<code>hStorage</code>	Handle to a storage that was returned by <code>USB_MTP_AddStorage()</code> .
<code>Event</code>	<p><code>Event</code> that occurred. The following events are currently supported:</p> <ul style="list-style-type: none"> • <code>USB_MTP_EVENT_OBJECTADDED</code> • <code>USB_MTP_EVENT_OBJECTREMOVED</code> • <code>USB_MTP_EVENT_STOREADDED</code> • <code>USB_MTP_EVENT_STOREREMOVED</code> • <code>USB_MTP_EVENT_OBJECTINFOCHANGED</code> • <code>USB_MTP_EVENT_STOREFULL</code> • <code>USB_MTP_EVENT_STORAGEINFOCHANGED</code>
<code>pPara</code>	<p>Pointer to additional information. This parameter depends on the event. In case of <code>Event =</code></p> <ul style="list-style-type: none"> • <code>USB_MTP_EVENT_OBJECTADDED</code> • <code>USB_MTP_EVENT_OBJECTREMOVED</code> • <code>USB_MTP_EVENT_OBJECTINFOCHANGED</code> <p><code>pPara</code> is a pointer to a filled <code>USB_MTP_FILE_INFO</code> structure.</p> <ul style="list-style-type: none"> • <code>USB_MTP_EVENT_STOREADDED</code> • <code>USB_MTP_EVENT_STOREREMOVED</code> • <code>USB_MTP_EVENT_STORAGEINFOCHANGED</code> <p><code>pPara</code> is not used and can be <code>NULL</code>.</p>

Return value

= 0 `Event` sent out successfully.
 ≠ 0 `Event` could not be sent.

Additional information

Sending an event notification to the MTP host makes sure that the MTP host is aware of changes in the file system of the storage. This function can also be used to notify that a storage has been added or removed.

Example

```
static void _GetFileInfo(const char * sPath, USB_MTP_FILE_INFO * pFileInfo) {
    const char * s;
    U8 AttrFS;
    U8 AttrMTP;

    memset(pFileInfo, 0, sizeof(USB_MTP_FILE_INFO));
    s = strrchr(sPath, '\\');
    if (s) {
        s++; // Go to the next character after '\\'.
    } else {
        s = sPath;
    }
}
```

```

//
// In case the file path starts with \ skip this.
//
if (*sPath == '\\') {
    sPath++;
}
pFileInfo->pFileName = (char *)s;
pFileInfo->pFilePath = (char *)sPath;
FS_GetFileTimeEx(pFileInfo->pFilePath, &pFileInfo->CreationTime,
                 FS_FILETIME_CREATE);
FS_GetFileTimeEx(pFileInfo->pFilePath, &pFileInfo->LastWriteTime,
                 FS_FILETIME_MODIFY);
pFileInfo->IsDirectory = 0;
AttrFS = FS_GetFileAttributes(pFileInfo ? pFilePath);
if (AttrFS & FS_ATTR_DIRECTORY) {
    pFileInfo->IsDirectory = 1;
}
AttrMTP = 0;
if (AttrFS & FS_ATTR_READ_ONLY) {
    AttrMTP |= MTP_FILE_ATTR_WP;
}
if (AttrFS & FS_ATTR_SYSTEM) {
    AttrMTP |= MTP_FILE_ATTR_SYSTEM;
}
if (AttrFS & FS_ATTR_HIDDEN) {
    AttrMTP |= MTP_FILE_ATTR_HIDDEN;
}
pFileInfo->Attributes = AttrMTP;
}

static int _WriteLogFile(const char * sLogFilePath) {
    char ac[30];
    FS_FILE * pFile;
    int r = 0;
    USB_MTP_FILE_INFO FileInfo = {0};

    if (FS_IsVolumeMounted("")) {
        //
        // Check whether file already exists
        //
        pFile = FS_FOpen(sLogFilePath, "r");
        if (pFile) {
            r = USB_MTP_EVENT_OBJECTINFOCHANGED;
            FS_Fclose(pFile);
        } else {
            r = USB_MTP_EVENT_OBJECTADDED;
        }
        pFile = FS_FOpen(sLogFilePath, "a+");
        if (pFile) {
            sprintf(ac, "OS_Time = %.8d\r\n", (int)OS_GetTime());
            FS_Write(pFile, ac, 20);
            FS_Fclose(pFile);
        } else {
            r = 0;
        }
    }
    _GetFileInfo(sLogFilePath, &FileInfo);
    //
    // Send events to the host.
    //
    USBD_MTP_SendEvent(_ahStorage[0], (USB_MTP_EVENT)r, &FileInfo);
    USBD_MTP_SendEvent(_ahStorage[0], USB_MTP_EVENT_STORAGEINFOCHANGED, NULL);
    return r;
}

```

9.4.2 Data structures

9.4.2.1 USB_MTP_FILE_INFO

Description

Structure which stores information about a file or directory.

Type definition

```
typedef struct {
    char * pFilePath;
    char * pFileName;
    U32    FileSize;
    U32    CreationTime;
    U32    LastWriteTime;
    U8     IsDirectory;
    U8     Attributes;
    U8     acId[];
} USB_MTP_FILE_INFO;
```

Structure members

Member	Description
<code>pFilePath</code>	Full path to file.
<code>pFileName</code>	Pointer to beginning of file/directory name in <code>pFilePath</code> .
<code>FileSize</code>	Size of the file in bytes. 0xFFFFFFFF when larger than 4GB.
<code>CreationTime</code>	The time and date when the file was created.
<code>LastWriteTime</code>	The time and date when the file was last modified.
<code>IsDirectory</code>	Set to 1 if the path points to a directory.
<code>Attributes</code>	Bitmask of file attributes (MTP_FILE_ATTR_...).
<code>acId</code>	Unique identifier which persists between MTP sessions.

Additional information

The date and time is formatted as follows:

Bit range	Value range	Description
0-4	0-29	2-second count
5-10	0-59	Minutes
11-15	0-23	Hours
16-20	1-31	Day of month
21-24	1-12	Month of year
25-31	0-127	Number of years since 1980

The following attributes are supported:

Bitmask	Description
MTP_FILE_ATTR_WP	File/directory can not be modified.
MTP_FILE_ATTR_SYSTEM	File/directory is required for the correct functioning of the system.
MTP_FILE_ATTR_HIDDEN	File/directory should not be shown to the user.

`acId` should be unique for each file and directory on the file system and it should be persistent between MTP sessions.

9.4.2.2 USB_MTP_INIT_DATA

Description

Structure which stores the parameters of the MTP interface.

Type definition

```
typedef struct {
    U8          EPIn;
    U8          EPOut;
    U8          EPInt;
    void        * pObjectList;
    U32         NumBytesObjectList;
    void        * pDataBuffer;
    U32         NumBytesDataBuffer;
    USB_MTP_INFO * pMTPInfo;
    U8          InterfaceNum;
    U32         NumBytesAllocated;
    U32         NumObjects;
} USB_MTP_INIT_DATA;
```

Structure members

Member	Description
EPIn	Endpoint for receiving data from host.
EPOut	Endpoint for sending data to host.
EPInt	Endpoint for sending events to host.
pObjectList	Pointer to a memory region where the list of MTP objects is stored. Should be 4 byte aligned.
NumBytesObjectList	Number of bytes allocated for the object list.
pDataBuffer	Pointer to a memory region to be used as communication buffer.
NumBytesDataBuffer	Number of bytes allocated for the data buffer.
pMTPInfo	Pointer to a <code>USB_MTP_INFO</code> structure. Filling this structure is mandatory.
InterfaceNum	Internal use.
NumBytesAllocated	Internal use.
NumObjects	Internal use.

Additional information

This structure holds the endpoints that should be used with the MTP interface. Refer to `USBD_AddEP()` for more information about how to add an endpoint.

The number of bytes in the `pDataBuffer` should be a multiple of USB maximum packet size. The number of bytes in the object list depends on the number of files/directories on the storage medium. An object is assigned to each file/directory when the USB host requests the object information for the first time.

9.4.2.3 USB_MTP_INFO

Description

Structure that is used when initialising the MTP module.

Type definition

```
typedef struct {
    const char * pManufacturer;
    const char * pModel;
    const char * pDeviceVersion;
    const char * pSerialNumber;
} USB_MTP_INFO;
```

Structure members

Member	Description
<code>pManufacturer</code>	Name of the device manufacturer.
<code>pModel</code>	Model name of the MTP device.
<code>pDeviceVersion</code>	Version of the MTP device.
<code>pSerialNumber</code>	Serial number of the MTP device. The serial number should contain exactly 32 hexadecimal characters. It must be unique among devices sharing the same model name and device version strings. The MTP device returns this string in the Serial Number field of the DeviceInfo dataset. For more information, refer to MTP specification.

9.4.2.4 USB_MTP_INST_DATA

Description

Structure which stores the parameters of storage driver.

Type definition

```
typedef struct {
    const USB_MTP_STORAGE_API * pAPI;
    const char                 * sDescription;
    const char                 * sVolumeId;
    USB_MTP_INST_DATA_DRIVER   DriverData;
} USB_MTP_INST_DATA;
```

Structure members

Member	Description
pAPI	Pointer to a structure that holds the storage device driver API.
sDescription	Human-readable string which identifies the storage. This string is displayed in Nautilus/Windows Explorer/etc.
sVolumeId	Unique volume identifier
DriverData	Driver data that are passed to the storage driver. Refer to <code>USB_MTP_INST_DATA_DRIVER</code> for detailed information about how to initialize this structure. This field must be up to 256 characters long but only the first 128 are significant and these must be unique for all storages of an MTP device.

Additional information

The MTP device returns the [sDescription](#) string in the Storage Description parameter and the [sVolumeId](#) in the Volume Identifier of the StorageInfo dataset. For more information, refer to MTP specification.

9.4.2.5 USB_MTP_INST_DATA_DRIVER

Description

Structure which stores the parameters passed to the storage driver.

Type definition

```
typedef struct {  
    const char * pRootDir;  
    U8          IsRemovable;  
} USB_MTP_INST_DATA_DRIVER;
```

Structure members

Member	Description
pRootDir	Path to directory to be used as the root of the storage.
IsRemovable	Internal use.

Additional information

[pRootDir](#) can specify the root of the file system or any other subdirectory.

9.4.2.6 USB_MTP_STORAGE_API

Description

Structure that contains callbacks to the storage driver.

Type definition

```
typedef struct {
    USB_MTP_STORAGE_INIT                * pfInit;
    USB_MTP_STORAGE_GET_INFO            * pfGetInfo;
    USB_MTP_STORAGE_FIND_FIRST_FILE    * pfFindFirstFile;
    USB_MTP_STORAGE_FIND_NEXT_FILE    * pfFindNextFile;
    USB_MTP_STORAGE_OPEN_FILE          * pfOpenFile;
    USB_MTP_STORAGE_CREATE_FILE        * pfCreateFile;
    USB_MTP_STORAGE_READ_FROM_FILE     * pfReadFromFile;
    USB_MTP_STORAGE_WRITE_TO_FILE      * pfWriteToFile;
    USB_MTP_STORAGE_CLOSE_FILE         * pfCloseFile;
    USB_MTP_STORAGE_REMOVE_FILE        * pfRemoveFile;
    USB_MTP_STORAGE_CREATE_DIR         * pfCreateDir;
    USB_MTP_STORAGE_REMOVE_DIR         * pfRemoveDir;
    USB_MTP_STORAGE_FORMAT             * pfFormat;
    USB_MTP_STORAGE_RENAME_FILE        * pfRenameFile;
    USB_MTP_STORAGE_DEINIT             * pfDeInit;
    USB_MTP_STORAGE_GET_FILE_ATTRIBUTES * pfGetFileAttributes;
    USB_MTP_STORAGE_MODIFY_FILE_ATTRIBUTES * pfModifyFileAttributes;
    USB_MTP_STORAGE_GET_FILE_CREATION_TIME * pfGetFileCreationTime;
    USB_MTP_STORAGE_GET_FILELAST_WRITE_TIME * pfGetFileLastWriteTime;
    USB_MTP_STORAGE_GET_FILE_ID        * pfGetFileId;
    USB_MTP_STORAGE_GET_FILE_SIZE      * pfGetFileSize;
} USB_MTP_STORAGE_API;
```

Structure members

Member	Description
pfInit	Initializes the storage medium.
pfGetInfo	Returns information about the storage medium such as storage capacity and the available free space.
pfFindFirstFile	Returns information about the first file in a given directory.
pfFindNextFile	Moves to next file and returns information about it.
pfOpenFile	Opens an existing file.
pfCreateFile	Creates a new file.
pfReadFromFile	Reads data from the current file.
pfWriteToFile	Writes data to current file.
pfCloseFile	Closes the current file.
pfRemoveFile	Removes a file from storage medium.
pfCreateDir	Creates a new directory.
pfRemoveDir	Removes a directory from storage medium.
pfFormat	Formats the storage.
pfRenameFile	Changes the name of a file or directory.
pfDeInit	De-initializes the storage medium.
pfGetFileAttributes	Reads the attributes of a file or directory.
pfModifyFileAttributes	Changes the attributes of a file or directory.
pfGetFileCreationTime	Returns the creation time of a file or directory.

Member	Description
pfGetFileLastWriteTime	Returns the time of the last modification made to a file or directory.
pfGetFileId	Returns the unique ID of a file or directory.
pfGetFileSize	Returns the size of a file in bytes.

Additional information

USB_MTP_STORAGE_API is used to retrieve information from the storage device driver or access data that needs to be read or written. Detailed information can be found in *MTP Storage Driver* on page 311.

9.4.2.7 USB_MTP_STORAGE_INFO

Description

Structure which stores information about a storage.

Type definition

```
typedef struct {
    U32    NumKBytesTotal;
    U32    NumKBytesFreeSpace;
    U16    FSType;
    U8     IsWriteProtected;
    U8     IsRemovable;
    char   DirDelimiter;
} USB_MTP_STORAGE_INFO;
```

Structure members

Member	Description
NumKBytesTotal	Storage capacity in kBytes
NumKBytesFreeSpace	Available free space on storage in kBytes
FSType	Type of file system as specified by MTP
IsWriteProtected	Set to 1 if the storage medium can not be modified
IsRemovable	Set to 1 if the storage medium can be removed from device
DirDelimiter	Character which separates the directory/file names in a path

9.4.3 Enums

9.4.3.1 USB_MTP_EVENT

Description

Enum containing the MTP event codes.

Type definition

```
typedef enum {
    USB_MTP_EVENT_UNDEFINED,
    USB_MTP_EVENT_CANCELTRANSACTION,
    USB_MTP_EVENT_OBJECTADDED,
    USB_MTP_EVENT_OBJECTREMOVED,
    USB_MTP_EVENT_STOREADDED,
    USB_MTP_EVENT_STOREREMOVED,
    USB_MTP_EVENT_DEVICEPROPCHANGED,
    USB_MTP_EVENT_OBJECTINFOCHANGED,
    USB_MTP_EVENT_DEVICEINFOCHANGED,
    USB_MTP_EVENT_REQUESTOBJECTTRANSFER,
    USB_MTP_EVENT_STOREFULL,
    USB_MTP_EVENT_DEVICERESET,
    USB_MTP_EVENT_STORAGEINFOCHANGED,
    USB_MTP_EVENT_CAPTURECOMPLETE,
    USB_MTP_EVENT_UNREPORTEDSTATUS,
    USB_MTP_EVENT_OBJECTPROPCHANGED,
    USB_MTP_EVENT_OBJECTPROPDESCCHANGED,
    USB_MTP_EVENT_OBJECTREFERENCESCHANGED
} USB_MTP_EVENT;
```

Enumeration constants

Constant	Description
USB_MTP_EVENT_UNDEFINED	This event code is undefined, and is not used
USB_MTP_EVENT_CANCELTRANSACTION	This event is used to initiate the cancellation of a transaction over transports which do not have their own mechanism for canceling transactions. Currently not used.
USB_MTP_EVENT_OBJECTADDED	This event informs the host about a new object that has been added to the storage.
USB_MTP_EVENT_OBJECTREMOVED	Informs the host that an object has been removed.
USB_MTP_EVENT_STOREADDED	This event indicates that a storage has been added to the device. It allows to dynamically show the available storages.
USB_MTP_EVENT_STOREREMOVED	This event indicates that a storage has been removed to the device. It allows to dynamically hide the available storages.
USB_MTP_EVENT_DEVICEPROPCHANGED	A property changed on the device has occurred. Currently not used.
USB_MTP_EVENT_OBJECTINFOCHANGED	This event indicates that the information for a particular object has changed and that the host should acquire the information once again.

Constant	Description
USB_MTP_EVENT_DEVICEINFOCHANGED	This event indicates that the capabilities of the device have changed and that the DeviceInfo should be requested again. Currently not used.
USB_MTP_EVENT_REQUESTOBJECTTRANSFER	This event can be used by the device to ask the host to initiate a file object transfer to him. Currently not used.
USB_MTP_EVENT_STOREFULL	This event should be sent when a storage becomes full.
USB_MTP_EVENT_DEVICERESET	Notifies the host about an internal reset. Currently not used
USB_MTP_EVENT_STORAGEINFOCHANGED	This event is used when information of a storage changes.
USB_MTP_EVENT_CAPTURECOMPLETE	Informs the host that the previously initiated capture acquire is complete. Currently not used.
USB_MTP_EVENT_UNREPORTEDSTATUS	This event may be implemented for certain transports in cases where the responder is unable to report events to the initiator regarding changes in its internal status. Currently not used.
USB_MTP_EVENT_OBJECTPROPCHANGED	Informs about a change in the object property of a specific object. Currently not used.
USB_MTP_EVENT_OBJECTPROPDESCCHANGED	This event informs that the property description of an object property has been changed and needs to be re-acquired. Currently not used.
USB_MTP_EVENT_OBJECTREFERENCESCHANGED	This event is used to indicate that the references on an object have been updated. Currently not used.

9.5 MTP Storage Driver

This section describes the emUSB-Device MTP storage interface in detail.

9.5.1 General information

This release comes with `USB_MTP_StorageFS` driver which uses `emFile` to access the storage medium. If you are using `emFile` this chapter can be ignored. This chapter is for those who wish to write a file system interface for a third-party file system.

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.

9.5.2 Interface function list

As described above, access to a storage media is realized through an API-function table of type `USB_MTP_STORAGE_API`. The structure is declared in `USB_MTP.h` and it is described in section on page 301

9.5.3 USB_MTP_STORAGE_API in detail

9.5.3.1 USB_MTP_STORAGE_INIT

Description

Initializes the storage medium.

Type definition

```
typedef void (USB_MTP_STORAGE_INIT)(          U8          Unit,
                                         const USB_MTP_INST_DATA_DRIVER * pDriverData);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>pFilePath</code>	Pointer to a <code>USB_MTP_INST_DATA_DRIVER</code> structure that contains all information that is necessary for the driver initialization. For detailed information about the <code>USB_MTP_INST_DATA_DRIVER</code> structure, refer to <code>USB_MTP_INST_DATA_DRIVER</code> .

Additional information

This function is called when the storage driver is added to emUSB-Device-MTP. It is the first function of the storage driver to be called.

9.5.3.2 USB_MTP_STORAGE_GET_INFO

Description

Returns information about storage medium such as capacity and available free space.

Type definition

```
typedef void (USB_MTP_STORAGE_GET_INFO)(U8 Unit,  
USB_MTP_STORAGE_INFO * pStorageInfo);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>pStorageInfo</code>	Pointer to a <code>USB_MTP_STORAGE_INFO</code> structure. For detailed information about the <code>USB_MTP_STORAGE_INFO</code> structure, refer to <code>USB_MTP_STORAGE_INFO</code> .

Additional information

Typically, this function is called immediately after the device is connected to USB host when the USB host requests information about the available storage mediums.

9.5.3.3 USB_MTP_STORAGE_FIND_FIRST_FILE

Description

Returns information about the first file in a specified directory.

Type definition

```
typedef int (USB_MTP_STORAGE_FIND_FIRST_FILE)(          U8          Unit,
                                                    const char    * pDirPath,
                                                    USB_MTP_FILE_INFO * pFileInfo);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>pDirPath</code>	Full path to the directory to be searched.
<code>pFileInfo</code>	out Information about the file/directory found.

Return value

= 0 File/directory found
 = 1 No more files/directories found
 < 0 An error occurred

Additional information

The "." and ".." directory entries which are relevant only for the file system should be skipped.

9.5.3.4 USB_MTP_STORAGE_FIND_NEXT_FILE

Description

Moves to next file and returns information about it.

Type definition

```
typedef int (USB_MTP_STORAGE_FIND_NEXT_FILE)(U8          Unit,
                                             USB_MTP_FILE_INFO * pFileInfo);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>pFileInfo</code>	<code>out</code> Information about the file/directory found.

Return value

= 0 File/directory found
 = 1 No more files/directories found
 < 0 An error occurred

Additional information

The "." and ".." directory entries which are relevant only for the file system should be skipped.

9.5.3.5 USB_MTP_STORAGE_OPEN_FILE

Description

Opens a file for reading.

Type definition

```
typedef int (USB_MTP_STORAGE_OPEN_FILE)(          U8      Unit,
                                                const char * pFilePath);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	<input type="text"/> Full path to file.

Return value

= 0 File opened
 ≠ 0 An error occurred

Additional information

This function is called at the beginning of a file read operation. It is followed by one or more calls to `USB_MTP_STORAGE_READ_FROM_FILE`. At the end of data transfer the MTP module closes the file by calling `USB_MTP_STORAGE_CLOSE_FILE`. If the file does not exist an error should be returned. The MTP module opens only one file at a time.

9.5.3.6 USB_MTP_STORAGE_CREATE_FILE

Description

Opens a file for writing.

Type definition

```
typedef int (USB_MTP_STORAGE_CREATE_FILE)(          U8          Unit,
                                                  const char * pDirPath,
                                                  USB_MTP_FILE_INFO * pFileInfo);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>pDirPath</code>	in Full path to directory where the file should be created.
<code>pFileInfo</code>	in Information about the file to be created. <code>pFileName</code> points to the name of the file. out <code>pFilePath</code> points to full path of created file, <code>pFileName</code> points to the beginning of file name in <code>pFilePath</code> .

Return value

= 0 File created and opened
 ≠ 0 An error occurred

Additional information

This function is called at the beginning of a file write operation. The name of the file is specified in the `pFileName` field of `pFileInfo`. If the file exists it should be truncated to zero length. When a file is created, the call to `USB_MTP_STORAGE_CREATE_FILE` is followed by one or more calls to `USB_MTP_STORAGE_WRITE_TO_FILE`. If `CreationTime` and `LastWriteTime` in `pFileInfo` are not zero, these should be used instead of the time stamps generated by the file system.

9.5.3.7 USB_MTP_STORAGE_READ_FROM_FILE

Description

Reads data from the current file.

Type definition

```
typedef int (USB_MTP_STORAGE_READ_FROM_FILE)(U8      Unit,
                                             U32      Off,
                                             void *  pData,
                                             U32      NumBytes);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>Off</code>	Byte offset where to read from.
<code>pData</code>	<code>out</code> Data read from file.
<code>NumBytes</code>	Number of bytes to read from file.

Return value

= 0 Data read from file
 ≠ 0 An error occurred

Additional information

The function reads data from the file opened by `USB_MTP_STORAGE_OPEN_FILE`.

9.5.3.8 USB_MTP_STORAGE_WRITE_TO_FILE

Description

Writes data to current file.

Type definition

```
typedef int (USB_MTP_STORAGE_WRITE_TO_FILE)(
    U8      Unit,
    U32     Off,
    const void * pData,
    U32     NumBytes);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>Off</code>	Byte offset where to read from.
<code>pData</code>	<code>in</code> Data to write to file.
<code>NumBytes</code>	Number of bytes to write to file.

Return value

= 0 Data written to file
 ≠ 0 An error occurred

Additional information

The function writes data to file opened by `USB_MTP_STORAGE_CREATE_FILE`.

9.5.3.9 USB_MTP_STORAGE_CLOSE_FILE

Description

Closes the current file.

Type definition

```
typedef int (USB_MTP_STORAGE_CLOSE_FILE)(U8 Unit);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.

Return value

= 0 File closed.
≠ 0 An error occurred

Additional information

The function closes the file opened by `USB_MTP_STORAGE_CREATE_FILE` or `USB_MTP_STORAGE_OPEN_FILE`.

9.5.3.10 USB_MTP_STORAGE_REMOVE_FILE

Description

Removes a file/directory from the storage medium.

Type definition

```
typedef int (USB_MTP_STORAGE_REMOVE_FILE)(          U8      Unit,  
                                             const char * pFilePath);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Full path to file/directory to be removed

Return value

= 0 File removed.
≠ 0 An error occurred

9.5.3.11 USB_MTP_STORAGE_CREATE_DIR

Description

Creates a directory on the storage medium.

Type definition

```
typedef int (USB_MTP_STORAGE_CREATE_DIR)(          U8          Unit,
                                                const char * pDirPath,
                                                USB_MTP_FILE_INFO * pFileInfo);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>pDirPath</code>	in Full path to directory where the directory should be created.
<code>pFileInfo</code>	in Information about the directory to be created. <code>pFileName</code> points to the directory name. out <code>pFilePath</code> points to full path of directory, <code>pFileName</code> points to the beginning of directory name in <code>pFilePath</code>

Return value

= 0 Directory created.
 ≠ 0 An error occurred

Additional information

If `CreationTime` and `LastWriteTime` in `pFileInfo` are not available, zero should be used instead of the time stamps generated by the file system.

9.5.3.12 USB_MTP_STORAGE_REMOVE_DIR

Description

Removes a directory and its contents from the storage medium.

Type definition

```
typedef int (USB_MTP_STORAGE_REMOVE_DIR)(          U8      Unit,  
                                           const char * pDirPath);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>pDirPath</code>	in Full path to directory to be removed.

Return value

= 0 Directory removed.
≠ 0 An error occurred

Additional information

The function should remove the directory and the entire file tree under it.

9.5.3.13 USB_MTP_STORAGE_FORMAT

Description

Initializes the storage medium.

Type definition

```
typedef int (USB_MTP_STORAGE_FORMAT)(U8 Unit);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.

Return value

= 0 Storage medium initialized.
≠ 0 An error occurred

Additional information

The file system layer has to differentiate between two cases, one where the MTP root directory is the same as the root directory of the file system and one where it is only a subdirectory of the file system. If `pRootDir` which was configured in the call to `USB_MTP_STORAGE_INIT`, points to a subdirectory of the file system, the storage medium should not be formatted. Instead, all the files and directories underneath `pRootDir` should be removed.

9.5.3.14 USB_MTP_STORAGE_RENAME_FILE

Description

Changes the name of a file or directory.

Type definition

```
typedef int (USB_MTP_STORAGE_RENAME_FILE)(U8 Unit,
                                           USB_MTP_FILE_INFO * pFileInfo);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>pFileInfo</code>	Pointer to a <code>USB_MTP_FILE_INFO</code> structure. in Information about the file/directory to be renamed. <code>pFilePath</code> member points to the full path and <code>pFileName</code> points to the new name. out <code>pFilePath</code> member points to full path of file/directory with the new name, <code>pFileName</code> points to the beginning of file/directory name in <code>pFilePath</code> . The other structure fields should also be filled.

Return value

= 0 File/directory renamed.
 ≠ 0 An error occurred

Additional information

Only the name of the file/directory should be changed. The path to parent directory should remain the same.

9.5.3.15 USB_MTP_STORAGE_DEINIT

Description

De-initializes the storage medium.

Type definition

```
typedef void (USB_MTP_STORAGE_DEINIT)(U8 Unit);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.

Additional information

Typically called when the application calls `USBD_Stop()` to de-initialize emUSB-Device.

9.5.3.16 USB_MTP_STORAGE_GET_FILE_ATTRIBUTES

Description

Returns the attributes of a file or directory.

Type definition

```
typedef int (USB_MTP_STORAGE_GET_FILE_ATTRIBUTES)(      U8      Unit,
                                                    const char * pFilePath,
                                                    U8      * pMask);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>pFilePath</code>	Full path to file or directory (0-terminated string).
<code>pMask</code>	out The bitmask of the attributes.

Return value

= 0 Information returned.
 ≠ 0 An error occurred

Additional information

This function is called only when the compile time switch `MTP_SAVE_FILE_INFO` is set to 0. For the list of supported attributes refer to `USB_MTP_FILE_INFO`.

9.5.3.17 USB_MTP_STORAGE_MODIFY_FILE_ATTRIBUTES

Description

Sets and clears file attributes.

Type definition

```
typedef int (USB_MTP_STORAGE_MODIFY_FILE_ATTRIBUTES)(          U8      Unit,
                                                            const char * pFilePath,
                                                            U8      SetMask,
                                                            U8      ClrMask);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>pFilePath</code>	Full path to file or directory (0-terminated string).
<code>SetMask</code>	The bitmask of the attributes which should be set.
<code>ClrMask</code>	The bitmask of the attributes which should be cleared.

Return value

= 0 Attributes modified.
 ≠ 0 An error occurred

Additional information

This function is called only when the compile time switch `MTP_SAVE_FILE_INFO` is set to 0. For the list of supported attributes refer to `USB_MTP_FILE_INFO`.

9.5.3.18 USB_MTP_STORAGE_GET_FILE_CREATION_TIME

Description

Returns the creation time of file or directory.

Type definition

```
typedef int (USB_MTP_STORAGE_GET_FILE_CREATION_TIME)(      U8      Unit,
                                                         const char * pFilePath,
                                                         U32      * pTime);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>pFilePath</code>	Full path to file or directory (0-terminated string).
<code>pTime</code>	out The creation time.

Return value

= 0 Creation time returned.
 ≠ 0 An error occurred

Additional information

This function is called only when the compile time switch `MTP_SAVE_FILE_INFO` is set to 0. For the list of supported attributes refer to `USB_MTP_FILE_INFO`.

9.5.3.19 USB_MTP_STORAGE_GET_FILELAST_WRITE_TIME

Description

Returns the time when the file or directory was last modified.

Type definition

```
typedef int (USB_MTP_STORAGE_GET_FILELAST_WRITE_TIME)(      U8      Unit,
                                                           const char * pFilePath,
                                                           U32      * pTime);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>pFilePath</code>	Full path to file or directory (0-terminated string).
<code>pTime</code>	out The modification time.

Return value

= 0 Modification time returned.
 ≠ 0 An error occurred

Additional information

This function is called only when the compile time switch `MTP_SAVE_FILE_INFO` is set to 0. For the list of supported attributes refer to `USB_MTP_FILE_INFO`.

9.5.3.20 USB_MTP_STORAGE_GET_FILE_ID

Description

Returns an ID which uniquely identifies the file or directory.

Type definition

```
typedef int (USB_MTP_STORAGE_GET_FILE_ID)(      U8      Unit,
                                               const char * pFilePath,
                                               U8      * pId);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>pFilePath</code>	Full path to file or directory (0-terminated string).
<code>pId</code>	out The unique ID of file or directory. Should point to a byte array <code>MTP_NUM_BYTES_FILE_ID</code> large.

Return value

= 0 ID returned.
 ≠ 0 An error occurred

Additional information

This function is called only when the compile time switch `MTP_SAVE_FILE_INFO` is set to 0.

9.5.3.21 USB_MTP_STORAGE_GET_FILE_SIZE

Description

Returns the size of a file in bytes.

Type definition

```
typedef int (USB_MTP_STORAGE_GET_FILE_SIZE)(      U8      Unit,
                                                const char * pFilePath,
                                                U32     * pFileSize);
```

Parameters

Parameter	Description
<code>Unit</code>	Logical unit number. Specifies for which storage medium the function is called.
<code>pFilePath</code>	Full path to file or directory (0-terminated string).
<code>pFileSize</code>	<code>out</code> The size of file in bytes.

Return value

= 0 Size of file returned.
 ≠ 0 An error occurred

Additional information

This function is called only when the compile time switch `MTP_SAVE_FILE_INFO` is set to 0.

Chapter 10

Communication Device Class (CDC)

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



10.1 Overview

The Communication Device Class - Abstract Control Model (CDC-ACM) is an abstract USB class protocol defined by the USB Implementers Forum. For simplicity CDC-ACM is often referred to as simply CDC. This protocol covers the handling of the following communication flows:

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

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. Starting in Windows 10, such a file is not necessary anymore. A generic `inf` is provided, handling devices/interfaces with a `Device-/InterfaceClass = 0x02` or `Device-/InterfaceClass = 0x02` and `Device-/InterfaceSubClass = 0x02`. Linux handles USB 2 virtual COM ports since Kernel Ver. 2.4. Further information can be found in the Linux Kernel documentation.

10.1.1 Configuration

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

10.1.2 CDC-ACM issues on Windows 10

Windows 10 comes with a re-designed driver for CDC-ACM. At the time of writing (June 2019 (re-confirmed in January 2021)) Windows 10 has an issue with large IN CDC transfers. Sometimes packets seems to disappear inside the Windows 10 USB stack.

The only workaround is to read in small chunks. Or to add a delay to the transfers. We have analysed this using a hardware USB analyser and a test program which reads data from the device. The device (USB high-speed) sends out data continuously in 512 bytes packets. Each packet has a unique, consecutive ID. The test application checks that the received packet always has the ID of the previous packet + 1. After a couple dozen packet the error usually appears and a packet ends up missing. When comparing the packets which are seen "on the wire" using the USB analyser with the packets which the Windows 10 program received it can be seen that sometimes packets are missing even though they were clearly successfully received by Windows 10. E.g. on the analyser one can see packets 30, 31, 32, 33 and on Windows 10 one can see 30, 31, 33. It would appear that the method through which data is read from the COM port (Windows API `ReadFile`, `ReadFile overlapped` or `ReadFileEx`) has no effect on the missing packets.

When using the same program on Windows 7 no issues can be seen.

10.2 The example application

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

Source code excerpt from USB_CDC_Echo.c:

```

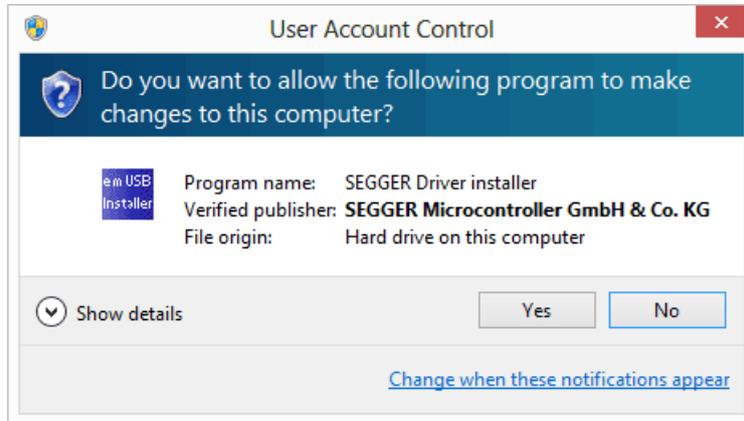
/*****
 *
 *      MainTask
 */
void MainTask(void);
void MainTask(void) {
    USB_CDC_HANDLE hInst;
    USBD_Init();
    hInst = _AddCDC();
    USBD_SetDeviceInfo(&_DeviceInfo);
    USBD_Start();
    while (1) {
        static char _ac[USB_HS_BULK_MAX_PACKET_SIZE];
        int          NumBytesReceived;

        //
        // Wait for configuration
        //
        while ((USBD_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 = USBD_CDC_Receive(hInst, &_ac[0], sizeof(_ac), 0);
        if (NumBytesReceived > 0) {
            USBD_CDC_Write(hInst, &_ac[0], NumBytesReceived, 0);
        }
    }
}

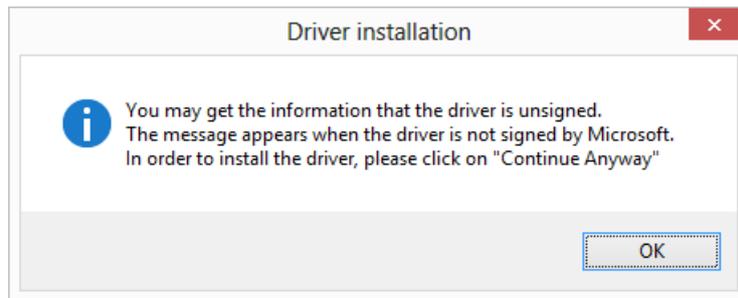
```

10.3 Installing the driver

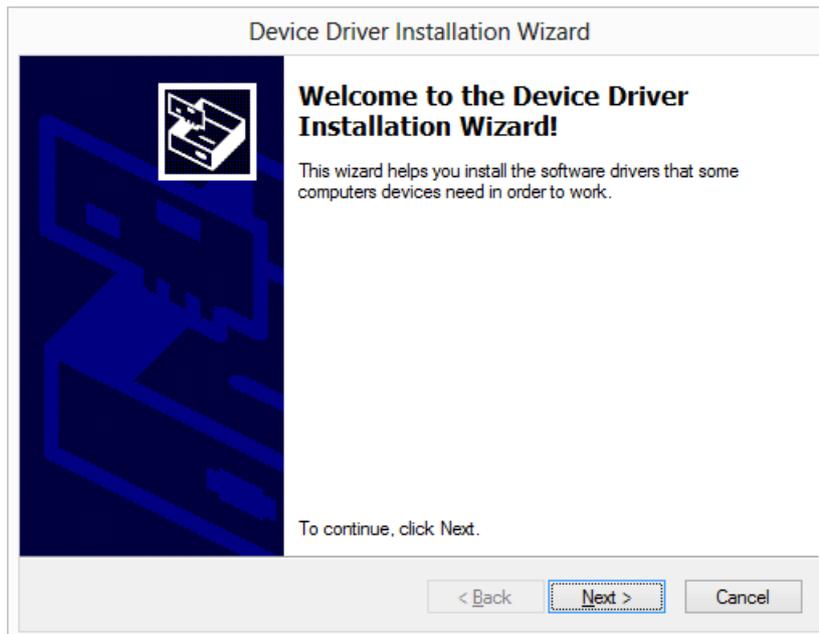
Before connecting the device to the PC, simply start the driver installer (InstallDriver.exe) which is located in Windows\USB\CDC\ folder. After the executable is executed Windows may show the UAC message that a specific install program is started. Confirm with Yes to start installation:

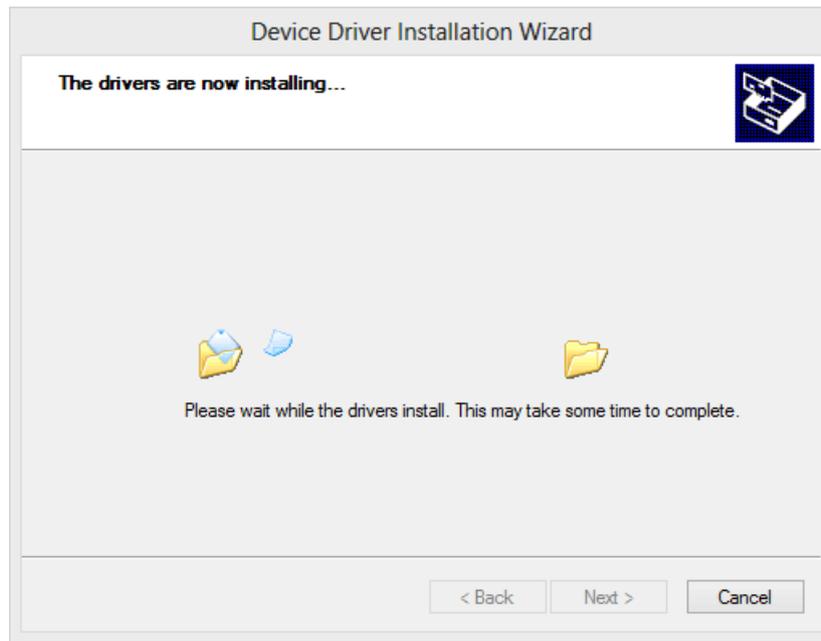


The driver installer will notify that Windows will show up a warning message that the driver is not signed by Microsoft. This is normal warning for driver that have been signed with a valid code signing certificate. Confirm read of the message by clicking on the OK button.



The Device Driver Installation Wizard dialog shows up. Click on Next to continue with the driver installation.

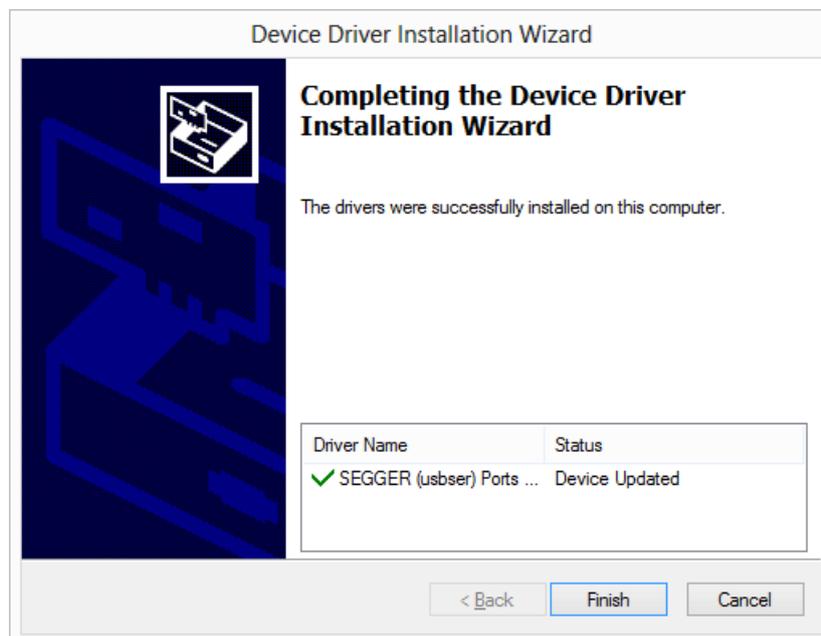




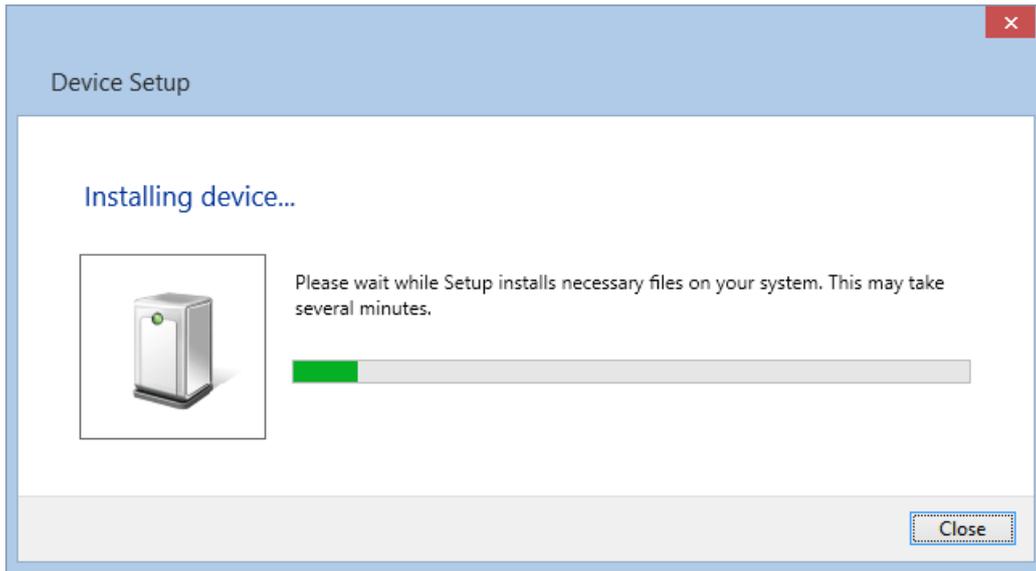
During the driver installation you may be asked whether the device software should be installed. Since the package is signed with a valid code signing certificate, simply answer the question with "Install".



Once the driver is installed the driver wizard will confirm that the driver is successfully installed.

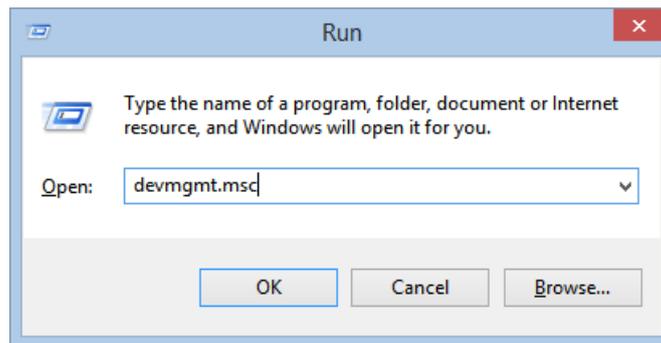


Now the target device can be connected with the PC. Windows will detect the new hardware.

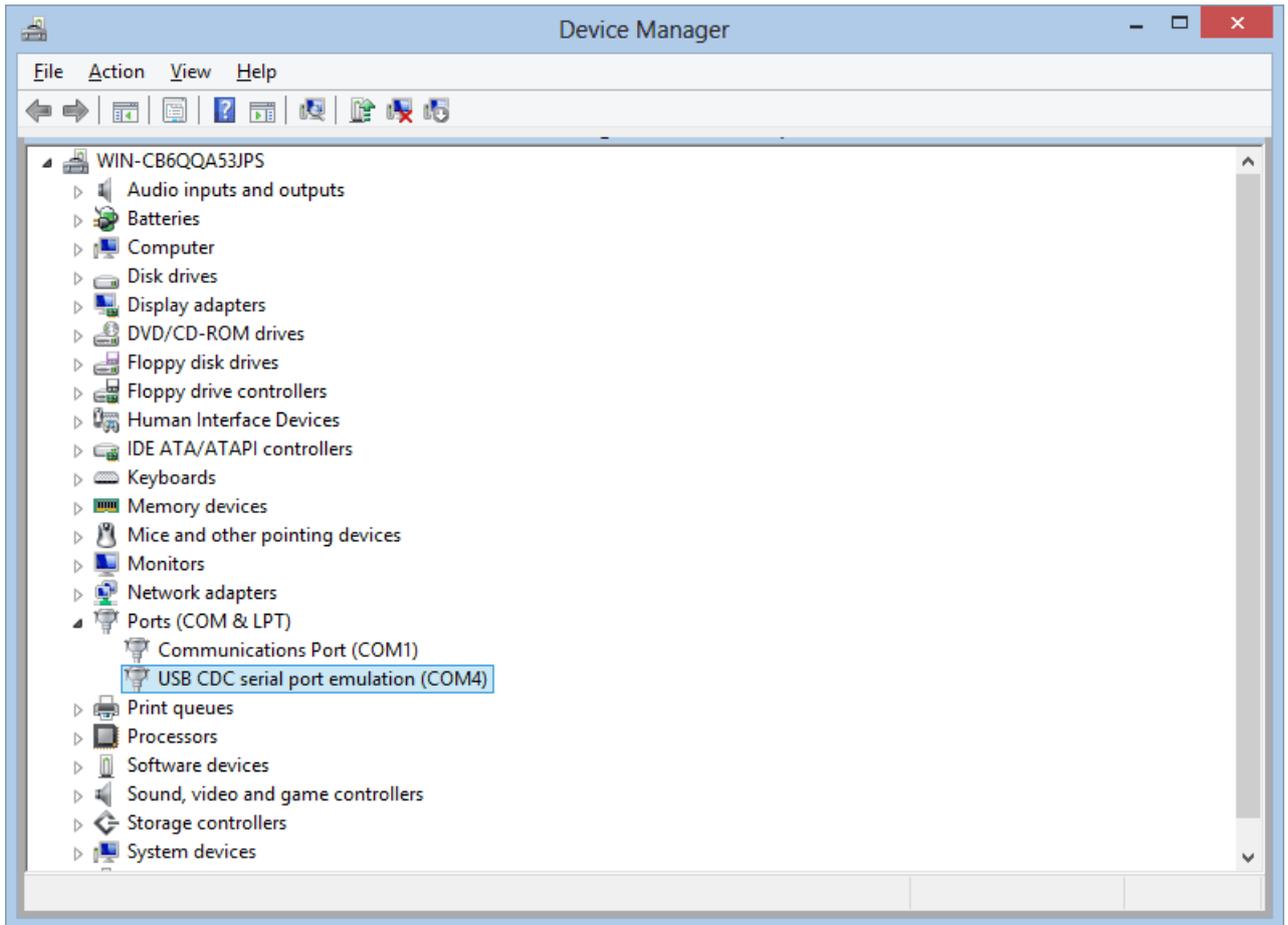


It will try to find the appropriate driver for the device. After the previous driver installation, the driver should be found quite fast and will install it for the device.

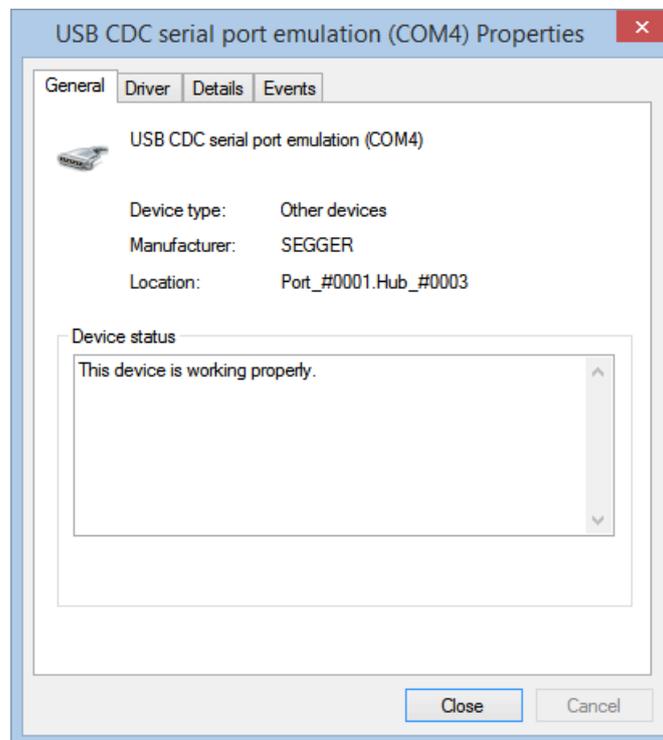
To verify that the driver installation was successful, open the device manager by opening the run dialog with [Windows Key]+[r] and type `devmgmt.msc` in the Open edit field:



The device manager will open and will show under Ports (COM & LPT) the installed device. Normally it will be called "USB CDC serial emulation (COMx)" whereas x the assigned COM port number is.



Double click on the device will show the device properties. Within the Device properties window the device status will be shown as the "The device is working properly".



10.3.1 The .inf file

The .inf file is required for installation.

It is defined as:

```

;
; Device installation file for
; USB 2 COM port emulation
; Copyright (c) 2006-2014 by SEGGER Microcontroller GmbH & Co. KG
;
; This file supports:
;   Windows 2000
;   Windows XP
;   Windows Server 2003 x86, x64
;   Windows Vista x86, x64
;   Windows 7 x86,x64
;   Windows Server 2008 x86,x64
;   Windows 8 x86,x64
;   Windows 8.1 x86,x64
;
[Version]
Signature="$Windows NT$"
Class=Ports
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}
Provider=%MFGNAME%
DriverVer=03/12/2015,6.0.2600.5
CatalogFile=usbser.cat

[Manufacturer]
%MFGNAME%=CDCDevice,NT,NTamd64

[DestinationDirs]
DefaultDestDir = 12

[CDCDevice.NT]
%DESCRIPTION%=DriverInstall,USB\VID_8765&PID_0234

[CDCDevice.NTamd64]
%DESCRIPTION%=DriverInstall,USB\VID_8765&PID_0234

[DriverInstall]
Include=mdmcpq.inf
CopyFiles=FakeModemCopyFileSection
AddReg=DriverInstall.AddReg

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

[DriverInstall.Services]
AddService=usbser, 0x00000002, DriverServiceInst

[DriverServiceInst]
DisplayName=%SERVICE%
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%12%\usbser.sys

[SourceDisksNames.amd64]
1=%DiskName%, ,

[SourceDisksFiles.amd64]

[SourceDisksNames.x86]

```

```
l=%DiskName%, ,
[SourceDisksFiles.x86]
[FakeModemCopyFileSection]
[Strings]
MFGNAME = "SEgger"
DESCRIPTION = "USB CDC serial port emulation"
SERVICE = "USB CDC serial port emulation"
DiskName="SEgger USB CDC Installation Disk"
```

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 are the value that have been passed by the function `USBD_SetDeviceInfo()`.

10.3.2 Signing the package

Newer Windows versions from Windows Vista on, will only install a driver when the driver package is signed. This signment must be done with a valid code signing certificate. For testing purpose the package also be signed in a testing environment, please refer to:

<https://msdn.microsoft.com/en-us/windows/hardware/drivers/install/test-signing-driver-packages>

10.3.3 Testing communication to the USB device

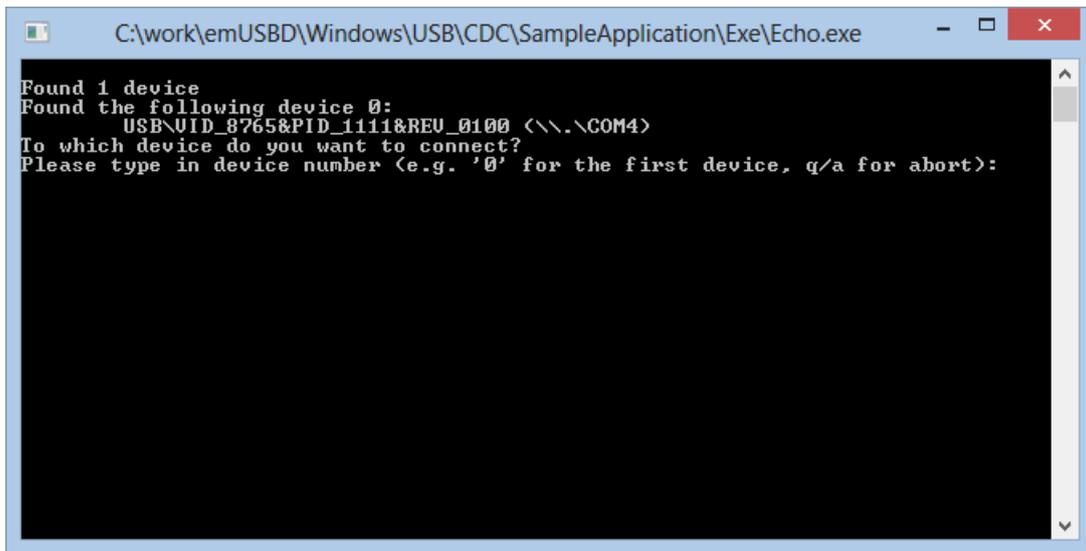
Accessing CDC on Windows

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. A simple Windows sample application is available to test the start application. The application is located in `Windows\USB\CDC\SampleApplication\Exe`.

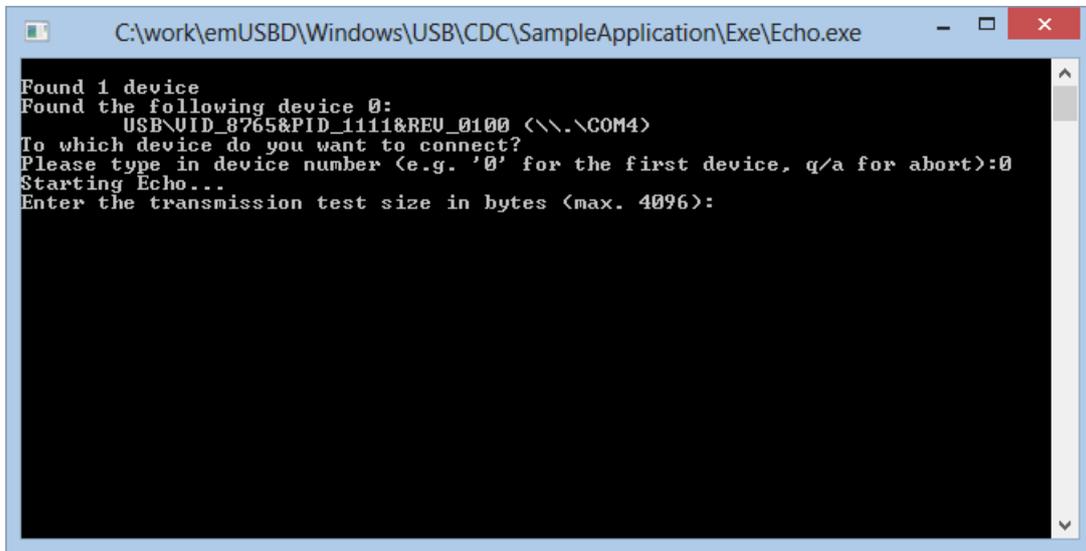
Alternatively any terminal program, such as PuTTY or TerraTerm or RealTerm, can be used to check the connectivity.

This section shows how to start and make the first run of the sample application.

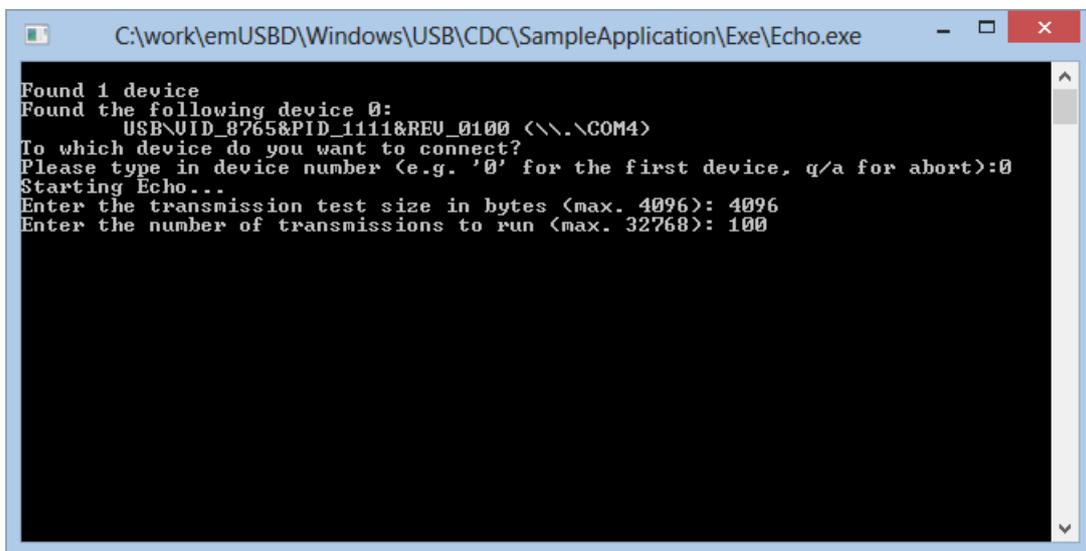
- Go to the `Windows\USB\CDC\SampleApplication\Exe` folder double click on the Echo application. A console window will be open and will show that one device has been found with the desired CDC Product and Vendor ID. Enter 0 to connect to that device.



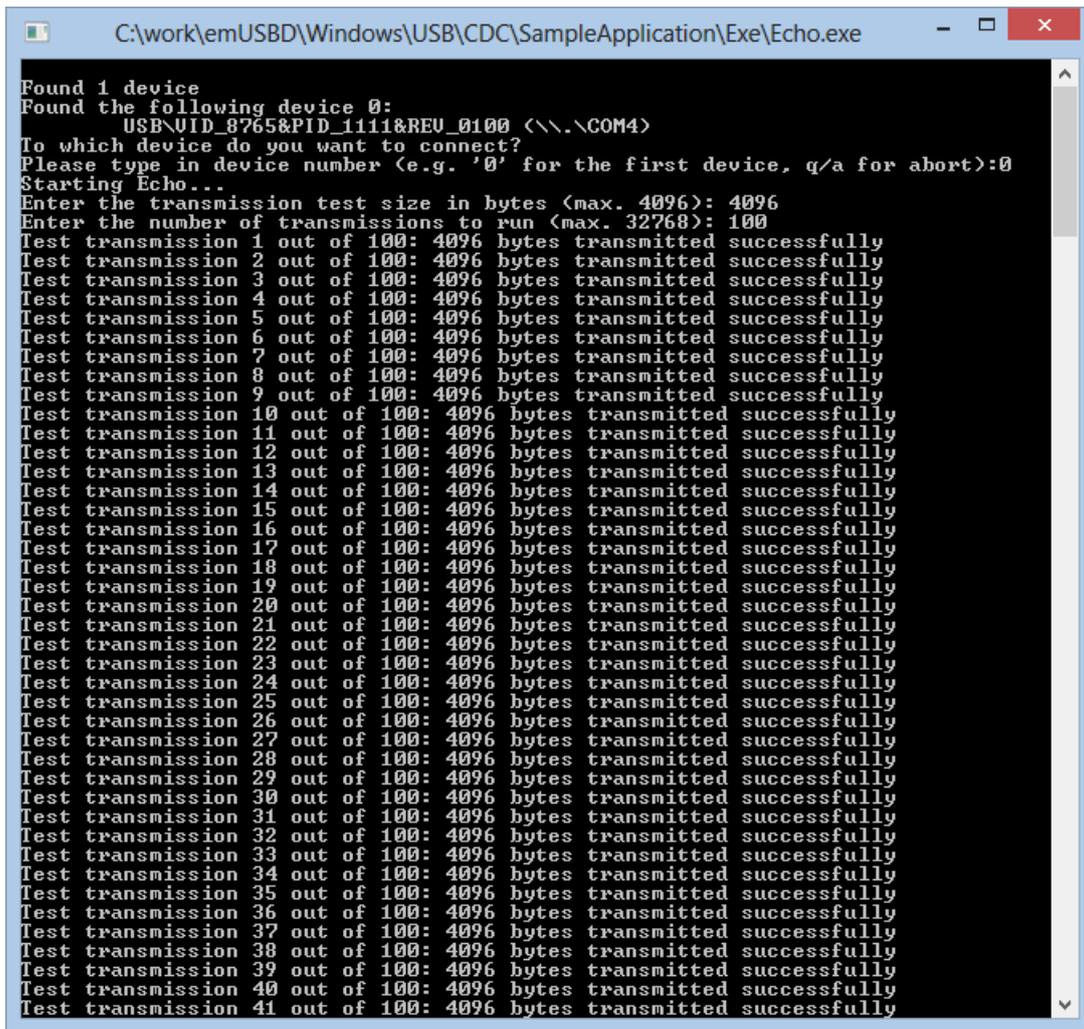
- The application will ask for the amount of bytes the application shall send to and receive from the target device.



- Now enter the number of repetitions the application shall send and receive to or from device and confirm with [Enter]..



- The test will run and should look like the following screenshot:



```
C:\work\emUSB\Windows\USB\CDC\SampleApplication\Exe\Echo.exe
Found 1 device
Found the following device 0:
    USB\VID_0765&PID_1111&REV_0100 (\.\COM4)
To which device do you want to connect?
Please type in device number (e.g. '0' for the first device, q/a for abort):0
Starting Echo...
Enter the transmission test size in bytes (max. 4096): 4096
Enter the number of transmissions to run (max. 32768): 100
Test transmission 1 out of 100: 4096 bytes transmitted successfully
Test transmission 2 out of 100: 4096 bytes transmitted successfully
Test transmission 3 out of 100: 4096 bytes transmitted successfully
Test transmission 4 out of 100: 4096 bytes transmitted successfully
Test transmission 5 out of 100: 4096 bytes transmitted successfully
Test transmission 6 out of 100: 4096 bytes transmitted successfully
Test transmission 7 out of 100: 4096 bytes transmitted successfully
Test transmission 8 out of 100: 4096 bytes transmitted successfully
Test transmission 9 out of 100: 4096 bytes transmitted successfully
Test transmission 10 out of 100: 4096 bytes transmitted successfully
Test transmission 11 out of 100: 4096 bytes transmitted successfully
Test transmission 12 out of 100: 4096 bytes transmitted successfully
Test transmission 13 out of 100: 4096 bytes transmitted successfully
Test transmission 14 out of 100: 4096 bytes transmitted successfully
Test transmission 15 out of 100: 4096 bytes transmitted successfully
Test transmission 16 out of 100: 4096 bytes transmitted successfully
Test transmission 17 out of 100: 4096 bytes transmitted successfully
Test transmission 18 out of 100: 4096 bytes transmitted successfully
Test transmission 19 out of 100: 4096 bytes transmitted successfully
Test transmission 20 out of 100: 4096 bytes transmitted successfully
Test transmission 21 out of 100: 4096 bytes transmitted successfully
Test transmission 22 out of 100: 4096 bytes transmitted successfully
Test transmission 23 out of 100: 4096 bytes transmitted successfully
Test transmission 24 out of 100: 4096 bytes transmitted successfully
Test transmission 25 out of 100: 4096 bytes transmitted successfully
Test transmission 26 out of 100: 4096 bytes transmitted successfully
Test transmission 27 out of 100: 4096 bytes transmitted successfully
Test transmission 28 out of 100: 4096 bytes transmitted successfully
Test transmission 29 out of 100: 4096 bytes transmitted successfully
Test transmission 30 out of 100: 4096 bytes transmitted successfully
Test transmission 31 out of 100: 4096 bytes transmitted successfully
Test transmission 32 out of 100: 4096 bytes transmitted successfully
Test transmission 33 out of 100: 4096 bytes transmitted successfully
Test transmission 34 out of 100: 4096 bytes transmitted successfully
Test transmission 35 out of 100: 4096 bytes transmitted successfully
Test transmission 36 out of 100: 4096 bytes transmitted successfully
Test transmission 37 out of 100: 4096 bytes transmitted successfully
Test transmission 38 out of 100: 4096 bytes transmitted successfully
Test transmission 39 out of 100: 4096 bytes transmitted successfully
Test transmission 40 out of 100: 4096 bytes transmitted successfully
Test transmission 41 out of 100: 4096 bytes transmitted successfully
```

Accessing CDC on Linux

On Linux no drivers are needed, the device should show up as /dev/ttyACM0 or similar. "sudo screen /dev/ttyACM0 115200" can be used to access the device.

Accessing CDC on macOS

On macOS no drivers are needed, the device should show up as /dev/tty.usbmo-dem13245678 or similar. The "screen" terminal program can be used to access the device.

10.4 Target API

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

10.4.1 Interface function list

Name	Description
API functions	
<code>USBD_CDC_Add()</code>	Adds a CDC-ACM class to the stack.
<code>USBD_CDC_CancelRead()</code>	Cancel a pending read operation.
<code>USBD_CDC_CancelWrite()</code>	Cancel a pending write operation.
<code>USBD_CDC_Read()</code>	Reads data from the host.
<code>USBD_CDC_ReadOverlapped()</code>	Reads data from the host asynchronously.
<code>USBD_CDC_Receive()</code>	Reads data from the host.
<code>USBD_CDC_ReceivePoll()</code>	Reads data from the host.
<code>USBD_CDC_ReadAsync()</code>	Reads data from the host asynchronously.
<code>USBD_CDC_SetOnBreak()</code>	Sets a callback in order to inform the application about a break condition sent by the host.
<code>USBD_CDC_SetOnLineCoding()</code>	Sets a user callback that shall be called when a <code>SET_LINE_CODING</code> request is sent to the device.
<code>USBD_CDC_SetOnControlLineState()</code>	Sets a user callback that shall be called when a <code>SET_LINE_STATE</code> request is sent to the device.
<code>USBD_CDC_SetOnRXEvent()</code>	Sets a callback function for the OUT endpoint that will be called on every RX event for that endpoint.
<code>USBD_CDC_SetOnTXEvent()</code>	Sets a callback function for the IN endpoint that will be called on every TX event for that endpoint.
<code>USBD_CDC_UpdateSerialState()</code>	Sets the new serial state.
<code>USBD_CDC_Write()</code>	Writes data to the host.
<code>USBD_CDC_WriteAsync()</code>	Sends data to the host asynchronously.
<code>USBD_CDC_WaitForRX()</code>	This function is to be used in combination with <code>USBD_CDC_ReadOverlapped()</code> and waits for the reading data transfer from the host to complete.
<code>USBD_CDC_PollForRX()</code>	This function is to be used in combination with <code>USBD_CDC_ReadOverlapped()</code> and waits for the reading data transfer from the host to complete.
<code>USBD_CDC_WaitForTX()</code>	This function is to be used in combination with a non-blocking call to <code>USBD_CDC_Write()</code> .
<code>USBD_CDC_PollForTX()</code>	This function is to be used in combination with a non-blocking call to <code>USBD_CDC_Write()</code> .
<code>USBD_CDC_WaitForTXReady()</code>	Waits (blocking) until the TX queue can accept another data packet.

Name	Description
<code>USBD_CDC_WriteSerialState()</code>	Sends the current control line serial state to the host.
<code>USBD_CDC_GetNumBytesRemToRead()</code>	This function is to be used in combination with <code>USBD_CDC_ReadOverlapped()</code> .
<code>USBD_CDC_GetNumBytesRemToWrite()</code>	This function is to be used in combination with a non-blocking call to <code>USBD_CDC_Write()</code> .
<code>USBD_CDC_GetNumBytesInBuffer()</code>	Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer.
Data structures	
<code>USB_CDC_INIT_DATA</code>	Initialization structure that is needed when adding a CDC interface to emUSB-Device.
<code>USB_CDC_LINE_CODING</code>	Structure that contains the new line-coding information sent by the host.
<code>USB_CDC_SERIAL_STATE</code>	Structure that contains the serial state that can be sent to the host.
<code>USB_CDC_CONTROL_LINE_STATE</code>	Structure that contains the new control line state sent by the host.

10.4.1.1 USBD_CDC_Add()

Description

Adds a CDC-ACM class to the stack.

Prototype

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

Parameters

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

Return value

Handle to a valid CDC instance. The handle of the first CDC instance is always 0.

Additional information

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

For creating more than one CDC instance please make sure the `USB_EnableIAD()` is called before, otherwise none but the first CDC instance will work correctly. The same is true for composite devices including CDC and another, different USB class.

10.4.1.2 USBD_CDC_CancelRead()

Description

Cancel a pending read operation.

Prototype

```
void USBD_CDC_CancelRead(USB_CDC_HANDLE hInst);
```

Parameters

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

Additional information

This function can be called when a pending asynchronous read operation (triggered by `USB_D_CDC_ReadOverlapped()`) should be canceled. The function can be called from any task.

The function can also be used to cancel a call to one of the blocking read functions (when called from a different task or interrupt function).

10.4.1.3 USBD_CDC_CancelWrite()

Description

Cancel a pending write operation.

Prototype

```
void USBD_CDC_CancelWrite(USB_CDC_HANDLE hInst);
```

Parameters

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

Additional information

This function shall be called when a pending asynchronous write operation (triggered by non-blocking call to `USB_D_CDC_Write()`) should be canceled. It can be called from any task.

The function can also be used to cancel a call to a blocking write functions (when called from a different task or interrupt function).

10.4.1.4 USBD_CDC_Read()

Description

Reads data from the host.

Prototype

```
int USBD_CDC_Read(USB_CDC_HANDLE hInst,
                  void *pData,
                  unsigned NumBytes,
                  unsigned Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USB_D_CDC_Ad-</code> <code>d()</code> .
<code>pData</code>	Pointer to a buffer where the received data will be stored.
<code>NumBytes</code>	Number of bytes to read.
<code>Timeout</code>	<code>Timeout</code> given in milliseconds. A zero value results in an infi- nite timeout.

Return value

= `NumBytes` Requested data was successfully read within the given timeout.
 ≥ 0 && < `NumBytes` `Timeout` has occurred (Number of bytes read before timeout).
 < 0 An error occurred.

Additional information

This function blocks the task until all data has been read or a timeout occurs. In case of a reset or a disconnect `USB_STATUS_ERROR` is returned.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USB_D_AddEP()`. This data can be retrieved by a later call to `USB_D_CDC_Receive()` / `USB_D_CDC_Read()`. See also `USB_D_CDC_GetNumBytesInBuffer()`.

In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 113).

10.4.1.5 USBDCDC_ReadOverlapped()

Description

Reads data from the host asynchronously.

Prototype

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

Parameters

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

Return value

- ≥ 0 Number of bytes that have been read from the internal buffer (success).
- $= 0$ No data was found in the internal buffer (success).
- < 0 An error occurred.

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, `USBDCDC_WaitForRX()` needs to be called.

Another synchronization method would be to periodically call `USBDCDC_GetNumBytesRemToRead()` in order to see how many bytes still need to be received (this method is preferred when a non-blocking solution is necessary).

The read operation can be canceled using `USBDCDC_CancelRead()`.

The buffer pointed to by [pData](#) must be valid until the read operation is terminated.

Example

See `USBDCDC_GetNumBytesRemToRead` on page 370.

10.4.1.6 USBDCDC_Receive()

Description

Reads data from the host. The function blocks until any data have been received. In contrast to `USBDCDC_Read()` this function does not wait for all of `NumBytes` to be received, but returns after the first packet has been received or after the timeout occurs. In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 113).

Prototype

```
int USBDCDC_Receive(USB_CDC_HANDLE  hInst,
                   void              * pData,
                   unsigned          NumBytes,
                   int               Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USBDCDC_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>Timeout</code>	<code>Timeout</code> given in milliseconds. A zero value results in an infinite timeout. If <code>Timeout</code> is -1, the function never blocks and only reads data from the internal endpoint buffer.

Return value

- > 0 Number of bytes that have been read within the given timeout.
- = 0 A timeout occurred (if `Timeout > 0`), zero packet received (not every controller supports this!), no data in buffer (if `Timeout < 0`) or the target was disconnected during the function call and no data was read so far.
- < 0 An error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling `USBDCDC_Receive()` will return as much data as is currently available up to the size of the buffer specified within the specified timeout. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return the number of bytes read so far. If the target was disconnected before this function was called, it returns `USB_STATUS_ERROR`.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USBDCDC_AddEP()`. This data can be retrieved by a later call to `USBDCDC_Receive()` / `USBDCDC_Read()`. See also `USBDCDC_GetNumBytesInBuffer()`.

A call of `USBDCDC_Receive(Inst, NULL, 0, -1)` can be used to trigger an asynchronous read that stores the data into the internal buffer.

10.4.1.7 USBD_CDC_ReceivePoll()

Description

Reads data from the host. The function blocks until any data have been received. In contrast to `USBDCDC_Read()` this function does not wait for all of `NumBytes` to be received, but returns after the first packet has been received or after the timeout occurs. In contrast to `USBDCDC_Receive()` this function will continue the read transfer asynchronously in case of a timeout.

Prototype

```
int USBDCDC_ReceivePoll(USB_CDC_HANDLE hInst,
                       void * pData,
                       unsigned NumBytes,
                       unsigned Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USBDCDC_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>Timeout</code>	<code>Timeout</code> given in milliseconds. A zero value results in an infinite timeout.

Return value

- > 0 Number of bytes that have been read within the given timeout.
- = 0 A timeout occurred (if `Timeout > 0`), zero packet received (not every controller supports this!), no data in buffer (if `Timeout < 0`) or the target was disconnected during the function call and no data was read so far.
- < 0 An error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling `USBDCDC_ReceivePoll()` will return as much data as is currently available up to the size of the buffer specified within the specified timeout. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return the number of bytes read so far. If the target was disconnected before this function was called, it returns `USB_STATUS_ERROR`.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USBDCDC_AddEP()`. This data can be retrieved by a later call to `USBDCDC_ReceivePoll()` / `USBDCDC_Read()`. See also `USBDCDC_GetNumBytesInBuffer()`.

If a timeout occurs, the read transfer is not affected. Data send from the host after the timeout is stored into the internal buffer of the endpoint and can be read by later calls to `USBDCDC_ReceivePoll()`.

If `Timeout = 0`, the function behaves like `USBDCDC_Receive()`.

10.4.1.8 USBD_CDC_ReadAsync()

Description

Reads data from the host asynchronously. The function does not wait for the data to be received. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_CDC_ReadAsync(USB_CDC_HANDLE      hInst,
                        USB_ASYNC_IO_CONTEXT * pContext,
                        int                   ShortRead);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USB_D_CDC_Ad-</code> <code>d()</code> .
<code>pContext</code>	Pointer to an I/O context containing parameters and pointer to the callback function.
<code>ShortRead</code>	<ul style="list-style-type: none"> 0: The transfer is completed successfully after all bytes have been read. 1: The transfer is completed successfully after one packet has been read.

10.4.1.9 USBDCDC_SetOnBreak()

Description

Sets a callback in order to inform the application about a break condition sent by the host.

Prototype

```
void USBDCDC_SetOnBreak(USB_CDC_HANDLE hInst,
                        USB_CDC_ON_SET_BREAK * pfOnBreak);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USBDCDC_Ad-</code> <code>d()</code> .
<code>pfOnBreak</code>	Pointer to the callback function.

Additional information

The callback is called in an ISR context, therefore it should should execute quickly.

The callback function has the following prototype:

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

Parameter	Description
<code>BreakDuration</code>	Length of the break signal in milliseconds. If <code>Break-</code> <code>Duration</code> is <code>0xFFFF</code> , this is counted as a permanent break condition. A <code>SendBreak</code> request with <code>Break-</code> <code>Duration</code> of <code>0x0000</code> will reset the break state.

10.4.1.10 USBDCDC_SetOnLineCoding()

Description

Sets a user callback that shall be called when a `SET_LINE_CODING` request is sent to the device.

Prototype

```
void USBDCDC_SetOnLineCoding(USB_CDC_HANDLE          hInst,
                             USB_CDC_ON_SET_LINE_CODING * pf);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USBDCDC_Ad-</code> <code>d()</code> .
<code>pf</code>	Pointer to the callback function.

Additional information

This function is used to register a user callback which notifies the application that the host has changed the line coding.

The callback is called in an ISR context, therefore it should execute quickly.

The callback function has the following prototype:

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

Parameter	Description
<code>pLineCoding</code>	Pointer to <code>USB_CDC_LINE_CODING</code> structure containing the new line coding parameters sent from the host.

10.4.1.11 USB_D_CDC_SetOnControlLineState()

Description

Sets a user callback that shall be called when a `SET_LINE_STATE` request is sent to the device.

Prototype

```
void USB_D_CDC_SetOnControlLineState(USB_CDC_HANDLE hInst,
                                     USB_CDC_ON_SET_CONTROL_LINE_STATE * pf);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USB_D_CDC_Ad-</code> <code>d()</code> .
<code>pf</code>	Pointer to the callback function.

Additional information

This function is used to register a user callback which notifies the application that the host has changed the control line state.

The callback is called in an ISR context, therefore it should execute quickly.

The callback function has the following prototype:

```
typedef void USB_CDC_ON_SET_CONTROL_LINE_STATE(USB_CDC_CONTROL_LINE_STATE * pLineState);
```

Parameter	Description
<code>pLineState</code>	Pointer to <code>USB_CDC_CONTROL_LINE_STATE</code> structure containing the new line state parameters sent from the host.

10.4.1.12 USBDCDC_SetOnRXEvent()

Description

Sets a callback function for the OUT endpoint that will be called on every RX event for that endpoint.

Prototype

```
void USBDCDC_SetOnRXEvent(USB_CDC_HANDLE hInst,
                          USB_EVENT_CALLBACK * pEventCb,
                          USB_EVENT_CALLBACK_FUNC * pfEventCb,
                          void * pContext);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USBDCDC_Add()</code> .
<code>pEventCb</code>	Pointer to a <code>USB_EVENT_CALLBACK</code> structure (will be initialized by this function).
<code>pfEventCb</code>	Pointer to the callback routine that will be called on every event on the USB endpoint.
<code>pContext</code>	A pointer which is used as parameter for the callback function.

Additional information

The `USB_EVENT_CALLBACK` structure is private to the USB stack. It will be initialized by `USBDCDC_SetOnRXEvent()`. The USB stack keeps track of all event callback functions using a linked list. The `USB_EVENT_CALLBACK` structure will be included into this linked list and must reside in static memory.

The callback function is called only, if a read operation was started using one of the `USBDCDC_Read...()` or `USBDCDC_Receive()` functions.

The callback function has the following prototype:

```
typedef void USB_EVENT_CALLBACK_FUNC(unsigned Events, void *pContext);
```

Parameter	Description
<code>Events</code>	A bit mask indicating which events occurred on the endpoint.
<code>pContext</code>	The pointer which was provided to the <code>USBDCDC_SetOnEvent</code> function.

Note that the callback function will be called within an ISR, therefore it should never block. The first parameter to the callback function will contain a bit mask for all events that triggered the call:

Event	Description
<code>USB_EVENT_DATA_READ</code>	Some data was received from the host on the endpoint.
<code>USB_EVENT_READ_COMPLETE</code>	The last read operation was completed.
<code>USB_EVENT_READ_ABORT</code>	A read transfer was aborted.

Example

```
// The callback function.
static void _OnEvent(unsigned Events, void *pContext) {
    unsigned NumBytes;

    if (Events & USB_EVENT_DATA_READ) {
        NumBytes = USBDCDC_GetNumBytesInBuffer(hInst);
        if (NumBytes) {
            //
            // The call to receive will read all data from
            // the internal buffer and will start a new transfer.
            // The new transfer will again generate a new event when new data arrives.
            //
            // Note that a new transfer is only started when
            // the internal buffer is completely empty.
            // (It will be empty if you read the number of bytes
            // USBDCDC_GetNumBytesInBuffer returns.)
            //
            r = USBDCDC_Receive(hInst, Buff, NumBytes, -1);
            if (r > 0) {
                <.. process data in Buff..>
            }
        }
    }
}

// Main program.
// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USBDCDC_SetOnRXEvent(hInst, &_usb_callback, _OnEvent, NULL);
// Trigger first read
USBDCDC_Receive(Inst, NULL, 0, -1);
<.. do anything else here while the data is processed in the callback ..>
```

10.4.1.13 USBDCDC_SetOnTXEvent()

Description

Sets a callback function for the IN endpoint that will be called on every TX event for that endpoint.

Prototype

```
void USBDCDC_SetOnTXEvent(USB_CDC_HANDLE      hInst,
                          USB_EVENT_CALLBACK  * pEventCb,
                          USB_EVENT_CALLBACK_FUNC * pfEventCb,
                          void                * pContext);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USBDCDC_Add()</code> .
<code>pEventCb</code>	Pointer to a <code>USB_EVENT_CALLBACK</code> structure (will be initialized by this function).
<code>pfEventCb</code>	Pointer to the callback routine that will be called on every event on the USB endpoint.
<code>pContext</code>	A pointer which is used as parameter for the callback function.

Additional information

The `USB_EVENT_CALLBACK` structure is private to the USB stack. It will be initialized by `USBDCDC_SetOnTXEvent()`. The USB stack keeps track of all event callback functions using a linked list. The `USB_EVENT_CALLBACK` structure will be included into this linked list and must reside in static memory.

The callback function is called only, if a write operation was started using one of the `USBDCDC_Write...()` functions.

The callback function has the following prototype:

```
typedef void USB_EVENT_CALLBACK_FUNC(unsigned Events, void *pContext);
```

Parameter	Description
<code>Events</code>	A bit mask indicating which events occurred on the endpoint.
<code>pContext</code>	The pointer which was provided to the <code>USBDCDC_SetOnEvent</code> function.

Note that the callback function will be called within an ISR, therefore it should never block.

The first parameter to the callback function will contain a bit mask for all events that triggered the call:

Event	Description
<code>USB_EVENT_DATA_SEND</code>	Some data was sent to the host, so that (part of) the user write buffer may be reused by the application.
<code>USB_EVENT_DATA_ACKED</code>	Some data was acknowledged by the host.
<code>USB_EVENT_WRITE_ABORT</code>	A write transfer was aborted.
<code>USB_EVENT_WRITE_COMPLETE</code>	All write operations were completed.

Example

```
// The callback function.

static void _OnEvent(unsigned Events, void *pContext) {
    if ((Events & USB_EVENT_DATA_SEND) != 0 &&
        // Check for last write transfer to be completed.
        USBDCDC_GetNumBytesRemToWrite(_hInst) == 0) {
        <.. prepare next data for writing..>
        // Send next packet of data.
        r = USBDCDC_Write(_hInst, &ac[0], 200, -1);
        if (r < 0) {
            <.. error handling..>
        }
    }
}

// Main program.

// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USBDCDC_SetOnTXEvent(hInst, &_usb_callback, _OnEvent, NULL);
// Send the first packet of data using an asynchronous write operation.
r = USBDCDC_Write(_hInst, &ac[0], 200, -1);
if (r < 0) {
    <.. error handling..>
}
<.. do anything else here while the whole data is send..>
```

10.4.1.14 USBD_CDC_UpdateSerialState()

Description

Sets the new serial state.

Prototype

```
void USBD_CDC_UpdateSerialState(      USB_CDC_HANDLE      hInst,
                                     const USB_CDC_SERIAL_STATE * pSerialState);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USB_D_CDC_Ad-</code> <code>d()</code> .
<code>pSerialState</code>	Pointer to the <code>USB_CDC_SERIAL_STATE</code> structure.

Additional information

This function updates the control line state internally. In order to inform the host about the serial state change, refer to the function `USB_D_CDC_WriteSerialState()`.

10.4.1.15 USBDCDC_Write()

Description

Writes data to the host. Depending on the `Timeout` parameter, the function may block until `NumBytes` have been written or a timeout occurs.

Prototype

```
int USBDCDC_Write(    USB_CDC_HANDLE  hInst,
                    const void      * pData,
                    unsigned        NumBytes,
                    int             Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USBDCDC_Ad-</code> <code>d()</code> .
<code>pData</code>	Pointer to data that should be sent to the host.
<code>NumBytes</code>	Number of bytes to be written.
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite. If <code>Timeout</code> is -1, the function returns immediately and the transfer is processed asynchronously.

Return value

= 0 Successful started an asynchronous write transfer or a timeout
has occurred and no data was written.

> 0 && < `NumBytes` Number of bytes that have been written before a timeout oc-
curred.

= `NumBytes` Write transfer successful completed.

< 0 An error occurred.

Additional information

This function also returns when the target is disconnected from host or when a USB reset occurred.

The USB stack is able to queue a small number of asynchronous write transfers (`Timeout` = -1). If a write transfer is still in progress when this function is called and the USB stack can not accept another write transfer request, the functions returns `USB_STATUS_EP_BUSY`. A synchronous write transfer (`Timeout` ≥ 0) will always block until the transfer (including all pending transfers) are finished.

In order to synchronize, `USBDCDC_WaitForTX()` needs to be called. Another synchronization method would be to periodically call `USBDCDC_GetNumBytesRemToWrite()` in order to see how many bytes still need to be written (this method is preferred when a non-blocking solution is necessary).

The write operation can be canceled using `USBDCDC_CancelWrite()`.

If `pData` = NULL and `NumBytes` = 0, a zero-length packet is sent to the host.

In case of a timeout, the write transfer is aborted (see *Timeout handling* on page 113).

The content of the buffer pointed to by `pData` must not be changed until the transfer has been completed.

10.4.1.16 USBD_CDC_WriteAsync()

Description

Sends data to the host asynchronously. The function does not wait for the data to be send. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_CDC_WriteAsync(USB_CDC_HANDLE      hInst,
                        USB_ASYNC_IO_CONTEXT * pContext,
                        char                  Send0PacketIfRequired);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USB_D_CDC_Ad-</code> <code>d()</code> .
<code>pContext</code>	Pointer to an I/O context containing parameters and pointer to the callback function.
<code>Send0PacketIfRequired</code>	Specifies that a zero-length packet shall be sent when the last data packet is a multiple of <code>MaxPacketSize</code> .

10.4.1.17 USBDCDC_WaitForRX()

Description

This function is to be used in combination with `USBDCDC_ReadOverlapped()` and waits for the reading data transfer from the host to complete.

Prototype

```
int USBDCDC_WaitForRX(USB_CDC_HANDLE hInst,
                     unsigned Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USBDCDC_Ad-</code> <code>d()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite.

Return value

- 0 Transfer completed.
- 1 `Timeout` occurred.

Additional information

This function shall be called in order to synchronize task with the read data transfer previously initiated. The function blocks until the number of bytes specified by `USBDCDC_ReadOverlapped()` has been read from the host. In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 113).

Example

```
if (USBDCDC_ReadOverlapped(hInst, &ac[0], 50) < 0) {
    <.. error handling..>
    return;
}
//
// USBDCDC_ReadOverlapped() will return immediately.
// Do something else while data may be transferred.
//
<...>
//
// Now wait until we get all 50 bytes.
// USBDCDC_WaitForRX() will block, until total of
// 50 bytes are read or timeout occurs.
//
if (USBDCDC_WaitForRX(hInst, timeout) != 0) {
    <.. timeout error handling..>
    return;
}
// Now we have 50 bytes of data.
// Process 50 bytes of data from ac[] here.
```

10.4.1.18 USBDCDC_PollForRX()

Description

This function is to be used in combination with `USBDCDC_ReadOverlapped()` and waits for the reading data transfer from the host to complete.

Prototype

```
int USBDCDC_PollForRX(USB_CDC_HANDLE hInst,
                     unsigned          Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USBDCDC_Ad-</code> <code>d()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite.

Return value

- 0 Transfer completed.
- 1 `Timeout` occurred.

Additional information

This function shall be called in order to synchronize task with the read data transfer previously initiated. The function blocks until the number of bytes specified by `USBDCDC_ReadOverlapped()` has been read from the host. In case of a timeout, the current transfer is not affected. The function may be called repeatedly until it does not report a timeout any more.

Example

```
if (USBDCDC_ReadOverlapped(hInst, &ac[0], 50) < 0) {
    <.. error handling..>
    return;
}
//
// USBDCDC_ReadOverlapped() will return immediately.
// While waiting for the data, we will blink a LED with 200 ms interval.
// USBDCDC_PollForRX() will return, if all data were read or 100 ms expired.
//
while ((r = USBDCDC_PollForRX(hInst, 100)) > 0) {
    ToggleLED();
}
if (r < 0) {
    <.. error handling..>
    return;
}
// Now we have 50 bytes of data.
// Process 50 bytes of data from ac[] here.
```

10.4.1.19 USBD_CDC_WaitForTX()

Description

This function is to be used in combination with a non-blocking call to `USBDCDC_Write()`. This function waits for the writing data transfer to the host to complete.

Prototype

```
int USBDCDC_WaitForTX(USB_CDC_HANDLE hInst,
                     unsigned Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USBDCDC_Ad-</code> <code>d()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite.

Return value

- 0 Transfer completed.
- 1 `Timeout` occurred.

Additional information

This function shall be called in order to synchronize task with the write data transfer previously initiated. This function blocks until the number of bytes specified by `USBDCDC_Write()` has been written to the host. In case of a timeout, the write transfer is aborted (see *Timeout handling* on page 113).

10.4.1.20 USBDCDC_PollForTX()

Description

This function is to be used in combination with a non-blocking call to `USBD_CDC_Write()`. This function waits for the writing data transfer to the host to complete.

Prototype

```
int USBD_CDC_PollForTX(USB_CDC_HANDLE hInst,
                      unsigned         Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USBD_CDC_Ad-</code> <code>d()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite.

Return value

- 0 Transfer completed.
- 1 `Timeout` occurred.

Additional information

This function shall be called in order to synchronize task with the write data transfer previously initiated. This function blocks until the number of bytes specified by `USBD_CDC_Write()` has been written to the host. In case of a timeout, the current transfer is not affected. The function may be called repeatedly until it does not report a timeout any more.

Example

```
if (USBD_CDC_Write(hInst, &ac[0], 50, -1) < 0) {
    <.. error handling..>
    return;
}
//
// USBD_CDC_Write() will return immediately.
// While waiting for the data to be transferred, we will blink a LED with
// 200 ms interval.
// USBD_CDC_PollForTX() will return, if all data were send or 100 ms expired.
//
while ((r = USBD_CDC_PollForTX(hInst, 100)) > 0) {
    ToggleLED();
}
if (r < 0) {
    <.. error handling..>
    return;
}
// Now all data have been send.
```

10.4.1.21 USBD_CDC_WaitForTXReady()

Description

Waits (blocking) until the TX queue can accept another data packet. This function is used in combination with a non-blocking call to `USBDCDC_Write()`, it waits until a new asynchronous write data transfer will be accepted by the USB stack.

Prototype

```
int USBDCDC_WaitForTXReady(USB_CDC_HANDLE hInst,
                           int             Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USBDCDC_Ad-</code> <code>d()</code> .
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite. If <code>Timeout</code> is neg- ative, the function will return immediately.

Return value

- = 0 A new asynchronous write data transfer will be accepted.
- = 1 The write queue is full, a call to `USBDCDC_Write()` would return `USB_S-`
`TATUS_EP_BUSY`.
- < 0 Error occurred.

Additional information

If `Timeout` is 0, the function never returns 1.

If `Timeout` is -1, the function will not wait, but immediately return the current state.

Example

```
// Always keep the write queue full for maximum send speed.
for (;;) {
    pData = GetNextData(&NumBytes);
    // Wait until stack can accept a new write.
    USBDCDC_WaitForTxReady(hInst, 0);
    // Put write transfer into the write queue.
    if (USBDCDC_Write(hInst, pData, NumBytes, -1) < 0) {
        <.. error handling..>
    }
}
```

10.4.1.22 USBDCDC_WriteSerialState()

Description

Sends the current control line serial state to the host.

Prototype

```
void USBDCDC_WriteSerialState(USB_CDC_HANDLE hInst);
```

Parameters

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

Additional information

This function shall be called in order to inform the host about the control serial state of the CDC instance. The current control line serial state can be set using `USBDCDC_UpdateSerialState()`.

10.4.1.23 USBDCDC_GetNumBytesRemToRead()

Description

This function is to be used in combination with `USBDCDC_ReadOverlapped()`. It returns the number of bytes which still have to be read during the transaction.

Prototype

```
int USBDCDC_GetNumBytesRemToRead(USB_CDC_HANDLE hInst);
```

Parameters

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

Return value

Number of bytes which still have to be read.

Additional information

Note that this function does not return the number of bytes that have been read, but the number of bytes which still have to be read. This function does not block.

Example

```
NumBytesReceived = USBDCDC_ReadOverlapped(hInst, &ac[0], 50);
if (NumBytesReceived < 0) {
    <.. error handling..>
}
if (NumBytesReceived > 0) {
    // Already had some data in the internal buffer.
    // The first 'NumBytesReceived' bytes may be processed here.
    <...>
} else {
    // Wait until we get all 50 bytes
    while (USBDCDC_GetNumBytesRemToRead(hInst) > 0) {
        USB_OS_Delay(50);
    }
}
```

10.4.1.24 USBDCDC_GetNumBytesRemToWrite()

Description

This function is to be used in combination with a non-blocking call to `USBDCDC_Write()`. It returns the number of bytes which still have to be written during the transaction.

Prototype

```
int USBDCDC_GetNumBytesRemToWrite(USB_CDC_HANDLE hInst);
```

Parameters

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

Return value

Number of bytes which still have to be written.

Additional information

Note that this function does not return the number of bytes that have been written, but the number of bytes which still have to be written. This function does not block.

Example

```
// NumBytesWritten will contain > 0 values
// if we had anything in the write buffer.
NumBytesWritten = USBDCDC_Write(hInst, &ac[0], TRANSFER_SIZE, -1);
if (NumBytesWritten < 0) {
    <.. error handling..>
}
// NumBytesToWrite shows how many bytes still have to be written.
while (USBDCDC_GetNumBytesRemToWrite(hInst) > 0) {
    USB_OS_Delay(50);
}
```

10.4.1.25 USBD_CDC_GetNumBytesInBuffer()

Description

Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer. This functions does not start a read transfer.

Prototype

```
int USBD_CDC_GetNumBytesInBuffer(USB_CDC_HANDLE hInst);
```

Parameters

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

Return value

Number of bytes which have been stored in the internal buffer.

Additional information

The number of bytes returned by this function can be read using `USB_D_CDC_Read()` without blocking.

10.4.2 Data structures

10.4.2.1 USB_CDC_INIT_DATA

Description

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

Type definition

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

Structure members

Member	Description
EPIn	Bulk IN endpoint for sending data to the host.
EPOut	Bulk OUT endpoint for receiving data from the host.
EPInt	Interrupt IN endpoint for sending status information.

10.4.2.2 USB_CDC_LINE_CODING

Description

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

Type definition

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

Structure members

Member	Description
DTERate	The data transfer rate for the device in bits per second.
CharFormat	Number of stop bits: <ul style="list-style-type: none"> • 0 - 1 Stop bit • 1 - 1.5 Stop bits • 2 - 2 Stop bits
ParityType	Specifies the parity type: <ul style="list-style-type: none"> • 0 - None • 1 - Odd • 2 - Even • 3 - Mark • 4 - Space
DataBits	Specifies the bits per byte: (5, 6, 7, 8)

10.4.2.3 USB_CDC_SERIAL_STATE

Description

Structure that contains the serial state that can be sent to the host.

Type definition

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

Structure members

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	1 - Break condition signaled.
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 a parity error.
OverRunError	When set to 1, the device indicates an over-run error.
CTS	Clear to Send: Deprecated, not used with USB.

Additional information

All members of the structure may have value 0 (false) or 1 (true).

10.4.2.4 USB_CDC_CONTROL_LINE_STATE

Description

Structure that contains the new control line state sent by the host.

Type definition

```
typedef struct {  
    U8  DTR;  
    U8  RTS;  
} USB_CDC_CONTROL_LINE_STATE;
```

Structure members

Member	Description
DTR	Data Terminal Ready.
RTS	Request To Send.

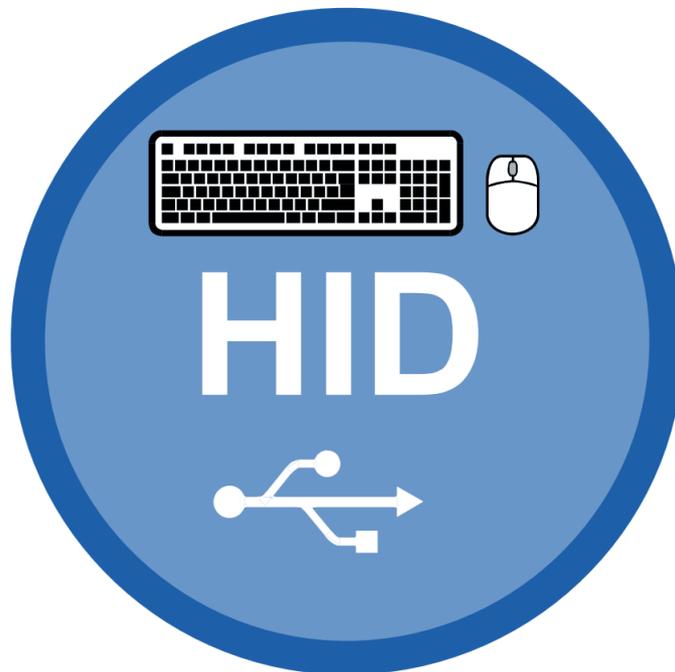
Additional information

All members of the structure may have value 0 (false) or 1 (true).

Chapter 11

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.



11.1 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.

11.1.1 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

11.1.2 Categories

Devices which are in the HID class generally fall into one of two categories:

*True HID*s and *vendor specific HID*s, explained below. One or more examples for both categories are provided.

11.1.2.1 True HID

True HID devices are devices which communicate directly with the host operating system, this includes devices which are used by a human to enter data, but do not directly exchange data with an application program running on the host.

Typical examples

- Keyboard
- Mouse and similar pointing devices
- Joystick
- Gamepad
- Bar-code reader
- Front-panel controls - for example, switches and buttons.

11.1.2.2 Vendor specific HID

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 cannot communicate with the device. When analyzing the report descriptor, the host finds that it cannot 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 lies between 0xFF00 and 0xFFFF.

An application has the chance to communicate with the particular device using API functions offered by the host. This enables 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 with older operating systems. Therefore it can be a good choice to select HID as a device class, especially if ease of use is important and high communication speed is not required.

Typical examples

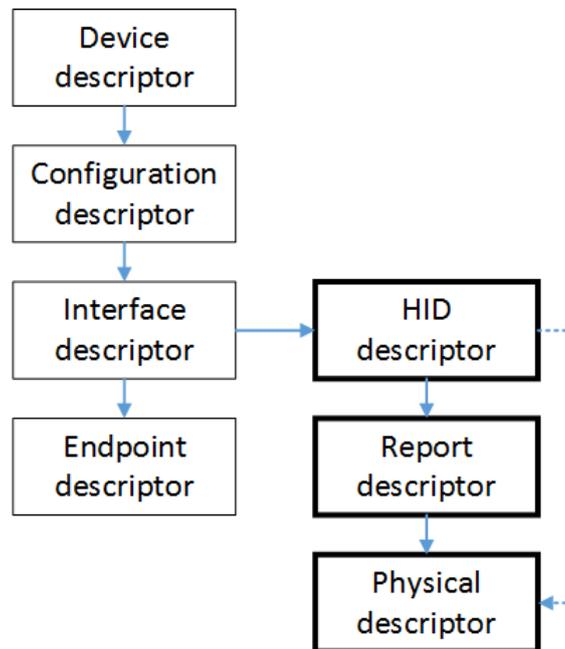
- Thermometer
- Voltmeter
- Low-speed JTAG emulator

- UPS (Uninterruptible power supply)

11.2 Background information

11.2.1 HID descriptors

This section presents an overview of 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, <https://www.usb.org>, for detailed information about the USB HID standard.



11.2.1.1 HID descriptor

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

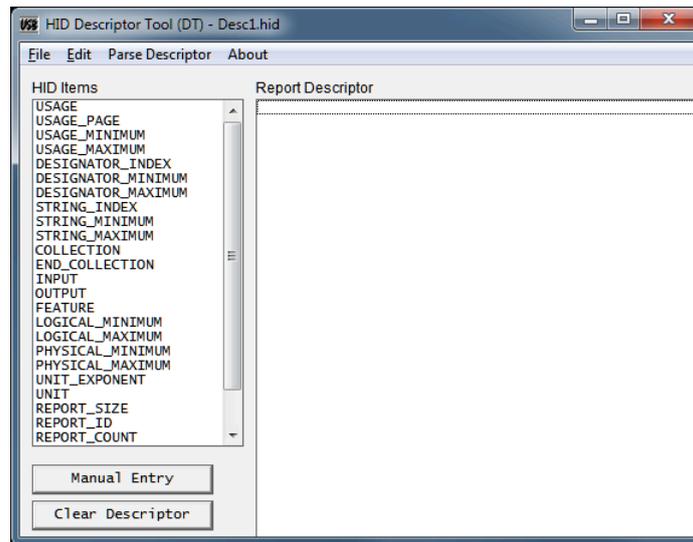
11.2.1.2 Report descriptor

Data 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 mice or keyboards. This implementation of HID always requires a report descriptor.

Using HID only transfers matching the report size are allowed, for example if a report is defined to be 64 bytes large in either direction only transfer of 64 bytes are allowed. If the application needs to transfer less data the packet must be padded by the application to match the report size. The report descriptor can define multiple reports of different sizes. In this case the first byte of the transfer must contain the report ID.

The USB Implementers Forum provides an application which helps to build and modify HID report descriptors. The HID Descriptor Tool can be downloaded from:

<https://www.usb.org/hid>



11.2.1.3 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 currently not supported.

11.3 Configuration

11.3.1 Initial configuration

To get emUSB-Device 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-Device should be used in your final product. Refer to the section *emUSB-Device Configuration* on page 48 for detailed information about the functions which must be adapted before you can release a final product version.

11.3.2 Final configuration

Generating a report descriptor

This step is only required if your product is a 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` to `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 examples use the value `0x12`; this value is defined at the top of the application program with the macro `USB_HID_DEFAULT_VENDOR_PAGE`.

11.4 Example application

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

The following start application files are provided:

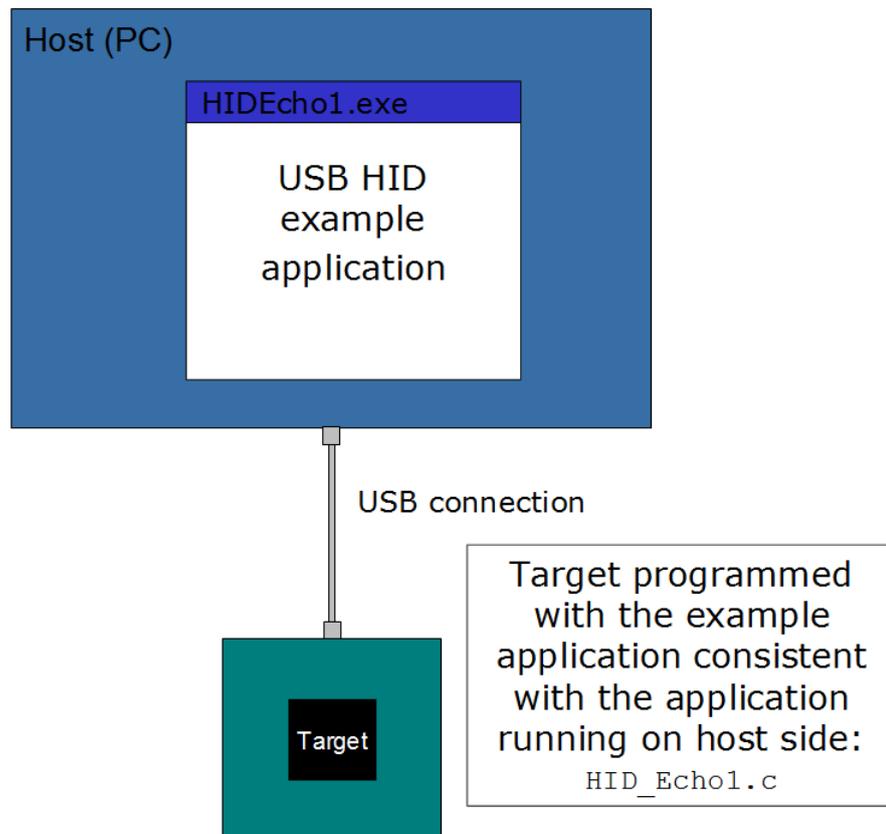
File	Description
USB_HID_Mouse.c	Simple mouse example. ("True HID" example)
USB_HID_Echo1.c	Modified echo server. ("vendor specific" example)

11.4.1 USB_HID_Mouse.c

USB_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, it moves the mouse cursor in an endless loop to the left and after a short delay back to the right.

11.4.2 USB_HID_Echo1.c

USB_HID_Echo1.c is a typical example for a "vendor-specific HID" implementation. The HID start application (USB_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 them back to the host.



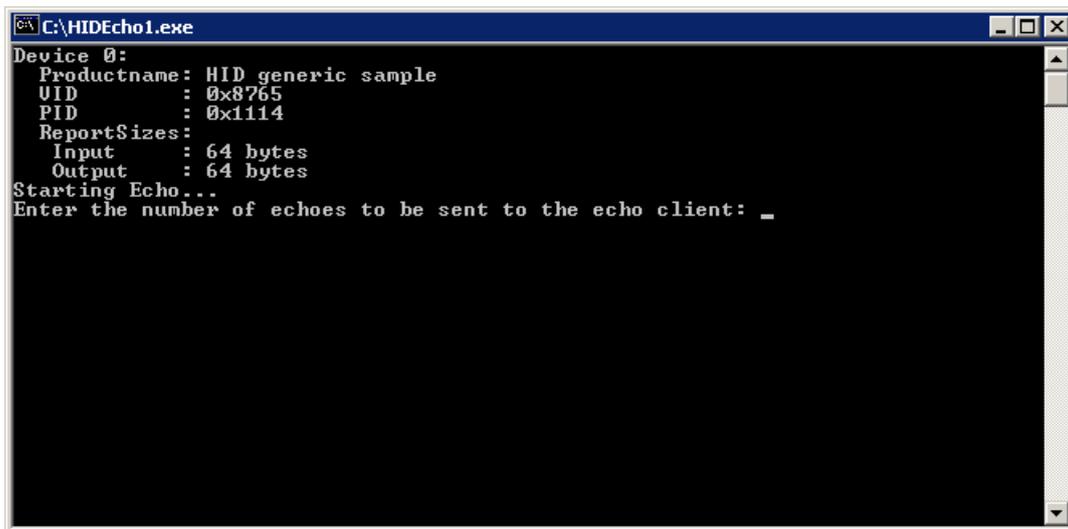
To use this application, include the source code file USB_HID_Echo1.c into your project and compile and download it into your target. Run HIDEcho1.exe after the target is connected to the host and the enumeration process has been completed. The PC application is supplied as executable in the Windows\USB\HID\SampleApp\Exe directory. The source code of the

PC example is also supplied. Refer to section Compiling the PC example application for more information to the PC example project.

11.4.2.1 Running the example

1. Add `USB_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 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 `USB_HID_Echo1.exe`:

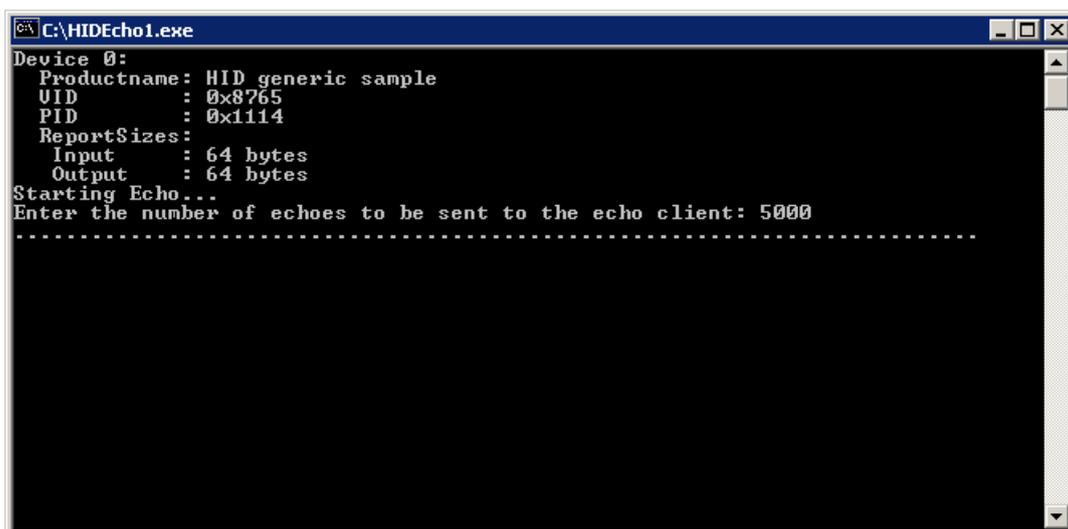


```

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 connected. 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
.....

```

11.4.2.2 Compiling the PC example application

Under Window you can build the sample by using the provided VisualStudio 2010 project. The source code of the example application is located in the subfolder `windows\USB\HID\SampleApp`. Open the file `USBHID_Start.sln` 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 <https://docs.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk> for more information.

Under Linux simply generate the executable by invoking `make` in the `Windows/USB/HID/SampleApp` folder in a shell

```
cd Windows/USB/HID/SampleApp
make
```

11.5 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. 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-Device are handled internally by the provided API functions.

11.5.1 Target interface function list

Function	Description
API functions	
<code>USBD_HID_Add()</code>	Adds HID class device to the USB interface.
<code>USBD_HID_GetNumBytesInBuffer()</code>	Returns the number of bytes available in the internal read buffer.
<code>USBD_HID_GetNumBytesRemToRead()</code>	Checks how many bytes still have to be read.
<code>USBD_HID_GetNumBytesRemToWrite()</code>	Checks how many bytes still have to be written.
<code>USBD_HID_Read()</code>	Reads data from the host with a given timeout.
<code>USBD_HID_ReadOverlapped()</code>	Reads data from the host asynchronously.
<code>USBD_HID_WaitForRX()</code>	This function is to be used in combination with <code>USBD_HID_ReadOverlapped()</code> .
<code>USBD_HID_WaitForTX()</code>	This function is to be used in combination with a non-blocking call to <code>USBD_HID_Write()</code> .
<code>USBD_HID_Write()</code>	Writes data to the host.
<code>USBD_HID_SetOnGetReportRequest()</code>	Allows to set a callback for the <code>GET_REPORT</code> command.
<code>USBD_HID_SetOnSetReportRequest()</code>	Allows to set a callback for the <code>SET_REPORT</code> control command.
<code>USBD_HID_ReadReport()</code>	Reads report data that was sent from the host via the control EP.
Data structures	
<code>USB_HID_INIT_DATA</code>	Initialization structure that is needed when adding a HID interface to emUSB-Device.
Type definitions	
<code>USB_HID_ON_GETREPORT_REQUEST_FUNC</code>	Callback function description which is set via <code>USBD_HID_SetOnGetReportRequest()</code> .
<code>USB_HID_ON_SETREPORT_REQUEST_FUNC</code>	Callback function description which is set via <code>USBD_HID_SetOnSetReportRequest()</code> .

11.5.2 HID Target API functions

11.5.2.1 USBD_HID_Add()

Description

Adds HID class device to the USB interface.

Prototype

```
USB_HID_HANDLE USBD_HID_Add(const USB_HID_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
<code>pInitData</code>	Pointer to a <code>USB_HID_INIT_DATA</code> structure. For detailed information refer to <code>USB_HID_INIT_DATA</code> .

Return value

`USB_HID_HANDLE`: Handle to the HID instance (can be zero).

Additional information

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

11.5.2.2 USBD_HID_GetNumBytesInBuffer()

Description

Returns the number of bytes available in the internal read buffer.

Prototype

```
unsigned USBD_HID_GetNumBytesInBuffer(USB_HID_HANDLE hInst);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to an HID handle which is returned by <code>USB_HID_Add()</code> .

Return value

≥ 0 Number of bytes in the internal read buffer.

11.5.2.3 USBD_HID_GetNumBytesRemToRead()

Description

Checks how many bytes still have to be read.

Prototype

```
unsigned USBD_HID_GetNumBytesRemToRead(USB_HID_HANDLE hInst);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to an HID instance.

Return value

≥ 0 Number of bytes which have not yet been read.

Additional information

This function is to be used in combination with `USB_HID_ReadOverlapped()`. After starting the read operation this function can be used to periodically check how many bytes still have to be read. Alternatively the blocking function `USB_HID_WaitForRX()` can be used.

11.5.2.4 USBD_HID_GetNumBytesRemToWrite()

Description

Checks how many bytes still have to be written.

Prototype

```
unsigned USBD_HID_GetNumBytesRemToWrite(USB_HID_HANDLE hInst);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to an HID instance.

Return value

≥ 0 Number of bytes which have not yet been written.

Additional information

This function is to be used in combination with a non-blocking call to `USB_HID_Write()`. After starting the write operation this function can be used to periodically check how many bytes still have to be written. Alternatively the blocking function `USB_HID_WaitForTX()` can be used.

11.5.2.5 USBD_HID_Read()

Description

Reads data from the host with a given timeout.

Prototype

```
int USBD_HID_Read(USB_HID_HANDLE  hInst,
                  void              * pData,
                  unsigned          NumBytes,
                  unsigned          Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to an HID instance.
<code>pData</code>	Pointer to a buffer where the received data will be stored.
<code>NumBytes</code>	Number of bytes to read.
<code>Timeout</code>	<code>Timeout</code> given in milliseconds. A zero value results in an infinite timeout.

Return value

= `NumBytes` Requested data was successfully read within the given timeout.
 ≥ 0, < `NumBytes` `Timeout` has occurred. Number of bytes that have been read within the given timeout.
 < 0 Returns a `USB_STATUS_ERROR`.

Additional information

This function blocks until the timeout has been reached, it has received `NumBytes` or until the device is disconnected from the host. This function blocks a task until all data has been read or a timeout occurs. In case of a reset or a disconnect `USB_STATUS_ERROR` is returned.

The host will always send transmissions which match the report size. In most cases it makes sense to set `NumBytes` to the report size.

When using multiple reports the first byte will contain the report ID.

11.5.2.6 USBD_HID_ReadOverlapped()

Description

Reads data from the host asynchronously.

Prototype

```
int USBD_HID_ReadOverlapped(USB_HID_HANDLE hInst,
                             void * pData,
                             unsigned NumBytes);
```

Parameters

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

Return value

- > 0 Number of bytes that have been read from the internal buffer (success).
- = 0 No data was found in the internal buffer (success).
- < 0 Error.

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_HID_WaitForRX()` needs to be called. Alternatively the function `USB_HID_GetNumBytesRemToRead()` can be called periodically to check whether all bytes have been written or not. The buffer pointed to by [pData](#) must be valid until the read operation is terminated.

The host will always send transmissions which match the report size. In most cases it makes sense to set [NumBytes](#) to the report size.

When using multiple reports the first byte will contain the report ID.

11.5.2.7 USBD_HID_WaitForRX()

Description

This function is to be used in combination with `USBD_HID_ReadOverlapped()`. After the read function has been called this function can be used to synchronize. It will block until the transfer is completed.

Prototype

```
int USBD_HID_WaitForRX(USB_HID_HANDLE hInst,  
                       unsigned        Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a HID instance.
<code>Timeout</code>	<code>Timeout</code> given in milliseconds. A zero value results in an infinite timeout.

Return value

- 0 Transfer completed.
- 1 `Timeout` occurred.

Additional information

In case of a timeout, a current transfer is canceled.

11.5.2.8 USBD_HID_WaitForTX()

Description

This function is to be used in combination with a non-blocking call to `USBD_HID_Write()`. After the write function has been called this function can be used to synchronise. It will block until the transfer is completed.

Prototype

```
int USBD_HID_WaitForTX(USB_HID_HANDLE hInst,  
                      unsigned Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a HID instance.
<code>Timeout</code>	<code>Timeout</code> given in milliseconds. A zero value results in an infinite timeout.

Return value

- 0 Transfer completed.
- 1 `Timeout` occurred.

Additional information

In case of a timeout, a current transfer is canceled.

11.5.2.9 USBD_HID_Write()

Description

Writes data to the host. Depending on the `Timeout` parameter, the function may block until `NumBytes` have been written or a timeout occurs.

Prototype

```
int USBD_HID_Write(    USB_HID_HANDLE  hInst,
                      const void      * pData,
                      unsigned        NumBytes,
                      int             Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to an HID instance.
<code>pData</code>	Pointer to data that should be sent to the host.
<code>NumBytes</code>	Number of bytes to write. Should match the report size.
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite. If <code>Timeout</code> is -1, the function returns immediately and the transfer is processed asynchronously.

Return value

= 0 Successful started an asynchronous write transfer or a timeout has occurred and no data was written.

> 0 && < `NumBytes` Number of bytes that have been written before a timeout occurred.

= `NumBytes` Write transfer successful completed.

< 0 Error occurred.

Additional information

This function also returns when the target is disconnected from host or when a USB reset occurred.

The USB stack is able to queue a small number of asynchronous write transfers (`Timeout = -1`). If a write transfer is still in progress when this function is called and the USB stack can not accept another write transfer request, the functions returns `USB_STATUS_EP_BUSY`.

In order to synchronize, `USB_HID_WaitForTX()` needs to be called. Another synchronization method would be to periodically call `USB_HID_GetNumBytesRemToWrite()` in order to see how many bytes still need to be written (this method is preferred when a non-blocking solution is necessary).

The content of the buffer pointed to by `pData` must not be changed until the transfer has been completed.

A transfer which does not match the report size will not be accepted by the host.

When using multiple reports the first byte must contain the report ID.

11.5.2.10 USBD_HID_SetOnGetReportRequest()

Description

Allows to set a callback for the GET_REPORT command. The GET_REPORT command is sent from the host to the device.

Prototype

```
void USBD_HID_SetOnGetReportRequest
(USB_HID_HANDLE hInst,
 USB_HID_ON_GETREPORT_REQUEST_FUNC * pfOnGetReportRequest);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to an HID instance.
<code>pfOnGetReportRequest</code>	Pointer to a function of type <code>USB_HID_ON_GETREPORT_REQUEST_FUNC</code> .

Additional information

See the description of `USB_HID_ON_GETREPORT_REQUEST_FUNC` for more details.

11.5.2.11 USBD_HID_SetOnSetReportRequest()

Description

Allows to set a callback for the SET_REPORT control command. The SET_REPORT command is sent from the host to the device.

Prototype

```
void USBD_HID_SetOnSetReportRequest  
(USB_HID_HANDLE hInst,  
 USB_HID_ON_SETREPORT_REQUEST_FUNC * pfOnSetReportRequest);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to an HID instance.
<code>pfOnSetReportRequest</code>	Pointer to a function of type <code>USB_HID_ON_SETREPORT_REQUEST_FUNC</code> .

Additional information

See the description of `USB_HID_ON_SETREPORT_REQUEST_FUNC` for more details.

11.5.2.12 USBD_HID_ReadReport()

Description

Reads report data that was sent from the host via the control EP. This function returns immediately and will not wait for a report send from the host. Can be used in combination with a callback function installed with `USB_HID_SetOnSetReportRequest()`.

Prototype

```
int USBD_HID_ReadReport(USB_HID_HANDLE hInst,
                       void * pData,
                       unsigned NumBytes);
```

Parameters

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

Return value

> 0 Number of bytes that have been read.
 = 0 No report was sent from the host.
 < 0 Error.

Additional information

The host will always send transmissions which match the report size. In most cases it makes sense to set `NumBytes` to the report size.

When using multiple reports the first byte will contain the report ID.

11.5.3 Data structures

11.5.3.1 USB_HID_INIT_DATA

Description

Initialization structure that is needed when adding a HID interface to emUSB-Device.

Type definition

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

Structure members

Member	Description
<code>EPIn</code>	Endpoint for sending data to the host.
<code>EPOut</code>	Endpoint for receiving data from the host.
<code>pReport</code>	Pointer to a report descriptor.
<code>NumBytesReport</code>	Size of the HID report descriptor in bytes.
<code>BuffSize</code>	Size of the buffer pointed to by <code>pBuff</code> . Must be at least the size of the output report.
<code>pBuff</code>	Pointer to a buffer for receiving reports from the host via endpoint 0 (<code>Set_Report</code> request).

Additional information

To be able to receive input reports from the host either an endpoint must be allocated (`EPOut`) or a buffer must be provided (`BufferSize`, `pBuff`). If both `EPOut = 0` and `BufferSize = 0`, then `USBD_HID_Read()` will not work and all requests from the host will be stalled by the USB stack. To receive Set Feature Report control commands the buffer is 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 and 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 for detailed information about 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: 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).

Example 1 (configure to receive reports via seperate endpoint)

```
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);
    InitData.EPOut = USB_AddEP(USB_DIR_OUT, USB_TRANSFER_TYPE_INT, Interval,
```

```
        &acBuffer[0], sizeof(acBuffer));
InitData.pReport = _aHIDReport;
InitData.NumBytesReport = sizeof(_aHIDReport);
USBD_HID_Add(&InitData);
}
```

Example 2 (configure to receive reports via endpoint 0)

```
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);
    InitData.pBuff = &acBuffer[0];
    InitData.BufferSize = sizeof(acBuffer);
    InitData.pReport = _aHIDReport;
    InitData.NumBytesReport = sizeof(_aHIDReport);
    USBD_HID_Add(&InitData);
}
```

11.5.4 Type definitions

11.5.4.1 USB_HID_ON_GETREPORT_REQUEST_FUNC

Description

Callback function description which is set via `USBD_HID_SetOnGetReportRequest()`.

Type definition

```
typedef int USB_HID_ON_GETREPORT_REQUEST_FUNC(
    USB_HID_REPORT_TYPE ReportType,
    unsigned             ReportId,
    const U8             * * pData,
    U32                  * pNumBytes);
```

Parameters

Parameter	Description
<code>ReportType</code>	HID report type, possible values are: <ul style="list-style-type: none"> • <code>USB_HID_REPORT_TYPE_INPUT</code> • <code>USB_HID_REPORT_TYPE_OUTPUT</code> • <code>USB_HID_REPORT_TYPE_FEATURE</code>
<code>ReportId</code>	The ID of the report for which the <code>GET_REPORT</code> request has been sent.
<code>pData</code>	in Pointer to a pointer to the data to send via <code>GET_REPORT</code> request.
<code>pNumBytes</code>	IN: Number of bytes requested. Out: Number of bytes that shall be sent.

Return value

- = 0 No data available. The stack will send a zero length packet as a response.
- = 1 Data is available. The stack will send data to the host.
- < 0 Data is handled by user application. `USBD_WriteEP0FromISR()` needs to be called from user context.

11.5.4.2 USB_HID_ON_SETREPORT_REQUEST_FUNC

Description

Callback function description which is set via `USBD_HID_SetOnSetReportRequest()`. The function is called after a `SET_REPORT` command was sent from the host via the control endpoint. The report should be read using `USBD_HID_ReadReport()`.

Type definition

```
typedef void USB_HID_ON_SETREPORT_REQUEST_FUNC(USB_HID_REPORT_TYPE ReportType,
                                               unsigned          ReportId,
                                               U32              NumBytes);
```

Parameters

Parameter	Description
<code>ReportType</code>	HID report type, possible values are: <ul style="list-style-type: none"> • <code>USB_HID_REPORT_TYPE_INPUT</code> • <code>USB_HID_REPORT_TYPE_OUTPUT</code> • <code>USB_HID_REPORT_TYPE_FEATURE</code>
<code>ReportId</code>	The ID of the report for which the <code>SET_REPORT</code> request has been sent.

Additional information

In case no EP Out was used with the HID interface, and a `USBD_HID_Read()` or `USBD_HID_ReadOverlapped()` is currently executed, then this function is not called and the read function is serviced instead.

11.6 Host API

This chapter describes the functions that can be used with host side (Windows, Linux, macOS). These functions are only required if the emUSB-Device-HID component is used 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 host HID 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.

11.6.1 Host API function list

Function	Description
API functions	
<code>USBHID_Close()</code>	Close the connection to an open device.
<code>USBHID_Open()</code>	Opens a handle to a device.
<code>USBHID_Init()</code>	Sets the specific vendor page, initializes the USB HID User API and retrieves the information of the HID device.
<code>USBHID_Exit()</code>	Closes the connection to all open devices and de-initializes the HID module.
<code>USBHID_Read()</code>	Reads an input report from device via the interrupt endpoint.
<code>USBHID_Write()</code>	Writes an output report to device.
<code>USBHID_GetNumAvailableDevices()</code>	Returns the number of available devices.
<code>USBHID_GetProductName()</code>	Stores the name of the device into <code>pBuffer</code> .
<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 USB product ID of the device.
<code>USBHID_GetVendorId()</code>	Returns the USB vendor ID of the device.
<code>USBHID_RefreshList()</code>	Refreshes the connection info list.
<code>USBHID_SetVendorPage()</code>	Sets the vendor page so that all HID devices with the specified page will be found.

11.6.2 HID Host API functions

11.6.2.1 USBHID_Close()

Description

Close the connection to an open device.

Prototype

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

Parameters

Parameter	Description
Id	Index of the HID device. This is the bit number of the mask returned by USBHID_GetNumAvailableDevices().

11.6.2.2 USBHID_Open()

Description

Opens a handle to a device.

Prototype

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

Parameters

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

Return value

- 0 O.K. Opening was successful or already opened.
- 1 Error. Handle to the device could not opened.

11.6.2.3 USBHID_Init()

Description

Sets the specific vendor page, initializes the USB HID User API and retrieves the information of the HID device.

Prototype

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

Parameters

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.

11.6.2.4 USBHID_Exit()

Description

Closes the connection to all open devices and de-initializes the HID module.

Prototype

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

11.6.2.5 USBHID_Read()

Description

Reads an input report from device via the interrupt endpoint.

Prototype

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

Return value

On Error: -1, No valid device Id used or the report size does not match with device. On success: Number of bytes that have be written.

11.6.2.6 USBHID_Write()

Description

Writes an output report to device.

Prototype

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

Return value

On Error: -1, No valid device Id used or the report size does not match with device. On success: Number of bytes that have been written.

11.6.2.7 USBHID_GetNumAvailableDevices()

Description

Returns the number of available devices.

Prototype

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

Parameters

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 NULL.

Return value

Number of available devices.

Additional information

`pMask` will be filled by this routine. It shall be interpreted as a 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`.

11.6.2.8 USBHID_GetProductName()

Description

Stores the name of the device into `pBuffer`.

Prototype

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

Parameters

Parameter	Description
<code>Id</code>	Index of the HID device. This is the bit number of the mask returned by <code>USBHID_GetNumAvailableDevices()</code> .
<code>pBuffer</code>	Pointer to a buffer for the product name.
<code>NumBytes</code>	Size of the buffer in bytes.

Return value

- 0 An error occurred.
- 1 Success.

11.6.2.9 USBHID_GetInputReportSize()

Description

Returns the input report size of the device.

Prototype

```
int USBHID_GetInputReportSize(unsigned Id);
```

Parameters

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

Return value

- = 0 An error occurred.
- ≠ 0 Size of the report in bytes.

11.6.2.10 USBHID_GetOutputReportSize()

Description

Returns the output report size of the device.

Prototype

```
int USBHID_GetOutputReportSize(unsigned Id);
```

Parameters

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

Return value

- = 0 An error occurred.
- ≠ 0 Size of the report in bytes.

11.6.2.11 USBHID_GetProductId()

Description

Returns the USB product ID of the device.

Prototype

```
U16 USBHID_GetProductId(unsigned Id);
```

Parameters

Parameter	Description
Id	Index of the HID device. This is the bit number of the mask returned by USBHID_GetNumAvailableDevices().

Return value

= 0 An error occurred.
≠ 0 Product ID.

11.6.2.12 USBHID_GetVendorId()

Description

Returns the USB vendor ID of the device.

Prototype

```
U16 USBHID_GetVendorId(unsigned Id);
```

Parameters

Parameter	Description
Id	Index of the HID device. This is the bit number of the mask returned by USBHID_GetNumAvailableDevices().

Return value

= 0 An error occurred.
≠ 0 Vendor ID.

11.6.2.13 USBHID_RefreshList()

Description

Refreshes the connection info list.

Prototype

```
void USBHID_RefreshList(void);
```

Additional information

Note that any open handles will be closed while refreshing the connection list.

11.6.2.14 USBHID_SetVendorPage()

Description

Sets the vendor page so that all HID devices with the specified page will be found.

Prototype

```
void USBHID_SetVendorPage(U8 Page);
```

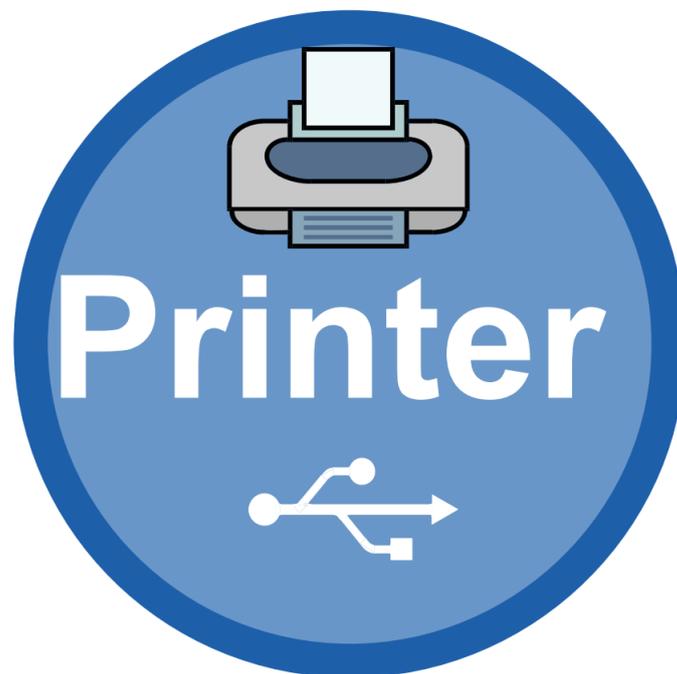
Parameters

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

Chapter 12

Printer Class

This chapter describes how to get emUSB-Device up and running as a printer device.



12.1 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.

12.1.1 Configuration

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

12.2 The example application

The start application (in the `Application` subfolder) is a simple data sink, which can be used to test emUSB-Device. The application receives data bytes from the host which it displays in the terminal I/O window of the debugger.

Part of source code of `USB_Printer.c`:

```
<...>
/*****
 *
 *     _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
*
*****/
static const USB_DEVICE_INFO _DeviceInfo = {
    0x8765,           // VendorId
    0x2114,           // ProductId, should be unique for this sample
    "Vendor",        // VendorName
    "Printer",       // ProductName
    "12345678901234567890" // SerialNumber
};
/*****
*
*   MainTask
*
* Function description
*   USB handling task.
*   Modify to implement the desired protocol
*/
void MainTask(void) {
    USBD_Init();
    USBD_SetDeviceInfo(&_DeviceInfo);
    USB_PRINTER_Init(&_PrinterAPI);
    USBD_Start();
    while (1) {
        //
        // Wait for configuration
        //
        while ((USB_D_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED))
            != USB_STAT_CONFIGURED)
        {
            BSP_ToggleLED(0);
            USB_OS_Delay(50);
        }
        //
        // Receive and process data.
        //
        USB_PRINTER_Task();
    }
}

```

12.3 Target API

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

12.3.1 Interface function list

Function	Description
API functions	
<code>USB_PRINTER_Init()</code>	Initializes the printer module.
<code>USB_PRINTER_Task()</code>	Processes the requests received from the USB Host.
<code>USB_PRINTER_TaskEx()</code>	Processes the requests received from the USB Host.
<code>USB_PRINTER_ConfigIRQProcessing()</code>	Configure printer class to process received data in USB interrupt.
<code>USB_PRINTER_Read()</code>	Reads data from the host.
<code>USB_PRINTER_ReadTimed()</code>	Reads data from the host with a given timeout.
<code>USB_PRINTER_Receive()</code>	Reads data from host.
<code>USB_PRINTER_ReceiveTimed()</code>	Reads data from host with a given timeout.
<code>USB_PRINTER_Write()</code>	Writes data to the host.
<code>USB_PRINTER_WriteTimed()</code>	Writes data to the host within a given timeout.
<code>USB_PRINTER_SetOnVendorRequest()</code>	Sets a callback function that is called when a setup vendor request is sent from the host to the printer.
<code>USB_PRINTER_SetClass()</code>	Sets a custom class/subclass/protocol for the printer class.
Data structures	
<code>USB_PRINTER_API</code>	Initialization structure that is needed when adding a printer interface to emUSB-Device.

12.3.2 API functions

12.3.2.1 USB_PRINTER_Init()

Description

Initializes the printer module.

Prototype

```
void USB_PRINTER_Init(USB_PRINTER_API * pAPI);
```

Parameters

Parameter	Description
<code>pAPI</code>	Pointer to an API table that contains all callback functions that are necessary for handling the functionality of a printer.

Additional information

After the initialization of general emUSB-Device, this is the first function that needs to be called when the printer class is used with emUSB-Device.

12.3.2.2 USB_PRINTER_Task()

Description

Processes the requests 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 functions registered in the call to `USB_PRINTER_Init()`. Do not call this function if you used `USB_PRINTER_ConfigIRQProcessing()`.

12.3.2.3 USB_PRINTER_TaskEx()

Description

Processes the requests received from the USB Host. Uses overlapped read operation for higher performance.

Prototype

```
void USB_PRINTER_TaskEx(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()`.

12.3.2.4 USB_PRINTER_ConfigIRQProcessing()

Description

Configure printer class to process received data in USB interrupt. Must be called after `USB_PRINTER_Init()` and before `USBD_Start()`. After calling this function, `USB_PRINTER_Task()` should never be called.

Prototype

```
void USB_PRINTER_ConfigIRQProcessing(void);
```

Additional information

The printer API function `USB_PRINTER_API -> pfOnDataReceived` is called within the USB interrupt context and must not block.

12.3.2.5 USB_PRINTER_Read()

Description

Reads data from the host.

Prototype

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

Parameters

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

Return value

= `NumBytes` Success.
< `NumBytes` Error occurred.

Additional information

This function blocks a task until all data has been read. In case of a reset or a disconnect `USB_STATUS_ERROR` is returned.

12.3.2.6 USB_PRINTER_ReadTimed()

Description

Reads data from the host with a given timeout.

Prototype

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

Parameters

Parameter	Description
<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>	Timeout in milliseconds. A zero value results in an infinite timeout.

Return value

= `NumBytes` Success.
≥ 0, < `NumBytes` Number of bytes that have been read within the given timeout.
< 0 Error.

Additional information

This function blocks a task until all data has been read or a timeout occurs. In case of a reset or a disconnect `USB_STATUS_ERROR` is returned.

12.3.2.7 USB_PRINTER_Receive()

Description

Reads data from host. The function blocks until any data has been received. In contrast to `USB_PRINTER_Read()` this function does not wait for all of `NumBytes` to be received, but returns after the first packet has been received.

Prototype

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

Parameters

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

Return value

- > 0 Number of bytes that have been read.
- = 0 Zero packet received (not every controller supports this!) or the target was disconnected during the function call.
- < 0 Error.

Additional information

If no error occurs, this function returns the number of bytes received. Calling `USB_PRINTER_Receive()` will return as much data as is currently available up to the size of the buffer specified. This function also returns when target is disconnected from host or when a USB reset occurred, it will then return the number of bytes read.

12.3.2.8 USB_PRINTER_ReceiveTimed()

Description

Reads data from host with a given timeout. The function blocks until any data has been received. In contrast to `USB_PRINTER_ReadTimed()` this function does not wait for all of `NumBytes` to be received, but returns after the first packet has been received or after the timeout has been reached.

Prototype

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

Parameters

Parameter	Description
<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>	Timeout in milliseconds.

Return value

- > 0 Number of bytes that have been read within the given timeout.
- = 0 Zero packet received (not every controller supports this!) or the target was disconnected during the function call.
- < 0 An error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling `USB_PRINTER_ReceiveTimed()` will return as much data as is currently available up to the size of the buffer specified within the specified timeout. This function also returns when target is disconnected from host or when a USB reset occurred, it will then return the number of bytes read.

12.3.2.9 USB_PRINTER_Write()

Description

Writes data to the host.

Prototype

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

Parameters

Parameter	Description
<code>pData</code>	Pointer to a buffer that contains the data to be sent.
<code>NumBytes</code>	Number of bytes to write.

Return value

≥ 0 Number of bytes that have been written.
< 0 Error.

Additional information

This function is blocking.

12.3.2.10 USB_PRINTER_WriteTimed()

Description

Writes data to the host within a given timeout.

Prototype

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

Parameters

Parameter	Description
<code>pData</code>	Pointer to a buffer that contains the data to be sent.
<code>NumBytes</code>	Number of bytes to write.
<code>ms</code>	Timeout in milliseconds. A zero value results in an infinite timeout. If <code>ms</code> is < 0 , the function does not block and may return <code>USB_STATUS_EP_BUSY</code> .

Return value

- > 0 Number of bytes that have been written before timeout.
- $= 0$ Timeout occurred.
- < 0 Error.

Additional information

If `ms` ≥ 0 , this function blocks the task until all data has been written or a timeout occurred. In case of a reset or a disconnect `USB_STATUS_ERROR` is returned.

12.3.2.11 USB_PRINTER_SetOnVendorRequest()

Description

Sets a callback function that is called when a setup vendor request is sent from the host to the printer. The callback must return "0" if it handled the request and "1" if it did not.

Prototype

```
void USB_PRINTER_SetOnVendorRequest(USB_ON_CLASS_REQUEST * pfOnVendorRequest);
```

Parameters

Parameter	Description
pfOnVendorRequest	Pointer to the callback function.

Additional information

Note that the callback will be called within an ISR, therefore it should never block. If it is necessary to send data from the callback function through endpoint 0, use the function `USBD_WriteEP0FromISR()`.

`USB_ON_CLASS_REQUEST` is defined as follows:

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

12.3.2.12 USB_PRINTER_SetClass()

Description

Sets a custom class/subclass/protocol for the printer class. Can be used to interface with proprietary manufacturer drivers.

Prototype

```
void USB_PRINTER_SetClass(U8 Class,  
                          U8 SubClass,  
                          U8 Protocol);
```

Parameters

Parameter	Description
Class	USB class ID overwrite (printer class default is 0x07)
SubClass	USB sub-class ID overwrite (printer class default is 0x01)
Protocol	USB protocol ID overwrite (emUSB-Device printer class default is 0x02)

Additional information

This function must be called after `USB_PRINTER_Init()` and before `USBD_Start()`.

12.3.2.13 USB_PRINTER_API

Description

Initialization structure that is needed when adding a printer interface to emUSB-Device. It holds pointers to callback functions the interface invokes when it processes a request from the USB host.

Type definition

```
typedef struct {
    USB_PRINTER_GET_DEVICE_ID_STRING * pfGetDeviceIdString;
    USB_PRINTER_ON_DATA_RECEIVED * pfOnDataReceived;
    USB_PRINTER_GET_HAS_NO_ERROR * pfGetHasNoError;
    USB_PRINTER_GET_IS_SELECTED * pfGetIsSelected;
    USB_PRINTER_GET_IS_PAPER_EMPTY * pfGetIsPaperEmpty;
    USB_PRINTER_ON_RESET * pfOnReset;
} USB_PRINTER_API;
```

Structure members

Member	Description
pfGetDeviceIdString	The library calls this function when the USB host requests the printer's identification string.
pfOnDataReceived	This function is called when data arrives from the USB host.
pfGetHasNoError	This function should return a non-zero value if the printer has no error.
pfGetIsSelected	This function should return a non-zero value if the printer is selected
pfGetIsPaperEmpty	This function should return a non-zero value if the printer is out of paper.
pfOnReset	The library calls this function if the USB host sends a soft reset command.

Additional information

Detailed information can be found in *USB_PRINTER_API in detail* on page 439.

12.4 Printer API

This section describes the emUSB-Device Printer API in detail.

12.4.1 General information

The interface includes multiple callback functions which have to be set by the user application. These functions are called by the emUSB-Device stack when the host makes the corresponding enquiries.

12.4.2 USB_PRINTER_API in detail

12.4.2.1 USB_PRINTER_GET_DEVICE_ID_STRING

Description

The library calls this function when the USB host requests the printer's identification string. This string shall conform to the IEEE 1284 Device ID Syntax.

Type definition

```
typedef const char * USB_PRINTER_GET_DEVICE_ID_STRING(void);
```

Return value

Pointer to the ID string.

Additional information

The return string shall conform to the IEEE 1284 Device ID.

Example

```
"CLASS:PRINTER;  
MODEL:HP LaserJet 6MP;  
MANUFACTURER:Hewlett-Packard;  
DESCRIPTION:Hewlett-Packard LaserJet 6MP Printer;  
COMMAND SET:PJL,MLC,PCLXL,PCL,POSTSCRIPT;"
```

12.4.2.2 USB_PRINTER_ON_DATA_RECEIVED

Description

This function is called when data arrives from USB host.

Type definition

```
typedef int USB_PRINTER_ON_DATA_RECEIVED(const U8 * pData,  
                                         unsigned NumBytes);
```

Parameters

Parameter	Description
<code>pData</code>	Pointer to the data.
<code>NumBytes</code>	Data length.

Return value

- = 0 More data can be accepted
- ≠ 0 No more data can be accepted, in this case a stall will be sent back to the host.

12.4.2.3 USB_PRINTER_GET_HAS_NO_ERROR

Description

This function should return a non-zero value if the printer has no error.

Type definition

```
typedef U8 USB_PRINTER_GET_HAS_NO_ERROR(void);
```

Return value

= 0	Error condition present.
≠ 0	No error.

12.4.2.4 USB_PRINTER_GET_IS_SELECTED

Description

This function should return a non-zero value if the printer is selected.

Type definition

```
typedef U8 USB_PRINTER_GET_IS_SELECTED(void);
```

Return value

= 0	Not selected.
≠ 0	Selected.

12.4.2.5 USB_PRINTER_GET_IS_PAPER_EMPTY

Description

This function should return a non-zero value if the printer is out of paper.

Type definition

```
typedef U8 USB_PRINTER_GET_IS_PAPER_EMPTY(void);
```

Return value

= 0 Has paper.
≠ 0 Out of paper.

12.4.2.6 USB_PRINTER_ON_RESET

Description

The library calls this function if the USB host sends a soft reset command.

Type definition

```
typedef void USB_PRINTER_ON_RESET(void);
```

Chapter 13

IP-over-USB (IP)

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



13.1 Overview

The IP component is a very convenient package when you need to use IP-based protocols over USB with different host operating systems. It consists of two different components - RNDIS and CDC-ECM. Combined with the smart capabilities of emUSB-Device-IP to form a cross-platform USB to Ethernet device that works on every common Host OS that can handle USB devices.

13.2 Using only RNDIS or CDC-ECM

Main problem between different Host OSES is that either one IP-over-USB class is supported which is then not supported on the other Host OS.

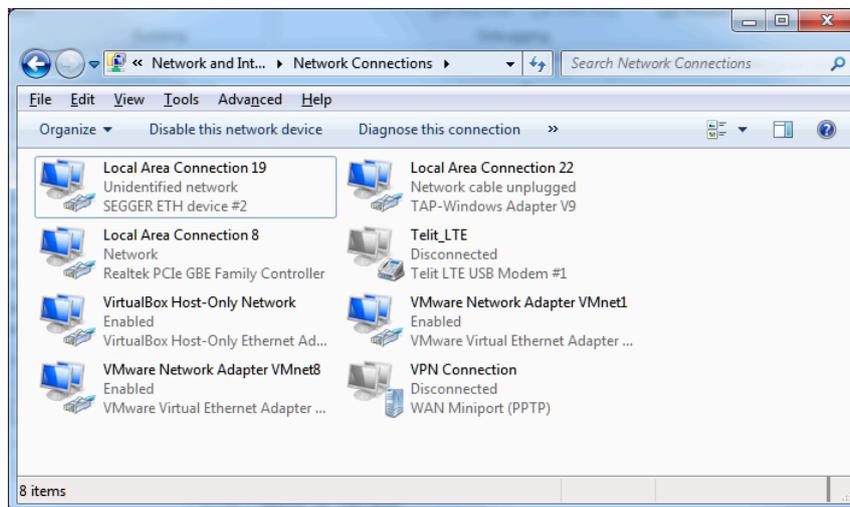
Host OS/Protocol	RNDIS	CDC-ECM
Windows	X	-
Linux	X	X
macOS	-	X
Free/Net/OpenBSD	X	X

Linux and all BSD distribution work with any IP-over-USB interface and therefore can be used with either RNDIS or CDC-ECM. macOS does not support RNDIS, third party tools may work but are not fully compatible and using a new version of macOS the driver or package may no longer work properly. Windows cannot handle CDC-ECM out-of-the-box. There are third-party drivers which can handle this but the driver package has to be licensed. Furthermore a new inf-file needs to be written for your device and as a consequence of that the driver package itself needs to be certified which involves further costs. Adding new CDC-ECM devices to the inf-file forces to resign that package once again.

SEGGER's IP-over-USB solution eliminates these limitations.

13.2.1 Working with emUSB-Device-IP

A IP-over-USB device connected to a PC running the Windows operating system is listed as a separate network interface in the "Network Connections" window as shown in this screenshot:



The `ping` command line utility can be used to test the connection to target as shown below. If the connection is correctly established the number of the lost packets should be 0.

```

Administrator: Command Prompt
C:\>ping segger.veth

Pinging segger.veth [fe80::622:c7ff:fe80::46] with 32 bytes of data:
Reply from fe80::622:c7ff:fe80::46: time<1ms
Reply from fe80::622:c7ff:fe80::46: time<1ms
Reply from fe80::622:c7ff:fe80::46: time<1ms
Reply from fe80::622:c7ff:fe80::46: time<1ms

Ping statistics for fe80::622:c7ff:fe80::46:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\>ping 100.127.8.1

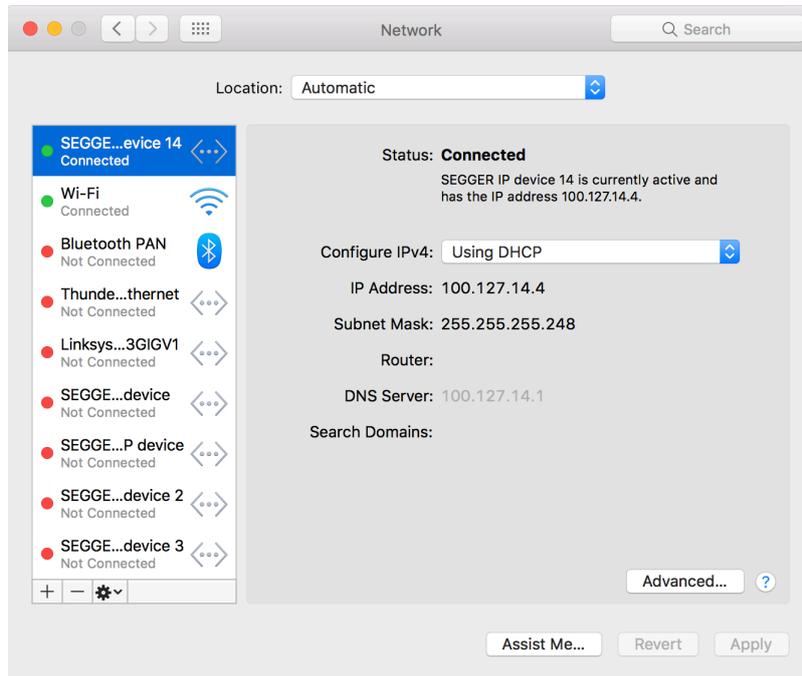
Pinging 100.127.8.1 with 32 bytes of data:
Reply from 100.127.8.1: bytes=32 time<1ms TTL=64
Reply from 100.127.8.1: bytes=32 time<1ms TTL=64
Reply from 100.127.8.1: bytes=32 time<1ms TTL=64
Reply from 100.127.8.1: bytes=32 time<1ms TTL=64

Ping statistics for 100.127.8.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\>_

```

On macOS IP-over-USB is similarly available:



And on Ubuntu:

```

anybody@ubuntu: ~
anybody@ubuntu:~$ ifconfig eth1
eth1      Link encap:Ethernet  HWaddr 04:22:c7:fe:00:9f
          inet addr:100.127.8.4  Bcast:100.127.8.7  Mask:255.255.255.248
          inet6 addr: fe80::3736:1b9:47c8:3d69/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1486  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:65 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1028 (1.0 KB)  TX bytes:8500 (8.5 KB)

anybody@ubuntu:~$

```

13.3 Configuration

13.3.1 Initial Configuration

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

13.3.2 Final configuration

The configuration must only be modified when emUSB-Device-IP is used in your final product. Refer to section *emUSB-Device Configuration* on page 48 to get detailed information about the general emUSB-Device configuration functions which have to be adapted.

Note

Due to an issue in Windows when using IP-over-USB within a multi-interface device the IP-over-USB interface must be added first. Otherwise it will not be recognized.

Note

Due to an issue with Windows 7 USB 3.0 drivers `USBD_EnableIAD()` must be used, even if the device containing the IP-over-USB interface is not a multi-interface device. Otherwise the device will not be recognized on USB 3.0 ports of a PC running Windows 7.

13.3.3 Class specific configuration

emUSB-Device-IP specific device information must be provided by the application via the function `USBD_IP_Add()`. A sample how to use this function can be found in the `IP_Config_IP_over_USB.c`. The file is located in the `Sample\IP` directory of the emUSB-Device shipment. The `IP_Config_IP_over_USB.c` provides a ready to use layer and configuration file to be used with *embOS* and *emNet*.

13.4 Running the sample application

The sample application can be found in the `Sample\IP\IP_Config_IP_over_USB.c` file of the emUSB-Device shipment. In order to use the sample application the SEGGER *emNet* middleware component is required. To test the emUSB-Device-IP component any of the *emNet* sample applications can be used in combination with `IP_Config_IP_over_USB.c`. After the sample application is started the USB cable should be connected to the PC and the chosen *emNet* sample can be tested by using the URL "usb.local".

13.4.1 IP_Config_IP_over_USB.c in detail

The main part of the sample application is implemented in the function `MainTask()` which runs as an independent task.

```
// IP_X_Config() - excerpt from IP_Config_IP_over_USB.c
void IP_X_Config(void) {
<...>
IP_SetIFaceConnectHook(IFaceId, _Connect);
IP_AddStateChangeHook(&_Hook, _OnHWChange)
<...>
}

// _Connect() - excerpt from IP_Config_IP_over_USB.c
static int _Connect(unsigned IFaceId) {
//
// Initialize the DHCP Server for this interface.
//
IP_DHCPS_Init(IFaceId);
//
// Initialize the USB stack
// and prepare to use the IP-over-USB connection
//
USBD_Init();
USBD_SetDeviceInfo(&USB_DeviceInfo);
//
// Enable IAD makes sure that the device is properly enumerated on Windows
// 7 on USB 3 Controller
//
USBD_EnableIAD();
_AddIP();
USBD_Start();
return 0; // Successfully connected.
}

// _AddIP() - excerpt from IP_Config_IP_over_USB.c
static void _AddIP(void) {
USB_IP_INIT_DATA InitData;

IP_MEMSET(&InitData, 0, sizeof(InitData));
InitData.EPOut
= USBD_AddeP(USB_DIR_OUT, USB_TRANSFER_TYPE_BULK, 0, _abReceiveBuffer, sizeof(_abReceiveBuffer));
InitData.EPIn = USBD_AddeP(USB_DIR_IN,
USB_TRANSFER_TYPE_BULK, 0, NULL, 0);
InitData.EPInt = USBD_AddeP(USB_DIR_IN, USB_TRANSFER_TYPE_INT,
5, NULL, 0);
InitData.pDriverAPI = &USB_Driver_IP_NI;
InitData.DriverData.pDriverData = (void *)_IFaceId;
InitData.pRnDisDevInfo = &_RNDIS_DeviceInfo;
USBD_IP_Add(&InitData);
}

// _OnHWChange() - excerpt from IP_Config_IP_over_USB.c
static void _OnHWChange(unsigned IFaceId, U8 AdminState, U8 HWState) {
unsigned IsReady;

IP_USE_PARA(AdminState);
```

```

    IsReady = HWState ? 1 : 0;
    _UpdateIPServiceOperation(IFaceId, IsReady);
}

// _UpdateIPServiceOperation() - excerpt from IP_Config_IP_over_USB.c
static void _UpdateIPServiceOperation(unsigned IFaceId, unsigned LinkStatus) {
    U32 ServerIpAddr;
    U32 SubnetMask;
    U8  USBAddr;

    if (IFaceId == _IFaceId) {
        if (LinkStatus) {
            //

            // As soon as we are enumerated, we know that the USB host has assigned a USB address to us.

            // We will use this USB address in order to assign the emNET interface a IP address.
            // The IP address is selected from a so-called IP pool for CGN
            (carrier grade NAT) - RFC6598
            // The IP pool for this is 100.64.0.0/10.
            // We will use 100.127.<USBAddr>.0/29 -> gives us 8 IP addresses, whereas
            6 are assignable.
            // This is enough for a client/host configuration.
            // Therefore we will have the following scenario
            // emNET interface: IP: 100.127.<USBAddr>.1
            // Host interface: 100.127.<USBAddr>.2 - assigned by our DHCP server.
            // SubnetMask = 255.255.255.248
            //
            USBAddr = USBD_GetUSBAddr();
            ServerIpAddr = IP_BYTES2ADDR(100, 127, USBAddr, 1);
            SubnetMask = IP_BYTES2ADDR(0xff, 0xff, 0xff, 0xf8);
            //
            // Configure the delayed exec parameters
            //
            //
            // IP_SetAddrMaskEx parameters
            //
            _SetAddrMaskExPara.Para0 = SEGGER_PTR2ADDR(IFaceId);
            _SetAddrMaskExPara.Para1 = SEGGER_PTR2ADDR(ServerIpAddr);
            _SetAddrMaskExPara.Para2 = SEGGER_PTR2ADDR(SubnetMask);
            //
            // IP_DHCPS_ConfigPool parameters
            //
            _DNSServer = ServerIpAddr;
            _DHCPS_ConfigPoolPara.Para0 = SEGGER_PTR2ADDR(IFaceId);
            _DHCPS_ConfigPoolPara.Para1 = SEGGER_PTR2ADDR(ServerIpAddr + 1);
            _DHCPS_ConfigPoolPara.Para2 = SEGGER_PTR2ADDR(SubnetMask);
            _DHCPS_ConfigPoolPara.Para3 = SEGGER_PTR2ADDR(4);
            //
            // IP_DHCPS_ConfigDNS parameters
            //
            _DHCPS_ConfigDNSAddPara.Para0 = SEGGER_PTR2ADDR(IFaceId);
            _DHCPS_ConfigDNSAddPara.Para1 = SEGGER_PTR2ADDR(&_DNSServer);
            _DHCPS_ConfigDNSAddPara.Para2 = SEGGER_PTR2ADDR(1);
            //
            // IP_DHCPS_Start(m)DNS parameters
            //
            IP_MEMSET(&_DNSConfig, 0, sizeof(_DNSConfig));
            _DNSConfig.sHostname = SERVER_NAME;
            _DNSConfig.TTL = 60;
            _DNSConfig.apSDConfig = _aSDConfig;
            _DNSConfig.NumConfig = SEGGER_COUNTOF(_aSDConfig);
            //
            // Queue now the the delayed exec commands
            //
            // Assign a local address 100.127.<USBAddr>.1/8
            IP_ExecDelayed(&_aIPExecDelayed[IP_SET_ADDRESS_MASK_OPERATION],
                IP_SetAddrMaskEx_Delayed,

```

```

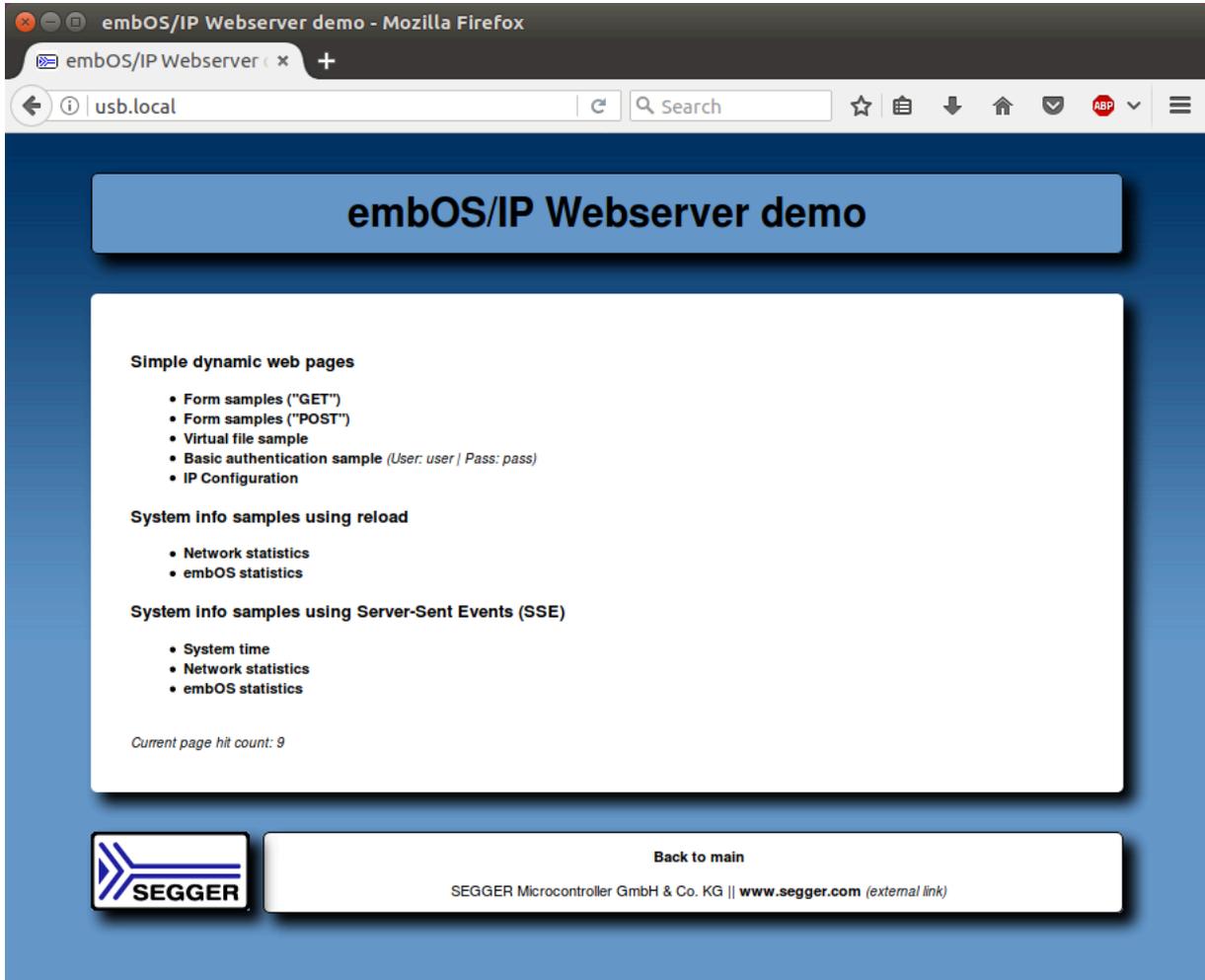
        &_SetAddrMaskExPara,
        (void *)"SetAddrMaskEx",
        _OnDone);
IP_ExecDelayed(&_aIPExecDelayed[IP_DHCP_CONFIGPOOL_OPERATION],
               IP_DHCP_ConfigPool_Delayed,
               &_DHCP_ConfigPoolPara,
               (void *)"IP_DHCP_ConfigPool",
               _OnDone);
IP_ExecDelayed(&_aIPExecDelayed[IP_DHCP_CONFIGDNSADDR_OPERATION],
               IP_DHCP_ConfigDNSAddr_Delayed,
               &_DHCP_ConfigDNSAddPara,
               (void *)"IP_DHCP_ConfigDNSAddr",
               _OnDone);
IP_ExecDelayed(&_aIPExecDelayed[IP_DHCP_START_OPERATION],
               IP_DHCP_Start_Delayed,
               SEGGER_ADDR2PTR(void, _IFaceId),
               (void *)"IP_DHCP_Start",
               _OnDone);
IP_ExecDelayed(&_aIPExecDelayed[IP_MDNS_SERVER_START_OPERATION],
               IP_MDNS_SERVER_Start_Delayed,
               &_DNSConfig,
               (void *)"IP_MDNS_SERVER_Start",
               _OnDone);
IP_ExecDelayed(&_aIPExecDelayed[IP_DNS_SERVER_START_OPERATION],
               IP_DNS_SERVER_Start_Delayed,
               &_DNSConfig,
               (void *)"IP_DNS_SERVER_Start",
               _OnDone);
}
if (LinkStatus == 0) {
    IP_ExecDelayed(&_aIPExecDelayed[IP_DHCP_HALT_OPERATION],
                  IP_DHCP_Halt_Delayed,
                  SEGGER_ADDR2PTR(void, _IFaceId),
                  (void *)"IP_DHCP_Halt",
                  _OnDone);
    IP_ExecDelayed(&_aIPExecDelayed[IP_MDNS_SERVER_STOP_OPERATION],
                  IP_MDNS_SERVER_Stop_Delayed,
                  NULL,
                  (void *)"IP_MDNS_SERVER_Stop",
                  _OnDone);
    IP_ExecDelayed(&_aIPExecDelayed[IP_DNS_SERVER_STOP_OPERATION],
                  IP_DNS_SERVER_Stop_Delayed,
                  NULL,
                  (void *) "IP_DNS_SERVER_Stop",
                  _OnDone);
}
}
}
}

```

The first step is to initialize the DHCP server component which assigns the IP address for the PC side. The target is configured with the IP address 100.127.<USBAddr>.1. The USB address is assigned by the host PC and is unique in the host system. This makes sure that when multiple devices using emUSB-Device-IP are connected they receive a different subnet. The DHCP server is configured to distribute IP addresses starting from 100.127.<USBAddr>.2, therefore the PC will receive the IP address 100.127.<USBAddr>.2. DNS and mDNS is enabled to allow name resolution.

13.5 emUSB-Device-IP + emNet as a "USB Webserver"

This method of using emUSB-Device-IP provides a unique customer experience where a USB device can provide a custom web page or any other service through which a customer can interact with the device.



Initially the PC recognizes an RNDIS device. In case of Windows XP and Vista a driver will be necessary (the corresponding inf-file can be found in the `Windows\USB\RNDIS\WinXP_Vista` folder), Windows 7 and above as well as Linux recognize RNDIS automatically. RNDIS from the viewpoint of the PC is a normal Network Interface Controller (NIC) and the PC handles it as such. The default behaviour is to request an IP address from a DHCP server. The PC retrieves an IP address from the DHCP-Server in the device. In our standard sample code the device has the local IP `100.127.<USBAddr>.1` and the PC will get `100.127.<USBAddr>.2` from the DHCP server. With this the configuration is complete and the user can access the web-interface located on the USB device via the DNS entry - "usb.local".

13.6 Target API

Function	Description
API functions	
<code>USBD_IP_Add()</code>	Adds support for the IP component to USB stack.
<code>USBD_IP_Task()</code>	Obsolete.
Data structures	
<code>USB_IP_INIT_DATA</code>	Structure which stores the parameters of the IP component.

13.6.1 API functions

13.6.1.1 USBD_IP_Add()

Description

Adds support for the IP component to USB stack. Internally CDC-ECM and RNDIS is initialized. The IP component switches automatically between the two.

Prototype

```
void USBD_IP_Add(const USB_IP_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
<code>pInitData</code>	Pointer to a filled <code>USB_IP_INIT_DATA</code> structure data.

13.6.1.2 USBD_IP_Task()

Description

Obsolete. Returns when USB is disconnected.

Prototype

```
void USBD_IP_Task(void);
```

13.6.2 Data structures

13.6.2.1 USB_IP_INIT_DATA

Description

Structure which stores the parameters of the IP component.

Type definition

```
typedef struct {
    U8          EPIn;
    U8          EPOut;
    U8          EPInt;
    const USB_IP_NI_DRIVER_API * pDriverAPI;
    USB_IP_NI_DRIVER_DATA      DriverData;
    const USB_RNDIS_DEVICE_INFO * pRndisDevInfo;
} USB_IP_INIT_DATA;
```

Structure members

Member	Description
EPIn	Bulk IN endpoint to send data packets to the USB host.
EPOut	Bulk OUT endpoint to receive data packets from the USB host.
EPInt	Interrupt IN endpoint to send notifications to the USB host.
pDriverAPI	Network interface driver API.
DriverData	Data passed at initialization to low-level driver.
pRndisDevInfo	Pointer to a filled <code>USB_RNDIS_DEVICE_INFO</code> structure.

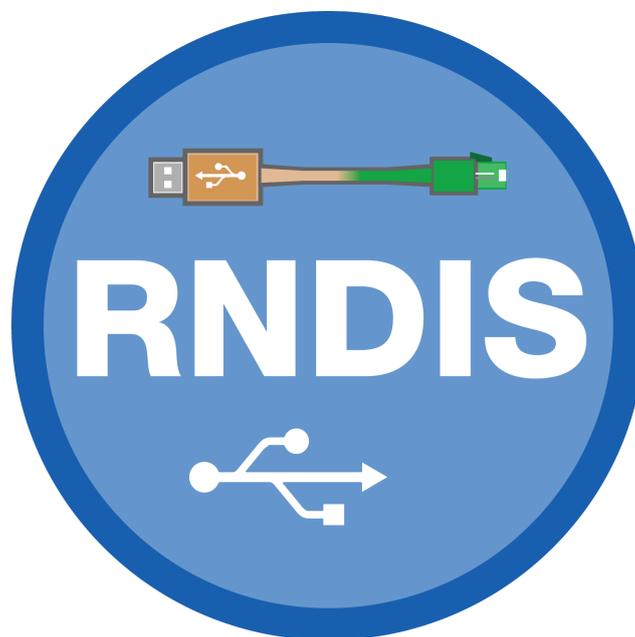
Additional information

This structure holds the endpoints that should be used with the IP component. Refer to `USBD_AddEP()` for more information about how to add an endpoint.

Chapter 14

Remote NDIS (RNDIS)

This chapter gives a general overview of the Remote Network Driver Interface Specification class and describes how to get the RNDIS component running on the target.



14.1 Overview

The Remote Network Driver Interface Specification (RNDIS) is a Microsoft proprietary USB class protocol which can be used to create a virtual Ethernet connection between a USB device and a host PC. A TCP/IP stack like *emNet* is required on the USB device side to handle the actual IP communication. Any available IP protocol (UDP, TPC, FTP, HTTP, etc.) can be used to exchange data. On a typical Cortex-M CPU running at 120 MHz, a transfer speed of about 5 MB/s can be achieved when using a high-speed USB connection.

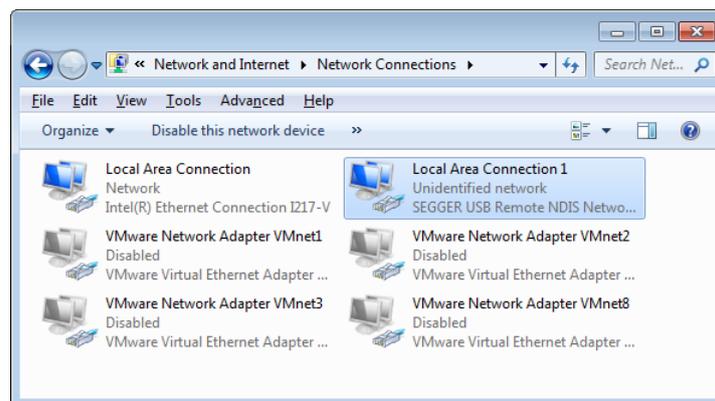
USB RNDIS is supported by all Windows operating systems starting with Windows XP, as well as by Linux with kernel versions newer than 2.6.34. An .inf file is required for the installation on Microsoft Windows systems older than Windows 7. The emUSB-Device-RNDIS package includes .inf files for Windows versions older than Windows 7. macOS will require a third-party driver to work with RNDIS, which can be downloaded from here: <https://joshuawise.com/horndis> which will only work for certain macOS versions.

emUSB-Device-RNDIS contains the following components:

- Generic USB handling
- RNDIS device class implementation
- Network interface driver which uses *emNet* as TCP/IP stack.
- A sample application demonstrating how to work with RNDIS.

14.1.1 Working with RNDIS

Any USB RNDIS device connected to a PC running the Windows operating system is listed as a separate network interface in the "Network Connections" window as shown in this screenshot:



The ping command line utility can be used to test the connection to target as shown below. If the connection is correctly established the number of the lost packets should be 0.

```

Administrator: C:\Windows\System32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

c:\Windows\System32>ping SEGGER.RNDIS

Pinging SEGGER.RNDIS [10.0.0.10] with 32 bytes of data:
Reply from 10.0.0.10: bytes=32 time<1ms TTL=64
Reply from 10.0.0.10: bytes=32 time<1ms TTL=64
Reply from 10.0.0.10: bytes=32 time<1ms TTL=64
Reply from 10.0.0.10: bytes=32 time<1ms TTL=64

Ping statistics for 10.0.0.10:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms

c:\Windows\System32>

```

14.1.2 Additional information

More technical details about RNDIS can be found here:

<https://docs.microsoft.com/en-us/windows-hardware/drivers/network/overview-of-remote-ndis--rndis->

14.2 Configuration

14.2.1 Initial Configuration

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

14.2.2 Final configuration

The configuration must only be modified when emUSB-Device is used in your final product. Refer to section *emUSB-Device Configuration* on page 48 to get detailed information about the general emUSB-Device configuration functions which have to be adapted.

Note

Due to an issue in Windows when using RNDIS within a multi-interface device the RNDIS interface must be added first. Otherwise it will not be recognized.

Note

Due to an issue with Windows 7 USB 3.0 drivers `USBD_EnableIAD()` must be used, even if the device containing the RNDIS interface is not a multi-interface device. Otherwise the device will not be recognized on USB 3.0 ports of a PC running Windows 7.

14.2.3 Class specific configuration

RNDIS specific device information must be provided by the application via the function `USB-D_RNDIS_SetDeviceInfo()` before the USB stack is started using `USBD_Start()`. A sample how to use this function can be found in the `IP_Config_RNDIS.c`. The file is located in the `Sample\RNDIS` directory of the emUSB-Device shipment. The `IP_Config_RNDIS.c` provides a ready to use layer and configuration file to be used with *embOS* and *emNet*.

14.3 Running the sample application

The sample application can be found in the `Sample\RNDIS\IP_Config_RNDIS.c` file of the emUSB-Device shipment. In order to use the sample application the SEGGER *emNet* middleware component is required. To test the emUSB-Device-RNDIS component any of the *emNet* sample applications can be used in combination with `IP_Config_RNDIS.c`. After the sample application is started the USB cable should be connected to the PC and the chosen *emNet* sample can be tested using the appropriate methods.

14.3.1 IP_Config_RNDIS.c in detail

The main part of the sample application is implemented in the function `MainTask()` which runs as an independent task.

```
// _Connect() - excerpt from IP_Config_RNDIS.c
static int _Connect(unsigned IFaceId) {
    U32 ServerIpAddress;
    U32 SubnetMask;

    ServerIpAddress = IP_BYTES2ADDR(10, 0, 0, 10);
    SubnetMask = IP_BYTES2ADDR(0xff, 0xff, 0xff, 0xf8);
    IP_SetAddrMaskEx(IFaceId, ServerIpAddress, SubnetMask);
    IP_DHCPS_ConfigPool(IFaceId, ServerIpAddress + 1, SubnetMask, 4);
    // Setup IP pool to distribute.
    IP_DHCPS_ConfigDNSAddr(IFaceId, &ServerIpAddress, 1);
    IP_DHCPS_Init(IFaceId);
    IP_DHCPS_Start(IFaceId);
    IP_NETBIOS_Init(IFaceId, _aNetNames, 0);
    // Init NetBIOS.
    IP_NETBIOS_Start(IFaceId);
    // Start NetBIOS.
    USBD_Init();
    USBD_SetDeviceInfo(&USB_DeviceInfo);
    USBD_RNDIS_SetDeviceInfo(&USB_RNDIS_DeviceInfo);
    //
    // Although we do not have a composite device, we enable IAD as a workaround
    // for the buggy Intel USB driver on Windows 7
    //
    USBD_EnableIAD();
    _AddRNDIS();
    USBD_Start();
    return 0; // Successfully connected.
}
```

The first step is to initialize the DHCP server component which assigns the IP address to the PC side. The target is configured with the IP address 10.0.0.10. The DHCP server is configured to distribute IP addresses starting from 10.0.0.11, therefore the PC will receive the IP address 10.0.0.11. Then the USB stack is initialized and the RNDIS interface is added to it. The function `_AddRNDIS()` configures all required endpoints.

```
// _AddRNDIS() - excerpt from IP_Config_RNDIS.c
static U8 _abReceiveBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
static void _AddRNDIS(void) {
    USB_RNDIS_INIT_DATA InitData;

    memset(&InitData, 0, sizeof(InitData));
    InitData.EPOut = USBD_AddeP(USB_DIR_OUT,
                                USB_TRANSFER_TYPE_BULK,
                                0,
                                _abReceiveBuffer, sizeof(_abReceiveBuffer));
    InitData.EPIn = USBD_AddeP(USB_DIR_IN, USB_TRANSFER_TYPE_BULK,
                                0, NULL, 0);
    InitData.EPInt = USBD_AddeP(USB_DIR_IN, USB_TRANSFER_TYPE_INT,
                                5, NULL, 0);
}
```

```
InitData.pDriverAPI = &USB_Driver_IP_NI;
InitData.DriverData.pDriverData = (void *)_IFaceId;
USBD_RNDIS_Add(&InitData);
}
```

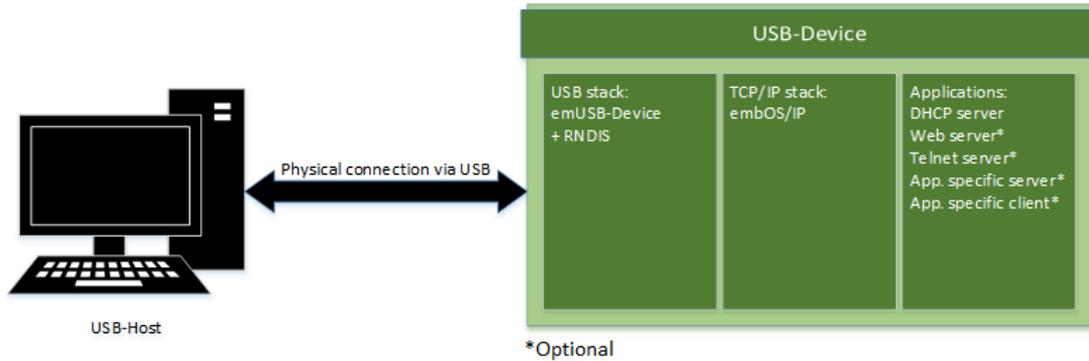
The size of `_acReceiveBuffer` buffer must be a multiple of USB max packet size. `USB_Driver_IP_NI` is the network interface driver which implements the connection to the IP stack. Optionally a HW address may be configured here, which is assigned to the PC network interface. If not set (`pHWAddr = NULL`), the HW address is generated automatically later while setting the interface up.

The IP stack is configured to use the network interface driver of `emUSB-Device-RNDIS`. For more information about the configuration of the IP stack refer to *emNet* manual.

```
// IP_X_Config() - excerpt from IP_Config.c
#include "USB_Driver_IP_NI.h"
void IP_X_Config(void) {
    <...>
    //
    // Add and configure the RNDIS driver.
    // The local IP address is 10.0.0.10/8.
    //
    IFaceId = IP_AddEtherInterface(&USB_IP_Driver);
    IP_SetIFaceConnectHook(IFaceId, _Connect);
    IP_SetIFaceDisconnectHook(IFaceId, _Disconnect);
    _IFaceId = IFaceId;
    <...>
}
```

14.4 RNDIS + emNet as a "USB Webserver"

This method of using RNDIS provides a unique customer experience where a USB device can provide a custom web page or any other service through which a customer can interact with the device.



Initially the PC recognizes an RNDIS device. In case of Windows XP and Vista a driver will be necessary, Windows 7 and above as well as Linux recognize RNDIS automatically. RNDIS from the viewpoint of the PC is a normal Network Interface Controller (NIC) and the PC handles it as such. The default behaviour is to request an IP address from a DHCP server. The PC retrieves an IP address from the DHCP-Server in the device. In our standard sample code the device has the local IP 10.0.0.10 and the PC will get 10.0.0.11 from the DHCP server. With this the configuration is complete and the user can access the web-interface located on the USB device via 10.0.0.10. To improve the ease-of-use NetBIOS can be used to give the device an easily readable name.

14.5 Target API

Function	Description
API functions	
<code>USBD_RNDIS_Add()</code>	Adds an RNDIS-class interface to the USB stack.
<code>USBD_RNDIS_Task()</code>	Obsolete.
<code>USBD_RNDIS_SetDeviceInfo()</code>	Provides device information used during USB enumeration to the stack.
Data structures	
<code>USB_RNDIS_INIT_DATA</code>	Structure which stores the parameters of the RNDIS interface.
<code>USB_RNDIS_DEVICE_INFO</code>	Device information that must be provided by the application via the function <code>USBD_RNDIS_SetDeviceInfo()</code> before the USB stack is started using <code>USBD_Start()</code> .
<code>USB_IP_NI_DRIVER_API</code>	This structure contains the callback functions for the network interface driver.
<code>USB_IP_NI_DRIVER_DATA</code>	Configuration data passed to network interface driver at initialization.

14.5.1 API functions

14.5.1.1 USBD_RNDIS_Add()

Description

Adds an RNDIS-class interface to the USB stack.

Prototype

```
void USBD_RNDIS_Add(const USB_RNDIS_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
<code>pInitData</code>	Pointer to initialization data.

Additional information

This function should be called after the initialization of the USB core to add an RNDIS interface to emUSB-Device. The initialization data is passed to the function in the structure pointed to by `pInitData`. Refer to `USB_RNDIS_INIT_DATA` for more information.

14.5.1.2 USBD_RNDIS_Task()

Description

Obsolete. Returns when USB is disconnected.

Prototype

```
void USBD_RNDIS_Task(void);
```

14.5.1.3 USBD_RNDIS_SetDeviceInfo()

Description

Provides device information used during USB enumeration to the stack.

Prototype

```
void USBD_RNDIS_SetDeviceInfo(const USB_RNDIS_DEVICE_INFO * pDeviceInfo);
```

Parameters

Parameter	Description
<code>pDeviceInfo</code>	Pointer to a <code>USB_RNDIS_DEVICE_INFO</code> structure containing the device information. Must point to static data that is not changed while the stack is running.

14.5.2 Data structures

14.5.2.1 USB_RNDIS_INIT_DATA

Description

Structure which stores the parameters of the RNDIS interface.

Type definition

```
typedef struct {
    U8                EPIn;
    U8                EPOut;
    U8                EPInt;
    const USB_IP_NI_DRIVER_API * pDriverAPI;
    USB_IP_NI_DRIVER_DATA DriverData;
    unsigned          DataInterfaceNo;
} USB_RNDIS_INIT_DATA;
```

Structure members

Member	Description
EPIn	Bulk IN endpoint for sending data to the host.
EPOut	Bulk OUT endpoint for receiving data from the host.
EPInt	Interrupt IN endpoint for sending status information.
pDriverAPI	Pointer to the Network interface driver API.
DriverData	Configuration data for the network interface driver.
DataInterfaceNo	Internal use.

Additional information

This structure holds the endpoints that should be used by the RNDIS interface ([EPIn](#), [EPOut](#) and [EPInt](#)). Refer to `USBD_AddEP()` for more information about how to add an endpoint.

14.5.2.2 USB_RNDIS_DEVICE_INFO

Description

Device information that must be provided by the application via the function `USB-D_RNDIS_SetDeviceInfo()` before the USB stack is started using `USBD_Start()`.

Type definition

```
typedef struct {
    U32    VendorId;
    char * sDescription;
    U16    DriverVersion;
} USB_RNDIS_DEVICE_INFO;
```

Structure members

Member	Description
<code>VendorId</code>	A 24-bit Organizationally Unique Identifier (OUI) of the vendor. This is the same value as the one stored in the first 3 bytes of a HW (MAC) address. Only the least significant 24 bits of the returned value are used.
<code>sDescription</code>	0-terminated ASCII string describing the device. The string is then sent to the host system.
<code>DriverVersion</code>	16-bit value representing the firmware version. The high-order byte specifies the major version and the low-order byte the minor version.

14.5.3 Driver interface

14.5.3.1 USB_IP_NI_DRIVER_API

Description

This structure contains the callback functions for the network interface driver.

Type definition

```
typedef struct {
    USB_IP_NI_INIT * pfInit;
    USB_IP_NI_GET_PACKET_BUFFER * pfGetPacketBuffer;
    USB_IP_NI_WRITE_PACKET * pfWritePacket;
    USB_IP_NI_SET_PACKET_FILTER * pfSetPacketFilter;
    USB_IP_NI_GET_LINK_STATUS * pfGetLinkStatus;
    USB_IP_NI_GET_LINK_SPEED * pfGetLinkSpeed;
    USB_IP_NI_GET_HWADDR * pfGetHWAddr;
    USB_IP_NI_GET_STATS * pfGetStats;
    USB_IP_NI_GET_MTU * pfGetMTU;
    USB_IP_NI_RESET * pfReset;
    USB_IP_NI_SET_WRITE_PACKET_FUNC * pfSetWritePacketFunc;
    USB_IP_NI_SET_REPORT_LINKSTATE_FUNC * pfSetReportLinkstateFunc;
} USB_IP_NI_DRIVER_API;
```

Structure members

Member	Description
pfInit	Initializes the driver.
pfGetPacketBuffer	Returns a buffer for a data packet.
pfWritePacket	Delivers a data packet to target IP stack.
pfSetPacketFilter	Configures the type of accepted data packets.
pfGetLinkStatus	Returns the status of the connection to target IP stack.
pfGetLinkSpeed	Returns the connection speed.
pfGetHWAddr	Returns the HW address of the PC.
pfGetStats	Returns statistical counters.
pfGetMTU	Returns the size of the largest data packet which can be transferred.
pfReset	Resets the driver.
pfSetWritePacketFunc	Allows to change the WritePacket callback which was set by pfInit .
pfSetReportLinkstateFunc	Allows to set the report link state change.

Additional information

The emUSB-Device-RNDIS/emUSB-Device-CDC-ECM component calls the functions of this API to exchange data and status information with the IP stack running on the target.

14.5.3.2 USB_IP_NI_DRIVER_DATA

Description

Configuration data passed to network interface driver at initialization.

Type definition

```
typedef struct {  
    const U8 * pHWAddr;  
    unsigned   NumBytesHWAddr;  
    void      * pDriverData;  
} USB_IP_NI_DRIVER_DATA;
```

Structure members

Member	Description
pHWAddr	Optional pointer to a HW address (or MAC address) of the host network interface.
NumBytesHWAddr	Number of bytes in the HW address. Typically 6 bytes.
pDriverData	Pointer to a user context.

Additional information

When [pHWAddr](#) is NULL the MAC is automatically generated.

14.6 RNDIS IP Driver

This section describes the emUSB-Device RNDIS IP stack interface in detail.

14.6.1 General information

This release comes with IP NI driver which uses emNet as the IP stack. If you are using emNet this chapter can be ignored. This chapter is for those who wish to write their own IP stack interface for a third-party IP stack.

The IP interface is handled through an API-table, which contains all relevant functions necessary for read/write operations and initialization.

14.6.2 Interface function list

As described above, access to network functions is realized through an API-function table of type `USB_IP_NI_DRIVER_API`. The structure is declared in `USB_Driver_IP_NI.h` and it is described in section *Data structures* on page 457

14.6.3 USB_IP_NI_DRIVER_API in detail

14.6.3.1 USB_IP_NI_INIT

Description

Initializes the driver.

Type definition

```
typedef unsigned (USB_IP_NI_INIT)(const USB_IP_NI_DRIVER_DATA * pDriverData,
                                   USB_IP_WRITE_PACKET * pfWritePacket);
```

Parameters

Parameter	Description
<code>pDriverData</code>	in Pointer to driver configuration data.
<code>pfWritePacket</code>	Callback function called by the IP stack to transmit a packet that should be sent to the USB host.

Return value

IP NI driver instance ID.

Additional information

This function is called when the RNDIS/ECM interface is added to the USB stack. Typically the function makes a local copy of the HW address passed in the `pDriverData` structure. For more information this structure refer to `USB_IP_NI_DRIVER_DATA`.

14.6.3.2 USB_IP_NI_GET_PACKET_BUFFER

Description

Returns a buffer for a data packet.

Type definition

```
typedef void * (USB_IP_NI_GET_PACKET_BUFFER)(unsigned Id,  
                                             unsigned NumBytes);
```

Parameters

Parameter	Description
Id	Instance ID returned from <code>USB_IP_NI_INIT</code> .
NumBytes	Size of the requested buffer in bytes.

Return value

`≠ NULL` Pointer to allocated buffer
`= NULL` No buffer available

Additional information

The function should allocate a buffer of the requested size. If the buffer can not be allocated a `NULL` pointer should be returned. The function is called when a data packet is received from PC. The packet data is stored in the returned buffer.

14.6.3.3 USB_IP_NI_WRITE_PACKET

Description

Delivers a data packet to target IP stack.

Type definition

```
typedef void (USB_IP_NI_WRITE_PACKET)(          unsigned Id,  
                                         const void * pData,  
                                         unsigned NumBytes);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .
<code>pData</code>	<code>in</code> Data of the received packet.
<code>NumBytes</code>	Number of bytes stored in the buffer.

Additional information

The function is called after a data packet has been received from USB. `pData` points to the buffer returned by the `USB_IP_NI_GET_PACKET_BUFFER` function.

14.6.3.4 USB_IP_NI_SET_PACKET_FILTER

Description

Configures the type of accepted data packets.

Type definition

```
typedef void (USB_IP_NI_SET_PACKET_FILTER)(unsigned Id,  
                                           U32      Mask);
```

Parameters

Parameter	Description
Id	Instance ID returned from <code>USB_IP_NI_INIT</code> .
Mask	Type of accepted data packets.

Additional information

The [Mask](#) parameter should be interpreted as a boolean value. A value different than 0 indicates that the connection to target IP stack should be established. When the function is called with the [Mask](#) parameter set to 0 the connection to target IP stack should be interrupted.

14.6.3.5 USB_IP_NI_GET_LINK_STATUS

Description

Returns the status of the connection to target IP stack.

Type definition

```
typedef int (USB_IP_NI_GET_LINK_STATUS)(unsigned Id);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .

Return value

`USB_IP_NI_LINK_STATUS_DISCONNECTED`
`USB_IP_NI_LINK_STATUS_CONNECTED`

Not connected to target IP stack.
Connected to target IP stack.

14.6.3.6 USB_IP_NI_GET_LINK_SPEED

Description

Returns the connection speed.

Type definition

```
typedef U32 (USB_IP_NI_GET_LINK_SPEED)(unsigned Id);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .

Return value

- `≠ 0` The connection speed in units of 100 bits/sec.
- `= 0` Not connected.

14.6.3.7 USB_IP_NI_GET_HWADDR

Description

Returns the HW address (MAC address) of the host network interface (PC).

Type definition

```
typedef void (USB_IP_NI_GET_HWADDR)(unsigned Id,  
                                     U8      * pAddr,  
                                     unsigned NumBytes);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .
<code>pAddr</code>	<code>out</code> The HW address.
<code>NumBytes</code>	Maximum number of bytes to store into <code>pAddr</code> .

Additional information

The returned HW address is the one passed to the driver in the call to `USB_IP_NI_INIT`. Typically the HW address is 6 bytes long.

14.6.3.8 USB_IP_NI_GET_STATS

Description

Returns statistical counters.

Type definition

```
typedef U32 (USB_IP_NI_GET_STATS)(unsigned Id,
                                int      Type);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
Type	The type of information requested. See table below.

Return value

Value of the requested statistical counter.

Additional information

The counters should be set to 0 when the USB_IP_NI_RESET function is called.

Permitted values for parameter Type	
USB_IP_NI_STATS_WRITE_PACKET_OK	Number of packets sent without errors to target IP stack.
USB_IP_NI_STATS_WRITE_PACKET_ERROR	Number of packets sent with errors to target IP stack.
USB_IP_NI_STATS_READ_PACKET_OK	Number of packets received without errors from target IP stack.
USB_IP_NI_STATS_READ_PACKET_ERROR	Number of packets received with errors from target IP stack.
USB_IP_NI_STATS_READ_NO_BUFFER	Number of packets received from target IP stack but dropped.
USB_IP_NI_STATS_READ_ALIGN_ERROR	Number of packets received from target IP stack with alignment errors.
USB_IP_NI_STATS_WRITE_ONE_COLLISION	Number of packets which were not sent to target IP stack due to the occurrence of one collision.
USB_IP_NI_STATS_WRITE_MORE_COLLISIONS	Number of packets which were not sent to target IP stack due to the occurrence of one or more collisions.

14.6.3.9 USB_IP_NI_GET_MTU

Description

Returns the maximum transmission unit, the size of the largest data packet which can be transferred.

Type definition

```
typedef U32 (USB_IP_NI_GET_MTU)(unsigned Id);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .

Return value

The MTU size in bytes. Typically 1500 bytes.

14.6.3.10 USB_IP_NI_RESET

Description

Resets the driver.

Type definition

```
typedef void (USB_IP_NI_RESET)(unsigned Id);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.

14.6.3.11 USB_IP_NI_SET_WRITE_PACKET_FUNC

Description

Changes the `USB_IP_WRITE_PACKET` callback which was added via `USB_IP_NI_INIT` to a different callback function. This function is only called by the stack when IP-over-USB is used. It is not called when RNDIS or ECM is used standalone.

Type definition

```
typedef void (USB_IP_NI_SET_WRITE_PACKET_FUNC)
              (unsigned          Id,
               USB_IP_WRITE_PACKET * pfWritePacket);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .
<code>pfWritePacket</code>	Callback function called by the IP stack to transmit a packet that should be sent to the USB host.

14.6.3.12 USB_IP_NI_SET_REPORT_LINKSTATE_FUNC

Description

Changes the `USB_IP_REPORT_LINKSTATE` callback. Normally this is called only once per initialization in order to allow to send notification to the host that the link state has been changed.

Type definition

```
typedef void (USB_IP_NI_SET_REPORT_LINKSTATE_FUNC)
              (unsigned          Id,
               USB_IP_REPORT_LINKSTATE * pfReportLinkState);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .
<code>pfReportLinkState</code>	Callback function called by the IP stack to notify the host that the link state has been changed.

Chapter 15

CDC-ECM

This chapter gives a general overview of the Communications Device Class / Ethernet Control Model class and describes how to get the ECM component running on the target.



15.1 Overview

The Communications Device Class / Ethernet Control Model is a USB class protocol of the USB Implementers Forum which can be used to create a virtual Ethernet connection between a USB device and a host PC. A TCP/IP stack like *emNet* is required on the USB device side to handle the actual IP communication. Any available IP protocol (UDP, TPC, FTP, HTTP, etc.) can be used to exchange data.

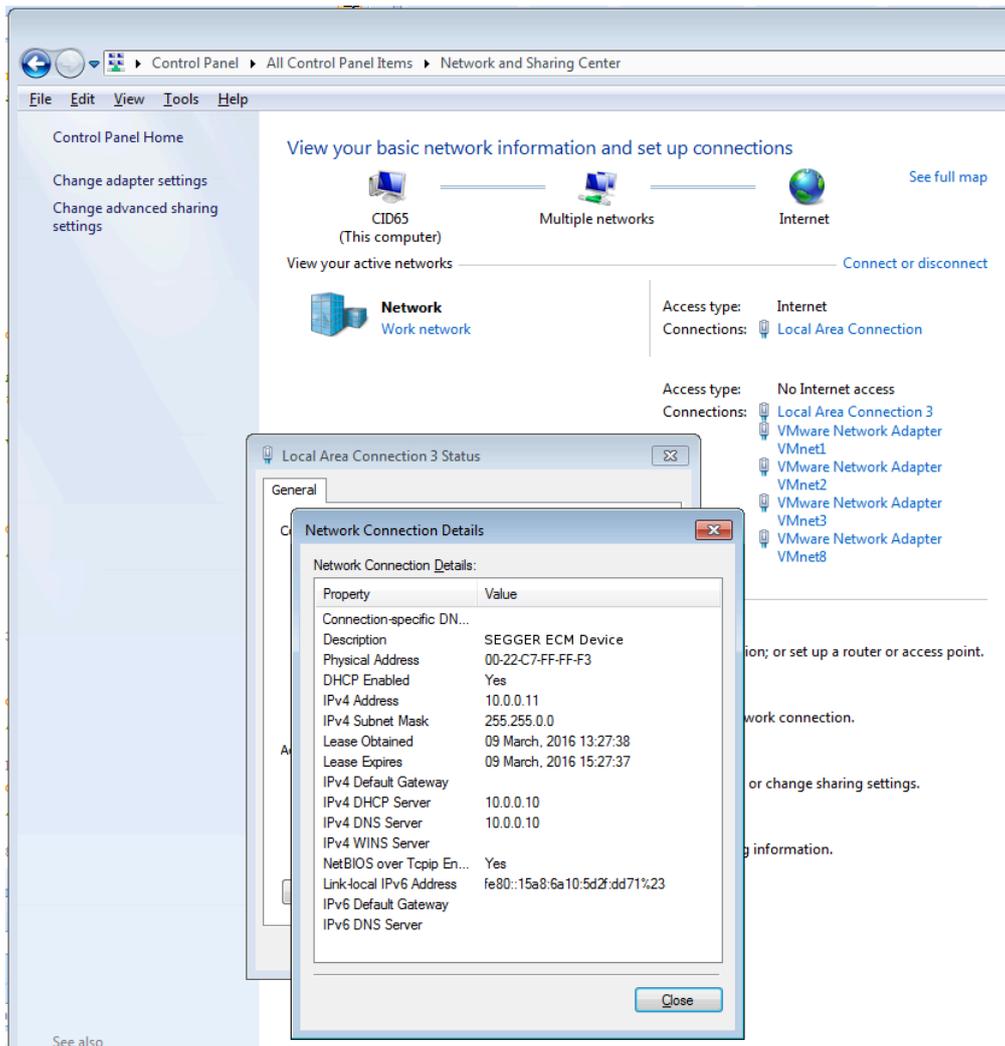
USB ECM is supported by the Linux operating system. To use it on Windows, a third party driver (not contained in emUSB-Device-ECM) has to be installed on the Windows system.

emUSB-Device-ECM contains the following components:

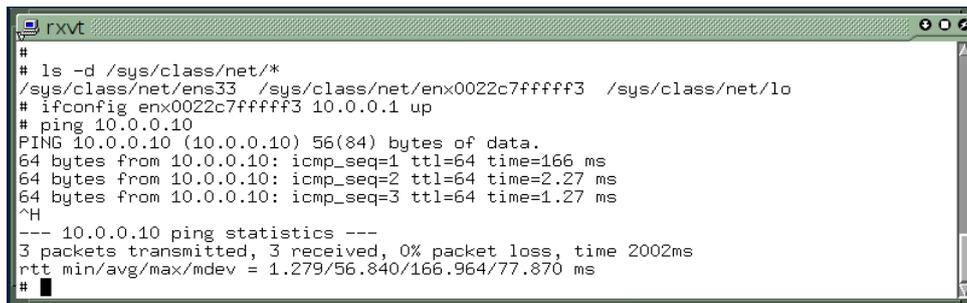
- Generic USB handling
- ECM device class implementation
- Network interface driver which uses *emNet* as TCP/IP stack.
- A sample application demonstrating how to work with ECM.

15.1.1 Working with CDC-ECM

Any USB ECM device connected to a PC running the Windows operating system (with a third-party CDC-ECM driver installed) is listed as a separate network interface in the "Network Connections" window as shown in this screenshot:



The `ping` command line utility can be used to test the connection to target as shown below. If the connection is correctly established the number of the lost packets should be 0. The following screenshot shows a manual configuration and ping on Linux.

A screenshot of a terminal window titled 'rxvt'. The terminal shows the following commands and output:

```
#  
# ls -ld /sys/class/net/*  
/sys/class/net/ens33 /sys/class/net/enx0022c7fffff3 /sys/class/net/lo  
# ifconfig enx0022c7fffff3 10.0.0.1 up  
# ping 10.0.0.10  
PING 10.0.0.10 (10.0.0.10) 56(84) bytes of data:  
64 bytes from 10.0.0.10: icmp_seq=1 ttl=64 time=166 ms  
64 bytes from 10.0.0.10: icmp_seq=2 ttl=64 time=2.27 ms  
64 bytes from 10.0.0.10: icmp_seq=3 ttl=64 time=1.27 ms  
^H  
--- 10.0.0.10 ping statistics ---  
3 packets transmitted, 3 received, 0% packet loss, time 2002ms  
rtt min/avg/max/mdev = 1.279/56.840/166.964/77.870 ms  
#
```

15.1.2 Additional information

More technical details about CDC-ECM can be found on <https://www.usb.org> in the Class definitions for Communication Devices 1.2 package: CDC Subclass for Ethernet Control Model Devices.

15.2 Configuration

15.2.1 Initial configuration

To get emUSB-Device-ECM up and running as well as doing an initial test, the configuration as delivered should not be modified. When using on Windows with a third party driver, the vendor id and product id must match the ids configured in the `.inf` file of the driver.

15.2.2 Final configuration

The configuration must only be modified when emUSB-Device is used in your final product. Refer to section *emUSB-Device Configuration* on page 48 to get detailed information about the general emUSB-Device configuration functions which have to be adapted.

15.3 Running the sample application

The sample application can be found in the `Sample\ECM\IP_Config_ECM.c` file of the emUSB-Device shipment. In order to use the sample application the SEGGER *emNet* middleware component is required. To test the emUSB-Device-ECM component any of the *emNet* sample applications can be used in combination with `IP_Config_ECM.c`. After the sample application is started the USB cable should be connected to the PC and the chosen *emNet* sample can be tested using the appropriate methods.

15.3.1 IP_Config_ECM.c in detail

The main part of the sample application is implemented in the function `MainTask()` which runs as an independent task.

```
// _Connect() - excerpt from IP_Config_ECM.c
static int _Connect(unsigned IFaceId) {
    U32 Server = IP_BYTES2ADDR(10, 0, 0, 10);
    IP_DHCPS_ConfigPool(IFaceId, IP_BYTES2ADDR(10, 0, 0, 11), 0xFF000000, 20);
    IP_DHCPS_ConfigDNSAddr(IFaceId, &Server, 1);
    IP_DHCPS_Init(IFaceId);
    IP_DHCPS_Start(IFaceId);
    USBD_Init();
    USBD_SetDeviceInfo(&USB_DeviceInfo);
    _AddECM();
    USBD_Start();
    return 0; // Successfully connected.
}
```

The first step is to initialize the DHCP server component which assigns the IP address for the PC side. The target is configured with the IP address 10.0.0.10. The DHCP server is configured to distribute IP addresses starting from 10.0.0.11, therefore the PC will receive the IP address 10.0.0.11. Then the USB stack is initialized and the ECM interface is added to it. The function `_AddECM()` configures all required endpoints and configures the HW address of the PC network interface.

```
// _AddECM() - excerpt from IP_Config_ECM.c
static U8 _abReceiveBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
static void _AddECM(void) {
    USB_ECM_INIT_DATA InitData;
    InitData.EPOut = USBD_AddeP(USB_DIR_OUT,
                                USB_TRANSFER_TYPE_BULK,
                                0,
                                _abReceiveBuffer, sizeof(_abReceiveBuffer));
    InitData.EPIn = USBD_AddeP(USB_DIR_IN, USB_TRANSFER_TYPE_BULK,
                                0, NULL, 0);
    InitData.EPInt = USBD_AddeP(USB_DIR_IN, USB_TRANSFER_TYPE_INT,
                                32, NULL, 0);
    InitData.pDriverAPI = &USB_Driver_IP_NI;
    InitData.DriverData.pDriverData = (void *)_IFaceId;
    #if 0
    InitData.DriverData.pHWAddr = "\x00\x22\xC7\xFF\xFF\xF3";
    InitData.DriverData.NumBytesHWAddr = 6;
    #endif
    USBD_ECM_Add(&InitData);
}
```

The size of `_acReceiveBuffer` buffer must be a multiple of USB max packet size. `USB_Driver_IP_NI` is the network interface driver which implements the connection to the IP stack. Optionally a HW address may be configured here, which is assigned to the PC network interface. If not set (`pHWAddr = NULL`), the HW address is generated automatically later while setting the interface up.

The IP stack is configured to use the network interface driver of emUSB-Device-ECM. For more information about the configuration of the IP stack refer to *emNet* manual.

```
// IP_X_Config() - excerpt from IP_Config.c
#include "USB_Driver_IP_NI.h"
void IP_X_Config(void) {
    <...>
    //
    // Add and configure the ECM driver.
    // The local IP address is 10.0.0.10/8.
    //
    IFaceId = IP_AddEtherInterface(&USB_IP_Driver);
    IP_SetAddrMask(0x0A00000A, 0xFF000000);
    IP_SetIFaceConnectHook(IFaceId, _Connect);
    IP_SetIFaceDisconnectHook(IFaceId, _Disconnect);
    _IFaceId = IFaceId;
    <...>
}
```

15.4 Target API

Function	Description
API functions	
USBD_ECM_Add ()	Adds an ECM-class interface to the USB stack.
USBD_ECM_Task ()	Obsolete.
Data structures	
USB_ECM_INIT_DATA	Initialization data for ECM interface.
USB_IP_NI_DRIVER_API	This structure contains the callback functions for the network interface driver.
USB_IP_NI_DRIVER_DATA	Configuration data passed to network interface driver at initialization.

15.4.1 API functions

15.4.1.1 USBD_ECM_Add()

Description

Adds an ECM-class interface to the USB stack.

Prototype

```
void USBD_ECM_Add(const USB_ECM_INIT_DATA * pInitData);
```

Parameters

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

Additional information

This function should be called after the initialization of the USB core to add an ECM interface to emUSB-Device. The initialization data is passed to the function in the structure pointed to by `pInitData`. Refer to `USB_ECM_INIT_DATA` for more information.

15.4.1.2 USBD_ECM_Task()

Description

Obsolete. Returns when USB is disconnected.

Prototype

```
void USBD_ECM_Task(void);
```

15.4.2 Data structures

15.4.2.1 USB_ECM_INIT_DATA

Description

Initialization data for ECM interface.

Type definition

```
typedef xxx {
    U8                EPIn;
    U8                EPOut;
    U8                EPInt;
    const USB_IP_NI_DRIVER_API * pDriverAPI;
    USB_IP_NI_DRIVER_DATA DriverData;
    unsigned          DataInterfaceNo;
} USB_ECM_INIT_DATA;
```

Structure members

Member	Description
EPIn	Bulk IN endpoint for sending data to the host.
EPOut	Bulk OUT endpoint for receiving data from the host. The buffer associated to this endpoint must be big enough to hold a complete IP packet.
EPInt	Interrupt IN endpoint for sending status information.
pDriverAPI	Pointer to the Network interface driver API. See USB_IP_NI_DRIVER_API .
DriverData	Configuration data for the network interface driver.
DataInterfaceNo	Internal use.

Additional information

This structure holds the endpoints that should be used by the ECM interface ([EPIn](#), [EPOut](#) and [EPInt](#)). Refer to [USBD_AddEP\(\)](#) for more information about how to add an endpoint.

15.4.3 Driver interface

15.4.3.1 USB_IP_NI_DRIVER_API

Description

This structure contains the callback functions for the network interface driver.

Type definition

```
typedef struct {
    USB_IP_NI_INIT * pfInit;
    USB_IP_NI_GET_PACKET_BUFFER * pfGetPacketBuffer;
    USB_IP_NI_WRITE_PACKET * pfWritePacket;
    USB_IP_NI_SET_PACKET_FILTER * pfSetPacketFilter;
    USB_IP_NI_GET_LINK_STATUS * pfGetLinkStatus;
    USB_IP_NI_GET_LINK_SPEED * pfGetLinkSpeed;
    USB_IP_NI_GET_HWADDR * pfGetHWAddr;
    USB_IP_NI_GET_STATS * pfGetStats;
    USB_IP_NI_GET_MTU * pfGetMTU;
    USB_IP_NI_RESET * pfReset;
    USB_IP_NI_SET_WRITE_PACKET_FUNC * pfSetWritePacketFunc;
    USB_IP_NI_SET_REPORT_LINKSTATE_FUNC * pfSetReportLinkstateFunc;
} USB_IP_NI_DRIVER_API;
```

Structure members

Member	Description
pfInit	Initializes the driver.
pfGetPacketBuffer	Returns a buffer for a data packet.
pfWritePacket	Delivers a data packet to target IP stack.
pfSetPacketFilter	Configures the type of accepted data packets.
pfGetLinkStatus	Returns the status of the connection to target IP stack.
pfGetLinkSpeed	Returns the connection speed.
pfGetHWAddr	Returns the HW address of the PC.
pfGetStats	Returns statistical counters.
pfGetMTU	Returns the size of the largest data packet which can be transferred.
pfReset	Resets the driver.
pfSetWritePacketFunc	Allows to change the WritePacket callback which was set by pfInit .
pfSetReportLinkstateFunc	Allows to set the report link state change.

Additional information

The emUSB-Device-RNDIS/emUSB-Device-CDC-ECM component calls the functions of this API to exchange data and status information with the IP stack running on the target.

15.4.3.2 USB_IP_NI_DRIVER_DATA

Description

Configuration data passed to network interface driver at initialization.

Type definition

```
typedef struct {  
    const U8 * pHWAddr;  
    unsigned   NumBytesHWAddr;  
    void      * pDriverData;  
} USB_IP_NI_DRIVER_DATA;
```

Structure members

Member	Description
pHWAddr	Optional pointer to a HW address (or MAC address) of the host network interface.
NumBytesHWAddr	Number of bytes in the HW address. Typically 6 bytes.
pDriverData	Pointer to a user context.

Additional information

When [pHWAddr](#) is NULL the MAC is automatically generated.

15.5 CDC-ECM IP Driver

This section describes the emUSB-Device CDC-ECM IP stack interface in detail.

15.5.1 General information

This release comes with IP NI driver which uses emNet as the IP stack. If you are using emNet this chapter can be ignored. This chapter is for those who wish to write their own IP stack interface for a third-party IP stack.

The IP interface is handled through an API-table, which contains all relevant functions necessary for read/write operations and initialization.

15.5.2 Interface function list

As described above, access to network functions is realized through an API-function table of type `USB_IP_NI_DRIVER_API`. The structure is declared in `USB_Driver_IP_NI.h` and it is described in section *Data structures* on page 457

15.5.3 USB_IP_NI_DRIVER_API in detail

15.5.3.1 USB_IP_NI_INIT

Description

Initializes the driver.

Type definition

```
typedef unsigned (USB_IP_NI_INIT)(const USB_IP_NI_DRIVER_DATA * pDriverData,
                                  USB_IP_WRITE_PACKET * pfWritePacket);
```

Parameters

Parameter	Description
<code>pDriverData</code>	in Pointer to driver configuration data.
<code>pfWritePacket</code>	Callback function called by the IP stack to transmit a packet that should be sent to the USB host.

Return value

IP NI driver instance ID.

Additional information

This function is called when the RNDIS/ECM interface is added to the USB stack. Typically the function makes a local copy of the HW address passed in the `pDriverData` structure. For more information this structure refer to `USB_IP_NI_DRIVER_DATA`.

15.5.3.2 USB_IP_NI_GET_PACKET_BUFFER

Description

Returns a buffer for a data packet.

Type definition

```
typedef void * (USB_IP_NI_GET_PACKET_BUFFER)(unsigned Id,  
                                             unsigned NumBytes);
```

Parameters

Parameter	Description
Id	Instance ID returned from <code>USB_IP_NI_INIT</code> .
NumBytes	Size of the requested buffer in bytes.

Return value

`≠ NULL` Pointer to allocated buffer
`= NULL` No buffer available

Additional information

The function should allocate a buffer of the requested size. If the buffer can not be allocated a `NULL` pointer should be returned. The function is called when a data packet is received from PC. The packet data is stored in the returned buffer.

15.5.3.3 USB_IP_NI_WRITE_PACKET

Description

Delivers a data packet to target IP stack.

Type definition

```
typedef void (USB_IP_NI_WRITE_PACKET)(          unsigned Id,  
                                         const void * pData,  
                                         unsigned NumBytes);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .
<code>pData</code>	<code>in</code> Data of the received packet.
<code>NumBytes</code>	Number of bytes stored in the buffer.

Additional information

The function is called after a data packet has been received from USB. `pData` points to the buffer returned by the `USB_IP_NI_GET_PACKET_BUFFER` function.

15.5.3.4 USB_IP_NI_SET_PACKET_FILTER

Description

Configures the type of accepted data packets.

Type definition

```
typedef void (USB_IP_NI_SET_PACKET_FILTER)(unsigned Id,  
                                           U32      Mask);
```

Parameters

Parameter	Description
Id	Instance ID returned from <code>USB_IP_NI_INIT</code> .
Mask	Type of accepted data packets.

Additional information

The [Mask](#) parameter should be interpreted as a boolean value. A value different than 0 indicates that the connection to target IP stack should be established. When the function is called with the [Mask](#) parameter set to 0 the connection to target IP stack should be interrupted.

15.5.3.5 USB_IP_NI_GET_LINK_STATUS

Description

Returns the status of the connection to target IP stack.

Type definition

```
typedef int (USB_IP_NI_GET_LINK_STATUS)(unsigned Id);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .

Return value

`USB_IP_NI_LINK_STATUS_DISCONNECTED`
`USB_IP_NI_LINK_STATUS_CONNECTED`

Not connected to target IP stack.
Connected to target IP stack.

15.5.3.6 USB_IP_NI_GET_LINK_SPEED

Description

Returns the connection speed.

Type definition

```
typedef U32 (USB_IP_NI_GET_LINK_SPEED)(unsigned Id);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .

Return value

≠ 0 The connection speed in units of 100 bits/sec.
= 0 Not connected.

15.5.3.7 USB_IP_NI_GET_HWADDR

Description

Returns the HW address (MAC address) of the host network interface (PC).

Type definition

```
typedef void (USB_IP_NI_GET_HWADDR)(unsigned Id,  
                                     U8      * pAddr,  
                                     unsigned NumBytes);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .
<code>pAddr</code>	<code>out</code> The HW address.
<code>NumBytes</code>	Maximum number of bytes to store into <code>pAddr</code> .

Additional information

The returned HW address is the one passed to the driver in the call to `USB_IP_NI_INIT`. Typically the HW address is 6 bytes long.

15.5.3.8 USB_IP_NI_GET_STATS

Description

Returns statistical counters.

Type definition

```
typedef U32 (USB_IP_NI_GET_STATS)(unsigned Id,
                                int      Type);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
Type	The type of information requested. See table below.

Return value

Value of the requested statistical counter.

Additional information

The counters should be set to 0 when the USB_IP_NI_RESET function is called.

Permitted values for parameter Type	
USB_IP_NI_STATS_WRITE_PACKET_OK	Number of packets sent without errors to target IP stack.
USB_IP_NI_STATS_WRITE_PACKET_ERROR	Number of packets sent with errors to target IP stack.
USB_IP_NI_STATS_READ_PACKET_OK	Number of packets received without errors from target IP stack.
USB_IP_NI_STATS_READ_PACKET_ERROR	Number of packets received with errors from target IP stack.
USB_IP_NI_STATS_READ_NO_BUFFER	Number of packets received from target IP stack but dropped.
USB_IP_NI_STATS_READ_ALIGN_ERROR	Number of packets received from target IP stack with alignment errors.
USB_IP_NI_STATS_WRITE_ONE_COLLISION	Number of packets which were not sent to target IP stack due to the occurrence of one collision.
USB_IP_NI_STATS_WRITE_MORE_COLLISIONS	Number of packets which were not sent to target IP stack due to the occurrence of one or more collisions.

15.5.3.9 USB_IP_NI_GET_MTU

Description

Returns the maximum transmission unit, the size of the largest data packet which can be transferred.

Type definition

```
typedef U32 (USB_IP_NI_GET_MTU)(unsigned Id);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .

Return value

The MTU size in bytes. Typically 1500 bytes.

15.5.3.10 USB_IP_NI_RESET

Description

Resets the driver.

Type definition

```
typedef void (USB_IP_NI_RESET)(unsigned Id);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .

15.5.3.11 USB_IP_NI_SET_WRITE_PACKET_FUNC

Description

Changes the `USB_IP_WRITE_PACKET` callback which was added via `USB_IP_NI_INIT` to a different callback function. This function is only called by the stack when IP-over-USB is used. It is not called when RNDIS or ECM is used standalone.

Type definition

```
typedef void (USB_IP_NI_SET_WRITE_PACKET_FUNC)
              (unsigned          Id,
               USB_IP_WRITE_PACKET * pfWritePacket);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .
<code>pfWritePacket</code>	Callback function called by the IP stack to transmit a packet that should be sent to the USB host.

15.5.3.12 USB_IP_NI_SET_REPORT_LINKSTATE_FUNC

Description

Changes the `USB_IP_REPORT_LINKSTATE` callback. Normally this is called only once per initialization in order to allow to send notification to the host that the link state has been changed.

Type definition

```
typedef void (USB_IP_NI_SET_REPORT_LINKSTATE_FUNC)
              (unsigned          Id,
               USB_IP_REPORT_LINKSTATE * pfReportLinkState);
```

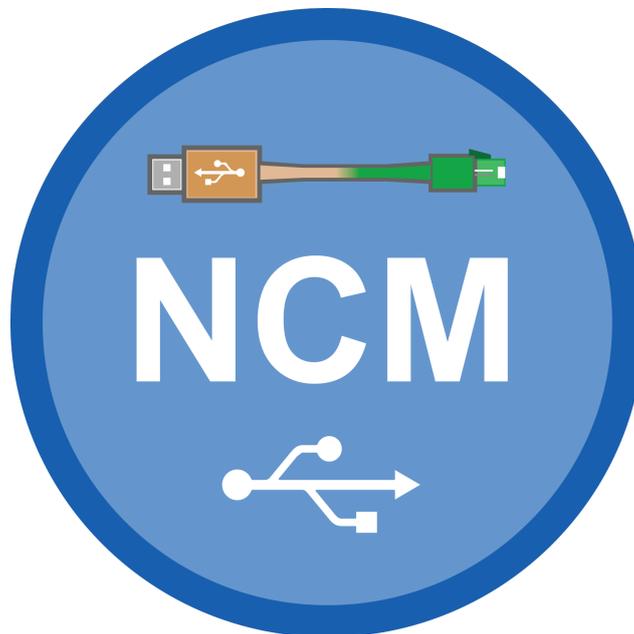
Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .
<code>pfReportLinkState</code>	Callback function called by the IP stack to notify the host that the link state has been changed.

Chapter 16

CDC-NCM

This chapter gives a general overview of the Communications Device Class / Network Control Model class and describes how to get the NCM component running on the target.



16.1 Overview

The Communications Device Class / Network Control Model is a USB class protocol of the USB Implementers Forum which can be used to create a virtual Ethernet connection between a USB device and a host PC. A TCP/IP stack like *emNet* is required on the USB device side to handle the actual IP communication. Any available IP protocol (UDP, TPC, FTP, HTTP, etc.) can be used to exchange data.

USB CDC-NCM is supported by the Linux (kernel > 2.6.38) and macOS operating systems out of the box. To use it on Windows, a third party driver (not contained in emUSB-Device-NCM) has to be installed on the Windows system.

emUSB-Device-NCM contains the following components:

- Generic USB handling
- NCM device class implementation
- Network interface driver which uses *emNet* as TCP/IP stack.
- A sample application demonstrating how to work with NCM.

16.1.1 Working with CDC-NCM

The `ping` command line utility can be used to test the connection to target. If the connection is correctly established the number of the lost packets should be 0.

16.1.2 Additional information

More technical details about CDC-NCM can be found on <https://www.usb.org> in the Network Control Model Devices Specification v1.0 and errata and Adopters Agreement package.

16.2 Configuration

16.2.1 Initial configuration

To get emUSB-Device-NCM up and running as well as doing an initial test, the configuration as delivered should not be modified. When using on Windows with a third party driver, the vendor id and product id must match the ids configured in the `.inf` file of the driver.

16.2.2 Final configuration

The configuration must only be modified when emUSB-Device is used in your final product. Refer to section *emUSB-Device Configuration* on page 48 to get detailed information about the general emUSB-Device configuration functions which have to be adapted.

16.3 Running the sample application

The sample application can be found in the `Sample\NCM\IP_Config_NCM.c` file of the emUSB-Device shipment. In order to use the sample application the SEGGER *emNet* middleware component is required. To test the emUSB-Device-NCM component any of the *emNet* sample applications can be used in combination with `IP_Config_NCM.c`. After the sample application is started the USB cable should be connected to the PC and the chosen *emNet* sample can be tested using the appropriate methods.

16.3.1 IP_Config_NCM.c in detail

The main part of the sample application is implemented in the function `MainTask()` which runs as an independent task.

```
// _Connect() - excerpt from IP_Config_NCM.c
static int _Connect(unsigned IFaceId) {
    U32 Server = IP_BYTES2ADDR(10, 0, 0, 10);
    IP_DHCPConfigPool(IFaceId, IP_BYTES2ADDR(10, 0, 0, 11), 0xFF000000, 20);
    IP_DHCPConfigDNSAddr(IFaceId, &Server, 1);
    IP_DHCPInit(IFaceId);
    IP_DHCPStart(IFaceId);
    USBD_Init();
    USBD_SetDeviceInfo(&USB_DeviceInfo);
    _AddNCM();
    USBD_Start();
    return 0; // Successfully connected.
}
```

The first step is to initialize the DHCP server component which assigns the IP address for the PC side. The target is configured with the IP address 10.0.0.10. The DHCP server is configured to distribute IP addresses starting from 10.0.0.11, therefore the PC will receive the IP address 10.0.0.11. Then the USB stack is initialized and the NCM interface is added to it. The function `_AddNCM()` configures all required endpoints and configures the HW address of the PC network interface.

```
// _AddNCM() - excerpt from IP_Config_NCM.c
static U8 _abReceiveBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
static void _AddNCM(void) {
    USB_NCM_INIT_DATA InitData;
    InitData.EPOut = USBD_AddeP(USB_DIR_OUT,
                                USB_TRANSFER_TYPE_BULK,
                                0,
                                _abReceiveBuffer, sizeof(_abReceiveBuffer));
    InitData.EPIn = USBD_AddeP(USB_DIR_IN, USB_TRANSFER_TYPE_BULK,
                                0, NULL, 0);
    InitData.EPInt = USBD_AddeP(USB_DIR_IN, USB_TRANSFER_TYPE_INT,
                                32, NULL, 0);
    InitData.pDriverAPI = &USB_Driver_IP_NI;
    InitData.DriverData.pDriverData = (void *)_IFaceId;
    USBD_NCM_Add(&InitData);
}
```

The size of `_acReceiveBuffer` buffer must be a multiple of USB max packet size. `USB_Driver_IP_NI` is the network interface driver which implements the connection to the IP stack. Optionally a HW address may be configured here, which is assigned to the PC network interface. If not set (`pHWAddr = NULL`), the HW address is generated automatically later while setting the interface up.

The IP stack is configured to use the network interface driver of emUSB-Device-NCM. For more information about the configuration of the IP stack refer to *emNet* manual.

```
// IP_X_Config() - excerpt from IP_Config.c
#include "USB_Driver_IP_NI.h"
void IP_X_Config(void) {
```

```
<...>
//
// Add and configure the NCM driver.
// The local IP address is 10.0.0.10/8.
//
IFaceId = IP_AddEtherInterface(&USB_IP_Driver);
IP_SetAddrMask(0x0A00000A, 0xFF000000);
IP_SetIFaceConnectHook(IFaceId, _Connect);
IP_SetIFaceDisconnectHook(IFaceId, _Disconnect);
_IFaceId = IFaceId;
<...>
}
```

16.4 Target API

Function	Description
API functions	
USBD_NCM_Add ()	Adds an NCM-class interface to the USB stack.
Data structures	
USB_NCM_INIT_DATA	Initialization data for NCM interface.
USB_IP_NI_DRIVER_API	This structure contains the callback functions for the network interface driver.
USB_IP_NI_DRIVER_DATA	Configuration data passed to network interface driver at initialization.

16.4.1 API functions

16.4.1.1 USBD_NCM_Add()

Description

Adds an NCM-class interface to the USB stack.

Prototype

```
void USBD_NCM_Add(const USB_NCM_INIT_DATA * pInitData);
```

Parameters

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

Additional information

This function should be called after the initialization of the USB core to add an NCM interface to emUSB-Device. The initialization data is passed to the function in the structure pointed to by `pInitData`. Refer to `USB_NCM_INIT_DATA` for more information.

16.4.2 Data structures

16.4.2.1 USB_NCM_INIT_DATA

Description

Initialization data for NCM interface.

Type definition

```
typedef xxx {
    U8                EPIn;
    U8                EPOut;
    U8                EPInt;
    const USB_IP_NI_DRIVER_API * pDriverAPI;
    USB_IP_NI_DRIVER_DATA    DriverData;
    unsigned            DataInterfaceNo;
} USB_NCM_INIT_DATA;
```

Structure members

Member	Description
EPIn	Bulk IN endpoint for sending data to the host.
EPOut	Bulk OUT endpoint for receiving data from the host. The buffer associated to this endpoint must be big enough to hold a complete IP packet.
EPInt	Interrupt IN endpoint for sending status information.
pDriverAPI	Pointer to the Network interface driver API. See USB_IP_NI_DRIVER_API .
DriverData	Configuration data for the network interface driver.
DataInterfaceNo	Internal use.

Additional information

This structure holds the endpoints that should be used by the NCM interface ([EPIn](#), [EPOut](#) and [EPInt](#)). Refer to [USBD_AddEP\(\)](#) for more information about how to add an endpoint.

16.4.3 Driver interface

16.4.3.1 USB_IP_NI_DRIVER_API

Description

This structure contains the callback functions for the network interface driver.

Type definition

```
typedef struct {
    USB_IP_NI_INIT * pfInit;
    USB_IP_NI_GET_PACKET_BUFFER * pfGetPacketBuffer;
    USB_IP_NI_WRITE_PACKET * pfWritePacket;
    USB_IP_NI_SET_PACKET_FILTER * pfSetPacketFilter;
    USB_IP_NI_GET_LINK_STATUS * pfGetLinkStatus;
    USB_IP_NI_GET_LINK_SPEED * pfGetLinkSpeed;
    USB_IP_NI_GET_HWADDR * pfGetHWAddr;
    USB_IP_NI_GET_STATS * pfGetStats;
    USB_IP_NI_GET_MTU * pfGetMTU;
    USB_IP_NI_RESET * pfReset;
    USB_IP_NI_SET_WRITE_PACKET_FUNC * pfSetWritePacketFunc;
    USB_IP_NI_SET_REPORT_LINKSTATE_FUNC * pfSetReportLinkstateFunc;
} USB_IP_NI_DRIVER_API;
```

Structure members

Member	Description
pfInit	Initializes the driver.
pfGetPacketBuffer	Returns a buffer for a data packet.
pfWritePacket	Delivers a data packet to target IP stack.
pfSetPacketFilter	Configures the type of accepted data packets.
pfGetLinkStatus	Returns the status of the connection to target IP stack.
pfGetLinkSpeed	Returns the connection speed.
pfGetHWAddr	Returns the HW address of the PC.
pfGetStats	Returns statistical counters.
pfGetMTU	Returns the size of the largest data packet which can be transferred.
pfReset	Resets the driver.
pfSetWritePacketFunc	Allows to change the WritePacket callback which was set by pfInit .
pfSetReportLinkstateFunc	Allows to set the report link state change.

Additional information

The emUSB-Device-RNDIS/emUSB-Device-CDC-ECM component calls the functions of this API to exchange data and status information with the IP stack running on the target.

16.4.3.2 USB_IP_NI_DRIVER_DATA

Description

Configuration data passed to network interface driver at initialization.

Type definition

```
typedef struct {  
    const U8 * pHWAddr;  
    unsigned   NumBytesHWAddr;  
    void      * pDriverData;  
} USB_IP_NI_DRIVER_DATA;
```

Structure members

Member	Description
pHWAddr	Optional pointer to a HW address (or MAC address) of the host network interface.
NumBytesHWAddr	Number of bytes in the HW address. Typically 6 bytes.
pDriverData	Pointer to a user context.

Additional information

When [pHWAddr](#) is NULL the MAC is automatically generated.

16.5 CDC-NCM IP Driver

This section describes the emUSB-Device CDC-NCM IP stack interface in detail.

16.5.1 General information

This release comes with IP NI driver which uses emNet as the IP stack. If you are using emNet this chapter can be ignored. This chapter is for those who wish to write their own IP stack interface for a third-party IP stack.

The IP interface is handled through an API-table, which contains all relevant functions necessary for read/write operations and initialization.

16.5.2 Interface function list

As described above, access to network functions is realized through an API-function table of type `USB_IP_NI_DRIVER_API`. The structure is declared in `USB_Driver_IP_NI.h` and it is described in section *Data structures* on page 457

16.5.3 USB_IP_NI_DRIVER_API in detail

16.5.3.1 USB_IP_NI_INIT

Description

Initializes the driver.

Type definition

```
typedef unsigned (USB_IP_NI_INIT)(const USB_IP_NI_DRIVER_DATA * pDriverData,
                                   USB_IP_WRITE_PACKET * pfWritePacket);
```

Parameters

Parameter	Description
<code>pDriverData</code>	in Pointer to driver configuration data.
<code>pfWritePacket</code>	Callback function called by the IP stack to transmit a packet that should be sent to the USB host.

Return value

IP NI driver instance ID.

Additional information

This function is called when the RNDIS/ECM interface is added to the USB stack. Typically the function makes a local copy of the HW address passed in the `pDriverData` structure. For more information this structure refer to `USB_IP_NI_DRIVER_DATA`.

16.5.3.2 USB_IP_NI_GET_PACKET_BUFFER

Description

Returns a buffer for a data packet.

Type definition

```
typedef void * (USB_IP_NI_GET_PACKET_BUFFER)(unsigned Id,  
                                             unsigned NumBytes);
```

Parameters

Parameter	Description
Id	Instance ID returned from <code>USB_IP_NI_INIT</code> .
NumBytes	Size of the requested buffer in bytes.

Return value

≠ `NULL` Pointer to allocated buffer
= `NULL` No buffer available

Additional information

The function should allocate a buffer of the requested size. If the buffer can not be allocated a `NULL` pointer should be returned. The function is called when a data packet is received from PC. The packet data is stored in the returned buffer.

16.5.3.3 USB_IP_NI_WRITE_PACKET

Description

Delivers a data packet to target IP stack.

Type definition

```
typedef void (USB_IP_NI_WRITE_PACKET)(          unsigned Id,  
                                         const void * pData,  
                                         unsigned NumBytes);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .
<code>pData</code>	<code>in</code> Data of the received packet.
<code>NumBytes</code>	Number of bytes stored in the buffer.

Additional information

The function is called after a data packet has been received from USB. `pData` points to the buffer returned by the `USB_IP_NI_GET_PACKET_BUFFER` function.

16.5.3.4 USB_IP_NI_SET_PACKET_FILTER

Description

Configures the type of accepted data packets.

Type definition

```
typedef void (USB_IP_NI_SET_PACKET_FILTER)(unsigned Id,  
                                           U32      Mask);
```

Parameters

Parameter	Description
Id	Instance ID returned from <code>USB_IP_NI_INIT</code> .
Mask	Type of accepted data packets.

Additional information

The [Mask](#) parameter should be interpreted as a boolean value. A value different than 0 indicates that the connection to target IP stack should be established. When the function is called with the [Mask](#) parameter set to 0 the connection to target IP stack should be interrupted.

16.5.3.5 USB_IP_NI_GET_LINK_STATUS

Description

Returns the status of the connection to target IP stack.

Type definition

```
typedef int (USB_IP_NI_GET_LINK_STATUS)(unsigned Id);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .

Return value

`USB_IP_NI_LINK_STATUS_DISCONNECTED`
`USB_IP_NI_LINK_STATUS_CONNECTED`

Not connected to target IP stack.
Connected to target IP stack.

16.5.3.6 USB_IP_NI_GET_LINK_SPEED

Description

Returns the connection speed.

Type definition

```
typedef U32 (USB_IP_NI_GET_LINK_SPEED)(unsigned Id);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .

Return value

≠ 0 The connection speed in units of 100 bits/sec.
= 0 Not connected.

16.5.3.7 USB_IP_NI_GET_HWADDR

Description

Returns the HW address (MAC address) of the host network interface (PC).

Type definition

```
typedef void (USB_IP_NI_GET_HWADDR)(unsigned Id,  
                                     U8      * pAddr,  
                                     unsigned NumBytes);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .
<code>pAddr</code>	<code>out</code> The HW address.
<code>NumBytes</code>	Maximum number of bytes to store into <code>pAddr</code> .

Additional information

The returned HW address is the one passed to the driver in the call to `USB_IP_NI_INIT`. Typically the HW address is 6 bytes long.

16.5.3.8 USB_IP_NI_GET_STATS

Description

Returns statistical counters.

Type definition

```
typedef U32 (USB_IP_NI_GET_STATS)(unsigned Id,
                                int      Type);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
Type	The type of information requested. See table below.

Return value

Value of the requested statistical counter.

Additional information

The counters should be set to 0 when the USB_IP_NI_RESET function is called.

Permitted values for parameter Type	
USB_IP_NI_STATS_WRITE_PACKET_OK	Number of packets sent without errors to target IP stack.
USB_IP_NI_STATS_WRITE_PACKET_ERROR	Number of packets sent with errors to target IP stack.
USB_IP_NI_STATS_READ_PACKET_OK	Number of packets received without errors from target IP stack.
USB_IP_NI_STATS_READ_PACKET_ERROR	Number of packets received with errors from target IP stack.
USB_IP_NI_STATS_READ_NO_BUFFER	Number of packets received from target IP stack but dropped.
USB_IP_NI_STATS_READ_ALIGN_ERROR	Number of packets received from target IP stack with alignment errors.
USB_IP_NI_STATS_WRITE_ONE_COLLISION	Number of packets which were not sent to target IP stack due to the occurrence of one collision.
USB_IP_NI_STATS_WRITE_MORE_COLLISIONS	Number of packets which were not sent to target IP stack due to the occurrence of one or more collisions.

16.5.3.9 USB_IP_NI_GET_MTU

Description

Returns the maximum transmission unit, the size of the largest data packet which can be transferred.

Type definition

```
typedef U32 (USB_IP_NI_GET_MTU)(unsigned Id);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .

Return value

The MTU size in bytes. Typically 1500 bytes.

16.5.3.10 USB_IP_NI_RESET

Description

Resets the driver.

Type definition

```
typedef void (USB_IP_NI_RESET)(unsigned Id);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .

16.5.3.11 USB_IP_NI_SET_WRITE_PACKET_FUNC

Description

Changes the `USB_IP_WRITE_PACKET` callback which was added via `USB_IP_NI_INIT` to a different callback function. This function is only called by the stack when IP-over-USB is used. It is not called when RNDIS or ECM is used standalone.

Type definition

```
typedef void (USB_IP_NI_SET_WRITE_PACKET_FUNC)
              (unsigned          Id,
               USB_IP_WRITE_PACKET * pfWritePacket);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .
<code>pfWritePacket</code>	Callback function called by the IP stack to transmit a packet that should be sent to the USB host.

16.5.3.12 USB_IP_NI_SET_REPORT_LINKSTATE_FUNC

Description

Changes the `USB_IP_REPORT_LINKSTATE` callback. Normally this is called only once per initialization in order to allow to send notification to the host that the link state has been changed.

Type definition

```
typedef void (USB_IP_NI_SET_REPORT_LINKSTATE_FUNC)
              (unsigned          Id,
               USB_IP_REPORT_LINKSTATE * pfReportLinkState);
```

Parameters

Parameter	Description
<code>Id</code>	Instance ID returned from <code>USB_IP_NI_INIT</code> .
<code>pfReportLinkState</code>	Callback function called by the IP stack to notify the host that the link state has been changed.

Chapter 17

Audio

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



17.1 Overview

The USB Audio device class is a USB class protocol which can be used to transfer sound data from a device to a host and vice versa.

Audio is supported by most operating systems out of the box and the installation of additional drivers is not required.

emUSB-Device-Audio comes as a complete package and contains the following:

- Generic USB handling
- USB Audio V1 device class implementation
- Sample application showing how to work with Audio

17.2 Introduction

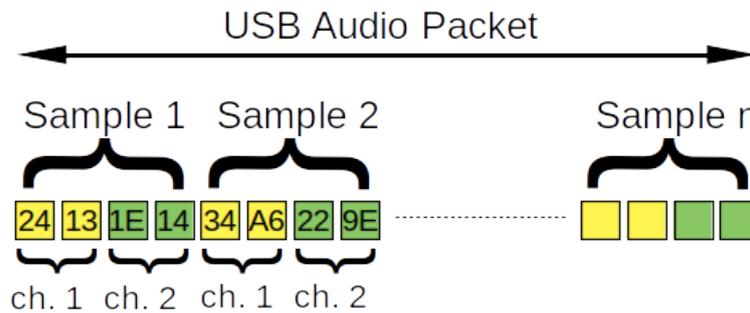
SEGGER’s implementation of the Audio class V1.0 is designed with minimal resource usage in mind, especially targeted to embedded devices. The implementation supports the usage of a “speaker” (input/output audio terminal with a feature terminal for controls) and a “microphone” (input/output audio terminal).

The speaker and microphone can be used independently of each other, both can be enabled at the same time allowing audio transfer in either direction (headset-like operation).

The Audio class supports adaptive synchronization for OUT endpoints and asynchronous synchronization for IN endpoints, synchronous synchronization type for both directions, commands SET_CUR, GET_CUR, SET_MIN, GET_MIN, SET_MAX, GET_MAX, SET_RES, GET_RES, for the speaker interface feature unit controls are supported (volume, mute, etc.).

Note
emUSB-Device-Audio does not provide drivers/codecs for any audio peripherals, writing a driver to interface with the audio hardware is the responsibility of the customer.

With emUSB-Device-Audio Audio data is transferred in the PCM encoding. The Audio class transfers multiple audio samples in a single packet. In the following sample the audio class is configured with 2 channels (stereo) and 16 bit data per channel:



The length of a complete audio packet is equal to the bits per sample rounded up to bytes, multiplied by the number of channels and the sample rate, then divided by 1000 as a packet is sent every millisecond. For a sample rate of 48000, 16 bits per sample, 2 channels the calculation is as follows:

$$48000 * 16/8 * 2 / 1000 = 192 \text{ bytes}$$

For a sample rate of 44100, 16 bits per sample, 2 channels the calculation is as follows:

$$44100 * 16/8 * 2 / 1000 = 176.4 \text{ bytes}$$

Since we can not transfer 0.4 bytes the audio packets need to be 176 bytes (44 samples) and each 10th packet (sample size divided by the remainder: 4 / 0.4) should contain 45 samples (180 bytes) to make sure the sample rate remains at 44100.

Note
On macOS (tested with Big Sur) when a device is using USB high-speed for an audio device the speaker interface will not be shown, unless the ISO endpoint type of the out endpoint is "USB_ISO_SYNC_TYPE_ASYNCHRONOUS". Microphone interfaces are not affected.

17.3 Configuration

17.3.1 Initial configuration

To get emUSB-Device-Audio up and running as well as doing an initial test, the configuration as delivered with the sample application should not be modified.

17.3.2 Final configuration

The configuration must only be modified when emUSB-Device is integrated in your final product. Refer to section *emUSB-Device Configuration* on page 48 for detailed information about the generic information functions which have to be adapted.

Windows

Windows systems save the audio settings for each USB Audio device inside the Windows registry (interfaces, number of channels, sub-frame-size, bit resolution and sample frequency). These values are saved for the USB Vendor ID and the USB Product ID. When a device with the same USB Vendor ID and USB Product ID enumerates a second time the audio settings are checked against the saved values inside the Windows registry. If the settings do not match the device will not function (Windows will not request any audio data from it).

It is not known why Windows behaves this way, other operating systems are not affected.

When developing a USB Audio device and experimenting with different sample rates, bit resolutions, etc. it is advised to remove the device from the registry after each change to the audio settings or to use a different USB product ID after every change.

17.3.3 Using the microphone interface

When using the microphone sample applications with a PC it is not immediately clear whether they work as the PC only receives the audio data. To listen to the data being sent from the target running emUSB-Device-Audio to your PC it is necessary to enable a loopback mode which will transfer the audio data from the microphone interface to the physical speakers connected to your PC.

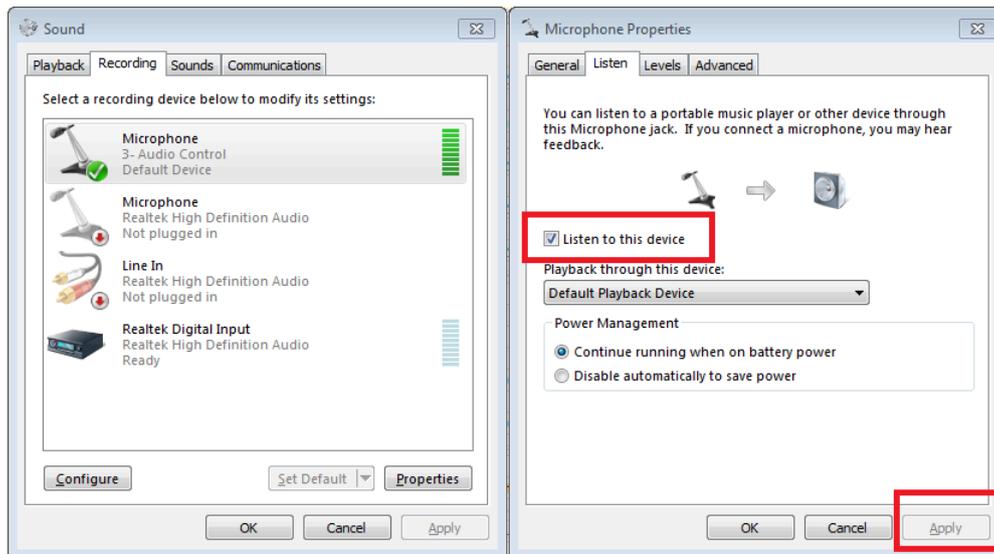
Linux

This guide assumes you are using pulse audio.

- Make sure the device running emUSB-Device-Audio microphone sample is selected as the default sound input device.
- Make sure your speakers (or headphones) are selected as the default sound output device.
- Run `pactl load-module module-loopback` to enable loopback.
- At this point you should hear the sound being produced by the microphone sample.
- You can run `pactl unload-module module-loopback` to disable the loopback mode.

Windows

- Make sure your speakers (or headphones) are selected as the default sound output device.
- In the sound configuration of the device running emUSB-Device-Audio microphone sample tick the "Listen to this device" checkbox and click "Apply".
- At this point you should hear the sound being produced by the microphone sample.



Mac

At the time of writing no built-in way of looping back audio is known. But there are a couple of third party applications out there which can enable loopback mode for macOS.

17.3.4 Using the speaker interface

When using the speaker sample applications the PC merely needs to be configured to use the device running emUSB-Device-Audio as the default output sound device.

17.4 Target API

Function	Description
API functions	
<code>USBD_AUDIO_Add()</code>	Adds an Audio interface to the USB stack.
<code>USBD_AUDIO_Read_Task()</code>	Task function of the Audio component which processes data received from host.
<code>USBD_AUDIO_Write_Task()</code>	Task function of the Audio component which processes data sent to the host.
<code>USBD_AUDIO_Start_Play()</code>	Starts providing audio data to the host using the microphone terminal of the audio class.
<code>USBD_AUDIO_Stop_Play()</code>	Stops providing audio data to the host.
<code>USBD_AUDIO_Start_Listen()</code>	Starts receiving audio data from the host using the speaker terminal of the audio class.
<code>USBD_AUDIO_Stop_Listen()</code>	Stops receiving audio data from the host.
<code>USBD_AUDIO_Set_Timeouts()</code>	Sets the timeouts used by the <code>USBD_AUDIO_Read_Task()</code> and <code>USBD_AUDIO_Write_Task()</code> when listen/play is active.
Data structures	
<code>USBD_AUDIO_INIT_DATA</code>	Initialization data for the Audio class instance.
<code>USBD_AUDIO_IF_CONF</code>	Initialization structure for an audio microphone/speaker interface.
<code>USBD_AUDIO_FORMAT</code>	Initialization data for a single audio format.
<code>USBD_AUDIO_UNITS</code>	This structure contains IDs used for a particular audio interface.
Function definitions	
<code>USBD_AUDIO_TX_FUNC</code>	Definition of the callback which is called when audio data is sent.
<code>USBD_AUDIO_RX_FUNC</code>	Definition of the callback which is called when audio data is received.
<code>USBD_AUDIO_CONTROL_FUNC</code>	Definition of the callback which is called when audio commands are received.

17.4.1 API functions

17.4.1.1 USBD_AUDIO_Add()

Description

Adds an Audio interface to the USB stack.

Prototype

```
USB_AUDIO_HANDLE USB_AUDIO_Add(const USB_AUDIO_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
<code>pInitData</code>	Pointer to a <code>USB_AUDIO_INIT_DATA</code> structure containing values for the initialization. This structure's memory must remain available while the audio class is being used. The application should not put this structure on the stack.

Return value

`USB_AUDIO_HANDLE` - Handle for the added Audio instance.

Additional information

After the initialization of USB core, this is the first function that needs to be called when an Audio interface is used with emUSB-Device. The structure `USB_AUDIO_INIT_DATA` has to be initialized before `USB_AUDIO_Add()` is called. Refer to `USB_AUDIO_INIT_DATA` for more information.

For the Audio component to be functional one or both of the following functions have to be created as a task: `USB_AUDIO_Read_Task()`, `USB_AUDIO_Write_Task()`.

17.4.1.2 USBD_AUDIO_Read_Task()

Description

Task function of the Audio component which processes data received from host. Handles operations of the speaker interface. Has to be created as a separate task.

Prototype

```
void USBD_AUDIO_Read_Task(void);
```

Additional information

Only necessary if the speaker interface is used. The function returns only when `USB-D_DeInit()` is called.

17.4.1.3 USBD_AUDIO_Write_Task()

Description

Task function of the Audio component which processes data sent to the host. Handles operations of the microphone interface. Has to be created as a separate task.

Prototype

```
void USBD_AUDIO_Write_Task(void);
```

Additional information

Only necessary if the microphone interface is used. The function returns only when `USB-D_DeInit()` is called.

17.4.1.4 USBD_AUDIO_Start_Play()

Description

Starts providing audio data to the host using the microphone terminal of the audio class.

Prototype

```
int USBD_AUDIO_Start_Play(          USBD_AUDIO_HANDLE  hInst,
                                const U8              * pBufIn);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid Audio instance, returned by <code>USBDAUDIO_Add()</code> .
<code>pBufIn</code>	Buffer initially used with IN transfers (microphone interface). Can be changed inside the <code>USBDAUDIO_TX_FUNC</code> callback.

Return value

= 0 Success.
 < 0 An error occurred.

Additional information

This function enables the registered TX user function (`USBDAUDIO_TX_FUNC`). The callback is called after every successful transfer and should move the buffer pointer to the next audio packet accordingly or fill the same buffer with new data. The callback is called in an interrupt context. The execution of the callback together with the internal routines must never take longer than 1 millisecond because the audio class must send one packet every millisecond.

17.4.1.5 USBD_AUDIO_Stop_Play()

Description

Stops providing audio data to the host.

Prototype

```
void USBD_AUDIO_Stop_Play(USB_D_AUDIO_HANDLE hInst);
```

Parameters

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

17.4.1.6 USBD_AUDIO_Start_Listen()

Description

Starts receiving audio data from the host using the speaker terminal of the audio class.

Prototype

```
int USBD_AUDIO_Start_Listen(USB_D_AUDIO_HANDLE  hInst,
                           U8                   * pBufOut);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid Audio instance, returned by <code>USB_D_AUDIO_Add()</code> .
<code>pBufOut</code>	Buffer initially used with OUT transfers (speaker interface). Can be changed inside the <code>USB_D_AUDIO_RX_FUNC</code> callback.

Return value

= 0 Success.
 < 0 An error occurred.

Additional information

This function enables the registered user callback function (`USB_D_AUDIO_RX_FUNC`) which is called before the host sends data to the target. Inside the callback you may read the received data. The callback is called in an interrupt context. The execution of the callback together with the internal routines must never take longer than 1 millisecond because the audio class must send one packet every millisecond.

17.4.1.7 USBD_AUDIO_Stop_Listen()

Description

Stops receiving audio data from the host.

Prototype

```
void USBD_AUDIO_Stop_Listen(USB_D_AUDIO_HANDLE hInst);
```

Parameters

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

17.4.1.8 USBD_AUDIO_Set_Timeouts()

Description

Sets the timeouts used by the `USBD_AUDIO_Read_Task()` and `USBD_AUDIO_Write_Task()` when listen/play is active.

Prototype

```
void USBD_AUDIO_Set_Timeouts(USBD_AUDIO_HANDLE hInst,  
                             unsigned          ReadTimeout,  
                             unsigned          WriteTimeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid Audio instance, returned by <code>USBD_AUDIO_Add()</code> .
<code>ReadTimeout</code>	Read (OUT) timeout in milliseconds.
<code>WriteTimeout</code>	Write (IN) timeout in milliseconds.

17.4.2 Data structures

17.4.2.1 USBD_AUDIO_INIT_DATA

Description

Initialization data for the Audio class instance.

Type definition

```
typedef struct {
    U8                EPIIn;
    U8                EPOut;
    unsigned          OutPacketSize;
    USBD_AUDIO_RX_FUNC * pfOnOut;
    USBD_AUDIO_TX_FUNC * pfOnIn;
    USBD_AUDIO_CONTROL_FUNC * pfOnControl;
    void              * pControlUserContext;
    U8                NumInterfaces;
    const USBD_AUDIO_IF_CONF * paInterfaces;
} USBD_AUDIO_INIT_DATA;
```

Structure members

Member	Description
EPIIn	Isosynchronous IN endpoint for sending data to the host. If microphone functionality is not desired set this to 0.
EPOut	Isosynchronous OUT endpoint for receiving data from the host. If speaker functionality is not desired set this to 0.
OutPacketSize	Size of a single audio OUT packet. Must be calculated as follows: (highest used) SampleRate * NumChannels * BitsPerSample / 8 / 1000
pfOnOut	Pointer to a function of type <code>USB_D_AUDIO_RX_FUNC</code> which handles incoming audio data. Needs to be set when the speaker interface is used.
pfOnIn	Pointer to a function of type <code>USB_D_AUDIO_TX_FUNC</code> which handles outgoing audio data. Needs to be set when the microphone interface is used.
pfOnControl	Pointer to a function of type <code>USB_D_AUDIO_CONTROL_FUNC</code> which handles audio commands. Always needs to be set.
pControlUserContext	Pointer to a user context which is passed to the <code>pfOnControl</code> function. Optional, can be <code>NULL</code> .
NumInterfaces	Number of elements in the <code>paInterfaces</code> array.
paInterfaces	Pointer to an array of structures of type <code>USB_D_AUDIO_IF_CONF</code> which contain configuration data for the audio interfaces.

17.4.2.2 USBD_AUDIO_IF_CONF

Description

Initialization structure for an audio microphone/speaker interface. Only one speaker and one microphone is supported.

Type definition

```
typedef struct {
    U8          Flags;
    U8          Controls;
    U8          TotalNrChannels;
    U8          NumFormats;
    const USBD_AUDIO_FORMAT * paFormats;
    U16         bmChannelConfig;
    U16         TerminalType;
    USBD_AUDIO_UNITS * pUnits;
} USBD_AUDIO_IF_CONF;
```

Structure members

Member	Description
<code>Flags</code>	Reserved. Set to zero.
<code>Controls</code>	Bitmask, a bit set to 1 indicates that the mentioned Control is supported: <ul style="list-style-type: none"> • b0: Mute • b1: Volume • b2: Bass • b3: Mid • b4: Treble • b5: Graphic Equalizer • b6: Automatic Gain • b7: Delay
<code>TotalNrChannels</code>	Number of audio channels for this interface.
<code>NumFormats</code>	Number of elements inside the <code>paFormats</code> array.
<code>paFormats</code>	Pointer to any array of <code>USB_D_AUDIO_FORMAT</code> structures.
<code>bmChannelConfig</code>	Bit map indicating the spatial locations of channels. Important: this value should not be left at 0 to avoid an issue with Windows. The bits correspond to the following locations: <ul style="list-style-type: none"> • b0: Left Front (L) • b1: Right Front (R) • b2: Center Front (C) • b3: Low Frequency Enhancement (LFE) • b4: Left Surround (LS) • b5: Right Surround (RS) • b6: Left of Center (LC) • b7: Right of Center (RC) • b8: Surround (S) • b9: Side Left (SL) • b10: Side Right (SR) • b11: Top (T) • b15..12: Reserved Channels are assigned to locations in ascending order. E.g. if b6 and b11 are set and the other bits are zero channel 0 will be "LC" and channel 1 will be "T". Having more channels than bits set in this bit map is valid, the channels which do not have a bit set will be considered to have a non-predefined spatial position.

Member	Description
<code>TerminalType</code>	<p>Defines the type of speaker/microphone for this interface. Only one speaker and one microphone is supported! The following defines can be used:</p> <ul style="list-style-type: none"> • <code>USB_AUDIO_TERMTYPE_INPUT_UNDEFINED</code> • <code>USB_AUDIO_TERMTYPE_INPUT_MICROPHONE</code> • <code>USB_AUDIO_TERMTYPE_INPUT_DESKTOP_MICROPHONE</code> • <code>USB_AUDIO_TERMTYPE_INPUT_PERSONAL_MICROPHONE</code> • <code>USB_AUDIO_TERMTYPE_INPUT_OMNI_DIRECTIONAL_MICROPHONE</code> • <code>USB_AUDIO_TERMTYPE_INPUT_MICROPHONE_ARRAY</code> • <code>USB_AUDIO_TERMTYPE_INPUT_PROCESSING_MICROPHONE_ARRAY</code> • <code>USB_AUDIO_TERMTYPE_OUTPUT_UNDEFINED</code> • <code>USB_AUDIO_TERMTYPE_OUTPUT_SPEAKER</code> • <code>USB_AUDIO_TERMTYPE_OUTPUT_HEADPHONES</code> • <code>USB_AUDIO_TERMTYPE_OUTPUT_HEAD_MOUNTED_DISPLAY_AUDIO</code> • <code>USB_AUDIO_TERMTYPE_OUTPUT_DESKTOP_SPEAKER</code> • <code>USB_AUDIO_TERMTYPE_OUTPUT_ROOM_SPEAKER</code> • <code>USB_AUDIO_TERMTYPE_OUTPUT_COMMUNICATION_SPEAKER</code> • <code>USB_AUDIO_TERMTYPE_OUTPUT_LOW_FREQUENCY_EFFECTS_SPEAKER</code>
<code>pUnits</code>	<p>Pointer to a structure of type <code>USBD_AUDIO_UNITS</code>. This structure is filled by the emUSB-Device Audio class during initialization.</p>

17.4.2.3 USBD_AUDIO_FORMAT

Description

Initialization data for a single audio format.

Type definition

```
typedef struct {
    U8    Flags;
    U8    NrChannels;
    U8    SubFrameSize;
    U8    BitResolution;
    U32   SamFreq;
} USBD_AUDIO_FORMAT;
```

Structure members

Member	Description
Flags	Reserved. Set to zero.
NrChannels	Number of channels in this format. Must never be greater than <code>USBDAUDIO_IF_CONF->TotalNrChannels</code> . While it is possible to configure less than the total number of channels for a format most host operating systems do not support such configurations.
SubFrameSize	Size of an audio frame in bytes. Must be able to hold BitResolution bits.
BitResolution	Number of bits inside the audio frame dedicated to audio data. (Any remaining bits are padding.)
SamFreq	Supported sample frequency in Hz.

17.4.2.4 USBD_AUDIO_UNITS

Description

This structure contains IDs used for a particular audio interface. The application should leave those values at zero, they are set by the stack after `USBD_AUDIO_Add()` has been called.

Type definition

```
typedef struct {
    U8  Flags;
    U8  InterfaceNo;
    U8  AltInterfaceNo;
    U8  InputTerminalID;
    U8  OutputTerminalID;
    U8  FeatureUnitID;
} USBD_AUDIO_UNITS;
```

Structure members

Member	Description
Flags	Reserved. Set to zero.
InterfaceNo	USB Interface number of the audio interface. Set by the emUSB-Device stack.
AltInterfaceNo	Alternate setting number of the USB interface. Set by the emUSB-Device stack.
InputTerminalID	ID of the input terminal. Set by the emUSB-Device stack.
OutputTerminalID	ID of the output terminal. Set by the emUSB-Device stack.
FeatureUnitID	ID of the feature unit. Set by the emUSB-Device stack.

17.4.3 Function definitions

17.4.3.1 USBD_AUDIO_TX_FUNC

Description

Definition of the callback which is called when audio data is sent. This callback is called in the context of `USB_D_AUDIO_Write_Task()`

Type definition

```
typedef void USBD_AUDIO_TX_FUNC(
    void * pUserContext,
    const U8 * * ppNextBuffer,
    U32 * pNextPacketSize);
```

Parameters

Parameter	Description
<code>pUserContext</code>	User context which is passed to the callback.
<code>ppNextBuffer</code>	Buffer containing audio samples which should match the configuration from microphone <code>USB_D_AUDIO_IF_CONF</code> . Initially this points to the <code>pBufIn</code> from the call to <code>USB_D_AUDIO_Start_Play</code> function. The user can change this pointer to a different buffer which will be used in the next transaction or fill the same buffer with new data.
<code>pNextBufferSize</code>	Size of the next buffer.

Example

```
static void _cbOnIn(void * pUserContext,
    const U8 ** ppNextBuffer,
    U32 * pNextPacketSize) {
    USB_MEMCPY(_pBufMic, _pDataSource, PACKET_SIZE_IN);
    *ppNextBuffer = _pBufMic;
    *pNextPacketSize = PACKET_SIZE_IN;
}
```

17.4.3.2 USBD_AUDIO_RX_FUNC

Description

Definition of the callback which is called when audio data is received. This callback is called in the context of `USBD_AUDIO_Read_Task()`. The default timeout is 50 milliseconds.

Type definition

```
typedef void USBD_AUDIO_RX_FUNC(void * pUserContext,
                                int    NumBytesReceived,
                                U8 * * ppNextBuffer,
                                U32 *  pNextBufferSize);
```

Parameters

Parameter	Description
<code>pUserContext</code>	User context which is passed to the callback.
<code>NumBytesReceived</code>	The number of bytes which have been read in this transaction.
<code>ppNextBuffer</code>	Buffer containing audio samples which should match the configuration from speaker <code>USBD_AUDIO_IF_CONF</code> . Initially this points to the <code>pBufOut</code> from the <code>USBD_AUDIO_INIT_DATA</code> structure. The user can change this pointer to a different buffer which will be used in the next transaction or leave it as it is and copy the data from this buffer elsewhere.
<code>pNextBufferSize</code>	Size of the next buffer.

Example

```

static U8  _acBuf1[BUFFER_SIZE];
static U8  _acBuf2[BUFFER_SIZE];
static U8 * _pBuf;
static U8 * _pBufComplete;
static U32 _NumBytesInFullBuffer;
static U32 _NumBytesInBuffer;

// Receive callback function.
static void _cbOnOut(void * pUserContext,
                    int NumBytesReceived,
                    U8 ** ppNextBuffer,
                    U32 * pNextBufferSize) {

    char MBEvent;

    //
    // Check if the _next_ transfer would still fit into the buffer.
    // If not - switch the buffer.
    //
    if ((_NumBytesInBuffer + NumBytesReceived + PACKET_SIZE_OUT) > BUFFER_SIZE) {
        //
        // Switch buffers.
        //
        if (_CurrentBuffer == 1) {
            _CurrentBuffer = 2;
            _pBuf = _acBuf2;
        } else {
            _CurrentBuffer = 1;
            _pBuf = _acBuf1;
        }
        _NumBytesInFullBuffer = _NumBytesInBuffer + NumBytesReceived;
        _NumBytesInBuffer = 0;
        MBEvent = BUFFER_FULL;
        //
        // Notify the task that a buffer is full.
        //
        if (OS_PutMailCond1(&MailBox, &MBEvent) != 0) {
            printf("Missed packet.");
        }
    } else {
        _pBuf += NumBytesReceived;
        _NumBytesInBuffer += NumBytesReceived;
    }
    *ppNextBuffer = _pBuf;
}

```

17.4.3.3 USBD_AUDIO_CONTROL_FUNC

Description

Definition of the callback which is called when audio commands are received. This callback is called in an interrupt context.

Type definition

```
typedef int USBD_AUDIO_CONTROL_FUNC(void * pUserContext,
                                     U8      Event,
                                     U8      Unit,
                                     U8      ControlSelector,
                                     U8      * pBuffer,
                                     U32     NumBytes,
                                     U8      InterfaceNo,
                                     U8      AltSetting);
```

Parameters

Parameter	Description
<code>pUserContext</code>	User context which is passed to the callback.
<code>Event</code>	Audio event ID.
<code>Unit</code>	ID of the feature unit. In case of <code>USB_AUDIO_PLAYBACK_*</code> and <code>USB_AUDIO_RECORD_*</code> : 0.
<code>ControlSelector</code>	ID of the control. In case of <code>USB_AUDIO_PLAYBACK_*</code> and <code>USB_AUDIO_RECORD_*</code> : 0.
<code>pBuffer</code>	In case of GET events: pointer to a buffer into which the callback should write the reply. In case of SET events: pointer to a buffer containing the command value. In case of <code>USB_AUDIO_PLAYBACK_*</code> and <code>USB_AUDIO_RECORD_*</code> : NULL.
<code>NumBytes</code>	In case of GET events: requested size of the reply in bytes. In case of SET events: number of bytes in <code>pBuffer</code> . In case of <code>USB_AUDIO_PLAYBACK_*</code> and <code>USB_AUDIO_RECORD_*</code> : 0.
<code>InterfaceNo</code>	The number of the USB interface for which the event was issued.
<code>AltSetting</code>	The alternative setting number of the USB interface for which the event was issued.

Return value

- = 0 Audio command was handled by the callback. The stack will send the reply.
- ≠ 0 Audio command was not handled by the callback. The stack will STALL the request.

Additional information

`USB_AUDIO_PLAYBACK_*` & `USB_AUDIO_RECORD_*` events are sent upon receiving a Set Interface USB request for Alternate Setting 1 for the respective interface (microphone or speaker). By default an Audio interface is set to Alternative Setting 0 in which it can not send or receive anything. The host switches the Alternative Setting to 1 when it has to send data to the device, this can be e.g. triggered by pressing "play" in your music player. Normally the host should switch the device back to Alternative Interface 0 when it has stopped sending audio data. This works well on Linux and OS X, but does not work reliably on Windows. When using Windows as a host it seems to depend on the application whether these events are generated or not. E.g. with some applications you will receive `USB_AUDIO_PLAYBACK_START` when "play" is pressed, but `USB_AUDIO_PLAYBACK_STOP` will not be sent when "pause" or "stop" is pressed. Relying on these events to check when the host has stopped sending data is not advised, instead set timeouts via `USB_AUDIO_Set_Timeouts` and check for timeouts inside your `USB_AUDIO_RX_FUNC` and `USB_AUDIO_TX_FUNC`.

Example

```
// Control callback function.
static int _cbOnControl(void * pUserContext,
                        U8 Event,
                        U8 Unit,
                        U8 ControlSelector,
                        U8 * pBuffer,
                        U32 NumBytes) {

    int r;

    r = 0;
    switch (Event) {
    case USB_AUDIO_SET_CUR:
        switch (ControlSelector) {
        case USB_AUDIO_MUTE_CONTROL:
            if (*pBuffer == 1) {
                _SetMute(1);
            } else {
                _SetMute(0);
            }
            break;
        default:
            r = 1;
            break;
        }
        break;
    <...>
    <handle other commands>
    <...>
    }
    return r;
}
```

Chapter 18

USB Video device Class (UVC)

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



18.1 Overview

The USB video device class (UVC) is a USB class protocol which can be used to transfer video data from a device to a host.

UVC is supported by most operating systems out of the box and the installation of additional drivers is not required.

emUSB-Device-UVC comes as a complete package and contains the following:

- Generic USB handling
- USB video device class implementation
- Sample application showing how to work with UVC

18.2 Configuration

18.2.1 Initial configuration

To get emUSB-Device-UVC up and running as well as doing an initial test, the configuration as delivered with the sample application should not be modified.

18.2.1.1 Uncompressed video format

Video data is transmitted using emUSB-Device Video in the uncompressed format. YUYV422 is the specific format used by the USB video device class. The format uses luminance (the brightness) and chrominance (the coloration) to display pictures. This is best explained by taking a look at the actual data, the first 8 bytes of a YUYV422 frame are defined as follows:

- 1 byte luminance of the first pixel (Y)
- 1 byte chrominance (blue) of the first and second pixel (U)
- 1 byte luminance of the second pixel (Y)
- 1 byte chrominance (red) of the first and second pixel (V)
- 1 byte luminance of the third pixel (Y)
- 1 byte chrominance (blue) of the third and fourth pixel (U)
- 1 byte luminance of the fourth pixel (Y)
- 1 byte chrominance (red) of the third and fourth pixel (V)

Using 1 byte for the chrominance of two pixels allows this format to save a byte per pixel when compared to the common RGB format (2 pixels YUYV422 - 4 bytes, 2 pixels RGB888 - 6 bytes).

Data must be provided in the YUYV422 format when using `USBD_UVC_Write()` or `USBD_UVC_WriteEx()`.

18.2.2 Final configuration

The configuration must only be modified when emUSB-Device is integrated in your final product. Refer to section *emUSB-Device Configuration* on page 48 for detailed information about the generic information functions which have to be adapted.

18.3 Target API

Function	Description
API functions	
<code>USBD_UVC_Add()</code>	Adds a UVC interface to the USB stack.
<code>USBD_UVC_Write()</code>	Writes frame data to the host.
<code>USBD_UVC_WriteEx()</code>	Writes frame data to the host using single packets.
<code>USBD_UVC_SetOnResolutionChange()</code>	Allows to set a callback which is called when the host changes the resolution of the UVC frame.
Data structures	
<code>USBD_UVC_INIT_DATA</code>	Initialization data for UVC interface.
<code>USBD_UVC_BUFFER</code>	Structure which contains information about the UVC ring buffer.
<code>USBD_UVC_DATA_BUFFER</code>	Structure which contains values for a single buffer.
<code>USBD_UVC_RESOLUTION</code>	Structure describing a valid image resolution.
Function prototypes	
<code>USB_UVC_ON_RESOLUTION_CHANGE</code>	Callback function description which is set via <code>USBD_UVC_SetOnResolutionChange()</code> .

18.3.1 API functions

18.3.1.1 USBD_UVC_Add()

Description

Adds a UVC interface to the USB stack.

Prototype

```
int USBD_UVC_Add(const USBD_UVC_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
<code>pInitData</code>	Pointer to a <code>USB_D_UVC_INIT_DATA</code> structure containing values for the initialization of the UVC module.

Return value

0 - Successfully added. 1 - An error occurred.

Additional information

After the initialization of USB core, this is the first function that needs to be called when a UVC interface is used with emUSB-Device. The structure `USB_D_UVC_INIT_DATA` has to be initialized before `USB_D_UVC_Add()` is called. Refer to `USB_D_UVC_INIT_DATA` for more information.

18.3.1.2 USBD_UVC_Write()

Description

Writes frame data to the host.

Prototype

```
int USBD_UVC_Write(const U8      * pData,
                  unsigned NumBytes,
                  U8      UserFlags);
```

Parameters

Parameter	Description
<code>pData</code>	Pointer to a buffer containing the frame data.
<code>NumBytes</code>	Number of bytes in the buffer.
<code>UserFlags</code>	Flags to be added to the frame. Following flags are currently supported: <ul style="list-style-type: none"> <code>USB_D_UVC_END_OF_FRAME</code> - Should be set with the last <code>USB_D_UVC_Write()</code> call for a single frame.

Return value

0 - All data written to the buffer. -1 - An error occurred (device disconnected).

Additional information

It is up to the application how much data it provides through this function, but providing a buffer containing a whole video frame will cause the least overhead. The application has to set the flag `USB_D_UVC_END_OF_FRAME` when the last data part of a frame was written via `USB_D_UVC_Write()`. Internally this function will write data into the buffers which have been initialized by the call to `USB_D_UVC_Add()`. This allows for the buffers to be filled with video data before data is requested by the host application. The data transmission itself happens inside an interrupt triggered event callback inside the UVC module.

With every transmission the UVC module must add a payload header to the transfer. Therefore if the application needs to achieve maximum throughput the application should write `MaxPacketSize - USB_D_UVC_PAYLOAD_HEADER_SIZE` chunks.

Do not mix usage of `USB_D_UVC_Write()` and `USB_D_UVC_WriteEx()`.

Example

Sample describing a write operation where a frame is entirely available in a single buffer:

```
USB_D_UVC_Write(h, WholeFrame, sizeof(WholeFrame), USB_D_UVC_END_OF_FRAME);
```

Sample describing a write operation where a frame is only available in chunks:

```
U32 NumBytesAtOnce;
U32 NumBytesTotal;
U8  Flags;

NumBytesTotal = 153600; // Fixed frame size.
NumBytesAtOnce = SEGGER_MIN(sizeof(SmallBuffer), NumBytesTotal);
Flags = 0;
while (NumBytesTotal) {
    USB_D_UVC_Write(h, SmallBuffer, NumBytesAtOnce, Flags);
    NumBytesTotal -= NumBytesAtOnce;
    NumBytesAtOnce = SEGGER_MIN(sizeof(SmallBuffer), NumBytesTotal);
    if (NumBytesTotal <= sizeof(SmallBuffer)) {
        Flags = USB_D_UVC_END_OF_FRAME; // This will be the last write for this frame.
    }
}
```

```
}
```

18.3.1.3 USBD_UVC_WriteEx()

Description

Writes frame data to the host using single packets.

Prototype

```
int USBD_UVC_WriteEx(U8          * pData,
                    unsigned NumBytes,
                    U8          UserFlags);
```

Parameters

Parameter	Description
<code>pData</code>	Pointer to a buffer containing the frame data. The buffer must provide <code>USB_D_UVC_PAYLOAD_HEADER_SIZE</code> bytes space at the start of the buffer.
<code>NumBytes</code>	Size of the buffer. Must only contain one packet, up to a maximum size of <code>USB_HS_ISO_HB_MAX_PACKET_SIZE</code> .
<code>UserFlags</code>	Flags to be added to the frame. Following flags are currently supported: <ul style="list-style-type: none"> <code>USB_D_UVC_END_OF_FRAME</code> - Should be set with the last <code>USB_D_UVC_Write()</code> call for a single frame.

Return value

0 - All data written to the buffer. -1 - An error occurred (device disconnected).

Additional information

This version of the write routine is optimized to be used with DMA capable targets. But can also speed up transfers with regular drivers. This write routine does not copy the UVC packet data internally, but sends it from the user buffer directly. When using DMA and cache the buffer should be aligned to a cache line boundary. The buffer must provide space (`USB_D_UVC_PAYLOAD_HEADER_SIZE` bytes) at the start of the buffer for the UVC module to insert the UVC packet header. The application must insert the video data after the header. The UVC module will automatically insert the correct header information and send the whole buffer to the host.

18.3.1.4 USBD_UVC_SetOnResolutionChange()

Description

Allows to set a callback which is called when the host changes the resolution of the UVC frame. The callback receives a frame index, which is a direct mapping of the aResolutions array in USBD_UVC_INIT_DATA.

Prototype

```
void USBD_UVC_SetOnResolutionChange(USB_UVC_ON_RESOLUTION_CHANGE * pfOnResChange);
```

Parameters

Parameter	Description
pfOnResChange	User callback of type USB_UVC_ON_RESOLUTION_CHANGE .

18.3.2 Data structures

18.3.2.1 USBD_UVC_INIT_DATA

Description

Initialization data for UVC interface.

Type definition

```
typedef struct {
    U8                EPIn;
    USBD_UVC_BUFFER  * pBuf;
    const USBD_UVC_RESOLUTION * aResolutions;
    U8                NumResolutions;
    U8                StillCaptureMethod;
    U8                Flags;
    USBD_UVC_CONTROLS * Controls;
} USBD_UVC_INIT_DATA;
```

Structure members

Member	Description
EPIn	Isochronous IN endpoint for sending data to the host.
pBuf	Pointer to a USBD_UVC_BUFFER structure.
aResolutions	Pointer to an array of USBD_UVC_RESOLUTION structures.
NumResolutions	Number of elements inside the aResolutions array.
StillCaptureMethod	Method of "still image capture" to use. Valid values: <ul style="list-style-type: none"> • 1 - The host software will extract the next available video frame. (default) • 2 - When the host requests a still image a callback will be called which has to provide a new (still) image frame. It only makes sense to use this method if your data source is able to provide better quality still images than the default quality of the video stream.
Flags	Various flags. Valid bits: <ul style="list-style-type: none"> • USBD_UVC_USE_BULK_MODE - In this mode UVC uses bulk endpoints instead of isochronous endpoints.
Controls	Pointer to a structure of type USBD_UVC_CONTROLS . The structure memory must remain available to the UVC class.

Additional information

This structure holds the endpoint that should be used by the UVC interface ([EPIn](#)). Refer to [USBD_AddEPEx\(\)](#) for more information about how to add an endpoint.

18.3.2.2 USBD_UVC_BUFFER

Description

Structure which contains information about the UVC ring buffer.

Type definition

```
typedef struct {
    USBD_UVC_DATA_BUFFER Buf[];
    volatile U8 NumBlocksIn;
    U8 RdPos;
    U8 WrPos;
    U8 Flags;
} USBD_UVC_BUFFER;
```

Structure members

Member	Description
Buf	Array of <code>USBD_UVC_DATA_BUFFER</code> elements.
NumBlocksIn	Number of currently used buffers.
RdPos	Buffer read position.
WrPos	Buffer write position.
Flags	Used by the UVC module automatically. Do not modify. 1 - WriteEx used.

Additional information

The number of buffers can be set with the `USBD_UVC_NUM_BUFFERS` define. Generally the user does not have to interact with this structure, but he has to provide the memory for it. When `USBD_UVC_USE_BULK_MODE` is used `USBD_UVC_NUM_BUFFERS` can be reduced to 1.

18.3.2.3 USBD_UVC_DATA_BUFFER

Description

Structure which contains values for a single buffer.

Type definition

```
typedef struct {
    U8      * pData;
    unsigned NumBytesIn;
    U8      Flags;
    U8      FrameID;
} USBD_UVC_DATA_BUFFER;
```

Structure members

Member	Description
<code>pData</code>	Pointer to a data buffer. When <code>USB_D_UVC_Write()</code> is used the user must set this pointer to a valid buffer of size <code>USB_D_UVC_DATA_BUFFER_SIZE</code> . When <code>USB_D_UVC_WriteEx()</code> is used the user must not modify this value.
<code>NumBytesIn</code>	Size of the packet.
<code>Flags</code>	<code>Flags</code> which will be sent with the packet.
<code>FrameID</code>	ID of the frame.

Additional information

The size of the buffers can be set with the `USB_D_UVC_DATA_BUFFER_SIZE` define. Ideally it should match the `MaxPacketSize` for the isochronous endpoint.

18.3.2.4 USBD_UVC_RESOLUTION

Description

Structure describing a valid image resolution.

Type definition

```
typedef struct {  
    unsigned Width;  
    unsigned Height;  
} USBD_UVC_RESOLUTION;
```

Structure members

Member	Description
Width	Width in pixels.
Height	Height in pixels.

18.3.3 Function prototypes

18.3.3.1 USB_UVC_ON_RESOLUTION_CHANGE

Description

Callback function description which is set via `USBD_UVC_SetOnResolutionChange()`.

Type definition

```
typedef void USB_UVC_ON_RESOLUTION_CHANGE(unsigned FrameIndex);
```

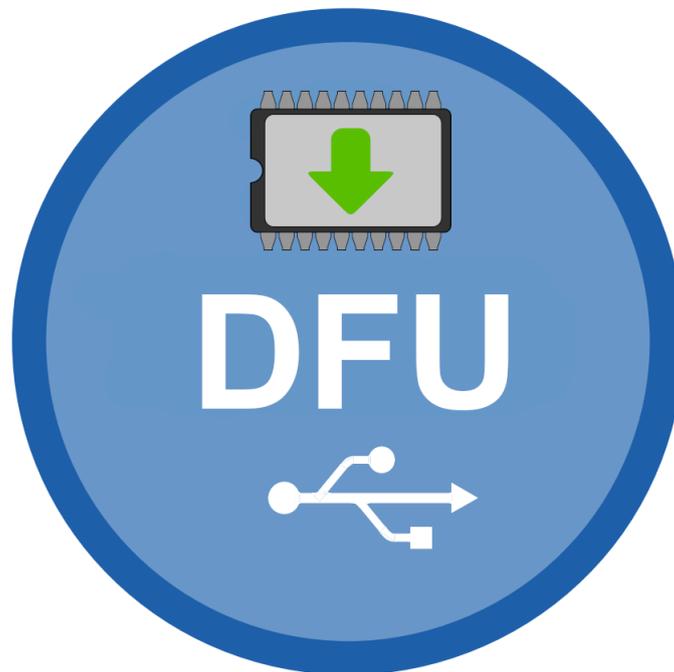
Parameters

Parameter	Description
<code>FrameIndex</code>	1-based index of the frame resolution.

Chapter 19

Device Firmware Upgrade (DFU)

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



19.1 Overview

The Device Firmware Upgrade class (DFU) is a USB class protocol which can be used to download and upload firmware images to and from a device.

emUSB-Device-DFU comes as a complete package and contains the following:

- Generic USB handling
- USB DFU class implementation (version 1.1)
- Sample application showing how to work with DFU

DFU is supported on most operating systems by common tools like `dfu-util`, see dfu-util.sourceforge.net.

19.1.1 Using DFU on Windows

In order to get emUSB DFU running with the WinUSB driver the function `USBD_DFU_SetMSDescInfo()` must be called in the target application.

Microsoft's Windows operating systems (Starting with XP Service Pack 2) contains a generic driver called WinUSB.sys that is used to handle all communication to a emUSB-Device running a DFU interface. If such device is connected to a Windows 8, 8.1 and 10 PC for the first time, Windows will install the WinUSB driver automatically. For Windows versions less than Windows 8, Microsoft provides a driver for Windows Vista and Windows 7 but this needs to be installed manually. A driver installation tool including the mentioned driver is available in the `Windows\USB\Bulk\WinUSBInstall`. Windows XP user can use the driver package located under `Windows\USB\Bulk\WinUSB_USBBulk_XP`.

Additionally the correct driver may not be loaded on Windows 7 systems because of an issue in the USB 3.0 stack of Windows 7, see *Issues on Windows 7* on page 644.

19.2 Configuration

19.2.1 Dual configuration mode

Typically a device that supports DFU has to provide two different configurations. It starts up in runtime mode with the DFU interface and other interfaces used for normal operation of the device. In this configuration the device does not allow download or upload of firmware files.

If the host sends a DFU detach request, the device has to reconfigure to DFU mode, usually providing only a single DFU interface. DFU mode may for example be implemented by running a bootloader build into the device. In order to switch to DFU mode after receiving the detach request from the host, the device has to shutdown and de-initialize the USB stack and start-up it again using the DFU configuration. Then firmware files can be downloaded.

This procedure is compliant to the USB Device Firmware Upgrade class specification. See sample application `USB_DFU_Start.c`.

19.2.2 Single configuration

However, some DFU capable devices do not offer two configurations. Instead they are always in runtime mode, but allow up- and download of firmware files without changing the configuration. emUSB-Device DFU class also supports this behavior which is called 'Mixed Mode', see sample application `USB_DFU_MixedMode_Start.c`.

19.3 Target API

Function	Description
API functions	
<code>USBD_DFU_Add()</code>	Adds a DFU class interface.
<code>USBD_DFU_Add_RunTime()</code>	Adds a DFU class interface for runtime mode only.
<code>USBD_DFU_AddAlternateInterface()</code>	Adds an alternative interface to the DFU configuration.
<code>USBD_DFU_SetMSDescInfo()</code>	Enables use of Microsoft OS Descriptors.
<code>USBD_DFU_SetPollTimeout()</code>	Set the poll timeout to be reported to the host on the next <code>GET_STATUS</code> setup request.
<code>USBD_DFU_Ack()</code>	Acknowledge download data received via the <code>USBD_DFU_DOWNLOAD</code> function.
<code>USBD_DFU_SetError()</code>	Signal an error to the host.
<code>USBD_DFU_ManifestComplt()</code>	Must be called by the application after the new firmware was installed successfully.
<code>USBD_DFU_GetStatusReqCnt()</code>	Return the number of times, the host has requested a status after calling one of the functions <code>USBD_DFU_Ack()</code> , <code>USBD_DFU_ManifestComplt()</code> or <code>USBD_DFU_GetStatusReqCnt()</code> .
<code>USBD_DFU_GetAlternateSetting()</code>	Returns the alternate interface setting that was set by the host.
Data structures	
<code>USB_DFU_INIT_DATA</code>	Initialization data for the DFU interface.
Function prototypes	
<code>USBD_DFU_DETACH_REQUEST</code>	Callback function is called when the host requests a <code>DETACH</code> , prompting the device to enter DFU mode.
<code>USBD_DFU_DOWNLOAD</code>	Callback function to handle download data to the application that was received from the host.
<code>USBD_DFU_UPLOAD</code>	Callback function to get upload data to be transferred to the host.

19.3.1 API functions

19.3.1.1 USBD_DFU_Add()

Description

Adds a DFU class interface.

Prototype

```
void USBD_DFU_Add(const USB_DFU_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
<code>pInitData</code>	Pointer to USB_DFU_INIT_DATA structure.

19.3.1.2 USBD_DFU_Add_RunTime()

Description

Adds a DFU class interface for runtime mode only. Using this function results in a smaller footprint than `USBD_DFU_Add()`.

Prototype

```
void USBD_DFU_Add_RunTime(const USB_DFU_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
<code>pInitData</code>	Pointer to <code>USB_DFU_INIT_DATA</code> structure. The field 'Mode' must be set to <code>USB_DFU_MODE_RUNTIME</code> .

19.3.1.3 USBD_DFU_AddAlternateInterface()

Description

Adds an alternative interface to the DFU configuration. Must be called after `USB_DFU_Add()` and before `USB_Start()`. This function must not be called in runtime only mode.

Prototype

```
void USBD_DFU_AddAlternateInterface(const char * pInterfaceName);
```

Parameters

Parameter	Description
<code>pInterfaceName</code>	Pointer to a string containing the name of the alternate interface. The pointer must remain valid during all USB operations (until <code>USB_DeInit()</code> is called).

19.3.1.4 USBD_DFU_SetMSDescInfo()

Description

Enables use of Microsoft OS Descriptors. A USB DFU device providing these descriptors is detected by Windows to be handled by the generic WinUSB driver.

Prototype

```
void USBD_DFU_SetMSDescInfo(void);
```

Additional information

This function must be called after the call to the function `USB_DFU_Add()` and before `USB-D_Start()`.

19.3.1.5 USBD_DFU_SetPollTimeout()

Description

Set the poll timeout to be reported to the host on the next GET_STATUS setup request.

Prototype

```
void USBD_DFU_SetPollTimeout(U32 PollTimeout);
```

Parameters

Parameter	Description
PollTimeout	Poll timeout in milliseconds.

19.3.1.6 USBD_DFU_Ack()

Description

Acknowledge download data received via the `USBD_DFU_DOWNLOAD` function.

Prototype

```
void USBD_DFU_Ack(void);
```

19.3.1.7 USBD_DFU_SetError()

Description

Signal an error to the host. Can be called by the application at any time. The device will respond with that error on the next 'Get Status' request from the host.

Prototype

```
void USBD_DFU_SetError(USB_DFU_ERROR_STATE Err);
```

Parameters

Parameter	Description
<code>Err</code>	Error code.

19.3.1.8 USBD_DFU_ManifestComplt()

Description

Must be called by the application after the new firmware was installed successfully.

Prototype

```
void USBD_DFU_ManifestComplt(void);
```

19.3.1.9 USBD_DFU_GetStatusReqCnt()

Description

Return the number of times, the host has requested a status after calling one of the functions `USB_DFU_Ack()`, `USB_DFU_ManifestComplt()` or `USB_DFU_GetStatusReqCnt()`.

Prototype

```
unsigned USBD_DFU_GetStatusReqCnt(void);
```

19.3.1.10 USBD_DFU_GetAlternateSetting()

Description

Returns the alternate interface setting that was set by the host.

Prototype

```
unsigned USBD_DFU_GetAlternateSetting(void);
```

Return value

- 0 - No alternate interface was selected by the host.
- 1..n - Number of alternate interface selected by the host.

19.3.2 Data structures

19.3.2.1 USB_DFU_INIT_DATA

Description

Initialization data for the DFU interface.

Type definition

```
typedef struct {
    I8          Mode;
    U8          Attributes;
    U16         DetachTimeout;
    U16         TransferSize;
    U16         Flags;
    const char  * pInterfaceName;
    USBD_DFU_DETACH_REQUEST * pfDetachRequest;
    USBD_DFU_DOWNLOAD * pfDownload;
    U8          * pBuffer;
    USBD_DFU_UPLOAD * pfUpload;
} USB_DFU_INIT_DATA;
```

Structure members

Member	Description
<code>Mode</code>	Operation mode of the DFU interface: <code>USB_DFU_MODE_RUNTIME</code> : The interface is in runtime mode only. Download of firmware data is not supported. <code>USB_DFU_MODE_DFU</code> : The interface is in DFU mode. <code>USB_DFU_MODE_MIXED</code> : The interface is in runtime mode but allows download of firmware data in this mode.
<code>Attributes</code>	Bit mask containing the DFU attributes. Combination of the <code>USB_DFU_ATTR_...</code> flags.
<code>DetachTimeout</code>	Time, in milliseconds, that the device will wait after receipt of the <code>DFU_DETACH</code> request.
<code>TransferSize</code>	Maximum number of bytes that the device can accept per control-write transaction.
<code>Flags</code>	RFU. Must be 0.
<code>pInterfaceName</code>	Name of the interface. Optional, may be <code>NULL</code> .
<code>pfDetachRequest</code>	Pointer to the callback function to request a detach. Used for <code>Mode = USB_DFU_MODE_RUNTIME</code> only.
<code>pfDownload</code>	Pointer to the callback function to receive download data. Used for <code>Mode ≠ USB_DFU_MODE_RUNTIME</code> only.
<code>pBuffer</code>	Pointer to a buffer to store download data. The size of the buffer must be ' <code>TransferSize</code> ' bytes.
<code>pfUpload</code>	Pointer to the callback function to get upload data. Optional. Used for <code>Mode ≠ USB_DFU_MODE_RUNTIME</code> only.

19.3.3 Function prototypes

19.3.3.1 USBD_DFU_DETACH_REQUEST

Description

Callback function is called when the host requests a DETACH, prompting the device to enter DFU mode. This function is executed in interrupt context. The detach and/or reinitialization must not be performed inside this function. Instead this function should only trigger a task to perform the required operation.

Type definition

```
typedef void USBD_DFU_DETACH_REQUEST(U16 Timeout);
```

Parameters

Parameter	Description
Timeout	Timeout provided by the host.

19.3.3.2 USBD_DFU_DOWNLOAD

Description

Callback function to handle download data to the application that was received from the host. The function is called in interrupt context and should return as fast as possible. Especially flash programming must not be done within this function. If `NumBytes` ≥ 0 , the application must respond either with a call to `USBD_DFU_Ack()` if the data could be processed successfully or by calling `USBD_DFU_SetError()` if an error occurred. These functions need not to be called from the `USBD_DFU_DOWNLOAD` function, but may be called later after processing the data. The host will wait for either `USBD_DFU_Ack()` or `USBD_DFU_SetError()` before starting another download.

Type definition

```
typedef void USBD_DFU_DOWNLOAD(int NumBytes,  
                               U16 BlockNum);
```

Parameters

Parameter	Description
<code>NumBytes</code>	Number of bytes received from the host. The data is stored in the buffer provided by <code>USB_DFU_INIT_DATA.pBuffer</code> . A value of 0 indicates the end of the data to be downloaded. A negative value means that the host has aborted the download.
<code>BlockNum</code>	Block sequence number provided by the host.

19.3.3.3 USBD_DFU_UPLOAD

Description

Callback function to get upload data to be transferred to the host. The function is called in interrupt context and should return as fast as possible.

Type definition

```
typedef int USBD_DFU_UPLOAD(
    int    bStart,
    U16    BlockNum,
    U16    NumBytes,
    const U8 ** ppData);
```

Parameters

Parameter	Description
<code>bStart</code>	1 = Start upload, 0 = continue upload.
<code>BlockNum</code>	Block sequence number provided by the host.
<code>NumBytes</code>	Number of bytes requested by the host.
<code>ppData</code>	out Pointer to the data to be transferred to the host.

Return value

Size of the data provided by the function (in bytes). A value < `NumBytes` (including 0) indicate the last part of the data. A negative value indicates an error. In case of an error, the function should also call `USB_DFU_SetError()`.

Chapter 20

Musical Instrument Digital Interface (MIDI)

This chapter gives a general overview of the Musical Instrument Digital Interface class and describes how to get the MIDI component running on the target.



20.1 Overview

The USB MIDI device class is a subclass of the USB audio class. Despite being a subclass of the audio class the protocol is almost entirely different. The MIDI class is able to transfer MIDI commands and MIDI data from a device to a host and vice versa.

MIDI is supported by most operating systems out of the box and the installation of additional drivers is not required.

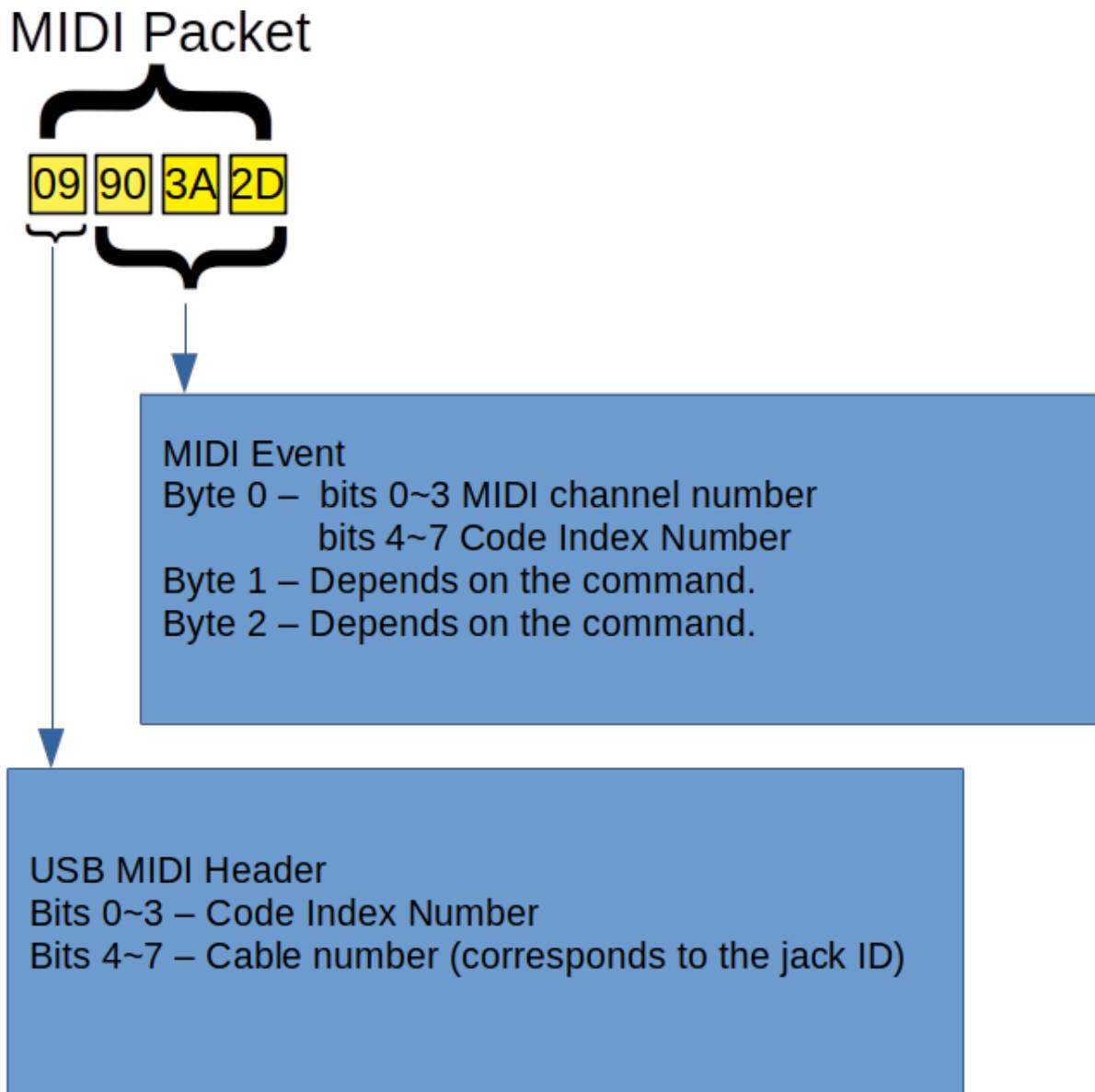
emUSB-Device-MIDI comes as a complete package and contains the following:

- Generic USB handling
- USB MIDI V1.0 device class implementation
- Sample application showing how to work with MIDI

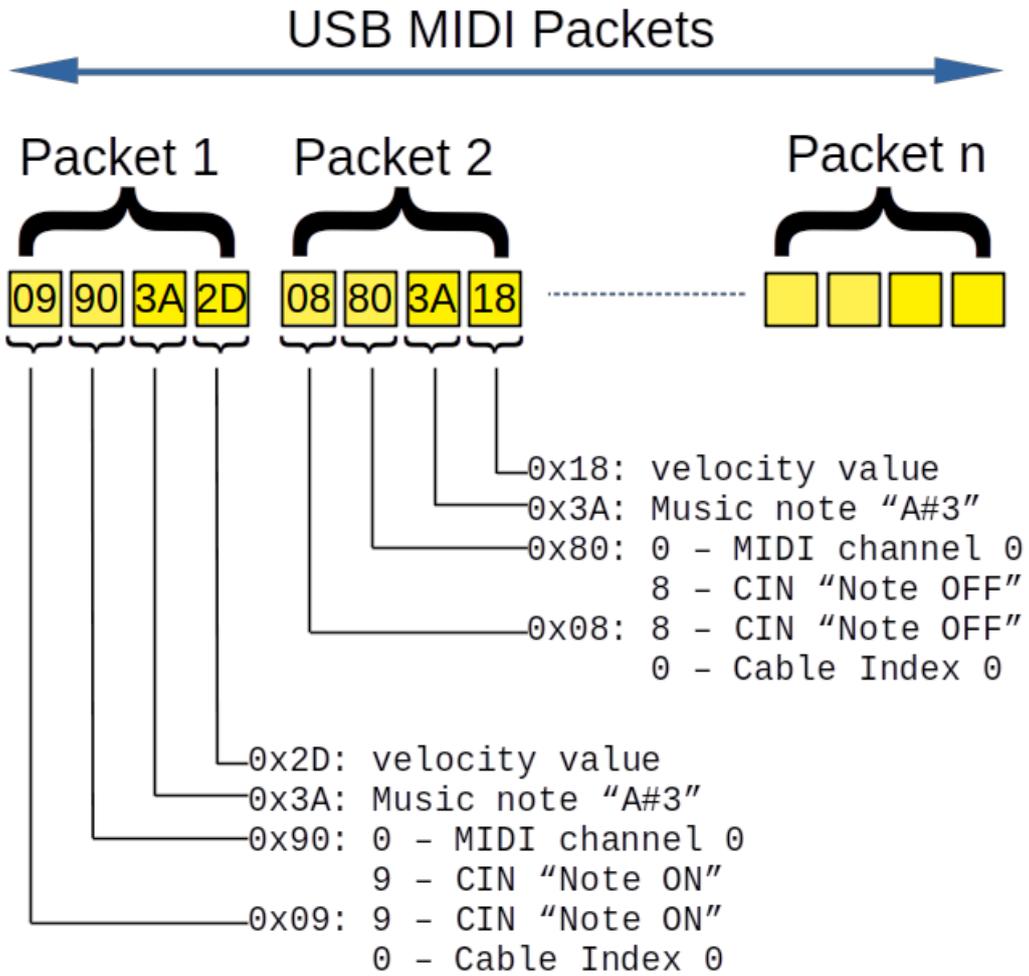
20.2 Introduction

SEGGER's implementation of the MIDI class V1.0 is designed with minimal resource usage in mind, especially targeted to embedded devices. The implementation supports an arbitrary jack configuration as well as writing USB MIDI packets from a MIDI stream and receiving USB MIDI from a host.

The following graphic describes the basic structure of a USB MIDI packet.



Below the MIDI commands "note ON" and "note OFF" can be seen in their decoded form.



20.3 Configuration

20.3.1 Initial configuration

To get emUSB-Device-MIDI up and running as well as doing an initial test, the configuration as delivered with the sample application should not be modified.

20.3.2 Final configuration

The configuration must only be modified when emUSB-Device is integrated in your final product. Refer to section *emUSB-Device Configuration* on page 48 for detailed information about the generic information functions which have to be adapted.

20.3.3 Testing MIDI on different operating systems

Linux

Install the third-party amidi command-line utility.

- Connect the device and call `amidi -l`, you should see a list of connected MIDI devices:

```
Dir Device      Name
IO  hw:2,0,0    MIDI device MIDI 1
```

- Using the device name you can either send data to the device or receive data from the device.

Sending:

```
amidi -p hw:2,0,0 -S '90 4E 30 80 4E 30' // 90 - Note on
                                           // 4E - Note "F#5"
                                           // 30 - Velocity value
                                           // 80 - Note off
                                           // 4E - Note "F#5"
                                           // 30 - Velocity value
```

Receiving:

```
amidi -p hw:2,0,0 -d
```

Windows

On Windows a third-party utility such as [MIDI-OX](#) can be used to monitor MIDI events.

Mac

On macOS a third-party utility such as [Snoize MIDI Monitor](#) can be used to monitor MIDI events.

20.4 Target API

Function	Description
API functions	
<code>USBD_MIDI_Init()</code>	Initialize the MIDI component.
<code>USBD_MIDI_Add()</code>	Adds a MIDI class interface to the USB stack.
<code>USBD_MIDI_ReceivePackets()</code>	Receives USB MIDI packets from the host.
<code>USBD_MIDI_GetNumPacketsInBuffer()</code>	Returns the number of MIDI packets that are available in the internal OUT endpoint buffer.
<code>USBD_MIDI_ConvertPackets()</code>	Converts USB MIDI packets to pure MIDI commands by stripping the USB header.
<code>USBD_MIDI_WritePackets()</code>	Writes USB MIDI packets to the host.
<code>USBD_MIDI_WriteStream()</code>	Sends MIDI data to the USB host.
Data structures	
<code>USBD_MIDI_INIT_DATA</code>	Initialization structure that is needed when adding a MIDI interface to emUSB-Device.
<code>USBD_MIDI_JACK</code>	Structure describing a MIDI IN or OUT jack.
<code>USBD_MIDI_PACKET</code>	Structure describing a MIDI packet.

20.4.1 API functions

20.4.1.1 USBD_MIDI_Init()

Description

Initialize the MIDI component.

Prototype

```
void USBD_MIDI_Init(void);
```

20.4.1.2 USBD_MIDI_Add()

Description

Adds a MIDI class interface to the USB stack.

Prototype

```
USBD_MIDI_HANDLE USBD_MIDI_Add(const USBD_MIDI_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
<code>pInitData</code>	Pointer to USBD_MIDI_INIT_DATA structure.

Return value

Handle to a valid MIDI instance. The handle of the first MIDI instance is always 0.

20.4.1.3 USBD_MIDI_ReceivePackets()

Description

Receives USB MIDI packets from the host. The function blocks until any data has been received or a timeout occurs (if `Timeout` \geq 0). In case of a timeout, the read transfer is aborted.

Prototype

```
int USBD_MIDI_ReceivePackets(USB_D_MIDI_HANDLE hInst,
                             USB_D_MIDI_PACKET * paPacket,
                             unsigned NumPackets,
                             int Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid MIDI instance, returned by <code>USB_D_MIDI_Add()</code> .
<code>paPacket</code>	Pointer to an array of <code>USB_D_MIDI_PACKET</code> structures.
<code>NumPackets</code>	Number of packets inside the <code>paPacket</code> array.
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite. If <code>Timeout</code> is -1, the function never blocks and only reads data from the internal endpoint buffer.

Return value

- > 0 Number of MIDI packets read.
- = 0 A timeout occurred (if `Timeout` > 0), no data in buffer (if `Timeout` < 0) or the target was disconnected during the function call and no data was read so far.
- < 0 Error occurred.

Additional information

This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return `USB_STATUS_ERROR`.

Periodically calling this function with `timeout = -1` can be used to poll for data.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USB_D_AddEP()`. This data can be retrieved by a later call to `USB_D_MIDI_ReceivePackets()`. See also `USB_D_MIDI_GetNumPacketsInBuffer()`.

20.4.1.4 USBD_MIDI_GetNumPacketsInBuffer()

Description

Returns the number of MIDI packets that are available in the internal OUT endpoint buffer.

Prototype

```
unsigned USBD_MIDI_GetNumPacketsInBuffer(USB_D_MIDI_HANDLE hInst);
```

Parameters

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

Return value

Number of packets that are available in the internal OUT endpoint buffer.

Additional information

If the host is sending more data than your target application has requested, the remaining data will be stored in an internal buffer. This function shows how many bytes are available in this buffer.

The number of packets returned by this function can be read using `USB_D_MIDI_ReceivePackets()`.

20.4.1.5 USBD_MIDI_ConvertPackets()

Description

Converts USB MIDI packets to pure MIDI commands by stripping the USB header. USB MIDI packets are usually provided through the use of the `USB_D_MIDI_ReceivePackets()` function.

Prototype

```
int USBD_MIDI_ConvertPackets(const USBD_MIDI_PACKET * paPacket,
                           unsigned NumPackets,
                           U8 * pBuf);
```

Parameters

Parameter	Description
<code>paPacket</code>	Pointer to an array of <code>USB_D_MIDI_PACKET</code> structures.
<code>NumPackets</code>	Number of packets inside the <code>paPacket</code> array.
<code>pBuf</code>	Buffer to write the MIDI commands into. The buffer must be $3 * \text{NumPackets}$ bytes large.

Return value

≥ 0 Number of MIDI packets converted.
 < 0 Error occurred.

Additional information

The jack ID is lost in this conversion. When using multiple jacks the user should take care not to mix packets addressed to different jacks.

20.4.1.6 USBD_MIDI_WritePackets()

Description

Writes USB MIDI packets to the host. Unlike with `USBD_MIDI_WriteStream()` the user must set the correct values for the USB MIDI header (CIN and jack ID) as these are not automatically filled in.

Prototype

```
int USBD_MIDI_WritePackets(    USBD_MIDI_HANDLE    hInst,
                              const USBD_MIDI_PACKET * paPacket,
                              unsigned          NumPackets,
                              int               Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid MIDI instance, returned by <code>USBD_MIDI_Add()</code> .
<code>paPacket</code>	Pointer to an array of <code>USBD_MIDI_PACKET</code> structures. The user must fill all fields. For MIDI events which do not use all 3 MIDI bytes the user must fill the unused bytes with zeroes.
<code>NumPackets</code>	Number of packets inside the <code>paPacket</code> array.
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite.

Return value

> 0 Number of written USB MIDI packets.
 = 0 A timeout occurred (if `Timeout > 0`).
 < 0 Error occurred.

Additional information

This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return `USB_STATUS_ERROR`.

20.4.1.7 USBD_MIDI_WriteStream()

Description

Sends MIDI data to the USB host. This function accepts a stream of MIDI commands and automatically adds the necessary USB MIDI header byte. Depending on the `Timeout` parameter, the function may block until `NumBytes` have been written or a timeout occurs.

Prototype

```
int USBD_MIDI_WriteStream(    USBD_MIDI_HANDLE  hInst,
                             U8                        JackID,
                             const U8                  * pData,
                             unsigned                  NumBytes,
                             int                       Timeout);
```

Parameters

Parameter	Description
<code>hInst</code>	Handle to a valid MIDI instance, returned by <code>USB_MIDI_Add()</code> .
<code>JackID</code>	Jack ID to use.
<code>pData</code>	Data that should be written.
<code>NumBytes</code>	Number of bytes to write.
<code>Timeout</code>	<code>Timeout</code> in milliseconds. 0 means infinite.

Return value

= 0 `Timeout` has occurred and no data was written.
 > 0 && < `NumBytes` Number of bytes that have been written before a timeout occurred.
 = `NumBytes` Write transfer successful completed.
 < 0 Error occurred.

Additional information

This function also returns when the target is disconnected from host or when a USB reset occurred.

20.4.2 Data structures

20.4.2.1 USBD_MIDI_INIT_DATA

Description

Initialization structure that is needed when adding a MIDI interface to emUSB-Device.

Type definition

```
typedef struct {
    U16          Flags;
    U8           EPIn;
    U8           EPOut;
    const USBD_MIDI_JACK * paJackList;
    unsigned     NumJacks;
} USBD_MIDI_INIT_DATA;
```

Structure members

Member	Description
Flags	Reserved for future use, must be 0.
EPIn	Bulk IN endpoint for sending data to the host.
EPOut	Bulk OUT endpoint for receiving data from the host.
paJackList	Pointer to an array containing all jacks for the MIDI interface.
NumJacks	Number of elements inside the paJackList array.

20.4.2.2 USBD_MIDI_JACK

Description

Structure describing a MIDI IN or OUT jack.

Type definition

```
typedef struct {
    U8          JackType;
    U8          JackID;
    U8          JackDir;
    U8          NrInputPins;
    U8          * paSourceID;
    U8          * paSourcePin;
    const char * pJackName;
} USBD_MIDI_JACK;
```

Structure members

Member	Description
JackType	USB_AUDIO_MIDI_EMBEDDED_JACK or USB_AUDIO_MIDI_EXTERNAL_JACK
JackID	Unique ID for the jack. Must not be zero.
JackDir	Jack direction, USB_AUDIO_MIDI_IN_JACK or USB_AUDIO_MIDI_OUT_JACK
NrInputPins	For IN jacks - set to zero. For OUT jacks - number of input pins for this MIDI OUT jack.
paSourceID	Only for OUT jacks. Pointer to an array containing the IDs of the entities to which the pin of this MIDI OUT Jack is connected.
paSourcePin	Only for OUT jacks. Pointer to an array containing the output pin numbers of the entities to which the input pins of this MIDI OUT Jack are connected.
pJackName	String describing the jack. Can be NULL.

20.4.2.3 USBD_MIDI_PACKET

Description

Structure describing a MIDI packet.

Type definition

```
typedef struct {
    U8  CableNumber_and_CIN;
    U8  MIDI_0;
    U8  MIDI_1;
    U8  MIDI_2;
} USBD_MIDI_PACKET;
```

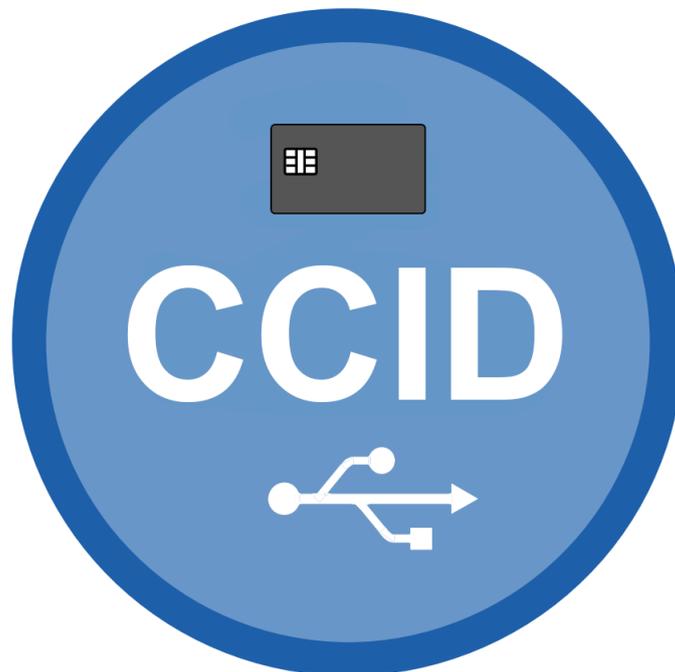
Structure members

Member	Description
CableNumber_and_CIN	b0-b3 - Code Index Number (CIN). b4-b7 - Cable Number (embedded MIDI Jack ID).
MIDI_0	First MIDI byte: b0-b3 - For CIN < 0xF: MIDI channel number. b4-b7 - Code Index Number (same as in b0-b3 of CableNumber_and_CIN).
MIDI_1	Second MIDI byte - Content depends on CIN.
MIDI_2	Third MIDI byte - Content depends on CIN.

Chapter 21

Smart Card Device Class (CCID)

This chapter gives a general overview of the CCID class.



21.1 Overview

The Smart Card Device Class (CCID) allows the implementation of CCID compatible smart card readers. The Integrated Circuit(s) Cards Interface is an abstract USB class protocol defined by the USB Implementers Forum.

The emUSB CCID class only handles the transport of CCID commands via USB. Processing of the smart card commands, including physical access to a smart card (if any) has to be done by the application.

A typical application will contain a loop, that

- Reads a CCID command from the host using `USBD_CCID_ReceiveCmd()`.
- Processes the command depending on the message type and parameters.
- Sends an answer back to the host using one of the `USBD_CCID_Send...()` functions.

emUSB-Device CCID comes as a complete package and contains the following:

- Generic USB handling
- USB CCID class implementation (version 1.1)
- Sample application showing how to implement a simple card reader

21.2 Target API

Function	Description
API functions	
<code>USBD_CCID_Init()</code>	Initialize the CCID component.
<code>USBD_CCID_Add()</code>	Adds interface for USB-CCID communication to emUSB-Device.
<code>USBD_CCID_ReceiveCmd()</code>	Receive a CCID command from the host.
<code>USBD_CCID_SendStatus()</code>	Send a response to a CCID command with message types <code>USB_CCID_MSG_ICC_POWER_OFF</code> , <code>USB_CCID_MSG_GET_SLOT_STATUS</code> , <code>USB_CCID_MSG_ICC_CLOCK</code> , <code>USB_CCID_MSG_T0APDU</code> or <code>USB_CCID_MSG_MECHANICAL</code> .
<code>USBD_CCID_SendDataBlock()</code>	Send a response to a CCID command with message types <code>USB_CCID_MSG_XFR_BLOCK</code> , <code>USB_CCID_MSG_ICC_POWER_ON</code> , or <code>USB_CCID_MSG_SECURE</code> .
<code>USBD_CCID_SendEscape()</code>	Send a response to a CCID command with message type <code>USB_CCID_MSG_ESCAPE_CMD</code> .
<code>USBD_CCID_SendParameters()</code>	Send a response to a CCID command with message type <code>USB_CCID_MSG_SET_RATE_AND_CLOCK</code> .
<code>USBD_CCID_SendDataRateAndClockFrequency()</code>	Send a response to a CCID command with message type <code>USB_CCID_MSG_SET_RATE_AND_CLOCK</code> .
<code>USBD_CCID_NotifySlotState()</code>	Send a notification about a new slot state to the host (via interrupt EP).
<code>USBD_CCID_NotifyHwError()</code>	Send a notification about a hardware error to the host (via interrupt EP).
Data structures	
<code>USB_CCID_INIT_DATA</code>	Initialization structure that is needed when adding a CCID interface to emUSB-Device.
<code>USB_CCID_PROPERTIES</code>	Declares all properties of a CCID device.
<code>USB_CCID_CMD</code>	Contains information about a CCID command send from the host.
<code>USB_CCID_PROTOCOL_DATA_T0</code>	Protocol parameters for T=0 protocol.
<code>USB_CCID_PROTOCOL_DATA_T1</code>	Protocol parameters for T=1 protocol.
Function prototypes	
<code>USBD_CCID_ABORT_CB</code>	Callback function to forward a CCID abort request from the host to the application.

21.2.1 API functions

21.2.1.1 USBD_CCID_Init()

Description

Initialize the CCID component.

Prototype

```
void USBD_CCID_Init(void);
```

21.2.1.2 USBD_CCID_Add()

Description

Adds interface for USB-CCID communication to emUSB-Device.

Prototype

```
void USBD_CCID_Add(const USB_CCID_INIT_DATA * pInitData,  
                  const USB_CCID_PROPERTIES * pProperties);
```

Parameters

Parameter	Description
<code>pInitData</code>	Pointer to a <code>USB_CCID_INIT_DATA</code> structure.
<code>pProperties</code>	Pointer to a <code>USB_CCID_PROPERTIES</code> containing all properties of the CCID device. The pointer must remain valid during all CCID operations.

21.2.1.3 USBD_CCID_ReceiveCmd()

Description

Receive a CCID command from the host.

Prototype

```
int USBD_CCID_ReceiveCmd(USB_CCID_CMD * pCmd,
                        unsigned      BuffSize,
                        U8             * pBuff,
                        unsigned      Timeout);
```

Parameters

Parameter	Description
<code>pCmd</code>	Pointer to a <code>USB_CCID_CMD</code> structure that will be filled by the function with information about the CCID command received.
<code>BuffSize</code>	Size of the buffer pointed to by <code>pBuff</code> . If a CCID command contains more data bytes than ' <code>BuffSize</code> ', then only ' <code>BuffSize</code> ' bytes are copied to the buffer and the remaining bytes are discarded.
<code>pBuff</code>	Buffer to receive the command data (<code>abData</code>).
<code>Timeout</code>	<code>Timeout</code> in ms to wait for a CCID command from the host.

Return value

> 0 Success.
 = 0 A timeout has occurred.
 < 0 An error occurred.

21.2.1.4 USBD_CCID_SendStatus()

Description

Send a response to a CCID command with message types `USB_CCID_MSG_ICC_POWER_OFF`, `USB_CCID_MSG_GET_SLOT_STATUS`, `USB_CCID_MSG_ICC_CLOCK`, `USB_CCID_MSG_T0APDU` or `USB_CCID_MSG_MECHANICAL`.

Prototype

```
int USBD_CCID_SendStatus(const USB_CCID_CMD * pCmd,
                        U8 Status,
                        U8 Error,
                        U8 ClockStatus);
```

Parameters

Parameter	Description
<code>pCmd</code>	Pointer to a <code>USB_CCID_CMD</code> structure returned from a call to <code>USB_D_CCID_ReceiveCmd()</code> .
<code>Status</code>	<code>Status</code> code. One of the <code>USB_CCID_STATUS_CMD_..</code> macros or'ed together with one of the <code>USB_CCID_STATUS_ICC_..</code> macros.
<code>Error</code>	<code>Error</code> code, if <code>Status</code> bit <code>USB_CCID_STATUS_CMD_FAIL</code> is set.
<code>ClockStatus</code>	<ul style="list-style-type: none"> • 0: Clock running. • 1: Clock stopped in state L. • 2: Clock stopped in state H. • 3: Clock stopped in an unknown state.

Return value

> 0 Success.
 = 0 A timeout has occurred.
 < 0 An error occurred.

21.2.1.5 USBD_CCID_SendDataBlock()

Description

Send a response to a CCID command with message types `USB_CCID_MSG_XFR_BLOCK`, `USB_CCID_MSG_ICC_POWER_ON`, or `USB_CCID_MSG_SECURE`.

Prototype

```
int USBD_CCID_SendDataBlock(const USB_CCID_CMD * pCmd,
                           U8                Status,
                           U8                Error,
                           U8                ChainParameter,
                           unsigned         DataLen,
                           const U8        * pData);
```

Parameters

Parameter	Description
<code>pCmd</code>	Pointer to a <code>USB_CCID_CMD</code> structure returned from a call to <code>USB_D_CCID_ReceiveCmd()</code> .
<code>Status</code>	<code>Status</code> code. One of the <code>USB_CCID_STATUS_CMD_..</code> macros or'ed together with one of the <code>USB_CCID_STATUS_ICC_..</code> macros.
<code>Error</code>	<code>Error</code> code, if <code>Status</code> bit <code>USB_CCID_STATUS_CMD_FAIL</code> is set.
<code>ChainParameter</code>	For extended APDU level, indicates if the response is complete, to be continued or if the command APDU can continue.
<code>DataLen</code>	Number of data bytes to be returned to the host.
<code>pData</code>	Pointer to the data to be returned to the host (<code>abData</code>).

Return value

> 0 Success.
 = 0 A timeout has occurred.
 < 0 An error occurred.

21.2.1.6 USBD_CCID_SendEscape()

Description

Send a response to a CCID command with message type `USB_CCID_MSG_ESCAPE_CMD`.

Prototype

```
int USBD_CCID_SendEscape(const USB_CCID_CMD * pCmd,
                        U8 Status,
                        U8 Error,
                        unsigned DataLen,
                        const U8 * pData);
```

Parameters

Parameter	Description
<code>pCmd</code>	Pointer to a <code>USB_CCID_CMD</code> structure returned from a call to <code>USBD_CCID_ReceiveCmd()</code> .
<code>Status</code>	<code>Status</code> code. One of the <code>USB_CCID_STATUS_CMD_..</code> macros or'ed together with one of the <code>USB_CCID_STATUS_ICC_..</code> macros.
<code>Error</code>	<code>Error</code> code, if <code>Status</code> bit <code>USB_CCID_STATUS_CMD_FAIL</code> is set.
<code>DataLen</code>	Number of data bytes to be returned to the host.
<code>pData</code>	Pointer to the data to be returned to the host (<code>abData</code>).

Return value

- > 0 Success.
- = 0 A timeout has occurred.
- < 0 An error occurred.

21.2.1.7 USBD_CCID_SendParameters()

Description

Send a response to a CCID command with message type `USB_CCID_MSG_SET_RATE_AND_CLOCK`.

Prototype

```
int USBD_CCID_SendParameters(const USB_CCID_CMD * pCmd,
                             U8                Status,
                             U8                Error,
                             U8                ProtocolNum,
                             const void       * pProtocolData);
```

Parameters

Parameter	Description
<code>pCmd</code>	Pointer to a <code>USB_CCID_CMD</code> structure returned from a call to <code>USBD_CCID_ReceiveCmd()</code> .
<code>Status</code>	<code>Status</code> code. One of the <code>USB_CCID_STATUS_CMD_..</code> macros or'ed together with one of the <code>USB_CCID_STATUS_ICC_..</code> macros.
<code>Error</code>	<code>Error</code> code, if <code>Status</code> bit <code>USB_CCID_STATUS_CMD_FAIL</code> is set.
<code>ProtocolNum</code>	<ul style="list-style-type: none"> 0: Protocol data for T=0 1: Protocol data for T=1
<code>pProtocolData</code>	Pointer to the protocol data structure (<code>USB_CCID_PROTOCOL_DATA_T0</code> or <code>USB_CCID_PROTOCOL_DATA_T1</code>)

Return value

> 0 Success.
 = 0 A timeout has occurred.
 < 0 An error occurred.

21.2.1.8 USBD_CCID_SendDataRateAndClockFrequency()

Description

Send a response to a CCID command with message type `USB_CCID_MSG_SET_RATE_AND_CLOCK`.

Prototype

```
int USBD_CCID_SendDataRateAndClockFrequency(const USB_CCID_CMD * pCmd,
                                             U8          Status,
                                             U8          Error,
                                             U32          ClockFrequency,
                                             U32          DataRate);
```

Parameters

Parameter	Description
<code>pCmd</code>	Pointer to a <code>USB_CCID_CMD</code> structure returned from a call to <code>USB_D_CCID_ReceiveCmd()</code> .
<code>Status</code>	<code>Status</code> code. One of the <code>USB_CCID_STATUS_CMD_..</code> macros or'ed together with one of the <code>USB_CCID_STATUS_ICC_..</code> macros.
<code>Error</code>	<code>Error</code> code, if <code>Status</code> bit <code>USB_CCID_STATUS_CMD_FAIL</code> is set.
<code>ClockFrequency</code>	Current setting of the ICC clock frequency in KHz.
<code>DataRate</code>	Current setting of the ICC data rate in bps.

Return value

- > 0 Success.
- = 0 A timeout has occurred.
- < 0 An error occurred.

21.2.1.9 USBD_CCID_NotifySlotState()

Description

Send a notification about a new slot state to the host (via interrupt EP).

Prototype

```
void USBD_CCID_NotifySlotState(unsigned Slot,  
                               unsigned State);
```

Parameters

Parameter	Description
<code>Slot</code>	<code>Slot</code> index (counting from 0).
<code>State</code>	New slot state. <ul style="list-style-type: none">• 0: No ICC present.• 1: ICC Present.

21.2.1.10 USBD_CCID_NotifyHwError()

Description

Send a notification about a hardware error to the host (via interrupt EP).

Prototype

```
void USBD_CCID_NotifyHwError(unsigned Slot,  
                             U8      SeqNum,  
                             U8      ErrorCode);
```

Parameters

Parameter	Description
<code>Slot</code>	<code>Slot</code> index (counting from 0).
<code>SeqNum</code>	Sequence number of bulk out command when the hardware error occurred.
<code>ErrorCode</code>	Error code.

21.2.2 Data structures

21.2.2.1 USB_CCID_INIT_DATA

Description

Initialization structure that is needed when adding a CCID interface to emUSB-Device.

Type definition

```
typedef struct {
    U8          EPIn;
    U8          EPOut;
    U8          EPInt;
    U8          * pBuff;
    USBD_CCID_ABORT_CB * pfAbort;
} USB_CCID_INIT_DATA;
```

Structure members

Member	Description
EPIn	Endpoint for sending data to the host.
EPOut	Endpoint for receiving data from the host.
EPInt	Endpoint for sending notification to the host. Optional, may be 0.
pBuff	Pointer to endpoint buffer for EPIn . Buffer should be able to hold one USB packet.
pfAbort	Callback function to signal an abort by the host. Optional, may be <code>NULL</code> .

21.2.2.2 USB_CCID_PROPERTIES

Description

Declares all properties of a CCID device.

Type definition

```
typedef struct {
    U16      Flags;
    U8      NumSlots;
    U8      VoltageSupport;
    U8      Protocols;
    U8      NumClocks;
    U8      DefaultClockIdx;
    U8      NumDataRates;
    U8      DefaultDataRateIdx;
    const U32 * pClocks;
    const U32 * pDataRates;
    U32      MaxIFSD;
    U32      Features;
    U32      MaxMessageLength;
    U8      ClassGetResponse;
    U8      ClassEnvelope;
    U8      LCDLines;
    U8      LCDColumns;
    U8      PINSupport;
    const char * pInterfaceName;
} USB_CCID_PROPERTIES;
```

Structure members

Member	Description
Flags	Reserved for future use, must be 0.
NumSlots	Number of card slots supported by the device (max. 4).
VoltageSupport	Supported voltages, see <code>USB_CCID_VOLTAGE...</code> macros.
Protocols	Supported protocols (T=0, T=1), see <code>USB_CCID_PROTOCOL...</code> macros.
NumClocks	Number of supported clock rates (number of entries in the table pointed to by <code>pClocks</code>). Must be ≥ 1 .
DefaultClockIdx	Index of the default clock within the table <code>pClocks</code> .
NumDataRates	Number of supported data rates (number of entries in the table pointed to by <code>pDataRates</code>). May be 0 to indicate a range. In this case <code>pDataRates</code> must contain 2 entries (min, max) and <code>DefaultDataRateIdx</code> must be 0.
DefaultDataRateIdx	Index of the default data rate within the table <code>pDataRates</code> .
pClocks	Table of all supported clock rates in KHz in increasing order.
pDataRates	Table of all supported data rates in bps in increasing order.
MaxIFSD	Indicates the maximum IFSD supported by CCID for protocol T=1.
Features	This value indicates what intelligent features the CCID has. The value is a bitwise OR operation performed on the macros <code>USB_CCID_FEATURE...</code>
MaxMessageLength	Maximum CCID message length.
ClassGetResponse	Significant only for CCID that offers an APDU level for exchanges. Indicates the default class value used by the CCID when it sends a Get Response command to perform the

Member	Description
	transportation of an APDU by T=0 protocol. Value <code>0xFF</code> indicates that the CCID echoes the class of the APDU.
ClassEnvelope	Significant only for CCID that offers an extended APDU level for exchanges. Indicates the default class value used by the CCID when it sends an Envelope command to perform the transportation of an extended APDU by T=0 protocol. Value <code>0xFF</code> indicates that the CCID echoes the class of the APDU.
LCDLines	Number of lines of the LCD display. 0 if no display supported.
LCDColumns	Number of characters per line of the LCD display. 0 if no display supported.
PINSupport	This value indicates what PIN support features the CCID has.
pInterfaceName	Name of the interface.

21.2.2.3 USB_CCID_CMD

Description

Contains information about a CCID command send from the host.

Type definition

```
typedef struct {
    U8      MessageType;
    U8      Slot;
    U8      SeqNum;
    U8      PowerSelect;
    U8      BWI;
    U8      LevelParameter;
    U8      ProtocolNumber;
    U8      ClockCommand;
    U8      ClassValid;
    U8      ClassGetResponse;
    U8      ClassEnvelope;
    U8      Function;
    unsigned DataLen;
} USB_CCID_CMD;
```

Structure members

Member	Description
MessageType	Message type, see <code>USB_CCID_MSG_...</code> macros.
Slot	Card slot index (counting from 0).
SeqNum	Command sequence number.
PowerSelect	Only valid for message type <code>USB_CCID_MSG_ICC_POWER_ON</code> . Contains voltage that is applied to the ICC: <ul style="list-style-type: none"> • 0 - Automatic Voltage Selection • 1 - 5.0 volts • 2 - 3.0 volts • 3 - 1.8 volts.
BWI	Only valid for message types <code>USB_CCID_MSG_XFR_BLOCK</code> and <code>USB_CCID_MSG_SECURE</code> . Used to extend the CCIDs Block Waiting Timeout for this current transfer. The CCID shall timeout the block after "this number multiplied by the Block Waiting Time" has expired.
LevelParameter	Only valid for message types <code>USB_CCID_MSG_XFR_BLOCK</code> and <code>USB_CCID_MSG_SECURE</code> . Use changes depending on the exchange level reported by the class descriptor in <code>dwFeatures</code> field: <ul style="list-style-type: none"> • Character level: Size of expected data to be returned by the bulk-IN endpoint. • Extended APDU level: Indicates if APDU begins or ends in this command.
ProtocolNumber	Only valid for message types <code>USB_CCID_MSG_SET_PARAMETERS</code> . <ul style="list-style-type: none"> • 0: Structure for protocol T=0. • 1: Structure for protocol T=1.
ClockCommand	Only valid for message types <code>USB_CCID_MSG_ICC_CLOCK</code> . <ul style="list-style-type: none"> • 0: restarts Clock. • 1: Stops Clock.
ClassValid	Only valid for message types <code>USB_CCID_MSG_TOAPDU</code> . Bit 0 = 1 indicates, that the field ClassGetResponse is valid. Bit 1 = 1 indicates, that the field ClassEnvelope is valid.

Member	Description
ClassGetResponse	Only valid for message types <code>USB_CCID_MSG_T0APDU</code> . Value to force the class byte of the header in a Get Response command. Value = <code>0xFF</code> indicates that the class byte of the Get Response command echoes the class byte of the APDU.
ClassEnvelope	Only valid for message types <code>USB_CCID_MSG_T0APDU</code> . Value to force the class byte of the header in a Envelope command. Value = <code>0xFF</code> indicates that the class byte of the Envelope command echoes the class byte of the APDU.
Function	Only valid for message types <code>USB_CCID_MSG_MECHANICAL</code> . This value corresponds to the mechanical function being requested: <ul style="list-style-type: none"> • 1 - Accept Card. • 2 - Eject Card. • 3 - Capture Card. • 4 - Lock Card. • 5 - Unlock Card.
DataLen	Size of data send with this command.

21.2.2.4 USB_CCID_PROTOCOL_DATA_T0

Description

Protocol parameters for T=0 protocol.

Type definition

```
typedef struct {
    U8  FindexDindex;
    U8  TCCKST0;
    U8  GuardTimeT0;
    U8  WaitingIntegerT0;
    U8  ClockStop;
} USB_CCID_PROTOCOL_DATA_T0;
```

Structure members

Member	Description
FindexDindex	see USB CCID specification.
TCCKST0	see USB CCID specification.
GuardTimeT0	see USB CCID specification.
WaitingIntegerT0	see USB CCID specification.
ClockStop	see USB CCID specification.

21.2.2.5 USB_CCID_PROTOCOL_DATA_T1

Description

Protocol parameters for T=1 protocol.

Type definition

```
typedef struct {
    U8  FindexDindex;
    U8  TCCKST1;
    U8  GuardTimeT1;
    U8  WaitingIntegerT1;
    U8  ClockStop;
    U8  IFSC;
    U8  NadValue;
} USB_CCID_PROTOCOL_DATA_T1;
```

Structure members

Member	Description
FindexDindex	see USB CCID specification.
TCCKST1	see USB CCID specification.
GuardTimeT1	see USB CCID specification.
WaitingIntegerT1	see USB CCID specification.
ClockStop	see USB CCID specification.
IFSC	see USB CCID specification.
NadValue	see USB CCID specification.

21.2.3 Function prototypes

21.2.3.1 USBD_CCID_ABORT_CB

Description

Callback function to forward a CCID abort request from the host to the application. The function is called in interrupt context and should return as fast as possible.

Type definition

```
typedef void USBD_CCID_ABORT_CB(U8 Slot,  
                                U8 SeqNum);
```

Parameters

Parameter	Description
Slot	Card slot index (counting from 0).
SeqNum	Sequence number of the command to abort.

Chapter 22

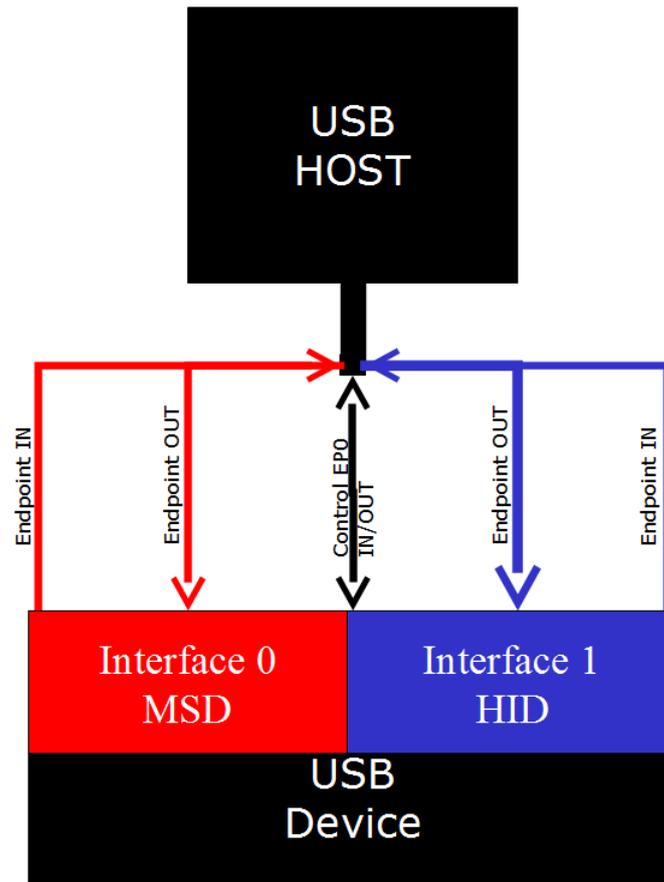
Combining USB components (Multi-Interface)

In some cases, it is necessary to combine different USB components in one device. This chapter will describe how to do this and which steps are necessary.

22.1 Overview

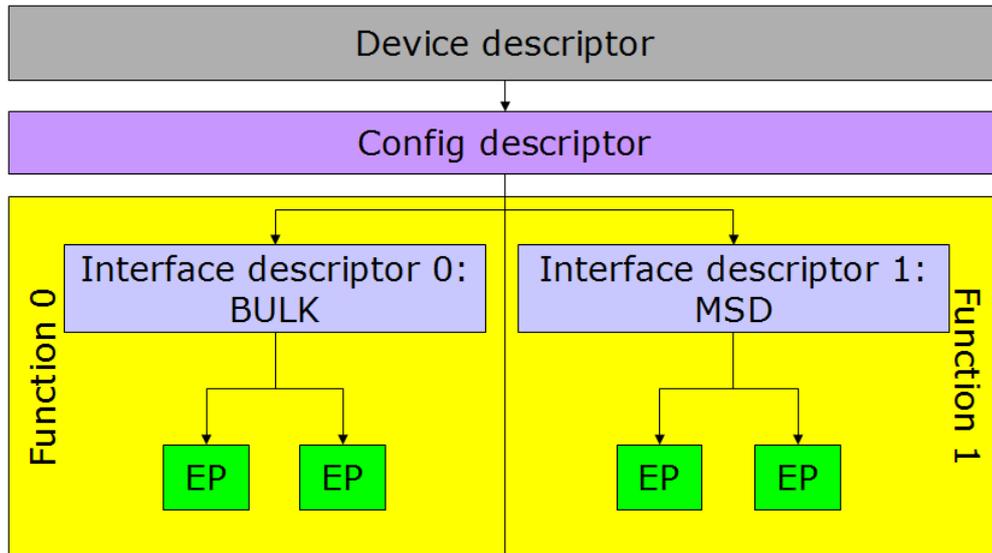
The USB specification allows implementation of more than one component (function) in a single device. This is achieved by combining two or more components. These devices will be recognized by the USB host as composite device and each component will be recognized as an independent device.

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. And the configuration of the data logger can be changed by using a BULK component, CDC component or even HID component.



22.1.1 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 at the interface descriptor level and thus makes it easy to combine different or even the same device classes into one device. Such devices classes are MSD, HID and generic bulk.

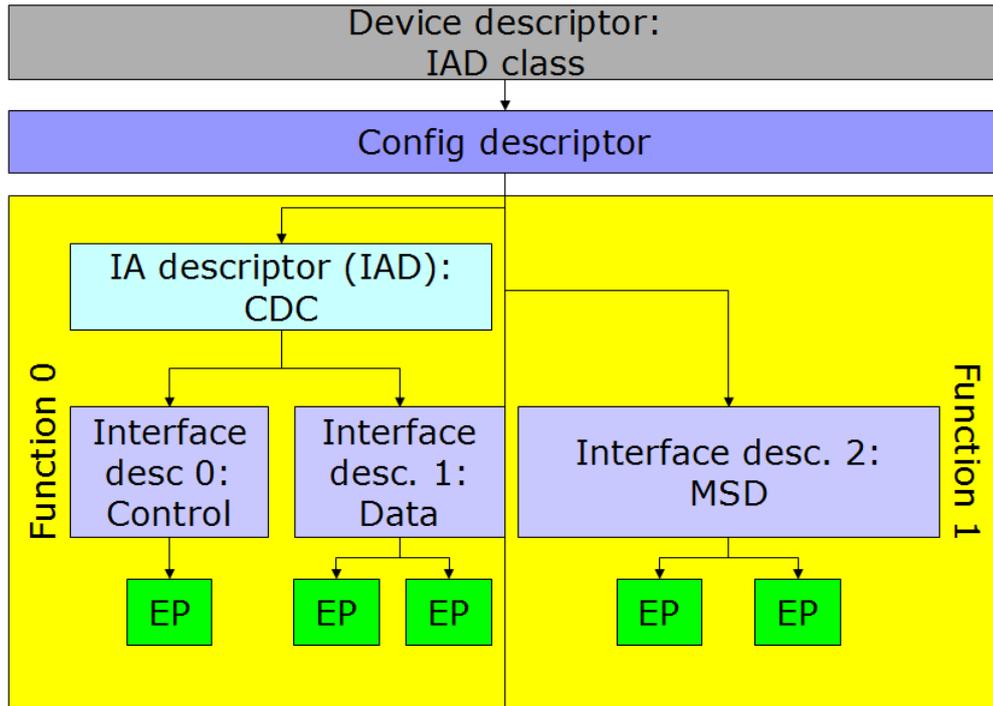


22.1.2 Multiple interface device classes

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 normally would prevent the combination of multiple interface device classes (for example, CDC) with any other component, but this can be avoided by using IAD.

22.1.3 IAD class

To remove the limitation described above the USB protocol defines a descriptor type that allows the combination of single interface device classes with multiple interface device classes. This descriptor is called an 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 hosts

At the time of writing, IAD is supported by:

- Windows XP with Service pack 2 and newer
- Linux Kernel 2.6.22 and higher
- macOS

22.2 Configuration

In general, no configuration is required. By default, emUSB-Device supports up to four interfaces. If more interfaces are needed the following macros must be modified:

Type	Macro	Default	Description
Numeric	USB_MAX_NUM_IF	4	Defines the maximum number of interfaces emUSB-Device shall handle.
Numeric	USB_MAX_NUM_IAD	3	Defines the maximum number of Interface Association Descriptors emUSB-Device shall handle.

22.3 How to combine

Combining different single interface emUSB-Device components (Bulk, HID, MSD) is an easy step, all that needs to be done is calling the appropriate `USBD_XXX_Add()` function. For adding the CDC component additional steps need to be taken. For detailed information refer to *emUSB-Device component specific modification* on page 636 and check the following sample.

Requirements

- Sufficient endpoints for all used device classes. Make sure that your USB device controller has enough endpoints available to handle all the interfaces that shall be integrated. The number of endpoints is limited by the USB controller hardware, this information is usually found in the specific MCU's reference manual.

Sample application

The following sample application uses embOS as the RTOS. This listing is taken from `USB_CompositeDevice_CDC_MSD.c`.

```

/*****
*          (c) SEGGER Microcontroller GmbH          *
*          The Embedded Experts                    *
*          www.segger.com                          *
*****/
----- END-OF-HEADER -----

File      : USB_CompositeDevice_CDC_MSD.c
Purpose   : Sample showing a USB device with multiple interfaces (CDC+MSD).
            This sample combines the functionality of USB_CDC_Echo.c
            and USB_MSD_FS_Start.c samples.

Additional information:
Preparations:
For CDC:
On Windows 8.1 and below the "usbser" driver is not automatically
assigned to the CDC-ACM device. To install the "usbser" driver
see \Windows\USB\CDC . The device can be accessed via COM port
emulation programs e.g. PuTTY.

On Linux no drivers are needed, the device should show up as
/dev/ttyACM0 or similar. "sudo screen /dev/ttyACM0 115200"
can be used to access the device.

On macOS no drivers are needed, the device should show up as
/dev/tty.usbmodem13245678 or similar. The "screen" terminal
program can be used to access the device.

For MSD:
The correct emFile configuration file has
to be included in the project. Depending on the hardware
it can be one of the following:
* FS_ConfigRAMDisk_23k.c
* FS_ConfigNAND_*.c
* FS_ConfigMMC_CardMode_*.c
* FS_ConfigNAND_*.c

Expected behavior:
For CDC:
After connecting the USB cable the PC registers a new COM port appears.
Terminal programs are able to open the COM port.
Any data sent should be received back from the target.

For MSD:
A new MSD volume is recognized by the PC.

```

```

Sample output:
The target side does not produce terminal output.
*/

/*****
*
*      #include section
*
*****/
*/
#include "USB.h"
#include "USB_CDC.h"
#include "BSP.h"
#include "USB_MSD.h"
#include "FS.h"
#include "RTOS.h"

/*****
*
*      Static const data
*
*****/
*/
//
// Information that is used during enumeration.
//
static const USB_DEVICE_INFO _DeviceInfo = {
    0x8765,          // VendorId
    0x1256,          // ProductId
    "Vendor",        // VendorName
    "MSD/CDC Composite device", // ProductName
    "1234567890ABCDEF" // SerialNumber
};
//
// String information used when inquiring the volume 0.
//
static const USB_MSD_LUN_INFO _Lun0Info = {
    "Vendor",        // MSD VendorName
    "MSD Volume",    // MSD ProductName
    "1.00",          // MSD ProductVer
    "134657890"     // MSD SerialNo
};

/*****
*
*      Static data
*
*****/
*/
// Data for MSD Task
static OS_STACKPTR int _aMSDStack[512]; /* Task stacks */
static OS_TASK _MSDTCB;                 /* Task-control-blocks */

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

/*****
*
*      _AddMSD
*
*      Function description
*      Add mass storage device to USB stack
*/

```

```

static void _AddMSD(void) {
    static U8 _abOutBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
    USB_MSD_INIT_DATA    InitData;
    USB_MSD_INST_DATA    InstData;

    InitData.EPIn  = USBD_AddEP(1, USB_TRANSFER_TYPE_BULK, 0, NULL, 0);
    InitData.EPOut = USBD_AddEP(0, USB_TRANSFER_TYPE_BULK, 0, _abOutBuffer, sizeof(_abOutBuffer));
    USBD_MSD_Add(&InitData);
    //
    // Add logical unit 0: RAM drive, using SDRAM
    //
    memset(&InstData, 0, sizeof(InstData));
    InstData.pAPI          = &USB_MSD_StorageByName;
    InstData.DriverData.pStart = (void *)"";
    InstData.pLunInfo = &_Lun0Info;
    USBD_MSD_AddUnit(&InstData);
}
/*****
*
*     _MSDTask
*
* Function description
* Add mass storage device to USB stack
*/
static void _MSDTask(void) {
    while (1) {
        while ((USBD_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED)) !=
        = USB_STAT_CONFIGURED) {
            USB_OS_Delay(50);
        }
        USBD_MSD_Task();
    }
}
/*****
*
*     _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
    USBD_Logf_Application("DTERate=%u, CharFormat=%u, ParityType=%u, DataBits=%u
\n",
        pLineCoding->DTERate,
        pLineCoding->CharFormat,
        pLineCoding->ParityType,
        pLineCoding->DataBits);
#else
    BSP_USE_PARA(pLineCoding);
#endif
}
/*****
*
*     _AddCDC
*
* Function description
* Add communication device class to USB stack
*/
static USB_CDC_HANDLE _AddCDC(void) {
    static U8 _abOutBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
    USB_CDC_INIT_DATA    InitData;

```

```

USB_CDC_HANDLE      hInst;

InitData.EPIn  = USBD_AddEP(USB_DIR_IN,  USB_TRANSFER_TYPE_BULK, 0, NULL, 0);
InitData.EPOut = USBD_AddEP(USB_DIR_OUT, USB_TRANSFER_TYPE_BULK, 0, _abOutBuffer, USB_HS_BULK);
InitData.EPInt = USBD_AddEP(USB_DIR_IN,  USB_TRANSFER_TYPE_INT, 64,  NULL, 0);
hInst = USBD_CDC_Add(&InitData);
USBD_CDC_SetOnLineCoding(hInst, _OnLineCoding);
return hInst;
}
/*****
 *
 *      Public code
 *
 *****/

/*****
 *
 *      MainTask
 *
 *  USB handling task.
 *  Modify to implement the desired protocol
 */
#ifdef __cplusplus
extern "C" { /* Make sure we have C-declarations in C++ programs */
#endif
void MainTask(void);
#ifdef __cplusplus
}
#endif
void MainTask(void) {
    USB_CDC_HANDLE hInstCDC;

    USBD_Init();
    USBD_EnableIAD();
    USBD_SetDeviceInfo(&_DeviceInfo);
    hInstCDC = _AddCDC();
    _AddMSD();
    USBD_Start();
    OS_CREATETASK(&_MSDTCB, "MSDTask", _MSDTask, 200, _aMSDStack);

    while (1) {
        char ac[64];
        int  NumBytesReceived;

        //
        // Wait for configuration
        //
        while ((USBD_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED)) !=
= USB_STAT_CONFIGURED) {
            BSP_ToggleLED(0);
            USB_OS_Delay(50);
        }
        BSP_SetLED(0);
        NumBytesReceived = USBD_CDC_Receive(hInstCDC, &ac[0], sizeof(ac), 0);
        if (NumBytesReceived > 0) {
            USBD_CDC_Write(hInstCDC, &ac[0], NumBytesReceived, 0);
        }
    }
}

/***** end of file *****/

```

22.4 emUSB-Device component specific modification

There are different steps for each emUSB-Device component. The next section shows what needs to be done on both sides: device and host-side.

22.4.1 CDC component

22.4.1.1 Device side

In order to combine the CDC component with other components, the function `USBD_EnableIAD()` needs to be called, otherwise the device will not enumerate correctly. Refer to section How to combine on page 390 and check the listing of the sample application.

22.4.1.2 Host side

Due to a limitation of the internal CDC serial driver of Windows, a composite device with CDC component and another device component(s) is only properly recognized by Windows XP SP3 and above. Linux kernel supports IAD with version 2.6.22. For Windows before Windows 10 the `.inf` file needs 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"
```

red - required modifications

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-Device stack when calling `USBD_CDC_Add()` function.

22.5 MSD and MTP combination feature

MTP is often preferable because of the advantages it offers over MSD. But MTP is not natively recognized by macOS. To provide a solution which works on all the major operating systems SEGGER developed a feature which presents the device as a single interface that enumerates as MTP or as MSD depending on the host PC's operating system.

22.5.1 Configuration

The code of the combination feature is contained in the files `USB_MSD_MTP.c` and `USB_MSD_MTP.h`. The MSD and MTP emUSB-Device packages are obviously also necessary for this to work. The combination feature is disabled by default and can be enabled by setting the configuration define `USB_SUPPORT_MSD_MTP_COMBINATION` to 1 inside your `USB_Conf.h` file.

This system works similar to MSD or MTP. The device information is added via `USBD_MSD_MTP_Add`, storage information is added for the MSD and MTP module (`USBD_MSD_AddUnit` and `USBD_MTP_AddStorage`). And finally the blocking function `USBD_MSD_MTP_Task` is called which handles all the communication as well as switching between MSD and MTP. Source code excerpt from the MSD+MTP sample `USB_MSD_MTP_FS_Start.c`:

```
static void _AddMSDMTP(void) {
    USB_MSD_INIT_DATA MSDInitData;
    USB_MSD_INST_DATA MSDInstData;
    USB_MTP_INIT_DATA MTPInitData;
    USB_MTP_INST_DATA MTPInstData;
    //
    // Set the endpoints in the MSD init data to the same Endpoints as MTP.
    //
    MTPInitData.EPIn = USBD_AddeP(1, USB_TRANSFER_TYPE_BULK,
                                   USB_HS_BULK_MAX_PACKET_SIZE,
                                   NULL, 0);
    MSDInitData.EPIn = MTPInitData.EPIn;
    MTPInitData.EPOut = USBD_AddeP(0, USB_TRANSFER_TYPE_BULK,
                                   USB_HS_BULK_MAX_PACKET_SIZE,
                                   _acReceiveBuffer,
                                   sizeof(_acReceiveBuffer));
    MSDInitData.EPOut = MTPInitData.EPOut;
    MTPInitData.EPInt = USBD_AddeP(1, USB_TRANSFER_TYPE_INT,
                                   10, NULL, 0);
    MTPInitData.pObjectList = _aObjectList;
    MTPInitData.NumBytesObjectList = sizeof(_aObjectList);
    MTPInitData.pDataBuffer = _aDataBuffer;
    MTPInitData.NumBytesDataBuffer = sizeof(_aDataBuffer);
    MTPInitData.pMTPInfo = &_MTPInfo;
    USBD_SetDeviceInfo(&_DeviceInfo);
    USBD_MSD_MTP_Add(&MSDInitData, &MTPInitData);
    //
    // Add logical unit 0 to MSD
    //
    memset(&MSDInstData, 0, sizeof(MSDInstData));
    MSDInstData.pAPI = &USB_MSD_StorageByName;
    MSDInstData.DriverData.pStart = (void *)"";
    MSDInstData.DriverData.pSectorBuffer = _aSectorBuffer;
    MSDInstData.DriverData.NumBytes4Buffer = sizeof(_aSectorBuffer);
    MSDInstData.pLunInfo = &_Lun0Info;
    USBD_MSD_AddUnit(&MSDInstData);
    //
    // Add a storage driver to MTP component.
    //
    MTPInstData.pAPI = &USB_MTP_StorageFS;
    MTPInstData.sDescription = "MTP volume";
    MTPInstData.sVolumeId = "0123456789";
    MTPInstData.DriverData.pRootDir = "";
    USBD_MTP_AddStorage(&MTPInstData);
}
```

```
}
```

22.5.2 Limitations

Because of the way libMTP recognizes MTP devices on Linux based systems a device which uses the MTP/MSD combination feature will be recognized as MSD. Unless the device is added into the libMTP device list (describing this procedure is beyond the scope of this documentation).

22.5.3 Target API

Function	Description
API functions	
USBD_MSD_MTP_Add()	Adds an MSD/MTP interface to the USB stack.
USBD_MSD_MTP_GetMode()	Returns current function mode.
USBD_MSD_MTP_Task()	Main task function of MSD/MTP combination which switches between MSD and MTP.

22.5.3.1 USBD_MSD_MTP_Add()

Description

Adds an MSD/MTP interface to the USB stack.

Prototype

```
int USBD_MSD_MTP_Add(const USB_MSD_INIT_DATA * pMSDInitData,  
                    const USB_MTP_INIT_DATA * pMTPInitData);
```

Parameters

Parameter	Description
pMSDInitData	Pointer to a USB_MSD_INIT_DATA structure.
pMTPInitData	Pointer to a USB_MTP_INIT_DATA structure.

Return value

0 = Successfully added.

Additional information

After the initialization of USB core, this is the first function that needs to be called when an MSD/MTP interface is used with emUSB-Device. The structures `USB_MTP_INIT_DATA` and `USB_MSD_INIT_DATA` have to be initialized before `USBD_MSD_MTP_Add()` is called. The bulk endpoints configured for MTP and MSD must be the same. Refer to `USB_MTP_INIT_DATA` and `USB_MSD_INIT_DATA` for more information.

22.5.3.2 USBD_MSD_MTP_GetMode()

Description

Returns current function mode.

Prototype

```
int USBD_MSD_MTP_GetMode(void);
```

Return value

USB_D_MSD_MTP_MODE_NOT_INITED	Not connected or not initialized yet.
USB_D_MSD_MTP_MODE_MSD	Running as MSD.
USB_D_MSD_MTP_MODE_MTP	Running as MTP.

Additional information

This function can be used to make sure that the application code does not access the storage medium when running as MSD.

22.5.3.3 USBD_MSD_MTP_Task()

Description

Main task function of MSD/MTP combination which switches between MSD and MTP.

Prototype

```
void USBD_MSD_MTP_Task(void);
```

Additional information

The function returns when the USB device is detached or suspended.

22.6 Issues on Windows 7

When connecting a Multi-interface device to Windows 7, sometimes a wrong driver is installed causing the device not to work.

The device needs to be handled as a composite device whereas Windows/Third party USB host software installs the driver of the first interface only.

This problem can be fixed manually using SEGGER's USB Composite Device Driver Fixer tool: <https://www.segger.com/downloads/free-utilities/CompositeDeviceFixer>.

22.6.1 Detailed description

The cause why the driver was wrongly selected is that some USB 3.0 controller drivers which also include their own USB 3.0 stack since the native USB host stack of Windows 7 cannot handle USB 3.0 devices (Microsoft introduced a native USB 3.0 stack only on Windows 8 and higher).

Normally the third-party USB 3.0 stacks handle the enumeration and data transfers correctly. Unless it comes to the MS OS descriptor handling. Almost all third-party stacks handle them incorrectly. Especially when it comes to devices which have multiple interfaces such as RNDIS+MSD or MTP+Bulk or RNDIS+WinUSB etc. or RNDIS+CDC. Even when IAD is used it is wrongly passed. In normal cases Windows would initially report that a composite device is detected. Afterwards the single interfaces are enumerated and the driver for each interface will be loaded. When MS OS Descriptors are used the information is passed to the dedicated interfaces. Those third-party USB 3.0 stack are NOT doing this. They pass this information to the device instead of the interface, which is wrong. Microsoft says in the MS OS Descriptor 2.0 Specification that the MS OS Descriptor information needs to be passed to the interfaces: [Microsoft OS 2.0 Descriptors Specification /April 2017 - Chapter Overview] "Scoping of MS OS descriptors With MS OS version 1.0 descriptors, Windows USB driver stack does not query for any MS OS descriptors if the device is a composite device, instead defers such queries to the USB Generic Parent Driver (Usbccgp.sys). The effect is that all MS OS descriptors are applied to specific composite functions, and none can be applied to the entire device itself."

Neither do those third-party USB host stacks implement a proper handling of the MS OS Descriptors nor do they pass the information properly to other drivers/stacks.

Chapter 23

Target OS Interface

This chapter describes the functions of the operating system abstraction layer.

23.1 General information

emUSB-Device includes an OS abstraction layer which should make it possible to use an arbitrary operating system together with emUSB-Device. To adapt emUSB-Device to a new OS one only has to map the functions listed below in section Interface function list 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.

23.1.1 Operating system support supplied with this release

emUSB-Device packages contain an abstraction layer for *embOS* (`USB_OS_embOSv5.c`). A kernel abstraction layer for using emUSB-Device without any RTOS (superloop) is also supplied (`USB_OS_None.c`).

Abstraction layers for the following operating systems are readily available:

- FreeRTOS
- μ C/OS-II
- μ C/OS-III
- CMSIS-RTX
- Keil-RTX
- ThreadX
- chibiOS
- CMX RTOS

Abstraction layers for other operating systems can be written upon request.

23.2 Interface function list

Name	Description
API functions	
<code>USB_OS_DeInit()</code>	Frees all resources used by the OS layer.
<code>USB_OS_Delay()</code>	Delays for a given number of ms.
<code>USB_OS_DecRI()</code>	Leave a critical region for the USB stack: Decrements interrupt disable count and enable interrupts if counter reaches 0.
<code>USB_OS_GetTickCnt()</code>	Returns the current system time in milliseconds or system ticks.
<code>USB_OS_IncDI()</code>	Enter a critical region for the USB stack: Increments interrupt disable count and disables interrupts.
<code>USB_OS_Init()</code>	This function initializes all OS objects that are necessary.
<code>USB_OS_Panic()</code>	Is called if the stack encounters a fatal error.
<code>USB_OS_Signal()</code>	Wakes the task waiting for signal.
<code>USB_OS_Wait()</code>	Blocks the task until <code>USB_OS_Signal()</code> is called for a given transaction.
<code>USB_OS_WaitTimed()</code>	Blocks the task until <code>USB_OS_Signal()</code> is called for a given transaction or a timeout occurs.

23.2.1 USB_OS_DeInit()

Description

Frees all resources used by the OS layer.

Prototype

```
void USB_OS_DeInit(void);
```

23.2.2 USB_OS_Delay()

Description

Delays for a given number of ms.

Prototype

```
void USB_OS_Delay(int ms);
```

Parameters

Parameter	Description
<code>ms</code>	Number of <code>ms</code> .

23.2.3 USB_OS_DecRI()

Description

Leave a critical region for the USB stack: Decrements interrupt disable count and enable interrupts if counter reaches 0.

Prototype

```
void USB_OS_DecRI(void);
```

Additional information

The USB stack will perform nested calls to `USB_OS_IncDI()` and `USB_OS_DecRI()`. This function may be called from a task context or from within an interrupt. If called from an interrupt, it need not do anything.

An alternate implementation would be to

- enable the USB interrupts,
- unlock the mutex or semaphore locked in `USB_OS_IncDI()`

if the disable count reaches 0.

This may be more efficient, because interrupts of other peripherals can be serviced while inside a critical section of the USB stack.

23.2.4 USB_OS_GetTickCnt()

Description

Returns the current system time in milliseconds or system ticks.

Prototype

```
U32 USB_OS_GetTickCnt(void);
```

Return value

Current system time.

23.2.5 USB_OS_IncDI()

Description

Enter a critical region for the USB stack: Increments interrupt disable count and disables interrupts.

Prototype

```
void USB_OS_IncDI(void);
```

Additional information

The USB stack will perform nested calls to `USB_OS_IncDI()` and `USB_OS_DecRI()`. This function may be called from a task context or from within an interrupt. If called from an interrupt, it need not do anything.

An alternate implementation would be to

- perform a lock using a mutex or semaphore and
- disable the USB interrupts.

This may be more efficient, because interrupts of other peripherals can be serviced while inside a critical section of the USB stack.

23.2.6 USB_OS_Init()

Description

This function initializes all OS objects that are necessary.

Prototype

```
void USB_OS_Init(void);
```

23.2.7 USB_OS_Signal()

Description

Wakes the task waiting for signal.

Prototype

```
void USB_OS_Signal(unsigned EPIndex,  
                  unsigned TransactCnt);
```

Parameters

Parameter	Description
<code>EPIndex</code>	Endpoint index. Signaling must be independent for all endpoints.
<code>TransactCnt</code>	Transaction counter. Specifies which transaction has been finished.

Additional information

This routine is typically called from within an interrupt service routine.

23.2.8 USB_OS_Wait()

Description

Blocks the task until `USB_OS_Signal()` is called for a given transaction.

Prototype

```
void USB_OS_Wait(unsigned EPIndex,  
                unsigned TransactCnt);
```

Parameters

Parameter	Description
<code>EPIndex</code>	Endpoint index. Signaling must be independent for all endpoints.
<code>TransactCnt</code>	Transaction counter. Specifies the transaction to wait for.

Additional information

The function must ignore signaling transactions other than given in `TransactCnt`. If this transaction was signaled before this function was called, it must return immediately.

This routine is called from a task.

23.2.9 USB_OS_WaitTimed()

Description

Blocks the task until `USB_OS_Signal()` is called for a given transaction or a timeout occurs.

Prototype

```
int USB_OS_WaitTimed(unsigned EPIndex,
                    unsigned ms,
                    unsigned TransactCnt);
```

Parameters

Parameter	Description
<code>EPIndex</code>	Endpoint index. Signaling must be independent for all endpoints.
<code>ms</code>	Timeout time given in <code>ms</code> .
<code>TransactCnt</code>	Transaction counter. Specifies the transaction to wait for.

Return value

0 Task was signaled within the given timeout.
 1 Timeout occurred.

Additional information

The function must ignore signaling transactions other than given in `TransactCnt`. If this transaction was signaled before this function was called, it must return immediately.

`USB_OS_WaitTimed()` is called from a task. This function is used by all available timed routines.

Alternatively this function may take the given timeout in units of system ticks of the underlying operating system instead of milliseconds. In this case all API functions that support a timeout parameter should also use system ticks for the timeout.

Chapter 24

Target USB Driver

This chapter describes how to configure a USB driver for emUSB-Device in detail.

24.1 General information

Purpose of the USB hardware interface

emUSB-Device 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 have already been tested with emUSB-Device 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, USB 2.0 and USB 3.0 controllers.

24.1.1 Available USB drivers

An always up to date list can be found at:

<https://www.segger.com/emusb-drivers.html>

24.2 Adding a driver to emUSB-Device

USB_D_Init() initializes the internals of the USB stack and is always the first function which the USB application has to call. USB_D_Init() will then call USB_D_X_Config(). This function should be used to perform the following tasks:

- Perform device specific hardware initialization if necessary.
- Add a USB driver to your project.
- Enable SuperSpeed by calling the function USB_D_EnableSuperSpeed() if possible and desired.
- Assign a memory area to be used for endpoint buffers if required by the driver, see USB_D_AssignMemory().
- Optionally install a HWAttach function.
- Install interrupt management functions.

You have to specify the USB device driver which should be used with emUSB-Device. For this, USB_D_AddDriver() should be called in USB_D_X_Config() with the identifier of the driver which is compatible to your hardware as parameter. Refer to the header file USB.h for a list of all supported devices and their valid identifiers.

The _HWAttach() function 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 may not be present if your hardware does not need to perform such actions. A _HWAttach() function may be registered to the stack by calling USB_D_SetAttachFunc() within USB_D_X_Config().

Additionally a function to enable the USB interrupt must be installed using the function USB_D_SetISREnableFunc().

Modify USB_D_X_Config(), _EnableISR() and if required, _HWAttach().

24.2.1 USB_D_X_Config()

Description

Configure the USB stack.

Prototype

```
void USB_D_X_Config(void)
```

Additional information

This function is always called from USB_D_Init().

Example

```
/* Example excerpt from USB_Config_SAM7A3.c */
#define PID_USB (27) // USB Identifier
#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)

static void _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 */
}

static void _EnableISR(USB_ISR_HANDLER * pfISRHandler) {
    *(U32*)(0xFFFFF080 + 4 * PID_USB) = (U32)pfISRHandler; // Set interrupt vector
    *(U32*)(0xFFFFF128)                = (1 << PID_USB);
    // Clear pending interrupt
    *(U32*)(0xFFFFF120)                = (1 << PID_USB); // Enable Interrupt
}

void USBD_X_Config(void) {
    USBD_AddDriver(&USB_Driver_AtmeSAM7A3);
    USBD_SetAttachFunc(_HWAttach);
    USBD_SetISREnableFunc(_EnableISR);
}
```

24.2.2 USBD_X_DisableInterrupt()

Description

This function is called by the stack in cases where the stack must perform a critical operation which can not be interrupted by a new incoming USB interrupt event.

Prototype

```
void USBD_X_DisableInterrupt(void);
```

Additional information

This function is MCU/USB controller specific. Normally it is defined in the hardware specific `USB_Config_*.c` file.

This function is only called by the stack if the define `USB_D_OS_USE_USBD_X_INTERRUPT` is set to 1 in `USB_Conf.h`. If this is not the case interrupts are disabled/enabled globally.

Example

```
/* Example excerpt from USB_Config_SEGGER_emPower.c */
void USBD_X_DisableInterrupt(void) {
    NVIC_DisableIRQ(USBHS_IRQn);
}

/* Example excerpt from USB_Config_Renesas_RSKRX71M.c */
void USBD_X_DisableInterrupt(void) {
    USB0_IER_USB0    &= ~(1uL << USB0_IER_USB0_BIT);
    USB0_IER_USBR0   &= ~(1uL << USB0_IER_USBR0_BIT);
}
```

24.2.3 USBD_X_EnableInterrupt()

Description

This function is called by the stack to enable USB interrupt(s) after they have been disabled by `USB_D_X_DisableInterrupt()`.

Prototype

```
void USBD_X_EnableInterrupt(void);
```

Additional information

This function is MCU/USB controller specific. Normally it is defined in the hardware specific `USB_Config_*.c` file.

This function is only called by the stack if the define `USB_D_OS_USE_USB_D_X_INTERRUPT` is set to 1 in `USB_Conf.h`. If this is not the case interrupts are disabled/enabled globally.

Example

```
/* Example excerpt from USB_Config_SEGGER_emPower.c */
void USBD_X_EnableInterrupt(void) {
    NVIC_EnableIRQ(USBHS_IRQn);
}

/* Example excerpt from USB_Config_Renesas_RSKRX71M.c */
void USBD_X_EnableInterrupt(void) {
    USB0_IER_USB0   |= (1uL << USB0_IER_USB0_BIT);
    USB0_IER_USBR0  |= (1uL << USB0_IER_USBR0_BIT);
}
```

24.3 Device driver specifics

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

This section lists the drivers which require special configuration and describes how to configure those drivers.

Restrictions caused by the USB controller hardware are also listed in this section.

24.3.1 LPC54/55xxx full-speed driver

Configuration

This driver needs a memory area to store the endpoint table and endpoint transfer buffers. By default the dedicated USB RAM attached to the high-speed controller is used. If the full-speed and high-speed controllers are used simultaneously, then a different memory area must be assigned to the driver by calling the function `USBD_AssignMemory()` in `USB-D_X_Config()`.

Minimum required memory: 256 bytes + 'maximum packet size' for each used non-control endpoint.

The memory area must be aligned to a 256-byte boundary.

24.3.2 LPC54/55xxx high-speed driver

This driver is used for the MCUs:

- LPC54608
- LPC540xx/54S0xx
- LPC55xxx
- iMXRT500

Restrictions

In some versions of the LPC54xxx MCUs, the high-speed device controller contains a serious bug: Under some circumstances the first byte of a data packet transferred to the host is changed to `0x00` (See LPC546xx errata sheet, Rev. 2.1, 23 October 2018, USB.15). Although the workaround suggested by NXP is implemented in the driver, data packets may still get corrupted if IN and OUT endpoints are active at the same time. This can't be avoided in many applications. This problem is known for the LPC54608 and some early samples of the LPC54018/LPC54S018. We recommend not to use the high-speed device controller on devices with this issue. When in doubt, check with NXP for a specific device.

The LPC55xxx high-speed device controller is not able to send ISO packets of size 1024 to the host. Only packet sizes up to 1023 bytes will work.

24.3.3 EHCI driver

This driver is used for the MCUs:

- LPC18xx
- LPC43xx
- Kinets (HS controller)
- iMX RT105x/RT106x
- Zynq70xx

Configuration

This driver needs a memory area to store the DMA descriptors and endpoint transfer buffers. By default a memory area declared inside the driver code is used. The size of this area can be configured by setting the preprocessor symbol `USB_ENDPOINT_BUFFER_POOL_SIZE` in `USB_Conf.h`.

Alternately the application may provide the memory to the driver: Then the dynamic memory variant of the driver must be added to the USB stack and the function `USBD_AssignMemory()` must be called.

Example

```
USBD_AddDriver(&USB_Driver_NXP_LPC43xx_DynMem);
USBD_AssignMemory(_MemPool, sizeof(_MemPool));
```

Minimum required memory (for systems without cached memory):

- 918 to 1536 bytes (depending on the number of endpoints the controller provides)
- For each used non-control OUT (RX) endpoint: 32 bytes + 'maximum packet size' of the endpoint
- For each used non-control IN (TX) endpoint: 64 bytes

Minimum required memory (for systems using cached memory):

- 1046 to 1664 bytes (depending on the number of endpoints the controller provides)
- For each used non-control endpoint (IN or OUT): 96 bytes + 2 * 'maximum packet size' of the endpoint

The memory area must be aligned to a 2048-byte boundary.

Cache support

If the driver is installed on a system using cached (data) memory, cache functions for cleaning and invalidating cache lines must be provided and set with `USBD_SetCacheConfig()`.

24.3.4 nRF52xxx, nRF53xx driver

Restrictions

A data transfer from the device to the host can't be canceled any more after the data packet was passed to the USB controller. Due to a restriction of the USB controller of the nRFxxxx MCUs, the packet remains in the hardware buffer until it can be transferred to the host. This even applies if the host is disconnected from the device.

This means that timeouts given to any of the USB Write functions do not work as expected. In case of a timeout, the function is terminated, but the data to be written is still pending.

24.3.5 Synopsys DWC2 driver (slave mode)

This driver does not use DMA and is applicable for the MCUs:

- STM32F105/107
- STM32F2xx
- STM32F4xx
- STM32F7xx
- STM32H7xx
- STM32L4x5/4x6/4x7/4x9
- XMC45xx
- EFM32GGxxx

Configuration

This driver needs a memory area for endpoint transfer buffers. By default a memory area declared inside the driver code is used. The size of this area can be configured by setting the preprocessor symbol `USB_ENDPOINT_BUFFER_POOL_SIZE` in `USB_Conf.h`.

For the STM32F7xx and STM32H7xx drivers the memory may be provided by the application instead. In this case the dynamic memory variant of the driver must be added to the USB stack and the function `USBD_AssignMemory()` must be called.

Example

```
USBD_AddDriver(&USB_Driver_ST_STM32F7xxHS_DynMem);
USBD_AssignMemory(_MemPool, sizeof(_MemPool));
```

Minimum required memory:

- 64 bytes
- For each used non-control OUT (RX) endpoint: The 'maximum packet size' of the endpoint

Restrictions

High bandwidth ISO transactions are not supported, therefore a maximum of 1024 bytes can be transferred per microframe in high-speed.

24.3.6 Synopsys DWC2 driver (DMA mode)

This driver can be used for the high-speed controllers of the MCUs:

- STM32F4xx
- STM32F7xx
- STM32H7xx

Configuration

This driver needs a memory area for endpoint transfer buffers, which must be provided by the application by calling the function `USBD_AssignMemory()` after `USBD_AddDriver()`.

Minimum required memory:

- 128 bytes
- For each used non-control endpoint (IN and OUT each): The 'maximum packet size' of the endpoint

All sized must be round up to a multiple of 4 bytes or the cache line size, if the system uses a cache.

Cache support

If the driver is installed on a system using cached (data) memory, cache functions for cleaning and invalidating cache lines must be provided and set with `USBD_SetCacheConfig()`.

Restrictions

High bandwidth ISO transactions are not supported, therefore a maximum of 1024 bytes can be transferred per microframe.

24.3.7 XHCI driver

Configuration

The function `USBD_EnableSuperSpeed()` must be called within `USBD_X_Config()`, if the device shall be able to operate at SuperSpeed. If the function is not called, the USB controller will enumerate in high-speed only.

This driver needs a memory area for endpoint transfer buffers which must be provided by the application using the `USBD_AssignMemory()` function.

Typical required memory (on a system with a data cache line size of 64 bytes):

- 1152 bytes
- For each used non-control OUT (RX) endpoint: The 'maximum packet size' of the endpoint + 512 bytes
- For each used non-control IN (TX) endpoint: 192 bytes

For optimal and reproducible memory allocation behaviour the memory area provided to `USBD_AssignMemory()` should be cache aligned and should not span a 64KB boundary.

Cache support

Cache functions for cleaning and invalidating cache lines must be provided and set with `USBD_SetCacheConfig()`.

24.3.8 Renesas RX driver

This driver is used for the MCUs:

- RX113
- RX231
- RX71M (full-speed controller)
- RX62N
- RX63N
- RX64M
- RX65N

Restrictions

Due to a hardware limitation the maximum packet size of isochronous endpoints is limited to 256 bytes (instead of the normally possible 1023 bytes).

24.3.9 AT91RM9200 driver

Restrictions

SETUP OUT transfers with more than 8 bytes can cause the controller to lock-up. A setup OUT transfer consists of a SETUP stage, an optional DATA OUT stage and a STATUS IN stage. The usage of status IN with preceding data is relatively rare, certain HID commands can trigger this, e.g. "SetFeature", in most other protocols setup transfers are rarely done in the OUT direction with a data stage. Usage of common protocols (MSD, CDC, etc.) should not be affected. The USB controller in this MCU appears to have a critical bug with status IN transactions which results in a complete lock-up of the controller until power cycle. The issue occurs during the status stage of setup transfers consisting of more than one data packet:

Works:

- SETUP packet
- 7 byte data OUT
- 0 byte status IN

Does not work:

- SETUP packet
- 8 byte data OUT
- 1 byte data OUT
- 0 byte status IN <-- During this transaction the IN token can be seen and on the device side, the ZLP transaction is started, but the controller locks up. After this the controller will no longer receive any interrupts.

Chapter 25

Support

25.1 Contacting support

Before contacting support please make sure that you are using the latest version of the emUSB-Device package. Also please check the chapter *Configuring debugging output* on page 44 and run your application with enabled debug support.

If you are a registered emUSB-Device user and you need to contact the emUSB-Device support please send the following information via email to ticket_emusb@segger.com*:

- Your emUSB-Device registration number.
- emUSB-Device version.
- A detailed description of the problem
- The configuration files `USB_Conf*. *`
- Any error messages.

Please also take a few moments to help us to improve our service by providing a short feedback when your support case has been solved.

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at <https://www.segger.com/legal/privacy-policy/>.

Chapter 26

Profiling with SystemView

This chapter describes how to configure and enable profiling of emUSB-Device using SystemView.

26.1 Profiling overview

emUSB-Device is instrumented to generate profiling information of API functions and driver-level functions.

These profiling information expose the run-time behavior of emUSB-Device in an application, recording which API functions have been called, how long the execution took, and revealing which driver-level functions have been called by API functions or events like interrupts.

The profiling information is recorded using SystemView.

SystemView is a real-time recording and visualization tool for profiling data. It exposes the true run-time behavior of a system, going far deeper than the insight provided by debuggers. This is particularly effective when developing and working with complex systems comprising an OS with multiple threads and interrupts, and one or more middleware components.

SystemView can ensure a system performs as designed, can track down inefficiencies, and show unintended interactions and resource conflicts.

The recording of profiling information with SystemView is minimally intrusive to the system and can be done on virtually any system. With SEGGER's Real Time Technology (RTT) and a J-Link, SystemView can record data in real-time and analyze the data live, while the system is running.

The emUSB-Device profiling instrumentation can be easily configured and set up.

26.2 Additional files for profiling

Additional files are required on target and PC side for full functionality of SystemView.

26.2.1 Additional files on target side

The SystemView module needs to be added to the application to enable profiling. If not already part of the project, download the sources from <https://www.segger.com/systemview.html> and add them to the project.

Also make sure that `USB_SYSVIEW.c` from the `/USB/` directory is included in the project.

26.2.2 Additional files on PC side

For fully functional and readable outputs in the SystemView PC application, a description file for the corresponding middleware is required. This description file extends the values sent from the target to fully readable text outputs.

While SystemView already comes with the most recent description files at the time the SystemView release has been built, these files might not be the latest available. The latest SystemView description files can be found in the emUSB-Device shipment in the folder `/Shared/SystemView/Description/`. You can copy these files over to the `Description` folder that comes with the SystemView package.

The version at the end of the SystemView description file does not have to match the exact version of the middleware it is used with. They are valid from this version onwards until a description file for a newer version is required.

26.3 Enable profiling

Profiling can be included or excluded at compile-time and enabled at run-time. When profiling is excluded, no additional overhead in performance or memory usage is generated. Even when profiling is enabled the overhead is minimal, due to the efficient implementation of SystemView.

To include profiling, define `USBD_SUPPORT_PROFILE` as 1 in the emUSB-Device configuration (`USB_Conf.h`) or in the project preprocessor defines.

Per default profiling is included when the global define `SUPPORT_PROFILE` is set.

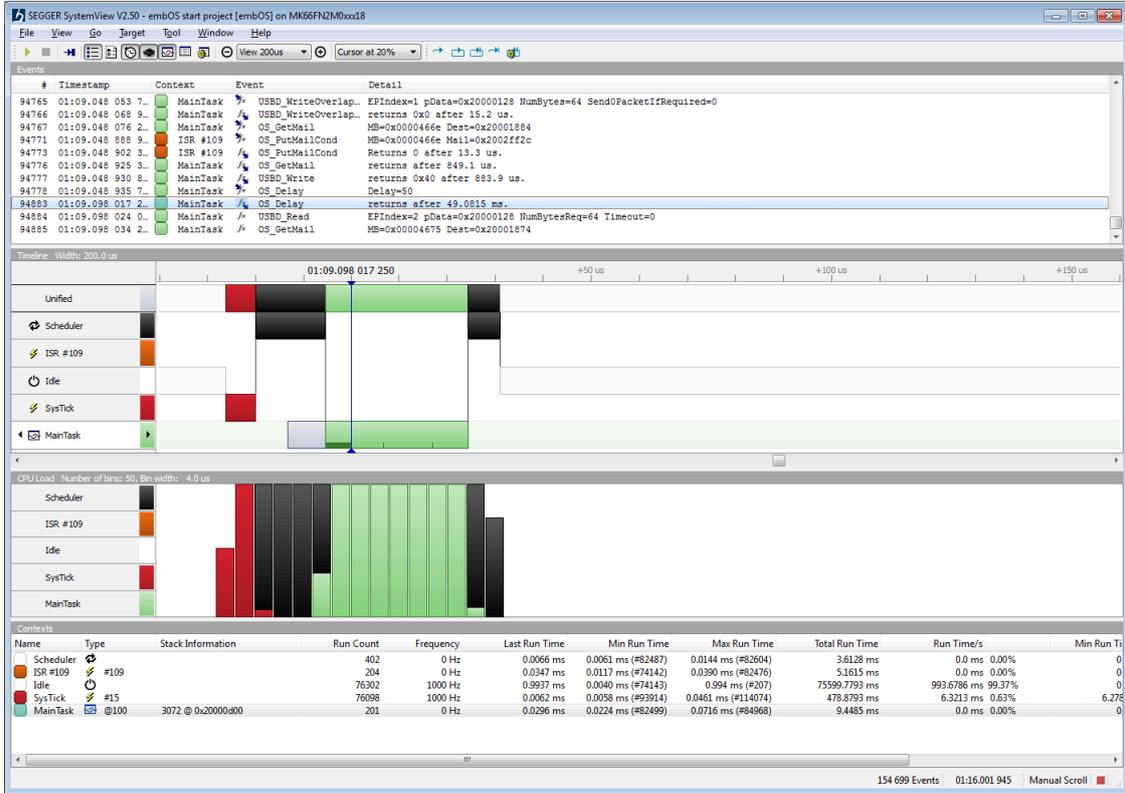
```
#if defined(SUPPORT_PROFILE) && (SUPPORT_PROFILE)
    #ifndef    USBD_SUPPORT_PROFILE
        #define USBD_SUPPORT_PROFILE        1
    #endif
#endif
```

To enable profiling at run-time, `USBD_SYSVIEW_Init()` needs to be called. Profiling can be enabled at any time, it is recommended to do this in the user-provided configuration `USBD_X_Config()`:

```
/* *****
 *
 *      USBD_X_Config
 *
 */
void USBD_X_Config(void) {
    ...
    #if USBD_SUPPORT_PROFILE
        USBD_SYSVIEW_Init();
    #endif
    ...
}
```

26.4 Recording and analyzing profiling information

When profiling is included and enabled emUSB-Device generates profiling events. On a system which supports RTT (i.e. ARM Cortex-M and Renesas RX) the data can be read and analyzed with SystemView and a J-Link. Connect the J-Link to the target system using the default debug interface and start the SystemView host application. If the system does not support RTT, SystemView can be configured for single-shot or postmortem mode. Please refer to the SystemView User Manual for more information.



Chapter 27

Debugging

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

27.1 Message output

The debug builds of emUSB-Device include a fine grained debug system which helps to analyze the correct implementation of the stack in your application. All modules of the USB stack can output logging and warning messages via terminal I/O, if the specific message type identifier is added to the log and/or warn filter mask. This approach provides the opportunity to get and interpret only the logging and warning messages which are relevant for the part of the stack that you want to debug.

By default, all of the warning messages and none of the logging messages are activated. All activated messages are forwarded to the functions `USB_X_Log()` and `USB_X_Warn()`. These functions are located in the source file `USB_ConfigIO.c` and may be customized or replaced if necessary.

27.2 API functions

Function	Description
Filter functions	
<code>USBD_AddLogFilter()</code>	Adds one or more message types to the logging filter.
<code>USBD_AddWarnFilter()</code>	Adds one or more message types to the warning filter.
<code>USBD_SetLogFilter()</code>	Sets the message type(s) for the logging filter.
<code>USBD_SetWarnFilter()</code>	Sets the message type(s) for the warning filter.
General debug functions/macros	
<code>USB_PANIC</code>	Called if the stack encounters a critical situation.
General helper prototypes	
<code>USB_X_Log()</code>	This function is called by the stack in debug builds with log output.
<code>USB_X_Warn()</code>	This function is called by the stack in debug builds with warning output.
<code>USB_OS_Panic()</code>	Is called if the stack encounters a fatal error.

27.2.1 USBD_AddLogFilter()

Description

Adds one or more message types to the logging filter.

Prototype

```
void USBD_AddLogFilter(U32 FilterMask);
```

Parameters

Parameter	Description
<code>FilterMask</code>	Specifies which logging messages should be added to the filter mask. Refer to <i>Message types</i> on page 690 for a list of valid values for parameter <code>FilterMask</code> .

Additional information

`USBD_AddLogFilter()` can also be used to remove a filter condition which was set before. It adds the specified filter to the filter mask via a disjunction.

Example

```
void Application (void) {
    USBD_AddLogFilter(USB_MTYPE_DRIVER); // Activate driver logging messages
    USBD_Init();
    /*
     * Do something
     */
}
```

27.2.2 USBD_AddWarnFilter()

Description

Adds one or more message types to the warning filter.

Prototype

```
void USBD_AddWarnFilter(U32 FilterMask);
```

Parameters

Parameter	Description
<code>FilterMask</code>	Specifies which warning messages should be added to the filter mask. Refer to <i>Message types</i> on page 690 for a list of valid values for parameter <code>FilterMask</code> .

Additional information

`USBD_AddWarnFilter()` can also be used to remove a filter condition which was set before. It adds the specified filter to the filter mask via a disjunction.

Example

```
void Application (void) {
    USBD_AddWarnFilter(USB_MTYPE_DRIVER); // Activate driver warning messages
    USBD_Init();
    /*
     * Do something
     */
}
```

27.2.3 USBD_SetLogFilter()

Description

Sets the message type(s) for the logging filter.

Prototype

```
void USBD_SetLogFilter(U32 FilterMask);
```

Parameters

Parameter	Description
<code>FilterMask</code>	Specifies which logging messages should be set to the filter mask. Refer to <i>Message types</i> on page 690 for a list of valid values for parameter <code>FilterMask</code> .

Additional information

This function can be called before `USB_Init()`. By default, none of filter conditions are set. The sample application contain a simple implementation which can be easily modified.

27.2.4 USBD_SetWarnFilter()

Description

Sets the message type(s) for the warning filter.

Prototype

```
void USBD_SetWarnFilter(U32 FilterMask);
```

Parameters

Parameter	Description
<code>FilterMask</code>	Specifies which warning messages should be set to the filter mask. Refer to <i>Message types</i> on page 690 for a list of valid values for parameter <code>FilterMask</code> .

Additional information

This function can be called before `USB_Init()`. By default, none of filter conditions are set. The sample application contain a simple implementation which can be easily modified.

27.2.5 USB_PANIC

Description

This macro is called by the stack code when it detects a situation that should not be occurring and the stack can not continue. The intention for the `USB_PANIC()` macro is to invoke whatever debugger may be in use by the programmer. In this way, it acts like an embedded breakpoint. This macro is mostly used in cases where emUSB-Device was configured improperly.

Prototype

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

Additional information

This macro maps to a function in debug builds only. If `USB_DEBUG > 0`, the macro maps to the stack internal function `USB_OS_Panic()`. It disables all interrupts to avoid further task switches, outputs `sError` via terminal I/O and loops forever. When using an emulator, you should set a breakpoint at the beginning of this routine or simply stop the program after a failure. The error message is passed to the function as parameter. In a release build, this macro is defined empty, so that no additional code will be included by the linker.

27.2.6 USB_X_Log()

Description

This function is called by the stack in debug builds with log output. In a release build, this function is not be linked in.

Prototype

```
void USB_X_Log(const char * s);
```

Parameters

Parameter	Description
<code>s</code>	Pointer to a string holding the log message.

27.2.7 USB_X_Warn()

Description

This function is called by the stack in debug builds with warning output. In a release build, this function is not be linked in.

Prototype

```
void USB_X_Warn(const char * s);
```

Parameters

Parameter	Description
<code>s</code>	Pointer to a string holding the warning message.

27.2.8 USB_OS_Panic()

Description

Is called if the stack encounters a fatal error.

Prototype

```
void USB_OS_Panic(const char * pErrMsg);
```

Parameters

Parameter	Description
<code>pErrMsg</code>	Pointer to a string holding the error message.

Additional information

In a release build this function is not linked in. The default implementation of this function disables all interrupts to avoid further task switches, outputs the error string via terminal I/O and loops forever. When using an emulator, you should set a break-point at the beginning of this routine or simply stop the program after a failure.

27.3 Message types

Description

The same message types are used for log and warning messages. Separate filters can be used for both log and warnings. For details, refer to `USBD_SetLogFilter()` and `USBD_SetWarnFilter()` as well as `USBD_AddLogFilter()` and `USBD_AddWarnFilter()` for more information about using the message types.

Definition

```
#define USB_MTYPE_INIT           (1UL << 0)
#define USB_MTYPE_CORE          (1UL << 1)
#define USB_MTYPE_CONFIG        (1UL << 2)
#define USB_MTYPE_DRIVER        (1UL << 3)
#define USB_MTYPE_ENUMERATION   (1UL << 4)
#define USB_MTYPE_CDC           (1UL << 7)
#define USB_MTYPE_HID           (1UL << 8)
#define USB_MTYPE_MSD           (1UL << 9)
#define USB_MTYPE_MSD_CDROM     (1UL << 10)
#define USB_MTYPE_MSD_PHY       (1UL << 11)
#define USB_MTYPE_MTP           (1UL << 12)
#define USB_MTYPE_PRINTER       (1UL << 13)
#define USB_MTYPE_RNDIS         (1UL << 14)
#define USB_MTYPE_VIRTUAL_MSD   (1UL << 16)
#define USB_MTYPE_UVC           (1UL << 17)
#define USB_MTYPE_ECM           (1UL << 18)
#define USB_MTYPE_AUDIO         (1UL << 19)
#define USB_MTYPE_NCM           (1UL << 20)
#define USB_MTYPE_MIDI          (1UL << 21)
#define USB_MTYPE_INFO          (1UL << 31)
```

Symbols

Definition	Description
USB_MTYPE_INIT	Activates output of messages from the initialization of the stack that should be logged.
USB_MTYPE_CORE	Activates output of messages from the core of the stack that should be logged.
USB_MTYPE_CONFIG	Activates output of messages from the configuration of the stack.
USB_MTYPE_DRIVER	Activates output of messages from the driver that should be logged.
USB_MTYPE_ENUMERATION	Activates output of messages from enumeration that should be logged. Note: Since enumeration is handled in an ISR, use this with care as the timing will be changed greatly.
USB_MTYPE_CDC	Activates output of messages from CDC module that should be logged when a CDC connection is used.
USB_MTYPE_HID	Activates output of messages from HID module that should be logged when a HID connection is used.
USB_MTYPE_MSD	Activates output of messages from MSD module that should be logged when a MSD connection is used.
USB_MTYPE_MSD_CDROM	Activates output of messages from MSD CDROM module that should be logged.
USB_MTYPE_MSD_PHY	Activates output of messages from MSD Physical layer that should be logged.
USB_MTYPE_MTP	Activates output of messages from MTP module that should be logged when a MTP connection is used.

Definition	Description
USB_MTYPE_PRINTER	Activates output of messages from Printer module that should be logged when Printer connection is used.
USB_MTYPE_RNDIS	Activates output of messages from RNDIS module that should be logged when a RNDIS connection is used.
USB_MTYPE_VIRTUAL_MSD	Activates output of messages from VirtualMSD module that should be logged when a VirtualMSD connection is used.
USB_MTYPE_UVC	Activates output of messages from UVC module that should be logged when a UVC connection is used.
USB_MTYPE_ECM	Activates output of messages from ECM module that should be logged when a ECM connection is used.
USB_MTYPE_AUDIO	Activates output of messages from Audio module that should be logged when an audio connection is used.
USB_MTYPE_NCM	Activates output of messages from NCM module that should be logged when a NCM connection is used.
USB_MTYPE_MIDI	Activates output of messages from MIDI module that should be logged when a MIDI connection is used.
USB_MTYPE_INFO	Non-maskable info messages

Chapter 28

Certification

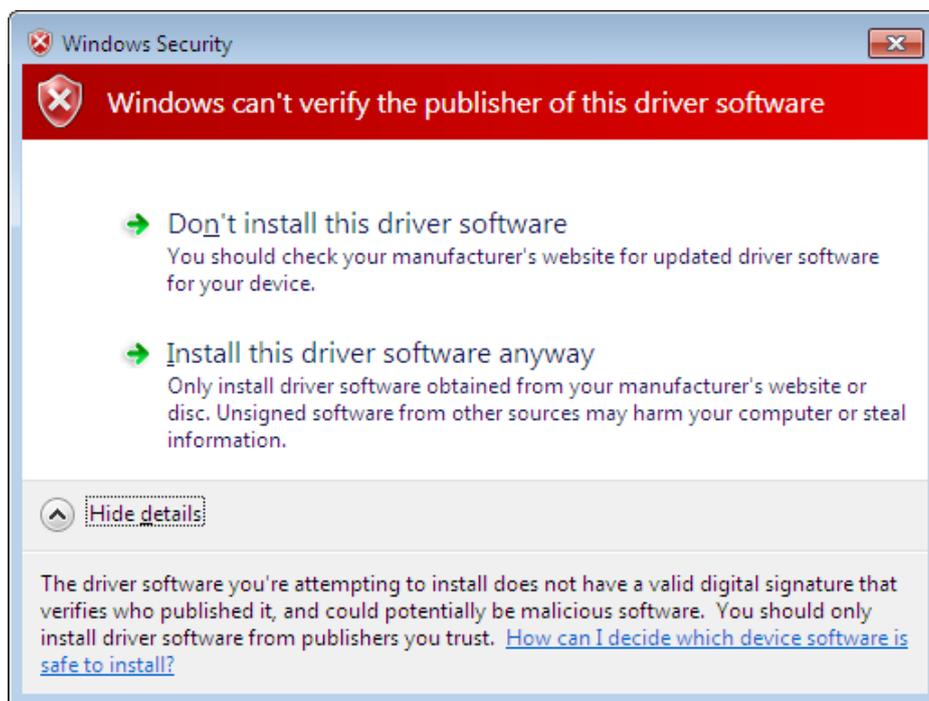
This chapter describes the process of USB driver certification with Microsoft Windows. This chapter is only relevant for drivers which require an .inf file.

28.1 What is the Windows Hardware Certification and why do I need it?

The Windows Hardware Certification (formerly known as “Windows Logo Certification”) process will sign a driver with a Microsoft certificate which signifies that the device is compatible and safe to use with Microsoft Windows operating systems.

If the device is using a call which requires `.inf` files and if the driver is not signed the user will be confronted with messages saying that the driver is not signed and may not be safe to use with Microsoft Windows. Depending on which Windows version you are using a different message will be shown. Users of Windows Server 2008, Windows Vista x64, Windows 7 x64 and newer versions of the respective distributions will be warned about the missing signature and the driver will show up as installed, but the driver will not be loaded. The user can override this security measure by hitting F8 on Windows start-up and selecting “Disable Driver Signature Enforcement” or editing the registry.

Microsoft Windows Vista/7:



28.2 Certification offer

Customers can complete the certification by themselves. But SEGGER also offers certification for our customers. To certify a device a customer needs a valid Vendor ID, registered at <https://www.usb.org> and a free Product ID. Using the Hardware Certification Kit a certification package is created. The package is sent to Microsoft for confirmation. After the confirmation is received from Microsoft the customer receives a `.cat` file which allows the drivers to be installed without problems.

28.3 Vendor and Product ID

A detailed description of the Vendor and Product ID can be found in chapter *Product / Vendor IDs* on page 39.

The customer can acquire a Vendor ID from the USB Implementers Forum, Inc. (<https://www.usb.org>). This allows to freely decide which Product ID is used for which product.

28.4 Certification without SEGGER Microcontroller

Certification can be completed by the customer themselves. The certification is a lengthy and complicated process. For further information, as well as the requirements see:

<https://msdn.microsoft.com/en-us/library/windows/hardware/jj124227.aspx>

Chapter 29

Performance & resource usage

This chapter covers the performance and resource usage of emUSB-Device. It contains information about the memory requirements in typical systems which can be used to obtain sufficient estimates for most target systems.

29.1 Memory footprint

emUSB-Device is designed to fit many kinds of embedded design requirements. Several features can be excluded from a build to get a minimal system. The code size depends on the API functions called by the application. The code was compiled for a Cortex-M4 CPU with the SEGGER compiler and size optimization. Note that the values are only valid for an average configuration.

The following table shows the approximate RAM and ROM requirement of emUSB-Device in bytes:

Component	ROM	RAM	Note
USB core	5600	1000	
Bulk	2000	100	
CDC	1300	100	
HID	1500	100	
MSD	4900	500	+ size of file system + configurable sector buffer of minimum 512 bytes (RAM)
MTP	14500	1700	+ size of file system + configurable file data buffer of minimum 512 bytes (RAM) + configurable object buffer (typically 4 kBytes RAM)
Printer	1000	2100	
RNDIS	5300	1500	+ size of the IP stack
ECM	2900	200	+ size of the IP stack
IP-Over-USB	7200	1700	+ size of the IP stack
VirtualMSD	8400	1000	+ heap of minimum 1700 bytes RAM
DFU	1000	0	
AUDIO	2300	100	
MIDI	1800	0	
CCID	1200	0	
Driver Atmel SAM3U	2000	600	
Driver Atmel SAM3X	1900	500	
Driver Atmel SAM3S	2200	100	
Driver Atmel SAM7S	2100	100	
Driver Atmel SAM9X25	1900	600	
Driver Atmel SAMA5D2x	2200	600	
Driver Atmel SAMV7	1700	600	
Driver EM EFM32GG990	2900	700	
Driver Freescale KHCI	2300	700	
Driver Freescale KinetisEHCI	2500	2600	
Driver Infineon XMC45xx	2800	400	
Driver NXP LPC17xx	1700	100	
Driver NXP LPC18xx	2500	4200	
Driver NXP LPC23xx	1500	100	
Driver NXP LPC43xx	2500	4200	

Component	ROM	RAM	Note
Driver Renesas RZ	2700	7800	
Driver Renesas RX	2400	700	
Driver Renesas SynergyS1	2200	600	
Driver Renesas SynergyFS	2300	700	
Driver Renesas SynergyHS	2500	4700	
Driver ST STM32x32	1600	200	
Driver ST STM32F107	2800	400	
Driver ST STM32F4xxFS	2900	400	
Driver ST STM32F4xxHS	3000	2900	
Driver ST STM32F7xxFS	2800	600	
Driver ST STM32F7xxHS	3000	4500	
Driver ST STM32L4xx	3000	600	
Driver ST STR91x	1300	0	
Driver TI AM335x	1400	500	
Driver TI OMAP L138	1300	2100	
Driver Xilinx Ultrascale0	3700	200	+ endpoint buffer RAM

Additionally 64 or 512 bytes of RAM (64 for full-speed and 512 for high-speed devices) are necessary for each OUT-endpoint as a data buffer. This buffer is assigned within the application.

29.2 Performance

The tests were run on a LPC4357 CPU running at 180 MHz using the USB Bulk component connected to a Linux host.

The following table shows the transfer speed of emUSB-Device:

Description	Speed
USB high-Speed controller (device to host)	44.1 MB/s
USB high-Speed controller (host to device)	41.8 MB/s
USB full-Speed controller	1200 kB/s

Chapter 30

FAQ

This chapter answers some frequently asked questions.

Q: When designing my hardware can I just permanently connect the D+ 1.5 kOhm pull-up resistor to 3.3V to save a MCU pin?

A: No, the pull-up being connected tells the USB host that the device is ready to communicate. Permanently connecting the pull-up is dangerous as the host may start to communicate with the device prematurely, before it has finished with the start-up. Furthermore, when using USB high-speed the device must disconnect the pull-up from D+ according to the USB 2.0 specification. Any currently known USB high-speed controller (with internal or external USB high-speed PHY) have internal logic to enable an internal pull-up initially. So for those controllers an external pull-up is not necessary.

Q: When using MSD can I read/write onto the storage medium when the device is connected to a USB host?

A: No, when a MSC device is connected to a USB host the host is the sole master of the storage medium. It can write or read at any point in time. Should the application try to access the storage medium at the same time as the host the results are unpredictable. To resolve this issue the device needs to detach the storage medium from the host, see `USBD_MSD_RequestDisconnect()`, `USBD_MSD_Disconnect()` and `USBD_MSD_WaitForDisconnection()`.

Q: Can I combine different USB components together?

A: Yes. See *Combining USB components (Multi-Interface)* on page 627.

Q: Do I need a real-time operating system (RTOS) to use the emUSB-Device-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 emUSB-Device-MSD 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 emUSB-Device-MSD?

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.