emUSB-Device

USB Device stack

CPU-independent

User & Reference Guide

Document: UM09001
Software version: 3.00d
Revision: 4
Date: June 8, 2016

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

Copyright notice

Trademarks

Contact address

SEGGER Microcontroller GmbH & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

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

Manual versions

This manual describes the current software version. If any error occurs, inform us and we will try to assist you as soon as possible.
Contact us for further information on topics or routines not yet specified.

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

Print date: June 8, 2016

Software	Revision	Date	By	Description
3.00d	4	160608	YR	Update to latest software version.
3.00c	3	160523	YR	Update to latest software version. Chapter Bulk communication: 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: Updated prototype for USBD_SetMaxPower. Chapter HID: Added new function for Setting a callback for SET_REPORT. Chapter Debugging: Changed all prototypes from USB_* to USBD_*.
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 emUSB-Device offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

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

Table 1.1: Typographic conventions

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

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

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

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

Corporate Office:
http://www.segger.com

United States Office:
http://www.segger-us.com

EMBEDDED SOFTWARE (Middleware)

emWin
Graphics software and GUI
emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display.

embOS
Real Time Operating System
embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources.

embOS/IP
TCP/IP stack
embOS/IP a high-performance TCP/IP stack that has been optimized for speed, versatility and a small memory footprint.

emFile
File system
emFile is an embedded file system with FAT12, FAT16 and FAT32 support. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.

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

SEGGER TOOLS

Flasher
Flash programmer
Flash Programming tool primarily for micro controllers.

J-Link
JTAG emulator for ARM cores
USB driven JTAG interface for ARM cores.

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

J-Link / J-Trace Related Software
Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.

Table of Contents

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/second) devices is approximately 1 Mbyte per second. 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/second) is also fully supported and is automatically handled. Using USB high speed mode with an ARM9 or faster could achieve values of approx. 18 MBytes/second and faster.

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

1.3 emUSB-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.
SmartMSD	emUSB-Device SmartMSD Component
CDC	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.
Core layer	
emUSB-Device-Core	TheemUSB-Device core is the intrinsic USB stack.
Hardware layer	
Driver	USB controller driver.

Table 1.1: emUSB-Device components

1.3.1 emUSB-Device-Bulk

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

1.3.1.1 Purpose of emUSB-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/second) and high speed (480 Mbit/second) transfer rates
- OS-abstraction: Can be used with any RTOS, but no OS is required for MSD-only devices

1.3.2.4 How does it work?

Use file system support from host OS

A device which uses emUSB-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.

Only provide file system code on the target if necessary

Mass storage devices like USB sticks do not require their own file system implementation. File system program code is only required if the application running on the target device has to access the stored data. The development of a file system is a 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 *www.segger.com/emfile.html* for more information about emFile.

1.3.3 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.

1.3.3.1 Typical applications

Typical applications are:

- Modem
- Telephone system
- Fax machine

1.3.4 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.

1.3.4.1 Typical applications

Typical examples

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

1.3.5 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.

1.3.5.1 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.

1.3.6 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.

1.3.6.1 Typical applications

Typical applications are:

- Laser/Inkjet printer
- CNC machine

1.3.7 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.

1.3.7.1 Typical applications

Typical applications are:

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

1.3.8 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.

1.3.8.1 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 Chapter 21 *Performance & resource usage* on page 473.
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 *Target OS Interface* on page 411.

1.4.2 Development environment (compiler)

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

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

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

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

1.5 File structure

The following table shows the contents of the emUSB-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. **Note:** Do not change the source code in this directory.
Windows	Contains Windows specific applications which can be used in conjunction with the device application samples.

Table 1.2: Supplied directory structure of emUSB-Device package

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 Kbytes/second to hundreds of Mbytes/second.
- A wide range of packet sizes and data transfer rates are supported.
- USB provides internal error handling. Together with the already mentioned hot-plug capability this greatly improves robustness.
- The provisions for powering connected devices dispense the need for extra power supplies for many low power devices.
- Several transfer modes are supported which ensures the wide applicability of USB.

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

2.1.2 Important USB Standard Versions

USB 1.1 (September 1998)

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

USB 2.0 (April 2000)

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

USB 3.0 (November 2008)

As all previous USB standards, USB 3.0 is fully forward and backward compatible. Existing cables and connectors may be reused but 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.

2.1.3 USB System Architecture

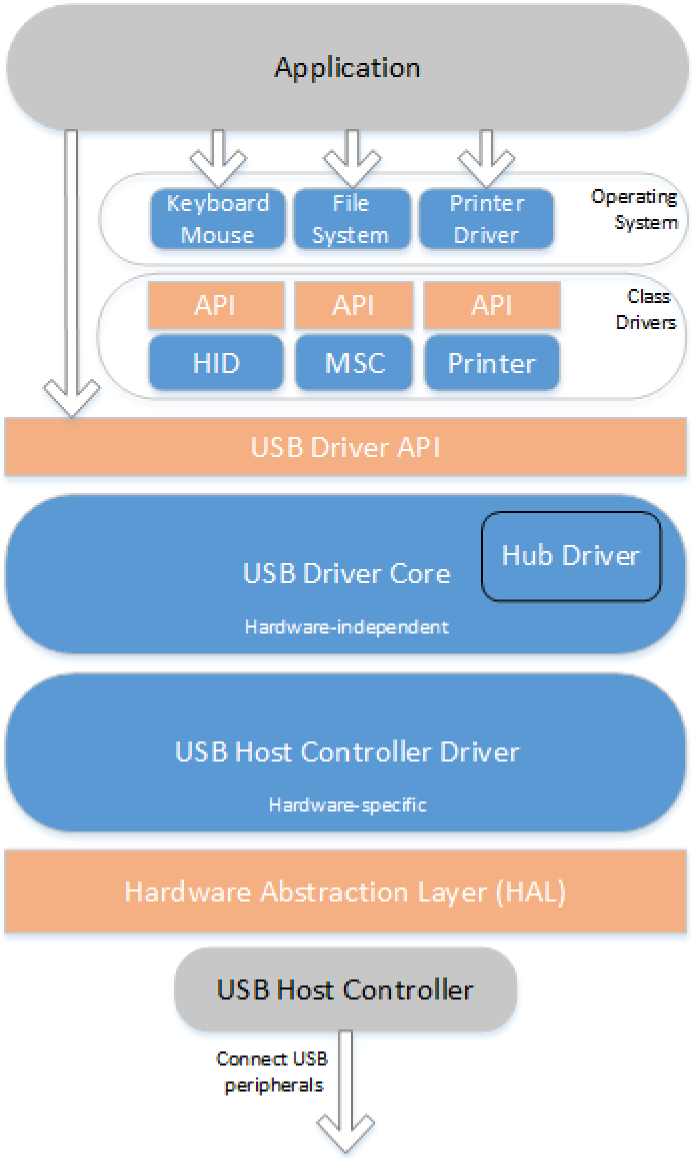
A USB system is composed of three parts - a host side, a device side and a physical bus. The physical bus is represented by the USB cable and connects the host and the device.

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

USB Host

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

- A host controller driver (HCD) which provides the functionality of the host controller hardware.
- The USB Driver (USBD) 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

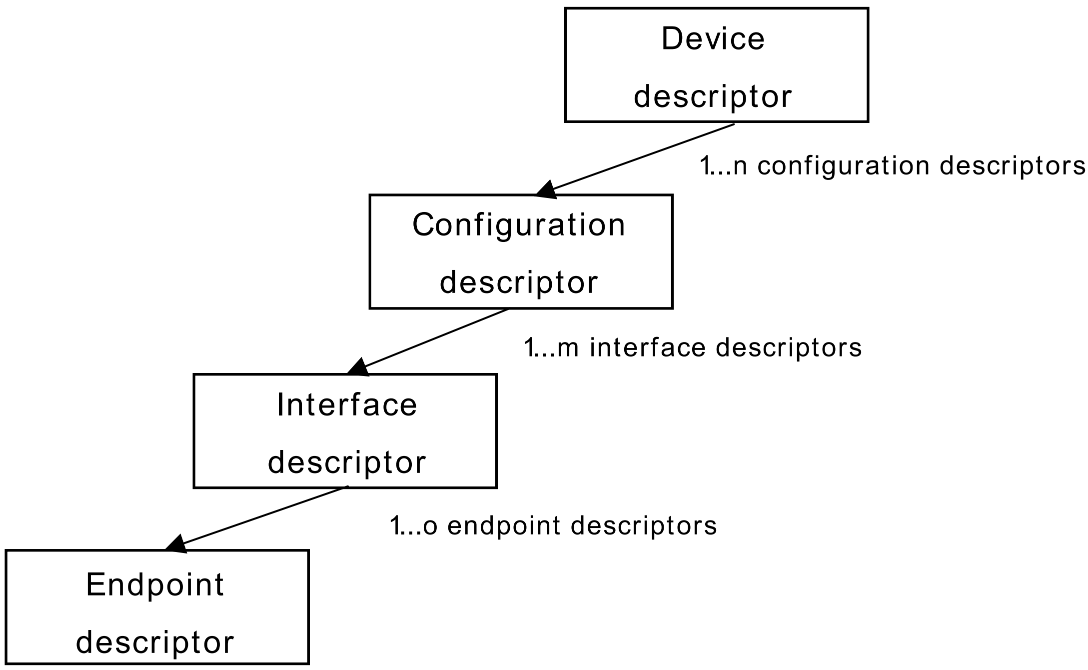
Two types of devices exist: hubs and functions. Hubs provide for additional USB attachment points. Functions provide capabilities to the host and are able to transmit or receive data or control information over the USB bus. Every peripheral USB device represents at least one function but may implement 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.

Isochronous transfers

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

Interrupt transfers

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

Bulk transfers

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

2.1.5 Setup phase / Enumeration

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

2.1.6 Product / Vendor IDs

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 v8 devices have the Vendor ID 0x1366 and Product ID 0x0101.

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.
Possible options to obtain a Vendor ID or Product ID are described in the chapter *Vendor and Product ID* on page 471.

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

2.4 References

For additional information see the following documents:

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

Chapter 3

Getting started

The first step in getting emUSB-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

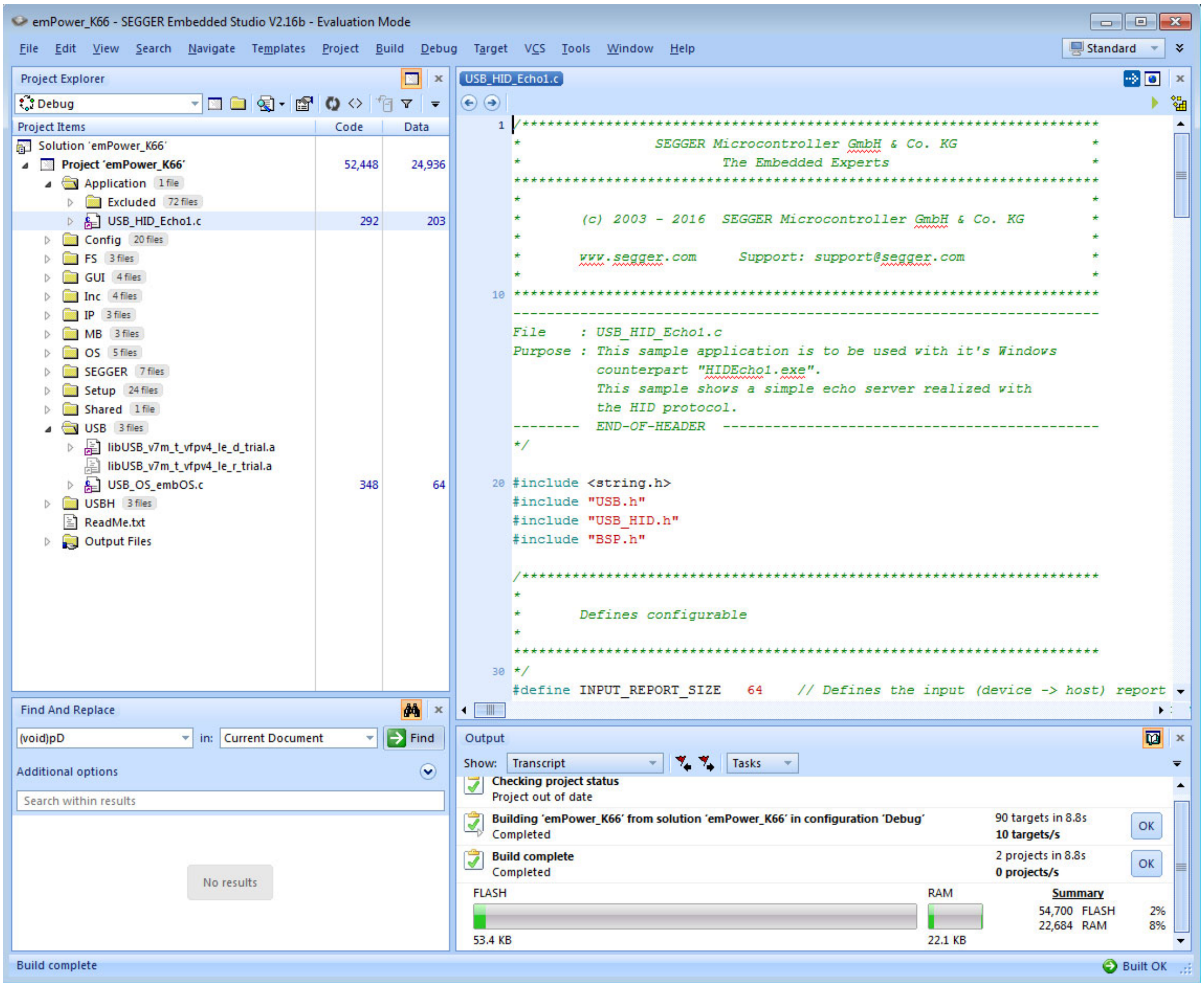
To get the USB up and running, 3 possible ways currently available:

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

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 SEGGER 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 make files; in this case, when we say "add to the project", this translates into "add to the make file".

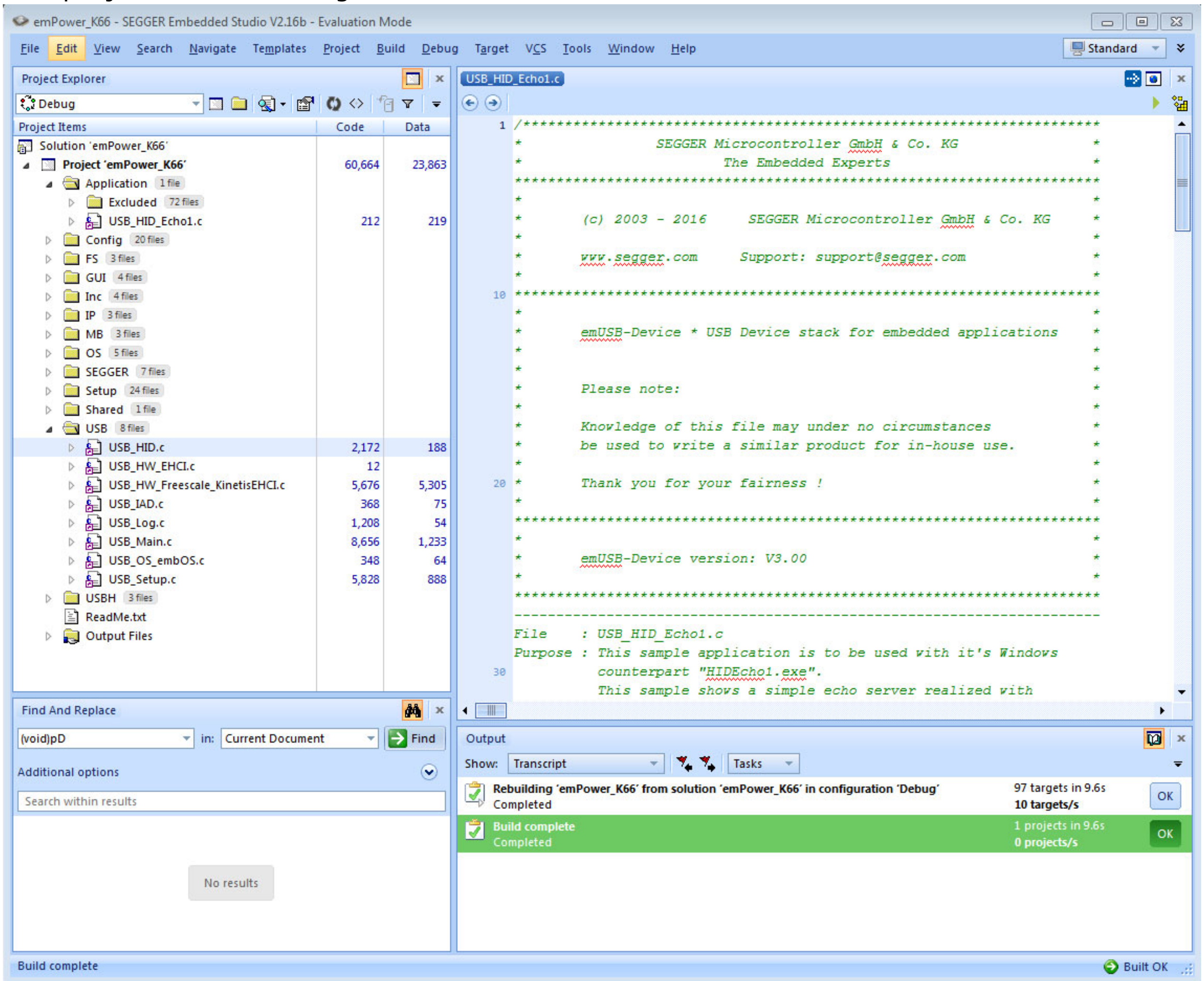
3.1.1 Upgrade a trial version available on the web with source code.

Simply download a trial package available from the SEGGER website.
After downloading, extract the trial project and open the workspace/project file which is located in the start folder.



All relevant source files from the emUSB-Device shipment need to be copied into the folders of the trial package.

Afterwards the project needs to be updated by adding the source files into the project and removing the USB libraries.



3.1.2 Upgrading an embOS Start project

Integrating emUSB-Device

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

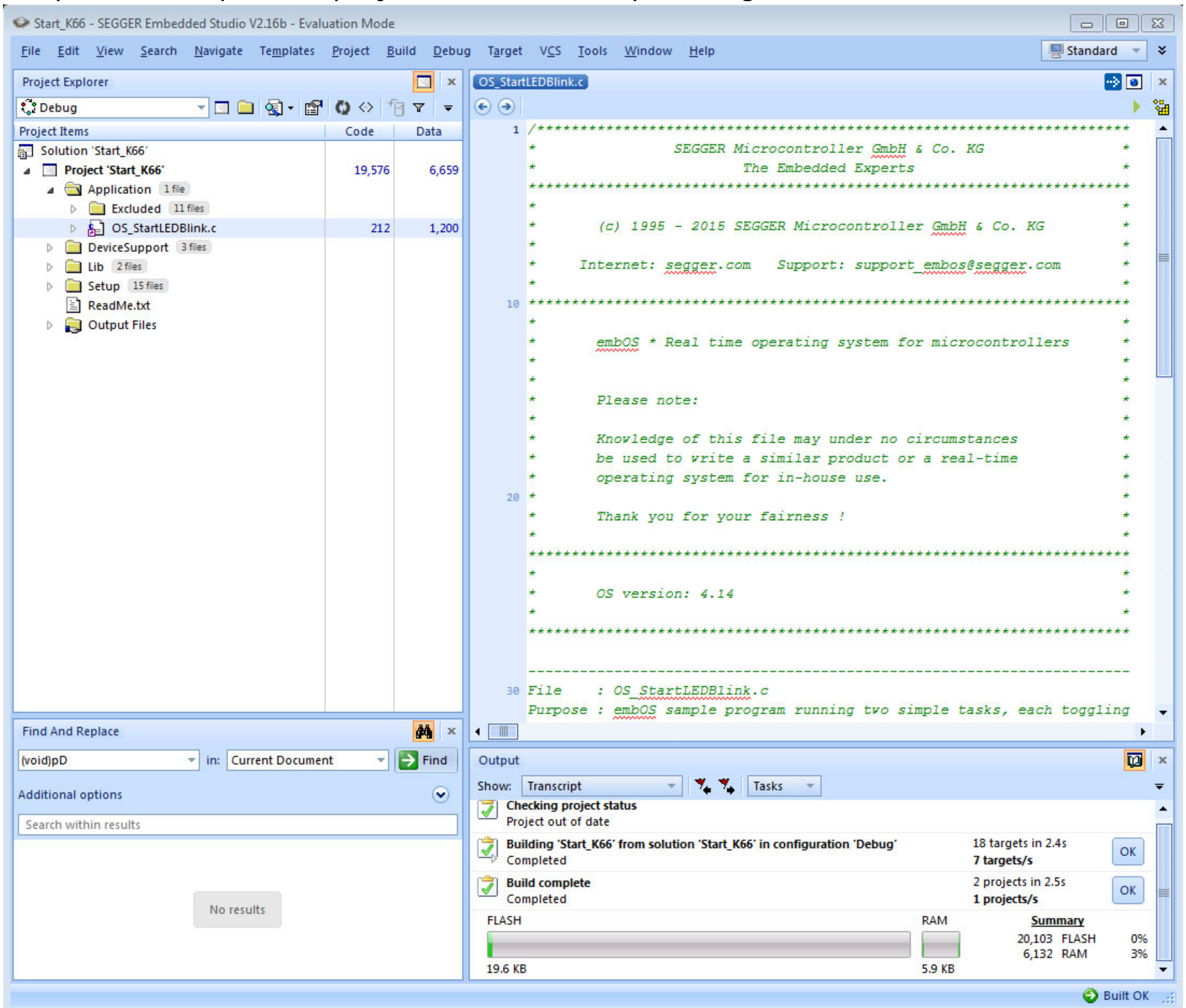
Procedure to follow

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

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

Step 1: Open an embOS start project

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



Step 2: Adding emUSB-Device to the start project

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

- `Config`
- `USB`
- `Sample\USB\OS\embOS`
- `BSP\<your hardware>\Setup`

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

Configuring the include path

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

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

3.1.3 Creating a project from scratch

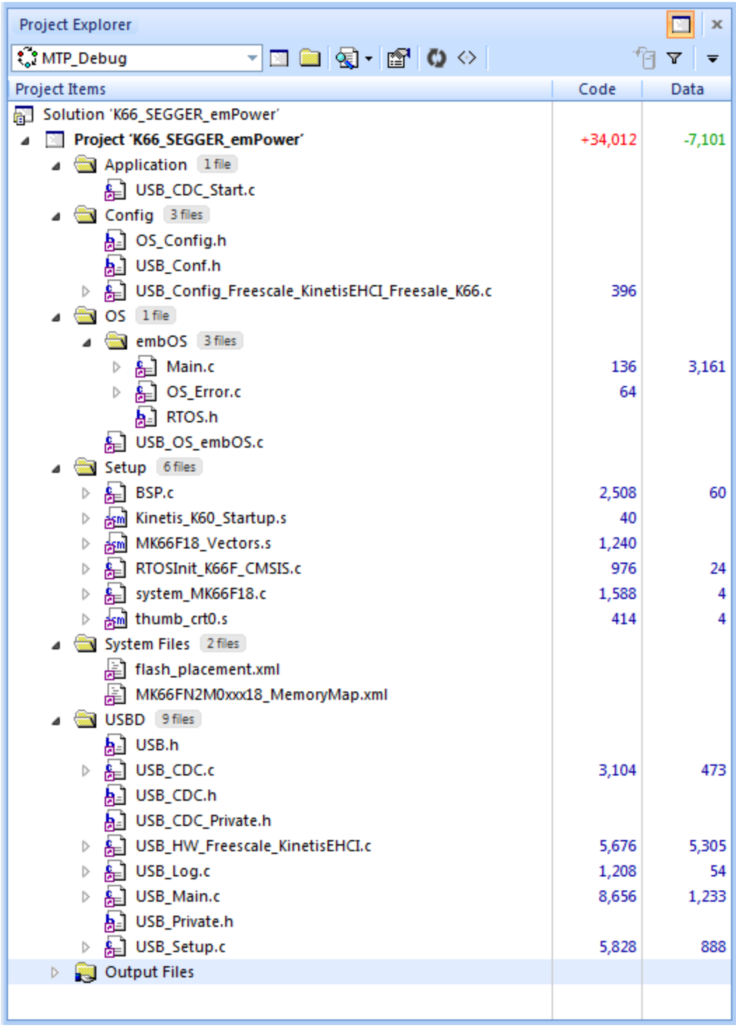
To get the target system to behave like a mass storage device or generic bulk device on the USB bus, a few steps have to be taken:

- A project or make file has to be created for the used toolchain.
- The configuration may need to be adjusted.
- The hardware routines for the USB controller have to be implemented.
- Add the path of the required USB header files to the include path.

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

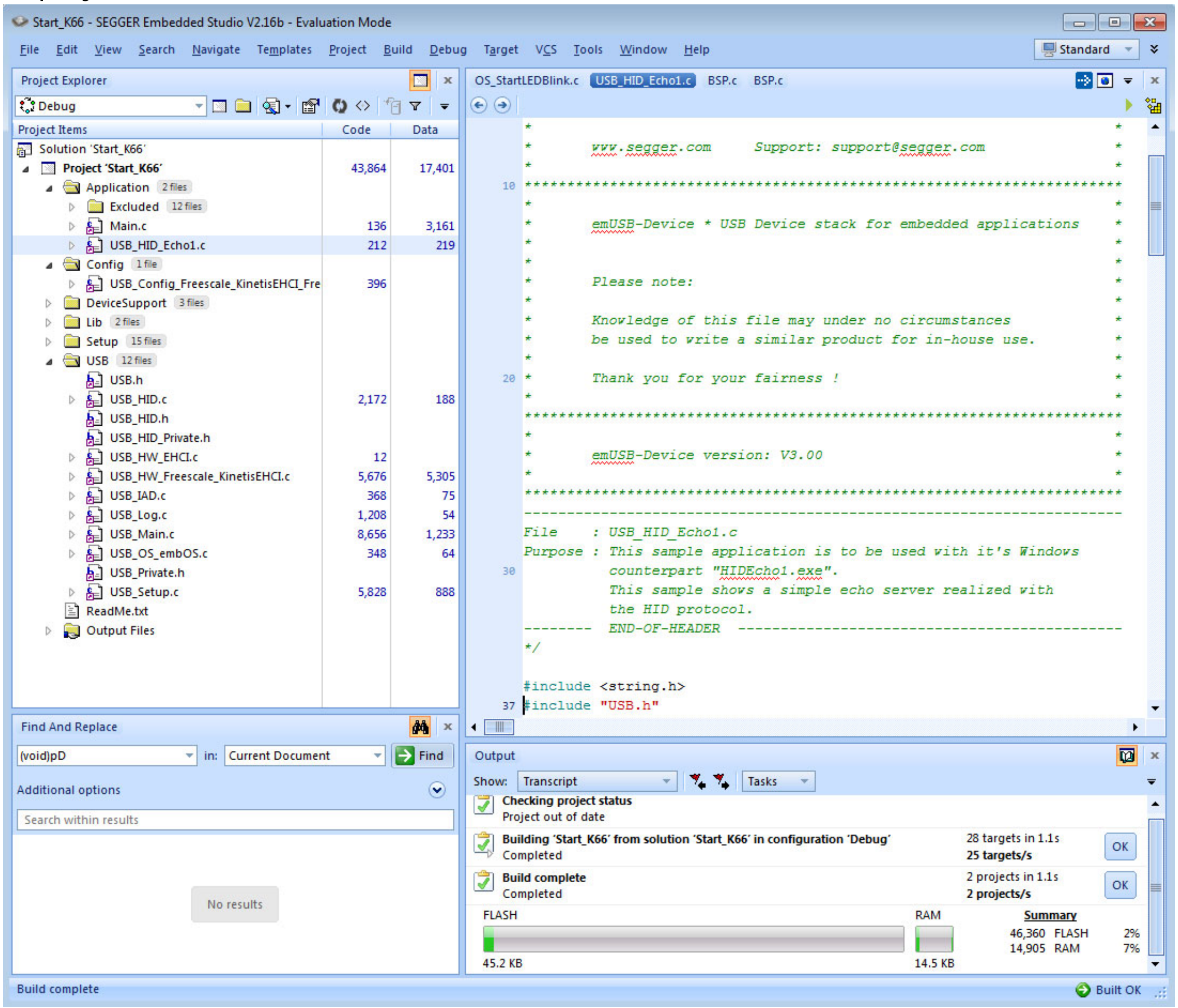
Creating the project or make file

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



3.2 Select the start application

For quick and easy testing of your emUSB-Device integration, start with the code found in the folder `Application`. Add `USB_HID_Echo1.c` as your applications to your project.



3.3 Build the project and test it

Build the project. It should compile without errors and warnings. If you encounter any problem during the build process, check your include path and your project configuration settings. To test the project, download the output into your target and start the application.
After connecting the USB cable to the target device, the mouse pointer should hop from left to right.

3.4 Configuration

An application using emUSB-Device must contain a structure containing the device identification information:

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

Member	Description
VendorId	Vendor ID of the target.
ProductId	Product ID of the target.
sVendorName	The manufacturer name.
sProductName	The product name of the target.
sSerialNumber	The serial number of the device.

Table 3.1: USB_DEVICE_INFO elements

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.
Used Product IDs	
0x1234	Example Product ID for all bulk samples.
0x1000	Example Product ID for all MSD samples.
0x1200	Example Product ID for the MSD CD-ROM sample.
0x1111	Example Product ID for all CDC samples.
0x1112	Example Product ID for HID mouse sample.
0x1114	Example Product ID for the vendor specific HID sample.
0x2114	Example Product ID for the Printer class sample.

Table 3.2: List of used Product and Vendor IDs

3.4.1 General emUSB-Device configuration

3.4.1.1 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()`.

Prototype

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

Additional information

The Vendor ID is assigned by the USB Implementers Forum (*www.usb.org*). For tests, the default number above (or pretty much any other number) can be used. However, you may not bring a product to market without having been assigned your own Vendor ID. For emUSB-Device-Bulk and 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 78 or *The .inf file* on page 265. Otherwise, the Windows host will be unable to locate the driver.

The Product ID in combination with the Vendor ID creates a worldwide unique identifier. For tests, you can use the default number above (or pretty much any other number). For emUSB-Device-Bulk and 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 78 or *The .inf file* on page 265. 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 minimun 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();
```


3.4.2 Additional required configuration functions for emUSB-MSD

Refer to *Configuration* on page 147 for more information about the required additional configuration functions for emUSB-MSD.

3.4.3 Descriptors

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

Chapter 4

USB Core

This chapter describes the basic functions of the USB Core.

4.1 Overview

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

General information

To communicate with the host, the example application project includes a USB-specific header `USB.h` and the emUSB-Device source files, if you have a source version of emUSB-Device. These files contain 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_Init()` 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 105 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 163 and *USB_MSD_INST_DATA* on page 165 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 293 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 322 for information about the initialization structure that is required when you want to add a HID interface.
3. Provice 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	
USBD_GetState()	Returns the state of the USB device.
USBD_Init()	Initializes the emUSB-Device Core.
USBD_IsConfigured()	Checks if the USB device is configured.
USBD_Start()	Starts the emUSB-Device core.
USBD_Stop()	Stops the emUSB-Device core.
USBD_DeInit()	Deinitializes the emUSB-Device Core.
USB configuration functions	
USBD_AddDriver()	Adds a USB device driver to the emUSB-Device stack.
USBD_SetISRMgmFunc()	Register interrupt management functions.
USBD_SetAttachFunc()	Register USB attach function.
USBD_AddEP()	Returns an endpoint "handle" that can be used for the desired USB interface.
USBD_SetDeviceInfo()	Provice device information to the emUSB-Device stack.
USBD_SetAddFuncDesc()	Sets a callback for setting additional information into the configuration descriptor.
USBD_SetClassRequestHook()	Sets a callback to handle class setup requests.
USBD_SetVendorRequestHook()	Sets a callback to handle vendor setup requests.
USBD_SetIsSelfPowered()	Sets whether the device is self-powered or not.
USBD_SetMaxPower()	Sets the target device's current consumption.
USBD_SetOnEvent()	Register callback function for RX and TX events.
USBD_SetOnRxEP0()	Sets a callback to handle data read of endpoint 0.
USBD_SetOnSetupHook()	Sets a callback to handle EP0 setup packets.
USB__WriteEP0FromISR()	Writes data to a USB EP.
USBD_StallEP()	Stalls an endpoint.
USBD_WaitForEndOfTransfer()	Waits for a data transfer to be ended.
USB IAD functions	
USBD_EnableIAD()	Allows to combine multi-interface device classes with single-interface classes.
USB RemoteWakeUp functions	
USBD_SetAllowRemoteWakeUp()	Allows the device to publish that remote wakeup is available.
USBD_DoRemoteWakeup()	Performs a remote wakeup to the host.

Table 4.1: Target USB Core interface function list

4.2.1 USB basic functions

4.2.1.1 USBD_GetState()

Description

Returns the state of the USB device.

Prototype

```
int USBD_GetState(void);
```

Return value

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

USB state flags	
USB_STAT_ATTACHED	Device is attached. (Note 1)
USB_STAT_READY	Device is ready.
USB_STAT_ADDRESSED	Device is addressed.
USB_STAT_CONFIGURED	Device is configured.
USB_STAT_SUSPENDED	Device is suspended.

Additional information

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

Note 1:
Attached in a USB sense of the word does not mean that the device is physically connected to the PCvia 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.

4.2.1.2 USBD_Init()

Description

Initializes the USB device with its settings.

Prototype

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


4.2.1.3 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.

4.2.1.4 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.

4.2.1.5 USBD_Stop()

Description

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

Prototype

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


4.2.1.6 USBD_DeInit()

Description

De-initializes the complete USB stack.

Prototype

```
void USBD_DeInit(void);
```

Additional information

This function also calls USBD_Stop() internally.
Not all drivers have a DeInit callback function, if you need to use DeInit and your driver does not have the callback - please contact SEGGER.

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 `USBD_X_Config()` which is implemented in `USB_Config_*.c`.

Prototype

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

Additional information

To add the driver, use `USBD_AddDriver()` with the identifier of the compatible driver. Refer to the section *Available USB drivers* on page 428 for a list of supported devices and their valid identifiers.

Example

```
/*********************************************************************
*
*       USBD_X_Config
*/
void USBD_X_Config(void) {
  BSP_USB_Init();
  USB_DRIVER_LPC17xx_ConfigAddr(0x2008C000);  // USB controller of LPC1788
                                              // is located @ 0x2008C000
  USBD_AddDriver(&USB_Driver_NXPLPC17xx);
  USBD_SetISRMgmFuncs(_EnableISR, USB_OS_IncDI, USB_OS_DecRI);
}
```


4.2.2.2 USBD_SetISRMgmFunc()

Description

Register interrupt management functions.

Prototype

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

void USBD_SetISRMgmFuncs(USB_ENABLE_ISR_FUNC *pfEnableISR,
                         USB_INC_DI_FUNC *pfIncDI,
                         USB_DEC_RI_FUNC *pfDecRI);
```

Parameter	Description
pfEnableISR	Pointer to function that installes the interrupt service routine for USB interrupts.
pfIncDI	Pointer to function that increments interrupt disable count and disables interrupts, see also USB_OS_IncDI() .
pfDecRI	Pointer to function that that decrements interrupt disable count and enable interrupts if counter reaches 0, see also USB_OS_DecRI().

Table 4.2: USBD_SetISRMgmFunc() parameter list

Additional information

This function must be called within USBD_X_Config() function. See "Adding a driver to emUSB-Device".

Example

See USBD_X_Config().

4.2.2.3 USBD_SetAttachFunc()

Description

Register interrupt management functions.

Prototype

```
typedef void  USB_ATTACH_FUNC (void);

void USBD_SetAttachFunc(USB_ATTACH_FUNC *pfAttach);
```

Parameter	Description
pfAttach	Pointer to function that installes a hardware attach routine.

Table 4.3: USBD_SetAttachFunc() parameter list

Additional information

This function must be called within `USBD_X_Config()` function. See "Adding a driver to emUSB-Device".

Example

See `USBD_X_Config()`.

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);
```

Parameter	Description
InDir	Specifies the direction of the desired endpoint. 1 - IN 0 - OUT
TransferType	Specifies the transfer type of the endpoint. The following values are allowed: USB_TRANSFER_TYPE_BULK USB_TRANSFER_TYPE_ISO USB_TRANSFER_TYPE_INT
Interval	Specifies the interval in for the endpoint. This value can be zero for a bulk endpoint.
pBuffer	Pointer to a buffer that is used for OUT-transactions. For IN-endpoints this parameter must be NULL.
BufferSize	Size of the buffer.

Table 4.4: USBD_AddEP() parameter list

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.
The frequency is specified in frames. When using USB low/full-speed one frame is sent every millisecond. When using USB high-speed one (micro)frame is sent every 0.125 µs.
For an endpoint of type USB_TRANSFER_TYPE_ISO the interval has to be 1.
For an endpoint of type USB_TRANSFER_TYPE_INT the interval has to be between 1 and 255.
For endpoints of type USB_TRANSFER_TYPE_BULK the value holds no relevance and has to be set to 0.

4.2.2.5 USBD_SetDeviceInfo()

Description

Provides device information used during USB enumeration to the stack.

Prototype

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

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

Table 4.5: USBD_SetDeviceInfo() parameter list

Additional information

See 3.4.1.1 for a description of the structure

Example

See 3.4.1.1

4.2.2.6 USBD_SetAddFuncDesc()

Description

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

Prototype

```
void USBD_SetAddFuncDesc(USB_ADD_FUNC_DESC * pfAddDescFunc);
```

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

Table 4.6: USBD_SetAddFuncDesc() parameter list

Additional information

USB_ADD_FUNC_DESC is defined as follows:

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


4.2.2.7 USBD_SetClassRequestHook()

Description

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

Prototype

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

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

Table 4.7: USBD_SetClassRequestHook() parameter list

Additional information

Note that the callback will be called within an ISR.
If it is necessary to send data from the callback function through endpoint 0, use the function `USB__WriteEP0FromISR()`.
`USB_ON_CLASS_REQUEST` is defined as follows:

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


4.2.2.8 USBD_SetVendorRequestHook()

Description

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

Prototype

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

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

Table 4.8: USBD_SetClassRequestHook() parameter list

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


4.2.2.9 USBD_SetIsSelfPowered()

Description

Sets whether the device is self-powered or not.

Prototype

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

Parameter	Description
IsSelfPowered	0 - Device is not self-powered. 1 - Device is self-powered.

Table 4.9: USBD_SetClassRequestHook() parameter list

Additional information

This function has to be called before USBD_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.

Prototype

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

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

Table 4.10: USBD_SetMaxPower() parameter list

Additional information

This function shall be called before USBD_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);
```

Parameter	Description
EPIndex	Endpoint handle
pEventCb	Pointer to a USB_EVENT_CALLBACK structure.
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

Table 4.11: USBD_SetOnEvent() parameter list

Additional information

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 has the follwing 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

Table 4.12: Event callback function parameter list

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
USB_EVENT_DATA_READ	Some data was received from the host on the endpoint.
USB_EVENT_DATA_SEND	Some data was send to the host, so that (part of) the user write buffer may be reused by the application.
USB_EVENT_DATA_ACKED	Some data was acknowledged by the host.
USB_EVENT_READ_COMPLETE	The last read operation was completed.
USB_EVENT_READ_ABORT	A read transfer was aborted.
USB_EVENT_WRITE_ABORT	A write transfer was aborted.
USB_EVENT_WRITE_COMPLETE	All write operations were completed.

Table 4.13: USB events

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_GetNumBytesToWrite(_hInst) == 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;
USBD_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_SetOnRxEP0()

Description

Sets a callback to handle non-setup data presented on endpoint 0.

Prototype

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

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

Table 4.14: USBD_SetOnRxEP0() parameter list

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_RX_FUNC` is defined as follows:

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


4.2.2.13 USBD_SetOnSetupHook()

Description

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

Prototype

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

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

Table 4.15: USBD_SetClassRequestHook() parameter list

Additional information

Note that the callback will be called within an ISR.
If it is necessary to send data from the callback function through endpoint 0, use the function `USB__WriteEP0FromISR()`.
`USB_ON_SETUP` is defined as follows:
```
typedef int USB_ON_SETUP(const USB_SETUP_PACKET * pSetupPacket);
```


4.2.2.14 USB__WriteEP0FromISR()

Description

Writes data to a USB EP.

Prototype

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

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

Table 4.16: USB_WriteEP0FromISR() parameter list

4.2.3 USB control functions

4.2.3.1 USBD_StallEP()

Description

Stalls an endpoint.

Prototype

```
void USBD_StallEP(U8 EPIndex);
```

Parameter	Description
EPIndex	Endpoint handle that needs to be stalled.

Table 4.17: USBD_StallEP() parameter list

4.2.3.2 USBD_WaitForEndOfTransfer()

Description

Waits for a data transfer to complete.

Prototype

```
int USBD_WaitForEndOfTransfer(U8 EPIndex, unsigned ms);
```

Parameter	Description
EPIndex	Endpoint handle to wait for end of transfer.
ms	Timeout in milliseconds, 0 means infinite wait.

Table 4.18: USBD_WaitForEndOfTransfer() parameter list

Return value

0: Transfer completed.
1: Timeout occurred.

4.2.4 USB IAD functions

4.2.4.1 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

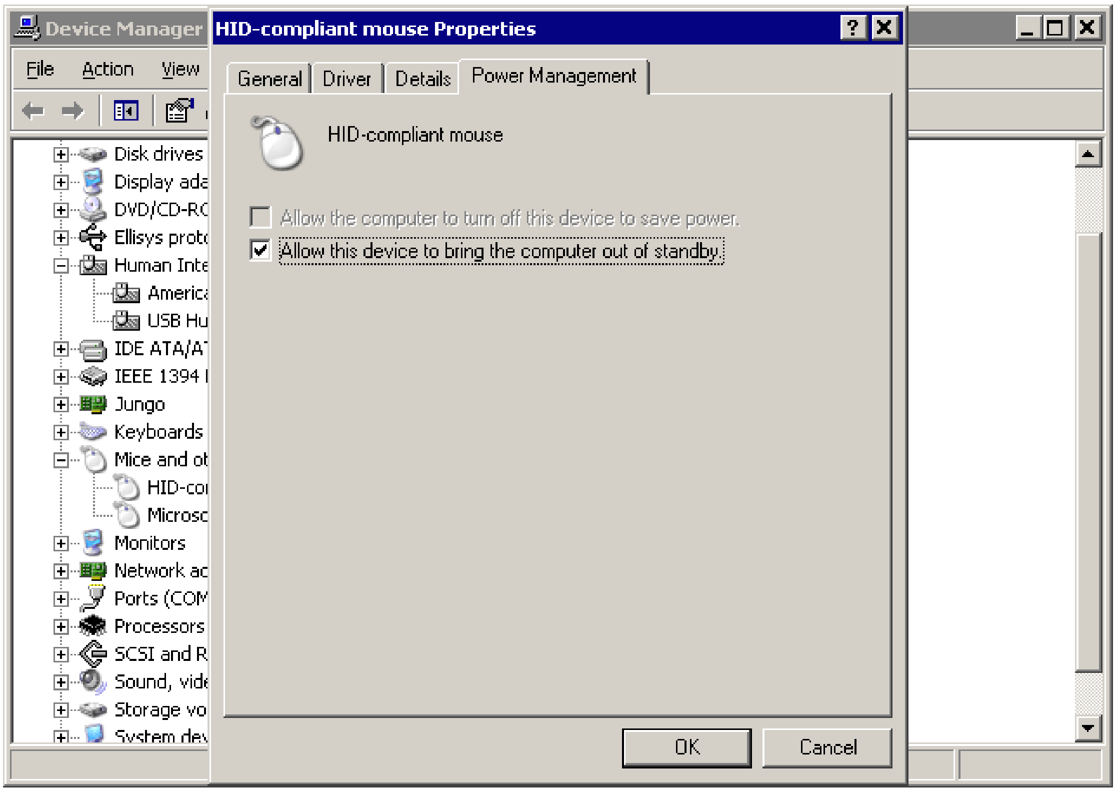
Simple device classes such as HID and MSD or BULK use only one interface descriptor to describe the class. The interface descriptor also contains the device class code. The CDC device class uses more than one interface descriptor to describe the class. The device class code will then be written into the device descriptor. It may be possible to add an interface which does not belong to the CDC class, but it may not be correctly recognized by the host, 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 the CDC component with any other component, please call `USBD_EnableIAD()` before adding the CDC component through `USBD_CDC_Add()`.

4.2.5 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. A sample is provided for adding such a dummy class.
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".



Mac OS X

Mac OS X supports remote wakeup for all device classes.

4.2.5.1 USBD_SetAllowRemoteWakeUp()

Description

Allows the device to publish that remote wake is available.

Prototype

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

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

Table 4.19: USBD_SetAllowRemoteWakeUp() parameter list

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.

4.2.5.2 USBD_DoRemoteWakeup()

Description

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

Prototype

```
void USBD_DoRemoteWakeUp(void);
```

Additional information

This function cannot be called from an ISR context

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 The Kernel mode driver (PC)

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

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

5.2.1 Why is a driver necessary?

In Microsoft's Windows operating systems, all communication with real hardware is implemented with *kernel-mode* drivers. Normal applications run in *user-mode*. In user mode, hardware access is not permitted. All access to hardware is done through the operating system and the operating system uses a kernel mode driver to access the actual hardware. In other words: every piece of hardware requires one or more kernel mode drivers to function. Windows supplies drivers for most common types of hardware, but it does not come with a generic bulk communication driver. It comes with drivers for certain classes of devices, such as keyboard, mouse and mass storage (for example, a USB stick). This makes it possible to connect a USB mouse without having to install a driver for it: Windows already has a driver for it.

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

5.2.2 Writing your own host driver

It is of course possible to write your own driver for your host system which will communicate with the emUSB-Device Bulk component on the target.
When writing your own driver, please take note that the bulk component uses an optimization where the zero-length-packet is not transmitted at the end of transfers which are a multiple of 2048 bytes. This optimization is designed to work with the emUSB-Device Windows driver, it can be disabled by adding the define USBD_BULK_REGULAR_2048_XFER to your USB_Conf.h.

5.2.3 Supported platforms

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

5.3 Installing the driver

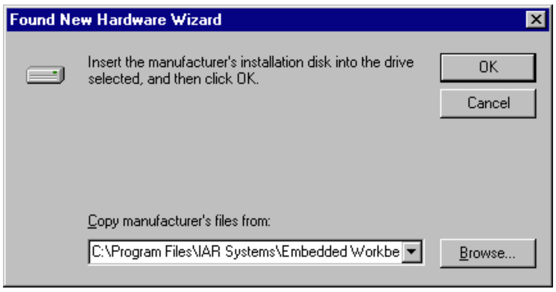
When the target device is plugged into the computer's USB port, or when the computer is first powered up after connecting the emUSB-Device device, Windows will detect the new hardware.

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

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

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

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



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

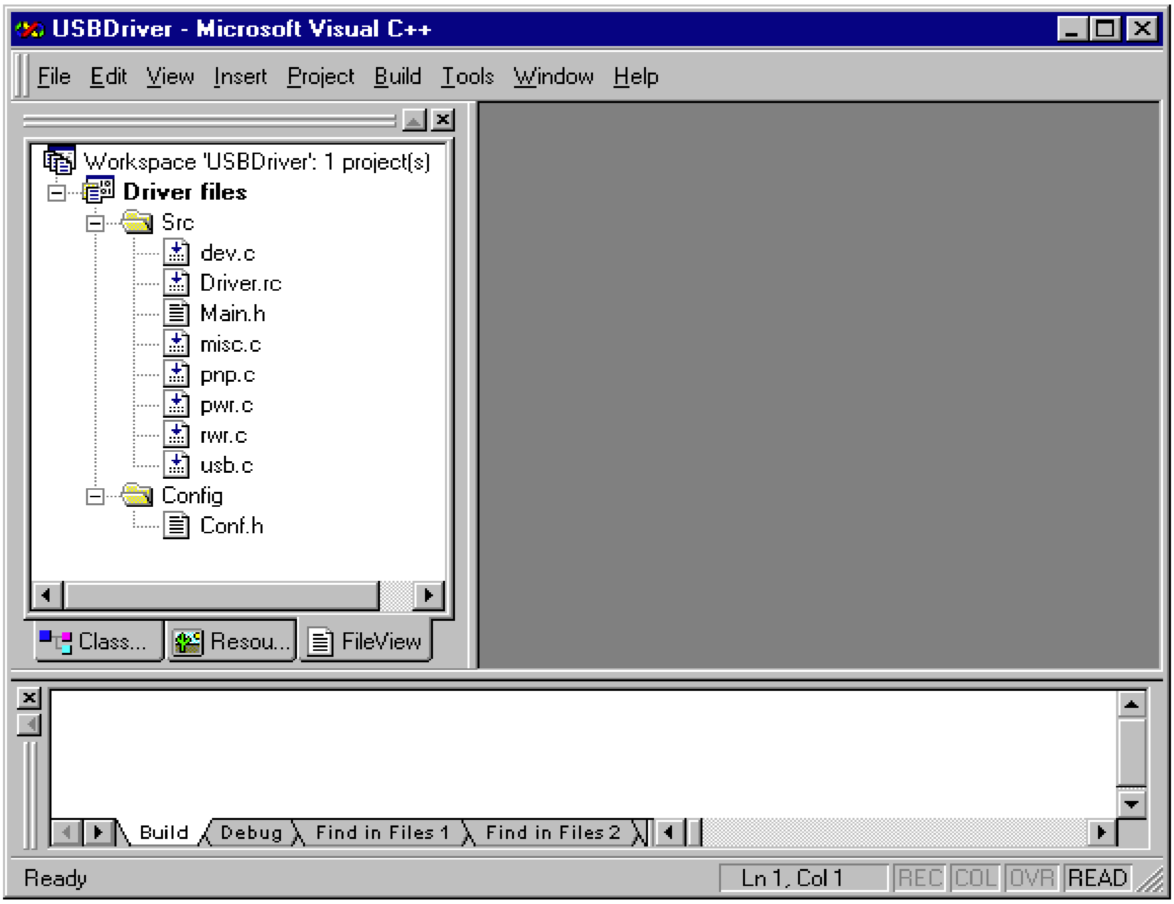
At this point, the installation is complete. Click the **Finish** button to dismiss the wizard.

5.3.1 Recompiling the driver

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

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

A workspace similar to the screenshot below is opened.



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

5.3.2 The .inf file

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

The shipped file is as follows:

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

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

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

[Manufacturer]
%MfgName%=DeviceList

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

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

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

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

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

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

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

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

[USBBULK.Files.Ext]
USBBulk.sys


;----------------------------------------------------------------;
```



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

red - required modifications
green - possible modifications

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

Replace the red marked positions with the personal Vendor ID (VID) and Product ID (PID). These changes must match the modifications in the configuration functions to work correctly.

The required modifications of the configuration functions are described in the section *Configuration* on page 40.

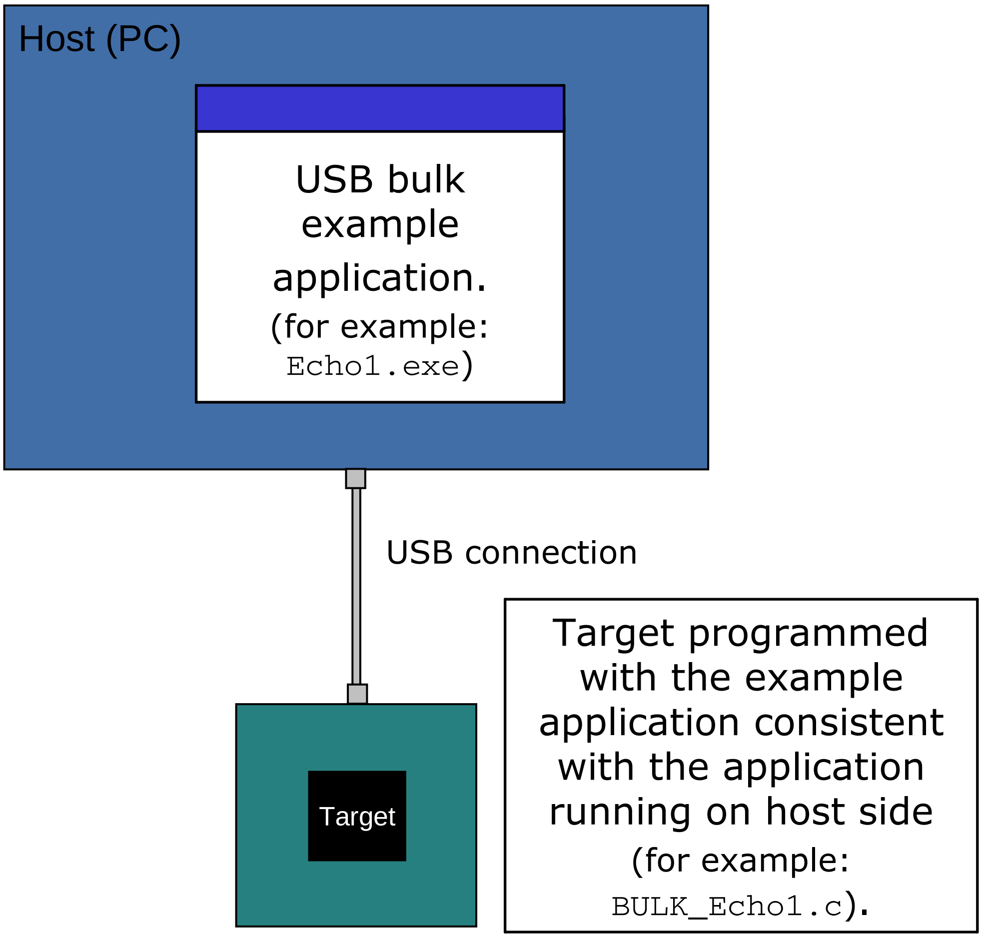
5.3.3 Configuration

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 section can later on be modified to match your real application. The configuration must only be modified if emUSB-Device should be used in a final product. Refer to section *Configuration* on page 40 for detailed information about the functions which must be adapted.

5.4 Example application

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

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



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

The following start application files are provided:

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

Table 5.1: Supplied sample applications

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

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

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

Table 5.2: Supplied host applications

For information how to compile the host examples refer to *Compiling the PC example application* on page 83.

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

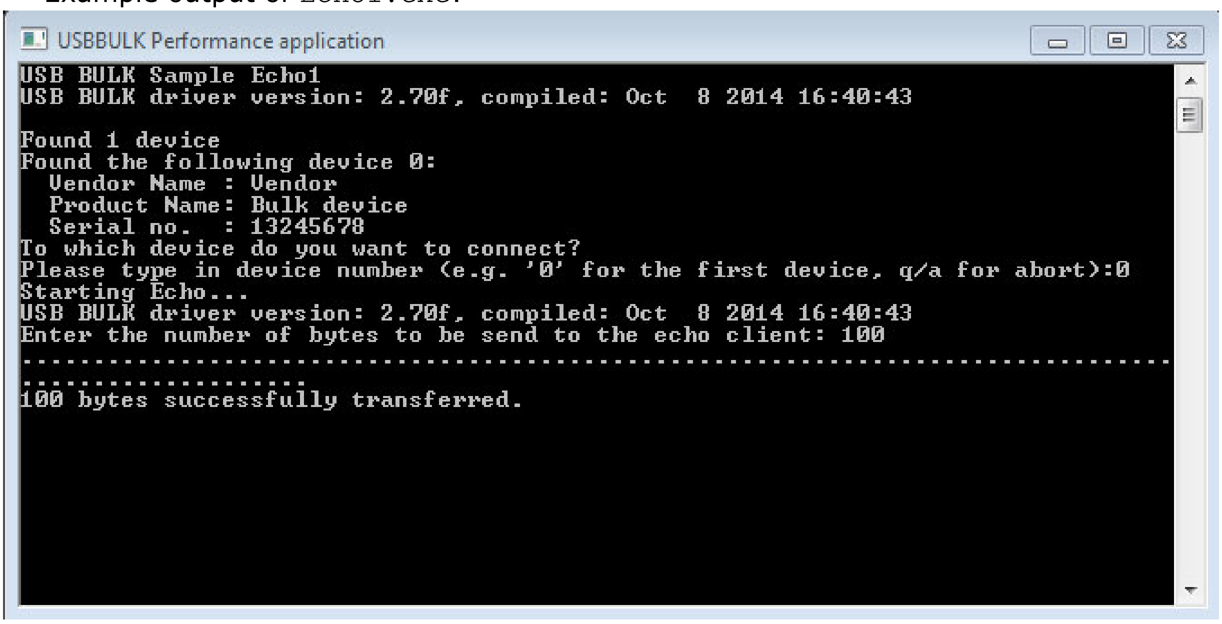
5.4.1 Running the example applications

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

To run one of the example applications, simply start the executable, for example by double clicking it. If the driver is not installed, the following message box should pop up.

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

Example output of `Echo1.exe`:



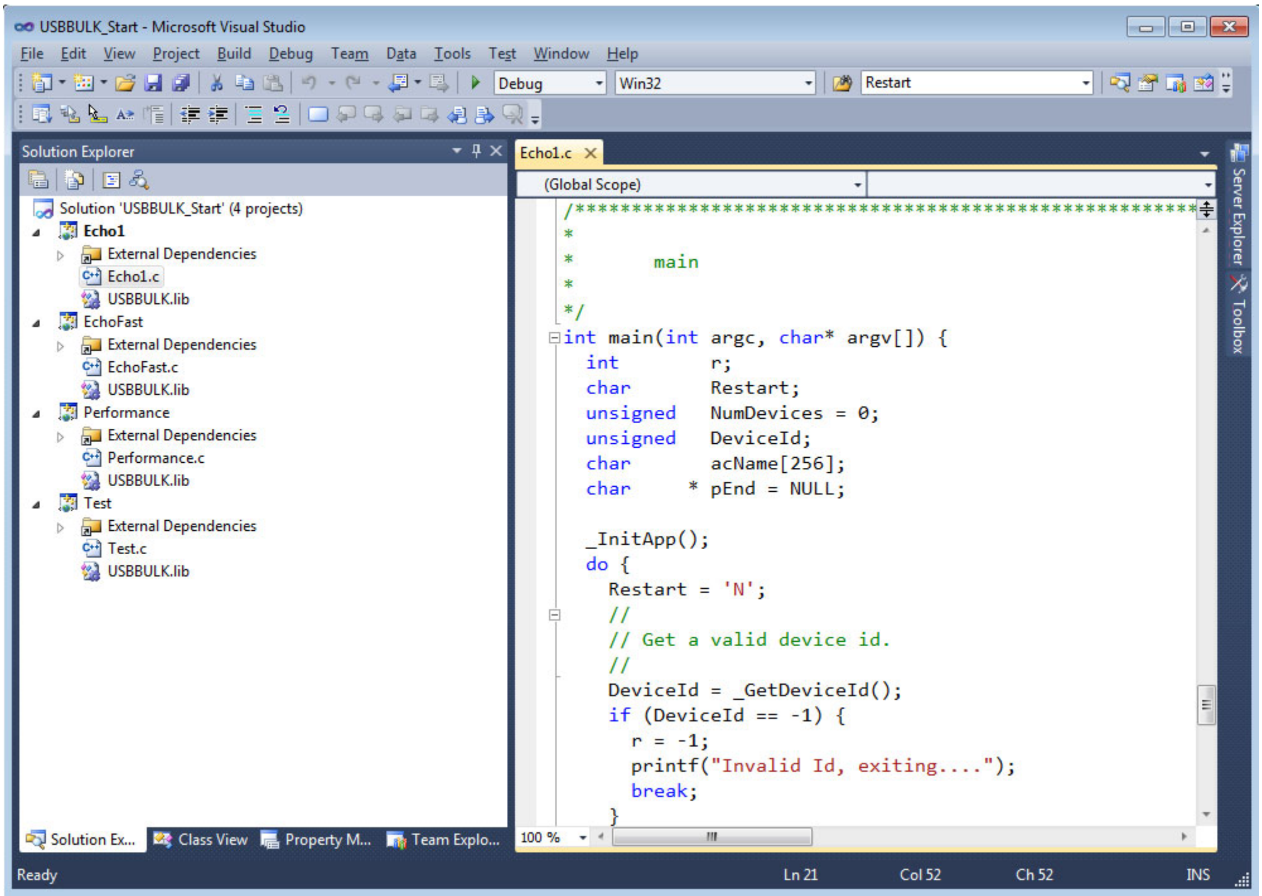
If the host example application can communicate with the emUSB-Device device, the example application will be in interactive mode for the `Echo1` and the `EchoFast` application. In case of an error, a message box is displayed.

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 (timeout)`	The PC sample application was not able to read.
`Wrong data read`	The result of the target sample application is not correct.

Table 5.3: List of error messages

5.4.2 Compiling the PC example application

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



The source code of the sample application is located in the subfolder `Bulk\WindowsAp-plication`. Open the file `USBBULK_Start.sln` and compile the source.

5.5 Target API

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

General information

To communicate with the host, the sample application project includes USB-specific header and source files (USB.h, USB_Main.c, USB_Setup.c, USB_Bulk.c, USB_Private.h). These files contain API functions to communicate with the USB host through the emUSB-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 host emUSB-Device kernel mode driver.
Therefore, all operations that need to write to or read from the emUSB-Device are handled internally by the provided API functions.

5.5.1 Target interface function list

Routine	Explanation
USB-Bulk functions	
USBD_BULK_Add()	Adds an USB-Bulk interface to emUSB-Device.
USBD_BULK_CancelRead()	Cancels a non-blocking read operation that is pending.
USBD_BULK_CancelWrite()	Cancels a non-blocking write operation that is pending.
USBD_BULK_GetNumBytesInBuffer()	Returns the number of byte in BULK-OUT buffer.
USBD_BULK_GetNumBytesRemToRead()	Returns the number of bytes which have to be read.
USBD_BULK_GetNumBytesRemToWrite()	Returns the number of bytes which have to be written.
USBD_BULK_Read()	USB-Bulk read.
USBD_BULK_ReadOverlapped()	Non-blocking version of USBD_BULK_Read().
USBD_BULK_Receive()	Read data from host and return immediately as soon as data has been received.
USBD_BULK_SetContinuousReadMode()	Enables continuous read mode.
USBD_BULK_SetOnRXEvent()	Adds a callback function for RX events.
USBD_BULK_SetOnTXEvent()	Adds a callback function for TX events.
USBD_BULK_TxIsPending()	Checks whether the IN endpoint is currently pending.
USBD_BULK_WaitForRX()	Waits for a non-blocking read operation that is pending.
USBD_BULK_WaitForTX()	Waits for a non-blocking write operation that is pending.
USBD_BULK_WaitForTXReady()	Wait until stack is ready to accept new write operation.
USBD_BULK_Write()	Starts a write operation.
USBD_BULK_WriteEx()	Starts a write operation that allows to specify whether a NULL packet shall be sent or not.
Data structures	
USB_BULK_INIT_DATA	Initialization structure which is required when adding a bulk interface.

Table 5.4: Target interface function list

5.5.2 USB-Bulk functions

5.5.2.1 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 );
```

Parameter	Description
pInitData	Pointer to USB_BULK_INIT_DATA structure.

Table 5.5: USBD_BULK_Add() parameter list

Return value

== 0xFFFFFFFF: New BULK Instance can not be created.
!= 0xFFFFFFFF: Handle to a valid BULK instance.

Additional information

USB_BULK_INIT_DATA is defined as follows:

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


5.5.2.2 USBD_BULK_CancelRead()

Description

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

Prototype

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

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().

Table 5.6: USBD_BULK_CancelRead() parameter list

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.

5.5.2.3 USBD_BULK_CancelWrite()

Description

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

Prototype

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

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().

Table 5.7: USBD_BULK_CancelWrite() parameter list

Additional information

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

5.5.2.4 USBD_BULK_GetNumBytesInBuffer()

Description

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

Prototype

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

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().

Table 5.8: USBD_BULK_GetNumBytesinBuffer() parameter list

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 `USBD_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 USBD_BULK_GetNumBytesInBuffer() it will return the number of bytes that are available in the internal buffer (49).

5.5.2.5 USBD_BULK_GetNumBytesRemToRead()

Description

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);
```

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().

Table 5.9: USBD_BULK_GetNumBytesRemToRead() parameter list

Return value

>= 0: Number of bytes which have not yet been read.
< 0: Error.

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);
  }
}
```


5.5.2.6 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);
```

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().

Table 5.10: USBD_BULK_GetNumBytesRemToWrite() parameter list

Return value

>= 0: Number of bytes which have not yet been written.
< 0: Error.

Additional information

Alternatively the blocking function USBD_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 (USBD_BULK_GetNumBytesRemToWrite(hInst) > 0) {
  USB_OS_Delay(50);
}
```


5.5.2.7 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 ms);
```

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

Table 5.11: USBD_BULK_Read() parameter list

Return value

== NumBytes: Requested data was succesfully read within the given timeout.
>= 0, < NumBytes: Timeout has occured.
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 occurrs.

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

5.5.2.8 USBD_BULK_ReadOverlapped()

Description

Reads data from the host asynchronously.

Prototype

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

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

Table 5.12: USBD_BULK_ReadOverlapped() parameter list

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, USBD_BULK_WaitForRX() needs to be called. Alternatively the function USBD_BULK_GetNumBytesRemToRead() can be called periodically to check whether all bytes have been read or not.
The read operation can be canceled using USBD_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 USBD_BULK_GetNumBytesRemToRead().

5.5.2.9 USBD_BULK_Receive()

Description

Reads data from host. The function blocks until any data has been received or a timeout occurs (if Timeout >= 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.

Prototype

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

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

Table 5.13: USBD_BULK_Receive() parameter list

Return value

> 0: Number of bytes that have been read.
== 0: A timeout occurred (if Timeout > 0) or no data in buffer (if Timeout < 0).
< 0: Error occurred.

Additional information

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

5.5.2.10 USBD_BULK_SetContinuousReadMode()

Description

Enables countinuous read mode in the USB stack. In this mode read transfers are processed by the USB stack independently of any USBD_BULK_Read...() function calls in the application as long as there is enough space in the internal buffer to receive another packet.

Prototype

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

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().

Table 5.14: USBD_BULK_SetContinuousReadMode() parameter list

Additional information

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

The USB stack will use the buffer that was provided by the application with USB_AddEP(). For optimal transfer speed, this buffer should have a size of at least 2 * MaxPacketSize. Normally MaxPacketSize for full-speed devices is 64 bytes and for high-speed devices 512 bytes.

Example

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


5.5.2.11 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);
```

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
pEventCb	Pointer to a USB_EVENT_CALLBACK structure.
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

Table 5.15: USBD_BULK_SetOnRXEvent() parameter list

Additional information

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 has the follwing 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

Table 5.16: Event callback function parameter list

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
USB_EVENT_DATA_READ	Some data was received from the host on the endpoint.
USB_EVENT_READ_COMPLETE	The last read operation was completed.
USB_EVENT_READ_ABORT	A read transfer was aborted.

Table 5.17: USB events

5.5.2.12 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);
```

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
pEventCb	Pointer to a USB_EVENT_CALLBACK structure.
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

Table 5.18: USBD_BULK_SetOnTXEvent() parameter list

Additional information

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 has the follwing 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

Table 5.19: Event callback function parameter list

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

Table 5.20: USB events

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 programm.


// 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..>
```

5.5.2.13 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);
```

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().

Table 5.21: USBD_BULK_TxIsPending() parameter list

Return value

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

5.5.2.14 USBD_BULK_WaitForRX()

Description

This function is used in combination with `USBD_BULK_ReadOverlapped()`, it waits for the read data transfer from the host to complete.

Prototype

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

Parameter	Description
`hInst`	Handle to a valid BULK instance, returned by USBD_BULK_Add().
`Timeout`	Timeout in milliseconds. 0 means infinite.

Table 5.22: USBD_BULK_WaitForRX() parameter list

Return value

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

Additional information

After starting the read operation via `USBD_BULK_ReadOverlapped()` this function can be used to wait until the transfer is complete.

5.5.2.15 USBD_BULK_WaitForTX()

Description

This function is used in combination with a non-blocking call to `USBD_BULK_Write()`, it waits for the write data transfer to the host to complete.

Prototype

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

Parameter	Description
`hInst`	Handle to a valid BULK instance, returned by USBD_BULK_Add().
`Timeout`	Timeout in milliseconds. 0 means infinite.

Table 5.23: USBD_BULK_WaitForTX() parameter list

Return value

0: Transfer completed.
1: Timeout occurred.

Additional information

After starting an asynchronous write operation via `Example` this function can be used to wait until the transfer is complete.

5.5.2.16 USBD_BULK_WaitForTXReady()

Description

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);
```

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

Table 5.24: USBD_BULK_WaitForTXReady() parameter list

Return value

0: A new asynchronous write data transfer will be accepted.
1: The write queue is full,
a call to USBD_BULK_Write() would return USB_STATUS_EP_BUSY.

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 (USBD_BULK_Write(hInst, pData, NumBytes, -1) < 0) {
    <.. error handling..>
  }
}
```


5.5.2.17 USBD_BULK_Write()

Description

Sends data to the USB host. Depending on the Timeout parameter, the function may block 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);
```

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

Table 5.25: USBD_BULK_Write() parameter list

Return value

== 0:	Successful started an asynchronous write transfer or a timeout has occured 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. A synchronous write transfer (Timeout >= 0) will always block until the transfer (including all pending transfers) are finished or a timeout occurs.

In order to synchronize, USBD_BULK_WaitForTX() needs to be called. Another synchronisation method would be to periodically call USBD_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 USBD_BULK_CancelWrite().

If pData == NULL and NumBytes == 0, a zero-length packet is send 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, not all data were written within 500ms ..>
} else {
  <.. write successful completed ..>
}
```

See also USBD_BULK_GetNumBytesRemToWrite().

5.5.2.18 USBD_BULK_WriteEx()

Description

This function behaves exactly like USBD_BULK_Write(). Additionaly sending of a zero lenght 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,
                      int Timeout
                        char Send0PacketIfRequired);
```

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
pData	Pointer to a buffer that contains the written data.
NumBytes	Number of bytes to write.
Timeout	Timeout in milliseconds. 0 means infinite. If Timeout is -1, the function returns immediately and the transfer is processed asynchronously.
Send0PacketIfRequired	Specifies that a zero-length packet shall be sent when the last data packet is a multiple of MaxPacketSize. Normally MaxPacketSize for full-speed devices is 64 bytes. For high-speed devices the normal packet size is between 64 and 512 bytes.

Table 5.26: USBD_BULK_WriteEx() parameter list

Return value

== 0:	Successful started an asynchronous write transfer or a timeout has occured 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, 0, 1);
// this will send 4 packets to the host with sizes: 512, 512, 512, 0
```


5.5.3 Data structures

5.5.3.1 USB_BULK_INIT_DATA

Description

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

Prototype

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

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

Table 5.27: USB_BULK_INIT_DATA elements

Example

Example excerpt from `BULK_Echo1.c`:

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

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


5.6 Host API

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

General information

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

Purpose of the USB Host API functions

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

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

Additional information can also be retrieved from the USB driver.

Notes

After software version 3.0 this chapter describes the host API which was previously described in a separate chapter - "Bulk Host API V2". The old API (Bulk Host API V1) is deprecated and will not be described in new versions of this document.

5.6.1 Bulk Host API V2 list

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

Function	Description
USB-Bulk basic functions	
USBBULK_Open()	Opens an existing device.
USBBULK_Close()	Closes an opened device.
USBBULK_Init()	Initializes the API module.
USBBULK_Exit()	Called on exit.
USBBULK_AddAllowedDeviceItem()	Sets the Vendor and Product IDs.
USB-Bulk direct input/output functions	
USBBULK_Read()	Reads from an opened device.
USBBULK_Write()	Writes data to the device.
USBBULK_WriteRead()	Writes and reads from the device.
USBBULK_CancelRead()	Cancels an initiated read.
USBBULK_ReadTimed()	Reads from an opened device with a timeout.
USBBULK_WriteTimed()	Writes data to the device with a time-out.
USBBULK_FlushRx()	Removes data from the receive buffer.
USB-Bulk control functions	
USBBULK_GetConfigDescriptor()	Returns the configuration descriptor of the device.
USBBULK_GetMode()	Returns the transfer mode of the device.
USBBULK_GetReadMaxTransferSize()	Returns the max size the driver can read at once.
USBBULK_GetWriteMaxTransferSize()	Returns the max size the driver can write at once.
USBBULK_ResetPipe()	Resets the pipes that are opened to the device.
USBBULK_ResetDevice()	Resets the device via a USB reset.
USBBULK_SetMode()	Sets the read and write mode of the device.
USBBULK_SetReadTimeout()	Sets the read timeout for an opened device.
USBBULK_SetWriteTimeout()	Sets the write timeout for an opened device.
USBBULK_GetEnumTickCount()	Returns the time when the USB device has been enumerated.
USBBULK_GetReadMaxTransferSizeDown()	Returns the max read transfer size of the device.
USBBULK_GetWriteMaxTransferSizeDown()	Returns the max write transfer size of the device.
USBBULK_SetReadMaxTransferSizeDown()	Sets the max read transfer size of the device.
USBBULK_SetWriteMaxTransferSizeDown()	Sets the max write transfer size of the device.
USBBULK_GetSN()	Gets the serial number of the device.
USBBULK_GetDevInfo()	Retrieves information about an opened USBBULK device.
USBBULK_GetProductName()	Returns the product name.
USBBULK_GetVendorName()	Returns the vendor name.

Table 5.28: Bulk Host API V2 function list

Function	Description
USB-Bulk general GET functions	
USBBULK_GetDriverCompileDate()	Gets the compile date of the driver.
USBBULK_GetDriverVersion()	Returns the driver version.
USBBULK_GetVersion()	Returns the USBBULK API version.
USBBULK_GetNumAvailableDevices()	Returns the number of available devices.
USBBULK_GetUSBId()	Returns the set Product and Vendor IDs.
Data structures	
USBBULK_DEV_INFO	Device information structure (Vendor ID, Product ID, SN, Device Name).

Table 5.28: Bulk Host API V2 function list

5.6.2 USB-Bulk Basic functions

5.6.2.1 USBBULK_Open()

Description

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

Prototype

```
USBBULK_API USB_BULK_HANDLE WINAPI USBBULK_Open (unsigned DevIndex);
```

Parameter	Description
`Id`	0..31 Device ID to be opened.

Table 5.29: USBBULK_Open() parameter list

Return value

!= 0: Handle to the opened device.
== 0: Device cannot be opened.

5.6.2.2 USBBULK_Close()

Description

Closes an opened device.

Prototype

```
USBBULK_API void WINAPI USBBULK_Close (USB_BULK_HANDLE hDevice);
```

Parameter	Description
hDevice	Handle to the device that shall be closed.

Table 5.30: USBBULK_Close() parameter list

5.6.2.3 USBBULK_Init()

Description

This function needs to be called before any other. This ensures that all structures and threads are initialized. It also sets a callback in order to be notified when a device is added or removed.

Prototype

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

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

Table 5.31: USBBULK_Init() parameter list

Example

```
U32       DeviceMask;

/*********************************************************************
*
*       _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 __stdcall _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);
<...>
}
```


5.6.2.4 USBBULK_Exit()

Description

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

Prototype

```
USBBULK_API void WINAPI 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.

5.6.2.5 USBBULK_AddAllowedDeviceItem()

Description

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

Prototype

```
USBBULK_API void WINAPI USBBULK_AddAllowedDeviceItem(U16 VendorId, U16
ProductId);
```

Parameter	Description
VendorId	The desired Vendor ID mask that shall be used with the USBBULK API
ProductId	The desired Product ID mask that shall be used with the USBBULK API

Table 5.32: USBBULK_SetUSBId() parameter list

Additional information

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

5.6.3 USB-Bulk direct input/output functions

5.6.3.1 USBBULK_Read()

Description

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

Prototype

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

Parameter	Description
hDevice	Handle to the opened device.
pBuffer	Pointer to a buffer that shall store the data.
NumBytes	Number of bytes to be read.

Table 5.33: USBBULK_Read() parameter list

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, cannot read from the device.

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.

5.6.3.2 USBBULK_Write()

Description

Writes data to the device.

Prototype

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

Parameter	Description
hDevice	Handle to the opened device.
pBuffer	Pointer to a buffer that contains the data.
NumBytes	Number of bytes to be written.

Table 5.34: USBBULK_Write() parameter list

Return value

== NumBytes:	All bytes have been successfully written.
> 0, < NumBytes:	Number of bytes that have been written. If short write transfers are not allowed (normal mode) this indicates a timeout.
== 0:	A timeout occurred, no data was written.
< 0:	Error, cannot write to the device.

Additional information

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.

If short write transfers are allowed (see USBBULK_SetMode()) the function returns after writing the minimal amount of data (either NumBytes or the maximal write transfer size, which can be read by using the function USBBULK_GetWriteMaxTransferSize()). Otherwise the function blocks until NumBytes were written. In both cases the function returns if a timeout occurs. The default timeout used can be set with USBBULK_SetWriteTimeout().

5.6.3.3 USBBULK_WriteRead()

Description

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

Prototype

```
USBBULK_API int WINAPI USBBULK_WriteRead(USB_BULK_HANDLE hDevice,
const void * pWrBuffer, int WrNumBytes, void * pRdBuffer, int RdNumBytes);
```

Parameter	Description
hDevice	Handle to the opened device.
pWrBuffer	Pointer to a buffer that contains the data.
WrNumBytes	Number of bytes to be written.
pRdBuffer	Pointer to a buffer that shall store the data.
RdNumBytes	Number of bytes to be read.

Table 5.35: USBBULK_WriteRead() parameter list

Return value

== NumBytes: All bytes have been successfully read after writing the data.
== 0: A timeout occurred during read.
< 0: Cannot read from the device after write.

Additional information

This function can not be used when short read mode (see USBBULK_SetMode()) is enabled .

5.6.3.4 USBBULK_CancelRead()

Description

This function cancels an initiated read operation.

Prototype

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

Parameter	Description
hDevice	Handle to the opened device.

Table 5.36: USBBULK_CancelRead() parameter list

5.6.3.5 USBBULK_ReadTimed()

Description

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

Prototype

```
USBBULK_API int WINAPI USBBULK_Read (USB_BULK_HANDLE hDevice,
                                     void * pBuffer,
                                     int NumBytes
                                     unsigned ms);
```

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

Table 5.37: USBBULK_ReadTimed() parameter list

Return value

> 0:	Number of bytes that have been read from device.
== 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.

5.6.3.6 USBBULK_WriteTimed()

Description

Writes data to the device within a given timeout.

Prototype

```
USBBULK_API int WINAPI USBBULK_Write (USB_BULK_HANDLE hDevice,
                                        const void * pBuffer,
                                        int NumBytes
                                        unsigned ms);
```

Parameter	Description
hDevice	Handle to the opened device.
pBuffer	Pointer to a buffer that contains the data.
NumBytes	Number of bytes to be written.
ms	Timeout in milliseconds.

Table 5.38: USBBULK_WriteTimed() parameter list

Return value

== NumBytes:	All bytes have been successfully written.
> 0, < NumBytes:	Number of bytes that have been written. If short write transfers are not allowed (normal mode) this indicates a timeout.
== 0:	A timeout occurred, no data was written.
< 0:	Error, cannot write to the device.

Additional information

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

If short write transfers are allowed (see USBBULK_SetMode()) the function returns after writing the minimal amount of data (either NumBytes or the maximal write transfer size, which can be read by using the function USBBULK_GetWriteMaxTransferSize()). Otherwise the function blocks until NumBytes were written. In both cases the function returns if a timeout occurs.

5.6.3.7 USBBULK_FlushRx()

Description

This function removes all data which was cached by the API from the internal receive buffer.

Prototype

```
USBBULK_API int WINAPI USBBULK_FlushRx (USB_BULK_HANDLE hDevice);
```

Parameter	Description
hDevice	Handle to the opened device.

Table 5.39: USBBULK_FlushRx() parameter list

5.6.4 USB-Bulk Control functions

5.6.4.1 USBBULK_GetConfigDescriptor()

Description

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

Prototype

```
USBBULK_API int WINAPI USBBULK_GetConfigDescriptor(USB_BULK_HANDLE hDevice,
                                                    void* pBuffer, int Size);
```

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

Table 5.40: USBBULK_GetConfigDescriptor() parameter list

Return value

== 0: Operation failed. Either an invalid handle was used or the buffer that shall store the config descriptor is too small.
!= 0: The operation was successful.

If the function succeeds, the buffer pointed by pBuffer contains the USB target device configuration descriptor.

5.6.4.2 USBBULK_GetMode()

Description

Returns the current mode of the device.

Prototype

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

Parameter	Description
hDevice	Handle to an opened device.

Table 5.41: USBBULK_GetMode() parameter list

Return value

USBBULK_MODE_BIT_ALLOW_SHORT_READ:
Short read mode is enabled.

USBBULK_MODE_BIT_ALLOW_SHORT_WRITE:
Short write mode is enabled.

0: Normal mode is set.

Additional information

A combination of both modes is possible (bitwise OR).

5.6.4.3 USBBULK_GetReadMaxTransferSize()

Description

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

Prototype

```
USBBULK_API unsigned WINAPI USBBULK_GetReadMaxTransferSize(USB_BULK_HANDLE
                                                           hDevice);
```

Parameter	Description
hDevice	Handle to an opened device.

Table 5.42: USBBULK_GetReadMaxTransferSize() parameter list

Return value

== 0: Operation failed. Either an invalid handle was used or the transfer size cannot be read.
!= 0: The operation was successful.

5.6.4.4 USBBULK_GetWriteMaxTransferSize()

Description

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

Prototype

```
USBBULK_API unsigned WINAPI USBBULK_GetWriteMaxTransferSize(USB_BULK_HANDLE
                                                            hDevice);
```

Parameter	Description
hDevice	Handle to an opened device.

Table 5.43: USBBULK_GetWriteMaxTransferSize() parameter list

Return value

== 0: Operation failed. Either an invalid handle was used or the transfer size cannot be read.
!= 0: The operation was successful.

5.6.4.5 USBBULK_ResetPipe()

Description

Resets the pipes that are opened to the device.
It also flushes any data the USB bulk driver would cache.

Prototype

```
USBBULK_API int WINAPI USBBULK_ResetPipe(USB_BULK_HANDLE hDevice);
```

Parameter	Description
hDevice	Handle to an opened device.

Table 5.44: USBBULK_ResetPipe() parameter list

Return value

== 0: Operation failed. Either an invalid handle was used or the pipes cannot be flushed.
!= 0: The operation was successful.

5.6.4.6 USBBULK_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

```
USBBULK_API int WINAPI USBBULK_ResetDevice(USB_BULK_HANDLE hDevice);
```

Parameter	Description
hDevice	Handle to an opened device.

Table 5.45: USBBULK_ResetDevice() parameter list

Return value

== 0: Operation failed. Either an invalid handle was used or the pipes cannot be flushed.
!= 0: The operation was successful.

Additional information

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

5.6.4.7 USBBULK_SetMode()

Description

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

Prototype

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

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

Table 5.46: USBBULK_SetMode() parameter list

Return value

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

Additional information

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, which can be read by using the function USBBULK_GetWriteMaxTransferSize()).

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);
}
```


5.6.4.8 USBBULK_SetReadTimeout()

Description

Sets the default read timeout for an opened device.

Prototype

```
USBBULK_API void WINAPI USBBULK_SetReadTimeout(USB_BULK_HANDLE hDevice,
                                               int Timeout);
```

Parameter	Description
hDevice	Handle to the opened device.
Timeout	Timeout in milliseconds.

Table 5.47: USBBULK_SetReadTimeout() parameter list

5.6.4.9 USBBULK_SetWriteTimeout()

Description

Sets a default write timeout for an opened device.

Prototype

```
USBBULK_API void WINAPI USBBULK_SetWriteTimeout (USB_BULK_HANDLE hDevice,
                                                 int Timeout);
```

Parameter	Description
hDevice	Handle to the opened device.
Timeout	Timeout in milliseconds.

Table 5.48: USBBULK_SeWriteTimeout() parameter list

5.6.4.10 USBBULK_GetEnumTickCount()

Description

Returns the time when the USB device was enumerated.

Prototype

```
USBBULK_API U32 WINAPI USBBULK_GetEnumTickCount(USB_BULK_HANDLE hDevice);
```

Parameter	Description
hDevice	Handle to an opened device.

Table 5.49: USBBULK_GetEnumTickCount() parameter list

Return value

The time when the USB device was enumerated by the driver given in Windows timer ticks (normally 1 ms. ticks).

5.6.4.11 USBBULK_GetReadMaxTransferSizeDown()

Description

Returns the maximum transfer size the driver supports when reading data from the device. In normal cases the maximum transfer size will be 2048 bytes. As this is a multiple of the maximum packet size, it is necessary that the device does not send a NULL-packet in this case. The Windows USB stack will stop reading data from the USB bus as soon as it reads all requested bytes.

Prototype

```
USBBULK_API U32 WINAPI USBBULK_GetReadMaxTransferSizeDown(USB_BULK_HANDLE
                                                          hDevice);
```

Parameter	Description
hDevice	Handle to an opened device.

Table 5.50: USBBULK_GetReadMaxTransferSizeDown() parameter list

Return value

!= 0: Max transfer size the driver will read from device.
== 0 : The transfer size cannot be read.

5.6.4.12 USBBULK_GetWriteMaxTransferSizeDown()

Description

Returns the maximum transfer size the driver will accept when writing data to the device.

Prototype

```
USBBULK_API U32 WINAPI USBBULK_GetWriteMaxTransferSizeDown(USB_BULK_HANDLE
                                                            hDevice);
```

Parameter	Description
hDevice	Handle to an opened device.

Table 5.51: USBBULK_GetWriteMaxtransferSizeDown() parameter list

Return value

== 0: Operation failed. Either an invalid handle was used or the transfer size cannot be read.
!= 0: The operation was successful.

5.6.4.13 USBBULK_SetReadMaxTransferSizeDown()

Description

Sets the number of bytes the driver will write down to the device at once.

Prototype

```
USBBULK_API unsigned WINAPI USBBULK_SetReadMaxTransferSizeDown(
                            USB_BULK_HANDLE hDevice,
                            U32 TransferSize);
```

Parameter	Description
hDevice	Handle to an opened device.
TransferSize	The number of bytes the driver will set as maximum.

Table 5.52: USBBULK_SetReadMaxTransferSizeDown() parameter list

Return value

== 0: Operation failed. Either an invalid handle was used or the mode cannot be set.
!= 0: The operation was successful.

5.6.4.14 USBBULK_SetWriteMaxTransferSizeDown()

Description

Sets the number of bytes the driver will write down to the device at once.

Prototype

```
USBBULK_API unsigned WINAPI USBBULK_SetWriteMaxTransferSizeDown(
                            USB_BULK_HANDLE hDevice,
                            U32 TransferSize);
```

Parameter	Description
hDevice	Handle to an opened device.
TransferSize	The number of bytes the driver will set as maximum.

Table 5.53: USBBULK_SetWriteMaxTransferSizeDown() parameter list

Return value

== 0: Operation failed. Either an invalid handle was used or the mode cannot be set.
!= 0: Max transfer size the driver will read from device.

5.6.4.15 USBBULK_GetSN()

Description

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

Prototype

```
USBBULK_API unsigned WINAPI USBBULK_GetSN(USB_BULK_HANDLE hDevice,
                                           U8 * pBuffer, unsigned NumBytes);
```

Parameter	Description
hDevice	Handle to an opened device.
pBuffer	Pointer to a buffer which shall store the serial number of the device.
NumBytes	Size of the buffer given in bytes.

Table 5.54: USBBULK_GetSN() parameter list

Return value

== 0: Operation failed. Either an invalid handle was used or the serial number cannot be read.
!= 0: The operation was successful.

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

5.6.4.16 USBBULK_GetDevInfo()

Description

Retrieves information about an opened USBBULK device.

Prototype

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

Parameter	Description
hDevice	Handle to the opened device.
pDevInfo	Pointer to a device info structure.

Table 5.55: USBBULK_GetDevInfo() parameter list

5.6.4.17 USBBULK_GetProductName()

Description

Retrieves the product name of an opened USBBULK device.

Prototype

```
USBBULK_API void WINAPI USBBULK_GetProductName(USB_BULK_HANDLE hDevice,
                                                char * sProductName,
                                                unsigned BufferSize);
```

Parameter	Description
hDevice	Handle to the opened device.
sProductName	Pointer to a buffer where the product name shall be saved.
BufferSize	Size of the product name buffer.

Table 5.56: USBBULK_GetProductName() parameter list

5.6.4.18 USBBULK_GetVendorName()

Description

Retrieves the vendor name of an opened USBBULK device.

Prototype

```
USBBULK_API int WINAPI USBBULK_GetVendorName(USB_BULK_HANDLE hDevice,
                                              char * sVendorName,
                                              unsigned BufferSize);
```

Parameter	Description
hDevice	Handle to the opened device.
sVendorName	Pointer to a buffer where the vendor name shall be saved.
BufferSize	Size of the vendor name buffer.

Table 5.57: USBBULK_GetVendorName() parameter list

5.6.5 USB-Bulk general GET functions

5.6.5.1 USBBULK_GetDriverCompileDate()

Description

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

Prototype

```
USBBULK_API unsigned WINAPI USBBULK_GetDriverCompileDate(char * s,
                                                         unsigned Size);
```

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

Table 5.58: USBBULK_GetDriverCompileDate() parameter list

Return value

== 0: Operation failed. The buffer that shall store the string is too small.
!= 0: The operation was successful.

If the function succeeds, the return value is nonzero and the buffer pointed by s contains the compile date and time of the emUSB-Device driver in the standard format: mm dd yyyy hh:mm:ss

5.6.5.2 USBBULK_GetDriverVersion()

Description

Returns the driver version of the driver, if the driver is loaded. Otherwise the function will return 0, as it can only determine the driver version when the driver is loaded.

Prototype

```
USBBULK_API unsigned WINAPI USBBULK_GetDriverVersion(void);
```

Return value

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

`<Major Version><Minor Version><Subversion>`. 24201 (Mmmrr) means 2.42a
If the function fails, the return value is zero; the version could not be retrieved.

5.6.5.3 USBBULK_GetVersion()

Description

Returns the USBBULK API version.

Prototype

```
USBBULK_API unsigned WINAPI USBBULK_GetVersion(void);
```

Return value

The version of the USBBULK API in the following format:
<Major Version><Minor Version><Subversion>. 24201 (Mmmrr) means 2.42a

5.6.5.4 USBBULK_GetNumAvailableDevices()

Description

Returns the number of connected USB-Bulk devices.

Prototype

```
USBBULK_API unsigned WINAPI USBBULK_GetNumAvailableDevices(U32 * pMask);
```

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

Table 5.59: USBBULK_GetNumAvailableDevices() parameter list

Return value

The return value is the number of available devices running emUSB-Device-Bulk. 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.

5.6.5.5 USBBULK_GetUSBId()

Description

Returns the set Product and Vendor ID mask that is used with the USBBULK API.

Prototype

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

Parameter	Description
hDevice	Handle to the opened device.
pVendorId	Pointer to a U16 variable that will store the Vendor ID.
pProductId	Pointer to a U16 variable that will store the Product ID.

Table 5.60: USBBULK_GetUSBId() parameter list

5.6.6 Data structures

5.6.6.1 USBBULK_DEV_INFO

Description

A structure which can hold the relavant information about a device.

Prototype

```
typedef struct _USBBULK_DEV_INFO {
  U16  VendorId;
  U16  ProductId;
  char acSN[256];
  char acDevName[256];
} USBBULK_DEV_INFO;
```

Member	Description
VendorId	An U16 which holds the device Vendor ID.
ProductId	An U16 which holds the device Product ID.
acSN	Array of chars which holds the serial number of the device.
acDevName	Array of chars which holds the device name.

Table 5.61: USBBULK_DEV_INFO elements

Chapter 6

Mass Storage Device Class (MSD)

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

6.1 Overview

The Mass Storage Device (MSD) is a USB class protocol defined by 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

6.2 Configuration

6.2.1 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.

6.2.2 Final configuration

The configuration must only be modified, when emUSB-Device is deployed in your final product. Refer to *Configuration* on page 40 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' though '9' or 'A' through 'F').

6.2.3 Class specific configuration functions

Beside the generic emUSB-Device configuration functions (*Configuration* on page 40), 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 it's 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);
```


6.2.4 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.

6.2.4.1 MSD_Start_StorageRAM.c in detail

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

```
/* MainTask() - excerpt from USB_MSD_Start_StorageRAM.c */

void MainTask(void);
void MainTask(void) {
  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 `USB_Init()`. The function `_AddMSD()` configures all required endpoints and assigns the used storage medium to the MSD component.

```
/* _AddMSD() - excerpt from MSD_Start_StorageRAM.c */

static void _AddMSD(void) {
  static U8 _abOutBuffer[USB_MAX_PACKET_SIZE];
  USB_MSD_INIT_DATA     InitData;
  USB_MSD_INST_DATA     InstData;

  InitData.EPIn  = USBD_AddEP(1, USB_TRANSFER_TYPE_BULK,
                              USB_MAX_PACKET_SIZE, NULL, 0);
  InitData.EPOut = USBD_AddEP(0, USB_TRANSFER_TYPE_BULK, USB_MAX_PACKET_SIZE,
                              _abOutBuffer, USB_MAX_PACKET_SIZE);
  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 Kbytes (46 sectors with a sector size of 512 bytes). You can increase the size of the RAM disk by modifying the macros `MSD_RAM_NUM_SECTORS` and `MSD_RAM_SECTOR_SIZE` (in multiples of 512), but the size must be at least 23 Kbytes otherwise a Windows host cannot format the disk.

```
/* AddMSD() - excerpt from MSD_Start_StorageRAM.c */

#define MSD_RAM_NUM_SECTORS  46
#define MSD_RAM_SECTOR_SIZE  512
```


6.3 Target API

Function	Description
API functions	
USBD_MSD_Add()	Adds an MSD-class interface to the USB stack.
USBD_MSD_AddUnit()	Adds a mass storage device to the emUSB-Device-MSD.
USBD_MSD_AddCDRom()	Adds a CD-ROM device to the emUSB-Device-MSD.
USBD_MSD_SetPreventAllowRemovalHook()	Sets a callback function to prevent/ allow removal of storage medium.
USBD_MSD_SetPreventAllowRemovalHookEx()	Sets a callback function to prevent/ allow removal of storage medium.
USBD_MSD_SetReadWriteHook()	Sets a callback function which is called with every read or write access to the storage medium.
USBD_MSD_Task()	Handles the MSD-specific protocol.
USBD_MSD_SetStartStopUnitHook()	Sets a callback function for the START STOP UNIT command.
Extended API functions	
USBD_MSD_Connect()	Connects the storage medium to the MSD.
USBD_MSD_Disconnect()	Disconnects the storage medium from the MSD.
USBD_MSD_RequestDisconnect()	Sets the DisconnectRequest flag.
USBD_MSD_UpdateWriteProtect()	Updates the IsWriteProtected flag for a storage medium.
USBD_MSD_WaitForDisconnection()	Waits for disconnection while timeout is not reached.
Data structures	
USB_MSD_INIT_DATA	emUSB-Device-MSD initialization structure that is needed when adding an MSD interface.
USB_MSD_INFO	emUSB-Device-MSD storage information.
USB_MSD_INST_DATA	Structure that is used when adding a device to emUSB-Device-MSD.
PREVENT_ALLOW_REMOVAL_HOOK	Callback invoked when the storage medium is removed.
PREVENT_ALLOW_REMOVAL_HOOK_EX	Callback invoked when the storage medium is removed.
READ_WRITE_HOOK	Callback invoked when accessing the storage medium.
USB_MSD_INST_DATA_DRIVER	Structure that is passed to the driver.
USB_MSD_STORAGE_API	Structure that contains callbacks to the storage driver.
START_STOP_UNIT_HOOK	Callback invoked when the START STOP UNIT command is received.

Table 6.1: List of emUSB-Device MSD interface functions and data structures

6.3.1 API functions

6.3.1.1 USBD_MSD_Add()

Description

Adds an MSD-class interface to the USB stack.

Prototype

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

Parameter	Description
pInitData	Pointer to a USB_MSD_INIT_DATA structure.

Table 6.2: USBD_MSD_Add() parameter list

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. Refer to *USB_MSD_INIT_DATA* on page 163 for more information.

6.3.1.2 USBD_MSD_AddUnit()

Description

Adds a mass storage device to emUSB-Device-MSD.

Prototype

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

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

Table 6.3: USBD_MSD_AddUnit() parameter list

Additional information

It is necessary to call this function immediately after USBD_MSD_Add().
This function will then add an R/W storage device such as a hard drive, MMC/SD cards or NAND flash etc., to emUSB-Device-MSD, which then will be used to exchange data with the host. The structure USB_MSD_INST_DATA must be initialized before USBD_MSD_AddUnit() is called. Refer to *USB_MSD_INST_DATA* on page 165 for more information.

6.3.1.3 USBD_MSD_AddCDRom()

Description

Adds a CD-ROM device to emUSB-Device-MSD.

Prototype

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

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

Table 6.4: USBD_MSD_AddCDRom() parameter list

Additional information

Similar to USBD_MSD_AddUnit(), this function should be called after USBD_MSD_Add(). The structure USB_MSD_INST_DATA must be initialized before USBD_MSD_AddCDRom() is called. Refer to *USB_MSD_INST_DATA* on page 165 for more information.

6.3.1.4 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)
```

Parameter	Description
pfOnPreventAllowRemoval	Pointer to the callback function `PREVENT_ALLOW_REMOVAL_HOOK`. For detailed information about the function pointer, refer to *PREVENT_ALLOW_REMOVAL_HOOK* on page 167.

Table 6.5: USBD_MSD_SetPreventAllowRemovalHook() parameter list

Additional information

The callback is called within the MSD task context. The callback must not block.

6.3.1.5 USBD_MSD_SetPreventAllowRemovalHookEx()

Description

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

Prototype

```
void USBD_MSD_SetPreventAllowRemovalHookEx(U8 Lun,
                      PREVENT_ALLOW_REMOVAL_HOOK_EX *
                      pfOnPreventAllowRemovalEx)
```

Parameter	Description
pfOnPreventAllowRemovalEx	Pointer to the callback function `PREVENT_ALLOW_REMOVAL_HOOK_EX`. For detailed information about the function pointer, refer to *PREVENT_ALLOW_REMOVAL_HOOK_EX* on page 168.

Table 6.6: USBD_MSD_SetPreventAllowRemovalHookEx() parameter list

Additional information

The callback is called within the MSD task context. The callback must not block.

6.3.1.6 USBD_MSD_SetReadWriteHook()

Description

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

Prototype

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

Parameter	Description
pfOnReadWrite	Pointer to the callback function READ_WRITE_HOOK. For detailed information about the function pointer, refer to *READ_WRITE_HOOK* on page 169.

Table 6.7: USBD_MSD_SetReadWriteHook() parameter list

6.3.1.7 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 `USBD_MSD_Task()` should be called. When the device is detached or is suspended, `USBD_MSD_Task()` will return.

6.3.2 Extended API functions

6.3.2.1 USBD_MSD_Connect()

Description

Connects the storage medium to the MSD module.

Prototype

```
void USBD_MSD_Connect(U8 Lun);
```

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

Table 6.8: USBD_MSD_Connect() parameter list

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 USBD_MSD_Disconnect() to carry out file system operations on the device application side.

6.3.2.2 USBD_MSD_Disconnect()

Description

Disconnects the storage medium from the MSD module.

Prototype

```
void USBD_MSD_Disconnect(U8 Lun);
```

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

Table 6.9: USBD_MSD_Disconnect() parameter list

Additional information

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

See USBD_MSD_RequestDisconnect() and USBD_MSD_WaitForDisconnection() for a graceful disconnection method.

6.3.2.3 USBD_MSD_RequestDisconnect()

Description

Sets the DisconnectRequest flag.

Prototype

```
void USBD_MSD_RequestDisconnect(U8 Lun);
```

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

Table 6.10: USBD_MSD_RequestDisconnect() parameter list

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, USBD_MSD_Connect() shall be called.

6.3.2.4 USBD_MSD_UpdateWriteProtect()

Description

Updates the IsWriteProtected flag for a storage medium.

Prototype

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

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

Table 6.11: USBD_MSD_UpdateWriteProtect() parameter list

Additional information

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

6.3.2.5 USBD_MSD_WaitForDisconnection()

Description

Waits for disconnection while timeout is not reached.

Prototype

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

Parameter	Description
Lun	0-based index for the unit number. Using only one storage medium, this parameter is 0.
TimeOut	Timeout given in timer ticks (not milliseconds!).

Table 6.12: USBD_MSD_WaitForDisconnection() parameter list

Return value

0: Error, timeout reached. Storage medium is not disconnected.
1: Success, storage medium is 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.

6.3.2.6 USBD_MSD_SetStartStopUnitHook()

Description

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

Prototype

```
void USBD_MSD_SetStartStopUnitHook(U8 Lun,
                                    START_STOP_UNIT_HOOK * pfOnStartStopUnit)
```

Parameter	Description
`pfOnStartStopUnit`	Pointer to the callback function START_STOP_UNIT_HOOK. For detailed information about the function pointer, refer to *PREVENT_ALLOW_REMOVAL_HOOK_EX* on page 168.

Table 6.13: USBD_MSD_SetStartStopUnitHook() parameter list

6.3.3 Data structures

6.3.3.1 USB_MSD_INIT_DATA

Description

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

Prototype

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

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

Table 6.14: USB_MSD_INIT_DATA elements

Additional Information

This structure holds the endpoints that should be used with the MSD interface. Refer to *USBD_AddDriver()* on page 52 for more information about how to add an endpoint.

6.3.3.2 USB_MSD_INFO

Description

emUSB-Device-MSD storage interface.

Prototype

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

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

Table 6.15: USB_MSD_INFO elements

6.3.3.3 USB_MSD_INST_DATA

Description

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

Prototype

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

Member	Description
pAPI	Pointer to a structure that holds the storage device driver API.
DriverData	Driver data that are passed to the storage driver. Refer to *USB_MSD_INST_DATA_DRIVER* on page 170 for detailed information about how to initialize this structure.
DeviceType	Determines the type of the device.
IsPresent	Determines if the medium is storage is present. For non-removable devices always 1.
pfHandleCmd	Optional pointer to a callback function which handles SCSI commands. `typedef U8 (USB_MSD_HANDLE_CMD)(U8 Lun);`
IsWriteProtected	Specifies whether the storage medium shall be write-protected.
pLunInfo	Pointer to a USB_MSD_LUN_INFO structure. Filling this structure is mandatory for each LUN.

Table 6.16: USB_MSD_INST_DATA elements

Additional Information

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

6.3.3.4 USB_MSD_LUN_INFO

Description

Structure that is used when adding a logical volume to emUSB-Device-MSD.

Prototype

```
typedef struct {
  char * pVendorName;
  char * pProductName;
  char * pProductVer;
  char * pSerialNo;
} USB_MSD_LUN_INFO;
```

Member	Description
pVendorName	Vendor name of the mass storage device. The string should be no longer than 8 bytes.
pProductName	Product name of the mass storage device. The product name string should be no longer than 16 bytes.
pProductVer	Product version number of the mass storage device. The product version string should be no longer than 4 bytes.
pSerialNo	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.

Table 6.17: USB_MSD_LUN_INFO elements

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()).

6.3.3.5 PREVENT_ALLOW_REMOVAL_HOOK

Description

Callback function to prevent/allow removal of storage medium. See *USBD_MSD_SetPreventAllowRemovalHook()* on page 153 for further information.

Prototype

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


6.3.3.6 PREVENT_ALLOW_REMOVAL_HOOK_EX

Description

Ex variant of PREVENT_ALLOW_REMOVAL_HOOK, this function definition additionally includes a parameter for the Lun index. See *USBD_MSD_SetPreventAllowRemovalHookEx()* on page 154 for further information.

Prototype

```
typedef void (PREVENT_ALLOW_REMOVAL_HOOK_EX)(U8 Lun, U8 PreventRemoval);
```


6.3.3.7 READ_WRITE_HOOK

Description

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

Prototype

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

Member	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 (1 for read, 0 for 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.

Table 6.18: READ_WRITE_HOOK elements

6.3.3.8 USB_MSD_INST_DATA_DRIVER

Description

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

Prototype

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

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

Table 6.19: USB_MSD_INST_DATA_DRIVER

Additional Information

This structure is passed to the storage driver. Therefore, the member of this structure can depend on the driver that is used.
For the storage driver that are shipped with this software the members of `USB_MSD_INST_DATA_DRIVER` have the following meaning:
`USB_MSD_StorageRAM`:

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

`USB_MSD_StorageByName`:

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

6.3.3.9 USB_MSD_STORAGE_API

Description

Structure that contains callbacks to the storage driver.

Prototype

```
typedef struct {
  void (*pfInit)             (U8              Lun,
                              const USB_MSD_INST_DATA_DRIVER * pDriverData);
  void (*pfGetInfo)          (U8              Lun,
                              USB_MSD_INFO * pInfo);
  U32  (*pfGetReadBuffer)    (U8              Lun,
                              U32             SectorIndex,
                              void         ** ppData,
                              U32             NumSectors);
  char (*pfRead)             (U8              Lun,
                              U32             SectorIndex,
                              void          * pData,
                              U32             NumSector);
  U32  (*pfGetWriteBuffer)   (U8              Lun,
                              U32             SectorIndex,
                              void         ** ppData,
                              U32             NumSectors);
  char (*pfWrite)            (U8              Lun,
                              U32             SectorIndex,
                              const void *    pData,
                              U32             NumSectors);
  char (*pfMediumIsPresent)  (U8              Lun);
  void (*pfDeInit)           (U8              Lun);
} USB_MSD_STORAGE_API;
```

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

Table 6.20: List of callback functions of USB_MSD_STORAGE_API

Additional Information

`USB_MSD_STORAGE_API` is used to retrieve information from the storage device driver or access data that needs to be read or written. Detailed information can be found in *Storage Driver* on page 173.

6.3.3.10 START_STOP_UNIT_HOOK

Description

Callback function which is called when a START STOP UNIT SCSI command is received.

Prototype

```
typedef void (START_STOP_UNIT_HOOK)(U8  Lun,
                                    U8  StartLoadEject);
```

Member	Description
Lun	Specifies the logical unit number which was accessed through read or write.
StartLoadEject	Binary OR of the SCSI LOEJ and START bits.

Table 6.21: START_STOP_UNIT_HOOK elements

Additional Information

The LOEJ (load eject) bit is located on bit position 1.
The START bit is located on bit position 0.
For further information please refer to the START STOP UNIT command description in the SCSI documentation.

6.4 Storage Driver

This section describes the storage interface in detail.

6.4.1 General information

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

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

6.4.1.1 Supported storage types

The supported storage types include:

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

6.4.1.2 Storage drivers supplied with this release

This release comes with the following drivers:

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

6.4.2 Interface function list

As described above, access to a storage medium is realized through an API-function table (`USB_MSD_STORAGE_API`). The storage functions are declared in `USB\MSD\USB_MSD.h`. The structure is described in section *Data structures* on page 163.

6.4.3 USB_MSD_STORAGE_API in detail

6.4.3.1 (*pfInit)()

Description

Initializes the storage medium.

Prototype

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

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
pDriverData	Pointer to a `USB_MSD_INST_DATA_DRIVER` structure that contains all information that is necessary for the driver initialization. For detailed information about the `USB_MSD_INST_DATA_DRIVER` structure, refer to *USB_MSD_INST_DATA_DRIVER* on page 170.

Table 6.22: (*pfInit)() parameter list

6.4.3.2 (*pfGetInfo)()

Description

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

Prototype

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

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
pInfo	Pointer to a USB_MSD_INFO structure. For detailed information about the USB_MSD_INFO structure, refer to *USB_MSD_INFO* on page 164.

Table 6.23: (*pfGetInfo)() parameter list

6.4.3.3 (*pfGetReadBuffer)()

Description

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

Prototype

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

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

Table 6.24: (*pfGetReadBuffer)() parameter list

Return value

Maximum number of consecutive sectors that can be read at once by the driver.

6.4.3.4 (*pfRead)()

Description

Reads one or multiple consecutive sectors from the storage medium.

Prototype

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

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

Table 6.25: (*pfRead)() parameter list

Return value

== 0: Success
!= 0: File System error code

6.4.3.5 (*pfGetWriteBuffer)()

Description

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

Prototype

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

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

Table 6.26: (*pfGetWriteBuffer)() parameter list

Return value

Maximum number of consecutive sectors that can be written into the buffer.

6.4.3.6 (*pfWrite)()

Description

Writes one or more consecutive sectors to the storage medium.

Prototype

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

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

Table 6.27: (*pfWrite)() parameter list

Return value

== 0: Success
!= 0: Error.

6.4.3.7 (*pfMediumIsPresent)()

Description

Checks if medium is present.

Prototype

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

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

Table 6.28: (*pfMediumIsPresent)() parameter list

Return value

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

6.4.3.8 (*pfDeInit)()

Description

Deinitializes the storage medium.

Prototype

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

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

Table 6.29: (*pfDeInit)() parameter list

Chapter 7

Smart Mass Storage Component (SmartMSD)

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

7.1 Overview

The SmartMSD 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 SmartMSD 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.

Smart MSD can access all data which has been created prior to the device being attached to the host, live data cannot be provided.

SmartMSD 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 formated 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, Mac OS X or any Linux distribution (including Android) which supports MSD, without installing any third party tools.

7.2 Configuration

7.2.1 Initial configuration

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

7.2.2 Final configuration

The configuration must only be modified if emUSB-Device is deployed in your final product. Refer to *Configuration* on page 40 for detailed information about the generic information functions which must be adapted.

7.2.3 Class specific configuration functions

For basic configuration please refer to the MSD chapter *Class specific configuration functions* on page 147.

In addition to the MSD configuration functions described in Chapter 6.2 "Configuration" the following SmartMSD functions are available.

Function	Description
emUSB-Device-SmartMSD configuration functions	
`USB_SmartMSD_X_Config()`	Configures the SmartMSD component.

Table 7.1: List of SmartMSD class specific configuration functions

7.2.3.1 USB_SmartMSD_X_Config()

Description

Main user configuration function of the SmartMSD component. This function is provided by the user.

Prototype

```
void USB_SmartMSD_X_Config(void);
```

Example

```
void USB_SmartMSD_X_Config(void) {
  //
  // Global configuration
  //
  USB_SmartMSD_AssignMemory(&_aMEMBuffer[0], sizeof(_aMEMBuffer));
  //
  // Setup LUN0
  //
  USB_SmartMSD_SetNumSectors(0, _SmartMSD_NUM_SECTORS);
  USB_SmartMSD_SetSectorsPerCluster(0, 32); // Anywhere from 1...128, needs to be 2^x
  USB_SmartMSD_SetNumRootDirSectors(0, 2);
  USB_SmartMSD_SetUserFunc(0, &_UserFuncAPI);
  USB_SmartMSD_SetVolumeID(0, "Virt0.MSD"); // Add volume ID
  //
  // Push const contents to the volume
  //
  USB_SmartMSD_AddConstFiles(0, &_aConstFiles[0], COUNTOF(_aConstFiles));
}
```

Additional information

During the call of USB_SmartMSD_Init() this user function is called in order to configure the SmartMSD 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 SmartMSD component. Otherwise USB_SMARTMSD_ON_PANIC is called.

7.2.4 Running the example application

The directory `Application` contains example applications that can be used with emUSB-Device and the SmartMSD component. To test the SmartMSD 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.

7.3 Target API

Function	Description
API functions	
USB_SmartMSD_Init()	Adds an MSD-class interface to the USB stack.
User supplied functions	
USB_SmartMSD_X_Config()	Configures SmartMSD. It sets all call-backs.
Configuration functions	
USB_SmartMSD_AssignMemory()	Assigns memory to the module.
USB_SmartMSD_SetUserFunc()	Sets various user-supplied functions.
USB_SmartMSD_SetNumRootDirSectors()	Sets the number of sectors reserved for the root directory.
USB_SmartMSD_SetVolumeID()	Sets volume ID (name) for Smart-MSD.
USB_SmartMSD_SetcbRead()	Sets the call-back for the read sector operation.
USB_SmartMSD_SetcbWrite()	Sets the call-back for the write sector operation.
USB_SmartMSD_AddConstFiles()	Adds constant files to SmartMSD.
USB_SmartMSD_SetNumSectors()	Sets the number of sectors available on device.
USB_SmartMSD_SetSectorsPerCluster()	Sets the number of sectors per cluster.
Data structures	
USB_SMARTMSD_CONST_FILE	Structure for displaying constant files.
USB_SMARTMSD_USER_FUNC_API	Structure for callback functions.
USB_SMARTMSD_FILE_INFO	File info structure.
USB_SMARTMSD_DIR_ENTRY	Structure for a directory entry.
USB_SMARTMSD_DIR_ENTRY_SHORT	Structure for a short directory entry.
USB_SMARTMSD_DIR_ENTRY_LONG	Structure for a long directory entry.
Function definitions	
USB_SMARTMSD_ON_READ_FUNC	Definition for the read callback.
USB_SMARTMSD_ON_WRITE_FUNC	Definition for the write callback.
USB_SMARTMSD_ON_PANIC	Definition for the panic callback.
USB_SMARTMSD_MEM_ALLOC	Definition for the memory alloc call-back.
USB_SMARTMSD_MEM_FREE	Definition for the memory free call-back.

Table 7.2: List of emUSB-Device SmartMSD interface functions and data structures

7.3.1 API functions

7.3.1.1 USB_SmartMSD_Init()

Description

Adds the SmartMSD component to the USB stack.

Prototype

```
void USB_SmartMSD_Init(void);
```

Additional information

After the initialization of emUSB-Device, this is the first function that needs to be called when the SmartMSD component is used with emUSB-Device. During the call of the said function the user function `USB_SmartMSD_X_Config()` is called in order to configure the storage itself.

7.3.1.2 USB_SmartMSD_X_Config()

Description

User supplied function that configures all storages of the SmartMSD component.

Prototype

```
void USB_SmartMSD_X_Config (void);
```

Additional information

This function is called automatically by USB_SmartMSD_Init() in order to allow to configure the storage volumes that SmartMSD should show after configuration. Only the following functions must be called in this context:

Allowed functions with USB_X_SmartMSD_Config
USB_SmartMSD_AssignMemory()
USB_SmartMSD_SetUserFunc()
USB_SmartMSD_SetNumRootDirSectors()
USB_SmartMSD_SetVolumeID()
USB_SmartMSD_SetcbRead()
USB_SmartMSD_SetcbWrite()
USB_SmartMSD_AddConstFiles()
USB_SmartMSD_SetNumSectors()
USB_SmartMSD_SetSectorsPerCluster()

Table 7.3: Allowed functions with USB_X_SmartMSD_Config

7.3.1.3 USB_SmartMSD_AssignMemory()

Description

Assigns memory to the module.

Prototype

```
void USB_SmartMSD_AssignMemory (U32 * p, U32 NumBytes);
```

Parameter	Description
p	Pointer to the memory which should be dedicated to SmartMSD.
NumBytes	Size of the memory block in bytes.

Table 7.4: USB_SmartMSD_AssignMemory() parameter list

7.3.1.4 USB_SmartMSD_SetUserFunc()

Description

Sets the default user callbacks for the SmartMSD component.

Prototype

```
void USB_SmartMSD_SetUserFunc(const USB_SMARTMSD_MSD_USER_FUNC_API *
pUserFunc);
```

Parameter	Description
pUserFunc	Pointer to a USB_SMARTMSD_USER_FUNC_API structure which holds the default function pointers for multiple functions.

Table 7.5: USB_SmartMSD_SetUserFunc() parameter list

Additional information

Check the description of USB_SMARTMSD_USER_FUNC_API for further details.

The default read and write callbacks can be overwritten by the per-LUN functions USB_SmartMSD_SetcbRead() and USB_SmartMSD_SetcbWrite().

7.3.1.5 USB_SmartMSD_SetNumRootDirSectors()

Description

Sets the number of sectors which should be used for root directory entries.

Prototype

```
void USB_SmartMSD_SetNumRootDirSectors(unsigned Lun, int NumRootDirSectors);
```

Parameter	Description
Lun	Specifies the logical unit number.
NumRootDirSectors	Number of sectors to be reserved for the root directory entries.

Table 7.6: USB_SmartMSD_SetNumRootDirSectors() parameter list

Additional information

The number of sectors reserved through this function is substracted from the number of sectors configured by USB_SmartMSD_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).

7.3.1.6 USB_SmartMSD_SetVolumeID()

Description

Sets the volume name for a specified LUN.

Prototype

```
int USB_SmartMSD_SetVolumeID(unsigned Lun, const char * sVolumeName);
```

Parameter	Description
Lun	Specifies the logical unit number.
sVolumeName	Pointer to the zero-terminated volume name.

Table 7.7: USB_SmartMSD_SetVolumeID() parameter list

Additional information

This function is optional, but can be helpful to identify the volume. If it is not used the host operating system chooses a default name for the volume.

7.3.1.7 USB_SmartMSD_SetcbRead()

Description

Sets a callback function for a specific LUN which gives information about the read sector-wise operations to the volume.

Prototype

```
void USB_SmartMSD_SetcbRead (unsigned Lun, USB_SMARTMSD_ON_READ_FUNC *
pfReadSector);
```

Parameter	Description
Lun	Specifies the logical unit number.
pfReadSector	Pointer to a user provided function of type USB_SMARTMSD_ON_READ_FUNC.

Table 7.8: USB_SmartMSD_SetcbRead() parameter list

Additional information

This callback is called each time a sector is read by the host. The callback should not block.
This callback supersedes the read callback set by USB_SmartMSD_SetUserFunc().

7.3.1.8 USB_SmartMSD_SetcbWrite()

Description

Sets a callback function for a specific LUN which gives information about the write sector-wise operations to the volume.

Prototype

```
void USB_SmartMSD_SetcbWrite (unsigned Lun, USB_SMARTMSD_ON_WRITE_FUNC *
pfWriteSector);
```

Parameter	Description
Lun	Specifies the logical unit number.
pfReadSector	Pointer to a user provided function of type USB_SMARTMSD_ON_WRITE_FUNC.

Table 7.9: USB_SmartMSD_SetcbWrite() parameter list

Additional information

This callback is called each time a sector is written by the host. The callback should not block.
This callback supersedes the write callback set by USB_SmartMSD_SetUserFunc().

7.3.1.9 USB_SmartMSD_AddConstFiles()

Description

Allows to add multiple files which should be shown on a SmartMSD volume as soon as it is connected. A common example would be a "Readme.txt" or a link to the company website.

Prototype

```
int USB_SmartMSD_AddConstFiles (unsigned Lun, USB_SMARTMSD_CONST_FILE * paConstFile, int NumFiles);
```

Parameter	Description
Lun	Specifies the logical unit number.
paConstFile	Pointer to an array of USB_SMARTMSD_CONST_FILE structures.
NumFiles	The number of items in the paConstFile array.

Table 7.10: USB_SmartMSD_AddConstFiles() parameter list

Additional information

For additional information please see USB_SMARTMSD_CONST_FILE.

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, 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                   FirstClust
  { "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 VMSD volume.
*/
void USB_VMSD_X_Config(void) {
  <...>
  USB_VMSD_AddConstFiles(1, &_aConstFiles[0], COUNTOF(_aConstFiles));
  <...>
}
```

7.3.1.10 USB_SmartMSD_SetNumSectors()

Description

Sets the number of sectors available on the volume.

Prototype

```
void USB_SmartMSD_SetNumSectors (unsigned Lun, int NumSectors);
```

Parameter	Description
Lun	Specifies the logical unit number.
NumSectors	Specifies the number of sectors for a LUN.

Table 7.11: USB_SmartMSD_SetNumSectors() parameter list

7.3.1.11 USB_SmartMSD_SetSectorsPerCluster()

Description

Sets the number of sectors per cluster.

Prototype

```
void USB_SmartMSD_SetSectorsPerCluster (unsigned Lun,
 int SectorsPerCluster);
```

Parameter	Description
Lun	Specifies the logical unit number.
SectorsPerCluster	Specifies the number of sectors for a LUN.

Table 7.12: USB_SmartMSD_SetSectorsPerCluster() parameter list

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.

7.3.2 Data structures

7.3.2.1 USB_SMARTMSD_CONST_FILE

Description

This structure contains information about a constant file which cannot be changed at run time and should be shown inside the SmartMSD volume (e.g. Readme.txt).
This structure is a parameter for the `USB_SmartMSD_AddConstFiles()` function.

Prototype

```
typedef struct {
  const char* sName;
  const U8* pData;
  int FileSize;
  U32 FirstClust;
} USB_SMARTMSD_CONST_FILE;
```

Member	Description
`sName`	Pointer to a zero-terminated string containing the filename.
`pData`	Pointer to the file data. Can be NULL.
`FileSize`	Size of the file. Normally the size of the data pointed to by `pData`.
`FirstClust`	Allows to reserve a cluster (block) for a file. This is done automatically when the value is zero.

Table 7.13: USB_SMARTMSD_CONST_FILE elements

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).

7.3.2.2 USB_SMARTMSD_USER_FUNC_API

Description

This structure contains the function pointers for user provided functions.
This structure is a parameter for the USB_SmartMSD_SetUserFunc() function.

Prototype

```
typedef struct _USB_SMARTMSD_USER_FUNC_API {
  USB_SMARTMSD_ON_READ_FUNC  * pfOnReadSector;            // Mandatory
  USB_SMARTMSD_ON_WRITE_FUNC * pfOnWriteSector;           // Mandatory
  USB_SMARTMSD_ON_PANIC      * pfOnPanic;                 // Optional
  USB_SMARTMSD_MEM_ALLOC     * pfMemAlloc;                // Optional
  USB_SMARTMSD_MEM_FREE      * pfMemFree;                 // Optional
} USB_SMARTMSD_USER_FUNC_API;
```

Member	Description
pfOnReadSector	Pointer to a callback function of type USB_SMARTMSD_ON_READ_FUNC which is called when a sector is read from the host. This function is mandatory and can not be NULL.
pfOnWriteSector	Pointer to a callback function of type USB_SMARTMSD_ON_WRITE_FUNC which is called when a sector is written from the host. This function is mandatory and can not be NULL.
pfOnPanic	Pointer to a user provided panic function of type USB_SMARTMSD_ON_PANIC. If this pointer is NULL the internal panic function is called.
pfMemAlloc	Pointer to a user provided alloc function of type USB_SMARTMSD_MEM_ALLOC. If this pointer is NULL the internal alloc function is called. If no memory block is assigned pfOnPanic is called.
pfMemFree	Pointer to a user provided free function of type USB_SMARTMSD_MEM_FREE. If this pointer is NULL the internal free function is called. If no memory block is assigned pfOnPanic is called.

Table 7.14: USB_SMARTMSD_USER_FUNC_API elements

Additional Information

The default callback functions for read and write are overwritten by the per-LUN read and write functions, which are set through USB_SmartMSD_SetcbRead() and USB_SmartMSD_SetcbWrite().

7.3.2.3 USB_SMARTMSD_FILE_INFO

Description

Structure used in the read and write callbacks.

Prototype

```
typedef struct {
  const USB_SMARTMSD_DIR_ENTRY* pDirEntry;
} USB_SMARTMSD_FILE_INFO;
```

Member	Description
pDirEntry	Pointer to a USB_SMARTMSD_DIR_ENTRY structure.

Table 7.15: USB_SMARTMSD_FILE_INFO elements

Additional Information

Check USB_SMARTMSD_ON_READ_FUNC, USB_SMARTMSD_ON_WRITE_FUNC and USB_SMARTMSD_DIR_ENTRY for more information.

7.3.2.4 USB_SMARTMSD_DIR_ENTRY

Description

Union containing references to directory entries. This union is a member of USB_SMARTMSD_FILE_INFO.

Prototype

```
typedef union {
  USB_SMARTMSD_DIR_ENTRY_SHORT ShortEntry;
  USB_SMARTMSD_DIR_ENTRY_LONG  LongEntry;
  U8                           ac[32];
} USB_SMARTMSD_DIR_ENTRY;
```

Member	Description
ShortEntry	Allows to access the entry as a "short directory entry".
LongEntry	Allows to access the entry as a "long directory entry".
ac	Allows to write directly to the structure without casting or using the members.

Table 7.16: USB_SMARTMSD_DIR_ENTRY elements

Additional Information

Check USB_SMARTMSD_DIR_ENTRY_SHORT and USB_SMARTMSD_DIR_ENTRY_LONG for more information.

7.3.2.5 USB_SMARTMSD_DIR_ENTRY_SHORT

Description

Structure used to describe an entry with a short file name. This structure is a member of USB_SMARTMSD_DIR_ENTRY.

Prototype

```
typedef struct {
  U8  acFilename[8];
  U8  acExt[3];
  U8  DirAttr;
  U8  NTRes;
  U8  CrtTimeTenth;
  U16 CrtTime;
  U16 CrtDate;
  U16 LstAccDate;
  U16 FstClusHI;
  U16 WrtTime;
  U16 WrtDate;
  U16 FstClusLO;
  U32 FileSize;
} USB_SMARTMSD_DIR_ENTRY_SHORT;
```

Member	Description
acFilename	File name, limited to 8 characters (short file name), padded with spaces (0x20).
acExt	File extension, limited to 3 characters (short file name), padded with spaces (0x20).
DirAttr	File attributes. Available attributes are listed below.
NTRes	Reserved for use by Windows NT.
CrtTimeTenth	Millisecond stamp at file creation time. This field actually contains a count of tenths of a second.
CrtTime	Creation time.
CrtDate	Date file was created.
LstAccDate	Last access date. Note that there is no last access time, only a date. This is the date of last read or write.
FstClusHI	High word of this entry's first cluster number.
WrtTime	Time of last write.
WrtDate	Date of last write.
FstClusLO	Low word of this entry's first cluster number.
FileSize	File size in bytes.

Table 7.17: USB_SMARTMSD_DIR_ENTRY_SHORT elements

Additional Information

The following file attributes are available for short dir entries:

Attribute	Explanation
USB_SMARTMSD_ATTR_READ_ONLY	The file is read-only.
USB_SMARTMSD_ATTR_HIDDEN	The file is hidden.
USB_SMARTMSD_ATTR_SYSTEM	The file is designated as a system file.
USB_SMARTMSD_ATTR_VOLUME_ID	This entry is the volume ID (volume name).
USB_SMARTMSD_ATTR_DIRECTORY	The file is a directory.
USB_SMARTMSD_ATTR_ARCHIVE	The file has the archive attribute.
USB_SMARTMSD_ATTR_LONG_NAME	The file has a long file name, see USB_SMARTMSD_DIR_ENTRY_LONG.

7.3.2.6 USB_SMARTMSD_DIR_ENTRY_LONG

Description

Structure used to describe an entry with a long file name. This structure is a member of USB_SMARTMSD_DIR_ENTRY.
This is for information only, the read and write callbacks only receive short file names.

Prototype

```
typedef struct {
  U8   Ord;
  U8   acName1[10];
  U8   Attr;
  U8   Type;
  U8   Chksum;
  U8   acName2[12];
  U16  FstClusLO;
  U8   acName3[4];
} USB_SMARTMSD_DIR_ENTRY_LONG;
```

Member	Description
Ord	The order of this entry in the sequence of long dir entries, associated with the short dir entry at the end of the long dir set.
acName1	Characters 1-5 of the long-name sub-component in this dir entry.
Attr	Attributes - must be USB_SMARTMSD_ATTR_LONG_NAME.
Type	If zero, indicates a directory entry that is a sub-component of a long name. Other values reserved for future extensions. Non-zero implies other types.
Chksum	Checksum of name in the short dir entry at the end of the long dir set.
acName2	Characters 6-11 of the long-name sub-component in this dir entry.
FstClusLO	Must be zero.
acName3	Characters 12-13 of the long-name sub-component in this dir entry.

Table 7.18: USB_SMARTMSD_DIR_ENTRY_LONG elements

7.3.2.7 USB_SMARTMSD_ON_READ_FUNC

Description

Callback function prototype that is used when calling the USB_SmartMSD_SetUserFunc() and USB_SmartMSD_SetcbRead() functions.

Prototype

```
typedef int USB_SMARTMSD_ON_READ_FUNC (U8 * pData, U32 Off, U32 NumBytes,
const USB_SMARTMSD_FILE_INFO * pFile);
```

Parameter	Description
pData	Pointer to a buffer in which the data is stored.
Off	Offset in the file which is read by the host.
NumBytes	Amount of bytes requested by the host.
pFile	Pointer to a USB_SMARTMSD_FILE_INFO strucure describing the file.

Table 7.19: USB_SMARTMSD_ON_READ_FUNC parameter list

Return value

== 0: Success.
!= 0: An error occurred.

7.3.2.8 USB_SMARTMSD_ON_WRITE_FUNC

Description

Callback function prototype that is used when calling the USB_SmartMSD_SetUserFunc() and USB_SmartMSD_SetcbWrite() functions.

Prototype

```
typedef int USB_SMARTMSD_ON_WRITE_FUNC (const U8 * pData, U32 Off, U32
NumBytes, const USB_SMARTMSD_FILE_INFO * pFile);
```

Parameter	Description
pData	Pointer to the data to be written (received from the host).
Off	Offset in the file which the host writes.
NumBytes	Amount of bytes to write.
pFile	Pointer to a USB_SMARTMSD_FILE_INFO strucure describing the file.

Table 7.20: USB_SMARTMSD_ON_WRITE_FUNC parameter list

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.

7.3.2.9 USB_SMARTMSD_ON_PANIC

Description

Callback function prototype that is called when a fatal, unrecoverable error occurs.

Prototype

```
typedef void USB_SMARTMSD_ON_PANIC (const char * sErr);
```

Parameter	Description
sErr	Pointer to a zero-terminated string describing the error.

Table 7.21: USB_SMARTMSD_ON_PANIC parameter list

7.3.2.10 USB_SMARTMSD_MEM_ALLOC

Description

Function prototype that is used when memory is being allocated by the SmartMSD module.

Prototype

```
typedef void * USB_SMARTMSD_MEM_ALLOC (U32 Size);
```

Parameter	Description
Size	Size of the required memory in bytes.

Table 7.22: USB_SMARTMSD_MEM_ALLOC parameter list

Return value

Pointer to the allocated memory or NULL.

7.3.2.11 USB_SMARTMSD_MEM_FREE

Description

Function prototype that is used when memory is being freed by the SmartMSD module.

Prototype

```
typedef void USB_SMARTMSD_MEM_FREE (void * p);
```

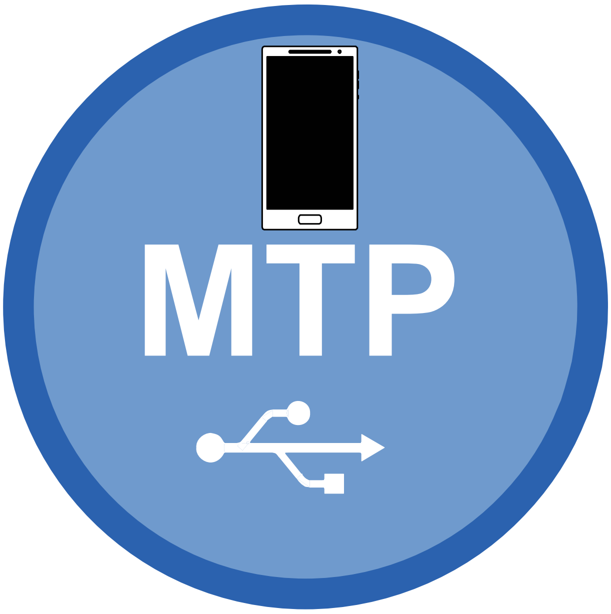
Parameter	Description
p	Pointer to a memory block which was previously allocated by USB_SMARTMSD_MEM_ALLOC.

Table 7.23: USB_SMARTMSD_MEM_FREE parameter list

Chapter 8

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.



8.1 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 audio and video files.

MTP is an alternative to Mass Storage Device (MSD) and it operates at the file level, in contrast to MSD which reads and writes sector data. This type of operation gives MTP some advantages over MSD:

- The cable can be safely removed during the 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

The current implementation of emUSB-Device-MTP has the following limitations:

- The device does not notify the host when the data on the storage medium changes (file added/removed, file size change, etc.)

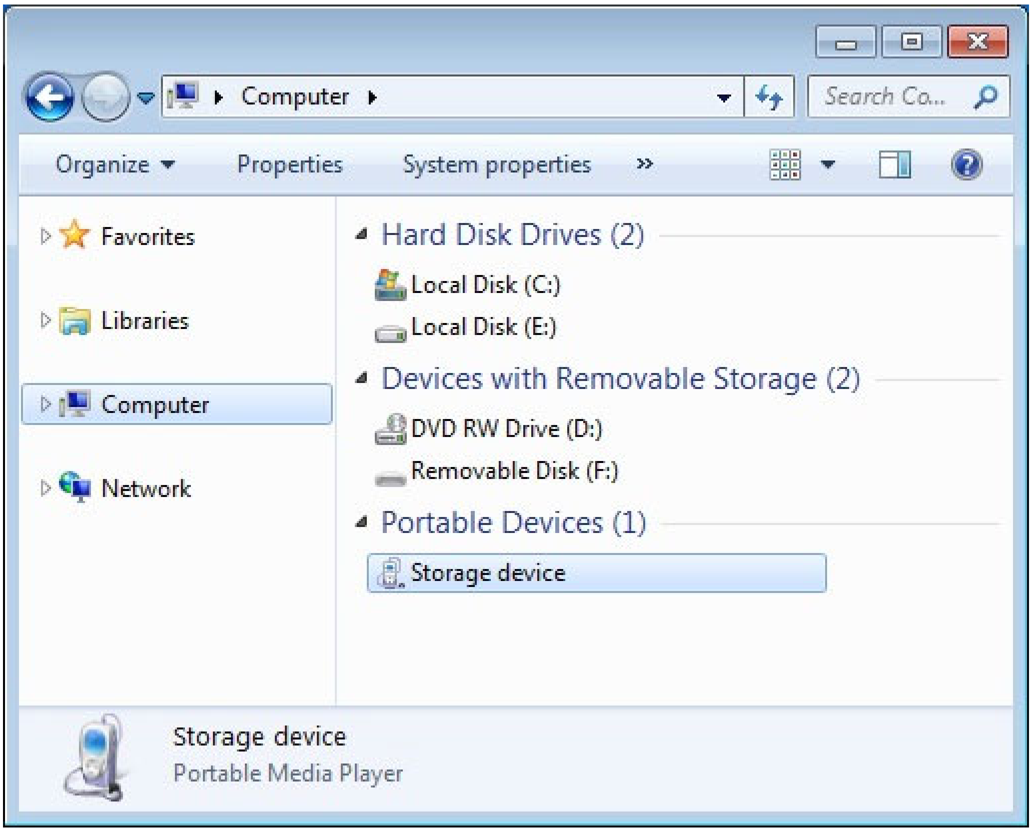
Get in contact with us if you need this feature to be supported.

emUSB-Device-MTP comes as a complete package and contains the following:

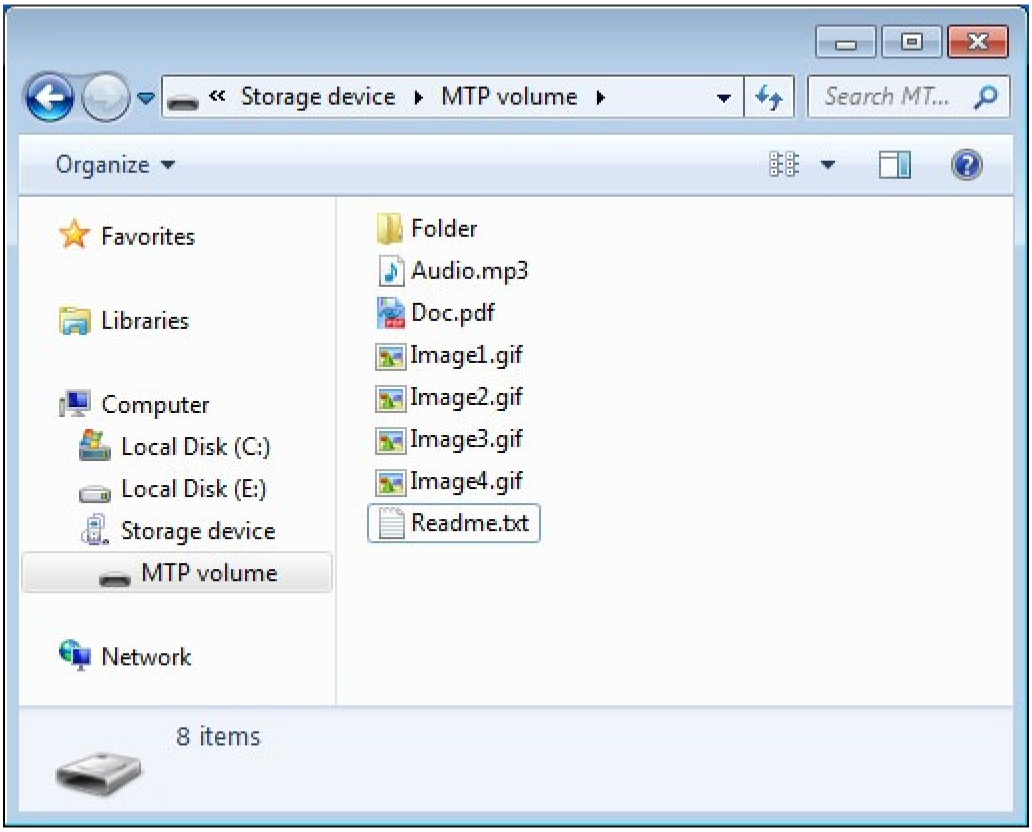
- Generic USB handling
- MTP device class implementation
- Storage driver which uses emFile
- Sample application showing how to work with MTP

8.1.1 Getting access to files

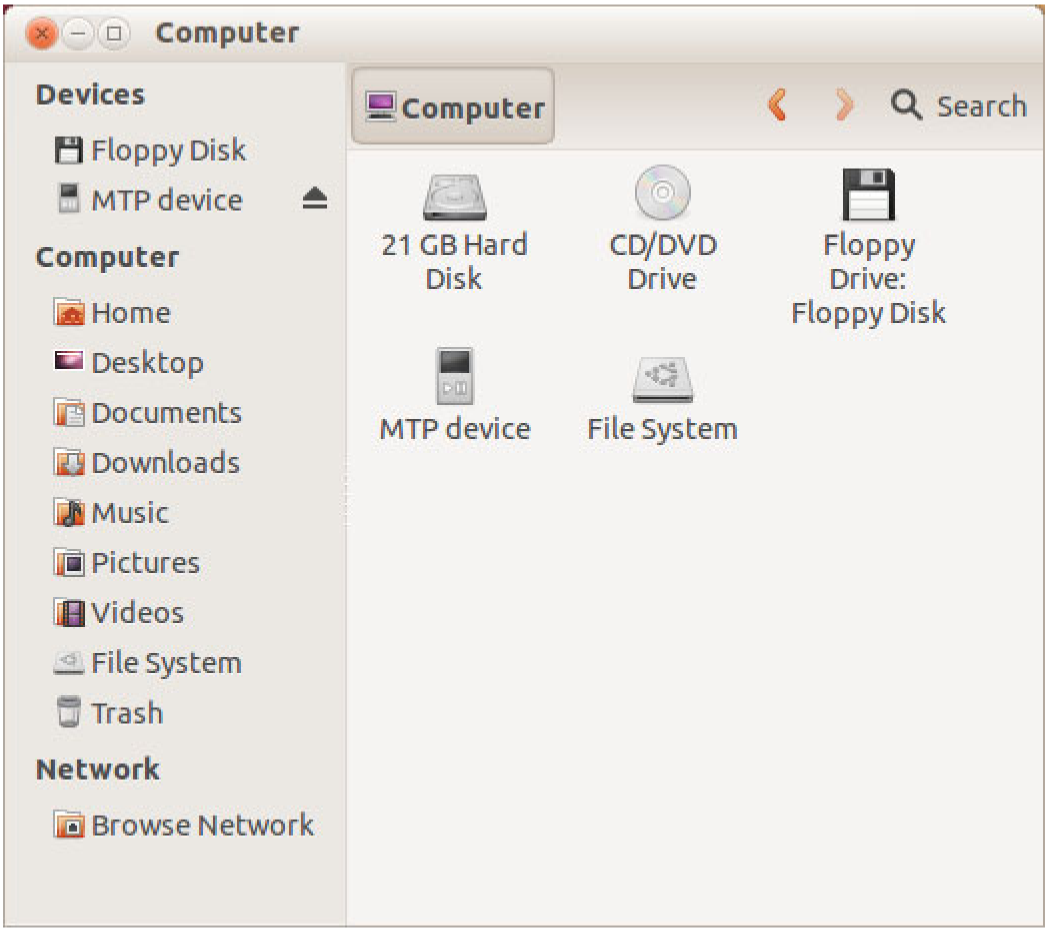
An MTP device will be displayed under the "Portable Devices" section in the "Computer" window when connected to a PC running the Microsoft Windows 7 operating system:



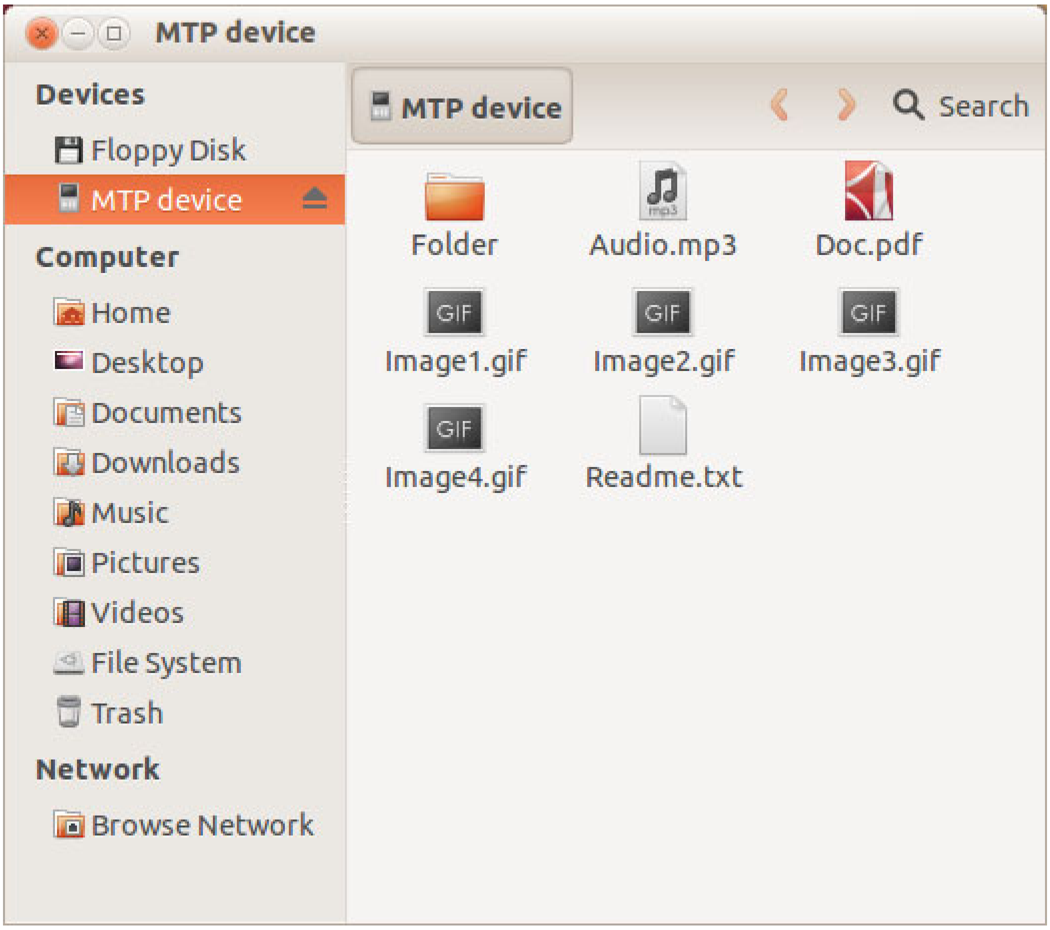
The file and directories stored on the device are accessed in the usual way using the Windows Explorer:



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.

8.1.2 Additional information

For more technical details about MTP and PTP follow these links:

MTP specification

PTP specification

8.2 Configuration

8.2.1 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.

8.2.2 Final configuration

The configuration must only be modified when emUSB-Device is integrated in your final product. Refer to section *Configuration* on page 40 for detailed information about the generic information functions which have to be adapted.

8.2.3 Class specific configuration

Beside the generic emUSB-Device configuration functions (*Configuration* on page 40), 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);
```


8.2.4 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_DEBUG_LEVEL	0	Sets the type of diagnostic messages output at runtime. It can take one of these values: 0 - no debug messages 1 - only error messages 2 - error and log messages
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.

Table 8.1: MTP configuration macros

8.3 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.

8.3.1 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_MAX_PACKET_SIZE, NULL, 0);
  InitData.EPOut               = USBD_AddEP(0, USB_TRANSFER_TYPE_BULK,
                                            USB_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 sample uses the `USB_MAX_PACKET_SIZE` define which is set to the correct value. 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.

8.4 Target API

Function	Description
API functions	
USBD_MTP_Add()	Adds an MTP interface to the USB stack.
USBD_MTP_AddStorage()	Adds a storage device to the emUSB-Device-MTP.
USBD_MTP_Task()	Handles the MTP communication.
USBD_MTP_SendEvent()	Sends an event notification to the MTP host.
Data structures	
USB_MTP_FILE_INFO	Stores information about a file or directory.
USB_MTP_INIT_DATA	Stores the MTP initialization parameters.
USB_MTP_INST_DATA	Stores the initialization parameters of storage driver.
USB_MTP_INST_DATA_DRIVER	Stores parameters that are passed to storage driver.
USB_MTP_STORAGE_API	Stores callbacks to the functions of storage driver.
USB_MTP_STORAGE_INFO	Stores information about the storage medium.
Enums	
USB_MTP_EVENT	Available events to be used with USBD_MTP_SendEvent().

Table 8.2: List of emUSB-Device MTP interface functions and data structures

8.4.1 API functions

8.4.1.1 USBD_MTP_Add()

Description

Adds an MTP-class interface to the USB stack.

Prototype

```
void USBD_MTP_Add(const USB_MTP_INIT_DATA * pInitData);
```

Parameter	Description
pInitData	Pointer to a USB_MTP_INIT_DATA structure.

Table 8.3: USBD_MTP_Add() parameter list

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* on page 226 for more information.

8.4.1.2 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);
```

Parameter	Description
pInstData	Pointer to a USB_MTP_INST_DATA structure which contains the parameters of the added storage driver.

Table 8.4: USBD_MTP_AddStorage() parameter list

Return value

== 0: Invalid handle, storage could not be added
!= 0: Valid handle, storage has been successfully added.

Additional information

It is necessary to call this function immediately after USBD_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* on page 228 for more information.

8.4.1.3 USBD_MTP_Task()

Description

Task which handles the MTP communication.

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.

8.4.1.4 USBD_MTP_SendEvent()

Description

Sends an event notification to the MTP host.

Prototype

```
void USBD_MTP_SendEvent(USB_MTP_STORAGE_HANDLE hStorage,
                        USB_MTP_EVENT           Event,
                        void                  * pPara);
```

Parameter	Description
hStorage	Handle to a storage that was returned by USBD_MTP_AddStorage().
Event	Event that occurred. The following events are currently supported: USB_MTP_EVENT_OBJECTADDED USB_MTP_EVENT_OBJECTREMOVED USB_MTP_EVENT_STOREADDED USB_MTP_EVENT_STOREREMOVED USB_MTP_EVENT_OBJECTINFOCHANGED USB_MTP_EVENT_STOREFULL USB_MTP_EVENT_STORAGEINFOCHANGED
pPara	Pointer to some additional information. This parameter depends on the event. In case of Event = USB_MTP_EVENT_OBJECTADDED USB_MTP_EVENT_OBJECTREMOVED USB_MTP_EVENT_OBJECTINFOCHANGED pPara is a pointer to filled USB_MTP_FILE_INFO structure. USB_MTP_EVENT_STOREADDED USB_MTP_EVENT_STOREREMOVED USB_MTP_EVENT_STORAGEINFOCHANGED pPara is not used and can be NULL.

Table 8.5: USBD_MTP_SendEvent() parameter list

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

```
/*********************************************************************
*
*       _GetFileInfo
*/
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;
}

/*********************************************************************
*
*       _WriteLogFile
*/
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);
  USBD_MTP_SendEvent(_ahStorage[0], (USB_MTP_EVENT)r, &FileInfo);
  USBD_MTP_SendEvent(_ahStorage[0], USB_MTP_EVENT_STORAGEINFOCHANGED, NULL);
  return r;
}
```


8.4.2 Data structures

8.4.2.1 USB_MTP_FILE_INFO

Description

Structure which stores information about a file or directory.

Prototype

```
typedef struct {
  char * pFilePath;
  char * pFileName;
  U32    FileSize;
  U32    CreationTime;
  U32    LastWriteTime;
  U8     IsDirectory;
  U8     Attributes;
  U8     acId[MTP_NUM_BYTES_FILE_ID];
} USB_MTP_FILE_INFO;
```

Member	Description
pFilePath	Pointer to full path to file.
pFileName	Pointer to beginning of file/directory name in pFilePath
FileSize	Size of the file in bytes.
CreationTime	Time and date when the file was created.
LastWriteTime	Time and data when the file was last modified.
IsDirectory	Set to 1 if the path points to a directory.
Attributes	Bitmask containing the file or directory attributes.
acId	Unique file/directory identifier.

Table 8.6: USB_MTP_FILE_INFO elements

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

acId should be unique for each file and directory on the file system and it should be persistent between MTP sessions.

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 user.

8.4.2.2 USB_MTP_INIT_DATA

Description

Structure which stores the parameters of the MTP interface.

Prototype

```
typedef struct {
  U8      EPIn;
  U8      EPOut;
  U8      EPInt;
  void * pObjectList;
  U32     NumBytesObjectList;
  void * pDataBuffer;
  U32     NumBytesDataBuffer;
  //
  // The following fields are used internally by the MTP component.
  //
  U8      InterfaceNum;
  U32     NumBytesAllocated;
  U32     NumObjects;
  USB_MTP_INFO * pMTPInfo;
} USB_MTP_INIT_DATA;
```

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.
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 USB_MTP_INFO structure. Filling this structure is mandatory.

Table 8.7: USB_MTP_INIT_DATA elements

Additional Information

This structure holds the endpoints that should be used with the MTP interface. Refer to *USBD_AddDriver()* on page 52 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.

8.4.2.3 USB_MTP_INFO

Description

Structure that is used when initialising the MTP module.

Prototype

```
typedef struct USB_MTP_INFO {
  char * pManufacturer;
  char * pModel;
  char * pDeviceVersion;
  char * pSerialNumber; // Must be exactly 32 hex characters long.
} USB_MTP_INFO;
```

Member	Description
pManufacturer	Name of the device manufacturer.
pModel	Model name of the MTP device.
pDeviceVersion	Version of the MTP device.
pSerialNumber	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.

Table 8.8: USB_MTP_INFO elements

8.4.2.4 USB_MTP_INST_DATA

Description

Structure which stores the parameters of storage driver.

Prototype

```
typedef struct {
  const USB_MTP_STORAGE_API * pAPI;
  const char                * sDescription;
  const char                * sVolumeId;
  USB_MTP_INST_DATA_DRIVER    DriverData;
} USB_MTP_INST_DATA;
```

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 Windows Explorer.
sVolumeId	Unique volume identifier.
DriverData	Driver data that are passed to the storage driver. Refer to *USB_MTP_INST_DATA_DRIVER* on page 229 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.

Table 8.9: USB_MTP_INST_DATA elements

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.

8.4.2.5 USB_MTP_INST_DATA_DRIVER

Description

Structure which stores the parameters passed to the storage driver.

Prototype

```
typedef struct {
  const char * pRootDir;
} USB_MTP_INST_DATA_DRIVER;
```

Member	Description
pRootDir	Path to directory to be used as the root of the storage.

Table 8.10: USB_MTP_INST_DATA_DRIVER

Additional Information

`pRootDir` can specify the root of the file system or any other subdirectory.

8.4.2.6 USB_MTP_STORAGE_API

Description

Structure that contains callbacks to the storage driver.

Prototype

```
typedef struct {
  void (*pfInit)                 (U8                          Unit,
                                  const USB_MTP_INST_DATA_DRIVER * pDriverData);
  void (*pfGetInfo)              (U8                          Unit,
                                  USB_MTP_STORAGE_INFO      * pStorageInfo);
  int  (*pfFindFirstFile)        (U8                          Unit,
                                  const char                * pDirPath,
                                  USB_MTP_FILE_INFO         * pFileInfo);
  int  (*pfFindNextFile)         (U8                          Unit,
                                  USB_MTP_FILE_INFO         * pFileInfo);
  int  (*pfOpenFile)             (U8                          Unit,
                                  const char                * pFilePath);
  int  (*pfCreateFile)           (U8                          Unit,
                                  const char                * pDirPath,
                                  USB_MTP_FILE_INFO         * pFileInfo);
  int  (*pfReadFromFile)         (U8                          Unit,
                                  U32                         Off,
                                  void                      * pData,
                                  U32                         NumBytes);
  int  (*pfWriteToFile)          (U8                          Unit,
                                  U32                         Off,
                                  const void                * pData,
                                  U32                         NumBytes);
  int  (*pfCloseFile)            (U8                          Unit);
  int  (*pfRemoveFile)           (U8                          Unit,
                                  const char                * pFilePath);
  int  (*pfCreateDir)            (U8                          Unit,
                                  const char                * pDirPath,
                                  USB_MTP_FILE_INFO         * pFileInfo);
  int  (*pfRemoveDir)            (U8                          Unit,
                                  const char                * pDirPath);
  int  (*pfFormat)               (U8                          Unit);
  int  (*pfRenameFile)           (U8                          Unit,
                                  USB_MTP_FILE_INFO         * pFileInfo);
  void (*pfDeInit)               (U8                          Unit);
  int  (*pfGetFileAttributes)    (U8                          Unit
                                  const char                * pFilePath,
                                  U8                        * pMask);
  int  (*pfModifyFileAttributes)(U8                           Unit,
                                  const char                * pFilePath,
                                  U8                          SetMask,
                                  U8                          ClrMask);
  int  (*pfGetFileCreationTime)  (U8                          Unit,
                                  const char                * pFilePath,
                                  U32                       * pTime);
  int  (*pfGetFileLastWriteTime)(U8                           Unit,
                                  const char                * pFilePath,
                                  U32                       * pTime);
  int  (*pfGetFileId)            (U8                          Unit,
                                  const char                * pFilePath,
                                  U8                        * pId);
  int  (*pfGetFileSize)          (U8                          Unit,
                                  const char                * pFilePath,
                                  U32                       * pFileSize);
} USB_MTP_STORAGE_API;
```

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.

Table 8.11: List of callback functions of USB_MTP_STORAGE_API

Member	Description
(*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)()	Deinitializes 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.
(*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.

Table 8.11: List of callback functions of USB_MTP_STORAGE_API

Additional Information

`USB_MTP_STORAGE_API` is used to retrieve information from the storage driver or to access data that needs to be read or written. Detailed information can be found in *Storage Driver* on page 236.

8.4.2.7 USB_MTP_STORAGE_INFO

Description

Structure which stores information about the storage medium.

Prototype

```
typedef struct {
  U32 NumKbytesTotal;
  U32 NumKbytesFreeSpace;
  U16 FSType;
  U8  IsWriteProtected;
  U8  IsRemovable;
} USB_MTP_STORAGE_INFO;
```

Member	Description
NumKbytesTotal	Capacity of storage medium in Kbytes.
NumKbytesFreeSpace	Available free space on storage medium in Kbytes.
FSType	Type of file system as specified in MTP.
IsWriteProtected	Set to 1 if the storage medium canDone.not be modified.
IsRemovable	Set to 1 if the storage medium can be removed from device.

Table 8.12: USB_MTP_STORAGE_INFO

8.5 Enums

This chapter describes the used enums defined in the header file USB_MTP.h.

Type	Description
USB_MTP_EVENT	Enum containing the MTP device event codes.

Table 8.13: emUSB-Device MTP type definition overview

8.5.0.1 USB_MTP_EVENT

Description

Enum containing the MTP even codes.

Prototype

```
typedef enum _USB_MTP_EVENT {
  USB_MTP_EVENT_UNDEFINED = 0x4000,
  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 = 0xC801,
  USB_MTP_EVENT_OBJECTPROPDESCCHANGED,
  USB_MTP_EVENT_OBJECTREFERENCESCHANGED
} USB_MTP_EVENT;
```

Enum value	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.
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.

Table 8.14: USB_MTP_EVENT enum.

Enum value	Description
USB_MTP_EVENT_REQUESTOBJECTTRANSFER	This event can be used by the device to ask the host to initiate an 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 staorage 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 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 an 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-aquired 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.

Table 8.14: USB_MTP_EVENT enum.

8.6 Storage Driver

This section describes the storage interface in detail.

8.6.1 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.

This release comes with `USB_MTP_StorageFS` driver which uses emFile to access the storage medium.

8.6.2 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 *Data structures* on page 225.

8.6.3 USB_MTP_STORAGE_API in detail

8.6.3.1 (*pfInit)()

Description

Initializes the storage medium.

Prototype

```
void (*pfInit)(U8 Unit, const USB_MTP_INST_DATA_DRIVER * pDriverData);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pDriverData	Pointer to a USB_MTP_INST_DATA_DRIVER structure that contains all information that are necessary for the driver initialization. For detailed information about the USB_MTP_INST_DATA_DRIVER structure, refer to *USB_MTP_INST_DATA_DRIVER* on page 229.

Table 8.15: (*pfInit)() parameter list

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.

8.6.3.2 (*pfGetInfo)()

Description

Returns information about storage medium such as capacity and available free space.

Prototype

```
void (*pfGetInfo)(U8 Unit, USB_MTP_STORAGE_INFO * pStorageInfo);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pStorageInfo	Pointer to a `USB_MTP_STORAGE_INFO` structure. For detailed information about the `USB_MTP_STORAGE_INFO` structure, refer to *USB_MTP_STORAGE_INFO* on page 232.

Table 8.16: (*pfGetInfo)() parameter list

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.

8.6.3.3 (*pfFindFirstFile)()

Description

Returns information about the first file in a specified directory.

Prototype

```
int (*pfFindFirstFile)(U8                   Unit,
                       const char         * pDirPath,
                       USB_MTP_FILE_INFO * pFileInfo);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pDirPath	Full path to the directory to be searched.
pFileInfo	IN: --- OUT: Information about the file/directory found.

Table 8.17: (*pfFindFirstFile)() parameter list

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.

8.6.3.4 (*pfFindNextFile)()

Description

Moves to next file and returns information about it.

Prototype

```
int (*pfFindNextFile)(U8 Unit, USB_MTP_FILE_INFO * pFileInfo);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFileInfo	IN: --- OUT: Information about the file/directory found.

Table 8.18: (*pfFindNextFile)() parameter list

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.

8.6.3.5 (*pfOpenFile)()

Description

Opens a file for reading.

Prototype

```
int (*pfOpenFile)(U8           Unit,
                  const char * pFilePath);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	IN: Full path to file. OUT ---.

Table 8.19: (*pfOpenFile)() parameter list

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 (*pfReadFromFile)(). At the end of data transfer the MTP module closes the file by calling (*pfCloseFile)(). If the file does not exists an error should be returned. The MTP module opens only one file at a time.

8.6.3.6 (*pfCreateFile)()

Description

Opens a file for writing.

Prototype

```
int (*pfCreateFile)(U8                    Unit,
                    const char          * pDirPath,
                    USB_MTP_FILE_INFO * pFileInfo);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pDirPath	IN: Full path to directory where the file should be created. OUT: ---
pFileInfo	IN: Information about the file to be created. pFileName points to the name of the file. OUT: pFilePath points to full path of created file, pFileName points to the beginning of file name in pFilePath.

Table 8.20: (*pfCreateFile)() parameter list

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 filed of pFileInfo. If the file exists it should be truncated to zero length. When a file is created, the call to (*pfCreateFile)() is followed by one or more calls to (*pfWriteToFile)(). If CreationTime and LastWriteTime in pFileInfo are not zero, these should be used instead of the time stamps generated by the file system.

8.6.3.7 (*pfReadFromFile)()

Description

Reads data from the current file.

Prototype

```
int (*pfReadFromFile)(U8     Unit,
                      U32    Off,
                      void * pData,
                      U32    NumBytes);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
Off	Byte offset where to read from.
pData	IN: --- OUT: Data read from file.
NumBytes	Number of bytes to read from file.

Table 8.21: (*pfReadFromFile)() parameter list

Return value

== 0: Data read from file
!= 0: An error occurred

Additional information

The function reads data from the file opened by (*pfOpenFile)().

8.6.3.8 (*pfWriteToFile)()

Description

Writes data to current file.

Prototype

```
int (*pfWriteToFile)(U8           Unit,
                     U32          Off,
                     const void * pData,
                     U32          NumBytes);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
Off	Byte offset where to write to.
pData	IN: Data to write to file OUT: ---
NumBytes	Number of bytes to write to file.

Table 8.22: (*pfWriteToFile)() parameter list

Return value

== 0: Data written to file
!= 0: An error occurred

Additional information

The function writes data to file opened by (*pfCreateFile)().

8.6.3.9 (*pfCloseFile)()

Description

Closes the current file.

Prototype

```
int (*pfCloseFile)(U8 Unit);
```

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

Table 8.23: (*pfCloseFile)() parameter list

Return value

== 0: File closed
!= 0: An error occurred

Additional information

The function closes the file opened by `(*pfCreateFile)()` or `(*pfOpenFile)()`.

8.6.3.10 (*pfRemoveFile)()

Description

Removes a file/directory from the storage medium.

Prototype

```
int (*pfRemoveFile)(U8           Unit,
                    const char * pFilePath);
```

Parameter	Description
Unit	Logical unit number. Specifies for which drive the function is called.
pFilePath	IN: Full path to file/directory to be removed OUT: ---

Table 8.24: (*pfRemoveFile)() parameter list

Return value

== 0: File removed
!= 0: An error occurred

8.6.3.11 (*pfCreateDir)()

Description

Creates a directory on the storage medium.

Prototype

```
int (*pfCreateDir)(U8                    Unit,
                   const char          * pDirPath,
                   USB_MTP_FILE_INFO   * pFileInfo);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pDirPath	IN: Full path to directory where the directory should be created. OUT: ---
pFileInfo	IN: Information about the directory to be created. pFileName points to the directory name. OUT: pFilePath points to full path of directory, pFileName points to the beginning of directory name in pFilePath

Table 8.25: (*pfCreateDir)() parameter list

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.

8.6.3.12 (*pfRemoveDir)()

Description

Removes a directory and its contents from the storage medium.

Prototype

```
int (*pfRemoveDir)(U8           Unit,
                   const char * pDirPath);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pDirPath	IN: Full path to directory to be removed. OUT: ---

Table 8.26: (*pfRemoveDir)() parameter list

Return value

== 0: Directory removed
!= 0: An error occurred

Additional information

The function should remove the directory and the entire file tree under it.

8.6.3.13 (*pfFormat)()

Description

Initializes the storage medium.

Prototype

```
int (*pfFormat)(U8 Unit);
```

Parameter	Description
`Unit`	Logical unit number. Specifies for which storage medium the function is called.

Table 8.27: (*pfFormat)() parameter list

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 `(*pfInit)()`, 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.

8.6.3.14 (*pfRenameFile)()

Description

Changes the name of a file or directory.

Prototype

```
int (*pfRenameFile)(U8 Unit, USB_MTP_FILE_INFO * pFileInfo);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFileInfo	IN: Information about the file/directory to be renamed. `pFilePath` points to the full path and `pFileName` points to the new name. OUT: `pFilePath` points to full path of file/directory with the new name, `pFileName` points to the beginning of file/directory name in `pFilePath`. The other structure fields should also be filled.

Table 8.28: (*pfRenameFile)() parameter list

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.

8.6.3.15 (*pfDeInit)()

Description

Deinitializes the storage medium.

Prototype

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

Parameter	Description
`Unit`	Logical unit number. Specifies for which storage medium the function is called.

Table 8.29: (*pfDeInit)() parameter list

Additional information

Typically called when the application calls `USB_Stop()` to deinitialize emUSB-Device.

8.6.3.16 (*pfGetFileAttributes)()

Description

Returns the attributes of a file or directory.

Prototype

```
int (*pfGetFileAttributes)(U8 Unit, const char * pFilePath, U8 * pMask);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Full path to file or directory (0-terminated string).
pMask	IN: --- OUT: The bitmask of the attributes.

Table 8.30: (*pfGetFileAttributes)() parameter list

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* on page 225.

8.6.3.17 (*pfModifyFileAttributes)()

Description

Sets and clears file attributes.

Prototype

```
int (*pfModifyFileAttributes)(U8           Unit,
                              const char * pFilePath,
                              U8           SetMask,
                              U8           ClrMask);;
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Full path to file or directory (0-terminated string).
SetMask	The bitmask of the attributes which should be set.
ClrMask	The bitmask of the attributes which should be cleared.

Table 8.31: (*pfModifyFileAttributes)() parameter list

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* on page 225.

8.6.3.18 (*pfGetFileCreationTime)()

Description

Returns the creation time of file or directory.

Prototype

```
int (*pfGetFileCreationTime)(U8 Unit, const char * pFilePath, U32 * pTime);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Full path to file or directory (0-terminated string).
pTime	IN: --- OUT: The creation time.

Table 8.32: (*pfGetFileCreationTime)() parameter list

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 encoding of the time value refer to *USB_MTP_FILE_INFO* on page 225.

8.6.3.19 (*pfGetFileLastWriteTime)()

Description

Returns the time when the file or directory was last modified.

Prototype

```
int (*pfGetFileLastWriteTime)(U8         Unit,
                              const char * pFilePath,
                              U32        * pTime);;
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Full path to file or directory (0-terminated string).
pTime	IN: --- OUT: The modification time.

Table 8.33: (*pfGetFileLastWriteTime)() parameter list

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 encoding of the time value refer to *USB_MTP_FILE_INFO* on page 225.

8.6.3.20 (*pfGetFileId)()

Description

Returns an ID which uniquely identifies the file or directory.

Prototype

```
int (*pfGetFileId)(U8 Unit, const char * pFilePath, U8 * pId);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Full path to file or directory (0-terminated string).
pId	IN: --- OUT: The unique ID of file or directory. Should point to a byte array MTP_NUM_BYTES_FILE_ID large.

Table 8.34: (*pfGetFileId)() parameter list

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.

8.6.3.21 (*pfGetFileSize)()

Description

Returns the size of a file in bytes.

Prototype

```
int (*pfGetFileSize)(U8 Unit, const char * pFilePath, U32 * pFileSize);
```

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Full path to file or directory (0-terminated string).
pFileSize	IN: --- OUT: The size of file in bytes.

Table 8.35: (*pfGetFileSize)() parameter list

Return value

== 0: Size of file returned
!= 0: An error occurred

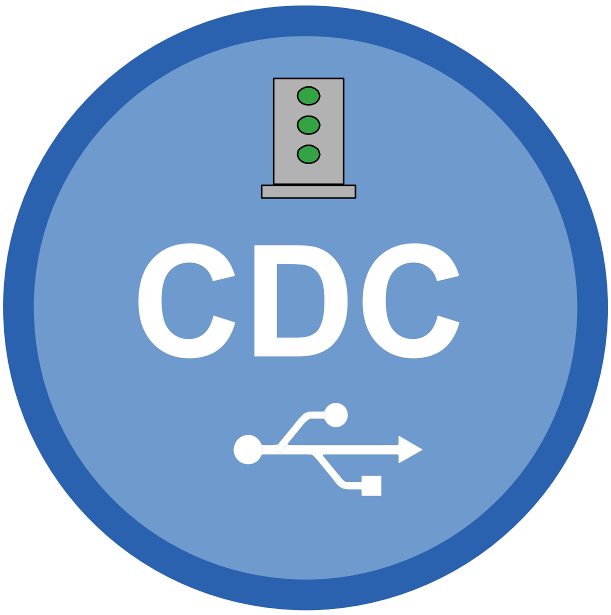
Additional information

This function is called only when the compile time switch `MTP_SAVE_FILE_INFO` is set to 0.

Chapter 9

Communication Device Class (CDC)

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



9.1 Overview

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

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

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

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

9.1.1 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.

9.2 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_Start.c`:

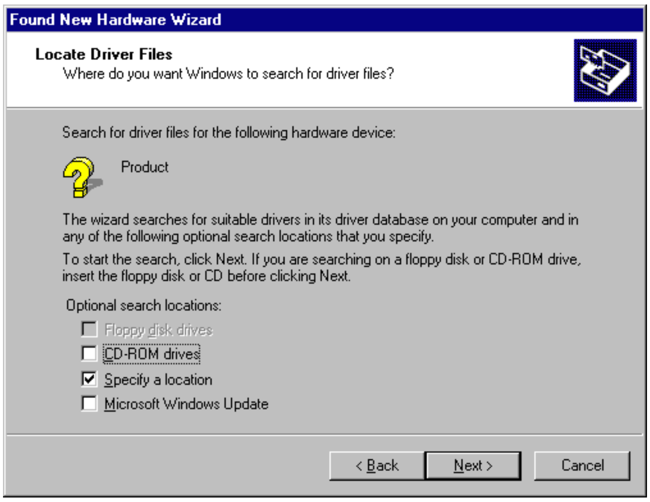
```
/*********************************************************************
*
*       MainTask
*
* USB handling task.
*   Modify to implement the desired protocol
*/
void MainTask(void);
void MainTask(void) {
  U32 i = 0;
  USB_CDC_HANDLE hInst;

  USBD_Init();
  hInst = _AddCDC();
  USBD_Start();
  while (1) {
    char ac[64];
    char acOut[30];
    int  NumBytesReceived;
    int  NumBytesToSend;

    //
    // 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);
    i++;
    NumBytesToSend = sprintf(acOut, "%.3lu: Received %d byte(s) - \"", i,
NumBytesReceived);
    if (NumBytesReceived > 0) {
      USBD_CDC_Write(hInst, &acOut[0], NumBytesToSend, 0);
      USBD_CDC_Write(hInst, &ac[0], NumBytesReceived, 0);
      USBD_CDC_Write(hInst, "\"\n\r", 3, 0);
    }
  }
}
```


9.3 Installing the driver

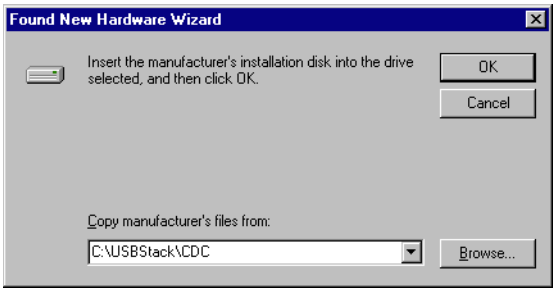
When the emUSB-Device-CDC sample application is up and running and the target device is plugged into the computer's USB port, Windows will detect the new hardware.

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

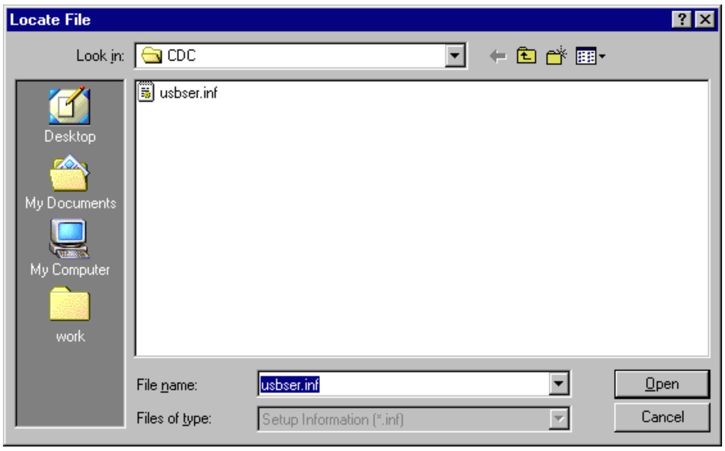
In the next step, you need to select the **Specify a location** option and click the **Next** button.



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



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



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

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

9.3.1 The .inf file

The `.inf` file is required for installation.

It is as follows:

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

[Manufacturer]
%MFGNAME%=USB2SerialDeviceList

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

[DestinationDirs]
USB2SerialCopyFiles=12
DefaultDestDir=12

[USB2SerialInstall]
CopyFiles=USB2SerialCopyFiles
AddReg=USB2SerialAddReg

[USB2SerialCopyFiles]
usbser.sys,,,0x20

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

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

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

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

red - required modifications
green - possible modifications

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

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

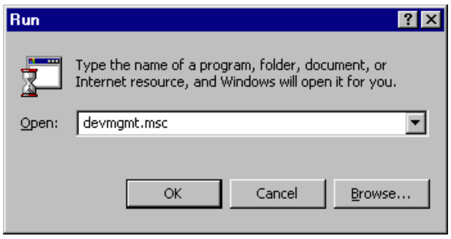
The required modifications of the file `USB_Conf.h` are described in the configuration chapter.

9.3.2 Installation verification

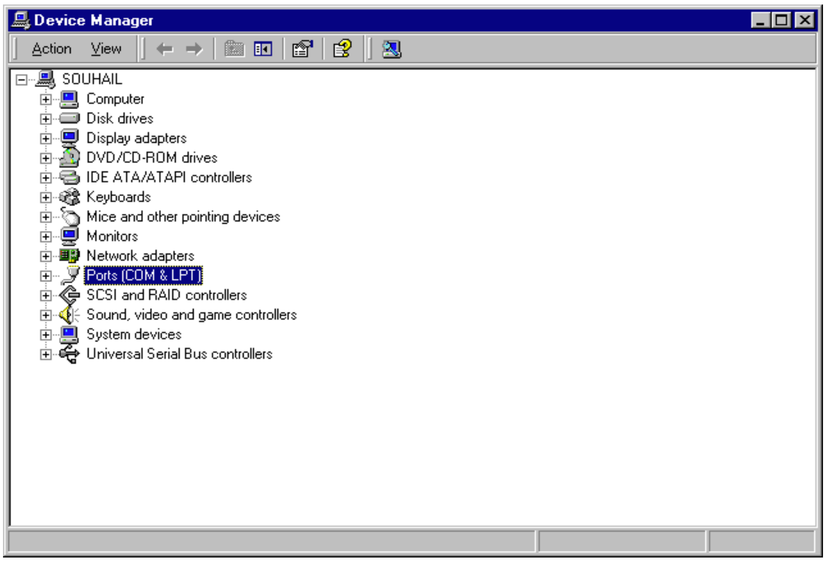
After the device has been installed, it can verify that the installation of the USB device was successful. Hence, take a look in the device manager to check that the USB device is displayed.

The following steps perform:

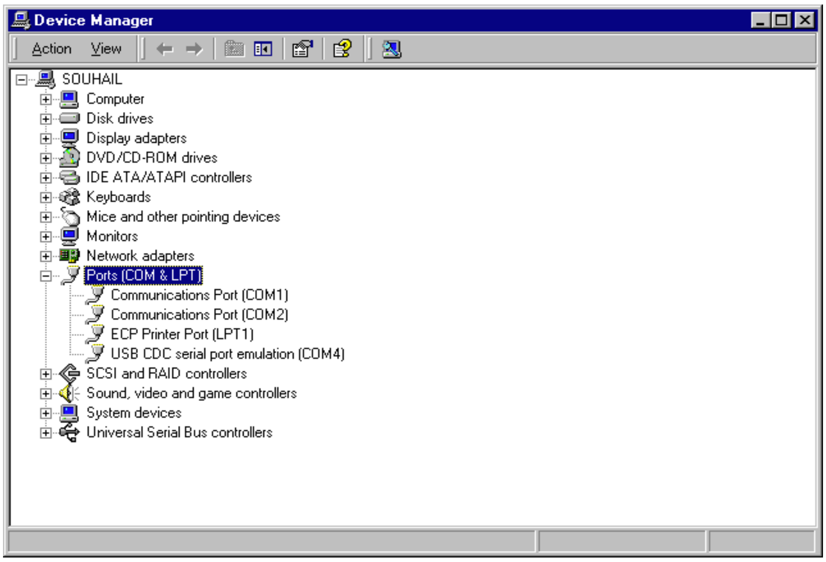
- Open the **Run** dialog box from the start menu.
  Type `devicemgmt.msc` and click **OK**:



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



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



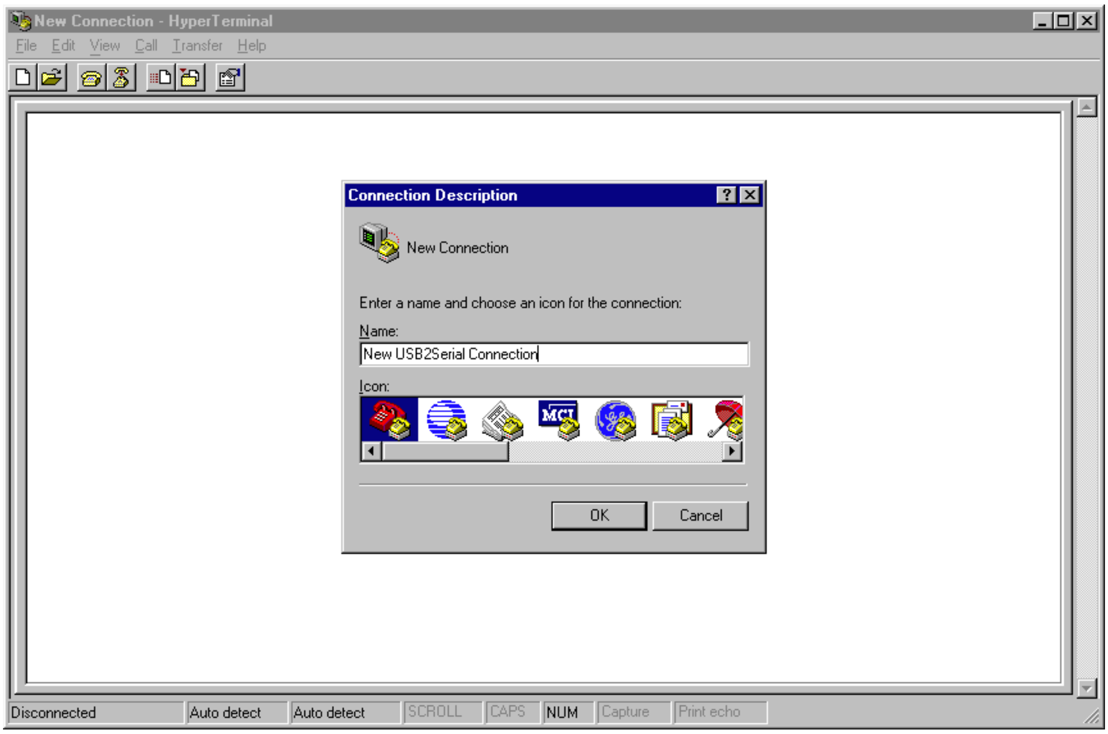
  You should see the **USB CDC serial port emulation (COM**`x`**)**, where `x` gives the COM port number has Windows has assigned to the device.

9.3.3 Testing communication to the USB device

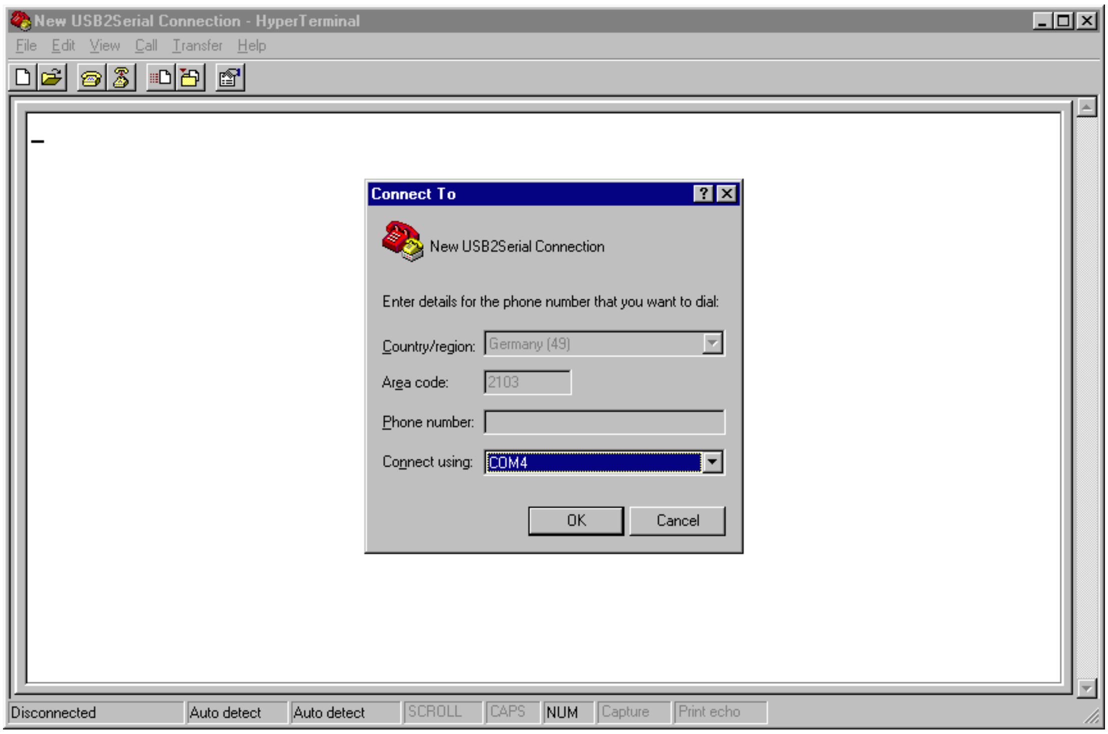
The start application is a simple echo server. This means each character that is entered and sent through the virtual serial port will be sent back by the USB device and will be shown by a terminal program. To test the communication to the device, a terminal program such as HyperTerminal, should be used.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

This section is relevant for Windows XP and below, for newer Windows versions please use a terminal program of your choice.

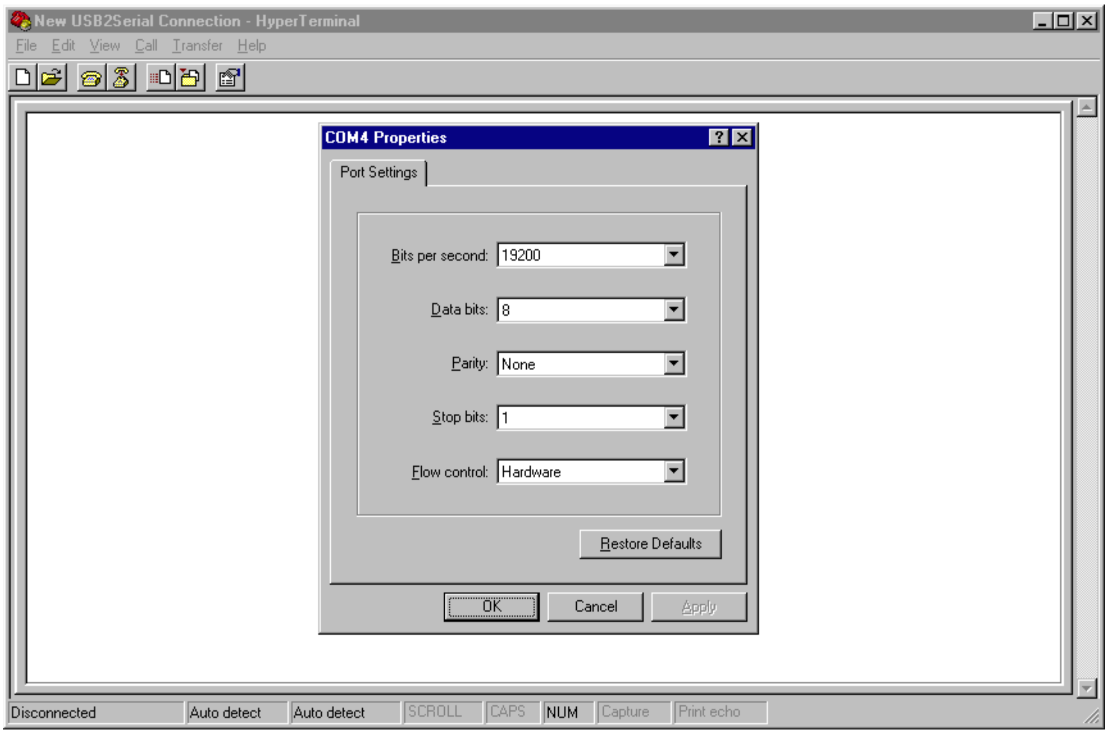
- Open the **Run** dialog box from the start menu.
  Type `hypertrm.exe` and press Enter key to open the HyperTerminal.
  HyperTerminal displays the Connection Description dialog.
  Give this new connection a name as shown below and click **OK**.



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

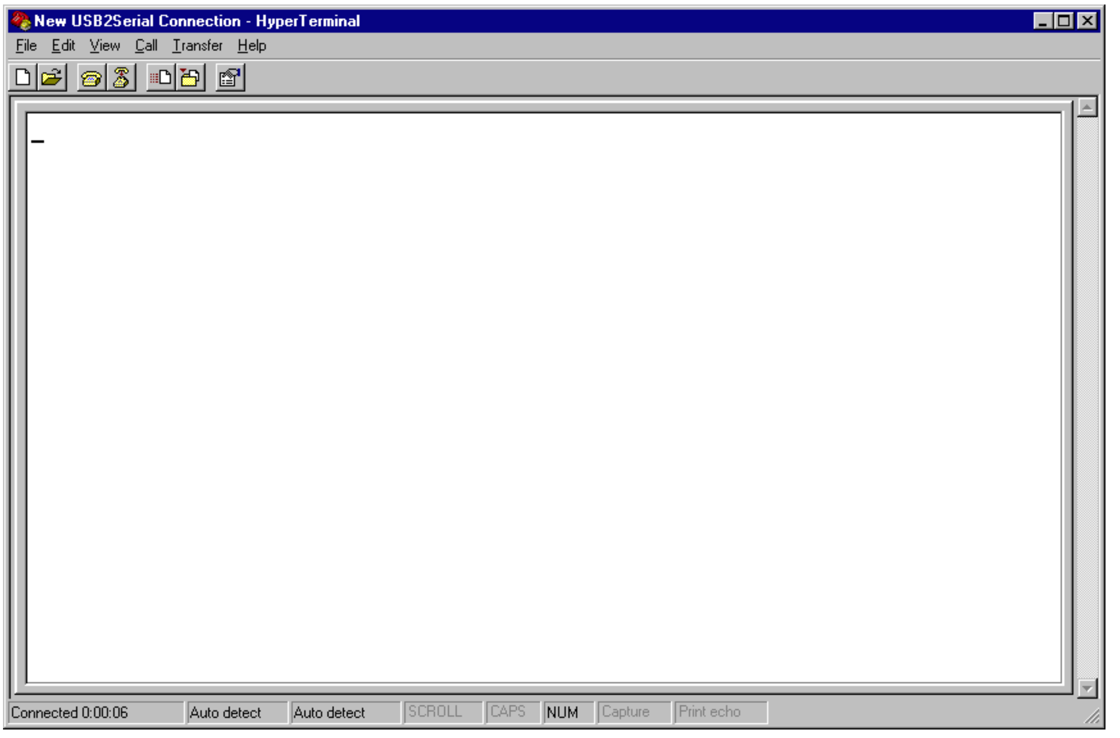


- The **COMx Property** dialog box is displayed to setup the connection properties. Setup the values as shown below:

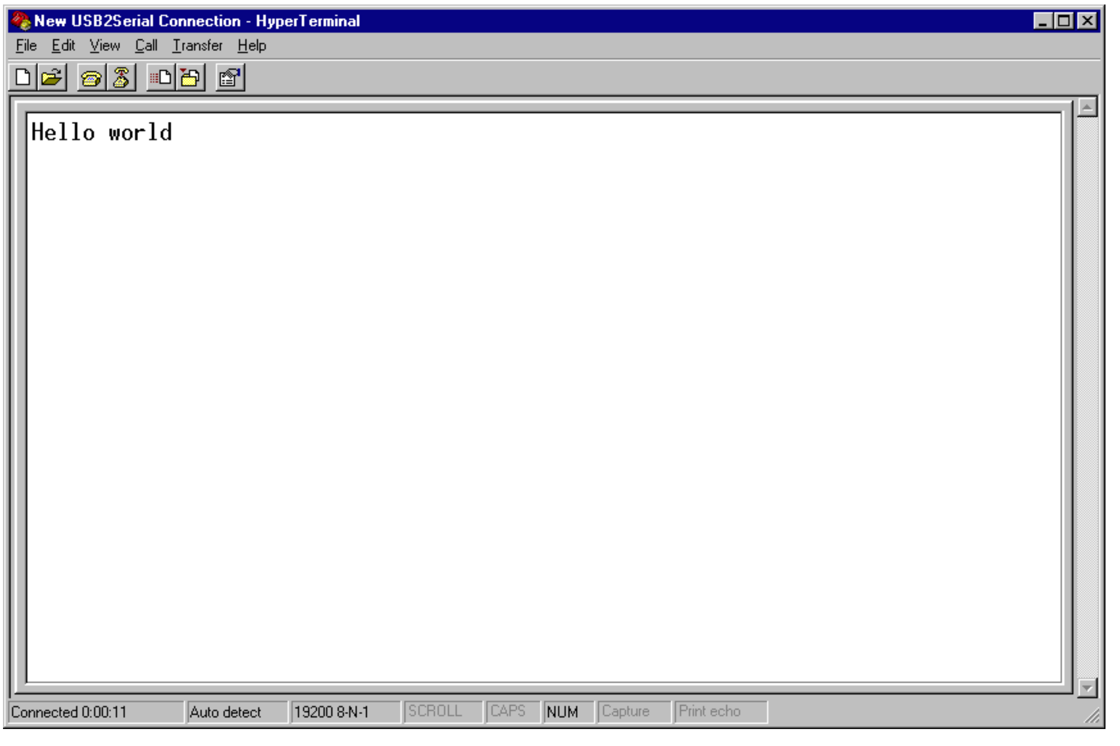


- To confirm your selection, click **OK**.

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



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



9.4 Target API

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

9.4.1 Interface function list

Name	Description
API functions	
USBD_CDC_Add()	Adds CDC-class to the emUSB-Device interface.
USBD_CDC_CancelRead()	Cancels an asynchronous read operation that is pending
USBD_CDC_CancelWrite()	Cancels an asynchronous read operation that is pending
USBD_CDC_Read()	Reads data from host.
USBD_CDC_ReadOverlapped()	Reads data from host asynchronously.
USBD_CDC_Receive()	Reads data from host.
USBD_CDC_SetOnBreak()	Sets a callback for receiving a SEND_BREAK by the host.
USBD_CDC_SetOnLineCoding()	Sets a callback for registering changing of the "line-coding" by the host.
USBD_CDC_SetOnControlLineState()	Sets a callback for registering changing of the "control-line-state" by the host.
USBD_CDC_SetOnRXEvent()	Adds a callback function for RX events.
USBD_CDC_SetOnTXEvent()	Adds a callback function for TX events.
USBD_CDC_UpdateSerialState()	Changes the current serial state.
USBD_CDC_Write()	Writes data to host.
USBD_CDC_WaitForRX()	Waits for reading data transfer from the Host to be ended.
USBD_CDC_WaitForTX()	Waits for writing data transfer to the Host to be ended.
USBD_CDC_WaitForTXReady()	Wait until stack is ready to accept new write operation.
USBD_CDC_WriteSerialState()	Sends the current serial state to the Host.
USBD_CDC_GetNumBytesRemToRead()	Returns the number of byte which still need to be read.
USBD_CDC_GetNumBytesRemToWrite()	Returns the number of byte which still need to be written.
USBD_CDC_GetNumBytesInBuffer()	Retrives the amount of bytes in the internal read buffer.
Data structures	
USB_CDC_INIT_DATA	Initialization structure that is needed when adding an CDC interface.
USB_CDC_ON_SET_BREAK	Callback function to receive a break condition sent by the host.
USB_CDC_ON_SET_LINE_CODING	Callback registering line-coding changes.
USB_CDC_ON_SET_CONTROL_LINE_STATE	Callback registering changes on control liste states.
USB_CDC_LINE_CODING	Structure that contains the new line-coding sent by the host.
USB_CDC_CONTROL_LINE_STATE	Structure that contains the new controll line state sent by the host.

Table 9.1: USB-CDC API overview

9.4.2 API functions

9.4.2.1 USBD_CDC_Add()

Description

Adds CDC class to the USB interface.

Prototype

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

Parameter	Description
pInitData	Pointer to a USB_CDC_INIT_DATA structure. For detailed information about the USB_CDC_INIT_DATA structure, refer to *USB_CDC_INIT_DATA* on page 293.

Table 9.2: USBD_CDC_Add() parameter list

Return value

== 0xFFFFFFFF: New CDC Instance can not be created.
!= 0xFFFFFFFF: Handle to a valid CDC instance.

Additional information

After the initialization of general emUSB-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 more than one CDC-Instance please make sure the USB_EnableIAD() is called before, otherwise none but the first CDC instance will work correctly.

9.4.2.2 USBD_CDC_CancelRead()

Description

Cancels a non-blocking write operation that is pending.

Prototype

```
void USBD_CDC_CancelRead(USB_CDC_HANDLE hInst);
```

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

Table 9.3: USBD_CDC_CancelRead() parameter list

Additional information

This function shall be called when a pending asynchronous read operation (triggered by USBD_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 differnet task or interrupt function).

9.4.2.3 USBD_CDC_CancelWrite()

Description

Cancels a non-blocking write operation that is pending.

Prototype

```
void USBD_CDC_CancelWrite(USB_CDC_HANDLE hInst);;
```

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

Table 9.4: USBD_CDC_CancelWrite() parameter list

Additional information

This function shall be called when a pending asynchronously write operation (triggered by non-blocking call to USBD_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 differnet task or interrupt function).

9.4.2.4 USBD_CDC_Read()

Description

Reads data from the host.

Prototype

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

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
ms	Timeout given in milliseconds. A zero value results in an infinite timeout.

Table 9.5: USBD_CDC_Read() parameter list

Return value

== NumBytes:	Requested data was succesfully read within the given timeout.
>= 0, < NumBytes:	Timeout has occured. Number of bytes that have been read within the given timeout.
< 0:	Returns a USB_STATUS_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.

9.4.2.5 USBD_CDC_ReadOverlapped()

Description

Reads data from the host asynchronously.

Prototype

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

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

Table 9.6: USBD_CDC_ReadOverlapped() parameter list

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, USBD_CDC_WaitForRX() needs to be called.
Another synchronisation method would be to periodically call USBD_CDC_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_CDC_CancelRead().
The buffer pointed to by pData must be valid until the read operation is terminated.

Example

See USBD_CDC_GetNumBytesRemToRead().

9.4.2.6 USBD_CDC_Receive()

Description

Reads data from host. The function blocks until any data has been received. In contrast to USBD_CDC_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 has been reached.

Prototype

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

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pBuffer	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
ms	Timeout given in milliseconds. A zero value results in an infinite timeout.

Table 9.7: USBD_CDC_Receive() parameter list

Return value

> 0: Number of bytes that have been read within the given timeout.
== 0: Timeout occured, zero packet received (not every controller supports this!) or the target was disconnected during the function call.
< 0: Returns a USB_STATUS_ERROR.

Additional information

If no error occurs, this function returns the number of bytes received.
Calling USBD_CDC_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 target is disconnected from host or when a USB reset occurred during the function call, it will then return the number of bytes read. If the target was disconnected before this function was called, it returns USB_STATUS_ERROR.

9.4.2.7 USBD_CDC_SetOnBreak()

Description

Sets a callback for receiving a SEND_BREAK by the host.

Prototype

```
void USBD_CDC_SetOnBreak(USB_CDC_HANDLE         hInst,
                         USB_CDC_ON_SET_BREAK * pfOnBreak);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pf	Pointer to the callback function `USB_CDC_ON_SET_BREAK`. For detailed information about the `USB_CDC_ON_SET_BREAK` function pointer, refer to *USB_CDC_ON_SET_BREAK* on page 294.

Table 9.8: USBD_CDC_SeOnBreak() parameter list

Additional information

This function is used to register a user callback which should notify the application about a break condition sent by the host. Refer to *USB_CDC_ON_SET_BREAK* on page 294 for detailed information. The callback is called in an ISR context, therefore it should should execute quickly.

9.4.2.8 USBD_CDC_SetOnLineCoding()

Description

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

Prototype

```
void USBD_CDC_SetOnLineCoding(USB_CDC_HANDLE hInst,
                              USB_CDC_ON_SET_LINE_CODING * pf);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pf	Pointer to the callback function USB_CDC_ON_SET_LINE_CODING. For detailed information about the USB_CDC_ON_SET_LINE_CODING function pointer, refer to *USB_CDC_ON_SET_LINE_CODING* on page 295.

Table 9.9: USBD_CDC_SetLineCoding() parameter list

Additional information

This function is used to register a user callback which notifies the application that the host has changed the line coding refer to *USB_CDC_ON_SET_LINE_CODING* on page 295 for detailed information. The callback is called in an ISR context, therefore it should execute quickly.

9.4.2.9 USBD_CDC_SetOnControlLineState()

Description

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

Prototype

```
void USBD_CDC_SetOnControlLineState(USB_CDC_HANDLE hInst,
                                    USB_CDC_ON_SET_CONTROL_LINE_STATE * pf);
```

Parameter	Description
`hInst`	Handle to a valid CDC instance, returned by USBD_CDC_Add().
`pf`	Pointer to the callback function `USB_CDC_ON_SET_CONTROL_LINE_STATE`. For detailed information about the `USB_CDC_ON_SET_CONTROL_LINE_STATE` function pointer, refer to *USB_CDC_ON_SET_CONTROL_LINE_STATE* on page 296.

Table 9.10: USBD_CDC_SetOnControlLineState() parameter list

Additional information

This function is used to register a user callback which notifies the application that the host has changed the line coding refer to *USB_CDC_ON_SET_CONTROL_LINE_STATE* on page 296 for detailed information. The callback is called in an ISR context, therefore it should execute quickly.

9.4.2.10 USBD_CDC_SetOnRXEvent()

Description

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

Prototype

```
void USBD_CDC_SetOnRXEvent(USB_CDC_HANDLE hInst,
                           USB_EVENT_CALLBACK *pEventCb,
                           USB_EVENT_CALLBACK_FUNC *pfEventCb,
                             void *pContext);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pEventCb	Pointer to a USB_EVENT_CALLBACK structure.
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

Table 9.11: USBD_CDC_SetOnRXEvent() parameter list

Additional information

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 has the follwing 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 USB_SetOnEvent function

Table 9.12: Event callback function parameter list

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
USB_EVENT_DATA_READ	Some data was received from the host on the endpoint.
USB_EVENT_READ_COMPLETE	The last read operation was completed.
USB_EVENT_READ_ABORT	A read transfer was aborted.

Table 9.13: USB events

9.4.2.11 USBD_CDC_SetOnTXEvent()

Description

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

Prototype

```
void USBD_CDC_SetOnTXEvent(USB_CDC_HANDLE hInst,
                           USB_EVENT_CALLBACK *pEventCb,
                           USB_EVENT_CALLBACK_FUNC *pfEventCb,
                             void *pContext);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pEventCb	Pointer to a USB_EVENT_CALLBACK structure.
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

Table 9.14: USBD_CDC_SetOnTXEvent() parameter list

Additional information

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 has the follwing 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 USB_SetOnEvent function

Table 9.15: Event callback function parameter list

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

Table 9.16: USB events

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_CDC_GetNumBytesRemToWrite(_hInst) == 0) {
            <.. prepare next data for writing..>
            // Send next packet of data.
            r = USBD_CDC_Write(_hInst, &ac[0], 200, -1);
            if (r < 0) {
              <.. error handling..>
            }
   }
}


// Main programm.


// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USBD_CDC_SetOnTXEvent(hInst, &_usb_callback, _OnEvent, NULL);

// Send the first packet of data using an asynchronous write operation.
r = USBD_CDC_Write(_hInst, &ac[0], 200, -1);
if (r < 0) {
  <.. error handling..>
}
<.. do anything else here while the whole data is send..>
```


9.4.2.12 USBD_CDC_UpdateSerialState()

Description

Updates the control line state of the.

Prototype

```
void USBD_CDC_UpdateSerialState(USB_CDC_HANDLE hInst,
                                USB_CDC_SERIAL_STATE * pSerialState);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pSerialState	Pointer to the `USB_CDC_SERIAL_STATE structure`, refer to *USB_CDC_SERIAL_STATE* on page 298.

Table 9.17: USBD_CDC_UpdateSerialState() parameter list

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 `USBD_CDC_WaitForTXReady()`.

9.4.2.13 USBD_CDC_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

```
void USBD_CDC_Write(USB_CDC_HANDLE hInst, const void* pData,
                    unsigned NumBytes, int Timeout);
```

Parameter	Description
`hInst`	Handle to a valid CDC instance, returned by USBD_CDC_Add().
`pData`	Pointer to data that should be sent to the host.
`NumBytes`	Number of bytes to write.
`Timeout`	Timeout in milliseconds. 0 means infinite. If `Timeout` is -1, the function returns immediately and the transfer is processed asynchronously.

Table 9.18: USBD_CDC_Write() parameter list

Return value

== 0:	Successful started an asynchronous write transfer or a timeout has occured 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. A synchronous write transfer (`Timeout` >= 0) will always block until the transfer (including all pending transfers) are finished.

In order to synchronize, `USBD_CDC_WaitForTX()` needs to be called. Another synchronisation method would be to periodically call `USBD_CDC_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_CDC_CancelWrite()`.

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

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

9.4.2.14 USBD_CDC_WaitForRX()

Description

This function is to be used in combination with USBD_CDC_ReadOverlapped(). This function waits for the reading data transfer from the host to complete.

Prototype

```
int USBD_CDC_WaitForRX(USB_CDC_HANDLE hInst, unsigned Timeout);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
Timeout	Timeout in milliseconds. 0 means infinite.

Table 9.19: USBD_CDC_WaitForRX() parameter list

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.
This function blocks until the number of bytes specified by USBD_CDC_ReadOverlapped() has been read from the host.

9.4.2.15 USBD_CDC_WaitForTX()

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_WaitForTX(USB_CDC_HANDLE hInst, unsigned Timeout);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
Timeout	Timeout in milliseconds. 0 means infinite.

Table 9.20: USBD_CDC_WaitForTX() parameter list

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.

9.4.2.16 USBD_CDC_WaitForTXReady()

Description

This function is used in combination with a non-blocking call to USBD_CDC_Write(), it waits until a new asynchronous write data transfer will be accepted by the USB stack.

Prototype

```
int USBD_CDC_WaitForTXReady(USB_CDC_HANDLE hInst, int Timeout);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
Timeout	Timeout in milliseconds. 0 means infinite. If Timeout is negative, the function will return immediately.

Table 9.21: USBD_CDC_WaitForTXReady() parameter list

Return value

0: A new asynchronous write data transfer will be accepted.
1: The write queue is full,
a call to USBD_CDC_Write() would return USB_STATUS_EP_BUSY.

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_CDC_WaitForTxReady(hInst, 0);
  // Issue write transfer.
  if (USBD_CDC_Write(hInst, pData, NumBytes, -1) < 0) {
    <.. error handling..>
  }
}
```


9.4.2.17 USBD_CDC_WriteSerialState()

Description

Sends the current control line serial state to the host.

Prototype

```
void USBD_CDC_WriteSerialState(USB_CDC_HANDLE hInst);
```

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

Table 9.22: USBD_CDC_WriteSerialState() parameter list

Additional information

This function shall be called in order to inform the host about the control serial state of the CDC instance. It may be called within the same function or in another task dedicated to sending the serial state.
This function blocks until the host has received the serial state.
The current control line serial state can be set using USBD_CDC_UpdateSerialState().

9.4.2.18 USBD_CDC_GetNumBytesRemToRead()

Description

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

Prototype

```
unsigned USBD_CDC_GetNumBytesRemToRead(USB_CDC_HANDLE hInst);
```

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

Table 9.23: USBD_CDC_GetNumBytesRemToRead() parameter list

Return value

>= 0: Number of bytes which still have to be read.
< 0: Error.

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 = USBD_CDC_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_CDC_GetNumBytesRemToRead(hInst) > 0) {
    USB_OS_Delay(50);
  }
}
```


9.4.2.19 USBD_CDC_GetNumBytesRemToWrite()

Description

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

Prototype

```
unsigned USBD_CDC_GetNumBytesToWrite(USB_CDC_HANDLE hInst);
```

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

Table 9.24: USBD_CDC_GetNumBytesToWrite() parameter list

Return value

>= 0: Number of bytes which still have to be written.
< 0: Error.

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


9.4.2.20 USBD_CDC_GetNumBytesInBuffer()

Description

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

Prototype

```
unsigned USBD_CDC_GetNumBytesInBuffer(USB_CDC_HANDLE hInst);
```

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

Table 9.25: USBD_CDC_GetNumBytesInBuffer() parameter list

Return value

> 0: Number of bytes which have been stored in the internal buffer.
== 0: Nothing stored yet.

Additional information

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

9.4.3 Data structures

9.4.3.1 USB_CDC_INIT_DATA

Description

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

Prototype

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

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

Table 9.26: USB_CDC_INIT_DATA elements

9.4.3.2 USB_CDC_ON_SET_BREAK

Description

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

Prototype

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

Member	Description
BreakDuration	The BreakDuration gives the length of time, in milliseconds, of the break signal.

Table 9.27: USB_CDC_ON_SET_LINE_CODING elements

Additional Information

This type of callback is used to notify the application that the host has sent a break condition. If BreakDuration is 0xFFFF, then the host will send a break until another SendBreak request is received with BreakDuration of 0x0000. Since the callback is called within an interrupt service routine it should execute quickly.

9.4.3.3 USB_CDC_ON_SET_LINE_CODING

Description

Callback function to register line-coding changes.

Prototype

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

Member	Description
pLineCoding	Pointer to `USB_CDC_LINE_CODING` structure

Table 9.28: USB_CDC_ON_SET_LINE_CODING elements

Additional Information

This type of callback is used to notify the application that the host has changed the line coding. For example the baud rate has been changed. The new “line-coding” is passed through the structure `USB_CDC_LINE_CODING`. Refer to *USB_CDC_LINE_CODING* on page 297 for more information about the elements of this structure. Since the callback is called within an interrupt service routine it should execute quickly.

9.4.3.4 USB_CDC_ON_SET_CONTROL_LINE_STATE

Description

Callback function to register control-line-state changes.

Prototype

```
typedef void USB_CDC_ON_SET_CONTROL_LINE_STATE(USB_CDC_CONTROL_LINE_STATE *
pLineState);
```

Member	Description
pLineState	Pointer to USB_CDC_CONTROL_LINE_STATE structure

Table 9.29: USB_CDC_ON_SET_CONTROL_LINE_STATE **elements**

Additional Information

This type of callback is used to notify the application that the host has changed the state of the control lines. The new “line-states” are passed through the structure USB_CDC_CONTROL_LINE_STATE. Refer to *USB_CDC_CONTROL_LINE_STATE* on page 299 for more information about the elements of this structure. Since the callback is called within an interrupt service routine it should execute quickly.

9.4.3.5 USB_CDC_LINE_CODING

Description

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

Prototype

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

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

Table 9.30: USB_CDC_LINE_CODING elements

9.4.3.6 USB_CDC_SERIAL_STATE

Description

Structure that contains the serial state that can be send to the host.

Prototype

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

Member	Description
DCD	Data Carrier Detect: Tells that the device is connected to the telephone line.
DSR	Data Set Read: Device is ready to receive data.
Break	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: Acknowledges RTS and allows the host to transmit.

Table 9.31: USB_CDC_LINE_CODING elements

Additional Information

All members of the structure may have value 0 (false) or 1 (true).

9.4.3.7 USB_CDC_CONTROL_LINE_STATE

Description

Structure that contains the new control line state sent by the host.

Prototype

```
typedef struct {
  U8 DTR;
  U8 RTS;
} USB_CDC_CONTROL_LINE_STATE;
```

Member	Description
DTR	Data Terminal Ready
RTS	Request To Send

Table 9.32: USB_CDC_CONTRL_LINE_STATE elements

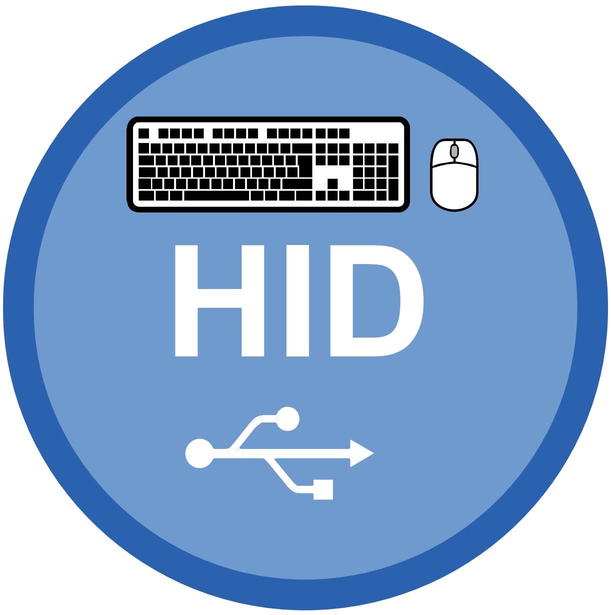
Additional Information

All members of the structure may have value 0 (false) or 1 (true).

Chapter 10

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.



10.1 Overview

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

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

Method of communication

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

10.1.1 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

10.1.2 Categories

Devices which are in the HID class generally fall into one of two categories:

True HIDs and *vendor specific HIDs*, explained below. One or more examples for both categories are provided.

10.1.2.1 True HIDs

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
- Game pad
- Front-panel controls - for example, switches and buttons.

10.1.2.2 Vendor specific HIDs

These are HID devices communicating with an application program. The host OS loads the same driver it loads for any “true HID” and will automatically enumerate the device, but it 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. 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.

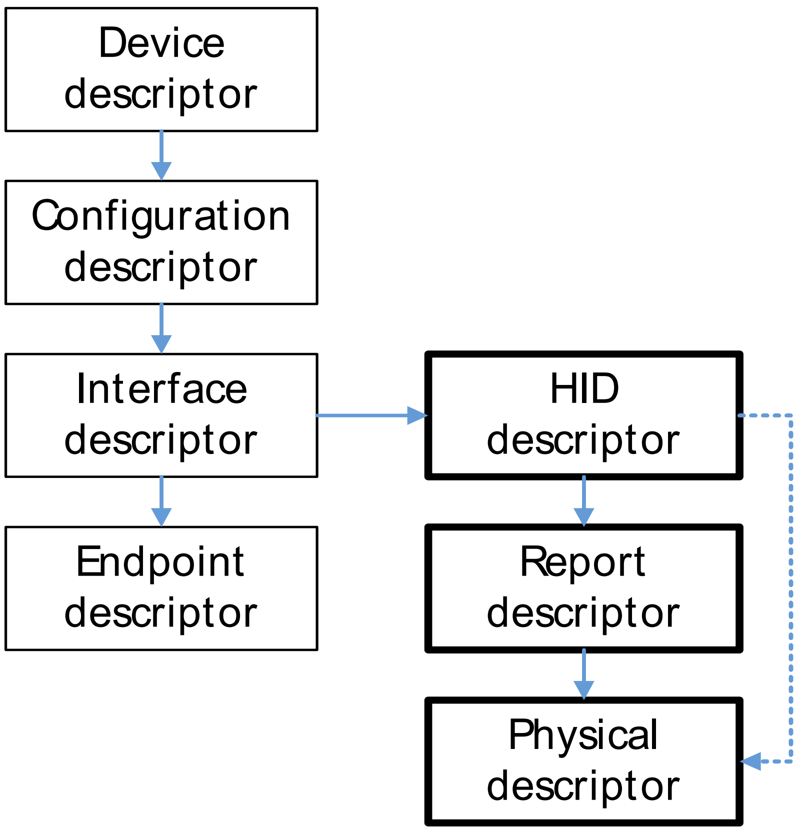
Typical examples

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

10.2 Background information

10.2.1 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, *www.usb.org*, for detailed information about the USB HID standard.



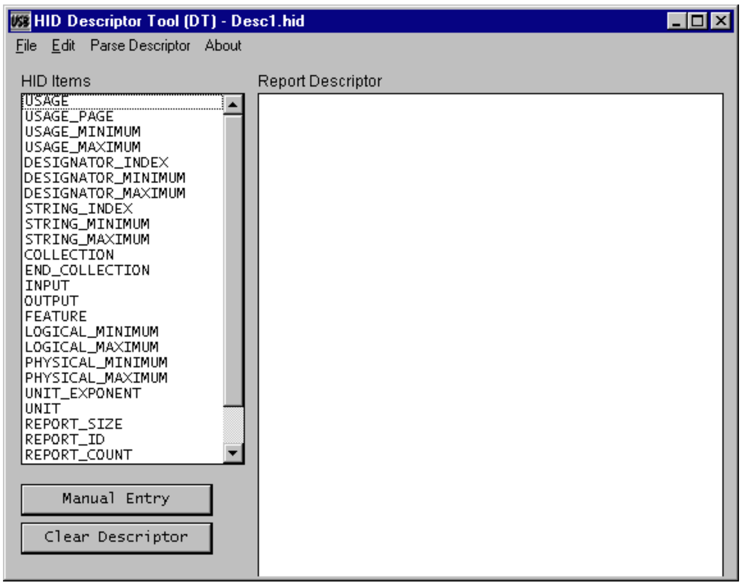
10.2.1.1 HID descriptor

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.

10.2.1.2 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.

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



10.2.1.3 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.

10.3 Configuration

10.3.1 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 *Configuration* on page 40 for detailed information about the functions which must be adapted before you can release a final product version.

10.3.2 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 example(s) use the value 0x12; this value is defined at the top of the application program with the macro `USB_HID_DEFAULT_VENDOR_PAGE`.

10.4 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
`HID_Mouse.c`	Simple mouse example. (“True HID” example)
`HID_Echo1.c`	Modified echo server. (“vendor specific” example)

Table 10.1: Supplied example HID applications

10.4.1 HID_Mouse.c

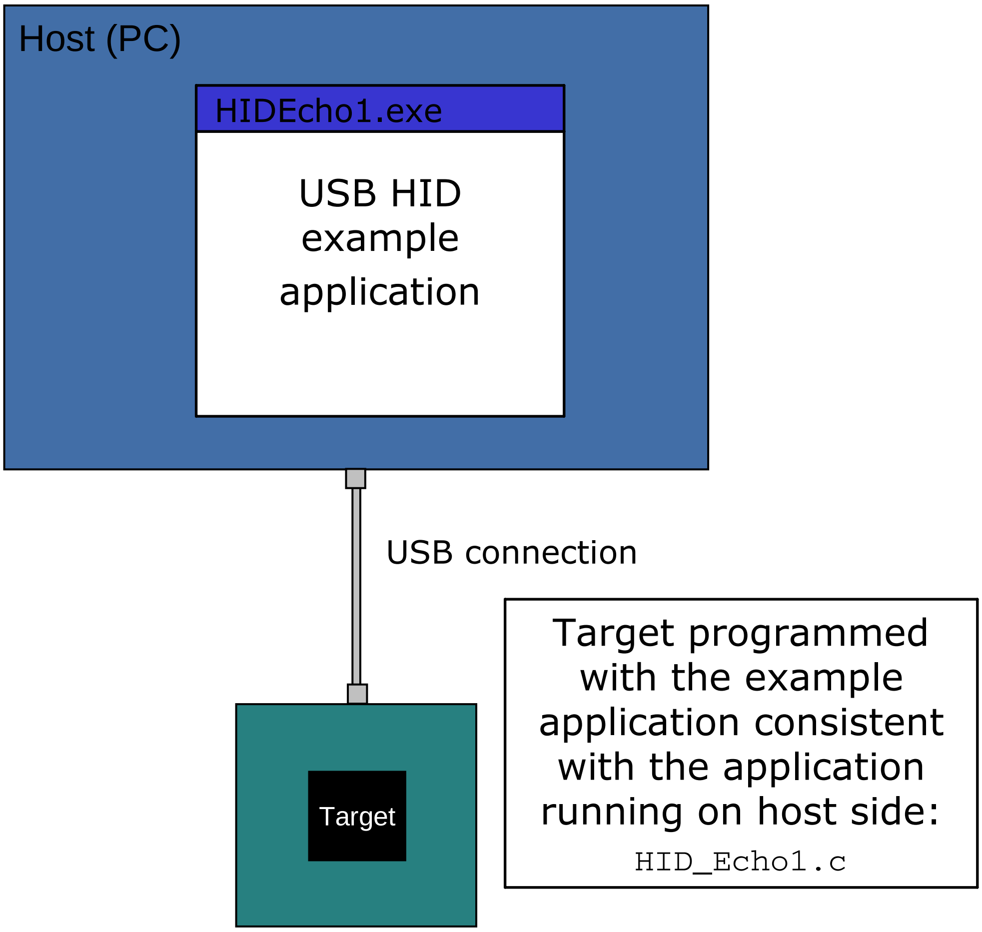
`HID_Mouse.c` is a typical example for a “true HID” implementation. The host identifies the device which is programmed with this example as a mouse. After the device is enumerated, it moves the mouse cursor in an endless loop to the left and after a short delay back to the right.

10.4.1.1 Running the example

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

10.4.2 HID_Echo1.c

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

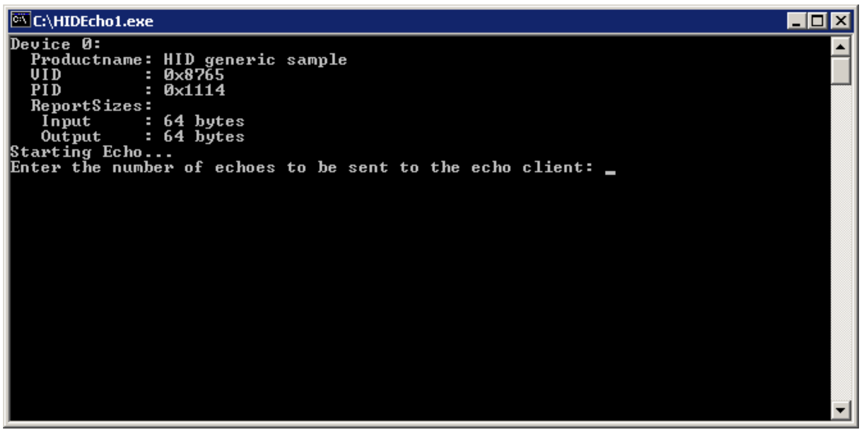


To use this application, include the source code file `HID_Echo1.c` into your project and compile and download it into your target. Run `HIDEcho1.exe` after the target is connected to the host and the enumeration process has been completed. The PC application is supplied as executable in the `HID\SampleApp\Exe` directory. The source code of the PC example is also supplied. Refer to section *Compiling the PC example application* on page 309 for more information to the PC example project.

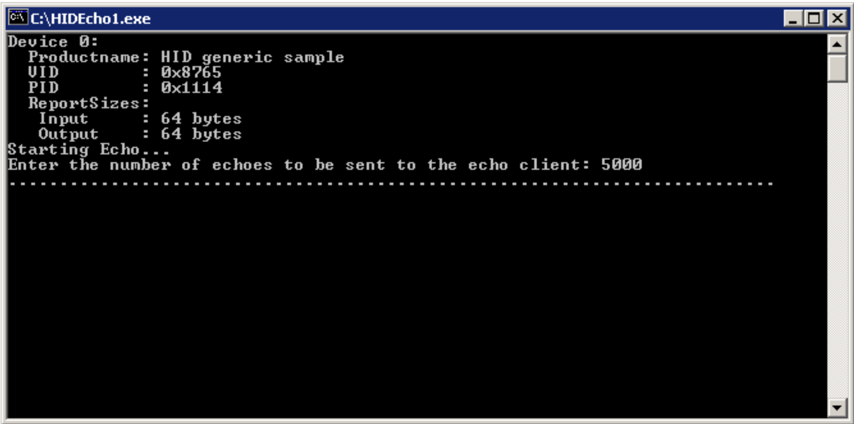
10.4.2.1 Running the example

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

Example output of `HID_Echo1.exe`:



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



10.4.2.2 Compiling the PC example application

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

Note: The Microsoft Windows Driver Development Kit (DDK) is required to compile the HID host example application. Refer to *http://www.microsoft.com/whdc/devtools/ddk/default.mspx* for more information.

10.5 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.

10.5.1 Target interface function list

Function	Description
API functions	
USBD_HID_Add()	Adds HID-class to the emUSB-Device interface.
USBD_HID_GetNumBytesInBuffer()	Returns the number of bytes in the internal read buffer.
USBD_HID_GetNumBytesRemToRead()	Returns the number of bytes which still have to be read.
USBD_HID_GetNumBytesRemToWrite()	Returns the number of bytes which still have to be written.
USBD_HID_Read()	Reads data from the host.
USBD_HID_ReadOverlapped()	Non-blocking version of USBD_HID_Read()
USBD_HID_WaitForRX()	Initiates a read data transfer.
USBD_HID_WaitForTX()	Waits for a non-blocking write operation that is pending.
USBD_HID_Write()	Writes data to the host.
USBD_HID_SetOnGetReportRequest	Sets a callback for the GET_REPORT HID command.
USBD_HID_SetOnGetReportRequest	Sets a callback for the GET_REPORT HID command.
Data structures	
USB_HID_INIT_DATA	Initialization structure that is required when adding a HID interface.

Table 10.2: USB-HID target interface function list

10.5.2 USB-HID functions

10.5.2.1 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);
```

Parameter	Description
pInitData	Pointer to a USB_HID_INIT_DATA structure. For detailed information about the USB_HID_INIT_DATA structure, refer to *USB_HID_INIT_DATA* on page 322.

Table 10.3: USBD_HID_Add() parameter list

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.

10.5.2.2 USBD_HID_GetNumBytesInBuffer()

Description

The function will return the number of bytes available in the internal read buffer.

Prototype

```
unsigned USBD_HID_GetNumBytesInBuffer (USB_HID_HANDLE hInterface);
```

Parameter	Description
hInterface	Handle to a HID instance.

Table 10.4: USBD_HID_GetNumBytesInBuffer() parameter list

Return value

>= 0: Number of bytes in the internal read buffer.

10.5.2.3 USBD_HID_GetNumBytesRemToRead()

Description

This function is to be used in combination with USBD_HID_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_HID_GetNumBytesRemToRead (USB_HID_HANDLE hInterface);
```

Parameter	Description
hInterface	Handle to a HID instance.

Table 10.5: USBD_HID_GetNumBytesRemToRead() parameter list

Return value

>= 0:   Number of bytes which have not yet been read.

Additional information

Alternatively the blocking function USBD_HID_WaitForRX() can be used.

10.5.2.4 USBD_HID_GetNumBytesRemToWrite()

Description

This function is to be used in combination with a non-blocking call to USBD_HID_Write().
After starting the write operation this function can be used to periodically check how many bytes still have to be written.

Prototype

```
unsigned USBD_HID_GetNumBytesRemToWrite (USB_HID_HANDLE hInterface);
```

Parameter	Description
hInterface	Handle to a HID instance.

Table 10.6: USBD_HID_GetNumBytesToWrite() parameter list

Return value

>= 0: Number of bytes which have not yet been written.

Additional information

Alternatively the blocking function USBD_HID_WaitForTX() can be used.

10.5.2.5 USBD_HID_Read()

Description

Reads data from the host with a given timeout. This function blocks until the timeout has been reached, it has received NumBytes or until the device is disconnected from the host.

Prototype

```
int USBD_HID_Read (USB_HID_HANDLE hInterface,
                   void* pData,
                   unsigned NumBytes,
                    unsigned ms);
```

Parameter	Description
hInterface	Handle to a HID instance.
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
ms	Timeout given in milliseconds. A zero value results in an infinite timeout.

Table 10.7: USBD_HID_Read() parameter list

Return value

== NumBytes:	Requested data was succesfully read within the given timeout.
>= 0, < NumBytes:	Timeout has occured. Number of bytes that have been read within the given timeout.
< 0:	Returns a USB_STATUS_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.

10.5.2.6 USBD_HID_ReadOverlapped()

Description

Reads data from the host asynchronously.

Prototype

```
int USBD_HID_ReadOverlapped (USB_HID_HANDLE hInterface,
                             void* pData,
                               unsigned NumBytes);
```

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

Table 10.8: USBD_HID_ReadOverlapped() parameter list

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, USBD_HID_WaitForRX() needs to be called. Alternatively the function USBD_HID_GetNumBytesRemToWrite() 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.

10.5.2.7 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 synchronise. It will block until the transfer is completed.

Prototype

```
int USBD_HID_WaitForRX (USB_HID_HANDLE hInterface, unsigned Timeout);
```

Parameter	Description
hInterface	Handle to a HID instance.
Timeout	Timeout given in milliseconds. A zero value results in an infinite timeout.

Table 10.9: USBD_HID_WaitForRX() parameter list

Return value

0: Transfer completed.
1: Timeout occurred.

10.5.2.8 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

```
void USBD_HID_WaitForTX (USB_HID_HANDLE hInterface, unsigned Timeout);
```

Parameter	Description
`hInterface`	Handle to a HID instance.
`Timeout`	Timeout given in milliseconds. A zero value results in an infinite timeout.

Table 10.10: USBD_HID_WaitForTX() parameter list

Return value

0: Transfer completed.
1: Timeout occurred.

10.5.2.9 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 hInterface,
                    const void* pData,
                    unsigned NumBytes,
                     int Timeout);
```

Parameter	Description
hInterface	Handle to a HID instance.
pData	Pointer to data that should be sent to the host.
NumBytes	Number of bytes to write.
Timeout	Timeout in milliseconds. 0 means infinite. If Timeout is -1, the function returns immediately and the transfer is processed asynchronously.

Table 10.11: USBD_HID_Write() parameter list

Return value

== 0:	Successful started an asynchronous write transfer or a timeout has occured 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, USBD_HID_WaitForTX() needs to be called. Another synchronisation method would be to periodically call USBD_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.

10.5.2.10USBD_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 hInterface,
                   USB_HID_ON_GETREPORT_REQUEST_FUNC * pfOnGetReportRequest);
```

Parameter	Description
hInterface	Handle to a HID instance.
pfOnGetReportRequest	Pointer to a function of type USB_HID_ON_GETREPORT_REQUEST_FUNC.

Table 10.12: USBD_HID_SetOnGetReportRequest() parameter list

Additional information

See the description of *USB_HID_ON_GETREPORT_REQUEST_FUNC* on page 324 for more details.

10.5.2.11 USBD_HID_SetOnSetReportRequest()

Description

Allows to set a callback for the SET_REPORT command. The SET_REPORT command is sent from the host to the device.

Prototype

```
void USBD_HID_SetOnGetReportRequest(USB_HID_HANDLE hInterface,
                    USB_HID_ON_SETREPORT_REQUEST_FUNC * pfOnSetReportRequest);
```

Parameter	Description
hInterface	Handle to a HID instance.
pfOnSetReportRequest	Pointer to a function of type USB_HID_ON_SETREPORT_REQUEST_FUNC.

Table 10.13: USBD_HID_SetOnSetReportRequest() parameter list

Additional information

See the description of *USB_HID_ON_SETREPORT_REQUEST_FUNC* on page 325 for more details.

10.5.3 Data structures

10.5.3.1 USB_HID_INIT_DATA

Description

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

Prototype

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

Member	Description
EPIn	Endpoint for sending data to the host.
EPOut	Endpoint for receiving data from the host.
pReport	Pointer to a report descriptor.
NumBytesReport	Size of the HID report descriptor.
BufferSize	Size of the buffer pointed to by pBuff
pBuff	Pointer to a buffer for receiving reports from the host via endpoint 0 (Set_Report request).

Table 10.14: USB_HID_INIT_DATA elements

Additional Information

This structure is used when the HID interface is added to emUSB-Device.

To be able to receive data from the host either an endpoint must be allocated (EPOut) or a buffer must be provided (BufferSize, pBuff). If 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.

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: *http://www.usb.org/developers/hidpage/*.

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

Example 1 (configure to receive reports via seperate endpoint)

Example excerpt from `HID_Mouse.c`:

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

  memset(&InitData, 0, sizeof(InitData));
  InitData.EPIn = USB_AddEP(USB_DIR_IN,  USB_TRANSFER_TYPE_INT, Interval, NULL, 0);
  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);
}
```

10.5.4 Type definitions

10.5.4.1 USB_HID_ON_GETREPORT_REQUEST_FUNC

Description

Callback function description which is used together with USBD_HID_SetOnGetReportRequest().

Prototype

```
typedef int USB_HID_ON_GETREPORT_REQUEST_FUNC(
    USB_HID_REPORT_TYPE ReportType,
    unsigned ReportId,
    const U8 ** ppData,
    U32 * pNumBytes
);
```

Member	Description
ReportType	HID report type, possible values are: USB_HID_REPORT_TYPE_INPUT USB_HID_REPORT_TYPE_OUTPUT USB_HID_REPORT_TYPE_FEATURE
ReportId	The ID of the report for which the GET_REPORT request has been sent.
ppData	[IN] == NULL -> previous data have been sent. != NULL -> Host has asked for data. Pointer to a pointer to the data to send via GET_REPORT request.
pNumBytes	[IN/OUT] == NULL -> previous data have been sent. != NULL -> Stores the number of bytes that shall be sent

Table 10.15: USB_HID_ON_GETREPORT_REQUEST_FUNC elements

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. USB_WriteEP0FromIsr needs to be called from user context.

Additional Information

In case ppData && pNumBytes == NULL return value is ignored.

10.5.4.2 USB_HID_ON_SETREPORT_REQUEST_FUNC

Description

Callback function description which is used together with USBD_HID_SetOnSetReportRequest().

Prototype

```
typedef void USB_HID_ON_SETREPORT_REQUEST_FUNC(
    USB_HID_REPORT_TYPE ReportType,
    unsigned ReportId,
    U32      NumBytes
);
```

Member	Description
ReportType	HID report type, possible values are: USB_HID_REPORT_TYPE_INPUT USB_HID_REPORT_TYPE_OUTPUT USB_HID_REPORT_TYPE_FEATURE
ReportId	The ID of the report for which the SET_REPORT request has been sent.
NumBytes	Number of bytes that will be sent from the host.

Table 10.16: USB_HID_ON_GETREPORT_REQUEST_FUNC elements

Additional Information

In case no EP Out was used with the HID interface, USBD_HID_Read can be used to read the report that has been sent from the host.

10.6 Host API

This chapter describes the functions that can be used with the Windows host system. This 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 USB-Bulk driver.

Purpose of the USB Host API functions

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

10.6.1 Host API function list

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

Table 10.17: USB-HID host interface function list

10.6.2 USB-HID functions

10.6.2.1 USBHID_Close()

Description

Closes the connection an open device.

Prototype

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

Parameter	Description
DeviceIndex	Index of the HID device. This is the bit number of the mask returned by `USBHID_GetNumAvailableDevices()`

Table 10.18: USBHID_Close() parameter list

10.6.2.2 USBHID_Open()

Description

Opens a handle to the device that shall be opened.

Prototype

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

Parameter	Description
DeviceIndex	Index of the HID device. This is the bit number of the mask returned by USBHID_GetNumAvailableDevices().

Table 10.19: USBHID_Open() parameter list

Return value

== 0: Opening was successful or already opened.
== 1: Error. Handle to the device could not opened.

10.6.2.3 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);
```

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

Table 10.20: USBHID_Init() parameter list

10.6.2.4 USBHID_Exit()

Description

Closes the connection to all open devices and deinitializes the HID module.

Prototype

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

10.6.2.5 USBHID_GetNumAvailableDevices()

Description

Returns the number of the available devices.

Prototype

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

Parameter	Description
pMask	Pointer to unsigned integer value which is used to store the bit mask of available devices. This parameter may be NULL.

Table 10.21: USBHID_GetNumAvailableDevices() parameter list

Return value

Returns the number of available devices.

Additional information

pMask will be filled by this routine. It shall be interpreted as bit mask where a bit set means this device is available. For example, device 0 and device 2 are available, if pMask has the value 0x00000005.

10.6.2.6 USBHID_GetProductName()

Description

Stores the name of the device into pBuffer.

Prototype

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

Parameter	Description
DeviceIndex	Index of the HID device. This is the bit number of the mask returned by USBHID_GetNumDevices().
pBuffer	Pointer to a buffer for the product name.
NumBytes	Size of the pBuffer in bytes.

Table 10.22: USBHID_GetProductName() parameter list

Return value

== 0: An error occurred.
== 1: Success.

10.6.2.7 USBHID_GetInputReportSize()

Description

Returns the input report size of the device.

Prototype

```
int USBHID_GetInputReportSize(unsigned Id);
```

Parameter	Description
DeviceIndex	Index of the HID device. This is the bit number of the mask returned by `USBHID_GetNumDevices()`.

Table 10.23: USBHID_GetInputReportSize() parameter list

Return value

== 0: An error occurred.
!= 0: Size of the report in bytes.

10.6.2.8 USBHID_GetOutputReportSize()

Description

Returns the output report size of the device.

Prototype

```
int USBHID_GetOutputReportSize(unsigned Id);
```

Parameter	Description
DeviceIndex	Index of the HID device. This is the bit number of the mask returned by `USBHID_GetNumDevices()`.

Table 10.24: USBHID_GetOutputReportSize() parameter list

Return value

== 0: An error occurred.
!= 0: Size of the report in bytes.

10.6.2.9 USBHID_GetProductId()

Description

Returns the Product ID of a device.

Prototype

```
U16 USBHID_GetProductId(unsigned Id);
```

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

Table 10.25: USBHID_GetProductId() parameter list

Return value

== 0: An error occurred.
!= 0: Product ID.

10.6.2.10USBHID_GetVendorId()

Description

Returns the Vendor ID of the device.

Prototype

```
U16 USBHID_GetVendorId(unsigned Id);
```

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

Table 10.26: USBHID_GetVendorId() parameter list

Return value

== 0: An error occurred.
!= 0: Vendor ID.

10.6.2.11 USBHID_RefreshList()

Description

Refreshes the connection list.

Prototype

```
void USBHID_RefreshList(void);
```

Additional information

Note that any open handle to the device will be closed while refreshing the connection list.

10.6.2.12 USBHID_SetVendorPage()

Description

Sets the vendor page so that all HID devices with the specified page will be found.

Prototype

```
void USBHID_SetVendorPage(U8 VendorPage);
```

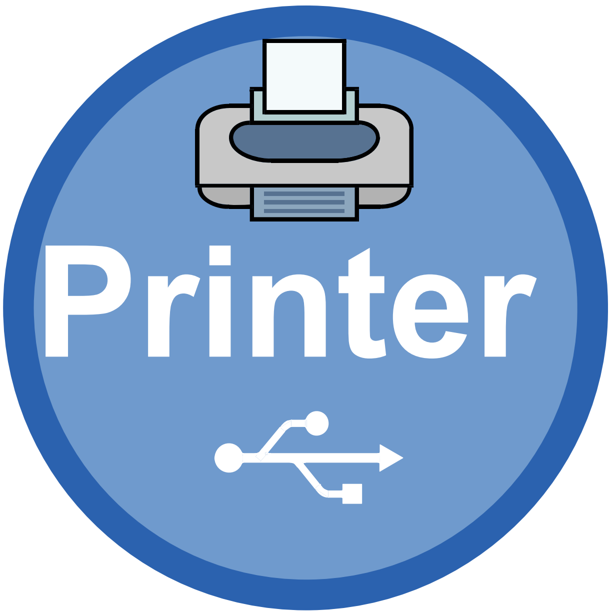
Parameter	Description
VendorPage	This parameter specifies the lower 8 bits of the vendor specific usage page number. It must be identical on both device and host.

Table 10.27: USBHID_SetVendorPage() parameter list

Chapter 11

Printer Class

This chapter describes how to get emUSB-Device up and running as a printer device.



11.1 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.

11.1.1 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.

11.2 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.

Source code of `USB_Printer.c`:

```
/*********************************************************************
*                   SEGGER MICROCONTROLLER GmbH & Co. KG             *
*           Solutions for real time microcontroller applications     *
**********************************************************************
*                                                                    *
*           (c) 2003-2011     SEGGER Microcontroller GmbH & Co KG    *
*                                                                    *
*           Internet: www.segger.com    Support:  support@segger.com *
*                                                                    *
**********************************************************************
*                                                                    *
*       USB device stack for embedded applications                   *
*                                                                    *
**********************************************************************
----------------------------------------------------------------------
File    : USB_Printer.c
Purpose : Sample implementation of USB printer device class
----------Literature--------------------------------------------------
Universal Serial Bus Device Class Definition for Printing Devices
Version 1.1 January 2000
--------  END-OF-HEADER  ---------------------------------------------
*/

#include <stdio.h>
#include <string.h>
#include "USB_PrinterClass.h"
#include "BSP.h"

/*********************************************************************
*
*       static data
*
**********************************************************************
*/
static U8 _acData[512];

/*********************************************************************
*
*       static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _GetDeviceIdString
*
*/
static const char * _GetDeviceIdString(void) {
  const char * s = "CLASS:PRINTER;MODEL:HP LaserJet 6MP;"
                   "MANUFACTURER:Hewlett-Packard;"
                   "DESCRIPTION:Hewlett-Packard LaserJet 6MP Printer;"
                   "COMMAND SET:PJL,MLC,PCLXL,PCL,POSTSCRIPT;";
  return s;
}

/*********************************************************************
*
*       _GetHasNoError
*
*/
static U8 _GetHasNoError(void) {
  return 1;
}

/*********************************************************************
*
*       _GetIsSelected
*
*/
static U8 _GetIsSelected(void) {
  return 1;
```



```
}

/*********************************************************************
*
*       _GetIsPaperEmpty
*
*/
static U8 _GetIsPaperEmpty(void) {
  return 0;
}

/*********************************************************************
*
*       _OnDataReceived
*
*/
static int _OnDataReceived(const U8 * pData, unsigned NumBytes) {
  USB_MEMCPY(_acData, pData, NumBytes);
  _acData[NumBytes] = 0;
  printf(_acData);
  return 0;
}

/*********************************************************************
*
*       _OnReset
*
*/
static void _OnReset(void) {

}

static USB_PRINTER_API _PrinterAPI = {
  _GetDeviceIdString,
  _OnDataReceived,
  _GetHasNoError,
  _GetIsSelected,
  _GetIsPaperEmpty,
  _OnReset
};

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

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);
void MainTask(void) {
  USBD_Init();
```

```
  USBD_SetDeviceInfo(&_DeviceInfo);
  USB_PRINTER_Init(&_PrinterAPI);
  USBD_Start();
  //
  // Loop: Receive data byte by byte, send back (data + 1)
  //
  while (1) {
    //
    // Wait for configuration
    //
    while ((USBD_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED))
           != USB_STAT_CONFIGURED) {
      BSP_ToggleLED(0);
      USB_OS_Delay(50);
    }
    USB_PRINTER_Task();
  }
}
/*************************** end of file ***************************/
```


11.3 Target API

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

11.3.1 Interface function list

Function	Description
API functions	
USB_PRINTER_Init()	Initializes the printer class.
USB_PRINTER_Task()	Processes the request from USB Host.
Advanced API functions	
USB_PRINTER_Read()	Reads data from the host.
USB_PRINTER_ReadTimed()	Reads data from host with a given timeout.
USB_PRINTER_Receive()	Reads data from host.
USB_PRINTER_ReceiveTimed()	Reads data from host with a given timeout.
USB_PRINTER_Write()	Writes data to the host.
USB_PRINTER_WriteTimed()	Writes data to the host with a given timeout.
Data structures	
USB_PRINTER_API	List of callback functions the PRINTER module should invoke when processing a request from the USB Host.

Table 11.1: USB-Printer interface API

11.3.2 API functions

11.3.2.1 USB_PRINTER_Init()

Description

Initializes the printer class.

Prototype

```
void USB_PRINTER_Init(USB_PRINTER_API * pAPI);
```

Parameter	Description
pAPI	Pointer to an API table that contains all callback functions that are necessary for handling the functionality of a printer.

Table 11.2: USB_PRINTER_Init() parameter list

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.

11.3.2.2 USB_PRINTER_Task()

Description

Processes the request received from the USB Host.

Prototype

```
void USB_PRINTER_Task(void);
```

Additional information

This function blocks as long as the USB device is connected to USB host. It handles the requests by calling the function registered in the call to `USB_PRINTER_Init()`.

11.3.2.3 USB_PRINTER_Read()

Description

Reads data from the host. This function blocks until NumBytes have been read or until the device is disconnected from the host.

Prototype

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

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

Table 11.3: USB_PRINTER_Read() parameter list

Return value

== NumBytes:	Number of bytes that have been read.
!= NumBytes:	Returns a USB_STATUS_ERROR.

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.

11.3.2.4 USB_PRINTER_ReadTimed()

Description

Reads data from the host with a given timeout.

Prototype

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

Parameter	Description
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
ms	Timeout given in milliseconds. A zero value results in an infinite timeout.

Table 11.4: USB_PRINTER_ReadTimed() parameter list

Return value

== NumBytes: Number of bytes that have been read within the given timeout.
!= NumBytes: Returns a USB_STATUS_ERROR or USB_STATUS_TIMEOUT.

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.

11.3.2.5 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);
```

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

Table 11.5: USB_PRINTER_Receive() parameter list

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: Returns a USB_STATUS_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.

11.3.2.6 USB_PRINTER_ReceiveTimed()

Description

Reads data from host. 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);
```

Parameter	Description
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
ms	Timeout given in milliseconds. A zero value results in an infinite timeout.

Table 11.6: USB_PRINTER_ReceiveTimed() parameter list

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: Returns a USB_STATUS_ERROR or USB_STATUS_TIMEOUT.

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.

11.3.2.7 USB_PRINTER_Write()

Description

Writes data to the host.

Prototype

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

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

Table 11.7: USB_PRINTER_Write() parameter list

Return value

> 0: Number of bytes that have been written.
== 0: Error.

Additional information

This function is blocking.

11.3.2.8 USB_PRINTER_WriteTimed()

Description

Sends data to the host with a timeout option.

Prototype

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

Parameter	Description
pData	Data that should be written.
NumBytes	Number of bytes to write.
ms	Timeout in milliseconds. A zero value results in an infinite timeout.

Table 11.8: USB_PRINTER_WriteTimed() parameter list

Return value

> 0: Number of bytes that have been written.
== 0: Error.

Additional information

This function blocks a task until all data has been written or a timeout occurred. In case of a reset or a disconnect USB_STATUS_ERROR is returned.
Data structures

11.3.2.9 USB_PRINTER_API

Description

Initialization structure that is needed when adding a printer interface to emUSB-Device. It holds pointer to callback functions the interface invokes when it processes request from USB host.

Prototype

```
typedef struct {
  const char * (*pfGetDeviceIdString)(void);
  int          (*pfOnDataReceived)(const U8 * pData, unsigned NumBytes);
  U8           (*pfGetHasNoError)(void);
  U8           (*pfGetIsSelected)(void);
  U8           (*pfGetIsPaperEmpty)(void);
  void         (*pfOnReset)(void);
} USB_PRINTER_API;
```

Member	Description
pfGetDeviceIdString	The library calls this function when the USB host requests the printer's identification string. This string shall confirm to the IEEE 1284 Device ID Syntax: Example: "CLASS:PRINTER;MODEL:HP LaserJet 6MP;MANUFACTURER:Hewlett-Packard;DESCRIPTION:Hewlett-Packard LaserJet 6MP Printer;COMMAND SET:PJL,MLC,PCLXL,PCL,POSTSCRIPT;"
pfOnDataReceived	This function is called when data arrives from USB host.
pfGetHasNoError	This function should return a non-zero value if the printer has no error.

Table 11.9: USB_PRINTER_API elements

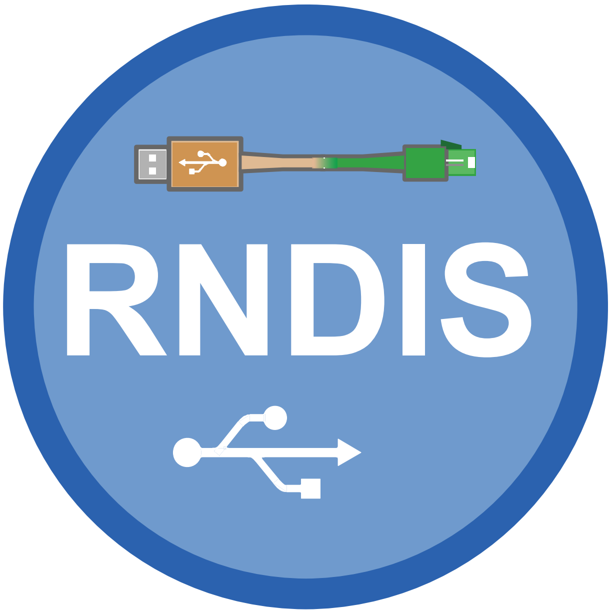
Member	Description
pfGetIsSelected	This function should return a non-zero value if the printer is selected.
pfGetIsPaperEmpty	This function should return a non-zero value if the printer is out of paper.
pfOnReset	The library calls this function if the USB host sends a soft reset command.

Table 11.9: USB_PRINTER_API elements

Chapter 12

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.



12.1 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 embOS/IP 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 120MHz, a transfer speed of about 5 MByte/sec can be achieved when using a high-speed USB connection.

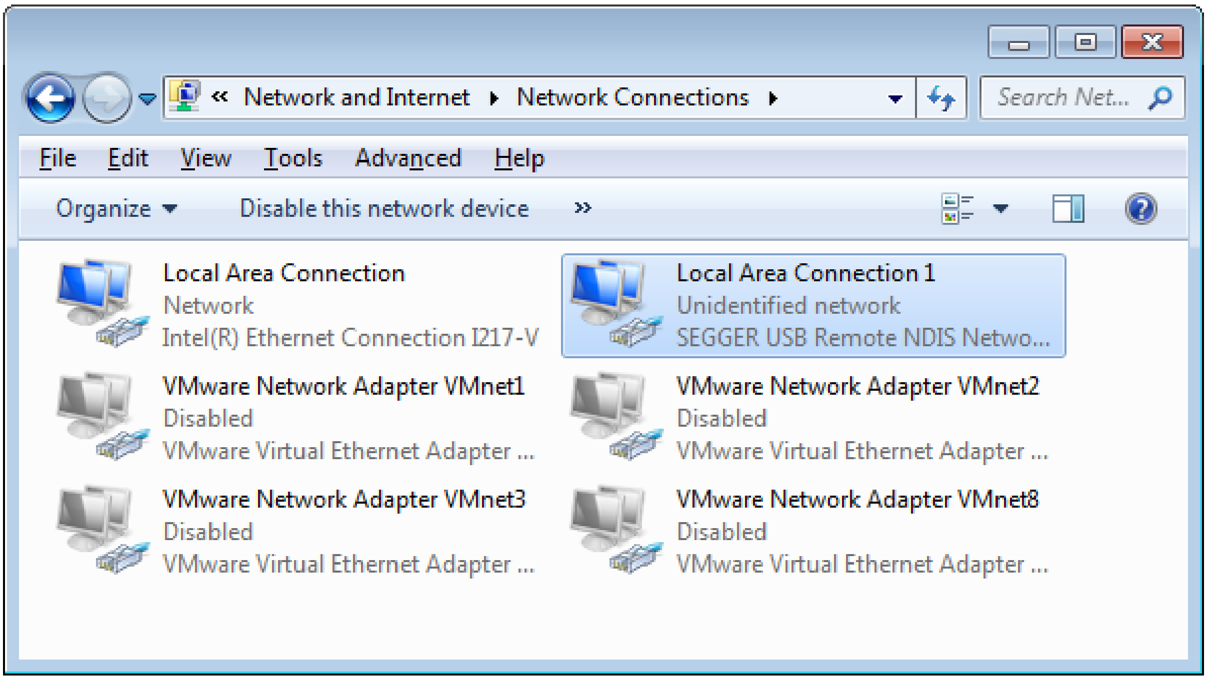
USB RNDIS is supported by all Windows operating systems starting with Windows XP, as well as by Linux with kernel versions newer than 2.6.34. An .inf file is required for the installation on Microsoft Windows systems older than Windows 7. The emUSB-Device-RNDIS package includes .inf files for Windows versions older than Windows 7. OS X will require a third-party driver to work with RNDIS, which can be downloaded from here: http://joshuawise.com/horndis

emUSB-Device-RNDIS contains the following components:

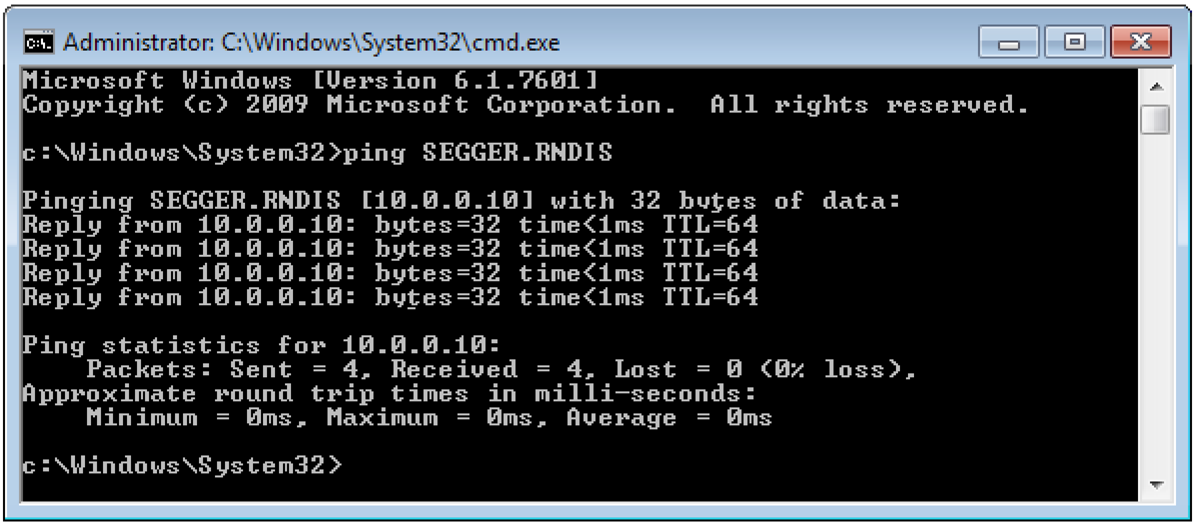
- Generic USB handling
- RNDIS device class implementation
- Network interface driver which uses embOS/IP as TCP/IP stack.
- A sample application demonstrating how to work with RNDIS.

12.1.1 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.



12.1.2 Additional information

More technical details about RNDIS can be found here:

http://msdn.microsoft.com/en-us/library/windows/hardware/
ff570660%28v=vs.85%29.aspx

12.2 Configuration

12.2.1 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.

12.2.2 Final configuration

The configuration must only be modified when emUSB-Device is used in your final product. Refer to section *Configuration* on page 40 to get detailed information about the general emUSB-Device configuration functions which have to be adapted.

12.2.3 Class specific configuration

RNDIS specific device information must be provided by the application via the function `USBD_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 embOS/IP.

12.3 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 embOS/IP middleware component is required. To test the emUSB-Device-RNDIS component any of the embOS/IP 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 choosen embOS/IP sample can be tested using the appropriate methods.

12.3.0.1 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 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);
  USBD_RNDIS_SetDeviceInfo(&USB_RNDIS_DeviceInfo);
  _AddRNDIS();
  OS_CREATETASK(&_RNDISTCB,"USB RNDISTask",
                _RndisTask, TASK_PRIO_RNDIS_TASK, _aRNDISStack);
  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 RNDIS interface is added to it. The function `_AddRNDIS()` configures all required endpoints and configures the HW address of the PC network interface.

```
// _AddRNDIS() - excerpt from IP_Config_RNDIS.c
static U8 _abReceiveBuffer[USB_MAX_PACKET_SIZE];

static void _AddRNDIS(void) {  USB_RNDIS_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,  5, NULL, 0);
  InitData.pDriverAPI  = &USB_Driver_IP_NI;
  InitData.DriverData.pHWAddr         = "\x00\x22\xC7\xFF\xFF\xF3";
  InitData.DriverData.NumBytesHWAddr  = 6;
  USBD_RNDIS_Add(&InitData);
}
```

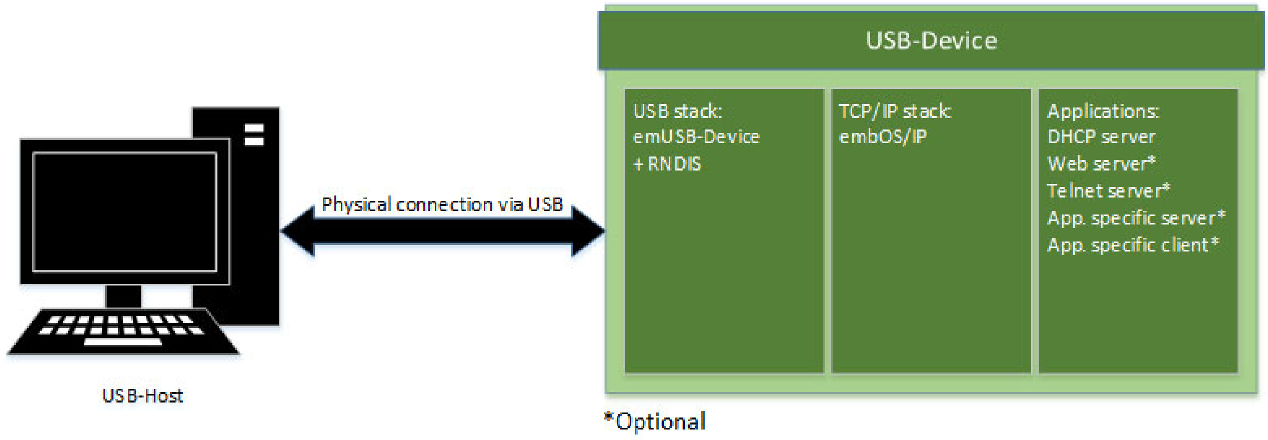
The size of _acReceiveBuffer buffer must be a multiple of USB max packet size. The `USB_MAX_PACKET_SIZE` define is set to the correct max packet size value for the corresponding speed (full or high) and is used in the samples to declare buffer sizes. `USB_Driver_IP_NI` is the network interface driver which implements the connection to the IP stack. The HW address configured here is assigned to the PC network interface. The HW address of the IP stack is configured in the `IP_X_Config()` function of embOS/IP as described below.

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 embOS**/**IP 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.
  //
  IP_AddEtherInterface(&USB_IP_Driver);
  IP_SetHWAddr("\x00\x22\xC7\xFF\xFF\xFF");
  IP_SetAddrMask(0x0A00000A, 0xFF000000);
  IP_SetIFaceConnectHook(IFaceId, _Connect);
  IP_SetIFaceDisconnectHook(IFaceId, _Disconnect);
  <...>
}
```


12.4 RNDIS + embOS/IP 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 behavior 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.

12.5 Target API

Function	Description
API functions	
USBD_RNDIS_Add()	Adds a RNDIS-class interface to the USB stack.
USBD_RNDIS_Task()	Handles the RNDIS protocol.
USBD_RNDIS_SetDeviceInfo()	Set device information used during USB enumeration.
Data structures	
USB_RNDIS_INIT_DATA	Initialization data for RNDIS interface.
USB_RNDIS_DEVICE_INFO	Network interface driver API functions.
USB_IP_NI_DRIVER_API	Configuration data for the network interface driver.
USB_IP_NI_DRIVER_DATA	Structure containing device information used during USB enumeration.

Table 12.1: List of emUSB-Device RNDIS module functions and data structures

12.5.1 API functions

12.5.1.1 USBD_RNDIS_Add()

Description

Adds an RNDIS-class interface to the USB stack.

Prototype

```
void USBD_RNDIS_Add(const USB_RNDIS_INIT_DATA * pInitData);
```

Parameter	Description
pInitData	IN: Pointer to a USB_RNDIS_INIT_DATA structure. OUT: ---

Table 12.2: USBD_RNDIS_Add() parameter list

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* on page 366 for more information.

12.5.1.2 USBD_RNDIS_Task()

Description

Handles the RNDIS protocol.

Prototype

```
void USBD_RNDIS_Task(void);
```

Additional information

The function should be called periodically after the USB device has been successfully enumerated and configured. The function returns when the USB device is detached or suspended. For a sample usage refer to *IP_Config_RNDIS.c in detail* on page 359.

12.5.1.3 USBD_RNDIS_SetDeviceInfo()

Description

Provides device information used during USB enumeration to the stack.

Prototype

```
void USBD_RNDIS_SetDeviceInfo(USB_RNDIS_DEVICE_INFO * pDeviceInfo);
```

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

Table 12.3: USBD_RNDIS_SetDeviceInfo() parameter list

Additional information

See for a description of the structure

12.5.2 Data structures

12.5.2.1 USB_RNDIS_INIT_DATA

Description

Initialization data for RNDIS interface.

Prototype

```
typedef struct USB_RNDIS_INIT_DATA {
  U8 EPIn;
  U8 EPOut;
  U8 EPInt;
  const USB_IP_NI_DRIVER_API * pDriverAPI;
  USB_IP_NI_DRIVER_DATA        DriverData;
} USB_RNDIS_INIT_DATA;
```

Member	Description
EPIn	Endpoint for sending data to the host.
EPOut	Endpoint for receiving data from the host.
EPInt	Endpoint for sending status information.
pDriverAPI	Pointer to the Network interface driver API. (See *USB_IP_NI_DRIVER_API* on page 368)
DriverData	Configuration data for the network interface driver. (See *USB_IP_NI_DRIVER_DATA* on page 374)

Table 12.4: USB_RNDIS_INIT_DATA elements

Additional information

This structure holds the endpoints that should be used by the RNDIS interface (EPin, EPOut and EPInt). Refer to *USBD_AddEP()* on page 55 for more information about how to add an endpoint.

12.5.2.2 USB_RNDIS_DEVICE_INFO

Description

Device information that must be provided by the application via the function USBD_RNDIS_SetDeviceInfo() before the USB stack is started using USBD_Start().

Prototype

```
typedef struct {
  U32   VendorId;
  char *sDescription;
  U16   DriverVersion;
} USB_RNDIS_DEVICE_INFO;
```

Member	Description
VendorId	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 retuned value are used.
sDecription	0-terminated ASCII string describing the device. The string is then sent to the host system.
DriverVersion	16-bit value representing the firmware version. The high-order byte specifies the major version and the low-order byte the minor version

Table 12.5: USB_RNDIS_DEVICE_INFO elements

12.5.3 Driver interface

This section describes the interface to IP stack.

12.5.3.1 USB_IP_NI_DRIVER_API

Description

This structure contains the callback functions for the network interface driver.

Prototype

```
typedef struct USB_IP_NI_DRIVER_API {
  void    (*pfInit)              (const USB_RNDIS_DRIVER_DATA * pDriverData,
                                  USB_IP_WRITE_PACKET *pfWritePacket);
  void *  (*pfGetPacketBuffer)   (unsigned NumBytes);
  void    (*pfWritePacket)       (const void * pData, unsigned NumBytes);
  void    (*pfSetPacketFilter)   (U32 Mask);
  int     (*pfGetLinkStatus)     (void);
  U32     (*pfGetLinkSpeed)      (void);
  void    (*pfGetHWAddr)         (U8 * pAddr, unsigned NumBytes);
  U32     (*pfGetStats)          (int Type);
  U32     (*pfGetMTU)            (void);
  void    (*pfReset)             (void);
} USB_IP_NI_DRIVER_API;
```

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.

Table 12.6: USB_IP_NI_DRIVER_API elements

Additional information

The emUSB-Device-RNDIS component calls the functions of this API to exchange data and status information with the IP stack running on the target.

(*pfInit)()

Description

Initializes the driver.

Prototype

```
void (*pfInit)(const USB_RNDIS_DRIVER_DATA * pDriverData,
               USB_IP_WRITE_PACKET *pfWritePacket);
```

Parameter	Description
pDriverData	IN: Pointer to driver configuration data. OUT: ---
pfWritePacket	Call back function called by the IP stack to transmit a packet that should be send to the USB host.

Table 12.7: (*pfInit)() parameter list

Additional information

This function is called when the RNDIS interface is added to 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* on page 374.

(*pfGetPacketBuffer)()

Description

Returns a buffer for a data packet.

Prototype

```
void * (*pfGetPacketBuffer)(unsigned NumBytes);
```

Parameter	Description
NumBytes	Size of the requested buffer in bytes.

Table 12.8: (*pfGetPacketBuffer)() parameter list

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.

(*pfWritePacket)()

Description

Delivers a data packet to target IP stack

Prototype

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

Parameter	Description
pData	IN: Data of the received packet. OUT: ---
NumBytes	Number of bytes stored in the buffer.

Table 12.9: (*pfWriteBuffer)() parameter list

Additional information

The function is called after a data packet has been received from USB. `pData` points to the buffer returned by the `(*pfGetPacketBuffer)()` function.

(*pfSetPacketFilter)()

Description

Configures the type of accepted data packets

Prototype

```
void (*pfSetPacketFilter)(U32 Mask);
```

Parameter	Description
Mask	Type of accepted data packets

Table 12.10: (*pfSetPacketFilter)() parameter list

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.

(*pfGetLinkStatus)()

Description

Returns the status of the connection to target IP stack.

Prototype

```
int (*pfGetLinkStatus)(void);
```

Return value

==USB_RNDIS_LINK_STATUS_CONNECTED:
Connected to target IP stack

==USB_RNDIS_LINK_STATUS_DISCONNECTED:
Not connected to target IP stack

(*pfGetLinkSpeed)()

Description

Returns the connection speed.

Prototype

```
U32 (*pfGetLinkSpeed)(void);
```

Return value

The connection speed in units of 100 bits/sec or 0 if not connected.

(*pfGetHWAddr)()

Description

Returns the HW address of PC.

Prototype

```
void (*pfGetHWAddr)(U8 * pAddr, unsigned NumBytes);
```

Parameter	Description
pAddr	IN: --- OUT: The HW address
NumBytes	Maximum number of bytes to store to pAddr

Table 12.11: (*pfGetHWAddr)() parameter list

Additional information

The returned HW address is the one passed to the driver in the call to (*pfInit)(). Typically the HW address is 6 bytes large.

(*pfGetStats)()

Description

Returns statistical counters.

Prototype

```
U32 (*pfGetStats)(int Type);
```

Parameter	Description
Type	The type of information requested. Can be one of these defines:

Table 12.12: (*pfGetStats)() parameter list

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.

Return value

Value of the requested statistical counter.

Additional information

The counters should be set to 0 when the `(*pfReset)()` function is called.

(*pfGetMTU)()

Description

Returns the size of the largest data packet which can be transferred.

Prototype

```
U32 (*pfGetMTU)(void);
```

Return value

The MTU size in bytes. Typically 1500 bytes.

(*pfReset)()

Description

Resets the driver.

Prototype

```
void (*pfReset)(void);
```


12.5.3.2 USB_IP_NI_DRIVER_DATA

Description

Configuration data passed to network interface driver at initialization.

Prototype

```
typedef struct USB_IP_NI_DRIVER_DATA {
  const U8 * pHWAddr;
  unsigned   NumBytesHWAddr;
} USB_IP_NI_DRIVER_DATA;
```

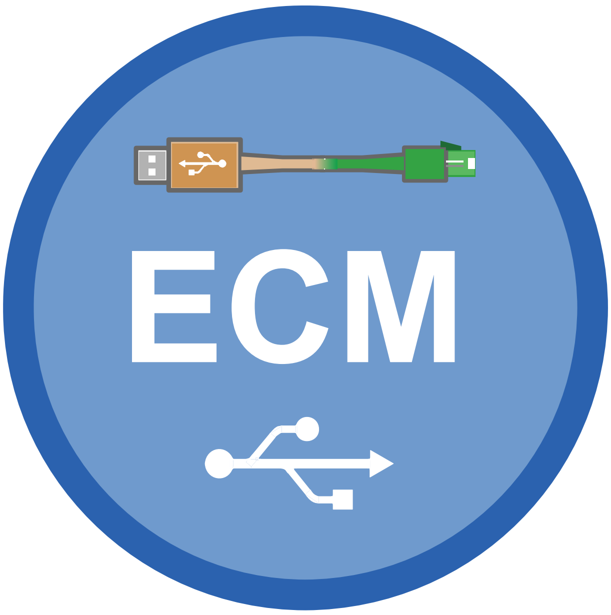
Member	Description
pHWAddr	HW address (or MAC address) of the host network interface.
NumBytesHWAddr	Number of bytes in the HW address. Typically 6 bytes.

Table 12.13: USB_IP_NI_DRIVER_DATA elements

Chapter 13

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.



13.1 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 embOS/IP 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.

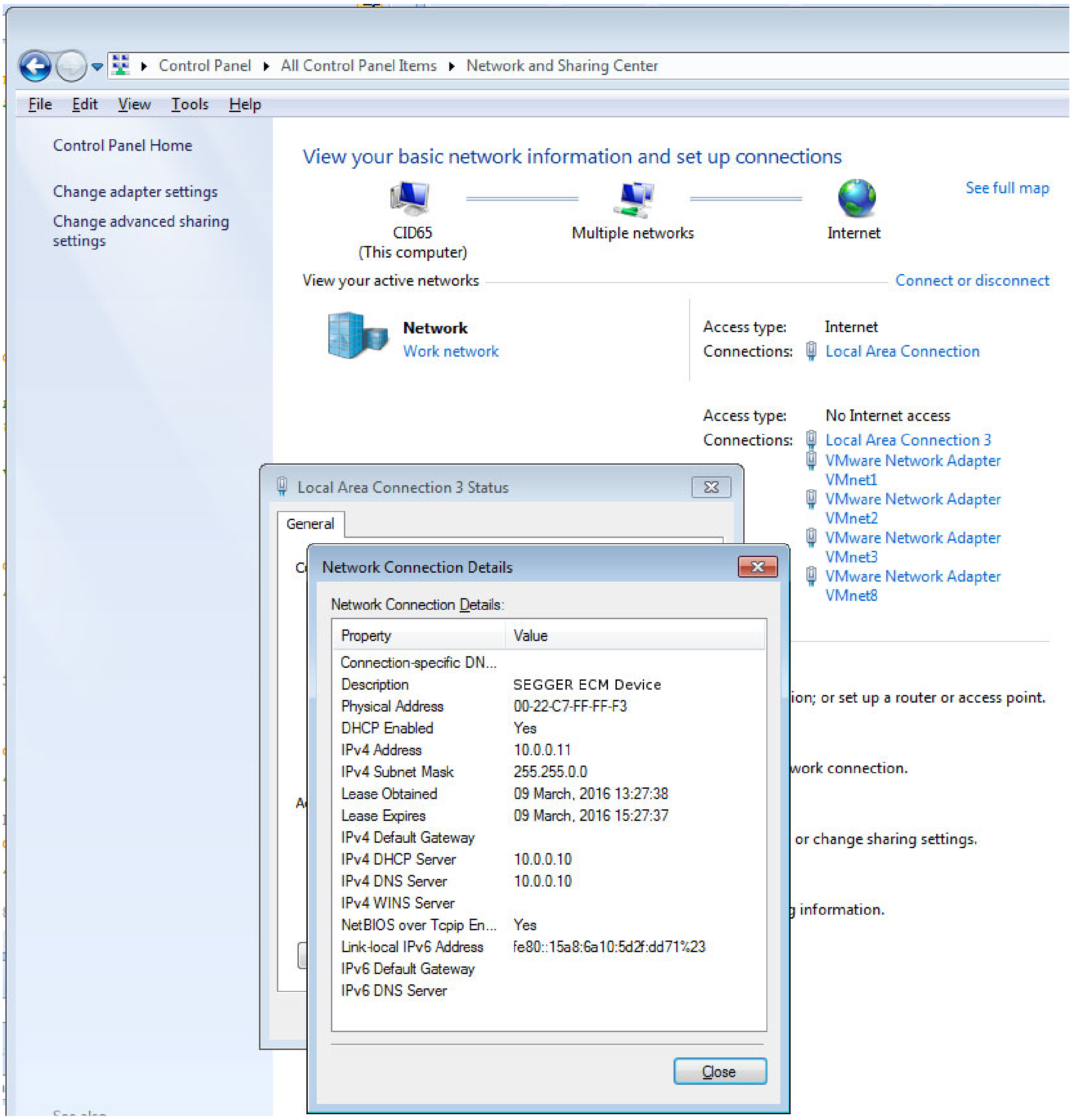
USB ECM is supported by the Linux operating system. To use it on Windows, a third party driver (not contained in emUSB-Device-ECM) has to be installed on the Windows system.

emUSB-Device-ECM contains the following components:

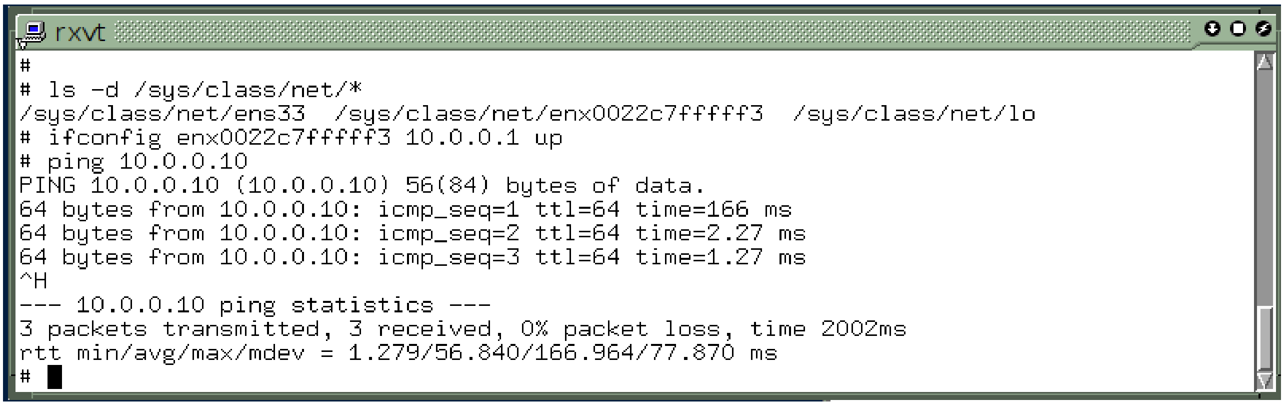
- Generic USB handling
- ECM device class implementation
- Network interface driver which uses embOS/IP as TCP/IP stack.
- A sample application demonstrating how to work with ECM.

13.1.1 Working with CDC/ECM

Any USB ECM 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. The following screenshot shows a manually configuration and ping on linux.



13.1.2 Additional information

More technical details about CDC/ECM can be found here:

http://www.usb.org/developers/docs/devclass_docs/CDC1.2_WMC1.1_052013.zip

13.2 Configuration

13.2.1 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.

13.2.2 Final configuration

The configuration must only be modified when emUSB-Device is used in your final product. Refer to section *Configuration* on page 40 to get detailed information about the general emUSB-Device configuration functions which have to be adapted.

13.3 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 embOS/IP middleware component is required. To test the emUSB-Device-ECM component any of the embOS/IP 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 choosen embOS/IP sample can be tested using the appropriate methods.

13.3.0.1 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_RNDIS.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();
  OS_CREATETASK(&_ECMTCB, "USB ECM Task", _ECMTask, TASK_PRIO_ECM_TASK, _aECMStack);
  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 end-points and configures the HW address of the PC network interface.

```
// _AddECM() - excerpt from IP_Config_ECM.c
static U8 _abReceiveBuffer[USB_MAX_PACKET_SIZE * 3];

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,  5, NULL, 0);
  InitData.pDriverAPI = &USB_Driver_IP_NI;
  InitData.DriverData.pHWAddr        = "\x00\x22\xC7\xFF\xFF\xF3";
  InitData.DriverData.NumBytesHWAddr = 6;
  USBD_ECM_Add(&InitData);
}
```

The size of _acReceiveBuffer buffer must be a multiple of USB max packet size and must be large enough to hold at least one complete ethernet packet (depending on the MTU usually >= 1500 bytes). `USB_Driver_IP_NI` is the network interface driver which implements the connection to the IP stack. The HW address configured here is assigned to the PC network interface. The HW address of the IP stack is configured in the `IP_X_Config()` function of embOS/IP as described below.

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 embOS**/**IP 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.
  //
  IP_AddEtherInterface(&USB_IP_Driver);
  IP_SetHWAddr("\x00\x22\xC7\xFF\xFF\xFF");
  IP_SetAddrMask(0x0A00000A, 0xFF000000);
  IP_SetIFaceConnectHook(IFaceId, _Connect);
  IP_SetIFaceDisconnectHook(IFaceId, _Disconnect);
  <...>
}
```


13.4 Target API

Function	Description
API functions	
USBD_ECM_Add()	Adds a RNDIS-class interface to the USB stack.
USBD_ECM_Task()	Handles the RNDIS protocol.
Data structures	
USB_ECM_INIT_DATA	Initialization data for RNDIS interface.
USB_IP_NI_DRIVER_API	Network interface driver API functions.
USB_IP_NI_DRIVER_DATA	Configuration data for the network interface driver.

Table 13.1: List of emUSB-Device ECM module functions and data structures

13.4.1 API functions

13.4.1.1 USBD_ECM_Add()

Description

Adds an ECM-class interface to the USB stack.

Prototype

```
void USBD_ECM_Add(const USB_ECM_INIT_DATA * pInitData);
```

Parameter	Description
pInitData	IN: Pointer to a USB_ECM_INIT_DATA structure. OUT: ---

Table 13.2: USBD_ECM_Add() parameter list

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_ECM_INIT_DATA* on page 384 for more information.

13.4.1.2 USBD_ECM_Task()

Description

Handles the ECM protocol.

Prototype

```
void USBD_ECM_Task(void);
```

Additional information

The function should be called periodically after the USB device has been successfully enumerated and configured. The function returns when the USB device is detached or suspended. For a sample usage refer to *IP_Config_ECM.c in detail* on page 379.

13.4.2 Data structures

13.4.2.1 USB_ECM_INIT_DATA

Description

Initialization data for ECM interface.

Prototype

```
typedef struct USB_ECM_INIT_DATA {
  U8 EPIn;
  U8 EPOut;
  U8 EPInt;
  const USB_IP_NI_DRIVER_API * pDriverAPI;
  USB_IP_NI_DRIVER_DATA        DriverData;
} USB_ECM_INIT_DATA;
```

Member	Description
EPIn	Endpoint for sending data to the host.
EPOut	Endpoint for receiving data from the host.
EPInt	Endpoint for sending status information.
pDriverAPI	Pointer to the Network interface driver API. (See *Driver interface* on page 385)
DriverData	Configuration data for the network interface driver. (See *Driver interface* on page 385)

Table 13.3: USB_ECM_INIT_DATA elements

Additional information

This structure holds the endpoints that should be used by the RNDIS interface (EPin, EPOut and EPInt). Refer to *USBD_AddEP()* on page 55 for more information about how to add an endpoint.

13.4.3 Driver interface

The ECM interface uses the same API to connect to the IP stack as the RNDIS class. Refer to section 12.5.3

Chapter 14

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.

14.1 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

14.2 Configuration

14.2.1 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.

14.2.2 Final configuration

The configuration must only be modified when emUSB-Device is integrated in your final product. Refer to section *Configuration* on page 40 for detailed information about the generic information functions which have to be adapted.

14.3 Running the sample application

The directory `Application` contains a sample application which can be used with emUSB-Device and the UVC component. To test the emUSB-Device-UVC component, the application should be built and then downloaded to target. Remove the USB connection and reconnect the target to the host.

14.3.1 USB_UVC_Start.c in detail

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

```
/*********************************************************************
*
*       _AddUVC
*
*  Function description
*    Add UVC mouse class to USB stack
*/
static USBD_UVC_HANDLE _AddUVC(void) {
  USBD_UVC_INIT_DATA InitData;

  memset(&InitData, 0, sizeof(InitData));
  InitData.EPIn     = USB_AddEP(USB_DIR_IN,  USB_TRANSFER_TYPE_ISO, 1, NULL, 0);
  InitData.pBuf     = &_Buf;
  return USBD_UVC_Add(&InitData);
}

<...>

/*********************************************************************
*
*       MainTask
*
* USB handling task.
*   Modify to implement the desired protocol
*/
void MainTask(void) {
  USBD_UVC_HANDLE h;

  USB_Init();
  h = _AddUVC();
  USB_Start();
  while (1) {

    //
    // Wait for configuration
    //
    while ((USB_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED)) !=
USB_STAT_CONFIGURED) {
      BSP_ToggleLED(0);
      USB_OS_Delay(50);
    }

    USBD_UVC_Write(h, test1, sizeof(test1), USBD_UVC_END_OF_FRAME);
    USBD_UVC_Write(h, test2, sizeof(test2), USBD_UVC_END_OF_FRAME);
    USBD_UVC_Write(h, test3, sizeof(test3), USBD_UVC_END_OF_FRAME);
    USBD_UVC_Write(h, test4, sizeof(test4), USBD_UVC_END_OF_FRAME);
  }
}
```


14.4 Target API

Function	Description
API functions	
USBD_UVC_Add()	Adds an UVC interface to the USB stack.
USBD_UVC_Write()	Writes frame data to the host.
Data structures	
USBD_UVC_INIT_DATA	Contains variables required for the UVC module intialisation.
USBD_UVC_BUFFER	Structure which describes the UVC ring buffer.
USBD_UVC_DATA_BUFFER	Structure which describes the UVC data buffer.

Table 14.1: List of emUSB-Device UVC interface functions and data structures

14.4.1 API functions

14.4.1.1 USBD_UVC_Add()

Description

Adds an UVC interface to the USB stack.

Prototype

```
void USBD_UVC_Add(const USBD_UVC_INIT_DATA * pInitData);
```

Parameter	Description
pInitData	Pointer to a USBD_UVC_INIT_DATA structure.

Table 14.2: USBD_UVC_Add() parameter list

Additional information

After the initialization of USB core, this is the first function that needs to be called when an UVC interface is used with emUSB-Device. The structure USBD_UVC_INIT_DATA has to be initialized before USBD_UVC_Add() is called. Refer to *USBD_UVC_INIT_DATA* on page 393 for more information.

14.4.1.2 USBD_UVC_Write()

Description

Write frame data to the host. This function is blocking.

Prototype

```
int USBD_UVC_Write (USBD_UVC_HANDLE hInst,
                    const void* pData,
                    unsigned NumBytes,
                    U8 Flags);
```

Parameter	Description
hInst	Handle to a UVC instance that was returned by USBD_UVC_Add().
pData	Pointer to a buffer containing the frame data.
NumBytes	Number of bytes in the buffer.
Flags	Flags to be added to the frame: USBD_UVC_END_OF_FRAME - Should be set with the last USBD_UVC_Write call for a single frame.

Table 14.3: USBD_UVC_Write() parameter list

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 USBD_UVC_END_OF_FRAME when the last data part was written via USBD_UVC_Write().

Internally this function will write data into the buffers which have been initialised by the call to USBD_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.

Example

Sample describing a write operation where a frame is entirely available in a single buffer:

```
USBD_UVC_Write(h, WholeFrame, sizeof(WholeFrame), USBD_UVC_END_OF_FRAME);
```

Sample describing a write operation where a frame is only available in chuncks:

```
U32 NumBytesAtOnce;
U32 NumBytesTotal;
U8  Flags;


NumBytesTotal = 153600; // Fixed frame size.
NumBytesAtOnce = SEGGER_MIN(sizeof(SmallBuffer), NumBytesTotal);
Flags = 0;
while (NumBytesTotal) {
  USBD_UVC_Write(h, SmallBuffer, NumBytesAtOnce, Flags);
  NumBytesTotal -= NumBytesAtOnce;
  NumBytesAtOnce = SEGGER_MIN(sizeof(SmallBuffer), NumBytesTotal);
  if (NumBytesTotal <= sizeof(SmallBuffer)) {
    Flags = USBD_UVC_END_OF_FRAME; // This will be the last write for this frame.
  }
}
```


14.4.2 Data structures

14.4.2.1 USBD_UVC_INIT_DATA

Description

Structure which stores the parameters of the UVC interface.

Prototype

```
typedef struct {
  U8                EPIn;
  USBD_UVC_BUFFER * pBuf;
} USBD_UVC_INIT_DATA;
```

Member	Description
EPIn	Endpoint for sending data to the host.
pBuf	Pointer to a USBD_UVC_BUFFER structure.

Table 14.4: USBD_UVC_INIT_DATA elements

Additional Information

This structure holds the endpoint that should be used with the UVC interface. Refer to *USBD_AddEP()* on page 55 for more information about how to add an endpoint.

14.4.2.2 USBD_UVC_BUFFER

Description

Structure which contains information about the UVC ring buffer.

Prototype

```
typedef struct _USBD_UVC_BUFFER {
  USBD_UVC_DATA_BUFFER  Buf[USBD_UVC_NUM_BUFFERS];
  unsigned              NumBlocksIn;
  unsigned              RdPos;
  unsigned              WrPos;
} USBD_UVC_BUFFER;
```

Member	Description
Buf	Array of USBD_UVC_DATA_BUFFER elements.
NumBlocksIn	Number of currently used buffers.
RdPos	Buffer read position.
WrPos	Buffer write position.

Table 14.5: USBD_UVC_BUFFER elements

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.

14.4.2.3 USBD_UVC_DATA_BUFFER

Description

Structure which contains values for a single buffer as well as the data buffer itself.

Prototype

```
typedef struct _USBD_UVC_DATA_BUFFER{
  U8   Data[USBD_UVC_DATA_BUFFER_SIZE];
  U8   Flags;
  U16  NumBytesIn;
  U8   FrameID;
} USBD_UVC_DATA_BUFFER;
```

Member	Description
Data	Data buffer for a single packet.
Flags	Flags which will be sent with the packet.
NumBytesIn	Size of the packet
FrameID	ID of the frame.

Table 14.6: USBD_UVC_DATA_BUFFER

Additional Information

The size of the buffers can be set with the USBD_UVC_DATA_BUFFER_SIZE define. Ideally it should match the MaxPacketSize for the isochronous endpoint, which will be 1024 in USB high-speed and 1023 in USB full-speed mode.
Generally the user does not have to interact with this structure, but he has to provide the memory for it.

Chapter 15

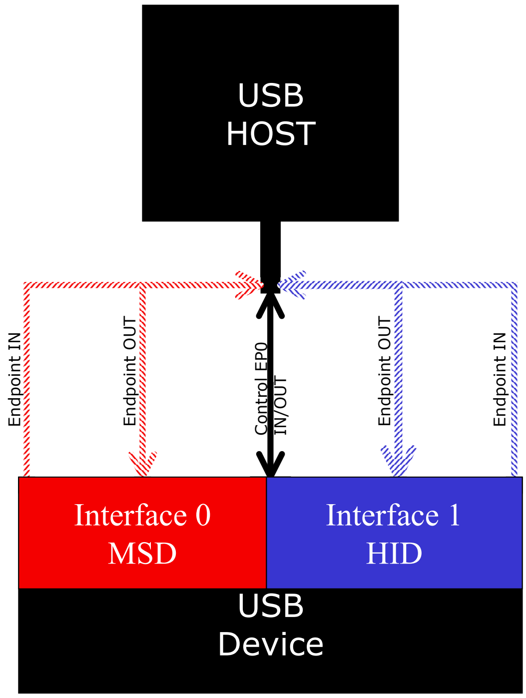
Combining USB components (Multi-Interface)

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.

15.1 Overview

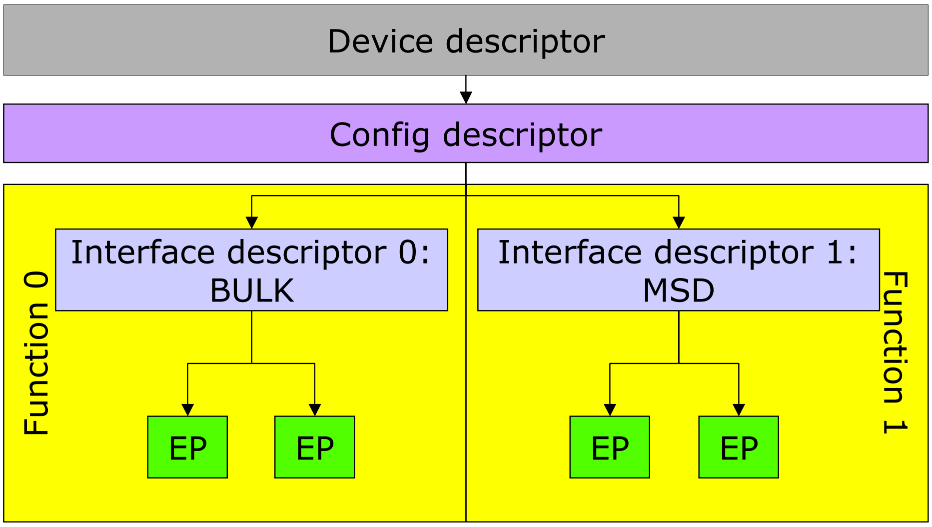
The USB specification allows implementation of more than one component (function) in a single device. This is achieved by combining two or more components. These devices will be recognized by the USB host as composite device and each component will be recognized as an independent device.

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.



15.1.1 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.

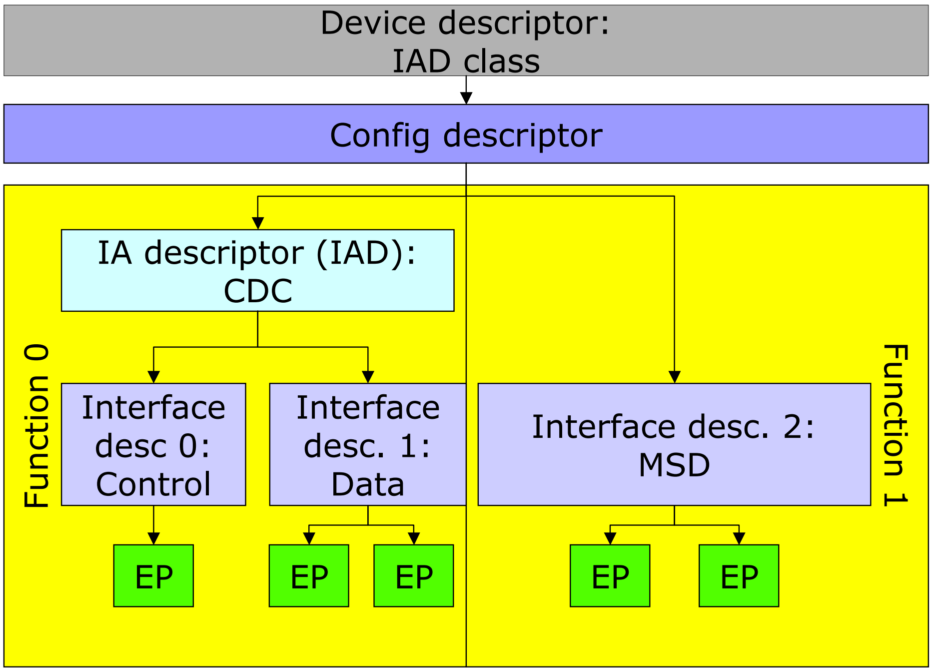


15.1.2 Multiple interface device classes

In contrast to the single interfaces classes there are classes with multiple interfaces such as CDC and AUDIO or VIDEO class. These classes define their class identifier in the device descriptor. All interface descriptors are recognized as part of the component that is defined in the device descriptor. This prevents the combination of multiple interface device classes (for example, CDC) with any other component.

15.1.3 IAD class

To remove this limitation the USB organization 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 HOST

At the time of writing, IAD is supported by:

- Windows XP with Service pack 2 and newer
- Linux Kernel 2.6.22 and higher

15.2 Configuration

In general, no configuration is required. By default, emUSB-Device supports up to four interfaces. If more interfaces are needed the following macro 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.

15.3 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 405 and check the following sample.

Requirements

- RTOS, every component requires a separate task.
- 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.

Sample application

The following sample application uses embOS as the RTOS. This listing is taken from `USB_CompositeDevice_CDC_MSD.c`.

```
/*********************************************************************
*                 SEGGER MICROCONTROLLER GmbH & Co. KG               *
*         Solutions for real time microcontroller applications       *
**********************************************************************
*                                                                    *
*         (c) 2003-2011     SEGGER Microcontroller GmbH & Co KG      *
*                                                                    *
*         Internet: www.segger.com    Support:  support@segger.com   *
*                                                                    *
**********************************************************************
*                                                                    *
*        USB device stack for embedded applications                  *
*                                                                    *
**********************************************************************
----------------------------------------------------------------------
File    : USB_CompositeDevice_CDC_MSD.c
Purpose : Sample showing a USB device with multiple interfaces (CDC+MSD).
--------  END-OF-HEADER  ---------------------------------------------
*/

#include <stdio.h>
#include <stdio.h>
#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_MAX_PACKET_SIZE];
  USB_MSD_INIT_DATA     InitData;
  USB_MSD_INST_DATA     InstData;

  InitData.EPIn  = USBD_AddEP(1, USB_TRANSFER_TYPE_BULK, USB_MAX_PACKET_SIZE, NULL,
0);
  InitData.EPOut = USBD_AddEP(0, USB_TRANSFER_TYPE_BULK, USB_MAX_PACKET_SIZE,
_abOutBuffer, USB_MAX_PACKET_SIZE);
  USB_MSD_Add(&InitData);
  //
  // Add logical unit 0: RAM drive, using SDRAM
  //
  memset(&InstData, 0,  sizeof(InstData));
  InstData.pAPI                     = &USB_MSD_StorageByName;
  InstData.DriverData.pStart        = (void *)"";
  InstData.pLunInfo = &_Lun0Info;
  USB_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);
    }
    USB_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
  printf("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 void _AddCDC(void) {
  static U8 _abOutBuffer[USB_MAX_PACKET_SIZE];
  USB_CDC_INIT_DATA     InitData;

  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_MAX_PACKET_SIZE);
  InitData.EPInt = USBD_AddEP(USB_DIR_IN,  USB_TRANSFER_TYPE_INT,   8,  NULL, 0);
  USBD_CDC_Add(&InitData);
  USBD_CDC_SetOnLineCoding(_OnLineCoding);
}


/*********************************************************************
*
*       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) {

  USBD_Init();
  USBD_EnableIAD();
  _AddCDC();
  _AddMSD();
  USBD_SetDeviceInfo(&_DeviceInfo);
  USBD_Start();
  BSP_SetLED(0);
  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(&ac[0], sizeof(ac));
    if (NumBytesReceived > 0) {
      USBD_CDC_Write(&ac[0], NumBytesReceived);
    }
  }
}

/**************************** end of file ***************************/
```

15.4 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.

15.4.1 BULK communication component

15.4.1.1 Device side

No modification on device side needs to be made.

15.4.1.2 Host side

Windows will recognize the device as a composite device. It will load the drivers for each interface.

In order to recognize the bulk interface in the composite device, the `.inf` file of the device needs to be modified.
Windows will extend the device identification string with the interface number. This has to be added to the device identification string in the `.inf` file.
The provided `.inf` file:

```
;
; Generic USBBulk driver setup information file
; Copyright (c) 2006-2008 by SEGGER Microcontroller GmbH & Co. KG
;
; This file supports:
;     Windows 2000
;     Windows XP
;     Windows Server 2003 x86
;     Windows Vista x86
;     Windows Server 2008 x86
;
[Version]
Signature="$Windows NT$"
Provider=%MfgName%
Class=USB
ClassGUID={36FC9E60-C465-11CF-8056-444553540000}
DriverVer=03/19/2008,2.6.6.0
CatalogFile=USBBulk.cat

[Manufacturer]
%MfgName%=DeviceList

[DeviceList]
%USB\VID_8765&PID_1234.DeviceDesc%=USBBulkInstall, USB\VID_8765&PID_1234&Mi_xx

[USBBulkInstall.ntx86]
CopyFiles=USBBulkCopyFiles

[USBBulkInstall.ntx86.Services]
Addservice = usbbulk, 0x00000002, USBBulkAddService, USBBulkEventLog

[USBBulkAddService]
DisplayName    = %USBBulk.SvcDesc%
ServiceType    = 1                  ; SERVICE_KERNEL_DRIVER
StartType      = 3                  ; SERVICE_DEMAND_START
ErrorControl   = 1                  ; SERVICE_ERROR_NORMAL
ServiceBinary  = %10%\System32\Drivers\USBBulk.sys

[USBBulkEventLog]
AddReg=USBBulkEventLogAddReg

[USBBulkEventLogAddReg]
HKR,,EventMessageFile,%REG_EXPAND_SZ%,"%%SystemRoot%%\System32\IoLogMsg.dll;%%System
Root%%\System32\drivers\USBBulk.sys"
HKR,,TypesSupported,  %REG_DWORD%,7


[USBBulkCopyFiles]
USBBulk.sys

[DestinationDirs]
DefaultDestDir = 10,System32\Drivers
USBBulkCopyFiles = 10,System32\Drivers
```



```
[SourceDisksNames.x86]
1=%USBBulk.DiskName%,,

[SourceDisksFiles.x86]
USBBulk.sys = 1


;-----------------------------------------------------------------;

[Strings]
MfgName="Segger"
USB\VID_8765&PID_1234.DeviceDesc="USB Bulk driver"
USBBulk.SvcDesc="USBBulk driver"
USBBulk.DiskName="USBBulk Installation Disk"

; Non-Localizable Strings, DO NOT MODIFY!
REG_SZ          = 0x00000000
REG_MULTI_SZ    = 0x00010000
REG_EXPAND_SZ   = 0x00020000
REG_BINARY      = 0x00000001
REG_DWORD       = 0x00010001


; *** EOF ***
```

Please add the red colored text to your .inf file and change xx with the interface number of the bulk component.
The interface number is a zero-based index and is assigned by the emUSB-Device stack when calling the USBD_BULK_Add() function. If you have called USBD_BULK_Add() prior to any other `USB_xxx_Add()` functions then the interface number will be `00`.
Please note that when USBD_CDC_Add() is called prior USBD_BULK_Add(), the interface number for the BULK component will be `02` since the CDC component uses two interfaces (in the example, `00` and `01`).

15.4.2 MSD component

15.4.2.1 Device side

No modification on device side needs to be made.

15.4.2.2 Host side

No modification on host side needs to be made.

15.4.3 CDC component

15.4.3.1 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 402 and check the listing of the sample application.

15.4.3.2 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 the `.inf` file needs to be modified.
As in the Bulk communication component, Windows will extend the device identification strings. Therefore the device identification string must be modified.
The provided `.inf` file:

```
;
;   Device installation file for
;   USB 2 COM port emulation
;
;
;
[Version]
Signature="$Windows NT$"
Class=Ports
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}
Provider=%MFGNAME%
LayoutFile=layout.inf
DriverVer=03/26/2007,6.0.2600.1
CatalogFile=usbser.cat

[Manufacturer]
%MFGNAME%=CDCDevice,NT,NTamd64

[DestinationDirs]
DefaultDestDir = 12

[CDCDevice.NT]
%DESCRIPTION%=DriverInstall,USB\VID_8765&PID_1111&Mi_xx

[CDCDevice.NTamd64]
%DESCRIPTION%=DriverInstall,USB\VID_8765&PID_0234&Mi_xx
%DESCRIPTION%=DriverInstall,USB\VID_8765&PID_1111&Mi_xx

[DriverInstall.NT]
```



```
Include=mdmcpq.inf
CopyFiles=FakeModemCopyFileSection
AddReg=DriverInstall.NT.AddReg

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

[DriverInstall.NT.Services]
AddService=usbser, 0x00000002, DriverServiceInst

[DriverServiceInst]
DisplayName=%SERVICE%
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%12%\usbser.sys

[Strings]
MFGNAME = "Manufacturer"
DESCRIPTION = "USB CDC serial port emulation"
SERVICE = "USB CDC serial port emulation"
```

Please add the red colored text to your `.inf` file and change `xx` with the interface number of the CDC component.
The interface number is a zero based index and is assigned by the emUSB-Device stack when calling USBD_CDC_Add() function.

15.4.4 HID component

15.4.4.1 Device side

No modification on device side needs to be made.

15.4.4.2 Host side

No modification on host device side needs to be made.

Chapter 16

Target OS Interface

This chapter describes the functions of the operating system abstraction layer.

16.1 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* on page 413 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.

16.1.1 Operating system support supplied with this release

In the current version, abstraction layers for embOS and µC/OS-II are available.

A kernel abstraction layer for using emUSB-Device without any RTOS (superloop) is also supplied.

Abstraction layers for other operating systems are available upon request.

16.2 Interface function list

Routine	Explanation
USB_OS_Delay()	Delays for a given number of ms.
USB_OS_DecRI()	Decrements the interrupt disable count and enables interrupts if the counter reaches 0.
USB_OS_GetTickCnt()	Returns the current system time in ticks.
USB_OS_IncDI()	Increments the interrupt disable count and disables interrupts.
USB_OS_Init()	Initializes the OS.
USB_OS_Panic()	Called if fatal error is detected.
USB_OS_Signal()	Wakes the task waiting for signal.
USB_OS_Wait()	Blocks the task until USB_OS_Signal() is called.
USB_OS_WaitTimed()	Blocks the task until USB_OS_Signal() is called or a timeout occurs.

Table 16.1: Target OS interface function list

16.2.1 USB_OS_Delay()

Description

Delays for a given number of ms.

Prototype

```
void USB_OS_Delay(int ms);
```

Parameter	Description
ms	Number of ms.

Table 16.2: USB_OS_Delay() parameter list

16.2.2 USB_OS_DecRI()

Description

Decrements interrupt disable count and enable interrupts if counter reaches 0.

Prototype

```
void USB_OS_DecRI(void);
```


16.2.3 USB_OS_GetTickCnt()

Description

Returns the current system time in ticks.

Prototype

```
U32 USB_OS_GetTickCnt(void);
```


16.2.4 USB_OS_IncDI()

Description

Increments interrupt disable count and disables interrupts.

Prototype

```
void USB_OS_IncDI(void);
```


16.2.5 USB_OS_Init()

Description

Initializes OS.

Prototype

```
void USB_OS_Init(void);
```

16.2.6 USB_OS_Panic()

Description

Halts emUSB-Device.

Prototype

```
void USB_OS_Panic(const char *pErrMsg);
```

Parameter	Description
pErrMsg	Pointer to error message string.

Table 16.3: USB_OS_Panic() parameter list

16.2.7 USB_OS_Signal()

Description

Wakes the task waiting for signal.

Prototype

```
void USB_OS_Signal(unsigned EPIndex);
```

Parameter	Description
EPIndex	Endpoint index.

Table 16.4: USB_OS_Signal() parameter list

Additional information

This routine is typically called from within an interrupt service routine.

16.2.8 USB_OS_Wait()

Description

Blocks the task until `USB_OS_Signal()` is called.

Prototype

```
void USB_OS_Wait(unsigned EPIndex);
```

Parameter	Description
EPIndex	Endpoint index.

Table 16.5: USB_OS_Wait() parameter list

Additional information

This routine is called from a task.

16.2.9 USB_OS_WaitTimed()

Description

Blocks the task until `USB_OS_Signal()` is called or a timeout occurs.

Prototype

```
int USB_OS_WaitTimed(unsigned EPIndex, unsigned ms);
```

Parameter	Description
EPIndex	Endpoint index.
ms	Timeout time given in ms.

Table 16.6: USB_OS_WaitTimed() parameter list

Return value

== 0: Task was signaled within the given timeout.
== 1: Timeout occurred.

Additional information

USB_OS_WaitTimed is called from a task. This function is used by all available timed routines.

16.3 Example

A configuration to use USB with embOS might look like the sample below. This example is also supplied in the subdirectory `OS\embOS\`.

```
/*********************************************************************
*                   SEGGER MICROCONTROLLER GmbH & Co. KG             *
*           Solutions for real time microcontroller applications     *
**********************************************************************
*                                                                    *
*           (c) 2003-2010     SEGGER Microcontroller GmbH & Co KG    *
*                                                                    *
*           Internet: www.segger.com    Support:  support@segger.com *
*                                                                    *
**********************************************************************
*                                                                    *
*         USB device stack for embedded applications                 *
*                                                                    *
**********************************************************************
----------------------------------------------------------------------
File    : USB_OS_embOS.c
Purpose : Kernel abstraction for embOS
          Do not modify to allow easy updates !
--------  END-OF-HEADER  ---------------------------------------------
*/


#include "USB_Private.h"
#include "RTOS.h"

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static OS_EVENT _aEvent[USB_NUM_EPS + USB_EXTRA_EVENTS];

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/
/*********************************************************************
*
*       USB_OS_Init
*
*  Function description:
*    This function shall initialize all event objects that are necessary.
*
*/
void USB_OS_Init(void) {
  unsigned i;

  for (i = 0; i < COUNTOF(_aEvent); i++) {
    OS_EVENT_Create(&_aEvent[i]);
  }
}

/***********************************************************
*
*       USB_OS_Signal
*
* Function description
*   Wake the task waiting for reception
*   This routine is typically called from within an interrupt
*   service routine
*
*/
void USB_OS_Signal(unsigned EPIndex) {
  OS_EVENT_Pulse(&_aEvent[EPIndex]);
}

/***********************************************************
*
*        USB_OS_Wait
*
* Function description
```



```
*   Block the task until USB_OS_SignalRx is called
*   This routine is called from a task.
*
*/
void USB_OS_Wait(unsigned EPIndex) {
  OS_EVENT_Wait(&_aEvent[EPIndex]);
}

/*********************************************************************
*
*        USB_OS_WaitTimed
*
* Function description
*   Block the task until USB_OS_Signal is called
*   or a time out occurs
*   This routine is called from a task.
*
*/
int USB_OS_WaitTimed(unsigned EPIndex, unsigned ms) {
  int r;
  r = (int)OS_EVENT_WaitTimed(&_aEvent[EPIndex], ms + 1);
  return r;
}

/*********************************************************************
*
*       USB_OS_Delay
*
* Function description
*   Delays for a given number of ms.
*
*/
void USB_OS_Delay(int ms) {
  OS_Delay(ms);
}

/*********************************************************************
*
*       USB_OS_DecRI
*
* Function description
*   Decrement interrupt disable count and enable interrupts
*   if counter reaches 0.
*
*/
void USB_OS_DecRI(void) {
  OS_DecRI();
}

/*********************************************************************
*
*        USB_OS_IncDI
*
* Function description
*   Increment interrupt disable count and disable interrupts
*
*/
void   USB_OS_IncDI(void) {
  OS_IncDI();
}

/*********************************************************************
*
*        USB_OS_Panic
*
* Function description
```

```
*   Called if fatal error is detected.
*/
void USB_OS_Panic(const char *pErrMsg) {
  while (pErrMsg);
}

/*********************************************************************
*
*        USB_OS_GetTickCnt
*
* Function description
*   Returns the current system time in ticks.
*/
U32 USB_OS_GetTickCnt(void) {
  return OS_Time;
}

/*************************** End of file ****************************/
```


Chapter 17

Target USB Driver

This chapter describes emUSB-Device hardware interface functions in detail.

17.1 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 controllers and USB 2.0 controllers, both as external chips and as part of microcontrollers.

17.1.1 Available USB drivers

An always up to date list can be found at:
http://www.segger.com/pricelist-emusb.html

The following device drivers are available for emUSB-Device:

Driver (Device)	Identifier
ATMEL AV32 UC3x	USB_Driver_Atmel_AT32UC3x
ATMEL AT91CAP9x	USB_Driver_Atmel_CAP9
ATMEL AT91SAM3S/AT91SAM4S	USB_Driver_Atmel_SAM3S
ATMEL AT91SAM3Uxx	USB_Driver_Atmel_SAM3US
ATMEL AT91SAM3x8	USB_Driver_Atmel_AT91SAM3X
ATMEL AT91RM9200	USB_Driver_Atmel_RM9200
ATMEL AT91SAM7A3	USB_Driver_Atmel_SAM7A3
ATMEL AT91SAM7S64	USB_Driver_Atmel_SAM7S
ATMEL AT91SAM7S128	USB_Driver_Atmel_SAM7S
ATMEL AT91SAM7S256	USB_Driver_Atmel_SAM7S
ATMEL AT91SAM7SE	USB_Driver_Atmel_SAM7SE
ATMEL AT91SAM7X128	USB_Driver_Atmel_SAM7X
ATMEL AT91SAM7X256	USB_Driver_Atmel_SAM7X
ATMEL AT91SAM9260	USB_Driver_Atmel_SAM9260
ATMEL AT91SAM9261	USB_Driver_Atmel_SAM9261
ATMEL AT91SAM9263	USB_Driver_Atmel_SAM9263
ATMEL AT91SAM9R64 ATMEL AT91SAM9RL64	USB_Driver_Atmel_SAMRx64
ATMEL AT91SAM9G20	USB_Driver_Atmel_SAM9G20
ATMEL AT91SAM9G45	USB_Driver_Atmel_SAM9G45
ATMEL AT91SAM9XE	USB_Driver_Atmel_SAM9XE
EnergyMicro EFM32GG	USB_Driver_EM_EFM32GG990
Freescale iMX25x	USB_Driver_Freescale_iMX25x
Freescale iMX28x	USB_Driver_Freescale_iMX28x
Freescale Kinetis K40	USB_Driver_Freescale_K40
Freescale Kinetis K60	USB_Driver_Freescale_K60
Freescale MCF227x	USB_Driver_Freescale_MCF227x
Freescale MCF225x	USB_Driver_Freescale_MCF225x
Freescale MCF51JMx	USB_Driver_Freescale_MCF51JMx
Freescale Vybrid	USB_Driver_Freescale_Vybrid
Fujitsu MB9BF50x	USB_Driver_Fujitsu_MB9BF50x
NXP LPC13xx	USB_Driver_NXP_LPC13xx

Table 17.1: List of included USB device drivers

Driver (Device)	Identifier
NXP LPC17xx	USB_Driver_NXP_LPC17xx
NXP LPC18xx	USB_Driver_NXP_LPC18xx
NXP LPC214x	USB_Driver_NXP_LPC214x
NXP LPC23xx	USB_Driver_NXP_LPC23xx
NXP LPC24xx	USB_Driver_NXP_LPC24xx
NXP LPC313x	USB_Driver_NXP_LPC313x
NXP LPC318x	USB_Driver_NXP_LPC318x
NXP LPC43xx	USB_Driver_NXP_LPC43xx
NXP (formerly Sharp) LH79524/5	USB_Driver_Sharp_LH79524
NXP (formerly Sharp) LH7A40x	USB_Driver_Sharp_LH7A40x
OKI 69Q62	USB_Driver_OKI_69Q62
Renesas H8S2472	USB_Driver_Renesas_H8S2472
Renesas H8SX1668R	USB_Driver_Renesas_H8SX1668R
Renesas RX62N	USB_Driver_Renesas_RX62N
Renesas RX63N/RX631	USB_Driver_Renesas_RX62N
Renesas SH7203	USB_Driver_Renesas_SH7203
Renesas SH7216	USB_Driver_Renesas_SH7216
Renesas SH7286	USB_Driver_Renesas_SH7286
Renesas (NEC) 78K0R-KE3L	USB_Driver_NEC_78F102x
Renesas (NEC) µPD720150	USB_Driver_NEC_uPD720150
Renesas (NEC) V850ESJG3H	USB_Driver_NEC_70F376x
ST STM32	USB_Driver_ST_STM32
ST STM32F105/107	USB_Driver_ST_STM32F107
ST STR71x	USB_Driver_ST_STR71x
ST STR750	USB_Driver_ST_STR750
ST STR912	USB_Driver_ST_STR91x
Toshiba TMPA900	USB_Driver_Toshiba_TMPA900
Toshiba TMPA910	USB_Driver_Toshiba_TMPA910
Texas Intruments MSP430X5529	USB_Driver_TI_MSP430
Texas Intruments (Luminary) LM3S9B9x	USB_Driver_TI_LM3S9B9x

Table 17.1: List of included USB device drivers

17.2 Adding a driver to emUSB-Device

USBD_Init() initializes the internals of the USB stack and is always the first function which the USB application has to call. USBD_Init() will then call USBD_X_Config(). This function should be used to perform the following tasks:

- Perform device specific hardware initialisation if neccessary.
- Add a USB driver to your project.
- 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, USBD_AddDriver() should be called in USBD_X_Config() with the identifier of the driver which is compatible to your hardware as parameter. Refer to the section *Available USB drivers* on page 428 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 USBD_SetAttachFunc() within USBD_X_Config().

Additionally three interrupt management fucntions must be installed unsing the function USBD_SetISRMgmFunc().

Modify USBD_X_Config(), _EnableISR() and if required, _HWAttach().

17.2.1 USBD_X_Config()

Description

Configure the USB stack.

Prototype

`void USBD_X_Config(void)`

Additional information

This function is always called from `USBD_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_AtmelSAM7A3);
  USBD_SetAttachFunc(_HWAttach);
  USBD_SetISRMgmFuncs(_EnableISR, USB_OS_IncDI, USB_OS_DecRI);
}
```


17.3 Interrupt handling

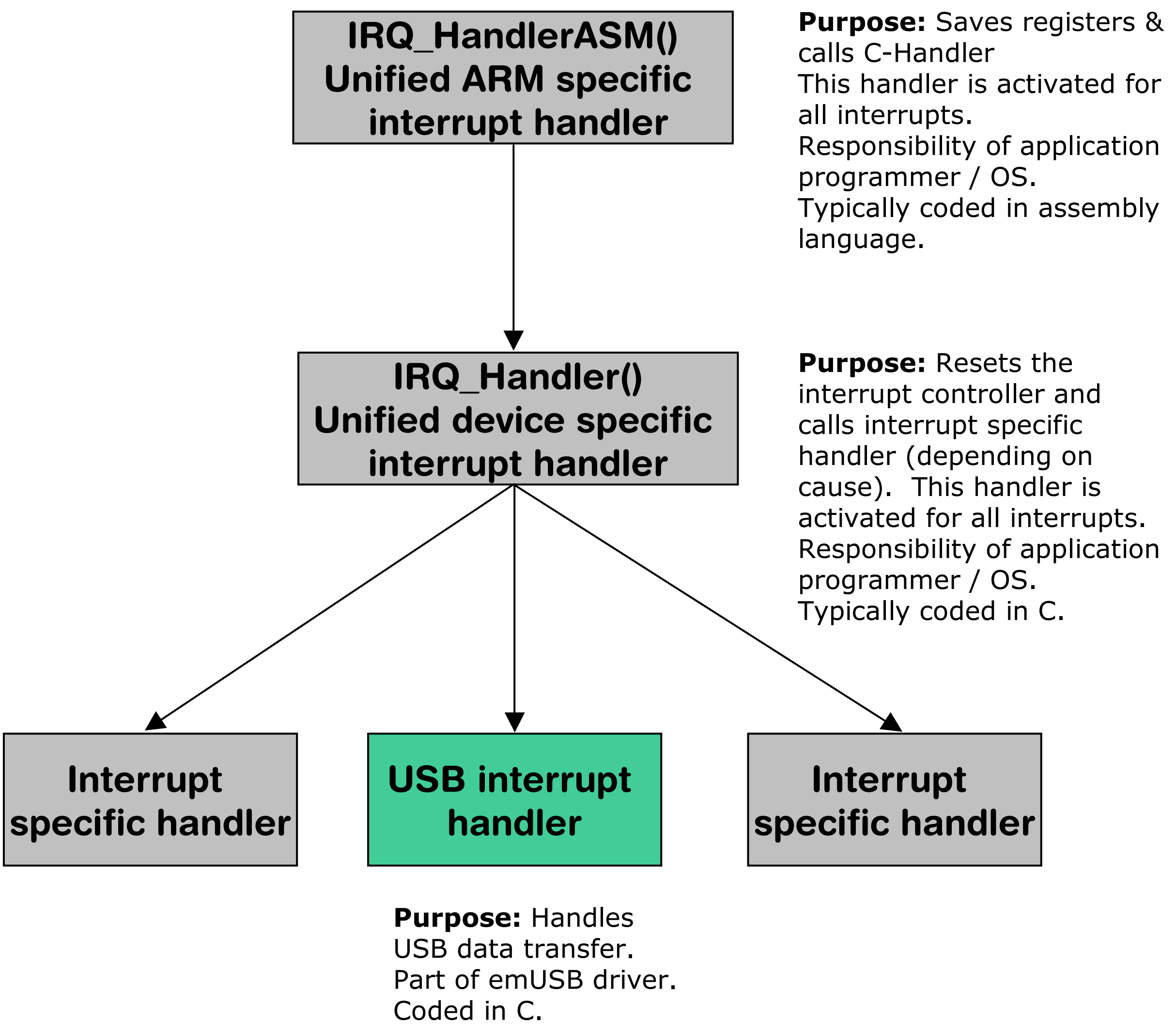
emUSB-Device is interrupt driven and optimized to be used with a real-time operating system. If you use embOS in combination with emUSB-Device, you can skip the following sections.

If you are not using embOS, you have to be familiar with how interrupts are handled on your target system. This includes knowledge about how the CPU handles interrupts, how and which registers are saved, the interrupt vector table, how the interrupt controller works and how it is reset.

17.3.1 ARM7 / ARM9 based cores

ARM7 and ARM9 cores will jump to IRQ vector address `0x18`, where a jump to an ARM specific IRQ handler should be located. This ARM specific IRQ handler calls a device specific interrupt handler which handles the interrupt controller.

The ARM specific interrupt handler is typically coded in assembly language. It has to ensure that no context information will be lost if an interrupt occurs. The environment of the interrupted function has to be restored after processing the interrupt. The environment of the interrupted function includes the value of the processor registers and the processor status register. The ARM specific interrupt handler calls a high-level interrupt handler which manages the call of the interrupt source specific service routine.



17.3.1.1 ARM specific IRQ handler

The ARM specific interrupt handler saves the context of the function which is interrupted, calls the high-level interrupt handler and restores the context. Sample implementations of the high-level handler are supplied in the following device specific sections.

Sample implementation interrupt handler

```
EXTERN  IRQ_Handler

IRQ_HandlerASM:
;
; Save temp. registers
;
        stmdb   SP!,{R0-R3,R12,LR}              ; push
;
; push SPSR (req. if we allow nested interrupts)
;
        mrs     R0, SPSR                        ; load SPSR
        stmdb   SP!,{R0}                        ; push SPSR_irq on IRQ stack
;
; Call "C" interrupt handler
;
        ldr     R0,=IRQ_Handler
        mov     LR,PC
        bx      R0
;
; pop SPSR
;
        ldmia   SP!, {R1}                       ; pop SPSR_irq from IRQ stack
        msr     SPSR_cxfs, R1
;
; Restore temp registers
;
        ldmia   SP!, {R0-R3,R12,LR}             ; pop
        subs    PC, LR, #4                      ; RETI
```


17.3.1.2 Device specifics ATMEL AT91CAP9x

The interrupt handler needs to read the address of the interrupt source specific handler function.

Sample implementation interrupt handler

```
#define _AIC_BASE_ADDR   (0xfffff000UL)
#define _AIC_IVR         (*(volatile unsigned int*)(_AIC_BASE_ADDR + 0x100))
#define _AIC_EOICR       (*(volatile unsigned int*)(_AIC_BASE_ADDR + 0x130))

typedef void       ISR_HANDLER(void);

void IRQ_Handler(void) {
  ISR_HANDLER* pISR;
  pISR = (ISR_HANDLER*) _AIC_IVR;      // Read interrupt vector to release
                                       // NIRQ to CPU core
  pISR();                              // Call interrupt service routine
  _AIC_EOICR = 0;                      // Reset interrupt controller =>  Restore
                                       // previous priority
}
```

17.3.1.3 Device specifics ATMEL AT91RM9200

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 434.

17.3.1.4 Device specifics ATMEL AT91SAM7A3

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 434.

17.3.1.5 Device specifics ATMEL AT91SAM7S64, AT91SAM7S128, AT91SAM7S256

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 434.

17.3.1.6 Device specifics ATMEL AT91SAM7X64, AT91SAM7X128, AT91SAM7X256

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 434.

17.3.1.7 Device specifics ATMEL AT91SAM7SE

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 434.

17.3.1.8 Device specifics ATMEL AT91SAM9260

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 434.

17.3.1.9 Device specifics ATMEL AT91SAM9261

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 434.

17.3.1.10 Device specifics ATMEL AT91SAM9263

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 434.

17.3.1.11Device specifics ATMEL AT91SAMRL64, AT91SAMR64

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 434.

17.3.1.12Device specifics NXP LPC214x

The interrupt handler needs to read the address of the interrupt source specific handler function.

Sample implementation interrupt handler

```
#define _VIC_BASE_ADDR    (0xFFFFF000)
#define _VIC_VECTORADDR   *(volatile unsigned int*)(_VIC_BASE_ADDR + 0x0030)

typedef void       ISR_HANDLER(void);

void IRQ_Handler(void) {
  ISR_HANDLER* pISR;
  pISR = (ISR_HANDLER*) _VIC_VECTORADDR;      // Get current interrupt handler
  pISR();                                     // Call interrupt service routine
  _VIC_VECTORADDR = 0;                        // Clear current interrupt pending
                                              // condition, reset VIC
}
```

17.3.1.13Device specifics NXP LPC23xx

For an example implementation of an interrupt handler function refer to *Device specifics NXP LPC214x* on page 436.

17.3.1.14Device specifics NXP (formerly Sharp) LH79524/5

For an example implementation of an interrupt handler function, please contact SEGGER, *www.segger.com*.

17.3.1.15Device specifics OKI 69Q62

For an example implementation of an interrupt handler function, please contact SEGGER, *www.segger.com*.

17.3.1.16Device specifics ST STR71x

For an example implementation of an interrupt handler function, please contact SEGGER, *www.segger.com*.

17.3.1.17Device specifics ST STR750

For an example implementation of an interrupt handler function, please contact SEGGER, *www.segger.com*.

17.3.1.18Device specifics ST STR750

For an example implementation of an interrupt handler function, please contact SEGGER, *www.segger.com*.

17.4 Writing your own driver

This section is only relevant if you plan to develop a driver for an unsupported device. Refer to *Available USB drivers* on page 428 for a list of currently supported devices.

Access to the USB hardware is realized through an API-function table. The structure `USB_HW_DRIVER` is declared in `USB\USB.h`.

17.4.1 Structure USB_HW_DRIVER

Description

Structure that contains callback function which manage the hardware access.

Prototype

```
typedef struct USB_HW_DRIVER {
  void     (*pfInit)                (void);
  U8       (*pfAllocEP)             (U8 InDir, U8 TransferType);
  void     (*pfUpdateEP)            (EP_STAT * pEPStat);
  void     (*pfEnable)              (void);
  void     (*pfAttach)              (void);
  unsigned (*pfGetMaxPacketSize)    (U8 EPIndex);
  int      (*pfIsInHighSpeedMode)   (void);
  void     (*pfSetAddress)          (U8 Addr);
  void     (*pfSetClrStallEP)       (U8 EPIndex, int OnOff);
  void     (*pfStallEP0)            (void);
  void     (*pfDisableRxInterruptEP)(U8 EpOut);
  void     (*pfEnableRxInterruptEP) (U8 EpOut);
  void     (*pfStartTx)             (U8 EPIndex);
  void     (*pfSendEP)              (U8 EPIndex, const U8 * p,
                                     unsigned NumBytes);
  void     (*pfDisableTx)           (U8 EPIndex);
  void     (*pfResetEP)             (U8 EPIndex);
  int      (*pfControl)             (U8 Cmd, void * p);

} USB_HW_DRIVER;
```

Member	Description
USB initialization functions	
`pfInit()`	Initializes the USB controller.
General USB functions	
`pfAttach()`	Indicates device attachment.
`pfEnable()`	Enables endpoint.
`pfControl`	Used to support additional driver functionality. This function is optional.
`pfSetAddress()`	Notifies the USB controller of the new address assigned by the host for it.
General endpoints functions	
`pfAllocEP`	Allocates an endpoint to be used with emUSB-Device.
`pfGetMaxPacketSize`	Returns the maximum packet size of an end-point.
`pfSetClrStallEP()`	Sets or clears the stall condition of the end-point.
`pfUpdateEP()`	Configures the USB controller's endpoint.
`pfResetEP()`	Resets an endpoint including resetting the data toggle of the endpoint.

Table 17.2: List of callback functions of USB_HW_DRIVER

Member	Description
Endpoint 0 (Control endpoint) related functions	
pfStallEP0()	Stalls endpoint 0.
OUT-endpoints functions	
pfDisableRxInterruptEP()	Disables OUT-endpoint interrupt.
pfEnableRxInterruptEP()	Enables OUT-endpoint interrupt.
IN-endpoints functions	
pfDisableTx	Disables IN endpoint transfers.
pfSendEP()	Sends data on the given IN-endpoint.
pfStartTx()	Starts data transfer on the given IN-endpoint.

Table 17.2: List of callback functions of USB_HW_DRIVER

17.4.2 USB initialization functions

17.4.2.1 (*pfInit)()

Description

Performs any necessary initializations on the USB controller.

Prototype

```
void (*pfInit)(void);
```

Additional information

The initializations performed in this routine should include what is needed to prepare the device for enumeration. Such initializations might include setting up endpoint 0 and enabling interrupts. It sets default values for EP0 and enables the various interrupts needed for USB operations.

17.4.3 General USB functions

17.4.3.1 (*pfAttach)()

Description

For USB controllers that have a USB Attach/Detach register (such as the OKI ML69Q6203), this routine sets the register to indicate that the device is attached.

Prototype

```
void (*pfAttach)(void);
```

17.4.3.2 (*pfEnable)()

Description

This function is used for enabling the USB controller after it was initialized.

Prototype

```
void (*pfEnable)(void);
```

Additional information

For most USB controllers this function can be empty. This function is only necessary for USB devices that reset their configuration data after an USB-RESET.

17.4.3.3 (*pfControl)()

Description

This function is used to support additional driver functionality. This function is optional.

Prototype

```
int (*pfControl)(U8 Cmd, void * p);
```

Parameter	Description
Cmd	Command that shall be executed.
p	Pointer to data, necessary for the command.

Table 17.3: (*pfControl)() parameter list

Return value

`== 0:` Command operation was successful.
`== 1:` Command operation was not successful.
`== -1:` Command was unknown.

Additional information

This control function is only called when available. This function will check or changes state of a device driver. Currently the following commands are available:

Command	Description
0	USB_DRIVER_CMD_SET_CONFIGURATION
1	USB_DRIVER_CMD_GET_TX_BEHAVIOR

Table 17.4: (*pfControl): Commands

17.4.3.4 (*pfSetAddress)()

Description

This function is used for notifying the USB controller of the new address that the host has assigned to it during enumeration.

Prototype

```
void (*pfSetAddress)(U8 Addr);
```

Parameter	Description
Addr	New address assigned by the USB host.

Table 17.5: (*pfSetAddress)() parameter list

Additional information

If the USB controller does not automatically send a 0-byte acknowledgment in the status stage of the control transfer phase, make sure to set a state variable to Addr and defer setting the controller's Address register until after the status stage. This is necessary because the host sends the token packet for the status stage to the default address (0x00), which means the device must still be using this address when the packet is sent.

17.4.4 General endpoint functions

17.4.4.1 (*pfAllocEP)()

Description

Allocates a physical endpoint to be used with emUSB-Device.

Prototype

```
U8 (*pfAllocEP)(U8 InDir, U8 TransferType);
```

Parameter	Description
InDir	Indicates the direction of the endpoint. 0 indicates an OUT-endpoint. 1 indicates an IN-endpoint.
TransferType	Specifies the transfer type for the desired endpoint. The following transfer types are available: USB_TRANSFER_TYPE_BULK USB_TRANSFER_TYPE_ISO USB_TRANSFER_TYPE_INT

Table 17.6: (*pfAllocEP)() parameter list

Return value

Index number of the logical endpoint. Allowed values are 1..15.

Additional information

This function is typically called after stack initialization, in order to have the right endpoint settings for building the descriptors correctly.
It is the responsibility of the driver engineer to give a valid logical endpoint number. If there is no valid endpoint for the desired configuration available, 0 should be returned.

17.4.4.2 (*pfGetMaxPacketSize)()

Description

Returns the maximum packet size of an endpoint.

Prototype

```
unsigned (*pfGetMaxPacketSize)(U8 EPIndex);
```

Parameter	Description
EPIndex	Endpoint index.

Table 17.7: (*pfGetMaxPacketSize)() parameter list

Return value

The maximum packet size in bytes.

17.4.4.3 (*pfSetClrStallEP)()

Description

Sets or clears the stall condition of an endpoint.

Prototype

```
void (*pfSetClrStallEP)(U8 EPIndex, int OnOff);
```

Parameter	Description
EPIndex	Endpoint that shall be stalled.
OnOff	Specifies if the stall condition shall be set or cleared. Whereas: 0 - Clears the stall condition. 1 - Set the stall condition.

Table 17.8: (*pfSetClrStallEP)() parameter list

Additional information

Typically, this function is called whenever a protocol/transfer error occurs.

17.4.4.4 (*pfUpdateEP)()

Description

Configures the USB controller's endpoint.

Prototype

```
void (*pfUpdateEP)(EP_STAT * pEPStat);
```

Parameter	Description
pEPStat	Pointer to EP_STAT structure that holds the information for the end-point.

Table 17.9: (*pfUpdateEP)() parameter list

Additional information

EP_STAT is defined as follows:

```
typedef struct {
  U16          NumAvailBuffers;
  U16          MaxPacketSize;
  U16          Interval;
  U8           EPType;
  BUFFER       Buffer;
  U8         * pData;
  volatile U32 NumBytesRem;
  U8           EPAddr;  // b[6:0]: EPAddr b7: Direction, 1: Device to Host (IN)
  U8           Send0PacketIfRequired;
} EP_STAT;
```

Before a hardware attach is done, this function is called to configure the desired end-points, so that the additional endpoints are ready for use after the enumeration phase.

17.4.4.5 (*pfResetEP)()

Description

Resets an endpoint including resetting the data toggle of the endpoint.

Prototype

```
void (*pfResetEP)(U8 EPIndex);
```

Parameter	Description
`EPIndex`	Endpoint that shall be reset.

Table 17.10: (*pfResetEP)() parameter list

Additional information

Resets the endpoint which includes setting data toggle to DATA0.
It is useful after removing a HALT condition on a BULK endpoint.
Refer to Chapter 5.8.5 in the USB Serial Bus Specification, Rev.2.0.
Note: Configuration of the endpoint needs to be unchanged. If the USB controller loses the EP configuration the `pfUpdateEP` of the driver shall be called.

17.4.5 Endpoint 0 (control endpoint) related functions

17.4.5.1 (*pfStallEP0)()

Description

This function is used for stalling endpoint 0 (by setting the appropriate bit in a control register).

Prototype

```
void (*pfStallEP0)(void);
```


17.4.6 OUT-endpoint functions

17.4.6.1 (*pfDisableRxInterruptEP)()

Description

Disables the OUT-endpoint interrupt.

Prototype

```
void (*pfDisableRxInterruptEP)(U8 EPIndex);
```

Parameter	Description
EPIndex	OUT-endpoint whose interrupt needs to be disabled.

Table 17.11: (*pfDisableRxInterruptEP)() parameter list

17.4.6.2 (*pfEnableRxInterruptEP)()

Description

Enables the OUT-endpoint interrupt.

Prototype

```
void (*pfEnableRxInterruptEP)(U8 EPIndex);
```

Parameter	Description
EPIndex	OUT-endpoint whose interrupt needs to be enabled.

Table 17.12: (*pfEnableRxInterruptEP)() parameter list

17.4.7 IN-endpoint functions

17.4.7.1 (*pfStartTx)()

Description

Starts data transfer on the given IN-endpoint.

Prototype

```
void (*pfStartTx)(U8 EPIndex);
```

Parameter	Description
EPIndex	IN-endpoint that needs to be enabled.

Table 17.13: (*pfStartTX)() parameter list

Additional information

This function is called to start sending data to the host.

Depending on the design of the USB controller, one of the following steps needs to be done:

If the USB controller sends a packet and waits for acceptance by the host, your application must:

- Enable IN-endpoint interrupt.
- Send a packet using `USB__Send(EPIndex)`.

If the USB controller waits for an IN-token, your application must:

- Enable the IN-endpoint interrupt.

17.4.7.2 (*pfSendEP)()

Description

Sends data on the given IN-endpoint.

Prototype

```
void (*pfSendEP)(U8 EPIndex, const U8 * p, unsigned NumBytes);
```

Parameter	Description
EPIndex	IN-endpoint that is used to send the data.
p	Pointer to a buffer that needs to be sent.
NumBytes	Number of bytes that needs to be sent.

Table 17.14: (*pfSendEP)() parameter list

Additional information

This function is called whenever data should be transferred to the host. Because `p` might not be aligned, it is the responsibility of the developer to care about the alignment of the USB controller buffer/FIFO.

17.4.7.3 (*pfDisableTx)()

Description

Disables IN-endpoint transfers.

Prototype

```
void (*pfDisableTx)(U8 EPIndex);
```

Parameter	Description
EPIndex	IN-endpoint that needs to be disabled.

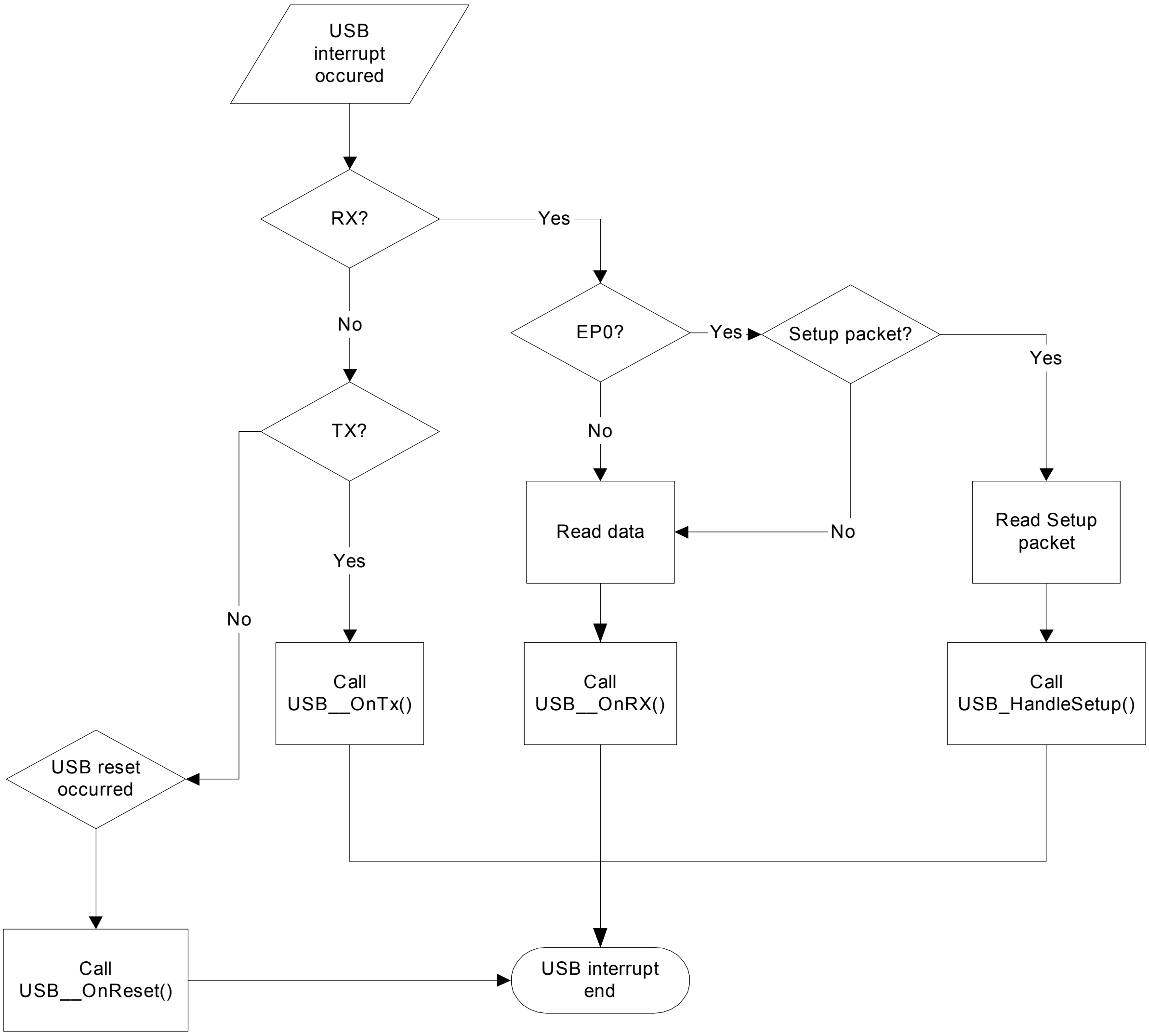
Table 17.15: (*pfDisableTx)() parameter list

Additional information

Normally, this function should disable the IN-endpoint interrupt. Some USB controllers do not work correctly after the IN interrupt is disabled, therefore this should be done by the software.

17.4.8 USB driver interrupt handling

emUSB-Device is interrupt driven. Therefore, it is necessary to have an interrupt handler for the used USB controller. For the drivers available this is already done. If you are writing your own USB driver the following schematic shows which functions need to be called when an USB interrupt occurs:



Function	Description
USB__HandleSetup()	Determines request type.
USB__OnBusReset()	Flushes the input buffer and set the "_IsInReset" flag.
USB__OnTx()	Handles a Tx transfer.
USB__OnRx()	Handles a Rx transfer.
USB__OnResume()	Resumes the device.
USB__OnSuspend()	Suspends the device.

Table 17.16: emUSB-Device interrupt handling functions

Chapter 18

Support

This chapter can help you if any problem occurs; this could be a problem with the tool chain, with the hardware, the use of the functions, or with the performance and it describes how to contact the support.

18.1 Problems with tool chain (compiler, linker)

The following shows some of the problems that can occur with the use of your tool chain. The chapter tries to show what to do in case of a problem and how to contact the support if needed.

18.1.1 Compiler crash

You ran into a tool chain (compiler) problem, not a problem with the software. If one of the tools of your tool chain crashes, you should contact your compiler support:

```
"Tool internal error, please contact support"
```

18.1.2 Compiler warnings

The code of the software has been tested with different compilers. We spend a lot of time on improving the quality of the code and we do our best to avoid compiler warnings. But the sensitivity of each compiler regarding warnings is different. So we can not avoid compiler warnings for unknown tools.

Warnings you should not see

This kind of warnings should not occur:

```
"Function has no prototype"
```

```
"Incompatible pointer types"
```

```
"Variable used without having been initialized"
```

```
'Illegal redefinition of macro'
```

Warnings you may see

Warnings such as the ones below should be ignored:

```
"Integer conversion, may lose significant bits"
```

```
'Statement not reached"
```

```
"Meaningless statements were deleted during optimization"
```

Most compilers offer a way to suppress selected warnings.

18.1.3 Compiler errors

We assume that the used compiler is ANSI C compatible. If it is compatible there should be no problem to translate the code.

18.1.4 Linker problems

Undefined externals

If your linker shows the error message “Undefined external symbols...” check if all required files have been included into the project.

18.2 Problems with hardware/driver

If your tools are working fine but your USB-Bulk device does not work may be one of the following helps to find the problem.

Stack size to low?

Make sure enough stack has been configured. We can not estimate exactly how much stack will be used by your configuration and with your compiler.

18.3 Contacting support

If you need to contact the support, send the following information

to support@segger.com:

- A detailed description of the problem
- The configuration file `USB_Conf.h`
- The error messages of the compiler

Chapter 19

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.

19.1 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 and a specific output callback was specified. 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, none of the warning messages or logging messages are activated. The provided samples contain a sample implementation how to set und output such warning and loggine messages.

19.2 API functions

Function	Description
Filter functions	
USB_SetLogFunc()	Sets the callback for outputting logging messages.
USB_SetWarnFunc()	Sets the callback for outputting warning messages.
USBD_AddLogFilter()	Adds an additional filter condition to the mask which specifies the logging messages that should be displayed.
USBD_AddWarnFilter()	Adds an additional filter condition to the mask which specifies the warning messages that should be displayed.
USBD_SetLogFilter()	Sets the mask that defines which logging message should be displayed.
USBD_SetWarnFilter()	Sets the mask that defines which warning message should be displayed.
General debug functions/macros	
USB_PANIC()	Called if the stack encounters a critical situation.
General helper prototypes	
USB_X_Log()	Template function that can be used for outputting the log messages.
USB_X_Warn()	Template function that can be used for outputting the warn messages.

Table 19.1: emUSB-Device debugging API function overview

19.2.1 USBD_SetLogFunc()

Description

Sets the function to output log messages.

Prototype

```
void USBD_SetLogFunc(void (*pfLog)(const char *));
```

Parameter

Parameter	Description
pfLog	Pointer to the function that should output the log messages

Table 19.2: USB_SetLogFunc() parameter list

Additional information

In debug build of the stack various log and warning output are generated. Those messages will only be output when an appropriate callback is set via USBD_SetLogFunc() and USBD_SetWarnFunc().

Example

```
static LogOutput(const char * s) {
  puts(s);
}


void Application(void) {
  USBD_SetLogFunc(LogOutput),
  USBD_SetLogFilter(USB_MTYPE_CORE | USB_MTYPE_INIT);
  USBD_Init();
  /*
   * Do some other USB related stuff
   */
```


19.2.2 USBD_SetWarnFunc()

Description

Sets the function to output warn messages..

Prototype

```
void USBD_SetWarnFunc(void (*pfLog)(const char *));
```

Parameter

Parameter	Description
pfLog	Pointer to the function that should output the warn messages.

Table 19.3: USBD_SetWarnFunc() parameter list

Additional information

Further information about warn messages can be found in the additional section of the function USBD_SetLogFunc().

Example

```
static WarnOutput(const char * s) {
  puts(s);
}

void Application(void) {
  USBD_SetWarnFunc(WarnOutput),
  USBD_SetWarnFilter(USB_MTYPE_CORE | USB_MTYPE_INIT);
  USBD_Init();
  /*
   * Do some other USB related stuff
   */
```

19.2.3 USBD_AddLogFilter()

Description

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

Prototype

```
void USBD_AddLogFilter(U32 FilterMask);
```

Parameter

Parameter	Description
FilterMask	Specifies which logging messages should be added to the filter mask. Refer to *Message types* on page 467 for a list of valid values for parameter FilterMask.

Table 19.4: USBD_AddLogFilter() parameter list

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

```
USBD_AddLogFilter(USB_MTYPE_DRIVER); // Activate driver logging messages
/*
 * Do something
 */
```


19.2.4 USBD_AddWarnFilter()

Description

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

Prototype

```
void USBD_AddWarnFilter(U32 FilterMask);
```

Parameter

Parameter	Description
FilterMask	Specifies which warning messages should be added to the filter mask. Refer to *Message types* on page 467 for a list of valid values for parameter FilterMask.

Table 19.5: USBD_USBD_AddWarnFilter() parameter list

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

```
USBD_AddWarnFilter(USB_MTYPE_DRIVER); // Activate driver warning messages
/*
 * Do something
 */
```


19.2.5 USBD_SetLogFilter()

Description

Sets a mask that defines which logging message that should be logged. Logging messages are only available in debug builds of emUSB-Device.

Prototype

```
void USBD_SetLogFilter(U32 FilterMask);
```

Parameter

Parameter	Description
`FilterMask`	Specifies which logging messages should be displayed. Refer to *Message types* on page 467 for a list of valid values for parameter `FilterMask`.

Table 19.6: USBD_SetLogFilter() parameter list

Additional information

This function can be called before `USBD_Init()`. By default, none of filter conditions are set. The sample application contain a simple implementation which can be easily modified.

19.2.6 USBD_SetWarnFilter()

Description

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

Prototype

```
void USBD_SetWarnFilter( U32 FilterMask );
```

Parameter

Parameter	Description
`FilterMask`	Specifies which warning messages should be displayed. Refer to *Message types* on page 467 for a list of valid values for parameter `FilterMask`.

Table 19.7: USBD_SetWarnFilter() parameter list

Additional information

This function can be called before `USBD_Init()`. By default, none of filter conditions are set. The sample application contain a simple implementation which can be easily modified.

19.2.7 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.

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 void USB_OS_Panic (const char * sError). USB_OS_Panic() disables all interrupts to avoid further task switches, outputs sError via terminal I/O and loops forever. When using an emulator, you should set a breakpoint at the beginning of this routine or simply stop the program after a failure. The error code is passed to the function as parameter.

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

19.2.8 USB_X_Log()

Description

Template function that can be used for outputting the log messages.

Prototype

```
void USB_X_Log(const char * s);
```

Parameter

Parameter	Description
s	Pointer to a string that should be output.

Table 19.8: USB_X_Log() parameter list

Additional information

This function is used in all samples provided with emUSB-Device.
It is used with the sample log implementation located under Sample\TermIO
Log output can be individually set to other functionby using the functions:
USBD_SetLogFunc().

19.2.9 USB_X_Warn()

Description

Template function that can be used for outputting the warn messages.

Prototype

```
void USB_X_Warn(const char * s);
```

Parameter

Parameter	Description
s	Pointer to a string that should be output.

Table 19.9: USB_X_Warn() parameter list

Additional information

This function is used in all samples provided with emUSB-Device.
It is used with the sample warn implementation located under Sample\TermIO
Warn output can be individually set to other functionby using the functions:
USBD_SetWarnFunc().

19.3 Message types

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()* on page 462 and *USBD_SetWarnFilter()* on page 463 as wells as *USBD_AddLogFilter()* on page 460 and *USBD_AddLogFilter()* on page 460 for more information about using the message types.

Symbolic name	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_TRANSFER	Activates output of messages from data transfers other than enumeration should be logged.
USB_MTYPE_IAD	Activates output of messages from the IAD module.
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 CD-ROM 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.
USB_MTYPE_PRINTER	Activates output of messages from Printer module that should be logged when a Printer connection is used.
USB_MTYPE_RNDIS	Activates output of messages from RNDIS-module that should be logged when a RNIDS connection is used.
USB_MTYPE_SMART_MSD	Activates output of messages from SmartMSD module that should be logged when a SmartMSD connection is used.
USB_MTYPE_UVC	Activates output of messages from UVC module that should be logged when a UVC connection is used.

Table 19.10: emUSB-Device message types

Chapter 20

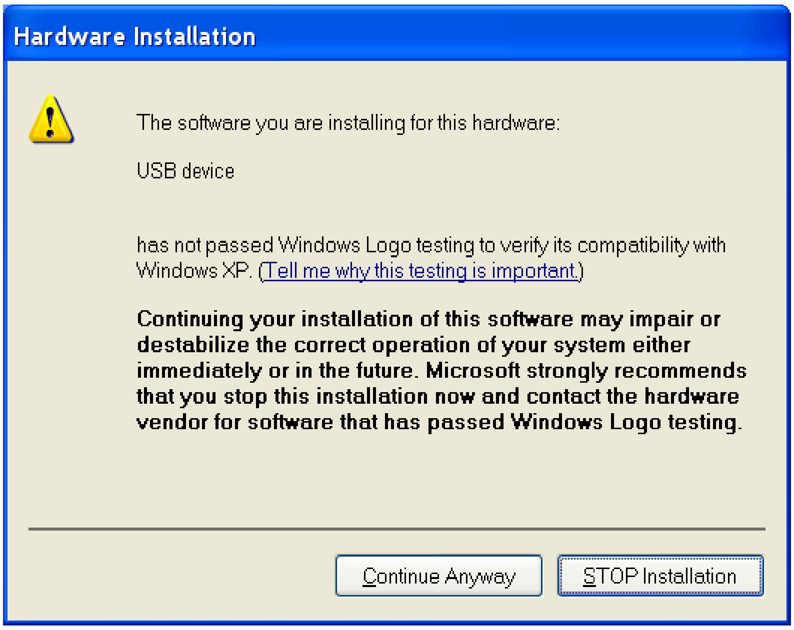
Certification

This chapter describes the process of USB driver certification with Microsoft Windows.

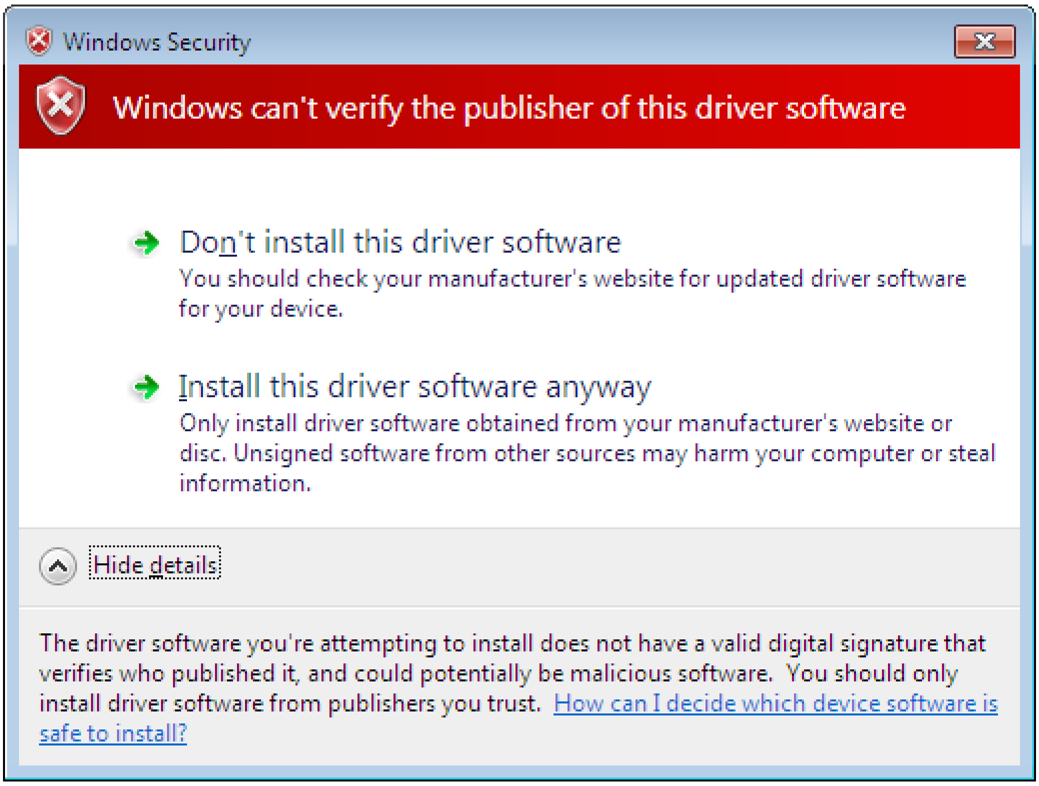
20.1 What is the Windows Logo Certification and why do I need it?

The Windows Logo Certification process will sign the driver with a Microsoft certificate which signifies that the device is compatible and safe to use with Microsoft Windows operating systems.
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 and Windows 7 x64 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 XP:



Microsoft Windows Vista/7:



20.2 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 www.usb.org and a free Product ID. Using the Microsoft Windows Logo 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.

20.3 Vendor and Product ID

A detailed description of the Vendor and Product ID can be found in chapter *Product / Vendor IDs* on page 31
The customer can acquire a Vendor ID from the USB Implementers Forum, Inc. (www.usb.org). This allows to freely decide which Product ID is used for which product.

20.4 Certification without SEGGER Microcontroller

Certification can be completed by the customer themselves. To complete the certification the Windows Logo Kit software is needed. It has to be installed on a Windows 2008 Server x64. A Code Signing certificate from Microsoft, two target devices and two client computers will also be needed, Windows 7 x86 and Windows 7 x64 respectively. After installing and setting up the WLK, the client software has to be downloaded via a Windows share from the Windows 2008 Server. The target devices will have to be connected to the client computer.

Using the WLK, the target devices can be selected and the appropriate tests can be scheduled. A few of the tests need human intermission and a few tests only run with one device, while others only run with two. The tests can take up to 15 hours. The tests have to be done separately for x86 and x64. Two separate submission packages have to be created for both architectures. The submission packages have to be consolidated using the Winqual Submission Tool and signed with the Code Sign certificate.

For further information, as well as the required software see:
http://msdn.microsoft.com/en-us/library/windows/hardware/gg487530.aspx

Please refer to Microsoft’s WLK documentation for a detailed description of the certification process.

Chapter 21

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.

21.1 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. Note that the values are only valid for the given configuration.

The tests were run on a Cortex-M4 CPU. The test program was compiled for size optimization.

21.1.1 ROM

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

Description	ROM
emUSB-Device core	app. 5.2 - 5.6 Kbytes
Bulk component	app. 220 - 400 Bytes
MSD component	app. 5 Kbytes + sizeof(Storage-Layer)*
HID component	app. 1000 Bytes
CDC component	app. 850 Bytes - 1.1 Kbytes
PrinterClass component	app. 460 Bytes
USB target driver	app. 1.2 - 3 Kbytes
MTP component	app. 10.5 kBytes sizeof(Storage-Layer)*
RNDIS component	app. 3 KBytes + sizeof(IP stack)
CDC-ECM component	app. 1.2 KBytes
SmartMSD component	app. 4 KBytes + sizeof(MSD)

* ROM size of emFile Storage app. 4 Kbytes.

21.1.2 RAM

The following table shows the RAM requirement of emUSB-Device:

Description	RAM
emUSB-Device core	app. 800 Bytes
Bulk component	app. 4 Bytes
MSD component	app. 270 Bytes + configurable sector buffer of minimum 512 bytes
HID component	app. 100 Bytes
CDC component	app. 100 Bytes
PrinterClass component	app. 2 Kbytes
USB target driver	< 1 KByte - 5 KByte. Highly depends on the number of endpoints, USB Speed and Controller architecture (DMA, buffer alignment, etc.)
MTP component	app. 1.2 KBytes + configurable file data buffer of minimum 512 bytes + configurable object buffer (typically 4 kBytes).

Description	RAM
RNDIS component	app. 1.8 KBytes + sizeof(IP stack)
CDC-ECM component	app. 1.6 KBytes + sizeof(IP stack)
SmartMSD component	app. 600 Bytes + sizeof(MSD)

Additionally 64 or 512 bytes (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.

21.2 Performance

The tests were run on a LPC4357 CPU running at 180 MHz using the USB Bulk and the USB MSD component.

The following table shows the transfer speed of emUSB-Device:

Description	Speed
USB High-Speed controller	39.2 MByte/sec
USB Full-Speed controller	1170 KByte/sec

Chapter 22

FAQ

This chapter answers some frequently asked questions.

Q: Which CPUs can I use emUSB-Device with?
A: It can be used with any CPU (or MPU) for which a C compiler exists. Of course, it will work faster on 16/32-bit CPUs than on 8-bit CPUs.

Q: Do I need a real-time operating system (RTOS) to use the USB-MSD?
A: No, if your target application is a pure storage application. You do not need an RTOS if all you want to do is running the USB-MSD stack as the only task on the target device. If your target application is more than just a storage device and needs to perform other tasks simultaneously, you need an RTOS which handles the multi-tasking.
We recommend using our embOS Real-time OS, since all example and trial projects are based on it.

Q: Do I need extra file system code to use the USB-MSD stack?
A: No, if you access the target data only from the host.
Yes, if you want to access the target data from within the target itself.
There is no extra file system code needed if you only want to access the data on the target from the host side. The host OS already provides several file systems. You have to provide file system program code on the target only if you want to access the data from within the target application itself.

Q: Can I combine different USB components together?
A: In general this is possible, by simply calling the appropriate add function of the USB component. See more information in *Combining USB components (Multi-Interface)* on page 397.

