

CDC/ACM Device Class

for Embedded USB Device Stack

Reference Manual

Version: 1.14.0
Date: 28 September 2011

Authors: Günter Hildebrandt

Thesycon Systemsoftware & Consulting GmbH
Werner-von-Siemens-Str. 2
D-98693 Ilmenau
Germany

Tel: +49 3677 8462 0
Fax: +49 3677 8462 18

<http://www.thesycon.de>

Copyright (c) 2006-2011 by Thesycon® Systemsoftware & Consulting GmbH
All Rights Reserved

Disclaimer

The information in this document is subject to change without notice. No part of this manual may be reproduced, stored in a retrieval system, or transmitted in any form or by any means electronic or mechanical, including photocopying and recording for any purpose other than the purchaser's personal use, without prior written permission from Thesycon® Systemsoftware & Consulting GmbH.

The software described in this document is furnished under the software license agreement distributed with the product. The software may be used or copied only in accordance with the terms of the license.

Trademarks

The following trade names are referenced within this manual:

Microsoft, Windows, Win32, Windows NT, Windows XP, and Visual C++ are either trademarks or registered trademarks of Microsoft Corporation.

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

Contents

Table of contents	9
1 Introduction	12
1.1 The USB Function Library	12
1.2 The CDC/ACM Device Class	12
2 Overview	13
2.1 Features of the Embedded USB Device Stack	13
2.2 Features of the USB Function Library	13
2.3 Features of the CDC/ACM Device Class	14
2.4 Restrictions of the USB Function Library	14
2.5 Known Problems	14
2.5.1 Full Speed Device on External High Speed Hub	14
2.6 Hardware Requirements	15
2.6.1 SoftConnect Circuit	15
2.6.2 VBus Detection	15
2.7 Supported Device Controllers	16
2.8 Memory Footprints	16
3 Architecture	18
3.1 Hardware Abstraction Layer	19
3.2 Thesycon Adaption Layer	20
3.3 USB Function Library	20
3.3.1 USB Function Core	20
3.3.2 USB Device Controller Driver	20
3.4 Device Classes	21
3.5 CDC/ACM Device Class	21
4 Configuration	22
4.1 Compile Time Configuration	22
4.1.1 USB Function Library	22
4.1.2 USB Descriptors	22
4.1.3 Device Controller	22
4.1.4 CDC/ACM Device Class	22
4.2 USB Descriptors	22

4.3	Vendor ID and Product ID	23
4.4	Serial Number	23
5	Usage	24
5.1	Initial Steps	24
5.2	Initialization	25
5.3	Synchronization	25
5.4	Callbacks	25
5.5	Interrupt Handling	26
5.6	VBus Sensing	27
5.7	Plug and Play Handling	27
5.8	Handling of Class and Vendor Requests	28
5.9	Error Recovery	28
5.10	Buffer Chain	28
5.11	Device Controller Specific Adaptation	29
5.11.1	Customization for Atmel AT91SAM based devices	29
5.11.2	Customization for Atmel AT91SAM3U based devices	30
5.11.3	Customization for Atmel AVR32 based devices	30
5.11.4	Customization for Philips/NXP ISP1582	32
5.11.5	Customization for Philips/NXP LPC23xx	33
5.11.6	Customization for Philips/NXP LPC24xx	33
5.11.7	Customization for Philips/NXP LPC17xx	34
5.11.8	Customization for Renesas H8SX/1653, H8SX/1663 Group	34
5.11.9	Customization for Renesas H8SX/1668 Group	36
5.11.10	Customization for Renesas H8S/2472 Group	37
5.11.11	Customization for Renesas H8S/2215 Group	38
5.11.12	Customization for Renesas M16C6C Group	39
5.11.13	Customization for STMicroelectronics STR91x	40
6	USB Function Library Reference	42
6.1	Initialization	42
	Usbf_Initialize	42
	Usbf_Enable	43
	Usbf_Disable	44
	Usbf_RegisterDeviceHandler	45
	Usbf_UnregisterDeviceHandler	46

Usbf_InterruptServiceRoutine	47
Usbf_ProcessInterrupt	48
Usbf_CheckVBus	49
Usbf_SetTraceMask	50
6.2 Callbacks	52
T_Usbf_DeviceEvent_Callback	52
T_Usbf_StartOfFrame_Callback	55
6.3 Structures	56
T_UsbfBufferDescriptor	56
T_UsbfDescriptorRecord	59
T_UsbfStringDescriptorRecord	60
T_UsbfDescriptors	61
T_UsbfDeviceHandler	63
6.4 Status Codes	64
USBF_STATUS_SUCCESS	64
USBF_STATUS_CANCELED	64
USBF_STATUS_BUFFER_OVERFLOW	64
USBF_STATUS_INVALID_PARAM	64
USBF_STATUS_PENDING	64
USBF_STATUS_FIFO_FULL	64
USBF_STATUS_FIFO_EMPTY	64
USBF_STATUS_ENDPOINT_STALLED	64
USBF_STATUS_NOT_OPENED	64
USBF_STATUS_HW_ACCESS_FAILED	64
USBF_STATUS_DEFERRED_DATA_STAGE	65
USBF_STATUS_STALL	65
USBF_STATUS_INVALID_DATA	65
USBF_STATUS_OPEN_ENDPOINT_FAILED	65
USBF_STATUS_INSTANCE_DEACTIVATED	65
USBF_STATUS_NOT_SUSPENDED	65
USBF_STATUS_NOT_ENABLED	65
USBF_STATUS_NOMEMORY	65
USBF_STATUS_ISO_CRC	65
USBF_STATUS_ISO_SEQUENCE	66
USBF_STATUS_UNDEFINED	66

7	CDC/ACM Device Class Reference	67
7.1	Initialization	67
	UsbCdcAcm_Initialize	67
	UsbCdcAcm_CreateInstance	68
	UsbCdcAcm_DeleteInstance	69
	UsbCdcAcm_ActivateInstance	70
	UsbCdcAcm_DeactivateInstance	72
7.2	API Functions	73
	UsbCdcAcm_RegisterCallbacks	73
	UsbCdcAcm_SetSerialState	75
	UsbCdcAcm_GetSerialLineCoding	76
	UsbCdcAcm_SubmitTxBuffer	77
	UsbCdcAcm_SubmitRxBuffer	79
	UsbCdcAcm_AbortTx	81
	UsbCdcAcm_AbortRx	82
	UsbCdcAcm_SetTraceMask	83
7.3	Callbacks	85
	T_UsbCdcAcm_LineStateChanged_Callback	85
	T_UsbCdcAcm_TxBufferCompletion_Callback	87
	T_UsbCdcAcm_RxBufferCompletion_Callback	88
7.4	Status Codes	89
	USBF_STATUS_NO_INSTANCE	89
8	Hardware Abstraction Layer Reference	90
8.1	API Functions	90
	UsbHal_WriteReg8	90
	UsbHal_BurstWriteReg8	91
	UsbHal_WriteReg16	92
	UsbHal_BurstWriteReg16	93
	UsbHal_WriteReg32	94
	UsbHal_BurstWriteReg32	95
	UsbHal_ReadReg8	96
	UsbHal_BurstReadReg8	97
	UsbHal_ReadReg16	98
	UsbHal_BurstReadReg16	99

UsbHal_ReadReg32	100
UsbHal_BurstReadReg32	101
UsbHal_FifoWrite	102
UsbHal_FifoRead	103
UsbHal_IsVBusPresent	104
UsbHal_SetSoftConnect	105
UsbHal_Delay	106
9 Adaption Layer Reference	107
9.1 API Functions	107
Tal_Printf	107
Tal_ZeroMemory	108
Tal_CopyMemory	109
Tal_CompareMemory	110
Tal_SetMemory	111
Tal_OffsetOfMember	112
Tal_GetTickCount	113
10 Demo Applications	114
10.1 Microsoft USB CDC/ACM Device Driver	114
10.2 Device Driver for the PC	114
10.2.1 Demo Limitation	114
10.2.2 Device Driver Installation	114
10.2.3 CDC/ACM Device Driver	115
10.2.4 CDC/ACM MI Device Driver	115
10.3 CDC/ACM Loopback Demo Application	115
10.3.1 Using the CDC/ACM Device	116
Index	117

References

- [1] Universal Serial Bus Revision 2.0 specification,
<http://www.usb.org>
- [2] Microsoft Developer Network (MSDN) Library,
<http://msdn.microsoft.com/library/>
- [3] Windows Driver Development Kit,
<http://msdn.microsoft.com/library/>
- [4] Universal Serial Bus Class Definitions for Communication Devices,
http://www.usb.org/developers/devclass_docs/usbc11.pdf

1 Introduction

The Embedded USB Device Stack is a software stack for any embedded controller. It can be used with any USB device controller. The USB function (this is the term the USB specification refer to a USB device) can either be included in the microcontroller or be connected to an external bus of the microcontroller.

This document gives an overview of the Embedded USB Device Stack. It describes its architecture, its features and its restrictions. Furthermore, it includes instructions for the integration in a project. The configuration and the usage of the Embedded USB Device Stack are described in separate chapters. Additionally the references of the programming interfaces are included in this document.

The Embedded USB Device Stack is available for different device controllers. Refer to [2.7](#) for an overview of supported device controllers.

1.1 The USB Function Library

The USB Function Library represents the lower layer of the Embedded USB Device Stack. It encapsulates the device controller specific USB implementation and provides generic USB functionality on its upper programming interface.

For the chapters depending on the USB Function Library the reader of this document is assumed to be familiar with the specification of the Universal Serial Bus Version 1.1 and 2.0 and with common aspects of C-based programming.

1.2 The CDC/ACM Device Class

The CDC/ACM Device Class represents the upper layer of the Embedded USB Device Stack: It extends the Embedded USB Device Stack with the implementation of a standard-compliant serial link (RS-232) emulation on top of the USB Function Library. This device class can be used by an embedded application to emulate a RS-232 serial link over USB without the necessity of dealing with the USB specification.

This document describes the architecture, the features and the programming interface of the CDC/ACM Device Class.

The reader of this document is assumed to be familiar with common aspects of C-based programming. For use of the CDC/ACM Device Class no advanced USB knowledge is required.

2 Overview

2.1 Features of the Embedded USB Device Stack

The Embedded USB Device Stack is designed to cover a USB function peripheral in an embedded environment. It consists of different modules which can be adjusted and combined to satisfy your requirements. This modular concept can easily be enhanced by your own implementations or by prepackaged components provided by Thesycon® Systemsoftware & Consulting GmbH.

The Embedded USB Device Stack provides the following features:

- handles all standard requests and the complete enumeration
- supports multi-configuration devices
- supports multi-interface devices
- implements an extended error recovery
- implemented in ANSI C
- independent of the processors memory alignment
- works in either endian mode
- simple hardware abstraction layer (HAL)
- simple adaption layer (TAL)
- can be used stand-alone or with an operating system
- compliant to the USB 2.0 specification
- compliant to USB Command Verifier and the WHQL certification (HCT12.1)

2.2 Features of the USB Function Library

The USB Function Library handles the register-based access to the specific device controller and provides a convenient and easy to use software API. Therefore no knowledge of the device controller specific USB implementation is required.

The USB Function Library provides the following features:

- encapsulated device controller specific implementation
- all USB standard requests are handled
- support for soft connect
- support for connect detection
- easy to use software API
- well documented API reference

2.3 Features of the CDC/ACM Device Class

The CDC/ACM Device Class handles the USB specific access to the USB Function Library and provides a convenient and easy to use software API. Therefore no advanced knowledge of the USB specification is required.

The CDC/ACM Device Class provides the following features:

- encapsulates USB specific implementation
- provides a serial port emulation on the host
- easy migration for existing applications using serial link
- easy to use software API
- well documented API reference
- appropriate PC driver for Windows is also available
- exposes a Win32-compatible COM port

2.4 Restrictions of the USB Function Library

The following restrictions exist:

- Only Interface and Endpoint are supported recipients for class/vendor requests. Refer to section 5.8 for more information about class/vendor requests.
- The USB Function Library is not reentrant and provides no synchronization primitives internally. Refer to section 5.3 for more details about the synchronization of the USB Function Library.

2.5 Known Problems

2.5.1 Full Speed Device on External High Speed Hub

All external hubs available today are USB 2.0 high-speed hubs. If such an external hub is attached to a PC then the USB connection between hub and PC operates at high speed (480 Mbps). If a USB full speed (12 Mbps) device is attached to one of the down stream ports of the hub then the hub is required to perform a speed conversion. In this mode the hub operates as a full-speed host which translates high-speed transactions received from the PC to execute them at full speed. This design is defined by the USB 2.0 specification.

Theyson found that there are issues with many external USB 2.0 high speed hubs if they operate in speed conversion mode. Many implementations are not fully compliant with the USB 2.0 specification and cause problems in this mode. For this reason, Theyson recommends to avoid using an external hub if it needs to operate in speed conversion mode. This applies if a full-speed device is attached to the external hub and the hub is connected to a PC.

Note that there is no problem if the external hub operates at the same speed at its upstream and its downstream port. This is the case if a high-speed device is connected to the hub. Note also that

there are no issues if a full-speed USB device is connected to a PC directly. In this case it will be attached to a full-speed host in the PC and no speed conversion takes place. Refer to the USB specification for a more detailed description of scenarios where speed conversion is required.

2.6 Hardware Requirements

2.6.1 SoftConnect Circuit

A USB device requires a switchable pull-up resistor which connects the USB D+ line to 3.3V. This SoftConnect feature enables the USB device to be connected/disconnected from USB by software. This ensures that the USB device is connected to USB after the software initialization finished and VBus is available.

If the SoftConnect feature is not realised inside the USB device it has to be implemented externally by the hardware designer using a GPIO pin. This pin has to be handled in the function **UsbHal_SetSoftConnect** to connect/disconnect the USB device by software.

The SoftConnect feature is required for USB bus powered and self powered devices. If SoftConnect is not available a race condition between the startup time of the micro-controller and the enumeration of the USB device occurs.

There exists a workaround in the USB Function Library for USB devices which are hard wired (no SoftConnect feature).

In the file **config.h** the define **USBF_CONNECT_R_HARD_WIRED** must be set to **1** to inform the USB Function Library that the USB device has no SoftConnect feature. Also the USB device may work without the SoftConnect feature it can not be guaranteed that it will work reliable on all host controllers. So this setting is only applicable if the hardware design can not be changed and it has to be treated as last chance. On default the define **USBF_CONNECT_R_HARD_WIRED** must be set to **0**. An compile error is generated if the define **USBF_CONNECT_R_HARD_WIRED** does not exist.

2.6.2 VBus Detection

The VBus detection enables the software to be informed about the current state of VBus. This is necessary for the software in order to decide when the USB device should be connected to USB. As required by the USB specification the SoftConnect feature will only be used by the software if VBus is present.

If the VBus detection is not realised inside the USB device it should be implemented externally by the hardware designer using a GPIO pin. This pin has to be interpreted in the function **UsbHal_IsVBusPresent** to inform the software of the VBus state. The application has to call **Usb_CheckVBus()** periodically so the USB Function Library checks if VBus is present.

The VBus detection is mainly required for self powered devices. On USB bus powered devices VBus can be assumed as always present.

Device Controller	Supported Speed(s)	Customization Section
Atmel AT91SAM based devices	Full Speed	5.11.1
Atmel AT91SAM3U based devices	Full Speed, High Speed	5.11.2
Atmel AVR32 based devices	Full Speed, High Speed	5.11.3
Philips/NXP ISP1582	Full Speed, High Speed	5.11.4
Philips/NXP LPC23xx	Full Speed	5.11.5
Philips/NXP LPC24xx	Full Speed	5.11.6
Philips/NXP LPC17xx	Full Speed	5.11.7
Renesas H8SX/1653 Group	Full Speed	5.11.8
Renesas H8SX/1663 Group	Full Speed	5.11.8
Renesas H8SX/1668 Group	Full Speed	5.11.9
Renesas H8S/2472 Group	Full Speed	5.11.10
Renesas H8S/2215 Group	Full Speed	5.11.11
Renesas M16C6C Group	Full Speed	5.11.12
STMicroelectronics STR91x	Full Speed	5.11.13

Table 1: Overview of supported device controllers

2.7 Supported Device Controllers

Thesycon's Embedded USB Device Stack currently supports the device controllers listed in the table below. Because the hardware implementation differs from each other some adaptations has to be made. Refer to section [5.11](#) for more information about the adaptation and the required compile time settings for a specific device controller.

2.8 Memory Footprints

The code size and the memory requirements mainly depend on the processor architecture, the bit alignment as well as the used compiler and its optimization settings.

The following values are achieved on an ARM7 CPU in the release build without debug information. The ARM RealView compiler was used with default optimization (-O2). Data buffers as well as USB descriptors are not considered because they belong to the application layer. They

Modul	Code + const data (FLASH)	Data (RAM)	Description
USB Function	≈ 10 KB	≈ 0.5 KB	native USBF with 2 endpoints
CDC/ACM	≈ 1.7 KB	≈ 0.2 KB	single CDC/ACM instance with 3 endpoints
RNDIS	≈ 3.5 KB	≈ 1.8 KB	single RNDIS instance with 3 endpoints
MSD	≈ 6 KB	≈ 0.4 KB	single MSD instance without CD-ROM support
MSD	≈ 7 KB	≈ 0.4 KB	single MSD instance with CD-ROM support
THID	≈ 0.8 KB	≈ 0.1 KB	single THID instance with 2 endpoints

Table 2: Overview of memory consumption and code size

have to be allocated in the application outside the Embedded USB Device Stack. Depending on the number of endpoints and the configuration of the device USB descriptors usually consume between 150 and 400 bytes of constant data.

For more detailed information on specific configurations please contact [usb\(at\)thesycon.de](mailto:usb(at)thesycon.de).

3 Architecture

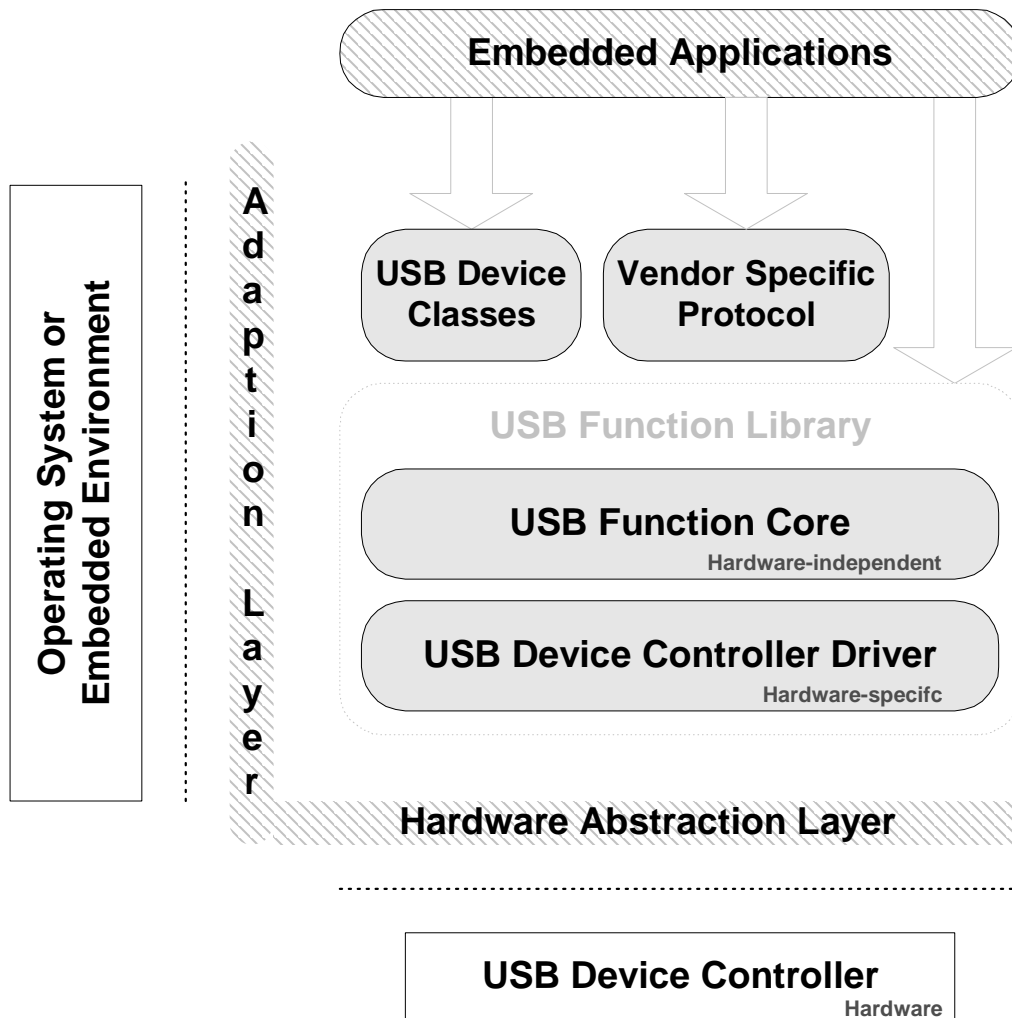


Figure 1: Components of the Embedded USB Device Stack

The Embedded USB Device Stack consists of the following components:

- **USB Device Controller Driver** - This component is part of the USB Function Library and encapsulates the device controller specific USB implementation. Refer to section 3.3 for more information.
- **USB Function Core** - This component is part of the USB Function Library and provides the generic USB functionality. Refer to section 3.3 for more information.
- **USB Device Class** - This component is optional to the USB Function Library. It could be the implementation of a defined protocol like RNDIS or represent a vendor specific protocol which provides additional functionality at a higher level. Refer to section 3.4 for more information about device classes.

The following software layers are required for the integration of the Embedded USB Device Stack:

- **Adaption Layer** - This component provides the abstraction to the operating system or embedded environment. Refer to section 3.2 for more information.
- **Hardware Abstraction Layer** - This component provides the abstraction of the hardware access. Refer to section 3.1 for more information.

The component called **embedded Application** implements the real functionality. This has to be implemented by the application designer upon the purpose of the USB device. Refer to section 10 for some simple demo applications implemented on top of the USB Function Library or a specific device class.

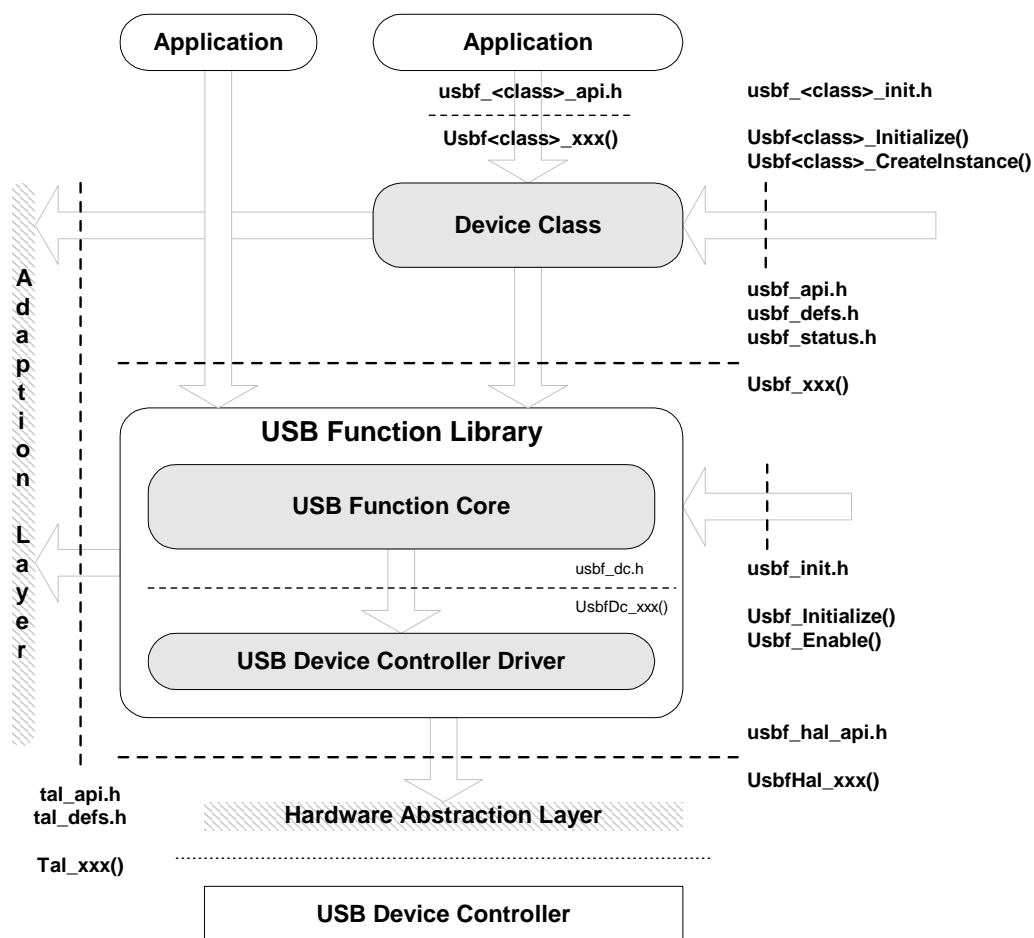


Figure 2: Architecture of the Embedded USB Device Stack

3.1 Hardware Abstraction Layer

This software layer provides the abstraction of the hardware access. It encapsulates the access to the registers of the device controller with the appropriate functions defined in the file

usbf_hal_api.h. These functions are device controller specific and have to be implemented by the application designer according to the used device controller and its environment.

Refer to section 2.7 to see which functions have to be implemented for a specific device controller.

Refer to section 8 for the API reference of the hardware abstraction layer.

3.2 Thesycon Adaption Layer

This software layer provides the adaption to the operating system or embedded environment. It provides common functionality like **memcpy** or **printf** with the appropriate functions defined in the file **tal_api.h**. These functions have to be implemented by the application designer according to the used operating system or embedded environment. Furthermore the adaption layer provides definitions of basic data types in the file **tal_defs.h**. It is possible that the application designer has to change these defines according to the deployed micro processor.

Refer to section 9 for the API reference of the adaption layer.

3.3 USB Function Library

The USB Function Library is a software layer which is designed to handle the complete USB function peripheral. It implements all USB standard requests, the required state machines and the data transfer. An embedded application program does not have to access the USB peripheral via registers, it simply uses the software API defined by the USB Function Library.

The software API of the USB Function Library is defined in the file **usbf_api.h**. This file contains the definitions of the functions which are implemented within the library. There are also callback functions defined which will be implemented by the application designer outside the library. But not all of these callback functions has to be provided. The implementation of some callback functions is optional. Refer to section 6.2 to see which callback functions have to be implemented.

3.3.1 USB Function Core

The USB Function Core implements the generic USB functionality. It is responsible for the enumeration and handles all kinds of standard requests. This part of the USB Function Library is common to all device controllers. The USB Function Core is independent from the USB Device Controller Driver implementation and has no access to any device specific functionality. It only uses the USB Device Controller Driver to implement the generic USB functionality.

3.3.2 USB Device Controller Driver

The USB Device Controller Driver implements the behavior of a specific device controller. It is responsible for initialization, register based access and interrupt handling of the chosen device. The USB Device Controller Driver does not implement any USB functionality itself. It is used by the USB Function Core to implement the generic USB functionality for a selected device controller.

3.4 Device Classes

Device classes for the Embedded USB Device Stack extend the USB Function Library by a defined protocol like RNDIS or CDC/ACM. A device class encapsulates the USB specific communication with the USB Function Library and provides an easy to use software API appropriate to the implemented protocol. Therefore no advanced USB knowledge is required on top of the device class API.

3.5 CDC/ACM Device Class

The CDC/ACM Device Class is an optional component of the Embedded USB Device Stack provided by Thesycon® Systemsoftware & Consulting GmbH. It is located on top of the USB Function Library. It extends the Embedded USB Device Stack by implementing a standard-compliant serial link (RS-232) emulation.

The CDC/ACM Device Class implements the Abstract Control Model (ACM) subclass defined in the USB Communication Device Class (CDC) specification. This protocol requires two USB interfaces, a data interface which consists of two endpoints (bulk in + bulk out) and a control interface which consists of one interrupt in endpoint. Furthermore, on EP0 (control endpoint) the device needs to implement the class-specific requests defined by the CDC specification. Because this USB protocol is compliant to the CDC/ACM standard on some operating systems (e.g. Linux, MacOS) the device can work with in-box drivers. Thesycon® Systemsoftware & Consulting GmbH also offers a compliant driver for Windows operating systems. Refer to <http://www.thesycon.de/cdcacm> for more information about the suitable CDC/ACM class driver for Windows.

On the lower edge the CDC/ACM Device Class communicates with the USB Function Library by using the software API defined in the file `usbf_api.h`. On the upper edge the CDC/ACM Device Class provides a software API for use by embedded applications. The functions of this software API are defined in the file `cls_cdcacm_api.h`. There are also callback functions defined which will be implemented by the application designer outside the CDC/ACM Device Class. Not all of these callback functions has to be provided. The implementation of some callback functions is optional. Refer to section 7.3 to see which callback functions have to be implemented.

4 Configuration

4.1 Compile Time Configuration

The compile time configuration of the Embedded USB Device Stack has to be customized to meet the requirements of the application and to adapt the various modules of the device stack to a specific environment.

All compile time settings that are available for a specific module are defined in the configuration file of the module. All module configuration files include the `config.h` configuration file at first. Thus, such a configuration file `config.h` has to be provided for a specific application to set required settings and to change default settings, if required.

The configuration file `config.h` is provided for each demo application of the Embedded USB Device Stack.

4.1.1 USB Function Library

For a description of all available configuration settings of the USB Function Library please refer to the file `/source/usbfunc/usbfunc_config.h`.

4.1.2 USB Descriptors

Each demo application of the Embedded USB Device Stack contains an example definition of the required USB descriptors. Some values of these descriptors can be adapted to a specific platform by configuration settings. Please refer to the file `/source/app_XXX/app_XXX_descriptors.h` for available configuration settings.

4.1.3 Device Controller

Please refer to section 5.11 for more information about the adaptation and required compile time settings for a specific device controller.

4.1.4 CDC/ACM Device Class

For a description of all available configuration settings of the CDC/ACM Device Class please refer to the file `/source/cls_cdcacm/cls_cdcacm_config.h`.

4.2 USB Descriptors

The USB descriptors are used to describe the USB device with its supported interfaces and endpoints. Examples for USB descriptors are provided for each demo application. The application designer has to customize some settings in order to use these descriptors in an own application. Refer to sections 4.1.2, 4.3 and 4.4 for more information about customizing the USB descriptors.

For detailed information about USB descriptors refer to [1] section 9.5.

4.3 Vendor ID and Product ID

The Vendor ID (VID) and the Product ID (PID) of the USB descriptors have to be provided by the manufacturer of a device. A VID can be obtained from the USB Implementers Forum. Refer to <http://www.usb.org> for more information. It is also possible to obtain a PID from Thesycon. In this case you have to use the VID of Thesycon. Please contact [info\(at\)thesycon.de](mailto:info(at)thesycon.de) for more information.

The PID specifies a certain device. The manufacturer of the device takes care about the PID used for a specific device. He is responsible that it is non-ambiguous for each device.

Note that the VID and PID provided in the USB descriptor are used on some operating systems (e.g. Windows) to identify the USB device.

4.4 Serial Number

The application designer has to decide if a serial number should be exposed for the USB device or not. This decision can influence the enumeration behavior on the host, especially on computers running a Windows operating system.

If the USB device exposes a serial number the Windows operating system installs the drivers for the device only once. Even if you use the USB device on a different USB port the device will be recognized as already installed.

If the USB device does not expose a serial number the Windows operating system installs the drivers for the device for each USB port the device is plugged in.

Note that if exposing a serial number, the serial number has to be unique for each USB device. Using a serial number for multiple devices will lead to errors.

5 Usage

5.1 Initial Steps

1. Configure the USB Function Library and the CDC/ACM Device Class to meet your requirements. You have to adjust the appropriate settings for the USB Function Library (file **usbf_config.h**) and for the CDC/ACM Device Class (file **cls_cdcacm_config.h**) in the configuration file **config.h**. Furthermore you have to customize the USB descriptors.

Refer to section 4 for detailed information about the configuration of the USB Function Library and CDC/ACM Device Class.

2. Adjust the basic integer types in **tal_defs.h** to match your environment. Implement the necessary functions declared in **usbf_hal_api.h** and **tal_api.h**.

Refer to section 2.7 to see which functions you have to implement for your specific device controller.

3. Implement the interrupt handling.

The functions for the interrupt handling are declared in the file **usbf_init.h**.

Refer to section 5.5 for detailed information about the interrupt handling in the USB Function Library.

4. Initialize the USB Function Library by a call to **Usbf_Initialize**. Initialize the CDC/ACM Device Class by a call to **UsbfCdcAcm_Initialize**. Create a CDC/ACM Device Class instance by a call to **UsbfCdcAcm_CreateInstance**.

The functions **UsbfCdcAcm_Initialize** and **UsbfCdcAcm_CreateInstance** are declared in the file **cls_cdcacm_init.h**. The function **Usbf_Initialize** is declared in the file **usbf_init.h**.

Refer to section 5.2 for detailed information about the initialization of the Embedded USB Device Stack.

5. Register a device handler by a call to **Usbf_RegisterDeviceHandler** which informs you about device related events.

The function **Usbf_RegisterDeviceHandler** is declared in the file **usbf_init.h**.

Refer to section 5.7 for more information about device events.

6. Enable the USB Function Library by a call to **Usbf_Enable**. After this call, the PC should recognize the USB device and the enumeration process should start. The operating system will now ask you for a driver for your device or will use a suitable in-box driver. Thesycon® Systemsoftware & Consulting GmbH also offers a suitable CDC/ACM device driver for Windows operating systems. Refer to <http://www.thesycon.de/cdcacm> for more information about the suitable CDC/ACM class driver for Windows.

The function **Usbf_Enable** is declared in the file **usbf_init.h**.

7. After this installation process the host will configure the device and the previously registered device handler will receive the device event **USBF_DEV_EVENT_CONFIGURE**.

Upon this event call **UsbCdcAcm_ActivateInstance** to activate the CDC/ACM Device Class instance.

After that your CDC/ACM Device Class instance is ready to use. Now you are able to communicate via the serial line using the functions declared in the file **cls_cdcacm_api.h**.

UsbCdcAcm_ActivateInstance is declared in the file **cls_cdcacm_init.h**.

5.2 Initialization

Before the USB Function Library can be used it has to be initialized first. This will be done by a call to **Usbf_Initialize**. To call this function the application designer has to provide the USB descriptors for the appropriate USB device. Refer to section 4.2 for more information about the USB descriptors.

After the call to **Usbf_Initialize** returned with a status of **USBF_STATUS_SUCCESS**, the USB Function Library is initialized successfully and ready to use. Now upper layers (e.g. a device class or the application itself) can be initialized. The USB communication can be started by a call to **Usbf_Enable**. Now the host should start the enumeration of the USB device.

The initialization functions of the USB Function Library are declared in the file **usbf_init.h**. The functions for the initialization of the device class are defined in the file **usbf_xxx_init.h** where **xxx** is an abbreviation for the appropriate device class.

5.3 Synchronization

The USB Function Library is not reentrant and has no synchronization primitives internally. All function calls must be synchronized external. This means that the application designer is responsible to synchronize all calls to the USB Function Library.

Only the function **Usbf_InterruptServiceRoutine** can be called without code synchronization.

In the debug version of the USB Function Library the code synchronization will be checked and an assert will fail if the code of the library will be re-entered.

The USB Function Library code does not block internally, all function calls return immediately.

The callback functions of the USB Function Library will always be called in a defined context. Within a callback function it is allowed to call some functions of the USB Function Library. Refer to the reference of the specified callback function to see in which context it will be called and which functions are allowed to call.

5.4 Callbacks

The Embedded USB Device Stack uses callbacks for communication with the application layer. This way it is possible to inform the user application about specific events from USB or from a device class. Callbacks are also used to return previously submitted buffers. Usually a buffer will be submitted by a user application and will remain pending until it is completely processed. These buffers will be completed by a call to the appropriate callback function.

Note that all callbacks are called from within the Embedded USB Device Stack. Mostly they are called in the context of **Usbf_ProcessInterrupt**. The execution in the Embedded USB De-

vice Stack is interrupted until the callback has returned. Therefore a callback should be processed in the application layer as soon as possible. Lengthy operations should be avoided. If you have to process or copy a data buffer you should put it in a queue and call your application from the firmware's main loop to process the queue or do lengthy operations deferred.

5.5 Interrupt Handling

The application can either register an interrupt service routine (ISR) and within this ISR call the function `Usbf_InterruptServiceRoutine` or call the function `Usbf_InterruptServiceRoutine` periodically within the main thread.

The function `Usbf_InterruptServiceRoutine` checks if an interrupt has been occurred which has to be handled by the USB Function Library. In this case it deasserts the interrupt line to avoid a permanent interrupt and returns with a value of `1`. A return value of `0` indicates that either no interrupt occurred or it was caused by a different device that use the same shared interrupt line.

If the function `Usbf_InterruptServiceRoutine` returns `1` the function `Usbf_ProcessInterrupt` has to be called in order to handle the interrupt. This call can be performed deferred in a different context.

Note that the application designer is responsible for code synchronization. Refer to section 5.3 for more information about the code synchronization in the USB Function Library.

Note that some USB hardware manufactures have build in racing conditions. This may require to call the function `Usbf_InterruptServiceRoutine` in polling mode with a time interval not larger a given limit. The limit typically is 5 ms.

Example for the interrupt handling in a simple main loop application:

Register an interrupt service routine (ISR) for the USB interrupt. In the ISR call the function `Usbf_InterruptServiceRoutine`. If it returns with a value of `1` set a global flag to indicate that the USB device has caused an interrupt which has to be handled. In the applications main loop check this flag periodically. If this flag is set, reset it and call `Usbf_ProcessInterrupt` to handle the interrupt. After the function `Usbf_ProcessInterrupt` returns, go on with the main loop.

All other functions of the USB Function Library (except for `Usbf_InterruptServiceRoutine`) has to be called in the context of `main()`. This ensures that the USB Function Library will not be re-entered and all function calls are synchronized.

The pseudo code for the interrupt handling in a simple main loop application is listed below:

```
static bool gIntFlag = false;

/* interrupt service routine */
void
UsbIsr() {
    /* check for USB interrupt, can it be handled by USB Function Library? */
    if ( 1 == Usbf_InterruptServiceRoutine() ) {
        gIntFlag = true;
    }
}

int
main() {
    ...
}
```

```

/* initialize USB Function Library */
if ( USBF_STATUS_SUCCESS != Usbf_Initialize() ) {
    return -1;
}

/* register USB interrupt service routine here */
...

/* enable USB device */
Usbf_Enable();

...

/* entering main loop */
for (;;) {

    ...

    /* check interrupt flag */
    if (gIntFlag) {
        /* reset flag */
        gIntFlag = false;
        /* process USB interrupt */
        Usbf_ProcessInterrupt();
    }

    ...
}
}

```

The functions for the interrupt handling are declared in the file `usbf_init.h`.

5.6 VBus Sensing

The sensing of VBus is essential for the USB Function Library. But not all device controllers provide the VBus sensing internally. Therefore the function `Usbf_CheckVBus` exists. This function has to be called periodically if the device controller does not support the VBus sensing internally. Refer to section 5.11 for more information about required device controller specific adaptations.

5.7 Plug and Play Handling

The application designer can register a device handler by a call to `Usbf_RegisterDeviceHandler`. The device handler callback function will be called every time a device related event occur. So the application designer can implement a specific behavior for the USB device depending on the occurred event.

Because of the multi-configuration support an endpoint can not be opened before the host has selected a configuration. Therefore data transfer should not be started before the device event handler receives the `USBF_DEV_EVENT_CONFIGURE` event.

Furthermore the events `USBF_DEV_EVENT_UNCONFIGURE`, `USBF_DEV_EVENT_RESET` and `USBF_DEV_EVENT_DEVICE_REMOVED` signal that all handles to opened endpoints became invalid because the endpoints were closed internally.

With a registered device handler it is possible for the application designer to implement some sort of Plug and Play handling for the USB device.

Refer to the function [T_Usbf_DeviceEvent_Callback](#) to get a list of supported device events.

5.8 Handling of Class and Vendor Requests

Only Interface and Endpoint are supported recipients for class/vendor requests. The `wIndex` field of the request must contain a valid number for the interface or endpoint respectively. Class/vendor requests containing other recipients will be stalled by the USB Function Library. Refer to [1] section 9.3 and 9.4 for more information on class/vendor requests.

5.9 Error Recovery

The USB Function Library implements an extended error recovery on bulk and interrupt endpoints. The host hardware repeats failed transactions on such endpoints up to three times. Then a Clear Feature Endpoint Halt request is send to the device.

The extended error recovery is based on a logical buffer size. Both, the device and the host software use buffers with this logical buffer size. The logical buffer size must be a multiple of the FIFO size.

If an error occurs the device is informed by the Clear Feature Endpoint Halt request. The host gets this information from the host controller. Both, the sender and the receiver of the data clear the FIFO and reset the data toggle bit. The extended error recovery is based on the following step. The sender starts to resubmit the complete logical buffer from the beginning. The receiver discards the data that have been received so far and fills the buffer from the beginning.

This method makes sure that no data is lost even if the data connection has a higher bit error rate.

Note that a small logical buffer reduces the performance but a larger logical buffer requires more memory on the device side.

It is allowed to send each amount of data with this method. A data block larger than the logical buffer is divided into logical buffers. A shorter data block is transferred in one logical buffer while the real transfer length is used. A logical buffer is completely transferred if it is filled completely or if a short packet is received.

5.10 Buffer Chain

The buffer descriptor of type [T_UsbfBufferDescriptor](#) defined for the USB Function Library can be used to chain multiple buffers together into a single buffer chain. Therefore the field `next` of a buffer descriptor has to point to the next valid buffer descriptor of type [T_UsbfBufferDescriptor](#). The field `next` of the last buffer descriptor in the buffer chain has to be set to `NULL`. If no buffer chain should be used the field `next` of the buffer descriptor has to be set to `NULL`.

Note: The buffer chain is only supported for sending of buffers, not receiving. Therefore the field `next` of a buffer descriptor used for receiving has to be set to `NULL`.

Buffers that are chained together are handled in the same way as a single buffer with the size that is the sum of all partial buffers. But only the fields of the first buffer descriptor in a buffer chain

will be changed. That means that on completion of a buffer chain, only the first buffer descriptor contains the valid status in the field **status**. Furthermore only the field **flags** of the first buffer descriptor will be recognized and changed. The field **bytesTransferred** is valid for each buffer descriptor and contains the valid number of bytes transferred for this buffer descriptor.

Refer to **Usbf_Read** and **Usbf_Write** or the appropriate submit functions of the used device class for detailed information of the fields which has to be set by the caller and which will be changed by the USB Function Library or device class.

5.11 Device Controller Specific Adaptation

The following sections contain information about the device controller specific adaptation process. This includes the listing of the required hardware abstraction layer functions as well as possible and necessary compile time settings for the specific device controller. Note that only compile time settings related to the specific device controller will be described. Refer to section 4.1.1 for more information about the general compile time settings for the USB Function Library and the device classes.

5.11.1 Customization for Atmel AT91SAM based devices

The Atmel AT91SAM based devices do not support VBus sensing and the SoftConnect feature internally. This has to be implemented externally by the hardware designer. For more information see section 2.6.

HAL Functions

The Atmel AT91SAM based devices use a 32-bit bus access. Therefore the following functions of the hardware abstraction layer (see section 8) have to be implemented:

- **UsbfHal_WriteReg32** (page 94)
- **UsbfHal_BurstWriteReg32** (page 95)
- **UsbfHal_ReadReg32** (page 100)
- **UsbfHal_BurstReadReg32** (page 101)
- **UsbfHal_IsVBusPresent** (page 104)
- **UsbfHal_SetSoftConnect** (page 105)

Compile Time Settings

No device controller specific compile time settings required.

5.11.2 Customization for Atmel AT91SAM3U based devices

The Atmel AT91SAM3U (Cortex M3 micro controller) use a 32bit access but do not support VBus sensing .

HAL Functions

The Atmel AT91SAM3U based devices use a 32-bit bus access. Therefore the following functions of the hardware abstraction layer (see section 8) have to be implemented:

- **UsbHal_WriteReg32** (page 94)
- **UsbHal_ReadReg32** (page 100)
- **UsbHal_IsVBusPresent** (page 104)

Compile Time Settings

The sum of all fifo sizes is the number of fifo's per endpoint multiplied with the maximum packet size per endpoint. This size can not exceed the maximum size of 4032 bytes. Because of this limitation dividing the maximum FIFO size on different endpoints is not easy and depends also from the usage of an data endpoint (designed for a high or low datarate).

- **USBF_HIGH_SPEED** - This has to be defined in the **config.h** to enable the device to act as a high speed USB device. If not defined the controller act as a full speed device.
- **PLT_DC_EP1_DOUBLE_BUFFERING** - This has to be defined in the platform config file **config.h**. Set to zero for a single buffered fifo and set to one for a double buffered fifo.
- **PLT_DC_EP2_DOUBLE_BUFFERING** - This has to be defined in the platform config file **config.h**. Set to zero for a single buffered fifo and set to one for a double buffered fifo.
- **PLT_DC_EP3_DOUBLE_BUFFERING** - This has to be defined in the platform config file **config.h**. Set to zero for a single buffered fifo and set to one for a double buffered fifo.
- **PLT_DC_EP4_DOUBLE_BUFFERING** - This has to be defined in the platform config file **config.h**. Set to zero for a single buffered fifo and set to one for a double buffered fifo.
- **PLT_DC_EP5_DOUBLE_BUFFERING** - This has to be defined in the platform config file **config.h**. Set to zero for a single buffered fifo and set to one for a double buffered fifo.
- **PLT_DC_EP6_DOUBLE_BUFFERING** - This has to be defined in the platform config file **config.h**. Set to zero for a single buffered fifo and set to one for a double buffered fifo.

5.11.3 Customization for Atmel AVR32 based devices

HAL Functions

The Atmel AVR32 based devices use a 32-bit bus access. Therefore the following functions of the hardware abstraction layer (see section 8) have to be implemented:

- [UsbHal_WriteReg32](#) (page 94)
- [UsbHal_BurstWriteReg32](#) (page 95)
- [UsbHal_ReadReg32](#) (page 100)
- [UsbHal_BurstReadReg32](#) (page 101)
- [UsbHal_FifoWrite](#) (page 102)
- [UsbHal_FifoRead](#) (page 103)

Compile Time Settings

The AVR32 family based controller AT32UC3A3 and AT32UC3A4 series contains an USB high and full speed device controller and needs the following defines config.h:

The sum of all fifo sizes is the number of fifo's (1=single buffered FIFO,2=double buffered FIFO) per data endpoint multiplied with the maximum FIFO size per endpoint. This size can not exceed the maximum size of 2304 bytes (without the control endpoint). The maximum FIFO size per endpoint must be setup in ascending order and can be different from maximum packet size that is defined in the interface descriptor. The FIFO size must be equal or greater than maximum packet size. This FIFO sizes are allowed for the define **DC_FIFO_SIZE_EPx** (x=endpoint number):**8, 16, 32,64,128,256,512,1024**. See also the demo application for more information. On not used data endpoint's set the value to **8**.

- **USBF_HIGH_SPEED** - This has to be defined in the **config.h** to enable the device to act as a high speed USB device. If not defined the controller act as a full speed device.
- **DC_FIFO_SIZE_EP1** - This has to be defined in the platform config file **config.h**. It is the maximum FIFO size, see above for more information.
- **DC_EP1_USE_DOUBLE_BUFFERING** - This has to be defined in the platform config file **config.h**. Set to zero for a single buffered fifo and set to one for a double buffered fifo.
- **DC_FIFO_SIZE_EP2** - This has to be defined in the platform config file **config.h**. It is the maximum FIFO size, see above for more information.
- **DC_EP2_USE_DOUBLE_BUFFERING** - This has to be defined in the platform config file **config.h**. Set to zero for a single buffered fifo and set to one for a double buffered fifo.
- **DC_FIFO_SIZE_EP3** - This has to be defined in the platform config file **config.h**. It is the maximum FIFO size, see above for more information.
- **DC_EP3_USE_DOUBLE_BUFFERING** - This has to be defined in the platform config file **config.h**. Set to zero for a single buffered fifo and set to one for a double buffered fifo.
- **DC_FIFO_SIZE_EP4** - This has to be defined in the platform config file **config.h**. It is the maximum FIFO size, see above for more information.
- **DC_EP4_USE_DOUBLE_BUFFERING** - This has to be defined in the platform config file **config.h**. Set to zero for a single buffered fifo and set to one for a double buffered fifo.

- **DC_FIFO_SIZE_EP5** - This has to be defined in the platform config file **config.h**. It is the maximum FIFO size, see above for more information.
- **DC_EP5_USE_DOUBLE_BUFFERING** - This has to be defined in the platform config file **config.h**. Set to zero for a single buffered fifo and set to one for a double buffered fifo.
- **DC_FIFO_SIZE_EP6** - This has to be defined in the platform config file **config.h**. It is the maximum FIFO size, see above for more information.
- **DC_EP6_USE_DOUBLE_BUFFERING** - This has to be defined in the platform config file **config.h**. Set to zero for a single buffered fifo and set to one for a double buffered fifo.
- **DC_FIFO_SIZE_EP7** - This has to be defined in the platform config file **config.h**. It is the maximum FIFO size, see above for more information.
- **DC_EP7_USE_DOUBLE_BUFFERING** - This has to be defined in the platform config file **config.h**. Set to zero for a single buffered fifo and set to one for a double buffered fifo.
- **DC_FIFO_SIZE_EP8** - This has to be defined in the platform config file **config.h**. It is the maximum FIFO size, see above for more information.
- **DC_EP8_USE_DOUBLE_BUFFERING** - This has to be defined in the platform config file **config.h**. Set to zero for a single buffered fifo and set to one for a double buffered fifo.

5.11.4 Customization for Philips/NXP ISP1582

HAL Functions

The Philips/NXP ISP1582 uses a 16-bit bus access. Therefore the following functions of the hardware abstraction layer (see section 8) have to be implemented:

- **UsbHal_WriteReg16** (page 92)
- **UsbHal_BurstWriteReg16** (page 93)
- **UsbHal_ReadReg16** (page 98)
- **UsbHal_BurstReadReg16** (page 99)
- **UsbHal_Delay** (page 106)

Note: The function **UsbHal_Delay** (page 106) will be called with the following values:

- **60** - This is the time the hardware requires after the endpoint index or control function register is written before these registers can be read again. Data corruption can occur if this period is neglected.
- **200** - This is the time the hardware requires for the buffer status register to get updated after writing to the FIFO. This is only applicable for double buffered endpoints.

These values are given by the manual and errata sheets of the ISP1582 but mainly depend on the used microcontroller. Therefore the application designer can adjust the timings within this function. It is also possible to define this function as an empty macro on slower microcontrollers where no wait cycles may be required.

Compile Time Settings

- **USBF_HIGH_SPEED** - This has to be defined in the file **usbf_config.h** to enable the ISP1582 to act as a high speed USB device. If not defined the ISP1582 act as a full speed device.

An endpoint configuration table has to be provided in the config file **usbf_config.h**. Refer to the **usbf_config.h** file of the demo applications for more information and a sample endpoint configuration table.

5.11.5 Customization for Philips/NXP LPC23xx

HAL Functions

The Philips/NXP LPC23xx series use 32-bit bus access. Therefore the following functions of the hardware abstraction layer (see section 8) have to be implemented:

- **UsbHal_WriteReg32** (page 94)
- **UsbHal_BurstWriteReg32** (page 95)
- **UsbHal_ReadReg32** (page 100)
- **UsbHal_BurstReadReg32** (page 101)
- **UsbHal_IsVBusPresent** (page 104)
- **UsbHal_SetSoftConnect** (page 105)

Compile Time Settings

No device controller specific compile time settings required.

5.11.6 Customization for Philips/NXP LPC24xx

HAL Functions

The Philips/NXP LPC24xx series use 32-bit bus access. Therefore the following functions of the hardware abstraction layer (see section 8) have to be implemented:

- **UsbHal_WriteReg32** (page 94)

- [UsbHal_BurstWriteReg32](#) (page 95)
- [UsbHal_ReadReg32](#) (page 100)
- [UsbHal_BurstReadReg32](#) (page 101)
- [UsbHal_IsVBusPresent](#) (page 104)
- [UsbHal_SetSoftConnect](#) (page 105)

Compile Time Settings

No device controller specific compile time settings required.

5.11.7 Customization for Philips/NXP LPC17xx

HAL Functions

The Philips/NXP LPC17xx series use 32-bit bus access. Therefore the following functions of the hardware abstraction layer (see section 8) have to be implemented:

- [UsbHal_WriteReg32](#) (page 94)
- [UsbHal_BurstWriteReg32](#) (page 95)
- [UsbHal_ReadReg32](#) (page 100)
- [UsbHal_BurstReadReg32](#) (page 101)
- [UsbHal_IsVBusPresent](#) (page 104)
- [UsbHal_SetSoftConnect](#) (page 105)

Compile Time Settings

No device controller specific compile time settings required.

5.11.8 Customization for Renesas H8SX/1653, H8SX/1663 Group

The internal VBus sensing is only available if the PullUp resistor on the D+ line is permanently connected. But this disables the SoftConnect feature. Because the SoftConnect feature is required it is recommend to use a GPIO pin for external VBus sensing. This way the VBus sensing and the SoftConnect feature is available. The GPIO pin representing the VBus state has to be checked in the function **UsbHal_IsVBusPresent**. The application should call the function

Usbf_CheckVBus periodically so the USB Function Library checks if VBus is present. For more information see section 2.6.

HAL Functions

The Renesas H8SX/1653 group use 8-bit register access. Therefore the following functions of the hardware abstraction layer (see section 8) have to be implemented:

- **UsbfHal_WriteReg8** (page 90)
- **UsbfHal_ReadReg8** (page 96)
- **UsbfHal_BurstWriteReg8** (page 91)
- **UsbfHal_BurstReadReg8** (page 97)
- **UsbfHal_IsVBusPresent** (page 104)

Compile Time Settings

- **USBF_DC_AUTO_HANDLES_SET_CONFIGURATION** - This has to be defined in the config file **usbf_config.h** because the USB SetConfiguration request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_SET_INTERFACE** - This has to be defined in the config file **usbf_config.h** because the USB SetInterface request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_SET_ADDRESS** - This has to be defined in the config file **usbf_config.h** because the USB SetAddress request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_SET_FEATURE** - This has to be defined in the config file **usbf_config.h** because the USB SetFeature request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_CLEAR_FEATURE** - This has to be defined in the config file **usbf_config.h** because the USB ClearFeature request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_GET_CONFIGURATION** - This has to be defined in the config file **usbf_config.h** because the USB GetConfiguration request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_GET_INTERFACE** - This has to be defined in the config file **usbf_config.h** because the USB GetInterface request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_GET_STATUS** - This has to be defined in the config file **usbf_config.h** because the USB GetStatus request is handled automatically by this device controller.

5.11.9 Customization for Renesas H8SX/1668 Group

The internal VBus sensing is only available if the PullUp resistor on the D+ line is permanently connected. But this disables the SoftConnect feature. Because the SoftConnect feature is required it is recommend to use a GPIO pin for external VBus sensing. This way the VBus sensing and the SoftConnect feature is available. The GPIO pin representing the VBus state has to be checked in the function `UsbHal_IsVBusPresent`. The application should call the function `UsbHal_CheckVBus` periodically so the USB Function Library checks if VBus is present. For more information see section 2.6.

HAL Functions

The Renesas H8SX/1668 group use 8-bit register access. Therefore the following functions of the hardware abstraction layer (see section 8) have to be implemented:

- `UsbHal_WriteReg8` (page 90)
- `UsbHal_ReadReg8` (page 96)
- `UsbHal_BurstWriteReg8` (page 91)
- `UsbHal_BurstReadReg8` (page 97)
- `UsbHal_IsVBusPresent` (page 104)

Compile Time Settings

- `USBF_DC_AUTO_HANDLES_SET_CONFIGURATION` - This has to be defined in the config file `usbf_config.h` because the USB SetConfiguration request is handled automatically by this device controller.
- `USBF_DC_AUTO_HANDLES_SET_INTERFACE` - This has to be defined in the config file `usbf_config.h` because the USB SetInterface request is handled automatically by this device controller.
- `USBF_DC_AUTO_HANDLES_SET_ADDRESS` - This has to be defined in the config file `usbf_config.h` because the USB SetAddress request is handled automatically by this device controller.
- `USBF_DC_AUTO_HANDLES_SET_FEATURE` - This has to be defined in the config file `usbf_config.h` because the USB SetFeature request is handled automatically by this device controller.
- `USBF_DC_AUTO_HANDLES_CLEAR_FEATURE` - This has to be defined in the config file `usbf_config.h` because the USB ClearFeature request is handled automatically by this device controller.
- `USBF_DC_AUTO_HANDLES_GET_CONFIGURATION` - This has to be defined in the config file `usbf_config.h` because the USB GetConfiguration request is handled automatically by this device controller.

- **USBF_DC_AUTO_HANDLES_GET_INTERFACE** - This has to be defined in the config file **usb_config.h** because the USB GetInterface request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_GET_STATUS** - This has to be defined in the config file **usb_config.h** because the USB GetStatus request is handled automatically by this device controller.

5.11.10 Customization for Renesas H8S/2472 Group

The internal VBus sensing is only available if the PullUp resistor on the D+ line is permanently connected. But this disables the SoftConnect feature. Because the SoftConnect feature is required it is recommend to use a GPIO pin for external VBus sensing. This way the VBus sensing and the SoftConnect feature is available. The GPIO pin representing the VBus state has to be checked in the function **UsbHal_IsVBusPresent**. The application should call the function **Usb_CheckVBus** periodically so the USB Function Library checks if VBus is present. For more information see section 2.6.

HAL Functions

The Renesas H8S/2472 group use 8-bit register access. Therefore the following functions of the hardware abstraction layer (see section 8) have to be implemented:

- **UsbHal_WriteReg8** (page 90)
- **UsbHal_ReadReg8** (page 96)
- **UsbHal_BurstWriteReg8** (page 91)
- **UsbHal_BurstReadReg8** (page 97)
- **UsbHal_IsVBusPresent** (page 104)

Compile Time Settings

- **H8S2472_DEVICE_CONTROLLER** - This has to be defined in the file **usb_config.h** in order to use the H8S/2472 device controller.
- **USBF_DC_AUTO_HANDLES_SET_CONFIGURATION** - This has to be defined in the file **usb_config.h** because the USB SetConfiguration request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_SET_INTERFACE** - This has to be defined in the config file **usb_config.h** because the USB SetInterface request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_SET_ADDRESS** - This has to be defined in the config file **usb_config.h** because the USB SetAddress request is handled automatically by this device controller.

- **USBF_DC_AUTO_HANDLES_SET_FEATURE** - This has to be defined in the config file `usbf_config.h` because the USB SetFeature request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_CLEAR_FEATURE** - This has to be defined in the config file `usbf_config.h` because the USB ClearFeature request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_GET_CONFIGURATION** - This has to be defined in the config file `usbf_config.h` because the USB GetConfiguration request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_GET_INTERFACE** - This has to be defined in the config file `usbf_config.h` because the USB GetInterface request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_GET_STATUS** - This has to be defined in the config file `usbf_config.h` because the USB GetStatus request is handled automatically by this device controller.

5.11.11 Customization for Renesas H8S/2215 Group

HAL Functions

The Renesas H8S/2215 group use 8-bit and 32-bit register access. Therefore the following functions of the hardware abstraction layer (see section 8) have to be implemented:

- [UsbfHal_WriteReg8](#) (page 90)
- [UsbfHal_WriteReg32](#) (page 94)
- [UsbfHal_ReadReg8](#) (page 96)
- [UsbfHal_ReadReg32](#) (page 100)
- [UsbfHal_BurstWriteReg8](#) (page 91)
- [UsbfHal_BurstWriteReg32](#) (page 95)
- [UsbfHal_BurstReadReg8](#) (page 97)
- [UsbfHal_BurstReadReg32](#) (page 101)
- [UsbfHal_SetSoftConnect](#) (page 105)

Compile Time Settings

- **USBF_DC_AUTO_HANDLES_CLEAR_FEATURE** - This has to be defined in the config file `usbf_config.h` because the USB ClearFeature request is handled automatically by this device controller.

- **USBF_DC_AUTO_HANDLES_GET_CONFIGURATION** - This has to be defined in the config file **usbf_config.h** because the USB GetConfiguration request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_GET_INTERFACE** - This has to be defined in the config file **usbf_config.h** because the USB GetInterface request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_GET_STATUS** - This has to be defined in the config file **usbf_config.h** because the USB GetStatus request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_SET_ADDRESS** - This has to be defined in the config file **usbf_config.h** because the USB SetAddress request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_SET_CONFIGURATION** - This has to be defined in the config file **usbf_config.h** because the USB SetConfiguration request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_SET_FEATURE** - This has to be defined in the config file **usbf_config.h** because the USB SetFeature request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_SET_INTERFACE** - This has to be defined in the config file **usbf_config.h** because the USB SetInterface request is handled automatically by this device controller.

5.11.12 Customization for Renesas M16C6C Group

The internal VBus sensing is only available if the PullUp resistor on the D+ line is permanently connected. But this disables the SoftConnect feature. Because the SoftConnect feature is required it is recommend to use a GPIO pin for external VBus sensing. This way the VBus sensing and the SoftConnect feature is available. The GPIO pin representing the VBus state has to be checked in the function **UsbfHal_IsVBusPresent**. The application should call the function **Usbf_CheckVBus** periodically so the USB Function Library checks if VBus is present. For more information see section 2.6.

HAL Functions

The Renesas M16C6C group use 8-bit register access. Therefore the following functions of the hardware abstraction layer (see section 8) have to be implemented:

- **UsbfHal_WriteReg8** (page 90)
- **UsbfHal_ReadReg8** (page 96)
- **UsbfHal_BurstWriteReg8** (page 91)
- **UsbfHal_BurstReadReg8** (page 97)

- [UsbHal_IsVBusPresent](#) (page 104)

Compile Time Settings

- **USBF_DC_AUTO_HANDLES_CLEAR_FEATURE** - This has to be defined in the config file `usbf_config.h` because the USB ClearFeature request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_GET_CONFIGURATION** - This has to be defined in the config file `usbf_config.h` because the USB GetConfiguration request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_GET_INTERFACE** - This has to be defined in the config file `usbf_config.h` because the USB GetInterface request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_GET_STATUS** - This has to be defined in the config file `usbf_config.h` because the USB GetStatus request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_SET_ADDRESS** - This has to be defined in the config file `usbf_config.h` because the USB SetAddress request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_SET_CONFIGURATION** - This has to be defined in the config file `usbf_config.h` because the USB SetConfiguration request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_SET_FEATURE** - This has to be defined in the config file `usbf_config.h` because the USB SetFeature request is handled automatically by this device controller.
- **USBF_DC_AUTO_HANDLES_SET_INTERFACE** - This has to be defined in the config file `usbf_config.h` because the USB SetInterface request is handled automatically by this device controller.

5.11.13 Customization for STMicroelectronics STR91x

HAL Functions

The STMicroelectronics STR91x series use 32-bit bus access. Therefore the following functions of the hardware abstraction layer (see section 8) have to be implemented:

- [UsbHal_WriteReg32](#) (page 94)
- [UsbHal_ReadReg32](#) (page 100)
- [UsbHal_FifoWrite](#) (page 102)

- **UsbHal_FifoRead** (page 103)
- **UsbHal_IsVBusPresent** (page 104)
- **UsbHal_SetSoftConnect** (page 105)

Compile Time Settings

An endpoint configuration table has to be provided in the config file **usb_config.h**. Refer to the **usb_config.h** file of the demo applications for more information and a sample endpoint configuration table.

6 USB Function Library Reference

6.1 Initialization

The following functions are used for the initialization of the USB Function Library. They are defined in the file `usbf_init.h`.

Usbf_Initialize

This function has to be called at first to initialize the USB Function Library.

Definition

```
T_UsbfStatus
Usbf_Initialize(
    const T_UsbfDescriptors* descriptors
);
```

Parameter

descriptors

This field points to a structure containing the USB descriptors. The USB descriptors must be designed compliant to the USB specification. The content of the descriptors must agree with the configuration data. Descriptors can be located either in RAM or ROM.

The storage for descriptors referenced by this structure must be persistent. The data structure pointed to by **descriptors** has to be valid during this call only.

Return Value

The function returns one of the following status codes:

USBF_STATUS_SUCCESS: - If the USB Function Library has been initialized successfully.

USBF_STATUS_INVALID_PARAM: - If one of the given descriptors is invalid.

Comments

This function must be called one time after power on reset to initialize the library. It is recommended to check the return value before enabling the USB device.

See Also

[T_UsbfDescriptors](#) (page 61)

[Usbf_Enable](#) (page 43)

Usbf_Enable

This function enables the USB device.

Definition

```
void  
Usbf_Enable( );
```

Comments

Enables the USB device by switching on the 1,5k resistor on the D+ line to 3.3 Volt if Vusb 4..5V on the USB connector is detected. This function can be called independently of the current connection state. The library takes care of the connection state and enables the function immediately if the function is connected to the host.

After this function is called the enumeration process on the host should start if the device is connected.

The USB Function Library has to be initialized by a call to **Usbf_Initialize** before this function can be called.

See Also

Usbf_Initialize (page 42)

Usbf_Disable (page 44)

Usbf_Disable

This function disables the USB device.

Definition

```
void  
Usbf_Disable( );
```

Comments

This function causes a virtual unplug of the device by switching off the 1.5k resistor from D+. All submitted buffers will be completed with the cancel status. All USB interrupts inclusive the resume interrupt will be disabled. If the embedded application calls this function it must wait for at least one second before the USB interface can be enabled again with the function **Usbf_Enable**.

See Also

Usbf_Enable (page 43)

Usbf_RegisterDeviceHandler

This function is used to register the callback functions for the device.

Definition

```
void  
Usbf_RegisterDeviceHandler(  
    T_UsbfDeviceHandler* deviceHandler  
);
```

Parameter

deviceHandler

This field points to a structure containing the callback function pointers. Each pointer is optional and can be NULL. See **T_UsbfDeviceHandler** for details. The storage for this data structure must be permanent.

Comments

This function can be called for each interface to register the callback functions. The storage for the **deviceHandler** must be provided by the caller and it must be permanent until the function **Usbf_UnregisterDeviceHandler** is called.

Refer to section 5.4 for more information about callback functions.

See Also

T_UsbfDeviceHandler (page 63),
Usbf_Initialize (page 42),
Usbf_UnregisterDeviceHandler (page 46)

Usbf_UnregisterDeviceHandler

This function is used to unregister the callback functions for the device.

Definition

```
void  
Usbf_UnregisterDeviceHandler(  
    T_UsbfDeviceHandler* deviceHandler  
);
```

Parameter

deviceHandler

This field points to a structure containing the callback function pointers.

Comments

After this function returns no callback function is called.

See Also

[T_UsbfDeviceHandler](#) (page 63),
[Usbf_Initialize](#) (page 42),
[Usbf_RegisterDeviceHandler](#) (page 45)

Usbf_InterruptServiceRoutine

This function checks if the interrupt was caused by the USB device and if it can be handled by the USB Function Library.

Definition

```
int  
Usbf_InterruptServiceRoutine( );
```

Return Value

The function returns 1 if the interrupt was caused by the USB function device. In this case the USB interrupts have been disabled to deassert the interrupt line.

The function returns 0 if the interrupt was not caused by the USB function device but caused by another device that shares the same interrupt line.

Comments

The main program implements an interrupt service routine for the interrupt vector the USB function controller is attached to. The main program has to call **Usbf_ProcessInterrupt** subsequently to process the pending USB library interrupt events if the interrupt was caused by the USB function device.

Note that the main program should never touch any of the USB function registers.

See Also

Usbf_ProcessInterrupt (page 48)

Usbf_ProcessInterrupt

This function processes outstanding USB function interrupt events.

Definition

```
void  
Usbf_ProcessInterrupt ( ) ;
```

Comments

This function has to be called by the main program when **Usbf_InterruptServiceRoutine** indicates (by a return value of 1) that there are outstanding interrupt events for the USB function. **Usbf_ProcessInterrupt** processes all outstanding events and clears the interrupt status.

Usually the callbacks issued by the library will run in the context of **Usbf_ProcessInterrupt**.

See Also

Usbf_InterruptServiceRoutine (page 47)

Usbf_CheckVBus

This function checks for the presence of VBus.

Definition

```
void  
Usbf_CheckVBus ( ) ;
```

Comments

This function checks for the presence of VBus and has to be called periodically if VBus sensing is not possible from within the USB function registers. Refer to the section [2.7](#) in the manual to see if this function has to be called for your specific device controller.

Usbf_SetTraceMask

This function sets an internal trace mask which filters trace messages produced by the USB function library.

Definition

```
void  
Usbf_SetTraceMask(  
    unsigned int mask  
);
```

Parameter

mask

Specifies the new trace mask to be set.

Comments

The USB Function Library trace mask is an internal global 32-bit integer variable. A specific bit position within that variable is assigned to every particular trace message built into USB Function Library. The message will be outputted if the corresponding bit is set and will be suppressed if the corresponding bit is cleared. This way, the current value of the trace mask determines the amount of trace messages produced by the USB Function Library.

Bit positions of trace mask are assigned as described below.

DBG_ERR

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

DBG_WARN

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

DBG_INFO

Informational messages.

DBG_FUNC

Print the function names of the USB Function Library.

DBG_EP0

Informational messages from the control endpoint.

DBG_EP

Informational messages from data endpoints.

DBG_DUMP_EP0

Dumps data from the control endpoint.

DBG_DUMP_EP

Dumps data from data endpoints.

DBG_INT

Informational messages about the interrupt handling.

DBG_EPLIST

Informational messages about the endpoint list used internally.

By default, the bits **DBG_ERR** and **DBG_WARN** are set and all other bits are cleared in the USB Function Library trace mask.

Note that the **DBG_XXX** constants specify a bit position and not the corresponding mask. Use the **DBG_BIT_MASK** macro to create the corresponding mask for an individual bit position.

Example:

```
Usbf_SetTraceMask(  
DBG_BIT_MASK(DBG_ERR) | DBG_BIT_MASK(DBG_WARN)  
| DBG_BIT_MASK(DBG_INFO)  
);
```

Trace support can be enabled/disabled at compile time. If **DBG** has a non zero value then the trace support is enabled. If trace support is disabled then a call to **Usbf_SetTraceMask** has no effect.

6.2 Callbacks

T_Usbf_DeviceEvent_Callback

This function will be called if a device specific event has been detected.

Definition

```
void  
T_Usbf_DeviceEvent_Callback(  
    void* deviceContext,  
    unsigned int event,  
    unsigned int param1  
);
```

Parameters

deviceContext

This parameter contains the field **deviceContext** of the data structure **T_UsbfDeviceHandler** which was passed to the function **Usbf_RegisterDeviceHandler**.

event

This parameter indicates one of the following events:

- **USBF_DEV_EVENT_RESET** - A USB bus reset has been detected. All USB specific actions are handled by the library. Pending data requests are canceled.
- **USBF_DEV_EVENT_SUSPEND** - A USB suspend signal has been detected. If the device is bus powered the embedded application should reduce the required current to 2.5 mA. It should stop the clock and enable the static interrupt for wakeup. If the device is self powered the embedded application has to decide if the clock should be stopped. The user is responsible for processor specific wakeup conditions (e.g. enable wakeup after stopping the CPU clock) and reducing the power consume of bus powered devices.
- **USBF_DEV_EVENT_RESUME** - A resume signal has been detected. If the clock was turned off it must be re-enabled. Depending on the used USB function, the user must enable all processor specific functions that have been disabled in the suspend state (e.g. low power functions and PLLs).
- **USBF_DEV_EVENT_CONFIGURE** - The device has been configured. The index of the configuration descriptor is passed in **param1**. This is not the bConfiguration value of the configuration descriptor. It is the zero based index of the configuration descriptor itself. After this event the data transfer may be started.
- **USBF_DEV_EVENT_UNCONFIGURE** - The device has been unconfigured. Pending requests are canceled and all endpoints are closed.
- **USBF_DEV_EVENT_ENABLE_REMOTE_WAKEUP** - The host enabled the remote wakeup feature for the device.

- **USBF_DEV_EVENT_DISABLE_REMOTE_WAKEUP** - The host disabled the remote wakeup feature for the device.
- **USBF_DEV_EVENT_SELECT_INTERFACE** - The host selected a new alternate setting for an interface. The LSB (bit 0-7) of **param1** contains the interface number and the MSB (bit 8-15) contains the new alternate setting. Pending requests on an endpoint which is reconfigured are canceled. The application must open the reconfigured endpoints and start the data transfer again.
- **USBF_DEV_EVENT_DEVICE_CONNECTED** - The host was turned on or the device has been connected to a host.
- **USBF_DEV_EVENT_DEVICE_REMOVED** - The host was turned off or the device has been disconnected from the host.

param1

Contains additional information for some events. Refer to the event description for more information. For all other events this parameter is 0.

Comments

After the **USBF_DEV_EVENT_CONFIGURE** event has been received the embedded application can start the data endpoint transfer. If one of the events **USBF_DEV_EVENT_RESET**, **USBF_DEV_EVENT_DEVICE_REMOVED** or **USBF_DEV_EVENT_UNCONFIGURE** has been received the function is not configured. All endpoints are closed. The embedded application must stop the data transfer.

This function is called in the context of:

- **Usbf_ProcessInterrupt.**
- **Usbf_CheckVBus.**

The following functions can be called from within this callback:

- **UsbfCdcAcm_CreateInstance,**
- **UsbfCdcAcm_DeleteInstance,**
- **UsbfCdcAcm_ActivateInstance,**
- **UsbfCdcAcm_DeactivateInstance,**
- **UsbfCdcAcm_RegisterCallbacks.**
- **UsbfCdcAcm_SetSerialState.**
- **UsbfCdcAcm_GetSerialLineCoding.**
- **UsbfCdcAcm_SubmitTxBuffer.**
- **UsbfCdcAcm_SubmitRxBuffer.**
- **UsbfCdcAcm_AbortTx.**
- **UsbfCdcAcm_AbortRx.**

Refer to section 5.4 for more information about callback functions.

See Also

Usbf_ProcessInterrupt (page 48)

Usbf_RegisterDeviceHandler (page 45)

T_Usbf_StartOfFrame_Callback

This function is called if a SOF (StartOfFrame) interrupt has been received.

Definition

```
void  
T_Usbf_StartOfFrame_Callback(  
    void* deviceContext,  
    T_UINT16 frameNumber  
);
```

Parameters

deviceContext

This parameter contains the field **deviceContext** of the data structure **T_UsbfDeviceHandler** that was passed to the function **Usbf_RegisterDeviceHandler**.

frameNumber

This field contains the current frame number. It is typically a 16 bit number if provided by the hardware. The last 11 bits are equal to the frame number on the bus.

Comments

If the application designer registers the StartOfFrame callback in the USB Function Library, the StartOfFrame interrupt will be enabled. This may cause additional CPU load. Therefore this callback should only be registered if it is required by the embedded application.

This function is called in the context of:

- **Usbf_ProcessInterrupt**.

Note: The implementation for this callback function is optional. Therefore the appropriate function pointer in the **T_UsbfDeviceHandler** structure for this callback function can be **NULL**.

Refer to section 5.4 for more information about callback functions.

See Also

T_UsbfDeviceHandler (page 63)

Usbf_RegisterDeviceHandler (page 45)

Usbf_ProcessInterrupt (page 48)

6.3 Structures

T_UsbfBufferDescriptor

The **T_UsbfBufferDescriptor** structure contains information about a data buffer used within the USB Function Library for reading and writing data.

Definition

```
typedef struct tag_UsbfBufferDescriptor{
    DLIST listEntry;
    T_UINT8* dataPtr;
    T_UINT16 dataSize;
    T_UINT16 bytesTransferred;
    T_UINT16 flags;
    T_UINT16 status;
    T_UsbfBufferDescriptor* next;
    void* usbContext;
} T_UsbfBufferDescriptor;
```

Members

listEntry

This field can be used by the current owner of the buffer descriptor to manage multiple buffer descriptors in a double linked list.

dataPtr

This member points to the beginning of valid data within the buffer. This field is set by the creator of the buffer and will be used by the USB Function Library to determine where the data should be read from or be written to. The value of **dataPtr** will not be changed by the USB Function Library.

The pointer can specify a byte address. No assumptions on the alignment are made inside the USB Function Library. But the data processing inside the USB Function Library will be more efficient if the buffer address is aligned to a 4 Byte boundary.

dataSize

Contains the size of the data buffer in bytes. This field is set by the creator of the buffer and will be used by the USB Function Library to determine the size of the buffer used for data processing. The value of **dataSize** will not be changed by the USB Function Library.

bytesTransferred

Contains the number of valid bytes in the buffer. This field is set by the USB Function Library to communicate the number of processed bytes to the caller.

flags

This member contains 0, **USBF_SEND_SHORT_PACKET**, **USBF_DWORD_PADDING** or **USBF_ISO_FLOW_FLAG**. The flag **USBF_SEND_SHORT_PACKET** can be used with the function **Usbf_Write**. If the flag is set the library completes the request with a short packet.

The library checks if an additional zero length packet is required. On return this flag is cleared. When the buffer concatenation with `next` is used, the flag must be set in the first USB buffer descriptor.

The flag **USBF_DWORD_PADDING** can be used with the function **Usbf_Write**. It forces the library to append at the end of the buffer values with 0x00 so that the total size of the buffer can be divided by 4 without rest. E.g. a string "0x01 0x02 0x03 0x04" is send with this flag as "0x01 0x02 0x03 0x04". A string "0x01 0x02 0x03 0x04 0x05" with size 5 is send as "0x01 0x02 0x03 0x04 0x05 0x00 0x00 0x00". When the buffer concatenation with `next` is used, this flag must be set in each part of the descriptor that should be padded.

The flag **USBF_ISO_FLOW_FLAG** is an additional status flag. The status is a hint that the USB host want to read isochronous data but no data are available at the time or the USB host send isochronous data to the device but the device must discard this data because of a full FIFO. Also if the returned buffer status is **USBF_STATUS_SUCCESS** the flag should be checked.

status

This field contains the status of the operation. It will be set by the USB Function Library to one of the following values:

USBF_STATUS_SUCCESS: - The operation was completed successfully.

USBF_STATUS_CANCELED: - The host has sent a Unconfigure or a Reset or the user aborted the buffer with **Usbf_Close** or **Usbf_Abort**.

USBF_STATUS_BUFFER_OVERFLOW: - A buffer overflow happened during a **Usbf_Read** operation. The buffer size was too small or not a multiple of the FIFO size. Electrical noise of the signals can cause this error on some USB interfaces.

When the buffer concatenation with `next` is used, the status of the complete chain is set in the first buffer.

next

This field can be used to build a buffer chain. Refer to section 5.10 for more information about using a buffer chain. If no buffer chain is used this parameter has to be set to **NULL**.

usbContext

This context pointer is for internal use within the USB Function Library.

Comments

This structure can be used to build a chain of buffers. Refer to section 5.10 for more information about using a buffer chain.

This structure has to be provided by the embedded application for the functions **UsbfCdcAcm_SubmitRxBuffer** and **UsbfCdcAcm_SubmitTxBuffer** .

Refer to **UsbfCdcAcm_SubmitRxBuffer** and **UsbfCdcAcm_SubmitTxBuffer** for detailed information of the fields which have to be set by the user and which will be set by the CDC/ACM device class.

See Also

UsbCdcAcm_SubmitRxBuffer (page 79)

UsbCdcAcm_SubmitTxBuffer (page 77)

T_UsbCdcAcm_TxBufferCompletion_Callback (page 87)

T_UsbCdcAcm_RxBufferCompletion_Callback (page 88)

T_UsbfDescriptorRecord

This structure contains information about a USB descriptor.

Definition

```
typedef struct tag_UsbfDescriptorRecord{
    unsigned int size;
    const T_UINT8* descriptor;
} T_UsbfDescriptorRecord;
```

Members

size

This field contains the size of the USB descriptor.

descriptor

This parameter points to the USB descriptor.

See Also

[T_UsbfDescriptors](#) (page 61)

T_UsbfStringDescriptorRecord

This structure contains information about a USB string descriptor.

Definition

```
typedef struct tag_UsbfStringDescriptorRecord{
    unsigned int index;
    unsigned int size;
    const T_UINT8* descriptor;
} T_UsbfStringDescriptorRecord;
```

Members

index

This field contains the index of the USB string descriptor.

size

This field contains the size of the USB string descriptor.

descriptor

This parameter points to the USB string descriptor.

See Also

[T_UsbfDescriptors](#) (page 61)

T_UsbfDescriptors

The **T_UsbfDescriptors** structure contains all necessary USB descriptors.

Definition

```
typedef struct tag_UsbfDescriptors{
    void T_UINT8* deviceDescriptor;
    unsigned int numberOfConfigurationDescriptors;
    const T_UsbfDescriptorRecord* configurationDescriptor;
    unsigned int numberOfStringDescriptors;
    const T_UsbfStringDescriptorRecord* stringDescriptors;
    const T_UINT8* deviceDescriptorHighSpeed;
    const T_UINT8* deviceQualifierFullSpeed;
    const T_UINT8* deviceQualifierHighSpeed;
    unsigned int numberOfConfigurationDescriptorsHighSpeed;
    const T_UsbfDescriptorRecord*
        configurationDescriptorsHighSpeed;
} T_UsbfDescriptors;
```

Members

deviceDescriptor

This field contains a pointer to the device descriptor.

numberOfConfigurationDescriptors

This field contains the number of available configuration descriptors.

configurationDescriptor

This field points to the first entry of an array of type **T_UsbfDescriptorRecord** containing the available configuration descriptors. The storage of this array must be provided by the caller and must be persistent for the complete runtime.

Each configuration descriptor contains the complete description of the interface and endpoint layout. Each descriptor consists of one configuration descriptor and all related interface, class and endpoint descriptors. The correctness of the `wTotalLength` field in the configuration descriptor is very important. Most C compilers do not allow to calculate the size of this structure in the structure definition with `sizeof()`. For that reason the library overwrites this field of the descriptor with the value passed in the field `size` of the **T_UsbfDescriptorRecord** structure. The caller has to provide the correct size of the configuration descriptor in the field `size` of the **T_UsbfDescriptorRecord** structure. This size can be easily calculated with `sizeof()`. Invalid descriptors will cause incorrect behavior of the library without returning an error.

numberOfStringDescriptors

This member contains the number of available string descriptors for this device.

stringDescriptors

This field points to the first entry of an array of type **T_UsbfStringDescriptorRecord**

containing the available string descriptors. The storage for this array must be provided by the caller and must be persistent for the complete runtime.

Each string descriptor starts with a length field (one byte) and a type field (one byte). The length field describes the size of the complete descriptor in bytes. The library overwrites this field with the value **size** of the **T_UsbfStringDescriptorRecord** structure. This size can be easily calculated with **sizeof()**.

All characters have to be given in UNICODE format. This means the size of each character is two bytes. The string is not zero terminated. The string descriptors are valid for full and high speed. If different strings are required define strings for full and high speed and use different index values in the speed related descriptors. The string descriptor on index 0 contains the language ID or a list of language IDs.

deviceDescriptorHighSpeed

This field is only required if **USBF_HIGH_SPEED** is defined. It contains the device descriptor for operation in high speed mode.

deviceQualifierFullSpeed

This field is only required if **USBF_HIGH_SPEED** is defined. It contains the device qualifier descriptor that is returned in high speed operation. It describes the capabilities for full speed.

deviceQualifierHighSpeed

This field is only required if **USBF_HIGH_SPEED** is defined. It contains the device qualifier descriptor that is returned in full speed operation. It describes the capabilities for high speed.

numberOfConfigurationDescriptorsHighSpeed

This field is only required if **USBF_HIGH_SPEED** is defined. It contains the number of high speed configurations.

configurationDescriptorsHighSpeed

This field is only required if **USBF_HIGH_SPEED** is defined. It contains an array of pointers. Each pointer is directed to a high speed configuration descriptor. The storage for the array and the descriptors must be provided by the caller and must be persistent for the complete runtime. See **configurationDescriptor** for details.

Comments

The storage for all descriptors must be provided by the caller and must be persistent for the complete runtime. The descriptors can be stored in Flash or RAM memory. The descriptors must be defined correctly and compliant to the USB specification. Otherwise the enumeration on the host can fail or the library may stop working. For performance reasons the descriptor parsing is not error tolerant.

See Also

[Usbf_Initialize](#) (page 42)

[T_UsbfDescriptorRecord](#) (page 59)

[T_UsbfStringDescriptorRecord](#) (page 60)

T_UsbfDeviceHandler

The T_UsbfDeviceHandler structure contains the pointers for the device related callback functions.

Definition

```
typedef struct tag_UsbfDeviceHandler{
    DLIST listEntry;
    void* deviceContext;
    T_Usbf_DeviceEvent_Callback* deviceEvent;
    T_Usbf_StartOfFrame_Callback* startOfFrame;
} T_UsbfDeviceHandler;
```

Members

listEntry

This field is used by the library internally.

deviceContext

This field contains the device context. It is passed unchanged to all callback functions as the first parameter. The application should use this pointer to distinguish between different instances.

deviceEvent

This field points to a function which will be called if a device related event occurs. This callback function is optional. Therefore this field can be **NULL**.

startOfFrame

This field points to a function which will be called if a start of frame event occurs. This callback function is optional. Therefore this field can be **NULL**. If different from **NULL** the library enables the SOF interrupt and each USB frame generates an interrupt.

Comments

Each function pointer has to contain a valid function address or **NULL**. The embedded application can pass a NULL pointer to a callback function which is not required.

The storage for this data structure must be permanent until the function **Usbf_UnregisterDeviceHandler** is called.

See Also

Usbf_Initialize (page 42)

Usbf_RegisterDeviceHandler (page 45)

Usbf_UnregisterDeviceHandler (page 46)

6.4 Status Codes

USBF_STATUS_SUCCESS

The operation completed successfully.

USBF_STATUS_CANCELED

The operation was canceled by the library. Refer to the appropriate function description to get more information why this error occurred.

USBF_STATUS_BUFFER_OVERFLOW

Either the buffer size provided by the user was too small to retrieve all data or the provided buffer size was not a multiple of the FIFO size. In both cases loss of data has been occurred.

USBF_STATUS_INVALID_PARAM

At least one of the parameters passed to the function is invalid.

USBF_STATUS_PENDING

The request to read from a buffer or to write to a buffer is pending. The completion function will be called later if the processing of the buffer has been finished.

USBF_STATUS_FIFO_FULL

No data could be written because the FIFO is full.

USBF_STATUS_FIFO_EMPTY

No data could be read because the FIFO is empty.

USBF_STATUS_ENDPOINT_STALLED

The specified endpoint is stalled.

USBF_STATUS_NOT_OPENED

The specified endpoint is not opened.

USBF_STATUS_HW_ACCESS_FAILED

The device controller could not be initialized. Check if the functions of the hardware abstraction layer are properly implemented to guarantee correct hardware access.

USBF_STATUS_DEFERRED_DATA_STAGE

The application can return this value in a setup request to indicate the data stage is completed later.

USBF_STATUS_STALL

The application wants to stall the control request.

USBF_STATUS_INVALID_DATA

An invalid data packet was received.

USBF_STATUS_OPEN_ENDPOINT_FAILED

Failed to open the endpoint.

USBF_STATUS_INSTANCE_DEACTIVATED

The instance is deactivated.

USBF_STATUS_NOT_SUSPENDED

The device is not in suspend state.

USBF_STATUS_NOT_ENABLED

Remote wakeup is not enabled by the host.

USBF_STATUS_NOMEMORY

There is no memory for this operation.

USBF_STATUS_ISO_CRC

Isochronous CRC error.

USBF_STATUS_ISO_SEQUENCE

Invalid data phase PID sequence detected on a isochronous endpoint.

USBF_STATUS_UNDEFINED

Undefined status, this is used for initialization of status fields.

7 CDC/ACM Device Class Reference

7.1 Initialization

The following functions are used for the initialization of the CDC/ACM Device Class. They are defined in the file `cls_cdcacm_init.h`.

UsbCdcAcm_Initialize

A call to this function initializes the CDC/ACM device class.

Definition

```
void  
UsbCdcAcm_Initialize( );
```

Comments

This function must be called one time before any other function of the CDC/ACM device class can be called.

See Also

[UsbCdcAcm_CreateInstance](#) (page 68)

UsbCdcAcm_CreateInstance

This function creates a CDC/ACM device class instance.

Definition

```
T_UsbfStatus  
UsbCdcAcm_CreateInstance(  
    T_UsbfCdcAcmInstanceHandle* instanceHandle  
);
```

Parameter

instanceHandle

After the function returns with success, this field contains the valid instance handle for the appropriate CDC/ACM instance.

Return Value

The function returns one of the following status codes:

USBF_STATUS_NO_INSTANCE: - Reached the maximum number of CDC/ACM device class instances.

USBF_STATUS_SUCCESS: - The CDC/ACM device class instance created successfully. The field **instanceHandle** contains the valid instance handle for this CDC/ACM device class instance.

Comments

This function must be called before any other instance related function can be called. Use [UsbCdcAcm_DeleteInstance](#) to delete the specified instance.

See Also

[UsbCdcAcm_ActivateInstance](#) (page 70)

[UsbCdcAcm_DeleteInstance](#) (page 69)

UsbCdcAcm_DeleteInstance

This function deletes a CDC/ACM device class instance previously created by a call to **UsbCdcAcm_CreateInstance**.

Definition

```
void  
UsbCdcAcm_DeleteInstance(  
    T_UsbCdcAcmInstanceHandle instanceHandle  
);
```

Parameter

instanceHandle

This parameter specifies the instance to be deleted by its instance handle.

Comments

This function deletes the specified CDC/ACM device instance. The instance has to be deactivated by a call to **UsbCdcAcm_DeactivateInstance** before this function is called.

See Also

[UsbCdcAcm_DeactivateInstance](#) (page 72)

[UsbCdcAcm_CreateInstance](#) (page 68)

UsbCdcAcm_ActivateInstance

This function activates the specified CDC/ACM device class instance.

Definition

```
T_UsbfStatus
UsbCdcAcm_ActivateInstance(
    T_UsbCdcAcmInstanceHandle instanceHandle,
    T_UINT16 initialPortState,
    T_UINT8 commInterfaceNumber,
    T_UINT8 dataInterfaceNumber,
    T_UINT8 commEpAddr,
    T_UINT8 dataInEpAddr,
    T_UINT8 dataOutEpAddr
);
```

Parameters

instanceHandle

This parameter specifies the CDC/ACM device class instance which should be activated.

initialPortState

This field contains the initial state of the serial line.

commInterfaceNumber

This parameter specifies the USB interface number for the communication interface.

dataInterfaceNumber

This parameter specifies the USB interface number for the data interface.

commEpAddr

This parameter specifies the USB endpoint address of the communication endpoint.

dataInEpAddr

This parameter specifies the USB endpoint address of the data IN endpoint.

dataOutEpAddr

This parameter specifies the USB endpoint address of the data OUT endpoint.

Return Value

The function returns one of the following status codes:

USBF_STATUS_OPEN_ENDPOINT_FAILED: - One of the specified endpoints could not be opened.

USBF_STATUS_SUCCESS: - The CDC/ACM device class instance was successfully activated. Now it can be used for serial line communication.

Comments

This function must be called to activate the specified CDC/ACM device class instance. The function opens all necessary endpoints and registers handlers for the used interfaces. The CDC/ACM device instance must be created by a call to **UsbCdcAcm_CreateInstance** before this function can be called. Call **UsbCdcAcm_DeactivateInstance** to deactivate the CDC/ACM device class instance.

See Also

UsbCdcAcm_CreateInstance (page 68)

UsbCdcAcm_DeactivateInstance (page 72)

UsbCdcAcm_DeactivateInstance

This function deactivates the specified CDC/ACM device class instance.

Definition

```
void  
UsbCdcAcm_DeactivateInstance(  
    T_UsbCdcAcmInstanceHandle instanceHandle  
);
```

Parameter

instanceHandle

This parameter specifies the CDC/ACM device class instance which should be deactivated.

Comments

This function must be called to deactivate the specified CDC/ACM device class instance. The function closes all used endpoints and unregisters the handlers for the used interfaces. Call [UsbCdcAcm_ActivateInstance](#) to activate the CDC/ACM device class instance again. Call [UsbCdcAcm_DeleteInstance](#) to delete the CDC/ACM device class instance.

See Also

[UsbCdcAcm_ActivateInstance](#) (page 70)

[UsbCdcAcm_DeleteInstance](#) (page 69)

7.2 API Functions

The following functions form the interface of the CDC/ACM Device Class. They are defined in the file `cls_cdcacm_api.h`.

UsbCdcAcm_RegisterCallbacks

This function registers the callback functions for the CDC/ACM device class.

Definition

```
void
UsbCdcAcm_RegisterCallbacks(
    T_UsbCdcAcmInstanceHandle instanceHandle,
    void* instanceContext,
    T_UsbCdcAcm_RxBufferCompletion_Callback*
rxBufferCompletion,
    T_UsbCdcAcm_TxBufferCompletion_Callback*
txBufferCompletion,
    T_UsbCdcAcm_LineStateChanged_Callback* lineStateChanged
);
```

Parameters

instanceHandle

This parameter specifies the instance of the CDC/ACM device class for which the callback functions should be registered.

instanceContext

This parameter points to the context which will be passed to each callback function. The context is optional and therefore can be **NULL**;

rxBufferCompletion

This parameter points to the RX completion routine. This callback is required and therefore must not be **NULL**.

txBufferCompletion

This parameter points to the TX completion routine. This callback is required and therefore must not be **NULL**.

lineStateChanged

This points to the routine which will be called if the serial line state of this CDC/ACM device class instance has changed. This callback is optional and therefore can be **NULL**.

Comments

This function has to be called for each CDC/ACM device class instance to register the callback functions.

Note: This function can be called even if the CDC/ACM device class instance is deactivated.
Refer to section 5.4 for more information about callback functions.

See Also

T_UsbCdcAcm_RxBufferCompletion_Callback (page 88)

T_UsbCdcAcm_TxBufferCompletion_Callback (page 87)

T_UsbCdcAcm_LineStateChanged_Callback (page 85)

UsbCdcAcm_SetSerialState

This function sets the state of the serial line.

Definition

```
void  
UsbCdcAcm_SetSerialState(  
    T_UsbCdcAcmInstanceHandle instanceHandle,  
    T_UINT16 lineState  
);
```

Parameters

instanceHandle

This parameter specifies the instance of the CDC/ACM device class for which the state should be set.

lineState

This parameter specifies the new line state to be set. It can contain one or more of the following flags:

- **CLS_CDCACM_LINE_STATE_CTS** - The **CTS** signal should be set.
- **CLS_CDCACM_LINE_STATE_DCD** - The **DCD** signal should be set.
- **CLS_CDCACM_LINE_STATE_DSR** - The **DSR** signal should be set.
- **CLS_CDCACM_LINE_STATE_BREAK** - The **BREAK** signal should be set.
- **CLS_CDCACM_LINE_STATE_RING** - The **RING** signal should be set.

Return Value

The function returns one of the following status codes:

USBF_STATUS_SUCCESS: - The state of the serial line has been set successfully.

USBF_STATUS_INSTANCE_DEACTIVATED: - The state of the serial line can not be set because the CDC/ACM device class instance is deactivated.

Comments

The application has to notify the CDC/ACM device class whenever the serial line state has changed.

Note: This function can only be called when the CDC/ACM device class instance is activated.

See Also

[T_UsbCdcAcm_LineStateChanged_Callback](#) (page 85)

[UsbCdcAcm_GetSerialLineCoding](#) (page 76)

UsbCdcAcm_GetSerialLineCoding

This function retrieves the properties of the serial line.

Definition

```
void  
UsbCdcAcm_GetSerialLineCoding(  
    T_UsbCdcAcmInstanceHandle instanceHandle,  
    T_UINT32* baudRate,  
    T_UINT8* stopBits,  
    T_UINT8* parity,  
    T_UINT8* dataBits  
);
```

Parameters

instanceHandle

This parameter specifies the instance of the CDC/ACM device class.

baudRate

This parameter retrieves the baud rate of the serial line.

stopBits

This parameter retrieves the number of stop bits:

- 0 - 1 stop bit
- 1 - 1.5 stop bits
- 2 - 2 stop bits

parity

This parameter retrieves the parity used for the serial line:

- 0 - None
- 1 - Odd
- 2 - Even
- 3 - Mark
- 4 - Space

dataBits

This parameter retrieves the number of data bits (5,6,7,8 or 16).

Comments

This function can be used by the application to retrieve the configuration parameters of the serial line.

Note: This function can be called even if the CDC/ACM device class instance is deactivated.

UsbCdcAcm_SubmitTxBuffer

This function submits a buffer descriptor for writing.

Definition

```
T_UsbfStatus
UsbCdcAcm_SubmitTxBuffer(
    T_UsbCdcAcmInstanceHandle instanceHandle,
    T_UsbfBufferDescriptor* txBufDesc
);
```

Parameters

instanceHandle

This parameter specifies the instance of the CDC/ACM device class for which the buffer should be transferred.

txBufDesc

This parameter points to the caller provided buffer descriptor of type **T_UsbfBufferDescriptor**. The caller provides the storage for the data payload and fills the buffer with data. The storage must be persistent until the completion routine for this buffer is called.

It is possible to submit a chain of buffers which will be submitted in one packet. Refer to section 5.10 for more information about using a buffer chain.

The caller has to set the fields **dataSize** and **dataPtr**. The field **next** has to point to the next valid buffer descriptor in the buffer chain or has to be set to **NULL** for no buffer chain. The fields **dataSize** and **dataPtr** will not be changed by the CDC/ACM device class. Only the field **flags** will be used internally. On completion of the buffer the fields **status** and **bytesTransferred** will be set.

Return Value

The function returns one of the following status codes:

USBF_STATUS_PENDING: - The buffer was submitted successfully to the CDC/ACM device class. A write buffer will never complete immediately because of the error handling within the USB Function Library. The completion function for this buffer will be called later.

USBF_STATUS_INVALID_PARAM: - This status is returned if the endpoint handle is not valid.

USBF_STATUS_INSTANCE_DEACTIVATED: - The buffer can not be submitted because the CDC/ACM device class instance is deactivated.

Comments

This request is queued in the USB Function Library. It is allowed to submit more than one buffer.

If the function returns with **USBF_STATUS_PENDING** the buffer descriptor and the data memory is owned by the USB Function Library. On completion of the submitted buffer the function **T_UsbfCdcAcm_TxBufferCompletion_Callback** will be called.

If the function returns with a different status code as **USBF_STATUS_PENDING** the completion function is never called for this buffer.

The buffer is returned if it is completely submitted or an error occurred.

Note: This function will never return **USBF_STATUS_SUCCESS**. This function can only be called when the CDC/ACM device class instance is activated.

See Also

T_UsbfBufferDescriptor (page 56)

T_UsbfCdcAcm_TxBufferCompletion_Callback (page 87)

UsbfCdcAcm_SubmitRxBuffer (page 79)

UsbCdcAcm_SubmitRxBuffer

This function submits a buffer descriptor for receiving.

Definition

```
T_UsbfStatus  
UsbCdcAcm_SubmitRxBuffer(  
    T_UsbCdcAcmInstanceHandle instanceHandle,  
    T_UsbfBufferDescriptor* rxBufDesc  
);
```

Parameters

instanceHandle

This parameter specifies the instance of the CDC/ACM device class for which the buffer should be received.

rxBufDesc

This parameter points to the receive buffer descriptor of type **T_UsbfBufferDescriptor**. The caller provides the storage for the payload data which must be persistent until the completion routine for this buffer is called.

The caller has to set the fields **dataSize** and **dataPtr**. The field **next** has to be set to **NULL**. On completion of the buffer, the CDC/ACM device class will set the fields **status** and **bytesTransferred**.

Return Value

The function returns one of the following status codes:

USBF_STATUS_SUCCESS: - The buffer was successfully processed by the CDC/ACM device class. The data of the buffer have been received immediately. If this status is returned the completion routine will not be called for this buffer.

USBF_STATUS_PENDING: - The buffer was submitted successfully to the CDC/ACM device class. The buffer cannot be completed immediately because not all data are available at this time. It will be completed later by a call to **T_UsbCdcAcm_RxBufferCompletion_Callback**.

USBF_STATUS_INVALID_PARAM: - This status is returned if the endpoint handle is not valid.

USBF_STATUS_INSTANCE_DEACTIVATED: - The buffer can not be submitted because the CDC/ACM device class instance is deactivated.

Comments

This request is queued in the USB Function Library. It is allowed to submit more than one buffer.

If the function returns with **USBF_STATUS_PENDING** the buffer descriptor is owned by the USB Function Library. On completion of the submitted buffer the function

T_UsbfCdcAcм_RxBufferCompletion_Callback will be called.

If the function returns with a different status code as **USBF_STATUS_PENDING** the completion function is never called for this buffer.

The buffer is returned if it is completely filled with data or an error occurred.

Note: This function can only be called when the CDC/ACM device class instance is activated.

See Also

T_UsbfBufferDescriptor (page 56)

T_UsbfCdcAcм_RxBufferCompletion_Callback (page 88)

UsbfCdcAcм_SubmitTxBuffer (page 77)

UsbCdcAcm_AbortTx

This function cancels all outstanding TX data buffers.

Definition

```
void  
UsbCdcAcm_AbortTx(  
    T_UsbCdcAcmInstanceHandle instanceHandle  
);
```

Parameter

instanceHandle

This parameter specifies the instance of the CDC/ACM device class.

Comments

This function cancels each buffer which was previously submitted to the CDC/ACM device class by a call to **UsbCdcAcm_SubmitTxBuffer**. The packet completion routine is called with status **USBF_STATUS_CANCELED** if the transfer is aborted.

Note: This function can be called even if the CDC/ACM device class instance is deactivated.

See Also

[T_UsbCdcAcm_TxBufferCompletion_Callback](#) (page 87)

[UsbCdcAcm_AbortRx](#) (page 82)

UsbCdcAcm_AbortRx

This function cancels all outstanding RX data buffers.

Definition

```
void  
UsbCdcAcm_AbortRx(  
    T_UsbCdcAcmInstanceHandle instanceHandle  
);
```

Parameter

instanceHandle

This parameter specifies the instance of the CDC/ACM device class.

Comments

This function cancels each buffer which was previously submitted to the CDC/ACM device class by a call to **UsbCdcAcm_SubmitRxBuffer**. The packet completion routine is called with status **USBF_STATUS_CANCELED** if the transfer is aborted.

Note: This function can be called even if the CDC/ACM device class instance is deactivated.

See Also

[T_UsbCdcAcm_RxBufferCompletion_Callback](#) (page 88)

[UsbCdcAcm_AbortTx](#) (page 81)

UsbCdcAcm_SetTraceMask

This function sets an internal trace mask which filters trace messages produced by the CDC/ACM device class.

Definition

```
void  
UsbCdcAcm_SetTraceMask(  
    unsigned int Mask  
);
```

Parameter

Mask

Specifies the trace mask to be set.

Comments

The trace mask is an internal global variable. A specific bit position within that variable is assigned to a particular type of trace message used within CDC/ACM device class. All Messages of the specified type will be printed out if the corresponding bit is set but will be suppressed if the corresponding bit is cleared. Consequently the current value of the trace mask determines the amount of trace messages produced by the CDC/ACM device class. Bit positions of the trace mask are assigned as described below.

DBG_ERR

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

DBG_WARN

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

DBG_INFO

Informational messages.

DBG_CDCACM

Print the function names.

DBG_RXDMP

Dump received buffers.

DBG_TXDMP

Dump transmit buffers.

DBG_CTRL

Prints information about the CDC/ACM control channel.

DBG_CTRLDMP

Dumps control buffers.

DBG_INNER

Prints the names of internal called functions.

DBG_INST

Prints instance related information.

By default, the bits **DBG_ERR** and **DBG_WARN** are set.

Note that the **DBG_XXX** constants specify bit positions and not the corresponding masks. Use the **CLS_CDCACM_TRCMSK** macro to create the corresponding mask for an individual bit position.

Example:

```
UsbCdcAcm_SetTraceMask(  
CLS_CDCACM_TRCMSK(DBG_ERR)  
|CLS_CDCACM_TRCMSK(DBG_INFO)  
);
```

Trace support can be enabled/disabled at compile time. If **DBG** has a non zero value the trace support is enabled. Note that a call to **UsbCdcAcm_SetTraceMask** has no effect if trace support is disabled.

Note: This function can be called even if the CDC/ACM device class instance is deactivated.

7.3 Callbacks

The callback functions of the CDC/ACM Device Class will always be called in a defined context. Refer to the reference of the specified callback function to see in which context it will be called.

T_UsbfCdcAcm_LineStateChanged_Callback

Notifies the application about a change of the serial line state.

Definition

```
void  
T_UsbfCdcAcm_LineStateChanged_Callback(  
    void* instanceContext,  
    T_UINT16 lineState,  
    T_UINT16 duration  
);
```

Parameters

instanceContext

The context pointer of the CDC/ACM device class instance.

lineState

This parameter specifies the changed state of the serial line with the following values:

- **CLS_CDCACM_LINE_STATE_RTS** - Received a **RTS** signal.
- **CLS_CDCACM_LINE_STATE_DTR** - Received a **DTR** signal.
- **CLS_CDCACM_LINE_STATE_BREAK** - Received a **BREAK** signal. The parameter **duration** contains the duration for the **BREAK** signal.

duration

In case of a **BREAK** signal this parameter specifies the duration of the break.

Comments

This function is called whenever the state of the serial line has changed.

This function is called in the context of:

- **Usbf_ProcessInterrupt**.

Refer to section 5.4 for more information about callback functions.

See Also

UsbCdcAcm_RegisterCallbacks (page 73)

UsbCdcAcm_SetSerialState (page 75)

UsbCdcAcm_GetSerialLineCoding (page 76)

T_UsbfCdcAcm_TxBufferCompletion_Callback

This callback function is called if a pending transmit buffer is completed.

Definition

```
void  
T_UsbfCdcAcm_TxBufferCompletion_Callback(  
    void* instanceContext,  
    T_UsbfBufferDescriptor* txBufDesc  
);
```

Parameters

instanceContext

The context pointer of the CDC/ACM device class instance.

txBufDesc

This parameter points to the completed buffer descriptor of type **T_UsbfBufferDescriptor**.

Comments

This function is called on completion for each pending transmit buffer which was successfully submitted to the CDC/ACM device class.

This function is called in the context of:

- **Usbf_ProcessInterrupt**.
- **UsbfCdcAcm_AbortTx**.

Refer to section 5.4 for more information about callback functions.

See Also

[UsbfCdcAcm_RegisterCallbacks](#) (page 73)

[T_UsbfCdcAcm_RxBufferCompletion_Callback](#) (page 88)

[UsbfCdcAcm_SubmitTxBuffer](#) (page 77)

[T_UsbfBufferDescriptor](#) (page 56)

T_UsbfCdcAcm_RxBufferCompletion_Callback

This callback function is called if a pending receive buffer is completed.

Definition

```
void  
T_UsbfCdcAcm_RxBufferCompletion_Callback(  
    void* instanceContext,  
    T_UsbfBufferDescriptor* rxBufDesc  
);
```

Parameters

instanceContext

The context pointer of the CDC/ACM device class instance.

rxBufDesc

This parameter points to the received buffer descriptor of type **T_UsbfBufferDescriptor**.

Comments

This function is called on completion for each pending receive buffer which was successfully submitted to the library.

This function is called in the context of:

- **Usbf_ProcessInterrupt**.
- **UsbfCdcAcm_AbortRx**.

Refer to section 5.4 for more information about callback functions.

See Also

[UsbfCdcAcm_RegisterCallbacks](#) (page 73)

[T_UsbfCdcAcm_RxBufferCompletion_Callback](#) (page 88)

[UsbfCdcAcm_SubmitRxBuffer](#) (page 79)

[T_UsbfBufferDescriptor](#) (page 56)

7.4 Status Codes

USBF_STATUS_NO_INSTANCE

Reached the maximum number of available instances.

8 Hardware Abstraction Layer Reference

8.1 API Functions

The following functions define the interface of the hardware abstraction layer. They are defined in the file `usbf_hal_api.h`.

UsbHal_WriteReg8

This function writes an 8-bit value to a specified register.

Definition

```
void  
UsbHal_WriteReg8(  
    T_UINT8* address,  
    T_UINT8 value  
);
```

Parameters

address

This field specifies the register by its address.

value

This field contains the value which should be written to the specified register.

Comments

Refer to section 2.7 to see if this function has to be implemented for your specific device controller.

See Also

[UsbHal_WriteReg16](#) (page 92)

[UsbHal_WriteReg32](#) (page 94)

[UsbHal_ReadReg8](#) (page 96)

UsbHal_BurstWriteReg8

This function writes multiple 8-bit values to a specified register.

Definition

```
void  
UsbHal_BurstWriteReg8(  
    T_UINT8* address,  
    const T_UINT8* buffer,  
    unsigned int count  
);
```

Parameters

address

This field specifies the register by its address.

buffer

This field points to the byte buffer containing the values to be written.

count

This field contains the number of 8-bit values to be written.

Comments

This function reads **count** times an 8-bit value from **buffer** and writes it to the register specified by **address**. After each write process the address of the **buffer** is incremented but the address of the register remains.

Refer to section 2.7 to see if this function has to be implemented for your specific device controller.

See Also

[UsbHal_BurstWriteReg16](#) (page 93)

[UsbHal_BurstWriteReg32](#) (page 95)

[UsbHal_BurstReadReg8](#) (page 97)

UsbHal_WriteReg16

This function writes an 16-bit value to a specified register.

Definition

```
void  
UsbHal_WriteReg16(  
    T_UINT16* address,  
    T_UINT16 value  
);
```

Parameters

address

This field specifies the register by its address.

value

This field contains the value which should be written to the specified register.

Comments

Refer to section [2.7](#) to see if this function has to be implemented for your specific device controller.

See Also

[UsbHal_WriteReg8](#) (page 90)
[UsbHal_WriteReg32](#) (page 94)
[UsbHal_ReadReg16](#) (page 98)

UsbHal_BurstWriteReg16

This function writes multiple 16-bit values to a specified register.

Definition

```
void  
UsbHal_BurstWriteReg16(  
    T_UINT16* address,  
    const T_UINT8* buffer,  
    unsigned int count  
);
```

Parameters

address

This field specifies the register by its address.

buffer

This field points to the byte buffer containing the values to be written.

count

This field contains the number of 16-bit values to be written.

Comments

This function reads **count** times a 16-bit value from **buffer** and writes it to the register specified by **address**. After each write process the address of the **buffer** is incremented but the address of the register remains.

Note: The byte buffer is assumed to be in little endian byte order like it is transferred over USB. Therefore it can be necessary to swap the byte ordering on big endian platforms.

Refer to section 2.7 to see if this function has to be implemented for your specific device controller.

See Also

[UsbHal_BurstWriteReg8](#) (page 91)

[UsbHal_BurstWriteReg32](#) (page 95)

[UsbHal_BurstReadReg16](#) (page 99)

UsbHal_WriteReg32

This function writes an 32-bit value to a specified register.

Definition

```
void  
UsbHal_WriteReg32(  
    T_UINT32* address,  
    T_UINT32 value  
);
```

Parameters

address

This field specifies the register by its address.

value

This field contains the value which should be written to the specified register.

Comments

Refer to section 2.7 to see if this function has to be implemented for your specific device controller.

See Also

[UsbHal_WriteReg8](#) (page 90)
[UsbHal_WriteReg16](#) (page 92)
[UsbHal_ReadReg32](#) (page 100)

UsbHal_BurstWriteReg32

This function writes multiple 32-bit values to a specified register.

Definition

```
void  
UsbHal_BurstWriteReg32(  
    T_UINT32* address,  
    const T_UINT8* buffer,  
    unsigned int count  
);
```

Parameters

address

This field specifies the register by its address.

buffer

This field points to the byte buffer containing the values to be written.

count

This field contains the number of 32-bit values to be written.

Comments

This function reads **count** times a 32-bit value from **buffer** and writes it to the register specified by **address**. After each write process the address of the **buffer** is incremented but the address of the register remains.

Note: The byte buffer is assumed to be in little endian byte order like it is transferred over USB. Therefore it can be necessary to swap the byte ordering on big endian platforms.

Refer to section 2.7 to see if this function has to be implemented for your specific device controller.

See Also

[UsbHal_BurstWriteReg8](#) (page 91)

[UsbHal_BurstWriteReg16](#) (page 93)

[UsbHal_BurstReadReg32](#) (page 101)

UsbHal_ReadReg8

This function reads 8 bit from a specified register.

Definition

```
T_UINT8  
UsbHal_ReadReg8(  
    T_UINT8* address  
);
```

Parameter

address

This field specifies the register to read from by its address.

Return Value

This function returns the 8-bit value read from the specified register.

Comments

Refer to section [2.7](#) to see if this function has to be implemented for your specific device controller.

See Also

[UsbHal_ReadReg16](#) (page 98)

[UsbHal_ReadReg32](#) (page 100)

[UsbHal_WriteReg8](#) (page 90)

UsbHal_BurstReadReg8

This function reads multiple 8-bit values from the specified register.

Definition

```
void
UsbHal_BurstReadReg8(
    T_UINT8* address,
    T_UINT8* buffer,
    unsigned int count
);
```

Parameters

address

This field specifies the register by its address.

buffer

This field points to the buffer for the values read.

count

This field contains the number of 8-bit values to be read.

Comments

This function reads **count** times an 8-bit value from the register specified by **address** and stores this value in **buffer**. After each read process the address of the **buffer** is incremented but the address of the register remains.

Refer to section 2.7 to see if this function has to be implemented for your specific device controller.

See Also

[UsbHal_BurstReadReg16](#) (page 99)

[UsbHal_BurstReadReg32](#) (page 101)

[UsbHal_BurstWriteReg8](#) (page 91)

UsbHal_ReadReg16

This function reads 16 bit from a specified register.

Definition

```
T_UINT16  
UsbHal_ReadReg16(  
    T_UINT16* address  
);
```

Parameter

address

This field specifies the register to read from by its address.

Return Value

This function returns the 16-bit value read from the specified register.

Comments

Refer to section 2.7 to see if this function has to be implemented for your specific device controller.

See Also

[UsbHal_ReadReg8](#) (page 96)
[UsbHal_ReadReg32](#) (page 100)
[UsbHal_WriteReg16](#) (page 92)

UsbHal_BurstReadReg16

This function reads multiple 16-bit values from the specified register.

Definition

```
void  
UsbHal_BurstReadReg16(  
    T_UINT16* address,  
    T_UINT8* buffer,  
    unsigned int count  
);
```

Parameters

address

This field specifies the register by its address.

buffer

This field points to the byte buffer for the values read.

count

This field contains the number of 16-bit values to be read.

Comments

This function reads **count** times a 16-bit value from the register specified by **address** and stores this value in **buffer**. After each read process the address of the **buffer** is incremented but the address of the register remains.

Note: The byte buffer is assumed to be in little endian byte order like it is transferred over USB. Therefore it can be necessary to swap the byte ordering on big endian platforms.

Refer to section 2.7 to see if this function has to be implemented for your specific device controller.

See Also

[UsbHal_BurstReadReg8](#) (page 97)

[UsbHal_BurstReadReg32](#) (page 101)

[UsbHal_BurstWriteReg16](#) (page 93)

UsbHal_ReadReg32

This function reads 32 bit from a specified register.

Definition

```
T_UINT32  
UsbHal_ReadReg32(  
    T_UINT32* address  
);
```

Parameter

address

This field specifies the register to read from by its address.

Return Value

This function returns the 32-bit value read from the specified register.

Comments

Refer to section 2.7 to see if this function has to be implemented for your specific device controller.

See Also

[UsbHal_ReadReg8](#) (page 96)
[UsbHal_ReadReg16](#) (page 98)
[UsbHal_WriteReg32](#) (page 94)

UsbHal_BurstReadReg32

This function reads multiple 32-bit values from the specified register.

Definition

```
void  
UsbHal_BurstReadReg32(  
    T_UINT32* address,  
    T_UINT8* buffer,  
    unsigned int count  
);
```

Parameters

address

This field specifies the register by its address.

buffer

This field points to the byte buffer for the values read.

count

This field contains the number of 32-bit values to be read.

Comments

This function reads **count** times a 32-bit value from the register specified by **address** and stores this value in **buffer**. After each read process the address of the buffer is incremented but the address of the register remains.

Note: The byte buffer is assumed to be in little endian byte order like it is transferred over USB. Therefore it can be necessary to swap the byte ordering on big endian platforms.

Refer to section 2.7 to see if this function has to be implemented for your specific device controller.

See Also

[UsbHal_BurstReadReg8](#) (page 97)

[UsbHal_BurstReadReg16](#) (page 99)

[UsbHal_BurstWriteReg32](#) (page 95)

UsbHal_FifoWrite

This function writes **count** bytes from **buffer** to **destination**.

Definition

```
void
UsbHal_FifoWrite(
    void* destination,
    const void* buffer,
    unsigned int count
);
```

Parameters

destination

This field specifies the address to write to.

buffer

This field points to the buffer containing the data to be written.

count

This field contains the number of bytes to write.

Comments

This function begins writing at **destination**. It writes **count** bytes from **buffer** to **destination** until **destination + count** is reached.

Note: The buffer is assumed to be in little endian byte order like it is transferred over USB. Therefore it can be necessary to swap the byte ordering on big endian platforms.

Refer to section 2.7 to see if this function has to be implemented for your specific device controller.

See Also

[UsbHal_FifoRead](#) (page 103)

UsbHal_FifoRead

This function reads **count** bytes from **source** to **buffer**.

Definition

```
void
UsbHal_FifoRead(
    void* source,
    void* buffer,
    unsigned int count
);
```

Parameters

source

This field specifies the address to read from.

buffer

This field points to the buffer where the data read should be stored.

count

This field contains the number of bytes to read.

Comments

This function begins reading at **source**. It reads **count** bytes until **source + count** is reached. The data read will be stored in **buffer**.

Note: The buffer is assumed to be in little endian byte order like it is transferred over USB. Therefore it can be necessary to swap the byte ordering on big endian platforms.

Refer to section 2.7 to see if this function has to be implemented for your specific device controller.

See Also

[UsbHal_FifoWrite](#) (page 102)

UsbHal_IsVBusPresent

This function determines if VBus is present or not.

Definition

```
int  
UsbHal_IsVBusPresent ( ) ;
```

Return Value

This function has to be implemented by the application designer. It has to return **0** if VBus is not present and **1** if VBus is present.

Comments

If the device controller does not support the VBus detection within the USB peripheral it has to be provided externally by this function. In this case refer to the manual of the device controller for more information on implementing the VBus detection.

Refer to section [2.7](#) to see if this function has to be implemented for your specific device controller.

UsbHal_SetSoftConnect

This function connects the device controller to the USB.

Definition

```
void  
UsbHal_SetSoftConnect(  
    unsigned int connectState  
);
```

Parameter

connectState

This field determines the connect state of the 1.5k resistor. A **connectState** value of **0** indicates that the device should be disconnected. If the device should be connected a value of **1** is used.

Comments

If the device controller does not support the soft connect feature within the USB peripheral this has to be provided externally by this function. In this case refer to the manual of the device controller for more information on implementing the soft connect feature.

Refer to section [2.7](#) to see if this function has to be implemented for your specific device controller.

UsbHal_Delay

This function performs a short delay required by some device controllers.

Definition

```
void  
UsbHal_Delay(  
    unsigned int nanoSeconds  
);
```

*Parameter***nanoSeconds**

This field contains the time for the delay in nano seconds.

Comments

This function will be used for very short delays of a few hundred nano seconds only. Refer to section 2.7 to see if this function has to be implemented for your specific device controller. There may be additional information about the usual period which is used for the specific device controller.

The implementation of this function may only be necessary for fast micro controllers. The application designer has to decide if this function has to be implemented or can be defined as empty macro.

9 Adaption Layer Reference

The file `tal_defs.h` contains basic type definitions which are used within the USB Function Library and its device classes. It can be necessary to adjust these definitions for your environment.

9.1 API Functions

The following functions define the interface of the adaption layer. They are defined in the file `tal_api.h` and must be implemented for your application.

Tal_Printf

A printf-compatible trace function.

Definition

```
void  
Tal_Printf(  
    const char* format,  
    ...  
);
```

Parameters

format

Specifies a pointer to the format string to print. The format string supports a subset of the printf-style formatting codes. The format codes to be supported are: %c, %d, %u, %s, %x, %X.

...

Specifies optional arguments for the format string, as in printf.

Comments

This function is part of the Thesycon adaption layer. It has to be implemented by the application and will be called by the library.

Tal_ZeroMemory

This function clears a memory block.

Definition

```
void  
Tal_ZeroMemory(  
    void* memory,  
    unsigned int numberOfBytes  
);
```

Parameters

memory

Points to the memory block to be cleared. The pointer specifies a byte address. No assumptions on the alignment must be made.

numberOfBytes

Specifies the size of the memory block to be cleared.

Comments

This function is part of the Thesycon adaption layer. It has to be implemented by the application and will be called by the library.

See Also

[Tal_CopyMemory](#) **[Tal_CompareMemory](#)** **[Tal_SetMemory](#)**

Tal_CopyMemory

This function copies a memory block.

Definition

```
void  
Tal_CopyMemory(  
    void* destination,  
    const void* source,  
    unsigned int numberOfBytes  
);
```

Parameters

destination

Points to the destination where the memory block should be copied to. The pointer specifies a byte address. No assumptions on the alignment must be made.

source

Points to the memory block to be copied. The pointer specifies a byte address. No assumptions on the alignment must be made.

numberOfBytes

Specifies the number of bytes to be copied.

Comments

Destination and source regions must not overlap.

This function is part of the Thesycon adaption layer. It has to be implemented by the application and will be called by the library.

See Also

[Tal_ZeroMemory](#) [Tal_CompareMemory](#) [Tal_SetMemory](#)

Tal_CompareMemory

This function compares characters in two buffers.

Definition

```
int  
Tal_CompareMemory(  
    void* buffer1,  
    const void* buffer2,  
    unsigned int numberOfBytes  
);
```

Parameters

buffer1

Points to the first buffer where the memory block should be compared. The pointer specifies a byte address. No assumptions on the alignment must be made.

buffer2

Points to the second buffer where the memory block should be compared with. The pointer specifies a byte address. No assumptions on the alignment must be made.

numberOfBytes

Specifies the number of bytes to be compared.

Return Value

The function returns a value which indicate the relationship between the buffers.

<0 - The return value is less than 0 if buffer1 is less than buffer2.

0 - The return value is 0 if buffer1 is identical to buffer2.

>0 - The return value is greater than 0 if buffer1 is greater than buffer2.

Comments

This function is part of the Thesycon adaption layer. It has to be implemented by the application and will be called by the library.

See Also

[Tal_ZeroMemory](#) [Tal_CopyMemory](#) [Tal_SetMemory](#)

Tal_SetMemory

This function Sets buffers to a specified character.

Definition

```
void  
Tal_SetMemory(  
    void* memory,  
    void* pattern,  
    unsigned int numberOfBytes  
);
```

Parameters

memory

Points to the memory block to be set. The pointer specifies a byte address. No assumptions on the alignment must be made.

pattern

Points to the character to set.

numberOfBytes

Specifies the size of the memory block to be set.

Comments

This function is part of the Thesycon adaption layer. It has to be implemented by the application and will be called by the library.

See Also

[Tal_ZeroMemory](#) [Tal_CopyMemory](#) [Tal_CompareMemory](#)

Tal_OffsetOfMember

This macro calculates the byte offset of a member within an object.

Definition

```
Tal_OffsetOfMember(  
    type ,  
    member  
);
```

Parameters

type

This parameter specifies the name of the object's type.

member

This parameter specifies the name of the member.

Comments

This macro has to be defined in the file **tal_impl.h**.

Normally it can be mapped to the compiler specific macro **offsetof()** defined in **stddef.h**.

Tal_GetTickCount

This function returns a tick count.

Definition

```
Tal_GetTickCount( ) ;
```

Return Value

A 32 bit counter value.

Comments

The returned 32 bit counter value is incremented every millisecond. It has to be in the range from zero to 0xFFFF FFFF. This function must implemented if the Thesycon test application (app_tb) or the Thesycon bootloader test application (bootloader_dfu) or the USB DFU class is used!

10 Demo Applications

This section describes the simple demo applications provided with the USB Function Library package. All demo applications can be downloaded for free at <http://www.thesycon.de>.

The demo applications are implemented either on top of the USB Function Library or on top of a specified device class of the Embedded USB Device Stack.

For more information about these demo applications refer to <http://www.thesycon.de> or contact [usb\(at\)thesycon.de](mailto:usb(at)thesycon.de).

10.1 Microsoft USB CDC/ACM Device Driver

For the CDC/ACM device class there is a CDC/ACM device driver included in Windows operating systems which can be used for the CDC/ACM Loop demo application. This driver is officially not supported by Microsoft and some customers also reported problems with reliability. Therefore we recommend to use our own CDC/ACM device driver. It is optimized for the Embedded USB Device Stack and provides enhanced error recovery mechanisms.

10.2 Device Driver for the PC

Thesycon also provides the appropriate device drivers for the PC which can be used to communicate with the embedded device. These drivers are optimized for use with the Embedded USB Device Stack and included in the software package for demonstration purposes. This section contains some useful information about the device drivers included in the package.

For more detailed information about a specific device driver or further device drivers provided by Thesycon refer to <http://www.thesycon.de> or contact [info\(at\)thesycon.de](mailto:info(at)thesycon.de).

10.2.1 Demo Limitation

The device drivers included in the software package were built as demo versions. They have no functional restrictions but a limitation in the time they can be used. Each device driver demo version has a demo timeout. If this timeout expires, the device driver will stop the communication with the device and report an appropriate error message. The device driver can not be used again to communicate with the device until the PC is rebooted. After that the device driver can be used again with no restriction until the demo timeout expires for the next time.

10.2.2 Device Driver Installation

There is a setup package included for each device driver provided by Thesycon. To install a device driver simply run the appropriate setup and follow the instructions on the screen. Previous installations will be detected automatically and uninstalled upon your confirmation. The setup will install the device driver for the appropriate USB device in the Windows operating system. If the hardware wizard pops up and asks for a suitable driver you can now let him search automatically and he will find the installed device driver for the appropriate embedded USB device. Furthermore the installer will add an entry to the 'Add or Remove Programs' list which you can use to

completely uninstall the device driver. This is possible even if the device is not connected to the PC.

10.2.3 CDC/ACM Device Driver

The USB CDC/ACM Class Driver for Windows can be used with the CDC/ACM loop demo application of the Embedded USB Device Stack. If installed, the driver exposes a serial (COM) port on the PC which can be used like a standard serial port. This allows to re-use existing Windows applications unmodified.

You can download the complete demo package of the USB CDC/ACM Class Driver for Windows at <http://www.thesycon.de/cdcacm>. Please contact [usb\(at\)thesycon.de](mailto:usb(at)thesycon.de) if you have any questions about this software.

Refer to <http://www.thesycon.de> for more information about CDC/ACM, embedded USB and USB drivers for Windows operating systems.

10.2.4 CDC/ACM MI Device Driver

The USB CDC/ACM MI Class Driver for Windows can be used with the CDC/ACM MI loop demo application of the Embedded USB Device Stack. If installed, the driver exposes two serial (COM) ports on the PC which can be used like standard serial ports. This allows to re-use existing Windows applications unmodified.

Please contact [usb\(at\)thesycon.de](mailto:usb(at)thesycon.de) if you have any questions about this software.

Refer to <http://www.thesycon.de> for more information about CDC/ACM, embedded USB and USB drivers for Windows operating systems.

10.3 CDC/ACM Loopback Demo Application

The CDC/ACM loopback demo application represents a plain sample for the usage of the CDC/ACM Device Class. Implemented directly on top of the CDC/ACM Device Class the demo application is designed to test the serial line communication. Furthermore this demo application provides a good starting point for implementing your own application. It implements a simple data transfer loop for sending all data received on the RX line back to the TX line. There is a variant of the application that uses a single instance of the CDC/ACM Device Class and another variant that demonstrates using two instances of the CDC/ACM Device Class simultaneously.

This demo application provides a USB device descriptor with two interfaces. A communication class interface with an INT IN endpoint and a data class interface with a BULK IN and a BULK OUT endpoint.

To perform data transfer between the host and the device you can use the USB CDC/ACM Class Driver for Windows provided by Thesycon® Systemsoftware & Consulting GmbH. The demo version of the driver is included within this package. For more information about the device driver refer to section [10.2](#).

10.3.1 Using the CDC/ACM Device

Program the device with the CDC/ACM loopback demo application firmware and connect the USB device to the Windows PC. The Windows operating system will recognize the new device and ask you for a suitable device driver if this is the first time you connect the device. The installation of the CDC/ACM device driver is described in section 10.2.

After successful installation of the appropriate device driver a new serial link will appear in the device manager.

Use a terminal application like HyperTerminal to open the appropriate serial port of the CDC/ACM device. Now you can send characters to the device which will be looped inside the device and immediately send back to your terminal application.

Please recognize the **echo** functionality. Usually a terminal application can be configured if it locally echoes the characters typed in or not. Depending on this setting it is possible that the characters typed in are displayed twice. In this case one is the locally echoed character.

You can also use the `chkcom` application provided within this package to check the communication with the new serial port. The application opens the serial port you specified by a command line parameter. It writes a buffer to the serial port and expects that this buffer will be looped back. Therefore it reads a buffer from the serial line and verifies it with the written buffer. You can get an online help which describes all available command line parameters by typing `'chkcom -h'`.

Index

T_Usbf_DeviceEvent_Callback, 52
T_Usbf_StartOfFrame_Callback, 55
T_UsbfBufferDescriptor, 56
T_UsbfCdcAcm_LineStateChanged_Callback, 85
T_UsbfCdcAcm_RxBufferCompletion_Callback, 88
T_UsbfCdcAcm_TxBufferCompletion_Callback, 87
T_UsbfDescriptorRecord, 59
T_UsbfDescriptors, 61
T_UsbfDeviceHandler, 63
T_UsbfStringDescriptorRecord, 60
Tal_CompareMemory, 110
Tal_CopyMemory, 109
Tal_GetTickCount, 113
Tal_OffsetOfMember, 112
Tal_Printf, 107
Tal_SetMemory, 111
Tal_ZeroMemory, 108

Usbf_CheckVBus, 49
Usbf_Disable, 44
Usbf_Enable, 43
Usbf_Initialize, 42
Usbf_InterruptServiceRoutine, 47
Usbf_ProcessInterrupt, 48
Usbf_RegisterDeviceHandler, 45
Usbf_SetTraceMask, 50
USBF_STATUS_BUFFER_OVERFLOW, 64
USBF_STATUS_CANCELED, 64
USBF_STATUS_DEFERRED_DATA_STAGE, 65
USBF_STATUS_ENDPOINT_STALLED, 64
USBF_STATUS_FIFO_EMPTY, 64
USBF_STATUS_FIFO_FULL, 64
USBF_STATUS_HW_ACCESS_FAILED, 65
USBF_STATUS_INSTANCE_DEACTIVATED, 65
USBF_STATUS_INVALID_DATA, 65
USBF_STATUS_INVALID_PARAM, 64
USBF_STATUS_ISO_CRC, 66
USBF_STATUS_ISO_SEQUENCE, 66
USBF_STATUS_NO_INSTANCE, 89
USBF_STATUS_NOMEMORY, 65
USBF_STATUS_NOT_ENABLED, 65
USBF_STATUS_NOT_OPENED, 64
USBF_STATUS_NOT_SUSPENDED, 65
USBF_STATUS_OPEN_ENDPOINT_FAILED, 65
USBF_STATUS_PENDING, 64
USBF_STATUS_STALL, 65

USBF_STATUS_SUCCESS, 64
USBF_STATUS_UNDEFINED, 66
Usbf_UnregisterDeviceHandler, 46
UsbfCdcAcm_AbortRx, 82
UsbfCdcAcm_AbortTx, 81
UsbfCdcAcm_ActivateInstance, 70
UsbfCdcAcm_CreateInstance, 68
UsbfCdcAcm_DeactivateInstance, 72
UsbfCdcAcm_DeleteInstance, 69
UsbfCdcAcm_GetSerialLineCoding, 76
UsbfCdcAcm_Initialize, 67
UsbfCdcAcm_RegisterCallbacks, 73
UsbfCdcAcm_SetSerialState, 75
UsbfCdcAcm_SetTraceMask, 83
UsbfCdcAcm_SubmitRxBuffer, 79
UsbfCdcAcm_SubmitTxBuffer, 77
UsbfHal_BurstReadReg16, 99
UsbfHal_BurstReadReg32, 101
UsbfHal_BurstReadReg8, 97
UsbfHal_BurstWriteReg16, 93
UsbfHal_BurstWriteReg32, 95
UsbfHal_BurstWriteReg8, 91
UsbfHal_Delay, 106
UsbfHal_FifoRead, 103
UsbfHal_FifoWrite, 102
UsbfHal_IsVBusPresent, 104
UsbfHal_ReadReg16, 98
UsbfHal_ReadReg32, 100
UsbfHal_ReadReg8, 96
UsbfHal_SetSoftConnect, 105
UsbfHal_WriteReg16, 92
UsbfHal_WriteReg32, 94
UsbfHal_WriteReg8, 90