

FUFA

Fujitsu USB Function API

for MB90330, MB96330 and MB91660 series

Reference Manual

Version: 2.1.6
Date: 05 February 2009

Authors: Steffen Weiss

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) 2005-2009 by Thesycon® Systemsoftware & Consulting GmbH

All Rights Reserved

Disclaimer

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 throughout 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	7
1 Introduction	9
2 Overview	11
2.1 Compiler	11
2.2 Features	11
2.3 Restrictions	12
3 Architecture	13
3.1 Typical Program Flow	14
3.2 Integration of the Library into an Application	17
3.3 Compile time Configuration	17
3.4 Library Interrupt Handler	17
3.5 Synchronization	17
3.5.1 Without Operating System	18
3.5.2 With Operating System	18
3.6 Data Transfer and Performance	18
3.6.1 Transfer Device to PC	18
3.6.2 Transfer PC to Device	18
3.7 Class and Vendor Requests	19
3.7.1 Error Handling	19
3.7.2 Class and Vendor Requests without data phase	19
3.7.3 Class and Vendor Requests with data phase from PC to Device	20
3.7.4 Class and Vendor Requests with data phase from Device to PC	20
3.8 Hardware Requirements	20
4 Programming Interface	23
4.1 API Functions	23
LibUsbSetupBuffer	24
LibUsbSetupHandshake	26
UsbLibInitialize	27
UsbLibEnable	29
UsbLibDisable	30
UsbLibWakeupPC	31

UsbLibGetFrameNumber	32
UsbLibRead	33
UsbLibWrite	35
UsbLibAbort	37
UsbLibSetStall	38
UsbLibClearStall	39
UsbControlEndpointInterrupt	40
UsbDataEndpointInterrupt	41
UsbFunctionInterrupt	42
UsbShortPacketInterrupt	43
UsbLibSetTraceMask	44
UsbHalSetTraceMask	46
UsbLibGetStatusStr	47
4.2 API Call Back Functions	48
USBLIB_SETUP_EVENT	49
USBLIB_SETUP_DATA_TRANSFERRED	50
USBLIB_DEVICE_EVENT	51
USBLIB_ENDPOINT_EVENT	52
USBLIB_START_OF_FRAME	53
USBLIB_TRANSFER_COMPLETION	54
4.3 Structures	55
USBLIB_DESCRIPTOR	56
USBLIB_EP_CFG	58
USBLIB_CONFIGURATION	59
USBLIB_CALLBACKS	60
USBLIB_BUFFER_DESCRIPTOR	62
4.4 Error Codes	64
USBLIB_STATUS_SUCCESS (0x0000L)	64
USBLIB_STATUS_ERROR (0x0001L)	64
USBLIB_STATUS_CANCELED (0x0002L)	64
USBLIB_STATUS_TRANSMISSION_ERROR (0x0003L)	64
USBLIB_STATUS_BUFFER_OVERFLOW (0x0004L)	64
USBLIB_STATUS_BUSY (0x0005L)	64
USBLIB_STATUS_INVALID_PARAM (0x0006L)	64
USBLIB_STATUS_PENDING (0x0007L)	64

5 Demo Application	65
5.1 USB Interface	65
5.2 Initial Steps	65
5.3 Configuration	66
5.4 Performance	66
5.5 Program size	67
5.6 Summary	67
6 Configuration of the Library	69
7 Related Documents	71
Index	73

1 Introduction

FUFA is a generic Universal Serial Bus (USB) firmware library for the 16-bit serie FUJITSU MB90330 and MB96330 and the 32-bit serie FUJITSU MB91660. It covers the entire USB function of the microcontroller and provides a convenient to use software API.

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

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 programming.

2 Overview

The FUFA library 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.

2.1 Compiler

Depending from the used microcontroller different compiler are in use.

- MB90330 serie: F2MC-16 Family SOFTUNE Workbench
- MB96330 serie: F2MC-16 Family SOFTUNE Workbench
- MB91660 serie: FR Family SOFTUNE Workbench

The FUFA library can be used to implement class conform device interfaces as well as vendor defined interfaces.

2.2 Features

The FUFA library provides the following features:

- full speed (12 Mbps) is supported, corresponds to USB Full Speed.
- handles all USB standard request without some automatic responses to standard commands by the chip
- supports all data transfer types: control, bulk, interrupt and isochronous (some USB functions does not support isochronous transfer)
- supports Class- and Vendor specific requests
- provides static USB events to the application, e.g. Reset, SetConfiguration, Resume and Suspend
- simple to use software interface
- supports DMA data transfer (not on all supported chips)
- interrupt driven model

If the FUFA library detects a non standard SETUP request and processing of vendor requests through the user are enabled, the library calls a routine to inform the user about a SETUP request. If the processing of user vendor requests are disabled the library handles all none standard SETUP requests.

2.3 Restrictions

Some restrictions that apply to the FUFA firmware library are listed below.

- The used microcontroller allow to work as USB Host or USB Function. It is not possible with the library to exit from the USB Function during runtime and to switch to the USB Host.
- It is not allowed to inhibit USB interrupt execution for more than some microseconds. Otherwise it cannot be guaranteed to respond descriptor requests correctly in time.
- An "Abort SETUP request" while in data IN stage is not recognized automatically. EP0 may become unusable.
- "Clear Feature Endpoint Stall" is not recognized by the USB function. Transfer data may get lost.
- The function library supports only one USB configuration at a time.
- Only one alternate setting (index 0) is supported.

3 Architecture

Figure 1 shows the FUFA library architecture.

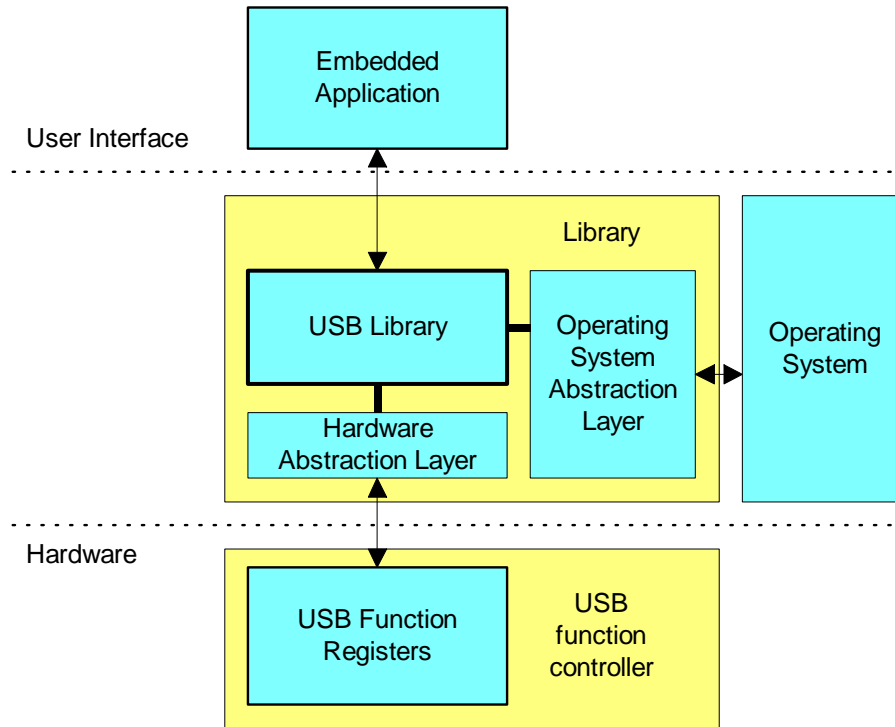


Figure 1: FUFA Software Architecture

The following modules are shown in Figure 1:

- The Embedded Application is the software running on the MB90330, MB96330 and MB91660 series. It uses the FUFA library to communicate via USB with the PC.
- The FUFA Library is divided into three parts: The USB Library, the Hardware Abstraction Layer (HAL), and the Operating System Abstraction Layer (OSAL). The USB Library handles the standard USB requests and the data transfer on an abstract level. The HAL performs the access to the registers of the USB function. The OSAL is the interface to the operating system. It requires a mechanism for synchronization, a debug print function, and some helper functions. It can be easily adapted to projects without a operating system.
- The Hardware contains the USB related register and the physical connection to the USB connector.

This document describes the FUFA interface and the interface of the OSAL. The interface between the USB library and the HAL is not described.

3.1 Typical Program Flow

- At first the application must call the function **UsbLibInitialize**. The application passes the USB descriptors, a USB configuration structure, and some call back function pointers to this routine. With this parameters the run time configuration of the FUFA library is complete.
- The next step is to wait for Vbus and to switch the 1,5k pull up resistor in the D+ line with a 3.3V source. If the device is USB bus powered the function **UsbLibEnable** do not wait for Vbus and switch the 1.5k pull up direct to 3.3V. If it is a self powered device **UsbLibEnable** enables a interrupt that do detect Vbus or the user must install a interrupt that checks Vbus.
- After detection of the USB configuration request the embedded application can start reading and writing on the USB interface by using the functions **UsbLibRead** and **UsbLibWrite**. The following modules are shown in Figure 2:
- If needed the embedded application can perform a virtual unplug of the USB device by calling the function **UsbLibDisable**.

Figures 2 and 3 shows the program flow for reading and writing.

**Program Flow sheet
for use of UsbLibRead() and UsbLibWrite() in the main()
context**

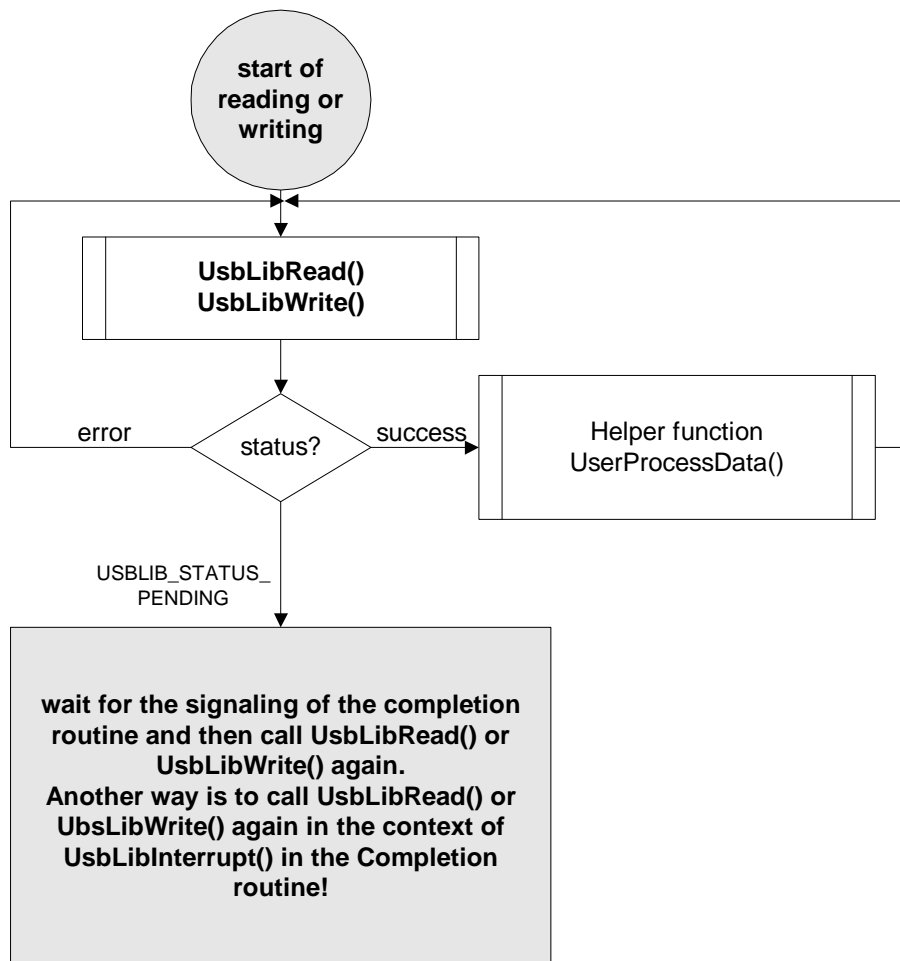


Figure 2: UsbLibRead and UsbLibWrite Program Flow in the main() context

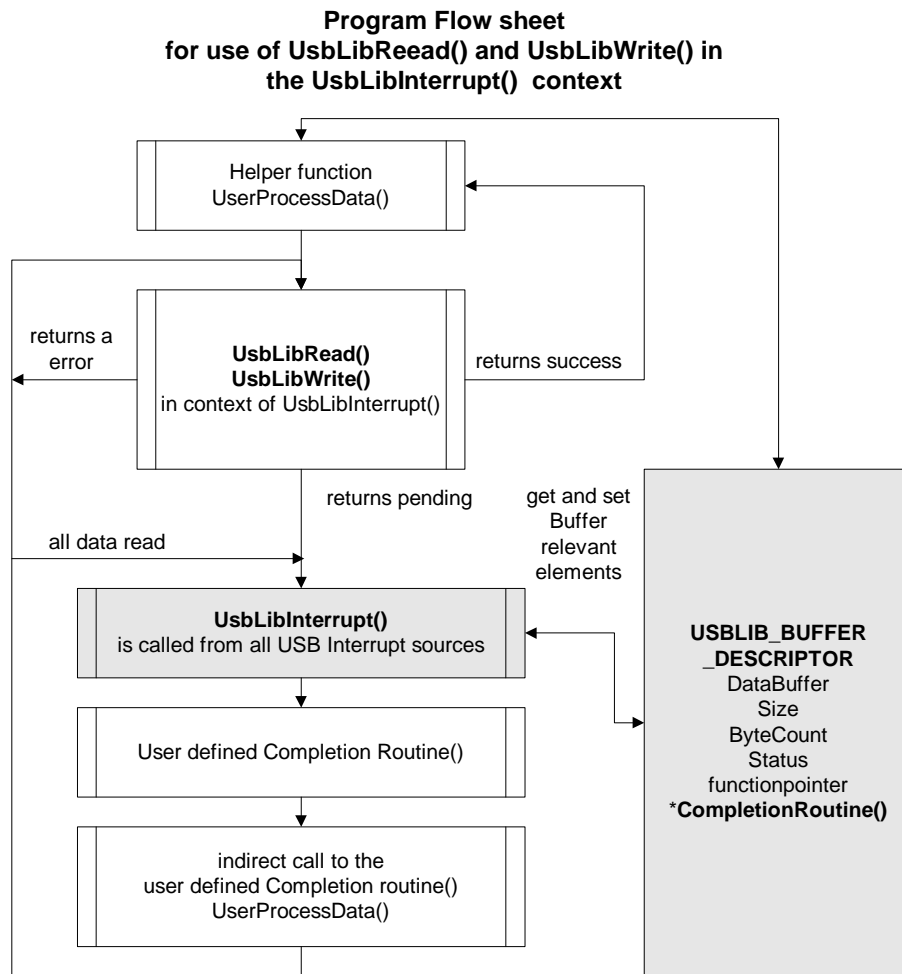


Figure 3: UsbLibRead and UsbLibWrite Program Flow in the UsbLibInterrupt() context

3.2 Integration of the Library into an Application

The library does not require any external libraries or function calls. The OSAL layer defines some prototypes that must be implemented from the user, see also files `<osal_impl.c>` and `<osal_impl.h>` in the platform directories. The header file `<func_api.h>` contains the definition of function prototypes, structures and status codes. It should be included by the embedded application. The other source code files should be added to the project of the embedded application. The data and code size of the library are describe in the section 5.5

3.3 Compile time Configuration

Some parameters can be defined at compile time. Please refer to the comments in the file `<func_conf.h>`.

3.4 Library Interrupt Handler

All needed USB specific interrupt routines are implemented in the FUFA library. To make this routines compiler and system independent they are not declared as interrupt routines. The user must implement the `interruptservice` routine(s), see also `vectors.c` in the platform directory, and must call the specific library functions. The IRQ level of all USB interrupts and other interrupts that uses the USB library must have the same IRQ level or if different levels are used, nested interrupts are forbidden. Because the handshake phase of endpoint zero and a stalled setup request is time critical the ISR that handles the control endpoint must return very quickly and should have a high interrupt periority in the system. This includes also preventing of larger traces or operations in callback functions.

3.5 Synchronization

The library has internal data structures and the USB function library requires special sequences which must not be interrupted. From this point of view the library is not reentrant. To synchronize the code execution in the library several methods are possible. A operating system may provide special objects like critical sections or semaphores. To synchronize code without an operating system typically some or all interrupts are disabled. Each of this method may have drawbacks to the application.

To understand the synchronization the internal operation of the library is important. All call back functions are called in the context of the FUFA function `UsbLibInterrupt()`. `UsbLibInterrupt()` is called from the USB interrupt service routines so all call back functions are running also in the context of the USB interrupt service routines. Call back functions are never called in another context.

The library provides in the OSAL layer two two macros `OSALEnter` and `OSALLeave` which are used in the library to synchronize the the library code. Synchronize means that if a library function is called from the application during the run time the USB host interrupt is disabled. It is not allowed to call library functions that wait for an USB event in an interrupt context with a higher or equal priority as the USB host interrupt(that would cause to a deadlock). In the programming interface the allowed caller process context is declared.

The library makes sure, that after each call of Enter a Leave function is called. The following information applies to the implementation of this macros in the demo application. The Enter macro set the interrupt level to a level where all USB interrupts are disabled. The Leave macro restore all interrupt levels. The global interrupt is only disabled for a short time if the interrupt level is changed. The global interrupt is always enabled if a API function is called. The once not synchronized API function is **UsbLibInitialize**. It does not change the global interrupt flag. Nested calls of Enter of Leave are possible. The macros can be adapted in one of the following ways.

3.5.1 Without Operating System

The Enter function stores the state of the interrupt level (depends from the used controller) and disable all interrupts. The Leave function restores the the interrupt level. To allow nested calls a reference counter has to be implemented.

3.5.2 With Operating System

The Enter function requests a synchronization event from the operating system. The synchronization event must allow to be entered multiple times by the same thread. If the object is in use and requested by a different thread the current thread must be suspended. The Leave function releases the synchronization event.

3.6 Data Transfer and Performance

For data endpoints the FUFA library supports continuous mode and DMA to achieve a high data through put. The library uses only one buffer per endpoint to save memory. The following discussion shows that one buffer is sufficient to give the embedded application enough time for reaction.

3.6.1 Transfer Device to PC

The library copies as much as possible data in the FIFO of the micro controller. The library completes the buffer if the last content is copied into the FIFO. At this point of time the content of the buffer is not yet completely submitted to the PC. The embedded application should be able to re-submit a new buffer with data before the data in the FIFO is transmitted to the PC. This time interval is determined by the size of the FIFO. On interrupt and isochronous pipes only one transmission per frame is performed. The transmission buffer of the used USB function is double buffered so the embedded application has about 2 frames time to refill these buffers.

3.6.2 Transfer PC to Device

After the device is configured the library enables all data endpoints. This enables the PC to transfer data into the FIFO's of OUT endpoints. Now the device can receive data. At this point the device can receive data until the free length of bytes in the FIFO (maximum free length is the maximum packet size * 2) is zero. So the device can read data after configuration without submit a buffer for reading.

If the embedded application now submit a buffer for reading the library checks the FIFO of the endpoint and copy the data to the user buffer. Are all bytes received the completion routine is called. If a buffer is completed the FIFO of the endpoint should be empty until the host send the next data.

3.7 Class and Vendor Requests

USB allows to use the endpoint 0 for class or vendor specific requests. Such a setup request consists of 3 phases:

- Setup phase,
- Data phase,
- Handshake phase.

The setup phase is always sent by the PC and it contains 8 bytes of data. Please refer to the USB specification for more details. 5 bytes of this setup packet can be defined by the user. Furthermore the setup contains the direction and length of the data phase. If the length field is set to zero the data phase is skipped.

The data phase on Windows PC's is limited to 4096 bytes. Other operating systems may allow a data phase up to 64 KB.

The handshake phase allow the device to acknowledge or stall the request. If the device cannot handle the setup request or the data transferred in the data phase, it can return this error to the PC software by stalling the endpoint.

The FUFA library supports all kinds of class and vendor requests. It implements two methods to handle the data phase. One method is called FIFO-based and the other is called buffer based. The embedded application can handle each request with a data phase with one of these methods. If the buffer based method is used the application passes a buffer which is large enough to handle the complete data phase. This make the handling for the embedded application easy but requires a lot of memory. If the FIFO-based method is used the application uses a buffer with the FIFO size of the control endpoint and transfers the data in FIFO sized chunks. Both methods are discussed in detail in the following sections. All information about a new SETUP packet or that data are transferred are sent to the user with callback functions that runs in the interrupt context.

3.7.1 Error Handling

The application can abort each setup request. That means the expected sequence setup, data, and handshake can be interrupted. The embedded application must be able to handle a new setup in each state. If the PC starts a new setup the old setup is discarded.

3.7.2 Class and Vendor Requests without data phase

The library calls the function **USBLIB_SETUP_EVENT** and passes the 8 bytes of the setup to the embedded application. It decodes and performs the request and enables the status phase by calling the function **LibUsbSetupHandshake**. It passes a flag to this function to indicate

if the request should be acknowledged or stalled. If the handshake was sent the library calls **USBLIB_SETUP_HANDSHAKE_COMPLETE**. A request without data phase is only allowed if the request direction bit is zero (Vendor OUT Request). To send short messages it is very efficient to use this request with the free setup fields wValue and/or wIndex.

3.7.3 Class and Vendor Requests with data phase from PC to Device

The library calls the function **USBLIB_SETUP_EVENT**. The embedded application decodes the request. If the request cannot be handled the embedded application can stall the request by calling **LibUsbSetupHandshake**. If the request should be handled FIFO-based, the application calls the function **LibUsbSetupBuffer** with the flag **USBLIB_FIFO_BASED** until a short packet is received or the complete length of the request is transferred. If the request should be handled buffer based, the embedded application calls the function **LibUsbSetupHandshake** one time with a buffer size equal to the length field of the setup request. The library calls the function **USBLIB_SETUP_DATA_TRANSFERRED** each time a buffer is filled. If all data has been transferred the application processes the data and enable the status phase with acknowledge or stall.

3.7.4 Class and Vendor Requests with data phase from Device to PC

Such a request is processed in the same way as a setup request with data phase from PC to Device is processed. The embedded application can return the requested amount of data or less. The minimum data length of this requests must for be always greater then one. If the embedded application uses the FIFO based method it must call the function **LibUsbSetupBuffer** the last time with a buffer size smaller than the FIFO size. The last call can be performed with a length of zero if the returned amount of data is less than the requested and the returned data size is a multiple of the control endpoint FIFO size. The control endpoint FIFO size can vary between the different USB function controller. The default size of the used USB function controller is 64 byte.

E.g. the PC requests 1024 bytes and the embedded application wants to return 64 bytes with a FIFO size of endpoint 0 of 64 bytes. The embedded application calls the function **LibUsbSetupBuffer** one time with a size of 64 and one time with a size of 0.

If the buffer based method is used the library takes care to send a short packet if it is required. In the example above the application must call one time the function **LibUsbSetupBuffer** with the buffer size 64.

3.8 Hardware Requirements

The following requirements must be fulfilled by the hardware.

- Between the USB line D+ and +3,3V must be a resistor with 1.5 k that is controled with an external transistor and connected to the HCONX pin. This enables the soft connect with the function **UsbLibEnable**. Also if the USB device is bus powered the HCONX pin must control this resistor.
- Only the MB90330 serie has include a Vbus detection. If use another chip serie the voltage line of the USB connector must be connected with an free 5 Volt tolerant input pin. Then a

interrupt routine or other routine must be installed that checks the Vbus pin state, refer also <hw_support.c> for a Vbus detection routine.

4 Programming Interface

4.1 API Functions

This section describes the API functions which are called by the embedded application.

LibUsbSetupBuffer

Call this function to pass a buffer for the data phase of a setup request.

Definition

```
USBLIB_STATUS  
LibUsbSetupBuffer(  
    unsigned int BufferSize,  
    void* Buffer,  
    unsigned char Flags  
);
```

Parameters

BufferSize

This field contains the size of the buffer. The size of the buffer must be fulfill the following conditions:

Direction IN and USBLIB_FIFO_BASED not set

The buffer size must be less or equal to the requested length.

Direction IN and USBLIB_FIFO_BASED set

The buffer size must be less or equal to the FIFO size. The last call of this function must be called with a buffer size less than the FIFO size. Zero is allowed. The sum of all buffer sizes used for one setup request must be less or equal to the to the requested length.

Direction OUT and USBLIB_FIFO_BASED not set

The buffer size must be grater than or equal to the requested length.

Direction OUT and USBLIB_FIFO_BASED set

The buffer size should always equal to the FIFO size. The end of the data phase is signaled by a short packet or if the sum of all transfers has reached the length parameter. The PC should never send more data than announced through the requested length.

Buffer

This field contains a pointer to data transfer buffer. The caller must provide the storage. The storage must be permanent until the function `USBLIB_SETUP_DATA_TRANSFERRED` is called. This pointer can be NULL if the `BufferSize` is 0. In this case the library sends a zero length data packet.

Flags

This field contains zero or the `USBLIB_FIFO_BASED` flag. If the `USBLIB_FIFO_BASED` flag is set the data exchange between the library and the embedded application takes place on the base of FIFO sized buffers. The function `LibUsbSetupBuffer` and `USBLIB_SETUP_DATA_TRANSFERRED` may be called repeatedly. This enables the embedded application to handle large data phases with a small memory usage. This is possible if the data can be processed sequentially.

Comments

This function is called in the context of the call back functions `USBLIB_SETUP_EVENT` or `USBLIB_SETUP_DATA_TRANSFERRED` if the length parameter in the setup request is greater than 0.

See Also

`USBLIB_SETUP_EVENT` (page 49)

`USBLIB_SETUP_DATA_TRANSFERRED` (page 50)

`LibUsbSetupHandshake` (page 26)

LibUsbSetupHandshake

Call this function to enable the handshake phase of a setup request.

Definition

```
USBLIB_STATUS  
LibUsbSetupHandshake(  
    unsigned char Flags  
);
```

Parameter

Flags

This field contains the flag `USBLIB_STALL_EP0` or 0. If the embedded application cannot handle the request it should set the flag `USBLIB_STALL_EP0`. The PC software gets a special error code which indicates that the request was not handled by the device. Hints if the flag `USBLIB_STALL_EP0` is used. Because of limitation of hardware the stall bit of the control endpoint is not reset automatically with the next `SETUP` token. This bit is reset in the control endpoint interrupt routine of the library. If the interrupt latency of the control endpoint is very bad (too many traces or other interrupts breaks the current USB function interrupt) it is possible that the next `SETUP` data stage is stalled. For that reason the `USBLIB_STALL_EP0` flag should be used with caution.

Comments

This function is called in the context of the call back functions `USBLIB_SETUP_EVENT` if the length parameter in the setup request is equal to 0. This function is called in the context of the callback function `USBLIB_SETUP_DATA_TRANSFERRED` if all required data has been transferred.

See Also

[USBLIB_SETUP_EVENT](#) (page 49)

[USBLIB_SETUP_DATA_TRANSFERRED](#) (page 50)

[LibUsbSetupBuffer](#) (page 24)

UsbLibInitialize

This function must be called to initialize the library.

Definition

```
USBLIB_STATUS
UsbLibInitialize(
    USBLIB_DESCRIPTOR* Descriptors,
    USBLIB_CONFIGURATION* Configuration,
    USBLIB_CALLBACKS* Callbacks,
    unsigned int Flags
);
```

Parameters

Descriptors

This structure contains pointers to USB descriptors. The USB descriptors must be designed compliant to the USB specification. The content of the descriptors must agree with the configuration data. The storage for this data structure must be permanent. It can be placed in RAM or FLASH.

Configuration

This structure contains the USB configuration. See [USBLIB_CONFIGURATION](#) and [USBLIB_EP_CFG](#) for details. The storage for this data structure must be permanent. It can be placed in RAM or FLASH.

Callbacks

This structure contains call back function pointers. Each pointer is optional and can be NULL. See [USBLIB_CALLBACKS](#) for details. The storage for this data structure must be permanent. It can be placed in RAM or FLASH.

Flags

This parameter contains a or'ed combination of the following flags.

USBLIB_USE_INTERRUPTS indicates that the library is used in interrupt mode.

Return Value

The function returns one of the following status codes:

USBLIB_STATUS_SUCCESS if the initialization was successful.

USBLIB_STATUS_INVALID_PARAM if a parameter was invalid.

Comments

This function must be called one time after power on reset to initialize the library. It is recommendable to check the return values before enable the usb device.

See Also

USBLIB_CONFIGURATION (page 59)

USBLIB_EP_CFG (page 58)

USBLIB_CALLBACKS (page 60)

UsbLibEnable

This function enables the USB interface.

Definition

```
USBLIB_STATUS  
UsbLibEnable( );
```

Return Value

If USB Bus Power is detected on the USB Connector the USB Function is attached to the USB Bus else an interrupt is enabled for detecting USB Bus Power.

Comments

If this function is successfully called, the PC loads the kernel device driver and enumerates the USB device. The embedded application should wait until the call back function **USBLIB_DEVICE_EVENT** with the event USBLIB_CONFIGURE is called. Then the application can start the data transfer by calling **UsbLibRead** and **UsbLibWrite**.

See Also

UsbLibInitialize (page 27)

UsbLibDisable (page 30)

UsbLibRead (page 33)

UsbLibWrite (page 35)

UsbLibDisable

This function disables the USB interface.

Definition

```
USBLIB_STATUS  
UsbLibDisable( ) ;
```

Return Value

The function fails if the library is not initialized.

Comments

This function causes a virtual unplug of the device by switch off the 1,5k resistor from the 3.3V. The kernel driver on the PC is unloaded. All submitted buffers will completed with the cancel status. All USB Interrupts inclusive the resume interrupt will 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 **UsbLibEnable**. All pending requests are canceled.

See Also

UsbLibInitialize (page 27)

UsbLibEnable (page 29)

UsbLibWakeupPC

This function wakes up a suspended USB bus and can be used to wake up a PC that is in stand by mode.

Definition

```
USBLIB_STATUS  
UsbLibWakeupPC( );
```

Return Value

The function fails if the device is not in suspended state.

Comments

This function must be called to wake up a suspended USB device. This function drives for a certain time a resume signal to the USB host. This function does work only if the USB device has been received a SetFeature RemoteWakeupEnable request.

See Also

UsbLibInitialize (page 27)

UsbLibEnable (page 29)

UsbLibGetFrameNumber

This function returns the current USB frame number.

Definition

```
unsigned int  
UsbLibGetFrameNumber ( ) ;
```

Return Value

The return value of the function is undefined if the library is not initialized and the device is not yet configured. The return value is the last USB frame number with 11 valid bits. The result is in the range between 0 and 2047.

Comments

The frame number may be useful to synchronize several devices.

See Also

UsbLibInitialize (page 27)
UsbLibEnable (page 29)
USBLIB_START_OF_FRAME (page 53)

UsbLibRead

This function submits a buffer to the driver which receives data from the PC.

Definition

```
USBLIB_STATUS
UsbLibRead(
    unsigned char Endpoint,
    USBLIB_BUFFER_DESCRIPTOR* BufferDesc
);
```

Parameters

Endpoint

This parameter contains the endpoint address with direction bit. For a read request the direction bit 0x80 is always zero.

BufferDesc

This is the pointer to the buffer descriptor. The caller provides the storage for the buffer and the data. The storage must be persistent until the completion routine is called or the function returned with a status code different to USBLIB_STATUS_PENDING.

Return Value

The function can return one of the following status codes:

USBLIB_STATUS_SUCCESS: The buffer was successfully processed by the library. The buffer contains valid data. This status is returned if the data was stored in the FIFO of the hardware before this function has been called. If this status is returned the completion routine is never called.

USBLIB_STATUS_PENDING: The buffer was submitted successfully to the library. The library cannot complete the buffer immediately because a DMA transfer has been started, no data are in the FIFO, or the amount of data in the FIFO is less than the buffer size and the last packet was no short packet.

USBLIB_STATUS_BUSY is returned because a different buffer is already submitted to the library. The library can handle only one buffer for each endpoint.

USBLIB_STATUS_INVALID_PARAM is returned if the endpoint is not valid.

Comments

If this function returns with the status **USBLIB_STATUS_PENDING** the buffer descriptor and the data memory is owned by the library. The library returns the buffer descriptor and the buffer to the application by calling the function **USBLIB_TRANSFER_COMPLETION**. The pointer to the completion function is passed in the buffer descriptor.

If the function returns with a different status code as `USBLIB_STATUS_PENDING` the completion function is never called. This makes sure that the completion function is called only in the context of a call to the function `UsbLibInterrupt`.

If the remaining buffer size is smaller than the size of the received data bytes, the status `USBLIB_STATUS_BUFFER_OVERFLOW` is returned to the completion routine. The `Count` parameter contains the number of bytes which are received so far.

The buffer is returned if it is filled completely or if a short packet is received. In the time interval where the next buffer is prepared the IC can transfer the data from the PC to the FIFO. This enables a continuous data transfer.

See Also

UsbLibAbort (page 37)

USBLIB_TRANSFER_COMPLETION (page 54)

UsbLibWrite (page 35)

USBLIB_DEVICE_EVENT (page 51)

UsbLibWrite

This function submits a buffer to the driver which transfers data to the PC.

Definition

```
USBLIB_STATUS
UsbLibWrite(
    unsigned char Endpoint,
    USBLIB_BUFFER_DESCRIPTOR* BufferDesc
);
```

Parameters

Endpoint

This parameter contains the endpoint address with direction bit. For a write request the direction bit is always 0x80.

BufferDesc

This is the pointer to the buffer descriptor. The caller provides the storage for the buffer and the data. The storage must be persistent until the completion routine is called or the function returned with a status code different to USBLIB_STATUS_PENDING.

Return Value

The function can return one of the following status codes:

USBLIB_STATUS_SUCCESS: The buffer was successfully processed by the library. The data of the buffer has been transferred to the FIFO immediately. If this status is returned the completion routine is never called.

USBLIB_STATUS_PENDING: The buffer was submitted successfully to the library. The library cannot complete the buffer immediately because a DMA transfer has been started or the data cannot completely copied into the FIFO.

USBLIB_STATUS_BUSY is returned because a different buffer is already submitted to the library. The library can handle only one buffer for each endpoint.

USBLIB_STATUS_ERROR is returned if the endpoint is not valid or the endpoint is stalled.

Comments

If this function returns with the status **USBLIB_STATUS_PENDING** the buffer descriptor and the data memory is owned by the library. The library returns the buffer descriptor and the buffer to the application by calling the function **USBLIB_TRANSFER_COMPLETION**. The pointer to the completion function is passed in the buffer descriptor.

If the function returns with a different status code as `USBLIB_STATUS_PENDING` the completion function is never called. This makes sure that the completion function is called only in the context of a call to the function `UsbLibInterrupt`.

If the remaining buffer size is smaller than the size of the received data bytes, the status `USBLIB_STATUS_BUFFER_OVERFLOW` is returned to the completion routine. The `Count` parameter contains the number of bytes which are received so far.

The buffer is returned if it is copied completely to the FIFO. In the time interval where the next buffer is prepared the IC can transfer the data from the FIFO to the PC. This enables a continuous data transfer.

See Also

UsbLibAbort (page 37)

USBLIB_TRANSFER_COMPLETION (page 54)

UsbLibRead (page 33)

USBLIB_DEVICE_EVENT (page 51)

UsbLibAbort

This function cancels a buffer which was previously successful submitted.

Definition

```
USBLIB_STATUS
UsbLibAbort(
    unsigned char Endpoint,
    USBLIB_BUFFER_DESCRIPTOR** BufferDesc
);
```

Parameters

Endpoint

This parameter contains the endpoint address with direction bit.

BufferDesc

This parameter contains a pointer to the buffer descriptor which was pending or NULL.

Return Value

The function can return one of the following status codes:

USBLIB_STATUS_SUCCESS: The buffer was successfully canceled.

USBLIB_STATUS_BUSY: The endpoint is busy and can not return the buffer. Repeat the call of UsbLibAbort after a timeout of 2ms.

USBLIB_STATUS_ERROR: If the endpoint is not valid or or the endpoint is stalled.

Comments

If this function is called successful the buffer is aborted and the status USBLIB_STATUS_CANCELED is set in the buffer descriptor. The Count parameter contains the number of bytes transferred so fare. The endpoint is not disabled. That means the data exchange between the FIFO and the PC can continue.

See Also

[UsbLibRead](#) (page 33)

[UsbLibWrite](#) (page 35)

[USBLIB_TRANSFER_COMPLETION](#) (page 54)

UsbLibSetStall

This function sets the state of an data endpoint to STALL.

Definition

```
USBLIB_STATUS  
UsbLibSetStall(  
    unsigned char Endpoint  
);
```

Parameter

Endpoint

This parameter contains the endpoint address with direction bit.

Return Value

The function can return one of the following status codes:

USBLIB_STATUS_SUCCESS: The operation was successful.

USBLIB_STATUS_INVALID_PARAM is returned if the endpoint is not valid.

Comments

If the endpoint is set to the STALL state, it returns STALL token on the USB to all requests. This function should only called if the function is in the configured state and no buffer is submitted to this endpoint. Because of limitation of hardware the STALL bit is not reset automatically if the host sends a clear feature endpoint STALL. Instead **UsbLibClearStall** has to be called. With the next clear feature endpoint STALL command the STALL bit is cleared.

See Also

UsbLibClearStall (page 39)

UsbLibClearStall

This function initializes the hardware to release the STALL state of an endpoint. The endpoint STALL state is released with the next Clear Feature Endpoint Halt command.

Definition

```
USBLIB_STATUS  
UsbLibClearStall(  
    unsigned char Endpoint  
);
```

Parameter

Endpoint

This parameter contains the endpoint address with direction bit.

Return Value

The function can return one of the following status codes:

USBLIB_STATUS_SUCCESS: The operation was successful.

USBLIB_STATUS_INVALID_PARAM is returned if the endpoint is not valid.

Comments

If this endpoint state is cleared the endpoint performs normal data transfers after receiving of a Clear Feature Endpoint Stall command. See also [UsbLibSetStall](#) for more information.

See Also

[UsbLibSetStall](#) (page 38)

UsbControlEndpointInterrupt

USB function interrupt routine for control endpoints

Definition

```
void  
UsbControlEndpointInterrupt(  
    void  
    );
```

Parameter

none

Return Value

none

UsbDataEndpointInterrupt

USB function interrupt routine for USB control events such SUSPEND

Definition

```
void  
UsbDataEndpointInterrupt(  
    void  
);
```

Parameter

none

Return Value

none

Comments

This function is called if +Vusb on the USB connector switch off or switch on, a SUSPEND,a RESUME,a USB Bus reset,a start of frame or a set configuration request is detected.

UsbFunctionInterrupt

USB function interrupt routine for data endpoints

Definition

```
void  
UsbFunctionInterrupt (  
    void  
);
```

Parameter

none

Return Value

none

Comments

This function is called if the function has new data or the last data has been successful sent to the host.

UsbShortPacketInterrupt

USB function interrupt routine if a short packet is received

Definition

```
void  
UsbShortPacketInterrupt(  
    void  
);
```

Parameter

none

Return Value

none

Comments

This function is called if a short packet is detected on the USB.

UsbLibSetTraceMask

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

Definition

```
void  
UsbLibSetTraceMask(  
    unsigned long Mask  
);
```

Parameter

Mask

Specifies the new trace mask to be set.

Comments

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

Bit positions of trace mask are assigned as described below.

DBG_ERR

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

DBG_WRN

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

DBG_INFO

Informational messages.

DBG_FUNC

Function names.

DBG_EP0

Print informations if a SETUP request is processed from the library. Because the trace is time critical it should be not used.

DBG_EP0_FLAGS

This trace prints out the state of control interrupt request flags. Because the trace is time critical it should be not used.

DBG_DUMP_EP0

Dumps data bytes of control stages .

DBG_EP

Data endpoint informations.

DBG_DUMP_EP

Dumps data of data endpoints.

DBG_DMA

Detailed DMA request informations on data endpoints.

By default, the bits **DBG_ERR** and **DBG_WRN** are set and all other bits are cleared in the trace mask.

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

Example:

```
UsbLibSetTraceMask(  
DBG_BIT_MASK(DBG_ERR) | DBG_BIT_MASK(DBG_WRN)  
);
```

In the debug version the trace support is enabled. If trace support is disabled then a call to **UsbLibSetTraceMask** has no effect.

See Also

UsbHalSetTraceMask (page 46)

UsbHalSetTraceMask

This function sets an internal trace mask which filters trace messages produced by the files in the hardware abstraction layer.

Definition

```
void  
UsbHalSetTraceMask(  
    unsigned long Mask  
);
```

Parameter

Mask

Specifies the new trace mask to be set.

Comments

The trace masks are the same as in the function [UsbLibSetTraceMask](#) but a mask named `DBG_SPECIAL` is added. For detailed information see also [UsbLibSetTraceMask](#).

Bit positions of trace mask are assigned as described below.

DBG_SPECIAL

This trace bits is used for testing the library an has different meanings.

See Also

[UsbLibSetTraceMask](#) (page 44)

UsbLibGetStatusStr

This function returns the status as a string constants.

Definition

```
const char*
UsbLibGetStatusStr(
    UBD_STATUS x
);
```

Parameter

x
This parameter specifies the status.

Return Value

An error string is returned.

Comments

This function returns only an error string if the debug version of the USB Bus Driver Core is used (DBG=1).

4.2 API Call Back Functions

This section describes the API functions, which are called by the embedded application and the callback functions which are registered by the embedded application.

USBLIB_SETUP_EVENT

This function is called, if a class or vendor specific setup request has been received.

Definition

```
void
USBLIB_SETUP_EVENT(
    unsigned char* Setup
);
```

Parameter

Setup

This field contains the 8 bytes setup data which are passed with each setup request from the PC to the device. Refer to the USB specification to get more information about the contents of this data.

Comments

A setup transmission can be started at each time by the PC. A new setup request terminates each setup request which was submitted earlier. This typically happens if the PC driver detects a timeout or transmission errors. If the requested length is greater than 0 the embedded application prepares a data buffer and submit it with a call to the function LibUsbSetupBuffer or it can call the function LibUsbSetupHandshake if the Request is a OUT Request with the length parameter equal to 0 to enable the handshake phase. See section "Class and Vendor Requests" for details.

See Also

LibUsbSetupBuffer (page 24)

LibUsbSetupHandshake (page 26)

USBLIB_SETUP_DATA_TRANSFERRED

This function is called, if a data transfer from a vendor or class request which has been started with a call to the function `LibUsbSetupBuffer` has been completed.

Definition

```
void
USBLIB_SETUP_DATA_TRANSFERRED(
    unsigned char * Setup,
    unsigned int Count,
    USBLIB_STATUS Status
);
```

*Parameters***Setup**

This field contains the 8 bytes setup data which are passed with each setup request from PC to device. This data field contains the same setup data which have been passed to the function `USBLIB_SETUP_EVENT`. The embedded application can use this field to identify the requests.

Count

This field contains the number of bytes which has been transferred from or to the buffer.

Status

This field contains `USBLIB_STATUS_SUCCESS` on success or `USBLIB_STATUS_CANCELED` if a new setup request has been started before the current request was finished. Furthermore `USBLIB_STATUS_CANCELED` can be returned if an USB reset occurs during a setup request.

Comments

In this callback function the embedded application can call the function **`LibUsbSetupBuffer`** again if FIFO based method is used. If the buffer based method is used or if the data phase was terminated with a short packet the application must call the function **`LibUsbSetupHandshake`**.

See Also

`USBLIB_SETUP_EVENT` (page 49)

`LibUsbSetupBuffer` (page 24)

`LibUsbSetupHandshake` (page 26)

USBLIB_DEVICE_EVENT

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

Definition

```
void
USBLIB_DEVICE_EVENT (
    unsigned int Event
);
```

Parameter

Event

This field contains the event which is one of the following:

USBLIB_RESET a USB reset has been detected. All USB specific actions are handled by the library. Pending data requests are canceled.

USBLIB_SUSPEND a USB suspend signal has been detected. If the device is bus powered the embedded application should reduce the required current to 0,5 mA (or 2.5 mA for high power devices). It should stop the clock and enable the static interrupt for wakeup. If the device is self powered it depends on the decision of the embedded application if the clock should be stopped.

USBLIB_RESUME a resume signal has been detected. If the clock was turned off it must be reenabled.

USBLIB_CONFIGURE the device has been configured. The data transfer may be started.

USBLIB_UNCONFIGURE the device has been unconfigured. Pending requests are canceled and endpoints are cleared.

Comments

This function is called in the context of `UsbLibInterrupt`. The embedded application can call the functions `UsbLibRead` and `UsbLibWrite` if the function is in the configured state, i.e. the event `USBLIB_CONFIGURE` is signaled. If the events `USBLIB_RESET` or `USBLIB_UNCONFIGURE` are received then the function is always unconfigured.

See Also

[USBLIB_CALLBACKS](#) (page 60)

USBLIB_ENDPOINT_EVENT

This function is called if a endpoint specific event has been detected.

Definition

```
void
USBLIB_ENDPOINT_EVENT(
    unsigned char Endpoint,
    unsigned int Event
);
```

Parameters

Endpoint

This field contains the endpoint address with direction bit where the event is related to.

Event

This field contains the event which is one of the following:

USBLIB_CLEAR_STALL the PC has detected a error during the data transmission and sends a Clear Feature Endpoint Stall to clear the error condition. The library clears the endpoint and restarts the data transmission from the beginning of the current buffer. The embedded application can cancel the buffer to force a different behaviour. Note: The PC software may send a Clear Feature Endpoint Stall at each time, maybe during initialization.

USBLIB_SET_STALL the PC forces the endpoint to go to the STALL state. The library set the endpoint to STALL and suspends normal data transfer.

See Also

USBLIB_CALLBACKS (page 60)

USBLIB_START_OF_FRAME

This function is called if a SOF has been received.

Definition

```
void
USBLIB_START_OF_FRAME(
    unsigned int FrameNumber
);
```

*Parameter***FrameNumber**

This field contains current frame number.

Comments

The frame number has 11 valid bits. To get this call back the library must be used in interrupt mode.

See Also

USBLIB_CALLBACKS (page 60)

USBLIB_TRANSFER_COMPLETION

This function is called, if a read or write operation has been completed.

Definition

```
void
USBLIB_TRANSFER_COMPLETION(
    USBLIB_BUFFER_DESCRIPTOR* BufferDesc
);
```

Parameter

BufferDesc

This is the same value which was passed to the [UsbLibRead](#) or [UsbLibWrite](#) function.

Comments

The callback function is called for each buffer which was previously successful submitted to the library. The buffer can be submitted back to the same endpoint with the functions [UsbLibRead](#) or [UsbLibWrite](#) in the completion routine. To prevent a calling chain the functions [UsbLibRead](#) or [UsbLibWrite](#) can called repeatedly from the completion routine, until the return values from this functions are USBLIB_STATUS_SUCCESS.

See Also

[UsbLibAbort](#) (page 37)

[UsbLibRead](#) (page 33)

[UsbLibWrite](#) (page 35)

[USBLIB_BUFFER_DESCRIPTOR](#) (page 62)

4.3 Structures

This section describes the required structures. Please refer to the documentation of the function to get as much as possible information.

USBLIB_DESCRIPTOR

The USBLIB_DESCRIPTOR structure contains information on the USB descriptors.

Definition

```
typedef struct _USBLIB_DESCRIPTOR{
    USB_DEVICE_DESCRIPTOR* DeviceDescriptor;
    int ConfigurationDescriptorSize;
    USB_CONFIGURATION_DESCRIPTOR* ConfigurationDescriptor;
    int NumberOfStringDescriptors;
    USB_STRING_DESCRIPTOR** StringDescriptors;
} USBLIB_DESCRIPTOR;
```

Members

DeviceDescriptor

This field contains a pointer to the device descriptor.

ConfigurationDescriptorSize

This field contains the complete size of the configuration descriptor in bytes.

ConfigurationDescriptor

The configuration descriptor contains the complete description of the interface and endpoint layout. It consists one configuration descriptor and all required interface, class, and endpoint descriptors. The correctness of the wTotalLength field in the configuration descriptor is very important. A invalid value can cause blue screen on Windows.

NumberOfStringDescriptors

This member contains the number of string descriptors.

StringDescriptors

This member contains a pointer to an array of string descriptors. 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. All characters must be given in UNICODE format. This means each character is two bytes large. The string is not zero terminated.

Comments

The storage for all descriptors must be provided by the caller and must be permanent. The descriptors can be stored in the Flash or Ram memory. The descriptors must be defined correctly and compliant to the USB specification. Otherwise the enumeration on the PC can fail.

See Also

[UsbLibInitialize](#) (page 27)

[USBLIB_CONFIGURATION](#) (page 59)

USBLIB_EP_CFG (page 58)

USBLIB_CALLBACKS (page 60)

USBLIB_EP_CFG

The USBLIB_EP_CFG structure contains information on the configuration of one endpoint.

Definition

```
typedef struct _USBLIB_EP_CFG{
    unsigned char EndpointAddress;
    char EndpointType;
    unsigned char Flags;
    unsigned int MaxPktSize;
} USBLIB_EP_CFG;
```

Members

EndpointAddress

This field contains the endpoint address with direction bit (0x80).

EndpointType

This field contains the endpoint type. It is one of USB_EP_TYPE_CONTROL, USB_EP_TYPE_ISO, USB_EP_TYPE_BULK, or USB_EP_TYPE_INT.

Flags

This field contains a or'ed combination of the following flags:

USBLIB_USE_DMA indicates that a DMA channel should be used for the data transfer from/to this endpoint. It is not possible to specify USBLIB_USE_DMA more than once.

MaxPktSize

This member contains the size of the data packets transferred via USB. It must be equal to the value in the endpoint descriptor.

Comments

The storage for this data structure must be provided by the caller and must be permanent. For each used endpoint such a structure must be provided. The parameters must fit the physical FIFO size.

See Also

[UsbLibInitialize](#) (page 27)

[USBLIB_CONFIGURATION](#) (page 59)

[USBLIB_CALLBACKS](#) (page 60)

USBLIB_CONFIGURATION

The USBLIB_CONFIGURATION structure contains information on the configuration of all used endpoints.

Definition

```
typedef struct _USBLIB_CONFIGURATION{
    unsigned char CfgCount;
    USBLIB_EP_CFG* EpCfg;
} USBLIB_CONFIGURATION;
```

Members

CfgCount

This field contains the number of endpoint configuration structure.

EpCfg

This field contains a pointer to an array of USBLIB_EP_CFG structures.

Comments

The storage for this data structure must be provided by the caller and must be permanent.

See Also

UsbLibInitialize (page 27)

USBLIB_EP_CFG (page 58)

USBLIB_CALLBACKS (page 60)

USBLIB_CALLBACKS

The USBLIB_CALLBACKS structure contains information on call back functions.

Definition

```
typedef struct _USBLIB_CALLBACKS{
    USBLIB_DEVICE_EVENT* DeviceEvents;
    USBLIB_ENDPOINT_EVENT* EndpointEvents;
    USBLIB_SETUP_EVENT* SetupEvent;
    USBLIB_SETUP_DATA_TRANSFERRED* SetupDataTransferred;
    USBLIB_START_OF_FRAME* StartOfFrameEvent;
} USBLIB_CALLBACKS;
```

Members

DeviceEvents

This field contains the function pointer to a function which receives device specific events.

EndpointEvents

This field contains the function pointer to a function which receives endpoint specific events.

SetupEvent

This field contains the function pointer to a function which receives setup specific events.

SetupDataTransferred

This field contains the function pointer to a function which receives events for a class or vendor specific data transfer.

StartOfFrameEvent

This field contains the function pointer to a function which is called if a Start Of Frame token has been received.

Comments

The storage for this data structure must be provided by the caller and must be permanent. Each function pointer has to contain a valid function address or NULL. The embedded application can pass a NULL pointer to a call back function which is not required. But even if no call back function is required this data structure must be passed to the function `UsbLibInitialize`.

See Also

[UsbLibInitialize](#) (page 27)

[USBLIB_DEVICE_EVENT](#) (page 51)

USBLIB_ENDPOINT_EVENT (page 52)
USBLIB_SETUP_EVENT (page 49)
USBLIB_SETUP_DATA_TRANSFERRED (page 50)
USBLIB_START_OF_FRAME (page 53)

USBLIB_BUFFER_DESCRIPTOR

The USBLIB_BUFFER_DESCRIPTOR structure contains information on a data buffer.

Definition

```
typedef struct _USBLIB_BUFFER_DESCRIPTOR{
    void* DataBuffer;
    unsigned int Size;
    unsigned int ByteCount;
    unsigned int Flags;
    USBLIB_TRANSFER_COMPLETION* CompletionRoutine;
    void* Context;
    USBLIB_STATUS Status;
} USBLIB_BUFFER_DESCRIPTOR;
```

Members

DataBuffer

This field contains a pointer to data buffer. The caller must provide the storage. The storage must be permanent until the buffer is returned.

Size

This member contains the size of the buffer. It is not changed by the library.

ByteCount

This member contains the number of valid bytes in the buffer.

Flags

This member contains 0 or the flag USBLIB_SEND_SHORT_PACKET. The flag USBLIB_SEND_SHORT_PACKET can be used with the function UsbLibWrite. If the flag is set the library sends an additional zero length packet if the ByteCount can be divided by the max transfer size of the endpoint. This flag can be used with bulk or interrupt endpoints.

CompletionRoutine

This member contains a function pointer to a completion routine. It must not be NULL.

Context

A caller defined context which is passed to the completion routine. Can be NULL;

Status

Returns the status of the operation. It can be one of the following values:

USBLIB_STATUS_SUCCESS: The operation was completed successfully.

USBLIB_STATUS_CANCELED: The PC has sent a Unconfigure or a Reset or the user has aborted the buffer with UsbLibAbort.

USBLIB_STATUS_TRANSMISSION_ERROR: A hardware transmission error has been occurs. The PC should send a Clear Feature Endpoint Stall request to

clear the error condition. Some error conditions cannot be recognized by the device. The error handling in the application should use the Clear Feature Endpoint Stall which is indicated by the call back function **USBLIB_ENDPOINT_EVENT**.

USBLIB_STATUS_BUFFER_OVERFLOW: A buffer overflow is happening during a **UsbLibRead()**. The buffer size was not a multiple of the **FifoSize** or less than the transferred bytes in the data structure **USBLIB_EP_CFG**. Electrical noise of the signals can cause this error on some USB interfaces.

Comments

This structure is passed by the embedded application to the functions **UsbLibRead** and **UsbLibWrite**. The library changes the contents of the **Buffer**, the **ByteCount**, and the **Status**. It does not change other values.

See Also

UsbLibRead (page 33)

UsbLibWrite (page 35)

UsbLibAbort (page 37)

4.4 Error Codes

USBLIB_STATUS_SUCCESS (0x0000L)

The operation has been successfully completed.

USBLIB_STATUS_ERROR (0x0001L)

The operation was completed with a generic error.

USBLIB_STATUS_CANCELED (0x0002L)

The operation was canceled by the API.

USBLIB_STATUS_TRANSMISSION_ERROR (0x0003L)

A transmission error has been occurs.

USBLIB_STATUS_BUFFER_OVERFLOW (0x0004L)

The amount of data was larger than the buffer size and the buffer size was not a multiple of the FIFO size.

USBLIB_STATUS_BUSY (0x0005L)

An other buffer is currently queued. Re-submit the buffer later again.

USBLIB_STATUS_INVALID_PARAM (0x0006L)

A parameter passed to the function was invalid.

USBLIB_STATUS_PENDING (0x0007L)

The request to read from buffer or to write to buffer is pending.

5 Demo Application

All applications consists of a common code and a platform specific code. The common code is located in the SOURCE directory and the name of this directory is app_XXX (XXX is the short name of the application). The implementation of the application depends from the used platform and is located in the platform directory. For more informations see also the Platform specific Readme in the <XXXREADME.TXT> (XXX is the name of the platform). The demo applications loops received data packets from an OUT data endpoint to an IN data endpoint. The isochronous application contains no loop, instead the USB Host can only read endpoint data.

5.1 USB Interface

All USB descriptors are defined in the module <app_XXX_descriptors.c> and <app_XXX_descriptors.h>. Platform specific defines are located in <func_conf.h>. (XXX is the short name of the application).

The most demo applications have two data endpoints in their USB Interface.

Common values of the USB descriptors:

- One configuration
- One interface
- Two data endpoints (This can be BULK INTERRUPT or ISOCHRONOUS endpoints).

To increase the performance of the demo application the User Buffer Size is 512 byte for all Loop applications. The isochronous IN transfer buffer size is also 512 byte. Because only isochronous IN transactions are available a two byte sequence number is inserted.

5.2 Initial Steps

The DEMO application performs the following steps after reset:

- HW_init() initialize the hardware.
- __EI()
- Optional xx_SetTraceMask()
- UsbLibInitialize()
- UsbLibEnable() If it is allowed the D+ 1.5k pull up is connected with a +3.3V power supply.
- BI_InitDemoApp() This function initialize the demo application.
- The main loop calls in a loop BI_ProcessDemoApp().

5.3 Configuration

In the common demo application folders APP_XXX (XXX is the short name of the application) are all needed USB descriptors defined. This module includes the func_conf.h file that is located in the platform specific folder of the application. The file func_conf.h contains additional defines for the configuration of the FUFA library and the application.

5.4 Performance

This section describes the data throughput. The test setup has the following parameters:

Host:	PC: Celeron 2,4Ghz IntelChipsatz 865PE, test program: USBIOAPP.EXE
USBIOAPP Parameter	
IN Pipe, OUT - Pipe:	
Size of Buffers:	64000
Number of Buffers:	5
File size:	2048
USB controller:	Intel 82801EB Universal Host
USB chip:	Evaluation CPU MB90330
Size of user buffer:	64 or 512 or 2048 bytes
Code optimization:	speed
Firmware:	release

Endpoint Type	Configuration	User buffer size	Data rate in Mbytes/s
Bulk OUT		64 byte	0,405
Bulk OUT		512 byte	0,8
Bulk OUT		2048 byte	0,48
Bulk OUT	DMA	64 byte	0,405
Bulk OUT	DMA	512 byte	0,448
Bulk OUT	DMA	2048 byte	1,018
Bulk IN		64 byte	0,448
Bulk IN		512 byte	0,448
Bulk IN		2048 byte	0,448
Bulk IN	DMA	64 byte	0,512
Bulk IN	DMA	512 byte	1,024
Bulk IN	DMA	2048 byte	1,024

If the demo application is running on a board with the MB90F337 then the following data rates can be measured.

USB chip: MB90F337
Size of user buffer: depends from the testmode
Code optimization: speed
Firmware: release

Endpoint mode	Configuration	User buffer size	Data rate in Mbytes/s
Bulk Loop		512 byte without DMA	0,393
Interrupt Loop	poll. interval=2 without DMA,packetsize=64	512 byte	0,06336
Isochronous IN	poll. interval=1 without DMA,packetsize=256	256 byte	0,253

The polling interval of 2 during the interrupt loop means that every 2ms a 64 byte packet is written to USB and every 2ms is reading an interrupt packet from USB, so the Datarate on the USB is $2 * 32\text{kb/s} = 64\text{kb/s}$.

5.5 Program size

The following table give a short summary for the used code and data size of a short USB application (two endpoints with a 64 bytes per buffer) with and without trace support. The code size depends also from the memory model. The library and the demo application use the medium memory model.

Demo project settings	Functionality	ROM size in bytes	RAM size with stack in bytes
Debug	full functionality with traces	16627	4397
Release	full functionality	7789	4035
Release	without Vendor Requests	7097	3903
Release	without Vendor Requests and DMA	5963	3645

5.6 Summary

The maximum bandwidth for bulk transfers in both directions is 1.1 Mbyte/s. The maximum bandwidth for isochronous transfer depends from the pipe FIFO size. To access the maximum bandwidth a user buffer size between 512 bytes and 2048 bytes and DMA is necessary.

6 Configuration of the Library

The FUFA library is configured with the configuration file `func_conf.h`. The file is located in the implementation folder of any application. Important defines are the followings:

- `DMA_SUPPORT`: Enables the DMA support in the library. Saves memory if not defined.
- `VENDOR`: Enable the vendor request support. To save memory remove `usbvendor.c` from the project.
- `INTERRUPT_LEVEL`: Interrupt Level used for all USB function Interrupts.
- `MAX_ENDPOINTS`: Maximum number of endpoints in the library. To save memory `MAX_ENDPOINTS` can set to the used endpoint number.
- `EP0_MAX_PACKET_SIZE` depends from the used USB controller. Full speed devices uses 8, 16, 32, or 64 bytes, the default value are 64 bytes.

7 Related Documents

- Universal Serial Bus Specification 1.1, <http://www.usb.org>
- Universal Serial Bus Specification 2.0, <http://www.usb.org>
- USB device class specifications (Audio, HID, Printer, etc.), <http://www.usb.org>
- USB 2.0, Hrsg. H. Kelm, Franzi's Verlag, 2001, ISBN 3-7723-7965-6
- USBIO Reference Manual, Version 2.0, <http://www.thesycon.de>

Index

- Buffer
 - Parameter of LibUsbSetupBuffer, 24
- BufferDesc
 - Parameter of USBLIB_TRANSFER_COMPLETION, 54
 - Parameter of UsbLibAbort, 37
 - Parameter of UsbLibRead, 33
 - Parameter of UsbLibWrite, 35
- BufferSize
 - Parameter of LibUsbSetupBuffer, 24
- ByteCount
 - Member of USBLIB_BUFFER_DESCRIPTOR, 62
- Callbacks
 - Parameter of UsbLibInitialize, 27
- CfgCount
 - Member of USBLIB_CONFIGURATION, 59
- CompletionRoutine
 - Member of USBLIB_BUFFER_DESCRIPTOR, 62
- Configuration
 - Parameter of UsbLibInitialize, 27
- ConfigurationDescriptor
 - Member of USBLIB_DESCRIPTOR, 56
- ConfigurationDescriptorSize
 - Member of USBLIB_DESCRIPTOR, 56
- Context
 - Member of USBLIB_BUFFER_DESCRIPTOR, 62
- Count
 - Parameter of USBLIB_SETUP_DATA_TRANSFERRED, 50
- DataBuffer
 - Member of USBLIB_BUFFER_DESCRIPTOR, 62
- Descriptors
 - Parameter of UsbLibInitialize, 27
- DeviceDescriptor
 - Member of USBLIB_DESCRIPTOR, 56
- DeviceEvents
 - Member of USBLIB_CALLBACKS, 60
- Endpoint
 - Parameter of USBLIB_ENDPOINT_EVENT, 52
 - Parameter of UsbLibAbort, 37
 - Parameter of UsbLibClearStall, 39
 - Parameter of UsbLibRead, 33
 - Parameter of UsbLibSetStall, 38
 - Parameter of UsbLibWrite, 35
- EndpointAddress
 - Member of USBLIB_EP_CFG, 58

EndpointEvents
 Member of USBLIB_CALLBACKS, 60

EndpointType
 Member of USBLIB_EP_CFG, 58

EpCfg
 Member of USBLIB_CONFIGURATION, 59

Event
 Parameter of USBLIB_DEVICE_EVENT, 51
 Parameter of USBLIB_ENDPOINT_EVENT, 52

Flags
 Member of USBLIB_BUFFER_DESCRIPTOR, 62
 Member of USBLIB_EP_CFG, 58
 Parameter of LibUsbSetupBuffer, 24
 Parameter of LibUsbSetupHandshake, 26
 Parameter of UsbLibInitialize, 27

FrameNumber
 Parameter of USBLIB_START_OF_FRAME, 53

LibUsbSetupBuffer, 24

LibUsbSetupHandshake, 26

Mask
 Parameter of UsbHalSetTraceMask, 46
 Parameter of UsbLibSetTraceMask, 44

MaxPktSize
 Member of USBLIB_EP_CFG, 58

NumberOfStringDescriptors
 Member of USBLIB_DESCRIPTOR, 56

Setup
 Parameter of USBLIB_SETUP_DATA_TRANSFERRED, 50
 Parameter of USBLIB_SETUP_EVENT, 49

SetupDataTransferred
 Member of USBLIB_CALLBACKS, 60

SetupEvent
 Member of USBLIB_CALLBACKS, 60

Size
 Member of USBLIB_BUFFER_DESCRIPTOR, 62

StartOfFrameEvent
 Member of USBLIB_CALLBACKS, 60

Status
 Member of USBLIB_BUFFER_DESCRIPTOR, 63
 Parameter of USBLIB_SETUP_DATA_TRANSFERRED, 50

StringDescriptors
 Member of USBLIB_DESCRIPTOR, 56

UsbControlEndpointInterrupt, 40

UsbDataEndpointInterrupt, 41

UsbFunctionInterrupt, 42
UsbHalSetTraceMask, 46
USBLIB_BUFFER_DESCRIPTOR, 62
USBLIB_CALLBACKS, 60
USBLIB_CONFIGURATION, 59
USBLIB_DESCRIPTOR, 56
USBLIB_DEVICE_EVENT, 51
USBLIB_ENDPOINT_EVENT, 52
USBLIB_EP_CFG, 58
USBLIB_SETUP_DATA_TRANSFERRED, 50
USBLIB_SETUP_EVENT, 49
USBLIB_START_OF_FRAME, 53
USBLIB_STATUS_BUFFER_OVERFLOW, 64
USBLIB_STATUS_BUSY, 64
USBLIB_STATUS_CANCELED, 64
USBLIB_STATUS_ERROR, 64
USBLIB_STATUS_INVALID_PARAM, 64
USBLIB_STATUS_PENDING, 64
USBLIB_STATUS_SUCCESS, 64
USBLIB_STATUS_TRANSMISSION_ERROR, 64
USBLIB_TRANSFER_COMPLETION, 54
UsbLibAbort, 37
UsbLibClearStall, 39
UsbLibDisable, 30
UsbLibEnable, 29
UsbLibGetFrameNumber, 32
UsbLibGetStatusStr, 47
UsbLibInitialize, 27
UsbLibRead, 33
UsbLibSetStall, 38
UsbLibSetTraceMask, 44
UsbLibWakeupPC, 31
UsbLibWrite, 35
UsbShortPacketInterrupt, 43

x

Parameter of UsbLibGetStatusStr, 47