Thesycon® Systemsoftware & Consulting GmbH

Fujitsu USB Firmware Api FUFA

MB90330 and MB90335 serie

Reference Manual

Version 1.01 **April 20, 2005**

Thesycon® Systemsoftware & Consulting GmbH
Werner-von-Siemens-Str. 2 · D-98693 Ilmenau · GERMANY

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

e-mail: info@thesycon.de
http://www.thesycon.de

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

1 Introduction

FUFA is a generic Universal Serial Bus (USB) firmware library for the 16-bit FUJITSU controller MB90330 and MB90335 serie. It covers the entire USB function of the micro controller 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 USB function block is build into the MB90330 micro controller and handles the lower layer of the USB protocol. The higher layer of the USB protocol must be implemented in the software of the micro controller. The quality of this software implementation is very important to work together with several operating systems and host controllers. By using the FUFA firmware library it is possible to get the USB interface up and running without spending the time and the effort of developing a new USB firmware.

2.1 Compiler

The FUFA library supports the following C-compiler:

- F2MC-16 Family SOFTUNE Workbench V30L31 with fs907s compiler

2.2 Features

The FUFA library provides the following features:

- supports USB 1.1 full speed data transfer
- handles all USB standard request without that are already implemented in the USB function.
- supports all data transfer types: control, bulk, interrupt and isochronous
- 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
- usage of continuous mode and DMA transfer
- interrupt driven model

The FUFA library can be used to implement class conform device interfaces as well as vendor defined interfaces. If use HID descriptors then can do this by implementing a call back routine for the vendor specific request. The callback routine is called if a non standard Get Descriptor request is received. The user must check the setup request and if this the rigth HID descriptor handle the request, see also the handling of vendor requests in the demo firmware.

2.3 Restrictions

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

- It is not allowed to inhibit USB interrupt execution for more than some microseconds. Otherwise it cannot be guarantied 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 by MB90330/90335 Series.
- If DMA is used on an OUT endpoint, then same data package sizes have to be used on host and function software.

3 Architecture

Figure 1 shows the FUFA library architecture.

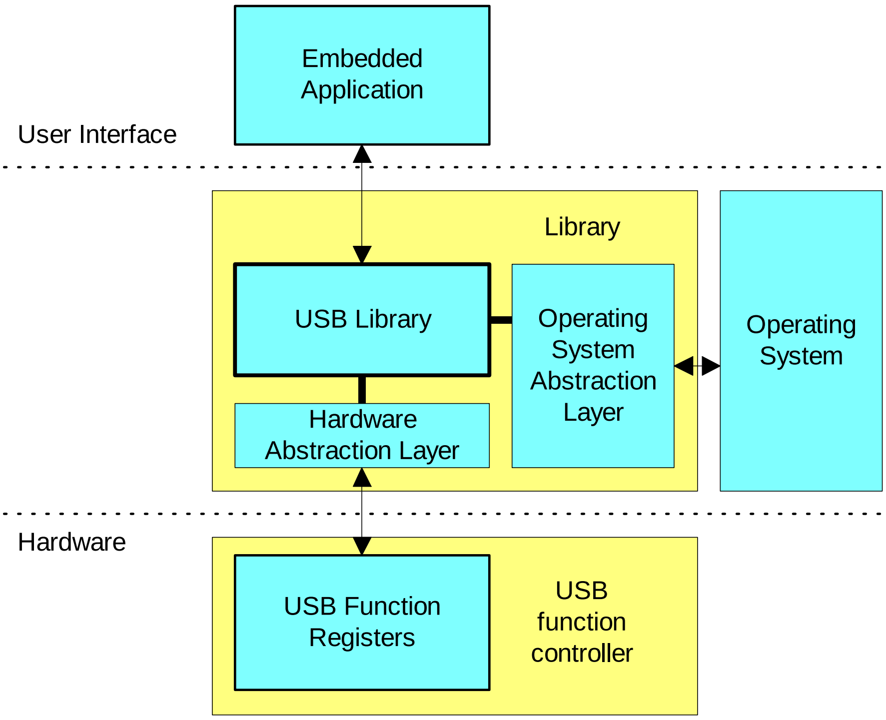

Figure 1: FUFA Software Architecture

The following modules are shown in Figure 1:

- The Embedded Application is the software running on the MB90330 and the MB90335 series. It uses the FUFA library to communicate via USB with the PC.
- The FUFA Library is divided in 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 function **UsbLibEnable** turns on the resistor between the USB D+ line and the 3.3V if +Vusb is detected on the power pin of the USB connector. If Vusb is not detected then the library waits for an Vusb Interrupt to switch the 1,5k resistor to 3,3V. After connection of the 3,3V with the 1,5k resistor the Host starts the enumeration to that device. Finally the Host software configure the device see also **Hardware Requirements**. The embedded user application is informed about that the device is configured.
- After detection of the configuration 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:
- 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.

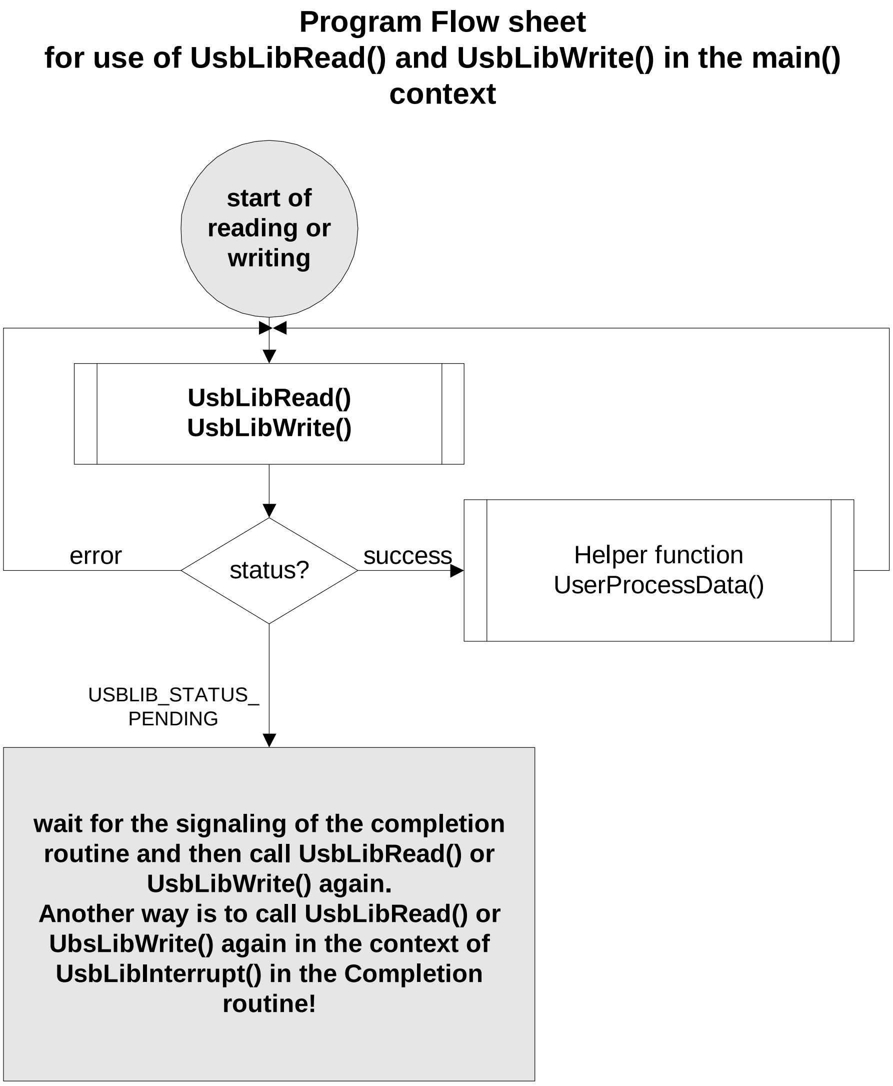

Figure 2: UsbLibRead and UsbLibWrite Program Flow in the main() context

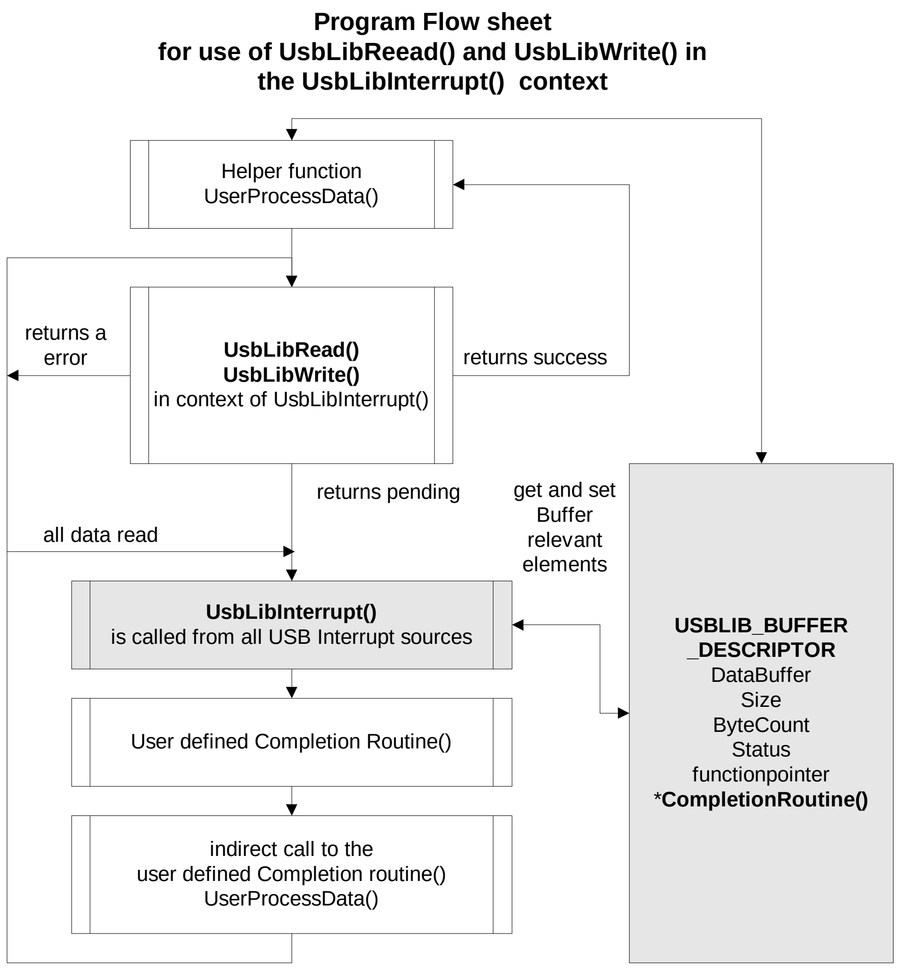

Figure 3: UsbLibRead and UsbLibWrite Program Flow in the UsbLibInterrupt() context

3.2 Integration of the Library into an Application

The library does not require any external libraries or function calls. The OSAL layer defines a debug print function. A system independet implementation of the print function is implemented in the file dbgprint.c. This file requires the include files <string.h> and <stdarg.h>. But the debug print function can be mapped to any other debug print capability of a existing project. 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 USB specific interrupt handler functions and the USB interrupt vector table are implemented in the FUFA library. The IRQ level of the function interrupts have the interrupt level one, so the user can implement a interrupt routine with a higher priority. The user interrupt routines that run at the highest priority must very quickly because a slow user interrupt routine can lead to that the USB device does not correctly enumerate (MB90330 USB specific). To optimize the library each interrupt handler passes the source of the interrupt to the function `UsbLibInterrupt`, which is defined in the Library sources.

3.5 Synchronization

The library has internal data structures and the USB function 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 in the USB interrupt handler so all call back functions are running also in the context of the USB function interrupt handler. Call back functions are never called in another context as the USB interrupt handler.

The library provides in the OSAL layer two functions Enter- and Leave-macros which can be used to synchronize the code. The library makes sure, that after each call of Enter the Leave function is called. The library does not call the macro Leave before a call back function is called. So the call back functions are running under the protection of the Enter and Leave macros. In some call back functions it is useful to call the library again. In this case the Enter macro may be called two or more times before the leave macro is called. The macros can be adapted in one of the following ways. The default implementation for the Enter() macro is DI() and for the Leave Mcro is EI().

3.5.1 Without Operating System

The Enter function stores the state of the global interrupt enable flag or the interrupt level (depends from the used controller) and disable all interrupts. The Leave function restores the value of the flag or the interrupt level.

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 continuos 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 endpoint 0 and transferees the data in FIFO sized chunks. Both methods are discussed in detail in the following sections.

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 Request without data phase

The library calls the function **USBLIB_SETUP_EVENT** and passes the 8 bytes of the setup to the emdedded 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 Request 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. Finally the library calls saaUSBLIB_SETUP_HANDSHAKE_COMPLETE.

3.7.4 Request 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 is 64Bytes.

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 circuit of the MB90330 serie

- 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 of the MB90330. This enables the soft connect with the function **UsbLibEnable**.
- The voltage line from the USB connector must be connected to the pin VBUS.

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 repeatly. 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 42)
USBLIB_SETUP_DATA_TRANSFERRED (page 43)
LibUsbSetupHandshake (page 24)

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.

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 42)
USBLIB_SETUP_DATA_TRANSFERRED (page 43)
LibUsbSetupBuffer (page 22)

UsbLibInitialize

This function must be called to initialize the library.

Definition

```
USBLIB_STATUS
UsbLibInitialize(
     USBLIB_DESCRIPTORS* 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 53)
USBLIB_EP_CFG (page 52)
USBLIB_CALLBACKS (page 54)

UsbLibEnable

This function enables the USB interface.

Definition

```
USBLIB_STATUS
UsbLibEnable();
```

Return Value

The function fails if the library is not initialized or if the device is selfpowered and not connected to a USB host (Usb Vcc is not detected).

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 25)
UsbLibDisable (page 28)
UsbLibRead (page 31)
UsbLibWrite (page 33)

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 cancelled.

See Also

UsbLibInitialize (page 25)
UsbLibEnable (page 27)

UsbLibWakeupPC

This function wakes up the PC.

Definition

```
USBLIB_STATUS
UsbLibWakeupPC(
     unsigned char WakeupEnable
     );
```

Parameter

WakeupEnable

The first time WakeupEnable must set to TRUE and the second time WakeupEnable must set to FALSE. The user must wait for 10ms between the two calls. After the second call the resume signaling to the PC ends.

Return Value

The function fails if the library is not initialized, the device is not in suspended state, or the USBLIB_ENABLE_REMOTE_WAKEUP was not send by the PC.

Comments

This function can be used to wake up a PC from the standby state after the function is set to the USB suspended state from the PC. This USB feature must be enabled by the PC driver. Some additional USB host controller may not support remote wake up. To make sure that a USB host controller supports this feature a standard USB mouse can be used to test the host controller. Normally remote wakeup does not work if the PC has entered the hibernate state. The function must be called twice, see the WakeupEnable parameter.

See Also

UsbLibInitialize (page 25)
UsbLibEnable (page 27)

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 25)
UsbLibEnable (page 27)
USBLIB_START_OF_FRAME (page 47)

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 fare.

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 continous data transfer.

See Also

UsbLibAbort (page 35)
USBLIB_TRANSFER_COMPLETION (page 48)
UsbLibWrite (page 33)
USBLIB_DEVICE_EVENT (page 45)

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 fare.

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 continous data transfer.

See Also

UsbLibAbort (page 35)
USBLIB_TRANSFER_COMPLETION (page 48)
UsbLibRead (page 31)
USBLIB_DEVICE_EVENT (page 45)

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. Wait then about 2ms and call UsbLibAbort again.

USBLIB_STATUS_ERROR: is returned 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 31)
UsbLibWrite (page 33)
USBLIB_TRANSFER_COMPLETION (page 48)

UsbLibSetStall

This function sets the state of an 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.

See Also

UsbLibClearStall (page 37)

UsbLibClearStall

This function clears the STALL state of an endpoint.

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 the endpoint state is cleared the endpoint performs normal data transfers. This function must be called if an endpoint was set to STALL state by calling the function **UsbLibSetStall**. It is not required to call this function during initialization and startup. If an Clear Feature Endpoint Stall is received the library clears the stall condition automatically.

See Also

UsbLibSetStall (page 36)

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 swtich 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.

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 grater 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 22)
LibUsbSetupHandshake (page 24)

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 42)
LibUsbSetupBuffer (page 22)
LibUsbSetupHandshake (page 24)

USBLIB_SETUP_HANDSHAKE_COMPLETE

This function is called after the handshake phase has been completed.

Definition

```
void
USBLIB_SETUP_HANDSHAKE_COMPLETE();
```

Comments

This call is for information only. The embedded application can recognize that the setup request was completed successful. The embedded application must not call LibUsbSetupHandshake or LibUsbSetupBuffer from this callback function.

See Also

LibUsbSetupBuffer (page 22)
LibUsbSetupHandshake (page 24)

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 embeded 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.

USBLIB_ENABLE_REMOTE_WAKEUP the PC has enables the device to perform a remote wakeup during the next suspend phase.

USBLIB_DISABLE_REMOTE_WAKEUP the PC has disabled the device to perform a remote wakeup during the next suspend phase.

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 54)

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 54)

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 54)

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 35)
UsbLibRead (page 31)
UsbLibWrite (page 33)
USBLIB_BUFFER_DESCRIPTOR (page 56)

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_DESCRIPTORS

The USBLIB_DESCRIPTORS structure contains information on the USB descriptors.

Definition

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

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 25)
USBLIB_CONFIGURATION (page 53)

USBLIB_EP_CFG (page 52)
USBLIB_CALLBACKS (page 54)

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 25)
USBLIB_CONFIGURATION (page 53)
USBLIB_CALLBACKS (page 54)

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 25)
USBLIB_EP_CFG (page 52)
USBLIB_CALLBACKS (page 54)

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_SETUP_HANDSHAKE_COMPLETE* SetupHandshakeComplete;
    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.

SetupHandshakeComplete
This field contains the function pointer to a function which receives events if a setup transfer is completed.

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 25)
USBLIB_DEVICE_EVENT (page 45)
USBLIB_ENDPOINT_EVENT (page 46)
USBLIB_SETUP_EVENT (page 42)
USBLIB_SETUP_DATA_TRANSFERRED (page 43)
USBLIB_SETUP_HANDSHAKE_COMPLETE (page 44)
USBLIB_START_OF_FRAME (page 47)

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 occured. The PC should send a Clear Feature Endpoint Stall request to clear the error condition. Some error conditions cannot 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 happen during a UsbLibRead(). The buffer size was not a multiple of the FifoSize or less then 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 31)
UsbLibWrite (page 33)
UsbLibAbort (page 35)

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 occured.

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 (0x0005)

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

USBLIB_STATUS_INVALID_PARAM (0x0006)

A parameter passed to the function was invalid.

USBLIB_STATUS_PENDING (0x0007)

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

5 Demo Application

5.1 USB Interface

The demo application uses the USB Function Library to build a USB interface with the following features:

- One configuration
- One interfase
- Vendor defined class
- Loop data on two bulk pipes
- Loop data on two interrupt pipes
- Read data from an isochronous pipe
- Two vendor requests: Write data and read data to the memory

The hardware is based on a USB1.1. function controller of the MB90F337 with the GLYN evaluation board EVBMB90F335. Other boards are also possible.

To translate the demo application refer to section 6. The buffer size for bulk endpoints is 512 bytes and for interrupt endpoints 256 bytes. The isochronous IN transfer buffer has a size of 512 byte. The isochronous buffer contains a additional 2 byte sequence number at the beginning of the buffer for internal tests.

5.2 Initial Steps

The DEMO application performs the following steps after reset:

- First initialize all interrupt request levels and enable the interrupt.
  InitIrqLevels();__set_il(7);__EI();
- Calling the functions DBG_init() to initialize the debug output device if compile with the DEBUG version. (see dbgprint.*, a powerful trace module and osal.h).
- The function DbgInit() and DbgSetMask() initialize the trace utility and controls the amount of traces.
- InitBuffer() initializes the user buffer for IN transfers with pattern.
- The function UsbLibInitialize() initialize the USB device and checks the USB configuration.
- If the +4..5V USB power is detected then UsbLibEnable() activate the USB function (3,3V to D+ with a 1,5k series resisitance).
- After connecting the device to the USB bus, the internal library function
  `UsbLibInterrupt()` is called from the various USB interrupt sources.
  The `UsbLibInterrupt()` parameters detects which USB event is occured.

- The main() run then in a infinite loop. First the main() poll gConfigure. The variable gConfigure is set from the user event call back function if the event USBLIB_CONFIGURE is detected. From this point the transfer starts.
- The functions UsbLibRead() and UsbLibWrite() are used to transfer data to or from a endpoint. The transfer starts with the setting of the variable gConfigure. The variable gConfigure is set with the user event USBLIB_CONFIGURE. Do always examine the return value of the library reading and writing functions to prevent a calling chain. The return value USBLIB_STATUS_SUCCESS means that the completion routine is not called and the result of the operation is stored in the user buffer. See also **UsbLibRead** and **UsbLibRead**.
- The functions ProcessWriteBuffer() and ProcessReadBuffer() must process the user buffer. Them checks the buffer status in the buffer descriptor and dump the data buffer.
- Because of that a loop transfer application must first read data from the IN endpoint and then put them to the OUT endpoint. To do this the variable gBulkReadEnable is set to false after call of UsbLibRead(). If the read transfer is successfull then ProcessBulkReadLoop() is called. ProcessBulkReadLoop() writes the received buffer with UsbLibWrite() to the IN fifo from where the data are sent to the host. If all data are sent ProcessBulkWriteLoop() is called. To prevent a copy from the user read buffer to the user write buffer the buffer descriptors for the IN and OUT transfers in the loop transfer application uses the same buffer (gBulkLoopBuffer) in the buffer descriptor. After sent all data to the Host the application checks the OUT fifo with UsbLibRead() again. Independent read and/or write applications are also possible.

5.3 Configuration

The Demo application is configured at compile time with several defines in the file config.h:

- EP1_IN_FLAG, EP1_OUT_FLAGS:
  define how the library transfers data between the user buffer and the endpoint FIFOs. See also **USBLIB_EP_CFG**.
- USB_ID_VENDOR, USB_ID_PRODUCT:
  Product and Vendor ID

Other device characteristics are not available in the DEMO 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* 32kb/s = 64kb/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 codesize 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 and Translation of the Library

The header file func_conf.h in the directory \src\inc contains some defines which configure the behavior of the library at compile time.

6.1 Hardware depended configurations

- DMA_SUPPORT: Enables the DMA support in the library. Saves memory if not defined.
- VENDOR:Enable the vendor request support. To save memory disable usbvend.c from the project.
- INTERRUPT_LEVEL: The interrupt level is the level that is used for usb 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 deafult value is 64 bytes.

6.2 Development environment

- Workbench: F2MC-16 Family SOFTUNE Workbench V30L31
- Compiler: fcc907s
- Emulator: MB2147-01 with the evaluation CPU MB90V330A and the USB evaluation board MB2031-01.
- Target board: GLYN EVBM90F335

The workspace file usbfunc.wsp contains all used project files. The procject files consists of the files func_demo.prj and func_lib.prj. Func_demo.prj and func_lib.prj contains different targets. To change a target from a project file first activate the project and then change the target. If the usb library file are changed then translate the library before translate the demo.

The library project path is \src\mb90\func and the demo programn project path
is \src\mb90\demo

The project configuration files for the library and the for the evaluation CPU are stored in the following locations:
\src\mb90\func\lib\V330chk

The project settings from the library and the demo differs in the C-Compiler defines DBG and USB_EVA_BOARD, the entries for the file names and the different target MCU's. The compiler model is medium.

The sources are stored in the following locations:
library sources: \src\mb90\func\lib\src

demo sources: \src\mb90\func\demo\src

All binaries are located in the same directory with different names. The directory is \src\bin with the subdirectories \chk and \fre.

binary	description
\src\bin\chk\func_demoV330.abs	debug version of the demo for the evaluation chip MB90V330A
\src\bin\fre\func_demoV330.abs	release version of the demo for the evaluation chip MB90V330A
\src\bin\chk\func_demoF337.abs	debug version of the demo for the USB chip MB90F337
\src\bin\fre\func_demoF337.abs	release version of the demo for the USB chip MB90F337
\src\bin\chk\func_libF337.lib	debug version of the usb library for the USB chip MB90F337 (medium model)
\src\bin\fre\func_libF337.lib	release version of the usb library for the USB chip MB90F337 (medium model)
\src\bin\chk\func_libF337.lib	debug version of the usb library for the USB chip MB90F337 (medium model)
\src\bin\fre\func_libF337.lib	release version of the usb library for the USB chip MB90F337 (medium model)
\src\bin\chk\func_libV330.lib	debug version of the usb library for the evaluation chip MB90V330A (medium model)
\src\bin\fre\func_libV330.lib	release version of the usb library for the evaluation chip MB90V330A (medium model)

6.3 Installation

6.3.1 Loading the device binaries

This explains the loading of the binaries on the GLYN evaluation board EVBMB90F335 with the MCU MB90F337.

Jumper settings:

- JP1: not used, open (TX from UART1 to DB9 connector X2)
- JP2: not used, open (RX from UART1 to DB9 connector X2)
- JP3:during downloading:1-2 (The red programming LED is on)
- JP3:after download:2-3 (The red programming LED is off)
- JP4: 2-3 (1,5k pull up resistor from D+)
- :JP5: open (Mode Pin MD1 = High)
- :JP6: open (+5V for USB Host connector, not used)
- :JP7-JP10: not available
- JP11: 1-2,close
- JP12: 1-2,close
- JP13: 1-2,close
- JP14-JP16: not available
- JP17: 1-2, close

1. Before starting the download set JP3 and check the other jumpers (s.a.jumper settings).
2. Connect the demo board with a USB cable to the PC (USB is needed for the power supply, the green power LED must go on).
3. Start the Flash utility and open the file func_demoF337.mhx.
4. Press the reset switch on the demoboard.
5. Select the Full Operation from the Flash utility and wait for end of programming.
6. Close the flash utility.
7. Disconnect the Demoboard from USB.
8. Set JP to 2-3.
9. Connect the Demoboard with the USB cable to the PC.

At the third point start the FUJITSU Flash Programmer and connect a free serial COM interface from the PC with the RS232 connector X1 from the board. The Flash Programmer tool MCU Programmer 16LX (V01L13 or a higher version) can be downloaded from the FUJITSU home page http://www.fme.gsdc.de.

Note that the target system must have a RS-232C signal driver for communication with the microcontroller UART. Connect the demo board with a external power of 9-12V or connect the board with a USB cable with a PC.

Check the settings in the programmer tool (target: MB90F337, Crystal Frquency 6Mhz)
and the SetEnvironment button.
Open the motorola hex file in the directory \src\bin\fre\func_demoF337.mhx. After reset the MCU press the FullOperation button in the programmer tool. After programming set the mode pins to normal mode JP3=1-2 (MD0=1, MD1=1, MD2 =0).

6.3.2 USB Device driver Installation

Install usbio_demo. After Installation connect the Demoboard with the PC and install the new USB device after connection. (see also USBIO Installation Wizard). In the Device Manager the device "USBIO Device: VID:0x815 PID:0x001" is displayed. If the device is not present check the USBIO device installation and the device.

6.3.3 Start the Test application

Start the demo90330.exe. Then click start to run the bulk loop transfer. If the device is not found a error message is displayed. All endpoints on the target board are configured without DMA so the data rate is about 0,5Mb/s. Only one transfer type at the same time is running.

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