

Fujitsu USB Minihost API-FUMA

MB90330 and MB90335 series

Reference Manual

Version 1.2

June 16, 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>

Copyright (c) 2003-2005 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
2.4 Performance	12
3 Architecture	13
3.1 USB HOST interrupt events and the library	14
3.2 Integration of the Library into an Application	16
3.3 Compile time configuration	18
3.4 Library interrupt handler	18
3.5 Synchronization	18
3.5.1 Without operating system	19
3.5.2 With operating system	19
3.6 Data Transfer and Performance	19
3.7 Setup Requests	19
3.7.1 Error Handling	20
3.7.2 Requests without data phase	20
3.7.3 Request with data OUT phase	20
3.7.4 Request with data IN phase	20
4 Programming Interface	21
4.1 API Functions	21
UMH_Init	22
UMH_EnumerateDevice	23
UMH_SetConfiguration	24
UMH_AddEndpoint	25
UMH_RemoveEndpoint	27
UMH_SetupRequestCpl	28
UMH_SetupRequest	29
UMH_AbortSetupRequest	30

UMH_Transfer	31
UMH_AbortTransfer	32
UMH_ResetDataToggleBit	33
UMH_SetPowerState	34
UMH_GetDeviceState	35
4.2 API callback functions	36
UMH_StallExecutionUserCallback	37
UMH_DEVICE_STATE_CALLBACK	38
UMH_COMPLETION	39
UMH_STATUS_COMPLETION	40
4.3 Enumeration Types	41
UMH_POWER_STATE	41
UMH_DEVICE_STATE	42
4.4 Error Codes	43
UMH_STATUS_SUCCESS (0x0000L)	43
UMH_STATUS_ERROR (0x0001L)	43
UMH_STATUS_BUSY (0x0002L)	43
UMH_STATUS_INVALID_PARAM (0x0003L)	43
UMH_STATUS_OVERRUN (0x0004L)	43
UMH_STATUS_TIMEOUT (0x0005L)	43
UMH_STATUS_COMPLETE (0x0006L)	43
UMH_STATUS_CRC (0x0007L)	43
UMH_STATUS_STALL (0x0008L)	43
UMH_STATUS_DELAYED (0x0009L)	43
UMH_STATUS_CANCELED (0x000AL)	44
UMH_STATUS_DATA_TOGGLE_MISMATCH (0x000BL)	44
UMH_STATUS_BITSTUFF (0x000CL)	44
5 Demo Application	45
5.1 HOST interface	45
5.2 Program flow	45
5.3 Traces	46
5.4 Running the demo application	46
5.4.1 Hardware requirements	46
5.4.2 Software requirements	46

5.4.3	Jumper settings	47
5.4.4	Programming	47
5.4.5	Starting the demo program	48
6	Configuration and translation of the library	49
6.1	Hardware depended configurations	49
6.2	Development environment	49
7	Related Documents	51
	Index	53

1 Introduction

FUMA is a generic Universal Serial Bus (USB) minihost library for the 16-bit FUJITSU MCU MB90330 and MB90335 series. It covers the entire USB minihost of the micro controller and provides a convenient way to use the API.

This document describes the architecture, the features and the programming interface of the FUMA 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 minihost 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. By using the FUMA firmware library it is possible to get the USB host interface up and running without spending the time and the effort of developing a new USB host firmware.

2.1 Compiler

The FUMA library supports the following C-compiler:

- F2MC-16 Family SOFTUNE Workbench V30L31 with FS907S compiler

2.2 Features

The FUMA library provides the following features:

- initialisation of the minihost
- support of USB 1.1 full speed and low speed devices
- registration of callback functions
- device enumeration including USB bus reset, SET ADDRESS request and setting of maximum packet size for the control endpoint
- suspend
- resume
- setup requests
- bulk transfer
- interrupt transfer

2.3 Restrictions

Some restrictions apply to the FUMA firmware library:

- The minihost library cannot be used if the USB function is used on the MB90330 series at the same time. The sharing of the FIFO's of the MCU hardware causes this limitation.
- isochronous transfer is not supported
- supports one USB device at the same time
- does not support Hub devices

2.4 Performance

The maximum bandwidth for bulk transfer is 270 kbyte/s. A higher bandwidth is not possible because no DMA transfer is available for this I/O-resource. To obtain the maximum bandwidth you should use the firmware release version. The buffer size should be at least 512 bytes.

3 Architecture

The Embedded Application is the software running on the MB90330 and the MB90335 series. It uses the FUMA library to communicate via the USB host by an USB function. The FUMA Library is divided into three parts: the USB host library, the Hardware Abstraction Layer (HAL), and the Operating System Abstraction Layer (OSAL). The USB host library handles the USB Bus requests and the data transfer on an abstract level. The HAL performs the access to the registers of the USB host. The OSAL layer 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 an operating system. The hardware contains the USB related registers and the physical connection to the USB host connector. This document describes the FUMA library. The Hardware Abstraction Layer is not described.

Figure 1 shows the FUMA library architecture.

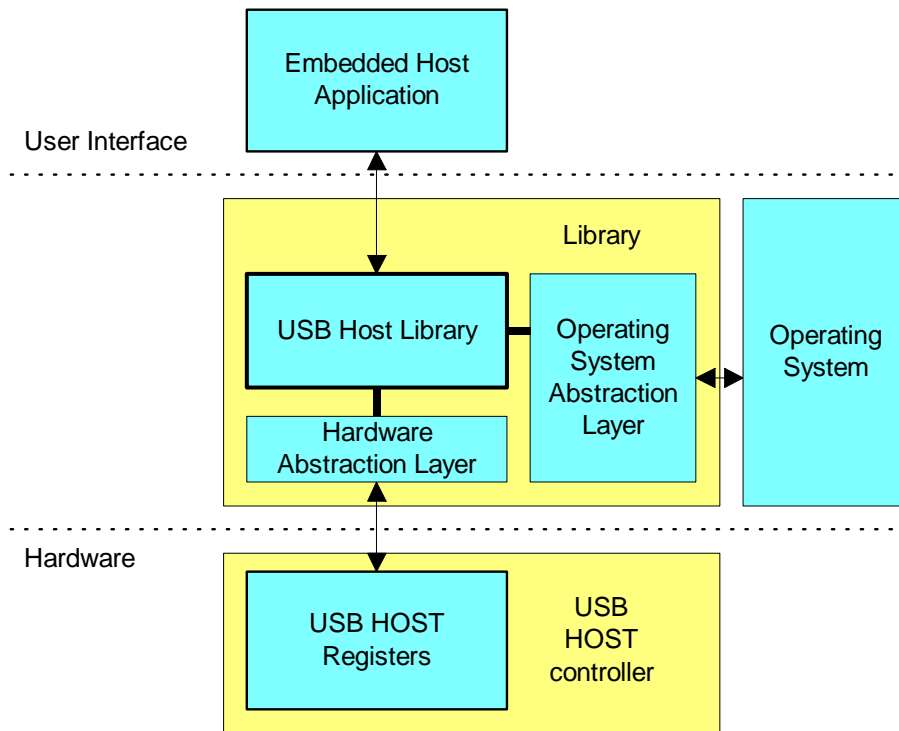


Figure 1: FUMA Software Architecture

The library does not require any external libraries or function calls out of `<OSALdbgp.c>`. The OSAL layer defines a debug print function. A system independent implementation of the print function is implemented in the file `<OSALdbgp.c>`. This file requires `<stdarg.h>`. But the debug print function can be mapped to any other debug print capability of an existing project. The header file `<usbminihost.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.

3.1 USB HOST interrupt events and the library

- **Device connect event**
A connected USB device is detected by the 3,3V on the D+ (full speed device) or D- line (low speed device). The internal function `UmhDevConnect()` sets the USB operation clock and starts the SOF token or keep alive signals. The user is then informed by the event `UMH_EVENT_CONNECT`. The device state is set to **UMH_DEVICE_STATE_POWERED**.
- **Device disconnect event**
`UmhDevRemove()` is called if the HOST detects a device removal and the event `UMH_EVENT_REMOVED` is signalled to the application.
- **Device resume signal detection event**
To get a resume event the device has to be set into suspend state with **UMH_SetPowerState** and the parameter `UMH_POWER_SUSPEND`. **UMH_SetPowerState** waits for the end of the current token and cancels all outstanding transfer requests. Then it stops the SOF token and puts the USB bus in suspended state. If a remote wakeup or the end of the resume signal is detected the internal function `UmhDevWakeUp()` is called and starts the SOF token. Now the application can submit new transfer requests.
- **Start of frame event**
Every millisecond a SOF interrupt is generated. The SOF interrupt calls the internal function `UmhDevSOFevent()`, see also figure 3, where all interval counters are decremented. If an interval counter is zero the scheduler starts the related interrupt endpoint request if active. Active means that a request was passed to that interrupt endpoint. If an interrupt token is completed the interval counter is restarted and decremented by the SOF interrupt until zero.
- **Token completion event**
A token completion event occurs when the USB device has sent the handshake token or an error is detected. In that case the token scheduler (`UmhTokenScheduler()` in Figure 3) is called. Received data are read from the FIFO and written to the transfer buffer. If the transfer is not complete the request is released and the next one is submitted to the USB bus. The completion routine **UMH_COMPLETION** is called if all data has been transferred.
- **Token send**
First the requested token (SETUP, IN or OUT token) is sent. The host's fifo's are used to copy the data for the transfer. If a transaction finishes the scheduler copies the data to the application buffer if necessary. If not all data has been transferred the scheduler does not repeat the same request but searches the next available request in the list.

USB events and related library functions

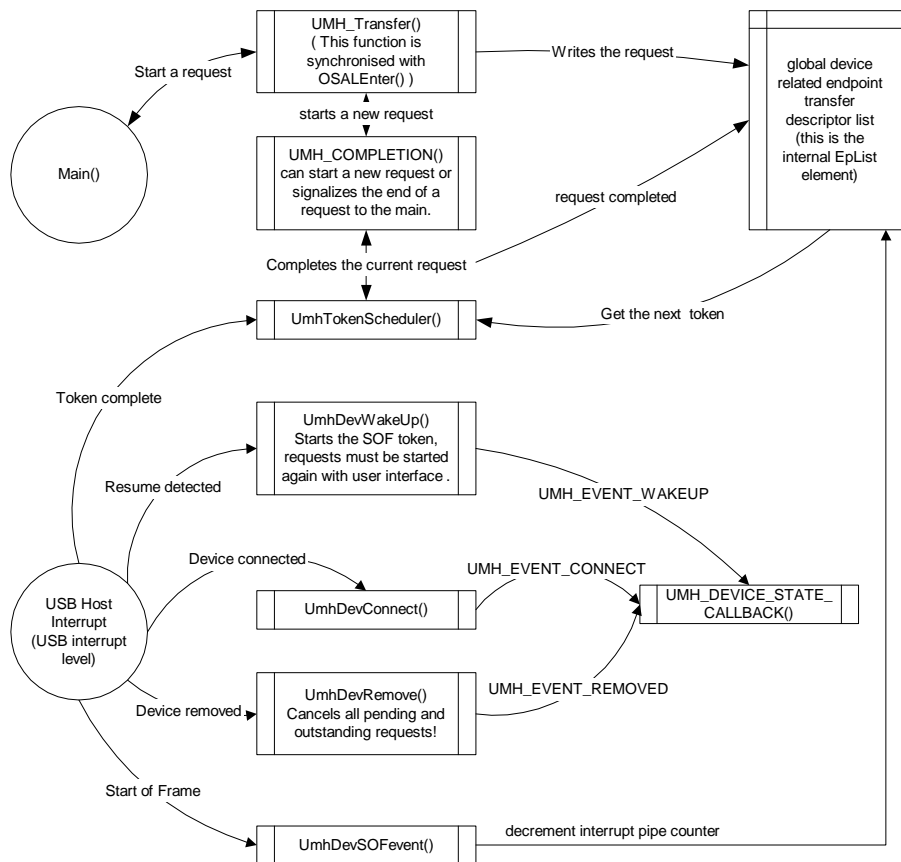


Figure 2: USB HOST events and the library

3.2 Integration of the Library into an Application

Figure 3 shows the use of the FUMA library. In the following the use of the library functions is explained. User defined helper function listed in figure 3 are omitted. The application must implement **UMH_StallExecutionUserCallback** that is called inside the library to wait for a specified time. This function must not return before the requested time has passed. It is possible to do additional work in this function, but the library must not call again.

- At first the application has to call the function **UMH_Init**. The application passes the user event callback function **UMH_DEVICE_STATE_CALLBACK** to this routine. The **UMH_DEVICE_STATE_CALLBACK** function informs the application about the connecting and removing of an USB device.
- The function **UMH_AddEndpoint** adds a data endpoint to the library if the application uses data endpoints.
- If the device is connected the event **UMH_EVENT_CONNECT** is set and the application begins with the device enumeration by calling **UMH_EnumerateDevice**. **UMH_EnumerateDevice** sends a USB bus reset, starts the SOF token and set the device address. The USB device descriptor is requested and the maximum packet size of endpoint zero is extracted.
- If **UMH_EnumerateDevice** returns with **UMH_STATUS_SUCCESS** the application can communicate with the control endpoint (endpoint zero). To transfer data with data endpoints the device must be in a configured state. **UMH_GetDeviceState** returns the value **UMH_DEVICE_CONFIGURED** if the device is configured. The device is configured with **UMH_SetConfiguration**. If **UMH_SetConfiguration** returns with **UMH_STATUS_SUCCESS** the user can start the communication with data endpoints.
- Control endpoint requests including vendor and class requests can be executed with **UMH_SetupRequest**.
- If the device is configured a data endpoint transfer can be started with **UMH_Transfer**. The application can schedule one transfer per data endpoint. The next data transfer can be started, if the completion routine is called with status success. In the case of an error, an error recovery (ResetPipe) is required.
Do always wait for the end of the previous transfer before starting a new one. Another solution is to start the second data transfer in the completion callback routine.
- The application has to stop the transfer if a device disconnect event is detected. After a device connection event the device has to begin with the device enumeration.

Figure 3 shows the program flow from the point of view of the user application.

Typical program flow of an embedded host application with the USB host library

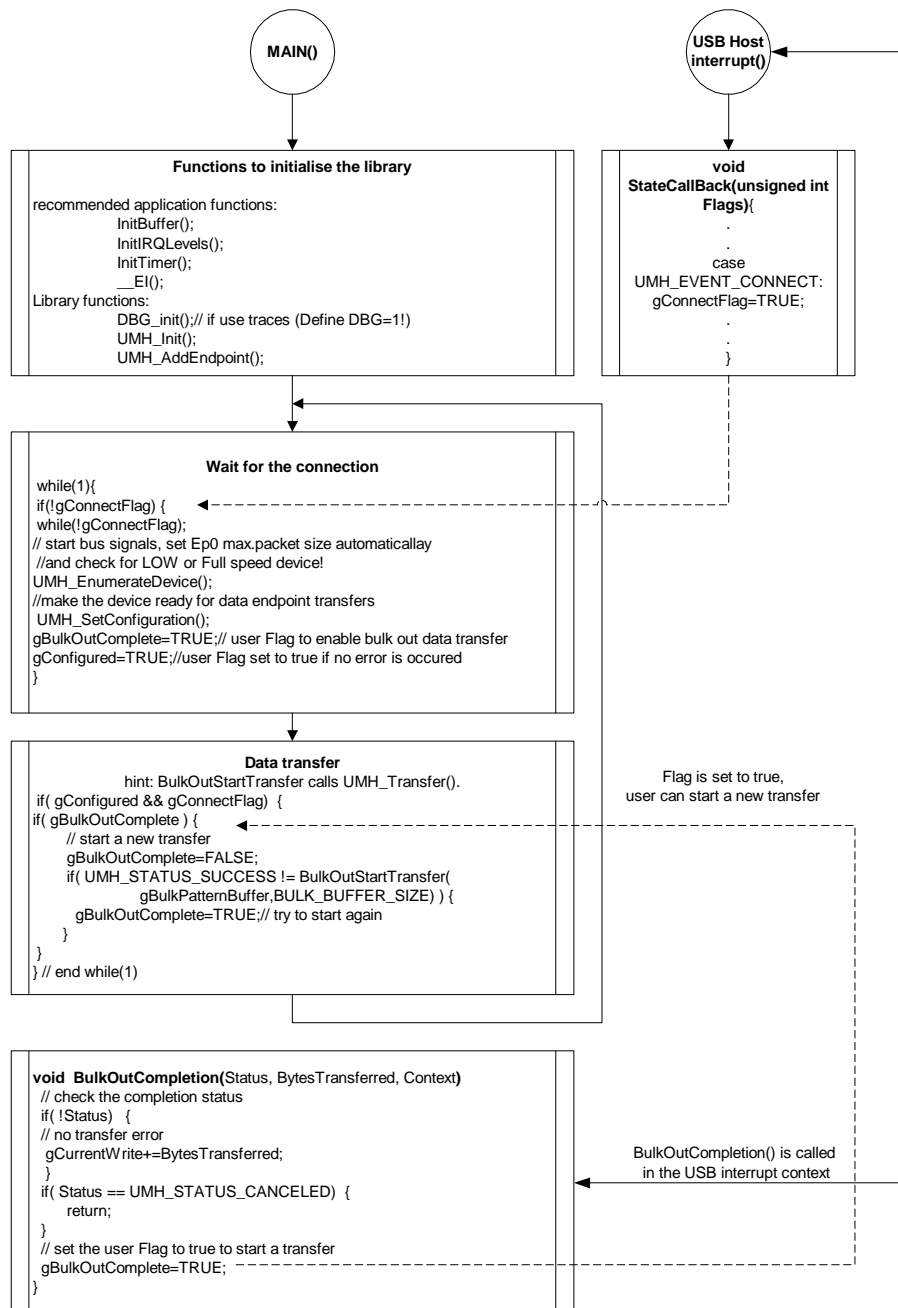


Figure 3: Program Flow in the main() context

3.3 Compile time configuration

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

3.4 Library interrupt handler

All USB host specific interrupt functions are implemented in the FUMA library. These functions are not declared with the `__interrupt` keyword. They has to be called from a user defined USB host interrupt handler function, see also `<usb_int.c>` and `<vectors.c>` in the demo application. The IRQ level of the USB host interrupt is defined with `USB_INT_LEVEL` in `<umh_haldef.h>`. This level is set by the internal function `UmhHalUsbIntEnable()` during library initialisation. To enable the USB interrupt the interrupt level has to be set to `USB_INT_LEVEL+1` or higher. The internal function `UmhLibInterrupt()` is called from the interrupt handler function and detects all active interrupt requests. If USB interrupts are broken by higher interrupts the transfer duration is larger.

3.5 Synchronization

The library has internal data structures and the USB host requires special sequences, which must not be interrupted. From this point of view the library is not re-entrant. To synchronize the code of the USB host interrupt handler with the host library interface several methods are used. An 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.

The USB host interrupt calls the internal function `UmhLibInterrupt()`. `UmhLibInterrupt()` then dispatches the event to the correct internal function so the most library functions runs in the context of the USB Host interrupt.

With the the file `<osal.h>` the library provides two macros `OSALEnter` and `OSALLeave` which are used to synchronize the library code. Synchronize means that a library function during the run time can not be interrupted of the USB host interrupt. It is not allowed to call library functions that wait for an USB event in an interrupt with a higher or equal priority (that would cause to a deadlock). In the programming interface the allowed caller process context is declared.

The callback functions are running under the protection of the `OSALEnter` and `OSALLeave` macros. So callback functions are always synchronised. In some callback functions it is useful to call a library function again. A synchronised library function can call another synchronised library function. The code is synchronised until after the execution of the last call (if make nested calls) of the `OSALLeave` macro.

The `OSALEnter` macro has to be inserted directly after the end of the local variables because the macro self defines a local helper variable.

The function reference describes the various process context during the run time. The macros can be adapted in one of the following ways.

3.5.1 Without operating system

The `OSALEnter` macro stores the state of the global interrupt flag or the interrupt level (depends on the used controller) and disables the interrupt. The `OSALLeave` macro restores the value of the interrupt enable flag or the interrupt level. It is possible to disable all interrupts and to disable only interrupts with a same or a lower priority. This depends from the compiler define `SYNC_EQUAL_LOWER_INTS`.

Depending from the value of the compiler define `SYNC_EQUAL_LOWER_INTS` not all interrupts are disabled after a call to a library function.

If `SYNC_EQUAL_LOWER_INTS` is zero then all interrupts are disabled if a library function is called from the application. This method has the advantage that library functions can also be called from interrupts with a higher priority as the USB host interrupt.

If the compilers define `SYNC_EQUAL_LOWER_INTS` has a value unequal zero then only interrupts with an equal or a lower priority are disabled if a library function is called. This method has the advantage that time critical interrupts are not disabled during execution of a library function and has the disadvantage that a library function can not be called from a interrupt routine with a higher priority as the USB host interrupt.

A call of the library functions from the `main()` is with both synchronize methods possible.

3.5.2 With operating system

The user must change the macros `OSALEnter` and `OSALLeave` in `<osal.h>` if it is not possible to use this with the Operating system. The `Enter` function requests a synchronization event from the operating system. The synchronization event is required to 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 has to be suspended. The `OSALLeave` macro releases the synchronization event.

3.6 Data Transfer and Performance

For data endpoints the FUMA library supports a mode to achieve a high data throughput. The library can handle one transfer buffer per endpoint. The application provides the buffer for the data transfer. The buffer size may be larger than the FIFO size of the endpoint. On an IN transfer the library completes the buffer if it is completely filled or if a short packet is received. To avoid buffer overruns the buffer size should be a multiple of the FIFO size. The time increase with the transfer buffer size until the request is completed.

3.7 Setup Requests

The FUMA library supports all available setup requests including class or vendor specific requests by the functions `UMH_SetupRequestCpl` and `UMH_SetupRequest`. These functions realize the setup request that consists of 3 phases (Please refer to the USB specification for more details):

- setup phase,
- data phase (optional),

- handshake phase.

The setup phase is always running. All bytes of this setup packet can be defined by the application. 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 is limited to 64 KB.

The handshake phase allows 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 an error by stalling the endpoint.

3.7.1 Error Handling

The application can abort each setup request with **UMH_AbortSetupRequest**. That means the sequence setup, data, and handshake can be interrupted. A not completed token in one of the setup stages cannot be interrupted. The user has to wait for the end of the current token.

3.7.2 Requests without data phase

The library calls the function **UMH_SetupRequest** and passes the 8 bytes of the setup to the embedded application. The direction bit in the `bmRequestType` field and the `wLength` field has to be zero. The parameter buffer has to be NULL to prevent a data phase. If the handshake status is acknowledged **UMH_SetupRequest** returns. If **UMH_SetupRequestCpl** is used the completion callback routine is called.

3.7.3 Request with data OUT phase

The library has to provide a transfer buffer for the data phase. **UMH_SetupRequest** is called with a valid buffer pointer and the length field in the setup request must not exceed the size of the parameter buffer. After all bytes have been sent to the device and the handshake status is acknowledged **UMH_SetupRequest** returns. If **UMH_SetupRequestCpl** is used the completion callback routine is called. If the device responds with NAK's for a determined time during data or handshake phase and no bus error is occurred the functions return `UMH_STATUS_TIMEOUT`.

3.7.4 Request with data IN phase

The request direction has to be set to one and the setup length field is set to the data length and has to be less or equal to the maximum buffer size. The returned length can be less or equal to the length field in the setup buffer.

4 Programming Interface

4.1 API Functions

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

UMH_Init

Initialise the USB Host circuit.

Definition

```
void
UMH_Init(
    UMH_DEVICE_STATE_CALLBACK DeviceState
) ;
```

Parameter

DeviceState

Device State notification handler. This callback function is always called from the USB interrupt context.

Comments

UMH_Init registers the device state notification handler. This function has to be called one time during initialisation. No device related actions are done. Do not call this function in the USB interrupt context.

See Also

[**UMH_DEVICE_STATE_CALLBACK**](#) (page 38)

UMH_EnumerateDevice

Enumerate the USB device.

Definition

```
UMH_STATUS
UMH_EnumerateDevice(
    unsigned int Address,
    UMH_STATUS_COMPLETION* CompletionFunction
);
```

Parameters

Address

USB Device address (must not be zero!)

CompletionFunction

Optional callback completion function. The callback function is always called in the process context of the caller before UMH_EnumerateDevice finishes.

Return Value

The function returns one of the following status codes:

USBLIB_STATUS_SUCCESS if successful,

UMH_STATUS_TIMEOUT if a timeout occurred during a HOST operation,

UMH_STATUS_INVALID_PARAM if a parameter was invalid,

UMH_STATUS_ERROR if the device is in disconnect or suspend state,

UMH_STATUS_STALL if the endpoint zero stalled a request.

Comments

After the device event UMH_EVENT_CONNECT is received the application has to call UMH_EnumerateDevice. The function first starts the SOF token makes a USB bus reset and set the device address. Then the control endpoint maximum packet size is queried. If the returned status is successful then all control endpoint related functions could be used. Do not call this function in the USB host interrupt context or in an interrupt context that has a higher or the same priority as the USB interrupts.

See Also

[UMH_STATUS_COMPLETION](#) (page 40)

UMH_SetConfiguration

Send a configuration request and reset the FIFOs and data toggle bits in the device.

Definition

```
UMH_STATUS
UMH_SetConfiguration(
    unsigned char ConfigurationValue,
    UMH_STATUS_COMPLETION* CompletionFunction
);
```

Parameters

ConfigurationValue

If the value is zero then the device is unconfigured and the device state is set to UMH_DEVICE_ADDRESSED.

CompletionFunction

Optional callback completion function. The function is always called in the process context of the caller before UMH_SetConfiguration finishes.

Return Value

The function returns one of the following status codes:

UMH_STATUS_SUCCESS if successful,

UMH_STATUS_INVALID_PARAM if the configuration value is unequal zero and the device is configured or the value is zero and the device is unconfigured,

UMH_STATUS_ERROR if a communication error occurs.

Comments

Do not call this function in the USB host interrupt context or in a interrupt context that has a higher or the same priority as the USB interrupt. This function waits always for a end of the operation.

See Also

[**UMH_EnumerateDevice**](#) (page 23)

[**UMH_STATUS_COMPLETION**](#) (page 40)

UMH_AddEndpoint

Adds an endpoint except the control endpoint to the device.

Definition

```
UMH_STATUS
UMH_AddEndpoint(
    UMH_HANDLE* Handle,
    unsigned char EndpointAddress,
    unsigned int FifoSize,
    unsigned int Interval,
    UMH_COMPLETION* CompletionFunction
);
```

Parameters

Handle

Caller provided parameter that returns handle to the endpoint.

EndpointAddress

Valid device endpoint address with direction bit. This value depends from the endpoint descriptor.

FifoSize

The fifo size of the used endpoint address. This value is part of the endpoint descriptor.

Interval

Polling interval in ms. Set the interval to zero for bulk endpoints and greater than 0 for interrupt endpoints. For interrupt EP's this value is part of the EP descriptor.

CompletionFunction

Pointer to the transfer completion function that is called if a transfer on that endpoint has been completed. The completion function is called in the USB interrupt context. If the transfer is aborted with [UMH_AbortTransfer](#) the context of the CompletionFunction is the same context where [UMH_AbortTransfer](#) is called.

Return Value

The function returns one of the following status codes:

UMH_STATUS_SUCCESS if successful,

UMH_STATUS_INVALID_PARAM if too many endpoints has been added.

Comments

A call to this function is allowed after [UMH_Init\(\)](#) was called. It is also possible to remove an endpoint with the function [UMH_RemoveEndpoint](#). Do not call this function

in the USB host interrupt context or in an interrupt context that has a higher or the same priority as the USB interrupt.

See Also

UMH_RemoveEndpoint (page 27)

UMH_AbortTransfer (page 32)

UMH_STATUS_COMPLETION (page 40)

UMH_RemoveEndpoint

Remove a data endpoint from the internal list.

Definition

```
UMH_STATUS  
UMH_RemoveEndpoint(  
    UMH_HANDLE* Handle  
);
```

Parameter

Handle

Valid endpoint handle. After the function returned the handle is invalid.

Return Value

The function returns one of the following status codes:

UMH_STATUS_SUCCESS if successful,

UMH_STATUS_ERROR for an invalid handle.

See Also

[UMH_AddEndpoint](#) (page 25)

[UMH_Transfer](#) (page 31)

UMH_SetupRequestCpl

Submit a setup request together with the dedicated callback completion function.

Definition

```
UMH_STATUS
UMH_SetupRequestCpl(
    unsigned char* Setup[8],
    unsigned char* Buffer,
    UMH_COMPLETION* CompletionFunction
);
```

Parameters

Setup[8]

Setup packet. The buffer length is given with byte 7 and 8. The direction of the data phase is given with bit 0x80 of the first byte. The caller is responsible to send valid setup packets. Do not use this function for SetAddress and SetConfiguration request.

Buffer

Data buffer or NULL if no data should be transferred.

CompletionFunction

User defined data transfer completion function.

Return Value

The function returns one of the following status codes:

UMH_STATUS_SUCCESS if successful,

UMH_STATUS_ERROR if the device is not connected or suspended,

UMH_STATUS_BUSY if another request is still pending, the request is not submitted.

Comments

The number of transferred byte in the data phase is returned in the parameter BytesTransferred of the completion function. Do not call this function in the USB host interrupt context or in an interrupt context that has a higher or the same priority as the USB interrupt.

See Also

[UMH_AbortSetupRequest](#) (page 30)

[UMH_EnumerateDevice](#) (page 23)

UMH_SetupRequest

Submit a setup request without the use of a callback function.

Definition

```
UMH_STATUS
UMH_SetupRequest(
    unsigned char* Setup[8],
    unsigned char* Buffer,
    unsigned int* BytesReturned
);
```

Parameters

Setup[8]

Setup packet. The buffer length is given with byte 7 and 8. The direction of the data phase is given with bit 0x80 of the first byte. The caller is responsible to send valid setup packets. Do not use this function for SetAddress and SetConfiguration request.

Buffer

Data buffer or NULL if no data should be transferred.

BytesReturned

Caller provided parameter that returns the number of bytes successfully transferred in the data phase.

Return Value

The function returns the following status codes:

UMH_STATUS_SUCCESS if successful,

UMH_STATUS_BUSY if another request is still pending, the request is not submitted,

UMH_STATUS_ERROR if the device state is invalid,

UMH_STATUS_TIMEOUT if a timeout occurred. The length of the timeout is determined by the define TIMEOUT_EP0_TRANSFER.

Other errors are returned if an communication error occurs.

Comments

After return the request has been completed. Do not call this function in the USB host interrupt context or in an interrupt context that has a higher or the same priority as the USB interrupt.

See Also

[UMH_AbortSetupRequest](#) (page 30)

[UMH_EnumerateDevice](#) (page 23)

UMH_AbortSetupRequest

Abort the current Setup request.

Definition

```
UMH_STATUS  
UMH_AbortSetupRequest ( ) ;
```

Return Value

The function returns the following status codes:

UMH_STATUS_ERROR - if currently no request is pending,

UMH_STATUS_SUCCESS - if successful,

UMH_STATUS_DELAYED - if the token from this request is pending the request cannot be aborted at once, the completion routine is called at a later time.

Comments

This function aborts the last submitted setup request. Only setup requests that have a completion routine can be aborted.

See Also

UMH_SetupRequestCpl (page 28)

UMH_Transfer

Perform a read or write request on a bulk or interrupt pipe.

Definition

```
UMH_STATUS
UMH_Transfer(
    UMH_HANDLE Handle,
    void* Context,
    unsigned char* Buffer,
    unsigned int BufferSize
);
```

Parameters

Handle

endpoint handle.

Context

User specific context pointer. This pointer is passed to the completion routine without changes.

Buffer

Pointer to the transfer buffer provided by the application.

BufferSize

Size of the buffer.

Return Value

The function returns the following status codes:

UMH_STATUS_SUCCESS - if successful,

UMH_STATUS_ERROR - The request could not be submitted because of an invalid device state.

UMH_STATUS_BUSY - another request is still pending, the request is not submitted.

Comments

If all data has been transmitted, a short packet is received or a transmission error occurred the completion function is called. Only one transfer per endpoint is allowed at the same time. This function can be called inside a completion function to start the next transfer.

See Also

[UMH_AddEndpoint](#) (page 25) [UMH_AbortTransfer](#) (page 32)

UMH_AbortTransfer

Aborts the current data transfer.

Definition

```
UMH_STATUS
UMH_AbortTransfer(
    UMH_HANDLE Handle
);
```

Parameter

Handle
Endpoint handle.

Return Value

The function returns the following status codes:
UMH_STATUS_SUCCESS - if successful,
UMH_STATUS_ERROR - if currently not request is pending,
UMH_STATUS_DELAYED - if the completion routine is called at a later time.

Comments

If returns with success the completion function has been called.

See Also

[**UMH_Transfer**](#) (page 31)

UMH_ResetDataToggleBit

Reset the data toggle bit for the specified endpoint.

Definition

```
void
UMH_ResetDataToggleBit(
    UMH_HANDLE Handle
);
```

Parameter

Handle
endpoint handle

Comments

Call this function if no request is pending on this endpoint. The function has to be called if the application sends a clear feature endpoint stall request.

See Also

[UMH_Transfer](#) (page 31)
[UMH_AddEndpoint](#) (page 25)

UMH_STATUS_INVALID_PARAM

UMH_SetPowerState

Set the USB bus power state.

Definition

```
UMH_STATUS
UMH_SetPowerState (
    enum UMH_POWER_STATE PowerState
) ;
```

Parameter

PowerState
Power state.

Return Value

The function returns the following status codes:

UMH_STATUS_SUCCESS - if successful,

UMH_STATUS_INVALID_PARAM - invalid power state or if the device is not connected or the device is not suspended and should be resumed.

Comments

If the device is first suspended and then disconnected from the USB bus at first a resume and then a remove event is detected. These events are also sent to the embedded host application. Do not call this function in the USB host interrupt context or in an interrupt context that has a higher or the same priority as the USB interrupt. This function waits always for the end of the operation with [UMH_StallExecutionUserCallback](#).

See Also

[UMH_GetDeviceState](#) (page 35)

[UMH_POWER_STATE](#) (page 41)

UMH_GetDeviceState

Return the current device state.

Definition

```
void
UMH_GetDeviceState(
    enum UMH_DEVICE_STATE* DeviceState,
    unsigned int* Flags
);
```

Parameters

DeviceState

Caller provided parameter that receives the device state.

Flags

Caller provided parameter that returns any combination (bit-wise or) of the following values:

UMH_STATE_FLAG_SUSPEND

If this flag is set the device has been suspended in the state returned by the parameter **DeviceState**.

See Also

[UMH_SetPowerState](#) (page 34)

[UMH_DEVICE_STATE](#) (page 42)

4.2 API callback functions

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

UMH_StallExecutionUserCallback

This function is called if a UMH function is called and the library has to wait in that function for the end of a operation. The function only runs in the user process context.

Definition

```
void
UMH_StallExecutionUserCallback(
    unsigned long time
);
```

Parameter

time
time the function has to wait, in 0.1 millisecond units

Comments

This function is called by the library API functions that run not in the USB interrupt context. The function has to be implemented by the embedded user application. The function must not return before the requested time is elapsed. It can do additional work but should not call the library again.

See Also

[**UMH_EnumerateDevice**](#) (page 23)

[**UMH_SetConfiguration**](#) (page 24)

[**UMH_SetupRequest**](#) (page 29)

UMH_DEVICE_STATE_CALLBACK

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

Definition

```
void
UMH_DEVICE_STATE_CALLBACK(
    unsigned int Flags
);
```

Parameter

Flags

Any combination of the following events that has been detected:

UMH_EVENT_REMOVED if a device disconnect event has been detected. The library handles USB specific actions. Pending data requests are cancelled.

UMH_EVENT_CONNECT if a device connect event has been detected.

UMH_EVENT_WAKEUP if a wakeup event has been detected. A wakeup event can be triggered by a resume signal, a device remote wakeup signal or the removing of a USB device.

Comments

The function is called in the USB interrupt context. In this function other library functions can be called, except the following:

- **UMH_EnumerateDevice**
- **UMH_SetConfiguration**
- **UMH_SetupRequest**
- **UMH_SetPowerState**

See Also

UMH_Init (page 22)

UMH_COMPLETION

Transfer completion call back function.

Definition

```
void
UMH_COMPLETION(
    UMH_STATUS Status,
    unsigned int BytesTransferred,
    void* Context
);
```

Parameters

Status

Final status of the operation. The status is UMH_STATUS_SUCCESS if successful, an error code otherwise.

BytesTransferred

Number of successfully transferred bytes.

Context

Context pointer that is associated with the transferred buffer. The context pointer has been passed by the embedded application to the transfer function.

Comments

This prototype is used for completion callbacks with the functions **UMH_SetupRequestCpl** and **UMH_Transfer**. It is possible to call other functions of the library from this callback function, except the following:

- **UMH_EnumerateDevice**
- **UMH_SetConfiguration**
- **UMH_SetupRequest**
- **UMH_SetPowerState**

See Also

UMH_AddEndpoint (page 25)

UMH_SetupRequestCpl (page 28)

UMH_STATUS_COMPLETION

Transfer completion call back function.

Definition

```
void
UMH_STATUS_COMPLETION(
    UMH_STATUS Status
);
```

Parameter

Status

Final status of the operation. The status is UMH_STATUS_SUCCESS if successful, an error code otherwise.

Comments

This prototype is used for completion callbacks with the functions UMH_ResetDevice, UMH_EnumerateDevice and UMH_SetConfiguration. It is possible to call other functions of the library from this callback function, except the following:

Do not call the following library functions:

- **UMH_EnumerateDevice**
- **UMH_SetConfiguration**
- **UMH_SetupRequest**
- **UMH_SetPowerState**

To avoid calling chains do not call **UMH_SetupRequestCpl** or **UMH_Transfer**, if the completion status of the callback function is not UMH_STATUS_SUCCESS.

See Also

UMH_EnumerateDevice (page 23)

UMH_SetConfiguration (page 24)

4.3 Enumeration Types

UMH_POWER_STATE

The UMH_POWER_STATE enumeration type defines different USB bus power states.

Definition

```
typedef enum {  
    UMH_POWER_RESUME,  
    UMH_POWER_SUSPEND  
} UMH_POWER_STATE;
```

Entries

UMH_POWER_RESUME

Send a resume signal on the USB bus, wakeup the device.

UMH_POWER_SUSPEND

Set the USB bus to suspended state.

See Also

[UMH_SetPowerState](#) (page 34)

UMH_DEVICE_STATE

The UMH_DEVICE_STATE enumeration type defines values that identify the current state of an USB device.

Definition

```
typedef enum {  
    UMH_DEVICE_DISCONNECT    = 0 ,  
    UMH_DEVICE_POWERED ,  
    UMH_DEVICE_DEFAULT ,  
    UMH_DEVICE_ADDRESSED ,  
    UMH_DEVICE_CONFIGURED  
} UMH_DEVICE_STATE;
```

Entries

UMH_DEVICE_DISCONNECT

Device is disconnected.

UMH_DEVICE_POWERED

Device is connected, powered and the USB bus has a high impedance state.

UMH_DEVICE_DEFAULT

Device has been reset, the USB bus works.

UMH_DEVICE_ADDRESSED

The device is addressed.

UMH_DEVICE_CONFIGURED

The device is configured.

Comments

If the device is configured data transfers can be started with the UMH_Transfer() function. In the addressed state only control endpoint transfers are allowed.

See Also

[UMH_GetDeviceState](#) (page 35)

4.4 Error Codes

UMH_STATUS_SUCCESS (0x0000L)

The operation has been successfully completed.

UMH_STATUS_ERROR (0x0001L)

The operation has been completed with a generic error.

UMH_STATUS_BUSY (0x0002L)

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

UMH_STATUS_INVALID_PARAM (0x0003L)

A parameter passed to the function was invalid.

UMH_STATUS_OVERRUN (0x0004L)

A data overrun error has been detected.

UMH_STATUS_TIMEOUT (0x0005L)

The operation has been timed out.

UMH_STATUS_COMPLETE (0x0006L)

A library operation has been completed.

UMH_STATUS_CRC (0x0007L)

A CRC error has been detected.

UMH_STATUS_STALL (0x0008L)

A STALL PID has been detected.

UMH_STATUS_DELAYED (0x0009L)

The completion routine is called to a later time.

UMH_STATUS_CANCELED (0x000AL)

The operation has been cancelled. If the USB device is removed from the HOST or the device is in the suspended state or a pending request is aborted the library completes this requests with UMH_STATUS_CANCELED.

UMH_STATUS_DATA_TOGGLE_MISMATCH (0x000BL)

A DATA toggle mismatch (DATA0/DATA1 tokens) has been detected.

UMH_STATUS_BITSTUFF (0x000CL)

A bit stuffing error has been detected.

5 Demo Application

The USB minihost demo application checks the USB port for a USB standard keyboard. If a standard keyboard is found the application sets a configuration and reads data from the keyboard. It transfers printable chars to ASCII codes and send it to UART 1 with a baudrate of 19,2 kbaud. Special keys like cursor moving, functions keys, or numblock keys are not translated.

5.1 HOST interface

The demo application uses the FUMA Library to build a HOST interface with the following behaviour.

- Supports low- and full speed keyboard devices.
- The elements of the interface descriptor must contain the values:
 - `bInterfaceClass=3`,
 - `bInterfaceSubClass=1`,
 - `bNumEndpoints=1`,
 - `bInterfaceProtocol=1`.
- The product or vendor ID is not checked.
- Supports removing and connecting of an USB device from the HOST while running.
- Reads the endpoint address and the transfer interval from the endpoint descriptor.
- Enables a interrupt IN endpoint transfer.
- No vendor requests are used.

The hardware is based on an USB1.1 Minihost controller MB90F337 with the GLYN evaluation board EVBMB90F335. It is possible that the software runs on other boards but it is not tested.

How to translate the demo application refer to [section 6](#).

5.2 Program flow

The DEMO application performs the following steps after reset:

- First initialise all interrupt request levels and enable the interrupt. `InitPWCTimer()` configures the timer for `UMH_StallExecutionUserCallback()` and `UmhInitUart1()` configures the UART to send ASCII codes. `InitIrqLevels()` and `__EI()` control the interrupt flags. `InitKbd()` set some global variables.
- **UMH_Init** initialises the FUMA library.
- **UMH_RemoveEndpoint** is called if the device is removed. Then the application waits for a connection.

- After the device has been connected the application waits for some milliseconds before call **UMH_EnumerateDevice** is called to start the USB communication with the keyboard device.
- USBKeyboardDetection() checks the interface- and endpoint descriptor.
If USBKeyboardDetection() returns successfully a data endpoint is added with UMH_AddEndpoint().
- Before the data transfer from the keyboard can be started the device is set in a configured state with **UMH_SetConfiguration**.
- If the device is configured and connected and the last transfer is not completed the data transfer from the keyboard is started with InterruptInStartTransfer() that calls **UMH_Transfer**.
- The transfer completion routine IntInCompletion() calls ProcessKbdPacket() to scan the incoming keyboard packet. If the user presses a key the ASCII code is sent to the UART1 and could be displayed with a terminal program.

5.3 Traces

The library and the demo program contain helpful traces to find out why a device does not work. The trace interface puts the traces to UART0 from the MB90330. The baudrate is 115200 baud, 8 data bits, 1 stop bit, no parity and no handshake signals. To enable the traces the compiler define `DBG=1` must be set and the debug version of the library has to be used. `<config.h>` contains traces that determine which traces are displayed. The define `TRACE_LIB_ENABLE` is set to 1 to see traces from the library code and the define `TRACE_USER_ENABLE` is set to 1 to see user traces. To see only errors and warnings of the library `TRACE_ONLY_LIB_ERRORS` is set to 1 and to see only errors and warning from the user code `TRACE_ONLY_USER_ERRORS` is set to 1. With the defines `LIB_DBG_MASK` and `USER_DBG_MASK1` it is possible to add or remove trace information from the library and the user code.

5.4 Running the demo application

5.4.1 Hardware requirements

- GLYN Demoboard EVBMB90F335 with MB90F337
- USB 1.1 keyboard
- USB cable
- Serial RS232 cable (1:1 extension cable male-female, no NULL modem cable)

5.4.2 Software requirements

- binary in the directory `\bin\fre\host_demoF337.mhx` (motorola hex file for MB90F337)
- FUJITSU Flash MCU programmer 16LX utility (Version V01L13 or higher)

5.4.3 Jumper settings

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

Jumper settings:

- JP1: If closed TX from UART1 is connected to DB9 connector X2.
- JP2: If closed RX from UART1 is connected to DB9 connector X2.
- JP3: during download 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
- JP12: 1-2
- JP13: 1-2
- JP14-JP16: not available
- JP17: 1-2

5.4.4 Programming

- Before starting the download set JP3 to 1-2 and check the other jumpers (see jumper settings).
- Connect a free serial COM interface from the PC with the RS232 connector X1 from the board.
- Connect the demo board with a USB cable to the PC or to an external power supply with a range from 9 to 12V.
- If no an external power supply is used a USB connection to a host is needed.
- After connecting the board with a power supply both green LED's and the red LED are on.
- Press the reset switch on the demo board.
- Open the motorola hex file \src\bin\fre\host_demoF337.mhx.
- Start the flash utility and open the file host_demoF337.mhx.

- 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>.
- Push the button "full operation" in the flash utility and wait for the end of the programming.
- Close the flash utility.
- Disconnect the Demoboard from USB.
- Set JP 3 to 2-3.
- Press the reset switch on the demoboard.
- Connect the host connector with a USB function.

Note if use another demo board the circuit must have a RS-232C signal driver for communication with the microcontroller UART. Set the target microcontroller to MB90F337 and set the crystal frequency to 6Mhz. If the target MB90F337 is not found the programmer tool must be updated. Check the COM port number in the SetEnvironment menu.

5.4.5 Starting the demo program

First connect the SUB-D connector X2 with the serial cable and a free RS232-interface. The baudrate for UART 1 is 19.2 kbaud. Do always use 1:1 cable male/female, not a NULL modem cable, with the GLYN demo board. Then start a terminal program (switch off all hardware handshakes and XON/XOFF). Connect the board with an external power supply. The LED's LED2 and LED3 should be on. Then connect the USB keyboard with the demo board and press a char key, e.g. "s". The character should now be displayed on the terminal window. If the terminal does not display a pressed key try to load the debug version of the demo program and check if the host can enumerate that keyboard. If the keyboard has special interface descriptor entries different from a normal USB HID-keyboard device then the function `USBKeyboardDetection()` returns an error.

6 Configuration and translation of the library

The header file `<umh_haldef.h>` in the source directory contains some defines which configure the behaviour of the library at compile time.

6.1 Hardware depended configurations

The timeouts are needed to control the USB HOST circuit in situations where the library waits for events from the HOST circuit.

- `MAX_ENDPOINTS`: defines the maximum numbers of endpoints including the control endpoint.
- `TOKEN_RETRY_NUMBER`: maximum number of token repetitions before returns
- `TIMEOUT_SET_ADDRESS`: maximum time in 0.1 millisecond units to wait for the Handshake status during a `SET_ADDRESS` request
- `TIMEOUT_EP0_TRANSFER`: maximum time in 0.1 millisecond units for a control endpoint transfer
- `TIMEOUT_RESUME_RESET`: maximum time in 0.1 millisecond units to wait for the reset or resume interrupt
- `USB_INT_LEVEL`: that is the interrupt level of the USB Host interrupts
- `EP0_MAX_PACKET_SIZE`: default maximum packet for endpoint zero for the USB HOST circuit During enumeration the correct value for endpoint zero is set.

The header file `<Osal.h>` contains two macros for Synchronisation the USB Host interrupts and the FUMA library interface.

- `OSALEnter`: Two implementations exist. The first one disables the USB Host interrupt and all interrupts with a lower or equal level. The second one disables all interrupts.
- `OSALLeave`: Two implementations exist. The first one restores the global interrupt enable flag and the second macro restores the interrupt level.

If the Compiler define `SYNC_EQUAL_LOWER_INTS` is not zero all interrupts are disabled.

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 `usbhost.wsp` contains the project for the library and the demo application. The demo application depends on the library project file `host_lib.prj`. The demo application has different targets to build the release and debug version for the MB90V330A and MB90F337. If the target is changed in the demo application the associated target in the library is set because of the dependency from the library.

The library project path is `\src\mb90\host\lib` and the demo program project path is `\src\mb90\host\demo`.

The project settings of the library and the demo differs in the Compiler defines `DBG` and `USB_EVA_BOARD`, the entries for the file names and different target MCU's. The compiler model is medium and is for both projects the same.

The sources are stored at the following locations:

library sources: `\src\mb90\host\lib\src`,

demo sources: `\src\mb90\host\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
<code>\src\bin\chk\host_demoV330.abs</code>	debug version of the demo for the evaluation chip MB90V330A
<code>\src\bin\fre\host_demoV330.abs</code>	release version of the demo for the evaluation chip MB90V330A
<code>\src\bin\chk\host_demoF337.abs</code>	debug version of the demo for the USB chip MB90F337
<code>\src\bin\fre\host_demoF337.abs</code>	release version of the demo for the USB chip MB90F337
<code>\src\bin\chk\host_libF337.lib</code>	debug version of the USB library for the USB chip MB90F337
<code>\src\bin\fre\host_libF337.lib</code>	release version of the USB library for the USB chip MB90F337
<code>\src\bin\chk\host_libF337.lib</code>	debug version of the USB library for the USB chip MB90F337
<code>\src\bin\fre\host_libF337.lib</code>	release version of the USB library for the USB chip MB90F337
<code>\src\bin\chk\host_libV330.lib</code>	debug version of the USB library for the evaluation chip MB90V330A
<code>\src\bin\fre\host_libV330.lib</code>	release version of the USB library for the evaluation chip MB90V330A

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

Address

Parameter of UMH_EnumerateDevice, [23](#)

Buffer

Parameter of UMH_SetupRequestCpl, [28](#)

Parameter of UMH_SetupRequest, [29](#)

Parameter of UMH_Transfer, [31](#)

BufferSize

Parameter of UMH_Transfer, [31](#)

BytesReturned

Parameter of UMH_SetupRequest, [29](#)

BytesTransferred

Parameter of UMH_COMPLETION, [39](#)

CompletionFunction

Parameter of UMH_AddEndpoint, [25](#)

Parameter of UMH_EnumerateDevice, [23](#)

Parameter of UMH_SetConfiguration, [24](#)

Parameter of UMH_SetupRequestCpl, [28](#)

ConfigurationValue

Parameter of UMH_SetConfiguration, [24](#)

Context

Parameter of UMH_COMPLETION, [39](#)

Parameter of UMH_Transfer, [31](#)

DeviceState

Parameter of UMH_GetDeviceState, [35](#)

Parameter of UMH_Init, [22](#)

EndpointAddress

Parameter of UMH_AddEndpoint, [25](#)

FifoSize

Parameter of UMH_AddEndpoint, [25](#)

Flags

Parameter of UMH_DEVICE_STATE_CALLBACK, [38](#)

Parameter of UMH_GetDeviceState, [35](#)

Handle

Parameter of UMH_AbortTransfer, [32](#)

Parameter of UMH_AddEndpoint, [25](#)

Parameter of UMH_RemoveEndpoint, [27](#)

Parameter of UMH_ResetDataToggleBit, [33](#)

Parameter of UMH_Transfer, [31](#)

Interval

Parameter of UMH_AddEndpoint, [25](#)

PowerState

- Parameter of UMH_SetPowerState, [34](#)
- Setup[8]
 - Parameter of UMH_SetupRequestCpl, [28](#)
 - Parameter of UMH_SetupRequest, [29](#)
- Status
 - Parameter of UMH_COMPLETION, [39](#)
 - Parameter of UMH_STATUS_COMPLETION, [40](#)
- time
 - Parameter of UMH_StallExecutionUserCallback, [37](#)
- UMH_AbortSetupRequest, [30](#)
- UMH_AbortTransfer, [32](#)
- UMH_AddEndpoint, [25](#)
- UMH_COMPLETION, [39](#)
- UMH_DEVICE_ADDRESSED
 - Entry of UMH_DEVICE_STATE, [42](#)
- UMH_DEVICE_CONFIGURED
 - Entry of UMH_DEVICE_STATE, [42](#)
- UMH_DEVICE_DEFAULT
 - Entry of UMH_DEVICE_STATE, [42](#)
- UMH_DEVICE_DISCONNECT
 - Entry of UMH_DEVICE_STATE, [42](#)
- UMH_DEVICE_POWERED
 - Entry of UMH_DEVICE_STATE, [42](#)
- UMH_DEVICE_STATE_CALLBACK, [38](#)
- UMH_DEVICE_STATE, [42](#)
- UMH_EnumerateDevice, [23](#)
- UMH_GetDeviceState, [35](#)
- UMH_Init, [22](#)
- UMH_POWER_RESUME
 - Entry of UMH_POWER_STATE, [41](#)
- UMH_POWER_STATE, [41](#)
- UMH_POWER_SUSPEND
 - Entry of UMH_POWER_STATE, [41](#)
- UMH_RemoveEndpoint, [27](#)
- UMH_ResetDataToggleBit, [33](#)
- UMH_SetConfiguration, [24](#)
- UMH_SetPowerState, [34](#)
- UMH_SetupRequestCpl, [28](#)
- UMH_SetupRequest, [29](#)
- UMH_StallExecutionUserCallback, [37](#)
- UMH_STATUS_BITSTUFF, [44](#)
- UMH_STATUS_BUSY, [43](#)
- UMH_STATUS_CANCELED, [44](#)
- UMH_STATUS_COMPLETE, [43](#)
- UMH_STATUS_COMPLETION, [40](#)
- UMH_STATUS_CRC, [43](#)

UMH_STATUS_DATA_TOGGLE_MISMATCH, [44](#)
UMH_STATUS_DELAYED, [44](#)
UMH_STATUS_ERROR, [43](#)
UMH_STATUS_INVALID_PARAM, [43](#)
UMH_STATUS_OVERRUN, [43](#)
UMH_STATUS_STALL, [43](#)
UMH_STATUS_SUCCESS, [43](#)
UMH_STATUS_TIMEOUT, [43](#)
UMH_Transfer, [31](#)