THESYCON
Thesycon® Systemsoftware & Consulting GmbH

FUMA
Fujitsu USB Minihost API

for MB90330, MB96330 and MB91660 series

Reference Manual

Version: 2.1.4
Date: 05 February 2009

Authors: Steffen Weiss

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

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

http://www.thesycon.de

Copyright (c) 2006-2009 by Thesycon® Systemsoftware & Consulting GmbH

All Rights Reserved

Disclaimer

Information in this document is subject to change without notice. No part of this manual may be reproduced, stored in a retrieval system, or transmitted in any form or by any means electronic or mechanical, including photocopying and recording for any purpose other than the purchaser's personal use, without prior written permission from Thesycon Systemsoftware & Consulting GmbH. The software described in this document is furnished under the software license agreement distributed with the product. The software may be used or copied only in accordance with the terms of the license.

Trademarks

The following trade names are referenced throughout this manual:

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

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

Contents

Table of Contents 7

1 Introduction 9

2 Overview 11

2.1 Compiler 11

2.2 Features 11

2.3 Restrictions 12

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 30

UMH_AbortSetupRequest 32

UMH_Transfer 33
UMH_AbortTransfer 34
UMH_ResetDataToggleBit 35
UMH_SetPowerState 36
UMH_GetDeviceState 37
Umh_DeviceStateStr 38
4.2 API callback functions 39
UMH_StallExecutionUserCallback 40
UMH_DEVICE_STATE_CALLBACK 41
UMH_COMPLETION 42
UMH_STATUS_COMPLETION 43
4.3 Enumeration Types 44
UMH_POWER_STATE 44
UMH_DEVICE_STATE 45
4.4 Error Codes 46
UMH_STATUS_SUCCESS (0x0000L) 46
UMH_STATUS_ERROR (0x0001L) 46
UMH_STATUS_BUSY (0x0002L) 46
UMH_STATUS_INVALID_PARAM (0x0003L) 46
UMH_STATUS_OVERRUN (0x0004L) 46
UMH_STATUS_TIMEOUT (0x0005L) 46
UMH_STATUS_COMPLETE (0x0006L) 46
UMH_STATUS_CRC (0x0007L) 46
UMH_STATUS_STALL (0x0008L) 46
UMH_STATUS_DELAYED (0x0009L) 46
UMH_STATUS_CANCELED (0x000AL) 47
UMH_STATUS_DATA_TOGGLE_MISMATCH (0x000BL) 47
UMH_STATUS_BITSTUFF (0x000CL) 47
UMH_STATUS_LENGTH (0x000DL) 47

5 Demo Applications **49**
5.1 USB HID demo application 49
5.1.1 USB Host interface 49
5.1.2 Program flow 49
5.1.3 Running the demo program 50

5.2 USB mass storage application 50
5.2.1 USB Host interface 50
5.2.2 Program flow 51
5.2.3 Running the demo program 52
5.3 Traces 52

6 Configuration and translation of the library 53
6.1 Hardware depended configurations 53

7 Related Documents 55

Index 57

1 Introduction

FUMA is a generic Universal Serial Bus (USB) Mini HOST library for the 16-bit serie FUJITSU MB90330 and MB96330 and the 32-bit serie FUJITSU MB91660. It covers the entire USB Mini HOST 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 FUMA library is designed to cover a USB host controller in an embedded environment. It consists of different modules which can be adjusted and combined to satisfy your requirements. This modular concept can easily be enhanced by your own implementations.

2.1 Compiler

Depending from the used microcontroller different compiler are in use.

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

2.2 Features

The FUMA library provides the following features:

- initialization of the USB Mini HOST
- support of USB 1.1 full speed and low speed
- 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 used microcontroller allow to work as USB Host or USB Function. It is not possible with the library to exit from the USB Host during runtime and to switch to the USB function.
- isochronous transfer is not supported
- supports one USB device at the same time
- does not support Hub devices
- not all host controller supports low speed

2.4 Performance

The maximum bandwidth for bulk transfer is about 270 kbyte/s on the MB90F337. To obtain the maximum bandwidth you should use the firmware release version. The data endpoint buffer size should be at least 512 bytes.

3 Architecture

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 USB host device registers. 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.

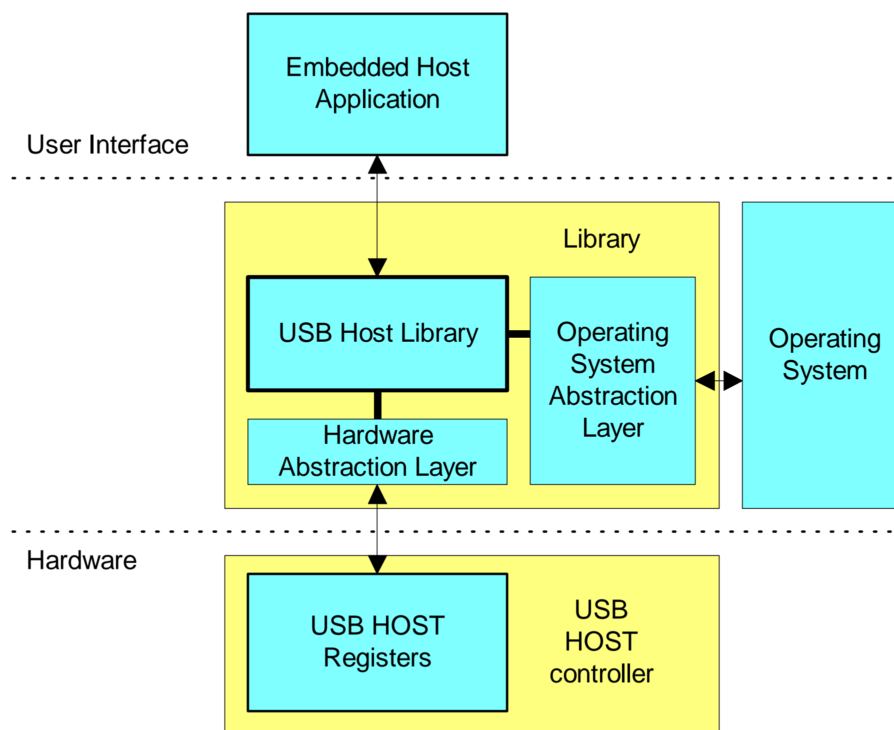

Figure 1: FUMA Software Architecture

The library does not require any external libraries or function calls out of <osal_impl.c>. The OSAL layer defines a debug print function. A system independent implementation of the print function is the function OSALPrint implemented in <osal_impl.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). This signal is detected from the USB Host controller. Then the internal function UmhDevConnect() sets the USB operation clock and starts needed frame token. The event UMH_EVENT_CONNECT informs the user about connection of an USB device. The internal object state of the connected USB device is set to **UMH_DEVICE_STATE_POWERED**.

- Device disconnect event
  UmhDevRemove() is called if a connected USB device is removed. The event UMH_EVENT_REMOVED informs the user about removing of a USB device.

- 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.

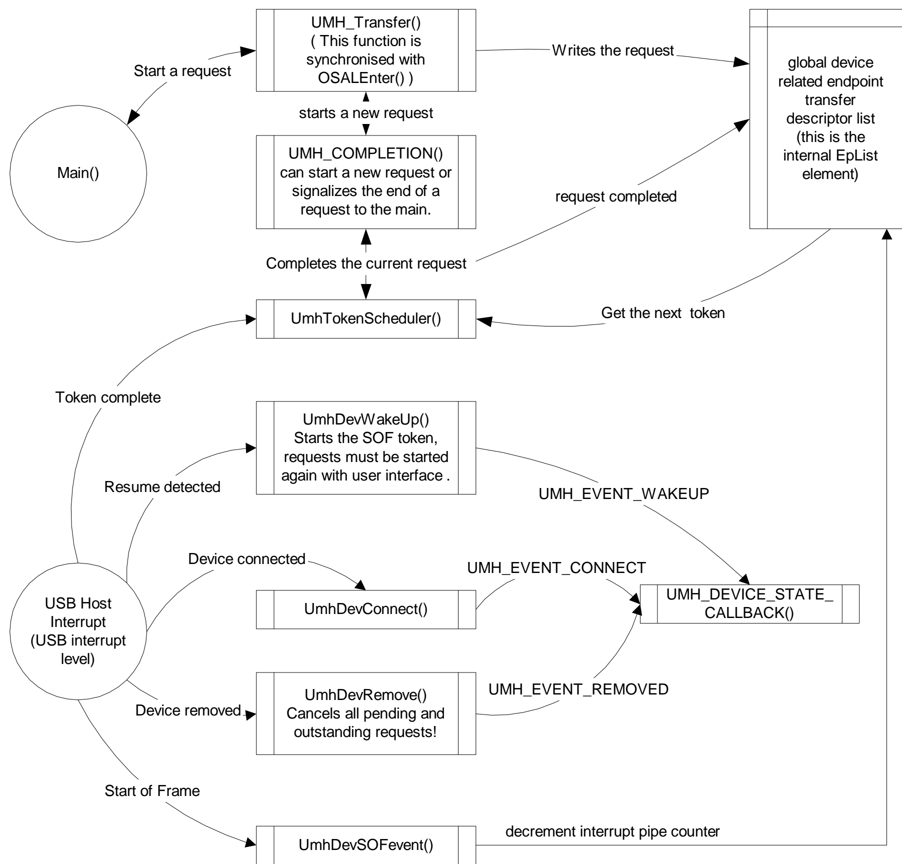

Figure 2: USB HOST events and the library

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 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.

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

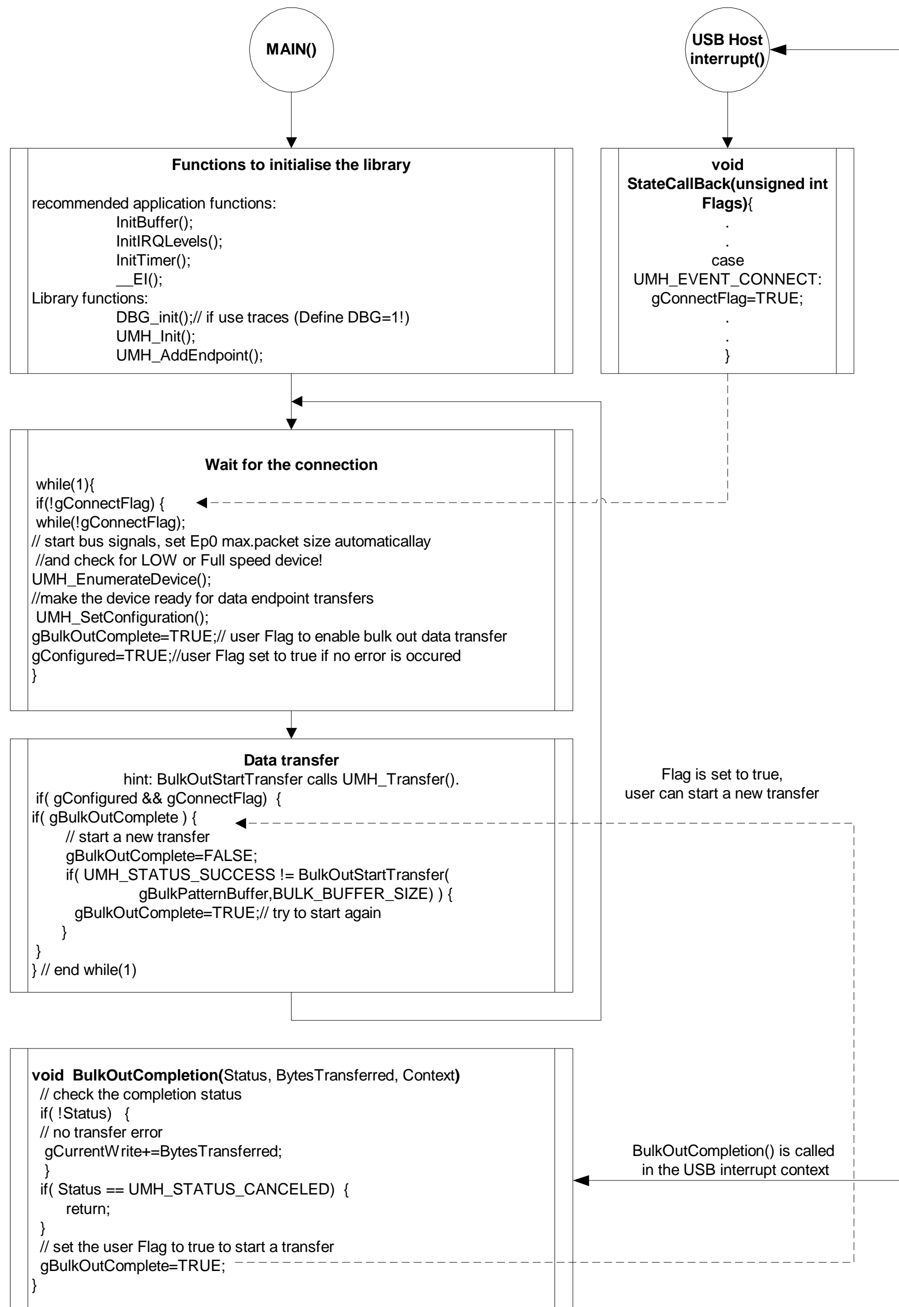

Figure 3: Program Flow in the main() context

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 initialization. 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 library 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()` dispatches the event to the correct internal function so the most library functions runs in the context of the interrupt service routine where `UmhLibInterrupt()` is called.

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 if a library function is called from the application during the run time the USB host interrupt is disabled. It is not allowed to call library functions that wait for an USB event in an interrupt context with a higher or equal priority as the USB host interrupt(that would cause to a deadlock). In the programming interface the allowed caller process context is declared.

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

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 disables the interrupt of all the USB Host interrupt sources. The OSALLeave macro enables all USB Host interrupt sources, see also the implementation of this macros in the demo application. If `UmhLibInterrupt()` is called always in a timer interrupt the OSALEnter macro disables the timer interrupt and OSALLeave enables the interrupt again.

It is needed to synchronize code between the main() loop and code that runs in the USB Host interrupt context how the FUMA callback functions the global interrupt can be disabled and enabled again in the main() loop context. The time between disable and enable of the global interrupt should be very short.

3.5.2 With operating system

The following implementation depends fro the used operating system. The OSALEnter macro 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

Initialize 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 initialization. No device related actions are done. Do not call this function in the USB interrupt context.

See Also

UMH_DEVICE_STATE_CALLBACK (page 41)

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 43)

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 43)

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 34)
UMH_STATUS_COMPLETION (page 43)

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 33)

UMH_SetupRequestCpl

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

Definition

```
UMH_STATUS
UMH_SetupRequestCpl(
      SETUP_PACKET Setup,
      unsigned char* Buffer,
      UMH_COMPLETION* CompletionFunction,
      void*  Context
      );
```

Parameters

Setup
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.

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

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 32)
UMH_EnumerateDevice (page 23)

UMH_SetupRequest

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

Definition

```
UMH_STATUS
UMH_SetupRequest(
      SETUP_PACKET* Setup,
      unsigned char* Buffer,
      unsigned int* BytesReturned
      );
```

Parameters

Setup
Setup packet. The buffer length is given in the length field of the setup packet. 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. If the setup length field is zero this parameter can be zero!

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. In that case he control transfer is aborted intern.
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. If return value is UMH_STATUS_TIMEOUT the request has been aborted. If return value is UMH_STATUS_BUSY call the function again.

See Also

UMH_AbortSetupRequest (page 32)
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_SUCCESS - if successful, request has been completed or no request is pending
UMH_STATUS_DELAYED - the completion routine is called to 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 sucessfull,
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 34)

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, request has been completed or no request is pending
UMH_STATUS_DELAYED - the pending request is completed to a later time.

Comments

If returns with success the completion function has been called.

See Also

UMH_Transfer (page 33)

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 called if the application sends a clear feature endpoint stall request.

See Also

UMH_Transfer (page 33)
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 on this host the first time is suspended and then disconnected from the USB bus this USB function detects a resume (if the USB cable is removed there is for a short time a resume signal) and then a remove. 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 37)
UMH_POWER_STATE (page 44)

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 36)
UMH_DEVICE_STATE (page 45)

Umh_DeviceStateStr

Returns the current device state as an string.

Definition

```
const char*
Umh_DeviceStateStr(
      int x
      );
```

Parameter

x
Stae of type UMH_DEVICE_STATE.

Comments

Can be used to trace the current state of the library host object. UMH_GetDeviceState() returns the current state. This funciton is only valid if the C-Define DBG is 1.

See Also

UMH_GetDeviceState (page 37)
UMH_DEVICE_STATE (page 45)

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. This function is not called in the USB interrupt 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 30)

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 canceled.

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.

UMH_EVENT_OVERCURRENT if a overcurrent condition has been detected on the host connector. This is a optional event and the application generates this event by calling the implementation of UMH_DEVICE_STATE_CALLBACK with parameter UMH_EVENT_OVERCURRENT. The user application can then handle this event.

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 36)

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 37)

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 canceled. 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.

UMH_STATUS_LENGTH (0x000DL)

A length error has been detected.

5 Demo Applications

Two demo applications are available, a USB HID (Human interface device class) demo application and a USB mass storage class demo application. The HID application is implemented in the platform plt_MB90 and plt_MB96. The USB mass storage application is implemented in the platform plt_mb91_bbf2004.

5.1 USB HID demo application

The common code of the demo application is located in the APP_HID directory. The implementation of the demo application is done in the platform directory. If a standard USB keyboard is found the demo reads data from the keyboard. Printable character are put out to a free RS232 interface on the board. Special keys like cursor moving, functions keys and not printable keys (Alt, Alt-Gr, Ctrl, Escape, Break, print..) are not translated. The left and right shift key and the caps lock key are used to distinguish between uppercase and lowercase characters. The keypad from the keyboard returns always the printable characters "0..9+-..". The "return key" is always translated in the ASCII-Code 0x0d. Some terminals programs adds a line feed to a incoming line end character, so the cursor begins at a new line if the return key is pressed. The keyboards LEDs for NUM LOCK, SCROLL LOCK and CAPS LOCK are switched on or off that depends from the state of the keys NUM LOCK, CAPS LOCK and SCROLL LOCK.

5.1.1 USB Host interface

The demo application uses the FUMA Library to build a HOST interface with the following behaviour. The used USB keyboard must have the following characteristics:

- USB low- or full speed
- Important interface descriptor values:
  - bInterfaceClass=3,
  - bInterfaceSubClass=1,
  - bNumEndpoints=1,
  - bInterfaceProtocol=1.
- Product or Vendor ID are not checked.

The application uses a interrupt IN request to get informations from the keyboard. Vendor requests are not used.

5.1.2 Program flow

The HID application performs the following steps after reset:

- First initialize 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** initializes 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 a key is pressed the Key Code is sent to a RS232 interface.

5.1.3 Running the demo program

If need Debug Traces an additional RS232 cable must be connected with a free RS232 port on the PC. To see the Key Codes of the keyboard the correct RS232 interface on the demo board must be connected with the PC, see also the platfrom readme. Before the program is started the USB keyboard must be connected with the board. Then downlaods the firmware and start it, see also the platform Readme.

5.2 USB mass storage application

This is a simple demo to see the functionality of the host library. The common code of the demo application is located in the app_mass_storage directory. The platform specific implementation is located in the platform directory. Not all memory sticks are supported. If the device is successfully enumerated the SCSI INQUIRY command is send to the device via the mass storage bulk only protocol. The content of the INQUIRY responce is printed out to the RS232 trace-interface (115200,n,1,8). After a delay of about one second the same command is repeated.

5.2.1 USB Host interface

The demo application uses the FUMA Library to build a HOST interface with the following behaviour. The minimum requirements of the used USB memory stick are:

- USB full speed is supported
- One configuration with one or more interfaces, Alternate settings in the same interface are not supported.
- One interface must contains following values:
  - bInterfaceClass=0x08,
  - bInterfaceSubClass=0x05
    bInterfaceSubClass=0x06
  - bNumEndpoints=2, this value shall be at least 2.
  - bInterfaceProtocol=0x50
- Product or Vendor ID are not checked.

The application uses a bulk IN and a Bulk OUT endpoint get informations from the USB memory stick.

5.2.2 Program flow

The application performs the following steps after reset:

- Before the USB library can be used some hardware specific initializations are be done in HW_init(). In this function the USB clock is set up,the USB block is enabled and additional interrupt service routines e.g. to detect Vusb are installed. A serial output for traces is also installed.
- Enable of global interrupts and setting of trace masks. MsInit() initialize the application
- **UMH_Init** initialize the host library.
- EnableVbus() helper function that puts outs +5V to the Vusb wire.
- To recover from a overcurrent condition (from power IC MB3841) a short routine in the main loop disables Vusb for a short time before Vusb is enabled again.
- MsProcessDevice() is called in the main loop. This is a state machine that executes all needed USB transfers with the device.
- To inform the application about USB host events MsRemoveDevice(), MsConnectDevice() and MsWakeup() are needed. This functions are called from StateCallBack().

If the device is connected the device is enumerated with UMH_EnumerateDevice(). MsCheck-DeviceSetConfiguration() checks the descriptors configure the device. The memory stick is reset with a USB mass storage reset. After tracing the INQUIRY data the state machine waits about one second before the next INQUIRY command is executed. If the device is removed the state machine goes into the state Dev_Disconnect. Then the application waits for a connect event.

5.2.3 Running the demo program

After starting the application a memory stick must be inserted. If the memory stick does not work it is useful to use the debug version. By default warning and errors traces are enabled. The memory stick can be removed and plugged in during runtime. Data on the memory stick are not changed.

5.3 Traces

The library and all demo applications contains helpful traces to find out why a device does not work. Only in the debug version all traces can be enabled, see also calling of trace functions in the main routines ans see also the plaftform specific Readme.

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 following timeouts are needed to control the USB HOST circuit in situations where the library waits for events from the HOST circuit. For more informations see also in the configuration file platform directory of the source code.

- 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.

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, 30
Parameter of UMH_Transfer, 33
BufferSize
Parameter of UMH_Transfer, 33
BytesReturned
Parameter of UMH_SetupRequest, 30
BytesTransferred
Parameter of UMH_COMPLETION, 42

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, 42
Parameter of UMH_SetupRequestCpl, 28
Parameter of UMH_Transfer, 33

DeviceState
Parameter of UMH_GetDeviceState, 37
Parameter of UMH_Init, 22

EndpointAddress
Parameter of UMH_AddEndpoint, 25

FifoSize
Parameter of UMH_AddEndpoint, 25
Flags
Parameter of UMH_DEVICE_STATE_CALLBACK, 41
Parameter of UMH_GetDeviceState, 37

Handle
Parameter of UMH_AbortTransfer, 34
Parameter of UMH_AddEndpoint, 25
Parameter of UMH_RemoveEndpoint, 27
Parameter of UMH_ResetDataToggleBit, 35
Parameter of UMH_Transfer, 33

Interval
Parameter of UMH_AddEndpoint, 25

PowerState
Parameter of UMH_SetPowerState, 36

Setup
Parameter of UMH_SetupRequestCpl, 28
Parameter of UMH_SetupRequest, 30
Status
Parameter of UMH_COMPLETION, 42
Parameter of UMH_STATUS_COMPLETION, 43

time
Parameter of UMH_StallExecutionUserCallback, 40

UMH_AbortSetupRequest, 32
UMH_AbortTransfer, 34
UMH_AddEndpoint, 25
UMH_COMPLETION, 42
UMH_DEVICE_ADDRESSED
Entry of UMH_DEVICE_STATE, 45
UMH_DEVICE_CONFIGURED
Entry of UMH_DEVICE_STATE, 45
UMH_DEVICE_DEFAULT
Entry of UMH_DEVICE_STATE, 45
UMH_DEVICE_DISCONNECT
Entry of UMH_DEVICE_STATE, 45
UMH_DEVICE_POWERED
Entry of UMH_DEVICE_STATE, 45
UMH_DEVICE_STATE_CALLBACK, 41
UMH_DEVICE_STATE, 45
Umh_DeviceStateStr, 38
UMH_EnumerateDevice, 23
UMH_GetDeviceState, 37
UMH_Init, 22
UMH_POWER_RESUME
Entry of UMH_POWER_STATE, 44
UMH_POWER_STATE, 44
UMH_POWER_SUSPEND
Entry of UMH_POWER_STATE, 44
UMH_RemoveEndpoint, 27
UMH_ResetDataToggleBit, 35
UMH_SetConfiguration, 24
UMH_SetPowerState, 36
UMH_SetupRequestCpl, 28
UMH_SetupRequest, 30
UMH_StallExecutionUserCallback, 40
UMH_STATUS_BITSTUFF, 47
UMH_STATUS_BUSY, 46
UMH_STATUS_CANCELED, 47
UMH_STATUS_COMPLETE, 46

UMH_STATUS_COMPLETION, 43
UMH_STATUS_CRC, 46
UMH_STATUS_DATA_TOGGLE_MISMATCH, 47
UMH_STATUS_DELAYED, 47
UMH_STATUS_ERROR, 46
UMH_STATUS_INVALID_PARAM, 46
UMH_STATUS_LENGTH, 47
UMH_STATUS_OVERRUN, 46
UMH_STATUS_STALL, 46
UMH_STATUS_SUCCESS, 46
UMH_STATUS_TIMEOUT, 46
UMH_Transfer, 33

x
Parameter of Umh_DeviceStateStr, 38