Contents

Preface ........................................................................................................................................................................  7

Introduction to IAR PowerPac USB Host ....................................................................................................................  9

What is IAR PowerPac USB Host ..................................................................................................................................  9
Features ...........................................................................................................................................................................  9

Getting started .......................................................................................................................................................... 11

Starting with an example project .................................................................................................................................. 11
Adding IAR PowerPac USB Host to an existing project ................................................................................................. 14
  Naming conventions for prebuilt IAR PowerPac USB Host libraries ........................................................................ 14
  Adding the USB host stack to an existing project ........................................................................................................ 14

Example applications .................................................................................................................................................. 15

Overview ....................................................................................................................................................................... 15

USB Host Core Functions ........................................................................................................................................... 17

Management Functions .................................................................................................................................................. 17
  USBH_Init() ............................................................................................................................................................ 17
  USBH_Exit() .......................................................................................................................................................... 17
  USBH_EnumerateDevices() ....................................................................................................................................... 17

API Functions .................................................................................................................................................................. 18
  USBH_CreateInterfaceList() .................................................................................................................................. 18
  USBH_DestroyInterfaceList() .................................................................................................................................. 19
  USBH_GetInterfaceID() ......................................................................................................................................... 19
  USBH_GetInterfaceInfo() ....................................................................................................................................... 20
  USBH_RegisterPnPNotification() .......................................................................................................................... 20
  USBH_UnregisterPnPNotification() ......................................................................................................................... 20
  USBH_RegisterEnumErrorNotification() .............................................................................................................. 21
  USBH_UnregisterEnumErrorNotification() ........................................................................................................... 21
  USBH_RepeatedEnumError() ................................................................................................................................ 21
  USBH_OpenInterface() ........................................................................................................................................... 21
  USBH_CloseInterface() .......................................................................................................................................... 22
  USBH_GetDeviceDescriptor() .................................................................................................................................. 22
  USBH_GetCurrentConfigurationDescriptor() .................................................................................................... 23
  USBH_GetInterfaceDescriptor() ................................................................................................................................ 23
  USBH_GetEndpointDescriptor() ................................................................................................................................ 24
  USBH_GetSerialNumber() ....................................................................................................................................... 24
  USBH_GetSpeed() ..................................................................................................................................................... 25
  USBH_GetFrameNumber() ...................................................................................................................................... 25
  USBH_GetInterfaceIDByHandle() .......................................................................................................................... 26
  USBH_SubmitUrb() ................................................................................................................................................... 26
  USBH_SetTraceMask() ............................................................................................................................................ 27
  USBH_GetStatusStr() .............................................................................................................................................. 28

Data Structures ............................................................................................................................................................ 29
  USBH_INTERFACE_MASK ......................................................................................................................................... 29
  USBH_INTERFACE_INFO .......................................................................................................................................... 30
  USBH_ENUM_ERROR ................................................................................................................................................ 31
USBH_EP_MASK .................................................................................................................................32
USBH_CONTROL_REQUEST ..................................................................................................................33
USBH_BULK_INT_REQUEST ..................................................................................................................34
USBH_ISO_FRAME ..............................................................................................................................34
USBH_ISO_REQUEST ............................................................................................................................34
USBH_ENDPOINT_REQUEST ..................................................................................................................35
USBH_SET_CONFIGURATION ..................................................................................................................35
USBH_SET_INTERFACE ...........................................................................................................................36
USBH_SET_POWER_STATE ......................................................................................................................36
URB ........................................................................................................................................................36
USBH_PNP_NOTIFICATION .......................................................................................................................37
USBH_HEADER ........................................................................................................................................37
USBH_SPEED ..........................................................................................................................................38
USBH_PNP_EVENT .....................................................................................................................................38
USBH_FUNCTION ......................................................................................................................................38
USBH_POWER_STATE ...............................................................................................................................40

**Function Types** ..................................................................................................................................40
USBH_ON_PNP_EVENT_FUNC ..................................................................................................................40
USBH_ON_ENUM_ERROR_FUNC ................................................................................................................41
USBH_ON_COMPLETION_FUNC ................................................................................................................41

**Use of undocumented functions** .......................................................................................................41

**Mass Storage Device** ..........................................................................................................................43

**Introduction** .........................................................................................................................................43
Overview ..................................................................................................................................................43
Features ...................................................................................................................................................43
Restrictions ...............................................................................................................................................43
Requirements ...........................................................................................................................................43
Example code ..........................................................................................................................................43
Supported Protocols ...............................................................................................................................43

**API Functions** ....................................................................................................................................44
USBH_MSD_Init() .....................................................................................................................................44
USBH_MSD_Init() .....................................................................................................................................45
USBH_MSD_ReadSectors() ........................................................................................................................45
USBH_MSD_WriteSectors() .......................................................................................................................46
USBH_MSD_GetUnitInfo() .........................................................................................................................46
USBH_MSD_GetStatus() ...........................................................................................................................47

**Data Structures** ..................................................................................................................................47
USBH_MSD_UNIT_INFO ..............................................................................................................................47

**Function Types** ..................................................................................................................................48
USBH_MSD_LUN_NOTIFICATION_FUNC ......................................................................................................48

**Error Codes** .......................................................................................................................................48
USBH_MSD_STATUS_SUCCESS ....................................................................................................................49
USBH_MSD_STATUS_ERROR ........................................................................................................................49
USBH_MSD_STATUS_PARAMETER .............................................................................................................49
USBH_MSD_STATUS_LENGTH ....................................................................................................................49
USBH_MSD_STATUS_TIMEOUT ..................................................................................................................49
USBH_MSD_STATUS_COMMAND_FAILED ...................................................................................................49
USBH_MSD_STATUS_INTERFACE_PROTOCOL ............................................................................................49
USBH_MSD_STATUS_INTERFACE_SUB_CLASS ............................................................................................49
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>64</td>
</tr>
<tr>
<td>Related Documents</td>
<td>65</td>
</tr>
<tr>
<td>Glossary</td>
<td>67</td>
</tr>
<tr>
<td>Index</td>
<td>69</td>
</tr>
</tbody>
</table>
Preface

Welcome to the IAR PowerPac™ USB Host User Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use IAR PowerPac USB Host to best suit your application requirements.

Who should read this guide

You should read this guide if you plan to develop a file system using IAR PowerPac USB Host and need to get detailed reference information about it. This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler)
- The C programming language
- The target processor
- Basic understanding of USB.

If you feel that your knowledge of C is not sufficient, we recommend *The C Programming Language* by Kernighan and Richie (ISBN 0-13-1103628), which describes the standard in C-programming and, in newer editions, also covers the ANSI C standard.

The chapter *Background information* on page 11 provides information about USB in general and the mass storage class in particular and might be of interest if you are new to USB.

How to use this guide

This guide explains all the functions and macros that IAR PowerPac USB Host offers. It assumes that you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Document conventions

**TYPOGRAPHIC CONVENTIONS FOR SYNTAX**

This guide uses the following typographic conventions:

<table>
<thead>
<tr>
<th>Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Keyword</strong></td>
<td>Text that you enter at the command-prompt or that appears on the display (that is system functions, file- or pathnames).</td>
</tr>
<tr>
<td><strong>Parameter</strong></td>
<td>Parameters in API functions.</td>
</tr>
<tr>
<td><strong>Sample</strong></td>
<td>Sample code in program examples.</td>
</tr>
<tr>
<td><strong>Reference</strong></td>
<td>Reference to chapters, tables and figures or other documents.</td>
</tr>
<tr>
<td><strong>GUIDElement</strong></td>
<td>Buttons, dialog boxes, menu names, menu commands.</td>
</tr>
<tr>
<td><strong>Emphasis</strong></td>
<td>Very important sections</td>
</tr>
</tbody>
</table>

*Table 1: Typographic conventions*
Introduction to IAR PowerPac USB Host

This chapter provides an introduction to using IAR PowerPac USB Host. It explains the basic concepts behind IAR PowerPac USB Host.

What is IAR PowerPac USB Host

IAR PowerPac USB Host is a CPU-independent USB Host stack.

IAR PowerPac USB Host is a high-performance library that has been optimized for speed, versatility and small memory footprint.

Features

IAR PowerPac USB Host is written in ANSI C and can be used on virtually any CPU.

Some features of IAR PowerPac USB Host:

- ISO/ANSI C source code
- High performance.
- Small footprint.
- No configuration required.
- Runs "out-of-the-box".
- Control, bulk and interrupt transfers
- Very simple host controller driver structure.
- USB Mass Storage Device Class available
- Works seamlessly with embOS and emFile (for MSD)
- Support for class drivers
- Support for external USB hub devices
- Support for devices with alternate settings
- Support for multi-interface devices
- Support for multi-configuration devices
- Royalty-free.
Getting started

This chapter provides a step-by-step introduction to using IAR PowerPac USB Host.

Starting with an example project

PowerPac includes an example workspace and projects for every supported CPU. We recommend to use these as a starting point for all your applications.

To get your new application running, you should proceed as follows:

1. Create a work directory for your application, for example c:\work
2. In the IAR Embedded Workbench IDE, select Example applications in the Startup dialog box.
3. If the Startup dialog box is closed, choose Help | Startup Screen to open it.
4. Select BoardSupport from the Example Applications list.
5. Select the project which is consistent to your hardware. Choose a destination folder for your project, for example c:\work. After generating the project of your choice, the screen should look for example like this:
6 Use drag and drop to move the RTOS start application (depending on start project either `Start_LED Blink.c` or `Start_2Tasks.c`) from the folder `Application` into the folder `Application\Excluded`. All IAR PowerPac USB Host example applications have the prefix `OS_USBH_`.

Move the IAR PowerPac USB Host example application of choice from the folder `Excluded` into the folder `Application`. For example, include `OS_USBH_HID.c` into your project. `OS_USBH_HID.c` adds support for human interface devices (HID) like mouses or keyboards to your project.
7 Check the configuration file `USBH_Config_<Targetname>.c`. All supplied projects come with a default configuration which matches the requirements of the example applications.

There is no further configuration necessary. Build the project. It should compile without errors and warnings.

8 Download the application into your target and start execution. You can connect every standard mouse or keyboard to your target. The following screenshot shows the terminal I/O output of the example application `OS_USBH_HID.c` if a mouse is connected to your board.
Adding IAR PowerPac USB Host to an existing project

You can easily get access to functionality in IAR PowerPac USB Host by adding a prebuilt library to your existing project.

**NAMING CONVENTIONS FOR PREBUILT IAR POWERPAC USB HOST LIBRARIES**

IAR PowerPac USB Host is shipped with different CPU-specific prebuilt libraries.

The name of the libraries consists of the following parts:

<ProductSpecifier><CPUSpecificBuildOptions><ConfigurationSpecifier>.<extension>

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductSpecifier</td>
<td>Type of library.</td>
<td>usbh: IAR PowerPac USB Host library</td>
</tr>
<tr>
<td>CPUSpecificBuildOptions</td>
<td>CPU and compiler-specific options.</td>
<td>For a detailed list of valid combinations, refer to the CPU and compiler-specific guide for the IAR PowerPac RTOS.</td>
</tr>
<tr>
<td>ConfigurationSpecifier</td>
<td></td>
<td>r: Release (should be used with RTOS release libraries.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d: Debug build of the IAR PowerPac USB Host libraries.</td>
</tr>
<tr>
<td>extension</td>
<td>The extension is a suffix to the name of a file applied to indicate its type.</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Library naming conventions

**ADDING THE USB HOST STACK TO AN EXISTING PROJECT**

If you want to use the USB host stack in an existing project, you have to add the IAR PowerPac USB Host library consistent to your project requirements.

1. Select **Project | Add files** and select **Library/Object Files (*.r*, *.a, *.o)** in the **Files of type** list. The default location of the prebuilt IAR PowerPac USB Host libraries is **Powerpac\USBH\Lib**. Choose the library of your choice and confirm it with the **Open** button.

2. Add the compatible configuration file to your project (for example, **USBH_Config_LPC17xx.c**). The configuration file is located in the subdirectory **Setup** of corresponding example project.

3. You have to add the include path to your project settings. The include path is the path in which the compiler looks for include files. In cases where the included files (typically header files, .h) do not reside in the same directory as the C file to compile, an include path needs to be set. To build the project with the added library, choose **Project | Options | C/C++ Compiler | Preprocessor** and add the following path to the **Additional include directories** field: **$TOOLKIT_DIR\PowerPac\USBH\Inc**. If you have changed the location of the PowerPac directory, change the path accordingly.

4. Include the required header file **USBH.h** in your source file. Compile your project. You should be able to build the project without any error or warning messages.
Example applications

In this chapter, you will find a description of each IAR PowerPac USB Host example application.

Overview

Various example applications for IAR PowerPac USB Host are supplied. These can be used for testing the correct installation and proper function of the device running IAR PowerPac USB Host.

The following start application files are provided:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_USBH_HID.c</td>
<td>Adds support for a human interface device (mouse or keyboard) to your project.</td>
</tr>
<tr>
<td>OS_USBH_MSD.c</td>
<td>Adds support for mass storage devices (USB sticks, HDDs) to your project.</td>
</tr>
</tbody>
</table>

Table 3: IAR PowerPac USB Host example applications

The example applications for the target-side are supplied in source code in the Application directory.
USB Host Core Functions

In this chapter, you will find a description of all API functions as well as all required data and function types.

Management Functions

The table below lists the available management functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_Init()</td>
<td>Basically initializes the USB Host stack.</td>
</tr>
<tr>
<td>USBH_Exit()</td>
<td>Is called on exit of the library.</td>
</tr>
<tr>
<td>USBH_EnumerateDevices()</td>
<td>Adds default endpoints for enumeration, sets the host controller into running state and starts the enumeration of the complete bus.</td>
</tr>
</tbody>
</table>

Table 4: IAR PowerPac USB Host management function overview

**USBH_Init()**

**Description**

Basically initializes the USB Host stack.

**Prototype**

`USBH_STATUS USBH_Init();`

**Additional information**

Has to be called one time during startup before any other function. The library initializes or allocates global resources within this function. The host controller must created and added to the bus driver at an later time.

**USBH_Exit()**

**Description**

Is called on exit of the library.

**Prototype**

`void USBH_Exit();`

**Additional information**

Has to be called on exit of the library. The library may free global resources within this function. This includes also the removing and deleting of added host controllers. After this function call, no other function of the library should be called.

**USBH_EnumerateDevices()**

**Description**

Adds default endpoints for enumeration, sets the host controller into running state and starts the enumeration of the complete bus.

**Prototype**

`void USBH_EnumerateDevices( USBH_HC_BD_HANDLE * HcBdHandle );`

**Additional information**

If this function returns the host controller runs and can detect USB devices.
API Functions

The table below lists the available API functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_CreateInterfaceList()</td>
<td>Generates a list of available interfaces.</td>
</tr>
<tr>
<td>USBH_DestroyInterfaceList()</td>
<td>Deletes a previously generated interface list.</td>
</tr>
<tr>
<td>USBH_GetInterfaceID()</td>
<td>Returns the interface ID for a specified interface.</td>
</tr>
<tr>
<td>USBH_GetInterfaceInfo()</td>
<td>Obtains information about a specified interface.</td>
</tr>
<tr>
<td>USBH_RegisterPnPNotification()</td>
<td>Registers a notification function for PnP events.</td>
</tr>
<tr>
<td>USBH_UnregisterPnPNotification()</td>
<td>Unregisters a previously registered notification for PnP events.</td>
</tr>
<tr>
<td>USBH_RegisterEnumErrorNotification()</td>
<td>Registers a port error enumeration notification.</td>
</tr>
<tr>
<td>USBH_UnregisterEnumErrorNotification()</td>
<td>Unregisters an registered port error enumeration notification.</td>
</tr>
<tr>
<td>USBH_RestartEnumError()</td>
<td>The enumeration for all devices that have failed the enumeration is restarted.</td>
</tr>
<tr>
<td>USBH_OpenInterface()</td>
<td>Opens the specified interface.</td>
</tr>
<tr>
<td>USBH_CloseInterface()</td>
<td>Closes a previously opened interface.</td>
</tr>
<tr>
<td>USBH_GetDeviceDescriptor()</td>
<td>Retrieves the device descriptor.</td>
</tr>
<tr>
<td>USBH_GetCurrentConfigurationDescriptor()</td>
<td>Retrieves the current configuration descriptor.</td>
</tr>
<tr>
<td>USBH_GetInterfaceDescriptor()</td>
<td>Retrieves the interface descriptor.</td>
</tr>
<tr>
<td>USBH_GetEndpointDescriptor()</td>
<td>Retrieves an endpoint descriptor.</td>
</tr>
<tr>
<td>USBH_GetSerialNumber()</td>
<td>Retrieves the serial number.</td>
</tr>
<tr>
<td>USBH_GetSpeed()</td>
<td>Retrieves the operation speed of the device.</td>
</tr>
<tr>
<td>USBH_GetFrameNumber()</td>
<td>Retrieves the current frame number.</td>
</tr>
<tr>
<td>USBH_GetInterfaceIDByHandle()</td>
<td>Retrieves the current frame number.</td>
</tr>
<tr>
<td>USBH_SubmitUrb()</td>
<td>Is used to submit an URB.</td>
</tr>
<tr>
<td>USBH_SetTraceMask()</td>
<td>Sets an internal trace mask which filters trace messages produced by the USB bus driver.</td>
</tr>
<tr>
<td>USBH_GetStatusStr()</td>
<td>Return the status as an string constants.</td>
</tr>
</tbody>
</table>

Table 5: IAR PowerPac USB Host API function overview

**USBH_CreateInterfaceList()**

**Description**

Generates a list of available interfaces.

**Prototype**

```c
USBH_INTERFACE_LIST_HANDLE USBH_CreateInterfaceList(
    USBH_INTERFACE_MASK * InterfaceMask,
    unsigned int * InterfaceCount );
```
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InterfaceMask</td>
<td>Input parameter of type USBH_INTERFACE_MASK which specifies a mask for the interfaces which should be listed.</td>
</tr>
<tr>
<td>InterfaceCount</td>
<td>Returns the number of available interfaces.</td>
</tr>
</tbody>
</table>

Table 6: USBH_CreateInterfaceList() parameter list

Additional information

The generated interface list is stored in the bus driver and must be deleted by a call to USBH_DestroyInterfaceList. The list contains a snapshot of interfaces available at the point of time where the function is called. This enables the application to have a fixed relation between the index and a USB interface in a list. The list is not updated if a device is removed or connected. A new list must be created to capture the current available interfaces.

USBH_DestroyInterfaceList()

Description

Deletes a previously generated interface list.

Prototype

```c
void USBH_DestroyInterfaceList(
    USBH_INTERFACE_LIST_HANDLE InterfaceListHandle );
```

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InterfaceListHandle</td>
<td>Contains the handle for the interface list. It must not be NULL.</td>
</tr>
</tbody>
</table>

USBH_DestroyInterfaceList() parameter list

Additional information

Deletes an interface list generated by a previous call to USBH_CreateInterfaceList. If an interface list is not deleted the library has a memory leak.

USBH_GetInterfaceID()

Description

Returns the interface ID for a specified interface.

Prototype

```c
USBH_INTERFACE_ID USBH_GetInterfaceID(
    USBH_INTERFACE_LIST_HANDLE InterfaceListHandle,
    unsigned int               Index );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InterfaceListHandle</td>
<td>Contains the handle for the interface list generated by a call to USBH_CreateInterfaceList.</td>
</tr>
<tr>
<td>Index</td>
<td>Specifies the zero based index for an interface in the list.</td>
</tr>
</tbody>
</table>

USBH_GetInterfaceID() parameter list

Return value

On success the interface ID for the interface specified by Index is returned. If the interface index does not exist the function returns 0.

Additional information

The interface ID identifies a USB interface as long as the device is connected to the host. If the device is removed and re-connected a new interface ID is assigned. The interface ID is even valid if the interface list is deleted. The function can return an interface ID even if the device is removed between the call to the function USBH_CreateInterfaceList and the call to this function. If this is the case, the function USBH_OpenInterface fails.
USBH_GetInterfaceInfo()

Description

Obtains information about a specified interface.

Prototype

```c
USBH_STATUS USBH_GetInterfaceInfo(
    USBH_INTERFACE_ID     InterfaceID,
    USBH_INTERFACE_INFO * InterfaceInfo );
```

Return value

Returns USBH_STATUS_SUCCESS or USBH_STATUS_DEVICE_REMOVED.

Additional information

Can be used to identify a USB interface without open it. More detailed information can be requests after the USB interface is opened.

USBH_RegisterPnPNotification()

Description

Registers a notification function for PnP events.

Prototype

```c
USBH_NOTIFICATION_HANDLE USBH_RegisterPnPNotification(
    USBH_PNP_NOTIFICATION * PnPNotification );
```

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PnPNotification</td>
<td>Contains a pointer to a caller provided structure USBH_PNP_NOTIFICATION. This structure must be filled in by the caller.</td>
</tr>
</tbody>
</table>

Return value

On success a valid handle is returned, or NULL on error.

Additional information

If a valid handle is returned, the function `USBH_UnregisterPnPNotification` must be called to release the notification. An application can register any number of notifications. The user notification routine is called in the context of an notify timer that is global for all USB bus PnP notifications. If this function is called while the bus driver has already enumerated devices that match the `USBH_INTERFACE_MASK` the function `USBH_PnpNotification` is called for each matching interface.

USBH_UnregisterPnPNotification()

Description

Unregisters a previously registered notification for PnP events.

Prototype

```c
void USBH_UnregisterPnPNotification( USBH_NOTIFICATION_HANDLE Handle );
```

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td>Contains the valid handle for a PnP notification previously registered by a call to <code>USBH_RegisterPnPNotification</code>.</td>
</tr>
</tbody>
</table>

Additional information

Has to be called for a PnP notification that was successfully registered by a call to `USBH_RegisterPnPNotification`. 
USBH_RegisterEnumErrorNotification()

Description
Registers a port error enumeration notification.

Prototype

```c
USBH_ENUM_ERROR_HANDLE USBH_RegisterEnumErrorNotification(
    void                       * Context,
    USBH_EnumErrorNotification * EnumErrorCallback );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>Is a user defined pointer that is passed unchanged to the notification callback function USBH_EnumErrorNotification.</td>
</tr>
<tr>
<td>EnumErrorCallback</td>
<td>Contains the notification function that is called from the library if a port enumeration error occurs.</td>
</tr>
</tbody>
</table>

Table 9: USBH_RegisterEnumErrorNotification() parameter list

Return value

On success a valid handle is returned, or NULL on error.

Additional information

If a valid handle is returned, the function USBH_RegisterEnumErrorNotification must be called to release the notification. The EnumErrorCallback callback routine is called in the context of the process where the interrupt status of a host controller is processed. It is forbidden to wait in that context.

USBH_UnregisterEnumErrorNotification()

Description
Unregisters an registered port error enumeration notification.

Prototype

```c
void USBH_UnregisterEnumErrorNotification( USBH_ENUM_ERROR_HANDLE Handle );
```

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td>Contains the valid handle for the notification previously returned from USBH_RegisterEnumErrorNotification.</td>
</tr>
</tbody>
</table>

Table 10: USBH_UnregisterEnumErrorNotification() parameter list

Additional information

Has to be called for a port enumeration error notification that was successfully registered by a call to USBH_RegisterEnumErrorNotification.

USBH_RestartEnumError()

Description
The enumeration for all devices that have failed the enumeration is restarted.

Prototype

```c
void USBH_RestartEnumError();
```

Additional information

The bus driver retries each enumeration again until the default retry count is reached.

USBH_OpenInterface()

Description
Opens the specified interface.
Prototype

```c
USBH_STATUS USBH_OpenInterface(
    USBH_INTERFACE_ID InterfaceID,
    U8 Exclusive,
    USBH_INTERFACE_HANDLE * InterfaceHandle );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InterfaceID</td>
<td>Specifies the interface to open by its interface ID. The interface ID can be obtained by USBH_PnpNotification or USBH_GetInterfaceID.</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Specifies if the interface should be opened exclusive or not. If the value is unequal of zero the interface is opened exclusive.</td>
</tr>
<tr>
<td>InterfaceHandle</td>
<td>Returns the handle for the opened interface on success.</td>
</tr>
</tbody>
</table>

Table 11: USBH_OpenInterface() parameter list

Return value

Returns success or an error. The function can fail if the device is or was removed or the device is opened exclusive by a different application. The function returns with error if the exclusive flag is true and a different application has an open handle to the function.

Additional information

The handle returned by this function is used by all other function that perform a data transfer. The returned handle must be closed with USBH_CloseInterface if it is no longer required.

**USBH_CloseInterface()**

Description

Closes the specified interface.

Prototype

```c
void USBH_CloseInterface( USBH_INTERFACE_HANDLE Handle );
```

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td>Contains the handle for an interface opened by a call to USBH_OpenInterface. It must not be NULL.</td>
</tr>
</tbody>
</table>

Table 12: USBH_CloseInterface() parameter list

Additional information

Each handle must be closed one time. The library access invalid memory if this function is called with an invalid handle.

**USBH_GetDeviceDescriptor()**

Description

Retrieves the device descriptor.

Prototype

```c
USBH_STATUS USBH_GetDeviceDescriptor(
    USBH_INTERFACE_HANDLE Handle,
    U8 * Descriptor,
    unsigned int Size,
    unsigned int * Count );
```
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td>Specifies the interface by its interface handle.</td>
</tr>
<tr>
<td>Descriptor</td>
<td>Points to a caller provided buffer that retrieves the device descriptor on success.</td>
</tr>
<tr>
<td>Size</td>
<td>Specifies the size of the caller provided buffer.</td>
</tr>
<tr>
<td>Count</td>
<td>Returns the length of the returned descriptor.</td>
</tr>
</tbody>
</table>

Return value

Success or device removed.

Additional information

Returns a copy of the device descriptor and does not access the device. If the buffer is smaller than the device descriptor the function returns the first part of it.

**USBH_GetCurrentConfigurationDescriptor()**

Description

Retrieves the current configuration descriptor.

Prototype

```c
USBH_STATUS USBH_GetCurrentConfigurationDescriptor(
    USBH_INTERFACE_HANDLE   Handle,
    U8                    * Descriptor,
    unsigned int            Size,
    unsigned int          * Count );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td>Specifies the interface by its interface handle.</td>
</tr>
<tr>
<td>Descriptor</td>
<td>Points to a caller provided buffer that retrieves the current configuration descriptor on success.</td>
</tr>
<tr>
<td>Size</td>
<td>Specifies the size of the caller provided buffer.</td>
</tr>
<tr>
<td>Count</td>
<td>Returns the number of valid bytes.</td>
</tr>
</tbody>
</table>

Return value

Success or device removed.

Additional information

Returns a copy of the current configuration descriptor. The descriptor is a copy that was stored during the device enumeration. The function returns the first part of the descriptor if the buffer is smaller than the descriptor. This descriptor contains all interface, endpoint, and possible class descriptors. The size is variable. The current configuration descriptor is the descriptor return to the request with the index 0 if the device was enumerated by the device the first time. It changes if the configuration is switch with `USBH_SET_CONFIGURATION`. Other configuration descriptors of a multi-configuration device can be requested with `USBH_FUNCTION_CONTROL_REQUEST`.

**USBH_GetInterfaceDescriptor()**

Description

Retrieves the interface descriptor.

Prototype

```c
USBH_STATUS USBH_GetInterfaceDescriptor(
    USBH_INTERFACE_HANDLE   Handle,
    unsigned int           AlternateSetting,
    U8                    * Descriptor,
    unsigned int            Size,
    unsigned int          * Count );
```
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td>Specifies the interface by its interface handle.</td>
</tr>
<tr>
<td>AlternateSetting</td>
<td>Specifies the alternate setting for this interface.</td>
</tr>
<tr>
<td>Descriptor</td>
<td>Points to a caller provided buffer that retrieves the interface descriptor on success.</td>
</tr>
<tr>
<td>Size</td>
<td>Specifies the size of the caller provided buffer.</td>
</tr>
<tr>
<td>Count</td>
<td>Returns the number of valid bytes in the descriptor.</td>
</tr>
</tbody>
</table>

Table 15: USBH_GetInterfaceDescriptor() parameter list

Return value

Success, device removed, or invalid parameter.

Additional information

returns a copy of an interface descriptor. The interface descriptor belongs to the interface that is identified by the USBH_INTERFACE_HANDLE. If the interface has different alternate settings the interface descriptors of each alternate setting can be requested. The function returns a copy of this descriptor that was requested during the enumeration. The interface descriptor is a part of the configuration descriptor.

**USBH_GetEndpointDescriptor()**

Description

Retrieves an endpoint descriptor.

Prototype

```c
USBH_STATUS USBH_GetEndpointDescriptor(
    USBH_INTERFACE_HANDLE   Handle,
    U8                      AlternateSetting,
    USBH_EP_MASK           * Mask,
    U8                    * Descriptor,
    unsigned int            Size,
    unsigned int          * Count );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td>Specifies the interface by its interface handle.</td>
</tr>
<tr>
<td>AlternateSetting</td>
<td>Specifies the alternate setting for the interface. The function returns endpoint descriptors that are inside the specified alternate setting.</td>
</tr>
<tr>
<td>Mask</td>
<td>Is of type USBH_EP_MASK and specifies a mask to select the endpoint.</td>
</tr>
<tr>
<td>Descriptor</td>
<td>Returns a pointer to a caller provided buffer that contains the endpoint descriptor on success.</td>
</tr>
<tr>
<td>Size</td>
<td>Specifies the size of the caller provided buffer.</td>
</tr>
<tr>
<td>Count</td>
<td>Returns the valid number of bytes written to the buffer.</td>
</tr>
</tbody>
</table>

Table 16: USBH_GetEndpointDescriptor() parameter list

Return value

Fails if the endpoint cannot be found or if the device is removed.

Additional information

Returns a copy of the endpoint descriptor that was captured during the enumeration. The endpoint descriptor is a part of the configuration descriptor.

**USBH_GetSerialNumber()**

Description

Retrieves the serial number.
Prototype

```c
USBH_STATUS USBH_GetSerialNumber(
    USBH_INTERFACE_HANDLE Handle,
    U8 * Descriptor,
    unsigned int Size,
    unsigned int * Count );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td>Specifies the interface by its interface handle.</td>
</tr>
<tr>
<td>Descriptor</td>
<td>Is a pointer to a caller provided buffer. It returns the serial number on success.</td>
</tr>
<tr>
<td>Size</td>
<td>Specifies the size of the caller provided buffer in bytes.</td>
</tr>
<tr>
<td>Count</td>
<td>Returns the number of bytes written to the buffer.</td>
</tr>
</tbody>
</table>

Table 17: USBH_GetSerialNumber() parameter list

Return value

Returns an error if the device is removed.

Additional information

Returns the serial number as a UNICODE string in USB little endian format. Count returns the number of valid bytes. The string is not zero terminated. The returned data does not contain a USB descriptor header. The descriptor is requested with the first language ID. This string is a copy of the serial number string that was requested during the enumeration. To request other string descriptors use `USBH_SubmitUrb`. If the device does not support a USB serial number string the function returns success and a length of 0.

**USBH_GetSpeed()**

Description

Retrieves the operation speed of the device.

Prototype

```c
USBH_STATUS USBH_GetSpeed(
    USBH_INTERFACE_HANDLE Handle,
    USBH_SPEED * Speed );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td>Specifies the interface by its interface handle.</td>
</tr>
<tr>
<td>Speed</td>
<td>Returns the operating speed of the device. It is of type USBH_SPEED.</td>
</tr>
</tbody>
</table>

Table 18: USBH_GetSpeed() parameter list

Return value

Returns an error if the device is removed.

Additional information

A high speed device can operate in full or high speed mode.

**USBH_GetFrameNumber()**

Description

Retrieves the current frame number.

Prototype

```c
USBH_STATUS USBH_GetFrameNumber(
    USBH_INTERFACE_HANDLE Handle,
    U32 * FrameNumber );
```
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td>Specifies the interface by its interface handle.</td>
</tr>
<tr>
<td>FrameNumber</td>
<td>Returns the current frame number on success.</td>
</tr>
</tbody>
</table>

Table 19: USBH_GetFrameNumber() parameter list

Return value

Returns an error if the device is removed.

Additional information

The frame number is transferred on the bus with 11 bits. This frame number is returned as a 16 or 32 bit number related to the implementation of the host controller. The last 11 bits are equal to the current frame. The frame number is increased each ms. This is the case for high speed, too. The returned frame number is related to the bus where the device is connected. The frame numbers between different host controllers can be different.

USBH_GetInterfaceIDByHandle()

Description

Retrieves the interface ID for a given interface.

Prototype

USBH_STATUS USBH_GetInterfaceIDByHandle(
    USBH_INTERFACE_HANDLE   Handle,
    USBH_INTERFACE_ID     * InterfaceID );

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td>Specifies the interface by its interface handle.</td>
</tr>
<tr>
<td>InterfaceID</td>
<td>Returns the interface ID on success.</td>
</tr>
</tbody>
</table>

Table 20: USBH_GetInterfaceIDByHandle() parameter list

Return value

Returns an error if the device is removed.

Additional information

Returns the interface ID if the handle to the interface is available. This may be useful if a Plug and Play notification is received and the application checks if it is related to a given handle. The application can avoid calls to this function if the interface ID is stored in the device context of the application.

USBH_SubmitUrb()

Description

Is used to submit an URB.

Prototype

USBH_STATUS USBH_SubmitUrb(
    USBH_INTERFACE_HANDLE   Handle,
    URB                   * Urb );
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td>Specifies the interface by its interface handle.</td>
</tr>
<tr>
<td>Urb</td>
<td>Input and output parameter. On input it contains the URB which should be submitted. On output it contains the submitted URB with the appropriate status and the received data if any. The storage for the URB must be permanent as long as the request is pending. The host controller can define special alignment requirements for the URB or the data transfer buffer.</td>
</tr>
</tbody>
</table>

Table 21: USBH_SubmitUrb() parameter list

Return value

The request can fail on different reasons. If the function returns USBH_STATUS_PENDING the completion function is called later. In all other cases the completion routine is not called. If the function returns success, the request was processed immediately. On error the request cannot be processed.

Additional information

If the status USBH_STATUS_PENDING is returned the ownership of the URB is passed to the bus driver. The storage of the URB must not be freed nor modified as long as the ownership is at the bus driver. The bus driver passes the URB back to the application by calling the completion routine. An URB that transfers data can be pending for a long time.

USBH_SetTraceMask()

Description

Sets an internal trace mask which filters trace messages produced by the USB bus driver

Prototype

void USBH_SetTraceMask( U32 Mask );

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask</td>
<td>Specifies the new trace mask to be set.</td>
</tr>
</tbody>
</table>

Table 22: USBH_SetTraceMask() parameter list

Additional information

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

Bit positions of trace mask are assigned as described below:

- **DBG_ERR**: Fatal errors and ASSERTs. It is recommended to always set this bit.
- **DBG_WRN**: Non-fatal errors. Warning messages. It is recommended to always set this bit.
- **DBG_INFO**: Informational messages.
- **DBG_FUNC**: Function names.
- **DBG_UPPER**: Print functions names with parameters of the upper interface.
- **DBG_EP**: Endpoint object and interface traces.
- **DBG_EP0**: Control endpoint object traces.
- **DBG_HOST**: Host object traces.
DBG_RHUB
RootHub object traces.

DBG_DRV
Driver object traces.

DBG_PNP
PNP notification traces.

DBG_DEV
Device object traces.

DBG_URBCT
Traces an URB counter for testing.

DBG_HUB
Hub object traces.

DBG_REFCT
Trace an internal reference counter.

DBG_SUBSTATE
Trace an helper sub state machine.

DBG_HUBNOTIFY
Trace hub device status notifications.

DBG_ADDREMOVE
Display informations about adding and removing of USB devices.

By default, the bits DBG_ERR and DBG_WRN are set and all other bits are cleared in the trace mask. Note that the DBG_xxx constants specify a bit position and not the corresponding mask. Use the DBG_BIT_MASK macro to create the corresponding mask for an individual bit position.

Example:
USBH_SetTraceMask(DBG_BIT_MASK(DBG_ERR)|DBG_BIT_MASK(DBG_WRN));

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

**USBH_GetStatusStr()**

**Description**

Return the status as an string constants.

**Prototype**

```c
const char * USBH_GetStatusStr( USBH_STATUS x );
```

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Specifies the status.</td>
</tr>
</tbody>
</table>

*Table 23: USBH_GetStatusStr() parameter list*

**Return value**

An error string is returned.

**Additional information**

Returns only an error string if the debug version of the library is used (DBG=1).
Data Structures

The table below lists the available data structures.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_INTERFACE_MASK</td>
<td>Input parameter to create an interface list or to register a PnP notification.</td>
</tr>
<tr>
<td>USBH_INTERFACE_INFO</td>
<td>Contains information about a USB interface and the related device.</td>
</tr>
<tr>
<td>USBH_ENUM_ERROR</td>
<td>Is used as an notification parameter for the USBH_EnumErrorNotification function.</td>
</tr>
<tr>
<td>USBH_EP_MASK</td>
<td>Input parameter to get an endpoint descriptor.</td>
</tr>
<tr>
<td>USBH_CONTROL_REQUEST</td>
<td>Is used as a union member for the URB data structure.</td>
</tr>
<tr>
<td>USBH_BULK_INT_REQUEST</td>
<td>Is used to transfer data from or to a bulk endpoint.</td>
</tr>
<tr>
<td>USBH_ISO_FRAME</td>
<td>Is used to define ISO transfer buffers.</td>
</tr>
<tr>
<td>USBH_ISO_REQUEST</td>
<td>Is used to transfer data to an ISO endpoint.</td>
</tr>
<tr>
<td>USBH_ENDPOINT_REQUEST</td>
<td>Is used as a union member for the URB data structure.</td>
</tr>
<tr>
<td>USBH_SET_CONFIGURATION</td>
<td>Is used as a union member for the URB data structure.</td>
</tr>
<tr>
<td>USBH_SET_INTERFACE</td>
<td>Is used as a union member for the URB data structure.</td>
</tr>
<tr>
<td>USBH_SET_POWER_STATE</td>
<td>Is used to set a power state.</td>
</tr>
<tr>
<td>URB</td>
<td>Basic structure for all asynchronous operations on the bus driver.</td>
</tr>
<tr>
<td>USBH_PNP_NOTIFICATION</td>
<td>Is used as an input parameter for the USBH_RegisterPnPNotification function.</td>
</tr>
<tr>
<td>USBH_HEADER</td>
<td>Defines the header of an URB.</td>
</tr>
<tr>
<td>USBH_SPEED</td>
<td>Is used to get the operation speed of a device.</td>
</tr>
<tr>
<td>USBH_PNP_EVENT</td>
<td>Is used as a parameter for the PnP notification.</td>
</tr>
<tr>
<td>USBH_FUNCTION</td>
<td>Is used as a member for the USBH_HEADER data structure.</td>
</tr>
<tr>
<td>USBH_POWER_STATE</td>
<td>Specifies some power states.</td>
</tr>
</tbody>
</table>

Table 24: IAR PowerPac USB Host data structure overview

USBH_INTERFACE_MASK

Definition

typedef struct tag_USBH_INTERFACE_MASK {
    U16 Mask;
    U16 VID;
    U16 PID;
    U16 bcdDevice;
    U8  Interface;
    U8  Class;
    U8  SubClass;
    U8  Protocol;
} USBH_INTERFACE_MASK;

Description

Input parameter to create an interface list or to register a PnP notification.
USBH_INTERFACE_INFO

**Definition**

typedef struct tag_USBH_INTERFACE_INFO {
    USBH_INTERFACE_ID InterfaceID;
    USB_DEVICE_ID     DeviceID;
    U16               VID;
    U16               PID;
    U16               bcdDevice;
    U8                Interface;
    U8                Class;
    U8                SubClass;
    U8                Protocol;
    unsigned int     OpenCount;
    U8                ExclusiveUsed;
    USB_SPEED         Speed;
    U8                SerialNumber[256];
    U8                SerialNumberSize;
} USBH_INTERFACE_INFO;

**Description**

Is used to get information about a device with the function USBH_GetInterfaceInfo.
## Members

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InterfaceID</td>
<td>Contains the unique interface ID. This ID is assigned if the USB device was successful enumerated. It is valid until the device is removed for the host. If the device is reconnected a different interface ID is assigned to each interface.</td>
</tr>
<tr>
<td>DeviceID</td>
<td>Contains the unique device ID. This ID is assigned if the USB device was successful enumerated. It is valid until the device is removed for the host. If the device is reconnected a different device ID is assigned. The relation between the device ID and the interface ID can be used by an application to detect which USB interfaces belong to a device.</td>
</tr>
<tr>
<td>VID</td>
<td>Contains the vendor ID.</td>
</tr>
<tr>
<td>PID</td>
<td>Contains the product ID.</td>
</tr>
<tr>
<td>bcdDevice</td>
<td>Contains the BCD coded device version.</td>
</tr>
<tr>
<td>Interface</td>
<td>Contains the USB interface number.</td>
</tr>
<tr>
<td>Class</td>
<td>Specifies the interface class.</td>
</tr>
<tr>
<td>Subclass</td>
<td>Specifies the interface sub class.</td>
</tr>
<tr>
<td>Protocol</td>
<td>Specifies the interface protocol.</td>
</tr>
<tr>
<td>OpenCount</td>
<td>Specifies the number of open handles for this interface.</td>
</tr>
<tr>
<td>ExclusiveUsed</td>
<td>Determines if this interface is used exclusive.</td>
</tr>
<tr>
<td>Speed</td>
<td>Specifies the operation speed of this interface.</td>
</tr>
<tr>
<td>SerialNumber[256]</td>
<td>Contains the serial number as a counted UNICODE string.</td>
</tr>
<tr>
<td>SerialNumberSize</td>
<td>Contains the length of the serial number in bytes.</td>
</tr>
</tbody>
</table>

*Table 26: USBH_INTERFACE_INFO() member list*

## USBH_ENUM_ERROR

### Definition

```c
typedef struct tag_USBH_ENUM_ERROR {
    int         Flags;
    int         PortNumber;
    USBH_STATUS Status;
    int         ExtendedErrorInformation;
} USBH_ENUM_ERROR;
```

### Description

Is used as an notification parameter for the `USBH_EnumErrorNotification` function. This data structure does not contain detailed information about the device that fails the enumeration because this information is not available in all phases of the enumeration.
Members

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flags</td>
<td>Additional flags to determine the location and the type of the error.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_ENUM_ERROR_EXTHUBPORT_FLAG</td>
<td>means the device is connected to an external hub.</td>
</tr>
<tr>
<td>USBH_ENUM_ERROR_RETRY_FLAG</td>
<td>the bus driver retries the enumeration of this device automatically.</td>
</tr>
<tr>
<td>USBH_ENUM_ERROR_STOP_ENUM_FLAG</td>
<td>the bus driver does not restart the enumeration for this device because all</td>
</tr>
<tr>
<td></td>
<td>retries has failed. The application can force the bus driver to restart the</td>
</tr>
<tr>
<td></td>
<td>enumeration by calling the function USBH_RestartEnumError.</td>
</tr>
<tr>
<td>USBH_ENUM_ERROR_DISCONNECT_FLAG</td>
<td>means the device has been disconnected during the enumeration. If the hub</td>
</tr>
<tr>
<td></td>
<td>port reports a disconnect state the device cannot be re-enumerated by the</td>
</tr>
<tr>
<td></td>
<td>bus driver automatically. Also the function USBH_RestartEnumError cannot</td>
</tr>
<tr>
<td></td>
<td>re-enumerate the device.</td>
</tr>
<tr>
<td>USBH_ENUM_ERROR_ROOT_PORT_RESET</td>
<td>means an error during the USB reset of a root hub port occurs.</td>
</tr>
<tr>
<td>USBH_ENUM_ERROR_HUB_PORT_RESET</td>
<td>means an error during a reset of an external hub port occurs.</td>
</tr>
<tr>
<td>USBH_ENUM_ERROR_INIT_DEVICE</td>
<td>means an error during the device initialization (e.g. no answer to a descriptor</td>
</tr>
<tr>
<td></td>
<td>request or it failed other standard requests.</td>
</tr>
<tr>
<td>USBH_ENUM_ERROR_INIT_HUB</td>
<td>means the enumeration of an external hub fails.</td>
</tr>
</tbody>
</table>

| PortNumber | Port number of the parent port where the USB device is connected. A flag in  |
|            | the PortFlags field determine if this is an external hub port.               |

<table>
<thead>
<tr>
<th>Status</th>
<th>Status of the failed operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExtendedErrorInformation</td>
<td>Internal information used for debugging.</td>
</tr>
</tbody>
</table>

Table 27: USBH_ENUM_ERROR() member list

**USBH_EP_MASK**

**Definition**

typedef struct tag_USBH_EP_MASK {
    U32 Mask;
    U8  Index;
    U8  Address;
    U8  Type;
    U8  Direction;
} USBH_EP_MASK;

**Description**

Is used as an input parameter to get an endpoint descriptor. The comparison with the mask is true if each member that is marked as valid by a flag in the mask member is equal to the value stored in the endpoint. E.g. if the mask is 0 the first endpoint is returned. If the Mask is set to USBH_EP_MASK_INDEX the zero based index can be used to address all endpoints.
**USBH_CONTROL_REQUEST**

**Definition**

```c
typedef struct tag_USBH_CONTROL_REQUEST {
    SETUP_PACKET  Setup;
    U8             Endpoint;
    void         *  Buffer;
    U32            Length;
} USBH_CONTROL_REQUEST;
```

**Description**

Is used to submit a control request. A control request consists of a setup phase, an optional data phase, and a handshake phase. The data phase is limited to a length of 4096 bytes. The Setup data structure must be filled in properly. The length field in the Setup must contain the size of the Buffer. The caller must provide the storage for the Buffer.

With this request each setup packet can be submitted. Some standard requests, like SetAddress, can be send but would destroy the multiplexing of the bus driver. It is not allowed to set the following standard requests:

- **SetAddress**
  - It is assigned by the bus driver during enumeration or USB reset.

  ```c
  Clear Feature Endpoint Halt
  Use USBH_FUNCTION_RESET_ENDPOINT instead. The function USBH_FUNCTION_RESET_ENDPOINT resets the data toggle bit in the host controller structures.
  ```

- **SetConfiguration**
  - Use USBH_SET_CONFIGURATION instead. The bus driver must take care on the interfaces and endpoints of a configuration. The function USBH_SET_CONFIGURATION updates the internal structures of the driver.
**USBH_BULK_INT_REQUEST**

**Definition**

```c
typedef struct tag_USBH_BULK_INT_REQUEST {
    U8    Endpoint;
    void * Buffer;
    U32   Length;
} USBH_BULK_INT_REQUEST;
```

**Description**

The buffer size can be larger than the FIFO size but a host controller implementation can define a maximum size for a buffer that can be handled with one URB. To get a good performance the application should use two or more buffers.

**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Endpoint</td>
<td>Specifies the endpoint address with direction bit.</td>
</tr>
<tr>
<td>Buffer</td>
<td>Pointer to a caller provided buffer.</td>
</tr>
<tr>
<td>Length</td>
<td>Contains the size of the buffer and returns the number of bytes transferred.</td>
</tr>
</tbody>
</table>

**USBH_ISO_FRAME**

**Definition**

```c
typedef struct tag_USBH_ISO_FRAME {
    U32         Offset;
    U32         Length;
    USBH_STATUS Status;
} USBH_ISO_FRAME;
```

**Description**

Is part of **USBH_ISO_REQUEST**. It describes the amount of data that is transferred in one frame.

**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset</td>
<td>Specifies the offset in bytes relative to the beginning of the transfer buffer.</td>
</tr>
<tr>
<td>Length</td>
<td>Contains the length that should be transferred in one frame.</td>
</tr>
<tr>
<td>Status</td>
<td>Contains the status of the operation in this frame. For an OUT endpoint this status is always success. For an IN point a CRC or Data Toggle error can be reported.</td>
</tr>
</tbody>
</table>

**USBH_ISO_REQUEST**

**Definition**

```c
typedef struct tag_USBH_ISO_REQUEST {
    U8             Endpoint;
    void         * Buffer;
    U32            Length;
} USBH_ISO_REQUEST;
```
unsigned int Flags;
unsigned int StartFrame;
unsigned int Frames;
}

**USBH_ISO_REQUEST**

**Description**

Is incomplete defined. That means the data structure consists of this data structure and an array of data structures `USBH_ISO_FRAME`. The size of the array is defined by Frames. Use the macro `USBH_GET_ISO_URB_SIZE` to get the size for an ISO URB.

**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Endpoint</td>
<td>Specifies the endpoint address with direction bit.</td>
</tr>
<tr>
<td>Buffer</td>
<td>Is a pointer to a caller provided buffer.</td>
</tr>
<tr>
<td>Length</td>
<td>On input this member specifies the size of the user provided buffer. On output it contains the number of bytes transferred.</td>
</tr>
<tr>
<td>Flags</td>
<td>This parameter contains 0 or the following flag:</td>
</tr>
<tr>
<td></td>
<td>USBH_ISO_ASAP If this flag is set the transfer starts as soon as possible and the parameter StartFrame is ignored.</td>
</tr>
<tr>
<td></td>
<td>StartFrame If the flag USBH_ISO_ASAP is not set this parameter StartFrame defines the start frame of the transfer. The StartFrame must be in the future. Use <code>USBH_GetFrameNumber</code> to get the current frame number. Add a time to the current frame number.</td>
</tr>
<tr>
<td>Frames</td>
<td>Contains the number of frames that are described with this structure.</td>
</tr>
</tbody>
</table>

| Table 32: USBH_ISO_REQUEST() member list |

**USBH_ENDPOINT_REQUEST**

**Definition**

```c
typedef struct tag_USBH_ENDPOINT_REQUEST {
    U8 Endpoint;
} USBH_ENDPOINT_REQUEST;
```

**Description**

Is used with the requests `USBH_FUNCTION_RESET_ENDPOINT` and `USBH_FUNCTION_ABORT_ENDPOINT`.

**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Endpoint</td>
<td>Specifies the endpoint address.</td>
</tr>
</tbody>
</table>

| Table 33: USBH_ENDPOINT_REQUEST() member list |

**USBH_SET_CONFIGURATION**

**Definition**

```c
typedef struct tag_USBH_SET_CONFIGURATION {
    U8 ConfigurationDescriptorIndex;
} USBH_SET_CONFIGURATION;
```

**Description**

Is used with the request `USBH_FUNCTION_SET_CONFIGURATION`.
Members

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfigurationDescriptorIndex</td>
<td>Specifies the index in the configuration description.</td>
</tr>
</tbody>
</table>

Table 34: USBH_SET_CONFIGURATION() member list

**USBH_SET_INTERFACE**

**Definition**

typedef struct tag_USBH_SET_INTERFACE {
    U8 AlternateSetting;
} USBH_SET_INTERFACE;

**Description**

is used with the request USBH_FUNCTION_SET_INTERFACE.

**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlternateSetting</td>
<td>Specifies the alternate setting.</td>
</tr>
</tbody>
</table>

Table 35: USBH_SET_INTERFACE() member list

**USBH_SET_POWER_STATE**

**Definition**

typedef struct tag_USBH_SET_POWER_STATE {
    USBH_POWER_STATE PowerState;
} USBH_SET_POWER_STATE;

**Description**

If the device is switched to suspend, there must be no pending requests on the device.

**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerState</td>
<td>Specifies the power state</td>
</tr>
</tbody>
</table>

Table 36: USBH_SET_POWER_STATE() member list

**URB**

**Definition**

typedef struct tag_URB {
    USBH_HEADER Header;
    union Request;
} URB;

**Description**

The following table lists the possible information types and associated structures:

<table>
<thead>
<tr>
<th>Request Type</th>
<th>Associated Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlRequest</td>
<td>USBH_CONTROL_REQUEST</td>
</tr>
<tr>
<td>BulkIntRequest</td>
<td>USBH_BULK_INT_REQUEST</td>
</tr>
<tr>
<td>IsoRequest</td>
<td>USBH_ISO_REQUEST</td>
</tr>
<tr>
<td>EndpointRequest</td>
<td>USBH_ENDPOINT_REQUEST</td>
</tr>
<tr>
<td>SetConfiguration</td>
<td>USBH_SET_CONFIGURATION</td>
</tr>
<tr>
<td>SetInterface</td>
<td>USBH_SET_INTERFACE</td>
</tr>
<tr>
<td>SetPowerState</td>
<td>USBH_SET_POWER_STATE</td>
</tr>
</tbody>
</table>

The URB is the basic structure for all asynchronous operations on the bus driver. All requests that exchanges data with the device are using this data structure. The caller has to provide the memory for this structure. The memory must be permanent until the completion function is called. This data structure is used to submit an URB.
**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>Contains the URB header of type USBH_HEADER. The most important parameters are the function code and the callback function.</td>
</tr>
<tr>
<td>Request</td>
<td>Is a union and contains information depending on the specific request of the USBH_HEADER.</td>
</tr>
</tbody>
</table>

**USBH_PNP_NOTIFICATION**

**Definition**

typedef struct tag_USBH_PNP_NOTIFICATION {
    USBH_PnpNotification * PnpNotification;
    void                 * Context;
    USBH_INTERFACE_MASK    InterfaceMask;
} USBH_PNP_NOTIFICATION;

**Description**

Is used as an input parameter for the USBH_RegisterPnPNotification function.

**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PnpNotification</td>
<td>Contains the notification function that is called from the library if a PnP event occurs.</td>
</tr>
<tr>
<td>Context</td>
<td>Contains the notification context that is passed unchanged to the notification function.</td>
</tr>
<tr>
<td>PowerState</td>
<td>Contains a mask for the interfaces for which the PnP notification should be called.</td>
</tr>
</tbody>
</table>

**USBH_HEADER**

**Definition**

typedef struct tag_USBH_HEADER {
    USBH_FUNCTION             Function;
    USBH_STATUS               Status;
    USBH_ON_COMPLETION_FUNC * Completion;
    void                    * Context;
    DLIST                     ListEntry;
} USBH_HEADER;

**Description**

All not described members of this structure are for internal use only. Do not use these members. A caller must fill in the members Function, Completion, and if required Context.

**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Describes the function of the request.</td>
</tr>
<tr>
<td>Status</td>
<td>After completion this member contains the status for the request.</td>
</tr>
<tr>
<td>Completion</td>
<td>Caller provided pointer to the completion function. This completion function is called if the function USBH_SubmitUrb returns USBH_STATUS_PENDING. If a different status code is returned the completion function is never called.</td>
</tr>
<tr>
<td>Context</td>
<td>Can be used by the caller to store a context for the completion routine. It is not changed by the library.</td>
</tr>
<tr>
<td>ListEntry</td>
<td>Can be used to keep the URB in a list. The owner of the URB can use this list entry. If the URB is passed to the library this member is used by the library.</td>
</tr>
</tbody>
</table>

---

**Table 37: URB() member list**

**Table 38: USBH_PNP_NOTIFICATION() member list**

**Table 39: USBH_HEADER() member list**
**USBH_SPEED**

**Definition**

typedef enum tag_USBH_SPEED {
    USBH_SPEED_UNKNOWN,
    USBH_LOW_SPEED,
    USBH_FULL_SPEED,
    USBH_HIGH_SPEED
} USBH_SPEED;

**Description**

Is used as a member in the `USBH_INTERFACE_INFO` data structure and to get the operation speed of a device.

**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_SPEED_UNKNOWN</td>
<td>The speed is unknown.</td>
</tr>
<tr>
<td>USBH_LOW_SPEED</td>
<td>The device operates at low speed.</td>
</tr>
<tr>
<td>USBH_FULL_SPEED</td>
<td>The device operates at full speed.</td>
</tr>
<tr>
<td>USBH_HIGH_SPEED</td>
<td>The device operates at high speed.</td>
</tr>
</tbody>
</table>

*Table 40: USBH_SPEED() member list*

**USBH_PNP_EVENT**

**Definition**

typedef enum tag_USBH_PNP_EVENT {
    USBH_AddDevice,
    USBH_RemoveDevice
} USBH_PNP_EVENT;

**Description**

Is used as a parameter for the PnP notification.

**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_AddDevice</td>
<td>Indicates that a device was connected to the host and new interface is available.</td>
</tr>
<tr>
<td>USBH_RemoveDevice</td>
<td>Indicates that a device has been removed.</td>
</tr>
</tbody>
</table>

*Table 41: USBH_PNP_EVENT() member list*

**USBH_FUNCTION**

**Definition**

typedef enum tag_USBH_FUNCTION {
    USBH_FUNCTION_CONTROL_REQUEST, 
    USBH_FUNCTION_BULK_REQUEST, 
    USBH_FUNCTION_INT_REQUEST, 
    USBH_FUNCTION_ISO_REQUEST, 
    USBH_FUNCTION_RESET_DEVICE, 
    USBH_FUNCTION_RESET_ENDPOINT, 
    USBH_FUNCTION_ABORT_ENDPOINT, 
    USBH_FUNCTION_SET_CONFIGURATION, 
    USBH_FUNCTION_SET_INTERFACE, 
    USBH_FUNCTION_SET_POWER_STATE
} USBH_FUNCTION;

**Description**

Is used as a member for the `USBH_HEADER` data structure. All function codes use the API function `USBH_SubmitUrb` and are handled asynchronously.
## Entries

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_FUNCTION_CONTROL_REQUEST</td>
<td>Is used to send an URB with a control request. It uses the data structure USBH_CONTROL_REQUEST. A control request includes standard, class and vendor defines requests. The standard requests SetConfiguration, SetAddress and SetInterface cannot be submitted by this request. These requests require a special handling in the driver. See USBH_FUNCTION_SET_CONFIGURATION and USBH_FUNCTION_SET_INTERFACE for details.</td>
</tr>
<tr>
<td>USBH_FUNCTION_BULK_REQUEST</td>
<td>Is used to transfer data to or from a bulk endpoint. It uses the data structure USBH_BULK_INT_REQUEST.</td>
</tr>
<tr>
<td>USBH_FUNCTION_INT_REQUEST</td>
<td>Is used to transfer data to or from an interrupt endpoint. It uses the data structure USBH_BULK_INT_REQUEST. The interval is defined by the endpoint descriptor.</td>
</tr>
<tr>
<td>USBH_FUNCTION_ISO_REQUEST</td>
<td>Is used to transfer data to or from an ISO endpoint. It uses the data structure USBH_ISO_FRAME. ISO transfer may not be supported by all host controllers.</td>
</tr>
<tr>
<td>USBH_FUNCTION_RESET_DEVICE</td>
<td>Sends an USB reset to the device. This causes a remove event for all interfaces of the device. After the device is successfully enumerated an arrival event is indicated. All interfaces get new interface ID's. This request uses only the URB header. If the driver indicates an arrival event the device is in a defined state because it is reseted and enumerated by the bus driver. This request can be part of an error recovery or part of special class protocols like DFU. The application should abort all pending requests and close all handles to this device. All handles become invalid.</td>
</tr>
<tr>
<td>USBH_FUNCTION_RESET_ENDPOINT</td>
<td>Clears an error condition on a special endpoint. If a data transfer error occurs that cannot be handled in hardware the bus driver stops the endpoint and does not allow further data transfers before the endpoint is reseted with this function. On a bulk or interrupt endpoint the host driver sends a Clear Feature Endpoint Halt request. This informs the device about the hardware error. The driver resets the data toggle bit for this endpoint. This request expects that no pending URBs are scheduled on this endpoint. Pending URBs must be aborted with the URB based function USBH_FUNCTION_ABORT_ENDPOINT. This function uses the data structure USBH_ENDPOINT_REQUEST.</td>
</tr>
<tr>
<td>USBH_FUNCTION_ABORT_ENDPOINT</td>
<td>Aborts all pending requests on a endpoint. The host controller calls the completion function with a status code USBH_STATUS_CANCELED. The completion of the URBs may be delayed. The application should wait until all pending requests has been returned by the driver before the handle is closed or USBH_FUNCTION_RESET_ENDPOINT is called.</td>
</tr>
<tr>
<td>USBH_FUNCTION_SET_CONFIGURATION</td>
<td>The driver selects the configuration defined by the configuration descriptor with the index 0 during the enumeration. If the application uses this configuration there is no need to call this function. If the application wants to activate a different configuration this function must be called.</td>
</tr>
</tbody>
</table>

Table 42: USBH_FUNCTION() member list
typedef enum tag_USBH_POWER_STATE {
    USBH_NORMAL_POWER,
    USBH_SUSPEND
} USBH_POWER_STATE;

Is used as a member in the USBH_SET_POWER_STATE data structure.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_NORMAL_POWER</td>
<td>The device is switched to normal operation.</td>
</tr>
<tr>
<td>USBH_SUSPEND</td>
<td>The device is switched to USB Suspend mode.</td>
</tr>
</tbody>
</table>

Is called by the library if a PnP event occurs and if a PnP notification was registered.

Contains information about a USB interface and the related device.

Is used as an notification parameter for the USB_EnumErrorNotification function.

Is called in the context of a TAL timer. In the context of this function all other API function of the bus driver can be called. The removed or added interface can be identified by the interface ID. The client can use this information to find the related USB Interface and close all handles if it was in use, to open it or to collect information about the interface.
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>Is the user defined pointer that was passed to USBH_RegisterPnPNotification. The library does not modify this parameter.</td>
</tr>
<tr>
<td>Event</td>
<td>Specifies the PnP event.</td>
</tr>
<tr>
<td>InterfaceID</td>
<td>Contains the interface ID of the removed or added interface.</td>
</tr>
</tbody>
</table>

Table 45: USBH_ON_PNP_EVENT_FUNC() parameter list

**USBH_ON_ENUM_ERROR_FUNC**

**Definition**

```c
typedef void USBH_ON_ENUM_ERROR_FUNC(
    void                  * Context,
    const USBH_ENUM_ERROR * EnumError );
```

**Description**

Is called in the context of a TAL timer or of a ProcessInterrupt function of a host controller. Before this function is called it must be registered with USBH_RegisterEnumErrorNotification. If an device is not successfully enumerated the function USBH_RestartEnumError can be called to re-start a new enumeration in the context of this function. This callback mechanism is part of the enhanced error recovery. In an embedded system with internal components connected with USB a central application may turn off the power supply for some device to force a reboot or to create an alert.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>Is a user defined pointer that was passed to USBH_RegisterEnumErrorNotification.</td>
</tr>
<tr>
<td>EnumError</td>
<td>Specifies the enumeration error. This pointer is temporary and must not be access after the functions returns.</td>
</tr>
</tbody>
</table>

Table 46: USBH_ON_ENUM_ERROR_FUNC() parameter list

**USBH_ON_COMPLETION_FUNC**

**Definition**

```c
typedef void USBH_ON_COMPLETION_FUNC ( tag_URB * Urb );
```

**Description**

Is called in the context of a TAL timer or of a ProcessInterrupt function of a host controller. Before this function is called it must be registered with USBH_RegisterEnumErrorNotification. If an device is not successfully enumerated the function USBH_RestartEnumError can be called to re-start a new enumeration in the context of this function. This callback mechanism is part of the enhanced error recovery. In an embedded system with internal components connected with USB a central application may turn off the power supply for some device to force a reboot or to create an alert.

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Urb</td>
<td>Contains the URB that was completed.</td>
</tr>
</tbody>
</table>

Table 47: USBH_ON_COMPLETION_FUNC() parameter list

---

**Use of undocumented functions**

Functions, variables and data-types which not explained in this manual are considered internal. They are in no way required to use the software. Your application should not use and rely on any of the internal elements, as only the documented API functions are guaranteed to remain unchanged in future versions of the software.
Mass Storage Device

This chapter describes the IAR PowerPac USB Host Mass storage device class driver and its usage.

Introduction

The USB Host MSD class software allows accessing USB Mass Storage Devices.

It implements the USB Mass Storage Device class protocols specified by the USB Implementers Forum. The entire API of this class driver is prefixed USBH_MSD_.

This chapter describes the architecture, the features and the programming interface of the code.

OVERVIEW

A mass storage device connected to the USB host is added to the file system as device. All operations on the device, such as formatting, reading / writing of files and directories are performed through the API of the file system. With emFile, the device name of the first MSD is "msd:0:"

FEATURES

The following features are provided:

- The command block specification and protocol implementation used by the connected device will be automatically detected.
- It is independent of the file system. An interface to emFile is provided.

RESTRICTIONS

The following restrictions relate to the USB Host library:

- The library supports only USB flash drives. Therefore not all protocol commands are implemented.

REQUIREMENTS

To use the MSD class driver to perform file and directory operations, a file system (typically emFile) is required.

EXAMPLE CODE

Example code which is provided in the file OS_USBH_MSD.c.

The example shows the capacity of the connected device, files in the root directory and creates and writes to a file.

SUPPORTED PROTOCOLS

The following table contains an overview about the implemented command protocols.

<table>
<thead>
<tr>
<th>Command block specification</th>
<th>Implementation</th>
<th>Related documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCSI transparent command set</td>
<td>All necessary commands for accessing flash devices.</td>
<td>Mass Storage Class Specification Overview Revision 1.2., SCSI-2 Specification September 1993 Rev.10 (X3T9.2 Project 275D)</td>
</tr>
<tr>
<td>SFF-8070i</td>
<td>All necessary commands for accessing flash devices.</td>
<td>SFF-8070i Specification for ATAPI Removable Rewritable Media Devices (SFF Committee:document SFF-8070 Rev 1.3)</td>
</tr>
</tbody>
</table>
The following table contains an overview about the implemented transport protocols.

<table>
<thead>
<tr>
<th>Protocol implementation</th>
<th>Implementation</th>
<th>Related documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bulk-Only transport</td>
<td>All commands implemented.</td>
<td>Universal Serial Bus Mass Storage Class Bulk-Only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transport Rev.1.0.</td>
</tr>
</tbody>
</table>

### API Functions

This chapter describes the USB Host MSD API functions. These functions are defined in the header file ”USBH.h”.

#### Function Description

- **USBH_MSD_Init()**
  
  Initializes the USBH MSD library.

  **Prototype**

  ```c
  int USBH_MSD_Init(
      USBH_MSD_LUN_NOTIFICATION_FUNC * pfLunNotification,
      void * pContext);
  ```

  **Parameters**

  - **pfLunNotification**
    
    Pointer to a function that shall be called when a new device notification is received. This means when a device is attached and ready or when it is removed.

  - **pContext**
    
    Pointer to a context that should be passed when the `pfLunNotification` is called.

  **Additional information**

  Performs basic initialization of the library. Has to be called before any other library function is called. It can be called again to reinitialize the library. In this case all internal states like added devices or handles are lost.
Example:

```c
/*
 * _cbOnAddRemoveDevice
 * Function description
 * Callback, called when a device is added or removed.
 * Call in the context of the USBH_Task.
 * The functionality in this routine should not block
 */
static void _cbOnAddRemoveDevice(
    void           * pContext,
    U8               DevIndex,
    USBH_MSD_EVENT  Event) {
  switch (Event) {
    case USBH_MSD_EVENT_ADD:
      printf("**** Device added\n");
      break;
    case USBH_MSD_EVENT_REMOVE:
      printf("**** Device removed\n");
      break;
    default:  // Should never happen
  }
}

// Init MSD, after call to FS_Init(). See example code in OS_USBH_MSD.c

USBH_MSD_Exit()

Description
Releases all resources, closes all handles to the USB bus driver and unregister all notification functions.

Prototype
int USBH_MSD_Exit(void);

Additional information
Has to be called if the application is closed before the USB bus driver is closed.

USBH_MSD_ReadSectors()

Description
Reads sectors from a USB Mass Storage device.

Prototype
void USBH_MSD_ReadSectors(
    U8   UnitId,
    U32  SectorIndex,
    U32  NumSectors,
    U8  * pBuffer );
```
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnitId</td>
<td>0-based Unit Id. The first unit in the system has UnitId of 0, the second one a value of 1. If you are dealing with multiple devices or devices with multiple LUNs, it is good practice to retrieve the UnitIds at run time.</td>
</tr>
<tr>
<td>SectorIndex</td>
<td>0-based sector index: of the first sector to read. First sector has index 0, second sector has index 1, ...</td>
</tr>
<tr>
<td>NumSectors</td>
<td>Determines the number of sectors to read.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>Pointer to a byte buffer. The caller is responsible for the storage of the buffer.</td>
</tr>
</tbody>
</table>

Table 50: USBH_MSD_ReadSectors() parameter list

Return Value

Returns USBH_MSD_STATUS_SUCCESS if the sectors have been successfully read from the device and copied to the Buffer. If reading from the specified device fails, the function returns USBH_MSD_STATUS_READ to indicate the error.

USBH_MSD_WriteSectors()

Description

Writes sectors to a USB Mass Storage device.

Prototype

```c
void USBH_MSD_WriteSectors(
    U8          UnitId,
    U32         SectorIndex,
    U32         NumSectors,
    const U8  * pBuffer );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnitId</td>
<td>0-based Unit Id. The first unit in the system has UnitId of 0, the second one a value of 1. If you are dealing with multiple devices or devices with multiple LUNs, it is good practice to retrieve the UnitIds at run time.</td>
</tr>
<tr>
<td>SectorIndex</td>
<td>0-based sector index: of the first sector to read. First sector has index 0, second sector has index 1, ...</td>
</tr>
<tr>
<td>NumSectors</td>
<td>Determines the number of sectors to write.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>Pointer to a buffer containing the data to be written.</td>
</tr>
</tbody>
</table>

Table 51: USBH_MSD_WriteSectors() parameter list

Return Value

Returns USBH_MSD_STATUS_SUCCESS if the sectors have been successfully copied from the Buffer and written to the device. If writing to the specified device fails the function returns USBH_MSD_STATUS_WRITE to indicate the error. The function returns USBH_MSD_STATUS_WRITE_PROTECT if the medium is write protected.

USBH_MSD_GetUnitInfo()

Description

Returns basic information about the logical unit (LUN).

Prototype

```c
int USBH_MSD_GetUnitInfo(
    U8          UnitId,
    USBH_MSD_UNIT_INFO * pInfo );
```
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnitId</td>
<td>0-based Unit Id. The first unit in the system has UnitId of 0, the second one a value of 1. If you are dealing with multiple devices or devices with multiple LUNs, it is good practice to retrieve the UnitIds at run time.</td>
</tr>
<tr>
<td>pInfo</td>
<td>Pointer to a caller provided storage buffer. It receives the information about the LUN in case of success.</td>
</tr>
</tbody>
</table>

Table 52: USBH_MSD_GetUnitInfo() parameter list

Return Value

Returns USBH_MSD_STATUS_SUCCESS in case of success. If the device is not a USB Mass Storage device, USBH_MSD_STATUS_ERROR will be returned. USBH_MSD_STATUS_TIMEOUT is returned if the function call timed out.

**USBH_MSD_GetStatus()**

Description

Checks the state of a device unit.

Prototype

```c
int USBH_MSD_GetStatus( U8 UnitId );
```

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnitId</td>
<td>0-based Unit Id. The first unit in the system has UnitId of 0, the second one a value of 1. If you are dealing with multiple devices or devices with multiple LUNs, it is good practice to retrieve the UnitIds at run time.</td>
</tr>
</tbody>
</table>

Table 53: USBH_MSD_GetStatus() parameter list

Return Value

If the device is working, USBH_MSD_STATUS_SUCCESS is returned. If the device does not work correctly or is disconnected the function returns USBH_MSD_STATUS_ERROR.

Data Structures

This chapter describes the used structures defined in the header file "USBH.h".

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_MSD_UNIT_INFO</td>
<td>Contains logical unit information.</td>
</tr>
</tbody>
</table>

Table 54: IAR PowerPac USB Host MSD data structure overview

**USBH_MSD_UNIT_INFO**

Definition

```c
typedef struct tag_USB_MSD_UNIT_INFO {
    U32 TotalSectors;
    U16 BytesPerSector;
} USBH_MSD_UNIT_INFO;
```

Description

Contains logical unit information.
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TotalSectors</td>
<td>Contains the number of total sectors available on the LUN.</td>
</tr>
<tr>
<td>BytesPerSector</td>
<td>Contains the number of bytes per sector.</td>
</tr>
</tbody>
</table>

Table 55: USBH_MSD_UNIT_INFO() parameter list

Function Types

This chapter describes the used structures defined in the header file "USBH.h".

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_MSD_LUN_NOTIFICATION_FUNC</td>
<td>Type of callback set in USBH_MSD_Init.</td>
</tr>
</tbody>
</table>

Table 56: IAR PowerPac USB Host MSD function type overview

**USBH_MSD_LUN_NOTIFICATION_FUNC**

**Definition**

typedef void tag_USB_MSD_LUN_NOTIFICATION_FUNC {
    void           pContext;
    U8             DevIndex;
    USBH_MSD_EVENT Event
} USBH_MSD_UNIT_INFO;

**Description**

This callback function is called when new a logical units is added or removed. To get detailed information USBH_MSD_GetLuns has to be called. The LUN indexes must be used to get access to an specified unit of the device.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pContext</td>
<td>Pointer to a context that was set by the user when the USBH_MSD_Init() was called.</td>
</tr>
<tr>
<td>DevIndex</td>
<td>Zero based index of the device that was attached or removed. First device has index 0, second one has index 1, etc.</td>
</tr>
<tr>
<td>Event</td>
<td>Gives information about the event that has occurred. The following events are currently available: USBH_MSD_EVENT_ADD_LUN - A device was attached. USBH_MSD_EVENT_REMOVE_LUN - A device was removed.</td>
</tr>
</tbody>
</table>

Table 57: USBH_MSD_LUN_NOTIFICATION_FUNC() parameter list

**Error Codes**

This chapter describes the error codes which are defined in the header file USH.h.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_MSD_STATUS_SUCCESS</td>
<td>(0)</td>
</tr>
<tr>
<td>USBH_MSD_STATUS_ERROR</td>
<td>(-1)</td>
</tr>
<tr>
<td>USBH_MSD_STATUS_PARAMETER</td>
<td>(-2)</td>
</tr>
<tr>
<td>USBH_MSD_STATUS_LENGTH</td>
<td>(-3)</td>
</tr>
<tr>
<td>USBH_MSD_STATUS_TIMEOUT</td>
<td>(-4)</td>
</tr>
<tr>
<td>USBH_MSD_STATUS_COMMAND_FAILED</td>
<td>(-5)</td>
</tr>
<tr>
<td>USBH_MSD_STATUS_INTERFACE_PROTOCOL</td>
<td>(-6)</td>
</tr>
<tr>
<td>USBH_MSD_STATUS_INTERFACE_SUB_CLASS</td>
<td>(-7)</td>
</tr>
<tr>
<td>USBH_MSD_STATUS_PIPE_STALLED</td>
<td>(-9)</td>
</tr>
<tr>
<td>USBH_MSD_STATUS_TRANSMISSION</td>
<td>(-10)</td>
</tr>
<tr>
<td>USBH_MSD_STATUS_SENSE_STOP</td>
<td>(-11)</td>
</tr>
</tbody>
</table>

Table 58: IAR PowerPac USB Host MSD error code overview
Mass Storage Device

USBH_MSD_STATUS_SUCCESS
Description
The operation has been successfully completed.

USBH_MSD_STATUS_ERROR
Description
The operation has been completed with an error.

USBH_MSD_STATUS_PARAMETER
Description
A parameter is incorrect.

USBH_MSD_STATUS_LENGTH
Description
The operation detected a length error.

USBH_MSD_STATUS_TIMEOUT
Description
The timeout of the operation has expired. This error code is used in all layers.

USBH_MSD_STATUS_COMMAND_FAILED
Description
This error is reported if the command code was sent successfully but the status returned from the device indicates a command error.

USBH_MSD_STATUS_INTERFACE_PROTOCOL
Description
The used interface protocol is not supported. The interface protocol is defined by the interface descriptor.

USBH_MSD_STATUS_INTERFACE_SUB_CLASS
Description
The used interface sub class is not supported. The interface sub class is defined by the interface descriptor.

USBH_MSD_STATUS_PIPE_STALLED
Description
A pipe is stalled. This error is reported from the USB driver layer.

USBH_MSD_STATUS_TRANSMISSION
Description
A USB bus error occurred. This may be caused by a CRC error, a toggle error or another USB bus error. This error is reported from the USB driver layer.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_MSD_STATUS_SENSE_REPEAT</td>
<td>(-12)</td>
</tr>
<tr>
<td>USBH_MSD_STATUS_WRITE_PROTECT</td>
<td>(-13)</td>
</tr>
</tbody>
</table>

Table 58: IAR PowerPac USB Host MSD error code overview (Continued)
**USBH_MSD_STATUS_SENSE_STOP**

**Description**

This error indicates that the device has not accepted the command. The execution result of the command is stored in the sense element of the unit. The library will not repeat the command.

**USBH_MSD_STATUS_SENSE_REPEAT**

**Description**

This error indicates that the device has not accepted the command. The execution result of the command is stored in the sense element of the unit. The library repeats the command after detection of the sense code.

**USBH_MSD_STATUS_WRITE_PROTECT**

**Description**

This error indicates that the medium is write protected. It will be returned by USBH_MSD_WriteSectors if writing to the medium is not allowed.
Human Interface Device

This chapter describes the IAR PowerPac USB Host Human interface device class driver and its usage.

**Introduction**

The USB Host HID class software allows accessing USB Human interface Devices.

It implements the USB Human interface Device class protocols specified by the USB Implementers Forum. The entire API of this class driver is prefixed USBH_HID_.

This chapter describes the architecture, the features and the programming interface of the code.

**OVERVIEW**

2 types of HIDs are currently supported: Keyboard and Mouse. For both, the application can set a callback routine which is called whenever a message from either one is received.

**RESTRICTIONS**

The following restrictions relate to the USB Host library:

- Currently, only Keyboard and mouse devices are supported. The HID report descriptor is not currently parsed.

**EXAMPLE CODE**

Example code which is provided in the file OS_USBH_HID.c.

The example outputs mouse and keyboard events to the terminal (using printf).

**API Functions**

This chapter describes the USB Host HID API functions. These functions are defined in the header file USBH.h.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_HID_Init()</td>
<td>Initializes the USBH HID library.</td>
</tr>
<tr>
<td>USBH_HID_Exit()</td>
<td>Releases all resources, closes all handles to the USB bus driver and unregisters all notification functions.</td>
</tr>
<tr>
<td>USBH_HID_SetOnMouseStateChange()</td>
<td>Set function to be called in case of mouse events.</td>
</tr>
<tr>
<td>USBH_HID_SetOnKeyboardStateChange()</td>
<td>Set function to be called in case of keyboard events.</td>
</tr>
</tbody>
</table>

*Table 59: IAR PowerPac USB Host HID API function overview*

**USBH_HID_INIT()**

**Description**

Initializes the USBH HID library.

**Prototype**

```c
int USBH_HID_Init();
```

**Additional information**

Performs basic initialization of the library. Has to be called before any other USB_HID function is called.
USBH_HID_Exit()

Description
Releases all resources, closes all handles to the USB bus driver and unregister all notification functions.

Prototype
int USBH_HID_Exit(void);

Additional information
Has to be called if the application is closed before the USB bus driver is closed.

USBH_HID_SETONMOUSESTATECHANGE()

Description
Set function to be called in case of mouse events.

Prototype
void USBH_HID_SetOnMouseStateChange (USBH_HID_ON_MOUSE_FUNC * pfOnChange);

Example

/*********************************************************************
*       _OnMouseChange
*/
static void _OnMouseChange(USBH_HID_MOUSE_DATA * pMouseData) {
    _MouseData = *pMouseData;
    _EventOccurred |= MOUSE_EVENT;
    OS_EVENT_Pulse(&_Event);
}

USBH_HID_SETONKEYBOARDSTATECHANGE()

Description
Set function to be called in case of keyboard events.

Prototype
void USBH_HID_SetOnKeyboardStateChange(USBH_HID_ON_KEYBOARD_FUNC * pfOnChange);

Example

/*********************************************************************
*       _OnKeyboardChange
*/
static void _OnKeyboardChange(USBH_HID_KEYBOARD_DATA * pKeyData) {
    _KeyData = *pKeyData;
    _EventOccurred |= KEYBOARD_EVENT;
    OS_EVENT_Pulse(&_Event);
}

Data Structures

This chapter describes the used structures defined in the header file USBH.h.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_HID_KEYBOARD_DATA</td>
<td>Contains keyboard state information.</td>
</tr>
<tr>
<td>USBH_HID_MOUSE_DATA</td>
<td>Contains mouse state information.</td>
</tr>
</tbody>
</table>

Table 60: IAR PowerPac USB Host HID data structure overview
**USBH_HID_KEYBOARD_DATA**

**Definition**

```c
typedef struct {
    unsigned code;
    int value;
} USBH_HID_KEYBOARD_DATA;
```

**Description**

Contains keyboard state information.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>Contains the keycode.</td>
</tr>
<tr>
<td>value</td>
<td>Keyboard state info. Refer to sample code for more information.</td>
</tr>
</tbody>
</table>

Table 61: USBH_HID_KEYBOARD_DATA() parameter list

**USBH_HID_MOUSE_DATA**

**Definition**

```c
typedef struct {
    int xChange;
    int yChange;
    int WheelChange;
    int ButtonState;
} USBH_HID_MOUSE_DATA;
```

**Description**

Contains mouse state information.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xChange</td>
<td>Change of x-position since last event</td>
</tr>
<tr>
<td>yChange</td>
<td>Change of y-position since last event</td>
</tr>
<tr>
<td>WheelChange</td>
<td>Change of wheel-position since last event (if wheel is present)</td>
</tr>
<tr>
<td>ButtonState</td>
<td>Each bit corresponds to one button on the mouse. If the bit is set, the corresponding button is pressed. Typically, bit 0 corresponds to the left mouse button, bit 1 corresponds to the right mouse button, bit 2 corresponds to the middle mouse button</td>
</tr>
</tbody>
</table>

Table 62: USBH_HID_MOUSE_DATA() parameter list

**Function Types**

This chapter describes the used structures defined in the header file `USBH.h`.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_HID_ON_KEYBOARD_FUNC</td>
<td>This callback function is called when a key is pressed or released.</td>
</tr>
<tr>
<td>USBH_HID_ON_MOUSE_FUNC</td>
<td>This callback function is called when the mouse is moved or a button is pressed or released.</td>
</tr>
</tbody>
</table>

Table 63: IAR PowerPac USB Host HID function type overview

**USBH_HID_ON_KEYBOARD_FUNC**

**Definition**

```c
typedef void (USBH_HID_ON_KEYBOARD_FUNC)(USBH_HID_KEYBOARD_DATA * pKeyData);
```
Description

This callback function is called when a key is pressed or released.

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pKeyData</td>
<td>Points to the structure containing status information.</td>
</tr>
</tbody>
</table>

Table 64: USBH_HID_ON_KEYBOARD_FUNC() parameter list

USBH_HID_ON_MOUSE_FUNC

Definition

typedef void (USBH_HID_ON_MOUSE_FUNC)(USBH_HID_MOUSE_DATA * pMouseData);

Description

This callback function is called when the mouse is moved or a button is pressed or released.

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMouseData</td>
<td>Points to the structure containing status information.</td>
</tr>
</tbody>
</table>

Table 65: USBH_HID_ON_MOUSE_FUNC() parameter list
Configuring IAR PowerPac USB Host

IAR PowerPac USB Host can be used without changing any of the compile-time flags. All compile-time configuration flags are preconfigured with valid values, which match the requirements of most applications. Network interface drivers can be added at runtime.

The default configuration of IAR PowerPac USB Host can be changed via compile-time flags which can be added to USBH_Conf.h. USBH_Conf.h is the main configuration file for the IAR PowerPac USB Host stack.

Runtime configuration

Every driver folder includes a configuration file with implementations of runtime configuration functions explained in this chapter. These functions can be customized.

**DRIVER HANDLING**

USBH_X_Config() is called at initialization of the USB Host stack. It is called by the USB Host stack during USBH_Init(). USBH_X_Config() should help to bundle the process of adding and configuring the driver.

**USBH_X_Config()**

**Description**
Helper function to prepare and configure the USB Host stack.

**Prototype**

```c
void USBH_X_Config(void);
```

**Additional information**
This function is called by the startup code of the USB Host stack from USBH_Init().

Compile-time configuration

The following types of configuration macros exist:

**Binary switches "B"**

Switches can have a value of either 0 or 1, for deactivated and activated respectively. Actually, anything other than 0 works, but 1 makes it easier to read a configuration file. These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macros.

**Numerical values "N"**

Numerical values are used somewhere in the code in place of a numerical constant. A typical example is the configuration of the sector size of a storage medium.

**Function replacements "F"**

Macros can basically be treated like regular functions although certain limitations apply, as a macro is still put into the code as simple text replacement. Function replacements are mainly used to add specific functionality to a module which is highly hardware-dependent. This type of macro is always declared using brackets (and optional parameters).
COMPILE-TIME CONFIGURATION SWITCHES

<table>
<thead>
<tr>
<th>Type</th>
<th>Symbolic name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>USBH_DEBUG</td>
<td>0</td>
<td>Macro to define the debug level of the IAR PowerPac USB Host build.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Debug macros

<table>
<thead>
<tr>
<th>Type</th>
<th>Symbolic name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>USBH_MEMCPY</td>
<td>memcpy (C-routine in standard C-library)</td>
<td>Macro to define an optimized memcpy routine to speed up the stack. An optimized memcpy routine is typically implemented in assembly language. Optimized version for the IAR compiler is supplied.</td>
</tr>
<tr>
<td>F</td>
<td>USBH_MEMSET</td>
<td>memset (C-routine in standard C-library)</td>
<td>Macro to define an optimized memset routine to speed up the stack. An optimized memset routine is typically implemented in assembly language.</td>
</tr>
<tr>
<td>F</td>
<td>USBH_MEMMOVE</td>
<td>memmove (C-routine in standard C-library)</td>
<td>Macro to define an optimized memmove routine to speed up the stack. An optimized memmove routine is typically implemented in assembly language.</td>
</tr>
<tr>
<td>F</td>
<td>USBH_MEMCMP</td>
<td>memcmp (C-routine in standard C-library)</td>
<td>Macro to define an optimized memcmp routine to speed up the stack. An optimized memcmp routine is typically implemented in assembly language.</td>
</tr>
</tbody>
</table>

Optimization macros

DEBUG LEVEL

IAR PowerPac USB Host can be configured to display debug information at higher debug levels to locate a problem (Error) or potential problem. To display information, IAR PowerPac USB Host uses the logging routines. These routines can be blank, they are not required for the functionality of IAR PowerPac USB Host. In a target system, they are typically not required in a release (production) build, since a production build typically uses a lower debug level.

If (USBH_DEBUG == 0):
used for release builds. Includes no debug options.
If (USBH_DEBUG == 1):
USBH_PANIC() is mapped to USBH_Panic().
If (USBH_DEBUG >= 2):
USBH_PANIC() is mapped to USBH_Panic() and logging support is activated. 
Debugging

IAR PowerPac USB Host comes with various debugging options. These includes optional warning and log outputs, as well as other run-time options which perform checks at run time as well as options to drop incoming or outgoing packets to test stability of the implementation on the target system.

Message output

The debug builds of IAR PowerPac USB Host include a fine grained debug system which helps to analyze the correct implementation of the stack in your application. All modules of the USB Host stack can output logging and warning messages via terminal I/O, if the specific message type identifier is added to the log and/or warn filter mask. This approach provides the opportunity to get and interpret only the logging and warning messages which are relevant for the part of the stack that you want to debug.

By default, all warning messages are activated in all IAR PowerPac USB Host sample configuration files. All logging messages are disabled except for the messages from the initialization phase.

API functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_SetLogFilter()</td>
<td>Sets the mask that defines which logging message should be displayed.</td>
</tr>
<tr>
<td>USBH_SetWarnFilter()</td>
<td>Sets the mask that defines which warning message should be displayed.</td>
</tr>
<tr>
<td>USBH_AddLogFilter()</td>
<td>Adds an additional filter condition to the mask which specifies the logging messages that should be displayed.</td>
</tr>
<tr>
<td>USBH_AddWarnFilter()</td>
<td>Adds an additional filter condition to the mask which specifies the warning messages that should be displayed.</td>
</tr>
<tr>
<td>USBH_LOG()</td>
<td>Called if the stack encounters a critical situation.</td>
</tr>
<tr>
<td>USBH_WARN()</td>
<td>Called if the stack encounters a critical situation.</td>
</tr>
<tr>
<td>USBH_PANIC()</td>
<td>Called if the stack encounters a critical situation.</td>
</tr>
</tbody>
</table>

Table 66: IAR PowerPac USB Host debugging API function overview

**USBH_SetLogFilter()**

**Description**

Sets a mask that defines which logging message should be logged. Logging messages are only available in debug builds of IAR PowerPac USB Host.

**Prototype**

```c
void USBH_SetLogFilter( U32 FilterMask );
```

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FilterMask</td>
<td>Specifies which logging messages should be displayed.</td>
</tr>
</tbody>
</table>

Table 67: USBH_SetLogFilter() parameter list

**Additional information**

Should be called from `USBH_X_Config()`. By default, the filter condition `USBH_MTYPE_INIT` is set.
IAR PowerPac™ USB Host

**USBH_SetWarnFilter()**

**Description**
Sets a mask that defines which warning messages should be logged. Warning messages are only available in debug builds of IAR PowerPac USB Host.

**Prototype**

```c
void USBH_SetWarnFilter( U32 FilterMask );
```

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FilterMask</td>
<td>Specifies which warning messages should be displayed.</td>
</tr>
</tbody>
</table>

*Table 68: USBH_SetWarnFilter() parameter list*

**Additional information**
Should be called from `USBH_X_Config()`. By default, all filter conditions are set.

**USBH_AddLogFilter()**

**Description**
Adds an additional filter condition to the mask which specifies the logging messages that should be displayed.

**Prototype**

```c
void USBH_AddLogFilter( U32 FilterMask );
```

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FilterMask</td>
<td>Specifies which logging messages should be added to the filter mask.</td>
</tr>
</tbody>
</table>

*Table 69: USBH_AddLogFilter() parameter list*

**Additional information**
`USBH_AddLogFilter()` can also be used to remove a filter condition which was set before. It adds/removes the specified filter to/from the filter mask via a disjunction.

**USBH_AddWarnFilter()**

**Description**
Adds an additional filter condition to the mask which specifies the warning messages that should be displayed.

**Prototype**

```c
void USBH_AddWarnFilter( U32 FilterMask );
```

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FilterMask</td>
<td>Specifies which warning messages should be added to the filter mask.</td>
</tr>
</tbody>
</table>

*Table 70: USBH_AddWarnFilter() parameter list*

**Additional information**
`USBH_AddWarnFilter()` can also be used to remove a filter condition which was set before. It adds/removes the specified filter to/from the filter mask via a disjunction.

**USBH_LOG()**

**Description**
This macro maps to a function in debug builds only. The function outputs logging messages. In a release build, this macro is defined empty.

**Prototype**

```c
USBH_LOG( const char * s );
```
Debugging

Parameter

Parameter | Description
---|---
s | Pointer to the string to be sent.

Table 71: USBH_LOG() parameter list

**USBH_WARN()**

Description

This macro maps to a function in debug builds only. The function outputs warning messages. In a release build, this macro is defined empty.

Prototype

USBH_WARN( const char * s );

Parameter

Parameter | Description
---|---
s | Pointer to the string to be sent.

Table 72: USBH_WARN() parameter list

**USBH_PANIC()**

Description

This macro is called by the stack code when it detects a situation that should not be occurring and the stack can not continue. The intention for the USBH_PANIC() macro is to invoke whatever debugger may be in use by the programmer. In this way, it acts like an embedded breakpoint.

Prototype

USBH_PANIC( const char * sError );

Parameter

Parameter | Description
---|---
sError | Pointer to the string to be sent.

Table 73: USBH_PANIC() parameter list

**Additional information**

This macro maps to a function in debug builds only. If USBH_DEBUG > 0, the macro maps to the stack internal function void USBH_Panic (const char * sError). USBH_Panic() disables all interrupts to avoid further task switches, outputs sError via terminal I/O and loops forever. When using an emulator, you should set a breakpoint at the beginning of this routine or simply stop the program after a failure. The error code is passed to the function as parameter.

In a release build, this macro is defined empty, so that no additional code will be included by the linker.

**Message types**

The same message types are used for log and warning messages. Separate filters can be used for both log and warnings.

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_MTYPE_INIT</td>
<td>Activates output of messages from the initialization of the stack that should be logged.</td>
</tr>
<tr>
<td>USBH_MTYPE_CORE</td>
<td>Activates output of messages from the core of the stack that should be logged.</td>
</tr>
<tr>
<td>USBH_MTYPE_ALLOC</td>
<td>Activates output of messages from the memory allocating module of the stack that should be logged.</td>
</tr>
<tr>
<td>USBH_MTYPE_DRIVER</td>
<td>Activates output of messages from the driver that should be logged.</td>
</tr>
<tr>
<td>USBH_MTYPE_MEM</td>
<td>Activates output of messages from the memory that should be logged.</td>
</tr>
</tbody>
</table>

Table 74: USB Host message types
<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_MTYPE_OHCI</td>
<td>Activates output of messages from the Open Host Controller Interface that should be logged.</td>
</tr>
<tr>
<td>USBH_MTYPE_UBD</td>
<td></td>
</tr>
<tr>
<td>USBH_MTYPE_PNP</td>
<td></td>
</tr>
<tr>
<td>USBH_MTYPE_DEVICE</td>
<td></td>
</tr>
<tr>
<td>USBH_MTYPE_HUB</td>
<td></td>
</tr>
<tr>
<td>USBH_MTYPE_MSD</td>
<td></td>
</tr>
<tr>
<td>USBH_MTYPE_HID</td>
<td></td>
</tr>
<tr>
<td>USBH_MTYPE_APPLICATION</td>
<td></td>
</tr>
</tbody>
</table>

Table 74: USB Host message types
OS integration

IAR PowerPac USB Host is designed to be used in a multitasking environment. The interface to the operating system is encapsulated in a single file, the USB-Host/OS interface. For emUSB-Host, all functions required for this USB-Host/OS interface are implemented in a single file which comes with IAR PowerPac USB Host.

This chapter provides descriptions of the functions required to fully support IAR PowerPac USB Host in multitasking environments.

General information

The complexity of the IP/OS Interface depends on the task model selected. All OS interface functions for IAR PowerPac USB Host are implemented in `USBH_OS.c` which is located in the root folder of the USB Host stack.

OS layer API functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General macros</strong></td>
<td></td>
</tr>
<tr>
<td>USBH_OS_Delay()</td>
<td>Blocks the calling task for a given time.</td>
</tr>
<tr>
<td>USBH_OS_DisableInterrupt()</td>
<td>Disables interrupts.</td>
</tr>
<tr>
<td>USBH_OS_EnableInterrupt()</td>
<td>Enables interrupts.</td>
</tr>
<tr>
<td>USBH_OS_GetTime32()</td>
<td>Returns the current system time in ticks. Returns the current system time in ms. On 32-bit systems, the value will wrap around after approximately 49.7 days. This is taken into account by the stack.</td>
</tr>
<tr>
<td>USBH_OS_Init()</td>
<td>Creates and initializes all objects required for task synchronization. These are 2 events (for USBH_Task and USBH_RxTask) and one semaphore for protection of critical code which may not be executed from multiple task at the same time.</td>
</tr>
<tr>
<td>USBH_OS_Lock()</td>
<td>The stack requires a single lock, typically a resource semaphore or mutex. This function locks this object, guarding sections of the stack code against other tasks. If the entire stack executes from a single task, no functionality is required here.</td>
</tr>
<tr>
<td>USBH_OS_Unlock()</td>
<td>Unlocks the single lock used locked by a previous call to USBH_OS_Lock().</td>
</tr>
<tr>
<td>USBH_OS_SignalNetEvent()</td>
<td>Wakes the USBH_Task if it is waiting for a NET-event or timeout in the function USBH_OS_WaitNetEvent().</td>
</tr>
<tr>
<td>USBH_OS_WaitNetEvent()</td>
<td>Called from USBH_Task only. Blocks until the timeout expires or a NET-event occurs, meaning USBH_OS_SignalNetEvent() is called from an other task or ISR.</td>
</tr>
<tr>
<td>USBH_OS_SignalRxEvent()</td>
<td>Wakes the USBH_RxTask if it is waiting for a NET-event or timeout in the function USBH_OS_WaitRxEvent().</td>
</tr>
<tr>
<td>USBH_OS_WaitRxEvent()</td>
<td>Optional. Called from USBH_RxTask, if it is used to receive data. Blocks until the timeout expires or a NET-event occurs, meaning USBH_OS_SignalRxEvent() is called from the ISR.</td>
</tr>
<tr>
<td>USBH_OS_WaitItem()</td>
<td>Suspend a task which needs to wait for a object. This object is identified by a pointer to it and can be of any type, for example a socket.</td>
</tr>
</tbody>
</table>

Table 75: Target OS interface function list
EXAMPLES

OS interface routine for IAR PowerPac USB Host

All OS interface routines are implemented in `USBH_OS.c` which is located in the root folder of the USB Host stack.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBH_OS_WaitItemTimed()</td>
<td>Suspend a task which needs to wait for an object. This object is identified by a pointer to it and can be of any type, for example a socket. The second parameter defines the maximum time in timer ticks until the event have to be signaled.</td>
</tr>
<tr>
<td>USBH_OS_SignalItem()</td>
<td>Sets an event object to signaled state, or resumes tasks which are waiting at the event object. Function is called from a task, not an ISR.</td>
</tr>
</tbody>
</table>

Table 75: Target OS interface function list
Performance & resource usage

This chapter covers the performance and resource usage of IAR PowerPac USB Host. It contains information about the memory requirements in typical systems which can be used to obtain sufficient estimates for most target systems.

Memory footprint

IAR PowerPac USB Host is designed to fit many kinds of embedded design requirements. Several features can be excluded from a build to get a minimal system. Note that the values are only valid for the given configuration.

System

The following table shows the hardware and the toolchain details of the project:

<table>
<thead>
<tr>
<th>Detail</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>ARM7</td>
</tr>
<tr>
<td>Tool chain</td>
<td>IAR Embedded Workbench for Cortex-M3, V5.30</td>
</tr>
<tr>
<td>Compiler options</td>
<td>Highest size optimization;</td>
</tr>
</tbody>
</table>

*Table 76: ARM7 sample configuration*

ROM

The following table shows the ROM requirement of IAR PowerPac USB Host:

<table>
<thead>
<tr>
<th>Description</th>
<th>ROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAR PowerPac USB Host core incl. driver</td>
<td>app. 20 KBytes</td>
</tr>
<tr>
<td>HID class support</td>
<td>app. 5 KBytes</td>
</tr>
<tr>
<td>MSD class support</td>
<td>app. 8 KBytes + sizeof(Filesystem)*</td>
</tr>
</tbody>
</table>

The memory requirements of a interface driver is about 1.5 - 2.0 Kbytes.

RAM

The following table shows the ROM requirement of IAR PowerPac USB Host:

<table>
<thead>
<tr>
<th>Description</th>
<th>ROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAR PowerPac USB Host core incl. driver</td>
<td>app. 20 KBytes</td>
</tr>
</tbody>
</table>

The memory requirements of a interface driver is about 1.5 - 2.0 Kbytes.

* ROM size of emFile File system is app. 10KBytes
## Performance System

<table>
<thead>
<tr>
<th>Detail</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>ARM7 with integrated MAC running with 48Mhz</td>
</tr>
<tr>
<td>Tool chain</td>
<td>IAR Embedded Workbench for Cortex-M3 V530</td>
</tr>
<tr>
<td>Compiler options</td>
<td>Highest speed optimization;</td>
</tr>
</tbody>
</table>

*Table 77: ARM7 sample configuration*

The following table shows the send and receive speed of IAR PowerPac USB Host:

<table>
<thead>
<tr>
<th>Description</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send speed</td>
<td>400-1000 KByte/sec</td>
</tr>
<tr>
<td>Receive speed</td>
<td>400-1000 KByte/sec</td>
</tr>
</tbody>
</table>
Related Documents

- USB device class specifications (Audio, HID, Printer, etc.), http://www.usb.org
### Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>Central Processing Unit. The “brain” of a microcontroller; the part of a processor that carries out instructions.</td>
</tr>
<tr>
<td><strong>EOT</strong></td>
<td>End Of Transmission.</td>
</tr>
<tr>
<td><strong>FIFO</strong></td>
<td>First-In, First-Out.</td>
</tr>
<tr>
<td><strong>ISR</strong></td>
<td>Interrupt Service Routine. The routine is called automatically by the processor when an interrupt is acknowledged. ISRs must preserve the entire context of a task (all registers).</td>
</tr>
<tr>
<td><strong>RTOS</strong></td>
<td>Real-time Operating System.</td>
</tr>
<tr>
<td><strong>Scheduler</strong></td>
<td>The program section of an RTOS that selects the active task, based on which tasks are ready to run, their relative priorities, and the scheduling system being used.</td>
</tr>
<tr>
<td><strong>Stack</strong></td>
<td>An area of memory with LIFO storage of parameters, automatic variables, return addresses, and other information that needs to be maintained across function calls. In multitasking systems, each task normally has its own stack.</td>
</tr>
<tr>
<td><strong>Superloop</strong></td>
<td>A program that runs in an infinite loop and uses no real-time kernel. ISRs are used for real-time parts of the software.</td>
</tr>
<tr>
<td><strong>Task</strong></td>
<td>A program running on a processor. A multitasking system allows multiple tasks to execute independently from one another.</td>
</tr>
<tr>
<td><strong>Tick</strong></td>
<td>The OS timer interrupt. Usually equals 1 ms.</td>
</tr>
</tbody>
</table>
Index

C
copyright notice ........................................... 2
Core data structures
   URB ........................................... 36
   USBH_BULK_INT_REQUEST  ....................... 34
   USBH_CONTROL_REQUEST ......................... 33
   USBH_ENDPOINT_REQUEST  ......................... 35
   USBH_ENUM_ERROR  ................................ 31
   USBH_EP_MASK .................................... 32
   USBH_FUNCTION ................................... 38
   USBH_HEADER ..................................... 37
   USBH_INTERFACE_INFO ......................... 30
   USBH_INTERFACE_MASK ......................... 29
   USBH_ISO_FRAME  ................................ 34
   USBH_ISO_REQUEST ................................ 34
   USBH_PNP_EVENT .................................. 38
   USBH_PNP_NOTIFICATION  ...................... 37
   USBH_POWER_STATE ................................ 40
   USBH_SET_CONFIGURATION ....................... 35
   USBH_SET_INTERFACE  ............................ 36
   USBH_SET_POWER_STATE ......................... 36
   USBH_SPEED ...................................... 38
Core function types
   USBH_ON_COMPLETION_FUNC ..................... 41
   USBH_ON_ENUM_ERROR_FUNC .................... 41
   USBH_ON_PNP_EVENT_FUNC ....................... 40
Core functions
   USBH_CloseInterface() ......................... 22
   USBH_CreateInterfaceList() ................... 18
   USBH_DestroyInterfaceList() .................. 19
   USBH_EnumerateDevices() ...................... 17
   USBH rteExit() .................................. 17
   USBH_GetCurrentConfigurationDescriptor() .... 23
   USBH_GetDeviceDescriptor() .................. 22
   USBH_GetEndpointDescriptor() ............... 24
   USBH_GetFrameNumber() ....................... 25
   USBH_GetInterfaceDescriptor() ............. 23
   USBH_GetInterfaceIDByHandle() .............. 26
   USBH_GetInterfaceID() ....................... 19
   USBH_GetInterfaceInfo() ...................... 20
   USBH_GetSerialNumber  ......................... 24
   USBH_GetSpeed() .................................. 25
   USBH_GetStatusStr()  ......................... 28
   USBH_Init() .................................. 17
   USBH_OpenInterface() ......................... 21
   USBH_RegisterEnumErrorNotification() ........ 21
   USBH_RegisterPnPNotification() .............. 20
   USBH_RestartEnumError() ..................... 21
   USBH_SetTraceMask()  ......................... 27
   USBH_SubmitUrb() .................................. 26
   USBH_UnregisterEnumErrorNotification() .... 21
   USBH_UnregisterPnPNotification() ........... 20

D
disclaimer ............................................. 2
document conventions .......................... 7

E
dition, of this guide ............................... 2
emUSB-Host
   Features ....................................... 9

guidelines, reading .................................. 7

M
MSD error codes
   USBH_MSD_STATUS_COMMAND_FAILED ............... 49
   USBH_MSD_STATUS_ERROR  .......................... 49
   USBH_MSD_STATUS_INTERFACE_PROTOCOL ........ 49
   USBH_MSD_STATUS_INTERFACE_SUB_CLASS ....... 49
   USBH_MSD_STATUS_LENGTH  ...................... 49
   USBH_MSD_STATUS_PARAMETER ................... 49
   USBH_MSD_STATUS_PIPE_STALLED ............... 49
   USBH_MSD_STATUS_SENSE_REPEAT ............... 49
   USBH_MSD_STATUS_SENSE_STOP ................. 49
   USBH_MSD_STATUS_SUCCESS ...................... 49
   USBH_MSD_STATUS_TIMEOUT ...................... 49
   USBH_MSD_STATUS_TRANSMISSION ............... 49
   USBH_MSD_STATUS_WRITE_PROTECT .............. 50
MSD functions
   USBH_MSD_GetLuns() .......................... 45, 52
   USBH_MSD_GetStatus() ......................... 47
   USBH_MSD_GetUnitInfo() ....................... 46
   USBH_MSD_Init() .......................... 44, 51–52
   USBH_MSD_ReadSectors() ....................... 45
   USBH_MSD_UNIT_INFO ......................... 47–48, 53
   USBH_MSD_WriteSectors() ..................... 46

O
OS integration
   API functions .................................. 61

P
part number, of this guide .......................... 2
publication date, of this guide .......................... 2