

# USB Multi-Interface Driver

## Reference Manual

Version: 1.16.0  
Date: 09 June 2011

Authors: Guenter Hildebrandt

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

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

<http://www.thesycon.de>



---

Copyright (c) 2005-2009 by Thesycon Systemsoftware & Consulting GmbH  
All Rights Reserved

## **Disclaimer**

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

## **Trademarks**

The following trade names are referenced throughout this manual:

Microsoft, Windows, Win32, Windows NT, Windows XP, Windows Vista, Windows 7 and Visual C++ are either trademarks or registered trademarks of Microsoft Corporation.

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



---

## Contents

<b>Table of contents</b>	<b>6</b>
<b>1 Introduction</b>	<b>9</b>
<b>2 Overview</b>	<b>11</b>
2.1 Platforms . . . . .	11
2.2 Features . . . . .	12
2.3 USB 2.0 and 3.0 Support . . . . .	13
<b>3 Architecture</b>	<b>15</b>
3.1 Location . . . . .	15
3.2 Features . . . . .	15
3.3 Use Cases . . . . .	16
<b>4 Customization</b>	<b>17</b>
4.1 Customization Overview . . . . .	17
4.2 Customization Steps . . . . .	18
4.3 Customizing thusbmi.inf . . . . .	19
4.3.1 Configuration of Names . . . . .	19
4.3.2 Configuration of Hardware ID . . . . .	20
4.3.3 Configuration of Software Interface Identifiers . . . . .	20
4.3.4 Update of the Driver Version . . . . .	21
4.3.5 Customizing Default Driver Settings . . . . .	21
4.4 Customizing Version Resources . . . . .	22
4.5 Digital Signature for Windows Vista . . . . .	22
4.5.1 Get an Authenticode Digital ID . . . . .	23
4.5.2 Get the Tools for Code Signing . . . . .	24
4.5.3 Create a Signature . . . . .	24
4.5.4 Modified System Behavior . . . . .	25
4.6 Installing USBMI Manually . . . . .	25
4.7 Uninstalling USBMI manually . . . . .	26
4.8 Creating a Driver Setup Package . . . . .	27
4.9 Installing the Driver Package at the Customer PC . . . . .	28
4.9.1 Without Pre-Installation . . . . .	28
4.9.2 With Pre-Installation . . . . .	28

4.10	WHQL Certification . . . . .	29
<b>5</b>	<b>Programming Interface Reference</b>	<b>30</b>
5.1	IO Control Codes . . . . .	30
	IOCTL_USBMICTRL_GET_DRIVER_INFO . . . . .	30
	IOCTL_USBMICTRL_GET_ACTIVE_CONFIGURATION . . . . .	31
	IOCTL_USBMICTRL_SET_ACTIVE_CONFIGURATION . . . . .	32
	IOCTL_USBMICTRL_GET_DESCRIPTOR . . . . .	33
	IOCTL_USBMICTRL_CLASS_OR_VENDOR_REQUEST . . . . .	34
	IOCTL_USBMICTRL_CYCLE_PORT . . . . .	35
	IOCTL_USBMICTRL_DISABLE_INTERFACE . . . . .	36
	IOCTL_USBMICTRL_ENABLE_INTERFACE . . . . .	37
	IOCTL_USBMICTRL_GET_INTERFACE_STATE . . . . .	38
	IOCTL_USBMICTRL_GET_HWID . . . . .	39
	IOCTL_USBMICTRL_SET_HWID . . . . .	40
5.2	Data Structures . . . . .	41
	GenericDriverInfo . . . . .	41
	UsbmictlConfig . . . . .	43
	UsbmictlConfInterface . . . . .	44
	UsbmictlHWID . . . . .	45
	UsbmictlDescriptorRequest . . . . .	46
	UsbmictlClassOrVendorRequest . . . . .	47
	UsbmictlInterface . . . . .	49
	UsbmictlInterfaceState . . . . .	50
5.3	Enumeration Types . . . . .	51
	UsbmictlRequestRecipient . . . . .	51
	UsbmictlRequestType . . . . .	52
<b>6</b>	<b>Debug Support</b>	<b>53</b>
6.1	Enable Debug Traces . . . . .	53
	<b>Index</b>	<b>55</b>

## References

- [1] Universal Serial Bus Specification 1.1,  
<http://www.usb.org>
- [2] Universal Serial Bus Specification 2.0,  
<http://www.usb.org>
- [3] USB device class specifications (Audio, HID, Printer, etc.),  
<http://www.usb.org>
- [4] Microsoft Developer Network (MSDN) Library,  
<http://msdn.microsoft.com/library/>
- [5] Windows Driver Development Kit,  
<http://msdn.microsoft.com/library/>
- [6] Windows Platform SDK,  
<http://msdn.microsoft.com/library/>



## 1 Introduction

The Multi-Interface (MI) driver is an replacement driver for the build in USB MI driver for Windows. This driver has enhanced features and fixes some problems of the build in USB MI driver. It handles a larger set of device classes and the Interface Associated Descriptor (IAD) in a correct way. It supports devices with a set of USB configurations (Multi-Configuration Devices). A proprietary API is used to switch between the configuration, submit endpoint 0 related requests and enable or disable single interfaces.

It supports a concept to create CD-less installations where the PC software is stored inside the device and it is installed automatically when the device is connected the first time to a PC. It allows the dynamically switching between different functional kernel drivers without re-installing the drivers.

This document describes the driver and the API.



## 2 Overview

### 2.1 Platforms

The USB MI driver driver supports the following operating system platforms:

- Windows 7
- Windows Vista
- Windows XP
- Windows 2000
- Windows Embedded Standard 7 (WES7)
- Windows Embedded Enterprise
- Windows Embedded POSReady
- Windows Embedded Server
- Windows XP embedded
- Windows Server 2008 R2
- Windows Server 2008
- Windows Server 2003
- Windows Home Server

The driver package contains 32 bit and 64 bit versions depend on operating system.

## 2.2 Features

The USB MI driver driver provides the following features:

- **USB Support.** The USB MI driver supports USB 2.0 full and high speed and USB 1.1.
- **Compatibility.** The USB MI driver is compatible to the Microsoft's Multi-Interface driver.
- **Plug&Play.** The driver fully supports hot plug and play.
- **Power Management.** The driver supports the Windows power management model.
- **Multiple USB Configurations.** The USB MI driver can be used with devices that implement multiple USB configurations. It supports switching between different USB configurations.
- **Multiple USB Interfaces.** The USB MI driver can be used with devices that implement multiple USB interfaces.
- **Multiple USB Devices.** Multiple USB devices can be controlled by the driver at the same time.
- **Interface Association Descriptor.** The use of Interface Association Descriptors is supported by USB MI driver.
- **Programming interface.** The USB MI driver provides a Win32 programming interface for use in C and C++ programs.
- **WHQL Certification.** The driver conforms to Microsoft's Windows Driver Model (WDM) and it can be certified by Windows Hardware Quality Labs (WHQL) for all current 32-bit and 64-bit operating systems.

### 2.3 USB 2.0 and 3.0 Support

The USB MI driver device driver supports USB 2.0 and the Enhanced Host Controller on Windows 2000, Windows XP, Windows Vista and W7. However, USB MI driver has to be used on top of the driver stack that is provided by Microsoft. Thesycon does not guarantee that the USB MI driver driver works in conjunction with a USB driver stack that is provided by a third party. For instance, third-party drivers are available for USB 2.0 host controllers from NEC, INTEL or VIA. Because the Enhanced Host Controller hardware interface is standardized (EHCI specification) the USB 2.0 drivers provided by Microsoft can be used with host controllers from any vendor. However, the user has to ensure that these drivers are installed.

Currently the first Extended Host Controllers are on the market. Microsoft did not release a bus driver with Windows 7 for this host controller type. The Extended Host Controller handles also full and high speed data traffic. If a customer connects your multiinterface device to a blue USB connector it runs with an Extended Host Controller and the related vendor provided bus driver. In the market there are different host controller drivers. Thesycon is not able to test the USB MI driver with all possible host drivers that are available on the market. For that reason we cannot give any warranty that the driver works with such host controllers.



## 3 Architecture

### 3.1 Location

The Multi-Interface device driver is a kernel mode driver for Windows. It is used for USB devices with multiple interfaces and/or multiple configurations. It is loaded on the USB Bus Driver interface USBDB. The upper interface of the driver is the same as the interface of the USBDB. Drivers on top of the MI driver work in the same way as a driver on top of the USBDB driver.

The driver has additional features like interface control or EP0 access. These additional features can be also used if the driver is installed on a USB device with a single USB interface. With the enhanced features this driver is not a simple replacement of the system provided MI driver.

### 3.2 Features

The driver has the following additional features:

- Build in support for a lot of classes, like CDC, Video, Audio, HID, Printer, Mass Storage, DFU, and others
- Full support for the Interface Associated Descriptors (IAD)
- User mode API to access the driver during the runtime
- Support for Multi-Configuration devices. The default configuration can be selected with INF file parameters. The configuration can be switched with the user mode API.
- Remote Wakeup support. If one child driver requests a remote wakeup the driver enables the device for wake up.
- The Hardware IDs for the child driver stacks can be generated in the same way as the Microsoft driver do it or they can be created with a different prefix.
- Access to EP0 requests, like Class or Vendor related requests, Get Descriptor requests and Cycle Port.
- The child driver stacks can be disabled and enabled with the private API.
- Possibility to customize the driver to avoid conflicts with other vendors that are using the same driver.
- The hardware ID of the child devices can be switched with the API. This enables the dynamic switching of functional child drivers with a software API and without re-installing the drivers.
- The switching between configurations supports a concept to create devices that store the required PC software on a mass storage partition. The software can be installed automatically when the device is connected the first time to a PC. The Multi-Interface driver switches the device than to the operational mode. This method is also known as CD-less installation.

### 3.3 Use Cases

The additional features of the MI driver are use full in a lot of cases.

E.g. to switch a device between normal operational mode and DFU mode a special request must be sent to the device. If all interfaces of the device are class compliant and a class driver from Windows are used there is no way to submit the request. The MI driver allows to send the request on the private API without modifying the existing drivers.

If the device has some configuration parameters that should be maintained by the PC, the private interface of the driver can be used to transfer these information to the device.

The Multi-Interface driver from Microsoft uses always the first USB configuration of the device. This MI driver can be used to switch between different configurations. This can be useful if the device cannot expose all required interfaces at the same time because of a limited number of endpoints or if the device should behave completely different in some situations.

This feature is useful for the device installation. The device may expose a USB Mass Storage interface on the first USB configuration. This volume may contain the PC software and a setup program. If the device is connected to a PC where the device software is not installed the volume becomes visible and the software is installed. If the MI driver is loaded on the device the private API can be used to switch to the second configuration that can expose the operational interfaces.

The private API can be used to identify a device instance by reading the serial number of the device with a string descriptor. Other device related information can be extracted from USB descriptors.

Some or all interfaces of the device can be disabled by the private API. This may be useful to save some band width. E.g. the Windows provided RNDIS driver has pending IN requests all the time it is running. These requests block some band width even if no data is transferred.

To disable a interfaces causes the higher driver stack to be unloaded. This may be useful to restart some driver stacks without administrator privileges or to create Plug and Play notifications.

The hardware ID of the child devices can be set by the user mode API. By setting different ID's the functional drivers can be switched in a fast an reliable way. No administrator privileges are required to perform the switch. This may be useful to switch a Windows CE based device between the Active Sync driver and a CDC ACM driver that provides a virtual COM port interface.

## 4 Customization

### 4.1 Customization Overview

The USB MI device driver supports various features that enable you to create a customized device driver package to be shipped with an end product. Customization includes:

- Modification of the file name of the driver executable,
- Modification of text strings shown at the Windows user interface,
- Definition of a unique software interface identifier,
- Adaptation of driver behavior for a specific device.

Note that the driver package which is shipped to end users should always be customized. This is required in order to avoid potential conflicts with other products of other vendors that are also using the USB MI device driver. Please consider the following example scenario: An end user buys product A which includes the USB MI device driver version 2.00. The user installs this driver on his machine. At a later point in time the user buys another product of another vendor which is called B. Product B includes the USB MI device driver version 2.30. When the user installs the device driver for Product B on his machine then a conflict will occur because another version of the driver is already installed on that machine.

There are several problems that may result from this conflict situation:

- **Driver version conflict**

We assume that during driver installation any existing device driver will be removed if the existing driver has an older version than the driver to be installed. If the existing driver is newer then no driver will be installed. In the example scenario described above this means: When product B is installed the existing driver (V2.00) will be replaced by a newer one (V2.30). The result is that both product A and product B now use the new driver. This should be fine for product B. However, product A will now run with driver version 2.30 which is critical because probably the product was never tested with that version. Thus, installation of a new product can break an existing installation of another product.

- **Driver software interface ambiguity**

If two or more different products (of different vendors) use the same driver then a conflict can arise when Windows applications open the device driver to communicate with the device. In the example scenario described above an application designed for product A could inadvertently open device B and try to configure the hardware which will probably not work.

- **Device naming ambiguity**

If two or more different products (of different vendors) use the same driver then a device name conflict could occur. Particularly, this applies to device names displayed in Device Manager. In the example scenario described above an end user could get confused if the item shown in Device Manager for product A is named identically to the item shown for product B.

The customization features of the USB MI driver enable you to avoid all of these conflict situations. A customized driver can be considered as a specific driver for a specific product. There

will be no overlaps with (customized) USB MI drivers shipped with other products of other vendors. In the example scenario described above, if both product A and product B are shipped with a customized driver then there will be two separate driver installations on the end user's machine. There will be two sets of driver files on hard disk and two separate drivers loaded into memory.

Note that it is possible to create a customized driver package which supports several products with similar properties, e.g. a product family of a vendor. In this case, if several products of the family are used on one machine, there will be only one set of driver files on hard disk and only one driver executable loaded into memory. A vendor can decide which of its products will be supported by a particular driver package and how many different driver packages need to be created.

To summarize the customization strategy the following rules are given:

1. A driver package provided to end users should always contain a customized driver. Do not ship the original driver provided by Thesycon together with your products.
2. If you offer a family of products, you may create a customized driver package that supports all products of this family. Windows applications shipped with the driver should be designed in such a way that all products of the family are supported. If an updated driver is delivered to end users, either as a software-only package or as part of a new product of the family, then you have to ensure that the new driver version works with all released products of the family.

In the following sections, the customization procedure is described in detail.

## 4.2 Customization Steps

Below, the steps needed to create a customized driver package are summarized. Some of these steps are required and some are optional.

- **Required:** Choose a new name for the driver. The driver package consists of two parts:
  - Windows 2000, Windows XP, Windows Server 2003, Windows Vista and Windows 7 32 bit: thusbmi.sys and thusbmi.inf
  - Windows XP, Windows 2003, Windows Vista and Windows 7 x64 Edition: thusbmi\_x64.sys and thusbmi\_x64.inf

The new names must not contain spaces and must not cause conflicts with drivers included with Windows. Make a private copy of the sys and inf files and rename all files to your new names. Edit the renamed .inf file to contain the new driver name. See section 4.3.1 on page 19 for detailed information.

- **Required:** Edit your .inf file to contain the correct hardware ID for your device. Refer to section 4.3.2 on page 20 for more information.
- **Required:** Create a private interface identifier (GUID). Specify that GUID in your .inf file and use that GUID in your applications to enumerate and open devices. See section 4.3.3 on page 20 for detailed information.
- **Optional:** Adapt default driver settings. Refer to section 4.3.5 on page 21 for more information.

- **Optional:** Edit the version resources contained in `thusbmi.sys`. See section 4.4 on page 22 for more information.

### 4.3 Customizing `thusbmi.inf`

The `thusbmi.inf` file is used to install the kernel-mode device driver `thusbmi.sys`. It has to conform to INF file conventions defined for WDM drivers. Refer to the Windows DDK documentation [5] for more information about INF files.

The `thusbmi.inf` file itself can be renamed to any name of your choice. However, the file name extension has to be `.inf`.

#### 4.3.1 Configuration of Names

In `thusbmi.inf` there is a Strings section that permits to define the driver name and some text strings that will be shown at the Windows user interface.

```
[Strings]
S_Provider="Thesycon"
S_Mfg="Thesycon"
S_DiskName="thusbmi driver disk"
S_DeviceDesc="USB Multi-Interface Driver"
S_ServiceName="thusbmi"
S_DriverName="thusbmi"
```

##### **S\_Provider**

This variable defines the provider of the driver. You should use your company's name here.

##### **S\_Mfg**

This variable defines the manufacturer of the driver. You should use your company's name here.

##### **S\_DeviceDesc**

This variable defines the device description which is shown during installation. When driver installation is finished the device description is shown in Device Manager next to the item representing your device. You should use your product's name here.

##### **S\_DiskName**

This variable defines the name of your installation disk. It should correspond to the label that is printed on the disk. The disk name is displayed by the operating system when it requests the user to insert the installation media.

##### **S\_ServiceName**

This variable defines the name of the service that is used to install the driver. Do not use spaces or special characters here. Typically the name of the driver is used. The service name must be unique on any PC.

##### **S\_DriverName**

This variable defines the name of the `thusbmi` driver executable which is `thusbmi.sys` by default. Note that the name is given without the `.sys` extension. You have to set this value to the file name

you want to use for `thusbmi.sys`. It is strongly recommended to use a vendor-specific driver name in order to avoid file naming conflicts with other drivers.

**Important:** The driver name must not contain spaces. If you modify `S_DriverName` then you have to modify the following sections as well:

```
[_CopyFiles_sys], [SourceDisksFiles].
```

Due to technical limitations, these sections cannot use the `S_DriverName` variable to specify the driver executable. Therefore, you have to edit them as well. Do a search for `thusbmi` to locate the lines to be modified.

### 4.3.2 Configuration of Hardware ID

Your USB device is initially enumerated by the system-provided USB bus driver. The bus driver uses vendor and product ID's from the device descriptor to create a unique hardware ID string according to the following scheme:

```
USB\VID_XXXX&PID_YYYY
```

The fields `XXXX` and `YYYY` will be replaced by the hexadecimal form of the vendor ID and the product ID. You can check the resulting hardware IDs by looking at the following registry key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB
```

The `thusbmi.inf` file needs to specify the resulting hardware ID for the device within the following section:

```
[_Models]
%S_DeviceDesc1%=_Install1, USB\VID_XXXX&PID_YYYY
```

Replace `XXXX` and `YYYY` by your specific values, for example:

```
[_Models]
%S_DeviceDesc1%=_Install1, USB\VID_152A&PID_0001
```

Windows can distinguish between different device release numbers. The device release number is part of the device descriptor in the field `bcdDevice`. If the INF file should work with one special device release number of a device the hardware ID can be specified in the following way:

```
[_Models]
%S_DeviceDesc1%=_Install1, USB\VID_152A&PID_0001&REV_ZZZZ
```

The parameter `ZZZZ` is the hexadecimal value of the `bcdDevice` value.

### 4.3.3 Configuration of Software Interface Identifiers

The USB MI driver creates a device instance for each of your USB devices connected to the system. A device instance exports the software interface which can be used to access the interface of the USB MI device driver. The software interface is unambiguously identified by a globally unique identifier (GUID). The USB MI driver allows you to define a private GUID that is used by your applications to enumerate and access your devices. The following section in the `thusbmi.inf` file is used to define a private interface GUID.

```
[_AddReg_HW]
;HKR,,DriverUserInterfaceGuid,%REG_SZ%,"{0160E02E-EEF7-4fe7-A8DD-5F0FA42CE191}"
```

You have to use `guidgen.exe` to create a fresh GUID. This tool is provided with the Microsoft Platform SDK [6] or as part of the Visual Studio development platform. Copy and paste the GUID into the `thusbmi.inf` section shown above and use this GUID instead of the GUID defined in the file `usbmictrl.h` in the symbolic constant `USBMICTRL_IID`. If this entry is deleted from the INF file the private interface of the driver is not exposed and cannot be used. Use the generated private GUID in all of your Windows applications to open the device driver. Do not use the default GUID `USBMICTRL_IID` provided by Thesycon in `thusbmi.h`. This way, it is guaranteed that your applications can identify your devices unambiguously.

`Guidgen.exe` permits you to export a statement that defines a GUID constant, for example:

```
// {5510F365-363E-407b-80A5-C663533E93B5}
static const GUID MyPrivateGUID =
{ 0x5510f365, 0x363e, 0x407b, { 0x80, 0xa5, 0xc6, 0x63, 0x53, 0x3e, 0x93, 0xb5 } };
```

Copy and paste this statement to the source code of your application(s) and use the GUID constant to enumerate and open devices. Note that you cannot use the GUID that is shown in the example above. You have to use `guidgen.exe` to create a new one.

#### 4.3.4 Update of the Driver Version

The key word `DriverVer` is used in the Version section or in the DD Install section of the INF file. It contains the version of the driver and the release date. The release date should be modified to the release date of the customized version. Please take care of the right date format.

#### 4.3.5 Customizing Default Driver Settings

The `.inf` file specifies some settings that define the default behavior of the driver. These settings are defined in the following section.

```
[_AddReg_HW]
HKR,,UsbConfiguration,%REG_DWORD%,0
HKR,,CrtlTimeout,%REG_DWORD%,1000
HKR,,PnpIdPrefix,%REG_SZ%,"MI"
```

##### **UsbConfiguration**

This parameter contains the index of the USB configuration that should be selected if the driver is loaded. If the device has only one configuration this parameter must be set to 0. This parameter is also used to store the currently selected configuration if it is switched with the interface of the driver.

##### **CrtlTimeout**

This parameter contains the timeout for descriptor and class or vendor requests in milliseconds. A control request should be completed with in 500 ms.

##### **PnpIdPrefix**

This parameter contains the prefix that is used to add the interface number to the PnP ID. The resulting PnP ID is:

```
usb\vid_VVVV&pid_PPPP&<PnpIdPrefix>_II
```

where **VVVV** is the hexadecimal representation of the vendor ID, **PPPP** is the hexadecimal representation of the product ID, `<PnpIdPrefix>` is the prefix defined with this key and **II** is the hexadecimal representation of the first USB interface number that belongs to the interface association. This PnP ID must be used in the INF files of the drivers that are installed on top of the MI device driver. The MI driver that is provided with the system uses the prefix "MI". To generate a compatible behavior the same prefix can be used.

### 4.4 Customizing Version Resources

The `thusbmi.sys` executable includes version resources. These information will be shown in Device Manager on the Driver Details dialog page or on the file's property page. You may want to modify the version resources to include your product's name or to modify copyright information. This can be done in a two-step procedure as described below.

1. Make a private copy of `thusbmi.sys`. Use Visual Studio to open the copy of `thusbmi.sys` in resource mode. You have to select Open as Resources in the File Open dialog. Edit the version resources according to your preferences and save the modified file.
2. Open a Command Prompt window and run the `UpdateChecksum.exe` tool on the modified file. You have to enter the following command line:

```
UpdateChecksum thusbmi.sys
```

The program `UpdateChecksum.exe` is part of the USB MI development kit.

### 4.5 Digital Signature for Windows Vista

Windows Vista has a new feature to verify a vendor of a software component. The vendor can add a signature to a software component to identify itself. This signature grants that the software was signed by the vendor and that the software was not modified after it was signed. If a signed plug and play driver is installed the first time Windows Vista shows the vendor name and the user can choose if he wants to

- Abort the installation,
- Allow the installation this time or
- Always trust the vendor and allow installations from this vendor.

If the user selects "always trust this vendor" the certificate of the vendor is stored in the certification manager under Trusted Vendors. If later a plug and play driver of this vendor is installed again, the installation can be performed silently without user interactions if the driver was pre-installed before the device is connected.

On Windows Vista x64 Edition it is required that the driver has a digital signature. Otherwise, the driver is not loaded on a normal system. To test a driver without signature on Windows Vista x64 the system can run with a kernel debugger. In this case the system loads a driver without signature. On Windows Vista 32 bit a driver without signature can be installed.

The signature of a driver is not the same as a WHQL certification. A driver can have a digital vendor signature without passing any WHQL tests.

A signature for a plug and play driver is always part of a .cat file. It is possible to add a signature to the .sys file, but it is accepted in Test Mode only. If the system runs in normal mode the signature is expected on the .cat file referenced by the .inf file.

Why can Thesycon not deliver a signed driver package? The signature becomes invalid if the driver or the INF file is modified. To install the driver on any device a customized INF file must be generated. At least the USB vendor and product ID's must match the ID's of the device. This modifications would make a signature from Thesycon invalid.

Why Thesycon cannot deliver all the tools required to create the signature? This is not possible because Microsoft does not allow re-distribution of the signing tools.

Why the Vendor that uses the USB MI driver should sign the driver and not Thesycon? The aim of the customization is to let the driver look like a driver that is developed by the vendor of the device. To direct support issues to the vendor and not to Thesycon it is required that the vendor of the device performs the signing.

The following sections will guide you through the process of obtaining an "Authenticode Digital ID", to maintain it, to get the tools required for signing and to create a signature for a plug and play driver. It is strongly recommended to follow these steps. It is not possible to perform the code signing on a Windows 2000 system. Windows XP or better is required.

To get more information about code signing, read the document "Kernel-Mode Code Signing Walkthrough" available on the Microsoft web site.

#### **4.5.1 Get an Authenticode Digital ID**

To make sure that the owner of a digital signature key is the entity that is described with this key a so called certification authority (CA) guarantees that this information is valid. Microsoft accepts a number of CA's for the Authenticode technology. The company VerySign is one of them.

At first you have to buy a signature key pair from one of the CA's. The key is valid for a given time interval. You can select the time interval during the order of the key pair. The time interval means that the key can be used in this interval to create new signatures. A signature that was created with a time stamp is valid after the key has expired.

The signature key delivered by VerySign typically consists of three parts:

- a certificate and a public key stored in a .spc file,
- a private key stored in a .pvk file and
- a password that protects the private key.

You may read the Microsoft document "Code-Signing Best Practices" to get more information how to maintain the key.

The key must be stored in the certificate manager to use it later. To do this, it must be first translated to a personal information exchange (.pfx) file format. This can be performed by the program PVK2PFX that is part of the SDK and the WDK. This program is a command line program. Please use the following command line to create the .pfx file:

```
pvk2pfx -pvk <pvk file name>.pvk -spc <spc file name>.spc  
-pi <password> -pfx <pfx file name>.pfx
```

To import the .pfx file to the certificate manager, perform a double click to the file. This starts the Certificate Import Wizard. Enter the password and make sure the certificate is stored in the Personal certificate store.

### 4.5.2 Get the Tools for Code Signing

To create a .cat file the program inf2cat.exe is required. This file is part of the "Winqual submission tools". Please download this package from the Microsoft web site and extract the file. The inf2cat.exe program requires the DLL's that starts with "Microsoft.Whos..." and that are part of the package.

The signtool.exe is required to attach the signature to the .cat file. This program exists in different versions with the same file name. Versions that are delivered with the .net framework and the SDK cannot be used to sign kernel mode drivers. Only the version that is part of the WDK is sufficient to create a signature for a kernel mode driver. So you have to install the WDK. You will find the signtool program under bin\SelfSign.

Kernel mode driver signing requires a cross certificate. This cross certificate must be downloaded at the Microsoft Web site "Microsoft Cross-certificates for Windows Vista Kernel Mode Code Signing". Please select the correct cross certificate for the CA where you have bought code signing ID.

Do not use a shared network folder to store this executables. The programs do not run correctly on a shared network folder.

### 4.5.3 Create a Signature

It is recommended to put all required tools in one folder and to add this location to the PATH variable. Each time the .inf or .sys file is modified the signature must be created again.

Create separate folders for the 32 bit and the 64 bit .inf and .sys files. The inf2cat program searches all files in the folder and cannot handle the 32 bit files in 64 bit mode correctly. To create the INF file for 32 bit files execute the following command line:

```
inf2cat.exe /driver:<folder with 32 bit files>  
/os:2000,XP_X86,Server2003_X86,Vista_X86
```

To create the cat file for the 64 bit files run:

```
inf2cat.exe /driver:<folder with 64 bit files>  
/os:XP_X64,Server2003_X64,Vista_X64
```

Finally the signature can be created. Run the following command line:

```
signtool sign /v /n <Certificate Name>
```

```
/ac <cross certificate with path>.cer  
/t http://timestamp.verisign.com/scripts/timestamp.dll  
<cat file with path>.cat
```

This program requires access to the Internet to get the time stamp from verisign.com. The Certificate Name can be displayed with the certificate manager. To open the certificate manager enter `certmgr.msc` on a command line. To show the signature of the .cat file use the property page. To verify the signature use the command line:

```
signtool verify /pa <path and name of the catalog file>.cat
```

#### 4.5.4 Modified System Behavior

If the driver package has a valid signature from a vendor the system behavior is modified in the following way:

On Windows 2000 and Windows XP the property page of the device manager shows that the driver does not have a valid signature. Both operating systems cannot validate the signature because the trusted chain of the certificates is not completely stored in the operating system. Both systems accept only a signature created by Microsoft during a WHQL certification process.

On Windows Vista and Vista x64 during the driver installation a message shows the name of the vendor that has signed the driver package. The user can select if the driver package should be installed or if the installation should be aborted. If the certificate of the vendor is stored in the Trusted Vendors section of the certificate manager and if the driver is pre-installed the driver is installed silently on Vista.

**Installation** This section discusses topics relating to the installation and un-installation of the USB Multi-Interface device driver.

## 4.6 Installing USBMI Manually

In order to install the USBMI driver manually you have to prepare an INF file that matches your device. Refer to section 4 on page 17 for more information.

The steps required to install the driver are described below.

- Connect your USB device to the system. After the device has been plugged in, Windows launches the New Hardware Wizard and prompts you for a device driver. Provide the New Hardware Wizard with the location of your installation files (e.g. `thusbmi.inf` and `thusbmi.sys` for Windows 2000/XP). Complete the wizard by following the instructions shown on screen. If the INF file matches your device, the driver should be installed successfully.

Note that, on Windows 2000, Windows XP and Windows Vista, the New Hardware Wizard shows a warning message that complains about the fact that the driver is not certified and digitally signed. You may ignore this warning and continue with driver installation. The USBMI driver is not certified because it is not an end-user product. When the USBMI driver is integrated into such a product, it is possible to get a certification and a digital signature from the Windows Hardware Quality Labs (WHQL).

Note that, on Windows Vista x64 the driver cannot be loaded without a digital signature. Please see section 4.5 on page 22 for more details.

- If the operating system contains a driver that is suitable for your device, the system does not launch the New Hardware Wizard after the device is plugged in. Instead of that a system-provided device driver will be installed silently. The operating system does not ask for a driver because it finds a matching entry for the device in its internal INF file data base.

This is the case for all devices with multiple USB interfaces. The installation of the system provided Multi-Interface driver can be suppressed, if the device reports the class code for the communication device class CDC (2) in the class field of the device descriptor.

You have to use the Device Manager to install the USBMI driver for a device for which a driver is already running. To start the Device Manager, right-click on the "My Computer" icon and choose Properties. In the Device Manager, right-click on your device and choose Properties. On the property page that pops up choose Driver and click the button labeled "Update Driver". The Upgrade Device Driver Wizard is started which is similar to the New Hardware Wizard described above. Provide the wizard with the location of your installation files (thusbmi.inf and thusbmi.sys) and complete driver installation by following the instructions shown on screen.

- The device manager does not allow to change the device class of a device. In the case that the INF file contains a different class as the class where the device is currently installed the device manager reports that your INF file does not contain matching information for your device even if the hardware ID matches the device. In this case you have to delete the existing driver in the device manager first.
- After the driver installation has been successfully completed your device should be shown in the Device Manager in the USB section.
- If the USBMI driver is installed successful it creates device nodes for each interface bundle of the connected device. The number of device nodes depends on the interface structure of the device. The system launches the hardware wizard again for each device node. To avoid a lot pop ups of the hardware wizard it is recommended to get a WHQL certification for the drivers. This enables a silent installation, if the driver is pre-installed.

### 4.7 Uninstalling USBMI manually

To uninstall the USBMI device driver for a given device, use the Device Manager. The Device Manager can be accessed by right-clicking the "My Computer" icon on the desktop, choosing "Properties" from the context menu and then opening the Device Manager window. Within the Device Manager window, double-click on the entry for the device and choose the property page labeled "Driver". There are two options to uninstall the USBMI device driver:

- Remove the selected device from the system by clicking the button "Uninstall". The operating system will re-install a driver the next time the device is connected or the system is rebooted.
- Install a new driver for the selected device by clicking the button "Update Driver". The operating system launches the Upgrade Device Driver Wizard which searches for driver files or lets you select a driver.

If the USBMI driver is uninstalled all drivers that are installed on the child device nodes disappear. In order to uninstall the device drivers for the child device nodes these driver must ne uninstalled first.

In order to avoid automatic and silent re-installation of USBMI by the operating system, it is necessary to manually remove the INF file used to install the USBMI driver.

During driver installation, Windows stores a copy of the INF file in its internal INF file data base located in %WINDIR%\INF\. The name of the INF file is changed before it is stored in the database. On Windows 2000 and Windows XP the INF file is stored as oemX.inf, where X is a decimal number.

The best way to find the correct INF file is to do a search for some significant string in all the INF files in the directory %WINDIR%\INF\ and its subdirectories. Note that on Windows 2000/XP/2003, by default the %WINDIR%\INF\ directory has the attribute Hidden. Therefore, by default the directory is not shown in Windows Explorer.

Once you have located the INF file, delete it. This will prevent Windows from reinstalling the USBMI driver. Instead, the New Hardware Wizard will be launched and you will be asked for a driver.

On Windows Vista the driver and the INF files are stored in the driver store. During uninstallation a check box can be selected to remove the driver form the driver store. If this box is checked the driver is also removed from the INF folder.

## 4.8 Creating a Driver Setup Package

A Setup Information File (INF) is required for proper installation of the USBMI device driver. This file describes the driver to be installed and defines the operations to be performed during the installation process.

The USBMI driver package consists of 2 INF files and 2 SYS files. See section 4 on page 17 for details.

An INF file is in ASCII text format. It can be viewed and modified with any text editor, for example Notepad.exe. The contents and the syntax of an INF file are documented in the Microsoft Windows DDK. For instructions on how to create a device-specific INF file, refer to chapter 4 on page 17.

The INF file is loaded and interpreted by a software component called Device Installer that is built into the operating system. The Device Installer is closely related to the Plug&Play Manager that handles connection and removal of USB devices. After the Plug&Play Manager has detected a new USB device, the system searches its internal INF file database, located in %WINDIR%\INF\, for a matching driver. If no driver can be found, the New Hardware Wizard pops up and asks the user for a driver.

The association of device and driver is based on a string called Hardware ID. The Plug&Play Manager builds the Hardware ID string from the USB descriptors. The string is prefixed by the bus identifier USB. An example for a Hardware ID string is:

```
USB\VID_152A&PID_0001
```

In order to prepare an installation disk that can be used to install the USBMI driver for your device,

the following steps are required.

- Copy the customized USBMI driver binary SYS to a floppy disk or to a directory location of your choice. Copy the customized INF file to the same location.

Provide the files, the .sys and the .inf files, on your installation media (floppy disk or CD-ROM).

### 4.9 Installing the Driver Package at the Customer PC

It is not recommended to copy the INF files and the SYS files manually to any system folder. Follow one of the next sections to perform your installation.

#### 4.9.1 Without Pre-Installation

The user connects the device to the PC. The system does not provide a matching device driver. The hardware wizard is launched. The user should guide the hardware wizard to your installation medium with the customized INF and SYS files.

This method has some drawbacks:

- The user must select the advanced driver installation to guide the wizard to the installation medium. Otherwise no driver is installed.
- The system requests the driver disk again if a device without USB serial number is connected to a different USB port or a device with a new serial number is connected to the PC.

It is not recommended to use this method.

#### 4.9.2 With Pre-Installation

Perform the following steps during the software setup:

- Copy the INF files and SYS files to a folder on the hard disk. On a x64 system copy the x64 .INF and .SYS files and on a 32 bit system copy the 32 bit files. The storage should be permanent. It should not be a system folder under the Windows tree. Typically the folder is `Program Files\<Company>\<Program>\<Version>\Drivers`.
- Call the system function `SetupCopyOEMInf ( )`. It requires administrator privileges. On a x64 Edition of the operating system this function must be called in a 64 bit process context. If the driver is not certified a warning may appear.
- Ask the user to connect the device to the PC. If the driver has a WHQL certification the installation is performed silent. Otherwise the hardware wizard is launched. The user can process the wizard by pressing the Next button all the time. A warning that the driver is not digitally signed may appear a second time.
- To update an existing driver call the system function `UpdateDriverForPlugAndPlayDevices()`. On a x64 Edition of the operating system this function must be called in a 64 bit process context.

For more advanced driver installation and un-installation we recommend the Device Installation Toolkit (DIT) provided by Thesycon. A free demo version can be downloaded on the Thesycon home page.

#### **4.10 WHQL Certification**

The WHQL certificate is a special driver signature from Microsoft. To get this certificate the driver and the device must pass the DTM (Device Test Manager) test bench. The test bench must be performed by the customer and the test results must be submitted to Microsoft. Thesycon can offer support to perform the WHQL test.

The WHQL test requires some additional files. Thesycon can deliver these files on request.

## 5 Programming Interface Reference

The interface that is exposed by the driver is based on a GUID (global unique identifier). The complete interface is defined in the file `usbmictrl.h`. This file contains the GUID, the IO control codes and the required data structure. To see how to open a driver with a GUID based interface please refer to the SDK. This document describes the data structures and the IO control code and the operation of the driver.

The GUID in the header file `usbmictrl.h` and in the INF file are examples, only. During the customization a new GUID should be created. See section 4 for details. This interface can be disabled if the entry `HKR, ,DriverUserInterfaceGuid` in the INF file is deleted.

### 5.1 IO Control Codes

#### **IOCTL\_USBMICTRL\_GET\_DRIVER\_INFO**

This operation retrieves the version information from the driver.

**lpInBuffer**

Is NULL.

**nInBufferSize**

Is zero.

**lpOutBuffer**

Points to a caller-provided buffer that will receive the `GenericDriverInfo` structure.

**nOutBufferSize**

Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`.

#### *Comments*

The application can use this function to see if the driver version and the API version has the expected values.

#### *See Also*

[GenericDriverInfo](#) (page 41)

**IOCTL\_USBMICTRL\_GET\_ACTIVE\_CONFIGURATION**

This operation retrieves the active configuration index.

**lpInBuffer**

Is NULL.

**nInBufferSize**

Is zero.

**lpOutBuffer**

Points to a caller-provided buffer that will receive the `UsbmictrlConfig` structure.

**nOutBufferSize**

Specifies the size, in bytes, of the buffer pointed to by **lpOutBuffer**.

*Comments*

The returned value is the zero based index of the currently used configuration descriptor.

*See Also*

**UsbmictrlConfig** (page 43)

**IOCTL\_USBMICTRL\_SET\_ACTIVE\_CONFIGURATION**

This operation sets a new USB configuration on the device.

**lpInBuffer**

Points to a caller-provided **UsbmictlConfig** structure.

**nInBufferSize**

Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**. This parameter must be set to `sizeof(UsbmictlConfig)` for this operation.

**lpOutBuffer**

Is NULL.

**nOutBufferSize**

Is zero.

*Comments*

This operation compares the active configuration with the configuration requested in this device. If both values are equal nothing is done. If a different configuration is requested the driver disables all child devices, sets the device in un-configured state, sets the new configuration and creates new child devices regarding to the reported configuration descriptor. All child driver gets a PNP remove event and new instances are loaded. The upper software layer see a behavior that is similar to a remove of the device and a connect with the new configuration. The private API of the MI driver is not disabled during this process.

*See Also*

**UsbmictlConfig** (page 43)

**IOCTL\_USBMICTRL\_GET\_DESCRIPTOR**

This operation retrieves an USB descriptor from the device.

**lpInBuffer**

Points to a caller-provided **UsbmictrlDescriptorRequest** structure.

**nInBufferSize**

Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**. This parameter must be set to `sizeof(UsbmictrlDescriptorRequest)` for this operation.

**lpOutBuffer**

Points to a caller-provided buffer that will receive the descriptor. The descriptor can have different length. If the requested length is shorter than the complete descriptor the first part is returned.

**nOutBufferSize**

Specifies the size, in bytes, of the buffer pointed to by **lpOutBuffer**.

*Comments*

This operation sends a GetDescriptor request on the bus to the device. The operation may take some time before the device answers.

*See Also*

**IOCTL\_USBMICTRL\_GET\_DRIVER\_INFO** (page 30)

**IOCTL\_USBMICTRL\_CLASS\_OR\_VENDOR\_REQUEST**

This operation sends an USB class or vendor request to the device.

**lpInBuffer**

Points to a caller-provided **UsbmictrlClassOrVendorRequest** structure and in the case of an OUT request to the data memory. The data memory starts direct behind the data structure.

**nInBufferSize**

Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**. This parameter must be set at least to `sizeof ( IOCTL_USBMICTRL_CLASS_OR_VENDOR_REQUEST )` for this operation. In the case of an OUT request it must be set to `sizeof ( IOCTL_USBMICTRL_CLASS_OR_VENDOR_REQUEST ) + Data_Buffer_Size`.

**lpOutBuffer**

Points to a caller-provided buffer for a IN request. If the request is an OUT transfer this parameter can be set to NULL.

**nOutBufferSize**

Specifies the size, in bytes, of the buffer pointed to by **lpOutBuffer**.

*Comments*

This request enables an application to send class or vendor requests to the device. Note: It is not possible to send standard requests with this function. If an error occurs the application can send the next request without error handling. The USB bus driver makes sure that the control request are serialized.

*See Also*

**UsbmictrlClassOrVendorRequest** (page 47)

**IOCTL\_USBMICTRL\_CYCLE\_PORT**

This operation simulates a disconnect and a re-connect of the device.

**lpInBuffer**

Is set to NULL.

**nInBufferSize**

Is set to 0.

**lpOutBuffer**

Is set to NULL.

**nOutBufferSize**

Is set to 0

*Comments*

The driver and all child drivers are unloaded if this request has finished. All handles become invalid. This request can be used to switch the device in a different operation mode, e.g. to make a device firmware upgrade.

*See Also*

**[IOCTL\\_USBMICTRL\\_GET\\_DESCRIPTOR](#)** (page 33)

## **IOCTL\_USBMICTRL\_DISABLE\_INTERFACE**

This operation disables the child driver stack for one interface.

**lpInBuffer**

Points to a caller-provided **UsbmictrlInterface** structure.

**nInBufferSize**

Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**.

**lpOutBuffer**

Is set to NULL.

**nOutBufferSize**

Is set to 0

### *Comments*

This command can be used to disable the child driver stack for one interface. The interface number is the number of the first USB interface that belongs to the interface bundle. The modification is stored in the registry and is permanent until the interface is enabled with **IOCTL\_USBMICTRL\_ENABLE\_INTERFACE**. The driver stack on the interface is unloaded and no longer visible in the device manager. The interface cannot longer accessed by applications.

### *See Also*

**IOCTL\_USBMICTRL\_ENABLE\_INTERFACE** (page 37)

**UsbmictrlInterface** (page 49)

**IOCTL\_USBMICTRL\_ENABLE\_INTERFACE**

This operation enables the child driver stack for one interface.

**lpInBuffer**

Points to a caller-provided **UsbmictlInterface** structure.

**nInBufferSize**

Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**.

**lpOutBuffer**

Is set to NULL.

**nOutBufferSize**

Is set to 0

*Comments*

This command can be used to enable the child driver stack for one interface. The interface number is the number of the first USB interface that belongs to the interface bundle. The modification is stored in the registry and is permanent until the interface is enabled with **IOCTL\_USBMICTRL\_ENABLE\_INTERFACE**. If the interface was disabled before the child driver stack is loaded. The hardware wizard may pop up to install the required drivers for the child device stack, if the interface is enabled the first time.

*See Also*

**IOCTL\_USBMICTRL\_ENABLE\_INTERFACE** (page 37)

**UsbmictlInterface** (page 49)

## **IOCTL\_USBMICTRL\_GET\_INTERFACE\_STATE**

This operation enables the child driver stack for one interface.

**lpInBuffer**

Points to a caller-provided **UsbmictrlInterface** structure.

**nInBufferSize**

Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**.

**lpOutBuffer**

Points to a caller-provided **UsbmictrlInterfaceState** structure.

**nOutBufferSize**

Specifies the size, in bytes, of the buffer pointed to by **lpOutBuffer**.

### *Comments*

This command can be used to get the configured state of an interface. This command is related to the selected configuration.

### *See Also*

**IOCTL\_USBMICTRL\_ENABLE\_INTERFACE** (page 37)

**UsbmictrlInterface** (page 49)

**IOCTL\_USBMICTRL\_GET\_HWID**

This operation get the current HWID for a interface.

**lpInBuffer**

Points to a caller-provided **UsbmictrlInterface** structure.

**nInBufferSize**

Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**.

**lpOutBuffer**

Points to a caller-provided **UsbmictrlHWID** structure.

**nOutBufferSize**

Specifies the size, in bytes, of the buffer pointed to by **lpOutBuffer**.

*Comments*

This command can be used to get the hardware ID of an configuration and an interface. It returns either the stored value for the registry or it generates the string for this interface. The function does not check whether the device has the requested configuration and interface. The return value is not defined if the IOCTL returns with an error code.

*See Also*

**IOCTL\_USBMICTRL\_SET\_HWID** (page 40)

**UsbmictrlInterface** (page 49) **UsbmictrlHWID** (page 45)

## **IOCTL\_USBMICTRL\_SET\_HWID**

This operation set the HWID for a interface.

**lpInBuffer**

Points to a caller-provided **UsbmictrlHWID** structure.

**nInBufferSize**

Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**.

**lpOutBuffer**

Is NULL.

**nOutBufferSize**

Is 0.

### *Comments*

This command can be used to set the hardware ID of a USB configuration and an interface. The new value is stored in the registry. It is used by the driver if the child device node is created the next time. To activate the new hardware ID disable and enable the device node. The function does not check whether the combination of USB configuration and index is valid. The hardware ID consists of two parts separated by a backslash. If the string does not contain a backslash the IOCTL return with invalid paramter.

### *See Also*

**IOCTL\_USBMICTRL\_SET\_HWID** (page 40)

**IOCTL\_USBMICTRL\_ENABLE\_INTERFACE** (page 37)

**IOCTL\_USBMICTRL\_DISABLE\_INTERFACE** (page 36)

**UsbmictrlHWID** (page 45)

## 5.2 Data Structures

### GenericDriverInfo

This structure contains the version information of the driver and the API.

#### Definition

```
typedef struct tagGenericDriverInfo{
    unsigned int interfaceVersionMajor;
    unsigned int interfaceVersionMinor;
    unsigned int driverVersionMajor;
    unsigned int driverVersionMinor;
    unsigned int driverVersionSub;
    unsigned int flags;
} GenericDriverInfo;
```

#### Members

##### **interfaceVersionMajor**

Contains the major version of the interface. This version is changed if the interface is modified in a non compatible way.

##### **interfaceVersionMinor**

Contains the minor version of the interface. This version is changed each time the interface is changed.

##### **driverVersionMajor**

Contains the major version of the device driver. This version is changed if the driver reaches a new quality.

##### **driverVersionMinor**

Contains the minor version of the device driver. This version is changed on new features.

##### **driverVersionSub**

Contains the sub version of the device driver. This sub version is changed at least each time a modification is done.

##### **flags**

This contains either zero or `GENERIC_INFOFLAG_CHECKED_BUILD`, if the driver version is a debug build.

#### Comments

Use this structure to check the compatibility of the application and the driver.

#### See Also

**IOCTL\_USBMICTRL\_GET\_DRIVER\_INFO** (page 30)

## UsbmictrlConfig

This structure contains the USB configuration index.

### *Definition*

```
typedef struct tagUsbmictrlConfig{
    unsigned int ConfigurationIndex;
} UsbmictrlConfig;
```

### *Member*

#### **ConfigurationIndex**

Contains the USB configuration index.

### *See Also*

**IOCTL\_USBMICTRL\_GET\_ACTIVE\_CONFIGURATION** (page 31)

**IOCTL\_USBMICTRL\_SET\_ACTIVE\_CONFIGURATION** (page 32)

## UsbmictrlConfInterface

This structure contains the USB configuration index and the interface number.

### *Definition*

```
typedef struct tagUsbmictrlConfInterface{
    unsigned char ConfigurationIndex;
    unsigned char Interface;
} UsbmictrlConfInterface;
```

### *Members*

#### **ConfigurationIndex**

Contains the USB configuration index. This is not the bConfiguration value reported in the configuration descriptor. It is the index used to request the configuration descriptor.

#### **Interface**

Contains the USB interface index.

### *See Also*

[IOCTL\\_USBMICTRL\\_GET\\_HWID](#) (page 39)

[IOCTL\\_USBMICTRL\\_SET\\_HWID](#) (page 40)

## UsbmictrlHWID

This structure contains the USB configuration index and the hardware ID.

### *Definition*

```
typedef struct tagUsbmictrlHWID{
    unsigned char ConfigurationIndex;
    unsigned char Interface;
    WCHAR HWID[64];
} UsbmictrlHWID;
```

### *Members*

#### **ConfigurationIndex**

Contains the USB configuration index. This is not the bConfiguration value reported in the configuration descriptor. It is the index used to request the configuration descriptor.

#### **Interface**

Contains the USB interface index.

#### **HWID[64]**

Contains a zero terminated unicode string with the hardware ID.

### *See Also*

[IOCTL\\_USBMICTRL\\_GET\\_HWID](#) (page 39)

[IOCTL\\_USBMICTRL\\_SET\\_HWID](#) (page 40)

## UsbmictrlDescriptorRequest

This structure specifies a USB descriptor request.

### Definition

```
typedef struct tagUsbmictrlDescriptorRequest{
    UsbmictrlRequestRecipient Recipient;
    UCHAR DescriptorType;
    UCHAR DescriptorIndex;
    USHORT LanguageId;
} UsbmictrlDescriptorRequest;
```

### Members

#### **Recipient**

Is one of the recipients of the request which are defines in [UsbmictrlRequestRecipient](#).

#### **DescriptorType**

Contains the descriptor type. Refer to [2] for more information.

#### **DescriptorIndex**

Contains the index of the descriptor. Refer to [2] for more information.

#### **LanguageId**

Contains the language ID of the descriptor. It is required for string descriptors, only. In other calls it is set to zero. Refer to [2] for more information.

### Comments

A USB descriptor contains information about the USB interface of the device.

### See Also

[UsbmictrlRequestRecipient](#) (page 51)

[IOCTL\\_USBMICTRL\\_GET\\_DESCRIPTOR](#) (page 33)

## UsbmictlClassOrVendorRequest

This structure specifies a class or vendor request.

### Definition

```
typedef struct tagUsbmictlClassOrVendorRequest{
    BOOLEAN Read;
    UsbmictlRequestType Type;
    UsbmictlRequestRecipient Recipient;
    UCHAR Request;
    USHORT Value;
    USHORT Value;
    USHORT Length;
} UsbmictlClassOrVendorRequest;
```

### Members

#### **Read**

If this parameter is **TRUE** the data phase transfers data from the device to the PC (IN). Otherwise the data transfer occurs from the PC to the device (OUT).

#### **Type**

Is one of the types of the request which are defines in [UsbmictlRequestType](#).

#### **Recipient**

Is one of the recipients of the request which are defines in [UsbmictlRequestRecipient](#).

#### **Request**

Contains the request byte of the Setup. Refer to [2] for more information.

#### **Value**

Contains the value word of the Setup. Refer to [2] for more information.

#### **Value**

Contains the index word of the Setup. Refer to [2] for more information.

#### **Length**

Contains the length word of the Setup. Refer to [2] for more information.

### Comments

A class or vendor request has a data phase with a size of up to 4096 bytes. The parameters **Request**, **Value**, and **Index** can be chosen by the application. The parameter **Length** contains the length of the data stage. If the parameter **Read** is **FALSE** the parameter **Length** can be set to 0.

### See Also

**UsbmictrlRequestType** (page 52)

**UsbmictrlRequestRecipient** (page 51)

**IOCTL\_USBMICTRL\_CLASS\_OR\_VENDOR\_REQUEST** (page 34)

## UsbmictrlInterface

This structure specifies a USB interface.

### Definition

```
typedef struct tagUsbmictrlInterface{
    UCHAR Interface;
} UsbmictrlInterface;
```

### Member

#### **Interface**

Is the interface number of the first USB interface of an interface bundle. The interface number is related to the currently selected configuration. The value is stored related to the current configuration. Each configuration can have different settings.

### Comments

A interface bundle is a set of USB interfaces that belongs together and that are handled by one device driver. Such a bundle can be described with an interface associated descriptor (IAD).

### See Also

[IOCTL\\_USBMICTRL\\_ENABLE\\_INTERFACE](#) (page 37)

[IOCTL\\_USBMICTRL\\_DISABLE\\_INTERFACE](#) (page 36)

[IOCTL\\_USBMICTRL\\_GET\\_INTERFACE\\_STATE](#) (page 38)

## UsbmictrlInterfaceState

This structure specifies a USB interface state.

### Definition

```
typedef struct tagUsbmictrlInterfaceState{
    UCHAR Enabled;
} UsbmictrlInterfaceState;
```

### Member

#### **Enabled**

If the value is different from 0 the USB interface is enabled. If it is zero the interface is not enabled.

### Comments

The storage for the interface state is related to the USB configuration. Each configuration has individual settings.

### See Also

[IOCTL\\_USBMICTRL\\_ENABLE\\_INTERFACE](#) (page 37)

[IOCTL\\_USBMICTRL\\_DISABLE\\_INTERFACE](#) (page 36)

[IOCTL\\_USBMICTRL\\_GET\\_INTERFACE\\_STATE](#) (page 38)

## 5.3 Enumeration Types

### UsbmictrlRequestRecipient

The UsbmictrlRequestRecipient enumeration type contains values that identify the recipient of an USB request.

#### *Definition*

```
typedef enum tagUsbmictrlRequestRecipient {  
    RecipientDevice = 0,  
    RecipientInterface,  
    RecipientEndpoint,  
    RecipientOther  
} UsbmictrlRequestRecipient;
```

#### *Entries*

##### **RecipientDevice**

The recipient of the request is the device.

##### **RecipientInterface**

The recipient of the request is the interface.

##### **RecipientEndpoint**

The recipient of the request is the endpoint.

##### **RecipientOther**

The recipient of the request is some other.

#### *Comments*

The request recipient is defined by the USB specification [2].

## UsbmictrlRequestType

The UsbmictrlRequestType enumeration type contains values that identify the kind of an USB request.

### Definition

```
typedef enum tagUsbmictrlRequestType{
    RequestTypeClass = 1,
    RequestTypeVendor
} UsbmictrlRequestType;
```

### Entries

#### **RequestTypeClass**

The request is a class specific request.

#### **RequestTypeVendor**

The request is a vendor specific request.

### Comments

The request type is defined by the USB specification [2].

## 6 Debug Support

### 6.1 Enable Debug Traces

The licensed version of the driver package contains a folder `drv_chk` with the debug version of the driver. This folder is not part of the demo version. The debug version of the driver can generate text messages on a kernel debugger or a similar application to view kernel output. These messages can help to analyze problems.

To enable the debug traces follow these steps.

- Install the driver with the normal setup.
- Copy the debug version of the driver to the folder `%SystemRoot%\system32\drivers`. If the driver has been renamed the debug version must be renamed in the same way.
- Reboot the PC to make sure the new driver is loaded. Connect your device.
- Open the registry editor. On Windows 2000/XP/Vista the following path must be opened:  
`\HKEY_LOCAL_MACHINE\System\CurrentControlSet\services\YOUR_SERVICE_NAME\`.

`YOUR_SERVICE_NAME` is the name of the service name defined in the INF file. It is typically the name of your driver.

Edit the `DWORD` value key `TraceMask`. The messages are grouped to special topics. Each topic can be enabled with a bit in the debug mask. To enable the messages on bit 5 the `DebugMask` must be set to `0x00000020`. The `DebugMask` contains the or'ed value of all active message bits.

- Disconnect all devices or reboot the PC to make sure the driver is loaded again. The driver reads the registry key `TraceMask` if it is started.
- Start a kernel debugger like `WinDbg` or an application like `DebugView` to receive the kernel traces. Connect your device.
- If your problem ends up in a blue screen of death or you see an unexpected re-boot of the PC please change the following settings: `System Properties -> Advanced -> Startup and Recovery -> Settings -> Write Debugging Information to "Kernel Memory Dump"`. Set the registry key `TraceLogSizeKB` in the same location as the `TraceMask` to `128`. This includes the last 128kb traces to the memory dump. Transfer the memory dump to `Thesycon` for analysis.

The following table summarize the meaning of the debug bits.

Table 1: Debug Mask Bit Content

Bit	Content
0	Fatal errors

Table 1: (continued)

<b>Bit</b>	<b>Content</b>
1	Warnings
2	Information
3	Extended information
8	USB requests from child devices

## Index

GenericDriverInfo, [41](#)

IOCTL\_USBMICTRL\_CLASS\_OR\_VENDOR\_REQUEST, [34](#)

IOCTL\_USBMICTRL\_CYCLE\_PORT, [35](#)

IOCTL\_USBMICTRL\_DISABLE\_INTERFACE, [36](#)

IOCTL\_USBMICTRL\_ENABLE\_INTERFACE, [37](#)

IOCTL\_USBMICTRL\_GET\_ACTIVE\_CONFIGURATION, [31](#)

IOCTL\_USBMICTRL\_GET\_DESCRIPTOR, [33](#)

IOCTL\_USBMICTRL\_GET\_DRIVER\_INFO, [30](#)

IOCTL\_USBMICTRL\_GET\_HWID, [39](#)

IOCTL\_USBMICTRL\_GET\_INTERFACE\_STATE, [38](#)

IOCTL\_USBMICTRL\_SET\_ACTIVE\_CONFIGURATION, [32](#)

IOCTL\_USBMICTRL\_SET\_HWID, [40](#)

UsbmictrlClassOrVendorRequest, [47](#)

UsbmictrlConfig, [43](#)

UsbmictrlConfInterface, [44](#)

UsbmictrlDescriptorRequest, [46](#)

UsbmictrlHWID, [45](#)

UsbmictrlInterfaceState, [50](#)

UsbmictrlInterface, [49](#)

UsbmictrlRequestRecipient, [51](#)

UsbmictrlRequestType, [52](#)