

USB CDC/ACM Class Driver

For Windows 2000, XP, Vista and 7

Reference Manual

Version 1.96

May 06, 2011

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

Tel: +49 3677 / 8462-0

Fax: +49 3677 / 8462-18

e-mail: info @ thesycon.de

<http://www.thesycon.de>

Copyright (c) 2005-2011 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

Table of Contents	7
1 Introduction	9
2 Overview	11
2.1 Platforms	11
2.2 Features	12
2.3 USB 2.0 and 3.0 Support	13
3 Architecture	15
3.1 CDCACMPO Driver USB protocols	16
3.1.1 CDC ACM Protocol	16
3.1.2 Bulk Only	17
3.1.3 Special Vendor Protocol	17
3.1.4 Auto Mode	18
3.2 Special Properties of the Driver	18
3.2.1 Static Device Node	18
3.2.2 Block Transfer	19
3.2.3 API function TransmitCommChar	19
3.2.4 Flow Control	19
3.2.5 Circular Buffer	19
3.2.6 Data Transfer	20
3.2.7 Startup Initialization	20
3.2.8 Vendor Defined Reset Pipe Command	20
3.2.9 Overlapped Mode	20
3.2.10 Enhanced Error Recovery	20
3.2.11 Power Management	21
3.2.12 Additional Interface	22
3.2.13 Device State Change Notifications	22
3.2.14 WHQL Certification	22
3.2.15 COM Port Numbers	23
3.2.16 USB Serial Number	23
3.2.17 Multiple interfaces on one device	23
4 Driver Customization	25

4.1	Customization Overview	25
4.2	Customization Steps	26
4.3	Customizing cdcacmpo.inf	27
4.3.1	Configuration of Names	27
4.3.2	Configuration of Hardware ID	28
4.3.3	Configuration of Software Interface Identifiers	29
4.3.4	Update of the Driver Version	29
4.3.5	Customizing Default Driver Settings	30
4.4	Customizing Version Resources	33
4.5	Digital Signature for Windows Vista and later	33
4.5.1	Get an Authenticode Digital ID	34
4.5.2	Get the Tools for Code Signing	35
4.5.3	Create a Signature	35
4.5.4	Modified System Behavior	36
4.6	Support for Windows CE and Windows Mobile	36
5	Driver Installation and Uninstallation	39
5.1	Driver Installation for Developers	39
5.1.1	Installing CDCACM Manually	39
5.2	Uninstalling CDCACM manually	40
5.3	Creating a Driver Setup Package	41
5.4	Installing CDCACM for End Users	41
5.4.1	Installing CDCACM with the PnP Driver Installer	41
5.4.2	Installing the CDCACM Driver with DIFx	42
6	PnP Interface	45
6.1	Plug and Play Notificator	45
	CPnPNotifyHandler class	45
	Member Functions	45
	HandlePnPMessage	45
	CPnPNotificator class	47
	Member Functions	47
	CPnPNotificator	47
	~CPnPNotificator	47
	Initialize	48
	Shutdown	50

EnableDeviceNotifications	51
DisableDeviceNotifications	52
6.2 Port Information	53
CPortInfo class	53
Member Functions	53
EnumeratePorts	53
GetPortCount	54
GetPortInfo	54
GetPortInfoByDevicePath	55
PortInfoData	56
6.3 PnP Notification Demo Application	58
7 Source Code Package	59
7.1 Translation	59
7.2 Structure of the Source Code	59
7.2.1 Driver	59
7.2.2 Device	59
7.2.3 Stream Classes	60
7.2.4 SerQueue	60
7.2.5 Helper Classes	60
8 Debug Support	61
8.1 Event Log Entries	61
8.1.1 Clear Feature Endpoint Halt	61
8.1.2 Cannot Create Symbolic Link	61
8.1.3 Descriptor Problems	61
8.1.4 Demo is Expired	61
8.2 Enable Debug Traces	61
9 Related Documents	65
Index	67

1 Introduction

The CDCACM driver is a generic device driver for Windows. It creates a virtual serial COM port interface and it can handle different USB protocols. See section 3.1 on page 16 for details about the supported protocols. The device driver supports USB 2.0 with the operating speeds full and high speed.

This document describes the architecture and the features of the CDCACM device driver. Furthermore, it includes instructions for installing and using the device driver.

The reader of this document is assumed to be familiar with the specification of the Universal Serial Bus Version 1.1 and 2.0 and with common aspects of Win32-based application programming.

2 Overview

2.1 Platforms

The CDCACM 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 CDCACM driver provides the following features:

- **USB Support.** The CDCACM class driver supports USB 2.0 full and high speed and USB 1.1.
- **COM Port.** The CDCACM class driver provides a virtual serial COM port. The provided virtual COM port is compatible with the Win32 serial port API. The virtual COM port can be used by standard Windows programs. The COM port name gets assigned automatically.
- **Static COM Port.** Optional the virtual COM port can get a static COM port behavior to support legacy applications. The application can keep the COM port opened while the device is removed and continue the communication if the device is reconnected.
- **USB Protocols.** Different USB protocols are supported.
- **Asynchronous Data Transfer.** Fully supports asynchronous (overlapped) data transfer operations
- **Error Correction.** An enhanced error correction is implemented.
- **Windows CE.** The CDCACM class driver is available for Windows CE and Windows Mobile.
- **Plug&Play.** The CDCACM class driver fully supports plug and play. It supports add/remove notifications. An additional GUID based interface enables the usage of PnP events. So a Plug&Play compliant port enumeration and identification method that is not based on COM port names is provided. A notification application demonstrates the detection of the correct COM port number without scanning the ports. A static COM behavior is optional available (see above).
- **Power Management.** The CDCACM class driver supports the Windows power management model.
- **Multiple USB Interfaces.** The CDCACM class driver can be used with devices that implement multiple USB interfaces. A separate network adapter instance will be created for each CDCACM interface. Thesycon offers a multi-interface driver that is required to build an individual device node for each interface. For more information, check out <http://www.thesycon.de> USB Multi Interface Driver.
- **Multiple USB Devices.** Multiple USB devices can be controlled by the driver at the same time
- **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 CDCACM device driver supports USB 2.0 and the Enhanced Host Controller on Windows 2000, Windows XP and Windows Vista and W7. However, CDCACM has to be used on top of the driver stack that is provided by Microsoft. Thesycon does not guarantee that the CDCACM 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 CDCACM 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 CDCACM 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

Figure 1 shows the USB driver stack that is part of the Windows operating system. All drivers are embedded within the WDM layered architecture.

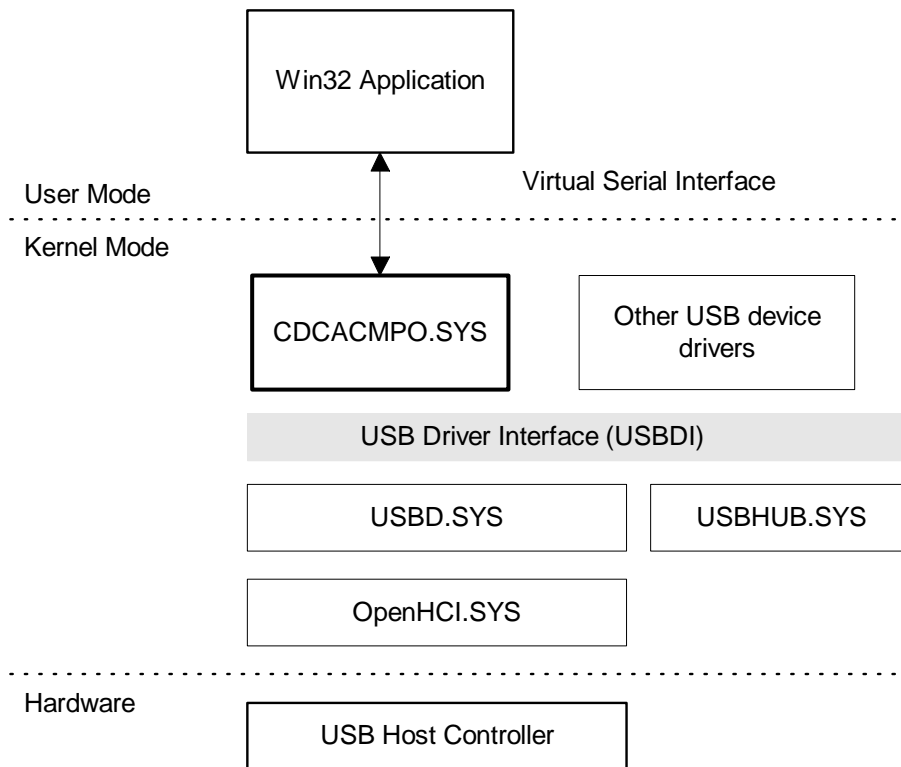


Figure 1: USB Driver Stack

The following modules are shown in Figure 1:

- **USB Host Controller** is the hardware component that controls the Universal Serial Bus. It also contains the USB Root Hub. There are two implementations of the host controller that support USB full speed: Open Host Controller (OHC) and Universal Host Controller (UHC). There is one implementation of the host controller that supports USB high speed: Enhanced Host Controller (EHC).
- **OpenHCI.SYS** is the host controller driver for controllers that conform with the Open Host Controller Interface specification. Optionally, it can be replaced by a driver for a controller that is compliant with UHCI (Universal Host Controller Interface) or EHCI (Enhanced Host Controller Interface). Which driver is used depends on the main board chip set of the computer. For instance, Intel chip sets contain Enhanced Host Controllers and Universal Host Controllers.
- **USB D.SYS** is the USB Bus Driver that controls and manages all devices connected to the USB. It is provided by Microsoft as part of the operating system.
- **USBHUB.SYS** is the USB Hub Driver. It is responsible for managing and controlling USB Hubs.

- CDCACMPO.SYS is a kernel mode driver that supports various USB protocols and emulates a virtual serial COM port.

The software interface that is provided by the operating system for use by USB device drivers is called USB Driver Interface (USBDI). It is exported by the USBD at the top of the driver stack. USBDI is an IRP-based interface. This means that each individual request is packaged into an I/O request packet (IRP), a data structure that is defined by WDM. The I/O request packets are passed to the next driver in the stack for processing and will return to the caller after completion.

3.1 CDCACMPO Driver USB protocols

The CDCACMPO driver supports three USB protocols:

- Communication Device Class (CDC) – Abstract Control Model (ACM)
- Bulk Only with two bulk pipes, optionally with an interrupt endpoint in the same interface
- Special vendor protocol with two bulk pipes

The driver is configured by the .INF file parameter `OperationMode` for the correct protocol.

3.1.1 CDC ACM Protocol

In this mode the driver expects a device with a interface that is compatible to the CDC ACM specification [3]. A special feature of the driver is the mapping of the CDC ACM signal `NETWORK_CONNECTION` to the CTS signal.

The driver accepts modifications of the interface. It accepts interfaces without an interrupt pipe. In this case the status lines will be emulated. It accepts one USB interface that contains the interrupt and the bulk endpoints. Such an interface can have a vendor specific class code.

In this mode the driver sends the following class requests:

- `SET_LINE_CODING`
- `SET_CONTROL_LINE_STATE`
- `SEND_BREAK`

Other optional class requests are not used by the driver. These three class request are defined as optional by the CDC specification. The driver can be configured with the following parameters in the INF file:

- `SendLineCoding`
- `SendLineState`
- `SendBreak`

If a parameter is set to 1 the correspondent class request is sent by the driver. If the parameter is set to 0 the request is not sent to the device. The default value for all three parameters is 1.

3.1.2 Bulk Only

The driver expects a vendor defined interface with two bulk pipes, one in IN and one in OUT direction. The class code, subclass code and protocol are not validated in this mode. The class has no additional alternate setting. If the interface does not contain an interrupt IN endpoint the driver does not send any class specific requests in this mode. The signals RTS and DTR are emulated. The initial value is active. The application can set any state. The current state can be requested by the application. The signals CTS, DSR, CDC and RI can be configured with the parameter `DefaultLineState`. In the bulk mode the default line state cannot be changed by the device.

3.1.3 Special Vendor Protocol

The idea of the special vendor protocol is to use only two USB endpoints and transfer the serial status signals between the device and the PC. This saves one USB endpoint in comparison to the standard CDC ACM interface. The device must support the class specific CDC ACM requests. The requests can be configured in the same way as described in the CDC ACM mode. The information about the status signals of the device are transferred in each first byte of a data transfer on the bulk IN endpoint. The first byte has the following structure

Table 1: Definition of the status byte.

Bit Value	Meaning
0x01	CTS signal active
0x02	DSR signal active
0x04	DCD signal active
0x08	RI signal active
0x10	BREAK signal active
0x40	Responds available

The device sends the status values each time a change occurs. It sends the status values after set configuration one time. The status byte can be sent separately without data. The status byte is always the first byte of each submitted FIFO. E.g. if the FIFO size of the device is 64 bytes and the device wants to send 64 bytes data, it prepares the first FIFO with one status byte and 63 data bytes and the second FIFO with one status byte and one data byte.

If the device sends 63 bytes it prepares the first FIFO with a status byte and 63 data bytes and the second FIFO with one status bytes. This is required to inform the PC driver that the data transfer is completed.

The data on the OUT pipe are transferred without additional information. The device has one USB interface with the class 0xff, subclass 0x01 and the protocol 0x00.

3.1.4 Auto Mode

The operational mode can be set to the Auto Mode option. In this mode the driver checks the interface descriptor. If the device has an interface with the class CDC (0x02) and the subclass ACM (0x02) the driver uses the CDC ACM protocol. If such an interface is not found the Bulk Only protocol is used. The Special Vendor Protocol is not detected in Auto Mode. This is the default value for the OperationMode set in the .INF file.

3.2 Special Properties of the Driver

3.2.1 Static Device Node

USB drivers are typically plug and play drivers. That means the driver is loaded when the device is connected and it is unloaded when the device is disconnected. A plug and play aware application can detect this changes and can close or open the handle to the device driver. The example `portNotificator` demonstrate the usage of the system provided plug and play notifications.

An application that is not plug and play compliant may have a problem if the device is removed. The handle to the device driver becomes invalid and even if the device is re-connected the handle stays invalid. Such an application must be closed and restarted to re-enable the communication with a plug and play driver.

The CDCACMPO driver has a special feature to enable legacy applications to use the opened handle after the device was removed and re-connected. This feature is enabled with the registry key `StaticDeviceObject`, see [4.3.5](#).

The driver behaves in the following way if the feature is enabled:

- When the device is removed the status lines CTS, DSR, RI and DCD are reported as inactive. If the application waits on events the change is signaled as an event.
- When the device is removed all requests that causes class specific requests are accepted but are not sent to the device.
- Read and write requests are stored in IRP queues. After the timeout expires the requests are returned with timeout and a transfer length set to 0.
- When the device is re-connected the status lines are set to the default value defined with the parameter `DefaultLineState`. The change is reported with events. As soon as the device reports new line states the reported states are indicated to the application.
- The read and write process is started and still pending write operations are executed. Pending read requests are used to indicate data to the application.

The device may lose all internal states if it is disconnected from the PC. Furthermore some data that are transferred while the device is disconnected may be lost. The read and write operations may be terminated with timeout while the device is disconnected. These effects may cause problems in an application that does not realize that the device is disconnected.

The static device object prevents the system from unloading the driver image from the memory. This may cause an unexpected behavior if the driver is updated. The update process may copy a new driver to the hard disk and install it. But the system may use the old driver until the system is

rebooted. For that reason the installation program must take care to terminate all applications that are using the COM ports before starting the installation process or to request the user to re-start the PC.

Even if this feature is enabled the CDCACMPO driver is loaded as a plug and play driver. If the device is not connected to the PC the COM port cannot be opened by an application and the device node is not visible in the device manager.

3.2.2 Block Transfer

The virtual COM port is an emulation of a physical COM port. The data transfer is organized in data blocks rather than characters on the serial port. This causes some differences in the API of the virtual port in comparison to a physical port.

There is no relation between the data blocks that are submitted to the serial COM port and the data blocks on the USB.

A data transfer from the device as well as from the PC must be terminated with a short packet under the following circumstances:

- The transferred amount of data can be divided by the FIFO size without a rest.
- The transferred data size is less the maximum buffer size defined in the INF file with the parameter `ReadBufferSize` or `WriteBufferSize`.
- There are no more data that can be transferred immediately after the current data transfer.

3.2.3 API function `TransmitCommChar`

A character that is submitted with the Win32 API function `TransmitCommChar` is transmitted in the normal data stream.

3.2.4 Flow Control

The flow control uses always the flow control of USB independent of the setting of any control signals and independent of the selected flow control model. USB flow control means, that the PC stops sending IN tokens if the application connected to the virtual COM port does not read the data and all internal buffers are full. The device sends NAK tokens on the OUT pipe if the FIFO in the device is full.

The signals that are transferred in CDC ACM and in special mode are only for informational purposes between the PC application and the application in the device. No signal has an influence on the behavior of the driver.

3.2.5 Circular Buffer

The driver has a circular buffer for the send and the receive direction. The size of the buffer can be modified with the Win32 API function `SetupComm`. If the size of the circular buffer is modified all data in the buffer are deleted. The driver has build in limits for the buffer size of 128 and

128000 bytes. The default value is 4096 bytes. The circular buffer is always in the data path. A small buffer can cause a higher CPU load if a larger band width is transferred.

3.2.6 Data Transfer

The data transfer is started if the COM port is opened. It is stopped if the COM port is closed. This saves USB band width if no application is interested to get data from the device.

3.2.7 Startup Initialization

If the COM port is opened the driver sends the standard USB request Clear Feature Endpoint Halt on each endpoint to synchronize the data toggle bits, to clear error conditions in the bus driver and to clear old data in the FIFOs. This request can be suppressed by the parameter ClearFeatureOnStart defined in the INF file. Some versions of Windows 2000 may have problems if you turn off the request. In the mode CDC ACM or in the Special Mode the driver sends the initial state of the signals RTS and DTR to the device.

3.2.8 Vendor Defined Reset Pipe Command

Some devices do not inform the software if a Clear Feature Endpoint Halt request is processed. To handle the request in software the driver can send a vendor defined request. The request is sent after the Clear Feature Endpoint Halt command. It can be enabled with the parameter VendorPipeReset in the INF file. It is turned off by default. The vendor defined request has the following layout:

Table 2: Definition of the vendor command reset pipe

bmRequestType	bRequest	wValue	wIndex	wLength
0x42	0x01	0x0000	Endpoint	0x0000

The bmRequestType 0x42 means OUT request (host to device), vendor defined request and the target is a endpoint.

3.2.9 Overlapped Mode

If the driver is opened in Overlapped mode each function can return the special status code ERROR_IO_PENDING. This behavior is allowed by the DDK and may be different than in other implementations of serial drivers. The PC application has to handle the Win32 API in a correct way. See the SDK [6] for more details.

3.2.10 Enhanced Error Recovery

The data transfer on USB is protected by CRC16. The host controller retries each transfer up to 3 times if a transfer error occurs. If the 3 retries of the host controller are not transfer the data block correctly the PC driver detects the error and starts the error handling.

The idea of the enhanced error recovery is to perform the error handling without any data loss. To perform the enhanced error recovery the PC driver and the device use a logical buffer size. This logical buffer size does not limit the amount of data that are transferred nor force the sender to send always data packets with the logical buffer size. It is the size of a data block that is repeated if during the data transfer a transmission error occurs that is not solved by the hardware.

The size of the logical data block must be a multiple of the physical FIFO size of the endpoint. There are two constraints that should be considered to select this size. The device must be able to store a logical data block in both directions and for each endpoint. If the logical buffer size is selected large, the device needs more memory. If the logical buffer size is selected small, the performance of the data transfer is reduced dramatically.

How does the normal data transfer work with the logical buffer? The PC sends a data block with the size of the logical buffer. This buffer is transferred in chunks of the FIFO size over the bus. The device collects all chunks into the logical buffer. If the last chunk is transferred completely it passes the logical buffer to the application. If the PC sends less data than the logical buffer size the device detects a short packet. In this case the contents received so far in the logical buffer are passed to the application. If the PC receives data it works in the same way. It is important that the device stores the logical buffer data until the last chunk is transferred successfully.

The error recovery works in the following way. Let's consider an OUT pipe first. The PC detects a transmission error during the transmission of a logical buffer. It sends a Clear Feature Endpoint Halt for the OUT endpoint. The device discards all data that are received so far in the logical buffer, clears the FIFO buffer and resets the data toggle bit to 0. The PC starts to retransmit the logical buffer from the beginning.

On an IN pipe it works in the same way. The device starts after the Clear Feature Endpoint Stall request to send the first chunk of the logical buffer.

The debug version of the driver has a special feature to simulate hardware transmission errors. In IN direction the driver uses a buffer with the size 1. This causes a buffer overflow error. Such an error is handled in the same way like a CRC error. In OUT direction the driver sends a standard request Set Feature Endpoint Halt. This causes the endpoint to stall the next data transfer.

The debug driver supports two additional configuration parameters:

- DbgRcvErrors
- DbgSendErrors

If the parameters are set to zero no error is generated. If the value is greater than zero the driver transfers the given number of logical buffers and generates an error. These parameters are stored in the registry in the hardware key of the driver. The path is

```
HKLM\system\CCS\enum\USB\VID_VVVV&PID_PPPP<serial number>\Device Parameters
```

The parameters are read each time the device is connected to the PC.

3.2.11 Power Management

In suspend mode the driver must stop the data transfer and abort all pending requests. If a buffer is partially transferred the abort process can cause a data loss. It is recommended that an applica-

tion registers for Power Management messages and stops the data transfer before the PC enters a suspend state.

3.2.12 Additional Interface

It may be a problem to find the correct COM port number to open a special device. The driver allows the user to configure an additional interface that is based on a GUID. An application can open the driver by enumerating the GUID. The returned handle can be used in the same way as a handle that was received by opening a COM port name. The advantage of this method is, that the application can enumerate for devices with a special interface and the COM port number must not be known to the application. The classes PnPNotificator and PortInfo demonstrate the usage of this interface and show how the COM port number can be determined.

The section 4.1 Customization describes how the INF file should be modified to use this feature.

3.2.13 Device State Change Notifications

The application is able to receive notifications when the state of a USB device changes. The Win32 API provides the function **RegisterDeviceNotification()** for this purpose. This is the way, an application is notified if a USB device is plugged in or removed.

The application should use the GUID of the additional interface to register for PNP notifications. This ensures that only notifications for the related devices are received.

Please refer to the Microsoft Platform SDK documentation for detailed information of the functions

RegisterDeviceNotification() and
UnregisterDeviceNotification().

If your application does not have a Windows or a service handle you can use the PnPNotificator class to receive the notifications.

3.2.14 WHQL Certification

The CDCACM driver is tested with the DTM test bench and can pass all tests of the WHQL certification process. It is not possible to deliver a generic certified driver because a lot of the tests are device related. For that reason each combination of a device and a driver must be tested and certified separately. Thesycon can support the certification process on request.

The advantages of a certified driver package are:

- Higher quality of the device interface and the PC driver.
- No warning during installation process.
- A simplified installation process.
- A silent installation with standard user rights if the driver was pre-installed with administrator privileges.

3.2.15 COM Port Numbers

The CDCACM driver is installed in the device class ports. This class has a class co-installer that assigns a free COM port number to the device. This is the official recommended way to obtain a new COM port number. If the driver is uninstalled the COM port number is freed. Unfortunately there are drivers on the market that do not use the correct way to obtain a free COM port number. If such a driver is installed it can happen that two drivers try to create a COM port with the same number. The second driver cannot do this because the link name is a global resource on Windows. The second driver that tries to create the link fails to start. This problem can be solved manually by assigning a free COM port number in the property page. The device must be removed and connected before this change becomes active.

3.2.16 USB Serial Number

Each USB device can report a serial number descriptor. If a device has a serial number descriptor Windows assigns always the same COM port number to the device regardless to which USB port the device is connected. The COM port number may change if the device is de-installed and installed again.

If the device does not report a serial number the device is installed each time it is connected to a different USB port and a new COM port number is assigned. This has some drawbacks:

- The number of COM ports is limited to 256. If the PC runs out of free COM port numbers the installation fails.
- It may be hard to find the correct device.

It is recommended to report a serial number in the USB device descriptor.

The USB serial number must be unique. Otherwise only one device per PC can be used and the WHQL test fails.

3.2.17 Multiple interfaces on one device

The CDCACM driver can be used with devices that implement multiple USB interfaces. In this case a multi-interface driver is required. This driver splits the interfaces to separate device nodes. On each device node a new driver stack can be installed.

The system provided multi-interface driver cannot be used together with the CDC ACM driver. The reason is that this CDC ACM interface consists of two USB interfaces. This two interfaces must be mapped to one device node. The system supplied multi-interface driver is not able to detect USB interfaces that belong together.

Thesycon provides a special multi-interface driver that can handle this problem in a correct way. This driver can be licensed separately.

4 Driver Customization

4.1 Customization Overview

The CDCACM 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 CDCACM device driver. Please consider the following example scenario: An end user buys product A which includes the CDCACM 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 CDCACM 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 CDCACM 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) CDCACM 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 and the .inf files. The driver package consists of the driver files `cdcacmpo.sys` and `cdcacmpo_x64.sys` as well as the INF files `cdcacmpo.inf` and `cdcacmpo_x64.inf`.

Make a private copy of the sys and inf files and rename the files to your new names. Edit the renamed .INF files to contain the new driver name. See section 4.3.1 on page 27 for detailed information.

- **Required:** Edit your .INF files to contain the correct hardware ID for your device. Refer to section 4.3.2 on page 28 for more information.
- **Required:** Create a private interface identifier (GUID). Specify that GUID in your .INF files and use that GUID in your applications to enumerate and open devices. See section 4.3.3 on page 29 for detailed information.
- **Optional:** Adapt default driver settings. Refer to section 4.3.5 on page 30 for more information.
- **Optional:** Edit the version resources contained in `cdcacm.sys`. See section 4.4 on page 33 for more information.

4.3 Customizing cdcacmpo.inf

The cdcacmpo.inf file is used to install the kernel-mode device driver cdcacmpo.sys on a 32 bit system. For x64 systems the files cdcacmpo_x64.inf and cdcacmpo_x64.sys are used. The .INF files have to conform to INF file conventions defined for WDM drivers. Refer to the Windows DDK documentation [5] for more information about INF files.

The cdcacmpo(_x64).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 cdcacmpo(_x64).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_DeviceDesc1="Virtual USB COM Port 1"
S_DiskName="CDC ACM Driver disk"
S_DriverName="cdcacmpo(_x64) "
S_ServiceName="cdcacmpo"
```

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_DeviceDesc1

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. The name can contain spaces but the size is limited to 32 characters.

S_DiskName

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

S_DriverName

This variable defines the name of the cdcacm driver executable which is cdcacmpo(_x64).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 cdcacmpo.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 `cdcacmpo` to locate the lines that have to be modified.

S_ServiceName

The service name must be unique on one PC as well as the driver file name. It should contain a name equal or similar to the driver file name without any extension. Use a name without spaces and special characters.

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_VVVV&PID_PPPP
```

The fields `VVVV` and `PPPP` 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
```

Additionally the hardware ID is displayed in the device manager under Windows XP SP2 and higher under Properties -> Details -> Hardware ID.

The `cdcacmpo(_x64).inf` file needs to specify the resulting hardware ID for the device within the following section:

```
[_Devices]
%S_DeviceDesc1%=_Install1, USB\VID_VVVV&PID_PPPP
```

Replace `VVVV` and `PPPP` by your specific values, for example:

```
[_Devices]
%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:

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

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

If your device contains more than one USB interface and the CDCACM driver should be loaded on an interface of your device the following hardware ID must be used:

```
[_Devices]
%S_DeviceDesc1%=_Install1, USB\VID_152A&PID_0001&MI_00
```

The two digits after MI are the USB interface number. If you want to load the driver for more than one interface you have to add a hardware ID for each interface, e.g.:

```
[_Devices]
%S_DeviceDesc1%=_Install1, USB\VID_152A&PID_0001&MI_00
%S_DeviceDesc1%=_Install1, USB\VID_152A&PID_0001&MI_02
```

4.3.3 Configuration of Software Interface Identifiers

The CDCACM driver creates a device instance for each of your USB devices or each interface of your USB devices connected to the system. A device instance exports a COM port number and an additional software interface which can be used by your applications to access the device. The additional software interface is unambiguously identified by a globally unique identifier (GUID). The CDCACM 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 cdcacmpo.inf file is used to define a private interface GUID.

```
[_AddReg_HW]
;HKR,,DriverUserInterfaceGuid,%REG_SZ%,"{????????-????-????-????-????????????}"
```

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 cdcacmpo(_x64).inf section shown above and uncomment the line specifying the DriverUserInterfaceGuid parameter. When done, the section should look like this:

```
[_AddReg_HW]
HKR,,DriverUserInterfaceGuid,%REG_SZ%,"{5510F365-363E-407b-80A5-C663533E93B5}"
```

Use the generated private GUID in all of your Windows applications to open the device driver. 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,,ReadBufferSize, %REG_DWORD%, 1024
HKR,,WriteBufferSize, %REG_DWORD%, 1024
HKR,,UseLogicBuffer, %REG_DWORD%, 0
HKR,,ReadBufferCount, %REG_DWORD%, 1
HKR,,WriteBufferCount,%REG_DWORD%, 1
HKR,,SendLineCoding, %REG_DWORD%, 1
HKR,,SendLineState, %REG_DWORD%, 1
HKR,,SendBreak, %REG_DWORD%, 1

; 0 CDCACM, 1 Bulk Only, 2 Bulk special, 3 - automatic
HKR,,OperationMode,%REG_DWORD%, 3
HKR,,ClearFeatureOnStart,%REG_DWORD%, 1
HKR,,VendorPipeReset,%REG_DWORD%, 0
HKR,,DefaultLineState,%REG_DWORD%, 0
HKR,,IgnorePurgeTxClear,%REG_DWORD%,0
HKR,,StaticDeviceObject,%REG_DWORD%,0
HKR,,DisConWriteMode,%REG_DWORD%,0

HKR,,DeviceObjectName,%REG_SZ%,"thcdcacm"

HKR,,DoNotSendShortPackets,%REG_DWORD%,0
```

ReadBufferSize

This parameter defines the size of the buffer that is used by the driver to read data from the device. The buffer size must be a multiple of the FIFO size. This value is important for the correct function of the error recovery. See the section 3.2.10 for details. It must be equal to the "logical buffer size". The default value of 0 means the FIFO size of the endpoint is used.

WriteBufferSize

This parameter defines the size of the buffer that is used by the driver to write data to the device. The buffer size must be a multiple of the FIFO size. This value is important for the correct function of the error recovery. See the section 3.2.10 for details. It must be equal to the "logical buffer size". The default value of 0 means the FIFO size of the endpoint is used.

UseLogicBuffer

If this parameter is 0 the driver sends a zero length packet if the TX buffer is empty and the transfer size can be divided by the FIFO size without a rest. This is the default setting. It works independent from the buffer size used in the device. It does not work with the enhanced error recovery, see section 3.2.10.

If this flag is set to 1 the driver sends a zero length packet if the transfer size is smaller than the logical buffer size (`WriteBufferSize`) and the transfer size can be divided by the FIFO size without a rest. The main difference between both modes is, that in the case that this flag is set to 1 no zero length packet is sent if the transfer size is equal to the logical buffer size. E.g. if `WriteBufferSize` is set to zero the driver uses the FIFO size as the logical buffer size and a zero length packet is never sent.

ReadBufferCount

This parameter defines the number of buffers that are used by the driver to read data. If this value is set to one the performance of the data transfer may be low. A larger value may increase the performance but may use more system resources. The value can be selected between 1 and 10. On Windows XP SP1 this value must be set to 1. Otherwise a bug in the USB driver stack causes an exchange in the order of the transferred buffers.

WriteBufferCount

This parameter defines the number of buffers that are used by the driver to write data. If this value is set to one the performance of the data transfer may be low. A larger value may increase the performance but may use more system resources. The value can be selected between 1 and 10.

OperationMode

This parameter is an enum type and must be in the range [0,3]. This parameter defines the expected USB protocol and the expected descriptors. See section 3.1 on page 16 for details about the supported protocols. If the protocol does not match the device interface the driver may fail to start.

The values have the following meaning:

- 0 – CDC ACM class protocol
- 1 – bulk only protocol, no status signals
- 2 – special bulk protocol with two endpoints and status signals
- 3 – automatic: In this mode the driver checks the interface descriptor. If the interface belongs to the CDC ACM class it used the CDC ACM mode. Otherwise it uses the bulk only mode. The special bulk protocol is not detected automatically.

SendLineCoding

This value is in the range [0,1]. If it is set to 1 the driver sends the ACM instruction SET_LINE_CODING in the appropriate mode.

SendLineState

This value is in the range [0,1]. If it is set to 1 the driver sends the ACM instruction SET_CONTROL_LINE_STATE in the appropriate mode.

SendBreak

This value is in the range [0,1]. If it is set to 1 the driver sends the ACM instruction SEND_BREAK in the appropriate mode.

ClearFeatureOnStart

This value is in the range [0,1]. If it is set to 1 the driver sends the standard instruction Clear Feature Endpoint Stall each time the COM port is opened and if the device wakes up from Standby or Hibernate. On Windows 2000 this flag is required and must be one. On later operating system this flag may be set to 0 to suppress this request.

VendorPipeReset

This value is in the range [0,1]. If it is set to 1 the driver sends the vendor defined reset pipe

command. This command is send after the standard request. The default setting is 0. See section 3.2.8 on page 20 for the layout of the request.

DefaultLineState

This value contains an or'ed value of the following flags:

- 0x10 – CTS active
- 0x20 – DSR active
- 0x40 – RI active
- 0x80 – DCD active

The value is returned as the modem status until the device returns the first valid value. In BULK_ONLY mode the device never overwrites this value.

If the device never overwrites the value at least CTS should be set to active. If the device overwrites the value all bits should be inactive as a default value.

IgnorePurgeTxClear

This value is in the range [0,1]. If it is set to 1 the command purge TX Clear is ignored by the driver. This may be activated if an application purges es OUT queue while there are still data to transfer. The default value is 0.

StaticDeviceObject

This value is in the range [0,1]. The default value is 0. If this value is set to 1 the driver creates an statical device object for the COM port. This means an application can keep the COM port opened while the device is removed. The handle to the COM port can be used after the device is connected again. See also 3.2.1 on page 18.

DisConWriteMode

This value is in the range [0,1]. The parameter is meaningful if the parameter StaticDeviceObject is set to 1. It has an influence to the handling of write requests in the time period when the device is disconnected. In mode 0 the timeout value that was configured by the application is used to handle the write requests. If the timeout expires the requests is returned with 0 bytes transferred and the status timeout. In mode 1 the driver simulates the transfer of the data on a physical COM port. With the buffer size and the baud rate a time period for the simulated data transfer is calculated. After this time the request is returned with status success and the information all bytes are transferred.

DeviceObjectName

This value is a string that contains the basic name for the device object. The complete name is constructed with by appending a number. The physical COM port driver uses 'Serial' as a prefix. Some applications accept the COM port only if the device object name has this prefix. But the usage of this prefix may corrupt the operation of the system provided COM port driver for physical COM ports. The problem may occur if physical COM ports are enabled and disabled with the device manager. The default value is 'thdcacm'.

ClearRtsDtrOnClose

This value is in the range [0,1]. If it is set to 1 the driver clears the RTS and DTR signal if the handle is closed. The class request SET_CONTROL_LINE_STATE is sent to the device if the configuration parameter SendLineState is set to 1.

DoNotSendShortPackets

This value is in the range [0,1]. If it is set to 1 the driver does not send short packets. The device must handle each FIFO content that is sent from the PC without waiting of a short packet.

4.4 Customizing Version Resources

The cdcacmpo(_x64).sys executable includes version resources. These information will be shown in the 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 cdcacmpo.sys. It is important to use a unsigned version of the driver. If a signed version is edited the .sys file cannot be used any more. Use Visual Studio to open the copy of cdcacmpo.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 cdcacmpo.sys`
The program UpdateChecksum.exe is part of the CDCACMPO driver package.

4.5 Digital Signature for Windows Vista and later

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 and later 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. Please note that the signature that is attached to a .sys file modifies the file. The CRC of such a file is modified, too, and a previously generated .cat file contains an invalid CRC.

Why can Thesycon not deliver a signed driver package? The signature becomes invalid if the driver or the .INF file are 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. The modification of these ID's 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 CDC ACM 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 Walk through" 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 it. 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. A digital ID from a different CA may require a different time stamp server. 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
```

The most common error occurs because the cross certificate is not added correctly. To check that the cross certificate is part of the signature open the file properties of the signed file -> Digital Signatures -> Details -> Advanced. The last two lines in the dialog box should contain 'unauthenticated attributes' and 'Counter Sign'. If this entry is missed you are using the wrong version of the signtool or the switch /ac was not used during signing.

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 whether the driver package should be installed or 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 Windows Vista.

4.6 Support for Windows CE and Windows Mobile

Windows CE has a built-in support for the USB device interface. The CDC/ACM PC driver can be used to communicate with this interface. It can be used if the hardware supports a USB device peripheral and a driver for this peripheral is part of the Windows CE image.

The USB device can be used with different protocol interfaces. Windows CE includes drivers for the serial and the RNDIS protocol. The serial emulation is not compliant to the CDC/ACM model. During the configuration process of the Windows CE platform one or all drivers can be removed. If you want to use the serial emulation make sure that the serial driver for the USB device (`serialusbfm.dll`) is part of the used image.

The active protocol driver for the USB device is selected by registry keys. See the Windows CE documentation "USB Function Client Driver Registry Settings" for details. Use the following entry to activate the serial emulation on the Windows CE system.

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers]
"DefaultClientDriver"="Serial_Class"
```

The TSP entry is not required and should be deleted:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\Serial_Class]
  "Tsp"="Unimodem.dll"
```

This interface is also used for the ActiveSync service. If this CDC/ACM driver is installed the ActiveSync service is no longer available. To avoid competitions between the ActiveSync PC driver and this CDC/ACM driver the USB Vendor ID (VID) and Product ID (PID) should be changed with the following registry keys:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\Serial_Class]
  "idVendor"=dword:FFFF
  "idProduct"=dword:0001
```

If you release your device the USB Vendor ID must be an official assigned ID from the USB implementers forum.

The CDCACM driver package contains two modified INF files for this Windows CE based interface. The files are located in the folder `.\idisk\windows_ce`. The following parameters are pre-configured for the Windows CE device:

```
HKR,,ReadBufferSize, %REG_DWORD%, 4096
HKR,,WriteBufferSize, %REG_DWORD%, 4096
HKR,,UseLogicBuffer, %REG_DWORD%, 0
HKR,,ReadBufferCount, %REG_DWORD%, 3
HKR,,WriteBufferCount, %REG_DWORD%, 3
HKR,,SendLineCoding, %REG_DWORD%, 0
HKR,,SendLineState, %REG_DWORD%, 0
HKR,,SendBreak, %REG_DWORD%, 0
HKR,,OperationMode, %REG_DWORD%, 1
HKR,,ClearFeatureOnStart, %REG_DWORD%, 1
HKR,,VendorPipeReset, %REG_DWORD%, 0
HKR,,DefaultLineState, %REG_DWORD%, 0x30
```

For details about the meaning of the parameters see the section 4.3.5. The parameters for the buffer size and the buffer count can be modified related to the needs of the implemented high level protocol. The parameters `SendLineCoding`, `SendLineState` and `SendBreak` are set to 0 to turn the class specific requests off. These are not supported by the Windows CE implementation.

The `OperationMode` is set to 1 to configure a bulk only device. The `DefaultLineState` parameter is set to 0x30. This sets the CTS and DSR status signals to active. This may be important for some programs that are using the COM port on the PC.

With this settings the communication between the PC and Windows CE with the Hyperterminal program was tested.

Note: During the tests we have seen that the flow control for data transfers from the PC to the device does not work correctly. We have detected the problem in the driver architecture of Windows CE. The Windows CE based application should try to read the data as fast as possible from the COM port and the PC application may send smaller chunks of data and wait for a responds for

each data chunk. The size of each data chunk should be equal or less than the read buffer size of the CE application. Such a protocol design can avoid the weakness in the implementation of the Windows CE driver.

5 Driver Installation and Uninstallation

This section discusses topics relating to the installation and un-installation of the CDCACM device driver.

5.1 Driver Installation for Developers

This section describes a method how developers can install and un-install the CDCACM driver on a PC.

5.1.1 Installing CDCACM Manually

The manually installation of driver should be used by developers, only.

In order to install the CDCACM driver manually you have to prepare an INF file that matches your device. Refer to section 5.3 on page 41 and section 4.3 on page 27 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. `cdcacmpo.inf` and `cdcacmpo.sys`). 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 some Windows systems 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 the driver installation. The CDCACM driver is not certified because it is not an end-user product. When the CDCACMPO 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 and later the driver cannot be loaded without a digital signature. Please see section 4.5 on page 33 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. A USB mouse or a USB keyboard are examples for such devices. 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.

You have to use the Device Manager to install the CDCACM 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 (`cdcacmpo.inf` and `cdcacmpo.sys`) and complete the driver installation by following the instructions shown on screen.

- After the driver installation has been successfully completed your device should be shown in the Device Manager in the Ports section. You may use the Properties dialog box of that entry to get the COM port number.
- To verify that the CDCACM drivers are working properly with your device, you should use the Hyperterminal to open the driver. If the device supports a ACSII based protocol you may enter a message and see the answer of the device.

Note: Windows Vista and later does not contain the Hyperterminal any more. Use any other terminal program.

5.2 Uninstalling CDCACM manually

To uninstall the CDCACM 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 CDCACM 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.

To avoid an automatic and silent re-installation of CDCACM by Windows 2000 and Windows XP, it is necessary to manually remove the .INF file used to install the CDCACM 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 strings 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 CDCACM driver. Instead, the New Hardware Wizard will be launched and you will be asked for a driver.

On Windows Vista and later 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.

5.3 Creating a Driver Setup Package

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

The CDCACM driver package consists of 2 INF files and 2 SYS files. See section 4.2 on page 26 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 WDK. For instructions on how to create a device-specific INF file, refer to chapter 4 on page 25.

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 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
```

5.4 Installing CDCACM for End Users

This section describes ways how the driver should be installed on a PC by the end user of the product.

5.4.1 Installing CDCACM with the PnP Driver Installer

Thesycon provides a PnP Driver Installer Package that can be used to install kernel mode drivers in a convenient and reliable way. This installer is not part of this package. It can be downloaded separately under the following link: <http://www.thesycon.de/pnpinstaller>.

The installation program can be run in an interactive mode with graphical user interface or it can be run in a command line mode. The command line mode is designed to integrate the driver installer into other installation programs. The GUI mode guides the user through the installation. It supports different languages.

The driver installer package can be customized. A detailed description of the customization options is part of the reference documentation.

The PnP Driver Installer Package can handle the driver installation and un-installation in different situations:

- During the first time installation the driver is pre-installed in the system. In this step the user needs administrator privileges. When the driver is certified the pre-installation of the driver is performed silent. This means the hardware wizard is not launched and the system does

not show a warning box for not certified software. When the device is connected to the PC during the installation the correct driver software is installed immediately. When a device is connected later to the PC the certified driver is installed silent. At the point of time where the device is connected to the PC no administrator privileges are required.

- The installer can perform a driver update. Old drivers and driver instances are removed from the system regardless whether the devices are connected or not. Exactly the driver version from the installer package is installed. This way of driver updates enables the upgrade to higher driver versions as well as the installation of older versions. The driver installation behaves in the same way as the first time installation.
- The installer can remove the driver software for a PnP device. This step can be performed by calling the installation program with a command line option or by using the appropriate option in the Windows control panel that is created during the driver installation. When the driver is removed all device nodes are uninstalled and the pre-installed drivers are removed. The installer makes sure that all drivers that are matching the USB VID and PID are removed from the system. The result is a system that behaves in the same way as when the driver software was never installed. The device nodes are left uninstalled. When a device is connected to the PC in this state the Found New Hardware wizard is launched.

The PnP Driver Installer Package supports all current Windows systems including 32 and 64 bit versions. The Thesycon team provides support and warranty for the product. A comprehensive documentation is part of the demo package available at <http://www.thesycon.de/pnpinstaller>.

5.4.2 Installing the CDCACM Driver with DIFx

The DIFx software has some limitations that are explained in this section. For that reason the installation with this tool is only the second choice and not recommended by Thesycon.

DIFx is the Driver Install Frameworks provided by Microsoft. This software component can be redistributed. DIFx consists of the driver framework for applications, the Driver Install Frameworks API and the Driver Package Installer. This section introduces the Driver Package Installer, only. For details to the DIFx framework please refer to the Microsoft documentation.

The Driver Package Installer is an executable program. It is available for x86 and x64 in two different executables with the same name 'DPInst.exe'.

The Driver Package Installer can be run in silent mode or with user interface. It can be used to uninstall a driver package and it can be configured with command line parameters.

The installer has some drawbacks:

- Depending on the device connection state it updates the driver for connected devices and it pre-installs the driver for not connected devices. If the device is connected later the system performs the normal driver rating and may select a different driver. If the driver is updated with the /f (force) flag other drivers are overwritten for connected devices. The result of the installation depends from the fact whether the device was connected or not during the installation process. It is not sure that the driver from the installation package is installed for the device.
- After uninstalling a driver the installer activates an other driver that is pre-installed for the device or it leaves the node uninstalled. This can be an older version of the driver or a demo

package. The state after uninstalling depends from other drivers that may be installed on the system.

6 PnP Interface

The CDCACM driver creates an additional vendor defined interface based on a GUID. This interface can be used to receive PnP notifications in an application and to determine the COM port name of the device that has arrived or that was removed.

This section describes the used mechanism and the API reference of two classes that cover the Windows API to simplify the access to this functionality. The source code of both classes is part of the driver packages.

6.1 Plug and Play Notificator

The Plug and Play Notificator consists out of the two classes **CPnPNotificator** and **CPnPNotifyHandler**. It can be used to register to a GUID based interface. It creates in the background a invisible Window and a thread. Both is used to receive WM_DEVICECHANGE messages from the system. This class is useful if the notification should be used in an application without a window like a console application or an DLL. If the application has a window handle it can simply uses the function **RegisterDeviceNotification**.

CPnPNotifyHandler class

The **CPnPNotifyHandler** class defines an interface that is used by the **CPnPNotificator** class to deliver device PnP notifications. This interface needs to be implemented by a derived class in order to receive Plug and Play (PnP) notifications.

Because it defines an interface, the class is an abstract base class.

Member Functions

CPnPNotifyHandler::HandlePnPMessage

This function is called by **CPnPNotificator** if a **WM_DEVICECHANGE** message is issued by the system for one of the registered device interface classes.

Definition

```
virtual void
HandlePnPMessage(
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam
) = 0;
```

*Parameters***uMsg**

The **uMsg** parameter passed to the `WindowProc` function. This parameter is set to **WM_DEVICECHANGE**. See the documentation of **WM_DEVICECHANGE** in the Windows Platform SDK for more information.

wParam

The **wParam** parameter passed to the `WindowProc` function. See the documentation of **WM_DEVICECHANGE** for more information.

lParam

The **lParam** parameter passed to the `WindowProc` function. See the documentation of **WM_DEVICECHANGE** for more information.

Comments

This function needs to be implemented by a class that is derived from **CPnPNotifyHandler**. A **CPnPNotifier** object calls this function in the context of its internal worker thread when the system issues a **WM_DEVICECHANGE** message for one of the device interface classes registered with **CPnPNotifier::EnableDeviceNotifications**.

Caution: MFC is not aware of the internal worker thread created by a **CPnPNotifier** instance. Consequently, no MFC objects should be touched in the context of this function. Furthermore, the implementation of **HandlePnpMessage** has to care about proper code synchronization when accessing data structures.

See Also

CPnPNotifier (page 47)

CPnPNotifier::Initialize (page 48)

CPnPNotifier::EnableDeviceNotifications (page 51)

CPnPNotifier class

This class implements a worker thread that uses a hidden window to receive device Plug and Play (PnP) notification messages (**WM_DEVICECHANGE**) issued by the system.

Member Functions**CPnPNotifier::CPnPNotifier**

Constructs a CPnPNotifier object.

Definition

```
CPnPNotifier();
```

CPnPNotifier::~CPnPNotifier

Destroys the CPnPNotifier object.

Definition

```
~CPnPNotifier();
```

CPnPNotificator::Initialize

Initializes the **CPnPNotificator** object, creates and starts the internal worker thread.

Definition

```
bool  
Initialize(  
    HINSTANCE hInstance,  
    CPnPNotifyHandler* NotifyHandler  
);
```

*Parameters***hInstance**

Provides an instance handle that identifies the owner of the hidden window to be created. In a DLL, specify the **hInstance** value passed to `DllMain`. In an executable, provide the **hInstance** value passed to `WinMain`. In a console application use **: :GetModuleHandle(NULL)** to obtain the instance handle.

NotifyHandler

Points to a caller-provided object that implements the **CPnPNotifyHandler** interface. This object will receive notifications issued by this **CPnPNotificator** instance.

Return Value

The function returns true if successful, false otherwise.

Comments

The function creates an internal worker thread that will register a window class and creates a hidden window. The system will post a **WM_DEVICECHANGE** message to this window if a Plug and Play event is detected for any of the registered device interface classes (see **CPnPNotificator::EnableDeviceNotifications**). The message will be retrieved by the worker thread and the thread calls **CPnPNotificator::HandlePnPMessage** passing the message parameters unmodified. The object that will receive the **CPnPNotifyHandler::HandlePnPMessage** calls is given in **NotifyHandler**. This object needs to be derived from **CPnPNotifyHandler** and implements the function **CPnPNotifyHandler::HandlePnPMessage**.

Note that a call to **Initialize** will initialize the worker thread only. In order to receive PnP notifications, **CPnPNotificator::EnableDeviceNotifications** needs to be called at least once.

The function fails if it is called twice for the same object.

See Also

CPnPNotifier::Shutdown (page 50)

CPnPNotifier::EnableDeviceNotifications (page 51)

CPnPNotifier::DisableDeviceNotifications (page 52)

CPnPNotifyHandler::HandlePnPMessage (page 45)

CPnPNotificator::Shutdown

Terminates the internal worker thread and frees resources.

Definition

```
bool  
Shutdown( ) ;
```

Return Value

The function returns true if successful, false otherwise.

Comments

This function terminates the internal worker thread, destroys the hidden window and frees all resources allocated by **CPnPNotificator::Initialize**. A call to this function will also delete all PnP notifications registered with **CPnPNotificator::EnableDeviceNotifications**.

It is safe to call this function if the object is not initialized. The function succeeds in this case.

See Also

CPnPNotificator::Initialize (page 48)

CPnPNotificator::EnableDeviceNotifications (page 51)

CPnPNotificator::DisableDeviceNotifications (page 52)

CPnPNotifier::EnableDeviceNotifications

Enables notifications for a given class of device interfaces.

Definition

```
bool  
    EnableDeviceNotifications(  
        const GUID& InterfaceClassGuid  
    );
```

Parameter

InterfaceClassGuid

Specifies the class of device interfaces which will be registered with the system to post PnP notifications to this object.

Return Value

The function returns true if successful, false otherwise.

Comments

Call this function once for each device interface class that should be registered by this **CPnPNotifier** object. When a PnP event occurs for one of the registered interface classes then the operating system posts a **WM_DEVICECHANGE** message to this object and the object calls **CPnPNotifyHandler::HandlePnPMessage** in the context of its internal worker thread.

CPnPNotifier::Initialize needs to be called before this function can be used.

See Also

CPnPNotifier::Initialize (page 48)

CPnPNotifier::DisableDeviceNotifications (page 52)

CPnPNotifyHandler::HandlePnPMessage (page 45)

CPnPNotifier::DisableDeviceNotifications

Disables notifications for a given class of device interfaces.

Definition

```
bool  
DisableDeviceNotifications(  
    const GUID& InterfaceClassGuid  
);
```

*Parameter***InterfaceClassGuid**

Specifies the class of device interfaces which will be unregistered so that no further PnP notifications will be posted to this object.

Return Value

The function returns true if successful, false otherwise.

Comments

After this call the **CPnPNotifier** object will stop to issue **CPnPNotifyHandler::HandlePnPMessage** calls for the specified device interface class.

It is safe to call this function if no notifications are currently registered for the specified device interface class. The function succeeds in this case.

A call to **CPnPNotifier::Shutdown** will disable all device notifications that are currently registered.

See Also

CPnPNotifier::EnableDeviceNotifications (page 51)

CPnPNotifier::Shutdown (page 50)

6.2 Port Information

CPortInfo class

The **CPortInfo** class defines an interface that is used to get information about available virtual COM ports created by the CDC ACM driver. The COM port name can be requested either with an index or with the device path received by the PnPNotifier class.

Member Functions

CPortInfo::EnumeratePorts

This function creates a internal list of all available virtual COM ports created by the CDC ACM driver.

Definition

```
DWORD  
EnumeratePorts(  
    const GUID* DriverInterface  
);
```

Parameter

DriverInterface

The **DriverInterface** parameter contains the **DriverUserInterfaceGuid** that is defined in the INF file of the CDCACM driver with the line
HKR,,DriverUserInterfaceGuid,%REG_SZ%,"40994DFA-45A8-4da7-8B58-ACC2D7CEA825".

This GUID must be modified during the customization of the driver. This makes sure that the notifiator finds exactly the requested devices. Please create a new GUID and replace it in the INF file and in your application.

Return Value

The function returns 0 on success or a windows error message.

Comments

The internal list of devices is created each time this function is called. The internal list contains the devices that are available while this function is called. The list is destroyed if the destructor of the class is called. If you want to find the COM port name of a removed device instance, you need a list that was created while the device was connected. This function must be called before all other functions of this class.

See Also

CPortInfo::GetPortCount (page 54)

CPortInfo::GetPortInfo (page 54)

CPortInfo::GetPortInfoByDevicePath (page 55)

CPortInfo::GetPortCount

This function returns the number of available virtual COM ports.

Definition

```
DWORD  
GetPortCount( );
```

Return Value

The function returns the number of available virtual COM ports.

Comments

It returns 0 if the function **EnumeratePorts** was not called successful or if no devices are connected.

See Also

CPortInfo::EnumeratePorts (page 53)

CPortInfo::GetPortInfo

This function returns the COM port information to an index.

Definition

```
DWORD  
GetPortInfo(  
    DWORD Index,  
    PortInfoData* PortInfo  
);
```

Parameters

Index

The **Index** parameter contains the zero based index to the internal device list created with **EnumeratePorts**. The function **GetPortCount** can be used to determine the number of available ports. Or this function can be called until the Windows error **ERROR_NO_MORE_ITEMS** is returned.

PortInfo

The **PortInfo** parameter contains a user provided data structure that receives the COM port information.

Return Value

The function returns 0 on success or a windows error message.

Comments

The index is related to the device list. As long as the function **EnumeratePorts** is not called the index returns always the same string. The relation between the index and COM port numbers can be assigned in a different way if the function **EnumeratePorts** is called.

See Also

[CPortInfo::EnumeratePorts](#) (page 53)
[CPortInfo::GetPortCount](#) (page 54)
[CPortInfo::GetPortInfoByDevicePath](#) (page 55)
[PortInfoData](#) (page 56)

CPortInfo::GetPortInfoByDevicePath

This function returns the COM port name to a device path.

Definition

```
DWORD  
GetPortInfoByDevicePath(  
    char* Path,  
    PortInfoData* PortInfo  
);
```

*Parameters***Path**

The **Path** parameter contains the device path as a zero terminated string. This string can be obtained in the PnP handler.

PortInfo

The **PortInfo** parameter contains a user provided data structure that receives the COM port information.

Return Value

The function returns 0 on success or a windows error message.

Comments

The function fails with the Windows error code **ERROR_NO_MORE_ITEMS** if the device path is not in the list of the devices.

See Also

CPortInfo::EnumeratePorts (page 53)

CPortInfo::GetPortCount (page 54)

CPortInfo::GetPortInfo (page 54)

PortInfoData (page 56)

PortInfoData

This structure contains information about the COM port.

Definition

```
typedef struct tagPortInfoData{
    DWORD Index;
    char PortName[MAX_PORT_NAME_SIZE];
    char SerialNumber[MAX_SERIAL_NUMBER_SIZE];
} PortInfoData;
```

*Members***Index**

Contains the index of the current device list. The index may change if the function **EnumeratePorts** is called.

PortName[MAX_PORT_NAME_SIZE]

Returns the port name, e.g. COM3. The string is zero terminated.

SerialNumber[[MAX_SERIAL_NUMBER_SIZE](#)]

Returns the serial number of the device. This can be the serial number that is reported from the device as a string descriptor with the index `iSerialNumber`. If the device does not support such serial number Windows creates a unique number for the device. The Windows created number changes if the device is connected to a different USB port. The string is zero terminated.

Comments

This structure is used to get information about a COM port.

See Also

[CPortInfo::GetPortInfo](#) (page 54)

[CPortInfo::GetPortInfoByDevicePath](#) (page 55)

6.3 PnP Notification Demo Application

The source code of this application can be found under `source\notifyapp`. It is designed as a console application to keep the code simple.

The class **CPortNotifier** is derived from **CPnPNotifyHandler** and **CPnPNotifier**. It implements the function that receives the **WM_DEVICECHANGE** messages. The main application has a helper function that prints all available virtual COM ports on the console window. The main application is very simple. It initialize the notifier class and enables notifications to the vendor defined interface. Than it prints the initial state of available virtual COM ports. If a PnP event occurs the notification handler calls also the print function that displays a list of all available devices. A real application should start the communication to the device if a device arrival is detected.

7 Source Code Package

The source code is not part of the normal distribution. The source code can be licensed separately from Thesycon.

7.1 Translation

- Unpack all files from the archive to your hard disk. Make sure that the path to the files does not contain spaces.
- Install the Windows Driver Kit (WDK) build 6000. Make sure that the path to the installation folder does not contain spaces.
- Edit the line `DDK_basedir=C:\WinDDK\6000` in the file `source\setdirs.cmd`. Set the path to the installation folder of your WDK.
- Open the project file `.\cdcacmpo.sln` with Visual Studio. The driver can be built in the debug and release version as well as in a x86 (32 bit) and a x64 (64 bit) versions. The executables can be found in the folders

```
bin\fre\i386
bin\chk\i386
bin\fre\amd64
bin\chk\amd64
```

7.2 Structure of the Source Code

The source code is based on class framework. The framework is implemented in the folders `libkn` and `libtb`. These classes contain an abstraction of the WDM object model. The implementation of the driver function is placed in the folder `cdcacmpo`. The following classes are used to implement the driver function.

7.2.1 Driver

This class abstracts the driver object. It is called when the driver is loaded and if a new device is connected. The function `OnAddDevice` creates a new instance of the `Device` class.

7.2.2 Device

This class represents the device and stores all states to maintain the device. It has a set of virtual functions to handle the requests from the API. These functions are

- `OnDispatchCreate()` ;
`OnDispatchCleanup()` ;
`OnDispatchClose()` ;

```
OnDispatchDeviceControl();  
OnDispatchInternalDeviceControl();  
OnDispatchRead();  
OnDispatchWrite();
```

Other functions are used to handle the plug and play management and to handle the power management.

7.2.3 Stream Classes

The Stream and the StreamIn and StreamOut classes implement the buffer handling with error recovery on the USB interface. They use the helper classes UsbBuf and UsbBufPool. The UsbBuf contains the data structures of a USB buffer. The class UsbBufPool is a container that stores the buffer.

7.2.4 SerQueue

This class contains the queue for read and write operations. It handles the timeout conditions and uses a circular buffer to store the data.

7.2.5 Helper Classes

Some other classes that are not mentioned here are used for some helper purposes and are not so important for the functionality.

8 Debug Support

8.1 Event Log Entries

If the driver detects a major problem during the startup process or when the COM port is opened it creates an entry in the event log of the system. The event log can be opened with the context menu on 'My Computer' -> Manage -> Event Viewer -> System. The entries are created in the category error.

Common problems are:

8.1.1 Clear Feature Endpoint Halt

The device cannot handle this request. Solution: Set the flag `ClearFeatureOnStart` to 0.

8.1.2 Cannot Create Symbolic Link

The COM port is already used by a different driver. Use the device manager -> Properties -> Port Setting -> Advanced to assign a free COM port to this driver and remove/re-connect the device. The problem occurs on PC where the assignment of COM port is defect. This can happen if registry keys modified manually. A different reason for this problem can be other drivers that do not allocate the COM port numbers in the correct way.

8.1.3 Descriptor Problems

The driver complains if the descriptors of the device does not have the expected USB interfaces. The reported problems depends on the setting `OperationMode`. To solve this modify the setting for the `OperationMode` related to the device interface or use a device with correct USB interfaces.

8.1.4 Demo is Expired

The demo version has an internal clock. If the demo period has been expired the driver fails all operations. To re-enable the driver reboot the PC. This does not occur with the release version.

8.2 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 the 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 in 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. DebugView is a free software that can be downloaded on the Microsoft web page. 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. Reproduce the BSOD again. Transfer the memory dump (typically %SystemRoot%\MEMORY.DMP) to Thesycon for analysis.

The following table summarizes the meaning of the debug bits.

Table 3: Trace Mask Bit Content

Bit	Content
0	Fatal errors
1	Warnings
2	Information
3	Extended information
4	Create symbolic link
7	Device Io control user API

Table 3: (continued)

Bit	Content
8	Data transport
9	read buffer handling
10	Time for timeouts of user requests
11	Wait mask
13	Data dump on USB API
14	Data dump on user API read/write
15	Send control signals
17	Serial state signaling

If you submit a problem description to Thesycon please include the following:

- operating system and Service Pack
- USB host controller (EHC, UH C, OHC) and usage of external USB hubs
- Driver version and driver build (release/debug/demo) that has been used
- used TraceMask if kernel Traces have been recorded
- The actions you have performed to cause the problem

Do not send memory dumps with e-mail. We can provide a FTP account.

9 Related Documents

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/>

Index

- ~CPnPNotifier
 - CPnPNotifier::~~CPnPNotifier, [47](#)
- CPnPNotifier
 - CPnPNotifier::CPnPNotifier, [47](#)
- CPnPNotifier, [47](#)
- CPnPNotifier::~~CPnPNotifier, [47](#)
- CPnPNotifier::CPnPNotifier, [47](#)
- CPnPNotifier::DisableDeviceNotifications, [52](#)
- CPnPNotifier::EnableDeviceNotifications, [51](#)
- CPnPNotifier::Initialize, [48](#)
- CPnPNotifier::Shutdown, [50](#)
- CPnPNotifyHandler, [45](#)
- CPnPNotifyHandler::HandlePnPMessage, [45](#)
- CPortInfo, [53](#)
- CPortInfo::EnumeratePorts, [53](#)
- CPortInfo::GetPortCount, [54](#)
- CPortInfo::GetPortInfoByDevicePath, [55](#)
- CPortInfo::GetPortInfo, [54](#)

DisableDeviceNotifications

- CPnPNotifier::DisableDeviceNotifications, [52](#)

DriverInterface

- Parameter of CPortInfo::EnumeratePorts, [53](#)

EnableDeviceNotifications

- CPnPNotifier::EnableDeviceNotifications, [51](#)

EnumeratePorts

- CPortInfo::EnumeratePorts, [53](#)

GetPortCount

- CPortInfo::GetPortCount, [54](#)

GetPortInfoByDevicePath

- CPortInfo::GetPortInfoByDevicePath, [55](#)

GetPortInfo

- CPortInfo::GetPortInfo, [54](#)

HandlePnPMessage

- CPnPNotifyHandler::HandlePnPMessage, [45](#)

hInstance

- Parameter of CPnPNotifier::Initialize, [48](#)

Index

- Member of PortInfoData, [56](#)
- Parameter of CPortInfo::GetPortInfo, [55](#)

Initialize

- CPnPNotifier::Initialize, [48](#)

InterfaceClassGuid

- Parameter of CPnPNotificator::DisableDeviceNotifications, [52](#)
- Parameter of CPnPNotificator::EnableDeviceNotifications, [51](#)
- lParam
 - Parameter of CPnPNotifyHandler::HandlePnPMessage, [46](#)
- NotifyHandler
 - Parameter of CPnPNotificator::Initialize, [48](#)
- Path
 - Parameter of CPortInfo::GetPortInfoByDevicePath, [56](#)
- PortInfo
 - Parameter of CPortInfo::GetPortInfoByDevicePath, [56](#)
 - Parameter of CPortInfo::GetPortInfo, [55](#)
- PortInfoData, [56](#)
- PortName[MAX_PORT_NAME_SIZE]
 - Member of PortInfoData, [56](#)
- SerialNumber[MAX_SERIAL_NUMBER_SIZE]
 - Member of PortInfoData, [57](#)
- Shutdown
 - CPnPNotificator::Shutdown, [50](#)
- uMsg
 - Parameter of CPnPNotifyHandler::HandlePnPMessage, [46](#)
- wParam
 - Parameter of CPnPNotifyHandler::HandlePnPMessage, [46](#)