Thesycon® Systemsoftware & Consulting GmbH

USB CDC/ACM Class Driver

For Windows 2000, XP, Vista, 7 and 8

Reference Manual

Version 2.0 **October 01, 2012**

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

Disclaimer

Trademarks

Contents

1 Introduction

The CDCACM driver is a generic device driver for Windows. It creates a virtual serial COM port interface and is capable of handling different USB protocols. See section 3.1 on page 16 for details on the supported protocols. The device driver supports USB 2.0 with the operating speeds full and high speed. It is tested on XHC USB 3.0 controllers in high speed mode and it is designed to work with super speed devices.

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 2.0, 3.0 and with common aspects of Win32-based application programming.

2 Overview

2.1 Platforms

The CDCACM driver package contains 32 bit and 64 bit versions. The driver supports the following operating system platforms:

- Windows 8
- 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

2.2 Features

The CDCACM driver provides the following features:

- USB Support. The CDCACM driver supports USB 3.0, USB 2.0 and USB 1.1. It supports full, high and super speed mode. See section 2.4 for more details on USB 3.0 and super speed support.
- 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 is assigned automatically.
- Static COM Port. Optionally the virtual COM port can support a static COM port behavior to support legacy applications. The application can keep the COM port open while the device is removed and continue communication if the device is reconnected.
- USB Protocols. Different USB protocols are supported.
- Asynchronous Data Transfer. Full support for asynchronous (overlapped) data transfer operations.
- Error Correction. Enhanced error correction is implemented.
- Windows CE. The CDCACM class driver can handle devices based on Windows CE and Windows Mobile.
- Plug&Play. The CDCACM class driver fully supports Plug&Play. With the help of a GUID based interface the driver signals add and remove notifications for applications. The PnP Notificator application demonstrates the usage of this interface.
- 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 serial port instance will be created for each CDCACM instance. Thesycon offers a multi-interface driver, which is required to build an individual device node for each interface. For more information, go to http://www.thesycon.de USB Multi Interface Driver.
- Multiple USB Devices. Multiple USB devices can be controlled by the driver at the same time
- Customizing. The CDCACM allows vendor- and product-specific adaptations.
- Installation/Un-installation. For customized driver installations Thesycon offers the PnP Driver Installer. Additional information are available at **http://www.thesycon.de/pnpinstaller**.
- 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 support

The CDCACM device driver supports USB 2.0 and the Enhanced Host Controller on Windows 2000, Windows XP, Windows Vista, Windows 7 and Windows 8. However, CDCACM must 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 USB driver stacks provided by third parties.

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 must ensure that these drivers are installed.

2.4 USB 3.0 Support

Currently more and more computers with Extended (USB3.0) Host Controllers (XHC) come to the market. The Extended Host Controller also handles full and high speed data traffic. If a customer connects your CDCACM device to a USB 3.0 host controller connector it will runs with an Extended Host Controller and the installed bus driver stack.

Microsoft has released the USB 3.0 bus driver for XHC Controllers with Windows 8. Other vendors of USB 3.0 controllers provide their own driver stacks for older operating systems. It is nearly impossible to test with all of these controllers and versions of USB driver stacks.

Thesycon tests using the Renesas and Intel controllers with the vendor-provided driver stacks on Windows 7 and the Microsoft-provided driver stack on Windows 8. The CDCACM driver can also work with USB 3.0 controllers from other vendors, but Thesycon cannot give a warranty that this works without problems.

Today most devices with a USB 3.0 super speed interface are mass storage devices. They use system-provided drivers. Other devices with a USB 3.0 interface are rare on the market. For that reason it is difficult to make comprehensive tests with USB 3.0 devices at the moment. The CDCACM driver is also designed to support USB 3.0 devices with super speed. However because of the test situation, Thesycon cannot give warranty that it works in every case. If you intend to plan a project using a USB 3.0 device please contact Thesycon (www.thesycon.de).

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.

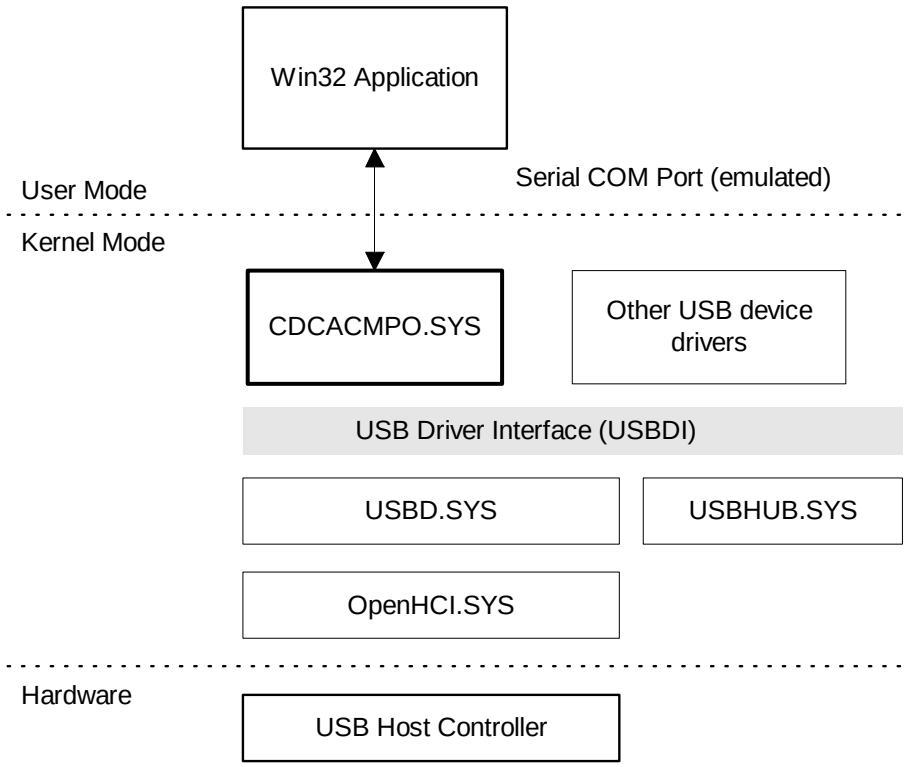

Figure 1: USB Driver Stack

The following modules are shown in Figure 1:

- The USB Host Controller is the hardware component that controls the Universal Serial Bus. It also contains the USB Root Hub. There are implementations for super speed with XHC controllers, high speed with EHC controllers and full speed with UHC and OHC controllers.
- 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), EHCI (Enhanced Host Controller Interface) or XHCI (Extended Host Controller Interface). The driver used depends on the main board chip set of the computer.
- USBD.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 provided by the operating system for use by USB device drivers is called the 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 CDCACM Driver USB protocols

The CDCACM 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 the IN and one in the 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 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 byte. 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&Play drivers. That means the driver is loaded when the device is connected and it is unloaded when the device is disconnected. A Plug&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&Play notifications.

An application that is not Plug&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 remains invalid. Such an application must be closed and restarted to re-enable the communication with a Plug&Play driver.

The CDCACM 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 4.6.1.

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 unexpected behavior if the driver is updated. The update process may copy a new driver to the hard disk and install it. However 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 CDCACM driver is loaded as a Plug&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 always uses 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 bandwidth if no application is interested in receiving 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 this request. In CDC/ACM mode or in 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 WDK and may be different than in other implementations of serial drivers. The PC application must 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 retrieds of the host controller do not transfer the data block

correctly, the PC driver detects the error and starts error handling.

The idea of enhanced error recovery is to perform error handling without any data loss. To perform 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 always send 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 selected is large, the device needs more memory. If the logical buffer size selected is 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 in the logical buffer so far 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 successful.

Error recovery works in the following way. Lets 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 the 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 as a CRC error. In the 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 data loss. It is recommended that an application registers for Power Management messages and stops the data transfer before the PC enters a suspended 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 does not have to 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 GUID is set in the file set_vars.cmd. See section 4.1 for details.

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 officially 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 two drivers may try to create a COM port with the same number. A second driver will not be able to do this because the link name is a global resource on Windows. The second driver that tries to create the link will fail 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 always assigns 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 will fail.

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 from Windows XP SP3 and later can handle the Interface Association Descriptor IAD descriptors. Please provide a IAD in your device descriptors for multi-interface device.

When you have the need to support Windows XP SP2 and earlier with a multi-interface device, 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 Overview

The CDCACM driver supports numerous features that enable a licensee to create a customized device driver package. A driver package which is shipped to end users **must be** customized. This is required in order to avoid potential conflicts with other products of other vendors that also use this driver. See also section 4.2 for a discussion on the reasons why a full driver customization is absolutely required.

Customization includes

- Modification of file names of all driver executables,
- Modification of text strings shown in the Windows user interface,
- Definition of unique software interface identifiers,
- Adaptation of driver parameters to fit the attached device.

Windows Vista and higher Windows versions have a new feature to verify a vendor of a software component. The vendor can add a digital 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.
Please note that it is not possible to install a driver without a digital signature on Windows 8 64-bit.
To get more information about code signing, please refer to the document "Kernel-Mode Code Signing Walkthrough" available on the Microsoft web site.
To add a digital signature to a software component, the vendor must own a Digital ID that supports the Microsoft Authenticode technology (sometimes called Microsoft Authenticode Digital ID). Such a Digital ID may be purchased from several certification authorities (CA), including Symantec (formerly Verisign). For details you may refer to
https://www.symantec.com/verisign/code-signing/microsoft-authenticode

4.2 Reason for Driver Customization

Thesycon's CDCACM driver is generic with respect to concrete products. The driver can be used (and shipped together) with many different products from various vendors. Any generic driver must be customized. Customization includes modification of USB VID and PID, choice of unique file names, assignment of unique identifiers (GUIDs) and modification of display names.

If no (or an incomplete) customization is done, the following situation can occur in the field:

1. User buys product A and installs it. Product A works.
2. Same user buys product B (from another vendor) and installs it. B ships with the same (non-customized) driver as A.

This situation results in a couple of potential problems:

- B driver .sys files overwrite A driver .sys files which reside in the Windows drivers directory. This can cause product A to stop working e.g. because a different driver version gets loaded now.
- Product A control panel detects a product B device and tries to work with it.
- Product B control panel detects a product A device and tries to work with it.
- Un-install of product A removes .sys files (and possibly other files) which causes product B to become non-functional.
- Un-install of product B removes .sys files (and possibly other files) which causes product A to become non-functional.

Important: To ship the driver to end users, a customized driver package must be created. Never ship the original driver package provided by Thesycon to end user!

4.3 Preparing Driver Package Builder

Thesycon provides a set of batch scripts and tools that generate driver packages and all required customization files automatically. This tool set is called Driver Package Builder.

Before the scripts can be used, the following steps are to be executed on the build machine:

1. Install SignTools Package.
   The following Microsoft tools are required:
   - signtools.exe (part of WDK Windows 7)
   - inf2cat.exe (part of WDK Windows 8)
   - cross certificate files (available on the Microsoft Web Site)

   For your convenience Thesycon has collected the required tools in a SignTools package installer. This driver package requires SignTools version 1.5.0. To get a free copy of this SignTools installer, contact Thesycon:
   info(at)thesycon(dot)de.

   The SignTool package will install all required tools into a directory of your choice and set the SIGNTOOLS environment variable to point to this directory.

2. Validate your Vendor Certificate
   Check if your Vendor Certificate is present under the Personal folder by launching the Certificates Microsoft Management Console. To do this, click Start - Run and enter
   `certmgr.msc`.

   The Personal folder should look as shown in figure 2.
   Please contact the publisher of your Vendor Certificate for information how to install the certificate.

   To verify whether the certificate is valid for Code Signing double-click on the certificate to show Certificate Information. The Certificate Information should look as shown in figure 3. For code signing it is required that the private key that corresponds to the certificate is available. This is indicated by the key sign at the bottom of the Certificate Information General page.

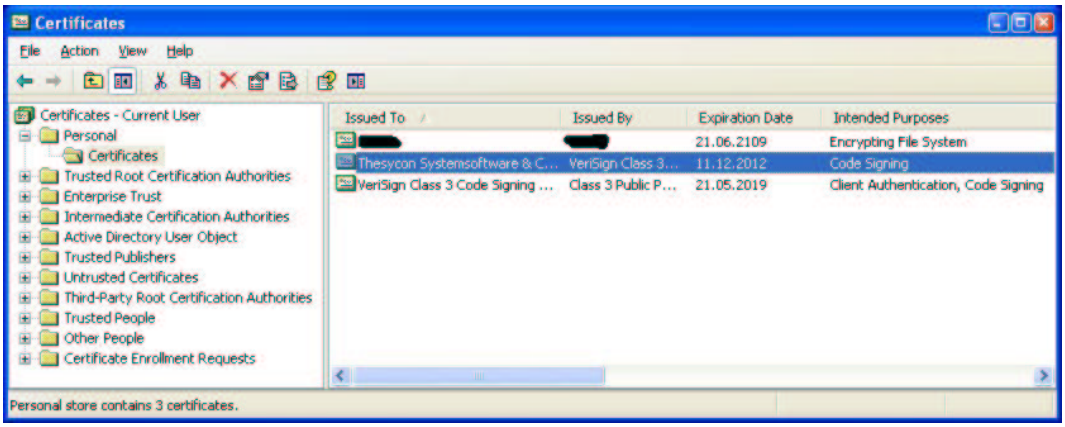

Figure 2: Certificates Microsoft Management Console

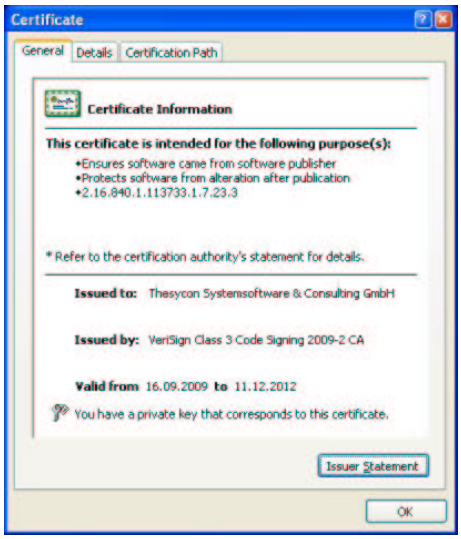

Figure 3: Certificate Information

3. Set Certificate Variables
   Edit the set_vendor_certificate.cmd script (located in the signing subdirectory of the driver kit) to specify the vendor certificate imported in the previous step. Set the following variables according to your certificate:

   - VENDOR_CERTIFICATE is the "issued to" name of the certificate as shown in certmgr.msc. This needs to be set to the correct name.
   - CROSS_CERTIFICATE is the file name of the cross certificate for your certificate provider. The default is correct for a VeriSign certificate.
   - SIGNTOOL_TIMESTAMP_URL is the URL of a trusted timestamp server. The default is correct for a VeriSign certificate.

   For details, please refer to the comments in set_vendor_certificate.cmd.

4. Prepare for GUID Generation
   GUIDs are generated using the guidgen.exe tool provided by Microsoft. The guidgen.exe

tool is part of Microsoft Visual Studio 2005 and Microsoft Visual Studio 2008. Alternatively, the tool can be downloaded at
http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=17252

4.4 Using Driver Package Builder

The Driver Package Builder batch scripts are located in the CustomPackageBuilder subdirectory of the driver kit.

To create your own customized driver package, please follow the steps below.

1. Driver Package Configuration
   The **`set_vars.cmd`** file in this directory serves as the configuration file for your driver package. Edit **`set_vars.cmd`** to customize your driver package. For details, please refer to section 4.6.

2. Driver Parameter Configuration
   The file **`driver_parameters.inc`** contains the driver parameters. You can modify the parameters to fit the driver behavior to your device. See section 4.6.1 for details.

3. Build Driver Package
   Open a Windows console in the CustomPackageBuilder directory and run the script create_drvpackages.cmd. Make sure your PC is connected to the Internet to get the certified time stamp from your certificate provider. This ensures that your digitally signed driver package is still valid after your signing key has been expired.

   The driver package will be created in the subfolder **`driver_package`**. If the prerequisites to create a digitally signed driver package are not given the script prints warning messages in the console. If you continue the script execution the script generates a non-digitally signed driver package in the folder **`driver_package_notsigned`**.

   A unsigned driver package cannot be used or installed on Windows 8 64-bit. Other systems complain during the installation with a warning box that the origin of the driver cannot determined and it is a potential risk to install the software. Do not ship unsigned driver packages to your customers.

   It is strongly recommended to check the INF files of the created driver package using the script check_inf_files.cmd. The script opens a .html file with the results of the check.

To create different packages make a copy of the folder CustomPackageBuilder and configure the second driver package in the copied folder.

Modifications to the driver package make the digital signature invalid. When manual changes to the INF file are required the digital signature can be updated with the script **`update_signature.cmd`**. Give the relative folder name of the modified driver package as a command line argument to the script. Keep in mind that the script **`create_driverpackages.cmd`** overwrites the modified driver package without warning. For that reason it is recommended to move the driver package to a different folder before manual changes are done.

4.5 Test Unsigned Drivers

Drivers that does not have a digital signature cannot be loaded on Windows 7 and Windows 8 x64 systems. Windows 7 loads the driver when the .sys file has a digital signature while Windows 8 requires a valid digital signature on the .cat file. The Driver Package Builder puts a signed .sys file to the non signed driver package. This allows you to test the driver package on Windows 7 x64 systems.

On both systems a not signed driver can be loaded when the "Driver Signature Enforcement" is disabled in the "Advanced Boot Options". On Windows 7 these boot options can be entered by pressing F8 during the boot process. On Windows 8 a spcial boot menu can be started with the command line "Shutdown.exe /r /o" or by holding the Shift Key while pressing Restart. For more details see the Windows documentation.

Note: The driver signing enforcement is turned off only for one boot process. This mehtod should be used only on development PCs.

4.6 Parameters Reference

VENDOR_NAME
The vendor of the driver package. This string is used as provider and manufacturer string (.inf file).

PRODUCT_NAME
The name of the product that uses the driver package.

DRIVER_NAME_BASE
The common part of the name of the driver package files. **This string must not include spaces or special characters.** This name should be unique in the world. You should add an abbreviation of your company name to the driver base name.

DRIVER_INTERFACE_GUID
Unique identifier of type GUID for the driver interface. Create a fresh GUID using guidgen.exe. **It is very important to use a fresh GUID here.**

INF_VID_PID_[1..16]
USB Vendor ID and Product ID of the USB devices that are supported by the driver package.
Format: VID_xxxx&PID_yyyy or VID_xxxx&PID_yyyy&MI_zz
A maximum of four different VID/PID pairs can be specified. If the driver is installed on a multi-interface architecture zz identifies the interface number.

INF_VID_PID_[1..16]_DESCRIPTION
Display name of the USB devices that are supported by the driver package. This name is displayed by the Windows Device Manager. A display name can be specified for each VID/PID pair.

4.6.1 Customizing Default Driver Settings

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

```
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 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 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 independently of the buffer size used in the device. It does not work with 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 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 a selected value 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 a selected value 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 1. On later operating systems 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 sent 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 the logical or-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 its 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 a static device object for the COM port. This means an application can keep the COM port open 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 over 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 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. However, 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 'thcdcacm'.

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.

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 manual installation of the driver should only be used by developers.

In order to install the CDCACM driver manually you have to create a customized driver package first. See section 4.1 on page 25 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.
- 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, a system-provided device driver will be installed silently. A USB mouse or a USB keyboard are examples for such devices. The operating system will not ask for a driver because it will find a matching entry for the device in its internal INF file data base.

  You must 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, which is similar to the New Hardware Wizard described above, will start. 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. Use any other terminal program.

5.2 Uninstalling CDCACM manually

To un-install 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 un-install the CDCACM device driver:

- Remove the selected device from the system by clicking the button "Un-install". 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 un-installation 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 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.

5.3.1 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 interactive mode with graphical user interface or it can be run in 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 silently. 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 later connected to the PC the certified driver is installed silently. 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. The exact driver version from the installer package is installed. This form of driver updating 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 match the USB VID and PID are removed from the system. The result is a system that behaves as if 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**.

5.3.2 Installing the CDCACM Driver with DIFx

The DIFx software has some limitations that are explained in this section. For that reason, installing the device with this tool is 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 only introduces the Driver Package Installer. For details on 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 un-install 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 non-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 on whether the device connected during the installation process or not. It is not certain that the driver from the installation package will be installed for the device.
- After un-installing a driver, the installer activates another 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 un-installing depends on 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 mechanism used 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&Play Notificator

The Plug&Play Notificator consists out of the two classes **`CPnPNotificator`** and **`CPnPNotifyHandler`**. It can be used to register to a GUID based interface. In the background it creates an invisible Window and a thread. Both are 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 such as 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&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 must be implemented by a class that is derived from **CPnPNotifyHandler**. A **CPnPNotificator** object calls this function in the context of its internal worker thread when the system issued a **WM_DEVICECHANGE** message for one of the device interface classes registered with **CPnPNotificator::EnableDeviceNotifications**.

Caution: MFC is not aware of the internal worker thread created by a **CPnPNotificator** 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

CPnPNotificator (page 41)
CPnPNotificator::Initialize (page 42)
CPnPNotificator::EnableDeviceNotifications (page 45)

CPnPNotificator class

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

Member Functions

CPnPNotificator::CPnPNotificator

Constructs a CPnPNotificator object.

Definition

```
CPnPNotificator();
```

CPnPNotificator::~CPnPNotificator

Destroys the CPnPNotificator object.

Definition

```
~CPnPNotificator();
```

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 create a hidden window. The system will post a **WM_DEVICECHANGE** message to this window if a Plug&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

CPnPNotificator::Shutdown (page 44)
CPnPNotificator::EnableDeviceNotifications (page 45)
CPnPNotificator::DisableDeviceNotifications (page 46)
CPnPNotifyHandler::HandlePnPMessage (page 39)

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 42)
CPnPNotificator::EnableDeviceNotifications (page 45)
CPnPNotificator::DisableDeviceNotifications (page 46)

CPnPNotificator::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 **CPnPNotificator** 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.

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

See Also

CPnPNotificator::Initialize (page 42)
CPnPNotificator::DisableDeviceNotifications (page 46)
CPnPNotifyHandler::HandlePnPMessage (page 39)

CPnPNotificator::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 **CPnPNotificator** 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 **CPnPNotificator::Shutdown** will disable all device notifications that are currently registered.

See Also

CPnPNotificator::EnableDeviceNotifications (page 45)
CPnPNotificator::Shutdown (page 44)

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 PnPNotificator 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 notificator 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 48)
CPortInfo::GetPortInfo (page 48)
CPortInfo::GetPortInfoByDevicePath (page 49)

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 47)

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 47)
CPortInfo::GetPortCount (page 48)
CPortInfo::GetPortInfoByDevicePath (page 49)
PortInfoData (page 50)

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 47)
CPortInfo::GetPortCount (page 48)
CPortInfo::GetPortInfo (page 48)
PortInfoData (page 50)

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 a 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 48)
CPortInfo::GetPortInfoByDevicePath (page 49)

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 **`CPortNotificator`** is derived from **`CPnPNotifyHandler`** and **`CPnPNotificator`**. 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 initializes the notificator class and enables notifications to the vendor defined interface. It than 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 from Thesycon separately.

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 build 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 functionOnAddDevice 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&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 are 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 the debug version of the driver. The debug version is not part of the demo package. During the customization a debug driver package in the sub-folder dbg is created. 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 customized debug driver. Reboot the PC if the system request a reboot.

- Open with the registry editor the following path:

  `\HKEY_LOCAL_MACHINE\System\CurrentControlSet\services\DRIVER_NAME_BASE(_x64)\`.

  DRIVER_NAME_BASE is set during customization of the 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 TraceMask must be set to 0x00000020. The TraceMask contains the logical or-value of all active message bits.

- Disconnect all devices or reboot the PC to make sure the debug 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. Start Debug View with administrator privileges. Enable the Capture options "Capture Kernel" and "Enable Verbose Kernel Output".

- 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
8	Data transport
9	read buffer handling
10	Time for timeouts of user requests

Table 3: (continued)

Bit	Content
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 (XHC, EHC, UHC, 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