Thesycon® Systemsoftware & Consulting GmbH

USBIO

USB Software Development Kit for Windows

For Windows 2000, XP, Vista, 7 and 8

Reference Manual

Version 2.71 **November 14, 2012**

Thesycon® Systemsoftware & Consulting GmbH

Werner-von-Siemens-Str. 2 · D-98693 Ilmenau · GERMANY

Tel: +49 3677 / 8462-0

Fax: +49 3677 / 8462-18

http://www.thesycon.de

Copyright (c) 1998-2012 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 license.

Trademarks

The following trade names are referenced throughout this manual:

Microsoft, Windows, Win32, Windows NT, Windows XP, Windows Vista, Windows 7, Windows 8 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

1 Introduction

USBIO is a generic Universal Serial Bus (USB) device driver for Windows. It is able to control any type of USB device and provides a convenient programming interface that can be used by Windows 32/64 bit applications. The USBIO device driver supports USB 1.1 and USB 2.0.

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

Note that there is a high-level programming interface for the USBIO driver that is based on Microsoft's COM technology. The USBIO COM Interface is included in the USBIO Development Kit. For more information refer to the USBIO COM Interface Reference Manual.

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 Windows 32/64 bit based application programming.

2 Overview

Current Windows operating systems have built-in supported for the Universal Serial Bus (USB). These systems include device drivers for the USB Host Controller hardware, for USB Hubs, and for some USB device classes. The USB device drivers provided by Microsoft support devices that conform with the appropriate USB device class definitions made by the USB Implementers Forum. USB devices that do not conform to one of the USB device class specifications, e.g. a new device class or a device under development, are not supported by device drivers included with the operating system.

In order to use devices that are not supported by the operating system itself, the vendor of such a device is required to develop a USB device driver. This driver has to conform to the Windows Driver Model (WDM) that defines a common driver architecture for Windows 2000, Windows XP, Windows Vista, Windows 7 and Windows 8. Writing, debugging, and testing of such a driver means considerable effort and requires a lot of knowledge about development of kernel mode drivers.

By using the generic USB device driver USBIO it is possible to get any USB device up and running without spending the time and the effort of developing a device driver. This might be especially useful during the development or testing of a new device, but in many cases it is also suitable to include the USBIO device driver in the final product. This means there is no need to develop and test a custom device driver for the USB-based product.

2.1 Platforms

The USBIO 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

Note that Windows NT 4.0 and Windows 95 are not supported by USBIO. Windows 98 and Windows ME are no longer supported by this version. If you need support for these older operating systems, ask Thesycon for a solution.

2.2 Features

The USBIO driver provides the following features:

- OS Support. The USBIO driver supports multiple Windows operating systems with the same package. A detailed list of supported operating systems is included in section 2.1 (page 21).
- 64-Bit Support. The USBIO driver supports all 64 bit Windows operating systems.
- USB Support. The USBIO driver supports USB 1.1 and USB 2.0 devices in low, full and high-speed mode. The USBIO driver also supports USB 2.0 devices that are connected to a USB 3.0 host controller with the Renesas bus driver. For details on USB 3.0 support please see section 2.7 at page 27.
- USB Transfer Types. The USBIO driver supports the USB Control, Interrupt, Bulk and Isochronous transfer types.
- USB Device Endpoints. The USBIO driver provides an interface similar to the Win32 file I/O interface to USB endpoints (pipes).
- Asynchronous Data Transfer. Fully supports asynchronous (overlapped) data transfer operations.
- Plug&Play. The USBIO driver fully supports hot Plug&Play. It also supports Plug&Play notifications for applications.
- Power Management. The driver supports the Windows power management model.
- Applications. The USBIO driver provides an interface to USB devices that can be used by any Windows 32/64 bit application.
- Multiple Applications. Multiple applications can use the USBIO at once.
- Multiple USB Configurations. The USBIO driver can be used with devices that implement multiple USB configurations. It also supports switching between different USB configurations.
- Multiple USB Interfaces. The USBIO driver can be used with devices that implement multiple USB interfaces. In this case a multi-interface driver is required. As an alternative to the Windows Multi-Interface Driver, Thesycon offers its own USB Multi-Interface Driver. Additional information is available at **http://www.thesycon.de/usbmultiinterface**.
- Multiple USB Devices. Multiple USB devices can be controlled by USBIO at once.
- Programming Interfaces. The USBIO provides a Windows programming interface for use in C, C++ and Java programs. A high-level, COM-based programming interface for use in Visual Basic and .NET based programs is also provided.
- FX Support. The USBIO driver package supports the firmware download for Cypress FX series ICs.

- Installation/Uninstallation. The USBIO provides an installation wizard that allows a quick and easy installation of the USBIO driver for any kind of USB device. The cleanup wizard supports automated deinstallation of the USBIO driver. The provided wizards are not suitable for customized driver installations. For that, Thesycon offers the PnP Driver Installer. Additional information is available at **http://www.thesycon.de/pnpinstaller**.
- INF Generation. The installation wizard included in the package provides a simply generation of an INF adapted to any connected USB hardware. This file can be used as a starting-point of customization.
- Vendor Specific User Interface. Driver supports vendor specific user interface ID for simple enumeration of vendor devices.
- Customization. The USBIO allows vendor- and product-specific adaptations. For more details look at section 4 (page 39).
- 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.

The USBIO device driver can be used to control any USB device from a Windows 32/64 bit application running in user mode. Examples of such devices are

- telephone and fax devices
- telephone network switches
- audio and video devices (e.g. cameras)
- measuring devices (e.g. oscilloscopes, logic analyzers)
- sensors (e.g. temperature, pressure)
- data converters (e.g. A/D converters, D/A converters)
- bus converters or adapters (e.g. RS 232, IEEE 488).

2.3 Restrictions

Some restrictions that apply to the USBIO device driver are listed below.

- If a particular kernel mode interface (e.g. WDM Kernel Mode Streaming or NDIS) has to be supported in order to integrate the device into the operating system, it is not possible to use the generic USBIO driver. However, in such a situation it is possible to develop a custom device driver based on the source code of the USBIO. Please contact Thesycon if you need support with such a kind of project.
- Although the USBIO device driver fully supports isochronous endpoints, there are some limitations with respect to isochronous data transfers. They result from the fact that the processing of isochronous data streams must be performed by an application running in user mode and there is no guaranteed response time for threads running in user mode. This may be critical for the implementation of some synchronization methods, for example when the data rate is controlled by loop-back packets (see the USB Specification, section 5 for synchronization issues of isochronous data streams).

  It is possible to support all kinds of isochronous data streams using the USBIO driver. However, the delays that might be caused by the thread scheduler of the operating system should be taken into consideration.
- There are some problems caused by the implementation of the operating system's USB driver stack. Thesycon encountered these problems during debugging and testing of the USBIO driver. For some of the problems there are work-arounds built into the USBIO driver. Others do still exist, however,there is no way to implement a work-around.

  Problems known to Thesycon are documented in *Problems.txt*, which is included in the USBIO Development Kit. We strongly recommend referring to this file when strange behavior is observed during application development.

2.4 Customization

Thesycon's USBIO is a generic driver. The driver can be used (and shipped together) with many different hardware products from various vendors. All generic drivers 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.

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 users.

For detailed information on the steps required to create a customized driver package, refer to section 4.

2.5 WHQL certification

The Windows Hardware Quality Labs (WHQL) of Microsoft provide test benches to verify the correct operation of kernel mode drivers. The aim of these tests is to ensure kernel mode drivers with high quality and perfect support for Windows based operating systems. Furthermore, the tests are related to the hardware. This makes sure that the hardware is compliant to existing specifications and can interoperate with the operating system without problems.

The test results of the WHQL test bench can be submitted to Microsoft. After a successful submission Microsoft creates .cat files with a digital signature. Kernel drivers with a digital signature from Microsoft can be installed silently and without warnings during the installation process. Preinstalled drivers can be installed by connecting the device without administrator privileges and without any user interaction. This makes the installation process of the kernel driver much simpler.

The .cat file contains CRC's for the .sys and the .inf files. The .inf file must be customized to support a new device. For that reason Thesycon cannot provide a driver package with a digital signature from Microsoft that can be used by a company which licenses the USBIO.

During the release process the USBIO driver passed the WHQL test bench with success. For that reason a customized driver package based on the USBIO driver should also pass the WHQL tests. Parts of the tests are related to the device operation. For that reason, the device must be compliant to the USB specification to receive successful test results.

To perform WHQL tests, special hardware setup and comprehensive knowledge about the test bench is required. Thesycon can perform the WHQL tests in the name of a customer for a customized driver package. For details ask Thesycon support.

2.6 USB 2.0 support

The USBIO device driver supports USB 2.0 and the Enhanced Host Controller on Windows 2000, Windows XP, Windows Vista, Windows 7 and Windows 8. However, USBIO must be used on top of the driver stack that is provided by Microsoft. Thesycon does not guarantee that the USBIO 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 to the 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.

2.7 USB 3.0 Support

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

Microsoft has released the USB 3.0 bus driver for XHCs 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 USBIO 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 this moment in time. The USBIO driver is 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).

2.8 USB over Ethernet Support

There are USB emulations over Ethernet on the market. These solutions provide the API of the USB bus driver with a third party driver. Thesycon does not guarantee that the USBIO runs with these solutions.

2.9 Selecting an Appropriate Programming Interface

The USBIO Development Kit supports the following programming interfaces:

- Standard C interface
- C++ interface
- Native Java interface
- Visual Basic Interface using USBIOCOM

The following table summarize the features of the interfaces:

Table 1: Features of the USBIO programming interfaces

Interface	Requires	High bandwidth	Source Code of library
Standard C	Driver only	yes	-
C++	USBIOLIB	yes	yes
Java	USBIOJAVA & Classes	yes	yes
Visual Basic COM	USBIOCOM	no	no
C# COM	USBIOCOM	no	no

The following sections give an overview to the different interfaces and enables the user to select the best interface for the application to build.

2.9.1 Standard C Interface

The C interface is directly exported by the USBIO.SYS kernel mode driver and does not need additional libraries. This interface uses the standard Win32 API functions CreateFile(), DeviceIoControl(), ReadFile(), and WriteFile(). The programmer can select whether the file handles are opened in synchronous or overlapped mode. The function call DeviceIoControl requires in most cases an additional data structure that must be prepared to pass the data to the driver. This requires a little bit more code.

This interface allows full low level access to the driver. It does not contain support for buffer pools or threads to archive high performance data transmission. Additional effort is required to add these features to a project. This interface should be selected if the programming language C is required. The documentation of this interface is part of this document.

2.9.2 C++ Interface

The C++ interface is implemented in the USBIOLIB library. The source code of the library is part of the distribution. The library is a static library which must be linked to the project. A different way to use the library is to include the source files into a project.

The C++ interface covers all functions of the driver interface. The classes simplifies the calls to the driver. Unused parameters which are required by the function DeviceIoControl must not be passed to the class methods.

The library implements a buffer pool and threads for high performance data exchange with the device. The programming effort in using this interface is smaller in comparison with the Standard C interface. The interface causes a very small overhead and can be used for devices that require a high bandwidth. The documentation of this interface is part of this document.

2.9.3 Native Java interface

The native Java interface is implemented in the USBIOJAVA.DLL and the Java classes. The source code of both modules is part of this package. The HTML documentation of the Java interface can be opened with the start menu.

This interface is demonstrated by a simple and a complex Java application. The source code of both applications is part of this package.

The Java classes contain buffer pools and threads. The interface allows high performance data exchange with devices.

2.9.4 USBIOCOM for Visual Basic or C#

The USBIOCOM interface is implemented in the USBIOCOM.DLL. The source code of the DLL is not part of the standard package but can be licensed separately. The COM interface covers all functions of the USBIO driver. It contains threads and buffer pools, which can be controlled by interface function calls.

The documentation of this interface is part of this package and can be found in the start menu.

The package includes simple and enhanced examples for Visual Basic and C#. For Visual Basic the data types match the COM data types.

Unfortunately the COM architecture creates a lot of internal threads and copies the data between the application and itself. This causes a higher CPU load in comparison to the native interfaces. It is also possible to reach the full data rate with full speed devices. For high speed devices with a high bandwidth requirement this interface should not be used.

C# uses the garbage collection to destroy objects that are no longer used. The garbage collection can sometimes interrupt the data transfer. It is not recommended to use C# if the PC has to work with near-realtime responsibility.

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.

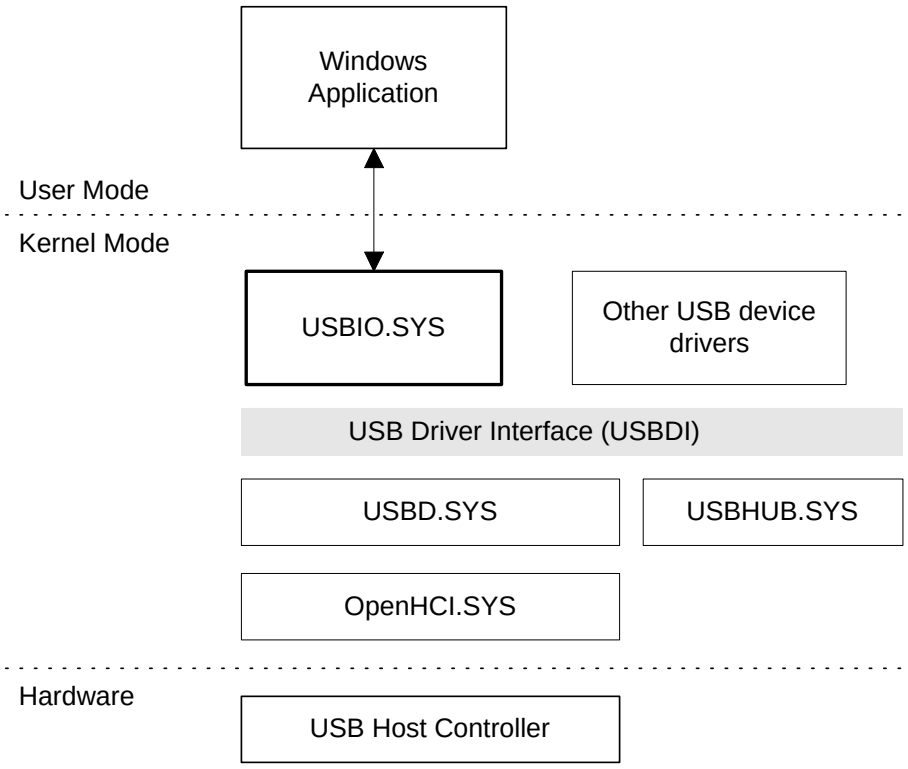

Figure 1: USB Driver Stack

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 1.1: Open Host Controller (OHC) and Universal Host Controller (UHC). There is one implementation of the host controller that supports USB 2.0: Enhanced Host Controller (EHC). The Extended Host Controller (XHC) supports USB 3.0 and all transfer speeds including super speed.
- 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). The driver used depends on the mainboard chip set of the computer. For instance, Intel chipsets contain Enhanced Host Controllers and Universal Host Controllers.
- 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.
- USBIO.SYS is the generic USB device driver USBIO.

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 the WDM. The I/O request packets are passed to the next driver in the stack for processing and returned to the caller after completion.

The USB Driver Interface is accessible only for kernel-mode drivers. Normally, there is no way to use this interface directly from applications that run in user mode. The USBIO device driver was designed to overcome this limitation. It connects to the USBDI at its lower edge and provides a private interface at its upper edge that can be used by Windows 32/64 bit applications. Thus, the USB driver stack becomes accessible to applications. A Windows 32/64 bit application is able to communicate with one or more USB devices by using the programming interface exported by the USBIO device driver. Furthermore, the USBIO programming interface may be used by more than one application or by multiple instances of one application at the same time.

The main design goal for the USBIO device driver was to make all the features that the USB driver stack provides at the USBDI level available to applications. For that reason the programming interface of the USBIO device driver (USBIOI) is closely related to the USBDI. However there is no one-to-one relationship for many of the functions.

3.1 USBIO Object Model

The USBIO device driver provides a communication model that consists of device objects and pipe objects. The objects are created, destroyed, and managed by the USBIO driver. An application can open handles to device objects and bind these handles to pipe objects.

3.1.1 USBIO Device Objects

Each USBIO device object is associated with a physical USB device that is connected to the USB. A device object is created by the USBIO driver if a USB is detected by the Plug&Play Manager of the operating system. This happens when a USB device is connected to the system. The USBIO driver is able to handle multiple device objects at the same time.

Each device object created by USBIO is registered with the operating system by using a unique identifier (GUID, Globally Unique Identifier). This identifier is called "Device Interface ID". All device objects managed by USBIO are identified by the same GUID. The GUID is defined in the USBIO Setup Information (INF) file. Based on the GUID and an instance number, the operating system generates a unique name for each device object. This name should not be interpreted by applications. Additionally, it should never be used directly or stored permanently.

It is possible to enumerate all the device objects associated with a particular GUID by using functions provided by the Windows Setup API. The Functions used for this purpose are:

```
SetupDiGetClassDevs()
SetupDiEnumDeviceInterfaces()
```

`SetupDiGetDeviceInterfaceDetail()`

The result of the enumeration process is a list of device objects currently created by USBIO. Each of the USBIO device objects corresponds to a device currently connected to the USB. For each device object an ambiguous device name string is returned. This string can be passed to `CreateFile()` to open the device object.

A default Device Interface ID (GUID) is built into the USBIO driver. This default ID is defined in USBIO_I.H. Each device object created by USBIO is registered by using this default ID. The default Device Interface ID is used by the USBIO demo application for device enumeration. This way, it is always possible to access devices connected to the USBIO from the demo application.

In addition, a user-defined Device Interface ID is supported by USBIO. This user-defined GUID is specified in the USBIO INF file by the `DriverUserInterfaceGuid` variable. If the user-defined interface ID is present at device initialization time USBIO registers the device with this ID. Thus, two interfaces – a default interface and a user-defined interface – are registered for each device. The default Device Interface ID should only be used by the USBIO demo application. Custom applications should always use a private user-defined Device Interface ID. This way, device naming conflicts are avoided.

Important:
Every USBIO customer should generate their own private device interface GUID. This is done by using the tool GUIDGEN.EXE from the Microsoft Platform SDK or the VC++ package. This private GUID is specified as user-defined interface in `DriverUserInterfaceGuid` in the USBIO INF file. The private GUID is also used by the customer's application for device enumeration. For that reason the generated GUID must also be included in the application. The macro `DEFINE_GUID()` can be used for that purpose. See the Microsoft Platform SDK documentation for further information.

As stated above, all devices connected to USBIO will be associated with the same device interface ID, which is also used for device object enumeration. Because of that, the enumeration process will return a list of all USBIO device objects. In order to differentiate the devices, an application should query the device descriptor or string descriptors. This way, each device instance can be identified unambiguously.

After the application has received one or more handles for the device, operations can be performed on the device by using a handle. If there is more than one handle to the same device, it makes no difference which handle is used to perform a certain operation. All handles that are associated with the same device behave in the same way.

3.1.2 USBIO Pipe Objects

The USBIO driver uses pipe objects to represent an active endpoint of the device. The pipe objects are created when the device configuration is set. The number and type of created pipe objects depend on the selected configuration. The USBIO driver does not control the default endpoint (endpoint zero) of a device, which is instead controlled by the USB bus driver, USBD. Because of that, there is no pipe object for endpoint zero and there are no pipe objects available until the device is configured.

In order to access a pipe the application has to create a handle by opening the device object as described above and attach it to a pipe. This operation is called a "Bind". After a Bind is

successfully established the application can use the handle to communicate with the endpoint that the pipe object represents. Each pipe may be bound only once, and a handle may only be bound to one pipe. This means there is always an one-to-one relation of pipe handles and pipe objects and that the application has to create a separate handle for each pipe it wants to access.

The USBIO driver also supports an "unbind" operation. This is used to delete a binding between a handle and a pipe. After an unbind is performed the handle may be reused to bind another pipe object and the pipe object can be used to establish a binding with another handle.

The following example is intended to explain the relationships described above. In Figure 2 a configuration is shown where one device object and two associated pipe objects exist within the USBIO data base.

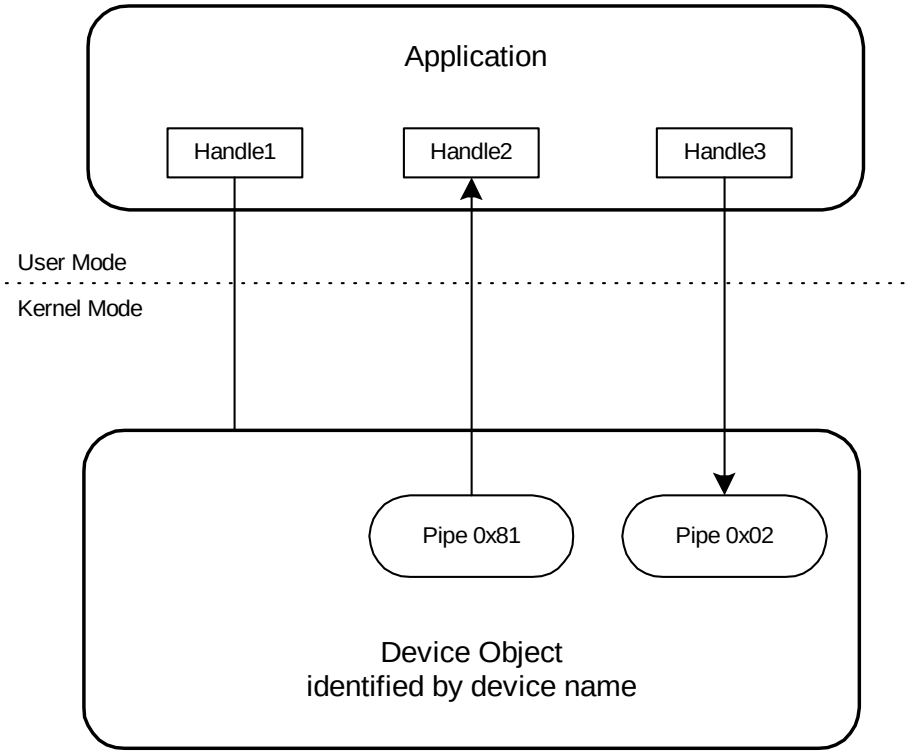

Figure 2: USBIO device and pipe objects example

The device object is identified by a device name as described in section 3.1.1 (page 32). A pipe object is identified by its endpoint address that also includes the direction flag at bit 7 (MSb). Pipe 0x81 is an IN pipe (transfer direction from device to host) and pipe 0x02 is an OUT pipe (transfer direction from host to device). The application has created three handles for the device by calling `CreateFile()`.

Handle1 is not bound to any pipe, therefore it can be used to perform device-related operations only. It is called a device handle.

Handle2 is bound to the IN pipe 0x81. By using this handle with the Win32 function `ReadFile()` the application can initiate data transfers from endpoint 0x81 to its buffers.

Handle3 is bound to the OUT pipe 0x02. By using Handle3 with the function `WriteFile()` the application can initiate data transfers from its buffers to endpoint 0x02 of the device.

Handle2 and Handle3 are called pipe handles. Note that while Handle1 cannot be used to communicate with a pipe, Handle2 and 3 can execute operations on the pipe and the device.

3.2 Establishing a Connection to the Device

The following code sample demonstrates the steps that are necessary at the USBIO API to establish a handle for a device and a pipe. The code is not complete and no error handling is included.

```
// include the interface header file of USBIO.SYS
#include "usbio_i.h"

// device instance number
#define DEVICE_NUMBER                   0

// some local variables
HANDLE FileHandle;
USBIO_SET_CONFIGURATION SetConfiguration;
USBIO_BIND_PIPE BindPipe;
HDEVINFO DevInfo;
GUID g_UsbioID = USBIO_IID;
SP_DEVICE_INTERFACE_DATA DevData;
SP_INTERFACE_DEVICE_DETAIL_DATA *DevDetail = NULL;
DWORD ReqLen;
DWORD BytesReturned;

// enumerate the devices
// get a handle to the device list
DevInfo = SetupDiGetClassDevs(&g_UsbioID,
                NULL,NULL,DIGCF_DEVICEINTERFACE|DIGCF_PRESENT);
// get the device with index DEVICE_NUMBER
SetupDiEnumDeviceInterfaces(DevInfo, NULL,
                &g_UsbioID, DEVICE_NUMBER, &DevData );
// get length of detailed information
SetupDiGetDeviceInterfaceDetail(DevInfo, &DevData, NULL,
                                        0, &ReqLen, NULL);
// allocate a buffer
DevDetail = (SP_INTERFACE_DEVICE_DETAIL_DATA*) malloc(ReqLen);
// now get the detailed device information
DevDetail->cbSize = sizeof(SP_INTERFACE_DEVICE_DETAIL_DATA);
SetupDiGetDeviceInterfaceDetail(DevInfo, &DevData, DevDetail,
                                        ReqLen, &ReqLen, NULL);
// open the device, use OVERLAPPED flag if necessary
// use DevDetail->DevicePath as device name
FileHandle = CreateFile(
                DevDetail->DevicePath,
                GENERIC_READ|GENERIC_WRITE,
                FILE_SHARE_WRITE|FILE_SHARE_READ,
                NULL,
                OPEN_EXISTING,
                0 /* or FILE_FLAG_OVERLAPPED */,
                NULL);
// setup the data structure for configuration
// use the configuration descriptor with index 0
SetConfiguration.ConfigurationIndex = 0;
// device has 1 interface
SetConfiguration.NbOfInterfaces = 1;
// first interface is 0
SetConfiguration.InterfaceList[0].InterfaceIndex = 0;
// alternate setting for first interface is 0
SetConfiguration.InterfaceList[0].AlternateSettingIndex = 0;
// maximum buffer size for read/write operation is 4069 bytes
SetConfiguration.InterfaceList[0].MaximumTransferSize = 4096;

// configure the device
DeviceIoControl(FileHandle,
```

```
                IOCTL_USBIO_SET_CONFIGURATION,
                &SetConfiguration, sizeof(SetConfiguration),
                NULL,0,
                &BytesReturned,
                NULL
                );

// setup the data structure to bind the file handle
BindPipe.EndpointAddress = 0x81; // the device has an endpoint 0x81
// bind the file handle
DeviceIoControl(FileHandle,
                IOCTL_USBIO_BIND_PIPE,
                &BindPipe, sizeof(BindPipe),
                NULL,0,
                &BytesReturned,
                NULL
                );

// read (or write) data from (to) the device
// use OVERLAPPED structure if necessary
ReadFile(FileHandle, ...);

// close file handle
CloseHandle(FileHandle);
```

Refer to the Win32 API documentation for the syntax and the parameters of the functions `SetupDiXxx()`, `CreateFile()`, `DeviceIoControl()`, `ReadFile()`, `WriteFile()`, `CloseHandle()`. The file handle can be opened with the `FILE_FLAG_OVERLAPPED` flag if asynchronous behavior is required.

More code samples that show the use of the USBIO programming interface are included in the USBIO Development Kit.

3.3 Power Management

Current Windows operating systems support system-level power management. That means that if the computer is idle for a given time, some parts of the computer can go into a sleep mode. A system power change can be initiated by the user or by the operating system itself, on a low battery condition for example. A USB device driver must also support the system power management. Each device which supports power switching must have a device power policy owner, which is responsible for managing the device power states in response to system power state changes. The USBIO driver is the power policy owner of the USB devices that it controls. In addition to the system power changes, the device power policy owner can initiate device power state changes.

Before the system goes into a sleep state, the operating system asks every driver if its device can go into the sleep state. If all active drivers return success the system shuts down. Otherwise, a message box appears on the screen and informs the user that the system is not able to go into sleep mode.

Before the system goes into a sleep state the driver has to save all the information that it needs to reinitialize the device (device context) when the system is resumed. Furthermore, all pending requests have to be completed and further requests have to be queued. In the device power states D1 or D2 (USB Suspend) the device context stored in the USB device will not be lost. Therefore, a device sleep state D1 or D2 is handled transparently by the application. In the state D3 (USB Off) the device context is lost. Because the information stored in the device is known only to the application (e.g. the current volume level of an audio device), the generic USBIO driver cannot

restore the device context in a general way. This must be done by the application. Note that Windows restores the USB configuration of the device (SET_CONFIGURATION request) after the system is resumed.

The behavior with respect to power management can be customized by INF file parameters. For example, if a measurement that takes a long time is to be performed the computer should be prevented from powering down. For a description of the supported INF file parameters, see also chapter 10 (page 303).

All registry entries describing device power states are DWORD parameters where the value 0 corresponds to `DevicePowerD0`, 1 to `DevicePowerD1`, and so on.

The parameter `PowerStateOnOpen` specifies the power state to which the device is set if the first file handle is opened. If the last file handle is closed the USB device is set to the power state specified in the entry `PowerStateOnClose`.

If at least one file handle is open for the device, the key `MinPowerStateUsed` describes the minimal device power state that is required. If this value is set to 0 the computer will never go into a sleep state. If this key is set to 2 the device can go into a suspend state but not into D3 (Off). A power-down request caused by a low battery condition cannot be suppressed by using this parameter.

If no file handle is currently open for the device, the key `MinPowerStateUnused` defines the minimal power state the device can go into. Thus, its meaning is similar to that of the parameter `MinPowerStateUsed`.

If the parameter `AbortPipesOnPowerDown` is set to 1, all pending requests submitted by the application are returned before the device enters a sleeping state. This switch should be set to 1 if the parameter `MinPowerStateUsed` is not D0. The pending I/O requests are returned with the error code `USBIO_ERR_POWER_DOWN`. This signals to the application that the error was caused by a power down event. The application may ignore this error and repeat the request. The re-submitted requests will be queued by the USBIO driver. They will be executed after the device is back in state D0.

3.4 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 way, an application will be notified if a USB device is plugged in or removed.

Please refer to the Microsoft Platform SDK documentation for detailed information on the functions `RegisterDeviceNotification()` and `UnregisterDeviceNotification()`. In addition, the source code of the USBIO demo application USBIOAPP provides an example.

The device notification mechanism is only available if the USBIO device naming scheme is based on Device Interface IDs (GUIDs). See section 3.1.1 (page 32) for details. We strongly recommend the use of this new naming scheme.

4 Driver Customization

4.1 Overview

The USBIO 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 are also using this driver. See also section 4.2 for a discussion of 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 and
- Adaptation of driver behavior for a specific device.

4.2 Reason for Driver Customization

Thesycon's USBIO driver is generic with respect to concrete products. The driver can be used (and shipped together) with many different products from various vendors. All generic drivers 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 customisation is not properly completed, 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 is now loaded.
- 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.
- Uninstall of product A removes .sys files (and possibly other files) which causes product B to become non-functional.
- Uninstall 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 Digital Signature

Windows Vista and later 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 the 64-bit version of Windows Vista or higher. For background 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 certificate that supports the Microsoft Authenticode technology (sometimes called Microsoft Authenticode Digital ID). Such a certificate can be purchased from several certification authorities (CA), including Symantec (formerly Verisign). For details you may refer to
`http://www.symantec.com/verisign/code-signing/microsoft-authenticode`

Alternatively, search the Web for for
"VeriSign Code Signing Certificates for Microsoft Authenticode".

4.3.1 Using the Driver without Code Signing Certificate

If you don’t own a code signing certificate you can still create a customized driver package as described in sections 4.4 and 4.5 below. However, during driver installation on the target system, Windows will show a warning message stating that the vendor of the driver software cannot be verified, see Figure 3. The user needs to confirm this warning in order to continue with driver installation. Windows shows this warning because the .cat files in the driver package are not digitally signed.

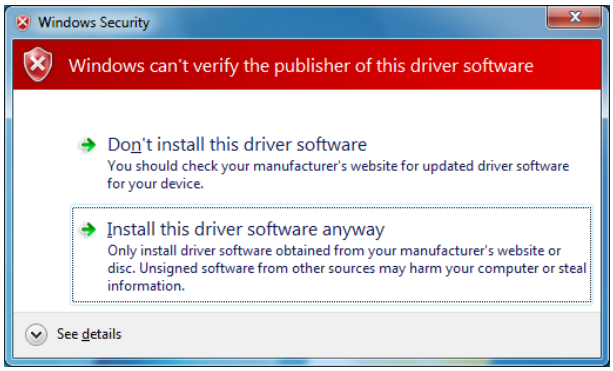

Figure 3: Windows security warning shown if .cat files are not digitally signed

The .sys files included in the driver package are signed with Thesycon’s certificate. If a user inspects the file properties of the .sys file (or driver details in Device Manager) then he/she will find out that the driver is provided by Thesycon.

For instructions on how to proceed without a code signing certificate, refer to specific comments given in the subsequent sections.

Drivers that does not have a digital signature cannot be loaded on the 64-bit variants of Windows 7 and Windows 8. Windows 7 loads the driver if the .sys file has a digital signature while Windows 8 requires a valid digital signature on the .cat file.

On both systems an unsigned driver can be loaded if 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 special 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 method should be used only on development PCs.

4.4 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 the Driver Package Builder.

Before the scripts can be used, the steps described in the following subsections must be executed on the machine on which Driver Package Builder will run.

4.4.1 Install SignTools

The following Microsoft tools are required:

- signtools.exe (part of WDK)
- inf2cat.exe (part of WDK)
- INF checker (part of WDK)

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.

To install the SignTools package, run `SignTools_v1.5.0.exe`.

NOTE: If you don't own a code signing certificate (see 4.3.1 above) then you still need to execute this step. The SignTools package is required to create the .cat files.

4.4.2 Check your Code Signing Certificate

NOTE: If you don't own a code signing certificate (see 4.3.1 above) then you can skip this step.

The certificate to be used for code signing needs to be present in the certificate store of the build machine. For information on how to get the certificate and how to import it into the certificate store, please refer to instructions published by your certificate provider.

Before you continue, you should check if the certificate is present in the Personal folder of the certificate store of the machine in which you want to run Driver Package Builder. To do this, click Start - Run and enter `certmgr.msc`. This launches the Certificates Microsoft Management Console. The Personal folder should contain your certificate and should look similar to figure 4.

To verify that the certificate is valid for code signing, double-click on the certificate to open Details view. The Certificate Information dialog should look as shown in figure 5. For code signing, it is required that the private key that corresponds to the certificate is available. This is indicated

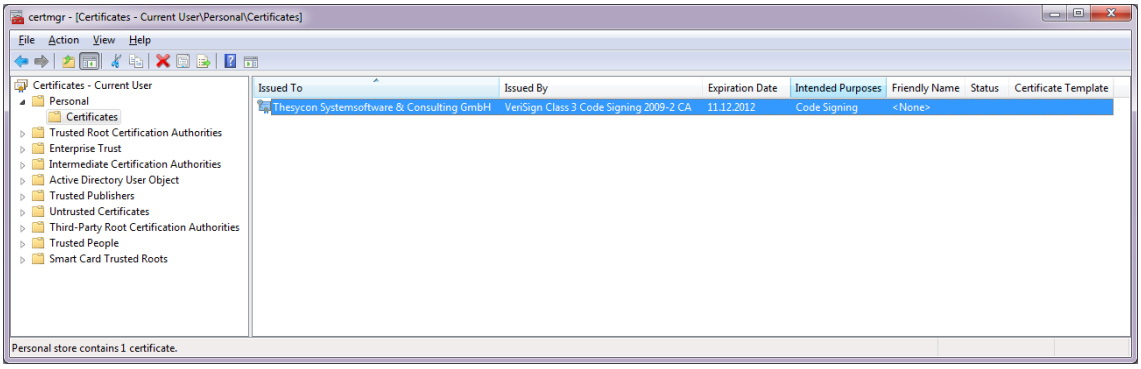

Figure 4: Certificates Microsoft Management Console

by the key icon at the bottom of the General page and the statement "You have a private key that corresponds to this certificate.". If the key icon is missing, only the public key of the certificate is installed and code signing will not be possible.

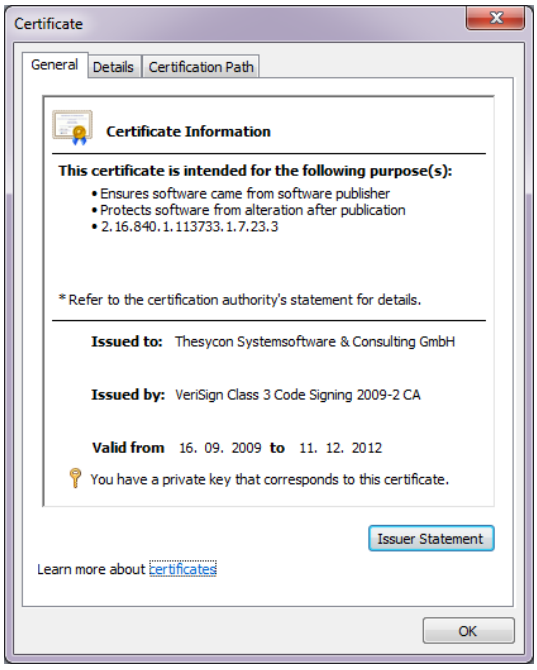

Figure 5: Certificate Information

4.4.3 Configure Certificate Variables

NOTE: If you don't own a code signing certificate (see 4.3.1 above) then you can skip this step. If no certificate is available then `VENDOR_CERTIFICATE` must not be defined which is the default in `set_vendor_certificate.cmd`.

If you own a code signing certificate, edit the `set_vendor_certificate.cmd` script contained in the `signing` subdirectory of the install folder. This script must refer to your certificate in the certificate store described in the previous step. In `set_vendor_certificate.cmd`,

set the variables described below according to your certificate. See also the comments and instructions given within the script itself. Note that alternatively you can add the variables to your build machine's environment.

- VENDOR_CERTIFICATE

  This variable needs to be set to your certificate's name exactly as shown in `certmgr.msc`. Do not surround the string with quotation marks. If your certificate name includes special chars, quote each of them with a preceding escape character (^).

  Example:

  ```
  set VENDOR_CERTIFICATE=MyCompany Ltd.
  ```

- CROSS_CERTIFICATE

  This variable specifies the cross certificate should be used. The cross certificate establishes a trust relationship between Microsoft and your certificate issuer. For latest information on cross certificates, see the article "Cross-Certificates for Kernel Mode Code Signing" in the MSDN library:

  ```
  http://msdn.microsoft.com/
  en-us/library/windows/hardware/gg487315.aspx
  ```

  Alternatively, ask your certificate provider which cross certificate is to be used.

  The cross certificate file must be placed into the `signing` subdirectory. In case of a VeriSign certificate, you should keep the default of `mscv-vsclass3.cer`.

- SIGNTOOL_TIMESTAMP_URL

  This variable specifies a trusted time stamp server to be used when the signature is created. Note that a signature should always be timestamped. In case of VeriSign, you can keep the default shown in `set_vendor_certificate.cmd`. If you are using a different certificate provider, refer to the documentation published by this provider.

4.4.4 Prepare for GUID Generation

Various Globally Unique Identifiers (GUID) are needed for the customization procedure. A GUID can be created easily by using Microsoft's `guidgen.exe` utility. This utility is part of the Microsoft Visual Studio 2005, Visual Studio 2008 and Visual Studio 2012 distribution. Note that it is not included in Visual Studio 2010.

The `guidgen.exe` utility can also be downloaded from this URL:

```
http://www.microsoft.com/
download/en/details.aspx?displaylang=en\&id=17252
```

4.5 Using Driver Package Builder

The Driver Package Builder batch scripts are located in the `DriverPackageBuilder` subdirectory of the driver kit. To create your own customized driver package, please follow the steps described in the following subsections.

NOTE: These steps are mandatory and must be executed to avoid problems in the field.

4.5.1 Customize your Driver Package

The driver package is configured by means of a set of variables defined in `setvars.cmd`. To configure your driver package, edit the file `setvars.cmd`.

The following step-by-step procedure includes the major customizations steps. Make sure that you check the value of each variable that is contained in `setvars.cmd`. For a detailed description of each variable, refer to section 4.6.

1. Set vendor and product strings

   Adapt the following strings to match your company and product:
   ```
   VENDOR_NAME=
   PRODUCT_NAME=
   DRIVER_NAME_BASE=
   ```

2. Generate unique software interface identifiers

   By using the `guidgen.exe` utility (see 4.4.4), generate a fresh GUID **for each** of the following variables:
   ```
   DRIVER_INTERFACE_GUID=
   DEVICE_CLASS_GUID=
   ```

3. Adapt USB VIDs and PIDs

   Each product (device model) is unambiguously identified by its USB vendor ID (VID) and USB product ID (PID). The `setvars.cmd` script allows you to define up to 16 devices which will all be supported by the same driver package. Set one or more of the `INF_VID_PID_xx` variables to match with your device's VID(s) and PID(s). Be careful to get the special format of the `INF_VID_PID_xx` string right. See also the comments and examples within `setvars.cmd`.

   Set `INF_VID_PID_xx_DESCRIPTION` to your product's name. This string will be shown in Device Manager as a product description.

   Clear all unused `INF_VID_PID_xx` variables, e.g.:
   ```
   set INF_VID_PID_05=
   set INF_VID_PID_05_DESCRIPTION=
   ```

4. 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 4.7 for details.

4.5.2 Build your Driver Package

Open a Windows console window (cmd.exe) in the `DriverPackageBuilder` directory. From the command line, run the `create_drvpackages.cmd` script. 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.

Your customized driver package will be created under the `driver_package` subdirectory.

NOTE: If you don't own a code signing certificate then the .sys files will be signed with Thesycon's certificate and the .cat files will not be digitally signed. This leads to a different behavior during driver installation. See 4.3.1 for more details. The unsigned driver package will be created under the `driver_package_unsigned` subdirectory.

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 `DriverPackageBuilder` 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`. Keep in mind that the script `create_driverpackages.cmd` overwrites the modified driver package without warning.

The scripts `create_driverpackages.cmd` and `update_signature.cmd` can be called with the command line parameter `no_sys_signing`. When this parameter is used the .sys files are not digitally signed. This can be helpful when the driver package should be used for the WHQL testing.

4.6 Customization Parameter Reference

4.6.1 VENDOR_NAME

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

4.6.2 PRODUCT_NAME

The name of the product that uses the driver package. It is used to generate the `Disk Name` string in the INF file.

4.6.3 DRIVER_NAME_BASE

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

4.6.4 DRIVER_INTERFACE_GUID

Unique identifier for the driver interface. Create a fresh GUID using `guidgen.exe` (see 4.4.4). This GUID is used to enumerate and open the device. **It is very important to use a fresh GUID here.**

4.6.5 DISABLE_DEFAULT_DRIVER_INTERFACE

Set this to 1 before you release your driver. When this parameter is 1 the USBIOAPP.exe and all other demo applications will not find your devices any more.

4.6.6 DISABLE_CLEANUP_WIZARD

Set this to 1 before you release your driver. When this parameter is 1 the Cleanup Wizard will not find your devices any more.

4.6.7 INF_VID_PID_[01..16]

USB Vendor ID(s) and Product ID(s) of the USB devices (models) that are supported by the driver package.

Format: `VID_xxxx&PID_yyyy` or `VID_xxxx&PID_yyyy&MI_zz`
xxxx specifies the VID in hex format, yyyy specifies the PID in hex format. If the driver is installed on a multi-interface architecture zz identifies the interface number.

4.6.8 INF_VID_PID_[01..16]_DESCRIPTION

Display name of the respective device model. This name is displayed in Windows Device Manager. A display name can be specified for each VID/PID pair.

4.6.9 DEVICE_CLASS

A customer defined string without spaces and special characters used in the registry to store the vendor defined class. If the device is of a standard class this should be the standard class' name. It is recommended to use a vendor defined class for this driver. Do not use the example string as device class name.

4.6.10 DEVICE_CLASS_GUID

A freshly created GUID in registry format like: "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxxxx". If a standard class is used in DEVICE_CLASS, this parameter must be the GUID of the standard class as defined by the WDK.

4.6.11 DEVICE_CLASS_DISPLAY_NAME

The name that is shown in the device manager for the class. If a standard class is used in DEVICE_CLASS, this string must be empty.

4.7 Customization Default Driver Settings

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

```
[_AddReg_HW]

HKR,,PowerStateOnOpen,       %REG_DWORD%, 0
HKR,,PowerStateOnClose,      %REG_DWORD%, 0
HKR,,MinPowerStateUsed,      %REG_DWORD%, 3
HKR,,MinPowerStateUnused,    %REG_DWORD%, 3
HKR,,EnableRemoteWakeup,     %REG_DWORD%, 0
HKR,,AbortPipesOnPowerDown, %REG_DWORD%, 1
HKR,,UnconfigureOnClose,     %REG_DWORD%, 1
HKR,,ResetDeviceOnClose,     %REG_DWORD%, 0
HKR,,MaxIsoPackets,          %REG_DWORD%, 512
HKR,,ShortTransferOk,        %REG_DWORD%, 1
HKR,,RequestTimeout,         %REG_DWORD%, 1000
HKR,,SuppressPnPRemoveDlg,  %REG_DWORD%, 1
HKR,,ConfigDescMinQuerySize,%REG_DWORD%, 0

;HKR,,ConfigIndex,           %REG_DWORD%, 0
;HKR,,Interface,             %REG_DWORD%, 0
;HKR,,AlternateSetting,      %REG_DWORD%, 0

;HKR,,FxFwFile,              %REG_SZ%,  "YourFirmwareFile.ihx"
;HKR,,FxBootloaderCheck,     %REG_DWORD%, 1
;HKR,,FxExtRamBase,          %REG_DWORD%, 0x2000
```

4.7.1 PowerStateOnOpen

This parameter can be in the range 0..3. It contains the power state that is set if the device is opened. The values have the following meaning

- 0 – Power on
- 1 – Suspend
- 2 – Suspend
- 3 – Power off

4.7.2 PowerStateOnClose

This parameter can be in the range 0..3. It contains the power state that is set if the device is closed. See `PowerStateOnOpen` for details.

4.7.3 MinPowerStateUsed

This parameter can be in the range 0..3. See `PowerStateOnOpen` for details. If the value is set to 3 the PC follows the power management setting. If the value is set to 0 the PC does not enter the standby or hibernate mode after a period of inactivity, if at least one handle to the driver is open. However the PC may enter a sleep state by explicit user interaction or on low battery.

4.7.4 MinPowerStateUnused

This parameter has the same meaning as MinPowerStateUsed with the exception that the rules apply if no application has an open handle to the device.

4.7.5 EnableRemoteWakeup

This parameter is a flag with the range 0..1. If it is set and if the device reports a remote wake up capability in the USB descriptors the remote wake up is enabled by the driver. Please note that a lot of additional conditions may prevent a remote wake up. Please check the remote wake up capability with a standard USB mouse if you have doubts that the USB port supports remote wake up correctly. This parameter can be overwritten with the API function **IOCTL_USBIO_SET_DEVICE_PARAMETERS**.

4.7.6 AbortPipesOnPowerDown

This parameter is a flag with the range 0..1. To get a WHQL certification this flag must be set to 1. If this flag is set the driver aborts all pending requests if the device enters a power down state. Please note: The application should register for power messages and stop all data traffic before the device enters a power down state. If the driver aborts the pending requests, a data lost can occur.

4.7.7 UnconfigureOnClose

This parameter is a flag with the range 0..1. If this flag is set the driver sets the device in the unconfigured state if the last handle is closed. This makes sense if the application expects an unconfigured device. This parameter can be overwritten with the API function **IOCTL_USBIO_SET_DEVICE_PARAMETERS**.

4.7.8 ResetDeviceOnClose

This parameter is a flag with the range 0..1. If this flag is set the driver sends a USB reset to the device if the last handle is closed. This parameter can be overwritten with the API function **IOCTL_USBIO_SET_DEVICE_PARAMETERS**.

4.7.9 MaxIsoPackets

The parameter MaxIsoPackets contains the number of isochronous frames that can be submitted with one request. It is possible to send a request with less isochronous frames in one request. The

driver pre-allocates memory to submit the ISO requests to the bus driver. This parameter has an influence to the memory usage of the driver.

4.7.10 ShortTransferOk

This parameter is a flag with the range 0..1. This flag defines the default behavior for IN transactions. If this flag is set the driver accepts data packets smaller than the FIFO size. This parameter can be overwritten with the API function **IOCTL_USBIO_SET_PIPE_PARAMETERS**.

4.7.11 RequestTimeout

This parameter specifies the default timeout interval, in milliseconds, that applies to requests to EP0. A value of zero means an infinite interval (i.e. timeout is disabled).

4.7.12 SuppressPnPRemoveDlg

This parameter is a flag with the range 0..1. If this flag is set there is no entry in the Systray to stop the device. The device can be removed without a warning from the system.

4.7.13 ConfigDescMinQuerySize

When this parameter is 0 the driver first requests the configuration descriptor with a size of 9 bytes. The first part of the configuration descriptor contains the total length information. The driver requests the complete configuration descriptor with this length. Some buggy devices were not able to answer correctly to the partial descriptor request. As a bug fix for such devices this parameter can be used to force the driver to request the configuration descriptor with the correct size on the first attempt. A modification to this parameter is required only if the device shows such a misbehavior and the device firmware cannot be changed any more.

4.7.14 ConfigIndex

This parameter specifies the index of the configuration descriptor that is used to configure the device when the driver starts. If this parameter is uncommented the driver configures the USB device during startup. If the configuration fails, the driver is not loaded. Please note that this parameter specifies the index of the configuration and is different to bConfiguration value in the configuration descriptor. This parameter can be used only if the driver is used with one USB interface.

4.7.15 Interface

This parameter specifies the interface number that should be configured. This value is considered only if the parameter ConfigIndex is set. The default value is 0.

4.7.16 AlternateSetting

This parameter specifies the alternate setting that should be configured. This value is considered only if the parameter ConfigIndex is set. The default value is 0.

4.7.17 Firmware Download Support for Cypress FX ICs

The parameters FxFwFile, FxBootloaderCheck and FxExtRamBase can be used to download a firmware file to a Cypress FX IC. See section 6 on page 57 for more details.

4.8 Customization USBIO COM DLL

If USBIOCOM.DLL is used it must also be customized.
Customization steps are described in USBIO COM Interface Reference Manual [1] chapter 4 (usbio_win_manual_cominterface.pdf).

5 Driver Installation and Uninstallation

This section discusses topics related to the installation and uninstallation of the USBIO device driver.

5.1 Driver Installation for Developers

This section describes some methods by which developers can install and uninstall the USBIO driver on a PC.

The USBIO installation and cleanup wizards are designed for developers. Neither application should be delivered to the final customers. A customer could make a system unusable by installing USBIO to an important system device.

5.1.1 The USBIO Installation Wizard

Using the wizard, the driver installation is managed interactively in a step-by-step procedure. The device for which the USBIO driver should be installed can be selected from a list. It is not necessary to manually edit or copy any files.

The steps required to install the USBIO device driver using the Installation Wizard are described below.

Note: The USBIO Installation Wizard does not work on Windows 8 x64. Please create a customized driver package with valid digital signature and install it manually.

- Make sure that you have administrator privileges to install device drivers on the system.
- Connect your USB device to the system. After connecting the device, Windows XP can launches the New Hardware Wizard. If the New Hardware Wizard is launched, complete it by clicking Next on each page and Finish on the last page.
- Start the USBIO Installation Wizard by selecting the appropriate shortcut from the Start menu. It is also possible to start the wizard directly by executing usbiowiz.exe.
- The first page shows some hints for the installation process. Click the Next button to continue.
- On the next page the wizard shows a list of all the USB devices currently connected to the system. Select the device for which you wish to install the USBIO driver.

  The Hardware ID (if available) and the Compatible ID will be shown for the selected device.

  A Hardware ID is a string that is used internally by the operating system to unambiguously identify the device.

  If your device is not shown in the list make sure it is plugged in properly and you have completed the New Hardware Wizard as described above. You may use the Device Manager to check if the device was enumerated by the system. The device should pass the Command Verifier Test from the USB Implementers Forum before the USBIO is installed. This makes sure that the device answers correctly to all USB standard requests.

  Use the Refresh button to scan for active devices again, and to rebuild the list.

To continue, click the Next button.

- The next page shows detailed information about the selected USB device. If a driver is already installed for the device, information about the driver is also shown. Verify that you have selected the correct device. If not, use the Back button to return to the device list and select another device.

  To install the USBIO driver for the selected device, click the Next button.

  Warning: If you install the USBIO driver for a device that is currently controlled by another device driver, the existing driver will be disabled. This will happen immediately. As a result, neither the operating system nor applications will be able to use the device.

  Warning: It is possible to install the USBIO for a USB keyboard or a USB mouse. Such devices cannot be used as system input devices if the USBIO driver is installed for it. This can make the system unusable.

- On the last page, the Installation Wizard shows the completion status of driver installation. If the installation was successful, the USBIO driver has been dynamically loaded by the operating system, and is now running.

  The USBIO Installation Wizard allows you to show the specific driver installation files (INF) generated for the device. In the opened folder you find the created INF files and SYS files for Windows 32 bit versions or for the x64 Edition. You can use these files later to install the USBIO driver manually.

  You can use the button labeled "Run Application" to start the demo application included in the USBIO package. Please refer to chapter 9 on page 299 for further information.

  To quit the USBIO Installation Wizard, click Finish.

5.1.2 The USBIO Cleanup Wizard

The USBIO Cleanup Wizard is designed for developers. This application must not be delivered to the final customers.

To start the USBIO Cleanup Wizard use the Start Menu or run the application USBIOcw.exe from the installation folder. The wizard can clean up the registry and it can remove the INF files that are copied by the system during the installation. The cleanup of the registry is equivalent to the manual de-installation with the device manager.

On the first page of the USBIO Cleanup Wizard the required tasks can be selected. If the INF files and the registry entries are removed the system will behave as if the driver was never installed when clean-up has completed.

On the second page, the wizard shows a list of installed devices. By default, all instances of the USBIO driver are selected. The selection can be changed to uninstall only some instances. By highlighting a device some detailed information is displayed in the lower box.

On the next page the INF files that were used to install the USBIO driver are listed. By default all INF files are selected. The selection can be changed. By highlighting a INF file some detailed information is displayed in the lower box.

On the last page the wizard shows a summary of all performed steps.

To leave the USBIO Cleanup Wizard press the Finish button.

Note: The Cleanup Wizard uses the .inf file parameter `CleanupWizard_DeviceIdentString`. When this parameter is removed from the .inf file the cleanup wizard will not remove the driver.

5.1.3 Installing USBIO Manually

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

In order to install the USBIO driver manually you have to create a customized driver package. Refer to section 4 on page 39 for more information.

The steps required to install the driver are described below.

- Copy the .inf, .sys and .cat files that are required to the hard disk. Use a folder like `c:\program files\<Vendor>\<Product>\<Version>\drivers`.
  Do not copy the files in system folders.

  If you install the driver files directly from a removable medium, Windows XP asks you on the next installation step to insert the medium again. Newer systems make a copy in the driver store.

- Connect your USB device to the system. Windows XP launches the New Hardware Wizard and prompts you for a device driver. On newer systems the device manager must be used to start a driver update. Provide the New Hardware Wizard the location of your installation files.

  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 will not launch the New Hardware Wizard after the device is plugged in. Instead, a system-provided device driver will be installed silently. Use the device manager to update the driver. Select "Let me pick from a list..." and "Have disk" to select your driver package.

- After the driver installation has been successfully completed your device should be shown in the Device Manager in the section corresponding to the device class you specified in the .inf file. You may use the Properties dialog box of that entry to verify that the USBIO driver is installed and running.

- To verify that the USBIO driver is working properly with your device, you should use the USBIO Demo Application USBIOAPP.EXE. Please refer to chapter 9 on page 299 for detailed information on the Demo Application.

5.1.4 Uninstalling USBIO manually

The manual deinstallation of driver should only be used by developers. If an important system driver is removed the system can become unusable.

To uninstall the USBIO device driver for a given device, use the Device Manager. There are two options to uninstall the USBIO 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.
- On Windows Vista and later, the system supports a checkbox to delete a driver from the system. When this box is selected the system performs a "roll back". This means the current driver is removed and a previous version is installed. If no previous version is available no driver is loaded when the device is reconnected.
- 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.

5.2 Installing USBIO for End Users

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

5.2.1 Installing USBIO 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. A Demo Version can be downloaded separately under the following link: **http://www.thesycon.de/pnpinstaller**.

The installation program can be run in an interactive mode with a 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 uninstallation 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 uncertified software. When the device is connected to the PC the correct driver software is installed immediately during installation. 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 of whether the devices are connected or not. The exact driver version from the installer package is installed. This way of updating drivers 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 matching 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**.

6 Cypress FX Support

The USBIO driver package supports the firmware download for Cypress FX series ICs. The driver can download the firmware from an Intel Hexadecimal file in ASCII format. The file must contain only "Data Records" and one "End of File Record". Other records are not supported. This is the usual format for 8 bit controllers.

6.1 Configuration

The firmware download feature can be enabled and configured by parameters in the INF file. The following parameters can be used.

6.1.1 FxFwFile

This parameter is a REG_SZ string with the file name of the firmware file. If this parameter is not set, the driver does not make a firmware download. If this parameter is set it should contain the complete file name of a Intel Hex firmware file without a path. The firmware file must be placed in the folder `%SystemRoot%\system32\drivers`.

If the firmware file cannot be opened or has invalid content the USBIO creates a log entry in the system log and does not start.

6.1.2 FxBootLoaderCheck

This parameter is a REG_DWORD and can have the values 0 or 1. If the parameter is set to 1, the driver reads 16 bytes from the internal memory. If the request is completed successfully it assumes the device is in boot loader mode and starts the firmware download.

The new LP series of FX2 ICs also supports this function if the vendor firmware is running. Additionally the driver asks for the string descriptor at index 0. If the string descriptor is 2 bytes long it assumes the boot loader is running. The vendor specific firmware should implement at least one language ID.

If this check determines that the vendor firmware is already running, the driver starts without a firmware download.

If the value is set to 0 the driver performs always a firmware download without any check.

6.1.3 FxExtRamBase

This parameter is a REG_DWORD and describes the first address of the external RAM. The download protocol has different requests to write data to the internal and to the external RAM. All records in the firmware file with a start address smaller than this parameter are written to the internal RAM. All other records are written to the external RAM.

We have seen that on some integrated circuits the write requests to external RAM fail and the IC stops responding. Please check whether the used IC support the firmware download to external RAM.

The default value is 0x4000.

6.2 Copy the Firmware File

If the firmware download feature of the driver is used, the firmware file becomes part of the driver package. The firmware file must be copied together with the driver. The name of the firmware file must be entered in the section `[_CopyFiles_sys]` and `[SourceDisksFiles]`.

```
[_CopyFiles_sys]
;YourFirmwareFile.ihx

[SourceDisksFiles]
;YourFirmwareFile.ihx=1
```

Replace the name `YourFirmwareFile.ihx` with your firmware file and uncomment the lines. It is not possible to include a path in the firmware name.

6.3 Operation of Firmware Download

The firmware download is performed if the parameter FxFwFile is set. If any error occurs during the firmware download the driver does not start and is marked in the device manager as non operable. Possible errors are:

- File not found
- Invalid file content, including invalid records
- Data transfer error to the IC

Before the download is started, the driver sets the CPU to a reset state. The switch to the reset state is performed by writing 0x01 to address 0xE600 for the FX2 and by writing 0x01 to the address 0x7F92 for the FX1. Nether write instruction has an influence if they are sent to the invalid device type. If the firmware download is completed, the CPU reset is cleared. The downloaded code is started from address 0. The downloaded firmware is responsible for performing a re-enumeration (disconnect and re-connect). During this step the driver is reloaded, it detects that the functional software is loaded and it exposes the user mode interface.

Note: The driver does not force a re-enumeration of the device after the download has completed.

During a firmware download the interface of the driver is not enabled. This means that an application does not get a PnP event if USBIO works as a download driver and the driver cannot be opened by the application. The driver expects that the device disconnects itself after the firmware download and emulates a PnP cycle.

The firmware download can be set up in two ways:

6.3.1 With two Product ID's

In this case the boot loader and the firmware report different product ID's in the device descriptor. One configuration in the INF file sets the parameter FxFwFile. A drawback of this method is the installation. It must be performed for the boot loader and the real device driver.

6.3.2 With one Product ID

In this case the boot loader and the firmware report the same vendor and product ID. The driver configuration sets the parameter FxBootLoaderCheck to 1. The firmware responds with a stall to the request "Read internal RAM" Setup: 0xC0 0xA0 .... and reports a language ID to the string descriptor request at index 0. This has the advantage that only one installation is required.

7 Programming Interface

This section describes the programming interface of the USBIO device driver in detail. The programming interface is based on Win32 functions.

Note that there is a high-level programming interface available which is based on Microsoft's COM technology. The USBIO COM Interface is included in the USBIO software package. For more information refer to the USBIO COM Interface Reference Manual.

7.1 Programming Interface Overview

This section lists all operations supported by the USBIO programming interface sorted by category.

7.1.1 Query Information Requests

Operation	Issued On	Bus Action
IOCTL_USBIO_GET_DRIVER_INFO	device	none
IOCTL_USBIO_GET_DEVICE_INFO	device	none
IOCTL_USBIO_GET_BANDWIDTH_INFO	device	none
IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER	device	none

7.1.2 Device-related Requests

Operation	Issued On	Bus Action
IOCTL_USBIO_GET_DESCRIPTOR	device	SETUP request on EP0
IOCTL_USBIO_SET_DESCRIPTOR	device	SETUP request on EP0
IOCTL_USBIO_SET_FEATURE	device	SETUP request on EP0
IOCTL_USBIO_CLEAR_FEATURE	device	SETUP request on EP0
IOCTL_USBIO_GET_STATUS	device	SETUP request on EP0
IOCTL_USBIO_GET_CONFIGURATION	device	SETUP request on EP0
IOCTL_USBIO_SET_CONFIGURATION	device	SETUP request on EP0
IOCTL_USBIO_UNCONFIGURE_DEVICE	device	SETUP request on EP0
IOCTL_USBIO_GET_INTERFACE	device	SETUP request on EP0
IOCTL_USBIO_SET_INTERFACE	device	SETUP request on EP0
IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST	device	SETUP request on EP0
IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST	device	SETUP request on EP0
IOCTL_USBIO_GET_DEVICE_PARAMETERS	device	none
IOCTL_USBIO_SET_DEVICE_PARAMETERS	device	none
IOCTL_USBIO_GET_CONFIGURATION_INFO	device	none
IOCTL_USBIO_ACQUIRE_DEVICE	device	none
IOCTL_USBIO_RELEASE_DEVICE	device	none

IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR	device	none
IOCTL_USBIO_RESET_DEVICE	device	reset on hub port, SET_ADDRESS request
IOCTL_USBIO_CYCLE_PORT	device	reset on hub port, SET_ADDRESS request
IOCTL_USBIO_SET_DEVICE_POWER_STATE	device	set properties on hub port
IOCTL_USBIO_GET_DEVICE_POWER_STATE	device	none

7.1.3 Pipe-related Requests

Operation	Issued On	Bus Action
IOCTL_USBIO_BIND_PIPE	device	none
IOCTL_USBIO_UNBIND_PIPE	pipe	none
IOCTL_USBIO_RESET_PIPE	pipe	none
IOCTL_USBIO_ABORT_PIPE	pipe	none
IOCTL_USBIO_GET_PIPE_PARAMETERS	pipe	none
IOCTL_USBIO_SET_PIPE_PARAMETERS	pipe	none
IOCTL_USBIO_SETUP_PIPE_STATISTICS	pipe	none
IOCTL_USBIO_QUERY_PIPE_STATISTICS	pipe	none
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN	pipe	SETUP request on endpoint
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT	pipe	SETUP request on endpoint

7.1.4 Data Transfer Requests

Operation	Issued On	Bus Action
`ReadFile` function	pipe	data transfer from IN endpoint to host
`WriteFile` function	pipe	data transfer from host to OUT endpoint

7.2 Control Requests

This section provides a detailed description of the I/O Control operations the USBIO driver supports through its programming interface. The I/O Control requests are submitted to the driver using the Win32 function `DeviceIoControl()` (see also chapter 3 on page 31). This function is defined as follows:

```
BOOL DeviceIoControl(
        HANDLE hDevice,           // handle to device of interest
        DWORD dwIoControlCode,    // control code of operation to perform
        LPVOID lpInBuffer,        // pointer to buffer to supply input data
        DWORD nInBufferSize,      // size of input buffer
        LPVOID lpOutBuffer,       // pointer to buffer to receive output data
        DWORD nOutBufferSize,     // size of output buffer
        LPDWORD lpBytesReturned,  // pointer to variable to receive
                                  // output byte count
        LPOVERLAPPED lpOverlapped // pointer to overlapped structure
                                  // for asynchronous operation
        );
```

Refer to the Microsoft Platform SDK documentation for more information.

The following sections describe the I/O Control codes that may be passed to the `DeviceIoControl()` function as `dwIoControlCode` and the parameters required for `lpInBuffer`, `nInBufferSize`, `lpOutBuffer`, `nOutBufferSize`.

IOCTL_USBIO_GET_DESCRIPTOR

The IOCTL_USBIO_GET_DESCRIPTOR operation requests a specific descriptor from the device.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_DESCRIPTOR for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_DESCRIPTOR_REQUEST** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_DESCRIPTOR_REQUEST) for this operation.

`lpOutBuffer`
Points to a caller-provided buffer that will receive the descriptor data.

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

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds.

Comments

The buffer that is passed to this function by means of `lpOutBuffer` should be large enough to hold the requested descriptor. Otherwise, only `nOutBufferSize` bytes from the beginning of the descriptor will be returned.

The size of the output buffer provided at `lpOutBuffer` should be a multiple of the FIFO size (maximum packet size) of endpoint zero.

If the request is completed successfully then the variable pointed to by `lpBytesReturned` is set to the number of descriptor data bytes returned in the output buffer.

See Also

USBIO_DESCRIPTOR_REQUEST (page 120)
IOCTL_USBIO_SET_DESCRIPTOR (page 67)

IOCTL_USBIO_SET_DESCRIPTOR

The IOCTL_USBIO_SET_DESCRIPTOR operation sets a specific descriptor of the device.

`dwIoControlCode`
Set to IOCTL_USBIO_SET_DESCRIPTOR for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_DESCRIPTOR_REQUEST** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_DESCRIPTOR_REQUEST) for this operation.

`lpOutBuffer`
Points to a caller-provided buffer that contains the descriptor data to be set.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This is equal to the number of descriptor data bytes to be transferred to the device.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes transferred if the request succeeds.

Comments

USB devices do not have to support a SET_DESCRIPTOR request. Consequently, most USB devices do not support the IOCTL_USBIO_SET_DESCRIPTOR operation.

Although the data buffer is described by the parameters `lpOutBuffer` and `nOutBufferSize` it provides input data for the request. Some confusion is caused by the naming scheme of the Windows API.

See Also

USBIO_DESCRIPTOR_REQUEST (page 120)
IOCTL_USBIO_GET_DESCRIPTOR (page 66)

IOCTL_USBIO_SET_FEATURE

The IOCTL_USBIO_SET_FEATURE operation is used to set or enable a specific feature.

`dwIoControlCode`
Set to IOCTL_USBIO_SET_FEATURE for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_FEATURE_REQUEST** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_FEATURE_REQUEST) for this operation.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

The SET_FEATURE request appears on the bus with the parameters specified in the **USBIO_FEATURE_REQUEST** data structure pointed to by `lpInBuffer`.

See Also

USBIO_FEATURE_REQUEST (page 122)
IOCTL_USBIO_CLEAR_FEATURE (page 69)

IOCTL_USBIO_CLEAR_FEATURE

The IOCTL_USBIO_CLEAR_FEATURE operation is used to clear or disable a specific feature.

`dwIoControlCode`
Set to IOCTL_USBIO_CLEAR_FEATURE for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_FEATURE_REQUEST** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_FEATURE_REQUEST) for this operation.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

The CLEAR_FEATURE request appears on the bus with the parameters specified in the **USBIO_FEATURE_REQUEST** data structure pointed to by `lpInBuffer`.

See Also

USBIO_FEATURE_REQUEST (page 122)
IOCTL_USBIO_SET_FEATURE (page 68)

IOCTL_USBIO_GET_STATUS

The IOCTL_USBIO_GET_STATUS operation requests status information for a specific recipient.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_STATUS for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_STATUS_REQUEST** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_STATUS_REQUEST) for this operation.

`lpOutBuffer`
Points to a caller-provided **USBIO_STATUS_REQUEST_DATA** data structure. This data structure will receive the results of the IOCTL operation.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This value has to be set to sizeof(USBIO_STATUS_REQUEST_DATA) for this operation.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds. The returned value will be equal to sizeof(USBIO_STATUS_REQUEST_DATA).

Comments

The GET_STATUS request appears on the bus with the parameters specified in the **USBIO_STATUS_REQUEST** data structure. On successful completion the IOCTL operation returns the data structure **USBIO_STATUS_REQUEST_DATA** in the buffer pointed to by `lpOutBuffer`.

See Also

USBIO_STATUS_REQUEST (page 123)
USBIO_STATUS_REQUEST_DATA (page 124)

IOCTL_USBIO_GET_CONFIGURATION

The IOCTL_USBIO_GET_CONFIGURATION operation retrieves the current configuration of the device.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_CONFIGURATION for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Points to a caller-provided **USBIO_GET_CONFIGURATION_DATA** data structure. This data structure will receive the results of the IOCTL operation.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This value has to be set to sizeof(USBIO_GET_CONFIGURATION_DATA) for this operation.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds. The returned value will be equal to sizeof(USBIO_GET_CONFIGURATION_DATA).

Comments

A GET_CONFIGURATION request appears on the bus. The data structure **USBIO_GET_CONFIGURATION_DATA** pointed to by `lpOutBuffer` returns the configuration value. A value of zero means "not configured".

See Also

USBIO_GET_CONFIGURATION_DATA (page 125)
IOCTL_USBIO_GET_INTERFACE (page 72)
IOCTL_USBIO_SET_CONFIGURATION (page 74)

IOCTL_USBIO_GET_INTERFACE

The IOCTL_USBIO_GET_INTERFACE operation retrieves the current alternate setting of a specific interface.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_INTERFACE for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_GET_INTERFACE** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_GET_INTERFACE) for this operation.

`lpOutBuffer`
Points to a caller-provided **USBIO_GET_INTERFACE_DATA** data structure. This data structure will receive the results of the IOCTL operation.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This value has to be set to sizeof(USBIO_GET_INTERFACE_DATA) for this operation.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds. The returned value will be equal to sizeof(USBIO_GET_INTERFACE_DATA).

Comments

A GET_INTERFACE request appears on the bus. The data structure **USBIO_GET_INTERFACE_DATA** pointed to by `lpOutBuffer` returns the current alternate setting of the interface specified in the **USBIO_GET_INTERFACE** structure.

Note: This request is not supported by the USB driver stack on Windows XP. Consequently, on Windows XP this IOCTL operation will be completed with an error code of USBIO_ERR_NOT_SUPPORTED (0xE0000E00).

See Also

USBIO_GET_INTERFACE (page 126)
USBIO_GET_INTERFACE_DATA (page 127)
IOCTL_USBIO_SET_CONFIGURATION (page 74)
IOCTL_USBIO_SET_INTERFACE (page 77)

IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR

The IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR operation stores a configuration descriptor to be used for subsequent set configuration requests within the USBIO device driver.

`dwIoControlCode`
Set to IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR for this operation.

`lpInBuffer`
Points to a caller-provided buffer that contains the descriptor data to be set.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This is equal to the number of descriptor data bytes to be stored.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

This IOCTL request may be used to store a user-defined configuration descriptor within the USBIO driver. The stored descriptor is used by the USBIO driver in subsequent **IOCTL_USBIO_SET_CONFIGURATION** operations. The usage of IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR is optional. If no user-defined configuration descriptor is stored USBIO will use the configuration descriptor provided by the device.

Note: This IOCTL operation is obsolete and should not be used. It was introduced in earlier versions of USBIO to work around problems caused by the Windows USB driver stack. The stack was not able to handle some types of isochronous endpoint descriptors correctly. In the meantime these problems have been fixed.

See Also

IOCTL_USBIO_SET_CONFIGURATION (page 74)

IOCTL_USBIO_SET_CONFIGURATION

The IOCTL_USBIO_SET_CONFIGURATION operation is used to set the device configuration.

`dwIoControlCode`
Set to IOCTL_USBIO_SET_CONFIGURATION for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_SET_CONFIGURATION** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_SET_CONFIGURATION) for this operation.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

A SET_CONFIGURATION request appears on the bus. The USB bus driver USBD generates additional SET_INTERFACE requests on the bus if necessary. The parameters used for the SET_CONFIGURATION and SET_INTERFACE requests are taken from the configuration descriptor that is reported by the device.

One or more interfaces can be configured with one call. The number of interfaces and the alternate setting for each interface have to be specified in the **USBIO_SET_CONFIGURATION** structure pointed to by `lpInBuffer`.

All pipe handles associated with the device will be unbound and all pending requests will be canceled. If this request returns successfully, new pipe objects are available. The IOCTL operation **IOCTL_USBIO_GET_CONFIGURATION_INFO** may be used to query all available pipes and interfaces.

The parameter `MaximumTransferSize` has a size limitation on various operating systems and host controllers. See Microsofts Knowledge Base Article t’t’Maximum size of USB transfers on various operating systemst’t’ KB 832430 for details.

See Also

USBIO_SET_CONFIGURATION (page 129)
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 83)
IOCTL_USBIO_UNCONFIGURE_DEVICE (page 76)
IOCTL_USBIO_GET_CONFIGURATION (page 71)

IOCTL_USBIO_GET_INTERFACE (page 72)
IOCTL_USBIO_SET_INTERFACE (page 77)

IOCTL_USBIO_UNCONFIGURE_DEVICE

The IOCTL_USBIO_UNCONFIGURE_DEVICE operation is used to set the device to its unconfigured state.

`dwIoControlCode`
Set to IOCTL_USBIO_UNCONFIGURE_DEVICE for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

A SET_CONFIGURATION request with the configuration value 0 appears on the bus. All pipe handles associated with the device will be unbound and all pending requests will be cancelled.

See Also

IOCTL_USBIO_SET_CONFIGURATION (page 74)
IOCTL_USBIO_GET_CONFIGURATION (page 71)
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 83)
IOCTL_USBIO_GET_INTERFACE (page 72)
IOCTL_USBIO_SET_INTERFACE (page 77)

IOCTL_USBIO_SET_INTERFACE

The IOCTL_USBIO_SET_INTERFACE operation sets the alternate setting of a specific interface.

`dwIoControlCode`
Set to IOCTL_USBIO_SET_INTERFACE for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_INTERFACE_SETTING** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_INTERFACE_SETTING) for this operation.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

A SET_INTERFACE request appears on the bus.

All pipe handles associated with the interface will be unbound and all pending requests will be cancelled. If this request returns with success, new pipe objects are available. The operation **IOCTL_USBIO_GET_CONFIGURATION_INFO** may be used to query all available pipes and interfaces.

If invalid parameters (e.g. non-existing Alternate Setting) are specified in the **USBIO_INTERFACE_SETTING** data structure an error status of USBIO_ERR_INVALID_PARAM will be returned. The previous configuration is lost in this case. The device has to be re-configured by using the IOCTL operation **IOCTL_USBIO_SET_CONFIGURATION**.

See Also

USBIO_INTERFACE_SETTING (page 128)
IOCTL_USBIO_GET_INTERFACE (page 72)
IOCTL_USBIO_SET_CONFIGURATION (page 74)
IOCTL_USBIO_GET_CONFIGURATION (page 71)
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 83)

IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST

The IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST operation is used to generate a class or vendor specific device request with a data transfer direction from device to host.

`dwIoControlCode`
Set to IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_CLASS_OR_VENDOR_REQUEST** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_CLASS_OR_VENDOR_REQUEST) for this operation.

`lpOutBuffer`
Points to a caller-provided buffer that will receive the data bytes transferred from the device during the data phase of the control transfer. If the class or vendor specific device request does not return any data this value can be set to NULL. `nOutBufferSize` has to be set to zero in this case.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This is equal to the length, in bytes, of the data transfer phase of the class or vendor specific device request. If this value is set to zero then there is no data transfer phase. `lpOutBuffer` should be set to NULL in this case.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds.

Comments

A SETUP request appears on the default pipe (endpoint zero) of the USB device with the parameters defined by means of the **USBIO_CLASS_OR_VENDOR_REQUEST** structure pointed to by `lpInBuffer`. If a data transfer phase is required an IN token appears on the bus and the successful transfer is acknowledged by an OUT token with a zero length data packet. If no data phase is required an IN token appears on the bus with a zero length data packet from the USB device for acknowledge.

If the request is completed successfully then the variable pointed to by `lpBytesReturned` is set to the number of data bytes successfully transferred during the data transfer phase of the control transfer.

See Also

USBIO_CLASS_OR_VENDOR_REQUEST (page 130)
IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST (page 79)

IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST

The IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST operation is used to generate a class or vendor specific device request with a data transfer direction from host to device.

`dwIoControlCode`
Set to IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_CLASS_OR_VENDOR_REQUEST** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_CLASS_OR_VENDOR_REQUEST) for this operation.

`lpOutBuffer`
Points to a caller-provided buffer that contains the data bytes to be transferred to the device during the data phase of the control transfer. If the class or vendor specific device request does not have a data phase this value can be set to NULL. `nOutBufferSize` has to be set to zero in this case.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This is equal to the length, in bytes, of the data transfer phase of the class or vendor specific device request. If this value is set to zero then there is no data transfer phase. `lpOutBuffer` should be set to NULL in this case.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes transferred from the buffer pointed to by `lpOutBuffer` if the request succeeds.

Comments

A SETUP request appears on the default pipe (endpoint zero) of the USB device with the parameters defined by means of the **USBIO_CLASS_OR_VENDOR_REQUEST** structure pointed to by `lpInBuffer`. If a data transfer phase is required an OUT token appears on the bus and the successful transfer is acknowledged by an IN token with a zero length data packet from the device. If no data phase is required an IN token appears on the bus and the device acknowledges with a zero length data packet.

If the request is completed successfully then the variable pointed to by `lpBytesReturned` is set to the number of data bytes successfully transferred during the data transfer phase of the control transfer.

Although the data buffer is described by the parameters `lpOutBuffer` and `nOutBufferSize`, it provides input data for the request. Some confusion is caused by the naming scheme of the Windows API.

See Also

USBIO_CLASS_OR_VENDOR_REQUEST (page 130)
IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST (page 78)

IOCTL_USBIO_GET_DEVICE_PARAMETERS

The IOCTL_USBIO_GET_DEVICE_PARAMETERS operation returns USBIO driver settings related to a device.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_DEVICE_PARAMETERS for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Points to a caller-provided **USBIO_DEVICE_PARAMETERS** data structure. This data structure will receive the results of the IOCTL operation.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This value has to be set to sizeof(USBIO_DEVICE_PARAMETERS) for this operation.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds. The returned value will be equal to sizeof(USBIO_DEVICE_PARAMETERS).

Comments

The default state of device-related settings is defined by a set of registry parameters that are read by the USBIO driver at startup. The current state can be retrieved by means of this request.

This IOCTL operation retrieves internal driver settings. It does not cause any action on the USB.

See Also

USBIO_DEVICE_PARAMETERS (page 132)
IOCTL_USBIO_SET_DEVICE_PARAMETERS (page 82)
IOCTL_USBIO_GET_PIPE_PARAMETERS (page 100)
IOCTL_USBIO_SET_PIPE_PARAMETERS (page 101)

IOCTL_USBIO_SET_DEVICE_PARAMETERS

The IOCTL_USBIO_SET_DEVICE_PARAMETERS operation is used to set USBIO driver settings related to a device.

`dwIoControlCode`
Set to IOCTL_USBIO_SET_DEVICE_PARAMETERS for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_DEVICE_PARAMETERS** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_DEVICE_PARAMETERS) for this operation.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

The default state of device-related settings is defined by a set of registry parameters that are read by the USBIO driver at startup. The current state can be modified by means of this request.

This IOCTL operation modifies internal driver settings. It does not cause any action on the USB.

See Also

USBIO_DEVICE_PARAMETERS (page 132)
IOCTL_USBIO_GET_DEVICE_PARAMETERS (page 81)
IOCTL_USBIO_GET_PIPE_PARAMETERS (page 100)
IOCTL_USBIO_SET_PIPE_PARAMETERS (page 101)

IOCTL_USBIO_GET_CONFIGURATION_INFO

The IOCTL_USBIO_GET_CONFIGURATION_INFO operation returns information about the pipes and interfaces that are available after the device has been configured.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_CONFIGURATION_INFO for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Points to a caller-provided **USBIO_CONFIGURATION_INFO** data structure. This data structure will receive the results of the IOCTL operation.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This value has to be set to sizeof(USBIO_CONFIGURATION_INFO) for this operation.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds. The returned value will be equal to sizeof(USBIO_CONFIGURATION_INFO).

Comments

This operation returns information about all active pipes and interfaces that are available in the current configuration.

See Also

USBIO_CONFIGURATION_INFO (page 138)
IOCTL_USBIO_SET_CONFIGURATION (page 74)
IOCTL_USBIO_GET_CONFIGURATION (page 71)
IOCTL_USBIO_SET_INTERFACE (page 77)
IOCTL_USBIO_GET_INTERFACE (page 72)

IOCTL_USBIO_RESET_DEVICE

The IOCTL_USBIO_RESET_DEVICE operation causes a reset at the hub port to which the device is connected.

`dwIoControlCode`
Set to IOCTL_USBIO_RESET_DEVICE for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

The following events occur on the bus if this IOCTL request is issued:

```
USB Reset
GET_DEVICE_DESCRIPTOR
USB Reset
SET_ADDRESS
GET_DEVICE_DESCRIPTOR
GET_CONFIGURATION_DESCRIPTOR
```

Note that the device receives two USB Resets and a new USB address will be assigned by the USB bus driver USBD.

After the IOCTL_USBIO_RESET_DEVICE operation is completed the device is in the unconfigured state. Furthermore, all pipes associated with the device will be unbound and all pending read and write requests will be cancelled.

The USBIO driver allows a USB reset request only if the device is configured. That means **IOCTL_USBIO_SET_CONFIGURATION** has been successfully executed. If the device is in the unconfigured state, this request returns with an error status. This limitation is caused by the behavior of Windows 2000. A system crash will occur on Windows 2000 if a USB Reset is issued for an unconfigured device. Therefore, USBIO does not allow to issue a USB Reset while the device is unconfigured.

If the device changes its USB descriptor set during a USB Reset the **IOCTL_USBIO_CYCLE_PORT** request should be used instead of IOCTL_USBIO_RESET_DEVICE.

This request does not work if the system-provided multi-interface driver is used.

See Also

IOCTL_USBIO_SET_CONFIGURATION (page 74)
IOCTL_USBIO_CYCLE_PORT (page 92)

IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER

The IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER operation returns the current value of the frame number counter that is maintained by the USB bus driver USBD.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Points to a caller-provided **USBIO_FRAME_NUMBER** data structure. This data structure will receive the results of the IOCTL operation.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This value has to be set to sizeof(USBIO_FRAME_NUMBER) for this operation.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds. The returned value will be equal to sizeof(USBIO_FRAME_NUMBER).

Comments

The frame number returned by this IOCTL operation is an unsigned 32 bit value. The lower 11 bits of this value correspond to the frame number value in the Start Of Frame (SOF) token on the USB.

See Also

USBIO_FRAME_NUMBER (page 139)

IOCTL_USBIO_SET_DEVICE_POWER_STATE

The IOCTL_USBIO_SET_DEVICE_POWER_STATE operation sets the power state of the USB device.

`dwIoControlCode`
Set to IOCTL_USBIO_SET_DEVICE_POWER_STATE for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_DEVICE_POWER** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_DEVICE_POWER) for this operation.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

The device power state is maintained internally by the USBIO driver. This request allows the application to change the current device power state.

If the device is set to a suspend state (any power state different from D0) then all pending requests should be cancelled before a new device power state is set by means of this IOCTL operation.

See also the sections 3.3 (page 36) and the description of the data structure **USBIO_DEVICE_POWER** for more information on power management.

See Also

USBIO_DEVICE_POWER (page 140)
IOCTL_USBIO_GET_DEVICE_POWER_STATE (page 88)

IOCTL_USBIO_GET_DEVICE_POWER_STATE

The IOCTL_USBIO_GET_DEVICE_POWER_STATE operation retrieves the current power state of the device.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_DEVICE_POWER_STATE for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Points to a caller-provided **USBIO_DEVICE_POWER** data structure. This data structure will receive the results of the IOCTL operation.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This value has to be set to sizeof(USBIO_DEVICE_POWER) for this operation.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds. The returned value will be equal to sizeof(USBIO_DEVICE_POWER).

Comments

The device power state is maintained internally by the USBIO driver. This request allows to query the current device power state.

See also the sections 3.3 (page 36) and the description of the data structure **USBIO_DEVICE_POWER** for more information on power management.

See Also

USBIO_DEVICE_POWER (page 140)
IOCTL_USBIO_SET_DEVICE_POWER_STATE (page 87)

IOCTL_USBIO_GET_BANDWIDTH_INFO

The IOCTL_USBIO_GET_BANDWIDTH_INFO request returns information on the current USB bandwidth consumption.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_BANDWIDTH_INFO for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Points to a caller-provided **USBIO_BANDWIDTH_INFO** data structure. This data structure will receive the results of the IOCTL operation.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This value has to be set to sizeof(USBIO_BANDWIDTH_INFO) for this operation.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds. The returned value will be equal to sizeof(USBIO_BANDWIDTH_INFO).

Comments

This IOCTL operation allows an application to check the bandwidth that is available on the USB. Depending on this information an application can select an appropriate device configuration, if desired.

See Also

USBIO_BANDWIDTH_INFO (page 116)
IOCTL_USBIO_GET_DEVICE_INFO (page 90)

IOCTL_USBIO_GET_DEVICE_INFO

The IOCTL_USBIO_GET_DEVICE_INFO request returns information on the USB device.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_DEVICE_INFO for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Points to a caller-provided **USBIO_DEVICE_INFO** data structure. This data structure will receive the results of the IOCTL operation.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This value has to be set to sizeof(USBIO_DEVICE_INFO) for this operation.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds. The returned value will be equal to sizeof(USBIO_DEVICE_INFO).

Comments

The **USBIO_DEVICE_INFO** data structure returned by this IOCTL request includes a flag that indicates whether a USB 2.0 device operates in high speed mode or not. An application can use this information to detect if a USB 2.0 device is connected to a hub port that is high speed capable.

See Also

USBIO_DEVICE_INFO (page 117)
IOCTL_USBIO_GET_BANDWIDTH_INFO (page 89)

IOCTL_USBIO_GET_DRIVER_INFO

The IOCTL_USBIO_GET_DRIVER_INFO operation returns version information about the USBIO programming interface (API) and the USBIO driver executable that is currently running.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_DRIVER_INFO for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Points to a caller-provided **USBIO_DRIVER_INFO** data structure. This data structure will receive the results of the IOCTL operation.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This value has to be set to sizeof(USBIO_DRIVER_INFO) for this operation.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds. The returned value will be equal to sizeof(USBIO_DRIVER_INFO).

Comments

An application should check if the API version of the USBIO driver that is currently running matches with the version it expects. Newer versions of the USBIO driver API are compatible with older versions. However, the backward compatibility is maintained at the source code level. Thus, applications should be recompiled if a newer version of the USBIO driver is used.

If an application is compiled then the USBIO API version that the application is using is defined by the constant USBIO_API_VERSION in *usbio_i.h*. At runtime the application should always check that the API version of the USBIO driver that is installed in the system is equal to the expected API version defined at compile time by the USBIO_API_VERSION constant. This way, problems caused by version inconsistencies can be avoided.

Note that the USBIO API version and the USBIO driver version are maintained separately. The API version number will be incremented only if changes are made at the API level. The driver version number will be incremented for each USBIO release. Typically, an application does not need to check the USBIO driver version. It can display the driver version number for informational purposes, if desired.

See Also

USBIO_DRIVER_INFO (page 118)
IOCTL_USBIO_GET_DEVICE_INFO (page 90)
IOCTL_USBIO_GET_BANDWIDTH_INFO (page 89)

IOCTL_USBIO_CYCLE_PORT

The IOCTL_USBIO_CYCLE_PORT operation causes a reset at the hub port to which the device is connected and a new enumeration of the device.

`dwIoControlCode`
Set to IOCTL_USBIO_CYCLE_PORT for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

The IOCTL_USBIO_CYCLE_PORT request is similar to the **IOCTL_USBIO_RESET_DEVICE** request except that from a software point of view a device disconnect/connect is simulated. This request causes the following events to occur:

- The USBIO device object that is associated with the USB device will be removed. The corresponding device handles and pipe handles become invalid and should be closed by the application.
- The operating system starts a new enumeration of the device. The following events occur on the bus:
  ```
  USB Reset
  GET_DEVICE_DESCRIPTOR
  USB Reset
  SET_ADDRESS
  GET_DEVICE_DESCRIPTOR
  GET_CONFIGURATION_DESCRIPTOR
  ```
- A new device object instance is created by the USBIO driver.
- The application receives a PnP notification that informs it about the new device instance.

After an application issued this request it should close all handles for the current device. It can open the newly created device instance after it receives the appropriate PnP notification.

This request should be used instead of **IOCTL_USBIO_RESET_DEVICE** if the USB device modifies its descriptors during a USB Reset. Particularly, this is required to implement the Device Firmware Upgrade (DFU) device class specification. Note that the USB device receives two USB Resets after this call. This does not conform to the DFU specification.

However, this is the standard device enumeration method used by the Windows USB bus driver (USBD).

The IOCTL_USBIO_CYCLE_PORT request does not work if the system-provided multi-interface driver is used.

See Also

IOCTL_USBIO_RESET_DEVICE (page 84)

IOCTL_USBIO_ACQUIRE_DEVICE

The IOCTL_USBIO_ACQUIRE_DEVICE acquires the device for exclusive use in the process context where the call is made.

`dwIoControlCode`
Set to IOCTL_USBIO_ACQUIRE_DEVICE for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

If a different process has an open handle this operation will fail. This operation returns successfully if the process has already called IOCTL_USBIO_ACQUIRE_DEVICE. If the operation completes successfully, no other process can open a handle to this device until this process closes all handles or this process calls **IOCTL_USBIO_RELEASE_DEVICE**

See Also

IOCTL_USBIO_RELEASE_DEVICE (page 95)

IOCTL_USBIO_RELEASE_DEVICE

The IOCTL_USBIO_RELEASE_DEVICE terminates the exclusive use of this device for this process.

`dwIoControlCode`
Set to IOCTL_USBIO_RELEASE_DEVICE for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

If this function returns successfully other processes can open the device. If the last handle is closed the exclusive use is terminated automatically.

See Also

IOCTL_USBIO_RELEASE_DEVICE (page 95)

IOCTL_USBIO_BIND_PIPE

The IOCTL_USBIO_BIND_PIPE operation is used to establish a binding between a device handle and a pipe object.

`dwIoControlCode`
Set to IOCTL_USBIO_BIND_PIPE for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_BIND_PIPE** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_BIND_PIPE) for this operation.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

This IOCTL operation binds a device handle to a pipe object. The pipe is identified by its endpoint address. Only the endpoints that are active in the current configuration can be bound. After this operation is successfully completed the pipe can be accessed using pipe related requests, e.g. read or write requests.

A handle can only be bound to one pipe object. The binding can be deleted by means of **IOCTL_USBIO_UNBIND_PIPE** and the handle can be bound to another pipe object by calling IOCTL_USBIO_BIND_PIPE again. However, it is recommended to create a separate handle for each pipe that is used to transfer data. This will simplify the implementation of an application.

This IOCTL operation only modifies the internal driver state. It does not cause any action on the USB.

See Also

USBIO_BIND_PIPE (page 141)
IOCTL_USBIO_UNBIND_PIPE (page 97)
IOCTL_USBIO_SET_CONFIGURATION (page 74)
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 83)

IOCTL_USBIO_UNBIND_PIPE

The IOCTL_USBIO_UNBIND_PIPE operation deletes the binding between a device handle and a pipe object.

`dwIoControlCode`
Set to IOCTL_USBIO_UNBIND_PIPE for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

After this operation is successfully completed the handle is unbound and can be used to bind another pipe. However, it is recommended to create a separate handle for each pipe that is used to transfer data. This will simplify the implementation of an application.

The IOCTL_USBIO_UNBIND_PIPE request can safely be issued on a handle that is not bound to a pipe object. The request has no effect in this case. However, the IOCTL operation will be completed with an error status of USBIO_ERR_NOT_BOUND.

It is not necessary to unbind a pipe handle before it is closed. Closing a handle unbinds it implicitly.

As a side-effect, the IOCTL_USBIO_UNBIND_PIPE operation resets the statistical data of the pipe and disables the calculation of the mean bandwidth. It must be enabled and configured by means of the **IOCTL_USBIO_SETUP_PIPE_STATISTICS** request when the pipe is reused.

This IOCTL operation modifies the internal driver state only. It does not cause any action on the USB.

See Also

IOCTL_USBIO_BIND_PIPE (page 96)
IOCTL_USBIO_SETUP_PIPE_STATISTICS (page 102)

IOCTL_USBIO_RESET_PIPE

The IOCTL_USBIO_RESET_PIPE operation is used to clear an error condition on a pipe.

`dwIoControlCode`
Set to IOCTL_USBIO_RESET_PIPE for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

If an error occurs while transferring data to or from the endpoint that is associated with the pipe object then the USB bus driver USBD halts the pipe. No further data transfers can be performed while the pipe is halted. Any read or write request will be completed with an error status of USBIO_ERR_ENDPOINT_HALTED. To recover from this error condition and to restart the pipe an IOCTL_USBIO_RESET_PIPE request has to be issued on the pipe.

The IOCTL_USBIO_RESET_PIPE operation causes a CLEAR_FEATURE(ENDPOINT_STALL) request on the USB. In addition, the endpoint processing in the USB host controller will be reinitialized.

Isochronous pipes will never be halted by the USB bus driver USBD. This is because on isochronous pipes, no handshake protocol is used to detect errors in the data transmission.

This IOCTL operation requires that the handle on which the operation is performed has been bound to a pipe object by means of **IOCTL_USBIO_BIND_PIPE**. Otherwise, the IOCTL operation will fail with an error status of USBIO_ERR_NOT_BOUND.

See Also

IOCTL_USBIO_ABORT_PIPE (page 99)
IOCTL_USBIO_BIND_PIPE (page 96)

IOCTL_USBIO_ABORT_PIPE

The IOCTL_USBIO_ABORT_PIPE operation is used to cancel all outstanding read and write requests on a pipe.

`dwIoControlCode`
Set to IOCTL_USBIO_ABORT_PIPE for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

All outstanding read or write requests on the pipe will be aborted and returned with an error status of USBIO_ERR_CANCELED.

This IOCTL operation requires that the handle on which the operation is performed has been bound to a pipe object by means of **IOCTL_USBIO_BIND_PIPE**. Otherwise, the IOCTL operation will fail with an error status of USBIO_ERR_NOT_BOUND.

See Also

IOCTL_USBIO_RESET_PIPE (page 98)
IOCTL_USBIO_BIND_PIPE (page 96)

IOCTL_USBIO_GET_PIPE_PARAMETERS

The IOCTL_USBIO_GET_PIPE_PARAMETERS operation returns USBIO driver settings related to a pipe.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_PIPE_PARAMETERS for this operation.

`lpInBuffer`
Not used with this operation. Set to NULL.

`nInBufferSize`
Not used with this operation. Set to zero.

`lpOutBuffer`
Points to a caller-provided **USBIO_PIPE_PARAMETERS** data structure. This data structure will receive the results of the IOCTL operation.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This value has to be set to sizeof(USBIO_PIPE_PARAMETERS) for this operation.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds. The returned value will be equal to sizeof(USBIO_PIPE_PARAMETERS).

Comments

The default state of pipe-related settings is defined by a set of registry parameters which are read by the USBIO driver at startup. The current state can be retrieved by means of this request.

Note that a separate set of pipe settings is maintained per pipe object. The IOCTL_USBIO_GET_PIPE_PARAMETERS request retrieves the actual settings of that pipe object that is bound to the handle on which the request is issued.

This IOCTL operation requires that the handle on which the operation is performed has been bound to a pipe object by means of **IOCTL_USBIO_BIND_PIPE**. Otherwise, the IOCTL operation will fail with an error status of USBIO_ERR_NOT_BOUND.

This IOCTL operation retrieves internal driver settings. It does not cause any action on the USB.

See Also

USBIO_PIPE_PARAMETERS (page 142)
IOCTL_USBIO_SET_PIPE_PARAMETERS (page 101)
IOCTL_USBIO_GET_DEVICE_PARAMETERS (page 81)
IOCTL_USBIO_SET_DEVICE_PARAMETERS (page 82)
IOCTL_USBIO_BIND_PIPE (page 96)

IOCTL_USBIO_SET_PIPE_PARAMETERS

The IOCTL_USBIO_SET_PIPE_PARAMETERS operation is used to set USBIO driver settings related to a pipe.

`dwIoControlCode`
Set to IOCTL_USBIO_SET_PIPE_PARAMETERS for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_PIPE_PARAMETERS** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_PIPE_PARAMETERS) for this operation.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

The default state of pipe-related settings is defined by a set of registry parameters which are read by the USBIO driver at startup. The current state can be modified by means of this request.

Note that a separate set of pipe settings is maintained per pipe object. The IOCTL_USBIO_SET_PIPE_PARAMETERS request modifies the settings of that pipe object that is bound to the handle on which the request is issued.

This IOCTL operation requires that the handle on which the operation is performed has been bound to a pipe object by means of **IOCTL_USBIO_BIND_PIPE**. Otherwise, the IOCTL operation will fail with an error status of USBIO_ERR_NOT_BOUND.

This IOCTL operation modifies internal driver settings. It does not cause any action on the USB.

See Also

USBIO_PIPE_PARAMETERS (page 142)
IOCTL_USBIO_GET_PIPE_PARAMETERS (page 100)
IOCTL_USBIO_GET_DEVICE_PARAMETERS (page 81)
IOCTL_USBIO_SET_DEVICE_PARAMETERS (page 82)
IOCTL_USBIO_BIND_PIPE (page 96)

IOCTL_USBIO_SETUP_PIPE_STATISTICS

The IOCTL_USBIO_SETUP_PIPE_STATISTICS request enables or disables a statistical analysis of the data transfer on a pipe.

`dwIoControlCode`
Set to IOCTL_USBIO_SETUP_PIPE_STATISTICS for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_SETUP_PIPE_STATISTICS** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_SETUP_PIPE_STATISTICS) for this operation.

`lpOutBuffer`
Not used with this operation. Set to NULL.

`nOutBufferSize`
Not used with this operation. Set to zero.

`lpBytesReturned`
Points to a caller-provided DWORD variable. The value of this variable is meaningless in the context of this IOCTL operation.

Comments

The USBIO driver is able to analyse the data transfer (outgoing or incoming) on a pipe and to calculate the average data rate on that pipe. A time averaging algorithm is used to continuously compute the mean value of the data transfer rate. In order to save resources (kernel memory and CPU cycles) the average data rate computation is disabled by default. It has to be enabled and to be configured by means of the IOCTL_USBIO_SETUP_PIPE_STATISTICS request before it is available to an application. See also **IOCTL_USBIO_QUERY_PIPE_STATISTICS** and **USBIO_PIPE_STATISTICS** for more information on pipe statistics.

Note that the statistical data is maintained separately for each pipe object. The IOCTL_USBIO_SETUP_PIPE_STATISTICS request has an effect on that pipe object only that is bound to the handle on which the request is issued.

If a pipe is unbound from the device handle by means of the **IOCTL_USBIO_UNBIND_PIPE** operation or by closing the handle then the average data rate computation will be disabled. It has to be enabled and configured when the pipe is reused. In other words, if the data rate computation is needed by an application then the IOCTL_USBIO_SETUP_PIPE_STATISTICS request should be issued immediately after the pipe is bound by means of the **IOCTL_USBIO_BIND_PIPE** operation.

This IOCTL operation requires that the handle on which the operation is performed has been bound to a pipe object by means of **IOCTL_USBIO_BIND_PIPE**. Otherwise, the IOCTL operation will fail with an error status of USBIO_ERR_NOT_BOUND.

This IOCTL operation modifies internal driver settings. It does not cause any action on the USB.

See Also

USBIO_SETUP_PIPE_STATISTICS (page 143)
USBIO_PIPE_STATISTICS (page 146)
IOCTL_USBIO_QUERY_PIPE_STATISTICS (page 104)
IOCTL_USBIO_BIND_PIPE (page 96)

IOCTL_USBIO_QUERY_PIPE_STATISTICS

The IOCTL_USBIO_QUERY_PIPE_STATISTICS operation returns statistical data related to a pipe.

`dwIoControlCode`
Set to IOCTL_USBIO_QUERY_PIPE_STATISTICS for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_QUERY_PIPE_STATISTICS** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_QUERY_PIPE_STATISTICS) for this operation.

`lpOutBuffer`
Points to a caller-provided **USBIO_PIPE_STATISTICS** data structure. This data structure will receive the results of the IOCTL operation.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This value has to be set to sizeof(USBIO_PIPE_STATISTICS) for this operation.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds. The returned value will be equal to sizeof(USBIO_PIPE_STATISTICS).

Comments

The USBIO device driver internally maintains some statistical data per pipe object. This IOCTL request allows an application to query the actual values of the various statistics counters. Optionally, individual counters can be reset to zero after queried. See **USBIO_QUERY_PIPE_STATISTICS** and **USBIO_PIPE_STATISTICS** for more information on pipe statistics.

The USBIO device driver is able to analyse the data transfer (outgoing or incoming) on a pipe and to calculate the average data rate on that pipe. In order to save resources (kernel memory and CPU cycles) this feature is disabled by default. It has to be enabled and to be configured by means of the **IOCTL_USBIO_SETUP_PIPE_STATISTICS** request before it is available to an application. Thus, before an application starts to (periodically) query the value of `AverageRate` that is included in the data structure **USBIO_PIPE_STATISTICS** it has to enable the continuous computation of this value by issuing an **IOCTL_USBIO_SETUP_PIPE_STATISTICS** request. The other statistical counters contained in the **USBIO_PIPE_STATISTICS** structure will be updated by default and do not need to be enabled explicitly.

Note that the statistical data is maintained separately for each pipe object. The IOCTL_USBIO_QUERY_PIPE_STATISTICS request retrieves the actual statistics of that pipe object that is bound to the handle on which the request is issued.

This IOCTL operation requires that the handle on which the operation is performed has been bound to a pipe object by means of **IOCTL_USBIO_BIND_PIPE**. Otherwise, the IOCTL operation will fail with an error status of USBIO_ERR_NOT_BOUND.

This IOCTL operation retrieves internal driver information. It does not cause any action on the USB.

See Also

USBIO_QUERY_PIPE_STATISTICS (page 144)
USBIO_PIPE_STATISTICS (page 146)
IOCTL_USBIO_SETUP_PIPE_STATISTICS (page 102)
IOCTL_USBIO_BIND_PIPE (page 96)

IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN

The IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN operation is used to generate a specific request (SETUP packet) for a control pipe with a data transfer direction from device to host.

`dwIoControlCode`
Set to IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_PIPE_CONTROL_TRANSFER** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_PIPE_CONTROL_TRANSFER) for this operation.

`lpOutBuffer`
Points to a caller-provided buffer that will receive the data bytes transferred from the device during the data phase of the control transfer. If the SETUP request does not return any data this value can be set to NULL. `nOutBufferSize` has to be set to zero in this case.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This is equal to the length, in bytes, of the data transfer phase of the SETUP request. If this value is set to zero then there is no data transfer phase. `lpOutBuffer` should be set to NULL in this case.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes returned in the buffer pointed to by `lpOutBuffer` if the request succeeds.

Comments

This request is intended to be used with additional control pipes a device might provide. It is not possible to generate a control transfer for the default endpoint zero by means of this IOCTL operation.

If the request is completed successfully then the variable pointed to by `lpBytesReturned` is set to the number of data bytes successfully transferred during the data transfer phase of the control transfer.

This IOCTL operation requires that the handle on which the operation is performed has been bound to a pipe object by means of **IOCTL_USBIO_BIND_PIPE**. Otherwise, the IOCTL operation will fail with an error status of USBIO_ERR_NOT_BOUND.

See Also

USBIO_PIPE_CONTROL_TRANSFER (page 148)
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT (page 107)
IOCTL_USBIO_BIND_PIPE (page 96)

IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT

The IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT operation is used to generate a specific request (SETUP packet) for a control pipe with a data transfer direction from host to device.

`dwIoControlCode`
Set to IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT for this operation.

`lpInBuffer`
Points to a caller-provided **USBIO_PIPE_CONTROL_TRANSFER** data structure. This data structure has to be initialized by the caller. It provides input parameters for the IOCTL operation.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`. This value has to be set to sizeof(USBIO_PIPE_CONTROL_TRANSFER) for this operation.

`lpOutBuffer`
Points to a caller-provided buffer that contains the data bytes to be transferred to the device during the data phase of the control transfer. If the SETUP request does not have a data phase this value can be set to NULL. `nOutBufferSize` has to be set to zero in this case.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. This is equal to the length, in bytes, of the data transfer phase of the SETUP request. If this value is set to zero then there is no data transfer phase. `lpOutBuffer` should be set to NULL in this case.

`lpBytesReturned`
Points to a caller-provided DWORD variable that will be set to the number of bytes transferred from the buffer pointed to by `lpOutBuffer` if the request succeeds.

Comments

This request is intended to be used with additional control pipes a device might provide. It is not possible to generate a control transfer for the default endpoint zero by means of this IOCTL operation.

If the request is completed successfully then the variable pointed to by `lpBytesReturned` is set to the number of data bytes successfully transferred during the data transfer phase of the control transfer.

Although the data buffer is described by the parameters `lpOutBuffer` and `nOutBufferSize` it provides input data for the request. Some confusion is caused by the naming scheme of the Windows API.

This IOCTL operation requires that the handle on which the operation is performed has been bound to a pipe object by means of **IOCTL_USBIO_BIND_PIPE**. Otherwise, the IOCTL operation will fail with an error status of USBIO_ERR_NOT_BOUND.

See Also

USBIO_PIPE_CONTROL_TRANSFER (page 148)
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN (page 106)
IOCTL_USBIO_BIND_PIPE (page 96)

7.3 Data Transfer Requests

The USBIO device driver exports an interface to USB pipes that is similar to files. For that reason the Win32 API functions `ReadFile()` and `WriteFile()` are used to transfer data to or from a pipe. The handle that is associated with the USB pipe is passed as `hFile` to this functions.

The `ReadFile()` function is defined as follows:

```
BOOL ReadFile(
        HANDLE hFile,     // handle of file to read
        LPVOID lpBuffer,  // pointer to buffer that receives data
        DWORD nNumberOfBytesToRead,  // number of bytes to read
        LPDWORD lpNumberOfBytesRead, // pointer to number of bytes read
        LPOVERLAPPED lpOverlapped    // pointer to OVERLAPPED structure
        );
```

The `WriteFile()` function is defined as follows:

```
BOOL WriteFile(
        HANDLE hFile,     // handle of file to write
        LPVOID lpBuffer,  // pointer to data to write to file
        DWORD nNumberOfBytesToWrite,   // number of bytes to write
        LPDWORD lpNumberOfBytesWritten,// pointer to number of bytes written
        LPOVERLAPPED lpOverlapped      // pointer to OVERLAPPED structure
        );
```

By using these functions it is possible to implement both synchronous and asynchronous data transfer operations. Both methods are fully supported by the USBIO driver. Refer to the Microsoft Platform SDK documentation for more information on using the `ReadFile()` and `WriteFile()` functions.

7.3.1 Bulk and Interrupt Transfers

For interrupt and bulk transfers the buffer size can be larger than the maximum packet size of the endpoint (physical FIFO size) as reported in the endpoint descriptor. However the buffer size must be equal to or smaller than the value specified in the MaximumTransferSize field of the **USBIO_INTERFACE_SETTING** structure on the Set Configuration call.

Bulk or Interrupt Write Transfers

The write operation is used to transfer data from the host (PC) to the USB device. The buffer is divided into data pieces (packets) of the endpoint's FIFO size. These packets are sent to the USB device. If the last packet of the buffer is smaller than the FIFO size, a smaller data packet is transferred. If the size of the last packet of the buffer is equal to the FIFO size this packet is sent. No additional zero packet is sent automatically. To send a data packet with length zero, set the buffer length to zero and use a NULL buffer pointer.

Bulk or Interrupt Read Transfers

The read operation is used to transfer data from the USB device to the host (PC). The buffer is divided into data pieces (packets) of the endpoint's FIFO size. The buffer size should be a multiple of the FIFO size, otherwise the last transaction can cause a buffer overflow error.

A read operation will be completed if the whole buffer is filled or a short packet is transmitted. A short packet is a packet that is smaller than the endpoint's FIFO size. To read a data packet with a length of zero, the buffer size has to be at least one byte. A read operation with a NULL buffer pointer will be completed with success without performing a read operation on the USB.

The behaviour during a read operation depends on the state of the flag `USBIO_SHORT_TRANSFER_OK` of the related pipe. This setting may be changed by using the **IOCTL_USBIO_SET_PIPE_PARAMETERS** operation. The default state is defined by the registry parameter `ShortTransferOk`. If the flag `USBIO_SHORT_TRANSFER_OK` is set a read operation that returns a data packet that is shorter than the endpoint's FIFO size is completed with success. Otherwise, every data packet from the endpoint that is smaller than the FIFO size will caus an error.

7.3.2 Isochronous Transfers

For isochronous transfers the data buffer that is passed to the `ReadFile()` or `WriteFile()` function has to contain a header that describes the location and the size of the data packets to be transferred. The rest of the buffer is divided into packets. Each packet is transmitted within a USB frame or microframe respectively. The packet size can vary for each frame. Even a packet size of zero bytes is allowed. This way, any data rate of the isochronous stream is supported.

In full-speed mode (12 Mbit/s), one isochronous packet is transmitted per USB frame. A USB frame corresponds to 1 millisecond. Thus, one packet is transferred per millisecond.

A USB 2.0 compliant device that operates in high-speed mode (480 Mbit/s) reports the isochronous frame rate for each isochronous endpoint in the corresponding endpoint descriptor. Normally, one packet is transferred per microframe. A microframe corresponds to 125 microseconds. However, it is possible to request multiple packet transfers per microframe. It is also possible to reduce the frame rate and to transfer one isochronous packet every N microframes.

The layout of a buffer that holds isochronous data is shown in figure 6. At the beginning, the buffer contains a **USBIO_ISO_TRANSFER_HEADER** structure of variable size. The rest of the buffer holds the data packets. The header contains a **USBIO_ISO_TRANSFER** structure that provides general information about the transfer buffer. An important member of this structure is NumberOfPackets. This parameter specifies the number of isochronous data packets contained in the transfer buffer. The maximum number of packets that can be used in a single transfer is limited by the USBIO configuration parameter `MaxIsoPackets`, which is defined in the registry. See also section 10 (page 303) for more information.

Each data packet that is contained in the buffer must be described by a **USBIO_ISO_PACKET** structure. For that purpose, the header contains an array of **USBIO_ISO_PACKET** structures. Because the number of packets contained in a buffer is variable, the size of this array is variable as well.

The Offset member of the **USBIO_ISO_PACKET** structure specifies the byte offset of the corresponding packet relative to the beginning of the whole buffer. The offset of each isochronous data packet has to be specified by the application for both read and write transfers. The Length member defines the length, in bytes, of the corresponding packet. For write transfers, the length of each isochronous data packet has to be specified by the application before the transfer is initiated. For read transfers, the length of each packet is returned by the USBIO driver after the transfer is finished. On both read and write operations, the Status member of **USBIO_ISO_PACKET** is

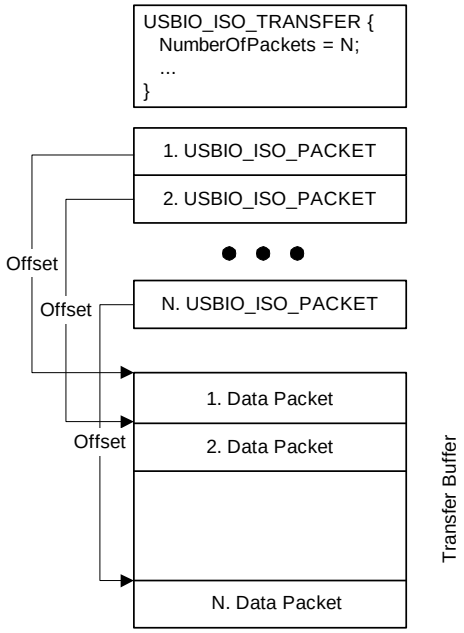

Figure 6: Layout of an isochronous transfer buffer

used to return the transfer completion status for the corresponding packet.

Isochronous data require a guaranteed bandwidth on the bus. To make sure that the host controller sends a gapless stream of IN or OUT tokens on the bus, there must be always a pending data transfer buffer at the host controller. To archive this the application must use the asynchronous (overlapped) IO requests. Some of the API's, such as the USBIOLIB, USBIOCOM interface, and the USBIO API DLL, have an internal support for buffer circulation and asynchronous IO.

The total number of frames in pending requests to one ISO pipe is limited to 1024 by the API documentation of the USBD driver. This number is calculated as the product of the number of pending buffers and the parameter NumberOfPackets. On Windows XP and XP SP1 this product must be less or equal to 256, see problems.txt.

Isochronous Write Transfers

There are some constraints that apply to isochronous write operations. The length of each isochronous packet must be less than or equal to the FIFO size of the respective endpoint. The data packets must be placed contiguously into the buffer. In other words, no gaps between packets are allowed. The Offset and Length member of all **USBIO_ISO_PACKET** structures must be initialized correctly by the application **before** the transfer is initiated.

Isochronous Read Transfers

There are some constraints that apply to isochronous read operations. The length of each packet reserved in the buffer should be equal to the FIFO size of the respective endpoint, otherwise, a data overrun error can occur. The data packets must be placed contiguously into the buffer. In other

words, no gaps between packets are allowed. The Offset member of all **USBIO_ISO_PACKET** structures must be initialized correctly by the application **before** the transfer is initiated. The length of each isochronous data packet received from the device is returned in the Length member of the corresponding **USBIO_ISO_PACKET** structure when the transfer of the whole buffer completes.

Note:
Because the length of an isochronous data packet that is received from the device may be smaller than the FIFO size, the data packets are not placed contiguously into the buffer. After the transfer of a buffer is complete, an application needs to evaluate the Length member of all **USBIO_ISO_PACKET** structures to learn about the amount of valid data available in the corresponding packet.

7.4 Error Handling

All data transfers on USB are protected with a CRC. The data transfers on bulk, interrupt, and control endpoints are additionally protected with a hardware handshake (ACK, NAK) and a hardware retry. This means if the receiver does not correctly receives a data block it does not send a handshake and the host controller retries the data transmission up to three times. The control endpoint has an additional error protection mechanism with a software-generated handshake signal which is a zero length data packet.

Nevertheless, a data transmission on the USB may be disturbed. The software on the PC and on the device site should implement a protocol to continue the data transfer after a transmission error. The following sections describe possible error handling mechanism for different transmission types.

7.4.1 Control Transfers

Typically, a control transfer changes the state of the device. If a control transfer fails, the PC can resubmit the same transfer. All possible error conditions on the control endpoint are automatically cleared with the next setup. If an error occurs, the PC software cannot determine if the error occurred in the handshake phase or in the setup phase. This means it cannot recover whether the setup request was performed partially or completely. This is not a problem if the setup sets an absolute state (e.g. a register value or a memory block) but may be a problem if a state is changed incrementally (e.g. a counter is incremented with a request). In this case a more complex error recovery is required.

7.4.2 Bulk and Interrupt Transfers

If a transmission error on a bulk or interrupt pipe occurs the host controller driver sets an internal error condition. This error condition must be reset with a call to the function ResetPipe or **IOCTL_USBIO_RESET_PIPE** by the PC software. This function call creates the standard request Clear Feature Endpoint Halt, which contains the endpoint number, on the USB. This request informs the device that the data transfer was completed with a error. Error handling can be performed in two ways:

- With data lost
- Without data lost

In the case of data loss, error handling is very simple. The device clears the related endpoint FIFO and continues sending or receiving data. The PC then submits the next buffer. This method can be used if a higher level protocol can run on a unreliable data connection.

To perform error recovery without data loss, both sites, the PC and the device software, must agree on a logical block size. The logical block size must be a multiple of the FIFO size. The device must be able to store a logical block until the last FIFO sized chunk is transferred successfully. Short transfers are allowed and must be finished with a USB short packet. If the device receives a Clear Feature Endpoint Halt it must clear the FIFO and reset the logical buffer. On an IN pipe (data transfer from device to PC) the device must resubmit the logical buffer from the beginning. On an OUT pipe it must clear the logical buffer and expect to receive the data of this logical block again. If the PC software works asynchronously (overlapped) it must stop the buffer circulation with

a call to the function AbortPipe or **IOCTL_USBIO_ABORT_PIPE**. It than calls the function ResetPipe or **IOCTL_USBIO_RESET_PIPE**. After this, it restarts the reading or writing on the pipe with all buffers then have been not yet successfully transmitted.

One problem with this method is the selection of a reasonable logical buffer size. If the logical buffer size is too small the data rate may be low and the CPU load too high. If the logical buffer size is large the device must spend a larger amount of memory on this buffer.

7.4.3 Isochronous Transfers

Isochronous transfers contain a CRC but are not handshaked. The sender of a isochronous data packet cannot determine if the data block is transferred correctly. The receiver can check the CRC for each packet. The USB specification defines that isochronous endpoints are never stopped. However the Windows host controller driver sets a internal error condition if the PC is too slow to transfer a continuos data stream. This error condition must be cleared with a call to the function ResetPipe or **IOCTL_USBIO_RESET_PIPE**. The host controller driver does not send a Clear Feature Endpoint Halt request on the bus.

A data transfer with error causes data loss on an isochronous pipe.

Tests in our laboratories have shown that an isochronous IN pipe with a data rate of 128 kbit on a full speed connection can run for 72 hours without error. On the other hand we completed a field test with about 100 PCs. We saw two PCs with a transmission error on the isochronous pipe about every 2 minutes. This shows the USB error rate can be very low but on some PCs the error rate can be much higher. It may be interesting to know that other devices such as memory sticks or still image cameras worked fine on the PCs that caused isochronous errors.

7.4.4 Testing Error Handling

Because the error rate on the USB is very low, it is difficult to test the correct behavior of error handling implementation. We tried to inject electrical signals into the USB lines. However if the injection disturbs the SOF, the parent HUB disables the device and error handling cannot be tested. A better idea is that the receiving site causes the receiving site causes a hardware error on a random or periodic basis. The PC software can do this by reading with a buffer size of only one byte. This causes the HC to report a buffer overrun error if the device sends more than one byte. The device can emulate a transmission error by setting the endpoint to stall. The PC software should handle the stall as a normal hardware error such as a CRC, Bitstuf, or EXACT error. If the error handling implementation is correct and error recovery without data loss is implemented, no data should be lost.

7.5 Data Structures

This section provides a detailed description of the data structures that are used in conjunction with the various input and output requests.

USBIO_BANDWIDTH_INFO

The USBIO_BANDWIDTH_INFO structure contains information on the USB bandwidth consumption.

Definition

```
typedef struct _USBIO_BANDWIDTH_INFO{
  ULONG TotalBandwidth;
  ULONG ConsumedBandwidth;
  ULONG reserved1;
  ULONG reserved2;
} USBIO_BANDWIDTH_INFO;
```

Members

TotalBandwidth
This field contains the total bandwidth, in kilobits per second, available on the bus. This bandwidth is provided by the USB host controller the device is connected to.

ConsumedBandwidth
This field contains the mean bandwidth that is already in use, in kilobits per second.

reserved1
This member is reserved for future use.

reserved2
This member is reserved for future use.

Comments

This structure returns the results of the **IOCTL_USBIO_GET_BANDWIDTH_INFO** operation.

See Also

IOCTL_USBIO_GET_BANDWIDTH_INFO (page 89)

USBIO_DEVICE_INFO

The USBIO_DEVICE_INFO structure contains information on the USB device.

Definition

```
typedef struct _USBIO_DEVICE_INFO{
  ULONG Flags;
  ULONG OpenCount;
  ULONG reserved1;
  ULONG reserved2;
} USBIO_DEVICE_INFO;
```

Members

Flags
This field contains zero or any combination (bitwise OR) of the following values.

USBIO_DEVICE_INFOFLAG_HIGH_SPEED
If this flag is set then the USB device operates in high speed mode. USB 2.0 devices should be connected to a hub port that is high speed capable.

Note that this flag does not indicate whether a device is capable of high speed operation, but rather whether it is in fact operating at high speed.

OpenCount
This member returns the number of open file handles to the device, including the handle which is used to submit this request.

reserved1
This member is reserved for future use.

reserved2
This member is reserved for future use.

Comments

This structure returns the results of the **IOCTL_USBIO_GET_DEVICE_INFO** operation.

See Also

IOCTL_USBIO_GET_DEVICE_INFO (page 90)

USBIO_DRIVER_INFO

The USBIO_DRIVER_INFO structure contains version information about the USBIO programming interface (API) and the USBIO driver executable.

Definition

```
typedef struct _USBIO_DRIVER_INFO{
  USHORT APIVersion;
  USHORT DriverVersion;
  ULONG DriverBuildNumber;
  ULONG Flags;
} USBIO_DRIVER_INFO;
```

Members

APIVersion
Contains the version number of the application programming interface (API) the driver supports. The format is as follows: upper 8 bit = major version, lower 8 bit = minor version. The numbers are encoded in BCD format. For example, V1.41 is represented by a numerical value of 0x0141.

The API version number will be incremented if changes are made at the API level. An application should check the API version at runtime. Refer to the description of the **IOCTL_USBIO_GET_DRIVER_INFO** request for detailed information on how this should be implemented.

DriverVersion
Contains the version number of the driver executable. The format is as follows: upper 8 bit = major version, lower 8 bit = minor version (Not in BCD format). For example, V1.41 is represented by a numerical value of 0x0129.

The driver version number will be incremented for each USBIO release. Typically, an application uses the driver version number only for informational purposes. Refer to the description of the **IOCTL_USBIO_GET_DRIVER_INFO** request for more information.

DriverBuildNumber
Contains the build number of the driver executable. This number will be incremented for each build of the USBIO driver executable. The driver build number should be read as an extension of the driver version number.

Flags
This field contains zero if the USBIO driver executable is a full version release build without any restrictions. Otherwise, this field contains any combination (bitwise OR) of the following values.

USBIO_INFOFLAG_CHECKED_BUILD
If this flag is set then the driver executable is a checked (debug) build. The checked driver executable provides additional tracing and debug features.

USBIO_INFOFLAG_DEMO_VERSION
If this flag is set then the driver executable is a DEMO version. The DEMO version has some restrictions. Refer to the file *ReadMe.txt* included in the USBIO package for a detailed description of these restrictions.

USBIO_INFOFLAG_LIGHT_VERSION
ATTENTION: The LIGHT version is no longer provided. The flag is documented only for compatibility with older versions.

If this flag is set then the driver executable is a LIGHT version. The LIGHT version has some functional restrictions.

USBIO_INFOFLAG_VS_LIGHT_VERSION
ATTENTION: The Vendor-Specific LIGHT version is no longer provided. The flag is documented only for compatibility with older versions.

If this flag is set in addition to USBIO_INFOFLAG_LIGHT_VERSION the driver executable is a Vendor-Specific LIGHT version that has specific restrictions. Refer to the file *ReadMe.txt* included in the USBIO package for a detailed description of these restrictions.

Comments

This structure returns the results of the **IOCTL_USBIO_GET_DRIVER_INFO** operation.

See Also

IOCTL_USBIO_GET_DRIVER_INFO (page 91)

USBIO_DESCRIPTOR_REQUEST

The USBIO_DESCRIPTOR_REQUEST structure provides information used to get or set a descriptor.

Definition

```
typedef struct _USBIO_DESCRIPTOR_REQUEST{
  USBIO_REQUEST_RECIPIENT Recipient;
  UCHAR DescriptorType;
  UCHAR DescriptorIndex;
  USHORT LanguageId;
} USBIO_DESCRIPTOR_REQUEST;
```

Members

`Recipient`
Specifies the recipient of the get or set descriptor request. The values are defined by the enumeration type **USBIO_REQUEST_RECIPIENT**.

`DescriptorType`
Specifies the type of descriptor to get or set. The values are defined by the Universal Serial Bus Specification 1.1, Chapter 9 and additional USB device class specifications.

Value	Meaning
1	Device Descriptor
2	Configuration Descriptor
3	String Descriptor
4	Interface Descriptor
5	Endpoint Descriptor
21	HID Descriptor

`DescriptorIndex`
Specifies the index of the descriptor to get or set.

`LanguageId`
Specifies the Language ID of the descriptor to get or set. This is used for string descriptors only. This field is set to zero for other descriptors.

Comments

This structure provides the input parameters for the **IOCTL_USBIO_GET_DESCRIPTOR** and the **IOCTL_USBIO_SET_DESCRIPTOR** operation.

See Also

USBIO_REQUEST_RECIPIENT (page 154)
IOCTL_USBIO_GET_DESCRIPTOR (page 66)
IOCTL_USBIO_SET_DESCRIPTOR (page 67)

USBIO_FEATURE_REQUEST

The USBIO_FEATURE_REQUEST structure provides information used to set or clear a specific feature.

Definition

```
typedef struct _USBIO_FEATURE_REQUEST{
  USBIO_REQUEST_RECIPIENT Recipient;
  USHORT FeatureSelector;
  USHORT Index;
} USBIO_FEATURE_REQUEST;
```

Members

Recipient
Specifies the recipient of the set feature or clear feature request. The values are defined by the enumeration type **USBIO_REQUEST_RECIPIENT**.

FeatureSelector
Specifies the feature selector value for the set feature or clear feature request. The values are defined by the recipient. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

Index
Specifies the index value for the set feature or clear feature request. The values are defined by the device. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

Comments

This structure provides the input parameters for the **IOCTL_USBIO_SET_FEATURE** and the **IOCTL_USBIO_CLEAR_FEATURE** operation.

See Also

USBIO_REQUEST_RECIPIENT (page 154)
IOCTL_USBIO_SET_FEATURE (page 68)
IOCTL_USBIO_CLEAR_FEATURE (page 69)

USBIO_STATUS_REQUEST

The USBIO_STATUS_REQUEST structure provides information used to request status for a specified recipient.

Definition

```
typedef struct _USBIO_STATUS_REQUEST{
  USBIO_REQUEST_RECIPIENT Recipient;
  USHORT Index;
} USBIO_STATUS_REQUEST;
```

Members

Recipient
Specifies the recipient of the get status request. The values are defined by the enumeration type **USBIO_REQUEST_RECIPIENT**.

Index
Specifies the index value for the get status request. The values are defined by the device. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

Comments

This structure provides the input parameters for the **IOCTL_USBIO_GET_STATUS** operation.

See Also

USBIO_REQUEST_RECIPIENT (page 154)
IOCTL_USBIO_GET_STATUS (page 70)

USBIO_STATUS_REQUEST_DATA

The USBIO_STATUS_REQUEST_DATA structure contains information returned by a get status operation.

Definition

```
typedef struct _USBIO_STATUS_REQUEST_DATA{
  USHORT Status;
} USBIO_STATUS_REQUEST_DATA;
```

Member

Status
Contains the 16-bit value that is returned by the recipient in response to the get status request. The interpretation of the value is specific to the recipient. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

Comments

This structure returns the results of the **IOCTL_USBIO_GET_STATUS** operation.

See Also

IOCTL_USBIO_GET_STATUS (page 70)

USBIO_GET_CONFIGURATION_DATA

The USBIO_GET_CONFIGURATION_DATA structure contains information returned by a get configuration operation.

Definition

```
typedef struct _USBIO_GET_CONFIGURATION_DATA{
  UCHAR ConfigurationValue;
} USBIO_GET_CONFIGURATION_DATA;
```

Member

ConfigurationValue
Contains the 8-bit value that is returned by the device in response to the get configuration request. The meaning of the value is defined by the device. A value of zero means the device is not configured. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

Comments

This structure returns the results of the **IOCTL_USBIO_GET_CONFIGURATION** operation.

See Also

IOCTL_USBIO_GET_CONFIGURATION (page 71)

USBIO_GET_INTERFACE

The USBIO_GET_INTERFACE structure provides information used to query the current alternate setting of an interface.

Definition

```
typedef struct _USBIO_GET_INTERFACE{
  USHORT Interface;
} USBIO_GET_INTERFACE;
```

Member

Interface
Specifies the interface number of the interface to be queried. The values are defined by the device. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

Comments

This structure provides the input parameters for the **IOCTL_USBIO_GET_INTERFACE** operation.

See Also

IOCTL_USBIO_GET_INTERFACE (page 72)

USBIO_GET_INTERFACE_DATA

The USBIO_GET_INTERFACE_DATA structure contains information returned by a get interface operation.

Definition

```
typedef struct _USBIO_GET_INTERFACE_DATA{
  UCHAR AlternateSetting;
} USBIO_GET_INTERFACE_DATA;
```

Member

AlternateSetting
Contains the 8-bit value that is returned by the device in response to a get interface request. The interpretation of the value is specific to the device. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

Comments

This structure returns the results of the **IOCTL_USBIO_GET_INTERFACE** operation.

See Also

IOCTL_USBIO_GET_INTERFACE (page 72)

USBIO_INTERFACE_SETTING

The USBIO_INTERFACE_SETTING structure provides information used to configure an interface and its endpoints.

Definition

```
typedef struct _USBIO_INTERFACE_SETTING{
  USHORT InterfaceIndex;
  USHORT AlternateSettingIndex;
  ULONG MaximumTransferSize;
} USBIO_INTERFACE_SETTING;
```

Members

InterfaceIndex
Specifies the interface number of the interface to be configured. The values are defined by the device. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

AlternateSettingIndex
Specifies the alternate setting to be set for the interface. The values are defined by the device. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

MaximumTransferSize
Specifies the maximum length, in bytes, of data transfers to or from endpoints of this interface. The value is user-defined and is valid for all endpoints of this interface. If no special requirement exists a value of 4096 (4K) should be used.

Comments

This structure provides input parameters for the **IOCTL_USBIO_SET_INTERFACE** and the **IOCTL_USBIO_SET_CONFIGURATION** operation.

See Also

IOCTL_USBIO_SET_INTERFACE (page 77)
IOCTL_USBIO_SET_CONFIGURATION (page 74)
USBIO_SET_CONFIGURATION (page 129)

USBIO_SET_CONFIGURATION

The USBIO_SET_CONFIGURATION structure provides information used to set the device configuration.

Definition

```
typedef struct _USBIO_SET_CONFIGURATION{
  USHORT ConfigurationIndex;
  USHORT NbOfInterfaces;
  USBIO_INTERFACE_SETTING
        InterfaceList[USBIO_MAX_INTERFACES];
} USBIO_SET_CONFIGURATION;
```

Members

`ConfigurationIndex`
Specifies the configuration to be set as a zero-based index. The given index is used to query the associated configuration descriptor (by means of a GET_DESCRIPTOR request). The configuration value that is contained in the configuration descriptor is used for the SET_CONFIGURATION request. The configuration value is defined by the device.

For single-configuration devices the only valid value for `ConfigurationIndex` is zero.

Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

`NbOfInterfaces`
Specifies the number of interfaces in this configuration. This is the number of valid entries in `InterfaceList`.

`InterfaceList[USBIO_MAX_INTERFACES]`
An array of **USBIO_INTERFACE_SETTING** structures that describes each interface in the configuration. There have to be `NbOfInterfaces` valid entries in this array.

Comments

This structure provides the input parameters for the **IOCTL_USBIO_SET_CONFIGURATION** operation.

See Also

USBIO_INTERFACE_SETTING (page 128)
IOCTL_USBIO_SET_CONFIGURATION (page 74)

USBIO_CLASS_OR_VENDOR_REQUEST

The USBIO_CLASS_OR_VENDOR_REQUEST structure provides information used to generate a class or vendor specific device request.

Definition

```
typedef struct _USBIO_CLASS_OR_VENDOR_REQUEST{
  ULONG Flags;
  USBIO_REQUEST_TYPE Type;
  USBIO_REQUEST_RECIPIENT Recipient;
  UCHAR RequestTypeReservedBits;
  UCHAR Request;
  USHORT Value;
  USHORT Index;
} USBIO_CLASS_OR_VENDOR_REQUEST;
```

Members

Flags
This field contains zero or the following value.

USBIO_SHORT_TRANSFER_OK
If this flag is set then the USBIO driver does not return an error if a data packet received from the device is shorter than the maximum packet size of the endpoint. If this flag is not set then a short packet causes an error condition.

Type
Specifies the type of the device request. The values are defined by the enumeration type **USBIO_REQUEST_TYPE**.

Recipient
Specifies the recipient of the device request. The values are defined by the enumeration type **USBIO_REQUEST_RECIPIENT**.

RequestTypeReservedBits
Specifies the reserved bits of the `bmRequestType` field of the SETUP packet.

Request
Specifies the value of the `bRequest` field of the SETUP packet.

Value
Specifies the value of the `wValue` field of the SETUP packet.

Index
Specifies the value of the `wIndex` field of the SETUP packet.

Comments

The values defined by this structure are used to generate an eight byte SETUP packet for the default control endpoint (endpoint zero) of the device. The format of the SETUP packet is defined by the Universal Serial Bus Specification 1.1, Chapter 9. The meaning of the values is defined by the device.

This structure provides the input parameters for the **IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST** and the **IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST** operation.

See Also

IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST (page 78)
IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST (page 79)
USBIO_REQUEST_TYPE (page 155)
USBIO_REQUEST_RECIPIENT (page 154)

USBIO_DEVICE_PARAMETERS

The USBIO_DEVICE_PARAMETERS structure contains USBIO driver settings related to a device.

Definition

```
typedef struct _USBIO_DEVICE_PARAMETERS{
  ULONG Options;
  ULONG RequestTimeout;
} USBIO_DEVICE_PARAMETERS;
```

Members

`Options`
This field contains zero or any combination (bitwise OR) of the following values.

`USBIO_RESET_DEVICE_ON_CLOSE`
If this option is set then the USBIO driver generates a USB device reset after the last handle for a device has been closed by the application. If this option is active then the `USBIO_UNCONFIGURE_ON_CLOSE` flag will be ignored.

The default state of this option is defined by the registry parameter `ResetDeviceOnClose`. Refer to section 4.7 (page 47) for more information.

`USBIO_UNCONFIGURE_ON_CLOSE`
If this option is set then the USBIO driver sets the USB device to its unconfigured state after the last handle for the device has been closed by the application.

The default state of this option is defined by the registry parameter `UnconfigureOnClose`. Refer to section 4.7 (page 47) for more information.

`USBIO_ENABLE_REMOTE_WAKEUP`
If this option is set and the USB device supports the Remote Wakeup feature the USBIO driver will support Remote Wakeup for the operating system. That means the USB device is able to awake the system from a sleep state. The Remote Wakeup feature is defined by the USB 1.1 specification.

The Remote Wakeup feature requires that the device is opened by an application and that a USB configuration is set (device is configured).

The default state of this option is defined by the registry parameter `EnableRemoteWakeup`. Refer to section 4.7 (page 47) for more information.

`RequestTimeout`
Specifies the time-out interval, in milliseconds, to be used for synchronous operations. A value of zero means an infinite interval (time-out disabled).

The default time-out value is defined by the registry parameter `RequestTimeout`. Refer to section 4.7 (page 47) for more information.

Comments

This structure is intended to be used with the **IOCTL_USBIO_GET_DEVICE_PARAMETERS** and the **IOCTL_USBIO_SET_DEVICE_PARAMETERS** operations.

See Also

IOCTL_USBIO_GET_DEVICE_PARAMETERS (page 81)
IOCTL_USBIO_SET_DEVICE_PARAMETERS (page 82)

USBIO_INTERFACE_CONFIGURATION_INFO

The USBIO_INTERFACE_CONFIGURATION_INFO structure provides information about an interface.

Definition

```
typedef struct _USBIO_INTERFACE_CONFIGURATION_INFO{
  UCHAR InterfaceNumber;
  UCHAR AlternateSetting;
  UCHAR Class;
  UCHAR SubClass;
  UCHAR Protocol;
  UCHAR NumberOfPipes;
  UCHAR reserved1;
  UCHAR reserved2;
} USBIO_INTERFACE_CONFIGURATION_INFO;
```

Members

InterfaceNumber
Specifies the index of the interface as reported by the device in the configuration descriptor.

AlternateSetting
Specifies the index of the alternate setting as reported by the device in the configuration descriptor. The default alternate setting of an interface is zero.

Class
Specifies the class code as reported by the device in the configuration descriptor. The meaning of this value is defined by USB device class specifications.

SubClass
Specifies the subclass code as reported by the device in the configuration descriptor. The meaning of this value is defined by USB device class specifications.

Protocol
Specifies the protocol code as reported by the device in the configuration descriptor. The meaning of this value is defined by USB device class specifications.

NumberOfPipes
Specifies the number of pipes that belong to this interface and alternate setting.

reserved1
Reserved field, set to zero.

reserved2
Reserved field, set to zero.

Comments

This structure returns results of the **IOCTL_USBIO_GET_CONFIGURATION_INFO** operation. It is a substructure within the **USBIO_CONFIGURATION_INFO** structure.

See Also

USBIO_CONFIGURATION_INFO (page 138)
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 83)

USBIO_PIPE_CONFIGURATION_INFO

The USBIO_PIPE_CONFIGURATION_INFO structure provides information about a pipe.

Definition

```
typedef struct _USBIO_PIPE_CONFIGURATION_INFO{
  USBIO_PIPE_TYPE PipeType;
  ULONG MaximumTransferSize;
  USHORT MaximumPacketSize;
  UCHAR EndpointAddress;
  UCHAR Interval;
  UCHAR InterfaceNumber;
  UCHAR reserved1;
  UCHAR reserved2;
  UCHAR reserved3;
} USBIO_PIPE_CONFIGURATION_INFO;
```

Members

PipeType
Specifies the type of the pipe. The values are defined by the enumeration type **USBIO_PIPE_TYPE**.

MaximumTransferSize
Specifies the maximum size, in bytes, of data transfers the USB bus driver USBD supports on this pipe. This is the maximum size of buffers that can be used with read or write operations on this pipe.

MaximumPacketSize
Specifies the maximum packet size of USB data transfers the endpoint is capable of sending or receiving. This is also referred to as FIFO size. The `MaximumPacketSize` value is reported by the device in the corresponding endpoint descriptor. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

EndpointAddress
Specifies the address of the endpoint on the USB device as reported in the corresponding endpoint descriptor.

The endpoint address includes the direction flag at bit position 7 (MSb).

Bit 7 = 0: OUT endpoint

Bit 7 = 1: IN endpoint

Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

Interval
Specifies the interval, in milliseconds, for polling the endpoint for data as reported in the corresponding endpoint descriptor. This value is meaningful for interrupt endpoints only. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

`InterfaceNumber`
Specifies the index of the interface the pipe belongs to. The value is equal to the field `InterfaceNumber` of the corresponding **USBIO_INTERFACE_CONFIGURATION_INFO** structure.

`reserved1`
Reserved field, set to zero.

`reserved2`
Reserved field, set to zero.

`reserved3`
Reserved field, set to zero.

Comments

This structure returns results of the **IOCTL_USBIO_GET_CONFIGURATION_INFO** operation. It is a substructure within the **USBIO_CONFIGURATION_INFO** structure.

See Also

IOCTL_USBIO_GET_CONFIGURATION_INFO (page 83)
USBIO_CONFIGURATION_INFO (page 138)

USBIO_CONFIGURATION_INFO

The USBIO_CONFIGURATION_INFO structure provides information about the interfaces and pipes available in the current configuration.

Definition

```
typedef struct _USBIO_CONFIGURATION_INFO{
  ULONG NbOfInterfaces;
  ULONG NbOfPipes;
  USBIO_INTERFACE_CONFIGURATION_INFO
        InterfaceInfo[USBIO_MAX_INTERFACES];
  USBIO_PIPE_CONFIGURATION_INFO     PipeInfo[USBIO_MAX_PIPES];
} USBIO_CONFIGURATION_INFO;
```

Members

`NbOfInterfaces`
Specifies the number of interfaces active in the current configuration. This value corresponds to the number of valid entries in the `InterfaceInfo` array.

`NbOfPipes`
Specifies the number of pipes active in the current configuration. This value corresponds to the number of valid entries in the `PipeInfo` array.

`InterfaceInfo[USBIO_MAX_INTERFACES]`
An array of **USBIO_INTERFACE_CONFIGURATION_INFO** structures that describes the interfaces that are active in the current configuration. There are `NbOfInterfaces` valid entries in this array.

`PipeInfo[USBIO_MAX_PIPES]`
An array of **USBIO_PIPE_CONFIGURATION_INFO** structures that describes the pipes that are active in the current configuration. There are `NbOfPipes` valid entries in this array.

Comments

This structure returns the results of the **IOCTL_USBIO_GET_CONFIGURATION_INFO** operation.

Note that the data structure includes only those interfaces and pipes that are activated by the current configuration according to the configuration descriptor.

See Also

IOCTL_USBIO_GET_CONFIGURATION_INFO (page 83)
USBIO_INTERFACE_CONFIGURATION_INFO (page 134)
USBIO_PIPE_CONFIGURATION_INFO (page 136)

USBIO_FRAME_NUMBER

The USBIO_FRAME_NUMBER structure contains information about the USB frame counter value.

Definition

```
typedef struct _USBIO_FRAME_NUMBER{
  ULONG FrameNumber;
} USBIO_FRAME_NUMBER;
```

Member

FrameNumber
Specifies the current value of the frame counter that is maintained by the USB bus driver USBD. The frame number is an unsigned 32 bit value. The lower 11 bits of this value correspond to the frame number value in the Start Of Frame (SOF) token on the USB.

Comments

This structure returns the results of the **IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER** operation.

See Also

IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER (page 86)

USBIO_DEVICE_POWER

The USBIO_DEVICE_POWER structure contains information about the power state of the USB device.

Definition

```
typedef struct _USBIO_DEVICE_POWER{
  USBIO_DEVICE_POWER_STATE DevicePowerState;
} USBIO_DEVICE_POWER;
```

Member

DevicePowerState
Specifies the power state of the USB device. The values are defined by the **USBIO_DEVICE_POWER_STATE** enumeration type.

Comments

This structure is intended to be used with the **IOCTL_USBIO_GET_DEVICE_POWER_STATE** and the **IOCTL_USBIO_SET_DEVICE_POWER_STATE** operations.

See Also

USBIO_DEVICE_POWER_STATE (page 156)
IOCTL_USBIO_GET_DEVICE_POWER_STATE (page 88)
IOCTL_USBIO_SET_DEVICE_POWER_STATE (page 87)

USBIO_BIND_PIPE

The USBIO_BIND_PIPE structure provides information on the pipe to bind to.

Definition

```
typedef struct _USBIO_BIND_PIPE{
  UCHAR EndpointAddress;
} USBIO_BIND_PIPE;
```

Member

EndpointAddress

Specifies the address of the endpoint of the USB device that corresponds to the pipe. The endpoint address is specified as reported in the corresponding endpoint descriptor. It identifies the pipe unambiguously.

The endpoint address includes the direction flag at bit position 7 (MSb).

Bit 7 = 0: OUT endpoint

Bit 7 = 1: IN endpoint

Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

Comments

This structure provides the input parameters for the **IOCTL_USBIO_BIND_PIPE** operation.

See Also

IOCTL_USBIO_BIND_PIPE (page 96)

USBIO_PIPE_PARAMETERS

The USBIO_PIPE_PARAMETERS structure contains USBIO driver settings related to a pipe.

Definition

```
typedef struct _USBIO_PIPE_PARAMETERS{
  ULONG Flags;
} USBIO_PIPE_PARAMETERS;
```

Member

Flags
This field contains zero or the following value.

USBIO_SHORT_TRANSFER_OK
If this flag is set then the USBIO driver does not return an error during read operations from a Bulk or Interrupt pipe if a packet received from the device is shorter than the maximum packet size of the endpoint. If this flag is not set then a short packet causes an error condition.

Note that this option is only meaningful for Bulk or Interrupt IN pipes. It has an effect only for read operations from Bulk or Interrupt pipes. For Isochronous pipes the flags in the appropriate ISO data structures are used (see **USBIO_ISO_TRANSFER**).

The default state of the USBIO_SHORT_TRANSFER_OK flag is defined by the registry parameter `ShortTransferOk`. Refer to section 4.7 (page 47) for more information.

Comments

This structure is intended to be used with the **IOCTL_USBIO_GET_PIPE_PARAMETERS** and the **IOCTL_USBIO_SET_PIPE_PARAMETERS** operations.

See Also

IOCTL_USBIO_GET_PIPE_PARAMETERS (page 100)
IOCTL_USBIO_SET_PIPE_PARAMETERS (page 101)
USBIO_ISO_TRANSFER (page 149)

USBIO_SETUP_PIPE_STATISTICS

The USBIO_SETUP_PIPE_STATISTICS structure contains information used to configure the statistics maintained by the USBIO driver for a pipe.

Definition

```
typedef struct _USBIO_SETUP_PIPE_STATISTICS{
  ULONG AveragingInterval;
  UCHAR reserved1;
  UCHAR reserved2;
} USBIO_SETUP_PIPE_STATISTICS;
```

Members

AveragingInterval
Specifies the time interval, in milliseconds, that is used to calculate the average data rate of the pipe. A time averaging algorithm is used to continuously compute the mean value of the data transfer rate. The USBIO driver internally allocates memory to implement an averaging filter. There are 2048 bytes of memory required per second of the averaging interval. To limit the memory consumption the maximum supported value of `AveragingInterval` is 5000 milliseconds (5 seconds). If a longer interval is specified then the **IOCTL_USBIO_SETUP_PIPE_STATISTICS** request will fail with an error status of **USBIO_ERR_INVALID_PARAMETER**. It is recommended to use an averaging interval of 1000 milliseconds.

If `AveragingInterval` is set to zero then the average data rate computation is disabled. This is the default state. An application should only enable the average data rate computation if it is needed. This will save resources (kernel memory and CPU cycles).

See also **IOCTL_USBIO_QUERY_PIPE_STATISTICS** and **USBIO_PIPE_STATISTICS** for more information on pipe statistics.

reserved1
This member is reserved for future use. It has to be set to zero.

reserved2
This member is reserved for future use. It has to be set to zero.

Comments

This structure provides the input parameters for the **IOCTL_USBIO_SETUP_PIPE_STATISTICS** operation.

See Also

IOCTL_USBIO_SETUP_PIPE_STATISTICS (page 102)
IOCTL_USBIO_QUERY_PIPE_STATISTICS (page 104)
USBIO_PIPE_STATISTICS (page 146)

USBIO_QUERY_PIPE_STATISTICS

The USBIO_QUERY_PIPE_STATISTICS structure provides options that modify the behaviour of the **IOCTL_USBIO_QUERY_PIPE_STATISTICS** operation.

Definition

```
typedef struct _USBIO_QUERY_PIPE_STATISTICS{
  ULONG Flags;
} USBIO_QUERY_PIPE_STATISTICS;
```

Member

Flags
This field contains zero or any combination (bitwise OR) of the following values.

USBIO_QPS_FLAG_RESET_BYTES_TRANSFERRED
If this flag is specified then the BytesTransferred counter will be reset to zero after its current value has been captured. The BytesTransferred counter is an unsigned 64 bit integer. It counts the total number of bytes transferred on a pipe, modulo 2^64.

USBIO_QPS_FLAG_RESET_REQUESTS_SUCCEEDED
If this flag is specified then the RequestsSucceeded counter will be reset to zero after its current value has been captured. The RequestsSucceeded counter is an unsigned 32 bit integer. It counts the total number of read or write requests that have been completed successfully on a pipe, modulo 2^32.

USBIO_QPS_FLAG_RESET_REQUESTS_FAILED
If this flag is specified then the RequestsFailed counter will be reset to zero after its current value has been captured. The RequestsFailed counter is an unsigned 32 bit integer. It counts the total number of read or write requests that have been completed with an error status on a pipe, modulo 2^32.

USBIO_QPS_FLAG_RESET_ALL_COUNTERS
This value combines the three flags described above. If USBIO_QPS_FLAG_RESET_ALL_COUNTERS is specified then all three counters BytesTransferred, RequestsSucceeded, and RequestsFailed will be reset to zero after their current values have been captured.

Comments

This structure provides the input parameters for the **IOCTL_USBIO_QUERY_PIPE_STATISTICS** operation.

See also the description of the **USBIO_PIPE_STATISTICS** data structure for more information on pipe statistics.

See Also

IOCTL_USBIO_QUERY_PIPE_STATISTICS (page 104)
USBIO_PIPE_STATISTICS (page 146)

USBIO_PIPE_STATISTICS

The USBIO_PIPE_STATISTICS structure contains statistical data related to a pipe.

Definition

```
typedef struct _USBIO_PIPE_STATISTICS{
  ULONG ActualAveragingInterval;
  ULONG AverageRate;
  ULONG BytesTransferred_L;
  ULONG BytesTransferred_H;
  ULONG RequestsSucceeded;
  ULONG RequestsFailed;
  ULONG reserved1;
  ULONG reserved2;
} USBIO_PIPE_STATISTICS;
```

Members

ActualAveragingInterval
A time averaging algorithm is used to continuously compute the mean value of the data transfer rate. This field specifies the actual time interval, in milliseconds, that is used to calculate the average data rate returned in `AverageRate`. Normally, this value corresponds to the interval that has been configured by means of the **IOCTL_USBIO_SETUP_PIPE_STATISTICS** operation. However, if the capacity of the internal averaging filter is not sufficient for the interval set then `ActualAveragingInterval` can be less than the averaging interval that has been configured.

If `ActualAveragingInterval` is zero then the data rate computation is disabled. The `AverageRate` field of this structure is always set to zero in this case.

AverageRate
Specifies the current average data rate of the pipe, in bytes per second. The average data rate will be continuously calculated if the `ActualAveragingInterval` field of this structure is not null. If `ActualAveragingInterval` is null then the data rate computation is disabled and this field is always set to zero.

The computation of the average data rate has to be enabled and to be configured explicitly by an application. This has to be done by means of the **IOCTL_USBIO_SETUP_PIPE_STATISTICS** request.

BytesTransferred_L
Specifies the lower 32 bits of the current value of the BytesTransferred counter. The BytesTransferred counter is an unsigned 64 bit integer. It counts the total number of bytes transferred on a pipe, modulo 2^64.

BytesTransferred_H
Specifies the upper 32 bits of the current value of the BytesTransferred counter. The BytesTransferred counter is an unsigned 64 bit integer. It counts the total number of bytes

transferred on a pipe, modulo 2^64.

RequestsSucceeded

Specifies the current value of the RequestsSucceeded counter. The RequestsSucceeded counter is an unsigned 32 bit integer. It counts the total number of read or write requests that have been completed successfully on a pipe, modulo 2^32.

On a bulk or interrupt pipe the term request corresponds to a buffer that is submitted to perform a read or write operation. Thus, this counter will be incremented by one for each buffer that was successfully transferred.

On an isochronous pipe the term request corresponds to an isochronous data frame. Each buffer that is submitted to perform a read or write operation contains several isochronous data frames. This counter will be incremented by one for each isochronous data frame that was successfully transferred.

RequestsFailed

Specifies the current value of the RequestsFailed counter. The RequestsFailed counter is an unsigned 32 bit integer. It counts the total number of read or write requests that have been completed with an error status on a pipe, modulo 2^32.

On a bulk or interrupt pipe the term request corresponds to a buffer that is submitted to perform a read or write operation. Thus, this counter will be incremented by one for each buffer that is completed with an error status.

On an isochronous pipe the term request corresponds to an isochronous data frame. Each buffer that is submitted to perform a read or write operation contains several isochronous data frames. This counter will be incremented by one for each isochronous data frame that is completed with an error status.

reserved1

This member is reserved for future use.

reserved2

This member is reserved for future use.

Comments

This structure returns results of the **IOCTL_USBIO_QUERY_PIPE_STATISTICS** operation.

See Also

IOCTL_USBIO_QUERY_PIPE_STATISTICS (page 104)
IOCTL_USBIO_SETUP_PIPE_STATISTICS (page 102)
USBIO_QUERY_PIPE_STATISTICS (page 144)

USBIO_PIPE_CONTROL_TRANSFER

The USBIO_PIPE_CONTROL_TRANSFER structure provides information used to generate a specific control request.

Definition

```
typedef struct _USBIO_PIPE_CONTROL_TRANSFER{
  ULONG Flags;
  UCHAR SetupPacket[8];
} USBIO_PIPE_CONTROL_TRANSFER;
```

Members

Flags
This field contains zero or the following value.

USBIO_SHORT_TRANSFER_OK
If this flag is set then the USBIO driver does not return an error if a data packet received from the device is shorter than the maximum packet size of the endpoint. If this flag is not set then a short packet causes an error condition.

SetupPacket[8]
Specifies the SETUP packet to be issued to the device. The format of the eight byte SETUP packet is defined by the Universal Serial Bus Specification 1.1, Chapter 9.

Comments

The values defined by this structure are used to generate an eight byte SETUP packet for a control endpoint. However, it is not possible to generate a control transfer for the default endpoint zero.

This structure provides the input parameters for the **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN** and the **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT** operation.

See Also

IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN (page 106)
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT (page 107)

USBIO_ISO_TRANSFER

The USBIO_ISO_TRANSFER data structure provides information about an isochronous data transfer buffer.

Definition

```
typedef struct _USBIO_ISO_TRANSFER{
  ULONG NumberOfPackets;
  ULONG Flags;
  ULONG StartFrame;
  ULONG ErrorCount;
} USBIO_ISO_TRANSFER;
```

Members

`NumberOfPackets`
Specifies the number of packets to be sent to or to be received from the device. Each packet corresponds to a USB frame or a microframe respectively. The maximum number of packets allowed in a read or write operation is limited by the registry parameter `MaxIsoPackets`. Refer to section 4.7 (page 47) for more information.

`Flags`
This field contains zero or any combination (bitwise OR) of the following values.

`USBIO_SHORT_TRANSFER_OK`
If this flag is set then the USBIO driver does not return an error if a data packet received from the device is shorter than the maximum packet size of the endpoint. If this flag is not set then a short packet causes an error condition.

`USBIO_START_TRANSFER_ASAP`
If this flag is set then the transfer will be started as soon as possible and the `StartFrame` parameter is ignored. This flag has to be used if a continuous data stream shall be sent to the isochronous endpoint of the USB device.

`StartFrame`
Specifies the frame number or microframe number respectively at which the transfer is to be started. The value has to be within a system-defined range relative to the current frame. Normally, this range is set to 1024 frames.

If `USBIO_START_TRANSFER_ASAP` is not specified in `Flags` then `StartFrame` has to be initialized by the caller. The caller must specify the frame number at which the first packet of the data transfer is to be transmitted. An error occurs if the frame number is not in the valid range, relative to the current frame number.

If `USBIO_START_TRANSFER_ASAP` is specified in `Flags` then the `StartFrame` value specified by the user will be ignored. After the transfer has been started and the write request has been completed the `StartFrame` field contains the frame number assigned to the first packet of the transfer.

`ErrorCount`

After the isochronous read or write request has been completed by the USBIO driver this member contains the total number of errors occurred during the data transfer. In other words, `ErrorCount` specifies the number of frames that caused an error. This field can be used by an application to check if an isochronous read or write request has been completed successfully.

Comments

This data structure is a substructure within the **USBIO_ISO_TRANSFER_HEADER** structure. It is the fixed sized part of the header.

See also section 7.3.2 (page 110) for more information on isochronous data transfers.

See Also

USBIO_ISO_TRANSFER_HEADER (page 152)

USBIO_ISO_PACKET

The USBIO_ISO_PACKET structure defines the size and location of a single isochronous data packet within an isochronous data transfer buffer.

Definition

```
typedef struct _USBIO_ISO_PACKET{
  ULONG Offset;
  ULONG Length;
  ULONG Status;
} USBIO_ISO_PACKET;
```

Members

Offset
Specifies the offset, in bytes, of the isochronous packet, relative to the start of the data buffer. This parameter has to be specified by the caller for isochronous read and write operations.

Length
Specifies the size, in bytes, of the isochronous packet. This parameter has to be specified by the caller for write operations. On read operations this field is set by the USBIO driver when the read request is completed.

Status
After the isochronous read or write request is completed by the USBIO driver this field specifies the completion status of the isochronous packet.

Comments

An array of `USBIO_ISO_PACKET` structures is embedded within the **USBIO_ISO_TRANSFER_HEADER** structure. One `USBIO_ISO_PACKET` structure is required for each isochronous data packet to be transferred.

See also section 7.3.2 (page 110) for more information on isochronous data transfers.

See Also

USBIO_ISO_TRANSFER_HEADER (page 152)

USBIO_ISO_TRANSFER_HEADER

The USBIO_ISO_TRANSFER_HEADER structure defines the header that has to be placed at the beginning of an isochronous data transfer buffer.

Definition

```
typedef struct _USBIO_ISO_TRANSFER_HEADER{
  USBIO_ISO_TRANSFER IsoTransfer;
  USBIO_ISO_PACKET IsoPacket[1];
} USBIO_ISO_TRANSFER_HEADER;
```

Members

`IsoTransfer`
This is the fixed-size part of the header. See the description of the **USBIO_ISO_TRANSFER** data structure for more information.

`IsoPacket[1]`
This array of **USBIO_ISO_PACKET** structures has a variable length. Each element of the array corresponds to an isochronous data packet that is to be transferred either to or from the transfer buffer.

The number of valid elements in `IsoPacket` is specified by the `NumberOfPackets` member of `IsoTransfer`. See the description of the **USBIO_ISO_TRANSFER** data structure for more information. The maximum number of isochronous data packets per transfer buffer is defined by the registry parameter `MaxIsoPackets`. Refer to section 4.7 (page 47) for more information.

Comments

A data buffer that is passed to `ReadFile` or `WriteFile` on an isochronous pipe has to contain a valid **USBIO_ISO_TRANSFER_HEADER** structure at offset zero. After this header the buffer contains the isochronous data which is divided into packets. The `IsoPacket` array describes the location and the size of each single isochronous data packet. The isochronous data packets have to be placed into the transfer buffer in such a way that a contiguous data area will be created. In other words, no gaps between isochronous data packets are allowed.

See also section 7.3.2 (page 110) for more information on isochronous data transfers.

See Also

USBIO_ISO_TRANSFER (page 149)
USBIO_ISO_PACKET (page 151)

7.6 Enumeration Types

USBIO_PIPE_TYPE

The USBIO_PIPE_TYPE enumeration type contains values that identify the type of a USB pipe or a USB endpoint, respectively.

Definition

```
typedef enum _USBIO_PIPE_TYPE{
  PipeTypeControl  = 0,
  PipeTypeIsochronous,
  PipeTypeBulk,
  PipeTypeInterrupt
} USBIO_PIPE_TYPE;
```

Comments

The meaning of the values is defined by the Universal Serial Bus Specification 1.1, Chapter 9.

See Also

USBIO_PIPE_CONFIGURATION_INFO (page 136)

USBIO_REQUEST_RECIPIENT

The USBIO_REQUEST_RECIPIENT enumeration type contains values that identify the recipient of a USB device request.

Definition

```
typedef enum _USBIO_REQUEST_RECIPIENT{
  RecipientDevice  = 0,
  RecipientInterface,
  RecipientEndpoint,
  RecipientOther
} USBIO_REQUEST_RECIPIENT;
```

Comments

The meaning of the values is defined by the Universal Serial Bus Specification 1.1, Chapter 9.

See Also

USBIO_DESCRIPTOR_REQUEST (page 120)
USBIO_FEATURE_REQUEST (page 122)
USBIO_STATUS_REQUEST (page 123)
USBIO_CLASS_OR_VENDOR_REQUEST (page 130)

USBIO_REQUEST_TYPE

The USBIO_REQUEST_TYPE enumeration type contains values that identify the type of a USB device request.

Definition

```
typedef enum _USBIO_REQUEST_TYPE{
  RequestTypeClass  = 1,
  RequestTypeVendor
} USBIO_REQUEST_TYPE;
```

Comments

The meaning of the values is defined by the Universal Serial Bus Specification 1.1, Chapter 9.

The enumeration does not contain the Standard request type defined by the USB specification. This is because the USB bus driver USBD supports Class and Vendor requests only at its programming interface. Standard requests are generated internally by the USBD.

See Also

USBIO_CLASS_OR_VENDOR_REQUEST (page 130)

USBIO_DEVICE_POWER_STATE

The USBIO_DEVICE_POWER_STATE enumeration type contains values that identify the power state of a device.

Definition

```
typedef enum _USBIO_DEVICE_POWER_STATE{
  DevicePowerStateD0  = 0,
  DevicePowerStateD1,
  DevicePowerStateD2,
  DevicePowerStateD3
} USBIO_DEVICE_POWER_STATE;
```

Entries

DevicePowerStateD0
Device fully on, normal operation.

DevicePowerStateD1
Suspend.

DevicePowerStateD2
Suspend.

DevicePowerStateD3
Device off.

Comments

The meaning of the values is defined by the Power Management specification.

See Also

USBIO_DEVICE_POWER (page 140)

7.7 Error Codes

USBIO_ERR_SUCCESS (0x00000000L)

The operation has been successfully completed.

USBIO_ERR_CRC (0xE0000001L)

A CRC error has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_BTSTUFF (0xE0000002L)

A bit stuffing error has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_DATA_TOGGLE_MISMATCH (0xE0000003L)

A DATA toggle mismatch (DATA0/DATA1 tokens) has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_STALL_PID (0xE0000004L)

A STALL PID has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_DEV_NOT_RESPONDING (0xE0000005L)

The USB device is not responding. This error is reported by the USB host controller driver.

USBIO_ERR_PID_CHECK_FAILURE (0xE0000006L)

A PID check has failed. This error is reported by the USB host controller driver.

USBIO_ERR_UNEXPECTED_PID (0xE0000007L)

An unexpected PID has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_DATA_OVERRUN (0xE0000008L)

A data overrun error has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_DATA_UNDERRUN (0xE0000009L)

A data underrun error has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_RESERVED1 (0xE000000AL)

This error code is reserved by the USB host controller driver.

USBIO_ERR_RESERVED2 (0xE000000BL)

This error code is reserved by the USB host controller driver.

USBIO_ERR_BUFFER_OVERRUN (0xE000000CL)

A buffer overrun has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_BUFFER_UNDERRUN (0xE000000DL)

A buffer underrun has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_NOT_ACCESSED (0xE000000FL)

A data buffer was not accessed. An isochronous data buffer was scheduled too late. The specified frame number does not match the actual frame number. This error is reported by the USB host controller driver.

USBIO_ERR_FIFO (0xE0000010L)

A FIFO error has been detected. This error is reported by the USB host controller driver. The PCI bus latency was too long.

USBIO_ERR_XACT_ERROR (0xE0000011L)

A XACT error has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_BABBLE_DETECTED (0xE0000012L)

A device is babbling. This error is reported by the USB host controller driver. The data transfer phase exceeds the USB frame length.

USBIO_ERR_DATA_BUFFER_ERROR (0xE0000013L)

A data buffer error has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_ENDPOINT_HALTED (0xE0000030L)

The endpoint has been halted by the USB bus driver USBD. This error is reported by the USB bus driver USBD. A pipe will be halted by USBD when a data transmission error (CRC, bit stuff, DATA toggle) occurs. In order to re-enable a halted pipe a **IOCTL_USBIO_RESET_PIPE** request has to be issued on that pipe. See the description of **IOCTL_USBIO_RESET_PIPE** for more information.

USBIO_ERR_NO_MEMORY (0xE0000100L)

A memory allocation attempt has failed. This error is reported by the USB bus driver USBD.

USBIO_ERR_INVALID_URB_FUNCTION (0xE0000200L)

An invalid URB function code has been passed. This error is reported by the USB bus driver USBD.

USBIO_ERR_INVALID_PARAMETER (0xE0000300L)

An invalid parameter has been passed. This error is reported by the USB bus driver USBD.

USBIO_ERR_ERROR_BUSY (0xE0000400L)

There are data transfer requests pending for the device. This error is reported by the USB bus driver USBD.

USBIO_ERR_REQUEST_FAILED (0xE0000500L)

A request has failed. This error is reported by the USB bus driver USBD.

USBIO_ERR_INVALID_PIPE_HANDLE (0xE0000600L)

An invalid pipe handle has been passed. This error is reported by the USB bus driver USBD.

USBIO_ERR_NO_BANDWIDTH (0xE0000700L)

There is not enough bandwidth available. This error is reported by the USB bus driver USBD.

USBIO_ERR_INTERNAL_HC_ERROR (0xE0000800L)

An internal host controller error has been detected. This error is reported by the USB bus driver USBD.

USBIO_ERR_ERROR_SHORT_TRANSFER (0xE0000900L)

A short transfer has been detected. This error is reported by the USB bus driver USBD. If the pipe is not configured accordingly a short packet sent by the device causes this error. Support for short packets has to be enabled explicitly. See **IOCTL_USBIO_SET_PIPE_PARAMETERS** and **IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST** for more information.

USBIO_ERR_BAD_START_FRAME (0xE0000A00L)

An incorrect start frame has been specified. This error is reported by the USB bus driver USBD.

USBIO_ERR_ISOCH_REQUEST_FAILED (0xE0000B00L)

An isochronous request has failed. This error is reported by the USB bus driver USBD.

USBIO_ERR_FRAME_CONTROL_OWNED (0xE0000C00L)

The USB frame control is currently owned. This error is reported by the USB bus driver USBD.

USBIO_ERR_FRAME_CONTROL_NOT_OWNED (0xE0000D00L)

The USB frame control is currently not owned. This error is reported by the USB bus driver USBD.

USBIO_ERR_NOT_SUPPORTED (0xE0000E00L)

The operation is not supported. This error is reported by the USB bus driver USBD.

USBIO_ERR_INVALID_CONFIGURATION_DESCRIPTOR (0xE0000F00L)

An invalid configuration descriptor was reported by the device. This error is reported by the USB bus driver USBD.

USBIO_ERR_INSUFFICIENT_RESOURCES (0xE8001000L)

There are not enough resources available to complete the operation. This error is reported by the USB bus driver USBD.

USBIO_ERR_SET_CONFIG_FAILED (0xE0002000L)

The set configuration request has failed. This error is reported by the USB bus driver USBD.

USBIO_ERR_USBD_BUFFER_TOO_SMALL (0xE0003000L)

The buffer is too small. This error is reported by the USB bus driver USBD.

USBIO_ERR_USBD_INTERFACE_NOT_FOUND (0xE0004000L)

The interface was not found. This error is reported by the USB bus driver USBD.

USBIO_ERR_INVALID_PIPE_FLAGS (0xE0005000L)

Invalid pipe flags have been specified. This error is reported by the USB bus driver USBD.

USBIO_ERR_USBD_TIMEOUT (0xE0006000L)

The operation has been timed out. This error is reported by the USB bus driver USBD.

USBIO_ERR_DEVICE_GONE (0xE0007000L)

The USB device is gone. This error is reported by the USB bus driver USBD.

USBIO_ERR_STATUS_NOT_MAPPED (0xE0008000L)

This error is reported by the USB bus driver USBD.

USBIO_ERR_CANCELED (0xE0010000L)

The operation has been canceled. This error is reported by the USB bus driver USBD. If the data transfer requests pending on a pipe are aborted by means of **IOCTL_USBIO_ABORT_PIPE** or `CancelIo` then the operations will be completed with this error code.

USBIO_ERR_ISO_NOT_ACCESSED_BY_HW (0xE0020000L)

The isochronous data buffer was not accessed by the USB host controller. An isochronous data buffer was scheduled too late. The specified frame number does not match the actual frame number. This error is reported by the USB bus driver, USBD.

USBIO_ERR_ISO_TD_ERROR (0xE0030000L)

The USB host controller reported an error in a transfer descriptor. This error is reported by the USB bus driver USBD.

USBIO_ERR_ISO_NA_LATE_USBPORT (0xE0040000L)

An isochronous data packet was submitted in time but failed to reach the USB host controller in time. This error is reported by the USB bus driver USBD.

USBIO_ERR_ISO_NOT_ACCESSED_LATE (0xE0050000L)

An isochronous data packet was submitted too late. This error is reported by the USB bus driver USBD.

USBIO_ERR_FAILED (0xE0001000L)

The operation has failed. This error is reported by the USBIO driver.

USBIO_ERR_INVALID_INBUFFER (0xE0001001L)

An invalid input buffer has been passed to an IOCTL operation. This error is reported by the USBIO driver. Make sure the input buffer matches the type and size requirements specified for the IOCTL operation.

USBIO_ERR_INVALID_OUTBUFFER (0xE0001002L)

An invalid output buffer has been passed to an IOCTL operation. This error is reported by the USBIO driver. Make sure the output buffer matches the type and size requirements specified for the IOCTL operation.

USBIO_ERR_OUT_OF_MEMORY (0xE0001003L)

There is not enough system memory available to complete the operation. This error is reported by the USBIO driver.

USBIO_ERR_PENDING_REQUESTS (0xE0001004L)

There are read or write requests pending. This error is reported by the USBIO driver.

USBIO_ERR_ALREADY_CONFIGURED (0xE0001005L)

The USB device is already configured. This error is reported by the USBIO driver.

USBIO_ERR_NOT_CONFIGURED (0xE0001006L)

The USB device is not configured. This error is reported by the USBIO driver.

USBIO_ERR_OPEN_PIPES (0xE0001007L)

There are open pipes. This error is reported by the USBIO driver.

USBIO_ERR_ALREADY_BOUND (0xE0001008L)

Either the handle is already bound to a pipe or the specified pipe is already bound to another handle. This error is reported by the USBIO driver. See **IOCTL_USBIO_BIND_PIPE** for more information.

USBIO_ERR_NOT_BOUND (0xE0001009L)

The handle is not bound to a pipe. This error is reported by the USBIO driver. The operation that has been failed with this error code is related to a pipe. Therefore, the handle has to be bound to a pipe before the operation can be executed. See **IOCTL_USBIO_BIND_PIPE** for more information.

USBIO_ERR_DEVICE_NOT_PRESENT (0xE000100AL)

The USB device has been removed from the system. This error is reported by the USBIO driver. An application should close all handles for the device. After it receives a Plug&Play notification it should perform a re-enumeration of devices.

USBIO_ERR_CONTROL_NOT_SUPPORTED (0xE000100BL)

The specified control code is not supported. This error is reported by the USBIO driver.

USBIO_ERR_TIMEOUT (0xE000100CL)

The operation has been timed out. This error is reported by the USBIO driver.

USBIO_ERR_INVALID_RECIPIENT (0xE000100DL)

An invalid recipient has been specified. This error is reported by the USBIO driver.

USBIO_ERR_INVALID_TYPE (0xE000100EL)

Either an invalid request type has been specified or the operation is not supported by that pipe type. This error is reported by the USBIO driver.

USBIO_ERR_INVALID_IOCTL (0xE000100FL)

An invalid IOCTL code has been specified. This error is reported by the USBIO driver.

USBIO_ERR_INVALID_DIRECTION (0xE0001010L)

The direction of the data transfer request is not supported by that pipe. This error is reported by the USBIO driver. On IN pipes read requests are supported only. On OUT pipes write requests are supported only.

USBIO_ERR_TOO_MUCH_ISO_PACKETS (0xE0001011L)

The number of isochronous data packets specified in an isochronous read or write request exceeds the maximum number of packets supported by the USBIO driver. This error is reported by the USBIO driver. Note that the maximum number of packets allowed per isochronous data buffer can be adjusted by means of the registry parameter `MaxIsoPackets`. Refer to section 4.7 (page 47) for more information.

USBIO_ERR_POOL_EMPTY (0xE0001012L)

The memory resources are exhausted. This error is reported by the USBIO driver.

USBIO_ERR_PIPE_NOT_FOUND (0xE0001013L)

The specified pipe was not found in the current configuration. This error is reported by the USBIO driver. Note that only endpoints that are included in the current configuration can be used to transfer data.

USBIO_ERR_INVALID_ISO_PACKET (0xE0001014L)

An invalid isochronous data packet has been specified. This error is reported by the USBIO driver. An isochronous data buffer contains an isochronous data packet with invalid `Offset` and/or `Length` parameters. See **USBIO_ISO_PACKET** for more information.

USBIO_ERR_OUT_OF_ADDRESS_SPACE (0xE0001015L)

There are not enough system resources to complete the operation. This error is reported by the USBIO driver.

USBIO_ERR_INTERFACE_NOT_FOUND (0xE0001016L)

The specified interface was not found in the current configuration or in the configuration descriptor. This error is reported by the USBIO driver. Note that only interfaces that are included in the current configuration can be used.

USBIO_ERR_INVALID_DEVICE_STATE (0xE0001017L)

The operation cannot be executed while the USB device is in the current state. This error is reported by the USBIO driver. It is not allowed to submit requests to the device while it is in a power down state.

USBIO_ERR_INVALID_PARAM (0xE0001018L)

An invalid parameter has been specified with an IOCTL operation. This error is reported by the USBIO driver.

USBIO_ERR_DEMO_EXPIRED (0xE0001019L)

The evaluation interval of the USBIO DEMO version has expired. This error is reported by the USBIO driver. The USBIO DEMO version is limited in runtime. After the DEMO evaluation period has expired every operation will be completed with this error code. After the system is rebooted the USBIO DEMO driver can be used for another evaluation interval.

USBIO_ERR_INVALID_POWER_STATE (0xE000101AL)

An invalid power state has been specified. This error is reported by the USBIO driver. Note that it is not allowed to switch from one power down state to another. The device has to be set to D0 before it can be set to another power down state.

USBIO_ERR_POWER_DOWN (0xE000101BL)

The device has entered a power down state. This error is reported by the USBIO driver. When the USB device leaves power state D0 and enters a power down state then all pending read and write requests will be canceled and completed with this error status. If an application detects this error status it can re-submit the read or write requests immediately. The requests will be queued by the USBIO driver internally.

USBIO_ERR_VERSION_MISMATCH (0xE000101CL)

The API version reported by the USBIO driver does not match the expected version. This error is reported by the USBIO C++ class library USBIOLIB.
See **IOCTL_USBIO_GET_DRIVER_INFO** for more information on USBIO version numbers.

USBIO_ERR_SET_CONFIGURATION_FAILED (0xE000101DL)

The set configuration operation has failed. This error is reported by the USBIO driver.

USBIO_ERR_ADDITIONAL_EVENT_SIGNALLED (0xE000101EL)

An additional event object that has been passed to a function, was signalled.

USBIO_ERR_INVALID_PROCESS (0xE000101FL)

This operation is not allowed for this process.

USBIO_ERR_DEVICE_ACQUIRED (0xE0001020L)

The device is acquired by a different process for exclusive usage.

USBIO_ERR_DEVICE_OPENED (0xE0001020L)

The device is opened by a different process.

USBIO_ERR_VID_RESTRICTION (0xE0001080L)

ATTENTION: The LIGHT version is no longer provided. The error code is documented only for compatibility with older versions.

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The LIGHT version does not support the Vendor ID reported by the USB device.

USBIO_ERR_ISO_RESTRICTION (0xE0001081L)

ATTENTION: The LIGHT version is no longer provided. The error code is documented only for compatibility with older versions.

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The LIGHT version does not support isochronous transfers.

USBIO_ERR_BULK_RESTRICTION (0xE0001082L)

ATTENTION: The LIGHT version is no longer provided. The error code is documented only for compatibility with older versions.

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The LIGHT version does not support bulk transfers.

USBIO_ERR_EP0_RESTRICTION (0xE0001083L)

ATTENTION: The LIGHT version is no longer provided. The error code is documented only for compatibility with older versions.

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The LIGHT version does not support class or vendor specific SETUP requests or the data transfer length exceeds the limit.

USBIO_ERR_PIPE_RESTRICTION (0xE0001084L)

ATTENTION: The LIGHT version is no longer provided. The error code is documented only for compatibility with older versions.

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The number of endpoints active in the current configuration exceeds the limit enforced by the LIGHT version.

USBIO_ERR_PIPE_SIZE_RESTRICTION (0xE0001085L)

ATTENTION: The LIGHT version is no longer provided. The error code is documented only for compatibility with older versions.

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The FIFO size of an endpoint of the current configuration exceeds the limit enforced by the LIGHT version.

USBIO_ERR_CONTROL_RESTRICTION (0xE0001086L)

ATTENTION: The LIGHT version is no longer provided. The error code is documented only for compatibility with older versions.

The operation has failed due to a functional restriction of the USBIO LIGHT version. This error is reported by the USBIO driver.

USBIO_ERR_INTERRUPT_RESTRICTION (0xE0001087L)

ATTENTION: The LIGHT version is no longer provided. The error code is documented only for compatibility with older versions.

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The LIGHT version does not support interrupt transfers.

USBIO_ERR_DEVICE_NOT_FOUND (0xE0001100L)

The specified device object does not exist. This error is reported by the USBIO C++ class library USBIOLIB. The USB device is not connected to the system or it has been removed by the user.

USBIO_ERR_DEVICE_NOT_OPEN (0xE0001102L)

No device object was opened. There is no valid handle to execute the operation. This error is reported by the USBIO C++ class library USBIOLIB.

USBIO_ERR_NO_SUCH_DEVICE_INSTANCE (0xE0001104L)

The enumeration of the specified devices has failed. There are no devices of the specified type available. This error is reported by the USBIO C++ class library USBIOLIB.

USBIO_ERR_INVALID_FUNCTION_PARAM (0xE0001105L)

An invalid parameter has been passed to a function. This error is reported by the USBIO C++ class library USBIOLIB.

USBIO_ERR_LOAD_SETUP_API_FAILED (0xE0001106L)

The library *setupapi.dll* could not be loaded. This error is reported by the USBIO C++ class library USBIOLIB. The Setup API that is exported by the system-provided *setupapi.dll* is part of the Win32 API. It is available in Windows 98 and later systems.

USBIO_ERR_DEVICE_ALREADY_OPENED (0xE0001107L)

The handle in this class instance was opened by an earlier call to the function open. Do not call the method open twice or close the handle before it is opened again.

USBIO_ERR_INVALID_DESCRIPTOR (0xE0001108L)

The device returned an invalid descriptor after a GetDescriptor request.

USBIO_ERR_NOT_SUPPORTED_UNDER_CE (0xE0001109L)

This operation or feature is not supported under Windows CE.

8 USBIO Class Library

8.1 Overview

The USBIO Class Library (USBIOLIB) contains classes which provide wrapper functions for all of the features supported by the USBIO programming interface. Using these classes in an application is more convenient than using the USBIO interface directly. The classes are designed to be capable of being extended. In order to meet the requirements of a particular application, new classes may be derived from the existing ones. The class library is provided fully in source code.

The following figure shows the classes included in the USBIOLIB and their relations.

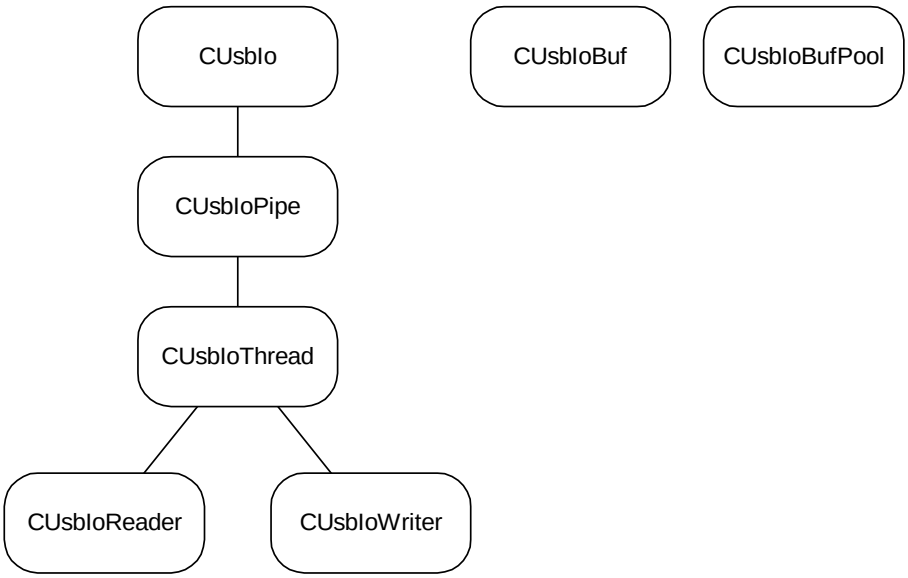

Figure 7: USBIO Class Library

8.1.1 CUsbIo Class

The class `CUsbIo` implements the basic interface to the USBIO device driver. It includes all functions that are related to a USBIO device object. Thus, by using an instance of the `CUsbIo` class all operations which do not require a pipe context can be performed.

The class also supports device enumeration and an `Open()` function that is used to connect an instance of the class to a USBIO device object. The handle that represents the connection is stored inside the class instance. It is used for all subsequent requests to the device.

For each device-related operation the USBIO driver supports, a member function exists in the `CUsbIo` class. The function takes the parameters that are required for the operation and returns the status that is reported by the USBIO driver.

8.1.2 CUsbIoPipe Class

The class `CUsbIoPipe` extends the `CUsbIo` class by functions that are related to a USBIO pipe object. An instance of the `CUsbIoPipe` class is associated directly with a USBIO pipe object.

In order to establish the connection to the pipe, the class provides a `Bind()` function. After a `CUsbIoPipe` instance is bound, pipe-related functions can be performed by using member functions of the class.

For each pipe-related operation that the USBIO driver supports a member function exists in the `CUsbIoPipe` class. The function takes the parameters that are required for the operation and returns the status that is reported by the USBIO driver.

The `CUsbIoPipe` class supports an asynchronous communication model for data transfers to or from the pipe. The `Read()` or `Write()` function is used to submit a data buffer to the USBIO driver. The function returns immediately indicating success if the buffer was sent to the driver successfully. There is no blocking within the `Read()` or `Write()` function. Therefore, it is possible to send multiple buffers to the pipe. The buffers are processed sequentially in the same order as they are submitted. The `WaitForCompletion()` member function is used to wait until the data transfer to or from a particular buffer is finished. This function blocks the calling thread until the USBIO driver has completed the I/O operation with the buffer.

In order to use a data buffer with the `Read()`, `Write()`, and `WaitForCompletion()` functions of the `CUsbIoPipe` class the buffer has to be described by a `CUsbIoBuf` object. The `CUsbIoBuf` helper class stores context information while the read or write operation is pending.

8.1.3 CUsbIoThread Class

The `CUsbIoThread` class provides basic functions needed to implement a worker thread that performs input or output operations on a pipe. It includes functions that are used to start and stop the worker thread.

The class does not implement the thread's main routine. This has to be done in a derived class. Thus, `CUsbIoThread` is an universal base class that simplifies the implementation of a worker thread that performs I/O operations on a pipe.

Note:
The worker thread created by `CUsbIoThread` is a native system thread. That means it cannot be used to call MFC (Microsoft Foundation Classes) functions. It is necessary to use `PostMessage`, `SendMessage` or some other communication mechanism to switch over to MFC-aware threads.

8.1.4 CUsbIoReader Class

The class `CUsbIoReader` extends the `CUsbIoThread` class by a specific worker thread routine that continuously sends read requests to the pipe. The thread's main routine gets buffers from an internal buffer pool and submits them to the pipe using the `Read()` function of the `CUsbIoPipe` class. After all buffers are submitted the routine waits for the first pending buffer to complete.

If a buffer is completed by the USBIO driver the virtual member function `ProcessData` is called with this buffer. Within this function the data received from the pipe should be processed. The `ProcessData` function has to be implemented by a class that is derived from `CUsbIoReader`. After that, the buffer is put back to the pool and the main loop is started from the beginning.

8.1.5 CUsbIoWriter Class

The class `CUsbIoWriter` extends the `CUsbIoThread` class by a specific worker thread routine that continuously sends write requests to the pipe. The thread's main routine gets a buffer from an internal buffer pool and calls the virtual member function `ProcessBuffer` to fill the buffer with data. After that, the buffer is sent to the pipe using the `Write()` function of the `CUsbIoPipe` class. After all buffers are submitted the routine waits for the first pending buffer to complete. If a buffer is completed by the USBIO driver the buffer is put back to the pool and the main loop is started from the beginning.

8.1.6 CUsbIoBuf Class

The helper class `CUsbIoBuf` is used as a descriptor for buffers that are processed by the class `CUsbIoPipe` and derived classes. One instance of the `CUsbIoBuf` class has to be created for each buffer. The `CUsbIoBuf` object stores context and status information that is needed to process the buffer asynchronously.

The `CUsbIoBuf` class contains a link element (Next pointer). This may be used to build a chain of linked buffer objects to hold them in a list. This way, the management of buffers can be simplified.

8.1.7 CUsbIoBufPool Class

The class `CUsbIoBufPool` is used to manage a pool of free buffers. It provides functions used to allocate an initial number of buffers, to get a buffer from the pool, and to put a buffer back to the pool.

8.1.8 CPnPNotificator Class

The class `CPnPNotificator` can be used to get Plug&Play notifications in a DLL or in an application that does not have a Window handle. To use the class derive a handler class from `CPnPNotifyHandler` and overwrite the function HandlePnPMessage(). Pass a pointer to an instance of this class to the function call `CPnPNotificator::Initialize()`.

If the application has a Window handle the API functions `RegisterDeviceNotification()` and `UnregisterDeviceNotification` should be used. See usbioappdlg.cpp and SDK for details.

8.2 Class Library Reference

CUsbIo class

This class implements the interface to the USBIO device driver. It contains only general device-related functions that can be executed without a pipe context. Pipe specific functions are implemented by the **CUsbIoPipe** class.

Member Functions

CUsbIo::CUsbIo

Standard constructor of the CUsbIo class.

Definition

```
CUsbIo();
```

See Also

CUsbIo::~CUsbIo (page 174)
CUsbIo::Open (page 177)

CUsbIo::~CUsbIo

Destructor of the CUsbIo class.

Definition

```
virtual
~CUsbIo();
```

See Also

CUsbIo::CUsbIo (page 174)
CUsbIo::Close (page 180)

CUsbIo::CreateDeviceList

Creates an internal device list.

Definition

```
static HDEVINFO
CreateDeviceList(
  const GUID* InterfaceGuid
  );
```

Parameter

`InterfaceGuid`
This is the predefined interface GUID of the USBIO device driver or a user defined GUID which must be inserted in the USBIO.INF file.

Return Value

Returns a handle to the device list if successful, or NULL otherwise.

Comments

The function creates a windows-internal device list that contains all matching interfaces. The device interface is identified by **`InterfaceGuid`**. A handle for the list is returned in case of success, or NULL is returned in case of error. The device list can be iterated by means of **CUsbIo::Open**.

The device list returned must be freed by a call to **CUsbIo::DestroyDeviceList**.

Note that CreateDeviceList is declared static. It can be used independently of class instances.

See Also

CUsbIo::DestroyDeviceList (page 176)
CUsbIo::Open (page 177)

CUsbIo::DestroyDeviceList

Destroy the internal device list.

Definition

```
static void
DestroyDeviceList(
  HDEVINFO DeviceList
  );
```

Parameter

DeviceList
A handle to a device list returned by **CUsbIo::CreateDeviceList**.

Comments

Use this function to destroy a device list that was generated by a call to **CUsbIo::CreateDeviceList**.

Note that DestroyDeviceList is declared static. It can be used independently of class instances.

See Also

CUsbIo::CreateDeviceList (page 175)

CUsbIo::Open

Open an USB device.

Definition

```
DWORD
Open(
  int DeviceNumber,
  HDEVINFO DeviceList,
  const GUID* InterfaceGuid,
  bool CheckApiVersion = true
  );
```

Parameters

`DeviceNumber`
Specifies the index number of the USB Device. The index is zero-based. Note that the association between this number and the USB device can change with each call to **CUsbIo::CreateDeviceList**.

`DeviceList`
A handle to the internal device list which was returned by the function **CUsbIo::CreateDeviceList** or NULL. For more information see below.

`InterfaceGuid`
Points to a caller-provided variable of type GUID. The specified GUID is the predefined interface GUID of the USBIO device driver, or a user-defined GUID which has to be defined in the USBIO.INF file. This parameter will be ignored if DeviceList is set to NULL. For more information, see below.

`CheckApiVersion = true`
If this parameter is true the open function fails with the error code **`USBIO_ERR_VERSION_MISMATCH`** if the API version of the used driver is less than the expected or the major version of the API is different.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

There are two options:

(A) `DeviceList != NULL`
The device list provided in DeviceList must have been built using **CUsbIo::CreateDeviceList**. The GUID that identifies the device interface must be provided in InterfaceGuid. **`DeviceNumber`** is used to iterate through the device list. It should start with zero and should

be incremented after each call to **Open**. If no more instances of the interface are available then the status code **USBIO_ERR_NO_SUCH_DEVICE_INSTANCE** is returned.
Note: This is the recommended way of implementing a device enumeration.

(B) DeviceList == NULL
The parameter InterfaceGuid will be ignored. DeviceNumber will be used to build an old-style device name that starts with the string defined by **USBIO_DEVICE_NAME** (see also the comments on **USBIO_DEVICE_NAME** in UsbIo.h).
Note: This mode should be used only if compatibility to earlier versions of USBIO is required. It will work only if the creation of static device names is enabled in the USBIO.INF file. It is not recommended to use this mode.

See Also

CUsbIo::CreateDeviceList (page 175)
CUsbIo::DestroyDeviceList (page 176)
CUsbIo::Close (page 180)
CUsbIo::IsOpen (page 185)
CUsbIo::IsCheckedBuild (page 186)
CUsbIo::IsDemoVersion (page 187)
CUsbIo::IsLightVersion (page 188)

CUsbIo::OpenPath

Open an USB device with a known device path.

Definition

```
DWORD
OpenPath(
  const TCHAR* DevicePath,
  bool CheckApiVersion = true
  );
```

Parameters

DevicePath
Specifies the device path that was returned either by **CUsbIo::GetDevicePathName**, by the function **CPnPNotifyHandler::HandlePnPMessage** or by the **WM_DEVICECHANGE** message.

CheckApiVersion = true
If this parameter is true the open function fails with the error code **USBIO_ERR_VERSION_MISMATCH** if the API version of the used driver is less than the expected or the major version of the API is different.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The function **CUsbIo::Open** uses an index to the device list to determine the device instance. The index of a device can be changed if a new device list is created. The **DevicePath** is valid as long as the device is connected to the PC. If the device reports a unique serial number as a USB descriptor the **DevicePath** describes exactly one device. This function is useful if a known device should be opened without performing a new enumeration.

See Also

CUsbIo::Open (page 177)
CUsbIo::GetDevicePathName (page 183)
CUsbIo::IsOpen (page 185)
CPnPNotifyHandler::HandlePnPMessage (page 291)

CUsbIo::Close

Close the USB device.

Definition

```
void
Close();
```

Comments

This function can be called if the device is not open. It does nothing in this case.

Any thread associated with the class instance should have been stopped before this function is called. See **CUsbIoThread::ShutdownThread**.

See Also

CUsbIo::CreateDeviceList (page 175)
CUsbIo::DestroyDeviceList (page 176)
CUsbIo::Open (page 177)
CUsbIoThread::ShutdownThread (page 258)

CUsbIo::GetDeviceInstanceDetails

Get detailed information on an USB device instance.

Definition

```
DWORD
GetDeviceInstanceDetails(
  int DeviceNumber,
  HDEVINFO DeviceList,
  const GUID* InterfaceGuid
  );
```

Parameters

DeviceNumber
Specifies the index of the USB Device. The index is zero-based. Note that the association between this number and the USB device can change with each call to **CUsbIo::CreateDeviceList**.

DeviceList
A handle to the internal device list which was returned by the function **CUsbIo::CreateDeviceList**.

InterfaceGuid
Points to a caller-provided variable of type GUID. The specified GUID is the predefined interface GUID of the USBIO device driver, or a user-defined GUID which has to be defined in the USBIO.INF file.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function retrieves detailed information on a device instance that has been enumerated by means of **CUsbIo::CreateDeviceList**. That information includes the device path name that has to be passed to CreateFile in order to open the device instance. The device path name is returned by **CUsbIo::GetDevicePathName**.

This function is used internally by the implementation of **CUsbIo::Open**. Normally, it is not called directly by an application.

See Also

CUsbIo::CreateDeviceList (page 175)
CUsbIo::DestroyDeviceList (page 176)
CUsbIo::Open (page 177)

CUsbIo::GetDevicePathName (page 183)

CUsbIo::GetDevicePathName

Returns the path name that is required to open the device instance.

Definition

```
const TCHAR*
GetDevicePathName();
```

Return Value

Returns a pointer to the path name associated with this device instance, or NULL. The returned pointer is only temporarily valid and should not be stored for later use. It becomes invalid if the device is closed. If no device is opened, the return value is NULL.

The return value is always NULL if the device was opened using case (A) described in the comments of the **CUsbIo::Open** function.

Comments

This function retrieves the device path name of the device instance. The path name is available after a device enumeration has been performed by a call to **CUsbIo::CreateDeviceList** and **CUsbIo::GetDeviceInstanceDetails** or **CUsbIo::Open** has been called.

This function is used internally by the implementation of **CUsbIo::Open**. Normally, it is not called directly by an application.

See Also

CUsbIo::CreateDeviceList (page 175)
CUsbIo::DestroyDeviceList (page 176)
CUsbIo::GetDeviceInstanceDetails (page 181)
CUsbIo::Open (page 177)
CUsbIo::Close (page 180)

CUsbIo::GetParentID

Returns a string that is unique for the parent hub.

Definition

```
const TCHAR*
GetParentID();
```

Return Value

Returns a pointer to a unique string to identify the parent hub. Returns NULL if something goes wrong.

Comments

This function retrieves parent hub identifier. The ID is available after a device enumeration has been performed by a call to **CUsbIo::CreateDeviceList** and **CUsbIo::GetDeviceInstanceDetails** or **CUsbIo::Open** has been called.

Two devices instances that return the same parent ID are connected to the same USB hub.

See Also

CUsbIo::CreateDeviceList (page 175)
CUsbIo::DestroyDeviceList (page 176)
CUsbIo::GetDeviceInstanceDetails (page 181)
CUsbIo::Open (page 177)
CUsbIo::Close (page 180)

CUsbIo::IsOpen

Returns TRUE if the class instance is attached to a device.

Definition

```
BOOL
IsOpen();
```

Return Value

Returns TRUE when a device was opened by a successful call to **Open**. Returns FALSE if no device is opened.

See Also

CUsbIo::Open (page 177)
CUsbIo::Close (page 180)

CUsbIo::IsCheckedBuild

Returns TRUE if a checked build (debug version) of the USBIO driver was detected.

Definition

```
BOOL
IsCheckedBuild();
```

Return Value

Returns TRUE if the checked build of the USBIO driver is running, FALSE otherwise.

Comments

The device must have been opened before this function is called.

See Also

CUsbIo::Open (page 177)
CUsbIo::IsDemoVersion (page 187)
CUsbIo::IsLightVersion (page 188)

CUsbIo::IsDemoVersion

Returns TRUE if the Demo version of the USBIO driver was detected.

Definition

```
BOOL
IsDemoVersion();
```

Return Value

Returns TRUE if the Demo version of the USBIO driver is running, FALSE otherwise.

Comments

The device must have been opened before this function is called.

See Also

CUsbIo::Open (page 177)
CUsbIo::IsCheckedBuild (page 186)
CUsbIo::IsLightVersion (page 188)

CUsbIo::IsLightVersion

Returns TRUE if the LIGHT version of the USBIO driver was detected.

Definition

```
BOOL
IsLightVersion();
```

Return Value

Returns TRUE if the LIGHT version of the USBIO driver is running, FALSE otherwise.

Comments

ATTENTION: The LIGHT version is no longer provided.

The device must have been opened before this function is called.

See Also

CUsbIo::Open (page 177)
CUsbIo::IsCheckedBuild (page 186)
CUsbIo::IsDemoVersion (page 187)

CUsbIo::IsOperatingAtHighSpeed

Returns TRUE if the USB 2.0 device is operating at high speed (480 Mbit/s).

Definition

```
BOOL
IsOperatingAtHighSpeed();
```

Return Value

Returns TRUE if the USB device is operating at high speed, FALSE otherwise.

Comments

If this function returns TRUE then the USB device operates in high speed mode. The USB 2.0 device should be connected to a hub port that is high speed capable.

Note that this function does not indicate whether a device is capable of high speed operation, but rather whether it is in fact operating at high speed.

This function calls **CUsbIo::GetDeviceInfo** to get the requested information.

The device must have been opened before this function is called.

See Also

CUsbIo::Open (page 177)
CUsbIo::GetDeviceInfo (page 193)

CUsbIo::GetDriverInfo

Get information on the USBIO device driver.

Definition

```
DWORD
GetDriverInfo(
  USBIO_DRIVER_INFO* DriverInfo
  );
```

Parameter

DriverInfo
Pointer to a caller-provided variable. The structure returns the API version, the driver version, and the build number.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_DRIVER_INFO** operation. See also the description of **IOCTL_USBIO_GET_DRIVER_INFO** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::Close (page 180)
USBIO_DRIVER_INFO (page 118)
IOCTL_USBIO_GET_DRIVER_INFO (page 91)

CUsbIo::AcquireDevice

Acquire the device for exclusive use.

Definition

```
DWORD
AcquireDevice();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_ACQUIRE_DEVICE** operation. See also the description of **IOCTL_USBIO_ACQUIRE_DEVICE** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::Close (page 180)
CUsbIo::ReleaseDevice (page 192)

CUsbIo::ReleaseDevice

Acquire the device for exclusive use.

Definition

```
DWORD
ReleaseDevice();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_RELEASE_DEVICE** operation. See also the description of **IOCTL_USBIO_RELEASE_DEVICE** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::Close (page 180)
CUsbIo::AcquireDevice (page 191)

CUsbIo::GetDeviceInfo

Get information about the USB device.

Definition

```
DWORD
GetDeviceInfo(
  USBIO_DEVICE_INFO* DeviceInfo
  );
```

Parameter

DeviceInfo
Pointer to a caller-provided variable. The structure returns information on the USB device. This includes a flag that indicates whether the device operates in high speed mode or not.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function can be used to detect if the device operates in high speed mode.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_DEVICE_INFO** operation. See also the description of **IOCTL_USBIO_GET_DEVICE_INFO** for further information.

See Also

CUsbIo::Open (page 177)
USBIO_DEVICE_INFO (page 117)
IOCTL_USBIO_GET_DEVICE_INFO (page 90)

CUsbIo::GetBandwidthInfo

Get information on the current USB bandwidth consumption.

Definition

```
DWORD
GetBandwidthInfo(
  USBIO_BANDWIDTH_INFO* BandwidthInfo
  );
```

Parameter

BandwidthInfo
Pointer to a caller-provided variable. The structure returns information on the bandwidth that is available on the USB.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The function enables an application to check the bandwidth that is available on the USB. Depending on this information an application can select an appropriate device configuration, if desired.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_BANDWIDTH_INFO** operation. See also the description of **IOCTL_USBIO_GET_BANDWIDTH_INFO** for further information.

See Also

CUsbIo::Open (page 177)
USBIO_BANDWIDTH_INFO (page 116)
IOCTL_USBIO_GET_BANDWIDTH_INFO (page 89)

CUsbIo::GetDescriptor

Get a descriptor from the device.

Definition

```
DWORD
GetDescriptor(
  void* Buffer,
  DWORD& ByteCount,
  USBIO_REQUEST_RECIPIENT Recipient,
  UCHAR DescriptorType,
  UCHAR DescriptorIndex = 0,
  USHORT LanguageId = 0
  );
```

Parameters

`Buffer`
Pointer to a caller-provided buffer. The buffer receives the requested descriptor.

`ByteCount`
When the function is called **`ByteCount`** specifies the size, in bytes, of the buffer pointed to by **`Buffer`**. After the function successfully returned **`ByteCount`** contains the number of valid bytes returned in the buffer.

`Recipient`
Specifies the recipient of the request. Possible values are enumerated by **USBIO_REQUEST_RECIPIENT**.

`DescriptorType`
The type of the descriptor to request. Values are defined by the USB specification, chapter 9.

`DescriptorIndex`
The index of the descriptor to request. Set to zero if an index is not used for the descriptor type, e.g. for a device descriptor.

`LanguageId`
The language ID of the descriptor to request. Used for string descriptors only. Set to zero if not used.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

If the size of the provided buffer is less than the total size of the requested descriptor then only the specified number of bytes from the beginning of the descriptor is returned.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_DESCRIPTOR** operation. See also the description of **IOCTL_USBIO_GET_DESCRIPTOR** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::GetDeviceDescriptor (page 197)
CUsbIo::GetConfigurationDescriptor (page 198)
CUsbIo::GetStringDescriptor (page 200)
CUsbIo::SetDescriptor (page 202)
USBIO_REQUEST_RECIPIENT (page 154)
IOCTL_USBIO_GET_DESCRIPTOR (page 66)

CUsbIo::GetDeviceDescriptor

Get the device descriptor from the device.

Definition

```
DWORD
GetDeviceDescriptor(
  USB_DEVICE_DESCRIPTOR* Desc
  );
```

Parameter

Desc
Pointer to a caller-provided variable that receives the requested descriptor.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

GetDeviceDescriptor calls **CUsbIo::GetDescriptor** to retrieve the descriptor. Thus, for detailed information see also **CUsbIo::GetDescriptor**.

The device must have been opened before this function is called.

See Also

CUsbIo::Open (page 177)
CUsbIo::GetDescriptor (page 195)
CUsbIo::GetConfigurationDescriptor (page 198)
CUsbIo::GetStringDescriptor (page 200)

CUsbIo::GetConfigurationDescriptor

Get a configuration descriptor from the device.

Definition

```
DWORD
GetConfigurationDescriptor(
  USB_CONFIGURATION_DESCRIPTOR* Desc,
  DWORD& ByteCount,
  UCHAR Index = 0
  );
```

Parameters

Desc
Pointer to a caller-provided buffer that receives the requested descriptor. Note that the size of the configuration descriptor depends on the USB device. See also the comments below.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Desc**. After the function successfully returned **ByteCount** contains the number of valid bytes returned in the buffer.

Index
The index of the descriptor to request.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

GetConfigurationDescriptor calls **CUsbIo::GetDescriptor** to retrieve the descriptor. Thus, for detailed information see also **CUsbIo::GetDescriptor**.

If the total size of the configuration descriptor is not known it can be retrieved in a two step process. With a first call to this function the fixed part of the descriptor which is defined by **USB_CONFIGURATION_DESCRIPTOR** is retrieved. The total size of the descriptor is indicated by the *wTotalLength* field of the structure. In a second step a buffer of the required size can be allocated and the complete descriptor can be retrieved with another call to this function.

The device must have been opened before this function is called.

See Also

CUsbIo::Open (page 177)

CUsbIo::GetDescriptor (page 195)
CUsbIo::GetDeviceDescriptor (page 197)
CUsbIo::GetStringDescriptor (page 200)

CUsbIo::GetStringDescriptor

Get a string descriptor from the device.

Definition

```
DWORD
GetStringDescriptor(
  USB_STRING_DESCRIPTOR* Desc,
  DWORD& ByteCount,
  UCHAR Index = 0,
  USHORT LanguageId = 0
  );
```

Parameters

Desc
Pointer to a caller-provided buffer that receives the requested descriptor. Note that according to the USB specification the maximum size of a string descriptor is 256 bytes.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Desc**. After the function successfully returned **ByteCount** contains the number of valid bytes returned in the buffer.

Index
The index of the descriptor to request. Set to 0 to retrieve a list of supported language IDs. See also the comments below.

LanguageId
The language ID of the string descriptor to request.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

GetStringDescriptor calls **CUsbIo::GetDescriptor** to retrieve the descriptor. Thus, for detailed information see also **CUsbIo::GetDescriptor**.

If this function is called with **Index** set to 0 the device returns a list of language IDs it supports. An application can select the correct language ID and use it in subsequent calls to this function.

The device must have been opened before this function is called.

See Also

CUsbIo::Open (page 177)
CUsbIo::GetDescriptor (page 195)
CUsbIo::GetDeviceDescriptor (page 197)
CUsbIo::GetConfigurationDescriptor (page 198)

CUsbIo::SetDescriptor

Set a descriptor of the device.

Definition

```
DWORD
SetDescriptor(
  const void* Buffer,
  DWORD& ByteCount,
  USBIO_REQUEST_RECIPIENT Recipient,
  UCHAR DescriptorType,
  UCHAR DescriptorIndex = 0,
  USHORT LanguageId = 0
  );
```

Parameters

`Buffer`
Pointer to a caller-provided buffer. The buffer contains the descriptor data to be set.

`ByteCount`
When the function is called **`ByteCount`** specifies the size, in bytes, of the descriptor pointed to by **`Buffer`**. After the function successfully returned **`ByteCount`** contains the number of bytes transferred.

`Recipient`
Specifies the recipient of the request. Possible values are enumerated by **USBIO_REQUEST_RECIPIENT**.

`DescriptorType`
The type of the descriptor to set. Values are defined by the USB specification, chapter 9.

`DescriptorIndex`
The index of the descriptor to set.

`LanguageId`
The language ID of the descriptor to set. Used for string descriptors only. Set to zero if not used.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

Note that most devices do not support the set descriptor request.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_SET_DESCRIPTOR** operation. See also the description of **IOCTL_USBIO_SET_DESCRIPTOR** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::GetDescriptor (page 195)
CUsbIo::GetDeviceDescriptor (page 197)
CUsbIo::GetConfigurationDescriptor (page 198)
CUsbIo::GetStringDescriptor (page 200)
USBIO_REQUEST_RECIPIENT (page 154)
IOCTL_USBIO_SET_DESCRIPTOR (page 67)

CUsbIo::SetFeature

Send a set feature request to the USB device.

Definition

```
DWORD
SetFeature(
  USBIO_REQUEST_RECIPIENT Recipient,
  USHORT FeatureSelector,
  USHORT Index = 0
  );
```

Parameters

Recipient
Specifies the recipient of the request. Possible values are enumerated by **USBIO_REQUEST_RECIPIENT**.

FeatureSelector
Specifies the feature selector value for the request. The values are defined by the recipient. Refer to the USB specification, chapter 9 for more information.

Index
Specifies the index value for the set feature request. The values are defined by the device. Refer to the USB specification, chapter 9 for more information.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_SET_FEATURE** operation. See also the description of **IOCTL_USBIO_SET_FEATURE** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::ClearFeature (page 205)
CUsbIo::GetStatus (page 206)
USBIO_REQUEST_RECIPIENT (page 154)
IOCTL_USBIO_SET_FEATURE (page 68)

CUsbIo::ClearFeature

Send a clear feature request to the USB device.

Definition

```
DWORD
ClearFeature(
  USBIO_REQUEST_RECIPIENT Recipient,
  USHORT FeatureSelector,
  USHORT Index  =0
  );
```

Parameters

Recipient
Specifies the recipient of the request. Possible values are enumerated by **USBIO_REQUEST_RECIPIENT**.

FeatureSelector
Specifies the feature selector value for the request. The values are defined by the recipient. Refer to the USB specification, chapter 9 for more information.

Index
Specifies the index value for the clear feature request. The values are defined by the device. Refer to the USB specification, chapter 9 for more information.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_CLEAR_FEATURE** operation. See also the description of **IOCTL_USBIO_CLEAR_FEATURE** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::SetFeature (page 204)
CUsbIo::GetStatus (page 206)
USBIO_REQUEST_RECIPIENT (page 154)
IOCTL_USBIO_CLEAR_FEATURE (page 69)

CUsbIo::GetStatus

Send a get status request to the USB device.

Definition

```
DWORD
GetStatus(
  USHORT& StatusValue,
  USBIO_REQUEST_RECIPIENT Recipient,
  USHORT Index = 0
  );
```

Parameters

StatusValue
If the function call is successful this variable returns the 16-bit value that is returned by the recipient in response to the get status request. The interpretation of the value is specific to the recipient. Refer to the USB specification, chapter 9 for more information.

Recipient
Specifies the recipient of the request. Possible values are enumerated by **USBIO_REQUEST_RECIPIENT**.

Index
Specifies the index value for the get status request. The values are defined by the device. Refer to the USB specification, chapter 9 for more information.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_STATUS** operation. See also the description of **IOCTL_USBIO_GET_STATUS** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::SetFeature (page 204)
CUsbIo::ClearFeature (page 205)
USBIO_REQUEST_RECIPIENT (page 154)
IOCTL_USBIO_GET_STATUS (page 70)

CUsbIo::ClassOrVendorInRequest

Sends a class or vendor specific request with a data phase in device to host (IN) direction.

Definition

```
DWORD
ClassOrVendorInRequest(
  void* Buffer,
  DWORD& ByteCount,
  const USBIO_CLASS_OR_VENDOR_REQUEST* Request
  );
```

Parameters

Buffer
Pointer to a caller-provided buffer. The buffer receives the data transferred in the IN data phase.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Buffer**. After the function successfully returned **ByteCount** contains the number of valid bytes returned in the buffer.

Request
Pointer to a caller-provided variable that defines the request to be generated.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST** operation. See also the description of **IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::ClassOrVendorOutRequest (page 208)
USBIO_CLASS_OR_VENDOR_REQUEST (page 130)
IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST (page 78)

CUsbIo::ClassOrVendorOutRequest

Sends a class or vendor specific request with a data phase in host to device (OUT) direction.

Definition

```
DWORD
ClassOrVendorOutRequest(
  const void* Buffer,
  DWORD& ByteCount,
  const USBIO_CLASS_OR_VENDOR_REQUEST* Request
  );
```

Parameters

Buffer
Pointer to a caller-provided buffer that contains the data to be transferred in the OUT data phase.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Buffer**. This is the number of bytes to be transferred in the data phase. After the function successfully returned **ByteCount** contains the number of bytes transferred.

Request
Pointer to a caller-provided variable that defines the request to be generated.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST** operation. See also the description of **IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::ClassOrVendorInRequest (page 207)
USBIO_CLASS_OR_VENDOR_REQUEST (page 130)
IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST (page 79)

CUsbIo::SetConfiguration

Set the device to the configured state.

Definition

```
DWORD
SetConfiguration(
  const USBIO_SET_CONFIGURATION* Conf
  );
```

Parameter

Conf
Points to a caller-provided structure that defines the configuration to be set.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device has to be configured before any data transfer to or from its endpoints can take place. Only those endpoints that are included in the configuration will be activated and can be subsequently used for data transfers.

If the device provides more than one interface all interfaces must be configured in one call to this function.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_SET_CONFIGURATION** operation. See also the description of **IOCTL_USBIO_SET_CONFIGURATION** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::UnconfigureDevice (page 210)
CUsbIo::GetConfiguration (page 211)
CUsbIo::SetInterface (page 213)
USBIO_SET_CONFIGURATION (page 129)
IOCTL_USBIO_SET_CONFIGURATION (page 74)

CUsbIo::UnconfigureDevice

Set the device to the unconfigured state.

Definition

```
DWORD
UnconfigureDevice();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_UNCONFIGURE_DEVICE** operation. See also the description of **IOCTL_USBIO_UNCONFIGURE_DEVICE** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::SetConfiguration (page 209)
CUsbIo::GetConfiguration (page 211)
IOCTL_USBIO_UNCONFIGURE_DEVICE (page 76)

CUsbIo::GetConfiguration

This function returns the current configuration value.

Definition

```
DWORD
GetConfiguration(
  UCHAR& ConfigurationValue
  );
```

Parameter

ConfigurationValue
If the function call is successful this variable returns the current configuration value. A value of 0 means the USB device is not configured.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The configuration value returned by this function corresponds to the *bConfiguration* field of the active configuration descriptor. Note that the configuration value does not necessarily correspond to the index of the configuration descriptor.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_CONFIGURATION** operation. See also the description of **IOCTL_USBIO_GET_CONFIGURATION** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::SetConfiguration (page 209)
IOCTL_USBIO_GET_CONFIGURATION (page 71)

CUsbIo::GetConfigurationInfo

Get information on the interfaces and endpoints available in the current configuration.

Definition

```
DWORD
GetConfigurationInfo(
  USBIO_CONFIGURATION_INFO* Info
  );
```

Parameter

Info
Points to a caller-provided variable that receives the configuration information.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The information returned is retrieved from the USBIO driver's internal data base. This function does not cause any action on the USB.

The device must have been opened and configured before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_CONFIGURATION_INFO** operation. See also the description of **IOCTL_USBIO_GET_CONFIGURATION_INFO** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::SetConfiguration (page 209)
USBIO_CONFIGURATION_INFO (page 138)
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 83)

CUsbIo::SetInterface

This function changes the alternate setting of an interface.

Definition

```
DWORD
SetInterface(
  const USBIO_INTERFACE_SETTING* Setting
  );
```

Parameter

Setting
Points to a caller-provided structure that specifies the interface and the alternate settings to be set.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened and configured before this function is called.

This function is a wrapper for the **IOCTL_USBIO_SET_INTERFACE** operation. See also the description of **IOCTL_USBIO_SET_INTERFACE** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::SetConfiguration (page 209)
CUsbIo::GetInterface (page 214)
USBIO_INTERFACE_SETTING (page 128)
IOCTL_USBIO_SET_INTERFACE (page 77)

CUsbIo::GetInterface

This function returns the active alternate setting of an interface.

Definition

```
DWORD
GetInterface(
  UCHAR& AlternateSetting,
  USHORT Interface = 0
  );
```

Parameters

AlternateSetting
If the function call is successful this variable returns the current alternate setting of the interface.

Interface
Specifies the index of the interface to be queried.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened and configured before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_INTERFACE** operation. See also the description of **IOCTL_USBIO_GET_INTERFACE** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::SetConfiguration (page 209)
CUsbIo::SetInterface (page 213)
IOCTL_USBIO_GET_INTERFACE (page 72)

CUsbIo::StoreConfigurationDescriptor

Store a configuration descriptor in the USBIO device driver.
This function is obsolete (see below).

Definition

```
DWORD
StoreConfigurationDescriptor(
  const USB_CONFIGURATION_DESCRIPTOR* Desc
  );
```

Parameter

Desc
Pointer to the configuration descriptor to be stored.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

Store a configuration descriptor in the USBIO device driver and use it for the next set configuration request. This allows to work around problems with certain devices.

The device must have been opened before this function is called.

Note: This function is obsolete and should not be used. It was introduced in earlier versions of USBIO to work around problems caused by the Windows USB driver stack. The stack was not able to handle some types of isochronous endpoint descriptors correctly. These problems have now been fixed and therefore the StoreConfigurationDescriptor work-around is obsolete.

This function is a wrapper for the **IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR** operation. See also the description of **IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::SetConfiguration (page 209)
IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR (page 73)

CUsbIo::GetDeviceParameters

Query device-related parameters from the USBIO device driver.

Definition

```
DWORD
GetDeviceParameters(
  USBIO_DEVICE_PARAMETERS* DevParam
  );
```

Parameter

DevParam
Points to a caller-provided variable that receives the current parameter settings.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_DEVICE_PARAMETERS** operation. See also the description of **IOCTL_USBIO_GET_DEVICE_PARAMETERS** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::SetDeviceParameters (page 217)
USBIO_DEVICE_PARAMETERS (page 132)
IOCTL_USBIO_GET_DEVICE_PARAMETERS (page 81)

CUsbIo::SetDeviceParameters

Set device-related parameters in the USBIO device driver.

Definition

```
DWORD
SetDeviceParameters(
  const USBIO_DEVICE_PARAMETERS* DevParam
  );
```

Parameter

DevParam
Points to a caller-provided variable that specifies the parameters to be set.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

Default device parameters are stored in the registry during USBIO driver installation. The default value can be changed in the INF file or in the registry. Device parameters set by means of this function are valid until the device is removed from the PC or the PC is booted. A modification during run-time does not change the default in the registry.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_SET_DEVICE_PARAMETERS** operation. See also the description of **IOCTL_USBIO_SET_DEVICE_PARAMETERS** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::GetDeviceParameters (page 216)
USBIO_DEVICE_PARAMETERS (page 132)
IOCTL_USBIO_SET_DEVICE_PARAMETERS (page 82)

CUsbIo::ResetDevice

Force an USB reset.

Definition

```
DWORD
ResetDevice();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function causes an USB reset to be issued on the hub port the device is connected to. This will abort all pending read and write requests and unbind all pipes. The device will be set to the unconfigured state.

Note: The device must be in the configured state when this function is called. ResetDevice does not work when the system-provided USB multi-interface driver is used (see also *problems.txt* in the USBIO package).

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_RESET_DEVICE** operation. See also the description of **IOCTL_USBIO_RESET_DEVICE** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::SetConfiguration (page 209)
CUsbIo::UnconfigureDevice (page 210)
CUsbIo::CyclePort (page 219)
CUsbIoPipe::Bind (page 230)
CUsbIoPipe::Unbind (page 233)
CUsbIoPipe::AbortPipe (page 243)
IOCTL_USBIO_RESET_DEVICE (page 84)

CUsbIo::CyclePort

Simulates a device disconnect/connect cycle.

Definition

```
DWORD
CyclePort();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function causes a device disconnect/connect cycle and an unload/load cycle for the USBIO device driver as well.

Note: CyclePort does not work on multi-interface devices (see also *problems.txt* in the USBIO package).

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_CYCLE_PORT** operation. See also the description of **IOCTL_USBIO_CYCLE_PORT** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::ResetDevice (page 218)
IOCTL_USBIO_CYCLE_PORT (page 92)

CUsbIo::GetCurrentFrameNumber

Get the current USB frame number from the host controller.

Definition

```
DWORD
GetCurrentFrameNumber(
  DWORD& FrameNumber
  );
```

Parameter

FrameNumber
If the function call is successful this variable returns the current frame number.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The returned frame number is a 32-bit value. The 11 least significant bits correspond to the frame number in the USB frame token.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER** operation. See also the description of **IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER** for further information.

See Also

CUsbIo::Open (page 177)
IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER (page 86)

CUsbIo::GetDevicePowerState

Returns the current device power state.

Definition

```
DWORD
GetDevicePowerState(
  USBIO_DEVICE_POWER_STATE& DevicePowerState
  );
```

Parameter

DevicePowerState
If the function call is successful this variable returns the current device power state.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_DEVICE_POWER_STATE** operation. See also the description of **IOCTL_USBIO_GET_DEVICE_POWER_STATE** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::SetDevicePowerState (page 222)
USBIO_DEVICE_POWER_STATE (page 156)
IOCTL_USBIO_GET_DEVICE_POWER_STATE (page 88)

CUsbIo::SetDevicePowerState

Set the device power state.

Definition

```
DWORD
SetDevicePowerState(
  USBIO_DEVICE_POWER_STATE DevicePowerState
  );
```

Parameter

DevicePowerState
Specifies the device power state to be set.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

In order to set the device to suspend it must be in the configured state.

When the device is set to suspend all pending read and write requests will be returned by the USBIO driver with an error status of **USBIO_ERR_POWER_DOWN**. An application can ignore this error status and submit the requests to the driver again.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_SET_DEVICE_POWER_STATE** operation. See also the description of **IOCTL_USBIO_SET_DEVICE_POWER_STATE** for further information.

See Also

CUsbIo::Open (page 177)
CUsbIo::GetDevicePowerState (page 221)
USBIO_DEVICE_POWER_STATE (page 156)
IOCTL_USBIO_SET_DEVICE_POWER_STATE (page 87)

CUsbIo::CancelIo

Cancels outstanding I/O requests issued by the calling thread.

Definition

```
BOOL
CancelIo();
```

Return Value

If the function succeeds the return value is TRUE, FALSE otherwise.

Comments

CancelIo cancels all outstanding I/O requests that were issued by the calling thread on the class instance. Requests issued on the instance by other threads are not cancelled.

This function is a wrapper for the Win32 function *CancelIo*. For a description of this function refer to the Win32 Platform SDK documentation.

CUsbIo::IoctlSync

Call a driver I/O control function and wait for its completion.

Definition

```
DWORD
IoctlSync(
  DWORD IoctlCode,
  const void* InBuffer,
  DWORD InBufferSize,
  void* OutBuffer,
  DWORD OutBufferSize,
  DWORD* BytesReturned
  );
```

Parameters

IoctlCode
The IOCTL code that identifies the driver function.

InBuffer
Pointer to the input buffer. The input buffer contains information to be passed to the driver. Set to NULL if no input buffer is needed.

InBufferSize
Size, in bytes, of the input buffer. Set to 0 if no input buffer is needed.

OutBuffer
Pointer to the output buffer. The output buffer receives information returned from the driver. Set to NULL if no output buffer is needed.

OutBufferSize
Size, in bytes, of the output buffer. Set to 0 if no output buffer is needed.

BytesReturned
Points to a caller-provided variable that, on a successful call, will be set to the number of bytes returned in the output buffer. Set to NULL if this information is not needed.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

IoctlSync is a generic support function that can be used to submit any IOCTL request to the USBIO device driver.

This function is used internally to handle the asynchronous USBIO API.

CUsbIo::GetDevInfoData

Returns a reference to the **SP_DEVINFO_DATA** structure of the USBIO device node.

Definition

```
const SP_DEVINFO_DATA&
GetDevInfoData();
```

Return Value

The function returns the **SP_DEVINFO_DATA** reference.

Comments

The return value is valid when the instance of the class was opened **CUsbIo::Open**. The returned reference can be used to get additional information about the device node or the parent device node.

To find out whether some devices are connected to the same USB hub the Win32 API functions **CM_Get_Parent** and **CM_Get_Device_ID** can be used. The required **DEVINST** is part of the data structure **SP_DEVINFO_DATA::DevInst**. On composite devices the function **CM_Get_Parent** must be called two times.

See Also

CUsbIo::Open (page 177)

CUsbIo::ErrorText

Translate an USBIO error code to a description string.

Definition

```
static TCHAR*
ErrorText(
  TCHAR* StringBuffer,
  DWORD MaxCharCount,
  DWORD ErrorCode
  );
```

Parameters

`StringBuffer`
A caller-provided string buffer that receives the text. The function returns a null-terminated ASCII or UNICODE string.

`MaxCharCount`
Specifies the maximum number of characters that can be stored in the buffer pointed to by **`StringBuffer`**.

`ErrorCode`
The error code to be translated.

Return Value

The function returns the **`StringBuffer`** pointer.

Comments

This function supports private USBIO error codes only. These codes start with a prefix of 0xE. The function cannot be used to translate general Windows error codes.

Note that ErrorText is declared static. It can be used independently of class instances.

Data Members

`HANDLE` **`FileHandle`**
This protected member contains the handle for the USBIO device object. The value is NULL if no open handle exists.

`OVERLAPPED` **`Overlapped`**
This protected member provides an OVERLAPPED data structure that is required to perform asynchronous (overlapped) I/O operations by means of the Win32 function **`DeviceIoControl`**. The Overlapped member is used by **CUsbIo::IoctlSync**.

`CRITICAL_SECTION` **`CritSect`**
This protected member provides a Win32 Critical Section object that is used to synchronize IOCTL operations on **`FileHandle`**. Synchronization is required because there is only one OVERLAPPED data structure per CUsbIo object, see also **CUsbIo::Overlapped**.

The CUsbIo object is thread-save. It can be accessed by multiple threads simultaneously.

`BOOL` **`CheckedBuildDetected`**
This protected member provides a flag. It is set to TRUE if a checked (debug) build of the USBIO driver has been detected, FALSE otherwise. The flag is initialized by the **CUsbIo::Open** function. See also **CUsbIo::GetDriverInfo** for more information.

The state of this flag can be queried by means of the function **CUsbIo::IsCheckedBuild**.

`BOOL` **`DemoVersionDetected`**
This protected member provides a flag. It is set to TRUE if a DEMO version of the USBIO driver has been detected, FALSE otherwise. The flag is initialized by the **CUsbIo::Open** function. See also **CUsbIo::GetDriverInfo** for more information.

The state of this flag can be queried by means of the function **CUsbIo::IsDemoVersion**.

`BOOL` **`LightVersionDetected`**
ATTENTION: The LIGHT version is no longer provided.

This protected member provides a flag. It is set to TRUE if a LIGHT version of the USBIO driver has been detected, FALSE otherwise. The flag is initialized by the **CUsbIo::Open** function. See also **CUsbIo::GetDriverInfo** for more information.

The state of this flag can be queried by means of the function **CUsbIo::IsLightVersion**.

`SP_DEVICE_INTERFACE_DETAIL_DATA*` **`mDevDetail`**
This private member contains information on the device instance opened by the CUsbIo object. It is used by the CUsbIo implementation internally.

`SP_DEVINFO_DATA` **`SP_DEVINFO_DATA`**
This private member contains information on the device info data opened by the CUsbIo object. It is used by the CUsbIo implementation internally.

`TCHAR*` **`mParentIdStr`**
This private member is either NULL or points to a string buffer that contains the parent ID. It is used by the CUsbIo implementation internally.

`static CSetupApiDll` **`smSetupApi`**

This protected member is used to manage the library *setupapi.dll*. The **CSetupApiDll** class provides functions to load the system-provided library *setupapi.dll* explicitly at run-time.

CUsbIoPipe class

This class implements the interface to an USB pipe that is exported by the USBIO device driver. It provides all pipe-related functions that can be executed on a file handle that is bound to an USB endpoint. Particularly, it provides the functions needed for a data transfer to or from an endpoint.

Note that this class is derived from **CUsbIo**. All general and device-related functions can be executed on an instance of CUsbIoPipe as well.

Member Functions

CUsbIoPipe::CUsbIoPipe

Standard constructor of the CUsbIoPipe class.

Definition

```
CUsbIoPipe();
```

See Also

CUsbIoPipe::˜CUsbIoPipe (page 229)

CUsbIoPipe::˜CUsbIoPipe

Destructor of the CUsbIoPipe class.

Definition

```
˜CUsbIoPipe();
```

See Also

CUsbIoPipe::CUsbIoPipe (page 229)

CUsbIoPipe::Bind

Open the device, if required, and bind this object to an endpoint of the USB device.

Definition

```
DWORD
Bind(
  int DeviceNumber,
  UCHAR EndpointAddress,
  HDEVINFO DeviceList = NULL,
  const GUID* InterfaceGuid = NULL,
  bool CheckApiVersion = true
  );
```

Parameters

`DeviceNumber`
Specifies the index number of the USB Device. The index is zero-based. Note that the association between this number and the USB device can change with each call to **CUsbIo::CreateDeviceList**. For more details see also **CUsbIo::Open**.

Note that this parameter is ignored if the device has already been opened. See the comments below.

`EndpointAddress`
Specifies the address of the endpoint to which to bind the object. The endpoint address is specified as reported in the corresponding endpoint descriptor.

The endpoint address includes the direction flag at bit position 7 (MSb).

Bit 7 = 0: OUT endpoint

Bit 7 = 1: IN endpoint

For example, an IN endpoint with endpoint number 1 has the endpoint address 0x81.

`DeviceList`
A handle to the internal device list which was returned by the function **CUsbIo::CreateDeviceList** or NULL. For more details see also **CUsbIo::Open**.

Note that this parameter is ignored if the device has already been opened. See the comments below.

`InterfaceGuid`
Points to a caller-provided variable of type GUID, or can be set to NULL. The provided GUID is the predefined interface GUID of the USBIO device driver, or a user-defined GUID which has to be defined in the USBIO.INF file. For more details see also **CUsbIo::Open**.

Note that this parameter is ignored if the device has already been opened. See the comments below.

`CheckApiVersion = true`
If this parameter is true the bind function fails with the error code **`USBIO_ERR_VERSION_MISMATCH`** if the API version of the used driver is less than the expected or the major version of the API is different.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

If an USB device has not already been opened this function calls **CUsbIo::Open** to attach the object to a device. It passes the parameters **`DeviceNumber`**, **`DeviceList`**, and **`InterfaceGuid`** unmodified to **CUsbIo::Open**. Thus, a device and an endpoint can be attached to the object in one step.

Alternatively, an application can attach a device first by means of **CUsbIo::Open** (derived from **CUsbIo**, and then in a second step attach an endpoint by means of Bind. The parameters **`DeviceNumber`**, **`DeviceList`**, and **`InterfaceGuid`** will be ignored in this case.

The device must be set to the configured state before an endpoint can be bound, see **CUsbIo::SetConfiguration**. Only endpoints that are included in the active configuration can be bound by this function and subsequently used for a data transfer.

Note that an instance of the CUsbIoPipe class can be bound to exactly one endpoint only. Consequently, one instance has to be created for each endpoint to be activated.

This function is a wrapper for the **IOCTL_USBIO_BIND_PIPE** operation. See also the description of **IOCTL_USBIO_BIND_PIPE** for further information.

See Also

CUsbIoPipe::Unbind (page 233)
CUsbIo::CreateDeviceList (page 175)
CUsbIo::Open (page 177)
CUsbIo::Close (page 180)
CUsbIo::SetConfiguration (page 209)
IOCTL_USBIO_BIND_PIPE (page 96)

CUsbIoPipe::BindPipe

Bind this object to an endpoint of the USB device.

Definition

```
DWORD
BindPipe(
  UCHAR EndpointAddress
  );
```

Parameter

EndpointAddress

Specifies the address of the endpoint to bind the object to. The endpoint address is specified as reported in the corresponding endpoint descriptor.

The endpoint address includes the direction flag at bit position 7 (MSb).

Bit 7 = 0: OUT endpoint

Bit 7 = 1: IN endpoint

For example, an IN endpoint with endpoint number 1 has the endpoint address 0x81.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

CUsbIo::Open must have been called before this call. This means an application can attach to a device first by means of **CUsbIo::Open** (derived from **CUsbIo**), and then in a second step attach an endpoint by means of BindPipe.

The device must be set to the configured state before an endpoint can be bound, see **CUsbIo::SetConfiguration**. Only endpoints that are included in the active configuration can be bound by this function and subsequently used for a data transfer.

Note that an instance of the CUsbIoPipe class can be bound to exactly one endpoint only. Consequently, one instance has to be created for each endpoint to be activated.

This function is a wrapper for the **IOCTL_USBIO_BIND_PIPE** operation. See also the description of **IOCTL_USBIO_BIND_PIPE** for further information.

See Also

CUsbIoPipe::Unbind (page 233)
CUsbIo::Open (page 177)
CUsbIo::SetConfiguration (page 209)
IOCTL_USBIO_BIND_PIPE (page 96)

CUsbIoPipe::Unbind

Delete the association between the object and an endpoint.

Definition

```
DWORD
Unbind();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

A call to this function causes all pending read and write requests to be aborted.

After this function was called the object can be bound to another endpoint. However, it is recommended to use a separate object for each endpoint. See also the comments on **CUsbIoPipe::Bind**.

It is not an error to call Unbind when no endpoint is currently bound. The function does nothing in this case.

Note that closing the device either by means of **CUsbIo::Close** or by destructing the object will also cause an unbind. Thus, normally there is no need to call Unbind explicitly.

This function is a wrapper for the **IOCTL_USBIO_UNBIND_PIPE** operation. See also the description of **IOCTL_USBIO_UNBIND_PIPE** for further information.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIo::Close (page 180)
IOCTL_USBIO_UNBIND_PIPE (page 97)

CUsbIoPipe::Read

Submit a read request on the pipe.

Definition

```
BOOL
Read(
  CUsbIoBuf* Buf
  );
```

Parameter

Buf
Pointer to a buffer descriptor to which the read buffer is attached. The buffer descriptor has to be prepared by the caller. See the comments below.

Return Value

Returns TRUE if the request was successfully submitted, FALSE otherwise. If FALSE is returned then the **Status** member of **Buf** contains an error code.

Comments

The function submits the buffer memory that is attached to **Buf** to the USBIO device driver. The caller has to prepare the buffer descriptor pointed to by **Buf**. Particularly, the **NumberOfBytesToTransfer** member has to be set to the number of bytes to read.

The call returns immediately (asynchronous behavior). After the function succeeds the read operation is pending. It will be completed later on by the USBIO driver when data is received from the device. To determine when the operation has been completed the function **CUsbIoPipe::WaitForCompletion** should be called.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIoPipe::ReadSync (page 238)
CUsbIoPipe::Write (page 235)
CUsbIoPipe::WaitForCompletion (page 236)
CUsbIoBuf (page 272)

CUsbIoPipe::Write

Submit a write request on the pipe.

Definition

```
BOOL
Write(
  CUsbIoBuf* Buf
  );
```

Parameter

Buf
Pointer to a buffer descriptor the write buffer is attached to. The buffer descriptor has to be prepared by the caller. See the comments below.

Return Value

Returns TRUE if the request was successfully submitted, FALSE otherwise. If FALSE is returned then the **Status** member of **Buf** contains an error code.

Comments

The function submits the buffer memory that is attached to **Buf** to the USBIO device driver. The buffer contains the data to be written. The caller has to prepare the buffer descriptor pointed to by **Buf**. Particularly, the **NumberOfBytesToTransfer** member has to be set to the number of bytes to write.

The call returns immediately (asynchronous behavior). After the function succeeded the write operation is pending. It will be completed later on by the USBIO driver when data has been sent to the device. To determine when the operation has been completed the function **CUsbIoPipe::WaitForCompletion** should be called.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIoPipe::WriteSync (page 240)
CUsbIoPipe::Read (page 234)
CUsbIoPipe::WaitForCompletion (page 236)
CUsbIoBuf (page 272)

CUsbIoPipe::WaitForCompletion

Wait for completion of a pending read or write operation.

Definition

```
DWORD
WaitForCompletion(
  CUsbIoBuf* Buf,
  DWORD Timeout = INFINITE,
  HANDLE AdditionalEvent = NULL
  );
```

Parameters

Buf
Pointer to the buffer descriptor that has been submitted by means of **CUsbIoPipe::Read** or **CUsbIoPipe::Write**.

Timeout
Specifies a timeout interval, in milliseconds. The function returns with a status code of **USBIO_ERR_TIMEOUT** if the interval elapses and the read or write operation is still pending.

When INFINITE is specified then the interval never elapses. The function does not return until the read or write operation is finished.

When 0 is specified the function returns immediately. It returns **USBIO_ERR_TIMEOUT** if the operation is still pending or **USBIO_ERR_SUCCESS** if the operation is completed.

AdditionalEvent
Optionally, specifies an additional event object which has been created by the caller. If this parameter is not NULL then the specified event object will be included in the wait operation which this function performs internally to wait for completion of the buffer. The function returns immediately with a status code of **USBIO_ERR_ADDITIONAL_EVENT_SIGNALLED** if the additional event object gets signalled, regardless of the status of the buffer.

This parameter is set to NULL if the caller does not require an additional event object to be included in the wait operation.

Return Value

The function returns **USBIO_ERR_TIMEOUT** if the timeout interval elapsed. It returns **USBIO_ERR_ADDITIONAL_EVENT_SIGNALLED** if an additional event object has been specified and this event was signalled before the buffer completed or the timeout interval elapsed.

If the read or write operation has been finished, the return value is the final completion status of the operation. Note that the **Status** member of **Buf** will also be set to the final completion status in this case. The completion status is 0 if the read or write operation has been successfully finished, an USBIO error code otherwise.

Comments

After a buffer was submitted to the USBIO device driver by means of **CUsbIoPipe::Read** or **CUsbIoPipe::Write** this function is used to wait for the completion of the data transfer. Note that WaitForCompletion can be called regardless of the return status of the **CUsbIoPipe::Read** or **CUsbIoPipe::Write** function. It returns always the correct status of the buffer.

Optionally, a timeout interval for the wait operation may be specified. When the interval elapses before the read or write operation is finished the function returns with a special status of **`USBIO_ERR_TIMEOUT`**. The data transfer operation is still pending in this case. WaitForCompletion should be called again until the operation is finished.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIoPipe::Read (page 234)
CUsbIoPipe::Write (page 235)
CUsbIoBuf (page 272)

CUsbIoPipe::ReadSync

Submit a read request on the pipe and wait for its completion.

Definition

```
DWORD
ReadSync(
  void* Buffer,
  DWORD& ByteCount,
  DWORD Timeout = INFINITE
  );
```

Parameters

Buffer
Pointer to a caller-provided buffer. The buffer receives the data transferred from the device.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Buffer**. After the function succeeds **ByteCount** contains the number of bytes successfully read.

Timeout
Specifies a timeout interval, in milliseconds. If the interval elapses and the read operation is not yet finished the function aborts the operation and returns with **USBIO_ERR_TIMEOUT**.

When INFINITE is specified then the interval never elapses. The function does not return until the read operation is finished.

Return Value

The function returns **USBIO_ERR_TIMEOUT** if the timeout interval elapsed and the read operation was aborted. If the read operation has been finished the return value is the completion status of the operation which is 0 for success, or an USBIO error code otherwise.

Comments

The function transfers data from the endpoint attached to the object to the specified buffer. The function does not return to the caller until the data transfer has been finished or aborted due to a timeout. It behaves in a synchronous manner.

Optionally, a timeout interval for the synchronous read operation may be specified. When the interval elapses before the operation is finished the function aborts the operation and returns with a special status of **USBIO_ERR_TIMEOUT**. In this case, it is not possible to determine the number of bytes already transferred. The library calls the Win32 API function **CancelIo()** to abort the current request. This function aborts all pending requests submitted with this thread and this handle. After a timeout error occurred **CUsbIoPipe::ResetPipe** should be called.

Note that there is some overhead involved when this function is used. This is due to a temporary Win32 Event object that is created and destroyed internally.

Note: Using synchronous read requests does make sense in rare cases only and can lead to unpredictable results. It is recommended to handle read operations asynchronously by means of **CUsbIoPipe::Read** and **CUsbIoPipe::WaitForCompletion**.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIoPipe::WriteSync (page 240)
CUsbIoPipe::Read (page 234)
CUsbIoPipe::WaitForCompletion (page 236)
CUsbIoPipe::ResetPipe (page 242)

CUsbIoPipe::WriteSync

Submit a write request on the pipe and wait for its completion.

Definition

```
DWORD
WriteSync(
  void* Buffer,
  DWORD& ByteCount,
  DWORD Timeout = INFINITE
  );
```

Parameters

Buffer
Pointer to a caller-provided buffer that contains the data to be transferred to the device.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Buffer**. This is the number of bytes to be transferred to the device. After the function succeeds **ByteCount** contains the number of bytes successfully written.

Timeout
Specifies a timeout interval, in milliseconds. If the interval elapses and the write operation is not yet finished the function aborts the operation and returns with **USBIO_ERR_TIMEOUT**.

When INFINITE is specified then the interval never elapses. The function does not return until the write operation is finished.

Return Value

The function returns **USBIO_ERR_TIMEOUT** if the timeout interval elapsed and the write operation was aborted. If the write operation has been finished the return value is the completion status of the operation which is 0 for success, or an USBIO error code otherwise.

Comments

The function transfers data from the specified buffer to the endpoint attached to the object. The function does not return to the caller until the data transfer has been finished or aborted due to a timeout. It behaves in a synchronous manner.

Optionally, a timeout interval for the synchronous write operation may be specified. When the interval elapses before the operation is finished the function aborts the operation and returns with a special status of **USBIO_ERR_TIMEOUT**. In this case, it is not possible to determine the number of bytes already transferred. The library calls the Win32 API function **CancelIo()** to abort the current request. This function aborts all pending requests submitted with this thread and this handle. After a timeout error occurred **CUsbIoPipe::ResetPipe** should be called.

Note that there is some overhead involved when this function is used. This is due to a temporary Win32 Event object that is created and destroyed internally.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIoPipe::ReadSync (page 238)
CUsbIoPipe::ResetPipe (page 242)

CUsbIoPipe::ResetPipe

Reset pipe.

Definition

```
DWORD
ResetPipe();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function resets the software state of a pipe in the USB driver stack. Additionally, on a bulk or interrupt pipe, a CLEAR_FEATURE Endpoint Stall request will be generated on the USB. This should reset the endpoint state in the device as well.

This function has to be used after an error condition occurred on the pipe and the pipe was halted by the USB drivers.

It is recommended to call ResetPipe every time a data transfer is initialized on the pipe.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_RESET_PIPE** operation. See also the description of **IOCTL_USBIO_RESET_PIPE** for further information.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIoPipe::AbortPipe (page 243)
IOCTL_USBIO_RESET_PIPE (page 98)

CUsbIoPipe::AbortPipe

Cancel all pending read and write requests on this pipe.

Definition

```
DWORD
AbortPipe();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function is used to abort pending I/O operations on the pipe. All pending buffers will be returned to the application with an error status. Note that it is not possible to determine the number of bytes already transferred to or from an aborted buffer.

After a call to this function and before the data transfer is restarted the state of the pipe should be reset by means of **CUsbIoPipe::ResetPipe**. See also the comments on **CUsbIoPipe::ResetPipe**.

Note that it will take some milliseconds to cancel all buffers. Therefore, AbortPipe should not be called periodically.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_ABORT_PIPE** operation. See also the description of **IOCTL_USBIO_ABORT_PIPE** for further information.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIoPipe::ResetPipe (page 242)
IOCTL_USBIO_ABORT_PIPE (page 99)

CUsbIoPipe::GetPipeParameters

Query pipe-related parameters from the USBIO device driver.

Definition

```
DWORD
GetPipeParameters(
  USBIO_PIPE_PARAMETERS* PipeParameters
  );
```

Parameter

PipeParameters
Points to a caller-provided variable that receives the current parameter settings.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_GET_PIPE_PARAMETERS** operation. See also the description of **IOCTL_USBIO_GET_PIPE_PARAMETERS** for further information.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIoPipe::SetPipeParameters (page 245)
USBIO_PIPE_PARAMETERS (page 142)
IOCTL_USBIO_GET_PIPE_PARAMETERS (page 100)

CUsbIoPipe::SetPipeParameters

Set pipe-related parameters in the USBIO device driver.

Definition

```
DWORD
SetPipeParameters(
  const USBIO_PIPE_PARAMETERS* PipeParameters
  );
```

Parameter

PipeParameters
Points to a caller-provided variable that specifies the parameters to be set.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_SET_PIPE_PARAMETERS** operation. See also the description of **IOCTL_USBIO_SET_PIPE_PARAMETERS** for further information.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIoPipe::GetPipeParameters (page 244)
USBIO_PIPE_PARAMETERS (page 142)
IOCTL_USBIO_SET_PIPE_PARAMETERS (page 101)

CUsbIoPipe::PipeControlTransferIn

Generates a control transfer (SETUP token) on the pipe with a data phase in device to host (IN) direction.

Definition

```
DWORD
PipeControlTransferIn(
  void* Buffer,
  DWORD& ByteCount,
  const USBIO_PIPE_CONTROL_TRANSFER* ControlTransfer
  );
```

Parameters

Buffer
Pointer to a caller-provided buffer. The buffer receives the data transferred in the IN data phase.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Buffer**. After the function successfully returned **ByteCount** contains the number of valid bytes returned in the buffer.

ControlTransfer
Pointer to a caller-provided variable that defines the request to be generated.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function is used to send a SETUP token to a Control type endpoint.

Note: This function cannot be used to send a SETUP request to the default endpoint 0.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN** operation. See also the description of **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN** for further information.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIoPipe::PipeControlTransferOut (page 248)

USBIO_PIPE_CONTROL_TRANSFER (page 148)
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN (page 106)

CUsbIoPipe::PipeControlTransferOut

Generates a control transfer (SETUP token) on the pipe with a data phase in host to device (OUT) direction.

Definition

```
DWORD
PipeControlTransferOut(
  const void* Buffer,
  DWORD& ByteCount,
  const USBIO_PIPE_CONTROL_TRANSFER* ControlTransfer
  );
```

Parameters

Buffer
Pointer to a caller-provided buffer that contains the data to be transferred in the OUT data phase.

ByteCount
When the function is called, **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Buffer**. This is the number of bytes to be transferred in the data phase. After the function successfully returned **ByteCount** contains the number of bytes transferred.

ControlTransfer
Pointer to a caller-provided variable that defines the request to be generated.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function is used to send a SETUP token to a Control type endpoint.

Note: This function cannot be used to send a SETUP request to the default endpoint 0.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT** operation. See also the description of **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT** for further information.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIoPipe::PipeControlTransferIn (page 246)

USBIO_PIPE_CONTROL_TRANSFER (page 148)
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT (page 107)

CUsbIoPipe::SetupPipeStatistics

Enables or disables a statistical analysis of the data transfer on the pipe.

Definition

```
DWORD
SetupPipeStatistics(
  ULONG AveragingInterval
  );
```

Parameter

AveragingInterval

Specifies the time interval, in milliseconds, that is used to calculate the average data rate of the pipe. A time averaging algorithm is used to continuously compute the mean value of the data transfer rate.

If **AveragingInterval** is set to zero then the average data rate computation is disabled. This is the default state. An application should only enable the average data rate computation if it is needed. This will save resources (kernel memory and CPU cycles).

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The USBIO driver is able to analyse the data transfer (outgoing or incoming) on a pipe and to calculate the average data rate on that pipe. A time averaging algorithm is used to continuously compute the mean value of the data transfer rate. In order to save resources (kernel memory and CPU cycles) the average data rate computation is disabled by default. It has to be enabled and to be configured by means of this function before it is available to an application. See also **CUsbIoPipe::QueryPipeStatistics** and **USBIO_PIPE_STATISTICS** for more information on pipe statistics.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_SETUP_PIPE_STATISTICS** operation. See also the description of **IOCTL_USBIO_SETUP_PIPE_STATISTICS** for further information.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIoPipe::QueryPipeStatistics (page 251)
IOCTL_USBIO_SETUP_PIPE_STATISTICS (page 102)

CUsbIoPipe::QueryPipeStatistics

Returns statistical data related to the pipe.

Definition

```
DWORD
QueryPipeStatistics(
  USBIO_PIPE_STATISTICS* PipeStatistics,
  ULONG Flags = 0
  );
```

Parameters

PipeStatistics
Points to a caller-provided variable that receives the statistical data.

Flags
This parameter is set to zero or any combination (bitwise OR) of the following values.

USBIO_QPS_FLAG_RESET_BYTES_TRANSFERRED
If this flag is specified then the BytesTransferred counter will be reset to zero after its current value has been captured. The BytesTransferred counter is an unsigned 64 bit integer. It counts the total number of bytes transferred on a pipe, modulo 2^64.

USBIO_QPS_FLAG_RESET_REQUESTS_SUCCEEDED
If this flag is specified then the RequestsSucceeded counter will be reset to zero after its current value has been captured. The RequestsSucceeded counter is an unsigned 32 bit integer. It counts the total number of read or write requests that have been completed successfully on a pipe, modulo 2^32.

USBIO_QPS_FLAG_RESET_REQUESTS_FAILED
If this flag is specified then the RequestsFailed counter will be reset to zero after its current value has been captured. The RequestsFailed counter is an unsigned 32 bit integer. It counts the total number of read or write requests that have been completed with an error status on a pipe, modulo 2^32.

USBIO_QPS_FLAG_RESET_ALL_COUNTERS
This value combines the three flags described above. If USBIO_QPS_FLAG_RESET_ALL_COUNTERS is specified then all three counters BytesTransferred, RequestsSucceeded, and RequestsFailed will be reset to zero after their current values have been captured.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The USBIO driver internally maintains some statistical data per pipe object. This function allows an application to query the actual values of the various statistics counters. Optionally, individual counters can be reset to zero after queried. See also **CUsbIoPipe::SetupPipeStatistics** and **USBIO_PIPE_STATISTICS** for more information on pipe statistics.

The USBIO driver is able to analyse the data transfer (outgoing or incoming) on a pipe and to calculate the average data rate on that pipe. In order to save resources (kernel memory and CPU cycles) this feature is disabled by default. It has to be enabled and to be configured by means of the function **CUsbIoPipe::SetupPipeStatistics** before it is available to an application. Thus, before an application starts, to (periodically) query the value of **`AverageRate`** that is included in the data structure **USBIO_PIPE_STATISTICS** it has to enable the continuous computation of this value by a call to **CUsbIoPipe::SetupPipeStatistics**. The other statistical counters contained in the **USBIO_PIPE_STATISTICS** structure will be updated by default and do not need to be enabled explicitly.

Note that the statistical data is maintained for each pipe object separately.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_QUERY_PIPE_STATISTICS** operation. See also the description of **IOCTL_USBIO_QUERY_PIPE_STATISTICS** for further information.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIoPipe::SetupPipeStatistics (page 250)
USBIO_PIPE_STATISTICS (page 146)
IOCTL_USBIO_QUERY_PIPE_STATISTICS (page 104)

CUsbIoPipe::ResetPipeStatistics

Reset the statistics counters of the pipe.

Definition

```
DWORD
ResetPipeStatistics();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The USBIO driver internally maintains some statistical data per pipe object. This function resets the counters BytesTransferred, RequestsSucceeded, and RequestsFailed to zero. See also **CUsbIoPipe::SetupPipeStatistics** and **USBIO_PIPE_STATISTICS** for more information on pipe statistics.

Note that this function calls **CUsbIoPipe::QueryPipeStatistics** to reset the counters.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

See Also

CUsbIoPipe::Bind (page 230)
CUsbIoPipe::SetupPipeStatistics (page 250)
CUsbIoPipe::QueryPipeStatistics (page 251)
USBIO_PIPE_STATISTICS (page 146)

CUsbIoThread class

This class provides a basic implementation of a worker-thread that is used to continuously perform I/O operations. CUsbIoThread is a base class for the **CUsbIoReader** and **CUsbIoWriter** worker-thread implementations.

The CUsbIoThread class contains pure virtual functions. Consequently, it is not possible to create an instance of the class.

Note that CUsbIoThread is derived from **CUsbIoPipe**. Thus, all USBIO functions can be executed on an instance of CUsbIoThread.

Member Functions

CUsbIoThread::CUsbIoThread

Constructs a CUsbIoThread object.

Definition

```
CUsbIoThread();
```

See Also

CUsbIoThread::˜CUsbIoThread (page 254)

CUsbIoThread::˜CUsbIoThread

Destructor of the CUsbIoThread class.

Definition

```
virtual
˜CUsbIoThread();
```

Comments

The internal worker-thread must have been terminated when the object's destructor is called. That means **CUsbIoThread::ShutdownThread** must be called before the object is destroyed.

See Also

CUsbIoThread::CUsbIoThread (page 254)
CUsbIoThread::ShutdownThread (page 258)

CUsbIoThread::AllocateBuffers

Allocate the internal buffer pool.

Definition

```
BOOL
AllocateBuffers(
  DWORD SizeOfBuffer,
  DWORD NumberOfBuffers
  );
```

Parameters

SizeOfBuffer
Specifies the size, in bytes, of the buffers to be allocated internally.

NumberOfBuffers
Specifies the number of buffers to be allocated internally.

Return Value

Returns TRUE in case of success, FALSE otherwise.

Comments

The function initializes an internal **CUsbIoBufPool** object. For more information on the parameters and the behavior of the function refer to the description of **CUsbIoBufPool::Allocate**.

See Also

CUsbIoThread::FreeBuffers (page 256)
CUsbIoBufPool (page 280)
CUsbIoBufPool::Allocate (page 282)

CUsbIoThread::FreeBuffers

Free the internal buffer pool.

Definition

```
void
FreeBuffers();
```

Comments

The function frees the buffers allocated by the internal **CUsbIoBufPool** object. For more information on the behavior of the function refer to the description of **CUsbIoBufPool::Free**.

See Also

CUsbIoThread::AllocateBuffers (page 255)
CUsbIoBufPool (page 280)
CUsbIoBufPool::Free (page 283)

CUsbIoThread::StartThread

Start the internal worker-thread.

Definition

```
BOOL
StartThread(
  DWORD MaxIoErrorCount = 3
  );
```

Parameter

MaxIoErrorCount
Specifies the maximum number of I/O errors caused by read or write operations that will be tolerated by the thread. The thread will terminate itself when the specified limit is reached.

Return Value

Returns TRUE in case of success, FALSE otherwise.

Comments

The internal worker-thread will be created. Possibly, it starts its execution before this function returns.

The error limit specified in **MaxIoErrorCount** prevents an endless loop in the worker-thread that can occur when the device permanently fails data transfer requests.

The internal buffer pool must have been initialized by means of **CUsbIoThread::AllocateBuffers** before this function is called.

Note: The internal worker-thread is a native system thread. That means it cannot call MFC (Microsoft Foundation Classes) functions. It is necessary to use *PostMessage*, *SendMessage* or some other communication mechanism to switch over to MFC-aware threads.

See Also

CUsbIoThread::AllocateBuffers (page 255)
CUsbIoThread::ThreadRoutine (page 263)
CUsbIoThread::ShutdownThread (page 258)

CUsbIoThread::ShutdownThread

Terminate the internal worker-thread.

Definition

```
BOOL
ShutdownThread();
```

Return Value

Returns TRUE in case of success, FALSE otherwise.

Comments

The function sets the member variable **`TerminateFlag`** to TRUE. Then it calls the virtual member function **CUsbIoThread::TerminateThread**. The implementation of **CUsbIoThread::TerminateThread** should cause the worker-thread to resume from a wait function and to terminate itself.

ShutdownThread blocks the caller until the worker-thread has been terminated by the operating system.

Calling ShutdownThread when the internal thread has not been started will not throw an error. The function does nothing in this case.

Note: This function has to be called before the CUsbIoThread object is destroyed. In other words, the worker-thread must have been terminated when the object's destructor **CUsbIoThread::~CUsbIoThread** is called.

See Also

CUsbIoThread::StartThread (page 257)
CUsbIoThread::~CUsbIoThread (page 254)
CUsbIoThread::TerminateThread (page 264)

CUsbIoThread::ProcessData

This handler is called by the worker-thread to process data that has been received from the device.

Definition

```
virtual void
ProcessData(
  CUsbIoBuf* Buf
  );
```

Parameter

Buf
Pointer to a buffer descriptor the buffer is attached to.

Comments

This handler function is used by the **CUsbIoReader** implementation of the worker-thread. It is called when data has been successfully received on the pipe. Note that the function is called in the context of the worker-thread.

There is a default implementation of ProcessData that is just empty. ProcessData should be overloaded by a derived class to implement a specific functionality.

An implementation of ProcessData should examine the fields **CUsbIoBuf::Status** and **CUsbIoBuf::BytesTransferred** to determine if there are valid data bytes in the buffer.

See Also

CUsbIoThread::StartThread (page 257)
CUsbIoThread::ProcessBuffer (page 260)
CUsbIoReader (page 266)
CUsbIoBuf (page 272)
CUsbIoBuf::Status (page 278)
CUsbIoBuf::BytesTransferred (page 278)

CUsbIoThread::ProcessBuffer

This handler is called by the worker-thread if a buffer must be filled with data before it will be submitted on the pipe.

Definition

```
virtual void
ProcessBuffer(
  CUsbIoBuf* Buf
  );
```

Parameter

Buf
Pointer to a buffer descriptor the buffer is attached to.

Comments

This handler function is used by the **CUsbIoWriter** implementation of the worker-thread. It is called when a buffer has to be filled before it will be submitted on the pipe. Note that the function is called in the context of the worker-thread.

There is a default implementation of ProcessBuffer. It fills the buffer with zeroes and sets the **CUsbIoBuf::NumberOfBytesToTransfer** member of **Buf** to the buffer's size. ProcessBuffer should be overloaded by a derived class to implement a specific functionality.

See Also

CUsbIoThread::StartThread (page 257)
CUsbIoThread::ProcessData (page 259)
CUsbIoWriter (page 269)
CUsbIoBuf (page 272)
CUsbIoBuf::NumberOfBytesToTransfer (page 278)

CUsbIoThread::BufErrorHandler

This handler is called by the worker-thread when a read or write operation has been completed with an error status.

Definition

```
virtual void
BufErrorHandler(
  CUsbIoBuf* Buf
  );
```

Parameter

Buf
Pointer to a buffer descriptor the failed buffer is attached to.

Comments

This handler function is used by both the **CUsbIoReader** and **CUsbIoWriter** implementation of the worker-thread. It is called when a read or write request failed. Note that the function is called in the context of the worker-thread.

There is a default implementation of BufErrorHandler that is just empty. BufErrorHandler should be overloaded by a derived class to implement a specific error handling.

An implementation of BufErrorHandler should examine the field **CUsbIoBuf::Status** to determine the reason for failing the read or write request.

See Also

CUsbIoThread::StartThread (page 257)
CUsbIoThread::ProcessData (page 259)
CUsbIoThread::ProcessBuffer (page 260)
CUsbIoReader (page 266)
CUsbIoWriter (page 269)
CUsbIoBuf (page 272)
CUsbIoBuf::Status (page 278)

CUsbIoThread::OnThreadExit

This notification handler is called by the worker-thread before the thread terminates itself.

Definition

```
virtual void
OnThreadExit();
```

Comments

The function is called in the context of the worker-thread.

There is a default implementation of OnThreadExit that is just empty. OnThreadExit can be overloaded by a derived class to implement a specific behavior.

See Also

CUsbIoThread::StartThread (page 257)
CUsbIoThread::ShutdownThread (page 258)

CUsbIoThread::ThreadRoutine

The main routine that is executed by the worker-thread.

Definition

```
virtual void
ThreadRoutine() = 0;
```

Comments

This is a pure virtual function. Consequently, it must be implemented by a derived class.

The derived classes **CUsbIoReader** and **CUsbIoWriter** implement this function.

See Also

CUsbIoThread::StartThread (page 257)
CUsbIoThread::ShutdownThread (page 258)
CUsbIoReader (page 266)
CUsbIoWriter (page 269)

CUsbIoThread::TerminateThread

This routine is called by **CUsbIoThread::ShutdownThread** to terminate the internal worker-thread.

Definition

```
virtual void
TerminateThread() = 0;
```

Comments

This is a pure virtual function. Consequently, it must be implemented by a derived class. Note that TerminateThread is called in the context of **CUsbIoThread::ShutdownThread**.

The derived classes **CUsbIoReader** and **CUsbIoWriter** implement this function.

See Also

CUsbIoThread::ShutdownThread (page 258)
CUsbIoReader (page 266)
CUsbIoWriter (page 269)

Data Members

`HANDLE` **`mThreadHandle`**
Specifies the priority of the thread. This member has to be set before the worker thread is started to use a specific priority. Otherwise, the default value of `THREAD_PRIORITY_HIGHEST` will be used.

`CUsbIoBufPool` **`BufPool`**
The internal buffer pool, see **CUsbIoThread::AllocateBuffers** and **CUsbIoThread::FreeBuffers**.

`HANDLE` **`ThreadHandle`**
The handle that identifies the worker-thread. The value is NULL if the worker-thread is not started.

`unsigned int` **`ThreadID`**
The thread ID assigned by the operating system for the worker-thread.

`volatile BOOL` **`TerminateFlag`**
This flag will be set to TRUE by **CUsbIoThread::ShutdownThread** to indicate that the worker-thread shall terminate itself.

`DWORD` **`MaxErrorCount`**
An error limit for the worker-thread's main loop. For a description see **CUsbIoThread::StartThread**.

`CUsbIoBuf*` **`FirstPending`**
Used by **CUsbIoReader** and **CUsbIoWriter** to implement a list of currently pending buffers.

`CUsbIoBuf*` **`LastPending`**
Used by **CUsbIoReader** and **CUsbIoWriter** to implement a list of currently pending buffers.

CUsbIoReader class

This class implements a worker-thread that continuously reads a data stream from a pipe.

Note that this class is derived from **CUsbIoThread** which provides the basic handling of the internal worker-thread.

Member Functions

CUsbIoReader::CUsbIoReader

Constructs a CUsbIoReader object.

Definition

```
CUsbIoReader();
```

See Also

CUsbIoReader::˜CUsbIoReader (page 266)

CUsbIoReader::˜CUsbIoReader

Destructor of the CUsbIoReader class.

Definition

```
virtual
˜CUsbIoReader();
```

Comments

The internal worker-thread must have been terminated when the object's destructor is called. That means **CUsbIoThread::ShutdownThread** must be called before the object is destroyed.

See Also

CUsbIoReader::CUsbIoReader (page 266)
CUsbIoThread::ShutdownThread (page 258)

CUsbIoReader::ThreadRoutine

The main routine that is executed by the worker-thread.

Definition

```
virtual void
ThreadRoutine();
```

Comments

This function implements the main loop of the worker-thread. It submits all buffers from the internal buffer pool to the driver and waits for the completion of the first buffer.

ThreadRoutine can be overloaded by a derived class to implement a different behavior.

See Also

CUsbIoThread::ThreadRoutine (page 263)
CUsbIoThread (page 254)
CUsbIoReader::TerminateThread (page 268)

CUsbIoReader::TerminateThread

This routine is called by **CUsbIoThread** when the worker-thread is to be terminated.

Definition

```
virtual void
TerminateThread();
```

Comments

The implementation of this function calls **CUsbIoPipe::AbortPipe**. This will cancel all pending read operations and cause the worker-thread to resume.

TerminateThread can be overloaded by a derived class to implement a different behavior.

See Also

CUsbIoThread::TerminateThread (page 264)
CUsbIoThread (page 254)
CUsbIoReader::ThreadRoutine (page 267)

CUsbIoWriter class

This class implements a worker-thread that continuously writes a data stream to a pipe.

Note that this class is derived from **CUsbIoThread** which provides the basic handling of the internal worker-thread.

Member Functions

CUsbIoWriter::CUsbIoWriter

Constructs a CUsbIoWriter object.

Definition

```
CUsbIoWriter();
```

See Also

CUsbIoWriter::~CUsbIoWriter (page 269)

CUsbIoWriter::~CUsbIoWriter

Destructor of the CUsbIoWriter class.

Definition

```
virtual
~CUsbIoWriter();
```

Comments

The internal worker-thread must have been terminated when the object's destructor is called. That means **CUsbIoThread::ShutdownThread** must be called before the object is destroyed.

See Also

CUsbIoWriter::CUsbIoWriter (page 269)
CUsbIoThread::ShutdownThread (page 258)

CUsbIoWriter::ThreadRoutine

The main routine that is executed by the worker-thread.

Definition

```
virtual void
ThreadRoutine();
```

Comments

This function implements the main loop of the worker-thread. It submits all buffers from the internal buffer pool to the driver and waits for the completion of the first buffer.

ThreadRoutine can be overloaded by a derived class to implement a different behavior.

See Also

CUsbIoThread::ThreadRoutine (page 263)
CUsbIoThread (page 254)
CUsbIoWriter::TerminateThread (page 271)

CUsbIoWriter::TerminateThread

This routine is called by **CUsbIoThread** when the worker-thread is to be terminated.

Definition

```
virtual void
TerminateThread();
```

Comments

The implementation of this function calls **CUsbIoPipe::AbortPipe**. This will cancel all pending read operations and cause the worker-thread to resume.

TerminateThread can be overloaded by a derived class to implement a different behavior.

See Also

CUsbIoThread::TerminateThread (page 264)
CUsbIoThread (page 254)
CUsbIoWriter::ThreadRoutine (page 270)

CUsbIoBuf class

This class is used as a buffer descriptor of buffers used for read and write operations.

Member Functions

CUsbIoBuf::CUsbIoBuf

Construct a CUsbIoBuf object.

Definition

```
CUsbIoBuf();
```

Comments

This is the default constructor. It creates an empty descriptor. No buffer is attached.

See Also

CUsbIoBuf::˜CUsbIoBuf (page 275)

CUsbIoBuf::CUsbIoBuf

Construct a CUsbIoBuf object and attach an existing buffer.

Definition

```
CUsbIoBuf(
  void* Buffer,
  DWORD BufferSize
  );
```

Parameters

Buffer
Points to a caller-provided buffer to be attached to the descriptor object.

BufferSize
Specifies the size, in bytes, of the buffer to be attached to the descriptor object.

See Also

CUsbIoBuf::˜CUsbIoBuf (page 275)

CUsbIoBuf::CUsbIoBuf

Construct a CUsbIoBuf object and allocate a buffer internally.

Definition

```
CUsbIoBuf(
  DWORD BufferSize
  );
```

Parameter

BufferSize
Specifies the size, in bytes, of the buffer to be allocated and attached to the descriptor object.

Comments

This constructor allocates a buffer of the specified size and attaches it to the descriptor object. The buffer will be automatically freed by the destructor of this class.

See Also

CUsbIoBuf::~CUsbIoBuf (page 275)

CUsbIoBuf::~CUsbIoBuf

Destructor for a CUsbIoBuf object.

Definition

```
~CUsbIoBuf();
```

Comments

The destructor frees a buffer that was allocated by a constructor. A buffer that has been attached after construction will not be freed.

See Also

CUsbIoBuf::CUsbIoBuf (page 272)

CUsbIoBuf::Buffer

Get buffer pointer.

Definition

```
void*
Buffer();
```

Return Value

The function returns a pointer to the first byte of the buffer that is attached to the descriptor object. The return value is NULL if no buffer is attached.

See Also

CUsbIoBuf::Size (page 277)

CUsbIoBuf::Size

Get buffer size, in bytes.

Definition

```
DWORD
Size();
```

Return Value

The function returns the size, in bytes, of the buffer that is attached to the descriptor object. The return value is 0 if no buffer is attached.

See Also

CUsbIoBuf::Buffer (page 276)

Data Members

`DWORD` **`NumberOfBytesToTransfer`**

This public member specifies the number of bytes to be transferred to or from the buffer in a subsequent read or write operation.

Note that this member has to be set before the read or write operation is initiated by means of **CUsbIoPipe::Read** or **CUsbIoPipe::Write**.

`DWORD` **`BytesTransferred`**

This public member indicates the number of bytes successfully transferred to or from the buffer during a read or write operation.

Note that this member will be set after a read or write operation is completed.

`DWORD` **`Status`**

This public member indicates the completion status of a read or write operation.

Note that this member will be set after a read or write operation is completed.

`CUsbIoBuf*` **`Next`**

This public member allows to build a chain of buffer descriptor objects. It is used by **CUsbIoBufPool**, **CUsbIoReader**, and **CUsbIoWriter** to manage buffer lists.

`BOOL` **`OperationFinished`**

This public member is used as a flag. If it is set to TRUE then it indicates that the data transfer operation is finished altogether. Read or write processing will be terminated by **CUsbIoReader** or **CUsbIoWriter**.

`DWORD_PTR` **`Context`**

This public member is a general purpose field. It will never be touched by any class in the USBIO class library. Thus, it can be used by an application to store a context value that it associates with the buffer object.

`void*` **`BufferMem`**

This protected member contains the address of the memory block that is attached to the CUsbIoBuf object. A value of NULL indicates that no memory block is attached.

`DWORD` **`BufferSize`**

This protected member contains the size, in bytes, of the memory block that is attached to the CUsbIoBuf object. The value is zero if no memory block is attached.

`OVERLAPPED` **`Overlapped`**

This protected member provides the OVERLAPPED data structure that is required to perform asynchronous (overlapped) I/O operations by means of the Win32 functions **`ReadFile`**, **`WriteFile`**, and **`DeviceIoControl`**. One instance of the OVERLAPPED structure is required per I/O buffer. The data structure stores context information while the asynchronous I/O operation is in progress. Most important, it provides a Win32 Event object that signals the completion of the asynchronous operation.

This member is used by **CUsbIoReader** and **CUsbIoWriter** to perform I/O operations.

BOOL **BufferMemAllocated**

This protected member provides a flag. It is set to TRUE if the memory block that is attached to the CUsbIoBuf object was allocated by a constructor of the class. The memory block has to be freed by the destructor in this case. The flag is set to FALSE if the memory block that is attached to the CUsbIoBuf object was provided by the user.

CUsbIoBufPool class

This class implements a pool of CUsbIoBuf objects. It is used by **CUsbIoReader** and **CUsbIoWriter** to simplify management of buffer pools.

Member Functions

CUsbIoBufPool::CUsbIoBufPool

Construct a CUsbIoBufPool object.

Definition

```
CUsbIoBufPool();
```

Comments

This is the default constructor. It creates an empty pool.

See Also

CUsbIoBufPool::~CUsbIoBufPool (page 281)

CUsbIoBufPool::~CUsbIoBufPool

Destructor for a CUsbIoBufPool object.

Definition

```
~CUsbIoBufPool();
```

Comments

The destructor frees all **CUsbIoBuf** objects allocated by the pool.

See Also

CUsbIoBufPool::CUsbIoBufPool (page 281)

CUsbIoBufPool::Allocate

Allocate all elements of the buffer pool.

Definition

```
BOOL
Allocate(
  DWORD SizeOfBuffer,
  DWORD NumberOfBuffers
  );
```

Parameters

SizeOfBuffer
Specifies the size, in bytes, of the buffers to be allocated internally.

NumberOfBuffers
Specifies the number of buffers to be allocated internally.

Return Value

Returns TRUE in case of success, FALSE otherwise.

Comments

The function allocates the required number of buffer descriptors (**CUsbIoBuf** objects). Then it allocates the specified amount of buffer memory. The total number of bytes to allocate is calculated as follows.

*TotalSize = NumberOfBuffers * SizeOfBuffer*

In a last step the buffers are attached to the descriptors and stored in an internal list.

The function fails by returning FALSE when an internal pool is already allocated. **CUsbIoBufPool::Free** has to be called before a new pool can be allocated.

See Also

CUsbIoBufPool::Free (page 283)

CUsbIoBufPool::Free

Free all elements of the buffer pool.

Definition

```
void
Free();
```

Comments

The function frees all buffer descriptors and all the buffer memory allocated by a call to **CUsbIoBufPool::Allocate**. The pool is empty after this call.

A call to Free on an empty pool is allowed. The function does nothing in this case.

Note that after a call to Free, another pool can be allocated by means of **CUsbIoBufPool::Allocate**.

See Also

CUsbIoBufPool::Allocate (page 282)

CUsbIoBufPool::Get

Get a buffer from the pool.

Definition

```
CUsbIoBuf*
Get();
```

Return Value

The function returns a pointer to the buffer descriptor removed from the pool, or NULL if the pool is exhausted.

Comments

The function removes a buffer from the pool and returns a pointer to the associated descriptor. The caller is responsible for releasing the buffer, see **CUsbIoBufPool::Put**.

See Also

CUsbIoBufPool::Put (page 285)
CUsbIoBufPool::CurrentCount (page 286)

CUsbIoBufPool::Put

Put a buffer back to the pool.

Definition

```
void
Put(
  CUsbIoBuf* Buf
  );
```

Parameter

Buf
Pointer to the buffer descriptor that was returned by **CUsbIoBufPool::Get**.

Comments

This function is called to release a buffer that was returned by **CUsbIoBufPool::Get**.

See Also

CUsbIoBufPool::Get (page 284)
CUsbIoBufPool::CurrentCount (page 286)

CUsbIoBufPool::CurrentCount

Get current number of buffers in the pool.

Definition

```
long
CurrentCount();
```

Return Value

The function returns the current number of buffers stored in the pool.

See Also

CUsbIoBufPool::Get (page 284)
CUsbIoBufPool::Put (page 285)

Data Members

`CRITICAL_SECTION` **`CritSect`**
This protected member provides a Win32 Critical Section object that is used to synchronize access to the members of the class. Thus, the CUsbIoBufPool object is thread-save. It can be accessed by multiple threads simultaneously.

`CUsbIoBuf*` **`Head`**
This protected member points to the first buffer object that is available in the pool. The buffer pool is managed by means of a single-linked list. This member points to the element at the head of the list. The **CUsbIoBuf** objects are linked by means of their **`Next`** member. The list is terminated by a **`Next`** pointer that is set to NULL.

Head is set to NULL if the buffer list is empty.

`long` **`Count`**
This protected member contains the number of buffers that are currently linked to the pool.

`CUsbIoBuf*` **`BufArray`**
This protected member points to the array of CUsbIoBuf objects that are allocated by the pool internally.

`char*` **`BufferMemory`**
This protected member points to the buffer memory block that is allocated by the pool internally.

CSetupApiDll class

This class provides a mean to load the system-provided *setupapi.dll* explicitly. The method is also called Run-Time Dynamic Linking.

The library *setupapi.dll* is part of the Win32 API and provides functions for managing Plug&Play devices. It is supported by Windows 98 and later systems. The file *setupapi.dll* is not available on older systems like Windows 95 and Windows NT. Therefore, if an application is implicitly linked to *setupapi.dll* then it will not load on older systems. In order to avoid such kind of problems the DLL should be loaded explicitly at run-time. The CSetupApiDll class provides the appropriate implementation.

Member Functions

CSetupApiDll::CSetupApiDll

Construct a CSetupApiDll object.

Definition

```
CSetupApiDll();
```

Comments

This is the default constructor. It initializes the object.

CSetupApiDll::~CSetupApiDll

Destructor for a CSetupApiDll object.

Definition

```
~CSetupApiDll();
```

Comments

The destructor frees the *setupapi.dll* library if loaded.

CSetupApiDll::Load

Load the system-provided library *setupapi.dll*.

Definition

```
BOOL
Load();
```

Return Value

The function returns TRUE if successful, FALSE otherwise.

Comments

The function loads the DLL and if successful initializes all function pointers to contain the address of the appropriate function.

The function can safely be called repeatedly. It returns TRUE if the *setupapi.dll* is already loaded.

See Also

CSetupApiDll::Release (page 290)

CSetupApiDll::Release

Release the *setupapi.dll* library.

Definition

```
void
Release();
```

Comments

The function frees the DLL and invalidates all function pointers.

The function can safely be called if the DLL is not loaded. It does not perform any operation in this case.

See Also

CSetupApiDll::Load (page 289)

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 needs to 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`** must guarantee proper code synchronization when accessing data structures.

See Also

CPnPNotificator (page 293)
CPnPNotificator::Initialize (page 294)
CPnPNotificator::EnableDeviceNotifications (page 297)

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`.

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 implement 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 called twice for the same object.

See Also

CPnPNotificator::Shutdown (page 296)
CPnPNotificator::EnableDeviceNotifications (page 297)

CPnPNotificator::DisableDeviceNotifications (page 298)
CPnPNotifyHandler::HandlePnPMessage (page 291)

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 294)
CPnPNotificator::EnableDeviceNotifications (page 297)
CPnPNotificator::DisableDeviceNotifications (page 298)

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 294)
CPnPNotificator::DisableDeviceNotifications (page 298)
CPnPNotifyHandler::HandlePnPMessage (page 291)

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 issuing **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 currently registered.

See Also

CPnPNotificator::EnableDeviceNotifications (page 297)
CPnPNotificator::Shutdown (page 296)

9 USBIO Demo Application

The USBIO Demo Application demonstrates the usage of the USBIO driver interface. It is based on the USBIO Class Library which covers the native API calls. The Application is designed to handle one USB device that can contain multiple pipes. It is possible to run multiple instances of the application, each connected to another USB device.

The USBIO Demo Application is a dialog based MFC (Microsoft Foundation Classes) application. The main dialog contains a button that allows the user to open an output window. All output data and all error messages are directed to this window. The button "Clear Output Window" discards the actual contents of the window.

The main dialog contains several dialog pages which allow to access the device-related driver operations. From the dialog page "Pipes" a separate dialog can be started for each configured pipe. The pipe dialogs are non-modal. More than one pipe dialog can be opened at a given point in time.

9.1 Dialog Pages for Device Operations

9.1.1 Device

This page allows the user to scan for available devices. The application enumerates the USBIO device objects currently available. It opens each device object and queries the USB device descriptor. The USB devices currently attached to USBIO are listed in the output window. A device can be opened and closed, and the device parameters can be requested or set.

Related driver interfaces:

- `CreateFile();`
- `CloseHandle();`
- **IOCTL_USBIO_GET_DEVICE_PARAMETERS** (page 81)
- **IOCTL_USBIO_SET_DEVICE_PARAMETERS** (page 82)

9.1.2 Descriptors

This page allows the user to query standard descriptors from the device. The index of the configuration and the string descriptors can be specified. The descriptors are dumped to the output window. Some descriptors are interpreted. Unknown descriptors are presented as HEX dump.

Related driver interfaces:

- **IOCTL_USBIO_GET_DESCRIPTOR** (page 66)

9.1.3 Configuration

This page is used to set a configuration, to unconfigure the device, or to request the current configuration.

Related driver interfaces:

- **IOCTL_USBIO_GET_DESCRIPTOR** (page 66)
- **IOCTL_USBIO_GET_CONFIGURATION** (page 71)
- **IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR** (page 73)
- **IOCTL_USBIO_SET_CONFIGURATION** (page 74)
- **IOCTL_USBIO_UNCONFIGURE_DEVICE** (page 76)

9.1.4 Interface

By using this page the alternate setting of a configured interface can be changed.

Related driver interfaces:

- **IOCTL_USBIO_SET_INTERFACE** (page 77)
- **IOCTL_USBIO_GET_INTERFACE** (page 72)

9.1.5 Pipes

This page allows the user to show all configured endpoints and interfaces by using the button "Get Configuration Info". A new non-modal dialog for each configured pipe can be opened as well.

Related driver interfaces:

- **IOCTL_USBIO_GET_CONFIGURATION_INFO** (page 83)
- **IOCTL_USBIO_BIND_PIPE** (page 96)
- **IOCTL_USBIO_UNBIND_PIPE** (page 97)

9.1.6 Class or Vendor Request

By using this page a class or vendor specific request can be send to the USB device.

Related driver interfaces:

- **IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST** (page 78)
- **IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST** (page 79)

9.1.7 Feature

This page can be used to send set or clear feature requests.

Related driver interfaces:

- **IOCTL_USBIO_SET_FEATURE** (page 68)
- **IOCTL_USBIO_CLEAR_FEATURE** (page 69)

9.1.8 Other

This page allows the user to query the device state, to reset the USB device, to get the current frame number, and to query or set the device power state.

Related driver interfaces:

- **IOCTL_USBIO_GET_STATUS** (page 70)
- **IOCTL_USBIO_RESET_DEVICE** (page 84)
- **IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER** (page 86)
- **IOCTL_USBIO_SET_DEVICE_POWER_STATE** (page 87)
- **IOCTL_USBIO_GET_DEVICE_POWER_STATE** (page 88)

9.2 Dialog Pages for Pipe Operations

Three different types of pipe dialogs can be selected. For IN pipes a **Read from pipe to file** dialog and a **Read from pipe to output window** dialog can be activated. For OUT pipes a **Write from file to pipe** dialog can be started. The pipe dialog **Read from pipe to output window** cannot be used with isochronous pipes.

When a new pipe dialog is opened it is bound to a pipe. If the dialog is closed the pipe is unbound. Each pipe dialog contains pipe-related and transfer-related functions. The first three dialog pages are the same in all pipe dialogs. The last page has a special meaning.

9.2.1 Pipe

By using this page it is possible to access the functions Reset Pipe, Abort Pipe, Get Pipe Parameters, and Set Pipe Parameters.

Related driver interfaces:

- **IOCTL_USBIO_RESET_PIPE** (page 98)
- **IOCTL_USBIO_ABORT_PIPE** (page 99)
- **IOCTL_USBIO_GET_PIPE_PARAMETERS** (page 100)
- **IOCTL_USBIO_SET_PIPE_PARAMETERS** (page 101)

9.2.2 Buffers

By means of this page the size and the number of buffers can be selected. For Interrupt and Bulk pipes the "Size of Buffer" field is relevant. For Isochronous pipes the "Number of Packets" field is relevant and the required buffer size is calculated internally. In the "Max Error Count" field a maximum number of errors can be specified. When this number is exceeded, the data transfer is aborted. Each successful transfer resets the error counter to zero.

9.2.3 Control

This dialog page allows to access user-defined control pipes. It cannot be used to access the default pipe (endpoint zero) of a USB device.

Related driver interfaces:

- **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN** (page 106)
- **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT** (page 107)

9.2.4 Read from Pipe to Output Window

This dialog page allows the user to read data from an Interrupt or Bulk pipe and to dump it to the output window. For large amounts of data the transfer may be slowed down because of the overhead involved with printing to the output window. The printing of the data can be enabled/disabled by the switch **Print to Output Window**.

Related driver interfaces:

- `ReadFile();`
- **IOCTL_USBIO_ABORT_PIPE** (page 99)

9.2.5 Read from Pipe to File

This dialog page allows the user to read data from the pipe to a file. This transfer type can be used for Isochronous pipes as well. The synchronization type of the Isochronous pipe has to be "asynchronous". The application does not support data rate feedback.

Related driver interfaces:

- `ReadFile();`
- **IOCTL_USBIO_ABORT_PIPE** (page 99)

9.2.6 Write from File to Pipe

This dialog page allows the user to write data from a file to the pipe. This transfer type can be used for Isochronous pipes as well. The synchronization type of the isochronous pipe has to be "asynchronous". The application does not support data rate feedback.

Related driver interfaces:

- `WriteFile();`
- **IOCTL_USBIO_ABORT_PIPE** (page 99)

10 Debug Support

10.1 Enable Debug Traces

The licensed version of the driver package contains the debug version of the driver. It is not part of the demo version. If the debug version is available, the package creator builds a signed release (rel) and debug (dbg) package. The debug version of the driver can generate text messages on a kernel debugger or a similar application to view kernel output like `DbgView`. These messages can help to analyze problems.

To enable the debug traces follow these steps.

- Create a customized driver package.
- Install the debug driver package in the `dbg` folder on your device.
- Reboot the PC to make sure the new driver is loaded. Connect your device.
- Open the registry editor with the following path:

  `\HKEY_LOCAL_MACHINE\System\CurrentControlSet\services\YOUR_SERVICE_NAME\`.

  YOUR_SERVICE_NAME is the name of the service name defined in the INF file.

  Edit the DWORD value key TraceMask. The messages are grouped to special topics. Each topic can be enabled with a bit in the trace mask. To enable the messages on bit 5 the TraceMask must be set to 0x00000020. The TraceMask 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 loaded.
- Use the program `DbgView` from Microsoft or a kernel debugger like WinDbg to receive the kernel traces. Connect your device. `DbgView` must be started with administrator privileges. Enable the "Capture Options" "Capture Kernel" and "Enable Verbose Kernel Output".
- The "Verbose Kernel Output" can be enabled generally by creating the registry key `Debug Print Filter` in the folder `HKLM\system\CurrentControlSet\Control\Session Manager`. Create the DWORD value `DEFAULT` with `0xf`. Reboot the PC.

The following table summarize the meaning of the debug bits.

Table 6: Trace Mask Bit Content

Bit	**Content**
0	Fatal errors
1	Warnings

Table 6: (continued)

Bit	Content
2	Information
3	PnP
4	Power management
5	Get/Set Descriptors
6	Open, close, create, cleanup
7	IO Control, dispatch
8	Read and Write
9	Submit Request
10	FX Firmware Download
17	Driver Entry
18	Delete Device
24	Dumps

11 Related Documents

References

[1] USBIO COM Interface Reference Manual, Thesycon GmbH,
http://www.thesycon.de

[2] Universal Serial Bus Specification 1.1,
http://www.usb.org

[3] Universal Serial Bus Specification 2.0,
http://www.usb.org

[4] USB device class specifications (Audio, HID, Printer, etc.),
http://www.usb.org

[5] Microsoft Developer Network (MSDN) Library,
http://msdn.microsoft.com/library/

[6] Windows Driver Development Kit,
http://msdn.microsoft.com/library/

[7] Windows Platform SDK,
http://msdn.microsoft.com/library/

Index