THESYCON

Thesycon® Systemsoftware&ConsultingGmbH

USBIO

Universal USB Device Driver
for Windows 98, Windows Millennium, and Windows 2000

Reference Manual

Version 1.41 **2000, December 20th**

Thesycon® Systemsoftware & Consulting GmbH

Wetzlarer Platz 1 · D-98693 Ilmenau · GERMANY

Tel: +49 3677 / 8462-0

Fax: +49 3677 / 8462-18

e-mail: USBIO@thesycon.de

http://www.thesycon.de

Copyright (c) 1998–2000 Thesycon Systemsoftware & Consulting GmbH

All Rights Reserved

Disclaimer

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

Trademarks

The following trade names are referenced throughout this manual:

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

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

Contents

Table of contents 8

1 Introduction 9

2 Overview 11

2.1 Platforms . . . 11

2.2 Features . . . 11

3 Architecture 13

3.1 USBIO Object Model . . . 14

3.1.1 USBIO Device Objects . . . 14

3.1.2 USBIO Pipe Objects . . . 15

3.2 Establishing a Connection to the Device . . . 17

3.3 Power Management . . . 18

3.4 Device State Change Notifications . . . 19

4 Programming Interface 21

4.1 Programming Interface Overview . . . 21

4.2 Control Requests . . . 22

IOCTL_USBIO_GET_DESCRIPTOR . . . 23

IOCTL_USBIO_SET_DESCRIPTOR . . . 24

IOCTL_USBIO_SET_FEATURE . . . 25

IOCTL_USBIO_CLEAR_FEATURE . . . 26

IOCTL_USBIO_GET_STATUS . . . 27

IOCTL_USBIO_GET_CONFIGURATION . . . 28

IOCTL_USBIO_GET_INTERFACE . . . 29

IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR . . . 30

IOCTL_USBIO_SET_CONFIGURATION . . . 31

IOCTL_USBIO_UNCONFIGURE_DEVICE . . . 32

IOCTL_USBIO_SET_INTERFACE . . . 33

IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST . . . 34

IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST . . . 35

IOCTL_USBIO_GET_DEVICE_PARAMETERS . . . 36

IOCTL_USBIO_SET_DEVICE_PARAMETERS . . . 37

IOCTL_USBIO_GET_CONFIGURATION_INFO . . . 38

IOCTL_USBIO_RESET_DEVICE 39
IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER 40
IOCTL_USBIO_SET_DEVICE_POWER_STATE 41
IOCTL_USBIO_GET_DEVICE_POWER_STATE 42
IOCTL_USBIO_GET_DRIVER_INFO 43
IOCTL_USBIO_CYCLE_PORT 44
IOCTL_USBIO_BIND_PIPE 45
IOCTL_USBIO_UNBIND_PIPE 46
IOCTL_USBIO_RESET_PIPE 47
IOCTL_USBIO_ABORT_PIPE 48
IOCTL_USBIO_GET_PIPE_PARAMETERS 49
IOCTL_USBIO_SET_PIPE_PARAMETERS 50
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN 51
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT 52
4.3 Data Transfer Requests 53
4.3.1 Bulk and Interrupt Transfers 53
Bulk or Interrupt Write Transfers 53
Bulk or Interrupt Read Transfers 53
4.3.2 Isochronous Transfers 54
Isochronous Write Transfers 55
Isochronous Read Transfers 55
4.4 Input and Output Structures 56
USBIO_DRIVER_INFO 57
USBIO_DESCRIPTOR_REQUEST 58
USBIO_FEATURE_REQUEST 59
USBIO_STATUS_REQUEST 60
USBIO_STATUS_REQUEST_DATA 61
USBIO_GET_CONFIGURATION_DATA 62
USBIO_GET_INTERFACE 63
USBIO_GET_INTERFACE_DATA 64
USBIO_INTERFACE_SETTING 65
USBIO_SET_CONFIGURATION 66
USBIO_CLASS_OR_VENDOR_REQUEST 67
USBIO_DEVICE_PARAMETERS 69
USBIO_INTERFACE_CONFIGURATION_INFO 70

USBIO_PIPE_CONFIGURATION_INFO 72
USBIO_CONFIGURATION_INFO 74
USBIO_FRAME_NUMBER 75
USBIO_DEVICE_POWER 76
USBIO_BIND_PIPE 77
USBIO_PIPE_PARAMETERS 78
USBIO_PIPE_CONTROL_TRANSFER 79
USBIO_ISO_TRANSFER 80
USBIO_ISO_PACKET 82
USBIO_ISO_TRANSFER_HEADER 83
4.5 Enumeration Types 84
USBIO_PIPE_TYPE 84
USBIO_REQUEST_RECIPIENT 85
USBIO_REQUEST_TYPE 86
USBIO_DEVICE_POWER_STATE 87
4.6 Error Codes 88

5 USBIO Class Library 91
5.1 CUsbIo Class 91
5.2 CUsbIoPipe Class 91
5.3 CUsbIoThread Class 92
5.4 CUsbIoReader Class 92
5.5 CUsbIoWriter Class 92
5.6 CUsbIoBuf Class 93
5.7 CUsbIoBufPool Class 93

6 USBIO Demo Application 95
6.1 Dialog Pages for Device Operations 95
6.1.1 Device 95
6.1.2 Descriptors 95
6.1.3 Configuration 95
6.1.4 Interface 96
6.1.5 Pipes 96
6.1.6 Class or Vendor Request 96
6.1.7 Feature 96
6.1.8 Other 97

6.2 Dialog Pages for Pipe Operations 97
6.2.1 Pipe 97
6.2.2 Buffers 97
6.2.3 Control 98
6.2.4 Read from Pipe to Output Window 98
6.2.5 Read from Pipe to File 98
6.2.6 Write from File to Pipe 98

7 Installation Issues **99**
7.1 Automated Installation: The USBIO Installation Wizard 99
7.2 Manual Installation: The USBIO Setup Information File 100
7.3 Uninstalling USBIO 103
7.4 Building a Customized Driver Setup 104

8 Registry Entries **107**

9 Related Documents **111**

Index **113**

1 Introduction

USBIO is a generic Universal Serial Bus (USB) device driver for Windows 98, Windows Millennium (ME), and Windows 2000. It is able to control any type of USB device and provides a convenient programming interface that can be used by Win32 applications.

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.

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

2 Overview

Support for the Universal Serial Bus (USB) is built into the Windows 98, Windows Millennium, and Windows 2000 operating systems. These systems include device drivers for the USB Host Controller hardware, for USB Hubs, and for some classes of USB devices. 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. in the case of 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 an USB device driver. This driver has to conform to the Win32 Driver Model (WDM) that defines a common driver architecture for Windows 98, Windows Millennium, and Windows 2000. 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. Especially, this might be useful during development or test of a new device. But in many cases it is also suitable to include the USBIO device driver in the final product. So there is no need to develop and test a custom device driver for the USB-based product at all.

2.1 Platforms

The USBIO driver supports the following operating system platforms:

- Windows 98 (Gold), the first release of Windows 98
- Windows 98 Second Edition (SE), the second release of Windows 98
- Windows Millennium, the successor to Windows 98
- Windows 2000, the successor to Windows NT
- Windows 2000 Service Pack 1

Note that Windows NT 4.0 and Windows 95 are not supported by USBIO.

2.2 Features

The USBIO driver provides the following features:

- Complies with the Win32 Driver Model (WDM)
- Supports Plug&Play
- Supports Power Management
- Provides an interface to USB devices that can be used by any Win32 application
- Provides an interface to USB endpoints (pipes) that is similar to files

- Fully supports asynchronous (overlapped) data transfer operations
- Supports the USB transfer types Control, Interrupt, Bulk, and Isochronous
- Multiple USB devices can be controlled by USBIO at the same time
- Multiple applications can use USBIO at the same time

The USBIO device driver can be used to control any USB device from a Win32 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)
- chip card devices

If a particular kernel mode interface (e.g. WDM Kernel Mode Streaming, 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 case it is possible to develop a custom device driver based on the source code of the USBIO though. Please contact Thesycon if you need support on such kind of project.

Although the USBIO device driver fully supports isochronous data pipes, there are some limitations with respect to isochronous data transfers. They result from the fact that the processing of the isochronous data streams has to be performed by the application which runs in user mode. 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, Chapter 5 for synchronization issues of isochronous data streams).

However, it is possible to support all kinds of isochronous data streams using the USBIO driver. But the delays that might be caused by the thread scheduler of the operating system should be taken into consideration.

3 Architecture

Figure 1 shows the USB driver stack that is part of the Windows 98, Windows Millennium, and Windows 2000 operating systems. All drivers are embedded within the WDM layered architecture.

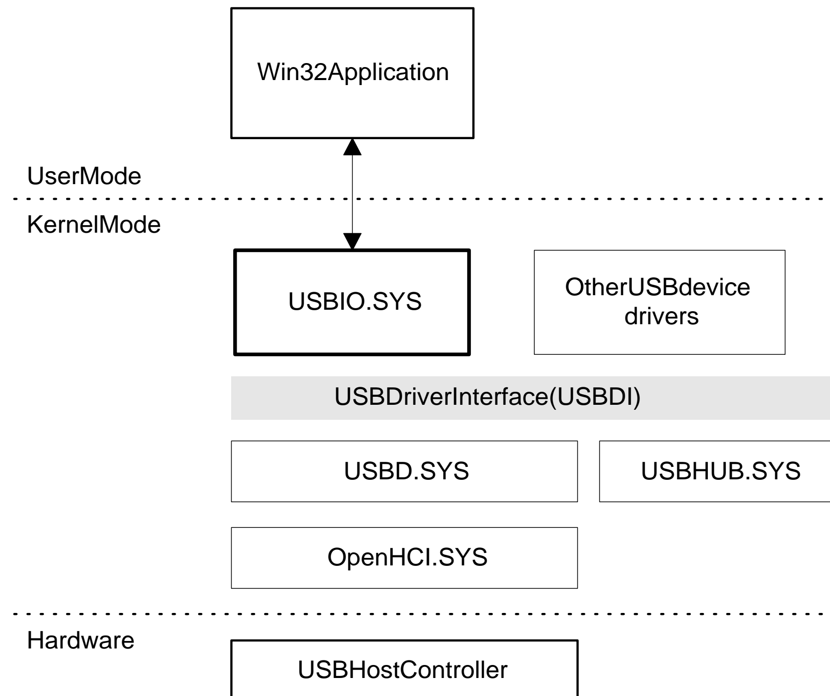

Figure 1: USB Driver Stack

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.
- OpenHCI.SYS is the host controller driver for controllers that conform with the Open Host Controller Interface specification. Optionally, it can be replaced by UHCD.SYS that is the Universal Host Controller Driver. Which driver is used depends on the mainboard chip set of the PC. For instance, Intel chipsets contain an Universal Host Controller.
- 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 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 for kernel mode drivers only. 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 Win32 applications. Thus, the USB driver stack becomes accessible to applications. A Win32 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 available to applications all the features that the USB driver stack provides at the USBDI level. For that reason the programming interface of the USBIO device driver (USBIOI) is closely related to the USBDI. But many of the functions cannot be translated in an one-to-one relationship.

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 in response to an Add Device request from the Plug&Play Manager of the operating 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 be considered as opaque by applications. 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 opaque 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, an user-defined Device Interface ID is supported by USBIO. This user-defined GUID is specified in the USBIO INF file by the `USBIO_UserInterfaceGuid` variable. If the user-defined interface ID is present at device initialization time USBIO registers the device with this ID. Thus, two interfaces – default and user-defined – 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 its 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 `USBIO_UserInterfaceGuid` 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 that 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 the same way.

Note:
Former versions of USBIO (up to V1.16) used a different device naming scheme. The device name was generated by appending an instance number to a common prefix. So the device names were static. In order to ensure compatibility USBIO still supports the old naming scheme. This feature can be enabled by defining a device name prefix in the variable `USBIO_DeviceBaseName` in the USBIO INF file. However, it is strongly recommended to use the new naming scheme based on Device Interface IDs (GUIDs), because it conforms with current Windows 2000 guidelines. The old-style static names should only be used if backward-compatibility with former versions of USBIO is required.

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. This endpoint is owned 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 "bind". After a binding 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 be bound to one pipe only. So there is always an one-to-one relation of pipe handles and pipe objects. This means that the application has to create a separate handle for each pipe it wants to access.

The USBIO driver also supports an "unbind" operation. That 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.

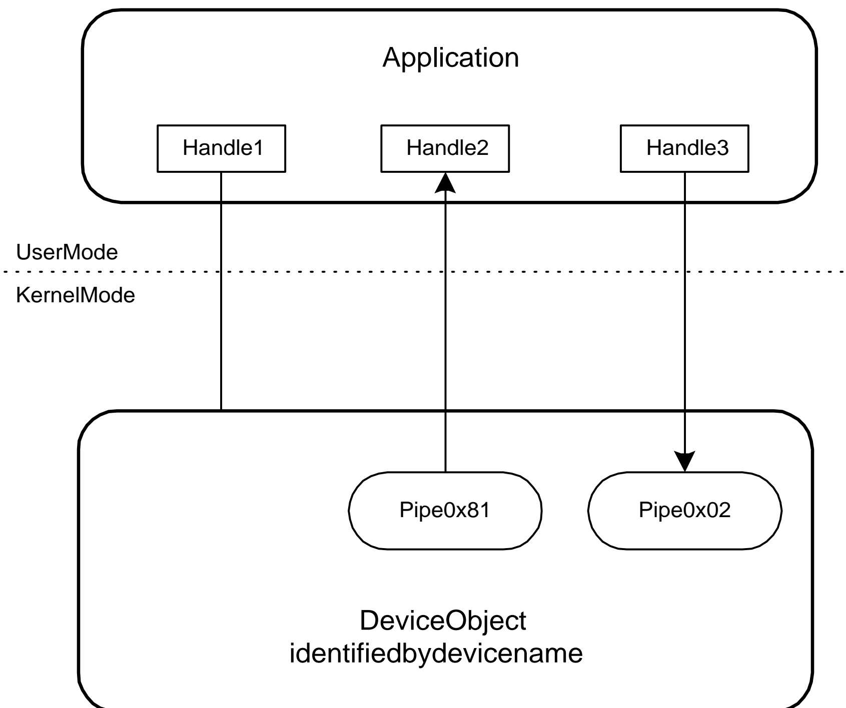

Figure 2: USBIO device and pipe objects example

The device object is identified by a device name as described in section 3.1.1. 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, any operation on the device can be executed by using Handle2 or Handle3, too.

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, 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 behaviour is required.

More code samples that show the usage of the USBIO programming interface can be found in the USBIO Class Library (USBIOLIB), the USBIO demo application (USBIOAPP), and the simple console applications ReaderCpp and ReadPipe.

3.3 Power Management

Windows 98, Windows Millennium, and Windows 2000 support system power management. That means that if the computer is idle for a given time, some parts of the computer can go into a sleeping mode. A system power change can be initiated by the user or by the operating system itself, on a low battery condition for example. An USB device driver has to support the system power management. Each device which supports power switching has to have a device power policy owner. It 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 goes down. Otherwise, a

message box appears on the screen and informs the user that the system is not able to go into the sleeping mode.

Before the system goes into a sleeping state the driver has to save all the information that it needs to reinitialize the device (device context) if 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 sleeping state D1 or D2 is handled transparent for the application. In the state D3 (USB Off) the device context is lost. Because the information stored in the device is known to the application only (e.g. the current volume level of an audio device), the generic USBIO driver cannot restore the device context in a general way. This has to be done by the application. Note that Windows 2000 restores the USB configuration of the device (SET_CONFIGURATION request) after the system is resumed.

The behaviour with respect to power management can be customized by registry parameters. For example, if a long time measurement should be performed the computer has to be prevented from going power down. For a description of the supported registry parameters, see also chapter 8.

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 different from 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 an USB device changes. The Win32 API provides the function **RegisterDeviceNotification** for this purpose. This way, an application will be notified if an 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 usage 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 for details. We strongly recommend to use this new naming scheme.

Note:
The function **`UnregisterDeviceNotification`** should not be used on Windows 98. There is a bug in the implementation that causes the system to become unstable. So it may crash at some later point in time. The bug seems to be "well known", it was discussed in some Usenet groups.

4 Programming Interface

4.1 Programming Interface Overview

Table 1: I/O Operations supported by the USBIO device driver.

Operation	Used On	Bus Action
IOCTL_USBIO_GET_DRIVER_INFO	device	-
IOCTL_USBIO_GET_DESCRIPTOR	device	request on default pipe
IOCTL_USBIO_SET_DESCRIPTOR	device	request on default pipe
IOCTL_USBIO_SET_FEATURE	device	request on default pipe
IOCTL_USBIO_CLEAR_FEATURE	device	request on default pipe
IOCTL_USBIO_GET_STATUS	device	request on default pipe
IOCTL_USBIO_GET_CONFIGURATION	device	request on default pipe
IOCTL_USBIO_GET_INTERFACE	device	request on default pipe
IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR	device	-
IOCTL_USBIO_SET_CONFIGURATION	device	request on default pipe
IOCTL_USBIO_UNCONFIGURE_DEVICE	device	request on default pipe
IOCTL_USBIO_SET_INTERFACE	device	request on default pipe
IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST	device	request on default pipe
IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST	device	request on default pipe
IOCTL_USBIO_GET_DEVICE_PARAMETERS	device	-
IOCTL_USBIO_SET_DEVICE_PARAMETERS	device	-
IOCTL_USBIO_GET_CONFIGURATION_INFO	device	-
IOCTL_USBIO_RESET_DEVICE	device	reset on hub port, USBD assigns USB address
IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER	device	-
IOCTL_USBIO_GET_DEVICE_POWER_STATE	device	-
IOCTL_USBIO_SET_DEVICE_POWER_STATE	device	set properties on hub port
IOCTL_USBIO_BIND_PIPE	device	-
IOCTL_USBIO_UNBIND_PIPE	pipe	-
IOCTL_USBIO_RESET_PIPE	pipe	request on default pipe
IOCTL_USBIO_ABORT_PIPE	pipe	-
IOCTL_USBIO_GET_PIPE_PARAMETERS	pipe	-
IOCTL_USBIO_SET_PIPE_PARAMETERS	pipe	-
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN	pipe	request on pipe
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT	pipe	request on pipe
`ReadFile`	pipe	data transfer from pipe (IN)
`WriteFile`	pipe	data transfer to pipe (OUT)

4.2 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). The **DeviceIoControl** 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.

`lpInBuffer`
Pointer to a buffer that contains an **USBIO_DESCRIPTOR_REQUEST** (page 58) data structure. This data structure has to be filled completely by the caller.

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

`lpOutBuffer`
Pointer to a buffer that will receive the descriptor data.

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

Comments

The buffer that is passed to this function in **`lpOutBuffer`** should be large enough to hold the requested descriptor, otherwise only a part of the descriptor will be returned. The size of the output buffer should be a multiple of the packet size of the default pipe (endpoint zero).

IOCTL_USBIO_SET_DESCRIPTOR

The IOCTL_USBIO_SET_DESCRIPTOR operation sets a specific descriptor of the device.

`lpInBuffer`
Pointer to a buffer that contains an **USBIO_DESCRIPTOR_REQUEST** (page 58) data structure. This data structure has to be filled completely by the caller.

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

`lpOutBuffer`
Pointer to a buffer that contains the descriptor data to be set.

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

Comments

USB devices do not have to support this operation.

IOCTL_USBIO_SET_FEATURE

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

lpInBuffer
Pointer to a buffer that contains an **USBIO_FEATURE_REQUEST** (page 59) data structure. This data structure has to be filled completely by the caller.

nInBufferSize
Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**, which has to be 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.

Comments

The SET_FEATURE request appears on the bus with the parameters specified in the IOCTL_USBIO_SET_FEATURE structure.

IOCTL_USBIO_CLEAR_FEATURE

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

`lpInBuffer`
Pointer to a buffer that contains an **USBIO_FEATURE_REQUEST** (page 59) data structure. This data structure has to be filled completely by the caller.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`lpInBuffer`**, which has to be 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.

Comments

The CLEAR_FEATURE request appears on the bus with the parameters specified in the IOCTL_USBIO_CLEAR_FEATURE structure.

IOCTL_USBIO_GET_STATUS

The IOCTL_USBIO_GET_STATUS operation requests status for a specific recipient.

lpInBuffer
Pointer to a buffer that contains an **USBIO_STATUS_REQUEST** (page 60) data structure. This data structure has to be filled completely by the caller.

nInBufferSize
Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**, which has to be sizeof(USBIO_STATUS_REQUEST) for this operation.

lpOutBuffer
Pointer to a buffer that will receive an **USBIO_STATUS_REQUEST_DATA** (page 61) data structure.

nOutBufferSize
Specifies the size, in bytes, of the buffer pointed to by **lpOutBuffer**, which has to be at least sizeof(USBIO_STATUS_REQUEST_DATA) for this operation.

Comments

The GET_STATUS request appears on the bus with the parameters specified in the **USBIO_STATUS_REQUEST** (page 60) structure. The function returns the structure **USBIO_STATUS_REQUEST_DATA** (page 61) which contains two bytes of data.

IOCTL_USBIO_GET_CONFIGURATION

The IOCTL_USBIO_GET_CONFIGURATION operation returns the current configuration of the device.

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

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

`lpOutBuffer`
Pointer to a buffer that will receive an **USBIO_GET_CONFIGURATION_DATA** (page 62) data structure.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`lpOutBuffer`**, which has to be at least sizeof(USBIO_GET_CONFIGURATION_DATA) for this operation.

Comments

A GET_CONFIGURATION request appears on the bus. The structure **USBIO_GET_CONFIGURATION_DATA** (page 62) returns the configuration value. A value of zero means "not configured".

IOCTL_USBIO_GET_INTERFACE

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

`lpInBuffer`
: Pointer to a buffer that contains an **USBIO_GET_INTERFACE** (page 63) data structure. This data structure has to be filled completely by the caller.

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

`lpOutBuffer`
: Pointer to a buffer that will receive an **USBIO_GET_INTERFACE_DATA** (page 64) data structure.

`nOutBufferSize`
: Specifies the size, in bytes, of the buffer pointed to by **`lpOutBuffer`**, which has to be at least sizeof(USBIO_GET_INTERFACE_DATA) for this operation.

Comments

A GET_INTERFACE request appears on the bus. The structure **USBIO_GET_INTERFACE_DATA** (page 64) returns the current alternate setting of the interface specified in USBIO_GET_INTERFACE.

IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR

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

lpInBuffer
Pointer to a buffer that contains the configuration descriptor data.

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

lpOutBuffer
Not used with this operation. Set to NULL.

nOutBufferSize
Not used with this operation. Set to zero.

Comments

This request may be used to store an user-defined configuration descriptor within the USBIO driver. The stored descriptor is used by the USBIO driver in subsequent **IOCTL_USBIO_SET_CONFIGURATION** (page 31) operations. The usage of IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR is optional. If no user-defined configuration descriptor is stored, USBIO uses the descriptor from the device.

There may be cases where the USBD driver provided by Microsoft with Windows does not process correctly the configuration descriptor that is reported by the device. This means it would not be possible to configure the device. In this situation the **IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR** request may be used to work around the problem. This request enables the application to use a modified configuration descriptor. The application can get the configuration descriptor using **IOCTL_USBIO_GET_DESCRIPTOR** (page 23), modifiy it appropriately and store it in the USBIO driver using the **IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR** request. Thus, the modified configuration descriptor will be passed to USBD when the device is configured.

The following is an example for the problem described above:

In the endpoint descriptor of an audio device the **bmAttributes** field contains two additional bits of information as defined by the audio class specification. The USBD does not recognize the pipe correctly and returns an invalid pipe type, when the additional bits in **bmAttributes** are not masked off. This has to be done by the application.

IOCTL_USBIO_SET_CONFIGURATION

The IOCTL_USBIO_SET_CONFIGURATION operation is used to set the device configuration.

lpInBuffer
Pointer to a buffer that contains an **USBIO_SET_CONFIGURATION** (page 66) data structure. This data structure has to be filled completely by the caller.

nInBufferSize
Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**, which has to be 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.

Comments

A SET_CONFIGURATION request appears on the bus. The USBD generates additional SET_INTERFACE requests on the bus if necessary.

All available interfaces have to be configured, or the request will fail. The number of interfaces and the alternate setting for each interface have to be specified in the structure **USBIO_SET_CONFIGURATION** (page 66).

All pipe handles associated with the device 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** (page 38) may be used to query all available pipes and interfaces.

By default, the configuration descriptor that is reported by the device is passed to the USBD. If an user-defined configuration descriptor is stored with **IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR** (page 30), this descriptor is used.

IOCTL_USBIO_UNCONFIGURE_DEVICE

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

`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.

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.

IOCTL_USBIO_SET_INTERFACE

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

`lpInBuffer`
Pointer to a buffer that contains an **USBIO_INTERFACE_SETTING** (page 65) data structure. This data structure has to be filled completely by the caller.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`lpInBuffer`**, which has to be 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.

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** (page 38) may be used to query all available pipes and interfaces.

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.

`lpInBuffer`
Pointer to a buffer that contains an **USBIO_CLASS_OR_VENDOR_REQUEST** (page 67) data structure. This data structure has to be filled completely by the caller.

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

`lpOutBuffer`
Pointer to a buffer that receives the data transferred from the device during the data phase of the control transfer. If the request does not return any data, this value can be NULL.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`lpOutBuffer`**. If this value is set to zero then there is no data transfer phase.

Comments

A SETUP request appears on the default pipe (endpoint zero) of the USB device with the given parameters. If a data 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.

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.

`lpInBuffer`
Pointer to a buffer that contains an **USBIO_CLASS_OR_VENDOR_REQUEST** (page 67) data structure. This data structure has to be filled completely by the caller.

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

`lpOutBuffer`
Pointer to a buffer that contains the data to be transferred to the device during the data phase of the control transfer. If the request has no data phase this value can be NULL.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`lpOutBuffer`**. If this value is set to zero then there is no data transfer phase.

Comments

A SETUP request appears on the default pipe (endpoint zero) of the USB device with the given parameters. If a data 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.

IOCTL_USBIO_GET_DEVICE_PARAMETERS

The IOCTL_USBIO_GET_DEVICE_PARAMETERS operation returns USBIO settings related to a device.

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

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

`lpOutBuffer`
Pointer to a buffer that will receive an **USBIO_DEVICE_PARAMETERS** (page 69) data structure.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`lpOutBuffer`**, which has to be at least sizeof(USBIO_DEVICE_PARAMETERS) for this operation.

Comments

The default state of the device parameters is defined by a set of registry parameters which are read by the USBIO driver at startup. The current state may be queried using this request.

IOCTL_USBIO_SET_DEVICE_PARAMETERS

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

`lpInBuffer`
Pointer to a buffer that contains an **USBIO_DEVICE_PARAMETERS** (page 69) data structure. This data structure has to be filled completely by the caller.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`lpInBuffer`**, which has to be 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.

Comments

The default state of the device parameters is defined by a set of registry parameters which are read by the USBIO driver at startup. The current state may be modified using this request.

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 is configured.

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

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

`lpOutBuffer`
Pointer to a buffer that will receive an **USBIO_CONFIGURATION_INFO** (page 74) data structure.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`lpOutBuffer`**, which has to be at least sizeof(USBIO_CONFIGURATION_INFO) for this operation.

Comments

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

IOCTL_USBIO_RESET_DEVICE

The IOCTL_USBIO_RESET_DEVICE operation causes a reset at the hub port in which the device is plugged in.

`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.

Comments

The following events occur on the bus if this request is issued:
USB Reset
GET_DEVICE_DESCRIPTOR
USB Reset
SET_ADDRESS
GET_DEVICE_DESCRIPTOR
GET_CONFIGURATION_DESCRIPTOR

All pipes associated with the device will be unbound and all pending requests will be cancelled. Note that the device receives two USB Resets and a new USB address is assigned by USBD. After this operation the device is in the unconfigured state.

The USBIO driver allows an USB reset request only if the device is configured. That means **IOCTL_USBIO_SET_CONFIGURATION** (page 31) was successfully executed. If the device is in the unconfigured state this request returns with an error status. This limitation is caused by the behaviour of Windows 2000. A system crash would occur on Windows 2000 if an USB Reset would be issued for an unconfigured device. Therefore, USBIO does not allow to issue an USB Reset while the device is configured.

If the device changes its USB descriptor set during an USB Reset the **IOCTL_USBIO_CYCLE_PORT** (page 44) request should be used instead of IOCTL_USBIO_RESET_DEVICE.

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

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 USBD.

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

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

`lpOutBuffer`
Pointer to a buffer that will receive an **USBIO_FRAME_NUMBER** (page 75) data structure.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`lpOutBuffer`**, which has to be at least sizeof(USBIO_FRAME_NUMBER) for this operation.

Comments

The returned frame number is a 32 bit value. The lower 11 bits of this value correspond to the frame number value in the Start Of Frame token on the bus.

IOCTL_USBIO_SET_DEVICE_POWER_STATE

The IOCTL_USBIO_SET_DEVICE_POWER_STATE operation sets the power state of the device.

lpInBuffer
Pointer to a buffer that contains an **USBIO_DEVICE_POWER** (page 76) data structure.

nInBufferSize
Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**, which has to be at least 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.

Comments

The device power state is maintained internally by the USBIO driver. This request may be used to change the current power state.

If the device is set to a power state different from D0 all pending requests should be cancelled before.

SeeAlso

See the section Power Management (page 18) and the description of the data structure **USBIO_DEVICE_POWER** (page 76) for details.

IOCTL_USBIO_GET_DEVICE_POWER_STATE

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

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

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

`lpOutBuffer`
Pointer to a buffer that will receive an **USBIO_DEVICE_POWER** (page 76) data structure.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`lpOutBuffer`**, which has to be at least sizeof(USBIO_DEVICE_POWER) for this operation.

Comments

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

IOCTL_USBIO_GET_DRIVER_INFO

The IOCTL_USBIO_GET_DRIVER_INFO operation returns version information about the USBIO API and the driver binary that is currently running.

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

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

`lpOutBuffer`
Pointer to a buffer that will receive an **USBIO_DRIVER_INFO** (page 57) data structure.

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

Comments

An application should check if the API version of the driver that is currently running matches with the version it expects.

IOCTL_USBIO_CYCLE_PORT

The IOCTL_USBIO_CYCLE_PORT operation causes a new enumeration of the device.

`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.

Comments

The IOCTL_USBIO_CYCLE_PORT request is similar to the **IOCTL_USBIO_RESET_DEVICE** (page 39) request, except that from the software point of view a disconnect/connect is simulated. This request causes the following events to occur:

- The USBIO device instance that is associated with the USB device will be removed. The corresponding device 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 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** (page 39) if the USB device modifies its descriptors during an 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).

This request does not work if the system-provided multi-interface driver is used. This driver expects that all function device drivers send a CYCLE_PORT request within 5 seconds.

IOCTL_USBIO_BIND_PIPE

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

`lpInBuffer`
Pointer to a buffer that contains an **USBIO_BIND_PIPE** (page 77) data structure. This data structure has to be filled completely by the caller.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`lpInBuffer`**, which has to be 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.

Comments

The pipe is identified by its endpoint address. Only endpoints from 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).

SeeAlso

IOCTL_USBIO_SET_CONFIGURATION (page 31) and
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 38)

IOCTL_USBIO_UNBIND_PIPE

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

`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.

Comments

After this operation is successfully completed the handle is unbound and may be used to bind another pipe. It is not necessary to unbind a pipe handle before it is closed. Closing a handle unbinds it implicitly.

IOCTL_USBIO_RESET_PIPE

The IOCTL_USBIO_RESET_PIPE operation clears an error condition on a pipe.

`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.

Comments

If an error occurs while transferring data from or to a pipe the USBD halts the pipe and returns an error code. No further transfers can be performed while the pipe is halted. To recover from this error condition and to restart the pipe an **`IOCTL_USBIO_RESET_PIPE`** has to be sent to the pipe.

This request causes that a CLEAR_FEATURE(ENDPOINT_STALL) request appears on the bus. In addition, the endpoint handling in the USB host controller will be reinitialized.

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

IOCTL_USBIO_ABORT_PIPE

The IOCTL_USBIO_ABORT_PIPE operation causes that all outstanding requests for the pipe are cancelled.

`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.

Comments

All outstanding read or write requests on the pipe are aborted and returned with an error status of **`USBIO_ERR_CANCELLED`**.

IOCTL_USBIO_GET_PIPE_PARAMETERS

The IOCTL_USBIO_GET_PIPE_PARAMETERS operation returns USBIO settings related to a pipe.

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

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

`lpOutBuffer`
Pointer to a buffer that will receive an **USBIO_PIPE_PARAMETERS** (page 78) data structure.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`lpOutBuffer`**, which has to be at least sizeof(USBIO_PIPE_PARAMETERS) for this operation.

Comments

The default state of the pipe parameters is defined by a set of registry parameters which are read by the USBIO driver at startup. The current state can be queried by using this request.

IOCTL_USBIO_SET_PIPE_PARAMETERS

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

`lpInBuffer`
Pointer to a buffer that contains an **USBIO_PIPE_PARAMETERS** (page 78) data structure. This data structure has to be filled completely by the caller.

`nInBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`lpInBuffer`**, which has to be 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.

Comments

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

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.

lpInBuffer
: Pointer to a buffer that contains an **USBIO_PIPE_CONTROL_TRANSFER** (page 79) data structure. This data structure has to be filled completely by the caller.

nInBufferSize
: Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**, which has to be sizeof(USBIO_PIPE_CONTROL_TRANSFER) for this operation.

lpOutBuffer
: Pointer to a buffer that receives the data transferred from the device during the data phase of the control transfer. If no data transfer is required the pointer may be NULL.

nOutBufferSize
: Specifies the size, in bytes, of the buffer pointed to by **lpOutBuffer**. If this value is set to zero then there is no data transfer phase.

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 with this operation.

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.

`lpInBuffer`
Pointer to a buffer that contains an **USBIO_PIPE_CONTROL_TRANSFER** (page 79) data structure. This data structure has to be filled completely by the caller.

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

`lpOutBuffer`
Pointer to a buffer that contains the data transferred to the device during the data phase of the control transfer. If no data transfer is required the pointer may be NULL.

`nOutBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`lpOutBuffer`**. If this value is set to zero then there is no data transfer phase.

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 with this operation.

4.3 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 from or to 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.

4.3.1 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. But the buffer size has to be equal or smaller than the value specified in the MaximumTransferSize field of the **USBIO_INTERFACE_SETTING** (page 65) 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 FIFO size of the endpoint. 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 FIFO size of the endpoint. The buffer size should be a multiple of the FIFO size. Otherwise the last transaction can cause a buffer overflow error.

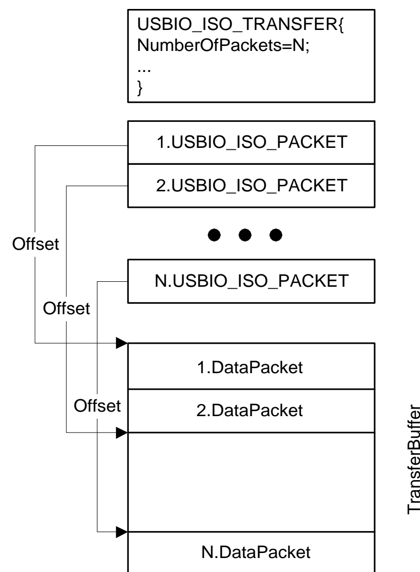

Figure 3: Layout of an isochronous transfer buffer

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 shorter than the FIFO size of the endpoint. For more information on receiving short packets see below. 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 will be completed with success by the system without performing a read operation on the USB.

The behaviour of the 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** (page 50) 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 FIFO size of the endpoint is completed with success. Otherwise, every data packet from the endpoint that is smaller than the FIFO size causes an error.

4.3.2 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. Therefore, the buffer that follows the header is divided into packets. Each packet is transmitted within an USB frame (normally 1 ms). The size of the packet can be different in each frame. This allows to support any data rate of the isochronous data stream.

The structure of the data buffer is shown in figure 3. The buffer contains an **USBIO_ISO_TRANSFER_HEADER** (page 83) structure of variable size at offset zero and the data packets. The header contains an **USBIO_ISO_TRANSFER** (page 80) structure that provides general information about the transfer buffer. An important member of this structure is the NumberOfPackets parameter. This parameter specifies the number of data packets contained in

the transfer buffer. The maximum number of packets that can be used in a single transfer is limited by the registry parameter `MaxIsoPackets`. Each data packet has to be described by an **USBIO_ISO_PACKET** (page 82) structure in the header. Because of that, the header contains a variable size array of **USBIO_ISO_PACKET** (page 82) elements.

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 and has to be filled by the application for write and for read transfers. The Length member defines the length, in bytes, of the packet. The packet length has to be specified by the application for write transfers and is returned by the USBIO on read transfers. The Status member is used to return the completion status of the transfer of the packet.

Isochronous Write Transfers

The sizes of the packets have to be less than or equal to the FIFO size of the endpoint. There must not be gaps between the data packets in the transfer buffer. The Offset and Length member of the USBIO_ISO_PACKET structures have to be initialized correctly before the transfer is started.

Isochronous Read Transfers

The size of each packet should be equal to the FIFO size. Otherwise a data overrun error can occur. The Offset member of the USBIO_ISO_PACKET structures has to be initialized correctly before the transfer is started. There must not be gaps between the data packets in the transfer buffer. The length of each received data packet is returned in the Length member of the corresponding USBIO_ISO_PACKET structure when the transfer completes.

Note:
Because the size of the received packets may be less than the FIFO size the data packets are not arranged continuously within the transfer buffer.

4.4 Input and Output Structures

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

USBIO_DRIVER_INFO

The USBIO_DRIVER_INFO structure contains version information about the driver binary and the programming interface.

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.

DriverVersion
Contains the version number of the driver executable.

The format is as follows: upper 8 bit = major version, lower 8 bit = minor version.

DriverBuildNumber
Contains the build number of the driver executable.

Flags
This field contains zero or any combination (bit-wise or) of the following values.

USBIO_INFOFLAG_CHECKED_BUILD
If this flag is set the driver that is currently running is a checked (debug) build.

USBIO_INFOFLAG_DEMO_VERSION
If this flag is set the driver that is currently running is a DEMO version that has some restrictions. Refer to ReadMe.txt for a description of the restrictions.

USBIO_INFOFLAG_LIGHT_VERSION
If this flag is set the driver that is currently running is a LIGHT version that has some restrictions. Refer to ReadMe.txt for a description of the restrictions.

Comments

This structure is an output of the **IOCTL_USBIO_GET_DRIVER_INFO** (page 43) operation.

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** (page 85).

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 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 for string descriptors. Set to zero for other descriptors.

Comments

This structure has to be used as an input for **IOCTL_USBIO_GET_DESCRIPTOR** (page 23) and **IOCTL_USBIO_SET_DESCRIPTOR** (page 24) requests.

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** (page 85).

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 has to be used as an input for **IOCTL_USBIO_SET_FEATURE** (page 25) and **IOCTL_USBIO_CLEAR_FEATURE** (page 26) requests.

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** (page 85).

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 has to be used as an input for **IOCTL_USBIO_GET_STATUS** (page 27) requests.

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 is an output of **IOCTL_USBIO_GET_STATUS** (page 27) requests.

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 is an output of **IOCTL_USBIO_GET_CONFIGURATION** (page 28) requests.

USBIO_GET_INTERFACE

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

Definition

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

Member

Interface
Specifies the interface number. The meaning is device-specific. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

Comments

This structure has to be used as an input for **IOCTL_USBIO_GET_INTERFACE** (page 29) requests.

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 is an output of **IOCTL_USBIO_GET_INTERFACE** (page 29) requests.

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. The value is 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 this interface. The value is defined by the device. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

`MaximumTransferSize`
Specifies the maximum size, 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 has to be used as an input for **IOCTL_USBIO_SET_INTERFACE** (page 33) and **IOCTL_USBIO_SET_CONFIGURATION** (page 31) requests.

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. The meaning of the value is defined by the device. 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** (page 65) structures that describes each interface in the configuration. There have to be NbOfInterfaces valid entries in this array.

Comments

This structure has to be used as an input for **IOCTL_USBIO_SET_CONFIGURATION** (page 31) requests.

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, 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. Otherwise, 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** (page 86).

Recipient
Specifies the recipient of the device request. The values are defined by the enumeration type **USBIO_REQUEST_RECIPIENT** (page 85).

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 control endpoint of the device. The format of the setup packet is defined by the Universal Serial Bus Specification 1.1, Chapter 9. The meanings of the values are device dependent.

This structure has to be used as an input for **IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST** (page 34) and **IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST** (page 35) operations.

USBIO_DEVICE_PARAMETERS

The USBIO_DEVICE_PARAMETERS structure contains device-specific parameter settings of the USBIO driver.

Definition

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

Members

`Options`

This field contains zero or any combination (bit-wise or) of the following values.

USBIO_RESET_DEVICE_ON_CLOSE

If this option is set, the USBIO driver generates an USB device reset after the last handle to the device was closed by the application. When this option is active the **`USBIO_UNCONFIGURE_ON_CLOSE`** flag will be ignored.

The default state of this option is defined by the registry parameter **`ResetDeviceOnClose`**.

USBIO_UNCONFIGURE_ON_CLOSE

If this option is set, the USBIO driver sets the USB device to its unconfigured state after the last handle to the device was closed by the application.

The default state of this option is defined by the registry parameter **`UnconfigureOnClose`**.

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. The USB device is able to wake 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 an USB configuration is set (device is configured).

The default state of this option is defined by the registry parameter **`EnableRemoteWakeup`**.

`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`**.

Comments

This structure is intended to be used with **IOCTL_USBIO_GET_DEVICE_PARAMETERS** (page 36) and **IOCTL_USBIO_SET_DEVICE_PARAMETERS** (page 37) operations.

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 the USB class specifications.

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

Protocol
Specifies the protocol code as reported by the device in the configuration descriptor. The meaning of this value is defined by the USB 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 is an output of **IOCTL_USBIO_GET_CONFIGURATION_INFO** (page 38) operations.

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** (page 84).

MaximumTransferSize
Specifies the maximum size, in bytes, of data transfers the 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 as 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. The 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 corresponds to the field `InterfaceNumber` of an **USBIO_INTERFACE_CONFIGURATION_INFO** (page 70) structure.

`reserved1`
Reserved field, set to zero.

`reserved2`
Reserved field, set to zero.

`reserved3`
Reserved field, set to zero.

Comments

This structure is an output of **IOCTL_USBIO_GET_CONFIGURATION_INFO** (page 38) operations. Only active pipes from the current configuration are returned.

USBIO_CONFIGURATION_INFO

The USBIO_CONFIGURATION_INFO structure provides information about all interfaces and all 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
Contains the number of interfaces. This is the number of valid entries in the **InterfaceInfo** structure.

NbOfPipes
Contains the number of pipes. This is the number of valid entries in the **PipeInfo** structure.

InterfaceInfo[USBIO_MAX_INTERFACES]
An array of **USBIO_INTERFACE_CONFIGURATION_INFO** (page 70) structures that describes the interfaces. There are **NbOfInterfaces** valid entries in this array.

PipeInfo[USBIO_MAX_PIPES]
An array of **USBIO_PIPE_CONFIGURATION_INFO** (page 72) structures that describes the pipes. There are **NbOfPipes** valid entries in this array.

Comments

This structure is an output of **IOCTL_USBIO_GET_CONFIGURATION_INFO** (page 38) operations. Only active pipes from the current configuration are returned.

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
Contains the current value of the frame counter maintained by the USBD.

Comments

This structure is an output of **IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER** (page 40) requests.

USBIO_DEVICE_POWER

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

Definition

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

Member

DevicePowerState
Contains the power state of the USB device. The values are defined by the **USBIO_DEVICE_POWER_STATE** (page 87) enumeration type.

Comments

This structure is used with **IOCTL_USBIO_GET_DEVICE_POWER_STATE** (page 42) and **IOCTL_USBIO_SET_DEVICE_POWER_STATE** (page 41) requests.

USBIO_BIND_PIPE

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

Definition

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

Member

EndpointAddress

Specifies the address of the endpoint on the USB device that shall be associated with the pipe. 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

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

Comments

This structure has to be used as an input for **IOCTL_USBIO_BIND_PIPE** (page 45) operations. Only active endpoints from the current configuration can be bound.

USBIO_PIPE_PARAMETERS

The USBIO_PIPE_PARAMETERS structure contains pipe specific parameter settings of the USBIO driver.

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, 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. Otherwise, a short packet causes an error condition. This option is meaningful for IN pipes only.

Comments

This structure is intended to be used with **IOCTL_USBIO_GET_PIPE_PARAMETERS** (page 49) and **IOCTL_USBIO_SET_PIPE_PARAMETERS** (page 50) operations. The default setting of this parameter can be changed by means of the registry parameter **ShortTransferOk**. This parameter 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** (page 80)).

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, 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. Otherwise, a short packet causes an error condition.

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

Comments

This structure has to be used as an input for **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN** (page 51) and **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT** (page 52) operations.

USBIO_ISO_TRANSFER

The USBIO_ISO_TRANSFER structure provides information used for isochronous data transfers.

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 received from the device. Each packet corresponds to an USB frame. The maximum number of packets in a read or write operation is limited by the registry parameter **MaxIsoPackets**.

Flags
This field contains zero or any combination (bit-wise or) of the following values.

USBIO_SHORT_TRANSFER_OK
If this flag is set, 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. Otherwise, a short packet causes an error condition.

USBIO_START_TRANSFER_ASAP
If this flag is set, 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 the transfer shall start with. The value has to be within a system-defined range relative to the current frame. The range is normally set to 1024 frames.

If **USBIO_START_TRANSFER_ASAP** is specified in **Flags**, this member has not to be set by the caller. It contains the frame number that the transfer started with, when the request is returned by the USBIO.

If **USBIO_START_TRANSFER_ASAP** is not specified in **Flags**, this member has to be set by the caller to the frame number this transfer shall start with. An error occurs if the frame number is outside of the valid range.

ErrorCount
Contains the total number of errors occurred during this transaction when the request is returned by the USBIO.

Comments

This structure is the fixed size part of the **USBIO_ISO_TRANSFER_HEADER** (page 83) that has to be used as an input for **`ReadFile`** and **`WriteFile`** operations with an isochronous pipe. The transfer buffer has to contain an **USBIO_ISO_TRANSFER_HEADER** (page 83) structure at offset zero.

USBIO_ISO_PACKET

The USBIO_ISO_PACKET structure defines the size and location of a single isochronous data packet within the transfer buffer that is used for isochronous data transfers.

Definition

```
typedef struct _USBIO_ISO_PACKET{
      ULONG Offset;
      ULONG Length;
      ULONG Status;
} USBIO_ISO_PACKET;
```

Members

`Offset`
Specifies the offset, in bytes, of the packet relative to the start of the data buffer. This parameter has to be specified by the caller for read and write operations.

`Length`
Specifies the size, in bytes, of the packet. This parameter has to be set by the caller for write operations. On read operations this field is set by the USBIO when the request is returned.

`Status`
Contains the final status code for the transfer of this packet when the request is returned by the USBIO.

Comments

A variable size array of **`USBIO_ISO_PACKET`** structures is part of the **USBIO_ISO_TRANSFER_HEADER** (page 83) that has to be used as an input for **`ReadFile`** and **`WriteFile`** operations with an isochronous pipe. An **`USBIO_ISO_PACKET`** structure is required for each data packet to be transferred. The maximum number of data packets is limited by the registry parameter **`MaxIsoPackets`**.

USBIO_ISO_TRANSFER_HEADER

The USBIO_ISO_TRANSFER_HEADER structure defines the header that has to be contained in the data buffers that are used for isochronous transfers.

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** (page 80) structure for more information.

`IsoPacket[1]`
This is a variable length array of **USBIO_ISO_PACKET** (page 82) structures. Each member defines an isochronous packet to be transferred. The number of valid entries in this array is defined by the **`NumberOfPackets`** field of **`IsoTransfer`**. The maximum number of data packets is limited by the registry parameter **`MaxIsoPackets`**.

Comments

The data buffer passed to **`ReadFile`** or **`WriteFile`** operations with an isochronous pipe has to contain a valid **USBIO_ISO_TRANSFER_HEADER** (page 83) 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 the data packets. Each data packet is transferred in a separate USB frame.

There must not be gaps between the data packets in the transfer buffer.

4.5 Enumeration Types

USBIO_PIPE_TYPE

The USBIO_PIPE_TYPE enumeration type contains values that identify the type of an USB pipe or an 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.

USBIO_REQUEST_RECIPIENT

The USBIO_REQUEST_RECIPIENT enumeration type contains values that identify the recipient of an 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.

USBIO_REQUEST_TYPE

The USBIO_REQUEST_TYPE enumeration type contains values that identify the type of an 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 only Class and Vendor requests are supported by the USBD interface. Standard requests are generated internally by the USBD.

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.

4.6 Error Codes

Table 2: Error codes defined by the USBIO device driver

USBIO_ERR_SUCCESS	(0x00000000L)
USBIO_ERR_CRC	(0xE0000001L)
USBIO_ERR_BTSTUFF	(0xE0000002L)
USBIO_ERR_DATA_TOGGLE_MISMATCH	(0xE0000003L)
USBIO_ERR_STALL_PID	(0xE0000004L)
USBIO_ERR_DEV_NOT_RESPONDING	(0xE0000005L)
USBIO_ERR_PID_CHECK_FAILURE	(0xE0000006L)
USBIO_ERR_UNEXPECTED_PID	(0xE0000007L)
USBIO_ERR_DATA_OVERRUN	(0xE0000008L)
USBIO_ERR_DATA_UNDERRUN	(0xE0000009L)
USBIO_ERR_RESERVED1	(0xE000000AL)
USBIO_ERR_RESERVED2	(0xE000000BL)
USBIO_ERR_BUFFER_OVERRUN	(0xE000000CL)
USBIO_ERR_BUFFER_UNDERRUN	(0xE000000DL)
USBIO_ERR_NOT_ACCESSED	(0xE000000FL)
USBIO_ERR_FIFO	(0xE0000010L)
USBIO_ERR_ENDPOINT_HALTED	(0xE0000030L)
USBIO_ERR_NO_MEMORY	(0xE0000100L)
USBIO_ERR_INVALID_URB_FUNCTION	(0xE0000200L)
USBIO_ERR_INVALID_PARAMETER	(0xE0000300L)
USBIO_ERR_ERROR_BUSY	(0xE0000400L)
USBIO_ERR_REQUEST_FAILED	(0xE0000500L)
USBIO_ERR_INVALID_PIPE_HANDLE	(0xE0000600L)
USBIO_ERR_NO_BANDWIDTH	(0xE0000700L)
USBIO_ERR_INTERNAL_HC_ERROR	(0xE0000800L)
USBIO_ERR_ERROR_SHORT_TRANSFER	(0xE0000900L)
USBIO_ERR_BAD_START_FRAME	(0xE0000A00L)
USBIO_ERR_ISOCH_REQUEST_FAILED	(0xE0000B00L)
USBIO_ERR_FRAME_CONTROL_OWNED	(0xE0000C00L)
USBIO_ERR_FRAME_CONTROL_NOT_OWNED	(0xE0000D00L)
USBIO_ERR_CANCELED	(0xE0010000L)
USBIO_ERR_CANCELING	(0xE0020000L)
USBIO_ERR_FAILED	(0xE0001000L)
USBIO_ERR_INVALID_INBUFFER	(0xE0001001L)
USBIO_ERR_INVALID_OUTBUFFER	(0xE0001002L)
USBIO_ERR_OUT_OF_MEMORY	(0xE0001003L)
USBIO_ERR_PENDING_REQUESTS	(0xE0001004L)
USBIO_ERR_ALREADY_CONFIGURED	(0xE0001005L)
USBIO_ERR_NOT_CONFIGURED	(0xE0001006L)
USBIO_ERR_OPEN_PIPES	(0xE0001007L)
USBIO_ERR_ALREADY_BOUND	(0xE0001008L)
USBIO_ERR_NOT_BOUND	(0xE0001009L)
USBIO_ERR_DEVICE_NOT_PRESENT	(0xE000100AL)
USBIO_ERR_CONTROL_NOT_SUPPORTED	(0xE000100BL)

Table 2: (continued)

USBIO_ERR_TIMEOUT	(0xE000100CL)
USBIO_ERR_INVALID_RECIPIENT	(0xE000100DL)
USBIO_ERR_INVALID_TYPE	(0xE000100EL)
USBIO_ERR_INVALID_IOCTL	(0xE000100FL)
USBIO_ERR_INVALID_DIRECTION	(0xE0001010L)
USBIO_ERR_TOO_MUCH_ISO_PACKETS	(0xE0001011L)
USBIO_ERR_POOL_EMPTY	(0xE0001012L)
USBIO_ERR_PIPE_NOT_FOUND	(0xE0001013L)
USBIO_ERR_INVALID_ISO_PACKET	(0xE0001014L)
USBIO_ERR_OUT_OF_ADDRESS_SPACE	(0xE0001015L)
USBIO_ERR_INTERFACE_NOT_FOUND	(0xE0001016L)
USBIO_ERR_INVALID_DEVICE_STATE	(0xE0001017L)
USBIO_ERR_INVALID_PARAM	(0xE0001018L)
USBIO_ERR_DEMO_EXPIRED	(0xE0001019L)
USBIO_ERR_INVALID_POWER_STATE	(0xE000101AL)
USBIO_ERR_POWER_DOWN	(0xE000101BL)
USBIO_ERR_VERSION_MISMATCH	(0xE000101CL)
USBIO_ERR_SET_CONFIGURATION_FAILED	(0xE000101DL)
USBIO_ERR_VID_RESTRICTION	(0xE0001080L)
USBIO_ERR_ISO_RESTRICTION	(0xE0001081L)
USBIO_ERR_BULK_RESTRICTION	(0xE0001082L)
USBIO_ERR_EP0_RESTRICTION	(0xE0001083L)
USBIO_ERR_PIPE_RESTRICTION	(0xE0001084L)
USBIO_ERR_PIPE_SIZE_RESTRICTION	(0xE0001085L)
USBIO_ERR_DEVICE_NOT_FOUND	(0xE0001100L)
USBIO_ERR_DEVICE_NOT_OPEN	(0xE0001102L)
USBIO_ERR_NO_SUCH_DEVICE_INSTANCE	(0xE0001104L)
USBIO_ERR_INVALID_FUNCTION_PARAM	(0xE0001105L)

5 USBIO Class Library

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.

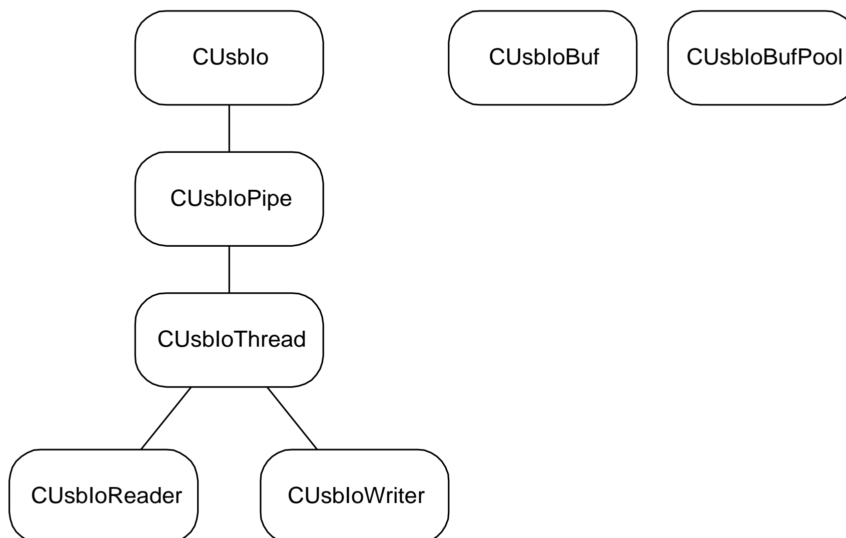

Figure 4: USBIO Class Library

5.1 CUsbIo Class

The class **CUsbIo** implements the basic interface to the USBIO device driver. It includes all functions that are related to an 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 **CUsbIo** class supports device enumeration and an **Open** function that is used to connect an instance of the class to an 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.

5.2 CUsbIoPipe Class

The class **CUsbIoPipe** extends the **CUsbIo** class by functions that are related to an USBIO pipe object. An instance of the **CUsbIoPipe** class is associated directly with an 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 from or to 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 were submitted. The **WaitForCompletion** member function is used to wait until the data transfer from or to 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.

5.3 CUsbIoThread Class

The class **CUsbIoThread** 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 **CUsbIoThread** 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.

5.4 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.

5.5 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.

5.6 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.

5.7 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.

6 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 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.

6.1 Dialog Pages for Device Operations

6.1.1 Device

This page allows 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 36)
- **IOCTL_USBIO_SET_DEVICE_PARAMETERS** (page 37)

6.1.2 Descriptors

This page allows 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 23)

6.1.3 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 23)
- **IOCTL_USBIO_GET_CONFIGURATION** (page 28)
- **IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR** (page 30)
- **IOCTL_USBIO_SET_CONFIGURATION** (page 31)
- **IOCTL_USBIO_UNCONFIGURE_DEVICE** (page 32)

6.1.4 Interface

By using this page the alternate setting of a configured interface can be changed.

Related driver interfaces:

- **IOCTL_USBIO_SET_INTERFACE** (page 33)
- **IOCTL_USBIO_GET_INTERFACE** (page 29)

6.1.5 Pipes

This page allows 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 38)
- **IOCTL_USBIO_BIND_PIPE** (page 45)
- **IOCTL_USBIO_UNBIND_PIPE** (page 46)

6.1.6 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 34)
- **IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST** (page 35)

6.1.7 Feature

This page can be used to send set or clear feature requests.

Related driver interfaces:

- **IOCTL_USBIO_SET_FEATURE** (page 25)
- **IOCTL_USBIO_CLEAR_FEATURE** (page 26)

6.1.8 Other

This page allows 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 27)
- **IOCTL_USBIO_RESET_DEVICE** (page 39)
- **IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER** (page 40)
- **IOCTL_USBIO_SET_DEVICE_POWER_STATE** (page 41)
- **IOCTL_USBIO_GET_DEVICE_POWER_STATE** (page 42)

6.2 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.

6.2.1 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 47)
- **IOCTL_USBIO_ABORT_PIPE** (page 48)
- **IOCTL_USBIO_GET_PIPE_PARAMETERS** (page 49)
- **IOCTL_USBIO_SET_PIPE_PARAMETERS** (page 50)

6.2.2 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.

6.2.3 Control

This dialog page allows to access user-defined control pipes. It cannot be used to access the default pipe (endpoint zero) of an USB device.

Related driver interfaces:

- **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN** (page 51)
- **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT** (page 52)

6.2.4 Read from Pipe to Output Window

This dialog page allows 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 48)

6.2.5 Read from Pipe to File

This dialog page allows 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 48)

6.2.6 Write from File to Pipe

This dialog page allows 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 48)

7 Installation Issues

This section discusses the topics related to installation of the USBIO device driver. Included is a description of how a customized driver setup can be built.

Important:
On Windows 2000 administrator rights are required to install a device driver. Because the USBIO driver is installed in the same way as any other Plug&Play device driver the installation requires administrator rights. Once the USBIO driver is installed standard user rights are sufficient to load the driver and to use the driver by accessing its programming interface.

7.1 Automated Installation: The USBIO Installation Wizard

Using the USBIO Installation Wizard is the quickest and easiest way for installing the USBIO device driver. This wizard performs the driver installation automatically in a step-by-step procedure. The device the USBIO driver will be installed for can be selected from a list. It is not necessary to manually edit or copy any files. After installation is complete the wizard allows to save the specific setup files that has been generated for the selected device. These files can be used at a later time to manually install the USBIO driver for the same device, without using the Installation Wizard.

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

- On Windows 2000 make sure you are logged on as an administrator or have enough privileges to install device drivers on the system. In general, special privileges are required to install device drivers on Windows 2000.

- Connect your USB device to the system. After plugging in the device Windows launches the New Hardware Wizard and prompts you for a device driver. Complete the New Hardware Wizard by clicking Next on each page and Finish on the last page. Windows either installs a system-provided driver or registers the device as "Unknown".

  Do not abort the New Hardware Wizard by clicking the Cancel button. This will prevent Windows from enumerating the device and storing enumeration information in the registry. As a result of this, the device is not visible in the system and the USBIO Installation Wizard is not able to install the driver for it.

  For some kinds of devices the system does not launch the New Hardware Wizard. A system-provided device driver will be installed silently. This will happen if the device belongs to a predefined device class, Human Interface Devices (HID), Audio Devices, or Printer Devices for example. The USBIO Installation Wizard is able to install the USBIO driver for such devices but this will disable any system-provided driver.

- 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 concerning the installation process. Click the Next button to continue. Note that you can abort the Installation Wizard at any time by clicking the Cancel button.

- On the next page the wizard shows a list containing all USB devices currently connected to the system. Select the device the USBIO driver shall be installed for. The Hardware 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. It is built from a bus identifier (USB), the 16-bit vendor ID (VID), the 16-bit product ID (PID), and optionally the revision code (REV). The IDs and the revision code are reported by the device in the USB Device Descriptor.

  If your device is not shown in the list make sure it is plugged in properly and you have finished 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 Manager can be accessed by right-clicking the "My Computer" icon on the desktop and then chosing Properties.

  Use the Refresh button to rescan for active devices 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 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, the device may no longer be used by the operating system and by applications. If the device belongs to the HID class, a mouse or a keyboard for example, this can cause problems.

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

  The USBIO Installation Wizard allows you to save the specific driver installation file (INF) that it generated for the device. The INF file is specific for the selected device because it contains the Hardware ID of that device. You can use the button labeled "Save INF file" to save the generated INF file with a name of your choice and in a location of your choice. The Installation Wizard copies also the USBIO driver binary file usbio.sys to the same location as the INF file. You can use these files at a later time to install the USBIO driver manually.

  You can use the button labeled "Run USBIO Application" to start the demo application that is included in the USBIO package. The application allows you to test several USB operations manually. Please refer to chapter 6 for further information.

  To quit the USBIO Installation Wizard, click Finish.

7.2 Manual Installation: The USBIO Setup Information File

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

An INF file is in ASCII text format. It can be viewed and modified with any text editor, Notepad for example. The contents and the syntax of an INF file are documented in the Microsoft Windows 2000 DDK.

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

The association of device and driver is based on a string that is called Hardware ID. The Plug&Play Manager builds the Hardware ID string from the 16-bit vendor ID (VID), the 16-bit product ID (PID), and optionally the revision code (REV). The string is pefixed by the bus identifier USB. Examples for Hardware ID strings are:

```
USB\VID_046D&PID_0100
USB\VID_046D&PID_C001&REV_0401
USB\CLASS_09&SUBCLASS_01&PROT_00
```

As shown in the last example a Hardware ID can also describe a device class and subclass. This makes it possible to provide a driver that will be used whenever the system detects a device that belongs to a specific device class. An example for such a kind of driver is the system-provided HID mouse driver. This driver is installed for any type of USB mouse, regardless of the vendor, the USB Vendor ID, and the USB Product ID. The driver selection is based on the class, subclass, and protocol identifiers. Please refer to the Microsoft Windows 2000 DDK for detailed information on Harware IDs and driver selection algorithms. Another good source of information are the INF files that ship with the operating system. They are located in a subdirectory of the Windows system directory, named "INF". Note that on Windows 2000 this subdirectory has a Hidden attribute by default.

In order to prepare an installation disk that can be used to install the USBIO driver for your device the following steps are required.

- Copy the USBIO driver binary usbio.sys to a floppy disk or to a directory location of your choice. Copy the INF file usbio.inf provided with the USBIO package to the same location. Note that you can choose any name for the INF file, based on your company name or your product name for example. But the file name extension has to be .inf. In the following discussion it is assumed the INF file is named usbio.inf.
- Open the usbio.inf file using a text editor, Notepad for example. Edit the **`[_Devices]`** section. There are various examples of Hardware ID strings prepared in this section. Select one of the examples that matches your needs. Usually, the very first example is appropriate. It associates the USBIO driver with your device by using the USB Vendor ID and Product ID. Remove the semi-colon at the start of the line and replace the `VID_XXXX` and `PID_XXXX` placeholders in the Hardware ID string by your USB Vendor ID and Product ID as shown in the examples above. Note that the IDs are given as 4-digit hexadecimal numbers.
- Edit the **`[Strings]`** section at the end of the usbio.inf file to modify the device description string for your device, defined by the value of **`S_DeviceDesc1`**. The device description text will be displayed in the Device Manager next to the icon that represents your device.
- Save the INF file to accommodate your changes.

Now you are prepared to start the driver installation. The required steps are described below.

- Connect your USB device to the system. After plugging in the device Windows launches the New Hardware Wizard and prompts you for a device driver. Provide the New Hardware Wizard with the location of your installation files (usbio.inf and usbio.sys). Complete the wizard by following the instructions shown on screen. If the INF file matches with your device the driver should be installed successfully.

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

- If the device belongs to a predefined device class that is supported by the operating system, the system does not launch the New Hardware Wizard after the device is plugged in. Instead of that a system-provided device driver will be installed silently. Human Interface Devices (HID) like mice and keyboards, Audio Devices, or Printer Devices are examples for such devices. The operating system does not ask for a driver because it finds a matching entry for the device's class and subclass ID in its internal INF file data base, as mentioned above.

  Use the Device Manager to install the USBIO driver for a device for that a driver is already running. To start the Device Manager choose Properties on the "My Computer" icon on the desktop. In the Device Manager locate your device and choose Properties on the entry. On the property page that pops up choose Driver and click the button labeled "Update Driver". The Upgrade Device Driver Wizard is started which is similar to the New Hardware Wizard mentioned above. Provide the wizard with the location of your installation files (usbio.inf and usbio.sys) and complete the driver installation by following the instructions shown on screen.

- For some device classes, especially HID devices like mice and keyboards, Windows does not allow you to install a driver with a different device class. That means you have to modify the device class entry in the **`[Version]`** section of the usbio.inf file to match with the device's class. The device class is specified by the keywords **`Class`** and **`ClassGUID`** in the **`[Version]`** section.

  For example, if you want to use a keyboard or a mouse to test the USBIO driver the new entries should be
  `Class=HIDClass` and
  `ClassGUID={745a17a0-74d3-11d0-b6fe-00a0c90f57da}`.

  The ClassGUID value that is associated with a device class can be found in system-provided INF files in `%WINDIR%\INF\` or in the Windows 2000 DDK documentation.

  Note that at least two drivers are used for USB keyboard and mouse devices. One belongs to the USB HID class and the other one belongs to the keyboard or mouse class. The keyboard or mouse driver runs on top of the USB HID driver. The USBIO driver can replace the USB HID driver only. In the Device Manager the HID driver is shown in a section labeled "Human Interface Devices". To be sure to replace the correct driver refer to the "Driver File Details" dialog in the Properties page of the entry. If the driver stack contains the file HIDUSB.SYS then you have selected the correct entry in the Device Manager.

- In the Device Manager the section "Universal Serial Bus controllers" contains an item labeled "USB Root Hub".

  Do not install USBIO for the USB Root Hub!

  The USB Root Hub is not an USB device. It is built into the USB host controller and is controlled by a special device driver provided by the operating system.

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

- In order 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 6 for detailed information on the Demo Application.

7.3 Uninstalling USBIO

In order to uninstall USBIO for a given device the Device Manager has to be used. The Device Manager can be accessed by right-clicking the "My Computer" icon on the desktop and then chosing Properties. In the Device Manager double-click on the entry of the device and choose the property page that is labeled "Driver". There are two options:

- Remove the device from the system by clicking the button "Uninstall". The operating system will reinstall a driver the next time the device is connected or the system is rebooted.

- Install a new driver for the device by clicking the button "Update Driver". The operating system launchs the Upgrade Device Driver Wizard which searchs for driver files or lets you select a driver.

In order to avoid that USBIO is reinstalled automatically and silently by the operating system it is necessary to manually remove the INF file that was used to install the USBIO driver.

During driver installation Windows stores a copy of the INF file in its internal INF file data base that is located in `%WINDIR%\INF\`. The original INF file is renamed and stored as `oemX.inf` for example, where X is a decimal number. The exact INF naming scheme depends on the operating system (Windows 2000 uses a slightly different scheme than Windows 98). The best way to find the correct INF file is to do a search for some significant string in all the INF files in the directory `%WINDIR%\INF\` and its subdirectories.

Note that on Windows 98 and Windows ME the INF file may also be stored in a directory named `%WINDIR%\INF\OTHER\`. Another naming scheme based on the provider name is used in that case.

Note also that on Windows 2000 the `%WINDIR%\INF\` directory has a Hidden attribute by default. Therefore, the directory is not shown in Windows Explorer by default.

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

7.4 Building a Customized Driver Setup

When the USBIO driver is included and shipped with a retail product some setup parameters should be customized. This is necessary because the USBIO device driver might be used by several vendors and it is possible that an user has two products and that both of them use the USBIO driver. This can cause conflicts with respect to the file name of the driver executable, the location of registry parameters, the device names, and the driver interface GUIDs used. To avoid such problems a vendor who redistributes the USBIO driver for use with a hardware product should choose a new file name for the driver binary, generate a private interface GUID, and select a private location in the registry to be used to store startup parameters. In order to do that the usbio.inf file has to be customized as well.

The following list shows the steps required to build a customized USBIO setup:

- Choose a new name for the driver binary file usbio.sys. The name should not cause conflicts with drivers provided by Windows. Rename the file usbio.sys to your new name.
- Rename the Setup Information file usbio.inf. You can choose any name you want. For instance, the name may be based on your companie's or your product's name. Note that the file extension should not be changed. It has to be ".inf".
- Edit the **`[_CopyFiles_sys]`** section in the INF file to include the new name of the driver binary.
- Edit the value **`S_DriverName`** in the **`[Strings]`** section to match with the new name you defined for the driver binary.
- Edit the **`[Strings]`** section in the INF file to modify text strings that are shown at the user interface level. You may change the following parameters:

  `S_Provider`
  `S_Mfg`
  `S_DeviceClassDisplayName`
  `S_DeviceDesc1`
  `S_DiskName`
  `S_ServiceDisplayName`
- Edit the following values in the **`[Strings]`** section to specify a location in the Registry that is used to store the USBIO driver's configuration parameters:

  `S_ConfigPath`
  `S_DeviceConfigPath1`

  Note that **`S_ConfigPath`** should specify a location that is a subkey of
  `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`.
  The name of the subkey should be the same as the name you choosed for the driver binary.
- Generate a private Globally Unique Identifier (GUID) to unambiguously identify the device instances that will be created by USBIO for your device. Use GUIDGEN.EXE from the Microsoft Platform SDK or from the Visual C++ package for this purpose. Copy the text representation of the GUID to the line in the INF file that defines the registry value **`USBIO_UserInterfaceGuid`**. Activate this line by removing the ";" at the beginning.

Use the private GUID in your application to search for available devices. GUIDGEN.EXE allows you to export a `static const struct GUID = {...}` statement that can be included in the source code of an applciation. For an example, refer to the source code of USBIOAPP or ReaderCpp.

- Edit the driver parameter settings in the sections **`_Parameters1_98`** and **`_Parameters1_NT`**. The parameters in **`_Parameters1_98`** define the default behaviour of the USBIO driver on Windows 98. The parameters in **`_Parameters1_NT`** define the default behaviour of the USBIO driver on Windows 2000. For a detailed description of the supported settings, refer to chapter 8.
- After you finished testing your INF file remove any lines and comments that are not needed. Especially, make sure that the word USBIO does not occur in the files you ship with your product. This is a requirement that is defined by the USBIO licensing conditions. See also the License Agreement you received with the USBIO package.

8 Registry Entries

The behaviour of the driver can be customized by startup parameters stored in the registry. The parameters are stored under a path that is specified in the INF file. This registry path is

`\HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\USBIO\Parameters`

by default.

The location can be customized by changing the **`S_ConfigPath`** and **`S_DeviceConfigPath1`** variables in the **`[Strings]`** section of the INF file.

The driver reads the parameters when a new device object is added. If a parameter does not exist when the driver attempts to read it, the driver creates the entry using an internal default value.

The following table lists all registry parameters.

Table 3: Registry parameters supported by the USBIO driver

Value	Min	Default	Max	Description
RequestTimeout	0	1000		Time-out interval for synchronous I/O requests, in milliseconds. Zero means infinite (no time-out).
ShortTransferOk	0	1	1	If set to 1 short packets in read transfers are allowed. If set to 0 short packets in read transfers cause errors.
UnconfigureOnClose	0	1	1	If set to 1 the device will be unconfigured when the last file handle is closed. If set to 0 the device state is not changed.
ResetDeviceOnClose	0	0	1	If set to 1 the device receives an USB reset if the last file handle is closed. If set to 0 the device state is not changed.
EnableRemoteWakeup	0	1	1	If set to 1 Remote Wakeup is enabled. If set to 0 Remote Wakeup is disabled.
MaxIsoPackets	16	64	512	Maximum number of packets allowed in an isochronous data transfer.
PowerStateOnOpen	0	0	3	Device power state that will be set when the device is opened (first handle is opened). 0...3 correspond to D0...D3

Table 3: (continued)

Value	Min	Default	Max	Description
PowerStateOnClose	0	3	3	Device power state that will be set when the device is closed (last handle is closed). 0. . . 3 correspond to D0. . . D3
MinPowerStateUsed	0	3	3	The minimum power state of the device while it is used (open handles exist). On system suspend the device is not allowed to go into states higher than this value. 0. . . 3 correspond to D0. . . D3 The value 0 (D0) means: no suspend allowed if the device is in use. The value 3 (D3) means: full suspend (off) allowed if the device is in use.
MinPowerStateUnused	0	3	3	The minimum power state of the device while it is not used (no open handles exist). On system suspend the device is not allowed to go into states higher than this value. 0. . . 3 correspond to D0. . . D3 The value 0 (D0) means: no suspend allowed if the device is not in use. The value 3 (D3) means: full suspend (off) allowed if the device is not in use.
AbortPipesOnPowerDown	0	0	1	Handling of outstanding read or write requests when the device goes into a suspend state (leaves D0): 1 = abort pending requests 0 = do not abort pending requests
SuppressPnPRemoveDlg	0	1	1	If this flag is set, Windows 2000 does not show a warning dialog if the device is removed.
DebugPort	0	0	3	Destination of trace messages for debugging purposes: 0 = kernel debugger or debug monitor 1. . . 3 = COM1. . . COM3 This parameter is available only if the debug (checked) build of the USBIO driver is used.

Table 3: (continued)

Value	Min	Default	Max	Description
DebugMask	0	3		Control of message output for debugging. This parameter is available only if the debug (checked) build of the USBIO driver is used.
DebugBaud	2.400	57.600	115.200	Baudrate selection for debug output to COM port. This parameter is available only if the debug (checked) build of the USBIO driver is used.

9 Related Documents

- Universal Serial Bus Specification 1.0, 1.1
- USB device class specifications (Audio, HID, Printer, etc.)
- Windows 2000 DDK Documentation
- Windows 98 DDK Documentation
- Microsoft Platform SDK Documentation

Index

IOCTL_USBIO_ABORT_PIPE, 48
IOCTL_USBIO_BIND_PIPE, 45
IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST, 34
IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST, 35
IOCTL_USBIO_CLEAR_FEATURE, 26
IOCTL_USBIO_CYCLE_PORT, 44
IOCTL_USBIO_GET_CONFIGURATION_INFO, 38
IOCTL_USBIO_GET_CONFIGURATION, 28
IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER, 40
IOCTL_USBIO_GET_DESCRIPTOR, 23
IOCTL_USBIO_GET_DEVICE_PARAMETERS, 36
IOCTL_USBIO_GET_DEVICE_POWER_STATE, 42
IOCTL_USBIO_GET_DRIVER_INFO, 43
IOCTL_USBIO_GET_INTERFACE, 29
IOCTL_USBIO_GET_PIPE_PARAMETERS, 49
IOCTL_USBIO_GET_STATUS, 27
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN, 51
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT, 52
IOCTL_USBIO_RESET_DEVICE, 39
IOCTL_USBIO_RESET_PIPE, 47
IOCTL_USBIO_SET_CONFIGURATION, 31
IOCTL_USBIO_SET_DESCRIPTOR, 24
IOCTL_USBIO_SET_DEVICE_PARAMETERS, 37
IOCTL_USBIO_SET_DEVICE_POWER_STATE, 41
IOCTL_USBIO_SET_FEATURE, 25
IOCTL_USBIO_SET_INTERFACE, 33
IOCTL_USBIO_SET_PIPE_PARAMETERS, 50
IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR, 30
IOCTL_USBIO_UNBIND_PIPE, 46
IOCTL_USBIO_UNCONFIGURE_DEVICE, 32

USBIO_BIND_PIPE, 77
USBIO_CLASS_OR_VENDOR_REQUEST, 67
USBIO_CONFIGURATION_INFO, 74
USBIO_DESCRIPTOR_REQUEST, 58
USBIO_DEVICE_PARAMETERS, 69
USBIO_DEVICE_POWER_STATE, 87
USBIO_DEVICE_POWER, 76
USBIO_DRIVER_INFO, 57
USBIO_FEATURE_REQUEST, 59
USBIO_FRAME_NUMBER, 75
USBIO_GET_CONFIGURATION_DATA, 62
USBIO_GET_INTERFACE_DATA, 64
USBIO_GET_INTERFACE, 63
USBIO_INTERFACE_CONFIGURATION_INFO, 70
USBIO_INTERFACE_SETTING, 65

USBIO_ISO_PACKET, 82
USBIO_ISO_TRANSFER_HEADER, 83
USBIO_ISO_TRANSFER, 80
USBIO_PIPE_CONFIGURATION_INFO, 72
USBIO_PIPE_CONTROL_TRANSFER, 79
USBIO_PIPE_PARAMETERS, 78
USBIO_PIPE_TYPE, 84
USBIO_REQUEST_RECIPIENT, 85
USBIO_REQUEST_TYPE, 86
USBIO_SET_CONFIGURATION, 66
USBIO_STATUS_REQUEST_DATA, 61
USBIO_STATUS_REQUEST, 60