Thesycon® Systemsoftware & Consulting GmbH

USBIO

USB Software Development Kit for Windows

Reference Manual

Version 2.0 **January 31, 2003**

Thesycon® Systemsoftware & Consulting GmbH

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

Tel: +49 3677 / 8462-0

Fax: +49 3677 / 8462-18

e-mail: USBIO@thesycon.de

http://www.thesycon.de

Disclaimer

Trademarks

The following trade names are referenced throughout this manual:

Microsoft, Windows, Win32, Windows NT, Windows XP, 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 **15**

1 Introduction **17**

2 Overview **19**

2.1 Platforms . . . 19

2.2 Features . . . 20

2.3 Restrictions . . . 21

2.4 USB 2.0 Support . . . 22

2.4.1 How to install USB 2.0 Host Controller Drivers on Windows 2000 . . . 23

2.4.2 How to install USB 2.0 Host Controller Drivers on Windows XP . . . 23

3 Architecture **25**

3.1 USBIO Object Model . . . 26

3.1.1 USBIO Device Objects . . . 26

3.1.2 USBIO Pipe Objects . . . 28

3.2 Establishing a Connection to the Device . . . 29

3.3 Power Management . . . 30

3.4 Device State Change Notifications . . . 31

4 Programming Interface **33**

4.1 Programming Interface Overview . . . 34

4.1.1 Query Information Requests . . . 34

4.1.2 Device-related Requests . . . 34

4.1.3 Pipe-related Requests . . . 36

4.1.4 Data Transfer Requests . . . 36

4.2 Control Requests . . . 37

IOCTL_USBIO_GET_DESCRIPTOR . . . 38

IOCTL_USBIO_SET_DESCRIPTOR . . . 39

IOCTL_USBIO_SET_FEATURE . . . 40

IOCTL_USBIO_CLEAR_FEATURE . . . 41

IOCTL_USBIO_GET_STATUS . . . 42

IOCTL_USBIO_GET_CONFIGURATION . . . 43

IOCTL_USBIO_GET_INTERFACE . . . 44

IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR . . . 45

IOCTL_USBIO_SET_CONFIGURATION 46
IOCTL_USBIO_UNCONFIGURE_DEVICE 47
IOCTL_USBIO_SET_INTERFACE 48
IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST 49
IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST 50
IOCTL_USBIO_GET_DEVICE_PARAMETERS 52
IOCTL_USBIO_SET_DEVICE_PARAMETERS 53
IOCTL_USBIO_GET_CONFIGURATION_INFO 54
IOCTL_USBIO_RESET_DEVICE 55
IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER 57
IOCTL_USBIO_SET_DEVICE_POWER_STATE 58
IOCTL_USBIO_GET_DEVICE_POWER_STATE 59
IOCTL_USBIO_GET_BANDWIDTH_INFO 60
IOCTL_USBIO_GET_DEVICE_INFO 61
IOCTL_USBIO_GET_DRIVER_INFO 62
IOCTL_USBIO_CYCLE_PORT 64
IOCTL_USBIO_BIND_PIPE 66
IOCTL_USBIO_UNBIND_PIPE 67
IOCTL_USBIO_RESET_PIPE 68
IOCTL_USBIO_ABORT_PIPE 69
IOCTL_USBIO_GET_PIPE_PARAMETERS 70
IOCTL_USBIO_SET_PIPE_PARAMETERS 71
IOCTL_USBIO_SETUP_PIPE_STATISTICS 72
IOCTL_USBIO_QUERY_PIPE_STATISTICS 74
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN 76
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT 77
4.3 Data Transfer Requests 79
4.3.1 Bulk and Interrupt Transfers 79
Bulk or Interrupt Write Transfers 79
Bulk or Interrupt Read Transfers 79
4.3.2 Isochronous Transfers 80
Isochronous Write Transfers 81
Isochronous Read Transfers 81
4.4 Data Structures 83
USBIO_BANDWIDTH_INFO 84

USBIO_DEVICE_INFO . . . 85
USBIO_DRIVER_INFO . . . 86
USBIO_DESCRIPTOR_REQUEST . . . 88
USBIO_FEATURE_REQUEST . . . 90
USBIO_STATUS_REQUEST . . . 91
USBIO_STATUS_REQUEST_DATA . . . 92
USBIO_GET_CONFIGURATION_DATA . . . 93
USBIO_GET_INTERFACE . . . 94
USBIO_GET_INTERFACE_DATA . . . 95
USBIO_INTERFACE_SETTING . . . 96
USBIO_SET_CONFIGURATION . . . 97
USBIO_CLASS_OR_VENDOR_REQUEST . . . 98
USBIO_DEVICE_PARAMETERS . . . 100
USBIO_INTERFACE_CONFIGURATION_INFO . . . 102
USBIO_PIPE_CONFIGURATION_INFO . . . 104
USBIO_CONFIGURATION_INFO . . . 106
USBIO_FRAME_NUMBER . . . 107
USBIO_DEVICE_POWER . . . 108
USBIO_BIND_PIPE . . . 109
USBIO_PIPE_PARAMETERS . . . 110
USBIO_SETUP_PIPE_STATISTICS . . . 111
USBIO_QUERY_PIPE_STATISTICS . . . 112
USBIO_PIPE_STATISTICS . . . 114
USBIO_PIPE_CONTROL_TRANSFER . . . 116
USBIO_ISO_TRANSFER . . . 117
USBIO_ISO_PACKET . . . 119
USBIO_ISO_TRANSFER_HEADER . . . 120
4.5 Enumeration Types . . . 121
USBIO_PIPE_TYPE . . . 121
USBIO_REQUEST_RECIPIENT . . . 122
USBIO_REQUEST_TYPE . . . 123
USBIO_DEVICE_POWER_STATE . . . 124
4.6 Error Codes . . . 125
USBIO_ERR_SUCCESS (0x00000000L) . . . 125
USBIO_ERR_CRC (0xE0000001L) . . . 125

USBIO_ERR_BTSTUFF (0xE0000002L) 125
USBIO_ERR_DATA_TOGGLE_MISMATCH (0xE0000003L) 125
USBIO_ERR_STALL_PID (0xE0000004L) 125
USBIO_ERR_DEV_NOT_RESPONDING (0xE0000005L) 125
USBIO_ERR_PID_CHECK_FAILURE (0xE0000006L) 125
USBIO_ERR_UNEXPECTED_PID (0xE0000007L) 125
USBIO_ERR_DATA_OVERRUN (0xE0000008L) 125
USBIO_ERR_DATA_UNDERRUN (0xE0000009L) 125
USBIO_ERR_RESERVED1 (0xE000000AL) 126
USBIO_ERR_RESERVED2 (0xE000000BL) 126
USBIO_ERR_BUFFER_OVERRUN (0xE000000CL) 126
USBIO_ERR_BUFFER_UNDERRUN (0xE000000DL) 126
USBIO_ERR_NOT_ACCESSED (0xE000000FL) 126
USBIO_ERR_FIFO (0xE0000010L) 126
USBIO_ERR_XACT_ERROR (0xE0000011L) 126
USBIO_ERR_BABBLE_DETECTED (0xE0000012L) 126
USBIO_ERR_DATA_BUFFER_ERROR (0xE0000013L) 126
USBIO_ERR_ENDPOINT_HALTED (0xE0000030L) 127
USBIO_ERR_NO_MEMORY (0xE0000100L) 127
USBIO_ERR_INVALID_URB_FUNCTION (0xE0000200L) 127
USBIO_ERR_INVALID_PARAMETER (0xE0000300L) 127
USBIO_ERR_ERROR_BUSY (0xE0000400L) 127
USBIO_ERR_REQUEST_FAILED (0xE0000500L) 127
USBIO_ERR_INVALID_PIPE_HANDLE (0xE0000600L) 127
USBIO_ERR_NO_BANDWIDTH (0xE0000700L) 127
USBIO_ERR_INTERNAL_HC_ERROR (0xE0000800L) 127
USBIO_ERR_ERROR_SHORT_TRANSFER (0xE0000900L) 128
USBIO_ERR_BAD_START_FRAME (0xE0000A00L) 128
USBIO_ERR_ISOCH_REQUEST_FAILED (0xE0000B00L) 128
USBIO_ERR_FRAME_CONTROL_OWNED (0xE0000C00L) 128
USBIO_ERR_FRAME_CONTROL_NOT_OWNED (0xE0000D00L) 128
USBIO_ERR_NOT_SUPPORTED (0xE0000E00L) 128
USBIO_ERR_INVALID_CONFIGURATION_DESCRIPTOR (0xE0000F00L) 128
USBIO_ERR_INSUFFICIENT_RESOURCES (0xE8001000L) 128

USBIO_ERR_SET_CONFIG_FAILED (0xE0002000L) 129
USBIO_ERR_USBD_BUFFER_TOO_SMALL (0xE0003000L) 129
USBIO_ERR_USBD_INTERFACE_NOT_FOUND (0xE0004000L) 129
USBIO_ERR_INVALID_PIPE_FLAGS (0xE0005000L) 129
USBIO_ERR_USBD_TIMEOUT (0xE0006000L) 129
USBIO_ERR_DEVICE_GONE (0xE0007000L) 129
USBIO_ERR_STATUS_NOT_MAPPED (0xE0008000L) 129
USBIO_ERR_CANCELED (0xE0010000L) 129
USBIO_ERR_ISO_NOT_ACCESSED_BY_HW (0xE0020000L) 129
USBIO_ERR_ISO_TD_ERROR (0xE0030000L) 130
USBIO_ERR_ISO_NA_LATE_USBPORT (0xE0040000L) 130
USBIO_ERR_ISO_NOT_ACCESSED_LATE (0xE0050000L) 130
USBIO_ERR_FAILED (0xE0001000L) 130
USBIO_ERR_INVALID_INBUFFER (0xE0001001L) 130
USBIO_ERR_INVALID_OUTBUFFER (0xE0001002L) 130
USBIO_ERR_OUT_OF_MEMORY (0xE0001003L) 130
USBIO_ERR_PENDING_REQUESTS (0xE0001004L) 130
USBIO_ERR_ALREADY_CONFIGURED (0xE0001005L) 131
USBIO_ERR_NOT_CONFIGURED (0xE0001006L) 131
USBIO_ERR_OPEN_PIPES (0xE0001007L) 131
USBIO_ERR_ALREADY_BOUND (0xE0001008L) 131
USBIO_ERR_NOT_BOUND (0xE0001009L) 131
USBIO_ERR_DEVICE_NOT_PRESENT (0xE000100AL) 131
USBIO_ERR_CONTROL_NOT_SUPPORTED (0xE000100BL) 131
USBIO_ERR_TIMEOUT (0xE000100CL) 131
USBIO_ERR_INVALID_RECIPIENT (0xE000100DL) 132
USBIO_ERR_INVALID_TYPE (0xE000100EL) 132
USBIO_ERR_INVALID_IOCTL (0xE000100FL) 132
USBIO_ERR_INVALID_DIRECTION (0xE0001010L) 132
USBIO_ERR_TOO_MUCH_ISO_PACKETS (0xE0001011L) 132
USBIO_ERR_POOL_EMPTY (0xE0001012L) 132
USBIO_ERR_PIPE_NOT_FOUND (0xE0001013L) 132
USBIO_ERR_INVALID_ISO_PACKET (0xE0001014L) 132
USBIO_ERR_OUT_OF_ADDRESS_SPACE (0xE0001015L) 133
USBIO_ERR_INTERFACE_NOT_FOUND (0xE0001016L) 133

USBIO_ERR_INVALID_DEVICE_STATE (0xE0001017L) 133
USBIO_ERR_INVALID_PARAM (0xE0001018L) 133
USBIO_ERR_DEMO_EXPIRED (0xE0001019L) 133
USBIO_ERR_INVALID_POWER_STATE (0xE000101AL) 133
USBIO_ERR_POWER_DOWN (0xE000101BL) 133
USBIO_ERR_VERSION_MISMATCH (0xE000101CL) 134
USBIO_ERR_SET_CONFIGURATION_FAILED (0xE000101DL) 134
USBIO_ERR_VID_RESTRICTION (0xE0001080L) 134
USBIO_ERR_ISO_RESTRICTION (0xE0001081L) 134
USBIO_ERR_BULK_RESTRICTION (0xE0001082L) 134
USBIO_ERR_EP0_RESTRICTION (0xE0001083L) 134
USBIO_ERR_PIPE_RESTRICTION (0xE0001084L) 134
USBIO_ERR_PIPE_SIZE_RESTRICTION (0xE0001085L) 135
USBIO_ERR_CONTROL_RESTRICTION (0xE0001086L) 135
USBIO_ERR_INTERRUPT_RESTRICTION (0xE0001087L) 135
USBIO_ERR_DEVICE_NOT_FOUND (0xE0001100L) 135
USBIO_ERR_DEVICE_NOT_OPEN (0xE0001102L) 135
USBIO_ERR_NO_SUCH_DEVICE_INSTANCE (0xE0001104L) 135
USBIO_ERR_INVALID_FUNCTION_PARAM (0xE0001105L) 135
USBIO_ERR_LOAD_SETUP_API_FAILED (0xE0001106L) 135

5 USBIO Class Library **137**

5.1 Overview 137
5.1.1 CUsbIo Class 137
5.1.2 CUsbIoPipe Class 137
5.1.3 CUsbIoThread Class 138
5.1.4 CUsbIoReader Class 138
5.1.5 CUsbIoWriter Class 139
5.1.6 CUsbIoBuf Class 139
5.1.7 CUsbIoBufPool Class 139
5.2 Class Library Reference 140
CUsbIo class 140
Member Functions 140
CUsbIo 140
~CUsbIo 140

CreateDeviceList 141
DestroyDeviceList 142
Open 143
Close 145
GetDeviceInstanceDetails 146
GetDevicePathName 148
IsOpen 149
IsCheckedBuild 150
IsDemoVersion 151
IsLightVersion 152
IsOperatingAtHighSpeed 153
GetDriverInfo 154
GetDeviceInfo 155
GetBandwidthInfo 156
GetDescriptor 157
GetDeviceDescriptor 159
GetConfigurationDescriptor 160
GetStringDescriptor 162
SetDescriptor 164
SetFeature 166
ClearFeature 167
GetStatus 168
ClassOrVendorInRequest 169
ClassOrVendorOutRequest 170
SetConfiguration 171
UnconfigureDevice 172
GetConfiguration 173
GetConfigurationInfo 174
SetInterface 175
GetInterface 176
StoreConfigurationDescriptor 177
GetDeviceParameters 178
SetDeviceParameters 179
ResetDevice 180
CyclePort 181

GetCurrentFrameNumber . . . 182
GetDevicePowerState . . . 183
SetDevicePowerState . . . 184
CancelIo . . . 185
IoctlSync . . . 186
ErrorText . . . 187
Data Members . . . 188
CUsbIoPipe class . . . 189
Member Functions . . . 189
CUsbIoPipe . . . 189
~CUsbIoPipe . . . 189
Bind . . . 190
Unbind . . . 192
Read . . . 193
Write . . . 194
WaitForCompletion . . . 195
ReadSync . . . 197
WriteSync . . . 199
ResetPipe . . . 201
AbortPipe . . . 202
GetPipeParameters . . . 203
SetPipeParameters . . . 204
PipeControlTransferIn . . . 205
PipeControlTransferOut . . . 207
SetupPipeStatistics . . . 209
QueryPipeStatistics . . . 210
ResetPipeStatistics . . . 212
CUsbIoThread class . . . 213
Member Functions . . . 213
CUsbIoThread . . . 213
~CUsbIoThread . . . 213
AllocateBuffers . . . 214
FreeBuffers . . . 215
StartThread . . . 216
ShutdownThread . . . 217

ProcessData . . . 218
ProcessBuffer . . . 219
BufErrorHandler . . . 220
OnThreadExit . . . 221
ThreadRoutine . . . 222
TerminateThread . . . 223
Data Members . . . 224
CUsbIoReader class . . . 225
Member Functions . . . 225
CUsbIoReader . . . 225
~CUsbIoReader . . . 225
ThreadRoutine . . . 226
TerminateThread . . . 227
CUsbIoWriter class . . . 228
Member Functions . . . 228
CUsbIoWriter . . . 228
~CUsbIoWriter . . . 228
ThreadRoutine . . . 229
TerminateThread . . . 230
CUsbIoBuf class . . . 231
Member Functions . . . 231
CUsbIoBuf . . . 231
CUsbIoBuf . . . 232
CUsbIoBuf . . . 233
~CUsbIoBuf . . . 234
Buffer . . . 235
Size . . . 236
Data Members . . . 237
CUsbIoBufPool class . . . 239
Member Functions . . . 239
CUsbIoBufPool . . . 240
~CUsbIoBufPool . . . 240
Allocate . . . 241
Free . . . 242
Get . . . 243

Put . . . 244
CurrentCount . . . 245
Data Members . . . 246
CSetupApiDll class . . . 247
Member Functions . . . 247
CSetupApiDll . . . 247
~CSetupApiDll . . . 247
Load . . . 248
Release . . . 249

6 USBIO Demo Application 251
6.1 Dialog Pages for Device Operations . . . 251
6.1.1 Device . . . 251
6.1.2 Descriptors . . . 251
6.1.3 Configuration . . . 251
6.1.4 Interface . . . 252
6.1.5 Pipes . . . 252
6.1.6 Class or Vendor Request . . . 252
6.1.7 Feature . . . 252
6.1.8 Other . . . 253
6.2 Dialog Pages for Pipe Operations . . . 253
6.2.1 Pipe . . . 253
6.2.2 Buffers . . . 253
6.2.3 Control . . . 254
6.2.4 Read from Pipe to Output Window . . . 254
6.2.5 Read from Pipe to File . . . 254
6.2.6 Write from File to Pipe . . . 254

7 Driver Installation and Uninstallation 255
7.1 USBIO Driver Executables . . . 255
7.2 Installing USBIO . . . 255
7.2.1 Automated Installation: The USBIO Installation Wizard . . . 255
7.2.2 Manual Installation: The USBIO Setup Information File . . . 257
7.3 Uninstalling USBIO . . . 260
7.3.1 Manual Uninstallation . . . 260
7.3.2 Automated Uninstallation: The USBIO Cleanup Wizard . . . 260

7.4 Building a Customized Driver Setup 262
7.5 Using USBIO on Windows XP Embedded 264

8 Registry Entries **265**

9 Related Documents **269**

Index **271**

1 Introduction

USBIO is a generic Universal Serial Bus (USB) device driver for Windows. It is able to control any type of USB device and provides a convenient programming interface that can be used by Win32 applications. The USBIO device driver supports USB 1.1 and USB 2.0.

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

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

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

2 Overview

Support for the Universal Serial Bus (USB) is built into the current Windows 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 a 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, Windows 2000, and Windows XP. 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.

2.1 Platforms

The USBIO driver supports the following operating system platforms:

- Windows 98 Second Edition (SE), the second release of Windows 98 (USB 1.1 only)
- Windows Millennium (ME), the successor to Windows 98 (USB 1.1 only)
- Windows 2000 with Service Pack 3 (USB 1.1 and USB 2.0)
- Windows XP with Service Pack 1 (USB 1.1 and USB 2.0)
- Windows XP Embedded (USB 1.1 only)

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

2.2 Features

The USBIO driver provides the following features:

- Supports USB 1.1 and USB 2.0
- 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

2.3 Restrictions

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

- If a particular kernel mode interface (e.g. WDM Kernel Mode Streaming or NDIS) has to be supported in order to integrate the device into the operating system, it is not possible to use the generic USBIO driver. However, in such a situation it is possible to develop a custom device driver based on the source code of the USBIO though. Please contact Thesycon if you need support on such kind of project.

- Although the USBIO device driver fully supports isochronous endpoints, there are some limitations with respect to isochronous data transfers. They result from the fact that the processing of 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.

- There are some problems caused by the implementation of the operating system's USB driver stack. Thesycon encountered these problems during debugging and testing of the USBIO driver. For some of the problems there are work-arounds built into the USBIO driver. Others do still exist because there is no way to implement a work-around.

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

- There are a lot of serious problems with the USB driver stack on Windows XP. Refer to *Problems.txt* to find a description of the problems that are known so far.

2.4 USB 2.0 Support

The USBIO device driver supports USB 2.0 and the Enhanced Host Controller on Windows 2000 and Windows XP. However, USBIO has to be used on top of the driver stack that is provided by Microsoft. Thesycon does not guarantee that the USBIO driver works in conjunction with a USB driver stack that is provided by a third party. For instance, third-party drivers are available for USB 2.0 host controllers from NEC or VIA. Because the Enhanced Host Controller hardware interface is standardized (EHCI specification) the USB 2.0 drivers provided by Microsoft can be used with host controllers from any vendor. However, the user has to ensure that these drivers are installed. In some cases this requires manual installation of the host controller driver. This will be discussed in more detail in the next sections.

Note that USBIO does not support USB 2.0 on Windows 98 and Windows ME. Microsoft does not provide USB 2.0 host controller drivers for these systems. Any third-party host controller drivers that may be available for Windows 98 and Windows ME are not supported by USBIO.

The USBIO driver is tested with Microsoft's driver stack on Windows 2000 and Windows XP on various host controllers. Table 1 summarizes the versions of the driver stack components that have been used for testing.

Table 1: Supported USB driver stack components

	Windows 2000 Service Pack 3	Windows XP Service Pack 1
usbehci.sys	5.00.2195.5652	5.1.2600.1106 (xpsp1.020828-1920)
usbport.sys	5.00.2195.5652	5.1.2600.1106 (xpsp1.020828-1920)
usbhub20.sys	5.00.2195.5605	—
usbhub.sys	5.00.2195.4413	5.1.2600.1106 (xpsp1.020828-1920)
usbd.sys	5.00.2195.4943	5.1.2600.0 (XPClient.010817-1148)

The drivers for the USB 2.0 host controller are not included in the original releases of Windows 2000 and Windows XP. They can be obtained by means of the Windows Update service or by installing a service pack. The following sections describe the driver installation procedure for each system in detail.

2.4.1 How to install USB 2.0 Host Controller Drivers on Windows 2000

The device drivers for the Enhanced Host Controller are available at the Windows Update server. To install the drivers, follow the steps described below.

- Install Service Pack 3 on the computer. Alternatively, you can launch the Windows Update service available in the Start menu.
- Open the Device Manager by choosing Manage from the context menu of the My Computer icon. In Device Manager locate the item that represents the USB 2.0 Enhanced Host Controller. If there is no driver currently installed for the host controller the item is in the group Other Devices and is labeled with a yellow exclamation mark. If there is already a driver installed for the host controller, a third-party driver for instance, then the item is located in the group Universal Serial Bus controllers.
- Open the property page of the Enhanced Host Controller item and select the Driver page. Choose Driver Details to display detailed information on the drivers that are currently in use. If the driver version is not correct, select Update Driver. This will launch the Upgrade Device Driver Wizard.
- Follow the instructions shown by the Upgrade Device Driver Wizard. The wizard prompts you to specify locations to be searched for new driver files. Make sure the location 'Microsoft Windows Update' is the only selection and continue.
- The wizard downloads the required files and installs the driver. If there was a driver already installed the wizard offers an option that allows you to select the new driver from a list. Make sure you select the driver provided by Microsoft.
- Open the property page again and verify that the correct drivers are installed now.

2.4.2 How to install USB 2.0 Host Controller Drivers on Windows XP

The device drivers for the Enhanced Host Controller are available at the Windows Update server and are included in the Service Pack 1 for Windows XP. To install the drivers, follow the steps described below.

- Install Service Pack 1 on the computer. Alternatively, you can launch the Windows Update service available in the Start menu.
- Open the Device Manager by choosing Manage from the context menu of the My Computer item in the Start menu. In Device Manager locate the item that represents the USB 2.0 Enhanced Host Controller. If there is no driver currently installed for the host controller the item is in the group Other Devices and is labeled with a yellow exclamation mark. If there is already a driver installed for the host controller, a third-party driver for instance, then the item is located in the group Universal Serial Bus controllers.
- Open the property page of the Enhanced Host Controller item and select the Driver page. Choose Driver Details to display detailed information on the drivers that are currently in use. If the driver version is not correct, select Update Driver. This will launch the Hardware Update Wizard.

- Follow the instructions shown by the Hardware Update Wizard. Normally, the wizard will install the correct drivers automatically. If the wizard prompts you to select the new driver from a list, make sure you select the driver provided by Microsoft.
- If there was a driver already installed for the host controller the Hardware Update Wizard possibly selects the existing driver again and does not upgrade to a new driver. In this case you should manually select the driver to be installed. To do so, select 'Install from a list or specific location' on the first dialog shown by the wizard. On the next dialog select 'Don't search'. In the next step, the wizard allows you to select the new driver from a list. Make sure you select the driver provided by Microsoft.
- Open the property page again and verify that the correct drivers are installed now.

3 Architecture

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

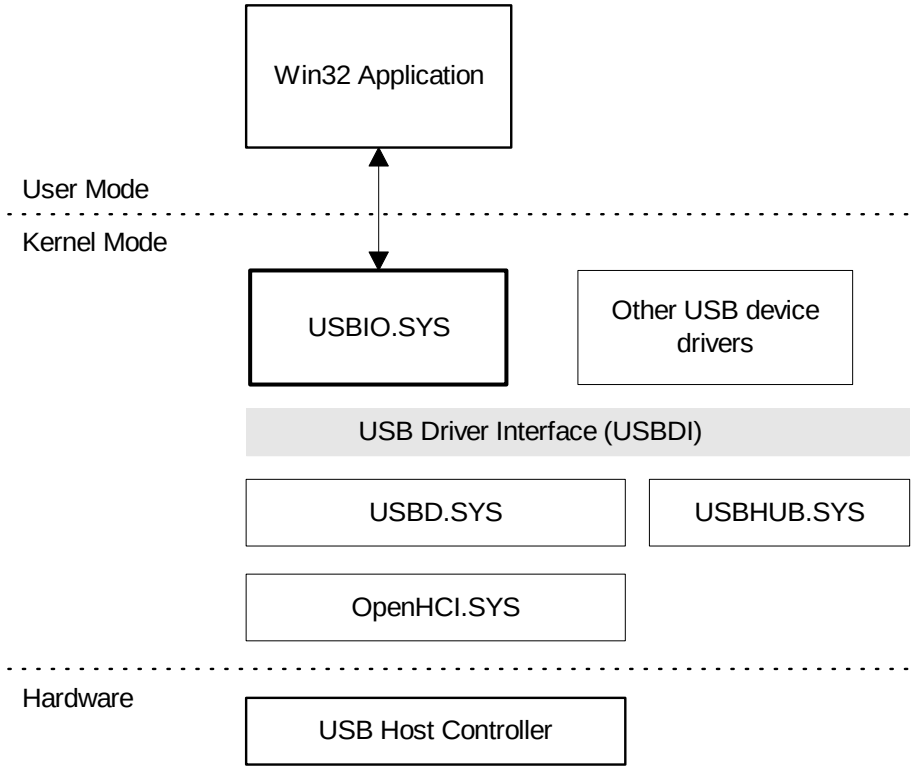

Figure 1: USB Driver Stack

The following modules are shown in Figure 1:

- USB Host Controller is the hardware component that controls the Universal Serial Bus. It also contains the USB Root Hub. There are two implementations of the host controller that support USB 1.1: Open Host Controller (OHC) and Universal Host Controller (UHC). There is one implementation of the host controller that supports USB 2.0: Enhanced Host Controller (EHC).
- OpenHCI.SYS is the host controller driver for controllers that conform with the Open Host Controller Interface specification. Optionally, it can be replaced by a driver for a controller that is compliant with UHCI (Universal Host Controller Interface) or EHCI (Enhanced Host Controller Interface). Which driver is used depends on the mainboard chip set of the computer. For instance, Intel chipsets contain Enhanced Host Controllers and Universal Host Controllers.
- USBD.SYS is the USB Bus Driver that controls and manages all devices connected to the USB. It is provided by Microsoft as part of the operating system.
- USBHUB.SYS is the USB Hub Driver. It is responsible for managing and controlling USB Hubs.

- USBIO.SYS is the generic USB device driver USBIO.

The software interface that is provided by the operating system for use by USB device drivers is called USB Driver Interface (USBDI). It is exported by the USBD at the top of the driver stack. USBDI is an IRP-based interface. This means that each individual request is packaged into an I/O request packet (IRP), a data structure that is defined by 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 for many of the functions there is no 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. The device may support USB 1.1, USB 2.0 or both. A device object is created by the USBIO driver if a USB is detected by the Plug&Play Manager of the operating system. This happens when a USB device is connected to the system. The USBIO driver is able to handle multiple device objects at the same time.

Each device object created by USBIO is registered with the operating system by using a unique identifier (GUID, Globally Unique Identifier). This identifier is called "Device Interface ID". All device objects managed by USBIO are identified by the same GUID. The GUID is defined in the USBIO Setup Information (INF) file. Based on the GUID and an instance number, the operating system generates a unique name for each device object. This name should 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, a 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 – a default interface and a user-defined interface – are registered for each device. The default Device Interface ID should only be used by the USBIO demo application. Custom applications should always use a private user-defined Device Interface ID. This way, device naming conflicts are avoided.

Important:
Every USBIO customer should generate 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 unambiguously identified.

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/XP 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.

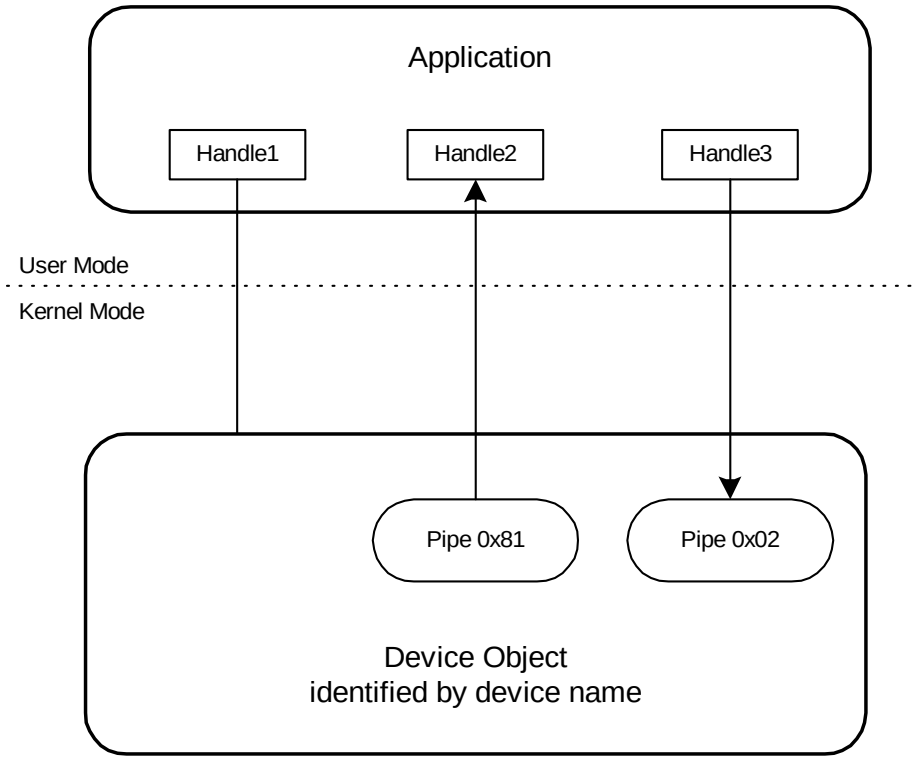

Figure 2: USBIO device and pipe objects example

The device object is identified by a device name as described in section 3.1.1 (page 26). 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 use of the USBIO programming interface are included in the USBIO Development Kit.

3.3 Power Management

Current Windows operating systems support system-level power management. That means that if the computer is idle for a given time, some parts of the computer can go into a 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. A 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 transparently 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/XP 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 (page 265).

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 a USB device changes. The Win32 API provides the function **`RegisterDeviceNotification()`** for this purpose. This way, an application will be notified if a USB device is plugged in or removed.

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

The device notification mechanism is only available if the USBIO device naming scheme is based on Device Interface IDs (GUIDs). See section 3.1.1 (page 26) 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

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

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

4.1 Programming Interface Overview

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

4.1.1 Query Information Requests

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

4.1.2 Device-related Requests

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

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

4.1.3 Pipe-related Requests

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

4.1.4 Data Transfer Requests

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

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.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_DESCRIPTOR for this operation.

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

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

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

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

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

Comments

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

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

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

See Also

USBIO_DESCRIPTOR_REQUEST (page 88)
IOCTL_USBIO_SET_DESCRIPTOR (page 39)

IOCTL_USBIO_SET_DESCRIPTOR

The IOCTL_USBIO_SET_DESCRIPTOR operation sets a specific descriptor of the device.

`dwIoControlCode`
Set to IOCTL_USBIO_SET_DESCRIPTOR for this operation.

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

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

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

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

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

Comments

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

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

See Also

USBIO_DESCRIPTOR_REQUEST (page 88)
IOCTL_USBIO_GET_DESCRIPTOR (page 38)

IOCTL_USBIO_SET_FEATURE

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

`dwIoControlCode`
Set to IOCTL_USBIO_SET_FEATURE for this operation.

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

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

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

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

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

Comments

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

See Also

USBIO_FEATURE_REQUEST (page 90)
IOCTL_USBIO_CLEAR_FEATURE (page 41)

IOCTL_USBIO_CLEAR_FEATURE

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

`dwIoControlCode`
Set to IOCTL_USBIO_CLEAR_FEATURE for this operation.

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

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

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

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

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

Comments

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

See Also

USBIO_FEATURE_REQUEST (page 90)
IOCTL_USBIO_SET_FEATURE (page 40)

IOCTL_USBIO_GET_STATUS

The IOCTL_USBIO_GET_STATUS operation requests status information for a specific recipient.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_STATUS for this operation.

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

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

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

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

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

Comments

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

See Also

USBIO_STATUS_REQUEST (page 91)
USBIO_STATUS_REQUEST_DATA (page 92)

IOCTL_USBIO_GET_CONFIGURATION

The IOCTL_USBIO_GET_CONFIGURATION operation retrieves the current configuration of the device.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_CONFIGURATION for this operation.

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

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

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

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

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

Comments

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

See Also

USBIO_GET_CONFIGURATION_DATA (page 93)
IOCTL_USBIO_GET_INTERFACE (page 44)
IOCTL_USBIO_SET_CONFIGURATION (page 46)

IOCTL_USBIO_GET_INTERFACE

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

`dwIoControlCode`
Set to IOCTL_USBIO_GET_INTERFACE for this operation.

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

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

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

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

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

Comments

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

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

See Also

USBIO_GET_INTERFACE (page 94)
USBIO_GET_INTERFACE_DATA (page 95)
IOCTL_USBIO_SET_CONFIGURATION (page 46)
IOCTL_USBIO_SET_INTERFACE (page 48)

IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR

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

`dwIoControlCode`
Set to IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR for this operation.

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

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

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

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

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

Comments

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

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

See Also

IOCTL_USBIO_SET_CONFIGURATION (page 46)

IOCTL_USBIO_SET_CONFIGURATION

The IOCTL_USBIO_SET_CONFIGURATION operation is used to set the device configuration.

`dwIoControlCode`
Set to IOCTL_USBIO_SET_CONFIGURATION for this operation.

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

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

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

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

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

Comments

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

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

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

See Also

USBIO_SET_CONFIGURATION (page 97)
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 54)
IOCTL_USBIO_UNCONFIGURE_DEVICE (page 47)
IOCTL_USBIO_GET_CONFIGURATION (page 43)
IOCTL_USBIO_GET_INTERFACE (page 44)
IOCTL_USBIO_SET_INTERFACE (page 48)

IOCTL_USBIO_UNCONFIGURE_DEVICE

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

`dwIoControlCode`
Set to IOCTL_USBIO_UNCONFIGURE_DEVICE for this operation.

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

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

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

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

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

Comments

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

See Also

IOCTL_USBIO_SET_CONFIGURATION (page 46)
IOCTL_USBIO_GET_CONFIGURATION (page 43)
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 54)
IOCTL_USBIO_GET_INTERFACE (page 44)
IOCTL_USBIO_SET_INTERFACE (page 48)

IOCTL_USBIO_SET_INTERFACE

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

`dwIoControlCode`
Set to IOCTL_USBIO_SET_INTERFACE for this operation.

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

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

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

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

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

Comments

A SET_INTERFACE request appears on the bus.

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

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

See Also

USBIO_INTERFACE_SETTING (page 96)
IOCTL_USBIO_GET_INTERFACE (page 44)
IOCTL_USBIO_SET_CONFIGURATION (page 46)
IOCTL_USBIO_GET_CONFIGURATION (page 43)
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 54)

IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST

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

`dwIoControlCode`
Set to IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST for this operation.

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

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

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

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

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

Comments

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

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

See Also

USBIO_CLASS_OR_VENDOR_REQUEST (page 98)
IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST (page 50)

IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST

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

`dwIoControlCode`
Set to IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST for this operation.

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

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

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

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

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

Comments

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

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

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

See Also

USBIO_CLASS_OR_VENDOR_REQUEST (page 98)
IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST (page 49)

IOCTL_USBIO_GET_DEVICE_PARAMETERS

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

`dwIoControlCode`
Set to IOCTL_USBIO_GET_DEVICE_PARAMETERS for this operation.

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

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

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

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

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

Comments

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

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

See Also

USBIO_DEVICE_PARAMETERS (page 100)
IOCTL_USBIO_SET_DEVICE_PARAMETERS (page 53)
IOCTL_USBIO_GET_PIPE_PARAMETERS (page 70)
IOCTL_USBIO_SET_PIPE_PARAMETERS (page 71)

IOCTL_USBIO_SET_DEVICE_PARAMETERS

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

`dwIoControlCode`
Set to IOCTL_USBIO_SET_DEVICE_PARAMETERS for this operation.

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

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

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

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

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

Comments

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

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

See Also

USBIO_DEVICE_PARAMETERS (page 100)
IOCTL_USBIO_GET_DEVICE_PARAMETERS (page 52)
IOCTL_USBIO_GET_PIPE_PARAMETERS (page 70)
IOCTL_USBIO_SET_PIPE_PARAMETERS (page 71)

IOCTL_USBIO_GET_CONFIGURATION_INFO

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

`dwIoControlCode`
Set to IOCTL_USBIO_GET_CONFIGURATION_INFO for this operation.

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

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

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

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

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

Comments

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

See Also

USBIO_CONFIGURATION_INFO (page 106)
IOCTL_USBIO_SET_CONFIGURATION (page 46)
IOCTL_USBIO_GET_CONFIGURATION (page 43)
IOCTL_USBIO_SET_INTERFACE (page 48)
IOCTL_USBIO_GET_INTERFACE (page 44)

IOCTL_USBIO_RESET_DEVICE

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

`dwIoControlCode`
Set to IOCTL_USBIO_RESET_DEVICE for this operation.

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

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

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

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

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

Comments

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

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

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

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

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

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

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

See Also

IOCTL_USBIO_SET_CONFIGURATION (page 46)
IOCTL_USBIO_CYCLE_PORT (page 64)

IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER

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

`dwIoControlCode`
Set to IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER for this operation.

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

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

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

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

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

Comments

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

See Also

USBIO_FRAME_NUMBER (page 107)

IOCTL_USBIO_SET_DEVICE_POWER_STATE

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

`dwIoControlCode`
Set to IOCTL_USBIO_SET_DEVICE_POWER_STATE for this operation.

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

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

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

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

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

Comments

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

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

See also the sections 3.3 (page 30) and the description of the data structure **USBIO_DEVICE_POWER** for more information on power management.

See Also

USBIO_DEVICE_POWER (page 108)
IOCTL_USBIO_GET_DEVICE_POWER_STATE (page 59)

IOCTL_USBIO_GET_DEVICE_POWER_STATE

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

`dwIoControlCode`
Set to IOCTL_USBIO_GET_DEVICE_POWER_STATE for this operation.

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

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

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

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

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

Comments

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

See also the sections 3.3 (page 30) and the description of the data structure **USBIO_DEVICE_POWER** for more information on power management.

See Also

USBIO_DEVICE_POWER (page 108)
IOCTL_USBIO_SET_DEVICE_POWER_STATE (page 58)

IOCTL_USBIO_GET_BANDWIDTH_INFO

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

`dwIoControlCode`
Set to IOCTL_USBIO_GET_BANDWIDTH_INFO for this operation.

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

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

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

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

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

Comments

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

See Also

USBIO_BANDWIDTH_INFO (page 84)
IOCTL_USBIO_GET_DEVICE_INFO (page 61)

IOCTL_USBIO_GET_DEVICE_INFO

The IOCTL_USBIO_GET_DEVICE_INFO request returns information on the USB device.

`dwIoControlCode`
Set to IOCTL_USBIO_GET_DEVICE_INFO for this operation.

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

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

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

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

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

Comments

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

See Also

USBIO_DEVICE_INFO (page 85)
IOCTL_USBIO_GET_BANDWIDTH_INFO (page 60)

IOCTL_USBIO_GET_DRIVER_INFO

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

`dwIoControlCode`
Set to IOCTL_USBIO_GET_DRIVER_INFO for this operation.

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

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

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

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

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

Comments

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

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

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

See Also

USBIO_DRIVER_INFO (page 86)

IOCTL_USBIO_GET_DEVICE_INFO (page 61)
IOCTL_USBIO_GET_BANDWIDTH_INFO (page 60)

IOCTL_USBIO_CYCLE_PORT

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

`dwIoControlCode`
Set to IOCTL_USBIO_CYCLE_PORT for this operation.

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

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

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

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

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

Comments

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

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

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

This request should be used instead of **IOCTL_USBIO_RESET_DEVICE** if the USB device modifies its descriptors during a USB Reset. Particularly, this is required to

implement the Device Firmware Upgrade (DFU) device class specification. Note that the USB device receives two USB Resets after this call. This does not conform to the DFU specification. However, this is the standard device enumeration method used by the Windows USB bus driver (USBD).

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

See Also

IOCTL_USBIO_RESET_DEVICE (page 55)

IOCTL_USBIO_BIND_PIPE

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

`dwIoControlCode`
Set to IOCTL_USBIO_BIND_PIPE for this operation.

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

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

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

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

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

Comments

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

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

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

See Also

USBIO_BIND_PIPE (page 109)
IOCTL_USBIO_UNBIND_PIPE (page 67)
IOCTL_USBIO_SET_CONFIGURATION (page 46)
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 54)

IOCTL_USBIO_UNBIND_PIPE

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

`dwIoControlCode`
Set to IOCTL_USBIO_UNBIND_PIPE for this operation.

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

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

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

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

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

Comments

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

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

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

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

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

See Also

IOCTL_USBIO_BIND_PIPE (page 66)
IOCTL_USBIO_SETUP_PIPE_STATISTICS (page 72)

IOCTL_USBIO_RESET_PIPE

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

`dwIoControlCode`
Set to IOCTL_USBIO_RESET_PIPE for this operation.

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

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

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

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

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

Comments

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

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

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

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

See Also

IOCTL_USBIO_ABORT_PIPE (page 69)
IOCTL_USBIO_BIND_PIPE (page 66)

IOCTL_USBIO_ABORT_PIPE

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

`dwIoControlCode`
Set to IOCTL_USBIO_ABORT_PIPE for this operation.

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

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

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

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

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

Comments

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

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

See Also

IOCTL_USBIO_RESET_PIPE (page 68)
IOCTL_USBIO_BIND_PIPE (page 66)

IOCTL_USBIO_GET_PIPE_PARAMETERS

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

`dwIoControlCode`
Set to IOCTL_USBIO_GET_PIPE_PARAMETERS for this operation.

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

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

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

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

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

Comments

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

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

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

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

See Also

USBIO_PIPE_PARAMETERS (page 110)
IOCTL_USBIO_SET_PIPE_PARAMETERS (page 71)
IOCTL_USBIO_GET_DEVICE_PARAMETERS (page 52)
IOCTL_USBIO_SET_DEVICE_PARAMETERS (page 53)
IOCTL_USBIO_BIND_PIPE (page 66)

IOCTL_USBIO_SET_PIPE_PARAMETERS

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

`dwIoControlCode`
Set to IOCTL_USBIO_SET_PIPE_PARAMETERS for this operation.

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

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

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

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

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

Comments

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

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

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

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

See Also

USBIO_PIPE_PARAMETERS (page 110)
IOCTL_USBIO_GET_PIPE_PARAMETERS (page 70)
IOCTL_USBIO_GET_DEVICE_PARAMETERS (page 52)
IOCTL_USBIO_SET_DEVICE_PARAMETERS (page 53)
IOCTL_USBIO_BIND_PIPE (page 66)

IOCTL_USBIO_SETUP_PIPE_STATISTICS

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

`dwIoControlCode`
Set to IOCTL_USBIO_SETUP_PIPE_STATISTICS for this operation.

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

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

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

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

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

Comments

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

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

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

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

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

See Also

USBIO_SETUP_PIPE_STATISTICS (page 111)
USBIO_PIPE_STATISTICS (page 114)
IOCTL_USBIO_QUERY_PIPE_STATISTICS (page 74)
IOCTL_USBIO_BIND_PIPE (page 66)

IOCTL_USBIO_QUERY_PIPE_STATISTICS

The IOCTL_USBIO_QUERY_PIPE_STATISTICS operation returns statistical data related to a pipe.

`dwIoControlCode`
Set to IOCTL_USBIO_QUERY_PIPE_STATISTICS for this operation.

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

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

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

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

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

Comments

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

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

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

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

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

See Also

USBIO_QUERY_PIPE_STATISTICS (page 112)
USBIO_PIPE_STATISTICS (page 114)
IOCTL_USBIO_SETUP_PIPE_STATISTICS (page 72)
IOCTL_USBIO_BIND_PIPE (page 66)

IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN

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

`dwIoControlCode`
Set to IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN for this operation.

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

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

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

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

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

Comments

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

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

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

See Also

USBIO_PIPE_CONTROL_TRANSFER (page 116)
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT (page 77)
IOCTL_USBIO_BIND_PIPE (page 66)

IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT

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

`dwIoControlCode`
Set to IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT for this operation.

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

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

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

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

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

Comments

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

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

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

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

See Also

USBIO_PIPE_CONTROL_TRANSFER (page 116)
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN (page 76)
IOCTL_USBIO_BIND_PIPE (page 66)

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** 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.

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

The behaviour during a read operation depends on the state of the flag **USBIO_SHORT_TRANSFER_OK** of the related pipe. This setting may be changed by using the **IOCTL_USBIO_SET_PIPE_PARAMETERS** operation. The default state is defined by the registry parameter **ShortTransferOk**. If the flag **USBIO_SHORT_TRANSFER_OK** is set a read operation that returns a data packet that is shorter than the 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. The rest of the buffer is divided into packets. Each packet is transmitted within a USB frame or microframe respectively. The packet size can vary for each frame. Even a packet size of zero bytes is allowed. This way, any data rate of the isochronous stream is supported.

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

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

The layout of a buffer that holds isochronous data is shown in figure 3. At the beginning, the buffer contains a **USBIO_ISO_TRANSFER_HEADER** structure of variable size. The rest of the buffer holds the data packets. The header contains a **USBIO_ISO_TRANSFER** structure that provides general information about the transfer buffer. An important member of this structure is NumberOfPackets. This parameter specifies the number of isochronous data packets contained in the transfer buffer. The maximum number of packets that can be used in a single transfer is limited by the USBIO configuration parameter **MaxIsoPackets** that is defined in the registry. See also section 8 (page 265) for more information.

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

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

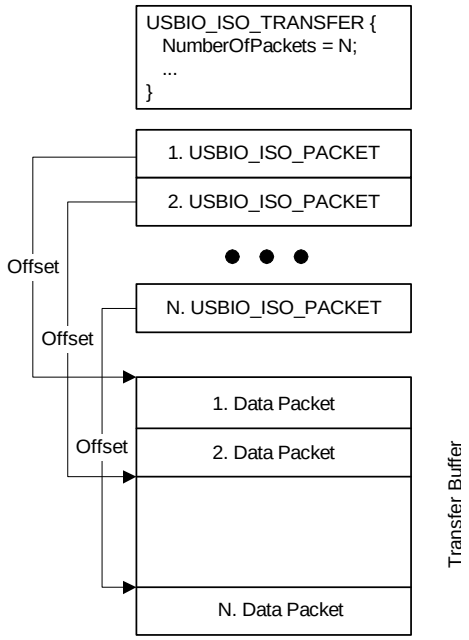

Figure 3: Layout of an isochronous transfer buffer

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

Isochronous Write Transfers

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

Isochronous Read Transfers

There are some constraints that apply to isochronous read operations. The length of each packet reserved in the buffer should be equal to the FIFO size of the respective endpoint. Otherwise, a data overrun error can occur. The data packets have to be placed contiguously into the buffer. In other words, there are no gaps between the packets allowed. The Offset member of all **USBIO_ISO_PACKET** structures has to be initialized correctly by the application before the transfer is initiated. The length of each isochronous data packet received from the device is returned in the Length member of the corresponding **USBIO_ISO_PACKET** structure when the transfer of the whole buffer completes.

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

USBIO_ISO_PACKET structures to learn about the amount of valid data available in the corresponding packet.

4.4 Data Structures

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

USBIO_BANDWIDTH_INFO

The USBIO_BANDWIDTH_INFO structure contains information on the USB bandwidth consumption.

Definition

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

Members

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

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

reserved1
This member is reserved for future use.

reserved2
This member is reserved for future use.

Comments

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

See Also

IOCTL_USBIO_GET_BANDWIDTH_INFO (page 60)

USBIO_DEVICE_INFO

The USBIO_DEVICE_INFO structure contains information on the USB device.

Definition

```
typedef struct _USBIO_DEVICE_INFO{
     ULONG Flags;
     ULONG reserved1;
     ULONG reserved2;
     ULONG reserved3;
} USBIO_DEVICE_INFO;
```

Members

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

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

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

reserved1
This member is reserved for future use.

reserved2
This member is reserved for future use.

reserved3
This member is reserved for future use.

Comments

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

See Also

IOCTL_USBIO_GET_DEVICE_INFO (page 61)

USBIO_DRIVER_INFO

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

Definition

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

Members

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

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

DriverVersion
Contains the version number of the driver executable. The format is as follows: upper 8 bit = major version, lower 8 bit = minor version. For example, V1.41 is represented by a numerical value of 0x0129.

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

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

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

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

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

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

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

Comments

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

See Also

IOCTL_USBIO_GET_DRIVER_INFO (page 62)

USBIO_DESCRIPTOR_REQUEST

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

Definition

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

Members

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

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

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

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

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

Comments

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

See Also

USBIO_REQUEST_RECIPIENT (page 122)
IOCTL_USBIO_GET_DESCRIPTOR (page 38)
IOCTL_USBIO_SET_DESCRIPTOR (page 39)

USBIO_FEATURE_REQUEST

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

Definition

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

Members

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

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

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

Comments

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

See Also

USBIO_REQUEST_RECIPIENT (page 122)
IOCTL_USBIO_SET_FEATURE (page 40)
IOCTL_USBIO_CLEAR_FEATURE (page 41)

USBIO_STATUS_REQUEST

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

Definition

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

Members

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

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

Comments

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

See Also

USBIO_REQUEST_RECIPIENT (page 122)
IOCTL_USBIO_GET_STATUS (page 42)

USBIO_STATUS_REQUEST_DATA

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

Definition

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

Member

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

Comments

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

See Also

IOCTL_USBIO_GET_STATUS (page 42)

USBIO_GET_CONFIGURATION_DATA

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

Definition

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

Member

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

Comments

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

See Also

IOCTL_USBIO_GET_CONFIGURATION (page 43)

USBIO_GET_INTERFACE

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

Definition

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

Member

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

Comments

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

See Also

IOCTL_USBIO_GET_INTERFACE (page 44)

USBIO_GET_INTERFACE_DATA

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

Definition

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

Member

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

Comments

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

See Also

IOCTL_USBIO_GET_INTERFACE (page 44)

USBIO_INTERFACE_SETTING

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

Definition

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

Members

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

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

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

Comments

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

See Also

IOCTL_USBIO_SET_INTERFACE (page 48)
IOCTL_USBIO_SET_CONFIGURATION (page 46)
USBIO_SET_CONFIGURATION (page 97)

USBIO_SET_CONFIGURATION

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

Definition

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

Members

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

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

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

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

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

Comments

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

See Also

USBIO_INTERFACE_SETTING (page 96)
IOCTL_USBIO_SET_CONFIGURATION (page 46)

USBIO_CLASS_OR_VENDOR_REQUEST

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

Definition

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

Members

Flags
This field contains zero or the following value.

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

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

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

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

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

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

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

Comments

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

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

See Also

USBIO_REQUEST_TYPE (page 123)
USBIO_REQUEST_RECIPIENT (page 122)
IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST (page 49)
IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST (page 50)

USBIO_DEVICE_PARAMETERS

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

Definition

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

Members

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

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

The default state of this option is defined by the registry parameter **`ResetDeviceOnClose`**. Refer to section 8 (page 265) for more information.

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

The default state of this option is defined by the registry parameter **`UnconfigureOnClose`**. Refer to section 8 (page 265) for more information.

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

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

The default state of this option is defined by the registry parameter **`EnableRemoteWakeup`**. Refer to section 8 (page 265) for more information.

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

The default time-out value is defined by the registry parameter **`RequestTimeout`**. Refer to section 8 (page 265) for more information.

Comments

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

See Also

IOCTL_USBIO_GET_DEVICE_PARAMETERS (page 52)
IOCTL_USBIO_SET_DEVICE_PARAMETERS (page 53)

USBIO_INTERFACE_CONFIGURATION_INFO

The USBIO_INTERFACE_CONFIGURATION_INFO structure provides information about an interface.

Definition

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

Members

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

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

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

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

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

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

reserved1
Reserved field, set to zero.

reserved2
Reserved field, set to zero.

Comments

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

See Also

USBIO_CONFIGURATION_INFO (page 106)
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 54)

USBIO_PIPE_CONFIGURATION_INFO

The USBIO_PIPE_CONFIGURATION_INFO structure provides information about a pipe.

Definition

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

Members

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

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

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

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

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

Bit 7 = 0: OUT endpoint

Bit 7 = 1: IN endpoint

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

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

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

`reserved1`
Reserved field, set to zero.

`reserved2`
Reserved field, set to zero.

`reserved3`
Reserved field, set to zero.

Comments

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

See Also

USBIO_CONFIGURATION_INFO (page 106)
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 54)

USBIO_CONFIGURATION_INFO

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

Definition

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

Members

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

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

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

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

Comments

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

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

See Also

USBIO_INTERFACE_CONFIGURATION_INFO (page 102)
USBIO_PIPE_CONFIGURATION_INFO (page 104)
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 54)

USBIO_FRAME_NUMBER

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

Definition

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

Member

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

Comments

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

See Also

IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER (page 57)

USBIO_DEVICE_POWER

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

Definition

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

Member

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

Comments

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

See Also

USBIO_DEVICE_POWER_STATE (page 124)
IOCTL_USBIO_GET_DEVICE_POWER_STATE (page 59)
IOCTL_USBIO_SET_DEVICE_POWER_STATE (page 58)

USBIO_BIND_PIPE

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

Definition

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

Member

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

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

Bit 7 = 0: OUT endpoint

Bit 7 = 1: IN endpoint

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

Comments

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

See Also

IOCTL_USBIO_BIND_PIPE (page 66)

USBIO_PIPE_PARAMETERS

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

Definition

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

Member

Flags
This field contains zero or the following value.

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

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

The default state of the USBIO_SHORT_TRANSFER_OK flag is defined by the registry parameter **ShortTransferOk**. Refer to section 8 (page 265) for more information.

Comments

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

See Also

IOCTL_USBIO_GET_PIPE_PARAMETERS (page 70)
IOCTL_USBIO_SET_PIPE_PARAMETERS (page 71)
USBIO_ISO_TRANSFER (page 117)

USBIO_SETUP_PIPE_STATISTICS

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

Definition

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

Members

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

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

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

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

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

Comments

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

See Also

IOCTL_USBIO_SETUP_PIPE_STATISTICS (page 72)
IOCTL_USBIO_QUERY_PIPE_STATISTICS (page 74)
USBIO_PIPE_STATISTICS (page 114)

USBIO_QUERY_PIPE_STATISTICS

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

Definition

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

Member

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

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

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

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

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

Comments

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

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

See Also

IOCTL_USBIO_QUERY_PIPE_STATISTICS (page 74)
USBIO_PIPE_STATISTICS (page 114)

USBIO_PIPE_STATISTICS

The USBIO_PIPE_STATISTICS structure contains statistical data related to a pipe.

Definition

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

Members

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

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

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

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

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

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

transferred on a pipe, modulo 2^64.

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

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

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

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

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

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

reserved1
This member is reserved for future use.

reserved2
This member is reserved for future use.

Comments

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

See Also

IOCTL_USBIO_QUERY_PIPE_STATISTICS (page 74)
USBIO_QUERY_PIPE_STATISTICS (page 112)
IOCTL_USBIO_SETUP_PIPE_STATISTICS (page 72)

USBIO_PIPE_CONTROL_TRANSFER

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

Definition

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

Members

Flags
This field contains zero or the following value.

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

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

Comments

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

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

See Also

IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN (page 76)
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT (page 77)

USBIO_ISO_TRANSFER

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

Definition

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

Members

`NumberOfPackets`
Specifies the number of packets to be sent to or to be received from the device. Each packet corresponds to a USB frame or a microframe respectively. The maximum number of packets allowed in a read or write operation is limited by the registry parameter **`MaxIsoPackets`**. Refer to section 8 (page 265) for more information.

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

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

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

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

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

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

`ErrorCount`

After the isochronous read or write request has been completed by the USBIO driver this member contains the total number of errors occurred during the data transfer. In other words, **`ErrorCount`** specifies the number of frames that caused an error. This field can be used by an application to check if an isochronous read or write request has been completed successfully.

Comments

This data structure is a substructure within the **USBIO_ISO_TRANSFER_HEADER** structure. It is the fixed sized part of the header.

See also section 4.3.2 (page 80) for more information on isochronous data transfers.

See Also

USBIO_ISO_TRANSFER_HEADER (page 120)

USBIO_ISO_PACKET

The USBIO_ISO_PACKET structure defines the size and location of a single isochronous data packet within an isochronous data transfer buffer.

Definition

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

Members

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

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

Status
After the isochronous read or write request is completed by the USBIO driver this field specifies the completion status of the isochronous packet.

Comments

An array of **USBIO_ISO_PACKET** structures is embedded within the **USBIO_ISO_TRANSFER_HEADER** structure. One **USBIO_ISO_PACKET** structure is required for each isochronous data packet to be transferred.

See also section 4.3.2 (page 80) for more information on isochronous data transfers.

See Also

USBIO_ISO_TRANSFER_HEADER (page 120)

USBIO_ISO_TRANSFER_HEADER

The USBIO_ISO_TRANSFER_HEADER structure defines the header that has to be placed at the beginning of an isochronous data transfer buffer.

Definition

```
typedef struct _USBIO_ISO_TRANSFER_HEADER{
     USBIO_ISO_TRANSFER IsoTransfer;
     USBIO_ISO_PACKET IsoPacket[1];
} USBIO_ISO_TRANSFER_HEADER;
```

Members

IsoTransfer
This is the fixed-size part of the header. See the description of the **USBIO_ISO_TRANSFER** data structure for more information.

IsoPacket[1]
This array of **USBIO_ISO_PACKET** structures has a variable length. Each element of the array corresponds to an isochronous data packet that is to be transferred either from or to the transfer buffer.

The number of valid elements in **IsoPacket** is specified by the **NumberOfPackets** member of **IsoTransfer**. See the description of the **USBIO_ISO_TRANSFER** data structure for more information. The maximum number of isochronous data packets per transfer buffer is defined by the registry parameter **MaxIsoPackets**. Refer to section 8 (page 265) for more information.

Comments

A data buffer that is passed to **ReadFile** or **WriteFile** on an isochronous pipe has to contain a valid **USBIO_ISO_TRANSFER_HEADER** structure at offset zero. After this header the buffer contains the isochronous data which is divided into packets. The **IsoPacket** array describes the location and the size of each single isochronous data packet. The isochronous data packets have to be placed into the transfer buffer in such a way that a contiguous data area will be created. In other words, there are no gaps allowed between the isochronous data packets.

See also section 4.3.2 (page 80) for more information on isochronous data transfers.

See Also

USBIO_ISO_TRANSFER (page 117)
USBIO_ISO_PACKET (page 119)

4.5 Enumeration Types

USBIO_PIPE_TYPE

The USBIO_PIPE_TYPE enumeration type contains values that identify the type of a USB pipe or a USB endpoint, respectively.

Definition

```
typedef enum _USBIO_PIPE_TYPE{
     PipeTypeControl  = 0,
     PipeTypeIsochronous,
     PipeTypeBulk,
     PipeTypeInterrupt
} USBIO_PIPE_TYPE;
```

Comments

The meaning of the values is defined by the Universal Serial Bus Specification 1.1, Chapter 9.

See Also

USBIO_PIPE_CONFIGURATION_INFO (page 104)

USBIO_REQUEST_RECIPIENT

The USBIO_REQUEST_RECIPIENT enumeration type contains values that identify the recipient of a USB device request.

Definition

```
typedef enum _USBIO_REQUEST_RECIPIENT{
     RecipientDevice  = 0,
     RecipientInterface,
     RecipientEndpoint,
     RecipientOther
} USBIO_REQUEST_RECIPIENT;
```

Comments

The meaning of the values is defined by the Universal Serial Bus Specification 1.1, Chapter 9.

See Also

USBIO_DESCRIPTOR_REQUEST (page 88)
USBIO_FEATURE_REQUEST (page 90)
USBIO_STATUS_REQUEST (page 91)
USBIO_CLASS_OR_VENDOR_REQUEST (page 98)

USBIO_REQUEST_TYPE

The USBIO_REQUEST_TYPE enumeration type contains values that identify the type of a USB device request.

Definition

```
typedef enum _USBIO_REQUEST_TYPE{
     RequestTypeClass  = 1,
     RequestTypeVendor
} USBIO_REQUEST_TYPE;
```

Comments

The meaning of the values is defined by the Universal Serial Bus Specification 1.1, Chapter 9.

The enumeration does not contain the Standard request type defined by the USB specification. This is because the USB bus driver USBD supports Class and Vendor requests only at its programming interface. Standard requests are generated internally by the USBD.

See Also

USBIO_CLASS_OR_VENDOR_REQUEST (page 98)

USBIO_DEVICE_POWER_STATE

The USBIO_DEVICE_POWER_STATE enumeration type contains values that identify the power state of a device.

Definition

```
typedef enum _USBIO_DEVICE_POWER_STATE{
     DevicePowerStateD0  = 0,
     DevicePowerStateD1,
     DevicePowerStateD2,
     DevicePowerStateD3
} USBIO_DEVICE_POWER_STATE;
```

Entries

DevicePowerStateD0
Device fully on, normal operation.

DevicePowerStateD1
Suspend.

DevicePowerStateD2
Suspend.

DevicePowerStateD3
Device off.

Comments

The meaning of the values is defined by the Power Management specification.

See Also

USBIO_DEVICE_POWER (page 108)

4.6 Error Codes

USBIO_ERR_SUCCESS (0x00000000L)

The operation has been successfully completed.

USBIO_ERR_CRC (0xE0000001L)

A CRC error has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_BTSTUFF (0xE0000002L)

A bit stuffing error has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_DATA_TOGGLE_MISMATCH (0xE0000003L)

A DATA toggle mismatch (DATA0/DATA1 tokens) has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_STALL_PID (0xE0000004L)

A STALL PID has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_DEV_NOT_RESPONDING (0xE0000005L)

The USB device is not responding. This error is reported by the USB host controller driver.

USBIO_ERR_PID_CHECK_FAILURE (0xE0000006L)

A PID check has failed. This error is reported by the USB host controller driver.

USBIO_ERR_UNEXPECTED_PID (0xE0000007L)

An unexpected PID has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_DATA_OVERRUN (0xE0000008L)

A data overrun error has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_DATA_UNDERRUN (0xE0000009L)

A data underrun error has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_RESERVED1 (0xE000000AL)

This error code is reserved by the USB host controller driver.

USBIO_ERR_RESERVED2 (0xE000000BL)

This error code is reserved by the USB host controller driver.

USBIO_ERR_BUFFER_OVERRUN (0xE000000CL)

A buffer overrun has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_BUFFER_UNDERRUN (0xE000000DL)

A buffer underrun has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_NOT_ACCESSED (0xE000000FL)

A data buffer was not accessed. This error is reported by the USB host controller driver. An isochronous data buffer was scheduled too late. The specified frame number does not match the actual frame number.

USBIO_ERR_FIFO (0xE0000010L)

A FIFO error has been detected. This error is reported by the USB host controller driver. The PCI bus latency was too long.

USBIO_ERR_XACT_ERROR (0xE0000011L)

A XACT error has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_BABBLE_DETECTED (0xE0000012L)

A device is babbling. This error is reported by the USB host controller driver. The data transfer phase exceeds the USB frame length.

USBIO_ERR_DATA_BUFFER_ERROR (0xE0000013L)

A data buffer error has been detected. This error is reported by the USB host controller driver.

USBIO_ERR_ENDPOINT_HALTED (0xE0000030L)

The endpoint has been halted by the USB bus driver USBD. This error is reported by the USB bus driver USBD. A pipe will be halted by USBD when a data transmission error (CRC, bit stuff, DATA toggle) occurs. In order to re-enable a halted pipe a **IOCTL_USBIO_RESET_PIPE** request has to be issued on that pipe. See the description of **IOCTL_USBIO_RESET_PIPE** for more information.

USBIO_ERR_NO_MEMORY (0xE0000100L)

A memory allocation attempt has failed. This error is reported by the USB bus driver USBD.

USBIO_ERR_INVALID_URB_FUNCTION (0xE0000200L)

An invalid URB function code has been passed. This error is reported by the USB bus driver USBD.

USBIO_ERR_INVALID_PARAMETER (0xE0000300L)

An invalid parameter has been passed. This error is reported by the USB bus driver USBD.

USBIO_ERR_ERROR_BUSY (0xE0000400L)

There are data transfer requests pending for the device. This error is reported by the USB bus driver USBD.

USBIO_ERR_REQUEST_FAILED (0xE0000500L)

A request has failed. This error is reported by the USB bus driver USBD.

USBIO_ERR_INVALID_PIPE_HANDLE (0xE0000600L)

An invalid pipe handle has been passed. This error is reported by the USB bus driver USBD.

USBIO_ERR_NO_BANDWIDTH (0xE0000700L)

There is not enough bandwidth available. This error is reported by the USB bus driver USBD.

USBIO_ERR_INTERNAL_HC_ERROR (0xE0000800L)

An internal host controller error has been detected. This error is reported by the USB bus driver USBD.

USBIO_ERR_ERROR_SHORT_TRANSFER (0xE0000900L)

A short transfer has been detected. This error is reported by the USB bus driver USBD. If the pipe is not configured accordingly a short packet sent by the device causes this error. Support for short packets has to be enabled explicitly. See **IOCTL_USBIO_SET_PIPE_PARAMETERS** and **IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST** for more information.

USBIO_ERR_BAD_START_FRAME (0xE0000A00L)

A bad start frame has been specified. This error is reported by the USB bus driver USBD.

USBIO_ERR_ISOCH_REQUEST_FAILED (0xE0000B00L)

An isochronous request has failed. This error is reported by the USB bus driver USBD.

USBIO_ERR_FRAME_CONTROL_OWNED (0xE0000C00L)

The USB frame control is currently owned. This error is reported by the USB bus driver USBD.

USBIO_ERR_FRAME_CONTROL_NOT_OWNED (0xE0000D00L)

The USB frame control is currently not owned. This error is reported by the USB bus driver USBD.

USBIO_ERR_NOT_SUPPORTED (0xE0000E00L)

The operation is not supported. This error is reported by the USB bus driver USBD.

USBIO_ERR_INVALID_CONFIGURATION_DESCRIPTOR (0xE0000F00L)

An invalid configuration descriptor was reported by the device. This error is reported by the USB bus driver USBD.

USBIO_ERR_INSUFFICIENT_RESOURCES (0xE8001000L)

There are not enough resources available to complete the operation. This error is reported by the USB bus driver USBD.

USBIO_ERR_SET_CONFIG_FAILED (0xE0002000L)

The set configuration request has failed. This error is reported by the USB bus driver USBD.

USBIO_ERR_USBD_BUFFER_TOO_SMALL (0xE0003000L)

The buffer is too small. This error is reported by the USB bus driver USBD.

USBIO_ERR_USBD_INTERFACE_NOT_FOUND (0xE0004000L)

The interface was not found. This error is reported by the USB bus driver USBD.

USBIO_ERR_INVALID_PIPE_FLAGS (0xE0005000L)

Invalid pipe flags have been specified. This error is reported by the USB bus driver USBD.

USBIO_ERR_USBD_TIMEOUT (0xE0006000L)

The operation has been timed out. This error is reported by the USB bus driver USBD.

USBIO_ERR_DEVICE_GONE (0xE0007000L)

The USB device is gone. This error is reported by the USB bus driver USBD.

USBIO_ERR_STATUS_NOT_MAPPED (0xE0008000L)

This error is reported by the USB bus driver USBD.

USBIO_ERR_CANCELED (0xE0010000L)

The operation has been cancelled. This error is reported by the USB bus driver USBD. If the data transfer requests pending on a pipe are aborted by means of **IOCTL_USBIO_ABORT_PIPE** or **CancelIo** then the operations will be completed with this error code.

USBIO_ERR_ISO_NOT_ACCESSED_BY_HW (0xE0020000L)

The isochronous data buffer was not accessed by the USB host controller. This error is reported by the USB bus driver USBD. An isochronous data buffer was scheduled too late. The specified frame number does not match the actual frame number.

USBIO_ERR_ISO_TD_ERROR (0xE0030000L)

The USB host controller reported an error in a transfer descriptor. This error is reported by the USB bus driver USBD.

USBIO_ERR_ISO_NA_LATE_USBPORT (0xE0040000L)

An isochronous data packet was submitted in time but failed to reach the USB host controller in time. This error is reported by the USB bus driver USBD.

USBIO_ERR_ISO_NOT_ACCESSED_LATE (0xE0050000L)

An isochronous data packet was submitted too late. This error is reported by the USB bus driver USBD.

USBIO_ERR_FAILED (0xE0001000L)

The operation has failed. This error is reported by the USBIO driver.

USBIO_ERR_INVALID_INBUFFER (0xE0001001L)

An invalid input buffer has been passed to an IOCTL operation. This error is reported by the USBIO driver. Make sure the input buffer matches the type and size requirements specified for the IOCTL operation.

USBIO_ERR_INVALID_OUTBUFFER (0xE0001002L)

An invalid output buffer has been passed to an IOCTL operation. This error is reported by the USBIO driver. Make sure the output buffer matches the type and size requirements specified for the IOCTL operation.

USBIO_ERR_OUT_OF_MEMORY (0xE0001003L)

There is not enough system memory available to complete the operation. This error is reported by the USBIO driver.

USBIO_ERR_PENDING_REQUESTS (0xE0001004L)

There are read or write requests pending. This error is reported by the USBIO driver.

USBIO_ERR_ALREADY_CONFIGURED (0xE0001005L)

The USB device is already configured. This error is reported by the USBIO driver.

USBIO_ERR_NOT_CONFIGURED (0xE0001006L)

The USB device is not configured. This error is reported by the USBIO driver.

USBIO_ERR_OPEN_PIPES (0xE0001007L)

There are open pipes. This error is reported by the USBIO driver.

USBIO_ERR_ALREADY_BOUND (0xE0001008L)

Either the handle is already bound to a pipe or the specified pipe is already bound to another handle. This error is reported by the USBIO driver. See **IOCTL_USBIO_BIND_PIPE** for more information.

USBIO_ERR_NOT_BOUND (0xE0001009L)

The handle is not bound to a pipe. This error is reported by the USBIO driver. The operation that has been failed with this error code is related to a pipe. Therefore, the handle has to be bound to a pipe before the operation can be executed. See **IOCTL_USBIO_BIND_PIPE** for more information.

USBIO_ERR_DEVICE_NOT_PRESENT (0xE000100AL)

The USB device has been removed from the system. This error is reported by the USBIO driver. An application should close all handles for the device. After it receives a Plug and Play notification it should perform a re-enumeration of devices.

USBIO_ERR_CONTROL_NOT_SUPPORTED (0xE000100BL)

The specified control code is not supported. This error is reported by the USBIO driver.

USBIO_ERR_TIMEOUT (0xE000100CL)

The operation has been timed out. This error is reported by the USBIO driver.

USBIO_ERR_INVALID_RECIPIENT (0xE000100DL)

An invalid recipient has been specified. This error is reported by the USBIO driver.

USBIO_ERR_INVALID_TYPE (0xE000100EL)

Either an invalid request type has been specified or the operation is not supported by that pipe type. This error is reported by the USBIO driver.

USBIO_ERR_INVALID_IOCTL (0xE000100FL)

An invalid IOCTL code has been specified. This error is reported by the USBIO driver.

USBIO_ERR_INVALID_DIRECTION (0xE0001010L)

The direction of the data transfer request is not supported by that pipe. This error is reported by the USBIO driver. On IN pipes read requests are supported only. On OUT pipes write requests are supported only.

USBIO_ERR_TOO_MUCH_ISO_PACKETS (0xE0001011L)

The number of isochronous data packets specified in an isochronous read or write request exceeds the maximum number of packets supported by the USBIO driver. This error is reported by the USBIO driver. Note that the maximum number of packets allowed per isochronous data buffer can be adjusted by means of the registry parameter **MaxIsoPackets**. Refer to section 8 (page 265) for more information.

USBIO_ERR_POOL_EMPTY (0xE0001012L)

The memory resources are exhausted. This error is reported by the USBIO driver.

USBIO_ERR_PIPE_NOT_FOUND (0xE0001013L)

The specified pipe was not found in the current configuration. This error is reported by the USBIO driver. Note that only endpoints that are included in the current configuration can be used to transfer data.

USBIO_ERR_INVALID_ISO_PACKET (0xE0001014L)

An invalid isochronous data packet has been specified. This error is reported by the USBIO driver. An isochronous data buffer contains an isochronous data packet with invalid **Offset** and/or **Length** parameters. See **USBIO_ISO_PACKET** for more information.

USBIO_ERR_OUT_OF_ADDRESS_SPACE (0xE0001015L)

There are not enough system resources to complete the operation. This error is reported by the USBIO driver.

USBIO_ERR_INTERFACE_NOT_FOUND (0xE0001016L)

The specified interface was not found in the current configuration or in the configuration descriptor. This error is reported by the USBIO driver. Note that only interfaces that are included in the current configuration can be used.

USBIO_ERR_INVALID_DEVICE_STATE (0xE0001017L)

The operation cannot be executed while the USB device is in the current state. This error is reported by the USBIO driver. It is not allowed to submit requests to the device while it is in a power down state.

USBIO_ERR_INVALID_PARAM (0xE0001018L)

An invalid parameter has been specified with an IOCTL operation. This error is reported by the USBIO driver.

USBIO_ERR_DEMO_EXPIRED (0xE0001019L)

The evaluation interval of the USBIO DEMO version has expired. This error is reported by the USBIO driver. The USBIO DEMO version is limited in runtime. After the DEMO evaluation period has expired every operation will be completed with this error code. After the system is rebooted the USBIO DEMO driver can be used for another evaluation interval.

USBIO_ERR_INVALID_POWER_STATE (0xE000101AL)

An invalid power state has been specified. This error is reported by the USBIO driver. Note that it is not allowed to switch from one power down state to another. The device has to be set to D0 before it can be set to another power down state.

USBIO_ERR_POWER_DOWN (0xE000101BL)

The device has entered a power down state. This error is reported by the USBIO driver. When the USB device leaves power state D0 and enters a power down state then all pending read and write requests will be cancelled and completed with this error status. If an application detects this error status it can re-submit the read or write requests immediately. The requests will be queued by the USBIO driver internally.

USBIO_ERR_VERSION_MISMATCH (0xE000101CL)

The API version reported by the USBIO driver does not match the expected version. This error is reported by the USBIO C++ class library USBIOLIB. See **IOCTL_USBIO_GET_DRIVER_INFO** for more information on USBIO version numbers.

USBIO_ERR_SET_CONFIGURATION_FAILED (0xE000101DL)

The set configuration operation has failed. This error is reported by the USBIO driver.

USBIO_ERR_VID_RESTRICTION (0xE0001080L)

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The LIGHT version does not support the Vendor ID reported by the USB device.

USBIO_ERR_ISO_RESTRICTION (0xE0001081L)

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The LIGHT version does not support isochronous transfers.

USBIO_ERR_BULK_RESTRICTION (0xE0001082L)

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The LIGHT version does not support bulk transfers.

USBIO_ERR_EP0_RESTRICTION (0xE0001083L)

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The LIGHT version does not support class or vendor specific SETUP requests or the data transfer length exceeds the limit.

USBIO_ERR_PIPE_RESTRICTION (0xE0001084L)

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The number of endpoints active in the current configuration exceeds the limit enforced by the LIGHT version.

USBIO_ERR_PIPE_SIZE_RESTRICTION (0xE0001085L)

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The FIFO size of an endpoint of the current configuration exceeds the limit enforced by the LIGHT version.

USBIO_ERR_CONTROL_RESTRICTION (0xE0001086L)

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The LIGHT version does not support control endpoints besides EP0.

USBIO_ERR_INTERRUPT_RESTRICTION (0xE0001087L)

The operation has failed due to a restriction of the USBIO LIGHT version. This error is reported by the USBIO driver. The LIGHT version does not support interrupt transfers.

USBIO_ERR_DEVICE_NOT_FOUND (0xE0001100L)

The specified device object does not exist. This error is reported by the USBIO C++ class library USBIOLIB. The USB device is not connected to the system or it has been removed by the user.

USBIO_ERR_DEVICE_NOT_OPEN (0xE0001102L)

No device object was opened. There is no valid handle to execute the operation. This error is reported by the USBIO C++ class library USBIOLIB.

USBIO_ERR_NO_SUCH_DEVICE_INSTANCE (0xE0001104L)

The enumeration of the specified devices has failed. There are no devices of the specified type available. This error is reported by the USBIO C++ class library USBIOLIB.

USBIO_ERR_INVALID_FUNCTION_PARAM (0xE0001105L)

An invalid parameter has been passed to a function. This error is reported by the USBIO C++ class library USBIOLIB.

USBIO_ERR_LOAD_SETUP_API_FAILED (0xE0001106L)

The library *setupapi.dll* could not be loaded. This error is reported by the USBIO C++ class library USBIOLIB. The Setup API that is exported by the system-provided *setupapi.dll* is part of the Win32 API. It is available in Windows 98 and later systems.

5 USBIO Class Library

5.1 Overview

The USBIO Class Library (USBIOLIB) contains classes which provide wrapper functions for all of the features supported by the USBIO programming interface. Using these classes in an application is more convenient than using the USBIO interface directly. The classes are designed to be capable of being extended. In order to meet the requirements of a particular application new classes may be derived from the existing ones. The class library is provided fully in source code.

The following figure shows the classes included in the USBIOLIB and their relations.

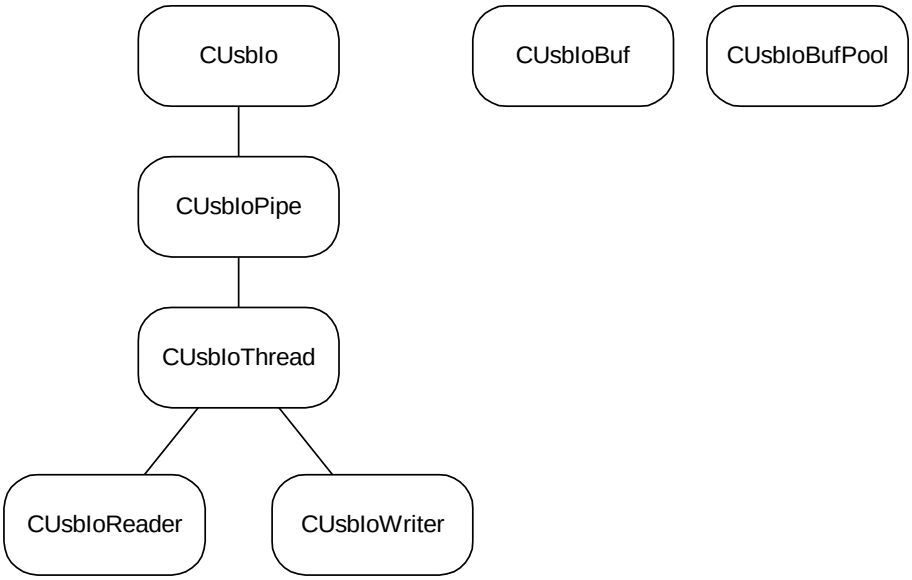

Figure 4: USBIO Class Library

5.1.1 CUsbIo Class

The class **`CUsbIo`** implements the basic interface to the USBIO device driver. It includes all functions that are related to a USBIO device object. Thus, by using an instance of the **`CUsbIo`** class all operations which do not require a pipe context can be performed.

The **`CUsbIo`** class supports device enumeration and an **`Open()`** function that is used to connect an instance of the class to a USBIO device object. The handle that represents the connection is stored inside the class instance. It is used for all subsequent requests to the device.

For each device-related operation the USBIO driver supports, a member function exists in the **`CUsbIo`** class. The function takes the parameters that are required for the operation and returns the status that is reported by the USBIO driver.

5.1.2 CUsbIoPipe Class

The class **`CUsbIoPipe`** extends the **`CUsbIo`** class by functions that are related to a USBIO pipe object. An instance of the **`CUsbIoPipe`** class is associated directly with a USBIO pipe object.

In order to establish the connection to the pipe the class provides a **Bind()** function. After a **CUsbIoPipe** instance is bound, pipe-related functions can be performed by using member functions of the class.

For each pipe-related operation that the USBIO driver supports a member function exists in the **CUsbIoPipe** class. The function takes the parameters that are required for the operation and returns the status that is reported by the USBIO driver.

The **CUsbIoPipe** class supports an asynchronous communication model for data transfers 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.

5.1.3 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.

5.1.4 CUsbIoReader Class

The class **CUsbIoReader** extends the **CUsbIoThread** class by a specific worker thread routine that continuously sends read requests to the pipe. The thread's main routine gets buffers from an internal buffer pool and submits them to the pipe using the **Read()** function of the **CUsbIoPipe** class. After all buffers are submitted the routine waits for the first pending buffer to complete. If a buffer is completed by the USBIO driver the virtual member function **ProcessData** is called with this buffer. Within this function the data received from the pipe should be processed. The **ProcessData** function has to be implemented by a class that is derived from **CUsbIoReader**. After that, the buffer is put back to the pool and the main loop is started from the beginning.

5.1.5 CUsbIoWriter Class

The class **CUsbIoWriter** extends the **CUsbIoThread** class by a specific worker thread routine that continuously sends write requests to the pipe. The thread's main routine gets a buffer from an internal buffer pool and calls the virtual member function **ProcessBuffer** to fill the buffer with data. After that, the buffer is sent to the pipe using the **Write()** function of the **CUsbIoPipe** class. After all buffers are submitted the routine waits for the first pending buffer to complete. If a buffer is completed by the USBIO driver the buffer is put back to the pool and the main loop is started from the beginning.

5.1.6 CUsbIoBuf Class

The helper class **CUsbIoBuf** is used as a descriptor for buffers that are processed by the class **CUsbIoPipe** and derived classes. One instance of the **CUsbIoBuf** class has to be created for each buffer. The **CUsbIoBuf** object stores context and status information that is needed to process the buffer asynchronously.

The **CUsbIoBuf** class contains a link element (Next pointer). This may be used to build a chain of linked buffer objects to hold them in a list. This way, the management of buffers can be simplified.

5.1.7 CUsbIoBufPool Class

The class **CUsbIoBufPool** is used to manage a pool of free buffers. It provides functions used to allocate an initial number of buffers, to get a buffer from the pool, and to put a buffer back to the pool.

5.2 Class Library Reference

CUsbIo class

This class implements the interface to the USBIO device driver. It contains only general device-related functions that can be executed without a pipe context. Pipe specific functions are implemented by the **CUsbIoPipe** class.

Member Functions

CUsbIo::CUsbIo

Standard constructor of the CUsbIo class.

Definition

```
CUsbIo();
```

See Also

CUsbIo::˜CUsbIo (page 140)
CUsbIo::Open (page 143)

CUsbIo::˜CUsbIo

Destructor of the CUsbIo class.

Definition

```
virtual
˜CUsbIo();
```

See Also

CUsbIo::CUsbIo (page 140)
CUsbIo::Close (page 145)

CUsbIo::CreateDeviceList

Creates an internal device list.

Definition

```
static HDEVINFO
CreateDeviceList(
     const GUID* InterfaceGuid
     );
```

Parameter

`InterfaceGuid`
This is the predefined interface GUID of the USBIO device driver or a user defined GUID which must be inserted in the USBIO.INF file.

Return Value

Returns a handle to the device list if successful, or NULL otherwise.

Comments

The function creates a windows-internal device list that contains all matching interfaces. The device interface is identified by **`InterfaceGuid`**. A handle for the list is returned in case of success, or NULL is returned in case of error. The device list can be iterated by means of **CUsbIo::Open**.

The device list returned must be freed by a call to **CUsbIo::DestroyDeviceList**.

Note that CreateDeviceList is declared static. It can be used independently of class instances.

See Also

CUsbIo::DestroyDeviceList (page 142)
CUsbIo::Open (page 143)

CUsbIo::DestroyDeviceList

Destroy the internal device list.

Definition

```
static void
DestroyDeviceList(
     HDEVINFO DeviceList
     );
```

Parameter

DeviceList
A handle to a device list returned by **CUsbIo::CreateDeviceList**.

Comments

Use this function to destroy a device list that was generated by a call to **CUsbIo::CreateDeviceList**.

Note that DestroyDeviceList is declared static. It can be used independently of class instances.

See Also

CUsbIo::CreateDeviceList (page 141)

CUsbIo::Open

Open an USB device.

Definition

```
DWORD
Open(
     int DeviceNumber,
     HDEVINFO DeviceList,
     const GUID* InterfaceGuid
     );
```

Parameters

DeviceNumber
Specifies the index number of the USB Device. The index is zero-based. Note that the association between this number and the USB device can change with each call to **CUsbIo::CreateDeviceList**.

DeviceList
A handle to the internal device list which was returned by the function **CUsbIo::CreateDeviceList** or NULL. For more information see below.

InterfaceGuid
Points to a caller-provided variable of type GUID. The specified GUID is the predefined interface GUID of the USBIO device driver, or a user-defined GUID which has to be defined in the USBIO.INF file. This parameter will be ignored if DeviceList is set to NULL. For more information, see below.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

There are two options:

(A) DeviceList != NULL
The device list provided in DeviceList must have been built using **CUsbIo::CreateDeviceList**. The GUID that identifies the device interface must be provided in InterfaceGuid. **DeviceNumber** is used to iterate through the device list. It should start with zero and should be incremented after each call to **Open**. If no more instances of the interface are available then the status code **USBIO_ERR_NO_SUCH_DEVICE_INSTANCE** is returned.
Note: This is the recommended way of implementing a device enumeration.

(B) `DeviceList == NULL`
The parameter InterfaceGuid will be ignored. DeviceNumber will be used to build an old-style device name that starts with the string defined by **`USBIO_DEVICE_NAME`** (see also the comments on **`USBIO_DEVICE_NAME`** in UsbIo.h).
Note: This mode should be used only if compatibility to earlier versions of USBIO is required! It will work only if the creation of static device names is enabled in the USBIO.INF file. It is not recommended to use this mode.

See Also

CUsbIo::CreateDeviceList (page 141)
CUsbIo::DestroyDeviceList (page 142)
CUsbIo::Close (page 145)
CUsbIo::IsOpen (page 149)
CUsbIo::IsCheckedBuild (page 150)
CUsbIo::IsDemoVersion (page 151)
CUsbIo::IsLightVersion (page 152)

CUsbIo::Close

Close the USB device.

Definition

```
void
Close();
```

Comments

This function can be called if the device is not open. It does nothing in this case.

Any thread associated with the class instance should have been stopped before this function is called. See **CUsbIoThread::ShutdownThread**.

See Also

CUsbIo::CreateDeviceList (page 141)
CUsbIo::DestroyDeviceList (page 142)
CUsbIo::Open (page 143)
CUsbIoThread::ShutdownThread (page 217)

CUsbIo::GetDeviceInstanceDetails

Get detailed information on an USB device instance.

Definition

```
DWORD
GetDeviceInstanceDetails(
     int DeviceNumber,
     HDEVINFO DeviceList,
     const GUID* InterfaceGuid
     );
```

Parameters

DeviceNumber
Specifies the index of the USB Device. The index is zero-based. Note that the association between this number and the USB device can change with each call to **CUsbIo::CreateDeviceList**.

DeviceList
A handle to the internal device list which was returned by the function **CUsbIo::CreateDeviceList**.

InterfaceGuid
Points to a caller-provided variable of type GUID. The specified GUID is the predefined interface GUID of the USBIO device driver, or a user-defined GUID which has to be defined in the USBIO.INF file.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function retrieves detailed information on a device instance that has been enumerated by means of **CUsbIo::CreateDeviceList**. That information includes the device path name that has to be passed to CreateFile in order to open the device instance. The device path name is returned by **CUsbIo::GetDevicePathName**.

This function is used internally by the implementation of **CUsbIo::Open**. Normally, it is not called directly by an application.

See Also

CUsbIo::CreateDeviceList (page 141)
CUsbIo::DestroyDeviceList (page 142)
CUsbIo::Open (page 143)

CUsbIo::GetDevicePathName (page 148)

CUsbIo::GetDevicePathName

Returns the path name that is required to open the device instance.

Definition

```
const char*
GetDevicePathName();
```

Return Value

Returns a pointer to the path name associated with this device instance, or NULL. The returned pointer is temporarily valid only and should not be stored for later use. It becomes invalid if the device is closed. If no device is opened, the return value is NULL.

The return value is always NULL if the device was opened using case (A) described in the comments of the **CUsbIo::Open** function.

Comments

This function retrieves the device path name of the device instance. The path name is available after a device enumeration has been performed by a call to **CUsbIo::CreateDeviceList** and **CUsbIo::GetDeviceInstanceDetails** or **CUsbIo::Open** has been called.

This function is used internally by the implementation of **CUsbIo::Open**. Normally, it is not called directly by an application.

See Also

CUsbIo::CreateDeviceList (page 141)
CUsbIo::DestroyDeviceList (page 142)
CUsbIo::GetDeviceInstanceDetails (page 146)
CUsbIo::Open (page 143)
CUsbIo::Close (page 145)

CUsbIo::IsOpen

Returns TRUE if the class instance is attached to a device.

Definition

```
BOOL
IsOpen();
```

Return Value

Returns TRUE when a device was opened by a successful call to **Open**. Returns FALSE if no device is opened.

See Also

CUsbIo::Open (page 143)
CUsbIo::Close (page 145)

CUsbIo::IsCheckedBuild

Returns TRUE if a checked build (debug version) of the USBIO driver was detected.

Definition

```
BOOL
IsCheckedBuild();
```

Return Value

Returns TRUE if the checked build of the USBIO driver is running, FALSE otherwise.

Comments

The device must have been opened before this function is called.

See Also

CUsbIo::Open (page 143)
CUsbIo::IsDemoVersion (page 151)
CUsbIo::IsLightVersion (page 152)

CUsbIo::IsDemoVersion

Returns TRUE if the Demo version of the USBIO driver was detected.

Definition

```
BOOL
IsDemoVersion();
```

Return Value

Returns TRUE if the Demo version of the USBIO driver is running, FALSE otherwise.

Comments

The device must have been opened before this function is called.

See Also

CUsbIo::Open (page 143)
CUsbIo::IsCheckedBuild (page 150)
CUsbIo::IsLightVersion (page 152)

CUsbIo::IsLightVersion

Returns TRUE if the Light version of the USBIO driver was detected.

Definition

```
BOOL
IsLightVersion();
```

Return Value

Returns TRUE if the Light version of the USBIO driver is running, FALSE otherwise.

Comments

The device must have been opened before this function is called.

See Also

CUsbIo::Open (page 143)
CUsbIo::IsCheckedBuild (page 150)
CUsbIo::IsDemoVersion (page 151)

CUsbIo::IsOperatingAtHighSpeed

Returns TRUE if the USB 2.0 device is operating at high speed (480 Mbit/s).

Definition

```
BOOL
IsOperatingAtHighSpeed();
```

Return Value

Returns TRUE if the USB device is operating at high speed, FALSE otherwise.

Comments

If this function returns TRUE then the USB device operates in high speed mode. The USB 2.0 device is connected to a hub port that is high speed capable.

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

This function calls **CUsbIo::GetDeviceInfo** to get the requested information.

The device must have been opened before this function is called.

See Also

CUsbIo::Open (page 143)
CUsbIo::GetDeviceInfo (page 155)

CUsbIo::GetDriverInfo

Get information on the USBIO device driver.

Definition

```
DWORD
GetDriverInfo(
     USBIO_DRIVER_INFO* DriverInfo
     );
```

Parameter

DriverInfo
Pointer to a caller-provided variable. The structure returns the API version, the driver version, and the build number.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_DRIVER_INFO** operation. See also the description of **IOCTL_USBIO_GET_DRIVER_INFO** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::Close (page 145)
USBIO_DRIVER_INFO (page 86)
IOCTL_USBIO_GET_DRIVER_INFO (page 62)

CUsbIo::GetDeviceInfo

Get information about the USB device.

Definition

```
DWORD
GetDeviceInfo(
     USBIO_DEVICE_INFO* DeviceInfo
     );
```

Parameter

DeviceInfo
Pointer to a caller-provided variable. The structure returns information on the USB device. This includes a flag that indicates whether the device operates in high speed mode or not.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function can be used to detect if the device operates in high speed mode.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_DEVICE_INFO** operation. See also the description of **IOCTL_USBIO_GET_DEVICE_INFO** for further information.

See Also

CUsbIo::Open (page 143)
USBIO_DEVICE_INFO (page 85)
IOCTL_USBIO_GET_DEVICE_INFO (page 61)

CUsbIo::GetBandwidthInfo

Get information on the current USB bandwidth consumption.

Definition

```
DWORD
GetBandwidthInfo(
     USBIO_BANDWIDTH_INFO* BandwidthInfo
     );
```

Parameter

BandwidthInfo
Pointer to a caller-provided variable. The structure returns information on the bandwidth that is available on the USB.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The function enables an application to check the bandwidth that is available on the USB. Depending on this information an application can select an appropriate device configuration, if desired.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_BANDWIDTH_INFO** operation. See also the description of **IOCTL_USBIO_GET_BANDWIDTH_INFO** for further information.

See Also

CUsbIo::Open (page 143)
USBIO_BANDWIDTH_INFO (page 84)
IOCTL_USBIO_GET_BANDWIDTH_INFO (page 60)

CUsbIo::GetDescriptor

Get a descriptor from the device.

Definition

```
DWORD
GetDescriptor(
     void* Buffer,
     DWORD& ByteCount,
     USBIO_REQUEST_RECIPIENT Recipient,
     UCHAR DescriptorType,
     UCHAR DescriptorIndex = 0,
     USHORT LanguageId = 0
     );
```

Parameters

Buffer
Pointer to a caller-provided buffer. The buffer receives the requested descriptor.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Buffer**. After the function successfully returned **ByteCount** contains the number of valid bytes returned in the buffer.

Recipient
Specifies the recipient of the request. Possible values are enumerated by **USBIO_REQUEST_RECIPIENT**.

DescriptorType
The type of the descriptor to request. Values are defined by the USB specification, chapter 9.

DescriptorIndex
The index of the descriptor to request. Set to zero if an index is not used for the descriptor type, e.g. for a device descriptor.

LanguageId
The language ID of the descriptor to request. Used for string descriptors only. Set to zero if not used.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

If the size of the provided buffer is less than the total size of the requested descriptor then only the specified number of bytes from the beginning of the descriptor is returned.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_DESCRIPTOR** operation. See also the description of **IOCTL_USBIO_GET_DESCRIPTOR** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::GetDeviceDescriptor (page 159)
CUsbIo::GetConfigurationDescriptor (page 160)
CUsbIo::GetStringDescriptor (page 162)
CUsbIo::SetDescriptor (page 164)
USBIO_REQUEST_RECIPIENT (page 122)
IOCTL_USBIO_GET_DESCRIPTOR (page 38)

CUsbIo::GetDeviceDescriptor

Get the device descriptor from the device.

Definition

```
DWORD
GetDeviceDescriptor(
     USB_DEVICE_DESCRIPTOR* Desc
     );
```

Parameter

Desc
Pointer to a caller-provided variable that receives the requested descriptor.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

GetDeviceDescriptor calls **CUsbIo::GetDescriptor** to retrieve the descriptor. Thus, for detailed information see also **CUsbIo::GetDescriptor**.

The device must have been opened before this function is called.

See Also

CUsbIo::Open (page 143)
CUsbIo::GetDescriptor (page 157)
CUsbIo::GetConfigurationDescriptor (page 160)
CUsbIo::GetStringDescriptor (page 162)

CUsbIo::GetConfigurationDescriptor

Get a configuration descriptor from the device.

Definition

```
DWORD
GetConfigurationDescriptor(
     USB_CONFIGURATION_DESCRIPTOR* Desc,
     DWORD& ByteCount,
     UCHAR Index = 0
     );
```

Parameters

Desc
Pointer to a caller-provided buffer that receives the requested descriptor. Note that the size of the configuration descriptor depends on the USB device. See also the comments below.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Desc**. After the function successfully returned **ByteCount** contains the number of valid bytes returned in the buffer.

Index
The index of the descriptor to request.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

GetConfigurationDescriptor calls **CUsbIo::GetDescriptor** to retrieve the descriptor. Thus, for detailed information see also **CUsbIo::GetDescriptor**.

If the total size of the configuration descriptor is not known it can be retrieved in a two step process. With a first call to this function the fixed part of the descriptor which is defined by **USB_CONFIGURATION_DESCRIPTOR** is retrieved. The total size of the descriptor is indicated by the *wTotalLength* field of the structure. In a second step a buffer of the required size can be allocated and the complete descriptor can be retrieved with another call to this function.

The device must have been opened before this function is called.

See Also

CUsbIo::Open (page 143)
CUsbIo::GetDescriptor (page 157)
CUsbIo::GetDeviceDescriptor (page 159)
CUsbIo::GetStringDescriptor (page 162)

CUsbIo::GetStringDescriptor

Get a string descriptor from the device.

Definition

```
DWORD
GetStringDescriptor(
     USB_STRING_DESCRIPTOR* Desc,
     DWORD& ByteCount,
     UCHAR Index = 0,
     UCHAR LanguageId = 0
     );
```

Parameters

Desc
Pointer to a caller-provided buffer that receives the requested descriptor. Note that according to the USB specification the maximum size of a string descriptor is 256 bytes.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Desc**. After the function successfully returned **ByteCount** contains the number of valid bytes returned in the buffer.

Index
The index of the descriptor to request. Set to 0 to retrieve a list of supported language IDs. See also the comments below.

LanguageId
The language ID of the string descriptor to request.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

GetStringDescriptor calls **CUsbIo::GetDescriptor** to retrieve the descriptor. Thus, for detailed information see also **CUsbIo::GetDescriptor**.

If this function is called with **Index** set to 0 the device returns a list of language IDs it supports. An application can select the correct language ID and use it in subsequent calls to this function.

The device must have been opened before this function is called.

See Also

CUsbIo::Open (page 143)
CUsbIo::GetDescriptor (page 157)
CUsbIo::GetDeviceDescriptor (page 159)
CUsbIo::GetConfigurationDescriptor (page 160)

CUsbIo::SetDescriptor

Set a descriptor of the device.

Definition

```
DWORD
SetDescriptor(
     const void* Buffer,
     DWORD& ByteCount,
     USBIO_REQUEST_RECIPIENT Recipient,
     UCHAR DescriptorType,
     UCHAR DescriptorIndex = 0,
     USHORT LanguageId = 0
     );
```

Parameters

Buffer
Pointer to a caller-provided buffer. The buffer contains the descriptor data to be set.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the descriptor pointed to by **Buffer**. After the function successfully returned **ByteCount** contains the number of bytes transferred.

Recipient
Specifies the recipient of the request. Possible values are enumerated by **USBIO_REQUEST_RECIPIENT**.

DescriptorType
The type of the descriptor to set. Values are defined by the USB specification, chapter 9.

DescriptorIndex
The index of the descriptor to set.

LanguageId
The language ID of the descriptor to set. Used for string descriptors only. Set to zero if not used.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

Note that most devices do not support the set descriptor request.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_SET_DESCRIPTOR** operation. See also the description of **IOCTL_USBIO_SET_DESCRIPTOR** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::GetDescriptor (page 157)
CUsbIo::GetDeviceDescriptor (page 159)
CUsbIo::GetConfigurationDescriptor (page 160)
CUsbIo::GetStringDescriptor (page 162)
USBIO_REQUEST_RECIPIENT (page 122)
IOCTL_USBIO_SET_DESCRIPTOR (page 39)

CUsbIo::SetFeature

Send a set feature request to the USB device.

Definition

```
DWORD
SetFeature(
     USBIO_REQUEST_RECIPIENT Recipient,
     USHORT FeatureSelector,
     USHORT Index = 0
     );
```

Parameters

Recipient
Specifies the recipient of the request. Possible values are enumerated by **USBIO_REQUEST_RECIPIENT**.

FeatureSelector
Specifies the feature selector value for the request. The values are defined by the recipient. Refer to the USB specification, chapter 9 for more information.

Index
Specifies the index value for the set feature request. The values are defined by the device. Refer to the USB specification, chapter 9 for more information.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_SET_FEATURE** operation. See also the description of **IOCTL_USBIO_SET_FEATURE** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::ClearFeature (page 167)
CUsbIo::GetStatus (page 168)
USBIO_REQUEST_RECIPIENT (page 122)
IOCTL_USBIO_SET_FEATURE (page 40)

CUsbIo::ClearFeature

Send a clear feature request to the USB device.

Definition

```
DWORD
ClearFeature(
     USBIO_REQUEST_RECIPIENT Recipient,
     USHORT FeatureSelector,
     USHORT Index  =0
     );
```

Parameters

Recipient
Specifies the recipient of the request. Possible values are enumerated by **USBIO_REQUEST_RECIPIENT**.

FeatureSelector
Specifies the feature selector value for the request. The values are defined by the recipient. Refer to the USB specification, chapter 9 for more information.

Index
Specifies the index value for the clear feature request. The values are defined by the device. Refer to the USB specification, chapter 9 for more information.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_CLEAR_FEATURE** operation. See also the description of **IOCTL_USBIO_CLEAR_FEATURE** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::SetFeature (page 166)
CUsbIo::GetStatus (page 168)
USBIO_REQUEST_RECIPIENT (page 122)
IOCTL_USBIO_CLEAR_FEATURE (page 41)

CUsbIo::GetStatus

Send a get status request to the USB device.

Definition

```
DWORD
GetStatus(
     USHORT& StatusValue,
     USBIO_REQUEST_RECIPIENT Recipient,
     USHORT Index = 0
     );
```

Parameters

StatusValue
If the function call is successful this variable returns the 16-bit value that is returned by the recipient in response to the get status request. The interpretation of the value is specific to the recipient. Refer to the USB specification, chapter 9 for more information.

Recipient
Specifies the recipient of the request. Possible values are enumerated by **USBIO_REQUEST_RECIPIENT**.

Index
Specifies the index value for the get status request. The values are defined by the device. Refer to the USB specification, chapter 9 for more information.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_STATUS** operation. See also the description of **IOCTL_USBIO_GET_STATUS** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::SetFeature (page 166)
CUsbIo::ClearFeature (page 167)
USBIO_REQUEST_RECIPIENT (page 122)
IOCTL_USBIO_GET_STATUS (page 42)

CUsbIo::ClassOrVendorInRequest

Sends a class or vendor specific request with a data phase in device to host (IN) direction.

Definition

```
DWORD
ClassOrVendorInRequest(
     void* Buffer,
     DWORD& ByteCount,
     const USBIO_CLASS_OR_VENDOR_REQUEST* Request
     );
```

Parameters

Buffer
Pointer to a caller-provided buffer. The buffer receives the data transferred in the IN data phase.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Buffer**. After the function successfully returned **ByteCount** contains the number of valid bytes returned in the buffer.

Request
Pointer to a caller-provided variable that defines the request to be generated.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST** operation. See also the description of **IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::ClassOrVendorOutRequest (page 170)
USBIO_CLASS_OR_VENDOR_REQUEST (page 98)
IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST (page 49)

CUsbIo::ClassOrVendorOutRequest

Sends a class or vendor specific request with a data phase in host to device (OUT) direction.

Definition

```
DWORD
ClassOrVendorOutRequest(
     const void* Buffer,
     DWORD& ByteCount,
     const USBIO_CLASS_OR_VENDOR_REQUEST* Request
     );
```

Parameters

Buffer
Pointer to a caller-provided buffer that contains the data to be transferred in the OUT data phase.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Buffer**. This is the number of bytes to be transferred in the data phase. After the function successfully returned **ByteCount** contains the number of bytes transferred.

Request
Pointer to a caller-provided variable that defines the request to be generated.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST** operation. See also the description of **IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::ClassOrVendorInRequest (page 169)
USBIO_CLASS_OR_VENDOR_REQUEST (page 98)
IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST (page 50)

CUsbIo::SetConfiguration

Set the device to the configured state.

Definition

```
DWORD
SetConfiguration(
     const USBIO_SET_CONFIGURATION* Conf
     );
```

Parameter

Conf
Points to a caller-provided structure that defines the configuration to be set.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device has to be configured before any data transfer from or to its endpoints can take place. Only those endpoints that are included in the configuration will be activated and can be subsequently used for data transfers.

If the device provides more than one interface all interfaces must be configured in one call to this function.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_SET_CONFIGURATION** operation. See also the description of **IOCTL_USBIO_SET_CONFIGURATION** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::UnconfigureDevice (page 172)
CUsbIo::GetConfiguration (page 173)
CUsbIo::SetInterface (page 175)
USBIO_SET_CONFIGURATION (page 97)
IOCTL_USBIO_SET_CONFIGURATION (page 46)

CUsbIo::UnconfigureDevice

Set the device to the unconfigured state.

Definition

```
DWORD
UnconfigureDevice();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_UNCONFIGURE_DEVICE** operation. See also the description of **IOCTL_USBIO_UNCONFIGURE_DEVICE** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::SetConfiguration (page 171)
CUsbIo::GetConfiguration (page 173)
IOCTL_USBIO_UNCONFIGURE_DEVICE (page 47)

CUsbIo::GetConfiguration

This function returns the current configuration value.

Definition

```
DWORD
GetConfiguration(
     UCHAR& ConfigurationValue
     );
```

Parameter

ConfigurationValue
If the function call is successful this variable returns the current configuration value. A value of 0 means the USB device is not configured.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The configuration value returned by this function corresponds to the *bConfiguration* field of the active configuration descriptor. Note that the configuration value does not necessarily correspond to the index of the configuration descriptor.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_CONFIGURATION** operation. See also the description of **IOCTL_USBIO_GET_CONFIGURATION** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::SetConfiguration (page 171)
IOCTL_USBIO_GET_CONFIGURATION (page 43)

CUsbIo::GetConfigurationInfo

Get information on the interfaces and endpoints available in the current configuration.

Definition

```
DWORD
GetConfigurationInfo(
     USBIO_CONFIGURATION_INFO* Info
     );
```

Parameter

Info
Points to a caller-provided variable that receives the configuration information.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The information returned is retrieved from the USBIO driver's internal data base. This function does not cause any action on the USB.

The device must have been opened and configured before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_CONFIGURATION_INFO** operation. See also the description of **IOCTL_USBIO_GET_CONFIGURATION_INFO** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::SetConfiguration (page 171)
USBIO_CONFIGURATION_INFO (page 106)
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 54)

CUsbIo::SetInterface

This function changes the alternate setting of an interface.

Definition

```
DWORD
SetInterface(
     const USBIO_INTERFACE_SETTING* Setting
     );
```

Parameter

Setting
Points to a caller-provided structure that specifies the interface and the alternate settings to be set.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened and configured before this function is called.

This function is a wrapper for the **IOCTL_USBIO_SET_INTERFACE** operation. See also the description of **IOCTL_USBIO_SET_INTERFACE** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::SetConfiguration (page 171)
CUsbIo::GetInterface (page 176)
USBIO_INTERFACE_SETTING (page 96)
IOCTL_USBIO_SET_INTERFACE (page 48)

CUsbIo::GetInterface

This function returns the active alternate setting of an interface.

Definition

```
DWORD
GetInterface(
     UCHAR& AlternateSetting,
     USHORT Interface = 0
     );
```

Parameters

AlternateSetting
If the function call is successful this variable returns the current alternate setting of the interface.

Interface
Specifies the index of the interface to be queried.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened and configured before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_INTERFACE** operation. See also the description of **IOCTL_USBIO_GET_INTERFACE** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::SetConfiguration (page 171)
CUsbIo::SetInterface (page 175)
IOCTL_USBIO_GET_INTERFACE (page 44)

CUsbIo::StoreConfigurationDescriptor

Store a configuration descriptor in the USBIO device driver.
This function is obsolete (see below).

Definition

```
DWORD
StoreConfigurationDescriptor(
     const USB_CONFIGURATION_DESCRIPTOR* Desc
     );
```

Parameter

Desc
Pointer to the configuration descriptor to be stored.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

Store a configuration descriptor in the USBIO device driver and use it for the next set configuration request. This allows to work around problems with certain devices.

The device must have been opened before this function is called.

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

This function is a wrapper for the **IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR** operation. See also the description of **IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::SetConfiguration (page 171)
IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR (page 45)

CUsbIo::GetDeviceParameters

Query device-related parameters from the USBIO device driver.

Definition

```
DWORD
GetDeviceParameters(
     USBIO_DEVICE_PARAMETERS* DevParam
     );
```

Parameter

DevParam
Points to a caller-provided variable that receives the current parameter settings.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_DEVICE_PARAMETERS** operation. See also the description of **IOCTL_USBIO_GET_DEVICE_PARAMETERS** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::SetDeviceParameters (page 179)
USBIO_DEVICE_PARAMETERS (page 100)
IOCTL_USBIO_GET_DEVICE_PARAMETERS (page 52)

CUsbIo::SetDeviceParameters

Set device-related parameters in the USBIO device driver.

Definition

```
DWORD
SetDeviceParameters(
     const USBIO_DEVICE_PARAMETERS* DevParam
     );
```

Parameter

DevParam
Points to a caller-provided variable that specifies the parameters to be set.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

Default device parameters are stored in the registry during USBIO driver installation. The default value can be changed in the INF file or in the registry. Device parameters set by means of this function are valid until the device is removed from the PC or the PC is booted. A modification during run-time does not change the default in the registry.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_SET_DEVICE_PARAMETERS** operation. See also the description of **IOCTL_USBIO_SET_DEVICE_PARAMETERS** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::GetDeviceParameters (page 178)
USBIO_DEVICE_PARAMETERS (page 100)
IOCTL_USBIO_SET_DEVICE_PARAMETERS (page 53)

CUsbIo::ResetDevice

Force an USB reset.

Definition

```
DWORD
ResetDevice();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function causes an USB reset to be issued on the hub port the device is connected to. This will abort all pending read and write requests and unbind all pipes. The device will be set to the unconfigured state.

Note: The device must be in the configured state when this function is called. ResetDevice does not work when the system-provided USB multi-interface driver is used (see also *problems.txt* in the USBIO package).

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_RESET_DEVICE** operation. See also the description of **IOCTL_USBIO_RESET_DEVICE** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::SetConfiguration (page 171)
CUsbIo::UnconfigureDevice (page 172)
CUsbIo::CyclePort (page 181)
CUsbIoPipe::Bind (page 190)
CUsbIoPipe::Unbind (page 192)
CUsbIoPipe::AbortPipe (page 202)
IOCTL_USBIO_RESET_DEVICE (page 55)

CUsbIo::CyclePort

Simulates a device disconnect/connect cycle.

Definition

```
DWORD
CyclePort();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function causes a device disconnect/connect cycle and an unload/load cycle for the USBIO device driver as well.

Note: CyclePort does not work on multi-interface devices (see also *problems.txt* in the USBIO package).

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_CYCLE_PORT** operation. See also the description of **IOCTL_USBIO_CYCLE_PORT** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::ResetDevice (page 180)
IOCTL_USBIO_CYCLE_PORT (page 64)

CUsbIo::GetCurrentFrameNumber

Get the current USB frame number from the host controller.

Definition

```
DWORD
GetCurrentFrameNumber(
     DWORD& FrameNumber
     );
```

Parameter

FrameNumber
If the function call is successful this variable returns the current frame number.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The returned frame number is a 32 bit value. The 11 least significant bits correspond to the frame number in the USB frame token.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER** operation. See also the description of **IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER** for further information.

See Also

CUsbIo::Open (page 143)
IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER (page 57)

CUsbIo::GetDevicePowerState

Returns the current device power state.

Definition

```
DWORD
GetDevicePowerState(
     USBIO_DEVICE_POWER_STATE& DevicePowerState
     );
```

Parameter

DevicePowerState
If the function call is successful this variable returns the current device power state.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_GET_DEVICE_POWER_STATE** operation. See also the description of **IOCTL_USBIO_GET_DEVICE_POWER_STATE** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::SetDevicePowerState (page 184)
USBIO_DEVICE_POWER_STATE (page 124)
IOCTL_USBIO_GET_DEVICE_POWER_STATE (page 59)

CUsbIo::SetDevicePowerState

Set the device power state.

Definition

```
DWORD
SetDevicePowerState(
     USBIO_DEVICE_POWER_STATE DevicePowerState
     );
```

Parameter

DevicePowerState
Specifies the device power state to be set.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

In order to set the device to suspend it must be in the configured state.

When the device is set to suspend all pending read and write requests will be returned by the USBIO driver with an error status of **USBIO_ERR_POWER_DOWN**. An application can ignore this error status and submit the requests to the driver again.

The device must have been opened before this function is called.

This function is a wrapper for the **IOCTL_USBIO_SET_DEVICE_POWER_STATE** operation. See also the description of **IOCTL_USBIO_SET_DEVICE_POWER_STATE** for further information.

See Also

CUsbIo::Open (page 143)
CUsbIo::GetDevicePowerState (page 183)
USBIO_DEVICE_POWER_STATE (page 124)
IOCTL_USBIO_SET_DEVICE_POWER_STATE (page 58)

CUsbIo::CancelIo

Cancels outstanding I/O requests issued by the calling thread.

Definition

```
BOOL
CancelIo();
```

Return Value

If the function succeeds the return value is TRUE, FALSE otherwise.

Comments

CancelIo cancels all outstanding I/O requests that were issued by the calling thread on the class instance. Requests issued on the instance by other threads are not cancelled.

This function is a wrapper for the Win32 function *CancelIo*. For a description of this function refer to the Win32 Platform SDK documentation.

CUsbIo::IoctlSync

Call a driver I/O control function and wait for its completion.

Definition

```
DWORD
IoctlSync(
     DWORD IoctlCode,
     const void* InBuffer,
     DWORD InBufferSize,
     void* OutBuffer,
     DWORD OutBufferSize,
     DWORD* BytesReturned
     );
```

Parameters

IoctlCode
The IOCTL code that identifies the driver function.

InBuffer
Pointer to the input buffer. The input buffer contains information to be passed to the driver. Set to NULL if no input buffer is needed.

InBufferSize
Size, in bytes, of the input buffer. Set to 0 if no input buffer is needed.

OutBuffer
Pointer to the output buffer. The output buffer receives information returned from the driver. Set to NULL if no output buffer is needed.

OutBufferSize
Size, in bytes, of the output buffer. Set to 0 if no output buffer is needed.

BytesReturned
Points to a caller-provided variable that, on a successful call, will be set to the number of bytes returned in the output buffer. Set to NULL if this information is not needed.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

IoctlSync is a generic support function that can be used to submit any IOCTL request to the USBIO device driver.

This function is used internally to handle the asynchronous USBIO API.

CUsbIo::ErrorText

Translate an USBIO error code to a description string.

Definition

```
static char*
ErrorText(
     char* StringBuffer,
     DWORD StringBufferSize,
     DWORD ErrorCode
     );
```

Parameters

`StringBuffer`
A caller-provided string buffer that receives the text. The function returns a null-terminated ASCII string.

`StringBufferSize`
Specifies the size, in bytes, of the buffer pointed to by **`StringBuffer`**.

`ErrorCode`
The error code to be translated.

Return Value

The function returns the **`StringBuffer`** pointer.

Comments

This function supports private USBIO error codes only. These codes start with a prefix of 0xE. The function cannot be used to translate general Windows error codes.

Note that ErrorText is declared static. It can be used independently of class instances.

Data Members

`HANDLE` **`FileHandle`**
This protected member contains the handle for the USBIO device object. The value is NULL if no open handle exists.

`OVERLAPPED` **`Overlapped`**
This protected member provides an OVERLAPPED data structure that is required to perform asynchronous (overlapped) I/O operations by means of the Win32 function **`DeviceIoControl`**. The Overlapped member is used by **CUsbIo::IoctlSync**.

`CRITICAL_SECTION` **`CritSect`**
This protected member provides a Win32 Critical Section object that is used to synchronize IOCTL operations on **`FileHandle`**. Synchronization is required because there is only one OVERLAPPED data structure per CUsbIo object, see also **CUsbIo::Overlapped**.

The CUsbIo object is thread-save. It can be accessed by multiple threads simultaneously.

`BOOL` **`CheckedBuildDetected`**
This protected member provides a flag. It is set to TRUE if a checked (debug) build of the USBIO driver has been detected, FALSE otherwise. The flag is initialized by the **CUsbIo::Open** function. See also **CUsbIo::GetDriverInfo** for more information.

The state of this flag can be queried by means of the function **CUsbIo::IsCheckedBuild**.

`BOOL` **`DemoVersionDetected`**
This protected member provides a flag. It is set to TRUE if a DEMO version of the USBIO driver has been detected, FALSE otherwise. The flag is initialized by the **CUsbIo::Open** function. See also **CUsbIo::GetDriverInfo** for more information.

The state of this flag can be queried by means of the function **CUsbIo::IsDemoVersion**.

`BOOL` **`LightVersionDetected`**
This protected member provides a flag. It is set to TRUE if a LIGHT version of the USBIO driver has been detected, FALSE otherwise. The flag is initialized by the **CUsbIo::Open** function. See also **CUsbIo::GetDriverInfo** for more information.

The state of this flag can be queried by means of the function **CUsbIo::IsLightVersion**.

`SP_DEVICE_INTERFACE_DETAIL_DATA*` **`mDevDetail`**
This private member contains information on the device instance opened by the CUsbIo object. It is used by the CUsbIo implementation internally.

`static CSetupApiDll` **`smSetupApi`**
This protected member is used to manage the library *setupapi.dll*. The **CSetupApiDll** class provides functions to load the system-provided library *setupapi.dll* explicitly at run-time.

CUsbIoPipe class

This class implements the interface to an USB pipe that is exported by the USBIO device driver. It provides all pipe-related functions that can be executed on a file handle that is bound to an USB endpoint. Particularly, it provides the functions needed for a data transfer from or to an endpoint.

Note that this class is derived from **CUsbIo**. All general and device-related functions can be executed on an instance of CUsbIoPipe as well.

Member Functions

CUsbIoPipe::CUsbIoPipe

Standard constructor of the CUsbIoPipe class.

Definition

```
CUsbIoPipe();
```

See Also

CUsbIoPipe::~CUsbIoPipe (page 189)

CUsbIoPipe::~CUsbIoPipe

Destructor of the CUsbIoPipe class.

Definition

```
~CUsbIoPipe();
```

See Also

CUsbIoPipe::CUsbIoPipe (page 189)

CUsbIoPipe::Bind

Bind the object to an endpoint of the USB device.

Definition

```
DWORD
Bind(
     int DeviceNumber,
     UCHAR EndpointAddress,
     HDEVINFO DeviceList = NULL,
     const GUID* InterfaceGuid = NULL
     );
```

Parameters

DeviceNumber
Specifies the index number of the USB Device. The index is zero-based. Note that the association between this number and the USB device can change with each call to **CUsbIo::CreateDeviceList**. For more details see also **CUsbIo::Open**.

Note that this parameter is ignored if the device has already been opened. See the comments below.

EndpointAddress
Specifies the address of the endpoint to bind the object to. The endpoint address is specified as reported in the corresponding endpoint descriptor.

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

Bit 7 = 0: OUT endpoint

Bit 7 = 1: IN endpoint

For example, an IN endpoint with endpoint number 1 has the endpoint address 0x81.

DeviceList
A handle to the internal device list which was returned by the function **CUsbIo::CreateDeviceList** or NULL. For more details see also **CUsbIo::Open**.

Note that this parameter is ignored if the device has already been opened. See the comments below.

InterfaceGuid
Points to a caller-provided variable of type GUID, or can be set to NULL. The provided GUID is the predefined interface GUID of the USBIO device driver, or a user-defined GUID which has to be defined in the USBIO.INF file. For more details see also **CUsbIo::Open**.

Note that this parameter is ignored if the device has already been opened. See the comments below.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

If an USB device has not already been opened this function calls **CUsbIo::Open** to attach the object to a device. It passes the parameters **`DeviceNumber`**, **`DeviceList`**, and **`InterfaceGuid`** unmodified to **CUsbIo::Open**. Thus, a device and an endpoint can be attached to the object in one step.

Alternatively, an application can attach a device first by means of **CUsbIo::Open** (derived from **CUsbIo**, and then in a second step attach an endpoint by means of Bind. The parameters **`DeviceNumber`**, **`DeviceList`**, and **`InterfaceGuid`** will be ignored in this case.

The device must be set to the configured state before an endpoint can be bound, see **CUsbIo::SetConfiguration**. Only endpoints that are included in the active configuration can be bound by this function and subsequently used for a data transfer.

Note that an instance of the CUsbIoPipe class can be bound to exactly one endpoint only. Consequently, one instance has to be created for each endpoint to be activated.

This function is a wrapper for the **IOCTL_USBIO_BIND_PIPE** operation. See also the description of **IOCTL_USBIO_BIND_PIPE** for further information.

See Also

CUsbIoPipe::Unbind (page 192)
CUsbIo::CreateDeviceList (page 141)
CUsbIo::Open (page 143)
CUsbIo::Close (page 145)
CUsbIo::SetConfiguration (page 171)
IOCTL_USBIO_BIND_PIPE (page 66)

CUsbIoPipe::Unbind

Delete the association between the object and an endpoint.

Definition

```
DWORD
Unbind();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

A call to this function causes all pending read and write requests to be aborted.

After this function was called the object can be bound to another endpoint. However, it is recommended to use a separate object for each endpoint. See also the comments on **CUsbIoPipe::Bind**.

It is not an error to call Unbind when no endpoint is currently bound. The function does nothing in this case.

Note that closing the device either by means of **CUsbIo::Close** or by destructing the object will also cause an unbind. Thus, normally there is no need to call Unbind explicitly.

This function is a wrapper for the **IOCTL_USBIO_UNBIND_PIPE** operation. See also the description of **IOCTL_USBIO_UNBIND_PIPE** for further information.

See Also

CUsbIoPipe::Bind (page 190)
CUsbIo::Close (page 145)
IOCTL_USBIO_UNBIND_PIPE (page 67)

CUsbIoPipe::Read

Submit a read request on the pipe.

Definition

```
BOOL
Read(
     CUsbIoBuf* Buf
     );
```

Parameter

Buf
Pointer to a buffer descriptor the read buffer is attached to. The buffer descriptor has to be prepared by the caller. See the comments below.

Return Value

Returns TRUE if the request was successfully submitted, FALSE otherwise. If FALSE is returned then the **Status** member of **Buf** contains an error code.

Comments

The function submits the buffer memory that is attached to **Buf** to the USBIO device driver. The caller has to prepare the buffer descriptor pointed to by **Buf**. Particularly, the **NumberOfBytesToTransfer** member has to be set to the number of bytes to read.

The call returns immediately (asynchronous behavior). After the function succeeded the read operation is pending. It will be completed later on by the USBIO driver when data is received from the device. To determine when the operation has been completed the function **CUsbIoPipe::WaitForCompletion** should be called.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

See Also

CUsbIoPipe::Bind (page 190)
CUsbIoPipe::ReadSync (page 197)
CUsbIoPipe::Write (page 194)
CUsbIoPipe::WaitForCompletion (page 195)
CUsbIoBuf (page 231)

CUsbIoPipe::Write

Submit a write request on the pipe.

Definition

```
BOOL
Write(
     CUsbIoBuf* Buf
     );
```

Parameter

Buf
Pointer to a buffer descriptor the write buffer is attached to. The buffer descriptor has to be prepared by the caller. See the comments below.

Return Value

Returns TRUE if the request was successfully submitted, FALSE otherwise. If FALSE is returned then the **Status** member of **Buf** contains an error code.

Comments

The function submits the buffer memory that is attached to **Buf** to the USBIO device driver. The buffer contains the data to be written. The caller has to prepare the buffer descriptor pointed to by **Buf**. Particularly, the **NumberOfBytesToTransfer** member has to be set to the number of bytes to write.

The call returns immediately (asynchronous behavior). After the function succeeded the write operation is pending. It will be completed later on by the USBIO driver when data has been sent to the device. To determine when the operation has been completed the function **CUsbIoPipe::WaitForCompletion** should be called.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

See Also

CUsbIoPipe::Bind (page 190)
CUsbIoPipe::WriteSync (page 199)
CUsbIoPipe::Write (page 194)
CUsbIoPipe::WaitForCompletion (page 195)
CUsbIoBuf (page 231)

CUsbIoPipe::WaitForCompletion

Wait for completion of a pending read or write operation.

Definition

```
DWORD
WaitForCompletion(
     CUsbIoBuf* Buf,
     DWORD Timeout = INFINITE
     );
```

Parameters

Buf
Pointer to the buffer descriptor that has been submitted by means of **CUsbIoPipe::Read** or **CUsbIoPipe::Write**.

Timeout
Specifies a timeout interval, in milliseconds. The function returns if the interval elapses and the read or write operation is still pending.

When INFINITE is specified then the interval never elapses. The function does not return until the read or write operation is finished.

Return Value

The function returns **USBIO_ERR_TIMEOUT** if the timeout interval elapsed. If the read or write operation has been finished the return value is the final completion status of the operation. Note that the **Status** member of **Buf** will also be set to the final completion status. The completion status is 0 if the read or write operation has been successfully finished, an USBIO error code otherwise.

Comments

After a buffer was submitted to the USBIO device driver by means of **CUsbIoPipe::Read** or **CUsbIoPipe::Write** this function is used to wait for the completion of the data transfer. Note that WaitForCompletion can be called regardless of the return status of the **CUsbIoPipe::Read** or **CUsbIoPipe::Write** function. It returns always the correct status of the buffer.

Optionally, a timeout interval for the wait operation may be specified. When the interval elapses before the read or write operation is finished the function returns with a special status of **USBIO_ERR_TIMEOUT**. The data transfer operation is still pending in this case. WaitForCompletion should be called again until the operation is finished.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

See Also

CUsbIoPipe::Bind (page 190)
CUsbIoPipe::Read (page 193)
CUsbIoPipe::Write (page 194)
CUsbIoBuf (page 231)

CUsbIoPipe::ReadSync

Submit a read request on the pipe and wait for its completion.

Definition

```
DWORD
ReadSync(
     void* Buffer,
     DWORD& ByteCount,
     DWORD Timeout = INFINITE
     );
```

Parameters

Buffer
Pointer to a caller-provided buffer. The buffer receives the data transferred from the device.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Buffer**. After the function succeeds **ByteCount** contains the number of bytes successfully read.

Timeout
Specifies a timeout interval, in milliseconds. If the interval elapses and the read operation is not yet finished the function aborts the operation and returns with **USBIO_ERR_TIMEOUT**.

When INFINITE is specified then the interval never elapses. The function does not return until the read operation is finished.

Return Value

The function returns **USBIO_ERR_TIMEOUT** if the timeout interval elapsed and the read operation was aborted. If the read operation has been finished the return value is the completion status of the operation which is 0 for success, or an USBIO error code otherwise.

Comments

The function transfers data from the endpoint attached to the object to the specified buffer. The function does not return to the caller until the data transfer has been finished or aborted due to a timeout. It behaves in a synchronous manner.

Optionally, a timeout interval for the synchronous read operation may be specified. When the interval elapses before the operation is finished the function aborts the operation and returns with a special status of **USBIO_ERR_TIMEOUT**. In this case, it is not possible to

determine the number of bytes already transferred. After a timeout error occurred **CUsbIoPipe::ResetPipe** should be called.

Note that there is some overhead involved when this function is used. This is due to a temporary Win32 Event object that is created and destroyed internally.

Note: Using synchronous read requests does make sense in rare cases only and can lead to unpredictable results. It is recommended to handle read operations asynchronously by means of **CUsbIoPipe::Read** and **CUsbIoPipe::WaitForCompletion**.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

See Also

CUsbIoPipe::Bind (page 190)
CUsbIoPipe::WriteSync (page 199)
CUsbIoPipe::Read (page 193)
CUsbIoPipe::WaitForCompletion (page 195)
CUsbIoPipe::ResetPipe (page 201)

CUsbIoPipe::WriteSync

Submit a write request on the pipe and wait for its completion.

Definition

```
DWORD
WriteSync(
     void* Buffer,
     DWORD& ByteCount,
     DWORD Timeout = INFINITE
     );
```

Parameters

Buffer
Pointer to a caller-provided buffer that contains the data to be transferred to the device.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Buffer**. This is the number of bytes to be transferred to the device. After the function succeeds **ByteCount** contains the number of bytes successfully written.

Timeout
Specifies a timeout interval, in milliseconds. If the interval elapses and the write operation is not yet finished the function aborts the operation and returns with **USBIO_ERR_TIMEOUT**.

When INFINITE is specified then the interval never elapses. The function does not return until the write operation is finished.

Return Value

The function returns **USBIO_ERR_TIMEOUT** if the timeout interval elapsed and the write operation was aborted. If the write operation has been finished the return value is the completion status of the operation which is 0 for success, or an USBIO error code otherwise.

Comments

The function transfers data from the specified buffer to the endpoint attached to the object. The function does not return to the caller until the data transfer has been finished or aborted due to a timeout. It behaves in a synchronous manner.

Optionally, a timeout interval for the synchronous write operation may be specified. When the interval elapses before the operation is finished the function aborts the operation and returns with a special status of **USBIO_ERR_TIMEOUT**. In this case, it is not possible to determine the number of bytes already transferred. After a timeout error occurred **CUsbIoPipe::ResetPipe** should be called.

Note that there is some overhead involved when this function is used. This is due to a temporary Win32 Event object that is created and destroyed internally.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

See Also

CUsbIoPipe::Bind (page 190)
CUsbIoPipe::ReadSync (page 197)
CUsbIoPipe::ResetPipe (page 201)

CUsbIoPipe::ResetPipe

Reset pipe.

Definition

```
DWORD
ResetPipe();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function resets the software state of a pipe in the USB driver stack. Besides, on a bulk or interrupt pipe a CLEAR_FEATURE Endpoint Stall request will be generated on the USB. This should reset the endpoint state in the device as well.

This function has to be used after an error condition occurred on the pipe and the pipe was halted by the USB drivers.

It is recommended to call ResetPipe every time a data transfer is initialized on the pipe.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_RESET_PIPE** operation. See also the description of **IOCTL_USBIO_RESET_PIPE** for further information.

See Also

CUsbIoPipe::Bind (page 190)
CUsbIoPipe::AbortPipe (page 202)
IOCTL_USBIO_RESET_PIPE (page 68)

CUsbIoPipe::AbortPipe

Cancel all pending read and write requests on this pipe.

Definition

```
DWORD
AbortPipe();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function is used to abort pending I/O operations on the pipe. All pending buffers will be returned to the application with an error status. Note that it is not possible to determine the number of bytes already transferred from or to an aborted buffer.

After a call to this function and before the data transfer is restarted the state of the pipe should be reset by means of **CUsbIoPipe::ResetPipe**. See also the comments on **CUsbIoPipe::ResetPipe**.

Note that it will take some milliseconds to cancel all buffers. Therefore, AbortPipe should not be called periodically.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_ABORT_PIPE** operation. See also the description of **IOCTL_USBIO_ABORT_PIPE** for further information.

See Also

CUsbIoPipe::Bind (page 190)
CUsbIoPipe::ResetPipe (page 201)
IOCTL_USBIO_ABORT_PIPE (page 69)

CUsbIoPipe::GetPipeParameters

Query pipe-related parameters from the USBIO device driver.

Definition

```
DWORD
GetPipeParameters(
     USBIO_PIPE_PARAMETERS* PipeParameters
     );
```

Parameter

PipeParameters
Points to a caller-provided variable that receives the current parameter settings.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_GET_PIPE_PARAMETERS** operation. See also the description of **IOCTL_USBIO_GET_PIPE_PARAMETERS** for further information.

See Also

CUsbIoPipe::Bind (page 190)
CUsbIoPipe::SetPipeParameters (page 204)
USBIO_PIPE_PARAMETERS (page 110)
IOCTL_USBIO_GET_PIPE_PARAMETERS (page 70)

CUsbIoPipe::SetPipeParameters

Set pipe-related parameters in the USBIO device driver.

Definition

```
DWORD
SetPipeParameters(
     const USBIO_PIPE_PARAMETERS* PipeParameters
     );
```

Parameter

PipeParameters
Points to a caller-provided variable that specifies the parameters to be set.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_SET_PIPE_PARAMETERS** operation. See also the description of **IOCTL_USBIO_SET_PIPE_PARAMETERS** for further information.

See Also

CUsbIoPipe::Bind (page 190)
CUsbIoPipe::GetPipeParameters (page 203)
USBIO_PIPE_PARAMETERS (page 110)
IOCTL_USBIO_SET_PIPE_PARAMETERS (page 71)

CUsbIoPipe::PipeControlTransferIn

Generates a control transfer (SETUP token) on the pipe with a data phase in device to host (IN) direction.

Definition

```
DWORD
PipeControlTransferIn(
     void* Buffer,
     DWORD& ByteCount,
     const USBIO_PIPE_CONTROL_TRANSFER* ControlTransfer
     );
```

Parameters

Buffer
Pointer to a caller-provided buffer. The buffer receives the data transferred in the IN data phase.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Buffer**. After the function successfully returned **ByteCount** contains the number of valid bytes returned in the buffer.

ControlTransfer
Pointer to a caller-provided variable that defines the request to be generated.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function is used to send a SETUP token to a Control type endpoint.

Note: This function cannot be used to send a SETUP request to the default endpoint 0.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN** operation. See also the description of **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN** for further information.

See Also

CUsbIoPipe::Bind (page 190)

CUsbIoPipe::PipeControlTransferOut (page 207)
USBIO_PIPE_CONTROL_TRANSFER (page 116)
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN (page 76)

CUsbIoPipe::PipeControlTransferOut

Generates a control transfer (SETUP token) on the pipe with a data phase in host to device (OUT) direction.

Definition

```
DWORD
PipeControlTransferOut(
     const void* Buffer,
     DWORD& ByteCount,
     const USBIO_PIPE_CONTROL_TRANSFER* ControlTransfer
     );
```

Parameters

Buffer
Pointer to a caller-provided buffer that contains the data to be transferred in the OUT data phase.

ByteCount
When the function is called **ByteCount** specifies the size, in bytes, of the buffer pointed to by **Buffer**. This is the number of bytes to be transferred in the data phase. After the function successfully returned **ByteCount** contains the number of bytes transferred.

ControlTransfer
Pointer to a caller-provided variable that defines the request to be generated.

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

This function is used to send a SETUP token to a Control type endpoint.

Note: This function cannot be used to send a SETUP request to the default endpoint 0.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT** operation. See also the description of **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT** for further information.

See Also

CUsbIoPipe::Bind (page 190)

CUsbIoPipe::PipeControlTransferIn (page 205)
USBIO_PIPE_CONTROL_TRANSFER (page 116)
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT (page 77)

CUsbIoPipe::SetupPipeStatistics

Enables or disables a statistical analysis of the data transfer on the pipe.

Definition

```
DWORD
SetupPipeStatistics(
     ULONG AveragingInterval
     );
```

Parameter

AveragingInterval
Specifies the time interval, in milliseconds, that is used to calculate the average data rate of the pipe. A time averaging algorithm is used to continuously compute the mean value of the data transfer rate.

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

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

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

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_SETUP_PIPE_STATISTICS** operation. See also the description of **IOCTL_USBIO_SETUP_PIPE_STATISTICS** for further information.

See Also

CUsbIoPipe::Bind (page 190)
CUsbIoPipe::QueryPipeStatistics (page 210)
IOCTL_USBIO_SETUP_PIPE_STATISTICS (page 72)

CUsbIoPipe::QueryPipeStatistics

Returns statistical data related to the pipe.

Definition

```
DWORD
QueryPipeStatistics(
     USBIO_PIPE_STATISTICS* PipeStatistics,
     ULONG Flags = 0
     );
```

Parameters

PipeStatistics
Points to a caller-provided variable that receives the statistical data.

Flags
This parameter is set to zero or any combination (bit-wise or) of the following values.

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

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

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

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

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

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

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

Note that the statistical data is maintained for each pipe object separately.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

This function is a wrapper for the **IOCTL_USBIO_QUERY_PIPE_STATISTICS** operation. See also the description of **IOCTL_USBIO_QUERY_PIPE_STATISTICS** for further information.

See Also

CUsbIoPipe::Bind (page 190)
CUsbIoPipe::SetupPipeStatistics (page 209)
USBIO_PIPE_STATISTICS (page 114)
IOCTL_USBIO_QUERY_PIPE_STATISTICS (page 74)

CUsbIoPipe::ResetPipeStatistics

Reset the statistics counters of the pipe.

Definition

```
DWORD
ResetPipeStatistics();
```

Return Value

The function returns 0 if successful, an USBIO error code otherwise.

Comments

The USBIO driver internally maintains some statistical data per pipe object. This function resets the counters BytesTransferred, RequestsSucceeded, and RequestsFailed to zero. See also **CUsbIoPipe::SetupPipeStatistics** and **USBIO_PIPE_STATISTICS** for more information on pipe statistics.

Note that this function calls **CUsbIoPipe::QueryPipeStatistics** to reset the counters.

The device must have been opened and the object must have been bound to an endpoint before this function is called, see **CUsbIoPipe::Bind**.

See Also

CUsbIoPipe::Bind (page 190)
CUsbIoPipe::SetupPipeStatistics (page 209)
CUsbIoPipe::QueryPipeStatistics (page 210)
USBIO_PIPE_STATISTICS (page 114)

CUsbIoThread class

This class provides a basic implementation of a worker-thread that is used to continuously perform I/O operations. CUsbIoThread is a base class for the **CUsbIoReader** and **CUsbIoWriter** worker-thread implementations.

The CUsbIoThread class contains pure virtual functions. Consequently, it is not possible to create an instance of the class.

Note that CUsbIoThread is derived from **CUsbIoPipe**. Thus, all USBIO functions can be executed on an instance of CUsbIoThread.

Member Functions

CUsbIoThread::CUsbIoThread

Constructs a CUsbIoThread object.

Definition

```
CUsbIoThread();
```

See Also

CUsbIoThread::˜CUsbIoThread (page 213)

CUsbIoThread::˜CUsbIoThread

Destructor of the CUsbIoThread class.

Definition

```
virtual
˜CUsbIoThread();
```

Comments

The internal worker-thread must have been terminated when the object's destructor is called. That means **CUsbIoThread::ShutdownThread** must be called before the object is destroyed.

See Also

CUsbIoThread::CUsbIoThread (page 213)
CUsbIoThread::ShutdownThread (page 217)

CUsbIoThread::AllocateBuffers

Allocate the internal buffer pool.

Definition

```
BOOL
AllocateBuffers(
     DWORD SizeOfBuffer,
     DWORD NumberOfBuffers
     );
```

Parameters

SizeOfBuffer
Specifies the size, in bytes, of the buffers to be allocated internally.

NumberOfBuffers
Specifies the number of buffers to be allocated internally.

Return Value

Returns TRUE in case of success, FALSE otherwise.

Comments

The function initializes an internal **CUsbIoBufPool** object. For more information on the parameters and the behavior of the function refer to the description of **CUsbIoBufPool::Allocate**.

See Also

CUsbIoThread::FreeBuffers (page 215)
CUsbIoBufPool (page 239)
CUsbIoBufPool::Allocate (page 241)

CUsbIoThread::FreeBuffers

Free the internal buffer pool.

Definition

```
void
FreeBuffers();
```

Comments

The function frees the buffers allocated by the internal **CUsbIoBufPool** object. For more information on the behavior of the function refer to the description of **CUsbIoBufPool::Free**.

See Also

CUsbIoThread::AllocateBuffers (page 214)
CUsbIoBufPool (page 239)
CUsbIoBufPool::Free (page 242)

CUsbIoThread::StartThread

Start the internal worker-thread.

Definition

```
BOOL
StartThread(
     DWORD MaxIoErrorCount = 3
     );
```

Parameter

MaxIoErrorCount
Specifies the maximum number of I/O errors caused by read or write operations that will be tolerated by the thread. The thread will terminate itself when the specified limit is reached.

Return Value

Returns TRUE in case of success, FALSE otherwise.

Comments

The internal worker-thread will be created. Possibly, it starts its execution before this function returns.

The error limit specified in **MaxIoErrorCount** prevents an end-less loop in the worker-thread that can occur when the device permanently fails data transfer requests.

The internal buffer pool must have been initialized by means of **CUsbIoThread::AllocateBuffers** before this function is called.

Note: The internal worker-thread is a native system thread. That means it cannot call MFC (Microsoft Foundation Classes) functions. It is necessary to use *PostMessage*, *SendMessage* or some other communication mechanism to switch over to MFC-aware threads.

See Also

CUsbIoThread::AllocateBuffers (page 214)
CUsbIoThread::ThreadRoutine (page 222)
CUsbIoThread::ShutdownThread (page 217)

CUsbIoThread::ShutdownThread

Terminate the internal worker-thread.

Definition

```
BOOL
ShutdownThread();
```

Return Value

Returns TRUE in case of success, FALSE otherwise.

Comments

The function sets the member variable **`TerminateFlag`** to TRUE. Then it calls the virtual member function **CUsbIoThread::TerminateThread**. The implementation of **CUsbIoThread::TerminateThread** should cause the worker-thread to resume from a wait function and to terminate itself.

ShutdownThread blocks until the worker-thread has been terminated by the operating system.

It is not an error to call ShutdownThread when the internal thread is not started. The function does nothing in this case.

Note: This function has to be called before the CUsbIoThread object is destroyed. In other words, the worker-thread must have been terminated when the object's destructor **CUsbIoThread::~CUsbIoThread** is called.

See Also

CUsbIoThread::StartThread (page 216)
CUsbIoThread::~CUsbIoThread (page 213)
CUsbIoThread::TerminateThread (page 223)

CUsbIoThread::ProcessData

This handler is called by the worker-thread to process data that has been received from the device.

Definition

```
virtual void
ProcessData(
     CUsbIoBuf* Buf
     );
```

Parameter

Buf
Pointer to a buffer descriptor the buffer is attached to.

Comments

This handler function is used by the **CUsbIoReader** implementation of the worker-thread. It is called when data has been successfully received on the pipe. Note that the function is called in the context of the worker-thread.

There is a default implementation of ProcessData that is just empty. ProcessData should be overloaded by a derived class to implement a specific functionality.

An implementation of ProcessData should examine the fields **CUsbIoBuf::Status** and **CUsbIoBuf::BytesTransferred** to determine if there are valid data bytes in the buffer.

See Also

CUsbIoThread::StartThread (page 216)
CUsbIoThread::ProcessBuffer (page 219)
CUsbIoReader (page 225)
CUsbIoBuf (page 231)
CUsbIoBuf::Status (page 237)
CUsbIoBuf::BytesTransferred (page 237)

CUsbIoThread::ProcessBuffer

This handler is called by the worker-thread if a buffer must be filled with data before it will be submitted on the pipe.

Definition

```
virtual void
ProcessBuffer(
     CUsbIoBuf* Buf
     );
```

Parameter

Buf
Pointer to a buffer descriptor the buffer is attached to.

Comments

This handler function is used by the **CUsbIoWriter** implementation of the worker-thread. It is called when a buffer has to be filled before it will be submitted on the pipe. Note that the function is called in the context of the worker-thread.

There is a default implementation of ProcessBuffer. It fills the buffer with zeroes and sets the **CUsbIoBuf::NumberOfBytesToTransfer** member of **Buf** to the buffer's size. ProcessBuffer should be overloaded by a derived class to implement a specific functionality.

See Also

CUsbIoThread::StartThread (page 216)
CUsbIoThread::ProcessData (page 218)
CUsbIoWriter (page 228)
CUsbIoBuf (page 231)
CUsbIoBuf::NumberOfBytesToTransfer (page 237)

CUsbIoThread::BufErrorHandler

This handler is called by the worker-thread when a read or write operation has been completed with an error status.

Definition

```
virtual void
BufErrorHandler(
     CUsbIoBuf* Buf
     );
```

Parameter

Buf
Pointer to a buffer descriptor the failed buffer is attached to.

Comments

This handler function is used by both the **CUsbIoReader** and **CUsbIoWriter** implementation of the worker-thread. It is called when a read or write request failed. Note that the function is called in the context of the worker-thread.

There is a default implementation of BufErrorHandler that is just empty. BufErrorHandler should be overloaded by a derived class to implement a specific error handling.

An implementation of BufErrorHandler should examine the field **CUsbIoBuf::Status** to determine the reason for failing the read or write request.

See Also

CUsbIoThread::StartThread (page 216)
CUsbIoThread::ProcessData (page 218)
CUsbIoThread::ProcessBuffer (page 219)
CUsbIoReader (page 225)
CUsbIoWriter (page 228)
CUsbIoBuf (page 231)
CUsbIoBuf::Status (page 237)

CUsbIoThread::OnThreadExit

This notification handler is called by the worker-thread before the thread terminates itself.

Definition

```
virtual void
OnThreadExit();
```

Comments

The function is called in the context of the worker-thread.

There is a default implementation of OnThreadExit that is just empty. OnThreadExit can be overloaded by a derived class to implement a specific behavior.

See Also

CUsbIoThread::StartThread (page 216)
CUsbIoThread::ShutdownThread (page 217)

CUsbIoThread::ThreadRoutine

The main routine that is executed by the worker-thread.

Definition

```
virtual void
ThreadRoutine() = 0;
```

Comments

The function is declared pure virtual. Consequently, it must be implemented by a derived class.

The derived classes **CUsbIoReader** and **CUsbIoWriter** implement this function.

See Also

CUsbIoThread::StartThread (page 216)
CUsbIoThread::ShutdownThread (page 217)
CUsbIoReader (page 225)
CUsbIoWriter (page 228)

CUsbIoThread::TerminateThread

This routine is called by **CUsbIoThread::ShutdownThread** to terminate the internal worker-thread.

Definition

```
virtual void
TerminateThread() = 0;
```

Comments

The function is declared pure virtual. Consequently, it must be implemented by a derived class. Note that TerminateThread is called in the context of **CUsbIoThread::ShutdownThread**.

The derived classes **CUsbIoReader** and **CUsbIoWriter** implement this function.

See Also

CUsbIoThread::ShutdownThread (page 217)
CUsbIoReader (page 225)
CUsbIoWriter (page 228)

Data Members

`CUsbIoBufPool` **`BufPool`**
The internal buffer pool, see **CUsbIoThread::AllocateBuffers** and **CUsbIoThread::FreeBuffers**.

`HANDLE` **`ThreadHandle`**
The handle that identifies the worker-thread. The value is NULL if the worker-thread is not started.

`unsigned int` **`ThreadID`**
The thread ID assigned by the operating system for the worker-thread.

`volatile BOOL` **`TerminateFlag`**
This flag will be set to TRUE by **CUsbIoThread::ShutdownThread** to indicate that the worker-thread shall terminate itself.

`DWORD` **`MaxErrorCount`**
An error limit for the worker-thread's main loop. For a description see **CUsbIoThread::StartThread**.

`CUsbIoBuf*` **`FirstPending`**
Used by **CUsbIoReader** and **CUsbIoWriter** to implement a list of currently pending buffers.

`CUsbIoBuf*` **`LastPending`**
Used by **CUsbIoReader** and **CUsbIoWriter** to implement a list of currently pending buffers.

CUsbIoReader class

This class implements a worker-thread that continuously reads a data stream from a pipe.

Note that this class is derived from **CUsbIoThread** which provides the basic handling of the internal worker-thread.

Member Functions

CUsbIoReader::CUsbIoReader

Constructs a CUsbIoReader object.

Definition

```
CUsbIoReader();
```

See Also

CUsbIoReader::˜CUsbIoReader (page 225)

CUsbIoReader::˜CUsbIoReader

Destructor of the CUsbIoReader class.

Definition

```
virtual
˜CUsbIoReader();
```

Comments

The internal worker-thread must have been terminated when the object's destructor is called. That means **CUsbIoThread::ShutdownThread** must be called before the object is destroyed.

See Also

CUsbIoReader::CUsbIoReader (page 225)
CUsbIoThread::ShutdownThread (page 217)

CUsbIoReader::ThreadRoutine

The main routine that is executed by the worker-thread.

Definition

```
virtual void
ThreadRoutine();
```

Comments

This function implements the main loop of the worker-thread. It submits all buffers from the internal buffer pool to the driver and waits for the completion of the first buffer.

ThreadRoutine can be overloaded by a derived class to implement a different behavior.

See Also

CUsbIoThread::ThreadRoutine (page 222)
CUsbIoThread (page 213)
CUsbIoReader::TerminateThread (page 227)

CUsbIoReader::TerminateThread

This routine is called by **CUsbIoThread** when the worker-thread shall be terminated.

Definition

```
virtual void
TerminateThread();
```

Comments

The implementation of this function calls **CUsbIoPipe::AbortPipe**. This will cancel all pending read operations and cause the worker-thread to resume.

TerminateThread can be overloaded by a derived class to implement a different behavior.

See Also

CUsbIoThread::TerminateThread (page 223)
CUsbIoThread (page 213)
CUsbIoReader::ThreadRoutine (page 226)

CUsbIoWriter class

This class implements a worker-thread that continuously writes a data stream to a pipe.

Note that this class is derived from **CUsbIoThread** which provides the basic handling of the internal worker-thread.

Member Functions

CUsbIoWriter::CUsbIoWriter

Constructs a CUsbIoWriter object.

Definition

```
CUsbIoWriter();
```

See Also

CUsbIoWriter::˜CUsbIoWriter (page 228)

CUsbIoWriter::˜CUsbIoWriter

Destructor of the CUsbIoWriter class.

Definition

```
virtual
˜CUsbIoWriter();
```

Comments

The internal worker-thread must have been terminated when the object's destructor is called. That means **CUsbIoThread::ShutdownThread** must be called before the object is destroyed.

See Also

CUsbIoWriter::CUsbIoWriter (page 228)
CUsbIoThread::ShutdownThread (page 217)

CUsbIoWriter::ThreadRoutine

The main routine that is executed by the worker-thread.

Definition

```
virtual void
ThreadRoutine();
```

Comments

This function implements the main loop of the worker-thread. It submits all buffers from the internal buffer pool to the driver and waits for the completion of the first buffer.

ThreadRoutine can be overloaded by a derived class to implement a different behavior.

See Also

CUsbIoThread::ThreadRoutine (page 222)
CUsbIoThread (page 213)
CUsbIoWriter::TerminateThread (page 230)

CUsbIoWriter::TerminateThread

This routine is called by **CUsbIoThread** when the worker-thread shall be terminated.

Definition

```
virtual void
TerminateThread();
```

Comments

The implementation of this function calls **CUsbIoPipe::AbortPipe**. This will cancel all pending read operations and cause the worker-thread to resume.

TerminateThread can be overloaded by a derived class to implement a different behavior.

See Also

CUsbIoThread::TerminateThread (page 223)
CUsbIoThread (page 213)
CUsbIoWriter::ThreadRoutine (page 229)

CUsbIoBuf class

This class is used as a buffer descriptor of buffers used for read and write operations.

Member Functions

CUsbIoBuf::CUsbIoBuf

Construct a CUsbIoBuf object.

Definition

```
CUsbIoBuf();
```

Comments

This is the default constructor. It creates an empty descriptor. No buffer is attached.

See Also

CUsbIoBuf::˜CUsbIoBuf (page 234)

CUsbIoBuf::CUsbIoBuf

Construct a CUsbIoBuf object and attach an existing buffer.

Definition

```
CUsbIoBuf(
     void* Buffer,
     DWORD BufferSize
     );
```

Parameters

Buffer
Points to a caller-provided buffer to be attached to the descriptor object.

BufferSize
Specifies the size, in bytes, of the buffer to be attached to the descriptor object.

See Also

CUsbIoBuf::˜CUsbIoBuf (page 234)

CUsbIoBuf::CUsbIoBuf

Construct a CUsbIoBuf object and allocate a buffer internally.

Definition

```
CUsbIoBuf(
     DWORD BufferSize
     );
```

Parameter

BufferSize
Specifies the size, in bytes, of the buffer to be allocated and attached to the descriptor object.

Comments

This constructor allocates a buffer of the specified size and attaches it to the descriptor object. The buffer will be automatically freed by the destructor of this class.

See Also

CUsbIoBuf::˜CUsbIoBuf (page 234)

CUsbIoBuf::~CUsbIoBuf

Destructor for a CUsbIoBuf object.

Definition

```
~CUsbIoBuf();
```

Comments

The destructor frees a buffer that was allocated by a constructor. A buffer that has been attached after construction will not be freed.

See Also

CUsbIoBuf::CUsbIoBuf (page 231)

CUsbIoBuf::Buffer

Get buffer pointer.

Definition

```
void*
Buffer();
```

Return Value

The function returns a pointer to the first byte of the buffer that is attached to the descriptor object. The return value is NULL if no buffer is attached.

See Also

CUsbIoBuf::Size (page 236)

CUsbIoBuf::Size

Get buffer size, in bytes.

Definition

```
DWORD
Size();
```

Return Value

The function returns the size, in bytes, of the buffer that is attached to the descriptor object. The return value is 0 if no buffer is attached.

See Also

CUsbIoBuf::Buffer (page 235)

Data Members

`DWORD` **`NumberOfBytesToTransfer`**
This public member specifies the number of bytes to be transferred to or from the buffer in a subsequent read or write operation.

Note that this member has to be set before the read or write operation is initiated by means of **CUsbIoPipe::Read** or **CUsbIoPipe::Write**.

`DWORD` **`BytesTransferred`**
This public member indicates the number of bytes successfully transferred to or from the buffer during a read or write operation.

Note that this member will be set after a read or write operation is completed.

`DWORD` **`Status`**
This public member indicates the completion status of a read or write operation.

Note that this member will be set after a read or write operation is completed.

`CUsbIoBuf*` **`Next`**
This public member allows to build a chain of buffer descriptor objects. It is used by **CUsbIoBufPool**, **CUsbIoReader**, and **CUsbIoWriter** to manage buffer lists.

`BOOL` **`OperationFinished`**
This public member is used as a flag. If it is set to TRUE then it indicates that the data transfer operation is finished altogether. Read or write processing will be terminated by **CUsbIoReader** or **CUsbIoWriter**.

`DWORD` **`Context`**
This public member is a general purpose field. It will never be touched by any class in the USBIO class library. Thus, it can be used by an application to store a context value that it associates with the buffer object.

`void*` **`BufferMem`**
This protected member contains the address of the memory block that is attached to the CUsbIoBuf object. A value of NULL indicates that no memory block is attached.

`DWORD` **`BufferSize`**
This protected member contains the size, in bytes, of the memory block that is attached to the CUsbIoBuf object. The value is zero if no memory block is attached.

`OVERLAPPED` **`Overlapped`**
This protected member provides the OVERLAPPED data structure that is required to perform asynchronous (overlapped) I/O operations by means of the Win32 functions **`ReadFile`**, **`WriteFile`**, and **`DeviceIoControl`**. One instance of the OVERLAPPED structure is required per I/O buffer. The data structure stores context information while the asynchronous I/O operation is in progress. Most important, it provides a Win32 Event object that signals the completion of the asynchronous operation.

This member is used by **CUsbIoReader** and **CUsbIoWriter** to perform I/O operations.

BOOL **BufferMemAllocated**

This protected member provides a flag. It is set to TRUE if the memory block that is attached to the CUsbIoBuf object was allocated by a constructor of the class. The memory block has to be freed by the destructor in this case. The flag is set to FALSE if the memory block that is attached to the CUsbIoBuf object was provided by the user.

CUsbIoBufPool class

This class implements a pool of CUsbIoBuf objects. It is used by **CUsbIoReader** and **CUsbIoWriter** to simplify management of buffer pools.

Member Functions

CUsbIoBufPool::CUsbIoBufPool

Construct a CUsbIoBufPool object.

Definition

```
CUsbIoBufPool();
```

Comments

This is the default constructor. It creates an empty pool.

See Also

CUsbIoBufPool::~CUsbIoBufPool (page 240)

CUsbIoBufPool::~CUsbIoBufPool

Destructor for a CUsbIoBufPool object.

Definition

```
~CUsbIoBufPool();
```

Comments

The destructor frees all **CUsbIoBuf** objects allocated by the pool.

See Also

CUsbIoBufPool::CUsbIoBufPool (page 240)

CUsbIoBufPool::Allocate

Allocate all elements of the buffer pool.

Definition

```
BOOL
Allocate(
     DWORD SizeOfBuffer,
     DWORD NumberOfBuffers
     );
```

Parameters

SizeOfBuffer
Specifies the size, in bytes, of the buffers to be allocated internally.

NumberOfBuffers
Specifies the number of buffers to be allocated internally.

Return Value

Returns TRUE in case of success, FALSE otherwise.

Comments

The function allocates the required number of buffer descriptors (**CUsbIoBuf** objects). Then it allocates the specified amount of buffer memory. The total number of bytes to allocate is calculated as follows.

*TotalSize = NumberOfBuffers * SizeOfBuffer*

In a last step the buffers are attached to the descriptors and stored in an internal list.

The function fails by returning FALSE when an internal pool is already allocated. **CUsbIoBufPool::Free** has to be called before a new pool can be allocated.

See Also

CUsbIoBufPool::Free (page 242)

CUsbIoBufPool::Free

Free all elements of the buffer pool.

Definition

```
void
Free();
```

Comments

The function frees all buffer descriptors and all the buffer memory allocated by a call to **CUsbIoBufPool::Allocate**. The pool is empty after this call.

A call to Free on an empty pool is allowed. The function does nothing in this case.

Note that after a call to Free, another pool can be allocated by means of **CUsbIoBufPool::Allocate**.

See Also

CUsbIoBufPool::Allocate (page 241)

CUsbIoBufPool::Get

Get a buffer from the pool.

Definition

```
CUsbIoBuf*
Get();
```

Return Value

The function returns a pointer to the buffer descriptor removed from the pool, or NULL if the pool is exhausted.

Comments

The function removes a buffer from the pool and returns a pointer to the associated descriptor. The caller is responsible for releasing the buffer, see **CUsbIoBufPool::Put**.

See Also

CUsbIoBufPool::Put (page 244)
CUsbIoBufPool::CurrentCount (page 245)

CUsbIoBufPool::Put

Put a buffer back to the pool.

Definition

```
void
Put(
     CUsbIoBuf* Buf
     );
```

Parameter

Buf
Pointer to the buffer descriptor that was returned by **CUsbIoBufPool::Get**.

Comments

This function is called to release a buffer that was returned by **CUsbIoBufPool::Get**.

See Also

CUsbIoBufPool::Get (page 243)
CUsbIoBufPool::CurrentCount (page 245)

CUsbIoBufPool::CurrentCount

Get current number of buffers in the pool.

Definition

```
long
CurrentCount();
```

Return Value

The function returns the current number of buffers stored in the pool.

See Also

CUsbIoBufPool::Get (page 243)
CUsbIoBufPool::Put (page 244)

Data Members

`CRITICAL_SECTION` **`CritSect`**
This protected member provides a Win32 Critical Section object that is used to synchronize access to the members of the class. Thus, the CUsbIoBufPool object is thread-save. It can be accessed by multiple threads simultaneously.

`CUsbIoBuf*` **`Head`**
This protected member points to the first buffer object that is available in the pool. The buffer pool is managed by means of a single-linked list. This member points to the element at the head of the list. The **CUsbIoBuf** objects are linked by means of their **`Next`** member. The list is terminated by a **`Next`** pointer that is set to NULL.

Head is set to NULL if the buffer list is empty.

`long` **`Count`**
This protected member contains the number of buffers that are currently linked to the pool.

`CUsbIoBuf*` **`BufArray`**
This protected member points to the array of CUsbIoBuf objects that are allocated by the pool internally.

`char*` **`BufferMemory`**
This protected member points to the buffer memory block that is allocated by the pool internally.

CSetupApiDll class

This class provides a mean to load the system-provided *setupapi.dll* explicitly. The method is also called Run-Time Dynamic Linking.

The library *setupapi.dll* is part of the Win32 API and provides functions for managing Plug&Play devices. It is supported by Windows 98 and later systems. The file *setupapi.dll* is not available on older systems like Windows 95 and Windows NT. Therefore, if an application is implicitly linked to *setupapi.dll* then it will not load on older systems. In order to avoid such kind of problems the DLL should be loaded explicitly at run-time. The CSetupApiDll class provides the appropriate implementation.

Member Functions

CSetupApiDll::CSetupApiDll

Construct a CSetupApiDll object.

Definition

```
CSetupApiDll();
```

Comments

This is the default constructor. It initializes the object.

CSetupApiDll::~CSetupApiDll

Destructor for a CSetupApiDll object.

Definition

```
~CSetupApiDll();
```

Comments

The destructor frees the *setupapi.dll* library if loaded.

CSetupApiDll::Load

Load the system-provided library *setupapi.dll*.

Definition

```
BOOL
Load();
```

Return Value

The function returns TRUE if successful, FALSE otherwise.

Comments

The function loads the DLL and if this was successful then it initializes all function pointers to contain the address of the appropriate function.

The function can safely be called repeatedly. It returns TRUE if the *setupapi.dll* is already loaded.

See Also

CSetupApiDll::Release (page 249)

CSetupApiDll::Release

Release the *setupapi.dll* library.

Definition

```
void
Release();
```

Comments

The function frees the DLL and invalidates all function pointers.

The function can safely be called if the DLL is not loaded. It does not perform any operation in this case.

See Also

CSetupApiDll::Load (page 248)

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 52)
- **IOCTL_USBIO_SET_DEVICE_PARAMETERS** (page 53)

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 38)

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 38)
- **IOCTL_USBIO_GET_CONFIGURATION** (page 43)
- **IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR** (page 45)
- **IOCTL_USBIO_SET_CONFIGURATION** (page 46)
- **IOCTL_USBIO_UNCONFIGURE_DEVICE** (page 47)

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 48)
- **IOCTL_USBIO_GET_INTERFACE** (page 44)

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 54)
- **IOCTL_USBIO_BIND_PIPE** (page 66)
- **IOCTL_USBIO_UNBIND_PIPE** (page 67)

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 49)
- **IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST** (page 50)

6.1.7 Feature

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

Related driver interfaces:

- **IOCTL_USBIO_SET_FEATURE** (page 40)
- **IOCTL_USBIO_CLEAR_FEATURE** (page 41)

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 42)
- **IOCTL_USBIO_RESET_DEVICE** (page 55)
- **IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER** (page 57)
- **IOCTL_USBIO_SET_DEVICE_POWER_STATE** (page 58)
- **IOCTL_USBIO_GET_DEVICE_POWER_STATE** (page 59)

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 68)
- **IOCTL_USBIO_ABORT_PIPE** (page 69)
- **IOCTL_USBIO_GET_PIPE_PARAMETERS** (page 70)
- **IOCTL_USBIO_SET_PIPE_PARAMETERS** (page 71)

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 a USB device.

Related driver interfaces:

- **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN** (page 76)
- **IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT** (page 77)

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 69)

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 69)

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 69)

7 Driver Installation and Uninstallation

7.1 USBIO Driver Executables

There are two separate driver executable files: **`usbio.sys`** for use on Windows 2000 and Windows XP, and **`usbio98.sys`** for use on Windows 98 and Windows ME. Depending on the operating system version, the correct driver executable file is installed by the USBIO .INF file. Thus, a complete USBIO device driver package consists of the following files:

- usbio.sys
- usbio98.sys
- usbio.inf

Note that the files should be renamed if a customized driver package is created. See section 7.4 (page 265) for detailed information.

The two USBIO driver executables usbio.sys and usbio98.sys behave identically. The only difference is that usbio.sys uses the kernel function **`MmGetSystemAddressForMdlSafe`** and usbio98.sys uses **`MmGetSystemAddressForMdl`** instead. In Windows 2000 and later versions the kernel call **`MmGetSystemAddressForMdl`** was replaced by a new function named **`MmGetSystemAddressForMdlSafe`**. Starting with Windows XP a device driver is required to use **`MmGetSystemAddressForMdlSafe`** rather than **`MmGetSystemAddressForMdl`**. Otherwise, the device driver cannot be certified by Microsoft's Windows Hardware Quality Labs (WHQL) and receive the Designed for Microsoft Windows logo. Unfortunately, the kernel function **`MmGetSystemAddressForMdlSafe`** is not available in older systems. For that reason, a separate executable is required to support Windows 98 and Windows ME.

7.2 Installing USBIO

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

Note:
On Windows 2000 and XP administrator privileges 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 access its programming interface.

7.2.1 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/XP 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/XP.

- 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, 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 without any user interaction. 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 for which the USBIO driver is to be installed. 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 (which is "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 have selected the correct device. If not, use the Back button to return to the device list and select another device.

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

Warning: If you install the USBIO driver for a device that is currently controlled by another device driver the existing driver will be disabled. This will happen immediately. As a result, 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 already 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 has been 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 binaries usbio.sys and usbio98.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 Development Kit. The application allows you to test several USB operations manually. Please refer to chapter 6 (page 251) for further information.

  To quit the USBIO Installation Wizard, click Finish.

7.2.2 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 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 searches 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 DDK for detailed information on Harware IDs and driver selection algorithms. Another suitable 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/XP 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 binaries usbio.sys and usbio98.sys to a floppy disk or to a directory of your choice. Copy the INF file usbio.inf provided with the USBIO Development Kit 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 semicolon 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, usbio.sys, usbio98.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, Windows XP, 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, usbio.sys, usbio98.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 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 a 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 (page 251) for detailed information on the Demo Application.

7.3 Uninstalling USBIO

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

7.3.1 Manual Uninstallation

In order to manual 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 choosing 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 launches the Upgrade Device Driver Wizard which searches 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/XP uses a slightly different scheme than Windows 98/ME). 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/XP 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.

The process described above can be automated by using the USBIO Cleanup Wizard. This procedure is described in the next section.

7.3.2 Automated Uninstallation: The USBIO Cleanup Wizard

The USBIO Cleanup Wizard makes it easy to completely remove the USBIO device driver from a system. The wizard performs uninstallation automatically in a step-by-step procedure. It is not necessary to manually remove files or registry entries.

Important:
On Windows 2000 and XP administrator privileges are required to execute the USBIO Cleanup Wizard.

The steps performed by the USBIO Cleanup Wizard are described below.

- The cleanup wizard removes the device instances created for USBIO from the registry. For each USB device that has been enumerated successfully the operating system creates a registry key called device instance key. On Windows 2000 and Windows XP these keys are located under `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB`. On Windows 98 and ME the location is `HKEY_LOCAL_MACHINE\Enum\USB`. When the USBIO device driver is installed for a USB device an additional registry key called driver key is created. By default, all registry keys associated with USBIO will be removed by the USBIO Cleanup Wizard. The wizard allows to de-select particular keys that are not to be removed.
- The cleanup wizard removes .INF files related to USBIO from the system's INF directory. The .INF files copied to the operating system's INF directory, e.g. `%WINDIR%\INF\`, during USBIO driver installation will be deleted. The cleanup wizard scans all .INF files and locates the USBIO setup INF files. By default, all .INF files associated with USBIO will be deleted. The wizard allows to de-select particular files that are not to be removed.

After the USBIO Cleanup Wizard has been executed any association between particular USB devices and USBIO is removed. When the USB device is connected to the system again, the Hardware Wizard is shown and a system-provided or vendor-provided device driver can be installed.

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 a user has two products and 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.

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

- Choose a new name for each of the driver executable files usbio.sys and usbio98.sys. The name should not cause conflicts with drivers provided by Windows. Rename the file usbio.sys and usbio98.sys respectively to your new name.
- Rename the Setup Information file usbio.inf. You can choose any name you want. For instance, the file name may be based on your company name or your product 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 executable file for Windows 2000 and XP.
- Edit the **[_CopyFiles_sys_98]** section to include the new name of the driver executable file for Windows 98 and ME.
- Edit the **[SourceDisksFiles]** section to include the new driver executable file names.
- Edit the values **S_DriverName** and **S_DriverName_98** in the **[Strings]** section to match each with the new name you defined for the corresponding driver executable. Note that the .sys extension is not specified.
- Edit the **[Strings]** section in the INF file to modify text strings that are shown at the user interface. 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 **`DriverUserInterfaceGuid`**. 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 application. Refer to the source code of USBIOAPP or ReaderCpp for an example.

- 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 and Windows ME.
  The parameters in **`[_Parameters1_NT]`** define the default behaviour of the USBIO driver on Windows 2000 and Windows XP. For a detailed description of the supported settings, refer to chapter 8 (page 265).

- 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.

7.5 Using USBIO on Windows XP Embedded

Windows XP Embedded is a retail version of the Windows XP operating system. The *Microsoft Windows Embedded Studio* is used to configure a Windows XP Embedded version for a particular hardware. The USBIO device driver can also be used on Windows XP Embedded.

In order to prepare USBIO for use on Windows XP Embedded the following steps are required:

- Create a USBIO driver package that is adapted to your device. Refer to section 7.4 (page 262) for detailed information on how to do that.
- Test the correct operation of the driver on a Windows XP operating system.
- Use the *Component Designer* of Windows Embedded Studio to create a new component. Refer to the Embedded Studio manual for more information.
- Use the *Component Database Manager* of Windows Embedded Studio to add the component to your data base.
- Use the *Target Designer* of Windows Embedded Studio to add the created component to your Windows XP Embedded project.

The USBIO device driver is loaded when the USB device is connected to the system running Windows XP Embedded. There is no user interaction required. The Hardware Wizard is not launched.

On Windows XP Embedded the USBIO COM interface is supported as well. The USBIO COM component that is implemented in USBIOCOM.DLL has to be registered before it can be used. This can be done in the Resources section of the *Component Designer* in Windows Embedded Studio. Alternatively, the library USBIOCOM.DLL can be registered by means of regsvr32.exe as described in the USBIO COM Interface Reference Manual.

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. By default this registry path is

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

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 6: 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 a 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.

Table 6: (continued)

Value	Min	Default	Max	Description
MaxIsoPackets	16	512	1024	Maximum number of packets allowed in an isochronous data transfer. Note that for isochronous transfers at full-speed a maximum packet count of 64 is sufficient for most devices (corresponds to 64 milliseconds per buffer). For high-speed transfers the maximum packet count should be at least 512. This is because one isochronous packet can be transferred per microframe.
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
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.

Table 6: (continued)

Value	Min	Default	Max	Description
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/XP 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.
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

- USBIO COM Interface Reference Manual, Thesycon GmbH, http://www.thesycon.de
- Universal Serial Bus Specification 1.1, http://www.usb.org
- Universal Serial Bus Specification 2.0, http://www.usb.org
- USB device class specifications (Audio, HID, Printer, etc.), http://www.usb.org
- Microsoft Windows DDK Documentation, http://msdn.microsoft.com
- Microsoft Platform SDK Documentation, http://msdn.microsoft.com

Index

~CSetupApiDll
    CSetupApiDll::~CSetupApiDll, 247
~CUsbIoBufPool
    CUsbIoBufPool::~CUsbIoBufPool, 240
~CUsbIoBuf
    CUsbIoBuf::~CUsbIoBuf, 234
~CUsbIoPipe
    CUsbIoPipe::~CUsbIoPipe, 189
~CUsbIoReader
    CUsbIoReader::~CUsbIoReader, 225
~CUsbIoThread
    CUsbIoThread::~CUsbIoThread, 213
~CUsbIoWriter
    CUsbIoWriter::~CUsbIoWriter, 228
~CUsbIo
    CUsbIo::~CUsbIo, 140

AbortPipe
    CUsbIoPipe::AbortPipe, 202
ActualAveragingInterval
    Member of USBIO_PIPE_STATISTICS, 114
AllocateBuffers
    CUsbIoThread::AllocateBuffers, 214
Allocate
    CUsbIoBufPool::Allocate, 241
AlternateSetting
    Member of USBIO_GET_INTERFACE_DATA, 95
    Member of USBIO_INTERFACE_CONFIGURATION_INFO, 102
    Parameter of CUsbIo::GetInterface, 176
AlternateSettingIndex
    Member of USBIO_INTERFACE_SETTING, 96
APIVersion
    Member of USBIO_DRIVER_INFO, 86
AverageRate
    Member of USBIO_PIPE_STATISTICS, 114
AveragingInterval
    Member of USBIO_SETUP_PIPE_STATISTICS, 111
    Parameter of CUsbIoPipe::SetupPipeStatistics, 209

BandwidthInfo
    Parameter of CUsbIo::GetBandwidthInfo, 156
Bind
    CUsbIoPipe::Bind, 190
Buf
    Parameter of CUsbIoBufPool::Put, 244
    Parameter of CUsbIoPipe::Read, 193
    Parameter of CUsbIoPipe::WaitForCompletion, 195

Parameter of CUsbIoPipe::Write, 194
Parameter of CUsbIoThread::BufErrorHandler, 220
Parameter of CUsbIoThread::ProcessBuffer, 219
Parameter of CUsbIoThread::ProcessData, 218
BufArray
Member of CUsbIoBufPool, 246
BufErrorHandler
CUsbIoThread::BufErrorHandler, 220
Buffer
Parameter of CUsbIo::ClassOrVendorInRequest, 169
Parameter of CUsbIo::ClassOrVendorOutRequest, 170
Parameter of CUsbIo::GetDescriptor, 157
Parameter of CUsbIo::SetDescriptor, 164
Parameter of CUsbIoBuf::CUsbIoBuf, 232
Parameter of CUsbIoPipe::PipeControlTransferIn, 205
Parameter of CUsbIoPipe::PipeControlTransferOut, 207
Parameter of CUsbIoPipe::ReadSync, 197
Parameter of CUsbIoPipe::WriteSync, 199
BufferMem
Member of CUsbIoBuf, 237
BufferMemAllocated
Member of CUsbIoBuf, 238
BufferMemory
Member of CUsbIoBufPool, 246
BufferSize
Member of CUsbIoBuf, 237
Parameter of CUsbIoBuf::CUsbIoBuf, 232, 233
Buffer
CUsbIoBuf::Buffer, 235
BufPool
Member of CUsbIoThread, 224
ByteCount
Parameter of CUsbIo::ClassOrVendorInRequest, 169
Parameter of CUsbIo::ClassOrVendorOutRequest, 170
Parameter of CUsbIo::GetConfigurationDescriptor, 160
Parameter of CUsbIo::GetDescriptor, 157
Parameter of CUsbIo::GetStringDescriptor, 162
Parameter of CUsbIo::SetDescriptor, 164
Parameter of CUsbIoPipe::PipeControlTransferIn, 205
Parameter of CUsbIoPipe::PipeControlTransferOut, 207
Parameter of CUsbIoPipe::ReadSync, 197
Parameter of CUsbIoPipe::WriteSync, 199
BytesReturned
Parameter of CUsbIo::IoctlSync, 186
BytesTransferred
Member of CUsbIoBuf, 237
BytesTransferred_H
Member of USBIO_PIPE_STATISTICS, 115

BytesTransferred_L
    Member of USBIO_PIPE_STATISTICS, 114

CancelIo
    CUsbIo::CancelIo, 185
CheckedBuildDetected
    Member of CUsbIo, 188
Class
    Member of USBIO_INTERFACE_CONFIGURATION_INFO, 102
ClassOrVendorInRequest
    CUsbIo::ClassOrVendorInRequest, 169
ClassOrVendorOutRequest
    CUsbIo::ClassOrVendorOutRequest, 170
ClearFeature
    CUsbIo::ClearFeature, 167
Close
    CUsbIo::Close, 145
Conf
    Parameter of CUsbIo::SetConfiguration, 171
ConfigurationIndex
    Member of USBIO_SET_CONFIGURATION, 97
ConfigurationValue
    Member of USBIO_GET_CONFIGURATION_DATA, 93
    Parameter of CUsbIo::GetConfiguration, 173
ConsumedBandwidth
    Member of USBIO_BANDWIDTH_INFO, 84
Context
    Member of CUsbIoBuf, 237
ControlTransfer
    Parameter of CUsbIoPipe::PipeControlTransferIn, 205
    Parameter of CUsbIoPipe::PipeControlTransferOut, 207
Count
    Member of CUsbIoBufPool, 246
CreateDeviceList
    CUsbIo::CreateDeviceList, 141
CritSect
    Member of CUsbIoBufPool, 246
    Member of CUsbIo, 188
CSetupApiDll
    CSetupApiDll::CSetupApiDll, 247
CSetupApiDll, 247
CSetupApiDll::~CSetupApiDll, 247
CSetupApiDll::CSetupApiDll, 247
CSetupApiDll::Load, 248
CSetupApiDll::Release, 249
CurrentCount
    CUsbIoBufPool::CurrentCount, 245
CUsbIoBufPool

CUsbIoBufPool::CUsbIoBufPool, 240
CUsbIoBufPool, 239
CUsbIoBufPool::~CUsbIoBufPool, 240
CUsbIoBufPool::Allocate, 241
CUsbIoBufPool::CurrentCount, 245
CUsbIoBufPool::CUsbIoBufPool, 240
CUsbIoBufPool::Free, 242
CUsbIoBufPool::Get, 243
CUsbIoBufPool::Put, 244
CUsbIoBuf
CUsbIoBuf::CUsbIoBuf, 231–233
CUsbIoBuf, 231
CUsbIoBuf::~CUsbIoBuf, 234
CUsbIoBuf::Buffer, 235
CUsbIoBuf::CUsbIoBuf, 231–233
CUsbIoBuf::Size, 236
CUsbIoPipe
CUsbIoPipe::CUsbIoPipe, 189
CUsbIoPipe, 189
CUsbIoPipe::~CUsbIoPipe, 189
CUsbIoPipe::AbortPipe, 202
CUsbIoPipe::Bind, 190
CUsbIoPipe::CUsbIoPipe, 189
CUsbIoPipe::GetPipeParameters, 203
CUsbIoPipe::PipeControlTransferIn, 205
CUsbIoPipe::PipeControlTransferOut, 207
CUsbIoPipe::QueryPipeStatistics, 210
CUsbIoPipe::ReadSync, 197
CUsbIoPipe::Read, 193
CUsbIoPipe::ResetPipeStatistics, 212
CUsbIoPipe::ResetPipe, 201
CUsbIoPipe::SetPipeParameters, 204
CUsbIoPipe::SetupPipeStatistics, 209
CUsbIoPipe::Unbind, 192
CUsbIoPipe::WaitForCompletion, 195
CUsbIoPipe::WriteSync, 199
CUsbIoPipe::Write, 194
CUsbIoReader
CUsbIoReader::CUsbIoReader, 225
CUsbIoReader, 225
CUsbIoReader::~CUsbIoReader, 225
CUsbIoReader::CUsbIoReader, 225
CUsbIoReader::TerminateThread, 227
CUsbIoReader::ThreadRoutine, 226
CUsbIoThread
CUsbIoThread::CUsbIoThread, 213
CUsbIoThread, 213
CUsbIoThread::~CUsbIoThread, 213

CUsbIoThread::AllocateBuffers, 214
CUsbIoThread::BufErrorHandler, 220
CUsbIoThread::CUsbIoThread, 213
CUsbIoThread::FreeBuffers, 215
CUsbIoThread::OnThreadExit, 221
CUsbIoThread::ProcessBuffer, 219
CUsbIoThread::ProcessData, 218
CUsbIoThread::ShutdownThread, 217
CUsbIoThread::StartThread, 216
CUsbIoThread::TerminateThread, 223
CUsbIoThread::ThreadRoutine, 222
CUsbIoWriter
    CUsbIoWriter::CUsbIoWriter, 228
CUsbIoWriter, 228
CUsbIoWriter::~CUsbIoWriter, 228
CUsbIoWriter::CUsbIoWriter, 228
CUsbIoWriter::TerminateThread, 230
CUsbIoWriter::ThreadRoutine, 229
CUsbIo
    CUsbIo::CUsbIo, 140
CUsbIo, 140
CUsbIo::~CUsbIo, 140
CUsbIo::CancelIo, 185
CUsbIo::ClassOrVendorInRequest, 169
CUsbIo::ClassOrVendorOutRequest, 170
CUsbIo::ClearFeature, 167
CUsbIo::Close, 145
CUsbIo::CreateDeviceList, 141
CUsbIo::CUsbIo, 140
CUsbIo::CyclePort, 181
CUsbIo::DestroyDeviceList, 142
CUsbIo::ErrorText, 187
CUsbIo::GetBandwidthInfo, 156
CUsbIo::GetConfigurationDescriptor, 160
CUsbIo::GetConfigurationInfo, 174
CUsbIo::GetConfiguration, 173
CUsbIo::GetCurrentFrameNumber, 182
CUsbIo::GetDescriptor, 157
CUsbIo::GetDeviceDescriptor, 159
CUsbIo::GetDeviceInfo, 155
CUsbIo::GetDeviceInstanceDetails, 146
CUsbIo::GetDeviceParameters, 178
CUsbIo::GetDevicePathName, 148
CUsbIo::GetDevicePowerState, 183
CUsbIo::GetDriverInfo, 154
CUsbIo::GetInterface, 176
CUsbIo::GetStatus, 168
CUsbIo::GetStringDescriptor, 162

CUsbIo::IoctlSync, 186
CUsbIo::IsCheckedBuild, 150
CUsbIo::IsDemoVersion, 151
CUsbIo::IsLightVersion, 152
CUsbIo::IsOpen, 149
CUsbIo::IsOperatingAtHighSpeed, 153
CUsbIo::Open, 143
CUsbIo::ResetDevice, 180
CUsbIo::SetConfiguration, 171
CUsbIo::SetDescriptor, 164
CUsbIo::SetDeviceParameters, 179
CUsbIo::SetDevicePowerState, 184
CUsbIo::SetFeature, 166
CUsbIo::SetInterface, 175
CUsbIo::StoreConfigurationDescriptor, 177
CUsbIo::UnconfigureDevice, 172
CyclePort
    CUsbIo::CyclePort, 181

DemoVersionDetected
    Member of CUsbIo, 188
Desc
    Parameter of CUsbIo::GetConfigurationDescriptor, 160
    Parameter of CUsbIo::GetDeviceDescriptor, 159
    Parameter of CUsbIo::GetStringDescriptor, 162
    Parameter of CUsbIo::StoreConfigurationDescriptor, 177
DescriptorIndex
    Member of USBIO_DESCRIPTOR_REQUEST, 88
    Parameter of CUsbIo::GetDescriptor, 157
    Parameter of CUsbIo::SetDescriptor, 164
DescriptorType
    Member of USBIO_DESCRIPTOR_REQUEST, 88
    Parameter of CUsbIo::GetDescriptor, 157
    Parameter of CUsbIo::SetDescriptor, 164
DestroyDeviceList
    CUsbIo::DestroyDeviceList, 142
DeviceInfo
    Parameter of CUsbIo::GetDeviceInfo, 155
DeviceList
    Parameter of CUsbIo::DestroyDeviceList, 142
    Parameter of CUsbIo::GetDeviceInstanceDetails, 146
    Parameter of CUsbIo::Open, 143
    Parameter of CUsbIoPipe::Bind, 190
DeviceNumber
    Parameter of CUsbIo::GetDeviceInstanceDetails, 146
    Parameter of CUsbIo::Open, 143
    Parameter of CUsbIoPipe::Bind, 190
DevicePowerState

Member of USBIO_DEVICE_POWER, 108
Parameter of CUsbIo::GetDevicePowerState, 183
Parameter of CUsbIo::SetDevicePowerState, 184
DevicePowerStateD0
Entry of USBIO_DEVICE_POWER_STATE, 124
DevicePowerStateD1
Entry of USBIO_DEVICE_POWER_STATE, 124
DevicePowerStateD2
Entry of USBIO_DEVICE_POWER_STATE, 124
DevicePowerStateD3
Entry of USBIO_DEVICE_POWER_STATE, 124
DevParam
Parameter of CUsbIo::GetDeviceParameters, 178
Parameter of CUsbIo::SetDeviceParameters, 179
DriverBuildNumber
Member of USBIO_DRIVER_INFO, 86
DriverInfo
Parameter of CUsbIo::GetDriverInfo, 154
DriverVersion
Member of USBIO_DRIVER_INFO, 86
dwIoControlCode
Parameter of IOCTL_USBIO_ABORT_PIPE, 69
Parameter of IOCTL_USBIO_BIND_PIPE, 66
Parameter of IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST, 49
Parameter of IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST, 50
Parameter of IOCTL_USBIO_CLEAR_FEATURE, 41
Parameter of IOCTL_USBIO_CYCLE_PORT, 64
Parameter of IOCTL_USBIO_GET_BANDWIDTH_INFO, 60
Parameter of IOCTL_USBIO_GET_CONFIGURATION_INFO, 54
Parameter of IOCTL_USBIO_GET_CONFIGURATION, 43
Parameter of IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER, 57
Parameter of IOCTL_USBIO_GET_DESCRIPTOR, 38
Parameter of IOCTL_USBIO_GET_DEVICE_INFO, 61
Parameter of IOCTL_USBIO_GET_DEVICE_PARAMETERS, 52
Parameter of IOCTL_USBIO_GET_DEVICE_POWER_STATE, 59
Parameter of IOCTL_USBIO_GET_DRIVER_INFO, 62
Parameter of IOCTL_USBIO_GET_INTERFACE, 44
Parameter of IOCTL_USBIO_GET_PIPE_PARAMETERS, 70
Parameter of IOCTL_USBIO_GET_STATUS, 42
Parameter of IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN, 76
Parameter of IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT, 77
Parameter of IOCTL_USBIO_QUERY_PIPE_STATISTICS, 74
Parameter of IOCTL_USBIO_RESET_DEVICE, 55
Parameter of IOCTL_USBIO_RESET_PIPE, 68
Parameter of IOCTL_USBIO_SET_CONFIGURATION, 46
Parameter of IOCTL_USBIO_SET_DESCRIPTOR, 39
Parameter of IOCTL_USBIO_SET_DEVICE_PARAMETERS, 53
Parameter of IOCTL_USBIO_SET_DEVICE_POWER_STATE, 58

Parameter of IOCTL_USBIO_SET_FEATURE, 40
Parameter of IOCTL_USBIO_SET_INTERFACE, 48
Parameter of IOCTL_USBIO_SET_PIPE_PARAMETERS, 71
Parameter of IOCTL_USBIO_SETUP_PIPE_STATISTICS, 72
Parameter of IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR, 45
Parameter of IOCTL_USBIO_UNBIND_PIPE, 67
Parameter of IOCTL_USBIO_UNCONFIGURE_DEVICE, 47

EndpointAddress
Member of USBIO_BIND_PIPE, 109
Member of USBIO_PIPE_CONFIGURATION_INFO, 104
Parameter of CUsbIoPipe::Bind, 190
ErrorCode
Parameter of CUsbIo::ErrorText, 187
ErrorCount
Member of USBIO_ISO_TRANSFER, 118
ErrorText
CUsbIo::ErrorText, 187

FeatureSelector
Member of USBIO_FEATURE_REQUEST, 90
Parameter of CUsbIo::ClearFeature, 167
Parameter of CUsbIo::SetFeature, 166
FileHandle
Member of CUsbIo, 188
FirstPending
Member of CUsbIoThread, 224
Flags
Member of USBIO_CLASS_OR_VENDOR_REQUEST, 98
Member of USBIO_DEVICE_INFO, 85
Member of USBIO_DRIVER_INFO, 87
Member of USBIO_ISO_TRANSFER, 117
Member of USBIO_PIPE_CONTROL_TRANSFER, 116
Member of USBIO_PIPE_PARAMETERS, 110
Member of USBIO_QUERY_PIPE_STATISTICS, 112
Parameter of CUsbIoPipe::QueryPipeStatistics, 210
FrameNumber
Member of USBIO_FRAME_NUMBER, 107
Parameter of CUsbIo::GetCurrentFrameNumber, 182
FreeBuffers
CUsbIoThread::FreeBuffers, 215
Free
CUsbIoBufPool::Free, 242

GetBandwidthInfo
CUsbIo::GetBandwidthInfo, 156
GetConfigurationDescriptor
CUsbIo::GetConfigurationDescriptor, 160
GetConfigurationInfo

CUsbIo::GetConfigurationInfo, 174
GetConfiguration
CUsbIo::GetConfiguration, 173
GetCurrentFrameNumber
CUsbIo::GetCurrentFrameNumber, 182
GetDescriptor
CUsbIo::GetDescriptor, 157
GetDeviceDescriptor
CUsbIo::GetDeviceDescriptor, 159
GetDeviceInfo
CUsbIo::GetDeviceInfo, 155
GetDeviceInstanceDetails
CUsbIo::GetDeviceInstanceDetails, 146
GetDeviceParameters
CUsbIo::GetDeviceParameters, 178
GetDevicePathName
CUsbIo::GetDevicePathName, 148
GetDevicePowerState
CUsbIo::GetDevicePowerState, 183
GetDriverInfo
CUsbIo::GetDriverInfo, 154
GetInterface
CUsbIo::GetInterface, 176
GetPipeParameters
CUsbIoPipe::GetPipeParameters, 203
GetStatus
CUsbIo::GetStatus, 168
GetStringDescriptor
CUsbIo::GetStringDescriptor, 162
Get
CUsbIoBufPool::Get, 243

Head
Member of CUsbIoBufPool, 246

InBuffer
Parameter of CUsbIo::IoctlSync, 186
InBufferSize
Parameter of CUsbIo::IoctlSync, 186
Index
Member of USBIO_CLASS_OR_VENDOR_REQUEST, 98
Member of USBIO_FEATURE_REQUEST, 90
Member of USBIO_STATUS_REQUEST, 91
Parameter of CUsbIo::ClearFeature, 167
Parameter of CUsbIo::GetConfigurationDescriptor, 160
Parameter of CUsbIo::GetStatus, 168
Parameter of CUsbIo::GetStringDescriptor, 162
Parameter of CUsbIo::SetFeature, 166
Info

Parameter of CUsbIo::GetConfigurationInfo, 174
Interface
Member of USBIO_GET_INTERFACE, 94
Parameter of CUsbIo::GetInterface, 176
InterfaceGuid
Parameter of CUsbIo::CreateDeviceList, 141
Parameter of CUsbIo::GetDeviceInstanceDetails, 146
Parameter of CUsbIo::Open, 143
Parameter of CUsbIoPipe::Bind, 190
InterfaceIndex
Member of USBIO_INTERFACE_SETTING, 96
InterfaceInfo[USBIO_MAX_INTERFACES]
Member of USBIO_CONFIGURATION_INFO, 106
InterfaceList[USBIO_MAX_INTERFACES]
Member of USBIO_SET_CONFIGURATION, 97
InterfaceNumber
Member of USBIO_INTERFACE_CONFIGURATION_INFO, 102
Member of USBIO_PIPE_CONFIGURATION_INFO, 105
Interval
Member of USBIO_PIPE_CONFIGURATION_INFO, 104
IOCTL_USBIO_ABORT_PIPE, 69
IOCTL_USBIO_BIND_PIPE, 66
IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST, 49
IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST, 50
IOCTL_USBIO_CLEAR_FEATURE, 41
IOCTL_USBIO_CYCLE_PORT, 64
IOCTL_USBIO_GET_BANDWIDTH_INFO, 60
IOCTL_USBIO_GET_CONFIGURATION_INFO, 54
IOCTL_USBIO_GET_CONFIGURATION, 43
IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER, 57
IOCTL_USBIO_GET_DESCRIPTOR, 38
IOCTL_USBIO_GET_DEVICE_INFO, 61
IOCTL_USBIO_GET_DEVICE_PARAMETERS, 52
IOCTL_USBIO_GET_DEVICE_POWER_STATE, 59
IOCTL_USBIO_GET_DRIVER_INFO, 62
IOCTL_USBIO_GET_INTERFACE, 44
IOCTL_USBIO_GET_PIPE_PARAMETERS, 70
IOCTL_USBIO_GET_STATUS, 42
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN, 76
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT, 77
IOCTL_USBIO_QUERY_PIPE_STATISTICS, 74
IOCTL_USBIO_RESET_DEVICE, 55
IOCTL_USBIO_RESET_PIPE, 68
IOCTL_USBIO_SET_CONFIGURATION, 46
IOCTL_USBIO_SET_DESCRIPTOR, 39
IOCTL_USBIO_SET_DEVICE_PARAMETERS, 53
IOCTL_USBIO_SET_DEVICE_POWER_STATE, 58
IOCTL_USBIO_SET_FEATURE, 40

IOCTL_USBIO_SET_INTERFACE, 48
IOCTL_USBIO_SET_PIPE_PARAMETERS, 71
IOCTL_USBIO_SETUP_PIPE_STATISTICS, 72
IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR, 45
IOCTL_USBIO_UNBIND_PIPE, 67
IOCTL_USBIO_UNCONFIGURE_DEVICE, 47
IoctlCode
    Parameter of CUsbIo::IoctlSync, 186
IoctlSync
    CUsbIo::IoctlSync, 186
IsCheckedBuild
    CUsbIo::IsCheckedBuild, 150
IsDemoVersion
    CUsbIo::IsDemoVersion, 151
IsLightVersion
    CUsbIo::IsLightVersion, 152
IsoPacket[1]
    Member of USBIO_ISO_TRANSFER_HEADER, 120
IsOpen
    CUsbIo::IsOpen, 149
IsOperatingAtHighSpeed
    CUsbIo::IsOperatingAtHighSpeed, 153
IsoTransfer
    Member of USBIO_ISO_TRANSFER_HEADER, 120

LanguageId
    Member of USBIO_DESCRIPTOR_REQUEST, 88
    Parameter of CUsbIo::GetDescriptor, 157
    Parameter of CUsbIo::GetStringDescriptor, 162
    Parameter of CUsbIo::SetDescriptor, 164
LastPending
    Member of CUsbIoThread, 224
Length
    Member of USBIO_ISO_PACKET, 119
LightVersionDetected
    Member of CUsbIo, 188
Load
    CSetupApiDll::Load, 248
lpBytesReturned
    Parameter of IOCTL_USBIO_ABORT_PIPE, 69
    Parameter of IOCTL_USBIO_BIND_PIPE, 66
    Parameter of IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST, 49
    Parameter of IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST, 50
    Parameter of IOCTL_USBIO_CLEAR_FEATURE, 41
    Parameter of IOCTL_USBIO_CYCLE_PORT, 64
    Parameter of IOCTL_USBIO_GET_BANDWIDTH_INFO, 60
    Parameter of IOCTL_USBIO_GET_CONFIGURATION_INFO, 54
    Parameter of IOCTL_USBIO_GET_CONFIGURATION, 43

Parameter of IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER, 57
Parameter of IOCTL_USBIO_GET_DESCRIPTOR, 38
Parameter of IOCTL_USBIO_GET_DEVICE_INFO, 61
Parameter of IOCTL_USBIO_GET_DEVICE_PARAMETERS, 52
Parameter of IOCTL_USBIO_GET_DEVICE_POWER_STATE, 59
Parameter of IOCTL_USBIO_GET_DRIVER_INFO, 62
Parameter of IOCTL_USBIO_GET_INTERFACE, 44
Parameter of IOCTL_USBIO_GET_PIPE_PARAMETERS, 70
Parameter of IOCTL_USBIO_GET_STATUS, 42
Parameter of IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN, 76
Parameter of IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT, 77
Parameter of IOCTL_USBIO_QUERY_PIPE_STATISTICS, 74
Parameter of IOCTL_USBIO_RESET_DEVICE, 55
Parameter of IOCTL_USBIO_RESET_PIPE, 68
Parameter of IOCTL_USBIO_SET_CONFIGURATION, 46
Parameter of IOCTL_USBIO_SET_DESCRIPTOR, 39
Parameter of IOCTL_USBIO_SET_DEVICE_PARAMETERS, 53
Parameter of IOCTL_USBIO_SET_DEVICE_POWER_STATE, 58
Parameter of IOCTL_USBIO_SET_FEATURE, 40
Parameter of IOCTL_USBIO_SET_INTERFACE, 48
Parameter of IOCTL_USBIO_SET_PIPE_PARAMETERS, 71
Parameter of IOCTL_USBIO_SETUP_PIPE_STATISTICS, 72
Parameter of IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR, 45
Parameter of IOCTL_USBIO_UNBIND_PIPE, 67
Parameter of IOCTL_USBIO_UNCONFIGURE_DEVICE, 47

lpInBuffer

Parameter of IOCTL_USBIO_ABORT_PIPE, 69
Parameter of IOCTL_USBIO_BIND_PIPE, 66
Parameter of IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST, 49
Parameter of IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST, 50
Parameter of IOCTL_USBIO_CLEAR_FEATURE, 41
Parameter of IOCTL_USBIO_CYCLE_PORT, 64
Parameter of IOCTL_USBIO_GET_BANDWIDTH_INFO, 60
Parameter of IOCTL_USBIO_GET_CONFIGURATION_INFO, 54
Parameter of IOCTL_USBIO_GET_CONFIGURATION, 43
Parameter of IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER, 57
Parameter of IOCTL_USBIO_GET_DESCRIPTOR, 38
Parameter of IOCTL_USBIO_GET_DEVICE_INFO, 61
Parameter of IOCTL_USBIO_GET_DEVICE_PARAMETERS, 52
Parameter of IOCTL_USBIO_GET_DEVICE_POWER_STATE, 59
Parameter of IOCTL_USBIO_GET_DRIVER_INFO, 62
Parameter of IOCTL_USBIO_GET_INTERFACE, 44
Parameter of IOCTL_USBIO_GET_PIPE_PARAMETERS, 70
Parameter of IOCTL_USBIO_GET_STATUS, 42
Parameter of IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN, 76
Parameter of IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT, 77
Parameter of IOCTL_USBIO_QUERY_PIPE_STATISTICS, 74
Parameter of IOCTL_USBIO_RESET_DEVICE, 55

Parameter of IOCTL_USBIO_RESET_PIPE, 68
Parameter of IOCTL_USBIO_SET_CONFIGURATION, 46
Parameter of IOCTL_USBIO_SET_DESCRIPTOR, 39
Parameter of IOCTL_USBIO_SET_DEVICE_PARAMETERS, 53
Parameter of IOCTL_USBIO_SET_DEVICE_POWER_STATE, 58
Parameter of IOCTL_USBIO_SET_FEATURE, 40
Parameter of IOCTL_USBIO_SET_INTERFACE, 48
Parameter of IOCTL_USBIO_SET_PIPE_PARAMETERS, 71
Parameter of IOCTL_USBIO_SETUP_PIPE_STATISTICS, 72
Parameter of IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR, 45
Parameter of IOCTL_USBIO_UNBIND_PIPE, 67
Parameter of IOCTL_USBIO_UNCONFIGURE_DEVICE, 47

lpOutBuffer

Parameter of IOCTL_USBIO_ABORT_PIPE, 69
Parameter of IOCTL_USBIO_BIND_PIPE, 66
Parameter of IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST, 49
Parameter of IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST, 50
Parameter of IOCTL_USBIO_CLEAR_FEATURE, 41
Parameter of IOCTL_USBIO_CYCLE_PORT, 64
Parameter of IOCTL_USBIO_GET_BANDWIDTH_INFO, 60
Parameter of IOCTL_USBIO_GET_CONFIGURATION_INFO, 54
Parameter of IOCTL_USBIO_GET_CONFIGURATION, 43
Parameter of IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER, 57
Parameter of IOCTL_USBIO_GET_DESCRIPTOR, 38
Parameter of IOCTL_USBIO_GET_DEVICE_INFO, 61
Parameter of IOCTL_USBIO_GET_DEVICE_PARAMETERS, 52
Parameter of IOCTL_USBIO_GET_DEVICE_POWER_STATE, 59
Parameter of IOCTL_USBIO_GET_DRIVER_INFO, 62
Parameter of IOCTL_USBIO_GET_INTERFACE, 44
Parameter of IOCTL_USBIO_GET_PIPE_PARAMETERS, 70
Parameter of IOCTL_USBIO_GET_STATUS, 42
Parameter of IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN, 76
Parameter of IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT, 77
Parameter of IOCTL_USBIO_QUERY_PIPE_STATISTICS, 74
Parameter of IOCTL_USBIO_RESET_DEVICE, 55
Parameter of IOCTL_USBIO_RESET_PIPE, 68
Parameter of IOCTL_USBIO_SET_CONFIGURATION, 46
Parameter of IOCTL_USBIO_SET_DESCRIPTOR, 39
Parameter of IOCTL_USBIO_SET_DEVICE_PARAMETERS, 53
Parameter of IOCTL_USBIO_SET_DEVICE_POWER_STATE, 58
Parameter of IOCTL_USBIO_SET_FEATURE, 40
Parameter of IOCTL_USBIO_SET_INTERFACE, 48
Parameter of IOCTL_USBIO_SET_PIPE_PARAMETERS, 71
Parameter of IOCTL_USBIO_SETUP_PIPE_STATISTICS, 72
Parameter of IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR, 45
Parameter of IOCTL_USBIO_UNBIND_PIPE, 67
Parameter of IOCTL_USBIO_UNCONFIGURE_DEVICE, 47

MaxErrorCount
Member of CUsbIoThread, 224
MaximumPacketSize
Member of USBIO_PIPE_CONFIGURATION_INFO, 104
MaximumTransferSize
Member of USBIO_INTERFACE_SETTING, 96
Member of USBIO_PIPE_CONFIGURATION_INFO, 104
MaxIoErrorCount
Parameter of CUsbIoThread::StartThread, 216
mDevDetail
Member of CUsbIo, 188

NbOfInterfaces
Member of USBIO_CONFIGURATION_INFO, 106
Member of USBIO_SET_CONFIGURATION, 97
NbOfPipes
Member of USBIO_CONFIGURATION_INFO, 106
Next
Member of CUsbIoBuf, 237
nInBufferSize
Parameter of IOCTL_USBIO_ABORT_PIPE, 69
Parameter of IOCTL_USBIO_BIND_PIPE, 66
Parameter of IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST, 49
Parameter of IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST, 50
Parameter of IOCTL_USBIO_CLEAR_FEATURE, 41
Parameter of IOCTL_USBIO_CYCLE_PORT, 64
Parameter of IOCTL_USBIO_GET_BANDWIDTH_INFO, 60
Parameter of IOCTL_USBIO_GET_CONFIGURATION_INFO, 54
Parameter of IOCTL_USBIO_GET_CONFIGURATION, 43
Parameter of IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER, 57
Parameter of IOCTL_USBIO_GET_DESCRIPTOR, 38
Parameter of IOCTL_USBIO_GET_DEVICE_INFO, 61
Parameter of IOCTL_USBIO_GET_DEVICE_PARAMETERS, 52
Parameter of IOCTL_USBIO_GET_DEVICE_POWER_STATE, 59
Parameter of IOCTL_USBIO_GET_DRIVER_INFO, 62
Parameter of IOCTL_USBIO_GET_INTERFACE, 44
Parameter of IOCTL_USBIO_GET_PIPE_PARAMETERS, 70
Parameter of IOCTL_USBIO_GET_STATUS, 42
Parameter of IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN, 76
Parameter of IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT, 77
Parameter of IOCTL_USBIO_QUERY_PIPE_STATISTICS, 74
Parameter of IOCTL_USBIO_RESET_DEVICE, 55
Parameter of IOCTL_USBIO_RESET_PIPE, 68
Parameter of IOCTL_USBIO_SET_CONFIGURATION, 46
Parameter of IOCTL_USBIO_SET_DESCRIPTOR, 39
Parameter of IOCTL_USBIO_SET_DEVICE_PARAMETERS, 53
Parameter of IOCTL_USBIO_SET_DEVICE_POWER_STATE, 58
Parameter of IOCTL_USBIO_SET_FEATURE, 40

Parameter of IOCTL_USBIO_SET_INTERFACE, 48
Parameter of IOCTL_USBIO_SET_PIPE_PARAMETERS, 71
Parameter of IOCTL_USBIO_SETUP_PIPE_STATISTICS, 72
Parameter of IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR, 45
Parameter of IOCTL_USBIO_UNBIND_PIPE, 67
Parameter of IOCTL_USBIO_UNCONFIGURE_DEVICE, 47
nOutBufferSize
Parameter of IOCTL_USBIO_ABORT_PIPE, 69
Parameter of IOCTL_USBIO_BIND_PIPE, 66
Parameter of IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST, 49
Parameter of IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST, 50
Parameter of IOCTL_USBIO_CLEAR_FEATURE, 41
Parameter of IOCTL_USBIO_CYCLE_PORT, 64
Parameter of IOCTL_USBIO_GET_BANDWIDTH_INFO, 60
Parameter of IOCTL_USBIO_GET_CONFIGURATION_INFO, 54
Parameter of IOCTL_USBIO_GET_CONFIGURATION, 43
Parameter of IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER, 57
Parameter of IOCTL_USBIO_GET_DESCRIPTOR, 38
Parameter of IOCTL_USBIO_GET_DEVICE_INFO, 61
Parameter of IOCTL_USBIO_GET_DEVICE_PARAMETERS, 52
Parameter of IOCTL_USBIO_GET_DEVICE_POWER_STATE, 59
Parameter of IOCTL_USBIO_GET_DRIVER_INFO, 62
Parameter of IOCTL_USBIO_GET_INTERFACE, 44
Parameter of IOCTL_USBIO_GET_PIPE_PARAMETERS, 70
Parameter of IOCTL_USBIO_GET_STATUS, 42
Parameter of IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN, 76
Parameter of IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT, 77
Parameter of IOCTL_USBIO_QUERY_PIPE_STATISTICS, 74
Parameter of IOCTL_USBIO_RESET_DEVICE, 55
Parameter of IOCTL_USBIO_RESET_PIPE, 68
Parameter of IOCTL_USBIO_SET_CONFIGURATION, 46
Parameter of IOCTL_USBIO_SET_DESCRIPTOR, 39
Parameter of IOCTL_USBIO_SET_DEVICE_PARAMETERS, 53
Parameter of IOCTL_USBIO_SET_DEVICE_POWER_STATE, 58
Parameter of IOCTL_USBIO_SET_FEATURE, 40
Parameter of IOCTL_USBIO_SET_INTERFACE, 48
Parameter of IOCTL_USBIO_SET_PIPE_PARAMETERS, 71
Parameter of IOCTL_USBIO_SETUP_PIPE_STATISTICS, 72
Parameter of IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR, 45
Parameter of IOCTL_USBIO_UNBIND_PIPE, 67
Parameter of IOCTL_USBIO_UNCONFIGURE_DEVICE, 47
NumberOfBuffers
Parameter of CUsbIoBufPool::Allocate, 241
Parameter of CUsbIoThread::AllocateBuffers, 214
NumberOfBytesToTransfer
Member of CUsbIoBuf, 237
NumberOfPackets
Member of USBIO_ISO_TRANSFER, 117

NumberOfPipes
    Member of USBIO_INTERFACE_CONFIGURATION_INFO, 102

Offset
    Member of USBIO_ISO_PACKET, 119
OnThreadExit
    CUsbIoThread::OnThreadExit, 221
Open
    CUsbIo::Open, 143
OperationFinished
    Member of CUsbIoBuf, 237
Options
    Member of USBIO_DEVICE_PARAMETERS, 100
OutBuffer
    Parameter of CUsbIo::IoctlSync, 186
OutBufferSize
    Parameter of CUsbIo::IoctlSync, 186
Overlapped
    Member of CUsbIoBuf, 237
    Member of CUsbIo, 188

PipeControlTransferIn
    CUsbIoPipe::PipeControlTransferIn, 205
PipeControlTransferOut
    CUsbIoPipe::PipeControlTransferOut, 207
PipeInfo[USBIO_MAX_PIPES]
    Member of USBIO_CONFIGURATION_INFO, 106
PipeParameters
    Parameter of CUsbIoPipe::GetPipeParameters, 203
    Parameter of CUsbIoPipe::SetPipeParameters, 204
PipeStatistics
    Parameter of CUsbIoPipe::QueryPipeStatistics, 210
PipeType
    Member of USBIO_PIPE_CONFIGURATION_INFO, 104
ProcessBuffer
    CUsbIoThread::ProcessBuffer, 219
ProcessData
    CUsbIoThread::ProcessData, 218
Protocol
    Member of USBIO_INTERFACE_CONFIGURATION_INFO, 102
Put
    CUsbIoBufPool::Put, 244

QueryPipeStatistics
    CUsbIoPipe::QueryPipeStatistics, 210

ReadSync
    CUsbIoPipe::ReadSync, 197
Read

CUsbIoPipe::Read, 193
Recipient
Member of USBIO_CLASS_OR_VENDOR_REQUEST, 98
Member of USBIO_DESCRIPTOR_REQUEST, 88
Member of USBIO_FEATURE_REQUEST, 90
Member of USBIO_STATUS_REQUEST, 91
Parameter of CUsbIo::ClearFeature, 167
Parameter of CUsbIo::GetDescriptor, 157
Parameter of CUsbIo::GetStatus, 168
Parameter of CUsbIo::SetDescriptor, 164
Parameter of CUsbIo::SetFeature, 166
Release
CSetupApiDll::Release, 249
Request
Member of USBIO_CLASS_OR_VENDOR_REQUEST, 98
Parameter of CUsbIo::ClassOrVendorInRequest, 169
Parameter of CUsbIo::ClassOrVendorOutRequest, 170
RequestsFailed
Member of USBIO_PIPE_STATISTICS, 115
RequestsSucceeded
Member of USBIO_PIPE_STATISTICS, 115
RequestTimeout
Member of USBIO_DEVICE_PARAMETERS, 100
RequestTypeReservedBits
Member of USBIO_CLASS_OR_VENDOR_REQUEST, 98
reserved1
Member of USBIO_BANDWIDTH_INFO, 84
Member of USBIO_DEVICE_INFO, 85
Member of USBIO_INTERFACE_CONFIGURATION_INFO, 102
Member of USBIO_PIPE_CONFIGURATION_INFO, 105
Member of USBIO_PIPE_STATISTICS, 115
Member of USBIO_SETUP_PIPE_STATISTICS, 111
reserved2
Member of USBIO_BANDWIDTH_INFO, 84
Member of USBIO_DEVICE_INFO, 85
Member of USBIO_INTERFACE_CONFIGURATION_INFO, 102
Member of USBIO_PIPE_CONFIGURATION_INFO, 105
Member of USBIO_PIPE_STATISTICS, 115
Member of USBIO_SETUP_PIPE_STATISTICS, 111
reserved3
Member of USBIO_DEVICE_INFO, 85
Member of USBIO_PIPE_CONFIGURATION_INFO, 105
ResetDevice
CUsbIo::ResetDevice, 180
ResetPipeStatistics
CUsbIoPipe::ResetPipeStatistics, 212
ResetPipe
CUsbIoPipe::ResetPipe, 201

SetConfiguration
    CUsbIo::SetConfiguration, 171
SetDescriptor
    CUsbIo::SetDescriptor, 164
SetDeviceParameters
    CUsbIo::SetDeviceParameters, 179
SetDevicePowerState
    CUsbIo::SetDevicePowerState, 184
SetFeature
    CUsbIo::SetFeature, 166
SetInterface
    CUsbIo::SetInterface, 175
SetPipeParameters
    CUsbIoPipe::SetPipeParameters, 204
Setting
    Parameter of CUsbIo::SetInterface, 175
SetupPacket[8]
    Member of USBIO_PIPE_CONTROL_TRANSFER, 116
SetupPipeStatistics
    CUsbIoPipe::SetupPipeStatistics, 209
ShutdownThread
    CUsbIoThread::ShutdownThread, 217
SizeOfBuffer
    Parameter of CUsbIoBufPool::Allocate, 241
    Parameter of CUsbIoThread::AllocateBuffers, 214
Size
    CUsbIoBuf::Size, 236
smSetupApi
    Member of CUsbIo, 188
StartFrame
    Member of USBIO_ISO_TRANSFER, 117
StartThread
    CUsbIoThread::StartThread, 216
Status
    Member of CUsbIoBuf, 237
    Member of USBIO_ISO_PACKET, 119
    Member of USBIO_STATUS_REQUEST_DATA, 92
StatusValue
    Parameter of CUsbIo::GetStatus, 168
StoreConfigurationDescriptor
    CUsbIo::StoreConfigurationDescriptor, 177
StringBuffer
    Parameter of CUsbIo::ErrorText, 187
StringBufferSize
    Parameter of CUsbIo::ErrorText, 187
SubClass
    Member of USBIO_INTERFACE_CONFIGURATION_INFO, 102

TerminateFlag
    Member of CUsbIoThread, 224
TerminateThread
    CUsbIoReader::TerminateThread, 227
    CUsbIoThread::TerminateThread, 223
    CUsbIoWriter::TerminateThread, 230
ThreadHandle
    Member of CUsbIoThread, 224
ThreadID
    Member of CUsbIoThread, 224
ThreadRoutine
    CUsbIoReader::ThreadRoutine, 226
    CUsbIoThread::ThreadRoutine, 222
    CUsbIoWriter::ThreadRoutine, 229
Timeout
    Parameter of CUsbIoPipe::ReadSync, 197
    Parameter of CUsbIoPipe::WaitForCompletion, 195
    Parameter of CUsbIoPipe::WriteSync, 199
TotalBandwidth
    Member of USBIO_BANDWIDTH_INFO, 84
Type
    Member of USBIO_CLASS_OR_VENDOR_REQUEST, 98

Unbind
    CUsbIoPipe::Unbind, 192
UnconfigureDevice
    CUsbIo::UnconfigureDevice, 172
USBIO_BANDWIDTH_INFO, 84
USBIO_BIND_PIPE, 109
USBIO_CLASS_OR_VENDOR_REQUEST, 98
USBIO_CONFIGURATION_INFO, 106
USBIO_DESCRIPTOR_REQUEST, 88
USBIO_DEVICE_INFO, 85
USBIO_DEVICE_PARAMETERS, 100
USBIO_DEVICE_POWER_STATE, 124
USBIO_DEVICE_POWER, 108
USBIO_DRIVER_INFO, 86
USBIO_ERR_ALREADY_BOUND, 131
USBIO_ERR_ALREADY_CONFIGURED, 131
USBIO_ERR_BABBLE_DETECTED, 126
USBIO_ERR_BAD_START_FRAME, 128
USBIO_ERR_BTSTUFF, 125
USBIO_ERR_BUFFER_OVERRUN, 126
USBIO_ERR_BUFFER_UNDERRUN, 126
USBIO_ERR_BULK_RESTRICTION, 134
USBIO_ERR_CANCELED, 129
USBIO_ERR_CONTROL_NOT_SUPPORTED, 131
USBIO_ERR_CONTROL_RESTRICTION, 135

USBIO_ERR_CRC, 125
USBIO_ERR_DATA_BUFFER_ERROR, 127
USBIO_ERR_DATA_OVERRUN, 125
USBIO_ERR_DATA_TOGGLE_MISMATCH, 125
USBIO_ERR_DATA_UNDERRUN, 126
USBIO_ERR_DEMO_EXPIRED, 133
USBIO_ERR_DEV_NOT_RESPONDING, 125
USBIO_ERR_DEVICE_GONE, 129
USBIO_ERR_DEVICE_NOT_FOUND, 135
USBIO_ERR_DEVICE_NOT_OPEN, 135
USBIO_ERR_DEVICE_NOT_PRESENT, 131
USBIO_ERR_ENDPOINT_HALTED, 127
USBIO_ERR_EP0_RESTRICTION, 134
USBIO_ERR_ERROR_BUSY, 127
USBIO_ERR_ERROR_SHORT_TRANSFER, 128
USBIO_ERR_FAILED, 130
USBIO_ERR_FIFO, 126
USBIO_ERR_FRAME_CONTROL_NOT_OWNED, 128
USBIO_ERR_FRAME_CONTROL_OWNED, 128
USBIO_ERR_INSUFFICIENT_RESOURCES, 129
USBIO_ERR_INTERFACE_NOT_FOUND, 133
USBIO_ERR_INTERNAL_HC_ERROR, 128
USBIO_ERR_INTERRUPT_RESTRICTION, 135
USBIO_ERR_INVALID_CONFIGURATION_DESCRIPTOR, 128
USBIO_ERR_INVALID_DEVICE_STATE, 133
USBIO_ERR_INVALID_DIRECTION, 132
USBIO_ERR_INVALID_FUNCTION_PARAM, 135
USBIO_ERR_INVALID_INBUFFER, 130
USBIO_ERR_INVALID_IOCTL, 132
USBIO_ERR_INVALID_ISO_PACKET, 133
USBIO_ERR_INVALID_OUTBUFFER, 130
USBIO_ERR_INVALID_PARAMETER, 127
USBIO_ERR_INVALID_PARAM, 133
USBIO_ERR_INVALID_PIPE_FLAGS, 129
USBIO_ERR_INVALID_PIPE_HANDLE, 127
USBIO_ERR_INVALID_POWER_STATE, 133
USBIO_ERR_INVALID_RECIPIENT, 132
USBIO_ERR_INVALID_TYPE, 132
USBIO_ERR_INVALID_URB_FUNCTION, 127
USBIO_ERR_ISO_NA_LATE_USBPORT, 130
USBIO_ERR_ISO_NOT_ACCESSED_BY_HW, 130
USBIO_ERR_ISO_NOT_ACCESSED_LATE, 130
USBIO_ERR_ISO_RESTRICTION, 134
USBIO_ERR_ISO_TD_ERROR, 130
USBIO_ERR_ISOCH_REQUEST_FAILED, 128
USBIO_ERR_LOAD_SETUP_API_FAILED, 136
USBIO_ERR_NO_BANDWIDTH, 127
USBIO_ERR_NO_MEMORY, 127

USBIO_ERR_NO_SUCH_DEVICE_INSTANCE, 135
USBIO_ERR_NOT_ACCESSED, 126
USBIO_ERR_NOT_BOUND, 131
USBIO_ERR_NOT_CONFIGURED, 131
USBIO_ERR_NOT_SUPPORTED, 128
USBIO_ERR_OPEN_PIPES, 131
USBIO_ERR_OUT_OF_ADDRESS_SPACE, 133
USBIO_ERR_OUT_OF_MEMORY, 130
USBIO_ERR_PENDING_REQUESTS, 131
USBIO_ERR_PID_CHECK_FAILURE, 125
USBIO_ERR_PIPE_NOT_FOUND, 132
USBIO_ERR_PIPE_RESTRICTION, 135
USBIO_ERR_PIPE_SIZE_RESTRICTION, 135
USBIO_ERR_POOL_EMPTY, 132
USBIO_ERR_POWER_DOWN, 134
USBIO_ERR_REQUEST_FAILED, 127
USBIO_ERR_RESERVED1, 126
USBIO_ERR_RESERVED2, 126
USBIO_ERR_SET_CONFIG_FAILED, 129
USBIO_ERR_SET_CONFIGURATION_FAILED, 134
USBIO_ERR_STALL_PID, 125
USBIO_ERR_STATUS_NOT_MAPPED, 129
USBIO_ERR_SUCCESS, 125
USBIO_ERR_TIMEOUT, 132
USBIO_ERR_TOO_MUCH_ISO_PACKETS, 132
USBIO_ERR_UNEXPECTED_PID, 125
USBIO_ERR_USBD_BUFFER_TOO_SMALL, 129
USBIO_ERR_USBD_INTERFACE_NOT_FOUND, 129
USBIO_ERR_USBD_TIMEOUT, 129
USBIO_ERR_VERSION_MISMATCH, 134
USBIO_ERR_VID_RESTRICTION, 134
USBIO_ERR_XACT_ERROR, 126
USBIO_FEATURE_REQUEST, 90
USBIO_FRAME_NUMBER, 107
USBIO_GET_CONFIGURATION_DATA, 93
USBIO_GET_INTERFACE_DATA, 95
USBIO_GET_INTERFACE, 94
USBIO_INTERFACE_CONFIGURATION_INFO, 102
USBIO_INTERFACE_SETTING, 96
USBIO_ISO_PACKET, 119
USBIO_ISO_TRANSFER_HEADER, 120
USBIO_ISO_TRANSFER, 117
USBIO_PIPE_CONFIGURATION_INFO, 104
USBIO_PIPE_CONTROL_TRANSFER, 116
USBIO_PIPE_PARAMETERS, 110
USBIO_PIPE_STATISTICS, 114
USBIO_PIPE_TYPE, 121
USBIO_QUERY_PIPE_STATISTICS, 112

USBIO_REQUEST_RECIPIENT, 122
USBIO_REQUEST_TYPE, 123
USBIO_SET_CONFIGURATION, 97
USBIO_SETUP_PIPE_STATISTICS, 111
USBIO_STATUS_REQUEST_DATA, 92
USBIO_STATUS_REQUEST, 91

Value
    Member of USBIO_CLASS_OR_VENDOR_REQUEST, 98

WaitForCompletion
    CUsbIoPipe::WaitForCompletion, 195
WriteSync
    CUsbIoPipe::WriteSync, 199
Write
    CUsbIoPipe::Write, 194