Thesycon® Systemsoftware & Consulting GmbH

USBIO COM Interface

USB Software Development Kit for Windows

COM Interface Reference Manual

Version 2.31 **November 23, 2005**

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

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

Contents

1 Introduction

The USBIO COM interface is a high level programming interface for the USBIO device driver. It is based on Microsoft's COM technology. The COM interface is an extension to the native driver interface. It is more convenient to use than the Win32 based native USBIO programming interface. This is especially true for programming languages that provide extensive support for COM objects.

The USBIO COM object can be used by any programming language that supports Microsoft's COM technology as Visual Basic and Delphi. The USBIO COM object supports USB 1.1 and USB 2.0. The full USBIO programming interface is supported by the USBIO COM object. The same level of functionality is available as at the native USBIO programming interface. The COM interface is designed to be easy and convenient to use and to provide high performance for time-critical operations.

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

The reader of this document is assumed to be familiar with the specification of the Universal Serial Bus, with the basics of Microsoft's COM technology, and with common aspects of Win32 based application programming.

2 Overview

The USBIO device driver provides a native programming interface that is based on the Win32 API. The Win32 functions *CreateFile()*, *DeviceIoControl()*, *ReadFile()*, and *WriteFile()* are used to communicate with the driver. In order to use this interface an application program has to deal with specific function codes and data structures. These codes and structures are defined for the programming languages C and C++, respectively. It may be difficult to handle the native USBIO programming interface in other programming languages.

The USBIO COM object provides a language independent programming interface that is based on Microsoft's COM technology. Furthermore, the USBIO COM interface simplifies the handling of asynchronous read and write operations. It implements internal worker-threads which handle read and write requests with high efficiency. The COM object internally allocates a buffer pool for each USB endpoint and handles the asynchronous buffer circulation. The client application is informed by COM events when data has been received or when free buffers for writing data are available. Read and write operations at the USBIO COM interface are synchronous. That means each read or write operation initiated by the application returns immediately.

Due to this design of the data path the implementation of the client application is very simple. No multi-threading at the application level is required. Nevertheless, the full data transfer speed of the USB can be reached when the USBIO COM interface is used. The internal worker-thread based, asynchronous architecture of the data path allows continuous data transfers from or to a USB device.

2.1 Platforms

The USBIO COM interface 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)

2.2 Features

The following list summarizes the features provided by the USBIO COM interface:

- Supports USB 1.1 and USB 2.0
- Conforms to the Microsoft COM specification
- Exports an interface that is based on *IDispatch*
- Supports the complete USBIO device driver programming interface
- Supports high-performance, continuous data transfers using internal worker-threads

- Allows a simple programming model for the application
- No multi-threading at the application level required
- Provides a synchronous interface for all operations
- Supports notifications of add and remove device events

2.3 Supported Programming Languages

The following programming languages and development tools are supported by the USBIO COM object.

- Microsoft Visual Basic 6.0
- Delphi 5.0

The USBIO COM object is tested and released for use with these languages only.

2.4 Restrictions

Some restrictions that apply to the USBIO COM interface are listed below.

- The USBIO COM object conforms to the Microsoft COM specification. The exported interface *IUSBIOInterface* is based on *IDispatch*. For that reason, virtually any programming language and scripting language that includes support for COM should be able to use the USBIO COM object. However, this is not tested and not supported by Thesycon for other languages than those listed in section 2.3.
- Microsoft's COM technology has an extension known as Distributed COM (DCOM). It allows to access COM objects on remote computers by using a network connection. The current version of the USBIO COM object is not tested and not released for use with the DCOM technology. Thesycon does not support such usage scenarios.

3 Architecture

Figure 1 below shows a USBIO client application, the USBIO COM object, the USBIO device driver, and its relation to the USB driver stack that is part of the Windows operating system. All drivers are embedded in the WDM (Windows Driver Model) layered architecture.

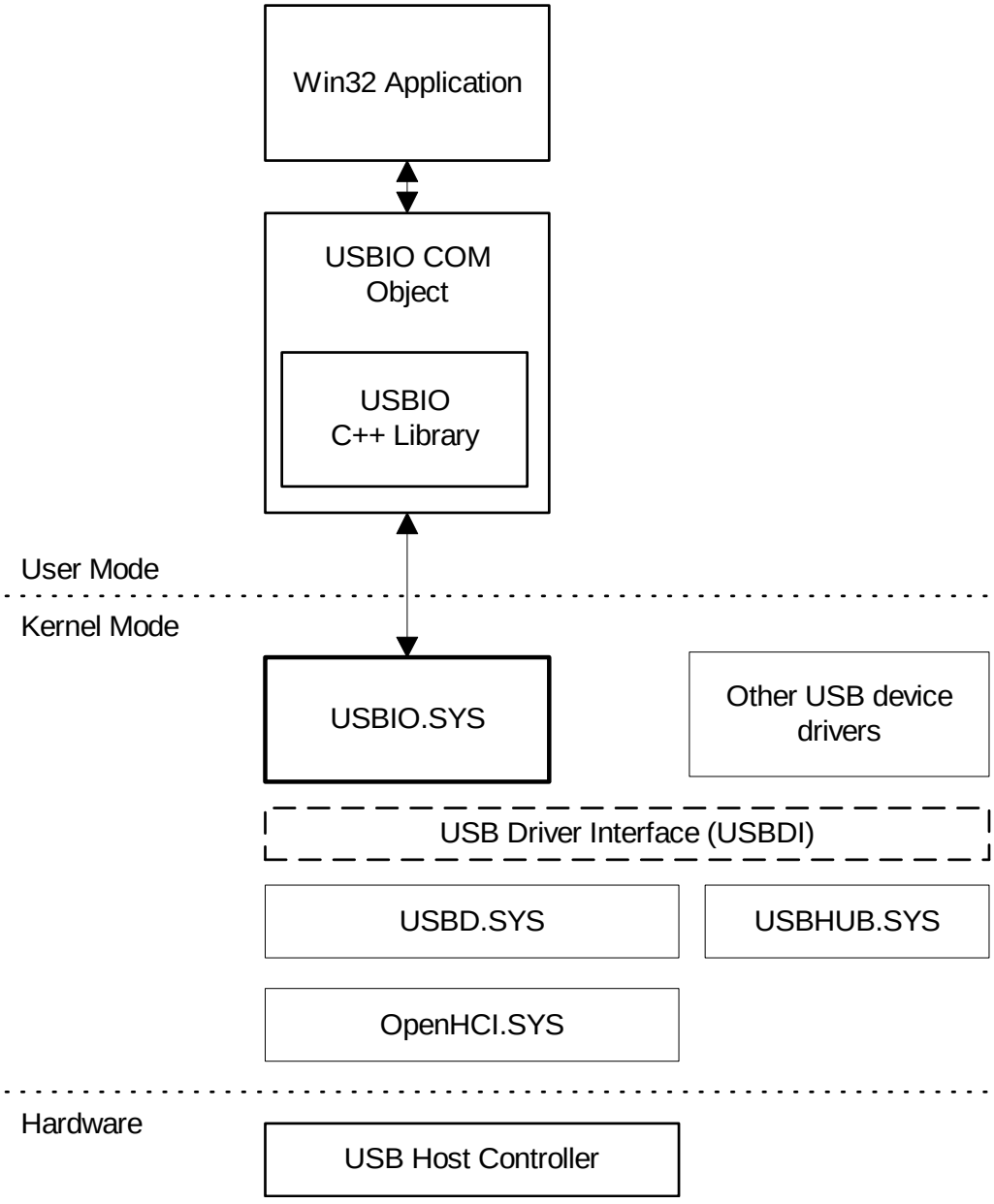

Figure 1: USB Driver Stack

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.
- USBIO COM object is the programming language independent interface for the USBIO device driver. It is built on top of the native USBIO programming interface. The implementation of the USBIO COM object is based on the USBIO C++ class library which is part of the USBIO package.
- Win32 Application is a client application that uses the USBIO COM object to control a USB device. It is written in a high level programming language like Visual Basic or Delphi.

Please refer to the USBIO Reference Manual for a detailed description of the driver architecture.

3.1 USBIO COM Object

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. Refer to the USBIO Reference Manual for further details.

An instance of the USBIO COM object is associated with a physical USB device that is connected to the USB. The device is controlled by the USBIO device driver. That means the USBIO device driver must have been installed for this device.

The USBIO COM instance is used to perform device-related operations. This includes sending of control requests like **SetConfiguration** to the USB device.

A USBIO COM instance can be associated with an endpoint of the USB device. An endpoint of the device is represented by a USBIO pipe object. Thus, the USBIO COM instance will be bound to a pipe. This binding is established by means of the function **Bind**. If a USBIO COM instance is bound to a pipe, or to an endpoint, respectively it is used to perform all pipe-related operations.

Particularly, this includes a data transfer by means of read and write operations from or to the USB endpoint.

A USBIO COM instance can be bound to exactly one endpoint only. In order to control several endpoints one instance of the USBIO COM object is required for each of them. Thus, several instances have to be created by the application.

Figure 2 shows the relations between USBIO COM object instances and USB endpoints.

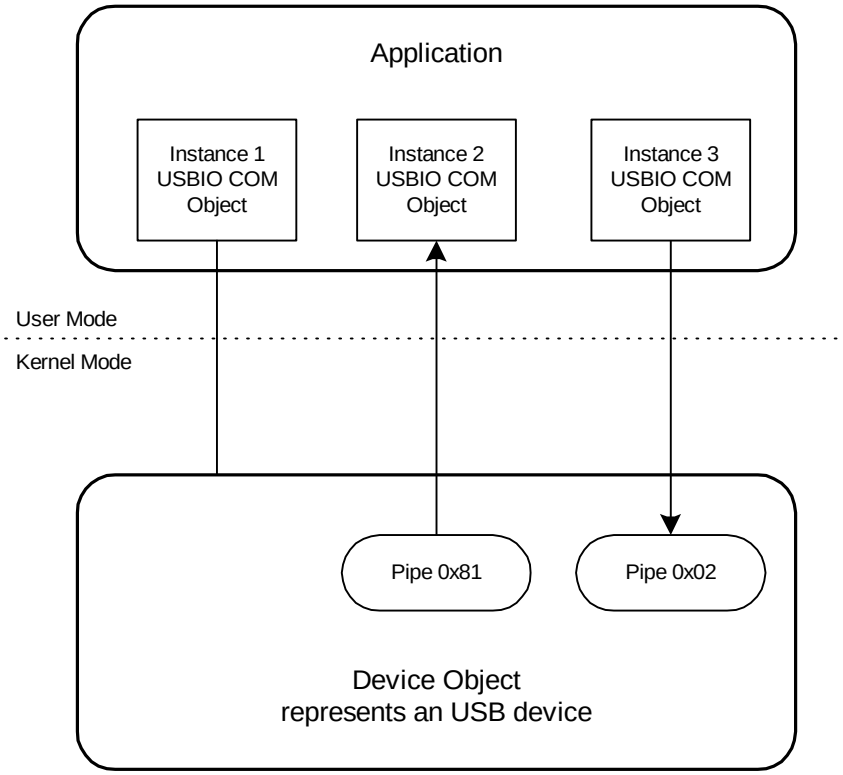

Figure 2: USBIO COM object instances and USB endpoints

As shown in Figure 2 Instance 1 is associated with the USB device but not bound to a pipe. It can be used to perform device-related operations only. Typically, such an instance will be used by an application to issue requests to endpoint zero (EP0) of the device.

Instance 2 is bound to the pipe object that represents endpoint 0x81 of the device which is used to transfer data from the device to the application (IN direction). Consequently, the application will use Instance 2 to perform read operations.

Instance 3 is bound to the pipe object that represents endpoint 0x02 of the device which is used to transfer data from the application to the device (OUT direction). Consequently, the application will use Instance 3 to perform write operations.

Note: While Instance 1 cannot be used to perform any pipe-related operations Instance 2 and Instance 3 can be used to perform all device-related operations as well. This is because all the instances are associated with the USB device in question.

4 Installation and Usage

4.1 Registering the USBIO COM Object

The implementation of the USBIO COM interface is contained in a dynamic link library named USBIOCOM.DLL. Before the USBIO COM interface can be used it must be registered on a computer. This makes the interface visible to applications and development tools.

The registration is performed automatically by the setup program during installation of the USBIO package. To register the USBIO COM object manually the following command line can be used:

```
regsvr32 usbiocom.dll
```

The utility regsvr32.exe is part of the Windows operating system. The registration has to be repeated if USBIOCOM.DLL is moved to a different folder.

Note: On Windows 2000 and Windows XP administrator privileges are required to perform the registration.

The USBIO COM interface requires the USBIO device driver for communication with the USB device. Therefore, the USBIO device driver must be installed properly for the USB device in question. For detailed installation instructions refer to the USBIO Reference Manual.

4.2 Unregistering the USBIO COM Object

The USBIO COM interface can be unregistered with the following command:

```
regsvr32 /u usbiocom.dll
```

After the USBIO COM interface is unregistered on a computer it is no longer visible to development tools and cannot be used by any application.

Note: On Windows 2000 and Windows XP administrator privileges are required to unregister the interface.

4.3 Using the USBIO COM Object with Visual Basic

The USBIO COM object has to be registered on the development machine as described in section 4.1. Note that this is done automatically when the USBIO package is installed on the computer.

Start Visual Basic and create a new project. Then select the menu item *Project/References*. A dialog pops up that shows a list of available COM objects. Select the entry labeled *USBIOCOM 2.0 Type Library* and confirm the dialog. This makes the USBIO COM object known to Visual Basic.

Create a global variable of type *USBIOCOMLib.USBIOInterface* as shown in the following example. Use the *WithEvents* attribute.

```
Dim WithEvents Device1 As USBIOCOMLib.USBIOInterface2
Dim Status As Long
Dim Devices As Long
```

Create an instance of the USBIO COM object and assign it to the variable. This should be done in a startup routine, e.g. *Form_Load*. Note that this instruction causes an exception if the USBIO COM object is not installed on the computer. The exception can be handled by the statement *On Error Resume Next* as shown below.

```
Device1 = New USBIOCOMLib.USBIOInterface2
On Error Resume Next
```

The instance can be used to call functions of the USBIO COM object as follows.

```
rem enumerate all devices
Device1.EnumerateDevices "{325ddf96-938c-11d3-9e34-0080c82727f4}", Devices
rem open the first USB device connected to the USBIO device driver
Device1.OpenDevice 0, Status
```

The first parameter in the function call of **EnumerateDevices** is the default GUID of the USBIO device driver. The GUID of the driver can be changed by editing the INF file. See the chapter *Building a Customized Driver Setup* in the USBIO Reference Manual for details. The function **EnumerateDevices** returns the number of available devices. The function **OpenDevice** opens the first USB device which has the device index zero.

When an event is sent from the USBIO COM object instance to the application an event-specific call-back function is called. The call-back functions have to be implemented as follows.

```
Sub Device1_ReadComplete(ByVal obj As Object)
Sub Device1_WriteComplete(ByVal obj As Object)
Sub Device1_WriteStatusAvailable(ByVal obj As Object)
```

For a detailed description of the event call-back functions **ReadComplete**, **WriteComplete**, and **WriteStatusAvailable** refer to section 6.

4.4 Using the USBIO COM Object with Delphi

The USBIO COM object has to be registered on the development machine as described in section 4.1. Note that this is done automatically when the USBIO package is installed on the computer.

Before it can be used the USBIO COM object must be imported into a Delphi project. To do that select the menu item *Projects/Import Typelibrary*. A list with available objects will be shown. Select the entry labeled *USBIOCOM 2.0 Type Library (Version 2.0)*. Make sure the check box labeled *Create a Component Wrapper* is active and press the *Install* button. Delphi automatically generates a wrapper class for the USBIO COM object in a unit called *USBIOCOMLib_TLB*. This unit has to be included in the project.

Note: The name of the wrapper class generated during the installation can be adjusted.

Create a variable of type *TUSBIOInterface2*. For example:

```
var usbiocom :  TUSBIOInterface2;
```

The variable has to be initialized during startup of the application, e.g. in FormCreate. This is shown below.

```
// create the USBIOInterface instance
usbiocom := TUSBIOInterface2.Create(MainDlg);
// set completion handlers
// note: The handler must be a method defined in an object.
usbiocom.OnReadComplete := MainDlg.ReadComplete;
usbiocom.OnWriteComplete := MainDlg.WriteComplete;
usbiocom.OnWriteStatusAvailable := MainDlg.WriteStatusAvailable;
// connect to the COM object, handle possible exceptions
try
  usbiocom.Connect;
except
  MessageDlg('USBIOCOM not installed', mtError, [mbOk], 0);
end;
```

The event call-back functions assigned to the USBIO COM instance have to be implemented as shown in the following examples.

```
procedure TMainDlg.ReadComplete(Sender: TObject; var Obj: OleVariant);
procedure TMainDlg.WriteComplete(Sender: TObject; var Obj: OleVariant);
procedure TMainDlg.WriteStatusAvailable(Sender: TObject; var Obj: OleVariant);
```

For a detailed description of the event call-back functions **ReadComplete**, **WriteComplete**, and **WriteStatusAvailable** refer to section 6.

4.5 Using the USBIO COM Object with other Programming Languages

In general, the USBIO COM object can be used with any programming language which supports Microsoft's COM technology. Refer to the documentation of the development tools for information on how to install and use type libraries.

Note: Thesycon does support the programming languages and development tools listed in section 2.3 only. Using the USBIO COM object with other languages and tools is not tested and not supported.

4.6 Debugging and Trace Support

The implementation of the USBIO COM object in USBIOCOM.DLL is available in two versions a release build and a debug build. Both versions are part of the distribution. The release build is installed by default. The debug build is located in the subdirectory *COMobj\Debug*. Both versions of the DLL provide the same interface and behave in the same way. The debug build includes traces that are useful for advanced debugging.

To install the debug build the file *COMobj\Debug\USBIOCOM.DLL* has to be registered. This is done in the same way as described in section 4.1. In the subdirectory *COMobj\Debug* run the following command:

```
regsvr32 usbiocom.dll
```

To install the release build again run the same command in the subdirectory *COMobj*.

After the debug build has been installed the USBIO COM object should be initialized once by starting and terminating the application. This will create the following registry key:

```
HKEY_LOCAL_MACHINE\Software\Thesycon\USBIO\USBIOCOM
```

In this registry key there is a value named **`DbgMask`**. This parameter allows to enable and disable debug traces. Every bit in the DWORD value enables a group of traces when it is set to one. The following table shows the assignment of bit positions.

Table 1: Trace control bits of the USBIO COM interface

Bit Nb	Meaning
0	Errors
1	Warnings
2	Information
4	Internal function calls
5	Internal read and write
6	Internal thread
8	COM function calls
9	COM read and write calls

For example, a value of 0x107 enables error, warning, and informational messages, and in addition

traces associated with COM function calls. All other trace messages are disabled.

The trace messages are output be means of the Win32 function *OutputDebugString* and therefore will be send to the system debugger.

In addition, the parameter **`DbgFile`** in the registry key mentioned above allows to define a file that receives the trace messages. The **`DbgFile`** value is of type REG_SZ.

Note: Enabling trace messages has an impact on the timing behavior of the USBIO COM object. Performance problems can occur when trace messages are produced.

5 USBIO COM Programming

5.1 Initializing a USB Device Instance

The first required step is the creation of a USBIO COM object instance. Then the call-back event handlers have to be initialized. The implementation of these initialization steps depends on the programming language used. For more information refer to sections 4.3 to 4.5 and to the documentation of the programming language.

As described in section 3.1 a USBIO COM object instance is associated with a USB device. The following steps have to be performed on an instance to establish this association.

EnumerateDevices( )

This function returns the number of USB devices currently connected to the USBIO device driver. If zero is returned either no devices are connected to the computer or the USBIO driver is not installed properly for the device(s) in question.

OpenDevice( )

This function opens one of the available devices. The device is selected by means of a zero-based index. After this call succeeds the USBIO COM object instance is attached to the specified device.

AddInterface( )

This function adds information on a USB interface provided by the device to an internal list. If the USB device has several interfaces this function has to be called once for each interface.

SetConfiguration( )

This function sets the USB device to the configured state. All the interfaces previously added by means of AddInterface() will be configured.

In order to transfer data to or from an endpoint of the device the USBIO COM object instance has to be bound to a USBIO pipe as described in the next section.

5.2 Initializing a USB Endpoint Instance

As described in section 3.1 a USBIO COM object instance has to be bound to a USB endpoint before any data transfer can take place. The following steps have to be performed on an instance in order to establish the association with an endpoint.

Note that any USBIO COM object instance has to be created and initialized before it can be used, as already mentioned in the previous section. For more information refer to sections 4.3 to 4.5 and to the documentation of the programming language.

OpenDevice( )

This function opens one of the available devices. The device is selected by means of a zero-based index. After this call succeeds the USBIO COM object instance is attached to the specified device.

Bind( )

This function binds the instance to the endpoint that is specified on the call. After the call succeeds the data path is established. Depending on the direction of the endpoint either read or write operations can be performed now.

The sequence described above has to be repeated for each endpoint that is to be used for transferring data. A separate USBIO COM object instance has to be created and initialized for each endpoint. This is because an instance can be bound to one endpoint only (see section 3.1).

5.3 Set Up a Data Transfer in IN Direction

To initiate a data transfer from a USB endpoint to the application the function **StartReading** is called on the USBIO COM object instance that is attached to the endpoint.

The USBIO COM object issues an event to inform the application that data has been received from the device. This results in a call to the call-back function **ReadComplete**.

The application reads the data received from the device by means of the functions **ReadData** and **ReadIsoData**, depending on the type of the endpoint.

To terminate the data transfer the function **StopReading** is called.

For a detailed description of all functions mentioned above refer to section 6.

5.4 Set Up a Data Transfer in OUT Direction

To initiate a data transfer from the application to a USB endpoint the function **StartWriting** is called on the USBIO COM object instance that is attached to the endpoint.

The application sends data to the device by means of the functions **WriteData** and **WriteIsoData**, depending on the type of the respective endpoint. The application stops writing when the error code **`USBIO_ERR_NO_BUFFER`** is returned. This indicates that all the internal write buffers are exhausted.

The USBIO COM object issues an event to inform the application when write buffers become available. This results in a call to the call-back function **WriteComplete**. The application continues sending data until all the buffers are filled again.

To terminate the data transfer the function **StopWriting** is called.

For a detailed description of all functions mentioned above refer to section 6.

5.5 Performance Considerations

The USBIO COM object supports low-speed, full-speed, and high-speed mode as specified by the USB specification 1.1 and USB specification 2.0 respectively. The maximum data rate is supported for each of the transfer types bulk, interrupt, and isochronous. However, an application that uses the USBIO COM interface causes more CPU load than an application that uses the native USBIO device driver interface. The reason for this fact is the overhead associated with calls to the COM object. This overhead includes parameter data conversion, memory allocation and copy operations, and thread switches. While some effort was made to minimize the overhead it is not possible to eliminate it completely.

In order to minimize the CPU load caused by a data transfer an application has to take care to arrange the data buffers properly. The number of calls to the USBIO COM object in a given time interval should be minimized. This can be achieved by sizing each buffer in such a way that it contains data for 40 to 100 milliseconds. Smaller buffers cause a higher CPU load because more function calls and thread switches are required in a given time interval.

In case of a data transfer from the host to the device (OUT endpoint) the USBIO COM object and the device driver stack automatically handle the fragmentation of a data buffer into smaller sub-buffers. The size of a sub-buffer corresponds to the size of the endpoint's FIFO.

In case of a data transfer from the device to the host (IN endpoint) the USBIO COM object and the device driver stack automatically handle the concatenation of sub-buffers into larger data buffers.

5.6 Bulk Endpoints

Usually, bulk endpoints are used to transfer large amounts of data that do not have special timing requirements. The transfer of bulk packets is protected by a checksum (CRC). A packet is repeated up to three times if a transmission error occurs.

According to the USB specification the maximum size of a bulk packet is 64 bytes for USB 1.1 and 512 bytes for USB 2.0. The following discussion assumes a device that operates in full-speed mode (12 Mbit/s). The maximum data rate that can be achieved with bulk transfers is about 1 megabyte per second. At this rate a suitable size of the data buffers is 64 kilobytes (1024 bulk packets). Then a buffer will be completed every 62.5 milliseconds. Consequently, the application has to process 16 data buffers per second. These calculations can be applied to high-speed mode (480 Mbit/s) accordingly.

As shown in the example the buffer size should be choosen according to the maximum data rate that is expected. The number of events (completed data buffers) to be processed per second is proportional to the CPU time that is consumed by the data transfer. As a rule of thumb the number of events to be processed by an application should not exceed 50 per second.

On read operations from a bulk endpoint (bulk IN transfer) the way buffers are handled by the USB driver stack has to be considered. A data buffer is completed and returned to the USBIO COM object either if it is filled completely (filled up with 1024 bulk packets in the example above) or if a short packet is received from the USB device. A short packet is a bulk packet whose length is shorter than the FIFO size of the endpoint. The packet length can be zero as well (zero packet). That way, the device is able to define the amount of data that is placed in a buffer.

On bulk transfers the size of a data buffer has to be a multiple of the endpoint's FIFO size. This condition is enforced by the USB device driver stack. If it is violated the buffer will be completed

with an error status (buffer overrun).

5.7 Interrupt Endpoints

Typically, interrupt endpoints are used to communicate asynchronous events. The transfer of data packets is protected by a checksum (CRC). A packet is repeated up to three times if a transmission error occurs.

If an application needs to receive an interrupt data packet immediately then the size of the data buffers should be set to be equal to the FIFO size of the interrupt endpoint. Note that in this configuration a high interrupt packet rate will cause a high CPU load.

If some kind of data streaming is implemented by means of an interrupt endpoint then larger data buffers should be used. The same conditions are true as stated in the previous section for bulk IN transfers.

5.8 Isochronous Endpoints

Isochronous endpoints are used to transfer data streams with certain timing constraints. Typically, isochronous data are audio or video streams. The transfer of isochronous packets is protected by a checksum (CRC). Packets will not be repeated by the bus protocol if transmission errors occur. However, the transmission status is reported for each isochronous packet.

A data buffer to be used for isochronous transfers is divided into sub-buffers. A sub-buffer corresponds to an isochronous packet which is transferred in a USB frame or a microframe respectively. Thus, in full-speed mode one sub-buffer is transferred every millisecond. In high-speed mode the packet rate can vary. It is defined by the device and reported in the endpoint descriptor. See also the USBIO Reference Manual for a detailed description of the layout of isochronous data buffers.

The number of sub-buffers (isochronous frames) placed in each data buffer determines the rate at which the application has to handle buffer completion events. For example, in full-speed mode 20 frames per data buffer mean that a buffer is completed every 20 milliseconds. The application has to process 50 data buffers per second in this case.

The number of events (completed data buffers) to be processed per second is proportional to the CPU time that is consumed by the isochronous data transfer. Thus, with respect to a low CPU load a large number of frames should be placed in a data buffer. On the other hand, this will increase the delay that is involved in the data path. In the example above the delay is at least 20 milliseconds. Consequently, with respect to a low transmission delay a small number of frames (e.g. 8) should be placed in each data buffer.

As the discussion shows, an application designer has to find a compromise between the CPU load caused by the buffer handling overhead and the delay involved in the data path. The decision should be made based on the individual application scenario.

The total number of sub-buffers (frames) per data buffer is limited to 1024. This condition is enforced by the USB device driver stack. Due to internal resource constraints (kernel memory) the USBIO device driver enforces an upper limit on the number of frames per buffer as well. By default, this limit is set to 512 on Windows 2000 and Windows XP, and set to 64 on Windows 98 and Windows ME. The value can be changed by means of a configuration parameter defined in the setup information file *usbio.inf*. For more information refer to the USBIO Reference Manual.

5.9 Control Endpoints

According to the USB specification endpoint 0 (EP0) is used for control transfers. However, it is possible that a device provides additional control-type endpoints. Control transfers are used to implement the standard requests defined by the USB specification, chapter 9. A device can define additional requests to be handled by endpoint 0. Such requests are called class-specific or vendor-specific.

A control transfer consists of three stages: control stage, data stage, and handshake stage. The data stage of a class-specific or vendor-specific request on endpoint 0 is limited to 4096 bytes.

5.10 An Example Scenario

This section shows an example of a data transfer from a bulk IN endpoint to the host in full-speed mode. The test setup is as follows:

- USB 1.1 compliant device in full-speed mode (12 Mbit/s)
- FIFO size of the bulk endpoint: 64 bytes
- Host CPU: Intel Pentium III, 866 MHz
- USB host controller: Universal Host Controller
- Operating system: Windows 2000

The test device sends data via the bulk IN endpoint as fast as possible. However, it is not able to answer with a data token to each IN token it receives from the host. Therefore, the test device limits the achievable data rate to approximately 810 KBytes per second.

Table 2: Data rate and CPU load as a function of the buffer size

Buffer size	Buffer completion interval	Data rate	CPU load
64 bytes	~ 3 ms	25 Kbytes/s	100%
128 bytes	~ 3 ms	52 Kbytes/s	100%
256 bytes	~ 3 ms	100 Kbytes/s	100%
1024 bytes	~ 3.5 ms	300 Kbytes/s	90%
4096 bytes	~ 6 ms	670 Kbytes/s	45%
16000 bytes	~ 20 ms	810 Kbytes/s	23%
64000 bytes	~ 80 ms	810 Kbytes/s	9%

The table shows that large buffers are needed to achieve the maximum data rate of the device. Furthermore, small buffers cause a high CPU load because for every buffer a transition between application and USBIO COM object is required. Each transition (function call and return) needs a certain amount of CPU cycles. The CPU load can be reduced by reducing the number of transitions per time interval. This behavior is discussed in sections 5.5 and 5.6.

The high CPU load can cause problems with the application's behavior. At a CPU load of 100% the application may not be able to process messages it receives from the user interface. This is because

the application's message loop is permanently busy with processing of messages generated by the USBIO COM object due to buffer completion events. This situation can be avoided when the buffer size is chosen in such a way that the resulting message rate is tolerable. As illustrated in Table 2 a rate of less than 50 events per second (20 ms interval) is acceptable.

When small buffers are used only a low data rate can be achieved. Aside from the high CPU load this is caused by the particular behavior of the USB host controller. After a data buffer is filled completely the host controller stops sending IN tokens to the respective endpoint. It continues sending IN tokens in the next USB frame (Open Host Controller) or in the USB frame after next (Universal Host Controller). Because one USB frame corresponds to 1 millisecond this wastes bandwith. In order to reduce the impact of this behavior on the resulting data rate large buffers should be used.

6 Programming Interface Reference

6.1 IUSBIOInterface2 Interface

The **`IUSBIOInterface2`** interface is exported by the USBIO COM object and enables applications to control an instance of that object.

The **`IUSBIOInterface2`** interface extends the **`IUSBIOInterface`** interface which was supported by earlier versions of the USBIO Development Kit.

The following table summarizes the members of **`IUSBIOInterface2`**. The methods and properties are described in detail in this section.

Table 3: Members of IUSBIOInterface2

Member	Description
EnumerateDevices	Enumerate USB devices
OpenDevice	Open a USB device
CloseDevice	Close a USB device
DevicePathName	The path name of the driver interface
GetDriverInfo	Get version information about the driver
IsCheckedBuild	TRUE if a debug build of the driver is running
IsDemoVersion	TRUE if a demo version of the driver is running
IsLightVersion	TRUE if a light version of the driver is running
DeviceOptions	Set or retrieve device options
DeviceRequestTimeout	Set or retrieve the request timeout parameter
GetDescriptor	Get a USB descriptor
GetDeviceDescriptor	Get the USB device descriptor
GetConfigurationDescriptor	Get a USB configuration descriptor
GetStringDescriptor	Get a USB string descriptor
SetDescriptor	Set a descriptor
AddInterface	Add a USB interface to the configuration
DeleteInterfaces	Delete all interfaces
SetConfiguration	Configure the USB device
GetConfiguration	Get the current configuration value
UnconfigureDevice	Set the USB device to unconfigured state
SetInterface	Change the alternate setting
GetInterface	Get the current alternate setting
ClassOrVendorInRequest	Issue a class or vendor IN request
ClassOrVendorOutRequest	Issue a class or vendor OUT request
SetFeature	Issue a set feature request
ClearFeature	Issue a clear feature request
GetDevicePowerState	Get current device power state
SetDevicePowerState	Set the device power state
ResetDevice	Force a USB reset
CyclePort	Simulate a disconnect/connect cycle
GetStatus	Issue a get status request
GetCurrentFrameNumber	Get current USB frame number

ErrorText	Translate an error code to a description text
Bind	Bind a USBIO COM instance to an endpoint
Unbind	Delete a binding
StartReading	Start data transfer from an endpoint
ReadData	Read data from an endpoint
ReadIsoData	Read data from an isochronous endpoint
StopReading	Stop data transfer from an endpoint
StartWriting	Start data transfer to an endpoint
WriteData	Write data to an endpoint
GetWriteStatus	Get completion status of a write operation
WriteIsoData	Write data to an isochronous endpoint
GetIsoWriteStatus	Get completion status of a write operation
StopWriting	Stop data transfer to an endpoint
ResetPipe	Reset pipe, clear an error condition
AbortPipe	Abort all pending I/O requests
ShortTransferOK	Allow or disallow short transfers
EndpointFifoSize	Query the size of the endpoint's FIFO
EnablePnPNotification	Enable Plug and Play notification events
DisablePnPNotification	Disable Plug and Play notification events
GetBandwidthInfo	Get information on the USB bandwidth consumption
IsOperatingAtHighSpeed	TRUE if a USB 2.0 device is operating at high-speed (480 Mbit/s)
SetupPipeStatistics	Enable and configure statistical analysis for a pipe
QueryPipeStatistics	Query statistical data related to the pipe

EnumerateDevices Method

This function enumerates all USB devices which are currently connected to the system and controlled by the USBIO device driver.

Definition

```
HRESULT
EnumerateDevices(
     [in] BSTR GUIDDriverInterface,
     [out] int* NumberOfDevices
     );
```

Parameters

GUIDDriverInterface
A string representation of a GUID that identifies the driver interface exported by the USBIO driver. This can be the default USBIO GUID {325ddf96-938c-11d3-9e34-0080c82727f4} that is defined in *usbio_i.h*. However, it is strongly recommended that a private GUID is used which has been generated by a tool like *guidgen.exe*. This is important in order to differentiate between various customizations of the USBIO product.

For the USBIO device driver the private GUID has to be defined in the setup information file *usbio.inf*. See the USBIO Reference Manual and the *usbio.inf* file for details.

NumberOfDevices
A variable that receives the number of USB devices which are currently connected to the USBIO device driver. The returned value can be zero which indicates that no devices are available. A return value of -1 indicates that the format of the specified GUID is invalid.

Comments

The function builds an internal list of currently available USB devices. Each device from this list can be opened by means of the function **OpenDevice**. A device is identified by a zero-based index. Thus, valid device numbers are in the range of zero to NumberOfDevices-1. The index of a device is temporary and will be re-assigned on the next call to this function.

Note that the internal device list is global. All instances of the USBIO COM object do access the same list. When **OpenDevice** is called on an instance the global device list that was built by a call to EnumerateDevices will be consulted to locate the device identified by the device index passed to the function. Because the list is global it does not make any difference which USBIO COM object instance is used to call EnumerateDevices. Besides, it is sufficient to call EnumerateDevices one one instance.

After a USB device is connected to the system this function has to be called before the device can be opened and used for I/O operations. Consequently, EnumerateDevices has to be called before **OpenDevice** is called.

See Also

OpenDevice (page 33)
CloseDevice (page 35)
DevicePathName (page 36)

OpenDevice Method

This function opens a USB device which is currently connected to the USBIO device driver and has been enumerated by **EnumerateDevices**.

Definition

```
HRESULT
OpenDevice(
     [in] int DeviceNumber,
     [out] int* Status
     );
```

Parameters

DeviceNumber
Identifies the device to be opened by its zero-based index in the internal device list built by **EnumerateDevices**. Valid device numbers range from zero to NumberOfDevices-1. NumberOfDevices is returned by **EnumerateDevices**.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

To identify the device OpenDevice uses the internal device list which has been built by a call to **EnumerateDevices**. The index of a device is temporary and will be re-assigned on the next call to **EnumerateDevices**. When more than one USB device is present on the bus then there is no guarantee that an one-to-one relation of device numbers and physical devices will be maintained over several calls to **EnumerateDevices**. In order to unambiguously identify a physical device an application should query USB descriptors, e.g. the USB device descriptor, and the serial number.

Note that the internal device list built by **EnumerateDevices** is global. Regardless of the object instance on that OpenDevice is called the same global list will be used to locate the device identified by **DeviceNumber**. For more information see also the description of **EnumerateDevices**.

It is recommended that device enumeration is done in a two step process:
(1) Call EnumerateDevices on one object instance.
(2) Call OpenDevice on each object instance to be associated with a device.

OpenDevice fails if the major USBIO driver version is different from the version of the USBIO COM object. Make sure that the USBIO device driver and the USBIO COM interface versions do match.

The operation fails if the device is acquired by a different process.

See Also

CloseDevice (page 35)
EnumerateDevices (page 31)
DevicePathName (page 36)
AcquireDevice (page 115)

CloseDevice Method

This function closes a USB device previously opened by a call to **OpenDevice**.

Definition

```
HRESULT
CloseDevice();
```

Comments

All read and write operations associated with the instance will be stopped by this call.

If the last USBIO COM object instance for a device is closed either the device is set to the unconfigured state or a USB reset is forced, depending on the configuration of the USBIO device driver. The behavior can be configured by means of the function **DeviceOptions**. The default behavior is defined by means of registry parameters. For details refer to the USBIO Reference Manual.

See Also

OpenDevice (page 33)
EnumerateDevices (page 31)
DeviceOptions (page 42)

DevicePathName Property

This read-only property returns the full device path string of the device instance.

Definition

```
HRESULT
DevicePathName(
     [out,retval] BSTR* DevicePathName
     );
```

Parameter

DevicePathName
A variable that receives the device path string.

Comments

The device path string returned by this function can be passed to the Win32 API function *CreateFile()* to open the USB device which is controlled by the USBIO driver. Refer to the USBIO Reference Manual and the Win32 documentation for further details.

This property can only be read after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)

GetDriverInfo Method

This function returns version information on the USBIO device driver.

Definition

```
HRESULT
GetDriverInfo(
     [out] int* APIVersion,
     [out] int* DriverVersion,
     [out] int* DriverBuildNumber,
     [out] int* Flags,
     [out] int* Status
     );
```

Parameters

APIVersion
A variable that receives the version number of the native USBIO device driver API. The format is as follows: bits 15..8 = major version, bits 7..0 = minor version. The numbers are encoded in BCD format.

DriverVersion
A variable that receives the version number of the USBIO device driver executable. The format is as follows: bits 15..8 = major version, bits 7..0 = minor version.

DriverBuildNumber
A variable that receives the build number of the USBIO device driver executable.

Flags
A variable that receives additional information encoded as bit flags. The value is zero or any combination (bit-wise or) of the following values.

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

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

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

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
IsCheckedBuild (page 39)
IsDemoVersion (page 40)
IsLightVersion (page 41)
USBIOCOM_INFO_FLAGS (page 124)

IsCheckedBuild Property

This read-only property returns TRUE if the checked (debug) build of the USBIO device driver is currently running, FALSE otherwise.

Definition

```
HRESULT
IsCheckedBuild(
     [out,retval] BOOL* Checked
     );
```

Parameter

Checked
A variable that will be set to TRUE or FALSE.

Comments

The USBIO driver debug build is useful for investigation of problems and advanced tracing. Refer to the USBIO Reference Manual for more information.

This property can only be read after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
IsDemoVersion (page 40)
IsLightVersion (page 41)
GetDriverInfo (page 37)

IsDemoVersion Property

This read-only property returns TRUE if the the demo version of the USBIO device driver is currently running, FALSE otherwise.

Definition

```
HRESULT
IsDemoVersion(
     [out,retval] BOOL* DemoVersion
     );
```

Parameter

DemoVersion
A variable that will be set to TRUE or FALSE.

Comments

The USBIO demo version enforces some restrictions. Refer to *ReadMe.txt* for detailed information.

This property can only be read after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
IsCheckedBuild (page 39)
IsLightVersion (page 41)
GetDriverInfo (page 37)

IsLightVersion Property

This read-only property returns TRUE if the the light version of the USBIO device driver is currently running, FALSE otherwise.

Definition

```
HRESULT
IsLightVersion(
     [out,retval] BOOL* LightVersion
     );
```

Parameter

LightVersion
A variable that will be set to TRUE or FALSE.

Comments

The USBIO light version enforces some restrictions. Refer to *ReadMe.txt* for detailed information.

This property can only be read after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
IsCheckedBuild (page 39)
IsDemoVersion (page 40)
GetDriverInfo (page 37)

DeviceOptions Property

Reading this property retrieves the current options set for the device instance.

Definition

```
HRESULT
DeviceOptions(
     [out,retval] int* Options
     );
```

Parameter

Options
A variable that receives the current device options, encoded as bit flags. See below for a detailed description.

DeviceOptions Property

Writing this property sets options for the device instance.

Definition

```
HRESULT
DeviceOptions(
     [in] int Options
     );
```

Parameter

Options
Specifies device options to be set, encoded as bit flags. The value is zero or any combination (bit-wise or) of the following values.

USBIOCOM_RESET_DEVICE_ON_CLOSE
If this option is set a USB device reset is sent to the device after the last USBIO COM object instance has closed the device by a call to **CloseDevice**.

USBIOCOM_UNCONFIGURE_ON_CLOSE
If this option is set the USB device will be unconfigured after the last USBIO COM object instance has closed the device by a call to **CloseDevice**.

USBIOCOM_ENABLE_REMOTE_WAKEUP
If this option is set the remote wake-up feature is enabled for the device. At least one USBIO COM object instance must be open for the device to allow the remote wake-up event to occur.

Comments

Default device options are stored in the registry during USBIO driver installation. The default value can be changed in the INF file or in the registry. Device options set by means of this property 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.

This property can only be read or written after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
USBIOCOM_DEVICE_OPTION_FLAGS (page 125)
DeviceRequestTimeout (page 44)

DeviceRequestTimeout Property

Reading this property retrieves the current time-out interval for control requests set for the device instance.

Definition

```
HRESULT
DeviceRequestTimeout(
     [out,retval] int* pVal
     );
```

Parameter

pVal
A variable that receives the current time-out interval. See below for a detailed description.

DeviceRequestTimeout Property

Writing this property sets the time-out interval for control requests for the device instance.

Definition

```
HRESULT
DeviceRequestTimeout(
     [in] int newVal
     );
```

Parameter

newVal
Specifies the time-out interval, in ms. A value of zero specifies an infinite interval. This forces the driver to wait until a request is completed.

Comments

This property allows to retrieve or set the time-out interval that applies to all control requests. The time-out interval is in effect for USB control requests only, particularly this includes EP0 requests. The time-out will not affect data transmission from or to endpoints.

Note that setting an infinite time-out interval may be useful to halt the device's firmware in a debugger.

A default time-out interval is stored in the registry during USBIO driver installation. The default value can be changed in the INF file or in the registry. A time-out interval set by means of this property is 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.

This property can only be read or written after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
DeviceOptions (page 42)

GetDescriptor Method

This function retrieves a descriptor from the USB device.

Definition

```
HRESULT
GetDescriptor(
     [in,out] SAFEARRAY(unsigned char)* Descriptor,
     [in,out] int* DescSize,
     [in] int Recipient,
     [in] int DescriptorType,
     [in] int DescriptorIndex,
     [in] int LanguageId,
     [out] int* Status
     );
```

Parameters

`Descriptor`
An array type variable that receives the requested descriptor. The caller has to provide this variable. The array will not be resized by the COM object. Consequently, the array has to provide enough storage for the descriptor.

`DescSize`
A variable that specifies the number of bytes to request. This can be less than the size of the descriptor. Only the specified part of the descriptor will be returned in this case. The number of bytes requested must not exceed 4096 and must be less than or equal to the number of bytes provided in **`Descriptor`**.

After the GetDescriptor function succeeds the **`DescSize`** variable will be set to the number of bytes returned in the **`Descriptor`** array.

`Recipient`
Specifies the recipient of the GET_DESCRIPTOR request. The values are defined by the enumeration type **USBIOCOM_REQUEST_RECIPIENT**.

`DescriptorType`
Specifies the type of descriptor to get from the device. The values are defined by the enumeration type **USBIOCOM_DESCRIPTOR_TYPE**. These numerical values are defined by the Universal Serial Bus Specification, Chapter 9 and additional device class specifications.

`DescriptorIndex`
Specifies the zero-based index of the descriptor to get from the device. The meaning depends on the descriptor type. For example, in case of the device descriptor **`DescriptorIndex`** is not used and should be set to zero.

`LanguageId`
Specifies the language ID of the string descriptor to get. Set to zero for other descriptor

types.

`Status`
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

This function allows to request any descriptor from the USB device. To query common descriptor types in a more comfortable way the specialized functions **GetDeviceDescriptor**, **GetConfigurationDescriptor**, and **GetStringDescriptor** are provided.

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
GetDeviceDescriptor (page 48)
GetConfigurationDescriptor (page 49)
GetStringDescriptor (page 51)
SetDescriptor (page 53)
USBIOCOM_REQUEST_RECIPIENT (page 127)
USBIOCOM_DESCRIPTOR_TYPE (page 131)

GetDeviceDescriptor Method

This function retrieves the device descriptor from the USB device.

Definition

```
HRESULT
GetDeviceDescriptor(
     [in,out] SAFEARRAY(unsigned char)* DeviceDescriptor,
     [in,out] int* DescSize,
     [out] int* Status
     );
```

Parameters

DeviceDescriptor
An array type variable that receives the requested descriptor. The caller has to provide this variable. The array will not be resized by the COM object. Consequently, the array has to provide enough storage for the descriptor.

DescSize
A variable that specifies the number of bytes to request. This can be less than the size of the descriptor. Only the specified part of the descriptor will be returned in this case. The number of bytes requested must not exceed 4096 and must be less than or equal to the number of bytes provided in **DeviceDescriptor**.

After the GetDeviceDescriptor function succeeds the **DescSize** variable will be set to the number of bytes returned in the **DeviceDescriptor** array.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

According to the USB specification the size of the device descriptor is 18 bytes.

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
GetConfigurationDescriptor (page 49)
GetStringDescriptor (page 51)
GetDescriptor (page 46)

GetConfigurationDescriptor Method

This function retrieves a configuration descriptor from the USB device.

Definition

```
HRESULT
GetConfigurationDescriptor(
     [in,out] SAFEARRAY(unsigned char)* ConfigDescriptor,
     [in,out] int* DescSize,
     [in] unsigned char Index,
     [out] int* Status
     );
```

Parameters

`ConfigDescriptor`
An array type variable that receives the requested descriptor. The caller has to provide this variable. The array will not be resized by the COM object. Consequently, the array has to provide enough storage for the descriptor.

`DescSize`
A variable that specifies the number of bytes to request. This can be less than the size of the descriptor. Only the specified part of the descriptor will be returned in this case. The number of bytes requested must not exceed 4096 and must be less than or equal to the number of bytes provided in **`ConfigDescriptor`**.

After the GetConfigurationDescriptor function succeeds the **`DescSize`** variable will be set to the number of bytes returned in the **`ConfigDescriptor`** array.

`Index`
Specifies the zero-based index of the configuration descriptor to be retrieved. The valid range depends on the number of configurations the device supports. If the device supports one configuration only **`Index`** has to be set to zero.

`Status`
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

According to the USB specification the total size of the configuration descriptor is indicated by the bytes 3 and 4 (*wTotalLength* field) of the descriptor. Refer to the specification for detailed information.

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
GetDeviceDescriptor (page 48)
GetStringDescriptor (page 51)
GetDescriptor (page 46)

GetStringDescriptor Method

This function retrieves a string descriptor from the USB device.

Definition

```
HRESULT
GetStringDescriptor(
     [in,out] SAFEARRAY(unsigned char)* StringDescriptor,
     [in,out] int* DescSize,
     [in] unsigned char Index,
     [in] int LanguageId,
     [out] int* Status
     );
```

Parameters

`StringDescriptor`
An array type variable that receives the requested descriptor. The caller has to provide this variable. The array will not be resized by the COM object. Consequently, the array has to provide enough storage for the descriptor.

`DescSize`
A variable that specifies the number of bytes to request. This can be less than the size of the descriptor. Only the specified part of the descriptor will be returned in this case. The number of bytes requested must not exceed 4096 and must be less than or equal to the number of bytes provided in **`StringDescriptor`**.

After the GetStringDescriptor function succeeds the **`DescSize`** variable will be set to the number of bytes returned in the **`StringDescriptor`** array.

`Index`
Specifies the zero-based index of the string descriptor to be retrieved. The index values are defined by the device. If **`Index`** is set to zero the device returns a list of language ID's it supports.

`LanguageId`
Specifies the language ID of the string descriptor to be retrieved. This should be a language ID which is supported by the device. A list of supported language ID's is returned if **`Index`** is set to zero.

`Status`
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

A string descriptor contains a UNICODE string. The first two bytes describe the type and the length of the descriptor. Refer to the USB specification for detailed information.

According to the USB specification the maximum size of a string descriptor is 256 bytes.

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
GetDeviceDescriptor (page 48)
GetConfigurationDescriptor (page 49)
GetDescriptor (page 46)

SetDescriptor Method

This function can be used to set a descriptor in the USB device.

Definition

```
HRESULT
SetDescriptor(
     [in,out] SAFEARRAY(unsigned char)* Descriptor,
     [in] int Recipient,
     [in] int DescriptorType,
     [in] int DescriptorIndex,
     [in] int LanguageId,
     [out] int* Status
     );
```

Parameters

Descriptor
An array type variable that provides the descriptor to set.

Recipient
Specifies the recipient of the SET_DESCRIPTOR request. The values are defined by the enumeration type **USBIOCOM_REQUEST_RECIPIENT**.

DescriptorType
Specifies the type of the descriptor to set. The values are defined by the enumeration type **USBIOCOM_DESCRIPTOR_TYPE**. These numerical values are defined by the Universal Serial Bus Specification, Chapter 9 and additional device class specifications.

DescriptorIndex
Specifies the zero-based index of the descriptor to set. The meaning depends on the descriptor type.

LanguageId
Specifies the language ID in case of a string descriptor. For other descriptor types this value is not used and should be set to zero.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

Note that most USB devices do not implement the SET_DESCRIPTOR request.

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
GetDescriptor (page 46)
GetDeviceDescriptor (page 48)
GetConfigurationDescriptor (page 49)
GetStringDescriptor (page 51)
USBIOCOM_REQUEST_RECIPIENT (page 127)
USBIOCOM_DESCRIPTOR_TYPE (page 131)

AddInterface Method

This function adds an interface to a list maintained internally by the USBIO COM object instance.

Definition

```
HRESULT
AddInterface(
     [in] int InterfaceIndex,
     [in] int AlternateSettingIndex,
     [in] int MaximumTransferSize,
     [out] int* Status
     );
```

Parameters

`InterfaceIndex`
Identifies the interface to be added. Valid values are defined by the device. Refer to the Universal Serial Bus Specification, Chapter 9 for more information.

`AlternateSettingIndex`
Identifies the alternate setting of the interface that shall be activated when the device is configured. Valid values are defined by the device. Refer to the Universal Serial Bus Specification, Chapter 9 for more information.

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

The buffer size used for read and write requests cannot exceed the **`MaximumTransferSize`** value, see **StartReading**, **StartWriting**.

`Status`
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

AddInterface is used to prepare the configuration of the USB device which will be set by **SetConfiguration**. AddInterface has to be called at least once before the device can be configured. It can be called several times to prepare all required interfaces.

If a device supports more than one interface all of them have to be included in the configuration. Otherwise, the call to **SetConfiguration** fails. Consequently, a call to AddInterface is required for each interface.

The parameters passed to AddInterface are not checked immediately. If the parameters are not valid for the given device the subsequent call to **SetConfiguration** will fail.

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
DeleteInterfaces (page 57)
SetConfiguration (page 58)
UnconfigureDevice (page 61)
StartReading (page 80)
StartWriting (page 88)

DeleteInterfaces Method

This function deletes all interfaces from the list maintained internally by the USBIO COM object instance.

Definition

```
HRESULT
DeleteInterfaces();
```

Comments

All the interfaces previously added by means of **AddInterface** will be deleted. The internal interface list is empty after this call.

See Also

AddInterface (page 55)
SetConfiguration (page 58)
UnconfigureDevice (page 61)

SetConfiguration Method

This function configures the USB device.

Definition

```
HRESULT
SetConfiguration(
     [in] int ConfigurationIndex,
     [out] int* Status
     );
```

Parameters

`ConfigurationIndex`
Specifies the configuration to set. The value given here is the zero-based index of the configuration descriptor that is related to the configuration to be set. The index is used to query the associated configuration descriptor (GET_DESCRIPTOR request). The configuration value *bConfiguration* that is contained in the configuration descriptor is used for the subsequent SET_CONFIGURATION request.

For a single-configuration device the only valid value of **`ConfigurationIndex`** is zero.

`Status`
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

This function configures the interfaces previously stored by means of the **AddInterface** method. The parameters specified on the **AddInterface** calls will be used. **AddInterface** has to be called at least once before the device can be configured.

The USB device has to be configured to activate its endpoints. After that, read and write operations can be performed.

The following functions can only be used after the device has been configured: **ResetDevice**, **CyclePort**, **SetDevicePowerState**.

The device must be configured to use the remote wake up feature and to support the system suspend state.

This function can only be used after the device was opened, see **OpenDevice**. The device should be configured after the call to the **OpenDevice** function.

See Also

OpenDevice (page 33)
AddInterface (page 55)

DeleteInterfaces (page 57)
SetInterface (page 62)
UnconfigureDevice (page 61)
ResetDevice (page 73)
CyclePort (page 74)
SetDevicePowerState (page 72)

GetConfiguration Method

This function retrieves the configuration currently set in the USB device.

Definition

```
HRESULT
GetConfiguration(
     [out] unsigned char* ConfigurationValue,
     [out] int* Status
     );
```

Parameters

ConfigurationValue
A variable that receives the configuration value *bConfiguration* defined by the configuration descriptor that is related to the current configuration.

The value 0 is returned if the USB device is not configured.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
UnconfigureDevice (page 61)

UnconfigureDevice Method

This function unconfigures the USB device.

Definition

```
HRESULT
UnconfigureDevice(
     [out] int* Status
     );
```

Parameter

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

All pending read and write operations will be aborted. All endpoints will be unbound from USBIO COM object instances.

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)

SetInterface Method

This function sets a new alternate setting for an interface and changes the maximum transfer size.

Definition

```
HRESULT
SetInterface(
     [in] int InterfaceIndex,
     [in] int AlternateSettingIndex,
     [in] int MaximumTransferSize,
     [out] int* Status
     );
```

Parameters

`InterfaceIndex`
Identifies the interface to be modified. Valid values are defined by the device. Refer to the Universal Serial Bus Specification, Chapter 9 for more information.

`AlternateSettingIndex`
Identifies the alternate setting of the interface to be set. Valid values are defined by the device. Refer to the Universal Serial Bus Specification, Chapter 9 for more information.

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

The buffer size used for read and write requests cannot exceed the **`MaximumTransferSize`** value, see **StartReading**, **StartWriting**.

`Status`
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

All read and write operations have to be stopped before this function is called.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**.

See Also

OpenDevice (page 33)
AddInterface (page 55)
SetConfiguration (page 58)
UnconfigureDevice (page 61)

StartReading (page 80)
StartWriting (page 88)

GetInterface Method

This function retrieves the current alternate setting of an interface.

Definition

```
HRESULT
GetInterface(
     [out] unsigned char* AlternateSetting,
     [in] int InterfaceIndex,
     [out] int* Status
     );
```

Parameters

AlternateSetting
A variable that receives the current alternate setting. The values are defined by the device. Refer to the Universal Serial Bus Specification, Chapter 9 for more information.

InterfaceIndex
Identifies the interface. Valid values are defined by the device. Refer to the Universal Serial Bus Specification, Chapter 9 for more information.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
SetInterface (page 62)

ClassOrVendorInRequest Method

This function is used to generate a class or vendor specific device request (SETUP packet) with a data transfer phase in device to host (IN) direction.

Definition

```
HRESULT
ClassOrVendorInRequest(
     [in,out] SAFEARRAY(unsigned char)* Buffer,
     [in,out] int* ByteCount,
     [in] int Flags,
     [in] int Type,
     [in] int Recipient,
     [in] int RequestTypeReservedBits,
     [in] int Request,
     [in] int Value,
     [in] int Index,
     [out] int* Status
     );
```

Parameters

Buffer
An array type variable that receives the data transferred from the device in the data in phase. The caller has to provide this variable. The array will not be resized by the COM object. Consequently, the array has to provide enough storage for the data to be transferred.

ByteCount
A variable that specifies the number of bytes to be transferred from the device in the data phase. This value is copied to the bytes 7 and 8 of the SETUP request. The byte count must not exceed 4096 and must be less than or equal to the number of bytes provided in **Buffer**.

After the ClassOrVendorInRequest function succeeds the **ByteCount** variable will be set to the number of bytes returned in the **Buffer** array.

Flags
This field contains zero or the following value.

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

Type
Specifies the type of the device request. The values are defined by the enumeration type **USBIOCOM_REQUEST_TYPE**. A value of **USBIOCOM_RequestTypeClass**

generates a class-specific request and a value of **USBIOCOM_RequestTypeVendor** generates a vendor-specific request.

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

RequestTypeReservedBits
Specifies the reserved bits of the *bmRequestType* field of the SETUP packet. Normally, this field is set to zero.

Request
Specifies the value of the *bRequest* field of the SETUP packet. This is an 8 bit value.

Value
Specifies the value of the *wValue* field of the SETUP packet. This is a 16 bit value.

Index
Specifies the value of the *wIndex* field of the SETUP packet. This is a 16 bit value.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

The generated control request can be either a vendor-specific or a class-specific request. The request is sent to the default endpoint 0 (EP0) of the device.

Note that this function cannot be used to generate standard USB requests.

Refer to the Universal Serial Bus Specification, Chapter 9 for detailed information on the format of control requests.

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
ClassOrVendorOutRequest (page 67)
USBIOCOM_REQUEST_TYPE
USBIOCOM_REQUEST_RECIPIENT

ClassOrVendorOutRequest Method

This function is used to generate a class or vendor specific device request (SETUP packet) with a data transfer phase in host to device (OUT) direction.

Definition

```
HRESULT
ClassOrVendorOutRequest(
     [in,out] SAFEARRAY(unsigned char)* Buffer,
     [in] int Flags,
     [in] int Type,
     [in] int Recipient,
     [in] int RequestTypeReservedBits,
     [in] int Request,
     [in] int Value,
     [in] int Index,
     [out] int* Status
     );
```

Parameters

Buffer
An array type variable that provides the data to be sent in the data out phase. The size of the array defines the number of bytes to be transferred.

Flags
Should be set to zero.

Type
Specifies the type of the device request. The values are defined by the enumeration type **USBIOCOM_REQUEST_TYPE**. A value of **USBIOCOM_RequestTypeClass** generates a class-specific request and a value of **USBIOCOM_RequestTypeVendor** generates a vendor-specific request.

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

RequestTypeReservedBits
Specifies the reserved bits of the *bmRequestType* field of the SETUP packet. Normally, this field is set to zero.

Request
Specifies the value of the *bRequest* field of the SETUP packet. This is an 8 bit value.

Value
Specifies the value of the *wValue* field of the SETUP packet. This is a 16 bit value.

Index
Specifies the value of the *wIndex* field of the SETUP packet. This is a 16 bit value.

`Status`
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

The generated control request can be either a vendor-specific or a class-specific request. The request is sent to the default endpoint 0 (EP0) of the device.

Note that this function cannot be used to generate standard USB requests.

Refer to the Universal Serial Bus Specification, Chapter 9 for detailed information on the format of control requests.

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
ClassOrVendorInRequest (page 65)
USBIOCOM_REQUEST_TYPE
USBIOCOM_REQUEST_RECIPIENT

SetFeature Method

This function is used to generate a SET_FEATURE device request.

Definition

```
HRESULT
SetFeature(
     [in] int Recipient,
     [in] int FeatureSelector,
     [in] int Index,
     [out] int* Status
     );
```

Parameters

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

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

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

Refer to the Universal Serial Bus Specification, Chapter 9 for detailed information on the format of control requests.

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
ClearFeature (page 70)
USBIOCOM_REQUEST_RECIPIENT (page 127)

ClearFeature Method

This function is used to generate a CLEAR_FEATURE device request.

Definition

```
HRESULT
ClearFeature(
     [in] int Recipient,
     [in] int FeatureSelector,
     [in] int Index,
     [out] int* Status
     );
```

Parameters

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

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

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

Refer to the Universal Serial Bus Specification, Chapter 9 for detailed information on the format of control requests.

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
SetFeature (page 69)
USBIOCOM_REQUEST_RECIPIENT (page 127)

GetDevicePowerState Method

This function retrieves the current power state of the device.

Definition

```
HRESULT
GetDevicePowerState(
     [out] int* DevicePowerState,
     [out] int* Status
     );
```

Parameters

DevicePowerState
A variable that receives the current device power state. The values are defined by the enumeration type **USBIOCOM_DEVICE_POWER_STATE**. The meaning of the values is defined by the Power Management specification.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

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

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
SetDevicePowerState (page 72)
USBIOCOM_DEVICE_POWER_STATE (page 130)

SetDevicePowerState Method

This function sets a new device power state.

Definition

```
HRESULT
SetDevicePowerState(
     [in] int DevicePowerState,
     [out] int* Status
     );
```

Parameters

DevicePowerState
Specifies the device power state to be set. The values are defined by the enumeration type **USBIOCOM_DEVICE_POWER_STATE**. The meaning of the values is defined by the Power Management specification.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

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

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
GetDevicePowerState (page 71)
USBIOCOM_DEVICE_POWER_STATE (page 130)

ResetDevice Method

This function forces a USB reset at the hub port in which the device is plugged in.

Definition

```
HRESULT
ResetDevice(
     [out] int* Status
     );
```

Parameter

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

All pipes associated with the device will be unbound and all pending read and write operations will be cancelled. After ResetDevice is called the device is in the unconfigured state.

If the device changes its USB descriptor set during a USB Reset the **CyclePort** method should be used instead of this function.

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

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
CyclePort (page 74)

CyclePort Method

This function simulates a disconnect/connect cycle at the hub port in which the device is plugged in.

Definition

```
HRESULT
CyclePort(
     [out] int* Status
     );
```

Parameter

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

The CyclePort function is similar to **ResetDevice** except that from the software point of view a disconnect/connect cycle is simulated. This causes a re-enumeration of the device. The USBIO device driver is unloaded and then loaded again.

After an application called this function it should close all instances open for the current device by calling **CloseDevice** with each of them. Then an enumeration of USB devices must be initiated again by using **EnumerateDevices**.

The CyclePort function should be used instead of **ResetDevice** 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).

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

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**.

See Also

EnumerateDevices (page 31)
OpenDevice (page 33)
CloseDevice (page 35)
SetConfiguration (page 58)
ResetDevice (page 73)

GetStatus Method

This function is used to generate a GET_STATUS device request.

Definition

```
HRESULT
GetStatus(
     [out] int* StatusValue,
     [in] int Recipient,
     [in] int Index,
     [out] int* Status
     );
```

Parameters

StatusValue
A variable that receives the status code that is returned by the recipient in response to the get status request. This is a 16 bit value. The meaning of the value is defined by the device. Refer to the Universal Serial Bus Specification, Chapter 9 for more information.

Recipient
Specifies the recipient of the device request. The values are defined by the enumeration type **USBIOCOM_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, Chapter 9 for more information.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)
USBIOCOM_REQUEST_RECIPIENT (page 127)

GetCurrentFrameNumber Method

This function retrieves the current value of the frame number counter maintained by the USB host controller driver.

Definition

```
HRESULT
GetCurrentFrameNumber(
     [out] int* FrameNumber,
     [out] int* Status
     );
```

Parameters

FrameNumber
A variable that receives the current frame number. This is a 32 bit value. The 11 least significant bits correspond to the current frame number on the bus.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)

ErrorText Method

This function translates a private status code returned by the USBIO COM interface to an error text string.

Definition

```
HRESULT
ErrorText(
     [in] int Status,
     [out,retval] BSTR* Text
     );
```

Parameters

Status
Specifies the status code to be translated.

Text
The method returns the error text string that corresponds to the given status code.

Comments

The function translates private status codes only that are defined by the USBIO device driver and the USBIO COM object. Those codes are listed in section 6.4.

Note that USBIO private status codes range from 0xE0000000 to 0xE000FFFF.

Status codes returned by Windows API functions cannot be translated by this function.

Bind Method

This function establishes a binding between the USBIO COM object instance and an endpoint.

Definition

```
HRESULT
Bind(
     [in] unsigned char EndpointAddress,
     [out] int* Status
     );
```

Parameters

`EndpointAddress`
Specifies the address of the endpoint to bind the object instance 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

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

`Status`
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

A USBIO COM object instance can be bound to one endpoint only. Consequently, one object instance has to be created for each active endpoint.

Only active endpoints from the current configuration can be bound.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**.

After the USBIO COM object instance has been successfully bound to an endpoint data transfer operations can be initiated for this endpoint, see **StartReading** and **StartWriting**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
Unbind (page 79)
StartReading (page 80)
StartWriting (page 88)

Unbind Method

This function deletes the binding between the USBIO COM object instance and an endpoint.

Definition

```
HRESULT
Unbind(
     [out] int* Status
     );
```

Parameter

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

All read and write operations related to the bound endpoint will be stopped.

After a call to Unbind the USBIO COM object instance can be bound to an endpoint again by another call to **Bind**.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
Bind (page 78)

StartReading Method

This function starts an internal worker-thread that performs a continuous data transfer from the endpoint that is attached to the USBIO COM object instance.

Definition

```
HRESULT
StartReading(
     [in] int SizeOfBuffer_IsoFramesInBuffer,
     [in] int NumberOfBuffers,
     [in] int MaxErrorCount,
     [out] int* Status
     );
```

Parameters

`SizeOfBuffer_IsoFramesInBuffer`

The meaning of this parameter depends on the type of the endpoint.

Bulk or Interrupt Endpoint

If the endpoint is of type bulk or interrupt this parameter specifies the size, in bytes, of the read buffers used internally by the worker-thread. The read buffer size cannot exceed the maximum transfer size configured for the endpoint. The maximum transfer size is specified on a call to **AddInterface** or **SetInterface**.

The buffer size should be a multiple of the endpoint's FIFO size. Otherwise, buffer overrun errors can occur.

Isochronous Endpoint

If the endpoint is of type isochronous this parameter specifies the number of isochronous frames in a buffer. The size of a read buffer, in bytes, is the product of **`SizeOfBuffer_IsoFramesInBuffer`** and the FIFO size of the endpoint. The read buffer size cannot exceed the maximum transfer size configured for the endpoint, otherwise the error code **`USBIO_ERR_INVALID_PARAMETER`** will be returned. The maximum transfer size is specified in a call to **AddInterface** or **SetInterface**.

The value of **`SizeOfBuffer_IsoFramesInBuffer`** should be in the range 16..64. See the comments below for further information.

`NumberOfBuffers`

This parameter specifies the number of buffers to be allocated internally by the worker-thread. The value of **`NumberOfBuffers`** should be in the range 5..50. See the comments below for further information.

`MaxErrorCount`

This parameter specifies the maximum value of an error counter that is maintained internally by the worker-thread. Each time a data transfer is completed with an error the error counter will be incremented. If the counter reaches the **`MaxErrorCount`** value the worker-thread will stop. The error counter is reset to zero on each successful data transfer

operation. This way, the worker-thread will be stopped automatically if a device constantly causes errors. Thus, an end-less loop causing a high CPU load will be avoided in this situation.

`Status`
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

The function allocates an internal pool of buffers to be used for the data transfer from the device's endpoint to the host. The size of each buffer and the number of buffers to allocate is specified by means of the parameters **`SizeOfBuffer_IsoFramesInBuffer`** and **`NumberOfBuffers`**.

In case of a bulk or interrupt endpoint the buffer size and the total amount of memory to allocate is calculated as follows.

BufferSize = SizeOfBuffer_IsoFramesInBuffer
*TotalSize = NumberOfBuffers * BufferSize*

In case of an isochronous endpoint the buffer size and the total amount of memory to allocate is calculated as follows.

*BufferSize = SizeOfBuffer_IsoFramesInBuffer * EndpointFifoSize*
*TotalSize = NumberOfBuffers * BufferSize*

In case of an isochronous transfer the parameter **`SizeOfBuffer_IsoFramesInBuffer`** should be choosen carefully. Because it specifies the number of isochronous data frames to be placed in a buffer the parameter has an impact on the delay that will occur in the data path. In full-speed mode, one isochronous data frame per millisecond is received by the host, but the application will not receive the buffer until it is completely filled with isochronous frames. Thus, large buffers cause a delay.

On the other hand, a small number of isochronous data frames per buffer will cause a high CPU load because the interval of buffer completion events is very short. A PC is not able to handle events efficiently at an interval of one or two milliseconds.

The isochronous data transmission is more stable when larger buffers are used because this is more tolerant regarding thread latencies. If the buffers are too small and the CPU is busy with other tasks transmission errors like **`USBIO_ERR_BAD_START_FRAME`** can occur.

An application has to find a compromise for the size of isochronous buffers. In full-speed mode, a value of 16 to 64 isochronous data frames per buffer is recommended. However, the following limitations should be considered.

On Windows 98 and Windows 98 SE the number of isochronous data frames per buffer must be less than or equal 32 (see also problems.txt in the USBIO package).

Because of a limitation of the system-provided USB host controller driver the product of **`SizeOfBuffer_IsoFramesInBuffer`** and **`NumberOfBuffers`** cannot exceed

1024. If this condition is not met the error code **USBIO_ERR_BAD_START_FRAME** is returned by the function **ReadIsoData**.

After the buffer pool was successfully allocated the function starts an internal worker-thread that handles the data transfer by means of asynchronous (overlapped) read requests. The thread implements a circulation of buffers. This way, it ensures that a continuous data transfer from the device is possible.

After the worker-thread is started the USB host sends IN tokens to the endpoint that is attached to the USBIO COM object instance.

When a data buffer is received from the device the worker-thread issues a **ReadComplete** event. The application should implement a handler for this event and when it is received read the data from the USBIO COM object instance by calling **ReadData** or **ReadIsoData** depending on the type of the endpoint.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**. Furthermore, the object instance must have been bound to an endpoint, see **Bind**.

Important: This function must not be called from the context of the following event handler functions: **ReadComplete**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
Bind (page 78)
ReadData (page 83)
ReadIsoData (page 85)
ReadComplete (page 119)
StopReading (page 87)

ReadData Method

This function is used to read the data received from a bulk or interrupt endpoint.

Definition

```
HRESULT
ReadData(
     [in,out] SAFEARRAY(unsigned char)* Buffer,
     [out] int* ByteCount,
     [out] int* Status
     );
```

Parameters

`Buffer`
An array type variable that receives the data bytes. The caller has to provide this variable. The array will not be resized by the COM object. Consequently, the array has to provide enough storage for the data.

`ByteCount`
A variable that specifies the number of bytes to read. This number has to be greater than zero and less than or equal than the number of bytes provided in **`Buffer`**.

After the ReadData function succeeds the **`ByteCount`** variable will be set to the number of bytes returned in the **`Buffer`** array.

`Status`
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

The function reads the received data from the intermediate storage that is maintained internally by the worker-thread. It does never block and wait for data to be received from the device. If no data is available the function returns immediately with a status code of **`USBIO_ERR_NO_DATA`**.

ReadData should be called when the application receives the event **ReadComplete** indicating that data is available. There is no need to call this function periodically (polling). This would cause an unnecessary CPU load. The USBIO COM object instance issues a **ReadComplete** event each time data becomes available.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**. Furthermore, the object instance must have been bound to an endpoint, see **Bind**, and the internal worker-thread must have been started, see **StartReading**.

This function should not be used after a call to **StopReading**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
Bind (page 78)
StartReading (page 80)
StopReading (page 87)
ReadComplete (page 119)
ReadIsoData (page 85)
ResetPipe (page 101)

ReadIsoData Method

This function is used to read the data received from an isochronous endpoint.

Definition

```
HRESULT
ReadIsoData(
     [in,out] SAFEARRAY(unsigned char)* Buffer,
     [out] int* ByteCount,
     [in,out] SAFEARRAY(int)* SubBufferLength_ErrorCode,
     [out] int* Status
     );
```

Parameters

`Buffer`
An array type variable that receives the data bytes. The caller has to provide this variable. The array will not be resized by the COM object. Consequently, the array has to provide enough storage for the data.

`ByteCount`
A variable that specifies the number of bytes to read. This number has to be less than or equal than the number of bytes provided in **`Buffer`**.

After the ReadIsoData function succeeds the **`ByteCount`** variable will be set to the number of bytes returned in the **`Buffer`** array.

`SubBufferLength_ErrorCode`
An array type variable that receives additional information on the isochronous frames received. The caller has to provide this variable. The array will not be resized by the COM object. Consequently, the array has to provide enough storage.

The array has to provide one integer per isochronous frame in the buffer. Therefore, the dimension of the array has to be equal or greater than the parameter **`SizeOfBuffer_IsoFramesInBuffer`** passed to **StartReading**.

`Status`
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Note that even if success is returned the data transfer of one or more isochronous frames could have been failed. In order to retrieve the status of single frames the contents of **`SubBufferLength_ErrorCode`** has to be examined.

Comments

The function reads the received data from the intermediate storage that is maintained internally by the worker-thread. It does never block and wait for data to be received from

the device. If no data is available the function returns immediately with a status code of **USBIO_ERR_NO_DATA**.

ReadIsoData should be called when the application receives the event **ReadComplete** indicating that data is available. There is no need to call this function periodically (polling). This would cause an unnecessary CPU load. The USBIO COM object instance issues a **ReadComplete** event each time data becomes available.

The function places the received isochronous frames contiguous in the array provided at **Buffer**. There will be no gaps between the frames. Note that the length of the frames varies. An application should examine the values returned in the **SubBufferLength_ErrorCode** array to learn about the length of each single frame.

The function returns an integer value per isochronous frame in the array provided at **SubBufferLength_ErrorCode**. The integer value provides status information on the corresponding frame. If the value is greater than or equal to zero the frame was received successfully. In this case, the integer value is the length of the isochronous data frame, in bytes. Note that the length of the frame can be zero. This is not an error.
If the value is negative the corresponding frame was received with error. In this case, the integer value is an error code.

Frames that were received with error will not be copied to the **Buffer** array.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**. Furthermore, the object instance must have been bound to an endpoint, see **Bind**, and the internal worker-thread must have been started, see **StartReading**.

This function should not be used after a call to **StopReading**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
Bind (page 78)
StartReading (page 80)
StopReading (page 87)
ReadComplete (page 119)
ReadData (page 83)

StopReading Method

This function stops the internal worker-thread that performs a continuous data transfer from the endpoint that is attached to the USBIO COM object instance.

Definition

```
HRESULT
StopReading();
```

Comments

After the worker-thread is stopped no further IN tokens will be send to the endpoint that is attached to the USBIO COM object instance.

Important: This function must not be called from the context of the following event handler functions: **ReadComplete**.

See Also

StartReading (page 80)
ReadData (page 83)
ReadIsoData (page 85)

StartWriting Method

This function starts an internal worker-thread that performs a continuous data transfer to the endpoint that is attached to the USBIO COM object instance.

Definition

```
HRESULT
StartWriting(
     [in] int SizeOfBuffer_IsoFramesInBuffer,
     [in] int NumberOfBuffers,
     [in] int MaxErrorCount,
     [in] BOOL WriteStatus,
     [out] int* Status
     );
```

Parameters

`SizeOfBuffer_IsoFramesInBuffer`

The meaning of this parameter depends on the type of the endpoint.

Bulk or Interrupt Endpoint

If the endpoint is of type bulk or interrupt this parameter specifies the size, in bytes, of the write buffers used internally by the worker-thread. The write buffer size cannot exceed the maximum transfer size configured for the endpoint. The maximum transfer size is specified on a call to **AddInterface** or **SetInterface**.

Isochronous Endpoint

If the endpoint is of type isochronous this parameter specifies the number of isochronous frames in a buffer. The size of a write buffer, in bytes, is the product of **`SizeOfBuffer_IsoFramesInBuffer`** and the FIFO size of the endpoint. The write buffer size cannot exceed the maximum transfer size configured for the endpoint, otherwise the error code **`USBIO_ERR_INVALID_PARAMETER`** will be returned. The maximum transfer size is specified in a call to **AddInterface** or **SetInterface**.

The value of **`SizeOfBuffer_IsoFramesInBuffer`** should be in the range 16..64. See the comments below for further information.

`NumberOfBuffers`

This parameter specifies the number of buffers to be allocated internally by the worker-thread. The value of **`NumberOfBuffers`** should be in the range 5..50. See the comments below for further information.

`MaxErrorCount`

This parameter specifies the maximum value of an error counter that is maintained internally by the worker-thread. Each time a data transfer is completed with an error the error counter will be incremented. If the counter reaches the **`MaxErrorCount`** value the worker-thread will stop. The error counter is reset to zero on each successful data transfer operation. This way, the worker-thread will be stopped automatically if a device

constantly causes errors. Thus, an end-less loop causing a high CPU load will be avoided in this situation.

`WriteStatus`
This parameter indicates whether the application wants to be informed on the completion status of each write operation.

If **`WriteStatus`** is set to TRUE the application has to implement a handler for the event **WriteStatusAvailable**. This event will be issued by the USBIO COM object instance after a write operation has been completed and the completion status of the operation is available. The application retrieves the status code by calling **GetWriteStatus** or **GetIsoWriteStatus**, depending on the type of the endpoint.
Note that if TRUE is specified the application is required to process the status codes by using **GetWriteStatus** or **GetIsoWriteStatus** to free the buffers and make them available for further write operations.

If **`WriteStatus`** is set to FALSE the application does not need to implement a handler for the event **WriteStatusAvailable**. The application will not be informed on the completion status of write operations. It does not call **GetWriteStatus** or **GetIsoWriteStatus**.
The buffers will be freed automatically after the write operation has been completed, regardless of the completion status. Thus, the buffers become available for further write operations.

`Status`
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

The function allocates an internal pool of buffers to be used for the data transfer from the host to the device's endpoint. The size of each buffer and the number of buffers to allocate is specified by means of the parameters **`SizeOfBuffer_IsoFramesInBuffer`** and **`NumberOfBuffers`**.

In case of a bulk or interrupt endpoint the buffer size and the total amount of memory to allocate is calculated as follows.

BufferSize = SizeOfBuffer_IsoFramesInBuffer
*TotalSize = NumberOfBuffers * BufferSize*

In case of an isochronous endpoint the buffer size and the total amount of memory to allocate is calculated as follows.

*BufferSize = SizeOfBuffer_IsoFramesInBuffer * EndpointFifoSize*
*TotalSize = NumberOfBuffers * BufferSize*

In case of an isochronous transfer the parameter **`SizeOfBuffer_IsoFramesInBuffer`** should be choosen carefully. Because it specifies the number of isochronous data frames placed in a buffer the parameter has an impact on the delay that will occur in the data path. In full-speed mode, one isochronous data frame per millisecond is sent by the host, but the application has to provide buffers that are completely filled with isochronous frames. Thus, large buffers cause a delay.

On the other hand, a small number of isochronous data frames per buffer will cause a high CPU load because the interval of buffer completion events is very short. A PC is not able to handle events efficiently at an interval of one or two milliseconds.

The isochronous data transmission is more stable when larger buffers are used because this is more tolerant regarding thread latencies. If the buffers are too small and the CPU is busy with other tasks transmission errors like **USBIO_ERR_BAD_START_FRAME** can occur.

An application has to find a compromise for the size of isochronous buffers. For full-speed mode a value of 16 to 64 isochronous data frames per buffer is recommended. However, the following limitations should be considered.

On Windows 98 and Windows 98 SE the number of isochronous data frames per buffer must be less than or equal 32 (see also problems.txt in the USBIO package).

Because of a limitation of the system-provided USB host controller driver the product of **SizeOfBuffer_IsoFramesInBuffer** and **NumberOfBuffers** cannot exceed 1024. If this condition is not met the error code **USBIO_ERR_BAD_START_FRAME** is returned by the function **ReadIsoData**.

After the buffer pool was successfully allocated the function starts an internal worker-thread that handles the data transfer by means of asynchronous (overlapped) write requests. The thread implements a circulation of buffers. This way, it ensures that a continuous data transfer to the device is possible.

After the worker-thread is started it waits until the application calls **WriteData** or **WriteIsoData**, depending on the type of the endpoint. Then the thread initiates write operations. This causes the USB host to start sending OUT tokens to the endpoint that is attached to the USBIO COM object instance.

An application provides data to be transferred to the device by calling **WriteData** or **WriteIsoData** in a loop until the error code **USBIO_ERR_NO_BUFFER** is returned. This indicates that the internal buffer space is exhausted. The application stops writing data.

When a write operation is completed and a free buffer becomes available the worker-thread issues a **WriteComplete** event. The application should implement a handler for this event and when it is received continue to write data by calling **WriteData** or **WriteIsoData** in a loop again.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**. Furthermore, the object instance must have been bound to an endpoint, see **Bind**.

Important: This function must not be called from the context of the following event handler functions: **WriteComplete**, **WriteStatusAvailable**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
Bind (page 78)
WriteData (page 92)
GetWriteStatus (page 94)

WriteIsoData (page 96)
GetIsoWriteStatus (page 98)
WriteComplete (page 120)
StopWriting (page 100)

WriteData Method

This function is used to write data to be send to a bulk or interrupt endpoint.

Definition

```
HRESULT
WriteData(
     [in,out] SAFEARRAY(unsigned char)* Buffer,
     [in] int UserId,
     [out] int* Status
     );
```

Parameters

Buffer
An array type variable that provides the data to be written. The size of the array determines the number of bytes to write. Note that writing of zero bytes is possible. This will result in a zero packet sent to the endpoint.

The number of bytes in **Buffer** cannot exceed the size of a write buffer specified in **SizeOfBuffer_IsoFramesInBuffer** at the call to **StartWriting**.

UserId
This parameter specifies an application-defined ID that can be used to identify the write buffer in a later call to **GetWriteStatus**.

If an application does not implement processing of write completion status codes this parameter should be set to zero.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Note that the value returned is not the final completion status of the write operation. The write operation can fail even if the returned status of this function indicates success. In order to retrieve the completion status of the write operation **GetWriteStatus** has to be used.

Comments

The function copies the provided data to the intermediate storage that is maintained internally by the worker-thread. It does never block and wait until data is sent to the device. If no intermediate buffer space is available the function returns immediately with a status code of **USBIO_ERR_NO_BUFFER**.

WriteData should be called again when the application receives the event **WriteComplete** indicating that buffer space is available. There is no need to call this function periodically (polling). This would cause an unnecessary CPU load. The USBIO COM object instance issues a **WriteComplete** event each time a buffer becomes available.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**. Furthermore, the object instance must have been bound to an endpoint, see **Bind**, and the internal worker-thread must have been started, see **StartWriting**.

This function should not be used after a call to **StopWriting**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
Bind (page 78)
StartWriting (page 88)
StopWriting (page 100)
GetWriteStatus (page 94)
WriteComplete (page 120)
WriteIsoData (page 96)
ResetPipe (page 101)

GetWriteStatus Method

This function is used to retrieve the completion status of a write operation to a bulk or interrupt endpoint.

Definition

```
HRESULT
GetWriteStatus(
     [out] int* UserId,
     [out] int* Status
     );
```

Parameters

`UserId`
A variable that receives the application-defined ID that was passed to **WriteData**. An application can use this ID to identify the write buffer.

`Status`
A variable that receives the final completion status of the write operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

This function can be used only if the parameter **`WriteStatus`** was set to TRUE at the call to **StartWriting**. See **StartWriting** for more information.

GetWriteStatus should be called when the application receives the event **WriteStatusAvailable** indicating that a write operation has been completed. There is no need to call this function periodically (polling). This would cause an unnecessary CPU load. The USBIO COM object instance issues a **WriteStatusAvailable** event each time a write operation gets completed.

After the call to GetWriteStatus returns the internal write buffer is marked as free and the event **WriteComplete** will be issued by the USBIO COM object instance.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**. Furthermore, the object instance must have been bound to an endpoint, see **Bind**, and the internal worker-thread must have been started, see **StartWriting**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
Bind (page 78)
StartWriting (page 88)

StopWriting (page 100)
WriteData (page 92)
WriteStatusAvailable (page 121)
WriteComplete (page 120)

WriteIsoData Method

This function is used to write data to be send to an isochronous endpoint.

Definition

```
HRESULT
WriteIsoData(
     [in,out] SAFEARRAY(unsigned char)* Buffer,
     [in,out] SAFEARRAY(int)* SubBufferLength,
     [in] int UserId,
     [out] int* Status
     );
```

Parameters

`Buffer`
An array type variable that provides the data to be written. The size of the array must be equal to the sum of all values passed in the array **`SubBufferLength`**.

`SubBufferLength`
An array type variable that specifies the length of each isochronous data frame to be transmitted. The caller has to provide this variable.

The array contains one integer per isochronous frame in the buffer. Therefore, the dimension of the array has to be equal to the parameter **`SizeOfBuffer_IsoFramesInBuffer`** passed to **StartWriting**. The integer value specifies the length of the corresponding isochronous data frame. See also the comments below.

`UserId`
This parameter specifies an application-defined ID that can be used to identify the write buffer in a later call to **GetIsoWriteStatus**.

If an application does not implement processing of write completion status codes this parameter should be set to zero.

`Status`
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Note that the value returned is not the final completion status of the write operation. The write operation can fail even if the returned status of this function indicates success. In order to retrieve the completion status of the write operation **GetIsoWriteStatus** has to be used.

Comments

The function copies the provided data to the intermediate storage that is maintained internally by the worker-thread. It does never block and wait until data is sent to the

device. If no intermediate buffer space is available the function returns immediately with a status code of **USBIO_ERR_NO_BUFFER**.

WriteIsoData should be called again when the application receives the event **WriteComplete** indicating that buffer space is available. There is no need to call this function periodically (polling). This would cause an unnecessary CPU load. The USBIO COM object instance issues a **WriteComplete** event each time a buffer becomes available.

The function expects the isochronous frames to be transmitted contiguous in the array provided at **Buffer**. There must be no gaps between the frames. Note that the length of the frames can vary. An application has to specify the length of each frame in the **SubBufferLength** array.

The array provided at **SubBufferLength** contains an integer value per isochronous data frame. This value specifies the length of the corresponding frame, in bytes. The length of an isochronous data frame can be zero. But, at least one frame in the buffer must have a length greater than zero. In other words, the total number of bytes in a buffer must not be zero.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**. Furthermore, the object instance must have been bound to an endpoint, see **Bind**, and the internal worker-thread must have been started, see **StartWriting**.

This function should not be used after a call to **StopWriting**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
Bind (page 78)
StartWriting (page 88)
StopWriting (page 100)
GetIsoWriteStatus (page 98)
WriteComplete (page 120)
WriteData (page 92)

GetIsoWriteStatus Method

This function is used to retrieve the completion status of a write operation to an isochronous endpoint.

Definition

```
HRESULT
GetIsoWriteStatus(
     [out] int* UserId,
     [in,out] SAFEARRAY(int)* StatusArray,
     [out] int* FrameCount,
     [out] int* Status
     );
```

Parameters

`UserId`
A variable that receives the application-defined ID that was passed to **WriteIsoData**. An application can use this ID to identify the write buffer.

`StatusArray`
An array type variable that receives status information on the isochronous frames transmitted. The caller has to provide this variable. The array will not be resized by the COM object. Consequently, the array has to provide enough storage.

The array has to provide one integer per isochronous frame in the buffer. Therefore, the dimension of the array has to be equal or greater than the parameter **`SizeOfBuffer_IsoFramesInBuffer`** passed to **StartWriting**.

`FrameCount`
A variable that will be set to the number of valid fields returned in **`StatusArray`**. This corresponds to the number of frames passed to **WriteIsoData**.

`Status`
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Note that the value returned is not the completion status of the isochronous write operation. To learn about the write completion status the values returned in **`StatusArray`** have to be examined.

Comments

This function can be used only if the parameter **`WriteStatus`** was set to TRUE at the call to **StartWriting**. See **StartWriting** for more information.

GetIsoWriteStatus should be called when the application receives the event **WriteStatusAvailable** indicating that a write operation has been completed. There is no need to call this function periodically (polling). This would cause an unnecessary CPU

load. The USBIO COM object instance issues a **WriteStatusAvailable** event each time a write operation gets completed.

After the call to GetIsoWriteStatus returns the internal write buffer is marked as free and the event **WriteComplete** will be issued by the USBIO COM object instance.

In the caller-provided **`StatusArray`** GetIsoWriteStatus returns an integer value per isochronous frame transmitted. The integer value is the completion status of the transmission of the corresponding frame. If the value is zero the frame was transmitted successfully. If an error occurred the value is one of the USBIO status codes listed in section 6.4.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**. Furthermore, the object instance must have been bound to an endpoint, see **Bind**, and the internal worker-thread must have been started, see **StartWriting**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
Bind (page 78)
StartWriting (page 88)
StopWriting (page 100)
WriteIsoData (page 96)
WriteStatusAvailable (page 121)
WriteComplete (page 120)

StopWriting Method

This function stops the internal worker-thread that performs a continuous data transfer to the endpoint that is attached to the USBIO COM object instance.

Definition

```
HRESULT
StopWriting();
```

Comments

After the worker-thread is stopped no further OUT tokens will be send to the endpoint that is attached to the USBIO COM object instance.

Note that the data that is maintained internally by the worker-thread but not yet sent to the device will be discarded by this call.

Important: This function must not be called from the context of the following event handler functions: **WriteComplete**, **WriteStatusAvailable**.

See Also

StartWriting (page 88)
WriteData (page 92)
WriteIsoData (page 96)

ResetPipe Method

This function clears an error condition on an endpoint.

Definition

```
HRESULT
ResetPipe(
     [out] int* Status
     );
```

Parameter

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

If an error occurs while transferring data from or to a bulk, interrupt, or isochronous endpoint the associated data pipe will be halted by the USB driver. No further data transfers on the pipe are possible. To clear this error condition and continue transferring data ResetPipe has to be called in this situation.

It is recommended to call this function before a read or write transfer is started by means of **StartReading** or **StartWriting**.

In case of a bulk or interrupt endpoint a call to this function causes a CLEAR_FEATURE(ENDPOINT_STALL) request to be send to the device.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**. Furthermore, the object instance must have been bound to an endpoint, see **Bind**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
Bind (page 78)
StartReading (page 80)
StartWriting (page 88)

AbortPipe Method

This function aborts all read and write operations currently in progress in the USBIO COM object instance.

Definition

```
HRESULT
AbortPipe(
     [out] int* Status
     );
```

Parameter

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

All outstanding read or write requests on the attached pipe are aborted and returned with an error status of **USBIO_ERR_CANCELED**. This will interrupt data transfers currently in progress.

This function should not be used on isochronous pipes. This can cause a system crash when the system-provided Open Host Controller driver is used. See also *Problems.txt* in the USBIO package.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**. Furthermore, the object instance must have been bound to an endpoint, see **Bind**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
Bind (page 78)
StartReading (page 80)
StartWriting (page 88)

ShortTransferOK Property

Reading this property returns TRUE if a data transfer shorter than the endpoint's FIFO size does not cause an error, FALSE otherwise. See below for a detailed description.

Definition

```
HRESULT
ShortTransferOK(
     [out,retval] BOOL* ShortTransfer
     );
```

Parameter

ShortTransfer
A variable that will be set to TRUE or FALSE.

ShortTransferOK Property

Setting this property to TRUE allows data transfers shorter than the endpoint's FIFO size for the endpoint that is bound to the USBIO COM object instance. Setting this property to FALSE causes short transfers to fail with an error status.

Definition

```
HRESULT
ShortTransferOK(
     [in] BOOL ShortTransfer
     );
```

Parameter

ShortTransfer
New value to be set, either TRUE or FALSE.

Comments

This parameter has an effect only for read operations from bulk or interrupt pipes.

The default setting of the **ShortTransferOK** property is stored in the registry during USBIO driver installation. The default value can be changed in the INF file or in the registry. The configuration set by means of this property is 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.

This property can only be read or written after the device was opened, see **OpenDevice**. Furthermore, the object instance must have been bound to an endpoint, see **Bind**.

See Also

OpenDevice (page 33)
Bind (page 78)

EndpointFifoSize Property

This read-only property returns the endpoint's FIFO size of the endpoint that is bound to the USBIO COM object instance.

Definition

```
HRESULT
EndpointFifoSize(
     [out,retval] int* pVal
     );
```

Parameter

pVal
A variable that receives the FIFO size, in bytes.

Comments

This property can only be read after the device was opened, see **OpenDevice**. Furthermore, the object instance must have been bound to an endpoint, see **Bind**.

See Also

OpenDevice (page 33)
Bind (page 78)

EnablePnPNotification Method

This function enables Plug and Play notification events.

Definition

```
HRESULT
EnablePnPNotification(
     [in] BSTR Guid,
     [out] int* Status
     );
```

Parameters

Guid
A string representation of a GUID that identifies the driver interface exported by the USBIO driver. This can be the default USBIO GUID {325ddf96-938c-11d3-9e34-0080c82727f4} that is defined in *usbio_i.h*. However, it is strongly recommended that a private GUID is used which has been generated by a tool like *guidgen.exe*. This is important in order to differentiate between various customizations of the USBIO product.

For the USBIO device driver the private GUID has to be defined in the setup information file *usbio.inf*. See the USBIO Reference Manual and the *usbio.inf* file for details.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

After a successful call to EnablePnPNotification the USBIO COM object will issue a **PnPAddNotification** event if a USB device is connected to the system and a **PnPRemoveNotification** event if a USB device is disconnected from the system.

If an application supports various types of devices and an individual GUID is defined for each of them EnablePnPNotification should be called for each GUID separately. This way, the application will receive Plug and Play notifications for every device it supports.

See Also

DisablePnPNotification (page 107)
PnPAddNotification (page 122)
PnPRemoveNotification (page 123)

DisablePnPNotification Method

This function disables Plug and Play notification events.

Definition

```
HRESULT
DisablePnPNotification(
     [in] BSTR Guid,
     [out] int* Status
     );
```

Parameters

Guid
A string representation of a GUID that identifies the driver interface. This is the same GUID that was passed to **EnablePnPNotification**.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

To disable Plug and Play notfication events this function has to be called for each GUID passed to a successful call of **EnablePnPNotification**.

See Also

EnablePnPNotification (page 106)
PnPAddNotification (page 122)
PnPRemoveNotification (page 123)

GetBandwidthInfo Method

This function returns information on the current USB bandwidth consumption.

Definition

```
HRESULT
GetBandwidthInfo(
     [out] int* TotalBandwidth,
     [out] int* ConsumedBandwidth,
     [out] int* Status
     );
```

Parameters

TotalBandwidth
A variable that will be set to 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
A variable that will be set to the mean bandwidth that is already in use, in kilobits per second.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

This function 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.

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)

IsOperatingAtHighSpeed Property

This read-only property returns TRUE if the USB 2.0 device is operating at high-speed (480 Mbit/s), FALSE otherwise.

Definition

```
HRESULT
IsOperatingAtHighSpeed(
     [out,retval] BOOL* HighSpeed
     );
```

Parameter

HighSpeed
A variable that will be set to TRUE or FALSE.

Comments

If this property 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 property does not indicate whether a device is capable of high-speed operation, but rather whether it is in fact operating at high-speed.

This property can only be read after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)

SetupPipeStatistics Method

This function enables or disables a statistical analysis of the data transfer on a pipe.

Definition

```
HRESULT
SetupPipeStatistics(
     [in] int AveragingInterval,
     [out] int* Status
     );
```

Parameters

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 **SetupPipeStatistics** function will fail. 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).

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

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 the description of the **QueryPipeStatistics** function for more information on pipe statistics.

Note that the statistical data is maintained separately for each pipe or endpoint, respectively. The **SetupPipeStatistics** function has an effect on that pipe only that is bound to the USBIO COM object instance.

If an endpoint is unbound from the USBIO COM object instance by means of the **Unbind** function or by deleting the instance then the average data rate computation will be disabled. It has to be enabled and configured when the endpoint is reused. In other words,

if the data rate computation is needed by an application then the **`SetupPipeStatistics`** function should be called immediately after the endpoint is bound by means of the **Bind** function.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**. Furthermore, the object instance must have been bound to an endpoint, see **Bind**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
Bind (page 78)
Unbind (page 79)
QueryPipeStatistics (page 112)

QueryPipeStatistics Method

This function retrieves statistical data related to the endpoint that is bound to the USBIO COM object instance.

Definition

```
HRESULT
QueryPipeStatistics(
     [out] int* ActualAveragingInterval,
     [out] int* AverageRate,
     [out] int* BytesTransferred_L,
     [out] int* BytesTransferred_H,
     [out] int* RequestsSucceeded,
     [out] int* RequestsFailed,
     [in] int Flags,
     [out] int* Status
     );
```

Parameters

ActualAveragingInterval
A variable that will be set to 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 **SetupPipeStatistics** function. 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** variable is always set to zero in this case.

AverageRate
A variable that will be set to the current average data rate of the pipe, in bytes per second. The average data rate will be continuously calculated if **ActualAveragingInterval** is not null. If **ActualAveragingInterval** is null then the data rate computation is disabled and **AverageRate** 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 **SetupPipeStatistics** function.

BytesTransferred_L
A variable that will be set to 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
A variable that will be set to 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`

A variable that will be set to 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 endpoint 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 endpoint 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`

A variable that will be set to 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 endpoint 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 endpoint 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.

`Flags`

Specifies options that modify the behaviour of the **`QueryPipeStatistics`** function, encoded as bit flags. The **`Flags`** value is set to zero or any combination (bit-wise or) of the following values.

USBIOCOM_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 and returned by this function. The BytesTransferred counter is an unsigned 64 bit integer. It counts the total number of bytes transferred on a pipe, modulo 2^64.

USBIOCOM_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 and returned by this function. 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.

USBIOCOM_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 and returned by this function. 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.

USBIOCOM_QPS_FLAG_RESET_ALL_COUNTERS

This value combines the three flags described above. If

USBIOCOM_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 and returned by this function.

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

The USBIO device driver internally maintains some statistical data per endpoint. 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.

The USBIO device driver is also 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 **SetupPipeStatistics** function before it is available to an application. Thus, before an application starts to (periodically) query the value of **AverageRate** it has to enable the continuous computation of this value by a call to **SetupPipeStatistics**. The other statistical counters returned by this function will be updated by default and do not need to be enabled explicitly.

Note that the statistical data is maintained separately for each pipe or endpoint, respectively. The **QueryPipeStatistics** function retrieves the actual statistics of that pipe that is bound to the USBIO COM object instance.

This function can only be used after the device was opened and configured, see **OpenDevice** and **SetConfiguration**. Furthermore, the object instance must have been bound to an endpoint, see **Bind**.

See Also

OpenDevice (page 33)
SetConfiguration (page 58)
Bind (page 78)
SetupPipeStatistics (page 110)
USBIOCOM_QUERY_PIPE_STATISTICS_FLAGS (page 132)

AcquireDevice Method

This function acquires a device for exclusive use.

Definition

```
HRESULT
AcquireDevice(
     [out] int* Status
     );
```

Parameter

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

If this function returns with success no other process can open this device. The same process can open additional handles to the device.

The operation fails with USBIO_ERR_DEVICE_OPENED if the device is opened by a different process.

This function can only be used after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)

ReleaseDevice Method

This function releases the exclusive use of a device.

Definition

```
HRESULT
ReleaseDevice(
     [out] int* Status
     );
```

Parameter

Status
A variable that receives the completion status of the operation. The returned value is zero in case of success, or an error code otherwise. Error codes are listed in section 6.4.

Comments

If this function returns with success other processes can open the device. The device is also released if the last handle of a process is closed.

This function can only be used after the device was opened, see **OpenDevice**.

See Also

CloseDevice (page 35)

OpenCount Property

This read-only property returns the number of all open handles to this device.

Definition

```
HRESULT
OpenCount(
     [out,retval] int* OpenCountValue
     );
```

Parameter

OpenCountValue
A variable that returns the count of open handles.

Comments

This property returns the number of all open handles. This includes handles that are opened by other applications and the handle with is used to make the function call.

If the property returns 0 the device is not opened.

This property can only be read after the device was opened, see **OpenDevice**.

See Also

OpenDevice (page 33)

6.2 _IUSBIOInterfaceEvents2 Interface

The **_IUSBIOInterfaceEvents2** interface provides the event call-back functions required to handle events issued by the USBIO COM object.

In some programming languages wrapper classes are used to encapsulate call-back functions. A wrapper class can be automatically generated by some language-specific tools. Note that the declaration of the functions in the wrapper class can be different from the declaration provided by **_IUSBIOInterfaceEvents2**.

The following table summarizes the members of **_IUSBIOInterfaceEvents2**. The methods are described in detail in this section.

Table 4: Members of _IUSBIOInterfaceEvents2

Member	Description
ReadComplete	Data is available for reading
WriteComplete	Free buffer space is available
WriteStatusAvailable	Write completion status is available
PnPAddNotification	A new USB device is connected to the system
PnPRemoveNotification	A USB device is removed from the system

ReadComplete Method

This function is called when the **ReadComplete** event is issued by the USBIO COM object instance.

Definition

```
HRESULT
ReadComplete(
     IDispatch* Obj
     );
```

Parameter

Obj
Pointer to the USBIO COM object instance.

Comments

The **ReadComplete** event is issued when data is available for reading by the application.

See Also

StartReading (page 80)
ReadData (page 83)
ReadIsoData (page 85)

WriteComplete Method

This function is called when the **WriteComplete** event is issued by the USBIO COM object instance.

Definition

```
HRESULT
WriteComplete(
     IDispatch* Obj
     );
```

Parameter

Obj
Pointer to the USBIO COM object instance.

Comments

The **WriteComplete** event is issued when buffer space is available. The application can write more data.

See Also

StartWriting (page 88)
WriteData (page 92)
WriteIsoData (page 96)
GetWriteStatus (page 94)
GetIsoWriteStatus (page 98)
WriteStatusAvailable (page 121)

WriteStatusAvailable Method

This function is called when the **WriteStatusAvailable** event is issued by the USBIO COM object instance.

Definition

```
HRESULT
WriteStatusAvailable(
     IDispatch* Obj
     );
```

Parameter

Obj
Pointer to the USBIO COM object instance.

Comments

The **WriteStatusAvailable** event is issued when a write operation was completed. The completion status of the operation can be retrieved.

See Also

StartWriting (page 88)
WriteData (page 92)
WriteIsoData (page 96)
GetWriteStatus (page 94)
GetIsoWriteStatus (page 98)
WriteComplete (page 120)

PnPAddNotification Method

This function is called when the **PnPAddNotification** event is issued by the USBIO COM object instance.

Definition

```
HRESULT
PnPAddNotification(
     IDispatch* Obj
     );
```

Parameter

Obj
Pointer to the USBIO COM object instance.

Comments

The **PnPAddNotification** event is issued when a USB device is connected to the host. A device enumeration should be performed by the application, see **EnumerateDevices**.

See Also

EnablePnPNotification (page 106)
DisablePnPNotification (page 107)
EnumerateDevices (page 31)
OpenDevice (page 33)
PnPRemoveNotification (page 123)

PnPRemoveNotification Method

This function is called when the **PnPRemoveNotification** event is issued by the USBIO COM object instance.

Definition

```
HRESULT
PnPRemoveNotification(
     IDispatch* Obj
     );
```

Parameter

Obj
Pointer to the USBIO COM object instance.

Comments

The **PnPRemoveNotification** event is issued when a USB device is disconnected from the host.

In order to determine the device instance which has been removed the application can send a standard request, for example **GetDeviceDescriptor**, to each active device instance. If the request succeeds the device is still present. If the request fails with an error code of **USBIO_ERR_DEVICE_NOT_PRESENT** the device has been removed.

An application should close all USBIO COM object instances open for a USB device that has been removed. This will free all associated resources.

See Also

EnablePnPNotification (page 106)
DisablePnPNotification (page 107)
CloseDevice (page 35)
EnumerateDevices (page 31)
PnPAddNotification (page 122)

6.3 Enumeration Types

This section describes the enumeration types used in conjunction with the **IUSBIOInterface** interface in detail.

USBIOCOM_INFO_FLAGS

This enumeration defines constants that provide information on the USBIO device driver that is currently running.

Definition

```
typedef enum {
     USBIOCOM_INFOFLAG_CHECKED_BUILD  = 0x00000010,
     USBIOCOM_INFOFLAG_DEMO_VERSION  = 0x00000020,
     USBIOCOM_INFOFLAG_LIGHT_VERSION  = 0x00000100
} USBIOCOM_INFO_FLAGS;
```

Entries

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

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

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

Comments

The values defined by **USBIOCOM_INFO_FLAGS** are used in conjunction with the **GetDriverInfo** method.

See Also

GetDriverInfo (page 37)

USBIOCOM_DEVICE_OPTION_FLAGS

This enumeration defines constants used to customize the behavior of the USBIO device driver.

Definition

```
typedef enum {
     USBIOCOM_RESET_DEVICE_ON_CLOSE  = 0x00000001,
     USBIOCOM_UNCONFIGURE_ON_CLOSE  = 0x00000002,
     USBIOCOM_ENABLE_REMOTE_WAKEUP  = 0x00000004
} USBIOCOM_DEVICE_OPTION_FLAGS;
```

Entries

USBIOCOM_RESET_DEVICE_ON_CLOSE
If this option is set a USB device reset is sent to the device after the last USBIO COM object instance has closed the device by a call to **CloseDevice**.

USBIOCOM_UNCONFIGURE_ON_CLOSE
If this option is set the USB device will be unconfigured after the last USBIO COM object instance has closed the device by a call to **CloseDevice**.

USBIOCOM_ENABLE_REMOTE_WAKEUP
If this option is set the remote wake-up feature is enabled for the device. At least one USBIO COM object instance must be open for the device to allow the remote wake-up event to occur.

Comments

The values defined by **USBIOCOM_DEVICE_OPTION_FLAGS** are used in conjunction with the **DeviceOptions** property.

See Also

DeviceOptions (page 42)
OpenDevice (page 33)
CloseDevice (page 35)

USBIOCOM_PIPE_OPTION_FLAGS

This enumeration defines constants used to define the behavior of data transfers.

Definition

```
typedef enum {
     USBIOCOM_SHORT_TRANSFER_OK  = 0x00010000
} USBIOCOM_PIPE_OPTION_FLAGS;
```

Entry

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

Comments

The values defined by **USBIOCOM_PIPE_OPTION_FLAGS** are used with various functions, see **ClassOrVendorInRequest**.

See Also

ClassOrVendorInRequest (page 65)

USBIOCOM_REQUEST_RECIPIENT

This enumeration defines constants that identify the recipient of a USB device request.

Definition

```
typedef enum {
     USBIOCOM_RecipientDevice  = 0,
     USBIOCOM_RecipientInterface  = 1,
     USBIOCOM_RecipientEndpoint  = 2,
     USBIOCOM_RecipientOther  = 3
} USBIOCOM_REQUEST_RECIPIENT;
```

Comments

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

See Also

GetDescriptor (page 46)
SetDescriptor (page 53)
ClassOrVendorInRequest (page 65)
ClassOrVendorOutRequest (page 67)
SetFeature (page 69)
ClearFeature (page 70)
GetStatus (page 75)

USBIOCOM_REQUEST_TYPE

This enumeration defines constants that identify the type of a USB device request.

Definition

```
typedef enum {
     USBIOCOM_RequestTypeClass  = 1,
     USBIOCOM_RequestTypeVendor  = 2
} USBIOCOM_REQUEST_TYPE;
```

Comments

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

Note that the enumeration does not contain the Standard request type defined by the USB specification. This is because the explicit generation of Standard requests is not supported by the USBD interface and by the USBIO driver, respectively. USB Standard requests are generated implicitely on various functions.

See Also

ClassOrVendorInRequest (page 65)
ClassOrVendorOutRequest (page 67)

USBIOCOM_PIPE_TYPE

This enumeration defines constants that identify the type of a USB pipe, or a USB endpoint, respectively.

Definition

```
typedef enum {
     USBIOCOM_PipeTypeControl  = 0,
     USBIOCOM_PipeTypeIsochronous  = 1,
     USBIOCOM_PipeTypeBulk  = 2,
     USBIOCOM_PipeTypeInterrupt  = 3
} USBIOCOM_PIPE_TYPE;
```

Comments

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

USBIOCOM_DEVICE_POWER_STATE

This enumeration defines constants that describe the power state of a USB device.

Definition

```
typedef enum {
     USBIOCOM_DevicePowerStateD0  = 0,
     USBIOCOM_DevicePowerStateD1  = 1,
     USBIOCOM_DevicePowerStateD2  = 2,
     USBIOCOM_DevicePowerStateD3  = 3
} USBIOCOM_DEVICE_POWER_STATE;
```

Entries

USBIOCOM_DevicePowerStateD0
Device fully on, normal operation.

USBIOCOM_DevicePowerStateD1
Device in suspend state.

USBIOCOM_DevicePowerStateD2
Device in suspend state.

USBIOCOM_DevicePowerStateD3
Device off.

Comments

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

The values defined by **USBIOCOM_DEVICE_POWER_STATE** are used with the functions **GetDevicePowerState** and **SetDevicePowerState**.

See Also

GetDevicePowerState (page 71)
SetDevicePowerState (page 72)

USBIOCOM_DESCRIPTOR_TYPE

This enumeration defines constants that identify the type of a USB descriptor.

Definition

```
typedef enum {
     USB_DeviceDescriptorType  = 0x01,
     USB_ConfigurationDescriptorType  = 0x02,
     USB_StringDescriptorType  = 0x03,
     USB_InterfaceDescriptorType  = 0x04,
     USB_EndpointDescriptorType  = 0x05,
     USB_HID_DescriptorType  = 0x21
} USBIOCOM_DESCRIPTOR_TYPE;
```

Entries

USB_DeviceDescriptorType
USB device descriptor.

USB_ConfigurationDescriptorType
USB configuration descriptor, which contains further configuration, interface, endpoint, and class specific descriptors.

USB_StringDescriptorType
USB string descriptor.

USB_InterfaceDescriptorType
USB interface descriptor.

USB_EndpointDescriptorType
USB endpoint descriptor.

USB_HID_DescriptorType
USB HID descriptor. This is a class-specific descriptor. Note that the code 0x21 is also used by other classes.

Comments

The values are defined by the Universal Serial Bus Specification, Chapter 9.

See Also

GetDescriptor (page 46)
SetDescriptor (page 53)

USBIOCOM_QUERY_PIPE_STATISTICS_FLAGS

This enumeration defines constants that are intended for modifying the behaviour of the **QueryPipeStatistics** function.

Definition

```
typedef enum {
     USBIOCOM_QPS_FLAG_RESET_BYTES_TRANSFERRED  = 0x00000001,
     USBIOCOM_QPS_FLAG_RESET_REQUESTS_SUCCEEDED  = 0x00000002,
     USBIOCOM_QPS_FLAG_RESET_REQUESTS_FAILED  = 0x00000004,
     USBIOCOM_QPS_FLAG_RESET_ALL_COUNTERS  = 0x00000007
} USBIOCOM_QUERY_PIPE_STATISTICS_FLAGS;
```

Entries

USBIOCOM_QPS_FLAG_RESET_BYTES_TRANSFERRED
Reset the BytesTransferred counter to zero after its current value has been captured and returned.

USBIOCOM_QPS_FLAG_RESET_REQUESTS_SUCCEEDED
Reset the RequestsSucceeded counter to zero after its current value has been captured and returned.

USBIOCOM_QPS_FLAG_RESET_REQUESTS_FAILED
Reset the RequestsFailed counter to zero after its current value has been captured and returned.

USBIOCOM_QPS_FLAG_RESET_ALL_COUNTERS
Reset all three counters BytesTransferred, RequestsSucceeded, and RequestsFailed to zero after their current values have been captured and returned. This value combines the three flags described above.

Comments

The values defined by **USBIOCOM_QUERY_PIPE_STATISTICS_FLAGS** are used in conjunction with the **QueryPipeStatistics** method.

See Also

QueryPipeStatistics (page 112)

6.4 Error Codes

This section lists the error codes returned by the members of the **`IUSBIOInterface`** interface.

All the private error codes defined by the USBIO device driver can be returned by members of **`IUSBIOInterface`** as well. Those codes are defined in *usbio_i.h* and described in the USBIO Reference Manual. In addition, the USBIO COM object can return the error codes listed in Table 5.

Note that USBIO private status codes range from 0xE0000000 to 0xE000FFFF.

An error code can be translated to a description text by means of the **`IUSBIOInterface`** method **ErrorText**.

Table 5: USBIO COM Error Codes

Symbolic Name	Code
USBIO_ERR_SUCCESS	0x00000000L
USBIO_ERR_NOT_ENUMERATED	0xE000A001L
USBIO_ERR_TOO_MANY_INTERFACES	0xE000A002L
USBIO_ERR_NO_INTERFACE	0xE000A003L
USBIO_ERR_START_THREAD_FAILED	0xE000A004L
USBIO_ERR_NO_DATA	0xE000A005L
USBIO_ERR_BUFFER_TOO_SMALL	0xE000A006L
USBIO_ERR_THREAD_IS_RUNNING	0xE000A007L
USBIO_ERR_INVALID_PIPE_TYPE	0xE000A008L
USBIO_ERR_NO_BUFFER	0xE000A009L
USBIO_ERR_BUFFER_TOO_LARGE	0xE000A00AL
USBIO_ERR_WRITE_NOT_STARTED	0xE000A00BL
USBIO_ERR_READ_NOT_STARTED	0xE000A00CL
USBIO_ERR_INVALID_ISO_BUFFER	0xE000A00DL
USBIO_ERR_STATUS_ARRAY_TOO_SMALL	0xE000A00EL
USBIO_ERR_INVALID_ARRAY	0xE000A00FL
USBIO_ERR_DEVICE_ALREADY_OPEN	0xE000A010L
USBIO_ERR_ALREADY_CALLED	0xE000A011L

7 Related Documents

- USBIO 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