

Cypress CyAPI Programmer's Reference

© 2003 Cypress Semiconductor

All rights reserved. No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher.

Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this document or from the use of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Table of Contents

Part I Overview	5
Part II CCyBulkEndPoint	5
1 BeginDataXfer()	6
2 CCyBulkEndPoint()	7
3 CCyBulkEndPoint()	7
Part III CCyControlEndPoint	7
1 BeginDataXfer()	8
2 CCyControlEndPoint()	9
3 CCyControlEndPoint()	9
4 Direction	9
5 Index	10
6 Read()	11
7 ReqCode	11
8 ReqType	12
9 Target	12
10 Value	13
11 Write()	13
Part IV CCyInterruptEndPoint	14
1 BeginDataXfer()	14
2 CCyInterruptEndPoint()	15
3 CCyInterruptEndPoint()	15
Part V CCylsocEndPoint	16
1 BeginDataXfer()	16
2 CCylsocEndPoint()	18
3 CCylsocEndPoint()	18
4 CreatePktInfos()	18
Part VI CCylsoPktInfo	19
Part VII CCyUSBDevice	20
1 AltIntfc()	21
2 AltIntfcCount()	21
3 BcdDevice	21

4	BcdUSB	21
5	BulkInEndPt	21
6	BulkOutEndPt	22
7	CCyUSBDevice()	22
8	~CCyUSBDevice()	24
9	Close()	24
10	Config()	25
11	ConfigAttrib	25
12	ConfigCount()	25
13	ConfigValue	25
14	ControlEndPt	25
15	DevClass	26
16	DeviceCount()	26
17	DeviceHandle()	26
18	DeviceName	26
19	DevProtocol	27
20	DevSubClass	27
21	DriverGUID()	27
22	DriverVersion	27
23	EndPointCount()	27
24	EndPointOf()	28
25	EndPoints	28
26	FriendlyName	29
27	GetDeviceDescriptor()	29
28	GetConfigDescriptor()	29
29	GetIntfcDescriptor()	29
30	GetUSBConfig()	29
31	Interface()	30
32	InterruptInEndPt	30
33	InterruptOutEndPt	31
34	IntfcClass	31
35	IntfcCount()	32
36	IntfcProtocol	32
37	IntfcSubClass	32
38	IsocInEndPt	32
39	IsocOutEndPt	33
40	IsOpen()	33
41	Manufacturer	33
42	MaxPacketSize	33

43	MaxPower	34
44	NtStatus	34
45	Open()	34
46	PowerState()	35
47	Product	35
48	ProductID	35
49	ReConnect()	35
50	Reset()	35
51	Resume()	36
52	SerialNumber	36
53	SetConfig()	36
54	SetAltIntfc()	36
55	StrLangID	36
56	Suspend()	37
57	USBAddress	37
58	USBDIVersion	37
59	UsbdStatus	37
60	UsbdStatusString()	37
61	VendorID	38

Part VIII CCyUSBConfig 38

1	AltInterfaces	39
2	bConfigurationValue	40
3	bDescriptorType	40
4	bLength	40
5	bmAttributes	40
6	bNumInterfaces	40
7	CCyUSBConfig()	40
8	CCyUSBConfig()	40
9	CCyUSBConfig()	41
10	~CCyUSBConfig	41
11	iConfiguration	41
12	Interfaces	41
13	wTotalLength	42

Part IX CCyUSBEndPoint 43

1	Abort()	43
2	Address	43
3	Attributes	44
4	BeginDataXfer()	44

5	bln	45
6	CCyUSBEndPoint()	46
7	CCyUSBEndPoint()	46
8	CCyUSBEndPoint()	46
9	DscLen	47
10	DscType	47
11	GetXferSize()	47
12	FinishDataXfer()	47
13	hDevice	48
14	Interval	49
15	MaxPktSize	49
16	NtStatus	49
17	Reset()	49
18	SetXferSize()	49
19	TimeOut	50
20	UsbdStatus	50
21	WaitForXfer()	50
22	XferData()	51
Part X CCyUSBInterface		52
1	bAlternateSetting	53
2	bAltSettings	54
3	bDescriptorType	54
4	CCyUSBInterface()	54
5	CCyUSBInterface()	54
6	bInterfaceClass	55
7	bInterfaceNumber	55
8	bInterfaceProtocol	55
9	bInterfaceSubClass	55
10	bLength	55
11	bNumEndpoints	55
12	EndPoints	56
13	iInterface	57

1 Overview

CyAPI.lib provides a simple, powerful C++ programming interface to USB devices. More specifically, it is a C++ class library that provides a high-level programming interface to the **CyUsb.sys** device driver. The library is only able to communicate with USB devices that are served by (i.e. matched to) this driver.

Rather than communicate with the driver via Windows API calls such as *SetupDiXxxx* and *DeviceIoControl*, applications call simpler CyAPI methods such as [Open](#), [Close](#), and [XferData](#) to communicate with USB devices.

To use the library, you need to include the header file, **CyAPI.h**, in files that access the **CCyUSBDevice** class. In addition, the statically linked **CyAPI.lib** file must be linked to your project. Versions of the .lib file are available for use with Microsoft Visual C++ 6 and 7, and Borland C++ Builder 6.0.

The library employs a **Device and EndPoints** use model. To use the library you must [create an instance](#) of the **CCyUSBDevice** class using the **new** keyword. A **CCyUSBDevice** object knows [how many USB devices](#) are attached to the **CyUsb.sys** driver and can be made to abstract any one of those devices at a time by using the [Open](#) method. An instance of **CCyUSBDevice** exposes several methods and data members that are device-specific, such as [DeviceName](#), [DevClass](#), [VendorID](#), [ProductID](#), and [SetAltIntfc](#).

When a **CCyUSBDevice** object is open to an attached USB device, its [endpoint](#) members provide an interface for performing data transfers to and from the device's endpoints. Endpoint-specific data members and methods such as [MaxPktSize](#), [TimeOut](#), [bIn](#), [Reset](#) and [XferData](#) are only accessible through endpoint members of a **CCyUSBDevice** object.

In addition to its simplicity, the class library facilitates creation of sophisticated applications as well. The **CCyUSBDevice** [constructor](#) automatically registers the application for Windows USB Plug and Play event notification. This allows your application to support "hot plugging" of devices. Also, the asynchronous [BeginDataXfer/WaitForXfer/FinishDataXfer](#) methods allow queuing of multiple data transfer requests on a single endpoint, thus enabling data streaming from the application level.

2 CCyBulkEndPoint

Header

CyUSB.h

Description

CCyBulkEndPoint is a subclass of the CCyUSBEndPoint abstract class. CCyBulkEndPoint exists to implement a bulk-specific [BeginDataXfer](#)() function.

Normally, you should not need to construct any of your own instances of this class. Rather, when an instance of CyUSBDevice is created, instances of this class are automatically created for all bulk endpoints as members of that class. Two such members of CyUSBDevice are [BulkInEndPt](#) and [BulkOutEndPt](#).

Example

```
// Find bulk endpoints in the EndPoints[] array
CCyBulkEndPoint *BulkInEpt = NULL;
CCyBulkEndPoint *BulkOutEpt = NULL;
```

```

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);
int eptCount = USBDevice->EndPointCount();

for (int i=1; i<eptCount; i++) {
    bool bIn = USBDevice->Endpoints[i]->Address & 0x80;
    bool bBulk = (USBDevice->Endpoints[i]->Attributes == 2);
    if (bBulk && bIn) BulkInEpt = (CCyBulkEndPoint *) USBDevice->Endpoints[i];
    if (bBulk && !bIn) BulkOutEpt = (CCyBulkEndPoint *) USBDevice->Endpoints[i];
}

```

2.1 BeginDataXfer()

Description

BeginDataXfer is an advanced method for performing asynchronous IO. This method sets-up all the parameters for a data transfer, initiates the transfer, and immediately returns, not waiting for the transfer to complete.

BeginDataXfer allocates a complex data structure and returns a pointer to that structure. [FinishDataXfer](#) de-allocates the structure. Therefore, it is imperative that each BeginDataXfer call have exactly one matching FinishDataXfer call.

You will usually want to use the synchronous [XferData](#) method rather than the asynchronous BeginDataXfer/WaitForXfer/FinishDataXfer approach.

Example

```

// This example assumes that the device automatically sends back,
// over its bulk-IN endpoint, any bytes that were received over its
// bulk-OUT endpoint (commonly referred to as a loopback function)

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

OVERLAPPED outOvLap, inOvLap;
outOvLap.hEvent = CreateEvent(NULL, false, false, "CYUSB_OUT");
inOvLap.hEvent = CreateEvent(NULL, false, false, "CYUSB_IN");

char inBuf[128];
ZeroMemory(inBuf, 128);

char buffer[128];
LONG length = 128;

// Just to be cute, request the return data before initiating the loopback
UCHAR *inContext = USBDevice->BulkInEndPt->BeginDataXfer(inBuf, length,
&inOvLap);
UCHAR *outContext = USBDevice->BulkOutEndPt->BeginDataXfer(buffer, length,
&outOvLap);

USBDevice->BulkOutEndPt->WaitForXfer(&outOvLap,100);
USBDevice->BulkInEndPt->WaitForXfer(&inOvLap,100);

USBDevice->BulkOutEndPt->FinishDataXfer(buffer, length, &outOvLap,outContext);
USBDevice->BulkInEndPt->FinishDataXfer(inBuf, length, &inOvLap,inContext);

CloseHandle(outOvLap.hEvent);
CloseHandle(inOvLap.hEvent);

```

2.2 CCyBulkEndPoint()

Description

This is the default constructor for the `CCyBulkEndPoint` class.

The resulting instance has most of its member variables initialized to zero. The two exceptions are [hDevice](#), which gets set to `INVALID_HANDLE_VALUE` and [TimeOut](#) which is set to 10,000 (10 seconds).

2.3 CCyBulkEndPoint()

Description

This constructor creates a legitimate `CCyBulkEndPoint` object through which bulk transactions can be performed on the endpoint.

The constructor is called by the library, itself, in the process of performing the [Open\(\)](#) method of the `CCyUSBDevice`.

You should never need to invoke this constructor. Instead, you should use the `CCyBulkEndPoint` objects created for you by the `CCyUSBDevice` class and accessed via its [EndPoints](#), [BulkInEndPt](#) and [BulkOutEndPt](#) members.

3 CCyControlEndPoint

Header

CyUSB.h

Description

`CCyControlEndPoint` is a subclass of the `CCyUSBEndPoint` abstract class.

Instances of this class can be used to perform control transfers to the device.

Control transfers require 6 parameters that are not needed for bulk, isoc, or interrupt transfers. These are:

[Target](#)
[ReqType](#)
[Direction](#)
[ReqCode](#)
[Value](#)
[Index](#)

All USB devices have at least one Control endpoint, endpoint zero. Whenever an instance of `CCyUSBDevice` successfully performs its [Open\(\)](#) function, an instance of `CCyControlEndPoint` called [ControlEndPt](#) is created. Normally, you will use this [ControlEndPt](#) member of `CCyUSBDevice` to perform all your Control endpoint data transfers.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

// Just for typing efficiency
CCyControlEndPoint *ept = USBDevice->ControlEndPt;

ept->Target      = TGT_DEVICE;
ept->ReqType     = REQ_VENDOR;
ept->Direction   = DIR_TO_DEVICE;
ept->ReqCode     = 0x05;
ept->Value       = 1;
ept->Index       = 0;

char buf[512];
ZeroMemory(buf, 512);
LONG buflen = 512;

ept->XferData(buf, buflen);
```

3.1 BeginDataXfer()

Description

BeginDataXfer is an advanced method for performing asynchronous IO.

This method sets-up all the parameters for a data transfer, initiates the transfer, and immediately returns, not waiting for the transfer to complete.

BeginDataXfer allocates a complex data structure and returns a pointer to that structure. [FinishDataXfer](#) de-allocates the the structure. Therefore, it is imperative that each BeginDataXfer call have exactly one matching FinishDataXfer call. You will usually want to use the synchronous [XferData](#) method rather than the asynchronous BeginDataXfer/WaitForXfer/FinishDataXfer approach.

Control transfers require 6 parameters that are not needed for bulk, isoc, or interrupt transfers. These are:

[Target](#)
[ReqType](#)
[Direction](#)
[ReqCode](#)
[Value](#)
[Index](#)

Be sure to set the value of these CCyControlEndPoint members before invoking the BeginDataXfer or [XferData](#) methods.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

// Just for typing efficiency
CCyControlEndPoint *ept = USBDevice->ControlEndPt;

OVERLAPPED OvLap;
OvLap.hEvent = CreateEvent(NULL, false, false, "CYUSB_CTL");
```

```

char buffer[128];
LONG length = 128;
ept->Target = TGT_DEVICE;
ept->ReqType = REQ_VENDOR;
ept->Direction = DIR_TO_DEVICE;
ept->ReqCode = 0x05;
ept->Value = 1;
ept->Index = 0;

PUCHAR Context = ept->BeginDataXfer(buffer, length, &OvLap);
ept->WaitForXfer(&OvLap, 100);
ept->FinishDataXfer(buffer, length, &OvLap, Context);

CloseHandle(OvLap.hEvent);

```

3.2 CCyControlEndPoint()

Description

This is the default constructor for the `CCyControlEndPoint` class.

It sets the class' data members to:

```

Target      = TGT_DEVICE
ReqType     = REQ_VENDOR
Direction   = DIR_TO_DEVICE
ReqCoe      = 0
Value       = 0
Index       = 0

```

3.3 CCyControlEndPoint()

Description

This is the primary constructor for the `CCyControlEndPoint` class.

It sets the class' data members to:

```

Target      = TGT_DEVICE
ReqType     = REQ_VENDOR
Direction   = DIR_TO_DEVICE
ReqCoe      = 0
Value       = 0
Index       = 0

```

3.4 Direction

Description

Direction is one of the essential parameters for a Control transfer and a data member of the `CCyControlEndPoint` class.

Legitimate values for the Direction member are `DIR_TO_DEVICE` and `DIR_FROM_DEVICE`.

Unlike Bulk, Interrupt and ISOC endpoints, which are uni-directional (either IN or OUT), the Control endpoint is bi-directional. It can be used to send data to the device or read data from the device. So, the direction of the transaction is one of the fundamental parameters required for each Control

transfer.

Direction is automatically set to DIR_TO_DEVICE by the [Write\(\)](#) method. It is automatically set to DIR_FROM_DEVICE by the [Read\(\)](#) method.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

// Just for typing efficiency
CCyControlEndPoint *ept = USBDevice->ControlEndPt;

ept->Target      = TGT_DEVICE;
ept->ReqType     = REQ_VENDOR;
ept->Direction   = DIR_TO_DEVICE;
ept->ReqCode     = 0x05;
ept->Value       = 1;
ept->Index       = 0;

char buf[512];
ZeroMemory(buf, 512);
LONG buflen = 512;

ept->XferData(buf, buflen);
```

3.5 Index

Description

Index is one of the essential parameters for a Control transfer and a data member of the CCyControlEndPoint class.

Index values typically depend on the specific ReqCode that is being sent in the Control transfer.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

// Just for typing efficiency
CCyControlEndPoint *ept = USBDevice->ControlEndPt;

ept->Target      = TGT_DEVICE;
ept->ReqType     = REQ_VENDOR;
ept->Direction   = DIR_TO_DEVICE;
ept->ReqCode     = 0x05;
ept->Value       = 1;
ept->Index       = 0;

char buf[512];
ZeroMemory(buf, 512);
LONG buflen = 512;

ept->XferData(buf, buflen);
```

3.6 Read()

Description

Read() sets the CyControlEndPoint [Direction](#) member to DIR_FROM_DEVICE and then calls [CCyUSBEndPoint::XferData\(\)](#).

The **buf** parameter points to a memory buffer where the read bytes will be placed.

The **len** parameter tells how many bytes are to be read.

Returns **true** if the read operation was successful.

Passes-back the actual number of bytes transferred in the **len** parameter.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

// Just for typing efficiency
CCyControlEndPoint *ept = USBDevice->ControlEndPt;

ept->Target = TGT_DEVICE;
ept->ReqType = REQ_VENDOR;
ept->ReqCode = 0x07;
ept->Value = 1;
ept->Index = 0;

char buf[512];
LONG bytesToRead = 64;

ept->Read(buf, bytesToRead);
```

3.7 ReqCode

Description

ReqCode is one of the essential parameters for a Control transfer and a data member of the CCyControlEndPoint class.

ReqCode values indicate, to the USB chip, a particular function or command that the chip should perform. They are usually documented by the USB chip manufacturer.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

// Just for typing efficiency
CCyControlEndPoint *ept = USBDevice->ControlEndPt;

ept->Target = TGT_DEVICE;
ept->ReqType = REQ_VENDOR;
ept->Direction = DIR_TO_DEVICE;
ept->ReqCode = 0x05;
ept->Value = 1;
ept->Index = 0;
```

```
char buf[512];
ZeroMemory(buf, 512);
LONG buflen = 512;

ept->XferData(buf, buflen);
```

3.8 ReqType

Description

ReqType is one of the essential parameters for a Control transfer and a data member of the CCyControlEndPoint class.

Legitimate values for the ReqType member are **REQ_STD**, **REQ_CLASS** and **REQ_VENDOR**.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

// Just for typing efficiency
CCyControlEndPoint *ept = USBDevice->ControlEndPt;

ept->Target      = TGT_DEVICE;
ept->ReqType     = REQ_VENDOR;
ept->Direction   = DIR_TO_DEVICE;
ept->ReqCode     = 0x05;
ept->Value       = 1;
ept->Index       = 0;

char buf[512];
ZeroMemory(buf, 512);
LONG buflen = 512;

ept->XferData(buf, buflen);
```

3.9 Target

Description

Target is one of the essential parameters for a Control transfer and a data member of the CCyControlEndPoint class.

Legitimate values for the Target member are **TGT_DEVICE**, **TGT_INTFC**, **TGT_ENDPT** and **TGT_OTHER**.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

// Just for typing efficiency
CCyControlEndPoint *ept = USBDevice->ControlEndPt;

ept->Target      = TGT_DEVICE;
ept->ReqType     = REQ_VENDOR;
ept->Direction   = DIR_TO_DEVICE;
```

```

ept->ReqCode   = 0x05;
ept->Value     = 1;
ept->Index     = 0;

char buf[512];
ZeroMemory(buf, 512);
LONG buflen = 512;

ept->XferData(buf, buflen);

```

3.10 Value

Description

Value is one of the essential parameters for a Control transfer and a data member of the `CCyControlEndPoint` class.

Values typically depend on the specific `ReqCode` that is being sent in the Control transfer.

Example

```

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

// Just for typing efficiency
CCyControlEndPoint *ept = USBDevice->ControlEndPt;

ept->Target     = TGT_DEVICE;
ept->ReqType    = REQ_VENDOR;
ept->Direction  = DIR_TO_DEVICE;
ept->ReqCode    = 0x05;
ept->Value      = 1;
ept->Index      = 0;

char buf[512];
ZeroMemory(buf, 512);
LONG buflen = 512;

ept->XferData(buf, buflen);

```

3.11 Write()

Description

`Write()` sets the `CyControlEndPoint` [Direction](#) member to `DIR_TO_DEVICE` and then calls [CCyUSBEndPoint::XferData\(\)](#).

The **buf** parameter points to a memory buffer where the read bytes will be placed.

The **len** parameter tells how many bytes are to be read.

Returns **true** if the write operation was successful.

Passes-back the actual number of bytes transferred in the **len** parameter.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

// Just for typing efficiency
CCyControlEndPoint *ept = USBDevice->ControlEndPt;

ept->Target    = TGT_DEVICE;
ept->ReqType   = REQ_VENDOR;
ept->ReqCode   = 0x07;
ept->Value     = 1;
ept->Index     = 0;

char buf[512];
ZeroMemory(buf, 512);
LONG bytesToSend = 128;

ept->Write(buf, bytesToSend);
```

4 CCyInterruptEndPoint

Header

CyUSB.h

Description

CCyInterruptEndPoint is a subclass of the [CCyUSBEndPoint](#) abstract class.

CCyInterruptEndPoint exists to implement a interrupt-specific [BeginDataXfer\(\)](#) function.

Normally, you should not need to construct any of your own instances of this class. Rather, when an instance of [CyUSBDevice](#) is created, instances of this class are automatically created as members of that class. Two such members of CyUSBDevice are [InterruptInEndPt](#) and [InterruptOutEndPt](#).

Example

```
// Find interrupt endpoints in the EndPoints[] array
CCyInterruptEndPoint *IntInEpt = NULL;
CCyInterruptEndPoint *IntOutEpt = NULL;
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

int eptCount = USBDevice->EndPointCount();
for (int i=1; i<eptCount; i++) {
    bool bIn = USBDevice->EndPoints[i]->Address & 0x80;
    bool bInt = (USBDevice->EndPoints[i]->Attributes == 3);

    if (bInt && bIn) IntInEpt = (CCyInterruptEndPoint *) USBDevice->EndPoints[i];
    if (bInt && !bIn) IntOutEpt = (CCyInterruptEndPoint *) USBDevice-
>EndPoints[i];
}
```

4.1 BeginDataXfer()

Description

BeginDataXfer is an advanced method for performing asynchronous IO. This method sets-up all the

parameters for a data transfer, initiates the transfer, and immediately returns, not waiting for the transfer to complete.

`BeginDataXfer` allocates a complex data structure and returns a pointer to that structure. [FinishDataXfer](#) de-allocates the structure. Therefore, it is imperative that each `BeginDataXfer` call have exactly one matching `FinishDataXfer` call.

You will usually want to use the synchronous [XferData](#) method rather than the asynchronous `BeginDataXfer/WaitForXfer/FinishDataXfer` approach.

Example

```
// This example assumes that the device automatically sends back,
// over its bulk-IN endpoint, any bytes that were received over its
// bulk-OUT endpoint (commonly referred to as a loopback function)

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

OVERLAPPED outOvLap, inOvLap;
outOvLap.hEvent = CreateEvent(NULL, false, false, "CYUSB_OUT");
inOvLap.hEvent = CreateEvent(NULL, false, false, "CYUSB_IN");

char inBuf[128];
ZeroMemory(inBuf, 128);

char buffer[128];
LONG length = 128;

// Just to be cute, request the return data before initiating the loopback
UCHAR *inContext = USBDevice->BulkInEndPt->BeginDataXfer(inBuf, length,
&inOvLap);
UCHAR *outContext = USBDevice->BulkOutEndPt->BeginDataXfer(buffer, length,
&outOvLap);

USBDevice->BulkOutEndPt->WaitForXfer(&outOvLap,100);
USBDevice->BulkInEndPt->WaitForXfer(&inOvLap,100);

USBDevice->BulkOutEndPt->FinishDataXfer(buffer, length, &outOvLap,outContext);
USBDevice->BulkInEndPt->FinishDataXfer(inBuf, length, &inOvLap,inContext);

CloseHandle(outOvLap.hEvent);
CloseHandle(inOvLap.hEvent);
```

4.2 CCyInterruptEndPoint()

Description

This is the default constructor for the `CCyInterruptEndPoint` class.

The resulting instance has most of its member variables initialized to zero. The two exceptions are [hDevice](#), which gets set to `INVALID_HANDLE_VALUE` and [TimeOut](#) which is set to 10,000 (10 seconds).

4.3 CCyInterruptEndPoint()

Description

This constructor creates a legitimate CCyInterruptEndPoint object through which interrupt transactions can be performed on the endpoint.

The constructor may be called by the library, itself, in the process of performing the [Open\(\)](#) method of the CCyUSBDevice.

You should never need to invoke this constructor. Instead, you should use the CCyInterruptEndPoint objects created for you by the CCyUSBDevice class and accessed via its [EndPoints](#), [InterruptInEndPoint](#) and [InterruptOutEndPoint](#) members.

5 CCyIsocEndPoint

Header
CyUSB.h

Description

CCyIsocEndPoint is a subclass of the [CCyUSBEndPoint](#) abstract class.

CCyIsocEndPoint exists to implement a isoc-specific [BeginDataXfer\(\)](#) function.

Normally, you should not need to construct any of your own instances of this class. Rather, when an instance of [CyUSBDevice](#) is created, instances of this class are automatically created as members of that class. Two such members of CyUSBDevice are [IsocInEndPoint](#) and [IsocOutEndPoint](#).

NOTE: For ISOC transfers, the buffer length and the endpoint's transfers size (see [SetXferSize](#)) must be a multiple of 8 times the endpoint's [MaxPktSize](#).

Example

```
// Find isoc endpoints in the EndPoints[] array
CCyIsocEndPoint *IsocInEpt = NULL;
CCyIsocEndPoint *IsocOutEpt = NULL;

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);
int eptCount = USBDevice->EndPointCount();

for (int i=1; i<eptCount; i++) {
    bool bIn = USBDevice->EndPoints[i]->Address & 0x80;
    bool bInt = (USBDevice->EndPoints[i]->Attributes == 1);
    if (bInt && bIn) IsocInEpt = (CCyIsocEndPoint *) USBDevice->EndPoints[i];
    if (bInt && !bIn) IsocOutEpt = (CCyIsocEndPoint *) USBDevice->EndPoints[i];
}
```

5.1 BeginDataXfer()

Description

BeginDataXfer is an advanced method for performing asynchronous IO. This method sets-up all the parameters for a data transfer, initiates the transfer, and immediately returns, not waiting for the transfer to complete.

BeginDataXfer allocates a complex data structure and returns a pointer to that structure. [FinishDataXfer](#) de-allocates the structure. Therefore, it is imperative that each BeginDataXfer call

have exactly one matching FinishDataXfer call.

You will usually want to use the synchronous [XferData](#) method rather than the asynchronous BeginDataXfer/WaitForXfer/FinishDataXfer approach.

NOTE: For ISOC transfers, the buffer length and the endpoint's transfers size (see [SetXferSize](#)) must be a multiple of 8 times the endpoint's [MaxPktSize](#).

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(hWnd);
CCyIsocEndPoint *IsoIn = USBDevice->IsocInEndPt;

if (IsoIn) {

    int pkts = 16;
    LONG bufSize = IsoIn->MaxPktSize * pkts;

    PCHAR          context;
    OVERLAPPED     inOvLap;
    PCHAR          buffer      = new UCHAR[bufSize];
    CCyIsoPktInfo *isoPktInfos = new CCyIsoPktInfo[pkts];

    IsoIn->SetXferSize(bufSize);

    inOvLap.hEvent = CreateEvent(NULL, false, false, NULL);

    // Begin the data transfer
    context = IsoIn->BeginDataXfer(buffer, bufSize, &inOvLap);

    // Wait for the xfer to complete.
    if (!IsoIn->WaitForXfer(&inOvLap, 1500)) {
        IsoIn->Abort();
        // Wait for the stalled command to complete
        WaitForSingleObject(inOvLap.hEvent, INFINITE);
    }

    int complete = 0;
    int partial  = 0;

    // Must always call FinishDataXfer to release memory of contexts[i]
    if (IsoIn->FinishDataXfer(buffer, bufSize, &inOvLap, context, isoPktInfos))

        for (int i=0; i< pkts; i++)
            if (isoPktInfos[i].Status)
                partial++;
            else
                complete++;

    } else
        partial++;

    delete buffer;
    delete [] isoPktInfos;
}
}
```

5.2 CCylsocEndPoint()

Description

This is the default constructor for the CCylsocEndPoint class.

The resulting instance has most of its member variables initialized to zero. The two exceptions are [hDevice](#), which gets set to INVALID_HANDLE_VALUE and [TimeOut](#) which is set to 10,000 (10 seconds).

5.3 CCylsocEndPoint()

Description

This constructor creates a legitimate CCylsocEndPoint object through which isochronous transactions can be performed on the endpoint.

The constructor is called by the library, itself, in the process of performing the [Open\(\)](#) method of the CCyUSBDevice.

You should never need to invoke this constructor. Instead, you should use the CCylsocEndPoint objects created for you by the CCyUSBDevice class and accessed via its [EndPoints](#), [IsocInEndPt](#) and [IsocOutEndPt](#) members.

5.4 CreatePktInfos()

Description

The **CreatePktInfos** method is provided for convenience.

It creates an array of [CCyIsoPktInfo](#) objects to be used in calls to [XferData](#) and [FinishDataXfer](#) for Isoc endpoints.

CreatePktInfos calculates the number of isoc packets that the driver will use to transfer a data buffer of **bufLen** bytes. This number is returned in the **packets** parameter.

CreatePktInfos also dynamically constructs an array of [CCyIsoPktInfo](#) objects and returns a pointer to the first element of that array. There are **packets** elements in the array.

After using the array of CCyPktInfo objects you must delete the array of objects yourself by calling **delete []**.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice();
CCyIsocEndPoint *IsoIn = USBDevice->IsocInEndPt;

if (IsoIn) {
    LONG    bufSize = 4096;
```

```

PUCHAR buffer = new UCHAR[bufSize];

CCyIsoPktInfo *isoPktInfos;
int          pkts;

// Allocate the IsoPktInfo objects, and find-out how many were allocated
isoPktInfos = IsoIn->CreatePktInfos(bufSize, pkts);

if (IsoIn->XferData(buffer, bufSize, isoPktInfos)) {

    LONG recvdBytes = 0;

    for (int i=0; i<pkts; i++)
        if (isoPktInfos[i].Status == 0)
            recvdBytes += isoPktInfos[i].Length;

}

delete [] buffer;
delete [] isoPktInfos;

}

```

6 CCyIsoPktInfo

The **CCyIsoPktInfo** class is defined as:

```

class CCyIsoPktInfo {
public:
    LONG Status;
    LONG Length;
};

```

When an Isoc transfer is performed, the data buffer passed to [XferData](#) or [BeginDataXfer](#) is logically partitioned, by the driver, into multiple packets of data. The driver returns status and length information for each of those packets.

The [XferData](#) and [FinishDataXfer](#) methods of [CCyUSBEndPoint](#) accept an optional parameter that is a pointer to an array of **CCyIsoPktInfo** objects. If this parameter is not NULL, the array will be filled with the packet status and length information returned by the driver.

If the value returned in the **Status** field is zero (USB_D_STATUS_SUCCESS) all the data in the packet is valid. Other non-zero values for the Status field can be found in the DDK include file, USBDI.H.

The value returned in the **Length** field indicates the number of bytes transferred in the packet. In ideal conditions, this number will be bufferLength / numPackets (which is the maximum capacity of each packet). However, fewer bytes could be transferred.

An array of **CCyIsoPktInfo** objects can be easily created by invoking the [CCyUSBIsocEndPoint::CreatePktInfos](#) method.

Example

```

CCyUSBDevice *USBDevice = new CCyUSBDevice();
CCyIsocEndPoint *IsoIn = USBDevice->IsocInEndPt;

if (IsoIn) {

```

```
LONG   bufSize = 4096;
PUCHAR buffer = new UCHAR[bufSize];

CCyIsoPktInfo *isoPktInfos;
int         pkts;

// Allocate the IsoPktInfo objects, and find-out how many were allocated
isoPktInfos = IsoIn->CreatePktInfos(bufSize, pkts);

if (IsoIn->XferData(buffer, bufSize, isoPktInfos)) {

    LONG recvdBytes = 0;

    for (int i=0; i<pkts; i++)
        if (isoPktInfos[i].Status == 0)
            recvdBytes += isoPktInfos[i].Length;

}

delete [] buffer;
delete [] isoPktInfos;

}
```

7 CCyUSBDevice

Header
CyUSB.h

Description

The CCyUSBDevice class is the primary entry point into the library. All the functionality of the library should be accessed via an instance of CCyUSBDevice.

Create an instance of CCyUSBDevice using the **new** keyword.

An instance of CCyUSBDevice is aware of all the USB devices that are attached to the USB driver and can selectively communicate with any ONE of them by using the [Open\(\)](#) method.

Example

```
// Look for a device having VID = 0547, PID = 1002

USBDevice = new CCyUSBDevice(Handle); // Create an instance of CCyUSBDevice

int devices = USBDevice->DeviceCount();

int vID, pID;
int d = 0;

do {
    USBDevice->Open(d); // Open automatically calls Close() if necessary
    vID = USBDevice->VendorID;
    pID = USBDevice->ProductID;
    d++;
} while ((d < devices) && (vID != 0x0547) && (pID != 0x1002));
```

7.1 AltIntfc()

Description

This function returns the current alternate interface setting for the device.

A return value of 255 (0xFF) indicates that the driver failed to return the current alternate interface setting.

Call [SetAltIntfc\(\)](#) to select a different alternate interface (changing the AltSetting).

Call [AltIntfcCount\(\)](#) to find-out how many alternate interfaces are exposed by the device.

7.2 AltIntfcCount()

Description

This function returns the number of alternate interfaces exposed by the device.

The primary interface (AltSetting == 0) is not counted as an alternate interface.

Example

A return value of 2 means that there are 2 alternate interfaces, in addition to the primary interface. Legitimate parameter values for calls to [SetAltIntfc\(\)](#) would then be 0, 1 and 2.

7.3 BcdDevice

Description

This data member contains the value of the **bcdDevice** member from the device's USB descriptor structure.

7.4 BcdUSB

Description

This data member contains the value of the **bcdUSB** member from the device's USB descriptor structure.

7.5 BulkInEndPt

Description

BulkInEndPt is a pointer to an object representing the first BULK IN endpoint enumerated for the selected interface.

The selected interface might expose additional BULK IN endpoints. To discern this, one would need to traverse the [EndPoints](#) array, checking the [Attributes](#) and [Address](#) members of each [CCyUSBEndPoint](#) object referenced in the array.

If no BULK IN endpoints were enumerated by the device, BulkInEndPt will be set to NULL.

Example

```

// Find a second bulk IN endpoint in the EndPoints[] array
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

CCyBulkEndPoint *BulkIn2 = NULL;

int eptCount = USBDevice->EndPointCount();

for (int i=1; i<eptCount; i++) {
    bool bIn = USBDevice->EndPoints[i]->bIn;
    bool bBulk = (USBDevice->EndPoints[i]->Attributes == 2);

    if (bBulk && bIn) BulkIn2 = (CCyBulkEndPoint *) USBDevice->EndPoints[i];
    if (BulkIn2 == USBDevice->BulkInEndPt) BulkIn2 = NULL;
}

```

7.6 BulkOutEndPt

Description

BulkOutEndPt is a pointer to an object representing the first BULK OUT endpoint enumerated for the selected interface.

The selected interface might expose additional BULK OUT endpoints. To discern this, one would need to traverse the [EndPoints](#) array, checking the [Attributes](#) and [Address](#) members of each [CCyUSBEndPoint](#) object referenced in the array.

If no BULK OUT endpoints were enumerated by the device, BulkInEndPt will be set to NULL.

Example

```

// Find a second bulk OUT endpoint in the EndPoints[] array
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

CCyBulkEndPoint *BulkOut2 = NULL;

int eptCount = USBDevice->EndPointCount();

for (int i=1; i<eptCount; i++) {
    bool bIn = USBDevice->EndPoints[i]->Address & 0x80;
    bool bBulk = (USBDevice->EndPoints[i]->Attributes == 2);

    if (bBulk && !bIn) BulkOut2 = (CCyBulkEndPoint *) USBDevice->EndPoints[i];
    if (BulkOut2 == USBDevice->BulkOutEndPt) BulkOut2 = NULL;
}

```

7.7 CCyUSBDevice()

Description

This is the constructor for the CCyUSBDevice class.

It registers the window of *hnd* to receive USB Plug and Play messages when devices are connected or

disconnected to/from the driver.

The object created serves as the programming interface to the driver whose GUID is passed in the *guid* parameter.

The constructor initializes the class members and then calls the [Open\(0\)](#) method to open the first device that is attached to the driver.

Parameters

hnd

hnd is a handle to the application's main window (the window whose WndProc function will process USB PnP events).

If you are building a console application or don't want your window to receive PnP events, you may omit the *hnd* parameter.

guid

guid is the GUID defined in the [Strings] section of the CyUsb.inf file (or your own named copy). If this parameter is omitted, *guid* defaults to CYUSBDRV_GUID.

If you don't want to register for PnP events, but you do want to pass your own driver GUID to the constructor, you will need to pass NULL as the *hnd* parameter.

Example 1

```
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    USBDevice = new CCyUSBDevice(Handle);
    CurrentEndPt = USBDevice->ControlEndPt;
}

// Overload MainForm's WndProc method to watch for PnP messages
// Requires #include <dbt.h>
void __fastcall TMainForm::WndProc(TMessage &Message)
{
    if (Message.Msg == WM_DEVICECHANGE) {
        // Tracks DBT_DEVICEARRIVAL followed by DBT_DEVNODES_CHANGED
        if (Message.WParam == DBT_DEVICEARRIVAL) {
            bPnP_Arrival = true;
            bPnP_DevNodeChange = false;
        }

        // Tracks DBT_DEVNODES_CHANGED followed by DBT_DEVICEREMOVECOMPLETE
        if (Message.WParam == DBT_DEVNODES_CHANGED) {
            bPnP_DevNodeChange = true;
            bPnP_Removal = false;
        }

        if (Message.WParam == DBT_DEVICEREMOVECOMPLETE) {
            bPnP_Removal = true;

            PDEV_BROADCAST_HDR bcastHdr = (PDEV_BROADCAST_HDR) Message.LParam;
            if (bcastHdr->dbch_devicetype == DBT_DEVTYP_HANDLE) {
```



```

PDEV_BROADCAST_HANDLE pDev = (PDEV_BROADCAST_HANDLE) Message.LParam;
if (pDev->dbch_handle == USBDevice->DeviceHandle())
    USBDevice->Close();
}
}

// If DBT_DEVNODES_CHANGED followed by DBT_DEVICEREMOVECOMPLETE
if (bPnP_Removal && bPnP_DevNodeChange) {
    Sleep(10);
    DisplayDevices();
    bPnP_Removal = false;
    bPnP_DevNodeChange = false;
}

// If DBT_DEVICEARRIVAL followed by DBT_DEVNODES_CHANGED
if (bPnP_DevNodeChange && bPnP_Arrival) {
    DisplayDevices();
    bPnP_Arrival = false;
    bPnP_DevNodeChange = false;
}
}

TForm::WndProc(Message);
}

```

Example 2

In the CyUSB.inf file :

```
[Strings]
CyUSB.GUID="{BE18AA60-7F6A-11d4-97DD-00010229B959}"
```

In some application source (.cpp) file:

```
GUID guid = StringToGUID("{BE18AA60-7F6A-11d4-97DD-00010229B959}");
CCyUSBDevice *USBDevice = new CCyUSBDevice(NULL, guid); // Does not register for
PnP events
```

7.8 ~CCyUSBDevice()

Description

This is the destructor for the CCyUSBDevice class. It calls the [Close\(\)](#) method in order to properly close any open handle to the driver and to deallocate dynamically allocated members of the class.

7.9 Close()

Description

The Close method closes the handle to the CyUSB driver, if one is open.

Dynamically allocated members of the CCyUSBDevice class are de-allocated. And, all "shortcut" pointers to elements of the [EndPoints](#) array (ControlEndPt, IsoIn/OutEndPt, BulkIn/OutEndPt, InterruptIn/OutEndPt) are reset to NULL.

Close() is called automatically by the `~CCyUSBDevice()` destructor. It is also called automatically by the `Open()` method, if a handle to the driver is already open.

Therefore, it is rare that you would ever need to call Close() explicitly (though doing so would not cause any problems).

7.10 Config()

Description

This method returns the current configuration index for the device.

Most devices only expose a single configuration at one time. So, this method should almost always return zero.

7.11 ConfigAttrib

Description

This data member contains the value of the **bmAttributes** field from the device's current configuration descriptor.

7.12 ConfigCount()

Description

This function returns the number of configurations reported by the device in the **bNumConfigurations** field of its device descriptor.

7.13 ConfigValue

Description

This data member contains the value of the **bConfigurationValue** field from the device's current configuration descriptor.

7.14 ControlEndPoint

Description

ControlEndPoint points to an object representing the primary Control endpoint, endpoint 0.

ControlEndPoint should always be the same value as `EndPoints[0]`.

Before calling the `XferData()` method for ControlEndPoint, you should set the object's control properties.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

// Just for typing efficiency
CCyControlEndPoint *ept = USBDevice->ControlEndPoint;

ept->Target      = TGT_DEVICE;
ept->ReqType     = REQ_VENDOR;
ept->Direction   = DIR_TO_DEVICE;
```

```
ept->ReqCode    = 0x05;
ept->Value      = 1;
ept->Index      = 0;

char buf[512];
ZeroMemory(buf, 512);
LONG buflen = 512;

ept->XferData(buf, buflen);
```

7.15 DevClass

Description

This data member contains the value of the **bDeviceClass** field from the open device's Device Descriptor.

7.16 DeviceCount()

Description

Returns the number of devices attached to the USB driver.

The value returned can be used to discern legitimate parameters for the [Open\(\)](#) method.

Example

```
// Look for a device having VID = 0547, PID = 1002

USBDevice = new CCyUSBDevice(Handle);

int devices = USBDevice->DeviceCount( );

int vID, pID;

int d = 0;
do {
    USBDevice->Open(d); // Open automatically calls Close( ) if necessary
    vID = USBDevice->VendorID;
    pID = USBDevice->ProductID;
    d++;
} while ((d < devices) && (vID != 0x0547) && (pID != 0x1002));
```

7.17 DeviceHandle()

Description

Returns the handle to the driver if the CCyUSBDevice is opened to a connected USB device. If no device is currently open, DeviceHandle() returns INVALID_HANDLE_VALUE.

7.18 DeviceName

Description

DeviceName is an array of characters containing the product string indicated by the device descriptor's iProduct field.

7.19 DevProtocol

Description

This data member contains the value of the **bDeviceProtocol** field from the open device's Device Descriptor.

7.20 DevSubClass

Description

This data member contains the value of the **bDeviceSubClass** field from the open device's Device Descriptor.

7.21 DriverGUID()

Description

Returns the Global Unique Identifier of the USB driver attached to the CCyUSBDevice.

See also: [CCyUSBDevice\(\)](#)

7.22 DriverVersion

Description

DriverVersion contains 4 bytes representing the version of the driver that is attached to the CCyUSBDevice.

7.23 EndPointCount()

Description

Returns the number of endpoints exposed by the currently selected interface (or Alternate Interface) plus 1.

The default Control endpoint (endpoint 0) is included in the count.

Example

```
// Find bulk endpoints in the EndPoints[] array
CCyBulkEndPoint *BulkInEpt = NULL;
CCyBulkEndPoint *BulkOutEpt = NULL;

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);
int eptCount = USBDevice->EndPointCount();

// Skip EndPoints[0], which we know is the control endpoint
for (int i=1; i<eptCount; i++) {
    bool bIn = USBDevice->EndPoints[i]->Address & 0x80;
    bool bBulk = (USBDevice->EndPoints[i]->Attributes == 2);

    if (bBulk && bIn) BulkInEpt = (CCyBulkEndPoint *) USBDevice->EndPoints[i];
    if (bBulk && !bIn) BulkOutEpt = (CCyBulkEndPoint *) USBDevice->EndPoints[i];
}
```

7.24 EndPointOf()

Description

Returns a pointer to the endpoint object in the EndPoints array whose [Address](#) property is equal to **addr**.

Returns NULL If no endpoint with Address = **addr** is found.

Example

```

UCHAR eptAddr = 0x82;
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);
CCyUSBEndPoint *EndPt = USBDevice->EndPointOf(eptAddr);
if (EndPt) EndPt->Reset( );

```

7.25 EndPoints

Description

EndPoints is a list of up to MAX_ENDPTS (16) pointers to endpoint objects.

The objects pointed to represent all the USB endpoints reported for the current USB interface/Alt interface of the device.

EndPoints[0] always contains a pointer to a [CCyControlEndPoint](#) representing the primary Control Endpoint (endpoint 0) of the device.

Unused entries in EndPoints are set to NULL.

Use [EndPointCount\(\)](#) to find-out how many entries in EndPoints are valid.

EndPoints is re-initialized each time [Open\(\)](#) or [SetAltIntfrc\(\)](#) is called.

NOTE:

[CCyUSBEndPoint](#) is an abstract class, having a pure virtual function [BeginDataXfer\(\)](#).

The objects pointed to by EndPoints** are, therefore, actually instances of [CCyControlEndPoint](#), [CCyBulkEndPoint](#), [CCyIsocEndPoint](#) or [CCyInterruptEndPoint](#).

Calling EndPoints[n]->XferData() automatically results in the correct XferData() function being invoked.

Example

```

// Count the bulk-in endpoints

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

int epCnt = USBDevice->EndPointCount();

bool bBulk, bIn;
int blkInCnt = 0;

for (int e=0; e<epCnt; e++) {
    bBulk = USBDevice->EndPoints[e]->Attributes == 2;
    bIn = USBDevice->EndPoints[e]->Address & 0x80;
}

```

```

        if (bBulk && bIn) blkInCnt++;
    }

```

7.26 FriendlyName

Description

FriendlyName is an array of characters containing the device description string for the open device which was provided by the driver's .inf file.

7.27 GetDeviceDescriptor()

Description

This function copies the current device's device descriptor into the memory pointed to by **descr**.

7.28 GetConfigDescriptor()

Description

This function copies the current device's configuration descriptor into the memory pointed to by **descr**.

7.29 GetIntfcDescriptor()

Description

This function copies the currently selected interface descriptor into the memory pointed to by **descr**.

7.30 GetUSBConfig()

Description

This function returns a copy of the [CCyUSBConfig](#) object indicated by **index**.

The **index** parameter must be less than [CCyUSBDevice::ConfigCount\(\)](#).

Example

```

// This code lists all the endpoints reported
//   for all the interfaces reported
//   for all the configurations reported
//   by the device.

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

String s;

for (int c=0; c<USBDevice->ConfigCount(); c++) {
    CCyUSBConfig cfg = USBDevice->GetUSBConfig(c);

    s.printf("bLength: 0x%x",cfg.bLength); EZOutputMemo->Lines->Add(s);
    s.printf("bDescriptorType: %d",cfg.bDescriptorType); EZOutputMemo->Lines-
>Add(s);
    s.printf("wTotalLength: %d (0x%x)",cfg.wTotalLength,cfg.wTotalLength);
EZOutputMemo->Lines->Add(s);
    s.printf("bNumInterfaces: %d",cfg.bNumInterfaces); EZOutputMemo->Lines-
>Add(s);
    s.printf("bConfigurationValue: %d",cfg.bConfigurationValue); EZOutputMemo-

```

```

>Lines->Add(s);
    s.printf("iConfiguration: %d",cfg.iConfiguration); EZOutputMemo->Lines-
>Add(s);
    s.printf("bmAttributes: 0x%x",cfg.bmAttributes); EZOutputMemo->Lines->Add(s);
    s.printf("MaxPower: %d",cfg.MaxPower); EZOutputMemo->Lines->Add(s);
    EZOutputMemo->Lines->Add("*****");

    for (int i=0; i<cfg.AltInterfaces; i++) {
        CCyUSBInterface *ifc = cfg.Interfaces[i];
        EZOutputMemo->Lines->Add("Interface Descriptor:" + String(i+1));
        EZOutputMemo->Lines->Add("-----");
        s.printf("bLength: 0x%x",ifc->bLength); EZOutputMemo->Lines->Add(s);
        s.printf("bDescriptorType: %d",ifc->bDescriptorType); EZOutputMemo->Lines-
>Add(s);
        s.printf("bInterfaceNumber: %d",ifc->bInterfaceNumber); EZOutputMemo-
>Lines->Add(s);
        s.printf("bAlternateSetting: %d",ifc->bAlternateSetting); EZOutputMemo-
>Lines->Add(s);
        s.printf("bNumEndpoints: %d",ifc->bNumEndpoints); EZOutputMemo->Lines-
>Add(s);
        s.printf("bInterfaceClass: %d",ifc->bInterfaceClass); EZOutputMemo->Lines-
>Add(s);

        for (int e=0; e<ifc->bNumEndpoints; e++) {
            CCyUSBEndPoint *ept = ifc->EndPoints[e+1];
            EZOutputMemo->Lines->Add("EndPoint Descriptor: " + String(e+1));
            EZOutputMemo->Lines->Add("-----");
            s.printf("bLength: 0x%x",ept->DscLen); EZOutputMemo->Lines->Add(s);
            s.printf("bDescriptorType: %d",ept->DscType); EZOutputMemo->Lines-
>Add(s);
            s.printf("bEndpointAddress: 0x%x",ept->Address); EZOutputMemo->Lines-
>Add(s);
            s.printf("bmAttributes: 0x%x",ept->Attributes); EZOutputMemo->Lines-
>Add(s);
            s.printf("wMaxPacketSize: %d",ept->MaxPktSize); EZOutputMemo->Lines-
>Add(s);
            s.printf("bInterval: %d",ept->Interval); EZOutputMemo->Lines->Add(s);
            EZOutputMemo->Lines->Add("*****");
        }
    }
}

```

7.31 Interface()

Description

Interface returns the index of the currently selected device interface.

Because Windows always represents different reported interfaces as separate devices, the CyUSB driver is only shown devices that have a single interface. This causes the Interface() method to always return zero.

7.32 InterruptInEndPt

Description

InterruptInEndPt is a pointer to an object representing the first INTERRUPT IN endpoint enumerated for the selected interface.

The selected interface might expose additional INTERRUPT IN endpoints. To discern this, one would need to traverse the [EndPoints](#) array, checking the [Attributes](#) and [Address](#) members of each [CCyUSBEndPoint](#) object referenced in the array.

If no INTERRUPT IN endpoints were enumerated by the device, InterruptInEndPt will be set to NULL.

Example

```
// Find a second Interrupt IN endpoint in the EndPoints[] array
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

CCyInterruptEndPoint *IntIn2 = NULL;
int eptCount = USBDevice->EndPointCount();

for (int i=1; i<eptCount; i++) {
    bool bIn = USBDevice->EndPoints[i]->Address & 0x80;
    bool bInt = (USBDevice->EndPoints[i]->Attributes == 3);

    if (bInt && bIn) IntIn2 = (CCyInterruptEndPoint *) USBDevice->EndPoints[i];
    if (IntIn2 == USBDevice->InterruptInEndPt) IntIn2 = NULL;
}
```

7.33 InterruptOutEndPt

Description

InterruptOutEndPt is a pointer to an object representing the first INTERRUPT OUT endpoint enumerated for the selected interface.

The selected interface might expose additional INTERRUPT OUT endpoints. To discern this, one would need to traverse the [EndPoints](#) array, checking the [Attributes](#) and [Address](#) members of each [CCyUSBEndPoint](#) object referenced in the array.

If no INTERRUPT OUT endpoints were enumerated by the device, InterruptOutEndPt will be set to NULL.

Example

```
// Find a second Interrupt OUT endpoint in the EndPoints[] array
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

CCyInterruptEndPoint *IntOut2 = NULL;
int eptCount = USBDevice->EndPointCount();

for (int i=1; i<eptCount; i++) {
    bool bIn = USBDevice->EndPoints[i]->Address & 0x80;
    bool bInt = (USBDevice->EndPoints[i]->Attributes == 3);

    if (bInt && !bIn) IntOut2 = (CCyInterruptEndPoint *) USBDevice->EndPoints[i];
    if (IntOut2 == USBDevice->InterruptInEndPt) IntOut2 = NULL;
}
```

7.34 IntfcClass

Description

This data member contains the bInterfaceClass field from the currently selected interface's interface descriptor.

7.35 IntfcCount()

Description

Returns the `bNumInterfaces` field of the current device's configuration descriptor.

This number does not include alternate interfaces that might be part of the current configuration.

Because Windows always represents different reported interfaces as separate devices, the CyUSB driver is only shown devices that have a single interface. This causes the `IntfcCount()` method to always return 1.

7.36 IntfcProtocol

Description

This data member contains the `bInterfaceProtocol` field from the currently selected interface's interface descriptor.

7.37 IntfcSubClass

Description

This data member contains the `bInterfaceSubClass` field from the currently selected interface's interface descriptor.

7.38 IsocInEndPt

Description

`IsocInEndPt` is a pointer to an object representing the first ISOCHRONOUS IN endpoint enumerated for the selected interface.

The selected interface might expose additional ISOCHRONOUS OUT endpoints. To discern this, one would need to traverse the `EndPoints` array, checking the `Attributes` and `Address` members of each `CCyUSBEndPoint` object referenced in the array.

If no ISOCHRONOUS IN endpoints were enumerated by the device, `IsocInEndPt` will be set to `NULL`.

Example

```
// Find a second Isoc IN endpoint in the EndPoints[] array
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);
CCyIsocEndPoint *IsocIn2 = NULL;

int eptCount = USBDevice->EndPointCount();

for (int i=1; i<eptCount; i++) {
    bool bIn = USBDevice->EndPoints[i]->Address & 0x80;
    bool bIsoc = (USBDevice->EndPoints[i]->Attributes == 1);

    if (bIsoc && bIn) IsocIn2 = (CCyIsocEndPoint *) USBDevice->EndPoints[i];
    if (IsocIn2 == USBDevice->IsocInEndPt) IsocIn2 = NULL;
}
```

7.39 IsocOutEndPt

Description

IsocOutEndPt is a pointer to an object representing the first ISOCHRONOUS OUT endpoint enumerated for the selected interface.

The selected interface might expose additional ISOCHRONOUS OUT endpoints. To discern this, one would need to traverse the [EndPoints](#) array, checking the [Attributes](#) and [Address](#) members of each [CCyUSBEndPoint](#) object referenced in the array.

If no ISOCHRONOUS OUT endpoints were enumerated by the device, IsocOutEndPt will be set to NULL.

Example

```
// Find a second Isoc OUT endpoint in the EndPoints[] array
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

CCyIsocEndPoint *IsocOut2 = NULL;
int eptCount = USBDevice->EndPointCount();

for (int i=1; i<eptCount; i++) {
    bool bIn = USBDevice->EndPoints[i]->Address & 0x80;
    bool bIsoc = (USBDevice->EndPoints[i]->Attributes == 1);

    if (bIsoc && !bIn) IsocOut2 = (CCyIsocEndPoint *) USBDevice->EndPoints[i];
    if (IsocOut2 == USBDevice->IsocOutEndPt) IsocOut2 = NULL;
}
```

7.40 IsOpen()

Description

IsOpen() returns **true** if CCyUSBDevice object has a valid handle to a device attached to the CyUSB driver.

When IsOpen() is **true**, the CCyUSBDevice object is ready to perform IO operations via its [EndPoints](#) members.

7.41 Manufacturer

Description

Manufacturer is an array of wide characters containing the manufacturer string indicated by the device descriptor's **iManufacturer** field.

7.42 MaxPacketSize

Description

This data member contains the value of the **bMaxPacketSize0** field from the open device's Device Descriptor structure.

7.43 MaxPower

Description

This data member contains the value of the **MaxPower** field of the open device's selected configuration descriptor.

7.44 NtStatus

Description

The NtStatus member contains the NTSTATUS returned by the driver for the most recent call to a non-endpoint IO method (SetAltIntfc, Open, Reset, etc.)

More often, you will want to access the [NtStatus](#) member of the CCyUSBEndPoint objects.

7.45 Open()

Description

The Open() method is one of the main workhorses of the library.

When Open() is called, it first checks to see if the CCyUSBDevice object is already opened to one of the attached devices. If so, it calls [Close\(\)](#), then proceeds.

Open() calls [DeviceCount\(\)](#) to determine how many devices are attached to the USB driver.

Open() creates a valid handle to the device driver, through which all future access is accomplished by the library methods.

Open() calls the driver to gather the device, interface, endpoint and string descriptors.

Open() results in the [EndPoints](#) array getting properly initialized to pointers of the default interface's endpoints.

Open() initializes the [ControlEndPt](#) member to point to an instance of [CCyControlEndPoint](#) that represents the device's endpoint zero.

Open() initializes the [BulkInEndPt](#) member to point to an instance of CCyBulkEndPoint representing the first Bulk-IN endpoint that was found. Similarly, the [BulkOutEndPt](#), [InterruptInEndPt](#), [InterruptOutEndPt](#), [IsocInEndPt](#) and [IsocOutEndPt](#) members are initialized to point to instances of their respective endpoint classes if such endpoints were found.

After Open() returns **true**, all the properties and methods of CCyUSBDevice are legitimate.

Open() returns **false** if it is unsuccessful in accomplishing the above activities. However, if Open() was able to obtain a valid handle to the driver, the handle will remain valid even after Open() returns **false**. (When open fails, it does not automatically call [Close\(\)](#).) This allows the programmer to call the [Reset\(\)](#) or [ReConnect\(\)](#) methods and then call Open() again. Sometimes this will allow a device to open properly.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);  
  
// Attempt to open device #0
```

```
if (USBDevice->DeviceCount() && !USBDevice->Open(0)) {
    USBDevice->Reset();
    USBDevice->Open(0);
}

if (! USBDevice->IsOpen()) return false;
```

7.46 PowerState()

This method returns the current power state of the device.

A return value of 1 indicates a power state of D0 (Device fully on).

A return value of 4 indicates a power state of D3 (Device fully asleep).

7.47 Product

Description

Product is an array of wide characters containing the product string indicated by the device descriptor's **iProduct** field.

7.48 ProductID

Description

This data member contains the value of **idProduct** from the open device's Device Descriptor structure.

Example

```
// Look for a device having VID = 0547, PID = 1002

USBDevice = new CCyUSBDevice(Handle); // Create an instance of CCyUSBDevice
int devices = USBDevice->DeviceCount();
int vID, pID;
int d = 0;

do {
    USBDevice->Open(d); // Open automatically calls Close() if necessary
    vID = USBDevice->VendorID;
    pID = USBDevice->ProductID;
    d++;
} while ((d < devices) && (vID != 0x0547) && (pID != 0x1002));
```

7.49 ReConnect()

Description

ReConnect() calls the USB device driver to cause the currently open USB device to be logically disconnected from the USB bus and re-enumerated.

7.50 Reset()

Description

Reset() calls the USB device driver to cause the currently open USB device to be reset.

This call causes the device to return to its initial power-on configuration.

7.51 Resume()

The Resume() method sets the device power state to D0 (Full on).

The method returns true if successful and false if the command failed.

7.52 SerialNumber

Description

SerialNumber is an array of wide characters containing the serial number string indicated by the device descriptor's **iSerialNumber** field.

7.53 SetConfig()

Description

This method will set the current device configuration to **cfg**, if **cfg** represents an existing configuration.

In practice, devices only expose a single configuration. So, while this method exists for completeness, it should probably never be invoked with a **cfg** value other than 0.

7.54 SetAltIntfc()

Description

SetAltIntfc() calls the driver to set the active interface of the device to **alt** .

If **alt** is not a valid alt interface setting, the method does nothing.

Legitimate values for **alt** are 0 to [AltIntfcCount\(\)](#).

Calling SetAltIntfc() causes all the [EndPoints](#) members of CCyUSBDevice to be re-assigned to objects reflecting the endpoints of the new alternate interface.

Returns **true** if the alternate interface was successfully set to **alt** .

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

int lastIntfc = USBDevice->AltIntfcCount( );

// Select the last Alternate Interface
USBDevice->SetAltIntfc(lastIntfc);
```

7.55 StrLangID

Description

This data member contains the value of **bString** field from the open device's first String Descriptor.

This value indicates the language of the other string descriptors.

If multiple languages are supported in the string descriptors and English is one of the supported languages, StrLangID is set to the value for English (0x0409).

7.56 Suspend()

The Suspend() method sets the device power state to D3 (Full asleep).

The method returns true if successful and false if the command failed.

7.57 USBAddress

Description

USBAddress contains the bus address of the currently open USB device.

This is the address value used by the Windows USBDI stack. It is not particularly useful at the application level.

7.58 USBDIVersion

Description

This data member contains the version of the USB Host Controller Driver in BCD format.

7.59 UsbdStatus

Description

The UsbdStatus member contains the USBDI_STATUS returned by the driver for the most recent call to a non-endpoint IO method (SetAltIntfc, Open, Reset, etc.)

More often, you will want to access the [UsbdStatus](#) member of the CCyUSBEndPoint objects.

7.60 UsbdStatusString()

Description

The UsbdStatusString method returns a string of characters in **s** that represents the UsbdStatus error code contained in **stat**.

The **stat** parameter should be the [UsbdStatus](#) member or a CCyUSBEndPoint::UsbdStatus member.

The format of the returned string, **s**, is:

```
"[state=SSSSSS status=TTTTTTTT]"
```

where SSSSSS can be "SUCCESS", "PENDING", "STALLED", or "ERROR".

Note:

There is no endpoint equivalent for this method. To interpret the UsbdStatus member of an endpoint object, call this method (CCyUSBDevice::UsbdStatusString) passing the UsbdStatus member of the endpoint.

7.61 VendorID

Description

This data member contains the value of `idVendor` from the open device's Device Descriptor structure.

Example

```
// Look for a device having VID = 0547, PID = 1002

USBDevice = new CCyUSBDevice(Handle); // Create an instance of CCyUSBDevice

int devices = USBDevice->DeviceCount();
int vid, pid;

int d = 0;
do {
    USBDevice->Open(d); // Open automatically calls Close() if necessary
    vid = USBDevice->VendorID;
    pid = USBDevice->ProductID;
    d++;
} while ((d < devices) && (vid != 0x0547) && (pid != 0x1002));
```

8 CCyUSBConfig

Header

CyUSB.h

Description

CCyUSBConfig represents a USB device configuration. Such configurations have one or more interfaces each of which exposes one or more endpoints.

When [CCyUSBDevice::Open\(\)](#) is called, an instance of CCyUSBConfig is constructed for each configuration reported by the open device's device descriptor. (Normally, there is just one.)

In the process of construction, CCyUSBConfig creates instances of [CCyUSBInterface](#) for each interface exposed in the device's configuration descriptor. In turn, the CCyUSBInterface class creates instances of [CyUSBEndPoint](#) for each endpoint descriptor contained in the interface descriptor. In this iterative fashion, the entire structure of `Configs->Interfaces->EndPoints` gets populated from a single construction of the CCyUSBConfig class.

The following example code shows how you might use the CCyUSBConfig class in an application.

Example

```
// This code lists all the endpoints reported
// for all the interfaces reported
// for all the configurations reported
// by the device.

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

String s;

for (int c=0; c<USBDevice->ConfigCount(); c++) {
```

```

CCyUSBConfig cfg = USBDevice->GetUSBConfig(c);

s.printf("bLength: 0x%x",cfg.bLength); EZOutputMemo->Lines->Add(s);
s.printf("bDescriptorType: %d",cfg.bDescriptorType); EZOutputMemo->Lines-
>Add(s);
s.printf("wTotalLength: %d (0x%x)",cfg.wTotalLength,cfg.wTotalLength);
EZOutputMemo->Lines->Add(s);
s.printf("bNumInterfaces: %d",cfg.bNumInterfaces); EZOutputMemo->Lines-
>Add(s);
s.printf("bConfigurationValue: %d",cfg.bConfigurationValue); EZOutputMemo-
>Lines->Add(s);
s.printf("iConfiguration: %d",cfg.iConfiguration); EZOutputMemo->Lines-
>Add(s);
s.printf("bmAttributes: 0x%x",cfg.bmAttributes); EZOutputMemo->Lines->Add(s);
s.printf("MaxPower: %d",cfg.MaxPower); EZOutputMemo->Lines->Add(s);
EZOutputMemo->Lines->Add("*****");

for (int i=0; i<cfg.AltInterfaces; i++) {
    CCyUSBInterface *ifc = cfg.Interfaces[i];
    EZOutputMemo->Lines->Add("Interface Descriptor:" + String(i+1));
    EZOutputMemo->Lines->Add("-----");
    s.printf("bLength: 0x%x",ifc->bLength); EZOutputMemo->Lines->Add(s);
    s.printf("bDescriptorType: %d",ifc->bDescriptorType); EZOutputMemo->Lines-
>Add(s);
    s.printf("bInterfaceNumber: %d",ifc->bInterfaceNumber); EZOutputMemo-
>Lines->Add(s);
    s.printf("bAlternateSetting: %d",ifc->bAlternateSetting); EZOutputMemo-
>Lines->Add(s);
    s.printf("bNumEndpoints: %d",ifc->bNumEndpoints); EZOutputMemo->Lines-
>Add(s);
    s.printf("bInterfaceClass: %d",ifc->bInterfaceClass); EZOutputMemo->Lines-
>Add(s);

    for (int e=0; e<ifc->bNumEndpoints; e++) {
        CCyUSBEndPoint *ept = ifc->EndPoints[e+1];
        EZOutputMemo->Lines->Add("EndPoint Descriptor: " + String(e+1));
        EZOutputMemo->Lines->Add("-----");
        s.printf("bLength: 0x%x",ept->DscLen); EZOutputMemo->Lines->Add(s);
        s.printf("bDescriptorType: %d",ept->DscType); EZOutputMemo->Lines-
>Add(s);
        s.printf("bEndpointAddress: 0x%x",ept->Address); EZOutputMemo->Lines-
>Add(s);
        s.printf("bmAttributes: 0x%x",ept->Attributes); EZOutputMemo->Lines-
>Add(s);
        s.printf("wMaxPacketSize: %d",ept->MaxPktSize); EZOutputMemo->Lines-
>Add(s);
        s.printf("bInterval: %d",ept->Interval); EZOutputMemo->Lines->Add(s);
        EZOutputMemo->Lines->Add("*****");
    }
}
}

```

8.1 AltInterfaces

Description

AltInterfaces contains the total number of interfaces exposed by the configuration (including the default interface). This value is the number of interface descriptors contained in the current configuration descriptor.

Because the [CCyUSBDevice::AltIntfcCount\(\)](#) method does not count the primary interface, it returns

CCyUSBConfig::AltInterfaces - 1.

8.2 bConfigurationValue

Description

bConfigurationValue contains value of the **bConfigurationValue** field from the selected configuration descriptor.

8.3 bDescriptorType

Description

bDescriptorType contains value of the **bDescriptorType** field from the selected configuration descriptor.

8.4 bLength

Description

bLength contains value of the **bLength** field from the selected configuration descriptor.

8.5 bmAttributes

Description

bmAttributes contains value of the **bmAttributes** field from the selected configuration descriptor.

8.6 bNumInterfaces

Description

bNumInterfaces contains value of the **bNumInterfaces** field from the selected configuration descriptor.

8.7 CCyUSBConfig()

Description

This is the default constructor for the CCyUSBConfig class.

This constructor simply sets all it's data members to zero.

8.8 CCyUSBConfig()

Description

This constructor creates a functional CCyUSBConfig object, complete with a populated Interfaces[] array.

During construction, the pConfigDescr structure is traversed and all interface descriptors are read, creating CCyUSBInterface objects.

This constructor is called automatically as part of the [CCyUSBDevice::Open\(\)](#) method. You should never need to call this constructor yourself.

8.9 CCyUSBConfig()

Description

This is the *copy* constructor for the CCyUSBConfig class.

This constructor copies all of the simple data members of **cfg**. Then, it walks through **cfg**'s list of [CCyUSBInterface](#) objects and makes copies of them, storing pointers to the new interface objects in a private, internal data array. (This is accomplished by calling the [copy constructor](#) for CCyUSBInterface.)

You should usually not call the copy constructor explicitly. Instead, use the [GetUSBConfig\(\)](#) method of the CCyUSBDevice class.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);
CCyUSBConfig cfg = USBDevice->GetUSBConfig(0);
```

8.10 ~CCyUSBConfig

Description

This is the destructor for the CCyUSBConfig class.

The destructor deletes all the dynamically constructed [CCyUSBInterface](#) objects that were created during construction of the object.

8.11 iConfiguration

Description

iConfiguration contains value of the **iConfiguration** field from the selected configuration descriptor.

8.12 Interfaces

Description

Interfaces is an array of pointers to [CCyUSBInterface](#) objects. One valid pointer exists in Interfaces[] for each alternate interface exposed by the configuration (including alt setting 0).

The [AltInterfaces](#) member tells how many valid entries are held in Interfaces.

Use [CCyUSBDevice::AltIntfcCount\(\)](#) and [CCyUSBDevice::SetAltIntfc\(\)](#) to access a configuration's alternate interfaces.

Example

```
// This code lists all the endpoints reported
// for all the interfaces reported
// for all the configurations reported
// by the device.
```

```

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

String s;

for (int c=0; c<USBDevice->ConfigCount(); c++) {
    CCyUSBConfig cfg = USBDevice->GetUSBConfig(c);

    s.printf("bLength: 0x%x",cfg.bLength); EZOutputMemo->Lines->Add(s);
    s.printf("bDescriptorType: %d",cfg.bDescriptorType); EZOutputMemo->Lines-
>Add(s);
    s.printf("wTotalLength: %d (0x%x)",cfg.wTotalLength,cfg.wTotalLength);
EZOutputMemo->Lines->Add(s);
    s.printf("bNumInterfaces: %d",cfg.bNumInterfaces); EZOutputMemo->Lines-
>Add(s);
    s.printf("bConfigurationValue: %d",cfg.bConfigurationValue); EZOutputMemo-
>Lines->Add(s);
    s.printf("iConfiguration: %d",cfg.iConfiguration); EZOutputMemo->Lines-
>Add(s);
    s.printf("bmAttributes: 0x%x",cfg.bmAttributes); EZOutputMemo->Lines->Add(s);
    s.printf("MaxPower: %d",cfg.MaxPower); EZOutputMemo->Lines->Add(s);
    EZOutputMemo->Lines->Add("*****");

    for (int i=0; i<cfg.AltInterfaces; i++) {
        CCyUSBInterface *ifc = cfg.Interfaces[i];
        EZOutputMemo->Lines->Add("Interface Descriptor:" + String(i+1));
        EZOutputMemo->Lines->Add("-----");
        s.printf("bLength: 0x%x",ifc->bLength); EZOutputMemo->Lines->Add(s);
        s.printf("bDescriptorType: %d",ifc->bDescriptorType); EZOutputMemo->Lines-
>Add(s);
        s.printf("bInterfaceNumber: %d",ifc->bInterfaceNumber); EZOutputMemo-
>Lines->Add(s);
        s.printf("bAlternateSetting: %d",ifc->bAlternateSetting); EZOutputMemo-
>Lines->Add(s);
        s.printf("bNumEndpoints: %d",ifc->bNumEndpoints); EZOutputMemo->Lines-
>Add(s);
        s.printf("bInterfaceClass: %d",ifc->bInterfaceClass); EZOutputMemo->Lines-
>Add(s);

        for (int e=0; e<ifc->bNumEndpoints; e++) {
            CCyUSBEndPoint *ept = ifc->EndPoints[e+1];
            EZOutputMemo->Lines->Add("EndPoint Descriptor: " + String(e+1));
            EZOutputMemo->Lines->Add("-----");
            s.printf("bLength: 0x%x",ept->DscLen); EZOutputMemo->Lines->Add(s);
            s.printf("bDescriptorType: %d",ept->DscType); EZOutputMemo->Lines-
>Add(s);
            s.printf("bEndpointAddress: 0x%x",ept->Address); EZOutputMemo->Lines-
>Add(s);
            s.printf("bmAttributes: 0x%x",ept->Attributes); EZOutputMemo->Lines-
>Add(s);
            s.printf("wMaxPacketSize: %d",ept->MaxPktSize); EZOutputMemo->Lines-
>Add(s);
            s.printf("bInterval: %d",ept->Interval); EZOutputMemo->Lines->Add(s);
            EZOutputMemo->Lines->Add("*****");
        }
    }
}

```

8.13 wTotalLength

Description

wTotalLength contains value of the **wTotalLength** field from the selected configuration descriptor.

9 CCyUSBEndPoint

Header

CyUSB.h

Description

CCyUSBEndPoint is an abstract class, having a pure virtual method, [BeginDataXfer\(\)](#). Therefore, instances of CCyUSBEndPoint cannot be constructed. [CCyControlEndPoint](#), [CCyBulkEndPoint](#), [CCyIsocEndPoint](#), and [CCyInterruptEndPoint](#) are all classes derived from CCyUSBEndPoint.

All USB data traffic is accomplished by using instances of the endpoint classes.

When a CCyUSBDevice is opened, a list of all the [EndPoints](#) for the current alt interface is generated. This list is populated with viable CCyUSBEndPoint objects, instantiated for the appropriate type of endpoint. Data access is then accomplished via one of these CCyUSBEndPoint objects.

9.1 Abort()

Description

Abort sends an IOCTL_ADAPT_ABORT_PIPE command to the USB device, with the endpoint address as a parameter. This causes an abort of pending IO transactions on the endpoint.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);
USBDevice->ControlEndPt->Abort();
```

9.2 Address

Description

Address contains the value of the **bEndpointAddress** field of the endpoint descriptor returned by the device.

Addresses with the high-order bit set (0x8_) are IN endpoints.

Addresses with the high-order bit cleared (0x0_) are OUT endpoints.

The default control endpoint ([ControlEndPt](#)) has Address = 0.

Example

```
// Find a second bulk IN endpoint in the EndPoints[] array
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);
CCyBulkEndPoint *BulkIn2 = NULL;

int eptCount = USBDevice->EndPointCount();
```

```

for (int i=1; i<eptCount; i++) {
    bool bIn = USBDevice->EndPoints[i]->Address & 0x80;
    bool bBulk = (USBDevice->EndPoints[i]->Attributes == 2);

    if (bBulk && bIn) BulkIn2 = (CCyBulkEndPoint *) USBDevice->EndPoints[i];
    if (BulkIn2 == BulkInEndPt) BulkIn2 = NULL;
}

```

9.3 Attributes

Description

Attributes contains the value of the **bmAttributes** field of the endpoint's descriptor.

The **Attributes** member indicates the type of endpoint per the following list.

- 0: Control
- 1: Isochronous
- 2: Bulk
- 3: Interrupt

Example

```

// Find a second bulk IN endpoint in the EndPoints[] array
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);
CCyBulkEndPoint *BulkIn2 = NULL;

int eptCount = USBDevice->EndPointCount();

for (int i=1; i<eptCount; i++) {
    bool bIn = USBDevice->EndPoints[i]->bIn;
    bool bBulk = (USBDevice->EndPoints[i]->Attributes == 2);

    if (bBulk && bIn) BulkIn2 = (CCyBulkEndPoint *) USBDevice->EndPoints[i];
    if (BulkIn2 == BulkInEndPt) BulkIn2 = NULL;
}

```

9.4 BeginDataXfer()

Description

Note that the **CCyUSBEndPoint** version of this method is a pure virtual function. There is no implementation body for this function in the **CCyUSBEndPoint** class. Rather, all the classes derived from **CCyUSBEndPoint** provide their own special implementation of this method.

BeginDataXfer is an advanced method for performing asynchronous IO. This method sets-up all the parameters for a data transfer, initiates the transfer, and immediately returns, not waiting for the transfer to complete.

BeginDataXfer allocates a complex data structure and returns a pointer to that structure. **FinishDataXfer** de-allocates the structure. Therefore, it is imperative that each **BeginDataXfer** call have exactly one matching **FinishDataXfer** call.

You will usually want to use the synchronous [XferData](#) method rather than the asynchronous BeginDataXfer/WaitForXfer/FinishDataXfer approach.

Example

```
// This example assumes that the device automatically sends back,
// over its bulk-IN endpoint, any bytes that were received over its
// bulk-OUT endpoint (commonly referred to as a loopback function)

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

OVERLAPPED outOvLap, inOvLap;
outOvLap.hEvent = CreateEvent(NULL, false, false, "CYUSB_OUT");
inOvLap.hEvent = CreateEvent(NULL, false, false, "CYUSB_IN");

char inBuf[128];
ZeroMemory(inBuf, 128);

char buffer[128];
LONG length = 128;

// Just to be cute, request the return data before initiating the loopback
UCHAR *inContext = USBDevice->BulkInEndPt->BeginDataXfer(inBuf, length,
&inOvLap);
UCHAR *outContext = USBDevice->BulkOutEndPt->BeginDataXfer(buffer, length,
&outOvLap);

USBDevice->BulkOutEndPt->WaitForXfer(&outOvLap,100);
USBDevice->BulkInEndPt->WaitForXfer(&inOvLap,100);

USBDevice->BulkOutEndPt->FinishDataXfer(buffer, length, &outOvLap,outContext);
USBDevice->BulkInEndPt->FinishDataXfer(inBuf, length, &inOvLap,inContext);

CloseHandle(outOvLap.hEvent);
CloseHandle(inOvLap.hEvent);
```

9.5 bIn

Description

bIn indicates whether or not the endpoint is an IN endpoint.

IN endpoints transfer data from the USB device to the Host (PC).

Endpoint addresses with the high-order bit set (0x8_) are IN endpoints. Endpoint addresses with the high-order bit cleared (0x0_) are OUT endpoints.

bIn is not valid for [CCyControlEndPoint](#) objects (such as CCyUSBDevice->ControlEndPt).

Example

```
// Find a second bulk IN endpoint in the EndPoints[] array

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);
```

```
CCyBulkEndPoint *BulkIn2 = NULL;

int eptCount = USBDevice->EndPointCount();

for (int i=1; i<eptCount; i++) {
    bool In = USBDevice->Endpoints[i]->bIn;
    bool bBulk = (USBDevice->Endpoints[i]->Attributes == 2);

    if (bBulk && In) BulkIn2 = (CCyBulkEndPoint *) USBDevice->Endpoints[i];
    if (BulkIn2 == USBDevice->BulkInEndPt) BulkIn2 = NULL;
}
```

9.6 CCyUSBEndPoint()

Description

This is the default constructor for the CCyUSBEndPoint class.

Because CCyUSBEndPoint is an abstract class, you cannot instantiate an object of CCyUSBEndPoint. That is, the statement

```
new CCyUSBEndPoint( );
```

would result in a compiler error.

The default constructor initializes most of its data members to zero. It sets the default endpoint Timeout to 10 seconds. It sets bIn to false, and sets hDevice to INVALID_HANDLE_VALUE.

9.7 CCyUSBEndPoint()

Description

This is the primary constructor for the CCyUSBEndPoint class.

Because CCyUSBEndPoint is an abstract class, you cannot instantiate an object of CCyUSBEndPoint. That is, the statement

```
new CCyUSBEndPoint(h, pEndPointDesc);
```

would result in a compiler error.

However, the constructor does get called (automatically) in the process of constructing derived endpoint classes.

This constructor sets most of its data members to their corresponding fields in the **pEndPointDescriptor** structure. It sets the default endpoint Timeout to 10 seconds. It sets its hDevice member to **h**.

9.8 CCyUSBEndPoint()

Description

This is the *copy* constructor for the CCyUSBEndPoint class.

This constructor copies all of the simple data members of ept.

Because `CCyUSBEndPoint` is an abstract class, you cannot invoke this constructor explicitly. Instead, it gets called as a side effect of invoking the copy constructors for [CCyControlEndPoint](#), [CCyBulkEndPoint](#), [CCyIsocEndPoint](#), and [CCyInterruptEndPoint](#).

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);  
  
CCyControlEndPoint *ctlEpt = new CCyControlEndPoint(*USBDevice->ControlEndPt);
```

9.9 DscLen

Description

DscLen contains the length of the endpoint descriptor as reported in the **bLength** field of the `USB_ENDPOINT_DESCRIPTOR` structure that was passed to the endpoint object's constructor. (Because the passed descriptor was an endpoint descriptor, this value should always be 0x07.)

This data member exists for completeness and debugging purposes. You should normally never need to access this data member.

9.10 DscType

Description

DscType contains the type of the endpoint descriptor as reported in the **bDescriptorType** field of the `USB_ENDPOINT_DESCRIPTOR` structure that was passed to the endpoint object's constructor. (Because the passed descriptor was an endpoint descriptor, this value should always be 0x05.)

This data member exists for completeness and debugging purposes. You should normally never need to access this data member.

9.11 GetXferSize()

Description

Each non-control endpoint has a transfer size that is some multiple of its `MaxPacketSize`. This transfer size can be adjusted programmatically.

The transfer size establishes the size of internal buffers used by the USB driver stack for performing data transfers. Larger values for the transfer size enable data transfers involving fewer transactions. However, those larger buffers also consume more available memory.

GetXferSize() returns the current transfer size setting for the endpoint.

9.12 FinishDataXfer()

Description

FinishDataXfer is an advanced method for performing asynchronous IO.

FinishDataXfer transfers any received bytes into **buf**. It sets the **len** parameter to the actual number of bytes transferred. Finally, **FinishDataXfer** frees the memory associated with the **pXmitBuf** pointer. This pointer was returned by a previous corresponding call to [BeginDataXfer](#).

The pointer to an OVERLAPPED structure, passed in the **ov** parameter, should be the same one that was passed to the corresponding `BeginDataXfer` method.

The **pktInfos** parameter is optional and points to an array of [CCyIsoPktInfo](#) objects. It should only be used for Isochronous endpoint transfers.

You will usually want to use the synchronous [XferData](#) method rather than the asynchronous `BeginDataXfer/WaitForXfer/FinishDataXfer` approach.

Example

```
// This example assumes that the device automatically sends back,
// over its bulk-IN endpoint, any bytes that were received over its
// bulk-OUT endpoint (commonly referred to as a loopback function)

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

OVERLAPPED outOvLap, inOvLap;
outOvLap.hEvent = CreateEvent(NULL, false, false, "CYUSB_OUT");
inOvLap.hEvent = CreateEvent(NULL, false, false, "CYUSB_IN");

char inBuf[128];
ZeroMemory(inBuf, 128);

char buffer[128];
LONG length = 128;

// Just to be cute, request the return data before initiating the loopback
UCHAR *inContext = USBDevice->BulkInEndPt->BeginDataXfer(inBuf, length,
&inOvLap);
UCHAR *outContext = USBDevice->BulkOutEndPt->BeginDataXfer(buffer, length,
&outOvLap);

USBDevice->BulkOutEndPt->WaitForXfer(&outOvLap,100);
USBDevice->BulkInEndPt->WaitForXfer(&inOvLap,100);

USBDevice->BulkOutEndPt->FinishDataXfer(buffer, length, &outOvLap,outContext);
USBDevice->BulkInEndPt->FinishDataXfer(inBuf, length, &inOvLap,inContext);

CloseHandle(outOvLap.hEvent);
CloseHandle(inOvLap.hEvent);
```

9.13 hDevice

Description

hDevice contains a handle to the USB device driver, through which all the IO is carried-out. The handle is created by the [Open\(\)](#) method of a [CCyUSBDevice](#) object.

The only reason to access this data member would be to call the device driver explicitly, bypassing the API library methods. *This is not recommended.*

You should never call `CloseHandle(hDevice)` directly. Instead, call the [Close\(\)](#) method of a [CCyUSBDevice](#) object.

Note that an instance of `CCyUSBDevice` will contain several [CCyUSBEndPoint](#) objects. Each of those will have the same value for their `hDevice` member. Also, the endpoint's `hDevice` member will be identical to its container `CCyUSBDevice` object's private `hDevice` member (accessed via the [DeviceHandle\(\)](#) method).

9.14 Interval

Description

Interval contains the value reported in the **`bInterval`** field of the `USB_ENDPOINT_DESCRIPTOR` structure that was passed to the endpoint object's constructor.

This data member exists for completeness and debugging purposes. You should normally never need to access this data member.

9.15 MaxPktSize

Description

MaxPktSize contains the value indicated by the **`wMaxPacketSize`** field of the `USB_ENDPOINT_DESCRIPTOR` structure that was passed to the endpoint object's constructor.

MaxPktSize is calculated by multiplying the low-order 11 bits of **`wMaxPacketSize`** by the value represented by `1 +` the next 2 bits (bits 11 and 12).

NOTE: For ISOC transfers, the buffer length and the endpoint's transfers size (see [SetXferSize](#)) must be a multiple of 8 times the endpoint's [MaxPktSize](#).

Example

If `wMaxPacketSize` is `0x1400` (binary = `0001 0100 0000 0000`)

$\text{MaxPktSize} = [100\ 0000\ 0000\ \text{binary}] * [10\ \text{binary} + 1] = 1024 * 3 = 3072$

9.16 NtStatus

Description

NtStatus member contains the error code returned from the last call to the `XferData` or `BeginDataXfer` methods.

9.17 Reset()

Description

The `Reset` method resets the endpoint, clearing any error or stall conditions on that endpoint.

Pending data transfers are not cancelled by the `Reset` method.

Call [Abort\(\)](#) for the endpoint in order force completion of any transfers in-process.

9.18 SetXferSize()

Description

Each non-control endpoint has a transfer size that is some multiple of its [MaxPktSize](#). This transfer

size can be adjusted programmatically.

The transfer size establishes the size of internal buffers used by the USB driver stack for performing data transfers. Larger values for the transfer size enable data transfers involving fewer transactions. However, those larger buffers also consume more available memory.

SetXferSize() sets the current transfer size setting for the endpoint. It automatically sets the transfer size to a multiple of the endpoint's [MaxPktSize](#) property that is greater or equal to the requested **xfer** size.

NOTE: For ISOC transfers, the buffer length and the endpoint's transfer size (see [SetXferSize](#)) must be a multiple of 8 times the endpoint's [MaxPktSize](#).

9.19 TimeOut

Description

TimeOut limits the length of time that a [XferData\(\)](#) call will wait for the transfer to complete.

The units of **TimeOut** is milliseconds.

NOTE : For [CCyControlEndPoint](#), the **TimeOut** is rounded down to the nearest 1000 ms, except for values between 0 and 1000 which are rounded up to 1000. Also, for [CCyControlEndPoint](#), a **TimeOut** of zero will wait forever unless the transaction completes. **TimeOut** has no effect on Isoc endpoints since they will always transfer data or fail immediately.

Example

```
char buffer[128];
LONG length = 128;

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

USBDevice->BulkOutEndPt->TimeOut = 1000; // 1 sec timeout
USBDevice->BulkOutEndPt->XferData(buf, length);
```

9.20 UsbdStatus

Description

UsbdStatus member contains an error code returned from the last call to the [XferData](#) or [BeginDataXfer](#) methods.

UsbdStatus can be decoded by passing the value to the [CCyUSBDevice::UsbdStatusString\(\)](#) method.

9.21 WaitForXfer()

Description

This method is used in conjunction with [BeginDataXfer](#) and [FinishDataXfer](#) to perform asynchronous IO.

The **ov** parameter points to the OVERLAPPED object that was passed in the preceding [BeginDataXfer](#) call.

Timeout limits the time, in milliseconds, that the library will wait for the transaction to complete.

You will usually want to use the synchronous [XferData](#) method rather than the asynchronous [BeginDataXfer/WaitForXfer/FinishDataXfer](#) approach.

Example

```
// This example assumes that the device automatically sends back,
// over its bulk-IN endpoint, any bytes that were received over its
// bulk-OUT endpoint (commonly referred to as a loopback function)

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

OVERLAPPED outOvLap, inOvLap;
outOvLap.hEvent = CreateEvent(NULL, false, false, "CYUSB_OUT");
inOvLap.hEvent = CreateEvent(NULL, false, false, "CYUSB_IN");

char inBuf[128];
ZeroMemory(inBuf, 128);

char buffer[128];
LONG length = 128;

// Just to be cute, request the return data before initiating the loopback
UCHAR *inContext = USBDevice->BulkInEndPoint->BeginDataXfer(inBuf, length,
&inOvLap);
UCHAR *outContext = USBDevice->BulkOutEndPoint->BeginDataXfer(buffer, length,
&outOvLap);

USBDevice->BulkOutEndPoint->WaitForXfer(&outOvLap,100);
USBDevice->BulkInEndPoint->WaitForXfer(&inOvLap,100);

USBDevice->BulkOutEndPoint->FinishDataXfer(buffer, length, &outOvLap,outContext);
USBDevice->BulkInEndPoint->FinishDataXfer(inBuf, length, &inOvLap,inContext);

CloseHandle(outOvLap.hEvent);
CloseHandle(inOvLap.hEvent);
```

9.22 XferData()

Description

XferData sends or receives **len** bytes of data from / into **buf**.

This is the primary IO method of the library for transferring data. Most data transfers should occur by invoking the **XferData** method of an instantiated endpoint object.

XferData calls the appropriate [BeginDataXfer](#) method for the instantiated class (one of [CCyBulkEndPoint](#), [CCyControlEndPoint](#), [CCyInterruptEndPoint](#), or [CCyIsocEndPoint](#)). It then waits for the transaction to complete (or until the endpoint's [Timeout](#) expires), and finally calls the [FinishDataXfer](#) method to complete the transaction.

For all non-control endpoints, the direction of the transfer is implied by the endpoint itself. (Each such endpoint will either be an IN or an OUT endpoint.)

For control endpoints, the [Direction](#) must be specified, along with the other control-specific parameters.

XferData performs synchronous (i.e. blocking) IO operations. It does not return until the transaction completes or the endpoint's TimeOut has elapsed.

Returns **true** if the transaction successfully completes before TimeOut has elapsed.

Note that the **len** parameter is a reference, meaning that the method can modify its value. The number of bytes actually transferred is passed back in **len**.

The **pktInfos** parameter is optional and points to an array of [CCyIsoPktInfo](#) objects. It should only be used for Isochronous endpoint transfers.

NOTE: For ISOC transfers, the buffer length and the endpoint's transfers size (see [SetXferSize](#)) must be a multiple of 8 times the endpoint's [MaxPktSize](#).

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

char buf[12] = "hello world";
LONG length = 11;

if (USBDevice->BulkOutEndPt)
    USBDevice->BulkOutEndPt->XferData(buf, length);
```

10 CCyUSBInterface

Header

CyUSB.h

Description

CCyUSBInterface represents a USB device interface. Such interfaces have one or more endpoints.

When [CCyUSBDevice::Open\(\)](#) is called, an instance of [CCyUSBConfig](#) is constructed for each configuration reported by the open device's device descriptor. (Normally, there is just one.)

In the process of construction, [CCyUSBConfig](#) creates instances of [CCyUSBInterface](#) for each interface exposed in the device's configuration descriptor. In turn, the [CCyUSBInterface](#) class creates instances of [CyUSBEndPoint](#) for each endpoint descriptor contained in the interface descriptor. In this iterative fashion, the entire structure of [Configs->Interfaces->EndPoints](#) gets populated from a single construction of the [CCyUSBConfig](#) class.

The below example code shows how you might use the [CCyUSBInterface](#) class in an application.

Example

```
// This code lists all the endpoints reported
//   for all the interfaces reported
//   for all the configurations reported
//   by the device.

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);
```

```

String s;

for (int c=0; c<USBDevice->ConfigCount(); c++) {
    CCyUSBConfig cfg = USBDevice->GetUSBConfig(c);

    s.printf("bLength: 0x%x",cfg.bLength); EZOutputMemo->Lines->Add(s);
    s.printf("bDescriptorType: %d",cfg.bDescriptorType); EZOutputMemo->Lines-
>Add(s);
    s.printf("wTotalLength: %d (0x%x)",cfg.wTotalLength,cfg.wTotalLength);
EZOutputMemo->Lines->Add(s);
    s.printf("bNumInterfaces: %d",cfg.bNumInterfaces); EZOutputMemo->Lines-
>Add(s);
    s.printf("bConfigurationValue: %d",cfg.bConfigurationValue); EZOutputMemo-
>Lines->Add(s);
    s.printf("iConfiguration: %d",cfg.iConfiguration); EZOutputMemo->Lines-
>Add(s);
    s.printf("bmAttributes: 0x%x",cfg.bmAttributes); EZOutputMemo->Lines->Add(s);
    s.printf("MaxPower: %d",cfg.MaxPower); EZOutputMemo->Lines->Add(s);
    EZOutputMemo->Lines->Add("*****");

    for (int i=0; i<cfg.AltInterfaces; i++) {
        CCyUSBInterface *ifc = cfg.Interfaces[i];
        EZOutputMemo->Lines->Add("Interface Descriptor: " + String(i+1));
        EZOutputMemo->Lines->Add("-----");
        s.printf("bLength: 0x%x",ifc->bLength); EZOutputMemo->Lines->Add(s);
        s.printf("bDescriptorType: %d",ifc->bDescriptorType); EZOutputMemo->Lines-
>Add(s);
        s.printf("bInterfaceNumber: %d",ifc->bInterfaceNumber); EZOutputMemo-
>Lines->Add(s);
        s.printf("bAlternateSetting: %d",ifc->bAlternateSetting); EZOutputMemo-
>Lines->Add(s);
        s.printf("bNumEndpoints: %d",ifc->bNumEndpoints); EZOutputMemo->Lines-
>Add(s);
        s.printf("bInterfaceClass: %d",ifc->bInterfaceClass); EZOutputMemo->Lines-
>Add(s);

        for (int e=0; e<ifc->bNumEndpoints; e++) {
            CCyUSBEndPoint *ept = ifc->EndPoints[e+1];
            EZOutputMemo->Lines->Add("EndPoint Descriptor: " + String(e+1));
            EZOutputMemo->Lines->Add("-----");
            s.printf("bLength: 0x%x",ept->DscLen); EZOutputMemo->Lines->Add(s);
            s.printf("bDescriptorType: %d",ept->DscType); EZOutputMemo->Lines-
>Add(s);
            s.printf("bEndpointAddress: 0x%x",ept->Address); EZOutputMemo->Lines-
>Add(s);
            s.printf("bmAttributes: 0x%x",ept->Attributes); EZOutputMemo->Lines-
>Add(s);
            s.printf("wMaxPacketSize: %d",ept->MaxPktSize); EZOutputMemo->Lines-
>Add(s);
            s.printf("bInterval: %d",ept->Interval); EZOutputMemo->Lines->Add(s);
            EZOutputMemo->Lines->Add("*****");
        }
    }
}

```

10.1 bAlternateSetting

Description

This data member contains the **bAlternateSetting** field from the currently selected interface's interface descriptor.

This data member exists for completeness and debugging purposes. You should normally never need

to access this data member.

10.2 bAltSettings

Description

This data member contains the number of valid alternate interface settings exposed by this interface.

For an interface that exposes a primary interface and two alternate interfaces, this value would be 3.

This data member exists for completeness and debugging purposes. You should normally never need to access this data member.

See [CCyUSBDevice::AltIntfcCount\(\)](#).

10.3 bDescriptorType

Description

This data member contains the **bDescriptorType** field of the `USB_INTERFACE_DESCRIPTOR` structure that was passed to the interface object's constructor. (Because the passed descriptor was an interface descriptor, this value should always be 0x04.)

This data member exists for completeness and debugging purposes. You should normally never need to access this data member.

10.4 CCyUSBInterface()

Description

This is the constructor for the [CCyUSBInterface](#) class.

It reads [bNumEndpoint](#) endpoint descriptors and creates the appropriate type of endpoint object for each one, saving a pointer to each new endpoint in the class's [EndPoints](#) array.

10.5 CCyUSBInterface()

Description

This is the *copy* constructor for the [CCyUSBInterface](#) class.

This constructor copies all of the simple data members of `intfc`. It then walks through `intfc`'s [EndPoints](#) array, making copies of every endpoint referenced there and storing pointers to the new copies in its own `EndPoints` array.

You should usually not call the copy constructor explicitly. It is called by the copy constructor for [CCyUSBConfig](#) when [CCyUSBDevice::GetUSBConfig\(\)](#) is called.

The below example shows how you could create a copy of the first interface exposed by a device.

Example

```
CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);  
  
CCyUSBConfig cfg = USBDevice->GetUSBConfig(0);
```

```
CCyUSBInterface iface = new CCyUSBInterface(*cfg.Interfaces[0]);
```

10.6 bInterfaceClass

Description

This data member contains the **bInterfaceClass** field from the currently selected interface's interface descriptor.

This data member exists for completeness and debugging purposes. You should normally never need to access this data member.

10.7 bInterfaceNumber

Description

This data member contains the **bInterfaceNumber** field from the currently selected interface's interface descriptor.

This data member exists for completeness and debugging purposes. You should normally never need to access this data member.

10.8 bInterfaceProtocol

Description

This data member contains the **bInterfaceProtocol** field from the currently selected interface's interface descriptor.

This data member exists for completeness and debugging purposes. You should normally never need to access this data member.

10.9 bInterfaceSubClass

Description

This data member contains the **bInterfaceSubClass** field from the currently selected interface's interface descriptor.

This data member exists for completeness and debugging purposes. You should normally never need to access this data member.

10.10 bLength

Description

This data member contains the **bLength** field from the currently selected interface's interface descriptor. It indicates the length of the interface descriptor. (Because the descriptor is an interface descriptor, this value should always be 0x09.)

10.11 bNumEndpoints

Description

This data member contains the **bNumEndpoints** field from the currently selected interface's interface

descriptor. It indicates how many endpoint descriptors are returned for the selected interface.

This data member exists for completeness and debugging purposes. You should normally never need to access this data member.

10.12 EndPoints

Description

This is the key data member of the CCyUSBInterface class. It is an array of pointers to CCyUSBEndPoint objects that represent the endpoint descriptors returned, by the device, for the interface.

The [CCyUSBDevice::EndPoints](#) member is actually a pointer to the currently selected interface's EndPoints array.

Example

```
// This code lists all the endpoints reported
// for all the interfaces reported
// for all the configurations reported
// by the device.

CCyUSBDevice *USBDevice = new CCyUSBDevice(Handle);

String s;

for (int c=0; c<USBDevice->ConfigCount(); c++) {
    CCyUSBConfig cfg = USBDevice->GetUSBConfig(c);

    s.printf("bLength: 0x%x",cfg.bLength); EZOutputMemo->Lines->Add(s);
    s.printf("bDescriptorType: %d",cfg.bDescriptorType); EZOutputMemo->Lines-
>Add(s);
    s.printf("wTotalLength: %d (0x%x)",cfg.wTotalLength,cfg.wTotalLength);
EZOutputMemo->Lines->Add(s);
    s.printf("bNumInterfaces: %d",cfg.bNumInterfaces); EZOutputMemo->Lines-
>Add(s);
    s.printf("bConfigurationValue: %d",cfg.bConfigurationValue); EZOutputMemo-
>Lines->Add(s);
    s.printf("iConfiguration: %d",cfg.iConfiguration); EZOutputMemo->Lines-
>Add(s);
    s.printf("bmAttributes: 0x%x",cfg.bmAttributes); EZOutputMemo->Lines->Add(s);
    s.printf("MaxPower: %d",cfg.MaxPower); EZOutputMemo->Lines->Add(s);
    EZOutputMemo->Lines->Add("*****");

    for (int i=0; i<cfg.AltInterfaces; i++) {
        CCyUSBInterface *ifc = cfg.Interfaces[i];
        EZOutputMemo->Lines->Add("Interface Descriptor:" + String(i+1));
        EZOutputMemo->Lines->Add("-----");
        s.printf("bLength: 0x%x",ifc->bLength); EZOutputMemo->Lines->Add(s);
        s.printf("bDescriptorType: %d",ifc->bDescriptorType); EZOutputMemo->Lines-
>Add(s);
        s.printf("bInterfaceNumber: %d",ifc->bInterfaceNumber); EZOutputMemo-
>Lines->Add(s);
        s.printf("bAlternateSetting: %d",ifc->bAlternateSetting); EZOutputMemo-
>Lines->Add(s);
        s.printf("bNumEndpoints: %d",ifc->bNumEndpoints); EZOutputMemo->Lines-
>Add(s);
        s.printf("bInterfaceClass: %d",ifc->bInterfaceClass); EZOutputMemo->Lines-
>Add(s);

        for (int e=0; e<ifc->bNumEndpoints; e++) {
```

```

        CCyUSBEndPoint *ept = ifc->Endpoints[e+1];
        EZOutputMemo->Lines->Add("EndPoint Descriptor: " + String(e+1));
        EZOutputMemo->Lines->Add("-----");
        s.printf("bLength: 0x%x", ept->DscLen); EZOutputMemo->Lines->Add(s);
        s.printf("bDescriptorType: %d", ept->DscType); EZOutputMemo->Lines-
>Add(s);
        s.printf("bEndpointAddress: 0x%x", ept->Address); EZOutputMemo->Lines-
>Add(s);
        s.printf("bmAttributes: 0x%x", ept->Attributes); EZOutputMemo->Lines-
>Add(s);
        s.printf("wMaxPacketSize: %d", ept->MaxPktSize); EZOutputMemo->Lines-
>Add(s);
        s.printf("bInterval: %d", ept->Interval); EZOutputMemo->Lines->Add(s);
        EZOutputMemo->Lines->Add("*****");
    }
}

```

10.13 iInterface

Description

This data member contains the **iInterface** field from the currently selected interface's interface descriptor.

This data member exists for completeness and debugging purposes. You should normally never need to access this data member.