Cypress CyAPI Programmer's Reference

Table of Contents

1 Overview

Library Overview

Top Next

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 **CyUsb3.sys** device driver. The library is only able to communicate with USB devices that are served by (i.e. bound to) this driver.

Rather than communicate with the driver via Windows API calls such as *SetupDiXxxx* and *DeviceIoControl*, applications can call simpler CyAPI methods such as Open, Close, and XferData to communicate with these 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 files are available for use with Microsoft Visual Studio 2008.

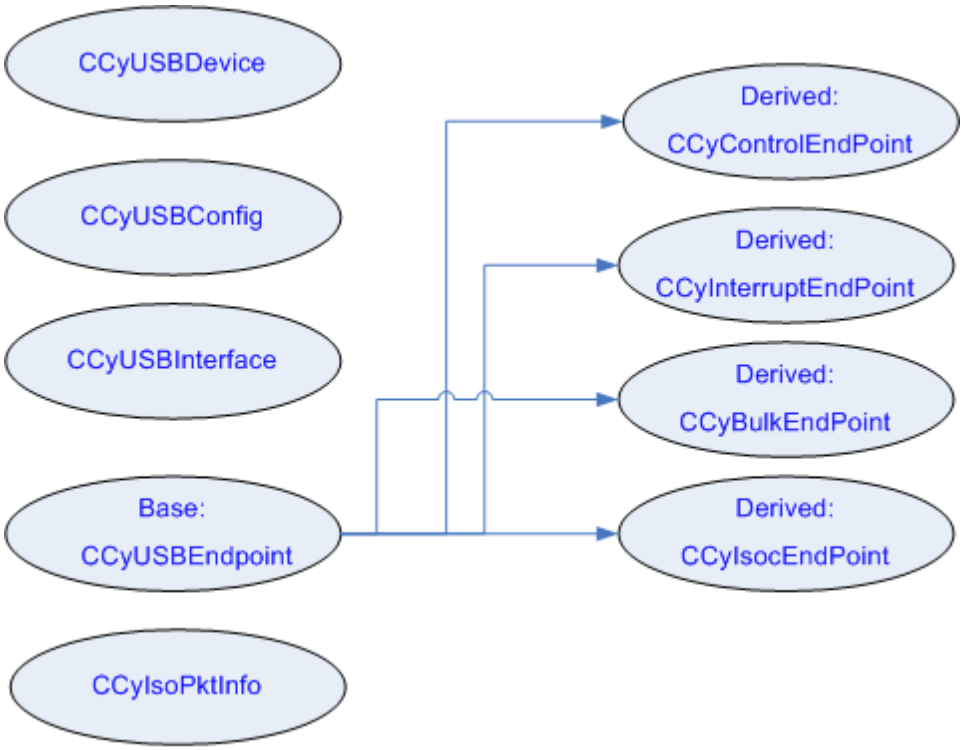
The library employs a ***Device and EndPoints*** use model. To use the library you must create an instance of the **CCyUSBDevice** class using the **new** keyword. A **CCyUSBDevice** object knows how many USB devices are attached to the **CyUsb3.sys** driver and can be made to abstract any one of those 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.

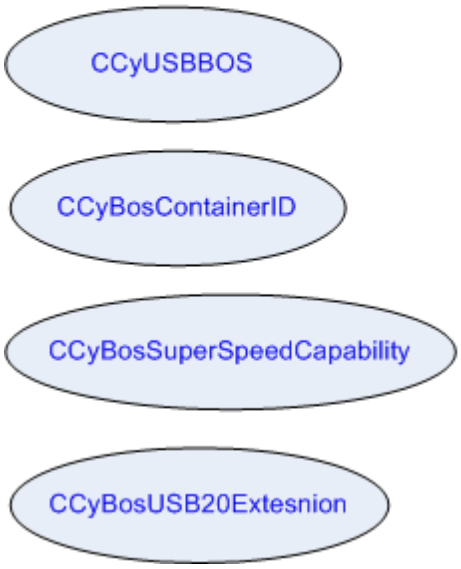
When a **CCyUSBDevice** object is open to an attached USB device, its endpoint members provide an interface for performing data transfers to and from the device's endpoints. Endpoint-specific data members and methods such as MaxPktSize, TimeOut, bIn, Reset and XferData are only accessible through endpoint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

2 Library Class Hierarchy

The class hierarchy diagram shown below illustrates the C++ CyAPI library interface classes.





3 USB3.0 Support Overview

USB3.0 Support Overview

Top Previous Next

Description

The Binary Device Object Store(BOS) descriptor defines a root descriptor that is similar to the configuration descriptor and a base descriptor for accessing a family of related descriptors. A host can read the wTotalLength field of the BOS descriptor to find the length of the device level descriptor set.

API

All BOS support APIs are incorporated in the CCyUSBDevice class.

- GetBosDescriptor()
- GetBosContainerIDDescriptor()
- GetBosSSCapabilityDescriptor()
- GetBosUSB20DeviceExtensionDesc()

Data Structure

All BOS data structure definitions are defined in the USB30_def.h header file.

- USB_BOS_DESCRIPTOR
- USB_BOS_CONTAINER_ID
- USB_BOS_SS_DEVICE_CAPABILITY
- USB_BOS_USB20_DEVICE_EXTENSION

Classes

All BOS class definitions are defined in the CyAPI.h header file.

- CCyUSBBOS
- CCyBOSContainerID
- CCyBOSSuperSpeedCapability
- CCyBOSUSB20Extension

Device Speed

Super speed variable is defined in the CCyUSBDevice class.

- bSuperSpeed

SuperSpeed Endpoint Companion descriptor

All Superspeed endpoint companion descriptor data variable definition is incorporated in the CCyUSBEndPoint. The following variables will be initialized with zero if device is USB2.0, and will be set to the the SuperSpeed endpoint companion descriptor values, for USB3.0 devices.

- ssdscLen
- ssdscType
- ssbytesperinterval
- ssbmAttribute
- ssmaxburst

Firmware Download

CyUSB library provides APIs to download a firmware binary to the FX3 device RAM or EEPROM connected to the FX3 device.

- CCyFX3Device

NOTE : Please note that this library does not support USB3.0 bulk streams and power management.

4 CCyBulkEndPoint

CCyBulkEndPoint Class

Previous Top Next

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 CCyUSBDevice is created, instances of this class are automatically created for all bulk endpoints as members of that class. Two such members of CCyUSBDevice are BulkInEndPt and BulkOutEndPt.

Example

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

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

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

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

4.1 BeginDataXfer( )

PUCHAR CCyBulkEndPoint::BeginDataXfer(
PCHAR buf, LONG len, OVERLAPPED *ov**)**

Previous Top Next

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(NULL);
OVERLAPPED outOvLap, inOvLap;

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

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

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

// 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 CCyBulkEndPoint( )

CCyBulkEndPoint::CCyBulkEndPoint (void**)** Previous Top Next

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).

4.3 CCyBulkEndPoint( )

CCyBulkEndPoint::CCyBulkEndPoint (HANDLE h, PUSB_ENDPOINT_DESCRIPTOR pEndPtDescriptor**)**

Previous Top Next

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.

5 CCyControlEndPoint

CCyControlEndPoint Class

Previous Top Next

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, isochronous 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( NULL);

// Obtain the control endpoint pointer
CCyControlEndPoint *ept = USBDevice->ControlEndPt;
unsigned char buf[ 512];
LONG buflen = 512;

// Send a vendor request ( bRequest = 0x05) to the device
ept->Target    = TGT_DEVICE;
ept->ReqType   = REQ_VENDOR;
ept->Direction = DIR_TO_DEVICE;
ept->ReqCode   = 0x05;
ept->Value     = 1;
ept->Index     = 0;

ZeroMemory( buf, 512);
ept->XferData( buf,  buflen);
```

5.1 BeginDataXfer( )

PUCHAR CCyControlEndPoint::BeginDataXfer (
PCHAR buf, LONG len, OVERLAPPED *ov**)**

Previous Top Next

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.

Control transfers require six parameters that are not needed for bulk, isochronous 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( NULL);

// Get a handle to the control endpoint
CCyControlEndPoint  *ept = USBDevice->ControlEndPt;

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

unsigned 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;

// Begin the transfer and wait for it to complete
PUCHAR Context = ept->BeginDataXfer( buffer, length, &OvLap);
```

```
ept->WaitForXfer(&OvLap,100);
ept->FinishDataXfer(buffer, length, &OvLap,Context);

CloseHandle(OvLap.hEvent);
```

5.2 CCyControlEndPoint( )

CCyControlEndPoint::CCyControlEndPoint(void**)** Previous Top Next

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
ReqCode     = 0
Value       = 0
Index       = 0
```

5.3 CCyControlEndPoint( )

CCyControlEndPoint::CCyControlEndPoint(HANDLE h, PUSB_ENDPOINT_DESCRIPTOR pEndPtDescriptor**)**

Previous Top Next

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
ReqCode     = 0
Value       = 0
Index       = 0
```

5.4 Direction

CTL_XFER_DIR_TYPE CCyControlEndPoint::Direction

Previous Top Next

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 Isochronous 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( NULL);

// Get a handle to the control endpoint
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;

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

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

5.5 Index

WORD CCyControlEndPoint::Index Previous Top Next

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(NULL);

// Get a handle to the control endpoint
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;

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

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

5.6 Read( )

bool CCyControlEndPoint::Read(PCHAR buf, LONG &len**)** Previous Top Next

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 specifies 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( NULL);

// Get a handle to the control endpoint
CCyControlEndPoint *ept = USBDevice->ControlEndPt;

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

unsigned char  buf[ 512];
LONG bytesToRead = 64;

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

5.7 ReqCode

UCHAR CCyControlEndPoint::ReqCode Previous Top Next

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(NULL);

// Get a handle to the control endpoint
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;

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

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

5.8 ReqType

CTL_XFER_REQ_TYPE CCyControlEndPoint::ReqType

Previous Top Next

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(NULL);

// Get a handle to the control endpoint
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;

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

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

5.9 Target

CTL_XFER_TGT_TYPE CCyControlEndPoint::Target

Previous Top Next

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(NULL);

// Get a handle to the control endpoint
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;

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

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

5.10 Value

WORD CCyControlEndPoint::Value Previous Top Next

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( NULL);

// Get a handle to the control endpoint
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;

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

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

5.11 Write( )

bool CCyControlEndPoint::Write(PCHAR buf, LONG &len**)**

Previous Top Next

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( NULL);

// Get a handle to the control endpoint
CCyControlEndPoint *ept = USBDevice->ControlEndPt;

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

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

// Call the write method to perform the control write transfer
ept->Write( buf,  bytesToSend);
```

6 CCyInterruptEndPoint

CCyInterruptEndPoint Class

Previous Top Next

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 CCyUSBDevice is created, instances of this class are automatically created as members of that class. Two such members of CCyUSBDevice are InterruptInEndPt and InterruptOutEndPt.

Example

```
//  Find interrupt endpoints in the EndPoints[] array
CCyInterruptEndPoint *IntInEpt = NULL;
CCyInterruptEndPoint  *IntOutEpt = NULL;

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

// Check all endpoints and get handles to any interrupt endpoints
for (int i=1; i<eptCount; i++) {
  bool   bIn  = ((USBDevice->EndPoints[i]->Address & 0x80)==0x80);
  bool   bInt = (USBDevice->EndPoints[i]->Attributes == 3);

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

6.1 BeginDataXfer( )

PUCHAR CCyInterruptEndPoint::BeginDataXfer
(PCHAR buf, LONG len, OVERLAPPED *ov**)**

Previous Top Next

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(NULL);
  OVERLAPPED outOvLap, inOvLap;

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

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

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

  // Request the return data before initiating the loopback
  UCHAR *inContext = USBDevice->InterruptInEndPt->BeginDataXfer(inBuf,
length, &inOvLap);
  UCHAR *outContext = USBDevice->InterruptOutEndPt->BeginDataXfer(buffer,
length, &outOvLap);

  USBDevice->InterruptOutEndPt->WaitForXfer(&outOvLap,100);
  USBDevice->InterruptInEndPt->WaitForXfer(&inOvLap,100);

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

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

6.2 CCyInterruptEndPoint( )

CCyInterruptEndPoint::CCyInterruptEndPoint (void**)** Previous Top Next

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).

6.3 CCyInterruptEndPoint( )

CCyInterruptEndPoint::CCyInterruptEndPoint(
HANDLE h, PUSB_ENDPOINT_DESCRIPTOR
pEndPtDescriptor**)**

Previous Top Next

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, InterruptInEndPt and InterruptOutEndPt members.

7 CCyIsocEndPoint

CCyIsocEndPoint Class

Previous Top Next

Header
CyUSB.h

Description

CCyIsocEndPoint is a subclass of the CCyUSBEndPoint abstract class.

CCyIsocEndPoint exists to implement an isochronous-specific BeginDataXfer( ) function.

Normally, you should not need to construct any of your own instances of this class. Rather, when an instance of CCyUSBDevice is created, instances of this class are automatically created as members of that class. Two such members of CCyUSBDevice are IsocInEndPt and IsocOutEndPt.

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

// Iterate through all endpoints and pick handles for ISOC endpoints
for  (int  i=1; i<eptCount;  i++) {
  bool  bIn = ((USBDevice->EndPoints[i]->Address  & 0x80)==0x80);
  bool  bInt = (USBDevice->EndPoints[i]->Attributes  == 1);
  if (bInt && bIn)
    IsocInEpt = (CCyIsocEndPoint *) USBDevice->EndPoints[i];
  if (bInt && !bIn)
    IsocOutEpt = (CCyIsocEndPoint *) USBDevice->EndPoints[i];
}
```

7.1 BeginDataXfer( )

PUCHAR CCyIsocEndPoint::BeginDataXfer (
PCHAR buf, LONG len, OVERLAPPED *ov**)**

Previous Top Next

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( NULL);
CCyIsocEndPoint *IsoIn = USBDevice->IsocInEndPt;

if (IsoIn) {
  int  pkts = 16;
  LONG bufSize = IsoIn->MaxPktSize * pkts;

  PUCHAR         context;
  OVERLAPPED     inOvLap;
  PUCHAR         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 each packet, check whether the transfer was completed
      for (int i=0; i< pkts; i++)
        if (isoPktInfos[i].Status)
          partial++;
        else
          complete++;
    }
    else {
      partial++;
    }

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

7.2 CCyIsocEndPoint( )

CCyIsocEndPoint::CCyIsocEndPoint (void) Previous Top Next

Description

This is the default constructor for the CCyIsocEndPoint 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).

7.3 CCyIsocEndPoint( )

CCyIsocEndPoint::CCyIsocEndPoint(HANDLE h, PUSB_ENDPOINT_DESCRIPTOR pEndPtDescriptor**)**

Previous Top Next

Description

This constructor creates a legitimate CCyIsocEndPoint 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 CCyIsocEndPoint objects created for you by the CCyUSBDevice class and accessed via its EndPoints, IsocInEndPt and IsocOutEndPt members.

7.4 CreatePktInfos( )

CCyIsoPktInfo* CCyIsocEndPoint::
CreatePktInfos(LONG bufLen, **int** &packets**)**

Previous Top Next

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);

    // Request the transfer and identify how much data got transferred
    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;
        }
      }
    }

    // Free the buffer and IsoPktInfo array
    delete [] buffer;
    delete [] isoPktInfos;
```

```
}
```

8 CCyIsoPktInfo

CCyIsoPktInfo

Previous Top Next

The **CCyIsoPktInfo** class is defined as:

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

When an Isochronous data 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 (USBD_STATUS_SUCCESS) all the data in the packet is valid. Other non-zero values for the Status field can be found in the Windows Driver Development Kit (DDK) include file, USBDI.H.

The value returned in the **Length** field indicates the number of bytes transferred in the packet. Under ideal conditions, this number will be bufferLength / numPackets (which is the maximum capacity of each packet). However, it is possible that fewer bytes could be transferred in any or all of the packets.

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);

    // Request the transfer and identify how much data got transferred
    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;
      }
    }
  }

  // Free the buffer and IsoPktInfo array
  delete [] buffer;
  delete [] isoPktInfos;
}
```

9 CCyFX3Device

CCyFX3Device

Top Previous Next

Description

API defined for this class will work with the FX3 boot devices only. The behaviour of each API is undefined for non-boot FX3 devices.

CCyFX3Device extends the functionality of CCyUSBDevice by adding methods to download firmware to the Cypress FX3 boot devices.

Note that any CCyUSBDevice class object can be cast into a CCyFX3Device object. However, only those that represent actual FX3 boot devices will function properly when the DownloadFw method of CCyFX3Device is invoked.

Please use the CCyUSBDevice class instead of CCyFX3Device class for non-boot devices.

Example

Example#1 Get instance for FX3-boot device and download a firmware binary to the device RAM.

```
CCyFX3Device *m_usbDevice = new CCyFX3Device();
FX3_FWDWNLOAD_ERROR_CODE dwld_status = FAILED;

// Open first USB device
if(m_usbDevice->Open(0))
{
  //Check if boot loader is running.
  status = m_usbDevice->IsBootLoaderRunning();
  if(status)
  {
    // Download the file C:\Bulkloop.img file to FX3 device RAM
    dwld_status = m_usbDevice->DownloadFw("C:\Bulkloop.img",RAM)
  }
}
```

Example#2 Get instance for FX3 non-boot device.

Refer to the example code on page CCyUSBDevice

9.1 DownloadFw( )

FX3_FWDWNLOAD_ERROR_CODE CCyFX3Device::
DownloadFw (char *fileName, FX3_FWDWNLOAD_MEDIA_TYPE enMediaType**)**

Top Previous Next

Description

The DownloadFw method of CCyFX3Device allows the user to download firmware to various media (RAM, I2C E2PROM and SPI FLASH) associated with a FX3 device.

The file name of the firmware file (*.img file format file only) is passed as the first parameter to the API.

The second parameter defines the Media Type using members of FX3_FWDWNLOAD_MEDIA_TYPE

The API returns a FX3_FWDWNLOAD_ERROR_CODE return code.

Examples:
NOTE : The sample example code provided is only a guideline and is not ready to compile code.

Example#1 Sample code for downloading FX3 firmware to RAM

```
CCyFX3Device *m_usbDevice = new CCyFX3Device();
FX3_FWDWNLOAD_ERROR_CODE dwld_status = FAILED;

if(m_usbDevice->Open(0))
{
  //Check if boot loader is running.
  status = m_usbDevice->IsBootLoaderRunning();
  if(status)
  {
    // Download the file C:\Bulkloop.img to FX3 device RAM
    dwld_status = m_usbDevice->DownloadFw("C:\Bulkloop.img",RAM);
  }
}
```

Example#2 Sample code for downloading FX3 firmware to I2C E2PROM

Step 1 First Download the Boot Programmer IMG file to RAM. The CyBootProgrammer.img file is available in the Cypress SS USBSuite installation directory: \Cypress USBSuite\bin\CyBootProgrammer.img

```
CCyFX3Device *m_usbDevice = new CCyFX3Device();
FX3_FWDWNLOAD_ERROR_CODE dwld_status = FAILED;

if(m_usbDevice->Open(0))
{
  //Check if boot loader is running.
  status = m_usbDevice->IsBootLoaderRunning();
  if(status)
  {
    // Download the boot programmer IMG file to RAM first
    dwld_status = m_usbDevice->DownloadFw("\Cypress
```

```
USBSuite\bin\CyBootProgrammer.img", RAM);
    }
  }
```

Step 2 Download actual IMG file to the I2C E2PROM.

Note: After downloading the boot programmer firmware, the device will be re-enumerated with different VID/PID. First time users will need to install the driver from the Cypress USBSuite\driver\bin\ directory for the boot programmer.

```
CCyFX3Device *m_usbDevice = new CCyFX3Device();
FX3_FWDWNLOAD_ERROR_CODE dwld_status = FAILED;

if(m_usbDevice->Open(0))
{
  // Download the Bulkloop IMG file to I2C E2PROM first
  dwld_status = m_usbDevice->DownloadFw("C:\Bulkloop.IMG", I2CE2PROM);
}
```

Example#3 Sample code for downloading FX3 firmware to SPI FLASH

Step 1 First Download the Boot Programmer IMG file to RAM. The CyBootProgrammer.img file is available in the Cypress SS USBSuite installation directory \Cypress USBSuite\bin\CyBootProgrammer.img

```
CCyFX3Device *m_usbDevice = new CCyFX3Device();
FX3_FWDWNLOAD_ERROR_CODE dwld_status = FAILED;

if(m_usbDevice->Open(0))
{
  //Check if boot loader is running.
  status = m_usbDevice->IsBootLoaderRunning();
  if(status)
  {
    // Download the boot programmer IMG file to RAM first
    dwld_status = m_usbDevice->DownloadFw("\Cypress
USBSuite\bin\CyBootProgrammer.img", RAM);
    }
  }
```

Step 2 Download actual IMG file to SPI FLASH.

Note: After downloading the boot programmer firmware, the device will be re-enumerated with different VID/PID. First time users will need to install the driver from the Cypress USBSuite\driver\bin\ directory for the boot programmer.

```
CCyFX3Device *m_usbDevice = new CCyFX3Device();
FX3_FWDWNLOAD_ERROR_CODE dwld_status = FAILED;

if(m_usbDevice->Open(0))
{
  // Download the Bulkloop IMG file to I2C E2PROM first
  dwld_status = m_usbDevice->DownloadFw("C:\Bulkloop.IMG", SPIFLASH);
}
```

9.2 IsBootLoaderRunning( )

bool IsBooTLoaderRunning (void) Top Previous Next

Description

The IsBootLoaderRunning function sends a vendor command to check the FX3 boot loader status. If boot loader is running, then it will return true; otherwise false.

Example

```
CCyFX3Device *m_usbDevice = new CCyFX3Device();
FX3_FWDWNLOAD_ERROR_CODE dwld_status = FAILED;

if(m_usbDevice->Open(0))
{
  //Check if boot loader is running.
  status = m_usbDevice->IsBootLoaderRunning();
  if(status)
  {
    // Download the file C:\Bulkloop.img to FX3 device RAM
    dwld_status = m_usbDevice->DownloadFw("C:\Bulkloop.img",RAM);
  }
}
```

10 CCyUSBDevice

CCyUSBDevice Class

Previous Top Next

Header
CyUSB.h

Description

The CCyUSBDevice class is the primary entry point into the CyAPI 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

```
//  Create an instance of CCyUSBDevice
CCyUSBDevice *USBDevice  = new  CCyUSBDevice( NULL);

//  Look for a device having VID = 04b4, PID = 00f1
int devices = USBDevice->DeviceCount( );
int vID, pID;
int d = 0;

do {
  // Open( ) automatically calls Close( ) if necessary
  USBDevice->Open( d);
  vID = USBDevice->VendorID;
  pID  = USBDevice->ProductID;
  d++;
} while  (( d < devices ) && ( vID != 0x04b4) &&  ( pID != 0x00f1));
```

10.1 AltIntfc( )

UCHAR CCyUSBDevice::AltIntfc(void**)** Previous Top Next

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.

Note:
As Windows represents each USB device interface as a separate device, the interface number need not be specified.

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.

10.2 AltIntfcCount( )

UCHAR CCyUSBDevice::AltIntfcCount(void**)** Previous Top Next

Description

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

The primary interface setting (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.

10.3 bHighSpeed

bool CCyUSBDevice::bHighSpeed Previous Top Next

Description

bHighSpeed indicates whether or not the device is a high speed USB device.

If the USB device represented is a high speed device, **bHighSpeed** will be **true**. Otherwise, **bHighSpeed** will be **false**.

Example

```
CCyUSBDevice  *USBDevice = new  CCyUSBDevice( NULL);
if (USBDevice->bHighSpeed) {
  // Do something
}
```

10.4 bSuperSpeed

bool CCyUSBDevice::bSuperSpeed Previous Top Next

Description

bSuperSpeed indicates whether or not the device is a USB Super Speed (USB 3.0) device.

If the USB device represented is a Super Speed device, **bSuperSpeed** will be **true**. Otherwise, **bSuperSpeed** will be **false**.

Example

```
CCyUSBDevice *USBDevice = new  CCyUSBDevice( NULL);
if (USBDevice->bSuperSpeed) {
  // Do something
}
```

10.5 BcdDevice

USHORT CCyUSBDevice::BcdDevice Previous Top Next

Description

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

10.6 BcdUSB

USHORT CCyUSBDevice::BcdUSB Previous Top Next

Description

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

10.7 BulkInEndPt

CCyBulkEndPoint* CCyUSBDevice::
BulkInEndPt

Previous Top Next

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( NULL);
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;
  }
}
```

10.8 BulkOutEndPt

**CCyBulkEndPoint* CCyUSBDevice::
BulkOutEndPt**

Previous Top Next

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, BulkOutEndPt will be set to NULL.

Example

```
// Find a second bulk OUT endpoint in the EndPoints[] array
CCyUSBDevice  *USBDevice = new  CCyUSBDevice(NULL);
CCyBulkEndPoint *BulkOut2 = NULL;
int  eptCount = USBDevice->EndPointCount();

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

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

10.9 CCyUSBDevice( )

CCyUSBDevice::CCyUSBDevice(HANDLE hnd = NULL, GUID guid = CYUSBDRV_GUID)

Previous Top Next

Description

This is the constructor for the CCyUSBDevice class.

It registers the Windows handle *hnd* to receive USB Plug and Play (PnP) 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 do not 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 do not want to register for PnP events, but want to pass your own driver GUID to the constructor, you will need to pass NULL as the *hnd* parameter.

Example #1

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

// Overload MainForm's WndProc method to watch for PnP messages
// Requires #include <dbt.h>
void MainForm::WndProc (
  Message &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;
    }
  }

  Form::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 = {0xBE18AA60, 0x7F6A, 0x11D4, 0x97, 0xDD, 0x00, 0x01, 0x02,
0x29, 0xB9, 0x59};
  CCyUSBDevice *USBDevice = new CCyUSBDevice(NULL, guid); // Does not
register for PnP events
```

10.10 ~CCyUSBDevice( )

CCyUSBDevice::~CCyUSBDevice(void) Previous Top Next

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.

10.11 Close( )

void CCyUSBDevice:: Close(void**)** Previous Top Next

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. All "shortcut" pointers to elements of the EndPoints array (ControlEndPt, IsocIn/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).

10.12 Config( )

UCHAR CCyUSBDevice::Config(void**)** Previous Top Next

Description

This method returns the current configuration index for the device.

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

10.13 ConfigAttrib

UCHAR CCyUSBDevice::ConfigAttrib Previous Top Next

Description

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

10.14 ConfigCount( )

UCHAR CCyUSBDevice::ConfigCount(void**)** Previous Top Next

Description

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

10.15 ConfigValue

UCHAR CCyUSBDevice::ConfigValue Previous Top Next

Description

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

10.16 ControlEndPt

CCyControlEndPoint* CCyUSBDevice::ControlEndPt

Previous Top Next

Description

ControlEndPt points to an object representing the primary control endpoint, endpoint 0.

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

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

Example

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

// Get a handle to the control endpoint
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;

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

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

10.17 DevClass

UCHAR CCyUSBDevice::DevClass Previous Top Next

Description

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

10.18 DeviceCount( )

UCHAR CCyUSBDevice::DeviceCount(void) Previous Top Next

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

```
//  Create an instance of CCyUSBDevice
CCyUSBDevice *USBDevice  = new  CCyUSBDevice( NULL);

//  Look for a device having VID = 04b4, PID = 00f1
int devices = USBDevice->DeviceCount( );
int vID, pID;
int d = 0;

do {
  // Open( ) automatically calls Close( ) if necessary
  USBDevice->Open( d);
  vID = USBDevice->VendorID;
  pID  = USBDevice->ProductID;
  d++;
} while  (( d < devices ) && ( vID != 0x04b4) &&  ( pID != 0x00f1));
```

10.19 DeviceHandle( )

HANDLE CCyUSBDevice::DeviceHandle(void) Previous Top Next

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.

10.20 DeviceName

char CCyUSBDevice::DeviceName [USB_STRING_MAXLEN]

Previous Top Next

Description

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

10.21 DevProtocol

UCHAR CCyUSBDevice::DevProtocol Previous Top Next

Description

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

10.22 DevSubClass

UCHAR CCyUSBDevice::DevSubClass Previous Top Next

Description

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

10.23 DriverGUID( )

GUID CCyUSBDevice::DriverGUID(void**)** Previous Top Next

Description

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

See also: CCyUSBDevice( )

10.24 DriverVersion

ULONG CCyUSBDevice::DriverVersion Previous Top Next

Description

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

10.25 EndPointCount( )

UCHAR CCyUSBDevice::EndPointCount(void**)** Previous Top Next

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(NULL);
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)==0x80);
  bool bBulk = (USBDevice->EndPoints[i]->Attributes == 2);

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

10.26 EndPointOf( )

CCyUSBEndPoint* CCyUSBDevice::EndPointOf(
UCHAR addr**)**

Previous Top Next

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

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

// Look for Endpoint 0x82 (2-IN)
UCHAR eptAddr = 0x82;
CCyUSBEndPoint *EndPt = USBDevice->EndPointOf( eptAddr);

// Reset the endpoint, if found
if (EndPt)
  EndPt->Reset();
```

10.27 EndPoints

CCyUSBEndPoint CCyUSBDevice::EndPoints** Previous Top Next

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 / alternate 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 SetAltIntfc( ) 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

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

int epCnt = USBDevice->EndPointCount();
bool bBulk, bIn;
int  blkInCnt = 0;

// Count the bulk-in endpoints
for (int e=0; e<epCnt; e++) {
  bBulk = (USBDevice->EndPoints[e]->Attributes  == 2);
  bIn   = ((USBDevice->EndPoints[e]->Address & 0x80)==0x80);
  if (bBulk && bIn)
    blkInCnt++;
}
```

10.28 FriendlyName

char CCyUSBDevice::FriendlyName [USB_STRING_MAXLEN]

Previous Top Next

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.

10.29 GetDeviceDescriptor( )

void CCyUSBDevice::GetDeviceDescriptor(
PUSB_DEVICE_DESCRIPTOR descr**)**

Previous Top Next

Description

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

10.30 GetBosDescriptor()

bool CCyUSBDevice::GetBosDescriptor(
PUSB_BOS_DESCRIPTOR descr**)**

Top Previous Next

Description

This function copies the device's Binary device Object Store (BOS) descriptor into memory pointed to by ***descr***. This function will return BOS descriptor only for USB 3.0 devices.

Return Value

True Operation successful.
False Operation failed (Not a USB 3.0 device)

10.31 GetBosUSB20DeviceExtensionDescriptor()

bool CCyUSBDevice::
GetBosUSB20DeviceExtensionDescriptor(
PUSB_BOS_USB20_DEVICE_EXTENSION descr**)**

Previous Top Next

Description

This function copies the device's USB 2.0 device extension descriptor into the **descr** buffer. This function will return the USB 2.0 Device extension descriptor only for USB 3.0 devices.

Return Value

True Operation successful.
False Operation failed (Not a USB 3.0 device)

10.32 GetBosContainerIDDescriptor()

bool CCyUSBDevice::
GetBosContainerIDDescriptor(
PUSB_BOS_CONTAINER_ID descr**)**

Previous Top Next

Description

This function copies the device's Container ID descriptor into the **descr** buffer. This function will return Container ID only for USB 3.0 devices, otherwise it will return false as a return value.

Return Value

True Operation successful.
False Operation failed (Not a USB 3.0 device)

10.33 GetBosSSCapabilityDescriptor()

bool CCyUSBDevice::
GetBosSSCapabilityDescriptor(
PUSB_BOS_SS_DEVICE_CAPABILITY descr**)**

Previous Top Next

Description

This function copies the device's Super Speed capability descriptor into the **descr** buffer. This function will return Super Speed capability descriptor only for USB 3.0 devices.

Return Value

True Operation successful.
False Operation failed (Not a USB 3.0 device)

10.34 GetConfigDescriptor( )

void CCyUSBDevice::GetConfigDescriptor(
PUSB_CONFIGURATION_DESCRIPTOR descr**)**

Previous Top Next

Description

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

10.35 GetIntfcDescriptor( )

void CCyUSBDevice::GetIntfcDescriptor(
PUSB_INTERFACE_DESCRIPTOR descr**)**

Previous Top Next

Description

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

10.36 GetUSBConfig( )

CCyUSBConfig CCyUSBDevice::GetUSBConfig(
int index**)**

Previous Top Next

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 snippet lists all the endpoints reported
  // by the device under all interfaces and configurations.

  CCyUSBDevice  *USBDevice = new  CCyUSBDevice( NULL);

  char buf[ 512];
  string s;

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

    // Print all configuration descriptor fields
    sprintf_s( buf,"bLength: 0x%x\n",cfg. bLength);
    s. append( buf);

    sprintf_s( buf,"bDescriptorType: %d\n",cfg. bDescriptorType);
    s. append( buf);

    sprintf_s( buf,"wTotalLength: %d (0x%x)\n",cfg. wTotalLength,cfg.
wTotalLength);
    s. append( buf);

    sprintf_s( buf,"bNumInterfaces: %d\n",cfg. bNumInterfaces);
    s. append( buf);

    sprintf_s( buf,"bConfigurationValue: %d\n",cfg. bConfigurationValue);
    s. append( buf);

    sprintf_s( buf,"iConfiguration: %d\n",cfg. iConfiguration);
    s. append( buf);

    sprintf_s( buf,"bmAttributes: 0x%x\n",cfg. bmAttributes);
    s. append( buf);

    sprintf_s( buf,"MaxPower: %d\n",cfg. MaxPower);
    s. append( buf);
    s. append( "**********************************\n");
```

```
cout<<s;
s.clear();

for (int i=0; i<cfg.AltInterfaces; i++)
{
  // Print all interface descriptor fields
  CCyUSBInterface  *ifc = cfg.Interfaces[i];
  sprintf_s(buf,"Interface  Descriptor:%d\n",(i+1));
  s.append(buf);

  sprintf_s(buf,"--------------------------------\n");
  s.append(buf);

  sprintf_s(buf,"bLength: 0x%x\n",ifc->bLength);
  s.append(buf);

  sprintf_s(buf,"bDescriptorType: %d\n",ifc->bDescriptorType);
  s.append(buf);

  sprintf_s(buf,"bInterfaceNumber: %d\n",ifc->bInterfaceNumber);
  s.append(buf);

  sprintf_s(buf,"bAlternateSetting: %d\n",ifc->bAlternateSetting);
  s.append(buf);

  sprintf_s(buf,"bNumEndpoints: %d\n",ifc->bNumEndpoints);
  s.append(buf);

  sprintf_s(buf,"bInterfaceClass: %d\n",ifc->bInterfaceClass);
  s.append(buf);
  sprintf_s(buf,"**********************************\n");s.append(buf);

  cout<<s;
  s.clear();

  for (int e=0; e<ifc->bNumEndpoints; e++)
  {
    // Print all endpoint descriptor fields
    CCyUSBEndPoint  *ept = ifc->EndPoints[e+1];
    sprintf_s(buf,"EndPoint  Descriptor:%d\n",(e+1));
    s.append(buf);

    sprintf_s(buf,"--------------------------------\n");
    s.append(buf);

    sprintf_s(buf,"bLength: 0x%x\n",ept->DscLen);
    s.append(buf);

    sprintf_s(buf,"bDescriptorType: %d\n",ept->DscType);
    s.append(buf);

    sprintf_s(buf,"bEndpointAddress: 0x%x\n",ept->Address);
    s.append(buf);
```

```
            sprintf_s(buf,"bmAttributes: 0x%x\n",ept->Attributes);
            s.append(buf);

            sprintf_s(buf,"wMaxPacketSize: %d\n",ept->MaxPktSize);
            s.append(buf);

            sprintf_s(buf,"bInterval: %d\n",ept->Interval);
            s.append(buf);
            s.append("**********************************\n");

            cout<<s;
            s.clear();
        }
    }
}
```

10.37 Interface( )

UCHAR CCyUSBDevice::Interface(void**)** Previous Top Next

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.

10.38 InterruptInEndPt

CCyInterruptEndPoint* CCyUSBDevice::InterruptInEndPt

Previous Top Next

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

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

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

10.39 InterruptOutEndPt

**CCyInterruptEndPoint* CCyUSBDevice::
InterruptOutEndPt**

Previous Top Next

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

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

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

10.40 IntfcClass

UCHAR CCyUSBDevice::IntfcClass Previous Top Next

Description

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

10.41 IntfcCount( )

UCHAR CCyUSBDevice::IntfcCount(void) Previous Top Next

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 configuration. Because Windows always represents different reported interfaces as separate devices, the CyUSB3 driver is only shown devices that have a single interface. This causes the IntfcCount( ) method to always return 1.

10.42 IntfcProtocol

UCHAR CCyUSBDevice::IntfcProtocol Previous Top Next

Description

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

10.43 IntfcSubClass

UCHAR CCyUSBDevice::IntfcSubClass Previous Top Next

Description

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

10.44 IsocInEndPt

CCyIsocEndPoint* CCyUSBDevice::IsocInEndPt Previous Top Next

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 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 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( NULL);
CCyIsocEndPoint *IsocIn2 = NULL;
int eptCount = USBDevice->EndPointCount();

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

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

10.45 IsocOutEndPt

CCyIsocEndPoint* CCyUSBDevice::
IsocOutEndPt

Previous Top Next

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

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

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

10.46 IsOpen( )

bool CCyUSBDevice::IsOpen(void**)** Previous Top Next

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.

10.47 Manufacturer

wchar_t CCyUSBDevice::Manufacturer [USB_STRING_MAXLEN]

Previous Top Next

Description

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

10.48 MaxPacketSize

UCHAR CCyUSBDevice::MaxPacketSize Previous Top Next

Description

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

10.49 MaxPower

UCHAR CCyUSBDevice::MaxPower Previous Top Next

Description

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

10.50 NtStatus

ULONG CCyUSBDevice::NtStatus Previous Top Next

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.

10.51 Open( )

bool CCyUSBDevice::Open(UCHAR dev**)** Previous Top Next

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( NULL);

//  Attempt to open device #0
if (USBDevice->DeviceCount() && !USBDevice->Open(0)) {
  // If Open() failed, retry Open() after resetting the device
  USBDevice->Reset();
  USBDevice->Open(0);
}

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

10.52 PowerState( )

UCHAR CCyUSBDevice::PowerState(void) Previous Top Next

This function is no longer supported. It is available to keep backward compatibility with legacy library and application.

10.53 Product

wchar_t CCyUSBDevice::Product [USB_STRING_MAXLEN]

Previous Top Next

Description

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

10.54 ProductID

USHORT CCyUSBDevice::ProductID Previous Top Next

Description

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

Example

```
//  Look for a device having VID = 04b4, PID = 00f0
CCyUSBDevice *USBDevice = new CCyUSBDevice( NULL);
int devices = USBDevice->DeviceCount();
int vID, pID;
int d = 0;

do {
  // Open automatically  calls Close() if necessary
  USBDevice->Open( d);
  vID = USBDevice->VendorID;
  pID  = USBDevice->ProductID;
  d++;
} while ((d < devices) && (vID != 0x04b4) && (pID != 0x00f0));
```

10.55 ReConnect( )

bool CCyUSBDevice::ReConnect(void**)** Previous Top Next

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.

10.56 Reset( )

bool CCyUSBDevice::Reset(void**)** Previous Top Next

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.

10.57 Resume( )

bool CCyUSBDevice::Resume(void**)** Previous Top Next

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

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

10.58 SerialNumber

wchar_t CCyUSBDevice::SerialNumber [USB_STRING_MAXLEN]

Previous Top Next

Description

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

10.59 SetConfig( )

void CCyUSBDevice::SetConfig(UCHAR cfg**)** Previous Top Next

Description

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

In general, devices only expose a single configuration. So, while this method exists for completeness; it should not need to be invoked with a **cfg** value other than 0.

10.60 SetAltIntfc( )

bool CCyUSBDevice::SetAltIntfc(UCHAR alt**)** Previous Top Next

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( NULL);
int lastIntfc = USBDevice->AltIntfcCount( );

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

10.61 StrLangID

USHORT CCyUSBDevice::StrLangID Previous Top Next

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).

10.62 Suspend( )

bool CCyUSBDevice::Suspend(void**)** Previous Top Next

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

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

10.63 USBAddress

UCHAR CCyUSBDevice::USBAddress Previous Top Next

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.

10.64 USBDIVersion

ULONG CCyUSBDevice::USBDIVersion Previous Top Next

Description

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

10.65 UsbdStatus

ULONG CCyUSBDevice::UsbdStatus Previous Top Next

Description

The UsbdStatus member contains the USBD_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.

10.66 UsbdStatusString( )

void CCyUSBDevice::UsbdStatusString(ULONG stat, PCHAR s**)** Previous Top Next

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.

10.67 VendorID

USHORT CCyUSBDevice::VendorID Previous Top Next

Description

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

Example

```
//  Look for a device having VID = 04b4, PID = 00f0
CCyUSBDevice *USBDevice = new CCyUSBDevice( NULL);
int devices = USBDevice->DeviceCount();
int vID, pID;
int d = 0;

do {
  // Open automatically  calls Close() if necessary
  USBDevice->Open( d);
  vID = USBDevice->VendorID;
  pID  = USBDevice->ProductID;
  d++;
} while ((d < devices) && (vID != 0x04b4) && (pID != 0x00f0));
```

11 CCyUSBConfig

CCyUSBConfig Class

Previous Top Next

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, each device has only one configuration.)

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 CCyUSBEndPoint for each endpoint descriptor contained in the interface descriptor. In this iterative fashion, the entire structure of Configurations->Interfaces->EndPoints gets populated from a single construction of the CCyUSBConfig class.

The following example code shows the use of the CCyUSBConfig class in an application.

Example

```
// This code snippet lists all the endpoints reported
// by the device under all interfaces and configurations.

CCyUSBDevice  *USBDevice = new  CCyUSBDevice(NULL);

char buf[512];
string s;

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

  // Print all configuration descriptor fields
  sprintf_s(buf,"bLength: 0x%x\n",cfg.bLength);
  s.append(buf);

  sprintf_s(buf,"bDescriptorType: %d\n",cfg.bDescriptorType);
  s.append(buf);

  sprintf_s(buf,"wTotalLength: %d (0x%x)\n",cfg.wTotalLength,cfg.
wTotalLength);
  s.append(buf);

  sprintf_s(buf,"bNumInterfaces: %d\n",cfg.bNumInterfaces);
  s.append(buf);
```

```
sprintf_s(buf,"bConfigurationValue: %d\n",cfg.bConfigurationValue);
s.append(buf);

sprintf_s(buf,"iConfiguration: %d\n",cfg.iConfiguration);
s.append(buf);

sprintf_s(buf,"bmAttributes: 0x%x\n",cfg.bmAttributes);
s.append(buf);

sprintf_s(buf,"MaxPower: %d\n",cfg.MaxPower);
s.append(buf);
s.append("**********************************\n");

cout<<s;
s.clear();

for (int i=0; i<cfg.AltInterfaces; i++)
{
  // Print all interface descriptor fields
  CCyUSBInterface  *ifc = cfg.Interfaces[i];
  sprintf_s(buf,"Interface  Descriptor:%d\n",(i+1));
  s.append(buf);

  sprintf_s(buf,"--------------------------------\n");
  s.append(buf);

  sprintf_s(buf,"bLength: 0x%x\n",ifc->bLength);
  s.append(buf);

  sprintf_s(buf,"bDescriptorType: %d\n",ifc->bDescriptorType);
  s.append(buf);

  sprintf_s(buf,"bInterfaceNumber: %d\n",ifc->bInterfaceNumber);
  s.append(buf);

  sprintf_s(buf,"bAlternateSetting: %d\n",ifc->bAlternateSetting);
  s.append(buf);

  sprintf_s(buf,"bNumEndpoints: %d\n",ifc->bNumEndpoints);
  s.append(buf);

  sprintf_s(buf,"bInterfaceClass: %d\n",ifc->bInterfaceClass);
  s.append(buf);
  sprintf_s(buf,"**********************************\n");s.append(buf);

  cout<<s;
  s.clear();

  for (int e=0; e<ifc->bNumEndpoints; e++)
  {
    // Print all endpoint descriptor fields
    CCyUSBEndPoint  *ept = ifc->EndPoints[e+1];
    sprintf_s(buf,"EndPoint  Descriptor:%d\n",(e+1));
```

```
            s.append(buf);

            sprintf_s(buf,"---------------------------------\n");
            s.append(buf);

            sprintf_s(buf,"bLength: 0x%x\n",ept->DscLen);
            s.append(buf);

            sprintf_s(buf,"bDescriptorType: %d\n",ept->DscType);
            s.append(buf);

            sprintf_s(buf,"bEndpointAddress: 0x%x\n",ept->Address);
            s.append(buf);

            sprintf_s(buf,"bmAttributes: 0x%x\n",ept->Attributes);
            s.append(buf);

            sprintf_s(buf,"wMaxPacketSize: %d\n",ept->MaxPktSize);
            s.append(buf);

            sprintf_s(buf,"bInterval: %d\n",ept->Interval);
            s.append(buf);
            s.append("*********************************\n");

            cout<<s;
            s.clear();
        }
    }
}
```

11.1 AltInterfaces

CCyUSBConfig::AltInterfaces Previous Top Next

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.

11.2 bConfigurationValue

UCHAR CCyUSBConfig::bConfigurationValue Previous Top Next

Description

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

11.3 bDescriptorType

UCHAR CCyUSBConfig::bDescriptorType Previous Top Next

Description

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

11.4 bLength

UCHAR CCyUSBConfig::bLength Previous Top Next

Description

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

11.5 bmAttributes

UCHAR CCyUSBConfig::bmAttributes Previous Top Next

Description

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

11.6 bNumInterfaces

UCHAR CCyUSBConfig::bNumInterfaces Previous Top Next

Description

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

11.7 CCyUSBConfig( )

CCyUSBConfig::CCyUSBConfig(void**)** Previous Top Next

Description

This is the default constructor for the CCyUSBConfig class.

This constructor simply sets all its data members to zero.

11.8 CCyUSBConfig( )

CCyUSBConfig::CCyUSBConfig(HANDLE handle, PUSB_CONFIGURATION_DESCRIPTOR pConfigDescr**)**

Previous Top Next

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.

11.9 CCyUSBConfig( )

CCyUSBConfig::CCyUSBConfig(CCyUSBConfig& cfg**)**

Previous Top Next

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( NULL);
CCyUSBConfig cfg = USBDevice->GetUSBConfig( 0);
```

11.10 ~CCyUSBConfig

CCyUSBConfig::~CCyUSBConfig(void) Previous Top Next

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.

11.11 iConfiguration

UCHAR CCyUSBConfig::iConfiguration Previous Top Next

Description

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

11.12 Interfaces

CCyUSBInterface* CCyUSBConfig::Interfaces [MAX_INTERFACES]

Previous Top Next

Description

Interfaces are 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 snippet lists all the endpoints reported
  // by the device under all interfaces and configurations.

  CCyUSBDevice  *USBDevice = new  CCyUSBDevice(NULL);

  char buf[512];
  string s;

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

    // Print all configuration descriptor fields
    sprintf_s(buf,"bLength: 0x%x\n",cfg.bLength);
    s.append(buf);

    sprintf_s(buf,"bDescriptorType: %d\n",cfg.bDescriptorType);
    s.append(buf);

    sprintf_s(buf,"wTotalLength: %d (0x%x)\n",cfg.wTotalLength,cfg.
wTotalLength);
    s.append(buf);

    sprintf_s(buf,"bNumInterfaces: %d\n",cfg.bNumInterfaces);
    s.append(buf);

    sprintf_s(buf,"bConfigurationValue: %d\n",cfg.bConfigurationValue);
    s.append(buf);

    sprintf_s(buf,"iConfiguration: %d\n",cfg.iConfiguration);
    s.append(buf);

    sprintf_s(buf,"bmAttributes: 0x%x\n",cfg.bmAttributes);
    s.append(buf);
```

```
sprintf_s(buf,"MaxPower:  %d\n",cfg.MaxPower);
s.append(buf);
s.append("*********************************\n");

cout<<s;
s.clear();

for (int i=0; i<cfg.AltInterfaces; i++)
{
  // Print all interface descriptor fields
  CCyUSBInterface  *ifc = cfg.Interfaces[i];
  sprintf_s(buf,"Interface  Descriptor:%d\n",(i+1));
  s.append(buf);

  sprintf_s(buf,"--------------------------------\n");
  s.append(buf);

  sprintf_s(buf,"bLength:  0x%x\n",ifc->bLength);
  s.append(buf);

  sprintf_s(buf,"bDescriptorType:  %d\n",ifc->bDescriptorType);
  s.append(buf);

  sprintf_s(buf,"bInterfaceNumber:  %d\n",ifc->bInterfaceNumber);
  s.append(buf);

  sprintf_s(buf,"bAlternateSetting:  %d\n",ifc->bAlternateSetting);
  s.append(buf);

  sprintf_s(buf,"bNumEndpoints:  %d\n",ifc->bNumEndpoints);
  s.append(buf);

  sprintf_s(buf,"bInterfaceClass:  %d\n",ifc->bInterfaceClass);
  s.append(buf);
  sprintf_s(buf,"*********************************\n");s.append(buf);

  cout<<s;
  s.clear();

  for (int e=0; e<ifc->bNumEndpoints; e++)
  {
    // Print all endpoint descriptor fields
    CCyUSBEndPoint  *ept = ifc->EndPoints[e+1];
    sprintf_s(buf,"EndPoint  Descriptor:%d\n",(e+1));
    s.append(buf);

    sprintf_s(buf,"--------------------------------\n");
    s.append(buf);

    sprintf_s(buf,"bLength:  0x%x\n",ept->DscLen);
    s.append(buf);

    sprintf_s(buf,"bDescriptorType:  %d\n",ept->DscType);
```

```
            s.append(buf);

            sprintf_s(buf,"bEndpointAddress: 0x%x\n",ept->Address);
            s.append(buf);

            sprintf_s(buf,"bmAttributes: 0x%x\n",ept->Attributes);
            s.append(buf);

            sprintf_s(buf,"wMaxPacketSize: %d\n",ept->MaxPktSize);
            s.append(buf);

            sprintf_s(buf,"bInterval: %d\n",ept->Interval);
            s.append(buf);
            s.append("**********************************\n");

            cout<<s;
            s.clear();
        }
    }
}
```

11.13 wTotalLength

USHORT CCyUSBConfig::wTotalLength Previous Top Next

Description

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

12 CCyUSBEndPoint

CCyUSBEndPoint Class

Previous Top Next

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 alternate 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.

12.1 Abort( )

void CCyUSBEndPoint::Abort(void) Previous Top Next

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( NULL);
USBDevice->ControlEndPt->Abort();
```

12.2 Address

UCHAR CCyUSBEndPoint::Address Previous Top Next

Description

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

Addresses with the high-order bit set (0x8Y) are IN endpoints.

Addresses with the high-order bit cleared (0x0Y) are OUT endpoints.

The default endpoint (ControlEndPt) has Address = 0.

Example

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

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

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

12.3 Attributes

UCHAR CCyUSBEndPoint::Attributes Previous Top Next

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( NULL);
CCyBulkEndPoint *BulkIn2 = NULL;
int eptCount = USBDevice->EndPointCount();

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

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

12.4 BeginDataXfer( )

virtual PUCHAR CCyUSBEndPoint::
BeginDataXfer(PCHAR buf, LONG len,
OVERLAPPED *ov**) = 0**

Previous Top Next

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(NULL);
  OVERLAPPED outOvLap, inOvLap;

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

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

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

  // 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);
```

12.5 bIn

bool CCyUSBEndPoint::bIn Previous Top Next

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( NULL);
CCyBulkEndPoint *BulkIn2 = NULL;
int eptCount = USBDevice->EndPointCount();

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

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

12.6 CCyUSBEndPoint( )

CCyUSBEndPoint::CCyUSBEndPoint(void**)** Previous Top Next

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.

12.7 CCyUSBEndPoint( )

CCyUSBEndPoint::CCyUSBEndPoint(HANDLE h, PUSB_ENDPOINT_DESCRIPTOR pEndPtDescriptor**)**

Previous Top Next

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, pEndPtDesc);
```

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 **pEndPtDescriptor** structure. It sets the default endpoint Timeout to 10 seconds. It sets its hDevice member to **h.**

12.8 CCyUSBEndPoint( )

CCyUSBEndPoint::CCyUSBEndPoint(CCyUSBEndPoint& ept**)**

Previous Top Next

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

12.9 DscLen

UCHAR CCyUSBEndPoint::DscLen Previous Top Next

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.

12.10 DscType

UCHAR CCyUSBEndPoint::DscType Previous Top Next

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.

12.11 GetXferSize( )

ULONG CCyUSBEndPoint::GetXferSize(void**)** Previous Top Next

Description

This function is no longer supported. It is available in the library to keep backward compatibility with legacy library and application.

For more information on USB transfer size please refer to the following link from Microsoft : http://msdn.microsoft.com/en-us/library/ff538112.aspx

Following is the maximum transfer size limit set into the CyUSB3.sys driver for various transfer types:

Bulk Transfers	4 MBytes
Interrupt Transfers	4 MBytes
Full Speed Isochronous Transfers	256 frames of data
High Speed and Super Speed Isochronous Transfers	1024 frames

12.12 FinishDataXfer( )

bool CCyUSBEndPoint::FinishDataXfer(PCHAR buf, LONG &len, OVERLAPPED *ov, PUCHAR pXmitBuf, CCyIsoPktInfo* pktInfos = **NULL)**

Previous Top Next

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 would have been returned by a previous corresponding call to BeginDataXfer.

The pointer to the 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 when present. 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(NULL);
  OVERLAPPED outOvLap, inOvLap;

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

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

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

  // 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);
```

12.13 hDevice

HANDLE CCyUSBEndPoint::hDevice Previous Top Next

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 endpont's hDevice member will be identical to its container CCyUSBDevice object's private hDevice member (accessed via the DeviceHandle( ) method).

12.14 Interval

UCHAR CCyUSBEndPoint::Interval Previous Top Next

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.

12.15 MaxPktSize

UCHAR CCyUSBEndPoint::MaxPktSize Previous Top Next

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.

For High Speed High Bandwidth endpoints, **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).

For USB 3.0 Bulk and Interrupt Endpoints, the **MaxPktSize** contains the value indicated by *wMaxPacketSize* field of the USB_ENDPOINT_DESCRIPTOR structure multiplied by the (1 + bMaxBurst) field of the Super Speed Companion descriptor.

For USB 3.0 Isochronous Endpoints, the **MaxPktSize** contains the value indicated by *wMaxPacketSize* field of the USB_ENDPOINT_DESCRIPTOR structure multiplied by the (1 + bMaxBurst) and (1 + bmAttributes) fields of the Super Speed Companion descriptor

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

Full Speed Device	
wMaxPacketSize = 0x03FF	1023 bytes
wMaxPacketSize = 0x0040	64 bytes
High Speed Device	
wMaxPacketSize = 0x0200	512 bytes
wMaxPacketSize = 0x0880	128 * (1 + 1) = 256 bytes
wMaxPacketSize = 0x1400	1024 * (1 + 2) = 3072 bytes
Super Speed Device	
Bulk, wMaxPacketSize = 0x0400, bMaxBurst = 0	1024 * (1 + 0) = 1024 bytes
Bulk, wMaxPacketSize = 0x0400, bMaxBurst = 0x07	1024 * (1 + 7) = 8192 bytes
Interrupt, wMaxPacketSize = 0x0400, bMaxBurst = 0x02	1024 * (1 + 2) = 3072 bytes
Isochronous, wMaxPacketSize = 0x0400, bMaxBurst = 0x03, bmAttributes = 0x02	1024 * (1 + 3) * (1 + 2) = 12288 bytes

12.16 NtStatus

ULONG CCyUSBEndPoint::NtStatus Previous Top Next

Description

NtStatus member contains the error code returned by the driver from the last call to the XferData, BeginDataXfer, FinishDataXFer or Abort methods.

This member has the same functionality as Windows System Error Code mapper, otherwise known by Win32 API name "GetLastError()". In a swiftly changing system environment, keeping a separate tracker to distinguish endpoint specific system errors from global system error is very helpful. So, this member capture endpoint specific NT status errors (if any exist).

This member will automatically be cleared by those library function calls which will in turn use windows driver API's for its intended functionality. So, it is recommended to refer to the NtStatus and UsbdStatus values to understand any failure returned by XferData or BeginDataXFer function calls.

NTStatus is a 32 bit numerical status defined by Microsoft. Please see the MSDN documentation for NTStatus for a description of each NTStatus value.

Please refer to the UsbdStatus member for more information on the USB related errors.

12.17 Reset( )

bool CCyUSBEndPoint::Reset(void**)** Previous Top Next

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 to force completion of any transfers in-process.

12.18 SetXferSize( )

void CCyUSBEndPoint::SetXferSize(ULONG xfer **)**

Previous Top Next

Description

This function is no longer supported. It is available to keep backward compatibility with legacy library and application.

For more information on USB transfer size, please refer to the following link from Microsoft: http://msdn.microsoft.com/en-us/library/ff538112.aspx

Following is the maximum transfer size limit set into the CyUSB3.sys driver for various transfer types:

Bulk Transfers	4 MBytes
Interrupt Transfers	4 MBytes
Full Speed Isochronous Transfers	256 frames of data
High Speed and Super Speed Isochronous Transfers	1024 frames

12.19 TimeOut

ULONG CCyUSBEndPoint::TimeOut Previous Top Next

Description

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

The unit 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.

Set the TimeOut values to 0xFFFFFFFF(INFINITE), to wait for infinite time on any transfer (Bulk, Isochronous, Interrupt or Control).

When TimeOut is set to 0 for bulk, interrupt or isochronous transfers; XferData does not wait for read/write operation to complete and will return immediately.

The TimeOut value 0 for control transfer is rounded up to 1000ms.

The default TimeOut for Bulk,Interrupt, Control, and Isochronous transfer is 10 seconds. User can override this value depending upon their application needs.

Example

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

if ( USBDevice->BulkOutEndPt) {
  unsigned char buf[128];
  LONG length = 128;

  // Set the timeout to 1 second
  USBDevice->BulkOutEndPt->TimeOut = 1000;
  USBDevice->BulkOutEndPt->XferData( buf, length);
}
```

12.20 UsbdStatus

ULONG CCyUSBEndPoint::UsbdStatus Previous Top Next

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.

Any failure in library function calls that will end-up performing an active USB transaction, will set this member to a value returned by Microsoft USBD Stack. This member will carry information about the USB failure occurred from the USB Host perspective. This member together with NtStatus provides detailed description about any functionality failure coming from the library APIs such as XferData, BeginDataXfer, FinishDataXfer etc.

This member will be cleared by any library APIs which initiate a new USB transaction.

12.21 WaitForXfer( )

bool CCyUSBEndPoint::WaitForXfer(
OVERLAPPED *ov, ULONG tOut**)**

Previous Top Next

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.

tOut 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(NULL);
  OVERLAPPED outOvLap, inOvLap;

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

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

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

  // 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);
```

12.22 XferData( )

bool CCyUSBEndPoint::XferData(PUCHAR buf, LONG &bufLen, CCyIsoPktInfo* pktInfos)

Previous Top Next

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. It calls Abort() method internally if the operation fails.

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.

Please refer XferData for Isochronous transfer for the usage of the XferData for isochronous transfers.

The code sample below demonstrates the usage of XferData() api for bulk and interrupt transfers.

Example

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

if ( USBDevice->BulkOutEndPt) {
  unsigned char buf[ 128];
  LONG length = 128;

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

12.23 ssdscLen

UCHAR CCyUSBEndPoint::ssdscLen Top Previous Next

Description

ssdscLen contains the length of the superspeed endpoint companion descriptor as reported in the *bLength* field of the USB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR structure that was passed to the endpoint object's constructor.

Because the passed descriptor was an endpoint descriptor, this value should always be 0x06. This data member exists for completeness and debugging purposes. You should normally never need to access this data member.

12.24 ssdscType

UCHAR CCyUSBEndPoint::ssdsType Top Previous Next

Description

ssdscType contains the type of the superspeed endpoint companion descriptor as reported in the bDescriptorType field of the USB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR structure that was passed to the endpoint object's constructor.

Because the passed descriptor was an endpoint descriptor, this value should always be 0x30. This data member exists for completeness and debugging purposes. You should normally never need to access this data member.

12.25 ssmaxburst

UCHAR CCyUSBEndPoint::ssmaxburst Top Previous Next

Description

ssmaxburst contains the value indicated by the bMaxBurst field of the USB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR structure that was passed to the endpoint object's constructor.

The ssmaxburst represent the maximum number of packets the endpoint can send or receive as part of a burst. Valid values are from 0 to 15. A value of 0 indicates that the endpoint can only burst one packet at a time and a value of 16 indicates that the endpoint can burst up to 16 packets at a time.

For endpoint of type control, this shall be set to 0.

12.26 ssbmAttribute

UCHAR CCyUSBEndPoint::ssbmAttribute Top Previous Next

Description

ssbmAttribute contains the value indicated by the bmAttributes field of the USB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR structure that was passed to the endpoint object's constructor.

ssbmAttribute represent different information based on the type of endpoint.

Bulk Endpoint	
Bits 4:0	MaxStream, the maximum number of streams supported by the endpoint. Valid values are from 0 to 15, where 0 indicates that the endpoint does not support Streams. For other values, the number of streams supported will be power (2, MaxStream). Note: The CyUsb3.sys driver and the CyAPI library do not support data transfers from/to stream enabled bulk endpoints.
Bits 7:5	Reserved. Shall be set to 0.
Control and Interrupt Endpoints	
Bits 7:0	Reserved. Shall be set to 0.
Isochronous Endpoints	
Bits 1:0	Mult. A zero based value that determines the number of burst transfers that the endpoint can support within a service interval.
Bits 7:2	Reserved. Shall be set to 0.

12.27 ssbytesperinterval

USHORT CCyUSBEndPoint::ssbyteperinterval Top Previous Next

Description

ssbyteperinterval contains the value indicated by the wBytesPerInterval field of the USB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR structure that was passed to the endpoint object's constructor.

For periodic (Isochronous or Interrupt) endpoints, this value represents the total number of bytes the endpoint will transfer every service interval.

For an Isochronous endpoint, this value is used to reserve the bus time in the schedule, required for the frame data payloads per 125us. The pipe may on an ongoing basis, actually use less bandwidth that that was reserved. If necessary, the device can report the actual bandwidth used via a non-USB defined mechanism.

13 CCyUSBInterface

CCyUSBInterface Class

Previous Top Next

Header
CyUSB.h

Description

CCyUSBInterface represents a USB device interface. Such interfaces can 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. (In most cases, there is just one configuration per device.)

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 CCyUSBEndPoint 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 the usage of the CCyUSBInterface class in an application.

Example

```
// This code snippet lists all the endpoints reported
// by the device under all interfaces and configurations.

CCyUSBDevice  *USBDevice = new  CCyUSBDevice(NULL);

char buf[512];
string s;

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

  // Print all configuration descriptor fields
  sprintf_s(buf,"bLength: 0x%x\n",cfg.bLength);
  s.append(buf);

  sprintf_s(buf,"bDescriptorType: %d\n",cfg.bDescriptorType);
  s.append(buf);

  sprintf_s(buf,"wTotalLength: %d (0x%x)\n",cfg.wTotalLength,cfg.
wTotalLength);
  s.append(buf);

  sprintf_s(buf,"bNumInterfaces: %d\n",cfg.bNumInterfaces);
  s.append(buf);
```

```
sprintf_s(buf,"bConfigurationValue: %d\n",cfg.bConfigurationValue);
s.append(buf);

sprintf_s(buf,"iConfiguration: %d\n",cfg.iConfiguration);
s.append(buf);

sprintf_s(buf,"bmAttributes: 0x%x\n",cfg.bmAttributes);
s.append(buf);

sprintf_s(buf,"MaxPower: %d\n",cfg.MaxPower);
s.append(buf);
s.append("**********************************\n");

cout<<s;
s.clear();

for (int i=0; i<cfg.AltInterfaces; i++)
{
  // Print all interface descriptor fields
  CCyUSBInterface  *ifc = cfg.Interfaces[i];
  sprintf_s(buf,"Interface  Descriptor:%d\n",(i+1));
  s.append(buf);

  sprintf_s(buf,"--------------------------------\n");
  s.append(buf);

  sprintf_s(buf,"bLength: 0x%x\n",ifc->bLength);
  s.append(buf);

  sprintf_s(buf,"bDescriptorType: %d\n",ifc->bDescriptorType);
  s.append(buf);

  sprintf_s(buf,"bInterfaceNumber: %d\n",ifc->bInterfaceNumber);
  s.append(buf);

  sprintf_s(buf,"bAlternateSetting: %d\n",ifc->bAlternateSetting);
  s.append(buf);

  sprintf_s(buf,"bNumEndpoints: %d\n",ifc->bNumEndpoints);
  s.append(buf);

  sprintf_s(buf,"bInterfaceClass: %d\n",ifc->bInterfaceClass);
  s.append(buf);
  sprintf_s(buf,"**********************************\n");s.append(buf);

  cout<<s;
  s.clear();

  for (int e=0; e<ifc->bNumEndpoints; e++)
  {
    // Print all endpoint descriptor fields
    CCyUSBEndPoint  *ept = ifc->EndPoints[e+1];
    sprintf_s(buf,"EndPoint  Descriptor:%d\n",(e+1));
```

```
            s.append(buf);

            sprintf_s(buf,"---------------------------------\n");
            s.append(buf);

            sprintf_s(buf,"bLength: 0x%x\n",ept->DscLen);
            s.append(buf);

            sprintf_s(buf,"bDescriptorType: %d\n",ept->DscType);
            s.append(buf);

            sprintf_s(buf,"bEndpointAddress: 0x%x\n",ept->Address);
            s.append(buf);

            sprintf_s(buf,"bmAttributes: 0x%x\n",ept->Attributes);
            s.append(buf);

            sprintf_s(buf,"wMaxPacketSize: %d\n",ept->MaxPktSize);
            s.append(buf);

            sprintf_s(buf,"bInterval: %d\n",ept->Interval);
            s.append(buf);
            s.append("*********************************\n");

            cout<<s;
            s.clear();
        }
    }
}
```

13.1 bAlternateSetting

UCHAR CCyUSBInterface::bAlternateSetting Previous Top Next

Description

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

13.2 bAltSettings

UCHAR CCyUSBInterface::bAltSettings Previous Top Next

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.

See CCyUSBDevice::AltIntfcCount( ).

13.3 bDescriptorType

UCHAR CCyUSBInterface::bDescriptorType Previous Top Next

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.

13.4 CCyUSBInterface( )

CCyUSBInterface::CCyUSBInterface:(HANDLE h, PUSB_INTERFACE_DESCRIPTOR pIntfcDescriptor**)**

Previous Top Next

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' EndPoints array.

13.5 CCyUSBInterface( )

CCyUSBInterface::CCyUSBInterface:(
CCyUSBInterface& intfc**)**

Previous Top Next

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( NULL);
CCyUSBConfig cfg = USBDevice->GetUSBConfig( 0);
CCyUSBInterface *iface = new CCyUSBInterface( *cfg. Interfaces[ 0]);
```

13.6 bInterfaceClass

UCHAR CCyUSBInterface::bInterfaceClass Previous Top Next

Description

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

13.7 bInterfaceNumber

UCHAR CCyUSBInterface::bInterfaceNumber Previous Top Next

Description

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

13.8 bInterfaceProtocol

UCHAR CCyUSBInterface::bInterfaceProtocol Previous Top Next

Description

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

13.9 bInterfaceSubClass

UCHAR CCyUSBInterface::bInterfaceSubClass Previous Top Next

Description

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

13.10 bLength

UCHAR CCyUSBInterface::bLength Previous Top Next

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.

13.11 bNumEndpoints

UCHAR CCyUSBInterface::bNumEndpoints Previous Top Next

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.

13.12 EndPoints

CCyUSBEndPoint* CCyUSBInterfac::EndPoints[MAX_ENDPTS]

Previous Top Next

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 snippet lists all the endpoints reported
  // by the device under all interfaces and configurations.

  CCyUSBDevice  *USBDevice = new  CCyUSBDevice(NULL);

  char buf[512];
  string s;

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

    // Print all configuration descriptor fields
    sprintf_s(buf,"bLength: 0x%x\n",cfg.bLength);
    s.append(buf);

    sprintf_s(buf,"bDescriptorType: %d\n",cfg.bDescriptorType);
    s.append(buf);

    sprintf_s(buf,"wTotalLength: %d (0x%x)\n",cfg.wTotalLength,cfg.
wTotalLength);
    s.append(buf);

    sprintf_s(buf,"bNumInterfaces: %d\n",cfg.bNumInterfaces);
    s.append(buf);

    sprintf_s(buf,"bConfigurationValue: %d\n",cfg.bConfigurationValue);
    s.append(buf);

    sprintf_s(buf,"iConfiguration: %d\n",cfg.iConfiguration);
    s.append(buf);

    sprintf_s(buf,"bmAttributes: 0x%x\n",cfg.bmAttributes);
    s.append(buf);

    sprintf_s(buf,"MaxPower: %d\n",cfg.MaxPower);
    s.append(buf);
```

```
s.append("*********************************\n");

cout<<s;
s.clear();

for (int i=0; i<cfg.AltInterfaces; i++)
{
  // Print all interface descriptor fields
  CCyUSBInterface  *ifc = cfg.Interfaces[i];
  sprintf_s(buf,"Interface  Descriptor:%d\n",(i+1));
  s.append(buf);

  sprintf_s(buf,"--------------------------------\n");
  s.append(buf);

  sprintf_s(buf,"bLength: 0x%x\n",ifc->bLength);
  s.append(buf);

  sprintf_s(buf,"bDescriptorType: %d\n",ifc->bDescriptorType);
  s.append(buf);

  sprintf_s(buf,"bInterfaceNumber: %d\n",ifc->bInterfaceNumber);
  s.append(buf);

  sprintf_s(buf,"bAlternateSetting: %d\n",ifc->bAlternateSetting);
  s.append(buf);

  sprintf_s(buf,"bNumEndpoints: %d\n",ifc->bNumEndpoints);
  s.append(buf);

  sprintf_s(buf,"bInterfaceClass: %d\n",ifc->bInterfaceClass);
  s.append(buf);
  sprintf_s(buf,"*********************************\n");s.append(buf);

  cout<<s;
  s.clear();

  for (int e=0; e<ifc->bNumEndpoints; e++)
  {
    // Print all endpoint descriptor fields
    CCyUSBEndPoint  *ept = ifc->EndPoints[e+1];
    sprintf_s(buf,"EndPoint  Descriptor:%d\n",(e+1));
    s.append(buf);

    sprintf_s(buf,"--------------------------------\n");
    s.append(buf);

    sprintf_s(buf,"bLength: 0x%x\n",ept->DscLen);
    s.append(buf);

    sprintf_s(buf,"bDescriptorType: %d\n",ept->DscType);
    s.append(buf);
```

```
            sprintf_s(buf,"bEndpointAddress:  0x%x\n",ept->Address);
            s.append(buf);

            sprintf_s(buf,"bmAttributes:  0x%x\n",ept->Attributes);
            s.append(buf);

            sprintf_s(buf,"wMaxPacketSize:  %d\n",ept->MaxPktSize);
            s.append(buf);

            sprintf_s(buf,"bInterval:  %d\n",ept->Interval);
            s.append(buf);
            s.append("**********************************\n");

            cout<<s;
            s.clear();
        }
    }
}
```

13.13 iInterface

UCHAR CCyUSBInterface::iInterface Previous Top

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.

14 CCyUSBBOS

CCyUSBBOS class

Top Previous Next

Header
CyAPI.h

Description

CCyUSBBOS represents a USB 3.0 device BOS descriptor.

When CCyUSBDevice::Open( ) is called, an instance of CCyUSBBOS is constructed if the current device is a USB 3.0 device.

In the process of construction, CCyUSBBOS creates instances for each capability. If device does not define specific capability in the BOS descriptor, then the valued of the instance will be NULL.

The capability types supported are: CCyBOS_USB20_DEVICE_EXT, CCyBOS_SS_DEVICE_CAPABILITY and CCyBOS_CONTAINER_ID.

The following example code shows how you can use the CCyUSBBOS class in an application.

Example

```
// This code  lists the BOS device capability descriptor
char buf[512];
string s;
CCyUSBDevice *USBDevice = new  CCyUSBDevice(NULL);
CCyUSBBOS *bos = USBDevice->UsbBos;

sprintf_s(buf,"BOS  Descriptor");
s.append(buf);
sprintf_s(buf,"---------------------------------");
s.append(buf);
sprintf_s(buf,"bLength: 0x%x",bos->bLength);
s.append(buf);
sprintf_s(buf,"bDescriptorType: %d",bos->bDescriptorType);
s.append(buf);
sprintf_s(buf,"wTotalLength: %d",bos->wTotalLength);
s.append(buf);
sprintf_s(buf,"bNumDeviceCaps: %d",bos->bNumDeviceCaps);
s.append(buf);
sprintf_s(buf,"*********************************");
s.append(buf);

cout<<s;
s.clear();

// Print the USB 2.0 Device Extension descriptor if found
if(bos->pUSB20_DeviceExt)
{
```

```
    CCyBosUSB20Extesnion *Usb20Ext = bos->pUSB20_DeviceExt;

    sprintf_s(buf,"USB20 Device Extension Descriptor");
    s.append(buf);
    sprintf_s(buf,"--------------------------------");
    s.append(buf);
    sprintf_s(buf,"bLength: 0x%x",Usb20Ext->bLength);
    s.append(buf);
    sprintf_s(buf,"bDescriptorType: %d",Usb20Ext->bDescriptorType);
    s.append(buf);
    sprintf_s(buf,"bDevCapabilityType: %d",Usb20Ext->bDevCapabilityType);
    s.append(buf);
    sprintf_s(buf,"bmAttribute: %d",Usb20Ext->bmAttribute);
    s.append(buf);
    sprintf_s(buf,"********************************");
    s.append(buf);

    cout<<s;
    s.clear();
  }

  // Print the SuperSpeed Device Capability descriptor if found
  if(bos->pSS_DeviceCap)
  {
    CCyBosSuperSpeedCapability  *ssCap = bos->pSS_DeviceCap;

    sprintf_s(buf,"Super Speed Device capability Descriptor");
    s.append(buf);
    sprintf_s(buf,"--------------------------------");
    s.append(buf);
    sprintf_s(buf,"bLength: 0x%x",ssCap->bLength);
    s.append(buf);
    sprintf_s(buf,"bDescriptorType: %d",ssCap->bDescriptorType);
    s.append(buf);
    sprintf_s(buf,"bDevCapabilityType: %d",ssCap->bDevCapabilityType);
    s.append(buf);
    sprintf_s(buf,"bmAttribute: %d",ssCap->bmAttribute);
    s.append(buf);
    sprintf_s(buf,"SpeedsSuported: %d",ssCap->SpeedsSuported);
    s.append(buf);
    sprintf_s(buf,"bFunctionalitySupported: %d",ssCap-
>bFunctionalitySupporte);
    s.append(buf);
    sprintf_s(buf,"bU1DevExitLat: %d",ssCap->bU1DevExitLat);
    s.append(buf);
    sprintf_s(buf,"bU2DevExitLat: %d",ssCap->bU2DevExitLat);
    s.append(buf);
    sprintf_s(buf,"********************************");
    s.append(buf);

    cout<<s;
    s.clear();
  }
```

```
// Print the Container ID descriptor if found
if(bos->pContainer_ID)
{
  CCyBosContainerID *ContID = bos->pContainer_ID;

  sprintf_s(buf,"Container ID Descriptor");
  s.append(buf);
  sprintf_s(buf,"--------------------------------");
  s.append(buf);
  sprintf_s(buf,"bLength: 0x%x",ContID->bLength);
  s.append(buf);
  sprintf_s(buf,"bDescriptorType: %d",ContID->bDescriptorType);
  s.append(buf);
  sprintf_s(buf,"bDevCapabilityType: %d",ContID->bDevCapabilityType);
  s.append(buf);
  sprintf_s(buf,"bReserved: %d",ContID->bReserved);
  s.append(buf);
  sprintf_s(buf,"ContainerID: %s",ContID->ContainerID);
  s.append(buf);
  sprintf_s(buf,"********************************");
  s.append(buf);

  cout<<s;
  s.clear();
}
```

14.1 pContainer_ID

CCyBosContainerID *pContainer_ID Top Previous Next

Description

pContainer ID is a CCyBOSContainerID object representing the USB 3.0 device container ID of BOS. It can be NULL if the device does not define this capability, or if the device is a USB 2.0 device.

14.2 pUSB20_DeviceExt

CCyBosUSB20Extesnion *pUSB20_DeviceExt Top Previous Next

Description

pUSB20_DeviceExt is a CCyBosUSB20Extesnion object that represents the USB 2.0 device extension capability of BOS. It can be null, if the device does not define this capability or if the device is a USB 2.0 device.

14.3 pSS_DeviceCap

CCyBosSuperSpeedCapability *pSS_DeviceCap

Top Previous Next

Description

pSS_DeviceCap is a CCyBosSuperSpeedCapability object representing the USB3.0 device Super Speed capability of BOS. It can be NULL, if the device does not define this capability, or if the device is a USB 2.0 device.

14.4 bLength

UCHAR bLength Top Previous Next

Description

This property returns the length of BOS descriptor.

14.5 bDescriptorType

UCHAR bDescriptorType Top Previous Next

Description

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

14.6 wTotalLength

USHORT wTotalLength Top Previous Next

Description

wTotalLength contains value of the wTotalLength field from the selected BOS descriptor.

14.7 bNumDeviceCaps

UCHAR bNumDeviceCaps Top Previous Next

Description

bNumDeviceCaps contains value of the **bNumberOfDeviceCapability** field from the selected BOS descriptor.

15 CCyBOSUSB20Extesnion

CCyBOSUSB20Extesnion class Top Previous Next

Header
CyAPI.h

Description

CCyBOSUSB20Extesnion represents the USB 2.0 device extension capability descriptor of a USB 3.0 device

If the device defines the USB 2.0 device extension capability, then an instance of this class will be instantiated by the CCyUSBBOS class.

The following example code shows usage of the CCyUSBBOS class and CCyBOSUSB20Extesnion class in an application.

Example

```
// This code  lists the BOS device capability descriptor
char buf[512];
string s;
CCyUSBDevice *USBDevice = new  CCyUSBDevice(NULL);
CCyUSBBOS *bos = USBDevice->UsbBos;

// Print the USB 2.0 Device Extension descriptor if found
if(bos->pUSB20_DeviceExt)
{
  CCyBosUSB20Extesnion *Usb20Ext = bos->pUSB20_DeviceExt;

  sprintf_s(buf,"USB20 Device Extension Descriptor");
  s.append(buf);
  sprintf_s(buf,"--------------------------------");
  s.append(buf);
  sprintf_s(buf,"bLength: 0x%x",Usb20Ext->bLength);
  s.append(buf);
  sprintf_s(buf,"bDescriptorType: %d",Usb20Ext->bDescriptorType);
  s.append(buf);
  sprintf_s(buf,"bDevCapabilityType: %d",Usb20Ext->bDevCapabilityType);
  s.append(buf);
  sprintf_s(buf,"bmAttribute: %d",Usb20Ext->bmAttribute);
  s.append(buf);
  sprintf_s(buf,"**********************************");
  s.append(buf);

  cout<<s;
  s.clear();
}
```

15.1 bLength

UCHAR bLength Top Previous Next

Description

bLength contains the value of bLength field of USB 2.0 Device extension descriptor.

15.2 bDescriptorType

UCHAR bDescriptorType Top Previous Next

Description

bDescriptorType contains the value of bDescriptorType field of USB 2.0 Device extension descriptor.

15.3 bDevCapabilityType

UCHAR bDevCapabilityType Top Previous Next

Description

bDevCapabilityType contains the value of bDevCapabilityType field of USB 2.0 Device extension descriptor.

15.4 bmAttribute

UINT bmAttribute Top Previous Next

Description

bmAttribute contains the value of bmAttribute field of USB 2.0 Device extension descriptor.

16 CCyBOSSuperSpeedCapability

CCyBOSSuperSpeedCapability class Top Previous Next

Header
CyAPI.h

Description

CCyBosSuperSpeedCapability represents the Super Speed device capability descriptor of a USB 3.0 device

If the device defines the SS device capability, then it will be instantiated by the CCyUSBBOS class.

The following example code shows the usage of the CCyBOSSuperSpeedCapability class in an application.

Examples

```
  // This code  lists the BOS device capability descriptor
  char buf[512];
  string s;
  CCyUSBDevice *USBDevice = new  CCyUSBDevice(NULL);
  CCyUSBBOS *bos = USBDevice->UsbBos;

  // Print the SuperSpeed Device Capability descriptor if found
  if(bos->pSS_DeviceCap)
  {
    CCyBosSuperSpeedCapability *ssCap = bos->pSS_DeviceCap;

    sprintf_s(buf,"Super Speed Device capability Descriptor");
    s.append(buf);
    sprintf_s(buf,"-------------------------------");
    s.append(buf);
    sprintf_s(buf,"bLength: 0x%x",ssCap->bLength);
    s.append(buf);
    sprintf_s(buf,"bDescriptorType: %d",ssCap->bDescriptorType);
    s.append(buf);
    sprintf_s(buf,"bDevCapabilityType: %d",ssCap->bDevCapabilityType);
    s.append(buf);
    sprintf_s(buf,"bmAttribute: %d",ssCap->bmAttribute);
    s.append(buf);
    sprintf_s(buf,"SpeedsSuported: %d",ssCap->SpeedsSuported);
    s.append(buf);
    sprintf_s(buf,"bFunctionalitySupported: %d",ssCap-
>bFunctionalitySupporte);
    s.append(buf);
    sprintf_s(buf,"bU1DevExitLat: %d",ssCap->bU1DevExitLat);
    s.append(buf);
    sprintf_s(buf,"bU2DevExitLat: %d",ssCap->bU2DevExitLat);
    s.append(buf);
    sprintf_s(buf,"*********************************");
```

```
    s.append(buf);

    cout<<s;
    s.clear();
  }
```

16.1 bLength

UCHAR bLength Top Previous Next

Description

Length contains the value of bLength field of SS Device capability descriptor.

16.2 bDescriptorType

UCHAR bDescriptorType Top Previous Next

Description

bDescriptorType contains the value of bDescriptorType field of SS Device capability descriptor.

16.3 bDevCapabilityType

UCHAR bDevCapabilityType Top Previous Next

Description

bDevCapabilityType contains the value of bDevCapabilityType field of SS Device capability descriptor.

16.4 bmAttribute

UCHAR bmAttribute Top Previous Next

Description

bmAttribute contains the value of bmAttribute field of SS Device capability descriptor.

16.5 SpeedsSuported

USHORT SpeedsSuported Top Previous Next

Description

SpeedsSuported contains the value of wSpeedsSupported field of SS Device capability descriptor.

16.6 bFunctionalitySupport

UCHAR bFunctionalitySupport Top Previous Next

Description

bFunctionalitySupport contains the value of bFunctionalitySupport field of SS Device capability descriptor.

16.7 bU1DevExitLat

UCHAR bU1DevExitLat Top Previous Next

Description

U1DevExitLat contains the value of U1DevExitLat field of SS Device capability descriptor.

16.8 bU2DevExitLat

USHORT bU2DevExitLat Top Previous Next

Description

bU2DevExit contains the value of bU2DevExit field of SS Device capability descriptor.

17 CCyBOSContainerID

CCyBOSContainerID class Top Previous Next

Header
CyAPI.h

Description

CCyBOSContainerID represents a USB 3.0 device container id descriptor.

If the device defines the SS device capability, then it will be instantiated by the CCyUSBBOS class.

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

Examples

```
// This code  lists the BOS device capability descriptor
char buf[512];
string s;
CCyUSBDevice *USBDevice = new  CCyUSBDevice(NULL);
CCyUSBBOS *bos = USBDevice->UsbBos;

// Print the Container ID descriptor if found
if(bos->pContainer_ID)
{
  CCyBosContainerID *ContID = bos->pContainer_ID;

  sprintf_s(buf,"Container ID Descriptor");
  s.append(buf);
  sprintf_s(buf,"--------------------------------");
  s.append(buf);
  sprintf_s(buf,"bLength:  0x%x",ContID->bLength);
  s.append(buf);
  sprintf_s(buf,"bDescriptorType:  %d",ContID->bDescriptorType);
  s.append(buf);
  sprintf_s(buf,"bDevCapabilityType:  %d",ContID->bDevCapabilityType);
  s.append(buf);
  sprintf_s(buf,"bReserved:  %d",ContID->bReserved);
  s.append(buf);
  sprintf_s(buf,"ContainerID:  %s",ContID->ContainerID);
  s.append(buf);
  sprintf_s(buf,"********************************");
  s.append(buf);

  cout<<s;
  s.clear();
}
```

17.1 bLength

UCHAR bLength Top Previous Next

Description

bLength contains the value of bLength field of Container ID descriptor.

17.2 bDescriptorType

UCHAR bDescriptorType Top Previous Next

Description

bDescriptorType contains the value of bDescriptorType field of Container ID descriptor.

17.3 bDevCapabilityType

UCHAR bDevCapabilityType Top Previous Next

Description

bDevCapabilityType contains the value of bDevCapabilityType field of Container ID descriptor.

17.4 bReserved

UCHAR bReserved Top Previous Next

Description

Reserved field of Container ID descriptor.

17.5 ContainerID

UCHAR [] ContainerID Top Previous Next

Description

ContainerID contains the value of ContainerID field of Container ID descriptor.

18 USB_BOS_USB20_DEVICE_EXTENSION

struct _USB_BOS_USB20_DEVICE_EXTENSION
defined in CyUSB30_def.h

Top Previous Next

Description

The USB_BOS_USB20_DEVICE_EXTENSION structure is filled-in by the GetBosUSB20DeviceExtensionDesc method of CCyUSBDevice. The structure is defined as:

```
typedef struct _USB_BOS_USB20_DEVICE_EXTENSION
{
  UCHAR bLength;
  UCHAR bDescriptorType;
  UCHAR bDevCapabilityType;
  UINT bmAttribute;
}USB_BOS_USB20_DEVICE_EXTENSION, *PUSB_BOS_USB20_DEVICE_EXTENSION;
```

Please refer to USB 3.0 specification section 9.6.2.1 for detailed description of each parameter.

19 USB_BOS_SS_DEVICE_CAPABILITY

struct _USB_BOS_SS_DEVICE_CAPABILITY
defined in CyUSB30_def.h

Top Previous Next

Description

The USB_BOS_SS_DEVICE_CAPABILITY structure is filled-in by the GetBosSSCapabilityDescriptor method of CCyUSBDevice.

The structure is defined as:

```
typedef struct _USB_BOS_SS_DEVICE_CAPABILITY
{
  UCHAR bLength;
  UCHAR bDescriptorType;
  UCHAR bDevCapabilityType;
  UCHAR bmAttribute;
  USHORT wSpeedsSuported;
  UCHAR bFunctionalitySupported;
  UCHAR bU1DevExitLat;
  USHORT bU2DevExitLat;
}USB_BOS_SS_DEVICE_CAPABILITY, *PUSB_BOS_SS_DEVICE_CAPABILITY;
```

Please refer to USB 3.0 specification section 9.6.2.2 for detailed description of each parameter.

20 USB_BOS_CONTAINER_ID

struct _USB_BOS_CONTAINER_ID
defined in CyUSB30_def.h

Top Previous Next

Description

The USB_BOS_CONTAINER_ID structure is filled-in by the GetBosContainedIDDescriptor method of CCyUSBDevice.

The structure is defined as:

```
typedef struct _USB_BOS_CONTAINER_ID
{
  UCHAR bLength;
  UCHAR bDescriptorType;
  UCHAR bDevCapabilityType;
  UCHAR bReserved;
  UCHAR ContainerID[USB_BOS_CAPABILITY_TYPE_CONTAINER_ID_SIZE];
}USB_BOS_CONTAINER_ID,*PUSB_BOS_CONTAINER_ID;
```

Please refer to USB 3.0 specification section 9.6.2.3 for detailed description of each parameter.

21 USB_BOS_DESCRIPTOR

struct _USB_BOS_DESCRIPTOR
defined in CyUSB30_def.h

Top Previous Next

Description

The USB_BOS_DESCRIPTOR structure is filled-in by the GetBosDescriptor method of CCyUSBDevice.

The structure is defined as:

```
typedef struct _USB_BOS_DESCRIPTOR
{
  UCHAR bLength;
  UCHAR bDescriptorType;
  USHORT wTotalLength;
  UCHAR bNumDeviceCaps;
}USB_BOS_DESCRIPTOR, *PUSB_BOS_DESCRIPTOR;
```

Please refer to USB 3.0 specification section 9.6.2 for detailed description of each parameter.

22 FX3_FWDWNLOAD_MEDIA_TYPE

enum FX3_FWDWNLOAD_MEDIA_TYPE Top Previous Next

Description

This enum defines the types of media to which FX3 firmware can be downloaded through the USB link. The following media types are defined:

RAM	- Download firmware to Ram.
I2CE2PROM	- Download firmware to I2C E2PROM.
SPIFLASH	- Download firmware to SPI FLASH.

23 FX3_FWDWNLOAD_ERROR_CODE

enum FX3_FWDWNLOAD_ERROR_CODE Top Previous Next

Description

This type defines status codes that can be returned by the DownloadFw method of the CCyFX3Device class.

Status Code	Description
SUCCESS	Firmware download successful
FAILED	Firmware download failed
INVALID_FILE	Invalid file
INVALID_MEDIA_TYPE	Given Input Media type is not supported
INVALID_FWSIGNATURE	Invalid Firmware Signature
DEVICE_CREATE_FAILED	Device Open failed
INCORRECT_IMAGE_LENGTH	Firmware image length is incorrect
SPILASH_ERASE_FAILED	SPI FLASH erase operation failed
I2CE2PROM_UNKNOWN_I2C_SIZE	Unknown I2CE2PROM size: unknown value parsed from 2nd byte of IMG file
CORRUPT_FIRMWARE_IMAGE_FILE	Corrupt Firmware image file

24 How to Link with CyAPI.lib

Please follow the below steps to add CyAPI.lib to your project:

1 Adding CyAPI.h header file path to your project
1.1 Select Project properties
1.2 Select the "General" node under "C/C++"
1.3 Add the path to the header files (for example, "C:\Program Files (x86)\Cypress\EZ-USB FX3 SDK\1.3\library\cpp\inc") under "Additional Include Directories".
2 Linking CyAPI.lib
2.1 Select Project properties.
2.2 Select 'Linker' node under the 'Configuration Properties'.
2.3 Select the 'Input' node under the 'Linker'.
2.4 Add lib path (including the lib name, for example, "C:\Program Files (x86)\Cypress\EZ-USB FX3 SDK\1.3\library\cpp\lib\x86\cyapi.lib") in the 'Additional Dependencies' edit box. Libraries for 32/64 bit available in the CySuiteUSB installation directory "library\cpp\lib". The directory 'x64' is for 64-bit library and the 'x86' directory is for 32-bit library.
3 Linking setupapi.lib in your project
3.1 Select Project properties.
3.2 Select 'Linker' node under the 'Configuration Properties'.
3.3 Select the 'Input' node under the 'Linker'.
3.4 Add lib 'setupapi.lib' in the 'Additional Dependencies' edit box. The setupapi.lib is a standard library and is available in the Microsoft SDK.

25 Features Not Supported

The Following features are not supported by CyAPI.lib

1. SET ADDRESS Feature

   The SET ADDRESS Request cannot be implemented through Control Endpoint.

2. SYNC FRAME

   The SYNC FRAME Request cannot be implemented through Control Endpoint.

3. USB 3.0 Bulk Streaming

   Data transfer on stream enabled BULK endpoints is not supported.

.

4. Set/Get Transfer size

The XferSize variable to get/set the transfer size of endpoint is no longer supported. Please refer XferSize for more information.

Index

- D -

- E -

- I -

- U -