Introduction

The USB Driver Toolkit provides the following components:

- **USB Control Panel** provides a variety of functions to assist in the development of the USB device. Please refer to the help provided with the control panel.

- **USB_Tray** is a tray application which indicates how many USB devices using the APA USB driver and a specific GUID are attached to the system. The GUID is selected using the GUID combo-box in the Device tab of the USB Control Panel; once the GUID is selected in the Control Panel the Tray uses this GUID to determine if devices are connected. The default GUID is the APA USB GUID specified in the example .inf files. Please refer to the Vendor and GUID section of this manual for more information on the importance of GUID's and their use with USB devices. A USB icon in the Tray without a red cross through it indicates that at least one device is present. Placing the mouse cursor over the icon displays a pop-up with the actual count.

- **APAUSB.sys** is WDM driver which supports the functions in the supplied DLL. It also correctly handles PnP and power management and has been tested on Win98 through WinXP.

- **USBDrvD.DLL** is a Win32 DLL which provides a C callable API for calling the WDM driver. It also performs various house keeping tasks so that they don't have to be repeated in every application that uses the driver. The majority of this document focuses on using the DLL to communicate with USB devices.

- **Bulkloop** is an application which allows different types of data to be continuously sent to and received from a device. It measures throughput, checks for errors, and logging of large numbers of transfers. Please refer to the help provide with Bulkloop for more information.

- **Firmware** examples for Cypress EZ-USB, FX, and FX2.

- **Sample** applications for Visual C++, Delphi, Visual Basic, and Borland C++ Builder. LabView VI's are available on request.

Driver installation

The driver is installed using a .inf file. The supplied apausb2k.inf and apausb98.inf file supports devices using the Vendor ID (VID) of 0xBD7 and Product ID (PID) of 0x1002. The .inf files are in the driver\Win98_ME and driver\Win2K_XP sub-directories of the install directory. These directories also contain the driver (apausb.sys). The Win2K_XP directory also contains a device property page dll (apausbprop.dll). The property page DLL is only for use on Win2k and WinXP. In addition, cyusb_98.inf and cyusb_2k.inf are supplied to allow the usb driver to install if several of the default Cypress VID/PID combinations are used with EZ-USB, FX, or FX2 development boards.

The apausb2k.inf file is for use on Windows 2000 and Windows XP. The apausb98.inf file is for use on Windows 98 through Windows ME.

When a device with this VID/PID is attached the "New Hardware Found" wizard will appear. Select "install device from specific location (advanced)" (for XP). Then browse to the appropriate directory. For Windows 98 and ME "c:\apa_usb\driver\win98_me" is used, and for Windows 2K and XP c:\apa_usb\driver\win2K_XP should be used.

The VID (0x0BD7) is property of Andrew Pargeter & Associates and <u>MUST NOT BE USED</u> in products not distributed by Andrew Pargeter & Associates or affiliates. However, the sample VID and PID can be used during development We will assume that you are using one of the Cypress development boards (EZ-USB, FX, or FX2). The boards use an I2C EEPROM to determine the VID and PID. Program the I2C chip as follows starting at address 0:

<u>EZ-USB</u>
Use 19.2k for connection to Keil debugger.
0xB0 0xD7 0x0B 0x02 0x10 0x01 0x00

<u>FX</u>

Default processor speed of 24mhz; use 19.2K for connection to Keil debugger
0XB4 0xD7 0x0B 0x02 0x10 0x01 0x00 **0x00**
High processor speed of 48mhz; use 38.4k for connection to Keil debugger.
0XB4 0xD7 0x0B 0x02 0x10 0x01 0x00 **0x04**

<u>FX2</u>

Default processor speed of 48mhz; use 38.4k for connection to Keil debugger.
0xC0 0xD7 0x0B 0x02 0x10 0x01 0x00 0x00

Vendor ID and GUID

The supplied *.inf files supports devices using the Vendor ID (VID) of 0xBD7 and Product ID (PID) of 0x1002. This VID is property of Andrew Pargeter & Associates (APA) and MUST NOT BE USED in products not distributed by Andrew Pargeter & Associates or affiliates. All firmware examples use this VID/PID combination.

The APA GUID of {506E28D2-9329-11d4-A398-0000C07754F8} is used by the sample applications and driver to provide a symbolic link for opening handles. This is the device class used during installation of the driver using the .inf file. This GUID is associated with the VID and PID used in the APA Sample USB device.

When using your own VID and PID you must define a GUID using the Microsoft supplied guidgen.exe and edit the .inf file replacing the APA GUID with yours and also use your GUID when opening handles to your device. The relevant sections of the .inf file are the ClassGUID in [Version] and APAClassGuid in [Strings]. *Note that the ClassGuid in the .inf file can't reference a defined string for some unknown reason, and if it does the .inf won't work properly.*

USB.ORG handles the assignment of VID's for a yearly fee. You can contact them at www.usb.org.
Alternatively, we can assign a PID under our VID for each product licensed for use with the APA USB driver. However, we do have a limit of 64,000 PID's that we can give out so this is a first come, first serve basis!

Programming Guidelines

The host application must use the Device DLL to access the device which the USB driver is managing. Most of the functions require a call to OpenDevice() before they are called with the exception of GetDevCount() and the Pipe* functions which require a call to PipeOpen() instead of OpenDevice(). A call to the appropriate DLL function must also be made to close any open handles before the application terminates. All of the functions are prefixed with USBDRVD_ to avoid conflicts with function names in your code.

Using HANDLE Return Values:

All of the open functions return a HANDLE type. These functions are OpenDevice(), OpenDevicePath(), PipeOpen(), and PipeOpenPath(). The HANDLE value returned by these functions can only be used in function call's to the USBDRV.DLL. *It cannot be used as a parameter in any of the Win32 function calls such as ReadFile(), WriteFile(), and CancelIO().* USBDRVD_PipeGetW32Handle() is provided to obtain the Win32 HANDLE for use with CancelIO(), GetOverlappedResult(), etc.

All of the functions which are not prefixed with "Pipe" require a HANDLE obtained using OpenDevice() or OpenDevicePath(). The Pipe functions such as PipeRead(), PipeReadTimeout() must be called using a HANDLE returned by PipeOpen() or PipeOpenPath().

Pipes and Endpoints:

Endpoints are physical buffers or memory on USB chips such as the EZ-USB. Each device has a fixed number of endpoints that are configured as either bulk, interrupt, or isochronous. Pipe numbers are associated with the order that endpoint descriptors appear in the interface descriptor, and NOT specifically with the endpoint numbers. When the USB driver loads and configures a device it associates each endpoint in the interface descriptor with a pipe number starting from number 0. The first endpoint defined is assigned pipe 0, the second pipe 1, and so on. *The pipe number used in the function calls may or may not be the* *same as the endpoint number that the pipe refers to*.

Building your application:

The install directory contains a subdirectory \DLL which contains the following:
USBDrvD.h - include file for functions and structures. Include this in your code.
USBDrvD.DLL - dll which must be present in your executable directory.
USBDrvD.lib - library file used when linking your application. Note that Borland C++ Builder requires a different .lib file which is located in the dll\Borland_C directory on the install path.

Note for Delphi and Visual Basic users: use the example app's. for Delphi and VB to get the DLL function definitions instead of using the USBDrvD.h include file. You will not be using the UsbDrvD.lib file either!

Dealing with surprise removal:

Surprise removal is a Windows term for what happens when you unplug the USB cable used with your device. It is very important that you detect when your device has been removed from the system and close any open handles that you have for the device. Failure to do so will hamper proper operation of the device when you attach it to the system once again. The most reliable method for determining if the number of your devices has changed is calling GetDevCount() periodically to check for device removal and attachment. Using this function along with GetDevicePath() (which allows you detect the USB device descriptor serial number) is an effective method for managing multiple devices.

The host application may also use the Windows API function **RegisterDeviceNotification()** to detect when a USB device is detached from, and attached to, the system. The VC++ MFC sample called **download** illustrates how to do this. Note that on Win98 calling UnRegisterDeviceNotification() causes system instability.

Note: *RegisterDeviceNotification() requires that dbt.h be included in your Visual C++ source file.* ***DEV_BROADCAST_DEVICEINTERFACE*** *is defined in this file. VC6.0 sp1 requires* ***#define WINVER 0x0500*** *be placed before this* *file is included. See the download example downloadDlg.cpp in the VC\Cypress directory for correct placement of #define WINVER 0x0500.*

Dealing with power management:

The host application must process the **WM_POWER** broadcast message and take appropriate action when going into hibernation mode, and coming out of hibernation mode. Generally, if USB devices loose power during hibernation, they will be enumerated upon power up and any open handles to the device will be orphaned. In this case any open handles to the USB device must be closed before going into hibernation. Also, depending on the ACPI support, even self powered devices will be enumerated once the system returns to normal power state.

Overview of DLL functions (API)

The following functions are exported and are linkable to a Visual C++ or any other environment that can import functions from a DLL (e.g. Visual Basic, Delphi). All DLL functions are exported using the _stdcall calling convention.

Basic functions:

- GetDriverVersion – Obtains the WDM driver version information
- GetDevCount – Obtains number of devices attached.
- GetDevicePath - Obtains the PnP device path for specified device.
- OpenDevice - Obtains a handle to the USB device for future functions calls.
- CloseDevice - Closes a handle obtained by a previous call to OpenDevice().
- OpenDevicePath - Obtains a handle to the USB device using the PnP device path for future function calls.

Port functions:

- ResetParentPort - Resets the parent port of the device
- GetParentPortStatus - Obtains the status of the parent port and the device.
- CyclePort - Disconnect and then reconnect the device.

Pipe functions:

- GetPipeCount – Obtains the pipe count on the specified device.
- GetPipeInfo - Obtains the pipe information for the specified pipe. The pipe information indicates direction, type (bulk, interrupt, ISO), transfer size, etc.
- AbortPipe – Aborts any pending I/O operations on the specified pipe
- ResetPipe - Resets a specified pipe.

Descriptor functions:

- GetDeviceDescriptor - Obtains the device descriptor.
- GetConfigDescriptor - Obtains the configuration descriptor.
- GetInterfaceDescriptor - Obtains the interface descriptor.
- GetEndpointDescriptor - Obtains the endpoint descriptor for the specified endpoint.

GetStringDescriptorLength - Returns the length of the specified string descriptor.

GetStringDescriptor - Obtains the specified string descriptor.

Read/Write functions:

InterruptRead - Read from a Interrupt type endpoint.

InterruptWrite - Write to an Interrupt type endpoint.

BulkRead - Read from a Bulk type endpoint.

BulkWrite - Write to a Bulk type endpoint.

Miscellaneous functions:

SelectInterface - Selects the specified interface and alternate setting.

GetLanguageIDs

GetNumberOfLanguageIDs

Vendor/Class Endpoint 0 functions:

VendorOrClassRequestOut - Issues commands to control endpoint 0 with a direction of OUT. Used to read from endpoint 0.

VendorOrClassRequestIn - Issues commands to control endpoint 0 with a direction of IN. Used to write to endpoint 0.

Cypress EZ-USB(tm) specific functions:

EZUSBDownloadRam - Download firmware to Cypress EZ-USB, FX, and FX2 RAM.

EZUSBDownloadI2C - Download firmware to Cypress EZ-USB, FX, and FX2 RAM I2C.

Pipe I/O Functions:

PipeOpen - Opens Pipe on specified device returning handle for pipe operations.

PipeClose - Given handle from prior USB_PipeOpen(), closes the pipe.

PipeWrite - WriteFile() wrapper function which supports overlapped I/O. Bulk or Interrupt is supported.

PipeRead - ReadFile() wrapper function which supports overlapped I/O. Bulk or Interrupt is supported.

PipeWriteTimeout - WriteFile() wrapper function with timeout. Bulk or Interrupt is supported.

PipeReadTimeout - ReadFile() wrapper function with timeout. Bulk or Interrupt is supported.

PipeAbort - Aborts any pending I/O operations on the pipe.

PipeReset - Resets the pipe.

Driver Configuration

There are several item in the in the registry that allow configuration of the driver.

The HKLM\System\CurrentControlSet\Services\APAUSB is the services key for the driver in the .inf file provided.

ClassGUID - class GUID used for creating symbolic names to access the driver. <u>This MUST be the same GUID that is used to open handles using the DLL functions.</u>

A subkey is used in the format Vid_nnnn&Pid_nnnn under the services key for device specific configuration. The "nnnn" portions are the hexidecimal, lower case, values for the VID and PID of the device you are using the driver with.

For example, the sample devices have a VID = 0x0bd7 and a PID = 0x1002 and the key where the device specific configuration information is as follows:

HKLM\System\CurrentControlSet\Services\APAUSBSYS\Vid_0bd7&Pid_1002

Here is a description of the configurable entries under this key:

AuthKey - the license information for licensed devices. This is supplied when you license a device.

RenumOnHibernate - if set to 1 causes the device to be hidden when resuming from hibernation. This is important if the device loses power during hibernation because it will appear to have been detached and re-attached to the system when the system comes out of hibernation. The system will unload the driver, enumerate the device again, and reload the driver. Hiding the device prevents the application from using it immediately after returning from hibernation when the driver unload/ and reload sequence is going to occur shortly (usually this happens within a few seconds).

MaxTransfer - set the maximum transfer size for bulk and interrupt transfers. The default is 4096 bytes. Setting this higher will increase throughput but use more system resources. Set this to the lowest value possible to improve sharing the USB connection with other devices and reduce system loading. Values in the 1-3 MB range optimize throughput on USB 2.0. Be careful to use multiples of 4096 (4k) when setting the maximum transfer size. Please read the topic about USB 2.0 Windows Service Pack Requirements.

Sample Code

Samples which provide a simple control panel application are provided for Visual C++, Visual Basic, Delphi, and Borland C++ Builder. Please see the samples directory and look for VB, Delphi and Borland_C.

The following Visual C++ 6.0 samples are provided in the samples/VC directory:

Note: All Visual C++ samples were built using VC++ version 6.0.

APA_USB - MFC app. that demonstrates a majority of the functions provided. This sample can be used as framework for build a test application for use with your USB device. It also shows how to process the Windows power broadcast and device notification messages.

blockread - console app. that reads from a bulk or interrupt endpoint. The sample shows how to use the following functions:

GetDevCount()	OpenDevice()	GetDriverVersion()
GetPipeInfo()	ResetPipe()	**BulkRead()**
CloseDevice()		

blockwrite - console app. that writes to a bulk or interrupt endpoint. The sample shows how to use the following functions:

GetDevCount()	OpenDevice()	GetDriverVersion()
GetPipeInfo()	ResetPipe()	**BulkWrite()**
CloseDevice()		

descriptor - console app. that reads and process the device, interface, and string descriptors. The sample shows how to use the following functions:

GetDeviceDescriptor()	GetConfigDescriptor()	GetInterfaceDescriptor()
GetEndpointDescriptor()	GetStringDescriptorLength()	GetStringDescriptor()

Sample Code

Samples which provide a simple control panel application are provided for Visual C++, Visual Basic, Delphi, and Borland C++ Builder. Please see the samples directory and look for VB, Delphi and Borland_C.

The following Visual C++ 6.0 samples are provided in the samples/VC directory:

Note: All Visual C++ samples were built using VC++ version 6.0.

APA_USB - MFC app. that demonstrates a majority of the functions provided. This sample can be used as framework for build a test application for use with your USB device. It also shows how to process the Windows power broadcast and device notification messages.

blockread - console app. that reads from a bulk or interrupt endpoint. The sample shows how to use the following functions:

GetDevCount()	OpenDevice()	GetDriverVersion()
GetPipeInfo()	ResetPipe()	**BulkRead()**
CloseDevice()		

blockwrite - console app. that writes to a bulk or interrupt endpoint. The sample shows how to use the following functions:

GetDevCount()	OpenDevice()	GetDriverVersion()
GetPipeInfo()	ResetPipe()	**BulkWrite()**
CloseDevice()		

descriptor - console app. that reads and process the device, interface, and string descriptors. The sample shows how to use the following functions:

GetDeviceDescriptor()	GetConfigDescriptor()	GetInterfaceDescriptor()
GetEndpointDescriptor()	GetStringDescriptorLength()	GetStringDescriptor()

GetDevCount()	OpenDevice()	CloseDevice()

piperead - console app. that reads from a bulk or interrupt endpoint. Uses the pipe functions and illustrates how to have a read timeout after a specified period of time. The sample shows how to use the following functions:

GetDevCount()	OpenDevice()	GetDriverVersion()
GetPipeInfo()	CloseDevice()	PipeOpen()
PipeReset()	**PipeRead()**	**PipeGetW32Handle()**
WaitForSingleObject()	GetOverlappedResult()	PipeClose()
(Win32 API)	(Win32 API)	

pipewrite - console app. that writes to a bulk or interrupt endpoint. Uses the pipe functions and illustrates how to have a write timeout after a specified period of time. The sample shows how to use the following functions:

GetDevCount()	OpenDevice()	GetDriverVersion()
GetPipeInfo()	CloseDevice()	PipeOpen()
PipeReset()	**PipeWrite()**	**PipeGetW32Handle()**
WaitForSingleObject()	GetOverlappedResult()	PipeClose()
(Win32 API)	(Win32 API)	

pipereadtimeout - console app. that reads from a bulk or interrupt endpoint. Uses the pipe functions and illustrates how to have a read timeout after a specified period of time. The sample shows how to use the following functions:

GetDevCount()	OpenDevice()	GetDriverVersion()
GetPipeInfo()	CloseDevice()	PipeOpen()
PipeReset()	**PipeReadTimeout()**	PipeClose()

pipewritetimeout - console app. that writes to a bulk or interrupt endpoint. Uses the pipe functions and illustrates how to have a write timeout after a specified period of time. The sample shows how to use the following functions:

GetDevCount()	OpenDevice()	GetDriverVersion()

GetPipeInfo()	CloseDevice()	PipeOpen()
PipeReset()	**PipeWriteTimeout()**	PipeClose()

download - MFC app that uses the Cypress specific functions to download firmware to RAM and I2C. Shows how to use the following functions:

GetDevCount()	OpenDevice()	VendorOrClassRequestOut()
EZUSBDownloadI2C()	EZUSBDownloadRAM()	CloseDevice()

This sample also shows how to use the Win32 API function RegisterDeviceNotification() to detect when a USB device is detached from, and attached to the system.

Distribution with your USB device

In order to create a driver disk for distribution with your USB device the following steps are taken:

- Create a GUID using guidgen.exe in the Microsoft platform SDK. There are several GUID utilities offered as freeware on the internet that can be used as well.

- Determine your VID/PID and obtain a license key from us.

- Use the sample .inf's supplied as templates to create your own. Files are supplied for Win98/ME and for Win2K/XP. Individual .inf files are supplied because they are easier to deal with and it reduces the chance of introducing problems when compared to a combined Win98 and Win2K .inf file. Understanding .inf files is hard enough without trying to create combined files. Future versions of the USB driver will have Windows version specific features so it is a good idea to use seperate .inf files from the start with your USB product.

- Provide your Win98/ME .inf and the apausb.sys (should be renamed) in a subdirectory for Win98/ME driver installation.

- Provide your Win2K/XP .inf, apausb.sys, and apausbprop.dll (both should be renamed) in a subdirectory for Win2K/XP driver installation.

- Provide the usbdrvd.dll (should be renamed) with your application.

Please note that only usbdrvd.dll, apausb.sys, and apausbprop.dll are allowed to be distributed with your application as stated by the license agreement.

Data Types (GUID and HANDLE)

Since it is not always easy to find the definition of the GUID data type in the Windows SDK, here is how it is defined:

```
typedef struct _GUID {          // size is 16
   DWORD Data1;
   WORD   Data2;
   WORD   Data3;
   BYTE  Data4[8];
} GUID;
```

All of the examples use this for declaring the GUID:

```
#include <initguid.h>
// {506E28D2-9329-11d4-A398-0000C07754F8}";
DEFINE_GUID(GUID_CLASS, 0x506e28d2, 0x9329, 0x11d4, 0xa3, 0x98, 0x00,
                                        0x00, 0xc0, 0x77, 0x54, 0xF8);
```

The **HANDLE** data type is defined in the SDK as:

```
typedef void *HANDLE;
```

Bulk Throughput FAQ

A common issue with USB bulk transfers is that developers see much lower transfer rates than they expect. There are several important things to consider as follows:

USB protocol - Taking USB 1.1 as an example (USB 2.0 is more complex, but the same concepts apply) bulk in/out transactions are scheduled on every 1ms time interval. If your transactions are not scheduled every 1ms for reason, but for some reason they are scheduled every 2ms you will see loss of throughput.

The maximum number of bulk data that can be transfered in/out per 1ms is 16 x 64 byte packets or 1024bytes. Let's assume that you always transfer 1024 bytes in the call to ReadFile/WriteFile. You will experience at least a 50% loss of througput because at least every other 1ms frame will have no data pay load because of the time it takes for the application to issue another request (usually takes somewhere in the order of 1.8ms).

In order to maximize throughput you must transfer large blocks of data in each ReadFile/WriteFile request.

USB Device and Firmware - The Cypress EZ-USB example called bulkloop (or ep_pair) uses a for-loop to copy each byte from the in buffer to the out buffer. This is an extremely slow operation and results in additional 1ms frames that have no data payload because the EZ-USB processor is to slow. The FX introduced DMA for this reason so that the endpoint buffers can be filled extremely quickly from either internal memory or FIFO's.

A quick note about USB 2.0 - if the FX2 8051 core is involved in the loading of the endpoint buffers then disappointing throughput will be seen. You must minimize the 8051's involvement in the endpoint processing as much as possible.

Host Application - the application must use advanced I/O methods in order to optimize throughput. It must make an effort to minimize processing the data between issuing the next read/write request. This is accomplished using overlapped I/O and thread techniques.

Deciding How to Read and Write Data

The DLL has several functions that allow you to read data from your device. There are also functions to write data to your device. All of the functions fall into either synchronous (blocking), and asynchronous (non-blocking) which is otherwise known as using the Windows overlapped I/O model.

The primary decision you need to make is whether you need synchronous or asynchronous access to your device for interrupt or bulk transfers. If you want to move a lot of data between your device and the host system then you should use the Pipe functions PipeRead(), PipeWrite(), PipeReadTimeout() and PipeWriteTimeout().

The blocking functions are simple to use but have obvious limitations; they were implemented to make migration from the Cypress EZ-USB generic driver easy to do. They are effective if your USB device is always ready to receive data, or is constantly sending data, and high data throughput rates are not required.

The asynchronous pipe functions are most commonly used because they allow high data rates and flexible management of R/W requests. <u>Even if you don't need high data rates it is still recommended that you use the pipe functions!</u>

PipeRead() and PipeWrite() are minimal wrappers around the Win32 functions ReadFile() and WriteFile() which do some sanity checking. Used in conjunction with PipeGetW32Handle(), the full flexibility of Windows overlapped I/O model is available. PipeRead() and PipeWrite() are very efficient ways to move a lot of data. Please see the examples for PipeRead() and PipeWrite() in the VC++ sample applications PipeRead and PipeWrite.

The PipeReadTimeout() and PipeWriteTimeout() are blocking functions which use overlapped I/O in order for them to timeout after a specified interval. They use the

Win32 functions GetOverlappedResult() and CancelIO() to manage the I/O request. They are as fast as PipeRead() and PipeWrite() and are ideal for use with development environments such as LabView where Win32 function calls are difficult to manage. They are also effective when using VC++, Delphi, Borland, etc. The timeout values should be several seconds to insure that an I/O request is not cancelled at the same time the USB device is completing it. The VC++ examples for PipeReadTimeout() and PipeWriteTimeout() are PipeReadTimeout and PipeReadTimeout. The functions are also used in the Borland C and Delphi examples.

The BulkRead(), InterruptRead(), BulkWrite(), and InterruptWrite() functions are very simple blocking i/o calls which use the handle returned by the OpenDevice() and OpenDevicePath() functions. Since they are blocking calls they will only return under the following conditions:

1 - the requested amount of data has been received.

2- the device is unplugged from the system.

3- an AbortPipe() is issued for the specified pipe.

4- a short packet or zero-length-packet (ZLP) is received from the device.

USB 2.0 Windows Service Pack Requirements

Windows 2000 and XP had a problem which prevents USB Zero Length Packets (ZLP's) or short packets (< 512 or 64 bytes depending on the endpoint configuration Max. Packet Size) from completing an I/O request where the requested size is more than 16 kBytes. This problem was fixed with Windows 2000 service pack 4 (Win2k SP4) and the Windows XP service pack 1 (WinXP SP1).

The issue arises when an application is requesting large blocks of data (> 16 kBytes) and the design of the USB dictates that either a ZLP or short packet is used to tell the host that the device is finished sending data. An example of this type of application would be a scanning device that takes variable length documents where the USB device detects the end of the document feed. This leads to a variable amount of data being sent to the host on each scan. Typically a single bulk read request will be pending when the scan ends and the USB device tells the host to complete the request and return the actual number of bytes transferred by sending a short packet or a ZLP to the host.

Note: The problem described in this topic has nothing to do with the USB driver. It is a bug in the Windows USB 2.0 host controller driver.

Win2k SP4 and WinXP SP1 are required under the following circumstances for the above mechanism to work properly:

- A USB 2.0 device has enumerated at high speed.
- Bulk or Interrupt transfer requests are greater than 16 kBytes.
- A ZLP or short packet is used to complete a read request.

USBDRVD_GetDevCount()

```
UINT USBDRVD_GetDevCount(IN GUID * DeviceClassGUID);
```

Returns the number of USB devices for the class GUID attached to the system or 0 if no devices are attached.

Example:

```
UINT DeviceCount = USBDRVD_GetDevCount(&GUID);
```

USBDRVD_GetDevicePath()

```
USBDRVD_API UINT USBDRVD_GetDevicePath(IN UINT DevNum, IN GUID *
        USB_GUID, OUT char * Dest, IN UINT Length);
```

This is used to get the PnP device path (also known as symbolic link). The return value is the length of the resulting string, or 0 on error.

Example:

```
UINT DevNum = 1; //we want the first device
char Path[MAX_PATH];

UINT Length = USBDRVD_GetDevicePath(DevNum, &GUID_CLASS, Path,
                                                    sizeof(Path)));

if( Length )
{
    ...do something with the Path
}
```

USBDRVD_OpenDevice()

```
HANDLE USBDRVD_OpenDevice(IN UINT DevNum, IN DWORD dwReserved,
        IN GUID * DeviceClassGUID);
```

Obtains a handle to the USB device for future functions calls. A return of INVALID_HANDLE_VALUE indicates an error.

Note: dwReserved must be specified as 0.

Example:

```
UINT DevNum = 1; //we want the first device

HANDLE hDevice = USBDRVD_OpenDevice(DevNum, 0, &GUID);

if( hDevice != INVALID_HANDLE_VALUE )
{
    ...do some work here...

    USBDRVD_CloseDevice(hDevice);
}
```

USBDRVD_OpenDevicePath()

```
USBDRVD_API HANDLE USBDRVD_OpenDevicePath(char *PnP_Path,
        DWORD dwReserved);
```

Obtains a handle to the USB device for future functions calls. A return of INVALID_HANDLE_VALUE indicates an error.

Note: dwReserved must be specified as 0.

Example:

```
UINT DevNum = 1; //we want the first device
char Path[MAX_PATH];

UINT Length = USBDRVD_GetDevicePath(DevNum, &GUID_CLASS, Path,
                                                  sizeof(Path)))


if( !Length )
{
    ...process error
}

HANDLE hDevice = USBDRVD_OpenDevicePath(Path, 0);

if( hDevice != INVALID_HANDLE_VALUE )
{
    ...do some work here...

    USBDRVD_CloseDevice(hDevice);
}
```

USBDRVD_CloseDevice()

```
void USBDRVD_CloseDevice(IN HANDLE hDevice);
```

Closes a handle obtain by a prior call to USBDRVD_OpenDevice().

See also: OpenDevice()

USBDRVD_ResetParentPort()

BOOL USBDRVD_ResetParentPort(IN HANDLE hDevice);

Executes a USB reset on the parent port of the device. Returns TRUE if successful, FALSE otherwise.

Example:

```
UINT DevNum = 0;
DWORD Flags = 0;

HANDLE hDevice = USBDRVD_OpenDevice(DevNum, Flags, &GUID);

if( hDevice != INVALID_HANDLE_VALUE )
{
    USBDRVD_ReseParentPort( hDevice );
    USBDRVD_CloseDevice( hDevice );
}
```

USBDRVD_GetParentPortStatus()

```
BOOL USBDRVD_GetParentPortStatus(IN HANDLE hDevice, OUT ULONG*
        pStatus)
```

Obtains the status of the parent port of the device. Returns TRUE if successful, FALSE otherwise.

Example:

```
ULONG Status;

if(USBDRVD_GetParentPortStatus(hDevice, &Status)
{
    if(Status != 0x03)
        USBDRVD_ResetParentPort(hDevice);
}
```

See also : ResetParentPort():

USBDRVD_CyclePort()

```
USBDRVD_API BOOL USBDRVD_CyclePort(HANDLE hDevice);
```

Use this function to cause the host controller to disconnect and enumerate a device again. TRUE is returned if successful, otherwise FALSE is returned.

Note: this function is not normally used by applications and is supplied for diagnostic/test purposes. We have observed different behavior depending on the Windows version used. Use with caution.

Example:

```
if( USBDRVD_CyclePort(hDevice) )
{
    // It's important to close this handle, and any other open handles,
    // now so the device can enumerate again!!
    USBDRVD_CloseDevice(hDevice);
}
```

USBDRVD_GetPipeCount()

```
UINT USBDRVD_GetPipeCount(IN HANDLE hDevice);
```

Obtains the pipe count on the specified device. The return value is the number of pipes found, or 0 if no pipes exist.

Example:

```
UINT PipeCount;

PipeCount = USBDRVD_GetPipeCount( hDevice );

if( PipeCount )
{
    ...do something with the PipeCount.
}
```

See also: GetPipeInfo()

USBDRVD_GetPipeInfo()

UINT USBDRVD_GetPipeInfo(IN HANDLE hDevice, IN ULONG PipeNum, OUT
    USBDRVD_PIPE_INFO pInfo);

Obtains the pipe information for the specified pipe. The return value is sizeof(USBDRVD_PIPE_INFO) if successful, 0 otherwise.

Example:

```
USBDRV_PIPE_INFO  PipeInfo;
UINT PipeCount;
UINT PipeNum = 0;
UINT RetSize;

PipeCount = USBDRVD_GetPipeCount( hDevice );

for( PipeNum = 0; PipeNum < PipeCount; PipeNum++)
{
    RetSize = USBDRVD_GetPipeInfo(hDevice, PipeNum, &PipeInfo));

    if( RetSize == sizeof(USBDRVD_PIPE_INFO) )
    {
        ...do something with PipeInfo
    }
}
```

USBDRVD_AbortPipe()

```
BOOL USBDRVD_AbortPipe(IN HANDLE hDevice, IN ULONG Pipe);
```

Aborts any pending I/O operations on the specified pipe. The return value is TRUE if successful, FALSE otherwise.
Use this function to terminate a pending read or write request issued using the blocking InterruptRead(), InterruptWrite(), BulkRead(), or BulkWrite() functions.

Note: *Use of this function on a pipe causes a pending transaction to be aborted. If no transaction is pending this function will cause the next transaction to fail and data sent from the device to satisfy that transaction is discarded.*

Example:

```
UINT Pipe = 0;  //we want the first pipe on the device

if( !USBDRVD_AbortPipe(hDevice, Pipe) )
{
    ...process error
}
```

USBDRVD_ResetPipe()

```
BOOL USBDRVD_ResetPipe(IN HANDLE hDevice, IN ULONG Pipe);
```

Resets a specified pipe. Generally used to clear a stall condition on a pipe. The return value is TRUE if successful, FALSE otherwise.

Example:

```
UINT Pipe = 0;  //we want the first pipe on the device

if( !USBDRVD_ResetPipe(hDevice, Pipe) )
{
    ...process error
}
```

USBDRVD_GetDeviceDescriptor()

```
UINT USBDRVD_GetDeviceDescriptor(IN HANDLE hDevice, IN OUT
        USBDRV_DEVICE_DESCRIPTOR  Device);
```

Obtains the device descriptor. The return value is sizeof(USBDRV_DEVICE_DESCRIPTOR) if successful, 0 otherwise.

Example:

```
USBDRV_DEVICE_DESCRIPTOR DeviceDesc;
UINT RetSize;

RetSize = USBDRVD_GetDeviceDescriptor( hDevice, &DeviceDesc );

if(RetSize == sizeof( USBDRV_DEVICE_DESCRIPTOR ))
{
   ...do something with DeviceDesc
}
```

USBDRVD_GetConfigDescriptor()

```
UINT USBDRVD_GetConfigDescriptor(IN HANDLE hDevice, IN OUT
        USBDRV_CONFIGURATION_DESCRIPTOR  ConfigDesc);
```

Obtains the configuration descriptor. The return value is sizeof(USBDRV_CONFIGURATION_DESCRIPTOR) if successful, 0 otherwise.

Example:

```
USBDRV_CONFIG_DESCRIPTOR ConfigDesc;
UINT RetSize;

RetSize = USBDRVD_GetDeviceDescriptor( hDevice, &ConfigDesc );

if(RetSize == sizeof( USBDRV_CONFIG_DESCRIPTOR ))
{
    ...do something with ConfigDesc
}
```

USBDRVD_GetInterfaceDescriptor()

```
UINT USBDRVD_GetInterfaceDescriptor(IN HANDLE hDevice, IN OUT
        USBDRV_INTERFACE_DESCRIPTOR  InterfaceDesc);
```

Obtains the interface descriptor. The return value is sizeof(USBDRV_INTERFACE_DESCRIPTOR) if successful, 0 otherwise.

Example:

```
USBDRV_INTERFACE_DESCRIPTOR InterfaceDesc;
UINT RetSize;

RetSize = USBDRVD_GetInterfaceDescriptor( hDevice, &InterfaceDesc );

if(RetSize == sizeof( USBDRV_INTERFACE_DESCRIPTOR ))
{
    ...do something with InterfaceDesc
}
```

USBDRVD_GetEndpointDescriptor()

```
UINT USBDRVD_GetEndpointDescriptor(IN HANDLE hDevice, IN UCHAR
        Endpoint, IN OUT USBDRV_ENDPOINT_DESCRIPTOR EndpointDesc);
```

Obtains the endpoint descriptor for the specified endpoint. The return value is sizeof(USBDRV_ENDPOINT_DESCRIPTOR) if successful, 0 otherwise.

Example:

```
USBDRV_INTERFACE_DESCRIPTOR InterfaceDesc;
USBDRV_ENDPOINT_DESCRIPTOR EndPointDesc;
UINT RetSize;

RetSize = USBDRVD_GetInterfaceDescriptor( hDevice, &InterfaceDesc );

if(RetSize == sizeof( USBDRV_INTERFACE_DESCRIPTOR ))
{
UINT Endpoint;

    for( Endpoint = 0; EndPoint < InterfaceDesc.bNumEndpoints; Endpoint ++)
    {
        RetSize = USBDRVD_GetEndpointDescriptor( hDevice, &EndPointDesc );

        if( RetSize == sizeof(USBDRVD_ENDPOINT_DESCRIPTOR))
        {
        ...do something with EndpointDesc
        }
    }
}
```

USBDRVD_GetStringDescriptorLength()

```
UINT USBDRVD_GetStringDescriptorLength(IN HANDLE hDevice, IN USHORT
        LangID, IN ULONG Index);
```

Returns the length of the string descriptor specified by Index. LangID is dependent on the firmware support for multiple languages. 0 is returned if no descriptor is found.

See also: GetStringDescriptor()

USBDRVD_GetStringDescriptor()

```
UINT USBDRVD_GetStringDescriptor(IN HANDLE hDevice, IN USHORT LangID,
        IN ULONG Index, IN ULONG Length, IN OUT PCHAR pString);
```

Obtains the string descriptor specified by Index. Returns the length of the string copied into buffer pointed to by pString which must be pre-allocated, or 0 if an error was encountered.

Example:

```
int i;
char *p;
int Length;

//get up to 16 string descriptors
for(i = 0; i < 16; i++)
{
        Length = USBDRVD_GetStringDescriptorLength(hDevice, 27, i);
        if(!Length)
                break;

        char *p = (char *)malloc(Length);
        if(!p)
        {
                printf("Insufficient Memory!");
                break;
        }

        Length = USBDRVD_GetStringDescriptor(hDevice, 27, i, Length, p);

        if(Length)
        {
                printf("=== Index %d ===\n", i);
                printf(p);
                printf("\n");
        }

        free(p);
}
```

USBDRVD_GetDriverVersion()

```
UINT USBDRVD_GetDriverVersion(IN HANDLE hDevice,IN OUT ULONG
        *pMajor,IN OUT ULONG *pMinor);
```

Copies the driver version into pMajor (major version) and pMinor (minor version). Returns TRUE on success, and FALSE otherwise.

Example:

```
ULONG Major, Minor;
if(USBDRVD_GetDriverVersion(hDevice, &Major, &Minor))
{
    printf("Driver version is %d.%d\n", Major, Minor);
}
```

USBDRVD_SelectInterface()

UINT USBDRVD_SelectInterface(IN HANDLE hDevice, IN BYTE Interface, IN
    BYTE AlternateSetting);

Selects the specified interface and alternate setting. Returns TRUE on success, and FALSE otherwise.

USBDRVD_InterruptRead()

ULONG USBDRVD_InterruptRead(HANDLE hDevice, ULONG Pipe, BYTE
    *pBuff, ULONG Count);

Use this function to read from an interrupt endpoint. Reads **Count** data from the specified **Pipe** into the buffer pointed to by **pBuff**. Returns the count of bytes read, or 0 on error. Call GetLastError() to get error codes.

Note: This is a blocking call. *The function will return under the following conditions:*

1 - the requested amount of data has been received.

2- the device is unplugged from the system.

3- an AbortPipe() is issued for the specified pipe.

4- a short packet or zero-length-packet (ZLP) is received from the device.

Example:

```
DWORD nBytes = USBDRVD_InterruptRead(hDevice, PipeNum, Buf, Count);

if(nBytes)
  printf("Read %d bytes from pipe %d\n", nBytes, PipeNum);
else
  printf("Error is %d\n", GetLastError());
```

USBDRVD_InterruptWrite()

```
ULONG USBDRVD_InterruptWrite(HANDLE hDevice, ULONG Pipe, BYTE
        *pBuff, ULONG Count);
```

Use this function to write to an interrupt endpoint. Writes **Count** data from the specified **Pipe** into the buffer pointed to by **pBuff**. Returns the count of bytes read, or 0 on error. Call GetLastError() to get error codes.

Note: This is a blocking call. The function will return under the following conditions:

1 - the requested amount of data has been sent.

2- the device is unplugged from the system.

3- an AbortPipe() is issued for the specified pipe.

Example:

```
BYTE Buff[64];
ULONG Count = 64;
ULONG PipeNum = 0;
DWORD nBytes = USBDRVD_InterruptWrite(hDevice, PipeNum, Buf, Count);

if(nBytes)
  printf("Wrote %d bytes to pipe %d\n", nBytes, PipeNum);
else
  printf("Error is %d\n", GetLastError());
```

USBDRVD_GetLanguageIDs()

```
ULONG USBDRVD_GetLanguageIDs(HANDLE hDevice, ULONG Count,
        USHORT *pIDS);
```

NOT YET IMPLEMENTED

USBDRVD_GetNumberOfLanguageIDs()

```
ULONG USBDRVD_GetNumberOfLanguageIDs(HANDLE hDevice, USHORT
        *pIDS);
```

NOT YET IMPLEMENTED

ULONG USBDRVD_VendorOrClassRequestOut()

```
ULONG USBDRVD_VendorOrClassRequestOut(
        HANDLE hDevice,
        UCHAR Type, // 1=class, 2 = vendor
        UCHAR Destination,    // 0=device, 1=interface 2=endpoint, 3=other
        // see the USB Specification for an explanation of the following
        UCHAR Request,// request
        USHORT Value,  // value
        USHORT Index,   // index
        PUCHAR pData, // data buffer
        ULONG Length); // length of data
```

Issues an OUT request to control endpoint 0. These are called vendor or class requests. Please refer to the USB specifications for more information on how to implement firmware and use this function with your device. The Cypress/Download example and the EZ-USB firmware examples illustrate some of the use of this functions along with control endpoint 0 firmware support.

Example:

The following code puts the EZ-USB 8051 processor in the hold state.

```
BYTE Setting = 0x01;
if(USBDRVD_VendorOrClassRequestOut(hDevice, 2, 0, 0xA0, 0x7F92, 0,
&Setting, 1) != 1)
{
   printf("Cannot set device state to Hold!\n");
}
```

ULONG USBDRVD_VendorOrClassRequestIn()

```
ULONG USBDRVD_VendorOrClassRequestIn(
        HANDLE hDevice,
        UCHAR Type,  // 1=class, 2 = vendor
        UCHAR Destination,    // 0=device, 1=interface 2=endpoint, 3=other
        // see the USB Specification for an explanation of the following
        UCHAR Request,// request
        USHORT Value,   // value
        USHORT Index,   // index
        PUCHAR pData, // data buffer
        ULONG Length);  // length of data
```

Issues an IN request to control endpoint 0. These are called vendor or class requests. Please refer to the USB specifications for more information on how to implement firmware and use this function with your device. The Cypress/Download example and the EZ-USB firmware examples illustrate some of the use of this functions along with control endpoint 0 firmware support.

See also: VendorOrClassRequestOut()

USBDRVD_BulkRead()

```
ULONG USBDRVD_BulkRead(HANDLE hDevice, ULONG Pipe, BYTE *pBuff,
        ULONG Count);
```

Use this function to read from an bulk endpoint. Reads **Count** data from the specified **Pipe** into the buffer pointed to by **pBuff**. Returns the count of bytes read, or 0 on error. Call GetLastError() to get error codes.

Note: This is a blocking call. The function will return under the following conditions:

1 - the requested amount of data has been received.

2- the device is unplugged from the system.

3- an AbortPipe() is issued for the specified pipe.

4- a short packet or zero-length-packet (ZLP) is received from the device.

Example:

```
DWORD nBytes = USBDRVD_BulkRead(hDevice, PipeNum, Buf, Count);

if(nBytes)
  printf("Read %d bytes from pipe %d\n", nBytes, PipeNum);
else
  printf("Error is %d\n", GetLastError());
```

USBDRVD_BulkWrite()

```
ULONG USBDRVD_BulkWrite(HANDLE hDevice, ULONG Pipe, BYTE *pBuff,
        ULONG Count);
```

Use this function to write to an bulk endpoint. Writes **Count** data to the specified **Pipe** from the buffer pointed to by **pBuff**. Returns the count of bytes written, or 0 on error. Call GetLastError() to get error codes.

Note: This is a blocking call. The function will return under the following conditions:

1 - the requested amount of data has been sent.

2- the device is unplugged from the system.

3- an AbortPipe() is issued for the specified pipe.

Example:

```
BYTE Buff[64];
ULONG Count = 64;
ULONG PipeNum = 0;
DWORD nBytes = USBDRVD_BulkWrite(hDevice, PipeNum, Buf, Count);

if(nBytes)
  printf("Wrote %d bytes to pipe %d\n", nBytes, PipeNum);
else
  printf("Error is %d\n", GetLastError());
```

USBDRVD_EZUSBDownloadRam()

```
USBDRVD_API DWORD USBDRVD_EZUSBDownloadRam(
        const HINSTANCE hinstApp,
        const HWND hwndParent,
        const HANDLE hDevice,
        enum CY_DEVICE_TYPE Type,
        const char * pPath,
        BOOL bRun)
```

Downloads firmware to Cypress EZ-USB, FX, and FX2 RAM. pPath points to and ASCIIZ file path for the Intel HEX file to be downloaded. Type is either EZUSB or FX2.
hinstApp and hwndParent can be NULL. If hinstApp is the HINSTANCE of the application and hwndParent is a valid window handle (HWND) a progress bar is displayed at the bottom of the window referenced by hwdnParent. If bRun is TRUE the firmware is executed after down loading by a vendor command to endpoint 0 to take the 8051 out of reset.

This function returns 0 if successful. A non-zero return value is compatible with error codes returned by GetLastError().

See also: CY_DEVICE_TYPE, Visual C++ Cypress\Download example

USBDRVD_EZUSBDownloadI2C()

```
USBDRVD_API int USBDRVD_EZUSBDownloadI2C(const HINSTANCE
        hinstApp,
        const HWND hwndParent,
        const HANDLE hDevice,
        const char * pPath)
```

Downloads firmware to I2C if the current firmware supports the control endpoint 0 vendor requests. *This function is not specific to Cypress EZUSB parts and can be used on any device firmware that supports the endpoint 0 request used by this function*. pPath points to and ASCIIZ file path for the Intel HEX file to be downloaded.
hinstApp and hwndParent can be NULL. If hinstApp is the HINSTANCE of the application and hwndParent is a valid window handle (HWND) a progress bar is displayed at the bottom of the window referenced by hwdnParent.

This function returns 0 if successful. A non-zero return value is compatible with error codes returned by GetLastError().

See also: EZ-USB bulkloop example for endpoint 0 support, Visual C++ Cypress\Download example

USBDRVD_PipeOpen()

HANDLE USBDRVD_PipeOpen(IN UINT DevNum, IN UINT Pipe, IN DWORD
dwFlagsAndAttributes, GUID * USB_GUID);

Opens Pipe on device DevNum returning handle for pipe operations.
INVALID_HANDLE_VALUE is returned if an error occurred.

Note: dwFlagsAndAttributes must be specified with FILE_FLAG_OVERLAPPED if overlapped I/O operations are to be performed -or- if the PipeWriteTimeout() and PipeReadTimeout() functions are to be used with the HANDLE

USBDRVD_PipeOpenPath()

HANDLE USBDRVD_PipeOpenPath(char *PnP_Path, IN UINT Pipe, IN DWORD dwFlagsAndAttributes);

Opens Pipe on device specified by the PnP_Path returning handle for pipe operations. INVALID_HANDLE_VALUE is returned if an error occurred. The PnP_Path is obtained using GetDevicePath().

Note: ***dwFlagsAndAttributes*** *must be specified with* ***FILE_FLAG_OVERLAPPED*** *if the PipeWriteTimeout() and PipeReadTimeout() functions are to be used with the HANDLE, or if overlapped i/o is used with the PipeWrite() and PipeRead() functions.*

USBDRVD_PipeClose()

void USBDRVD_PipeClose(IN HANDLE hPipe);

Given hPipe from prior USBDRVD_PipeOpen() or USBDRVD_PipeOpenPath(), closes the pipe.

USBDRVD_PipeGetW32Handle()

HANDLE USBDRVD_PipeGetW32Handle(IN HANDLE hPipe);

Given hPipe from prior USBDRVD_PipeOpen() or USBDRVD_PipeOpenPath(), this function returns the Win32 HANDLE that can be used in Win32 file I/O function calls such as GetOverlappedResult() and CancelIO(), or it returns INVALID_HANDLE_VALUE if hPipe is not valid.

You can call ReadFile() or WriteFile() directly using the HANDLE returned by this function. If you want to call ReadFileEx() or WriteFileEx() to transfer data on a pipe you need to use PipeGetWin32Handle().

DO NOT call the Win32 API function CloseHandle() passing the HANDLE from PipeGetWin32Handle()!! *Use PipeClose() instead.*

The VC++ sample applications PipeRead and PipeWrite show the correct use of this function.

An explanation of why this function is included is that the return value from the PipeOpen functions return a HANDLE value which is actually an address to a structure. Within the structure is a member that contains the actual Win32 HANDLE. The actual Win32 HANDLE is needed when doing overlapped i/o using the PipeRead() and PipeWrite() functions in order to call GetOverlappedResult() and CancelIO(). The implementation of PipeGetW32Handle() is as follows:

```
typedef struct _PipeHandle_ {
  ULONG PipeNum;
  HANDLE hPipe;
  DWORD dwFlagsAndAttributes;
  char Tag[3];
} PIPEHANDLE, * PPIPEHANDLE;

HANDLE USBDRVD_PipeGetW32Handle(HANDLE hPipe)
{
  PPIPEHANDLE pPipeInfo = (PPIPEHANDLE)hPipe;
  if(!IsPipeHandleValid(pPipeInfo))
    return INVALID_HANDLE_VALUE;
  return pPipeInfo->hPipe;
}
```

USBDRVD_PipeWrite()

```
BOOL USBDRVD_PipeWrite(IN HANDLE hPipe,
        IN OUT LPVOID lpBuffer,                    // data buffer
        OUT DWORD nNumberOfBytesToWrite,           // # of bytes to write
        IN OUT LPDWORD lpNumberOfBytesWritten,     // number of bytes written
        OUT LPOVERLAPPED lpOverlapped              // pointer overlapped
                                                   structure
        );
```

WriteFile() wrapper function. The parameters and return code are identical to the Win32 functions WriteFile(), except the hPipe must be obtained using a previous call to USBDRVD_PipeOpen() or USBDRVD_PipeOpenPath().

USBDRVD_PipeRead()

```
BOOL USBDRVD_PipeRead(IN HANDLE hPipe,
        IN OUT LPVOID lpBuffer,                    // data buffer
        OUT DWORD nNumberOfBytesToRead,            // # of bytes to read
        IN OUT LPDWORD lpNumberOfBytesRead,        // number of bytes read
        OUT LPOVERLAPPED lpOverlapped              // pointer to overlapped
                                                   structure
        );
```

ReadFile() wrapper function. The parameters and return code are identical to WriteFile(), except the hPipe must be obtained using a previous call to USBDRVD_PipeOpen() or USBDRVD_PipeOpenPath().

USBDRVD_PipeWriteTimeout()

```
BOOL USBDRVD_PipeWriteTimeout(IN HANDLE hPipe,
        IN OUT LPVOID lpBuffer,                    // data buffer
        OUT DWORD nNumberOfBytesToWrite,           // # of bytes to write
        IN OUT LPDWORD lpNumberOfBytesWritten,     // number of bytes written
        IN DWORD dwTimeout                         //timeout in ms.
        );
```

WriteFile() wrapper function. The parameters and return code are identical to WriteFile() except the hPipe must be obtained using a previous call to PipeOpen() and a timeout value in milliseconds is specified instead of lpOverlapped.

A **timeout** is indicated if this function returns FALSE and GetLastError() returns 995 (ERROR_OPERATION_ABORTED).

Note: *In order for this function to work a prior call to USBDRVD_PipeOpen() or USBDRVD_PipeOpenPath() must be made with* ***dwFlagsAndAttributes = FILE_FLAG_OVERLAPPED*** *since the function uses overlapped I/O for the timeout processing.*

The following is from the Windows sdk and needs to be defined if the DLL is being used with Visual Basic, etc.

```
#define FILE_FLAG_OVERLAPPED          0x40000000
```

USBDRVD_PipeReadTimeout()

```
BOOL USBDRVD_PipeReadTimeout(IN HANDLE hPipe,
        IN OUT LPVOID lpBuffer,                        // data buffer
        OUT DWORD nNumberOfBytesToRead,                // # of bytes to read
        IN OUT LPDWORD lpNumberOfBytesRead,            // number of bytes read
        OUT DWORD dwTimeout                            // timeout in ms.
        );
```

ReadFile() wrapper function. The parameters and return code are identical to WriteFile() except the hPipe must be obtained using a previous call to PipeOpen() and a timeout value in milliseconds is specified instead of lpOverlapped.

A **timeout** is indicated if this function returns FALSE and GetLastError() returns 995 (ERROR_OPERATION_ABORTED).

Note: *In order for this function to work a prior call to USBDRVD_PipeOpen() or USBDRVD_PipeOpenPath() must be made with* ***dwFlagsAndAttributes = FILE_FLAG_OVERLAPPED*** *since the function uses overlapped I/O for the timeout processing.*

The following is from the Windows sdk and needs to be defined if the DLL is being used with Visual Basic, etc.

```
#define FILE_FLAG_OVERLAPPED          0x40000000
```

USBDRVD_PipeAbort()

BOOL USBDRVD_PipeAbort(IN HANDLE hPipe);

Aborts any pending I/O operations on the specified pipe. The return value is TRUE if successful, FALSE otherwise.
This function is not normally required in a well designed application. It can be used to terminate I/O requests when the application is shutting down so R/W threads can exit quickly.

Note: *Use of this function on a pipe causes a pending transaction to be aborted. If no transaction is pending this function will cause the next transaction to fail, and data sent to or from the device to satisfy that transaction is discarded. DON'T use AbortPipe() after a call to PipeRead(), PipeWrite(), PipeReadTimeout(), or PipeWriteTimeout(). Doing so will cause the next R/W request on the pipe to fail because you are telling the USB stack to abort the current or the next I/O request.*

USBDRVD_PipeReset()

```
BOOL USBDRVD_ResetPipe(IN HANDLE hPipe);
```

Resets a specified pipe. Generally used to terminate a stall condition. The return value is TRUE if successful, FALSE otherwise.

USBDRV_PIPE_TYPE

```
typedef enum _USBDRV_PIPE_TYPE {
        PipeTypeControl,
        PipeTypeIsochronous,
        PipeTypeBulk,
        PipeTypeInterrupt
        } USBDRV_PIPE_TYPE;
```

CY_DEVICE_TYPE

```
enum CY_DEVICE_TYPE {
        EZUSB,
        FX2,
        CY_INVALID
        };
```

USBDRV_PIPE_INFO

```
typedef struct _USBDRV_PIPE_INFO {
        USHORT MaximumPacketSize;// Maximum packet size for this pipe
        UCHAR EndpointAddress;        // 8 bit USB endpoint address/direction
        //The following comes from the USB endpoint descriptor
        UCHAR Interval;               // Polling interval in ms if interrupt pipe
        USBDRV_PIPE_TYPE PipeType; // PipeType identifies type of transfer
        valid for this pipe
        ULONG MaximumTransferSize;     // Maximum size for a single request in
        bytes.
        } USBDRV_PIPE_INFO, *PUSBDRV_PIPE_INFO;
```

USBDRV_DEVICE_DESCRIPTOR

```
typedef struct _USBDRV_DEVICE_DESCRIPTOR {
        UCHAR bLength;
        UCHAR bDescriptorType;
        USHORT bcdUSB;
        UCHAR bDeviceClass;
        UCHAR bDeviceSubClass;
        UCHAR bDeviceProtocol;
        UCHAR bMaxPacketSize0;
        USHORT idVendor;
        USHORT idProduct;
        USHORT bcdDevice;
        UCHAR iManufacturer;
        UCHAR iProduct;
        UCHAR iSerialNumber;
        UCHAR bNumConfigurations;
        } USBDRV_DEVICE_DESCRIPTOR,
        *PUSBDRV_DEVICE_DESCRIPTOR;
```

USBDRV_CONFIGURATION_DESCRIPTOR

```
typedef struct _USBDRV_CONFIGURATION_DESCRIPTOR {
        UCHAR bLength;
        UCHAR bDescriptorType;
        USHORT wTotalLength;
        UCHAR bNumInterfaces;
        UCHAR bConfigurationValue;
        UCHAR iConfiguration;
        UCHAR bmAttributes;
        UCHAR MaxPower;
        } USBDRV_CONFIGURATION_DESCRIPTOR,
        *PUSBDRV_CONFIGURATION_DESCRIPTOR;
```

USBDRV_INTERFACE_DESCRIPTOR

```
typedef struct _USBDRV_INTERFACE_DESCRIPTOR {
        UCHAR bLength;
        UCHAR bDescriptorType;
        UCHAR bInterfaceNumber;
        UCHAR bAlternateSetting;
        UCHAR bNumEndpoints;
        UCHAR bInterfaceClass;
        UCHAR bInterfaceSubClass;
        UCHAR bInterfaceProtocol;
        UCHAR iInterface;
        }USBDRV_INTERFACE_DESCRIPTOR,
        *PUSBDRV_INTERFACE_DESCRIPTOR;
```

USBDRV_ENDPOINT_DESCRIPTOR

```
typedef struct _USBDRV_ENDPOINT_DESCRIPTOR {
        UCHAR bLength;
        UCHAR bDescriptorType;
        UCHAR bEndpointAddress;
        UCHAR bmAttributes;
        USHORT wMaxPacketSize;
        UCHAR bInterval;
        }USBDRV_ENDPOINT_DESCRIPTOR,
*PUSBDRV_ENDPOINT_DESCRIPTOR;
```

USBDRV_STRING_DESCRIPTOR

```
typedef struct _USBDRV_STRING_DESCRIPTOR {
        UCHAR bLength;
        UCHAR bDescriptorType;
        WCHAR bString[1];
        }USBDRV_STRING_DESCRIPTOR,
        *PUSBDRV_STRING_DESCRIPTOR;
```

EZUSB and FX

The following examples are provided:

BulkLoop - provides several in/out bulk pairs and one interrupt pair. Data written to the out endpoint is copied to the corresponding in endpoint. Uses a relatively slow byte copy to do this.

BulkWrite - provides several out bulk endpoints. Data written to the device is discarded immediately making the endpoint available for the next 1ms USB frame.

Bulkread - provides several in endpoints. The firmware loads an incrementing byte sequence into each endpoint buffer. Uses a relatively slow byte copy to load the buffers.

FX2

The following example is provided:

BulkLoop - provides several in/out bulk pairs and one interrupt pair. Data written to the out endpoint is copied to the corresponding in endpoint. Uses a relatively slow byte copy to do this.

More examples will be provided in future releases.

Keil Monitor

The Keil monitor can be set and loaded from the control panel. *Set Monitor* and *Load Monitor* will load the monitor. Make sure the correct target is selected! Please see the control panel help for more information.

The Cypress USB development kit for the EX-USB, FX, FX2, and SX2 includes monitors for use with the Keil debugger and UV2. They are located in the **<Install Dir>\target\monitor** directory. Please see the readme.txt in this directory for specifics.

Here is an overview:

mon-ext-sio1-e0.hex
The 64K monitor for the Keil debugger for the EZ-USB and FX.
This version uses sio1 and external memory from 0xe000-0xef75.

mon-ext-sio1-c0.hex
The 64K monitor for the Keil debugger for the FX2.
This version uses sio1 and external memory from 0xc000-0xcf75.

mon-ext-sio0-c0.hex
The 64K monitor for the Keil debugger for the EZ-USB, FX and FX2.
This version uses sio0 and external memory from 0xc000-0xcf75.

mon-int-sio1.hex
A version of the monitor for the Keil debugger that uses only internal memory.
This version uses sio 1 and loads at 0x0000-0x1075.

mon-int-sio0.hex
A version of the monitor for the Keil debugger that uses only internal memory.
This version uses sio 0 and loads at 0x0000-0x1075.

Index