EZ-USB General Purpose Driver Specification

2/23/99

1. Introduction

The EZ-USB General Purpose Driver (referred to throughout this document as the GPD or the EZ-USB GPD) is a general purpose device driver that can be used to interface with an EZ-USB based peripheral. The driver provides a user mode interface to common USB device requests and data transfers. The driver serves two purposes. First, it aids the device or firmware developer. Coupled with the EZ-USB control panel, it allows a developer to test his device's ability to perform standard USB device requests and data transfers. Second, it provides example code for USB device driver development. A custom driver or mini-driver can be created using the general purpose driver as a starting point.

1.1 Objective of the Specification

The purpose of this document is to describe the interface to the EZ-USB GPD from a user mode application perspective. User mode applications are programs like Word, Internet Explorer and Photoshop. User mode applications have access to a wide variety of operating system services. They can access files, manipulate data and interact with the user through the Windows GUI. However, user mode applications are not allowed to directly access hardware. To access hardware a user mode application must go through an intermediate agent, the device driver. Device drivers typically run in a privileged execution mode called kernel mode. The EZ-USB GPD is a kernel mode device driver.

For those interested in understanding the inner workings of a kernel mode USB driver, the source code for the EZ-USB GPD is provided with the EZ-USB Development Kit.

1.2 Intended Audience

The intended audience for this specification are developers who need to write a custom application to communicate with their USB device using the EZ-USB GPD. Readers should be proficient at C programming and should have a good understanding of USB. An appendix at the end of this document provides sources for further information.

1.3 EZ-USB Interface Support

The EZ-USB GPD can communicate with a single device interface at a time. It will be possible to select any interface/alternate setting that the device has, however it will not be possible to communicate with multiple interfaces simultaneously on multi-interface devices. Support for multiple interface devices will be handled using multiple instances of the driver. This is alluded to in the Microsoft document on USB Plug and Play (USB Multi-Configuration Driver). At enumeration time, the EZ-USB GPD will attempt to select Interface 0 alternate setting 0.

2. Building the EZ-USB General Purpose Device Driver

Building the GPD requires the Microsoft WDM DDK (Windows 98 DDK or Windows 2000 Beta DDK) and Microsoft Visual C++ 5.0. The DDK is available for download from Microsoft at http://www.microsoft.com/hwdev

The WDM DDK is only required if you plan to modify the EZ-USB GPD. If you don't need to modify the driver, it is available in compiled form in the file ezusb.sys.

For User mode applications, you may use any compiler that supports the Win32 functions CreateFile() and DeviceIoControl(). The provided sample code is targeted at Microsoft Visual C++ 5.0.

3. Loading the EZ-USB General Purpose Device Driver

This section will describe the basics of how a USB device driver gets loaded and the specifics of loading the EZ-USB GPD for your device. As always, for a detailed explanation you should consult Microsoft's documentation.

3.1 INF File Basics

All USB devices have a Vendor ID (VID) and a Product ID (PID) which are reported to Windows in the device descriptor. Windows uses the VID and PID to find the appropriate device driver. The INF file is what ties a VID/PID combination to a specific driver.

```
;
;       FILE:   EZUSB.INF
;

[Version]
signature="$CHICAGO$"
Class=USB
Provider=%Anchor%
LayoutFile=LAYOUT.INF

[Manufacturer]
%Anchor%=Anchor

[PreCopySection]
HKR,,NoSetupUI,,1

[DestinationDirs]
DefaultDestDir=11

[Anchor]
%USB\VID_0547&PID_0080.DeviceDesc%=EZUSBDEV, USB\VID_0547&PID_0080
%USB\VID_0547&PID_0080.DeviceDesc%=EZUSBDEV, USB\VID_0547&PID_1002

[ControlFlags]
ExcludeFromSelect=*             ; removes all devices here from the device installer list

[EZUSBDEV]
AddReg=EZUSBDEV.AddReg

[EZUSBDEV.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,ezusb.sys


[Strings]
Anchor="AnchorChips"
USB\VID_0547&PID_0080.DeviceDesc="Anchor Chips EZ-USB Development Board (ezusb.sys)"
```

3.2 The Registry

An INF file is used to associate a device with a device driver the first time the device is attached to the system. This information is then stored in the system registry so that subsequent device attachments will complete faster and not require user interaction. Most of the interesting USB information is stored in the system registry under the key

\HKEY_LOCAL_MACHINE\Enum\USB

The system registry can be viewed and modified using the utility regedit.exe, which is shipped with the operating system. Use caution when modifying the registry.

4. User Mode Interface to the GPD

All User mode access to the EZ-USB GPD is through I/O Control calls. A user mode application first gets a handle to the device driver via a call to the Win32 function CreateFile(). The user mode application then uses the Win32 function DeviceIoControl() to submit an I/O control code and related input and output buffers to the driver through the handle returned by CreateFile().

4.1 Symbolic Links

The EZ-USB GPD can communicate with multiple EZ-USB devices. For each EZ-USB device attached to the host, the driver creates a symbolic link of the form **ezusb-*i***, where ***i*** is an instance index starting at 0. If there are 3 EZ-USB devices attached to the host, then there will exist 3 symbolic links: **ezusb-0**, **ezusb-1** and **ezusb-2**. The symbolic link name is used when calling CreateFile() to get a handle to the device driver. What CreateFile() is really doing is getting a handle to the device object created by the device driver. The following code sample demonstrates getting a handle to EZ-USB device **ezusb-0**.

```
HANDLE DeviceHandle;

DeviceHandle = CreateFile("\\\\.\\ezusb-0",
      GENERIC_WRITE,
      FILE_SHARE_WRITE,
      NULL,
      OPEN_EXISTING,
      0,
      NULL);
```

4.2 Device I/O Control

User mode applications use the Win32 function DeviceIoControl() to send requests to device drivers. The following code sample demonstrates submitting a request to the EZ-USB GPD to read the device's USB Device Descriptor. It uses the DeviceHandle returned in the previous example.

```
PVOID  pvBuffer = NULL;
DWORD  nBytes = 0;

PvBuffer = malloc(sizeof (Usb_Device_Descriptor));

bResult = DeviceIoControl (DeviceHandle,
                  IOCTL_EZUSB_GET_DEVICE_DESCRIPTOR,
                  NULL,                                 // no input buffer
                  0,                                    // input buffer size
                  pvBuffer,                             // buffer to hold the device descriptor
                  sizeof (Usb_Device_Descriptor),       // size of the output buffer
                  &nBytes,                              // actual bytes returned
                  NULL);                                // not overlapped
```

5. I/O Control Code (IOCTL) Reference

The following sections describe in detail the various IOCTLs supported by the EZ-USB GPD. For more information on CreateFile() and DeviceIoControl() consult Microsoft's Win32 documentation.

Anchor Chips has also provided several small example programs that demonstrate user mode interaction with the EZ-USB GPD.

The following table shows the function prototype for DeviceIoControl(). The EZ-USB GPD IOCTL reference will use the same function argument names.

```
BOOL DeviceIoControl(
      HANDLE       hDevice,            // handle to device of interest
      DWORD        dwIoControlCode,    // control code of operation to perform
      LPVOID       lpInBuffer,         // pointer to buffer to supply input data
      DWORD        nInBufferSize,      // size of input buffer
      LPVOID       lpOutBuffer,        // pointer to buffer to receive output data
      DWORD        nOutBufferSize,     // size of output buffer
      LPDWORD      lpBytesReturned,    // pointer to variable to receive output
                                       // byte count
      LPOVERLAPPED lpOverlapped        // pointer to overlapped structure for
                                       // asynchronous operation
);
```

EZ-USB GPD IOCTLs and their corresponding input/output structures are defined in the Anchor Chips provided header file EZUSBSYS.H. Standard USB structures are defined in the file USB100.H which is provided with the WDM DDK.

5.1 Standard Device Request IOCTLs

The standard device request IOCTLs are used to perform the standard USB device requests as defined in Chapter 9 of the *Universal Serial Bus Specification Version 1.0*.

5.1.1 IOCTL_Ezusb_GET_DEVICE_DESCRIPTOR

Causes the driver to issue the USB standard device request GET_DESCRIPTOR of type DEVICE to the device.

dwIoControlCode	IOCTL_Ezusb_GET_DEVICE_DESCRIPTOR
lpInBuffer	NULL
nInBufferSize	0
lpOutBuffer	Pointer to a user allocated buffer to receive the device descriptor. Must be large enough to hold the entire USB Device Descriptor as defined in USB100.H
nOutBufferSize	sizeof(USB_DEVICE_DESCRIPTOR)

5.1.2 IOCTL_Ezusb_GET_CONFIGURATION_DESCRIPTOR

Causes the driver to issue the USB standard device request GET_DESCRIPTOR of type CONFIGURATION to the device. This request returns not only the configuration descriptor, but also interface, endpoint and class specific descriptors.

dwIoControlCode	IOCTL_Ezusb_GET_ CONFIGURATION_DESCRIPTOR
lpInBuffer	NULL
nInBufferSize	0
lpOutBuffer	Pointer to a user allocated buffer to receive the configuration and additional descriptors. If the size of all descriptors exceeds the size of this buffer, only *nOutBufferSize* bytes of data will be returned. The total size of all descriptors is returned in *wTotalLength* field of the configuration descriptor. Since it is not possible to know the size of the descriptor data in advance, call this function first with a small buffer in order to determine the actual size of descriptor data and then call it again with an appropriately sized buffer.
nOutBufferSize	sizeof(lpOutBuffer)

5.1.3 IOCTL_Ezusb_GET_STRING_DESCRIPTOR

Read a string descriptor from the USB device.

dwIoControlCode	IOCTL_Ezusb_GET_STRING_DESCRIPTOR

lpInBuffer	Pointer to a GET_STRING_DESCRIPTOR_IN structure.
	
	`typedef struct _GET_STRING_DESCRIPTOR_IN`
	`{`
	`  UCHAR  Index;`
	`  USHORT  LanguageId;`
	`} GET_STRING_DESCRIPTOR_IN, *PGET_STRING_DESCRIPTOR_IN;`
	
	Index is the index of the desired string descriptor. *LanguageId* corresponds to the Microsoft specified LANGID value and determines the language of the string.

nInBufferSize	sizeof(GET_STRING_DESCRIPTOR_IN)

lpOutBuffer	Pointer to a user allocated buffer to receive the string descriptor. If the size of the descriptor exceeds the size of this buffer, only *nOutBufferSize* bytes of data will be returned. The total size of the string is returned in the *bLength* field of the string descriptor. Since it is not possible to know the size of the string descriptor in advance, call this function first with a small buffer in order to determine the actual size of descriptor data and then call it again with an appropriately sized buffer.
nOutBufferSize	sizeof(lpOutBuffer)

5.1.4 IOCTL_Ezusb_SETINTERFACE

Use this IOCTL to switch between different alternate settings.

dwIoControlCode	IOCTL_Ezusb_SETINTERFACE

lpInBuffer	Pointer to a SET_INTERFACE_IN structure.
	
	`typedef struct _SET_INTERFACE_IN`
	`{`
	`  UCHAR interfaceNum;`
	`  UCHAR alternateSetting;`
	`} SET_INTERFACE_IN, *PSET_INTERFACE_IN;`
	
	interfaceNum corresponds to the *bInterfaceNumber* field in the interface descriptor of the desired alternate setting. *alternateSetting* corresponds to the *bAlternateSetting* field in the interface descriptor of the desired alternate setting.

nInBufferSize	sizeof(SET_INTERFACE_IN)
lpOutBuffer	NULL
nOutBufferSize	0

5.2 Data Transfer IOCTLs

5.2.1 IOCTL_EZUSB_BULK_READ

Read (USB IN) from the specified bulk or interrupt pipe. This IOCTL will block the calling thread until the data transfer completes.

dwIoControlCode	IOCTL_EZUSB_BULK_READ

lpInBuffer	Pointer to a BULK_TRANSFER_CONTROL structure.
	
	`typedef struct _BULK_TRANSFER_CONTROL`
	`{`
	`  ULONG pipeNum;`
	`} BULK_TRANSFER_CONTROL, *PBULK_TRANSFER_CONTROL;`

nInBufferSize	sizeof(BULK_TRANSFER_CONTROL)
lpOutBuffer	Buffer to hold data read from the device.
nOutBufferSize	sizeof(lpOutBuffer) This parameter determines the total size of the USB transfer. The transfer size must be less than 64KB.

5.2.2 IOCTL_EZUSB_BULK_WRITE

Write (USB OUT) to the specified bulk pipe. This IOCTL will block the calling thread until the data transfer completes.

dwIoControlCode	IOCTL_EZUSB_BULK_WRITE

lpInBuffer	Pointer to a BULK_TRANSFER_CONTROL structure.
	
	`typedef struct _BULK_TRANSFER_CONTROL`
	`{`
	`  ULONG pipeNum;`
	`} BULK_TRANSFER_CONTROL, *PBULK_TRANSFER_CONTROL;`

nInBufferSize	sizeof(BULK_TRANSFER_CONTROL)
lpOutBuffer	Buffer of data to write to the device.
nOutBufferSize	sizeof(lpOutBuffer) This parameter determines the total size of the USB transfer. The transfer size must be less than 64KB.

5.2.3 IOCTL_EZUSB_ISO_READ

Read (USB IN) from the specified isochronous pipe. This IOCTL will block the calling thread until the data transfer completes. Due to a quirk in the USB host controller driver (HCD) an ISO pipe must be reset before starting a new transfer. This can be accomplished using IOCTL_Ezusb_RESETPIPE which is documented elsewhere in this document.

dwIoControlCode	IOCTL_EZUSB_ISO_READ

lpInBuffer	Pointer to an ISO_TRANSFER_CONTROL structure.
	
	`typedef struct _ISO_TRANSFER_CONTROL`
	`{`
	`  ULONG PipeNum;`
	`  ULONG PacketSize;`
	`  ULONG PacketCount;`
	`  ULONG FramesPerBuffer;`
	`  ULONG BufferCount;`
	`} ISO_TRANSFER_CONTROL, *PISO_TRANSFER_CONTROL;`
	
	PipeNum is the zero based pipe number associated with the endpoint to read from.
	PacketSize is the amount of ISO data to read during each frame. This value usually corresponds to the max packet size of the ISO endpoint, but can be less.
	PacketCount is the number of frames of ISO data to read from the device. ISO transfers occur every USB frame (1ms). For example, a *PacketCount* of 3000 would indicate 3 seconds of ISO data. This parameter must be evenly divisible by the product of the next two parameters that is:
	(*PacketCount* mod (*FramesPerBuffer* * *BufferCount*)) must be zero.
	FramesPerBuffer is the number of USB frames of data to transfer in a single URB (USB Request Block). 10 is a good default value.
	BufferCount is the number of transfer URBs to use for this transfer. 2 is a good default value.
	
	The last two parameters probably deserve some explanation. In order to maintain an ISO stream, the driver must maintain at least two sets of ISO transfer buffers, so that when one set completes the next set of transfers can be ready to go. The driver ping pongs between these two sets of frame buffers until the transfer completes. *BufferCount* determines how many buffer sets the driver ping pongs between, and *FramesPerBuffer* determines how many frames of USB data are represented by each of those buffers.

nInBufferSize	sizeof(ISO_TRANSFER_CONTROL)

lpOutBuffer	Buffer to hold data read from the device and to hold the isochronous transfer descriptors for the transfer. The size of this buffer should be:
	
	(*PacketCount* *
	(*PacketSize* + sizeof(USBD_ISO_PACKET_DESCRIPTOR))
	
	Upon return, the begnning of this buffer will contain the actual data read from the device. The actual data is followed by an array of USBD_ISO_PACKET_DESCRIPTOR structures. This array can be used by the caller to determine the results of the transfer on a per packet basis.
	
	`typedef struct _USBD_ISO_PACKET_DESCRIPTOR`
	`{`
	`ULONG Offset;`
	`ULONG Length;`
	`USBD_STATUS Status;`
	`} USBD_ISO_PACKET_DESCRIPTOR, *PUSBD_ISO_PACKET_DESCRIPTOR;`

nOutBufferSize	size of lpOutBuffer

5.2.4 IOCTL_EZUSB_ISO_WRITE

Write (USB OUT) to the specified isochronous pipe. This IOCTL will block the calling thread until the data transfer completes. Due to a quirk in the USB host controller driver (HCD) an ISO pipe must be reset before starting a new transfer. This can be accomplished using IOCTL_Ezusb_RESETPIPE which is documented elsewhere in this document.

dwIoControlCode	IOCTL_EZUSB_ISO_WRITE

lpInBuffer	Pointer to an ISO_TRANSFER_CONTROL structure.
	
	`typedef struct _ISO_TRANSFER_CONTROL`
	`{`
	`ULONG PipeNum;`
	`ULONG PacketSize;`
	`ULONG PacketCount;`
	`ULONG FramesPerBuffer;`
	`ULONG BufferCount;`
	`} ISO_TRANSFER_CONTROL, *PISO_TRANSFER_CONTROL;`
	
	PipeNum is the zero based pipe number associated with the endpoint to write to.
	PacketSize is the amount of ISO data to write during each frame. This value usually corresponds to the max packet size of the ISO endpoint, but can be less.

	PacketCount is the number of frames of ISO data to write to the device. ISO transfers occur every USB frame (1ms). For example, a *PacketCount* of 3000 would indicate 3 seconds of ISO data. This parameter must be evenly divisible by the product of the next two parameters that is: (*PacketCount* mod (*FramesPerBuffer* * *BufferCount*)) must be zero. *FramesPerBuffer* is the number of USB frames of data to transfer in a single URB (USB Request Block). 10 is a good default value. *BufferCount* is the number of transfer URBs to use for this transfer. 2 is a good default value. The last two parameters probably deserve some explanation. In order to maintain an ISO stream, the driver must maintain at least two sets of ISO transfer buffers, so that when one set completes the next set of transfers can be ready to go. The driver ping pongs between these two sets of frame buffers until the transfer completes. *BufferCount* determines how many buffer sets the driver ping pongs between, and *FramesPerBuffer* determines how many frames of USB data are represented by each of those buffers.
nInBufferSize	sizeof(ISO_TRANSFER_CONTROL)

lpOutBuffer	Buffer to write to the device and to hold the isochronous transfer descriptors for the transfer. The size of this buffer should be:
	
	(*PacketCount* *
	(*PacketSize* + sizeof(USBD_ISO_PACKET_DESCRIPTOR))
	
	The begnning of this buffer will contain the data to write to the device. The actual data is followed by an array of USBD_ISO_PACKET_DESCRIPTOR structures. This array can be used by the caller to determine the results of the transfer on a per packet basis.
	
	`typedef struct _USBD_ISO_PACKET_DESCRIPTOR`
	`{`
	`  ULONG Offset;`
	`  ULONG Length;`
	`  USBD_STATUS Status;`
	`} USBD_ISO_PACKET_DESCRIPTOR, *PUSBD_ISO_PACKET_DESCRIPTOR;`

nOutBufferSize	size of lpOutBuffer

5.2.5 IOCTL_EZUSB_START_ISO_STREAM

Starts an IN ISO stream which is maintained at the driver level. ISO streaming will continue until a call is made to IOCTL_EZUSB_STOP_ISO_STREAM.

- As ISO data is read, it is copied into a buffer for retrieval from user mode via a call to IOCTL_EZUSB_READ_ISO_BUFFER. If this buffer fills up, new data is discarded until room is available in the buffer. Buffer size is determined by the PacketCount parameter.
- Prior to starting an ISO stream, you must reset the ISO pipe using IOCTL_Ezusb_RESETPIPE.
- The driver only supports a single ISO IN stream.

dwIoControlCode	IOCTL_EZUSB_START_ISO_STREAM

lpInBuffer	Pointer to an ISO_TRANSFER_CONTROL structure.
	
	`typedef struct _ISO_TRANSFER_CONTROL`
	`{`
	`  ULONG PipeNum;`
	`  ULONG PacketSize;`
	`  ULONG PacketCount;`
	`  ULONG FramesPerBuffer;`
	`  ULONG BufferCount;`
	`} ISO_TRANSFER_CONTROL, *PISO_TRANSFER_CONTROL;`
	
	PipeNum is the zero based pipe number associated with the endpoint to read from.
	PacketSize is the amount of ISO data to read during each frame. This value usually corresponds to the max packet size of the ISO endpoint.
	PacketCount determines the size of the buffer that arriving ISO data will be copied to. For example, a *PacketCount* of 3000 would indicate 3 seconds of ISO data. This parameter must be evenly divisible by the product of the next two parameters that is: (*PacketCount* mod (*FramesPerBuffer* * *BufferCount*)) must be zero.
	FramesPerBuffer is the number of USB frames of data to transfer in a single URB (USB Request Block). 10 is a good default value.
	BufferCount is the number of transfer URBs to use for this transfer. 2 is a good default value.
	
	The last two parameters probably deserve some explanation. In order to maintain an ISO stream, the driver must maintain at least two sets of ISO transfer buffers, so that when one set completes the next set of transfers can be ready to go. The driver ping pongs between these two sets of frame buffers until the transfer completes. *BufferCount* determines how many buffer sets the driver ping pongs between, and *FramesPerBuffer*

	determines how many frames of USB data are represented by each of those buffers.
nInBufferSize	sizeof(ISO_TRANSFER_CONTROL)
lpOutBuffer	NULL
nOutBufferSize	0

5.2.6 IOCTL_EZUSB_STOP_ISO_STREAM

Sets a flag in the driver that will stop the ISO IN stream started with IOCTL_EZUSB_START_ISO_STREAM.

dwIoControlCode	IOCTL_EZUSB_START_ISO_STREAM
lpInBuffer	NULL
nInBufferSize	0
lpOutBuffer	NULL
nOutBufferSize	0

5.2.7 IOCTL_EZUSB_ READ_ISO_BUFFER

This call is used in conjuntion with IOCTL_EZUSB_START_ISO_STREAM. Once an ISO stream has been started, this IOCTL may be used to retrieve any data that has been read.

dwIoControlCode	IOCTL_EZUSB_ READ_ISO_BUFFER

lpInBuffer	Pointer to an ISO_TRANSFER_CONTROL structure.
	
	`typedef struct _ISO_TRANSFER_CONTROL`
	`{`
	`ULONG PipeNum;`
	`ULONG PacketSize;`
	`ULONG PacketCount;`
	`ULONG FramesPerBuffer;`
	`ULONG BufferCount;`
	`} ISO_TRANSFER_CONTROL, *PISO_TRANSFER_CONTROL;`
	
	PipeNum ignored
	PacketSize is the amount of ISO data to read during each frame. This value usually corresponds to the max packet size of the ISO endpoint, but can be less. Value should be identical to the value

	used in the corresponding call to IOCTL_EZUSB_START_ISO_STREAM. *PacketCount* is the maximum number of packets of data to read from the ISO buffer. *FramesPerBuffer* ignored *BufferCount* ignored
nInBufferSize	sizeof(ISO_TRANSFER_CONTROL)

lpOutBuffer	Buffer to hold data read from the device and to hold the isochronous transfer descriptors for the transfer. The size of this buffer should be:
	
	(*PacketCount* *
	(*PacketSize* + sizeof(USBD_ISO_PACKET_DESCRIPTOR))
	
	Upon return, the begnning of this buffer will contain the actual data read from the device. The actual data is followed by an array of USBD_ISO_PACKET_DESCRIPTOR structures. This array can be used by the caller to determine the results of the transfer on a per packet basis.
	
	`typedef struct _USBD_ISO_PACKET_DESCRIPTOR`
	`{`
	`  ULONG Offset;`
	`  ULONG Length;`
	`  USBD_STATUS Status;`
	`} USBD_ISO_PACKET_DESCRIPTOR, *PUSBD_ISO_PACKET_DESCRIPTOR;`

nOutBufferSize	size of lpOutBuffer

5.3 Miscellaneous IOCTLs

5.3.1 IOCTL_EZUSB_VENDOR_OR_CLASS_REQUEST

Sends a Vendor or Class specific request to the control endpoint.

dwIoControlCode	IOCTL_EZUSB_VENDOR_OR_CLASS_REQUEST

lpInBuffer	Pointer to a VENDOR_OR_CLASS_REQUEST_CONTROL structure.
	
	`typedef struct _VENDOR_OR_CLASS_REQUEST_CONTROL`
	`{`
	`  // transfer direction (0=host to device, 1=device to host)`
	`  UCHAR direction;`
	
	`  // request type (1=class, 2=vendor)`

	UCHAR requestType;
	
	// recipient (0=device,1=interface,2=endpoint,3=other)
	UCHAR recepient;
	//
	// see the USB Specification for an explanation of the
	// following paramaters.
	//
	UCHAR requestTypeReservedBits;
	UCHAR request;
	USHORT value;
	USHORT index;
	} VENDOR_OR_CLASS_REQUEST_CONTROL,
	*PVENDOR_OR_CLASS_REQUEST_CONTROL;

nInBufferSize	sizeof(VENDOR_OR_CLASS_REQUEST_CONTROL)
lpOutBuffer	Depending on the transfer direction, this field points to a buffer of data to send to the control endpoint or to receive data read from the endpoint. For dataless control transfers, this parameter will be NULL.
nOutBufferSize	sizeof(lpOutBuffer) This parameter determines the total size of the data phase of the control transfer. For dataless control transfers it should be 0.

5.3.2 IOCTL_EZUSB_GET_CURRENT_FRAME_NUMBER

Returns the current USB frame number. This IOCTL does not cause any USB transfers to occur. It simply queries the host controller driver for the current frame number.

dwIoControlCode	IOCTL_EZUSB_GET_CURRENT_FRAME_NUMBER
lpInBuffer	NULL
nInBufferSize	0
lpOutBuffer	Pointer to a ULONG. The current frame number is returned here.
nOutBufferSize	sizeof(ULONG)

5.3.3 IOCTL_Ezusb_GET_PIPE_INFO

Returns an Interface Information structure describing the pipes of the currently selected interface and alternate setting.

dwIoControlCode	IOCTL_Ezusb_GET_PIPE_INFO
lpInBuffer	NULL
nInBufferSize	0
lpOutBuffer	Pointer to a user allocated buffer to receive the Interface Information structure. This buffer should be sufficiently large to hold the entire structure. The interface information strucure is defined by USBD_INTERFACE_INFORMATION in USBDI.H which is part of the WDM DDK.
nOutBufferSize	sizeof(lpOutBuffer)

5.3.4 IOCTL_Ezusb_RESETPIPE

Resets the specified pipe.

dwIoControlCode	IOCTL_Ezusb_RESETPIPE
lpInBuffer	Pointer to a ULONG containing the pipe number to reset.
nInBufferSize	sizeof(ULONG)
lpOutBuffer	NULL
nOutBufferSize	0

5.3.5 IOCTL_Ezusb_ABORTPIPE

Aborts all pending transfers on the specified pipe.

dwIoControlCode	IOCTL_Ezusb_ABORTPIPE
lpInBuffer	Pointer to a ULONG containing the pipe number to abort.
nInBufferSize	sizeof(ULONG)

lpOutBuffer	NULL
nOutBufferSize	0

5.3.6 IOCTL_EZUSB_GET_DRIVER_VERSION

Returns version information about the EZ-USB device driver.

dwIoControlCode	IOCTL_EZUSB_GET_DRIVER_VERSION
lpInBuffer	NULL
nInBufferSize	0

lpOutBuffer	Pointer to a EZUSB_DRIVER_VERSION structure.
	`typedef struct _EZUSB_DRIVER_VERSION`
	`{`
	`  WORD  MajorVersion;`
	`  WORD  MinorVersion;`
	`  WORD  BuildVersion;`
	`} EZUSB_DRIVER_VERSION, *PEZUSB_DRIVER_VERSION;`

nOutBufferSize	sizeof(EZUSB_DRIVER_VERSION)

5.4 EZ-USB Specific IOCTLs

5.4.1 IOCTL_Ezusb_ANCHOR_DOWNLOAD

Downloads data to EZ-USB RAM starting at address 0.

dwIoControlCode	IOCTL_Ezusb_ANCHOR_DOWNLOAD
lpInBuffer	Buffer of data to download to EZ-USB RAM
nInBufferSize	Size of the download buffer. Must be <= 7KB.
lpOutBuffer	NULL
nOutBufferSize	0

5.4.2 IOCTL_EZUSB_ANCHOR_DOWNLOAD

Downloads data to EZ-USB RAM starting at the specified address. This IOCTL will only download to the EZ-USB's internal RAM.

dwIoControlCode	IOCTL_EZUSB_ANCHOR_DOWNLOAD

lpInBuffer	Pointer to an ANCHOR_DOWNLOAD_CONTROL structure.
	
	`typedef struct _ANCHOR_DOWNLOAD_CONTROL`
	`{`
	`  WORD Offset;`
	`} ANCHOR_DOWNLOAD_CONTROL, *PANCHOR_DOWNLOAD_CONTROL;`
	
	Offset specifies the offset within EZ-USB RAM to download to.

nInBufferSize	sizeof(ANCHOR_DOWNLOAD_CONTROL)
lpOutBuffer	Buffer of data to download to EZ-USB RAM
nOutBufferSize	size of the output buffer. This parameter determines the size of the Anchor Download.