

Intel[®] Platform Controller Hub EG20T

USB Client Driver for Windows* Programmer's Guide

March 2011



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/#/en_US_01.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. Go to: <http://www.intel.com/products/processor%5Fnumber/> for details.

α Intel® Hyper-Threading Technology requires a computer system with a processor supporting Intel® HT Technology and an Intel® HT Technology-enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. For more information including details on which processors support Intel® HT Technology, see http://www.intel.com/products/ht/hyperthreading_more.htm.

β Intel® High Definition Audio requires a system with an appropriate Intel® chipset and a motherboard with an appropriate CODEC and the necessary drivers installed. System sound quality will vary depending on actual implementation, controller, CODEC, drivers and speakers. For more information about Intel® HD audio, refer to <http://www.intel.com/>.

γ 64-bit computing on Intel® architecture requires a computer system with a processor, chipset, BIOS, operating system, device drivers and applications enabled for Intel® 64 architecture. Performance will vary depending on your hardware and software configurations. Consult with your system vendor for more information.

δ Intel® Virtualization Technology requires a computer system with an enabled Intel® processor, BIOS, virtual machine monitor (VMM) and, for some uses, certain computer system software enabled for it. Functionality, performance or other benefits will vary depending on hardware and software configurations and may require a BIOS update. Software applications may not be compatible with all operating systems. Please check with your application vendor.

ε The original equipment manufacturer must provide Intel® Trusted Platform Module (Intel® TPM) functionality, which requires an Intel® TPM-supported BIOS. Intel® TPM functionality must be initialized and may not be available in all countries.

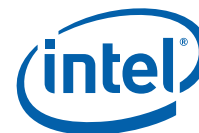
θ For Enhanced Intel SpeedStep® Technology, see the [Processor Spec Finder](#) or contact your Intel representative for more information.

I²C* is a two-wire communications bus/protocol developed by Philips. SMBus is a subset of the I²C* bus/protocol and was developed by Intel. Implementations of the I²C* bus/protocol may require licenses from various entities, including Philips Electronics N.V. and North American Philips Corporation.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Inside, Core Inside, i960, Intel, the Intel logo, Intel AppUp, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, the Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel Sponsors of Tomorrow., the Intel Sponsors of Tomorrow. logo, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, InTru, the InTru logo, InTru soundmark, Itanium, Itanium Inside, MCS, MMX, Moblin, Pentium, Pentium Inside, skool, the skool logo, Sound Mark, The Journey Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2011, Intel Corporation and/or its suppliers and licensors. All rights reserved.



Contents

1.0	Introduction	5
2.0	Operating System (OS) Support	6
3.0	Dependencies	7
4.0	USB Client Driver API Details	8
4.1	Features	8
4.2	Interface Details	8
4.3	IOCTL Usage Details	8
4.3.1	IOCTL_SET_USB_DESCRIPTOR	8
4.3.2	IOCTL_USB_GET_CONFIG_NUMBER	9
4.4	Structures and Enumerations	9
4.5	Error Handling	9
4.6	Inter-IOCTL dependencies	9
4.7	Supporting USB Data Transfer	9
4.7.1	Control Transfer	9
4.7.2	Bulk Transfer	9
4.7.2.1	ReadFile for Bulk Out	9
4.7.2.2	WriteFile for Bulk In	10
4.7.3	Interrupt Transfer	11
4.7.4	Isochronous Transfer	11
5.0	Programming Guide	12
5.1	Basic Flow	12
5.2	Opening the Device	13
5.2.1	Using GUID Interface Exposed by the Driver	14
5.3	Setting USB Client Configuration	14
5.3.1	Filling USB Descriptors	15
5.4	Getting Configured Number	21
5.5	Opening the Endpoint	21
5.5.1	Getting Endpoint Path Name	22
5.6	Reading Data	23
5.7	Writing Data	23
5.8	Closing the Endpoint	24
5.9	Closing the Device	24

Tables

1	Supported IOCTLs	8
2	Endpoint Path Number	22



Revision History

Date	Revision	Description
March 2011	002	Updated OS support in Section 2.0, "Operating System (OS) Support" on page 6 Updated Section 4.1, "Features" on page 8 Updated Table 1, "Supported IOCTLs" on page 8 Updated Section 4.7, "Supporting USB Data Transfer" on page 9 Updated Section 5.0, "Programming Guide" on page 12
September 2010	001	Initial release



1.0 Introduction

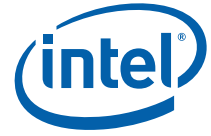
This document provides the programming details of the USB Client driver for Windows*. This includes information about the interfaces exposed by the driver and how to use those interfaces to drive the USB hardware.



2.0 Operating System (OS) Support

The USB Client driver is supported by the following operating systems:

No	OS	Notes
1	Microsoft Windows XP*	Service Pack 3
2	Windows Embedded Standard*	2009
3	Windows Embedded POSReady*	2009
4	Microsoft Window 7*	
5	Windows Embedded Standard 7*	



3.0 Dependencies

This driver is only dependent upon appropriate OS driver installation. Also, this driver is not dependent upon any other software delivered.



4.0 USB Client Driver API Details

This section provides information about the interfaces exposed by the USB Client driver. The current implementation of the driver's interfaces are exposed through Input/Output Controls (IOCTLs), which can be called from the application (user mode) using the DeviceIoControl Win32 API (refer to the MSDN documentation for more details on this API). The following sections provide information about the IOCTLs and how to use them to configure the USB hardware to work properly.

4.1 Features

The USB Client driver supports:

- High-speed (480 MHz) and full-speed (12 MHz).
- Control transfer, bulk transfer, and interrupt transfer modes. It does not support isochronous transfer mode
- 4 IN and 4 OUT physical endpoints.
- User-configurable endpoint information (set at initialization).
- All the standard device requests.
Requests other than those listed below, are responded to automatically by hardware.
 - GET_DESCRIPTOR is responded by software.
 - SET_DESCRIPTOR is not supported (returns STALL).
 - SYNC_FRAME is not supported (returns STALL).
 - Extension of class requests is possible.
 - Extension of vendor requests is possible.
- DMA in all endpoint.
- Hot Plug.

4.2 Interface Details

Table 1 lists IOCTLs supported by the driver.

Table 1. Supported IOCTLs

No	IOCTL	Remarks
1	IOCTL_SET_USB_DESCRIPTOR	Set the USB descriptor configuration information
2	IOCTL_USB_GET_CONFIG_NUMBER	Get a configured number after USB enumeration.

4.3 IOCTL Usage Details

This section assumes a single client model where there is a single application-level program configuring the USB Client interface and initiating I/O operations. The following file contains the details of the IOCTLs and data structures used.

- `ioh_udc_ioctl.h` – contains IOCTL definitions

4.3.1 IOCTL_SET_USB_DESCRIPTOR

This IOCTL is called to configure the USB Client controller with the descriptor information, such as device descriptor, configuration descriptor and interface descriptor.



4.3.2 IOCTL_USB_GET_CONFIG_NUMBER

This IOCTL is called to get a configuration number. When this USB client does not connect to the host, this request gets zero. After this USB client connects to the host and the emulation completes, this gets the configured number, which was set by SET_CONFIGURATION of USB standard request.

4.4 Structures and Enumerations

The USB Client driver does not define structures or enumerations for the API. The structures and enumerations are defined by Windows* Driver Development Kit.

4.5 Error Handling

Since the IOCTL command is implemented using Windows* API, the return value of the call is dependent on and defined by the OS. In Windows*, the return value will be a nonzero value. If the error is detected within or outside the driver, an appropriate system defined value will be returned by the driver.

4.6 Inter-IOCTL dependencies

There are no inter-IOCTL dependencies. Once this driver has been loaded successfully, the IOCTLs above can be used in any order.

4.7 Supporting USB Data Transfer

Four transfer types are defined with USB specification. This section describes the USB data transfer supported by the USB Client driver.

- Control transfer
- Bulk transfer
- Interrupt transfer

4.7.1 Control Transfer

The USB Client driver responds to all USB standard requests. An application does not have to handle Control Transfer.

Note: The USB Client driver does not support SET_DESCRIPTOR and SET_INTERFACE of the USB standard request. The driver responds normally to the request but it does not process the request. The USB Client driver does not support USB class requests and USB vendor requests.

4.7.2 Bulk Transfer

For Bulk OUT transfer, the USB client controller receives data from the host. An application should read data from USB client controller, using the ReadFile function of the Win32 API.

For Bulk IN transfer, an application should write data to the USB client controller, using the WriteFile function of the Win32 API. The USB client controller sends data to a host.

4.7.2.1 ReadFile for Bulk Out

This is the syntax of ReadFile Win32 API.



```
BOOL WINAPI ReadFile(  
    __in        HANDLE hFile,  
    __out       LPVOID lpBuffer,  
    __in        DWORD nNumberOfBytesToRead,  
    __out_opt   LPDWORD lpNumberOfBytesRead,  
    __inout_opt LPOVERLAPPED lpOverlapped  
);
```

For the USB Client driver, the parameters are specified as follows:

<i>hFile</i> [in]	A handle to the endpoint of the USB client. The hFile parameter must be created with read access by the CreateFile function.
<i>lpBuffer</i> [out]	A pointer to the buffer that receives the data read from the USB client controller.
<i>nNumberOfBytesToRead</i> [in]	The maximum number of bytes to be read. An application can read data 32,768 bytes per read from the USB client driver. This number should be a multiple of the maximum packet size of this endpoint.
<i>lpNumberOfBytesRead</i> [out]	A pointer to the variable that receives the number of bytes read. This should be used to know the number of received data from the USB client controller.

When the ReadFile function is used with the USB client controller, it returns when one of the following conditions occur:

- The number of bytes requested is read.
- After a short packet is received, the end data of a short packet is read.
- An error occurs.

4.7.2.2 WriteFile for Bulk In

This is the syntax of WriteFile Win32 API.

```
BOOL WINAPI WriteFile(  
    __in        HANDLE hFile,  
    __in        LPCVOID lpBuffer,  
    __in        DWORD nNumberOfBytesToWrite,  
    __out_opt   LPDWORD lpNumberOfBytesWritten,  
    __inout_opt LPOVERLAPPED lpOverlapped  
);
```

For the USB Client driver, the parameters are specified as follows:



Parameter	Description
<i>hFile</i> [in]	A handle to the endpoint of the USB client. The <i>hFile</i> parameter must be created with the write access by the <code>CreateFile</code> function.
<i>lpBuffer</i> [in]	A pointer to the buffer containing the data to be written to the USB client.
<i>nNumberOfBytesToWrite</i> [in]	The number of bytes to be written to the file or device. A value of zero should not be specify to this variable. Write operations are limited to 32,768 bytes per write.

When the **WriteFile** function is used with the USB client controller, it returns when one of the following conditions occur:

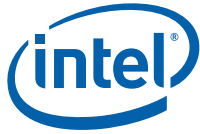
- The number of bytes requested is written.
- An error occurs.

4.7.3 Interrupt Transfer

For Interrupt OUT or IN transfer, an application should use the **ReadFile** or **WriteFile** function of the Win32 API as well as Bulk OUT or IN transfer. For more detail, see the [Section 4.7.2.1](#) and [Section 4.7.2.2](#).

4.7.4 Isochronous Transfer

The USB Client driver does not support isochronous transfer.



5.0 Programming Guide

This section describes the basic procedure for using the USB Client driver from a user mode application. All operations are through the IOCTLs exposed by the USB Client driver. Refer to [Section 4.3](#) for details on the IOCTLs. The steps involved in accessing the USB client driver from the user mode application are described below:

- Open the device
- Set USB descriptors
- Get configured number
- Open the endpoint
- Read or write data
- Close the endpoint
- Close the device

5.1 Basic Flow

The basic flow to use this driver from a user mode application is to open the device, set USB descriptors, get configured number, open the endpoints, read/write data, close the endpoints, and close the device.

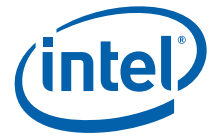
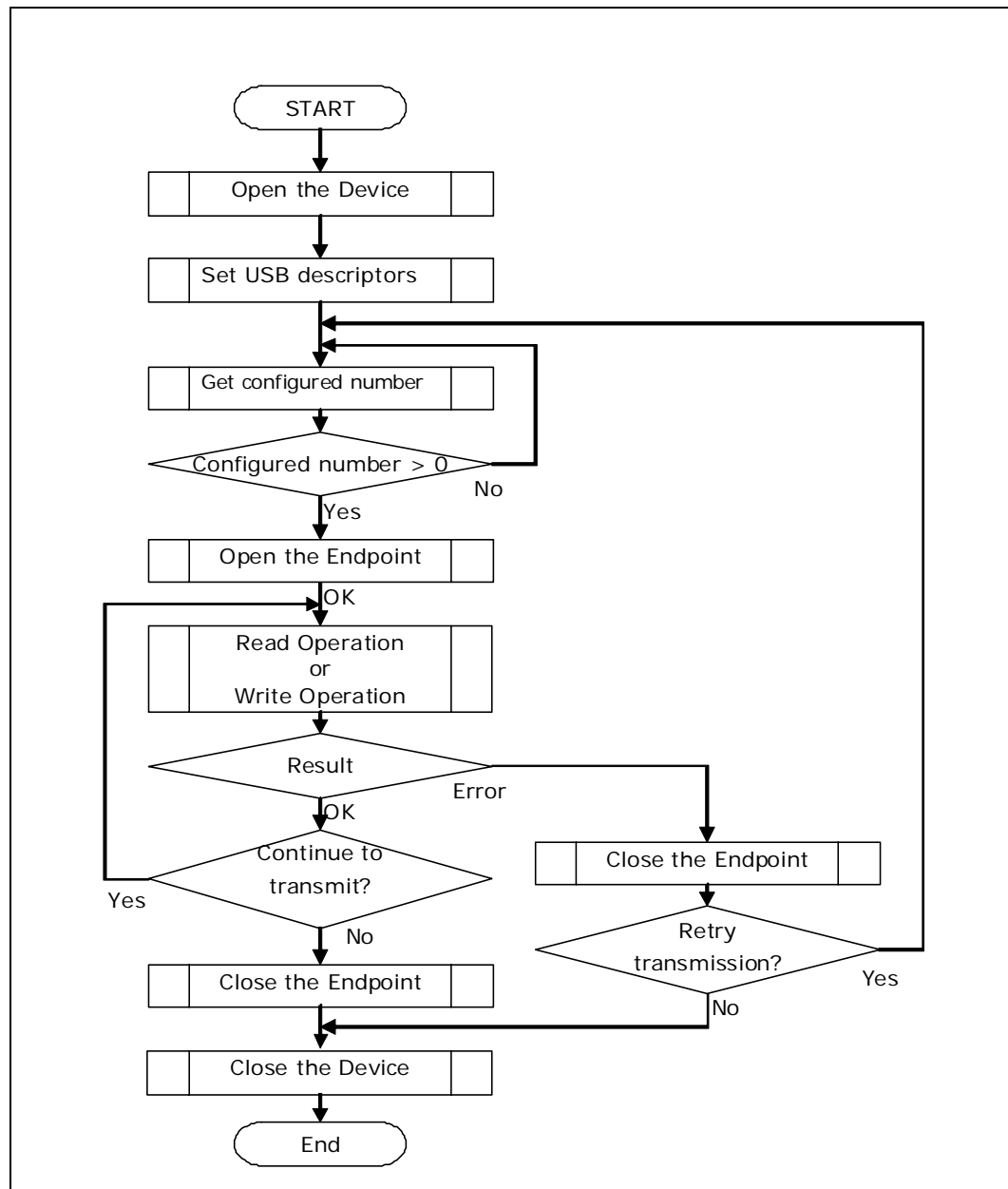


Figure 1. Basic Flow



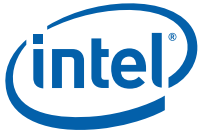
5.2 Opening the Device

The USB client is opened using **CreateFile** Win32 API. To get the device name, refer to Section 5.2.1.

```

/* Open the USB client driver */
GetDevicePath((LPGUID) &GUID_DEVINTERFACE_IOHUSB, DevicePath);

```



```
hDevice = CreateFile(DevicePath, GENERIC_WRITE, FILE_SHARE_WRITE,  
NULL, OPEN_EXISTING, 0, NULL);
```

5.2.1 Using GUID Interface Exposed by the Driver

A device interface class is a way of exporting device and driver functionality to other system components, including other drivers, as well as user-mode applications. A driver can register a device interface class, and then enable an instance of the class for each device object to which user-mode I/O requests might be sent. The Intel® PCH EG20T USB Client driver registers the following interface.

No	Interface Name
1	GUID_DEVINTERFACE_IOHUSB

This is defined in `ioh_udc_ioctl.h`.

Device interfaces are available to both kernel-mode components and user-mode applications. User-mode code can use **SetupDiXxx** functions to find out about registered, enabled device interfaces.

Please refer to the following site to get the details about SetupDiXxx functions.

<http://msdn.microsoft.com/en-us/library/ff549791.aspx>

5.3 Setting USB Client Configuration

`IOCTL_USB_SET_DESCRIPTOR` is used to initialize and configure the settings for the USB client.

DeviceIoControl Win32 API is used for sending information to the USB Client driver.

```
USB_DESCRIPTOR usbDescriptor = {0};  
  
/* Fill this descriptor with desired data */  
  
DWORD dwSize = 0;  
  
DeviceIoControl(hDevice,  
IOCTL_SET_USB_DESCRIPTOR,  
&usbDescriptor,  
sizeof(usbDescriptor),  
NULL,  
0,  
&dwSize,  
NULL);
```



5.3.1 Filling USB Descriptors

Define a descriptor structure as follows:

- USB_DEVICE_DESCRIPTOR is defined for High-Speed connection.
- USB_DEVICE_QUALIFIER_DESCRIPTOR is defined for Full-Speed connection.
- Two USB_CONFIGURATION_DESCRIPTOR are defined, each for HS connection and FS connection.
- Only one USB_INTERFACE_DESCRIPTOR is defined in the USB_CONFIGURATION_DESCRIPTOR.
- Six USB_ENDPOINT_DESCRIPTOR are able to be defined in the USB_CONFIGURATION_DESCRIPTOR.

```
#pragma pack(1)

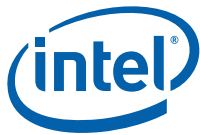
typedef struct _USB_DESCRIPTOR_EXAMPLE {
    USB_DEVICE_DESCRIPTOR usbDeviceDesc;
    USB_DEVICE_QUALIFIER_DESCRIPTOR usbDeviceQualDesc;

    struct _HighSpeedConfig1 {
        USB_CONFIGURATION_DESCRIPTOR usbConfigDesc;
        USB_INTERFACE_DESCRIPTOR usbInterfaceDesc;
        USB_ENDPOINT_DESCRIPTOR usbEndPointDesc1;
        USB_ENDPOINT_DESCRIPTOR usbEndPointDesc2;
    } HSConfig;

    struct _FullSpeedConfig1 {
        USB_CONFIGURATION_DESCRIPTOR usbOSpConfigDesc;
        USB_INTERFACE_DESCRIPTOR usbInterfaceDesc;
        USB_ENDPOINT_DESCRIPTOR usbEndPointDesc1;
        USB_ENDPOINT_DESCRIPTOR usbEndPointDesc2;
    } FSConfig;

    struct _UsbStringDescriptor1 {
        BYTE    bLength;
        BYTE    bDescriptorType;
        WORD    String[USB_STRING_SIZE1];
    } usbStringDesc1;

    struct _UsbStringDescriptor2 {
```



```
        BYTE    bLength;
        BYTE    bDescriptorType;
        WORD    String[USB_STRING_SIZE2];
    } usbStringDesc2;

    struct _UsbStringDescriptor3 {
        BYTE    bLength;
        BYTE    bDescriptorType;
        WORD    String[USB_STRING_SIZE3];
    } usbStringDesc3;
} USB_DESCRIPTOR_EXAMPLE, *USB_DESCRIPTOR_EXAMPLE;

#pragma pack()
```

Note: This structure must not include any excessive space. Therefore, this must be defined using '#pragma pack (1)'.

Fill USB_DEVICE_DESCRIPTOR and USB_DEVICE_QUALIFIER_DESCRIPTOR as follows:

```
/* Filling Device Descriptor */
pUSBDesc->usbDeviceDesc.bLength =
    sizeof(USB_DEVICE_DESCRIPTOR);
pUSBDesc->usbDeviceDesc.bDescriptorType=
    USB_DEVICE_DESCRIPTOR_TYPE;
pUSBDesc->usbDeviceDesc.bcdUSB= 0x0200;
pUSBDesc->usbDeviceDesc.bDeviceClass= 0;
pUSBDesc->usbDeviceDesc.bDeviceSubClass = 0;
pUSBDesc->usbDeviceDesc.bDeviceProtocol = 0;
pUSBDesc->usbDeviceDesc.bMaxPacketSize0 = 0x40;
pUSBDesc->usbDeviceDesc.idVendor= 0x****;
pUSBDesc->usbDeviceDesc.idProduct= 0x****;
pUSBDesc->usbDeviceDesc.bcdDevice= 0x001;
pUSBDesc->usbDeviceDesc.iManufacturer= 1;
pUSBDesc->usbDeviceDesc.iProduct= 2;
pUSBDesc->usbDeviceDesc.iSerialNumber= 0;
pUSBDesc->usbDeviceDesc.bNumConfigurations= 1;

/* Filling Device QualifierDescriptor */
```




```

pUSBDesc->usbDeviceQualDesc.bLength =
    sizeof(USB_DEVICE_QUALIFIER_DESCRIPTOR);
pUSBDesc->usbDeviceQualDesc.bDescriptorType =
    USB_DEVICE_QUALIFIER_DESCRIPTOR_TYPE;
pUSBDesc->usbDeviceQualDesc.bcdUSB= 0x0200;
pUSBDesc->usbDeviceQualDesc.bDeviceClass= 0;
pUSBDesc->usbDeviceQualDesc.bDeviceSubClass = 0;
pUSBDesc->usbDeviceQualDesc.bDeviceProtocol = 0;
pUSBDesc->usbDeviceQualDesc.bMaxPacketSize0 = 0x40;
pUSBDesc->usbDeviceQualDesc.bReserved= 0;
pUSBDesc->usbDeviceQualDesc.bNumConfigurations= 1;

```

Fill USB_CONFIGURATION_DESCRIPTOR for High-Speed connection as follows:

- EP1 OUT is used as a Bulk Out transfer endpoint.
- EP1 IN is used as a Bulk In transfer endpoint.

```

/* Filling Configuration Descriptor */
pUSBDesc->HSConfig.usbConfigDesc.bLength =
    sizeof(USB_CONFIGURATION_DESCRIPTOR);
pUSBDesc->HSConfig.usbConfigDesc.bDescriptorType =
    USB_CONFIGURATION_DESCRIPTOR_TYPE;
pUSBDesc->HSConfig.usbConfigDesc.wTotalLength=
    sizeof(USB_CONFIGURATION_DESCRIPTOR)
    + sizeof(USB_INTERFACE_DESCRIPTOR)
    + sizeof(USB_ENDPOINT_DESCRIPTOR) * 2;
pUSBDesc->HSConfig.usbConfigDesc.bNumInterfaces= 1;
pUSBDesc->HSConfig.usbConfigDesc.bConfigurationValue = 1;
pUSBDesc->HSConfig.usbConfigDesc.iConfiguration= 0;
pUSBDesc->HSConfig.usbConfigDesc.bmAttributes= 0xc0;
pUSBDesc->HSConfig.usbConfigDesc.MaxPower= 1;

/* Filling Interface Descriptor */
pUSBDesc->HSConfig.usbInterfaceDesc.bLength=
    sizeof(USB_INTERFACE_DESCRIPTOR);
pUSBDesc->HSConfig.usbInterfaceDesc.bDescriptorType=
    USB_INTERFACE_DESCRIPTOR_TYPE;

```



```
pUSBDesc->HSConfig.usbInterfaceDesc.bInterfaceNumber = 0x0;
pUSBDesc->HSConfig.usbInterfaceDesc.bAlternateSetting= 0x0;
pUSBDesc->HSConfig.usbInterfaceDesc.bNumEndpoints= 0x2;
pUSBDesc->HSConfig.usbInterfaceDesc.bInterfaceClass= 0xff;
pUSBDesc->HSConfig.usbInterfaceDesc.bInterfaceSubClass= 0xff;
pUSBDesc->HSConfig.usbInterfaceDesc.bInterfaceProtocol= 0xff;
pUSBDesc->HSConfig.usbInterfaceDesc.iInterface= 0x0;

/* Filling Endpoint Descriptor - 1 */
pUSBDesc->FSConfig.usbEndPointDesc1.bLength=
    sizeof(USB_ENDPOINT_DESCRIPTOR);
pUSBDesc->FSConfig.usbEndPointDesc1.bDescriptorType=
    USB_ENDPOINT_DESCRIPTOR_TYPE;
pUSBDesc->FSConfig.usbEndPointDesc1.bEndpointAddress = 0x01;
pUSBDesc->FSConfig.usbEndPointDesc1.bmAttributes = 0x2;
pUSBDesc->FSConfig.usbEndPointDesc1.wMaxPacketSize= 0x200;
pUSBDesc->FSConfig.usbEndPointDesc1.bInterval= 0x0;

/* Filling Endpoint Descriptor - 2 */
pUSBDesc->FSConfig.usbEndPointDesc2.bLength=
    sizeof(USB_ENDPOINT_DESCRIPTOR);
pUSBDesc->FSConfig.usbEndPointDesc2.bDescriptorType=
    USB_ENDPOINT_DESCRIPTOR_TYPE;
pUSBDesc->FSConfig.usbEndPointDesc2.bEndpointAddress = 0x81;
pUSBDesc->FSConfig.usbEndPointDesc2.bmAttributes = 0x2;
pUSBDesc->FSConfig.usbEndPointDesc2.wMaxPacketSize= 0x200;
pUSBDesc->FSConfig.usbEndPointDesc2.bInterval= 0x0;
Fill USB_CONFIGURATION_DESCRIPTOR for Full-Speed connection as follows:


- EP1 OUT is used as a Bulk Out transfer endpoint.
- EP1 IN is used as a Bulk In transfer endpoint.


/* Filling Configuration Descriptor */
pUSBDesc->HSConfig.usbConfigDesc.bLength =
    sizeof(USB_CONFIGURATION_DESCRIPTOR);
pUSBDesc->HSConfig.usbConfigDesc.bDescriptorType =
```



```
        USB_OTHER_SPEED_CONFIGURATION_DESCRIPTOR_TYPE;
pUSBDesc->HSConfig.usbConfigDesc.wTotalLength=
        sizeof(USB_CONFIGURATION_DESCRIPTOR)
        + sizeof(USB_INTERFACE_DESCRIPTOR)
        + sizeof(USB_ENDPOINT_DESCRIPTOR) * 2;
pUSBDesc->HSConfig.usbConfigDesc.bNumInterfaces= 1;
pUSBDesc->HSConfig.usbConfigDesc.bConfigurationValue = 1;
pUSBDesc->HSConfig.usbConfigDesc.iConfiguration= 0;
pUSBDesc->HSConfig.usbConfigDesc.bmAttributes= 0xc0;
pUSBDesc->HSConfig.usbConfigDesc.MaxPower= 1;

/* Filling Interface Descriptor */
pUSBDesc->HSConfig.usbInterfaceDesc.bLength=
        sizeof(USB_INTERFACE_DESCRIPTOR);
pUSBDesc->HSConfig.usbInterfaceDesc.bDescriptorType=
        USB_INTERFACE_DESCRIPTOR_TYPE;
pUSBDesc->HSConfig.usbInterfaceDesc.bInterfaceNumber = 0x0;
pUSBDesc->HSConfig.usbInterfaceDesc.bAlternateSetting= 0x0;
pUSBDesc->HSConfig.usbInterfaceDesc.bNumEndpoints= 0x2;
pUSBDesc->HSConfig.usbInterfaceDesc.bInterfaceClass= 0xff;
pUSBDesc->HSConfig.usbInterfaceDesc.bInterfaceSubClass= 0xff;
pUSBDesc->HSConfig.usbInterfaceDesc.bInterfaceProtocol= 0xff;
pUSBDesc->HSConfig.usbInterfaceDesc.iInterface= 0x0;

/* Filling Endpoint Descriptor - 1 */
pUSBDesc->FSConfig.usbEndPointDesc1.bLength=
        sizeof(USB_ENDPOINT_DESCRIPTOR);
pUSBDesc->FSConfig.usbEndPointDesc1.bDescriptorType=
        USB_ENDPOINT_DESCRIPTOR_TYPE;
pUSBDesc->FSConfig.usbEndPointDesc1.bEndpointAddress = 0x01;
pUSBDesc->FSConfig.usbEndPointDesc1.bmAttributes = 0x2;
pUSBDesc->FSConfig.usbEndPointDesc1.wMaxPacketSize= 0x40;
pUSBDesc->FSConfig.usbEndPointDesc1.bInterval= 0x0;
```



```
/* Filling Endpoint Descriptor - 2 */
pUSBDesc->FSConfig.usbEndPointDesc2.bLength=
    sizeof(USB_ENDPOINT_DESCRIPTOR);
pUSBDesc->FSConfig.usbEndPointDesc2.bDescriptorType=
    USB_ENDPOINT_DESCRIPTOR_TYPE;
pUSBDesc->FSConfig.usbEndPointDesc2.bEndpointAddress = 0x81;
pUSBDesc->FSConfig.usbEndPointDesc2.bmAttributes = 0x2;
pUSBDesc->FSConfig.usbEndPointDesc2.wMaxPacketSize= 0x40;
pUSBDesc->FSConfig.usbEndPointDesc2.bInterval= 0x0;
Fill USB_STRING_DESCRIPTOR_TYPE for High-Speed connection as follows:
/* Filling String Descriptor - 0 */
pUSBDesc->usbStringDesc1.bLength= 4;
pUSBDesc->usbStringDesc1.bDescriptorType=
    USB_STRING_DESCRIPTOR_TYPE;
pUSBDesc->usbStringDesc1.String[0] = 0x0409;

/* Filling String Descriptor - 1 */
pUSBDesc->usbStringDesc2.bLength= 26;
pUSBDesc->usbStringDesc2.bDescriptorType=
    USB_STRING_DESCRIPTOR_TYPE;
pUSBDesc->usbStringDesc2.String[0] = 'M';
pUSBDesc->usbStringDesc2.String[1] = 'a';
pUSBDesc->usbStringDesc2.String[2] = 'n';
pUSBDesc->usbStringDesc2.String[3] = 'u';
pUSBDesc->usbStringDesc2.String[4] = 'f';
pUSBDesc->usbStringDesc2.String[5] = 'a';
pUSBDesc->usbStringDesc2.String[6] = 'c';
pUSBDesc->usbStringDesc2.String[7] = 't';
pUSBDesc->usbStringDesc2.String[8] = 'u';
pUSBDesc->usbStringDesc2.String[9] = 'r';
pUSBDesc->usbStringDesc2.String[10] = 'e';
pUSBDesc->usbStringDesc2.String[11] = '.';

/* Filling String Descriptor - 2 */
```



```

pUSBDesc->usbStringDesc3.bLength= 18
pUSBDesc->usbStringDesc3.bDescriptorType=
        USB_STRING_DESCRIPTOR_TYPE;
pUSBDesc->usbStringDesc3.String[0] = 'P';
pUSBDesc->usbStringDesc3.String[1] = 'r';
pUSBDesc->usbStringDesc3.String[2] = 'o';
pUSBDesc->usbStringDesc3.String[3] = 'd';
pUSBDesc->usbStringDesc3.String[4] = 'u';
pUSBDesc->usbStringDesc3.String[5] = 'c';
pUSBDesc->usbStringDesc3.String[6] = 't';
pUSBDesc->usbStringDesc3.String[7] = '.';

```

Note: The bLength of each string descriptor structure should be same as the size of itself structure.

5.4 Getting Configured Number

IOCTL_USB_GET_CONFIG_NUMBER is used to get the a configured number from USB Client.

DeviceIoControl Win32 API is used for sending information to the USB Client driver.

```

DWORD nConfigNum;
DWORD dwSize;

DeviceIoControl(hDevice,
                IOCTL_USB_GET_CONFIG_NUMBER,
                &nConfigNum,
                sizeof(DWORD), /* This should be 4(bytes). */
                NULL,
                0,
                &dwSize,
                NULL);

```

When nConfigNum is not zero, it is available to open each endpoint, except EPO.

5.5 Opening the Endpoint

The endpoint of the USB client is opened using **CreateFile** Win32 API. To get the endpoint name, refer to 5.7.1.

```

PCHAR pEndpointPathName;

```



```
HANDLE hEndpoint;

/* Get a file path name. */

pEndpointPathName =
"\\?\pci#ven_8086&dev_8808&subsys_00000000&rev_01#5&1b6ba73f&0&1400b8#{66fde0f3-
69eb-427f-a91a-e3ce4c13b36a}\2";

/* Open the endpoint. */

hEndpoint = CreateFile(pEndpointPathName,
    GENERIC_WRITE | GENERIC_READ,
    FILE_SHARE_WRITE | FILE_SHARE_READ,
    NULL,
    OPEN_EXISTING,
    0,
    NULL);
```

5.5.1 Getting Endpoint Path Name

To open the endpoint for Bulk/Interrupt transfer, an endpoint path name is required. An endpoint path name is generated by adding an endpoint path number to the device path name. Refer to [Section 5.2](#) on how to identify device path name. [Table 2](#) shows endpoint path numbers to endpoint address.

Table 2. Endpoint Path Number

Endpoint Path Number	Endpoint Address	Endpoint Name	Endpoint Direction	Note
0	0x00	EP0	OUT	For only control transfer. These are not used for Bulk/Interrupt transfer.
1	0x80	EP0	IN	
2	0x01	EP1	OUT	-
3	0x81	EP1	IN	-
4	0x02	EP2	OUT	-
5	0x82	EP2	IN	-
6	0x03	EP3	OUT	-
7	0x83	EP3	IN	-

An endpoint address is defined as *bEndpointAddress* of an endpoint descriptor's member.

Example:

When the device was opened the following device path name was used.

```
DevicePathName =
"\\?\pci#ven_8086&dev_8808&subsys_00000000&rev_01#5&1b6ba73f&0&1400b
8#{66fde0f3-69eb-427f-a91a-e3ce4c13b36a}";
```



To open the **EP1 OUT**, The following endpoint path name is used.

```
EndpointPathNumber = 2;
EndpointPassName =
"\\?\pci#ven_8086&dev_8808&subsys_00000000&rev_01#5&1b6ba73f&0&1400b
8#{66fde0f3-69eb-427f-a91a-e3ce4c13b36a}\2";
```

To open the **EP1 IN**, The following endpoint path name is used.

```
EndpointPathNumber = 3;
EndpointPassName =
"\\?\pci#ven_8086&dev_8808&subsys_00000000&rev_01#5&1b6ba73f&0&1400b
8#{66fde0f3-69eb-427f-a91a-e3ce4c13b36a}\3";
```

5.6 Reading Data

ReadFile is used for read operations for Bulk/Interrupt Out transfer.

```
HANDLE hEndpoint;
ULONG nBytesRead;
PBYTE pBuffer;
DWORD dwResult;
ULONG nBytesResult;

/* Set maximum transfer size. */
nByteRead = 2048;

/* Allocate memory. */
pBuffer = (PBYTE)malloc(nByteRead);

/* Request a read transfer. */
dwResult = ReadFile(hEndpoint,
                    pBuffer,
                    nBytesRead,
                    &nBytesResult,
                    NULL);

/* Process the read data. */
```

Note: *nBytesRead* should be equal or larger than the data size that the host sends to the client, and should be a multiple of `MAX_PACKET_SIZE` of the target endpoint.

5.7 Writing Data

WriteFile is used for read operations for Bulk/Interrupt IN transfer.



```
HANDLE hEndpoint;

ULONG nBytesWrite;

PBYTE pBuffer;

DWORD dwResult;

ULONG nBytesResult;

/* Set transfer size. */
nBytesWrite = 2000;

/* Allocate memory. */
pBuffer = (PBYTE)malloc(nBytesWrite);

/* Prepare write data */

/* Request a write transfer. */
dwResult = WriteFile(hEndpoint,
                    pBuffer,
                    nBytesWrite,
                    &nBytesResult,
                    NULL);
```

5.8 Closing the Endpoint

Once all the operations related to the USB client driver are completed, the endpoint handle must be freed by calling the **CloseHandle** Win32 API.

```
CloseHandle(hEndpoint);
```

5.9 Closing the Device

Once all the operations related to the USB Client driver are completed, the device handle must be freed by calling the **CloseHandle** Win32 API.

```
CloseHandle(hDevice);
```