Developing
USB
PC *Peripherals*

Using the Intel 8x930Ax USB Microcontroller

Wooi Ming Tan

Annabooks

San Diego

Developing USB PC Peripherals

by

Wooi Ming Tan

Published by:

Annabooks
11838 Bernardo Plaza Court
San Diego, CA 92128-2414
USA
619-673-0870 800-462-1042 Fax 619-673-1432
info@annabooks.com, http://www.annabooks.com

Printed in the United States of America

ISBN 0-929392-38-8 First Printing July

1997

Information provided in this publication is derived from various sources, standards, and analyses. Any errors or omissions shall not imply any liability for direct or indirect consequences arising from the use of this information. The publisher, author, and reviewers make no warranty for the correctness or for the use of this information, and assume no liability for direct or indirect damages of any kind arising from technical interpretation or technical explanations in this book, for typographical or printing errors, or for any subsequent changes.

The publisher and author reserve the right to make changes in this publication without notice and without incurring any liability.

Disclaimer

The information or comments in this book reflects the author's own opinions; it does not represent the view of Intel Corporation, USB IF, Microsoft, or any other vendors.

The design example depicted in this book is not production-worthy, designers are required to modify the design to adhere to FCC and USB device class specifications. Readers assume all risks if they wish to follow the example; the author and publisher are not responsible or liable for any injuries or mishaps caused directly or indirectly by following the design example.

About the Author

Wool Ming is a Senior Technical Marketing Engineer for Intel USB Microcontrollers. He focuses on USB, microcontroller architecture, firmware development, and the utilization of microcontrollers in various USB applications. He has been involved in the USB world since the early days of USB development. He provides technical consultations to various major OEMs and has helped them to develop and demonstrate their USB products at several international tradeshows.

Wooi Ming has a Master of Engineering Degree in Electrical and Electronics Engineering from the National University of Singapore and a Bachelor Degree in Engineering from the University of Canterbury, New Zealand. His Master's research was on the behavior of the third generation digital cordless protocols in fading channels. He has programmed in C, C++, and various microcontroller assembly languages, using the IBM PC/DOS, Windows operating system, VAX/VMS, and UNIX environments. He has experience with data networks and VAX/VMS systems planning and management, and in the manufacturing of wireless communications systems.

Dedication

This book is dedicated to
my parents,
my wife Pat, and
my sisters Nee and Ling.

Acknowledgements

The author would like to thank
all the members of the USB team
from CEG (Chandler and Penang), 1AL and
the USE IF for
their support.

Contents

Table of Figures

Foreword

Universal Serial Bus (USB) is one of the most important developments in PC peripheral interconnect technology since the introduction of serial and parallel ports in the early 1980's. It is fast becoming the port of choice for many new digital video conferencing cameras, scanners, monitors, PC telephony equipment, and Human Interface Devices such as keyboards, games, and pointing devices. The benefits of USB, such as ease of use, true plug and play, high performance, and reduced overall system cost, are just a few of the reasons this technology has gone from specification to product deployment in less than two years.

Wooi Ming Tan has provided in this book a very timely introduction to the concepts of USB as well as a very practical guide on how to design USB peripheral products. With his early involvement in USB at Intel Corporation, he has faced many of the design issues that engineers will likely encounter as they begin product development. The book also provides a useful overview for those just interested in gaining a more thorough understanding of USB, such as project management, sales, and marketing personnel involved in serial bus interconnect.

I am very pleased to see this book providing an easy step-by-step methodology to allow more USB products to quickly come to market while following a thorough design practice. Wooi Ming Tan's experience in USB and microcontroller technology certainly comes across in the book, allowing the reader to save many hours in USB peripheral design.

Stephen Whalley
Chairperson, USB Implementers Forum

1. What Will This Book Do For You?

"Everybody talks about USB, what is it?"

"Can USB benefit my current products?"

"How hard is it to understand USB?"

"How do we go about designing a USB device?" "Where do I start?"

"There are USB hosts, WDM, USB devices, firmware..., where can I get all the information?" "I need all of this info in one place!"

"Do I need to develop a device driver for my USB device?"

"Will USB get rid of the PC add-in cards?" "Wow, if I don't need the add-in cards for my devices, I can reduce the cost by..."

"Can our current RS-232 device use the benefits of USB?"

"How many engineers or computer scientists do I need to design a USB device?"

"Where can I find a summary of USB protocol?" "The spec is over 250 pages!"

"Do I need to attend a class to understand USB ? Device Driver Class?"

"Where do I get more info to develop my USB device?"

This book is written with the aim of answering these questions. It is intended for USB device project managers, USB design engineers, USB device marketing staff, or USB sales personnel. It is also meant for those who are interested in USB; who would like to know more about USB and how to start developing USB devices. A step by step method is used to guide you in developing USB devices. The book is meant not only for beginners, but also for those who have some USB experience. They can

use this book to develop a more complete view of the steps required to design a USB device.

The book shows you the steps required to start developing USB devices; it discusses USB protocol, a USB microcontroller and its firmware operating model, USB device drivers based on the Windows Driver Model (WDM), Windows Applications, and the development tools needed to develop USB devices. The book also includes a design example and its source codes. By studying the code examples provided on the enclosed diskette, designers can get a good insight into the requirements of developing the firmware codes, drivers, and application software. This book and the code examples can easily cut weeks of development time off your USB device design effort.

1.1 What is USB, Anyway ?

Lots of PC users have had bad experiences installing PC peripherals. We connect external devices to our PC, but before long, we run out of serial or parallel ports. Once we start to open the PC box to install expansion cards, add-in cards, etc., things become even more complicated. We face a complex and astonishing number of dip switches, jumper cables, software drivers, IRQ settings, and I/O addresses that must be configured. We are typically required to go through hundreds of pages of the installation guides just to install our peripherals successfully. Sometimes, we even need to call the dreaded "Please hold for the next available representative" technical hot-lines to get things straightened out.

The era has finally arrived to forget about all the troubles of installing new PC devices. This is the era of the Universal Serial Bus (USB). The USB Specification Rev 1.0[1] was finally released to the world on January 15, 1996; royalty free! The spec is a joint collaboration among seven leading computer, telecommunications, and software companies: Compaq, Digital Equipment Corporation, IBM PC Company, Intel, Microsoft, NEC, and Northern Telecom. Many other companies now see the potential of USB and have jumped on the bandwagon. To date, the USB-Implementers Forum[2] has over 300 members. PCs with USB connectors have started appearing in the market and many USB devices are currently available with more becoming available soon.

USB brings true Plug and Play convenience to PC users for the first time in PC history. It makes the process of adding a new peripheral as

easy as plugging a phone line into the connector on the wall. We do not have to worry about the number of available ports, the dip switches, the jumpers, the IRQs, the installation guides, etc. All that is required to install a new peripheral is to connect the peripheral to a USB socket. The PC will recognize the new device and activate the appropriate applications. We can even install the device when the PC is up and running; USB is truly Plug and Play! Not only that, USB also creates an opportunity for exciting new applications, like digital cameras, multimedia devices, telephony devices, and multi-user games. It opens up huge opportunities for companies to profit from these exciting PC devices.

This is what this book is all about; guiding you, step by step, in developing a USB device. The book is meant for anyone that wants to know more about USB and USB devices. It also includes code examples on the enclosed diskette to give readers a thorough understanding of the steps involved in developing a USB device. The disk includes sample code for USB firmware, the WDM driver and application software as a reference for anyone who wishes to develop USB devices. Use of this code is described in Chapter 8.

1.2 Outline of the Book

There are nine chapters in this book. Chapters 2 through 7 consist of various steps required to develop a USB device, as shown in Figure 1.1. Each of the chapters contain only relevant material to get you started in executing the steps. References are quoted appropriately throughout the book to direct you to extra or more detailed information.

With this organization of the book, readers who know USB, for instance, can skip Chapter 2 and go straight to Chapter 3 for the "Development Environment". I'll show you how some of the development tasks can be done in parallel thereby reducing the elapsed time to develop a USB device and shortening the time to market for the device. The steps will be discussed in more detail in later chapters.

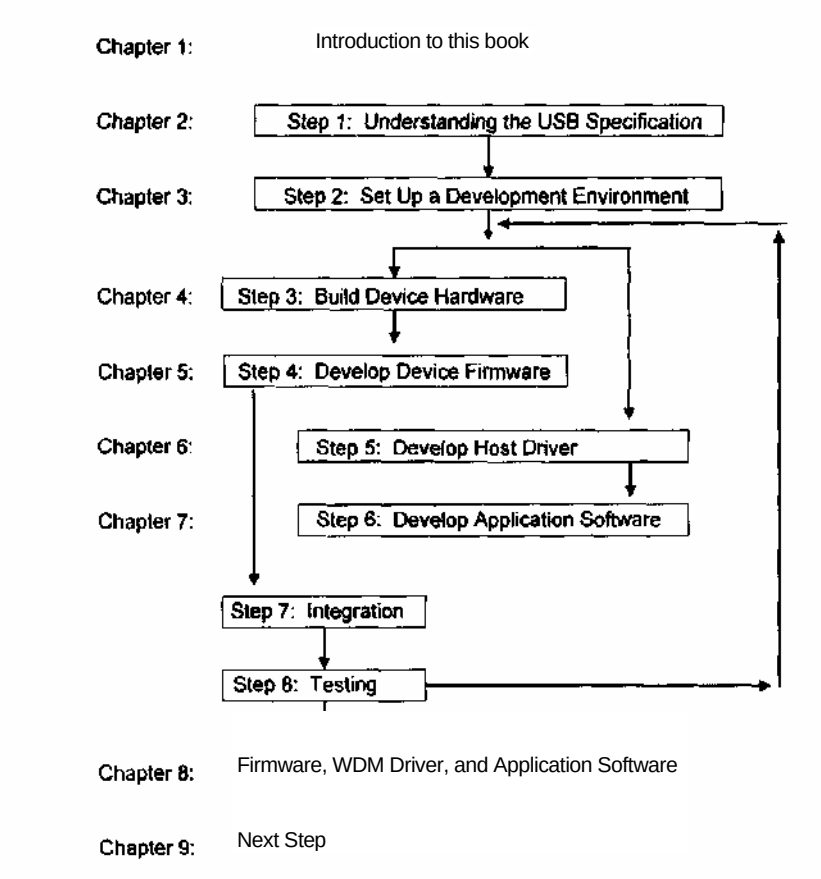


Figure 1-1. Chapter Layout

1.3 References

These references will appear several times throughout the book. They will be designated by their number in subscript next to the referral For example, you'll see something like this [1].

[1] Compaq, Digital Equipment Corporation, IBM PC Company, Intel, Microsoft, NEC, and Northern Telecom, "USB *Specification* 1.0", revision 1.0, Jan 19,1996. http://www.usb.org

[2] To join the USB Implementers Forum, contact:
USB IMPLEMENTERS FORUM
JF2-51, 2111 NE 25th Avenue
Hillsboro, OR 97124 For more
information, see
http://www.usb.org

[3] *"8x930Ax, 8x930Hx USB Microcontroller User's Manual",* order no: 272949-001, September 1996. It can be ordered from: Intel Corporation Literature Sales P.O. Box 7641
Mt. Prospect, IL 60056-7641 Or call
1-800-879-4683 or
http://www.intel.com/design/usb/manuals

[4] *"8x93QAx USB Microcontroller Datasheet",* order no: 272917-003, February 1997, and the *"8x930Ax Specification Update",* order no: 272940 available at http://www.intel.com/design/usb.

[5] Helen Custer, *"Inside Windows* NT", Microsoft Press, a Division of Microsoft Corporation, 1993.

[6) Dave Hamilton and Mickey Williams, *"Programming Windows NT 4",* Sams Publishing, 1996.

[7] Ori Gurewich and Nathan Gurewich, *"Teach Yourself Visual C++ 4 in 21 Days",* Sams Publishing, 1996.

1.4 Feedback to the Author

It was my top priority to ensure that all the information provided is as correct as possible, and as simple as possible to understanding. However, if you find any errors or omissions that are critical to this book, please send your feedback to the author. Thank you — and get into the BUS.

E-Mail address: wtan@annabooks.com

2. Step 1: Understand the USB Specification

The first step to design a USB device is to understand the USB Specification. The USB Specification 1.0 has 267 pages; however, the good news is that we do not have to go through all the pages before starting to design a USB device. This chapter gives sufficient knowledge of the USB Specification, made simple to comprehend, for us to start designing a USB device. For more details, refer to the USB Specification 1.0[1]

2.1 Summary of USB Protocol

USB is a 12 Mbps serial channel that is shared by a wide variety of PC peripherals, up to 127 peripherals total can be mounted in a single PC. It has four wires: two wires for supplying the power, and two wires for data transfer. It is a token-based bus protocol, where the USB host is the master of the bus. The host broadcasts tokens on the bus and a device that detects a match in its address, as described in the token, responds to the host. The 12 Mbps bandwidth of the bus is framed into 1 ms slots and shared by all the devices in the bus in a time division multiplexed (TDM) manner. The host has knowledge of all the bus access requirements of its devices, and schedules all the bus activities.

2.2 USB Objectives

USB was developed to achieve three main goals:

• Ease of use

It achieves this goal by providing a hot, out-of-box, Plug and Play environment. Users do not have to open the PC case, configure interrupts, set dip switches, and so forth, just to install a new PC peripheral. All you have to do is attach the USB device to a USB connector. The USB PC will recognize the newly attached device instantly and exchange information with the device about its

manufacturer, its type, its version number, and so forth. If a USB device is detached from the PC, the PC will recognize this and notify the application software of the device. The application may then notify users that the device has been removed.

• **PC-phone connection**

USB provides the ability to support real time data (e.g., voice, audio, and compressed video), and asynchronous messaging. For example, before USB, it was extremely difficult to provide a voice path together with control data into or out of a PC using serial or parallel ports. With USB, the voice and control data are divided into different pipes that go into or out of a PC. This is achieved using the "endpoint" notation (more of this in later sections). Moreover, USB can bring multiple voice paths into and out of the PC, which cannot be easily done by serial or parallel ports.

• **Port expansion**

USB provides a low-cost PC port extension and allows up to 127 device attachments for a single PC host. The low-cost objective is one of the reasons that the USB rate is at 12 Mbps, as higher rates usually mean more expensive shielding, chip sets, components, etc. There are, in fact, two categories of USB devices that can be connected to the same USB host; one consists of full speed devices at 12 Mbps and the other type consists of low speed devices at 1.5 Mbps. Examples of low speed devices are mice and keyboards, which do not require the full 12 Mbps speed and are very cost sensitive.

With the 12 Mbps (and 1.5 Mbps low speed) bus in mind, USB is not targeted for very high speed applications like disk drives or full motion video applications. It is targeted for low and medium speed applications, like keyboard, mice, modems, CTI, low-speed scanners, low-speed printers, and low-speed cameras. As a rule of thumb, a device that requires more than 4 Mbps of bandwidth constantly is not suitable as a USB device.

2.3 USB Bus Topology

The USB Specification defines the interconnections and communications between two main elements of a USB system: USB host PC and USB devices. Its physical interconnection topology is a tiered star, as shown in Figure 2-1.

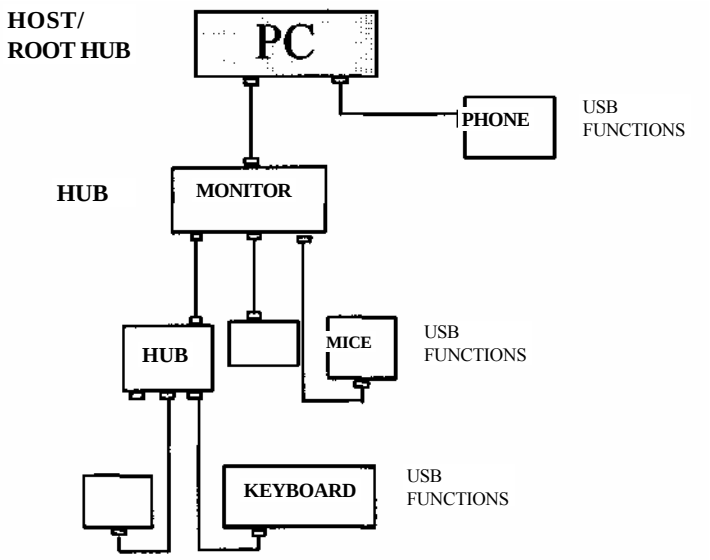


Figure 2-1. USB bus topology

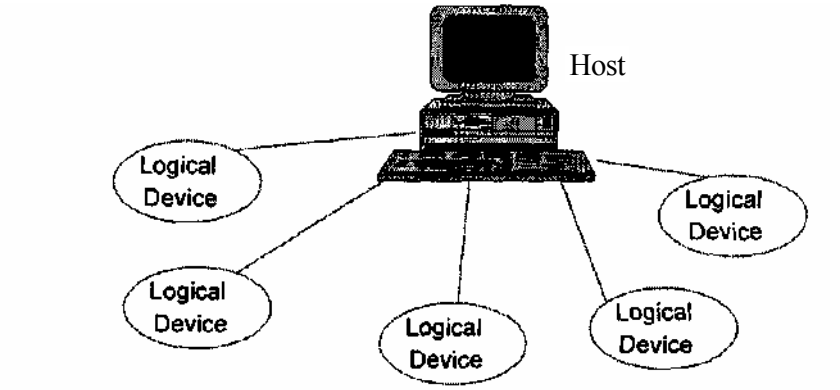
There is only ONE host in a USB system. The USB Host is a PC with a USB host controller and it may be implemented in a combination of hardware, firmware, or software. In this book, we focus on the USB devices. The details of the USB host can be found in Chapter 10 of the USB Specification[1]. The host is the master of the USB system. It controls and schedules all the activities in its system. A root hub is integrated within the host to provide one or more attachment points.

There are two types of USB devices: USB Hub and USB functions. The USB hubs provide extra attachment points for the system. We can think of them as the socket extensions used to connect additional electrical appliances. The USB functions provide capabilities to the system, e.g., keyboard, joystick, etc. The USB functions are the focus of this book and the terms "function" and "device" are used interchangeably in this book.

2.4 USB Data Flow Model

USB provides a protocol for communications between the devices and the host. Although the bus topology of a USB system is a tiered star (Figure 2-1), the connection of the USB devices to the USB host can be thought of as one-to-one, as shown in Figure 2-2. This is referred to as

the logical connections of the USB system. The data flow model is based



on these logical connections.

Figure 2-2. Logical connections from host to USB devices

2.5 USB Pipe Concept (Device Address and Endpoint)

USB communication can be viewed using a pipe concept, as shown in Figure 2-3.

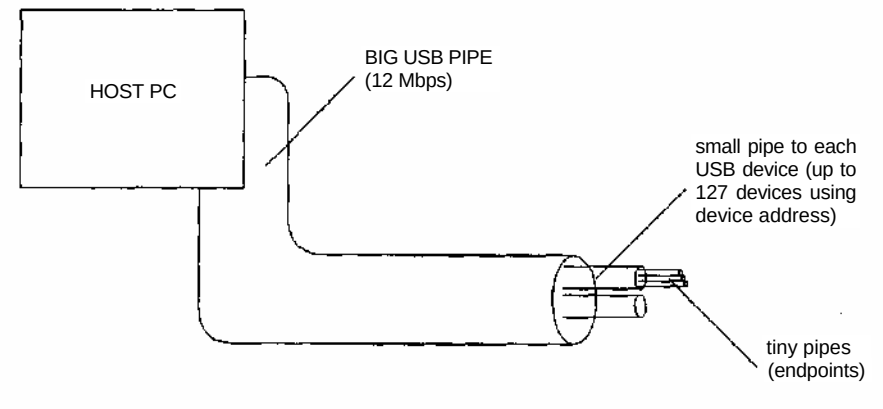


Figure 2-3. **Concept of the USB pipes and endpoints**

It consists of a big pipe (12 Mbps) and up to 127 small pipes. Each of the small pipes connects to a USB device. There are 7 address bits in the USB token (more details in section USB Packet Formats) which can

address up to 128 devices. However, address 000 OOOOB, called the USB default address, is used for all devices when they are first attached. Thus USB can support up to 127 devices. Each of the small pipes connected to a USB device can be further divided into smaller pipes, which we refer to as tiny pipes, as shown in Figure 2-3.

There can be up to 16 pairs of these tiny pipes from a single small pipe. This is because there are 4 "endpoint" bits in a USB token and the token can be identified as IN or OUT tokens. After a device receives an IN token, it will transmit data to the host, and if it receives an OUT token, it will anticipate data from the host. The "endpoints" are associated with tiny pipes here to illustrate the USB pipes concept clearly to readers. These endpoints (or tiny pipes) are the most important concept designers need to understand. In a multimedia USB device, for example, one of the endpoints may carry data, one may carry voice and the other may carry control information. All these packets are required to be treated differently, for instance, data for file transfers (e.g. files to be sent to printers) requires high accuracy in delivery, often less than 1×10^{-10} in bit error rate. On the other hand, raw voice data can tolerate up to 1×10^{-3} in bit error rate but cannot tolerate excessive delay (typically delay of > 20 ms is considered bad). However, USB is designed to carry all of this different information and it can all be communicated with through the USB connection, with the endpoint concept used to separate according to data type, sources, etc.

2.6 USB Transfer Types

There are four USB transfer types; i.e. (1) control, (2) isochronous, (3) interrupt, and (4) bulk.

(1) Control Transfer

Control transfers are bi-directional and intended to support configuration, command or status communications between the host and functions. A control transfer consists of 2 or 3 stages: a setup stage, data stage (may or may not exist) and status stage. In the setup stage, the host sends a request to the function. In the data stage, data transfer occurs in the direction as indicated in the setup stage, and in the status stage, the function returns a handshake to the host.

Each USB device is required to implement the endpoint pair 0 as control transfer endpoints. It is used to exchange information (see enumeration section under USB Functions) when the device is first

attached to the host. USB ensures that the control transfer is delivered without error. This is done using CRC error checking (see section USB Packet Formats) and re-transmission if the error cannot be recovered.

(2) Isochronous Transfer

Isochronous transfer can be uni-directional or bi-directional. It is intended to be used to transfer information that requires a constant rate and can tolerate errors. A good application example of this transfer type is 64 kbps Pulse Code Modulation (PCM) voice data. The voice data requires a constant rate and can tolerate errors (typically up to a bit error rate of $1x10^{-3}$).

Isochronous transfer has the following characteristics:

— guaranteed access to USB with bounded latency, and

— no retrying in case of a delivery failure due to error.

The maximum packet size for isochronous transfer is 1023 bytes per USB frame of 1 ms. This implies that the maximum transfer achievable using isochronous transfer is 8,184 Mbps.

(3) Interrupt

Interrupt transfer is uni-directional and only inputs to the host. It is used to support data transfers that are small in size and happen infrequently. The interrupt transfer of USB is of a polling type; that is, the host asks if the interrupt endpoints have any data to send according to the frequency requested by the endpoints. For full speed devices, the endpoint can specify the polling period of 1 ms to 255 ms. For low speed devices, the polling period is from 10 ms to 255 ms. Thus, the fastest polling frequency is 1 kHz for full speed devices. In case of delivery failure due to errors, a retry of transfer will be carried out in the next polling period. A good example of a device using interrupt transfer is a keyboard, where it sends a small amount of data to the host when a key is pressed.

(4) Bulk Transfer

The bulk transfer can be uni- or bi-directional. It is used to support endpoints that need to communicate large amounts of data accurately but where the time of delivery is not critical. The bulk transfer is designed to claim unused bandwidth of the USB, and therefore the provision of USB access time to bulk transfers is on a bandwidth available basis. In case of delivery failure due to errors, the transfer is retried. A typical usage is a scanner, where a large amount of data needs to be delivered accurately, but not immediately.

2.7 USB Mechanical and Electrical

There are two types of USB connectors: series A and series B. Series A connectors are shown in Figure 2-4.

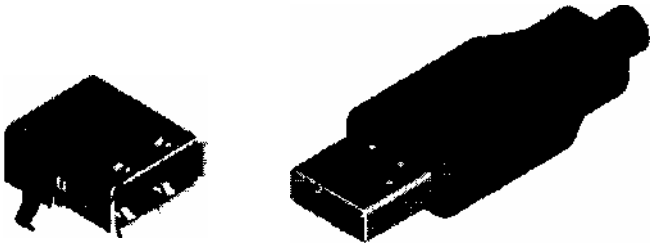


Figure 2-4. USB connectors (series A)

Series A connectors are used to connect to downstream devices. For example, there are series A connectors on the back of a USB PC where the root hub is located. Series B connectors are used to connect to upstream hubs or devices, as depicted in Figure 2-5.

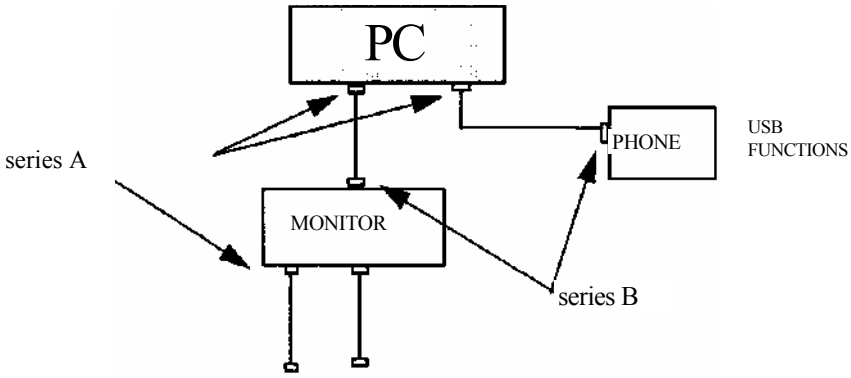


Figure 2-5. Usage of series A and series B connectors

The standard USB cable consists of one pair of 20-28 AWG wires for power distribution with another 28 AWG pair twisted, with shield and overall jacket. The cross section of the cable is shown in Figure 2-6. This cable is used for devices operating at 12 Mbps (Full Speed).

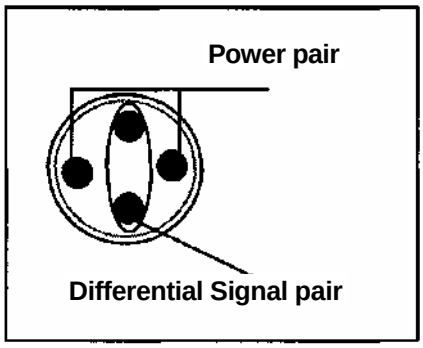


Figure 2-6. Cross section of a full speed USB cable

For low speed devices operating at 1.5 Mbps, an alternative cable of identical gauge but without the twisted conductors and shield can be used.

As shown in Figure 2-6, there are four USB wires; two for power supply (V_{bus} and GND) and two for signaling (D+ and D-). For a full speed device, a 1.5 kΩ ± 5% pull up resistor is required on the D+ line, as shown in Figure 2-7.

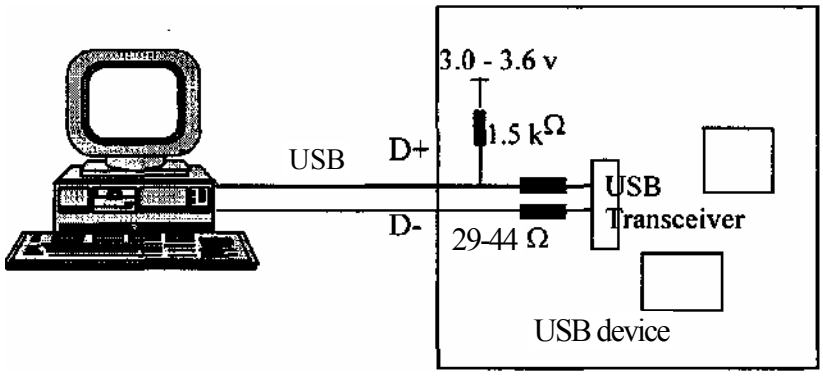


Figure 2-7. Connection of external resistors in a full speed device

Series resistors of (typically 29 - 44 Ω) are also required to connect to the USB wires, as shown in Figure 2-7, for impedance matching. For low speed devices, the 1.5 kΩ ± 5% resistor must be tied from the D- wire to a voltage source between 3.0 and 3.6 v.

2.7.1 Power Supply from USB Wires

USB provides limited power supply to its devices. The voltage supply is between 4.75v-5.25v (V_{bus}) from the root hub or a self powered hub port and the 0v ground (GND). A bus-powered device can draw a maximum of 500 mA if attached to the root hub or a self-powered hub. The definition of a bus-powered device is that the device draw current from the USB bus directly. Devices that have their own power supply are called self-powered devices. Before the requested amount of current is granted by the USB host, the device must draw no more than 100 mA from the USB when it first attaches to the bus. For devices that are attached to a bus-powered hub, the maximum amount of current supply permitted is 100 mA (section 7.2.1 of USB spec 1.0).

2.8 USB Packet Formats

USB packets fit into one of these types: token, SOF, data, handshake, and special packets.

2.8.1 Token Packets

USB transaction are always initiated by the host and are started with a token packet in the format shown in Figure 2-8.

Figure 2-8. A USB token packet

2.8.1.1 SYNC

All packets begin with a synchronization (SYNC) field. It is used by the input circuitry to align incoming data with the local clock and it is 8 bits in length.

2.8.1.2 PID

A packet identifier (PID) immediately follows the SYNC field. A PID consists of four bits followed by a four-bit check field, as shown in Figure 2-9.

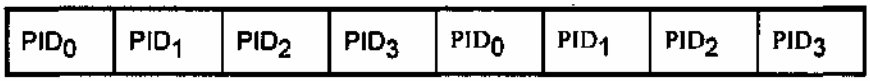


Figure 2-9. PTD packet format

The PID indicates the type of packet that is transmitted (differentiated by PID_0 and PID_1); either token, data, handshake or special packet.

These four types of packets are subdivided into various identifiers (differentiated by PID_2 and PID_3):

Packet Type	PID Name	PID_3; PID_2	PID_1; PID_0
Token	OUT	00	01
Token	IN	10	01
SOF	SOF	01	01
Token	SETUP	11	01
Data	DATA0	00	11
Date	DATA1	10	11
Handshake	ACK	00	10
Handshake	NAK	10	10
Handshake	STALL	11	10
Special	PRE	11	00

Note that bits in USB are sent out least significant bit (LSB) first, followed by next LSB and completed with the most significant bit (MSB) last. For all the diagrams in this chapter, packets are displayed in a left to right reading mode showing the order they would be moving across the bus.

2.8.1.3 ADDR

The 7-bit function address (ADDR) field specifies which device the packet is intended for. The host assigns a unique address to each device during USB enumeration process when it is first attached to the bus.

2.8.1.4 ENDP

The 4-bit endpoint (ENDP) further specifies which tiny pipe of the device the packet is intended for. For example, a device has endpoint 0 and endpoint 1 configured, the host can choose to communicate to endpoint 0 of this device at this instant, ignoring endpoint 1.

2.8.1.5 CRC

Cyclic Redundancy Checks (CRCs) provide error checking for the non-PID fields of the USB packet. The FID is not protected by this CRC as it has self error checking, as shown in Figure 2-9 above.

2.8.2 Start of Frame Packets

Start of Frame (SOF) packets, as shown in Figure 2-10, are broadcast by the host once every 1.00 ms ± 0.05.

Figure 2-10. A USB SOF packet

It consists of a PID indicating the SOF packet followed by an 11-bit frame number.

2.8.3 Data Packets

In USB, the host performs the management of the bus, and devices respond to requests from the host. If the host requests information from a device it will send an IN token addressed to that device endpoint and the device will response with a USB data packet. The format of the data packet is as shown in Figure 2-11, with the PID indicating DATA0 or DATA1.

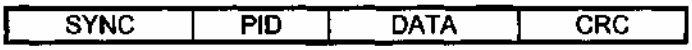


Figure 2-11. A USB data packet

If the transfer type of this particular endpoint is bulk, control or interrupt, the host will response with an acknowledgment when it receives the data without error. If the transfer type is isochronous, no handshake packet will be sent. The host will send an OUT token followed by a data packet if it wishes to transfer data to the device.

2.8.4 Handshake Packets

The handshake packets consist of only a PID, as shown in Figure 2-12. They are used to report ACK, NACK or STALL (as indicated by the PID) for transactions that require handshakes.

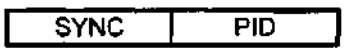


Figure 2-12. A USB handshake packet

2.8.5 Special Packets

The special preamble (PRE) packet is sent by the host when it wishes to communicate to low speed devices. The host will send this PRE packet first, before sending low speed 1.5 Mbps packets to communicate with low speed devices.

2.9 USB Protocol by Transfer Types

This section describes in detail USB protocol for various transfer types. The control transfer is different from the other transfer types as it is completed over a few USB frames. The other transfer types are completed within a frame.

2.9.1 Control Transfers

Control transfers have two or three stages; setup, data (may or may not be required) and status stage. The best way to understand how a control transfer works is to look at an example, as shown in Figure 2-13. It shows a complete control transfer between a function and a host, as captured by a CATC USB bus analyzer (see Chapter 3 for a more detailed discussion of a bus analyzer).

Packet #	
178	Sync(_0000001) SOF(0xA5) Frame #(0x12C) CRC5(0x1F)

start of setup stage:

179	Sync(_0000001) SOF(0xA5) Frame #(0x12D) CRC5(0x00)
180	Sync(00000001) SETUP(0xB4) ADDR(0x08) ENDP(0x0) CRC5(0x06)
181	Sync(00000001) DATA0(0xC3) DATA(80 06 00 01 00 00 80 00) CRC16(0xB129)
182	Sync(00000001) ACK(0x4B)

start of data stage:

183	Sync(00000001) SOF(0xA5) Frame #(0x12E) CRC5(0x02)
184	Sync(00000001) IN(0x96) ADDR(0x08) ENDP(0x0) CRC5(0x06)
185	Sync(00000001) DATA1(0xD2) DATA(12 01 00 01 00 00 00 08) CRC16(0xC8E7)
186	Sync(00000001) ACK(0x4B)

187	Sync(_0000001) SOF(0xA5) Frame #(0x12F) CRC5(0x1D)
188	Sync(00000001) SOF(0xA5) Frame #(0x130) CRC5(0x0A)
189	Sync(00000001) IN(0x96) ADDR(0x08) ENDP(0x0) CRC5(0x06)
190	Sync(00000001) DATA0(0xC3) DATA(86 80 AD 0D 01 00 00 00) CRC16(0xC16F)
191	Sync(00000001) ACK(0x4B)

192	Sync(_0000001) SOF(0xA5) Frame #(0x131) CRC5(0x15)

193	Sync(_0000001) SOF(0xA5) Frame #(0x132) CRC5(0x17)
194	Sync(00000001) IN(0x96) ADDR(0x08) ENDP(0x0) CRC5(0x06)
195	Sync(00000001) DATA1(0xD2) DATA(00 01) CRC16(0xFCF1)
196	Sync(00000001) ACK(0x4B)
197	Sync(00000001) SOF(0xA5) Frame #(0x133) CRC5(0x08)

start of status stage:

198	Sync(00000001) SOF(0xA5) Frame #(0x134) CRC5(0x16)
199	Sync(00000001) OUT(0x87) ADDR(0x08) ENDP(0x0) CRC5(0x06)
200	Sync(00000001) DATA1(0xD2) DATA() CRC16(0x0000)
201	Sync(00000001) ACK(0x4B)
202	Sync(00000001) SOF(0xA5) Frame #(0x135) CRC5(0x09)
203	Sync(00000001) SOF(0xA5) Frame #(0x136) CRC5(0x0B)

Figure 2-13. An example of control transfer

In the beginning there is no activity in the bus, and the host sends out the 1 ms interval SOF packets as shown in packet #178 of Figure 2-13. Note that the time interval between any two SOF packets is 1 ms, for example there is an 1 ms period between packet #178 and packet #179. As shown above, each USB frame of 1 ms (between SOFs) is boxed together to display the single frames.

In the 1 ms period starting by the packet #179, the host sends out a SETUP token (packet #180) and 8 bytes of setup data (packet #181) to a device. These eight bytes of data specify what type of request the host wants from the device. The first byte specifies the characteristic of the request. For this case, it is 80H which means that the direction of the subsequent packets will be from device to host, the request type is a standard request and the recipient of this packet is the device. Section 9.3 of the USB Specification[1] enables us to decode what is meant by the setup data. The second byte is 06H which can be decoded as a "GET DESCRIPTOR" request, and so on. The last word (2-byte) is 0080H which is the expected length of the packets that the device is required to send in the data stage (data stage exists if this word is not OOOOH). This is the setup stage for a control transfer. The device, after the correct reception of this setup data, issues an ACK handshake packet (packet #182) immediately. Note that all of these transfers take place within 1 ms, between SOF packet #179 and SOF packet #183.

Since the host requests to get data from the device, it will send an IN token after the setup packet, as shown in the USB frame starting with

packet #183. The device responds with the data the host requested. This is the beginning of the data stage of the control transfer. Note that the control transfer is done over a few USB frames as signified by the multiple SOF tokens between the packets. The data stage carries on until the device finishes the required data. The length of the device's descriptor is stated in the first byte of the data; i.e., for our example, it is 12H as in packet #185. To decode the device's descriptor, please refer to section 9.6.1 of the USB Specification[1] Note that this first byte of the device's descriptor (12H) overrides the previous expected length of the packets (0080H) by the host, as indicated in packet #181.

The status stage of the device is shown in the USB frame starting with packet #198. The host in this case sends out an OUT token. The token for the status stage is always opposite to that of the data stage. In our example, we have IN tokens in the data stage and thus an OUT token and a null packet will be sent in the status stage. Note that there are some control transfers that have no data stage, for example the "SET CONFIGURATION" setup packets. For this case, the host will issue an IN token for the status stage.

2.9.2 Isochronous Transfer

Isochronous transfers have token and data phases. It is completed within a USB frame. If the host sends an IN token, the device will transmit data to the host. If the device receives an OUT token from the host, it will anticipate data from the host immediately after the OUT token. There is no handshake phase for isochronous transfers.

2.9.3 Bulk Transfer

The bulk transfer is similar to the isochronous transfer except that it has a handshake phase after the data phase. This is to ensure that the data is sent or received accurately. An ACK handshake packet will be sent if the data is received without error, by the host or by the device. The device can also return a NAK or STALL handshake. A NAK handshake indicates that the device is temporarily unable to perform the request by the host. A STALL handshake indicates that there is an error condition on the device and it requires host software intervention. Note that the NAK and STALL handshake packets are only sent out by the device.

2.9.4 Interrupt Transfer

Interrupt transfer is similar to the bulk transfer, except it consists solely of IN tokens in its token phase. Upon receiving the IN token, the device may return data, or a NAK or STALL packet. If the device has no new interrupt data to return, it returns a NAK handshake in the data phase. If the device is stalled and requires host software intervention, it returns a STALL handshake. If an interrupt is pending, the function returns the interrupt information as a data packet. Upon receiving the data packet, the host returns an ACK handshake if the data was received error free or it returns no handshake if the data packet was corrupted. As mentioned, only devices are allowed to return NACK packets as a handshake; the USB host returns an ACK packet or returns no packet at all.

2.10 USB Functions

All USB devices (hub or hubless) are required to have a default address of 000 OOOOB upon powering up. They are also required to have their endpoint 0 configured as the control endpoint pair (transmit and receive pair).

2.10.1 USB Enumeration Steps

When a USB device is attached to the bus, the host uses a process known as bus enumeration to identify and manage the newly attached device. This process enables the "Hot Plug and Play" feature of USB. The device goes through the following stages, as shown in Figure 2-14.

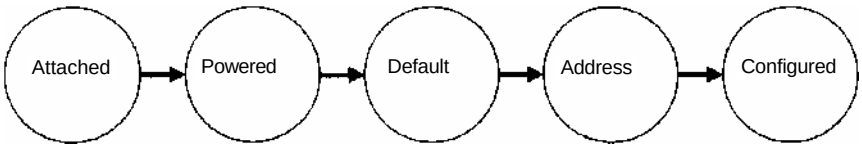


Figure 2-14. USB enumeration state diagram

When a USB device is attached, it undergoes the following actions:

1. The hub (or root hub) where the device is attached informs the host that an event has occurred. There are two 15K pull down resistors
connected to the D+ and D- USB wires in the hub port or root hub in the host. When the device is attached, its pull-up resistor (see USB
Mechanical and Electrical section) will cause the signal level of either D+ or D- to rise, thereby signaling the attachment of a USB device. The device is now considered to be in the attached stage (see Figure 2-14).
2. The host inquires the hub regarding the nature of the event.
3. The host issues a "port enable and reset" command to the hub where the device is attached.
4. The hub will then issue a USB reset signal to that port for 10 ms and provide 100 mA of power supply to the device after the reset signal is completed. The device now is considered to be in the powered stage, as shown in Figure 2-14. From the device point of view, the reset signal is the first signal it sees when it is attached to the hub or root hub. After the reset signal, the device is in its default stage, where it corresponds to the host with its default address (address 0 and endpoint 0).
5. The host will then initiate a GET DESCRIPTOR ("device" descriptor type) setup packet to the device, and the device needs to respond accordingly.

   After the first GET DESCRIPTOR (device) packet, an IN token will be issued and the device is required to reply accordingly. Among the data sent to the host in the device descriptor packet is the actual maximum data payload size the device can handle. The host will use this information for subsequent communication to this device. After the successful reception of the descriptor packet, the host will issue a SET ADDRESS command.
6. Using this SET ADDRESS command, the host assigns a unique address to the device. This moves the device to the address stage.
7. The host then issues a GET DESCRIPTOR (device) again as the previous GET DESCRIPTOR command is terminated pre-maturely.
8. After successful completion of the previous step, the host issues a GET DESCRIPTOR ("configuration" descriptor type) setup packet to acquire the configuration of the device.
9. Based on the configuration information received, the host assigns a configuration value to the device, via SET CONFIGURATION setup packet. The device (if bus powered) may now draw the amount of V_{bus} power described in its configuration. The device is now in the
configured stage (refer to Figure 2-14) and is ready for use.

2.10.2 A Snap-Shot of a Typical USB Bus Enumeration

A typical communication flow during a USB enumeration is captured by a CATC bus analyzer tool (see Chapter 3 for details) and is shown to give a detailed insight into the USB bus enumeration process.

A USB bus reset is issued when the device first plugs into a USB hub or the root hub. This is the followed by the GET DESCRIPTOR (device) setup packet as shown in packets #29 and #30. The 2nd byte value of 06H of the data (packet #30) indicates that this is a GET DESCRIPTOR command, as stated in Section 9.3 of the USB Specification[1].

Packet #	
27	Sync(00000001) SOF(0xA5) Frame #(0x48E) CRC5(0x0F)
28	Sync(00000001) SOF(0xA5) Frame #(0x48F) CRC5(0x10)
29	Sync(00000001) SETUP(0xB4) ADDR(0x00) ENDP(0x0) CRC5(0x08)
30	Sync(00000001) DATA0(0xC3) DATA(80 06 00 01 00 00 80 00) CRC16(0xB129)
31	Sync(_0000001) ACK(0x4B)

The device we plugged in acknowledges the reception of this GET DESCRIPTOR (device) packet and issues an ACK (packet #31) handshake. The device then replies to the following IN token with an 8-byte data packet, as shown in packet #34. Note that the last byte of this packet is 08H which indicates the size of endpoint 0 of the device.

32	Sync(00000001) SOF(0xA5) Frame #(0x490) CRC5(0x07)
33	Sync(00000001) IN(0x96) ADDR(0x00) ENDP(0x0) CRC5(0x08)
34	Sync(00000001) DATA1(0xD2) DATA(12 01 00 01 00 00 00 08) CRC16(0xC8E7)
35	Sync(__000001) ACK(0x4B)

The host receives the packet from our device successfully and it sends an ACK handshake back to our device, as indicated by packet #35.

36	Sync(00000001) SOF(0xA5) Frame #(0x491) CRC5(0x18)

The host then proceeds to the status stage of this control transfer by sending an OUT token and a null packet to our device, as shown in packet #38 and #39 respectively. Our device sends a handshake packet

37	Sync(00000001) SOF(0xA5) Frame #(0x492) CRC5(0x1A)
38	Sync(00000001) OUT(0x87) ADDR(0x00) ENDP(0x0) CRC5(0x08)
39	Sync(00000001) DATA1(0xD2) DATA() CRC16(0x0000)
40	Sync(_0000001) ACK(0x4B)
41	Sync(_0000001) SOF(0xA5) Frame #(0x493) CRC5(0x05)

The host PC then proceeds to send the SET ADDRESS command to the device, as shown in packet #122. The second byte of 05H can be decoded accordingly (packet #40).

Our device acknowledges the SET ADDRESS command (packet #123) as the SET ADDRESS command. The 3rd and 4th bytes of packet #122 are 02H, which is the new address assigned by the host to the

119	Sync(00000001) SOF(0xA5) Frame #(0x4F7) CRC5(0x12)

120	Sync(00000001) SOF(0xA5) Frame #(0x4F8) CRC5(0x02)
121	Sync(00000001) SETUP(0xB4) ADDR(0x00) ENDP(0x0) CRC5(0x08)
122	Sync(00000001) DATA0(0xC3) DATA(00 05 02 00 00 00 00 00) CRC16(0xD768)
123	Sync(_0000001) ACK(0x4B)

124	Sync(_0000001) SOF(0xA5) Frame #(0x4F9) CRC5(0x1D)
125	Sync(00000001) IN(0x96) ADDR(0x00) ENDP(0x0) CRC5(0x08)
126	Sync(00000001) DATA1(0xD2) DATA() CRC16(0x0000)
127	Sync(_0000001) ACK(0x4B)

and the host then proceeds to the status stage of this control transfer, as depicted by packets #125 to #127. Note that, before and during this status stage of SET ADDRESS, our device still communicates with the host using its default address 00H.

Only after this status stage, will the device be required to change its address to 02H as indicated by the SET ADDRESS command. The host will use this new assigned address to communicate with our device in subsequent transactions. For example, in packet #180, the ADDR of the setup packets is 02H.

128	Sync(00000001) SOF(0xA5) Frame #(0x4FA) CRC5(0x1F)

Standard SOF packets appearing every 1 ms are issued in this gap.

After the SET ADDRESS command, the host follows with a GET DESCRIPTOR (device) command as shown in packet #181. The second byte of the data, 06H, indicates a GET DESCRIPTOR command and the 3rd and 4th bytes showing 0001H indicate that the descriptor type is "device".

178	Sync(00000001) SOF(0xA5) Frame #(0x52C) CRC5(0x1A)

179	Sync(00000001) SOF(0xA5) Frame #(0x52D) CRC5(0x05)
180	Sync(00000001) SETUP(0xB4) ADDR(0x02) ENDP(0x0) CRC5(0x15)
181	Sync(00000001) DATA0(0xC3) DATA(80 06 00 01 00 00 80 00) CRC16(0xB129)
182	Sync(_0000001) ACK(0x4B)

The host then issues a series of IN tokens to attain descriptor information from our device. This is shown in packets #184, #189 and #194. The device responds accordingly and a total of 18 bytes are sent to the host as required by the USB Specification.

183	Sync(00000001) SOF(0xA5) Frame #(0x52E) CRC5(0x07)
184	Sync(00000001) IN(0x96) ADDR(0x02) ENDP(0x0) CRC5(0x15)
185	Sync(__000001) DATA1(0xD2) DATA(12 01 00 01 00 00 00 08) CRC16(0xC8E7)

186	Sync(00000001) ACK(0x4B)
187	Sync(00000001) SOF(0xA5) Frame #(0x52F) CRC5(0x18)
188	Sync(00000001) SOF(0xA5) Frame #(0x530) CRC5(0x0F)
189	Sync(00000001) IN(0x96) ADDR(0x02) ENDP(0x0) CRC5(0x15)
190	Sync(__000001) DATA0(0xC3) DATA(86 80 AD 0D 00 01 00 00) CRC16(0xCB53)
191	Sync(00000001) ACK(0x4B)
192	Sync(00000001) SOF(0xA5) Frame #(0x531) CRC5(0x10)
193	Sync(00000001) SOF(0xA5) Frame #(0x532) CRC5(0x12)
194	Sync(00000001) IN(0x96) ADDR(0x02) ENDP(0x0) CRC5(0x15)
195	Sync(00000001) DATA1(0xD2) DATA(00 01) CRC16(0xFCF1)
196	Sync(00000001) ACK(0x4B)
197	Sync(00000001) SOF(0xA5) Frame #(0x533) CRC5(0x0D)
198	Sync(00000001) SOF(0xA5) Frame #(0x534) CRC5(0x13)
199	Sync(00000001) OUT(0x87) ADDR(0x02) ENDP(0x0) CRC5(0x15)
200	Sync(00000001) DATA1(0xD2) DATA() CRC16(0x0000)
201	Sync(00000001) ACK(0x4B)
202	Sync(00000001) SOF(0xA5) Frame #(0x535) CRC5(0x0C)
203	Sync(00000001) SOF(0xA5) Frame #(0x536) CRC5(0x0E)

Again, more SOF packets are issued in this gap.

After finishing the GET DESCRIPTOR (device) command, the host then issues a GET DESCRIPTOR (configuration) to our device, as shown in packet #255. The 4th byte of this data, 02H, indicates a descriptor type of "configuration". Our device responds accordingly. In the response, the 3rd and 4th byte value of 0019H (packet #259) indicates the length of data to be returned for this configuration. For this case, 25 bytes of data will be returned, as shown in packets #259, #264, #269 and #274.

252	Sync(00000001) SOF(0xA5) Frame #(0x567) CRC5(0x0C)
253	Sync(00000001) SOF(0xA5) Frame #(0x568) CRC5(0x1C)
254	Sync(00000001) SETUP(0xB4) ADDR(0x02) ENDP(0x0) CRC5(0x15)
255	Sync(00000001) DATA0(0xC3) DATA(80 06 00 02 00 00 80 00) CRC16(0x9329)
256	Sync(00000001) ACK(0x4B)
257	Sync(_0000001) SOF(0xA5) Frame #(0x569) CRC5(0x03)
158	Sync(00000001) IN(0x96) ADDR(0x02) ENDP(0x0) CRC5(0x15)
259	Sync(00000001) DATA1(0xD2) DATA(09 02 19 00 01 02 00 40) CRC16(0xFFD9)
260	Sync(00000001) ACK(0x4B)
261	Sync(00000001) SOF(0xA5) Frame #(0x56A) CRC5(0x01)
262	Sync(00000001) SOF(0xA5) Frame #(0x56B) CRC5(0x1E)
263	Sync(00000001) IN(0x96) ADDR(0x02) ENDP(0x0) CRC5(0x15)
264	Sync(00000001) DATA0(0xC3) DATA(19 09 04 00 00 01 00 00) CRC16(0xED6B)

265	Sync(__000001) ACK(0x4B)
266	Sync(00000001) SOF(0xA5) Frame #(0x56C) CRC5(0x00)
267	Sync(00000001) SOF(0xA5) Frame #(0x56D) CRC5(0x1F)
268	Sync(00000001) IN(0x96) ADDR(0x02) ENDP(0x0) CRC5(0x15)
269	Sync(00000001) DATA1(0xD2) DATA(00 00 07 05 81 03 08 00) CRC16(0xB5FD)
270	Sync(00000001) ACK(0x4B)
271	Sync(00000001) SOF(0xA5) Frame #(0x56E) CRC5(0x1D)
272	Sync(00000001) SOF(0xA5) Frame #(0x56F) CRC5(0x02)
273	Sync(00000001) IN(0x96) ADDR(0x02) ENDP(0x0) CRC5(0x15)
274	Sync(__000001) DATA0(0xC3) DATA(10) CRC16(0x82CE)
275	Sync(_0000001) ACK(0x4B)
276	Sync(00000001) SOF(0xA5) Frame #(0x570) CRC5(0x15)
277	Sync(00000001) SOF(0xA5) Frame #(0x571) CRC5(0x0A)
278	Sync(00000001) OUT(0x87) ADDR(0x02) ENDP(0x0) CRC5(0x15)
279	Sync(00000001) DATA1(0xD2) DATA() CRC16(0x0000)
280	Sync(00000001) ACK(0x4B)
281	Sync(_0000001) SOF(0xA5) Frame #(0x572) CRC5(0x08)
282	Sync(00000001) SOF(0xA5) Frame #(0x573) CRC5(0x17)

More SOF packets appear here.

After the successful completion of the GET DESCRIPTOR (configuration) command, the host issues a SET CONFIGURATION command, as indicated by packet #334, and our device is required to act accordingly. The 2nd byte value of 09H (packet #334) indicates that this is a SET CONFIGURATION command.

332	Sync(00000001) SOF(0xA5) Frame #(0x5A5) CRC5(0x06)
333	Sync(00000001) SETUP(0xB4) ADDR(0x02) ENDP(0x0) CRC5(0x15)
334	Sync(00000001) DATA0(0xC3) DATA(00 09 02 00 00 00 00 00) CRC16(0xE468)
335	Sync(_0000001) ACK(0x4B)
336	Sync(_0000001) SOF(0xA5) Frame #(0x5A6) CRC5(0x04)
337	Sync(00000001) IN(0x96) ADDR(0x02) ENDP(0x0) CRC5(0x15)
338	Sync(00000001) DATA1(0xD2) DATA() CRC16(0x0000)
339	Sync(_0000001) ACK(0x4B)
340	Sync(00000001) SOF(0xA5) Frame #(0x5A7) CRC5(0x1B)
341	Sync(00000001) SOF(0xA5) Frame #(0x5A8) CRC5(0x0B)
342	Sync(00000001) SOF(0xA5) Frame #(0x5A9) CRC5(0x14)

After this last sequence of the enumeration, the device is ready for use.

2.10.3 USB Reset, Suspend, Resume and Remote Wakeup

USB devices are required to conform to the USB reset, suspend and resume signaling. The remote wakeup capability of a device is optional. The reset signal is to ensure the robustness of the USB device. When the USB device sees a SEO (single-ended zero; both D+ and D- are low) on the USB wires for more than 2.5 μsec, it's required to treat the signal as a USB reset signal. Typically the host issues a USB reset signal of 10 ms in length. After the USB reset is removed, all devices that receive the reset signal are set to their default address and are in the unconfigured state (default stage). All devices must be able to accept new device addresses via a SET ADDRESS setup token no later than 10 ms after the reset is completed (see section 7.1.4.3, USB Specification[11]).

Suspend and resume signaling are used to reduce power consumption of bus-powered USB devices as the overall power supply is limited in a USB system. The bus-powered devices must go into a suspend mode when they see a constant idle state on the USB wires for more than 3 ms. Any bus activity, including the SOF packets, will keep the devices from going into suspend mode. In the suspend mode, the devices must draw less than 500 μA from the bus (see Section 7.1.4.4 of the USB Specification[11]). Once the device is in this mode, its operation can be resumed by receiving a non-idle signaling from the host. The device can also signal the system to resume operation if it has the remote wakeup capability (see section 7.1.4.5 of USB Specification[11]). Note that the remote wakeup is considered to be an exceptional event in the USB protocol because it is initiated by devices (with the remote wakeup capability configured). The host initiates all other activities in a USB system.

2.11 USB Host

The basic flow and interrelationships of the USB communications model are shown in Figure 2-15. The model is divided into three layers; Function, USB Device and USB Bus Interface Layers. Black arrows indicate the actual communication flow. All communications between the host and the device ultimately occur on the USB bus interface layer. However, there are logical host-device interfaces between each horizontal layer shown with the gray arrows. This layering concept is widely used to describe a communication system, and is based on the seven-layer OSI model. Basically, the entities in the same layer have a virtual connection or "peer-to-peer" communication between them. The entities in the lower layer provide services to their next entities in the

higher layer. In this fashion, a PC application software designer, for example, does not have to worry about the physical connections of the devices. The applications will work as long as the guidelines or specifications to use the services provided by the "device" layer are followed.

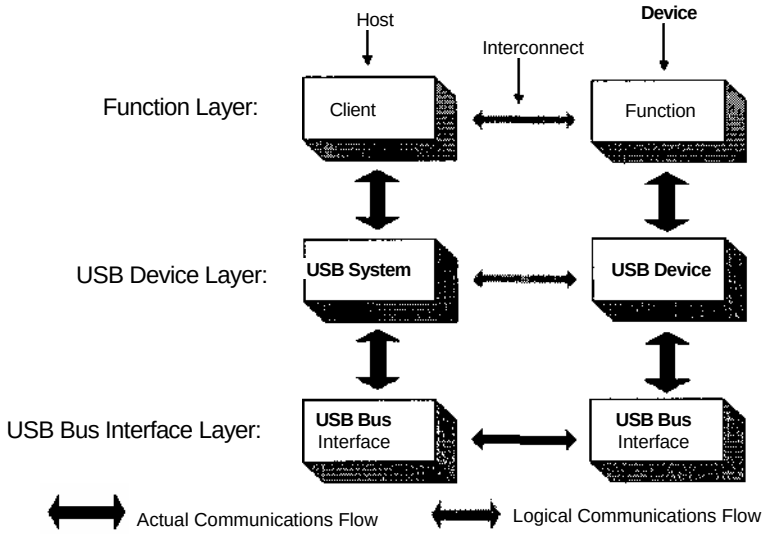


Figure 2-15. Inter-layer communication model

As shown in Figure 2-15, the lowest layer of the communication model is the USB Bus Interface layer. It handles the physical connection between the host and the USB device. On the host side, this is the USB host controller hardware. The next layer, the USB Device Interface Layer, uses the services provided by the USB Bus Interface layer to manage the data transfer between the host and the USB device. This USB Device layer in the host has three basic components:

- Host Software (optional)
- USB Driver (USBD)
- Host Controller Driver (HCD)

The HCD provide an interface for USBD to interact to the host controller hardware. The interface between the USBD and the HCD is known as the Host Controller Driver Interface. The Microsoft Windows 95 operating system supports Open Host Controller Interface (OHCI) and Universal Host Controller Interface (UHCI). The USBD, in turn, provides services to the Client Layer. The Client Layer describes all of

the software entities that are responsible for directly interacting with their USB devices. Certain device classes of USBD's are provided with the operating system. If the driver required by a device is not provided by the operating system, a custom driver needs to be developed. Chapter 6 describes the development process of USB drivers.

3. Step 2: Set Up a Development Environment

With a basic understanding of the USB Specification, you are now ready to design a USB device. For the purpose of this book, and because of the flexibility and compliance to the USB Specification, this book examines using the Intel 8x930Ax USB microcontroller to design a USB device. The 8x930Ax is the first general purpose USB microcontroller available.

Let us first re-examine at the steps required to develop a USB device, as mentioned in Chapter 1. Figure 3-1 summarizes the tasks needed to develop a USB device.

Step 3 to step 6 can be taken in parallel if you have a tight schedule and if your resources permit. Typically, for a simple USB device, a minimum of two designers are required; one to focus on the USB device hardware and firmware, and the other concentrates on the device drivers and application software on the USB host. Companies wishing to design more complicated USB devices need to gather more resources into the project; for example; one for hardware, one for firmware, one for device driver and one for application software. To design very complicated devices, staff of up to 15 persons for each USB project might be needed.

This chapter describes the process of setting up a development environment using the 8x930Ax microcontroller solution for the device side. It should be noted that setting up a development environment for other solutions would be similar.

3.1 Development Environment (Device Side)

There are two main development environments in creating a USB device; one focusing on the USB device itself and the other on the USB host. A typical development environment to build and test the device firmware and hardware is shown in Figure 3-2.

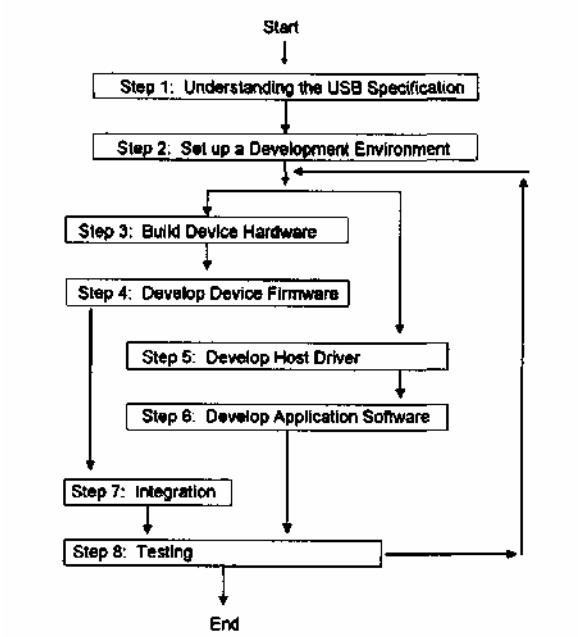


Figure 3-1. Tasks to design a USB device

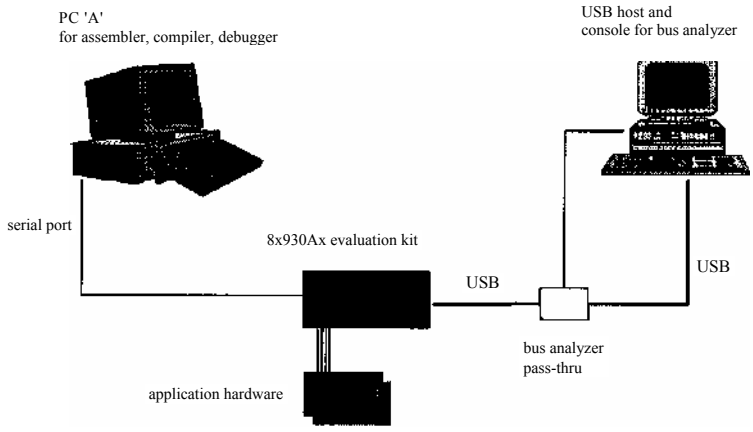


Figure 3-2. Development environment (device side)

We first purchase an 8x930Ax USB Evaluation Kit. It consists of a 8x930Ax microcontroller on an evaluation board, an evaluation copy of an assembler, compiler and debugger either from PLC, Tasking or Keil, and Apbuilder software which helps designers understand the

architecture of the 8x930Ax microcontroller. The contact numbers to purchase this Evaluation Kit are listed in the next section. In the development environment above, firmware engineers can write their codes on a PC (called PC 'A' in Figure 3-2), and then test the codes by downloading and then executing them on the evaluation board. The codes are downloaded through the serial port of the PC. Firmware, called RISM, residing in the Evaluation Kit performs the communication between the PC and the evaluation board through the serial port. The evaluation board also provides an interface to the USB device we will build. Hardware designers can test out prototypes by interfacing to this evaluation board.

Testing the device firmware can be done in a modular fashion. For example, you can try out the operation of the firmware for the device without USB functionality first. In the other firmware module, we can test the USB functions independently. When both firmware modules work well separately, they can be merged together and the testing can continue. A code example for the firmware to do a USB bus enumeration, called USB_ENUM.ASM, is on the enclosed diskette and listed in Appendix A. This code is meant as a design example and it must be modified to suit the operation of the test device.

As shown in Figure 3-2, we use a separate USB host PC to send out USB packets to test and debug our firmware on the USB device. The Contact Numbers section provides information on some of the USB "ready" PCs that can currently be purchased from your local computer stores. These PCs can then be configured to be the USB host system for testing purposes. One software tool you'll need is the Single Step Transaction Debugger (SSTD) that can be used in the host. This SSTD allows us to send a single USB packet through the USB wires to test our firmware in the evaluation board. In this manner, we can test the operation of our code when it receives a single USB packet.

A bus analyzer is also included in the development environment, as shown in Figure 3-2. It is used to capture and display all the USB traffic on the USB wires that the analyzer taps into. This is useful for testing and debugging purposes because it allows you to visualize what is actually happening on the USB wires. This is a third party tool and the contact information is listed in the Contact Numbers section.

In some cases, designers may choose to use a USB in-circuit emulator (ICE) to help test and debug the code. For example, an ICE enables emulating the performance of the 8x930Ax when code execution is performed from internal ROM. The Evaluation kit, as described earlier,

executes code from external memory. The speed of code execution from external memory is slower than the speed of code execution from internal ROM, thus making the internal ROM choice preferable if you have the available resources. More detailed descriptions of the 8x930Ax microcontroller and its firmware operating model are available in Chapters 4 and 5, respectively.

3.2 Development Environment (Host Side)

On the host side, two PC systems are recommended to develop the USB drivers and applications software. To develop the USB drivers, you'll need a PC running the Windows NT operating system, version 4.0 or higher, Microsoft Visual C/C++ Professional Version 4.0 or higher, and a subscription to the Microsoft Developer Network, MSDN (level II, professional level). The subscription to the MSDN is to obtain the Windows NT DDK and Win32 SDK (please refer to the Contact Numbers section for more information). You'll also need to obtain the USB DDK from Microsoft by contacting the e-mail address: usbbeta@microsoft.com or contact the USB-IF. This software then needs to be installed in the PCs; the recommended sequence of installation of the software kits is as follows: Windows NT first, then Visual C++ Professional Edition 4.0 (or higher), followed by Win32 SDK, Windows NT DDK, and finally the USB DDK. With all of these components in place, you are now ready to develop the USB driver for your USB device.

The USB driver is developed in the Windows NT PC as shown in Figure 3-3. After its compiled into an executable format, the driver is transferred to the host PC for code execution and debugs. The host PC is required to operate a Windows 95 operating system [either OEM Software Release 2.1 (OSR 2.1, also codenamed 'Detroit') or the 'Memphis' operating system] with the Microsoft Windows Driver Model (WDM) stack loaded. The WDM stack can also be obtained from *usbbeta@microsoft.com, memreq@microsoft.com,* or the USB-IF.

For *driver development:*
Windows NT™ OS
- *Win32 SDK*
- *Windows NT™ DDK*
- *USB DDK*
- *Microsoft® Visual C/C++*

For driver execution and test:
Windows 95 OS
- *Windows 95 with WDM stack*

Figure 3-3. Development environment (host USB driver)

So, in summary, to develop a USB device driver, two PCs are recommended; one PC to develop the driver and the other to execute and debug the driver, as shown in Figure 3-3. We can also use debugging software, like SoftICE™, to debug the driver that we have developed.

It is very important to know if the driver for the device is to be bundled with the operating system in the time frame we need. If the driver is bundled together with the operating system, you do not have to develop the driver yourself. All that you are required to do is to follow the USB device class specifications when you design the device. Details about device class specifications will be discussed in Chapter 6.

On the host applications side, we can use Microsoft Visual C++ Developer Studio AppWizard to create the applications. Chapter 7 will discuss the development of the application in greater detail.

3.3 Contact Numbers

Description	Contact Numbers	http://www.
Intel USB930EVALKIT; 8x930Ax Evaluation Kit and Adapter Card, Apbuilder, evaluation copy software, (hardware and software)	US: (800) 628 8686 UK: 44 01793 431155 Israel: 972 03 548 3200 Apac: 852 2844 4555 Japan: 81 298478511	intel.com/design/usb intel.com/design/usb /manual - for the manual
Intel Apbuilder software can also be obtained from (software)		intel.com/design/BUILDER /apbldr
Single Step Transaction Debugger (SSTD) (software)		intel.com/design/usb /swsup
USB host commercially available (as of publication date): • IBM: model 2176C6Y or higher • Toshiba: model name Infinia . Sony: model PCV70, VAIO • Compaq; model number 4762 or higher • Siemens: model Scenic Pro C5 (PC hardware)		pc.ibm.com/aptiva toshiba.com
CATC USB Detective bus analyzer, USB Traffic Generator, USB Host Production Tester (hardware)	US: (800) 638 2423 (408) 727 6600 Europe: 46 40 59 2200 Apac: 886 02 278 1938 Japan: 81 332451351	catc.com
Microsoft Software Developer Network (MSDN) Information	US: (800) 759 5474 fax:US: (206) 936 2490 msdn@microsoft.com	microsoft.com/msdn
To get beta copy of Microsoft WDM USB DDK	usbbela@ m icrosoft . com memreq@microsoft.com	

Contact Numbers

Description	Contact Numbers	http://www.
SoftICE debugger for USB driver (Win 95}	US: (603) 889-2386	numega.com
TASKING C Toolset 251/930; assembler, compiler and debugger (software)	US: (61 7} 320 9400 Europe: 31 33 55 8584	tasking.com
Keil Software DK/251; assembler, compiler and debugger (software)	US: (800) 348 8051 Europe: 49 8946 5057 Apac:657492162	keil.com
Production Languages Corporation (PLC) COMPASS/251; assembler, compiler and debugger (software)	US: (800) 525 6289 Europe (81 7} 599 8363	plcorp.com
SystemSoft 8x930 Success Program: Turnkey solutions on USB device drivers (software)	US: (805) 486 6686 Taiwan: 545 5370	systemsoft.com
Phoenix Technologies System Essentials Utility; class/device driver, firmware	US: (408) 654 9000 Eur: +44 1483 243200 Apac:+88627188956 Japan: +81 3 3374 6555	ptltd.com
American Megatrends Inc AMIBIOS 95+ USB; BIOS USB capability	US: (770) 246 8600	megatrends.com
Metalink RVM-930 emulator or EM-930 8x930x ROM emulator (hardware emulator)	US: (602) 926 0797 Europe: 49 8091 56960 Korea: 82 2 517 3707	
Nohau EMUL251 POD930 8x930Ax In Circuit Emulator (hardware emulator)	US: (408) 866 1820 Europe: 46 40 922425 Korea: 82 2 784 7841 Japan: 81 0334050511	nohau.com
iSYSTEMS iC2000 PowerEmulator; (hardware emulator)	US: (207) 236 9055 Europe: 49 81 31 70610 Korea: 82 2 783 7835 Japan: 81 334050511	isystem.com

4. Step 3: Developing The Device Hardware

As mentioned in the previous chapter, the Intel 8x930Ax USB microcontroller will be the reference base for the USB device we will build. This chapter examines the architecture of the 8x930Ax microcontroller its USB module, and how we can use it in our device.

4.1 Overview of the 8x930Ax USB Microcontroller

The 8x930Ax microcontroller is the world's first USB Spec 1.0 compliant microcontroller commercially available. It is meant to be used to develop a hubless USB device. A detailed description of the 8x930Ax microcontroller can be found in the "8x930Ax, 8x930Hx USB Microcontroller User's Manual"[3] and datasheet[4]. This section will summarize the important features of the microcontroller.

The 8x930Ax is based on a MCS 251 core, and has various on-chip peripherals (an industrial standard MCS 51 Universal Asynchronous Receiver and Transmitter (UART), 5 modules of programmable counter arrays (PCA), 3 general purpose timers/counters, a hardware watchdog timer) and a USB module. It is available in various ROM and RAM sizes as shown in Figure 4-1.

Part Name	ROM size (bytes)	RAM size (bytes)
80930AD	0	1K
83930AD	8K	1K
83930AE	16K	1K

Figure 4-1. 8x930Ax product options

4.1.1 MCS 251 Core

The MCS 251 core is considered to be the next generation core of the MCS 51 8-bit core that has been widely used for over 15 years. The MCS

251 has a 3-stage pipeline CPU core which offers higher performance as compared to the MCS 51. When the pipeline is full and code is executed from internal memory, some instructions can be completed in one state time; one state time is equivalent to one oscillator clock for the case of the 8x930Ax when its internal Phase Lock Loop (PLL) is enabled. To enable the PLL, pins PLLSEL2, PLLSEL1 and PLLSEL0 need to be pulled high, high, and low respectively. The 8x930Ax is also binary code compatible to the MCS51 core. Using MCS51 instructions in the MCS251 core gives users up to a five times performance improvement. If new MCS 251 instructions are used (many new instructions have been introduced to give 16-bit type of capability; for example 16 bit multiplication and divide), up to a 15 times performance improvement can be achieved over the MCS 51 core.

The MCS 251 core is designed with a register-based architecture and has 40-byte registers that can be accessed as bytes, words, or double words. The registers can also be used for register to register addressing. For example, we can move a byte between registers using just one instruction:

MOV R6, R0.

This instruction can be completed in one state time. For the case of the 8x930Ax at 12 MHz, this translates to 83.33 ns.

Compared to the accumulator-based architecture of its predecessor, the 8xC51Fx with a MCS 51 core, which required two instructions and 12 state times (one state time is two oscillator clocks for 8xC51Fx) to achieve the similar goal:

```
MOV A, R0          ; requires 6 state time
MOV R6, A.         ; require 6 state time
```

At a 12 MHz crystal frequency, the 8xC51Fx takes 83.33x2x6x2=1999.92 ns to complete these two instructions to achieve the transfer of data from RO to R6.

4.1.2 Memory Organization

In terms of addressing space, the 8x930Ax can address up to 256 Kbytes, providing you with greater flexibility. Figure 4-2 shows the addressing space of the 8x930Ax. Note that the MCS 251 core has the capability of addressing up to 16 Mbytes of address space, but the 8x930Ax is limited to 256 Kbytes.

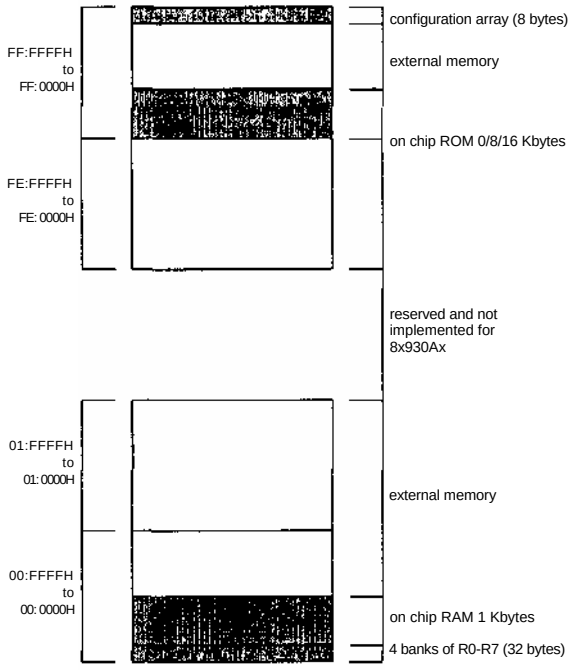


Figure 4-2. 8x930Ax addressing space when configured to 256 Kbytes of addressing.

This 256 Kbyte is user-selectable, however, so you can choose whether you would like to have 256 Kbyte, 128 Kbyte or 64 Kbyte of addressibility by programming the configuration bytes of the 8x930Ax. The configuration bytes also allows you to choose the number of wait states required to interface to external memory, to configure source or binary mode, etc. for the 8x930Ax. You can select no wait state, one, two, or three wait states to interface to external memory. There is also a wait pin to provide a real time wait function to interface with slower external components. The configuration bytes are located in FF:FFF8H and FF:FFF9H locations for the 8x930Ax with internal ROM. For the ROMless version, you are required to put these two configuration bytes in the highest F8 and F9 locations of external memory. For example, if you use a 32 Kbyte EPROM, you would put the configuration bytes in the locations 7FF8H and 7FF9H; and for the case of 64 Kbytes EPROM, you would put the configuration bytes in FFF8H and FFF9H. Chapter 4 of the 8x930Ax, 8x930Hx USB Microcontroller User's Manual[3] describes the configuration bytes in detail.

The 8x930Ax has a 24-bit program counter and a 24-bit data pointer to address up to 256 Kbyte of addressing space. We can use the double word indexing capability of the registers to address to the 256 Kbyte of address space. For example,

MOV @DR28, R4

Similarly, there are EJMP addr24 and ECALL addr24 to jump or call to a subroutine pointed to by the 24 bit address, addr24, respectively.

After chip reset, the program counter of the 8x930Ax is initialized to the FF0000H location; this is where the first line of code should be placed.

4.1.3 External Memory Interface

There are two modes available for the 8x930Ax to interface to external memory; page and non-page modes. In non-page mode, the lower address (A0-A7) and data byte (D0-D7) are multiplexed using port 0. In this mode, it takes two states to fetch a code from external memory with 0 wait state. On the other hand, using page mode, a code can be fetched in one state time. In this page mode, the data (D0-D7) is multiplexed with higher address byte (A8-A15) in port 2, as shown in Figure 4-3.

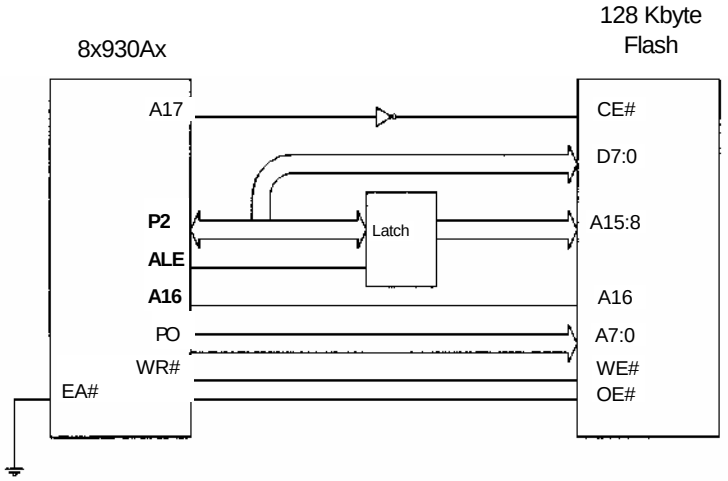


Figure 4-3. Page mode for external memory access

In this mode, the first code fetch takes two states. But, for successive code fetches that are within the same 256-byte block of memory, it only requires one state. This is because the ALE latches the higher address

(A8-A15) only once; and the address that was latched previously is reused for subsequent code fetches that are within the same 256-byte block. Port 2 is now available to serve as the data bus, while port 0 carries the lower address byte. The ALE is not asserted for the second fetch, so this fetch takes only one state time. Another advantage of limiting the toggling of ALE signal is the reduction of unwanted electromagnetic radiation.

For data reads and writes to RAM devices, for example, three states are required, either in page or non-page mode. We can add extra wait states (up to three wait states) or use the real time wait function if we are interfacing to slower devices. With 0 wait state, a byte can be read or written in three state time. This gives the maximum throughput of 1 byte/3 oscillator clock cycles at 12 MHz, i.e., 32 Mbps, without taking into account other firmware overhead.

The 8x930Ax also has four 8-bit I/O ports for general propose use. Ports 0 and 2 will be occupied if code execution is from external memory. Port 0 has an open-drained structure and ports 1, 2 and 3 have weak pull-ups.

4.1.4 Interrupt System

The 8x930Ax employs a program interrupt method, where the interrupt operation branches to a particular address space, called the Interrupt Service Routine (ISR) vector address, and performs a subroutine as stated by the instructions in that vector address. When the subroutine completes and the return interrupt instruction (RETI) is issued, the execution of the program resumes at the point where the interrupt occurs. For example, assume that we are executing our program, looping from a program counter ranging from FF:0200H to FF:0500H. When an interrupt occurs, caused by, say, timer 1, the value of the program counter before branching to the ISR will be stored into a stack and the program execution will vector to the address of the timer 1 ISR, which is FF:001BH. The 8x930Ax would execute whatever instructions are at that location and when it finishes the ISR, i.e., after the RETI instruction, the previous value of the program counter is restored. Our program execution will "return" back to where the interrupt occurred (between FF:0200H and FF:0500H in our example).

The 8x930Ax has 11 interrupt sources; 10 maskable and a TRAP instruction. The maskable sources are:

- 2 external interrupts
- 1 PCA interrupt
- 1 serial port interrupt
- 3 timer interrupts
- 3 USB interrupts

The TRAP interrupt is intended for tools' development only. All the maskable interrupts have interrupt request flags associated with them. Some of these flags tell us the exact source of the interrupt. Let us take the serial port interrupt for example. When a serial port interrupt occurs at ISR vector address of FF:0023H, our firmware is required to check the RI or TI flags (of SCON special function register (SFR)) whether the interrupt is caused by a reception (RI flagged) or transmission (TI flagged) of the serial port. All of these maskable interrupts have four user-selectable priorities and can be individually enabled/disabled. There is also one global enable bit, EA bit, in the IEN0 SFR, to enable or disable all of the maskable interrupts.

There are three interrupt sources associate with the USB module of the 8x930Ax:

- one for the transmit and receive interrupts associated with each endpoint pair,
- one for the start of frame (SOF) interrupt, and
- the other one for the USB suspend, resume, or reset interrupt.

The transmit and receive interrupts occur when a non-isochronous endpoint is transmitted or received in a USB packet. When an endpoint is configured as isochronous, no interrupt occurs from the reception or transmission of a USB packet. In this case, we use the SOF interrupt to manage the data of this endpoint. The SOF interrupt occurs when a SOF packet is received every 1 ms or when a SOF packet should have been received; this is the case when the so-called artificial SOF is received. The artificial SOF is generated internally by the 8x930Ax frame timer to simulate the reception of the SOF when the true SOF packet is not received successfully. The suspend or resume interrupt occurs when there are suspend or resume conditions detected on the USB wires. For the case of suspend interrupt, the USB module of the 8x930Ax will issue a USB suspend interrupt when there are no USB bus activities detected for more than 3 ms (as defined by the USB Specification).

4.1.5 8x930Ax On-Chip Peripherals

There are a number of peripherals integrated in the 8x930Ax to give users greater flexibility in using the microcontroller. They include three timers/counter units, a hardware watchdog timer, a programmable counter array (PCA), and a serial port (UART).

4.1.5.1 Timer/Counter Units

There are three general purpose timer/counter units in the microcontroller. They can be used as general purpose 8-bit, 13-bit or 16-bit timers/counters, timers for captured events on an external pin, timers for outputting a programmable clock signal on an external pin, or as a baud rate generator for the on-chip serial port (UART), etc.

4.1.5.2 Hardware Watchdog Timer

When initiated by firmware, the watchdog timer starts running, and unless the firmware intervenes, it will reach a maximum count and reset the controller. It is used to return the device to a known stage in the event of unrecoverable errors.

4.1.5.3 Programmable Counter Array (PCA)

The PCA has its own timer and five capture/compare modules that can be used to perform several functions, such as:

- capturing the timer value in response to a transition on an input pin
- generating an interrupt request when the timer matches a stored value
- toggling an output pin when the timer matches a stored value
- generating a Pulse Width Modulation (PWM) signal, which can be used as a D/A converter
- serving as a software watchdog timer

4.1.5.4 Serial Port (UART)

There is one configurable, four-mode, two wire serial port in the 8x930Ax. Mode 0 enables the serial port to behave as a half duplex synchronous serial port; one pin (TxD pin) outputs the clock signal and the other pin (RxD pin) receives and transmits data. The baud rate is at *2* Mbps for the 8x930Ax operating on a crystal clock of 12 MHz with PLL enabled. In mode 1, 2 or 3, the serial port is in its asynchronous mode

with one pin (TxD pin) to transmit and the other pin (RxD pin) to receive the data. In these modes, the data frame can be 10 or 11 bits; one start bit, eight data bits, (one programmable ninth bits) and one stop bit. The maximum programmable baud rate is 750.0 Kbaud for the 8x930Ax, using rimer 2 as a baud rate generator.

4.1.6 8x93QAx USB Module

The on-chip USB module of the 8x930Ax provides an excellent and simple interface to the USB wires. The detailed connection of this USB interface is described in the Interface with the 8x930Ax section. The USB module consists of a transceiver, a serial bus interface engine (SIE), a function interface unit (FIU) and USB FIFOs, as shown in Figure 4- 4.

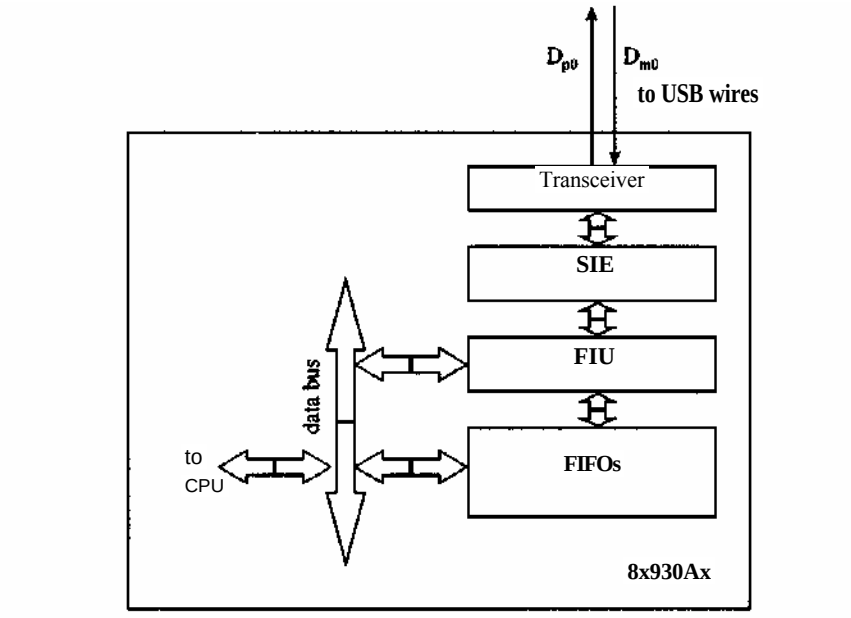


Figure 4- 4. The USB module of the 8x930Ax

The input/output pins (D_{p0} and D_{m0}) of the transceiver enable it to connect directly (with a series resistor each) to the USB wires of D+ and D- (see Interface with the 8x930Ax section). The SIE unit provides packet decoding/generation, CRC generation and checking, NRZI encoding/decoding, and bit stuffing capabilities, as required by the USB Specification. The FIU provides an interface for the SIE to the FIFOs and then to the MCS 251 core of the 8x930Ax controller. There are four pairs of FIFOs in the 8x930Ax, corresponding to four pairs of USB endpoints.

Endpoint (in bytes)	Transmit	Receive
0	16	16
1 (configurable)	256 512 1024 0	256 512 0 1024
2	16	16
3	16	16

Note that the 8x930Ax can also be configured to have six endpoint pairs (by setting the SIXEPPEN bit in the EPCONFIG SFR (see the 8x930Ax, 8x930Hx USB Microcontroller User's Manual[3] for details)

All the endpoints support isochronous and non-isochronous transfer. For example, endpoint pair 0 can be configured as control endpoints (as required by USB Specification), endpoint pair 1 as isochronous endpoints, endpoint pair 2 as bulk endpoints and endpoint pair 3 as interrupt endpoints. All the FIFOs can support up to two datasets. For example, you can divide the transmit endpoint 2 into two data sets of 8 byte FIFOs. Fill in the first 8 bytes of the dataset, and while waiting for the USB token for this endpoint, fill in the other 8 bytes for the second dataset for this same endpoint.

From the device point of view, you only have to take care to fill in or take out the data from the FIFOs. The rest of the USB requirements, like CRC, NRZI coding/decoding, bit stuffing, etc., are done by the SIE. For example, to transmit a packet of 10 bytes to the USB wires through endpoint 0 would require transferring the 10 bytes of data into the FIFO of endpoint 0, and then writing a byte count of 10 into the byte counter register. In assembly language, this would be:

```
; assuming we have the 10 bytes of data in address location 00:A000H to 00;A009H
MOV       WR30, #A000H
MOV       R6, @WR30              ; R6 as temporary storage
MOV       EPINDEX, 00H           ; select endpoint 0
MOV       TXDAT, R6              ; move the data to transmit FIFO of endpoint 0

INC       WR30, #03 H            ; increase our data pointer
MOV       R6, ©WR30              ; move the next byte of data
MOV       TXDAT, R6

INC       WR30, #01H

; and so on until we finish moving 10 bytes of data to TXDAT, and then

MOV       TXCNTL, #0AH           ; write to the byte count register
```

After writing the byte count TXCNTL, you can continue to perform other functions for the USB device. When an IN token arrives for the endpoint 0, the SIE will transfer these 10 bytes AUTOMATICALLY into the USB wires with appropriate CRC, bit-stuffing and NRZI coding, according to the USB Specification.

The FIFO size for endpoint pair 1 is configurable with 1 Kbyte total available. If you configure the endpoint to be 1 Kbyte for transmitting, 0 bytes in receiving, and use the FIFOs in dual dataset mode for isochronous transfer, it can transmit at the rate of 4,096 Kbps (512 byte per USB frame of 1 ms).

4.1.7 SFRs Associated with the USB Module

There are 26 Special Function Register(s) (SFRs) related to the operation of the USB module in the 8x930Ax microcontroller. Their descriptions are shown in the following table:

Mnemonic	Description
EPCON	Endpoint Control Register. To configure the operation of the endpoint specified by the EPINDEX. It is used to configure, stall, enable control, transmit, and receive capability in the endpoint.
EPCONFIG	Endpoint Configuration Register. To select four endpoint pairs (defualt) or six endpoint pairs.
EPINDEX	Endpoint Index Register. To select the appropriate endpoint.
FADDR	Function Address Register. To store the address for the device after receiving the SET_ADDRESS command from the host during device enumeration. Its default value is 00H as required by the USB Specification.
FIE	USB Function Interrupt Enable. To enable or disable the receive and transmit done for the 4 endpoint pairs.
F1E1	USB Function Interrupt Enable 1 . To enable or disable the receive and transmit done (for endpoint pairs four and five if selected, see EPCONFIG).
FIFLG	USB Function Interrupt Flag Register. To store the transmit and receive done interrupt flags when the endpoints are configured as non-isochronous endpoints (for the four endpoint pairs).
FIFLG1	USB Function Interrupt Flag 1 . To store the transmit and receive done interrupt flags when the endpoints are configured as non-isocronous endpoints (for endpoint pairs five and six if selected, see EPCONFIG).
IEN1	Interrupt Enable Register 1. To enable individual programmable interrupts for the USB interrupts.
IPL1	Interrupt Priority Low Register 1 . To establish relative priority for programmable interrupt. It is used in conjunction with IPH1 .

Mnemonic	Description
IPH1	Interrupt Priority High Register 1 . To establish relative priority for programmable interrupts. It is used in conjunction with IPL1 .
PCON1	To store the USB suspend, resume, reset flags. These flags will be set accordingly when there is a USB suspend, resume, and reset.
RXCNTH	Receive FIFO Byte Count High Register. To store the MSB byte count for the data packets received in the receive FIFO specified by EPINDEX.
RXCNTL	Receive FIFO Byte Count Low Register. To store the LSB byte count for the data packets received in the receive FIFO specified by EPINDEX.
RXCON	Receive FIFO Register. To control the receive FIFO specified by the EPINDEX. tt enables or disables the flushing of the FIFO, isochronous transfer, and automatic receive management.
RXDAT	Receive FIFO Data Register. Receive FIFO data, as specified by EPINDEX, is read from this register. It serves as the interface between the microcontroller and the USB FIFOs.
RXFLG	Receive FIFO Flag Register. To store the flags indicating the status of the data packets in the receive FIFO as specified by the EPINDEX. It has the receive index flag, empty flag, full flag, underrun flag, and overrun flag.
RXSTAT	Endpoint Receive Status Register. To store the status indicators of the re-ceive FIFO specified by EPINDEX. It contains receive endpoint sequence, receive setup token, overwritten, end overwrite, receive data sequence overwrite, receive void, receive error, and receive acknowledge bits.
SOFH	Start of Frame High Register. To store the upper 3 bits of the 1 1-bit time stamp from the SOF packets from the host. It also contains SOF token receive acknowledgment bit, any SOF bit, SOF interrupt enable bit, frame timer lock bit, and SOF# pin disable bit.
SOFL	Start of Frame Low Register. To store the lower eight bits of the 1 1-bit time stamp from the SOF packets from the host.
TXCNTH	Transmit Count High Register. To store the MSB byte count for the data packets to be transmited in the transmit FIFO specified by EPINDEX.
TXCNTL	Transmit Count Low Register. To store the LSB byte count for the data packets to be transmited in the transmit FIFO specified by EPINDEX.
TXCON	Transmit FIFO Control Register. To control the transmit FIFO specified by the EPINDEX. It enables or disables the flushing of the FIFO, isochronous transfer, and automatic transmit management.
TXDAT	Transmit FIFO Data Register. Transmit FIFO data, as specified by EPINDEX, is written to this register. It serves as the interface between the microcontroller and the USB FIFOs.
TXFLG	Transmit FIFO Flag Register, To store the flags indicating the status of the data packets in the transmit FIFO as specified by the EPINDEX. It has the transmit index, empty, full, underrun, and overrun flags.
TXSTAT	Endpoint Transmit Status Register. To store the status indicators of the transmit FIFO specified by EPINDEX. It contains transmit endpoint sequence, overwritten, end overwrite, transmit void, transmit error, and transmit acknowledge bits.

It is very useful to organize these SFRs into logical groups for better understanding of their operations:

USB functions generic SFRs:	EPCONFIG.FADDR, SOFH, SOFL, PCON1, IEN1, IPL1, and IPH1	
Endpoint specified SFRs: (Affected by EPINDEX)	EPCON RXDAT RXCNTL RXFLG RXCNTH RXSTAT RXCON	TXCNTH TXDAT TXCNTL TXFLG TXCON TXSTAT
Endpoint specified SFRs: (Not affected by EPINDEX)	FIE, FIEI, FIFLG, AND FIFLG1	

4.2 Interface with the 8x930Ax

The interface of the 8x930Ax to USB is simple; the USB module even has a transceiver integrated. You only need to connect series resistors of 27 to 33 Ω to the D_{p0} and D_{m0} pins, and to connect a resistor of 1.5 *kΩ.* to the D_{p0} (corresponding to the D+ line of the USB wires), as shown in Figure 4-5. The connection of the pull-up 1.5 kΩ resistor is to configure the 8x930Ax for a full speed device. If you use the 8x930Ax as a low speed USB device, the 1.5 *kΩ.* pull up resistor is required to connect to the D_{m0} (corresponding to the D- line of the USB wires) instead. The Evaluation Kit (see Chapter 3 for more details) provides a schematic of the evaluation board and it can be used as a reference.

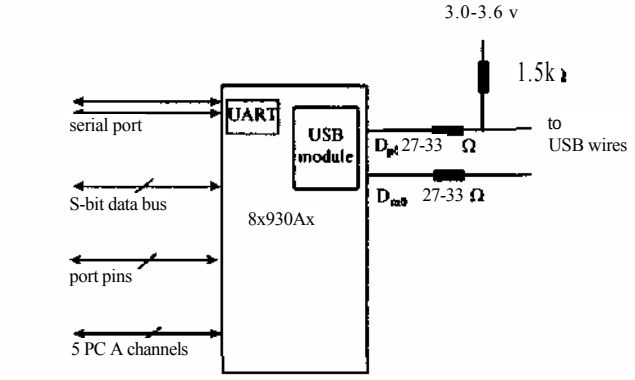


Figure 4-5. Interfaces of the 8x930Ax

To interface the 8x930Ax on the other side of USB, use the general purpose I/O ports, the external memory bus, the PCA, or the serial port, as shown in Figure 4-5. The 8x930Ax has 256 Kbytes of addressibility that can be used to map to different components of our USB device.

5. Step 4: Develop the Device Firmware

This chapter describes the techniques in developing the device firmware using the 8x930Ax microcontroller. Sample code for the USB enumeration process is included on the enclosed diskette, ready for you to program into a 32 Kbyte EPROM or FLASH memory. Once programmed, this EPROM or FLASH can directly plug into the EPROM socket of the 8x930Ax Evaluation Board. It will then be ready for USB enumeration when you plug its USB wires into a USB host. You can also design your own hardware and use this programmed EPROM or FLASH to achieve the same purpose.

5.1 Overview of Firmware Resources

5.1.1 USB Firmware Overhead

To handle the USB protocol, extra firmware needs to be included together with your application-specific firmware. The size of the USB firmware overhead depends on how complete and how many USB options are used. At minimum, the firmware needs to support all requirements of Chapter 9 of the USB Specification 1.0[1]. If more commands, such as HID device class (more details on device class specifications are in Chapter 6) commands are supported, the size of the firmware will increase accordingly. Based on the sample code that is on the enclosed diskette, the extra code required to support USB is approximately 1 to 2 Kbytes in size (using the 8x930Ax).

5.1.2 Application-Specific Firmware

Application-specific firmware for the device will require some minor changes. For example, if the microcontroller is used in a USB keyboard, it is still required to scan the keyboard and encode the data when a key is pressed. However, after encoding the data, the microcontroller is required to transfer the data to its transmit FIFO to get ready for

transmission through the USB wires. For the ease of debugging, you can develop the firmware in two separate modules initially; one for USB-related routines and the other for application-specific routines. When the two firmware modules are fully tested, you can then integrate them together. The USB firmware provided with this book will give you a good start in developing the USB routines. You will need to develop the application-specific routines separately and then merge them together for the USB device.

5.2 Firmware Relationship with Device Class Driver

The firmware developed for the device must work closely with the device driver on the host; and thus the firmware designers and device driver designers are often required to work closely together. There are various device drivers available bundled together with Microsoft Windows 95 and future operating system releases. If drivers are available for the class of device you intend to develop, you are not required to write your own driver. What you are required to do is to follow exactly the device class specification when designing the hardware and the firmware of the device. For example, most keyboards available today in the market work with the device driver provided by Windows 95. You are not required to install any device drivers using a diskette from the keyboard manufacturers. The same scenario applies to USB keyboards. For example, if the keyboard driver is provided by the Windows 95 operating system and you follow the "keyboard" specification when designing your keyboard, you do not have to provide any device drivers to end users. In Chapter 6, we describe in greater detail how to go about designing a device driver. You only need to develop the device driver if the required driver is not bundled with the Windows operating systems you are designing for, or if the device does not follow the device class specifications. You may wish to differentiate your products by adding extra features that are not supported or included in the "generic" device drivers.

5.3 The 8x930Ax Operating Model

To help you understand the operation of the USB firmware, an operating model of a typical USB device is explained in this section. The model is shown in Figure 5-1.

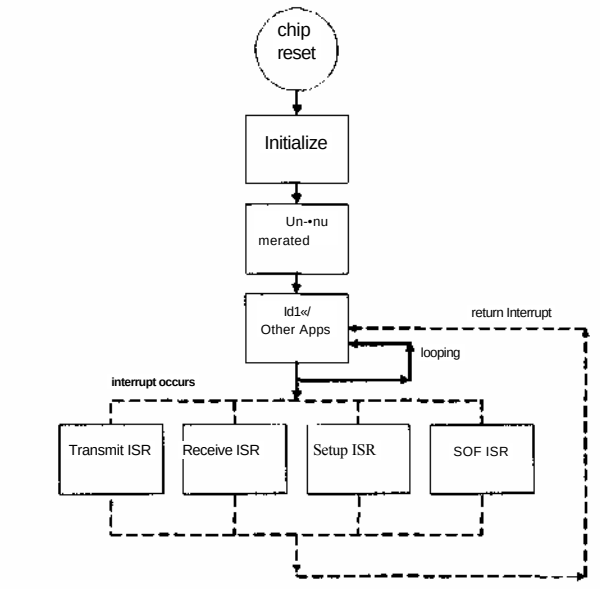


Figure 5-1. USB device program flow

5.3.1 Initialization

After chip reset, the device will execute its initialization routine to initialize the USB modules, peripherals, or user defined flags. The USB Specification allows a device to have multiple interfaces. For example, you can have endpoint 1 configured for bulk transfer, endpoint 2 as interrupt for interface one; and for interface two, you can have endpoint 1 configured as an isochronous endpoint and endpoint 2 as an interrupt endpoint. Depending on which interface we choose, endpoint 1 handles either bulk or isochronous transfer. To keep this operating model simple, let's use only one interface for the USB device.

You will need to prepare your device to perform the various transfer types that are required. Endpoint pair 0 is required to handle control transfer, as dictated by the USB Specification. For Endpoint pairs 1, 2, 3, you can choose to configure them to be control, bulk, interrupt, or isochronous endpoints as required by the device. To configure an endpoint pair to handle control transfer, it is required to set the control endpoint CTLEP, receive endpoint enable RXEPEN, and the transmit endpoint enable TXEPEN bits of the EPCON register for that particular endpoint. To configure a transmit endpoint for isochronous transfer, set the transmit isochronous data TXISO bit in the TXCON register for that

particular endpoint, and so forth. The sample code on the enclosed disk gives an example of how to configure these endpoints.

5.3.2 Un-enumerated Stage

After the initialization of the device, it proceeds to the un-enumerated stage. In this stage, it will perform some non-USB functions or just loop endlessly waiting for a setup token. When a setup token arrives, an interrupt will be generated and the device will execute its interrupt service routine (ISR) to handle the reception of the setup token. After the first setup token, the USB host will issue a series of IN, OUT, and SETUP packets to enumerate the device, and the device is required to respond accordingly. The enumeration process has been discussed in detail in Chapter 2 of this book.

After successful completion of the enumeration process, the device proceeds to the idle stage, looping endlessly or performing some application specified codes, as shown in Figure 5-1. From this stage, it can transit to either (1) transmit, (2) receive, (3) setup, or (4) receive SOF interrupt service routines depending on the application.

5.3.2.1 Transmit Routine (for non-isochronous transfer)

The transmit routine is used to transmit data to the host through non-isochronous endpoints. For example, say the 8x930Ax is in a USB keyboard and endpoint 3 is configured as an interrupt endpoint during enumeration. The device is now in its idle stage, as shown in Figure 5-1. When a key is pressed, an interrupt is generated and the microcontroller activates its scanning sequences to determine which key was pressed, as shown in Figure 5-2. After identifying which key was pressed, the microcontroller will load appropriate data into the FIFO of transmit endpoint 3, and then write a byte count to the byte count register of endpoint 3. After that, it would go back to the idle stage, waiting for the next interrupt.

When an IN token arrives for this endpoint 3, the SIE would automatically transmit the data that was in the FIFO to the host. When the transmission completes, a transmit done interrupt will be generated, as shown in Figure 5-2. The microcontroller will then service this interrupt by checking corresponding flags and status registers to ensure that the transmission was completed successfully. This signifies the completion of the transmit routine, and the device will go back to its idle/application stage, as shown in Figure 5-1.

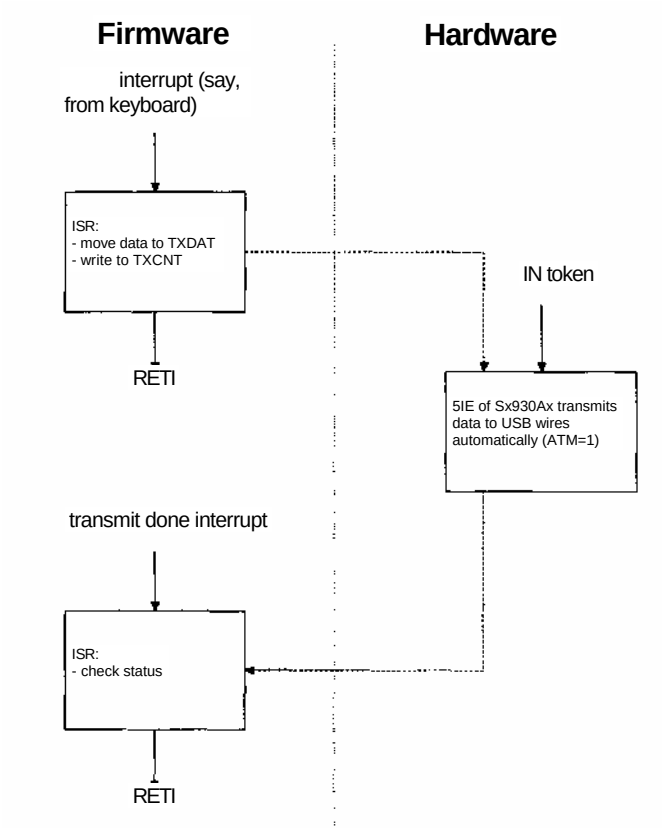


Figure 5-2. Operation of transmit routines

5.3.2.2 Receive Routine (for non-isochronous transfer)

The receive routine is used when receiving data from the host using non-isochronous transfer. For example, assume a USB printer has endpoint 1 configured for bulk transfer in its initialization stage. Note that the maximum packet size for bulk transfer is 64 bytes per transfer as specified by the USB Specification. For bulk transfer, however, it is possible to have multiple transfers within a frame if the bus bandwidth is available. To print a file, the host will send an OUT token to endpoint 1 of the printer. The SIE will automatically receive the data and place it in the receive FIFO of endpoint 1. The SIE will also update the corresponding receive byte count register with the total number of bytes received. When this reception is completed, a receive done interrupt will be generated and the microcontroller comes into action. It will execute

its receive interrupt service routine, as shown in Figure 5-1. First it will check the corresponding flag and status register to ensure the correct reception of the data. After that, it will move the data received for storage, manipulation, or transfer to other printer chips. The process is depicted in Figure 5-3. After this, it will go back into its idle/application stage waiting for another interrupt.

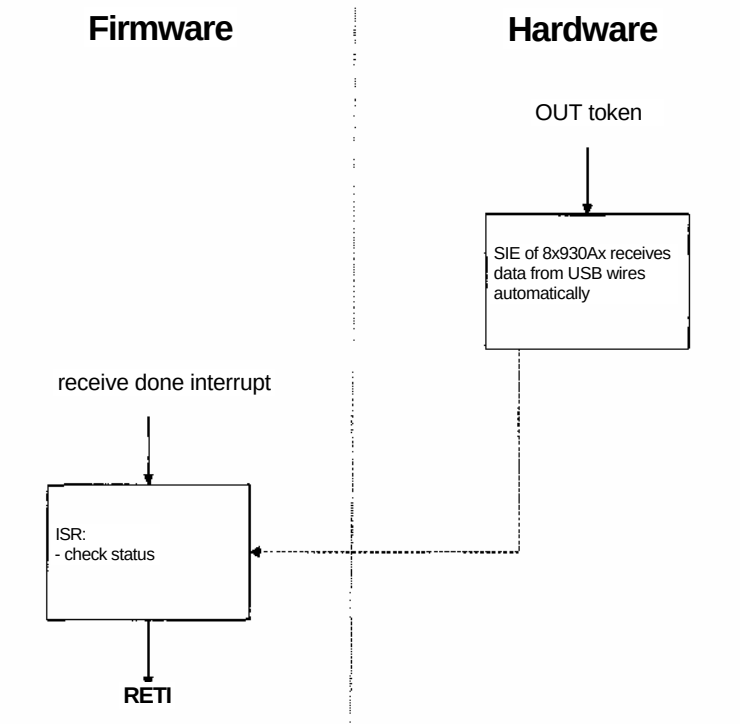


Figure 5-3. Operation of receive routines

5.3.2.3 Setup Routines (for control transfer)

Every USB device is required to have its endpoint 0 configured for control transfer. When a setup token arrives, the device will activate its setup routines. Other endpoints can also be configured for control transfer if required. The setup routines that handle control transfer need special attention because the control transfer is typically completed over more than one USB frame. Pre-defined flags are needed to keep track of which stage the control transfer is in. In a control transfer, the first packet received will be a setup packet. This setup packet has 8 bytes of

data as specified by the USB Specification (Chapter 9 of the USB Specification[11]). The device needs to decode the 8-bytes of data and determine which action is required and whether there will be a data stage. Once the decoding and the setting up of the user flags have been completed, it will go back to its idle stage. It will wait there for the next stage of the control transfer. The next stage of the control transfer will be either a status or data stage (optional) as indicated in the setup packet that was just received. When a control transfer is completed (after status stage), all the user flags that were set need to be cleared and made ready for the next interrupt event. This setup routine is discussed in greater detail in the USB Enumeration Code section.

5.3.2.4 SOF Routines (for control of isochronous transfer)

The host issues an SOF token at the nominal rate of 1.0 ms. An interrupt can be generated (if enabled) when the microcontroller receives the SOF tokens. An SOF interrupt will also be generated even if a valid SOF token is not received due to error. In this case, the interrupt is caused by an "artificial" SOF generated by the frame timer of the 8x930Ax. The SOF routines are used to manage the transfer of isochronous data. Its flow diagram is shown in Figure 5-4.

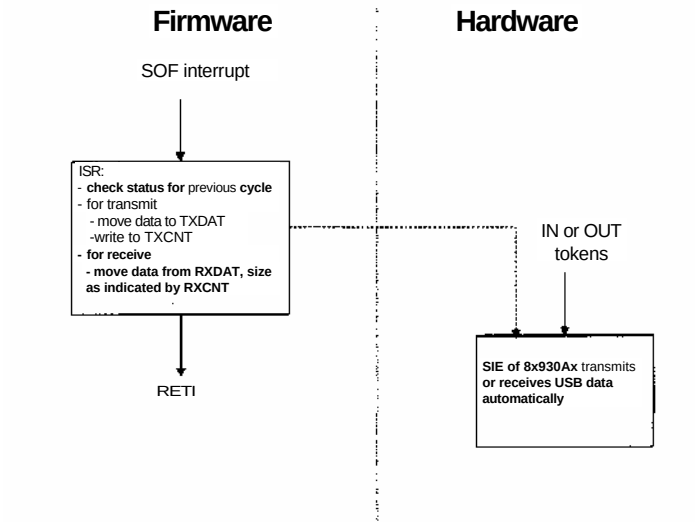


Figure 5-4. Operation of SOF routine

For example, say seven to nine bytes of data are stored every frame into internal RAM of the device from a 64 kbps PCM codec. The SOF interrupt can be used to initiate transfer of these bytes into the transmit FIFO for transmission to the host. The same SOF interrupt can also be used for the transfer of isochronous data from the host. Note that you'll need to use the FIFO in dual packet mode for isochronous transfer. The dual packet mode provides a method to match the rate of voice codec, 64 kbps in this example, to the USB rate of 12 Mbps. The user manual of the 8x930Ax[3] describes the above routines in greater details.

5.4 USB Enumeration Code

A USB enumeration code example, USB_ENUM.ASM, is included on the enclosed diskette and listed in Appendix A for your reference. It enables the 8x930Ax to perform the USB enumeration when a device first attaches to the host. Since the enumeration is mandatory, the code example can be re-used. It should be modified and integrated with other firmware modules to complete the firmware for your device.

In this section, the operating model of the enumeration code is explained. It can be used as an example for handling USB control transfers described in earlier sections.

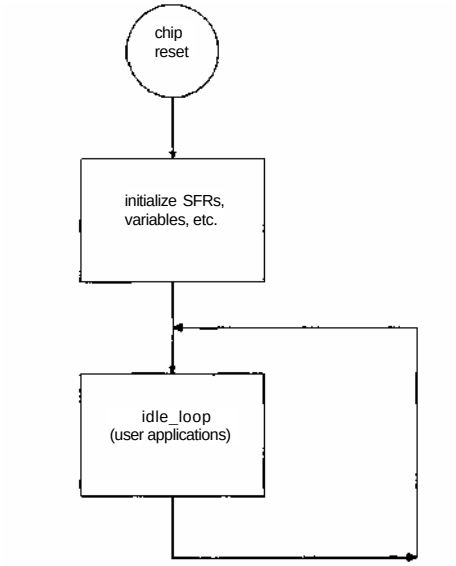


Figure 5-5. Flow diagram of the enclosed USB_ENUM.ASM

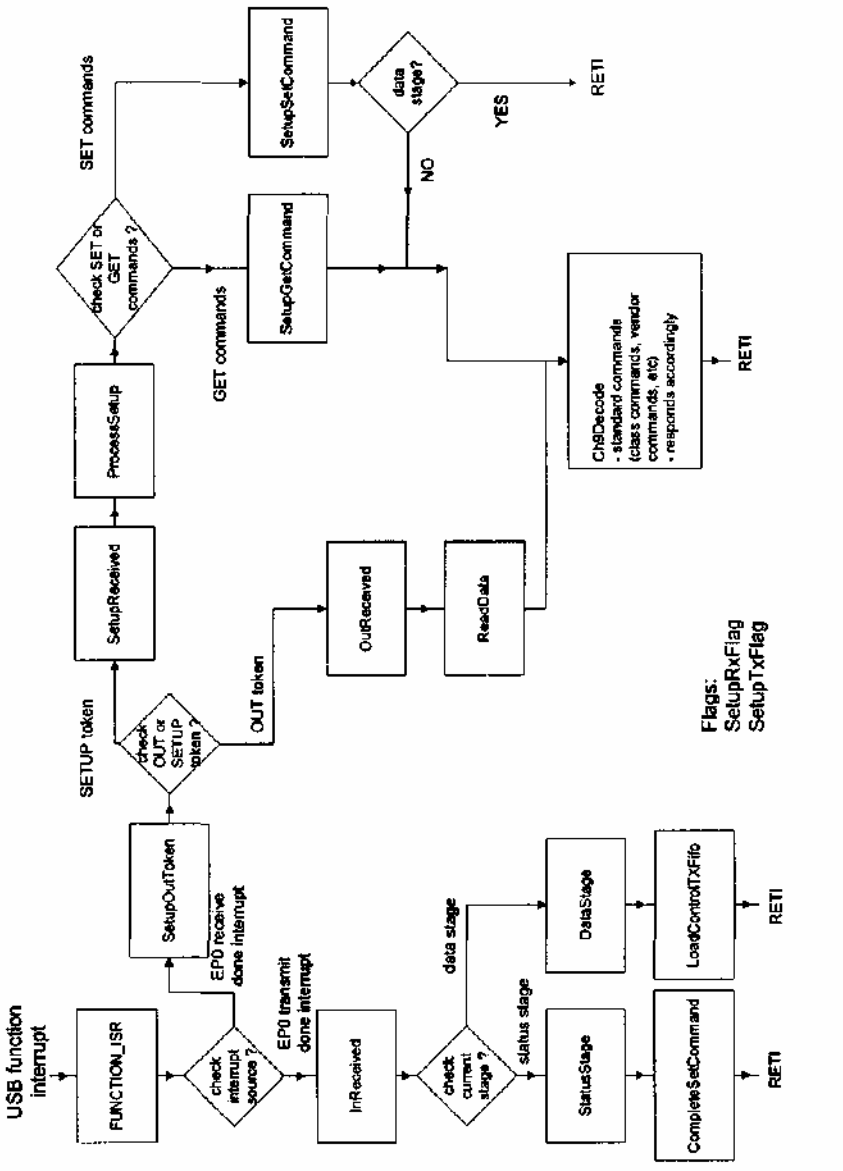


Figure 5-6. Flow diagram of the USB_ENUM.ASM

Figure 5-5 and Figure 5-6 show the flow diagrams of the enumeration code, USB_ENUM.ASM. The complete listing of the code is in Appendix A. This USB_ENUM.ASM is ready to be assembled and programmed

into a 32 Kbyte Flash or EPROM. It was tested with the PLC assembler (see Chapter 3 for details), and you may need to make minor modifications for other assemblers.

The code starts with the initialization of SFRs, user flags, and the stack pointer. It defines two important user flags, SetupRxFlag and SetupTxFlag, to keep track of the different stages of the control transfer; either the setup, data, or status stage.

After the initialization, the code goes into an idle loop. Note that port 1 of the 8x930Ax is used to control the LEDs of the Evaluation Board (comes with the Evaluation Kit). In this loop, you can add some application-specific firmware for your device. Or alternatively, you can design your application firmware to be interrupt driven. Consider a joystick for example: when you press a "fire" button, the hardware can be designed to cause an external interrupt to the microcontroller. The firmware will then branch to execute the ISR of this external interrupt and send appropriate USB packets to the host to inform it of the status change of the "fire" button.

Back to the USB_ENUM.ASM example; after initialization, the firmware will be looping in the idle loop waiting for an interrupt, as shown in Figure 5-5. When a USB interrupt arrives, a FUNCTION_ISR ISR will be executed, as shown in

Figure 5-6. This ISR will first check the source of the interrupt and if the source is caused by transmit endpoint 0, it will branch to an InReceived subroutine. If the interrupt is caused by receive endpoint 0, it branches to a SetupOutToken subroutine. In the SetupOutToken subroutine, it needs to check the data received for whether the token received was a SETUP or IN token. The firmware decodes the data received (refer to Chapter 9 of the USB Specification[1]) and processes the data accordingly. After various steps, the code will perform a Ch9Decode subroutine where it responds appropriately to the token received. This subroutine can be further expanded to handle other types of commands, like class commands or vendor specified commands if needed or desired.

On the InReceived subroutine side, it will first check the stage of the control transfer by using the user flags, SetupRxFlag and SetupTxFlag. The code can then respond accordingly. You'll want to note one special case of control transfer, i.e., SET ADDRESS, where the device's address is to be set (by writing to the FADDR SFR) only after the status stage. This is different from other control transfers because the executions of the commands are completed after the status stage, rather than before.

6. Step 5: Develop the USB Driver

This chapter describes how to go about developing the USB driver. But first, an introduction to the concept of device class specifications and device class drivers.

6.1 Device Class Specification and Driver

What is a USB device class? In the context of USB, a class is a grouping of devices that have certain attributes in common. Grouping these devices together allows the development of a class specified host-based driver. This class specification serves as a framework defining the operation of devices belonging to that class. This allows the development of class drivers from parties other than the manufacturer of the device itself. For example, a keyboard manufacturer does not need to include a driver if the "keyboard" class driver is bundled with the operating system and if he follows the device class specifications. In fact, keyboards belong to the HID (human input device) class of USB devices. Thus, the hardware vendors can concentrate on developing their hardware while other software/driver vendors can design the drivers.

All USB devices must meet the USB specifications. In addition to this, a class specified device must follow the class requirements exactly. The class specifications dictate how a USB device communicates with the driver. Figure 6-1 illustrates the relationship between a USB device and the host. In the bottom layer, the hardware of the USB device communicates through the actual USB wires to the host hardware. In the middle layer, the USB system software uses USB specifications to communicate with the USB device. In the highest level, the client driver interacts with the device's functions as specified by the class specification.

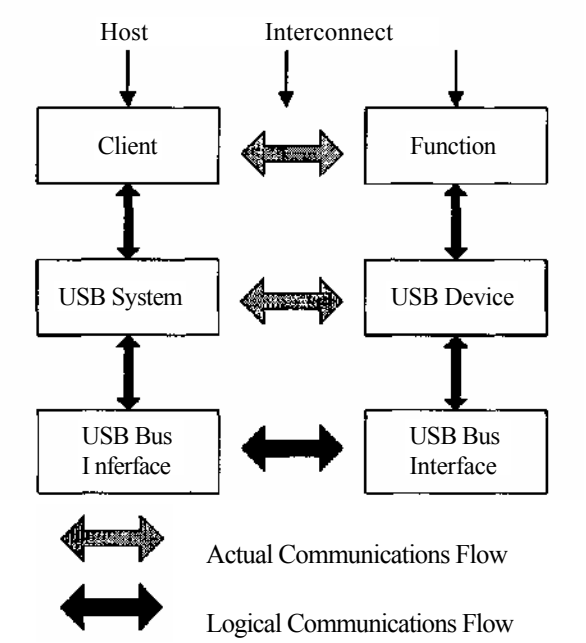


Figure 6-1. Relationship of the USB host and the USB device

There are currently 11 classes defined in USB: common, audio, communications, hub, human interface, image, PC legacy, physical interface, power, printer, and storage. Some specifications are in Rev 0.7 (meaning the first draft incorporating initial round of comments from the members of the device working group). Some specifications are in Rev 0.9 (meaning all revisions are accepted by the working group and the specification is placed on the web sites for public comment) or Rev 1.0 (meaning it is ready for implementation by device vendors, and will be posted on web sites). The device working groups get together every 4-6 weeks for a face-to-face meeting. The members of the groups come from various software and hardware companies. For more information on the device working groups, please contact the USB-IF, or the USB web site at http: \\www.usb.org.

6.2 Overview of WDM

Microsoft publicly announced the Win32 Driver Model (WDM) at WinHEC 96. It is intended to provide a common driver architecture for Windows NT, Windows 95, and future Windows operating systems. It is

based on existing Windows NT driver models with additional Plug and Play and power management support. Thus, using the WDM, a driver developed for Windows NT can be used in the other Windows environments as well. The WDM supports new bus architectures, like USB and the IEEE 1394 Firewire protocol.

Figure 6-2 shows a simplified view of the WDM software stack. It is a more detailed view of the host side of Figure 6-1. The class device drivers are just below the "user mode" API layer. While the device class drivers are specified by the device working groups, individual manufacturers can choose to develop their own mini-driver to differentiate their products or to be in the market before the class drivers are available. All these class drivers and mini-drivers interact to a USB driver. In the layer below the USB driver, there are UHCI (Universal Host Controller Interface) and OHCI (Open Host Controller Interface) drivers which interact with the host controller hardware. Both the OHCI and UHCI host controllers are supported by the WDM architecture. The host controllers will then communicate to the USB devices via the Serial Interface Engine (SIE) and the USB wires.

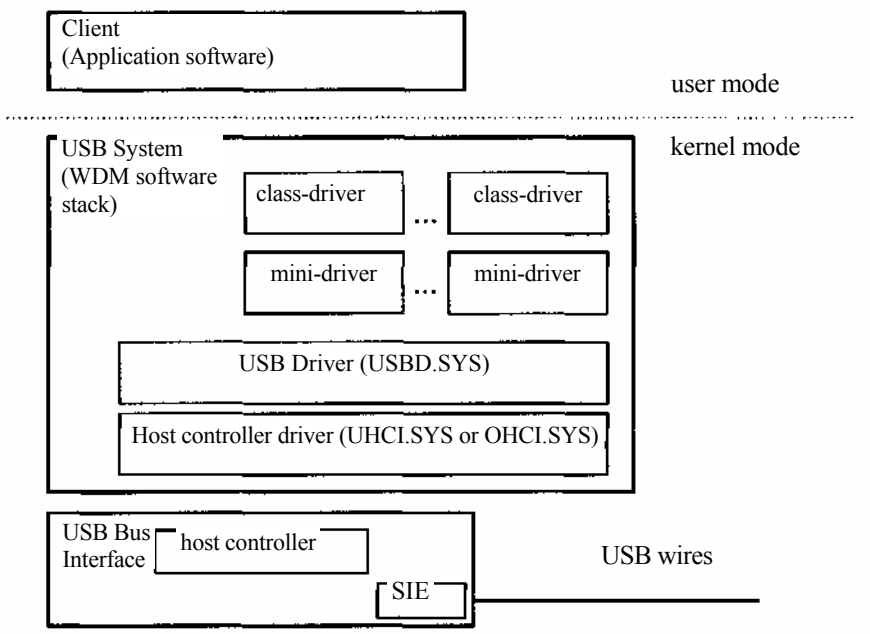


Figure 6-2. WDM software stack

6.3 Windows NT System Overview

Since the WDM is based on the Windows NT driver model, it is essential that we understand the Windows NT operating system and its system architecture[5].

Windows NT is considered to be Microsoft's highly portable, next generation, 32-bit Windows multitasking operating system. It divides the work that needs to be done among its processes. Each process has its resources, like memory, system resources, and at least one thread of execution. A thread is an executable unit within an NT process. The operating system executes one thread for a short time and then switches to another thread, running each thread in turn. This gives the multitasking "illusion" of the operating system to the users.

The Windows NT design was based on the combination of three models;

- **Client/server and layered model**

It uses the client/server model to provide multiple operating environments to its users (e.g. Windows, MS-DOS, OS/2, and POSIX). Based on this model, the operating system is divided into several processes, each of which implements a single set of services. Each server runs in user mode, executing a loop that checks if a client has requested its service. The client, which can be either another operating system or an application program, requests a service by sending a message to the server. An operating system running in a kernel mode delivers the message to the server. After the server performs the operation, the kernel returns the results to the client in another message, as shown in Figure 6-3.

The operating system code runs in a privileged mode (referred to as kernel mode) and has access to the system data and hardware. The applications, however, run in a non-privileged mode (referred to as user mode) with a limited set of interfaces and limited access to system data. When a user mode program calls a system service, the processor traps the call and then switches the calling thread to kernel mode. When the system service completes, the operating system switches the thread back to user mode.

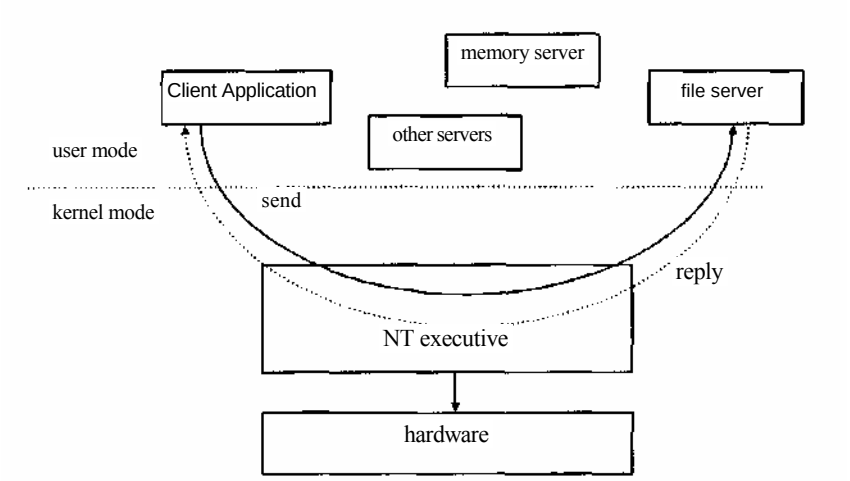


Figure 6-3. A client/server operating system

The client/server approach results in a more robust operating system. This is because each server runs in a separate user-mode process; a single server can fail and be restarted without crashing or corrupting the rest of the operating system. In simple terms, users only need to "end-task"/"kill" the application that "hangs" and continue with their work in other applications on Windows NT systems. In some other operating systems, users typically need to reboot the entire system if one application "hangs".

The Windows NT operating system model is also based on a layered model in its executive I/O system, as well as in the lowest portion of NT executive: the NT kernel and the hardware abstraction layer (HAL). The kernel mode of Windows NT is called the NT executive. It consists of various components that implement virtual memory management, object (resource) management, I/O and file systems, inter-process communication, and portions of the security system. All other components of the Windows NT executive are layered on the NT kernel and HAL components. The NT kernel performs low level operating system functions like thread scheduling, interrupt handling, etc. The HAL manipulates the hardware directly.

- **Object model**

Windows NT uses objects to represent and manage system resources and dispense them to users. Any system resource that can be shared by more than one process, including files, shared memory, and physical devices, is implemented as an object and manipulated by using object services. In this fashion, any changes or additions to hardware, for

example, require only code modifications in the object and the services layer that operate the object; other system codes remain the same. This makes the system codes easier to maintain.

The operating system that uses the object model has other advantages as well:

=> The operating system accesses and manipulates its resources uniformly.

=> It simplifies the implementation of security because all objects are protected in the same way.

=> Objects provide a convenient and uniform paradigm for sharing resources between two or more processes.

- **Symmetric multiprocessing**

This model allows Windows NT to achieve maximum performance in a multi-processor computing environment.

6.3.1 Windows NT Structure

The structure of Windows NT can be divided into two portions: the user-mode (the Windows NT protected subsystems) and the kernel-mode (the NT Executive), as shown in Figure 6-4. The Windows NT servers run in user-mode and are called the protected subsystem because each one resides in a separate process whose memory is protected from other processes by the Windows NT executive's virtual memory system. These subsystems communicate by passing messages (via the NT executive).

Each server or protected subsystem provides an API that programs can call. When an application (or another server calls an API routine), a message is sent to the server that implements the API routine through the NT executive's local procedure call (LPC) facility. The LPC is a message-passing mechanism used by Windows NT. The server replies by sending a message back to the caller.

There are two types of Windows NT protected subsystems: environment subsystems and integral subsystems. The environment subsystem provides an API specific to an operating system, such as OS/2, POSIX, or MS-DOS as shown in Figure 6-4. The most important environment subsystem of Windows NT is the Win32 subsystem. This subsystem makes the Microsoft's 32-bit Windows API available to application programs. The Win32 environment also provides Windows NT's graphical user interface and controls all user input and application output. The integral subsystems are servers that perform important

operating system tasks such as security subsystem, networking subsystem, etc.

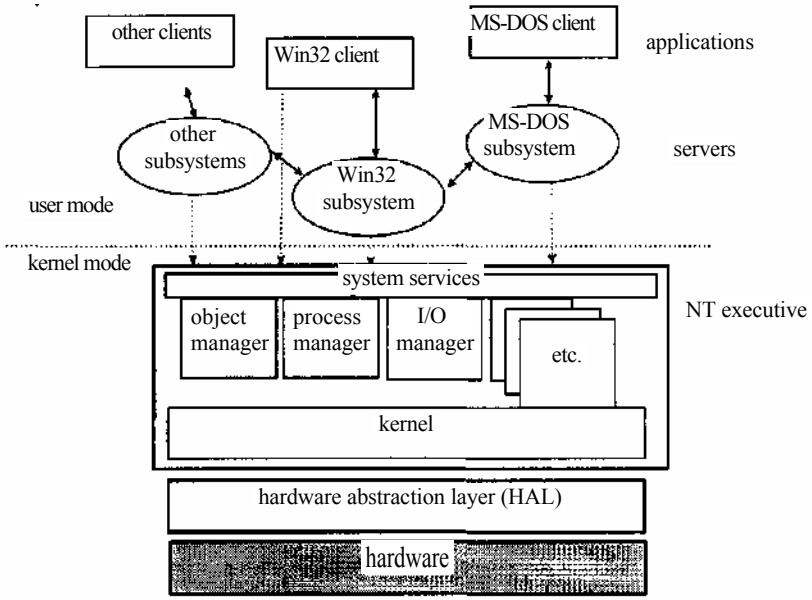


Figure 6-4. Windows NT block diagram

The NT Executive is the kernel-mode portion of the Windows NT system. It consists of the various components listed below. Each of these components implements two sets of functions: system services, which environment subsystems and other executive components can call upon, and internal routines, which are only available to components within the NT executive. The responsibilities of the executive components are:

- Object manager -Creates, manages, and removes NT executive objects which represent the system resources.
- Security reference monitor - Enforces security policies on local computers.
- Process manager - Creates and terminates processes and threads.
- Local procedure call (LPC) facility - Provides a mechanism to pass messages between a client process and a server process on the same computer.
- Virtual Memory (VM) manager - Implements virtual memory, a memory management scheme that provides a large, private address space for each process.
- NT Kernel - Provides operating system functions like interrupt response, thread scheduling, multi-processors synchronization,

etc. It also supplies a set of elemental objects and interfaces which the other NT executive uses.

- I/O system - Consists of a group of components responsible for processing the inputs and outputs of a variety of devices. It includes the following sub-components:
  => I/O manager - implements input/output facilities and establishes a model for the NT executive I/O
  => File systems
  => Network re-director and network server
  => NT executive device driver which directly manipulates hardware.
  => Cache manager
- Hardware abstraction layer (HAL) - A layer of code between the NT executive and the hardware platform on which Windows NT is running.

Developing a USB driver requires focusing on the I/O System of the Windows NT operating system.

6.4 Windows NT I/O System Overview

The NT executive's I/O system is a collection of operating system codes that accept I/O requests from the user- or kernel-mode processes and delivers them to I/O devices. In the context of USB, the I/O system can be considered an interface between system processes and USB devices.

Figure 6-5 depicts a simplified view of the Windows NT I/O system structure. Every I/O request is represented by an I/O request packet (IRP) which travels from one I/O system component to another. An IRP is a data structure that controls how the I/O operation is processed at each stage along the way.

The I/O manager, as shown in Figure 6-5, defines an orderly framework within which I/O requests are delivered to file systems and device drivers. The I/O manager creates an IRP that represents each I/O operation, passes the IRP to the correct drivers, and disposes of the packet when the I/O operation is completed. A driver, on the other hand, receives the IRP and performs the operation as specified by the IRP, and then either passes it back to the I/O manager after completion or passes it on to another driver (via the I/O manager) for further processing.

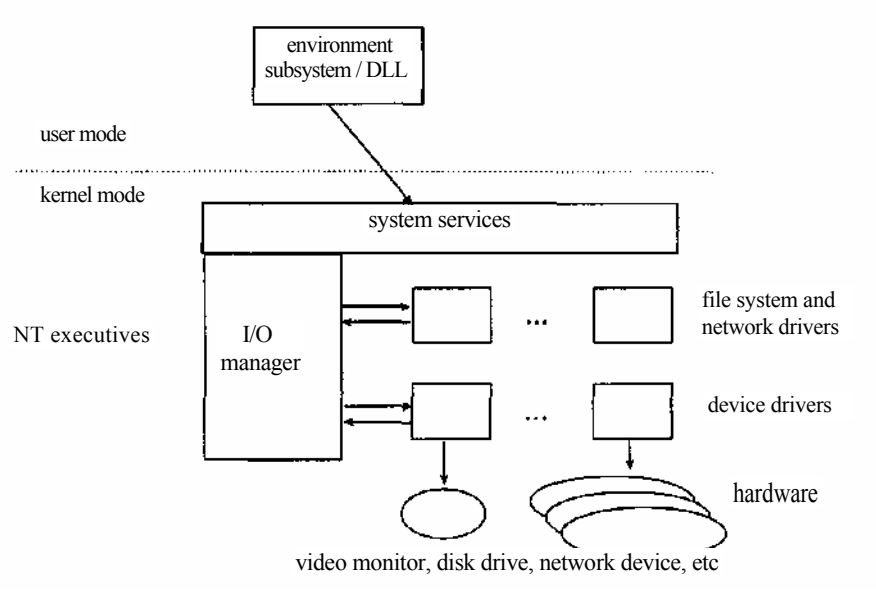


Figure 6-5. Windows NT I/O system

The I/O manager provides functions common to all drivers, like a function that allows one driver to call another. The driver can call those functions to carry out their I/O processing. The I/O manager also manages the buffer for I/O requests, provides time-out support for drivers, and provides flexible I/O facilities to allow environment subsystems to implement their respective I/O APIs.

6.4.1 NT Object Model

In Windows NT, any I/O device is treated as if it was a file, and is referred to as a virtual file. Windows NT programs perform I/O on virtual files, manipulating them using file handles. Within the NT executive, the file handles are referred to as file objects. Application program can call the NT file object services to read from a file, write to a file, or perform other operations. The I/O manager dynamically directs these virtual file requests to real files or physical devices. Applications open a file by using a standard library function, and a handle to an NT executive file object is returned to the application.

6.5 WDM Device Driver Example

To build WDM drivers, we need both the Windows NT DDK and the WDM DDK with support for USB, as mentioned in Chapter 3. The Windows NT driver model is documented in the Windows NT DDK. The WDM USB DDK provides a set of header files, software routines, and libraries, in addition to the Windows NT DDK to build the WDM drivers for USB devices.

A WDM driver example is provided in the USB program examples on the enclosed diskette and its usage is discussed in the Chapter 8. The following section should give you the basic overview of a typical code structure for a WDM driver.

As in any other Windows NT driver, the "DriverEntry(...)" routine is invoked when the driver is loaded. It can be treated as the "starting point" of the driver code as the following example illustrates.

```
NTSTATUS
DriverEntry(IN PDRIVER_OBJECT pDriverObject, IN PUNICODE_STRING
RegistryPath)
/* Purpose: Main entry point for the driver, wilt be invoked once when the driver is
loaded */

{

 PDEVICE_OBJECT    pDeviceObject; NTSTA
 TUS ntStatus = STA TUS_SUCCESS ;
 UNICODE_STR1NG objectName;

 //Set up dispatch entry points for the driver.
 pDriverObject->MajorFunction[IRP_MJ_CREATE]        = Test_Create ;
 pDriverObject->MajorFunction[IRP_MJ_CLOSE]      = Test_Create;

pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]      =
Test_ProcessIOCTL;
pDriverObject->MajorFunction[lRP_MJ_PNP_POWER] = Test_Dispatch;

 pDriverObject-> DriverUnload              = Test_Unload;
 pDriverObject-> Driver Extension->AddDevice      = Test_PnPAddDevice;

 return    ntStatus; }
```

The "DriverEntry(...)" routine dictates the initialization of the driver using "pDriverObject->MajorFunction[lRP.MLCREATE]", "[IRP_MJ__CLOSE]", and "pDriverObject->DriverUnload". For a WDM driver, it also considers the "[IRP_MJ_DEVICE_CONTROL]," "'[IRP_MJ_PNP_POWER]," and the "pDriverObject->DriverExtension->AddDevice".

The "pDriverObject->DriverExtension->AddDevice" is directed to the Test_PnPAddDevice(...) routine for this case, where it creates the file handler to this device driver. This is invoked when a USB device is plugged into a host PC. Among other things, the Test_PnPAddDevice(...) routine creates a driver object (using IoCreateDevice(...)) and sets up a symbolic link for the driver (using IoCreateSymbolicLink(...)). The creation of the symbolic link enables user mode applications to open and operate on the driver object.

The "[IRP_MJ_PNP_POWER]" has several MinorFunctions (MN) defined and it also provides for the Plug and Play and power management schemes for the devices. For example, one of the MinorFunctions, "[IRP_MN_START_DEVICE]" will get all the descriptors (device, configuration, and string) of the USB devices and set the configuration for each device.

The "[IRP_MJ_DEVICE_CONTROL]" provides device control functions to the devices as required by the application. In the example provided, the "get command - one" menu is activated in the application program, causing the application thread to issue an IRP request, IRP_Test_ GET_CONFIGURATION_DESCRIPTOR. The request will eventually cause the activation of the Test_ProcessIOCTL routine, as directed by the "[IRP_MJ_DEVICE_CONTROL]". The Test.ProcessIOCTL routine will then further decode the IRP.

After decoding the IRP command in the Test_ProcessIOCTL routine, a USB Request Block (URB) will be made (in the Test_ GetConfigDescriptor(...) routine). This is followed by the Test_ CallUSBD(...) routine to perform the USBD call and carry out the corresponding USB functions. In the ex-ample below, the Test_ProcessIOCTL(...), Test_GetConfigDescriptor(...), UsbBuildGetDescriptorRequest(...), and Test_CallUSBD(...) routines will be called (in the sequence as stated) when the IRP_Test_GET_ CONFIGURATION_DESCRIPTOR IRP is made by the application software.

```
/* example to handle the IRP_MJ_DEVICE_CONTROL */
/* DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]==Test_ProcessIOCTL;
 */

//================================================

NTSTATUS Test_ProcessIOCTL          (INPDEVICE_OBJECT DeviceObject, IN PIRP
Irp)
{
   PIO_STACK_LOCATION   irpStack;
   PVOID ioBuffer; ULONG
   inputBufferLength; ULONG
   outputBufferLength;
   PDEVICE_EXTENSION   device Extension;
   ULONG ioControlCode; NTSTATUS
   ntStatus; ULONG length; PUCHAR pch;

   // Get a pointer to the current location in the Irp.
   irpStack = loGetCurrentlrpStackLocation (Irp);
   Irp->IoStatus.Status = STATUS_SUCCESS;
   lrp->IoStatus.Information = 0;

   //Get a pointer to the device extension
   deviceExtension = DeviceObject->DeviceExtension;
   ioBuffer          = Irp->Associatedlrp. SystemBuffer;
   inputBufferLength = irpStack~>Parameters.DeviceloControl. InputBufferLength;
   outputBufferLength = irpStack-> Parameters. DeviceloControl.OutputBufferLength;
   ioControlCode = irpStack->Parameters.DevicehControl. IoControlCode;

switch (ioControlCode) {

          /* ... other cases ... */
          case IRP_Test_GET_CONFIGURA TION_DESCR1PTOR:
                    length = Test GetConfigDescriptor (DeviceObject, ioBuffer,
                        outputBufferLength);
                    Irp->loStatus. Information = length;
                    Irp->IoStatus.Status = STATUS_SUCCESS;
          break;
          /* ... other cases... */
          default:
               Irp->IoStatus.Status = STATUS_INVALID_PARAMETER;
   }
ntStatus = Irp~>IoStatus.Status; hCompleteRequest
(Irp, IO_NO_INCREMENT); return ntStatus; }
```

```
//=============================================================
/* example of a URB */

ULONG Test_GetConfigDescriptor (IN P DEVICE_OBJECTDeviceObject,
                                PVOID          pvOutputBuffer, ULONG          ul Length)
{
   PDEVlCEj;XTENSION    device Extension = NULL;
   NTSTATUS          ntStatus          = STATUS_SUCC'ESS;
   PURB              urb          = NULL;
   ULONG             length       = 0;

   deviceExtension = DeviceObject~>DeviceExtension;
   urb = ExAlhcatePool(NonPagedPool,
                sizeof(struct _URB CONTROL_DESCRIPTOR_REQUEST)); if
   (urb) {
         if (pvOutputBuffer) {
               UsbBuildGetDescriptorRequest(urb,
                  (USHORT) sizeof (struct _URB_CONTROL_DESCRlPTOR__REQUEST),
                  USB CONFIGURATION_DESCRIPTOR_TYPE, //descriptor type
                  0,                                 //index
                        0,                           //language ID
                       pvOutputBuffer,               //transfer buffer
                       NULL,                      I/MDL
                       ulLength,                  //buffer length
                       NULL);                     //link
      ntStatus = Test_CaIlUSBD(DeviceObject, urb); }
      else {
         ntStatus = STATUS_NO_MEMORY;
      }
      length = urb->UrbControlDescriptorRequest.TransferBufferLength;
      Ex Free Pool (urb);
      } else {
      ntStatus = STATUS_NO_MEMORY;
   }
   return length;
}
```

The management of the IRPs and URBs thread scheduling is performed by the Windows NT operating system. Figure 6-6 depicts the high level program flow of this operation.

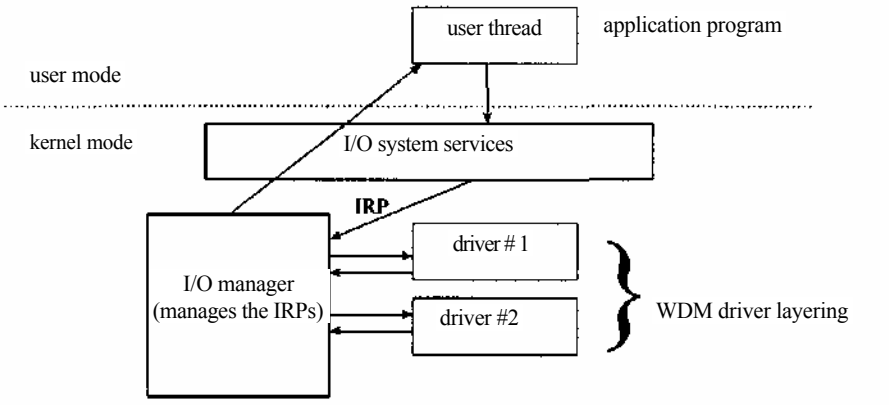


Figure 6-6. An I/O request example

6.6 Loading the WDM Drivers

When we plug in a USB device to the USB host PC, the host will enumerate the device. Among other things, the device descriptor information will be gathered by the host. The Vendor identification information, *idVendor* (assigned by the USB-IF), Product identification information, *idProduct* (assigned by the device manufacturer), and Device release number, *bcdDevice* (assigned by the device manufacturer), of this descriptor (refer to Chapter 9.6.1 of the USB specification[1]) will then be used by the host to determine which driver to load. The synthesis of these values is used to match the Hardware ID in an INF file (Windows NT Information file). When a match is found, the INF file with the matching Hardware ID is used to load the driver.

The Hardware ID in the INF file is described in the "device description" entry. An example of an INF file to load the driver uhcd.sys for the device (one of the devices that the INF file supports) with

idVendor = 8086 (hex)

idProduct = 7020 (hex)

is shown in the following code (Please refer to [Manufacturer] and [Intel.Section] of the entries):

```
; an INF file example:
; USB.INF - This file contains descriptions of all the HDC (USB controller)
; *** Created 10/2/95 (Creation Date)

[Version]
signature="$CHICAGO$"
Class=USB
ClassGUID={36FC9E60-C465-JJCF-8056-444553540000}
Provider=%Msft%
LayoutFile=LA YOUT.INF

[DestinationDirs]
DefaultDestDir=11       ;
LDID_SYS

; ================== Class Sections ===== ==============
[Classlnstall]
Addreg=USBClassReg

[USBClassReg]
HKR,,,, %USBClassName%
HKR,,Icon,,-20

; ================ Table of content ====================
[Manufacturer]
%Generic. Mfg%=Generic, Section
%GenericHub.Mfg%=GenericHub.Section
%Compaq, Mfg%=Compaq.Section %lntel.
Mfg%=Intel. Section %Philips.
Mfg%=Philips.Section %NEC.
Mfg%=NEC.Section %PLX. Mfg%=PLX.
Section % VIA. Mfg%= VIA.Section

; ================= Generic ========================
[Generic. Section]
%PCI\CC_OC0310. DeviceDesc%=OpenHCD. Dev, PCI\CC_0C0310
%PCI\CC_OC0300.DeviceDesc%=UniversalHCD.Dev,PCI\CC_0C0300
%USB\ROOT_HUB.DeviceDesc%=StandardHub.Dev,USB\ROOT_HUB



;-----------------------------------------------------------------------
[OpenHCD.DEV]
AddReg=OpenHCD.AddReg, USB.AddReg
```

```
CopyFiles=OpenHCD. CopyFiles, USB. CopyFiles

[OpenHCD.AddReg]
HKR,, NTMPDriver,, openhci. sys
HKR^EnumPropPages,, "sysclass.dll, USBEnumPropPages"

[OpenHCD, CopyFiles]
openhcisys


[UniversalHCD.Dev]
AddReg=UniversalHCD.AddReg, USB.AddReg
CopyFiles=UniversalHCD. CopyFiles, USB. CopyFiles

[UniversalHCD.AddReg]
HKR,,NTMPDriver,,uhcd.sys
HKR,,EnumPropPages,, "sysclass.dll, USBEnumPropPages"

[UniversalHCD. CopyFiles]
uhcd.sys


[StandardHub.Dev]
AddReg=Hub.AddReg
CopyFiles=Hub. CopyFiles

;==================Compaq====================
[Compaq. Section]
%PCI\VEN_OE11&DEV_A0F8.DeviceDesc%=OpenHCD.Dev,PCI\VEN_OEl
l
&DEV_A0F8

;==================Intel====================
[Intel.Section]
%PCI\ VEN_8086&DE V_7020. DeviceDesc%=UniversalHCD. Dev, PCl\
VEN_80
86&DEV_7020
%USB\VID_8086&PID_9303.DeviceDesc%=StandardHub.Dev, USB\ VID_808
6&PID_9303

;==================Philips====================
[Philips.Section]
%USB\VID_0471&PID_0201.DeviceDesc%=StandardHub.Dev,USB\VlD_047
1&PID_0201
```

```
;====================== NEC======================
[NEC. Sect ion]
%USB\ VlD_0409&PID_55AA.DeviceDesc%=StandardHub.Dev, USB\ VID_040
9&PID_55AA

; ================ Generic HUB ======================
[GenericHub. Section]
%USB\CLASS_09&SUBCLASS_01.DeviceDesc%=StandardHub.Dev,USB\CLA
SS_09&SUBCLASS_01
%USB\CLASS_09,DeviceDesc%=StandardHub.Dev,USB\CLASS_09


;====================== PLX======================
[PLX.Section]
%PCI\ VEN_10B5&DE Y_9060. DeviceDesc% =PLX. Dev, PCI\ VEN_10B5&DE V
_9060

[PLX.Dev]
AddReg=PLX.AddReg, USB.AddReg
CopyFiles=PLX. CopyFiles

[PLX.AddReg]
HKR,, NTMPDriver, .pcspeak. sys

[PLX.CopyFiles]
pc speak, sys


;====================== Via======================
[VIA.Section]
%PC1\ VEN_1106&DEV_3038. DeviceDesc%=UniversalHCD. Dev, PCI\ VEN_11
06&DEV_3038

;====================== Global======================
[USB.AddReg]
HKR,,DevLoader, *NTKERN

[USB. Copy Files]
usbdsys

[HUB.AddReg] HKR,,DevLoader,,
*NTKERN
HKR>tNTMPDriver,,usbhub.sys
```

[HUB.CopyFiles]

[ControlFlags]

[DestinationDirs]
DefaultDestDir = 11 ; System directory

[Strings]
Msft="Microsoft"
USBClassName= "Universal serial bus controller"

Generic. Mfg= "(Standard USB Host Controller) "
PCI\CC_0C0310.DeviceDesc="Standard OpenHCD USB Host Controller"
PCI\CC_0C0300.DeviceDesc="Standard Universal PCI to USB Host Controller" USB\ROOT_HUB.DeviceDesc="USB Root Hub"

Compaq. Mfg= "Compaq "
PCI\VEN_0E11&DEV_A0F8.DeviceDesc="Compaq PCI to USB OpenHost Controller"

InteLMfg="lntel"
PCI\ VEN_8086&DEV_7020.DeviceDesc="Intel 8237ISB PCI to USB Universal Host Controller"
USB\VID_8086&PID_9303.DeviceDesc="Intel 82930HX Hub with 3 downstream ports "

Philips. Mfg= "Philips "
USB\VID_0471&PID_020LDeviceDesc="Philips USB Hub"

NEC.Mfg="NEC"
USB\VID_0409&PID_55AA.DeviceDesc="NEC USB Hub"

GenericHub.Mfg="(Generic USB Hub)"
USB\CLASS_09&SUBCLASS_01.DeviceDesc="Generic USB Hub"
USB\CLASS_09.DeviceDesc="Generic USB Hub"

PLX. Mfg= "PLX Technology "
PCI\VEN_10B5&DEV_9060.DeviceDesc="PLX USB Test Board"

VIA.Mfg="VIA Technologies"
PCI\VEN_1106&DEV_3038.DeviceDesc="VIA VT83C572/VT82C586 PCI to USB Universal Host Controller"

After matching the *idVendor* and *idProduct* pair, the driver uhcd.sys will be loaded (see the "[UniversalHCD.Dev]" and the "[UniversalHCD.CopyFiles]" sections above.

Thus, to ensure the proper loading of a driver, device vendors should follow the following guidelines when defining the descriptor of their devices:

- Do not use the same values for the *idVendor* and *idProduct* pair for different devices. The *idVendor* and *idProduct* pair is used to differentiate devices from one another.
- Use the *bcdDevice* values to differentiate revisions of what is essentially the same device to associate different versions of the driver.

If the device has multi-configuration, the *bNumConfiguration* from the device descriptor will also be used. The white paper "USB Plug and Play IDs and Selecting Device Drivers to Load" located at http://www.microsoft.com/HWDEV/busbios/usbpnp.htm, also describes the drivers loading process.

7. Step 6: Developing Host Application Software

The host application software provides a GUI to end-users and communicates to the WDM drivers of the USB devices. In this chapter, we describe a way to develop host application software using Microsoft Visual C++ 4.0 and the Microsoft Developer Studio. Sample application software is also included on the enclosed diskette and in Appendix C for your reference.

7.1 Microsoft Developer Studio

The main part of the Visual C++ package is the Developer Studio. It is an Integrated Development Environment for editing, compiling, and debugging programs in a single environment. The Developer Studio includes several tools to create the application software. Among other things, it has:

- a C/C++ compiler
- an integrated editor
- a Component Gallery to store, organize, and reuse our software routines, and
- an integrated debugger for steping through and checking the source code.

The Microsoft Developer Studio also includes five "wizards" that can be used to simplify the development of Windows programs. The five "wizards" are:

AppWizard - Used to create the basic outline of a Windows program. It supports three program types: 1) Single Document and 2) Multiple Document applications based on the Document/View architecture, and 3) dialog box-based programs, in which a dialog box serves as the main window for the application. The basic idea of the Document/View architecture is to separate the data handling classes from the classes that handle the user interface. The view class is responsible to provide a "view port" where we can manage the document. The document class is responsible for controlling all the data[6].

MFC DLL AppWizard - Used to create DLLs that use the Microsoft Foundation Classes (MFC) library. Three types of DLL projects are supported: regular DLLs that use static linking to MFC, regular DLLs dynamically linked to MFC, and extension DLLs, that are always dynamically linked to MFC.

Custom AppWizard - Used to create customized AppWizards from scratch based on an existing AppWizard type or based on an existing project.

ClassWizard - Used to define the classes used in a program created with AppWizard. You can add classes and functions that control how messages received by each class are handled. It also helps to manage controls that are contained in dialog boxes.

OLE (Object Linking and Embedding) ControlWizard - Used to create the basic framework of an OLE control. An OLE control is a customized control that supports a defined set of interfaces and is used as a reusable component. OLE controls replace the Visual Basic Controls of VBXs that were used in the 16-bit version of the Windows operating system.

7.2 Develop Application Software using AppWizard

In this section, we describe a way to build a skeleton of the application software using the AppWizard portion of the Microsoft Developer Studio. After the installation of Microsoft Visual C++ and the Developer Studio, activate the Developer Studio application icon. To start a new application development, choose File - New from the application menu. Then select the Project Workspace from the options given, as shown in Figure 7-1.

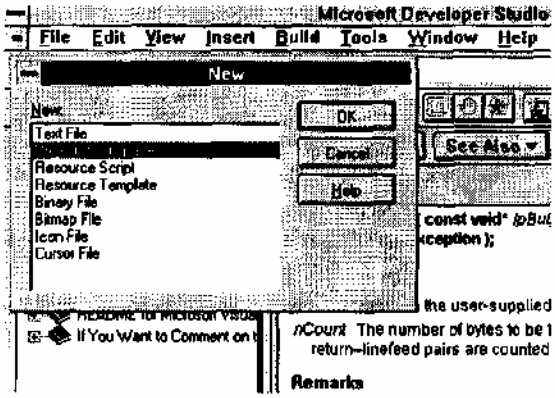


Figure 7-1. Starting a new project workspace

After opening the new project workspace, select the MFC AppWizard (exe) option to activate the AppWizard application to help build the skeleton software, as shown in Figure 7-2.

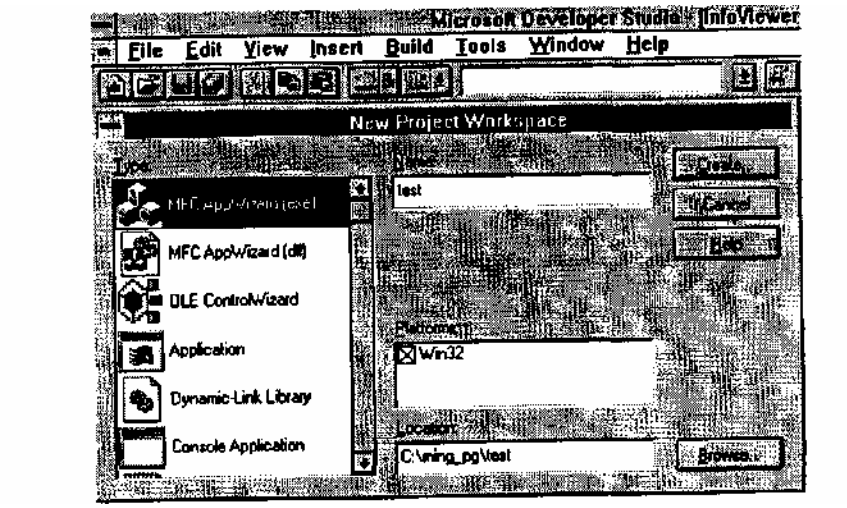


Figure 7-2. Using AppWizard

For the purpose of this example, use the AppWizard to create a Single Document application. To do this, choose the Single document option, as shown in Figure 7-3.

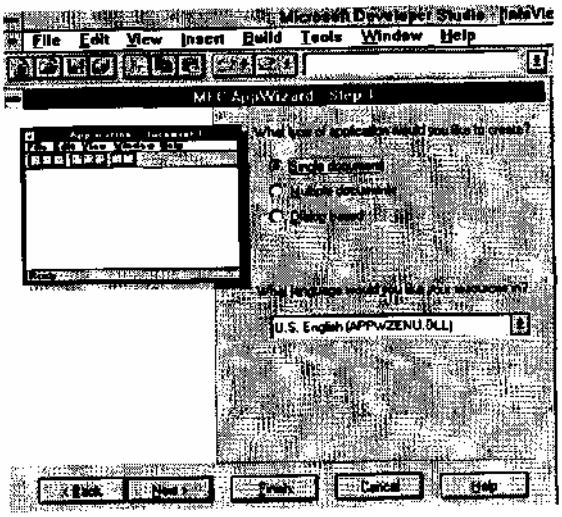


Figure 7-3. Step 1 of the of the AppWizard

There are several steps after the above "Single document" step. You'll need to follow the setup routine and select the desired options according to the requirements of the application. After completing all these steps, the AppWizard will create a skeleton of the program. A few program files will be generated accordingly.

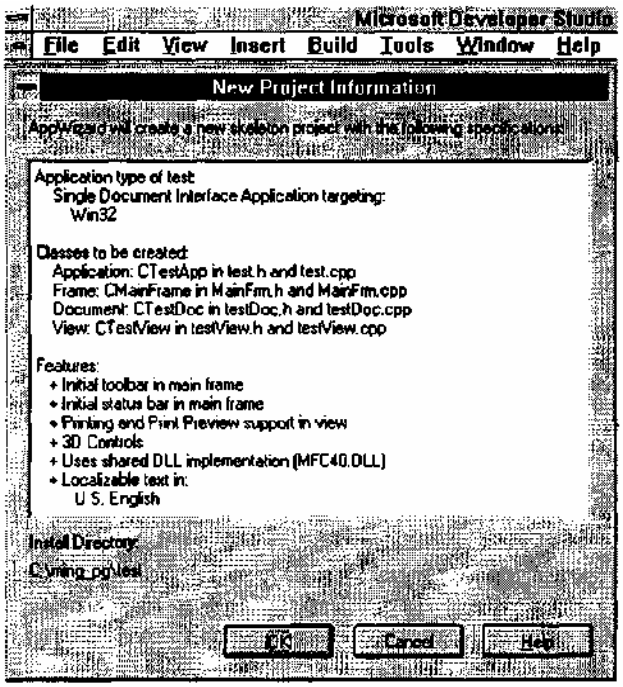


Figure 7-4. A skeleton project created by AppWizard

In this example, four C/C++ files and the corresponding header files are generated, as shown in Figure 7-4.

- test.cpp is the application class for the program, derived from the CWinApp
- MainFrm.cpp is the main frame class of the program
- testDoc.cpp is the program's document class, derived from CDocument
- testView.cpp is the program's view class, derived from the CView

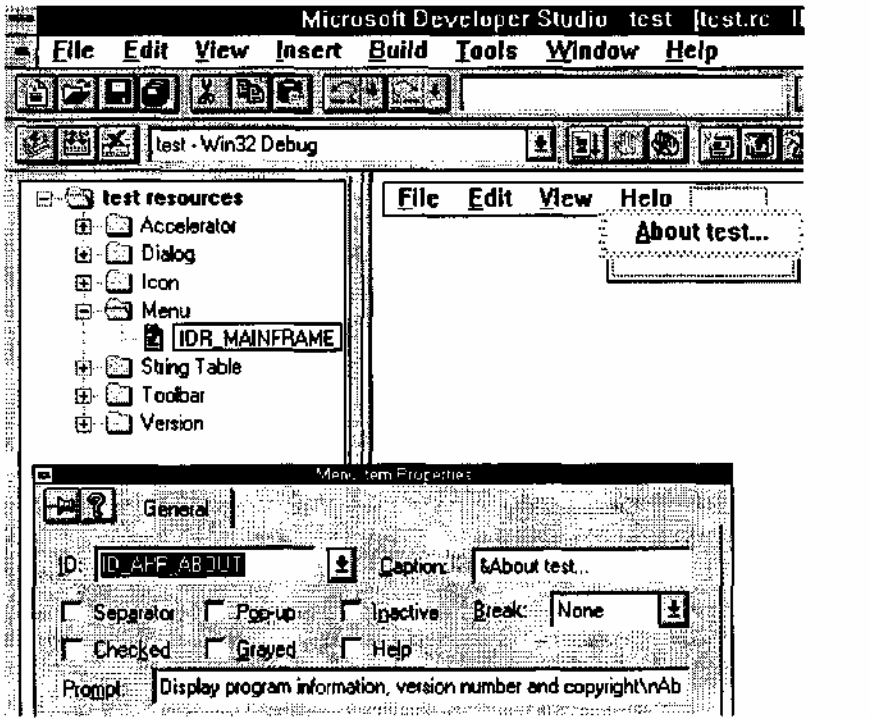


Figure 7-5. Menu item properties of the "About test..."

These C/C++ files provide basic software routines to build our project. To examine these C/C++ files, look at an example. Choose to investigate the About test... menu item. View it using the Menu folder from the Resource View tool (by clicking on the Resource View tab) in the project workspace window, as shown in Figure 7-5.

By double clicking the About test... item in the menu of the adjacent window, you can see the "Menu Item Properties" windows. As shown in Figure 7-5, an ID has already been defined by AppWizard to be the "ID_APP_ABOUT" for this case. This "ID__APP_ABOUT" ID is used in the application routine ON.COMMAND (ID_APP_ABOUT, OnAppAbout) of the BEGIN_MESSAGE_MAP(...) routine in the test.cpp file, as shown in Figure 7-6.

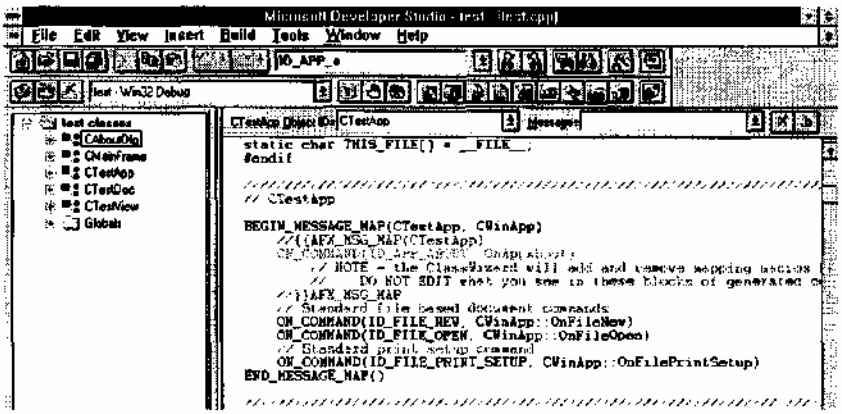


Figure 7-6. ID_APP_ABOUT in the BEGIN_MESSAGE_MAP routine

With this definition done, the OnAppAbout() routine, as depicted in Figure *7-7,* will be invoked when the "About test..." menu item of the application software is selected. Similarly, other menu functions and their corresponding IDs can be defined this way to provide additional functionalities to the application software. A more detailed discussion on how to create various menu items and other window functions can be found in *Programming Windows NT* 4[6] and *Teach Yourself Visual* C++ *4 in 21 Days*[7].

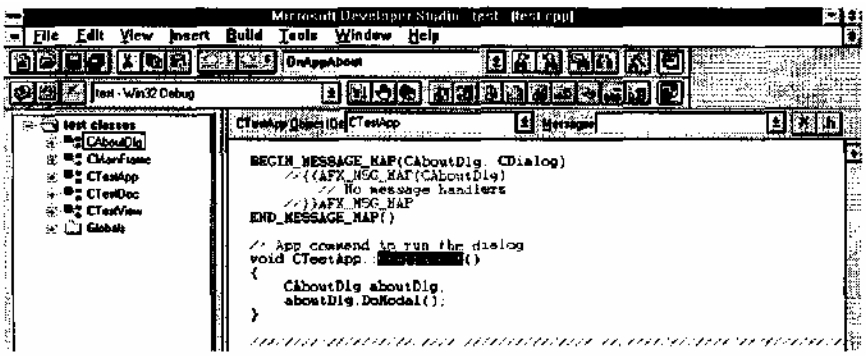


Figure 7-7. OnAppAbout() routine

7.3 Linking the Application Software to the WDM driver

As mentioned in Chapter 6, when a USB device is plugged into a host PC, a corresponding driver will be loaded. A "symbolic link" will also be created by the driver routine (using IoCreateSymbolicLink(...)) during

the loading of this driver. This "symbolic link" enables user mode applications to open and operate on the driver object. On the application software side, a file handle must be created to match the "symbolic link" that was created by the device driver. For example, consider the device driver with the following code:

```
// in the device driver routines
NTSTATUS DriverPnPPowerEvent(...) {

loCreateSymbolicLink("\\DosDevices\\TRYING_00", "\\Devices\\T00");
        /* T00 is the object name created using IoCreateDevice(...) routine */
```

This driver code can be "linked" to the application software using the following routine to create a file handle with the matching "TRYING_00" symbolic link.

```
// in the application software codes
Int CMainFrame::OnCreate(...) {

hDrvrHnd=CreateFile("\\\\. \\TRY1NG_00",  GENERICJVRITE | GENERIC_READ,
FILE_SHARE_WRITE | FILE_SHARE_READ, NULL, OP EN _EXISTING, 0, NULL);

}
```

In this manner, the application software can "link" and "communicate" with the device driver of the USB device to perform various application functions.

7.4 Communicating to the WDM Driver

After the above loading and linking of the driver object to the application software, the application is ready to "communicate" to the driver. A system routine, DeviceIoControl(...), is used by the application software to send command code directly to the device driver, making the corresponding device perform specified functions. The DeviceIo-Control(...) routine has the following parameters:

```
BOOL DeviceIoControl(

   HANDLE hDevice, DWORD          // the handle to the device driver
   dwhControlCode, LPVOID         //control code of operation to perform
   IpInBuffer, DWORD              //pointer to the input data buffer
   nlnBufferSize LPVOID           //size of the input buffer
   IpOutBuffer, DWORD             //pointer to the output data buffer
   nOutBufferSize, LPDWORD        //size of the output buffer
   IpBytesReturned,               //pointer to variable to receive output byte count
   LPOVERLAPPED IpOverlapped      //pointer to overlapped structure for asynch. op.

  );
```

For example, say a menu item has been defined to send a packet of information to a USB device through the WDM driver. Define the menu item "Get Command - One" for this purpose. In the "Menu Item Properties", define the ID for this "Get Command - One" function, naming it "ID_GET_ONE". This "ID_GET_ONE" is then used in the application routine ON_COMMAND(ID_GET_ONE, OnGetOne) of the BEGIN_MESSAGE_MAP(...) routine, as shown below.

In the OnGetOne() routine, include the following code to send an IRP, using the DeviceIoControl(...) routine, to the USB device driver.

```
// application software code example
//headers files ...

BEGIN_MESSA GE_MAP(CVsbappApp, CWinApp)

        ON_COMMAND(ID__APP_ABOUT, OnAppAbout)

        ON_COMMAND(ID_SET_ONE, OnSetOne)
        ON_COMMAND(ID_SET_TWO, OnSetTwo)
        ON_COMMAND(ID_GET_ONE, OnGetOne)
        ON COMMAND(ID_GET_TWO, OnGetTwo)

        //Standardfile based document commands ON_COMMAND(ID_F1LE_NEW,
        CWinApp::OnFileNew) ON_COMMAND(ID_F1LE_OPEN, CWinApp::
        OnFileOpen) // Standard print setup command
        ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)

END_MESSAGE_MAP()

//... others routines ...
//-----------------------------------------------------------------------------------------------
```

```
//=========================================
//OnGetOne() routine

void CUshappApp::OnGetOne() {
 PVOID   pvBuffer              = 0;
 pvBuffer = malloc (sizeof(Usb_Device_Descriptor) + 64);

 //Perform the Get-Descriptor IRP
 DeviceloControl (hDrvrHnd,
                  lRP_Test_GET_CONFIGURA TION_DESCRIPTOR,
                   pvBuffer,
                  size of (Usb_Device_Descriptor),
                 pvBuffer,
                  sizeof (Usb_Device_Descriptor),
                  &dwStatus,
                  NULL);

 free(pvBuffer);
  AfxMessageBox("completed 'Get Command-One' ");

}
```

With this definition, when the "Get Command - One" menu item is chosen, the application will send the "IRP_Test_GET_CONFIG-URATION_DESCRIPTOR" IRP to the WDM driver. The driver, as described in Chapter 6, will then process the IRP and generate a corresponding URB to perform the specified USB function. The code examples on the enclosed diskette and in the Appendices will give you a more detailed insight into the software programs required for USB devices.

8. USB Application Software, WDM Driver, and Firmware Examples

This chapter describes the USB application software, WDM driver, and firmware examples that are included on the enclosed diskette. These examples illustrate how application software interacts and communicates with the WDM device driver of a USB device. The USB device uses a 8x930Ax microcontroller with its firmware to handle USB enumeration and commands. To demonstrate the examples, you'll need a USB host PC and a 8x930Ax evaluation board as shown in Figure 8-1.

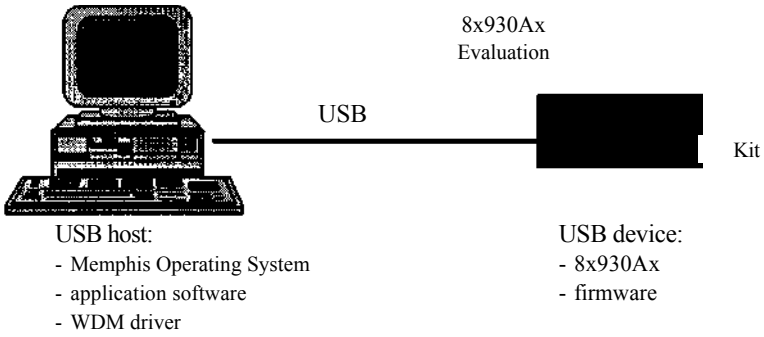


Figure 8-1. Lab setup to use the USB code examples

The host PC runs on a Microsoft's Memphis Developer release (build 1351) operating system, which supports USB Plug and Play devices. The application software and WDM driver examples run on the host PC. To get a copy of the Memphis Developer Release operating system, contact Microsoft or the USB-IF. On the other side of the USB wires, use the enclosed firmware example and the Intel 8x930Ax evaluation board as the USB device.

Note: The code examples included here are not production worthy, and are meant to depict some concepts highlighted in the book. Users are required to modify the application software, device driver and firmware to suit their requirements and to make the programs more robust as well as to adhere to specifications. The code examples are provided "as is" without warranty of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and/or fitness for a particular purpose.

8.1 Content Description of the Enclosed Diskette

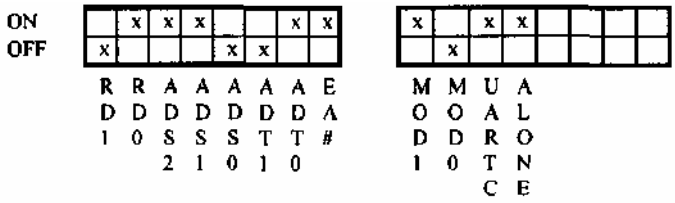
There are several files included in the enclosed diskette:

Filename	Description	Tools used
test_rom.ihx	A hex file in Intel hex format for the USB device to perform enumeration and handle some control transfer functions. It is ready to be programmed into a 32 Kbyte EPROM. This EPROM is then plugged into the 8x930Ax evaluation board which will then behave as a USB device. The corresponding firmware source code file, in assembly language format is usb_enum.asm located in the \device directory.	It is assembled using a PLC 3.06 assembler. The corresponding project file is test_rom.pro. Other assemblers can also be used but users may need to do minor modifications to the asm file.
testdrv.sys	A WDM driver for the USB device to be loaded onto the host. The corresponding source files (in C/C++ format) and header files are located in the \host\driver directory. This driver is ready to be loaded using the corresponding test.inf file in the floppy's root directory.	It is built using the Microsoft NT DDK, USB DDK, and Visual C/C++ compiler. It can be built under Free Build or Checked Build Environment after the DDK is installed. The file titled "sources" in the \host\driver directory defines the source files' components, libraries, and compilation environment.
test_app.exe	An executable application software for the USB host PC. Its corresponding source files (in C/C++ format) and header files are located in the \host\app directory. The test_app.exe is compiled with dynamic link library, whereas the test_app.exe compressed in the test_app.zip is compiled with static library.	It is compiled using Microsoft Visual C/C++ and Developer Studio with the project work-space file test_app.mdp. The test_app.mdp defines all the required files and resources.

8.2 Setting Up the Demonstration

After completing the lab setup as described in the previous sections, it is fairly easy to demonstrate and use the application software, WDM driver, and firmware examples.

First, program the Intel hex file test_rom.ihx into a 32 Kbyte EPROM and then plug it into the 8x930Ax evaluation board (replacing the ROM device that is on the evaluation board). Note the settings of various dip-switches of the board, especially the EA#, page/non-page, and binary/source mode switches. The EA# pin of the 8x930Ax microcontroller needs to be enabled (pulled low) so that the 8x930Ax executes its code from external memory. The page/non-page dip-switch needs to be set to correspond to the setting for the external memory bus of the evaluation board. The binary/source mode dip-switch must be set to binary mode for this example. For the purpose of this demonstration, set the dip-switches of the evaluation board to:



Please refer to the 8x930Ax Evaluation Board User's Manual[3] (see Section 3.3 for how to obtain manual) for detailed descriptions of the board.

After setting up and powering up the evaluation board, it is ready to be used as a USB device. The LED that is connected to port pin p1.0 will blink to indicate that the board is operational. At this point, connect the evaluation board to the USB host using a USB cable.

This should activate the enumeration process, and on the host PC an "Update Device Driver Wizard - Unknown Device" screen will pop up, as shown in Figure 8-2.

At this point, the host has performed the USB enumeration and, among other things, it has obtained the vendor and product identifications of the newly plugged-in device. However, the host cannot find a corresponding information (.inf) file that matches the product and vendor identifications of the device. When this happens, the host will ask for the location of the .inf file so that it can load the correct driver. Insert the enclosed diskette and direct the host to the location of the .inf file, which is in the floppy's root directory.

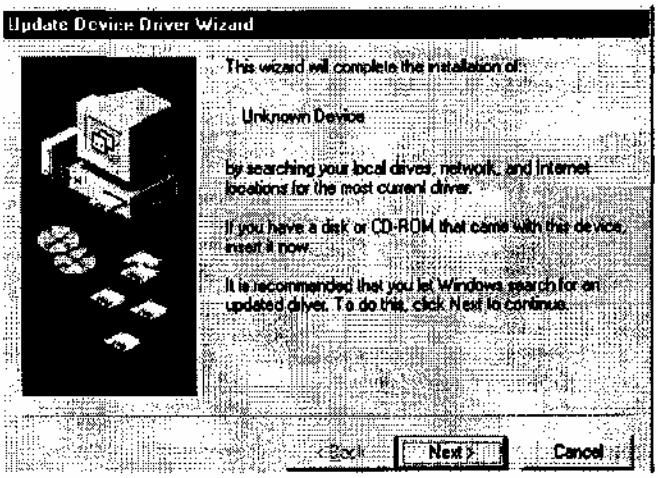


Figure 8-2. A "unknown device" screen on the host PC

Note that this "Update Device Driver Wizard" process is only required when plugging in the device for the first time, since the host PC could not find the corresponding .inf file. For subsequent plug-in events, the host will load the device driver without the "Update Device Driver Wizard" process and interactions from users. This depicts the Plug and Play advantage of USB systems. They can recognize the newly plugged-in device and load the appropriate device driver automatically.

Once directed, the host PC loads the .inf file that matches the product and device identifications of the newly plugged-in device. Note that, for the purpose of this example, arbitrary product and device identifications are used. You are required to change these identifications in the device firmware and .inf file accordingly.

Based on the information in the .inf file, the host PC can determine which driver to load and will then ask for the location of the driver, as shown in Figure 8-3.

The driver, testdrv.sys, is located in the root directory of the enclosed disk, so choose the "Other Locations..." button and direct the host to the correct drive.

The host PC will then load the driver into its operating system. This completes the loading of the WDM driver for the newly-attached USB device.

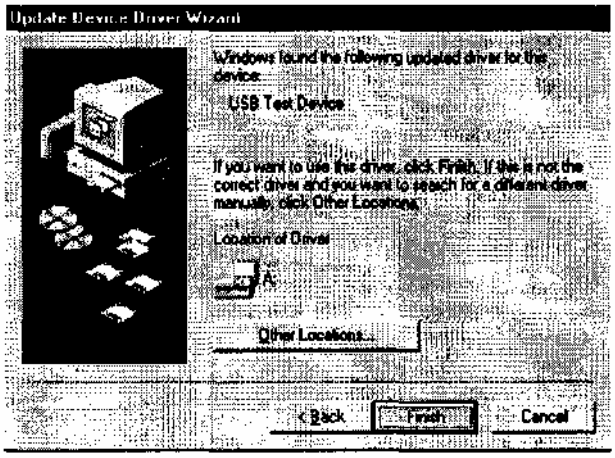


Figure 8-3. Loading process of WDM driver for the newly attached device

Now activate the host application software, test_app.exe (located in the root directory of the floppy), to communicate with the USB device. An "Untitled - A Simple USB Program Example" user interface, as shown in Figure 8-4, should appear.

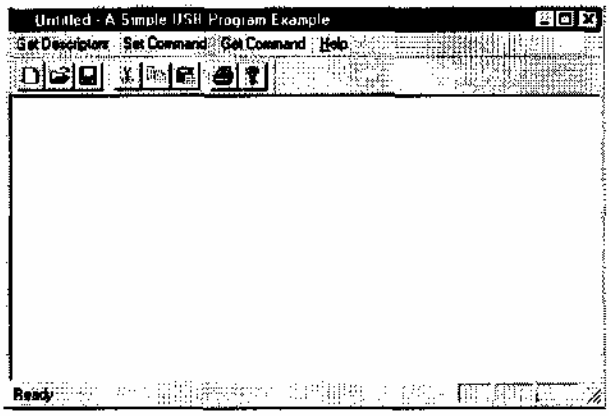


Figure 8-4. Test_app.exe application software user interface

If the driver is loaded successfully, the "Get Descriptors", "Set Command", and "Get Command" menu items are highlighted (not "grayed"), as shown in Figure 8-4. You can activate these menu items to perform some simple USB transactions, as shown in Figure 8- 5. Various USB packets will be generated and can be seen if you insert a bus analyzer in the USB wires path, as shown in Figure 8-6.

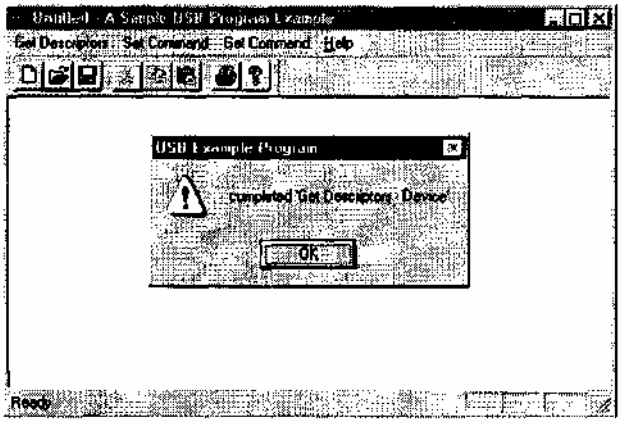


Figure 8- 5. Activate the menu item to generate a USB transaction

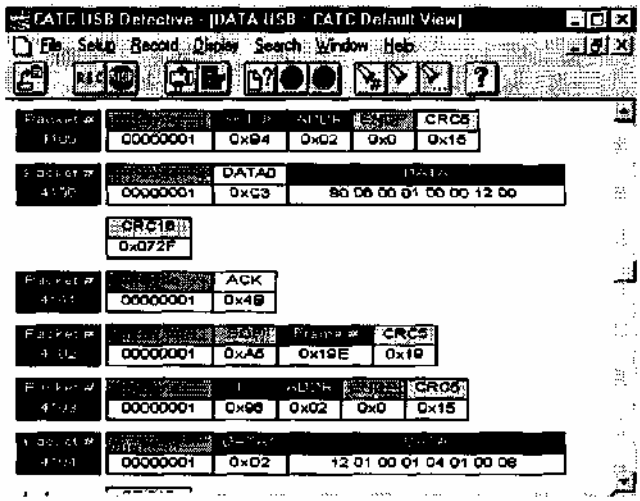


Figure 8- 6. USB transactions as captured by a CATC bus analyzer

The application software, WDM driver, and firmware in this example only perform some very simple and basic USB functions. In this example, only endpoint 0 is enabled and used to perform USB enumeration and various control functions. Nevertheless, it provides a simple USB example allowing you a good basic understanding of the operation of the USB host, WDM driver, and the USB device.

To get other code examples for application software, WDM drivers, or device firmware performing various USB transfer types, you can contact

the Intel web-site at http://www.intel.com/design/usb/swsup. USB code examples can also be found in the Microsoft USB DDK package.

8.3 Deleting Driver and ID Information from the Host

When you're done with the example, you might want to delete the product information and the device driver from the host PC. To do this, you'll need to delete the product information from the registry of the host PC. First, activate the registers editor, regedit.exe, located in your \windows directory. Using the regedit, locate the registry information of the device, which is in the \HKEY_LOCAL_MACHINE\Enum\USB directory, as shown in Figure 8-7. Delete this production information by selecting the product and vendor identification information, then press the "delete" key.

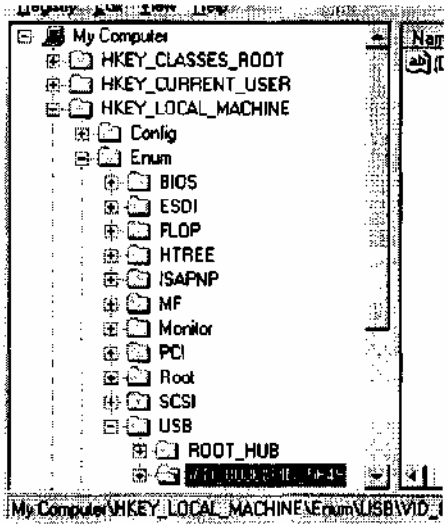


Figure 8-7. Locating the product information in the registry

To delete the corresponding driver example (testdrv.sys) or the inf file (test.inf), use the Windows Explorer or MS-DOS commands. The driver is located in the \windows\system directory and the .inf file is stored in the \windows\inf\other directory (a "hidden" directory).

9. Conclusion

We have discussed all the major steps and tasks required to develop USB devices. We have covered the synopsis of the USB specification (rev 1.0), development environment, tool requirements, a USB microcontroller and its firmware operating model, WDM driver, device class specification, and host application software. We have also investigated the interactions and communications between the host application software and the WDM driver of the USB device.

Code examples of the host application software, WDM driver and device firmware are included on the enclosed diskette and in the Appendices for your reference. The code examples and steps to use the code examples are also discussed in Chapter 8 of the book. Although these codes are not production worthy, they give you a good insight into USB systems; from both the host side and the device side. These codes can be used as a reference, and can be further improved and modified to ensure their robustness and conformance to the USB protocol, device class, FCC, and other relevant specifications and regulations that are required for commercial devices.

Once a developed prototype reaches a certain stage of maturity, you can participate in the USB PlugFest (sometimes called the USB Compatibility Workshop) to test the prototype. The USB PlugFest is organized by the USB-IF, generally on a bi-monthly basis, for developers to test their prototypes to ensure conformance to various USB hosts. Developers from host and device vendors can get together to test and debug their systems on a real time basis. For more information about the PlugFest, contact the USB-IF (http://www.usb.org).

It is the purpose of this book to provide you with all the information needed about USB systems and how to develop USB devices. With this guidance, it is hoped that you can start developing your own USB devices, make prototypes, and build production units.

Get into the USB bus!

10. Glossary

ACK	Acknowledgment. Handshake packet indicating a positive acknowledgment.
Apbuilder	Software from Intel. It helps you to become familiar with the 8x930Ax and other architectures.
API	Application Programming Interface. A set of routines that you can make use of to request and carry out certain services performed by an operating system.
Bandwidth	The amount of data transmitted per unit of time, typically in units of bits per sec (bps).
Big Endian	A method of storing data that places the most significant byte of multiple byte values at lower storage addresses.
Bit	A binary unit of information used in computers. A bit has the choice of two possibilities, typically represented by a logical one (1) or zero (0).
Bit Stuffing	Insertion of a "0" bit into a data stream of *"V* to cause an electrical transition on the data wires to allow the locking of PLL circuitry.
bps	Bit per second. It is used as a unit of measurement for data transmission rates.
Bulk transfer	One of four USB transfer types. It represents a non-periodic, large bursty data transmission, typically used for a transfer that can use any available bandwidth and that can also be delayed until bandwidth is available.
Bus enumeration	A process of communication between a USB host and a USB device when the device is first plugged into the bus. It is used for identifying the USB devices.
Byte	A data element that is eight bits in size.
Client	Software resident on USB host that interacts with host software to arrange data transfer between a function and a host. The client is often the data provider or consumer of the transferred data.

Client/Server Model	An application or operating system model where the system is divided into processes (server), each of which provides a set of specialized services to other processes (clients).
COM port	Communications port. An eight-bit asynchronous serial port typically used on personal computers.
Control transfer	One of the four USB transfer types. It supports configuration, command, or status type communications between client and function.
CRC	Cyclic redundancy check. A check performed on the transmitted data to see if errors have occurred in transmitting, reading, or writing the data. The CRC is typically stored or transmitted with the actual data and at the receiving end, the stored or transmitted CRC is compared to a calculated CRC to determine if errors have occurred.
CTI	Computer Telephony Integration.
Default address	An address defined by the USB Specification and used by a USB device when it is first powered or reset to communicate to the host. The default address is 00H.
Default pipe	A message pipe created by the USB host system software to communicate with USB devices.
Device	A logical or physical entity that performs a USB function. It may be referred to as a single hardware component, a collection of hardware components, or a function that provides specified services, e.g. a fax modern device.
Device Object	A system object that represents a physical, logical, or virtual device and describes its characteristics. It is associated with a driver object. See Driver Object.
DLL	Dynamic-Link Library. An application programming interface (API) routine that user-mode applications can access. The code for the API routine is not included in the user's executable image. Instead, the operation system automatically "links" the executable image to point to DLL procedures at runtime.

Downstream	The direction of flow of communication from the host PC to its peripherals.
Driver	When referring to hardware, it is an I/O pad that drives an external load. When referring to software, it is a program responsible for interfacing to a hardware device.
Driver Object	A system object that represents a driver on the system and provides the address of the driver's entry points to the I/O manager. A driver object can be associated with multiple device objects. See device object.
DWORD	Double word. Data that is two words in size (four bytes or 32 bits).
Endpoint	A uniquely identifiable portion of a USB device that is the source or sink of information in a communication flow between a host and a device.
Endpoint Address	An address location specified in a USB token to a particular device.
Environment Subsystem	A protected subsystem (server) that provides an application programming interface (API) andenvironment - such as Win32, MS-DOS, OS/2 - on Windows NT. The subsystem captures API calls and implements them by calling Windows NT services.
EOP	End of packet.
EPROM EEPROM (or E^2PROM)	Electrically (Erasable) Programmable Read Only Memory. A non-volatile memory storage technology.
Evaluation Kit	In this book, an evaluation kit for the Intel 8x930Ax microcontroller. It consists of a 8x930Ax, evaluation board, evaluation copy of assembler, compiler and debugger from a third part vendor, and Apbuilder software.
File Handle	A handle to a file object, which is used by application software to communicate to the object.
Frame	The time frame from the start of SOF token to the start of SOF token of the subsequent token. This period has a nominal value of 1 ms, as defined by the USB Specification.

Full-duplex	Data transmission occurring in both directions simultaneously.
HAL	Hardware Abstraction Layer. A DLL that protects the NT executive from variations in different vendors' hardware platforms. This makes the operating system more portable.
Handshake packet	A packet that acknowledges or rejects a specified condition. See ACK and NACK.
Host	The host computer system where the USB host controller is installed. It includes the host hardware platform and the operating system in use.
Hub	A USB device that provides additional connections to the USB bus.
Interrupt transfer	One of four USB transfer types. Its transfer characteristics are small data, non-periodic, low frequency, bounded latency, device initiated (but polled by host) communications typically used to notify the host of a device service need.
I/O Manager	One of the Windows NT executive components. It defines an orderly framework within which I/O requests are accepted and delivered to file systems and device drivers.
IRP	I/O Request Packet. A data structure used to represent an I/O request and to control its processing. It is created by the I/O manager and is passed to one or more drivers for processing. When the drivers finish performing the operation, the I/O manager completes the I/O and deletes the IRP.
ISDN	Integrated Services Digital Network. An internationally accepted standard for voice, data, and signaling using public switched telephone networks. All transmissions are digital from end-to-end.
Isochronous transfer	One of four USB transfer types. It is used to transmit a stream of data whose timing is implied by the delivery rate. Isochronous transfer provides periodic, continuous communication between host and device.

Kernel Mode	The privileged processor mode where Windows NT system code runs. A thread running in kernel mode has access to system memory and to hardware.
kbps	kilo bit per second, 1 kbps equals 1000 bps.
Little Endian	Method of storing data that places the least significant byte of multiple byte values at lower storage address.
Mbps	Mega bits per second.
Modem	Modulator/Demodulator. Component that converts signals between analog and digital. Typically used to send digital information from a computer over a telephone network which is usually analog.
MSB	Most Significant Bit.
Multitasking	A processor's execution of more than one thread by context switching. This provides the illusion that all threads are executing simultaneously.
NACK	Negative acknowledgment. Handshake packet indicating a negative acknowledgment.
NRZI	Non Return to Zero Invert. A method of encoding serial data in which ones and zeroes are represented by opposite and alternating high and low voltages where there is no return to zero (reference) voltage between encoded bits. Eliminates the need for clock pulses.
NT Executive	The section of the Windows NT operating system that runs in kernel mode to provide process structure, inter-process communication, memory management, object management, thread scheduling, interrupt processing, I/O capabilities, networking, and object security.
Object	Host software or data structure representing a Universal Serial Bus entity.
Object Manager	The components of the NT executive that creates, deletes, and names operating system resources, which are stored as objects.

OSI	Open System Interconnect reference model. A software model defined by the International Standards Organization (ISO) that standardizes levels of service and types of interaction for networked computers and communication systems.
Packet	A group of data for transmission. Packets typically contain three elements: control information (e.g., source/ destination/ and length)/ the data to be transferred, and error detection and correction bits.
Packet ID	A field in a Universal Serial Bus packet that indicates the type of packet, and by inference the format of the packet and the type of error detection applied to the packet.
PCI	Peripheral Component Interconnect. A 32- or 64-bit/ processor independent expansion bus used on personal computers.
Physical	A device that has a physical implementation; e.g. speakers/ telephones, and CD players.
device Pipe	A logical abstraction representing the association between an endpoint on a device and software on the host. A pipe has several attributes; for example/ a pipe may transfer data as streams (Stream Pipe) or messages (Message Pipe).
PLL	Phase Locked Loop. A circuit that acts as a phase detector to keep an oscillator in phase with an incoming frequency.
PnP	Plug and Play, A technology for configuring I/O devices to use non-conflicting resources in a host. Resources managed by Plug and Play include I/O address ranges, memory address ranges, IRQs, and DMA channels.
Polling	Asking multiple devices, one at a time, if they have any data to transmit.
POTS	Plain Old Telephone Service. - Basic service supplied by standard single line analog telephones, telephone lines, and access to public switched networks.
Process	A logical division of labor in an operating system. For Windows NT, it consists of a virtual address space, an executable program, one or more threads

	of execution, some portion of the user's resource quotas, and the system resources that the operating system has allocated to the process's threads.
Protocol	A specific set of rules, procedures, or conventions relating to format and timing of data transmission between two devices.
Request	A request made to a Universal Serial Bus device contained within the data portion of a SETUP packet.
Root hub	A Universal Serial Bus hub directly attached to the host controller. This hub is attached to the host; tier 0.
Root port	The upstream port on a hub.
SCSI	Small Computer Systems Interface. A local I/O bus that allows peripherals to be attached to a host using generic system hardware and software.
SFR(s)	Special Function Registers. Registers used by microcontrollers, for our case the 8x930Ax, as control panels, and status indicators to its firmware.
SOF	Start of Frame. The SOF is the first transaction in each frame. SOF allows endpoints to identify the start of frame and synchronize internal endpoint clocks to the host.
Stage	One part of the sequence composing a control transfer; i.e., the setup stage, the data stage, and the status stage.
Synchronization Type	Type A classification that characterizes an isochronous endpoint's capability to connect to other isochronous endpoints.
TDM	Time Division Multiplexing. - A method of transmitting multiple signals (data, voice, and/or video) simultaneously over one communications medium by interleaving a piece of each signal one after another.
Thread	An executable unit that belongs to one (and only one) process. It consists of a program counter, a user-mode stack, a kernel mode stack, and a set of register values. All threads in a process have equal access to the process's address space, object handles, and other resources. Threads are implemented as objects.

Thread Scheduling	The process of examining the queue of threads that are ready to execute and selecting one to run next; performed by NT executives.
Time-out	The detection of a lack of bus activity for some predetermined interval.
Token packet	A type of packet that identifies what transaction is to be performed on the bus.
Transaction	The delivery of service to an endpoint; consists of a token packet, optional data packet, and optional handshake packet. Specific packets are allowed/required based on the transaction type.
Transfer Type	Determines the characteristics of the data flow between a software client and its function. Four Transfer types are defined: control, interrupt bulk, and isochronous.
UART	Universal Asynchronous Receiver and Transmitter. An on-chip industrial standard MCS 51 type 4-mode, two wire serial port of the 8x930Ax microcontroller.Includes hubs and functions. See Device.
Universal Serial Bus Software	The hardware interface between the Universal Serial Bus cable and a Universal Serial Bus device. This includes the protocol engine required for all Universal Serial Bus devices to be able to receive and send packets.
Upstream	The host-based software responsible for managing the interactions between the host and the attached Universal Serial Bus devices.
USB	The direction of data flow towards the host.
USBD	Universal Serial Bus.
User Mode	Universal Serial Bus Driver. -- The host resident software entity responsible for providing common services to clients that are manipulating one or more functions on one or more Host Controllers.
	The non-privileged processor mode in which application code runs. A thread running in user mode can gain access to the system only by calling system services.

11. Appendices

11.1 Firmware Code Example For Intel 8x930Ax USB Microcontroller

```
;Usb_enum.ASM
;Firmware for the 8x930Ax. Ax2, Ax3 and Ax4 to perform
USB ;enumeration.      It is to be programmed into a 32 Kbyte
EPROM ;of the Evaluation kit to be used as a USB Test Device

;rev 1.0 (3 Mar 97)



INCLUDE "8x930Ax.h"

FIFO_SIZE             equ         008h
GET_COMMAND           equ         080h
SETUP_PHASE           equ       000h
DATA_PHASE            equ        001h
STATUS_PHASE          equ         002h
EP0_MAX_PACKET_SIZE         equ
08h NULL_DATA_PACKET              equ
000h

org 0ff:7ff8H
      db 0D6h           ;congifuration bytes ; Binary, Mode-Paged Mode,
      etc db 0EFh

;;org 00:4000H ; if use PLC and RISM org
0ff:0000H1jmp     main

;;org 00:4043H
org 0ff:0043H
      ljmp        SOF_ISR

;; org 00:404BH
```

```
org Off:004BH
      ljmp      FUNCTION_ISR

;; org 00:4080H
org 0ff:0080H

; Main

mam:
      mov   SP,       #00h
      mov   SPH,      #01h          ; Set the stack to start @ 00:0500h
      mov   DPXL,     #0ffh         ; Set DPXL to point at the FLASH device
                                    ; to access constants
Init_Variables:
      mov   R0, #00h
      mov   Beatl.R0
      mov   Beatl+1, R0             ; zeroing counter

      mov   SetupRxFlag, R0                 ; SetupSeq is set to
      mov   SetupTxFlag, R0                 ; Setup_PHASE

      mov   ControlBuffLocation, R0         ; zeroing pointer
      mov   ControlBuffLocation+1, R0
      mov   ControlBuffLocation+2, R0

      mov   ControlBuffBytesLeft, R0        ; zeroing counter

      mov   WR0, #0                 ; zeroing counter
      mov   WR2, WR0


Init_Usb:                   ; configure endpoint 0 (EP0)
      mov       EPINDEX, #00        ; select EP0
      nop
      nop
      mov       EPCON,   #3Fh       ; enables as control endpoint
                                    ; single packet mode, receive enabled ;
                                    and transmit enabled

      orl       FIE,     #03h       ; enable EP0 TX & RX
      interrupts.
      setb      IEN1.1     ; enable interrupts

      setb      SOFIE      ; enable SOF interrupt
```

```
        setb    IEN1.0      ; enable SOF ISR

        setb    EA          ; enable glabal interrupts

        mov     p1, #00H    ; P1 is the LED indicator in eval kit
        CLR     LC          ; increase the CPU speed from 3 MHz to 12 MHz

        ljmp    Idle_loop


; Idel_loop
; device looping here awating interrupts

Idle_loop:
   nop
   nop          ; can implement user applications here
   nop

LedBeat:
        mov     WR0,    Beatl
        inc     WR0,    #1
        mov     Beatl, WR0
        cmp     WR0,    #7FFFh
        j1      ClrBit
        setb    Pl.0
        ljmp
Idle_loop ClrBit:
        clr     Pl.0
        ljmp    Idle_loop



; FUNCTION_ISR
; device jumps here when a receive done or transmit done interrupt occurs

FUNCTION_ISR: push
        ACC push    B
        push    EPINDEX

ChecklnterruptSource:
F_EP0_RX:
```

```
        jnb         FRXD0, F_EP0_TX
        mov         EPINDEX,#00
        lCall       SetupOutToken          ; 930Ax has received a packet from host PC

F_EP0_TX:
        jnb         FTXDO, F_EP1_TX
        mov         EPINDEX,#00
        lCall       InReceived             ; 930Ax has sent a packet to host PC

F_EP1_TX:

EXIT_FUNCTION_ISR:
        pop         EPINDEX
        pop         B pop
        ACC reti



; SOF_ISR at every 1ms
; do nothing here for our example
;
SOF_ISR:

        clr         ASOF
        cpl         p1.3          ; complement p1.3 as indication

reti


; SetupOutToken
; check whether a Setup or an OUT token is received
;
SetupOutToken:

        anl         SBI,     #EP0_RX_CLR              ; Clear the interrupt bit.
        ljmp        ProcCommand

ProcCommand:
        mov         A,          RXSTAT
        jnb         ACC.6, CheckOutStatusPhase              ; check if this is a setup
        packet??
```

```
        mov         A,
        TXFLG anl       A,
        #0C0h jZ
        NoEP0Error

NoEP0Error:
        lCall   SetupReceived

        ;;!!! moved setb     RXFFRC                  ; Update receive FIFO state

        ;;!!! moved clr      RXSETUP

        ljmp        ReturnProcessOutToken

CheckOutStatusPhase:
        mov         A,      SetupRxFlag
        cjne    A,      #STATUS_PHASE, OutReceived      ; the status phase of a
                                                        ;"GET" command?

        setb    RXFFRC          ; Update receive FIFO state
        setb    TXCLR           ; Flush the Transmit FIFOS in case
                                ; a null packet is still left.

        mov         A,      #SETUP_PHASE        ; Update the flag

        mov   SetupRxFlag,   A   mov
        SetupTxFlag,        A       ljmp
        ReturnProcessOutToken


        ;     control Write Data Stage
        ; If this is a control write command with a datastage then this
        ; Routine will be called on all data stages of the control write.
        ; When all the data has been collected(Bytes received=wLength)
        ; the actual Routine is called. The size of data is limited to this
        ; buffer length.
        ; The code must then call SetUpcontrolWriteStatusStage to allow
        ; the status stage to continue.

OutReceived:
        cjne    A,      #DATA_PHASE, ReturnProcessOutToken ;
        Are we Processing a control Write, i.e. Set Descriptor... ;
        Added to handle control writes with data stages. ; When a
        control Write with a data stage is detected
```

```
        ; the data is placed in the buffer 'CntlWriteDataBuffer'

        push    R0
        mov     A,      #CntlWriteDataBuffer            ; Get location of
        buffer

        add     A,      CntlWriteDataPntr   ; Add the offset
        mov     R0,     A
        ; R0 now contains the location to start storing the data.

                                    ; Update the data stored in memory
        mov     A,      RXCNT           ; Get number of bytes to move
        add     A,      CntlWriteDataPntr           ; Add number
        received
        mov     CntlWriteDataPntr, A    ; update memory variable

        mov     A,      RXCNT
        JZ      RdDone                  ; If no data in the buffer, then exit.

ReadData:
        mov     @R0,    RXDAT           ; Get the data and store it.
        inc     R0
        djnz    ACC,    ReadData
        pop     R0
RdDone:
        setb    RXFFRC                  ; Update receive FIFO state

; Now check to see if this was the last ; read by checking if (Bytes
received=wLength)
        mov     A,      CntlWriteDataPntr
        cjne    A,      wLength+1, NeedMoreData         ; If this is not equal to
        the
                                                ; expected, then ljmp around.

ProcesscontrolWriteData:

        push    DPX             ; Processing the jump table will
                                ; corrupt DPX. Save it here
        lCall   Ch9Decode       ; proceed to Process it

        pop     DPX             ; Restore DPX
        ljmp    ReturnProcessOutToken

NeedMoreData:
CheckCommand2:
```

```
ReturnProcessOutToken:
      Ret



; SetupReceived
; device braches here when a setup token received

SetupReceived:
      mov       A,       RXCNTO        ; Get the no. of bytes
      clr       CY

      subb      A,       #8            ; 8 is length of all setup packets.
      JNZ       ReturnSetup            ; If less than 8 bytes reed. ; Return

      clr       EDOVW
      mov       COMMAND_BUFFER,     RXDATO        ;
      bmRequestType
      mov       COMMAND_BUFFER+1, RXDATO        ; bRequest
      mov       COMMAND_BUFFER+3, RXDATO        ; wValue LSB
      mov       COMMAND_BUFFER+2, RXDATO        ; wValue MSB
      mov       COMMAND_BUFFER+5, RXDATO        ; wlndex LSB
      mov       COMMAND_BUFFER+4, RXDATO        ; wlndex MSB
      mov       COMMAND_BUFFER+7, RXDATO        ; wLength LSB
      mov       COMMAND_BUFFER+6, RXDATO        ; wLength MSB

      push      DPX             ; Processing the jump table will
                                ; corrupt DPX. Save it here

      setb      RXFFRC          ;;!!! moved here for OHCI
      systems clr       RXSETUP         ;;!!! ditto

      lCall     ProcessSetup
      pop       DPX             ; Restore DPX

ReturnSetup:
      Ret



; Process Setup
; decode the setup packet received
;
ProcessSetup:
```

```
        mov         ControlBuffBytesLeft,
        #00h
        mov        ControlBuffBytesLeft + 1,
        #00h
        mov        A,        bmRequestType
         jb        ACC.7,            SetupGetCommand

SetupSetCommand:
        mov         SetupRxFlag, #DATA_PHASE            ; Advance flag to next
        state mov         SetupTxFlag, #STATUS_PHASE
                          ; Now check to see if this is a control write with a data stage,
        mov         A,        wLength
        orl         A,        wLength+1
        jz          Ch9Decode               ; If this is a no data command then Process it

        mov         CntlWriteDataPntr, #0         ; Initilize the data pointer for future
                                                  ; control write ;
                                    data stages.
        ret                          ; If we are expecting data then don't
                                     ; Process the command yet. ret and wait ;
                                     for the rest of the data to come in

SetupGetCommand:
        mov         SetupRxFlag, #STATUS_PHASE          ; Advance flag to next
        state mov         SetupTxFlag, #DATA_PHASE



; Ch9Decode
; to decode the bMRequestType according to chapter 9 of the USB spec
; we decode the bmRequestType first
>
Ch9Decode:

        mov         A,          bmRequestType  ; Get the value.

        ; check standards commands ?? jb
        ACC.6,      OthersCommands jb
        ACC.5,      OthersCommands

        ; divide into set or get commands jb
        ACC.7, GetCommands ljmp
        SetCommands
```

```
GetCommands:
    anl     ACC, #1FH       ; ignore bit 5,6,7 bit since they have been decoded
    cjne    A, #00H,next_c01
    ljmp    StandardGetDeviceCommand

    next_c01:
    cjne    A, #01H, next_c02
    ljmp    StandardGetlnterfaceCommand

    next c02:
    cjne    A, #02H, next_c03
    ljmp    StandardGetEndpointCommand

    next_c03:
    cjne    A,#03H, next_c04
    ljmp    StandardGetOtherCommand

    next_c04:
    ljmp    OthersCommands

SetCommands:
    anl     ACC, #1FH       ; ignore bit 5,6,7 bit since they have been decoded
    cjne    A, #00H, next_c11
    ljmp    StandardSetDeviceCommand

    next_c11:
    cjne    A, #01H, next_c12
    ljmp    StandardSetlnterfaceCommand

    next_c12:
    cjne    A, #02H, next_c13
    ljmp    StandardSetEndpointCommand

    next_c13:
    cjne    A, #03H, next_c14
    ljmp    StandardSetOtherCommand

    next_c14:

OthersCommands: ; not
implemented yet ret
```

```
  ;STANDARD COMMANDS
  ;further decode the standards commands

StandardGetDeviceCommand: mov        A,        bRequest cjne
      A,       #GET_DESCRIPTOR, CheckGetConfiguration

      m ov       A,       b Descr iptorType
      cjne       A,       #DEVICE_DESCR, CheckConfigDescriptor

      ; — GET DESCRIPTOR, DEVICE -----
      mov        ControlBuffLocation,
#LOW HIGH16(BEGIN_DEVICE
DESCRIPTOR)
      mov        ControlBuffLocation+l,
#HIGH
LOW16(BEGIN_DEVICE_DESCRIPTOR)
      mov        ControlBuffLocation+2, #LO
W
LOW16(BEGIN_DEVICE_DESCRIPTOR)
      mov        A, #12h
      mov        B, #00h
      ljmp       LoadBuffer

CheckConfigDescriptor:
      cjne     A,        #CONFIG_DESCR, CheckStringDescriptor

      ;.— GET DESCRIPTOR, CONFIGURATION -—
      mov        ControlBuffLocation,
#LOW
HIGH16(BEGIN_CONFIG_DESCRIPTOR)
      mov        ControlBuffLocation+1,
#HIGH
LOW16(BEGIN_CONFIG_DESCRIPTOR)
      mov        ControlBuffLocation+2,
#LOW LOW16(BEGIN_CONFIG
DESCRIPTOR)
      mov        A, #LOW
(END_CONFIG_DESCRIPTOR-BEGIN_CONFIG_DESC
RIPTOR)
      mov        B, #HIGH (END_CONFIG_DESCRIPTOR
-BEGIN_CONFIG_DESCRIPTOR)
      ljmp       LoadBuffer

CheckStringDescriptor:
      cjne     A,       #STRING_DESCR,
      ReturnBADSTDGetDeviceCommand
      ; ---------- GET DESCRIPTOR, CONFIGURATION -------
      mov        ControlBuffLocation,     #LOW HIGH 16(STRING_1)
      mov        ControlBuffLocation+1, #HIGH LOW16(STRING_1)
```

```
    mov   ControlBuffLocation+2, #LOW   LOW16(STRING_1)

    mov   A,     #LOW (STRING_2-STRING_1)
    mov   B,      #HIGH(STRING_2-STRING_1)
    ljmp  LoadBuffer


LoadBuffer:             ; Compare to see which is shorter.
                        ; The amount asked for or the amount available.

    push     ACC
    push     B
    clr      CY                 ; A=Actual-wLength=AskedFor
    subb     A, wLength+1
    mov      A, B
    subb     A, wLength
    jc       AskedFor_IsLarger

LengthsMatch:
wLengthlsSmaller:               ; If Asked for is smaller, replace
                                ; actual with asked for.
    pop      B
    pop      ACC
    mov      B, wLength
    mov      A, wLength+1
    ljmp     Loadlt

AskedFo_IsLarger:
    pop   B
    pop   ACC
Loadlt:   ; from now on, wLength = bytes remaining
    mov   ControlBuffBytesLeft,    B
    mov   ControlBuffBytesLeft+1, A
    lCall LoadControlTXFifo
    ljmp  ReturnSTDGetDeviceCommand

CheckGetConfiguration:
    cjne     A,      #GET_CONFIGURATlON,CheckGetStatus

    ;- GET CONFIGURATION

    ljmp     ReturnSTDGetDeviceCommand
CheckGetStatus:
```

```
        cjne    A,      #GET STATUS, ReturnSTDGetDeviceCommand
        ;————————————————————————
        ;- GET DEVICE STATUS

        ljmp    ReturnSTDGetDeviceCommand

ReturnBADSTDGetDeviceCommand:
        push    EPINDEX mov
        EPINDEX, #01 mov        TXDAT,
        #12h mov        TXCNT,#01h pop
        EPINDEX

        orl     EPCON0, #0C0h               ; Stall EP0

ReturnSTDGetDeviceCommand: ret


; Standard Get Endpoint Commands
;
StandardGetEndpointCommand:

        mov     A,      bRequest
        cjne    A,      #GET_STATUS,
        ReturnBadSTDGetEPCommand

                ;- GET ENDPOINT STATUS
        mov     A,      wIndex + 1
        anl     A,      #0Fh            ; Mask off all but the endpoint value
        orl     EPINDEX,A               ; Point the Index register at the

        mov     A,      wIndex + 1   ; Start by clearing out R0
        jb      ACC.7, GetlnStallStatus

GetOutStallStatus;
        mov     A,      EPCON0
        jnb     ACC.6, NotStalled
        mov     A,      #01             ; Return Stalled
        ljmp    DoneWithCommand

GetlnStallStatus:
        mov     A,
        EPCON0 jnb      ACC.7,
        NotStalled
```

```
        mov     A,      #01


DoneWithCommand:
NotStalled:
        anl     EPINDEX,#80h            ; Point the index back to EP0
        mov     TXDATO, A
        mov     TXDATO, #00h
        mov     TXCNTO, #02


ReturnBadSTDGetEPCommand: push
        EPINDEX  mov    EPINDEX,
        #01 mov     TXDAT, #13h
        mov     TXCNT,#01h pop
        EPINDEX

        orl     EPCON0, #0C0h           ; Stall EP0
ReturnSTDGetEPCommand: Ret


; for single packer Control read status
;
SetUpSinglePacketControlReadStatusStage:

        mov   wLength,      #00
        mov   wLength+1,    #00
        mov   SetupRxFlag, #STATUS_PHASE        ; Advance flag to next state
        mov   SetupTxFlag, #DATA_PHASE
        setb  TXOE                              ; Enable data transmit
        Ret



; Standard Set Endpoint Command:

StandardSetEndpointCommand:
        push    EPINDEX
        mov     A,      wIndex+l        ; Get the endpoint of the stall to clear
        anl     A,      #0Fh    ; Find out if this is an EP0 CLear Stall Command,
        orl     EPINDEX,A       ; Setup the EPINDEX to point at correct index.
```

```
        mov     A,      wValue+1
        cjne    A,      #ENDPOINT_STALL, ReturnBadSTDSetEPCommand

        mov     A,      bRequest
        cjne    A,      #CLEAR_FEATURE, CheckSetEndpointFeature

ClearEndpointFeature:

                ;- CLEAR ENDPOINT STALL
ClearEndpointStall:
        mov     A,      wIndex+l                ; Get the endpoint of the stall to
        clear
        anl     A,      #0Fh    ; Find out if this is an EP0 CLear Stall Command.
        JNZ     ClearNonEP0Stall

        anl     EPCON, #03Fh            ; For EP0 Clear both TX & RX stall bits
        ljmp    ReturnSTDSetEPCommand

ClearNonEP0Stall:
        mov     A,      wIndex+l        ; Get the endpoint of the stall to clear
        JB      ACC.7, ClearlnStall     ; For Non EP0, examining the direction
                                        ; bit as well.
ClearOutStall:
        anl     EPCON0, #CLEAR_OUT_STALL_MASK
        ljmp    ReturnSTDSetEPCommand
ClearlnStall:
        anl     EPCON0, #CLEAR_IN_STALL_MASK
        ljmp    ReturnSTDSetEPCommand


CheckSetEndpointFeature:
        cjne    A,      #SET_FEATURE, ReturnBadSTDSetEPCommand


SetEndpointFeature:

                ;- SET ENDPOINT STALL
SetEndpointStall:
        mov     A,      wIndex+l        ; Get the endpoint to stall
        JB      ACC.7, SetInStall
SetOutStall:
        orl     EPCON0, #SET_OUT_ STALL_ MASK
```

```
        ljmp
ReturnSTDSetEPCommand SetInStall:
        orl         EPCON0,
        #SET_IN_STALL_MASK ljmp
        ReturnSTDSetEPCommand

ReturnBadSTDSetEPCommand:
        push        EPINDEX mov
        EPINDEX, #01 mov
        TXDAT, #14h mov
        TXCNT,#01h pop
        EPINDEX

        orl         EPCON0, #0C0h               ;Stall EP0
ReturnSTDSetEPCommand: pop
        EPINDEX ljmp
        SetUpControlWriteStatusStage



; for standard set device commands
; StandardSetDeviceCommand:

        mov         A,          bRequest
        cjne        A,
#SET_CONFIGURATION,CheckSetDeviceAddress
SetDeviceConfiguration:

                ;- SET DEVICE CONFIGURATION ljmp

        ReturnSTDSetDevice

CheckSetDeviceAddress:
        cjne        A,          #SET_ADDRESS,
        ReturnBadSetDeviceCommand

                ;- SET DEVICE CONFIGURATION

; special case: Set Address should just return. The command will be ;
executed following the status stage ljmp         ReturnSTDSetDevice
ReturnSTDSetDevice:
        ljmp        SetUpControlWriteStatusStage

ReturnBadSetDeviceCommand:
```

```
        push   EPINDEX
        mov    EPINDEX, #01
        mov    TXDAT, #20h
        mov    TXCNT, #01h
        pop    EPINDEX

        orl    EPCON0,         #0C0h        ; Stall Endpoint
        ret

StandardSetlnterfaceCommand:
StandardGetlnterfaceCommand:
StandardSetOtherCommand:
StandardGetOtherCommand; ret


; InReceived
; device branches here when IN token received
;
InReceived:

        anl      FIS,     #EP0_TX_CLR        ; clear the interrupt bit
        ljmp     ChecklnStatusPhase

ChecklnStatusPhase:

        mov      A,       SetupTxFlag      ; read state variable
        cjne     A,       #STATUS_PHASE,SendDataBack ; Should this be the
        end to
                                                    ; a setup sequence
StatusPhaseDone:
        lCall    CompleteSetCommand
        mov      SetupTxFlag, #SETUP_PHASE          ; Set state var. to
        expect setup.
        mov      SetupRxFlag, #SETUP_PHASE

        ljmp     ReturnProcessIn

SendDataBack:
        cjne     A,       #DATA_PHASE,
        ReturnProcessIn lCall     LoadControlTXFifo

ReturnProcessIn:
        Ret
```

```
 ;CompleteSetCommand

CompleteSetCommand:
      mov        A,          bRequest        ; If this was not a Std. Set

                                             ; command then return.

      cjne       A,         #SET_ ADDRESS,

  CheckNextCommand ;- SET ADDRESS

DoSetFuncAddress:
      mov        FADDR, wValue+1        ; Set to new address
      ljmp        ReturnCompleteSet

CheckNextCommand:
ReturnCompleteSet:
ret

;
; SetUpControlWriteStatusStage
; SetUpControlWriteStatusStage:
      mov        TXCNTO, #00            ; Setup Null Packet
;            setb      TXOEO          ; Enable data transmit
      Ret

;
; LoadControlTXFifo

LoadControlTXFifo:
      push       R0
      push        DPX
      mov         DPXL,      ControlBuffLocation       ; Get location of data to
      send, mov         DPH,       ControlBuffLocation+1 mov          DPL,
      ControlBuffLocation+2

      mov         A,         ControlBuffBytesLeft      ; First check to see if any data is
                                                       ; available
      orl         A,         ControlBuffBytesLeft+1 ; to
      send, jnz         CntlDataAvail
```

```
        mov     R0,     #0              ; if there is data do normal flow
        ljmp    ControlArmTx            ; if none, do null packet

CntlDataAvail
        mov     R0,     #0              ; Number of bytes in FIFO- Always <=16
        mov     A,      DPXL
        JZ      DataInRAM


  Data is in ROM, use instructions to pull from ROM,

DataInROM:

        mov     A,      #00h

        move    A,      @A+DPTR         ; Get Data to transmit

        mov     TXDATO, A
        inc     DPTR
        inc     R0                      ; Increment the number of bytes in FIFO
        djnz    ControlBuffBytesLeft+1, ROMCheckMaxPacket       ; If not
        zero,
                                                                ; then continue.

ROMCheckUpperl:
        mov     A,      ControlBuffBytesLeft    ; If upper byte of
        wLength is also
                                                ; zero, buffer is empty.
        jz      ControlTxUpd                    ; Are we done with the buffer
        dec     A                       ; If it's not zero dec. the upper byte as well,
        mov     ControlBuffBytesLeft, A ; And store it.

ROMCheckMaxPacket:
        cjne    R0,     #EP0_MAX_PACKET_SIZE, DatalnROM ; Loop
        until
                                                        ; FIFO is Full
        ljmp    ControlTxUpd            ; Done with this FIFO.



; Data is in RAM, use instructions to pull from RAM.
; DatalnRAM:
        movx    A,      @DPTR           ; Get Data to transmit
        mov     TXDATO, A
        inc     DPTR
        inc     R0                      ; Increment the number of bytes in FIFO
```

```
    djnz    ControlBuffBytesLeft+l, RAMCheckMaxPacket       ; If not zero,
                                                            ; then continue.

RAMCheckUpper1:
    mov     A,      ControlBuffBytesLeft            ; If upper byte of
                    ; ControlBuffBytesLeft is also zero, buffer is empty,
    jz      ControlTxUpd                ; Are we done with the buffer
    dec     A                   ; If it's not zero dec. the upper byte as well.
    mov     ControlBuffBytesLeft, A         ; And store it.

RAMCheckMaxPacket:
    cjne    R0,     #EP0_MAX_PACKET_ SIZE, DataInRAM    ; Loop until
                                                        ; FIFO is Full


ControlTxUpd
    mov     ControlBuffLocation+1, DPH      ; Update read
    pointers, mov   ControlBuffLocation+2, DPL

ControlArmTx:
    mov     TXCNTO, R0          ; Write count into TXCNT register
    setb    TXOE                ; Enable data transmit
ReturnLoadCntl:
    pop     DPX
    pop     R0
    Ret



; DEVICE DESCRIPTOR
; 8x930Ax is a Big Endian Microcontroller. Words and DWords are stored with
the
; LSB in the numerlCally higher address.
; FUNCTION DESCRIPTOR

; *** Users are   required to change the descriptor accordingly ***


BEGIN_DEVICE DESCRIPTOR:
    gDevice_bLength:        db  12h     ;DEVICE DESCRIPTOR LENGTH
    gDevice_bDescriptorType:    db DEVICE_DESCR
    gDevice_bcdUSB:         dw 0001h ; Version 1.00 compliant
```

```
        ;;   gDevice_bDeviceClass:                db   00h
        gDevice_bDeviceClass:             db  04h  ;  for
        testing

        gDevice_bDeviceSubClass:                db                01h
        gDevice_bDeviceProtocol:                db                00h
        gDevice_wMaxPacketSizeO: db 08h ;8 byte max for EP0

        gDevice_widVendor:              dw 8680h          ; Intel Vendor ID
        (8086) gDevice_widProduct:              dw 4956h          ; product
        ID ;; gDevice_widProduct:             dw 8888h

        gDevice_bcdDevice:              dw 0001h ;device version 1.00
        gDevice_iManufacturer:          db 0h         ;These three fields are
        supposed
        gDevice_iProduct:          db 0h         ;to contain the index of strings
gDevice_iSerialNumber:          db 0h         ;describing device.
gDevice_bNumConfigurations: db 1
END_DEVICE_DESCRIPTOR:

;*---------- Initialize global Config descriptor ---------- */
BEGIN_CONFIG_DESCRIPTOR:
        gConfig_bLength:              db 09h
        gConfig_bDescriptorType:          db CONFIG_DESCR
        gConfig_bTotaiLength:             db
END_CONFIG_DESCRIPTOR -BEGIN_CONFIG_DESCRIPTOR
        gConfig_bCorrection:          db 0
        gConfig_bNumInterfaces:             db 1    ;
        NUM_OF_INTERFACES;
        gConfig_bConfigurationValue:     db 2
        gConfig_iConflguration:             db 0
        gConfig_bmAttributes:         db 040h
        gConfig_MaxPower:             db 025              ;50ma

;           /* ------- Initialize global Interface descriptor ..... */
BEGIN_INTERFACE_DESCRIPTOR:
        gInterface_bLength:           db END_INTERFACE_DESCRIPTOR -
BEGIN JNTERFACE^DESCRIPTOR
        gInterface_bDescriptorType:       db INTERFACE_DESCR
        gInterface_bInterfaceNumber:      db 0
        gInterface_bAlternateSetting: db 0
        gInterface_bNumEndpoints:         db 01h
        gInterface_bInterfaceClass:       db 00h
        gInterface_bInterfaceSubClass: db 00h
        gInterface_bInterfaceProtocol: db 00h
        glnterface_iInterface:              db 0
```

```
END_INTERFACE_DESCRIPTOR:

END_CONFIG_DESCRIPTOR:

STRING_LOC_TABLE:
STRING_1:          db LOW(STRING_2-STRING_1)
            db STRING_DESCR
            db "USB Test Firmware"
STRING 2:


; RAM MEMORY MAP
;- Control variables need to be placed between 20h & 7fh for the addressing
mode
;- If RISM is running, RISM needs 20h-3fH so be careful running RISM.
i
DEFINE OUR_DATA_SEG, SPACE=DATA
SEGMENT OUR DATA SEG


; Buffer and associated variables for Control Endpoint management
)
SetupTxFlag:             ds 1
SetupRxFlag:             ds 1

COMMAND_BUFFER:
StandardDeviceRequest:
bmRequestType:             ds 1
bRequest:                ds   1
wValue:
wFeatureSelector:
bDescriptorType:                 ds
1
bDescriptorlndex:                ds
1
wTargetSelector:
wIndex:                  ds 2
wLength:                 ds 2
CntlWriteDataBuffer:           ds 8

CntlWriteDataPntr:           ds 1              ; Used to point into the
CntlWriteDataBuffer.

ControlBuffLocation    ds 3
ControlBuffBytesLeft   ds 2

Beat1:                     ds 2

END
```

11.2 A WDM Driver Code Example

```
//============================
||
//    Testdrv.c
//
//    rev 1.0
//
//    USB device driver for USB Device Example
//    kernel mode driver
//
//    to be compiled with
//    - NT 4.0 DDK (MSDN DDK Jan 97 release or later)
//    - USB DDK (Aug 96 release or later)
//
//=========================================

//define DRIVER
// Include files needed for WDM driver support; from NT DDK
//include "wdm.h"
//include "stdarg.h"
//include "stdio.h"

// Include files needed for USB support; from USB DDK
//include "usbdi.h" //include "usbdlib.h" //include "usb.h"

//include "Testdrv.h"      // headers specified to this driver



//=========================================
NTSTATUS
DriverEntry (
   IN PDRIVER_OBJECT DriverObject,
   IN PUNICODE STRING RegistryPath
   ) /*
   Entry point for loading of the driver
   This is where the driver is called when the driver is being loaded
```

```
   by the I/O system.

   DriverObject - pointer to the driver object
   RegistryPath - pointer to a Unicode string representing the path
            to driver-specific key in the registry
   Return Value:
STATUS_SUCCESS if successful,
STATUS_UNSUCCESSFUL otherwise
*/

{
   NTSTATUS ntStatus = STATUS_SUCCESS;
   PDEVICE_OBJECT deviceObject = NULL;

   DriverObject->MajorFunction[IRP_MJ CREATE] = Test_Create;
   DriverObject->MajorFunction[IRP_MJ_CLOSE] = Test_Create;
   DriverObject->DriverUnload = Test_UnIoad;

   DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]=
Test_ProcessIOCTL;
   DriverObject->MajorFunction[IRP_MJ WRITE] = Test_Write;
   DriverObject->MajorFunction[IRP_MJ_READ] = Test_Read;

   DriverObject->MajorFunction[IRP_MJ_PNP_POWER] = Test Dispatch;
   DriverObject->DriverExtension->AddDevice = Test_PnPAddDevice;

   return ntStatus; }

//=================================
=
NTSTATUS
Test_Dispatch(
   IN PDEVICE OBJECT DeviceObject,
   IN PIRP          Irp
   ) /*
   Dispacthing the IRPs sent to the device.

   DeviceObject - pointer to a device object
   Irp         - pointer to an I/O Request Packet
   Return Value: NTSTATUS
*/
{
```

```
PIO_STACK_LOCATION irpStack, nextStack;
PDEVICE_EXTENSION deviceExtension;
NTSTATUS ntStatus;

Irp->IoStatus.Status = STATUS_SUCCESS;
Irp->IoStatus.Information = 0;

// Get a pointer to the current location in the Irp. This is where //
the function codes and parameters are located. irpStack =
loGetCurrentIrpStackLocation (Irp);

// Get a pointer to the device extension
deviceExtension = DeviceObject->DeviceExtension;

switch (irpStack->MajorFunction)

   { case IRP_MJ_PNP_POWER:

     // This IRP is for Plug and Play and Power Management messages for //
     your device.

      switch (irpStack->MinorFunction)
         { case IRP_MN_START_DEVICE:

// We pass the Irp down first nextStack =
IoGetNextlrpStackLocation(Irp);
ASSERT(nextStack !=NULL); RtlCopyMemory(nextStack,
irpStack, sizeof(IO_STACK_LOCATION));

            // This will be deviceExtension->StackDeviceObject in future //
            revisions of this driver
             ntStatus = IoCallDriver(deviceExtension->PhysicalDeviceObject,
Irp);

             // begin our configuration actions on the device
             ntStatus = Test_StartDevice(DeviceObject);

          break; //IRP_MN_START_DEVICE case

          IRP_MN_STOP_DEVICE: Test Cleanup

          (DeviceObject);
```

```
ntStatus = Test_StopDevice(DeviceObject); break;

//IRP_MN_STOP_DEVICE case

IRP_MN_REMOVE_DEVICE: Test_Cleanup

(DeviceObject); ntStatus =

Test_RemoveDevice(DeviceObject);

   // Delete the link to the Stack Device Object, and delete the //
   Functional Device Object we created
   IoDetachDevice(deviceExtension->StackDeviceObject);

loDeleteDevlce (DeviceObject); break;

//IRP_MN_REMOVE_DEVICE case

IRP_MN_SET POWER:

   switch (irpStack->Parameters.Power.Type)
      { case SystemPowerState: case
      DeviceSpecificPowerState:
         // TODO: Your device may need to handle these loctls.
           break; //SystemPowerState & DeviceSpecificPowerState

      case DevicePowerState:
         switch (irpStack->Parameters.Power.State.DeviceState)
            { case PowerDeviceD3:
               break; case
            PowerDeviceD2:
               break; case
            PowerDeviceD1:
               break; case
            PowerDeviceD0:
               break; } // switch on
         Power.State.DeviceState

         break;

   //DevicePowerState }// switch

   on Power.Type
```

```
        break; //IRP_MN_SET_POWER case

        IRP_MN_QUERY_POWER: // Look at what

        type of power query this is

          switch (irpStack->Parameters.Power.Type)
            { case SystemPowerState: case
            DeviceSpecificPowerState:
              // TODO: Your device may need to handle these loctls.
              break; //SystemPowerState & DeviceSpecificPowerState


            case DevicePowerState:
              switch (irpStack->Parameters.Power.State.DeviceState)
                { case PowerDeviceD2:
                  break; case
                PowerDeviceD1:
                  break; case
                PowerDeviceD3:
                  break; } //switch on
              Power.State.DeviceState

              break;

          //DevicePowerState }//switch on

          Power.Type break;

          //IRP_MN_QUERY_POWER

        case IRP_MN_QUERY_STOP_DEVICE:
          break;                                    case
        IRP_MN_QUERY_REMOVE_DEVICE:
          break;                                 case
        IRP_MN_CANCEL_STOP_DEVICE:
          break;                                    case
        IRP_MN_CANCEL_REMOVE_DEVICE:
          break;

        default:
          break;
}
```

```
nextStack = IoGetNextIrpStackLocation(Irp);
ASSERT(nextStack !=NULL);
RtlCopyMemory(nextStack, irpStack,
sizeof(IO_STACK_LOCATION));

        // All PNP_POWER messages get passed to the PhysicalDeviceObject //
        (which in future revisions of this driver will be the StackDeviceObject) //
        we were given in PnPAddDevice.

        // This will be deviceExtension->StackDeviceObject in future revisions
        //of this driver ntStatus =
           IoCallDriver(deviceExtension->PhysicalDeviceObject, Irp);

        // If lower layer driver marked the Irp as pending then reflect that by
        // calling IoMarklrpPending.
        if (ntStatus = STATUS_PENDING) {
           IoMarklrpPending(Irp); }
        else {} //if ntStatus

      goto Test_Dispatch_Done; break;

      //IRP_MJ_PNP_POWER default:

         Irp->IoStatus.Status = STATUS_INVALID_PARAMETER;

      } ntStatus =

   Irp->IoStatus.Status;

   loCompleteRequest (Irp,
               IO_NO_INCREMENT
               );

Test_Dispatch_Done:
        return ntStatus;

}


VOID Test_Unload(IN PDRIVER_OBJECT DriverObject)
```

```
/*
   Free all the allocated resources, etc.
   TODO: This is a placeholder for driver writer to add code on unload

   DriverObject - pointer to a driver object

   Return Value; None
*/
{
}
//=====================================================
NTSTATUS
Test_StartDevice(
   IN PDEVICE_OBJECT DeviceObject) /*
   Initializes a given instance of the Test Device on the USB.

   DeviceObject - pointer to the device object for this instance of a
            Test Device

   Return Value: NT status code
*/
{
   PDEVICE_EXTENSION deviceExtension;
   NTSTATUS ntStatus;
   PUSB_DEV1CE_DESCRIPTOR deviceDescriptor = NULL;
   PURB urb; ULONG siz;

   deviceExtension = DeviceObject->DeviceExtension;
   deviceExtension->NeedCleanup = TRUE;

      // Get some memory from then non paged pool (fixed, locked system //
      memory) for use by the USB Request Block (urb) for the specific USB //
      Request we will be performing below (a USB device request). urb =
      ExAllocatEP0ol(NonPagedPool,
                  sizeof(struct_URB_Control_DESCRIPTOR_REQUEST));

   if (urb) {

      siz = sizeof(USB_DEVICE_DESCRTPTOR);
```

```
    // Get some non paged memory for the device descriptor contents
        deviceDescriptor = ExAllocatepool(NonPagedPool,
                                siz);

    if (deviceDescriptor) {

// Use a macro in the standard USB header files to build the URB
UsbBuildGetDescriptorRequest(urb, (USHORT) sizeof (struct
URB_Control_DESCRIPTOR_REQUEST),
                                USB_DEVICE_DESCRJPTOR_TYPE,
                                0,
                                0,
                                deviceDescriptor,
                                NULL,
                                siz,
                                NULL);

      // Get the device descriptor
            ntStatus = Test_CallUSBD(DeviceObject, urb);

    } else {
       ntStatus =
    STATUS_NO_MEMORY; }

    if (NT_SUCCESS(ntStatus)) {
    // Put a ptr to the device descriptor in the device extension for easy // access.
    We will free this memory when the // device is removed. See the
    "Test_RemoveDevice" code. deviceExtension->DeviceDescriptor =
    deviceDescriptor; deviceExtension->Stopped = FALSE; } else if
    (deviceDescriptor) {
          // If the bus transaction failed, then free up the memory created to hold
          // the device descriptor, since the device is probably non-functional
              ExFreEP0ol(deviceDescriptor);
      }

    ExFreEP0ol(urb);

  } else {
    ntStatus = STATUS_NO_MEMORY; }
```

```
      // If the Get Descriptor call was successful, then configure the device.
      if (NT_SUCCESS(ntStatus)) {
      ntStatus = Test_ConfigureDevice(DeviceObject);
   }

   return ntStatus;
}


NTSTATUS
Test_RemoveDevice(
   IN PDEVICE_OBJECT DeviceObject
   )
/*

   Removes a given instance of a Test Device device on the USB.

   DeviceObject - pointer to the device object for this instance of a Test Device

   Return Value: NT status
*/
{
   PDEVICE_EXTENSION deviceExtension;
   NTSTATUS ntStatus = STATUS_SUCCESS;

   deviceExtension = DeviceObject->DeviceExtension;

   if (deviceExtension->DeviceDescriptor) {
      ExFreepool(deviceExtension->DeviceDescriptor);
   }

   // Free up any interface structures in our device extension if
   (deviceExtension->Interface !=NULL)
   { ExFreepool(deviceExtension->Interface);
   }

   return ntStatus;
}
// ==============================
NTSTATUS
Test_StopDevice(
   IN PDEVICE_OBJECT DeviceObject
```

```
    )
/*
    Stops a given instance of a test Device device on the USB.

    DeviceObject - pointer to the device object for this instance of a Test Device

    Return Value: NT status
*/
{
    PDEVICE_EXTENSION deviceExtension;
    NTSTATUS ntStatus - STATUS_SUCCESS;
    PURB urb; ULONG siz;

    deviceExtension = DeviceObject->DeviceExtension;

    // Send the select configuration urb with a NULL pointer for the configuration
    // handle, this closes the configuration and puts the device in the
    // 'unconfigured' state.
    siz = sizeof(struct_URB_SELECT_CONFIGURATION);

    urb = ExAllocatePool(NonPagedPool,
                    siz);

    if (urb) {
        NTSTATUS status;

        UsbBuildSelectConfigurationRequest(urb,
                                (USHORT) siz, NULL);

        status = Test CallUSBD(DeviceObject, urb);

        ExFreePooI(urb); }
    else {
        ntStatus - STATUS_NO_MEMORY; }

    return ntStatus;
}

//=============================
NTSTATUS
```

```
Test_PnPAddDevice(
   IN PDRIVER_OBJECT DriverObject,
   IN PDEVICE_OBJECT PhysicalDeviceObject
   )
/*
   This Routine is called to create a new instance of the device

   DriverObject - pointer to the driver object for this instance of Test
   PhysicalDeviceObject - pointer to a device object created by the bus

   Return Value:
   STATUS_SUCCESS if successful,
   STATUS_UNSUCCESSFUL
   otherwise
*/
{
   NTSTATUS              ntStatus = STATUS_SUCCESS;
   PDEVICE_OBJECT        deviceObject = NULL;
   PDEVICE_EXTENSION     deviceExtension;


   // create our functional device object (FDO)
   ntStatus =
      Test_CreateDeviceObject(DriverObject, &deviceObject, 0);

   if (NT_SUCCESS(ntStatus)) {
      deviceExtension = deviceObject->DeviceExtension;

      deviceObject->Flags &= ~DO_DEVICE_INITIALIZING;

      // Add more flags here if your driver supports other specific
      // behavior. For example, if your IRP_MJ_READ and IRP_MJ_WRITE
      // handlers support DIRECT JO, you would set that flag here.

      deviceExtension->PhysicalDeviceObject=PhysicalDeviceObject;

      // Attach to the StackDeviceObject. This is the device object that we //
      use to send Irps and Urbs down the USB software stack

      deviceExtension->StackDeviceObject =
         IoAttachDeviceToDeviceStack(deviceObject, PhysicalDeviceObject);

      ASSERT (deviceExtension->StackDeviceObject != NULL);
```

```
    }
    return ntStatus;
}

//=================================================
=
NTSTATUS
Test_CreateDeviceObject(
    IN PDRIVER_OBJECT DriverObject,
    IN PDEVICE_OBJECT * DeviceObject,
    LONG Instance
    )
/*
    Creates a Functional DeviceObject

    DriverObject - pointer to the driver object for device
    DeviceObject - pointer to DeviceObject pointer to return
                created device object. Instance
    - instnace of the device create.

    Return Value:
    STATUS_SUCCESS if successful,
    STATUS_UNSUCCESSFUL otherwise
*/
{
    NTSTATUS ntStatus;
    WCHAR deviceLinkBuffer[] = L"\\DosDevices\\Test-0";
    UNICODE_STRING deviceLinkUnicodeString;
    WCHAR deviceNameBuffer[] = L"\\Device\\Test-0";
    UNICODE_STRING deviceNameUnicodeString;
    PDEVICE_EXTENSION deviceExtension;

    deviceLinkBuffer[19] = (USHORT) ('0' + Instance);
    deviceNameBuffer[15] = (USHORT) ('0' + Instance);

    RtlInitUnicodeString (&deviceNameUnicodeString,
                    deviceNameBuffer);

    ntStatus =IloCreateDevice (DriverObject,
                        sizeof (DEVICE EXTENSION),
                        &deviceNameUnicodeString,
                        FILE_DEVICE_UNKNOWN, 0,
```

```
                        FALSE,
                        DeviceObject);


    if (NT_SUCCESS(ntStatus)) {
        RtlInitUnicodeString(&deviceLinkUnicodeString,
                        deviceLinkBuffer);

        ntStatus = IoCreateSymbolicLink (&deviceLinkUnicodeString,
                                &deviceNameUnicodeString);

        // Initialize our device extension
        deviceExtension = (PDEVICE_EXTENSION)
((*DeviceObject->DeviceExtension);

        RtlCopyMemory(deviceExtension->DeviceLinkNameBuffer,
                deviceLinkBuffer, sizeof(deviceLinkBuffer));

        deviceExtension->ConfigurationHandle = NULL;
        deviceExtension->DeviceDescriptor = NULL;
        deviceExtension->NeedCleanup = FALSE;

        // Initialize our interface
        deviceExtension->Interface = NULL;

    }

    return ntStatus;
}


VOID Test_Cleanup(PDEVICE_OBJECT DeviceObject)
/*
        Cleans up certain elements of the device object. This is called when the

        device is being removed from the system

        DeviceObject - pointer to DeviceObject

        Return Value: None.
 */
 {
```

```
    PDEVTCE_EXTENSION deviceExtension;
    UNICODE_STRING deviceLinkUnicodeString;

    deviceExtension = DeviceObject->DeviceExtension;

    if (deviceExtension->NeedCleanup)

    { deviceExtension->NeedCleanup = FALSE;

      RtlInitUnicodeString(&deviceLinkUnicodeString,
                    deviceExtension->DeviceLinkNameBuffer);

      IoDeleteSymbolicLink(&deviceLinkUnicodeString);
    }
}



NTSTATUS
Test_CallUSBD(
    IN PDEVICE_OBJECT DeviceObject,
    IN PURB Urb
    )
/*
    Passes a Usb Request Block (URB) to the USB class driver (USBD)

    Note that we create our own IRP here and use it to send the request to the
      USB software subsystem. This means that this Routine is essentially
      independent of the IRP that caused this driver to be called in the first
      place. The IRP for this transfer is created, used, and then destroyed in this
      Routine.

    DeviceObject - pointer to the device object for this instance of a Test Device
    Urb          - pointer to Urb request block

    Return Value:
    STATUS_SUCCESS if successful,
    STATUS_UNSUCCESSFUL otherwise
*/
{
    NTSTATUS ntStatus, status = STATUS_SUCCESS;
    PDEVICE_EXTENSION deviceExtension; P1RP irp;
```

```
KEVENT event;
IO_STATUS_BLOCK ioStatus;
PIO_STACK_LOCATION nextStack;

deviceExtension = DeviceObject->DeviceExtension;

// issue a synchronous request (see notes above)
KeInitializeEvent(&event, NotificationEvent, FALSE);

irp = IoBuildDeviceIoControlRequest(
        IOCTL_INTERNAL_USB_SUBMIT_URB,
        deviceExtension->PhysicalDeviceObject,
        NULL,
        0,
        NULL,
        0,
        TRUE, /* INTERNAL */
        &event,
        &ioStatus);

// Prepare for calling the USB driver stack
nextStack = IoGetNextIrpStackLocation(irp);
ASSERT(nextStack !=NULL);

// Set up the URB ptr to pass to the USB driver stack
nextStack->Parameters.Others.Argumentl = Urb;

// Call the USB class driver to perform the operation. If the returned status // is
PENDING, wait for the request to complete. ntStatus =
IoCallDriver(deviceExtension->PhysicalDeviceObject, irp);

if (ntStatus = STATUS_PENDING) {

   status =
            KeWaitForSingleObject( &
            event, Suspended,
            KernelMode, FALSE,
            NULL);

} else {
   ioStatus.Status = ntStatus;
```

```
    }

    II USBD maps the Error code for us. USBD uses Error codes in its URB //
    structure that are more insightful into USB behavior. To allow more insight //
    into the specific USB Error that occurred, your driver may wish to examine //
    the URB's status code (Urb->UrbHeader.Status) as well. ntStatus =
    ioStatus.Status;

    return ntStatus; }

//=====================================================
NTSTATUS
Test_ConfigureDevice(IN PDEVICE_OBJECT DeviceObject)
/*
    Configures the USB device via USB-specific device requests and interaction
       with the USB software subsystem.

    DeviceObject - pointer to the device object for this instance of the Test
Device

    Return Value: NT
    status code
*/
{
    PDEVICE_EXTENSION deviceExtension;
    NTSTATUS ntStatus;
    PURB urb = NULL;
    ULONG siz;
    PUSB_CONFIGURATION DESCRIPTOR configurationDescriptor =
NULL;

    deviceExtension = DeviceObject->DeviceExtension;

    // Get memory for the USB Request Block (urb).
    urb = ExAllocatEP0ol(NonPagedPool,
                    sizeof(struct_URB_Control_DESCRIPTOR_REQUEST));

    if (urb != NULL) {

       // Set size of the data buffer. Note we add padding to cover hardware faults
       // that may cause the device to go past the end of the data buffer siz =
       sizeof(USB_CONFlGURATION_DESCRIPTOR) + 16;
```

```
    // Get the nonpaged pool memory for the data buffer
          configurationDescriptor = ExAllocatepool(NonPagedPool,
                                    siz);

    if (configurationDescriptor !=NULL) {

       UsbBuildGetDescriptorRequest(urb,
                                (USHORT) sizeof (struct
_URB_control_DESCRIPTOR_REQUEST),
                                USB_CONFIGURATION_DESCRIPTOR_TYPE,
                                0,
                                0,
                                configurationDescriptor,
                                NULL,
                                sizeof (USB_CONFIGURATION_DESCRIPTOR),/*
Get only the configuration descriptor */
                                NULL);

       ntStatus = Test_CallUSBD(DeviceObject, urb);


    } else {
       ntStatus = STATUS_NO_MEMORY;
       goto
    Exit_TestConfigureDevice; }//if-else

    // Free up the data buffer memory just used
    ExFreepool(configurationDescriptor);
    configurationDescriptor = NULL;

    // Determine how much data is in the entire configuration descriptor //
    and add extra room to protect against accidental overrun siz =
    configurationDescriptor->wTotalLength + 16;

    // Get nonpaged pool memory for the data buffer
          configurationDescriptor = ExAllocatepool(NonPagedPool, siz);

    // Now get the entire Configuration Descriptor
    if (configurationDescriptor !=NULL) {

       UsbBuildGetDescriptorRequest(urb,
```

```
                        (USHORT) sizeof (struct
URB_Control_DESCRIPTOR_REQUEST),
                         USB_CONFIGURATION_DESCRIPTOR_TYPE,
                        0,
                        0,
                        configurationDescriptor,
                        NULL,
                        siz, // Get all the descriptor data

                        NULL);

      ntStatus = Test_CallUSBD(DeviceObject, urb); if

      (NT_SUCCESS(ntStatus)) {

      } else {
         //Error in getting configuration descriptor
         goto Exit_TestConfigureDevice;
      }//else

   } else {
      // Failed getting data buffer (configurationDescriptor) memory
      ntStatus = STATUS_NO_MEMORY;
      goto
   Exit_TestConfigureDevice; }//if-else

} else {
   // failed getting urb memory
   ntStatus = STATUS_NO_MEMORY;
   goto
Exit_TestConfigureDevice; }//if-else

// We have the configuration descriptor for the configuration //
we want.
//Now we issue the SelectConfiguration command to get // the
pipes associated with this configuration, if
(configurationDescriptor) { // Get our pipes
   ntStatus      =      Test_SelectInterfaces(DeviceObject,
                         configurationDescriptor,  NULL  //
                         Device not yet configured
                         );
}//if
```

```
Exit_TestConfigureDevice:

   // Clean up and exit this Routine
   if(urb != NULL){
   ExFreepool(urb);                    // Free urb memory
   }//if

   if (configurationDescriptor != NULL) {
      ExFreepool(configurationDescriptor);// Free data buffer
   }//if

   return ntStatus;
}


NTSTATUS
Test_SelectInterfaces(
   IN PDEVICE_OBJECT DeviceObject,
   IN PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor,
   IN PUSBD_INTERFACE_INFORMATION Interface
   ) /*
   Initializes a Test Device with multiple interfaces

   DeviceObject           - pointer to the device object for this instance of the Test
Device
ConfigurationDescriptor - pointer to the USB configuration descriptor
containing the interface and endpoint descriptors.
   Interface              - pointer to a USBD Interface Information Object
                        - If this is NULL, then this driver must choose its interface
based on driver-specific
                          criteria, and the driver must also CONFIGURE the device.
                        - If it is NOT NULL, then the driver has already been given an
interface and
                          the device has already been configured by the parent of this
device driver.

   Return Value: NT status
*/
{
   PDEVICE_EXTENSION deviceExtension;
   NTSTATUS ntStatus;
```

```
PURB urb;
ULONG siz, numberOflnterfaces, j;
UCHAR numberOfPipes, alternateSetting, MylnterfaceNumber;
PUSB_INTERFACE_DESCRIPTOR interfaceDescriptor;
PUSBD_INTERFACE_INFORMATION interfaceObject;

deviceExtension = DeviceObject->DeviceExtension;
   MylnterfaceNumber = SAMPLE_ INTERFACE_NBR;

if (Interface ==NULL) {

   // This example driver only supports one interface. This can be extended //
   to be a dynamlCally allocated array by your driver. numberOflnterfaces =
   ConfigurationDescriptor->bNumInterfaces;

         numberOflnterfaces =1;
   numberOfPipes = 0;          // Initialize to zero

   // We use alternate interface setting 0 for all interfaces
   // This is a simplification and is due to change in future releases of this
   // driver. If your driver supports alternate settings, you will have to do
   // more work to switch between alternate settings.
   alternateSetting = 0;

   // Call a USBD helper function that returns a ptr to a USB Interface //
   Descriptor given a USB Configuration Descriptor, an Inteface Number, //
   and an Alternate Setting for that Interface interfaceDescriptor =
      USBD_ParseConfigurationDescriptor(ConfigurationDescriptor,
                              MylnterfaceNumber, //interface number (this is
blnterfaceNumber from interface descr)
                              alternateSetting);

   ASSERT(interfaceDescriptor != NULL);

   // Add to the tally of pipes in this configuration
   numberOfPipes += interfaceDescriptor->bNumEndpoints;

   // Now that we have looked at the interface, we configure the device so that
   // the remainder of the USBD objects will come into existence (ie., pipes, //
   etc.) as a result of the configuration, // thus completing the configuration
   Process for the USB device.
   //
```

```
    II Allocate a URB big enough for this Select Configuration request

    siz =
GET SELECT_CONFIGURATION_REQUEST_SIZE(numberOflnterfaces,
numberOPipes);

    urb = ExAllocatepool(NonPagedPool,
                siz);

    if (urb) {
      interfaceObject = (PUSBD_INTERFACE_INFORMATION)
(&(urb->UrbSelectConfiguration.Interface));

      // set up the input parameters in our interface request structure.
      interfaceObject->Length =
          GET_USBD_INTERFACE_SIZE(interfaceDescriptor->b
NumEndpoints);

      interfaceObject->InterfaceNumber =
interfaceDescriptor->bInterfaceNumber;
      interfaceObject->AlternateSetting =
interfaceDescriptor->bAlternateSetting;
      interfaceObject->numberOPipes =
interfaceDescriptor->bNumEndpoints;

    // We set up a default max transfer size for the endpoints. Your driver will //
    need to change this to reflect the capabilities of your device's endpoints. for
    (j=0; j<interfaceDescriptor->bNumEndpoints; j++)
    { interfaceObject->Pipes[j].MaximumTransferSize =
          USBD_DEFAULT_MAXIMUM_TRANSFER_SIZE;
      }

      UsbBuildSelectConfigurationRequest(urb,
                          (USHORT) siz,
                          ConfigurationDescriptor);

      ntStatus = Test_CallUSBD(DeviceObject, urb);

      if (NT_SUCCESS(ntStatus) &&
USBD_SUCCESS(urb->UrbSelectConfiguration.Status)) {

        // Save the configuration handle for this device
        deviceExtension->ConfigurationHandle =
```

```
                urb->UrbSelectConfiguration.ConfigurationHandIe;

            deviceExtension->Interface = ExAllocatePool(NonPagedPool,
                                                interfaceObject->Length);

            if (device£xtension->Interface) {

                RtlCopyMemory(deviceExtension->Interface, interfaceObject,
interfaceObject->Length);

                // Dump the pipe info
                for (j=0; j<interfaceObject->NumberOfPipes; j++)
                   { PUSBD_PIPE_INFORMATION
                   pipeInformation;

                   pipeInformation = &deviceExtension->Interface->Pipes[j]; }

            }

         }// if selectconfiguration request was successful }

      else {

      ntStatus = STATUS_NO_MEMORY; }//if

   urb alloc passed }//if Interface was not

   NULL return ntStatus;

}

//=============================================================
NTSTATUS
Test_Read(
   IN PDEVICE_OBJECT    DeviceObject,
   IN PIRP Irp
   )
/*
   This function is called for a IRP_MJ_READ.
   TODO: Add functionality here for your device driver if it handles that IRP
code.
```

```
   DeviceObject - pointer to the device object for this instance of the Test
device.
   Irp          - pointer to IRP

   Return Value: NT status
*/
{
      NTSTATUS ntStatus = STATUS_SUCCESS;
      return (ntStatus);




//=================================================
NTSTATUS
Test_Write(
   IN PDEVICE_OBJECT DeviceObject,
   IN PIRP Irp
   )
/*
   This function is called for a IRP_MJ_WRITE.
   TODO: Add functionality here for your device driver if it handles that IRP
code.

   DeviceObject - pointer to the device object for this instance of the Test device.
   Irp                - pointer to IRP

   Return Value: NT status
*/


      NTSTATUS ntStatus = STATUS_SUCCESS;
      return (ntStatus);




//=========================
NTSTATUS
Test_Create(
   IN PDEVICE_OBJECT DeviceObject,
   IN PIRP Irp
   )
```

```
/*
    This is the Entry point for CreateFile calls from user mode apps (apps may
open "\\.\Test-x\yyzz"
    where yy is the interface number and zz is the endpoint address).

    Here is where you would add code to create symbolic links between
endpoints
    (i.e., pipes in USB software terminology) and User Mode file names. You are
    free to use any convention you wish to create these links, although the above
    convention offers a way to identify resources on a device by familiar file and
    directory structure nomenclature.

    DeviceObject - pointer to the device object for this instance of the Test
device

    Return Value: NT status
*/
{ NTSTATUS ntStatus;

   Irp->IoStatus.Status - STATUS_SUCCESS;
   Irp->IoStatus.Information = 0;

      // Create all the symbolic links here
   ntStatus = Irp->IoStatus.Status;

   loCompleteRequest (Irp,
                IO_NO_INCREMENT
                );

   return ntStatus;

}


NTSTATUS Test_Close(
   IN PDEVICE OBJECT DeviceObject,
   IN PIRP Irp
   ) /*
   Entry point for CloseHandle calls from user mode apps to close handles they
have opened
```

```
   DeviceObject - pointer to the device object for this instance of the Test device
   Irp          - pointer to an irp

   Return Value: NT status
*/
{ NTSTATUS ntStatus - STATUSJSUCCESS;

      return ntStatus;

}
}
//=================================================
NTSTATUS
Test_ProcessIOCTL(
   IN PDEVICE_OBJECT   DeviceObject,
   IN PIRP Irp
   )
/*
   This is where all the DeviceIoControl codes are handled. You can expand
and add more code
   here to handle IOCTL/IRP codes that are specific to your device driver.

   DeviceObject - pointer to the device object for this instance of the test device.

   Return Value: NT status
*/
{
   PIO_STACK_LOCATION irpStack; PVOID
   ioBuffer; ULONG inputBufferLength;
   ULONG outputBufferLength;
   PDEVICE_EXTENSION deviceExtension;
   ULONG ioControlCode; NTSTATUS ntStatus;
   ULONG length; PUCHARpch;

   // Get a pointer to the current location in the Irp. This is where //
   the function codes and parameters are located. irpStack =
   IoGetCurrentIrpStackLocation (Irp);
```

```
    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;

    // Get a pointer to the device extension
    deviceExtension = DeviceObject->DeviceExtension;

    ioBuffer          = Irp->AssociatedIrp.SystemBuffer;
    inputBufferLength                                =
irpStack->Parameters.DeviceIoControl.InputBuffe
rLength;
    outputBufferLength                                =
irpStack->Parameters.DeviceIoControl.OutputBuffe
rLength;

    ioControlCode = irpStack->Parameters.DeviceIoControl.IoControlCode;

    // Handle loctls from User mode
    switch (ioControlCode) {

    case    IRP_Test_GET_PIPE_INFO:
       // inputs - none
       // outputs • we copy the interface information structure that we have
       //          stored in our device extension area to the output buffer which
       //          will be reflected to the user mode application by the IOS.
       length = 0; pch =
       (PUCHAR) ioBuffer;

       if (deviceExtension->Interface)
          { RtlCopyMemory(pch+length,
                    (PUCHAR) deviceExtension->Interface,
                     deviceExtension->lnterface->Length);

          length +=
       deviceExtension->Interface->Length; }


       Irp->IoStatus.Information = length;
       Irp->IoStatus.Status = STATUS_SUCCESS;

       break;

      case IRP_Test_GET_DEVICE_DESCRIPTOR:
       // inputs - pointer to a buffer in which to place descriptor data
       // outputs - we put the device descriptor data, if any is returned by the
       // device in the system buffer and then we set the length in the Information
```

```
        II field in the Irp, which will then cause the system to copy the buffer back //
        to the user's buffer

        length = Test_GetDeviceDescriptor (DeviceObject, ioBuffer);

        Irp->IoStatus.Information = length;
        Irp->IoStatus.Status = STATUS_SUCCESS;

        break;

    caseIRP_Test_GET_CONFIGURATION__DESCRIPTOR:

        // inputs - pointer to a buffer in which to place descriptor data
        // outputs - we put the configuration descriptor data, if any is returned by
        // the device in the system buffer and then we set the length in the
        // Information field in the Irp, which will then cause the system to copy the
        // buffer back to the user's buffer

        length = Test_GetConfigDescriptor (DeviceObject, ioBuffer,
outputBufferLength);

        Irp->IoStatus. Information = length;
        Irp->IoStatus.Status = STATUS_SUCCESS;

        break;


    default:

        Irp->IoStatus.Status =
    STATUS_INVALID_PARAMETER; }// switch on
    ioControlCode

    ntStatus = Irp->IoStatus.Status;

    loCompleteRequest (Irp,
                    IO_NO_ INCREMENT
                    );

    return ntStatus;

}
```

```
ULONG
Test_GetDeviceDescriptor(
   IN PDEVICE_OBJECT DeviceObject,
   PVOID             pvOutputBuffer
   )
/*
   Gets a device descriptor from the given device object

   DeviceObject - pointer to the test device object

   Return Value: Number of valid bytes in data buffer
*/
{
   PDEVICE_EXTENSION      deviceExtension = NULL;
   NTSTATUS               ntStatus        = STATUS_SUCCESS;
   PURB                   urb             =NULL;
   ULONG                  length          = 0;

   deviceExtension = DeviceObject->DeviceExtension;

   urb = ExAllocatepool(NonPagedPool,

                  sizeof(struct_URB_Control_DESCRIPTOR_REQUEST));

   if (urb) {

      if (pvOutputBuffer) {

         UsbBuildGetDescriptorRequest(urb,
                        (USHORT) sizeof (struct
_URB_Control_DESCRIPTOR_REQUEST),
                     USB_DEV1CE_DESCRIPTOR_TYPE,       //descriptor type
                       0,                        //index
                       0,                        //language ID
                       pvOutputBuffer,               //transfer buffer
                       NULL,                     //MDL
                       sizeof(USB_DEVICE DESCRIPTOR), //buffer length
                       NULL);                    //link

         ntStatus = Test_CallUSBD(DeviceObject, urb);

      } else {
         ntStatus = STATUS_NO_MEMORY; }
```

```
      II Get the length from the Urb
      length = urb->UrbControlDescriptorRequest.TransferBufferLength;

      ExFreepool(urb);

   } else { ntStatus =
      STATUS_NO_MEMORY;
   }

   return length;
}


//=============================================================
/* not tested it yet
NTSTATUS       SetOne(lN PDEVICE_OBJECT pDeviceObject,
PUSBTESTCTRLDESC pTestCtrlDesc)

//       perform a "set" setup packet
//
//         Return Value: ntstatus


 {
  NTSTATUS ntStatus;
  PURB          pUrb;

pTemp=ExAllocatepool(NonPagedPool, sizeof(USBDISPDESC));
pUrb=ExAllocatepool(NonPagedPool, sizeof(struct
_URB_Control_VENDOR_OR_CLASS_REQUEST));

  if (pUrb)   { RtICopyMemory(pTemp, pTestCtrlDesc,

   sizeof(USBTESTDESC));

   UsbBuildVendorRequest(pUrb,
URB_FUNCTION_CLASS_DEVICE,     //Command sizeof(struct
_URB Control_VENDOR_OR_CLASS_REQUEST),
                    !(USBD_TRANSFER_DIRECTION_IN),// Direction Bit: OUT
                    0,                        // Reserved Bits !
                    0x04,                       //Request (OxF1)
                    pTestCtrlDesc->usControlCode, // Value      (OxF0)
```

```
                    0x0,                    // Index      (0x00)
                    pTemp,                    // Transfer Buffer
                    NULL,                     // Transfer Buffer MDL
                     0x08,                   // Transfer buffer length: Size of display
                                            // descriptor.
                    NULL);

    ntStatus=MakeUSBDCall(pDeviceObject,pUrb);

    ExFreepool(pUrb);
   }

  ExFreepool(pTemp);
  return ntStatus;
}
*/



//=========================================================
ULONG
Test_GetConfigDescriptor(
    IN PDEVICE_OBJECT DeviceObject,
    PVOID             pvOutputBuffer,
    ULONG             ulLength
    )
/*
    Gets configuration descriptors from the given device object

    DeviceObject      - pointer to the test device object
    pvOutputBuffer - pointer to the buffer where the data is to be placed
    ulLength              - length of the buffer

    Return Value: Number of valid bytes in data buffer
*/
{
    PDEVICE_EXTENSION    deviceExtension = NULL;
    NTSTATUS             ntStatus             =
    STATUS_SUCCESS;
    PURE                 urb            =NULL;
    ULONG                length                 = 0;

    deviceExtension = DeviceObject->DeviceExtension;

    urb = ExAllocatepool(NonPagedPool,
                         sizeof(struct_URB_control_DESCRIPTOR_REQUEST));
```

```
   if (urb){ if

      (pvOutputBuffer) {

         UsbBuildGetDescriptorRequest(urb,
                              (USHORT) sizeof (struct
_URB_CONTROL_DESCRIPTOR_REQUEST),
                              USB_CONFIGURATION_DESCRIPTOR_TYPE,
//descriptor type
                              0,                    //index
                              0,                    //language ID
                              pvOutputBuffer,             //transfer buffer
                              NULL,                 //MDL
                              ulLength,             //buffer length
                              NULL);                //link

         ntStatus = Test_CallUSBD(DeviceObject, urb);

      } else {
         ntStatus = STATUS_NO_MEMORY; }

      // Get the length from the Urb
      length - urb->UrbControlDescriptorRequest.TransferBufferLength;

      ExFreePool(urb);

   } else { ntStatus =
      STATUS_NO_MEMORY;
   }

   return length;
}
```

11.3 An Application Software Code Example

```
//=======================================
// MainFrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
//include "Test_app.h"
include "MainFrm.h"

#ifdef_DEBUG
//define new DEBUG_NEW
//undefTHIS_FILE
static char THIS_FILE[] = _FILE_;
#endif


// CMainFrame IMPLEMENT_DYNCREATE(CMainFrame,

CFrameWnd)

BEGIN MESSAGE_MAP(CMainFrame, CFrameWnd)
        //{{AFX MSG_MAP(CMainFrame)
        ON_WM_CREATE()

        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

extern CUsbappApp theApp;

static UINT indicators[] =
{
        ID_SEPARATOR,          // status line indicator
        ID_INDICATOR_CAP
        S,
        ID_INDICATOR_NUM,
        ID_lNDICATOR_SCRL,
};
```

```
// CMainFrame construction/destruction
CMainFrame::CMainFrame()
{
        // TODO: add member initialization code here
}

CMainFrame::~CMainFrame()
        {}


int CMainFrame::OnCreate(LPCREATESTRUCT   IpCreateStruct)
{
        CString             DevName;

        CMenu *             pcMenu, * pcSubMenu;
        int uDevCnt=0, uCnt=0, nTT=0;

        if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
                return -1;

        if (!m_wndToolBar.Create(this) ||
                !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
        {
                TRACEO("FaiIed to create toolbar\n);
                return -1;          // fail to
        create }

        if(!m_wndStatusBar.Create(this) ||
                !m_wndStatusBar.SetIndicators(indicators,
                  sizeof(indicators)/sizeof(UINT)))
        !
                TRACE0("Failed to create status bar\n");
                return -1;          // fail to create
        }

        // TODO: Remove this if you don't want tool tips or a resizable toolbar
        m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
                CBRS_TOOLTIPS | CBRS_FLYBY |
CBRS_SIZE_DYNAMIC);

        // TODO: Delete these three lines if you don't want the toolbar to
        // be dockable
```

```
        m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
        EnableDocking(CBRS_ALIGN ANY);
        DockControlBar(&m_wndToolBar);

   if((pcMenu=GetMenu()) !=NULL)
        { pcSubMenu=pcMenu->GetSubMenu(0);

                DevName.Format("\\\\.\\Test-0");

                hDrvrHnd=CreateFile(DevName,
                                                GENERIC_WRITE |
GENERIC_READ,
                                                FILE_SHARE_WRITE |
FILE_SHARE_READ,
                                                NULL,
                                                OPEN_EXISTING,
                                                0,
                                                NULL);

                // enable the menu items if successful...
                if (hDrvrHnd != INVALID_HANDLE_VALUE)
                        {
                        theApp.hDrvrHnd=hDrvrHnd;
                        theApp.OnlnitTest();
                        pcMenu->EnableMenu!tem(0,
MF_ENABLED|MF BYPOSITION);
                        pcMenu->EnableMenu!tem( 1,
MF_ENABLED|MF_BYPOSITION);
                        pcMenu->EnableMenu!tem(2,
MF_ENABLED|MF BYPOSITION);
                        pcMenu->EnableMenu!tem(3,
MF_ENABLED|MF_BYPOSITION);
       pcMenu->EnabIeMenu!tem(4,
MF_ENABLED|MF_BYPOSITION); }

        }

        DrawMenuBar();
        return 0;

}
```

```
//=====================================
BOOL CMainFrame:;PreCreateWindow(CREATESTRUCT& cs)
{
        // TODO: Modify the Window class or styles here by modifying
        // the CREATESTRUCT cs
        return CFrameWnd::PreCreateWindow(cs);
}


// CMainFrame diagnostics
#ifdef_DEBUG
void CMainFrame: :AssertValid() const
{
        CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
        CFrameWnd::Dump(dc);
}

#endif // DEBUG


// CMainFrame message handlers
void CMainFrame::OnDevdesc()
{

        // TODO: Add your command handler code here

}

void CMainFrame::OnConfdesc()
{

        // TODO: Add your command handler code here

}

void CMainFrame::OnAlldesc()
{

        // TODO: Add your command handler code here
```

```
}

void CMainFrame::OnStrdesc()
{
        // TODO: Add your command handler code here

}

void CMainFrame::OnInfdesc()
{
        // TODO: Add your command handler code here

}

void CMainFrame::PostNcDestroy()
{
        // TODO: Add your specialized code here and/or call the base class

        CFrameWnd::PostNcDestroy();
}
```

```
// test_app.cpp: Implement the class behaviors for the application software
//rev 1.0
//


^include "stdafx.h"
^include "Test_app.h"
^include "MainFrm.h"
//include "T_appDoc.h"
^include "T_appView.h"

^include "main.h"
#include "devioctl.h"
^include "testdrv.h"

#ifdef_DEBUG
^define new DEBUG_NEW
#undefTHIS_FILE
static char THIS_FILE[] = _FILE_;
#endif



BEGIN_MESSAGE_MAP(CUsbappApp,CWinApp)
        //{{AFX_MSG_MAP(CUsbappApp)
        ON_COMMAND(ID_APP_ABOUT,OnAppAbout)

        // added here as well for set and get

        ON_COMMAND(ID_GETDES_DEVICE, OnGetdesDevice)

        ON_COMMAND(ID_SET_ONE, OnSetOne)
        ON_COMMAND(ID_SET_TWO, OnSetTwo)

        ON_COMMAND(ID_GET_ONE, OnGetOne)
        ON_COMMAND(ID_GET_TWO, OnGetTwo)

  //}}AFX_MSG_MAP

        // Standard file based document commands
```

```
ON_COMMAND(ID_FILE_NEW,CWinApp::OnFiIeNew)
ON_COMMAND(ID_FILE_OPEN,CWinApp::OnFileOpen) //
Standard print setup command
ON_COMMAND(ID_FILE_PRINT_SETUP,
CWinApp::OnFilePrintSetup)

END_MESSAGE_ MAP()



CUsbappApp::CUsbappApp()
{
        // TODO: add construction code here,
        // Place all significant initialization in Initlnstance
}


CUsbappApp theApp;

BOOL CUsbappApp::InitInstance()
{
        // Standard initialization
        // If you are not using these features and wish to reduce the size // of
        your final executable, you should remove from the following // the
        specific initialization routines you do not need.

#ifdef_AFXDLL
        Enable3dControls();             // Call this when using MFC in a shared
DLL #else
        Enable3dControlsStatic(); // Call this when linking to MFC statically
#endif

        LoadStdProfileSettings(); // Load standard INI file options //
                                  (including MRU)

    // Register the application's document templates. Document templates //
    serve as the connection between documents, frame windows and views.

        CSingleDocTemplate* pDocTemplate;
        pDocTemplate = new CSingleDocTemplate(
                IDR_MAINFRAME,
                RUNTIME_CLASS(CUsbappDoc),
                RUNTIME_CLASS(CMainFrame),
                                        // main SDI frame window
```

```
                RUNTIME_CLASS(CUsbappView));
        AddDocTemplate(pDocTemplate);

        // Parse command line for standard shell commands, DDE, file open
        CCommandLinelnfo cmdlnfo;
        ParseCommandLine(cmdlnfo);

        // Dispatch commands specified on the command line
        if (!ProcessShellCommand(cmdlnfo)) return FALSE;

        return TRUE;
}


// CAboutDlg dialog USBD for App About
class CAboutDlg : public CDialog
{ publi
c:
        CAboutDlg();

// Dialog Data
        //{{AFX_DATA(CAboutDlg) enum
        {IDD = IDD_ABOUTBOX };
        //}}AFX_DATA

        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CAboutDlg)
        protected:
        virtual void DoDataExchange(CDataExchange* pDX);
                                                    // DDX/DDV support
        //}}AFX_VIRTUAL

// Implementation
protected:
        //{(AFX_MSG(CAboutDlg)
                // No message handlers
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

CAboutDIg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
```

```
        //{{AFX_DATA INIT(CAboutDlg)
        //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX DATA_MAP(CAboutDlg)
        //}}AFX_DATA_MAP }

BEGIN_MESSAGE_MAP(CAboutDlg,CDia1og)

        //{{AFX_MSG_MAP(CAboutDlg)
        // No message handlers
//}}AFX_MSG_MAP END_MESSAGE_MAP()

// App command to run the dialog
void CUsbappApp::OnAppAbout()
{
        CAboutDlg aboutDlg;
        aboutDlg.DoModal();
}



// CUsbappApp commands
int CUsbappApp::ExitInstance()
{
        // TODO: Add your specialized code here and/or call the base class
   CloseHandle(hDrvrHnd);
        free(pTestDesc);
        return
CWinApp::ExitInstance(); }


void CUsbappApp: :OnGetdesDevice()
{
        PVOID pvBuffer                  =0;
        pvBuffer = malloc (sizeof (Usb_Device_Descriptor) + 64);

        // Perform the Get-Descriptor IOCTL
```

```
   DeviceloControl (hDrvrHnd,
                IRP_Test_GET_DEVICE_DESCRIPTOR,
                pvBuffer,
                sizeof (Usb_Device_Descriptor),
                pvBuffer,
                sizeof (Usb_Device_Descriptor),
                &dwStatus,
                NULL);

          free(pvBuffer);
          AfxMessageBox("completed 'Get Descriptors - Device' ");

}


void CUsbappApp::OnGetOne()
{
          PVOID pvBuffer                    =0;
          pvBuffer = malloc (sizeof (Usb_Device_Descriptor) +
          64);

          // Perform the Get-Descriptor IRP
   DeviceloControl (hDrvrHnd,

IRP_Test GET_CONFIGURATION_DESCRIPTOR,
                                              pvBuffer,
                sizeof (Usb_Device_Descriptor),
                pvBuffer,
                sizeof (Usb_Device_Descriptor),
                &dwStatus, NULL);

          free(pvBuffer);
          AfxMessageBox("completed 'Get Command - One' ");

}


void CUsbappApp::OnGetTwo()
{
          PVOID pvBuffer                    =0;
          pvBuffer = malloc (sizeof (Usb_Device_Descriptor) + 64);

          // Perform the Get-Descriptor IRP
```

```
   DeviceIoControl (hDrvrHnd,

IRP_Test_GET_CONFIGURATION_DESCRIPTOR,
                                                pvBuffer,
                sizeof (Usb Device_Descriptor),
                pvBuffer,
                sizeof (Usb_Device_Descriptor),
                &dwStatus, NULL);

          free(pvBuffer);
          AfxMessageBox("completed 'Get Command - Two' ");
}



void CUsbappApp::OnSetOne() //
not implemented yet
{
                AfxMessageBox("This function has not been implemented ");
}

/*{
          PVOID pvBuffer                    =0;
          pvBuffer = malloc (sizeof (Usb_Device_Descriptor) + 64);

          // Perform the Get-Descriptor IRP
   DeviceloControl (hDrvrHnd,
                                                IRP TEST_GETTWO,
                                                pvBuffer,
                sizeof (Usb_Device_Descriptor),
                pvBuffer,
                sizeof (Usb_Device_Descriptor),
                &dwStatus, NULL);

          free(pvBuffer);
          AfxMessageBox("completed 'Set Command - One' ");
}*/


void CUsbappApp::OnSetTwo()
// not implemented yet
```

```
{
                AfxMessageBox(" This function has not been implemented ");
}



BOOL CUsbappApp::InitApplication()
{
        // TODO: Add your specialized code here and/or call the base class

        return CWinApp::InitApplication();
}



void CUsbappApp::OnInitTest()
{

    if (hDrvrHnd == INVALID_HANDLE_VALUE) {
      MessageBox(NULL, "Failed to Locate USB Tset Devices", "Error",
MB_OK);
                        }
    else { if(pTestDesc !=
      NULL)
                                {
          free(pTestDesc);
                                }
  //pTestDesc=malloc(sizeof(USBTESTCTRLDESC)); //
  memset(pTestDesc, 0, sizeof(USBTESTCTRLDESC));

  }

}
```

12. Index

r

U

V

W