CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

MICRO SERIES

Mike Zerkus, John Lusher, & Jonathan Ward

USB Primer

Practical Design Guide

Part 1 of 4

Before getting into the nitty-gritty of working on Universal Serial Bus projects, you need to know the basics. But Mike, John, and Jon offer more than an intro to USB. Their demo gets you ready to work on your own.

Universal serial bus (USB) promises to be the next major advance in PC functionality, completing the PC's transition to a plug-and-play system. But, for all its possibilities, USB is bit of a mystery.

For the average engineer with an idea for a USB product or who has been commanded to convert an existing system, the journey to enlightenment can be an arduous struggle. Rather than merely providing information on USB, we want to show you how to get your USB device up and running.

As a high-speed bus for connecting external devices to the PC, USB is the next step for external peripherals on Windows and Macintosh computers. By allowing hot-plug installation, reconfiguration becomes less of a hassle.

USB enables 127 devices to be on the bus simultaneously. This arrangement solves the problem of limited serial ports.

USB operates at 12 Mbps (there is a low-speed mode of 1.5 Mbps for some devices), and it supports isochronous and asynchronous data transfers. Because USB devices can be bus powered, the transformer ganglion behind the computer can be reduced.

PC users now have a simple user-friendly peripheral bus that supports up to 127 devices and that can be installed without configuring or altering their current system. Gone are the days of figuring out which interrupt settings and I/O addresses were available and altering the device's settings to fit the available resources.

With USB, you just plug the device into the port. The OS takes care of the rest. There are no jumpers, power packs, powerdowns, resets, or taking the case off. The PC automatically installs the appropriate driver and configures the device as needed.

HOW DOES USB WORK?

RS-232 serial communication with UARTs, transfer rates, stop bits, and so on traces its heritage back to mechanical devices in the days of teletype. In the heyday of TTY, you could repair a UART with a wrench. Adjusting the transfer rate was more like tuning a car than working on electronics.

USB doesn't represent an electronic analog of a mechanical system. In a USB system, the line between hardware and software function is blurred. USB exploits the full potential of a computerized communication system.

The two sides of a USB system are the device and the host. The device side consists of the USB device (e.g., modem or printer), which usually contains a USB microcontroller (e.g., the Intel '930) and the code to properly initiate USB communication to the host. The host side is the PC running an OS that supports USB. The device and host communicate over the USB cable.

USB devices can be self-powered or bus powered, so they can be produced without including a bulky wall-mount

transformer. The device gets its power from the host computer or USB hub.

BUS TOPOLOGY

USB uses a tiered/star bus topology in which each device plugs into a hub. The hub is a traffic cop that enforces the low-level rules of the bus. Figure 1 shows the physical arrangement of a USB system. For the most part, hubs are transparent.

Classes are the device categories that share common I/O requirements. In USB there are currently 11 classes: common class, audio, communications, hub, human interface device (HID), image, monitor, physical interface device (PID), power, printer, and storage.

Classes introduce a set of standard drivers native to the OS (Windows 98) and enable you to use them as is, write your own driver, or have a mini-driver.

PACKETS

A packet is a combination of special fields. All packets begin with the Sync field to synchronize the phase-locked loop and have a packet identifier (PID) that helps USB devices determine the kind of packet being sent. The packet is followed by address information, a frame number, or data. There are four types of packets; each has several subtypes.

The first packet type—the token, shown in Figure 2a—is a 24-bit data packet that represents what is happening over the bus. The first eight bits represent the packet identifier. The next seven bits are the address of the device that the host is communicating with. The next four bits are the endpoint address, which is where the data is going in the device. And, the last five bits are the CRC to check the token for errors.

There are four types of tokens—In, Out, Start of Frame (SOF), and Setup. Check the glossary of terms in Design Forum for more details. An SOF packet is illustrated in Figure 2b.

As you see in Figure 2c, data packets contain PIDs for further data error-checking. Data packets alternate between DATA0 and DATA1. The only exception to this format is the Setup packet, which always uses the DATA0 packet.

Data packets have a format of the DATA0/1 PID followed by the data, which ranges in length from 0 to 1023 bytes. The packet is checked with a 16-bit CRC field.

The handshake packet is shown in Figure 2d. These packets inform the sender of the data as to the condition of the received data packet. Handshake packets are ACK, NAK, and STALL.

The special preamble packet establishes low-speed communication on the bus. This token is sent full speed to the hubs, and the hubs then enable their low-speed outputs.

DESCRIPTORS

The descriptor includes general information about the device. The Vendor ID and Product ID fields play the key role in the enumeration of the device. The descriptor also informs the host about the number of configurations of the device.

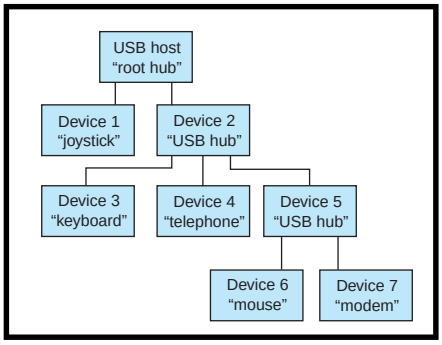

Figure 1—*This diagram shows the physical arrangement of a USB system.*

Configuration descriptors tell the host the number of interfaces, the device's power requirements, and its attributes. Interface descriptors are the number of endpoints and what class they belong to as well as the interface protocol.

Endpoint descriptors describe the direction and attributes of the endpoints belonging to a specific interface, including the address of endpoint, direction of endpoint, attribute of endpoint, and maximum packet size.

DATA TRANSFERS

A transfer or transaction consists of a number of packets moving back and forth along the bus between the host and a device. There are four types of data transfers in a USB system:

- control—controls the bus, bidirectional, setup, data, status
- bulk—asynchronous

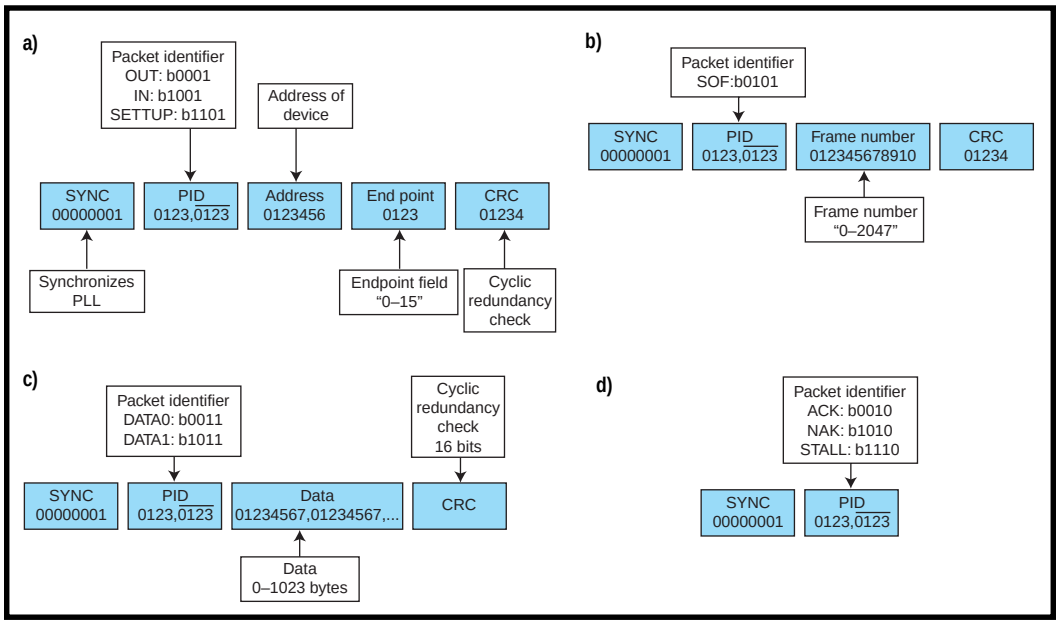

Figure 2—*These diagrams show four different types of packets: token (**a**), SOF (**b**), data (**c**), and handshake (**d**).*

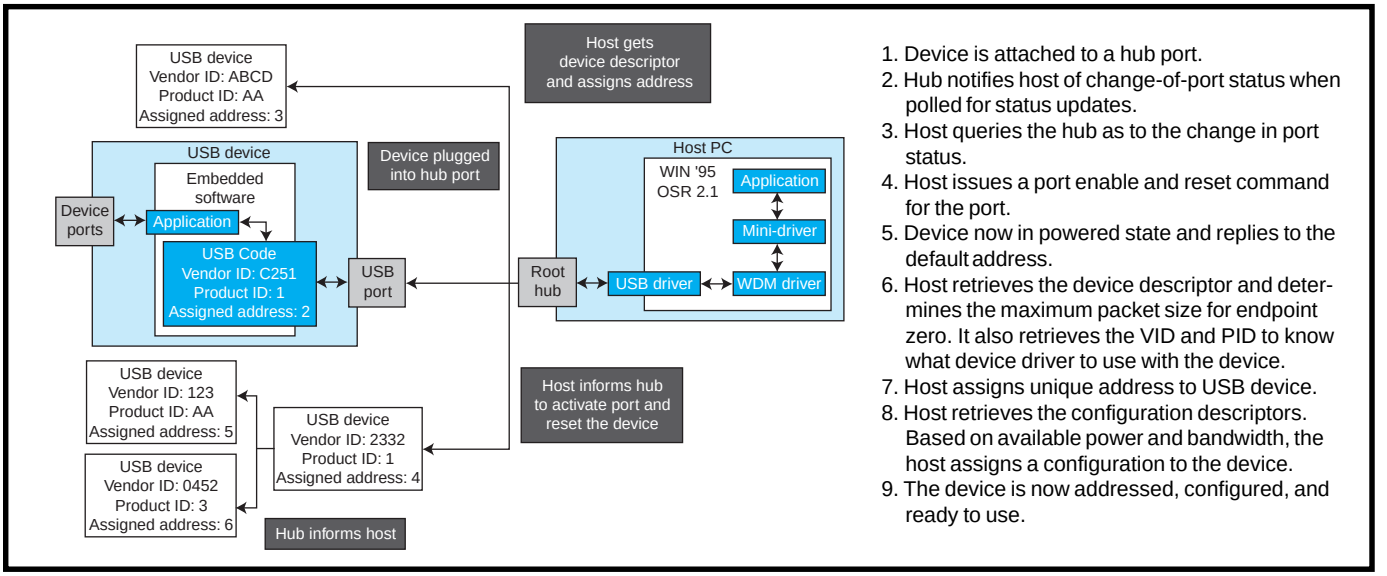

Figure 3—*This diagram and the accompanying list illustrate the enumeration of a USB device.*

data, bidirectional, CRC
- isochronous—time-critical data, no CRC, unidirectional, up to 1023 bytes per frame, guaranteed bandwidth per frame
- interrupt—receives data at timed intervals, input only, 1–255-ms intervals

Enumeration is the bus-configuration process, which takes place anytime the bus is started or a device is plugged into or unplugged from the bus. This process is shown in Figure 3.

The whole USB system is not provided by any one vendor. The OS provides some parts; other parts come from third parties and the developer.

For the next few years, most USB development projects will have to function with both Windows 95 and 98. There are some key items to be aware of when using USB with Windows 95.

Windows 95 doesn't have native USB support. You must have OSR 2.1 build 1214 or better installed on the system.

Windows 95 also has some minor bugs. One such bug is when the USB device has no alternate settings. If this occurs, Windows 95 freezes up when the device is unplugged.

Windows 98 handles USB right out of the box and resolves the above-mentioned bug. It also has a program to assist in developing USB peripherals. `usbview.exe` enables you to monitor the activity on the USB bus as well as get the device descriptors.

END-TO-END EXAMPLE

As promised, here's an example of how to get a USB device working. Using a commercially available evaluation board, the goal of this system is to blink LEDs on the eval board from the PC and to blink indicators on the PC screen from the eval board.

For this project, you need the Anchor Chips EZ-USB evaluation kit V.C or better, the USB specification, Windows PC with USB support, Windows 95 (OSR 2.1 build 1214 or better) or Windows 98, and Visual C++ or Visual Basic (V.5.0 or better). To produce drivers, you need Windows 98 and the Windows 98 DDK.

We hooked up Port A of the Anchor Chips device to a switch/LED circuit and created a DLL using the driver's `IOCTL` functions. A VB program calls the DLL and gets the data from the driver. Basically, VB requests device descriptors by calling the DLL (passes an empty pointer to buffer) and the DLL calls the driver using `IOCTL_Ezusb_GET_DEVICE_DESCRIPTOR`.

Data is passed to a buffer, and the buffer is filled and returned to VB. The driver calls the `USBD.SYS` driver to communicate with the device and OS.

HARDWARE TESTING

When you get your development board, you want to make sure the hardware works. First, install the software, which puts the EZ-USB driver into the Windows system and the install information (`INF`) file into the `INF` directory.

The `INF` file informs Windows as to what driver to load for the particular vendor ID (VID) and product ID (PID) combination. The USB Implementers Forum provides the VID; you assign the PID.

Once the software is installed, plug in the USB device with the included USB cable. It's impossible to hook up the cable backwards because the cable has two different connectors (A and B).

When the device is connected, the red light lights up, signaling that the board has power. Windows informs you that it has found new hardware, finds the appropriate `INF` file, and installs the driver for the new device.

All information concerning driver and VID/PID combinations are in the Windows system registry. If, during development, you need to delete these entries, they are located at `HKEY_LOCAL_MACHINE\Enum\USB` and `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Class\USB`.

You can use `regedit.exe` to edit and browse the registry entries on your computer. You can now unplug and replug the device as much as you need. Until you delete the registry entries, Windows remembers what driver to load and doesn't inform you of any hardware detection again. This step is called enumeration.

Enumeration is when the OS recognizes that there is new hardware on

the bus and determines its particular needs. The appropriate driver is then loaded and it gives the device a unique address. Enumeration takes place each time you plug a device on the bus and on bootup of Windows.

At this point you should try the software package that comes with the evaluation kit. EZ-USB Control Panel lets you get the descriptors from the USB device, download firmware to the device, and run the Keil debugger.

THE GOAL

Our goal is to build a demo of a working USB device. The concept is that a user application sends data to a USB device and vice versa. Our USB device is the Anchor Chips development board running firmware we created.

On the host side, we have an application made using Visual Basic. The program communicates with the device via the general-purpose driver (GPD) from Anchor Chips and a DLL we created to implement a bridge between the user interface (VB) and the GPD.

The user interface is an experiment board with four DIP switches and four LEDs. The user selects a four-digit binary combination that appears on the LEDs and vice versa for the DIP switches.

In our VB program, we call functions in a DLL that communicates with the GPD. Listing 1 shows how to call the DLL that communicates with the USB device. It also shows how to parse up the device descriptor and read and write a byte from the USB device.

A DLL was written to communicate between the Visual Basic program and the GPD. The DLL gets a device handle to the device driver in question.

We want to communicate to the first instance of the device driver (i.e., the first device to use this driver). If we get a valid handle, we can communicate to it. Otherwise, the device isn't on the bus and the driver isn't loaded. We communicate to the driver via `DeviceIOControl`. This function passes data to and from the device driver and it returns success or failure.

Listing 2 shows how a DLL can be used to communicate with the GPD. The DLL does the necessary communicating with the system driver and if

Listing 1—*This Visual Basic code calls a DLL to get the device descriptor.*

```
Private Type UnsignedInt        ' Type define to overcome VB's limitation
                                ' in not having unsigned 16-bit numbers
  lobyte As Byte
  hibyte As Byte
End Type

Private Type LongData
  Number As Long
End Type

Private Type USB_DD             ' Type define for USB Device Descriptor
  Descriptor_Length As Byte
  Descriptor_Type As Byte
  Spec_Release As UnsignedInt
  Device_Class As Byte
  Device_SubClass As Byte
  Device_Protocol As Byte
  Max_Packet_Size As Byte
  Vendor_ID As UnsignedInt
  Product_ID As UnsignedInt
  Device_Release As UnsignedInt
  Manufacturer As Byte
  Product As Byte
  Serial_Number As Byte
  Number_Configurations As Byte
End Type

'DLL functions used to communicate with USB device
Private Declare Function ReadBulkByte Lib "lusher_USB.dll" (ByRef InByte As Byte,
  ByVal PipeNumber As Byte, ByVal DeviceDriver As String) As Integer
Private Declare Function WriteBulkByte Lib "lusher_USB.dll" (ByVal OutByte As
  Byte, ByVal PipeNumber As Byte, ByVal DeviceDriver As String) As IntegerPrivate
Declare Function GetDeviceDescriptor Lib "lusher_USB.dll" (ByRef DevDes As USB_DD,
   ByVal DeviceDriver As String) As Integer
' get device descriptor and parse it into appropriate fields
Private Sub Get_USB_Device_Descriptor()
   ' Gets device descriptor from the USB device as well as verify
   '   that USB is communicating correctly and that correct
   '   source code is running
  Dim CheckData As Byte
  Dim ProdID As LongData
  Dim VendID As LongData
  Dim SpecRel As LongData
  Dim DevRel As LongData
  Dim USB_Device_Descriptor As USB_DD
  Dim Result As Integer

  Result = GetDeviceDescriptor(USB_Device_Descriptor, "\\.\ezusb-0")

  ' If all transactions met with success, then set up screen and
  '   allow user interaction
  ' Else alert user to the fact that no USB device is present and
  '   do not allow user interaction
  If Result = 0 Then
     Call USBError        ' Informs user of error and set form attributes
                          '   to offline mode
     Exit Sub
  End If

  Status.Caption = "USB Device Connected"
  Status.ForeColor = &HFF00&
  LSet ProdID = USB_Device_Descriptor.Product_ID
  LSet VendID = USB_Device_Descriptor.Vendor_ID
  LSet SpecRel = USB_Device_Descriptor.Spec_Release
  LSet DevRel = USB_Device_Descriptor.Device_Release
  DesForm.Type.Caption = USB_Device_Descriptor.Descriptor_Type
  DesForm.Spec.Caption = SpecRel.Number
  DesForm.Class.Caption = USB_Device_Descriptor.Device_Class
  DesForm.SubClass.Caption = USB_Device_Descriptor.Device_SubClass
  DesForm.Protocol.Caption = USB_Device_Descriptor.Device_Protocol
  DesForm.PacketSize.Caption =
    USB_Device_Descriptor.Max_PAacket_Size
  DesForm.VendorID.Caption = VendID.Number
  DesForm.ProductID.Caption = ProdID.Number
  DesForm.DevRel.Caption = DevRel.Number
  ProductID.Caption = ProdID.Number
  VendorID.Caption = VendID.Number
End Sub

' Example calls to read and write functions
Result = WriteBulkByte(Data_Out, 0, "\\.\ezusb-0")
Result = ReadBulkByte(Data_In, 7, "\\.\ezusb-0")
```

Listing 2—*This example DLL handles calling the EZ-USB driver.*

```
// ReadBulkData(BYTE *OutBuffer, BYTE NumberOfBytes, BYTE PipeNumber,
//   LPSTR DeviceDriver)
// Function reads data from specific pipe over USB port for device in question
int _stdcall ReadBulkData(BYTE *OutBuffer, BYTE NumberOfBytes, BYTE PipeNumber,
  LPSTR DeviceDriver)
{
   HANDLE hUSB_DeviceHandle; // Declare variables
   DWORD  nBytes = 0;
   BOOL bResult;
   BULK_TRANSFER_CONTROL bulkControl;
   BYTE Input[64];
   BYTE index;

   bulkControl.pipeNum = (ULONG)PipeNumber;
   if (NumberOfBytes > 64 || NumberOfBytes < 1)
   // Limit ammount of transfer to 64 bytes maximum
   // If greater than 64 or less than 1 then return an error
   {
      return 0;
   }
   // Get handle to USB device in question
   hUSB_DeviceHandle = CreateFile(DeviceDriver, GENERIC_WRITE,
    FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, NULL);
   if(hUSB_DeviceHandle == INVALID_HANDLE_VALUE)
   {
   return 0;        // If not a good handle then abort!
}

// Else it is a good handle; read data to USB pipe by calling system driver
  bResult = DeviceIoControl(hUSB_DeviceHandle,IOCTL_EZUSB_BULK_READ, &bulkControl,
    sizeof(BULK_TRANSFER_CONTROL), &Input[0], NumberOfBytes, &nBytes, NULL);
  CloseHandle(hUSB_DeviceHandle);  // Close handle
  // Fill result with that of input array
  for (index = 0; index < NumberOfBytes; index++)
  {
    *OutBuffer = Input[index];
    OutBuffer++;
  }
  return (int)bResult;   // Return our result: success or failure
}

// WriteBulkData(BYTE *InBuffer, BYTE PipeNumber, LPSTR DeviceDriver)
// Function writes data from specific pipe over USB port for device in question

int _stdcall WriteBulkData(BYTE *InBuffer, BYTE NumberOfBytes, BYTE PipeNumber,
  LPSTR DeviceDriver)
{
   HANDLE hUSB_DeviceHandle;    // Declare variables
   DWORD  nBytes = 0;
   BOOL bResult;
   BULK_TRANSFER_CONTROL bulkControl;
   BYTE Output[64];
   BYTE index;

   bulkControl.pipeNum = (ULONG)PipeNumber;
   // Limit ammount of transfer to 64 bytes maximum
   // If greater than 64 or less than 1, return an error
   if (NumberOfBytes > 64 || NumberOfBytes < 1)
   {
     return 0;
   }
   // Fill output array with that of input buffer
   for (index = 0; index < NumberOfBytes; index++)
   {
     Output[index] = *InBuffer;
     InBuffer++;
   }
   // Get handle to USB device in question
   hUSB_DeviceHandle = CreateFile(DeviceDriver, GENERIC_WRITE,
      FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, NULL);
   if(hUSB_DeviceHandle == INVALID_HANDLE_VALUE)
   {
     return 0;        // If not a good handle, abort!
   }
   // Else it is a good handle; write data from USB pipe by calling system driver
  bResult = DeviceIoControl(hUSB_DeviceHandle,IOCTL_EZUB_BULK_WRITE,&bulkControl,
    sizeof(BULK_TRANSFER_CONTROL), &Output[0], NumberOfBytes, &nBytes, NULL);
  CloseHandle(hUSB_DeviceHandle);  // Close handle
  return (int)bResult;   // Return our result: success or failure
}
```

there is an error, responds to the calling application with an error status. This arrangement provides an easy-to-use interface to the GPD.

FIRMWARE

For your USB microcontroller, we recommend you have the full version of the C compiler because the example files may exceed the evaluation limit of most evaluation-level compilers. Most of the code needed to communicate with the host is already written. Just fill in your peripheral and I/O code.

The development kit has two firmware files called `PERIPH.C` and `FW.C`. These files (supplied by Anchor Chips) contain the framework for the whole 8051-based USB control code. The `PERIPH.C` source file contains the polling loop code segment, as well as the endpoint interrupts for communicating with the host. You merely write your peripheral code in the poll loop.

When data is to be exported, a set of ISRs is called (seven in and seven out). These are the endpoints of the communication pipes. In the initialization section of the code, you need to set the direction of the port pins. For our example, port A is used. The upper nibble is input and the lower nibble is output. Listing 3 shows example routines from `PERIPH.C`.

In USB the host initiates all communications. If the device has something to tell the host, it must place the data into an output array (IN1BUF[0]).

After the firmware is finished, it must download to the chip because Anchor Chips' USB paradigm calls for the device-side application code to be transferred on startup of the processor. There are two methods for downloading the firmware—B0 load and B2 load. We describe B0 here.

The micro is basically a state machine that does simple USB tasks without 8051 code. Using the B0 protocol, the firmware is sent over the USB to the chip and an external EEPROM contains the device descriptor (VID and PID). This information tells Windows to load a driver.

The driver was made using the `ezloader.sys` driver source file, which lets you implement your firmware as part of the driver. The device

is enumerated to download the firmware to the micro's RAM.

The micro reenumerates and reports a new device descriptor. We used the same VID but different PIDs (x8000 for download, x8001 for device). The new VID/PID combination tells Windows to load the real driver (EZUSB.SYS).

An INF file tells Windows which VID/PID combination goes with each driver. Listing 4 is a typical INF file entry for the VID/PID combo. Our VID is 0x06E5, and the PIDs are 0x8000 and 0x8001. The sample INF file tells Windows which drivers to load according to the VID and PID information that the system retrieves from the device.

READY TO GO

Basically, you treat most of the firmware code as if you were in regular 8051 development, except that the code resides in the POLL loop, not MAIN. There are ample instructions in the kit manuals. The GPD is well documented, and their program handles the rest.

Mike Zerkus has 15 years of experience working on devices and inventions ranging from space devices to consumer products. Mike is the president of CM Research, a development company that specializes in bringing products from concept through prototype to production. You may reach him at mzerkus@cmresearch.com.

John Lusher is an electrical engineer and has been involved with USB development for the last two years. You may reach him at jlusher@lushertech.com.

Jonathan Ward is president of Keil Software and has been involved in the design, implementation, and documentation of embedded systems since the early 1980s. You may reach him via (972) 735-8052.

RESOURCES

A glossary of USB terms, a checklist for building a USB device, and a list of USB suppliers are available online in Design Forum in May.

Listing 3—*Here are a couple of sample routines from PERIPH.C.*

```
void TD_Poll(void)            // Called repeatedly while device is idle
{
  OUTA = byte_in;             // Place byte retrieved from endpoint on port A
  byte_out = 0xF0 & PINSA;    // Read port A and mask to get only upper 4 bits
}
void ISR_Ep1out(void) interrupt 0
{
  if (OUT1BUF[0] == 0x80)     // If READ COMMAND (0x80) then send data to host
  {
    IN1BUF[0] = byte_out;
    IN1BC = 1;  // Inform processor it has a byte to send out to host
  }
  else          // Else set variable to the input byte
  {
    byte_in = OUT1BUF[0];
  }
  OUT1BC = 0;                 // Arm OUT so it can receive next packet
  EZUSB_IRQ_CLEAR();          // Clear the IRQ
  OUT07IRQ = bmEP1;
}
```

Listing 4—*This code shows you an example INF file.*

```
;FILE:   EXAMPLE.INF

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

[Manufacturer]
%Example%=Example

[PreCopySection]
HKR,,NoSetupUI,,1

[DestinationDirs]
DefaultDestDir=11

[LusherTech]
;
%USB\VID_06E5&PID_8000.DeviceDesc%=FIRMWARE, USB\VID_06E5&PID_8000
%USB\VID_06E5&PID_8001.DeviceDesc%=USBDEV01, USB\VID_06E5&PID_8001

[ControlFlags]
ExcludeFromSelect=*         //removes all devices from device installer list

[FIRMWARE]
AddReg=FIRMWARE.AddReg

[FIRMWARE.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,firmdown.sys

[USBDEV01]
AddReg=USBDEV01.AddReg

[USBDEV01.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,ezusb.sys

[Strings]
Example="Example USB"
USB\VID_06E5&PID_8000.DeviceDesc="USB Firmware Download"
USB\VID_06E5&PID_8001.DeviceDesc="USB Actual Device"
```