Driver for the Cypress Starter Kit

One of the most helpful aids when first starting out in something new is examples. Cypress forgot this when they introduced their USB Starter Kit. But then for $99 U.S., you can't expect the world. The USB Thermometer Driver and Application was developed under contract by System Solutions 72410.77@compuserve.com. The code is not freely available.

What I have done is modified the WDM ISO USB driver distributed with the Windows Device Driver Kit for use with the Cypress Digital Themometer. I've kept it basically the same, so you can actually run the Digital Temperature Application on this driver to see that in-fact it does work! I will cop some flap for this, as the driver isn't really a good example to start with, especially if you know little about WDM Programming, as you will learn some bad habits. I'll point these out.

As the driver is based on copyright material developed by Microsoft, I will not distribute the entire driver. What I will attempt to do, is show you the modifications necessary to change the driver so that it works with the Cypress USB Starter Kit. This hopefully will also give you a better understanding as we work through the example.

The driver is simply the ISO_USB driver featured in the Microsoft DDK with modifications in IsoUsb_CreateDeviceObject so you can talk to the Kernel Mode Device Driver using "\.\\Thermometer_0". This allows the Cypress application to talk to the driver, rather than using the GUID which Microsoft uses in their example.

The IOCTL handler has been totally modified to handle calls from the thermometer application which comes with the Cypress Kit. This allows you to run the Cypress Thermometer Application on this device driver. It includes all the IOCTL Control Code 4 Functions such as Set LED Brightness, Read Thermometer, Read Port, Write Port, Read RAM, Write RAM & Read ROM as per the Cypress Starter Kit User Guide(Ver 0.993) Page 48.

Modifiying the IOCTL Call Handler.

O.K., forget the modifications. We will start the handler for the IRP_MJ_DEVICE_CONTROL from scratch. Delete or rename IOCTLISO.C.

Below is the table of functions we will have to implement. This is simular to the table provided by Cypress for compatibility.

Command	Command Value				Out Value			
	MSB			LSB	MSB			LSB
Set LED Brightness	-	-	Brightness	0x0E	-	-	-	Status
Read Thermometer	-	-	-	0x0B	Button	Sign	Temp	Status
Read Port	-	-	Port	0x14	-	-	Value	Status
Write Port	-	Value	Port	0x15	-	-	-	Status
Read RAM	-	-	Address	0x16	-	-	Value	Status
Read RAM	-	Value	Address	0x17	-	-	-	Status
Read ROM	-	Index	NA	0x18	-	-	Value	Status

At first glance, you would expect the Cypress USB MCU to send the temperature using an Interrupt transfer periodically. After all, the Cypress USB MCU returns an Endpoint Descriptor for EP1 as type equals Interrupt, Maximum Packet Size of 8 Bytes and an Interrupt Interval of 10mS. The String Descriptor for this Endpoint returns "Endpoint1 10ms Interrupt Pipe".

At this stage you jump straight to the vector table. The 128us timer is not used, the 1024us timer is, and the endpoint1 interrupt is not used. The Interrupt Service Routine for the 1024us interrupt, simply checks for activity, sets the suspend accordantly, helps with the button de-bounce. Maybe the 10mS Temperature Interrupt is done in the main loop?

Jumping to the code for the main loop shows we wait for enumeration, Set the Enumeration LED, Read Temperature, Update Brightness, and Set new Brightness. Maybe it's in the Read Temperature Routine? The Read Temperature Routine firstly initialises the results, reads the temperature from the DS1620 and stores it in the FIFO for Endpoint 1.

So where is the code for the Interrupt Transfer? Good question, you tell me? (Have I overlooked something?)

Now what if we were to implement a couple of functions. Maybe ReadRAM, WriteRAM? We could then check the status of the switch by reading a value in RAM. We could read the temperature, provided the temperature was stored in a RAM Location. Umm, life would be easy. We could change the LED Brightness if we write the brightness to a location and set the Update Brightness Semaphore!

This is what I believe has been done. Please correct me if I'm mistaken!

IOCTL Codes

The documentation for the Cypress kit would suggest there is only one valid IO Control Code, IOCTL 4. All driver functions are called within this IOCTL Code. This is certainly not recommended practice. Microsoft has a macro called CTL_CODE which is defined in DEVIOCTL.H which is included in the Device Driver Kit.

```
// Macro definition for defining IOCTL and FSCTL function control codes.  Note
// that function codes 0-2047 are reserved for Microsoft Corporation, and
// 2048-4095 are reserved for customers.
//

#define CTL_CODE( DeviceType, Function, Method, Access ) (                 \
    ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method) \
)
```

The two least significant bits (Method) map the buffer transfer type which determines how the buffer is passed in IOCTL calls. We need to use Method Buffered, thus both bits must be zero.

```
#define METHOD_BUFFERED                 0
#define METHOD_IN_DIRECT                1
#define METHOD_OUT_DIRECT               2
#define METHOD_NEITHER                  3
```

The function defines which function we wish to call. The Kits has used Function 1, thus generating an IOCTL code of 4. The Access Type is FILE_ANY_ACCESS (0x00), and the Device Type is Zero. This is not defined in DEVIOCTL.H and Microsoft has reserved Device Types of 0-32767. We should probably be using something like FILE_DEVICE_UNKNOWN 0x00000022. Likewise Function Codes codes 0-2047 are reserved for Microsoft Corporation, and 2048-4095 are reserved for customers.

On top of the IOCTL4, I have added extra IOCTL Calls to all the descriptors, Get the Status, and Send custom URB_FUNCTION_VENDOR_ENDPOINT Commands, should you wish to later add more Vendor Commands to the firmware.

Supporting Functions

Before we start with processing the User Mode Functions, we should write some supporting functions, which will make life easier later to come. Essential in this example is a Vendor Command Function. Others which will benefit us are Get Descriptor, which will return all the descriptors and get status which is seldom used in this example.

Vendor Command

The Vendor Command accepts the following inputs,

```
UCHAR               Request
USHORT              Value
USHORT              Index
```

The request, is used to specify the Vendor Request Function.

```
ULONG
VendorCommand(
    IN PDEVICE_OBJECT   DeviceObject,
     IN UCHAR           Request,
    IN USHORT           Value,
    IN USHORT           Index,
    PVOID               ioBuffer
     )
{

    NTSTATUS ntStatus;
    PURB urb;

    urb = ExAllocatePool(NonPagedPool, sizeof(struct
                                            _URB_CONTROL_VENDOR_OR_CLASS_REQUEST));

     if (urb)
     {
```

```
        UsbBuildVendorRequest(urb,
                              URB_FUNCTION_VENDOR_ENDPOINT,
                              sizeof(struct _URB_CONTROL_VENDOR_OR_CLASS_REQUEST),
                              (USBD_TRANSFER_DIRECTION_IN | USBD_SHORT_TRANSFER_OK),
                              0,       // Reserved Bits
                              Request,       // Request
                              Value,   // Value
                              Index,   // Index
                              ioBuffer,      // Transfer Buffer
                              NULL,    // TransferBufferMDL OPTIONAL
                              8,       // Transfer Buffer Lenght
                              NULL);   // Link OPTIONAL

        ntStatus = IsoUsb_CallUSBD(DeviceObject, urb);

        ExFreePool(urb);

        return(urb->UrbControlVendorClassRequest.TransferBufferLength);
       }
  else return(0);

}
```

The routine is quite common. We first allocate a URB from NonPaged Memory. Should this complete successfully, we build ourselves a Vendor Request and pass it to the underlying USB Driver using IsoUsb_CallUSBD which is a function provided in the ISO_USB source. Once complete we free up any resources, and return the Buffer Length.

More information on this routine can be found by looking up UsbBuildVendorRequest() in the DDK.

Get Descriptor

The Get Descriptor Routine is very simular to the above Vendor Command. However if we specify the transfer buffer length too long, the routine will wait until the desired number of bytes have been returned. If we make it too short, we don't read the full descriptor.

How we tackle this, is to make a call to get descriptor with a length of 1 bytes, which returns the first byte in the descriptor, the length. (See USB Specifications.) Then armed with this length, we can make another call, this time requesting the full descriptor lenght.

If we call UsbBuildGetDescriptorRequest with a Descriptor type of USB_CONFIGURATION_DESCRIPTOR_TYPE (0x02), then all interface, endpoint, class-specific, and vendor-specific descriptors for the configuration are retrieved. When we read the first byte, we are in-fact reading only the first byte of the Configuration Descriptor which will give us a length of 0x09 and not the length of all interface, endpoint, class-specific, and vendor-specific descriptors.

The DDK documentation only includes the first three descriptor types. However five types are defined in USB100.H. I don't believe the last two work.

Descriptor Types

USB_DEVICE_DESCRIPTOR_TYPE	0x01
USB_CONFIGURATION_DESCRIPTOR_TYPE	0x02
USB_STRING_DESCRIPTOR_TYPE	0x03
USB_INTERFACE_DESCRIPTOR_TYPE	0x04
USB_ENDPOINT_DESCRIPTOR_TYPE	0x05

Index – Specifies which descriptor to get.

Language ID – Specifies the language ID for String Descriptors. It's possible to have devices which have the string descriptors encoded in different languages. For calls other than String Descriptors, use a value of Zero.

```
ULONG
GetDescriptor(
       IN PDEVICE_OBJECT   DeviceObject,
        IN UCHAR           DescriptorType,
       IN UCHAR            Index,
       IN USHORT           LanguageId,
       PVOID               ioBuffer
```

```
        )
{

       NTSTATUS ntStatus;
       PURB urb;
       PUCHAR pch;
       ULONG length;
       pch = (PUCHAR) ioBuffer;

       urb = ExAllocatePool(NonPagedPool, sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST));

       if (urb)
          {
          Get Descriptor Length
          UsbBuildGetDescriptorRequest(urb,
                                (USHORT) sizeof (struct _URB_CONTROL_DESCRIPTOR_REQUEST),
                                DescriptorType,
                                Index,
                                LanguageId,
                                ioBuffer,
                                NULL,
                                0x01,    // Transmit Lenght. Read First Byte.
                                NULL);   // Link

          ntStatus = IsoUsb_CallUSBD(DeviceObject, urb);
          ExFreePool(urb);

         // Check Length of Incomming Data.
         if (urb->UrbControlDescriptorRequest.TransferBufferLength == 0) return(0);

         urb = ExAllocatePool(NonPagedPool, sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST));

         length = pch[0];

         if (DescriptorType == USB_CONFIGURATION_DESCRIPTOR_TYPE) length = 1024;

         if (urb)
          {
           UsbBuildGetDescriptorRequest(urb,
                                (USHORT) sizeof (struct
_URB_CONTROL_DESCRIPTOR_REQUEST),
                                DescriptorType,
                                Index,
                                LanguageId,
                                ioBuffer,
                                NULL,
                                length,  // Length of Descriptor String
                                NULL);       // Link

          ntStatus = IsoUsb_CallUSBD(DeviceObject, urb);
           ExFreePool(urb);

          return(urb->UrbControlDescriptorRequest.TransferBufferLength);
         }
          else return(0);
         }
        else return(0);
}
```

Brad Cook from 5M Consulting points out that the cypress firmware reports a language ID of 0x109 which is undefined. the correct code (for "US English", anyway) is 0x0409. This is reported in the String Language Description. Brad discovered this when he ran the USBCheck application and it was unable to decode the UNICODE strings returned from the therm board.

User Mode Functions

To work out which vendor commands translate to which user mode functions, we have to dig into the firmware. I've generated a flow chart for the EndPoint0 ISR handler.

Ping	0x00
Read ROM	0x01
Read RAM	0x02

Write RAM	0x03
Read Port	0x04
Write Port	0x05

Set LED Brightness

The following code segment shows how the driver handles *Set LED Brightness* Requests. It sends two Vendor Commands to Write to RAM Locations. The first request writes the new brightness to address 0x2C, which is defined in the firmware as gbLEDBrightness. Once the LED Brightness has been set, we must tell the microcontroller to update the brightness by setting a semaphore. We do this by writing 0x01 to location 0x2B (gbLEDBrightnessUpdate).

```
case 0x0E: // Set LED Brightness

            temp = pch[1];          // Store Brightness

            length = VendorCommand(DeviceObject,
                                   0x03, // WriteRAM
                                   0x2C, // gbLEDBrightness
                                   (USHORT)pch[1],
                                   ioBuffer);

            length = VendorCommand(DeviceObject,
                                   0x03, // WriteRAM
                                   0x2B, // gbLEDBrightnessUpdate
                                   0x01, // TRUE
                                   ioBuffer);

            pch[0] = 0;        //Status
            pch[1] = temp;     //Brightness
            Irp->IoStatus.Information = 2;
            Irp->IoStatus.Status = STATUS_SUCCESS;

            ntStatus = STATUS_SUCCESS;
            break;
```

Read Thermometer

As we have discussed, the Read Thermometer command is no more complex that simply reading a RAM location in the Cypress MCU. The temperature is stored at 0x33 (gbThermTempRead), the sign is stored at 0x34 (gbThermTempRead2). This command also checks the status of the button. The MCU loops around reading the status of the button, doing any debouncing and saving the results at 0x7A (gbButtonPushed).

```
case 0x0B: // Read Thermometer

            length = VendorCommand(DeviceObject,
                                   0x02, // ReadRAM
                                   0x33, // gbThermTempRead
                                   0,
                                   ioBuffer);

            temp = pch[1]; // Store Temperature

            length = VendorCommand(DeviceObject,
                                   0x02, // ReadRAM
                                   0x34, // gbThermTempRead2
                                   0,
                                   ioBuffer);

            temp2 = pch[1]; // Store Sign

            length = VendorCommand(DeviceObject,
                                   0x02, // ReadRAM
                                   0x7A, // gbButtonPushed
                                   0,
                                   ioBuffer);
            pch[3] = pch[1]; //Move Button
            pch[0] = 0;      //Status
            pch[1] = temp;   //Temperature
            pch[2] = temp2;  //Sign
            Irp->IoStatus.Information = 4;
```

```
            Irp->IoStatus.Status = STATUS_SUCCESS;
            ntStatus = STATUS_SUCCESS;
            break;
```

Read/Write Port, Read/Write RAM, Read ROM

The read/Write Port/Ram and Read Rom functions are pretty well self-explanatory. Some of these must be implemented for the thermometer application to work. It must use these functions to determine if the thermometer is attached.

```
        case 0x14: // Read Port

                    length = VendorCommand(DeviceObject,
                                0x04,   // ReadPort
                                pch[1], // Address
                                0,
                                ioBuffer);
                    pch[0] = 0;      //Status
                    pch[1] = pch[2]; //Value
                    Irp->IoStatus.Information = 2;
                    Irp->IoStatus.Status = STATUS_SUCCESS;
                    ntStatus = STATUS_SUCCESS;
                    break;

        case 0x15: // Write Port

                    length = VendorCommand(DeviceObject,
                                0x05,   // WritePort
                                pch[1], // Address
                                pch[2], // Value
                                ioBuffer);
                    Irp->IoStatus.Information = 1;
                    Irp->IoStatus.Status = STATUS_SUCCESS;
                    ntStatus = STATUS_SUCCESS;
                    break;

        case 0x16: // Read RAM

                    length = VendorCommand(DeviceObject,
                                0x02,   // ReadRAM
                                pch[1], // Address
                                0,
                                ioBuffer);
                    pch[0] = 0;      //Status
                    pch[1] = pch[2]; //Value
                    Irp->IoStatus.Information = 2;
                    Irp->IoStatus.Status = STATUS_SUCCESS;
                    ntStatus = STATUS_SUCCESS;
                    break;

        case 0x17: // Write RAM

                    length = VendorCommand(DeviceObject,
                                0x03,   // WriteRAM
                                pch[1], // Address
                                pch[2], // Value
                                ioBuffer);
                    Irp->IoStatus.Information = 1;
                    Irp->IoStatus.Status = STATUS_SUCCESS;
                    ntStatus = STATUS_SUCCESS;
                    break;

        case 0x18: // Read ROM

                    length = VendorCommand(DeviceObject,
                                0x01,   // ReadROM
                                pch[1], // Address
                                0,
                                ioBuffer);
                    pch[0] = 0;      //Status
                    pch[1] = pch[2]; //Value
                    Irp->IoStatus.Information = 2;
```

```
                    Irp->IoStatus.Status = STATUS_SUCCESS;
                    ntStatus = STATUS_SUCCESS;
                    break;


case 8: //Reads a Vendor Command

       pch = (PUCHAR) ioBuffer;
       length = VendorCommand(DeviceObject, (UCHAR)pch[0], (USHORT)pch[1], (USHORT)pch[2],
ioBuffer);
        Irp->IoStatus.Information = length;
        Irp->IoStatus.Status = STATUS_SUCCESS;
        ntStatus = STATUS_SUCCESS;
       break;

     case 12: // GetDescriptor(s)

        pch = (PUCHAR) ioBuffer;
        length = GetDescriptor(DeviceObject, (UCHAR)pch[0], (UCHAR)pch[1], (USHORT)pch[2],
ioBuffer);
       Irp->IoStatus.Information = length;
        Irp->IoStatus.Status = STATUS_SUCCESS;
        ntStatus = STATUS_SUCCESS;
       break;
```