TEAM FLY

14

Bulk Transfers for Any CPU

Chapter 6 introduced FTDI Chip's FT245BM and FT245BM device controllers, which enable just about any CPU with a parallel or asynchronous serial interface to communicate with a USB host. The chips handle enumeration and other tasks with no USB-specific programming required. This chapter presents two example applications plus some tips on designing with these controllers.

Two Projects

Both example applications interface to Microchip PICMicro 16F877 microcontrollers. The first example uses an FT232BM, which has an asynchronous serial interface. The second example uses an FT245BM, which has a parallel interface. The firmware is written for microEngineering Labs' PicBasic Pro Basic compiler, but can be adapted to other languages.

As you'll see, writing device firmware and host applications for these chips requires very little knowledge of USB. An understanding of USB can help you understand the devices' abilities and limits, however.

Asynchronous Serial Interface

The FT232BM has a USB port and an asynchronous serial interface that can connect to an external CPU.

The Circuit

Figure 14-1 shows an example circuit. A DLP Design's DLP-232M module contains the FT232BM chip, an EEPROM for storing configuration data, and a USB connector. I built the circuit using microEngineering Labs, Inc.'s LAB-X2 board, which has a 40-pin DIP socket for the PICMicro 16F877 microcontroller, a power-supply regulator, and a 40-pin header that provides access to the '877's port pins. You can use just about any FT232BM circuit based on FTDI Chip's example schematic and any CPU with an asynchronous serial port. If you use the LAB-X2 board, remove the MAX232 chip from its socket (because the '877's serial-port pins connect to the '232BM instead), and switches S1 and S2 on the board won't be available if you use hardware handshaking.

To send data to the host computer, the '877's firmware writes serial data to its TX output, which connects to the DLP-232M's RXD input. This pin in turn connects to RXD on the '232BM. On receiving data at RXD, the '232BM sends the data out its USB port to the host computer.

On receiving USB data from the host, the '232BM writes the data to its TXD output, which connects to RX on the DLP-232M and to the RX input on the '877. The microcontroller's firmware reads the data received at RX.

The circuit has two optional LEDs that flash when the '232BM is sending data to the PC or receiving data from the PC.

The example circuit includes connections for hardware handshaking. With the '232BM and the '877 configured to use hardware handshaking, the

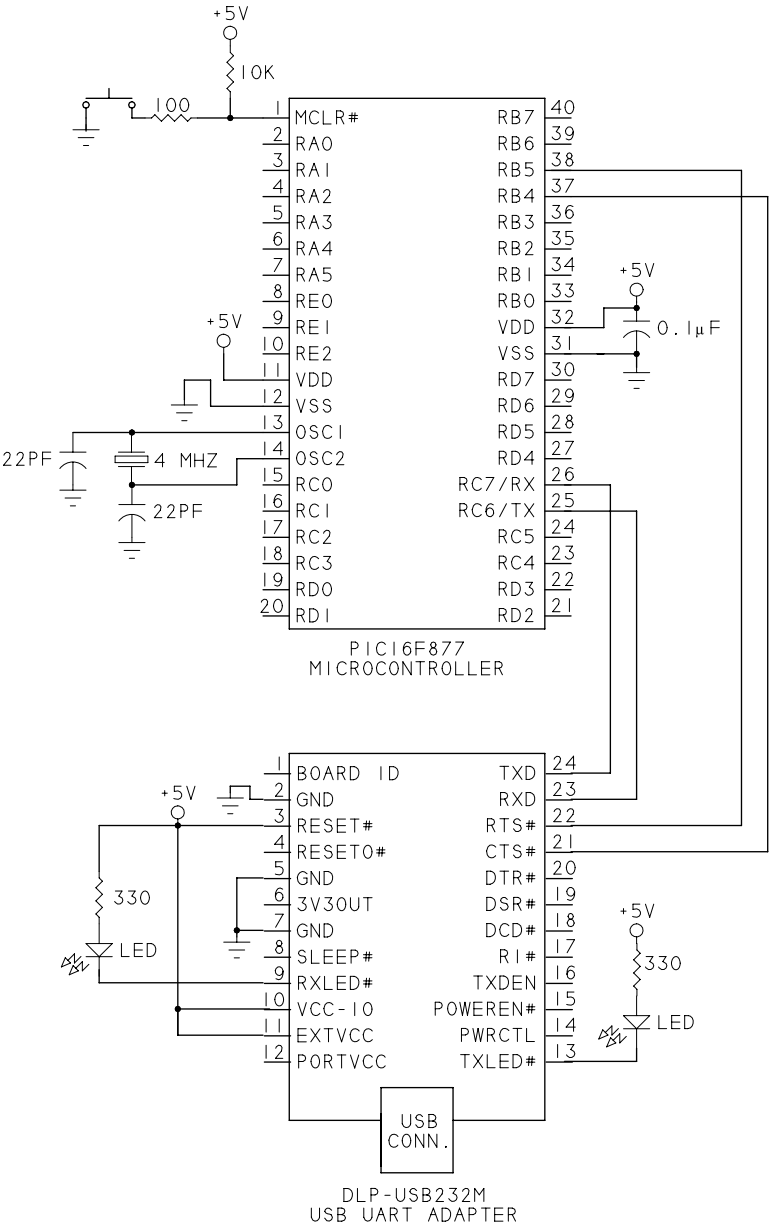

Figure 14-1: FTDI Chip's FT232BM USB UART can interface to just about any CPU with an asynchronous serial port.

’232BM transmits only when the ’877 has brought RTS# low, and the ’877 should transmit only when the ’232BM has brought CTS# low. If the corresponding handshaking signal is high, the sender should wait. Most microcontrollers (including the ’877) don’t have dedicated pins for serial-port handshaking but can use any spare port pins monitored and controlled by firmware.

As shown, the circuits use their own +5V power supply. To use bus power for the DLP-USB232M, connect VCC-IO, EXTVCC, and RESET# to PORTVCC. Circuits that use bus power must draw no more than 100 milliamperes until the host configures the USB device and must limit their current in the Suspend state, as described in Chapter 16. FTDI Chip provides information on how to ensure that a bus-powered device meets USB’s power specifications.

Program Code

Programming a CPU for asynchronous serial communications with a ’232BM requires no knowledge of USB protocols. The link between the ’232BM and the device’s CPU is an asynchronous serial link. The device’s CPU doesn’t have to know anything about the communications between the ’232BM and the USB host. The program code will vary depending on whether the device contains a hardware UART/USART and on the programming language. The ’877 has a hardware USART that transmits on the TX pin and receives on the RX pin. An interrupt can cause a routine to execute when a byte has arrived at the serial port and when a byte has transmitted.

Listing 14-1 demonstrates communications between an ’877 and ’232BM in PicBasic Pro. The ’877 reads a byte received from the host, increments the byte, and sends the byte back to the host. For handshaking, the code defines one port bit (PORTB.4) as the CTS output and one port bit (PORTB.5) as the RTS input.

The program brings CTS low to indicate that the ’877 is ready to receive a byte. CTS connects to the ’232BM’s RTS# input. On receiving a byte from the PC and determining that RTS# is low, the ’232BM writes the byte to

```
' Registers that relate to serial communications:

' Automatically clear any receive overflow errors.
DEFINE HSER_CLROERR 1

' Set the baud rate.
DEFINE HDER_BAUD 2400

' Enable the serial receiver.
DEFINE HSER_RCSTA 90h

' Enable the serial transmitter.
DEFINE HER_TXSTA 20h

' Handshaking bits. Use any spare port bits.
CTS                 VAR    PORTB.4
RTS                 VAR    PORTB.5

error               VAR    BYTE
byte_was_received   VAR    BIT
serial_in           VAR    BYTE
serial_out          VAR    BYTE

' The CTS output connects to the '232BM's RTS# input.
' The RTS input connects to the '232BM's CTS# output.
OUTPUT CTS
INPUT  RTS

' On detecting a hardware interrupt, jump to interrupt_service.
ON INTERRUPT GOTO interrupt_service

' Enable global and peripheral interrupts.
INTCON = %11000000

' Enable the serial receive peripheral interrupt.
PIE1 = %00100000

' Tell the '232BM it's OK to send a byte.
byte_was_received = 0
CTS = 0
```

Listing 14-1: PicBasic Pro code to enable a PICMicro 16F877 to communicate with an FTDI Chip FT232BM. (Sheet 1 of 3)

```
' The main program loop.
loop:

    ' Find out if a serial byte was received.
    if byte_was_received = 1 then

        ' Find out if the '232BM is ready to receive a byte.
        if RTS = 0 then

            ' Increment the received byte.
            If (serial_in = 255) then
                serial_out = 0
            else
                serial_out = (serial_in + 1)
            endif

            ' Write the incremented byte to the serial port.
            HSEROUT [serial_out]

            ' Prepare to receive another byte.
            byte_was_received = 0
            CTS = 0
        endif
    endif

GOTO loop
```

Listing 14-1: PicBasic Pro code to enable a PICMicro 16F877 to communicate with an FTDI Chip FT232BM. (Sheet 2 of 3)

```
' Disable interrupt processing in the interrupt-service routine.
DISABLE INTERRUPT
interrupt_service:

    ' This routine executes on detecting a hardware interrupt.

    ' Find out if a byte was received on the serial port.
    if ((PIR1 & %00100000) = %00100000) then

        ' Set CTS high to prevent receiving more serial data.
        CTS = 1

        ' Find out if there was a framing error.
        error = (RCSTA & %000000100)

        ' Store the byte in serial_in.
        ' This also clears any framing-error flag.
        HSERIN [serial_in]

        ' If a byte was received without error,
        ' set byte_was_received = 1 to tell the main program loop
        ' that it should send a byte to the serial port.
        ' Otherwise, set CTS= 0 to enable receiving another byte.

        if error = 0 then
            byte_was_received = 1
        else
            CTS = 0
        endif
    endif

RESUME

' Re-enable interrupt processing.
ENABLE INTERRUPT
```

Listing 14-1: PicBasic Pro code to enable a PICMicro 16F877 to communicate with an FTDI Chip FT232BM. (Sheet 3 of 3)

the ’877’s RX input. The byte’s arrival triggers an interrupt, and an interrupt-service routine reads the byte and sets a variable to inform the main program loop.

When the ’232BM’s CTS# output is low, the ’232BM is ready to receive a byte. CTS# connects to RTS on the ’877. After a byte has been received and RTS is low, the main program loop increments the received byte and writes the byte to the TX output. The ’232BM sends the received byte on to the host via the chip’s USB port. The ’877 sets CTS low to inform the ’232BM that it’s OK to send another byte.

If the ’232BM is installed using an INF file that specifies FTDI Chip’s virtual COM port (VCP) drivers, the driver causes the operating system to create a virtual COM port for communicating with the device. To access the device, you can use any application that can communicate with COM-port devices, including the HyperTerminal accessory provided with Windows.

In the PC software, set the COM port’s parameters to match what the microcontroller’s circuit uses. For example, Listing 14-1 uses a Baud rate of 2400 bits/second and the default settings of 8 data bits, 1 Stop bit, and no parity. The PC doesn’t use the COM-port parameters to communicate with the ’232BM, but the driver passes the parameters to the ’232BM in vendor-specific requests. The ’232BM uses the parameters when communicating over its asynchronous serial interface.

The Host Programming section in this chapter has more about FTDIChip’s VCP driver and the alternative D2XX Direct driver.

Parallel Interface

The FT245BM is similar to the ’232BM, but with an 8-bit parallel interface in place of the ’232BM’s asynchronous serial interface.

The Circuit

Figure 14-2 shows an example circuit. A DLP Design’s DLP-245M module contains the FT245BM chip, an EEPROM, and a USB connector. As with the previous circuit, I used microEngineering Labs’ LAB-X2 board with a

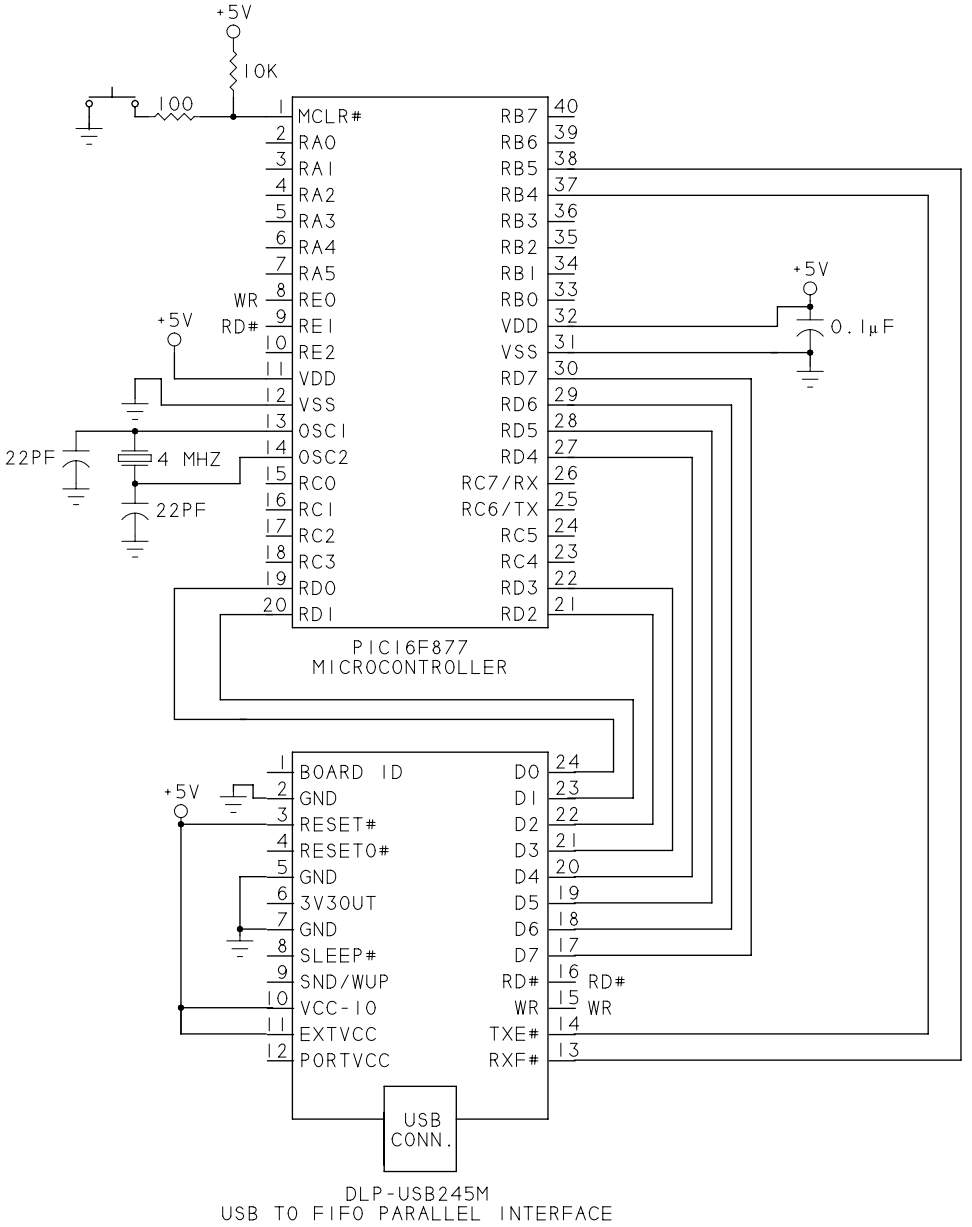

Figure 14-2: FTDI Chip's FT245BM USB FIFO has an 8-bit parallel interface.

PICMicro 16F877 microcontroller. You can use just about any FT245BM circuit based on FTDI Chip's example schematic and any CPU with a parallel I/O port and four additional spare port pins. If you use the LAB-X2 board, switches S1 and S2 on the board won't be available.

Figure 14-3 shows timing diagrams for reading and writing to the '245BM. When the PC has written a byte to the '245BM, the chip brings its RXF# output low to indicate that a byte is available. To read the byte, the external CPU brings the '245BM's RD# input low. The '245BM then places the byte on data pins D0–D7, and the external CPU can read the byte, bringing RD# high when the read operation is complete. When another byte is available for the external CPU to read, the '245BM brings RXF# low again.

To write a byte to the PC, the external CPU brings WR high and waits if necessary for the '245BM to bring its TXE# output low to indicate the chip is ready to receive a byte. The external CPU then places the byte on data pins D0–D7 and brings WR low, causing the '245BM to copy the byte into its transmit buffer and bring TXE# high. The chip sends the byte to the host over the USB port. The external CPU brings WR high to prepare for the next transfer. When ready to read another byte, the '245BM brings TXE# low again and the external CPU can write another byte to the data lines.

In Figure 14-2, the data port is Port D on the 16F877. The handshaking signals use bits on Port B and Port E. You can use any spare port pins to interface to the '245BM's data pins and status and control signals. The power connections are the same as for the '232BM.

Program Code

Listing 14-2 is PICBasic Pro code that waits to receive a byte from the host via the '245BM, increments the byte, and sends it back to the host. The firmware checks the state of RXF# before attempting to read a byte. Another option would be to use a hardware interrupt to cause the CPU to take action when RXF# goes low, indicating there is a byte available to be read. The firmware checks the state of TXE# before attempting to write a byte.

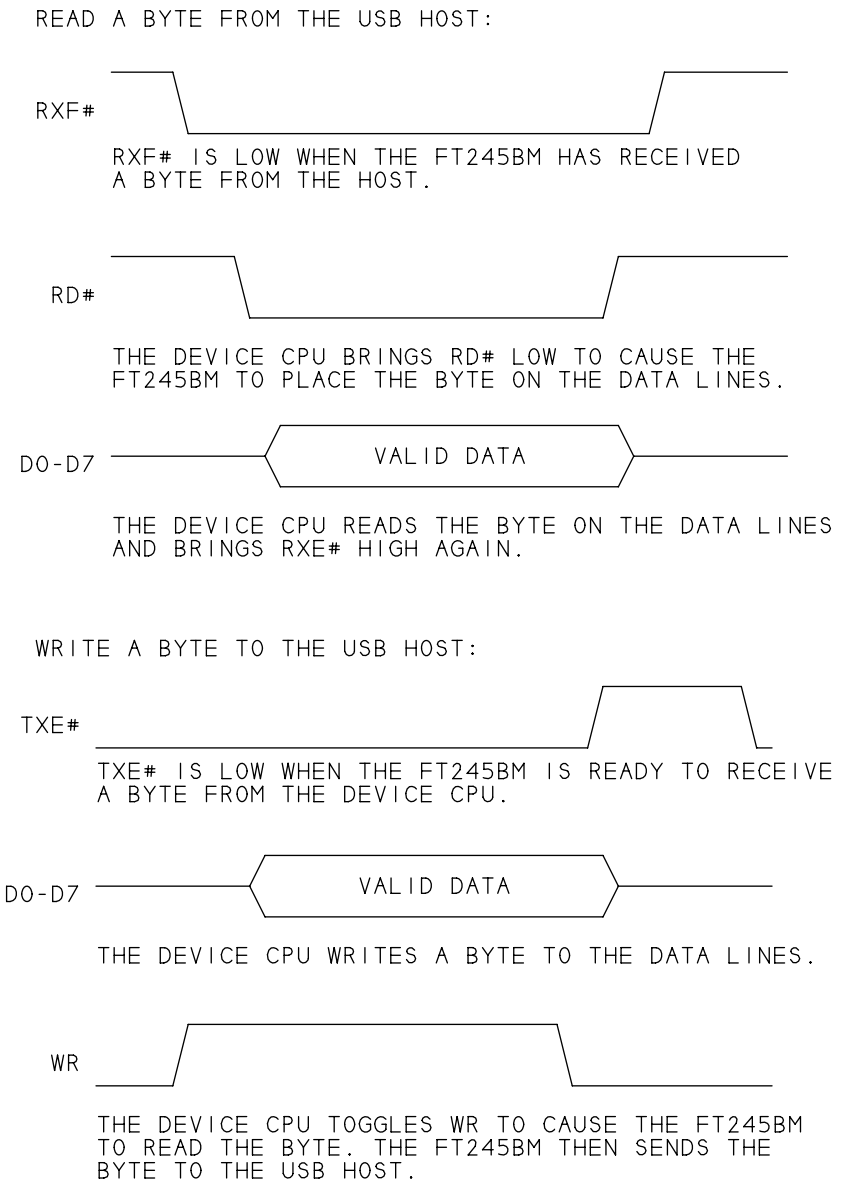

Figure 14-3: In a read operation, the device CPU reads a byte from the FT245BM. In a write operation, the device CPU writes a byte to the FT245BM.

```
' The PICMicro waits to receive a byte from the PC,
' increments the byte, and sends it back to the PC.

' An FT245BM provides the interface between a parallel port
' on the PICMicro and a USB port on the PC.

INPUTS              CON $FF
OUTPUTS             CON 0

' Parallel interface data bits.

data_port           VAR PORTD
data_port_direction VAR TRISD

' Parallel interface control outputs.
RD                  VAR PORTE.1
WR                  VAR PORTE.0

' Parallel interface status inputs.
RXF                 VAR PORTB.5
TXE                 VAR PORTB.4

data_in             VAR BYTE
data_out            VAR BYTE
read_or_write       VAR BIT

' Configure the control signals as outputs.

OUTPUT RD
OUTPUT WR

' Configure the status signals as inputs.

INPUT RXF
INPUT TXE

'Set RD and WR to their default (inactive) states.
RD = 1
WR = 0
```

Listing 14-2: PICBasic Pro code to enable a PICMicro 16F877 to read a byte from an FTDI FT245BM (Sheet 1 of 4).

```
' If read_or_write = 1,
' the PICMicro is waiting to receive a byte from the '245BM.
' If read_or_write = 0,
' the PICMicro is waiting to send a byte to the '245BM.

read_or_write = 1

'The main program loop.
loop:

    if (read_or_write = 1) then

        ' The PICMicro is waiting to receive a byte.
        ' Find out if a byte is available.

        if (RXF = 0) then

            ' RXF = 0, indicating the '245BM has a byte
            ' available to read.

            'Configure the data port's bits as inputs.
            data_port_direction = INPUTS

            ' Bring RD low to cause the '245BM to place a byte
            ' on the data port.
            RD = 0

            ' Store the byte in data_in.
            data_in = data_port

            ' Bring RD high.
            RD = 1

            ' Do something with the received byte.
            gosub process_received_data

        endif
```

Listing 14-2: PICBasic Pro code to enable a PICMicro 16F877 to read a byte from an FTDI FT245BM (Sheet 2 of 4).

```
    else

        ' The PICMicro is ready to send a byte.
        ' Find out if the '245BM is ready to receive a byte.

        if (TXE = 0) then

            ' TXE = 0, indicating that the '245BM is ready
            ' to receive a byte.

            ' Configure the data port's bits as outputs.
            data_port_direction = OUTPUTS

            ' Write the byte to the data port.
            data_port = data_out

            ' Bring WR high, then low, to cause the '245BM
            ' to read the byte.
            WR = 1
            WR = 0

            'The PICMicro is now ready to receive a byte.
            read_or_write = 1

        endif

    endif

GoTo loop
```

Listing 14-2: PICBasic Pro code to enable a PICMicro 16F877 to read a byte from an FTDI FT245BM (Sheet 3 of 4).

```
process_received_data:

    ' This example program increments the received byte
    ' and sets read_or_write = 0 to cause the PICMicro to send
    ' the byte back to the '245BM and thus on to the host.

    If (data_in = 255) then
        data_out = 0
    else
        data_out = (data_in + 1)
    endif

    ' The PICMicro is now ready to send a byte.
    read_or_write = 0

return
```

Listing 14-2: PICBasic Pro code to enable a PICMicro 16F877 to read a byte from an FTDI FT245BM (Sheet 4 of 4).

As with the previous circuit, PC applications can communicate with the ’245BM circuit using FTDIChip’s VCP driver or D2XX Direct driver.

Host Programming

When communicating with FTDI Chip’s controllers, the VCP driver is a natural choice if you’re upgrading an existing device that uses RS-232 or another asynchronous serial interface. FTDI Chip also provides the D2XX Direct driver, which enables applications to use vendor-specific functions to communicate with the ’232BM and ’245BM.

Reasons for using the D2XX driver include faster transfers, more control over communications with the external CPU, and the option to use Bit Bang mode.

Using the D2XX Direct Driver

Applications can communicate with the D2XX driver using either FTDI Chip's original Classic functions or an alternate set of functions that emulate Windows API functions.

Table 14-1 lists the Classic interface functions. The basic functions for exchanging data are FT_Open, FT_Read, FT_Write, and FT_Close. Additional functions enable configuring the controller and accessing the EEPROM. A few functions apply only to the '232BM's handshaking signals and serial-communication parameters.

An alternative to the Classic functions is the FT-Win32 API function set (Table 14-2). These functions emulate standard Win32 API functions such as CreateFile, ReadFile, and WriteFile. The functions are convenient if you have existing code that uses Win32 functions but want to use the D2XX driver. The FT-Win32 functions don't support setting the latency timer, EEPROM access, or Bit Bang mode.

An application should use either the Classic interface or the FT-Win32 interface, not both.

Selecting a Driver

Chapter 9 explained how Windows selects a driver for communicating with a device. The FTDI Chip controllers are a special case because they have two driver options and because the controllers can function using the default Vendor ID and Product ID. The defaults are your only option if your controller doesn't interface to an EEPROM. To avoid confusion about which driver the host should assign to the device, it's best to use an EEPROM with a unique Vendor ID/Product ID pair.

Although the '232BM and '245BM have different interfaces to their device CPUs, the two chips appear identical to the host computer. The host computer doesn't care what the device controller does with the data beyond the USB port. Devices that use both chips can use the same drivers and INF files. The '245BM can use the VCP driver even though it doesn't have the asynchronous serial interface found on conventional COM ports, and the

Table 14-1: Applications can use these Classic interface functions to communicate with devices that use FTDI Chip's D2XX direct driver. (Sheet 1 of 2)

Function	Purpose
FT_EE_Read	Read the EEPROM data in the FT_PROGRAM_DATA structure.
FT_EE_Program	Program the EEPROM with data in an FT_PROGRAM_DATA structure.
FT_EE_UARead	Read the EEPROM's user area.
FT_EE_UASize	Get the size of the EEPROM's user area.
FT_EE_UAWrite	Write to the EEPROM's user area.
FT_GetBitMode	In Bit Bang mode, read the bits.
FT_GetLatencyTimer	Get the value of the latency timer.
FT_GetModemStatus*	Get the states of modem status signals.
FT_GetQueueStatus	Get the number of characters in the receive buffer.
FT_GetStatus	Get the number of characters in the transmit and receive buffers and the event status.
FT_ListDevices	Get the number of connected devices and device strings.
FT_Open	Get a handle to access a device. Specifies the device by number.
FT_OpenEx	Get a handle to access a device. Specifies the device by serial number or description.
FT_Purge	Clear the transmit and receive buffers.
FT_Read	Read the specified number of bytes from the device.
FT_ResetDevice	Reset the device.
FT_SetBaudRate*	Set the baud rate, including non-standard rates.
FT_SetBitMode	Enable/disable Bit Bang mode and configure Bit Bang pins as inputs or outputs.
FT_SetBreakOff	Reset the Break condition.
FT_SetChars	Specify an event character and an error character.
FT_SetDataCharacteristics*	Set the number of bits per character, number of Stop bits, and parity.
FT_SetDTR*	Set DTR.
FT_SetEventNotification	Set conditions (character received or change in modem status) for an event notification.
FT_SetFlowControl*	Select a flow-control method.
FT_SetLatencyTimer	Set the latency timer (default = 16 milliseconds).

Table 14-1: Applications can use these Classic interface functions to communicate with devices that use FTDI Chip's D2XX direct driver. (Sheet 2 of 2)

Function	Purpose
FT_SetRTS*	Set RTS.
FT_SetTimeouts	Set timeouts for reading and writing to the device.
FT_SetUSBParameters	Set the USB maximum transfer size (default = 4096).
FT_Write	Write the specified bytes to the device.
*Not supported or needed by FT245BM (except SetBaudRate in Bit Bang mode).	

'232BM can use the D2XX driver even though it has an asynchronous serial interface that you might expect would use a COM-port driver.

Using Unique IDs

The most reliable way to be sure Windows assigns the driver you want is to assign your own Vendor ID and Product ID or use FTDI Chip's Vendor ID with a unique Product ID that you request from FTDI Chip. Store the IDs in an EEPROM that interfaces to the controller and place the IDs in an INF file for the driver you want to use. Then Windows will always know what driver to assign. (When assigning the driver, Windows will copy the INF file and rename it *oemxx.inf*, where *xx* is a unique number, so it won't matter if other vendors have also edited the INF file provided by FTDI Chip.)

Cautions When Using the Default IDs

If the device uses the default Vendor ID and Product ID, Windows may insist on selecting the driver for you, and you may not get the result you want, especially if your INF file isn't signed. (See Chapter 9 for more about signed drivers and INF files.) To avoid this problem when using the default IDs, during device installation, delete, move, or rename any signed INF files that specify the unwanted driver for the default IDs and disconnect from the Internet to prevent Windows from downloading a driver. You don't want to require end users to go to this much trouble when installing your device, however.

Table 14-2: As an alternative to the Classic functions, applications can use these FT-WIn32 functions, which emulate Windows API functions, to communicate with controllers that use FTDI Chip's D2XX direct driver.

Function	Purpose
FT_W32_ClearCommBreak	Clear the Break condition.
FT_W32_ClearCommError	Get the device status and information about a communications error.
FT_W32_CloseHandle	Close a handle obtained with FT_W32_CreateFile.
FT_W32_CreateFile	Obtain a handle to access a device. Specifies the device by serial number or description.
FT_W32_EscapeCommFunction*	Set or clear RTS, DTS, or Break.
FT_W32_GetCommModemStatus*	Get the states of modem status signals.
FT_W32_GetCommState	Get the communication parameters in a device control block.
FT_W32_GetCommTimeouts	Get the values of the read and write timeouts.
FT_W32_GetLastError	Get a status code for the last operation on the device. Success = non-zero; failure = zero.
FT_W32_GetOverlappedResult	Get the result of an overlapped operation.
FT_W32_PurgeComm	Terminate outstanding reads and/or writes and/or clear the read and/or write buffers.
FT_W32_ReadFile	Read the specified number of bytes from the device.
FT_W32_SetCommBreak	Put communications in the Break state.
FT_W32_SetCommMask	Specify events to monitor.
FT_W32_SetCommState*	Set the communication parameters in a device control block.
FT_W32_SetCommTimeouts	Set timeout values for reads and writes to the device.
FT_W32_SetupComm	Set the size of the read and write buffers.
FT_W32_WaitCommEvent	Wait for an event.
FT_W32_WriteFile	Write the specified bytes to the device.
*Not supported or needed by FT245BM.	

If you must use the default Vendor ID and Product ID, you can give each device a serial number to help distinguish the device from other devices with the same Vendor ID and Product ID. As Chapter 9 explained, Windows creates a hardware key for each device with a serial number and uses these entries to remember which driver to use. If there are no serial numbers,

Windows uses the physical port to identify the device, and the port can change as users remove and reattach devices. You could still run into problems, however, because you can't control the serial numbers of devices from other sources. So two devices that use the default Vendor ID and Product ID could end up with the same serial number.

Avoiding COM-port Proliferation

Windows by default creates a new COM port for every device that uses the VCP driver and has a serial number. If you're testing a batch of devices, you can quickly reach the maximum of 256 COM ports. To free up some port numbers, use Windows' Device Manager to uninstall devices you no longer need. Another solution (for in-house testing environments only!) is to edit *ftdibus.inf* to cause Windows to assign the VCP driver only to devices attached to a specified physical port or ports, and to cause Windows to create a single COM port for all of these devices, even if they have different serial numbers. FTDI Chip provides an application note with details about how to edit the INF file to accomplish this.

Performance Tips

When using FTDI Chip's controllers, there are several things you can do to get the best possible performance. The tips that follow show how to help data transfer as quickly as possible and how to prevent lost data.

Speed Considerations

In considering the rate of data transfer when using FTDI Chip's controllers, you need to consider both the transfer rate between the host computer and the device controller and the transfer rate between the device controller and the device's CPU.

Because the device controllers use bulk transfers, the amount of time required to transfer a specific amount of data between the host PC and the device controller can vary depending on how busy the bus is. The asynchronous serial and parallel interfaces can also slow things down if the transmitting end has to wait for the receiving end to indicate that it's ready to receive

a byte. And of course the asynchronous serial interface can be no faster than the selected baud rate.

Using either the VCP or D2XX driver, an endpoint on a ’232BM can transfer up to 3 Megabits/sec. This works out to 300 kilobytes/sec. assuming one Stop bit and one Start bit. To achieve this rate, the controller’s asynchronous serial port must use a baud rate of 3 Megabits/sec.

A ’245BM endpoint can transfer up to 300 kilobytes/sec. using the VCP driver and 1 Megabyte/sec. using the D2XX driver. For the fastest transfers, use the D2XX driver.

Minimizing Latency

For IN transfers of less than 62 data bytes, there are several ways to cause data to transfer more quickly. By default, the controller’s bulk IN endpoint NAKs IN packets unless one of the following is true:

- The transmit buffer contains at least 62 bytes.
- At least 16 milliseconds has elapsed since the last IN packet was ACKed.
- An event character is enabled and was received by the device.
- For the ’232BM only, CTS, DSR, DCD, or RI has changed state.

If any of the above is true, the controller returns two status bytes followed by the entire contents of the transmit buffer or 62 bytes, whichever is less.

For devices that must send less than 62 bytes to the host without delay, the D2XX driver has a function that can change the latency timer from its default value of 16 milliseconds. The allowed range is from 1 to 255 milliseconds. For the shortest latency, set the timer to 1, and the device will send status bytes and any data if at least 1 millisecond has elapsed since the last bulk IN packet was ACKed.

Event characters enable the host to request a device to send data immediately. The D2XX driver has a function that can define a character as a special event character. After receiving the event character, the controller sends status bytes and up to 62 data bytes in response to the next IN packet. The received event character is embedded in the data and the device firmware is

responsible for recognizing and discarding the character if it's not part of the meaningful data.

A '232BM can also be prompted to send data by changing the state of one of its handshaking inputs. And of course any of the controllers can force the data to transmit by padding the transmit buffer so it contains 62 bytes.

Preventing Lost Data

The example programs in this chapter use handshaking to enable each end of the asynchronous serial or parallel link to indicate when it's OK to send data. Handshaking isn't needed if both ends of the link have buffers large enough to store received data until the CPU can read it. Devices like Parallax Inc.'s Basic Stamp, which can accept serial data only when executing a SerialIn statement, will almost certainly need handshaking to prevent missed data.

When a CPU writes asynchronous serial data to a '232BM, the chip stores the received data in a 384-byte transmit buffer and sends the data to the host in response to IN packets as described above. Because the interface uses bulk transfers, there's no guarantee of when the host will request the data. If the bus is busy or the host is occupied with other tasks, USB communications with the device may have to wait. If the transmit buffer is full and the CPU continues to send data to the '232BM, data will be lost. Handshaking provides a way for the '232BM to let the device's CPU know when it's OK to send data. At the host, an application can usually reserve a generous buffer to hold data until the application can use it.

In the other direction, application software on the host writes data to a buffer. The host's driver sends the data in the buffer to the '232BM in OUT bulk transfers. The '232BM can store up to 128 bytes received from the host. If the buffer is full, the '232BM returns NAKs in response to attempts to send more data. The '232BM sends the data received from the host to the device's CPU via the asynchronous serial link. The CPU and related circuits that receive the data from the '232BM may have a very small buffer or no buffer at all. If there is a chance that the '232BM will write data faster than the CPU can deal with it, handshaking can prevent lost data.

The ’232BM supports three handshaking methods. The example programs use the RTS# and CTS# pins. The DTR# and DSR# pins can be used in the same way. A circuit can also use both pairs as defined in the TIA/EIA-232 standard. A third option is Xon/Xoff software handshaking, which uses dedicated codes embedded in the data to request stopping and starting transmissions.

The ’245BM has the same buffers as the ’232BM. The chip supports handshaking via the RXF# and TXE# pins, which enable each end to indicate when it’s ready to receive data, and by the RD# and WR signals, which indicate when a read or write operation is complete.

EEPROM Programming

The D2XX Direct Driver enables application software to read and write to an EEPROM that connects to a ’232BM or ’245BM over a Microwire synchronous serial interface.

EEPROM Data

An EEPROM is required if you want to customize any of a variety of device characteristics, including the Vendor ID, Product ID, or support for remote wakeup. Listing 14-3 shows a C structure that contains the values an application can write to an EEPROM using the D2XX driver and FT_EE_Program function.

An EEPROM can also store data in a user area. Host applications can read and write to this area, but the device’s CPU can access the user area only when the USB controller is in the Reset state.

Editing the Data

FTDI Chip provides an MPROG utility that enables storing a new Vendor ID, Product ID, serial number, and other information in an EEPROM that interfaces to a ’232BM or ’245BM. A complication is that the utility requires the D2XX driver to be assigned to the controller, yet Windows may balk at assigning the D2XX driver to a device that uses the default IDs. To

```
typedef struct ft_program_data {

  DWORD Signature1;        // Header - must be 0x00000000
  DWORD Signature2;        // Header - must be 0xFFFFFFFF
  DWORD Version;           // Header - FT_PROGRAM_DATA version
                           //  0 = original,
                           //  1 = contains FT2232C extensions
  WORD VendorId;           // Vendor ID (0x0403)
  WORD ProductId;          // Product ID (0x6001)
  char *Manufacturer;      // Pointer to Manufacturer string
                           //  ("FTDI")
  char *ManufacturerId;    // Pointer to Manufacturer ID string
                           //  ("FT")
  char *Description;       // Pointer to Device descr. string
                           //  ("USB HS Serial Converter")
  char *SerialNumber;      // Pointer to Serial Number string
                           //  ("FT000001" if fixed, or NULL)
  WORD MaxPower;           // Max. required bus current (mA) (44)
  WORD PnP;                // Plug and Play:
                           //  disabled (0), enabled (1)
  WORD SelfPowered;        // power source: bus (0), self (1)
  WORD RemoteWakeup;       // remote wakeup available:
                           //  no (0), yes (1)
  //
  // Rev4 (-BM series) extensions
  //
  UCHAR Rev4;              // Chip series:
                           //  -BM series (0), other (non-zero)
  UCHAR IsoIn;             // IN endpoint:
                           //  bulk (0), isochronous (non-zero)
  UCHAR IsoOut;            // OUT endpoint:
                           //  bulk (0), isochronous (non-zero)
  UCHAR PullDownEnable;    // pull-down mode:
                           //  not enabled (0), enabled (1)
  UCHAR SerNumEnable;      // serial number:
                           // enabled (non-zero), not enabled (0)
  UCHAR USBVersionEnable; // USBVersion enabled?
                           //  yes (non-zero), no (0)
  WORD USBVersion;         // USB version (BCD) (0x0200 = USB2.0)
```

Listing 14-3: The EEPROM data structure for an FTDI Chip device. Bold text indicates default values. Adapted from FTDI Chips' D2XX Programmer's Guide. (Sheet 1 of 2)

```
  // FT2232C extensions

  UCHAR Rev5;             // FT2232C chip? yes (non-zero), no (0)
  UCHAR IsoInA;           // "A" channel IN endpoint:
                          //  bulk (0), isochronous (non-zero)
  UCHAR IsoInB;           // "B" channel IN endpoint:
                          //  bulk (0), isochronous (non-zero)
  UCHAR IsoOutA;          // "A" channel OUT endpoint:
                          //  bulk (0), isochronous (non-zero)
  UCHAR IsoOutB;          // "B" channel OUT endpoint:
                          //  bulk (0), isochronous (non-zero)
  UCHAR PullDownEnable5;  // pull-down mode:
                          //  not enabled (0), enabled (1)
  UCHAR SerNumEnable5;    // serial number:
                          // enabled (non-zero), not enabled (0)
  UCHAR USBVersionEnable5;// USBVersion enabled?
                          //  yes (non-zero), no (0)
  WORD USBVersion5;       // USB version (BCD) (0x0200 = USB2.0)
  UCHAR AIsHighCurrent;   // "A" channel is high current?
                          //  yes (non-zero), no (0)
  UCHAR BIsHighCurrent;   // "B" channel is high current?
                          //  yes (non-zero), no (0)
  UCHAR IFAIsFifo;        // "A" channel is 245 FIFO?
                          //  yes (non-zero), no (0)
  UCHAR IFAIsFifoTar;     // "A" channel is 245 FIFO CPU target?
                          //  yes (non-zero), no (0)
  UCHAR IFAIsFastSer;     // "A" channel is Fast Serial?
                          //  yes (non-zero), no (0)
  UCHAR AIsVCP;           // "A" channel uses VCP driver?
                          //  yes (non-zero), no (0)
  UCHAR IFBIsFifo;        // "B" channel is 245 FIFO?
                          //  yes (non-zero), no (0)
  UCHAR IFBIsFifoTar;     // "B" channel is 245 FIFO CPU target?
                          //  yes (non-zero), no (0)
  UCHAR IFBIsFastSer;     // "B" channel is Fast Serial?
                          //  yes (non-zero), no (0)
  UCHAR BIsVCP;           // "B" channel uses VCP driver?
                          //  yes (non-zero), no (0)
} FT_PROGRAM_DATA, *PFT_PROGRAM_DATA;
```

Listing 14-3: The EEPROM data structure for an FTDI Chip device. Bold text indicates default values. Adapted from FTDI Chips' D2XX Programmer's Guide. (Sheet 2 of 2)

enable running MPROG on a device that has the default Vendor ID and Product ID and uses the VCP driver, FTDI Chip provides an application that changes the Product ID to a special "D2XX Recovery" Product ID (6006h) and an INF file that specifies the D2XX driver for devices with that Product ID. You can then run MPROG and store your final Vendor ID and/or Product ID in the EEPROM. An alternative is to use other methods to program the EEPROMs before interfacing them to the controllers.