6

Chip Choices

When you need to select a USB controller for a project, the good news is that there are plenty of chips to choose from. The down side is that deciding what controller to use in a project can be overwhelming at first.

As with any project involving embedded systems, the decision depends on what functions the chip has to perform, cost, availability, and ease of development. Ease of development depends on the availability and quality of development tools, device-driver software for the host, and sample code, plus your experience with and preferences for device architecture and language compilers.

This chapter is a guide to selecting a USB controller, including a tutorial about what you need to consider and descriptions of a sampling of chips with a range of abilities. The chips covered include inexpensive ones with simple architectures and basic USB support as well as more full-featured, high-end chips. Chapter 20 discusses controllers for use in USB On-The-Go devices.

Components of a USB Device

Every USB device must have the intelligence to implement the USB protocol. The device must detect and respond to requests and other events at its USB port. The device must be able to provide data to be sent on the bus and retrieve and use data received on the bus. A microcontroller or application-specific integrated circuit (ASIC) typically performs these functions in the device.

Controller chips vary in how much firmware support they require for USB communications. Some controllers require little more than accessing a series of registers to provide and retrieve USB data. Others require the device firmware to handle more of the protocol, including managing the sending of descriptors to the host, setting data-toggle values, and ensuring that the appropriate handshake packets are sent.

Some controllers have a general-purpose CPU on chip. Others must interface to an external CPU that handles the non-USB tasks and communicates with the USB controller as needed. These chips are sometimes called USB interface chips to distinguish them from microcontrollers with USB capabilities. All USB controllers have a USB port along with whatever buffers, registers, and other I/O capabilities the controller requires to accomplish its tasks. A controller chip with a general-purpose CPU has either program and data memory on-chip or an interface to these in external memory.

For high-volume applications that require fast performance, another option is to design and manufacture an ASIC. Several sources offer synthesizable VHDL and Verilog Source code for use in custom ASICs.

Not all controller chips support all four transfer types, and a controller may support one or more bus speeds. Many controllers support fewer than the maximum number of endpoint addresses (1 control endpoint and 30 other endpoint addresses) because few devices need the maximum number.

The USB Controller

A typical USB controller contains a USB transceiver, a serial interface engine, buffers to hold USB data, and registers to store configuration, status, and control information relating to USB communications.

The Transceiver

The USB transceiver provides a hardware interface between the device's USB connector and the circuits that control USB communications. The transceiver is typically on-chip, but some controllers allow interfacing to an external transceiver.

The Serial Interface Engine

The circuits that interface to the transceiver form a unit called the serial interface engine (SIE). The SIE typically handles the sending and receiving of data in transactions. The SIE doesn't interpret or use the data, but just sends the data that has been made available and stores any data received. A typical SIE does all of the following:

- Detect incoming packets.
- Send packets.
- Detect and generate Start-of-Packet, End-of-Packet, Reset, and Resume signaling.
- Encode and decode data in the format required on the bus (NRZI with bit stuffing).
- Check and generate CRC values.
- Check and generate Packet IDs.
- Convert between USB's serial data and parallel data in registers or memory.

Implementing these functions requires about 2500 gates.

Buffers

USB controllers use buffers to store recently received data and data that's ready to be sent on the bus. In some chips, such as PLX Technology's

NET2272, the CPU accesses the buffers by reading and writing to registers, while others, such as Cypress Semiconductor's EZ-USB, reserve a portion of data memory for the buffers.

Buffers that hold transmitted or received data are often structured as FIFO (first in, first out) buffers. Each read of a receive FIFO returns the byte that has been in the buffer the longest. Each write to a transmit FIFO stores a byte that will transmit after all of the bytes already in the buffer have transmitted. An internal pointer to the next location to be read or written to increments automatically as the firmware reads or writes to the FIFO.

In some chips, such as Cypress' enCoRe series, the USB buffers are in ordinary data memory and the firmware explicitly selects each location to read and write to. There is no pointer that increments automatically when the firmware reads or writes to the buffers. The bytes in the USB transmit buffer go out in order from the lowest address to the highest, and the bytes in a USB receive buffer are stored in the order they arrive, from lowest address to highest. These buffers technically aren't FIFOs, but are sometimes called that anyway.

To enable faster transfers, some chips have double buffers that can store two full sets of data in each direction. While one block is transmitting, the firmware can write the next block of data into the other buffer so the data will be ready to go as soon as the first block finishes transmitting. In the receive direction, the extra buffer enables a new transaction's data to arrive before the firmware has finished processing data from the previous transaction. The hardware automatically switches, or ping-pongs, between the two buffers. Some high-speed controllers, such as Cypress' EZ-USB FX2 series, support quadruple buffers.

Configuration, Status, and Control Information

USB controller chips typically have registers that hold information about what endpoints are enabled, the number of bytes received, the number of bytes ready to transmit, Suspend-state status, error-checking information, and other information about how the interface will be used and the current status of transmitted or received data. For example, setting a bit in a config-

uration register may enable an endpoint. The number of registers, their contents, and how to access them vary with the chip family. Because these details vary with the chip or chip family, the low-level device firmware for USB communications is specific to each chip or chip family.

Clock

USB communications require a timing source, typically provided by a crystal oscillator. Low-speed devices can sometimes use a less expensive ceramic resonator. Some controllers have on-chip clock circuits and don't require an external timing source.

Other Device Components

In addition to a USB interface, the circuits in a typical USB device include a CPU, program and data memory, other I/O interfaces, and additional features such as timers and counters. These circuits may be in the controller chip or in separate components.

CPU

A USB device's CPU controls the chip's actions by executing instructions in the firmware stored in the chip. If the USB controller has a CPU on-chip, the CPU may be based on a general-purpose microcontroller such as the 8051 or PICMicro, or the CPU may be an architecture developed specifically for USB applications. An interface-only USB controller can interface to any CPU with a compatible interface.

Program Memory

The program memory holds the code that the CPU executes. The program code assists in USB communications and carries out whatever other tasks the chip is responsible for. This memory may be in the microcontroller or in a separate chip.

The program storage may use any of a number of memory types: ROM, EPROM, EEPROM, Flash memory, or RAM. All except RAM (unless it's battery-backed) are nonvolatile; the memory retains its data after powering down. The amount of program memory may range from a couple of kilo-

bytes on up. Chips that can access memory off-chip may support a Megabyte or more of program memory.

Another name for the code stored in program memory is firmware, which indicates that the memory is non-volatile and not as easily changed as program code that can be loaded into RAM, edited, and re-saved on disk. In this book, I use the term firmware to refer to a controller's program code, with the understanding that the code may be stored in a variety of memory types, some more volatile than others.

ROM (read-only memory) must be mask-programmed at the factory and can't be erased. It's practical only for product runs in the thousands.

EPROM (erasable programmable ROM) is user-programmable. Many chips have inexpensive programming hardware and software available. To erase an EPROM, you insert the chip into an EPROM eraser, which exposes the circuits beneath the chip's quartz window to ultraviolet light. Data sheets rarely specify the number of erase/reprogram cycles that the chip can withstand, but it's typically at least 100.

OTP (one-time programmable) PROMs are a cheaper, non-erasable alternative to erasable EPROMs. Internally, they're identical to EPROMs, and you program them exactly like EPROMs. The difference is that the chips lack the window for erasing. The erasable varieties are useful for product development. Then to save cost, you can switch to OTP PROMs for the final product run. Many microcontrollers have both EPROM and OTP PROM variants.

Flash memory is another electrically-erasable memory technology that is popular because it doesn't need a quartz window and often doesn't need the special programming voltage required by other EPROMs. Current Flash-memory technology enables around 100,000 erase/reprogram cycles. Because Flash memory is easily reprogrammable, it's handy for making changes during project development and for programming the final firmware in low-volume projects

EEPROM (electrically erasable PROM) also doesn't need a window, nor does it need the special programming voltage required by other EPROMs. EEPROMs tend to have longer access times than Flash memory. EEPROMs

are available both with the parallel interface used by EPROMs and Flash memory, and with a variety of synchronous serial interfaces: Microwire, I^2C, and SPI. Serial EEPROMs are useful for storing small amounts of data that changes only occasionally, such as configuration data, including the Vendor ID and Product ID. Cypress' EZ-USB controllers can store their firmware in a serial EEPROM and load the firmware into RAM on powering up. Current EEPROM technology enables around 10 million erase/reprogram cycles.

RAM (random-access memory) can be erased and rewritten endlessly, but the stored data disappears when the chip powers down. It's possible to use RAM for program storage by using battery backup or by loading the code from a PC on each power-up. Any CPU with external program memory can use battery-backed RAM for program storage. Cypress Semiconductor's EZ-USB chips can use RAM for program storage, along with special hardware and driver code that loads code into the chip on power up or attachment. Host-loadable RAM has no limit on the number of erase/rewrite cycles. For battery-backed RAM, the limit is the battery life. Access times for RAM are fast.

Data Memory

Data memory provides temporary storage during program execution. The contents of data memory may include data received from the USB port, data to be sent to the USB port, values to be used in calculations, or anything else the chip needs to remember or keep track of. Data memory is usually RAM. Typical amounts of internal data memory are 128 to 1024 bytes.

Other I/O

Every USB controller has an interface to the world outside of itself in addition to the USB port. An interface-only chip must have a local bus or other interface to the device's CPU. (An exception is FTDI Chip's controllers used in Bit Bang mode to implement basic inputs and outputs.) Most chips also have a series of general-purpose input and output (I/O) pins that can connect to other circuits. A chip may have built-in support for other serial interfaces, such as an asynchronous interface for RS-232, or synchronous

interfaces such as I^2C, Microwire, and SPI. Some chips have special-purpose interfaces. For example, the Philips UDA1325 is a stereo USB codec for audio applications and contains an I^2S (Inter-IC Sound) digital stereo playback input and output.

Other Features

A device controller chip may have additional features such as hardware timers, counters, analog-to-digital and digital-to-analog converters, and pulse-width-modulation (PWM) outputs. Just about anything that you might find in a general-purpose microcontroller is likely to be available in a USB device controller.

Simplifying Device Development

In selecting a chip for a project, an obvious consideration is finding a controller that meets the hardware requirements of the product being designed. In addition, project development will be easier and quicker if you select a controller chip with all of the following:

- A chip architecture and programming language that you're familiar with.
- Detailed and well-organized hardware documentation.
- Well-documented, bug-free example firmware for an application similar to yours.
- A development system that enables easy downloading and debugging of firmware.

In addition, your project will progress more quickly if the host system can use a class driver included with the operating system or a well-documented and bug-free driver provided by the chip vendor or another source and usable as-is or with minimal modifications.

These are not trivial considerations! The right choices will save you many hours and much aggravation.

Device Requirements

In selecting a device controller suitable for a project, these are some of the areas to consider:

How fast does the data need to transfer? A device's rate of data transfer depends on whether the device supports low, full, or high speed, the transfer type being used, and how busy the bus is. As a device designer, you don't control how busy a user's bus will be, but you can select a speed and transfer type that give the best possible performance for your application.

If a product requires no more than low-speed interrupt and control transfers, a low-speed chip may save money in circuit-board design, components, and cables. HID-class devices can use low-speed chips. But remember that low-speed devices can transfer only eight data bytes per transaction, and the USB specification limits the guaranteed bandwidth for an interrupt endpoint to 800 bytes/second, which is much less than the bus speed of 1.5 Megabits/second. Even if low speed is feasible, don't rule out full or high speed automatically. Implementing low speed's slower edge rates increases the manufacturing cost of low-speed controllers, so the controller chips themselves may not be cheaper. You may find a full-speed or even a high-speed chip that can do the job at the same or even a lower price.

Compared to low and full speed, circuit-board design for high-speed devices is more critical and can add to the cost of a product. In most cases, devices that support high speed should also support full speed to enable them to work with 1.x hosts and hubs.

How many and what type of endpoints? Each endpoint address is configured to support a transfer type and direction. A device that does only control transfers needs just the default endpoint. Interrupt, bulk, or isochronous transfers require additional endpoint addresses. Not all chips support all transfer types. Most support fewer than the maximum possible number of endpoints.

Must the firmware be easily upgradable? For program memory, some devices use windowed EPROM, OTP PROM, or other memory that isn't easily erased and re-written. To change the program, you need to insert a

new chip or remove, erase, re-program, and replace the chip. Cypress' EZ-USB has an easier way, with the ability to load firmware from the host into RAM on each power up or attachment. Another option is to store the program code in electrically reprogrammable Flash memory or EEPROM. This memory can be in the device controller or in an external chip. The *Device Firmware Upgrade* class specification describes a mechanism for loading firmware from a host to a device. Chapter 7 has more about this class.

Does the device require a flexible cable? One reason why mice are almost certain to be low-speed devices is that the less stringent requirements for low-speed cables mean that the cable can be thinner and more flexible. However, USB 2.0-compliant low-speed cables have the same requirements as full and high speed except that the braided outer shield and twisted pair are recommended, but not required.

Does the device require a long cable? A cable that attaches to a low-speed device can be no longer than three meters, while full-speed cables can be five meters.

What other hardware features and abilities are needed? Other things to consider are the amount of general-purpose or specialized I/O, the size of program and data memory, on-chip timers, and other special features that a particular device might require.

Chip Documentation

Most vendors supplement their chips' data sheets with technical manuals, application notes, example code, and other documentation. The best way to get a head start on writing firmware is to begin with example code that's similar to your application. Working from an example is much easier than trying to put something together from scratch. Chip and tool vendors vary widely in the amount and quality of documentation and example code provided, so it's worth checking the manufacturers' Web sites to find out what's available before you commit to a chip. In some cases you can find code examples from other sources, especially via the Internet, from other users who are willing to share what they've written.

Driver Choices

The other side of programming a USB device is the driver and application software at the host. Here again, examples are useful.

If your device fits into a class supported by the operating systems the device will run under, you don't have to worry about writing or finding a device driver. For example, applications can access a HID-class device using standard API functions that communicate with the HID drivers included with Windows.

Some vendors provide a generic driver that you can use to exchange data with the device. An examples is Cypress' CyUsb driver, which is a general-purpose driver suitable for communicating with any device that contains a Cypress controller and doesn't belong to a standard class. Silicon Laboratories is another manufacturer that provides a general-purpose driver for use with the company's chips. Chapter 7 and Chapter 8 have more about classes and device drivers.

Debugging Tools

Ease of debugging also makes a big difference in how easy it is to get a project up and running. Products that can help include development boards and software offered by chip vendors and other sources. A protocol analyzer is also very useful during debugging. Chapter 17 has more about protocol analyzers.

Development Boards from Chip Vendors

Chip manufacturers offer development boards and debugging software to make it easier for developers to test and debug new designs. A development board enables you to load a program from a PC into the chip's program memory, or into circuits that emulate the chip's hardware.

Silicon Laboratories' C8051F32x controllers include a dedicated 2-wire debugging interface that uses no additional memory or port bits on the chip. These chips don't require using an emulator or assigning of chip resources to debugging.

The debugging software provided with a development board is typically a monitor program that runs on a PC and enables you to control program execution and watch the results. Standard features include the ability to step through a program line by line, set breakpoints, and view the contents of the chip's registers and memory. You can run the monitor program and a test application at the same time. You can see exactly what happens inside the chip when it communicates with your application.

If you have a development system for your favorite microcontroller family, you may be able to use the system for USB developing as well.

Boards from Other Sources

If you're on a strict budget, inexpensive printed-circuit boards from a variety of vendors can serve as an alternative to the development kits offered by chip manufacturers. You can also use these boards as the base for one-of-a-kind or small-scale projects, saving the time and expense of designing and making a board for the controller chip.

I/O Boards. A typical board contains a USB controller and connector along with a variety of I/O pins that you can connect to external circuits of your own design. The EZ-USB family is a natural choice for this type of board because its firmware is downloadable from the host, so you don't have to worry about programming hardware. Several sources offer boards with EZ-USB chips.

The USB I2C/IO board from DeVaSys Embedded Systems (Figure 6-1) contains an AN2131 EZ-USB chip, a connector with 20 bits of I/O, an I^2C interface for synchronous serial communications, and an asynchronous serial interface. The on-board 24LC128 is an I^2C EEPROM that can store 16 kilobytes of data, including a Vendor ID, Product ID, and firmware. The board can load its firmware from EEPROM or from the host on attachment or power-up.

DeVaSys provides the board's schematic and a free custom device driver and firmware that enable applications to open communications and read and write to ports, including the I^2C port. If you prefer, you can load your own

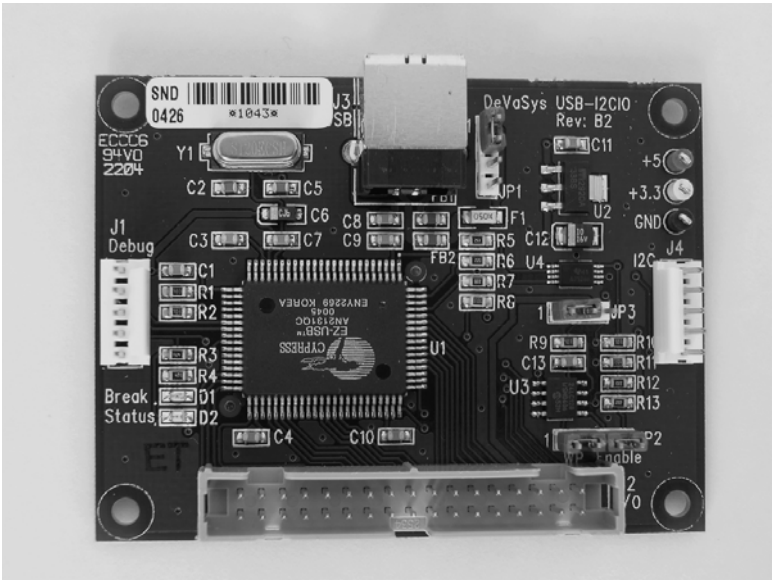

Figure 6-1: The USB I2C/IO board from DeVaSys contains an EZ-USB and a variety of options for I/O.

firmware into the device and use your own driver or a driver provided by Windows.

Other sources offer similar boards using the EZ-USB and other controllers.

Emulating a Device with a PC. Another option that can be useful in the early stages of developing is using a PC to emulate a device. You can use the compilers, debuggers, and other software tools you're familiar with on your PC and compile, run, and debug the device code on the PC.

PLX Technology's NET2272 PCI-RDK is a development kit that enables using a PC as a device when developing code using PLX Technology's NET2272 USB interface chip. The kit includes a PCI card with a header that attaches to a daughter card that contains a NET2272. You can install the PCI card in a PC and write applications that perform the role of device firmware that communicates with the interface chip. The application can run as a console application on the PC.

The USB connector on the PCI card can connect to any USB host. When development on the emulated device is complete, you can port the firmware to run on the CPU that the final design will use. If you want to use the development kit's circuits, you can remove the daughter board from the PCI card and wire the daughter board to your device's hardware.

Of course, there may be timing differences on the emulated device, and the PC won't have the same hardware architecture as the device, but the ease of developing on a PC can help in getting the code for enumerating and basic data transfers working quickly.

Controllers with Embedded CPUs

The following descriptions of USB controllers with embedded CPUs will give an idea of the range of chips available. The chips described are a sampling, and new chips are being released all the time, so any new project warrants checking the latest offerings.

If you have a favorite CPU family, the chances are good that a USB-capable variant is available. Controllers that are compatible with existing chip families have two advantages. Many developers are already familiar with the architecture and instruction set. And selecting a popular family means that programming and debugging tools are available, and example code and other advice is likely to be available from other users.

The family with the most sources for device controllers is the venerable 8051. Intel originated the 8051 family and was the first to release 8051-compatible USB controllers (the 8x930 and 8x931). Intel no longer offers USB-capable 8051s, but other manufacturers do. Controllers compatible with other families are available as well, including Atmel's AVR, Microchip's PICmicro, and Freescale Semiconductor's 68HC05 and 68HC08. Table 6-1 lists a variety of chips that are compatible with popular microcontroller families.

Some device controllers contain CPUs designed specifically for USB applications. Instead of adding USB capability to an existing architecture, the

Table 6-1: USB controller chips that are compatible with popular microcontroller families are available from many sources.

Compatibility	Manufacturer	Chips	Bus Speed
Atmel AVR	Atmel	AT43USB35x, AT76C713	Full
Freescale/Motorola 68HC05	Freescale Semiconductor	68HC05JB3/4	Low
Freescale/Motorola 68HC08	Freescale Semiconductor	68HC08JB8	Low
Freescale/Motorola PowerPC	Freescale Semiconductor	MCF5482	Full/High
Infineon C166	Infineon	C161U	Full
Intel 80C186	AMD	Am186CC	Full
Intel 8051	Atmel	AT89C513x	Full
	Cypress Semiconductor	EZ-USB, EZ-USB FX	Full
		EZ-USB FX2	Full/High
	Prolific Technology	PL-23xx	Full
		PL-25xx	Full/High
	Silicon Laboratories	C8051F32x	Full
	Standard Microsystems Corporation (SMSC)	USB97Cxxx, USB222x	Full, Full/High
	Texas Instruments	TUSB3210/3410	Full
		TUSB6250	Full/High
Microchip PIC16	Microchip Technology	PIC16C7x5	Low
Microchip PIC18	Microchip Technology	PIC18F2455/2550/ 4455/4550	Full/High
STMicroelectronics ST7, ST9	STMicroelectronics	ST7265X, ST7263, ST92163	Low, Full

designs are optimized for USB from the start. Cypress Semiconductor's enCoRe family is an example.

For common applications such as keyboards, drives, and interface converters, there are application-specific controllers that include hardware to support a particular application. The vendor often provides example firmware

and software drivers when needed as well. Chapter 7 has more about controllers for specific applications.

The chips described below each contain a CPU and a USB controller.

Microchip PIC18F4550

Microchip Technology's PICmicro microcontrollers have many fans because of the chips' low cost, wide availability, many variants, speed, and low power consumption. The PIC18F4550 is a PICmicro microcontroller with a USB controller that can function at low and full speeds. Microchip offers several other full-speed variants with different combinations of features.

Architecture

The chip is a member of Microchip's high-performance, low-cost PIC18 series. Program memory is Flash memory. The chip also has 256 bytes of EEPROM. A bootloader routine can upgrade firmware via the USB port.

The chip has 34 I/O pins that include a 10-bit analog-to-digital converter, a USART, a synchronous serial port that can be configured to use I^2C or SPI, enhanced PWM capabilities, and two analog comparators.

The USB module and CPU can use separate clock sources, enabling the CPU to use a slower, power-saving clock.

USB Controller

The USB controller supports all four transfer types and up to 30 endpoint addresses plus the default endpoint. The endpoints share 1 kilobyte of buffer memory, and transfers can use double buffering. For isochronous transfers, USB data can transfer directly to and from a streaming parallel port.

For each enabled endpoint address, the firmware must reserve memory for a buffer and a buffer descriptor. The buffer descriptor consists of four registers. The status register contains status information and the two highest bits of the endpoint's byte count. The byte count register plus the two bits in the status register contain the number of bytes to be transmitted or sent in an IN transaction or the number of bytes expected or received in an OUT

transaction. The address low register and address high register contain the starting address for the endpoint's buffer in RAM.

The microcontroller's CPU and the USB SIE share access to the buffers and buffer descriptors. A UOWN bit in the buffer descriptor's status register determines whether the CPU or SIE owns a buffer and its buffer descriptor. The SIE has ownership when data is ready to transmit or when waiting to receive data on the bus. When the SIE has ownership, the CPU should not attempt to access the buffer or buffer descriptor, except to read the UOWN bit. When readying an endpoint to perform a transfer, the last operation the firmware should perform is updating the status register to set UOWN to pass ownership to the SIE. When a transaction completes, the SIE clears the UOWN bit, passing ownership back to the CPU.

Each endpoint number also has a control register that can enable either a control endpoint, an IN endpoint, an OUT endpoint, or a pair of IN and OUT endpoints with the same endpoint number. Other bits in the register can stall the endpoint and disable handshaking (for isochronous transactions).

Additional registers store the device's address on the bus and contain status and control information for USB communications and interrupts.

Microchip provides USB Firmware Framework code and example applications for USB communications. The firmware is written for Microchip's C18 C compiler. The Framework code is structured to make it as easy as possible to develop firmware for devices in different classes and vendor-specific devices. Chapter 11 has more about using this chip.

Two other USB-capable microcontrollers from Microchip are the PIC16C745 and PIC16C765. These are less flexible because they support low speed only and their program memory is EPROM instead of Flash memory.

Cypress EZ-USB

Cypress Semiconductor's EZ-USB family includes full-speed and full/high speed controllers. The chips support a variety of options for storing firm-

ware, including loading firmware from the host on each power-up or attachment.

The EZ-USB family originated with Anchor Chips, which Cypress acquired in 1999. You may see the name *Anchor* in older documentation.

Architecture

The EZ-USB's architecture is similar to Maxim Integrated Products/Dallas Semiconductor's DS80C320, which is an 8051 whose core has been redesigned for enhanced performance. The chip uses four clock cycles per instruction cycle, compared to the original 8051's twelve. Each instruction takes between one and five instruction cycles. On average, an EZ-USB is 2.5 times as fast as an 8051 with the same clock speed.

The instruction set is compatible with the 8051's. All of the combined code and data memory is RAM. There is no non-volatile memory on-chip. However, the chips support non-volatile storage in I^2C serial EEPROM and in external parallel memory.

The EZ-USB family includes three series: the basic EZ-USB (AN21XX), the FX (CY7C646XX), and the FX2 (CY7C68013). Within each series are chips that vary in features such as the number of I/O pins or availability of an external data bus. Table 6-2 summarizes the features of each series. The FX series adds faster I/O and a general programmable interface that supports configurable, automated handshaking. The FX2 series adds support for high speed.

Keil Software has a C compiler for the EZ-USB family, or you can use assembly code. The compiler has a limited but free evaluation version. Cypress provides Frameworks firmware in C to handle much of the work of USB communications.

USB Controller

Some of the EZ-USB chips support the maximum number of endpoints and all four transfer types. Chips with fewer endpoints are also available. The EZ-USB's many options for storing firmware make its architecture

Table 6-2: Cypress Semiconductor's EZ-USB family is compatible with the 8051 microcontroller.

Feature	AN21xx (EZ-USB)	CY7C646xx (EZ-USB-FX)	CY7C68013 (EZ-USB-FX2)
Speed	Full	Full	Full/High
Number of endpoints	13, 16, 31	31	11
Compatibility	80C320, 8051	80C320, 8051	80C320, 8051
RAM (bytes)	256 + 4-8K combined data and program memory	256 + 4-8K combined data and program memory	256 + 8K combined data and program memory
Program memory type	RAM, serial EEPROM, external parallel	RAM, serial EEPROM, external parallel	RAM, serial EEPROM, external parallel
Internal program memory (bytes)	4–8K combined data and program memory	4–8K combined data and program memory	8K combined data and program memory
External memory bus (bytes)	64K	64K	one or two 64K buses
General-purpose I/O pins	16–24	16–40	16–40
Other I/O	2 UARTs, I^2C	2 UARTs, I^2C	2 UARTs, I^2C
Power Supply Voltage	3–3.6	3–3.6	3–3.6
Number of Pins	44, 48, 80	52, 80, 128	56, 100, 128

more complicated compared to other chips. The options are useful because they make the chip very flexible, so I'll describe them in some detail.

When an EZ-USB wants to use firmware stored in the host, the device enumerates twice. On boot up or device attachment, the host attempts to enumerate the device. But how can the host enumerate a device with no stored firmware? Every EZ-USB contains a core that knows how to respond to enumeration requests and can control communications when the device first attaches to the bus. The EZ-USB core is independent from the 8051 core that normally controls the chip after enumeration. The EZ-USB core communicates with the host while holding the 8051 core in the reset state.

The EZ-USB core also responds to vendor-specific requests that enable the chip to receive, store, and run firmware received from the host. For basic

testing, the core circuits also enable the device to transfer data using all four transfer types without any firmware programming.

A ReNum register bit determines whether the EZ-USB or 8051 core responds to requests at Endpoint 0. On power-up, ReNum is zero and the EZ-USB core controls Endpoint 0. When ReNum is set to one, the 8051 core controls Endpoint 0.

The source of an EZ-USB's firmware depends on two things: the contents of the initial bytes in an external EEPROM and the state of the chip's EA input. On power-up and before enumeration, the EZ-USB core attempts to read bytes from a serial EEPROM on the chip's I^2C interface. The result, along with the state of the chip's EA input, tell the core what to do next: use the default mode, load firmware from the host, load firmware from EEPROM, or boot from code memory on the external parallel data bus (Table 6-3). Chips in all three EZ-USB series can use the methods described below. The values in the first EEPROM locations vary depending on whether the chip is an EZ-USB, EZ-USB-FX or EZ-USB-FX2. The description below uses the values for the basic EZ-USB. Table 6-3 has the values for the other series.

Default Mode. The default mode is the most basic mode of operation and doesn't use the serial EEPROM or other external memory. The EZ-USB core uses this mode if EA is logic low and either the core detects no EEPROM or the first byte read from EEPROM is not B0h or B2h.

When the host enumerates the device, the EZ-USB core responds to requests. During this time, the 8051 core is in the reset state. This reset state is controlled by a register bit in the chip. The host can request to write to this bit to place the chip in and out of reset. This reset affects the 8051 core only and is unrelated to USB's Reset signaling.

The descriptors retrieved by the host identify the device as a Default USB Device. The host matches the retrieved Vendor ID and Product ID with values in a Cypress-provided INF file that instructs the host to load one of Cypress' general purpose drivers (either the CyUsb driver or the older General Purpose Driver) to communicate with the chip. The ReNum bit remains at zero.

Table 6-3: An EZ-USB can run firmware from four sources.

Firmware Source	State of EA pin	First Byte in Serial EEPROM
Load from host on re-enumerating	Don't care	EZ-USB: B0h EZ-USB-FX: B4h EZ-USB-FX2: C0h
Load from serial EEPROM	Don't care	EZ-USB: B2h EZ-USB-FX: B6h EZ-USB-FX2: C2h
Default USB Device	L	No EEPROM present or EZ-USB: not B0h or B2h, EZ-USB-FX: not B4h or B6h, EZ-USB-FX2: not C0h or C2h,
External parallel memory	H	No EEPROM present or EZ-USB: not B0h or B2h, EZ-USB-FX: not B4h or B6h, EZ-USB-FX2: not C0h or C2h

This default mode is intended for use in debugging. You can use this mode to get the USB interface up and transferring data. In addition to supporting transfers over Endpoint 0, the Default USB Device can use the other three transfer types on other endpoints. All of this is possible without having to write any firmware or device drivers.

Load Firmware from the Host. The core can also read identifying bytes from the EEPROM on power up and provide this information to the host during enumeration. If the first value read from the EEPROM is B0h, the core reads EEPROM bytes containing the chip's Vendor ID, Product ID, and release number. On device attachment or system boot up, the host uses these bytes to find a matching INF file that identifies a driver for the device. The driver contains firmware to download to the device before re-enumerating. Cypress provides instructions for building a driver with this ability.

The driver uses the vendor-specific Firmware Load request to download the firmware to the device. The firmware contains a new set of descriptors and the code the device will run. For example, a HID-class device will have report descriptors and code for transferring HID report data.

On completing the download, the driver causes the chip to exit the reset state and run the firmware. By writing to a register that controls the chip's

DISCON# pin, the firmware causes the device to electrically emulate removal from, then reattachment to the bus. The pin either pulls up or floats one end of a resistor whose opposite end connects to D+. The pin indicates device attachment when pulled up and device removal when floating. The firmware also sets ReNum to 1 to cause the 8051 core, instead of the EZ-USB core, to respond to requests at Endpoint 0.

On detecting the emulated re-attachment, the host enumerates the device again, this time retrieving the newly stored descriptors and using the information in them to select a device driver to load.

The obvious advantage to storing the firmware on the host is easy updates. To update the firmware, you just store the new version on the host and the driver sends the firmware to the device on the next power up or attachment. There's no need to replace the chip or use special programming hardware or software. The disadvantages are increased complexity of the device driver, the need to have the firmware available on the host, and longer enumeration time.

Load Firmware from EEPROM. A third mode of operation provides a way for the chip to store its firmware in an external serial EEPROM. If the first byte read from the EEPROM is B2h, the core loads the EEPROM's entire contents into RAM on power-up. The EEPROM must contain the Vendor ID, Product ID, and release number as well as all descriptors required for enumeration and whatever other firmware and data the device requires. On exiting the reset state, the device has everything it needs for USB communications. The core sets the ReNum bit to 1 on completing the loading of the code. When enumerating the device, the host reads the stored descriptors and loads the appropriate driver. There is no re-enumeration.

Run Code from External Parallel Memory. If no EEPROM is detected, or if the first byte isn't B0h or B2h, and if EA is a logic high, the chip boots from code memory on the external parallel data bus. This memory can be EPROM, EEPROM, Flash memory, or battery-backed RAM. The memory contains the descriptors and other firmware. ReNum is set to 1. The host enumerates the device and loads a driver, and there is no re-enumeration.

Cypress enCoRe II

The chips in Cypress Semiconductor's enCoRe II series (yes, that odd capitalization is how Cypress has trademarked the name) are inexpensive, low-speed controllers with an instruction set optimized for USB communications.

CPU Architecture

The enCoRe II series is the latest in Cypress' offerings of low-speed controllers. The chips are similar to the original enCoRe controllers except that the program memory is Flash memory instead of OTP EPROM. The architecture is unique to Cypress, so to program in assembly code, you'll need to learn a new instruction set. However, the instruction set is small and learning the syntax should be fairly painless if you have experience with assembly-code programming. A C compiler is also available.

The series includes chips with varying amounts of program memory, number of I/O pins, and packaging. The options include up to 256 bytes of RAM, 8 kilobytes of Flash memory, and 36 I/O pins, with two of the pins serving as the USB interface.

The chips contain internal oscillators that eliminate the need to add external crystals or resonators. The USB port can be configured for PS/2 (synchronous serial) communications to enable a pointing device to support both interfaces. When USB mode is disabled, the two USB pins can serve as a serial-programming-mode interface for Flash programming.

USB Controller

The enCoRe II controllers have three endpoints, the required Endpoint 0 plus endpoints 1 and 2 for interrupt transfers. The chip can support one interrupt IN endpoint and one interrupt OUT endpoint, or two interrupt endpoints in the same direction. Each endpoint has an 8-byte buffer in RAM. USB communications require a fair amount of firmware support, so example code is helpful.

Freescale MC68HC908JB16

Freescale Semiconductor's MC68HC08 family of 8-bit microcontrollers includes chips with Flash memory and support for low-speed USB. The MC68HC908JB16 is an example. Freescale Semiconductor was created in 2004 when Motorola, Inc., spun off its Semiconductor Products sector.

Architecture

The MC68HC08 family is an upgrade to Freescale's popular MC68HC05 family. The 'HC08 chips are faster and more efficient, and the object code is upward compatible with 'HC05 code.

The 'HC908JB16 contains 16 kilobytes of Flash memory and 21 I/O pins. Two of the I/O pins are the USB interface. Some of the other I/O pins have hardware support for synchronous serial communications and a keyboard interface. A monitor ROM enables Flash-memory programming and debugging over an asynchronous serial interface using a single pin on the chip.

USB Controller

The USB controller is low speed and supports Endpoint 0, one interrupt IN endpoint, and one endpoint that can be configured as interrupt IN or interrupt OUT.

Freescale MCF5482 ColdFire

An example of a high-end controller with USB capability is Freescale Semiconductor's MCF5482 ColdFire microprocessor. The chip contains a 32-bit CPU and a full/high-speed USB device controller plus an Ethernet controller and plenty of other I/O. A request processor automatically processes many standard USB requests. For example, on receiving a Get_Descriptor request, the request processor retrieves the requested descriptor from RAM and returns the descriptor to the host. The chip supports Endpoint 0 and seven additional endpoint addresses.

Controllers that Interface to External CPUs

A controller that interfaces to an external CPU enables you to add USB to just about any microcontroller circuit. A disadvantage is the need to use two chips, while other controllers combine the CPU and USB controller on one chip. Also, example circuits and code for USB communications using your CPU may not be available.

Controllers that interface to an external CPU may support a command set for USB-related communications, or the controller may just use a series of registers to store USB data and configuration, status, and control information.

Most interface chips have a local data bus that uses a parallel interface to communicate with the CPU. For fast transfers with external memory, many chips support direct memory access (DMA). In a device with a DMA controller, the CPU can set up a transfer that reads or writes a block of data into or from data memory without CPU intervention. For CPUs that don't have external parallel buses, a few controllers can use a synchronous or asynchronous serial interface. An interrupt pin can signal the CPU when the controller has received USB data or needs new data to send.

Table 6-4 compares a selection of interface chips. The following descriptions will give an idea of the range of chips available. New chips are being released all the time, so any new project warrants checking the latest offerings.

National Semiconductor USBN9603

National Semiconductor's USBN9603 can interface to any CPU with a parallel data bus, a Microwire interface, or even just four spare I/O pins controlled entirely in firmware to support Microwire communications.

Architecture

The '9603 has a serial interface engine for handling USB communications, USB endpoint buffers, and status and control registers. A CPU can access

Table 6-4: A Selection of USB Controllers that Interface to an external CPU.

Company	Chips	CPU Interface	Bus Speed
Agere Systems	USS-820D	Parallel	Full
FTDI Chip	FT232BM	Asynchronous serial	Full
	FT245BM	Parallel	Full
National Semiconductor	USBN9603/4	Parallel, Microwire	Full
Philips Semiconductors	PDIUSBD12, ISP1181/83	Parallel	Full
	ISP1581	Parallel	Full/High
PLX Technology	NET22272	Parallel	Full/High

the endpoint buffers and status and control registers at addresses 00h through 3Fh via an external, local bus.

The chip offers three options for accessing the local data bus: non-multiplexed parallel, multiplexed parallel, and Microwire synchronous serial.

Most CPUs with external data buses can use one of the parallel interfaces with little or no additional logic. For faster transfers of blocks of data, the chip supports a burst mode where the CPU writes a starting address to the controller chip, then transmits or receives multiple bytes at consecutive addresses. The external CPU must also support this mode. The parallel interfaces also support DMA transfers.

For microcontrollers that don't have an external parallel data bus, the '9603 offers a solution in its Microwire interface. Microwire requires just four lines and can interface to just about any microcontroller with four spare I/O pins. The interface uses data lines serial in (SIN) and serial out (SOUT), a chip select (CS), and a clock line (SYNC). Command/address and data bytes shift in and out, bit by bit, using transitions on the SYNC line as a timing reference. The external CPU controls SYNC. There is no minimum SYNC frequency, and the signal doesn't have to have a constant frequency; the CPU can toggle the line as needed. The interface just has to be fast enough to keep up with the USB traffic. If the USB port transfers only small, occasional blocks of data, you can program Microwire communications in firm-

ware. Some microcontrollers, such as National Semiconductor's COP888, have support for Microwire built in.

USB Controller

The '9603 supports seven endpoint addresses: Endpoint 0 for control transfers, three IN endpoints, and three OUT endpoints. Endpoint 0's buffer is 8 bytes; the others are 64 bytes. An endpoint can receive a packet larger than the buffer size if the firmware reads incoming data fast enough to prevent the buffer from overflowing. In a similar way, an endpoint can send a packet larger than the buffer size if the firmware writes to the buffer fast enough to prevent the buffer from emptying. The USBN9604 is an identical chip except that its chip reset also resets the chip's clock-generation circuit. The '9604 is recommended for use in bus-powered devices.

Philips Semiconductors ISP1181B

The ISP1181B from Philips Semiconductors is a full-speed chip that interfaces to an external CPU over a parallel interface.

Architecture

The chip has a serial interface engine for handling USB traffic, a configurable 8- or 16-bit data bus, and a 2-bit address bus. The controller communicates with a CPU via a command set. When address bit A0 = 1, the controller interprets the lower byte on the data bus as a command. For commands that are followed by data, the CPU sets address bit A0 = 0 and transfers data to or from a register or endpoint buffer.

The chip supports multiplexed and non-multiplexed address buses and DMA transfers.

The ISP1183 is a low-power version with an 8-bit data bus and 32 pins, compared to 48 pins on the '1181. An earlier Philips chip, the PDIUSBD12, has a similar architecture but is less capable, with a slower data bus and fewer USB endpoints.

USB Controller

The ’1181B’s USB controller supports Endpoint 0 plus up to 14 additional endpoint addresses. All enabled endpoints share 2462 bytes of buffer memory. The control endpoint has 64-byte buffers. The amount of memory allocated to each of the other endpoint addresses is configurable. Isochronous and bulk endpoints are double buffered.

Firmware controls when the chip attaches to the bus. The chip appears detached from the bus until the external CPU sends a command to switch an internal pull-up resistor onto the bus’s D+ line. The firmware-controlled connection can give the chip time to initialize on power up before being enumerated by the host.

A status output can connect to an LED that lights when a USB connection has been established and blinks on data transfers.

Philips Semiconductors ISP1581

The ISP1581 from Philips Semiconductors is a full/high-speed controller that interfaces to an external CPU over a parallel interface.

Architecture

The chip has a serial interface engine for handling USB traffic, a 16-bit data bus, and an 8-bit address bus. An external CPU can communicate with the controller by accessing a series of registers. The controller supports multiplexed and non-multiplexed address buses and DMA transfers.

USB Controller

The USB controller supports full and high speeds. In addition to Endpoint 0, the chip can support up to seven IN endpoint addresses and seven OUT endpoint addresses. All enabled endpoints share 8 kilobytes of buffer memory. The control endpoint has 64-byte buffers. The amount of memory allocated to each of the other endpoint addresses is configurable, and any of these endpoint addresses can use double buffering.

Firmware controls when the chip attaches to the bus. An external pull-up resistor connects to the chip’s RPU pin and to a pull-up voltage. After a

hardware reset, the chip appears detached from the bus until the external CPU sets a register bit that causes the chip to switch the pull-up onto the bus's D+ line. This firmware-controlled connection can give the chip time to initialize on power up before being enumerated by the host.

PLX Technology NET2272

PLX Technology, Inc.'s NET2272 is a full/high-speed chip that interfaces to an external CPU over a parallel interface. PLX Technology acquired Netchip Technology, Inc. and its USB controllers in 2004.

Architecture

A series of registers hold configuration data and other information. Packet buffers hold USB data that has been received and data that is ready to transmit. The parallel interface has 5 address bits and 16 data bits. Transfers to and from the packet buffers can be 8 or 16 bits.

The registers store status and control information and the data received in the last Setup transaction. The CPU also uses registers to read and write endpoint data from and to the packet buffers.

The '2272 supports three modes for accessing its registers. In direct address mode, the five address bits specify a register to read or write to. In multiplexed address mode, the CPU places the register address on the data bits and the '2272 reads the address on the falling edge of the ALE control signal. In indirect address mode, the CPU uses the lowest address bit to distinguish between a register address pointer (0) and data (1). The CPU writes a register address pointer to specify a configuration register and then reads or writes data at the address pointed to. Direct and multiplexed address modes can access only the registers from 00h to 1Fh, which typically contain the information accessed most frequently. Indirect address mode can access all registers. The controller also supports DMA transfers. A CPU can write to the '2272 at up to 60 Megabytes/sec. and can read from the '2272 at up to 57 Megabytes/sec (in DMA mode).

To access endpoint data in the packet buffers, the CPU selects an endpoint by writing to the Endpoint Page Select register or the DMA Endpoint Select

register and then accesses the data by reading or writing to the Endpoint Data register.

USB Controller

The ’2272’s USB controller supports full and high speeds and all four transfer types. The controller has three physical endpoints in addition to Endpoint 0. A device that needs more endpoints can use virtual endpoints, where one or more logical endpoints share a physical endpoint’s resources. The device firmware must switch resources between the logical and physical endpoints as needed.

Endpoint 0 has a 128-byte buffer, and the other endpoints share 3 kilobytes of packet buffers. Two of the endpoints can use double buffers. On receiving a Setup packet, the device firmware must read the request and provide any data to return to the host. After a failed IN or OUT transaction, an endpoint automatically recovers and waits for the host to retry.

FTDI Chip FT232BM and FT245BM

Future Technology Devices International (FTDI) Chip offers controllers that take a different approach to USB design. The FT232BM USB UART and FT245BM USB FIFO are interface chips that manage enumeration and other bus communications completely in hardware. The chips are designed for use with host drivers provided by FTDI Chip. The controllers require no USB-specific firmware at all, though you can use an EEPROM to store values for some items in the descriptors. The device firmware only needs to provide data for the controller to send and retrieve received data. Because the USB communications are handled entirely in hardware and use FTDI Chip’s driver, you can even use FTDI Chips’ Vendor ID in devices you develop and market.

These controllers can be a good solution if your device doesn’t require a standard class driver and you need no more than one bulk or isochronous port in each direction

Architecture

Both chips are full speed. The FT245BM has a parallel interface and the FT232BM has an asynchronous serial interface.

Table 6-5 shows the functions of the pins on the ’245BM. The parallel interface has 8 data lines and four handshaking signals. The names of the handshaking signals are from the perspective of the external CPU that interfaces to the chip. The RXF# output is low when the CPU can read a byte received from the host. The CPU strobes RD# to read the byte. In the other direction, the TXE# output goes low when the CPU can write a byte to send to the USB host, and the CPU strobes WR to write the byte into the ’245BM’s buffer. The external CPU can use a data bus or any spare port pins to access the ’245BM.

The ’232BM converts between USB and an asynchronous serial interface. Table 6-6 shows the functions of the pins. The serial interface includes a TXD data output, an RXD data input, and pins for standard RS-232 handshaking signals (RTS, CTS, DTR, DSR, DCD, and RI). A TXDEN output is high when data is transmitting on TXD. This output can interface directly to the transmit-enable input of an RS-485 transceiver, eliminating the need to enable the transmitter using firmware and additional hardware. The ’232BM functions as a DTE as defined by the EIA/TIA-232 standard. (The RS-232 ports on PCs are also DTEs.) On a DTE, the TXD, RTS#, and DTR signals are outputs, and the RXD, CTS#, and DSR signals are inputs. A device that functions as a DCE has complimentary signals. (For example, TXD is an input and RXD is an output.) To connect two DTEs to each other, use a null-modem cable that swaps the signal pairs so each output connects to its corresponding input.

To create a USB/RS-232 converter, use a Maxim MAX3245 or similar chip to convert between the ’232BM’s 5V logic signals and RS-232 voltages. In a similar way, you can interface the ’232BM to an RS-485 transceiver. Chapter 7 has more about using the ’232BM to convert devices to USB from these legacy interfaces.

Both chips also support a Bit Bang mode, where the chip operates as a very basic controller without requiring a connection to an external CPU. On the

Table 6-5: Pinout of the FT245BM USB FIFO.

Pin	Name	I/O	Description
1	EESK	Output	EEPROM clock
2	EEDATA	Output	EEPROM data
3	VCC	Power	+4.35 to 5.25V
4	RESET#	Input	Reset the chip
5	RSTOUT#	Output	Output of the Reset Generator
6	3V3OUT	Output	Regulated +3.3V output
7	USBDP	I/O	D+ USB data
8	USBDM	I/O	D- USB data
9	GND	Power	Ground
10	PWREN#	Output	Goes low when device is configured, goes high in Suspend state
11	SI/WU	IN	Send USB data on next bulk IN/request remote wakeup
12	RXF#	Output	Goes low when the FIFO contains data that the CPU can read
13	VCCIO	Power	+3.0 to 5.25V
14	TXE#	Output	Goes low when the CPU can write data into the FIFO
15	WR	Input	On H > L transition, writes D0–D7 to the transmit FIFO buffer
16	RD#	Input	On H > L transition, places a byte from the receive FIFO buffer on D0–D7 for the CPU to read
17	GND	Power	Ground
18	D7	I/O	Data bit 7
19	D6	I/O	Data bit 6
20	D5	I/O	Data bit 5
21	D4	I/O	Data bit 4
22	D3	I/O	Data bit 3
23	D2	I/O	Data bit 2
24	D1	I/O	Data bit 1
25	D0	I/O	Data bit 0
26	VCC	Power	+4.35 to 5.25V
27	XTIN	Input	Crystal oscillator cell input
28	XTOUT	Output	Crystal Oscillator cell output
29	AGND	Power	Analog ground
30	AVCC	Power	Analog power supply
31	TEST	Input	Bring high to enable Test mode
32	EECS	I/O	EEPROM Chip Select

Table 6-6: Pinout of the FT232BM USB UART.

Pin	Name	I/O	Description
1	EESK	Output	EEPROM clock
2	EEDATA	Output	EEPROM data
3	VCC	Power	+4.35 to 5.25V
4	RESET#	Input	Reset the chip
5	RSTOUT#	Output	Output of the Reset Generator
6	3V3OUT	Output	Regulated +3.3V output
7	USBDP	I/O	D+ USB data
8	USBDM	I/O	D- USB data
9	GND	Power	Ground
10	SLEEP#	Output	Goes low in Suspend state
11	RXLED#	Output	Receive LED driver, open-collector
12	TXLED#	Output	Transmit LED driver, open-collector
13	VCCIO	Power	+3.0 to 5.25V
14	PWRCTL	Input	Tie low for bus power, high for self power
15	PWREN#	Output	Goes low when device is configured, goes high in Suspend state
16	TXDEN	Output	Transmit enable for RS-485
17	GND	Power	Ground
18	RI#	Input	Ring Indicator
19	DCD	Output	Data Carrier Detect
20	DSR#	Input	Data Set Ready
21	DTR#	Output	Data Terminal Ready
22	CTS#	Input	Clear To Send
23	RTS#	Output	Request To Send
24	RXD	Input	Receive Data
25	TXD	Output	Transmit Data
26	VCC	Power	+4.35 to 5.25V
27	XTIN	Input	Crystal oscillator cell input
28	XTOUT	Output	Crystal Oscillator cell output
29	AGND	Power	Analog ground
30	AVCC	Power	Analog power supply
31	TEST	Input	Bring high to enable Test mode
32	EECS	I/O	EEPROM Chip Select

’245BM, the data-bus pins function as an 8-bit I/O port. On the ’232BM, the data and handshaking pins are the I/O port. You can use FTDI Chip’s driver to configure the pins as inputs or outputs in any combination. The outputs can control LEDs, relays, or other circuits. The inputs can interface to switches and logic-gate outputs. Host applications can read and write to the I/O pins over the USB connection.

USB Controller

Unlike other device controllers, the ’232BM and ’245BM aren’t designed as general-purpose devices that can be programmed to use any host driver. Instead, FTDI Chip offers two driver options, a Virtual COM Port Driver and a D2XX Direct Driver.

With the Virtual COM Port driver, the device appears to the host as if the device were connected to a COM (RS-232) port. In most cases, an RS-232 device converted to USB with a ’232BM requires no changes to application software that accesses the device. Under Windows, applications can access a device with a ’232BM using standard API functions (ReadFile, WriteFile) or other classes, libraries, or toolkits for COM-port communications. The ’245BM can use the Virtual COM Port driver as well.

If you don’t want to use COM-port programming, need faster performance, or want to use Bit Bang mode, FTDI Chip provides the D2XX Direct Driver, which provides a series of vendor-specific functions that applications can use to communicate with the device.

The chips support a Microwire interface to an EEPROM that can store vendor-specific values for items such as a Vendor ID, Product ID, strings that contain a serial number, manufacturer, and product description, and specifying whether the device is bus- or self-powered. If there is no EEPROM data for an item, the controller uses a default value. FTDI Chip provides a utility that programs the information into an EEPROM connected to a ’232BM or ’245BM.

With no EEPROM, the chips use FTDI Chips’ Vendor ID and Product ID. On request, FTDI Chip will also grant the right for your device to use their Vendor ID and a Product ID that FTDI Chip assigns to you. Eliminating

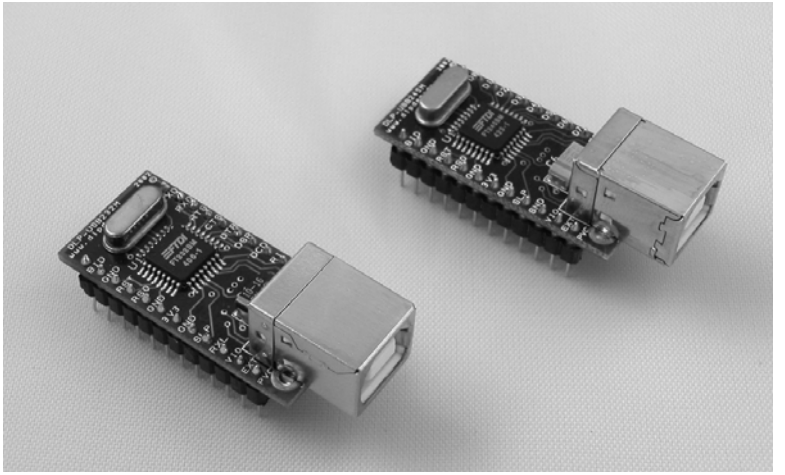

Figure 6-2: For easy prototyping with FTDI Chips' controllers, use DLP Design's DLP-USB232M and DLP-USBS245M modules.

the need to buy a Vendor ID is a huge advantage for developers of inexpensive products that sell in small quantities.

Both chips have a 384-byte transmit buffer and a 128-byte receive buffer. The '245BM's data bus can transfer data at up to 1 Megabyte/sec. The '232BM's asynchronous serial port can transfer data at up to 3 Million baud, which works out to 300 kilobytes/sec. with one Start bit and one Stop bit per byte.

The chips use bulk transfers by default. A driver for isochronous transfers is also available.

Another controller from FTDI Chip is the FT2232C Dual USB UART/FIFO. The chip contains two controllers that each support several configurations. The options include the equivalent of a '232BM or '245BM interface, a synchronous serial interface, and an 8051-compatible parallel interface. A fast, optoisolated serial-interface mode enables creating an isolated synchronous interface using external optoisolators. A high-drive-level option enables the I/O pins to source and sink up to 6 milliamperes (at 3.2V minimum for source current and 0.6V maximum for sink current).

All of the chips are available in surface-mount packages only. For easy prototyping, a variety of sources provide circuit boards that contain a controller chip, EEPROM, a USB connector, and headers for easy attachment to your CPU and other circuits. One source is DLP Design, whose DLP-USB232M and DLP-USBS245M modules are circuit boards mounted on 24-pin DIP sockets (Figure 6-2). The circuits on the boards are similar to those in FTDI Chip's example schematics.

Chapter 14 has more about designing devices using these chips, including example applications.