Firmware Considerations for the Cypress Semiconductor CY7C63xxx USB Microcontroller Family

Introduction

This application note provides information that will aid in the development of firmware for the Cypress CY7C63xxx family of low-speed USB microcontrollers. The intention is to provide firmware developers with some useful insights into the intricacies of the devices. This document is divided into two sections. Section 1 provides general firmware tips and techniques to the firmware developer. Section 2 provides information on how to write firmware routines to perform USB communication.

For additional information on firmware development for the CY7C63xxx devices, please refer to the CY7C63xxx datasheets and the CYASM Assembler User's Guide. Reference designs (including firmware source code) for the USB mouse, joystick, and keyboard are also available with the purchase of a CY3650/3651 Developer's Kit. To purchase these kits, please call your local Cypress Sales Office. All other product information can be found at the Cypress Internet website: http://www.cypress.com.

General USB information, including the USB Specification 1.0 and specific device class specifications, can be found at the USB Implementers Forum (USB-IF) website for developers: http://www.usb.org/developers.

Firmware Tips and Techniques

CPU Version Differences

There are two different CPU versions that exist in the Cypress USB microcontroller family. The CY7C630/1/2xx contain CPU A, while the CY7C634/5xx contain CPU B. The differences between CPU A and CPU B are in their instruction sets.

CPU A and B Instructions That Are Different

Table 1. CPU A and B Instructions That Are Different

Instruction	Opcode	Description
RET	3Fh	Return (flags handled differently)

CPU A restores the C and Z flags from stack when RET is executed. CPU B does not alter the C and Z flags when RET is executed.

CPU A Instructions That Are Not in CPU B

Table 2. CPU A Instructions Only

Instruction	Opcode	Description
IPRET	1Eh	I/O write, pop, return

CPU A disables interrupts by clearing the Global Interrupt register (0x20). That means interrupt service routines (ISR) always had a form like this:

```
PUSH A ; save accumulator
<body> ; do something
MOV A, [interrupt_mask] ; load interrupt
mask
IOWR GLOBAL_INTERRUPT ;enable interrupts
POP A ; restore accumulator
RET ;return
```

The problem with this approach is that another interrupt could occur after the interrupts were enabled and before either the POP or RET instructions were executed. In an interrupt rich environment, the nesting of successive interrupts could cause the program and data stacks to grow until the firmware crashed.

The IPRET instruction was added to fix this problem. The idea is to combine the last three instructions of an ISR into one atomic instruction. The effect is to ensure the stacks are restored at the end of interrupt service before another interrupt can occur. The ISRs now have a form like this:

```
PUSH A
<body>

MOV A,[interrupt_mask]
IPRET GLOBAL_INTERRUPT
```

In this way, the stacks do not grow to excess and crash the firmware.

CPU B Instructions That Are Not in CPU A

Table 3. CPU B Instructions Only

Instruction	Opcode	Description
CALL	5xh	Call address in upper 4KB of PROM
DI	70h	Disable Interrupts
EI	72h	Enable Interrupts
MOV X,A	40h	Move A into X
MOV A,X	41h	Move X into A
MOV PSP,A	60h	Move A into Program Stack Pointer
RETI	73h	Return from Interrupt

The whole interrupt enable/disable mechanism was revised in CPU B. The mechanism combines hardware and software control without writing over the Global Interrupt Enable register (0x20), thus eliminating the need to restore its contents prior to returning from the ISR (required for CPU A). The hard-

ware control disables interrupts when an interrupt is acknowledged and enables interrupts when RETI is executed at the end of an ISR. The format of an ISR in CPU B looks as follows:

```
PUSH A
<body>
POP A
RETI
```

The software can enable an interrupt by writing to register 0x20 and executing the EI instruction. Interrupts can be disabled in software by executing the DI instruction. The EI instruction must be used at least once in the Reset routine (vector 0, ROM address 0000h) to enable interrupts for the first time. Subsequently, these instructions can be used to control the nesting of interrupts within ISRs.

The added CALL instruction allows the firmware to call subroutines in the upper 4 KB of ROM. Refer to the next topic, Program Memory Organization, for more information about this instruction.

The added MOV instructions allow more flexible usage of the X register, and the MOV PSP,A instruction allows the firmware to initialize the program stack pointer.

Program Memory (ROM) Organization

The program memory (ROM) organization is shown in *Figure 1*. The initial part of the ROM, beginning with address 0000h, contains the Interrupt Vector Table where each vector location contains a JMP ISR_address instruction. The remainder of the ROM is left for user code. The user code section can contain both program code as well as ROM data tables. ROM data tables are defined using the DB, DS, DSU, DW, and DWL instructions. ROM data can be read into the accumulator using the INDEX instruction.

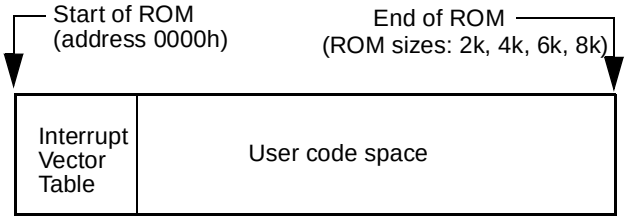

Figure 1. Program Memory (ROM) Organization

Program Counter

The program counter (pc) is composed of two 8 bit registers, pcl (low byte) and pch (high byte). The lower 6 bits of pch and all 8 bits of pcl form a 14 bit address. Thus, the microcontrollers can address up to 16 KB of program memory. Since the largest EPROM size that the CY7C63xxx family has is 8 KB, not all program counter bits are used (i.e., only 13 bits are needed for an 8-KB EPROM).

Memory Pages

The program memory is organized into 256 byte pages, such that the pch register contains the memory page number, and the pcl register contains the offset into that particular memory page.

When the pcl register (page offset) increments above the value FFh, the pch register (page number) is not automatically incremented by the CPU. Since the pcl register would wrap around to 00h while pch stayed the same, the firmware would once again execute instructions at the beginning of the same memory page, and never be able to execute instructions in the next page. To prevent this, a special instruction called XPAGE is used to increment the page number.

Table 4. XPAGE Instruction

Instruction	Opcode	Description
XPAGE	1Fh	Increments the memory page number (pch)

By default, the CYASM assembler automatically inserts XPAGE instructions at the last location of the memory page to increment pch. This has the effect of moving the instruction that would have been last on one page to the first location of the next page. For two byte instructions starting two bytes from the end of a page, a NOP instruction is placed before the XPAGE so that both bytes of the instruction are forced onto the next page.

Additionally, the assembler provides the XPAGEON and XPAGEOFF directives to control XPAGE insertion. For instance, the XPAGEOFF directive should be used prior to the definition of any ROM data tables to prevent data corruption by the insertion of XPAGE opcodes.

4-KB Boundary

The CY7C634/635xx family can have up to 8 KB of EPROM for program code space. For these microcontrollers, it is important to distinguish the lower 4 KB from the upper 4 KB of ROM, due to the fact that there are some inherent limitations to using the upper 4 KB. For instance, the INDEX (Fxh), CALL (9xh), and jump instructions only accept a 12-bit address operand. With these instructions, the firmware is limited to execute in the lower 4 KB. Furthermore, since the interrupt vector table for the CY7C63xxx family begins at program address 0000h, and each vector location contains a two-byte JMP instruction (8xh) with a 12-bit address operand, all interrupt service routines (ISRs) must begin in the lower 4 KB.

To make use of the upper 4 KB, a long CALL instruction (5xh) was added to the instruction set of CPU B (see *Table 3*). When the assembler sees a CALL instruction with an address operand greater than 12 bits, it uses opcode 5xh with the lower 12 bits of the destination address instead of using opcode 9xh. The long CALL instruction allows the firmware to access subroutines in the upper 4 KB by setting bit 13 of the program counter to 1 so that the computed address will be in the upper 4 KB. Once a subroutine in the upper 4 KB is being executed, the address operands for INDEX, CALL (9xh), and jump instructions refer to locations in the upper 4 KB. The execution of the matching RET instruction to the long CALL will restore the program counter to its original location in the lower 4 KB, since all 14 bits of the return address would be popped off the program stack into the program counter.

Please note that there is no way to call subroutines located in the lower 4 KB from the upper 4 KB. However, if an interrupt is generated while executing a subroutine in the upper 4 KB, the interrupt service routine (ISR) can still be executed even though it is located in the lower 4 KB. Prior to calling the appropriate ISR, the 14-bit program counter, along with the zero and carry flags, are pushed onto the program stack by the

CPU. Upon execution of the RETI instruction, the CPU will pop all 14 bits of the program counter, as well as the zero and carry flags, from the program stack so that once the ISR is complete, execution can resume in the upper 4KB.

Data Memory (RAM) Organization

The CY7C63xxx provide two different RAM sizes for data storage and USB communication: 128 bytes (CY7C630/1/2xx) and 256 bytes (CY7C634/5xx). The data RAM is organized around 4 major components: the program stack, the data stack, user-defined variables, and the USB endpoint FIFOs. Of the four components, only the endpoint FIFOs have fixed locations (near the top of RAM): the CY7C630/1/2xx have two 8 byte FIFOs from addresses 70h to 7Fh, and the CY7C634/5xx have three 8 byte FIFOs from addresses E8h to FFh. The endpoint FIFOs are used as communication buffers to send and/or receive data from the USB host.

The program stack pointer (psp) points to the top of the program stack, and the data stack pointer (dsp) points to the top of the data stack. The program stack grows upward in RAM by incrementing the psp, while the data stack grows downward by decrementing the dsp. Upon reset, both the psp and the dsp are initialized to RAM address 00h. Since the data stack grows by decrementing the dsp, the dsp needs to be moved higher in RAM to avoid wrapping around to the top of RAM and overwriting the endpoint FIFOs. This is accomplished by the SWAP A, DSP instruction.

Table 5. SWAP A, DSP instruction

Instruction	Opcode	Description
SWAP A, DSP	30h	Swaps the contents of A with that of the DSP

Typically, the value of the DSP is set during the first few instructions of the Reset vector (vector 0, ROM address 00h). The code fragment looks as follows:

```
MOV A, data_stack_start_address
SWAP A, DSP
```

A typical data memory organization is shown in *Figure 2*. As stated earlier, the endpoint FIFOs are located at the top of RAM. The program stack is located at the bottom of RAM, and the data stack is located above the program stack. The beginning address of the data stack should be chosen high enough so that the two stacks do not grow into each other. Finally, the remaining memory space between the data stack and the endpoint FIFOs is left for user-defined variables.

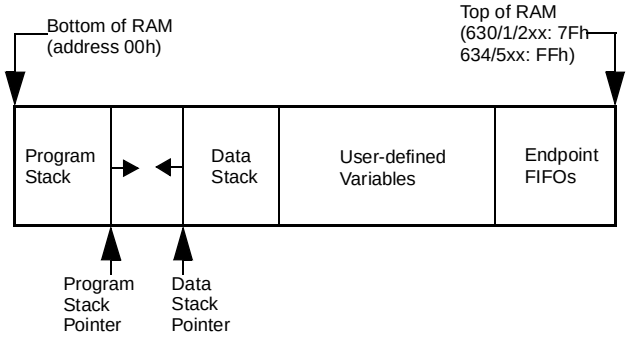

Figure 2. Data Memory (RAM) Organization

Conditional Jumps

The conditional jump instructions are shown below in *Table 6*.

Table 6. Conditional Jump Instructions

Instruction	Opcode	Description
JZ address	Axh	Jump to address if zero flag is set
JNZ address	Bxh	Jump to address if zero flag is not set
JC address	Cxh	Jump to address if carry flag is set
JNC address	Dxh	Jump to address if carry flag is not set

The CPU requires only 4 cycles to execute a conditional jump instruction when the jump is not taken. Otherwise, the instruction takes 5 cycles, as stated in the CYASM Assembler User's Guide.

Suspend and Resume

The CY7C63xxx family provides a suspend mode in which the microcontroller can conserve power by stopping the clock oscillator and timers, as well as powering down the microcontroller. This feature is used primarily to comply with the USB Specification 1.0 which states that devices are required to go into suspend mode after detecting 3 ms of no USB bus activity, and must consume less than 500 µA of current. The state of the bus (active, idle) can be determined by reading the Bus Activity bit of the USB Status and Control Register (bit 0 of register 0x13 in the CY7C630/1/2xx; bit 3 of register 0x1F in the CY7C634/5xx). When the Bus Activity bit is set to 1, the bus is active. Otherwise, the bus is idle. Note that this bit is a sticky bit and must be cleared prior to reading the state of the bus. The 1-ms interrupt service routine is the most appropriate place to detect the idle bus condition (since it can most easily time the 3-ms idle duration required to put the device into suspend mode).

The firmware can put the device into suspend mode by writing a 09h to the Status and Control Register (0xFF). This will simultaneous set the Suspend bit (bit 3) and the Run bit (bit 0) to 1, allowing the controller to wake up under certain conditions. The wake-up, or resume, conditions for the CY7C630/1/2xx family are:

- USB bus activity
- Cext interrupt
- GPIO interrupt

The resume conditions for the CY7C634/5xx family are:

- USB bus activity
- GPIO interrupt

The following code fragment specifies how to put the microcontroller into suspend mode:

```
MOV A, 09h
IOWR FFh
NOP
```

Once the IOWR instruction is executed, the device immediately goes into suspend mode. When one of the wake-up con-

ditions occurs, the CPU will execute the next instruction immediately after the IOWR. If an interrupt causes the resume, the ISR would be delayed by this next instruction. For this reason, it is a good idea to use a NOP right after the IOWR. This prevents the possibility of the next instruction from altering the state of the device in a way that would affect the execution of the ISR.

Watchdog Timer and Reset

The CY7C63xxx has a Watchdog Timer that prevents the device from hanging up or getting stalled because of some firmware mishap. The Watchdog Timer increments every 1.024 ms. If it is allowed to count to roughly 8 ms, a Watchdog Reset will occur (resetting the controller). The Watchdog Reset bit in the Status and Control register (bit 6 of register 0xFF) will be set to 1 to record this event. To prevent this, the Watchdog Timer must be cleared prior to reaching 8 ms. This is accomplished by a write to the Watchdog Timer Clear register (0x21 in 630/1/2xx, 0x26 in 634/5xx):

```
IOWR Watchdog_timer_clear
```

For some applications, the above instruction can be placed in the 1-ms ISR, thereby ensuring that the Watchdog Timer is cleared every millisecond. However, this method does not fully utilize the benefit of this reset in the event that the firmware mainloop gets stalled, since the Watchdog reset will not occur. In these cases, it is better to put the IOWR instruction within the mainloop, while ensuring that the delay between writes to the register never exceeds 8 ms. For additional robustness, RAM data variables can be used to save the state of the device prior to the Watchdog Reset, such that the Reset routine (vector 0, ROM address 0000h) can allow the device to recover in a good state.

In the CY7C630/1/2xx, the Watchdog Reset bit in the Status & Control register disables the SIE output drivers when it is set to 1 (either by firmware or by a Watchdog Reset event). The SIE will still respond to packets sent to the device by the USB host, but will not be able to send packets.

USB Communication

The CY7C63xxx support the two types of low-speed USB communication transactions defined in chapter 5 of the USB Specification 1.0: control transfers and interrupt transfers. Both types of transfers are accomplished using device endpoints. Each endpoint has an associated 8 byte communication FIFO, control registers, and is capable of generating interrupts to the CPU.

Control Transfers

Control transfers are bidirectional transfers that take place over the Control Pipe. The Control Pipe is a message pipe that connects the USB host to endpoint 0 of the USB device. Control transfers take the form of USB host-initiated requests followed by device responses. The request is structured as a SETUP token packet followed by a DATA0 packet containing 8 bytes of data that describe the request. Please refer to Chapter 9 of the USB Specification 1.0 for a description of the standard USB control requests.

There are 3 basic types of control transfers that the host can initiate: Control Read, Control Write, and No Data Control. These transfers can be organized into three separate transaction stages: a Setup Stage, a Data Stage, and a Status Stage (see *Table 7*). Further information on Control Transfers can be found in Section 8.5.2 of the USB Specification 1.0.

Table 7. Control Transfers

Transfer Type	Setup Stage	Data Stage	Status Stage
Control Read	SETUP	IN...IN	OUT
Control Write	SETUP	OUT...OUT	IN
No Data Control	SETUP	-	IN

Setup Stage

The Setup Stage transaction consists of a SETUP token packet followed by a DATA0 packet containing 8 data bytes that describe the request sent from the host to the device. See section 9.3 of the USB Specification 1.0 for a description of the SETUP data. The device sends an ACK handshake packet to the host to acknowledge receipt of the SETUP data.

Data Stage

The Data Stage transaction is used to transfer data between the host and the device. The data will be sent in 8 byte packets beginning with the DATA1 PID. Subsequent data packets will have the PID toggled between DATA0 and DATA1. The final packet will have a data payload of 8 bytes or less.

A Control Read transfer will move data from the device to the host. For each data packet, the host will send an IN token packet. The device will either respond with a DATA packet (followed by an ACK from the host), a NAK handshake packet (indicating the device is busy), or a STALL handshake packet (indicating an error has occurred, e.g., an invalid command has been received).

A Control Write transfer will move data from the host to the device. Prior to each data packet, the host will send an OUT token packet. The device will either respond with an ACK handshake packet (acknowledging data reception), a NAK handshake packet, or a STALL handshake packet.

No Data Control transfers do not have a Data Stage.

Status Stage

The Status Stage transaction indicates completion of the entire transfer. The transfer direction of the Status Stage is opposite to that of the prior Data Stage.

For Control Read transfers, the host sends an OUT token packet followed by a zero-length DATA1 packet. The device responds with an ACK (completing the Status Stage), NAK, or STALL.

For Control Write and No Data Control transfers, the host sends an IN token packet. The device responds with a zero-length DATA1 packet to acknowledge the Status Stage (followed by an ACK from the host), a NAK, or STALL.

Interrupt Transfers

Interrupt transfers are unidirectional data transfers between the host and the device over an Interrupt Pipe. The Interrupt Pipe is a stream pipe for which no structure is imposed on the data. Version 1.0 of the USB Specification only provides for

interrupt transfers from the device to the host (IN transfers). Subsequent versions of the specification will likely provide for host to device transfers (OUT transfers), but these will not be discussed in this document. Interrupt transfers are single-stage transactions consisting of a direction token packet (IN, OUT), a DATA0/1 packet, and/or a handshake packet (ACK, NAK, STALL).

IN Interrupt Transfers

A low-speed endpoint (other than the control endpoint, endpoint 0) can be setup as an IN Interrupt endpoint with a polling rate between 10 ms to 255 ms. For each polling interval, the USB host sends an IN token packet over the Interrupt Pipe to the interrupt endpoint. The device can respond with a DATA0/1 packet containing 0 to 8 bytes of data or a NAK handshake packet (indicating that no data is required to be sent during this interval).

Enumeration

When a USB device is first attached to the USB bus, the USB host enumerates the device. Enumeration is the process of identifying and configuring the newly attached device through a series of Control Transfers between the host and the device. There are six enumeration steps for all USB devices:

1. USB Bus Reset
2. Get Device Descriptor using default address 0
3. Set Device Address
4. Get Device Descriptor using new address
5. Get Configuration Descriptor
6. Set Configuration

More information on each of these requests as well as the various descriptors can be found in Chapter 9 of the USB Specification 1.0.

USB Bus Reset

The host sends a Single-Ended Zero (SE0) signal, where both D+ and D– are low, for 10 ms. At this point, endpoint 0 of the device should be enabled to accept SETUP token packets.

Get Device Descriptor (address 0)

The USB host sends a control request (endpoint 0) to device address 0 (default address) to get the first 8 bytes of the Device Descriptor from device. The request follows the protocol of a Control Read transfer. *Figure 3* shows an example USB bus trace of for this control request. Each packet of the request transaction is listed on a separate line, with the sender of each packet listed to the left.

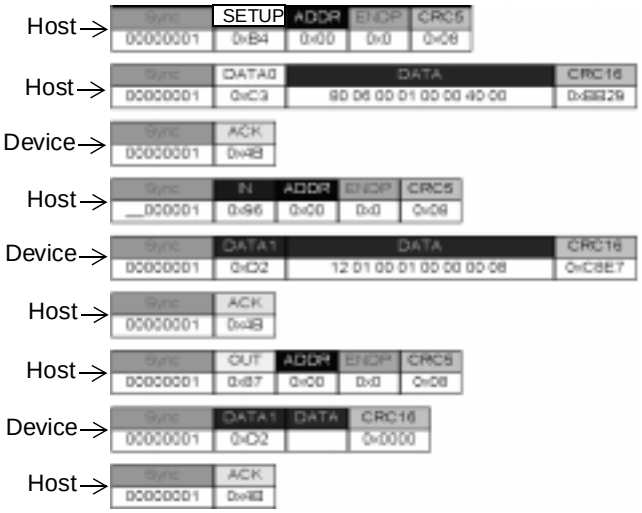

Figure 3. Get Device Descriptor Request (address 0)

Set Device Address

The USB host sends a Set Address control request to give the newly attached device an address for future communication. This request follows the No Data Control protocol (see *Figure 4*).

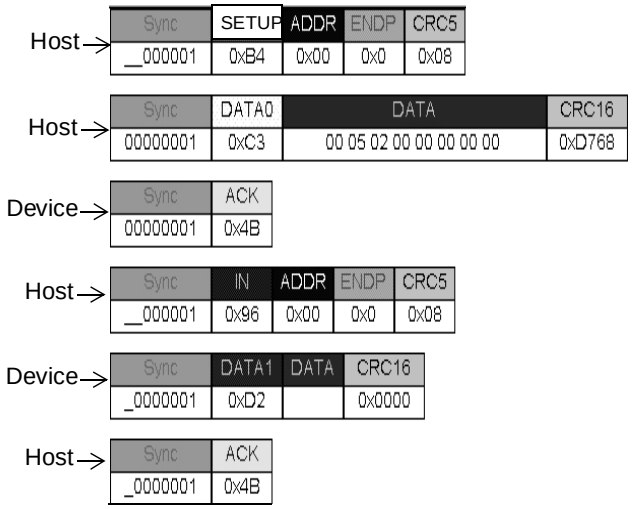

Figure 4. Set Address Request

Get Device Descriptor (new address)

The USB host sends a request to get the entire Device Descriptor, but this time it uses the new address given to the device. Once again, this request follows the Control Read protocol. In the *Figure 5* example, the new address is 0x02.

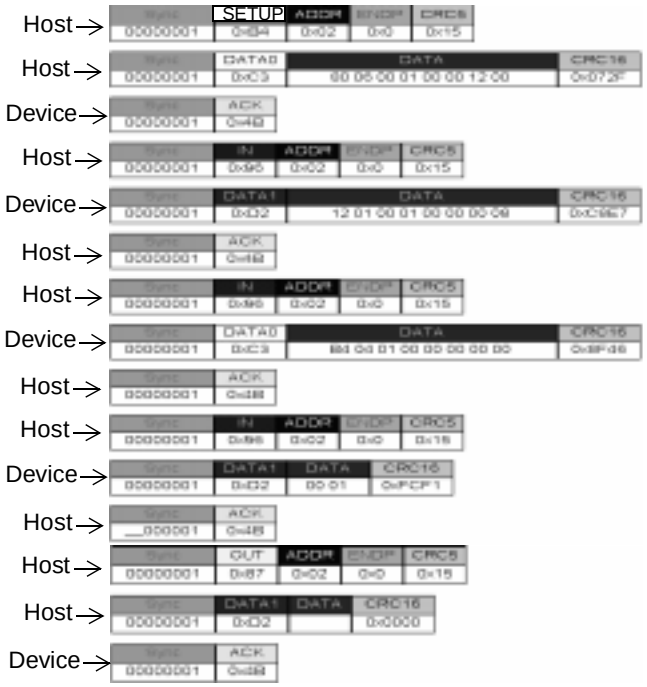

Figure 5. Get Device Descriptor Request (new address)

Get Configuration Descriptor

The USB host sends a control request to get the device's Configuration Descriptor. This request also follows the Control Read protocol. This request requires that the device will send four descriptors in response to this request:

- Configuration Descriptor
- Interface Descriptor(s)
- Class Descriptor(s) (if any)
- Endpoint Descriptor(s)

The host can send multiple requests for Configuration Descriptors if the Device Descriptor specifies multiple configurations. Each Configuration Descriptor specifies a configuration of interfaces and endpoints that can be enabled by the host. *Figure 6* shows the device sending a single Configuration Descriptor (configuration value 1).

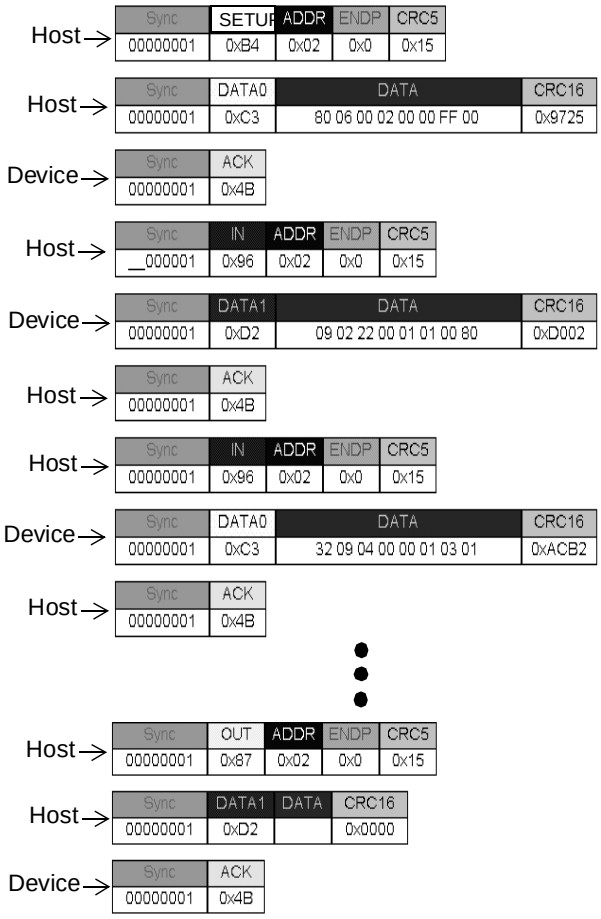

Figure 6. Get Configuration Descriptor Request

Set Configuration

The USB host selects one of the configurations that it identified from the previous request. At this point, the device is in the 'configured' state and the endpoint(s) defined for the selected configuration are enabled. Alternatively, the host can select configuration 0 which puts the device in an 'unconfigured' state where all endpoints (other than the control endpoint) are disabled. This request follows the No Data Control protocol. *Figure 7* shows an example of the USB host selecting configuration value 1 for the device.

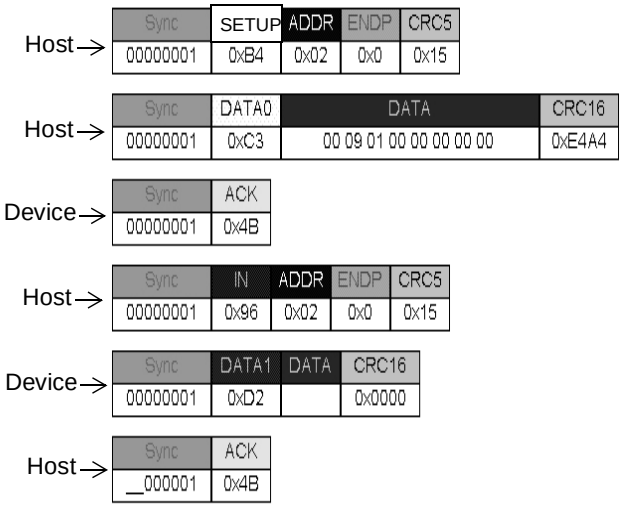

Figure 7. Set Configuration Request

Data Reports

Once configured, the device can enable the endpoint(s) defined in the selected configuration. In this example, endpoint 1 is enabled as an IN interrupt endpoint with a polling rate of 10ms. *Figure 8* shows two device responses to IN token packets sent to endpoint 1:

- a DATA packet is sent
- a NAK token packet is sent

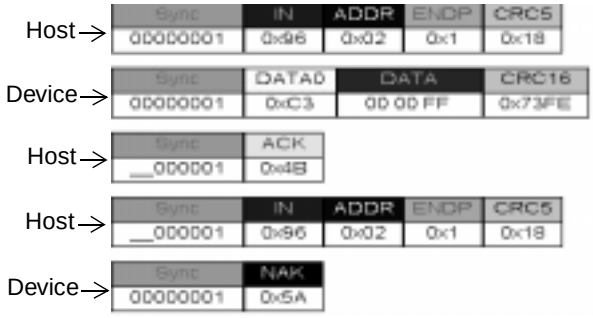

Figure 8. Endpoint 1 Transactions

The interrupt endpoint can be used to send data reports to the USB host. However, since the USB Specification 1.0 has not defined an OUT interrupt endpoint, the host is restricted to using control transfers to send data to the device. One such control request is Set Report which is defined for the Human Interface Device (HID) class of devices such as mice, keyboards, and joysticks. Further information on the HID class can be found in the Device Class Definition for HID 1.0.

The Set Report control request follows the Control Write protocol. See *Figure 9*.

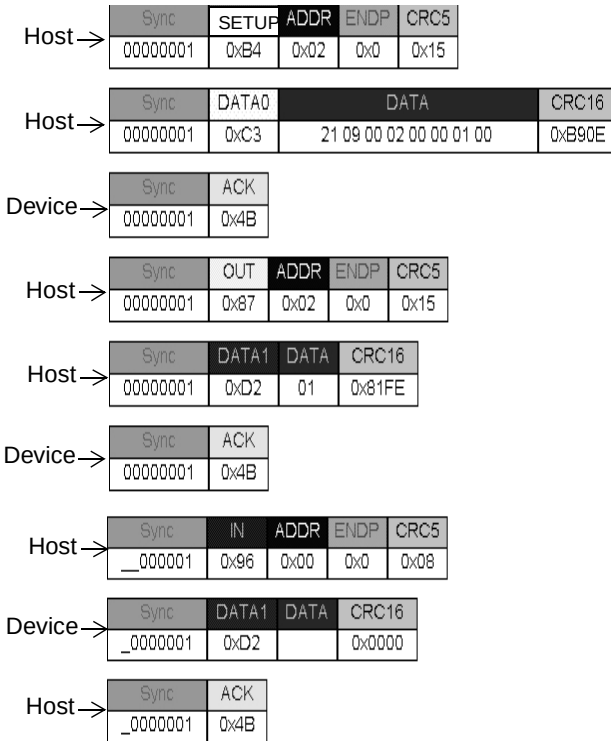

Figure 9. Set Report Request

Endpoint Communication with the CY7C630/1/2xx

The CY7C630/1/2xx have two endpoints: endpoint 0 for control transfers and endpoint 1 for interrupt transfers. The USB Device Address Register (0x12) specifies the address that the device will respond to. Upon reset, this register is set to 0s so that the Serial Interface Engine (SIE) is ready to respond to the default address (address 0). During enumeration, the USB host will send a Set Address control request with a new device address. This new address should be stored in register 0x12 once the No Data Control transfer completes its Status Stage.

Endpoint 0

The endpoint 0 FIFO is located from RAM address 70h to 77h. Endpoint 0 can generate interrupts to the CPU after data reception to or transmission from the endpoint 0 FIFO. Interrupt vector 3 (ROM address 0006h) should contain a JMP EP0_ISR instruction, where EP0_ISR is the beginning address of the endpoint 0 interrupt service routine. To enable endpoint 0 interrupts, bit 3 of the Global Interrupt Enable Register (0x20) should be set to 1. This is normally done in the Reset routine after a USB bus reset occurs. The USB Reset bit, bit 5, of the Status and Control Register (0xFF) is set to 1 when a bus reset is issued by the host. The Reset routine can read this bit to differentiate between a USB reset and a Power-on reset (bit 4). If a Watchdog reset occurs (bit 6 of 0xFF is set to 1), then the SIE drivers are disabled. There are three registers used to control communication to/from endpoint 0: the USB Endpoint 0 RX Register (0x14), the USB Endpoint 0 TX Configuration Register (0x10), and the USB Status and Control Register (0x13). The control logic for both the send

and receive functions for endpoint 0 are distributed over these three registers.

The receive control is primarily located in the RX register (0x14), however bit 4 of the TX register (0x10) specifies the validity of the data received from a SETUP or OUT data stage (i.e., CRC, PID, or bitstuffing error). Register 0x14 records information pertaining to packets sent to endpoint 0 by the USB host: PID type (bits[2:0]), data toggle setting (bit 3), and number of bytes received, including the 2 CRC bytes (bits[7:0]). Three PID types can be recorded: SETUP (bit 0), IN (bit 2), and OUT (bit 1).

The TX register controls how the endpoint will respond to IN packets: send data (bit 7 set to 1), send NAK (bit 7 set to 0), or send STALL (bit 5 set to 1). Bits[3:0] specify the number of bytes to send to the host. Bit 6 specifies the data toggle setting (DATA0/1). After bit 7 is set to 1 to enable a data transmission, the bit will be cleared once the host sends an ACK. The endpoint 0 ISR can read this bit to determine when a transmission has completed.

The Enable Outs bit (bit 4) and Status Outs bit (bit 3) of the USB Status and Control register (0x13) specify how endpoint 0 will respond to the reception of an OUT data stage or a Status Stage OUT packet. If bit 4 is set to 1 while bit 3 is set to 0 (as well as the Stall bit, bit 5, of 0x10), then OUT data bytes will be written to the endpoint 0 FIFO (the OUT bit and Count bits of the RX register will be updated as well). If bit 3 is set to 1 (bit 4 set to 0), then the endpoint 0 logic will check to make sure that a valid Status Stage OUT has been received (0 data bytes), and no data will be written to the FIFO.

Endpoint 1

The endpoint 1 FIFO is located from RAM address 78h to 7Fh. This endpoint can only send data from the device to the host, and is used as an IN interrupt endpoint. Interrupts can be generated to the CPU when data is transmitted from endpoint 1. Interrupt vector 4 (ROM address 0008h) should contain a JMP EP1_ISR instruction, where EP1_ISR is the beginning address of the endpoint 1 interrupt service routine. To enable endpoint 1 interrupts, bit 4 of the Global Interrupt Enable Register (0x20) should be set to 1. This should be done after receiving a Set Configuration control request from the host that selects a configuration which enables this endpoint.

The USB Endpoint 1 TX Configuration register (0x11) controls data transmission from endpoint 1. To enable the endpoint, bit 4 should be set to 1. Otherwise, all IN token packets sent to the device will be ignored. If bit 7 is set to 0, then IN packets will be NAK'd. If bit 7 is set to 1, then data bytes will be sent from the endpoint 1 FIFO. Bits[3:0] set the number of bytes to send, and bit 6 sets the data toggle setting (DATA0/1). After bit 7 is set to 1, the bit will be cleared once the data is acknowledged by the host. The endpoint 1 ISR can read this bit to determine if a data transmission is complete. Bit 5 is the Stall bit. Setting this bit to 1 will STALL any IN packets sent to the endpoint until the bit is cleared.

Endpoint Communication with the CY7C634/5xx

The CY7C634/5xx have 3 endpoints: endpoint 0 for control transfers and endpoints 1 and 2 for interrupt transfers. The USB Device Address Register (0x10) specifies the address that the device will respond to in bits[6:0]. Bit 7 of this register determines whether the address is enabled (set to 1) to respond to USB traffic or disabled (set to 0) to ignore traffic. Upon reset, this register is set to 0s so that the Serial Interface Engine (SIE) is disabled and the address is set to 0. Bit 7 should be set to one after receiving a USB Bus Reset (the USB Bus Reset bit, bit 5, of the Processor Status and Control Register, 0xFF, will be set to 1) to enable the default address 0. Please note that, unlike the CY7C630/1/2xx, the USB Bus Reset is not a true reset (Power-on and Watchdog resets are true resets). Instead, the USB Bus Reset is an interrupt to the CPU. The USB Bus Reset interrupt can be enabled by setting bit 0 of the Global Interrupt Enable Register (0x20) to 1. Interrupt vector 1 (ROM address 0002h) should contain a JMP USB_RESET instruction, where USB_RESET is the beginning address of the USB Bus Reset interrupt service routine.

During enumeration, the USB host will send a Set Address control request with a new device address. This new address should be stored in register 0x10 once the No Data Control transfer completes its Status Stage.

Each endpoint has a Mode and Counter register associated with it. The Mode register determines the response of the endpoint to specific packet types according to the mode written to the bits [3:0], as well as specifying the specific PIDs received in bits [7:4]. The mode encoding and responses are defined in section 16 of the CY7C634/5xx datasheet. The Counter register specifies the data toggle setting (bit 7), validity of received data for SETUP and OUT tokens (bit 6), and the number of bytes transmitted from or received in the endpoint FIFO (bits[3:0]).

Endpoint 0

The endpoint 0 FIFO is located from RAM address F8h to FFh. Endpoint 0 can generate interrupts to the CPU after data reception to or transmission from the endpoint 0 FIFO. Interrupt vector 4 (ROM address 0008h) should contain a JMP EP0_ISR instruction, where EP0_ISR is the beginning address of the endpoint 0 interrupt service routine. To enable endpoint 0 interrupts, bit 0 of the USB Endpoint Interrupt Enable Register (0x21) should be set to 1. This is normally done in the USB Bus Reset ISR after a USB Bus Reset occurs.

The USB Endpoint 0 Mode register is 0x12, and the Endpoint 0 Counter register is 0x11. The SIE can lock out writes to these registers by the firmware while it is updating their contents. To unlock the registers, the firmware must first read the register:

```
    IORD 11h
```

or

```
    IORD 12h
```

Any write to these registers by the firmware should be done in a loop where read and a compare follows the write to ensure that the data was written to the register. For instance, to write the value 'mode' to the endpoint 0 Mode register:

```
IORD 12h ; unlock the EP0 counter register
loop:
    MOV A, mode
    IOWR 12h
    IORD 12h
    CMP A, mode
    JNZ loop
```

Endpoints 1 and 2

The endpoint 1 FIFO is located from RAM address F0h to F7h, and the endpoint 2 FIFO is located from E8h to EFh. Both of these endpoints, unlike the CY7C630/1/2xx, can be set up as either IN or OUT interrupt endpoints (however, the USB Specification 1.0 only defines an IN interrupt endpoint). Interrupts can be generated to the CPU when data is received in or transmitted from the endpoint FIFO. Interrupt vector 5 (ROM address 000Ah) should contain a JMP EP1_ISR instruction, where EP1_ISR is the beginning address of the endpoint 1 interrupt service routine. Interrupt vector 6 (ROM address 000Ch) should contain a JMP EP2_ISR instruction, where EP2_ISR is the beginning address of the endpoint 2 interrupt service routine. To enable endpoint 1 or 2 interrupts, bit 1 or 2 of the USB Endpoint Interrupt Enable Register (0x21) should be set to 1. This should be done after receiving a Set Configuration control request from the host that selects a configuration which enables either of these endpoints. The USB Endpoint 1 Mode register is 0x14, and the Endpoint 1 Counter register is 0x13. The USB Endpoint 2 Mode register is 0x16, and the Endpoint 2 Counter register is 0x15. There is no write lock problem with these registers (as with endpoint 0).

USB Communication Routines for CY7C630/1/2xx

The following firmware routines describe how to use the endpoints in the CY7C630/1/2xx to communicate with the USB host. The routines are:

- Endpoint 0 ISR
- Control Read
- Control Write
- No Data Control
- Endpoint 1 ISR
- Send EP1 Data

Endpoint 0 ISR

```
;****************************************
; Interrupt handler: endpoint_zero
; Purpose: This interrupt routine handles the
; specially reserved control endpoint 0 and
; parses SETUP packets by calling the
; ParseSetup routine (not shown). The
; ParseSetup routine calls the Control_read,
; Control_write, or No_data_control routines
; to handle the subsequent IN or OUT packets
; appropriately.
;
; This routine is primarily responsible for
; enumeration and configuration of the
; hardware.
;****************************************
EP0_ISR:
    push A ; save accumulator on stack
    ; load RX status register into A
    iord USB_EP0_RX_Status ; (0x14)
    ; make sure SETUP packet received
    ; otherwise ignore interrupt
    and A, 01h
    jz ep0_continue

    ; load RX status register into A
    iord USB_EP0_TX_Config ; (0x10)
    ; make sure setup data is valid
    ; otherwise ignore interrupt
    and A, 10h
    jnz ep0_continue

    ; load RX status register into A
    iord USB_EP0_TX_Config ; (0x10)
    ; make sure we received 10 bytes
    ; (8 + 2 CRC), otherwise ignore
    ; interrupt
    and A, F0h
    cmp A, A0h
    jnz ep0_continue

    ; disable endpoint zero interrupts
    ; so that subsequent packets don't
    ; cause interrupt nesting
    mov A,[interrupt_mask]
    and A, 0F7h
    mov [interrupt_mask], A
    iowr Global_Interrupt ; (0x20)

    ; execute the ParseSetup routine to
    ; determine which control request was
    ; sent and handle it appropriately
    call ParseSetup

    ; re-enable endpoint zero interrupts
    ; in the interrupt_mask variable
    mov A, [interrupt_mask]
    or A, 08h
    mov [interrupt_mask], A
ep0_continue:
    ; enable interrupts, pop A, and exit
    mov A, [interrupt_mask]
    ipret Global_Interrupt
```

Control Read Routine

```
;****************************************
; Function: Control_read
; Purpose: Performs the control read
; operation as defined by the USB
; specification: SETUP-IN-IN-IN...OUT
; It is used to send descriptors to the USB
; host.
;
; data_start: must be set to the descriptor
; info as an offset from the beginning of the
; control_read_table (ROM table)
;
; data_count: must be set to the size of the
; descriptor
;
; premature_setup: set to 1 if a SETUP
; was received by the host during the
; time that we were in this routine.
;****************************************
Control_read:
    push A ; save A on stack
    push X ; save X on stack
    mov A, 00h ; clear data 0/1 bit
    ; save data toggle setting
    mov [endp0_data_toggle], A

control_read_data_stage:
    mov X, 00h
    mov A, 00h
    mov [loop_counter], A
    ;clear setup bit
    iowr USB_EP0_RX_Status ; (0x14)

    ; check setup bit
    iord USB_EP0_RX_Status
    and A, 01h
    ; if not cleared, another setup
    ; has arrived. Set premature flag and
    ; exit ISR
    jnz control_read_premature_setup
    ; set StatusOuts bit so that we STALL
    ; invalid Status OUTs
    mov A, 08h
    iowr USB_Status_Control ; (0x13)
    mov A, [data_count]
    cmp A, 00h
    jz control_read_status_stage

; loop to load data into the FIFO
FIFO_load_loop:
    mov A, [data_start]
    index control_read_table
    ; load FIFO buffer
    mov [X + endpoint_0], A
    inc [data_start]
    inc X
    inc [loop_counter]
    dec [data_count]
    ; exit if descriptor is done
    jz FIFO_load_done
    ; or exit if 8 bytes sent
    mov A, [loop_counter]
    cmp A, 08h
    jnz FIFO_load_loop

FIFO_load_done:
    ; check setup bit
    iord USB_EP0_RX_Status
    and A, 01h
    ; if not cleared, another setup
    ; has arrived. Set flag and exit ISR
    jnz control_read_premature_setup
    ; perform data toggle
    mov A, [endp0_data_toggle]
    xor A, 40h
    mov [endp0_data_toggle], A
    ; enable data response to IN packets
    or A, 80h
    ; write number of bytes to send
    or A, [loop_counter]
    iowr USB_EP0_TX_Config ; (0x10)

; wait for the data to be transferred
wait_control_read:
    iord USB_EP0_TX_Config
    and A, 80h
    jz control_read_data_stage
    ; check if OUT or SETUP was sent by
    ; the host
    iord USB_EP0_RX_Status
    and A, 01h
```

```
    ; set flag and exit if SETUP
    jnz control_read_premature_setup
    iord USB_EP0_RX_Status
    and A, 02h
    ; exit if OUT
    jnz done_control_read
    jmp wait_control_read

; keep looping until we get the Status OUT
; or another SETUP
control_read_status_stage:
    ; check if OUT or SETUP was sent by
    ; the host
    iord USB_EP0_RX_Status
    and A, 01h
    ; set flag and exit if SETUP
    jnz control_read_premature_setup
    iord USB_EP0_RX_Status
    and A, 02h
    ; exit if OUT
    jnz done_control_read
    jmp control_read_status_stage


control_read_premature_setup:
    mov A, 1 ; set flag to true
    mov [premature_setup], A

done_control_read:
    pop X ; restore X from stack
    pop A ; restore A from stack
    ret ; exit subroutine
```

Control Write Routine

```
;****************************************
; Function: Control_write
; Purpose: Performs the control write
; operation as defined by the USB
; specification: SETUP-OUT-IN
; This routine is set up to receive
; 1 data packet (8 bytes) only (i.e.,
; it does not handle multiple OUTs).
; The data_ok flag is set to 1 (true) if the
; received data is valid, otherwise it is
; set to 0 (false).
; This routine waits until the status
; stage IN packet is sent by the host,
; but will NAK it. The caller of this
; routine should call the No_data_control
; routine after it has processed
; the incoming data to complete the
; handshake.
;
; premature_setup: set to 1 if a SETUP
; was received by the host during the
; time that we were in this routine.
;****************************************
Control_write:
    push A ; save A on stack
    ;clear setup bit
    iowr USB_EP0_RX_Status ; (0x14)
    mov A, 0 ; clear data_ok flag
    mov [data_ok], A
    ; check setup bit
    iord USB_EP0_RX_Status
    and A, 01h
    ; if not cleared, another setup
    ; has arrived. Set flag and exit ISR
    jnz control_write_premature_setup

    ; enable OUT data to be written to
    ; FIFO
    mov A, 10h ; set Enable Outs bit to 1
    iord USB_Status_Control

    ; wait for data
; keep looping until we get the OUT
wait_control_write:
    iord USB_EP0_RX_Status
    and A, 01h
    ; set flag and exit ISR if rec'd SETUP
    jnz control_write_premature_setup
    iord USB_EP0_RX_Status
    and A, 02h ; check for OUT packet
    jz wait_control_write
check_out_data:
    iord USB_EP0_RX_Status
    ; make sure we received 10 bytes
    and A, F0h
    cmp A, A0h
    jnz send_stall
    ; clear RX register
```

```
    iowr USB_EP0_RX_status
    iord USB_EP0_TX_Config
    ; make sure data is valid
    and A, 10h
    ; ready to move into status stage
    jz control_write_status_stage
    ; keep looping
    jmp wait_control_write

; wait until Status Stage IN is received
control_write_status_stage:
    iord USB_EP0_RX_Status
    and A, 01h
    ; set flag and exit if SETUP received
    jnz control_write_premature_setup
    iord USB_EP0_RX_Status
    ; if OUT received, then grab new data
    and A, 02h
    jnz check_out_data ; go check OUT data
    iord USB_EP0_RX_Status
    and A, 04h ; check IN bit
    jz control_write_status_stage
    ; host is sending Status IN, but
    ; don't handshake now - let
    ; No_data_control routine do this
    mov A, 0
    ; stop responding to OUTs
    iowr USB_Status_Control
    ; set data ok flag to true so that
    ; calling routine can use the data
    mov A, 1
    mov [data_ok], A
    jmp done_control_write

send_stall: ; STALL IN / OUT
    mov A, 0 ; clear reg. 0x13
    iowr USB_Status_Control
    mov A, 20h ; set STALL bit in reg. 0x10
    iowr USB_EP0_TX_Config
    ; set data ok flag to false so that
    ; calling routine doesn't use the
    ; bad data
    mov A, 0
    mov [data_ok], A
    jmp done_control_write

control_write_premature_setup:
    mov A, 1 ; set flag to true
    mov [premature_setup], A

; we're done
done_control_write:
    pop A ; restore A from stack
    ret ; exit subroutine
```

No Data Control Routine

```
;****************************************
; Function: No_data_control
; Purpose: Performs the no data control
; operation as defined by the USB
; specification: SETUP-IN
; This routine completes the handshake for
; the Status Stage IN by sending a DATA1
; packet with 0 bytes of data.
;
; premature_setup: set to 1 if a SETUP
; was received by the host during the
; time that we were in this routine.
;****************************************
No_data_control:
    ; clear the RX status register
    iowr USB_EP0_RX_Status
    ; set up TX register for DATA1 PID and
    ;0 byte transfer
    mov A, C0h
    iowr USB_EP0_TX_Config

wait_nodata_sent:
    iord USB_EP0_RX_Status
    and A, 01h
    ; set flag and exit if we get SETUPs
    jnz no_data_control_premature_setup
    iord USB_EP0_TX_Config
    ; wait for the data to be transferred
    and A, 80h
    jnz wait_nodata_sent
    jmp done_no_data_control

no_data_control_premature_setup:
    mov A, 1 ; set flag to true
    mov [premature_setup], A
```

```
done_no_data_control:
    ret ; return to caller
```

Endpoint 1 ISR

```
;****************************************
; Interrupt handler: endpoint_one
; Purpose: This interrupt routine handles the
; interrupt endpoint 1. This interrupt
; happens every time a host sends an
; IN to endpoint 1. The data to send (NAK or
; data bytes) is already loaded, so this
; routine just prepares for the next packet
;**************************************
EP1_ISR:
    push A ; save accumulator on stack
    push X ; save X on stack

    iord USB_EP1_TX_Config
    ; return NAK when data is not ready
    and A, 7Fh
    xor A, 40h; flip data 0/1 bit
    iowr USB_EP1_TX_Config

    ; clear endpoint 1 FIFO
    mov X, 8
    mov A, 0
ep1_fifo_clear:
    mov [X + endpoint_1 - 1], A
    dec X
    jnz ep1_fifo_clear

    ; return from interrupt
    pop X
    mov A, [interrupt_mask]
    ipret Global_Interrupt
```

Send Endpoint 1 Data Routine

```
;**************************************
; Function: Send_EP1_Data
; Purpose: Sends 8 bytes from ep 1 FIFO
; to the host over the interrupt pipe.
; (Data must be loaded into FIFO before
; calling this routine.)
;**************************************
Send_EP1_Data:
    push A
    iord USB_EP1_TX_Config
    ; keep the data 0/1 bit
    and A, DataToggle
    or A, 98h ; enable 8 byte transmission
    iowr USB_EP1_TX_Config

    ; wait for host to send ACK
    ; while we wait, the EP1 interrupt
    ; will occur when ACK is received
wait_ep1_ack: ; wait for host to ACK
    iord USB_EP1_TX_Config
    and A, 80h
    jnz wait_ep1_ack

    pop A
    ret ; exit subroutine
```

USB Communication Routines for CY7C634/5xx

The following firmware routines describe how to use the endpoints in the CY7C634/5xx to communicate with the USB host. The routines are:

- USB Bus Reset ISR
- Endpoint 0 ISR
- Control Read
- Control Write
- No Data Control
- Endpoint 1 ISR
- Send Endpoint 1 Data

Additionally, there are some auxiliary functions for convenience:

- Set EP0 Mode
- SendStall

USB Bus Reset ISR

```
;**************************************
; Interrupt handler: USB Bus Reset
; Purpose: This interrupt routine handles the
; USB bus reset event.
;**************************************
USB_Bus_Reset_ISR:
    push A ; save accumulator on stack
    ; clear the Bus Reset bit in the
    ; Status & Control register
    iord Status_Control ; (0xFF)
    and A, DFh
    iowr Status_Control

    ; enable one msec timer interrupt (bit
```

```
    ; 2) and bus reset interrupt (bit 0)
    mov A, 05h
    iowr Global_Interrupt ; (0x20)

    ; enable endpoint zero interrupt only
    mov A, 01h
    iowr Endpoint_Interrupt ; (0x21)

    ; unlock EP0 Mode register
    iord EP_A0_Mode ; (0x12)

    ; set EP0 mode to Ignore IN/OUT (0100)
    ; but accept SETUPs
    mov A, 04h
    iowr EP_A0_Mode

    ; enable device address 0
    mov A, 80h
    iowr USB_Device_Address ;(0x10)

    mov A, 0 ; disable endpoint one
    iowr EP_A1_Mode ; (0x14)

    pop A ; restore accumulator from stack
    reti ; return from interrupt
```

Endpoint 0 ISR

```
;****************************************
; Interrupt handler: endpoint_zero
; Purpose: This interrupt routine handles the
; specially reserved control endpoint 0 and
; parses SETUP packets by calling the
; ParseSetup routine (not shown). The
; ParseSetup routine calls the Control_read,
; Control_write, or No_data_control routines
; to handle the subsequent IN or OUT packets
; appropriately.
;
; This routine is primarily responsible for
; enumeration and configuration of the
; hardware.
;****************************************
USB_EP0_ISR:
    push A ; save accumulator on stack
    iord EP_A0_Mode ; (0x12)
    ; do nothing unless we received
    ; a SETUP token packet
    and A, 80h
    jz done_EP0
    ; check if data is valid
    iord EP_A0_Counter ; (0x11)
    and A, 40h
    ; otherwise STALL IN/OUT
    jz stall_in_out
    ; make sure we received 10 data bytes
    ; (8 data bytes + 2 CRC bytes)
    iord EP_A0_Counter
    and A, 0Fh
    cmp A, 0Ah
    ; otherwise STALL IN/OUT
    jnz stall_in_out
    ; check if data toggle is 0
    iord EP_A0_Counter
    and A, 80h
    ; otherwise STALL IN/OUT
    jnz stall_in_out

    ; unlock the mode register
    iord EP_A0_Mode
ep0_clear_pid_bits:
    ; write same mode (NAK IN/OUT, 0001)
    ; to clear PID bits
    mov A, 01h
    iowr EP_A0_Mode
    iord EP_A0_Mode
    ; make sure the write wasn't locked
    ; out
    cmp A, 01h
    jnz ep0_clear_pid_bits

    ; disable endpoint zero interrupt
    iord Endpoint_Interrupt
    and A, FEh
    iowr Endpoint_Interrupt
    ei ; enable other interrupts
    call ParseSetup ; parse SETUP packet

    ; did we get a premature SETUP?
    mov A, [premature_setup]
    cmp A, 0
    ; no, then enable EP0 int and return
    jz enable_ep0_interrupt
```

```
    mov A, 0
    ; yes, then clear flag and return
    ; (we'll get another interrupt for the
    ; new SETUP)
    mov [premature_setup], A
    jmp enable_ep0_interrupt

    ; accept SETUP, stall IN/OUT
stall_in_out:
    call SendStall
    jmp done_EP0

enable_ep0_interrupt:
    di ; disable interrupts
    ; enable endpoint zero interrupt
    iord Endpoint_Interrupt
    or A, 01h
    iowr Endpoint_Interrupt

done_EP0:
    pop A ; restore accumulator from stack
    ; return from ISR (interrupts will be
    ; enabled)
    reti
```

Set Endpoint 0 Mode Routine

```
;****************************************
; Function: set_ep0_mode
; Purpose: Sets the endpoint 0 mode
; register (0x12) to the mode stored
; in the A register. This routine performs
; the unlocking of the register to ensure
; that the write to the register was
; successful.
;
; premature_setup is set to 1 if a SETUP
; was received by the host during the
; time that we were in this routine.
;****************************************
set_ep0_mode:
    mov [ep0mode], A
set_ep0_unlock:
    mov A, [ep0mode]
    iowr EP_A0_Mode
    iord EP_A0_Mode
    ; confirm write was not locked
    cmp A, [ep0mode]
    jnz ep0_mode_check_setup
    jmp done_set_ep0_mode

ep0_mode_check_setup:
    ; did we receive a premature SETUP?
    and A, 80h
    ; no, then try to set mode again
    jz set_ep0_unlock
    ; yes, then set flag and return
    mov A, 1
    mov [premature_setup], A

done_set_ep0_mode:
    ret
```

SendStall Routine

```
;****************************************
; Function: SendStall
; Purpose: This routine stores the
; Stall IN OUT mode (0011) into A and
; calls set_ep0_mode to store the value
; in the ep0 mode register. This will
; cause ep 0 to send STALLs to any IN or
; OUT packets received.
;****************************************
SendStall:
    push A
    ; set EP0 mode to Stall IN/OUT (0011)
    mov A, 03h
    call set_ep0_mode
    pop A
    ret
```

Control Read Routine

```
;****************************************
; Function: Control_read
; Purpose: Performs the control read
; operation as defined by the USB
; specification: SETUP-IN-IN-IN...OUT
; It is used to send descriptors to the USB
; host.
;
; data_start: must be set to the descriptor
; info as an offset from the beginning of the
; control_read_table (ROM table)
```

```
;
; data_count: must be set to the size of the
; descriptor
;
; premature_setup: set to 1 if a SETUP
; was received by the host during the
; time that we were in this routine.
;****************************************
Control_read:
    push A ; save A on stack
    ; copy first eight bytes to FIFO
    call SendBuffer
    ; return if we received a premature
    ; SETUP
    mov A, [premature_setup]
    cmp A, 0
    jnz done_control_read

    ; unlock Counter register
    iord EP_A0_Counter
    ; set data 0/1 bit
    mov A, 80h
    ; write correct byte count
    or A, [byte_count]
    mov [temp], A
control_read_unlock1:
    mov A, [temp]
    iowr EP_A0_Counter
    ; confirm write was unlocked
    iord EP_A0_Counter
    cmp A, [temp]
    jz control_read_data_stage
    ; if we were locked out, check for
    ; premature SETUP
    iord EP_A0_Mode
    and A, 80h
    jz control_read_unlock1
    ; we received premature SETUP
    mov A, 1
    ; set flag, and return
    mov [premature_setup], A
    jmp done_control_read

control_read_data_stage:
    ; accept IN, SETUP, and Status OUT
    ; (ACK IN - STATUS OUT mode, 1111)
    mov A, 0Fh
    call set_ep0_mode

    ; return if we received a premature
    ; SETUP
    mov A, [premature_setup]
    cmp A, 0
    jnz done_control_read

; wait for the data to be sent
wait_control_read:
    iord EP_A0_Mode ; read mode register
    ; wait for one of the PID bits to be
    ; set
    and A, F0h
    jz wait_control_read

check_PID_received:
    ; check if Status OUT packet received
    iord EP_A0_Mode
    and A, 20h
    jnz control_read_OUT_received
    ; check if we received a premature
    ; SETUP
    iord EP_A0_Mode
    and A, 80h
    ; no, then IN was received
    jz control_read_IN_received
    ; yes, then set flag and return
    mov A, 1
    mov [premature_setup], A
    jmp done_control_read

control_read_IN_received:
    ; make sure ACK was received
    iord EP_A0_Mode
    and A, 10h
    jz wait_control_read
    ; have we sent all the data?
    mov A, [data_count]
    cmp A, 0
    ; no, then send the next data packet
    jnz start_next_transfer

    ; we've sent all the data so host
    ; should send us Status OUT
```

```
    ; accept SETUP/OUT, stall IN
    ; (Status OUT only mode, 0010)
    mov A, 02h
    call set_ep0_mode

    ; return if we received a premature
    ; SETUP
    mov A, [premature_setup]
    cmp A, 0
    jnz done_control_read

    ; wait for the next packet
    jmp wait_control_read

; start the next transfer
start_next_transfer:
    call SendBuffer ; load the FIFO

    ; return if we received a premature
    ; SETUP
    mov A, [premature_setup]
    cmp A, 0
    jnz done_control_read

    ; unlock Counter register
    iord EP_A0_Counter
    and A, 80h ; keep data 0/1 bit
    ; toggle the data 0/1 bit
    xor A, 80h
    ; store the correct byte count
    or A, [byte_count]
    mov [temp], A
control_read_unlock2:
    mov A, [temp]
    ; write the Counter register
    iowr EP_A0_Counter
    ; confirm the write was not locked
    iord EP_A0_Counter
    cmp A, [temp]
    jz control_read_data_stage
    ; if we were locked out, check for
    ; premature SETUP
    iord EP_A0_Mode
    and A, 80h
    jz control_read_unlock2
    ; we received premature SETUP
    ; set flag, and return
    mov A, 1
    mov [premature_setup], A
    jmp done_control_read

; Status OUT received, so set EP0 mode to
; STATUSOUTONLY mode (0010)
control_read_OUT_received:
    ; accept SETUP/OUT, stall IN
    mov A, 02h
    call set_ep0_mode

done_control_read:
    pop A ; restore A
    ret ; return to caller
```

SendBuffer Routine

```
;****************************************
; Function:  SendBuffer
; Purpose: This routine is called by the
; Control_read routine to load the EP0
; FIFO with 8 bytes of data from the
; ROM table. It uses data_count and
; data_start as defined in Control_read.
;
; premature_setup: set to 1 if a SETUP
; was received by the host during the
; time that we were in this routine.
;****************************************
SendBuffer:
    push A ; save A on stack
    ; clear the PID bit in Mode
    ; accept SETUP, NAK IN/OUT (mode 0001)
    mov A, 01h
    call set_ep0_mode

    ; return if we received a premature
    ; SETUP
    mov A, [premature_setup]
    cmp A, 0
    jnz done_sendbuffer

    push X ; save X on stack
    mov A, 0 ; load count into A
sendloop:
    swap A,X ; move count into X
```

```
    ; load ROM index into A
    mov A, [data_start]
    ; get the ROM byte
    index control_read_table
    ; save the byte in FIFO
    mov [X + endpoint_0], A
    ; increment ROM index
    inc [data_start]
    inc X ; increment the count in X
    swap A,X ; move count to A
    cmp A,8 ; compare count to 8
    jnz sendloop ; keep copying

    ; default to 8-bytes
    mov [byte_count], A
    ; update byte counter
    mov A, [data_count]
    sub A, 8 ; subtract 8-bytes sent
    jnc doneSend
    ; update the byte count in memory
    mov A, [data_count]
    mov [byte_count], A
    mov A,0 ; done

doneSend:
    ; update byte counter
    mov [data_count], A
    pop X ; restore X from stack

done_sendbuffer:
    pop A ; restore A
    ret ; return to caller
```

Control Write Routine

```
;****************************************
; Function:  Control_write
; Purpose: Performs the control write
; operation as defined by the USB
; specification: SETUP-OUT-IN
; This routine is set up to receive
; 1 data packet (8 bytes) only (i.e.,
; it does not handle multiple OUTs).
; The data_ok flag is set to 1 (true) if the
; received data is valid, otherwise it is
; set to 0 (false).
; This routine waits until the status
; stage IN packet is sent by the host,
; but will NAK it. The caller of this
; routine should call the No_data_control
; routine after it has processed
; the incoming data to complete the
; handshake.
;
; premature_setup: set to 1 if a SETUP
; was received by the host during the
; time that we were in this routine.
;****************************************
Control_write:
    push A ; save A on stack
    mov A, 0 ; clear data_ok flag
    mov [data_ok], A
    ; accept OUT, enable TX0 IN
    ; ACK OUT STATUS IN mode (1011)
    mov A, 0Bh
    call set_ep0_mode

    ; return if we received a premature
    ; SETUP
    mov A, [premature_setup]
    cmp A, 0
    jnz done_control_write

; wait until we get the OUT byte
wait_control_write_OUT:
    iord EP_A0_Mode ; read mode register
    ; did we get the OUT packet?
    and A, 20h
    jz control_write_check_OUT
    iord EP_A0_Mode ; read mode register
    ; did we receive premature SETUP?
    and A, 80h
    jz wait_control_write_OUT
    ; yes, then set flag and return
    mov A, 1
    mov [premature_setup], A
    jmp done_control_write

; we received the OUT byte, make sure the data
; is valid
control_write_check_OUT:
    ; read Mode register
    iord EP_A0_Mode
```

```
    and A, 10h; did we ACK the OUT data?
    ; since we ignored the data, the host
    ; will try to send it again
    ; jump back to the beginning
    jz Control_write
    ; read counter register
    iord EP_A0_Counter
    and A, 0Fh
    ; is count 10 (8 data + 2 CRC)?
    cmp A, 0Ah
    ; we ACK'd the data (so data is valid)
    ; but we did not receive the right #
    ; of bytes - send STALL to any further
    ; INs and OUTs
    jnz control_write_send_stall

    ; accept SETUP, NAK IN/OUT
    ; NAK IN OUT mode (0001)
    mov A, 01h
    call set_ep0_mode

    ; return if we received a premature
    ; SETUP
    mov A, [premature_setup]
    cmp A, 0
    jnz done_control_write

; wait until host sends us Status IN
control_write_wait_IN:
    iord EP_A0_Mode ; read mode register
    and A, 20h ; did we get an OUT?
    ; if so, jump back to beginning to get
    new OUT byte
    jnz Control_write

    iord EP_A0_Mode ; read mode register
    and A, 40h ;did we get the Status IN?
    ; if so, return so that we can process
    ; the OUT byte before No_data_control
    ; responds to the Status IN
    jnz control_write_status_in

    iord EP_A0_Mode ; read mode register
    ; did we get a premature SETUP?
    and A, 80h
    ; no, then wait for Status IN
    jz control_write_wait_IN
    mov A, 1 ; yes, set flag and return
    mov [premature_setup], A
    jmp done_control_write

control_write_send_stall:
    ; set data ok flag to false so that
    ; calling routine does not use the
    ; bad data
    mov A, 0
    mov [data_ok], A
    call SendStall
    jmp done_control_write

control_write_status_in:
    ; set data ok flag to true so that
    ; calling routine knows data is good
    mov A, 1
    mov [data_ok], A

done_control_write:
    pop A
    ret ; return
```

No Data Control Routine

```
;****************************************
; Function:  No_data_control
; Purpose: Performs the no data control
; operation as defined by the USB
; specification: SETUP-IN
; This routine completes the handshake for
; the Status Stage IN by sending a DATA1
; packet with 0 bytes of data.
;
; premature_setup: set to 1 if a SETUP
; was received by the host during the
; time that we were in this routine.
;****************************************
No_data_control:
    push A
    ; put EP0 in Status In Only mode
    ; (0110) to transmit zero-length
    ; data packet to Status IN
    mov A, 06h
    call set_ep0_mode
```

```
    ; return if we received a premature
    ; SETUP
    mov A, [premature_setup]
    cmp A, 0
    jnz done_nodata_control

; wait for the zero-length transfer to complete
wait_nodata_sent:
    iord EP_A0_Mode ; read mode register
    ; did we receive premature SETUP?
    and A, 80h
    jz check_nodata_ack
    ; yes, then set flag and return
    mov A, 1
    mov [premature_setup], A
    jmp done_nodata_control

check_nodata_ack:
    iord EP_A0_Mode ; read mode register
    and A, 10h ; wait for ACK bit high
    jz wait_nodata_sent

    ; set mode to Ignore IN/Out (0100)
    mov A, 04h ;still accept SETUPs
    call set_ep0_mode

done_nodata_control:
    pop A
    ret ; return
```

Endpoint 1 ISR

```
;****************************************
; Interrupt handler: endpoint_one
; Purpose: This interrupt routine handles the
; interrupt endpoint 1.  This interrupt
; happens every time a host sends an
; IN to endpoint 1.  The data to send (NAK or
; data bytes) is already loaded, so this
; routine just prepares for the next packet.
;
; A similar interrupt handler can be
; written for endpoint 2 (just use
; the mode register 0x16 and counter
; register 0x15).
;****************************************
USB_EP1_ISR:
    push A ; save accumulator on stack
    ; test whether ACK bit is set
    ; to know that the last data
    ; transmission was successful
    iord EP_A1_Mode
    and A, 10h
    ; do nothing if we don't have an ACK
    ; bit
    jz doneEP1

    iord EP_A1_Counter
    ; flip data 0/1 bit after
    ; a successful data transfer
    xor A, 80h
    iowr EP_A1_Counter

    push X
    mov X,7 ; load loop counter
    mov A,0

copyloop:
    ; clear the endpoint buffer
    mov [X + endpoint_1], A
    dec X
    jnc copyloop
    pop X

doneEP1:
    pop A ; restore accumulator from stack
    reti ; return from interrupt
```

Send EP1 Data Routine

```
;****************************************
; Function:  Send_EP1_Data
; Purpose: Sends 8 bytes from ep 1 FIFO
; to the host over the interrupt pipe.
; (Data must be loaded into FIFO before
; calling this routine.)
;
; A similar interrupt handler can be
; written for endpoint 2 (just use
; the mode register 0x16 and counter
; register 0x15).
;;****************************************
Send_EP1_Data:
```

```
    push A ; save accumulator on stack
    iord EP_A1_Counter ; (0x13)
    and A, 80h ; keep data toggle setting
    or A, 8h ; packet size
    iowr EP_A1_Counter

    ; set EP1 mode to ACKIN IN (1101)
    ; to enable packet transmission when
    ; an IN is received from the host
    mov A, 0Dh
    iowr EP_A1_Mode ; (0x14)

wait_data_sent:
    ; wait for data acknowledge (ACK bit
    ; set) before loading register again
    iord EP_A1_Mode
    and A, 10h
    jz wait_data_sent

    ; set endpoint 1 mode to NAKIN (1100)
    ; also clears ACK bit
    mov A, 0Ch
    iowr EP_A1_Mode

    pop A ; restore accumulator
    ret ; return
```

Conclusion

The CY7C63xxx family of microcontrollers provide a flexible, low-cost solution for the development of low-speed USB microcontrollers. With the availability of good development tools, such as the CYASM assembler, the CY3650/3651 Development Kits, and the information contained in this document, the task of firmware development is greatly simplified.