A Beginners Guide to Programming the CY7C630/10x using Cyasm

Introduction

The purpose of this application note is to help programmers begin programming the CY7C630/1xx microcontroller. This application note will show some very basic implementations that are widely used in many applications. It is the purpose here to make programming the CY7C630/10x microcontroller a little bit easier. Sometimes the things that are done most often aren't the most straight forward problems to solve; and for beginners a simple overview is more helpful than a full specification. This application note should help these problems become as straight forward as possible and help to save time in the development process.

Introduction to the CY7C630/1xx

The CY7C630/1xx is a microcontroller with up to 8 KB of program memory and up to 16 general purpose I/O pins implemented as two byte wide ports. This microcontroller is well-suited for low-speed USB peripheral devices such as mice and game controllers. This microcontroller has a number of registers that are used for set-up, control, and data transfer. It is the intention of this application note to show some examples of firmware that are commonly used in final implementations, to explain what registers are effected, and what registers must be configured to complete these tasks.

Examples Included in this Application Note

1. Watchdog Reset
2. GPIO Ports
3. Resistive
4. Current Sink
5. Interrupts
6. GPIO Interrupt setup
7. GPIO Interrupt Detection
8. Delay Timers
9. USB
10. Endpoint Receive
11. Endpoint Transmit
12. Bus Reset
13. CEXT interrupt
14. Send 0 Byte Data Packet

Watchdog Reset

The Watchdog Timer on these devices will reset the microcontroller every 8 ms if this timer is not reset by firmware. Resetting the Watchdog Timer is done by writing any value to the I/O address of the Watchdog Register (0x21). The IOWR function supplied by the assembler is the only function that can be used to write to any I/O register. This function takes the value that is in the accumulator and writes that value to the specified register. Since any write to the Watchdog Register will reset it, it is only necessary to perform the following instruction to reset the watchdog timer:

```
IOWR Watchdog
```

Assuming that Watchdog is defined in the beginning of the firmware using the following assembly instruction:

```
Watchdog:equ21h
```

21h is defined in the device specification as the I/O address of the Watchdog register.

GPIO Ports

The CY7C6300x parts have 12 I/O pins and the CY7C6310x parts have 16 I/O pins. These I/O pins are called general purpose because they are not assigned to a specific I/O interface and are under the control of the firmware. These pins can be configured into resistive and current sink modes and can also act as a configurable interrupt source. These are described in the following four sections.

I/O Port Set-up: Resistive

The I/O ports can be configured as a resistive pull up by enabling an internal pull-up resistor available on each pin. This mode can be used as a very weak '1' being read at the output of the pin. When in this state it is very easy to pull a General Purpose I/O pin LOW with an external sink.

Setting up a port for resistive mode consists of writing a '1' into the Data register bit associated with that pin and a '0' into the pull-up register bit associated as well. This is done with the following piece of firmware (assuming Port 0/Pin 0):

```
Mov A, 01h
IOWR Port0_Data
Mov a, 00h
IOWR Port0_Pullup
```

This assumes Port0_Data and Port0_Pullup constants are assigned in the same way that the Watchdog Register was assigned in section 1.0.

I/O Port Set-up: Current Sink

When it is necessary to sink current using the GPIO pins, this is the state that the I/O pin will need to be set-up in. Port 0 on these devices is a low current port (up to 1.5 mA) and Port 1 is a high current port (up to 24 mA). Low current port pins are capable of driving photo-transistors and high current ports are capable of driving LED's. To implement this configuration on a single I/O pin change the values stored in the two registers shown in the example with the values that are shown in the following firmware example.

```
Mov A, 00h
IOWR Port0_Data
Mov A, 00h   ;Or 01h
IOWR Port0_Pullup
```

Interrupts

For these devices all interrupts that are available are defined in the microcontroller data sheet. The following piece of firmware creates a interrupt vector table that is used whenever any interrupt occurs. Each interrupt created is generated by hardware and the microcontroller automatically jumps to the corresponding subroutine. The names of these subroutines can be any name the programmer would like to give but the jump table is specified as shown in the following code. It is necessary that the jump to the interrupt service routine be correct in the jump table because this is a hardware level issue and cannot be changed by firmware. Where the firmware jumps to is up to the programmer, but when an interrupt occurs it is up to the hardware to decide which interrupt occurred and jump to the correct address of the interrupt service routine.

```
org 00h
jmp reset_int
jmp 128us_int
jmp 1ms_int
jmp EndPoint0_int
jmp EndPoint1_int
jmp reset_int
jmp GPIO_int
jmp Wakeup_int
```

When servicing an interrupt, the microcontroller disables all interrupts, clears the latch for the current interrupt, pushes the PC, CF, and ZF values on to the program stack, and then uses the CALL function to call the correct interrupt service routine. It is the responsibility of the firmware to re-enable interrupts, pop the values for PC, CF, and ZF off the stack and use the IPRET instruction to return from the interrupt. Once the above jump table is created, interrupts must be enabled using the Global Interrupt Enable Register (GIER, 0x20), and a corresponding interrupt service routine must be written for each enabled interrupt. An example of setting up the global interrupt enable register is shown in the following section.

GPIO Interrupt Set-up

Any one of the GPIO pins on these devices can be programmed as an external interrupt pin. Configuring a pin as an interrupt source is quite easy and will be shown here. The challenge becomes apparent when having more than one GPIO interrupt set-up on the system, and having to determine which GPIO caused the interrupt. However, once any one GPIO interrupt has occurred, no other GPIO interrupts may occur until the pin that activated the interrupt is returned to its inactive state. Detecting GPIO interrupts will be discussed in the following section. Interrupts on GPIO pins can be set-up as either rising edge or falling edge polarity. To set-up any port pin as an interrupt it is first necessary to decide whether the interrupt should be rising or falling edge. Rising edge is set by a '1' in the corresponding pull-up register and falling edge is set by a '0' in the corresponding pull-up register.

```
Mov A, 00h
IOWR Port0_Pullup
```

The above code will set-up all port0 pins as falling edge triggered interrupts. However it is next necessary to enable specific pins as interrupts.

```
Mov A, 01h
IOWR Port0_Interrupt_Enable
```

The above code enables Port 0 / Pin 0 as an interrupt. All other pins on Port 0 will not be allowed to cause an interrupt until they are set in the same manner. Multiple interrupts can be enabled at any time by writing a '1' to the corresponding bit in the interrupt enable register.

The final step in setting up an interrupt on a GPIO pin is to enable GPIO interrupts in general. This is done by enabling the GPIO interrupt bit in the global interrupt enable register, which can be done with the following assembly instructions.

```
IORD GIER;read Global interrupt register
OR A, 40h
IOWR GIER;write back to the register
```

The above firmware does not simply write a value of 40h or 01000000 to this register because that would change all other interrupts that were previously enabled to disabled. The above code only changes the bit corresponding to the GPIO interrupt enables and leaves all other enable bits alone. All three of the registers shown above must be set correctly for any GPIO interrupt to occur.

GPIO Interrupt Detection

The process of detecting which GPIO pin caused an interrupt can be a very simple or a somewhat difficult task depending on the number of GPIO interrupts that are enabled. This is because all GPIO interrupts cause a single interrupt in the interrupt jump vector table. Therefore it is the responsibility of the interrupt service routine to determine which interrupt occurred. If only one GPIO interrupt source is enabled it is obvious that this pin is responsible for the GPIO interrupt. However if two or more GPIO interrupts are enabled, some code must be written to accommodate this problem.

It is very important to repeat a previous statement that says “once any one GPIO interrupt has occurred no other GPIO interrupts will be allowed until that initial pin is reset to its inactive state.” This is important because it allows us to check the current state of all pins directly after the interrupt occurred and check these against known values to see if they were responsible for creating the interrupt.

Delay Timers

It is quite often necessary to implement some sort of time delay in firmware. This can be done in two ways. One is to use the interrupts caused by the system timer (i.e. the 1.024-ms interrupt and the 128 µs interrupt) and use the interrupt service routines for these interrupts to increment a counter and jump when the counter corresponds to the correct delay interval that is desired. The second way is to create a delay loop which will wait for a specified period of time.

```
Delay:
```

```
mov A, ffh
mov [Delay_time1], A
mov [Delay_time2], A
   loop1:
mov A, [Delay_time1]
cmp A, 00H
jz loop2
nop
dec [Delay_time1]
jmp loop1
   loop2:
mov A, FFh
mov [Delay_time1], A
mov A, [Delay_time2]
cmp A, 00h
jz return
dec [Delay_time2]
jmp loop1
   return:
ret
```

The above firmware shows an example of a delay that uses two delay loops for longer delay times. The delay of the above loop is ~170 ms. Other values of delay time can be assigned by changing the values moved into the temp variables Delay_time1 and Delay_time2 and adding NOP instructions into the loops. NOPs take four clock cycles and add this much delay every time the loop is run through. Loops can also be added or taken away for longer or shorter delays. Delay time can be found by looking at the following list file segment, which shows the number of cycles ([cycles]) to perform each operation (each cycle is 1/12 MHz = 83.33 ns).

```
0057             Delay:
0057 19 FF [04]      mov A, ffh
0059 31 52 [05]      mov [Delay_time1], A
005B 31 54 [05]      mov [Delay_time2], A
005D               loop1:
005D 1A 52 [06]      mov A, [Delay_time1]
005F 16 00 [04]      cmp A, 00H
0061 A0 77 [05]      jz loop2
0072 20    [04]      nop
0073 27 52 [07]      dec [Delay_time1]
0075 80 5D [05]      jmp loop1
0077               loop2:
0077 19 FF [04]      mov A, FFh
0079 31 52 [05]      mov [Delay_time1], A
007B 1A 54 [06]      mov A, [Delay_time2]
007D 16 00 [04]      cmp A, 00h
007F A0 85 [05]      jz return
0081 27 54 [07]      dec [Delay_time2]
0083 80 5D [05]      jmp loop1
0085               return:
0085 3F    [08]      ret
```

USB Functions

Since these parts are commonly used for USB-type applications this section will describe some very basic yet very fundamental pieces in firmware used when creating any USB application.

Endpoint Receive

Endpoint receives are one of the most important aspects of USB programming because all communication done through USB must use at least one endpoint. An endpoint can be described as the communications channel (or pipe) used in a USB system. Most of the USB data tasks in these microcontrollers is taken care of in hardware by the serial interface engine. However some very important set-up and control firmware is necessary to control the SIE. As shown in the Interrupts section, interrupts for each endpoint that will be used must be set-up in the interrupt jump table. Also the interrupt must be enabled in the global interrupt enable register.

CY7C630/10x microcontrollers offer a buffer size of 8 bytes for each of its two endpoints. It is necessary for this buffer to be empty before data is transmitted from the host computer because the data coming from the host computer will overwrite the buffer at that time.

The Serial Interface Engine (SIE) takes care of most of the communications tasks that are necessary for USB. Upon receiving information from the host computer, this data is stored in the endpoint buffer and an endpoint interrupt is generated. It is the responsibility of the firmware to deal with the information that is in the buffer after the interrupt occurs. This is done in the Interrupt Service Routine (ISR), as shown below. The following firmware uses examples of endpoint 0 since this is the endpoint that is most commonly used, because it is the control endpoint.

```
;***********************************
;End point 0 interrupt service Routine
;***********************************
EndPoint0_int:
Push A
iord USB_SCR;Read this to enable outputs or
A, 08h         ; ACK's get sent back Automat-
ically
iowr USB_SCR
iord EnPt0_RX_Status;read to check packet
type
and A, 07h
cmp A, 01h     ;Is it a Setup Packet
jz Setup_Packet_Recieved
```

```
cmp A, 02h      ;Is it an Out Packet
jz Out_Packet_Recieved
jmp In_Packet_Received  ;Must be an In Packet
```

The firmware shown above reads the information in the endpoint status register to see what type of packet was received from the host computer. This can be done by querying the 3 bits in that register corresponding to SETUP, IN and OUT packets. After the type of packet is known the corresponding jump routine is done as shown in the following firmware sample, which deals with set-up packets.

```
Setup_Packet_Recieved:
mov A, [EnPt0_Byte1];what kind of setup
packet
cmp A, 5;Set_address?
jz Set_Address_Loop
cmp A, 6;Get_Descriptor?
jz Send_Device_Descriptor
```

The above firmware next tries to decipher the SETUP packet to determine which type of SETUP packet was received. Each type of SETUP packet then has a subroutine defined for it. These will not be shown here because they can be different for each application. Definitions for these can be found in the USB Specification. Other types of token packets can be received. These are IN and OUT packets and example routines for each of these is shown below.

```
In_Packet_Recieved:
mov A, [Getting_Desc];Still more to read
from
cmp A, 01h
jnz done2
mov A, [Desc_Length]
cmp A, 00h
jz Send0ByteDataPacket
mov A, 00h;reset X to 00h
push A
pop X
      loop2:
push X
pop A
cmp A, 08h
jz done2
index device_desc_table
mov [X+Endpoint_0], A
inc X
dec [Desc_Length]
jmp loop2

      done2:
mov A, 08h;re-enable endpoint0 interrupt
ipret global_int


Out_Packet_Recieved:
mov A, 08h;re-enable endpoint0 interrupt
ipret global_int
```

Endpoint Transmit

To transmit data out of the device to the host there are a few things that must be done prior to the actual transmission. Again, here most of the actual transmission is taken care of by the USB SIE hardware inside of the microcontroller. It is only necessary for the programmer to move the data to the endpoint buffer and set the correct SIE configuration in the transmit configuration register, before the transaction is prompted by the host, and the SIE follows through with the transaction.

The firmware is responsible for moving the desired data into the eight-byte endpoint data buffer supplied in the device. After this is done the firmware is responsible for setting up the transfer by specifying the number of bytes to be transmitted, enabling the endpoint, and setting the correct data toggle bit corresponding to that piece of data. This is done as follows:

```
Mov a, Byte0;mov data to endpoint buffer
Mov EP0_Byte0, a
.... ;Total # of bytes to sendup to 8
mov a, total_bytes ;the number of bytes being
sent
or a, 80h;enable IN packets (device trans-
mit)
or a, Data01  ;use correct Data01 value for
next trans
iowr  ep0_tx_reg ; setup In Register
```

The code above must be modified to move the correct amount of data into the EP0 buffer and also must be modified to write the correct data toggle for the specific packet being sent. Remember that the buffer for each endpoint is a maximum of eight bytes so any data greater than eight bytes must be sent in multiple packets.

Suspend

It is a requirement of the USB specification that all USB devices support a low-power suspend mode. In low-power mode any device is required to draw no more than 500 µA when in suspend. A high-power device that is enabled as a remote wakeup device may draw a maximum current of 2.5 mA during suspend. The Cypress microcontrollers that are being explained here support this feature. To put a device into suspend, write a '1' to the suspend bit in the Status Control Register (SCR, 0xFF). Setting the Suspend bit stops the oscillator and the interrupt timers, and powers down the microcontroller. The only components of the device that are not shut down are the USB transceiver, the GPIO interrupt logic, and the Cext interrupt. Any enabled event in any of these three pieces of logic will cause the microcontroller to come out of suspend and return to its functional state.

When coming out of suspend the hardware implements the next instruction after the instruction that put the device into suspend mode. If the event that brought the controller out of suspend was an interrupt this interrupt service routine will be implemented after implementing the instruction following the instruction that put the device into suspend. Therefore the next instruction after a suspend is usually a NOP so that the device can jump the ISR directly after coming out of suspend.

CEXT interrupt

The CY7C630/10x microcontrollers have a feature which is intended for devices that are in suspend mode and need to be awaken periodically to be polled. The CEXT pin on the microcontroller must be tied to ground through a capacitor and tied to V_{CC} with a resistor. The RC time constant created by these two components can be tuned by changing the values of R and C. This circuit will generate an interrupt after the amount of time setup by the RC circuit has expired. It is the responsibility of the firmware to reset this pin and interrupt for the next time the device goes into suspend. Resetting the CEXT pin consists of writing a '0' to the CEXT register (0x22) which discharges the capacitor. If the firmware wants the CEXT interrupt to be enabled the next time the microcontroller goes into suspend the firmware must enable this interrupt in the GIER and a '1' must be written to disable the open-drain output driver. The next time the device enters suspend the RC circuit will start from zero volts and charge up the capacitor creating a rising edge to trigger the device. This pin is used for very low power consumption when waking up the microcontroller from suspend for periodic polling of the system components.

Send 0 Byte Data Packet

When the microcontroller is receiving data from the host computer, it must reply to these transactions with an acknowledgement (or handshake). To acknowledge successfully receiving the data from the host the microcontroller must send back an empty data packet to the host. This handshake packet is set-up using the following piece of firmware.

```
;***********************************
;Send 0 Byte packet to signal end of trans
;***********************************
Send0ByteDataPacket:
   ;Send a data packet with 0 bytes.
;Use this handshake after receiving an OUT
;data packet.
;Enable responding to IN packets and set Data
;0/1 to Data 1.
     mov A, C0h
     iowr EnPt0_TX_Config
   ;Enable interrupts.
     mov A, 08h;[int_mask]
     iowr global_int
WaitForDataToTransfer:
   ;Wait for the data to transfer.
   ;Clear the watchdog timer
     iowr Watchdog
;Bit 7 of USB_EP0_TX_Config is cleared when
;the host acknowledges
   ;receiving the data.
     iord EnPt0_TX_Config
     and A, 80h
     jnz WaitForDataToTransfer
     ret
```

Conclusion

It was the purpose of this application note to gather some of the most basic, yet widely used aspects of the CY7C630/1xx Cypress USB microcontrollers. The CY7C630/10x are very easy to use, and are very practical for use in USB applications after only a small amount of introduction.