Lab 6: Introduction to Interupts

EE 234: Section 2

Michael Mentele

Partner: Jonathan Quissenberry

October 30, 2012

**Introduction**

Similar to Lab 3 in this lab we will use read/write operations to read the state of buttons on Port A to perform different operations for a 4-LED display counting in binary. However the major difference in this lab the use of interrupts.

Interrupts are signals to the processor that the CPU needs to handle something immediately. So the CPU finishes the current instruction then saves where it is in the current program and handles the interrupt. Following that it returns back to normal operation.

We will use timer interrupts to ensure that every second we update the LED display with the new number. Then we will engage an external interrupt so that the current display is stored on peripheral LEDs with the rising edge of SW1.

There are four essential cases that we are concerned with in this lab:

1. You press and release PB1 (Count Up)
2. You press and release PB2 (Count Down)
3. You press and release PB1 & PB2 (Stop)
4. Following the first press and release of PB1 & PB2 (Reset)
5. Flip on SW1 for INT1 from low to high causes an external interrupt to display the current value of the counter on the peripheral LEDs.

**Hardware**

|  |  |
| --- | --- |
| Quantity | Item |
| 1 | Digilent Cerebot MX4cK Embedded Controller Board |
| 1 | USB A -> micro B programming cable |
| 1 | Switch module |
| 1 | Peripheral LED module |

**Table 1: Hardware**

**Software**

**Design Overview**

Port A is used to read and write to the buttons and onboard LEDs respectively. Port B is used to write to the peripheral LEDs and PORTX is used to read the peripheral switch module. There are three code modules where meaningful ‘work’ is done.

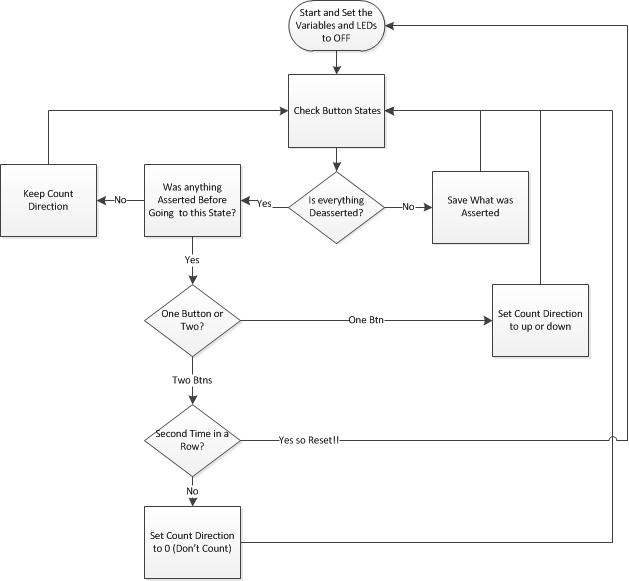
1. Determine\_operation
2. Timer\_interrupt
3. External\_Interupt

**Determine Operation**

In “determine\_operation” we use .DATA variables to remember what buttons were pushed so that we know what to do in the timer interrupt handler. Some of those variables are listed below along with a brief description.

1. Count\_dir – determines what direction to count in the interrupt timer handler
2. Stop – tell the interrupt timer handler to hold the previous values
3. Both – used as a conditional so that the system waits for a deassertion (because both buttons were pressed) before changing the count directions, setting stop = 1, or resetting on the next timer interrupt cycle.

We use essentially the same process as from Lab 3 but instead of a hard coded delay the clock is based off the timer and written during the event handler of the corresponding timer interrupt.



**Figure 1: Determine Operation**

**Timer Interrupts**

Timer interrupt uses the variables named above to count or perform the necessary operations in the handler. The basic process is show in the flowchart below. To set the timer interrupt to happen every one second we had to do some clock manipulation.



**External Interrupts**

The external interrupt process was much simpler. All that was done was that the current state up the on-board counter at the time of the interrupt 1 rising edge was written to the peripheral LEDs and that values was displayed until another overwrote it.

**Binary Counter**

Because you are using a button as the input you are waiting for a de-assertion to before you change which your count direction. Variables are used to remember what was pressed prior to a timer interrupt. These variables are stored as the program continually cycles; checking the button states in a loop that takes a second to time out to the counting function. After 1 second the program counts based on what was the last bit asserted in the check states section. This design scheme allows the system to go “up” or “down” from its current state. This was accomplished by saving the action to take at the next interrupt when all the buttons are de-asserted.

**Test Procedure and Results**

**Methodology**

Testing was performed from a “Top Down” viewpoint where functionality is tested first followed by a debugging phase before finally being tested by an independent user. We also made great use of the step through function in the iterative design project. This greatly reduced the time it took to debug the program before getting a testable prototype.

**Procedure**

1. Test Correct Operation
   1. Test each possible case
      1. Does it count up and down?
      2. Does it stop?
      3. Does it reset?
      4. Do the operations execute in the correct sequence and after the corresponding buttons are pressed?
      5. Do operations happen in even time intervals in due to the timer interrupts?
      6. Does the external interrupt work?
2. Attempt to Break Current Build
   1. Counting Up
      1. Count Up from Stop
         1. Does it continuously count up after PB1 is pressed?
            1. Hold down PB1. Does anything change?
            2. Assert PB1 several times.
      2. Count Up after Counting down
         1. Does it wait until after the current cycle to count up?
         2. Hold down PB1. Does anything change?
      3. Count Up from Reset
         1. Does it begin counting?
         2. Hold down PB1. Does anything change?
   2. Counting Down
      1. Repeat Similar sequence as for Counting Up
   3. Stopping & Resetting
      1. Hold one button down for a few seconds and then assert the other.
      2. Do the same thing for de-asserting the buttons.
   4. Timer Interrupt
      1. Does it time for a second?
   5. External Interupt
      1. Does it save the current value?
      2. Does it hold it?
      3. Can we overwrite it by flipping the switch again?
3. Let someone else play it!

Using Murphy ’s Law… and another perspective; this is an effective way to achieve unexpected results.

**Results**

When starting the board it waited with the LEDs off until a button was pressed. When button one was pressed it counted up in binary across the LEDs starting on the right. So from the right it counted up like so:

000 -> 001 -> 010 -> 011 -> 100 -> 101 -> 110 -> 111 -> … 1111-> 000

0 1 2 3 4 5 6 7 15 0

When you click button two from the start state it counts down in the opposite manner from 15 to 0 and back to 15.

When an opposing button is clicked when counting up or down on the next second interval it will count the opposite direction. So you press button one between states while counting down then when it counts next it will count up. Meaning if in the ‘0’ state and you press PB1 while counting down it will go to the ‘1’ state when it counts.

Pressing both buttons freezes the count as expected and pressing them both sets all the LEDs to off and resets the program and waits for another input to occur.

The timer1 interrupt timed evenly and the external interrupt stored values correctly. When you flipped SW1 it would store the on-board LEDs to the peripheral LED display. It will do so for 100+ iterations… possibly 101 as well.

The program meets the design criteria for the lab.

**Answers to Questions**

**1. Explain how you were able to achieve one second increments/decrements with your assembly solution. As part of your answer provide the basic mathematical computation(s) and value(s) of register(s) required to acquire your one second interval.**

The definition of Hz is Periods/S so for this lab in order to have an interrupt every one second we need to have our clock speed and timer period ratio to be 1.

We begin by setting the system clock speed to 80 MHz as shown in the code below.

#pragma config FNOSC = PRIPLL

#pragma config FPLLMUL = MUL\_20

#pragma config FPLLIDIV = DIV\_2

#pragma config FPBDIV = DIV\_8 /\* Divide SYSCLK by 8 ==> PBCLK \*/

#pragma config FPLLODIV = DIV\_1

The peripheral clock speed is also set by dividing the SYSCLK by 8. This should give you a PBCLK of 10 MHz. Now we will use a prescaler of 256 by setting bits 4 and 5 to 1s. This reduces the clock speed to 39.0625 KHz. So we set our PR1 register to the same amount to achieve a frequency of 1 Hz.

\*\*\* In actuality, for unknown reasons, it did not work out this way. However one possibility is that the SYSCLK was not running at the optimal 80 MHz frequency as it is a real world hardware component but that is just speculation.

For our system we used a 40MHz PBCLK. Our period register was 0x08FF and our pre-scalar was set to 256.

**2. Explain trade-offs between using a 16-bit versus a 32-bit timer to solve this problem.**

A 32-bit timer with our board would mean using two timers. The thirty two bit allows for more resolution if you so desire. This also means using up more of your timers however and you only have a finite amount. This means that for timing with a large SYSCLK you will need to scale down your clock because your period timer will be limited to about 65K whereas with a 32-bit you can have a period or count up to about 4 billion.

**Conclusion**

Interrupts add a whole new cadre of applicable functionality for solution generation. The scope of projects we can easily manage is exponentially increased with the introduction to this tool. Our design functioned as needed though I would like to clean up the code so it is more reusable.

Something to think about is that we jumped back to main when we reset inside a loop so even though it overwrites JR are we going to run out of memory if we reset a bunch of times? One way to fix this would be to set the count direction to 0 and reset the LEDs without jumping back to main. Depends on what your definition of ‘reset’ is.