ECE 3720

Microcomputer Interfacing Laboratory

Section 6

Timothy Driscoll

Date Performed: October / 10 / 2019

Lab 6

ABSTRACT:

A lab designed to show the application of the PIC32MX150F128D microcontroller’s peripheral pin select functionality to create a project with multiple interrupts.

**INTRODUCTION:**

In this lab, we created a new project that showed the implementation of peripheral pin select on a microcontroller to enable the use of multiple interrupts. By using the microcontroller’s peripheral pin select (PPS) we were able to enable a second interrupt besides INT0, which is the only hard-mapped external interrupt on the microcontroller. By then creating two interrupt service routines and using INT0 and INT1 triggered by rotating and optical encoder we created a functional lab. The microcontroller was programmed to count from 0 to 15 and wrap around. As the optical encoder is rotated clockwise it will increase the count by one and as it is rotated counter clockwise it will decrease that count by one. The count was to be displayed on four LEDs. The optical encoder had and A and B output that corresponded to the two interrupts and were used to determine whether the count should be incremented or decremented.

**EXPERIMENTAL PROCEDURES:**

In this lab, we started by unpacking the microcontroller and attaching the chip kit in the proper orientation to ensure that the pins wouldn’t have a flipped result. Then we could start wiring the circuit to complete the project. This circuit is represented by figure 1 below. The circuit includes the two necessary external hardware components, the four LEDs and the Grayhill 61C Optical Encoder. The LEDs were specifically wired to pins 20-23 which correspond to registers B0-B4. This was done to allow us to easily model and reuse some of the code needed in lab 5 and lab 1, which both implemented a count displayed on four LEDs. Outputs A and B were then wired to pin 37 (register B7) and pin 33 (register C4) respectively. Register B7 was specifically used because INT0 is the only hard-mapped external interrupt on the microcontroller and it is connected to pin 37. Register C4 was chosen because it was a 5V tolerant pin that INT1 could be mapped to using PPS. After the completion of the circuit wiring, we created a new project using the MPLAB X IDE software. After creating the project, we added a new C type source file where we wrote the code in order to have the microcontroller function properly. The code can be viewed below in the figures and tables section. The flow of this code starts with the interrupt service routine for the INT0 external interrupt. This block of code tells the microcontroller what to do when the INT0 interrupt has occurred. The next block of code that was included was the second interrupt service routine which is for the INT1 external interrupt. This block of code tells the microcontroller what to do when the INT1 interrupt has occurred. Both of the interrupt service routines are used to determine whether the global variable “count” should increment or decrement based off of the input values of A and B. Now looking at the main function INTEnableSystemMultiVectoredInt() had to be included to tell the microcontroller to handle interrupts. Then the PPSInput() function was called in order to map INT1 as an input to register C4. After this the interrupt parameters were set for both interrupts, the count variable was initialized to 0, the LEDs were declared as outputs and the interrupts were set as inputs. Within in the infinite while loop the code including setting the LEDs equal to the count and lastly restricting the count from 0-15. If count decremented below 0 it would wrap to 15 and if it was incremented greater than 15 it would wrap to 0. The program could then be loaded to the microcontroller and the circuit could be tested using the NI-ELVIS Board and software.

**RESULTS and DISCUSSION:**

In this lab, we were successfully able to get the desired output from the project and achieve all the desired goals described in the slides. This involved having the four LEDs light up displaying a count that would sequentially increase from 0 to 15 as the optical encoder was rotated clockwise. If the optical encoder was rotated counter clockwise the count displayed on the LEDs would sequentially decrease. The project also worked so that the count would wrap, so when the count reached 15 it would wrap back to 0 and when the count hit 0 it would wrap up to 15. For this lab to work there were two main things we needed implement which included mapping another interrupt to a given pin and correctly implementing the two interrupts. Since there is only one hard mapped interrupt on the microcontroller, we had to use peripheral pin select to map the second interrupt to a given pin. Since the interrupts were both inputs in order to map the second interrupt, we had to use the PPSInput function. The PPSInput function excepted three different arguments which included the group, function and pin. The pin and function were easily identified to be INT1 and C4 respectively. Where INT1 was the desired external interrupt and C4 was the register corresponding to the pin we wanted the interrupt to be mapped to. The last parameter that needed to be imputed was the group, this value could be found by looking at far right side of table 11-1 in the PIC32 Datasheet and determining which group INT1 corresponded to (1-4). After mapping the second external interrupt we were able to initialize the parameters of both interrupts which were to be the same. To start the interrupts were triggered on a rising edge, but after each time the interrupt was triggered the edge detection had to be inverted. This inversion occurred because after each rising edge the next time the interrupt was to be triggered was at the corresponding falling edge and vice versa. In order to set this up we used the INTCON register and set it to 0 and in each ISR we used the != command to invert the value in the register. There were three other registers that had to be setup in order to properly initialize the interrupts. These registers included the IEC, IFS and IPC. The IEC register was set to a one for both in order to enable the interrupts. The IFS register was set to a zero for both to indicate that the interrupt has not yet occurred. Lastly the IPC was initialized to a 1 for INT0 and 2 for INT1 to give the interrupts the highest and second highest priority. After the interrupt was triggered it sets a flag which tells the microcontroller that an interrupt has occurred. The microcontroller then saves the main state and looks up to the ISR corresponding to the specific interrupt. Each of the two ISR contained the code that was necessary to determine whether the optical encoder had been rotated clockwise or counter clockwise. The code we implemented to solve this problem involved a series of if statements that looked at the values of both A and B to determine whether the count should be incremented or decremented. For INT0s ISR the logic evaluated how A changed with relation to the value of B. For INT1s ISR the logic evaluated how B changed with relation to the value of A. Both ISRs the flow included getting the state of the two interrupts (values of A and B) using two port commands, determine the direction of rotation, clear the interrupt flag and lastly invert the INTCON register. The only error that I encountered in this lab related to not inverting the INTCON register in each ISR. This resulted in the count incrementing/decrementing sporadically instead of with each individual rotation step. This happened because the interrupts were only being triggered on the rising edges. For the lab to function properly the interrupts had to be triggered on both the rising edges and their corresponding falling edges.

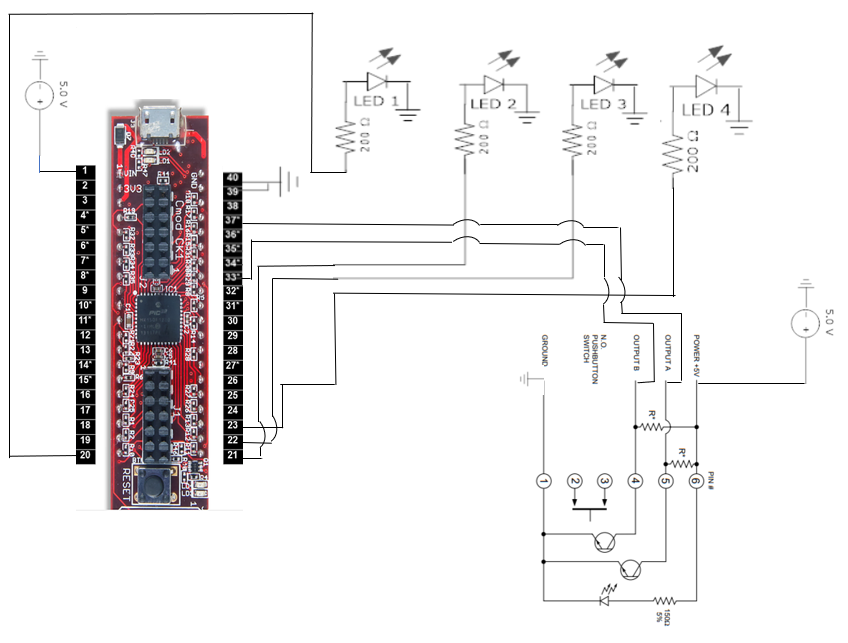
**CONCLUSSION:**

We have come to the conclusion that in this lab there were some key procedural points that needed to be learned in order to achieve all the goals. To start we needed to first understand how we could map another interrupt to a certain pin on the microcontroller. By implementing the PPSInput() function we were map INT1 to the pin of our desire. The next key point that we had to understand was how interrupts functioned and how the microcontroller handled them. I described the interrupt functionality in the section above. After learning the functionality of the interrupt, we could successfully initialize and write an ISR to have the optical encoder trigger our desired results. We then had to understand the logic behind the optical encoder. Understanding the logic allowed us to determine the direction of rotation so we could properly increment or decrement the count. In order to put the lab all together It was critical that we learned how to determine the proper registers that corresponded to the needed pins as described in the Lab 6 PowerPoint. Lastly, we needed to rely on the PIC32 Datasheet in order to determine how to map a new interrupt to a pin and setup the interrupt with the correct code to achieve the desired outputs. By working through the lab, I was able to determine all the necessary procedural points to understand and achieve the desired outputs for the lab.

**REFERENCES:**

Clemson University’s ECE 372 Lab 6 PowerPoint.

**FIGURES AND TABLES:**

****

**Figure 1: Wiring for lab 6 (Pin connections described in experimental procedures)**

**CODE:**

void \_\_ISR(3) interruptA()

{

int A = PORTBbits.RB7; // Get output A from pin 37

int B = PORTCbits.RC4; // Get output B from pin 33

if (B == 1)

{

if (A == 0)

count++;

else

count--;

}

else

{

if (A == 1)

count++;

else

count--;

}

// Clear interrupt flag

IFS0bits.INT0IF = 0;

/\* Upon firing interrupt polarity must change to look

change in output \*/

INTCONbits.INT0EP = !INTCONbits.INT0EP;

}

void \_\_ISR(7) interruptB()

{

int A = PORTBbits.RB7; // Get output A from pin 37

int B = PORTCbits.RC4; // Get output B from pin 33

if (A == 1)

{

if (B == 0)

count--;

else

count++;

}

else

{

if (B == 1)

count--;

else

count++;

}

// Clear interrupt flag

IFS0bits.INT1IF = 0;

/\* Upon firing interrupt polarity must change to look

change in output \*/

INTCONbits.INT1EP = !INTCONbits.INT1EP;

}

main(){

INTEnableSystemMultiVectoredInt();

// Reset register B to read digital

ANSELB = 0;

// Map INT1 to pin 33

PPSInput(4, INT1, RPC4);

// Set interrupt settings

INTCONbits.INT0EP = INTCONbits.INT1EP = 1; // Rising edge

IEC0bits.INT0IE = IEC0bits.INT1IE = 1; // Enable interrupt

IPC0bits.INT0IP = 1; // Set INT0 priority 1

IPC1bits.INT1IP = 2; // Set INT1 priority to 2

// Clear all interrupt flags so that interrupts aren't triggered on boot

IFS0 = 0;

count = 0;

TRISB = 0x00; //Set all B register pins to output.

// Set pins 33 and 37 to input

TRISBbits.TRISB7 = 1;

TRISCbits.TRISC4 = 1;

while(1)

{

// Check bounds to see if wrap-around is required

if (count > 15)

count = 0;

else if (count < 0)

count = 15;

LATB = count; //Output count to B

}

}