Laboratory 7
Rotary Pulse Generator

NAME: Ryan Barker

CLASS: ECE 372 Section 006

DATE: October 9th, 2014

OBJECTIVE: The object of this lab was to design and implement an interface to a rotary switch that would light the LEDs on the board in such a way that the they appeared to increase in position (lit LED moved up one) when the dial was turned right and decrease in position (lit LED moved down one) when the dial was turned left. This required two interrupts (One each for a change on either of the input pins) and an insane amount of manipulation to the port polarity select (PPS) registers for the input ports. Operation and interfacing of the hardware was done by a C program.

EQUIPMENT USED: Freescale PBMCUSLK Student Learning Kit
Rotary Pulse Generator
Two 8.2 kΩ resistors (to limit the current from the power supply to the micro-controller)
Lab Workstation with CodeWarior C/C++ IDE

PROCEDURE: The design of the interface was discussed in lab and port P pin 0 and port J pin 7 were used to take input from the rotary pulse generator to the micro-controller. The rotary switch's output cycled through four states, and a change on either of the input pins was used to predict the next state of the pin, update the PPS register for that pin accordingly, and change the state of the light. The hardwired LEDs were used, so the output ports were port B pins 4-7 and therefore did not need to be wired manually. Once the system was wired, a C program was written to control the interface, which was then tested to verify proper operation.

DESCRIPTION OF CODE: Before I even begin to describe operation of the code, we must establish that the PPS registers work by pulling that each of a pin low or high and checking those pins for either a rising or falling edge depending on if a one or zero is stored in the PPS register for that pin. A zero in PPSP pin 0 means that P0 will be pulled high and checked for a falling edge (1 to 0). A 1 in PPSJ pin 7 means that J7 will be pulled low and checked for a rising edge (0 to 1).

That being said, initialization for the code works by first setting up port P channel 0 and port J channel 7 for input. It then enables pulling resistors on those pins and sets port B channels 4-7 as output.

Now, the complicated part. The initial state of PPSP and PPSJ depend on whatever is currently stored in P0 and J7, which will be the initial state. So, before we have done anything, we have to read in the values in port P and port J and set PPSP and PPSJ accordingly. If P0 is a zero, PPSP is set to 0xFF to check P0 for a rising edge, since the next change in P0 will be from a zero to a one. If P0 is a one, PPSP is set to 0xFE to check P0 for a falling edge, since the next change in P0 will be from a one to a zero. The same

logic applies to J7 and PPSJ, however the hex values used are different since we are setting or clearing PPSJ pin 7 (They are 0xFF and 0x7F).

Finally, interrupts are enabled for P0 and J7 and then enabled globally. The code then enters an infinite loop that waits for an interrupt before doing anything.

Inside the interrupts (56 for P and 24 for J according to the vector table), the state of ports P and J are read, and then the PPSP register (in the port P interrupt) or PPSJ register (in the port J interrupt) is updated based on what the next change will be in that pin (the logic is identical to the logic used in initialization of these registers). Also, depending on the current state of P0 and P7, PORTB is updated to the correct LED configuration. Finally, these interrupts write either a 0x01 to PIFP or 0x80 PIFJ, depending on the interrupt that was called (which clears the interrupt flag on the pin it was set on).

OBSERVATIONS AND DISCUSSIONS: Circuit design this time was very simple (it only required four wires), and most of the work done was in the code, as described above. Dynamic interrupt handling is a complicated process that requires very extensive care and special attention to the PPS registers, so they can accurately predict the next state change on their respective pins of their respective ports. Note that the resistors used were necessary to drop some voltage and limit current levels so the five volt supply didn't blow up the micro-controller when hooked into the inputs.

CONCLUSIONS: The main thing I learned from this lab is that rotary pulse generation is complicated and requires extensive manipulation of the port polarity select registers to achieve proper operation. These register must be reset to the correct values every time a state change on their respective ports occur to accurately predict the next state change of the pins, allowing for interrupts on those pins to be called correctly.

SIGNATURE: *"This report is accurate to the best of my knowledge and is a true representation of my laboratory results."*

Signed Ryan Barker.

C CODE

Headers and Main:

```c
#include <hidef.h>      /* common defines and macros */
#include <mc9s12dt256.h>     /* derivative information */
#include "pbs12dslk.h"
#include "lcd.h"

#pragma LINK_INFO DERIVATIVE "mc9s12dt256"

void interrupt_function1();
void interrupt_function2();

void main(void)
{
   DDRP = 0xFE;  // Bit 0 = Input
   DDRJ = 0x7F;  // Bit 7 = Input

   PERP = 0x01;  // Bit 0 = Enable Pulling
   PERJ = 0x80;  // Bit 7 = Enable Pulling

   DDRB = 0xF0;  // Enable Port B pins 4-7 for Output
   PORTB = 0xFF;  // Initialize LEDs off

   if((PTP & 0x01) == 0x00) {
      if ((PTJ & 0x80) == 0x00) {
      // P = 0 J = 0
      PPSP = 0xFF;  // Check P0 for rising edge
      PPSJ = 0xFF;  // Check J7 for rising edge
      } else {
      // P = 0 J = 1
      PPSP = 0xFF;  // Check P0 for rising edge
      PPSJ = 0x7F;  // Check J7 for falling edge
      }
   } else {
      if ((PTJ & 0x80) == 0x00) {
      // P = 1 J = 0
      PPSP = 0xFE;  // Check P0 for falling edge
      PPSJ = 0xFF;  // Check J7 for rising edge
      } else {
      // P = 1 J = 1
      PPSP = 0xFE;  // Check P0 for falling edge
      PPSJ = 0x7F;  // Check J7 for falling edge
      }
   }

   PIEP = 0x01;  // Bit 0 = Enable Interrupts
   PIEJ = 0x80;  // Bit 7 = Enable Interrupts

   EnableInterrupts;

   while(1);
   /* please make sure that you never leave this function */
}
```

Port P Interrupt:

```
void interrupt 56 interrupt_function1() {
   if((PTP & 0x01) == 0x00) {
      if ((PTJ & 0x80) == 0x00) {
      // P = 0 J = 0
      PPSP = 0xFF;  // Check P0 for rising edge
      PORTB = 0x70; // State 1
      } else {
      // P = 0 J = 1
      PPSP = 0xFF;  // Check P0 for rising edge
      PORTB = 0xE0; // State 4
      }
   } else {
      if ((PTJ & 0x80) == 0x00) {
      // P = 1 J = 0
      PPSP = 0xFE;  // Check P0 for falling edge
      PORTB = 0xB0; // State 2
      } else {
      // P = 1 J = 1
      PPSP = 0xFE;  // Check P0 for falling edge
      PORTB = 0xD0; // State 3
      }
   }

   PIFP = 0x01;  // Clear interrupt on pin 0
}
```

Port J Interrupt:

```
void interrupt 24 interrupt_function2() {
   if((PTP & 0x01) == 0x00) {
      if ((PTJ & 0x80) == 0x00) {
      // P = 0 J = 0
      PPSJ = 0xFF;  // Check J7 for rising edge
      PORTB = 0x70; // State 1
      } else {
      // P = 0 J = 1
      PPSJ = 0x7F;  // Check J7 for falling edge
      PORTB = 0xE0; // State 4
      }
   } else {
      if ((PTJ & 0x80) == 0x00) {
      // P = 1 J = 0
      PPSJ = 0xFF;  // Check J7 for rising edge
      PORTB = 0xB0; // State 2
      } else {
      // P = 1 J = 1
      PPSJ = 0x7F;  // Check J7 for falling edge
      PORTB = 0xD0; // State 3
      }
   }

   PIFJ = 0x80;  // Clear interrupt on pin 7
}
```