USC Viterbi School of Engineering

Ming Hsieh Department of Electrical Engineering

EE 109L - Introduction to Embedded Systems

Spring 2015 Lab Assignment #8
Rotary Encoders

1 Introduction

In this lab exercise you will learn how rotary encoders work. The outputs of the encoder will be connected to your Arduino and turning the encoder will cause the Arduino to change a numerical value up or down and then display the value on an LCD display. In the second part of the lab, the performance of the program will be improved by using interrupts to monitor the inputs from the encoder.

This lab will also introduce the concept of splitting your program into multiple files. All software projects of medium size or larger have the source code divided into separate files to make editing and debugging easier. If this lab we will put all the LCD routines in a separate file.

2 The Rotary Encoder

A rotary encoder is used to determine the angular position of a something that rotates. For example, you are building a weather monitoring system and one of the devices providing input to the system is a weather vane that rotates to point in the direction the wind is blowing. The weather vane could be attached to a rotary encoder and the system would read the inputs from the encoder to see what angle the vane is pointing. Rotary encoders are also commonly used in devices that have a knob on them that the user needs to turn to adjust something.

Rotary encoders come in two types: absolute encoders and incremental encoders. An absolute encoder has outputs to provide a binary number for each angular position of the shaft (0000 = 0°, 0001 = 22.5°, 0010 = 45°, etc.) The number of bits the encoder provides determines the resolution of the angle that can encoded. For example a 10-bit absolute encoder can resolve angles to 360/1024 = 0.35°.

Incremental encoders (Fig. 1), also called quadrature encoders, are also available in a variety of resolutions (16, 64, 256, etc. per revolution) but they only provides information about whether the shaft has been turned clockwise or counter-clockwise from the previous position, nothing about the actual position of the shaft. As the shaft of one of these encoders is turned, it opens and closes two switches (Fig. 2) to generate two binary

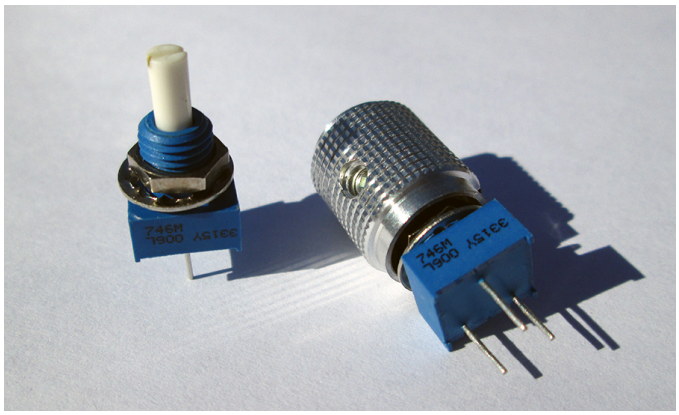

Figure 1: Incremental rotary Encoders

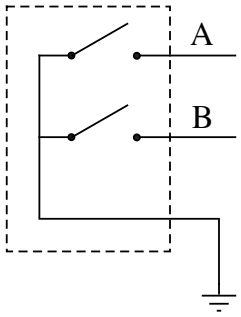

Figure 2: Internal circuit of the encoder

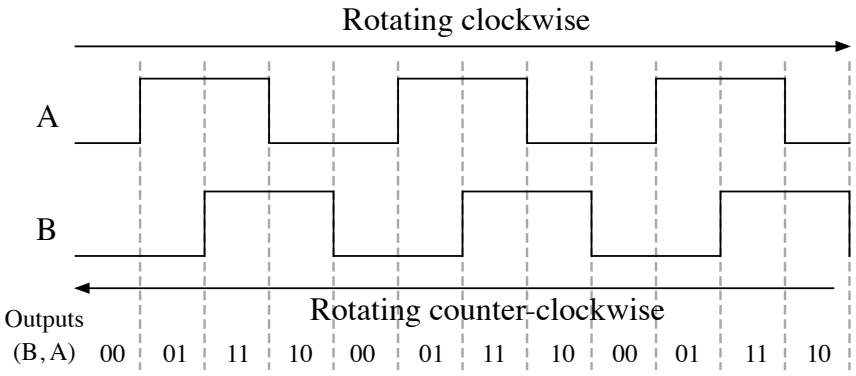

Figure 3: Output of the pins of an encoder while being rotated

signals. The two signals both look like a 50% duty cycle square wave (Fig. 3) but they are 90° out of phase with each other, which gives the appearance when drawn that one of the square waves is lagging behind the other by one quarter of a period.

To see how this can work, examine Fig. 3 and move along the waveforms from left to right as they would be generated when the control is being rotated clockwise. If we consider the two signals as the outputs of state machine where B is the MSB and A is the LSB, then the output code goes through the sequence 00, 01, 11, 10, 00, 01, etc. as the control is rotated.

Now try moving along the waveforms from right to left (control being rotated counter-clockwise) and we can see that the output code goes through the sequence 00, 10, 11, 01, 00, etc. as the control is rotated. Note that the sequence is different depending on whether the control is being rotated clockwise or counter-clockwise. In fact every one of the transitions from one two-bit code to another in the clockwise direction is different from the ones that appear when rotating counter-clockwise. As a result whenever there is a code change the direction of rotation can be determined by knowing the old code and the new code.

The above sequences of numbers is called a Gray code and has the property that as you go from one number in the sequence to the next, in either direction, only one bit changes at every transition. Gray codes, usually with many more than two bits in them, are also used in absolute rotary encoders used to encode the position information in electomechanical devices such as scanners, printers, manufacturing equipment and many others. The property that only one bits changes at a time helps to eliminate errors in sensing the device's position. If a normal binary code was used there would be numerous places in the sequence where multiple bits would have to change simultaneously. With mechanical equipment, it's impossible to ensure that all the bits would change at exactly the same time and this can lead to errors as to the position or status of the device.

Using the two-bit Gray code number as a state variable we can construct a state diagram (Fig. 4) for a state machine that describes the output of the encoder. Each of the four states corresponds to one of the four sets of output values the control can generate. From each state there are two transitions to an adjacent state, one for each of the code bits that might change. One transition is for clockwise rotation and the other is for counter-clockwise rotation. The direction of rotation is marked on the transition line.

The encoders provided for the lab exercise go through 64 states in the process of one revolution of the knob. The pins on the bottom are arranged with two pins (A and B) on one side, and the pin that goes to ground is on the opposite side. It's not too important which pin is A or B since swapping them only makes the state machine count in the opposite direction for a given direction of rotation.

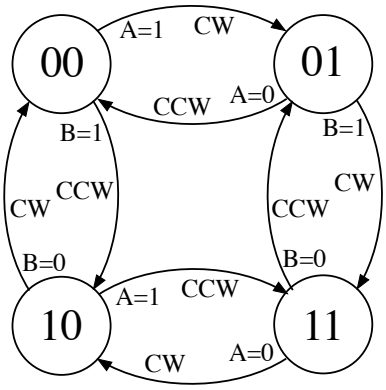

Figure 4: State diagram for a rotary encoder

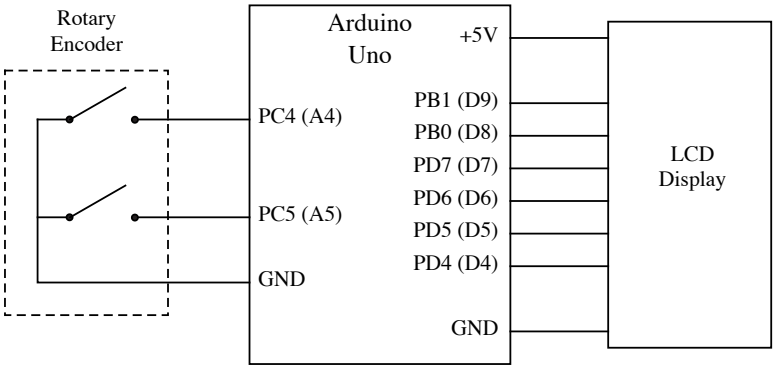

Figure 5: Circuit diagram for Lab 8

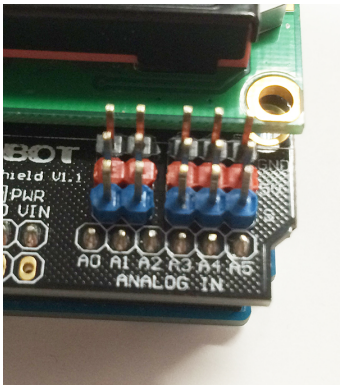

Figure 6: Input pins on the LCD shield

3 The Circuit

Build the circuit shown in Fig. 5 on your breadboard with the rotary encoder and make the necessary connections between the breadboard and your Arduino. Note that the connections to the LCD are made by simply plugging the Arduino and LCD together.

Once the LCD is mounted on the Arduino, you no longer have access the connectors on the Arduino to make connections to the various I/O ports pins as you were able to do in previous labs. However the ones that are not being used by the LCD can still be accessed by using the pins on the top of the LCD board. In the bottom right corner of the LCD (see Fig. 6) are three row of five pins with different colored bases. The top row (black base) are all ground connections. The second row (red base) are all $+5V$ connections. The bottom row (blue base) are connections to I/O ports A1 through A5 (PC1 through PC5). The pins for A4 and A5 are the ones just above the A4 and A5 labels on the board.

From the instructor get a pack of male-female jumper wires. The female end can be pushed down onto a pin on the LCD shield and the male end can then be inserted into one of the breadboard holes. Use these jumpers to make the three connections needed between the breadboard and the Arduino as shown in the schematic.

4 Split Program Into Multiple Files

As with previous labs, create a `lab8a` folder in your `e109` folder and put a copy of the `Makefile` in there. Pick one of your program files from an earlier lab and copy it to this folder and name it `lab8a.c`. This lab will use the LCD functions that you developed in Lab 6 and also used in Lab 7, but rather than just copying and pasting the functions into the main `lab8a.c` source code file for this lab, we're going to put them in a separate file. Create a new file "`lcd.c`" and put all the LCD functions (`init_lcd`, `stringout`, `moveto`, `writedata`, `writecommand`, and `writenibble`) in that file (don't put them in `lab8a.c`). Also create a new header file "`lcd.h`" and into that file copy the function prototypes from your Lab 6 or Lab 7 code for all the functions just listed EXCEPT for the one for `writenibble`. The file should have just 5 lines in it. At the top of your new `lcd.c` file put the following lines

```
#include <avr/io.h>
#include <util/delay.h>

#include "lcd.h"
```

and after that put a copy of the function declaration for the `writenibble` routine. The reason the declaration for `writenibble` is not included in the `lcd.h` file is that that function is only used by the other functions in the `lcd.c` file, so there is no reason for it to be known by the rest of the program. Think of it as a private function that can only be used by the other functions in `lcd.c`.

The `#include` line for the `lcd.h` must use double quotes instead of the '`<`' and '`>`' characters. The angle brackets tell the compiler to look for the header file in the development software directories and are always used for including header files that are part of the development software. The double quotes tell it to look for it in the local directory are should be used with header files you create.

Once you have the LCD routines in a file by themselves, you can use them from `lab8a.c` or any other files you create for this lab by just putting the line

```
#include "lcd.h"
```

near the top of the file. This will read in the five function declaration from the `lcd.h` file.

The last thing that has to be changed to work with multiple source code files is in the `Makefile`. The line that lists the program object files needs to include the names of all the "`.o`" files that must be linked together. For this lab it should be

```
OBJECTS  = lab8a.o lcd.o
```

5 Program 8A

For lab program 8A, the encoder will be used to control the incrementing and decrementing of a signed variable and the value of that variable will be displayed on the LCD shield. The variable starts with a value of zero and then increments or decrements by one for each state change of the encoder outputs. Turning the encoder in one direction increments the count, turning in the opposite direction decrements it. Since there are 64 encoder state changes for each revolution, the count should go up or down 64 each time the encoder is turned one revolution. The encoder outputs will be read by the microcontroller by polling the appropriate PIN registers bits. Interrupts should **not** be used in part 8A.

1. Two I/O port bits will be required for reading the encoder output into the Arduino. With the LCD attached to the Uno, several of the I/O ports bits are used by the LCD. Port D, bits 4-7, Port B, bits 0-2, and Port C, bit 0 are used by the LCD. In addition Port D, bits 0 and 1 should not be used since they are used by the USB interface for programming. To standardize things, everyone should use the A4 and A5 bits (Port C, bits 4 and 5). Make sure to enable the pull-up resistors for these two port bits.

2. Use the LCD routines that you have used in previous labs to handle the display. Move all the code related to the LCD into a separate file as described above. Make sure you have added the name of the

LCD file (with the "`.o`" extension) to the "`OBJECTS`" line of the Makefile. When you do a "`make`", the development software will link the separate binary files together to create the executable file.

3. Allocate an "`int`" variable to store the count value that the encoder will increment and decrement. Your program should start by initializing the count variable to zero and displaying this on the LCD.

4. Your program should continually monitor the two input bits to see if either of them has changed value. As the encoder is rotated in either direction, the input values will change and the program should determine whether to increment or decrement the variable and display the new value on the LCD. You should be able to rotate the knob back and forth and see the numbers change on the display. Also confirm that the numbers can go both positive and negative.

5. These encoders go through 64 states per revolution. Confirm that rotating the knob one full revolution in either direction makes the count change by 64.

6 Pin Change Interrupts

In this part of the lab we will modify the program to use interrupts. To see the reason for doing this, try a simple experiment.

1. Turn the knob on your rotary encoder so the black line is visible and lined up with something like a corner of the encoder or some other feature.

2. Use the reset button on the LCD shield or the Uno to restart the program and zero the count value.

3. Spin the control knob through one full revolution as quickly as possible.

4. Note the count value on the LCD.

If the code was keeping up with the encoder outputs, the count value should have gone to 64 (or -64). However the count is probably less than that meaning that the program missed some of the encoder state transitions. This was most likely caused by the program being stuck in a delay routine in the LCD code when one or more transitions occurred. One way to prevent this from happening is to make the state changes trigger an interrupt. Regardless of what the program is doing at the time, the interrupt will cause execution to jump to the interrupt service routine to handle the state change. The program can service the interrupt while executing a delay routine and prevent encoder state transitions from being lost.

To implement this we will use the Pin Change Interrupt capability of the microcontroller. All of the I/O pins in the three ports (B, C and D) are capable of generating an interrupt if the pin does a $0 \rightarrow 1$ or $1 \rightarrow 0$ transition. While individual pins in a port can trigger an interrupt, there is only one interrupt vector (and associated Interrupt Service Routine) for each port. A change on any pin in Port B will result in executing the ISR associated with the PCINT0 vector. Similarly, pins in Port C can invoke the PCINT1 ISR, and pins in Port D can invoke the PCINT2 ISR.

Pin Change Interrupts for a port are enabled by setting the `PCIE0`, `PCIE1` or `PCIE2` bits in the `PCICR` register. In addition there is an 8-bit mask register (`PCMSK0`, `PCMSK1` and `PCMSK2`) associated with each port that determines which pins can generate an interrupt. If a bit in the mask register is set to a one, then a $0 \rightarrow 1$ or $1 \rightarrow 0$ transition on the corresponding bit in the I/O port will cause a Pin Change Interrupt. If a bit in the mask register is a zero, then changes on the corresponding I/O port bit will not cause an interrupt.

The Pin Change Interrupts are ideal for our application of watching for state changes on the two bits from the encoder. Once the PC interrupts are enabled on the two bits, any change of state will cause the interrupt to occur. The ISR can include the code that determines what state change happened and whether to increment or decrement the count value.

7 Program 8B

Make a `lab8b` folder inside the `ee109` folder as you did before for part 8A. Copy the files from the `lab8a` folder to the `lab8b` folder and rename the main program file to `lab8b.c`. Also make changes to the `Makefile` so it compiles the `lab8b.c` program. The following changes and/or additions can be made to the program.

1. As with any program that uses interrupts it will need an "`#include <avr/interrupt.h>`" line somewhere near the start of the program.
2. To get the interrupts working
   (a) Set the `PCIE1` bit in the `PCICR` register to enable the pin change interrupts on Port C (PCINT1).
   (b) Set the bits in the `PCMSK1` mask register to enable interrupts for the PC4 and PC5 I/O lines. For PC4 and PC5, these are bits `PCINT12` and `PCINT13`.
   (c) Enable global interrupts.
3. Write the "`ISR(PCINT1_vect)`" interrupt service routine. Into this routine you should move the code from the main program loop that checks the state of the input bits and adjusts the count value accordingly. Don't move the code to display the count on the LCD to the ISR. We don't want the microcontroller tied up writing to the LCD while interrupts are disabled in the ISR.
4. The main loop of the program should now just check to see if the count value has changed and if so, update the displayed value on the LCD.

Any variable that needs to be accessed from both the main program and then ISR must be declared as a global variable at the top of the program (not in the main routine). In addition these variables should be declared with the "`volatile`" keyword to tell the compiler that their contents can change outside the main stream of code execution.

After the the changes have been incorporated into the Lab 8B program, try running it and confirm that rotating the encoder still makes the count value go up and down. Once that is working, then try the experiment of rotating the control quickly and see if the ISR-based code does a better job of keeping up with the state changes. Ideally you should be able to spin the knob one revolution and see the count go up or down by 64 which shows that it is not losing any transitions.

8 Results

Once you have the assignments working demonstrate them to one of the instructors. Turn in a copy of the source code for both programs through the link on the class web site.

EE109 Lab 8B Grading Rubric

Name: _____

TA/Instructor Successful Demo initials: _____

Item	Outcome	Score	Max.
Initialization			
• Provided code to initialize LCD	Yes/No		1
• Provided code to initialize appropriate PORT capabilities	Yes/No		1
• Correctly initialized pin change interrupts	Yes/No		1
Encoder Operation			
• Code to read input bits and determine state	Yes/No		3
• Code to adjust count based on state change	Yes/No		3
• Provided an ISR for pin change interrupts	Yes/No		2
• Correct display of count on the LCD	Yes/No		3
Code Organization			
• Separate file for LCD functions	Yes/No		2
• Properly indented code	Yes/No		2
• Well-commented code	Yes/No		2
Total			20
Open ended comments:			