1 Introduction

This semester’s class project is to build a device that measures how fast an object is moving as it passes two sensors and then relays that speed to a remote device. On the measuring device, the time between tripping the sensors, and the speed in cm/sec is displayed on the LCD. The speed value is sent over a serial link to similar speed trap device which displays the received speed. On the remote device the user can adjust a speed threshold, and if the speed received from the other device is above the threshold, it sounds an alarm.

2 Speed Trap Overview

A block diagram of our speed trap is shown in Fig. 1 and it will have the following features.

- Two LED light sources, and two phototransistors light detectors.
- An LED to indicate that a speed measurement is in progress
- An LCD display for showing the measured time delay and speed. The display is also used for setting the speed threshold and displaying the speed received from the remote device.
- A buzzer for playing an alarm tone.
- A control knob for selecting the speed threshold.
- A serial interface (RS-232) to another speed trap unit. When the local unit measures a speed it sends it to the remote unit. When a speed is received from the remote unit it is displayed on the LCD.

3 Operation of the Speed Trap

The operation of the speed trap is relatively simple.

- The speed trap device is always monitoring the state of the first of the two LED-phototransistor pairs.
- When the first sensor indicates that the path between its LED and phototransistor has been blocked, the system starts counting in milliseconds. It also lights up the timing indicator LED to indicate that a timing event is in progress. When the second sensor shows that the path between its LED and phototransistor has been blocked, the timing stops, and the indicator LED is turned off.
- Whenever a timing event has completed, the unit displays the time in milliseconds on the LCD display. It also uses this value to determine the speed in cm/sec that the object was moving when traveling between the two sensors and displays this on the LCD also. There is no specified distance that you must place your two sensors apart on your breadboard. The holes on the breadboard are 0.1 inches apart and wherever you place the sensors, you will need this distance in order to make the speed calculation.
If the user turns the rotary encoder knob, the “maximum speed” number is changed up or down and displayed on the LCD. This number can change from 1 to 99 cm/sec. The maximum speed number should also be stored in the Arduino’s EEPROM non-volatile memory. This means the maximum speed value is stored in memory that is retained even if the power is turned off making it unnecessary for the user to set the speed threshold each time the device is turned on.

When a speed measuring event is completed, the local unit transmits the speed over the serial port to the remote unit. The protocol used for sending and receiving the speed data is specified in Section 5.5. The receiving unit inputs the bytes and then displays that speed in cm/sec on the LCD. For this project it should be possible to do a “loop back” of the transmitted data to the received data port so the speed trap system is essentially sending the data to itself.

If a received speed is greater than the maximum speed set by the rotary encoder, the unit sounds a short alarm using the buzzer.

4 Hardware

Most of the components used in this project have been used in previous labs, and your C code from the other labs can be reused if that helps. The buzzer was used in Lab 4 (ADC) and you can use some of that code to play a tone on the buzzer. The rotary encoder is the same as in Labs 9 and 10. You should use the encoder code that uses interrupts rather than polling.

4.1 Object detector

The two detectors are made from two white LEDs and two phototransistors. The LEDs and phototransistors are positioned so the light from the LED is illuminating the phototransistor unless some object blocks the path (see Fig. 2). The LEDs don’t have to be controlled by the Arduino and can be wired to be on all the time. The phototransistors should each be connected to a digital input port bit and monitored using the pin change interrupts similar to how you handle the rotary encoder inputs. When the phototransistor can “see” the light from the LED the digital input will be a one. When an object blocks the light path, the input will go to a zero.

The LEDs should be installed on one side of the center channel of the breadboard, and the phototransistors opposite them on the other side of the channel. For the LED the longer lead (on side of the case that is
rounded) should go to the positive voltage. For the phototransistor, the shorter lead (on the side of the case that is flattened) should go to the positive voltage. The curved top of the LEDs should be pointing at the phototransistors, and their curved top should be pointing at the LEDs. Both components should be positioned so that some small object about the size of a credit card can be slid along the channel past both sensors without striking them.

The distance between the two sensors will be used in the speed calculation. You can put them wherever you want on the board depending on where your other components are, but the sensors should be at least 1.5 inches apart.

4.2 Serial Interface

The serial interface between speed trap devices will use an RS-232 link to send the speed data between the units. The serial input and output of the Arduino uses voltages in the range of 0 to +5 volts. These are usually called “TTL compatible” signal levels since this range was standardized in the transistor-transistor logic (TTL) family of integrated circuits. Normally for RS-232 communications these voltages must be converted to positive and negative RS-232 voltages levels in order for it to be compatible with another RS-232 device. However for this project we will skip using the voltage converters and simply connect the TTL level signals between the project boards.

For all the speed trap devices to be capable of communicating with others, use the following parameters for the USART0 module in the Arduino. Refer to the slides from the lecture on serial interfaces for information on how to configure the USART0 module for these settings.

- Baud rate = 9600
- Asynchronous operation
- Eight data bits
- No parity
- One stop bit

4.3 Tri-State Buffer

As was seen in Lab 9, if the received data from another device is connected directly to the RX input (Arduino port D0) it creates a conflict with a signal used to program the Arduino’s microcontroller. Both the other device and the programming hardware try to put an active logic level on the D0 input and this can prevent the programming from succeeding. When this happens you will get an error messages like this.

```
avrdude: stk500_recv(): programmer is not responding
avrdude: stk500_getsync() attempt 1 of 10: not in sync: resp=0x00
```
The solution for this is to use a tri-state gate to isolate the other device from the D0 input until after the programming is finished. The gate you will using is a 74LS125 that contains four non-inverting buffers (see Fig. 3). Each of the four buffers have one input, one output and an enable input that controls the buffer’s tri-state output. When the gate is enabled it simply copies the input logic level to the output (0 $\rightarrow$ 0, 1 $\rightarrow$ 1). However when the gate is disabled its output is in the tri-state or hi-Z state regardless of the input. In that condition, the gate output has no effect on any other signal also attached to the output.

As shown in Fig. 4, the received data from the other device should go to one of the buffer inputs, and the output of the buffer should be connected to the D0 port of the Arduino. The enable signal is connected to any I/O port bit that is available for use. When the Arduino is being programmed all the I/O lines become inputs and this will effectively put a logic one on the tri-state buffer’s enable line. This disables the output (puts it in the hi-Z state) and the programming can take place. Once your program starts, all you have to do is make that I/O line an output and put a zero in the PORT bit. This will enable the buffer and now the other device’s received data will pass through the buffer to the RX serial port.

### 4.4 Arduino Ports

The Arduino Uno has 20 general purpose I/O lines in Ports B, C and D. However most of these are shared with other modules and can not be used for general purpose I/O if that module has to be used. In this lab you will be using the serial communications modules which requires using specific I/O lines. In addition, the LCD shield requires using certain port bits. Fig. 5 shows which I/O port bits are allocated for various purposes.
**Figure 5: Use of I/O port bits in this project**

**LCD** - The LCD shield uses PD4-PD7 for the data lines, PB0 and PB1 for control, PB2 for the backlight, and PC0 is the analog signal from the buttons.

**RS-232** - The USART0 serial interface module uses PD0 and PD1 for received (RX) and transmitted (TX) data, respectively.

**Port B, bit 5** - This I/O bit has an LED connected to it on the Arduino board and this can prevent it from being used as an input. You might be able to use it as an output, but avoid using it as an input. In addition, do not use it to control the tri-state buffer. While it can be used for some other output signal, the LED causes it to not work correctly during the flash programming to control the buffer.

When working on your design to read input signals from the sensors and the rotary encoder, make sure to use only the I/O port bits that are not already in use by one of the above modules or the shield. Also think about how your program will be communicating with the devices and how that might affect your decision as to which ports to use for the various devices. For example, if you want to use Pin Change Interrupts to watch for one of the sensors to be activated, it might be better to have both sensors on the same port so one ISR can handle both of them. In addition it might be a good idea to not have them on the port that the rotary encoder is using since your code for that may use Pin Change Interrupts and you may want to have the encoder handled by a different ISR than the sensors. A little planning in advance as to how your program will work can lead to significant simplifications in the software.

### 4.5 Hardware Construction Tips

The LEDs, phototransistors, rotary encoder, and 74LS125, and the various capacitors and resistors should all be mounted on your breadboard. It’s strongly recommended that you try to wire them in a clean and orderly fashion. Don’t use long wires that loop all over the place to make connections. You will have over 10 wires going from the Arduino to the breadboard so don’t make matters worse by having a rat’s nest of other wires running around on the breadboard. Feel free to cut off the leads of the LEDs, phototransistors, capacitors and resistors so they fit down close to the board when installed.

Make use of the bus strips along each side of the breadboard for your ground and +5V connections. Use the red for power, blue for ground. There should only be one wire for ground and one for +5V coming from your Arduino to the breadboard. All the components that need ground and/or +5V connections on the breadboard should make connections to the bus strips, not wired back to the Arduino.
5 Software

Your software should be designed in a way that makes testing the components of the project easy and efficient. In previous labs we worked on putting all the LCD routines in a separate file and this practice should be continued here. Consider having a separate file for the encoder routines and its ISR, and another one for the serial interface routines. Code to handle the two sensors can either be in separate files or in the main program since there isn’t much code for these. All separate code files must be listed on the OBJECTS line of the Makefile to make sure everything gets linked together properly.

5.1 Improving Your Makefile

In class we discussed how the “make” program uses the data in the “Makefile” to compile the various modules that make up a program. This project may require several source code files, some with accompanying “.h” header files, so the generic Makefile should be modified to describe this. For example, let’s say you have four C files for the project and four header files:

- The main program is in speedtrap.c and has some global variables and functions declared in speedtrap.h
- The LCD routines are in lcd.c with global declarations in lcd.h
- The functions to handle the rotary encoder are in encoder.c with global declarations in encoder.h
- The functions for the serial I/O are in serial.c with global declarations in serial.h

Let’s also say that speedtrap.h is “included” in all the C files, and the header files for the LCD, encoder and serial routines are included in the speedtrap.c file. In this situation, the following lines should be added to the Makefile after the “all: main.hex” and before the “.c.o” line as shown below.

```
all:   main.hex

speedtrap.o: speedtrap.c speedtrap.h lcd.h encoder.h serial.h
lcd.o:      lcd.c lcd.h speedtrap.h
encoder.o:  encoder.c encoder.h speedtrap.h
serial.o:   serial.c serial.h speedtrap.h
```

Adding all the dependencies to the Makefile will make sure that any time a file is edited, all the affected files will be recompiled the next time you type make.

5.2 Measuring Time

For measuring the time between when the start sensor is blocked and when the stop sensor is blocked, it is recommended that you use the 16-bit TIMER1 to generate interrupts every 0.001 seconds. When the start sensor is actuated, start the timer so it is generating interrupts. Each time TIMER1 generates an interrupt (every 1ms) its ISR can increment the count of the number of milliseconds that has elapsed. When the stop sensor is actuated, stop the timer.

5.3 Buzzer

In Lab 7 you worked with producing tones of different frequencies from the buzzer. Those tones were done with code that used delays of half the desired output period between operations to make the output signal go high or low. The result was a squarewave signal at the desired frequency.

The problem with this method is that the program is locked into the delay routines while they measure out the selected delay time. A better way to create the tones is by using a timer to generate interrupts at the desired rate, and when each interrupt occurs the ISR changes the state of the output bit driving the buzzer.
To receive full credit for the buzzer output generation, use one of the two 8-bit timers (TIMER0 or TIMER2) to generate the buzzer signal. If you use the delay routines as in Lab 7, you will still receive partial credit for this task.

5.4 EEPROM Routines

The avr-gcc software includes several routines that you can use for accessing the EEPROM. To use these functions, your program must have this “include” statement at the beginning of the file that uses the routines.

```c
#include <avr/eeprom.h>
```

**eeprom_read_byte** - This function reads one byte from the EEPROM at the address specified and returns the value. It takes one argument, the EEPROM address (0-1023) to read from. For example to read a byte from EEPROM address 100:

```c
x = eeprom_read_byte((void *) 100);
```

**eeprom_update_byte** - This function writes one byte to the EEPROM at the address specified. It takes two arguments, the address to write to and the value of the byte to be stored there. For example to write the byte 0x47 to address 200 in the EEPROM:

```c
eeprom_update_byte((void *) 200, 0x47);
```

Your code should use the above routines to store the maximum speed value in the EEPROM whenever it has been changed. Since the maximum speed is from 1 to 99, this only requires writing a single byte to the EEPROM. You can choose any address in the EEPROM address range (0 to 1023) to store the value. When your program starts up it should read the value from the EEPROM, but it must then test the value to see if it is valid. If the EEPROM has never been programmed, it contains all 0xFF values. If you read the EEPROM data and the value is not in the range 1 to 99, then your program should ignore this number and revert to using a default speed value that is defined in your source code.

**Warning!** The EEPROM on the microcontroller can be written to about 100,000 times and after that it will probably stop working. This limit should be well beyond anything we need for this project but it’s very important that you make sure you don’t have the above EEPROM writing routines in some sort of loop that might go out of control and do 100,000 writes before you realize the program isn’t working right. Your code should only write to the EEPROM when the maximum speed value has changed.

5.5 Serial Interface Routines

The serial data link between two speed trap units uses a simple protocol:

- Data sent using ASCII characters for easier debugging
- Accommodates varying sizes of speed data text strings
- Allows the system to recover from errors such as a partially transmitted or garbled data packet

In many devices that use serial links these features are implemented using relatively complex data structures and interrupt service routines so the processor does not have to spend much time doing polling of the receiver and transmitter. We’ll do it in a simpler manner that should work fine for this application.

The protocol for communicating the measured speed value between two speed trap units will consist of a string of bytes in this format:

- The start of the string is indicated by the ‘@’ character.
• Up to four ASCII digits (‘0’ through ‘9’) representing the speed in **millimeters per second** as an integer value (no fractional speed). For example if the measured speed was 23.7 cm/sec, it should send the three ASCII characters 237. The maximum value that can be sent is 9999. You device should only send the necessary digits. For example if the speed is 7.4 cm/sec, you only need to send 74. You should not send 0074.

• After all characters for the speed has been sent the end of the speed data string is indicated by sending the ‘$’ character.

5.5.1 Transmitting Data

When your software determines that the speed measuring event has been completed, it should call a routine that sends the characters for the speed to the remote unit. The serial link is much slower than the processor so the program has to poll the transmitter to see when it can put the next character to be sent in the transmitter’s data register for sending. The **UCSR0A** register contains a bit (**UDRE0** - USART Data Register Empty) that tells when it’s ready to be given the next character to send. While this bit is a zero, the transmitter is busy sending a previous character and the program must wait for it to become a one. Once the **UDRE0** bit becomes a one, the program can store the next character to be sent in the **UDR0** register.

```c
while (( UCSR0A & (1 << UDRE0 )) == 0) { }
UDR0 = next_character;
```

While your program is waiting for all the characters to be transmitted it should still respond to interrupts from modules with interrupts enabled, but it does not have to reflect any changes on the display until after all the data has been sent and it’s back in the main program loop.

5.5.2 Receiving Data

Receiving the speed data from the remote unit is a bit more complicated since you have no idea when the remote unit will send the data. One simple way to implement this is to have your program check for a received character each time through the main loop. If one has been received, then call a function that waits for the rest of the characters and when complete displays the speed on the LCD. Unfortunately this method of receiving characters has one very bad design flaw in it. If for some reason the string is incomplete, maybe only the first half of the string was sent, the device will sit in the receiver subroutine forever waiting for the rest of the data and the ‘$’ that marks the end of the transmission.

A better solution, and one that should be implemented in your program, is to use interrupts for the received data. Receiver interrupts are enabled by setting the **RXCIE0** bit to a one in the **UCSR0B** register. When a character is received the hardware executes the ISR with the vector name “**USART_RX_vect**”.

For reading the incoming speed data, each time a character is received, an interrupt is generated and the ISR determines what to do with the character. After all the characters have been received, the ISR sets a global variable to indicate that a complete remote speed value has been received and is available. When the main part of the program sees this variable has been set, it gets the value and displays it. By using the interrupts, the program is never stuck waiting for a character that might never come.

It is also important to consider all the possible errors that might occur in the transmission of the date, such as missing start (‘@’) character, missing or corrupted speed characters, missing end (‘$’) character, etc. The software must make sure all of these situations are handled cleanly and don’t leave the device in an inoperable state.

To implement this, use the following variables.

- A 5 byte buffer for storing the data from the remote sensor as it comes in (the 4 data bytes and a ‘\0’ byte, at the end.)

- A global variable to act as a data started flag that tells whether or not the start character (‘@’) has been received indicating data is to follow.

- A variable that tells how many data characters have been received and been stored in the buffer so far. This also tells the ISR where in the buffer it should store the next character.
A global variable to act as a data valid flag to indicate that the '$' has been received and the buffer contains a valid speed string. This variable should be zero while receiving the speed data, and set to one only after receiving the '$' that marks the end of the sequence.

The ISR uses these three variables to properly receive the data.

- If the ISR receives a '@', this indicates the start of a new speed data sequence even if the previous sequence was incomplete. Set the data start variable to a one, and clear buffer count to 0. Also set the the valid data flag to zero to indicate that you now have incomplete data in the buffer.

- If the ISR receives a '$', and the buffer count is greater than zero (meaning the sequence has started) set the valid data flag variable to a one to indicate complete data in the buffer. However if the end transmission character is received but there is no speed data (nothing in the buffer between the '@' and the '$', the flag variable should not be set to a one.

- If a sequence has started and a character in the range of 0 to 9 is received, store it in the next buffer position and increment the buffer count. If after the start of a sequence something other than the number 0 through 9 or the end of transmission marker '$' is received, reset the data started flag to zero to discard what has been received so far. This will set up the ISR to wait for the next transmission. Your code should also make sure there is room in the buffer for the data. If the data tries to overrun the length of the buffer this would imply two transmissions have somehow been corrupted into looking like one, and in this case you should set the data started flag back to zero to discard this transmission.

The main program can check the data valid variable each time through the main loop. When it sees it has been set to a one, it can call a function to convert the speed data from from a string of ASCII characters to a fixed-point binary number (see Sec 5.6). It should probably also clear the data valid variable to a zero so it doesn’t re-read the same data the next time through the loop.

5.6 Using sscanf to Convert Numbers

In Lab 5 you learned how to use the “snprintf” function to convert a binary number into a string of ASCII characters. Now we need to go the other way, from a string of ASCII characaters into single binary fixed-point number. For this we can use the “sscanf” function that is part of the the standard C library.

**Important:** As with using snprintf, in order to use sscanf you must have the following line at the top of the program with the other #include statements.

```
#include <stdio.h>
```

The sscanf function is called in the following manner

```
sscanf(buffer, format, arg1, arg2, arg3, ...);
```

where the arguments are

**buffer** – A char array containing the items to be converted to binary values.

**format** – The heart of the sscanf function is a character string containing formatting codes that tell the function exactly how you want it to convert the characters it finds in input string. More on this below.

**arguments** – After the format argument comes zero or more pointers to where the converted values are to be stored. For every formatting code that appears in the format argument, there must be a corresponding argument containing the a pointer to where to store the converted value.

The format argument tells sscanf how to format the output string and has a vast number of different formatting codes that can be used. The codes all start with a percent sign and for now we will only be working with one of them:

**%d** – Used to format decimal integer numbers. When this appears in the format string, the characters in the input string will be interpreted as representing a decimal integer number, and they will be converted to the corresponding binary value. The result will be stored in the variable that the corresponding argument points to.
The format string must have the same number of formatting codes as there are arguments that follow it in the function call. Each formatting code tells how to convert something in the input string into its corresponding argument. The first code tells how to convert something that is stored where “arg1” points, the second code is for “arg2”, etc.

Example: Assume you have a char array containing the characters representing three numbers. The code below would convert them into the three int variables.

```c
char buf[] = "12 543 865";
int num1, num2, num3;

sscanf(buf, "%d %d %d", &num1, &num2, &num3);
```

The arguments are pointers to where the three values are to be stored by using the form “&num1” which makes the argument a pointer to num1 rather than the value of num1. After the function has executed, the variables “num1”, “num2” and “num3” will contain the binary values 12, 543 and 865.

Important: The “%d” formatting code tells sscanf that the corresponding argument is a pointer to an int (4 byte) variable. When it converts the characters to a binary value it will store it in 4 bytes. If you wish to store a value in a “short” (two bytes), or a “char” (one byte) variable, you must modify the format code. The formatting code “%hd” tells it to store a 2 byte short, and “%hhd” tells it to store a 1 byte char.

Here’s the above example but modified to use three different variable types.

```c
char buf[] = "12 543 865";
char num1;
short num2;
int num3;

sscanf(buf, "%hhd %hd %d", &num1, &num2, &num3);
```

6 Building Your Design

It’s important that you test the hardware and software components individually before expecting them to all work together. Here’s a one possible plan for putting it together and testing it.

1. Build the two sensors by installing the two LEDs and the two phototransistors. Write code to test the sensors. If the first sensor shows the light it blocked write “Start” somewhere on the LCD. If the second sensor is blocked write “Stop” on the LCD.

2. Write the code to initialize the timer to measure 1 millisecond intervals. Confirm that the program is generating interrupts at the correct intervals by having the ISR make an output bit go high and then low. Start the timer and watch the output bit on a scope. You should be able to see narrow pulses 1ms apart.

3. Write code to start the timing when the first sensor is activated, and stop when the second is activated. Write the time in milliseconds to the LCD.

4. Install the timing LED and add code to make it go on when timing starts and go off when it stops.

5. Convert the elapsed time in milliseconds to the speed in cm/sec. This should be done without using any floating point arithmetic. Write the speed value to the LCD after each timing event.

6. Add code to use the rotary encoder to set the maximum speed value. Check that this allows you to adjust the value between 1 and 99 cm/sec.

7. For testing, add code that checks the maximum speed value against the speed measured on your unit. Eventually it will use the speed from a remote unit but for now use the local speed. If the speed is above the maximum value, write something to the LCD that says so.
8. Add the code to sound the buzzer. Use some of your code from the ADC lab to play a tone for a short time (1 second or less). Make the tone play whenever the speed exceeds the maximum speed and confirm that this is working.

9. Write code to store the maximum speed value in the EEPROM, and read the EEPROM value when the program starts. Confirm that this is working by adjusting the speed value and cycling the power on the project. It should start up and display the speed you had set before. Make sure to add code that checks that the speed you read from the EEPROM is a valid value.

10. Install the 74LS125 tri-state buffer. Write code to enable the buffer after the program starts. Check that you can program the Arduino with the serial connections in place. This means the tri-state buffer is doing what it is supposed to do.

11. Write test code that continually sends data out the serial interface and use the oscilloscope to examine the output signal from the D1 port of the Arduino (transmitted data). For 9600 baud, the width of each bit should be 1/9600 sec. or 104 µsec. Check that the width of the bits is correct, and that the voltage levels are correct.

12. Write code to send the speed value in mm/sec out the serial interface in the format specified in Sec. 5.5 and check with the scope that it’s being transmitted after a timing event. Check that the transmitted packet matches the specified protocol with the start and end characters present.

13. Write code to receive the speed in mm/sec. and display the speed in cm/sec on the LCD. This also should be done without using any floating point routines.

14. Change the program so the received speed value is the one being compared with the maximum speed set by the rotary encoder.

15. Do a “loopback” test of your serial interface. Connect the D1 (TX) pin to the input pin on the 74LS125 so you are sending data to yourself. Run a timing event and see if the remote speed is the same as the one you displayed on the LCD as the local speed.

Scope Usage: Connect the scope to the RS-232 signal line and capture a data transfer from the transmitter to the receiver. Show to an instructor and get checked off.

At this point you have all the individual components working so it’s time to check that everything is working as specified. Perform the following checks on your speedtrap.

1. Moving an object so it trips the start and stop sensors to cause a timing event and shows a correct time value. Check that the timing LED was on during the event.

2. Check that the calculated speed is correct for the measured time and the known distance between the two sensors.

3. Check that the rotary encoder can be used to adjust the maximum speed between 1 and 99 cm/sec.

4. After changing the maximum speed, cycle the power on your Arduino and check that the modified maximum speed is displayed when it starts up again. This shows that your data has been stored in the EEPROM correctly and you were able to read it back.

5. Use three wires to hook your speed trap to another one in the class. Between the boards connect ground to ground, transmitted data to received data, and received data to transmitted data. Confirm that both units can send their timing data to the other for displaying.

6. Check that if the received speed from the remote unit is above the maximum speed value the buzzer sounds.