USC Viterbi School of Engineering

Ming Hsieh Department of Electrical Engineering

EE 109L - Introduction to Embedded Systems

Fall 2018 Project Assignment
Thermostat with Remote Sensor

1 Introduction

This semester's class project is to build a thermostat similar to ones that are installed in most homes. It will monitor the room temperature and control heating and air conditioning devices based on the desired temperature set on the thermostat. In addition it will have a connection to a remote device (another student's thermostat) and exchange temperature information with that device.

2 Thermostat Overview

In its simplest form a thermostat measures the temperature in a room and then decides whether to activate a heater to warm the room, or to activate an air conditioner to cool it. A block diagram of our thermostat is shown in Fig. 1 and it will have the following features.

- A temperature sensor that determines the temperature in the room
- An LCD display for showing the current temperature in the room and the high and low temperature threshold settings.
- A control knob for adjusting the high and low temperature settings.
- Two buttons for selecting whether the knob is used to set the high temperature or the low temperature.
- Red and green LEDs to indicate that the heater or air conditioner has been turned on.
- A serial interface (RS-232) to another thermostat. Each unit will send its local temperature to the remote unit and receive the remote unit's temperature.

3 Operation of the Thermostat

The operation of the thermostat is relatively simple.

- The thermostat is always reading the local room temperature from the temperature sensor IC and displaying the value in degrees Fahrenheit on the LCD.
- Setting the high and low temperature thresholds is done by first pressing one of the two buttons to select which threshold to adjust, high or low. The LCD must indicate which setting will be adjusted if the knob is turned.
- The rotary encoder is used set the high and the low temperature thresholds. As the user rotates the knob the temperature threshold should be shown on the LCD display. If they then press the other button they can adjust the other threshold.

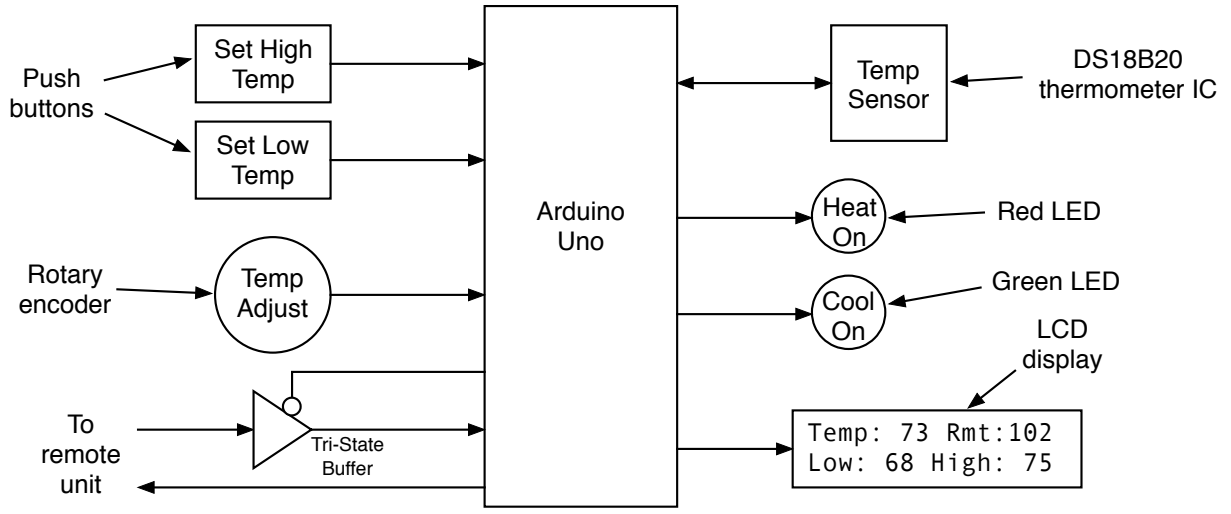

Figure 1: Block diagram of the thermostat

- The thresholds can be set to any value between 40 and 100 degrees. The software must make sure that the setting are never adjusted to where the low threshold is higher than the high threshold.
- If the temperature in the room goes above the high threshold, the thermostat turns on the A/C, which we indicate by lighting up the green LED. Once the temperature is equal to or below the high threshold the A/C goes off. Similarly, if the temperature goes below the low threshold the heater (red LED) should go on, and once the temperature is equal to or above the low threshold the heater goes off. If the temperature is between the thresholds, both the heater and the A/C are off.
- The high and low temperature thresholds are stored in the Arduino's EEPROM non-volatile memory so the values are retained even if the power is turned off. This makes it unnecessary for the user to set the thresholds each time the device is turned on.

3.1 Remote Temperature Interface

The thermostat has a serial connection to a device that sends it the temperature at a remote location, and your thermostat can act as the remote device for another thermostat. Each time the local temperature changes the thermostat transmits the temperature in degrees Fahrenheit over the serial link. The thermostat should display both the local temperature as measured by the temperature sensor on your breadboard, and also the remote temperature as received over the serial link. The protocol used for sending and receiving the temperature data is specified on Section 5.6.

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 two pushbuttons are the same as used in Lab 3. They can be hooked to any of the I/O ports that are available. The LEDs are the same as those used in previous labs and can also be driven from any available I/O port (with a suitable current limiting resistor.) The rotary encoder is the same as in lab 8, and the serial interface is similar to that used in lab 9.

4.1 DS18B20 Temperature Sensor

A new component to this project is the DS18B20 temperature sensor. This is an integrated circuit that connects to the microcontroller using a single wire called a "1-Wire" interface by the manufacturer. A

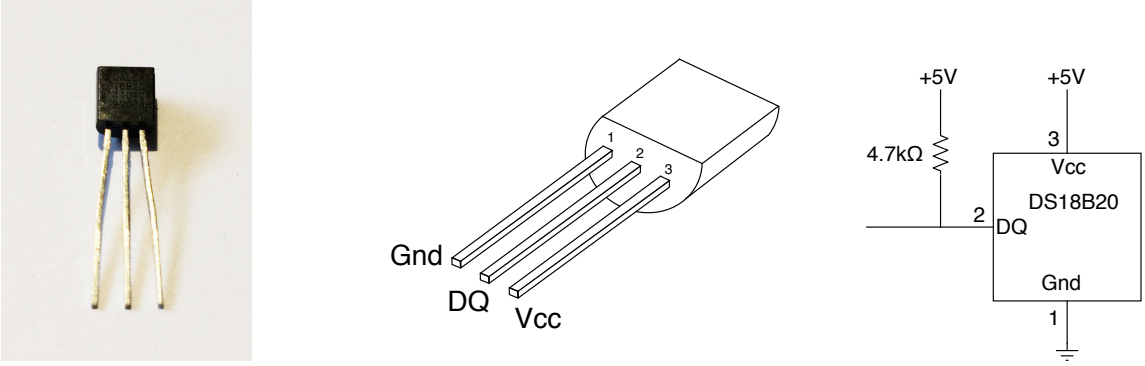

Figure 2: The DS18B20 temperature sensor IC

picture, pin diagram and schematic diagram of the DS18B20 is shown in Fig. 2. The IC needs to have ground and power connections provided on pins 1 and 3 respectively. The Arduino communicates with the DS18B20 over the "DQ" pin which is pin 2. Note from the schematic that the DQ line **must** have an external 4.7kΩ pull-up resistor on it. This is not an option, without it the 1-Wire bus will not operate. You **can not** use the port bit's internal pull-up instead of the external resistor.

4.2 Serial Interface

The serial interface between two thermostats will use an RS-232 link to send the temperature 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 thermostats 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
avrdude: stk500_recv(): programmer is not responding
avrdude: stk500_getsync() attempt 2 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 be using is a 74LS125 that contains four non-inverting buffers (see Fig. 3). Each of the four buffers has 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.

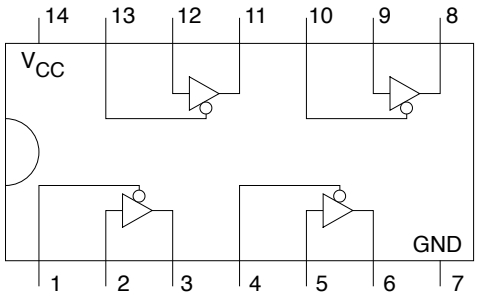

Figure 3: 74LS125 Tri-State buffer

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 devices'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. For example, in this project 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.

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.

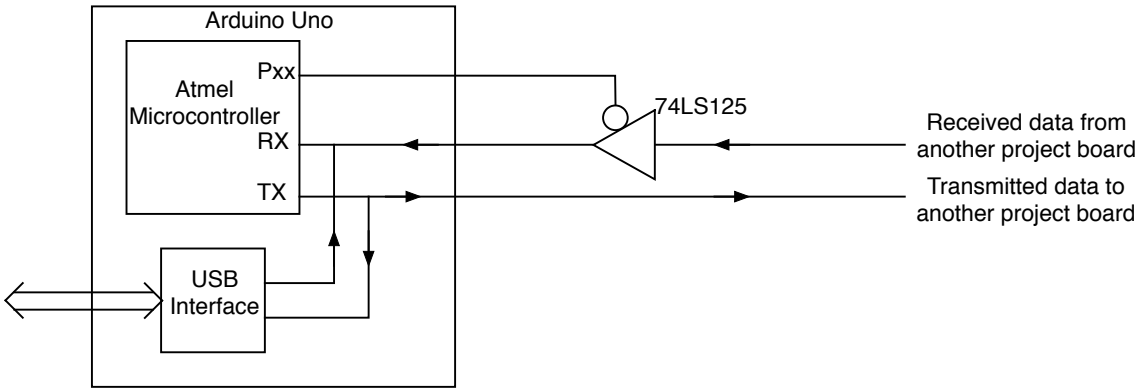

Figure 4: Using the 74LS125 Tri-State buffer to interface to another device

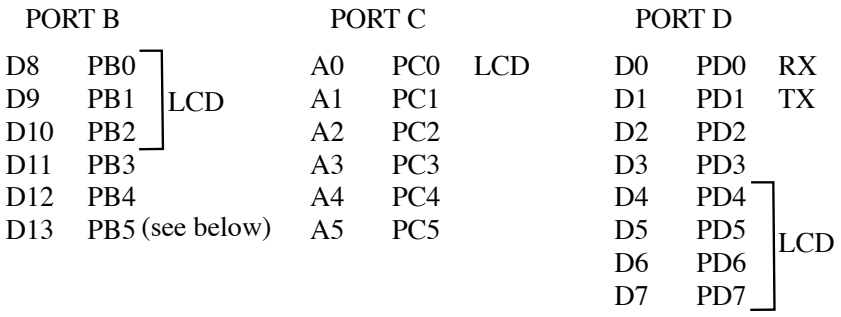

Figure 5: Use of I/O port bits in this project

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.

When working on your design to read input signals from the buttons and the rotary encoder, and send output to the LEDs, 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 a one of the buttons to be pressed, it might be better to have both buttons 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 buttons. 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 buttons, LEDs, rotary encoder, DS18B20 and 74LS125 ICs, and the various 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 about 12 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 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 buttons and the two LEDs 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 thermostat.c and has some global variables and functions declared in thermostat.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 thermostat.h is "included" in all the C files, and the header files for the LCD, encoder and serial routines are included in the thermostat.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

thermostat.o: thermostat.c thermostat.h lcd.h encoder.h serial.h
lcd.o:         lcd.c lcd.h thermostat
encoder.o:     encoder.c encoder.h thermostat.h
serial.o:      serlal.c serial.h thermostat.h

.c.o
```

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 Accessing Global Variables

In this project you will probably need to use global variables that are accessed in multiple sources files. This section describes the recommended way to do that.

In order to use a variable it must be **defined** somewhere. We define variables with lines like

```
char a, b;
int x, y;
```

Global variables can be defined in any source code file but must be defined outside of any function, such as at the start of a file before any of the functions (like "main()"). Variables can only be defined **once** since when a variable is defined, the compiler allocates space for the variable and you can't have a variable stored in multiple places.

If you want to use that global variable in another source code file, the compiler needs to be told about that variable when compiling the other file. You can't put another definition of the variable in the file for the reason given above. Instead we use a variable **declaration**. The declaration of a global variable (one that is defined in a different file) is done using the "extern" keyword. The "extern" declaration tells the compiler the name of the variable and what type it is but does not cause any space to be allocated when doing the compilation of the file.

For example, let's say the global variable "result" is accessed in two files, project.c and stuff.c, and you decide to define it in the project.c file. The project.c file would contain the line defining the variable

```
int result;
```

and the stuff.c file would need to declare the variable with the line

```
extern int result;
```

in order to use the variable in that file. If the "extern" keyword was omitted, both files would compile correctly, but when they are linked together to create one executable, you would get an error about "result" being multiply defined.

If you have global variables that need to be accessed in multiple files it's recommended that you put the declarations in a ".h" file that can be included in all the places where they may be needed. For the example above, create a "project.h" file that contains the line

```
extern int result;
```

and then in both project.c and stuff.c add the line

```
#include "project.h"
```

It doesn't cause problems to have the declaration of a variable, like from an ".h" file, included in the file that also contains the definition of the same variable. The compiler knows how to handle this correctly.

5.3 DS18B20 Routines

From the class web site you can download the file ds18b20.c that contains software to communicate with the DS18B20 temperature sensor over the 1-Wire bus. **The routines in this file assume the DS18B20 is connected to Port C, pin 5.** If you decide to connect it to some other I/O pin, the routines in ds18b20.c will have to be modified to match your configuration. We've provided some of the routines that do the work necessary for using the DS18B20. The two top-level routines that make up the API (Application Programming Interface) to the sensor are these.

ds_init - Initializes the 1-Wire bus and the DS18B20.

ds_temp - Initiates a temperature conversion, waits for it to complete and reads **two bytes of data** from the DS18B20 containing the temperature in degrees Celsius.

Below is a sample of some code that uses these routines to read the temperature data.

```
void ds_init(void);
void ds_temp(unsigned char *);

main()
{
    unsigned char t[2];

    ds_init();       // Initialize the DS18B20

    while (1) {
        ds_temp(t);  // Read the temperature data
        /*
          Process the values returned in t[0]
          and t[1] to find the temperature.
        */
    }
}
```

The ds_init routine only has to be called once at the start of the program, not each time you want to read the temperature.

When ds_temp is called, the temperature data is returned in the two element unsigned char array as shown in Fig. 6. Element 0 of the array contains the eight least significant bits. Bits 3, 2, 1 and 0 are the fractional part of the temperature value. Element 1 of the array contains the four most significant bits of the temperature value. The result is essentially a 12-bit 2's complement number stored in two 8-bit bytes with the sign extended to fill it out to a 16-bit 2's complement number. The two bytes together represent the temperature in degrees Celsius times 16. For example if the temperature is 27.5 degrees Celsius, the two

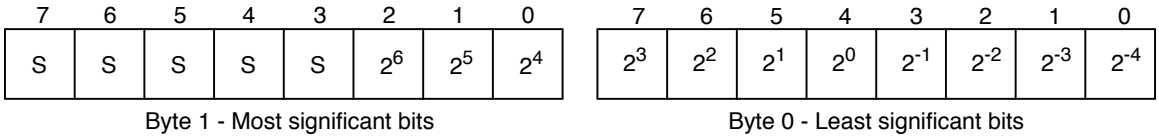

Figure 6: Temperature values are returned in a two element array

values in the array elements are `0x01` and `0xB8`. Taken as a 16 bit value, `0x01B8` is equal to 440_{10}, and this is equal to 27.5×16.

The two API routines described above make use of several other routines in `ds18b20.c` that are not intended to be seen or used by a user of the API. These routines to the actual work of communicating with the DS18B20 to acquire the temperature data.

ds_reset - Send a reset pulse to the DS18B20. This routine is provided for you.

ds_write1bit - Send a 1 bit to the DS18B20. This routine is provided for you.

ds_write0bit - Send a 0 bit to the DS18B20. This routine you will have to write.

ds_readbit - Read a single bit from the DS18B20. This routine you will have to write.

ds_writebyte - Send one 8-bit byte to the DS18B20. This routine is provided for you.

ds_readbyte - Read one 8-bit byte from the DS18B20. This routine is provided for you.

The bus master (your Aruduino) sends bits to, and receives bits from, the slave device (the DS18B20) by signaling over the 1-Wire bus. Unlike RS-232 where each bit has a specified duration, the signaling on a 1-Wire bus is based on how long the bus master holds the bus in one state or the other. For example `ds_write1bit` sends a 1 bit to the DS18B20 by holding the bus low for $1\mu s$ to $15\mu s$ (our code uses $2\mu s$), then making the bus high for the reset of the $60\mu s$ time slot.

The code for most of the above routines is provided in the `ds18b20.c` file but you will have to write the code for the `ds_write0bit` and `ds_readbit` routines. The DS18B20 datasheet describes the timing for writing ones and zeros, and for reading a bit from the DS18B20 on pages 15-17. You should refer to these pages when completing the code for the these routines.

Hint: Compare the code in the `write1bit` routine to the diagram in the datasheet on page 16 to see how the C code implements the required time periods when the output should be a one or a zero. Use that example to write the `ds_write0bit` and `ds_readbit` routines.

5.4 Temperature Calculations

The DS18B20 converts the temperature into a two's complement 12-bit value (8-bits for the integer portion and a 4-bit fractional portion) representing the temperature in degrees Celsius, and stores these in two bytes that can be transferred from the DS18B20 by your program. The DS18B20 datasheet at the bottom of page 3 and the top of page 4 describes the format of how the temperature data is stored, and includes several examples of how values are represented.

Your program should convert these two bytes to an Fahrenheit value. The Fahrenheit value can be an integer value with no fractional degree included, however the bits for a fractional part of a degree Celsius must be properly factored into the conversion.

Hint: The `ds_temp` routine returns the temperature in two bytes. Combine these two 8-bit values into a single 16-bit signed variable before doing any operations to convert it to Fahrenheit.

Converting from Celsius temperature to Fahrenheit is usually done with the formula

$$F = \frac{9}{5}C + 32$$

however in this project **all temperature calculations must be done in integer (fixed point) arithmetic.** Do **NOT** use any floating point numbers (`float` or `double`) or floating point arithmetic.

When using integer arithmetic, and variables with a limited numeric range, it's easy to go wrong. For example if we do the following

```
unsigned char c, f;
f = (9 / 5) * c + 32;
```

the integer division of $\frac{9}{5}$ will be 1 and give result of $F = C + 32$. On the other hand, if we do it as

```
unsigned char c, f;
f = (9 * c) /  5 + 32;
```

The product "`9 * c`" will probably give a result outside the range of the unsigned char variable and result in an overflow. To get the correct answer, you must do the conversion using variables of the right size and do the calculations in the proper order.

5.5 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.

```
#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:

```
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:

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

Your code should use the above routines to store the high and low temperature thresholds in the EEPROM whenever it has been changed. You can assume the thresholds will always be in the range of 40 to 100 degrees, and these numbers can be stored in an 8-bit "char" variable and only requires writing a single byte to the EEPROM for each value. You can choose any addresses in the EEPROM address range (0 to 1023) to store the values. 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 40 to 100, then your program should ignore this number and revert to using a default threshold 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 a threshold value has changed.

5.6 Serial Interface Routines

The serial data link between two thermostats uses a simple protocol:

- Data sent using ASCII characters for easier debugging
- Accommodates varying sizes of temperature 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 temperature between two thermostats will consist of a string of up to six bytes in this order:

- The start of the string is indicated by the `'@'` character.
- A plus or minus sign ('`+`' or '`-`').
- Up to three ASCII digits ('0' through '9') representing the temperature in **degrees Fahrenheit** as an integer value (no fractional degrees). You device should only send the necessary digits. For example if the temperature is 68 degrees, you only need to send `68`. You should not send `068`.
- After all characters for the temperature have been sent the end of the temperature data string is indicated by sending the `'$'` character.

5.6.1 Transmitting Data

When your software determines that the local temperature has changed, it should call a routine that sends the characters for the temperature data 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.

```
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.

5.6.2 Receiving Data

Receiving the temperature 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 temperature 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 temperature 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 temperature 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 temperature 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 handle possible errors in the received data, the thermostat should operate on the principle that if an error of **any kind** is detected in the data being received, it discards the current remote temperature data that it was in the midst of receiving and goes back to waiting for the next temperature data to start coming in. It does not have to notify the main program that an error was encountered, or try to correct the error, or guess what the correct data was. Simply throw it away and go back to waiting for a new data packet.

To implement this, use the following variables to properly receive the data:

- 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 `'$'` has been received and the buffer contains a valid temperature string. This variable should be zero while receiving the temperature data, and set to one only after receiving the `'$'` that marks the end of the sequence.

If the ISR receives a `'@'`, this indicates the start of a new temperature 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. This sets up the ISR to wait for the next transmission.

For the next characters that arrive, your code must check to see if they are valid.

- If the ISR receives a '`+`' or '`-`', and it's not first character after the `'@'`, something has gone wrong so reset the data started flag to zero to discard this packet.
- After that, if something other than the number 0 through 9 or the end of transmission marker `'$'` is received, reset the data started flag to zero to discard what has been received so far. This will set up the ISR to wait for the next transmission.
- If a sequence has started and a correct character is received, store it in the next buffer position and increment the buffer count. Your code should also make sure there is room in the buffer for the data. If the data tries to overrun the length of the buffer this would imply two transmissions have somehow been corrupted into looking like one, and in this case you should set the data started flag back to zero to discard this transmission.
- If the ISR receives a `'$'`, 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 temperature data (nothing in the buffer between the `'@'` and the `'$'`, the flag variable should not be set to a one.

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 temperature data from from a string of ASCII characters to a fixed-point binary number (see Sec 5.7). 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.

5.7 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.

```
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** variable. When it converts the characters to a binary value it will store it in the correct number of bytes for that variable type. If you wish to store a value in a "short", or a "char" 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.

```
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. Install the LCD shield and write test code to confirm that you can write messages on both lines of the display. Use your file of LCD routines from the previous labs to implement this.

2. Install the two buttons. Write code to test reading the button presses. If the "High" button was pressed write "High" somewhere on the LCD. If the "Low button was pressed write "Low" on the LCD. Note that since we have separate buttons for the two modes we only need to detect that the button was pressed. We don't have to deal with debouncing or having delays when sensing that a button is down.

3. Install the rotary encoder. Write code to read the state changes and change the low and high temperature thresholds depending on which button was pressed last. Use your encoder software from Lab 7 that used the Pin Change Interrupts to implement this.

4. Install the LEDs. Write test code to turn these on or off based on the low and high settings. For example, turn the green LED on if the high setting is above 80 and off if it's below. Similarly turn the red LED on if the low setting is below 60 and off if above. This is not the way the thermostat will eventually function when operating but here we are just testing the LEDs by using the rotary encoder.

5. Edit the `ds18b20.c` file to finish the two routines that were left for you to complete.

6. Install the DS18B20 thermometer IC. Write code that uses the provided routines to initialize the IC and then read the temperature from it. At first you can display the two bytes of temperature data as four hex digits. Put this code in a loop so it continuously read the temperature and displays it on the LCD. Is the displayed value about what you would expect to see? Try heating up the DS18B20 but pressing your figure on it, or cool it be blowing on it, and confirm that the temperature changes.

7. Once you know the sensor is returning valid data, implement the code to convert the Celsius temperature to Fahrenheit.

8. Install the 74LS125 tri-state buffer. Write code to enable the 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.

9. 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 104μsec.

10. Write code to send the temperature value out the serial interface in the format specified in Sec. 5.6 and check with the scope that it's being transmitted after the temperature changes. Check that the transmitted packet matches the specified protocol with the start and end characters present.

11. Write code to receive the temperature and display it on the LCD. This also should be done without using any floating point routines.

12. 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. Try heating and cooling your sensor and see if the remote temperature is the same as the one you displayed on the LCD as the local temperature.

    Scope Usage: Connect the scope to the serial communications line and capture a data transfer from the transmitter to the receiver. Show to a instructor and get checked off.

At this point you have all the individual components working so it's time to integrate everything (buttons, encoder, DS18B20, LEDs, LCD) together so it operates as specified. Once that is done, perform the following checks on your thermostat.

1. The high and low setting button properly switch it between setting the high and low thresholds, and the selection is indicated on the LCD.

2. Check that the rotary encoder can be used to adjust both the high and low thresholds. Confirm that the low setting is prevented from being greater than the high setting, and vice versa.

3. Set the high and low thresholds to being just above and just below the current indicated temperature. Change the temperature by heating and cooling the DS18B20 and confirm that the heating and A/C LEDs go on and off as expected.

4. Use three wires to hook your thermostat 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 thermostats are displaying the correct temperature from the other one. Heat and cool both DS18B20's and confirm that the displayed remote temperatures update properly and are correct.