USC Viterbi School of Engineering

Ming Hsieh Department of Electrical Engineering

EE 109L - Introduction to Embedded Systems

Fall 2014 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 should be relatively simple. At all times the thermostat is periodically reading the room temperature from the thermometer IC and displaying the value in degrees Fahrenheit on the LCD.

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 control knob is used set both the high and the low temperature setting. Since there is only one control knob, the two pushbuttons are used to select which temperature setting the knob is now controlling. The user sets the high and low temperatures by first pressing one of the two button and the LCD should confirm in some way to the user which temperature threshold is being adjusted. As the user rotates the encoder the temperature setting should be shown on the LCD display. If they then press the other button they can adjust the other threshold. The software should make sure that the low limit is always less than or equal to the high limit.

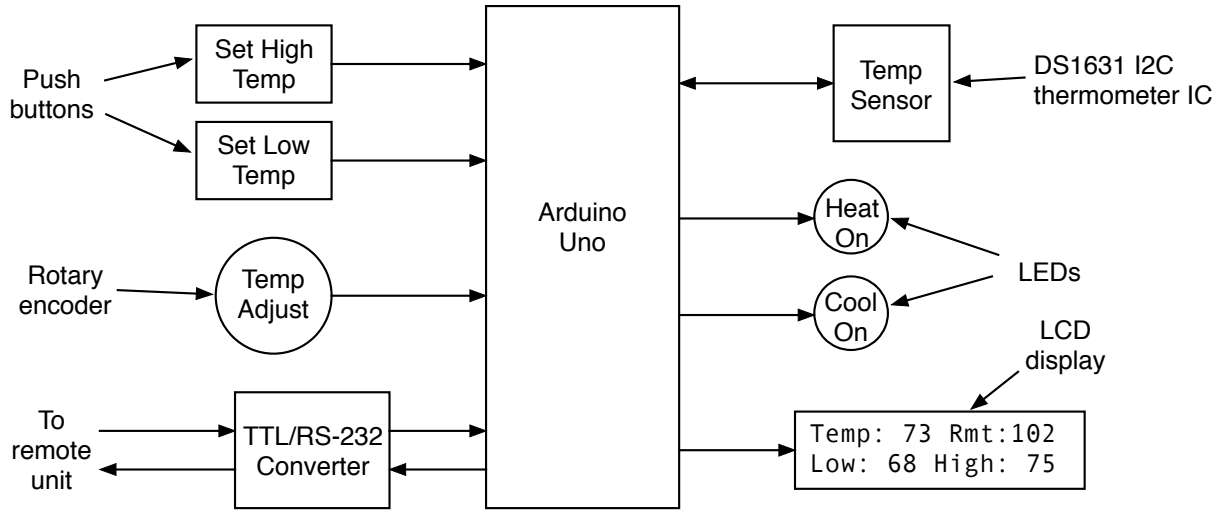

Figure 1: Block diagram of the thermostat

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 like the remote device for another thermostat. Each time the local temperature in degrees Fahrenheit changes the thermostat transmits the temperature over the serial link as a string of four ASCII characters. The format of the string is the first character is either a plus or minus sign, and this is followed by three digits. For example, if the temperature is 77°, then the transmitted string is "`+077`".

The thermostat should display both the local temperature as measured by the DS1631 sensor on your protoboard, and also the remote temperature as received over the serial link.

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 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 labs 7 and 8, and the LCD was used in labs 5, 6 and 7.

4.1 DS1631 Temperature Sensor

A new component to this project is the DS1631 temperature sensor. This is an integrated circuit that connects to the microcontroller using the I^2C 2-wire serial bus as described in class. A picture, pin diagram and schematic diagram of the DS1631 is shown in Fig. 2. The IC needs to have power and ground connections provided on pins 8 and 4 respectively. In addition, pins 5, 6 and 7 must also be grounded. These pins determine part of the address the DS1631 has on the I^2C bus and grounding all three give it an address that is consistent with the software routines provided for the project. The I^2C data and clock signals from the Arduino are connected to pins 1 and 2. Note from the schematic that both the data and clock lines **must** have 10kΩ pull-up resistors on them. This is not an option, without them the I^2C bus will not operate.

The ATmega328P on the Arduino has a hardware module that implements an I^2C interface and we will be using that. Students will be provided with two files "`ds1631.o`" and "`ds1631.h`" that contain a library of I^2C routines that will allow their program to communicate with the DS1631 to configure it and then read the temperature.

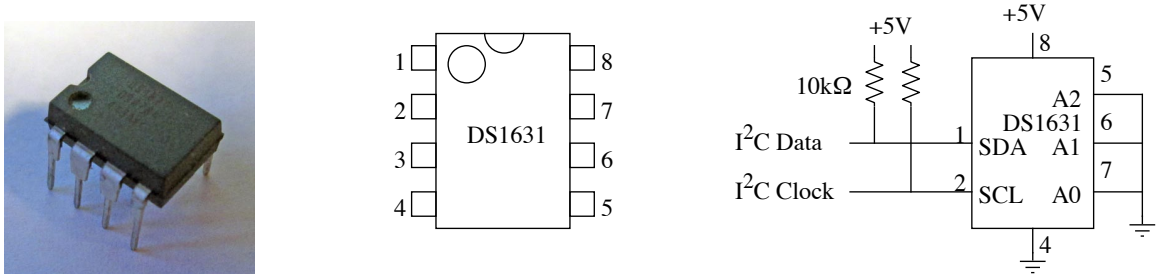

Figure 2: The DS1631 temperature sensor IC

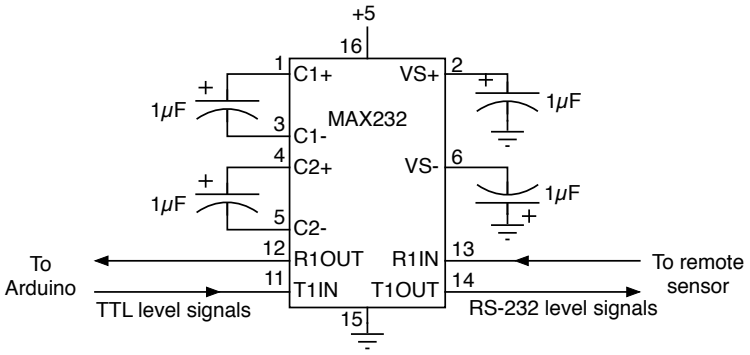

Figure 3: Using a MAX232 chip to interface between TTL and RS-232 signals

4.2 Serial Interface

The other new component in this lab is the serial interface to the remote sensor. This will use an RS-232 link to send the temperature data back and forth 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. These have to be converted to the RS-232 voltages levels in order for it to be compatible with another RS-232 device. A MAX232 integrated circuit converts between TTL and RS-232 signal levels. As shown in Fig. 3 the MAX232 also requires four small capacitors in order to operate. The capacitors are polarized meaning that it makes a difference which end is connected to positive or negative voltage. When installing the capacitors next to the MAX232, look for a band of negative signs on the side of the capacitor to indicate which wire coming out the bottom is for the negative connection.

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

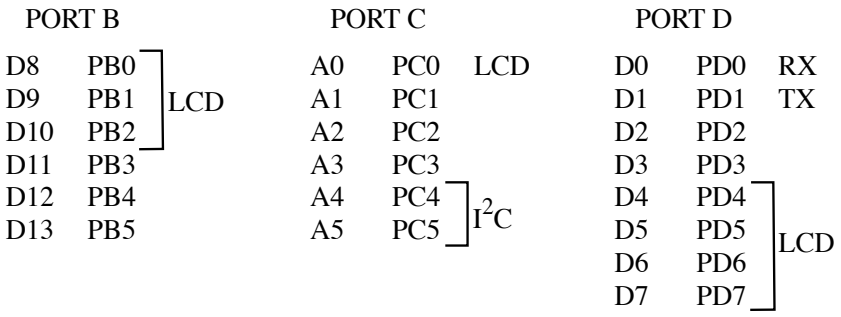

Figure 4: Use of I/O port bits in this project

4.3 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 both the I^2C and the serial communications modules, both of which require using specific I/O lines. In addition, the LCD shield requires using certain port bits. Fig. 4 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.

I^2C - PC4 and PC5 are used for the I^2C data and clock, respectively.

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

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.

4.4 Construction Tips

The buttons, LEDs, rotary encoder, DS1631 IC and the MAX232 IC and it's four capacitors should all be mounted on your protoboard. 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 protoboard so don't make matters worse by having a rat's nest of other wires running around on the protoboard. Feel free to cut off the leads of the LEDs, capacitors and resistors so they fit down close to the board when installed.

Make use of the bus strips along each side of the protoboard 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 protoboard. All the components that need ground and/or $+5V$ connections on the protoboard 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 Lab 7 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 DS1631 Routines

The `ds1631.o` file contains software to communicate with the DS1631 temperature sensor. To avoid getting bogged down in the I^2C code we've provided three routines that do all the work necessary for using the DS1631.

ds1631_init - Initializes the I^2C bus and the DS1631.

ds1631_conv - Sets the DS1631 to do continuous temperature conversions.

ds1631_temp - Read two bytes from the DS1631 containing the temperature in degrees Celsius.

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

```
void ds1631_init(void);
void ds1631_conv(void);
void ds1631_temp(unsigned char *);

main()
{
    unsigned char t[2];

    ds1631_init();       // Initialize the DS1631

    ds1631_conv();       // Set the DS1631 to do conversions

    while (1) {
        ds1631_temp(t);  // Read the temperature data
        /*
          Process the values returned in t[0]
          and t[1] to find the temperature.
        */
    }
}
```

The `ds1631_init` and `ds1631_conv` routines only have to be called once at the star of the program, not each time you want to read the temperature.

When `ds1631_temp` is called, the temperature data is returned in the two element `unsigned char` array. The first array element contains the temperature in degrees Celsius (Centigrade). The second element is either `0x00` or `0x80`, and a non-zero value indicates an extra half degree of temperature. For example if the temperature is 24.0 degrees Celsius, the two values in the array elements are `0x18` and `0x00` (`0x18` = 24_{10}). If the temperature is 24.5 degrees Celsius, the values are `0x18` and `0x80`.

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

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

However 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 of 9 and `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.

5.2 Serial Interface Routines

In many devices that use serial links these are implemented using relatively complex data structures and interrupts so the processor does not have to spend much time doing polling of the receiver and transmitter. We'll do it in a much simpler manner that should work fine for this application. When your software determines that the local temperature has changed, it should call a routine that sends the four characters as described in Section 3.1. As the characters are being sent, this will require doing polling of the transmitter to see when the next character can be put in the transmitter's buffer for sending. While this is going on your program may still respond to Pin Change Interrupts from the encoder or buttons but does not have to reflect these changes on the display until after all the data has been sent and it's back in the main program loop.

For receiving temperature data from a remote device, your program should check the receiver each time through the main program loop to see if a character has been received. This can be done by examining the `RXC0` bit in the `UCSR0A` register.

```
if (UCSR0A & (1 << RXC0)) {
    // character has been received
}
```

If `RXC0` is a one, a character has been received and is in the `UDR0` register waiting to be read. The program should go into a subroutine that reads the character and checks to make sure that received character is a `+` or `-`, the first character of a correctly formatted temperature data string. If it isn't then the subroutine should return without waiting for any more characters. If the character is correct then the subroutine should go into a loop reading the next three characters (the digits) to finish receiving the string. Note that the received string is not exactly what should be displayed on the LCD. For example If the received string is "`+068`", the LCD should display "`68`".

This method of receiving characters has one very bad design flaw in it. If the string is incomplete, maybe only two digits are sent, the thermostat will sit in the receiver subroutine forever waiting for the third digit. Can you recommend (and maybe implement) a way to prevent the thermostat from getting locked up like this?

6 Testing 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 7B 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. Install the DS1631 thermometer IC. Write code that uses the provided I^2C routines to configure the IC and then read the temperature from it. 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 DS1631 but pressing your figure on it, or cool it be blowing on it, and confirm that the temperature changes.

6. Install the MAX232 RS-232 interface IC and the four capacitors that go with it. Write test code that continually sends data out the serial interface and use the oscilloscope to examine the output signal from the MAX232. For 9600 baud, the width of each bit should be 1/9600 sec. or 104nsec. Check that the width of the bits is correct, and that the voltage levels are correct.

At this point you have all the individual components working so it's time to integrate everything (buttons, encoder, DS1631, 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 .

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 DS1631 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 DS1631's and confirm that the displayed remote temperatures update properly and are correct.