USC Viterbi School of Engineering

Ming Hsieh Department of Electrical Engineering

EE 109L - Introduction to Embedded Systems

Spring 2017 Project Assignment
Alarm Clock with Indoor/Outdoor Thermometer

1 Introduction

This semester's class project is to build an digital alarm clock that includes a thermometer for displaying both the local temperature and a remote temperature. Like any alarm clock, it displays the current time and can be set to sound an alarm at a selected time. The remote temperature can be assumed to have come from an outdoor sensor but in our case we will use another student's alarm clock project as the remote sensor. The clock uses the two temperature readings to show an indication as to whether it cooler or warmer outside than it is inside.

2 Alarm Clock Overview

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

- An LCD display to show the current time, the alarm time, the local temperature and the remote temperature.
- A button for selecting the time field to adjust.
- A control knob for adjusting the time and alarm settings.
- A temperature sensor that determines the local temperature.
- A serial interface (RS-232) to another alarm clock. Each unit will send its local temperature to the remote unit and receive the remote unit's temperature.
- Red and green LEDs to indicate if the remote temperature is above or below the local temperature.
- A "snooze" button for delaying the alarm.

3 Operation of the Alarm Clock

The following is a list of the feature of the alarm clock and how they operate.

- In normal operation, the clock is advancing the displayed time count every second, much the same way your stopwatch lab operated. It shows four fields:
  - Day of the week
  - Hours
  - Minutes
  - Seconds

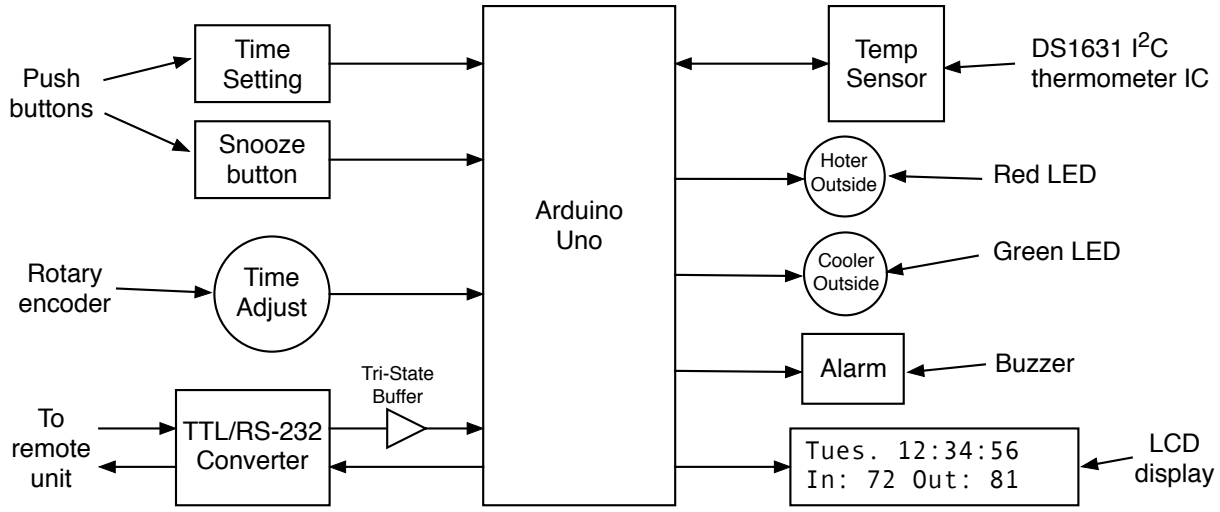
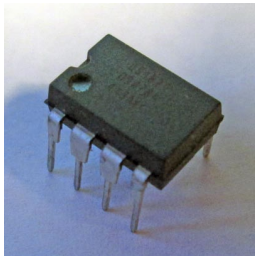
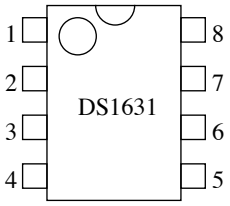


Figure 1: Block diagram of the alarm clock

To simply things slightly we will use 24-hour time format (00:00:00 to 23:59:59) rather than the more common 12 hour format which then requires an AM/PM indicator.

- Every time the minute value increments, the time should be copied to the Arduino's EEPROM memory, and whenever the clock is started up, the initial time value should be loaded from EEPROM. This way it doesn't start at "00:00:00" every time the program is restarted.
- When the clock time matches the alarm time, the buzzer should sound for the next 5 seconds and then go silent.
- Each time the "Time Setting" button is pressed the display changes to one of five time settings that can be adjusted: the day of the week, the hour, the minutes, the alarm hour, and the alarm minutes. The next press of the setting button cycles it back to the normal clock display.
- The rotary encoder is used adjust the time settings when in one of the five time adjustment modes. As the user rotates the knob the setting being adjusted should be shown on the LCD display. When the time setting button is pressed to move to the next field or to exit the time setting mode, this setting is stored.
- The clock is always reading the local room temperature from the thermometer IC and displaying the value in degrees Fahrenheit on the LCD. Whenever the temperature changes, it sends this new temperature out the serial interface using a protocol specified in Section 5.5
- The clock receives a remote temperatures over the serial interface from a remote device, like another one of our alarm clocks. Whenever a remote temperature is received it is also displayed on the LCD and remains until an updated temperature is received.
- If the remote temperature is above the local temperature, the red LED goes on to show that it's currently hotter outside then inside. Conversely, if the remote temperature is below the local temperature the green LED goes on. If the two temperatures are the same, both LEDs should be off.
- While the alarm is sounding, if the "snooze" button is pressed, the alarm shuts off but then sounds again in 30 seconds for 5 seconds. This process can be repeated.





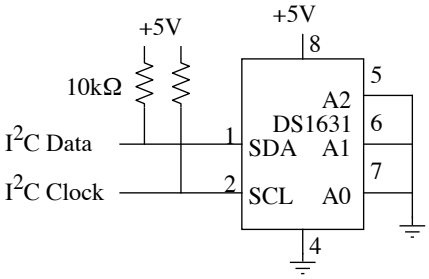


Figure 2: The DS1631 temperature sensor IC

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 labs 9 and 10, and the LCD was used in labs 5 through 9.

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.

The I^2C bus consists of two wires, one for clock and one for data. The I^2C module of the Arduino's microcontroller uses Port C, bit 4 for the data and Port C, bit 5 for the clock. In the labs that used the rotary encoder we were using these I/O bits for the encoder inputs. Since the I^2C module has to use PC4 and PC5, and this can't be changed, **you will have to use two other I/O bits for your encoder, and change your software routines accordingly.**

4.2 Serial Interface

The other new component in this lab is the serial interface to the remote alarm clock. This 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. 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.

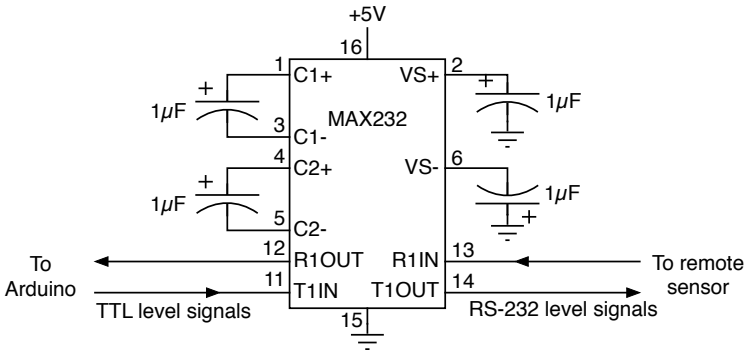


Figure 3: Using a MAX232 chip to interface between TTL and RS-232 signals

For all the alarm clocks 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

If the received data output of the MAX232 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 MAX232 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 MAX232 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. 4). 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. 5, the received data from the MAX232 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 MAX232's received data will pass through the buffer to the RX serial port.

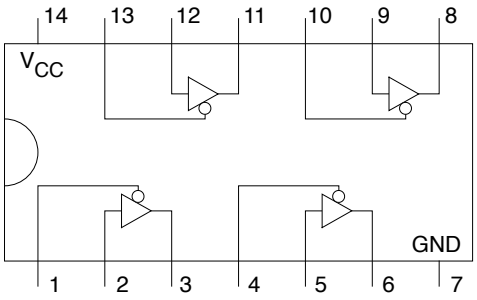


Figure 4: 74LS125 Tri-State buffer

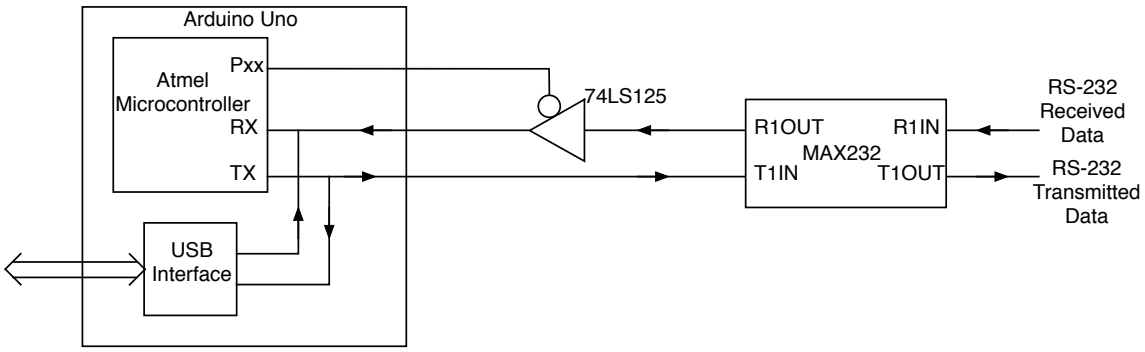


Figure 5: Using the 74LS125 Tri-State buffer to interface the MAX232 to the Arduino

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 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. 6 shows which I/O port bits are allocated for various purposes.

PORT B			PORT C			PORT D		
D8	PB0	LCD	A0	PC0	LCD	D0	PD0	RX
D9	PB1		A1	PC1		D1	PD1	TX
D10	PB2		A2	PC2		D2	PD2	
D11	PB3		A3	PC3		D3	PD3	
D12	PB4		A4	PC4	I^2C	D4	PD4	LCD
D13	PB5 (see below)		A5	PC5		D5	PD5	
						D6	PD6	
						D7	PD7	

Figure 6: 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.

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.

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 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 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 Construction Tips

The buttons, LEDs, buzzer, rotary encoder, DS1631, MAX232 and 74LS125 ICs, 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 about 14 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, 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 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.

5.1 Improving Your Makefile

For a larger project like this, the "`Makefile`" is a key part of putting it all together properly. 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 and all the the separate code files must be listed on the `OBJECTS` line of the `Makefile` to make sure everything gets linked together properly.

Some of the source code files may also have a associated header or "`.h`" file. To properly compile everything the the generic `Makefile` should be modified to describe the relationship between all the files. For example, let's say you have four C files for the project and four header files. **Note: This is just an example. You may not actually have all these files in your project.**

- The main program is in `clock.c` and has some global variables and function prototypes declared in `clock.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 `clock.h` is "included" in all the C files, and the header files for the LCD, encoder and serial routines are included in the `clock.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

clock.o:     clock.c clock.h lcd.h encoder.h serial.h
lcd.o:       lcd.c lcd.h clock.h
encoder.o:   encoder.c encoder.h clock.h
serial.o:    serlal.c serial.h clock.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 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 start of the program, not each time you want to read the temperature.

WARNING! None of the `ds1631` routines described above will work unless the DS1631 is properly installed on the breadboard (including the pull-up resistors) and the I^2C data and clock signals are connected to the Arduino. If the DS1631 is not installed correctly, calling any of these routines will probably cause the program to hang and do nothing while it waits for the I^2C device to respond.

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`.

5.3 Temperature Calculations

Your program should convert the two bytes that contain the temperature in degrees Celsius to a Fahrenheit value. The second byte that contains a value to indicate an extra half degree Celsius as described above should be properly factored into the conversion. **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. 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.

5.4 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.

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

eeprom_read_byte - This function reads one byte from the EEPROM at the address specifed 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 day, hours and minutes values in the EEPROM whenever these have changed during normal clock operation. Since all of these values each fit in a single byte or "unsigned char" variable, this only requires writing three bytes 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 values from the EEPROM, but it must then test the values to see if they are valid. If the EEPROM has never been programmed, it contains all 0xFF values. If you read the EEPROM data and any of the values are not in the proper ranges, then your program should ignore these numbers and revert to using default time values 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 the minute value changes.

5.5 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 simpler manner that should work fine for this application.

The temperature value will be sent between alarm clocks using a string of bytes in this format:

- The start of the string is indicated by the `'#'` characcter.
- A `'+'` or `'-'` sign.
- Up to three ASCII digits ('0' through '9') representing the integer temperature in degrees Fahrenheit. (no fractional temperature). For example if the measured temperature was 71.4 degrees, it should send the ASCII characters `71`. You only need to send the necessary digits. For example if the temperature is 81 degrees, you only need to send `81`. You don't have to send `081`.
- After all characters for the temperature have been sent the end of the temperature data string is indicated by sending the `'$'` character.

5.5.1 Transmitting Data

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

5.5.2 Receiving Data

Receiving the temperature data from the remote unit is a bit more complicated since you have no idea when it will be sending 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 ('+' or '-') 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 implement this, use the following variables.

- A 5 byte buffer for storing the data from the remote sensor as it comes in (the sign, 3 digits, 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.

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

- 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.
- 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.
- 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 temperature 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 6 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** (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.

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

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

6 Building Your Project

We recommend building your project in a modular manner that allows you get one part of it working before moving on to the next. When the project is graded it's better to have only half of the features working correctly than to have all the feature installed but none of them working. It's also important that you test the hardware and software components individually before expecting them to all work together. Here's 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. Write code to use TIMER1 (the 16-bit time) to implement the clock. It should show the four fields (day-of-the-week, hours, minutes and seconds) and the displayed values should advance at the proper rate.

3. Add code to copy the clock time to the EEPROM whenever the minute changes. Add other code to read the clock values from the EEPROM when the program starts. You should be able to turn the clock off and back on, and the time will pick up where it left off.

4. Install the "Time Setting" button and write code so that each time the button is pressed the clock changes to a different state, one for each of the five time values that has be adjusted, and one for the running clock display. The clock should cycle through all six states as the button is pressed. For the states used for changing time values, display a message about which field is being shown, and the current value.

5. Install the rotary encoder. You can use some of the code from Lab 9 that used the Pin Change Interrupts to handle the rotary encoder but it will probably have to be modified. Add code to use the rotary encoder to change the numbers for each of the five clock and alarm settings. Confirm that each time field can be changed over it's full range, but not beyond the limits of the value. Once you have changed one or more numbers, when you return to the clock display it should reflect these changes.

6. Install the buzzer and make it sound for 5 seconds when the clock time and the alarm time match. In Lab 4 you provided a signal to the buzzer by turning a PORT bit on and off at a selected rate by using a delay function. A better way is to use a timer to generate interrupts each time the output signal is to be changed. Try using the 8-bit TIMER0 for this. When the alarm is to be sounded, turn TIMER0 on (set its prescaler to the correct value). Five seconds later turn it off (set the prescaler bits to all zeros). In the TIMER0 ISR add code to change the state of the output pin whenever the ISR is called.

7. 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 reads the temperature and displays it on the LCD. Is the displayed value about what you would expect to see? Try heating up the DS1631 by pressing your figure on it, or cool it by blowing on it, and confirm that the temperature changes.

   Scope Usage: Connect the scope to the I^2C clock and data lines and capture an I^2C data transfer from the DS1631 to the microcontroller. Show to a instructor and get checked off.

8. Install the MAX232 RS-232 interface IC, the four capacitors that go with it, and the 74LS125 tri-state buffer. Write code for sending data out the serial interface and send some test temperature values. 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 104μsec. Check that the width of the bits is correct, and that the voltage levels are correct. Check that the format of the temperature string is correct according to the protocol specification in Sec. 5.5.

9. Add code for receiving the remote temperature data as discussed in Sec. 5.5.2. You can always user your LCD to print out the contents of the received data buffer if needed. Convert the received temperature to something that can be displayed and show it on the LCD.

10. Do a "loopback" test of your serial interface. Connect the `T1OUT` pin to the `R1IN` pin so you are sending data to yourself. Try heating and cooling your sensor and see if the remote temperature value changes whenever the local one does.

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

11. Install the LEDs. Write code to turn the red one on if the remote temperature is higher then the local temperature, and turn the green one on if the remote temperature is lower. **Note:** If you can't get the serial interface to work to receive a remote temperature, use a fake value like 70 degrees as the remote temperature so you an demonstrate that the LEDs work properly based on the local temperature and this fake remote temperature.

12. Install a button for the "Snooze" function. If the button is pressed while the alarm is sounding turn it off but have it come back on for 30 seconds later for 5 seconds. Can you implement this function in a way that **doesn't** change the settings for the alarm time?

Once the above is done, perform the following checks on your clock.

1. Clock correctly shows the time and advances properly.
2. Confirm that the clock values are being stored and retrieved from the EEPROM.
3. The "Time Setting" button moves between all the fields of the clock time and alarm time that can be set.
4. Check that the rotary encoder can be used to set any of the time fields.
5. Check that the alarm sounds when the clock time matches the alarm time.
6. Confirm that the local temperature is being displayed properly on the LCD. Heat and cool the sensor to check that it operates correctly.
7. Use three wires to hook your clock 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 clocks 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, and the red and green LEDs operate correctly to show whether it is hotter or colder at the remote device.