Alarm System (ABET)

CSE 3442 - Embedded Systems I

By: William Wallace

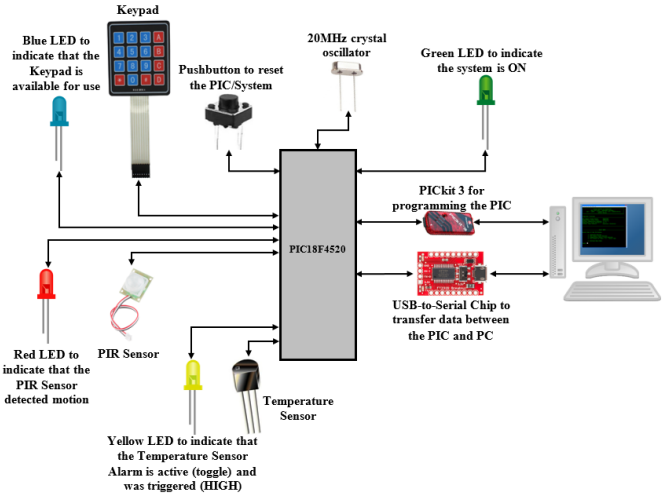
SPRING 2018

5/11/18

CSE 3442 Embedded Systems Lab 7 Alarm System

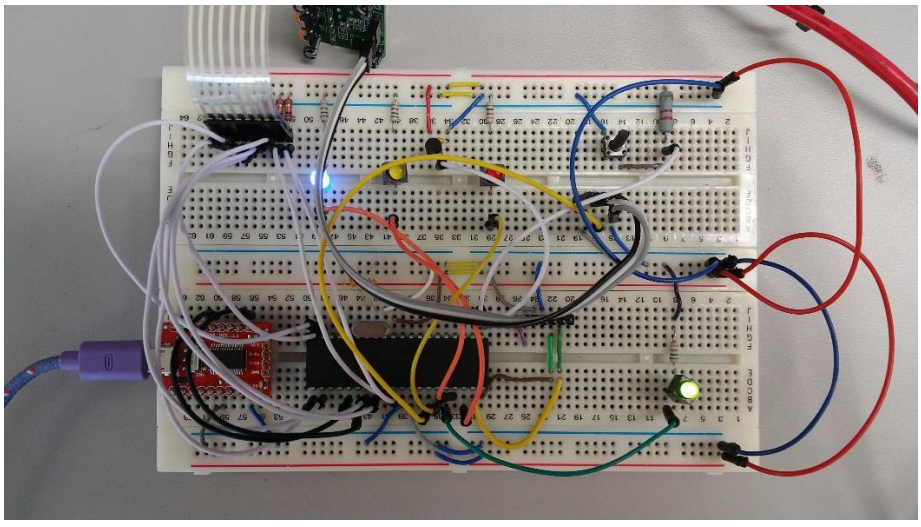
Alarm System Description:

The alarm system designed for this lab was instructed to have the following features: be able to set a password to access the alarm system settings. These settings include editing alarm statuses, changing the temperature threshold for the temperature sensor, utilize a motion sensor that will trigger an alarm using interrupts, and input options. The user will also be able to reset the system using a button located on the breadboard. Important data like saved passwords, alarm statuses, the temperature threshold, and current input selection won't be deleted after the reset button is pushed. This data will be saved in the EEPROM of the PIC. The system has four LEDs to give visual feedback to the user and a keypad as an optional way to interface with the system. The alarm system that was completed for this lab met all requirements and no bonus material was implemented.

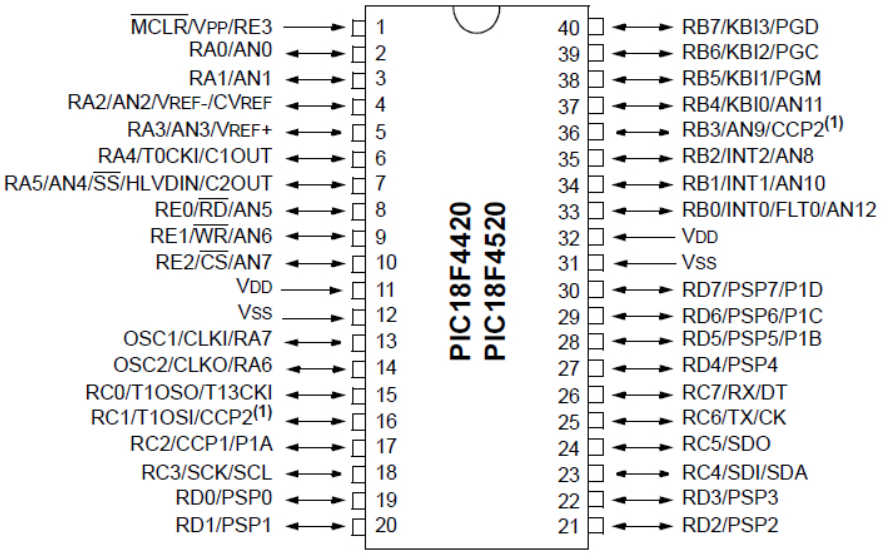


Overview of Process:

The first step I took for this lab was the breadboard implementation. I breadboarded the entire circuit starting from the USB-to-serial chip to the PIC18F4520. Below is an image showing how the system was breadboarded.



The LEDs for the system were hooked up to the PIC's PORTB and the keypad was connected to PORTC of the PIC. AN0 was used for the analog reading from the temperature sensor. A 20MHz crystal was used in this implementation and the motion sensor was hooked up to INT1 on the PIC. The pinout of the PIC used in this alarm system is shown below.



After breadboarding the alarm system, I first set up the configuration bits. For the configuration bits, I needed to set it up so that the external crystal would be used and that the reset would be enabled. Next, I worked on getting the serial communication to function properly. I first had to set up the TRISC settings for the RX and TX pins on the PIC. The RX was set to input and the TX was set as the output. The next settings include: asynchronous mode, low speed BAUD, and 8-bit transmission. Based on these settings and a desired BAUD rate of 9600, I found the SPBRG value using this equation:

$$X = \frac{20MHz}{64 * 9600} - 1$$

The value obtained from this equation was about 31.55, I settled with using the value of 31 (this value was chosen over 32 because 31 was located in the table for this specific PIC). The last thing I had to do for the communication was enabling the serial port RX and TX and enabling the continuous receiver. The code for all of these settings are shown below.

```
//COMMUNICATION
TRISCbits.RC7 = 1;  //RX is input
TRISCbits.RC6 = 0;  //TX is output
SPBRG = 31;         //Low Speed: 9,600 BAUD for Fosc = 20MHz
TXSTAbits.SYNC = 0; //Asynchronous mode
TXSTAbits.BRGH = 0; //Low speed BAUD rate
RCSTAbits.RX9 = 0;  //8-bit transmission
RCSTAbits.SPEN = 1; //Serial port enabled (RX and TX active)
RCSTAbits.CREN = 1; //Enable continuous receiver (ON)
```

To test if the serial communication was properly set up, I read user input from a keyboard and verified that whatever was sent to the PIC was displayed correctly. To test if the breadboard design was correct, I set up the appropriate TRIS settings and tried to turn on the four LEDs. I tested that values were being returned from both the motion sensor and the temperature sensor to finish up the testing for the breadboard. Next, I worked on the user interface and the menus for all the various settings that can be changed by the user. The menus were set up using switch statements with the alarm statuses and various information always located above the options. After creating the menus, I set up the functions for all the various actions the user can make. The first was the password initialization and change password functions. Followed by the temperature sensor settings and the sensor alarm. I then did the input options to finish up the menu functions (specifics for ADC, timers, interrupts will appear in later sections). Lots of testing was done on how user input was used and how the menu was navigated as well as testing to make sure important information like passwords were being saved in the EEPROM. Invalid user input handling and various quirks dealing with user input took most of the time during this project.

PIC Interrupts:

There was a total of three interrupts for the alarm system. The highest priority interrupt was the motion sensor while the timer and the temperature sensor were both low priority interrupts. Because the motion sensor had the highest priority, the alarm for the sensor could go off at any moment during operation. Once the alarm was triggered the high priority interruption would execute. This function locks the alarm system until the user inputs a correct password. While this is occurring, a red LED would light up to indicate that the motion alarm was triggered. For the temperature sensor, after the timer is interrupted the ADC takes effect and when that also interrupts, a series of checks take place. The first check tries to see if the temperature alarm was set by the user. If the alarm was turned on then the calculated temperature (Fahrenheit) is compared to the set temperature threshold. If the temperature exceeds the threshold, the alarm system goes into a triggered state and won't continue until the user inputs a correct password. Once a correct password is entered, the user has the ability to change the temperature threshold or continue back to the menu.

Timers:

The only timer used in this project was TMR0. This timer was used to give periodic sampling for the ADC calculation. The settings set for the timer are shown in the figure below.

```
//TMR0 SETTINGS
T0CONbits.TMR0ON = 0; //Stop TMR0
T0CONbits.T08BIT = 0; //16bit
T0CONbits.T0CS = 0;   //Internal Clock
T0CONbits.PSA = 0;    //Pre-scaler assigned
```

Timer 0 was set to low priority (same as the temperature sensor) and had the interrupt enabled. Every time the timer gets interrupted the low priority interrupt function turns on the ADC. This allows for periodic readings of the temperature. Because no bonus material was implemented in this project, there was no need for any more timer modules. Timer 0 for periodic ADC readings was the only thing necessary.

Timer and ADC calculations:

Using the timer settings shown in the last figure, I calculated the timer pre-scaler and preload like so:

The desired delay time was set to 1 second and with the frequency at 20MHz/4 = 5MHz. The timer was set to be 16-bit so the range of the timer was 0-65,536. To find the cycles/sec I used the formula:

$$X = d * Fin$$

This gave a value of 5,000,000. This value needed to be scaled below 65,536 so I proceeded to divide the number using the PIC's pre-scalars until the value fit the criteria. When 5,000,000 was divided by the 128 pre-scaler I got the value 39,062. This value was subtracted from 65,536 to get the preload value. This came out to be 26,474 which was then converted to HEX to load into the two timer registers TMR0H and TMR0L. The values were 0x67 and 0x6A respectively.

Now for the ADC calculations I used the following settings: channel 0 (AN0), turned on the ADC, set Vss and Vdd, right justified, 8 TAD, Fosc/4, and enabled the interrupt for ADC. The calculation to find the TAD was this: Knowing that the Fosc = 20MHz and using the equation below,

$$Tosc = \frac{1}{20MHz} = \frac{1}{20}\mu s = 0.2\mu s$$

Also knowing that a single T_{AD} had to be at least 1.6µs I multiplied until the 0.2µs was at least 1.6µs. By multiplying by 8 I was able to get exactly 1.6µs so 8 T_{AD} was set in the settings. For the ADC once the low and high bits were read from the registers ADRESL and ADRESH (last 2 bits from ADRESH), both values were added and stored in a single variable (long). To counter the 10mV/C scaling for the temperature sensor, the value was multiplied by 500 (0-5V reference) and divided by 1023 (10-bit ADC). Finally, the voltage offset subtracted 50 from that value (0.5V offset).

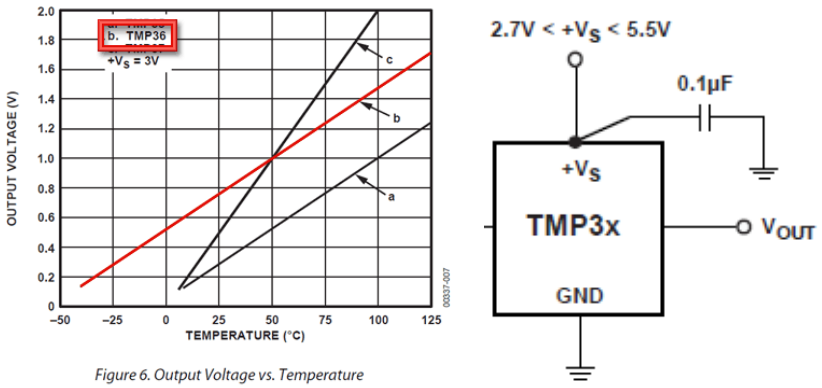


Figure 6. Output Voltage vs. Temperature

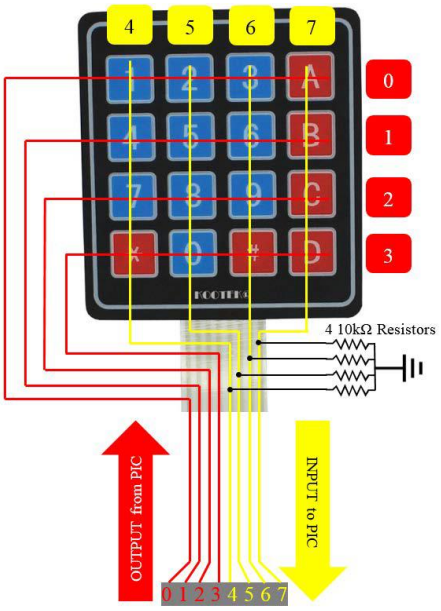
The value was now in terms of Celsius. To convert to Fahrenheit the formula below was used (separating the formula into two lines fixed a weird bug in MPLAB not displaying the value properly).

$$F = Celsius * 1.8;$$

$$F += 32;$$

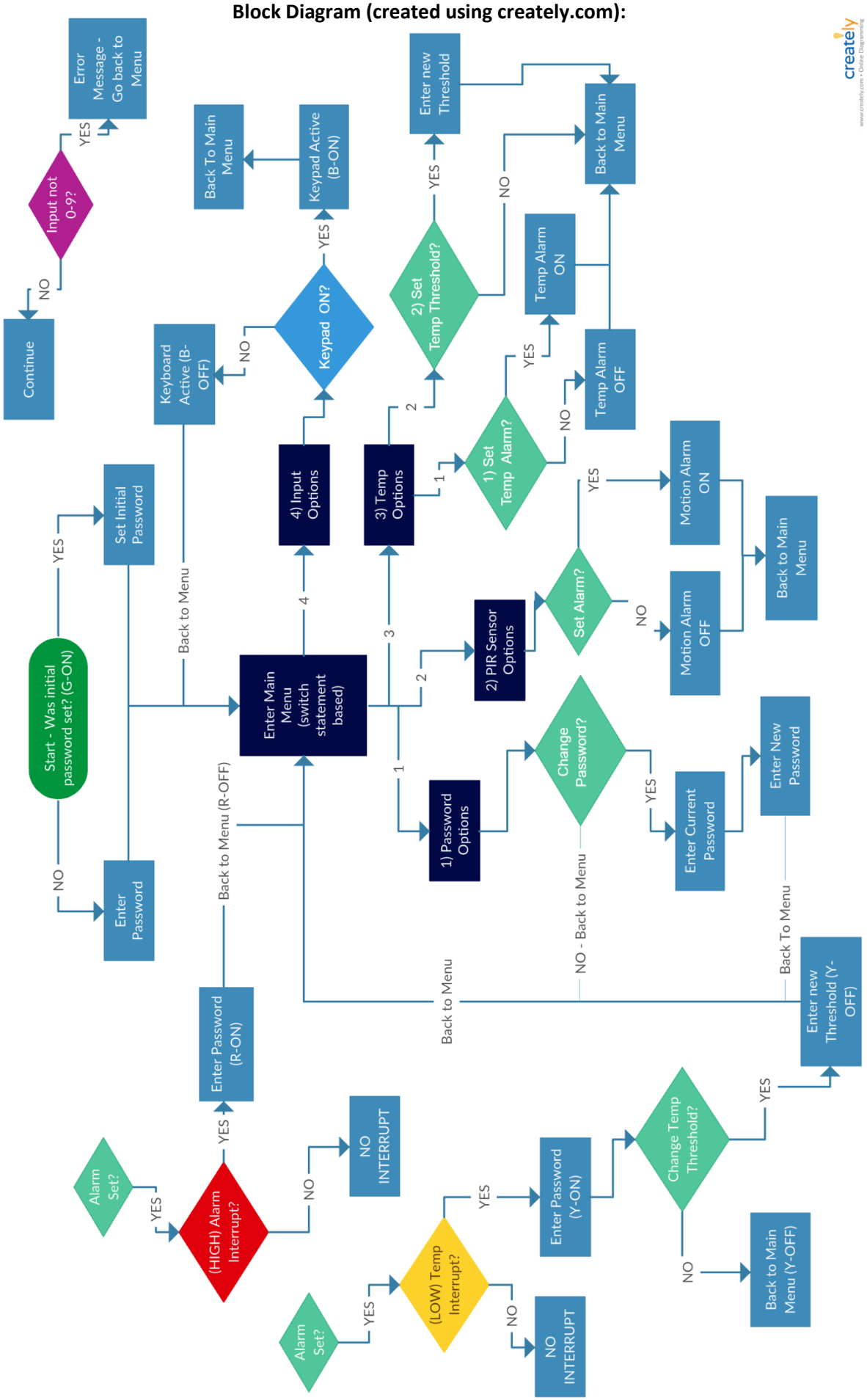
Keypad Logic:

I connected the pins for the keypad to PORTD of the PIC and set the input and output TRISD settings based on the image to the right. After setting the TRISD bits, I set the first row on the keypad high and the other three rows low. I then used a series of if statements to check the columns for that row. If the first row was set high and the first column as also set high then the button for '1' must have been pressed. The function then returns an int of the char '1' (49). The same logic is used for the remaining 3 rows of the keypad. My function for the keypad has all of this logic placed inside an always true while loop, so until a button is pressed the function will continue to poll every column and row. This function will only run if the blue LED is turned on and the keypad is enabled. Code outside of the function dictates how the blue LED alternates between ON and OFF when the buttons are pressed.




```
//POLL Keypad Buttons - Set one row high at a time and poll each column
PORTDbits.RD0 = 1; //Set Row 1 High
PORTDbits.RD1 = 0;
PORTDbits.RD2 = 0;
PORTDbits.RD3 = 0;
if(PORTDbits.RD4 == 1 && PORTDbits.RD5 == 0 && PORTDbits.RD6 == 0 && PORTDbits.RD7 == 0)
{
    return '1';
}
if(PORTDbits.RD4 == 0 && PORTDbits.RD5 == 1 && PORTDbits.RD6 == 0 && PORTDbits.RD7 == 0)
{
    return '2';
}
if(PORTDbits.RD4 == 0 && PORTDbits.RD5 == 0 && PORTDbits.RD6 == 1 && PORTDbits.RD7 == 0)
{
    return '3';
}
if(PORTDbits.RD4 == 0 && PORTDbits.RD5 == 0 && PORTDbits.RD6 == 0 && PORTDbits.RD7 == 1)
{
    return 'A';
}
```

Block Diagram (created using creately.com):



Encountered Problems:

One problem I encountered early in the project was that I forgot to hook up the ground and power rails on the breadboard. While I was testing my LEDs, I noticed none of them would turn on and that half of my circuit seemed to not be powered. Another problem encountered during the project, was the EEPROM read and write functionality. I found that the xc8 compiler version I had installed on my computer stopped supporting the write to EEPROM function, so I had to manually write my own function to write to the EEPROM. Weirdly enough, the native read EEPROM function still worked so I didn't have to write a function for that. The final major problem encountered during this lab had to do with the keypad function. At first, I had the function return the number 1 not the value 49 for 1. The way I implemented the menus for the system were all based on the character 1 (ascii 49) not the number value 1 so the returned value of 1 did nothing. The simple fix was to put single quotes around the numbers that were being returned to make the function return the ascii value.

Suggestions for the future:

The only suggestion I have is to have the project available sooner to allow for more time to work on the alarm system before finals week. Other than that, I felt the project had enough challenge to be rewarding and was a good conglomeration of the classes' material.