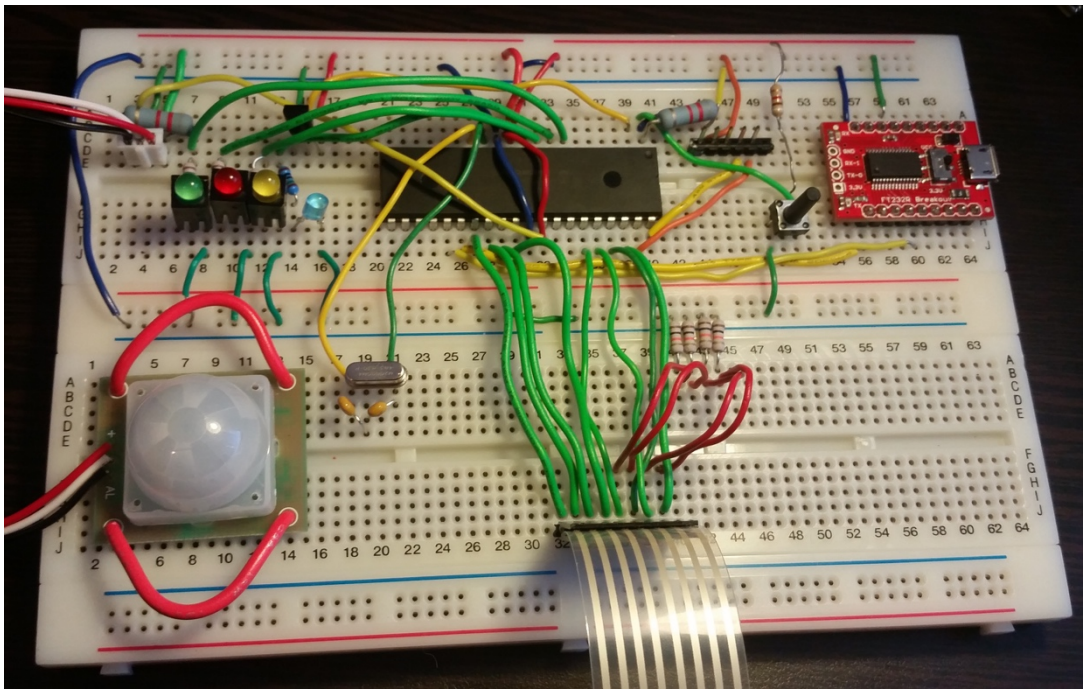
STANDALONE ALARM SYSTEM

Using the PIC18F4520 Microcontroller



Alejandro Limón Naranjo

Prepared For: Dr. Gergely Záruba

Table of Contents

Introduction

The standalone alarm system is a multiple priority trigger based alarm system programmed on the PIC18F4520 microcontroller in C. More specifically, it was programmed via in-circuit programming using a PICkit3 and the MPLAB X IDE.

The main idea behind the security of the alarm is that the user sets authentication at the first boot up, and from there on will be enforced based on the passcode given. The alarm itself is triggered through a rise in temperature, or any unusual motion picked up from either sensor.

There are multiple configurations for this alarm system including: changing the passcode, turning on the motion alarm, turning on the temperature alarm or just changing the main input from the keyboard to a keypad hooked on to the circuit.

The system's status can always be monitored through the LED's. The LED colors corresponding to the status are as follows: green indicated a running system, red indicated a trigger in motion, yellow a sample of temperature and blue indicated the keypad is ready for use. Alternatively, the system's status can be seen on the output of the screen above the menu options.

Overview

The first process in the alarm system was to create a design based on the requirements given. To start implementing functionality the circuit had to first be built.

Process

The process behind putting the alarm system together was to first install the component in the system, implement its functionality, and test the component. After installing each new component, the older parts are not only tested, but also the system as a whole. Pretty standard engineering process.

The order in which the process was implemented was to first add the microcontroller component (the most important component), and test to see if it would power on via programming an LED to light up on boot up. By doing this it was killing three birds with one stone. The in-circuit programming workflow, LED, and microcontroller were all tested during this stage.

The second stage was adding the USB-to-Serial chip. This is the second most important component of the system, not only because it adds the serial communication element to the microcontroller, but because this is the main power source of the microcontroller. It was integrated into the circuit, connected to the appropriate pins on the microcontroller and tested. In code it was tested by setting the correct control, reception and transmitting registers. Furthermore, it was a bit code intensive when adding a circular buffer to the system.

The third stage was to add the clock; a few configurations were done in software to allow the microcontroller to read the correct external clock. In hardware it was necessary to add a capacitor in parallel with the clock and then connecting both pins to RA7 and RA6.

The fourth stage was integrating the 4X4 matrix keypad. To do this some logic had to be learned and read based on the characteristics of how each key is read (more detail on how this works later). Another LED was attached during the testing of this component, and it would light up at the touch of each key. It was a bit code intensive, and probably the most difficult to integrate into the system due to delay and logic errors.

The fifth stage was integrating the motion sensor. It was a fairly simple component to test, because of the nature of its functionality. I set off an LED at the trigger of motion, and to test it in code I was polling for a 0 (since it's active low). The polling was only used for testing (more later on how it is actually used in the real system).

The sixth stage was integrating the temperature sensor. For this some calculations were necessary to see what was actually being read. The actual hardware placement was fairly easy, but the software handling was the difficult part of this component (more on the logic later).

The last stage was creating the user interface, and systematic logic. This is the stage where things needed to be turned on with certain triggers and user input.

Planning

The planning of the alarm system can be broken down into two parts. Hardware and software.

The hardware aspect of planning was mainly making sure all the components would be close to their pins on the microcontroller. The best way to plan this was to create a schematic of the circuit itself for when it is rewired after testing, all the components would be in an ideal place. A smaller aspect of the planning was to checklist all the components to make sure everything that was needed was there.

The software aspect of planning dealt a lot with flags. In other words, certain integer variables that can only be true or false. This is mainly used from when multiple things are running. A list of global flags was written on a sheet of paper. The following flags were used:

ACCESS – used as 1 when the user entered the correct passcode, 0 if any of the alarms are triggered or a reset happened.

CHECKTEMP – 1 when the temperature needs to be compared to the threshold for an alarm trigger, 0 when first started.

THRESHOLDTRIGGERED- 1 when the temperature reaches passed the threshold, and interrupt is triggered, always 0 if CHECKTEMP is 0 or if temperature is below threshold.

MOTIONSENSORTRIGGERED- 1 when motion is detected and interrupt is enables, 0 otherwise.
KEYPADON – 1 when the user turns on keypad, 0 otherwise.
KEYPRESSED- 1 when a key is pressed, 0 otherwise.
MOTIONON- 1 when a user turns on the motion sensor, 0 otherwise.
RUN- always 1, unless something terrible happens.

Another aspect of planning was working the flow of the program user interface and seeing where turn on things, or what would happen if an interrupt occurs in the menu. This is seen in more detail in the state transition UML (glorified flow diagram).

Solution

After testing the components, and planning some of the functionality. The final solution was in place, and the components were interweaved appropriate, the circuit was rewired for a cleaner look, and the user interface was tweaked. Some refactoring was done to ensure lines of code could be reused by implementing them in functions.

Using the PIC Interrupts

The first step in starting writing interrupts is to set up the actual interrupt using assembly code, and redirect to the interrupt handler. In order to do this, a pragma must be setup. A pragma in C is binding the code to go to a certain address in the microcontroller program memory. Address 0x08 is a high interrupt in the memory layout and 0x018 is a lower interrupt. The memory locations must be set to trigger the interrupt. Once it is in the correct memory location a "_asm" call is used in order to allow us to insert assembly code since it is the only way to redirect to an interrupt handler.

Below is a basic code sample of how this is done:

```
#pragma code HIPRI = 0x0008
void HIPRI ( void )
{
 _asm
  GOTO ISRHigh
 _endasm


}
#pragma code LOWPRI = 0x00018
void LOWPRI( void )
{

 _asm
  GOTO ISRLow
 _endasm

}
```

Motion Sensor Interrupt

After the interrupt is triggered for a high priority the idea is to go to the interrupt handler. Since our high priority is a motion trigger, it will go to the handling of the motion. First we must check the flag in the INTCON register to see if it is the correct interrupt. To have an external interrupt happen, we need to connect the component to the correct pin on the microcontroller that enables us to use interrupts. In this case the pin was on PORTB0 and hence anything connected to this will ALWAYS trigger a high interrupt due to the nature of the pin.

Below is the code that defines our pin as a flag to check the interrupt, and the interrupt and the checking of the flag:

```
#define ALARMTRIGGERED INTCONbits.INT0F
void ISRHigh(void);
…
#pragma interrupt ISRHigh
void ISRHigh( void )
{

    if( ALARMTRIGGERED )
    {
      //handling goes here and can be seen in the source code
    }
}
```

The actual handling of the interrupt in a nutshell was to set the access flag to 0 and motion triggered flag to 1. On the user interface perspective, the user is notified that a motion was triggered, and is prompted to provide the passcode until it is entered correctly. If the user is authenticated in the correct manner, they are provided with the option to turn off the interrupt.

The next step of implementing the interrupt is to set the correct control registers, and turn the interrupt on and off appropriately.

Below is the code of the setting of the registers with an explanation of what each thing does in the comments:

```
RCONbits.IPEN = 1; // this turns on priorities in the interrupts
INTCONbits.INT0IE = 0; // this turns off the interrupt 0 enable for clarifying purposes
INTCONbits.GIE_GIEH = 1; // enable global interrupts
INTCONbits.PEIE_GIEL = 1;// enable peripheral interrupts
INTCONbits.INT0IF = 0;// set the triggered flag to 0
INTCON2bits.INTEDG0 = 0; // we need to have this interrupt be negative triggered because the
motion sensor is active low
```

Perhaps one of the most important flags in the motion sensor interrupt is to turn it on, to do this a function was created for clarity purposes. Below is the code to show how this is done:

```
void motionsensorIntON()
{
  INTCONbits.INT0IE = 1;
}
void motionsensorIntOFF()
{
  INTCONbits.INT0IE = 0;
```

```
}
```
After this point it's the user interface's turn to take over and use the functions appropriately!

Temperature/ Timer Interrupt

It was shown above that the pragma code for lower interrupts was in place, and the timer is a lower priority than motion sensor.

The timer interrupt is slightly different because of the way the values of the timer have to be loaded in order to get a certain timing. The timer is always on, the only thing that changes is whether we want to compare the temperature to a threshold (more on that later).

Below is the code for the interrupt handler:

```
#define TIMERTRIGGERED INTCONbits.TMR0IF
…
void ISRLow( void )
{
   if(TIMERTRIGGERED)
   {
     TMR0H = TimerValH;
     TMR0L = TimerValL;
    //more handling goes here such as ADC
   }
}
```

A more detailed description of how the timer, and ADC actually works can be seen further in the lab report.

Using the PIC Timers

The way the timer works is through interrupts there is a flag in the INTCON register that allows us to see whether the timer is done. What is meant by the timer is done? Well the timer is done once enough instruction cycles have gone through to reach a value (more detailed calculation explanation later). This value has to be calculated by hand and then once it is known what it is based on the FOSC (clock speed). The value is loaded into the TMR0H and TMR0L registers. Why are there two registers? Since a 16-bit timer is being used, and each register is 8 bits, we need two.

To be more specific on the use of the Timer0 use in the alarm system; the timer is used to sample the ADC conversion (AKA the value that is read from the temperature sensor). The timer is also used for a menu timeout, motion sensor turn-on and temperature sensor turn-on.

The timer is always running, but in order to control whether these is a menu timeout, or any of the turn-ons, there is a count variable for each one of the user setting. The timer goes off every second by deriving the value needed, so if there is a count and the user inputted count is reached, it will turn on the peripheral or remove access from the user.

Just like the motion sensor interrupt, there are two functions that turn on the timer and a few settings in the INTCON registers that can be seen below:

```
INTCONbits.TMR0IE = 1;// enables the timer
INTCON2bits.TMR0IP = 0; // makes the timer a low priority
…..
void tempsensorIntON()
{
  T0CONbits.TMR0ON = 1;
  TMR0H = TimerValH;
  TMR0L = TimerValL;
}
void tempsensorIntOFF()
{
   T0CONbits.TMR0ON = 0;
}
```

Analog-to-Digital Converter / Timer Calculations

In order to get the analog digital converter and timer working, there needs to be computation behind the scenes. What computations? For example, the temperature reads a value, but it is not in Fahrenheit or Celsius it is a voltage, and to make things even more complicated the microcontroller reads that voltage as an analog value! In the timer perspective as mentioned above a certain value needs to be "preloaded" in order to have the interrupt go off, and there needs to be some calculations to know what needs to be preloaded.

ADC Calculations

Before diving into the formulas used. The concept has to be understood that the ADC reads in a 16-bit value, and it is digital. First, the value must be read into our memory. The ADC by default sends the value in ADRESH and ADRESL TEMPORARILY. Losing the value would make the ADC process useless. In this system the values are dumped two variables first:

```
readingHIGH = ADRESH;
readingLOW = ADRESL;
```

After acquiring the values, they need to be concatenated into one numeric value. The way the ADRESH and ADRESL registers work is that the H is the higher 8 bits and the L is the lower 8 bits. Some simple logic is done to concatenate two 8 bit numbers into a 16-bit number.

```
readingCONCAT = ((readingHIGH << 8) + readingLOW);
```

The way this works is it shifts the reading high to the left as the top 8 bits, as it should be, and it is added to the lower reading. Once we have the value we need, we need to make sense of the value. This is a 10-bit value that represents the true voltage. To convert this value to voltage we use revert the ratio. For example, if we have a value that is 512/1023 in a 5 volt system that represents a .5 ratio, and since we have a 5 volt system it is multiplied by (5*512)/1023 = 2.5 volts. To simply the code and we all know division is more of a pain on processing than multiplication, we find the value of 5/1023 to be 0.004882814 and multiply it times what we have read.

After getting the correct voltage, our temperature sensor is known to have a .5 offset. Which mean we need to subtract .5 of the voltage to the voltage we read in order to get a semi-correct reading.

According to the temperature sensor data sheet, 10mV gives us 1 degree in Celsius. So doing some math here we can figure out the simplified formula. 10mV = . 01V, .01V = 1C, 100(.01)V = 1V, so multiply 100 by both sides to get the Celsius in terms of 1 volt. 100(.01)V = 100(1)C, and hence our formula is 1V = 100C.

To get the Celsius value from the voltage we read we need to subtract the offset and multiply by 100. It's not done there, it is necessary to convert it to Fahrenheit, so the classic Celsius to Fahrenheit equation is used. Due to the defect in the temperature sensor, it always seems to be off by 6 degrees, so it is added at the end.

Below is the code that does everything that was explained:

```
voltage = readingCONCAT * 0.004882814;
temperatureC = (voltage - 0.5) * 100;
temperatureF = ((int) temperatureC * (9.0/5.0) + 32.0)+6;
```

Timer Calculations

To get the preload value that was explained earlier, it is necessary to know our current clock or FOSC. In this case have a clock of 20mHz. To explain this in a clearer manner, an example with a slower clock will be used in calculations. Let's say there is 1Hz clock and we have a pre-scale value of 4. First it is necessary to find the time for one clock tick, so 1/1 equals a 1 second clock tick with. The way the prescaler works, it is a value that is divided by FOSC/4. The FOSC in this example is 1 so the value we obtain is 1/8 . So if you want the timer to run for 1 second you need to have a preload value 2^16 – 1/8.

(2^(16))- (1/(FOSC/4*PRESCALER))*(TIMEWANTEDINSECONDS) = PRELOADVALUE

In this case X is the value we put preload into the timer register. Applying the formula to system that is being used:

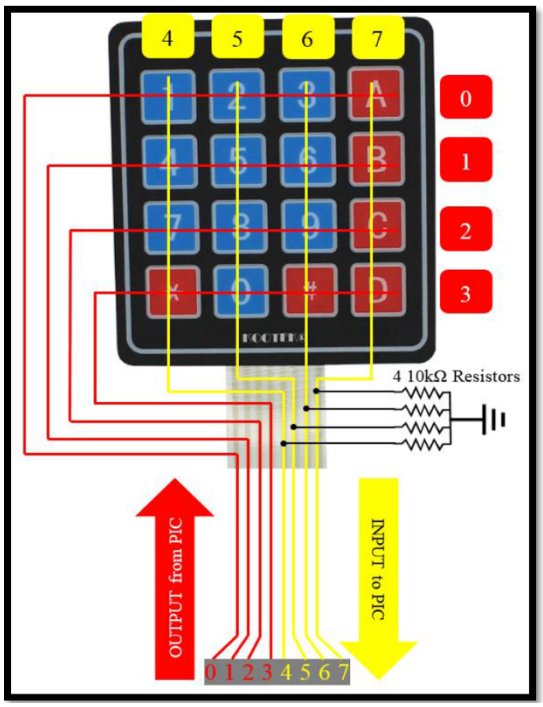
PRESCALAR = 128
65536 - (1/(20,000,000Hz/4*128))*(1) = PRELOADVALUE
PRELOADVALUE = 26,473.5

In hex this is 0x6769

Once the value is loaded into the preload register of the timer, it will cause an interrupt when it counts down from that value to 0. So to run the timer for another second, we need to keep loading 0x6769 to the timer.

Keypad Logic

To read a key from the the keypad, there needs to be some inputs and outputs defined to fully understand the process.



The rows on the keypad as you can see are outputs and the columns are inputs to the PIC. Before starting the logic, all the inputs and outputs had to be defining using the TRIS statement.

To actually read from each key, you must set the certain row high and then check each column to see if it has activated. To do this, each row can be seen as an iteration in a loop, and the other rows are deactivated during this process.

After finding which key was pressed, it is necessary to wait for the key to be un-pressed, so there will be a loop waiting for the column to no longer be high. Before the character is inserted to the buffer there also needs to be a de-bouncing period. In this case have de-bounced the key for .1275 seconds to ensure there won't be a brute force of inserted characters.

There is a flag mentioned at the beginning called KEYPRESSED. This flag will go high once a character goes into the buffer. This will stop the reading of the keypad to start over and read again. There is another flag that is called SENDINPUT. This will go high once the enter key (the pound sign) is pressed. This will exit the keypad handling function.

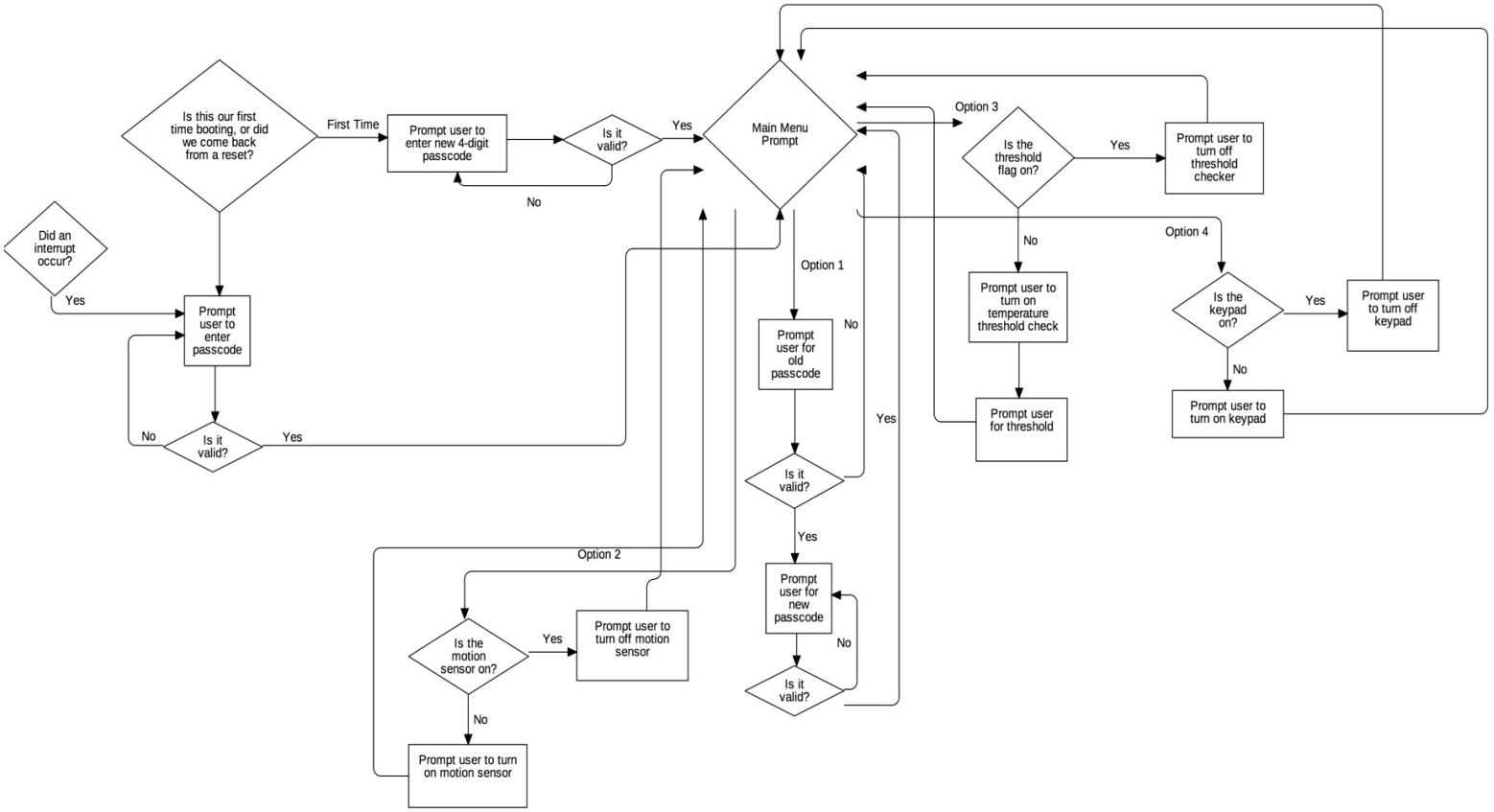
To better understand how the logic was implemented, the code is shown below for one row:

```
KEYPRESSED = 0;
   /*Start the handling for the 1st Row*/
  while( SENDINPUT == 0 )
  {
    while( KEYPRESSED == 0 )
    {
      for(i = 0; i < 4 ; i++)
      {

        if( i == 0 )
        {
          R2 = R3 = R4 = LOW;
          R1 = HIGH;
          if(C1 == HIGH)
          {
            void Delay50ms();
            while(C1)
            {
               debounce();
               insertChar('1');
            }
          }
          else if(C2 == HIGH)
          {
            void Delay50ms();
            while(C2)
            {
               debounce();
               insertChar('2');
            }

          }
          else if(C3 == HIGH)
          {
            void Delay50ms();
            while(C3)
            {
               debounce();
               insertChar('3');
            }
          }
          else if(C4 == HIGH)
          {
            void Delay50ms();
            /*SENDINPUT = 1;
            break;*/
            while(C4)
            {
               debounce();
               insertChar('A');
            }
          }
          else{} … //the next rows follow
```

Alarm System Block Diagram



Problems Encountered

Problem: How to avoid clunky wiring?

Solution: Try to use as little wire as possible, move the components closer to their actual pin assignments, use as many as the voltage and ground lines on the breadboard as possible, and keep any of the external USB connection on the outside border of the breadboard.

Problem: The MPLAB X IDE on MAC OS X El Capitan can't read the PICkit3, why?

Solution: Reinstall the MPLAB X IDE, update the IDE or get a Windows computer

Problem: The MPLAB IDE is not reading any voltage, why?

Solution: Make sure your PICkit3 is giving voltage to the circuit if your usb-to-serial is not yet connected. If it is connected, make sure the usb-to-serial is giving the correct output voltage.

Problem: The X component is not responding, why?

Solution: Triple check wiring, TRIS input assignments in the code, and replace the component with one identical.

Problem: There are multiple things in the buffer after I touched a key, what happened?

Solution: The keypad was not de-bounced, and a proper delay is needed to it will not brute force the buffer.

Problem: The temperature sensor is reading weird values, why?

Solution: Check to see if the polarity of the temperature sensor pins are in the right place, add an offset when grabbing the voltage values.

Problem: The motion sensor interrupt is not going off, why?

Solution: Make sure the TRISB0 port is set at input, and it is not modified in any other part of the code. Triple check the INTCON registers to ensure you have what you need, clear any flags at the beginning. MOST IMPORTANT, clear the flag after handling an interrupt.

Problem: The serial port is not sending is not receiving or sending anything, why?

Solution: Check to see if the TX is connected to RX on the PIC and vise-versa. It is human nature to try to connect TX to TX.

Problem: The motion sensor interrupt keeps going off, what is happening? I'm not moving.

Solution: NEVER write values to EEPROM inside the interrupt handler, the EEPROM requires all interrupts to be turned off to write to it. Make sure the motion sensor interrupt is set to trigger on negative edge.

Problem: I pressed a key on the keypad and its giving my all the values in that column or the wrong key.

Solution: Again, check to make sure you are de-bouncing. You have to have each row as an iteration in a loop to ensure that nothing else is being checked. Also, remember to set the other rows to 0 while checking the main row.