**ELEC342 Laboratory Report 2**

**Arman Chowdhury (43925197)**

1. **Introduction**

We were given the task of designing and building a simple fire alarm system using a PIC18F14K22 microcontroller in assembly language. We were required to use interrupts to design an event driven system. In the design of our fire alarm system special consideration had to be given to the wiring and initialization of the chip, the different timers being used, the different software and hardware interrupts used amongst various other things. We made close references to the datasheet in the building process.

1. **Procedure and Discussion**

Throughout the design process there were many key points in which crucial decisions were made in relation to the design of the alarm system. These are highlighted below:

* **Wiring and Design Overview**

From Section 7 (Interrupts) of the PIC18F14K22 datasheet we discovered the different registers and pins required for our design. The overview of our design and wiring is shown below.

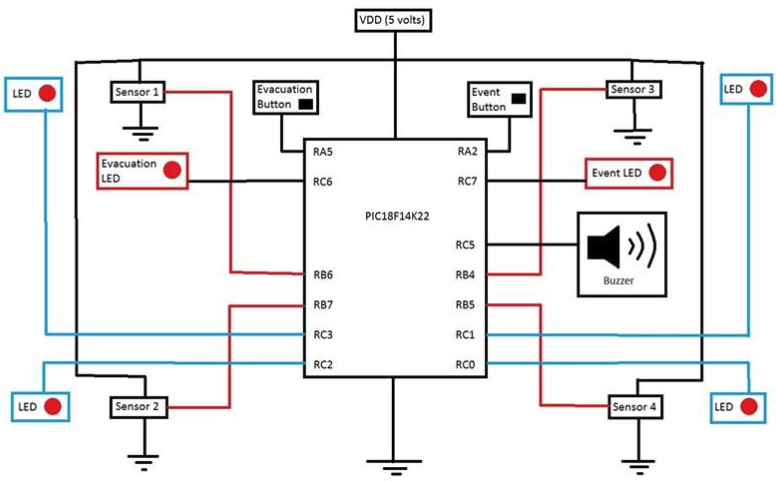


Figure : Wiring Diagram

We discovered that the PIC controller can have multiple interrupt sources and an interrupt priority feature that allows most interrupt sources to be assigned a high priority level or a low priority level. The PIC can have both software generated interrupts and hardware generated interrupts. This was crucial in our design and we took full advantage of these features. There are twelve registers which are used to control interrupt operations. We used most of them in implementing our alarm system.

We initially took a bottom-up approach when designing the alarm system. We looked at what the outcome was and started designing based on that. We ended up designing a state machine with 5 states. However, the states were merely markers on our journey to the end. All instructions and decisions were made when transitioning between states.

* **Initialization**

To use the PIC controller, various registers needed to be initialized. Since the I/O pins we were using were spread over PORTA, PORTB and PORTC, we had to initialize TRISA, TRISB and TRISC registers. Configuring these three registers would set the I/O pins to either function as an input or an output.

For all three registers, we needed the pins to be set to 1 for an input, else the pins were set to 0. This resulted in the following bit sequences for the TRISA, TRISB and TRISC registers.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| TRISC | RC7 | RC6 | RC5 | RC4 | RC3 | RC2 | RC1 | RC0 |
| Bit | 0 | 0 | 0 | X | 0 | 0 | 0 | 0 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| TRISB | RB7 | RB6 | RB5 | RB4 | RB3 | RB2 | RB1 | RB0 |
| Bit | 1 | 1 | 1 | 1 | x | x | x | x |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| TRISA | RA7 | RA6 | RA5 | RA4 | RA3 | RA2 | RA1 | RA0 |
| Bit | x | x | 1 | x | x | 1 | x | x |

Pin RC5 was used for our buzzer output, since it was the only pin capable of carrying out a PWM signal. The other pins were all LED outputs. Pins RB7 to RB4 served as our smoke detector inputs. Pins RA5 and RA2 were our emergency and reset buttons.

We also had to set our desired clock speed of 250KHz. 250KHz was chosen because this made our timer delays closer to the required values and the system ran at a reasonable speed which made it easier for debugging. The lights would occasionally flicker if we had forgotten to clear an interrupt flag bit. If we opted for a higher clock speed, we wouldn’t be able to pick up these slight light flickering. We used the OSCON register to define our system clock. The following bit sequence was pushed into the OSCON register.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| OSCON | IDLEN | IRCF2 | IRCF1 | IRCF0 | OSTS | HFIOFS | SCS1 | SCS0 |
| Bit | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |

Since our system was an interrupt driven system, we had to set up PORTB and PORTA to cause an interrupt on change. So, we pushed the following values into IOCA and IOCB.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| IOCA | U-0 | U-0 | IOCA5 | IOCA4 | IOCA3 | IOCA2 | IOCA1 | IOCA0 |
| Bit | x | x | 1 | x | x | 1 | x | x |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| IOCB | IOCB7 | IOCB6 | IOCB5 | IOCB4 | IOCB3 | IOCB2 | IOCB1 | IOCB0 |
| Bit | 1 | 1 | 1 | 1 | x | X | x | x |

Once these registers were configured, we performed a read from PORTA and PORTB so that the values on either side of the latches were the same and there were no pending interrupts once the system was booted up.

The next registers, we had to configure were the interrupt control registers - INTCON, INTCON2 and INTCON3.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| INTCON | GIE | PEIE | TMR0IE | INT0IE | RABIE | TMR0IF | INT0IF | RABIF |
| Bit | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |

We enabled unmasked interrupts and peripheral interrupts by setting Bit-7 and Bit-6 to 1. For our 3second timer interrupt we used Timer0. So, we enabled timer0 to interrupt on overflow by setting Bit-5 to 1. Bit-3 was set to 1 to enable interrupts to occur from changes on PORTA or PORTB. We left the rest as their default values since our system didn’t have any external interrupts.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| INTCON2 | /RABPU | X | X | X | U-0 | TMR0IP | U-0 | RABIP |
| Bit | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |

In our design, we termed all interrupts as high priority and so Bit-0 and Bit-2 were set to 1. All other bit’s in the INTCON2 register were kept at their default values since they didn’t concern our design.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| INTCON3 | X | X | U-0 | X | X | U-0 | X | X |
| Bit | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Since all bits in this register were responsible for external interrupts, we disabled all of them.

We used Timer0, Timer1 and Timer2 in our system for various purposes. Timer0 was responsible for causing an interrupt after three seconds of it being started. The T0CON register was responsible for configuring this timer. We set the following bit sequence to T0CON.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T0CON | TMR0ON | T08BIT | T0CS | T0SE | PSA | T0PS2 | T0PS1 | T0PS0 |
| Bit | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

To allow our timer to have more precision we opted for a 16-Bit timer rather than an 8-Bit, so Bit-6 is set to 1. We were using the internal oscillator for our clock and so we set Bit-5 and Bit-4 to 0 and 1 respectively. We assigned the timer’s pre-scaler for us to get as close as possible to the desired time of 3 seconds. We chose 1:256 as our pre-scaler for maximum precision. Thus, we set Bit-4 to 0 and the rest to 1’s.

Timer1 was used in an analogous way, but for a different purpose. We used Timer1 for our flashing lights and siren routines. The following table shows the bit sequence we pushed into T1CON, to configure Timer1.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T1CON | RD16 | T1RUN | T1CKPS1 | T1CKPS0 | T1OSCEN | /T1SYNCH | TMR1CS | TMR1ON |
| Bit | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

Bit-7 was set to 1 because we were using Timer1 as a 16-Bit timer. This was to keep our timers consistent with each other and for precision. Since our system clock was defined by the internal oscillator and not from Timer1 oscillator, we set Bit-6 to 0. We picked the highest pre-scaler, because of the extra precision. It also turned out that using a larger pre-scaler allowed us to start our timer at a much higher value. This meant that our timer was less power hungry. So, Bit-5 and Bit-4 were set to 1. We enabled the Timer1 oscillator by setting Bit-3 to 1. Since our design didn’t make use of an external clock and used the internal clock, we kept Bit-2 at its default value and set Bit-1 to 0.

Timer2 was used to send a PWM signal to our buzzer so that different sirens could be played in the case of an emergency. We set the following bit sequence to T2CON.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T2CON | U-0 | T2OUTPS3 | PS2 | PS1 | PS0 | TMR2ON | T2CKPS1 | T2CKPS0 |
| Bit | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

Bit-6 to Bit-3 were set to 1 since that defined our post-scale to 1:16. Bit-1 and Bit-0 were set to 0 due to our pre-scaler being 1:1.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PIE1 | U-0 | ADIE | RCIE | TXIE | SSPIE | CCP1IE | TMR2IE | TMR1IE |
| Bit | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Since one of the peripheral interrupts we were using was Timer1, we enabled interrupt on overflow by setting Bit-0 to 1. All other bits were kept at their default value.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PIE2 | OSCFIE | C1IE | C2IE | EEIE | BCLIE | U-0 | TMR3IE | U-0 |
| Bit | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

We also kept the other peripheral interrupt register at its default values since none of its features were being used in our system.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RCON | IPEN | SBOREN | U-0 | /RI | /TO | /PD | /POR | /BOR |
| Bit | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Bit-7 was the only bit that concerned our design since it disabled priority levels on interrupts. All our interrupts were set to high priority.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| CCP1CON | P1M1 | P1M0 | DC1B1 | DC1B0 | CCP1M3 | CCP1M2 | CCP1M1 | CCP1M0 |
| Bit | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |

The CCP1CON register was responsible for our PWM signal being sent to the buzzer. We selected a single output since the other options were concerned with motor control via a half-bridge or a full-bridge. Thus, Bit-7 and Bit-6 were set to 0. Since our PWM output was sent to the buzzer and we wanted to generate sound through this signal, we set up the duty cycle to 50% to replicate a sin wave.

* **Building All Required Functions**

When designing the system, we decided to build the different functioning parts first. We made a list of all the components that were required in the design and started implementing and testing them separately, one by one. The different components that went into designing our fire alarm system are specified below.

1. **Display Lights**

The first design choice that we had to make was regarding the placement of the sensor inputs and their corresponding output LEDs. We knew that RC5 had to be used for the buzzer output and we had to accommodate the Event LED and the Evacuate LED. So, we decided on the bottom four I/O pins of PORTC to be the corresponding sensor indicator lights. With the aid of a simple swap-function we could quickly display the current sensor inputs (top four bits of PORTB) on PORTC (bottom four bits). This was not all we had to do. We also needed a way to display all required lights as we transitioned between the states. So, we had to carry over our previous LED values. We achieved this by creating an 8-bit register called “LED\_State” which held the current values of all the LEDs on the board. We then made a function called “displayleds” which simply pushed the values stored in this register onto PORTC. We called upon this “displayleds” function after each transition was complete. This ensured that values were not overwritten and lights did not flicker.

1. **Flashing Lights**

We then needed to make the LEDs on the board flash in the case of a Warning or Evacuation sequence. The easiest way to make this happen was to exclusive-or the lights with a series of 1’s. We found that, “exclusive-or-ing” our current LED value with a series of 1’s changed them to their conjugate. The diagram below gives a clearer idea of this in actions.

|  |  |
| --- | --- |
| Not-Flash | |
| Current | 0 0 0 1 1 1 0 0 |
| XOR\_Leds | 1 1 1 1 1 1 1 1 |
| Result | 1 1 1 0 0 0 1 1 |
| Flash | |
| Current | 0 0 0 1 1 1 0 0 |
| XOR\_Leds | 0 0 0 0 0 0 0 0 |
| Result | 0 0 0 1 1 1 0 0 |

So, we created another register called “XOR\_Leds” which changed from being all 1’s to all 0’s each time the “flashleds” subroutine was called. This in turn would the lights.

Then we needed to decide on a time delay for this flashing to occur. We used timer 1 to make a 0.5s delay and a 0.25s delay, for our Warning mode and Evacuate mode sequences. The math behind us achieving these numbers are shown below.

And for the 0.25s delay we had the following mathematics.

These hexadecimal values were pushed into the TMR1H and TMR1L registers to set the timer up to count to 0.5 seconds and 0.25 seconds. So, every time Timer1 would time out, it would generate an interrupt. The interrupt was then used to flip the values of the lights and reset the timer to start counting again.

Once the flashing part was decided upon, we then had to incorporate the buzzer. We used Timer2 to send a PWM signal to the buzzer. We made the duty cycle 50% to represent a sin wave. We changed the frequency to produce different sounds in Warning mode and in Evacuate Mode. In our flashing subroutines, we had to enable the PWM module and set the desired frequency. Once that was done, we simply turned on Timer2 and the buzzer would sound and alarm. This wasn’t all we had to do. We later found that, we would have to toggle the buzzer on and off to make an alarm noise. The way we decided to do this was again based on the “XOR\_Leds” register So, if this register was all 0’s we would turn Timer2 off and vice versa.

Once this was done we ended up with two flashing lights routines, “Flashy\_Fast” and “Flashy\_Slow”. However, since they both used Timer1, when an interrupt from Timer1 occurred we had to differentiate between the two. This was easily done by determining which state the system was in. If the state was in Warning mode, it would run, “Set\_Flashy\_Slow” and “Set\_Buzzer\_LowPitch”, and if the system was in Evacuate mode, it would run “Set\_Flashy\_Fast” and “Set\_Buzzer\_HighPitch”. This was all done within the “flashleds” routine.

1. **Three Second Timer**

The next component we had to design was the three second timer. This was another timer driven interrupt that would run in the background of our program. We used Timer0 to make this happen. The following mathematical calculations show how we managed to get a three second delay using timer0.

We pushed these values into the TMR0H and TMR0L registers. We used a function called, “Set\_Time\_3Sec” to push these desired values into their respective registers and start the timer. Similarly, like Timer1, when Timer0 overflowed it produced an interrupt. Based off this interrupt and which state the system was currently in, we could make decisions.

1. **Stop Functions**

Since our design hinged upon timers overflowing and multiple interrupts occurring in between states, we had to create subroutines to stop these timers. There was a function dedicated to stop Timer0, called “interrupt\_3Sec” and “Stop\_Time\_3Sec” and another to stop Timer1, called “Stop\_Flashy\_All”.

1. **Triggered and Current**

We realized in the later stages of our development that we needed to store all historical values of the respective sensors that were set off. So, when we performed our Midway reset, these values would appear. We created a register called, “Triggered” which stored all historical values or the sensors. However, we also needed a way of making sure that only new sensor values were added to this register. That is where the “Current” register came into play. This register was simply used to sample the current sensor values and compare them those stored in “Triggered”. If they were different, then we added them to “Triggered”, else we ignored the sensor. The comparison was again done using the following truth table.

|  |  |  |
| --- | --- | --- |
| Triggered | Current | Result |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

So, the truth table yielded the following Boolean Equation.

This method was heavily used in decision making in our design.

1. **Evacuate Subroutine**

Since our system incorporated an emergency button, we decided that it would be best to design an evacuate subroutine called, “doEvac”. So, whenever the emergency button was pressed, the system would cancel all other instructions and refer to this subroutine. Since pressing the emergency button would cause an interrupt on change. In this subroutine, the three second timer was stopped, system was set to “Evacuate State”, the emergency LED would be light and the system performed “Set\_Flashy\_fast”. From this point, onwards the system would just continue to stay in evacuate state.

1. **Reset Subroutine**

Similarly, for, the reset button we decided to have a subroutine which cleared the board and set the system back to square 1. This function was used extensively in the development of our system. It cleared all file registers and stopped all timers and flash sequences.

1. **Midway Reset Subroutine**

Our design requirements stated that when pressing the reset button in the evacuate state, the system should shop flashing and show which sectors were on fire. A straightforward way we accounted for this was create yet another subroutine which forced the system into State 4, stopped the flash routines, and cleared all file registers. This function also, pushed the values of “Triggered” onto PORTC with the aid of the “displayleds” function.

* **Design Choices**

In the implementation of our design we had to make many design choices since the full specifications were not given to us. Many corner cases were not covered and so we made executive decisions. The way our system works is that, when an interrupt occurs, it enters the High Interrupt Service Routine. In there, there are checks in place that differentiate between the interrupts flags and send the system off to the appropriate section of the program. Each type of interrupt has its own section of code which it has to run through. In each section, the system’s State was determined and accordingly instructions were carried out. The following are some of the many design choices that we had to make while developing this alarm system.

1. **State 0 (Happy State)**

When in State 0 (Happy State), a sensor is triggered, the system is designed to immediately change to State 1. A three second timer is started and the sensor that was set off is added to “Triggered”. However, the specifications do not account for the reset and emergency button being pressed. Our system is designed to move directly into Evacuate State once the emergency button is pressed. The instructions are all carried out in the transitions from one state to the next. The states are merely just markers to guide the full system. Once there is an interrupt on change on RA5, the system runs the, “doEvac” routine.

Our system is also designed to stay in Happy State when the reset button is pressed. However, there was a distinct button bounce problem we faced. RA2 is an active-low switch and unlike RA5 it isn’t a pulse. So, there are two interrupts on change that occur from one press of the reset button, - one for the falling edge, and another for the rising edge. Since our program only accounted for the falling edge of the button, the rising edge would have been considered a sensor press and thus would cause the system to travel to State 1.

To avoid this problem, we introduced a register called the “ResetFlag”. So, when the reset button is pressed, it would set the “ResetFlag” and the program would then branch off to perform the “doReset” routine. When the button is let go, for the rising edge, there is another pending interrupt for the system. This time, when the system runs through, we check specifically for the rising edge. The program directs the system to a marker placed in the subroutine, “State0Button” called “marker0”. At this marker, the “ResetFlag” is checked for being high, and is set to low. And the system, returns from interrupt and resumes normal operation.

The reason why we accounted for the corner case of the Reset Button, is because the fire officials may accidentally press the reset Button twice while “hard-resetting” the system.

1. **State 1**

There are multiple scenarios that can take place when an interrupt occurs in State 1. When a new sensor is set off, our system is immediately designed to move to State 2 (Warning Mode) and adds the new sensor value to “Triggered”. The previous three second timer is cancelled and a new one is started, and the slow flashing lights routine is set off. The Boolean Equation previously mentioned is used to determine if the new sensor is a unique one or not. If it isn’t then the system is designed to remain in its current state of operation until a different interrupt occurs.

When the evacuate button is pressed, the system is designed to cancel the three second timer and branch to the “doEvac” routine and perform the instructions specified. We decided on this because and evacuation is given the highest priority. In the case of a fire, the building must be notified of an evacuation as soon as possible.

When the Reset Button is pressed, the system is designed to cancel the three second timer and move straight to Happy State by performing the “doReset” routine. Since There may just be a false alarm in one of the sectors, pressing the reset Button should negate the system elevating the situation.

However, at the end of the three seconds Timer0 generates a software interrupt. This is when the system samples the sensors using the “Current” register. It checks if this register is all 0 or not. If it is, there was a false alarm and the system can return to Happy State on its own. If the same sensor is still triggered then the program fails the previous check, and moves to State 2 (Warning State).

If it was a false alarm, the program branches to the “FalseAlarm” marker, else the “RealDeal” marker.

1. **State 2 (Warning State)**

Once again there are multiple scenarios our program goes through based off the different interrupts that occur.

When the system has reached State 2, the slow flashing lights routine starts. Each time Timer1 overflows and an interrupt occurs, the systems continues to run the same slow flashing lights routine.

Again, when a new sensor is set off, the system is designed to stop the three second timer, add the new sensor value to “Triggered”, and move to State 3 (Evacuate State).

The system also moves to Evacuate State once the Emergency Button is pressed through the “doEvac” routine.

When the three seconds is up and Timer0 overflows, the system conducts the same check that it did previously. It samples the sensor values and checks if the sensors are all 0. If they are then the program remains in Warning State unless other interrupts occur to change its State. We have purposely checked for if all sensors are 0, because all other corner cases are covered by the other interrupts.

If a new sensor was triggered at any time, the System would move to Evacuate State. If the fire in the two triggered sectors still exist and produce smoke for the first three seconds, then the system will automatically move to Evacuate State as well. Thus, we only check if the sensors are 0 or not.

In the case that the Reset Button is pressed, the system is designed to do a Midway Reset. This is to simulate a fireman has arrived on scene and has pressed the Reset Button and is viewing the areas or sectors that were set off by smoke. This is a feature that we implemented to help protect the building better. This makes it easier for the fireman to check the affected sectors more thoroughly.

In this case we didn’t need to account for a switch bounce of the Reset button. When the “doMidWayReset” routine is carried out, the system is forced to enter State 4 (reset State – i). In this State, all interrupts from PORTA and PORTB are ignored, except for the emergency button.

**Possible Improvement:**

A simple improvement that could be made to the system, is to check if any given sensor is triggered for any given three seconds. This would require starting the three second timer one a sensor is set off and letting timer0 count till it overflows. Once the timer0 interrupt has occurred, the system should only check for that single sensor being 0. This would be far more accurate in a real fire scenario.

1. **State 3 (Evacuate State)**

When the system has been elevated to Evacuate mode, it loops in this State. This is the highest level the situation can be elevated to.

When the Reset Button is pressed, the system branches to “doMidWayReset”. In this state, if a new sensor is set off, the system just adds this to “Triggered” in the background and displays it when a Midway Reset is performed.

When the evacuate button is pressed, it is simply ignored. This is because, if it is not ignored, then the system starts the Evacuate sequence each time the button is pressed. If the button is held down, the System freezes.

We saw this as a potential problem. What if there was a faulty emergency button? What if someone smashed the emergency button too hard and it was stuck in its pressed position?

We fixed this issue by ignoring the emergency button. Since all states pointed to the Evacuate state once the emergency button was pressed, there was no need to incorporate that into our design. This resulted in a flawless design and the only interrupt that would break its endless loop would be a Midway Reset. Again, this simulates a fireman arriving on scene and pressing the Reset button.

1. **State 4 (Reset State - i)**

There are two things that can happen when the system is in this state. If the evacuate button is pressed, the system branches straight into “doEvac” and enters Evacuate State.

Other than that, the system starts a three second timer. At the end of the three second timer, the system moves to State 5 (reset State – ii).

The switch bounce was a factor here, ever time the reset button was pressed, a new timer would start. We needed a way of ignoring all the reset button presses except for the first one.

We implemented the same method as previously used for the “ResetFlag”. But, instead of “ResetFlag”, we had “Timer0Set”. The same principles applied and it gave us the same results.

This ensured that, even if the fireman resetting the system, got the time wrong, he would not be penalized. Even if the reset button is held down, the system still starts the three second timer.

All other sensors are ignored in this state. This was a design choice made by our team of skilled engineers. The only way the Reset State – I could be entered is if, the situation was elevated to a Warning or Evacuate State. This would only happen, if a real fire was detected by the smoke detectors. Thus, the only thing left to elevate the situation would be an evacuation. Thus, we have only accounted for this.

**Possible Improvement:**

A possible improvement in our design could be to cause the system to move to evacuate, if a new sensor is triggered. This would let the people in the building and surrounding the building know that there is still danger within.

1. **State 5 (Reset State – ii)**

State 5 is a hidden state and was created to make it easier to perform a full Reset on the system. It still carried over all functionality of State 4. Visually the system appears to be in State 4 but internally the system has entered State 5. Essentially the system ignores all inputs other than the Reset Button and the Emergency Button. The reset Button causes the system to do a full reset and return to Happy State.

**Possible Improvement:**

A possible improvement in our design could be to cause the system to move to evacuate, if a new sensor is triggered. This would let the people in the building and surrounding the building know that there is still danger within.

1. **Alternative Method and Possible Improvements**

An alternative method of doing this would be to assign the Emergency Button and the Reset button their own interrupts. This can be done by enabling the INT2 interrupt in INTCON3. We can also use RA0 as the Evacuate button and assign it tis own interrupt, INT1. This can also be enabled in INTCON3.

In this control register we can make the interrupt happen on the rising edge or the falling edge of the button press. Thus, this eliminates us from having a ResetFlag. It makes our program more streamline and perform better.

Another improvement we could have made is to use a modular design approach. We were attempting this at first by setting up the different components and using then as we moved along through the program. However, that idea fell apart once we tried to differentiate the different interrupts. The checks done in our High Interrupt Service routine was the choke-point in our program. Having a modular design would have made it easier for us to change things since the states wouldn’t be interdependent on one another. The only thing relating them would be data.

Another more farfetched idea that we initially had was to have a polling-based system. This would mean that we would constantly have to check for which interrupt had occurred. This isn’t a wonderful way of doing this, the design we chose is far superior.

1. **References**

**[1] Chinmay’s ELEC342 Fire Alarm report, Chinmay Vaishnav, 4th June 2017**

**[2] Microchip, PIC18F14K22, Sections 2, 7 - 13**

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; \*

; Microchip licenses this software to you solely for use with Microchip \*

; products. The software is owned by Microchip and/or its licensors, and is \*

; protected under applicable copyright laws. All rights reserved. \*

; \*

; This software and any accompanying information is for suggestion only. \*

; It shall not be deemed to modify Microchip?s standard warranty for its \*

; products. It is your responsibility to ensure that this software meets \*

; your requirements. \*

; \*

; SOFTWARE IS PROVIDED "AS IS". MICROCHIP AND ITS LICENSORS EXPRESSLY \*

; DISCLAIM ANY WARRANTY OF ANY KIND, WHETHER EXPRESS OR IMPLIED, INCLUDING \*

; BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS \*

; FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL \*

; MICROCHIP OR ITS LICENSORS BE LIABLE FOR ANY INCIDENTAL, SPECIAL, \*

; INDIRECT OR CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, HARM TO \*

; YOUR EQUIPMENT, COST OF PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY OR \*

; SERVICES, ANY CLAIMS BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY \*

; DEFENSE THEREOF), ANY CLAIMS FOR INDEMNITY OR CONTRIBUTION, OR OTHER \*

; SIMILAR COSTS. \*

; \*

; To the fullest extend allowed by law, Microchip and its licensors \*

; liability shall not exceed the amount of fee, if any, that you have paid \*

; directly to Microchip to use this software. \*

; \*

; MICROCHIP PROVIDES THIS SOFTWARE CONDITIONALLY UPON YOUR ACCEPTANCE OF \*

; THESE TERMS. \*

; \*

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; \*

; Filename: Fire Alarm Program \*

; Date: 4/06/2017 \*

; File Version: 4 \*

; Author: Arman Chowdhury and Chinmay Vaishnav \*

; Company: Macquarie University \*

; Description: Fire Alarm Program code for Assignment 2 of ELEC342, Session 1 of 2017. \*

; \*

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; \*

; Notes: In the MPLAB X Help, refer to the MPASM Assembler documentation \*

; for information on assembly instructions. \*

; \*

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; \*

; Known Issues: This template is designed for relocatable code. As such, \*

; build errors such as "Directive only allowed when generating an object \*

; file" will result when the 'Build in Absolute Mode' checkbox is selected \*

; in the project properties. Designing code in absolute mode is \*

; antiquated - use relocatable mode. \*

; \*

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; \*

; Revision History:

Started on 6/4/17 by Arman Chowdhury

Lab session 1: 15/05/17 by Arman Chowdhury and Chinmay Vaishnav

Lab session 2: 22/05/17 by Arman Chowdhury and Chinmay Vaishnav

Lab session 3: 19/05/17 by Arman Chowdhury and Chinmay Vaishnav

Final Review: 04/06/17 by Arman Chowdhury and Chinmay Vaishnav \*

AUTHOR: Arman Chowdhury

CO – AUTHOR: Chinmay Vaishnav

; \*

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Processor Inclusion

;

; TODO Step #1 Open the task list under Window > Tasks. Include your

; device .inc file - e.g. #include <device\_name>.inc. Available

; include files are in C:\Program Files\Microchip\MPLABX\mpasmx

; assuming the default installation path for MPLAB X. You may manually find

; the appropriate include file for your device here and include it, or

; simply copy the include generated by the configuration bits

; generator (see Step #2).

;

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; TODO INSERT INCLUDE CODE HERE

; PIC18F14K22 Configuration Bit Settings

; Assembly source line config statements

#include "p18f14k22.inc"

; CONFIG1H

CONFIG FOSC = IRC ; Oscillator Selection bits (Internal RC oscillator)

CONFIG PLLEN = OFF ; 4 X PLL Enable bit (PLL is under software control)

CONFIG PCLKEN = ON ; Primary Clock Enable bit (Primary clock enabled)

CONFIG FCMEN = OFF ; Fail-Safe Clock Monitor Enable (Fail-Safe Clock Monitor disabled)

CONFIG IESO = OFF ; Internal/External Oscillator Switchover bit (Oscillator Switchover mode disabled)

; CONFIG2L

CONFIG PWRTEN = OFF ; Power-up Timer Enable bit (PWRT disabled)

CONFIG BOREN = OFF ; Brown-out Reset Enable bits (Brown-out Reset disabled in hardware and software)

CONFIG BORV = 19 ; Brown Out Reset Voltage bits (VBOR set to 1.9 V nominal)

; CONFIG2H

CONFIG WDTEN = OFF ; Watchdog Timer Enable bit (WDT is controlled by SWDTEN bit of the WDTCON register)

CONFIG WDTPS = 32768 ; Watchdog Timer Postscale Select bits (1:32768)

; CONFIG3H

CONFIG HFOFST = OFF ; HFINTOSC Fast Start-up bit (The system clock is held off until the HFINTOSC is stable.)

CONFIG MCLRE = OFF ; MCLR Pin Enable bit (RA3 input pin enabled; MCLR disabled)

; CONFIG4L

CONFIG STVREN = OFF ; Stack Full/Underflow Reset Enable bit (Stack full/underflow will not cause Reset)

CONFIG LVP = OFF ; Single-Supply ICSP Enable bit (Single-Supply ICSP disabled)

CONFIG BBSIZ = OFF ; Boot Block Size Select bit (1kW boot block size)

CONFIG XINST = OFF ; Extended Instruction Set Enable bit (Instruction set extension and Indexed Addressing mode disabled (Legacy mode))

; CONFIG5L

CONFIG CP0 = OFF ; Code Protection bit (Block 0 not code-protected)

CONFIG CP1 = OFF ; Code Protection bit (Block 1 not code-protected)

; CONFIG5H

CONFIG CPB = OFF ; Boot Block Code Protection bit (Boot block not code-protected)

CONFIG CPD = OFF ; Data EEPROM Code Protection bit (Data EEPROM not code-protected)

; CONFIG6L

CONFIG WRT0 = OFF ; Write Protection bit (Block 0 not write-protected)

CONFIG WRT1 = OFF ; Write Protection bit (Block 1 not write-protected)

; CONFIG6H

CONFIG WRTC = OFF ; Configuration Register Write Protection bit (Configuration registers not write-protected)

CONFIG WRTB = OFF ; Boot Block Write Protection bit (Boot block not write-protected)

CONFIG WRTD = OFF ; Data EEPROM Write Protection bit (Data EEPROM not write-protected)

; CONFIG7L

CONFIG EBTR0 = OFF ; Table Read Protection bit (Block 0 not protected from table reads executed in other blocks)

CONFIG EBTR1 = OFF ; Table Read Protection bit (Block 1 not protected from table reads executed in other blocks)

; CONFIG7H

CONFIG EBTRB = OFF ; Boot Block Table Read Protection bit (Boot block not protected from table reads executed in other blocks)

RADIX DEC

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;

; TODO Step #2 - Configuration Word Setup

;

; The 'CONFIG' directive is used to embed the configuration word within the

; .asm file. MPLAB X requires users to embed their configuration words

; into source code. See the device datasheet for additional information

; on configuration word settings. Device configuration bits descriptions

; are in C:\Program Files\Microchip\MPLABX\mpasmx\P<device\_name>.inc

; (may change depending on your MPLAB X installation directory).

;

; MPLAB X has a feature which generates configuration bits source code. Go to

; Window > PIC Memory Views > Configuration Bits. Configure each field as

; needed and select 'Generate Source Code to Output'. The resulting code which

; appears in the 'Output Window' > 'Config Bits Source' tab may be copied

; below.

;

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; TODO INSERT CONFIG HERE

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;

; TODO Step #3 - Variable Definitions

;

; Refer to datasheet for available data memory (RAM) organization assuming

; relocatible code organization (which is an option in project

; properties > mpasm (Global Options)). Absolute mode generally should

; be used sparingly.

;

; Example of using GPR Uninitialized Data

;

; GPR\_VAR UDATA

; MYVAR1 RES 1 ; User variable linker places

; MYVAR2 RES 1 ; User variable linker places

; MYVAR3 RES 1 ; User variable linker places

;

; ; Example of using Access Uninitialized Data Section (when available)

; ; The variables for the context saving in the device datasheet may need

; ; memory reserved here.

INT\_VAR UDATA\_ACS

LED\_State RES 1

XOR\_Leds res 1

State res 1

Triggered res 1

Current res 1

ResetFlag res 1

Timer0Set res 1

; W\_TEMP RES 1 ; w register for context saving (ACCESS)

; STATUS\_TEMP RES 1 ; status used for context saving

; BSR\_TEMP RES 1 ; bank select used for ISR context saving

;

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; TODO PLACE VARIABLE DEFINITIONS GO HERE

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Reset Vector

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

RES\_VECT CODE 0x0000 ; processor reset vector

GOTO START ; go to beginning of program

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; TODO Step #4 - Interrupt Service Routines

;

; There are a few different ways to structure interrupt routines in the 8

; bit device families. On PIC18's the high priority and low priority

; interrupts are located at 0x0008 and 0x0018, respectively. On PIC16's and

; lower the interrupt is at 0x0004. Between device families there is subtle

; variation in the both the hardware supporting the ISR (for restoring

; interrupt context) as well as the software used to restore the context

; (without corrupting the STATUS bits).

;

; General formats are shown below in relocatible format.

;

;------------------------------PIC16's and below--------------------------------

;

; ISR CODE 0x0004 ; interrupt vector location

;

; <Search the device datasheet for 'context' and copy interrupt

; context saving code here. Older devices need context saving code,

; but newer devices like the 16F#### don't need context saving code.>

;

; RETFIE

;

;----------------------------------PIC18's--------------------------------------

;

ISRHV CODE 0x0008

GOTO HIGH\_ISR

ISRLV CODE 0x0018

GOTO LOW\_ISR

ISRH CODE ; let linker place high ISR routine

HIGH\_ISR

btfsc INTCON, RABIF

bra Button

btfsc PIR1, TMR1IF

call flashLeds

btfsc INTCON, TMR0IF

bra Timer

; <Insert High Priority ISR Here - no SW context saving>

RETFIE FAST

ISRL CODE ; let linker place low ISR routine

LOW\_ISR

; <Search the device datasheet for 'context' and copy interrupt

; context saving code here>

RETFIE

;

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; TODO INSERT ISR HERE

;

; State List

;

State\_Happy equ 0 ; MUST be 0

State\_Warn equ 2

State\_Evac equ 3

State\_Reset equ 4

State\_Half\_Way equ 5

State\_0 equ 0

State\_1 equ 1

State\_2 equ 2

State\_3 equ 3

State\_4 equ 4

State\_5 equ 5

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; MAIN PROGRAM

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MAIN\_PROG CODE ; let linker place main program

START

clrf Timer0Set

movlw B'00011100' ; Set clock to 250kHz

movwf OSCCON

CLRF PIE1

CLRF PIE2

clrf State ; go to State\_Happy

clrf LED\_State

clrf XOR\_Leds

clrf Triggered

clrf PORTC

clrf TRISC ; PORTC outputs

movlw 0xff

movwf TRISB ; PORTB inputs

movwf TRISA

clrf ANSEL

clrf ANSELH

movlw B'11110000' ; IOCB enabled on bits 4-7

movwf IOCB

movlw B'00100100' ; IOCA enabled RA2 and RA5

movwf IOCA

movf PORTB,W

movf PORTA,W

movlw B'00010111' ; config for stage 2

movwf T0CON

movlw B'10110000' ; initalise Timer 1, either fast or slow flash

movwf T1CON

bsf PIE1, 0 ; enables TMR1 interrupt on overflow

;

CLRF RCON

movlw B'11101000' ; enable INT0 and IOCB interrupts [ENABLE BIT 5 FOR STAGE 2]

movwf INTCON

movlw B'11110101' ; IOCB is high priority (Bit 0)

movwf INTCON2

clrf INTCON2 ; no INT1 or INT2

clrf INTCON3 ; no INT1 or INT2

movlw B'00111100' ; Buzzer Initialization

movwf CCP1CON

movlw B'00001111'

movwf CCPR1L

movlw B'01111000'

movwf T2CON

clrf ResetFlag

; TODO Step #5 - Insert Your Program Here

mainLoop

nop

bra mainLoop

Button

movf State, W

bnz NotState0Button

State0Button

movf PORTA, W

btfsc PORTA, 5

bra doEvac

btfss PORTA, 2

setf ResetFlag

btfss PORTA, 2

bra Full\_Reset

btfsc ResetFlag, 1

bra marker0

swapf PORTB,W

movwf Triggered

iorwf LED\_State

movlw State\_1

movwf State

call Set\_Time\_3Sec

bsf LED\_State, 7

call displayLeds

marker0

btfsc ResetFlag, 1

clrf ResetFlag

bcf INTCON, RABIF

RETFIE FAST

NotState0Button ;Check for State\_1

movf State, W

xorlw State\_1

btfss STATUS, 2

bra NotState1Button

bra State1Button

State1Button

movf PORTA, W

btfsc PORTA, 5

bra doEvac

btfss PORTA, 2

setf ResetFlag

btfss PORTA, 2

bra Full\_Reset

btfsc ResetFlag, 1

bra marker1

bcf INTCON, RABIF

swapf PORTB, W

movwf Current

comf Triggered, W

andwf Current, W

btfsc STATUS, 2

RETFIE FAST

swapf PORTB, W

iorwf Triggered

iorwf LED\_State

movlw State\_2

movwf State

call interrupt\_3Sec

call Set\_Time\_3Sec

call Slow\_Flash

bcf INTCON, RABIF

bsf LED\_State, 7

call displayLeds

marker1

btfsc ResetFlag, 1

clrf ResetFlag

RETFIE FAST

NotState1Button

movf State, W

xorlw State\_2

btfss STATUS, 2

bra NotState2Button

bra State2Button

State2Button

movf PORTA, W

btfsc PORTA, 5

bra doEvac

btfss PORTA, 2

bra Semi\_Reset

bcf INTCON, RABIF

swapf PORTB, W

movwf Current

comf Triggered, W

andwf Current, W

btfsc STATUS, 2

RETFIE FAST

swapf PORTB, W

iorwf Triggered

movwf Triggered

iorwf LED\_State

movlw State\_3

movwf State

bsf LED\_State, 7

bra doEvac

RETFIE FAST

NotState2Button

movf State, W

xorlw State\_3

btfss STATUS, 2

bra NotState3Button

bra State3Button

State3Button

movf PORTA, W

btfss PORTA, 2

bra Semi\_Reset

bcf INTCON, RABIF

swapf PORTB, W

movwf Current

comf Triggered, W

andwf Current, W

btfsc STATUS, 2

RETFIE FAST

swapf PORTB, W

iorwf Triggered

movwf Triggered

iorwf LED\_State

movlw State\_3

movwf State

bsf LED\_State, 7

RETFIE FAST

NotState3Button

movf State, W

xorlw State\_4

btfss STATUS, 2

bra NotState4Button

bra State4Button

State4Button

btfss Timer0Set, 1

call Set\_Time\_3Sec

btfss Timer0Set, 1

setf Timer0Set

movf PORTA, W

btfsc PORTA, 5

bra doEvac

btfss PORTA, 2

bra Semi\_Reset

bcf INTCON, RABIF

movlw State\_4

movwf State

call displayLeds

RETFIE FAST

NotState4Button

movf State, W

xorlw State\_5

btfss STATUS, 2

bra NotState5Button

bra State5Button

State5Button

call displayLeds

movf PORTA, W

btfsc PORTA, 5

bra doEvac

btfss PORTA, 2

setf ResetFlag

btfss PORTA, 2

bra Full\_Reset

btfsc ResetFlag, 1

bra marker5

marker5

btfsc ResetFlag, 1

clrf ResetFlag

bcf INTCON, RABIF

RETFIE FAST

NotState5Button

movf PORTA,W

swapf PORTB,W

bcf INTCON, RABIF

RETFIE FAST

Timer

movf State, W

xorlw State\_1

btfss STATUS, 2

bra NotState1Timer

bra State1Timer

State1Timer

bcf INTCON, TMR0IF

swapf PORTB, W

iorlw 0x00

btfss STATUS, 2

bra Evacuate

bra FalseAlarm

FalseAlarm

movlw State\_0

movwf State

call Stop\_Flash

clrf Triggered

clrf Current

clrf LED\_State

clrf XOR\_Leds

call displayLeds

movlw B'00110000' ; Turns PWM module off.

movwf CCP1CON ; So, Buzzer is off.

bcf T2CON, 2

RETFIE FAST

Evacuate

movlw State\_2

movwf State

swapf PORTB, W

iorwf Triggered

movwf Triggered

iorwf LED\_State

call interrupt\_3Sec

call Set\_Time\_3Sec

call Slow\_Flash

bcf INTCON, RABIF

bsf LED\_State, 7

call displayLeds

RETFIE FAST

NotState1Timer

movf State, W

xorlw State\_2

btfss STATUS, 2

bra NotState2Timer

bra State2Timer

State2Timer

bcf INTCON, TMR0IF

swapf PORTB, W

iorlw 0x00

btfss STATUS, 2

bra doEvac

bra StayHere

StayHere

RETFIE FAST

NotState2Timer

movf State, W

xorlw State\_4

btfss STATUS, 2

bra NotState4Timer

bra State4Timer

State4Timer

clrf Timer0Set

bcf INTCON, TMR0IF

movlw State\_5

movwf State

call interrupt\_3Sec

RETFIE FAST

NotState4Timer

movf PORTA, W

swapf PORTB, W

bcf INTCON, TMR0IF

RETFIE FAST

Semi\_Reset

movlw State\_4

movwf State

bcf INTCON, RABIF

;call interrupt\_3Sec

call Stop\_Flash

clrf Current

clrf XOR\_Leds

movf Triggered, W

iorwf LED\_State

bsf LED\_State, 7

call displayLeds

movlw B'00110000' ; Turns PWM module off.

movwf CCP1CON ; So, Buzzer is off.

bcf T2CON, 2

RETFIE FAST

Full\_Reset

bcf INTCON, RABIF

call Stop\_Flash

clrf State

clrf Triggered

clrf Current

clrf LED\_State

clrf XOR\_Leds

call displayLeds

movlw B'00110000' ; Turns PWM module off.

movwf CCP1CON ; So, Buzzer is off.

bcf T2CON, 2

RETFIE FAST

doEvac

call interrupt\_3Sec

movlw State\_Evac

movwf State

movlw 0x80

iorwf LED\_State

bcf INTCON, RABIF

call Fast\_Flash

call displayLeds

RETFIE FAST

flashLeds

bcf PIR1,TMR1IF

movlw State\_Warn

cpfseq State

bra NotStateWarn ; if (state == State\_Warn) {

flash\_State\_Warn

call Slow\_Flash ; Slow\_Flash();

bra flashCommon ; } else {

NotStateWarn

call Fast\_Flash ; Fast\_Flash();

; }

flashCommon ; both fast and slow flashing reach this point to invert and display the actual leds

comf XOR\_Leds

call displayLeds

return

interrupt\_3Sec

bcf INTCON,TMR0IF

call Stop\_Time\_3Sec

return

displayLeds

movf LED\_State, W

xorwf XOR\_Leds, W

movwf PORTC

return

Set\_Buzzer\_HighPitch

movlw B'10000000'

movwf PR2

return

Set\_Buzzer\_LowPitch

movlw B'11111110'

movwf PR2

return

Fast\_Flash

bcf PIR1,TMR1IF

movlw 0xf8

movwf TMR1H

movlw 0x5f

movwf TMR1L

bsf T1CON, 0 ; turn timer on

movlw B'00111100' ; PWM module enabled for Buzzer

movwf CCP1CON

call Set\_Buzzer\_HighPitch ; Sets High Pitch

bsf T2CON, 2 ; Toggle Buzzer

movf XOR\_Leds

btfsc STATUS, 2

bcf T2CON, 2

return

Slow\_Flash

bcf PIR1,TMR1IF

movlw 0xf0

movwf TMR1H

movlw 0xbd

movwf TMR1L

bsf T1CON, 0 ; turn timer on

movlw B'00111100' ; PWM module enabled for Buzzer

movwf CCP1CON

call Set\_Buzzer\_LowPitch ; Sets Low Pitch

bsf T2CON, 2 ; Toggle Buzzer

movf XOR\_Leds

btfsc STATUS, 2

bcf T2CON, 2

return

Stop\_Flash

bcf T1CON, 0

bcf PIR1,TMR1IF

return

Set\_Time\_3Sec

movlw 0xfd

movwf TMR0H

movlw 0x24

movwf TMR0L

bsf T0CON,TMR0ON

return

Stop\_Time\_3Sec

call Stop\_Flash

bcf T0CON,TMR0ON

;clrf LED\_State

;clrf XOR\_Leds

call displayLeds

return

END