*2021/2022*

**

**Embedded Burglar Alarm System**

Microprocessor Based Systems

EEM7016

*Simone Perfetto | 2114731*

Table of Contents

[ABSTRACT 1](#_Toc102992787)

[INTRODUCTION 1](#_Toc102992788)

[METHODOLOGY 2](#_Toc102992789)

[PROTEUS DESIGN 2](#_Toc102992790)

[CONFIGURATION IN MPLABX IDE 3](#_Toc102992791)

[USE OF MICROCONTROLLER PORTS 4](#_Toc102992792)

[IMPLEMENTING TIMERS 4](#_Toc102992793)

[Timer 0 4](#_Toc102992794)

[Timer 1 6](#_Toc102992795)

[LOOPS 7](#_Toc102992796)

[STATE MACHINES 7](#_Toc102992797)

[FUNCTIONS. 8](#_Toc102992802)

[STEPPING THROUGH THE CODE 9](#_Toc102992803)

[CONCLUSION 14](#_Toc102992805)

[REFERENCES 14](#_Toc102992806)

[APPENDIX 15](#_Toc102992807)

Table of Figures

[Figure 1) Proteus design of a burglar system. 2](#_Toc102964679)

[Figure 2) Flow Chart: TMR0 (MicrochipDeveloperHelp, 2008) 4](#_Toc102964680)

[Figure 3) TMR0 config 6](#_Toc102964681)

[Figure 4) TMR1 config 6](#_Toc102964682)

[Figure 5) Flow Chart of State Machines (C. Enrique Ortiz, 2004) 8](#_Toc102964683)

[Figure 6) Function ResetSeconds() 8](#_Toc102964684)

[Figure 7) Enumerate data type 9](#_Toc102964685)

[Figure 8) State Machine code p.1 9](#_Toc102964686)

[Figure 9) State Machine code p.2 10](#_Toc102964687)

[Figure 10) State Machine code p.3 11](#_Toc102964688)

[Figure 11) BuzzerFunc() function p.1 12](#_Toc102964689)

[Figure 12) BuzzerFunc() function p.2 12](#_Toc102964690)

[Figure 13) BuzzerFunc() function p.3 13](#_Toc102964691)

[Figure 14) BuzzerTest() function 13](#_Toc102964692)

[Figure 15) LCD config p.1 15](#_Toc102964693)

[Figure 16) LCD config p.2 16](#_Toc102964694)

[Figure 17)LCD config p.3 17](#_Toc102964695)

[Figure 18) LCD config p.4 18](#_Toc102964696)

[Figure 19) LCD config p.5 19](#_Toc102964697)

[Figure 20) LEDs config 20](#_Toc102964698)

[Figure 21) Main source file 21](#_Toc102964699)

# ABSTRACT

A burglar system must embed appropriate sets of LEDs, buttons, LCD, PIR sensor and of course the right Microcontroller in order to detect break ins and correctly send communications following the specifications required.

PIC18F452 microcontroller was chosen for this design due to its fast performance from RISC architecture, low faulty PIC percentage and power consumption and its ease in programming.

# INTRODUCTION

MPLAB® X Integrated Development Environment (IDE) is an expandable, highly configurable software program that incorporates tools that have the power to configure, develop and debug embedded designs for microcontrollers (Microchip, 2020).

This program was sided by Proteus 8 Professional software to simulate applications of a Microcontroller PICKIT and Demo board to design and develop an embedded burglar system.

Microcontrollers are small devices and therefore less powerful than computers, with limited resources. They are sized for simple tasks and thus need a programming language that is suitable to output simple tasks, that is why C programming language is often for microcontroller configuring, such as the one presented in this paper.

The system has been programmed to Activate, Deactivate, Reset, Power On/Off, Test and simulate the signal of the PIR Sensor being breached. The latter was simulated by the use of a button as a component such as a PIR sensor is not provided by Proteus therefore requires the user to build it. For the sake of keeping the task simple and focus on the main purpose, the PIR Sensor was simulated by a button that resembles the signal being breached when it is pressed, after which the set of functions in the Breach state of the system will follow as per the specifications describe.

# METHODOLOGY

## PROTEUS DESIGN

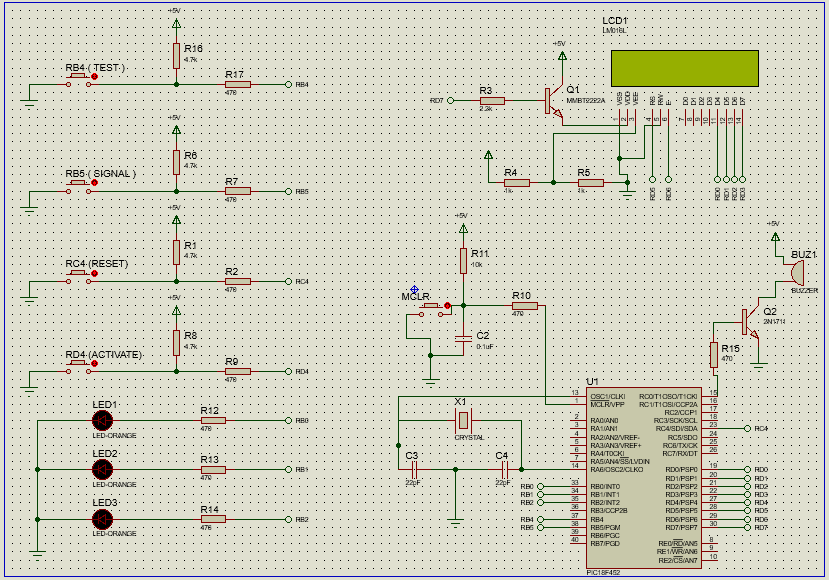


Figure 1) Proteus design of a burglar system.

The four buttons on the top left are all set up with pull down resistors in order to connect the unused inputs to ground, where the voltage is 0V, thus keeping the input LOW. This also stops inputs of digital gates from floating randomly when no input condition is called (Electronics tutorials, 2018). Using **TRISC |=0x10** will set the pin RC4 as input ( 0x10 being the hex value of 2^4 for pin 4 of a specific Port) for the button to work when pressed, this process was used for buttons Activate and Reset. Whilst Signal button, like Test and Buzzer, was manually configured as input using

“ **define Test PORTBbits.RB5** “

in the buttons header file, followed by **“ TRISbits.RB5 = 1;** “ in the main source file.

The purpose of every button in the design functions as it is described in the figure:

1. **MCLR**: when the program is started, the system starts turned off. Pressing this button turns the system on, indicated by the top Led (LED1) as it turns on until the system is switched back off. The button’s probe connects to the Master Clear Reset pin of the microcontroller, so it can be both used to reset the entire system once or turn it off completely
2. **ACTIVATE**: this button’s function is to activate the alarm when the system has been turned on. The default state of the system is the deactivate state, the LCD will indicate this by printing “**DEACTIVATED**”.

In the Activated state, the LCD will print “**ACTIVATED**” on the screen while the LED2 turns on and starts blinking for 6 seconds. When only two seconds are left in the 6 second timer, the LCD will clear. The PIR Sensor is considered enabled, therefore the user needs to either reset the system or exit the premises before the timer runs out.

1. **SIGNAL**: when the 6 seconds run out, the LCD will print “**ALARM ON**” to communicate that the system is in alert mode for any break-in, in this state the LED2 will stay permanently on until reset.

Pressing the signal button will simulate that a person has been detected within the range, and will send the system into “**SENSOR TRIGGERED**” state, making the LED3 blink and the buzzer sound an intermittent sound every second for 5 seconds.

When the 5 seconds run out and the system has not been reset, the alarm will confirm the breach and enter the “**BREACH ALERT**“ mode. The buzzer will output an alarm for 5 seconds before turning of for the same amount of time and repeating until the system is reset or turned off, while the LCD prints out each of those states.

1. **RESET:** the system can be reset at any time by pressing the reset button. When this is done, the LEDs will turn off with the buzzer and the LCD will clear to print “**DEACTIVATED**” again. This way, the system returns to its deactivated state.
2. **TEST:** in order to make sure the system works without turning the PIR sensor on, a test function was implemented. When the relative button is pressed by the user, the LCD will print “**WARNING**” while the buzzer plays a 5 seconds sound with 5 seconds break until the system is reset with the respective button that holds that function.

## CONFIGURATION IN MPLABX IDE

For the embedded system to work as described the microcontroller had to be configured as follows:

**Crystal Oscillator**: this is the resonator of microcontrollers made from high-quality quartz crystal wafers creating electric signal at square or sine wave with the chosen frequency. It provides an internal clock for the microcontroller and its frequency will determine the number of instructions given to the microcontroller every second. The crystal was set to the standard XT mode with a 4MHz frequency (one million instruction per second given to the microcontroller). Different resonators can be used such as ceramic resonators, but having a crystal oscillator provides a higher Q factor, stability, accuracy and more customizable frequency and range (the crystal has more modes available than the ceramic resonator).

The PIC microcontroller was then configured by turning off Watchdog timer, brown out reset and low voltage programming while setting the oscillator in XT mode, which allows for a range from 200kHz to 4MHz.

Two capacitors at 22pF are used in parallel to the crystal to resonate with the crystal’s inductance allowing it to oscillate at the chosen mode.

## USE OF MICROCONTROLLER PORTS

The ports are the collection of pins of that same register that connect the microcontroller to the embedded devices and configured as either input (logic value 1) or output (logic value 0).

The PORT or LAT write operation writes to the LAT of that port so these operations are seen as the same.

On the other hand, when a read operation occurs, the PORT reads the physical state of the voltage level (known as the physical state) of the pins while the LAT reads the LAT register.

**PORTS**

* **PORT A:** this port was left unused expect for pin RA6 for its alternate function as OSC2 pin to connect the oscillator to the PIC microcontroller.
* **PORT B:** this port has been used for both inputs and outputs as the first three pins (RB0 : RB2) have been programmed as outputs for the LEDs to turn on, off or blink as per the specifications of the system. On the other hand, the pins RB4 : RB5 have been programmed as inputs to implement the buttons required for the signal and testing.
* **PORT C:** pins RC1 and RC4 were connected to buzzer and reset button respectively.
* **PORT D:** this port was applied to connect the pins of the LCD to the microcontroller as follows:

**VSS --> GND**

**VDD --> +5V**

**VEE --> RD6**

**RS ---> RD5**

**R/W ---> GND**

**EN ---> RD6**

**D4 ---> RD0**

**D5 ---> RD1**

**D6 ---> RD2**

**D7 ---> RD3**

**A (+LED) ----> +5V**

**K (-LED) ----> GND**

## IMPLEMENTING TIMERS

The PIC18F452 microcontroller has internal clocks, thanks to the oscillator, which can be manipulated to our advantage to set up timers in order to run multiple required instructions at the same time, as a better alternative to using Delay functions which would interrupt the program to run only one task at a time.

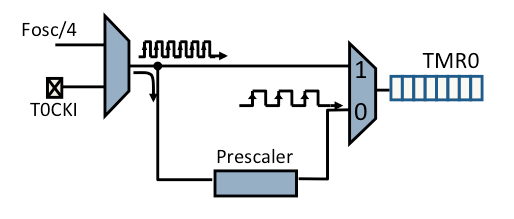


Figure 2) Flow Chart: TMR0 (MicrochipDeveloperHelp, 2008)

Each timer uses its own register:

1. **Timer 0, TMR0.**
2. **Timer 1, TMR1.**

### Timer 0

In order to use T0 effectively, the Interrupt (**TMR0IE**) has to be enabled and Interrupt Flag (**TMR0IF**) ahs to be given a starting value of 0, meaning that TMR0 has not overflowed.

Next, the register was set up as follows:

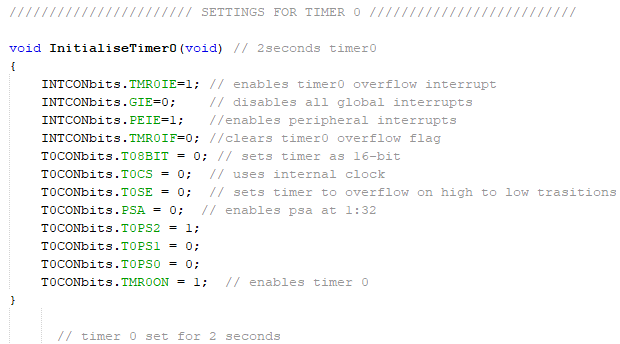


Figure 3) TMR0 config

With these settings, the timer is set to overflow every 2 seconds. Since the required amount is 4 seconds and 6 seconds for the functions needed in the specifications, the timer overflow condition can simply be multiplied by 2 and 3 in the “ if – else “ statements, so that the timer will overflow 2 and 3 times respectively before the function is called.

### Timer 1

This timer was applied to make a 1 second timer overflows in order for the required LEDs to blink 1 every second until required to turn off or stay on. For the latter, Timer 0 was applied as explained previously. Differently from TMR0, TMR1 is a 16-bit timer only, thus its respective register was set up as follows.

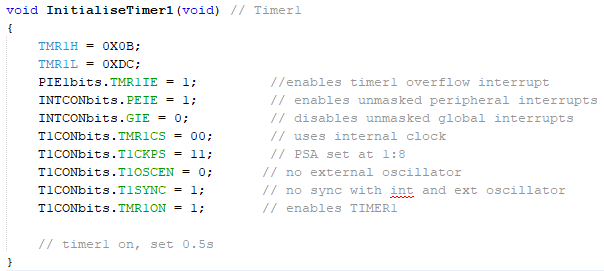


Figure 4) TMR1 config

with these settings, TMR1 is set at 500ms, which when multiplied by 2 in the “if-else” statement conditions it will overflow twice before the function is called, adding up to a second.

For both timers, simulator tool and breakpoints were used to test that the exact amount of time had passed before the program stops at the breakpoint.

## LOOPS

Loops are a programming element that repeat a portion of code a set number of times until the desired process is complete (Jon Mattingly, 2022)

They are the fundamental part on the programming side of this project that makes everything work as it should. When the system is deactivated, the microcontroller keeps being instructed on what step it needs to take next when an embedded device, configured as input, changes state. For example: when a button is pressed the microcontroller is instructed to light the LEDs up.

The only loop required for this project was the **WHILE** loop: every code within this loop get repeated after the last line has occurred, unless a break statement is placed, in which case the compiler exits and proceeds to the next line of code outside the while loop.

## STATE MACHINES

After the first while loop in the main source code file, when the function

“ **DoStateMachine**(); “ is called, the rest of the code that assembled the system holistically is present inside a state machine model. State machines are coded behavioural models consisting of a finite number of states which determine the output and transition that the machine performs form one state to the next.

The burglar system has been divided into the following states:

## DEAC\_STATE - Deactivated state.

## ACT\_STATE – Activated state

## ALARM\_STATE – Alarm On state

## The next states such as System Triggered and Breach Alert have no been set as actual states but have been implemented within the function “**BuzzerFunc**() “.

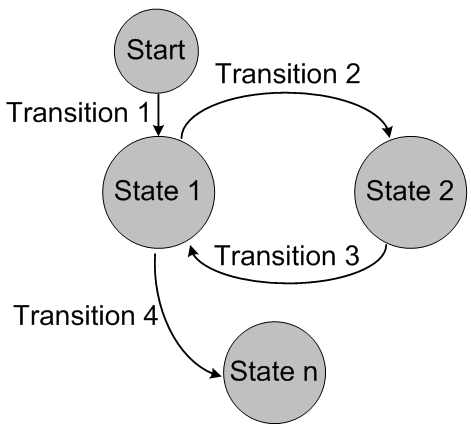


Figure 5) Flow Chart of State Machines (C. Enrique Ortiz, 2004)

## 

### FUNCTIONS.

Functions help make the code much neater and give it more structure than programming the same lines of code. Calling a function makes it possible to repeat a collection of lines of code everytime they are needed, giving that function a specific job and saving time.

Example:

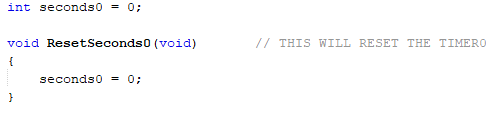


Figure 6) Function ResetSeconds0()

Where seconds is a variable collecting the counter from TMR1, this function resets the timer by setting the value of the counter back to zero. This type of function requires no arguments and need to return no value, hence the two “ **VOID** “ statements.

For a function to be called in another source code file, a header file must be created with a prototype version of the function in order for it to be called, granted the header file has been included with the appropriate preprocessor “**#include**”.

## STEPPING THROUGH THE CODE

The states are collected in the data type ‘**enumerate’**, collecting a set of values called members/constants which can be used with variables.

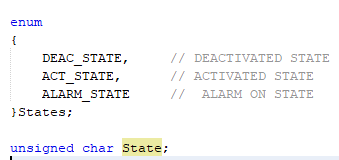


Figure 7) Enumerate data type

Each state is called inside the “ **void DoStateMachine (void)** “ function, when the respective stage of the alarm is met, through a set of “ if-else “ statements:

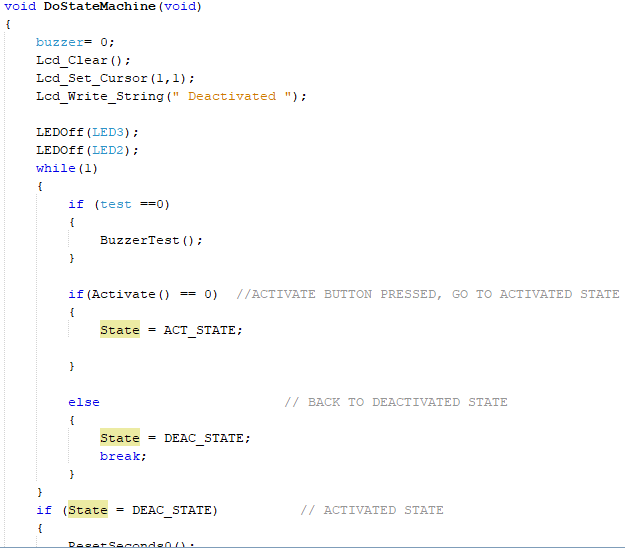


Figure 8) State Machine code p.1

the while loop repeats the entire block of code so that when the state changes the compiler makes the transition to the next block of code correctly.

* If the test button is pressed, the test state is triggered.
* If the activate button is pressed, the activated state is triggered.
* If neither of the buttons are pressed, the system stays in the deactivated state.

Every state consists of its own block of code for their specific purpose and functionality, as shown in Figure 9:

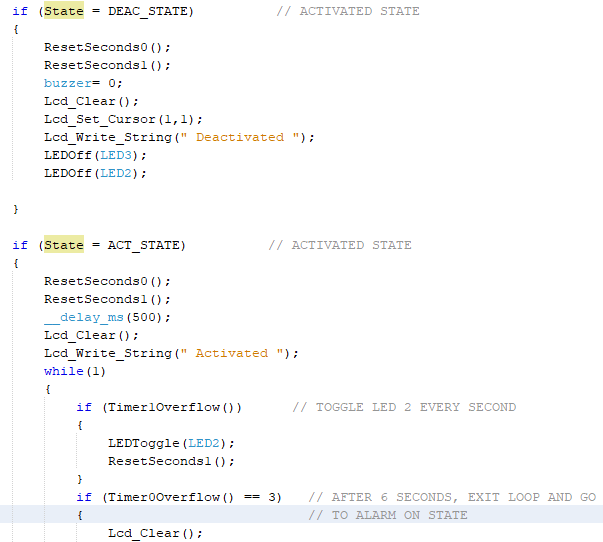


Figure 9) State Machine code p.2

This sets the Activated state to reset the timers and clear the LCD from the “**Deactivated**“ message to then print “**Activated**”. Another while loop consists of a set of “if-else” statements that make the LED2 blink before clear the LCD to step on the next state when TMR0 overflows 3 times, shown in Figure 10.

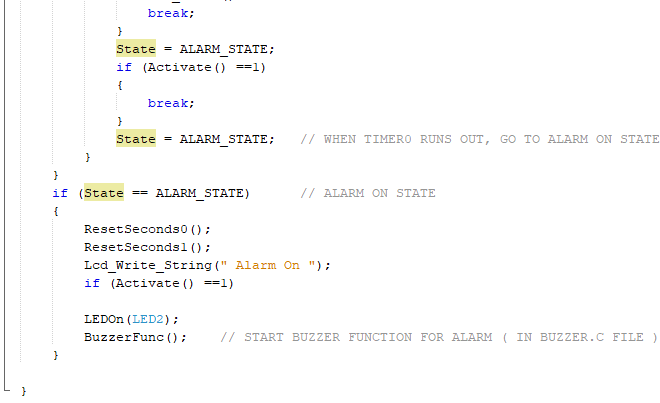


Figure 10) State Machine code p.3

At Alarm On state the timers are reset again and the LCD prints “**Alarm On**” while the LED2 is turned on permanently before calling the **BuzzerFunc**() function containing the remaining code for the case where the system has not been reset to its default state.

Inside the **BuzerFunc**() function, the buzzer pin is set as an output in order for it to stay quiet until the alarm is set off. Another set of “if-else” statements set conditions based on timers and again for the LEDs to blink and the buzzer to emit sounds as the specifications require, in both System Triggered and Breach Alert states, unless the Activate/Deactivate button is pressed again, setting the system back to default state.

If the system is not deactivated before the Breach Alert stage is reached, the only way to return to default state will be by pressing the Reset button, programmed as a set of **Reset1**() functions which contain **break** statements to exit every **while** loop in the code., demonstrated in **Figure 11.**

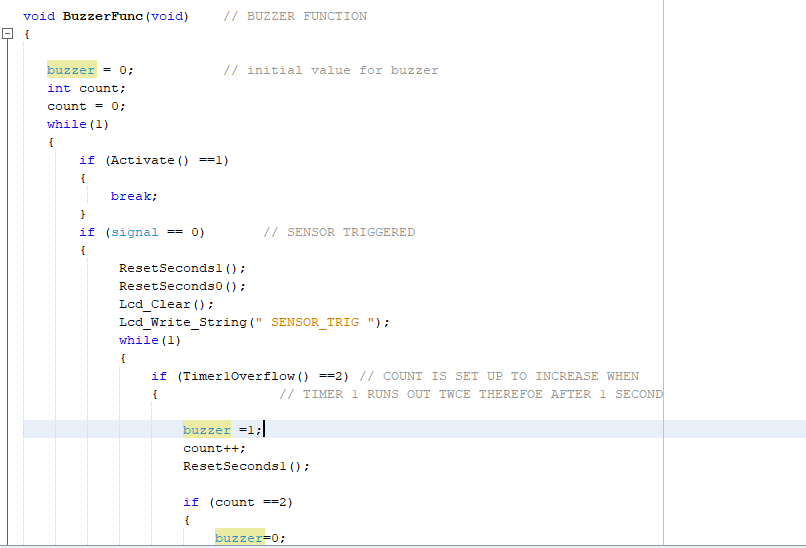


Figure 11) BuzzerFunc() function p.1



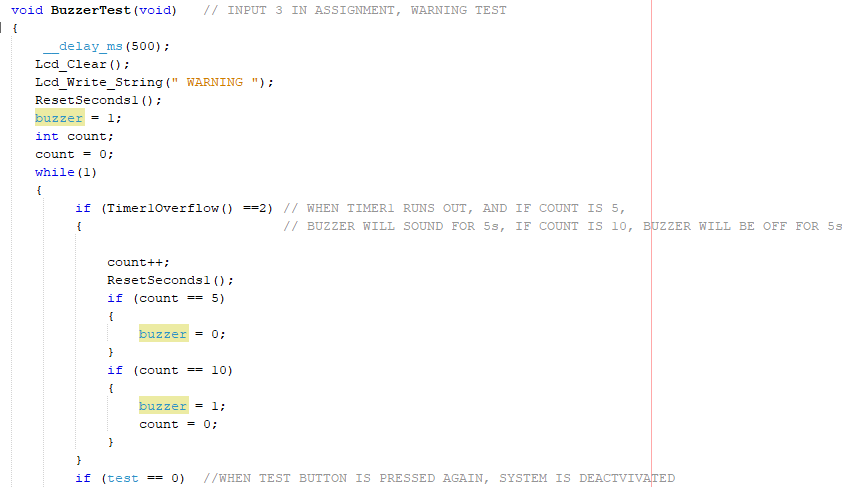
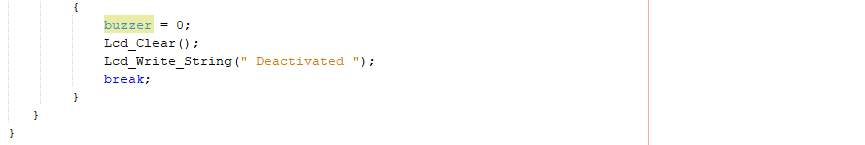
Figure 12) BuzzerFunc() function p.2

# 

Figure 13) BuzzerFunc() function p.3

The final mode of the system to now look at is the Reset mode.

Figure 12 shows the function **BuzzerTest**() which starts with clearing the LCD to print the “**WARNING**” message before resetting the timer and setting the buzzer to 1, which makes the buzzer sound go off immediately after the **Test** button is pressed The “**if**-**else**” statements fulfil he same purpose in the **BuzzerFunc**() function except the buzzer will sound for 5 seconds on and off intermittently until the Reset button is pressed to return to the default state of the system.

 Figure 14) BuzzerTest() function

# CONCLUSION

With the proper use of C language and proteus designing it was possible to develop a fully working burglar alarm (PIR sensor omitted) respecting the specifications required. The reset button was originally meant to be pressed for 4 seconds before it would reset the system fully, although it makes little difference to the output and time consumption. Experience with timers showed to be necessary in order to fulfil every task and although TMR0 could have been enough, the configuration of TMR1 allowed for flexibility with the timers to fulfil specific duties, which came in handy.

Some issues were encountered with making the loops repeating some of the code for buttons which made the functions start prematurely or randomly, this was solved by adding some short delays in the program, although debouncing can be a better alternative.

# REFERENCES

1. Ibrahim, D. 2006;2011;, PIC BASIC projects: 30 projects using PIC BASIC and PIC BASIC PRO, [Rev.].;2; edn, Elsevier, London;Amsterdam;.
2. Morton, J. 2001, PIC: your personal introductory course, 2nd edn, Newnes, Oxford.
3. Tina, K.G., Bhadra, R. & Srinivasan, N. 2007, "PIC: Protein Interactions Calculator", Nucleic acids research, vol. 35, no. Web Server issue, pp. W473-W476.
4. Margush, T.S. 2012;2011;, Some assembly required: assembly language programming with the AVR microcontroller, CRC Press, New York;London, [England];Boca Raton, [Florida];.
5. Langbridge, J.A. 2015, Arduino sketches: tools and techniques for programming wizardry, John Wiley & Sons, Indianapolis, Indiana.
6. Morton, J. 2005, The PIC microcontroller: your personal introductory course, 3rd edn, Elsevier/Newnes, Amsterdam;Boston;.
7. Iovine, J. 2004, PIC microcontroller project book: for PIC Basic and PIC Basic Pro compilers, 2nd edn, McGraw-Hill Professional, London;New York;.
8. Katzen, S. 2001, The quintessential PIC microcontroller, Springer, London.
9. Predko, M. 2007, Programming and customizing the PIC microcontroller, 3rd edn, Tab, London;New York;.
10. Nussey, J. & Books24x7, I. 2013, Arduino for dummies, 1st edn, John Wiley, Chichester.
11. Ibrahim, D. & EBSCO ebooks 2002, Microcontroller-based temperature monitoring and control, Newnes, Oxford.
12. Schwartz, M. & Mohanta, P. 2014, Arduino networking: connect your projects to the Web using the Arduino Ethernet library, 1st edn, Packt Publishing, Birmingham, [England].
13. Iovine, J. 2000, PIC microcontroller project book, McGraw-Hill, London;New York;.
14. Ibrahim, D. 2005, Microcontroller based applied digital control, John Wiley, Chichester.
15. Morton, J. 2005, The PIC Microcontroller, Elsevier Science & Technology, Jordan Hill.
16. Smith, J.R. & Books24x7, I. 2005, Programming the PIC Microcontroller with MBASIC, Elsevier Science & Technology, Oxford.

## 

## APPENDIX

Word count: 2401

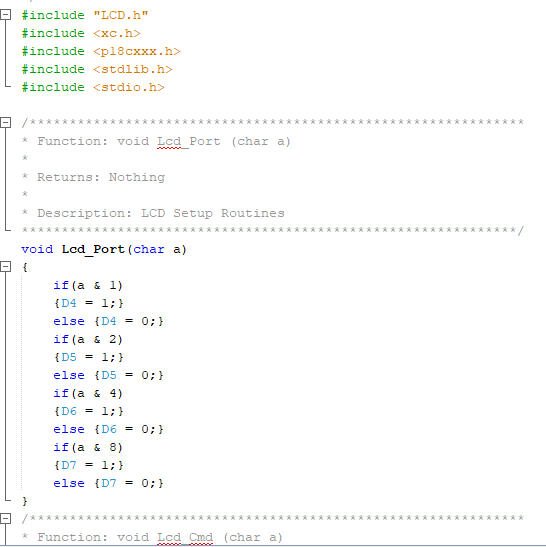


Figure 15) LCD config p.1

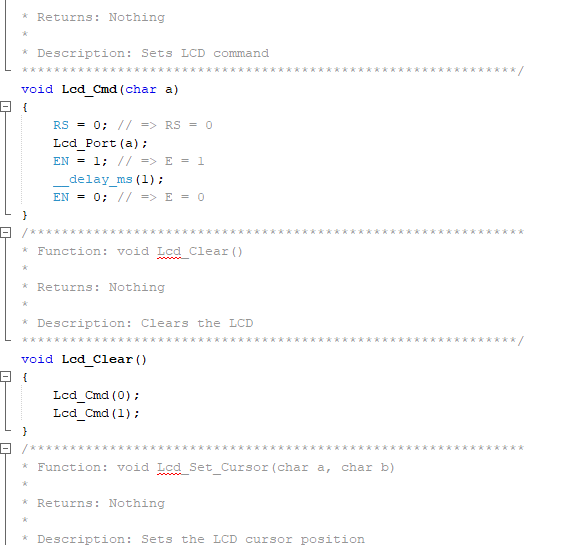


Figure 16) LCD config p.2

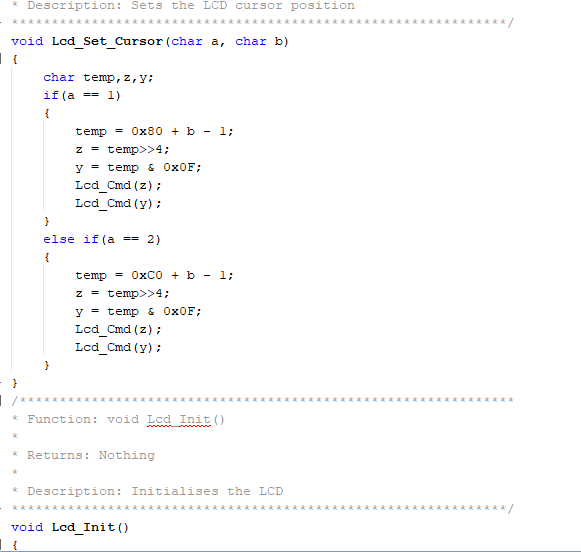


Figure 17)LCD config p.3

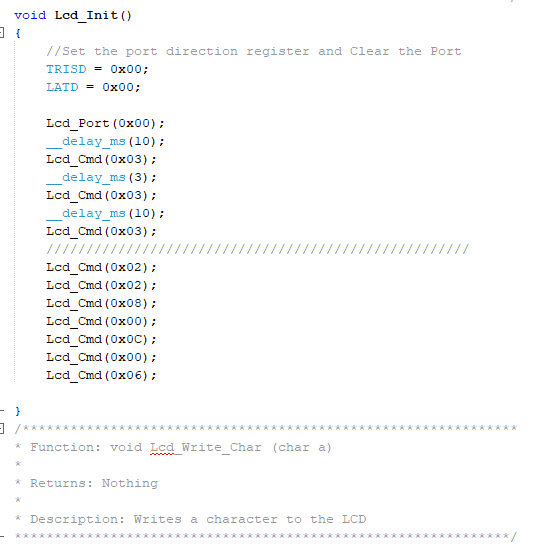


Figure 18) LCD config p.4

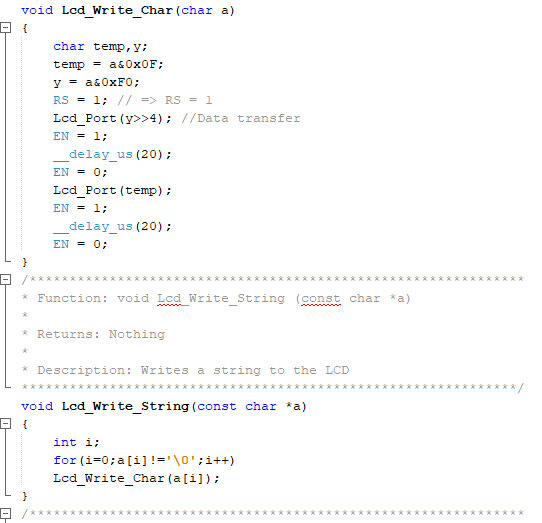


Figure 19) LCD config p.5

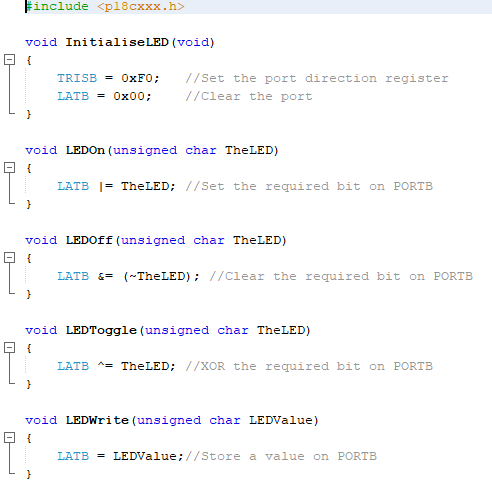


Figure 20) LEDs config

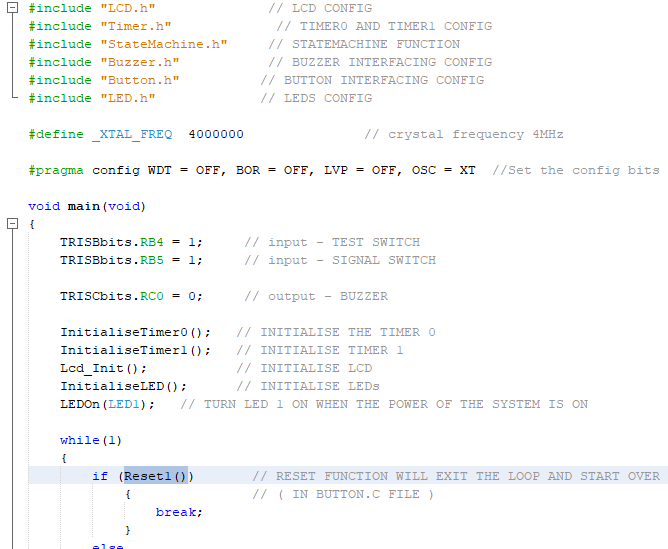
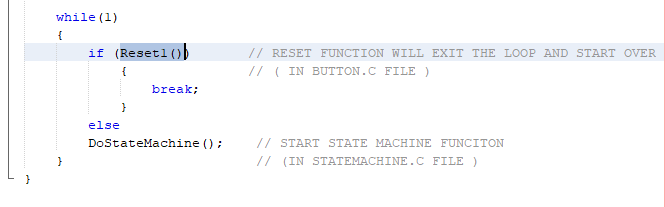


Figure 21) Main source file