3-Zone Alarm System – Atmega32A

Introduction To Embedded Systems
EEET2256

Aiden Contini – S3780445
Thirandi Fernando – S3823593
15.10.2021

1 Executive Summary

This project offers an exploration into the different approaches which can be taken when developing real-world applications for the ATmega32A (8-bit) microcontroller. This project involved the design and construction of a 3-zone alarm system similar to security systems which are present in countless buildings around the world. The primary aim of this venture was to create a fully functioning alarm controller able to simulate the functions that a user would expect to be present in a finished retail product. The secondary object was to develop an understanding of the pros and cons that exist when choosing to design and write microcontroller code in C compared to Assembler, and objectively uncover when one approach may be better than another. Overall, this project found that the scalability and ease granted when coding in C far outweighed the level of control that exists when coding in Assembler, however it did provide an insight into the times when one approach may be more favorable to the task at hand.

Table of Contents

2 Introduction

This project dealt specifically in designing a user-interactable 3-zone alarm system that makes use of the hardware available on the OUSB-IO board. This alarm system was to use the 16-key matrix keypad (Appendix 3) to be the main form of input to the microcontroller to the outside world. In this way, the first three columns were made available to a user, and these were to be used to enter and change alarm passcodes, where the fourth column was to be used to simulate "triggering" the alarm.

The approach taken to develop this alarm system involved allowing a user to utilize all ten numbers on the keypad as a part of their passcode, with the hash and asterisk keys utilized to specify system functions. In the C implementation, the hash key was used when the system is unarmed to enter the "change passcode" mode of the system. The Assembler code had similar functionality, but also took advantage of the asterisk key as well. In the fourth column, the buttons 'A', 'B' and 'C' were to be used to simulate different zones being triggered. This was to mimic the way an alarm system may work on a large area, as it is very important for people monitoring the alarms to know exactly what area motion may have been detected, triggering the alarm. Putting a system such as this in place makes monitoring the area much easier so it was important to ensure the microcontroller could communicate to a user exactly what was happening on a system level.

The main form of communication from the microcontroller to a user was to be through the eight on-board LEDs. This unfortunately prevented the quality of information that could be presented to a user (displaying of numbers was restricted to their binary equivalent). It was important that an effective and well thought out user interface was developed with the hardware constraints in mind, to ensure that a user would be able to understand the usage of the microcontroller with little confusion. The LEDs were used to output innumerous amounts of information including each keypad value pressed, the state of the alarm (armed/disarmed), the triggered zone when the alarm is triggered and to acknowledge the success (or failure) of inputs when they occur.

The scope of this project involved developing this solution in both the Assembler and C programming languages, taking the time to analyze the ways in which either approach may be of benefit or detriment. Microchip Studio 7.0 was used extensively alongside the datasheets for the microcontroller and OUSB-IO board to develop and analyze both solutions effectively and reach relevant conclusions surrounding the code implementations [1] [2].

3 Background

In the present day, microcontrollers are used everywhere, from computer keyboards to implantable medical devices since they are exceptional at process control. During this project, we focused on the use of microcontrollers in security systems, another application that's dominant in today's world. Security systems comprise of many input sensory devices and outputs such as flashing lights and sound alarms. The focus of this three-zone alarm system was to develop a working security system that monitors 3 different locations, with time outs, delays, inputs, and outputs. This design can be implemented in work buildings, hospitals or even in personal residences, as it is easy to use and keep track of. Gathering knowledge from lectures, laboratories and tutorial classes previously held, this system was designed using advanced processes such as timed interrupts and delays for user convenience [3] [4]. The fourth subsection of this report, titled "Technical Work and Results" go through a detailed overview of the design development headway.

4 Technical Work and Results

To begin this project, a great deal of effort was put into planning the way in which the code would be written and ensuring that both the C and Assembler code would operate in the same or similar way. This planning phase did not consider the limitations either approach but rather acted as an overview of how the code would function. A flow chart of the overall code was developed to act as a point of reference while writing the code and this is featured in Figure 1. This flow chart outlines the exact minimum requirements for the code functionality and the process involved in achieving these outcomes.

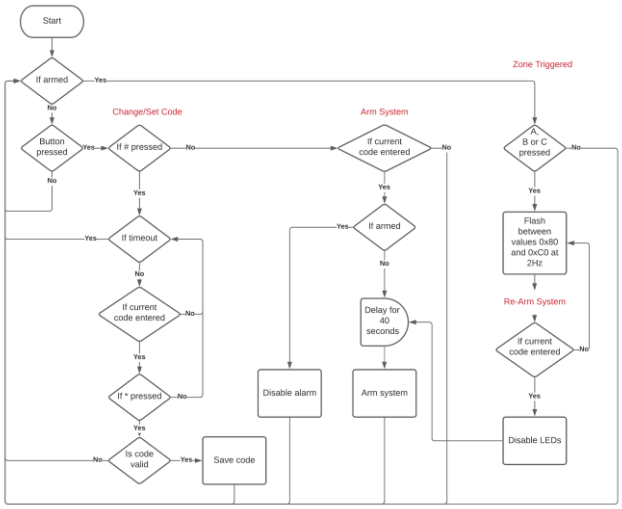


Figure 1. Flow chart outlining the overall code function

A description of what the flow chart outlines is as follows:

- The system begins in the disarmed state and waits for user input. It checks for either the hash key or the correct code being entered.
- If the hash key is entered, the code will wait for the correct code entry. After registering this it will allow the user to set a new code.
- If the correct code is entered, the system will be put into the armed state.
- Once the system is in the armed state, it waits for either the correct code being entered, or a zone being triggered.
- If a zone is triggered, the LEDs should flash to simulate an alarm.
- If the correct code is entered at any point while the system is armed, the system will become disarmed.

This flow chart identifies 3 main features of the system, and the following sections will describe how these features were achieved in both Assembler and C. The main functions of this system are:

- Arm/Disarm the system
- Allow for the code to be changed
- Simulate the alarm being triggered

4.1 Assembler Code

Developing the assembler code for this project was a challenge involving limitations in storage space for variables and restricting linear structure. The most basic of the program is to be able to read the keypad entries, so initially, code previously written in past laboratories were transferred on to this project. This code reads the keypad and maps it to the correct integer value that corresponds to the pressed key on a table.

Once this basic function was complete, the main portion of the code was structured using assembler "labels". When the board is first powered up, the program ensures that all registers are clear, and the input/output ports are configured as per the requirements. During this phase, the alarm system is set to be *disarmed*. Then it moves on to a main loop that runs until the device is turned off. In this loop, user input is constantly checked until a keypress has been detected. Once a keypress has been noticed, this loop is exited depending on the systems armed status. Next, the relevant parts of code for *armed* and *disarmed* states are run.

First considering the piece of code that runs during the alarm's *disarmed* state; the program checks if the pressed button is the hash (#) key. If it is, then it proceeds to check for the current code to be input. After the user enters the correct code, they must press the star key before entering the new code (4 digits only). If the incorrect passcode was initially entered, then the program returns to the main loop where it keeps checking for a user input to be triggered. Continuing to the next part of the program, if the hash key was not registered, the user input is examined for the first digit of the correct code, then the rest of the current code, subsequently arming the system. In case of an incorrect passcode being entered, the program returns to the main loop where it keeps checking for user input (refer figure 2).

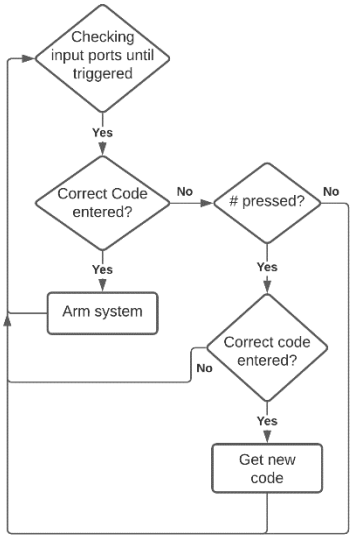


Figure 2. Process map when system disarmed.

Now moving on to the part of microcode that runs when the alarm is *armed*: the algorithm checks for either Zones A, B or C to be triggered. If any zones are detected, the program enters an infinite loop where it remains until the correct passcode has been entered. Within this, the bits 0 to 3 of PORTB are used to display the detected zone and bits 4 to 7 are used to display the strobe light and siren going off. Bit 6 was used to simulate a strobe light going off; it flashes at 2Hz 4 times to indicate this. Proceeding this, the LEDs will show which zone was triggered, the armed state of the alarm and the alarm going off, as described in the figure below:

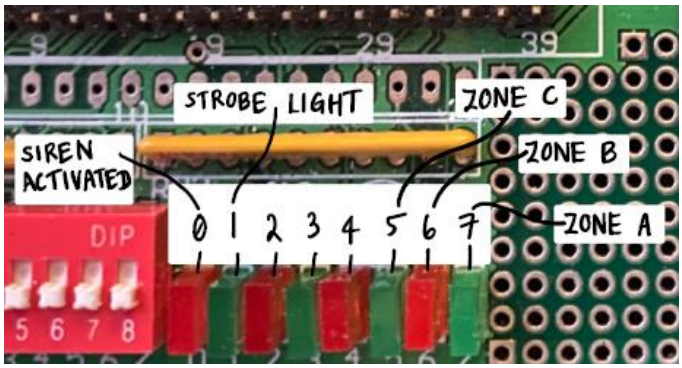


Figure 3. LED descriptions, when the alarm is triggered (the bits are mapped in reverse on the OUSB board, which is how the bits have been shown in this image.)

Inside the forever loop when the alarm is triggered, input is continuously checked, until detected, and if the correct code was entered the system will proceed to the armed state but disable all sirens and alarms that were set off earlier. When no zones are activated, input by the user is analyzed as normal. If hash was pressed, the user can change the passcode in the same manner as earlier mentioned, else simply entering the correct code will disarm the system. If any incorrect code was detected, the program will move back to main loop checking for any inputs, while remaining in the current state (armed/disarmed). Before the code initiates this loop, the alarm is first set off and the occupants are given 40s to leave the area.

This project was developed using the fully functionality of the OUSB board's LEDs. This code was written in such a way that when the user presses the keypad, this number, letter, or symbol is represented in their decimal format on the last 4 LEDs (bits 4 to 7), refer figure 2. It also needs to be mentioned that when the system is not triggered, bits 0 and 1 (as seen in figure 2) are used to display the armed/disarmed status of the alarm system. Only bit 0 will light up if the system is offline, and only bit 1 will be shown if the system is armed.

The above sections the main functionality of the code. Certain limitations emerged during the development process, the first being restricted use of memory. When developing assembler code, it was essential that the registers used are not overlapped, and since there were many variables for this system, the stack-pointer was shown useful. Temporary variables were pushed and popped, using them for various user defined functions. Most of these functions were developed in a reusable manner. Registers 1-15 were used for storing the input code and counters, whereas registers 16-31 were optimized for returning user inputs and large delay counters. The delays were generated using the internal clock time of the OUSB board, 12MHz, giving a period of 83.3nS [5].

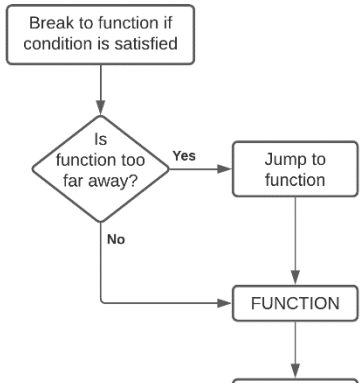


Figure 2. Process map for "Jump" Functions

Further, another obstacle faced during assembly code development was the limiting range of break statements. To call functions that were out of range, additional "jump" functions were written, shown in the flowchart in figure 4.

Reflecting on the code, there are a few discrepancies were noticed. First is that there is no "time out" for the entering of the passcode by the user, as any advanced security system must have. Another inconsistency observed is, that when disarming the system, if the passcode has the digit "0" in it, only the system-armed LED is shown, as the decimal representation of this digit is zero itself, therefore emptying out the last few LEDs. This maybe misleading to the user.

4.2 C Code

To develop the C code, a portion of code previously created to read keypad inputs was ported over to this project. This code had the underlying functionality of reading each key press from the 16-key matrix keypad and mapping the values to their ASCII equivalents. This code then had to be built upon to achieve the full functionality required in the project.

Numerous decisions had to be made when implementing the various functions of the system and these will be discussed in detail here.

The first part of functionality outlines was to ensure a reusable block of code was created to authenticate the correct code being entered. A requirement of this was to be reusable because authenticating the correct code is something that will occur throughout all the major functions of the code. To achieve this a function was created which reads in and stores each consecutive value entered and if these 4 values correctly match the current code it returns success. This code is written in such a way that it ensures the integrity of the order of values entered. An important decision made in this section is how to handle erroneous conditions such as a user only entering 3 values instead of 4 or entering the incorrect code. To handle this, interrupts were used to place a 5 second timeout on any time the correct code was required. This ensured that no matter the conditions, the correct code would continue to be searched for until either the correct code was entered, or the timeout occurred. This functionality can be seen in the flow chart in Figure 5.

To obtain this 5 second delay, some calculations had to be done and some leeway was granted to approximate this delay, as accuracy was not the most necessary in this case. By slowing the speed of timer 0 down to $\frac{1}{1024}$ times its original speed, 1 timer count was found to take 85.3µs. This meant it would take 0.02 seconds for the timer to count from 0-255 and flip the overflow flag. Applying this to a delay of 5 seconds, the overflow flag would have to be flipped 230 times before this delay was completed. To implement this, a variable was set to increment every time the overflow flag was set by the interrupt and once this variable reached 230 a timeout would occur.

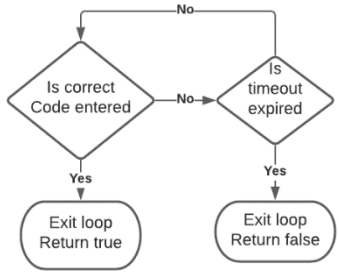


Figure 3. Flow chart depicting the process of checking the correct code

Once checking for the correct code was completed, extending this code to correctly arm and disarm the system was a simple task of checking for the correct code and flipping the current state of the system.

The next phase was to ensure the user was able to easily change the code of the system. A decision was made to ensure that this feature was only available when the system is in disarmed state. This was to ensure that once the system is armed, no tampering could potentially occur to the controller. To change the code first the hash key must be entered followed by the system's current code. This then instructs the system that an authority is attempting to change the code, and the next 4 keys entered will be the new code. The code will stay in this position waiting for 4 key entries until they have occurred.

This section brought to attention the necessity of using the on-board LEDs to inform a user of the state of the system. This is a way to provide feedback to the user of what values they are pressing. To ensure that this was taken advantage of the system was set up to display the ASCII value (in binary) of every key that was

pressed in the upper 4 bits of the microcontroller. This allowed for interaction with the user as it meant that the user could see exactly what values they were entering. This was deemed very important when setting a new code, as a user should be able to identify and double check what values they are entering to ensure that they don't enter incorrect values.

Finally, the last function to implement was the simulation of an alarm being triggered. While the system is armed, the system will continue to check the keypad for input. If a key was pressed, it would check the value of this key and if it were the A, B or C keys the system would become triggered. The code would continue to be triggered until the correct code is entered, which then stops the alarm and returns the system to the armed state. The system can also be disarmed without the alarm being triggered. A flow chart of this functionality is displayed in Figure 6.

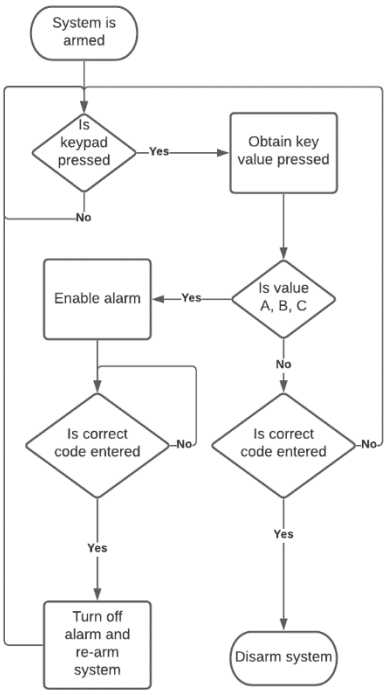


Figure 4. Flow chart depicting the system when it is in the armed state

One discrepancy was found when implementing this part of the code compared to the design requirements, and this surrounded the requirement of having the upper 4 LEDs flash between values of 0x8 (1000b) and 0xC (1100b) at a rate of 2Hz. This functionality was mapped out originally, but it was found difficult to implement this using interrupt sources while there is already another interrupt on timer 0 (Figure 5) in place. To utilize both these interrupts at the same time would require a nested interrupt which is an advanced concept out of the scope of this project. If this were to be implemented, the way it would be done would be to slow the time down to $\frac{1}{1024}$ times its original speed. This would mean that the timer would overflow every 0.02s, meaning that this overflow would have to occur 23 times before changing the value on port B. As this was not able to be achieved, instead when an alarm was triggered a static value was placed on port B instead. The lower 4 bits would be set to 0011b, and the upper 4 bits would be set to either 0x8, 0x4 or 0x2 for the zones A, B and C respectively.

Another user interface decision was also made involving signifying to a user when the system is moving between different phases of the functions. This would be to flash all LEDs (0xFF on port B) at a rate of 2Hz 4 times. This would occur any time the system was switching between different states, such as when the system is being armed or disarmed. Finally, the system would also use the lower 4 bits LEDs to statically display the current state of the system. This means displaying 0x01 for armed, and 0x02 for disarmed on the LEDs. As this would use the lower 4 LEDs, this value would be independent on any altercations happening to the upper 4 LEDs (such as displaying the value of the key pressed).

4.3 Overall Results

4.3.1 Assembler Code

The assembler code was tested on the physical OUSB board, and the following images summarize the seen outputs.

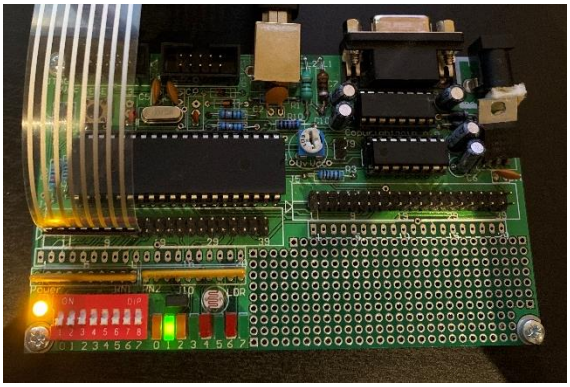


Figure 7. The LED display when the alarm system is armed, and no zones have been triggered.

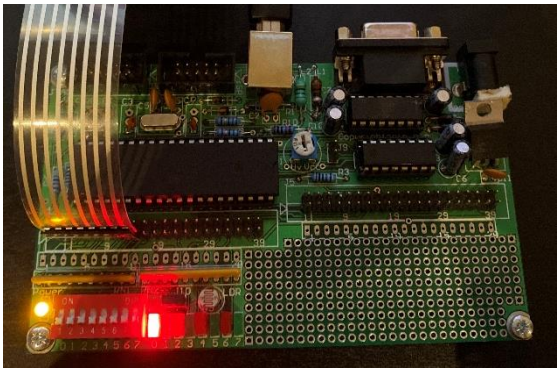


Figure 8. The LED display when the system is disarmed.

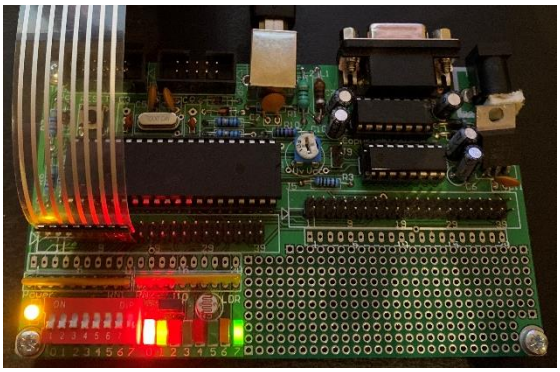


Figure 9. The LED display when Zone A has been triggered (while system was armed).

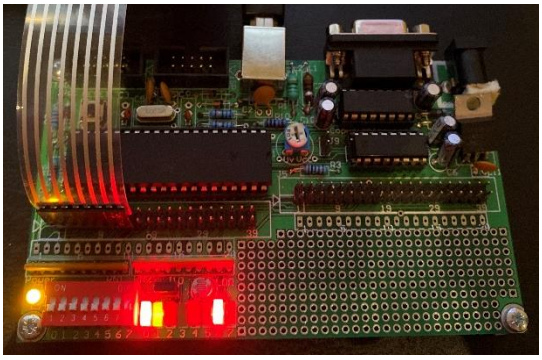


Figure 10. The LED display when Zone B is triggered (while the system is armed).

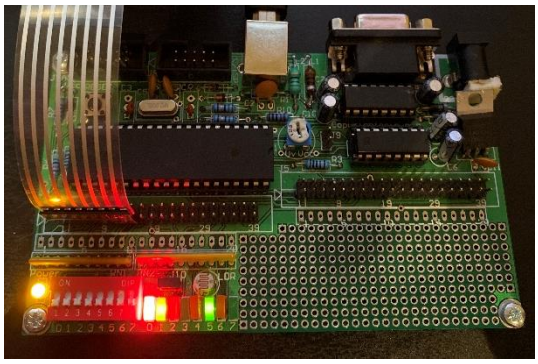


Figure 11. The LED display when Zone C is triggered (while the system is armed).

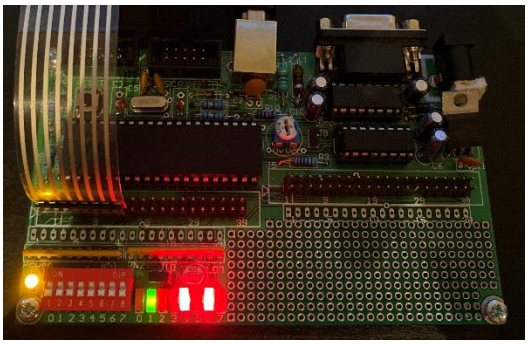


Figure 12. The LED display when the system is armed, and you enter a value. In this scenario, 5 has been pressed, which has been shown in the last 4 LEDs.

4.3.2 C code

Testing the C code for this project was carried out through simulation via Microchip Studio, the images below summarize its basic functionality.

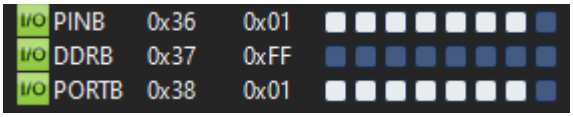


Figure 13. This screen capture shows the action of PORTB (i.e., the LEDs) when the system is armed, and no zones have been triggered.

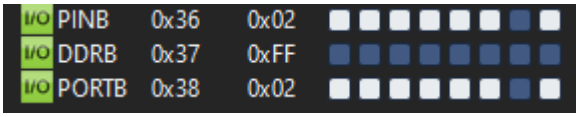


Figure 14. PORTB when the system is disarmed.

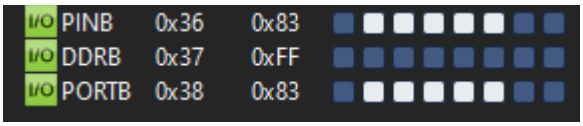


Figure 15. PORTB while the system is armed, and Zone A has been triggered.

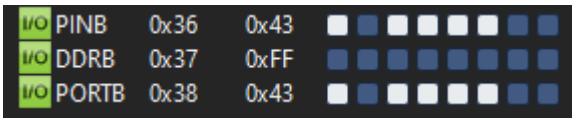


Figure 16. PORTB while the system is armed, and Zone B has been triggered.

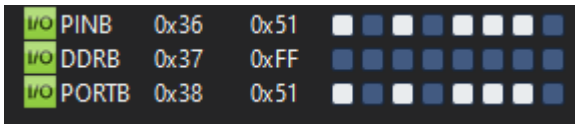


Figure 17. PORTB while the system is armed, and Zone C has been triggered.

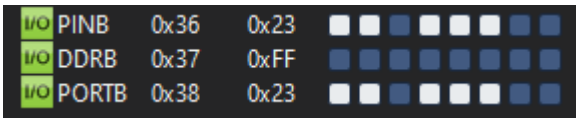


Figure 18. PORTB when user input (in this scenario, 5 on the keypad has been pressed) while the system is armed.

4.3.3 Results Comparison

Testing was carried out for the same conditions so that a clear comparison between the results maybe observed. As can be seen from the screenshots and captures of the OUSB board, the outcomes seen are the same. However, it must be noted that the OUSB board has PORTB in reverse, so the simulation results are mapped out in reverse on the board. It can be concluded that the assembler code and C code have the same structure and produce the same results, although they were developed seperately. It should be stated that the assembler code flash an LED at 2Hz when a zone has been triggered, acting as an strobe light, meeting the project's requirement criteria.

5 Discussion and Conclusion

During this project timeline, a three-zone alarm system using an OUSB-IO board and a membrane keyboard was developed. The requirements included triggering a specific zone through the *A*, *B* and *C* keys on the keypad, a single LED flashing at 2Hz (every 0.5s) to simulate a strobe light while another is lit to indicate a siren and a delay to occur that simulates the occupants leaving the area. All of the above requirements are achieved by this framework and the occupants are given 40 seconds after the siren and strobe light are activated to leave the triggered area.

Development of the assembly code for this task was found quite challenging. The linear execution of code in assembly and labelling bits of code, which is similar to user defined functions in C, were proven useful. Many comparison statements, break statements, "relative jumps" and the stack pointer were all utilized to obtain the desired outcomes.

Looking at the C code, advanced concepts such as interrupts, and internal microcontroller clocks were used for the development of this portion of microcode. During this phase, many re-usable user defined functions were written, that have been used continuously through out the main segment of the program. Interrupt service routines were used for delays. For this part of the project, the code was developed to be user-friendly, with more complex delays uses as “time-outs”. As an example, the user will have 5 seconds to enter the code, if the timer runs out, then system will simply go back to its former state and continue monitoring the three different zones. This function is necessary in any state-of-the-art alarm system as it increases the overall security.

However, there are a few discrepancies between the two versions of the code that must be discussed. Firstly, flashing a single LED at 2Hz was achieved by the assembler code, but since the C code uses interrupt service routines, such a function was proven difficult without complex features such as running two interrupts at the same time. The other disagreement between the two models is the absence of additional delays used as time-outs for better security since the assembler version does not use the internal clocks or interrupts. Instead, the assembler version was developed to run infinitely when the correct code is entered, meaning the user has any desired time to enter the new code, or to the enter the *correct* passcode. Nevertheless, the discrepancies in the assembler code are covered in the C code and the ones noticed in the C code is covered in assembler version.

One last point that must be brought to light is the difference between the two compiled code versions. When writing to the board, it was seen that 1454 bytes of data were written, whereas when writing the C code, 2522 bytes of data were written, although both achieve the same outcomes. The assembler code is approximately 1000 bytes smaller than that of C.

This project was a steep learning curve where we learned about the differences between approaches, and much collaborative research was done. While C is cheaper to write, it had a less granular structure; in comparison to this assembly is more expensive to write but was easier to tailor the code as per the design. Everything considered, the deliverables achieve the requirements stated, when both simulated and deployed on physical hardware.

6 Bibliography

[1] Microchip Technologies Inc, "AVR Instruction Set Manual," 2021.

[2] Microchip Technologies Inc, "megaAVR Data Sheet," 2018.

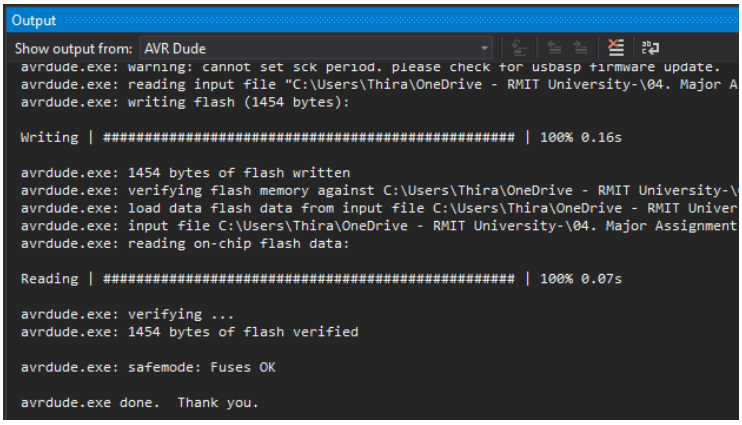
[3] G. Matthews, "Tutorial 4 (Weeks 8 and 9)," Melbourne, 2021.

[4] G. Matthews, "Lecture 9 - Introduction to Interrupts," Melbourne, 2021.

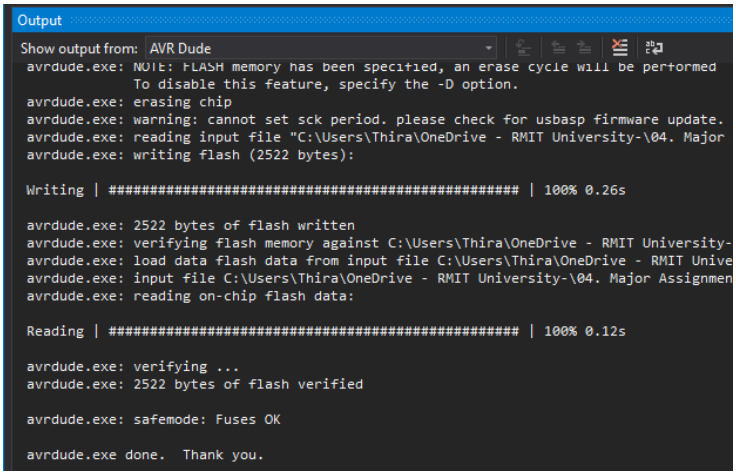
[5] B. Mulvey and T. Morland, "AVR Delay Loop Calculator," 2018. [Online]. Available: http://darcy.rsgc.on.ca/ACES/TEI4M/AVRdelay.html. [Accessed 12 10 2021].

7 Appendix

Appendix 3: 16-Key Matrix Keypad



Appendix 2: Screenshot from deploying the assembler code on a physical OUSB board. It can be seen that 1454 bytes of data was written.



Appendix 1: Screenshot from deploying the C code on a physical OUSB board. It can be seen that 2522 bytes of data was written.