Lebanese American University

School of Engineering



**Microprocessors Lab**

**COE 324 - 33**

**Final Project**

**Simon Game**

**Submitted to: Ms. Rayana Jaafar**

Prepared by:

Clara Akiki 201405063

Joseph Attieh 201501102

*December 4th, 2017*

Table of Contents

[Introduction 5](#_Toc500099295)

[Task1 6](#_Toc500099296)

[Task 2 13](#_Toc500099297)

[Task3 16](#_Toc500099298)

[Task 4 18](#_Toc500099299)

[Bonus 22](#_Toc500099300)

[Conclusion 23](#_Toc500099301)

[References 25](#_Toc500099302)

# Table of Figures

[Figure 1 - SIMON GAME scrolling to the right 7](#_Toc500099303)

[Figure 2 - Pins of the MCU 9](file:///C:\Users\user\Documents\mco-project.docx#_Toc500099304)

[Figure 3 - Port B bits connections 9](#_Toc500099305)

[Figure 4 - Code of the delay we used to make the scrolling to 3s 11](file:///C:\Users\user\Documents\mco-project.docx#_Toc500099306)

[Figure 5 - Code of Task 1 (1) 11](file:///C:\Users\user\Documents\mco-project.docx#_Toc500099307)

[Figure 6 - Code of Task 1 (2) 12](file:///C:\Users\user\Documents\mco-project.docx#_Toc500099308)

[Figure 7 - Modulus Down Counter Register (MCCTL) 13](#_Toc500099309)

[Figure 8 - Code of the customized delay 15](file:///C:\Users\user\Documents\mco-project.docx#_Toc500099310)

[Figure 9 - Code to get the sequence of random numbers 15](file:///C:\Users\user\Documents\mco-project.docx#_Toc500099311)

[Figure 10 - Code to get one random number 15](file:///C:\Users\user\Documents\mco-project.docx#_Toc500099312)

[Figure 11 - Code of Task 3 17](file:///C:\Users\user\Documents\mco-project.docx#_Toc500099313)

[Figure 12 - SCICR1 18](#_Toc500099314)

[Figure 13 - SCICR2 19](#_Toc500099315)

[Figure 14 - Code of Task 4 21](file:///C:\Users\user\Documents\mco-project.docx#_Toc500099316)

[Figure 15 - Pin connections of the MCU 22](file:///C:\Users\user\Documents\mco-project.docx#_Toc500099317)

[Figure 16 - Buzzer's info on the back of the board 22](file:///C:\Users\user\Documents\mco-project.docx#_Toc500099318)

[Figure 17 - Code of the subroutines that play the sound 23](file:///C:\Users\user\Documents\mco-project.docx#_Toc500099319)

[Figure 18 - Code of the delay used in the previous subroutines 23](file:///C:\Users\user\Documents\mco-project.docx#_Toc500099320)

# Table of tables

[Table 1 - MCPR1 and MCPR0 (prescaler) 14](#_Toc500099321)

[Table 2 - Functions of LOOPS and RSRC 18](file:///C:\Users\user\Documents\mco-project.docx#_Toc500099322)

# Introduction

In this project, we are going to implement a SIMON GAME! Simon is an electronic game of memory skill invented by Ralph H. Baer and Howard J. Morrison with software programming by Lenny Cope. The device creates a series of tones and lights and requires a user to repeat the series. If the user succeeds, the series becomes progressively longer and more complex. Once the user fails, the game is over. In this project, we will realize this game by doing several tasks covering topics previously done in the LABs.

# Task1

In this task, we are required first to show on the Screen “Simon Game” scrolling to the right on the first line of the display and “Welcome” on the second line of the display. Then, we will use the push buttons of our board to choose the level of difficulty.

First, we configure the SPI registers, DDRM and MODRR. Then, we initialize our display by sending certain instructions (covered in lab3). We chose to disable the cursor and the blinking by sending the instruction #$0C to the display. We should note that in the next code, we used the instruction #$80 responsible for getting the cursor to the beginning of the first line without clearing the screen.

Let’s proceed to print “Simon Game”. We proceed with a similar logic as the one used in lab4. To create the message and to be able to access it, we use the **MESSAGE1 DC.B 'Simon Game ’**. Now, MESSAGE1 is a pointer to the first character of the string that we want to show since each address in the memory is able to store 8 bits (thus 2 hexadecimal digits = one character). Then, we use **LENGTH1 DC.B 11** that will simply contain the length of our message. We need three variables: position, offset and counter. We use **POSITION1 DC.B 0**, **OFFSET1 DC.B 0** and **COUNTER1 DC.B 11** to create these variables and we make sure at the beginning of our code to resend the content of these variables to their memory locations so that each time we enter this part of our code, the values are reinitialized (this will be beneficial when we will be jumping to the beginning of our code). The offset variable is used to keep track of the number of characters that we take from the memory. The position keeps track of how many characters we have printed on the screen. The counter variable keeps track of the index of the first character that we should print so we would be able to simulate the scrolling.

We start our code by putting the cursor on the beginning of the first line by using **LDAA #$80** then by sending that instruction to the LCD (**JSR SENDINSTR**). Since our addresses are 16 bits (4 hexadecimals), we can only save them in registers X or Y. We load the pointer using **LDX #MESSAGE1** that loads the address of our message in register X. Then, we load the offset in register B (**LDAB OFFSET**) so we could be able to use **LDAA B, X** to load in A the character from the memory (the address of the character loaded in A is equal to the address in X with an offset located in B). Then, we send this character to the LCD (JSR **SENDDATA**) so it could be printed. To move to the next character, we increment the offset by using I**NCB**. We compare the offset with the string length (**CMPB #LENGTH1**); if it is equal to the length, we clear the offset (**CLRB**) so we could be able to start from the first character in our message. We make sure to store the new offset in the memory OFFSET1 after each time we compare (**STAB OFFSET1**). After that, we increment the content of POSITION1 (**INC POSITION1**) then we load it in register B (**LDAB POSITION1**).Then, we compare the content of register B (the position value) with 8 (**CMPB #8**) so we can keep track of the number of characters already printed (the LCD can show a maximum of 8 characters). If it is not equal to 8 (if less than 8 characters are printed), we repeat the process of printing till it prints 8 characters (**BNE PrintSG**). When 8 characters are printed, we use **LDAA #$80** then **JSR** **SENDINSTR** to return to the beginning of the first line and we clear the position (**CLR POSITION1**). After that, we decrement the counter (each time we print 8 characters, we decrement it so we could know from which offset we will start printing later so we can simulate the scrolling effect) by using **DEC COUNTER1** then **LDAB COUNTER1**. Then, we proceed to printing “Welcome” on the middle of the screen by sending each character to the screen (first, we use **LDAA #$A8** then **JSR SENDINSTR** to show characters on the second line on the screen then for example to send a ‘W’ to the screen we use **LDAA #‘W’** then **JSR SENDDATA**). After that, we use **LDAA #$80** and **JSR SENDINSTR** to go back to the beginning of the first line without erasing the content of the screen.

Then, we store the value of COUNTER1 in OFFSET1 since we want the message to scroll to the right (**STAB OFFSET1**). We compare the value of this offset with 5 (**CMPB #5**). If it is equal to 5, that means that the last 8 characters printed are “Game Sim” which means that we scrolled our message one time. We can exit the loop. Otherwise, we keep on printing till the offset/counter is 5. This is best illustrated below, where the value of our COUNTER1 and OFFSET1 are showed after each time we print the characters on the display. Also, the position is clearly showed.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| position1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  | counter1/offset1 |
|  | S | I | M | O | N |  | G | A |  | 10 |
|  |  | S | I | M | O | N |  | G |  | 9 |
|  | E |  | S | I | M | O | N |  |  | 8 |
|  | M | E |  | S | I | M | O | N |  | 7 |
|  | A | M | E |  | S | I | M | O |  | 6 |
|  | G | A | M | E |  | S | I | M |  | 5 |

Figure 1 - SIMON GAME scrolling to the right

We want all this scrolling to spend only 3 seconds on the screen. We incorporated this concept using the DELAY subroutine (no interrupts). We want to uniformly add a delay to each character so that the total message printed lasts 3s. So, we change the delay used in the SPI, SENDDATA and SENDINSTR. Instead of counting the number of times we entered DELAY, we added in the subroutine a variable that we incremented each time we enter this subroutine (after DBNE X, \* in the delay we added INC NBR where NBR is a memory location that we allocated). We found that our delay subroutine was entered 249 times (NBR was holding 249). Since the MCU works at 2𝑀𝐻𝑧, the MCU wastes 0.5 𝜇𝑠 in 1 cycle. Thus, the number of cycles required to waste 3s is cycles. Since in our delay we are using DBNE and since DBNE uses 3 cycles, we are required to execute all the DBNEs in times. Thus, and . Thus, we put 8032 in the register that we are using in our delay subroutine to achieve a “Simon Game ” scrolling once for a duration of 3s.

After clearing the display using the instruction #$01 (**LDAA #$01** and **JSR SENDDATA**), we proceed into displaying “Pick the level of difficulty” scrolling to the left. This is done in the same way we did in Lab4.

We start our code by putting the cursor on the beginning of the first line by using **LDAA #$80** then by sending that instruction to the LCD (**JSR SENDINSTR**). We load the register X with the address from where the first character of our message is present (**LDX #MESSAGE2**). Then, we load the offset in register B (**LDAB OFFSET2)** so we could be able to use **LDAA B, X** to load in A the character from the memory (the address of the character loaded in A is equal to the address in X with an offset located in B). Then, we send this character to the LCD (**JSR SENDDATA)** so it could be printed. To move to the next character, we increment the offset by using **INCB.** We compare the offset with the string length (**CMPB #LENGTH2)**; if it is equal to the length, we clear the offset (**CLRB**) so we could be able to start from the first character in our message. We make sure to store the new offset in the memory OFFSET2 after each time we compare (**STAB OFFSET2**). After that, we increment POSITION2 (**INC POSITION2**) then we load it in register B (**LDAB POSITION2**). Then, we compare the content of register B (the position value) with 8 (**CMPB #8**) so we can keep track of the number of characters already printed (the LCD can show a maximum of 8 characters). If it is not equal (if less than 8 characters are printed), we repeat the process of printing till it prints 8 characters (**BNE PrintPL**). When 8 characters are printed, we reset the position (**CLR POSITION2**). After that, we increment the counter (each time we print 8 characters, we increment it so we could know from which offset we will start printing later so we can simulate the scrolling effect) by using **INC COUNTER2** then **LDAB COUNTER2.** Then, we compare the counter with the length of the message (**CMPB #LENGTH2**). If it is equal (that means that all the message was printed once), we reset the counter to 0 (**CLR**  **COUNTER2**). Else, we place the counter in the offset (**MOVB COUNTER2, OFFSET2**) so we could continue the printing. Then, we branch to the beginning.

We are asked to use the push buttons PB1, PB2 and PB3 of the board to select a level of difficulty. In other words, if any push button is pressed, we should exit the loop that prints “Pick the level of difficulty ” and choose if the level is easy, medium or hard. We chose to connect those push buttons to PORTB. Since they are considered as inputs. Since we are going to use also LEDS later as outputs, we should configure PORTB pins as outputs or inputs. We use DDRB to configure this by using **MOVB #$F0, DDRB** in our code. We will have to connect the push buttons to three ports of PB3, PB2, PB1 and PB0.

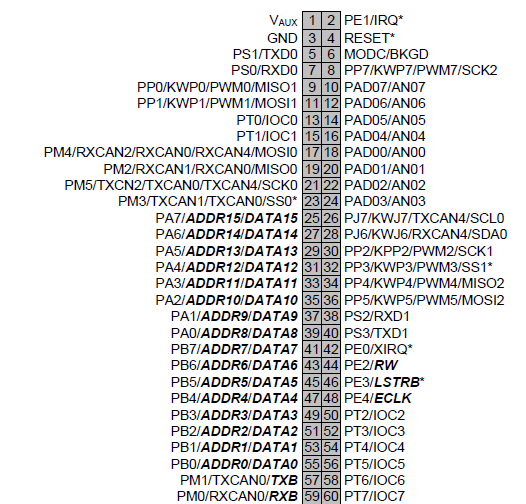


Figure - Pins of the MCU

So, we disconnect the jumpers connecting PORTB to the switches and we physically connect the push buttons to PORTB (either by connecting wires where the jumper was or by connecting it like in the picture below). We connected the push button PB1 to the pin PB0 (pin 55), the push button PB2 to the pin PB1 (pin 53) and the push button PB3 to the pin PB2 (pin 51).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| PB7 | PB6 | PB5 | PB4 | PB3 | PB2 | PB1 | PB0 |
| LEDs | | | |  | PushB3 | PushB2 | PushB1 |

Figure 3 - Port B bits connections

So, to exit the “Pick level of difficulty ”, we use **BITA** to test the bit representing each push Button. BITA does an AND operation of the contents of register A with bits (in other words check if the bit given is 0 or not). So, we load the bits of PORTB in register A (**LDAA PORTB**). Then we check each bit and we branch accordingly. If the push button PB1 is pressed then bit 0 of PORTB is 0 and **BITA #1** gives a 0 and we branch to EASY where we will handle what will happen (**BEQ EASY**). The same happens with push buttons PB2 and PB3 where we branch to MEDIUM and HARDER. If no push button is pressed, we keep on scrolling “Pick level of difficulty ” and we do not exit this loop.

Once the user selects a level, we are required to initialize the sequence length to either 5, 10 and 15 for the levels Easy, Medium and Hard respectively. Then, we should display the selected level on the first line of the LCD while a countdown from 3 to 1 is displayed on the second line. First, we use **LEVEL** **DC.B 0** so it could hold the value of the level/length of sequence we are going to use. If the user presses a button, it initializes (stores) the corresponding length of sequence in this variable then we print the level on the screen. For example, if the user has pressed PB1, the program branches to EASY where we initialize LEVEL to 5 by using **LDAB #5** and **STAB LEVEL**. Then, we print EASY in the middle of the display by loading each character and sending it to the screen. We do the same with MEDIUM and HARD but we initialize LEVEL to different values and we send the corresponding characters of each one. After we end up printing, we jump to COUNTDOWN where the countdown from 3 to 0 will happen on the second line.

To count down on the second line of the display, we first put the cursor on the second line of the screen by using **LDAA #$A8** and **JSR SENDINSTR**. We send blank characters to the display so that the countdown appears at the middle. Then, we use **CDOWN DC.B 3** to store the number 3 that we will use to track the countdown. First, we load the contents of CDOWN in register A (**LDAA CDOWN**) and we add #48 to this number so we can get its ascii equivalent(**ADDA #48**). Then, we send it to the screen and we decrement this CDOWN (**JSR SENDDATA** then **DEC CDOWN**). We keep on repeating the process (**BNE RepeatCNT**) until CDOWN is 0. At this stage, we would have counted down from 3 to 0. At the end of this method, we initialize CDOWN to 3 in case we wanted to reuse this part of our code so that it could work.

The full commented code of this task is found in the next page (the full code is found on the CD).

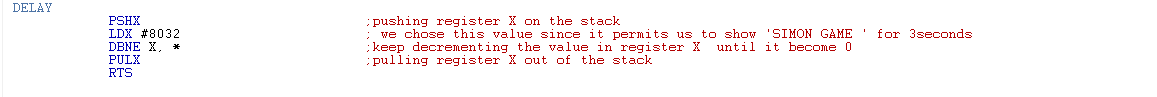


Figure - Code of the delay we used to make the scrolling to 3s

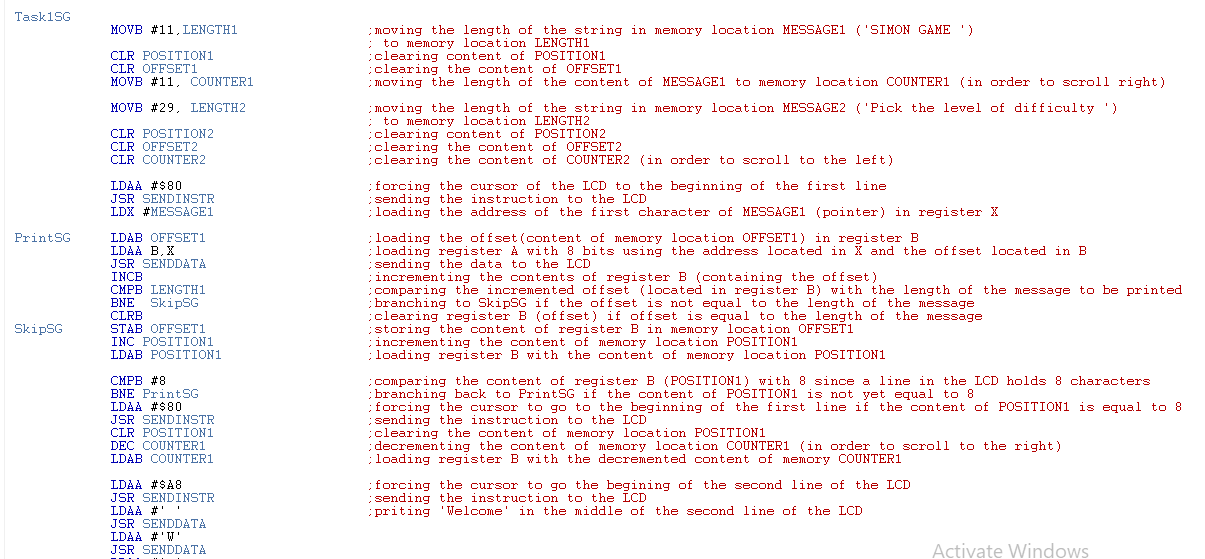
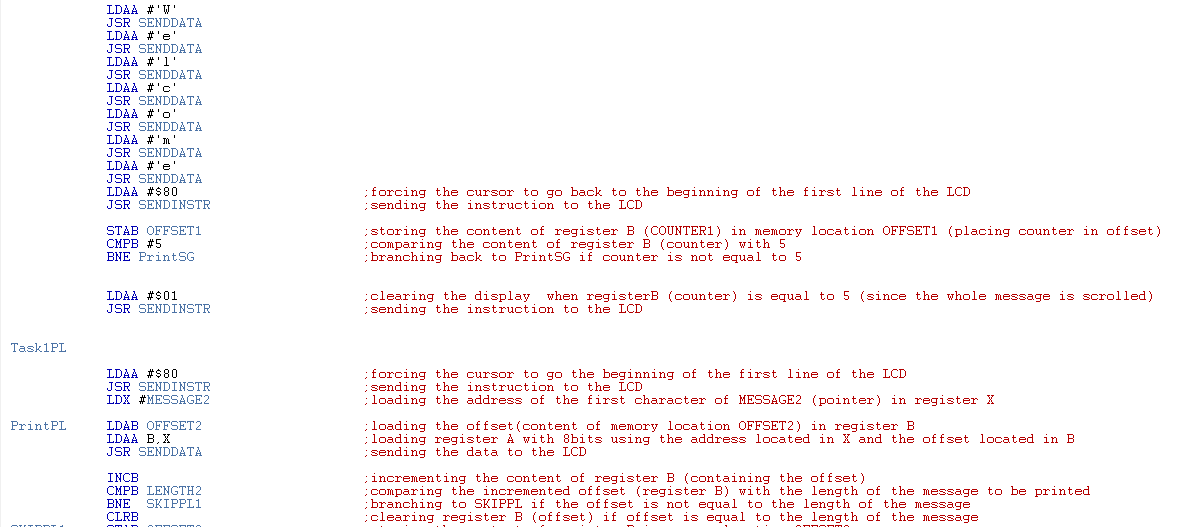
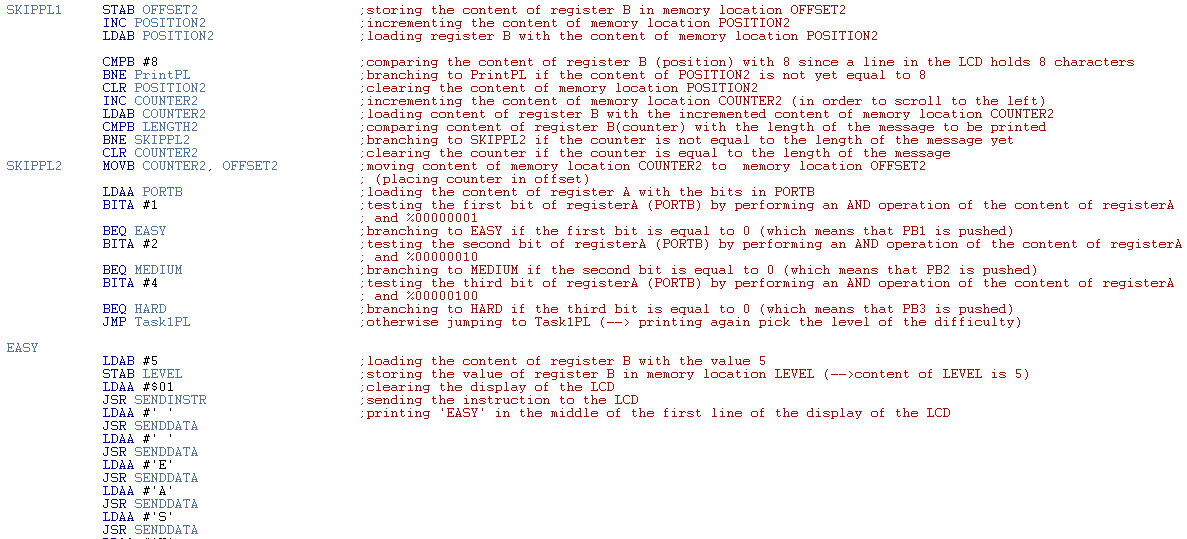


Figure - Code of Task 1 (1)



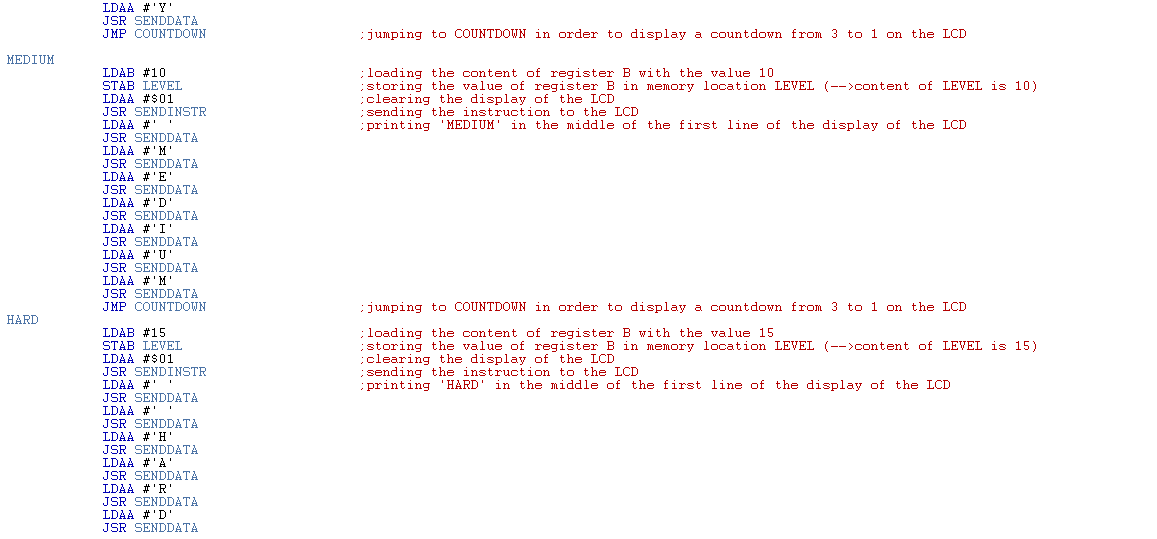
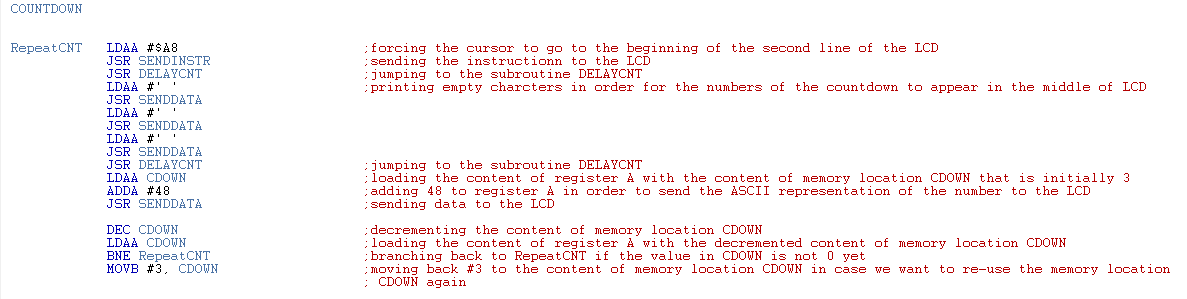


Figure - Code of Task 1 (2)

# Task 2

In this task, we are required to use a subroutine that generates random numbers between 1 and 4 inclusive and that put them in memory locations.

We will be creating a subroutine that generates all the sequence that we want. Since generating all the numbers once results in some patterns, we created a subroutine that generates one number at a time. We made sure to save all the registers used on the stack (**PSHX PSHA** ) and to pull them before RTS (**PULA PULX**). Our logic consists of loading the MCCNT with a big value and pulling it at a certain time of our code (**LDD MCCNT**). We take this value and we compute the remainder of its division by 4 (**LDX #4**). The IDIV return the remainder in register D, thus register B (remainder small). We get numbers that are either 0, 1, 2 or 3. Finally, we add to the remainder 1 so we can get a value between 1 and 4(**ADDB #1**) to get the random number generated.

However, doing so will result in a pattern in the numbers since the randomness of our code depends on the time that we pull the value from the MCCNT. This time depends of the time the user presses the push buttons since the MCCNT keeps on decrementing and reloading. So, we created a Customized Delay Subroutine. This subroutine is a delay that varies each time it is called. This delay is the same delay we used before, but instead using a fixed number to generate a delay and waste cycles, we used a variable random value. In fact, this delay uses the same concept that we used to generate the random number; we pull a random value from the MCCNT and we use DBNE to create a customized delay. Invoking this subroutine before pulling from the MCCNT and before getting the random number will make the data even more random.

We configure the MCCTL (Modulus Down Counter Control Register)

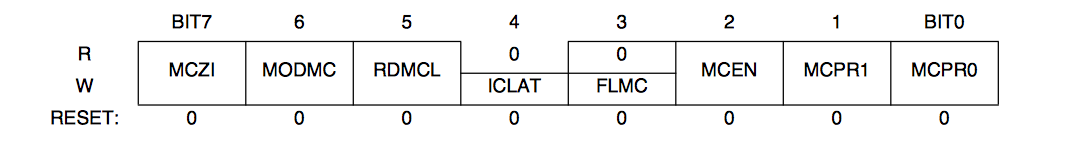


Figure 7 - Modulus Down Counter Register (MCCTL)

- Bit 7 – MCZI: Modulus Counter Underflow Interrupt Enable. Since we do not need interrupts in this task, we set this bit to **0**.

-Bit 6 – MODMC: Modulus Mode Enable. We want the modulus mode enabled so when the counter reaches $0000, the counter is loaded with the latest value written to the modulus count register. According to the datasheet, we set this bit to **1**.

-Bit 5 – RDMCL: Read Modulus Down-Counter Load. If we want to read the content of the modulus count register, we want it to return the present value of the count register instead of the contents of the load register. So, we set this bit to **0**.

-Bits 4 – 3: As we see in Figure 5, bit 4 and bit 3 are **0** by default (read of those bits will always return zero).

-Bit 2: MCEN: Modulus Down-Counter Enable. We want the modulus counter enabled, so we set this bit to **1**.

- Bits 1-0: MCPR1, MCPR0: Modulus Counter Prescaler select. These two bits define the division rate of the modulus counter prescaler. The prescaler is set according to the table below:

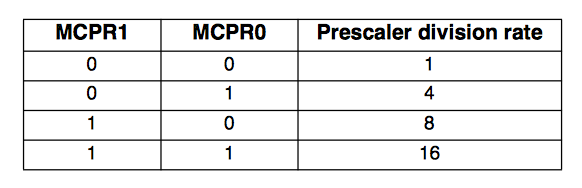


Table 1 - MCPR1 and MCPR0 (prescaler)

Since we want a count very high, that means that we want a small frequency. Thus, we choose the biggest prescaler that is 16. Therefore, we set bit 1 and bit 0 to **1**.

So, to use this register in this lab, we should move **#%01000111 = #$47** toMCCTL**.** Since we want a big number in the MCCNT and since it is a 16 bit register, we use MOVW #$FFFF, MCCNT.

Finally, we want to store the random numbers generated in the memory (**SEQUENCE DC.B 0**) according to the level of our game (length of our sequence). We jump to the subroutine RANDOM that generates a random number and puts it in register B. We load the address where we want to store the random numbers in register X (**LDX #SEQUENCE**). Then, we load the offset in register A (**LDAA VOFFSET)** so we could be able to use **STAB A, X** to store the content of B to the address that is equal to the address in X with an offset located in A. Then we increment the offset (**INC VOFFSET**) and we keep on repeating the procedure until the offset is equal to the LEVEL (**LDAA VOFFSET CMPA LEVEL BNE REPEAT**). After finishing, we clear the offset to make sure that if we want to use this method again, no problem will occur.

The commented code of this task can be found on the next page.

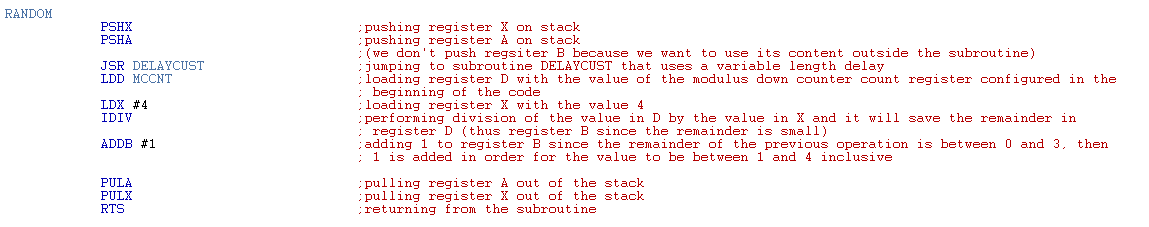
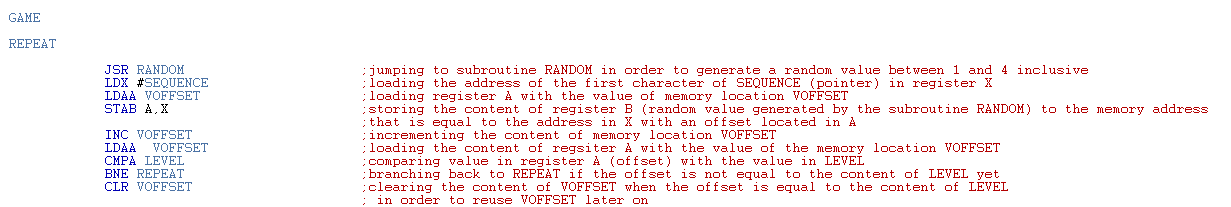
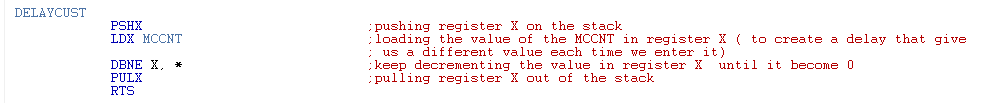


Figure - Code of the customized delay

Figure - Code to get the sequence of random numbers

Figure - Code to get one random number

# Task3

Following the generation of our random numbers, we are required to create a subroutine that will light the corresponding LEDs for a short period. Basically, the random number 1 should light LED1 for a certain period, random number 2 should light LED 2 for a certain period … This subroutine will only take as input the combination generated by the random number generator subroutine.

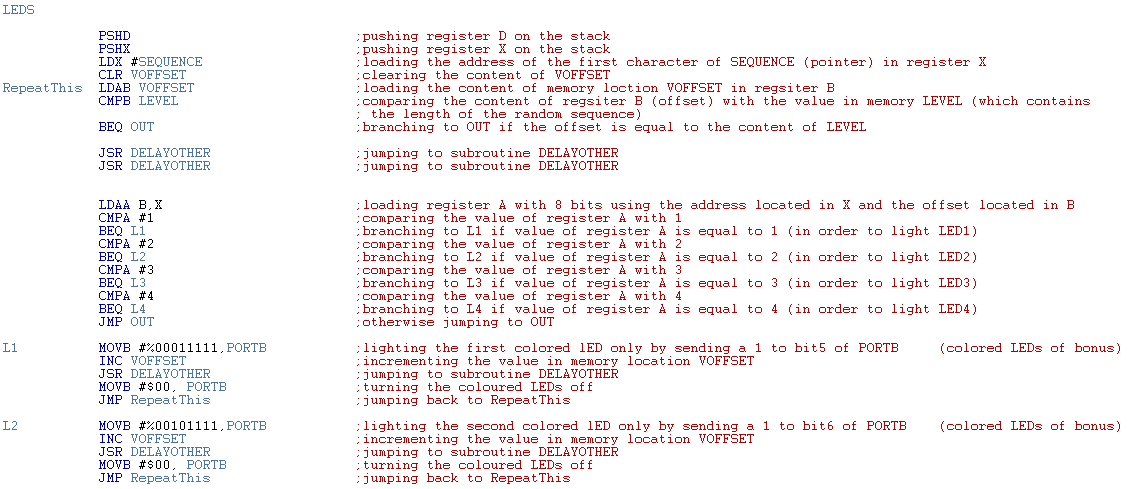
We create the subroutine called LEDs. We make sure to save all the registers on the stack in the beginning of our subroutine (**PSHD PSHX**) to be able to restore them at the end of our code (**PULX PULD**).

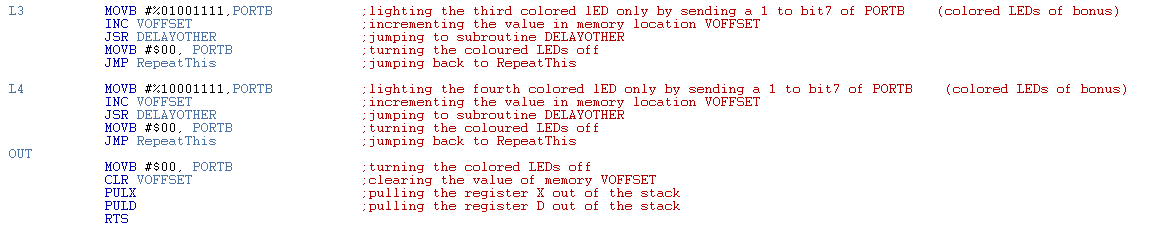
We make sure just after we already set up our PORTB using DDRB to turn the LEDs off. The LEDs on the board needed 1 in their corresponding bit in PORTB so they turn off and a 0 so that they turn on. Since we chose to make the bonus, and since when we connected the LEDs they lighted up when we sent a 1, so we set the bit corresponding to the LED we want to light up to 1 instead of a 0 (the connections are explained later in the bonus).

Then, we need to get the random numbers that we already stored in the memory so that we light the corresponding LED. We load each number of this sequence using LDX #SEQUENCE that loads the address of our message in register X. Then, we load the offset in register B (**LDAB OFFSET**) so we could be able to use **LDAA B, X** to load in A the number in our sequence. We stop getting the numbers from the memory when the number of numbers we got is equal to the LEVEL (**CMPB LEVEL** and **BEQ OUT**). If the number we got is 1, we light up the corresponding LED in L1 (**CMPA #1** and **BEQ L1**). If the number is 2, we light up the corresponding LED in L2 (**CMPA #2** and **BEQ L2**). And so on till L4. In L1, we light up LED1 by setting bit 4 of PORTB to 1. Since we want LED1 to light up only while all the LEDs are off, we use **MOVB #%00011111, PORTB**. Then we increment the offset we are using to go through the numbers we stored in the memory (**INC VOFFSET**). Then we turn off all the LEDs so that we can later turn on all the LEDs and so the user can clearly see what is the LED on. We make sure to insert delays between each time we change the state of the LED and between LEDs so that the user can observe clearly the lighted LED (for that purpose we created a new delay that is bigger than the older delay called **DELAYOTHER**). We use the same logic in L2, L3 and L4 to light up LED2, LED3 and LED4 by setting bit 5, bit 6 and bit 7 respectively to 1 to light the LED and by clearing this bit before we leave this part of the code. Then, we jump to the part of the subroutine that got us the number from the memory and we redo the same logic until the VOFFSET is equal to LEVEL. That means that the numbers in our sequence are equal to the number in the sequence and we branch to OUT (**BEQ OUT**) where we light off all the LEDs and we clear the VOFFSET (so we can use VOFFSET again) and we exit this subroutine.

We call this subroutine just after we stored all the numbers in memory (**JSR LEDS**).

The commented code of this task can be found on the next page.





4

5

6

Figure - Code of Task 3

# Task 4

In this task, we are required to perform serial communication between the MCU and our PC/Laptop using SCI and Putty to get the numbers from the user. The combination sent by the user is made up of numbers between 1 and 4 where each number corresponds to the LEDs 1 to 4 respectively. We chose to do this task without interrupts.

We should configure the registers used in this task.

A- SCI Control Register 1 (SCICR1)

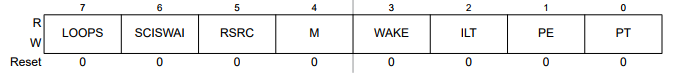
The register above has the following bits:

Figure 12 - SCICR1

- Bit 7 – LOOPS: Since we want normal operation enabled, we set this bit to **0**.

-Bit 6 – SCISWAI: Since we want the SCI enabled in wait mode, we set this bit to **0**.

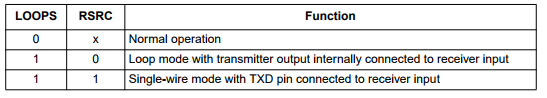
- Bit 5 – RSRC: Receiver Source Bit

Table - Functions of LOOPS and RSRC

Since the LOOPS is set to 0, it does not matter what is the value of this bit because we are using normal operation. We set this bit to **0**.

- Bit 4 – M: Since we want one start bit, eight data bits, one stop bit, we set this bit to **0**

- Bit 3 – WAKE: Since we want the Idle line wakeup, we set this bit to **0**.

- Bit 2 – ILT: We want the Idle character bit count to begin after the start bit, so we set this bit to **0**.

- Bit 1- PE: We want the Parity function disabled, so we set this bit to **0**.

- Bit 0 – PT: The Parity function is disabled, so it does not matter what we set this bit. We set it to **0**.

So, to configure this register, we use **MOVB** **#$00, SCI0CR1**

B- SCI Control Register 2 (SCICR2)

The register above has the following bits:

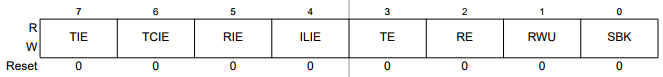


Figure 13 - SCICR2

This register will enable us to send and receive.

- Bit 7, 6 and 5: Since do not want to use interrupts in this task, we set those bits to **0**.

- Bit 4 –ILIE: Since we won’t use the idle mode, we set this bit to **0**.

- Bit 3 –TE: Since we want the transmitter enabled, we set this bit to **1**.

- Bit 2 –RE: Since we want the receiver enabled, we set this bit to **1**.

- Bit 1 –RWU: Since we need to work in the normal operation, we set this bit to **0**.

- Bit 0 –SBK: Since we won’t need break bits, we set this bit to **0**.

So, to configure this register, we use **MOVB** **#%00001100, SCI0CR2** or **MOVB #$0C, SCI0CR2.**

C- SCI Baud Rate Registers (SCIBDH, SCIBDL)

The SCI baud rate register is used to determine the baud rate of the SCI. To get how much we should store in those registers, we proceed like we did in Lab7 and we use the formula used there to get

So, to configure those registers, we use **MOVB #$0D, SCIBDL** and **MOVB #$00, SCIBDH**.

Now that we configured the registers, we proceed with our code. First, we use **BCLR SCI0SR1, $20, PUTTY** to keep checking bit 5 (Receive Data Register Full Flag) of the SCI0SR1. While this bit is 0, we keep branching to this statement so we can recheck it. While this bit is 0, data is not available in SCI data register. When this bit become 1, the received data available in SCI data register which means that the user started entering the sequence on putty. To compare the character entered by the user with the corresponding character in our sequence, we use **LDX #SEQUENCE**, **LDAB VOFFSET** and **LDAA B,X** to load this character in register A. Then, we add the 48 to the number that we got from the memory (since when the user enters 1 it is represented by ASCII). Then we compare the character we got with the contents of SCI0DRL that holds one number of the sequence entered by the user (**CMPA SCI0DRL**). If they are not equal, that means that the user entered a wrong number and we branch to FALSE. If it is equal, we clear the status register (by using **LDAB SCI0SR1** and **LDAB SCI0DRL**) so that the SCI0DRL is loaded with the next number in the sequence entered by the user. We keep on doing this until the VOFFSET is equal to the LEVEL (the number of numbers we compared is equal to the length of our sequence). If it is equal, this means that the user entered all the sequence correctly and we branch to TRUE.

Now, let’s write the codes where we are going to handle when the user wins or loses. When the user win, we branch to TRUE. First, we jump to the subroutine that generates a sound (**JSR BUZZWIN** ). This subroutine will be explained in the bonus question. Then, we use the instruction #$01 to clear the screen and we send each character in “Correct” to the screen (**LDAA # ‘C’** **JSR SENDDATA**, **LDAA #‘o’** **JSR SENDDATA** and so on ). Then, we jump to the DELAYOTHER function. Then, we increment the LEVEL to increase the sequence of numbers to be showed. Then, we clear the screen to print “BE READY” so that the user gets ready to see the new sequence of LEDs. Then, we clear the status register (**LDAB SCI0SR1** then **LDAB SCI0DRL**) and the VOFFSET to make sure that we can replay the game and that we are able to enter the PUTTY subroutine. Then, we jump to GAME where we generate random numbers, light the LEDS accordingly and wait for the user’s input.

When the user loses, we branch to FALSE. First, we jump to the subroutine that generates a sound from the buzzer (**JSR BUZZLOOSE**). This subroutine will be explained in the bonus question. Then, we use the instruction #$01 to clear the screen and we send each character in “You Lost” to the screen (**LDAA # ‘Y’ JSR SENDDATA**, **LDAA #‘o’ JSR SENDDATA** and so on ). Then, we jump to the DELAYOTHER function. Then, we clear the status register (**LDAB SCI0SR1** then **LDAB SCI0DRL**) to make sure that we can replay the game and that we are able to enter the PUTTY subroutine. Then, we jump to the beginning of our code where we made sure that we reinitialized all of our variables.

The commented code for this task can be found in the next page.

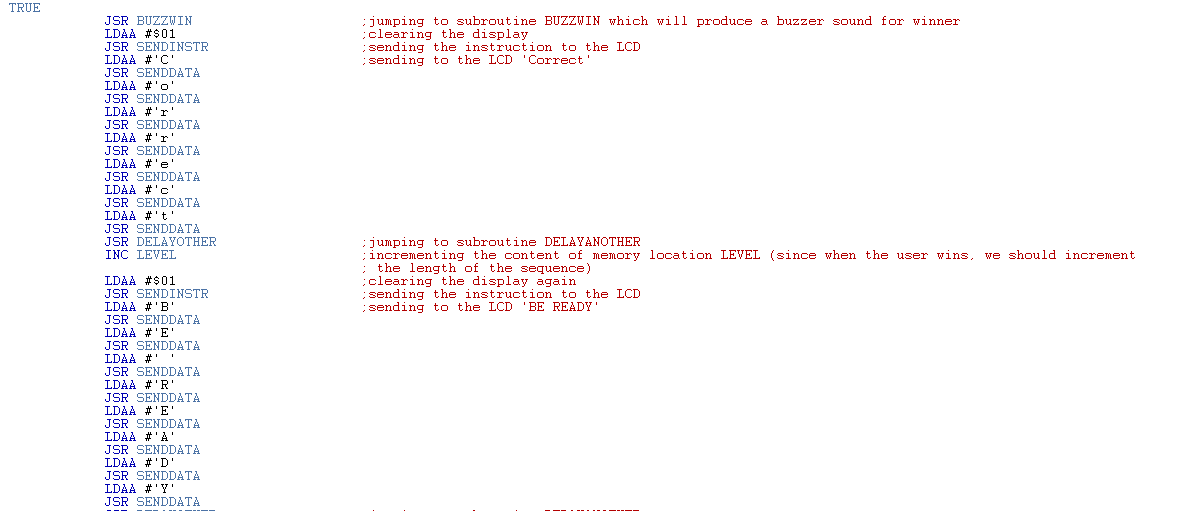
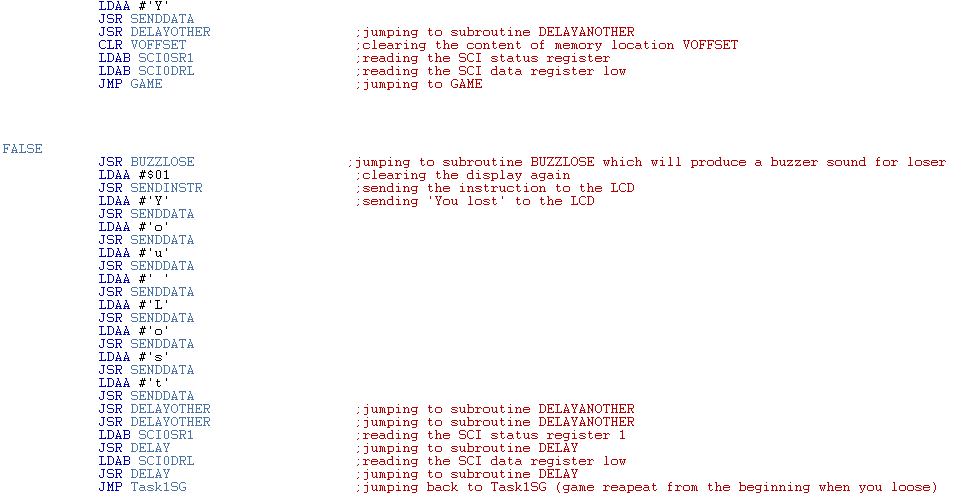
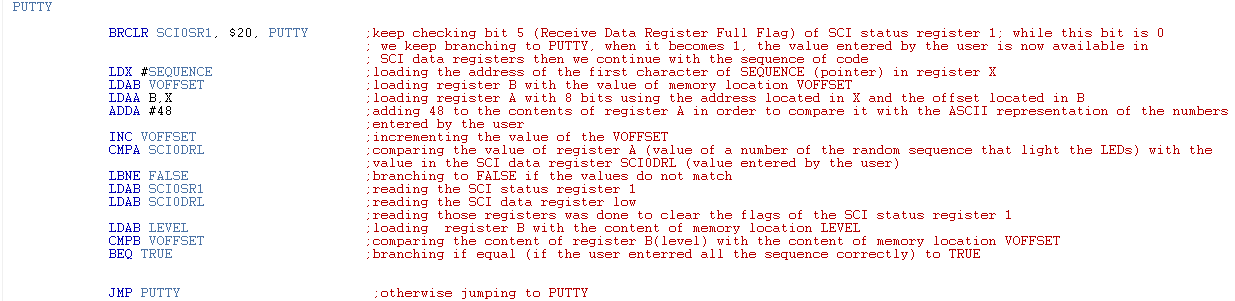


Figure - Code of Task 4

# Bonus

In this task we should use two different buzzer sounds for when the user loses or answers correctly. Also, we should use colored LED lights.

First, we should connect the buzzer. When we look at the back of the MCU, we see that the buzzer is connected internally to the MCU port 13. Port 13 corresponds to PT0 which means that it corresponds to bit 0 of PORT T. Since the buzzer give us a sound, we set it as an output by MOVB #%00000001, DDRT just like we do with PORTB (we do not care about PT7, PT6 ,… PT1).

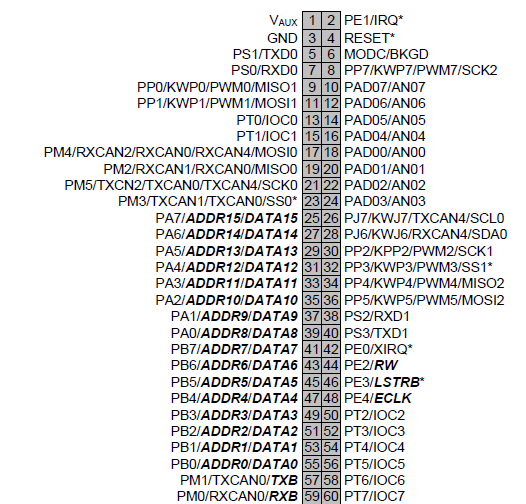


Figure - Pin connections of the MCU



Figure - Buzzer's info on the back of the board

The buzzer works in a way that the sound that generates depends on the frequency. The frequency of the working buzzer(number of time the buzzer toggles per second) depends on the delay when the buzzer switches from 1 to 0. Thus, in both BUZZWIN and BUZZLOOSE subroutine we used **MOVB #$01,PTT** to set PT0 to 1, we used a delay of our own and we used **MOVB #$00,PTT** followed by delay again. This was done so that the buzzer keeps switching between the two states. The delay number was chosen in a way that the sound of the buzzer that was generated was the one that we wanted to use. We should note that we called the DELAYW two times in BUZZWIN (doubled the delay) so that we get different sounds for the buzzer. Also, we used a variable BUZZ so that we hear the sound of the buzzer for a certain duration ( we keep on decrementing this variable using **DEC BUZZ** until it is 0 where we exit the subroutine and then we restore BUZZ to its original value so this subroutine can be reused later).

Now that we have talked about the buzzer, we want to change the LEDs used. First, we can disconnect the jumpers responsible for the LEDs on the board so that they stop working. Then, we connect our four LEDs alongside resistors to the board. LED1 is connected to PB4 (pin 47), LED2 to PB5 (pin 45), LED3 to PB6 (pin 43) and LED4 to PB7 (pin 41). Each resistor is connected in series with its LED. The other end of the resistor should be connected to the ground (pin 3). All the pins are clearly showed in the picture above. Throughout our code, we made sure to set the bit corresponding to each LED to 1 when we want to light up the LED and 0 when we want to turn it off.

The subroutines to make the sound of the buzzer are shown below:

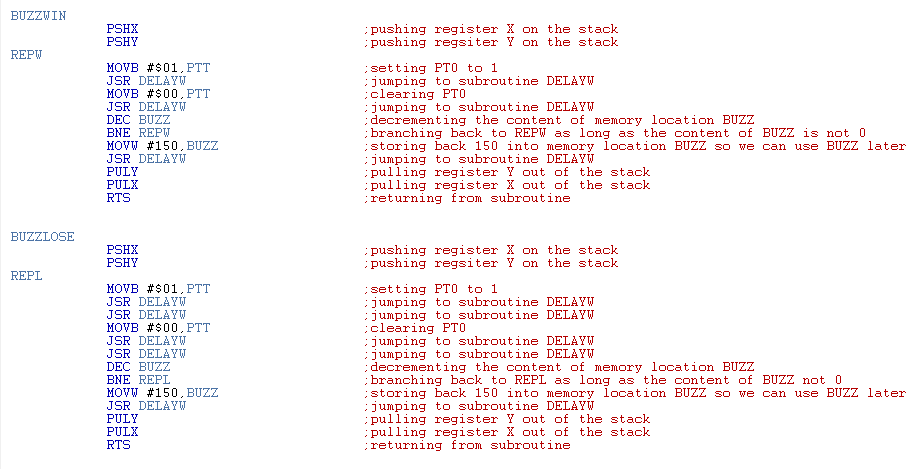


Figure - Code of the subroutines that play the sound

# Conclusion

Figure - Code of the delay used in the previous subroutines

In this project, we learned how to develop a Simon Game from scratch. First, we printed two messages on the screen: a “Simon Game ” scrolling to the right for 3 seconds and a “Pick the level of difficulty ” scrolling to the left. Then, we connected three push buttons to the MCU so that the user can pick one level of difficulty using those push buttons. Then, we used a random generator subroutine to generate a random sequence. After that, we took that sequence and we lighted up the LEDs according this sequence. Finally, we configure putty and we enabled the user to enter the sequence and he either win or lose.

# References

* <https://www.ece.cmu.edu/~ece348/labs/docs/APS12C128_module/SLKS12UG_module_user_guide.pdf>
* <http://services.sea.lau.edu.lb/academia/courses/coe324/handouts/Datasheets/Interrupt%20Registers%20Reference.pdf>