COE324 - Section 33 Microprocessors Lab – Final Project	**GRADE**

Table of Contents

Table of Contents 2

Table of Figures 3

Table of Tables 4

Introduction 5

Part I: Introducing the Buzzer 6

- ❖ What is a Buzzer 6
- ❖ Locating the Buzzer 7
- ❖ Configuring the Registers 9
- ❖ Interrupts for Buzzer: 10
- ❖ Frequencies and Durations for each Musical Note 14

Part II: Methodology: 17

Part III: Code 22

- ❖ Playing the Songs 22
  - ➢ Song 1: 22
  - ➢ Song 2: 27

Conclusion 29

References: 31

Table of Figures

Figure 1: Buzzer Picture 6
Figure 2: Buzzer Connection on the Schematic (1/3) 7
Figure 3: Buzzer Connection in the Schematic (2/3) 7
Figure 4: Buzzer Connection in the Schematic (3/3) 8
Figure 5: PTT and DDRT in the block diagram 8
Figure 6: DDRT Register 9
Figure 7: DDRT Register Description 9
Figure 8: PTT Register 10
Figure 9: PTT Register Description 10
Figure 10: Initializing the Buzzer 10
Figure 11: Interrupt vector for MCCTL 11
Figure 12: Interrupt Vector code for the buzzer 11
Figure 13: 16-Bit Modulus Down-Counter FLAG Register (MCFLG) 12
Figure 14: IntrBuzz Code 12
Figure 15: Modulus Counter Prescaler Select 13
Figure 16: Local Oscillator on Schematic 13
Figure 17: Notes Frequency according to Octaves 14
Figure 18: Whole note 17
Figure 19: Half note 17
Figure 20: Quarter note 17
Figure 21: Musical Notes and their Beat Values 18
Figure 22: Song 1 Notes 18
Figure 23: Song 2 Notes 19
Figure 24: Notes of the first song (Au clair de la lune) 23
Figure 25: Code of Playing the Do note of the first song 24
Figure 26: Code of Playing the Mi note of the first song 25
Figure 27: code sample of the full note Do 26
Figure 28: Notes of the second song (I love you, you love me) 27
Figure 29: Corresponding French note of the English notes 27
Figure 30: Code of Playing the Fa note of the second song 28
Figure 31: Flowchart 29

Table of Tables

Table 1: ABC and Solfege System Equivalence........ 15
Table 2: Frequency of each note........ 16
Table 3: Value into MCCNT for each note........ 16

Introduction

In this project, we will first introduce the buzzer, which will interact with the MCU to play the desired sound. We'll use the Buzzer to play two different songs once we've gotten to know it. Each Rhythm can be selected using a distinct push button. If the other push button is pressed, the second song starts playing. While the songs are playing on the buzzer, we will use the LEDs to blink to the rhythm in a specific pattern and the LCD to display the lyrics.

Part I: Introducing the Buzzer

❖ What is a Buzzer

A buzzer is an electric component known as an audio signaling device that can be one of the different types:

- Electromechanical: design using bare metal disc and an electromagnet. It generates sound by the movement of disc and magnetism concepts.
- Piezoelectric: utilizes the piezoelectric ceramic’s piezoelectric effect and pulse current in order to vibrate the metal plate and generate noise. Typically made with: resonance box, multi resonator, impedance matching…
- Mechanical: uses solenoid coil, oscillator, housing, and magnet to vibrate. The vibrating component is put in the outside.

A buzzer is mainly used to include features of sound in the system some of components where the buzzer is used are timers, alarm devices, and printers. It can be integrated into the breadboard or PCB. The buzzer’s mission is to transform the audio signal into sound. In order for the buzzer to work, we need to supply it with a DC power system. The buzzer is generally connected to switch in order to turn ON/OFF the buzzer at the necessary time interval.

A buzzer usually has 2 pins which are positive and negative as shown below in the pin configuration:

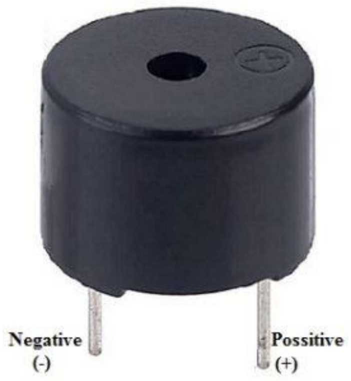


Figure 1: Buzzer Picture

❖ Locating the Buzzer

As for the buzzer on the board, we located it on the breadboard, and using that we were able to locate it on the schematic in the PBMCUSLK_SCH_D_0 datasheet.

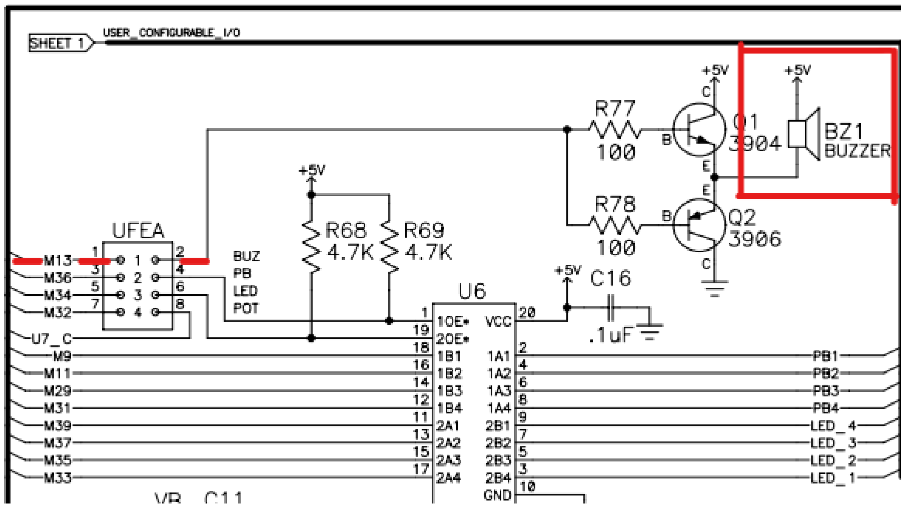


Figure 2: Buzzer Connection on the Schematic (1/3)

Then we started tracking this connection:

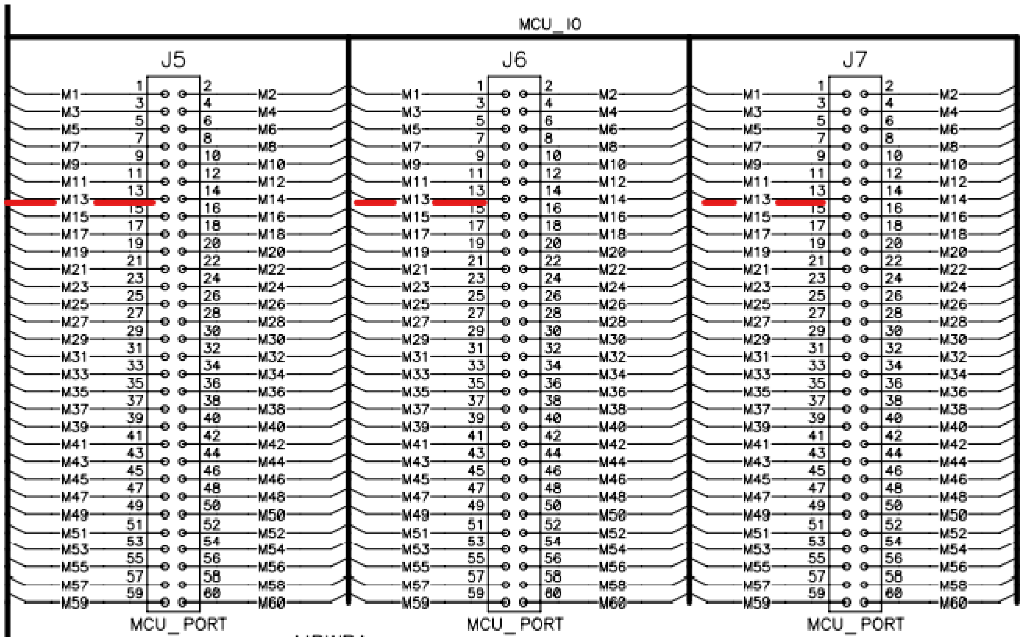


Figure 3: Buzzer Connection in the Schematic (2/3)

If we look to our board, we can realize that the MCU is connected to J6 through J1. So, the pins at J1 are directly connected to the pins of the J6. Thus, the 13th pin of J6 representing the connection to the buzzer, is directly connected to the 13th pin of J1 which is PT0/IOC0. Note that this is inside another datasheet which is PBMCUSLK_SCH_D_0

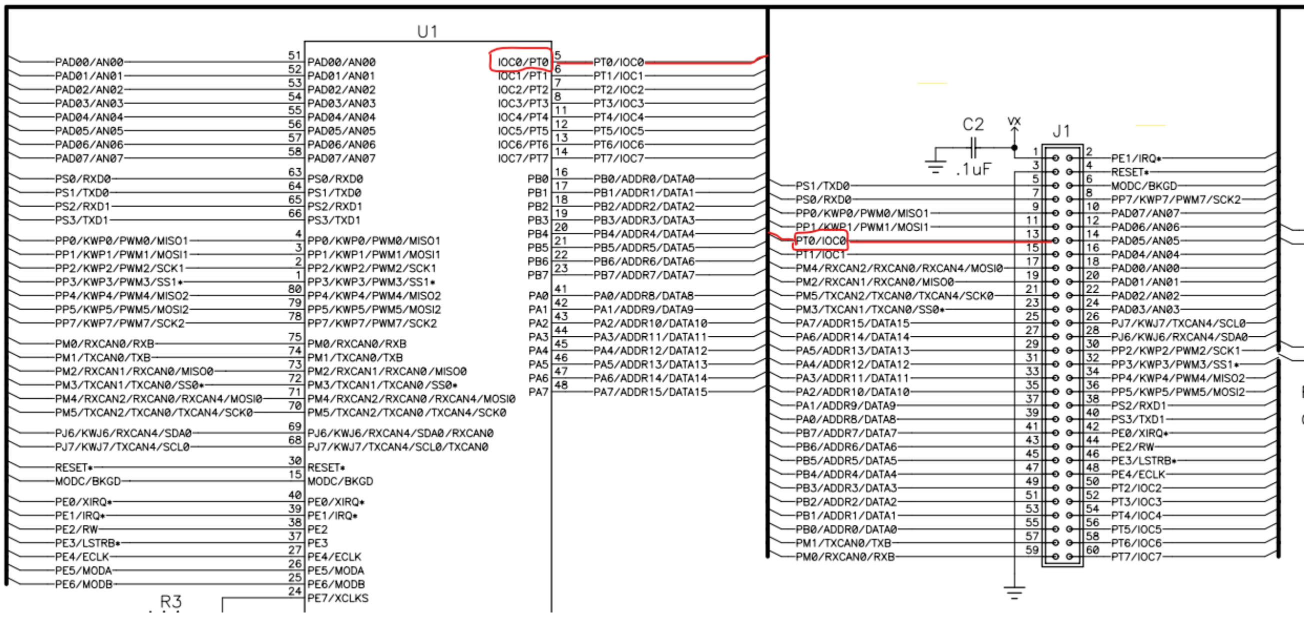


Figure 4: Buzzer Connection in the Schematic (3/3)

Therefore, the buzzer is connected to the MCU through the pin PT0. To further investigate this pin, we move into the block diagram of the MCU which contains the silicon components, we find this pin connected to Port T (PTT) which in return is related to the Port T Data Direction Register (DDRT).

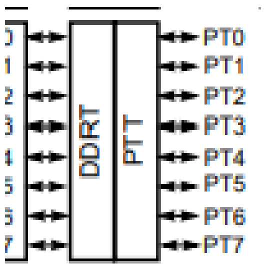


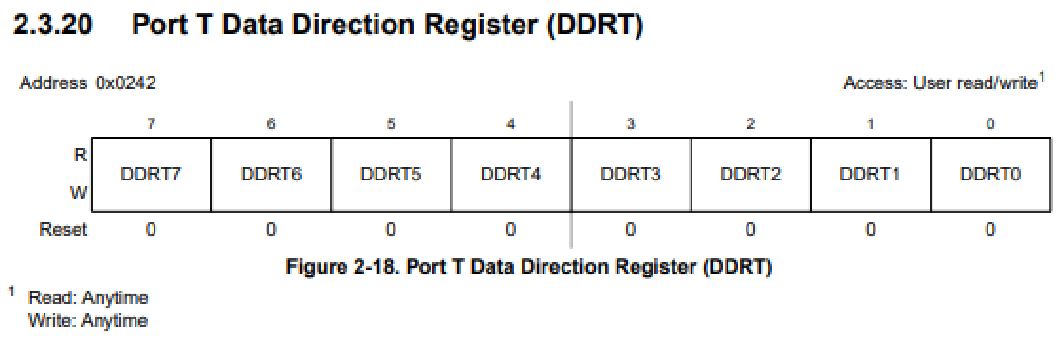
Figure 5: PTT and DDRT in the block diagram

As such, we can investigate the DDRT register and Port T to have better understanding of how to initialize the buzzer.

❖ Configuring the Registers

Since the buzzer is connected to bit PT0 meaning to bit 0 of the port, we are only interested in bit 0 of the Port T and the DDRT

1) DDRT:



2.3.20 Port T Data Direction Register (DDRT)

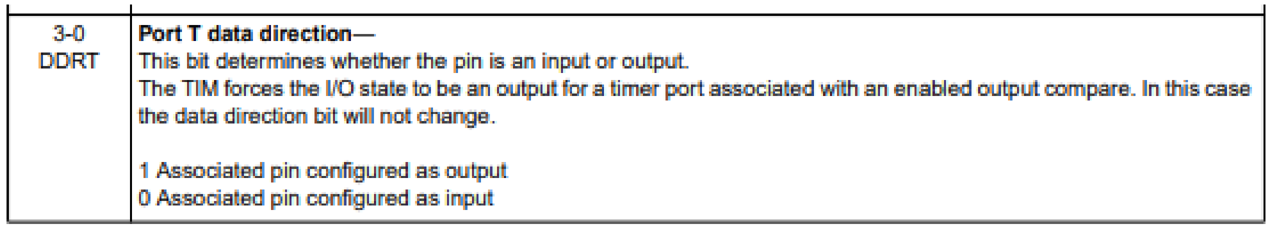
Address 0x0242 Access: User read/write[1]

	7	6	5	4	3	2	1	0
R/W	DDRT7	DDRT6	DDRT5	DDRT4	DDRT3	DDRT2	DDRT1	DDRT0
Reset	0	0	0	0	0	0	0	0

Figure 2-18. Port T Data Direction Register (DDRT)

[1] Read: Anytime
Write: Anytime

Figure 6: DDRT Register

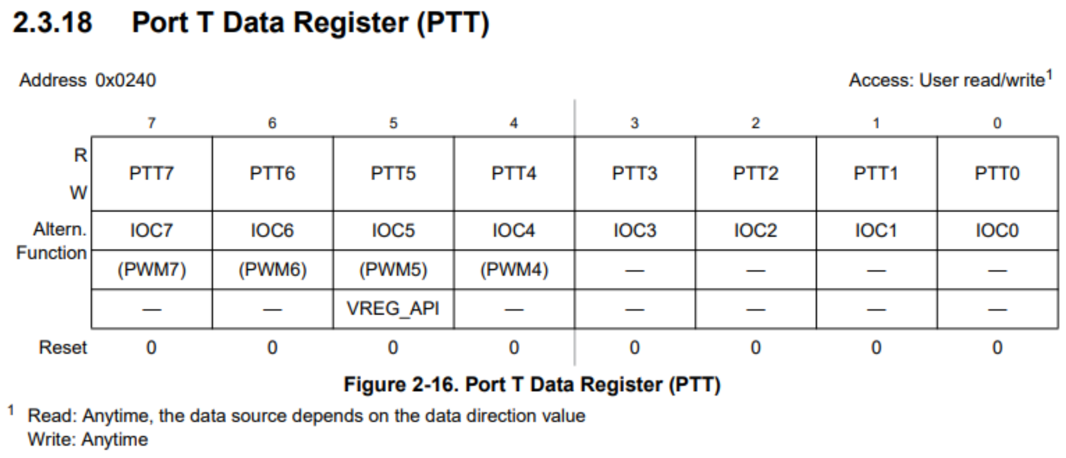


3-0 DDRT	**Port T data direction—** This bit determines whether the pin is an input or output. The TIM forces the I/O state to be an output for a timer port associated with an enabled output compare. In this case the data direction bit will not change. 1 Associated pin configured as output 0 Associated pin configured as input

Figure 7: DDRT Register Description

As was mentioned before, we are only interested in the 0 bit of the DDRT. Since the buzzer should produce a sound as output, we need to configure bit 0 to 1 in order to enable it as output as shown in datasheet in the figure above. That's why we use the instruction: **MOVB #$01, DDRT** configuring the last bit as output.

2) Port T Data Register PTT:



2.3.18 Port T Data Register (PTT)

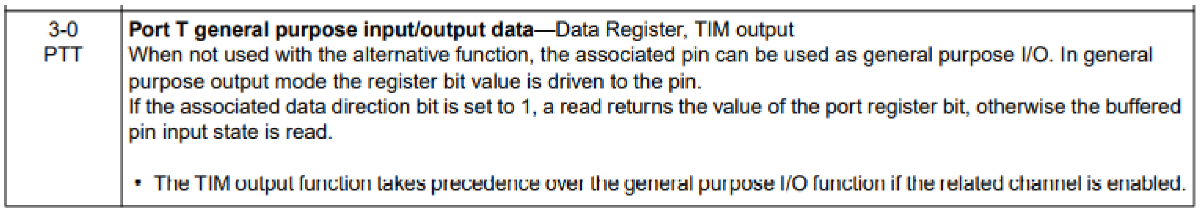
Address 0x0240 Access: User read/write[1]

	7	6	5	4	3	2	1	0
R / W	PTT7	PTT6	PTT5	PTT4	PTT3	PTT2	PTT1	PTT0
Altern. Function	IOC7	IOC6	IOC5	IOC4	IOC3	IOC2	IOC1	IOC0
	(PWM7)	(PWM6)	(PWM5)	(PWM4)	—	—	—	—
	—	—	VREG_API	—	—	—	—	—
Reset	0	0	0	0	0	0	0	0

Figure 2-16. Port T Data Register (PTT)

[1] Read: Anytime, the data source depends on the data direction value
Write: Anytime

Figure 8: PTT Register



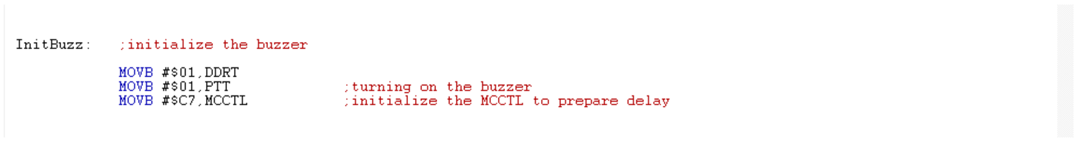
3-0 PTT	**Port T general purpose input/output data**—Data Register, TIM output When not used with the alternative function, the associated pin can be used as general purpose I/O. In general purpose output mode the register bit value is driven to the pin. If the associated data direction bit is set to 1, a read returns the value of the port register bit, otherwise the buffered pin input state is read. • The TIM output function takes precedence over the general purpose I/O function if the related channel is enabled.

Figure 9: PTT Register Description

As mentioned above, to be able to read and return the value of the port register bit, we need to set our bit into 1. That's why we used the instruction: **MOVB #$01, PTT** which turns on our buzzer. This register will derive the clock signal for each music note.

❖ Interrupts for Buzzer:

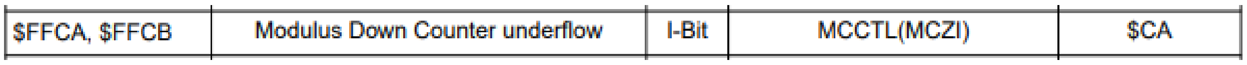
In order to work with buzzer, we need to work inside an interrupt. As was explained previously in Lab 3, in order to create an atmosphere for the interrupt, we rely of MCCTL which should have its bits set to $C7 as explained in the previously mentioned lab. So, we used: **MOVB #$C7, MCCTL.** Thus, the initializing code of the buzzer is as follows:



```
InitBuzz:    ;initialize the buzzer

             MOVB #$01,DDRT
             MOVB #$01,PTT            ;turning on the buzzer
             MOVB #$C7,MCCTL          ;initialize the MCCTL to prepare delay
```

Figure 10: Initializing the Buzzer

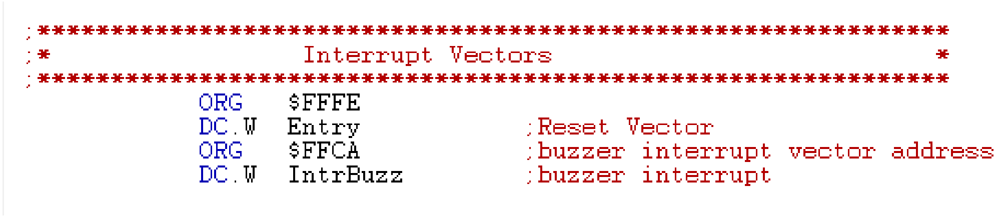
Also, we need to add the interrupt vector to make sure that we have implemented out interrupt. This was also done previously in the lab. The origin of the vector can be found in the Interrupt Registers Reference Datasheet:



$FFCA, $FFCB	Modulus Down Counter underflow	I-Bit	MCCTL(MCZI)	$CA

Figure 11: Interrupt vector for MCCTL

Therefore, we set our interrupt vector as follows:



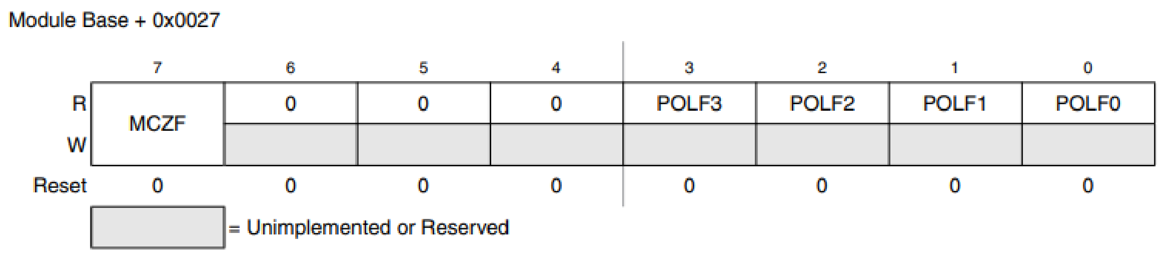
```
;**************************************************************
;*                  Interrupt Vectors                         *
;**************************************************************
              ORG   $FFFE
              DC.W  Entry           ;Reset Vector
              ORG   $FFCA           ;buzzer interrupt vector address
              DC.W  IntrBuzz        ;buzzer interrupt
```

Figure 12: Interrupt Vector code for the buzzer

Now, we go and deal inside the IntrBuzz which will be responsible for creating a clock signal with frequency of the note according to the value given in MCCTL which will be further explained later. In order to create this clock signal, we need to alternate the status of bit 0 of the PTT so it would generate a clock between high and low. As such, if we configure bit 0 of PTT as 1, we are returning a value of the port register as explained previously. This means we have a high voltage value, so our clock is high. When this bit is 1, we are only reading without returning the value which means there is no voltage value, so the clock is zero.

Now in order to change the state of this bit, we used the EOR instruction where we first loaded the status of PTT into A, used the instruction EORA #1 to only exor the last bit of register A, and finally stored this value back into PTT.

Finally, we need to recall the **MCFLG**, short for Modulus Down Counter Flag Register, used inside the interrupt to help achieve it.



Module Base + 0x0027

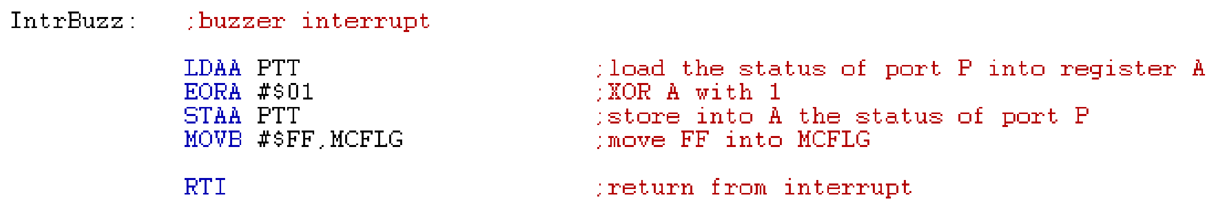
	7	6	5	4	3	2	1	0
R	MCZF	0	0	0	POLF3	POLF2	POLF1	POLF0
W		▨	▨	▨	▨	▨	▨	▨
Reset	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 13: 16-Bit Modulus Down-Counter FLAG Register (MCFLG)

What we are mostly concerned with in this register is the MCZF (the 7th bit). This bit serves as a flag which is set when the modulus down-counter reaches $0000. In other words, when the counter reaches zero, the flag is set. A write 1 to this bit clears this flag whereas a write 0 has no effect. Therefore, we clear it by using **MOVB #$FF, MCFLG** when we use the RTI instruction.

Therefore, the code is as follows:



```
IntrBuzz:    ;buzzer interrupt

             LDAA PTT                        ;load the status of port P into register A
             EORA #$01                       ;XOR A with 1
             STAA PTT                        ;store into A the status of port P
             MOVB #$FF,MCFLG                 ;move FF into MCFLG

             RTI                             ;return from interrupt
```

Figure 14: IntrBuzz Code

For this interrupt to work, we need to specify the duration of this interrupt. This is done by loading into MCCNT, which acts as the down counter for the interrupt, a value represented by the duration for the high clock of each music note the buzzer has to play. This value can be retrieved to the below equation which was developed in Lab 3:

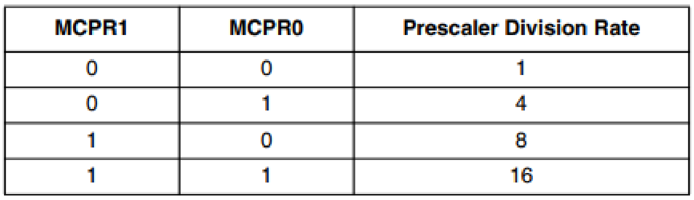
$$\boldsymbol{value\ into\ the\ MCCNT = \frac{f_{oscillator}}{16} \times duration}$$

Where:

- **16: A recall to Lab 3**

The value 16 comes from the Modulus Counter which is a 16-bit control register, that, as the name suggests, counts up or down a certain value. Bits MCPR1 and MCPR0 are the Modulus Counter Prescaler Select. These bits specify the prescaler division rate of the modulus counter.

This prescaling scales the frequency. If we want to prescale the frequency as large as possible, we need to set MCPR1 and MCPR0 to 11 which would prescale the frequency with a factor of 16. On the contrary, setting MCPR1 and MCPR2 to 00 would prescale the frequency with a factor of 1.



MCPR1	MCPR0	Prescaler Division Rate
0	0	1
0	1	4
1	0	8
1	1	16

Figure 15: Modulus Counter Prescaler Select

- ***$f_{oscillator}$:***

This is the frequency of the oscillator we obtained from lab 1 as 4KHz.

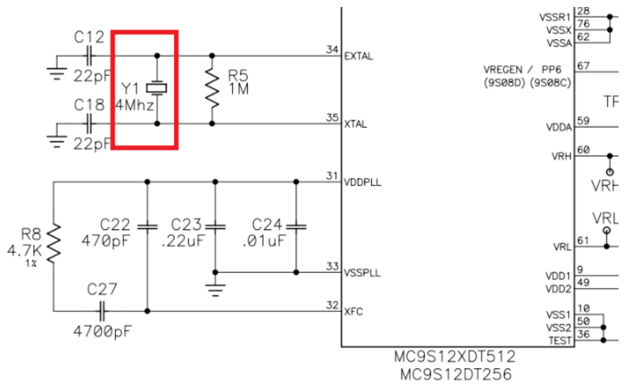


Figure 16: Local Oscillator on Schematic

- **Duration:**

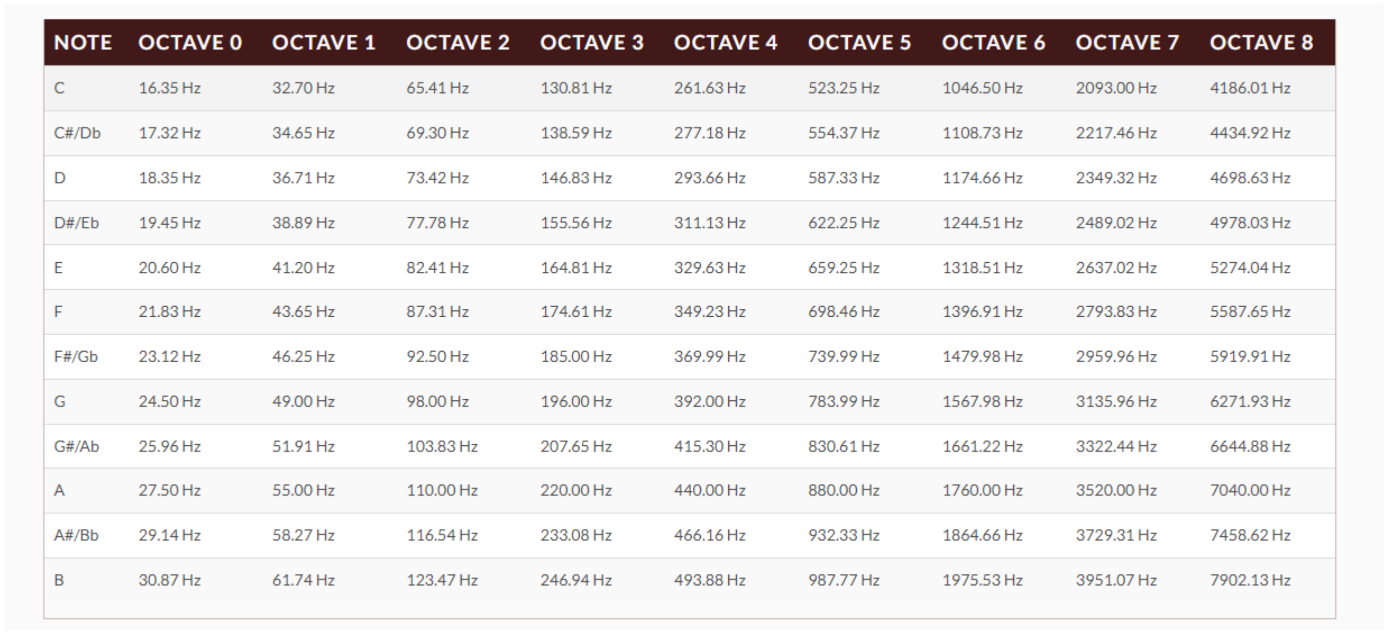
This duration depends on the period of each note. To be more precise, each note will have a frequency and will be driven by a clock of period T. However, half of the period, the clock will be low (as explained previously). Therefore, the duration will be equal to half of the period of the musical note's clock (T/2); here is where the clock is high and the buzzer has to play the note.

❖ Frequencies and Durations for each Musical Note

In order to get the duration of each note, we need to, first, get the frequency which will lead us to the period of the node (T=1/f) and then divide the period by 2 as mentioned previously.

$$duration = \frac{T_{note}}{2} \; where \; T_{note} = \frac{1}{f_{note}}$$

As per of our research, we figured out that each note has its own frequency which also depends on the octave the notes are played on. This is shown in the below figure:



NOTE	OCTAVE 0	OCTAVE 1	OCTAVE 2	OCTAVE 3	OCTAVE 4	OCTAVE 5	OCTAVE 6	OCTAVE 7	OCTAVE 8
C	16.35 Hz	32.70 Hz	65.41 Hz	130.81 Hz	261.63 Hz	523.25 Hz	1046.50 Hz	2093.00 Hz	4186.01 Hz
C#/Db	17.32 Hz	34.65 Hz	69.30 Hz	138.59 Hz	277.18 Hz	554.37 Hz	1108.73 Hz	2217.46 Hz	4434.92 Hz
D	18.35 Hz	36.71 Hz	73.42 Hz	146.83 Hz	293.66 Hz	587.33 Hz	1174.66 Hz	2349.32 Hz	4698.63 Hz
D#/Eb	19.45 Hz	38.89 Hz	77.78 Hz	155.56 Hz	311.13 Hz	622.25 Hz	1244.51 Hz	2489.02 Hz	4978.03 Hz
E	20.60 Hz	41.20 Hz	82.41 Hz	164.81 Hz	329.63 Hz	659.25 Hz	1318.51 Hz	2637.02 Hz	5274.04 Hz
F	21.83 Hz	43.65 Hz	87.31 Hz	174.61 Hz	349.23 Hz	698.46 Hz	1396.91 Hz	2793.83 Hz	5587.65 Hz
F#/Gb	23.12 Hz	46.25 Hz	92.50 Hz	185.00 Hz	369.99 Hz	739.99 Hz	1479.98 Hz	2959.96 Hz	5919.91 Hz
G	24.50 Hz	49.00 Hz	98.00 Hz	196.00 Hz	392.00 Hz	783.99 Hz	1567.98 Hz	3135.96 Hz	6271.93 Hz
G#/Ab	25.96 Hz	51.91 Hz	103.83 Hz	207.65 Hz	415.30 Hz	830.61 Hz	1661.22 Hz	3322.44 Hz	6644.88 Hz
A	27.50 Hz	55.00 Hz	110.00 Hz	220.00 Hz	440.00 Hz	880.00 Hz	1760.00 Hz	3520.00 Hz	7040.00 Hz
A#/Bb	29.14 Hz	58.27 Hz	116.54 Hz	233.08 Hz	466.16 Hz	932.33 Hz	1864.66 Hz	3729.31 Hz	7458.62 Hz
B	30.87 Hz	61.74 Hz	123.47 Hz	246.94 Hz	493.88 Hz	987.77 Hz	1975.53 Hz	3951.07 Hz	7902.13 Hz

Figure 17: Notes Frequency according to Octaves

Note that in the figure above, the ABC notion, which is mostly used in the United States, is employed and it can be easily solfege system (DO, RE, MI..) according to the below table:

Solfege System	**ABC**
Do	C
Re	D
Mi	E
Fa	F
Sol	G
La	A
Si	B

Table 1: ABC and Solfege System Equivalence

To have everything clear, an octave is the distance between one note and the next note of the same name. When we move from one octave to another, the frequency doubles. The middle

However, this buzzer does not work for the Octave 4 as it produces some heavy sound with a low pitch. We realized that we needed to increase the octave for a better sound as per of our research. Therefore, Octave 5 was the ultimate and best choice.

Therefore, the buzzer works on the octave 5 which makes the frequency of each note as follows:

Notes	Frequency (Hz)
Do	523.25
Re	587.33
Mi	659.25
Fa	698.46
Sol	783.99
La	880
Si	987.77

Table 2: Frequency of each note

We calculated then the value to be loaded into MCCNT for each note by the formula: which was developed previously.

$$\boldsymbol{value\ into\ the\ MCCNT = \frac{f_{oscillator}}{16} \times duration}$$

$$\boldsymbol{duration = \frac{T_{note}}{2}\ where\ T_{note} = \frac{1}{f_{note}}}$$

Notes	Frequency (Hz)	Period (T =1/f) (s)	Duration= T/2 (s)	Value into MCCNT
Do	523.25	0.001911132	0.000955566	238
Re	587.33	0.00170262	0.00085131	212
Mi	659.25	0.001516875	0.000758438	190
Fa	698.46	0.001431721	0.000715861	178
Sol	783.99	0.001275526	0.000637763	160
La	880	0.001136364	0.000568182	142
Si	987.77	0.001012381	0.000506191	126

Table 3: Value into MCCNT for each note

So now, when we want to play each note, we need to first move into the MCCNT the corresponding value.

Part II: Methodology:

To tackle the main objective of the project, it was crucial to examine the musical notes and their relative length and frequencies. Following the modern staff notation, the duration or time length of a specific note is determined by how long it lasts relative to the other notes.

1. **Whole note:** A whole note in musical notation is equivalent to or lasting as long as two half notes or four quarter-notes. A whole note represents 4 beats

Figure 18: Whole note

2. **Half note:** A whole note in musical notation is equivalent to or lasting as long as two quarter notes. A half note represents two beats

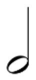


Figure 19: Half note

3. **Quarter note:** A quarter note is a note that is played for one quarter of the duration of a whole note. A quarter note represents one beat

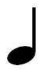


Figure 20: Quarter note

Notes	Name		Value
𝅝	Semibreve	Whole note	4 beats
𝅗𝅥	Minim	Half note	2 beats
♩	Crotchet	Quarter note	1 beat
♪	Quaver	Eighth note	½ beat
𝅘𝅥𝅯	Semi-quaver	Sixteenth note	¼ beat
♫	2 Quavers	2 Eighth notes	1 beat
♬	4 Semi-quavers	4 Sixteenth notes	1 beat

Figure 21: Musical Notes and their Beat Values

For this project, we chose the songs, *Au Clair de la Lune* and *I Love You (Barney Song)*. As such, we have chosen the verses "Au clair de la lune mona mi Pierrot" of the first song and "I love you you love we're a happy family" of the second song.

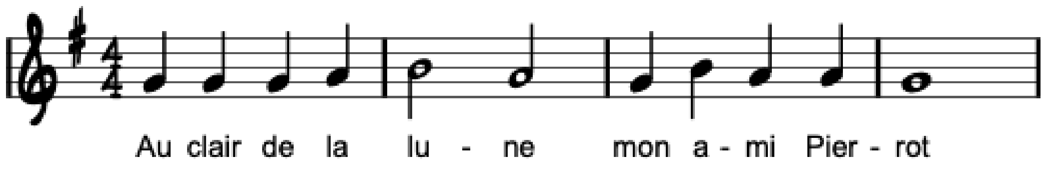


Figure 22: Song 1 Notes

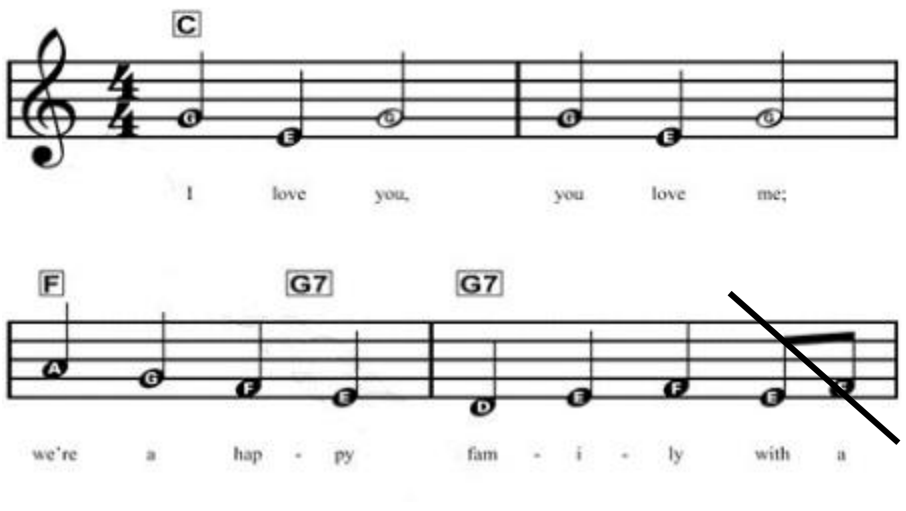


Figure 23: Song 2 Notes

Bearing the above information in mind, we can begin with our methodology. Since we have been asked to utilize the LCD in order to display the songs' lyrics and the push buttons to alternate between the songs, as a first step, it is intuitive that we used previous implementations of both from the Labs we worked on before:

1. Initializing the SPI, LCD, and push buttons, including the sending data, instructions, and a byte through the SPI are the blocks of code required.
2. Configuring the LCD Delay as well as the push buttons interrupt vector.
3. Including the initialization of the buzzer plus its relative interrupt vector, as explained in the above section.

Now that we have gotten the intuitive steps out of the way, we proceed with the rest of our methodology which is examining the notes of the two songs. We realize that three recurring notes are the Whole note (semibreve), Quarter note (crotchet), and the Half note (minim). In order to play the note for a specified period of time, we implemented delays for each, relative to the reference note. Since it is recurring the most – we chose the Quarter note as reference. Ultimately, the Half note would include two times the delay and the Whole note would include four times the delay of the Quarter note.

The next thing to bear in mind is the lyrics sent with each note. One of our main objectives was to synchronize the lyrics with the notes being played. Synchronization in code logic would translate to sending the exact letters/characters along with their corresponding note. As seen in codes of previous labs, sending characters/instructions would yield to a delay with each character/instruction being sent. For the sake of this code, this delay would yield a significant benefit. Each note being played requires a specific amount of time that determines how long it lasts, as mentioned in the previous paragraph. This can be achieved through the delays resulting from the sending data/instruction subblocks, without the need of a new delay subroutine. To determine the exact amount of delay needed to be sent with each note, we followed the proceeding methodology.

Examining figures 22 and 23, we deduced that the maximum number of letters associated with each note is four. As a result, we took four characters as reference to the amount of delay to be implemented. Since our maximum word to be sent out with one note is four, we would branch to the "send character" subroutine **four** times, resulting in **eight "LCD Delay"** branches. And since the four-character words are mostly sent with a Quarter note, we established a reference delay to our reference note (Quarter note). Then, following the musical standard explained above, we can safely deduce that the Half note would include 16 LCD Delays and the Whole note would include 32 LCD Delays. However, it is important to note that the word "clair" was considered as an exception. This decision was based on an attempt to establish a reference of five "send character" delays (equivalent to 10 LCD Delays) and realizing that it would result in an undesired slower tempo. To compensate for it and retain the four "send character" reference, we decided to send the "c" character along with the previous note.

We can also notice some of the notes are accompanied by one, two, or three characters only and not four. Therefore, to respect the musical standard of note lengths and preserve consistency, we would include specific number of branching to "LCD Delay" that would meet the reference we have set. For instance, the first note of the third bar in song 1 is accompanied by the lyric "mon". The musical note – Quarter note – is accompanied by three letters which would yield three "send data" branches and not four following the reference set. In order not to break the synchronization we've set in terms of lyrics, as in sending out the next character of the next word with "mon", as well as respecting the reference, we would include 2 more LCD Delays (an equivalent of 1 send data instruction in terms of delay). This would result in a total of 4 Send Data delays or equivalently 8 LCD Delays, adjusting back to the reference we set and respecting the musical standard of the length of a Quarter note.

Finally, we were asked to work with push buttons to alternate between the two songs. For that, we utilized the subroutine that would perform this check **twice** during every note. We included this subroutine twice in order to minimize the lag when pressing one of the buttons. In other words, if we were to include *one* branching to the check subroutine, we might encounter a case where the checking was done seconds before actually pressing the button. So, in order to avoid such a case and ensure that no lag would occur, we included the branching to the check subroutine *twice* in *every* note. Also, it is important to mention that the delays caused by this branching would be minimal and it is safe to say that the same desired tempo is still achieved.

Another thing worth mentioning is that when one of the buttons is pressed, the check subroutine would branch to either of two additional subroutines whose sole purpose is to return both display and cursor to the original position (first line / address 0) and then branch to the desired song. This sort of implementation was chosen based on the fact that including this instruction at the beginning of the check subroutine *itself* would always return the cursor to the first line even when the buttons are *not* pressed, which is definitely not desired. Also, placing this address at the beginning of the songs would lead to an undesired delay at the beginning of each song, meaning that the songs would take too long to start which is also undesirable.

In short, through this methodology, we were able to achieve synchronization of the lyrics and notes and follow the musical standard length of each note relative to the other. The latter is done by choosing a reference delay for one note and deriving the delay for the other note through the relationship that the different notes have with respect to each other. We have also achieved synchronization of lyrics by sending the specific words associated with each note.

Part III: Code

The two songs that we decided to play using the buzzer were chosen because they cover most types of notes: Do Re Mi Fa Sol La, except Si.

For every note in our songs we used the following LED toggling combination, which turn on at the beginning of the note and turns off when the note ends:

1. **For Do:** Only LEDs 2 and 4 are turned on: using **MOVB #$A0, DDRB**
2. **For Re:** Only LEDs 1 and 3 are turned on: using **MOVB #$50, DDRB**
3. **For Mi:** Only LEDs 1 and 4 are turned on: using **MOVB #$90, DDRB**
4. **For Fa:** Only LEDs 2 and 3 are turned on: using **MOVB #$60, DDRB**
5. **For Sol:** Only LEDs 3 and 4 are turned on: using **MOVB #$C0, DDRB**
6. **For La:** Only LEDs 1 and 2 are turned on: using **MOVB #$30, DDRB**
7. **For Si:** Only LEDs 2, 3 and 4 are turned on: using **MOVB #$E0, DDRB**

❖ **The delays:**

To reiterate how our delays work in a clearer manner:

- The delay of a quarter note is equal to 4 SENDATA/SENDINST delays, and since they each contain 2 LCD Delay, the delay of a quarter note is equal to 8 LCD Delay.
- The delay of a half note is equivalent to two quarter notes, which is equivalent to eight SENDATA/SENDINST delays, and because they each contain two LCD Delay, the delay of a half note is equivalent to 16 LCD Delay.
- The delay of a full note is equivalent to two half notes, which is equivalent to 16 SENDATA/SENDINST delays, and because they each contain two LCD Delay, the delay of a full note is equivalent to 32 LCD Delay.

❖ Playing the Songs

➢ Song 1:

The notes of the first song are the following:

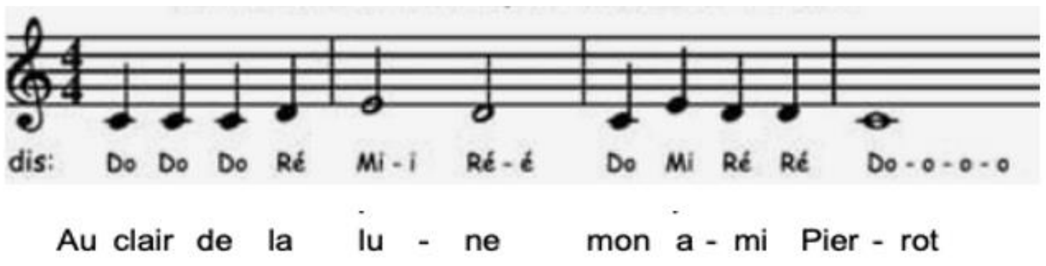


Figure 24: Notes of the first song (Au clair de la lune)

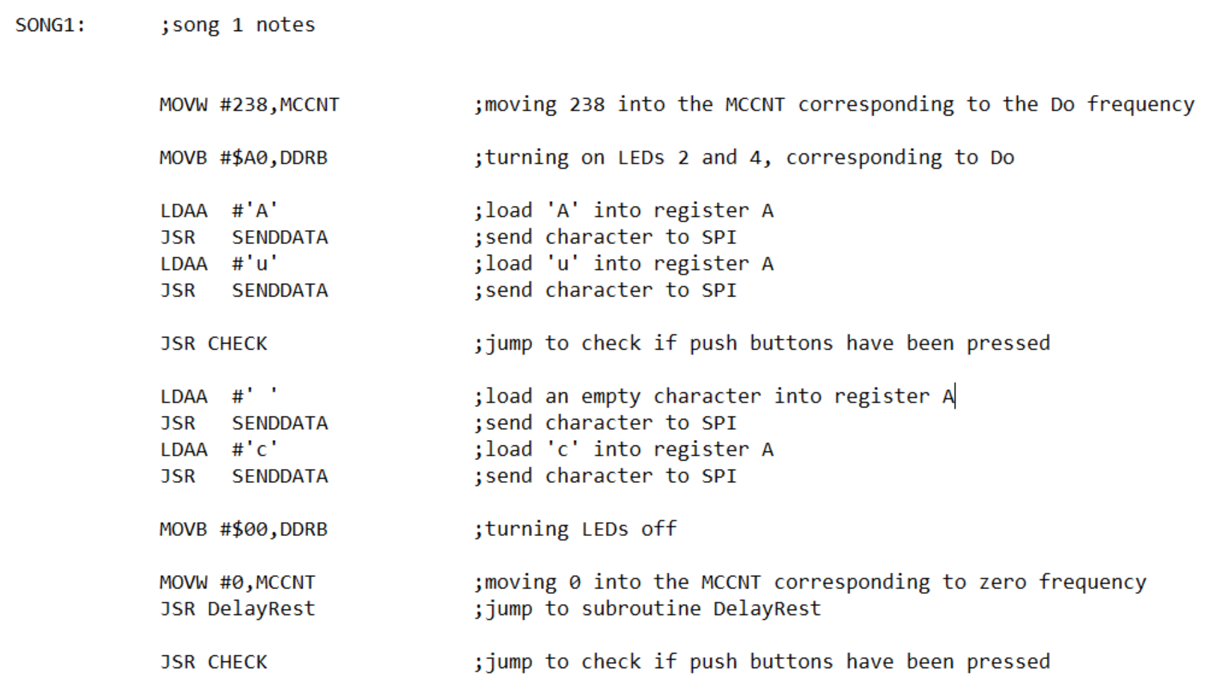
The above song's notes are four quarter notes, as shown in the diagram above: Do Do Do Re followed by two half notes: Mi, Re. Then there are four quarter notes: Do Mi Re Re. And this song ends with a whole note Do. This song was chosen because it covers the three types of notes, it contains consecutive notes, and it includes three notes, which are: Do, Re, and Mi with consecutive notes such as the first 3 Do.

- **To play the first quarter note Do:**

As previously stated, in order to play the Do quarter, we first moved to the interrupt counter MCCNT 238 using: **MOVW #238, MCCNT**. Then we activated the LED combination corresponding to the Do note: **MOVB #$A0, DDRB.** And we added a quarter-note delay while printing the song's lyrics. To do so, we printed the song's lyrics: "Au" which corresponds to the note being played. We first loaded the first letter, "A," into register A, then sent the character to SPI using the SENDATA subroutine to print it on the LCD, which contains two LCD Delay, then loaded the second character, "u" and printed it then an empty character, and finally printed the character "c" from clair to achieve four SENDDATA delays, which corresponds to a quarter note. Furthermore, we turned off the LEDs: **MOVB #$00, DDRB.** Finally, to distinguish between two consecutive **same** notes and stop playing the first one, we moved to MCCNT 0 using: **MOVW #0, MCCNT**, which loads a frequency 0 into it, and we jumped to a small delay using JSR DelayRest to hear the 0-frequency note. Note that between the SENDDATAs and delays in each note we included two button checks which checks what button is pressed and play the song accordingly by branching to a check subroutine.

We played the rest quarter notes using the same approach as we did to play the first one.

Below is a code sample of the explained note:



```
SONG1:      ;song 1 notes

            MOVW #238,MCCNT          ;moving 238 into the MCCNT corresponding to the Do frequency

            MOVB #$A0,DDRB           ;turning on LEDs 2 and 4, corresponding to Do

            LDAA  #'A'               ;load 'A' into register A
            JSR   SENDDATA           ;send character to SPI
            LDAA  #'u'               ;load 'u' into register A
            JSR   SENDDATA           ;send character to SPI

            JSR CHECK                ;jump to check if push buttons have been pressed

            LDAA  #' '               ;load an empty character into register A
            JSR   SENDDATA           ;send character to SPI
            LDAA  #'c'               ;load 'c' into register A
            JSR   SENDDATA           ;send character to SPI

            MOVB #$00,DDRB           ;turning LEDs off

            MOVW #0,MCCNT            ;moving 0 into the MCCNT corresponding to zero frequency
            JSR DelayRest            ;jump to subroutine DelayRest

            JSR CHECK                ;jump to check if push buttons have been pressed
```

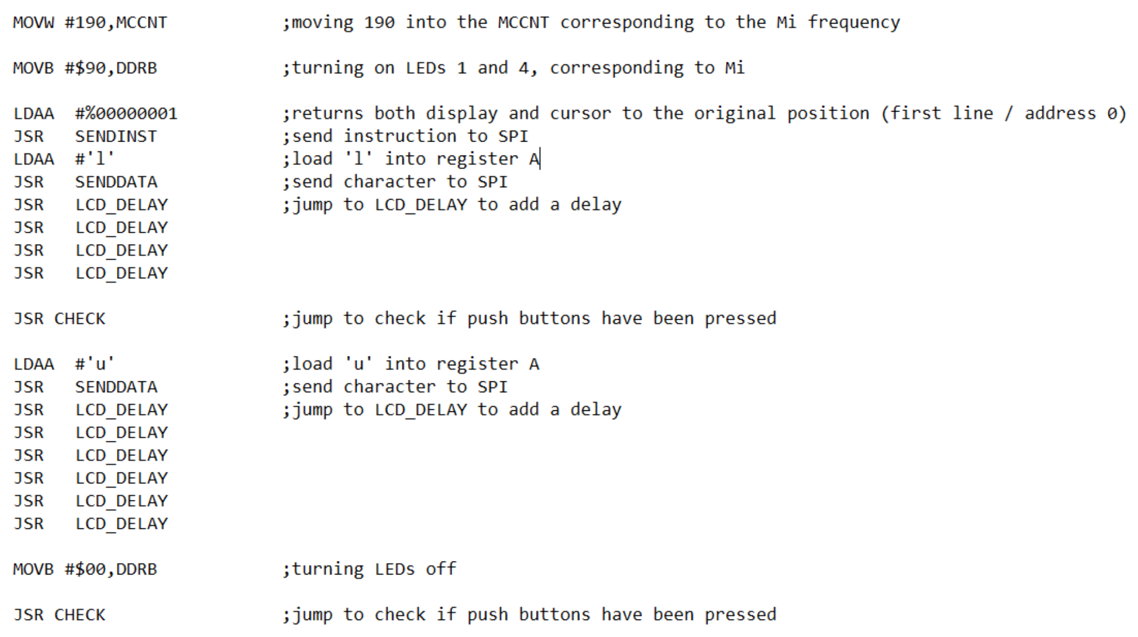
Figure 25: Code of Playing the Do note of the first song

- **To play the half note Mi:**

We first moved to the interrupt counter MCCNT 190 using: MOVW #238, MCCNT. Then we activated the LED combination corresponding to the Mi note: MOVB #$00, DDRB**.** And we added a half-note delay while printing the song's lyrics. We moved both display and cursor to the original position (first line / address 0) which introduce 1 SENDINST delay since the screen is full. To print the lyrics of the song: "lu" which corresponds to the note being played we first loaded the first letter which is "l" into register A then we send character to SPI using the SENDATA subroutine which contains 2 LCD_Delay then we loaded the second character "u". To achieve 8 SENDDATA delays which corresponds to a half note we added 10 LCD_Delay which corresponds to 5 SENDDATA delay. Moreover, we turned off the LEDs using **MOVB #$00, DDRB.** Finally, in order to differentiate between two consecutive notes and stop playing the first note we used: **MOVW #0, MCCNT** which loads a frequency 0 into it and we jumped to a small delay using **JSR DelayRest** in order to recognize and give some time for this 0-frequency note. Please note between the SENDDATA/SENDINST in each note we included two switches checks which checks what switch is pressed and play the song accordingly by branching to a check subroutine.

We played the rest half notes using the same approach as we did to play the first one.

The following is screenshot of the code of the explained note:



```
MOVW #190,MCCNT          ;moving 190 into the MCCNT corresponding to the Mi frequency

MOVB #$90,DDRB           ;turning on LEDs 1 and 4, corresponding to Mi

LDAA  #%00000001         ;returns both display and cursor to the original position (first line / address 0)
JSR   SENDINST           ;send instruction to SPI
LDAA  #'l'               ;load 'l' into register A
JSR   SENDDATA           ;send character to SPI
JSR   LCD_DELAY          ;jump to LCD_DELAY to add a delay
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY

JSR CHECK                ;jump to check if push buttons have been pressed

LDAA  #'u'               ;load 'u' into register A
JSR   SENDDATA           ;send character to SPI
JSR   LCD_DELAY          ;jump to LCD_DELAY to add a delay
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY

MOVB #$00,DDRB           ;turning LEDs off

JSR CHECK                ;jump to check if push buttons have been pressed
```

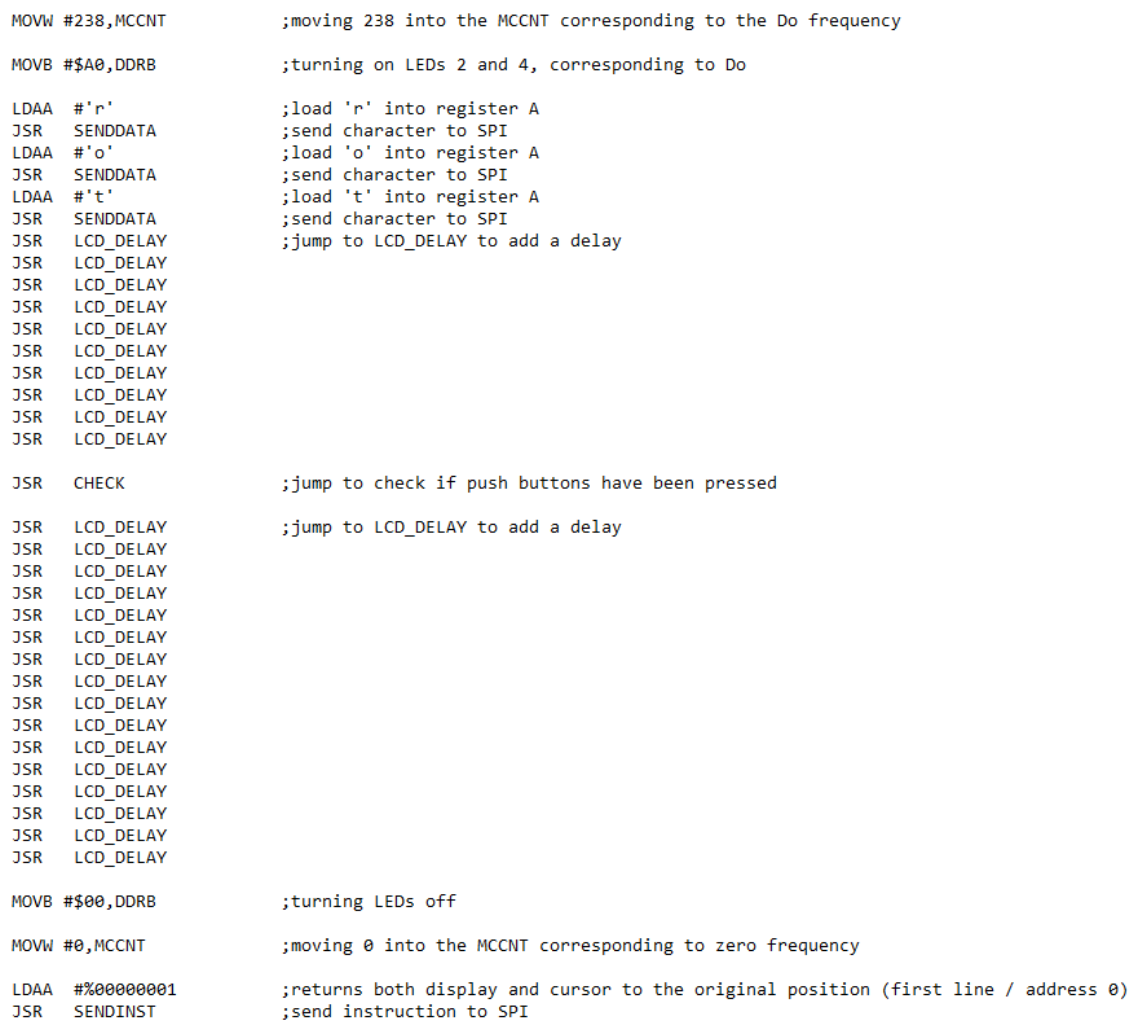
Figure 26: Code of Playing the Mi note of the first song

- **To play the full note Do at the end of the song:**

We first moved to the interrupt counter MCCNT 238 using: **MOVW #238, MCCNT**. Then we activated the LED combination corresponding to the Do note: **MOVB #$A0, DDRB.** And we added a full-note delay while printing the song's lyrics on the LCD_Delay. To print the lyrics of the song: “rot” which corresponds to the note being played we first loaded the first letter which is “r” into register A then we send character to SPI using the SENDATA subroutine which contains 2 LCD_Delay then we loaded the second character “o” and lastly the letter “t”. To achieve 16 SENDDATA delays in total, which corresponds to a half note we added 26 LCD_Delay equivalent to 13 SENDDATA delays. Moreover, we turned off the LEDs using **MOVB #$00, DDRB**. Finally, in order to differentiate between two consecutive notes and stop playing the first note we moved to MCCNT 0: **MOVW #0, MCCNT** which loads a frequency 0. Finally, we returned both display and the cursor to the original position (first line / address 0) using **LDAA**

#%00000001, JSR   SENDINST which will introduce small delays to hear the 0-frequency note. Please note between the lyrics and the delays we included two switches checks which checks what switch is pressed and play the song accordingly by branching to a check subroutine.

Below a code sample of the explained note:



```
MOVW #238,MCCNT          ;moving 238 into the MCCNT corresponding to the Do frequency

MOVB #$A0,DDRB           ;turning on LEDs 2 and 4, corresponding to Do

LDAA  #'r'               ;load 'r' into register A
JSR   SENDDATA           ;send character to SPI
LDAA  #'o'               ;load 'o' into register A
JSR   SENDDATA           ;send character to SPI
LDAA  #'t'               ;load 't' into register A
JSR   SENDDATA           ;send character to SPI
JSR   LCD_DELAY          ;jump to LCD_DELAY to add a delay
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY

JSR   CHECK              ;jump to check if push buttons have been pressed

JSR   LCD_DELAY          ;jump to LCD_DELAY to add a delay
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY
JSR   LCD_DELAY

MOVB #$00,DDRB           ;turning LEDs off

MOVW #0,MCCNT            ;moving 0 into the MCCNT corresponding to zero frequency

LDAA  #%00000001         ;returns both display and cursor to the original position (first line / address 0)
JSR   SENDINST           ;send instruction to SPI
```

Figure 27: code sample of the full note Do

➢ Song 2:

The notes of the second song are the following:

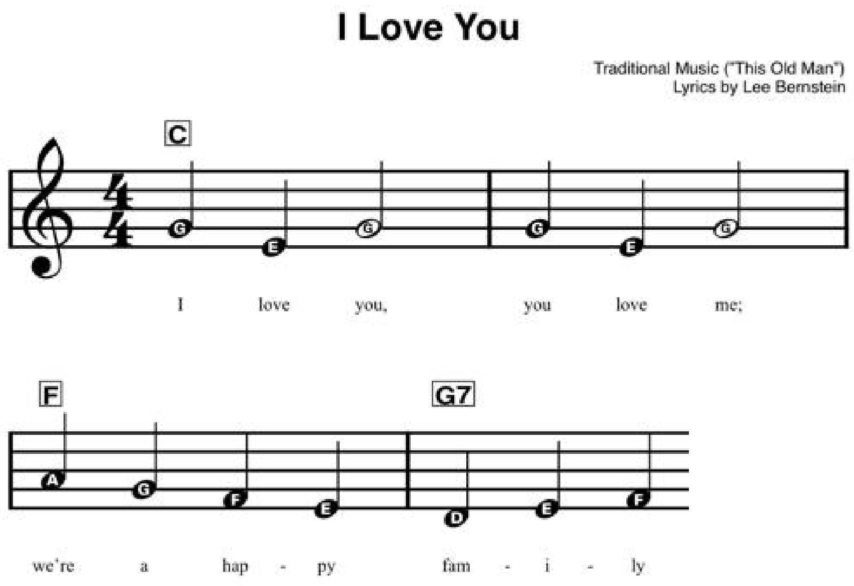
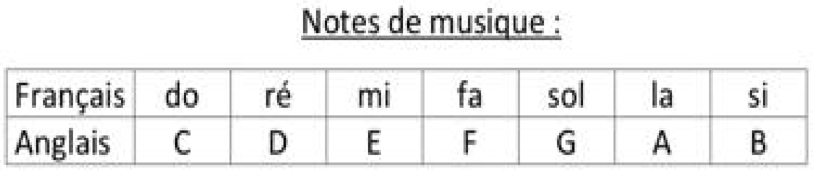

I Love You
Traditional Music ("This Old Man")
Lyrics by Lee Bernstein
C
F
G7
I love you, you love me;
we're a hap - py fam - i - ly

Figure 28: Notes of the second song (I love you, you love me)

The notes in the above song are English notes, their corresponding French notes are the following:



Notes de musique :

Français	do	ré	mi	fa	sol	la	si
Anglais	C	D	E	F	G	A	B

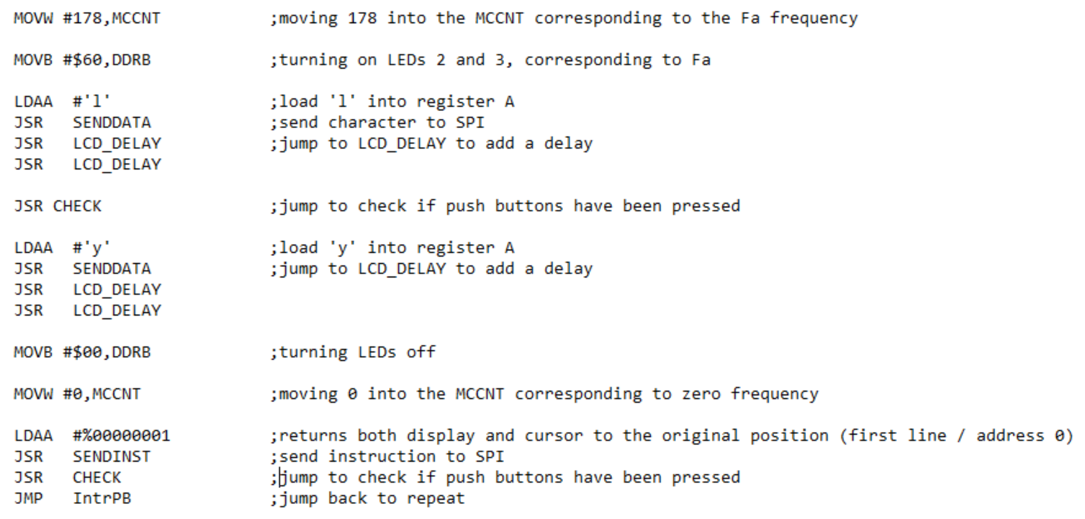
Figure 29: Corresponding French note of the English notes

As shown in the above figure, the notes of the following song are two quarter notes: G and E, which correspond to Sol Mi, followed by one half note, Sol. The same notes are then played again. These notes are then followed by four quarter notes: La Sol Fa Mi. Then there are three quarter notes, Re Mi Fa. This song was chosen because it covers many types of notes, which were not covered in the first song: Fa, Sol, and La. All the quarter and half notes were played the same way we did in the previous song.

- **To play the last note Fa note in the second line**

We first moved to the interrupt counter MCCNT 178 using: **MOVW #178, MCCNT**. Then we activated the LED combination corresponding to the Fa note: **MOVB #$60, DDRB.** And we added a quarter-note delay while printing the song's lyrics on the LCD_Delay. To print the lyrics of the song: “ly” which corresponds to the note being played we first loaded the first letter which is “l” into register A then we send character to SPI using the SENDATA subroutine which contains 2 LCD_Delay then we loaded the second character “y” and print it on the screen. To achieve 4 SENDDATA delays in total corresponding to a quarter note we added 4 LCD_Delay equivalent to 2 SENDDATA delays. Furthermore, we used **MOVB #$00, DDRB** to turn off the LEDs. Finally, in order to differentiate between two consecutive notes and stop playing the note being played we moved to MCCNT 0: **MOVW #0, MCCNT** which loads a frequency 0. Finally, we need to return both display and cursor to the original position (first line / address 0) using **LDAA #%00000001, JSR SENDINST** since it is the last note in our song and which will introduce small delays for the 0 frequency to get heard. Please note between the SENDDATA we included two switches checks which checks what switch is pressed and play the song accordingly by branching to a check subroutine.

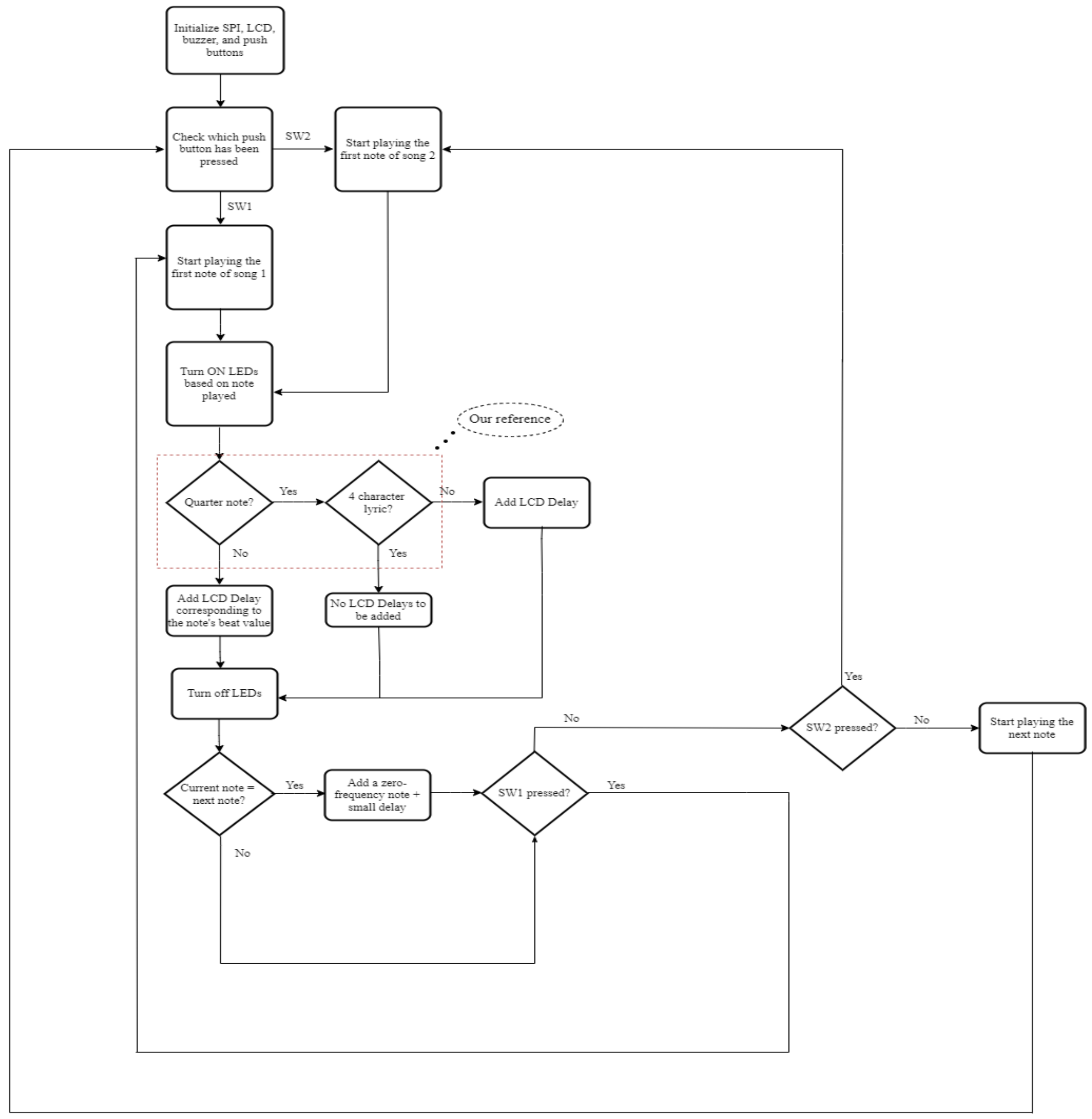
Here is a code sample of the explained note:



```
MOVW #178,MCCNT          ;moving 178 into the MCCNT corresponding to the Fa frequency

MOVB #$60,DDRB           ;turning on LEDs 2 and 3, corresponding to Fa

LDAA  #'l'               ;load 'l' into register A
JSR   SENDDATA           ;send character to SPI
JSR   LCD_DELAY          ;jump to LCD_DELAY to add a delay
JSR   LCD_DELAY

JSR CHECK                ;jump to check if push buttons have been pressed

LDAA  #'y'               ;load 'y' into register A
JSR   SENDDATA           ;jump to LCD_DELAY to add a delay
JSR   LCD_DELAY
JSR   LCD_DELAY

MOVB #$00,DDRB           ;turning LEDs off

MOVW #0,MCCNT            ;moving 0 into the MCCNT corresponding to zero frequency

LDAA  #%00000001         ;returns both display and cursor to the original position (first line / address 0)
JSR   SENDINST           ;send instruction to SPI
JSR   CHECK              ;jump to check if push buttons have been pressed
JMP   IntrPB             ;jump back to repeat
```

Figure 30: Code of Playing the Fa note of the second song

As a summary, the below flowchart would illustrate the steps followed to achieve the project requirements:



Initialize SPI, LCD, buzzer, and push buttons
Check which push button has been pressed
SW2
Start playing the first note of song 2
SW1
Start playing the first note of song 1
Turn ON LEDs based on note played
Our reference
Quarter note?
Yes
4 character lyric?
No
Add LCD Delay
No
Yes
Add LCD Delay corresponding to the note's beat value
No LCD Delays to be added
Turn off LEDs
Yes
No
SW2 pressed?
No
Start playing the next note
Current note = next note?
Yes
Add a zero-frequency note + small delay
SW1 pressed?
Yes
No

Figure 31: Flowchart

Conclusion

At the end of this project, we became familiar with the usage of a new electric component, the buzzer. We were able to combine that with prior knowledge to create a real-world application that can play music based on the inputs of a user-controlled interface consisting of two push buttons. Furthermore, a pre-programmed algorithm was created to control the LEDs' behavior based on the current music note being played. While the songs are playing, the lyrics can be displayed on the LCD.

References:

Automatic music transcription from ... - researchgate.net. (n.d.). Retrieved November 30, 2021, from https://www.researchgate.net/publication/334131988_Automatic_Music_Transcription_From_Monophonic_Signals.

Duration - Note Lengths in Written Music. (2021, April 11). https://human.libretexts.org/@go/page/1421

N. (2021, July 26). *Buzzer: Working, types, circuit, Advantages & Disadvantages*. ElProCus. Retrieved November 30, 2021, from https://www.elprocus.com/buzzer-working-applications/.

Sammy, P. by, & Sammy. (2021, October 24). *Music note frequency chart - music frequency chart*. MixButton. Retrieved November 30, 2021, from https://mixbutton.com/mixing-articles/music-note-to-frequency-chart/.

Axiom Manufacturing. (2007). PBMCUSLK AXM-0392 (n.d)

Axiom Manufacturing. (2008). CSMB12D AXM-0436 (A1)

CHALMERS. (2006). Embedded Systems Instruction Set Overview.(n.d): CHALMERS: Department of Computer Science and Engineering

EGGN 482 – Microcontroller Architecture and Interfacing – Professor William Hoff. (n.d). HCS12 Architecture.(n.d): Colorado School of Mines

Freescale Semiconductor. (2006). MC3S12RG128 Data Sheet (1.06)

Freescale Semiconductor. (2012). MC9S12XS256 Reference Manual (1.13)

Hitachi, Ltd., (1998). HD44780U (LCD-II) (Dot Matrix Liquid Crystal Display Controller/Driver) (0.0): Nippon Bldg., 2-6-2, Ohte-machi, Chiyoda-ku, Tokyo 100-0004, Japan

Motorola, Inc. (2000). SPI Block User Guide (02.06)

Motorola, Inc. (2003). MC9S12DT256 Device User Guide. (03.00)

NXP. (2015). 74HC595; 74HCT595 8-bit serial-in, serial or parallel-out shift register with output latches; 3-state (7)