University of British Columbia
Electrical and Computer Engineering
ELEC 291/ELEC 292 Term 2 2022
Instructor: Dr. Jesus Calvino-Fraga
Section 201

Project 1 – Capacitive Sensor Reaction Game

Student #	**Student Name**	**% Points**	**Signature**
15766819	Danial Jaber	95	DJ
70375001	Mark Tan	100	MT
88693502	Aman Prakash	105	AP

Date of Submission: March 4th, 2022

Table of Contents

Title Page 1

Introduction 3

Investigation 4

Idea Generation 4

Investigation Design 6

Data Collection 7

Data Synthesis 7

Analysis of Results 8

Design 8

Use of Process 8

Need and Constraint Identification 9

Problem Specification 9

Solution Generation 10

Solution Evaluation 11

Safety/Professionalism 12

Detailed Design 12

Solution Assessment 17

Live-Long Learning 18

Conclusions 19

References 20

Bibliography 20

Appendix 1: Relevant Code Blocks 20

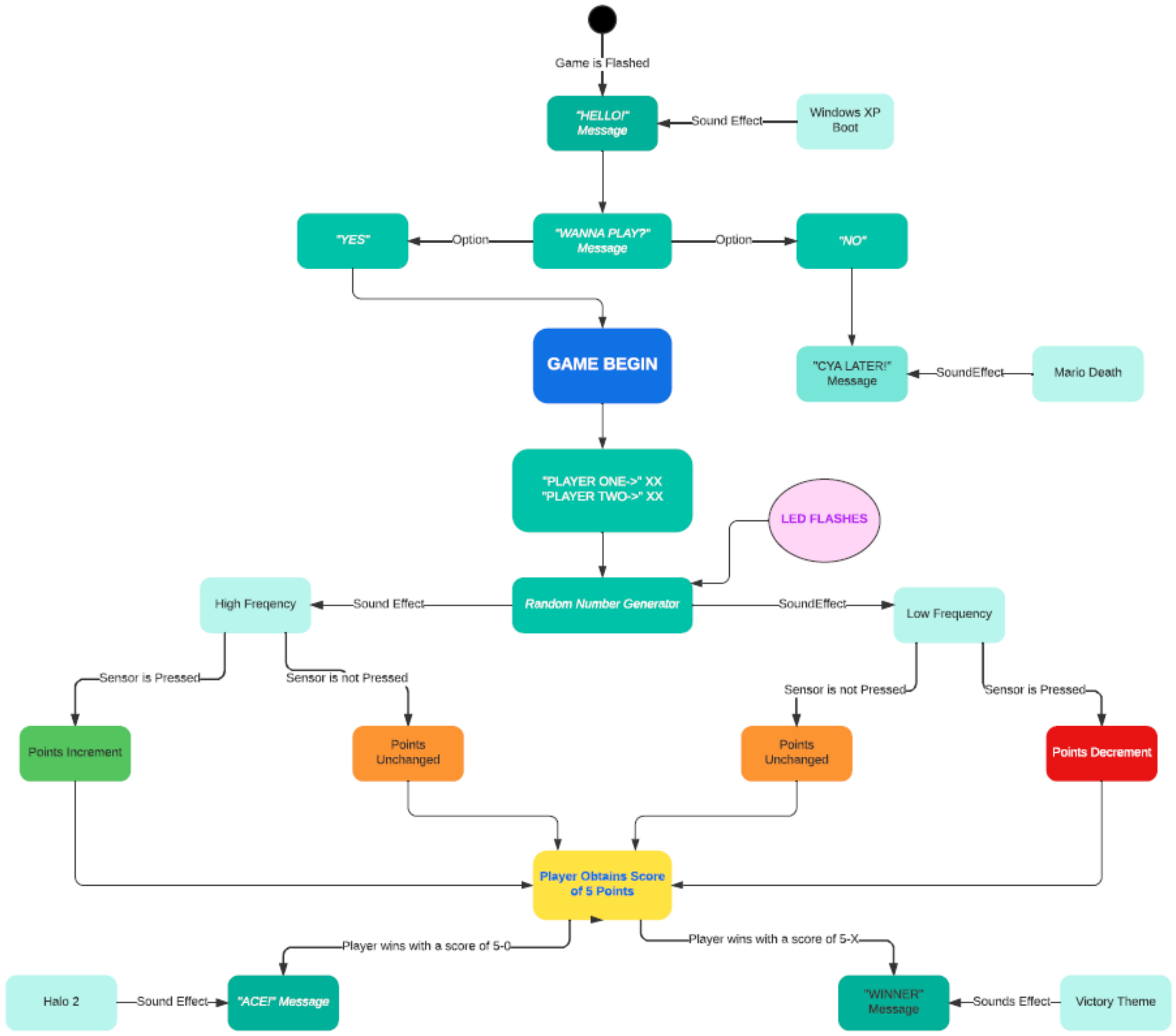
Appendix 2: Music Score Sheets 38

Appendix 3: Pictures of Logic Diagrams, Schematics, and Tables related to Hardware 39

Introduction

Our team engineered a two-person game that rewarded the player with a faster reaction time. Based on a random number generator, a high or low frequency would be outputted on a CEM-1302 speaker. Using di-electric material, we built capacitors that acted as sensors. When a high frequency was emitted, a player would hit their respective sensor to obtain a point. If they hit their sensor during a low frequency, a point would be taken away. These points were kept track of on a Liquid Crystal Display (LCD) and the first player to 5 points would win the game.

This was the design objective and main functionality of the game. We also added bonus features to enhance the experience of the players and make the game more professional. The game would start with a message on the LCD saying “HELLO!” with the Windows XP Boot sound effect. Then the game would ask the prompt “WANNA PLAY?” with the options “YES” and “NO.” Using push buttons, players could control a cursor on the LCD that hovered beside these options and then select them. If a player selected “NO,” a message saying, “CYA LATER!” would be displayed with the Mario Death music being played. If a player selected “YES,” the game would begin with “PLAYER ONE->” and their score on the first line of the LCD along with “PLAYER TWO->” and their score on the second line of the LCD. If a player won the game with a score of 5-0 against their opponent, “ACE!” would be displayed on their respective line along with the Halo 2 theme music being played. If a player won with a regular score, “WINNER” would be displayed on their respective line along with the Victory theme music being played. Our team also thought it would be a unique feature for an LED to flash every time a random number was generated. We also included additional branching in our code to ensure that only the points of the first player to hit their sensor would increment or decrement. We have added the following system block diagram to display the overall design (a PDF of the diagram is also attached if the image is unclear):



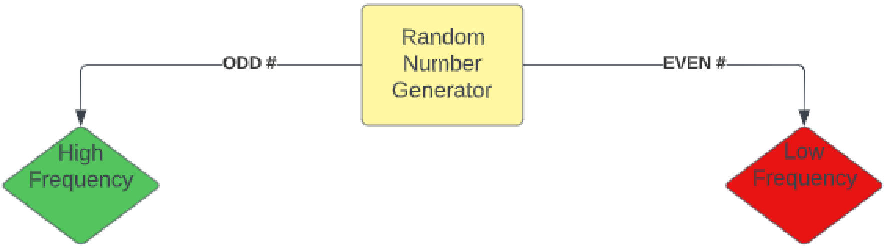
Investigation

Idea Generation

Our team was provided with code for a pseudo-random number generator (RNG) called "Random" along with how to initialize a random seed. We were also provided with knowledge of how to call "Random" to play a tone and a branch (function) called "Wait_Random" that would allow us to use intervals of random time.

This gave our team the idea that by using the RNG, we could decide which frequency would be randomly played. This would prevent the players from getting used to a pattern of frequencies, making the game dull. We also realized that by using “Wait_Random,” we could play each frequency for a random time interval. This would keep players on edge while participating since they would not be able to anticipate when the frequency would change.

Given these tools, we determined that we needed to play a high or low frequency at a specific type of random number that was generated. We developed the following hypothesis: if the number was even, the speaker would emit a low frequency and if the number was odd, the speaker would emit a high frequency. Our idea is represented by the picture below:



We then hypothesized that if we went to a high frequency, the code would check for the sensor being hit and the point incrementing for a player. The same logic would apply for a low frequency and the point decrementing. After some thought, we had discovered that the “Wait_Random” function provided to us would pause the code whenever we called it. This would mean that our team would have to implement a method where the code would check for a point incrementing or decrementing independent of “Wait_Random.” We hypothesized using iterations that would loop through certain branches checking for the sensor being hit a specific number of times. We also needed only the points of the first player who hit the sensor to increment or decrement. This meant we needed to code that would check for a change in a player’s points and skip the other player’s branches where their points would change.

As for our bonus features, we wanted our project to feel like an actual arcade game. This meant that we needed a greeting screen and a prompt asking to continue to the game. We also needed a way to gratify an excellent play along with different sound effects to add character to the game.

Investigation Design

At each developmental step of the programming process, we aimed to find a means of testing. For the RNG generator, we included a test call to actively display each number being generated, allowing us to quickly find any repetitive issues. We also added a red LED which was useful when seeing the frequency of each number being generated and if there were any wait delays causing inconsistencies in the flow of the code. When creating the code to recognize a player triggering the sensor, we had to also add a test call to show the change in the period when sensor pressure was applied. This allowed us to determine a sensitivity threshold that was quite accurate for scoring. These test calls were used to be able to read and analyse data to check for functional code and improve the efficiency or accuracy of different data. Despite these test calls being useful, they were removed once the code was sure to work since gameplay data was reserved for the LCD screen display.

In our gameplay design, we had to add flag code blocks to ensure scoring consistency. When both players triggered their sensor in close time proximity, the score had to be awarded once. **(Appendix 1.h)** In addition, double scores had to be prevented for players who triggered their sensors in quick succession, which was also achieved using specific code blocks. Flag code blocks were also implemented to prevent scoring bounds from being extended over 5 and below 0 as players' scores can go both up and down **(Appendix 1.i)**.

Data Collection

Printing data to the CrossIDE output window is not possible. Therefore, we used the LCD to obtain information on the scores, text, and logic of our assembly code. Additionally, a speaker and red LED were used to obtain more information about our assembly code when implementing new features. The speaker provided sound data that could be used to verify the different frequencies being played. The red LED provided information on when a new random number was generated. For example, we obtained information on our scoring system by displaying the scores of both players on the LCD while pressing the capacitor plates during various high- and low-frequency sounds.

Data Synthesis

Our general process for synthesising the data collected was as follows:

1. Review the data collected from the test against our expected data.
2. If results do not meet expectations, locate the section of code or hardware that may be causing the issue. If they do meet expectations, go to step 5.
3. Review project documents and specifications along with applying changes in the code or hardware to fix the problem.
4. Perform a new test and record the data
5. If results match expectations, continue implementing new features. If not, go to step 2.

Examples: We first checked to see if our scoring system worked without the sound. Only after reviewing the data from the test and being satisfied with the results, we attempted to incorporate the speaker element into our game. After adding the speaker element and testing that the new scoring system worked, we went on to add different branches for different types of end-game scores.

Analysis of Results

Our team assessed the validity of the conclusions from the data synthesis by repeating the tests a reasonable number of times until the results were consistent with our expectations. This would also catch edge cases that we may have missed in our previous tests. For example, to ensure our scoring system was robust, we attempted to have player one’s capacitor plate press slightly after player two’s capacitor plate was pressed during a high-frequency sound and checked that only player two would receive the point during that sound frequency.

Design

Use of Process

Upon beginning the project, the first step was understanding the objective and purpose of the project, while noting down all the files and resources provided to us. Following this, the brainstorming and general discussion process began, which includes reviewing potential areas to start with such as all our needs and limitations, what equipment will be needed, and road mapping all the group’s meeting sessions. Our brainstorming included drafting up a couple of different rough system block diagrams as we had more than one initial idea to approach the problem.

We were quick to divide up complex components of the project from reviewing the provided lecture information. We noted that the implementation of the sensor reception would be a singular task, as well as the scoring up/down the system, and the music notes system. Following this approach, we grouped up sections and spent time together discussing how each of them should be ordered and completed. We also noted the completed alarm clock project (lab 2) and the completed capacitance project (lab 3) which were both strong starting points for components of the project.

Need and Constraint Identification

Determining the needs and requirements of the project were completed before developing the game and code, yet they were addressed in the development process.

In making a game, the experience must not be repetitive. This was a need our group quickly identified that indicated that there had to be variations to the victory process and that the sound trigger scoring process must be random and unpredictable at every turn. We knew this could potentially be implemented using a sort of randomization feature of the alarm sounding and wait times. We also noted that the game had to be functional and play consistently as a basic need, which would require optimising the sensors and making the start/end process of the game smooth.

We identified that the game source code had to be neat and scalable, which meant it had to be easily adaptable for more than the present two-player count. In addition, it had to be neat and readable with ample comments and strong naming conventions to ensure any reader could understand and replicate components of the project.

Problem Specification

When developing the game, it was important to make the experience enjoyable for all users and to avoid repetitiveness. Since the basis of the game revolves around hitting a foil sensor at certain times to score points, we made sure to make the sensors as responsive as possible without glitching out and incorrectly triggering. We spent time to make sure players could score with reasonable pressing effort, players couldn’t double trigger the sensor, and the player who scored first was always accurately awarded the point. This ensured a functional and fair game for anyone who played. We also knew that the game had to have twists and the sound trigger scoring process had to be unique. To do so, the RNG system was implemented to make the sounds of the game irregular, keeping each player on edge.

We aimed to ensure that players could strive for something better than just winning. If a player scored five points while their opponent had zero, they won with a special outro victory song, which made the game more exciting and interested players in playing for longer to win in that manner.

A general need was that the game needed to be multiplayer, which required the code to be organised to have a working sensor timer for each active player. Additionally, we made sure the code branches and associated variables were easily named so that the game code could be easily scaled up for 3 or more players by simply duplicating sections and following the presently used naming convention. It was noted that the only limitation to the player count using our source code style was the number of LM555 chips and breadboard space available for them to be connected. In addition, displaying information for more than two players quickly gets chaotic on the two-row LCD.

As the code neared completion and revision, we began to organise the branches, rename unclear variables, and add comments to code blocks that were difficult to understand at face value.

Solution Generation

When determining the approach to solve the project, we had to choose between using Lab 2 or Lab 3 as our starting point. We initially thought to use the working clock to run the scoring and wait for the multiplayer gameplay, however, we quickly found that this skipped and overlooked fundamental aspects that would be difficult to include later such as sounds and scoring decrements. Instead, we chose to use the capacitance sensor lab code to develop the project groundwork. We considered that the capacitance sensor code could be duplicated to account for two or more players. Once we could read any form of input from both players, we could look to add thresholds and scoring systems.

In designing a solution, a great amount of time had to be allocated to the breadboard, and again we looked to use the lab 3 breadboard layout as a template, with the capacitance sensor circuit block having to be duplicated and connected respectively for both players. All the discussions held between the members of our group aimed to map out how to route our project through each step, making sure to take advantage of all prior work, and brainstorm at least two design solution paths, one of which ended up working.

Solution Evaluation

To meet project criteria, the coding process had to be understandable from a reader's perspective. Aside from good naming conventions and ample comments, the sequence of the code had to make sense. Since programming in Assembly 8051 is a sequential branching process, the order in which we designed everything needed to make sense. For instance, the whole sound system of the project began to run the moment the game started and stopped when the game ended, indicating that it would be best understood if initiated immediately after the code branches that began the game was run **(Appendix 1.m)**. It could have been written later or in a different place, but it would confuse both writers and readers of the code. This example applied to many sections as we followed a logical sequence when running through our concept design. This helped us greatly during the debugging process and when we scaled the game for two players.

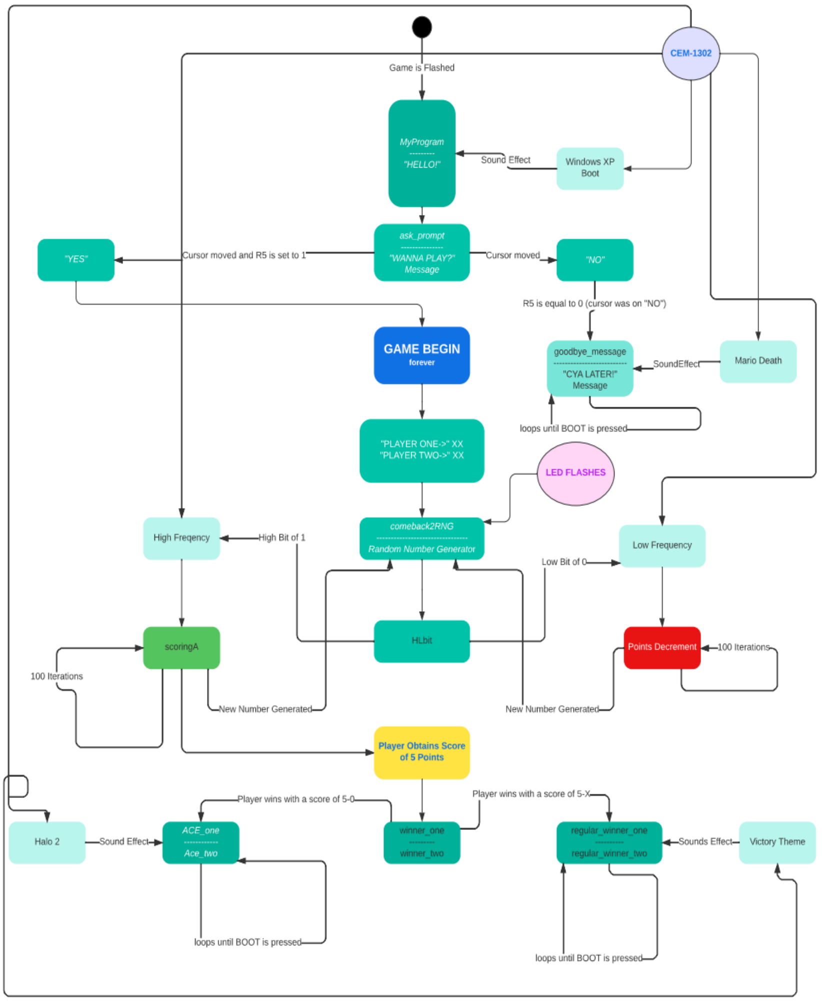
We chose our final circuit design as it was very neat and quite scalable. We were able to adjust the position of the LCD and use flattened wire connections in a manner that made it easy to add a second 555 timer with extra space for even 3 or 4 players on the board. As we decided to base our work on lab 3, our code design choices were based on that too, since we began with making the sensors read data before bothering with scoring and synchronising the sounds with the player inputs.

Safety/Professionalism

When building our circuit, we made sure to sketch out the breadboard layout to ensure all components are connected and to ensure there are no circuit dangers or risks of damaging components. When building, we made sure to use wires that are properly cut and undamaged, as well as chip/transistor components that have unbent pins. We also made sure to keep a clean workspace with no food or drink and clean, dry hands when working with any components.

Detailed Design

The following picture shows a Software Block Diagram that exhibits the overall design of our project (a PDF of the diagram is also attached if the image is unclear):



We had hypothesized using an even number generated to emit a low frequency and an odd number to emit a high frequency given the sample code **(Appendix 1.a)**. We wanted to see what HLbit was using the LCD. We accomplished this **(Appendix 1.b)** and noticed the HLbit referred to a high or low bit since we observed a fluctuation between 1's and 0's on the LCD. We then iterated back to our design and realized that the high frequency being played would depend on HLbit being 1 and the low frequency would depend on HLbit being 0 **(Appendix 1.c)**.

From here, we needed to extrapolate a way for the code to check for points incrementing if it went to a high frequency or points decrementing if it went to a low frequency. We made the branches emit the high frequency of 2,100 Hz and the low frequency of 2,000 Hz where the code would jump to check if a sensor was being hit **(Appendix 1.d)**. If the HLbit was 1, we would go to the branch "Freq_High" where the CEM-1302 would emit 2,100 Hz and then jump to a branch called "scoringA" where the code would check for incrementing points. Inside "scoringA," we added code that would cause 100 loops throughout this branch to check for a point incrementing or not **(Appendix 1.e)**. This would be independent of the "Wait_Random" function that freezes the code for a random amount of time. We needed this feature so that a player would not have to hit their sensor immediately when the frequency changes. Due to the 100 loops, the code would check if they hit their sensor through 100 iterations.

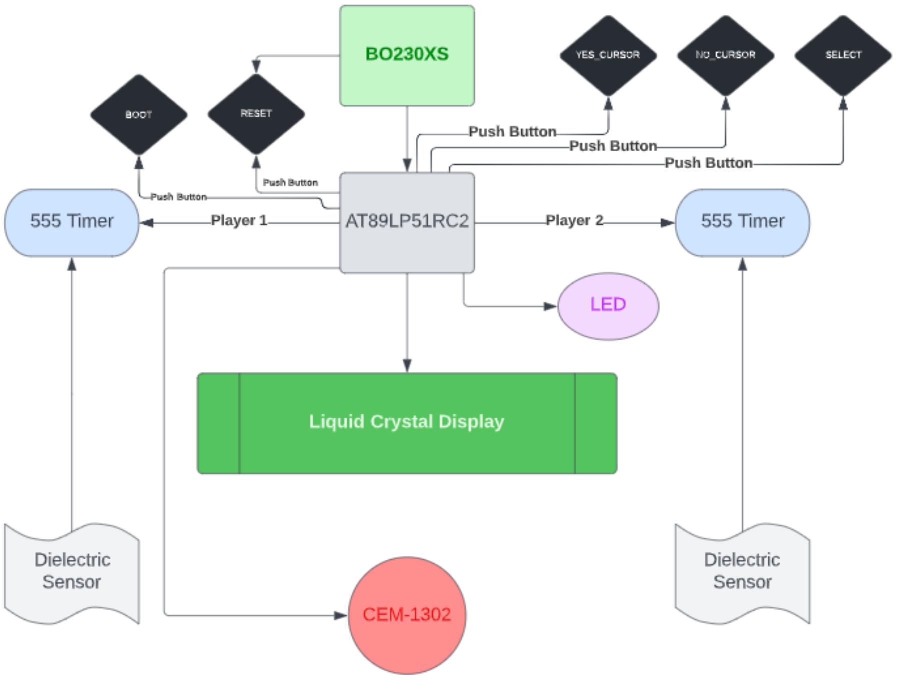
Though the dielectric material used to make the sensors functions as a capacitor, our team made the executive decision to calculate the period **(Appendix 1.f)** instead since there is a smaller margin of error when doing so. We then tested to see the base period when no one was touching the sensors by using the LCD which we learned to be a value fluctuating around 200,000 units. We then made the threshold of 210,000 units so if a sensor exceeded this value, the code would jump to a branch to increment a point **(Appendix 1.g)**.

We then made a branch that would add a point to a player’s score **(Appendix 1.h)** where we used several registers as certain “flags.” All these registers were initialized as #0x00 in MyProgram. The code being alluded to for this explanation refers to how the points were incremented for Player 1 (label for one of the players). The register R4 functioned as a flag that would check if Player 2 (label for the other player) had hit their sensor first. If so, R4 would hold the value of 1 meaning that the code responsible to add a point for Player 1 would be skipped. R0 was used as a flag to ensure that if Player 1 had hit their sensor during a high frequency, the points would not jump from 0 directly to 5. This was done by setting R0 to 1 when a point was added so that when another iteration was made if R0 equalled 1, the code responsible to increment a point for Player 1 would be skipped. If the sensor is not pressed or as the hand is lifted, 0 is moved back into R0 to reset the flag so that a point could be incremented when another high frequency is emitted. Similar to R4, R3 is the flag that checks if Player 1 has come first. R3 is given the value of 1 when Player 1 has scored a point. R3 and R4 essentially ensure that if a player is second to hit the sensor, their score would not be changed. In the code, we also exhibit how the scoring system has a maximum of 5 points using the techniques we learned from labs before this project.

The same method of incrementing a point for Player 1 is done for Player 2 using different registers as flags along with branches named similarly but still being distinct to avoid errors. The code used to calculate and obtain the period remains universal for both incrementing and decrementing branches. Now, if the RNG had caused HLbit to be 0, a low frequency would have been emitted. From “Freq_Low,” the code would have jumped to “scoringB.” This branch is very similar to “scoringA” except here we check for decrementing points. The code being alluded to for this explanation refers to how the points were decremented for the Player. A branch is used to subtract a point from Player 1’s score and ensure the score does not drop below 00 **(Appendix 1.i)**. The same logic and code are applied for Player 2.

Our bonus features required musical notes to be translated into frequencies for the CEM-1302 to emit **(Appendix 1.j)**. We included a message reading “HELLO!” **(Appendix 1.k)** while the Windows XP Boot **(Appendix 1.l)** sound effect played. The sheet music **(Appendix 2.a)** was translated into notes. After this, the prompt “WANNA PLAY?” was asked along with “YES” and “NO” as options **(Appendix 1.m)**. Push buttons were used to toggle between the options and select them **(Appendix 1.n)**. If “NO” was selected, the code would skip to a goodbye message **(Appendix 1.o)** along with the Mario Death sound effect **(Appendix 1.p)** being played translated from sheet music **(Appendix 2.b)**. This was accomplished using R5 which functioned as a flag. If the button NO_CURSOR was pressed in “cursor_no,” R5 was given the value of 0. If SELECT was pressed when R5 was 0, then the code would just to “goodbye_message.” Anytime YES_CURSOR was pressed in “cursor_yes,” R5 was given the value of 1. This meant that in “choose_select” if R5 was equal to 1, then the code would skip to “forever” where the game would begin. For the next explanation regarding if a player won, a branch called “winner_one” will be referred to that specifically is jumped to if Player 1 reaches 5 points **(Appendix 1.q)**. Here “testb” which holds the score of Player 2 is checked to see if it equals zero. If so, this means that Player 1 has achieved a score of 5-0 which our team calls an “Ace” and the code skips to a branch called “ACE_one.” Here the message “ACE!” is displayed on the respective player’s line **(Appendix 1.r)** along with the Halo 2 theme song **(Appendix 1.s)** is played which was translated from sheet music **(Appendix 2.d)**. If “testb” is not zero, then the code jumps to a branch called “regular_win_one” where the message “WINNER” is displayed **(Appendix 1.t)** with the Victory theme music **(Appendix 1.u)** collected from YouTube **(Appendix 2.c)**.

The following picture shows a Hardware Block Diagram that exhibits the overall design of our project (a PDF of the diagram is also attached if the image is unclear):



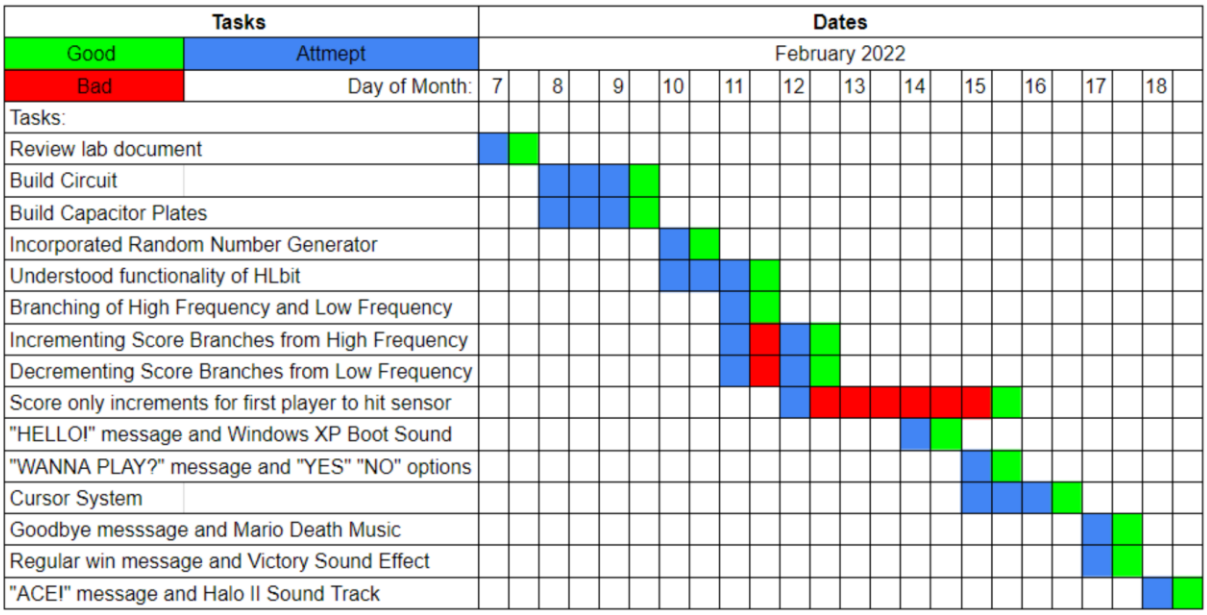
The information collected for the hardware explanation comes from the ATMEL User Manual **[1]** or the slides from our ELEC 291 lectures that derived schematics from it. Our team used interrupts which are events that trigger automatic execution of predetermined code. Our code contained Interrupt Services Routines (ISR) which disrupted the normal flow of code. This can be triggered by various events, but we specifically relied on timers overflowing **(Appendix 3.a)**. We analysed logic diagrams behind the functionality of Timer 0 and 1 **(Appendix 3.b)**. A schematic of the interface between the 8051 microcontroller and BO230XS **(Appendix 3.c)** was used and built upon to create the hardware for our project.

Information regarding 555 timers comes from the LM555 Timer Manual **[2]** or the slides from our ELEC 291 lectures that derived schematics from it. Two 555 timers **(Appendix 3.d)** were used. They were set up in a way where they would trigger themselves and free run as multivibrators **(Appendix 3.e)**. The external capacitor charges via *Register A + Register B* and discharges through *Register B* between ⅓ VCC and ⅔ VCC. When triggered, the frequency is independent of the supply voltage. This is how the periods were obtained from the dielectric sensors.

The 8051 microcontroller is attached to these two 555 timers along with a CEM-1302 speaker that was configured to emit loud frequencies **(Appendix 3.f)**. Push buttons were also attached for the YES_CURSOR, NO_CURSOR, and SELECT variables declared in the code. These helped move the cursor and select the options "YES" or "NO." An LED was also attached to the microcontroller and flashed every time a new random number was generated. Finally, the LCD was also attached to the microcontroller to display various texts **(Appendix 3.g)**.

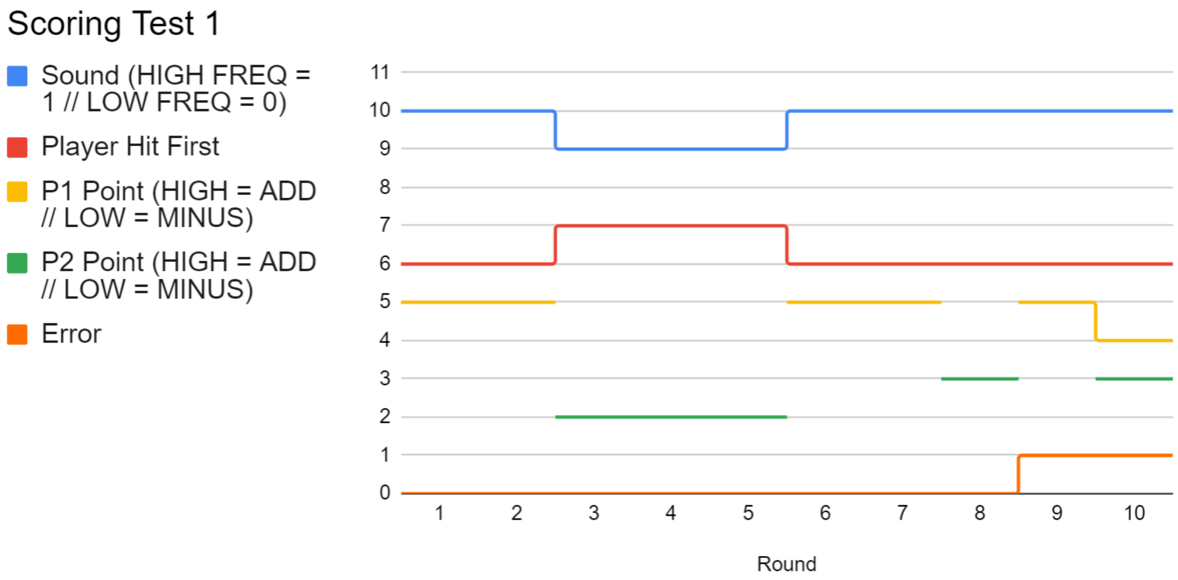
Solution Assessment

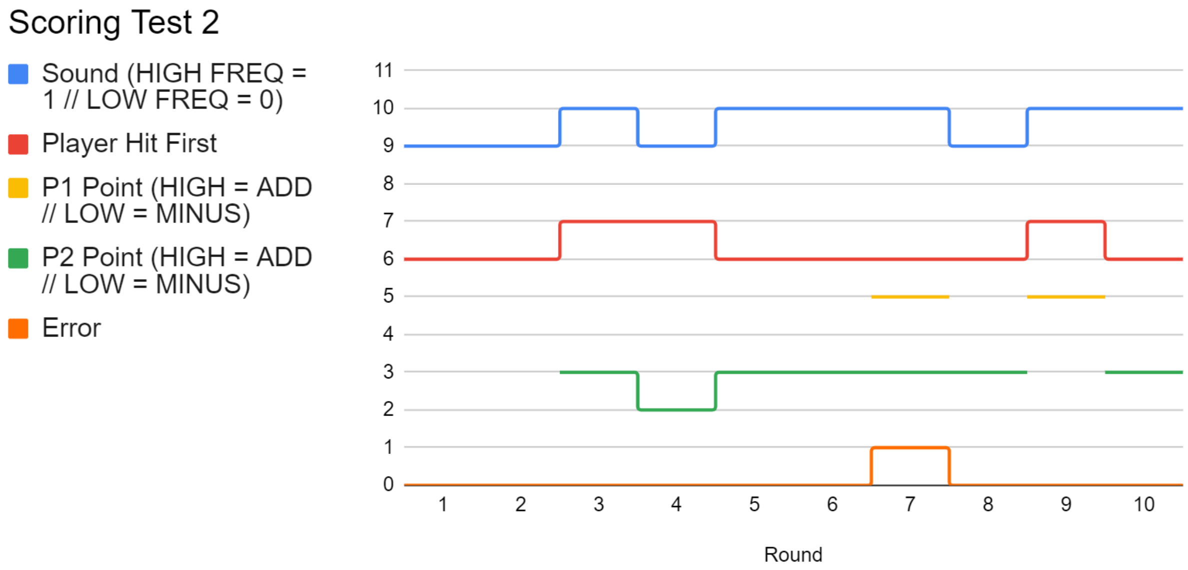
This Gantt Chart displays our intuitive approach on how we assessed the design performance based on requirements, needs, and constraints.



Tasks		Dates																							
🟩 Good	🟦 Attmept	February 2022																							
🟥 Bad	Day of Month:	7		8		9		10		11		12		13		14		15		16		17		18	
Tasks:																									
Review lab document		🟦	🟩																						
Build Circuit				🟦	🟦	🟦	🟩																		
Build Capacitor Plates				🟦	🟦	🟦	🟩																		
Incorporated Random Number Generator								🟦	🟩																
Understood functionality of HLbit								🟦	🟦	🟦	🟩														
Branching of High Frequency and Low Frequency										🟦	🟩														
Incrementing Score Branches from High Frequency										🟦	🟥	🟦	🟩												
Decrementing Score Branches from Low Frequency										🟦	🟥	🟦	🟩												
Score only increments for first player to hit sensor												🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟩						
"HELLO!" message and Windows XP Boot Sound																🟦	🟩								
"WANNA PLAY?" message and "YES" "NO" options																		🟦	🟩						
Cursor System																		🟦	🟦	🟦	🟩				
Goodbye messsage and Mario Death Music																						🟦	🟩		
Regular win message and Victory Sound Effect																						🟦	🟩		
"ACE!" message and Halo II Sound Track																								🟦	🟩

Below are two examples of data that we collected while testing our scoring system with two players and various sensor hit timings:





Live-Long Learning

In creating musical sequences for various states of the game (i.e. game start, game-winner), an understanding of musical notes and frequencies had to be made. For instance, to implement the Halo 2 Theme Song (special ace winner sound), sheet music had to be read and notes had to be compared with a note/frequency chart. Different octaves could be reached by doubling or halving the frequency and accidentals (sharps and flats) could be included by varying the frequencies appropriately. This technique was applied to additional sounds such as the Windows XP Boot sound (starting device), Victory Sound, and Mario Game Death Sound (game cancelled).

Interrupts were a common feature that was included to complete certain features such as the capacitance timers running for each of the two players. To understand them, the Atmel AT89LP51RC2 datasheet documentation had to be reviewed. Sections that included pin instructions and logic diagrams had to be understood to ensure that interrupts were called appropriately and functioned as intended.

Conclusions

Our project was a functional two-player sensor game that awarded points to players for triggering their sensor at the high beep of a sounding speaker. Upon flashing and connecting the game breadboard, an introductory “HELLO!” message appears, and a bonus Windows XP boot sound is played, followed by a prompt “WANNA PLAY?” A player is allowed to move the cursor left or right to select “YES” or “NO” and select their choice using the three available board buttons. If a player selects “NO”, a bonus Mario Death music is played and the message “CYA LATER!” is displayed, whereas if a player selects “YES”, the game begins. A player winning a regular match is greeted with a “WINNER” label on their line and a victory sound effect. A player winning 5-0 is greeted with “ACE!” and the special Halo 2 theme music is played instead. A red LED is frequently blinking at the rate of new random numbers being generated, which are used in randomising the game sounds. In addition, the code includes additional branches to ensure only one press by a player is recognized at a time. Translated musical notes as frequency commands are included, which provide the groundwork to play any desired song by knowing the notes and rhythm.

The project teamwork spanned over two weeks, taking frequent 4-5 hour sections at each phase of the process, totalling over 25 hours of work. Some sections of the project were completed physically together such as components that required multiple perspectives and testing. Others were debugged and tested independently as some code blocks were individually designed and best understood by a specific member.

References

[1] "8-bit flash microcontroller with 24K/32K bytes program ..." [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/doc3722.pdf. [Accessed: 04-Mar-2022].

[2] "LM555 timer datasheet (rev. D) - texas instruments." [Online]. Available: https://www.ti.com/lit/ds/symlink/lm555.pdf. [Accessed: 04-Mar-2022].

Bibliography

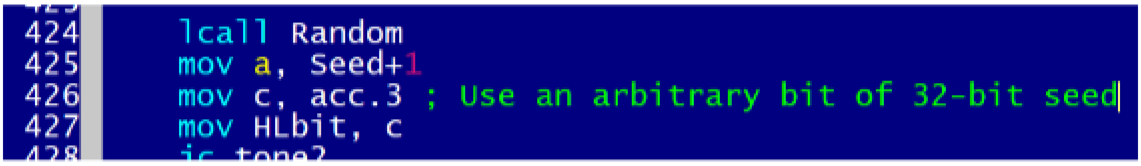
Lucid visual collaboration suite: Log in. [Online]. Available: https://lucid.app/. [Accessed: 04-Mar-2022].

"IEEE Citation Generator," *Citation Machine, a Chegg service*. [Online]. Available: https://www.citationmachine.net/ieee. [Accessed: 04-Mar-2022].

ARM Ltd and ARM Germany GmbH, " CJNE," *8051 Instruction Set Manual: CJNE*. [Online]. Available: https://www.keil.com/support/man/docs/is51/is51_cjne.htm#:~:text=The%20CJNE%20instruction%20compares%20the,continues%20with%20the%20next%20instruction. [Accessed: 04-Mar-2022].

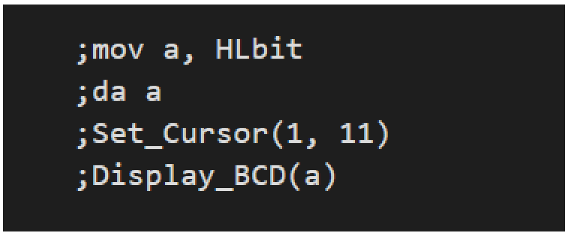
Appendix 1: Relevant Code Blocks

a)



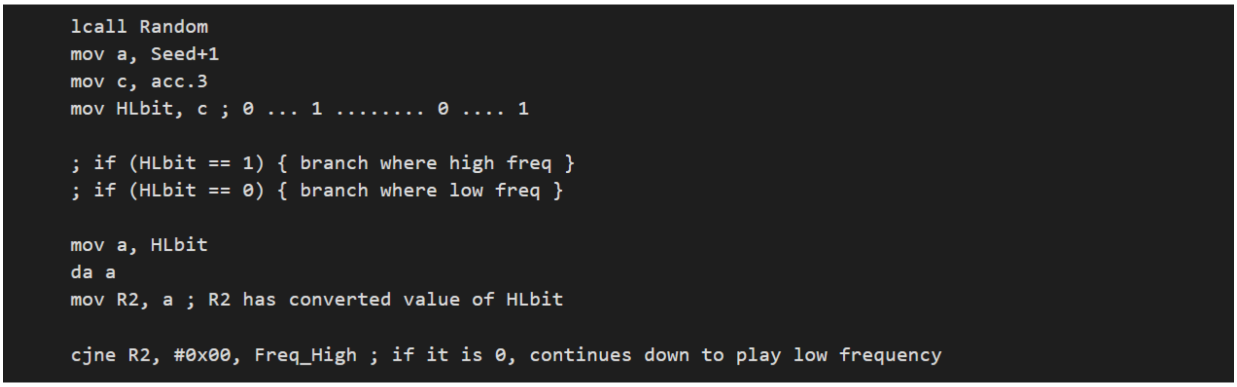
```
    lcall Random
    mov a, Seed+1
    mov c, acc.3 ; Use an arbitrary bit of 32-bit seed
    mov HLbit, c
    jc tone2
```

b)



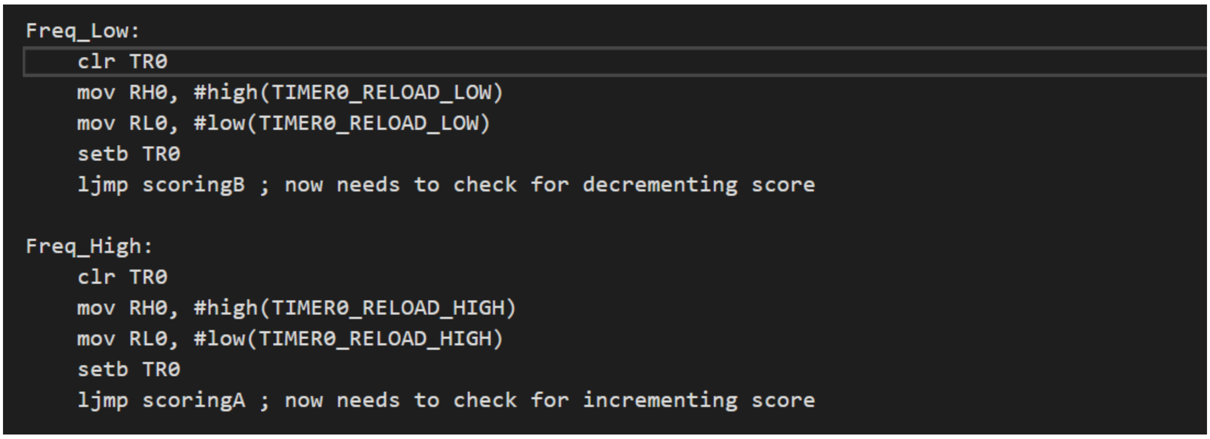
```
    ;mov a, HLbit
    ;da a
    ;Set_Cursor(1, 11)
    ;Display_BCD(a)
```

c)



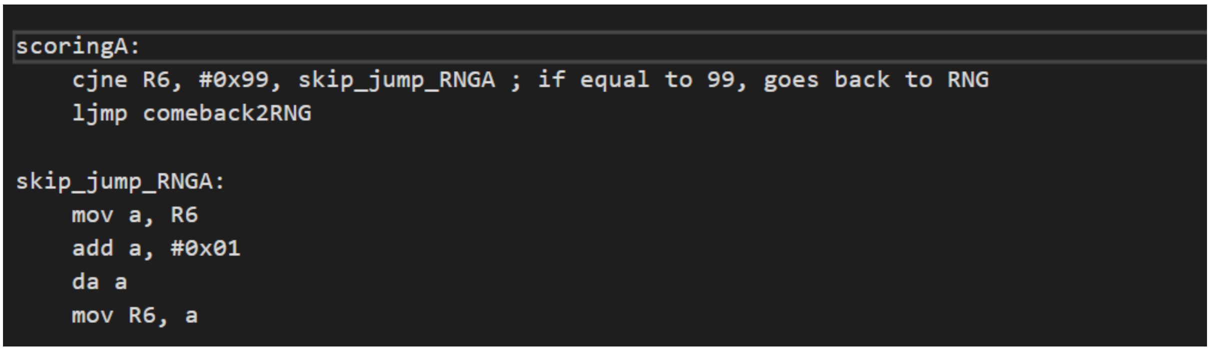
```
    lcall Random
    mov a, Seed+1
    mov c, acc.3
    mov HLbit, c ; 0 ... 1 ........ 0 .... 1

    ; if (HLbit == 1) { branch where high freq }
    ; if (HLbit == 0) { branch where low freq }

    mov a, HLbit
    da a
    mov R2, a ; R2 has converted value of HLbit

    cjne R2, #0x00, Freq_High ; if it is 0, continues down to play low frequency
```

d)



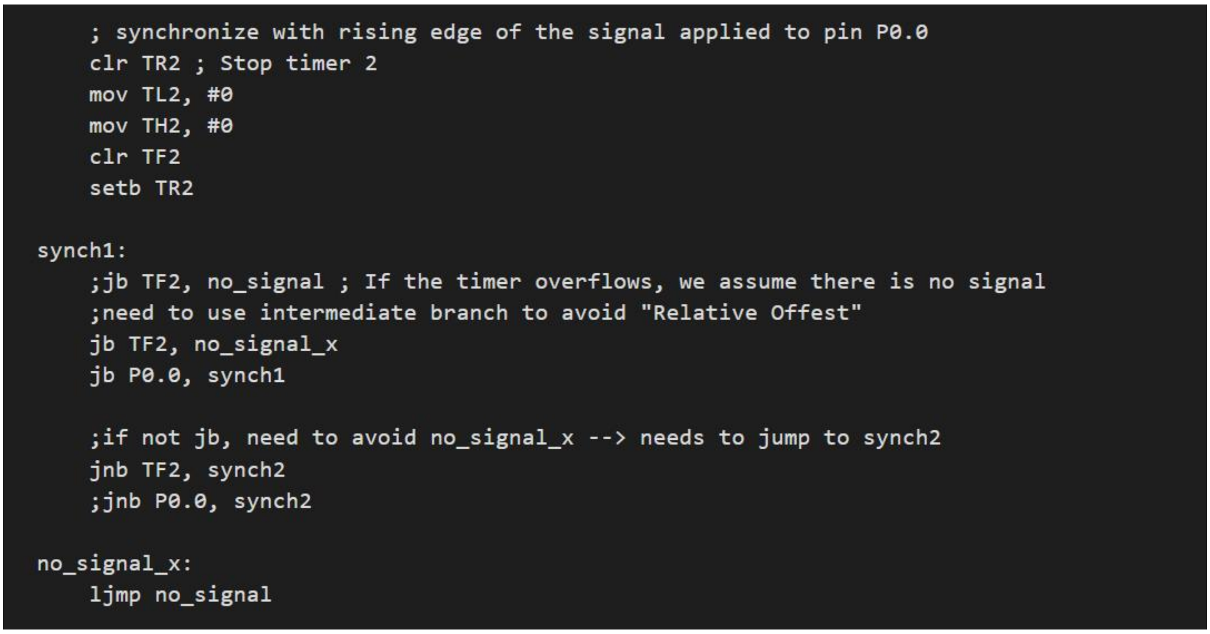
```
Freq_Low:
    clr TR0
    mov RH0, #high(TIMER0_RELOAD_LOW)
    mov RL0, #low(TIMER0_RELOAD_LOW)
    setb TR0
    ljmp scoringB ; now needs to check for decrementing score

Freq_High:
    clr TR0
    mov RH0, #high(TIMER0_RELOAD_HIGH)
    mov RL0, #low(TIMER0_RELOAD_HIGH)
    setb TR0
    ljmp scoringA ; now needs to check for incrementing score
```

e)

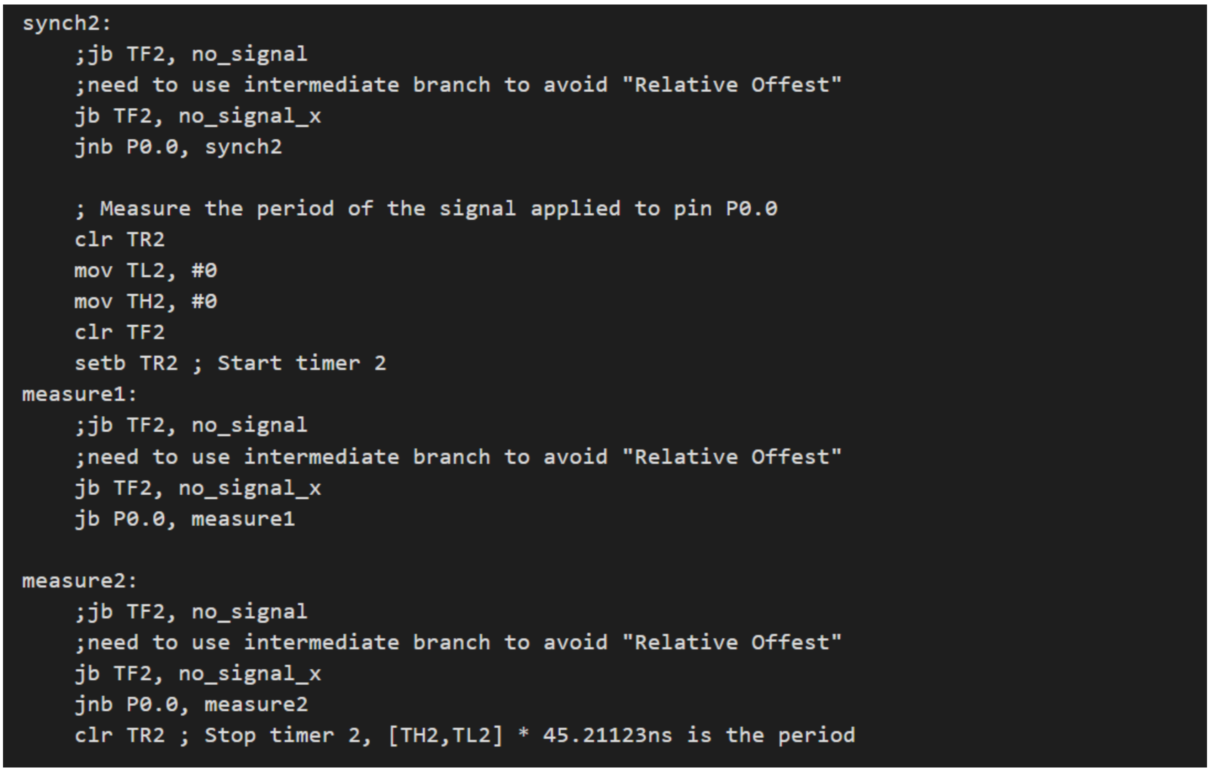


```
scoringA:
    cjne R6, #0x99, skip_jump_RNGA ; if equal to 99, goes back to RNG
    ljmp comeback2RNG

skip_jump_RNGA:
    mov a, R6
    add a, #0x01
    da a
    mov R6, a
```

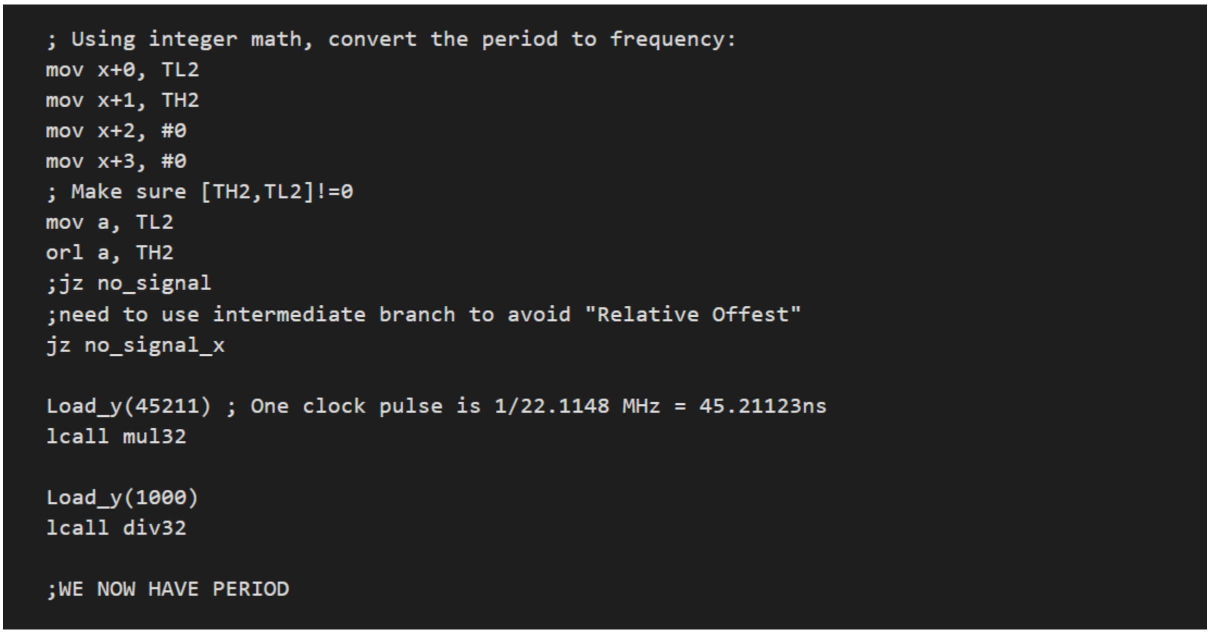
f)



```
    ; synchronize with rising edge of the signal applied to pin P0.0
    clr TR2 ; Stop timer 2
    mov TL2, #0
    mov TH2, #0
    clr TF2
    setb TR2

synch1:
    ;jb TF2, no_signal ; If the timer overflows, we assume there is no signal
    ;need to use intermediate branch to avoid "Relative Offest"
    jb TF2, no_signal_x
    jb P0.0, synch1

    ;if not jb, need to avoid no_signal_x --> needs to jump to synch2
    jnb TF2, synch2
    ;jnb P0.0, synch2

no_signal_x:
    ljmp no_signal
```

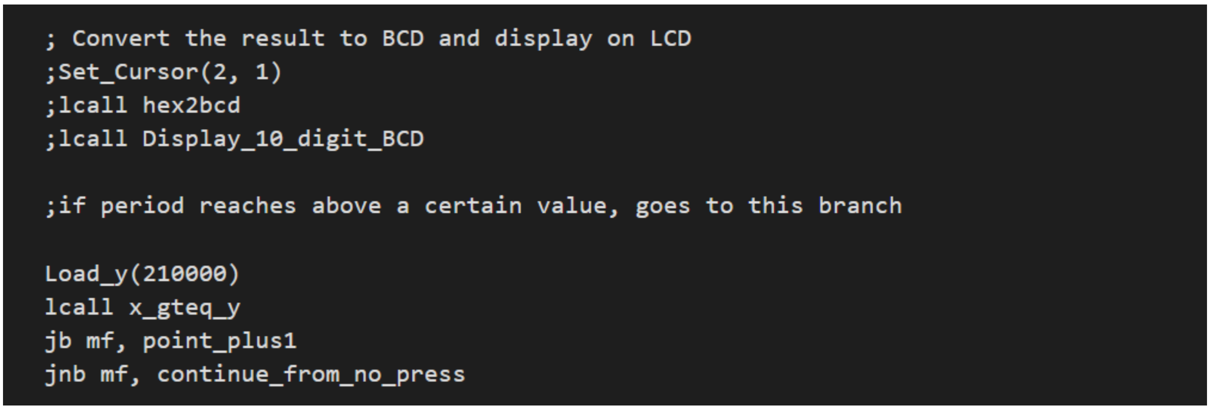


```
synch2:
    ;jb TF2, no_signal
    ;need to use intermediate branch to avoid "Relative Offest"
    jb TF2, no_signal_x
    jnb P0.0, synch2

    ; Measure the period of the signal applied to pin P0.0
    clr TR2
    mov TL2, #0
    mov TH2, #0
    clr TF2
    setb TR2 ; Start timer 2
measure1:
    ;jb TF2, no_signal
    ;need to use intermediate branch to avoid "Relative Offest"
    jb TF2, no_signal_x
    jb P0.0, measure1

measure2:
    ;jb TF2, no_signal
    ;need to use intermediate branch to avoid "Relative Offest"
    jb TF2, no_signal_x
    jnb P0.0, measure2
    clr TR2 ; Stop timer 2, [TH2,TL2] * 45.21123ns is the period
```



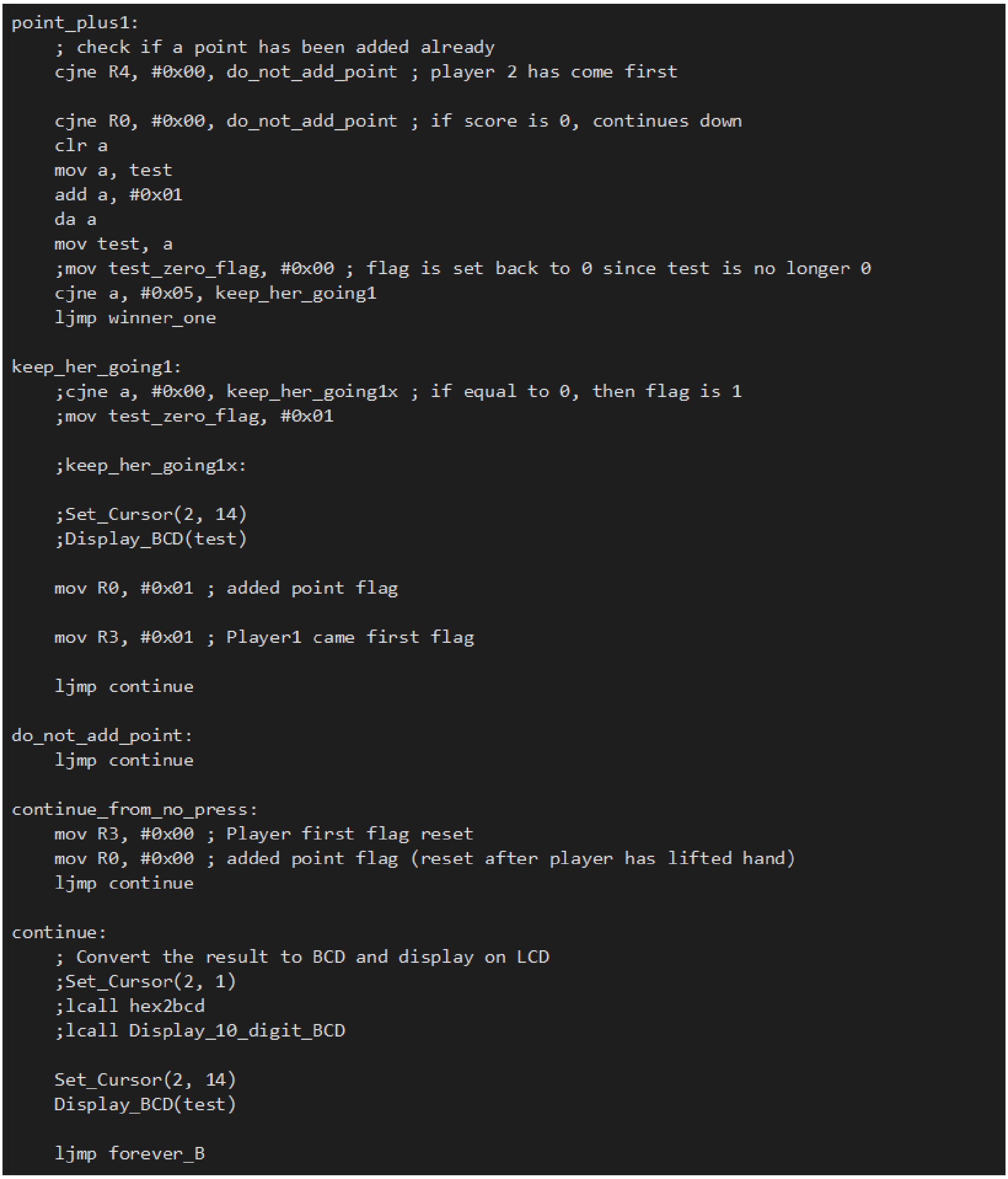
```
  ; Using integer math, convert the period to frequency:
  mov x+0, TL2
  mov x+1, TH2
  mov x+2, #0
  mov x+3, #0
  ; Make sure [TH2,TL2]!=0
  mov a, TL2
  orl a, TH2
  ;jz no_signal
  ;need to use intermediate branch to avoid "Relative Offest"
  jz no_signal_x

  Load_y(45211) ; One clock pulse is 1/22.1148 MHz = 45.21123ns
  lcall mul32

  Load_y(1000)
  lcall div32

  ;WE NOW HAVE PERIOD
```

g)



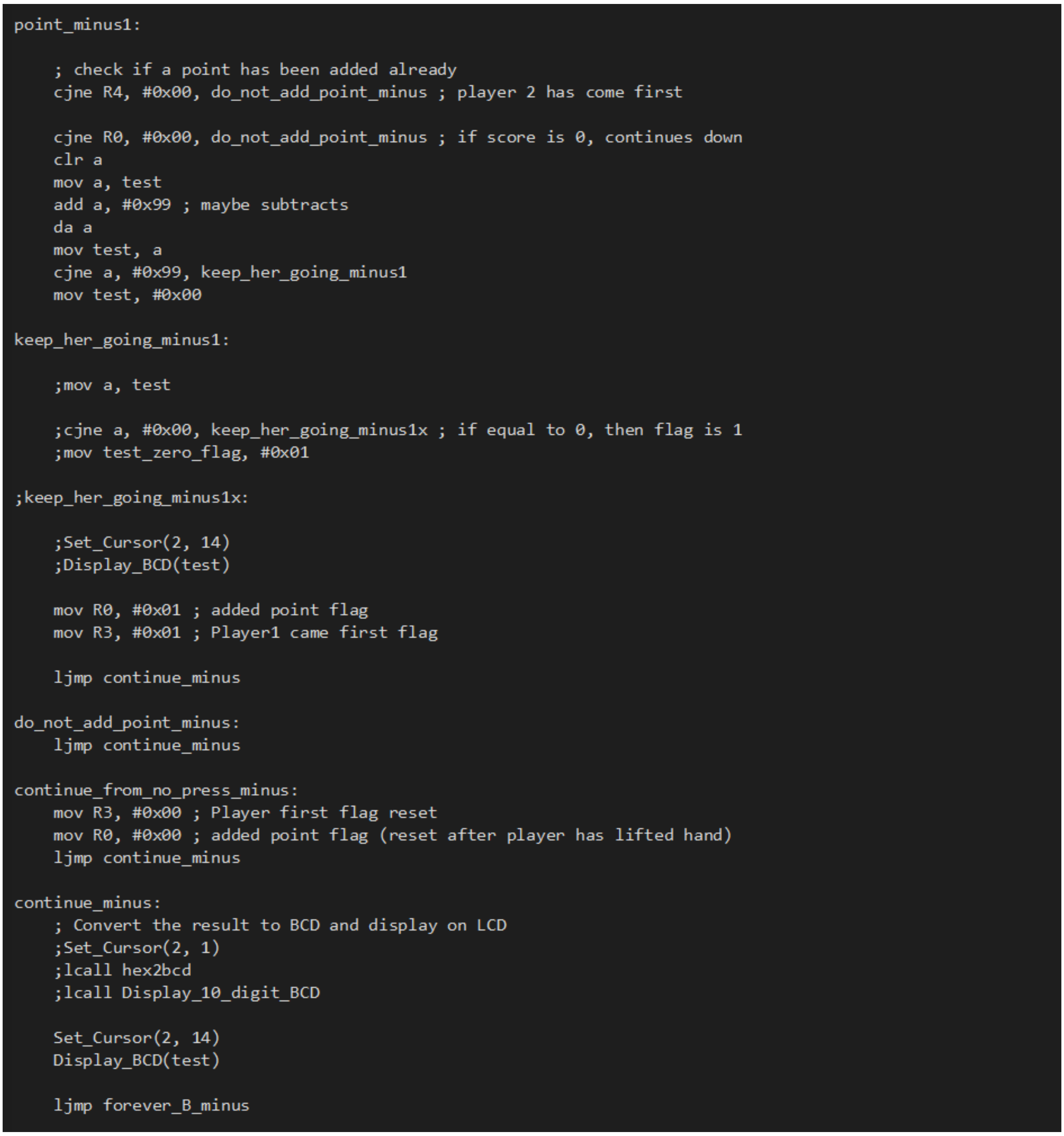
```
  ; Convert the result to BCD and display on LCD
  ;Set_Cursor(2, 1)
  ;lcall hex2bcd
  ;lcall Display_10_digit_BCD

  ;if period reaches above a certain value, goes to this branch

  Load_y(210000)
  lcall x_gteq_y
  jb mf, point_plus1
  jnb mf, continue_from_no_press
```

h)



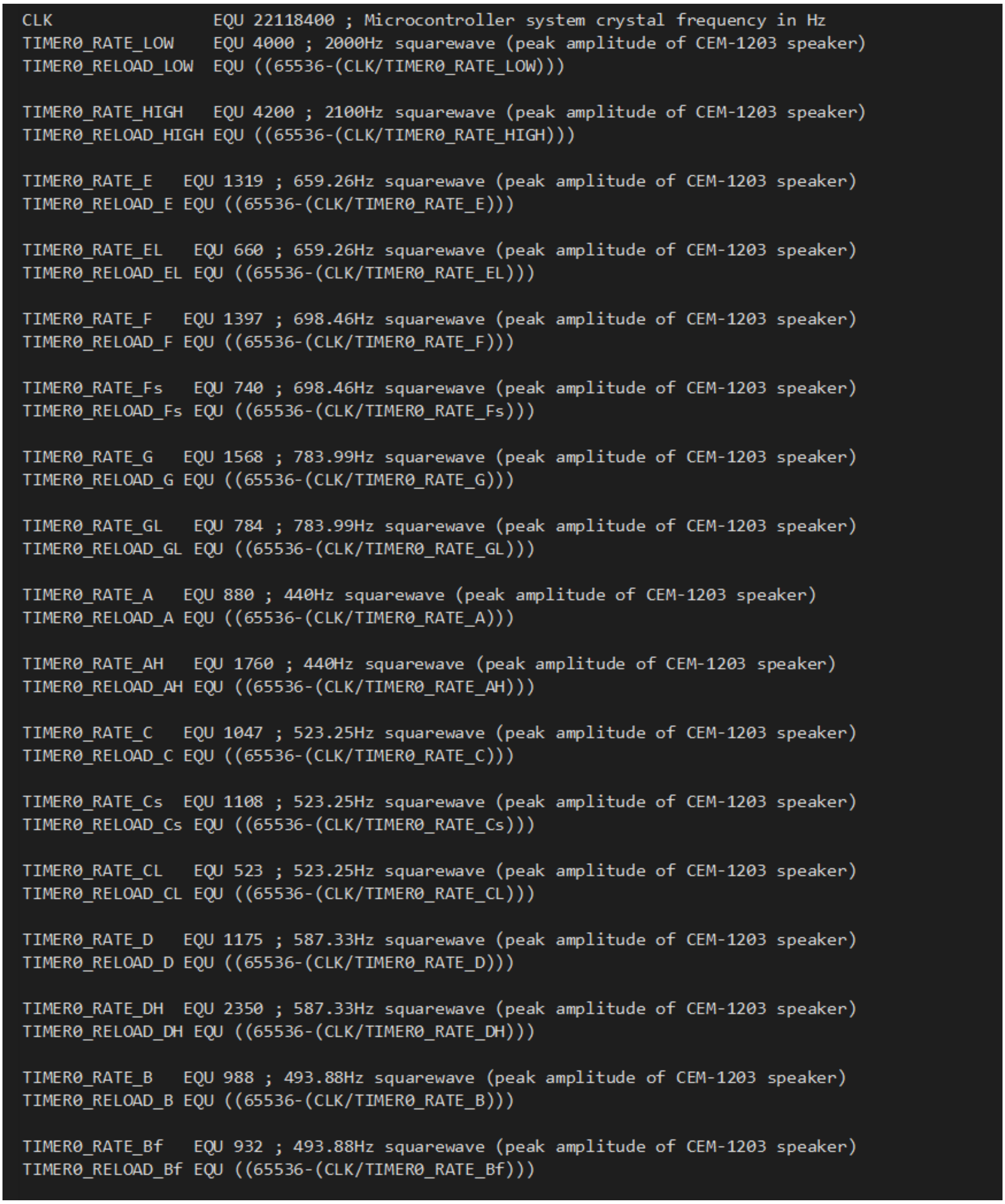
```
point_plus1:
    ; check if a point has been added already
    cjne R4, #0x00, do_not_add_point ; player 2 has come first

    cjne R0, #0x00, do_not_add_point ; if score is 0, continues down
    clr a
    mov a, test
    add a, #0x01
    da a
    mov test, a
    ;mov test_zero_flag, #0x00 ; flag is set back to 0 since test is no longer 0
    cjne a, #0x05, keep_her_going1
    ljmp winner_one

keep_her_going1:
    ;cjne a, #0x00, keep_her_going1x ; if equal to 0, then flag is 1
    ;mov test_zero_flag, #0x01

    ;keep_her_going1x:

    ;Set_Cursor(2, 14)
    ;Display_BCD(test)

    mov R0, #0x01 ; added point flag

    mov R3, #0x01 ; Player1 came first flag

    ljmp continue

do_not_add_point:
    ljmp continue

continue_from_no_press:
    mov R3, #0x00 ; Player first flag reset
    mov R0, #0x00 ; added point flag (reset after player has lifted hand)
    ljmp continue

continue:
    ; Convert the result to BCD and display on LCD
    ;Set_Cursor(2, 1)
    ;lcall hex2bcd
    ;lcall Display_10_digit_BCD

    Set_Cursor(2, 14)
    Display_BCD(test)

    ljmp forever_B
```

i)



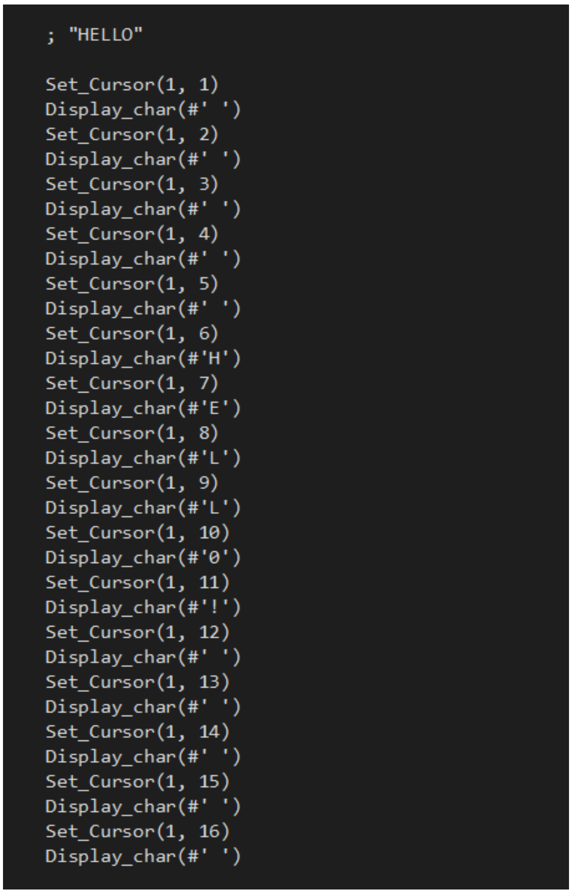
```
point_minus1:

    ; check if a point has been added already
    cjne R4, #0x00, do_not_add_point_minus ; player 2 has come first

    cjne R0, #0x00, do_not_add_point_minus ; if score is 0, continues down
    clr a
    mov a, test
    add a, #0x99 ; maybe subtracts
    da a
    mov test, a
    cjne a, #0x99, keep_her_going_minus1
    mov test, #0x00

keep_her_going_minus1:

    ;mov a, test

    ;cjne a, #0x00, keep_her_going_minus1x ; if equal to 0, then flag is 1
    ;mov test_zero_flag, #0x01

;keep_her_going_minus1x:

    ;Set_Cursor(2, 14)
    ;Display_BCD(test)

    mov R0, #0x01 ; added point flag
    mov R3, #0x01 ; Player1 came first flag

    ljmp continue_minus

do_not_add_point_minus:
    ljmp continue_minus

continue_from_no_press_minus:
    mov R3, #0x00 ; Player first flag reset
    mov R0, #0x00 ; added point flag (reset after player has lifted hand)
    ljmp continue_minus

continue_minus:
    ; Convert the result to BCD and display on LCD
    ;Set_Cursor(2, 1)
    ;lcall hex2bcd
    ;lcall Display_10_digit_BCD

    Set_Cursor(2, 14)
    Display_BCD(test)

    ljmp forever_B_minus
```

j)



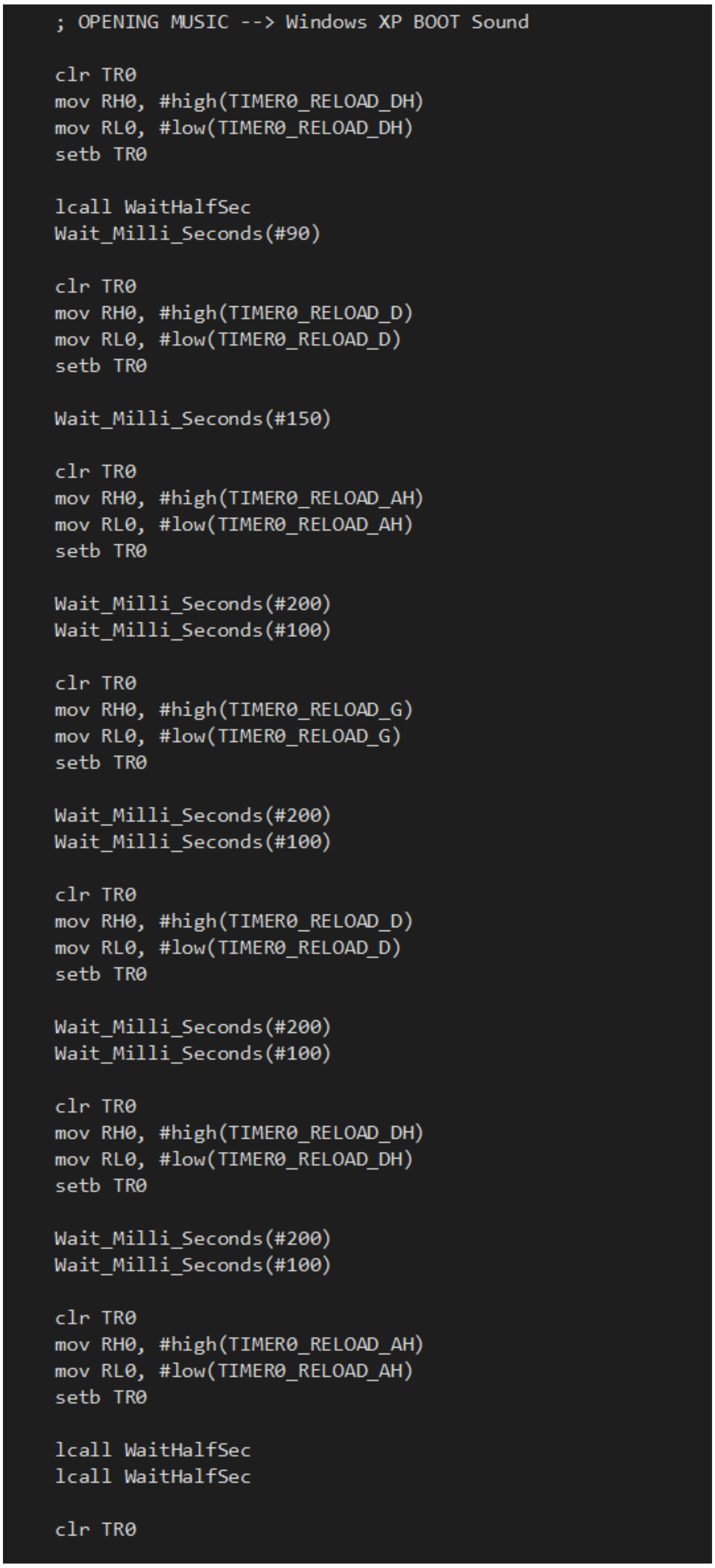
```
CLK                EQU 22118400 ; Microcontroller system crystal frequency in Hz
TIMER0_RATE_LOW    EQU 4000 ; 2000Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_LOW  EQU ((65536-(CLK/TIMER0_RATE_LOW)))

TIMER0_RATE_HIGH   EQU 4200 ; 2100Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_HIGH EQU ((65536-(CLK/TIMER0_RATE_HIGH)))

TIMER0_RATE_E   EQU 1319 ; 659.26Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_E EQU ((65536-(CLK/TIMER0_RATE_E)))

TIMER0_RATE_EL   EQU 660 ; 659.26Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_EL EQU ((65536-(CLK/TIMER0_RATE_EL)))

TIMER0_RATE_F   EQU 1397 ; 698.46Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_F EQU ((65536-(CLK/TIMER0_RATE_F)))

TIMER0_RATE_Fs   EQU 740 ; 698.46Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_Fs EQU ((65536-(CLK/TIMER0_RATE_Fs)))

TIMER0_RATE_G   EQU 1568 ; 783.99Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_G EQU ((65536-(CLK/TIMER0_RATE_G)))

TIMER0_RATE_GL   EQU 784 ; 783.99Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_GL EQU ((65536-(CLK/TIMER0_RATE_GL)))

TIMER0_RATE_A   EQU 880 ; 440Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_A EQU ((65536-(CLK/TIMER0_RATE_A)))

TIMER0_RATE_AH   EQU 1760 ; 440Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_AH EQU ((65536-(CLK/TIMER0_RATE_AH)))

TIMER0_RATE_C   EQU 1047 ; 523.25Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_C EQU ((65536-(CLK/TIMER0_RATE_C)))

TIMER0_RATE_Cs   EQU 1108 ; 523.25Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_Cs EQU ((65536-(CLK/TIMER0_RATE_Cs)))

TIMER0_RATE_CL   EQU 523 ; 523.25Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_CL EQU ((65536-(CLK/TIMER0_RATE_CL)))

TIMER0_RATE_D   EQU 1175 ; 587.33Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_D EQU ((65536-(CLK/TIMER0_RATE_D)))

TIMER0_RATE_DH  EQU 2350 ; 587.33Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_DH EQU ((65536-(CLK/TIMER0_RATE_DH)))

TIMER0_RATE_B   EQU 988 ; 493.88Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_B EQU ((65536-(CLK/TIMER0_RATE_B)))

TIMER0_RATE_Bf   EQU 932 ; 493.88Hz squarewave (peak amplitude of CEM-1203 speaker)
TIMER0_RELOAD_Bf EQU ((65536-(CLK/TIMER0_RATE_Bf)))
```

k)



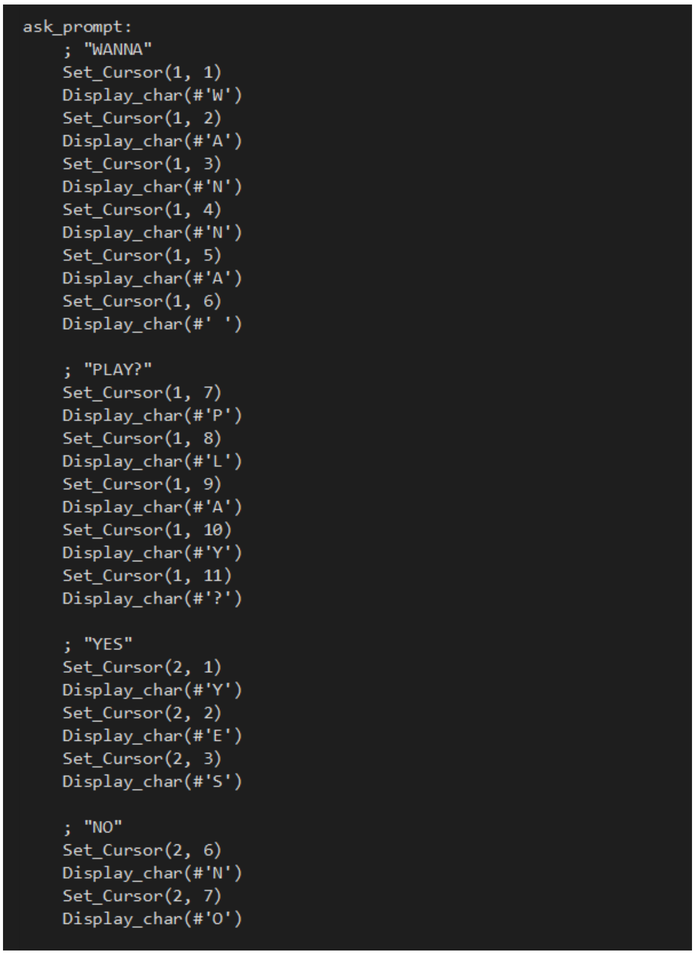
```
; "HELLO"

Set_Cursor(1, 1)
Display_char(#' ')
Set_Cursor(1, 2)
Display_char(#' ')
Set_Cursor(1, 3)
Display_char(#' ')
Set_Cursor(1, 4)
Display_char(#' ')
Set_Cursor(1, 5)
Display_char(#' ')
Set_Cursor(1, 6)
Display_char(#'H')
Set_Cursor(1, 7)
Display_char(#'E')
Set_Cursor(1, 8)
Display_char(#'L')
Set_Cursor(1, 9)
Display_char(#'L')
Set_Cursor(1, 10)
Display_char(#'0')
Set_Cursor(1, 11)
Display_char(#'!')
Set_Cursor(1, 12)
Display_char(#' ')
Set_Cursor(1, 13)
Display_char(#' ')
Set_Cursor(1, 14)
Display_char(#' ')
Set_Cursor(1, 15)
Display_char(#' ')
Set_Cursor(1, 16)
Display_char(#' ')
```

l)



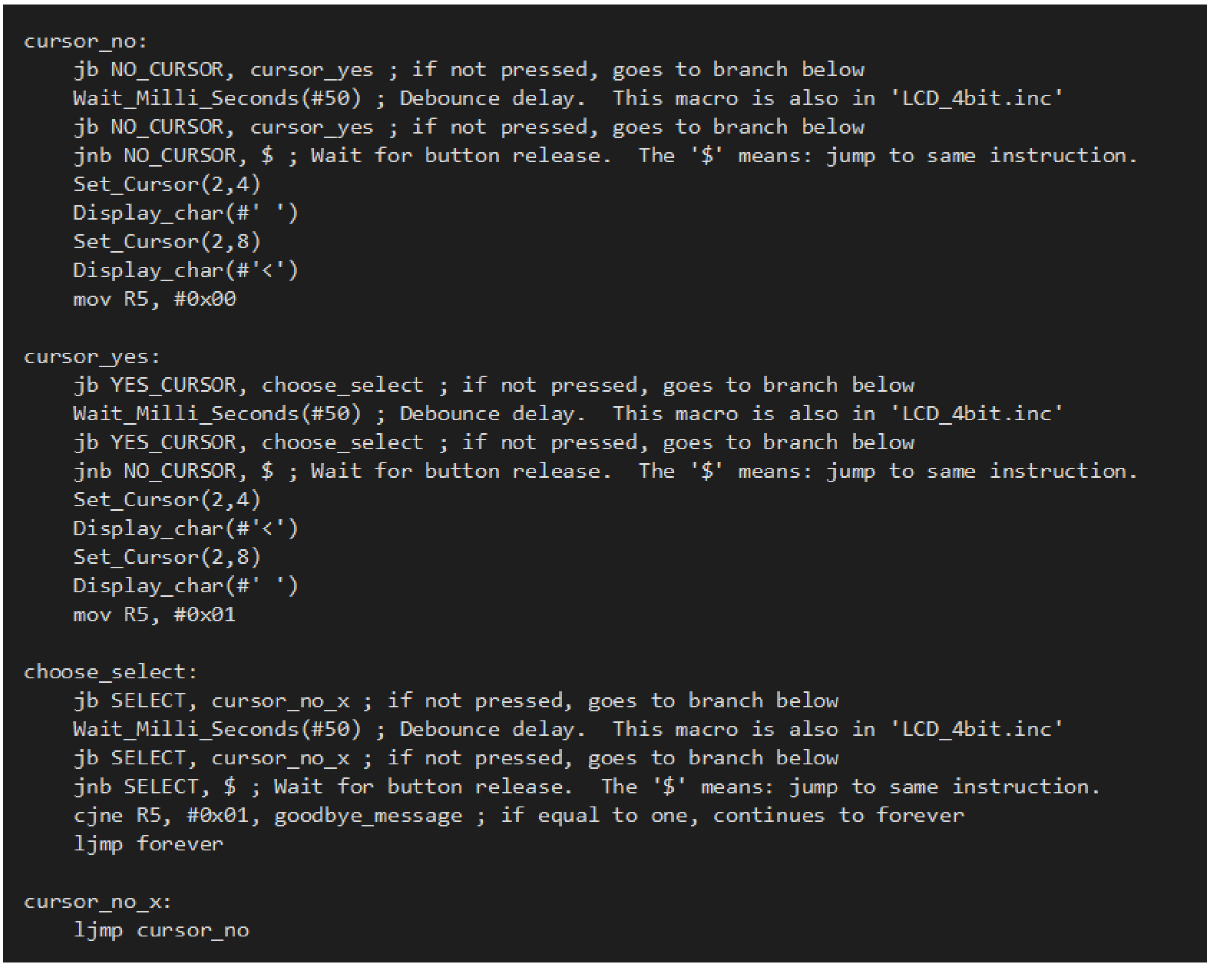
```
; OPENING MUSIC --> Windows XP BOOT Sound

clr TR0
mov RH0, #high(TIMER0_RELOAD_DH)
mov RL0, #low(TIMER0_RELOAD_DH)
setb TR0

lcall WaitHalfSec
Wait_Milli_Seconds(#90)

clr TR0
mov RH0, #high(TIMER0_RELOAD_D)
mov RL0, #low(TIMER0_RELOAD_D)
setb TR0

Wait_Milli_Seconds(#150)

clr TR0
mov RH0, #high(TIMER0_RELOAD_AH)
mov RL0, #low(TIMER0_RELOAD_AH)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#100)

clr TR0
mov RH0, #high(TIMER0_RELOAD_G)
mov RL0, #low(TIMER0_RELOAD_G)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#100)

clr TR0
mov RH0, #high(TIMER0_RELOAD_D)
mov RL0, #low(TIMER0_RELOAD_D)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#100)

clr TR0
mov RH0, #high(TIMER0_RELOAD_DH)
mov RL0, #low(TIMER0_RELOAD_DH)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#100)

clr TR0
mov RH0, #high(TIMER0_RELOAD_AH)
mov RL0, #low(TIMER0_RELOAD_AH)
setb TR0

lcall WaitHalfSec
lcall WaitHalfSec

clr TR0
```

m)



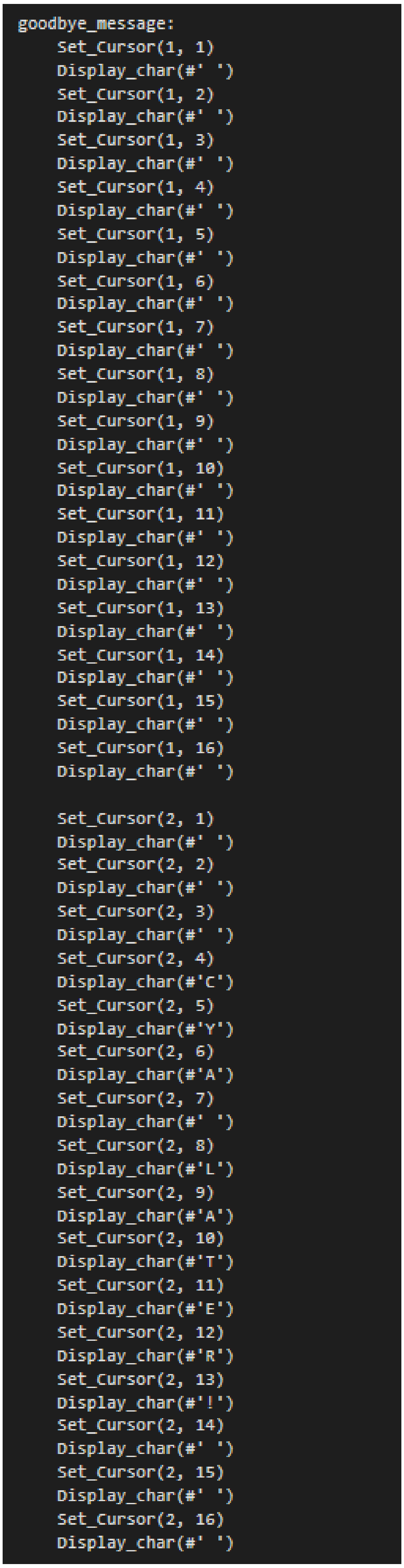
```
ask_prompt:
    ; "WANNA"
    Set_Cursor(1, 1)
    Display_char(#'W')
    Set_Cursor(1, 2)
    Display_char(#'A')
    Set_Cursor(1, 3)
    Display_char(#'N')
    Set_Cursor(1, 4)
    Display_char(#'N')
    Set_Cursor(1, 5)
    Display_char(#'A')
    Set_Cursor(1, 6)
    Display_char(#' ')

    ; "PLAY?"
    Set_Cursor(1, 7)
    Display_char(#'P')
    Set_Cursor(1, 8)
    Display_char(#'L')
    Set_Cursor(1, 9)
    Display_char(#'A')
    Set_Cursor(1, 10)
    Display_char(#'Y')
    Set_Cursor(1, 11)
    Display_char(#'?')

    ; "YES"
    Set_Cursor(2, 1)
    Display_char(#'Y')
    Set_Cursor(2, 2)
    Display_char(#'E')
    Set_Cursor(2, 3)
    Display_char(#'S')

    ; "NO"
    Set_Cursor(2, 6)
    Display_char(#'N')
    Set_Cursor(2, 7)
    Display_char(#'O')
```

n)



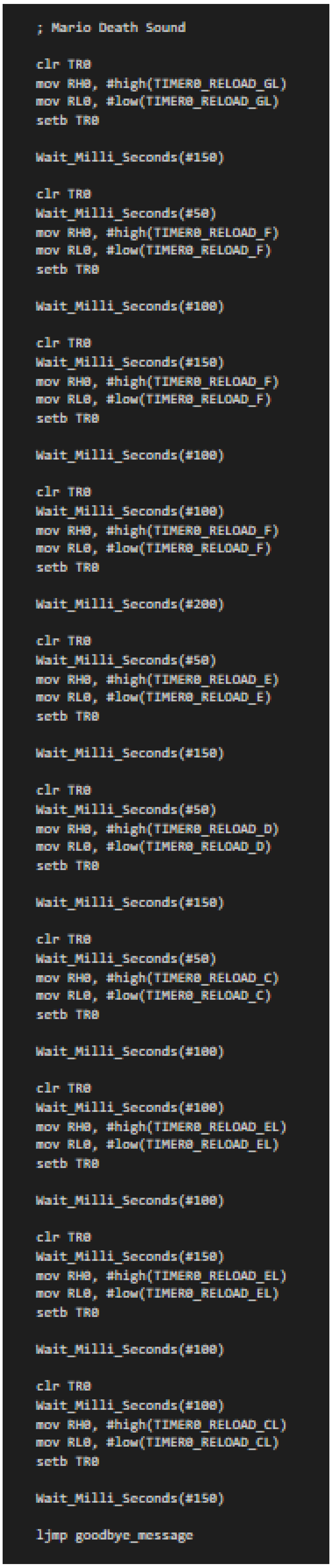
```
cursor_no:
    jb NO_CURSOR, cursor_yes ; if not pressed, goes to branch below
    Wait_Milli_Seconds(#50) ; Debounce delay.  This macro is also in 'LCD_4bit.inc'
    jb NO_CURSOR, cursor_yes ; if not pressed, goes to branch below
    jnb NO_CURSOR, $ ; Wait for button release.  The '$' means: jump to same instruction.
    Set_Cursor(2,4)
    Display_char(#' ')
    Set_Cursor(2,8)
    Display_char(#'<')
    mov R5, #0x00

cursor_yes:
    jb YES_CURSOR, choose_select ; if not pressed, goes to branch below
    Wait_Milli_Seconds(#50) ; Debounce delay.  This macro is also in 'LCD_4bit.inc'
    jb YES_CURSOR, choose_select ; if not pressed, goes to branch below
    jnb NO_CURSOR, $ ; Wait for button release.  The '$' means: jump to same instruction.
    Set_Cursor(2,4)
    Display_char(#'<')
    Set_Cursor(2,8)
    Display_char(#' ')
    mov R5, #0x01

choose_select:
    jb SELECT, cursor_no_x ; if not pressed, goes to branch below
    Wait_Milli_Seconds(#50) ; Debounce delay.  This macro is also in 'LCD_4bit.inc'
    jb SELECT, cursor_no_x ; if not pressed, goes to branch below
    jnb SELECT, $ ; Wait for button release.  The '$' means: jump to same instruction.
    cjne R5, #0x01, goodbye_message ; if equal to one, continues to forever
    ljmp forever

cursor_no_x:
    ljmp cursor_no
```

o)



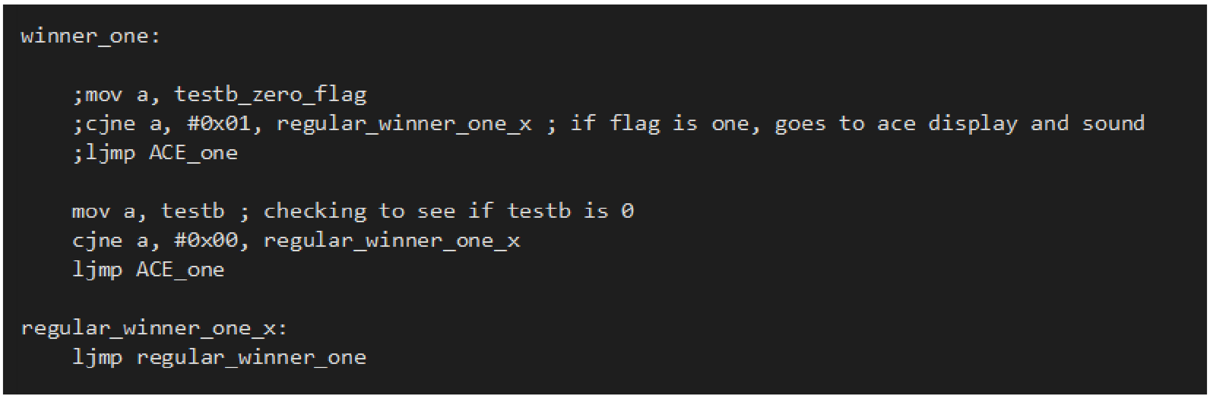
```
goodbye_message:
    Set_Cursor(1, 1)
    Display_char(#' ')
    Set_Cursor(1, 2)
    Display_char(#' ')
    Set_Cursor(1, 3)
    Display_char(#' ')
    Set_Cursor(1, 4)
    Display_char(#' ')
    Set_Cursor(1, 5)
    Display_char(#' ')
    Set_Cursor(1, 6)
    Display_char(#' ')
    Set_Cursor(1, 7)
    Display_char(#' ')
    Set_Cursor(1, 8)
    Display_char(#' ')
    Set_Cursor(1, 9)
    Display_char(#' ')
    Set_Cursor(1, 10)
    Display_char(#' ')
    Set_Cursor(1, 11)
    Display_char(#' ')
    Set_Cursor(1, 12)
    Display_char(#' ')
    Set_Cursor(1, 13)
    Display_char(#' ')
    Set_Cursor(1, 14)
    Display_char(#' ')
    Set_Cursor(1, 15)
    Display_char(#' ')
    Set_Cursor(1, 16)
    Display_char(#' ')

    Set_Cursor(2, 1)
    Display_char(#' ')
    Set_Cursor(2, 2)
    Display_char(#' ')
    Set_Cursor(2, 3)
    Display_char(#' ')
    Set_Cursor(2, 4)
    Display_char(#'C')
    Set_Cursor(2, 5)
    Display_char(#'Y')
    Set_Cursor(2, 6)
    Display_char(#'A')
    Set_Cursor(2, 7)
    Display_char(#' ')
    Set_Cursor(2, 8)
    Display_char(#'L')
    Set_Cursor(2, 9)
    Display_char(#'A')
    Set_Cursor(2, 10)
    Display_char(#'T')
    Set_Cursor(2, 11)
    Display_char(#'E')
    Set_Cursor(2, 12)
    Display_char(#'R')
    Set_Cursor(2, 13)
    Display_char(#'!')
    Set_Cursor(2, 14)
    Display_char(#' ')
    Set_Cursor(2, 15)
    Display_char(#' ')
    Set_Cursor(2, 16)
    Display_char(#' ')
```

p)



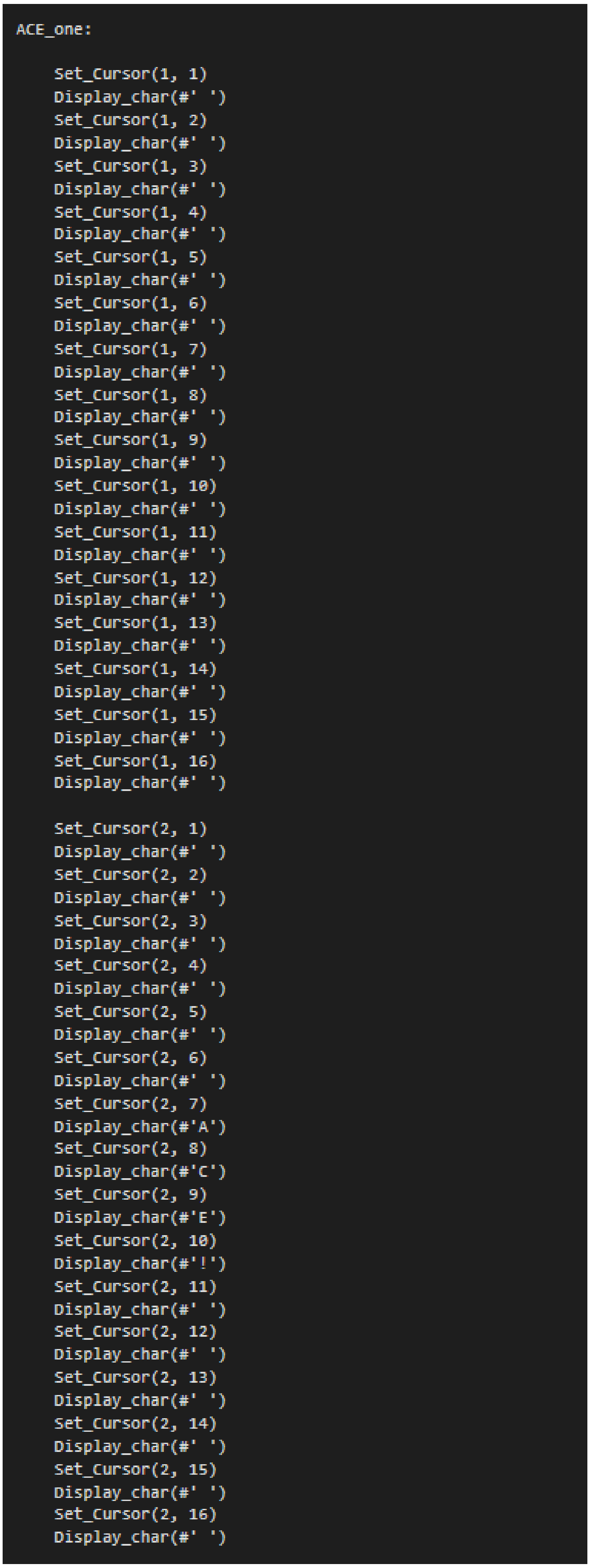
```
; Mario Death Sound

clr TR0
mov RH0, #high(TIMER0_RELOAD_GL)
mov RL0, #low(TIMER0_RELOAD_GL)
setb TR0

Wait_Milli_Seconds(#150)

clr TR0
Wait_Milli_Seconds(#50)
mov RH0, #high(TIMER0_RELOAD_F)
mov RL0, #low(TIMER0_RELOAD_F)
setb TR0

Wait_Milli_Seconds(#100)

clr TR0
Wait_Milli_Seconds(#150)
mov RH0, #high(TIMER0_RELOAD_F)
mov RL0, #low(TIMER0_RELOAD_F)
setb TR0

Wait_Milli_Seconds(#100)

clr TR0
Wait_Milli_Seconds(#100)
mov RH0, #high(TIMER0_RELOAD_F)
mov RL0, #low(TIMER0_RELOAD_F)
setb TR0

Wait_Milli_Seconds(#200)

clr TR0
Wait_Milli_Seconds(#50)
mov RH0, #high(TIMER0_RELOAD_E)
mov RL0, #low(TIMER0_RELOAD_E)
setb TR0

Wait_Milli_Seconds(#150)

clr TR0
Wait_Milli_Seconds(#50)
mov RH0, #high(TIMER0_RELOAD_D)
mov RL0, #low(TIMER0_RELOAD_D)
setb TR0

Wait_Milli_Seconds(#150)

clr TR0
Wait_Milli_Seconds(#50)
mov RH0, #high(TIMER0_RELOAD_C)
mov RL0, #low(TIMER0_RELOAD_C)
setb TR0

Wait_Milli_Seconds(#100)

clr TR0
Wait_Milli_Seconds(#100)
mov RH0, #high(TIMER0_RELOAD_EL)
mov RL0, #low(TIMER0_RELOAD_EL)
setb TR0

Wait_Milli_Seconds(#100)

clr TR0
Wait_Milli_Seconds(#150)
mov RH0, #high(TIMER0_RELOAD_EL)
mov RL0, #low(TIMER0_RELOAD_EL)
setb TR0

Wait_Milli_Seconds(#100)

clr TR0
Wait_Milli_Seconds(#100)
mov RH0, #high(TIMER0_RELOAD_CL)
mov RL0, #low(TIMER0_RELOAD_CL)
setb TR0

Wait_Milli_Seconds(#150)

ljmp goodbye_message
```

q)



```
winner_one:

    ;mov a, testb_zero_flag
    ;cjne a, #0x01, regular_winner_one_x ; if flag is one, goes to ace display and sound
    ;ljmp ACE_one

    mov a, testb ; checking to see if testb is 0
    cjne a, #0x00, regular_winner_one_x
    ljmp ACE_one

regular_winner_one_x:
    ljmp regular_winner_one
```

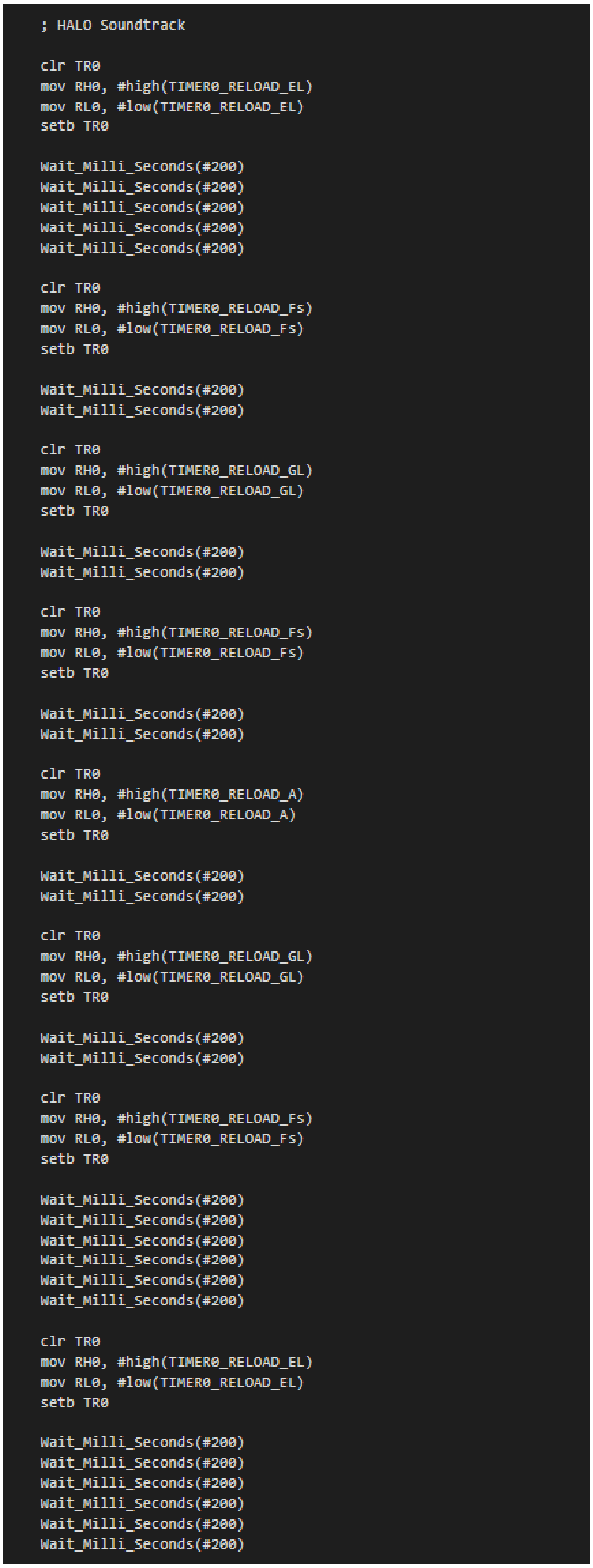
r)



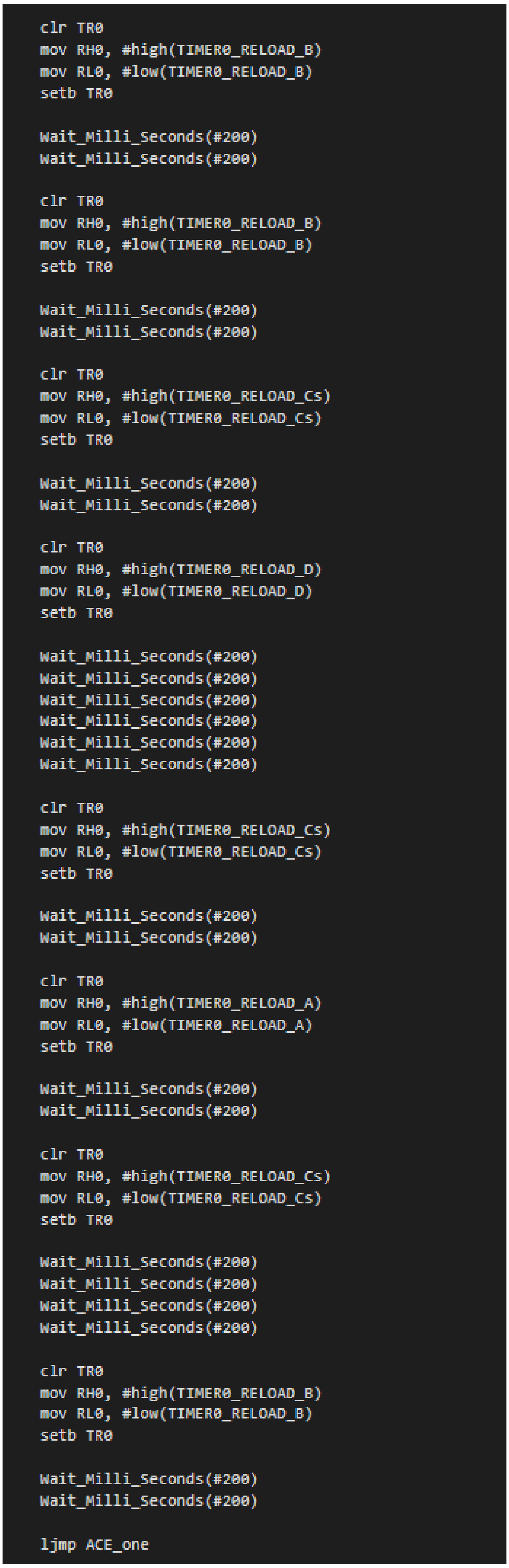
```
ACE_one:

    Set_Cursor(1, 1)
    Display_char(#' ')
    Set_Cursor(1, 2)
    Display_char(#' ')
    Set_Cursor(1, 3)
    Display_char(#' ')
    Set_Cursor(1, 4)
    Display_char(#' ')
    Set_Cursor(1, 5)
    Display_char(#' ')
    Set_Cursor(1, 6)
    Display_char(#' ')
    Set_Cursor(1, 7)
    Display_char(#' ')
    Set_Cursor(1, 8)
    Display_char(#' ')
    Set_Cursor(1, 9)
    Display_char(#' ')
    Set_Cursor(1, 10)
    Display_char(#' ')
    Set_Cursor(1, 11)
    Display_char(#' ')
    Set_Cursor(1, 12)
    Display_char(#' ')
    Set_Cursor(1, 13)
    Display_char(#' ')
    Set_Cursor(1, 14)
    Display_char(#' ')
    Set_Cursor(1, 15)
    Display_char(#' ')
    Set_Cursor(1, 16)
    Display_char(#' ')

    Set_Cursor(2, 1)
    Display_char(#' ')
    Set_Cursor(2, 2)
    Display_char(#' ')
    Set_Cursor(2, 3)
    Display_char(#' ')
    Set_Cursor(2, 4)
    Display_char(#' ')
    Set_Cursor(2, 5)
    Display_char(#' ')
    Set_Cursor(2, 6)
    Display_char(#' ')
    Set_Cursor(2, 7)
    Display_char(#'A')
    Set_Cursor(2, 8)
    Display_char(#'C')
    Set_Cursor(2, 9)
    Display_char(#'E')
    Set_Cursor(2, 10)
    Display_char(#'!')
    Set_Cursor(2, 11)
    Display_char(#' ')
    Set_Cursor(2, 12)
    Display_char(#' ')
    Set_Cursor(2, 13)
    Display_char(#' ')
    Set_Cursor(2, 14)
    Display_char(#' ')
    Set_Cursor(2, 15)
    Display_char(#' ')
    Set_Cursor(2, 16)
    Display_char(#' ')
```

r)

s)

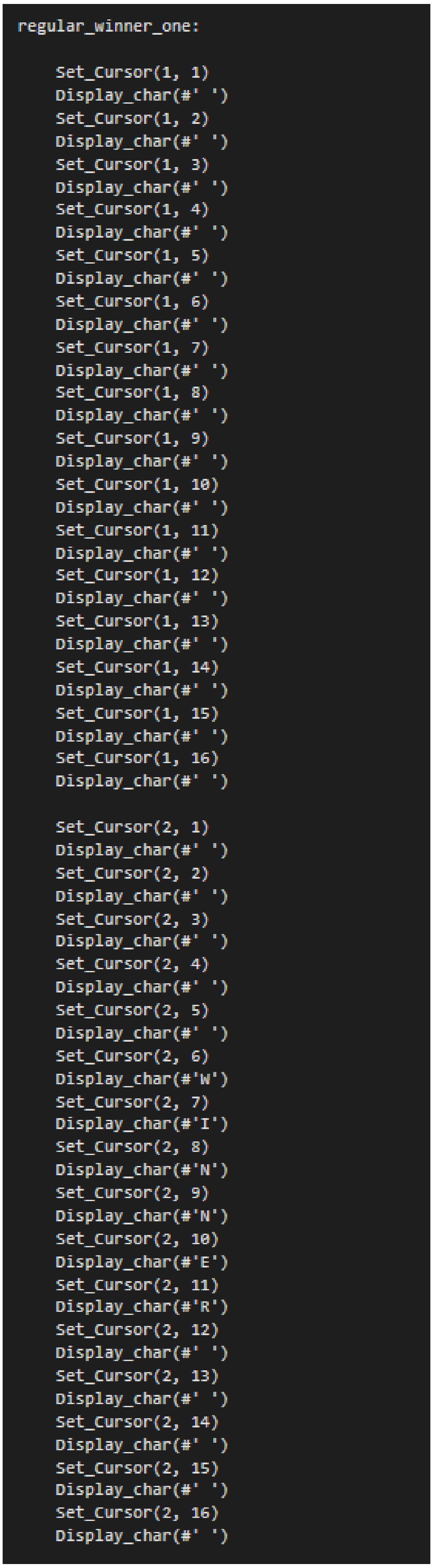


```
; HALO Soundtrack

clr TR0
mov RH0, #high(TIMER0_RELOAD_EL)
mov RL0, #low(TIMER0_RELOAD_EL)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

clr TR0
mov RH0, #high(TIMER0_RELOAD_Fs)
mov RL0, #low(TIMER0_RELOAD_Fs)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

clr TR0
mov RH0, #high(TIMER0_RELOAD_GL)
mov RL0, #low(TIMER0_RELOAD_GL)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

clr TR0
mov RH0, #high(TIMER0_RELOAD_Fs)
mov RL0, #low(TIMER0_RELOAD_Fs)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

clr TR0
mov RH0, #high(TIMER0_RELOAD_A)
mov RL0, #low(TIMER0_RELOAD_A)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

clr TR0
mov RH0, #high(TIMER0_RELOAD_GL)
mov RL0, #low(TIMER0_RELOAD_GL)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

clr TR0
mov RH0, #high(TIMER0_RELOAD_Fs)
mov RL0, #low(TIMER0_RELOAD_Fs)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

clr TR0
mov RH0, #high(TIMER0_RELOAD_EL)
mov RL0, #low(TIMER0_RELOAD_EL)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
```



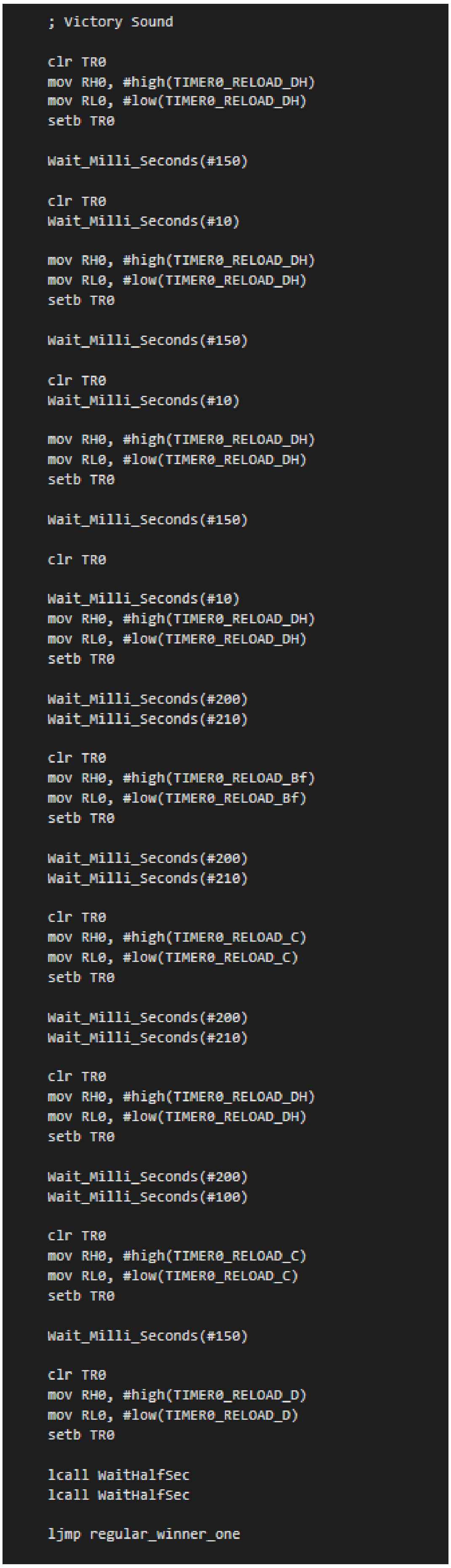
```
clr TR0
mov RH0, #high(TIMER0_RELOAD_B)
mov RL0, #low(TIMER0_RELOAD_B)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

clr TR0
mov RH0, #high(TIMER0_RELOAD_B)
mov RL0, #low(TIMER0_RELOAD_B)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

clr TR0
mov RH0, #high(TIMER0_RELOAD_Cs)
mov RL0, #low(TIMER0_RELOAD_Cs)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

clr TR0
mov RH0, #high(TIMER0_RELOAD_D)
mov RL0, #low(TIMER0_RELOAD_D)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

clr TR0
mov RH0, #high(TIMER0_RELOAD_Cs)
mov RL0, #low(TIMER0_RELOAD_Cs)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

clr TR0
mov RH0, #high(TIMER0_RELOAD_A)
mov RL0, #low(TIMER0_RELOAD_A)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

clr TR0
mov RH0, #high(TIMER0_RELOAD_Cs)
mov RL0, #low(TIMER0_RELOAD_Cs)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

clr TR0
mov RH0, #high(TIMER0_RELOAD_B)
mov RL0, #low(TIMER0_RELOAD_B)
setb TR0

Wait_Milli_Seconds(#200)
Wait_Milli_Seconds(#200)

ljmp ACE_one
```

t)



```
regular_winner_one:

    Set_Cursor(1, 1)
    Display_char(#' ')
    Set_Cursor(1, 2)
    Display_char(#' ')
    Set_Cursor(1, 3)
    Display_char(#' ')
    Set_Cursor(1, 4)
    Display_char(#' ')
    Set_Cursor(1, 5)
    Display_char(#' ')
    Set_Cursor(1, 6)
    Display_char(#' ')
    Set_Cursor(1, 7)
    Display_char(#' ')
    Set_Cursor(1, 8)
    Display_char(#' ')
    Set_Cursor(1, 9)
    Display_char(#' ')
    Set_Cursor(1, 10)
    Display_char(#' ')
    Set_Cursor(1, 11)
    Display_char(#' ')
    Set_Cursor(1, 12)
    Display_char(#' ')
    Set_Cursor(1, 13)
    Display_char(#' ')
    Set_Cursor(1, 14)
    Display_char(#' ')
    Set_Cursor(1, 15)
    Display_char(#' ')
    Set_Cursor(1, 16)
    Display_char(#' ')

    Set_Cursor(2, 1)
    Display_char(#' ')
    Set_Cursor(2, 2)
    Display_char(#' ')
    Set_Cursor(2, 3)
    Display_char(#' ')
    Set_Cursor(2, 4)
    Display_char(#' ')
    Set_Cursor(2, 5)
    Display_char(#' ')
    Set_Cursor(2, 6)
    Display_char(#'W')
    Set_Cursor(2, 7)
    Display_char(#'I')
    Set_Cursor(2, 8)
    Display_char(#'N')
    Set_Cursor(2, 9)
    Display_char(#'N')
    Set_Cursor(2, 10)
    Display_char(#'E')
    Set_Cursor(2, 11)
    Display_char(#'R')
    Set_Cursor(2, 12)
    Display_char(#' ')
    Set_Cursor(2, 13)
    Display_char(#' ')
    Set_Cursor(2, 14)
    Display_char(#' ')
    Set_Cursor(2, 15)
    Display_char(#' ')
    Set_Cursor(2, 16)
    Display_char(#' ')
```

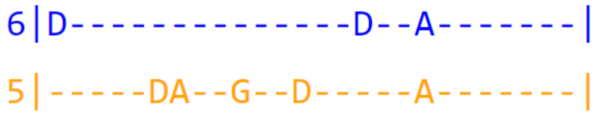
u)



```
    ; Victory Sound

    clr TR0
    mov RH0, #high(TIMER0_RELOAD_DH)
    mov RL0, #low(TIMER0_RELOAD_DH)
    setb TR0

    Wait_Milli_Seconds(#150)

    clr TR0
    Wait_Milli_Seconds(#10)

    mov RH0, #high(TIMER0_RELOAD_DH)
    mov RL0, #low(TIMER0_RELOAD_DH)
    setb TR0

    Wait_Milli_Seconds(#150)

    clr TR0
    Wait_Milli_Seconds(#10)

    mov RH0, #high(TIMER0_RELOAD_DH)
    mov RL0, #low(TIMER0_RELOAD_DH)
    setb TR0

    Wait_Milli_Seconds(#150)

    clr TR0

    Wait_Milli_Seconds(#10)
    mov RH0, #high(TIMER0_RELOAD_DH)
    mov RL0, #low(TIMER0_RELOAD_DH)
    setb TR0

    Wait_Milli_Seconds(#200)
    Wait_Milli_Seconds(#210)

    clr TR0
    mov RH0, #high(TIMER0_RELOAD_Bf)
    mov RL0, #low(TIMER0_RELOAD_Bf)
    setb TR0

    Wait_Milli_Seconds(#200)
    Wait_Milli_Seconds(#210)

    clr TR0
    mov RH0, #high(TIMER0_RELOAD_C)
    mov RL0, #low(TIMER0_RELOAD_C)
    setb TR0

    Wait_Milli_Seconds(#200)
    Wait_Milli_Seconds(#210)

    clr TR0
    mov RH0, #high(TIMER0_RELOAD_DH)
    mov RL0, #low(TIMER0_RELOAD_DH)
    setb TR0

    Wait_Milli_Seconds(#200)
    Wait_Milli_Seconds(#100)

    clr TR0
    mov RH0, #high(TIMER0_RELOAD_C)
    mov RL0, #low(TIMER0_RELOAD_C)
    setb TR0

    Wait_Milli_Seconds(#150)

    clr TR0
    mov RH0, #high(TIMER0_RELOAD_D)
    mov RL0, #low(TIMER0_RELOAD_D)
    setb TR0

    lcall WaitHalfSec
    lcall WaitHalfSec

    ljmp regular_winner_one
```

Appendix 2: Music Score Sheets

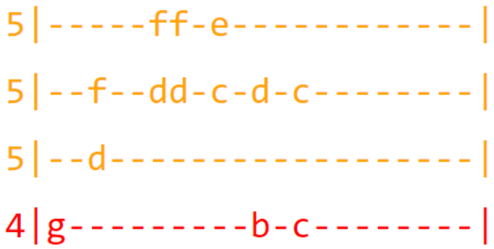
a) Windows XP Boot Sound



```
6|D--------------D--A-------|
5|-----DA--G--D-----A-------|
```

“Windows XP startup and Shutdown sounds,” *Windows XP Startup and Shutdown Sounds* ~. [Online]. Available: https://pianoletternotes.blogspot.com/2017/10/windows-xp-startup-and-shutdown-sounds.html. [Accessed: 04-Mar-2022].

b) Mario Game Death Sound



```
5|-----ff-e------------|
5|--f--dd-c-d-c--------|
5|--d------------------|
4|g---------b-c--------|
```

“Super mario death theme,” *Super Mario Death Theme* ~. [Online]. Available: https://pianoletternotes.blogspot.com/2017/10/super-mario-death-theme.html. [Accessed: 04-Mar-2022].

c) Standard Victory Sound

“Victory sound effect - youtube.” [Online]. Available: https://www.youtube.com/watch?v=xP1b_uRx5x4. [Accessed: 05-Mar-2022].

Notes were determined through listening - Mark Tan is a musical genius

d) Halo 2 Theme

```
4|e-------F---g---F---a---g-|

4|--F-----------e-----------|

5|--------C---d-------C-----|
4|----b-------------------a-|

5|--C-----------------------|
4|------b---b---------------|
```

“Halo Theme (easy version),” *Halo Theme (Easy Version)* ~. [Online]. Available: https://pianoletternotes.blogspot.com/2019/06/halo-theme-easy-version.html. [Accessed: 04-Mar-2022].

Appendix 3: Pictures of Logic Diagrams, Schematics, and Tables related to Hardware

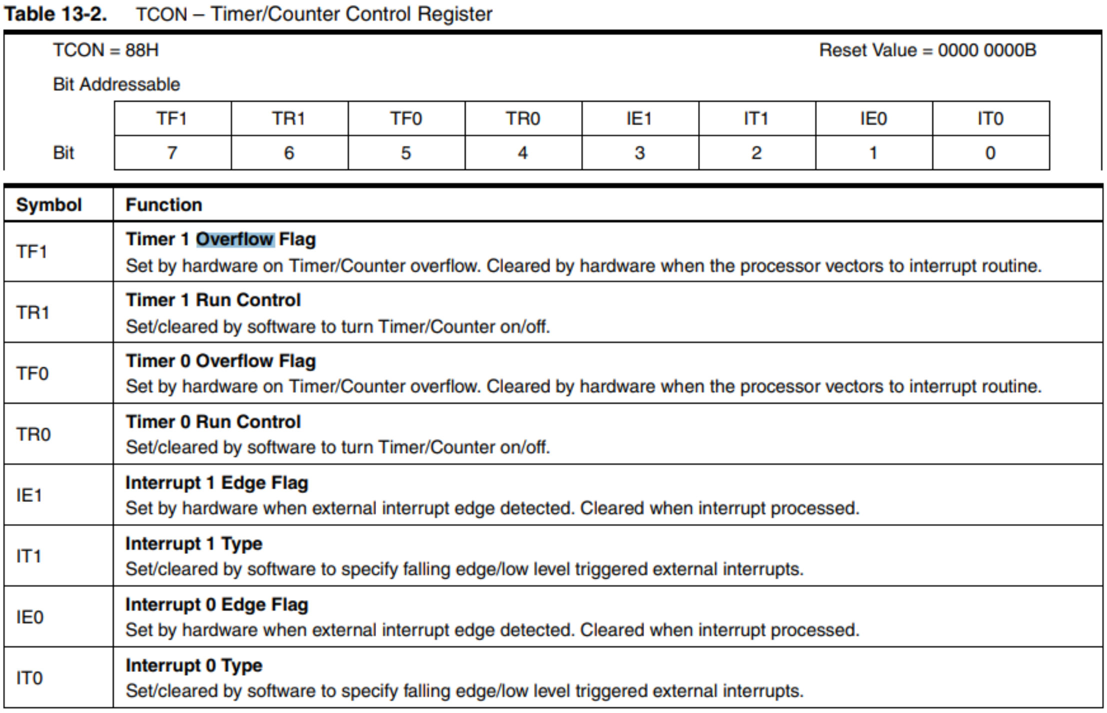


Table 13-2. TCON – Timer/Counter Control Register

TCON = 88H Reset Value = 0000 0000B

Bit Addressable

	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
Bit	7	6	5	4	3	2	1	0

Symbol	Function
TF1	**Timer 1 Overflow Flag** Set by hardware on Timer/Counter overflow. Cleared by hardware when the processor vectors to interrupt routine.
TR1	**Timer 1 Run Control** Set/cleared by software to turn Timer/Counter on/off.
TF0	**Timer 0 Overflow Flag** Set by hardware on Timer/Counter overflow. Cleared by hardware when the processor vectors to interrupt routine.
TR0	**Timer 0 Run Control** Set/cleared by software to turn Timer/Counter on/off.
IE1	**Interrupt 1 Edge Flag** Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
IT1	**Interrupt 1 Type** Set/cleared by software to specify falling edge/low level triggered external interrupts.
IE0	**Interrupt 0 Edge Flag** Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
IT0	**Interrupt 0 Type** Set/cleared by software to specify falling edge/low level triggered external interrupts.

a)

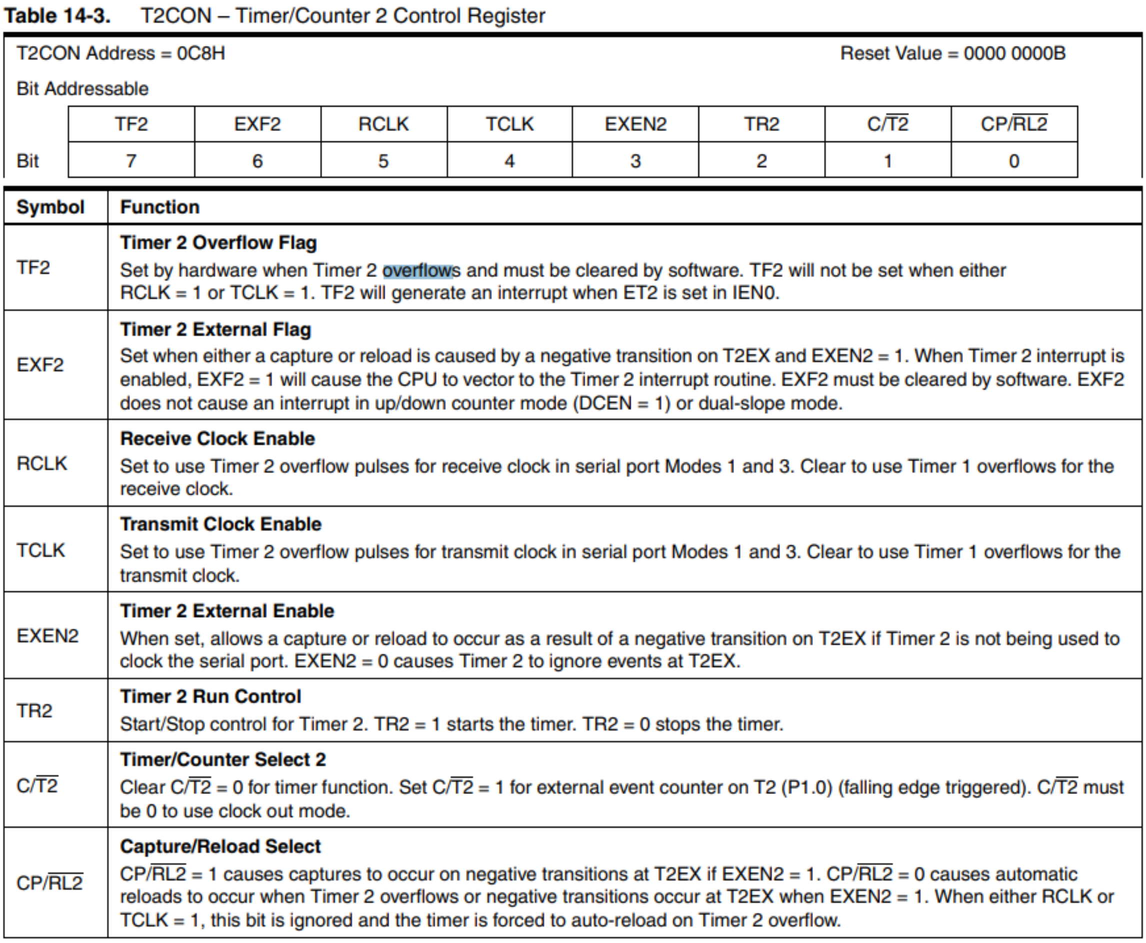


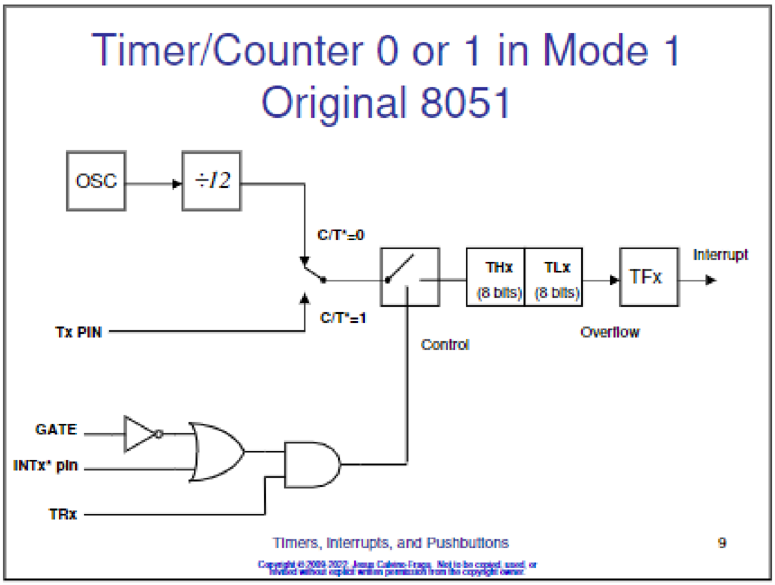
Table 14-3. T2CON – Timer/Counter 2 Control Register

T2CON Address = 0C8H Reset Value = 0000 0000B

Bit Addressable

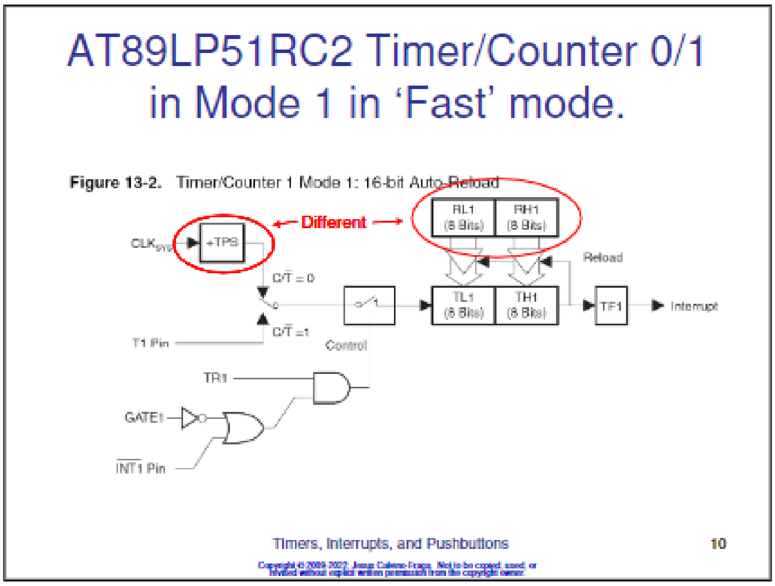
	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{\text{T2}}$	CP/$\overline{\text{RL2}}$
Bit	7	6	5	4	3	2	1	0

Symbol	Function
TF2	**Timer 2 Overflow Flag** Set by hardware when Timer 2 overflows and must be cleared by software. TF2 will not be set when either RCLK = 1 or TCLK = 1. TF2 will generate an interrupt when ET2 is set in IEN0.
EXF2	**Timer 2 External Flag** Set when either a capture or reload is caused by a negative transition on T2EX and EXEN2 = 1. When Timer 2 interrupt is enabled, EXF2 = 1 will cause the CPU to vector to the Timer 2 interrupt routine. EXF2 must be cleared by software. EXF2 does not cause an interrupt in up/down counter mode (DCEN = 1) or dual-slope mode.
RCLK	**Receive Clock Enable** Set to use Timer 2 overflow pulses for receive clock in serial port Modes 1 and 3. Clear to use Timer 1 overflows for the receive clock.
TCLK	**Transmit Clock Enable** Set to use Timer 2 overflow pulses for transmit clock in serial port Modes 1 and 3. Clear to use Timer 1 overflows for the transmit clock.
EXEN2	**Timer 2 External Enable** When set, allows a capture or reload to occur as a result of a negative transition on T2EX if Timer 2 is not being used to clock the serial port. EXEN2 = 0 causes Timer 2 to ignore events at T2EX.
TR2	**Timer 2 Run Control** Start/Stop control for Timer 2. TR2 = 1 starts the timer. TR2 = 0 stops the timer.
C/$\overline{\text{T2}}$	**Timer/Counter Select 2** Clear C/$\overline{\text{T2}}$ = 0 for timer function. Set C/$\overline{\text{T2}}$ = 1 for external event counter on T2 (P1.0) (falling edge triggered). C/$\overline{\text{T2}}$ must be 0 to use clock out mode.
CP/$\overline{\text{RL2}}$	**Capture/Reload Select** CP/$\overline{\text{RL2}}$ = 1 causes captures to occur on negative transitions at T2EX if EXEN2 = 1. CP/$\overline{\text{RL2}}$ = 0 causes automatic reloads to occur when Timer 2 overflows or negative transitions occur at T2EX when EXEN2 = 1. When either RCLK or TCLK = 1, this bit is ignored and the timer is forced to auto-reload on Timer 2 overflow.



Timer/Counter 0 or 1 in Mode 1
Original 8051

Timers, Interrupts, and Pushbuttons 9



AT89LP51RC2 Timer/Counter 0/1
in Mode 1 in 'Fast' mode.

Figure 13-2. Timer/Counter 1 Mode 1: 16-bit Auto-Reload

Timers, Interrupts, and Pushbuttons 10

b)

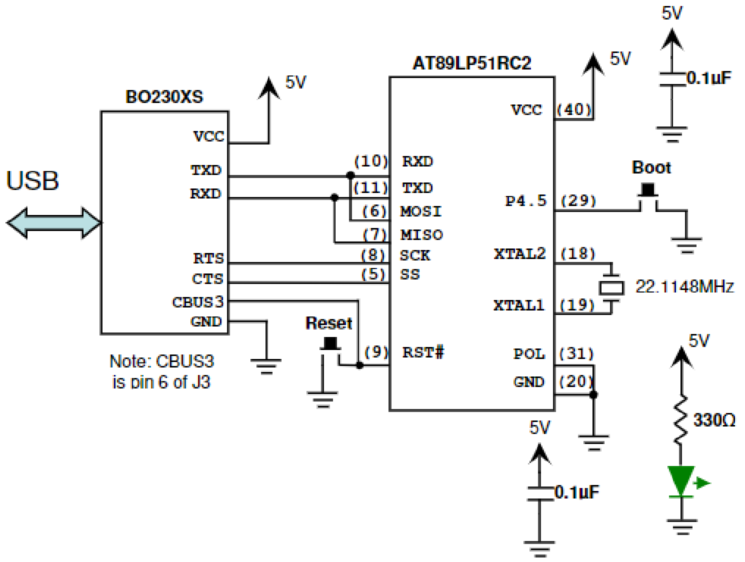
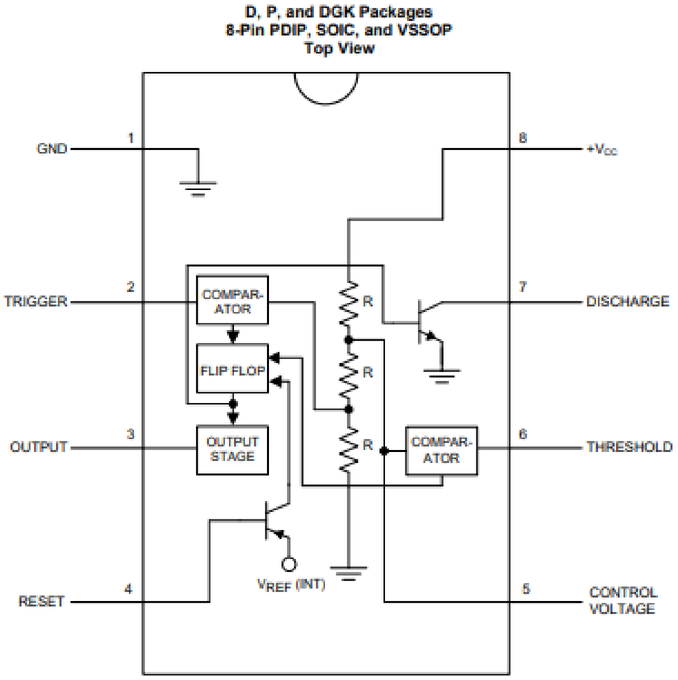


Figure 1. Circuit schematic of the microcontroller system.

c)



d)

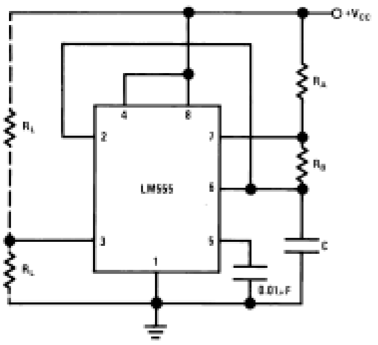
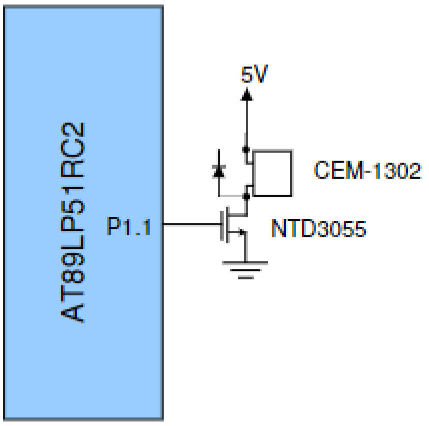
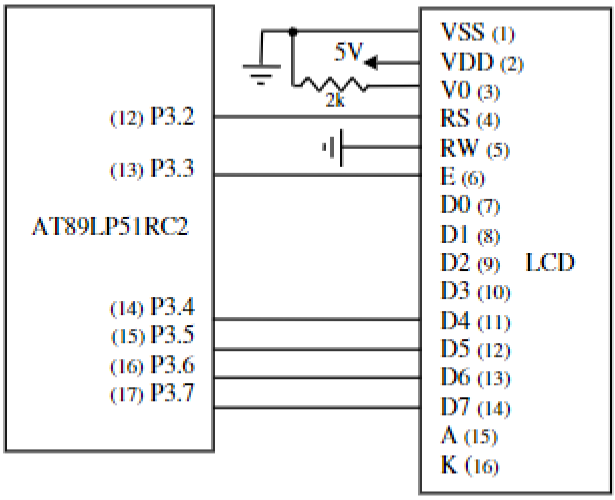


Figure 14. Astable

e)


AT89LP51RC2
5V
CEM-1302
P1.1
NTD3055

f)


(12) P3.2
(13) P3.3
AT89LP51RC2
(14) P3.4
(15) P3.5
(16) P3.6
(17) P3.7
5V
2k
VSS (1)
VDD (2)
V0 (3)
RS (4)
RW (5)
E (6)
D0 (7)
D1 (8)
D2 (9)
LCD
D3 (10)
D4 (11)
D5 (12)
D6 (13)
D7 (14)
A (15)
K (16)

g)