ECSE 444 Final Project Report

Interstellar Exploration Game

Group 1: Ao Shen, 260898863, Byron Chen, 260892558, Kevin Li, 260924898, Kua Chen, 260856888

***Abstract*—In this project, a game, namely Interstellar Exploration Game, was designed by combining the features provided by STM32, i.e. gyroscope of I^{2}C, UART, DAC Speaker and QSPI Flash Memory. By doing unit tests for each feature and the integration tests, the whole game is proved to be bug-free. Furthermore, to ensure a good user experience, exploratory tests were done many times as the last part of the project.**

***Index Terms*—STM32, I^{2}C, UART, DAC Speaker, QSPI Flash Memory, C Code**

I. Introduction

IN this project, a game called Interstellar Exploration Game is designed and implemented by using the features from STM32, i.e. gyroscope of I^{2}C, UART, DAC Speaker and QSPI Flash Memory. In this game, the player can use the board like a steering wheel to control the satellite on the terminal transmitted by UART to avoid obstacles. During the game, DAC Speaker would play different music stored in QSPI Flash Memory according to the hard level of the game. By distributing each feature to different team members, the efficiency of the development was ensured. Furthermore, all features were tested before integrating, and integration tests were done after all parts were combined. Since the user experience is essential for a game, exploratory tests have been done many times for adjusting the steering sensitivity of the satellite and the hardness of the game.

II. Methodology

This project can be divided into four parts, i.e. Steering Recognition, Music Playing, UART & UI Design and Logic Design. The rest of the section will describe how each part was implemented in details.

A. Steering Recognition – Gyroscope

The satellite's movement is controlled by the degree velocity of the STM32 board, like a steering wheel. The board should be held horizontally with the USB wire pointing up when playing the game. Since the gyroscope does not directly measure the actual degree velocity, but the angular acceleration of the board along three axes instead. Therefore, the values are transformed into real numbers through integration and conversion. The integration implementation is shown in Listing 1, whereas the conversion from angular velocity to degree velocity is shown in Formula 1.

```c
    /**
      * @brief Do the  integration for the gyroscope
      *
      * @param sample_x: Raw value of x
      * @param sample_y: Raw value of y
      * @param sample_z: Raw value of z
      *
      * @retval None
      */
    void gyroIntegrate(int sample_x, int sample_y,
    int sample_z)
    {
        ax += sample_x - previous_gyro[0];
        ay += sample_y - previous_gyro[1];
        az += sample_z - previous_gyro[2];
    }
```

Listing 1. Integration of the Gyroscope Raw Data

$$degree\_x = \frac{angular\_x \times s}{r} \tag{1}$$

Where the s is the sampling rate of the gyroscope and r is the gyro resolution in degrees per second. In this project, the sampling rate is set to be 100Hz, and the gyro resolution is set to be 0.07 according to the sensitivity of 2000 mdps/LSB of the gyroscope according to the table shown in Figure 11 [2]. Since the project is using timer interrupt, it is important to avoid using the floating-point numbers. In this case, both $anular\_x$ and $\frac{s}{r}$ are multiplied by 10 and round to 14286. As a result, the actual data is calculated following Formula 2.

$$degree\_x = \frac{angular\_x \times 10}{14286} \tag{2}$$

As mentioned before, the board should be held horizontally with the USB wire pointing up. Therefore, only one axis, Z, is involved in deciding the turning direction of the satellite. If the result is smaller or equal to -100, the $turning\_flag$ would be set to 1, which signifies the tuning right on UI. When the result is greater or equal to 100, the $turning\_flag$ would be set to -1, which signifies the turning left instruction. If none of the conditions is satisfied, $turning\_flag$ will stay with the value of 0.

B. Music Playing – DAC Speaker & QSPI Flash Memory

Music is an essential part of a game. It makes the game more exciting and can also be used as an interaction with the players. In this project, a piece of music is created consisting of 8 different tones, i.e. D5, D6, A5, GG5, G5, F5, C5 and A5. All of these notes are stored in the form of an array, and the array's length is calculated by Formula 3.

$$Length = \frac{System\ Clock\ Frequency}{Counter \times Frequency\ of\ Signal} \tag{3}$$

To be noticed, since the sampling rate is chosen to be 44.1kHz whereas the system clock frequency is chosen to be 80MHz, the counter of the timer is set to be $round(\frac{80MHz}{44.1kHz}) = 1814$ in this project. The frequencies of each tone are shown in Figure 1 [1].

In order to create a sine wave efficiently, arm_sin_f32 in the arm_math library is used in this lab. Since the period of a sine wave is 2π in radians, this wave should reach the maximum amplitude at $\frac{\pi}{2}$ and minimum amplitude at $\frac{3\pi}{2}$. The 8-bit resolution is used to store the tones to save the memory, which means the sine wave amplitude should be from 0 to 256. Furthermore, since the DAC value should be scaled by 2/3 to ensure the sound made is continuous, the amplitudes of the sine wave are calculated by Formula 4.

$$0.33 \times (1 + sin(\frac{2i \cdot \pi}{Length} \times 256) \tag{4}$$

The implementation of generating a D5 tone is shown in Listing 2.

```
  for (uint32_t i = 0; i < 16; i++) {
    float32_t radians = 3.14 / 8 * i;
    sine_wave[i] = (uint32_t) (2047.5 * (1 +
  arm_sin_f32(radians)));
  }
```

Listing 2. Creation of the Sine Wave

Since a rhythm is made by combining many different notes, it is impractical to represent a rhythm with one single array due to the enormous memory space requirements. As a result, QSPI Flash is used as a better method to store the rhythm. The setup of the QSPI Flash is done in CubeMx and is shown in Figure 12. To notice that, the mode of the OCTOSPI1 should be chosen as Quad SPI since data is transferred, received or both over four lines[3]. In order to provide audio feedback to the players, two different versions of the same piece of music are created but with different lasting times in the array to create different bps. At the beginning of the game, a slow version of this music would be displayed by DAC; if the player can achieve a score higher than 150, the fast version of this music would be displayed by DAC to signify that the generation of obstacles becomes faster. Furthermore, the music's changing can distract the player and make them more nervous during the game.

The fast version and slow version of the music are stored in the Flash with different start addresses, namely $slow\_music\_addr$ (0x00020000) and $game\_music\_addr$ (0x00000000). A big gap is ensured between the two starting addresses to give them enough space for storing the music. Both rhythms use the same notes. The only difference is the duration of each note. Each note would be stored 50 times for the fast rhythm, and for the slow rhythm, each note would be stored 100 times. To make the music more soundable, a pulse note is also set between each note during the creation of the music.

Before writing the music into the QSPI Flash, the blocks are erased by using the function $BSP\_QSPI\_Erase\_Block$ in a for loop with different addresses. After initializing the QSPI, the music is written into it. By checking the variable table in the debugger, the length for the fast and slow versions are 92460 and 184760, respectively. As a result, two arrays, namely $megalovania\_slow$ and $megalovania\_fast$, are created with the lengths for future usage. The slow version would be stored into the corresponding array whenever the user starts the game by calling the function $BSP\_QSPI\_Read((uint8\_t^*)megalovania\_slow, 0x00020000, 184760)$. DMA is implemented after getting this value by calling the function $HAL\_DAC\_Stop\_DMA$, which stops the previous DMA conversion and $HAL\_DAC\_Start\_DMA$, which outputs the data in an array of a given length. The same thing is done to the fast version whenever the score reaches 150.

C. UART & UI Design

The user interface shows all the things for a game. This game consists of three game statuses: before-game, in-gaming, and after-game.

In the before-game stage, a welcome message would be printed on the terminal, as shown in Figure 2 and Figure 7. It indicates that users press the blue button to start the game.

The UI of the satellite, whose position is controlled by the user and the obstacle whose position is randomly generated is shown in Figure 5 and Figure 6, respectively. The satellite is always placed at the bottom, but the user can control the horizontal movement of the satellite to switch tracks to avoid obstacles. Both of them are in the size of 5 characters, which is consistent with the width of each track on the map. As shown in Figure 3, the map mainly consists of 4 tracks, and there is some visualization of the wave on its left to simulate the movement of the track. An example can be found in Figure 8.

In the after-game status, an ending message will be printed. This message is split into two, as shown in Figure 4. This is because we can place the score between the two components of the ending message. The appearance on the terminal for this status is shown in Figure 9.

The essence of this game is car-racing, where users drive the car (satellite in our case) and avoid obstacles. To give a visual illusion that the satellite is moving forward, considering simple relativity, our implementation is: the map (ground) is moving backward, and the satellite (car) stands still. As shown in Figure 3, the game map consists of 10 rows. The ”moving downward” is implemented as moving the 1st row down to the 2nd row, 2nd row down to the 3rd row, 3rd row down to the 4th row, Etc. , simultaneously. In practice, an easier way to achieve it is moving rows in reverse order: move the 9th row to the 10th row, then move the 8th row to the 9th row, move the 7th row to the 8th row, Etc., sequence.

The above discussion covers the user interface, which is purely made of characters. In order to transfer these characters from the board to the screen, the UART feature is used to achieve the transmission of characters. We should enable USART1 in CubeMX and need to configure pins to PB7 and PB6 and select $USART1\_RX$ and $USART1\_TX$ for PB7 and PB6, respectively. These two particular pins are selected because they are connected to the USB port, and the board is communicating with the computer through a USB port with a Cable. Other configuration settings work well with the default values. Data transmission can be achieved by calling the function named $HAL\_UART\_Transmit$. The parameters required for this function are the address of the UART handle type, the character string starting address, the size of the string and the timeout.

D. Logic Design

The logical level is the skeleton of the whole project. Its main task is to integrate all other functions and enable data transfer between different features.

- Update the screen output.
- Read and update player position information.
- Generate/update obstacle position information.
- Collision detection.
- Score system.
- Control game refresh frequency.

We maintain a one-dimensional array of length 4 to represent the obstacle information. The index of a non-zero number in the array is the same as the track where the obstacle is located. Moreover, this non-zero number represents the vertical distance from the current obstacle to the satellite. If there is no obstacle on the map, the arrays would consist of all zeros. Suppose all the index values in this one-dimensional array are 0s. In that case, we generate a random number between 0 and 3 to represent the new obstacle's track and place it at the farthest distance (10) from the player by setting the value of this index. The player will have the illusion of a constant stream of incoming meteorites. The random number is generated by referring to the $rand$ method in the $stdlib$ library. The remainder of the random number divided by 4 is taken to ensure that the generated random number is within 0 to 3.

Since the game only supports the left and right player's movement, an integer represents the player's position. This value indicates the orbit where the player-controlled satellite is located. The player's position would be updated by adding or subtracting one unit from this value each time. This value is given by the gyroscope described in Section A.

In this project, all the stuff is implemented inside the infinite while loop in the primary function, and after each iteration, a 10ms delay is implemented by HAL_Delay to achieve a 100Hz update frequency. Here is the main design of this project: since the whole program is designed inside one single thread, how to control the frequency of different parts becomes a big problem. For example, if the update frequency for the satellite is the same as that of the obstacles, the game would not work. As a result, the updating rate of the map and obstacle is set to be once every $MAX\_COUNT - 1$ iteration by using a counter and a mod function. By doing this, the updating of the map and obstacles can be separated from the updating of the satellite, where the movement of the satellite is updated at every single iteration. Furthermore, to indicate the starting and restarting of the game, GPIO interrupt is enabled for checking the blue button. In the stage of before-game and after-game, pressing the blue button will bring the program to the next stage.

The frequency of the collision detection is the same as the updating rate of the map and obstacle. Whenever the counter reaches the max value, the satellite's position would be fetched to check if the obstacle is on the same track as the satellite does with a distance of 0 to the satellite. If this happens, a game over signal would be sent, and the status would be changed to after-game status.

The scoring system for this game would update the score for the player each time the map is updated. As a result, the frequency of updating it is the same as the collision detection. Moreover, the challenging level would be changed to a score of 50 and a score of 150. Whenever it reaches the thresholds, MAX_COUNT, with an initial value of 3, would be minus by 1. This means that the refreshing speed of the map and the obstacle would be faster, which can make the whole game harder than the previous level.

III. Tests

A. Unit Tests

1) Steering Recognition – Gyroscope: The test for the gyroscope is done by printing out the values of the raw and actual data onto the terminal using UART. The board was rotated in different directions during the testing to see if the gyroscope was working correctly. Furthermore, the actual data is compared to the raw data to see if the calculation is correct.

Moreover, The sensitivity of the turning recognition needs to be suitable for the players. As a result, the thresholds can not be too large or small. In this part, the values of the thresholds were set to be 200 and -200, which are not the final values. The final values for the thresholds are tested in the exploratory test explained in Section C.

2) DAC Speaker: The testing of the DAC speaker is conducted to ensure the sound of musical notes is correct and the speaker is working. There are 65 notes generated in total, and 8 notes are used. Each generated note was displayed in this project and compared to the actual musical note to ensure the sound quality. The connection of the wires is shown in Figure 10. Furthermore, after writing the music, it was also played by the DAC speaker to see if it is outputting the music as preconceived.

3) QSPI Flash Memory: QSPI Flash memory was tested by using the debugger. After writing the game's music to the QSPI Flash memory, it was loaded to an array after. From the variable table, the values in the array were compared with the written music to see if it was working correctly.

4) UART & UI Design: This part of testing is divided into small tests. The first test is testing the UART: whether the message could be printed in the terminal. The second test is checking whether the map could be updated correctly. The "update" means moving each row to the row below it. The third test is checking whether the satellite could be integrated into the blank space of each track. Satellite position is stored as an integer value ranging from 0 to 3 inclusively. 0 represents the leftmost track, whereas 3 represents the rightmost track. The fourth test is checking whether the obstacle can be added to the 1st row (very top row). Since the obstacle will move from top to bottom, it has two integers to record its position.

B. Integration Tests

Because our tasks are relatively independent and each person is responsible for a different feature, the focus of testing is the integration test, where we specify the input and output of each method in order to improve the efficiency of the external parameters required for other members use and call. After checking that the functionality of each part is bug-free, we add one feature at a time to our application and test it every time. The order of integration was as follows:

- Main Logic
- UI Updates
- Player Location Updates
- Music
- Scoring System

C. Exploratory Tests

After making sure that all parts work together without errors, the following values have been tweaked according to the exploratory tests:

- The threshold of the degree acceleration for turning the satellite
- The program frequency
- The frequency of the game with different difficulties (MAX_COUNT)
- The farthest distance of the obstacles from the satellite
- The score threshold for increasing the difficulty of the game

This part is essential since it tightly relates to the user experience. After the α-test by the team members and updating of the data, four more people were invited to the β-test. After the tests, the experience of the players are recorded and all the parameters above were tweaked carefully.

IV. Results

All the unit tests and integration tests were passed easily.

During the exploratory tests, all the parameters above were tweaked carefully. The threshold of the degree acceleration for turning the satellite was set to be 100 without any delay. In the beginning, a delay after turning is detected implemented to avoid overturning the satellite. However, this reduced the fluency of the whole game to a considerable extent. After changing the threshold many times, it is set to be perfect for the players. The refreshing game frequency was set to be 50Hz which caused the refreshing of the map to be too slow, which influenced the fun of the game. After the tests, it is set to be 100Hz. Furthermore, the initial value of MAX_COUNT is set to 3, the farthest distance is set to be 10, and the score threshold for increasing the game's difficulty is set to be 50 and 150 at the end. All of them are obtained from the exploratory tests consisting of α-test and β-test.

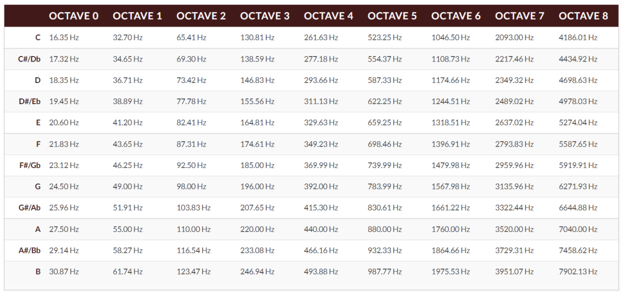
V. Discussion & Summary

After doing more than 70 times of exploratory tests with well-scheduled unit tests and integration tests, the whole project is checked to be bug-free and fun to play. The three UIs are shown in Figure 7, Figure 8 and Figure 9, which explain how to continue the game. The player should hold the board vertically with the USB wire up whenever playing the game. Furthermore, the speaker should be connected as shown in Figure 10 for the music to be played.

VI. Work Distribution

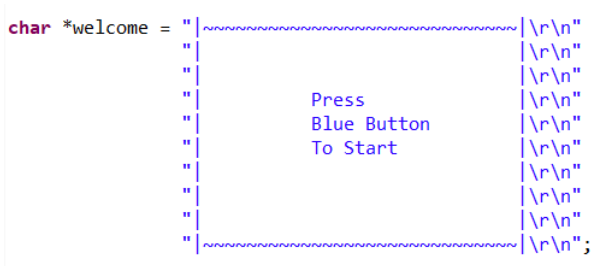
All the members participated in all parts of the project. Ao is mainly responsible for the logical part of the game. Byron is mainly responsible for the I^2C and DAC Speaker implementation. Kevin is mainly responsible for the QSPI Flash Memory implementation. Kua is mainly responsible for the UART connection and UI design. Besides, all members are responsible for the exploratory tests.

Appendix A
Pictures



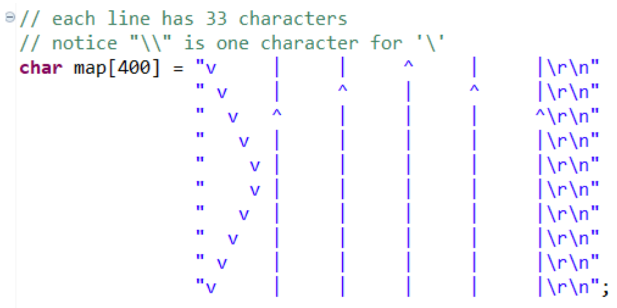
	OCTAVE 0	OCTAVE 1	OCTAVE 2	OCTAVE 3	OCTAVE 4	OCTAVE 5	OCTAVE 6	OCTAVE 7	OCTAVE 8
C	16.35 Hz	32.70 Hz	65.41 Hz	130.81 Hz	261.63 Hz	523.25 Hz	1046.50 Hz	2093.00 Hz	4186.01 Hz
C#/Db	17.32 Hz	34.65 Hz	69.30 Hz	138.59 Hz	277.18 Hz	554.37 Hz	1108.73 Hz	2217.46 Hz	4434.92 Hz
D	18.35 Hz	36.71 Hz	73.42 Hz	146.83 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

Fig. 1. The Frequencies of the Tones [1]



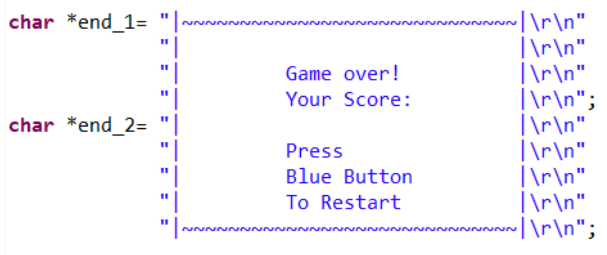
```c
char *welcome = "|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|\r\n"
                "|                             |\r\n"
                "|                             |\r\n"
                "|          Press              |\r\n"
                "|          Blue Button        |\r\n"
                "|          To Start           |\r\n"
                "|                             |\r\n"
                "|                             |\r\n"
                "|                             |\r\n"
                "|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|\r\n";
```

Fig. 2. The Welcome Message



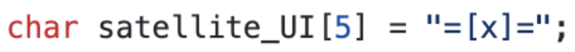
```c
// each line has 33 characters
// notice "\\" is one character for '\'
char map[400] = "v     |     |     ^     |     |\r\n"
                " v    |     ^     |     ^     |\r\n"
                "  v   ^     |     |     |     ^\r\n"
                "   v  |     |     |     |     |\r\n"
                "    v |     |     |     |     |\r\n"
                "    v |     |     |     |     |\r\n"
                "   v  |     |     |     |     |\r\n"
                "  v   |     |     |     |     |\r\n"
                " v    |     |     |     |     |\r\n"
                "v     |     |     |     |     |\r\n";
```

Fig. 3. The Game Map



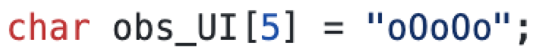
```c
char *end_1= "|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|\r\n"
             "|                             |\r\n"
             "|         Game over!          |\r\n"
             "|         Your Score:         |\r\n";
char *end_2= "|                             |\r\n"
             "|         Press               |\r\n"
             "|         Blue Button         |\r\n"
             "|         To Restart          |\r\n"
             "|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|\r\n";
```

Fig. 4. Game Checkout Message



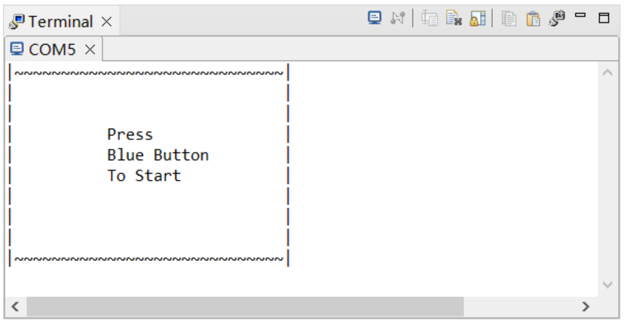
```c
char satellite_UI[5] = "=[x]=";
```

Fig. 5. Satellite Appearance



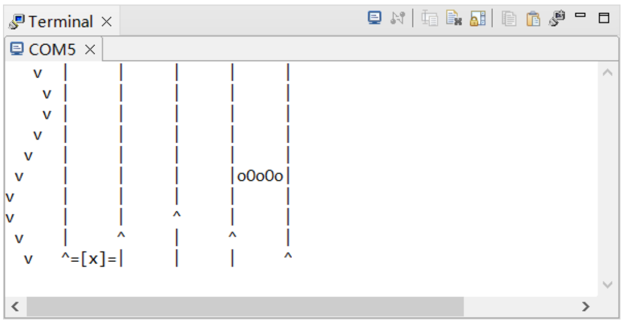
```c
char obs_UI[5] = "o0o0o";
```

Fig. 6. Obstacle Appearance



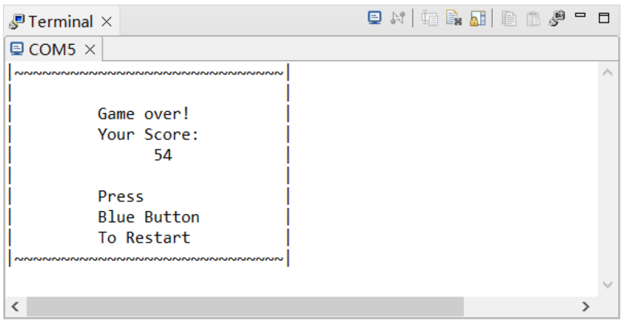
```
|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
|                             |
|                             |
|          Press              |
|          Blue Button        |
|          To Start           |
|                             |
|                             |
|                             |
|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
```

Fig. 7. Welcome Screen



```
   v  |     |     |     |     |
    v |     |     |     |     |
    v |     |     |     |     |
   v  |     |     |     |     |
  v   |     |     |     |     |
 v    |     |     |     |o0o0o|
v     |     |     |     |     |
v     |     |     ^     |     |
 v    |     ^     |     ^     |
  v   ^=[x]=|     |     |     ^
```

Fig. 8. Gaming Screen



```
|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
|                             |
|         Game over!          |
|         Your Score:         |
            54
|                             |
|         Press               |
|         Blue Button         |
|         To Restart          |
|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
```

Fig. 9. Ending Screen

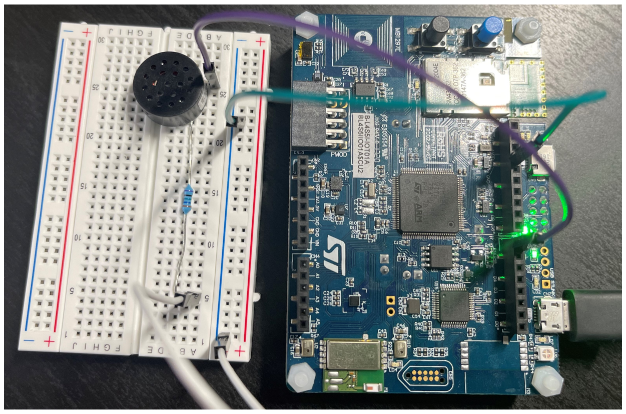
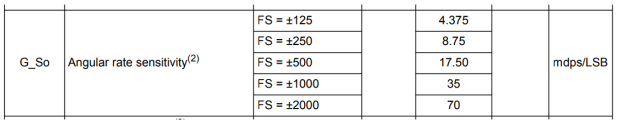
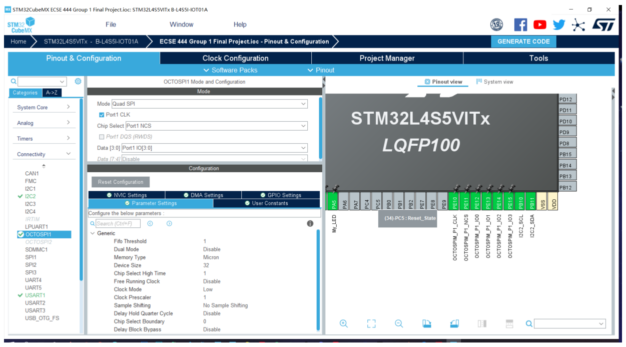


Fig. 10. The Connection of the Board When Outputting DAC Channel 1



G_So	Angular rate sensitivity$^{(2)}$	FS = ±125		4.375		mdps/LSB
		FS = ±250		8.75		
		FS = ±500		17.50		
		FS = ±1000		35		
		FS = ±2000		70		

Fig. 11. Angular Rate Sensitivity of the Gyroscope [2]



OCTOSPI1 Mode and Configuration

Mode

Mode: Quad SPI
Port1 CLK ✓
Chip Select: Port1 NCS
Port1 DQS (RWDS)
Data [3:0]: Port1 IO[3:0]
Data [7:4]: Disable

Configuration

Configure the below parameters :

Generic

```
Fifo Threshold            1
Dual Mode                 Disable
Memory Type               Micron
Device Size               32
Chip Select High Time     1
Free Running Clock        Disable
Clock Mode                Low
Clock Prescaler           1
Sample Shifting           No Sample Shifting
Delay Hold Quarter Cycle  Disable
Chip Select Boundary      0
Delay Block Bypass        Disable
```

Fig. 12. The Setup of QSPI in CubeMX

References

[1] music-note-to-frequency-chart. MixButton. 2021. WebSite: https://mixbutton.com/mixing-articles/music-note-to-frequency-chart/

[2] iNEMO inertial module: always-on 3D accelerometer and 3D gyroscope. STM. LSM6DSL. WebSite: https://www.st.com/resource/en/datasheet/lsm6dsl.pdf

[3] Quad-SPI interface on STM32 microcontrollers and microprocessors. STM. AN4760. WebSite: https://www.st.com/resource/en/application_note/an4760-quadspi-interface-on-stm32-microcontrollers-and-microprocessors–stmicroelectronics.pdf