Team Iota - Final Report

Chengyang Zhong
cz7rd@virginia.edu

Bassam Mohmaud
bm7tu@virginia.edu

Yimin Gao
yg9bq@virginia.edu

Zhenghong Chen
qus9bh@virginia.edu

I. Introduction

Our team made a retro airplane war game. The plane destroys the enemy by firing lasers. We use and expand from the knowledge we have learned in class content, 4 mini projects and finial project part 1 for our final game design. We have optimized the program itself, including using semaphores, adding new threads, optimizing thread management, and designing logic through game mechanisms. At the same time, we also proposed and used new hardware LED. We were going to use the buzzer, but we have not succeeded yet.

II. System Description

In our system, it contains Producer, Consumer, Laser, Restart, CubeRefresh, GameBegin, Display, LEDworker and some CubeThreads. Based on each independent Cube, there is an independent thread. At the same time, based on special threads, we also have some special functions. Joystick and ADC are based on the joystick to control the movement of the aircraft. SW1 and SW2 establish two foreground threads, Laser and Restart, by establishing Button1 and Button2 background threads. Random Generation generates random numbers for easy use by threads. Just like in our program, the number of targets appearing, the specific position they appear, and the direction of movement are randomly determined by random numbers. At the same time, we output through two parts. The first is LCD, which is our main output channel. By controlling the pixels on the screen, we can display what we want to present, although they may come from different threads, different functions, and generated at different times. Secondly, we use LED components for output. RGB LED can display 3 different lights, red, green and blue. Based on the three primary colors, we can match the light we want. At the same time, we can set the display frequency of the LED and transmit information through flashing. Based on the above part, we carried out our game design.

Based on Part1, we have made corresponding modifications to the system thread function and game mechanism. Firstly, The joystick controls a spaceship instead of a simple crosshair. We have modified the size and shape of the control target, and optimized the collision mechanism. At the same time, we changed the icon to use the bitmap function. This makes our control object a red and white aircraft with a fine-looking look.Then enemy aircrafts spawn from the top of the screen. In our design, the target aircraft appears from a random position at the top of the screen. And in the next moment, move in one of the three directions randomly, down, right, left, and, under the action of the algorithm, the target aircraft will not overlap. After that, our spaceship can shoot lasers. By pressing button 1, our aircraft can fire a laser to attack the target. Through the laser we can reduce the number of targets on the screen, and at the same time, we can get score. And life reduction after crashing into an enemy aircraft, or enemy aircraft reaching the bottom of the screen. In our design, each group has 5 health points. There are two situations where our health will decrease. This means that in our design, we have two ways to reduce our life value. First, if our target appears at the top and disappears from the bottom, it means that if we missed it, it will continue to move forward and directly harm us. In our thinking, there is a great possibility that it will increase as the target's moving speed increases. At the same time, as we designed, the target does not move upwards, so the target always approaches downwards. Make it move randomly, reduce the chance of hitting, and increase the difficulty of the game.Regarding game start and game over screen, we have also made corresponding changes. The game cover we added, this is the layout of our game, this is the theme of our game and the name of our aircraft H-351. At the same time, when the game starts, the game ends, and the game restarts, we have all created corresponding pages.

In terms of procedures, we have made a lot of changes based on Part 1, which allows our procedures to be more complete and complete quickly. We have added new semaphores, new threads, and new judgment methods to make them conform to the design of our game. We plan to use more BoosterPack hardware features, such as RGB LED to indicate life and buzzer to play sound effects. In the actual design process, we completed the design of the RGB LED. In this part, we discovered the port driver of the LED and realized it. However, we have not implemented the sound effect based on the buzzer at present, and we need more time to do it.

In the process of playing the game, we can first see the cover of our game, and then we will start our game. In the game, we use the joystick to control the movement of our aircraft and choose a suitable location to destroy the enemy. At the same time, we also need to prevent being hit by other target aircraft. When we are in a suitable position, we press button 1 to launch the laser. When there is a target directly in front of the aircraft, our launch mechanism can destroy the target and get scores. If you are not careful enough to make the enemy hit the plane or start from the top of the screen, go through the entire screen, and slip away from the bottom of the screen, we will lose a bit of health in both cases. The player's health will not only be displayed on the screen, but will also be displayed through LEDs. When the player has 4 or 5 health points, the LED is displayed in green. When the player has 3 points of health, the LED is displayed in yellow.

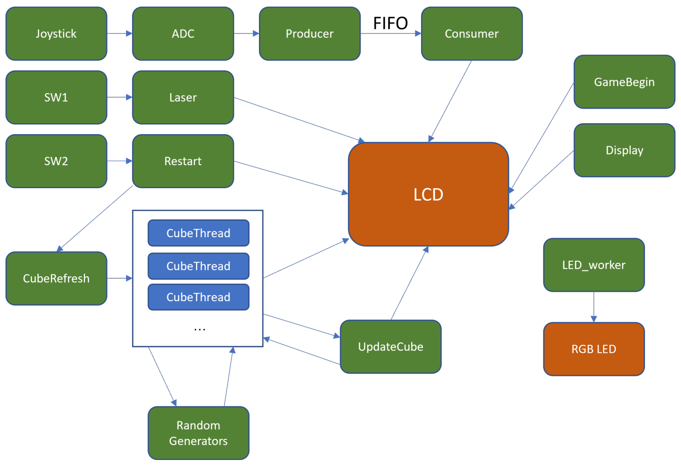

Fig. 1. Data flow diagram

When the player has only 1 or 2 health points, the LED will be red and start to blink. When the player's life value is 0, the system will determine that the game is over, and display the end of the game string on the screen. At this time, the player can press button 2 to restart the game, and the player can press button 2 to restart the game at any time.

III. Design Implementation

A. Game Algorithm

1) Deadlock prevention: To implement deadlock prevention, we defined a new block called BlockArray to store all the occupied locations on the LCD. There are two attributes for each item of the BlockArray: position[2] and available. position[2] has the x-axis and y-axis of the cube and the available indicates if the position is valid. It is similar to the TCB linked list datatype we have in class. If the available is zero, the position is valid that it is occupied by one of the cubes. While if it is one , the position is not valid that any new thread could take this spot of the BlockArray.
We also added another attribute in the cube struct called index which helps to update the location in the BlockArray and check if there will be potential deadlock. Anytime a new cubethread is created, the thread generates from the top of the screen and has a randomized x-axis. It firstly checks if the randomized location is already in the BlockArray and the available is zero at the same time. If it is occupied, the thread keeps randomizing new locations until there is no potential deadlocks. After that, the thread finds the first available spot in the BlockArray and defines its index and set the available in the BlockArray to 0. Then whenever the thread calls the UpdateCube thread, it will be really easy to pass all the parameters by passing the pointer of the cube. The updatecube thread randomizes the direction of the cube(either left, right or down) and gets the next location, then it checks if the next location is already in BlockArray. If yes, then it keeps randomizing and getting new locations until there is no potential deadlocks. Whenever the cubethread is killed, it sets the available of the item in the BlockArray it links to 1 to let other threads use the spot.

2) Laser Thread and Hit Logic: For the laser, the program creates a new thread called laser any time the user pushes button 1. The laser stays on the screen for 15 ms which we used OS_Sleep to achieve. During the 15 ms the laser is valid and we have a global variable called bulletv to indicate if it is valid. After the 15 ms, the thread removes the laser from the screen and kills itself. As for the hit logic, we have it in the cube thread. So each time the cube thread works, it checks if bulletv is one. If yes, then it checks if the laser is in the range of the current cube. If it is in the range of the cube, the thread kills itself and increment the score.

3) Game Flow: We added the game flow to our project. When the program firstly starts, we added a foreground thread called gamebegin which shows the welcome image and the message game begins on the LCD. The message stays on the LCD for 1000 ms and then it kills itself. As for the game over message, we have it in the CubeRefresh thread. So each time the CubeRefresh thread checks the cube number, it also checks if life reaches zero. When the life reaches zero, the thread kills all foreground threads using NumSamples = RUNLENGTH and displays the message game over.

4) Some other changes and configurations based on part1: All enemy crafts appear randomly from the top of the screen and have randomized movement afterwards while they cannot go upwards (either left, right or down). The program also provides 12 x 12 cells for the enemy crafts to move randomly instead of the original 6 x 6. We also have different logic for killing the cube thread (through laser and crush) and decrementing the life (crushing into enemy crafts or enemy crafts escapes from the bottom). Besides that, we also have different images representing the cubes and the crosshair.

B. BoosterPack Hardware Features: Using the RGB LED

1) RGB LED Pinouts and Initialization: The BoosterPack daughter board connects to the main LaunchPad board through the 40 pin GPIO slots, as shown in Fig. 2 [1]. The datasheet of the BoosterPack indicates that the RGB LED uses pin 39, 38, and 37 for the red, green, and blue channels respectively. According to the LaunchPad datasheet, these three pins are eventually mapped to PF3, PB3, and PC4 ports on the micro-controller.

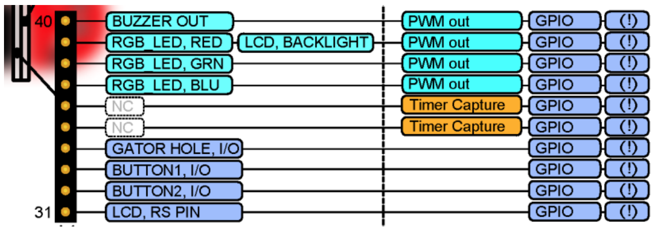

Fig. 2. BoosterPack RGB LED Pinouts [1]

In order to use the RGB LED, these three ports need to be initialized and configured as digital GPIO output. The process follows a strict initialization sequence as part of the LED device driver. Since the RGB LED uses port B, C, and F, clocks need to be activated for these three ports. After the clock is stabilized, the driver unlocks GPIO for port F because this port is locked by default. The driver then disables analog features and configure pins as GPIO output. The alt features on these three ports also need to be disabled, and the driver can finally enable digital I/O.

2) Color Change and Blinking Through a Periodic Thread: In addition to device initialization, the driver also provides several wrapper functions to conveniently set the color of the RGB LED. The color is configured by programming the GPIO data registers of the red, green, and blue channels. Note that the LED also supports PWM signals. However, this project does not implement PWM signals for LED because complex brightness and color transitions are not necessary in the current game design.

The RGB LED is updated through a periodic background thread that runs at 4 Hz. In the current RTOS implementation, OS_AddPeriodicThread() adds a periodic background thread using either Timer1A or Timer4A. Since Timer1A has been occupied by the joystick Producer thread, Timer4A is used to trigger periodic background threads for the LED. During each execution, the function checks the life variable to determine the color. It also toggles between on and off when the life count is very low. The frequency of the timer is only 4 Hz because blinking and color change do not require frequent executions. A slow thread also avoids affecting performance in the main game logic.

C. Sound From Piezo Buzzer

Our team tried to implement sound effects through out the game that indicate space ship hits, losing life, game start, and game end. Unfortunately, we weren't able to make it work with our board. The difficulties in implementations arisen because the code example for the buzzer in the book was using a DAC instead of a PWM signal for the production of the sounds. The other example was given in an Energia code. Energia is an entirely different IDE from Keil and it uses other built in libraries to implement the sounds. Migrating the code from Energia to Keil wasn't easy to do. The only way left to implement the buzzer was to look into the data sheet and figure out exactly which pins to initialize and what functionalities we would need to activate for these pins. We found that the Buzzer is connected to pin PF2 and it has to be connected as GPIO that uses the built in PWM model on the board. The data sheet was very helpful, but, unfortunately, we still weren't able to get it to completely initialize. What were able to do however, is transform the sound files from WAV into frequencies that the the buzzer can produce. We used the provided code in section 15.4 from [2] to convert the sound.

IV. Evaluation and Results

To evaluate the game algorithm, firstly all the threads should not block each other. With round robin scheduler implemented, we did not have such problems. After that, we want to evaluate the basic function of the game including deadlock prevention, hit logic, the game flow function and the restart function. Basically the cubes should not be stuck when they are next to each other with the algorithm we have (no deadlocks), each cube should be killed whenever the laser hits it or the spaceship crushes into it. The game flow should show the game begins page when the game starts at the first time and display the message game over and kill all foreground threads at the same time when life goes to zero. Also, the restart button should be able to restart whenever the game is over or not. The varying conditions were tested several times to ensure that the system will function normally and meet all the expectations. All the conditions here were met for the game algorithm and the game flows fluently without any deadlocks or breakdowns.

TABLE I
Foreground Threads Comparison Table

Thread Name	Functionality	Max Number of Threads	Cooperative
Consumer	Reads Joystick input from FIFO	1	Yes
Display	Updates The LCD with score and life	1	Yes
CubeRefresh	Increases the umber of Cubes	1	Yes
CubeThread	Cube movement and hit logic for each cube	4	No
Laser	Displays the laser for a short period	1	OneShot

V. Lessons Learned

The Insights and Lessons learned from this project mainly revolve around the scheduling method and how we would change it if we were to add more features such as playing sounds or using the accelorometer. Since there isn't a lot of support for RTOS online and we had to figure out how to make things work as efficiently and simple as possible. We decided to go with the Round Robin scheduling for our foreground threads. OS-Suspend() was also used in some of the threads to make them more cooperative.

A. Round Robin scheduling

This method of scheduling gives a specific time slice for each thread to run, 2 milliseconds in our case. The maximum number of threads that can run at the same time is 8 Which effectively allocates a maximum of 16 milliseconds for each round of execution. Due to the efficiency and simplicity of Round the Robin Robin scheduling technique, it was the best method to implement in this project.

B. Other Techniques for Scheduling

In the case of increasing the number of features/threads to the project, the maximum time for a full cycle of execution would increase as well. With the current implementation of the scheduler, the time delay between tasks would proportionally increase with the number of threads added. Therefore, implementing other scheduling methods and techniques would help reduce the delay problem.

1) Priority Scheduling with Aging: One of these techniques we discussed is adding priority and aging variables to each thread. This technique the addition of more threads while also giving all threads a chance to run. It does so by increasing the aging variable of the highest priority thread until it reaches a certain point. Then it puts the highest priority thread to sleep so other threads can run and then repeats this process for other the other threads too.

2) Changing the Current RTOS: The other method we discussed to solve this problem is to use a different RTOS that implements frequency and deadlines for each thread similarly to what we learned in the lectures. This implementation would allow each thread to run on it's own pace based on how important it is to be running at any point of time.

VI. Team Responsibilities

Yimin Gao was responsible for implementing deadlock prevention, adding the game flow, customizing the cube movement from part 1, debugging and optimizing the game logic. Zhenghong Chen was responsible for implementing the Consumer, Laser, CubeRefresh and CubeThread. In addition, he also drew the images for spaceship and enemy crafts, and changed the hit logic. Chenyang Zhong completed the driver bring-up of BoosterPack's RGB LED with handlers of color change and blinking feature. He also helped with debugging and optimizing game logics. Bassam Mohmaud was responsible for implementing the sounds using the Peizo buzzer. He also discussed the overall ideas and themes of the final game and what new techniques we would implement in part 2.

VII. Conclusion

We learned many different things through out the implementation of the space laser game. First, we discussed what the theme of the project is going to be. Then, we brainstormed ways of making the game more interactive while using the different hardware parts on our board, to finally getting everything to work together so we can start testing the game. We went through various themes for the game, but we decided to go for a space ship laser battle style. The cubes were then replaced with enemy spaceships and the cross-hair was replaced with the play space ship that shoots lasers at enemies. Then, we analyzed the different hardware parts on the board that would fit the theme of the game. The two parts that seemed to fit the theme of the project were the RGB LED lights and the Piezo Buzzer. The LED were implemented to indicate the lives a player has by changing colors and blinking, The Buzzer on the other hand, was going to be used to play crashing sounds whenever the laser hit an enemy spaceship. Due to technical difficulties, the Buzzer wasn't fully implemented, so it wasn't included in the project. Finally, we implemented ways to prevent deadlock and used a round robin scheduler that allowed all the threads to have reasonable time slice that didn't cause any noticeable performance issues. Overall, we managed to create a game that follows the requirements of the project. The game ran smoothly and gave satisfying results.

References

[1] *BOOSTXL-EDUMKII Educational BoosterPack Plug-in Module Mark II User's Guide*, Texas Instrument, Texas Instruments, Post Office Box 655303, Dallas, Texas 75265, 2017.

[2] J. Vaälvano, *Embedded Systems - Shape The World*. https://users.ece.utexas.edu/ valvano/Volume1/E-Book/, 2012.