ECE 6501: Final Project Report

Nayiri Krzysztofowicz, Vivian Lin, Deke Lueker, Nojan Sheybani

I. Introduction

The project that Team Iota created, called 'Green Eggs and Homa,' consists of modifications to the original cube game that was completed in Part 1 of this assignment. New features were added to the game to make it more complex and more enjoyable. Some of the features added to the project had similar implementations to the mini projects, while others were not covered specifically in the Advanced Embedded course. The project was successful in creating a more enjoyable experience than the original cube game.

II. System Description

A. Project Goals

Green Eggs and Homa initially promised four new features:

- Start menu
- Pushbutton to 'hit' blocks
- Block color affects properties
- Beep on a 'hit'

The game delivered all of the features shown above, as well as some additional features. The extra features serve to give the game a unique personality and make it a more memorable experience. The additional features can be seen below:

- Background music
- Levels
- Game over screen
- High score
- Bitmap implementation

Some of the added features tie in closely with a previously promised feature, such as the background music and the buzzer. This allowed features to be added that drastically changed the experience of the game while not requiring large amounts of new code to function properly. Other features, such as the bitmaps, did not relate to anything that was promised. Features such as this were added simply because the team had extra time and thought it would set their game apart and make it a memorable experience.

B. Game Structure

Green Eggs and Homa still plays like the original cube game, with some minor changes. It should be thought of as an iteration on that game instead of a new game. The first difference is that there is a start screen in Green Eggs and Homa. This allows players to change settings to tailor the game to their liking. It also gives players the ability to prepare themselves before starting the game. This is important because blocks disappear after a certain amount of time and decrease a player's life when this happens.

Once the player starts the game, some of the larger changes become apparent. The crosshair has changed from the original cube game. Now, it is represented by a bitmap image of ham. The blocks have also changed from colored rectangles to eggs with varying yolk colors. The color of the yolk affects the value of the egg, with green eggs being worth 5 points while the other eggs are worth 1 point. This rule is shown to the player at the start of the game in the menu. The player's current score and life are still shown in the bottom segment of the LCD screen. As the player's score increases, the game will change what level the player is on. To notify the player, the new level is briefly displayed on the bottom of the LCD in a bright font. As the level increases, the 'lifetime' of the cube decreases, which increases the difficulty and makes it more likely the player will miss some cubes. This process ensures the game will end in a finite amount of time.

While the game is playing, there is music playing in the background. The volume of the music can be changed in the start menu, and it can be disabled altogether by setting volume to zero. Whenever the player successfully hits a cube, a 'beep' sound is produced that helps notify the player of a successful hit. The note played for the hit notification is distinct from those in the music so the player does not confuse the sounds. When the game ends, all of the cubes disappear and a game over screen is displayed. The screen shows the player what score they reached, as well as the highscore for that session. From the game over screen, the restart button can be used to take the player back to the start menu. The data flow is shown in Figure 1, and overall program structure in Figure 2.

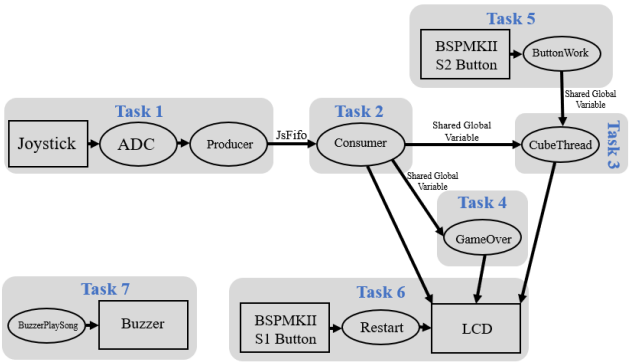

Fig. 1. Dataflow in the game, split by thread.

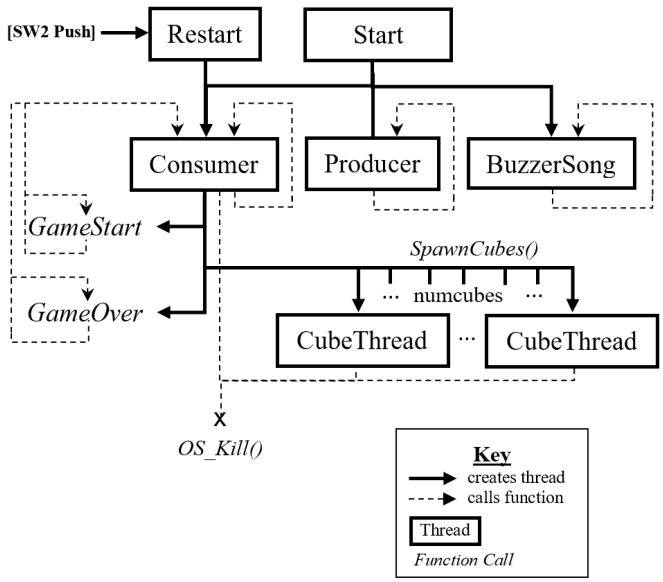

Fig. 2. Program structure, showing all relevant threads and function calls.

III. Design and Implementation

A. Start Menu

This menu was implemented by running the GameStart() function. This function has 2 screens: a rules screen and a settings screen. The user sees the rules screen first and can progress to the settings screen with a button hit. The setting offers users the option to increase their number of lives, starting level, and volume of music. In order to implement this, we made a function called BSP_LCD_DrawArrow() in order to increase or decrease the default settings. We implemented bounds and treated the arrows as a "matrix" with axes i and j to be able to move to an arrow relative to the current position. The user's current position is tracked using the i and j indices, which are incremented or decremented when the joystick moves. The position is also highlighted bright green so that the user always knows where he/she is. Once on an arrow, button B1 can be pushed in order to perform the desired setting change. Once the user has moved down to the start option and pressed B1, the game starts.

B. Button Hits

The idea of having the cursor move over the blocks to get points seemed a bit too easy, as a user could just blindly move the cursor around the screen and still get points. We realized that Button 1 was already configured to add a background thread from our previous mini projects, so we decided to reconfigure it to hit blocks. On every button hit, the coordinates of the cursor are checked. If the cursor's coordinates correspond to the coordinates of a cube on the screen, that cube's thread is killed the score is incremented with the respective amount of points.

C. Adding Sound

The source of sound on the Educational Boosterpack MK 11 was a magnetic buzzer controlled by Port F, pin 2 on the TM4C microcontroller. PF2 is a GPIO pin, with a special PWM functionality that would set the digital output signal to have a pre-set period and duty cycle. The pin was set to PWM count-down mode, in which the internal microcontroller timer would start at the value loaded into the load register and decrement it until reaching zero, whereby load would be reset to its original value. When LOAD reached the value in the cmpA register, the output would be flipped. Thus the load register set the period (note) of the output signal, and the cmpA register set the duty cycle (volume). When the pin voltage was driven low, the buzzer would output sound. The PWM is summarized in Figure 3. To program the background song, we first found the notes to the song online, then translated these to period values. These values were passed into the PWM pin for each note of the song. The song was in a separate thread that was in an infinite loop and never killed, calling OS_Sleep() after setting the period of each note to play the note for a set amount of time.

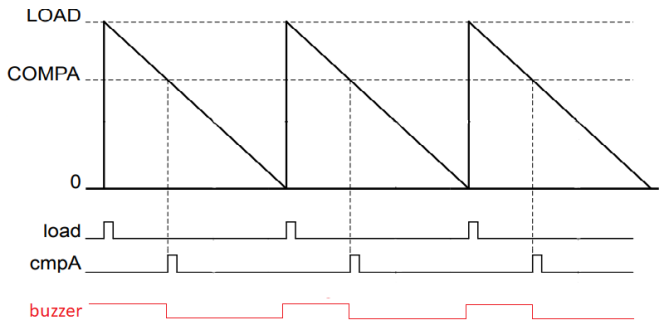

Fig. 3. PWM count-down mode on TM4C, with load and cmpA values shown and resulting pin output controlling the buzzer (red).

D. Levels and Block Points

Without having levels, the only sense of progress was having a high score. We decided to add levels so that the user could see their score translated to actual progress throughout the game, such as a high level. Adding levels also allowed us to implement increasing levels of difficulty as the user progresses through the game. We also decided to make green eggs worth 5 points instead of the normal 1 point for other eggs. This was done by adding a points field to our Cube struct. Every time a cube is hit, the cube's points are added to the score right before the cube's thread is killed. Every 15 points the user scores, the level is increased and all the cubes' lifetimes are decreased by 1000 ms. The default starting lifetime is 10000 ms, but the user can choose to start at a higher level, which would result in a lower starting lifetime. The following equations are used for calculating the level and starting lifetime for a cube:

$$Level = StartingLevel + Score/15$$
$$Lifetime = StartingLifetime - Level * 1000$$

The lifetime is lower bounded at 3000 milliseconds so that it is always possible for the user to get points. Every time the level increases, a green level indicator briefly flashes in the bottom left corner where the score is usually shown.

E. Deadlock Prevention

Since our game used multithreading to control the blocks, deadlocks between two blocks would occur when each block would attempt to move to the block occupied by the other block. This scenario is illustrated in Figure 4.

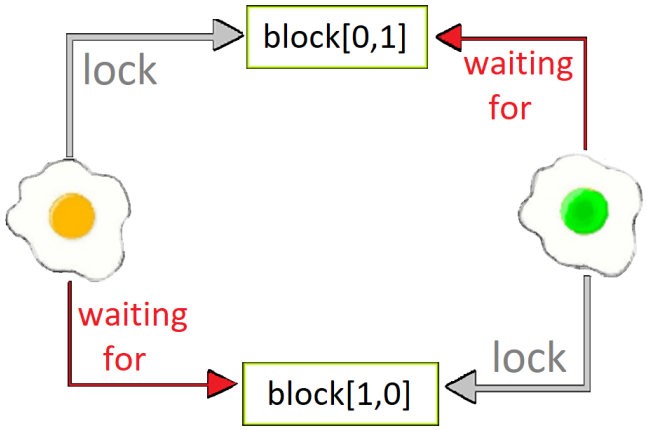

Fig. 4. The deadlock issue.

Our solution to prevent deadlocks was simple; when a block attempted to move to a block that was already occupied, we would re-generate a direction so that the block could keep moving. The deadlock can be seen in Figure and the prevention process is summarized in Figure 5.

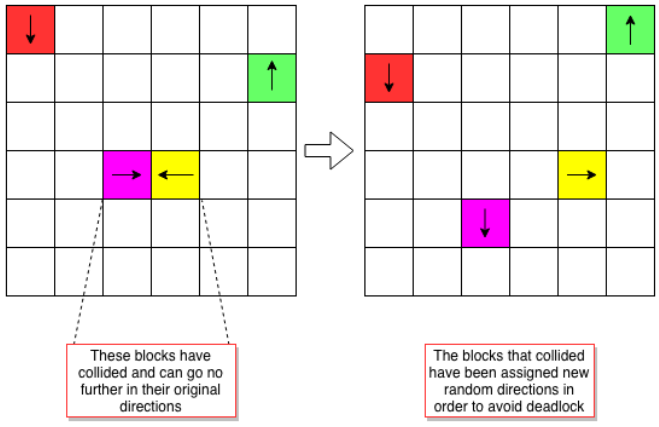

Fig. 5. The issue with deadlocks when moving the multithreaded cubes.

F. Bitmaps

The microcontroller displayed bitmaps on the LCD by sending the color of each pixel in the image. The LCD displayed 16-bit colors, so each pixel required sending two bytes of data. We converted images to their char array equivalent online, and for each pixel sent the next two elements of the array. To eliminate the undesirable background from the ham cursor, the BSP_LCD_DrawBitmap() function was redefined in a new function, BSP_LCD_DrawHam(), to send only non-zero (non-black) pixels. This was done by redefining the address window with SetAddrWindow(). In order to erase the ham cursor, the BSP_LCD_DrawHam() function is also able to draw it in all black by XORing the color value with itself.

G. Game Over

The Game Over screen is implemented as a thread that is added from the Consumer thread when the player has run out of lives. The screen shows the player's score, calculates the new high score if it has changed, and displays the high score. While this thread is running, all other threads except the Buzzer_Play_Song() are killed. If the player presses B2, then the GameOver thread is killed and the Restart thread is added to restart the game.

IV. Evaluation/Results

The main way we evaluated our game was through playing it. With our game, it was pretty obvious when something we thought we implemented was not working. For instance, when we hit the button over a block and it does not disappear, we know that something is wrong. This was different from how we have been taught to test throughout the course as we were not worried as much about the standard RTOS metrics, such as latency, as we were using a dynamic priority scheduler with aging, which we deduced would be the best scheduler for our performance based off the conclusions we drew in Mini Project 4. Instead of focusing on timing metrics, we made a list of all the things we should be checking for, either in the game or in the watch window, and observing throughout each run of the game:

1) Start menu plays "Mo Bamba" and "I Love It" plays throughout gameplay
2) Life, Volume, and Score increase/decrease with button push on respective arrow on start menu
3) Starting Life, Volume, and Score correspond to what user input on start menu
4) Button push while cursor on cube make buzzing sound
5) Button push while cursor on cube kills cube's thread (cube disappears) and score is properly increased
6) Every 15 points, the starting lifetime of cubes is decreased by 1000 ms
7) Every 15 points, the level is increased by 1
8) The starting lifetime never goes below 3000 ms
9) Green eggs add 5 points to the score, yellow eggs add 1 point
10) Restart button is deactivated on start menu
11) Restart button takes user back to start menu during gameplay or on game over screen
12) Game over screen shows user's last score and highest score of the current session
13) Basic LCD glitches when drawing and erasing bitmaps

V. Lessons Learned

A. Memory Management

Implementing the bitmaps immediately added a large amount of data to our program, causing us to exceed the 32 kilobyte memory limit imposed by Keil on programming the microcontroller. Each $n \times n$ image required $2n^2$ bytes of memory, so we began reducing the memory usage of our program by making all bitmaps smaller in dimension. We then optimized the code as much as possible, removing extraneous functions, imported files, and global variables to allow the bitmaps to be included. Table I shows increases in memory usage for our program, both in the form of code (from adding functionality to the game) and read-only (RO) data (from the bitmaps).

TABLE I
Memory Usage

(bytes)	Part 1	Part 2
Code	13,906	17,086
RO-data	1438	9750
RW-data	272	436
ZI-data	10,904	10972
Total: C+RO+RW	15,616	27,272

B. Debugging a Game

While working on this project, it became clear how difficult debugging games can be. While developing the code for this project, many bugs popped up. It was often very apparent when there were problems due to the graphics of the game, but it was not always easy to find out what was causing the bug. This is mostly due to how we can recognize once a bug has happened, but there is no way to rewind the program. Due to the number of possible game states and the difficulty in controlling the game while running the debugging software, it was almost impossible to pause the code right before a bug happened. This meant that most of the debugging that we had to do was largely without the use of the debugging software.

An example of a bug that was in our game was a block would not correctly disappear at certain times. This issue exclusively happened when the cursor was over the block. The block would not disappear from its original position when all the blocks moved, instead an image of the block would stay there while the cube thread would draw another block adjacent to the original position. Because the old position is not associated with a thread, it cannot be hit and therefore it just stayed in its position until another block moved into that space. Without a great approach to debug this issue, we had to identify all the functions responsible for drawing and erasing a cube, and then think of all of the edge cases because we could not debug right before the issue happened. This approach proved to be much more difficult and time consuming than debugging code during the previous mini-projects. This experience leads to the question of what tools modern game developers have to identify and recreate situations that cause bugs in their projects.

C. Threads Add Complication

Throughout the game-development process, we grew to understand the extent of complications that arise when writing a multi-threaded program. The non-deterministic behavior of the threads made debugging extremely difficult, as we would have to run the game many times before seeing a particular bug and figuring out its causes. Also, the shared data meant the code had more overhead, both in implementation and memory usage to ensure proper unlocking and locking of shared resources. Finally, we struggled with thread synchronization. For example, the Restart function required killing all cube threads to return the user to the menu screen. Since there is no function to kill a particular thread, we would set a restart flag and call OS_Sleep($time$), waiting $time$ milliseconds so that each thread could be scheduled, see the restart flag set, and be killed before proceeding with Restart. This process is summarized in Figure 6. The alternative implementation to the multi-threaded cubes was to have them be "event-driven," and all run in one continuously-looping thread. This would eliminate both deadlock and synchronization problems.

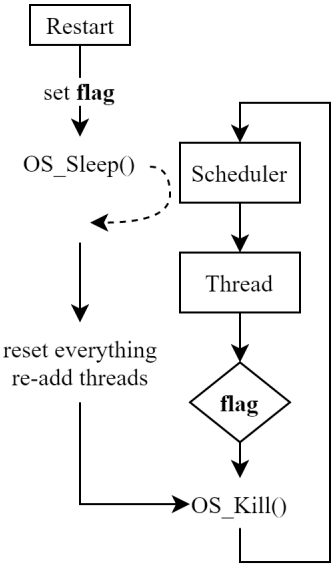

Fig. 6. Thread synchronization with Restart thread.

VI. Team Responsibilities

A. Nayiri

Nayiri worked on setting up the cube threads and some of the deadlock prevention. She also configured the buzzer and set up the background song, as well as process for importing the bitmaps.

B. Vivian

Vivian worked on detecting cube hits, implementing the second pushbutton with Deke, and redefining hit point locations

for the new cursor. Vivian also helped Nayiri redefine how bitmaps and strings are drawn. Lastly, Vivian helped Deke and Nojan improve the start screen.

C. Deke

Deke worked on implementing the second pushbutton into the game and using it to hit the cubes. This section was also later worked on by Vivian. Deke then worked on how to add different properties to different colored blocks. Lastly, Deke helped Nojan create an initial start screen. The start screen was later further improved by Vivian.

D. Nojan

Nojan worked on a some of the deadlock prevention and making levels to the game (decreasing blocks lifetimes). Nojan then worked on the start menu with Deke. The start menu was further improved by other team members. Finally, Nojan worked on the game over screen and added the ability to store a high score and display it on the game over screen.

VII. Conclusion

This project provided the opportunity to apply the real-time operating system developed throughout the semester to a final product: Green Eggs and Homa. After implementing the basic functionality of the game in Part One, Team Iota focused on the game design, adding new features around a Green Eggs and Ham theme. While creating Green Eggs and Homa, Team Iota faced many road blocks, such as the complexity of multithreading, the technical details of interfacing with low-level drivers, and the real-world constraints of memory usage. Through these, the team members learned both how to utilize an RTOS effectively and how to interface with the hardware efficiently. Finally, Team Iota was able to apply these skills and successfully create an enjoyable game.

Acknowledgment

The authors would like to thank Professor Alemzadeh teaching them multi-threaded RTOS programming and serving as inspiration for the final project. They would also like to acknowledge the teaching assistants of the course, especially Mustafa Hotaki and Ailec Wu, who proved helpful during office hours thoughout the course.

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

[2] Valvano, J. W. (2016). *Embedded Systems Playground*. Retrieved December 17, 2018, from https://github.com/glennlopez/EmbeddedSystems.Playground/blob/master/01\%20-\%20Independent\%20Study/UTAustinX_1201_RTOS/TM4C123Valvanoware/inc/BSP.c