Team Gamma – Final Report

William Trace LaCour
Justin Vaughn Javier
Jon Gustafson
Josh Neal

I. Introduction

The team set out to produce a game in the style of a retro arcade shooter. Using the tools and implementations acquired over the course of the semester the team developed the game as a set of threads which were scheduled carefully to generate a game that is reminiscent of the game Galaga. Special focus was paid on the usage of multiple threads to learn about and gain experience with some of the lessons learned in this course specifically. Race conditions and memory management lessons were also explored as the team began to debug and explore alternate implementations.

II. System Description

The final project consisted of the implementation of a retro-style arcade shooter resembling Galaga or Space Invaders. Random enemies were created at the top of the screen and printed as one of four enemy bitmaps. A player ship was also created in the place of the crosshair from the previous project. The player ship was also a bitmap which was created using sprites from the original game. The ship has mobility over the bottom portion of the screen which is similar to the original game's implementation.

Player shots are threads created through the generation of an interrupt routine. The projectile created by these interrupt routines lives until the shot strikes an enemy or goes off the screen. If the shot strikes an enemy the enemy is set to inactive and the enemy is erased. When it strikes an enemy, the projectile is also erased and the shot thread is killed. If an enemy reaches the bottom of the screen without being hit, the player's life count is decremented and the enemy is erased. If the enemy strikes the player ship the game is ended immediately.

The main menu was implemented to improve user-friendliness and to allow general user input. The main menu was implemented as a thread which existed until the user entered into the game. At the time the game started the menu thread was killed. From the menu, the user could also hit button 1 and display a screen showing the programmer's names. When button 2 was hit, the game began by erasing the menu and handing control of a ship over to the player. The game would also initiate the necessary tasks needed to generate enemy ships, initialize the player's life count to three and set the player's score to zero. At this point, the game became fully functional and the player could begin playing the game. From this stage, the game would continue until a game over condition was triggered. This game over condition must be triggered either through a player's life count being reduced to zero or the player ship colliding with an enemy ship.

III. Design Implementation

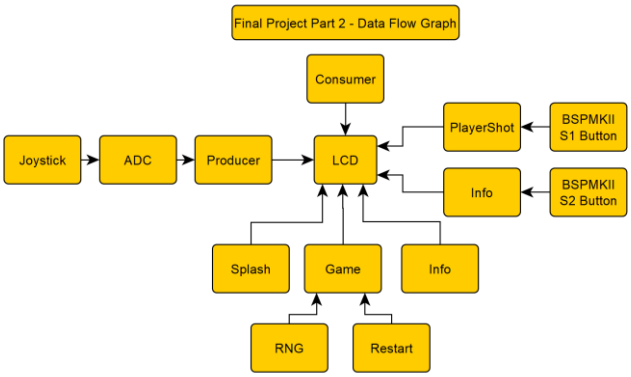
This project used several threads to implement a variation on the classic Galaga game. These threads were scheduled and run through a priority scheduler with aging. This particular scheduling scheme was utilized because it allowed for the use of priority preemption with some capacity for starvation prevention. It was hypothesized that these attributes would be beneficial for a game's operating system because games rely on external inputs and many times need to respond to these inputs very quickly. While the quick response is necessary, it is also necessary for games to avoid starvation. Even if a user hits many buttons or inputs quickly, it is still necessary for other game elements to continue executing in a timely manner to obtain optimal user satisfaction. In addition to the use of several threads, the system also employed a data structure to keep track of all the enemies currently on the screen. In the current game implementation this data structure would store information for eight individual enemies.

When the game was initially launched, a thread was run to display the start page. This start page displayed a Galaga bitmap on the LCD screen and allowed the user to hit one of the two buttons located on the launchpad. Pressing button 2 would cause the system to launch the consumer and game thread. The consumer thread would check the FIFO to determine if the producer thread had stored any new data. If new data was available, the consumer thread would pop that data out of the FIFO and update the player controlled ship's position. After the ship's position was updated, a collision detection routine was run. This collision detection algorithm would linearly search through the data structure of enemies. If an enemy was active on the screen, the system would calculate the absolute value in the difference between that enemy's pixel coordinates and the player controlled ship's pixel coordinates. If the horizontal coordinates were within 14 pixels of each other and the vertical coordinates were within 14 pixels of each other then the system would register a collision. If a collision were detected, a game over event would be triggered by killing the active threads, resetting all global variables, and re-launching the thread that displays the menu page. If no collision were detected, the consumer would suspend itself to cooperatively allow other threads to run.

Next, the game thread would run. This thread carried out most of the processing required by the game, but its primary responsibility was to control the motion and creation of new enemies. Every time the thread would run, it would move all the currently active enemies down the LCD screen towards the player. After this motion was completed, the uppermost row of the LCD screen was empty and would no longer have a new enemy. To keep the stream of enemies continuous, the system would randomly pick a column of the LCD screen that does not currently have an enemy in it and would create a new enemy of a random ship type and assign it to this column. This enemy would also be added to the active list of enemies so that it could be moved the next time this thread is executed. The game thread would also determine if an enemy had reached the bottom of the screen. If an enemy had made it this far, then the player's life is decremented by one and the enemy is deactivated. This thread would also check the player's life value and if the value hit zero, initiate the procedure needed to restart the game.

Another type of thread that would be run in the system was the interrupt routine triggered by pressing button 2. This interrupt routine would then spin up a "PlayerShot" thread. This thread had the highest priority in the system and would create a projectile object on the LCD screen. It would also

increment the projectile forward on the LCD every time this thread was run. After the location of the projectile had been updated, a hit detection routine was run. This routine iterated through all active enemies and determined if any of the projectile's hitbox lied inside the enemy's hitbox. If this overlap was detected, the enemy would be marked as inactive, erased from the screen, the player score would increase, and the projectile thread would kill itself.



IV. Evaluation and Results

The team was successful in implementing the Galaga style game using multi-threading. To verify that the game and operating system were functioning as intended, the game was played multiple times. To thoroughly test the system, these games played involved with various death types and scores. These varying conditions were tested to insure that the system would function properly under a variety of edge cases. Specifically, the collision of the player ship with enemies was tested in multiple directions including diagonal collisions. Player shot thread successfully interacted with enemies and was killed as expected when a collision with an enemy or the edge of the screen is reached.

V. Lessons Learned

Many different lessons were learned through the process of creating this game. These lessons were both technical and practical in nature. From a technical standpoint, the team realized that it is important to consider the memory limitations of embedded systems. During the creation of this project, a similar technical limitation was discovered. Since the version of Keil used in this class was a free trial version, it artificially restricts the size of programs and memory that can be programmed into the microcontroller. With the version of Keil used this semester, the limitation imposed by Keil was 32 kilobytes. While this limitation was not entirely synonymous with the limitations associated with a chip itself, it had very similar implications on way the program was constructed. For example in this project, the bitmaps used took up a large amount of space which meant that other parts of the program needed to be reduced or less bitmaps had to be used. This design consideration convinced the team to leave out a classic game mode that would have involved the use of more of these bitmaps.

Another difficulty that the project presented was debugging the code. As previously mentioned, the nature of Real-Time Operating systems allows for the possible presence of race conditions. The fact that the race-condition cannot be predictably recreated similar to conventional sequential programming was a difficulty that the team had to deal with because merely setting a breakpoint does not isolate the error.

An example of a race condition found during this project was caused by an unregulated context switch between the restart and consumer threads. The restart thread had the task of clearing the screen and printing the main menu. Occasionally, the system would clear the screen, but score and life information were printed at the bottom of the menu screen. This erroneous behavior was likely due to a context switch from the reset thread to the consumer thread. The system began to reset, which cleared the screen, but before the reset thread could finish its tasks needed to reset

the game, the scheduler performed a context switch to the consumer thread. The consumer thread would execute and print the score and life information to the screen before another context switch was performed. This second context switch continued execution of the restart thread and finished the restart procedure. By allowing the code to be executed in this sequence of events, the system allowed an incorrect menu screen to be printed. To rectify this race condition, a semaphore was added to the beginning and the ending of the reset and consumer threads. This semaphore prevented consumer from running when restart had already begun and also prevented restart from running until consumer concluded its tasks. By synchronizing threads in this manner, the race condition was removed and the program executed as expected.

The process of debugging this race condition taught the team many lessons. One lesson the team learned, was why thread synchronization is important. Even in higher level programming languages, there are some applications that require the use of a multithreaded approach. By controlling thread execution at a fundamental level in this project, the team members are now much more capable of not only implementing an RTOS, but also of working on other multithreaded applications.

A practical lessons learned were project management techniques. Logistics was relatively complicated when the team worked remotely from each other. This would cause either redundant effort or code development from different members. The use of Github required careful communication and timing between team members to make sure that a push to the repository does not write over unintegrated code from a previous push from a different team member. The team found a collaboration software called TeamViewer that allowed members to remote access to another personal computer. Using this software the team collaborated and developed code remotely on one member's computer. This prevented any redundant code development.

VI. TEAM RESPONSIBILITIES

Jon Gustafson was responsible for developing the hit detection and collision detection needed between the player's spaceship, the enemy spaceship, and the projectile. Jon was also responsible for the debugging of the code. Justin Vaughn Javier was also involved in debugging as well as writing the project report. William Trace LaCour was responsible for the movement mechanics of the ships and projectiles. Josh Neal was primarily responsible for collating and implementing the bitmaps. Josh was also responsible for implementing the program user interface.

VII. CONCLUSION

The process of creating a game on a microcontroller running an RTOS was a very educational one. Through the process of seeing different semaphores, scheduling methodologies, time slices, and thread synchronization techniques, many useful things were learned that will be beneficial not only in embedded development, but in all forms of software development. In addition to being an enriching experience, the process of working on a game from conception to finished implementation was very rewarding. Working on this project also made the team members better debuggers because they not only successfully debugged their own code, but also code written by other individuals.

It was also concluded that the game met all the goals of its design. While the game was slightly

different than the original Galaga, it was still true to the spirit of the game and shared many of the same mechanics. It was also confirmed that the team members understood the use and application of an RTOS. Without having an in-depth understanding of an RTOS and its functionality, it would have been very difficult to successfully implement a Galaga style game on an embedded system.

REFERENCES

[1] "lcd-image-converter," SourceForge. [Online]. Available:https://sourceforge.net/projects/lcd-image-converter/. [Accessed: 19-Dec-2018].

[2] "TeamViewer – Remote Support, Remote Access, Service Desk, Online Collaboration and Meetings," TeamViewer. [Online]. Available: https://www.teamviewer.com/en-us/. [Accessed: 19-Dec-2018].

[3] "Arcade - Galaga - Galaga - The Spriters Resource."[Online].Available: https://www.spriters-resource.com/arcade/galaga/sheet/26482/. [Accessed: 19-Dec-2018].