ECE 4501: Advanced Embedded Computing Systems Final Project Report

John Berberian (ccg3sr), Sarah Hemler (seh5pp),
Paul Karhnak (zxz2hm), and Casey Ladd (jdm8wk)

I. Introduction: An Overview of the Project

The project served as a demonstration of the capabilities of a Texas Instruments (TI) Tiva TM4C123GH6PM microcontroller unit with an attached Educational BoosterPack MKII header board. The TM4C123GH6PM ("Tiva") itself ran a lightweight real-time operating system (RTOS) which coordinate threads and interfaced with peripherals like an LCD screen, pushbutton general-purpose inputs, and a buzzer, many of which were located on the BoosterPack MKII board. A small game based on the classic video game *DOOM* was developed to run on the board, including the emulation of music from *DOOM* and the creation of similar sprites inspired by the game. Even if the concept was at first a jocular reference to video game modding culture (Craddock [1]), the project evolved to be an earnest demonstration of the question: "Can it run *DOOM*?"

II. Team Responsibilities

John Berberian (ccg3sr) worked together with Sarah Hemler (seh5pp) to implement much of the game's graphical capability; furthermore, Berberian developed the deadlock prevention scheme present in the final game. He also developed most of the gameplay elements and startup sequence. Hemler was overall responsible for managing the group and coordinating efforts where needed; credit for the game idea also lies with Hemler. Paul Karhnak (zxz2hm) wrote the random number generator, attempted some early work with encoding bitmap images, put together an early version of the deadlock prevention measures, and contributed to documentation. Casey Ladd (jdm8wk) spearheaded the development of the game's interactive sound effects through the microcontroller's pulse width modulation (PWM) capabilities.

III. Design and Implementation

A. Graphic Design

As an improvement upon the basic cube game design, we decided to use animated sprites in our game. We created 16 x 16 pixel "blocks" to logically divide the liquid crystal display (LCD) graphical output device deployed on the microcontroller, and then used FireAlpaca and Paint.NET to draw up a bitmap file of an array of 16×16 pixel images. The Tiva BoosterPack add-on can render graphics in 16-bit "high color"- red is defined in 5 bits, green is defined in 6 bits, and blue is defined in 5 bits. The original color palette of DOOM conforms to this restriction, so we used that palette for our work. Once everything was drawn, we used a python script, created by John Berberian, to convert the bitmap file into a readable bitmap array. We then defined that array in our MOOD_graphics.h file.

```
/* begin images/l_4_particles.bmp */
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x4000, 0x0000,
0x4000, 0x4000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x4000, 0x6000, 0x5800, 0x5800,
0x5800, 0x5800, 0x0000, 0x4000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x4000, 0x0000, 0x4000, 0x6000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x5800, 0x4000, 0x4000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x7000, 0x7000,
0x0000, 0x7000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x6000, 0x0000, 0x0000, 0x7000, 0x8800, 0x0000,
0x7800, 0x0000, 0x7000, 0x8800, 0x0000, 0x7000, 0x4000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x7000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x7000, 0x0000, 0x0000, 0x7000, 0x4000,
0x4000, 0x0000, 0x8800, 0x0000, 0x7800, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x4000, 0x0000, 0x5800, 0x7000, 0x0000,
0x4000, 0x0000, 0x8800, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x8800, 0x0000, 0x0000, 0x4000,
0x0000, 0x6000, 0x0000, 0x7000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x7000, 0x0000,
0x0000, 0x0000, 0x4000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x7000, 0x8800, 0x0000, 0x0000, 0x4000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x7000, 0x0000, 0x0000,
0x0000, 0x0000, 0x4000, 0x7000, 0x0000, 0x7000, 0x0000, 0x0000,
0x0000, 0x0000, 0x6000, 0x0000, 0x7000, 0x0000, 0x4000, 0x4000,
0x0000, 0x8800, 0x7000, 0x0000, 0x0000, 0x6000, 0x4000, 0x0000,
0x0000, 0x4000, 0x8800, 0x6000, 0x0000, 0x5800, 0x0000, 0x0000,
0x7000, 0x7000, 0x8800, 0x0000, 0x7000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x4000, 0x0000, 0x6000, 0x0000, 0x8800, 0x8800,
0x0000, 0x0000, 0x0000, 0x4000, 0x7000, 0x4000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x4000, 0x0000, 0x0000, 0x0000,
0x0000, 0x4000, 0x0000, 0x0000, 0x4000, 0x0000, 0x0000, 0x0000,
/* end images/l_4_particles.bmp */
```

Fig. 1. A section of our bitmap array representing a single sprite.

In addition to our bitmaps, our display also included text. This included instructions and information on splash screens at the beginning and end of the game, and a readout of lives, ammo, and score that appeared during gameplay and on the game over screen. The score counter locks when the game ends, allowing the player to see how well they performed. All of these were created using the BSP_LCD_Drawstring command, on a separate rendering thread.

In order to reduce the complexity of the semaphore systems involved in the game, we chose to dedicate a thread (the "renderer" thread) to updating the state of the LCD screen. We made a synchronized MPSC FIFO for serializing the state updates - threads would push small event objects to the FIFO, describing an update that needs to be made to the screen (render sprite at location, erase sprite at location, display title screen, update ammo and life text, etc.). The renderer thread had only one job: to dequeue events from that FIFO and

Fig. 2. Sprites used in Mood

render them to the screen using the BSP_LCD commands provided. This prevented the screen state from being corrupted by synchronizing the writes from multiple threads, and without the possibility of complex deadlocking bugs. The crosshair was drawn by the Consumer thread, and access to the LCD between the two threads was controlled by a semaphore, LCDFree. We considered moving crosshair rendering to the renderer thread to simplify the code further, but ran out of time.

B. Random Number Generator

An xorshift pseudorandom number generator (PRNG, referred to by simply "RNG") as proposed by Marsaglia [2] was implemented in this project. The xorshift RNG was selected for its efficiency and ease of implementation: featuring computationally efficient operations like bit shifts, XORs, and moves, the xorshift generator was ideal for deployment within "cube" (demon) threads which had to run as quickly as possible within the RTOS.

Fundamentally, the xorshift generator simply applied the shift and XOR operations to a seed in a three-line algorithm:

```
CurrentRandomValue ^=
    (CurrentRandomValue << 5);
CurrentRandomValue ^=
    (CurrentRandomValue >> 27);
CurrentRandomValue ^=
    (CurrentRandomValue << 25);

return CurrentRandomValue;
```

The shift amounts 5, 27, and 25 form a triple of numbers proposed in Marsaglia [2] for sufficient randomness. The seeding itself is done by calling the kernel's previously implemented `OS_Time()` routine which reads out the value of a hardware timer. The `OS_Time()` call is done within the game, however, and not strictly on startup; thus, seeding the random number generator is in and of itself a random process that is conducive to proper pseudorandom number generation. Furthermore, the seeding portion of the random number generator ensures that the seed is nonzero by polling the return value of `OS_Time()` until the return value is nonzero. A nonzero seed ensures the ability of the RNG to properly produce a sequence of random numbers.

The random number generation was used on a per-thread level to direct the threads controlling game sprites to "move" in certain directions around the board. Furthermore, the RNG was deployed as part of a deadlock prevention strategy when threads needed to be synchronized in their use of the common LCD resource.

C. Interactive Sound Effects

In general, three types of sound effects were produced in the game: music, life lost jingle and point-scored jingle. Another thread for the microcontroller was initialized with the task to cycle through the sound types to determine which specific one was to be used at that instant. The sound was created using a buzzer connected to the TI Educational BoosterPack, and the specific frequencies sent to the buzzer were created using a pulse width modulator from the TI microcontroller.

Pulse width modulation (PWM) is a technique on the microcontroller that can digitally encode analog signal levels via counters used to generate a square wave and the duty cycle to modulate the square wave. This specific microcontroller has two PWM modules with four PWM generator blocks and a control block. Each generator block produces two signals that share the same timer and frequency.

Due to the nature of the buzzer on the TI Educational Booster, only one signal can be sent as only one pin is connected to it via pin J4-40. The pin that corresponds to that pin on the microcontroller is pin PF2. For this pin, the PWM is created by the Motion Control Module 1 PWN 6 and controlled by Module 1 PWM Generator 6. This pin is also connected to the blue LED on the microcontroller board, and so whatever frequency is sent is also received by the LED and the buzzer.

In the game, the PWM was initialized with these properties: the clock was enabled to PWM 1 module and enabled to clock to PORTF in the system control registers, PF2 was set as a digital output pin in the general-purpose input/outputs port f registers, generator 3 was enabled with having a select down count mode and would reload and clear when the output matched in the PWM 1 generator 3 registers, and a clock division was enabled in the system control run-mode clock configuration register. For every frequency that was created, a load value and a duty cycle were generated. The load value was based on the division of the clock frequency with the divider divided by the desired frequency and stored in the load register of generator 3. The duty was always set to 50% of the value of the load creating a 50% duty cycle for the square wave and stored in the compare register of generator 3. A clock division of 8 was chosen to be able to produce a load value in which

frequencies as low as 50Hz could be produced by the PWM and therefore sent to the buzzer.

For every pitch that the game wanted to play on the buzzer, it ran through four steps. First, the PWM 1 channel 6 output is enabled with a specific load value and duty cycle. Second, a tempo global variable and a parameter that represents the duration of the pitch initiate an OS_Sleep on the sound thread. Then, the PWM 1 channel 6 output is disabled stopping the sound. Finally, another OS_Sleep on the sound thread is called for some duration that creates a separation between two different notes, similar to staccato in music theory.

Three songs were created for the game: E2M6 [3], which plays from when the title shows to when the gameplay starts; E1M1 [4], which plays during gameplay and continues through the level screens; and E2M3 [5], which plays at the game over screen. All three songs were stored in two-dimensional arrays of some value representing the length of the song and by three elements: the note, which is defined as the frequency of the pitch, the duration of the note, and the duration of the rest that comes after the note. We included a single-beat rest after every note to create a distinctive staccato sound, which resulted in the soundtrack having a sharper, more intense feeling. These arrays were all stored in a single header file, "sound array.h". An example of the array structure is as follows:

```
static uint16_t example_song[][3] = {
  {E3, 1, 1},
  {E3, 1, 1},
  {E4, 1, 1},
  {E3, 1, 1},
  {E3, 1, 1},
  {D4, 1, 1},
  ...
  };
```

The sound thread is initialized the same way as the render thread and is also connected with its own FIFO as well as semaphores that detect if there is an incoming sound to play and if the PWM is ready. The FIFO allows for 8 specific actions: whether the background music is enabled, disabled, or toggled, whether a point is scored, whether a life is lost, and whether the effect sounds are enabled, disabled, or toggled. If the FIFO detects an event, it signals a specific data signal using data.command and wakes up the sound thread. There are four cases for the sound thread to choose from: music, the point-scored sound effect, the life-lost sound effect, and nothing.

Due to the nature of the music being constantly being played in the background: a variable called BackgroundMusic is used to check if music is enabled. Once enabled, it checks what scene the game is in at the moment, using the currentTrack variable, and plays that specific track of music. The song starts at the beginning and an index musicPosition tracks the array and cycles through the song allowing it to repeat. If the variable currentTrack changes, the index musicPosition is reset, as all changes in currentTrack occur when the game moves to a new process (title to gameplay, gameplay to game end, game end to title, or a reset).

There are two specific sound effect sounds that play with no interruption with other sounds that occur: point-scored sound and life-loss sound. While the background music is playing, if one of those events occurs, then the music stops and it plays that specific sound effect the whole way through. Then, the music begins playing at the same place that it was interrupted. If both sound effects occur within a time frame when one is still finishing and the other is just starting, the one starting overwrites the other one from finishing.

D. Deadlock Prevention

Since each demon sprite ("cube" object in the terms of the project requirements) was a 16-pixel by 16-pixel bitmap image and the LCD had dimensions of 128 pixels by 128 pixels, the LCD was divided into a grid of 16-pixel by 16-pixel "cells" to apportion to the demons. A corresponding two-dimensional array of semaphores was implemented to safely synchronize LCD resource use between demon threads.

A utility function `OS_Try()` was developed to allow threads to attempt to acquire semaphores for adjacent LCD "cells" without invoking a context switch if such cells were unavailable. This is in contrast to the previously implemented `OS_Wait()` family of utilities in the kernel that suspend the calling thread and prompt a context switch if a semaphore is currently unavailable. Since demon speed was a critical design value, threads were allowed to remain awake and instead use whatever remained of their computation time to randomly generate a new direction and try for the semaphore in that new direction.

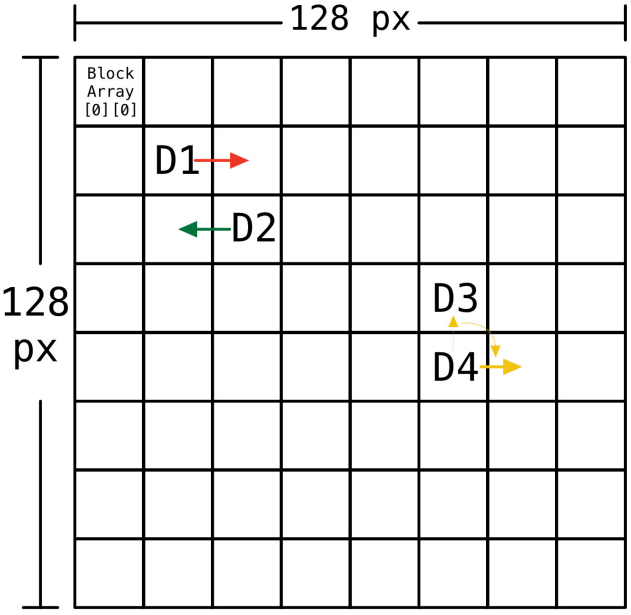

Fig. 3. A graphic depicting a sample conflict resolution using the deadlock prevention scheme.

Fig. 3 demonstrates an example scenario that the deadlock

prevention scheme can resolve. The LCD is logically divided into an 8x8 two-dimensional matrix of cells where each demon (indicated as `D1`, `D2`, `D3`, or `D4` in the graphic) occupies one cell at one instant. If a demon calculates its direction and can atomically check and acquire the semaphore in that direction using `OS_Try()`, the demon thread simply acquires the new semaphore and adjusts its position accordingly. In this manner, demon threads `D1` and `D2` in the graphic can move to adjacent cells without conflict since they compete with no other threads to acquire the semaphores corresponding to those cells. The diagram displays an example conflict, however, between demon threads `D3` and `D4`: `D4` attempts to move "upwards" into the cell that `D3` currently occupies and is not vacating. When `D4` calls `OS_Try()`, `OS_Try()` will immediately return in `D4` and indicate an unsuccessful acquisition of the adjacent cell's semaphore. As an alternative, the `D4` will randomly generate another direction; the diagram supposes that `D4` recalculates its direction as moving to the right. Since the cell to the right is free, `D4` acquires this cell instead and moves to the adjacent cell to the right as its next position.

The deadlock prevention scheme eliminates circular wait and, to a certain extent, imposes resource preemption in threads' use of LCD cells. Most critically, the ability of threads to regenerate random directions prevents an indefinite circular. While it is true that a thread may regenerate the same direction and try for an adjacent cell that it has already unsuccessfully attempted to move toward, the thread is theoretically no more likely to try again in that direction than it is to try to acquire an adjacent semaphore in a different direction that is available. In their long-term behavior, threads will not be specifically and consistently waiting on adjacent cells in a way which creates the persistent circular wait required for deadlock. Furthermore, since a thread has the potential to generate a new random direction and "stop" itself from pursuing an unavailable adjacent semaphore, threads can preempt themselves to a certain extent by forcing themselves to look to other LCD resources when certain cells are at first unavailable. By eliminating one necessary condition and substantially reducing another, the design thus prevented deadlock; physical testing on the Tiva board supported this conclusion.

IV. Results and Evidence

Our final product fulfilled the following requirements:

- The crosshair moves under the control of the joystick
- Object disappears when overlapped by the reticle, or when its time expires
- Score increases when an object is successfully "killed"; life decreases when an object's lifetime expires without being successfully targeted
- Game ends when life = 0
- Game restarts when pushing a button. In our game's case, the game can be restarted any time by pressing select, or on the game over screen by pressing S1.
- Sound effects play when an object is targeted, and a song plays when the game over condition is reached
- Effective deadlock prevention was implemented

Our final product intentionally differed from the parameters, with justification, in the following ways:

- $1 \leq n \leq 5$ cubes should be displayed at all times: rather than a number of cubes displaying on the screen at any given time, we chose to break our game up into level "scenes," where, during the earlier levels, between two to five enemies would spawn at the start of each scene. In order to scale up difficulty, we increased the number of possible enemies to spawn at higher levels.
- Objects should move in their initially designated directions until reaching the LCD boundaries, having their lifetime expire, or until being pointed in a different direction as a means of avoiding deadlock: We found the predictable movement of blocks back and forth across the screen made for uninteresting and predictable gameplay, so we opted to instead have all movements determined by RNG. This allowed our enemy objects to achieve some level of evasiveness, making for a more challenging experience.

Our project added the following bonus features:

- Our enemy objects used different sprites depending on the direction of their movement.
- Our enemy sprites would play an animation when they were nearing the end of their lifetime, giving the player a small window of warning.
- Our game used an "ammo" counter, requiring the player to "reload" their ammo with the S2 button before continuing to defeat enemies. This reload disabled the hit detection of the cursor for a brief period of time, requiring the player to think strategically about when they would reload their ammunition.
- Our game had a full soundtrack, which included a theme for the start screen, the main gameplay loop, and the game over screen. These soundtracks did not prevent other sound effects from being played during gameplay.

A. Gameplay Verification

Live testing on the Tiva board confirmed that the game deliverable satisfied the design requirements.

In Fig. 4, live testing confirmed that the board could successfully render the bitmap images in the way the game's design expected. Rendering bitmaps in this way thus enhances the player experience by delivering them a visually interesting, detailed start screen.

As shown in Fig. 5, live testing was likewise able to verify the game's initial state. The player begins the game with sensible parameters and a satisfactory idea of what they should do to continue playing the game while avoiding a "game over".

Similarly, Fig. 6 demonstrates that the gameplay maintains a correct state in response to player input. The game not only initializes with appropriate parameters, but appropriately responds to player input through the joystick as well as the events player input can cause, such as demons being eliminated from the screen. Furthermore, Fig. 6 demonstrates

Fig. 4. MOOD start screen demonstrating creative, detailed use of bitmap images to enhance the product.

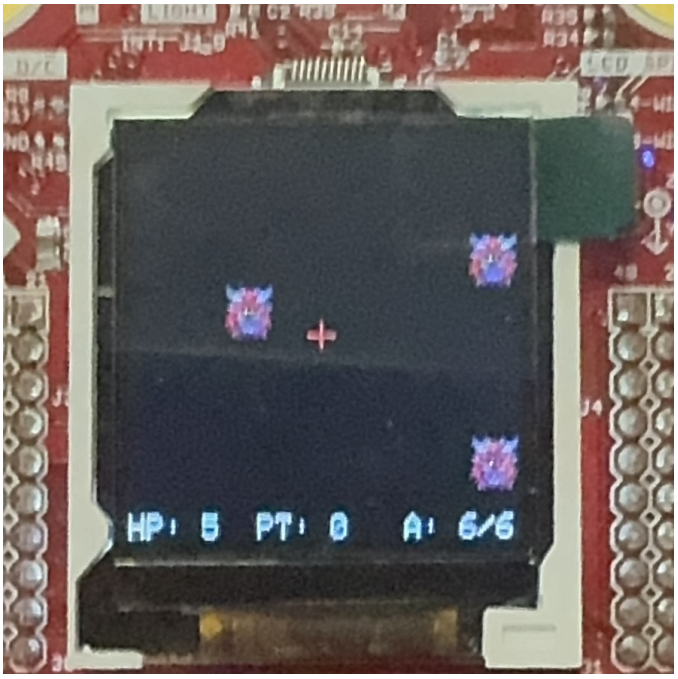

Fig. 5. Initial gameplay view demonstrating bitmap sprites for demons and game-maintained counters for lives left, score, and ammunition.

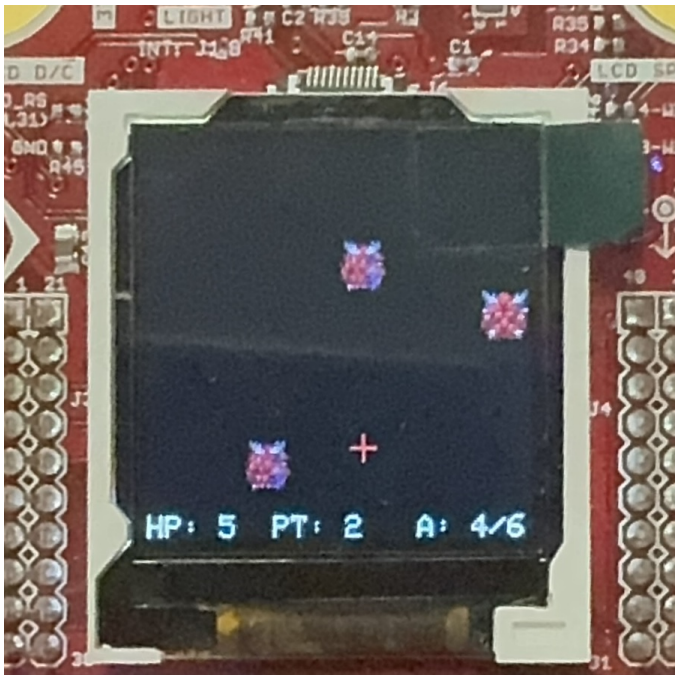

Fig. 6. In-progress gameplay view demonstrating responsiveness to player input. The player's score increases, and ammo correspondingly decreases, as the player collides with demons and eliminates them from the screen.

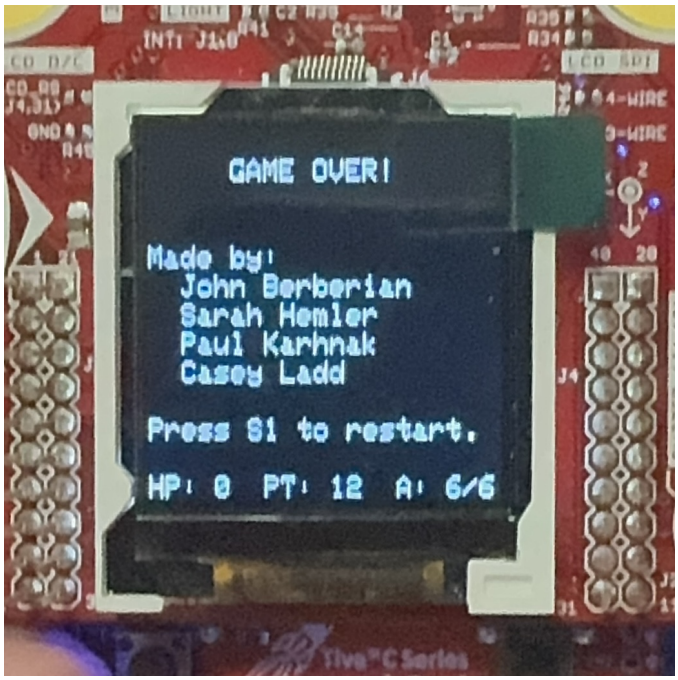

Fig. 7. Game over screen demonstrating a controlled, graceful exit from the main gameplay in preparation for a player-initiated restart.

that multiple different sprites were successfully used for the demons to create a more interesting, visually appealing player experience.

Lastly, Fig. 7 demonstrates an appropriate "game over" progression if the player loses all of their lives. Once the player's lives reach zero, the screen halts gameplay, reports final statistics for that particular game, and gives the player

appropriate instructions if they wish to restart the game.

Given this visual evidence, it may be concluded that the game deliverable not only satisfies appropriate design specifications, but exceeds the minimum requirements to provide an intriguing experience for the player.

V. Lessons Learned

When we started this project, we believed we could easily fit our program into the microcontroller, no matter how big and ambitious it got. However, we did not anticipate that we would run into licensing issues. When we included all of our sprites in the program, we found that only John could still compile the project - all the others were limited to a maximum binary size of 32KB because of the licensing restrictions on the ARM compiler. We attempted to overcome this while keeping the same compiler version, but eventually realized that the only path available to us was to update to a newer compiler that supports the new non-commercial licensing scheme. As such, we had to switch to compiler version 6, which required some modification to the base OS, specifically the `parrotdelay` function in `LCD.c`. We split it off into its own assembly file, and called it from there.

We did not realize how time- and resource-intensive a project of this magnitude would require. The process of creating this game has given us some real insight into what we need to put forward as a team in order to make our goals a reality. While we started out with discrete task assignments, we found that, as the deadline approached, our division of labor became more fluid, and real-time communication became all the more important.

We also learned that it is helpful in complex software projects to introduce simplifications to reduce semaphore contentions.

VI. Bibliography

References

[1] D. Craddock, *Knee-deep in the ports: Ranking the best (and worst) versions of doom*, Available at https://www.shacknews.com/article/116670/knee-deep-in-the-ports-ranking-the-best-and-worst-versions-of-doom (2020/03/17).

[2] G. Marsaglia, "Xorshift rngs," *Journal of Statistical Software*, no. 14, pp. 1–6, 2003.

[3] R. Prince, *Doom e2m6 sinister (synthesia)*, Available at https://www.youtube.com/watch?v=vrru6UROo6w.

[4] J. Lowe, *At doom's gate [e1m1]*, Available at https://musescore.com/user/31157784/scores/7719857.

[5] B. Prince, *Intermission: E2m3 from doom (1993)*, Available at https://musescore.com/user/6921406/scores/6808741.