Advanced Embedded Final Report

Cube Game Design Analysis

Team Delta: Christian Bliss, Gabriel Gladstone, Sharon Lu, Daniel Xue

***Abstract*— We implement the baseline Cube Game and extend it to include bitmap images, evil cubes, and accelerometer cursor control. Using the Mini-Project 4 Solution code for our baseline RTOS, joystick cursor control, and LCD control, we developed new libraries for random number generation, cube generation and movement, interactive sound effects, bitmap graphics, and accelerometer movement. These components were individually tested and integrated into a cohesive system, which was shown to both successfully prevent deadlock and meet the requirements put forth for the Final Project in two video demonstrations.**

CONTENTS

I. INTRODUCTION

The point of this final project was to use the real-time operating system implemented in the four mini-projects of the class to implement and extend the Cube Game on the TIVA C Launchpad with the Educational Boosterpack MkII header board. This involved integrating prior methods of joystick control and LCD display with the new tasks of cube generation and movement, interactive sound design, and deadlock prevention. Per guidance from course staff, the key to our approach was to systematically implement and debug each component of the project in pairs. We then used Git and Github to coordinate source control, testing, and integration of each component until the final product was completed.

We included two novel features to the base Cube Game: 1) the addition of "enemy" cubes that tracked the cursor and decreased player life upon collision and 2) the ability to toggle between joystick and accelerometer control of the cursor. Such additions were enabled by our parallel work methods, where one pair could focus on debugging cube movement while the other focused on implementing and testing the new features. Additional interactive graphical and sound effects were also added for quality of play as well.

The rest of this work goes into an overview of the system data flow, team responsibilities, technical details regarding the novel system components, verification of the technical requirements put forth for the game (as well as a link to our video demonstration), and finally, lessons learned before concluding.

II. SYSTEM DESCRIPTION

A data flow diagram of the cube game system can be found in Figure 1. Much of the Mini-Project 4 Solution code was largely kept with minimal modifications for joystick control, cursor display, and score and life display. Additionally, many of the same functions for interacting with button input and LCD output were kept from the Mini-Project 4 Solution code. Most of the remaining methods and functions in the 'main.c' were modified to implement both the baseline cube game and our game extensions.

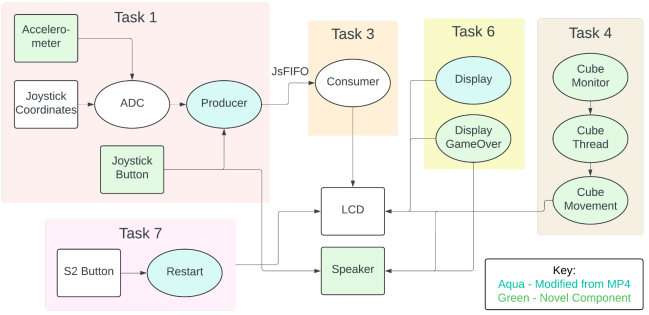


Fig. 1. System data flow diagram. Green indicates a novel component, whereas blue indicates a significantly modified component.

For instance, Task 1 in the Mini-Project 4 solution was modified to allow the board's analog-to-digital converter (ADC) to initialize and receive input from the accelerometer, and the Producer method similarly had to be modified to take in input from the Joystick select button. Likewise, Mini-Project 4's Task 6 was modified to feature multiple Display methods, one modified from the original to display Score and Life in real-time while the other instead displayed the game over screen. Task 7 from Mini-Project 4 was modified to have the Restart method reset not only the Mini-Project 4 tasks, but also the newly added tasks to the

system. With the addition of interactive sound effects, each of these tasks now not only shared the LCD but also the speaker on the Educational Boosterpack MkII board. Note that the tasks associated with the UART Interpreter and the S1 Button in Mini-Project 4 were not included, since we did not connect them to any related tasks for our Cube Game implementation.

For new additions to the system, Task 4 of Mini-Project 4 was modified to manage the initialization, movement, and deadlock prevention of the cubes in the cube game by using one CubeMonitor thread, 1-5 CubeThreads, and a virtual six by six table of semaphores corresponding to each block. Details of the implementation of cube generation and movement are described in Section IV.

III. Team Responsibilities

As a team, we had weekly status meetings to discuss our progress and write our weekly milestones. Members were responsible for and successfully completed the tasks shown in Table I.

TABLE I. Team Responsibilities

Team Member	Responsibilities
Christian Bliss	● Code cube movement and wall detection ● Create deadlock prevention mechanisms to be tested by forcing deadlock between multiple cubes, including innocent and enemy cubes.
Gabriel Gladstone	● Enable sound interfacing and sound effects. ● Create and validate the random library. ● Enable accelerometer for cursor movement. ● Generate and implement bitmaps for cube graphical effects (e.g. life left, eye tracking). ● Edit the video demonstration.
Sharon Lu	● Code block generation, initialization, and game monitoring. ● Add crosshair and cube collision detection. ● Implement game reset with S2. ● Record the video demonstration.
Daniel Xue	● Port over MP4 Solution joystick code ● Add new display for Score and Life ● Add cube lifetime and move timers ● Integrate game over screens and sound effect with game reset mechanism ● Create preset cube initializations for deadlock prevention testing ● Write up weekly progress reports

IV. Design and Implementation

In this section, we cover the implementation of the novel features to our cube game. Since joystick movement and score and life display were based on existing code from the Mini-Project 4 solution, their implementation has already been discussed in Section II.

A) Random Number Generation

The random number generator used in the project was a linear-feedback shift register (LFSR), specifically a Fibonacci LFSR. A Fibonacci LFSR produces a random number from XORing and right-shifting its previous state. The Wikipedia implementation of a Fibonacci LFSR was used and validated [1]. We chose to use this method of random number generation because bit-shifting and manipulation is not computationally intensive.

The LFSR was used to implement the get_rand(upper bound) function. This function returns an integer in the range [0,upper bound). The get_rand function was important in providing randomness to cube spawning, cube movement, and other decisions like cube color and lifetime. The LFSR returns a 16-bit random number, so to get a number within the desired range, the 16-bit number was masked. This was straightforward for numbers that are multiples of 2. For calls such as get_rand(5), the LFSR was masked to produce a number 0-7 and a new random number would be generated from LFSR until it was in the range 0-5.

We first validated the randomness of the LFSR implementation with the starting seed. As can be seen in Table II, calling get_rand(n) where $n \in \{2, 4, 8, 16\}$ returned the expected probability results for 1 element.

TABLE II. LFSR Probability Verification with Fixed Starting Seed (0xACE1u)

Trials	*Range*	*Expected Probability per element*	*Probability of 1 element (arbitrary)*
100,000	[0,2)	0.5	0.499470
100,000	[0,4)	0.25	0.249590
100,000	[0,8)	0.125	0.124980
100,000	[0,16)	0.0625	0.062760

With the pseudo-randomness validated, we now had to find a way to provide the function with a good random starting seed. We decided to use OS_Time() to generate the starting seed. In our implementation, OS_Time() would be called and the seed would be set upon the first call of get_rand(). However, in the following table, we verified the seed with OS_Time() being called right after launch. As seen in the table, OS_Time() generates a 32-bit number, and only the lowest 16-bits are needed for the seed. Even in this worst-case scenario, when OS_time is called right after launch, we see that the seeds on multiple calls are different and therefore an adequate amount of randomness is added to the function.

TABLE III. RNG Seeds Generated with OS_Time after Launch

Trial	Responsibilities
1	0x14BE**1FC4**
2	0x14BE**1F7E**
3	0x14BE**2346**
4	0x14BE**2E76**
5	0x14BE**1B00**
6	0x14BE**2E6A**

These seeds were tested as shown in Table IV and like the fixed starting seed, they result in the expected probability.

TABLE IV. LFSR PROBABILITY VERIFICATION WITH VARIABLE STARTING SEED FROM OS_TIME (IE 2E76)

Trials	*Range*	*Expected Probability per element*	*Probability of 1 element (arbitrary)*
100,000	[0,2)	0.5	0.500900
100,000	[0,4)	0.25	0.250130
100,000	[0,8)	0.125	0.124770
100,000	[0,16)	0.0625	0.062280

The above results verified the LFSR's pseudo-random number generation and random seeding. Because of this, our RNG library provides strong randomness and replayability to our game.

B) Cube Generation

The cube generation implementation consists of three main parts: the cube array initialization, the cube threads, and the cube monitor. These components are called in the main function at startup as well as the S2-triggered Restart thread.

Since the LCD screen is divided into a six by six matrix, a six by six cube array is used to hold the semaphores acting as mutexes for each area. These semaphores are all initialized to one ('free') in the cube array initialization at startup. Two other semaphores are also set in the cube array initialization function: 1) the cubeNumFlag signals when there are no more cubes on the screen and is initialized to 0, and 2) the cubeNumMutex ensures that only one thread can update the cube number at one time and is initialized to 1.

Cube Threads are responsible for initializing each individual cube with its properties, such as design, evil or not evil, direction, and lifetime. Each Cube Thread then generates a random position for the cube to spawn by calling the random number generator and mapping the resulting value to a spot on the grid. If the grid spot is already occupied by a cube or the crosshair, the cube thread loops through adjacent grid positions until a free spot is found for the initialized cube. Once a free grid position is found, the cube is then drawn on the screen. While the player still has lives, a loop checks whether the cube has been hit by the crosshair, updates the cube's lifetime, decrements the player's life if the cube's lifetime expires, and moves the cube.

Finally, the Cube Monitor is responsible for monitoring the cubeNumFlag. When there are no more cubes on the screen and the player still has lives, the last cube on screen will signal the cubeNumFlag and the Cube Monitor will generate a random number of Cube Threads from one to five.

C) Cube Movement and Deadlock Prevention

If player life is not zero and the cube has not been hit, then each CubeThread will periodically update its cube state and call the MoveCube function. This function takes in a cube's current position, and attempts to move said cube over one space in the grid in the cube's current direction. It does so by first checking if the cube is moving into a wall or if its next space is occupied by another cube. If its next space is obstructed, then the MoveCube function returns 0, and the CubeThread randomly selects a new direction for the cube to move in. If the next space in the cube array is free, then the CubeThread will acquire the new position's semaphore and then move the cube into the space as shown in Figure 2.

By having each cube check the semaphore of the next space before attempting to claim it, it is highly unlikely that a cube will wait on a space that is not free in practice. If two adjacent cubes are attempting to claim each other's spaces, both will simply find that the other's space is occupied before attempting to acquire each other's semaphore, and a different random direction will be chosen for both cubes until they are able to move into a free space.

```
// check if moving to different spot
if (x_cube != x_next || y_cube != y_next) {
    // if different spot not free don't move
    if (CubeArray[x_next][y_next].BlockFree.Value == 0) return 0;

    OS_bWait(&CubeArray[x_next][y_next].BlockFree);
    EraseCube(*c);
    c->position[0] = x_next;
    c->position[1] = y_next;
    UpdateCubeState(c);
    DrawCube(*c);
    OS_bSignal(&CubeArray[x_cube][y_cube].BlockFree);
    return 1;
}
// if not moving, return 0
return 0;
```

Fig. 2. MoveCube deadlock prevention mechanism code.

D) Enemy Cube Movement

For one of our extensions, we implemented 'evil' cubes which chase the cursor to make the player lose life. Instead of the typical scenario where each cube randomly chooses a direction to move until they run into a wall or another cube, the 'evil' cubes always choose the direction that gets them closest to the current position of the cursor, as shown by the code in Figure 3. If the cursor's position is diagonal relative to the 'evil' cube i.e. both north and east, the 'evil' cube will randomly select between the two directions when deciding its next move. In the case that the comparison with the cursor's position returns an invalid result, the enemy cube simply picks its next randomly.

```
direction_t next_cube_direction(Cube c, uint16_t x, uint16_t y) {
    if (c.evil) {
        uint16_t cube_x_block = c.position[0];
        uint16_t cube_y_block = c.position[1];
        uint16_t cursor_x_block = x / CUBEHORIZDEM;
        uint16_t cursor_y_block = y / CUBEVERTDEM;

        if (cursor_y_block == cube_y_block && cursor_x_block > cube_x_block) {
            return RIGHT;   // or east

        } else if (cursor_y_block < cube_y_block && cursor_x_block > cube_x_block) {
            return get_rand(2) ? RIGHT : UP;    // randomly choose between east and north

        } else if (cursor_y_block < cube_y_block && cursor_x_block == cube_x_block) {
            return UP;      // or north
```

Fig. 3. Next Cube Direction Implementation for 'Evil' Cubes

E) Bitmap Images

To enhance the gaming experience, we decided to use our own pixel art of 21 by 19 sprites for innocent cubes and evil cubes. To store each sprite we used the R5G6B5 color format, which encodes each full-color pixel in sixteen bits where the red and blue channels are stored in five bits and the green

channel in six. These 16-bit sprites could be drawn onto the LCD using the BSP_LCD_DrawBitmap function shown in Figure 4 to draw a bitmap image from bottom left to top right at the specified location and size.

```
void BSP_LCD_DrawBitmap(int16_t x, int16_t y, const uint16_t *image,
                        int16_t w, int16_t h);
```

Fig. 4. BSP_LCD_DrawBitmap function.

We had to account for the different method of position handling when drawing visuals on the LCD. Because the bitmap images started at the bottom left instead of the top left, we had to offset the y position by the length of the vertical dimension minus one. For a 21 by 19 pixel grid, this would be a y-offset of $19 - 1 = 18$, as shown in Figure 5.

```
// Draw cube
OS_bWait(&LCDFree);
BSP_LCD_DrawBitmap(c.position[0] * CUBEHORIZDEM,
                   (c.position[1] * (CUBEVERTDEM)) + CUBEVERTDEM - 1,
                    evil_cube_state[c.cube_state], 21, 19);
OS_bSignal(&LCDFree);
```

Fig. 5. BSP_LCD_DrawBitmap function y-offset calculation.

Each pixel art sprite also communicates cube state to the player, as shown in Figures 6 and 7. The innocent cubes change colors and facial expressions from green to red as their life decreases. If the cursor does not hit the innocent cube in time, then the user loses a life. The evil cube's eyes also follow the cursor around the screen, letting the user know that the evil cubes are chasing them. For both cubes we used a finite state machine to track and update visual states. The innocent cubes update based on their life available and the evil cubes update based on the user's cursor position relative to their position.

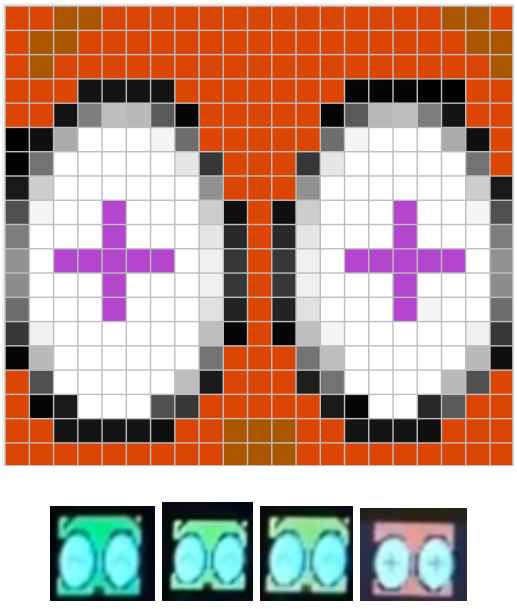


Fig. 6. Sample innocent cube sprite (top). Innocent cube color and facial expression depends on their lifetime, where cubes with less life appear more red and distressed (bottom).

Fig. 7. Sample evil cube sprites. Evil cubes eyes' track cursor position until they are able to finally hit the cursor or they despawn.

F) Sound Effect Generation

The piezo buzzer on the Educational BoosterPack MKII generates sound effects to notify a game over, a cube destroyed, toggling between move modes, and more. These sound effects provide the user clear feedback and enhance the gaming experience.

Sound functionality is initialized as shown in Figure 8 with the associated page numbers for reference from the TM4C datasheet [2]. To generate sound, a pulse width modulation signal (PWM) is produced with a certain frequency and sent to the pin connecting to the Piezo Buzzer. First, the clock is enabled for PortF because the piezo buzzer uses the PF2 pin [3]. Because the system clock runs at 80MHz, and the load value is only up to 16 bits, we decided to enable the PWM clock divider at its maximum division (64). This allows us to use a wider range of frequencies. Next, the PWM signal is set to produce a digital output on PF2. Finally, the PWM signal is set up, and the default frequency is determined.

```
void Sound_Init(void)
{
    // Clock setting for PWM and GPIO Port.
    SYSCTL_RCGCPWM_R |= (1 << 1);                  // Enable clock for PWM1, pg. 354
    SYSCTL_RCGCGPIO_R |= (1 << 5);                 // Enable system clock to PortF, pg. 406
    SYSCTL_RCC_R |= SYSCTL_RCC_USEPWMDIV;          // Enable PWM clock divider (as driver)),
    SYSCTL_RCC_R |= (7 << 17);                     // Divide PWM clock by 64 pg.255

    // Setting of PF2 pin for M1PWM6 channel output pin
    GPIO_PORTF_AFSEL_R |= (1 << 2);                // Enable alt funct on PF2, pg.671
    GPIO_PORTF_PCTL_R = GPIO_PCTL_PF2_M1PWM6;      // Assign PWM signals in the appropriate
    GPIO_PORTF_DEN_R |= (1 << 2);                  // Set as digital pin, pg. 683

    PWM1_3_CTL_R &= (~(1 << 0));                   // Disable Generator 3 counter, pg.1244
    PWM1_3_CTL_R &= (~(1 << 1));                   // Select down count mode, pg.1244
    Update_Note(NOTE_G2);                          // Default note,
    PWM1_3_GENA_R |= (1 << 2) | (1 << 3) | (1 << 7);// Set PWM output when counter reloaded
    PWM1_3_CTL_R |= (1 << 0);                      // Enable Generator 3 counter, pg.1244

    // init the speaker
    OS_InitSemaphore(&SpeakerFree, 1);
}
```

Fig. 8. Sound Initialization

Sound effects are produced by changing the load value and the PWM_CMPA register value. The load value in down counter mode determines the frequency of a note. We know the driving frequency with the clock divider is $\frac{80\,MHz}{64} = 1.25\,MHz$. Thus, to get the frequency of a note, we set the load value to $\frac{1.25\,MHz}{Note\ frequency}$. For a 50% duty cycle, the PWM_CMPA register is set to the load value divided by two. The clock cycles specified in the load value produce roughly

the desired frequency. By enabling, waiting, and disabling the PWM signal, and updating the note, we can produce sound effects.

Because these sound effects are short, we need them to be atomic. Thus, we use an additional semaphore as a mutex to allow each thread to play each sound effect to its full duration before the next can be played. If we were to implement background game music, this would not work, but is suitable for sound effects that are short.

G) Game Over Screen

When the user's life drops to zero but the reset button has not been hit, a game over screen is displayed with the user's final score and instructions for resetting. This is done by setting a restart to 1, which causes the ongoing Display thread to add a new DisplayGameOver thread and then terminate. This new thread draws the Game Over screen, plays a Game Over sound effect, and then waits until the game has been restarted before terminating.

H) Reset Button

When the player presses the S2 button, the game is reset to its initial state, similar to the startup sequence in the main function. The score and lives are set to their starting values, and a random number of cubes from 1 to 5 is generated. This is accomplished by killing the existing threads by setting life to 0, setting the OS to sleep briefly to ensure that all relevant foreground threads are terminated, then adding all of the task threads to the system again.

I) Accelerometer Movement

The accelerometer from the Educational BoosterPack MKII is used as an alternative way to move the cursor. If the user presses down on the joystick, they can toggle between moving with the joystick and moving with the accelerometer by tilting the microcontroller. The accelerometer is configured similarly to the joystick—as both peripherals return analog information through the ADC. Figure 8 outlines the accelerometer initialization with the associated page numbers for reference from the TM4C datasheet [2].

```
// 3-axis Accelerometer
// x axis: J3.23/PD0/AIN7
// y axis: J3.24/PD1/AIN6
// z axis: J3.25/PD2/AIN5
// the accelerometer uses sample sequencer 2
void Accelerometer_Init(void){
  SYSCTL_RCGCGPIO_R |= 0x00000008;                  // Activate clock for Port D, pg.340
  GPIO_PORTD_AMSEL_R |= ((1<<0) | (1<<1) | (1<<2));  // Enable analog functions for PD0,PD1,PD2, pg
  GPIO_PORTD_DIR_R &= ~((1<<0) | (1<<1) | (1<<2));   // Make PD0,PD1,PD2 input, pg. 663
  GPIO_PORTD_AFSEL_R |= ((1<<0) | (1<<1) | (1<<2));  // Enable alt functions on PD0,PD1,PD2 input,
  GPIO_PORTD_DEN_R &= ~((1<<0) | (1<<1) | (1<<2));   // Enable analog functionality/Disable digital
  ADC0_ACTSS_R &= ~(1<<2);          // 10) disable sample sequencer 2, pg.821
  ADC0_EMUX_R &= ~0x0F00;           // 11) seq2 is software trigger, pg. 833
  ADC0_SSMUX2_R = 0x0467;           // 12) set channels for SS2, pg.867, AIN7 then AIN6 then AIN4
  ADC0_SSCTL2_R = 0x0060;           // 13) no TS0 D0 IE0 END0 TS1 D1, yes IE1 END1
  ADC0_IM_R &= ~(1<<2);             // 14) disable SS2 interrupts, pg. 825
  ADC0_ACTSS_R |= (1<<2);           // 15) enable sample sequencer 2, pg.821
}
```

Fig. 9. Accelerometer Initialization

As shown, the Port D clock is first enabled as the accelerometer has Port D pins. Then the accelerometers x,y, and z axis sensors are enabled for analog functionality. Finally, ADC sampling is set up to sample the x axis, y axis, and joystick button presses. Note that the z-axis can be easily added for future extensions to the game.

With the accelerometer initialized, data from the ADC could be read with the function in Figure 9. The only difference between this function and the joystick is the hardware peripheral read for the x and y values. No other code had to be modified for movement control toggling as the interfaces for input were the same.

```
#define SELECT    (*((volatile uint32_t *)0x40024040))  /* PE4 */
void Accelerometer_Input(uint16_t *x, uint16_t *y, uint8_t *select){
  ADC0_PSSI_R = (1<<2);             // 1) initiate SS2, pg. 846
  while((ADC0_RIS_R&(0x04))==0){};  // 2) wait for conversion done, pg.823
  *x = ADC0_SSFIFO2_R;              // Read x, pg.860
  *y = ADC0_SSFIFO2_R;              // Read y
  // *z = ADC0_SSFIFO2_R;           // Read z
  *select = SELECT;                 // return 0(pressed) or 0x10(not pressed)
  ADC0_ISC_R |= (1<<2);             // 4) acknowledge completion
}
```

Fig. 10. Accelerometer Reading

From testing, we did notice that the accelerometer has a smaller change in values, so in the UpdatePosition function, we made its response more sensitive to change.

```
int UpdatePosition(uint16_t rawx, uint16_t rawy, jsDataType *data)
{
    uint8_t step = accelerometer == 0 ? 9 : 7;
    if (rawx > origin[0])
    {
        x = x + ((rawx - origin[0]) >> step);
```

Fig. 11. UpdatePosition sensitivity based on movement mode.

While toggling movement mode by pressing the joystick usually worked, we did notice some unresponsiveness in testing. To try to deal with this, we added a sound effect after toggling to prevent a possible double toggle and to alert the user when the mode has changed.

We thought this was good bonus functionality, because it enhanced the user experience by allowing them to interact with our game in a different way and it allowed us to learn more about interfacing with hardware in an embedded system.

V. Evaluation Results

To test the deadlock prevention system, we created multiple preset cube initializations to force a potential deadlock. The most complex of these feature four pairs of adjacent evil and non-evil cubes each initialized such that they each attempt to move into an occupied space for their first movement. Without proper deadlock prevention and mutual exclusion of each block, these cubes exhibit "phasing" behavior where they could instantly switch positions in the first movement period, or just downright deadlock and disappear as both cubes are blocking on a shared semaphore that prevents them from re-drawing their cube's current position or detecting collisions. Instead, we were able to show that our system prevents both behaviors for all eight cubes in a video demonstration across multiple random trials of the same preset scenario, with each cube clearly picking different, non-obstructed directions to travel in. The link to this video demonstration can be found here: https://drive.google.com/file/d/1kNghERYqKS2Ca15LmC-Kr-t2nEyPptUM/view?usp=sharing.

After verifying deadlock prevention, all components of the system were integrated and verified to meet the requirements for both the baseline Cube Game and our extensions in the video demonstration linked here: https://youtu.be/grG__p6bj7c?si=gjuQT-3BSgpi2T_j, which

shows each of the above novel components successfully implemented and integrated into the complete game.

VI. Lessons Learned

From implementing new hardware functionality such as sound effects and the accelerometer, we learned how to better read and troubleshoot with datasheets [2,3]. What was most useful in our troubleshooting was drawing connections between hardware implementations. Hardware like the accelerometer and joystick produce very similar data so could be implemented in similar ways. Modular system design allowed us to minimize time integrating in new hardware peripheral components to use for implementing and verifying more bonus features.

We also learned the importance of understandable code. When working in a large group and large codebase it can be difficult to collaborate and incorporate everyone's work into the project. Struggling with adding functionality to team members' base code taught us to abstract out complexity, avoid code coupling, and keep the implementation simple. Documenting and discussing code implementation over the course of development allowed us to leverage each of our strengths for improving the project.

In addition to datasheet reading and troubleshooting, learning to use the Keil debugging tools and create simple test cases played a major role in our ability to verify the overall real-time operating system. Without the creation of proper preset test cases for verifying the deadlock prevention system, we would have no easy way to replicate test results for the deadlock system due to the random nature of the system. By creating preset scenarios, we could create unique, repeatable scenarios to test various aspects of our cube movement and deadlock prevention systems.

VII. Conclusion

For our Advanced Embedded final project, our team successfully implemented both the baseline cube game and extended it to use unique control schemes and different cube behaviors. This was done by dividing up each component between developer and debugger pairs on the team, and working in parallel to implement all parts of the final project in time. While there were some elements of the base game we pulled from the Mini-Project 4 Solution, the vast majority of our project's code in the main.c was either significantly modified or newly added. After separately individually testing each component, we were able to test the system as a whole to show that it met each of the requirements of the base game in our video demonstration and class presentation.

There are a number of possible improvements to the project that we would consider if we had additional time to work on this project. First, increasing the grid size and speed of cubes might improve the challenge of the game, which is not very difficult in its current iteration. Second, a number of UI improvements such as a start screen and a clearer description of controls might improve new user experience. Finally, as our classmate pointed out, our deadlock prevention system features a non-atomic check and acquire, which may still lead to a race condition which could cause deadlocks in rare cases. While I believe that we could easily modify our current implementation to use their suggestion of OS_Try, we were sadly unable to do so as we had to return our equipment at the end of the presentation block.

VIII. Acknowledgements

The team would like to acknowledge and thank Prof. Homa Alemzadeh and the teaching assistants, Xugui Zhou, Keshara Weerasinghe, and Kidus Fasil for their instruction and guidance throughout this project and the semester as a whole.

IX. References

[1] "Linear-feedback shift register," *Wikipedia*. Mar. 31, 2024. Accessed: Apr. 26, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Linear-feedback_shift_register&oldid=1216586010#Galois_LFSRs

[2] "TM4C123GH6PM data sheet, product information and support | TI.com." Accessed: Apr. 26, 2024. [Online]. Available: https://www.ti.com/product/TM4C123GH6PM

[3] "BOOSTXL-EDUMKII Daughter card | TI.com." Accessed: Apr. 26, 2024. [Online]. Available: https://www.ti.com/tool/BOOSTXL-EDUMKI

X. Appendix: Final Project Survey Results

Thank you for your feedback on the Final Project.

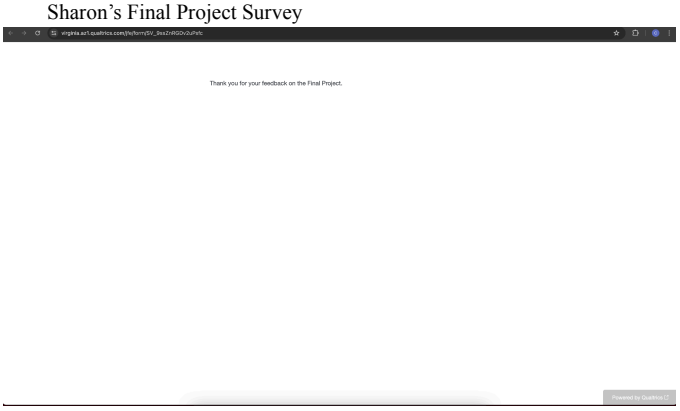
Gabe's Final Project Survey

Thank you for your feedback on the Final Project.

Daniel's Final Project Survey

Thank you for your feedback on the Final Project.

Sharon's Final Project Survey



Christian's Final Project Survey