

Advanced Embedded Final Project - Team Alpha

Jacob Blindenbach
University of Virginia
jab7dq@virginia.edu

William Zhang
University of Virginia
wyz3sp@virginia.edu

James Houghton
University of Virginia
jth5zs@virginia.edu

Guy Verrier
University of Virginia
gju7qw@virginia.edu

1 INTRODUCTION & OVERVIEW

Our final project is a combination of our work in Part 1, where we completed the Cube Game Design, and Part 2, an expansion of the Cube Game features, including a persistent highscore system, a 'power-up' feature, and bitmap graphics. Key take-aways from our game design are fulfillment of project requirements and development of impactful new features reminiscent of real-world games.

1.1 Terminology

To accurately discuss our system, we introduce precise language to describe our various components.

1.1.1 Block and Cube. In this report, we use the term "block" to mean any place in the grid in which a "cube" may be placed. Because we are using a 6x6 grid, there are precisely 36 blocks. A "cube" is an object lying in a block that has a thread controlling it which may or may not contain a power-up.

1.1.2 Reinitialization. We use "reinitialization" to mean the event when cubes must be placed in the Block Grid again. This occurs when all cubes have died.

1.2 Important Functions

1.2.1 Cube Movement Synchronization. Satisfying the requirement that each cube must be moved by their own thread, this function was challenging. Cube movement is done by each cube's own thread: MoveCubeThread. To synchronize the movement, there is a supervisor thread InitAndSyncBlocks.

When the cubes need to be moved, InitAndSyncBlocks must first determine how many cubes are dead, n . It will then Signal the MoveCubesSem semaphore n times and Wait on DoneMovingCubesSem n times. Each cube thread, Waiting on MoveCubesSem, will be able to move once, Signal the DoneMovingCubesSem semaphore, and wait on ThrottleSem semaphore. After InitAndSyncBlocks wakes up from its n Waits, it will signal ThrottleSem n times.

The reason we use a "throttle" semaphore here is to prevent any single Cube from stealing a move from a different cube. Without a throttle, a cube may perform its moving logic and Wait on MoveCubeSem before one of the other cubes has had the chance to check the value of MoveCubeSem.

1.2.2 Cube Movement. For a cube to move, it will determine each direction that it can move in. A cube can move to any adjacent block that does not already contain a cube. There are 36 semaphores that determine which blocks are taken. When the value of any semaphore is 0, it means that the block is taken.

1.2.3 Checking Cube Intersections. Unless something else is going on, a cube will be constantly checking whether or not it has intersected with the crosshair. This is done by calling CheckBlockIntersection. If a cube does not intersect the crosshair, its thread will go to Suspend to allow other cubes to check.

1.2.4 Cube Drawing. Cube drawing is handled by its own thread. The thread, DrawCubes, waits on a semaphore before redrawing. This allows any thread to notify the cube drawing thread that a redraw is required.

Diagram. See figure 1 for a visual representation of our functions and data flows.

2 DESCRIPTION OF NEW FEATURES

In this section, we provide a high-level description of the features that were added in part 2.

2.1 Highscore System

The largest system that was added was an arcade-like highscoring system. This system allows a player to save their score into the memory of the microcontroller. They are able to save a three-letter name with their score, so that they can identify their own scores.

To select a name, we display three letters on the screen. The player will then use the joystick to modify each letter and to change which letter they are modifying. When modifying a letter, the player can scroll through the alphabet, both forwards and backwards.

After saving their score, they can view the top 4 scores saved in the device.

2.1.1 Persistence. We want the highscore leaderboard to persist through device power cycles. To do this, we will need to write the leaderboard to some non-volatile memory that the microcontroller has access to. Details about the choices we made and the implementation are discussed in the implementation section.

2.2 Power-Ups

The other system that was introduced for part 2 was the power-up system. Power-ups are attached to cubes. When a cube with a power-up is hit by the player, the power-up takes effect. Each cube is assigned a single power-up (including no power-up), and that power-up will dictate the cube's color. The effects and colors are detailed below.

2.2.1 No Power-Up. A cube without a power-up will appear as blue. Nothing special happens when it is hit by the player.

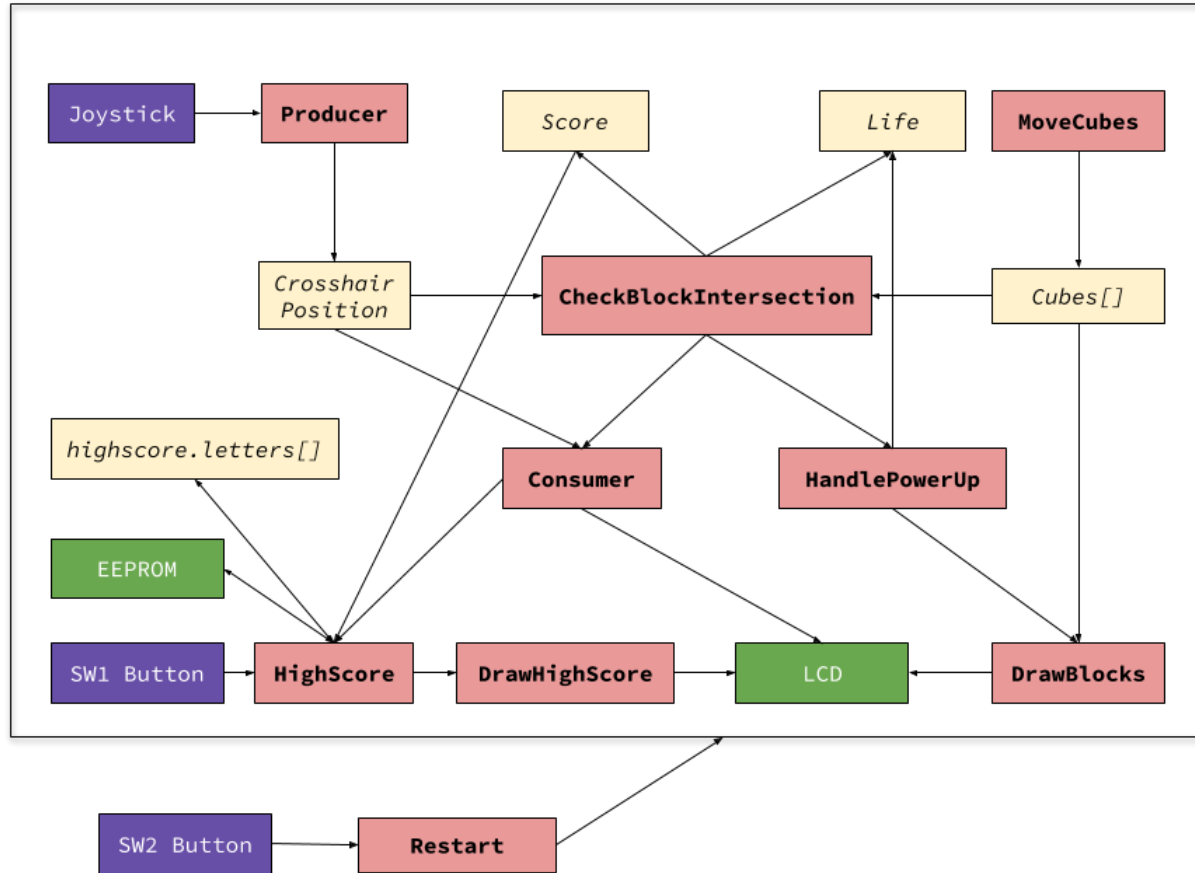


Figure 1: A dataflow diagram for our final project. Green boxes indicate hardware resources, red boxes indicate functions, purple boxes indicate inputs, and yellow boxes are shared variables.

2.2.2 Life. When a Life power-up is triggered, the player's Life is incremented. A cube with a Life power-up will appear as red.

2.2.3 Crosshair Size. This power-up is more complicated than the Life power-up. When a Crosshair Size power-up is triggered, the crosshair's size is doubled for 5 seconds. After 5 seconds, the crosshair will revert to its regular size. If the power-up is hit twice within 5 seconds, the reset timer is set back to 5 seconds. A cube with a Crosshair Size power-up will appear as green.

2.2.4 Crosshair Speed-Up. This power-up is similar to the Crosshair Size power-up. When a Crosshair Speed-Up power-up is triggered, the crosshair's speed is doubled for 2 seconds. After 2 seconds, the crosshair will revert to its regular speed. If the power-up is hit twice within 2 seconds, the reset timer is set back to 2 seconds. A cube with a Crosshair Speed-Up power-up will appear as yellow.

2.2.5 Crosshair Slow Down. This is an anti-power-up; it slows down the crosshair. If a Crosshair Speed-Up power-up is collected,

then the slow down is cancelled and the crosshair speed will be fast. Cubes with the Slow Down power-up will not decrement the player's life when dying of age. The Slow Down power-up is colored gray.

2.2.6 Freeze. This power-up freezes all the cubes on the screen for 2 seconds. The cubes on screen will not move nor will they lose life. Like the Crosshair Speed-Up and Size power-ups, if an additional Freeze power-up is collected, the freeze will continue for another 2 seconds. The Freeze power-up is colored cyan.

2.3 Bitmap Sprites

Another feature that was added for part 2 was the replacement of the simple colored cube objects with 18x18 bitmap sprites. Each of the added power-ups that were added, as described in the previous section, had a corresponding new bitmap sprite. The power-ups and their corresponding sprites are shown below.







Power-up	Description	Sprite
No Power-Up	Coin	
Life	Health Pack	
Crosshair Size	Magic Mushroom	
Crosshair Speed-Up	Sanic (gotta go fast)	
Crosshair Slow Down	Internet Meme Frog	
Freeze	Snowflake	

Table 1: Table of power-ups, their descriptions, and their sprites.

3 IMPLEMENTATION OF NEW FEATURES

3.1 Highscore System

3.1.1 Name Selection. Allowing the user to select a three-letter name was challenging. We had to repurpose the Producer function to work with this application instead of just moving the crosshair. To do this, while selecting a name, we constantly reset the crosshair position back to the center of the screen and find the difference between the new position of the center. This allows us to figure out when the crosshair is moved.

For cursor movement, the cursor is only moved when joystick is initially moved. The joystick must be set back to the center before the cursor will move again. For letter selection, there is no throttling so that the user can quickly scroll through all the letters.

3.1.2 The Leaderboard. The leaderboard data is stored with an array of four entries. Each entry contains the three-letter string that the players have entered and the score.

When a new entry needs to be added, we first linearly search for the first entry that has a lower score than the one we are entering. If no such entry exists, we don't enter the new score. If an entry is found, we move it and all entries after it one place over, discarding the lowest entry. We then place the new entry in the newly available place in the array.

3.1.3 High Score Storage. We wanted the leaderboard data to persist even when power to the microcontroller was lost. From the Tiva C Series datasheet, we notice our microcontroller had EEPROM which can store non-volatile data.

To get the EEPROM initialized and controllable, we had to use drivers created by Tiva. We found their github page and downloaded their source code but it took a while to figure out how to compile everything because there was no documentation. The compilation process involved opening a new keil project with a project file in the driver code. Compiling their drivers into a library file and importing that library file into our original project.

Once we compiled the drivers, we had functions for reading and writing to the EEPROM. The work flow would be that every time we computed the new leaderboard data, we would dump it into the EEPROM, and every time we started up the microcontroller, we would read in the leaderboard data from the EEPROM. This would work unless the EEPROM did not have the leaderboard data which would happen when the code is first executed. To fix this, we tagged the leaderboard data dump with a magic number, and

when we read in the EEPROM data we would check if the magic number was read in. If it was, we knew that the data we read from the EEPROM was valid and usable. If not, then we would ignore the EEPROM data and reinitialize the leaderboard data. This way if the code was executed for the first time, the EEPROM wouldn't contain our magic number and the leaderboard data would be initialized to empty, and on subsequent loads the EEPROM data would populate the leaderboard data.

We ran into some issues reading in the raw EEPROM data and populating the leaderboard data with it. The EEPROM Read function return a `uint32_t` array which had to be casted into a high score struct which was 64 bits.

3.2 Power-Ups

3.2.1 Life. The Life power-up was the simplest to implement. All that was required when a Life power-up was triggered was to acquire the player information lock, `InfoSem`, increment the player's Life, and release `InfoSem`.

3.2.2 Crosshair Size. The Crosshair Size power-up was more complicated to implement. This is a timed-based power-up (i.e., it is only active for a certain amount of time). To do this, when a Crosshair Size power-up is triggered, we spawn a thread. The thread will change the size of the crosshair (protected by a semaphore). The thread will then be put to sleep for 5 seconds. After 5 seconds, the crosshair size will be reset.

To handle the case when a user triggers the power-up multiple times within 5 seconds, there is a global variable that counts how many times the power-up has been triggered. Every read and modification done to this variable is guarded by a lock. If the global variable did not change after the 5 seconds were up, the crosshair size is reverted. If it did change, we know that another thread will reset the crosshair size later, so we do simply exit and do nothing.

To allow for a variable-size crosshair, we needed to change the crosshair drawing function, `BSP_LCD_DrawCrosshair`, to take a crosshair size parameter.

The variable crosshair size posed other challenges as well. When reverting to a smaller size, the crosshair sometimes would not be cleared properly. To overcome this, we simply clear the crosshair every time as if it were the larger crosshair. This did not impose any significant differences, other than fixing the visual bug.

3.2.3 Crosshair Speed. The Crosshair Speed power-up is time-based in the same way as the Crosshair Size power-up. Instead of modifying the size of the crosshair, we modify a global variable called speed. The variable controls how much to shift the velocity in the Producer. When speed is 0, its default value, the speed is not modified. After a Crosshair Speed power-up is triggered, speed is set to 1, and the velocity will be shifted by 1 bit, multiplying it by 2.

3.2.4 Freeze. Freeze was easy to add as the implementation was very similar to the speed power-up. From a design perspective, we had several options for freezing the cubes on screen.

- (1) Call `OS_Sleep()` on all the cube threads: This was the first idea where all the cube threads would sleep for the duration of the freeze. This approach would have worked if the inter-section computation between the cubes and crosshair did not happen in the cube thread. Because this happens, putting

the cube threads to sleep would prevent the crosshair from eliminating the cubes it crosses.

- (2) Return no available directions for frozen cubes: In our code, if a cube has no available directions to move in, it will not move. Thus, by setting the possible directions to zero when a cube is frozen it will act frozen. Unfortunately, this will not prevent the life of the cube from decrements. So this approach was not used.
- (3) Pretend the cube is "dead": If we pretend the cube is dead and do not perform any computations on it besides crosshair intersection, the cube will not move and its life will not decrement. Additionally, we reduce the run time because the cube thread skips the move and decrement functions and almost immediately calls `OS_Suspend()`. This is the option we chose.

3.2.5 Crosshair Slow Down. The slow down power-up code is almost exactly the same as the speed power-up code because it uses the same variables and semaphores. This design choice made it simple to add the feature that a speed power-up overrides the slow down power-up and vice versa. The only significant change was that a slow down power-up cube does not decrease the life when it is not collected. The life value is decremented in the cube thread right before the cube is killed. Adding an additional check that the cube's power-up is not a slow down power-up before decrementing life, completes this feature.

3.2.6 Difficulties. Jacob found that power-ups stayed active after a restart. There were several ways to fix this. The simplest is to reset the modifiers to their initial value, and not cleaning up the reset threads. The reset threads won't take effect when exiting, so this was found to be safe.

3.3 Bitmap Sprites

The original project code included the header file `LCD.h`, which contained a prewritten function to draw a bitmap image on the screen called `BSP_LCD_DrawBitmap`. The function takes `x` and `y` coordinates, a pointer to a `uint16_t` C array containing values representing the color of a pixel, as well as the width and height of the desired image. The challenge then became turning the desired images into sprites that could be passed to the function.

3.3.1 Conversion Process. Since the cubes in the original game were 18x18, it was decided that the sprites would also have the same dimensions. This meant that the C array for each sprite needed to be of length 324. To convert image files into C arrays, a Windows executable that specifically converted 24-bit .bmp files, `BmpConvert16.exe`, was found in the `ST7735_4C123.zip` resource on Jonathan Valvano's webpage. The process for generating a C array for use as a sprite was as follows:

- (1) Crop the desired image to have square or relatively square dimensions
- (2) Color the background of the image to match the background of the game (black)
- (3) Flip the red and blue color channels for each color present in the image
- (4) Scale down the image to 18x18
- (5) Export the image as a 24-bit .bmp file

- (6) Run the `BmpConvert16.exe` program to generate the C array

3.3.2 Code Modifications. In order to avoid bloating the `Main.c` file, the generated bitmap C arrays were put in a separate header file, called `bitmap.h`. The bitmap C arrays were then added to the `Main.c` file with an `#include` statement at the top of the file.

Additionally, to actually draw the sprites, the `DrawCubes()` function was modified as well. Using the defined enumerated type `PowerUp`, a case statement was added in order to call the `BSP_LCD_DrawBitmap` function with a pointer to the corresponding bitmap C array for the power-up type and an offset `y`-coordinate to account for the way the `BSP_LCD_DrawBitmap` function draws the bitmap on the screen.

3.3.3 Difficulties. The `BmpConvert16.exe` conversion program did not take into account the fact that in a .bmp file, the color channels are stored in reverse order (BGR) instead of the normal (RGB). This led to an issue where the red and blue color channels for the sprites were flipped, causing the sprites to appear on the LCD screen as the wrong colors. This was solved by simply flipping the red and blue color channels in the image conversion process.

It was also found that the `BSP_LCD_DrawBitmap` function drew the bitmap from bottom left corner, instead of from the top left corner. This caused an issue of sprites being drawn in the wrong position. To combat this, the `y` coordinate passed into the function was adjusted by adding the block height and subtracting one.

4 EVALUATION & RESULTS

The main result of our work in part 2 is a much more fun, complete cube game. In part 1, we established the basics of the cube game:

- (1) Cubes spawn in rounds and move randomly.
- (2) A player's crosshair moves to touch and kill the cubes, adding to the player's score.
- (3) When all cubes die, more cubes are spawned.
- (4) If a cube's lifetime expires before being touched, the player loses a life.
- (5) If the player's life goes to 0, the game ends.
- (6) Pressing SW2 restarts the game.

Our work in part 2 extends this in several ways:

- (1) Allows a player to save their score, even though power cycles.
- (2) Adds variety to the player's traits with power-ups.
- (3) Adds variety to the cubes' looks with bitmap drawing.
- (4) Improved joystick control.

The end result was a fun, playable, responsive, bug-free game. Even with the newly added features, there was no more noticeable delay on user input despite the additional processing and joystick sampling that was done for part 2.

The three demo videos can be found here:

- (1) <https://youtu.be/0mV5w0kYnA8>
- (2) <https://youtu.be/TUwJ-5sQj-w>
- (3) https://youtu.be/1hYUKJOL8_g

The videos demonstrate all the functions of our game, as listed above.

5 LESSONS LEARNED

5.1 Trade-offs

During the brainstorming phase of this project, we learned to discard interesting ideas that would have taken too long for what they provide. Let's consider an example: cube flickering with the Frozen power-up.

5.1.1 Cube Flickering When Frozen. Normally when cubes are drawn, a black rectangle is drawn on top of its previous position, and then it is redrawn elsewhere. This works well when the cube is moving, but when it is stuck in one place (like when using the Frozen power-up), it flickers every second. Because of the way cube drawing works in our project, this would be a small chore to fix, and wouldn't bring a lot of benefit to the project. For this reason, this idea was discarded.

5.2 Simplicity and Debugging

To get anything to work, we had to have clear, simple contracts between threads. These contracts are just criteria that a thread must comply with to operate properly. This might include something like:

- (1) This thread must acquire a lock, release a lock, and suspend afterwards (relevant for cube movement synchronization).
- (2) This thread must use a lock to guard accesses to a particular variable.

The contracts had to be simple and clear, and they also had to be formulated properly so that, by adhering to them, the entire system would work. This was mostly a concern for Part 1, where most of the synchronization-heavy code was implemented, but we still needed to keep these principles in mind for Part 2, especially when working on the power-up implementations.

6 TEAM RESPONSIBILITIES

6.1 Jacob Blindenbach

- (1) Wrote part 1 report.
- (2) Took video for part 1 demo.
- (3) Coded parts 1 and 2, freeze, slow down, power-up reset, EEPROM persistent storage of high scores.
- (4) Wrote corresponding parts in final report.

6.2 James Houghton

- (1) Structured and wrote most of the part 1 code.
- (2) Structured and wrote most of the part 2 code, including the code for power-ups (Life, Crosshair Speed-Up, and Crosshair Size), highscore name-entering, and the leaderboard.
- (3) Wrote the portions of the report detailing the aforementioned Part 2 features.
- (4) Added support for bitmap drawing and a draft Life power-up bitmap.

6.3 Guy "Jack" Verrier

- (1) Wrote part of the report.
- (2) Created part of the slides.
- (3) Initial random number generator.
- (4) Created the dataflow diagram.

6.4 William Zhang

- (1) Implemented LFSR pseudo-random number generation.
- (2) Added more advanced crosshair movement code.
- (3) Created bitmaps for every power-up and for the basic cube.
- (4) Took video for part 2 demo.

7 CONCLUSION

By adding features to the Cube Game, we've made it more similar to real-world arcade games. Features such as the Highscore System, Power-Ups, and bitmap graphics contribute to the enjoyment and featureset of the embedded application. Importantly, the Highscore system contributes to replayability: users that can see their old scores and strive to beat them. A leaderboard is emblematic of this principle. By displaying to the player their greatness or not-greatness, it compels them to play again. Power-ups are also important and are a value-adding feature. Adding variability to each block adds texture to gameplay, further increasing replayability.

We also learned lessons during development, including the importance of discarding interesting ideas that wouldn't fit our development timeline. While even after expanding on the completed game, we're still far from the games available commercially today. This project enabled us to develop skills and learn the process of developing applications for real-time embedded systems and served as a satisfying and worthy final project.