Bit Dodger

By: Devon Merz, Tiger Yang, and Travis Shivers

Project Overview:

Starting off, we knew that we wanted to create an enjoyable and challenging game with simple and easy-to-use controls. Our inspiration was early platform video games. We wanted to make the animations aesthetically pleasing at retro-esque while the accompanying sound bits would both entertain and be fitting to the scenario. In order to do this, we began work on a project called Bit Dodger. This would be a retro "8-bit" game where the player tries to dodge falling bombs while collecting falling coins. To make the game more suspenseful, we added a countdown timer indicated by an LED transitioning from blue to red. If the time runs out, then the player loses. However, if the player collects coins, they get time back on the countdown timer. In fact, if the player collects 10 coins, they win the game!

Game Overview:

The screen consists of an 8 by 8 LED array. Our character, "green", starts the game at the bottom left of the matrix. A timer LED at the top cycles from blue to red, running out at red and causing a loss. Two buttons allow the player to move left or right along the bottom row. Bombs, "red", are spawned randomly along the top row of the screen, and fall down one spot at a time along the column which it spawned. Contact with a bomb deducts time from the player. Coins are also spawned randomly along the bottom row, contact with which adds time for the player. A win is triggered after 10 coins are collected.

Modes:

Before and after the game, we have LED animations and sounds that add to the user experience. Before the game starts, we have an intro animation which is a spiral of LEDs that changes color (Figure 1). This is accompanied by a repetitive and fun sounding noise, which entices the player to start the game. In order to start the game, the player can press either the left or right buttons. If the player wins, then the screen flashes green and plays a happy jingle (Figure 3). If the player loses by running out of time, the screen flashes red and a sad song plays (Figure 4). Alternatively, if the player loses by hitting a bomb, red lines form on the row and column where the collision occurred (Figure 5).

Interrupts:

In order to optimize the amount of time the board runs in low power mode, we utilized interrupts in order to wake the code up periodically and run necessary functions. Our first interrupt, the watchdog timer, reruns the main code periodically to check for new turns and win/lose conditions. The button press interrupt flags when the left or right button is pressed and includes debouncing so the button must be pressed for 4 cycles. Lastly, the SPI Buffer uses interrupts in order to sleep in between the transfer of bytes. While the buffer is updating with the new value to

transmit, the MSP goes into sleep mode. This prevents the SPI Transmit Buffer from becoming overloaded with more than one byte at a time.

Randomness:

Our module Rand8( ) is used in order to generate a random value between zero and seven. First, the module uses the on-board clock to generate a random seed. It then implements a Linear-Feedback Shift Register (LFSR) which transforms the seed into a new value every turn. It then takes the three Least Significant Bits (LSBs) to generate the random number. This number is used to determine in which column a new item will appear. It also is used to determine whether the item is a bomb or a coin. If the number generated is a zero, then the new item is a coin. Otherwise, if the number is greater than zero, then the new item is a bomb. In this way, we set the ratio of coins to bombs to be 1 : 7.

LEDs:

In order to keep the lights working, 67 bytes must be updated and transmitted each turn serially to the board:

- Byte 1: Start Frame - 0x00000000
- Byte 2: Time LED Frame - Ranges from Blue (Max) to Red (Min)
- Bytes 3-66: Display LED Frame- Green (Player), Yellow (Coin), Red (Bomb)
- Byte 67: End Frame - 0xFFFFFFFF

SPI:

The SPI buffer only has room to transmit one byte at a time; therefore, each byte in the LED output array must be stored and sent individually. Between each byte, the buffer updates and no transmission occurs.

Piezzo:

For each note, the MSP generates a square wave with 50% duty cycle to minimize presence of additional frequencies. The further the square wave differs from 50%, the more additionally higher frequency components beyond a sine wave there will be. The frequency of the square wave determined by frequency of note. For example, the note of A corresponds to 440Hz. Overall, the Piezzo module contains songs to play for each of the the 3 main animations: the start song, win song, and lose song.

Deep Dive on Coding Decisions:

We spent quite a bit of time making the code modular, easy to debug, and optimized for low memory. We tried to limit the scope of all of our variables so we wouldn't have a bunch of global variables sitting around in memory all the time taking up those precious 512 bytes of ram. We were quite worried about running out of RAM since we were addressing 65 LEDs, and storing the entire SPI frame for each of those LEDs would take 4 bytes each, totaling 256 bytes just for the frames. So, instead of storing the entire SPI frame, we heavily compressed the data down and made our own special 256 bit color space. This allowed us to store all the color information for all the LEDs in only a total of 64 bytes.

In addition we aimed for a separation of our game logic from the actual hardware interfacing logic to allow us to easily debug one vs. the other and see what was really going wrong when we were having issues with the program. For example, we decided to create a frame buffer to create this abstraction layer between our game logic. Our game simply writes whatever it wants to display to the framebuffer with a simple interface that we designed that could easily be extended to support a wide variety of games and graphical applications using the led screen. As a result, the game doesn't have to worry at all about the specific details of how the SPI works or interrupts at all. In addition, on the flipside, this means that our hardware interface logic doesn't have to worry at all about the game logic. It doesn't care at all what a bomb or a player is. It just will output whatever is in the framebuffer to the LED screen over SPI. This layered approach is incredibly elegant and saved us a bunch of debugging work. The framebuffer allowed us to do many things in relatively small amounts of memory and with greater computational efficiency than if we didn't have it.

We worked on implementing simple frontend APIs to the normally rather annoying ways of dealing with the asynchronous SPI communication with waiting for the receive interrupt before you can transmit again. Our main accomplishment with this was the creation of a simple SendSpiByte function that simply sent out the desired byte over SPI, and blocked until it was done. This means that any function in the code can use it without having to worry about what interrupts are. It's such an elegant interface that simplified our backed logic quite a bit.

We also made sure to keep our code quality to a high standard to make sure that we could easily read each other's' code and be able to troubleshoot it without having to spend unnecessary time trying to figure out what variables do what and what global state might be affected. Even though we wrote all of the code in C, we still tried to emulate as functional of a style as possible, since that is the purest form of programming.

We modeled our program like a rather simple state machine with well defined transitions and necessary contracts that each state must promise to do before transitioning to the next state, like cleaning up any global state or resetting or reinitializing any global arrays or frame buffers. Again, contracts like these helped ensure that we could work on different parts of the code at the same time and be more confident that our pieces would work together.

Appendix:

Figure 1: Start Mode

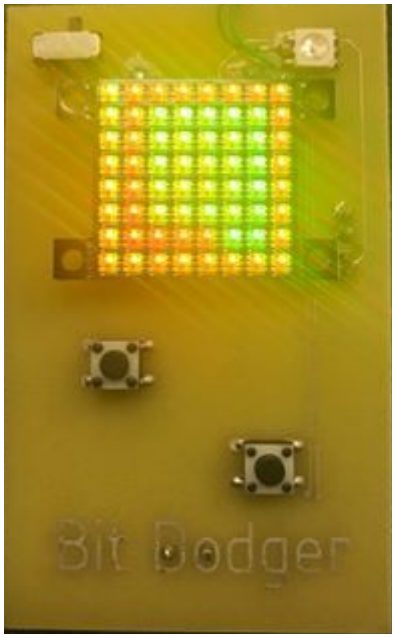


Figure 2: Game Mode

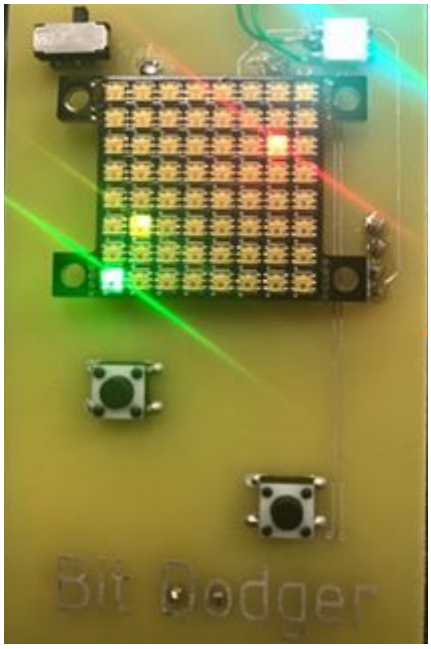


Figure 3: Win animation

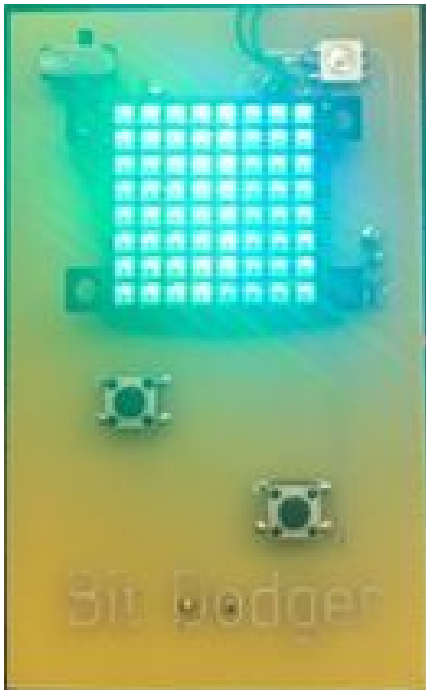


Figure 4: Time Loss

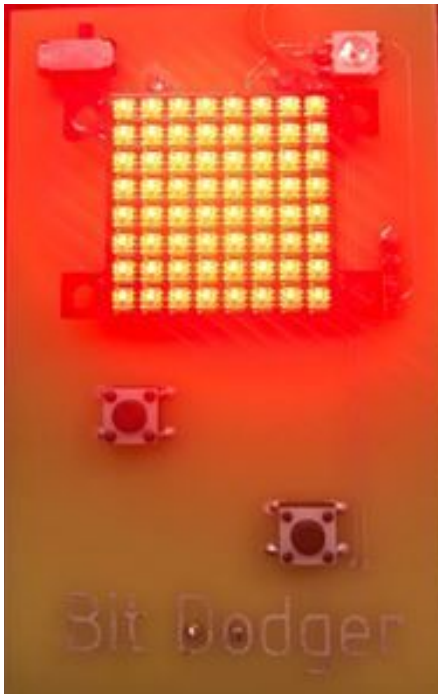


Figure 5: Bomb Loss

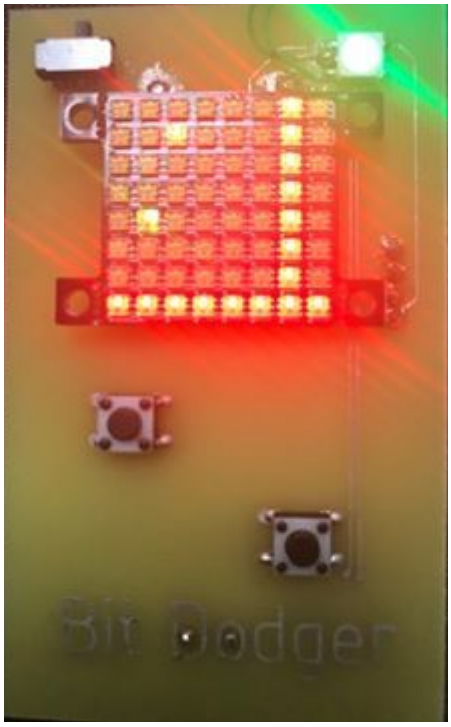


Figure 6: Front of PCB

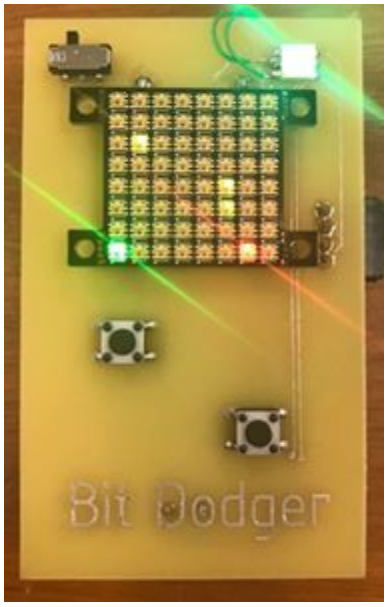
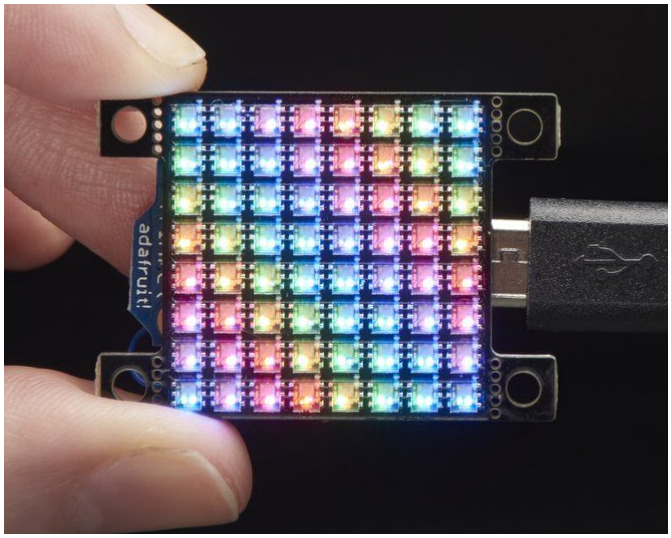


Figure 7: Back of PCB

.

Figure 8: DotStar Matrix



..

Figure 9: Components

1: Adafruit DotStar 8x8 LED Pixel Matrix
2: Buttons
1: DotStar LED
1: On/Off Switch
1: Piezzo Buzzer
1: Female Header
1: 47 kOhm Res.
3: 1 uF Caps.