Final Project: Arcade Baseball
Andrew Tsakiris (art85) and Nathan Stack (nts28)
ECE 3140 / CS 3420: Embedded Systems
May 17, 2019

Video Link: here

Introduction and Description:

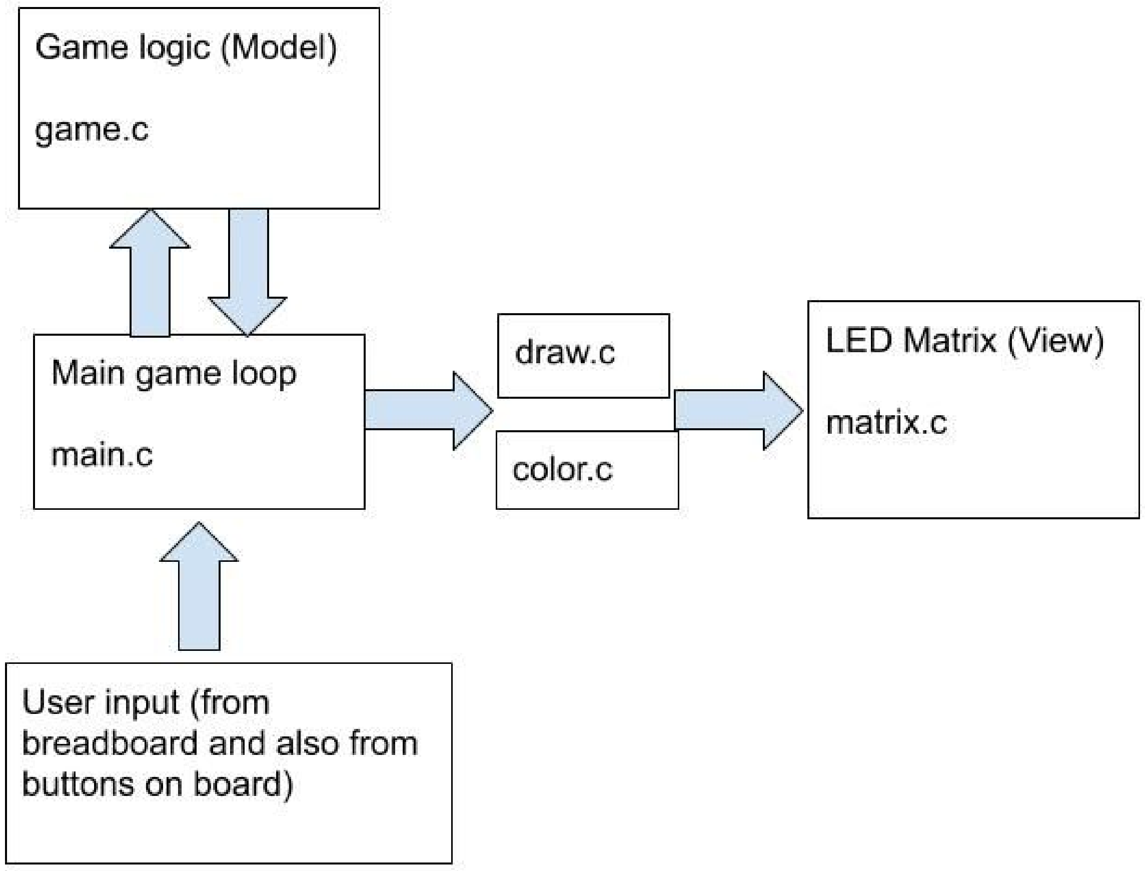
Arcade Baseball is an original arcade game inspired by electronic toys that many of us grew up playing. It is a game based heavily on strategy and skill, but has randomized features that make sure every game is competitive and unique.

Gameplay: Arcade Baseball is a 2 player game with a batter and a pitcher that alternate roles between half-innings. The batter has control of one button. The pitcher has control of a button and 2 switches (used to select 1 of 4 pitch types). The batter signals that he is ready by pressing his button. The pitcher then has time to select which of 4 pitches he would like to throw with his 2 switches (fastball, curveball, splitter, or knuckleball). When ready, the pitcher presses his button to initiate the pitch speed meter. The pitcher attempts to press his button while the white ball is in the green zone--the closer he is, the faster the pitch will go! Then, the pitch animation begins, and the batter swings (presses his button) while the ball is in the batter box. The result of the pitch is shown on the screen and the state of the game is updated accordingly. If one player reaches three outs, the two teams will switch sides (and controls) and continue playing until the 9 inning game is over.

Features:

- Hitting: the result of a swing is influenced by where the ball is in relation to the batter box when the batter swings. Different zones of the batter box are mapped to probability distributions that determine the outcome of the swing. For example, if the batter swings when the ball is on the edge of the box, an out or foul is likely, while a homerun or double is very unlikely. If the batter swings while the ball is in the middle of the box, big hits are more likely than outs and singles.
- Pitching: the speed of a pitch is determined by where in the pitch meter the pitcher stops the ball; the closer to green, the faster the pitch. The pitcher can also specify between 4 pitch types with his switches. Each of these 4 pitches has a different path. Additionally, there is randomness in each type of pitch. For example, not all curve balls will follow the same path each time. In fact, sometimes they will be strikes and sometimes balls! The batter must refrain from swinging at pitches that do not pass through the batter box.
- Baseball Rules Supported: Arcade Baseball simulates baseball in great detail. In addition to supporting all types of hits and advancing runners accordingly, the system handles normal strikes versus fouls. A batter cannot strike out on a foul ball. The system also updates runner positions on walks according to MLB baseball rules. Each player gets 3 outs per inning.

System Description and Diagram



Our software design roughly follows the Model View Controller design pattern. The main components of our software design are:

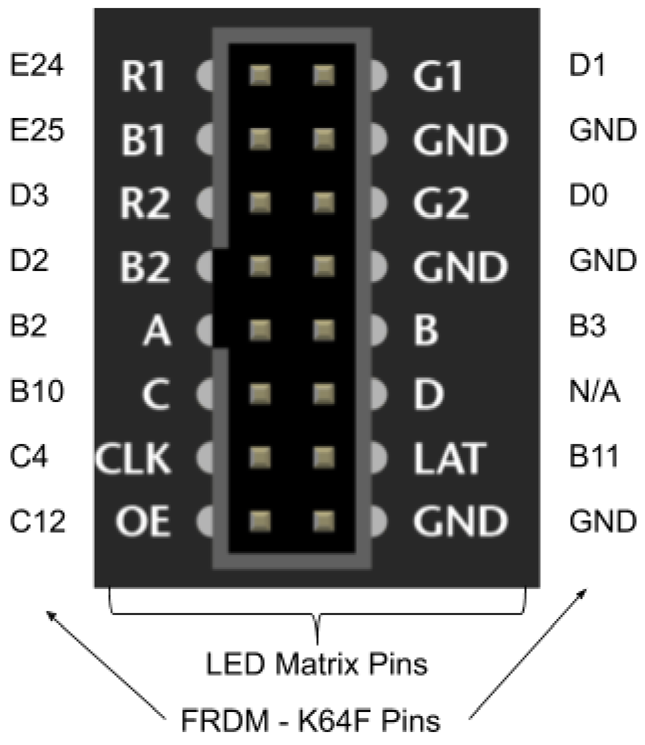
- Matrix.c: handles updating and displaying the matrix. The main game loop draws pixels onto the matrix and updates the display every 1/700th of a second.
- Game.c: handles the game logic of Arcade Baseball. Responsible for maintaining a game state (with balls, strikes, outs, etc) and updating the game based on results of pitches (strikes, balls, hits, fouls, etc).
- Main.c: drives the main game loop and interprets user input. Triggers animation of the game play during pitches, makes calls to game.c to update its internal game state, etc.
- Pins.c: configures the pins for the LED matrix and for input and output from the board.
- Draw.c: has useful functions that draw more complicated shapes onto the LED matrix through successive calls to drawPixel (defined in matrix.c).
- Color.c: defines color_t struct to keep track of a color represented by R, G, and B values. Matrix.c stores an array of these colors internally to use to light up LEDs on the matrix.

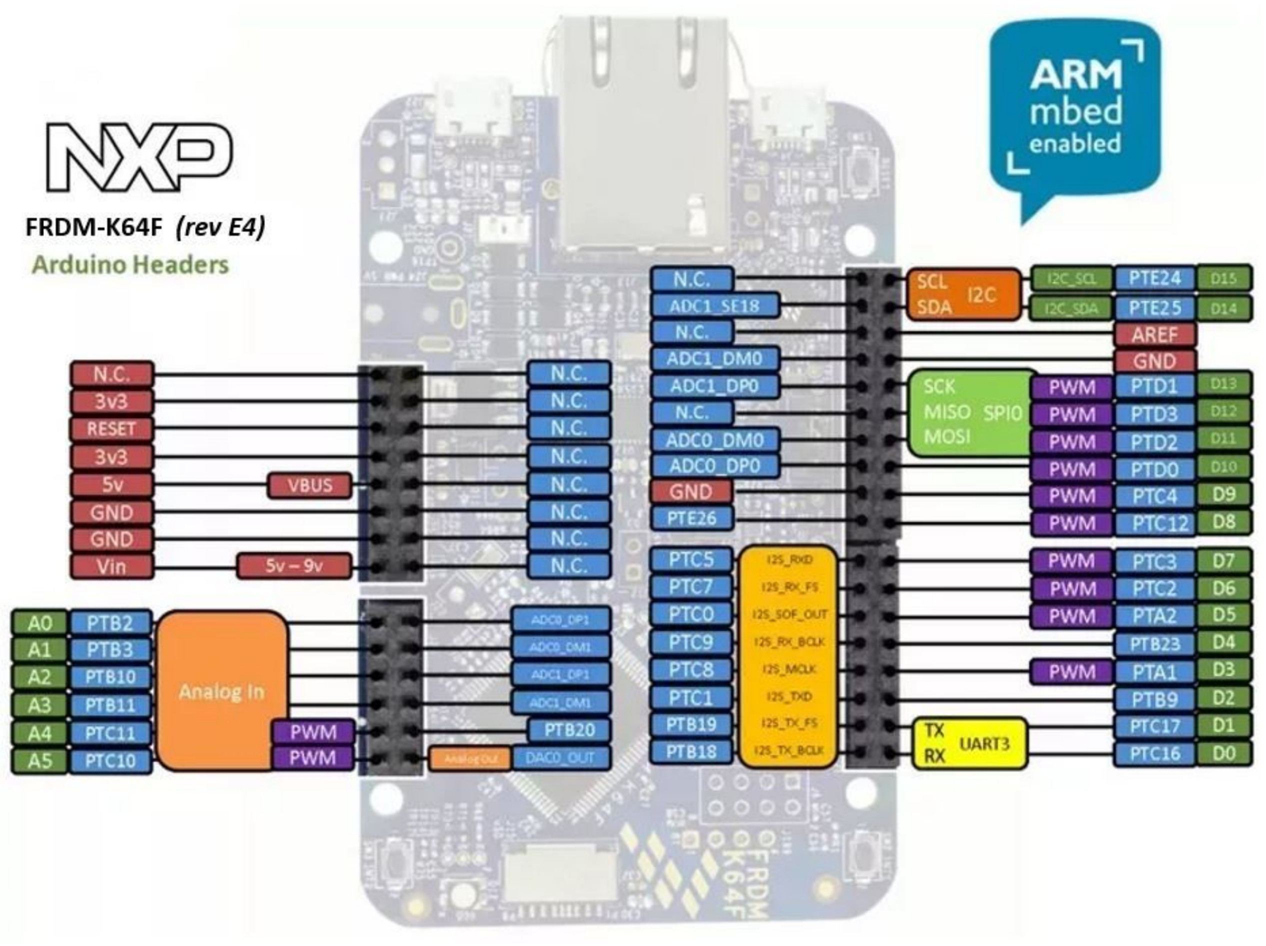
Hardware description

LED Matrix / Board Connection

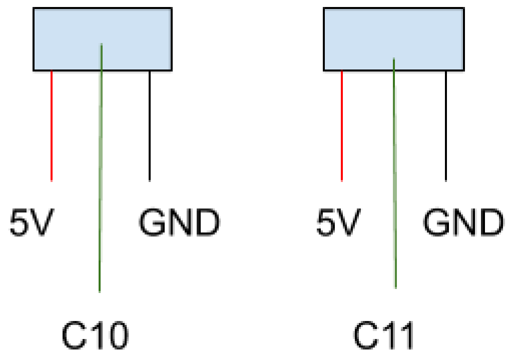
The LED matrix interfaces with the FRDM K64F through simple GPIO. The diagram below shows which input pins on the LED board the output pins of the FRDM are wired to.

Note: This wiring configuration was borrowed from Michael Xiao and Nathaniel Diamond’s project from Spring 2018. We did not see a reason to change the configuration of the pins. They created a protoboard that connected the pins of the FRDM to a ribbon cable, which in turn connects to the LED Matrix. We used the exact same method, and therefore we do not take credit for it.





4 more connections were added to the protoboard to support the pitch-selecting switches. These switches are two independent SPDT switches. The switch either connects the switch pin to ground or to power through simple GPIO. On the FRDM K64F, the switches are connected to C10 and C11, where binary input is read through GPIO.
Additionally, SW2 and SW3 are used for batter/pitcher input directly on the board. These buttons are connected to port C pin 6 and port A pin 4.



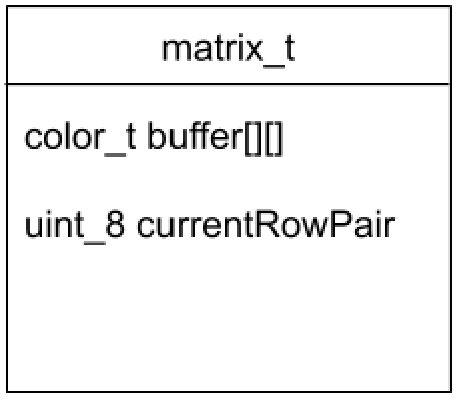
Bill of Materials

Item	**Catalog #**	**Quantity**	**Price**
FRDM-K64F	Mouser: 41-FRDM-K64F	1	$35 (provided)
16x32 RGB Adafruit LED Matrix	Adafruit Product ID: 420	1	$24.95
Breadboard-friendly SPDT Slide Switch	Adafruit Product ID:805	2	$0.95
BusBoard Prototype Systems BB1460 (Breadboard)	Mouser: 854-BB1460	1	$14

<u>Detailed Software Description</u>

LED Matrix

Interaction with the LED matrix is implemented mainly in matrix.c. The configuration of all the pins described in the hardware description are configured in pins.c and initialized in the main game before any interaction with the LED matrix occurs. The main data structure representing the matrix is shown to the right. Importantly, the buffer of colors is a two dimensional array containing the color (a struct with r, g,b values) at each pixel in the LED matrix. Buffer is 16x32.



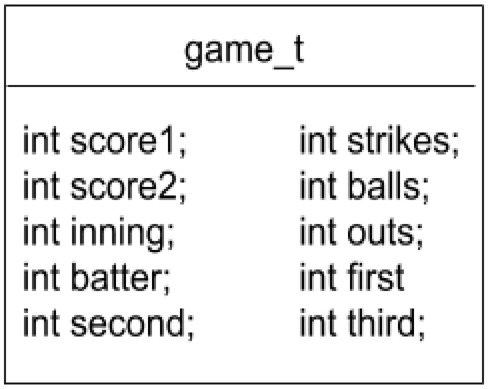
matrix.c contains two important functions to interact with the matrix: drawPixel() and updateDisplay().

- drawPixel(row, col, color): drawPixel simply updates the internal state of the buffer. We found that there were some peculiarities in the matrix display (row 0 was being drawn on row 7, for example) so we fixed that peculiarity in this method. The effect, however, is that a user can update the internal matrix state at (row, col) with a specified color (type color_t has three integer fields for r, g, and b).

    - There is also a *fillRect* function that makes repeated calls to drawPixel() to draw a specified rectangle.
- updateDisplay(): this method is called repeatedly to actually light up the LEDs on the matrix. The LED's that get lit up are defined by the buffer. We call this method repeatedly (700 times a second) via a PIT Timer interrupt. The matrix is configured so that it can only drive 2 rows of LEDs at once. Therefore, each call to updateDisplay() actually only lights up 2 rows of LEDs. However, we cycle all possible row pairs with an internal counter in matrix_t (currentRowPair) and increment this counter by 1 with each call to updateDisplay(). This is how the function works:
    - First, the row pair selected is inputted to the matrix with the A,B,C pins (each binary, can represent up to 8 row pairs. Makes sense since there are 16 rows).
    - Then, we send a series of packets of 6 bits, clocking in between each packet. The 6 bits correspond to data for one individual column for these 2 rows (i,e. 2 pixels). Each pixel has 3 bits (R1, G1, B1 for the "top" row in the pair and R2, G2, B2 for the "bottom" row in the pair). Whether to send a 1 or a 0 for each bit is determined by the value of the color in the buffer at that location.
    - Latch the data and then increment the row selector by 1.
    - Note: we only support 0 or 1 for each RGB value, not the full 0-255 RGB color range.

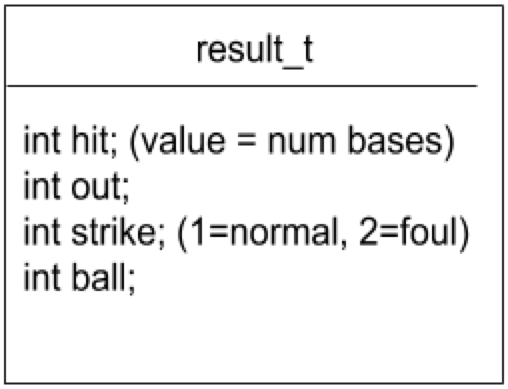
Game State + Logic

The game state is represented by a game_t struct. All of the game logic to create and update this state based on the results of at-bats are handled within game.c. This is the "model" part of the Model-View-Controller design pattern. In draw.c, drawState() updates the LED matrix based on this state. Another important type is result_t, which represents the result of an at-bat. The function updateGameState(game_t, result_t) updates the state of the game based on the result. This function takes care of runs being scored, runners advancing bases, and all of the traditional baseball rules described in the first section.



game_t

```
int score1;      int strikes;
int score2;      int balls;
int inning;      int outs;
int batter;      int first
int second;      int third;
```



result_t

```
int hit; (value = num bases)
int out;
int strike; (1=normal, 2=foul)
int ball;
```

Main Game Loop

Arcade Baseball is essentially a looped sequence of events. Before entering the loop, the interface with the LED matrix, tactile switches, and on-board push buttons are configured. An initial game state is created and drawn to the matrix's buffer. Then, a PIT timer is enabled and on each interrupt calls updateDisplay() so that the LED matrix continually updates and shows the appropriate information on the screen. The rest of the game is the game loop:

1. Use drawState() to draw the current game state. Busy wait for batter interrupt.
2. An interrupt is fired when the batter presses his button, which sets a global variable to a specific value, breaking the main processes out of its busy wait. The at-bat screen (pitcher's mound and batter box) is displayed and the main process enters another busy wait.
3. An interrupt is fired when the pitcher presses his button, signaling that he has selected his pitch with the 2 switches, and is ready to pitch. This prompts a ball to appear and begin moving back in forth in front of the meter. This is done by drawing the ball in a location, busy waiting, and then erasing the pixel the ball was in. Then the ball location is changed and the animation loop runs again. At this time, the values of the switches are read and used to determine the pitch path.
4. An interrupt is fired again when the pitcher presses his button, and the location of the ball in relation to the meter is saved by the interrupt handler. This value is used to set the speed of the pitch.
5. The ball then flashes 3 times and follows the path determined by the create_pitch() function. The create_pitch() function uses the values of the switches to create a pitch. The pitch is represented as a two dimensional array of size (100,2). The array is essentially a list of tuples, where each tuple in the list represents the change in row and col for that given frame of animation. The main loop then animates the pitch as in part 3, using the i'th tuple in the pitch array to update the location of the ball in the i'th frame of animation to the proper location in the i'th+1 frame.
6. An interrupt is fired from the batter pressing his button, and the location of the ball is saved by the interrupt handler. This also stops the pitch animation.
7. A result is produced by produce_result() based on the position the ball was in when it was stopped by the batter interrupt. A ball closer to the center of the batter box has a higher probability of being a good hit than a ball far from or outside the batter box.
8. The result is displayed on the LED matrix and then busy waits for about 1 second.
9. The internal game state (game_t) is updated via updateGameState(), and then the new state is drawn onto the board via drawState() and the loop continues until the stopping condition (last out in the 9th inning) is reached.

Use of Interrupts

- Updating the Game State. In order to see a continuous display, a call to updateDisplay() must be made several hundred times per second. To do this, we enable a timer interrupt on PIT channel 0 to fire 700 times per second. The interrupt handler is simple: just reset the interrupt flag and call updateDisplay(). We used this timer interrupt because it allows us to just start the timer and then not have to worry about explicitly updating the display throughout the game loop. Instead, the matrix will always be automatically displaying its current contents.

- Pitcher and Batter Buttons. We use interrupts on Port C and Port A to detect the rising edge of the two buttons on the FRDM-K64F board. Interrupts are useful here because we can have one loop running for an animation of the ball moving and have the button interrupt change a global variable at any time to stop the animation and save the location of the ball when the button is pressed. Polling the switches continuously would have been cumbersome to do in the same animation loop, and we could have lost precision in the recorded reaction times if we did so.

Use of Polling

- Pitch Select Switches. The switches on the breadboard have no interrupts associated with them, but rather their values are just read after the pitcher presses his button to signal he is ready. We wanted the pitcher to be able to adjust his pitch selectors whenever he desired (so the batter would not see him changing them). The program does not care about every time the switches are moved, but rather all that matters is their state when the pitcher presses his button to signal he is ready. It is therefore much easier to just poll the switches once for their current state after the button is pressed.

Testing

All of our testing was done incrementally, mainly through visual output on the LED matrix once it began working, and also through the debugger, mainly by tracking variables and their values.

- LED Matrix: we wrote many test programs to ensure that drawing and updating the LED matrix worked (in test.c). For example: animating a rectangle moving back and forth across the screen, filling the entire screen with different colors, etc.
- Game Logic: the main point of testing was to test the transition between states based on results. To do this, we wrote a function to display the state on the LED board. Then, we made the results of at-bats hard-coded as particular results, so we could ensure that the expected behavior would occur. We also fed the game starting states that were not empty (i.e. states with runners already placed on base, with 2 outs, etc to facilitate testing every part of the game logic).
- Input Buttons: we used the debugger extensively to test interrupts with the input buttons on the board. We used watch variables to keep track of when interrupts were being triggered.
- Switches: we used the debugger to watch variables linked to the value of the switch input before integrating them into our game.
- Animation + Gameplay: we tested actual hitting and pitching entirely by actually playing our game. The game is in one large loop, so we really only needed to test one pitch at a time. Once one pitch and hit works, it followed logically that a series of pitches would work as well. We continually tested the pitching and hitting mechanisms with different inputs to detect if they worked.
- Final testing: we played the game several times, trying all sorts of different inputs to see if there were any bugs. We also let other people play the game to see if they could

introduce bugs. We found edge case bugs (like when buttons were pressed when we were not expecting them to be) from this and were able to fix them accordingly.

Results and Challenges

We achieved what we proposed! The most difficult part of the project was certainly configuring the LED matrix. At first, we attempted to use mbed and other libraries to interface with the board. This worked up until the point where we tried to use interrupts with input buttons. This interfered with the matrix in a way we did not understand, so we elected to write the configuration and interface with the matrix from scratch. Once we got this working, the project moved extremely quickly with the exception of a few bugs. The most significant was a repeated hard-fault being caused by accidentally re-allocating space for color structs every time a pixel was drawn. This was fixed by creating a couple of global colors and reusing them via references. It took very long to find this bug however. Next time, perhaps we would have added more complicated peripheral devices (like a joystick or multi-option button) that used more complicated interfaces than GPIO. This likely could enhance our game and also be a good educational experience. Overall, we are very happy with how Arcade Baseball came out, and it exceeded our expectations.

Work Distribution

We met several time to brainstorm ideas for our project and the features we wanted to include. The first part of the project was completed via pair programming. This first part was simply getting the LED matrix to work. We met several times to work on this in person. Once we got the LED matrix working, we roughly divided the work as follows:
Andrew - in charge of writing the game logic (separate from any display). Creating different types of pitches, hits, etc. Logic to map certain results to different hits/pitches.
Nathan - displaying game state on the board. Animating on the board. Creating logic for pitch-select switches.
We also worked on configuring interrupts and the push buttons on the board together.
We completed most of the project together in the Maker Space in Phillips Hall. All soldering and hardware setup was done together in the Maker Space. We used Github for our code and Google Docs for planning and this report.

References

Michael Xiao and Nathaniel Diamond Project from Spring 2018
https://confluence.cornell.edu/display/ece3140/LED+Matrix+Arcade+Game+Suite
FRDM K64F Reference Manual
http://cache.freescale.com/files/microcontrollers/doc/ref_manual/K64P144M120SF5RM.pdf
FRDM K64F User Guide
http://cache.freescale.com/files/32bit/doc/user_guide/FRDMK64FUG.pdf
Adafruit LED Matrix “How it Works” + Tutorial
https://learn.adafruit.com/32x16-32x32-rgb-led-matrix/how-the-matrix-works
ECE 3140 / CS 3420 Lab 2, specifically dealing with interrupts and GPIO configuration.