SNAKE

Kevin Ying (kzy2) and Morena Rong (mr942)

Introduction

Our project implements the classic Nokia phone game, SNAKE using the FRDM K64F board and an LED matrix.

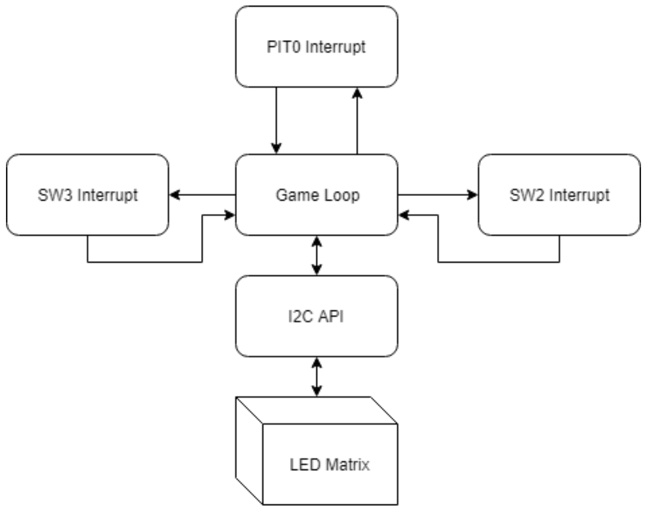
Users can play the game SNAKE on a two-dimensional LED matrix. The snake moves automatically approximately every 0.1 seconds and in a forward direction unless the user dictates it to turn left or right. One food item is on the board at all times and the snake can eat the food and increase in length if it collides into it. The goal of this game is to get to the snake to the longest length possible without colliding into itself or the border of the board.

High Level Description

The SNAKE game is a simple two-dimensional video game, popularized by its inclusion on early Nokia phones. The player controls a snake, which is a contiguous chain of grid spaces on a two dimensional grid. The snake is constantly moving forward and the player is able to control it by indicating when to turn left or right. Once a player indicates turning left or right, it then continues to move in the new forward direction, with each segment of the snake following the path which the head took such that every segment moves exactly one grid space every time step. Additionally on the grid there is always one food item which is one grid space, where if the snake “eats” it the snake grows longer by one grid space. The snake initially start with length 2. The food item then randomly regenerates in an unoccupied location every time the previous food item is eaten. The player loses the game when the snake dies by either colliding into the border of the grid or with itself.

We created our version of SNAKE using the FRDM-K64F board with its hardware buttons and an LED screen. The snake moves on a timer set to approximately 0.1 seconds. The two buttons on the board allow the player to control the snake to turn left or right, and the GUI with the snake and food item are displayed on a LED matrix board.

System diagram



(Figure 1. System Diagram)

The game consists of a large loop which initializes and deinitializes game state variables as needed. It is periodically interrupted by PIT interrupts, which causes the snake to move one step forward. Button inputs also cause interrupts, which change the way in which the snake moves on the next update. All changes to the game's state are then communicated over I2C with the LED Matrix.

There is also another periodic timer which acts as a random number seed generator.

Major Design Decisions

Use of Adafruit IS31FL3731

We used the Adafruit IS31FL3731 because of its low cost, small profile, and the use of I2C. Since the IS31FL3731 has a microcontroller to implement PWM for its attached LED Matrix, it allows for ease of construction of the hardware circuitry, decreasing the latency in the number of communications needed between the FRDM K64F and its display. We took advantage of the uVision I2C libraries to increase familiarity with the I2C protocol, while while still allowing for robust communication with the IS31FL3731.

We chose a baud rate of 100kHz in order to increase the smoothness of game transitions while remaining firmly within the limits of the driver hardware. This is especially important because the I2C API used is non-blocking, so that function calls will not return until the IS31FL3731 acknowledges a data transfer. This means that if information is sent to the driver more quickly, the program stalls for less time within interrupt service routines and away from processing other inputs, a good general coding practice.

Buttons

We used the on-board switch buttons as the main user inputs to the SNAKE game. Our goal was to have convenient inputs which the user would be familiar with. The use of onboard buttons also adds to the slim profile of the SNAKE game. To give both button inputs the same priority, we disabled the NMI feature on the K64F. In order to make the most of the limit hardware buttons available, we chose to have the buttons indicate a relative direction, such that no matter which direction the snake is moving, the button on the bottom left of the K64F marks the intention to make a 90-degree counterclockwise turn, and the bottom-right hardware button marks the intention to make a 90-degree clockwise turn. This allows for the left and right button positions to mirror the direction that the snake will turn to, if the snake's current heading is considered forward.

Interrupt-Based Code Structure

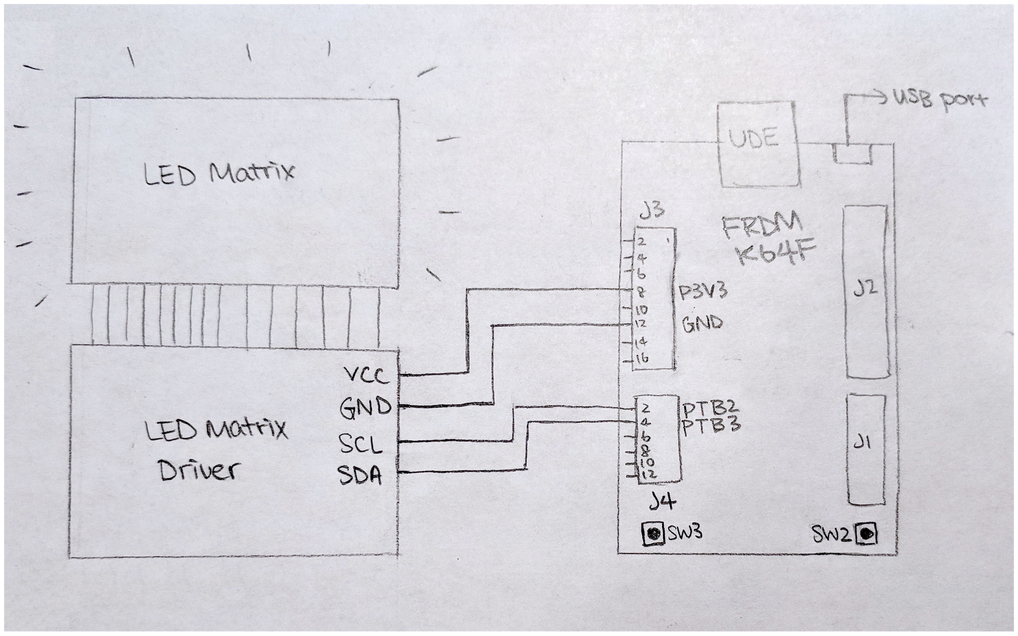
Since the actions in the game are generally event-driven, we wanted the central thread of our code to process as little information as possible. Instead, our design philosophy was to separate out actions into separate flags which would be set and interpreted during interrupt handlers. This allowed us to use an event-based mindset without the additional rigidity of implementing a state machine. The specific value loaded into PIT0 to drive changes to the game state was chosen to facilitate smooth game play and set a reasonable level of difficulty.

Use of Counter as Seed

In order to generate random placements for food locations, we used PIT1 to generate an interrupt every 5ms which kept track of the time elapsed. This allowed us to generate random seeds based on the time of the first button input without requiring access to other on-board sensors. The value of 5ms was chosen to balance any potential lag of processing the ISR with having the greatest number of seeds which would be chosen in normal gameplay.

Hardware Description

Our external hardware is connected to the FRDM board and to the power source as shown in the schematic below.



(Figure 2. System schematic design)

In addition to the FRDM K64F board ($37.19), we purchased a LED Matrix Driver and the corresponding cool-white LED Matrix from Adafruit. To connect these to our board, we also acquired a breadboard, wires, and solder.

LED Matrix Driver:

- Adafruit 16x9 Charlieplexed PWM LED Matrix Driver - IS31FL3731 (Product ID: 2946)
- Price: $5.95
- Link: https://www.adafruit.com/product/2946

LED Matrix:

- LED Charlieplexed Matrix - 9x16 LEDs - Cool White (Product ID: 2974)
- Price: $7.95
- Link: https://www.adafruit.com/product/2974

Detailed Software Description

To communicate with the peripherals used in this project, first we set up the pins for the switch buttons SW3 and SW2 to be used as GPIO's. A button press will therefore call the interrupt request handlers for the specific input and we implemented the specific functionality of button presses in these IRQHandler functions. I2C pin modes and protocols were used to connect the LED matrix and LED matrix driver. A more detailed description of how we did this can be found under the How-to article for using I2C for the K64F.

In designing structures to hold information for abstract structures, we created two of our own data structures: snake_part_type and food_item_type. The snake_part holds information about a single segment of the snake, including its coordinates and the previous and next segments of the snake it is attached to. In essence, the entire snake structure was kept in a doubly-linked-list, where each entry of the list contains a pointer to the next and previous entry of the list. The food_item contains the coordinates of the food item currently on the board. To update the coordinates of the a new food item to be in a random, unoccupied grid space, we also made use of empty_spaces, which is a list of which coordinate location is not occupied, and is therefore a valid spawning location of the new food.

The main routines that are code uses are init_game_state(), game_loop(), move_food(), the process of moving the snake in the next direction, and collision detection.

init_game_state()

- Initializes the game state in the beginning of a new game. This function creates the initial snake of length 2 in the middle of the board and generates a food item in a random location, ready to start the game.
- Resets the empty_spaces list

game_loop()

- Called every timer interrupt and updates the game state to reflect a single iteration of movement in the snake.
- Determines whether or not the game is over using the collision detection routines for the snake's body and the grid borders (body_collision() and border_collision().
- Determines whether or not the snake has "eaten" a food item using the food collision detection routines (food_collision()). If it was, then ensuring the snake's length increases and a food item regenerates.
- Moves the snake in the direction that was intended by the user using helper functions such as new_head_coord(), remove_tail(), and move_to().

move_food()

- Generates a new food location, which is guaranteed to be empty at the moment of regeneration.
- Using the empty_spaces list and generating a random number, finds an unoccupied empty grid space and sets the food's new coordinates to be that random space.

The majority of the software components were written by the group members. The only components that were pulled from outside sources and ported into our project's code were the Adafruit GFX library for Arduino and the Adafruit IS31FL3731 library for the LED matrix driver that we had used. The code from the library was modified to fit our needs and work with our FRDM board.

Some software choices we had made but did not end up working included having an extra data structure for coordinates and attempting to use the system time to generate seeds for the random number generator. We ended up not using a coordinate type because we realized it was unnecessary for both collision detection and having the snake move, and would further complicate our code. Additional steps to malloc() and free() coordinate types would also slow down the execution of the code. In order to generate a different seed for srand(), which we used to locate random unoccupied spaces for food, we first attempted to use the time library. However, we realized that the FRDM board itself does not have an operating system and therefore does not have system time. Instead, we added a second PIT timer to count up and used this as the seed.

Testing

In general, we tried to test each component of our software as individually and incrementally as possible. In order to test incrementally, we created test cases for each of the peripherals. For example, for the switch buttons, we initially had the IRQ Handlers toggle LEDs on every interrupt, meaning that we were able to physically see the operation of the buttons. It also highlighted if there were debouncing issues, since sometimes a single button press would cause multiple toggles of the LED.

For testing the operation of the LED board, we wrote a test case which looped through every pixel and incremented a counter variable which was then assigned to each LED as a brightness (mod 256). This allowed to us to clearly see when I2C started to work with a clear visible cue.

For testing the operation of the snake game mechanics, we modified the game code to hardcode certain variables such as *turn, game_started* an d the initial head, tail, and food positions. We then ran the program in debug mode in order to watch the values of variables affected by the new additions to our code. In addition, once we began to have a moving snake, we began writing test cases which hardcoded score values, in order to test the functionality of the draw_num function on the LED matrix.

After the point of having a flashable game, testing mainly took the form of running the program on the board to see how the game would react to specific events, and individual test cases were phased out. For example, one common testing tool was eating food. This allowed us to make sure that a new food item was being placed in a new location, and was placed in particularly high-valued and low-valued locations, indicating that the random number generator was working properly, as well as that the score was being incremented properly and displayed at the end of the game. Testing also often included playing the game multiple times in a row without resetting, which allowed us to check that the empty_spaces variable was being reset properly and no memory leaks were occurring.

Results

Our final project culminated in a working and version of SNAKE, with the graphic user interface displayed on a 16x9 LED matrix. The snake is user-controlled by the two hardware buttons on the FRDM board that indicates a left or right turn for the snake to take. We had successfully implemented and combined the game design involved with SNAKE with many of the concepts involved with embedded systems, such as using I2C peripherals, using interrupts and interrupt handlers, and creating concrete data structures in C to represent abstract structures and hold data. Overall we had achieved what we had proposed for this project with the exception of our idea of expanding this project and implementing a tilt-controlled SNAKE using an accelerometer. An additional feature that we added to our project involves flashing a blue and red LED on the FRDM board itself when the left and right switch buttons are pressed, respectively. This allows the user to know their button press was properly processed and is also a visual cue indicating which direction the snake will be turning. We evaluated our design mainly qualitatively. We adjusted the timing of the PIT0 timer and the brightness of the LEDs based on both visual appeal and feedback from play testing.

The following video shows two rounds of game play without resetting the board. The first round is played entirely by turning left and ending in a collision with the snake's own body. The second round is played entirely by turning right and ending in a collision with the border.

(Figure 3. Video of two rounds of game play)

The only safety concern for our design is the brightness of the LEDs causing eye damage and the sharp header pins that are used to connect all the pieces of hardware together. To ensure safety, we trimmed the prongs of the pins down and lowered the brightness level of the LEDs such that they are not too bright.

Furthermore, we implemented the game such that the gameplay and visuals are fluid and natural. The user interface is extremely usable. The brightness of the LEDs were chosen to be clear and bright but not too bright to view for an extended period of time. The food items are displayed brighter than the snake to provide contrast and engender a sense of satisfaction when the snake "eats" the food. Once game is over, the user's score for the game are displayed in a seven-segment display style. The option to immediately start a new game by pressing either of the switch buttons rather than resetting the board further adds fluidity in gameplay, allowing the user to start a new round as soon as they die.

Conclusions

Overall, the implemented SNAKE game met all of our base expectations. The gameplay feels fluid and can become natural with practice, and the score display engenders feelings of competition. Particularly challenging was working through the implementation of I2C on the K64F for the first time, the implementation of the snake as a linked-list to save the number of required operations, and the use of random numbers for generating new locations for the food object.

If we were to continue work on this project, the highest priority would be to increase the user interface usability for the game. As we discussed in our proposal, we would have been excited to work with the accelerometer or more hardware buttons in order to make the movement system of the game more intuitive. In its current state, it can be difficult for new players to get used to using both hardware buttons, a problem which would be alleviated by the use of four peripheral buttons which would remain in an orientation which directly correlates with the desired movement direction of the snake, instead of a relative orientation. Other portions of our design which would be improved would be the use of high score tracker across the rounds, as well as the ability to select between various difficulties for the game, as differentiated by the duration between movements of the snake (which could also increase as the snake length grows). It would also be useful to better take advantage of version control software, such as GitHub.

Intellectual Property

Further work on this project would be unlikely to be hindered by intellectual property considerations. The Adafruit GFX library has a very open BSD license, and code from the Adafruit IS31FL3731 library was never directly copied without modification. All parts used are publicly and commercially available. However, due to the simplicity of the project, it is unlikely to be advanced enough to allow for patent or publishing opportunities.

Work Distribution

Once we had decided what wanted to do for this project, we selected the external hardware to use to display the SNAKE game. The project was built up incrementally between the two partners in parts. Some of the major components that constituted the project includes modifying the Adafruit LED Matrix library code to compile and work with the FRDM board, implementing peripherals and interrupts to work together seamlessly, creating the game structure, game state, and software components to allow game play, and creating a method to generate new food items in random unoccupied locations.

Both partners had an equal opportunity to play an active role in all major tasks. Pair programming was used extensively throughout this project. While one partner may be typing up the code, the other partner would be overlooking the code, while simultaneously brainstorming methods to implement new components or improving what we already had. If for a portion of the project was created while the other partner was not physically next to the other while typing it up, both partners would come together as soon as possible and gain a good understanding of the code. In determining what components each partner chose to spearhead, we played to our strengths and Kevin chose to take the lead on more hardware design related components and Morena chose to take the lead on more software design related components. We communicated both in person and online through Facebook Messenger and shared our code through a Google Drive. We mainly used Kevin's laptop because his had Windows 10 operating system while Morena has Mac OS X.

Specific tasks carried out by Kevin:

- Soldering and wiring external hardware to our FRDM board
- Timer and button initialization and resolving common errors with switch button use
- Modifying and condensing the LED Matrix's Arduino library code to use for our FRDM board
    - I2C initialization and helper functions
    - board_enable()
    - draw_pixel(int x, int y, int brightness)
    - clear()
- Creating the numbers display functions to display the end game score on the LED matrix (i.e. draw_num(int num, int brightness))
- Generating a new food item in a random, unoccupied grid location on the matrix (move_food())
- Writing this report:
    - High Level Description
    - Testing
    - Conclusions
    - References, Peripherals, Software Reuse
    - Additional I2C and LED Driver/Matrix reference pages

Specific tasks carried out by Morena:

- Choosing and ordering the external hardware
- Forming the game play logic and writing this out as the game state code
    - intial_game_state()
    - collision detection methods: body_collision(), border_collision(), food_collision()
    - game_loop()
- Creating game play structures and related methods
    - snake_part and food_item
    - remove_tail()
    - move_to()
    - new_head_coord()
- Interrupt handlers and main method's logic and code
    - Allowing for new games after a game over without having to reset the board
- Writing this report:
    - Introduction
    - Hardware Description
    - Software Description
    - Results
    - Work Distribution

References, Peripherals, Software Reuse

Datasheets and References

- FRDM-K64F Freedom Module User’s Guide, User's Guide, Rev. 1, 08/2016
    - Provided in class documents
    - Available at https://www.nxp.com/docs/en/user-guide/FRDMK64FUG.pdf
- K64 Sub-Family Reference Manual, Rev. 2, January 2014
    - Provided in class documents
    - Available at http://www.mouser.com/ds/2/813/K64P144M120SF5RM-1074828.pdf
- Looking at this guide to I2C Non-blocking Communication was equal parts interesting and distracting, but did give further insights to the operation of I2C so it is included.

Peripherals

- Adafruit IS31FL3731
    - The Adafruit IS31FL3731 is a driver for a 16x9 board of Charlieplexed (multiplexed) LEDs, each of which is individually dimmable over I2C. It is also the driver for the 15x7 Charlieplexed Featherwing LED matrix, which can be found at https://www.adafruit.com/product/3134.
    - Datasheet: https://cdn-learn.adafruit.com/assets/assets/000/030/994/original/31FL3731.pdf?1457554773

    - Vendor site: https://www.adafruit.com/product/2946
- LED Charlieplexed Matrix - 9x16 Cool White LEDs
    - These are 9x16 Charlieplexed LEDs designed to match with the Adafruit 16x9 Charlieplexed PWM LED Matrix Driver - IS31FL3731.
    - EagleCAD: https://github.com/adafruit/Adafruit-IS31FL3731-CharliePlex-LED-Breakout-PCB
    - Vendor site: https://www.adafruit.com/product/2974

Code

- Code was borrowed from course lab documents. In particular, utils.c and utils.h draw heavily from their Lab 3 counterparts.
    - Some other parts of GPIO or PIT initialization and the use of a current time in milliseconds were also copied from previous lab assignments, both student and instructor code.
- Code was ported from the Adafruit IS31FL3731 library, as well as the more general Adafruit GFX library for Arduino
    - Most code had to be heavily modified to work with C instead of C++, as well as to interface with the K64F instead of an Arduino. In particular, the code for initialization and sending data over I2C needed to be heavily modified. However, segments of the code, such as most of the body of draw_line, were directly copied with minor changes.