Michael Tin (862321571), James Krejci (862300183)

EE128 Final Project - “Whack-a-Mole” Game

Section 021 | TA: Ziliang Zhang

I. Project Description

This Final Project is a rendition of “Whack-a-Mole”, which is a popular arcade game where a player scores points by hitting pop-up targets.

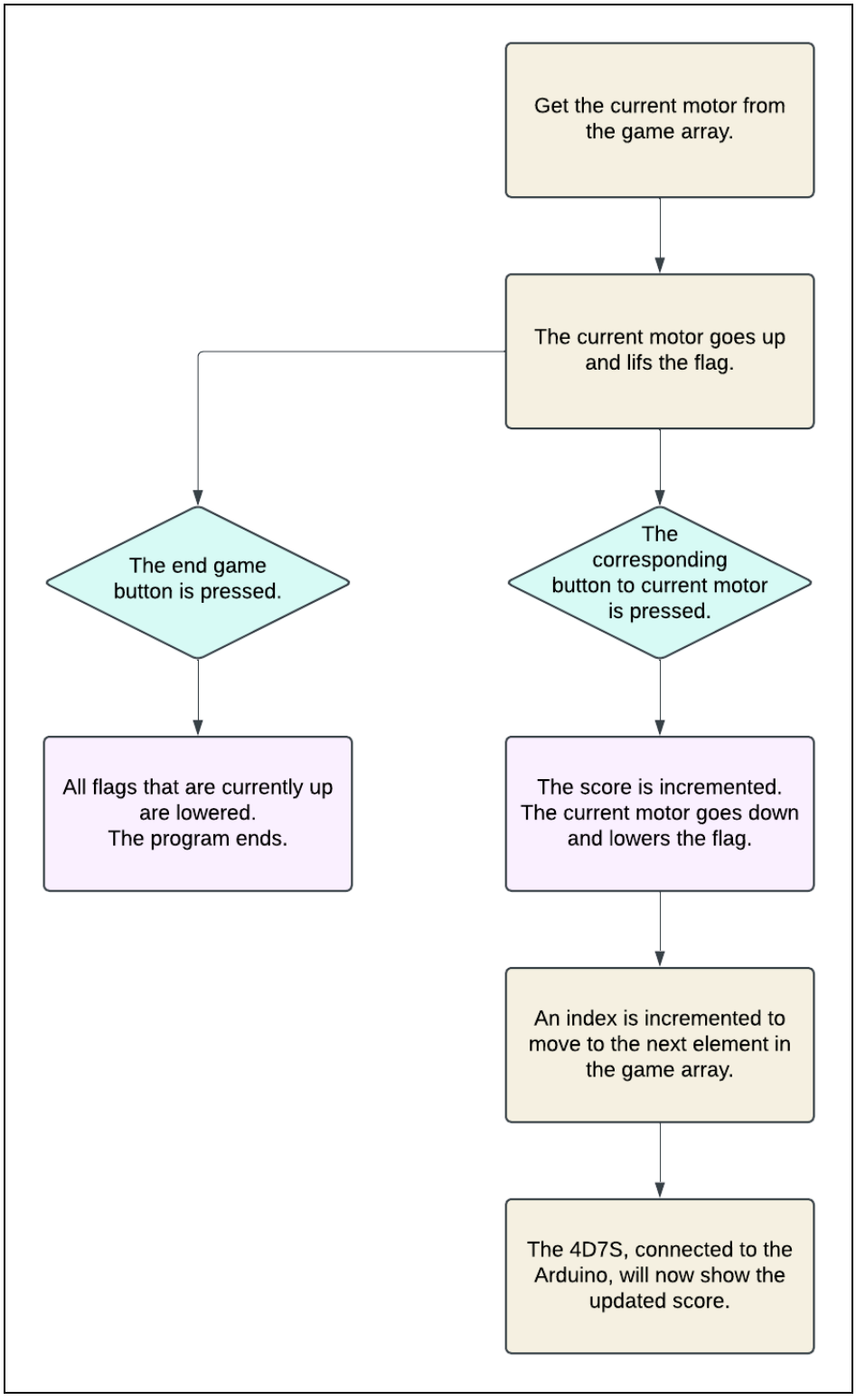
In this particular version, we utilize four ELEGOO ULN2003 stepper motors, each equipped with a blue flag to act as a target. There is also a button for each of the stepper motors, which acts as the player input.

- The game initializes, a stepper motor rotates and lifts up the flag.
- The player will then press the corresponding button for that stepper motor.
- This represents the “hitting” of the target. The motor will rotate the other way to lower the flag.
- The score increments by one; then, a different stepper motor will rotate and lift up its flag.
- This game sequence continues until the “end” button is pressed, upon which all flags that are currently up will be lowered. The game ends.
- The score is constantly displayed on a 4-Digit 7-Segment Display (4D7S).

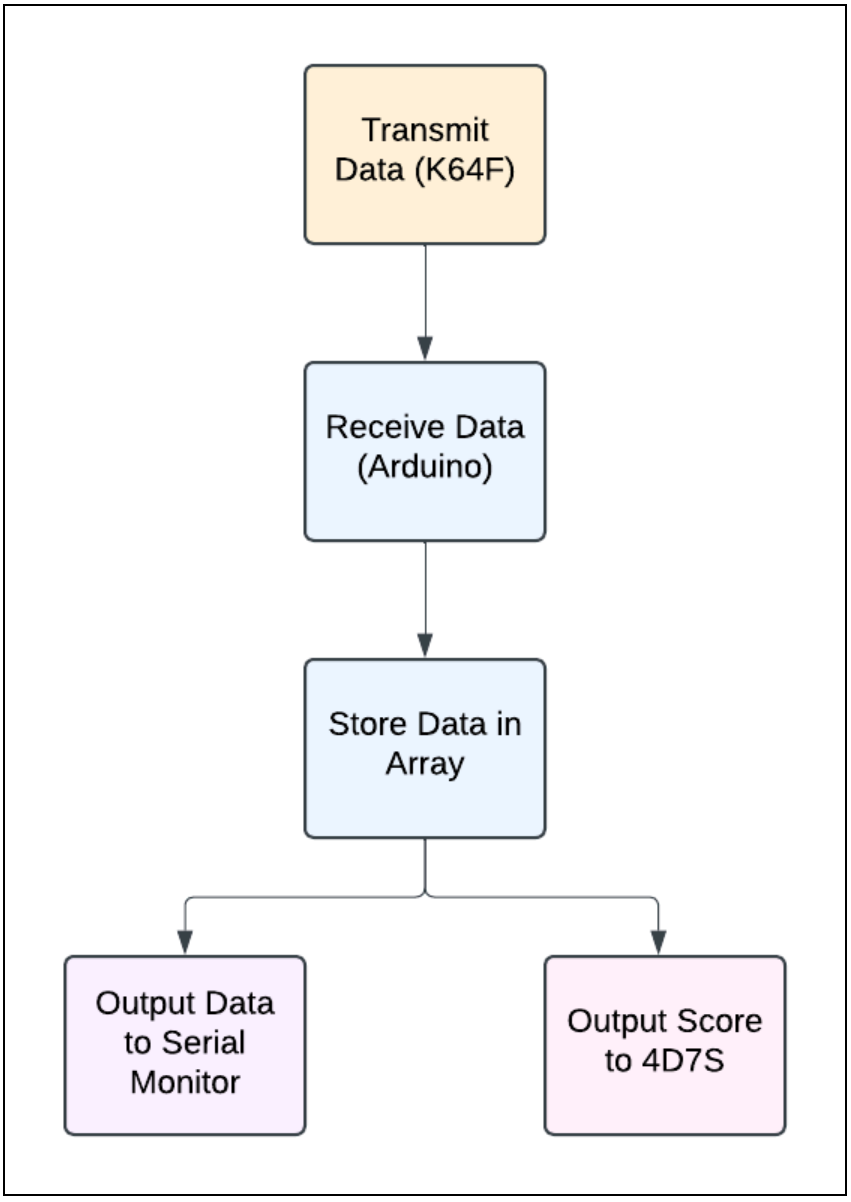
We identify the following requirements to achieve the goal of this Final Project:

1. General Purpose Input/Output (GPIO): The ability to configure Pins on the microcontroller as either Input or Output.
2. Stepper Motors: The ability to individually control the rotation of each Stepper Motor using phases.
3. UART/SPI Communication: The ability to communicate with the computer using UART and send data to another microcontroller using SPI.
4. 4-Digit 7-Segment (4D7S) Display: The ability to output the current score of the game to a 4-Digit 7-Segment (4D7S) Display, using strobing.

II. System Design



Overall Block Diagram



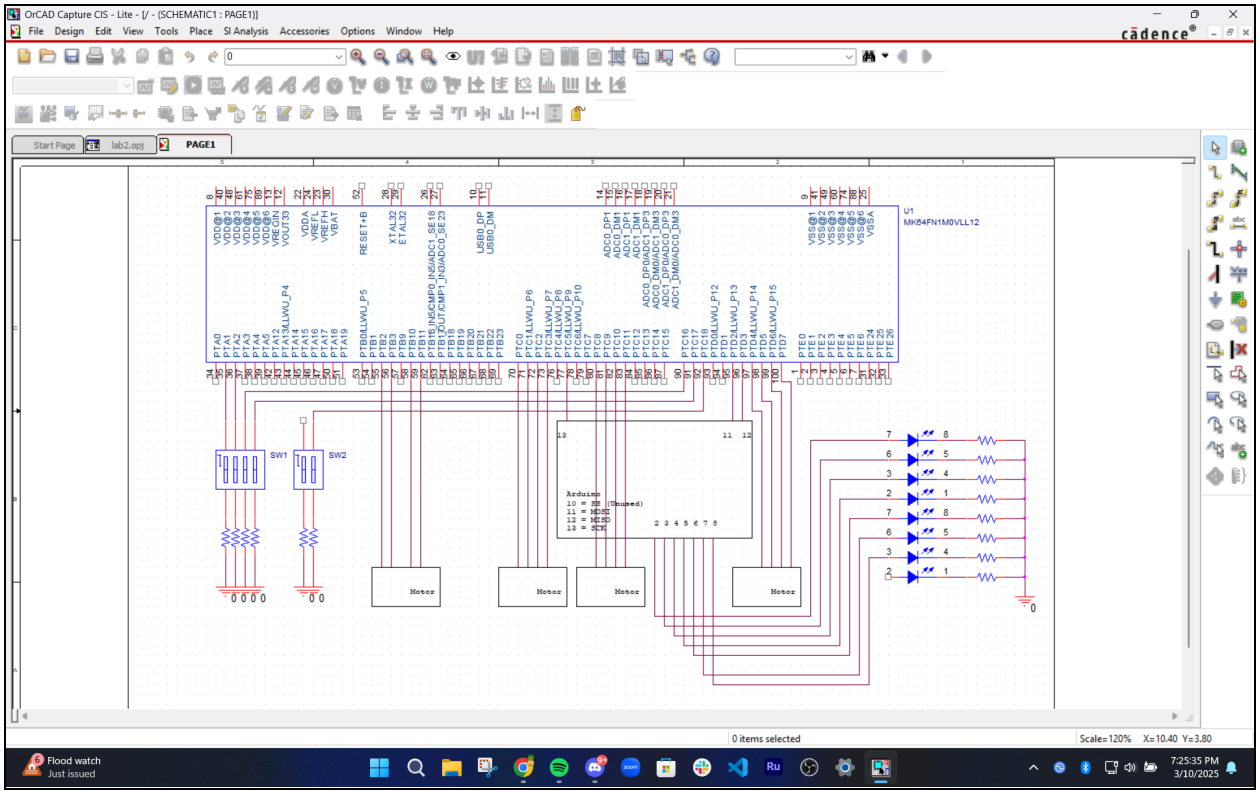
SPI Communication Diagram

III. Implementation Details

Component Name	MCU	Pin(s) Assignment
OCR 6×6×7 Tactile Push Button (5)	K64F	Button 0: PORT C17 Button 1: PORT A1 Button 2: PORT A2 Button 3: PORT C16 Button Reset: PORT C18
ELEGOO ULN2003 5V Stepper Motor (5)	K64F	Motor 0: PORT D4 PORT D5 PORT D6 PORT D7 Motor 1: PORT C0 PORT C1 PORT C2 PORT C3 Motor 2: PORT C8 PORT C9 PORT C10 PORT C11 Motor 3: PORT B2 PORT B3 PORT B10 PORT B11

4-Digit 7-Segment Display (4D7S) (1)	Arduino	Segment A: D2 Segment B: D3 Segment C: D4 Segment D: D5 Segment E: D6 Segment F: D7 Segment G: D8 Digit 1: A0 Digit 2: A1 Digit 3: A2 Digit 4: A3

SPI Line	Pin on Arduino	Pin of K64F
MOSI	D11	PORT D2
MISO	D12	PORT D3
SCK	D13	PORT C5
SS	D10 (Unused)	Unused



Cadence Capture CIS Schematic (Hardware Design)

Source Code - K64F	main.c

```
/* ###################################################################
**     Filename    : main.c
**     Project     : EE128_Project
**     Processor   : MK64FN1M0VLL12
**     Version     : Driver 01.01
**     Compiler    : GNU C Compiler
**     Date/Time   : 2025-03-08, 22:52, # CodeGen: 0
**     Abstract    :
**         Main module.
**         This module contains user's application code.
**     Settings    :
**     Contents    :
**         No public methods
**
** ###################################################################*/
/*!
** @file main.c
** @version 01.01
** @brief
**         Main module.
**         This module contains user's application code.
*/
/*!
**  @addtogroup main_module main module documentation
**  @{
*/
/* MODULE main */


/* Including needed modules to compile this module/procedure */
#include "Cpu.h"
#include "Events.h"
#include "Pins1.h"
#include "CsIO1.h"
#include "IO1.h"
#include "SM1.h"
/* Including shared modules, which are used for whole project */
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"
#include "PDD_Includes.h"
#include "Init_Config.h"
/* User includes (#include below this line is not maintained by Processor Expert) */
#include "MK64F12.h"
/*lint -save  -e970 Disable MISRA rule (6.3) checking. */

int current_motor = 0;
int array_index = 0;
```

```
int array_size = 100;
int game[100] = {
    0, 3, 2, 1, 0, 2, 1, 3, 1, 0, 3, 2, 1, 3, 0, 1, 3, 1, 2, 0,
    1, 2, 3, 0, 2, 0, 1, 3, 2, 1, 0, 2, 0, 3, 2, 1, 2, 1, 3, 0,
    3, 0, 1, 2, 1, 3, 2, 0, 1, 2, 3, 1, 2, 0, 1, 3, 0, 2, 0, 1,
    3, 0, 2, 3, 0, 1, 3, 2, 0, 3, 1, 0, 3, 2, 0, 1, 3, 0, 3, 2,
    0, 2, 1, 3, 2, 3, 0, 1, 3, 1, 2, 0, 1, 2, 0, 3, 0, 3, 1, 2
};
int score = 0;

void software_delay(unsigned long delay)
{
    while (delay > 0) delay--;
}

void motor_up(int index) {
       int phases[8] = {0b0001, 0b0011, 0b0010, 0b0110, 0b0100, 0b1100, 0b1000, 0b1001};
       int i = 0;
       for (int count = 0; count < 1024; count++) {
              software_delay(0xBB8);
              if (index == 0) {
                     GPIOD_PDOR = ((GPIOD_PDOR & 0x0F) | (phases[i] << 4));
              }
              else if (index == 1) {
                     GPIOC_PDOR = ((GPIOC_PDOR & 0xFF0) | (phases[i]));
              }
              else if (index == 2) {
                     GPIOC_PDOR = ((GPIOC_PDOR & 0x0FF) | (phases[i] << 8));
              }
              else if (index == 3) {
                     GPIOB_PDOR = ((GPIOB_PDOR & 0x3F3) | ((phases[i] & 0xC) << 8) |
((phases[i] & 0x3) << 2));
              }
              i++;
              if (i > 7) {
                     i = 0;
              }
       }
}

void motor_down(int index) {
       int phases[8] = {0b0001, 0b0011, 0b0010, 0b0110, 0b0100, 0b1100, 0b1000, 0b1001};
       int i = 7;
       for (int count = 0; count < 1024; count++) {
              software_delay(0xBB8);
              if (index == 0) {
                     GPIOD_PDOR = ((GPIOD_PDOR & 0x0F) | (phases[i] << 4));
              }
              else if (index == 1) {
                     GPIOC_PDOR = ((GPIOC_PDOR & 0xFF0) | (phases[i]));
              }
              else if (index == 2) {
```

```
                        GPIOC_PDOR = ((GPIOC_PDOR & 0x0FF) | (phases[i] << 8));
                }
                else if (index == 3) {
                        GPIOB_PDOR = ((GPIOB_PDOR & 0x3F3) | ((phases[i] & 0xC) << 8) |
((phases[i] & 0x3) << 2));
                }
                i--;
                if (i < 0) {
                        i = 7;
                }
        }
}

unsigned char write[512];

int main(void)
/*lint -restore Enable MISRA rule (6.3) checking. */
{
  /* Write your local variable definition here */

  /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.                    ***/

  /* Write your code here */
  uint32_t delay;
  int len;
  LDD_TDeviceData *SM1_DeviceData;
  SM1_DeviceData = SM1_Init(NULL);

  SIM_SCGC5 |= SIM_SCGC5_PORTA_MASK;
  SIM_SCGC5 |= SIM_SCGC5_PORTB_MASK;
  SIM_SCGC5 |= SIM_SCGC5_PORTC_MASK;
  SIM_SCGC5 |= SIM_SCGC5_PORTD_MASK;

  PORTA_PCR1 = 0x100; // Configure Pin 1 for GPIO
  PORTA_PCR2 = 0x100; // Configure Pin 2 for GPIO

  PORTB_PCR2 = 0x100; // Configure Pin 2 for GPIO
  PORTB_PCR3 = 0x100; // Configure Pin 3 for GPIO
  PORTB_PCR10 = 0x100; // Configure Pin 10 for GPIO
  PORTB_PCR11 = 0x100; // Configure Pin 11 for GPIO

  PORTC_PCR0 = 0x100; // Configure Pin 0 for GPIO
  PORTC_PCR1 = 0x100; // Configure Pin 1 for GPIO
  PORTC_PCR2 = 0x100; // Configure Pin 2 for GPIO
  PORTC_PCR3 = 0x100; // Configure Pin 3 for GPIO
  PORTC_PCR4 = 0x100; // Configure Pin 4 for GPIO
  PORTC_PCR8 = 0x100; // Configure Pin 8 for GPIO
  PORTC_PCR9 = 0x100; // Configure Pin 9 for GPIO
  PORTC_PCR10 = 0x100; // Configure Pin 10 for GPIO
  PORTC_PCR11 = 0x100; // Configure Pin 11 for GPIO
```

```
PORTC_PCR16 = 0x100; // Configure Pin 16 for GPIO
PORTC_PCR17 = 0x100; // Configure Pin 17 for GPIO
PORTC_PCR18 = 0x100; // Configure Pin 18 for GPIO

PORTD_PCR4 = 0x100; // Configure Pin 4 for GPIO
PORTD_PCR5 = 0x100; // Configure Pin 5 for GPIO
PORTD_PCR6 = 0x100; // Configure Pin 6 for GPIO
PORTD_PCR7 = 0x100; // Configure Pin 7 for GPIO

// Set Port A Pins 1-2 as Input
GPIOA_PDDR &= ~(1 << 1);
GPIOA_PDDR &= ~(1 << 2);

// Set Port B Pins 2-3, 10-11 as Output
GPIOB_PDDR |= 0xC0C;

// Set Port C Pins 0-3, 8-11 as Output
GPIOC_PDDR |= 0xF1F;

// Set Port C Pins 16-18 as Input
GPIOC_PDDR &= ~(1 << 16);
GPIOC_PDDR &= ~(1 << 17);
GPIOC_PDDR &= ~(1 << 18);

// Set Port D Pins 4-7 as Output
GPIOD_PDDR |= 0xF0;

// Start the game and put the first motor up.
motor_up(game[array_index]);

// Set current motor to first element in the array.
current_motor = game[array_index];

for (;;) {
    // UART and SPI
    printf("Score \t: %4d\n", score);
    len = sprintf(write, "Score \t: %4d\n", score);
    GPIOC_PDOR |= 0x10;
    SM1_SendBlock(SM1_DeviceData, &write, len);
    for(delay = 0; delay < 300000; delay++); //delay
    GPIOC_PDOR &= 0xEF;

    // Read buttons
    int button_0 = (GPIOC_PDIR & (1 << 17));
    int button_1 = (GPIOA_PDIR & (1 << 1));
    int button_2 = (GPIOA_PDIR & (1 << 2));
    int button_3 = (GPIOC_PDIR & (1 << 16));
    int button_end = (GPIOC_PDIR & (1 << 18));

    if ((current_motor == 0) && (button_0 != 0)) {
       score++;
       motor_down(0);
```

```
      array_index++;
      if (array_index >= array_size) {
         array_index = 0;
      }
      if (game[array_index] == 1) {
         motor_up(1);
         current_motor = 1;
      }
      else if (game[array_index] == 2) {
         motor_up(2);
         current_motor = 2;
      }
      else if (game[array_index] == 3) {
         motor_up(3);
         current_motor = 3;
      }
   }

   else if ((current_motor == 1) && (button_1 != 0)) {
      score++;
      motor_down(1);
      array_index++;
      if (array_index >= array_size) {
         array_index = 0;
      }
      if (game[array_index] == 0) {
         motor_up(0);
         current_motor = 0;
      }
      else if (game[array_index] == 2) {
         motor_up(2);
         current_motor = 2;
      }
      else if (game[array_index] == 3) {
         motor_up(3);
         current_motor = 3;
      }
   }

   else if ((current_motor == 2) && (button_2 != 0)) {
      score++;
      motor_down(2);
      array_index++;
      if (array_index >= array_size) {
         array_index = 0;
      }
      if (game[array_index] == 0) {
         motor_up(0);
         current_motor = 0;
      }
      else if (game[array_index] == 1) {
         motor_up(1);
```

```
            current_motor = 1;
         }
         else if (game[array_index] == 3) {
            motor_up(3);
            current_motor = 3;
         }
      }

      else if ((current_motor == 3) && (button_3 != 0)) {
         score++;
         motor_down(3);
         array_index++;
         if (array_index >= array_size) {
            array_index = 0;
         }
         if (game[array_index] == 0) {
            motor_up(0);
            current_motor = 0;
         }
         else if (game[array_index] == 1) {
            motor_up(1);
            current_motor = 1;
         }
         else if (game[array_index] == 2) {
            motor_up(2);
            current_motor = 2;
         }
      }

      else if (button_end != 0) {
         motor_down(current_motor);
         score = 0;
         array_index = 0;
         break;
      }

    }
  /* For example: for(;;) { } */

  /*** Don't write any code pass this line, or it will be deleted during code generation.
***/
  /*** RTOS startup code. Macro PEX_RTOS_START is defined by the RTOS component. DON'T
MODIFY THIS CODE!!! ***/
  #ifdef PEX_RTOS_START
    PEX_RTOS_START();                  /* Startup of the selected RTOS. Macro is defined by
the RTOS component. */
  #endif
  /*** End of RTOS startup code.  ***/
  /*** Processor Expert end of main routine. DON'T MODIFY THIS CODE!!! ***/
  for(;;){}
  /*** Processor Expert end of main routine. DON'T WRITE CODE BELOW!!! ***/
} /*** End of main routine. DO NOT MODIFY THIS TEXT!!! ***/
```

```
/* END main */
/*!
** @}
*/
/*
** ###################################################################
**
**     This file was created by Processor Expert 10.5 [05.21]
**     for the Freescale Kinetis series of microcontrollers.
**
** ###################################################################
*/
```

Variables:

`int current_motor`	Integer variable to keep track of the current motor in the game.
`int array_index`	Integer variable to keep track of the current index in the game array.
`int array_size`	Integer variable that holds the size of the game array (prevent out-of-bounds access).
`int game[]`	Integer array that contains the sequence of the motors. This array was randomly generated prior to uploading the code to the K64F.
`int score`	Integer variable to keep track of the current score.

Helper Functions:

`void motor_up()`	Rotates the motor 90 degrees to lift up the flag. The function passes in a variable called `index` which refers to the corresponding motor. To make a 90 degree rotation, we increment through the `phases` array a total of 1,024 times (a full 360 degree rotation is 4,096 steps). At every step in the array, we cast the 4 bits to the output pins of the corresponding motor. We use bit masking and bit shifting to make sure that other bits are untouched during the operation.
`void motor_down()`	Rotates the motor 90 degrees to lower the flag. Uses the same logic as `motor_up()`, with the only difference being that we decrement through the `phases` array (7 to 0) instead of incrementing.

Initializations:
We first configure Clock Gating for PORT A, PORT B, PORT C, and PORT D.

We then configure Pins for GPIO, and then configure each individual Pin as either Input or Output.

We start the game by rotating the first motor and lifting up its flag.

We then enter the for loop. At every iteration of the for loop, we print the score to the computer terminal (UART) and send the score to the Arduino (SPI).

The score is sent to the Arduino as a message in the following format:
"Score \t: %4d\n", score

```
Score   :    0
```

Main Loop:

The main sequence is as follows:

1. The current motor goes up and lifts the flag.
2. IF: The corresponding button to the current motor is pressed:
    a. The score is incremented.
    b. The current motor goes down and lowers the flag.
    c. An index is incremented to move to the next element in the game array.
    d. The 4D7S, connected to the Arduino, will now show the updated score.
3. IF: The “end game” button is pressed.
    a. All flags that are currently lifted up are lowered.
    b. The program ends.

Source Code - Arduino	sketch_mar9a.ino

```
#include <SPI.h>
char buff [255];
volatile byte indx;
volatile boolean process;
int length;
int nums[10] = {0b1111110, 0b0110000, 0b1101101, 0b1111001, 0b0110011, 0b1011011, 0b1011111,
0b1110000, 0b1111111, 0b1111011};
char a = ' ';
char b = ' ';

void outNum(int num) {
  digitalWrite(2, bitRead(nums[num], 6)); // A
  digitalWrite(3, bitRead(nums[num], 5)); // B
  digitalWrite(4, bitRead(nums[num], 4)); // C
  digitalWrite(5, bitRead(nums[num], 3)); // D
  digitalWrite(6, bitRead(nums[num], 2)); // E
  digitalWrite(7, bitRead(nums[num], 1)); // F
  digitalWrite(8, bitRead(nums[num], 0)); // G
}

void writeData(char a, char b) {
  if (a == ' ') {
    digitalWrite(A0, HIGH);
    digitalWrite(A1, HIGH);
    digitalWrite(A2, LOW);
    digitalWrite(A3, HIGH);
    outNum(0);
  }
  else {
    digitalWrite(A0, HIGH);
    digitalWrite(A1, HIGH);
    digitalWrite(A2, LOW);
    digitalWrite(A3, HIGH);
    outNum(a - '0');
  }
  delay(5);
  if (b == ' ') {
    digitalWrite(A0, HIGH);
    digitalWrite(A1, HIGH);
    digitalWrite(A2, HIGH);
    digitalWrite(A3, LOW);
    outNum(0);
  }
  else {
    digitalWrite(A0, HIGH);
    digitalWrite(A1, HIGH);
    digitalWrite(A2, HIGH);
    digitalWrite(A3, LOW);
    outNum(b - '0');
```

```
  }
  delay(5);
}

void setup (void) {
  Serial.begin (115200);
  pinMode(MISO, OUTPUT); // have to send on master in so it set as output
  pinMode(2, OUTPUT);
  pinMode(3, OUTPUT);
  pinMode(4, OUTPUT);
  pinMode(5, OUTPUT);
  pinMode(6, OUTPUT);
  pinMode(7, OUTPUT);
  pinMode(8, OUTPUT);
  pinMode(A0, OUTPUT);
  pinMode(A1, OUTPUT);
  pinMode(A2, OUTPUT);
  pinMode(A3, OUTPUT);
  pinMode(A5, OUTPUT); // for testing
  SPCR |= _BV(SPE); // turn on SPI in slave mode
  indx = 0; // buffer empty
  process = false;
  SPI.attachInterrupt(); // turn on interrupt
}

ISR (SPI_STC_vect) // SPI interrupt routine
{
  byte c = SPDR; // read byte from SPI Data Register

  if (indx < sizeof(buff)) {
    buff[indx++] = c; // save data in the next index in the array buff
    if (c == '\n') {
      digitalWrite(A5, HIGH);
      buff[indx - 1] = 0; // replace newline ('\n') with end of string (0)
      delay(1000);
      digitalWrite(A5, LOW);
      process = true;
    }
  }
}

void loop (void) {
  if (process) {
    process = false; //reset the process
    length = strlen(buff);
    Serial.println (buff); //print the array on serial monitor
    Serial.print(buff[length - 2]);
    Serial.println(buff[length - 1]);
    a = buff[length - 2];
    b = buff[length - 1];
    indx= 0; //reset button to zero
  }
```

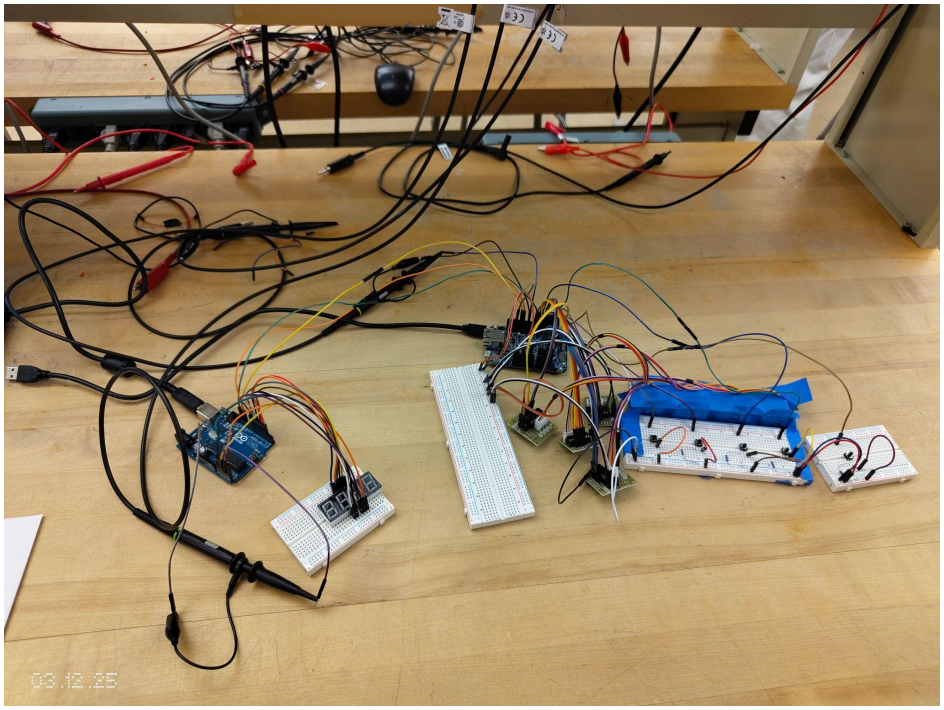
```
  writeData(a, b);
}
```

The Arduino code is simple and straightforward.

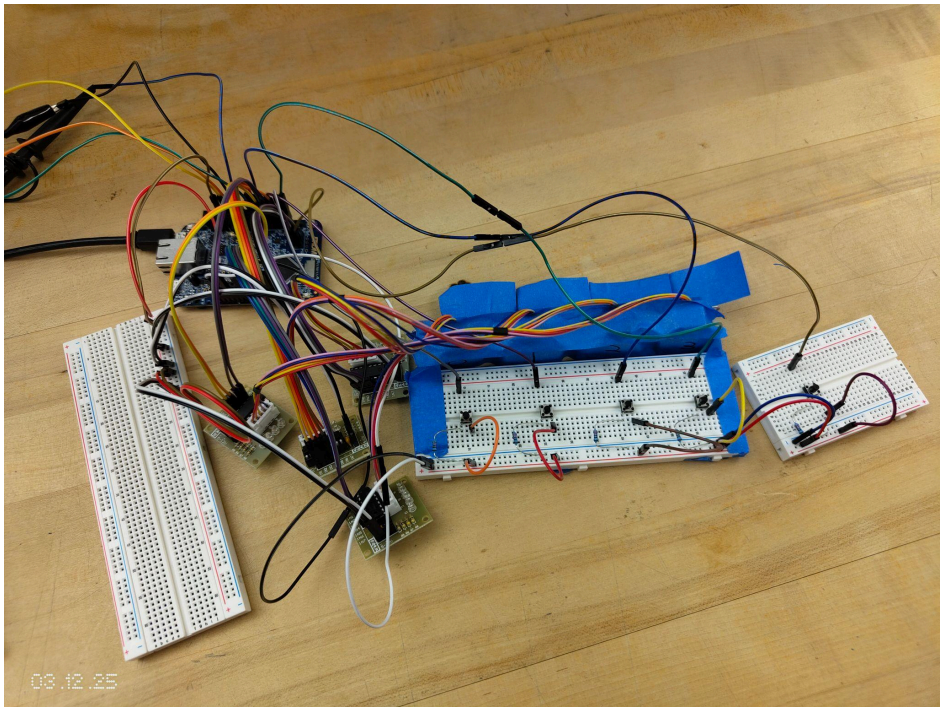
We have an SPI interrupt routine; when a message is received, it stores the entire message in a buffer array and then stores a null terminator at the end (upon encountering a newline character). Then, it sets the `process` flag to true.

In the main loop, we do the following:

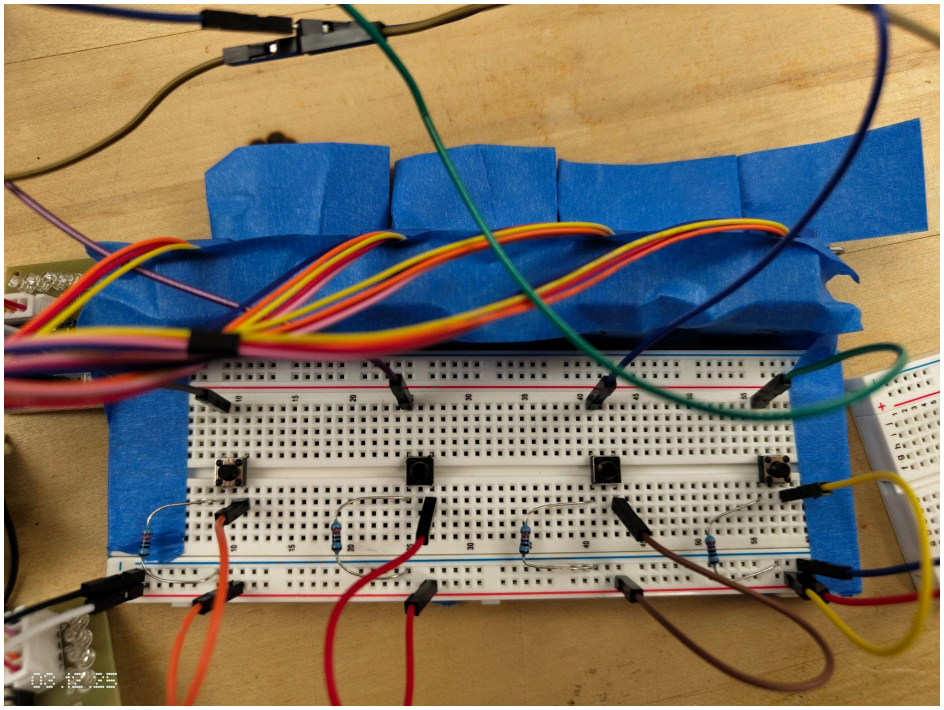
- If the `process` flag is true:
    - Reset the `process` flag.
    - Print the received string to the Serial Monitor (this was mainly for debugging purposes and making sure the SPI worked correctly).
    - Sets character variables `a` and `b` to the third last and second last characters in the array, which is where the score is stored.
- Output character variables `a` and `b` to the 4D7S Display, using strobing. This is handled by the custom function `writeData()`.



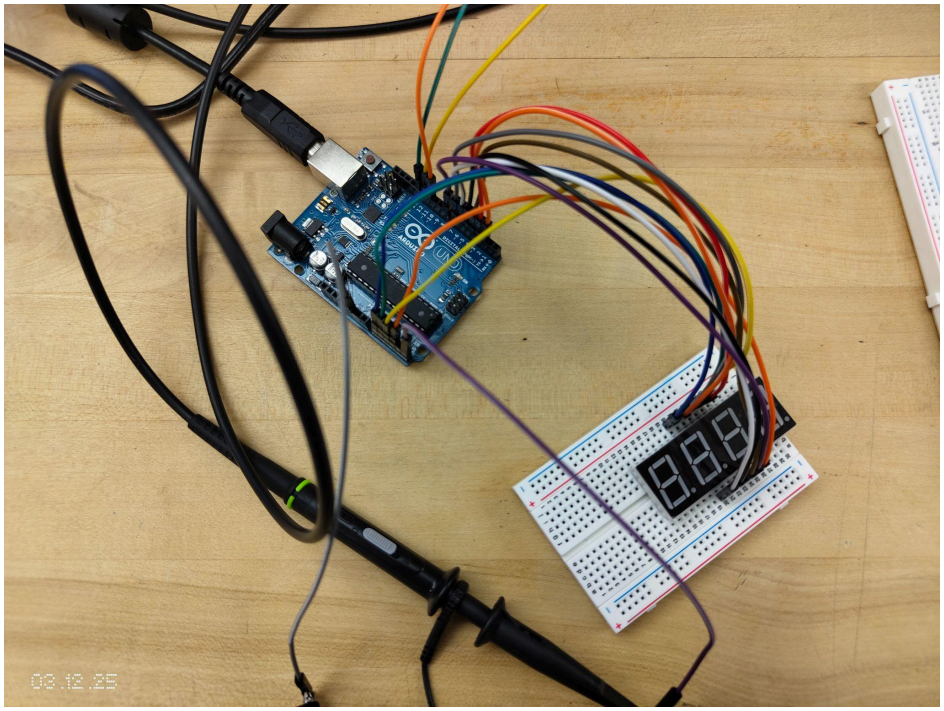
Circuit Board Picture (1) - Complete System



Circuit Board Picture (2) - K64F Connections



Circuit Board Picture (3) - Motors and Buttons



Circuit Board Picture (4) - Arduino and 4D7S

IV. Evaluation

In Section I (Project Description), we identified four main requirements of the project. These requirements are 1) GPIO, 2) UART/SPI Communication, 3) Stepper Motor Control, and 4) 4D7S Display.

We successfully implemented those requirements into the project, as outlined in Section 3 (Implementation Details).

To verify that our system works properly and consistently, we have to make sure that it can account for various edge cases.

Edge Case #1: Multiple buttons are pressed at once.
We tested this and verified that the game continues as normal; no additional flags are erroneously lifted up or lowered, and the score does not increment more than it is supposed to. This is the expected behavior.

Edge Case #2: Buttons are pressed after the player has ended the game.
We tested this and verified that the game does not reactivate if the player presses buttons after ending the game. This is the expected behavior.

Edge Case #3: The player has gone through the entire game array.
We accounted for this edge case in the code. The `array_index` variable resets to 0 once it is greater than or equal to the `array_size` variable.

Based on our testing, we can conclude that our system can maintain full functionality even in rare edge cases.

Additionally, SPI Communication is a critical component of our system. We need to ensure that the communication speed is sufficient to transmit the necessary data between the K64F and the Arduino.

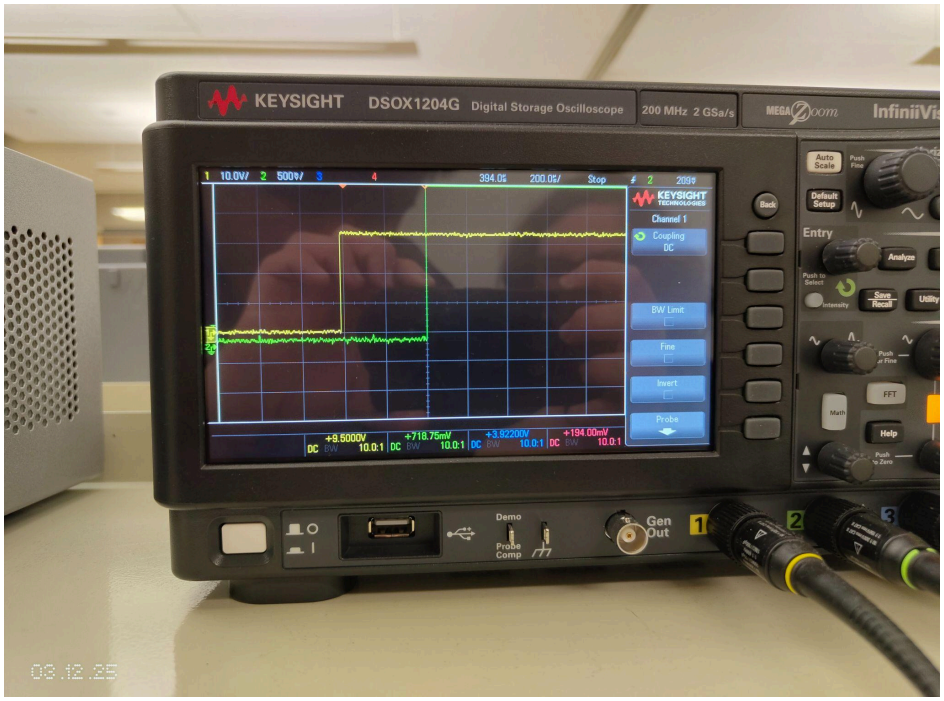
We can use the method of measuring the latency of a string from Lab 6.

"This can be achieved by setting a single GPIO pin on the K64F high to signify the beginning of transmission. The end of transmission can be marked by setting an Arduino pin HIGH when the Arduino detects the newline character ('\n') in the SPI ISR."

```
for (;;) {
      // UART and SPI
      printf("Score \t: %4d\n", score);
      len = sprintf(write, "Score \t: %4d\n", score);
      GPIOC_PDOR |= 0x10;
      SM1_SendBlock(SM1_DeviceData, &write, len);
      for(delay = 0; delay < 300000; delay++); //delay
      GPIOC_PDOR &= 0xEF;
      ...
  }
```

```
ISR (SPI_STC_vect) // SPI interrupt routine
{
  byte c = SPDR; // read byte from SPI Data Register

  if (indx < sizeof(buff)) {
    buff[indx++] = c; // save data in the next index in the array buff
    if (c == '\n') {
      digitalWrite(A5, HIGH);
      buff[indx - 1] = 0; // replace newline ('\n') with end of string (0)
      delay(1000);
      digitalWrite(A5, LOW);
      process = true;
    }
  }
}
```



The end-to-end latency of transmitting one string is 416µs (or 0.416ms).

We should also verify that this result is reasonable (and that it is not due to an error with our hardware setup).

We are transmitting the following message:

```
Score   :    0
```

This is a total of 14 characters, which take up 8 bits each. Therefore, we are transmitting a total of 14 * 8 = 112 bits. Because SPI transmits one bit per clock cycle (tick), we need a total of 112 ticks to transmit the full message.

The SPI clock rate is configured to be 374.5 KHz. We can extrapolate that each tick takes 1 / 374.5 KHz = 2.67µs

(2.67µs / tick) * (112 ticks) = 299.04µs = 0.299ms

The end-to-end latency we measured was 416µs. As mentioned in the lecture, SPI has some overhead, which explains the difference between the two values.

Therefore, the reading from the oscilloscope is reasonable.
We can conclude that SPI does operate at a sufficient speed to transmit the data between the two microcontrollers.

We created a video to demonstrate our system's complete functionality.

Michael Tin, James Krejci - EE128 Final Project Demo

V. Discussions

Problem(s) Encountered	Solution
We attempted to configure PORT A0 as an input for the “end game” button. However, we were getting no reading from this particular pin and the code was not working properly as a result.	We fixed the issue by switching the “end game” button to PORT C18.
We attempted to configure PORT A0 as an output for measurement of the end-to-end latency of transmitting a string over SPI. However, we were unable to output any HIGH or LOW signals from this particular pin.	We fixed the issue by switching to using PORT C4.
After initially configuring the K64F SPI, we noticed that it was not transmitting any strings to the Arduino.	We realized that we were mistakenly setting the K64F SPI Pins as GPIO Pins during initialization. We removed this code and the issue was fixed.

Current Implementation	Possible Improvement(s)
We randomly generate a sequence of 100 numbers for an array. We include this array in the program that we upload onto the K64F. This array is the game sequence (the order in which the flags will lift up).	We can create a function to pick a random flag to lift up. This way, we will not have to generate a new random sequence every time we run the program.
The game ends when the player presses the “end game” button.	We can have this button simply reset the score to zero and restart the game, rather than merely ending the game.
The game runs in an infinite loop until the “end game” button is pressed.	We can add a timer and see how high of a score the player can get within a certain time. We can add another 7-Segment Display to show the timer.

VI. Roles and Responsibilities of Group Members

Michael Tin

- Completed the code and software implementation for the system
- Section 3 (Implementation Details)
- Section 4 (Evaluation)

James Krejci

- Completed the wiring and hardware implementation for the system.
- Section 1 (Project Description)
- Section 2 (System Design)
- Section 5 (Discussions)

VII. Brief Conclusion

This Final Project is a rendition of “Whack-a-Mole”, which is a popular arcade game where a player scores points by hitting pop-up targets.

In our system design, we utilize four Stepper Motors with blue flags as the targets and four buttons as the player input. Additionally, we utilize SPI Communication to send the current score to an Arduino. There is a 4 Digit 7 Segment (4D7S) Display connected to the Arduino which shows the player’s score.

The game follows a sequence where 1) the flag is lifted up, 2) the player presses the corresponding button, 3) the flag is lowered, and 4) the score is updated. This process repeats until the player presses the “end game” button, upon which the game ends.

We identified four main requirements for our system, which are 1) GPIO, 2) Stepper Motor Control, 3) UART/SPI Communication, and 4) 4D7S Display.

We successfully evaluated our system’s functionality by accounting for various edge cases, such as multiple button presses at once and pressing buttons after the game has ended. We also verified that the SPI Communication between the K64F and the Arduino is sufficient for transmitting data in a timely manner.

This Final Project successfully demonstrates a “Whack-a-Mole” game and meets all the initial requirements outlined in the report.