**Michael Jacques**

**Embedded Systems Final Project – Arcade System**

**Project Enhancements / Independent Design**

**Functionality:**

**Coin Collecting**

Background: The first addition I made to the game came from a bug I was having while creating the base game. While I was still working on the base game, I had an issue where if the player jumped and tried to move left or right at the same time, the character would move, but the jumping character would not be erased from the screen. This meant that there were two characters being drawn to the screen at the same time. The only way to get the frozen jumping character off the screen was to jump back into the frozen character’s box. Then the game would play normally again. This made me think of the idea of adding coins to be collected while playing the game.

Functionality: I created a new XY object called “coin” that would allow me to print a coin to the screen at different x and y values. I thought that the best place to put the coin code was in the RefreshScreen thread, because the coin would only move when the screen was updating. I then wrote two new functions shown below, drawCoin() and getNewCoinLocation().

Graphical user interface, text, application

Description automatically generated

The first function sets the cursor to the set x and y values and prints an asterisk symbol to represent the coin. The second function erases the current coin and then gets new x and y values for the coin using the rand() function. If the random value is at the current location of the character or the obstacle, a new coin location is created.

To print the coin and detect collisions between the character and the coin, I updated the RefreshScreen state machine. First, in the REFRESHSTART transition, I ran both functions above to get an initial coin drawn to the screen. Then in the DRAW state, drawCoin() is run at every clock tick so that the coin is always there. To detect collision, I added logic in the DRAW transition where it checks if the current HeroLocation.x and HeroLocation.y matches the current coin.x and coin.y. If true, a new coin location is created.

Gameplay:

When the game first starts, a coin is printed at a random location and the obstacle starts moving towards the player

Background pattern

Description automatically generated

When the player collects the coin, a new coin is printed at a different location

Background pattern

Description automatically generated with medium confidence

**Points System**

Background: After coming up with the coin idea, I thought it would be cool to keep track of a players score while they are playing the game. I decided to assign two point values: 1 point for avoiding an obstacle, and 2 points for collecting a coin.

Functionality: I first created an int variable called “score” to keep track of the players score throughout the game. To keep track of the score, I updated the RefreshScreen state machine. The code below shows the logic for updating the score.

Text

Description automatically generated with low confidence

This code is from the DRAW transition of the RefreshScreen state machine. First, if a collision is detected, the game ends before any score can be updated. But, if there is no collision, I first check to see if an obstacle was just avoided by comparing the previous x and y values of the character and obstacle. If an obstacle was avoided, the score is increased by 1. Then, I check to see if a coin was just collected, and if true the score is increased by 2. When the game ends, this score is printed to the screen so that players can see how well they did.

**Name / High Score Tracker**

Background: Once I had the points system in place, I wanted a way to keep track of the high score. Then I thought it would be cool to have the player input their initials so that the game could keep track of the score that goes with the initials. Then, the high score could be printed at the end of each round along with the initials of the person that owns the high score.

Functionality: First, I created variables to store the current players initials, the high score, and the initials of the owner of the high score. I also created a Boolean variable that is true when a new high score is set and false when it isn’t a high score. I also created an array of characters with every letter in it which the player uses to select their initials at the beginning of the game.

Calendar

Description automatically generated with medium confidence

Then, in the REFRESHSTART state of the RefreshScreen state machine, I included a function getInitials() (shown below), that lets the user select their initials using the joystick. A letter from the array is printed to the LCD and then the user can move through the letters by moving the joystick left and right. When the button is pressed, the letter is selected.

Text

Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated

These initials are stored in the name array and are used at the end of the game. When the player loses the game, the checkHighscore() function (shown below) is run. This function checks if the current score is higher than the high score, and if true it sets the new high score and initials of the new high score owner. It also sets the high score Boolean value to true or false.

Graphical user interface, text, application, Word

Description automatically generated

Then, in the displayGameOver() function, the high score Boolean variable is used to determine what is displayed to the screen (either a new high score was set, or the current players score is shown and then the high score is shown after.

Gameplay:

When the game first starts, the user is prompted to enter their first and second initials. They can do this by using the joystick to move left and right between the letters and then pushing the button to select the letter.

Graphical user interface, application

Description automatically generated with medium confidence

A picture containing calendar

Description automatically generated

After each of the above statements are printed to the LCD the user sees a letter on the screen and they can either select it or use the joystick to move to the next letter.

Background pattern

Description automatically generated

When the player makes contact with an obstacle in the game, “GAME OVER!” is printed to the screen and the game over sound effect is played.

Graphical user interface

Description automatically generated

Then, there are two options for what is printed next. If the current players score is higher than the current highscore, “NEW HIGHSCORE!” is printed to the screen and then the players initials are printed along with their score.

A picture containing graphical user interface

Description automatically generated

A picture containing background pattern

Description automatically generated

If a new high score is not set, the current players score is printed, and then the highscore holders initials and score are printed.

A picture containing graphical user interface

Description automatically generated

A picture containing calendar

Description automatically generated

**Music / Sound Effects**

Background: The last feature I wanted to add to my game was background music and sound effects. I used the “Coconut Mall” music from Mario Kart as my background music, I used the Mario coin and win level music as sound effects, and I created my own jump sound.

Functionality: First, I connected two buzzers to my breadboard, one for sound effects, and one for background music. I created two sound effects, one for collecting a coin, and one for a player jumping. The functions jumpSound() and coinSound() shown below define the tones that the buzzer plays. These functions are in the PlayerActions and RefreshScreen tabs, respectively.

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Then, I created a new thread called Music, that is used to play the background music and the win music if a new high score is set. The state machine for the Music tab is shown below.

Diagram

Description automatically generated

To play the music, I created arrays to store the tones and respective delays for the background music and the win music. This way when the backgroundMusic() or winMusic() functions are run, one note is played and then the index is increased so that the next note is played at the next function call. These arrays are shown below.

Scatter chart

Description automatically generated

I decided to play the music this way so that I can immediately stop the music when the game ends. I used two functions backgroundMusic() and winMusic() (shown below) that are used to play the music.

Graphical user interface, text, application, email

Description automatically generated

**Course Concepts:**

**Additional Threads**

I created an additional thread for my game called Music that was used to play the background music and the new high score music. By controlling the buzzer from a separate thread, I was able to play music on the buzzer while the game was still running. This meant that the music could play whether or not I was printing things to the LCD screen. I believe that using an additional thread helped me to make the background music run as smooth as possible while the game was being played.

**Significant Programming Complexity**

Throughout the projects, I made use of significant programming complexity to accomplish the desired functionality of the game. I added programming complexity when creating the coin system, creating the scoring system, creating the initials/highscore system, and creating the background and win music functions.

When adding the coin system, I created a new XY object for the coin and then created two new functions, drawCoin() and getNewCoinLocation() which I described above. These functions involved using random numbers as well as additional collision logic between the character and the coin.

When creating the scoring system, I created logic to determine if an obstacle had been avoided (shown above in the Points System section). This was difficult because there were times when the obstacle and the character would be moving towards each other and they would actually swap x-values and not actually collide with each other. In order to fix this problem, I changed the logic so that I was comparing the x-value of the character with the current and previous x-value of the obstacle. This way I could see if the character and the obstacle had gone through each other and end the game. Then, I was able to compare the previous x and y locations of the character and the obstacle to see if the character had successfully avoided an obstacle.

The initials/highscore system was the most complex part of my code. I used an array of characters with each letter in it that the user could select their initial from. There is logic within the getInitials() function where the user can move left and right along the letters by using the joystick. I also included logic so that the user cannot scroll behind “A” or past “Z” in the list. Then I used the jumpFlag variable to see if the user had picked a letter. By pushing the button, the jumpFlag is set to 1 and the letter is saved. This is repeated for the second initial and then both initials are stored in a 2-character array that is used at the end of the game to display the initials.

For the background and win music, I originally wrote out each individual tone and delays as their own lines of code and then polled the state of the RefreshScreen state machine to see if the music should continue playing. This worked to play the music, but the code was very confusing and long. In order to make the code simpler and more compact, I created arrays that store both the tones and delays of the background and win music. These arrays are looped through one note at a time every time the backgroundMusic() or winMusic() functions are called. I used these functions within a Music state machine that allowed me to determine exactly when the music should play and not play.

**Interfacing with Additional Hardware**

I interfaced with two buzzers in order to get the desired music and sound effects to play. In order to use the buzzers, I had to learn how to connect them to the MSP432, as well as how to program them to play music. I found a table that showed which frequencies correspond to each note with the buzzer, and from that I was able to transpose sheet music I found online into tones that the buzzer would play. I used the tone() and noTone() functions to play the music. These functions worked well to do what I needed them to, but while reading the documentation of the functions, I found a small issue. Even though I had connected two circuits to my MSP432 board, the tone() function could only play a note to one buzzer at a time (even though they were connected to different pins. This made me rethink how the music should be written into the code, and to solve the problem, I decided to make it so that the jump and coin sound effects would always take priority over the background music. This meant that when the player jumped or a coin was collected, the music would stop, the sound effect would play, and then the music would start playing again right where it left off. Luckily, all of my sound effects are short enough that there is not a noticeable difference in the music while playing the game.

**New State Machine**

In order to play the background music and the new highscore music, I created a new state machine shown below.

Diagram

Description automatically generated

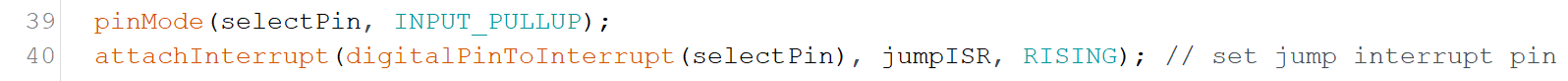
There are three states in this state machine, MUSICWAIT, BACKGROUND, and WIN. The default state is MUSICWAIT. In this state, no music is played and the values of backgroundFlag and winFlag are read from the RefreshScreen tab. Then, when either the backgroundFlag == 1 or the winFlag == 1, a transition occurs to the proper state, and the music plays.

**Base Game Functionality**

**Course Concepts:**

**Interrupt Handling**

Interrupt handling supported the base game functionality in two ways. First, the joystick select pin was used as an interrupt so that as soon as the button on the joystick is pressed, the hero character is able to jump. The pin was set as an input pin and then was turned into an interrupt with the function jumpISR() being called every time the button is pressed. The code for this is shown below.



Then, the jumpISR() function set the jumpFlag to 1 every time the button was pressed and this variable was used in the state machine to determine when the character should jump.

The other form of interrupt that was used in every thread was a timer interrupt. In each thread, a different timer was created and linked to an interrupt service routine that would run at every clock tick. These interrupts were used to cycle through the state machines at specific time intervals and made it much easier to synchronize different actions in the game.

**Concurrent State Machines / Protothreading**

In the base game, there are two state machine running, one in the PlayerActions tab, and the other in the RefreshScreen tab. By creating separate tabs for these state machines, refreshing the screen and getting the characters location can happen independently of each other as well as simultaneously.

The state machine in the PlayerActions tab is used to determine the hero character’s location by getting input from the joystick. The state machine determines if the character should move forward, move backward, or jump, and updates the character’s location accordingly.

The state machine in the RefreshScreen is used to update the screen to reflect the changes in the character’s and obstacle’s positions. This state machine is also used to determine if a collision occurs and if so, the state machine moves into the OVER state, where the displayGameOver() function is run and the players score is displayed. This state machine is shown below

Diagram

Description automatically generated

**Analog-to-Digital Conversion**

In order to make use of the joystick, an analog pin was used to determine which direction the joystick was being pushed. Thresholds were determined through testing the joystick and are shown below.

A picture containing text, orange, close

Description automatically generated

Then, in the playerActionTimerISR(), there is logic included to determine which direction the character should move, and flags are set to reflect this change of location in the state machine. If the analog value of the joystick x-position was less than the left threshold, the backwardFlag was set to 1, and if the x-position was between the left and right thresholds, the forwardFlag was set to 1. This logic is shown below.

Graphical user interface, application, Word

Description automatically generated

**Synchronous State Machines / Timing**

In order for the timing of the game to work properly, the clock frequency of each state machine was set to synchronize the game. First, the RefreshScreen state machine was set to run every 250 ms which is the fastest refresh rate this model of LCD can handle. Then, the PlayerActions state machine was set to 130 ms, which is a little more than half of the RefreshScreen’s. This is because in order for the character to move along the screen at each RefreshScreen tick, the PlayerActions state machine must move through two states to update the location. For example, if the player was to move forward, the PlayerActions state machine would spend one tick at the WAITING state, and then one tick at the FORWARD state. These two states need to happen within one RefreshScreen tick. Through testing, I discovered that when the PlayerAction state machine was set to a 125 ms clock speed, when the joystick was held in a certain direction, the player would move two x-locations over in each clock tick instead of one. To fix this I added an extra 5 ms delay to make it 130 ms total, and this allows the character to move move x-location at a time while also moving at the same speed as the obstacle.