Embedded Systems
Project 6

Date of Submission: April 28, 2019

Joel Yuhas : jxy8307@rit.edu
Athaxes Alexandre : axa2012@rit.edu

Overview

The STM32 Discovery board is a microcontroller capable of running “bare-metal(no operating system)” programs. This project interfaced with FreeRTOS and was initialized using CubeMX to create a “game”. This game involved 2 servo motors where one would be controlled by the computer and the other one by the player. The objective of the game was to have the player match their servo angle to the computer servo angle. The player would move their servo by physically moving the STM32 board, which used the internal of the gyroscopic sensor to measure its angular velocity. The system would use the input data gathered from the gyroscope to change the duty cycle of the PWM connected to the player servo. The computer would move its corresponding servo to a random location. Depending on how quickly the player was able to match the angle of the computer, the player would be awarded points. The final score was communicated to the player via a UART terminal.Overall the final product achieved everything that was required and was ultimately successful.A more advanced version of the game that went beyond the requirements was created as well.

Areas of Focus

Project Assignments	Team Member
Programming of Threads	Joel + Athaxes
Programming of Timing	Joel + Athaxes
Programming of RNG	Joel + Athaxes
Programming of Bonus Features	Joel
Report	Joel + Athaxes

Analysis/Design

A real-time “mirror” game was designed and implemented using embedded systems. This system utilized 2 servo motors, a gyroscopic sensor, and a UART terminal as the main components. These components were set up using CUBE and FreeRTOS. A starter project was provided by the profesor, which laid the foundation for the main structure.

Two main threads were created, a “computer” thread and a “player” thread. Each thread was responsible for handling the PWM for their respective servos. Channels 1 and 2 were used from timer 3 in order to achieve the PWM. Because the same servos from the past projects were being used, the setup was largely identical to the setup in the past. It should be noted that the PWM on channel 2 was inverted for some reasons, where the duty cycle was low when it should of been high. Even after going through CUBE, the cause of this could not be determined. In order to work around it, the duty cycle was inverted on the software by subtracting it from the period.

The player task handled the gyroscope inputs and outputs, and converted those outputs into a corresponding duty cycle. The PWM was set up in such a way that the ARR was 20,000. This put the required duty cycle between 500 and 2000. Values outside of this range were found to negatively affect the servos that were being used. The code below shows how the Y axis of the gyroscope was transformed into a workable PWM signal.

```
                BSP_GYRO_GetXYZ(gyro_velocity_1);   // get raw values from gyro device

                        // Getting gyro values
            for(int ii=0; ii<3; ii++) {
                gyro_angle_1[ii] = (int32_t)(gyro_velocity_1[ii] /
                GYRO_THRESHOLD_DETECTION);
            }

                        // integrating gyro values into servo values
                        player_servo_position += gyro_angle_1[2];
                        if (player_servo_position <= 500){
                                player_servo_position = 500;
                        } else if (player_servo_position >= 2000){
                                player_servo_position = 2000;
                        }


                        // updating servo values
                        set_pwm_pulse(&htim5, TIM_CHANNEL_2,
(20000-player_servo_position));
```

The gyro angle was calculated by dividing the gyro inputs by the GYRO_THRESHOLD, which was 10000. This was done to weed out small input values so that only large movements would be recorded. The gyro angle itself was a number, usually between -50 to 50, which corresponded to changes in the boards angular velocity. A negative value was a counter clockwise movement while a positive one was clockwise. This code was placed in a thread that updated every 10 ms.

The computer thread used a random number generator to move the servo to a random location. This number was between 500-2000, which was the range for the required PWM signal. The servo did not have to move to a discrete location, but could fall anywhere in that range. This thread also calculated it the player servo was within a specific tolerance to count as a “match”, which was +/- 5%.

Lastly, this thread also dealt with calculating the score and outputting it to the UART. The score was calculated by giving the player 10 points every time they “matched” the computer servo. If the player did not match the computer within 5 seconds, then it would be counted as a “miss”. Each consecutive hit increased the “combo” by 1 for a max up to 4 times. The game concluded after 10 attempts and would print a final score. The UART was set up in such a way so that it would show the player, in real time, their score, combo, turn number, and even servo position. The code below shows how these features were implemented within the computer thread.

```
        if (total_counter < 10){

                if (timer_counter <= 50){

                //sprintf(buf, "Play!\r\n");

                //vPrintString(buf);

                // checking if player is withing +/- 5% of the computer position

                if (player_servo_position <= (computer_servo_position +
(computer_servo_position*0.05)) && player_servo_position >= (computer_servo_position
-(computer_servo_position*0.05))){

                        points = points + (10*combo);

                        combo = combo *2;
```

```
                    timer_counter = 0;

                    buf2[total_counter] = '$';


                    // set next position

                    computer_servo_position = (((HAL_RNG_GetRandomNumber(&Rng_Handle)) %
        1501) + 500.0);

                    set_pwm_pulse(&htim5, TIM_CHANNEL_1, computer_servo_position);

                total_counter++;

                 } else {

                    buf2[total_counter] = '-';

                }

                timer_counter++;

                // timer outside 5 seconds

        } else {

                //sprintf(buf, "Oof!\n\r");

                //vPrintString(buf);

                buf2[total_counter] = '!';

                timer_counter = 0;

                combo = 1;

                computer_servo_position = (((HAL_RNG_GetRandomNumber(&Rng_Handle)) % 1501) +
500.0);

                // set next position

                set_pwm_pulse(&htim5, TIM_CHANNEL_1, computer_servo_position);

                total_counter++;


        }

        vPrintString(buf2);


        sprintf(buf3, "Turn: %d\t Points: %d\t Combo: %2.0d\t Time: %2.0d\t Position: %4.0d\t
%4.0d\r\n", total_counter, points, combo, timer_counter, (int)player_servo_position,
(int)computer_servo_position);

        vPrintString(buf);

        sprintf(buf3, "\033[2A");

        vPrintString(buf);


} else {

         if(total_counter == 10){

         sprintf(buf, "\n\n\n\rFinal Score: %d\n\r", points);

         vPrintString(buf);
```

```
 }
total_counter++;
```

Once the game was created, tested, and found functional, a more advanced game was created. The advanced game only affected the scoring and difficulty levels, all of which were handled in the computer task. In the advanced game, the player was assigned 5 “lives” every time the player missed a servo match, they would lose a life. Additionally, there was no 10 turn limit. The game would only end once the player had lost all of their lives. In order to increase the difficulty, the amount of time between turns would slowly decrease from 5 seconds down to a minimum of 1 second. Additionally, there was the implementation of a “bonus” alongside the combo. The longer the player lasted, the larger the bonus got, which would directly increase their score. Overall, the end product resulted in a game that got progressively harder as it went on, but offered more and more points. The code below details some of the changes made.

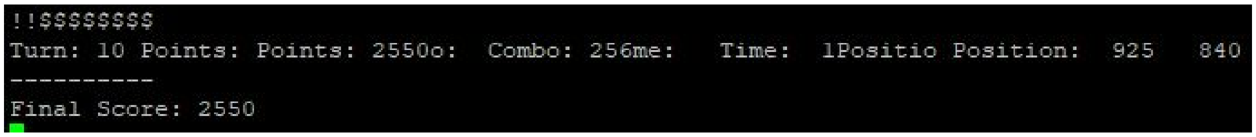
```
        // still within time seconds
        if (lives > 0){
                if (timer_counter <= time_challenge){
                        // checking if player is withing +/- 5% of the computer position
                        if (player_servo_position <= (computer_servo_position +
(computer_servo_position*0.05)) && player_servo_position >= (computer_servo_position
-(computer_servo_position*0.05))){
                         points = points + (10*combo)*((total_counter/5)+1);
                         if (combo <= 4){
                                 combo++;
                         }
                         timer_counter = 0;


                         // set next position
                        computer_servo_position = (((HAL_RNG_GetRandomNumber(&Rng_Handle)) %
1501) + 500.0);
                         set_pwm_pulse(&htim5, TIM_CHANNEL_1, computer_servo_position);
                         total_counter++;


                         if(time_challenge >= 10){
                        time_challenge = time_challenge - 1;
                        }
                } else {
                         buf2[total_counter] = '-';
                }
                timer_counter++;
```

Test Plan

The testing was very simple for the program. Under the hood, there were many components interlaying and communication with one another. But for the final product, if the servo moved in relation to how the board moved, then the initialization functions were set up correctly. Other than the servo and board moving as one, the system also had to have the following functions working correctly in order for the game to operate

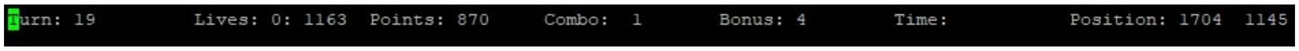
- If the player and computer servo were within a certain tolerance of one another, than the points would increment.
- The computer servo had to move the computer servo to a random location.
- The player was only allowed 5 seconds to match the servos. If they did not get it in that time then the computer servo would move anyway.
- The game had to end after 10 iterations for the default implementation
- The game had to output the final score using UART

All of these functions were vital to having the game work, which meant that play testing it would test everything all at once. Both the working default and advanced projects were demonstrated to the TA. The following figure shows the output of the UART terminal for the defualt game while Figure 2 shows the output termianl for the advanced game. Note that the top line would constantly update as the game was being played.



```
!!$$$$$$$$
Turn: 10 Points: Points: 2550o:  Combo: 256me:   Time:  1Positio Position:  925   840
----------
Final Score: 2550
```

Figure 1. Default UART Output.



```
Turn: 19         Lives: 0: 1163  Points: 870     Combo:  1       Bonus: 4        Time:           Position: 1704  1145
```

Figure 2. Advanced UART Output

Lessons Learned

Overall this exercise was very exciting. Advanced sensors such as gyroscopes were able to be used as inputs, which then could be used to create outputs for other sensors. It was a unique experience that really demonstrated how far the team had come. Additionally, the code was very easy to produce Since a more robust understanding of CUBE and FreeRTOS had been developed at the time of creating the project, more time was spent developing the systems rather than debugging and re-learning how to create them. In the end, not only were all the base requirements fulfilled, they were also surprised with the creation of the adanced game. The final product ended up being an outstanding success while the whole design process served as an excellent opportunity to interface with new sensors and gadgets.