

Lab 4 Part II: Automation & Security System for Cattle Farms

Nakseung Choi, 1572578

29-May-2022

Kejin Li, 1978130

Assignment: ECE474 Final Project

Introduction:

For the final project, we developed an automation and security system for cattle farms. The motivation of this project comes from Choi's uncle who runs a cattle farm alone with over 500 cows in South Korea. His day starts at 5 AM where he drives to the farm. Each day, he goes to the farm 3-5 times to feed the cattle, clean the farm, replace fodders, and check up on the cows. The farm is located in the countryside where it is vulnerable to thieves attempting to steal livestock with a truck in the middle of the night. Choi always thought that having an automation & security system would help his uncle significantly. As the name of our project implies, there are two main modes: automation mode and security mode. In addition, we also completed the part 3, which is run four tasks using free-RTOS. Two tasks are identical to the speaker and external LED task from the previous lab. The difference is that they now operate with the free-RTOS library. The other two functions are employed to compute the elapsed time for five FFTs.

Methods and Techniques:

To begin, a 5V DC power supply was employed to supply the input voltage to an LCD screen, 3-6V DC motor, DHT11 sensor, HC-SR501 PIR motion sensor, and three servo motors. For example, running the 3-6V DC motor outside of its absolute maximum voltage rating will affect its life and will kill it eventually. Even though the Arduino board is pretty accurate with the 5V output, the 5V DC power supply module was employed redundantly for overvoltage protection. We used a multimeter to check the voltage at the positive rail on the Arduino board. The figures are shown below:

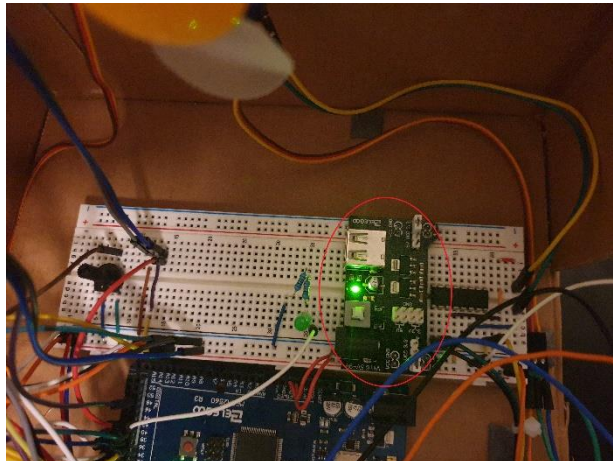


Figure 1. 5V power supply module.

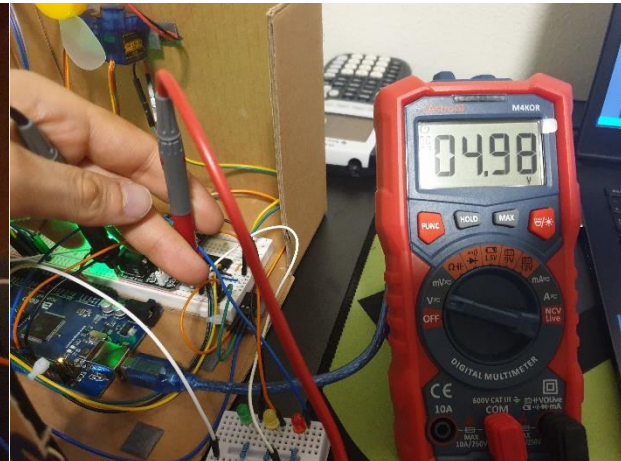


Figure 2. 5V in the positive rail.

Serial.print function was mainly used to debug several minor issues when sending and receiving data using xQueue function. For example, four different xQueuesSentToBack functions were used to send data to three servo motors and 3-6V DC motors. Using a Serial function with an if function with a condition of `!= pdTrue`, it allowed us to notice when a xQueueSendToBack function

fails to send data to xQueueReceive located in different cpp files. As we were expanding our project, the project became a multi-file project that made it difficult to write and debug our codes. Serial function was widely utilized to debug small data sending/receiving issues, which is shown below:

```

case 'B':
    if(*temp == 1){
        mctrl = MOTOR_RESET;
        if(xQueueSendToBack(TaskMotorQueue, &mctrl, 0) != pdTRUE){
            Serial.println("Failed to send to queue");
        }
    }
    break;

// servo now
case '5':
    if(*temp == 1){
        sctrl = UP;
        if(xQueueSendToBack(TaskServoQueue, &sctrl, 0) != pdTRUE){
            Serial.println("Failed to send to queue");
        }
    }
    break;
29
30 motorcontrol temp;
31 int speed = 0;
32 for(;;){
33     temp = MOTOR_RESET;
34     xQueueReceive(TaskMotorQueue, &temp, portMAX_DELAY);
35     Serial.print("Hey I got it!1: ");
36     Serial.println(temp);
37     switch(temp){
38     case MOTOR_RESET:
39         speed = 0;
40         analogWrite(6, speed);
41         *reset = 1;
42         break;

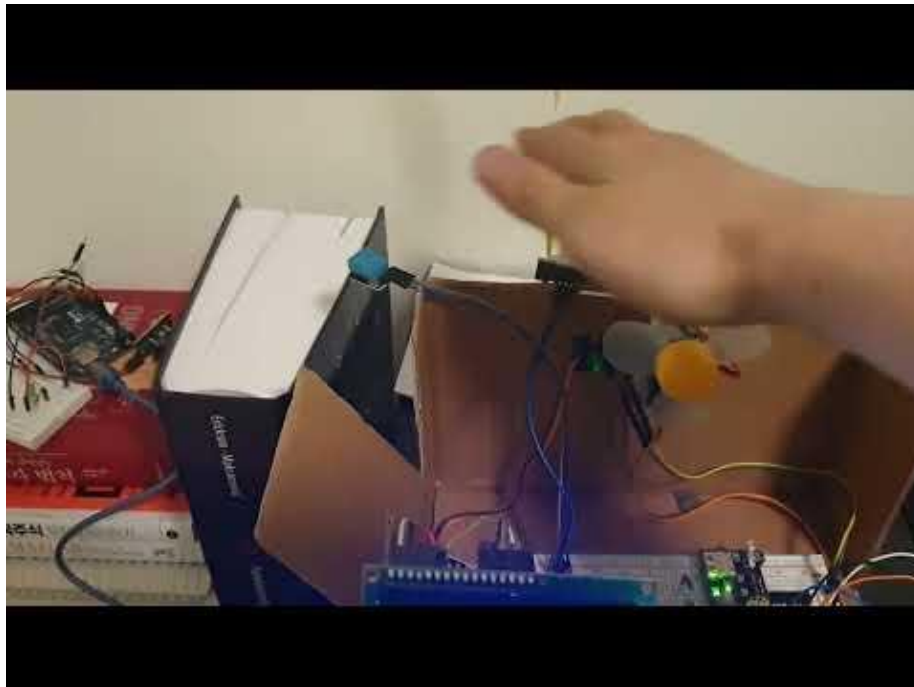
```

Figure 3 and 4. Serial prints (with if statement) are used to debug issues associated with xQueue.

In addition, keeping RT1 (blinking an external LED for 100ms) on all the time was useful. This allowed us to detect some errors in coding easily. For instance, the LED would not blink or stay on when there was an issue with priority setup or any collision between tasks.

Experimental Results:

To walk you through how the entire system works, users must type the passcode to activate the automation mode. Otherwise, the security mode remains on to protect the cows and farm. A HC-SR501 PIR motion sensor was installed to detect any moving objects at the entrance. When it is detected, the LEDs and alarm goes off. To turn them off, users must type the passcode on the keypad. The demo is shown below:



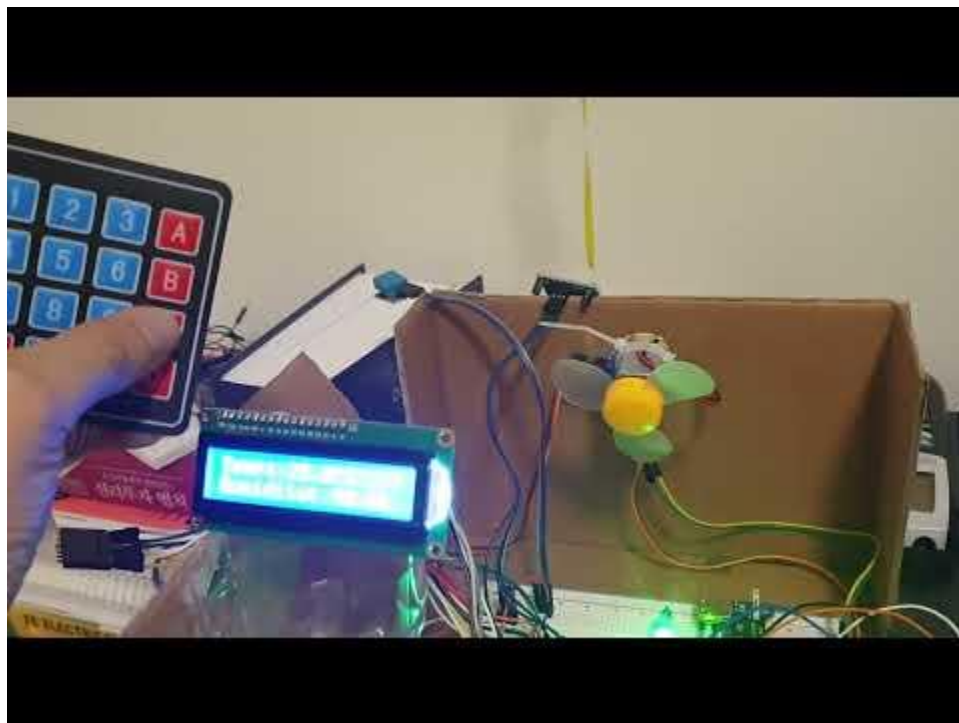
Video 1. Demo for the functionality of the security system.

As the automation mode is back on, the LCD screen starts to display the temperature and humidity of the farm because DHT11 is running whether it is in security mode or automation mode.



Figure 5. Temperature and humidity displayed on the LCD screen.

Users can control the fan using the membrane keypad. They can set the direction of the fan (including auto mode) and the speed of the fan. This is used for the purpose of having ventilation and a cooling system. Without this, the livestock would be living in dire conditions.



Video 2. Demo for the ventilation/cooling system.

Another servo motor was used for the food bank. Pressing 4 and 6, users can open and close the food bank to feed the cows. The figures are shown below:



Figure 6. Food bank open



Figure 7. Food bank closed

We also set the temperature and humidity threshold in case the farm gets too hot.

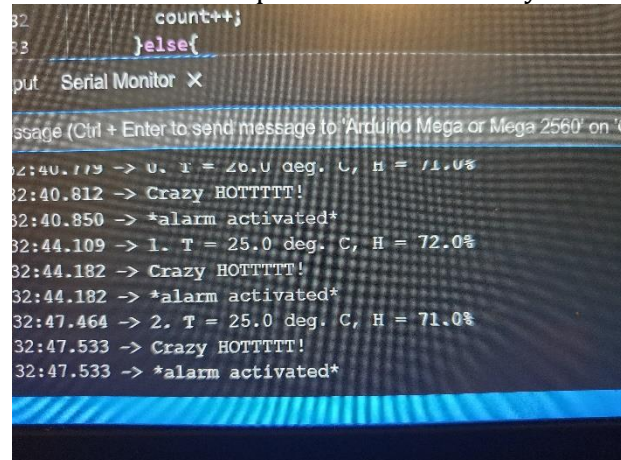


Figure 8 and 9. the LCD screen shows the alarm being activated as the temperature goes above 27°C or the humidity goes above 70%.

When users press "A" again, the automation mode turns off and switches back to the security system mode, activating the motion sensor and alarm system.

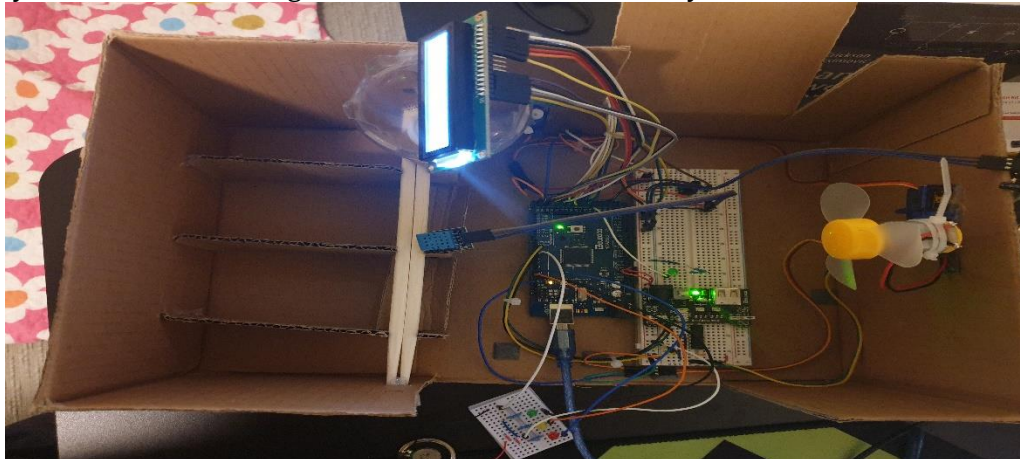


Figure 10. Overview of the system.

Experimental Results for Lab 4 part 3:

For Task 1(flashing an external LED) and Task 2(playing “Close Encounters Theme”), the same codes from the previous labs were employed but created using the free-RTOS library. As the figure 11 shows, it takes an average of 239.55 seconds for one FFT computation.

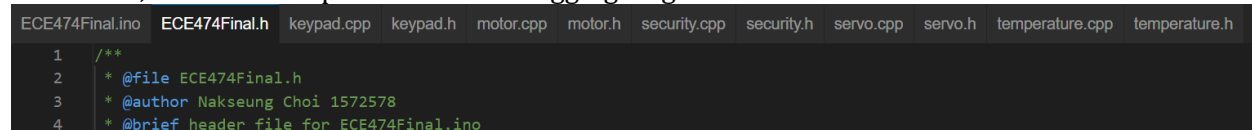
```
19:33:17.143 -> Elapsed time for 5 FFTs: 239.80
19:33:17.396 -> Elapsed time for 5 FFTs: 239.20
19:33:18.609 -> Elapsed time for 5 FFTs: 239.40
```

Figure 11. a screen shot of Lab 4 part 3 result.

Code Documentation:

1. Code Documentation for Final Project.

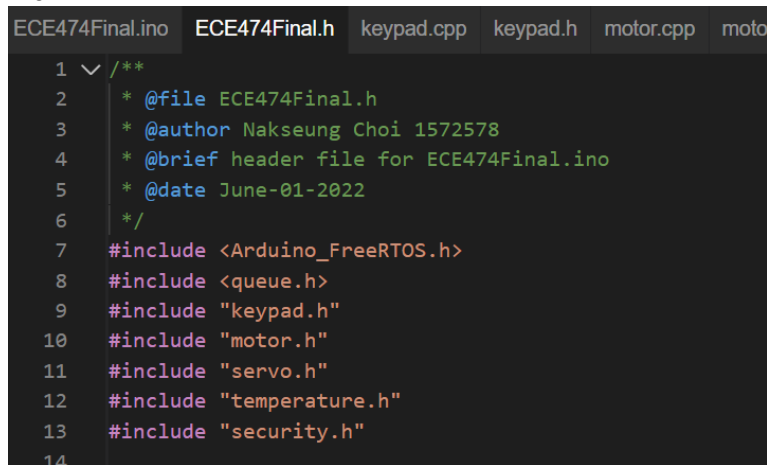
As we expanded our project, we realized that it would be beneficial to split each function into separate files with a separate header. Not only did it allow us to work on different tasks at the same time, but it also helped us in the debugging stage.



```
ECE474Final.ino  ECE474Final.h  keypad.cpp  keypad.h  motor.cpp  motor.h  security.cpp  security.h  servo.cpp  servo.h  temperature.cpp  temperature.h
1  /**
2   * @file ECE474Final.h
3   * @author Nakseung Choi 1572578
4   * @brief header file for ECE474Final.ino
```

Figure 12. Each function was created in a different cpp file with a header file.

To synchronize all of the tasks, we included all header files that contain function prototypes in the main header file.



```
ECE474Final.ino  ECE474Final.h  keypad.cpp  keypad.h  motor.cpp  moto
1  /**
2   * @file ECE474Final.h
3   * @author Nakseung Choi 1572578
4   * @brief header file for ECE474Final.ino
5   * @date June-01-2022
6   */
7  #include <Arduino_FreeRTOS.h>
8  #include <queue.h>
9  #include "keypad.h"
10 #include "motor.h"
11 #include "servo.h"
12 #include "temperature.h"
13 #include "security.h"
14
```

Figure 13. main header file including all of the other header files.

Starting with the keypad.cpp, we used a switch/case to assign different functions to different keys. For example, several enums (enumeration) were created in the main header file for the state of the fan, servos, and temperature. These were used everywhere and were constantly updating the state of each task. For example, when users press “1”, mctrl (motorcontrol enum) was set to FASTER. Using a xQueue function, this data was sent to the motor.cpp file to run the motor. Similarly, when users press “4,” sctrl was set to either LEFT, RIGHT, UP, or DOWN and then, xQueueReceived created in the servo.cpp will receive the data to move the corresponding servo.

```

switch(customKey){
    case '1':
        if(*temp == 1){
            mctrl = FASTER;
            if(xQueueSendToBack(TaskMotorQueue, &mctrl, 0) != pdTRUE){
                Serial.println("Failed to send to queue");
            }
        }
        break;

    case '2':
        if(*temp == 1){
            mctrl = SLOWER;
            if(xQueueSendToBack(TaskMotorQueue, &mctrl, 0) != pdTRUE){
                Serial.println("Failed to send to queue");
            }
        }
        break;

    case '9':
        if(*temp == 1){
            sctrl = RIGHT;
            if(xQueueSendToBack(TaskServoQueue, &sctrl, 0) != pdTRUE){
                Serial.println("Failed to send to queue");
            }
        }
        break;

    case '8':
        if(*temp == 1){
            sctrl = SERVO_RESET;
            if(xQueueSendToBack(TaskServoQueue, &sctrl, 0) != pdTRUE){
                Serial.println("Failed to send to queue");
            }
        }
        break;
}

```

Figure 14. Send data to motor and servo cpp files

```

xQueueReceive(TaskMotorQueue, &temp, portMAX_DELAY);
Serial.print("Hey I got it!1: ");
Serial.println(temp);
switch(temp){
    case MOTOR_RESET:
        speed = 0;
        analogWrite(6, speed);
        *reset = 1;
        break;

    case PAUSE:
        speed = 0;
        analogWrite(6, speed);
        Serial.print("Fan off.");
        break;

    case SERVO_RESET:
        degree = 90;
        Serial.println("RESET");
        myservo.write(degree);
        myservo2.write(degree);
        break;

    case UP:
        degree = 75;
        Serial.println("UP");
        myservo2.write(degree);
        break;
}

```

Figure 15. Receive data from keypad.cpp file.

To make certain tasks run either in security mode or automation mode, an int variable called mode was created in the setup file in the main function. This variable later goes into TaskKeypad, Tasktemperature, and TaskSecurity such that these tasks know which mode is activated either to run or stop tasks. The figure is shown below:

```

void setup() {
    Serial.begin(9600);
    while(!Serial){;}

    xTaskCreate(TaskExternalLED, "ExternalLED", 128, NULL, 3, NULL);

    volatile int auto_mode = 0;
    xTaskCreate(TaskMotor, "Motor", 500, &auto_mode, 1, NULL);
    xTaskCreate(TaskServo, "Servo", 500, &auto_mode, 1, NULL);
    volatile int mode = 0;
    xTaskCreate(TaskKeypad, "Keypad", 500, &mode, 1, NULL); // 0
    xTaskCreate(TaskTemperature, "temperature", 500, &mode, 1, NULL); // 0
    xTaskCreate(TaskSecurity, "security", 128, &mode, 1, NULL);
}

```

Figure 16. int mode going into Tasks. These tasks are set to run in different modes.

In each task, a pointer was created to store this variable and later used mainly with if statement to run or stop tasks in different modes. In the figure below, `int *temp` stores `(int*)pvParameters`. This pointer was later used as a condition for an if statement to run this task only when the security system is on. (0 indicates security mode and 1 indicates automation mode.)

```
int *temp = (int*)pvParameters;
unsigned long time = 0;
int count = 0;
int deactivate = 0;
tempthreshold threshold;

while(1){
    threshold = NORMAL;
    // xQueueReceive(TaskTemperatureQueue, &threshold, portMAX_DELAY);
    if(*temp == 0){
        pirValue = digitalRead(pirPin);
        if(pirValue == 1){
            time = time + 10000;
            deactivate = 1;
        }
        if(threshold == SUPERHOT){
            time = time + 10000;
            deactivate = 1;
        }
    }
}
```

Figure 17. `*temp` stores mode from the main .ino to run it during security mode.

Referring to one of the given example codes from the library, the LCD screen was set. This LCD display provides two rows that users can display anything in. To write something in the first row, `lcd.setCursor(0,0)` had to be initialized first and `lcd.print("something you want to display")` to display what you want. Similarly, `lcd.setCursor(0,1)` was initialized to display words in the second row. The figures are shown below:

```
lcd.setCursor(0,0);
lcd.print("Temp: ");
lcd.print(temperature, 1);
lcd.print((char)223);
lcd.print("C ");
lcd.setCursor(0,1);
lcd.print("Humidity: ");
lcd.print(humidity, 1);
lcd.print("% ");
```

Figure 18. how to display words on the LCD display.

Lastly, to be able to turn on LEDs and alarms using the HC-RS510 motion sensor, it was assigned to pin 48. Whenever the motion detector detects a moving object, the pin reads the input as 1. Using `digitalWrite`, we set to turn on LEDs and alarms when the pin receives 1. The figure is shown below:

```

pirValue = digitalRead(pirPin);
if(pirValue == 1){
    time = time + 10000;
    deactivate = 1;
}
if(threshold == SUPERHOT){
    time = time + 10000;
    deactivate = 1;
}
while(time != 0){
    digitalWrite(ledPin1, pirValue);
    vTaskDelay(pdMS_TO_TICKS(100));
    LED_PORTB &= ~(PIN12);
    digitalWrite(ledPin2, pirValue);
    vTaskDelay(pdMS_TO_TICKS(100));
    LED_PORTB &= ~(PIN11);
}

```

Figure 19. pirValue stores the input 1 from pin 48 and digitalWrite turns on the LEDs.

2. Code Documentation for Lab 4 Part 3

For lab 4 part 3, we created a total of 4 free-RTOS tasks: a speaker, an external LED, and two for RT3p1 and RT4p0. For the speaker and external LED tasks, we brought the same codes that we did in lab 3 and made them run in Free-RTOS. The figures are shown below:

```

xTaskCreate(TaskSpeaker, "TaskSpeaker", 128, NULL, 1, NULL);
xTaskCreate(TaskExternalLED, "ExternalLED", 128, NULL, 1, NULL);

```

Figure 20. Free RTOS tasks for the speaker and external LED

For the new tasks, FFTs, we created three functions: RT3p0, RT3p1, and RT4p0. Each function has a different functionality, which is described below:

RT3p0:

This is a scheduler for RT3p1 and RT4p0. RT3p1 and RT4p0 were created as a task using a free-RTOS function, xTaskCreate, as well as a double array that stores a number of the random variables (512 for us) we declared. In addition, two xQueue functions were created, one with the size of the double pointer and the other with the size of long. We also iterated through the stack sizes for RT4p0 and found that 4500 is sufficient to run RT4p0 for 512 random values.

RT3p1:

This is a free RTOS task that receives the double array with the 512 random variables through xTaskCreate from RT3p0. To do that, we created a double pointer to store the double array. The figure is shown below:

```

void RT3p1( void *pvParameters ){
    long wall_clock_time;
    double *temp = (double*)pvParameters;
}

```

Figure 21. RT3p1 receives a double array with 512 random numbers from RT3p0.

This data is sent to RT4p0 and used to compute 5 FFTs. This function also receives “wall_clock_time” from RT4p0. This long variable is divided by 5 (NFFTS) and is printed out.

RT4p0:

This is another free RTOS task in which all computations happen. This receives the double array from RT3p1. Using a built-in FFT library function, FFT.Compute() and two for loops, it computes the 5 FFTs. To be able to find the time that takes for the 5 FFTs, we created a unsigned long variable called “time” and set it to millis() in the beginning of the function. And then, after the computation is done, this time is subtracted by millis() again. This gives us the time that took for 5 FFTs to compute. And then, this time is sent to RT3p1 and divided by 5 (NFFTS) to be printed out.

Overall Performance Summary:

Lab 4 part 3 was pretty straightforward. It took us only 2 hours. However, for the project, we wished we had a little more time. We would have implemented a remote control as a secondary control method to run tasks in automation mode. We also would like to have added more features to our system, such as a cleaning system for animal waste. Within the given amount of time, however, we managed our time and collaborated well to be able to take our project this far. We are satisfied with the outcome of our project.

Teamwork Breakdown:

For this project, Kejin and I have met up several times to work on the hardware together. Coding was done remotely by both of us. We equally contributed to this project, learned from each other, and completed this project. Both of us were responsible for building architecture, coding, debugging, integrating hardware and software. Whenever we had time, we set up a meeting in the lab, discussed, and shared our codes and ideas with each other.

Discussion and conclusions:

The hardest part of this project was the time constraint. We were given only one and half weeks to come up with an idea for this project that actually solves a real-world problem and implement it on the Arduino board. It was challenging to think creatively in a short period of time. Initially, we tried using all of the sensors and components, hoping that it would help us with idea crafting. However, thanks to Choi’s uncle, as soon as we came up with this idea, we immediately got started working on the system. Overall, it was a good opportunity to learn more about a real-time operating system that is actively being used in the industry. Integrating hardware and software was a rewarding experience. With this skillset, we feel that we can make anything using the free-RTOS and the Arduino board. We would like to continue to study more about embedded systems and different MCUs.