Student Declaration of Authorship

HERIOT WATT UNIVERSITY
UK | DUBAI | MALAYSIA

Course code and name:	B31DG – Embedded Software
Type of assessment:	**Individual**
Coursework Title:	Assignment 3
Student Name:	Jack B Mavor
Student ID Number:	H00298763

Declaration of authorship. By signing this form:

- **I declare** that the work I have submitted for individual assessment OR the work I have contributed to a group assessment, is entirely my own. I have NOT taken the ideas, writings or inventions of another person and used these as if they were my own. My submission or my contribution to a group submission is expressed in my own words. Any uses made within this work of the ideas, writings or inventions of others, or of any existing sources of information (books, journals, websites, etc.) are properly acknowledged and listed in the references and/or acknowledgements section.

- I confirm that I have read, understood and followed the University's Regulations on plagiarism as published on the University's website, and that I am aware of the penalties that I will face should I not adhere to the University Regulations.

- I confirm that I have read, understood and avoided the different types of plagiarism explained in the University guidance on Academic Integrity and Plagiarism

Student Signature *(type your name):* *Jack Burgoyne Mavor*

Date: *14/04/2022*

Copy this page and insert it into your coursework file in front of your title page.
For group assessment each group member must sign a separate form and all forms must be included with the group submission.

Your work will not be marked if a signed copy of this form is not included with your submission.

Embedded software – Assignment 3

15/04/2022

Jack B. Mavor

H0028763

Github repo: B31DG_AS3_jbm5 (source files and revision log available)

Demo video: Demo video

Problem

In this assignment the systems built in assignment 2 are rebuilt using a real time operating system (RTOS). The RTOS will carry out the following series of tasks at varying frequencies, these tasks and their rates are as follows:

Task	Frequency (Hz)	Period (ms)	Description
1	50	20*	Output a basic digital signal that will be used as a basic watchdog
2	5	200	Check a push button signal
3	1	1000	Measure the frequency of a square waveform
4	24	42*	Read an analogue input signal
5	24	42*	Filter the analogue signal
6	10	100	Sleep the computer for 1000 ticks using the "__asm__ __volatile__ ("nop");" command
7	3	333*	Determine an error code based on the current filtered analogue signal
8	3	333*	Output the error code via an LED
9	0.2	5000	Output various data points in a CSV style serial output

*Rounded to the nearest millisecond

Development

While there are many open source RTOS's available, for this project I will be using "freeRTOS" as the ESP32 Arduino library includes a version of it built in. As most of the tasks have identical functionality to their assignment 2 counterparts their core functionality will remain the same, however these functions have been modified to avoid using global variables as much as possible. Where a task still requires access to variables it has been adjusted to either use Mutex's or Queue's. For any task requiring access the "data" struct and its components, a mutex is used to prevent memory access issues (this is shown in the bellow code snippet), without this protection the memory assigned to the struct variable could become corrupted due to being accessed by multiple tasks at the same time.

```
// Mutex protection statement
// prevents memory access issues
// attempts to take the mutex access token
// only continues if the access token is availible
if(xSemaphoreTake(data_mut, portMAX_DELAY) == pdTRUE){
    duration = pulseIn(PULSE_IN, LOW);
    data.frequency = 1000000.0 / (duration * 2);

    // return mutex access token
    xSemaphoreGive(data_mut);
}
```

Figure 1: Task 3 using a mutex to determine is it's safe to access the struct

For any task that required one way transfer of a single variable a queue was used, this was used for 2 cases, for passing the analogue voltage value from task 4 to task 5, and for passing the averaged analogue voltage from task 5 to task 7. In both cases a 1 entry queue was used as only the most recent versions of these data points were required by the other tasks.

```
// Mutex protection statement
// prevents memory access issues
// attempts to take the mutex access token
// only continues if the access token is availible
if(xSemaphoreTake(data_mut, portMAX_DELAY) == pdTRUE){

    for (int i = 0; i < 4; i++)
    {
        y += analogs[i];
    }

    y = y / 4;

    data.analog = y;

    xQueueSend(average_queue, &y, 100);

    // return mutex access token
    xSemaphoreGive(data_mut);
}
```

Figure 2: task 5 adding the average value to the queue

```
// tasks next item from average queue and assign is to variable x
if (xQueueReceive(average_queue, &x, 100)){
    if ( x > (3.3 / 2))
    {
        error_code = 1;
    }
    else
    {
        error_code = 0;
    }
}
```

Figure 3: Task 7 receiving and dequeuing the average value from the queue

While the completed code contains very little in the way of wasted CPU usage, there are a couple of tasks that required the usage of either delays directly of functions that use delays. In particular, task 1 and task 3.

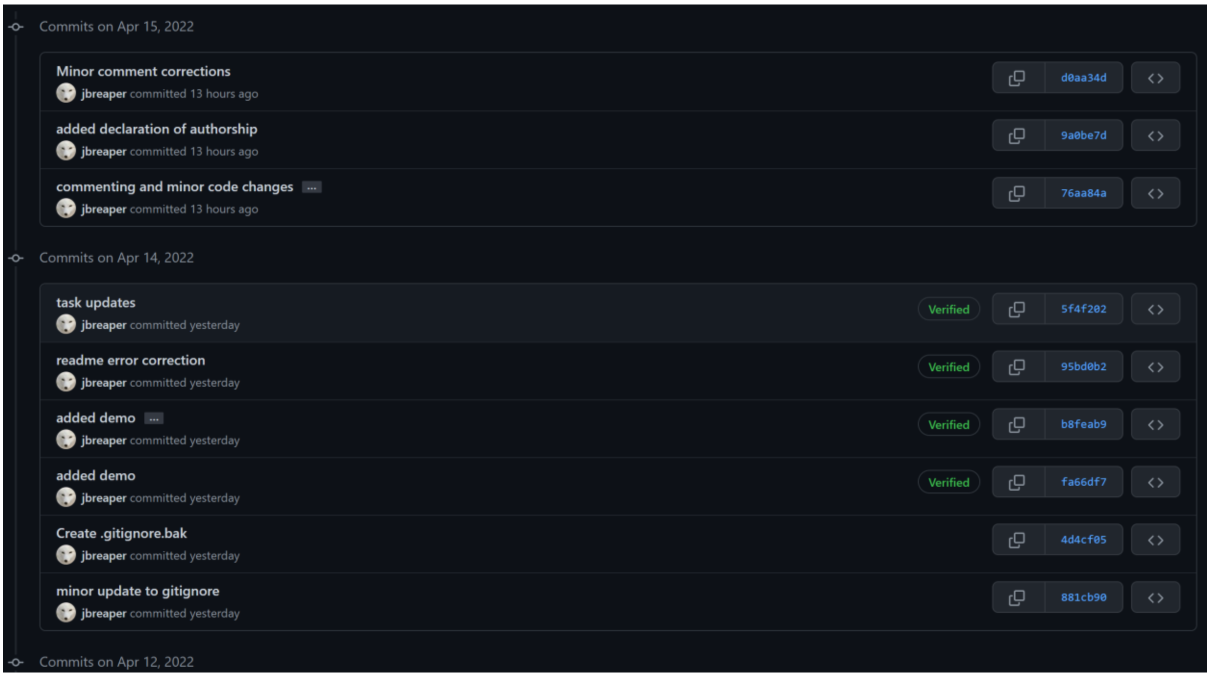
- Task 1 uses the "delayMicrosecond();" function which, due to the length of the delay, this should not extend the amount of time that the task uses the CPU.
- Task 3 uses the "pulseIn();" function, this function measures the width of a pulse by waiting until a pulse has completed.

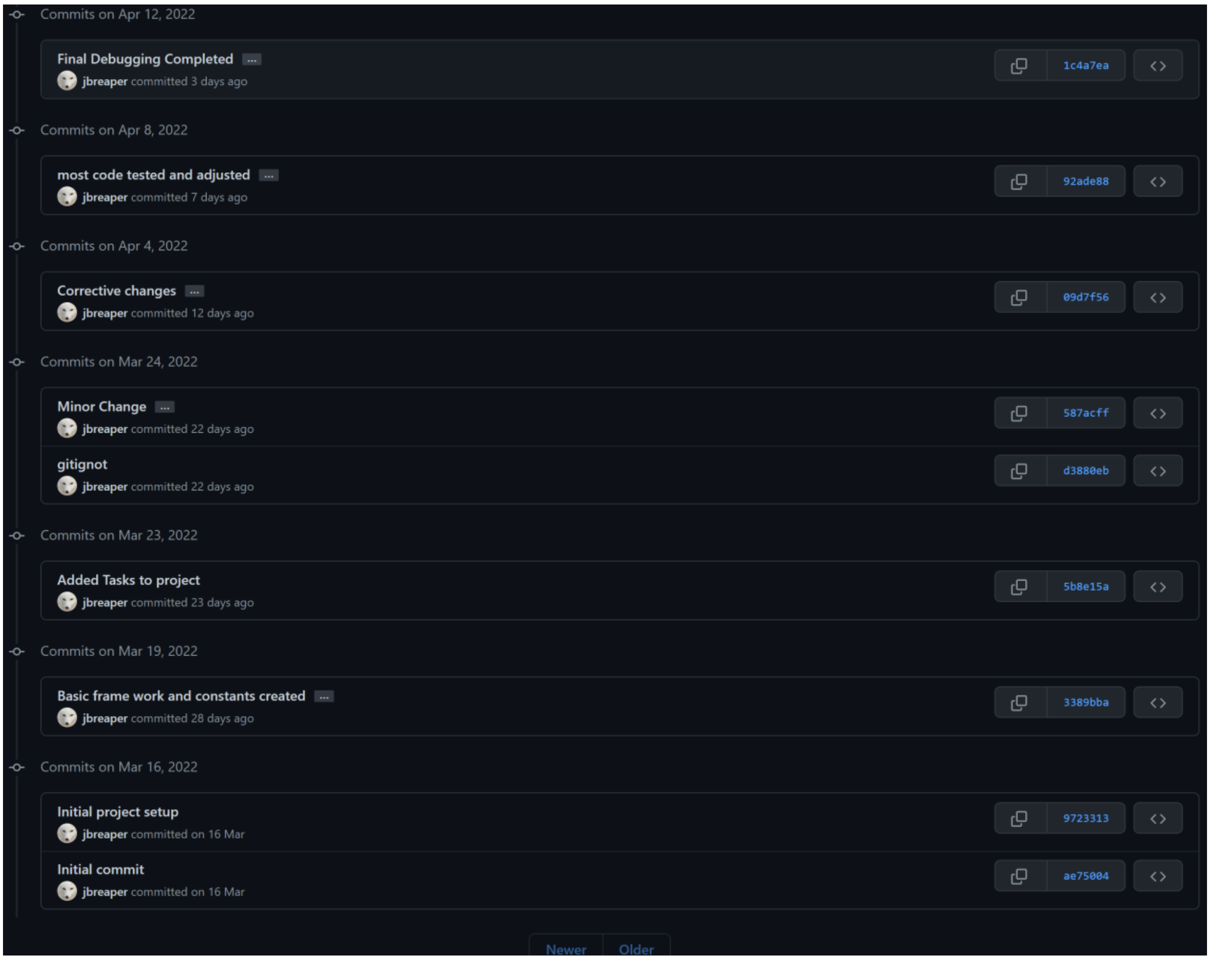
While steps were taken to try and ensure that memory usage was kept to a minimum, such as minimising the number of global variables and redundant variables, some additional avenues for memory waste reduction were not implemented during this project. The main method of memory optimisation that was not appropriately implemented was determining the "high water mark" for each of the tasks. This would have been done using the "uxTaskGetStackHighWaterMark()" function built into freeRTOS, this function returns a value representing the number of bytes (while this is task in normal freeRTOS would be in words, the ESP version of the function uses bytes) between the tasks usage and the stack size. After using this function, a task's stack size would then be shrunk by half that number, this is repeated until the stack size has been minimised while also not overflowing.

Additionally, the task priorities were based upon knowledge of which ones are likely to be more important to the overall functionality of the project as well as some basic testing. However, these priorities could be improved though further testing and additional analysis of the possible schedules.

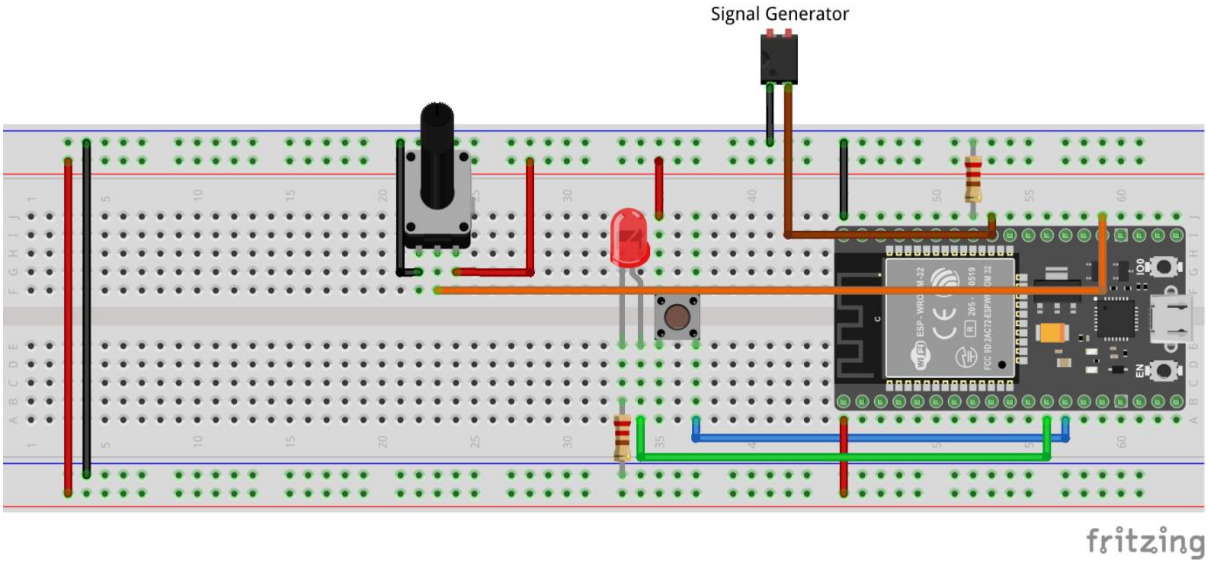
Appendix

A. Commit History





B. Wiring Diagram



C. Code

```
/* Embbeded Software - Assignment 3
 * This project was programmed for for the ESP32 microcontroller, it uses freeRTOS to run a series of tasks
 * based on the provided brief.
 *
 * -------
 *  BRIEF
 * -------
 * For this assignment the tasks created in assignment 2 were recreated to utilise the freeRTOS framework.
 * An additional requirement was added for this assignment, that being that in order for task 9 to print
 * the data, the button from task 2 must be pressed.
 * This the RTOS system must complete the following tasks at the related frequencies.
 * _____________________________________________________________________________________________________________________
 * |  Task  | Description                                                                  | Frequency   | Periodicity |
 * _____________________________________________________________________________________________________________________
 * |  1     | Output a (digital) watchdog waveform (with same length and period of the ‘Normal’ |    50Hz     |   20ms      |
 * |        | operation of SigB in Assignment 1). Timings should be within 5%.             |             |             |
 * _____________________________________________________________________________________________________________________
 * |  2     | Monitor one digital input (to be connected to a pushbutton/switch or a signal |    5Hz      |   200ms     |
 * |        | generator for students using Proteus).                                        |             |             |
 * ____\________________________________________________________________________________________________________________
 * |  3     | Measure the frequency of a 3.3v square wave signal. The frequency will be in the |    1Hz      |   1000ms    |
 * |        | range 500Hz to 1000Hz and the signal will be a standard square wave (50% duty |             |             |
 * |        | cycle). Accuracy to 2.5% is acceptable.                                       |             |             |
 * _____________________________________________________________________________________________________________________
 * |  4     | Read one analogue input. The analogue input must be connected to a maximum of |    24Hz     |   42ms      |
 * |        | 3.3V, using a potentiometer.                                                  |             |             |
 * _____________________________________________________________________________________________________________________
 * |  5     | Compute filtered analogue value, by averaging the last 4 readings.           |    24Hz     |   42ms      |
 * |        |                                                                               |             |             |
 * _____________________________________________________________________________________________________________________
 * |  6     | Execute 1000 times the following instruction:                                 |    10Hz     |   100ms     |
 * |        | __asm__ __volatile__ ("nop");                                                 |             |             |
 * |        | The statement could be repeated using a single loop, or broken down into multiple |             |             |
 * |        | loops (e.g. to be executed in different slots of the cyclic executive).      |             |             |
 * _____________________________________________________________________________________________________________________
 * |  7     | Perform the following check:if (average_analogue_in > half of maximum range for |    3Hz      |   333ms     |
 * |        | analogue input):                                                              |             |             |
 * |        |     error_code = 1                                                            |             |             |
 * |        | else:                                                                         |             |             |
 * |        |     error_code = 0                                                            |             |             |
 * _____________________________________________________________________________________________________________________
 * |  8     | Visualise error_code using an LED.                                            |    3Hz      |   333ms     |
 * |        |                                                                               |             |             |
 * _____________________________________________________________________________________________________________________
 * |  9     | Check is the push button is currently in it's pressed state, if it is log the |    0.3Hz    |   5000ms    |
 * |        | following information every five (5) seconds (in comma separated            |             |             |
 * |        | format, e.g. CSV) to the serial port:                                         |             |             |
 * |        |   - State of the digital input (pushbutton / switch);                        |             |             |
 * |        |   - Frequency value (Hz, as an integer);                                      |             |             |
 * |        |   - Filtered analogue input.                                                  |             |             |
 * _____________________________________________________________________________________________________________________
 */


// determine the number of cores availible
#if CONFIG_FREERTOS_UNICORE
#define ARDUINO_RUNNING_CORE 0
#else
#define ARDUINO_RUNNING_CORE 1
#endif

// pin assignments
#define LED 14          // Pin G14, Used to output error signal
#define WD 19           // Pin G19, Used to output the watchdog signal
#define PB1 12          // Pin G12, Used to read the state of a push button
#define A_IN 4          // Pin G4,  Used to read an analogue input signal
#define PULSE_IN 18     // Pin G18, Used to read a digital input signal

// Rate of task (ms)
/* while hz to ms gives 20.4, this
```

```c
 * has been rounded to 42 as the ESP32's
 * freeRTOS tick rate is 1ms
 */
#define R_T1 20         // 50 Hz
#define R_T2 200        // 5 Hz
#define R_T3 1000       // 1 Hz
/* while hz to ms gives 41.666666...this
 * has been rounded to 42 as the ESP32's
 * freeRTOS tick rate is 1ms
 */
#define R_T4 42         // 24 Hz
#define R_T5 42         // 24 Hz
#define R_T6 100        // 10 Hz
/* while hz to ms gives 333.33333... this
 * has been rounded to 333 as the ESP32's
 * freeRTOS tick rate is 1ms
 */
#define R_T7 333        // 3 Hz
#define R_T8 333        // 3 Hz
#define R_T9 1000       // 0.2 Hz

// alternate to vtaskdelay that allows the delay to
// be defined using milliseconds rather than ticks
#define TaskDelay(x) vTaskDelay(x/portTICK_PERIOD_MS)

// prototype functions for tasks
void task_1(void *pvParameters);
void task_2(void *pvParameters);
void task_3(void *pvParameters);
void task_4(void *pvParameters);
void task_5(void *pvParameters);
void task_6(void *pvParameters);
void task_7(void *pvParameters);
void task_8(void *pvParameters);
void task_9(void *pvParameters);
```

```c

// Defined Queue handles for analog signal
// and avaraged analogue signal queues
static QueueHandle_t analog_queue;
static QueueHandle_t average_queue;

// Mutex to protect the "data" struct variable
static SemaphoreHandle_t data_mut;

// error code variable for low analogue voltage check
volatile int error_code;

// Struct to collect all data to be output in task 9
struct Data
{
    bool button = false;
    int frequency = 0;
    float analog = 0;
} data;

// generate pulse of with 50us
void task_1(void *pvParameters)
{
    (void) pvParameters;
    for (;;)
    {
        digitalWrite(WD, HIGH);
        // 50 microsecond delay
        delayMicroseconds(50);
        digitalWrite(WD, LOW);

        // delays task for rate
        // before restarting
        TaskDelay(R_T1);
    }
```

```c
}

// read input of a button on pin PB1
void task_2(void *pvParameters)
{
    (void) pvParameters;
    for (;;)
    {
        // Mutex protection statement
        // prevents memory access issues
        // attempts to take the mutex access token
        // only continues if the access token is availible
        if(xSemaphoreTake(data_mut, portMAX_DELAY) == pdTRUE){
            data.button = digitalRead(PB1);

            // return mutex access token
            xSemaphoreGive(data_mut);
        }

        // delays task for rate
        // before restarting
        TaskDelay(R_T2);
    }
}

// determine frequency of digital signal on pin PULSE_IN
void task_3(void *pvParameters)
{
    float duration = 0;

    (void) pvParameters;
    for (;;)
    {
        // Mutex protection statement
        // prevents memory access issues
```

```
        // attempts to take the mutex access token
        // only continues if the access token is availible
        if(xSemaphoreTake(data_mut, portMAX_DELAY) == pdTRUE){
            duration = pulseIn(PULSE_IN, LOW);
            data.frequency = 1000000.0 / (duration * 2);

            // return mutex access token
            xSemaphoreGive(data_mut);
        }

        // delays task for rate
        // before restarting
        TaskDelay(R_T3);
    }
}

// read analogue input on pin A_IN
void task_4(void *pvParameters)
{
    int x = 0;

    (void) pvParameters;
    for (;;)
    {
        x = analogRead(A_IN);
        xQueueSend(analog_queue, &x, 100);

        // delays task for rate
        // before restarting
        TaskDelay(R_T4);
    }
}

// Average last 4 analog input readings
void task_5(void *pvParameters)
{
    float analogs[4] = {0, 0, 0, 0};

    (void) pvParameters;
    for (;;)
    {
        int x = 0;
        float y = 0;

        if(xQueueReceive(analog_queue, &x, 100)){
            for (int i = 1; i < 4; i++)
            {
                analogs[i - 1] = analogs[i];
            }

            analogs[3] = x * (3.3 / 4095);
        }

        // Mutex protection statement
        // prevents memory access issues
        // attempts to take the mutex access token
        // only continues if the access token is availible
        if(xSemaphoreTake(data_mut, portMAX_DELAY) == pdTRUE){

            for (int i = 0; i < 4; i++)
            {
                y += analogs[i];
            }

            y = y / 4;

            data.analog = y;

            xQueueSend(average_queue, &y, 100);

            // return mutex access token
            xSemaphoreGive(data_mut);
        }

        // delays task for rate
        // before restarting
        TaskDelay(R_T5);
    }
}

// use "__asm__ __volatile__ ("nop");" 1000 times
void task_6(void *pvParameters)
{
    (void) pvParameters;
    for (;;)
    {
        for (int i = 0; i < 1000; i++)
        {
            __asm__ __volatile__("nop");
        }

        // delays task for rate
        // before restarting
        TaskDelay(R_T6);
    }
}

// determine error code based on average analogue reading
void task_7(void *pvParameters)
{
    (void) pvParameters;
    for (;;)
    {
        float x = 0;

```

```
        // tasks next item from average queue and assign is to variable x
        if (xQueueReceive(average_queue, &x, 100)){
            if ( x > (3.3 / 2))
            {
                error_code = 1;
            }
            else
            {
                error_code = 0;
            }
        }

        // delays task for rate
        // before restarting
        TaskDelay(R_T7);
    }
}

// light LED based on error code
void task_8(void *pvParameters)
{
    (void) pvParameters;
    for (;;)
    {
        digitalWrite(LED, error_code);

        // delays task for rate
        // before restarting
        TaskDelay(R_T8);
    }
}

// print; button PB1 state, Frequency of PULSE_IN, and average of alalogue input A_IN
// This data is presented in a CSV format
void task_9(void *pvParameters)
{
    (void) pvParameters;
    for (;;)
    {
        // Mutex protection statement
        // prevents memory access issues
        // attempts to take the mutex access token
        // only continues if the access token is availible
        if(xSemaphoreTake(data_mut, portMAX_DELAY) == pdTRUE){
            if (data.button == 1)
            {
                Serial.print(data.button);
                Serial.print(", \t\t\t");
                Serial.print(data.frequency);
                Serial.print(", \t\t");
                Serial.print(data.analog);
                Serial.print("\n");
            }

            // return mutex access token
            xSemaphoreGive(data_mut);
        }

        // delays task for rate
        // before restarting
        TaskDelay(R_T9);
    }
}

void setup()
{
    Serial.begin(115200);

    // setup pins
    pinMode(LED, OUTPUT);
    pinMode(WD, OUTPUT);
    pinMode(PB1, INPUT);
    pinMode(A_IN, INPUT);
    pinMode(PULSE_IN, INPUT);

    // header to assist with reading the CSV formatted output
    Serial.println("-------------------------------------");
    Serial.println("Switch, \tFrequency, \tInput");
    Serial.println("-------------------------------------");

    //setup mutex in order to protect the "data" struct
    data_mut = xSemaphoreCreateMutex();

    //setup queues for analog readings and averaged analogue readings
    analog_queue = xQueueCreate(1, sizeof(float));
    average_queue = xQueueCreate(1, sizeof(float));

    // setup freeRTOS tasks for each of the task functions
    /* xTaskCreate(
    *       task function,
    *       name for debugging purposes,
    *       stack size,
    *       pvParamer
    */

    xTaskCreate(
        task_1,
        "task 1",
        512,
        NULL,
        4,
        NULL);

    xTaskCreate(
        task_2,
```

```
        "task 2",
        512,
        NULL,
        3,
        NULL);

    xTaskCreate(
        task_3,
        "task 3",
        1024,
        NULL,
        3,
        NULL);

    xTaskCreate(
        task_4,
        "task 4",
        1024,
        NULL,
        2,
        NULL);

    xTaskCreate(
        task_5,
        "task 5",
        1024,
        NULL,
        1,
        NULL);

    xTaskCreate(
        task_6,
        "task 6",
        512,
        NULL,
        3,
        NULL);

    xTaskCreate(
        task_7,
        "task 7",
        1024,
        NULL,
        4,
        NULL);

    xTaskCreate(
        task_8,
        "task 8",
        512,
        NULL,
        3,
        NULL);

    xTaskCreate(
        task_9,
        "task 9",
        1024,
        NULL,
        4,
        NULL);
}

void loop(){}
```