COMP ENG 4DS4

Lab 2 - Introduction to FreeRTOS

Professor: Mohamed Hassan

Submitted on *February 28th, 2025* by: GROUP 07

Aidan Mathew - mathea40 - 400306142

Aaron Rajan - rajana8 - 400321812

Sameer Shakeel - shakes4 - 400306710

Chris Jiang - jiangc46 - 400322193

adheres to the Academic Integrity Policy of McMaster University and the Code of Conduct of the Professional Engineers of Ontario. **Submitted by Aidan Mathew, Aaron Rajan, Sameer Shakeel, and Chris Jiang.**

Declaration of Contributions:

All students contributed to the lab report and did the write-up.

Student	Contributions
Aidan Mathew	Worked together on all problems, demo P1
Aaron Rajan	Worked together on all problems, demo P4
Chris Jiang	Worked together on all problems, demo P2
Sameer Shakeel	Worked together on all problems, demo P3

Experiments

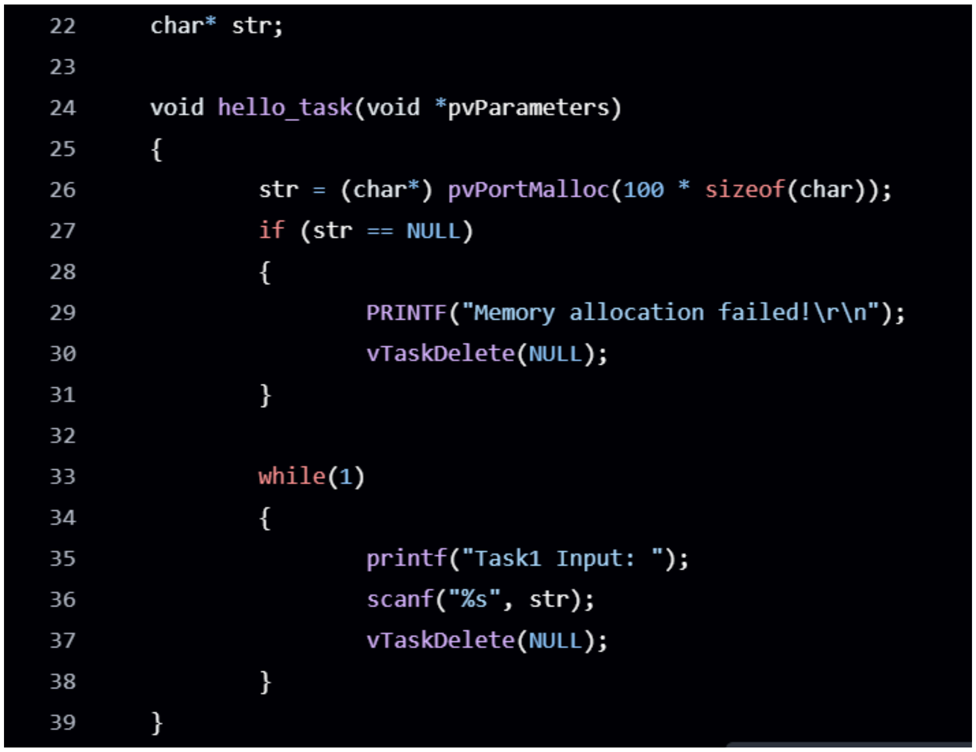
Experiment 1 - FreeRTOS Hello World

In part A, we created a task that prints “Hello World” in the console. In part B, we built over the previous part by creating another task that accepts arguments passed from xTaskCreate. To accept the string as an argument to the function, we modify the 4th parameter from NULL to a void pointer that points to the string defined at the top of the file.

Problem 1: Write a program that contains two tasks. The first task is responsible for taking an input string from the user through the console and **saving the string globally**. After receiving the string, the task should delete itself. On the other hand, the second task waits until the first string is available, and then it keeps printing it every second. The priority of the first task is 2 while it is 3 for the second task.

In our solution, we create two FreeRTOS tasks. The first one, “hello_task” accepts a user input string, stores it globally, and then deletes itself. The second one, “hello_task2” prints the stored string every second.

See below for our Task 1 implementation:



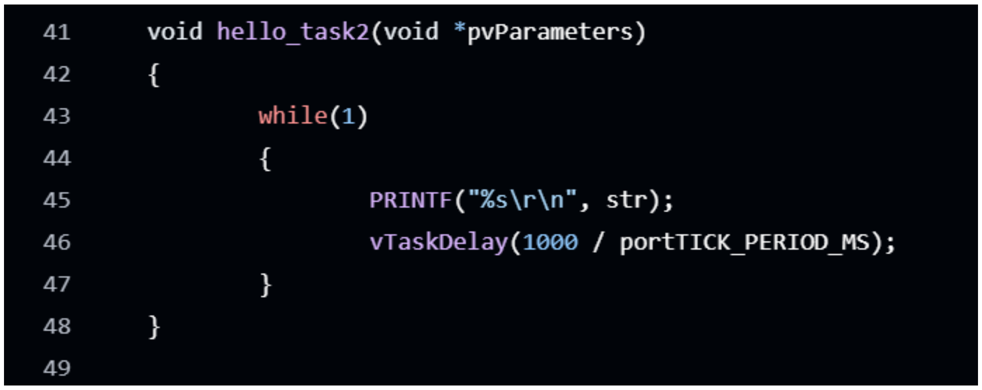
```
char* str;

void hello_task(void *pvParameters)
{
        str = (char*) pvPortMalloc(100 * sizeof(char));
        if (str == NULL)
        {
                PRINTF("Memory allocation failed!\r\n");
                vTaskDelete(NULL);
        }

        while(1)
        {
                printf("Task1 Input: ");
                scanf("%s", str);
                vTaskDelete(NULL);
        }
}
```

This task dynamically allocates memory for the string using pvPortMalloc(). FreeRTOS requires dynamic memory allocation to manage heap usage. If the string is NULL then the allocation fails. The while loop then waits to read a string from the user input and stores it in the global "str" variable. Finally, vTaskDelete(NULL) is called after getting input. This prevents unnecessary CPU usage.

See below for our Task 2 implementation:



```
void hello_task2(void *pvParameters)
{
        while(1)
        {
                PRINTF("%s\r\n", str);
                vTaskDelay(1000 / portTICK_PERIOD_MS);
        }
}
```

This task reads the global string variable and continuously prints the stored string every second. This uses vTaskDelay to pause execution when in the waiting state, allowing other tasks to run.

Experiment 2 - Inter-task Communication and Synchronization

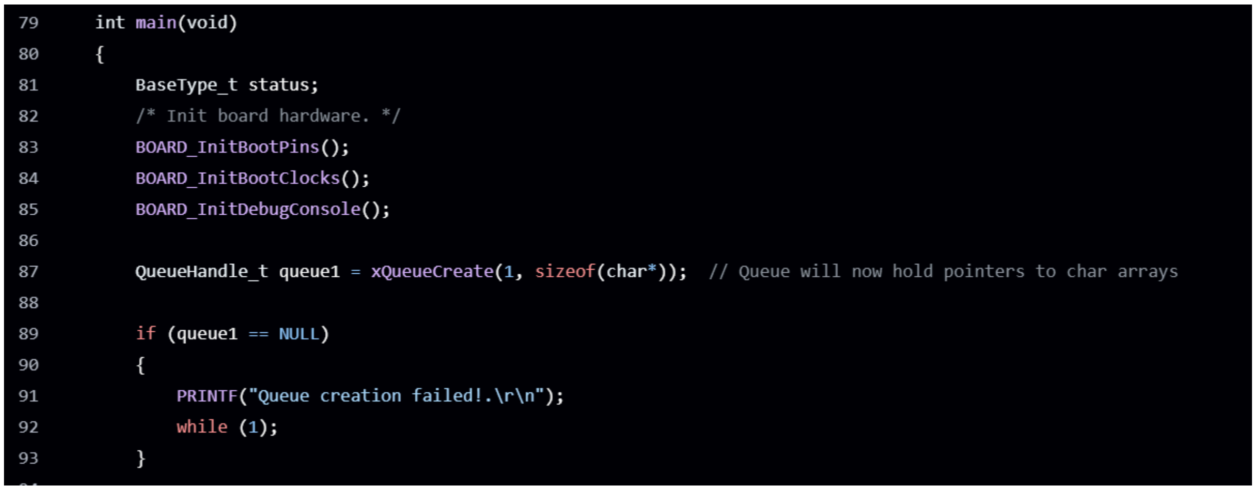
This experiment introduced three common techniques for inter-task communication and synchronization.

<u>Part A – Queues</u>

Queues in FreeRTOS allow tasks to send and receive data in a thread-safe way. A producer task writes to the queue, and a consumer task reads from it. First, we set up the queue in the main function that can store one integer. We then created the producer and consumer tasks, which pass the queue as an argument to both of the tasks so they can access it. Since both of the tasks have a priority of 2, FreeRTOS uses Round-Robin scheduling. We then defined the producer queue which increments a counter every second and then sends the counter to the queue using "xQueueSendToBack()". If the queue is full, the task blocks indefinitely (postMAX_DELAY) until space is available. We then use vTaskDelay() to delay for one second, allowing other tasks to run. Next, we defined the consumer queue, which waits for data in the queue using "xQueueRecieve()". This blocks indefinitely until data is available and then prints the received counter value.

Problem 2: Repeat Problem 1 using queues.

In problem 2 we modified problem 1 by using a queue instead of a global variable to transfer a string between two tasks. We first created a queue that can store 1 element at a time (in this case a “char*” pointer). The queue can hold one pointer to a string at a time. See below for the code snippet.

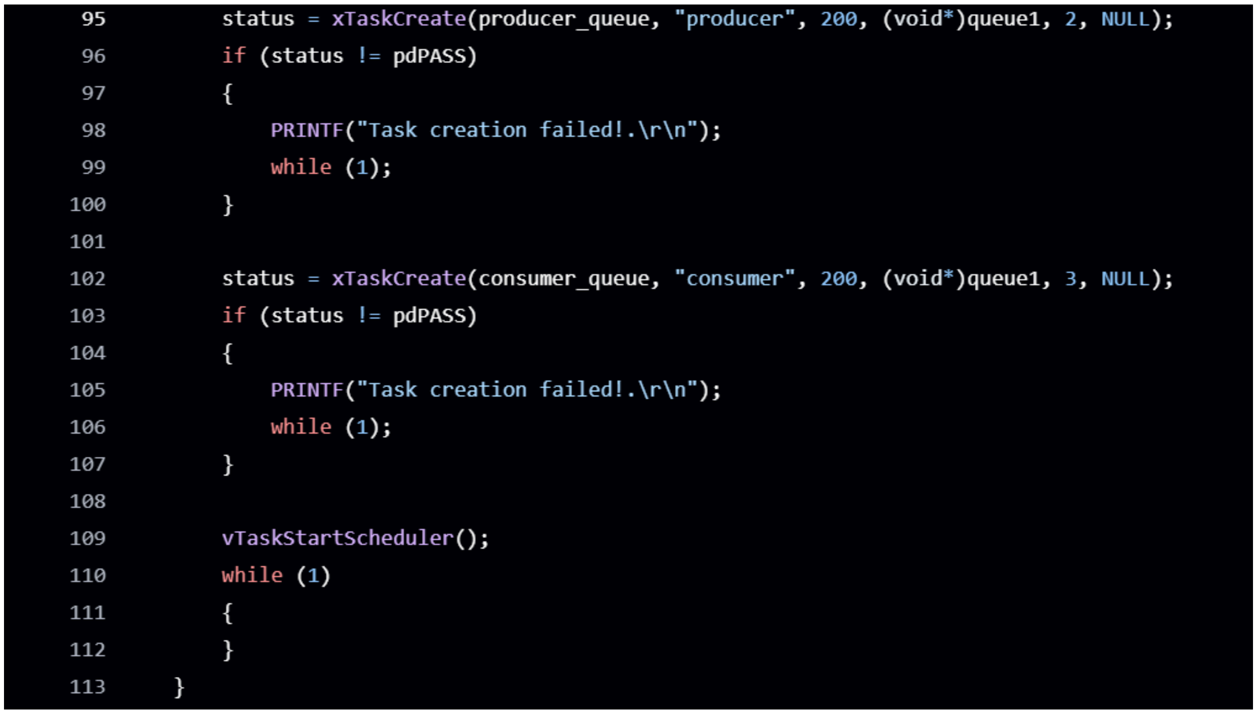


```
int main(void)
{
    BaseType_t status;
    /* Init board hardware. */
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();

    QueueHandle_t queue1 = xQueueCreate(1, sizeof(char*));  // Queue will now hold pointers to char arrays

    if (queue1 == NULL)
    {
        PRINTF("Queue creation failed!.\r\n");
        while (1);
    }
```

As seen above, if the queue is NULL then the queue creation fails, so it prints an error message. Next, we created the tasks, producer_queue task with priority 2, and consumer_queue task with priority 3. We then passed the queue as a parameter so both tasks could access the same queue. See the screenshot below for the code implementation.

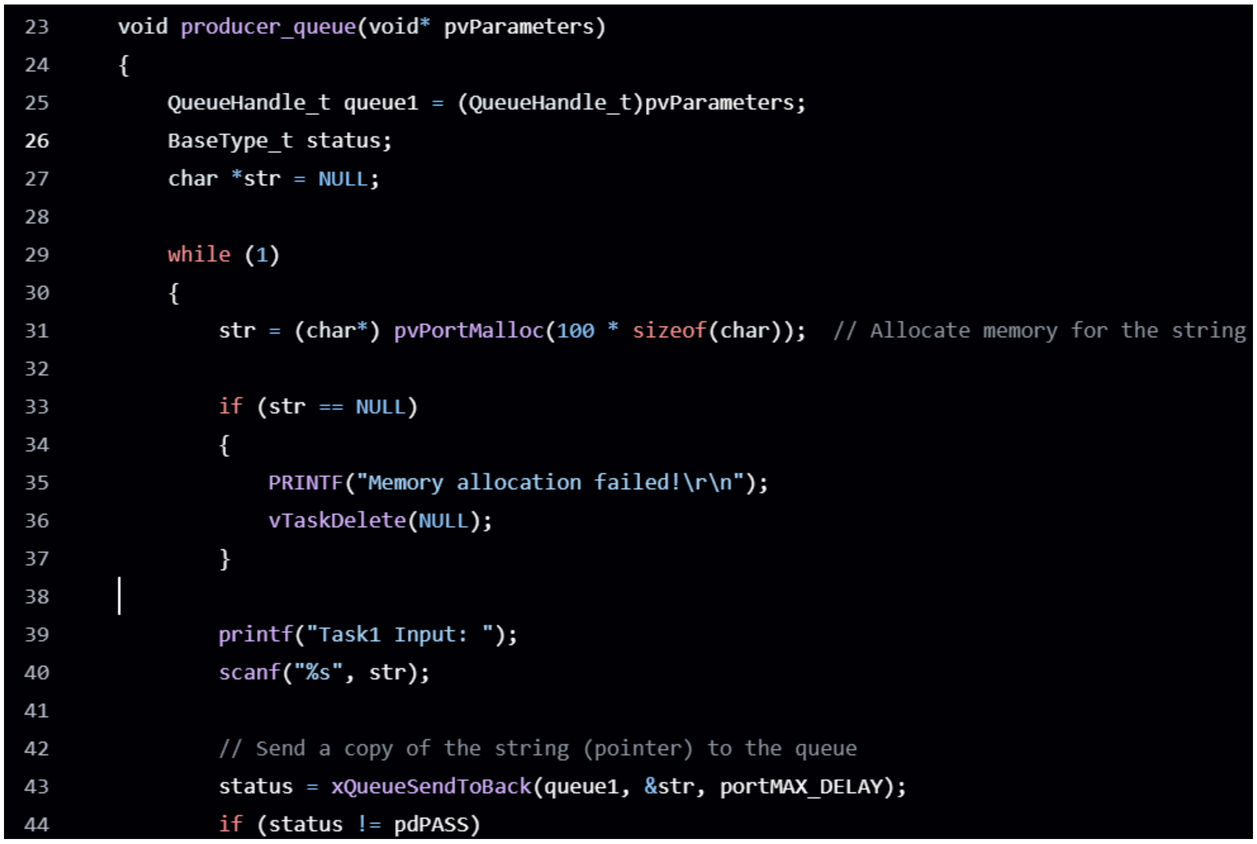


```
    status = xTaskCreate(producer_queue, "producer", 200, (void*)queue1, 2, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1);
    }

    status = xTaskCreate(consumer_queue, "consumer", 200, (void*)queue1, 3, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1);
    }

    vTaskStartScheduler();
    while (1)
    {
    }
}
```

Now the producer task is ready first. This task accepts user input and sends it to the queue. The producer_queue function first retrieves the queue handle from pvParamters and declares “str” as a pointer that will store the user input. Similar to problem 1, it then dynamically allocates memory, takes the user input and sends the data back to the queue using xQueueSendToBack(). See the code snippet below for the producer_queue function.



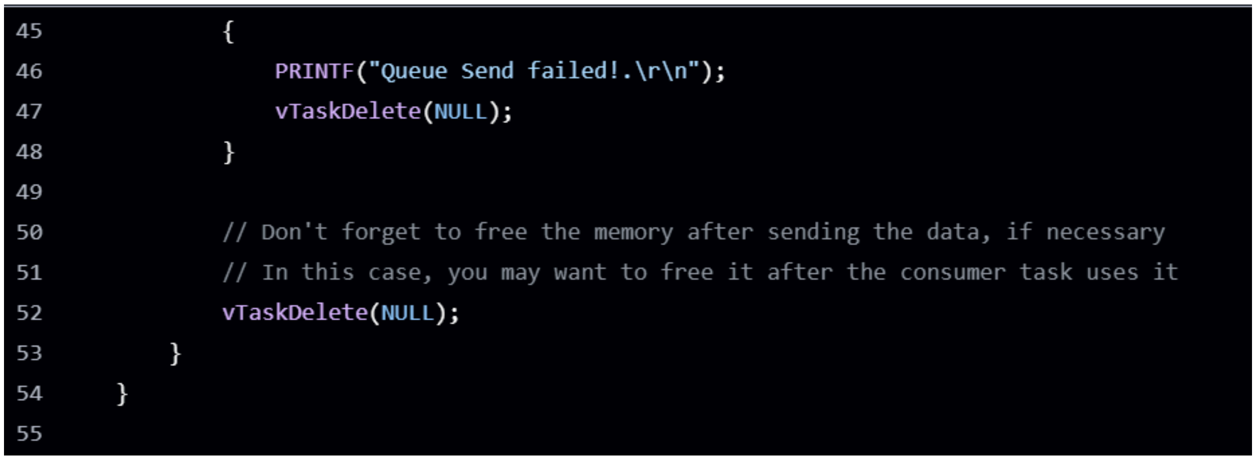
```c
void producer_queue(void* pvParameters)
{
    QueueHandle_t queue1 = (QueueHandle_t)pvParameters;
    BaseType_t status;
    char *str = NULL;

    while (1)
    {
        str = (char*) pvPortMalloc(100 * sizeof(char));  // Allocate memory for the string

        if (str == NULL)
        {
            PRINTF("Memory allocation failed!\r\n");
            vTaskDelete(NULL);
        }

        printf("Task1 Input: ");
        scanf("%s", str);

        // Send a copy of the string (pointer) to the queue
        status = xQueueSendToBack(queue1, &str, portMAX_DELAY);
        if (status != pdPASS)
```

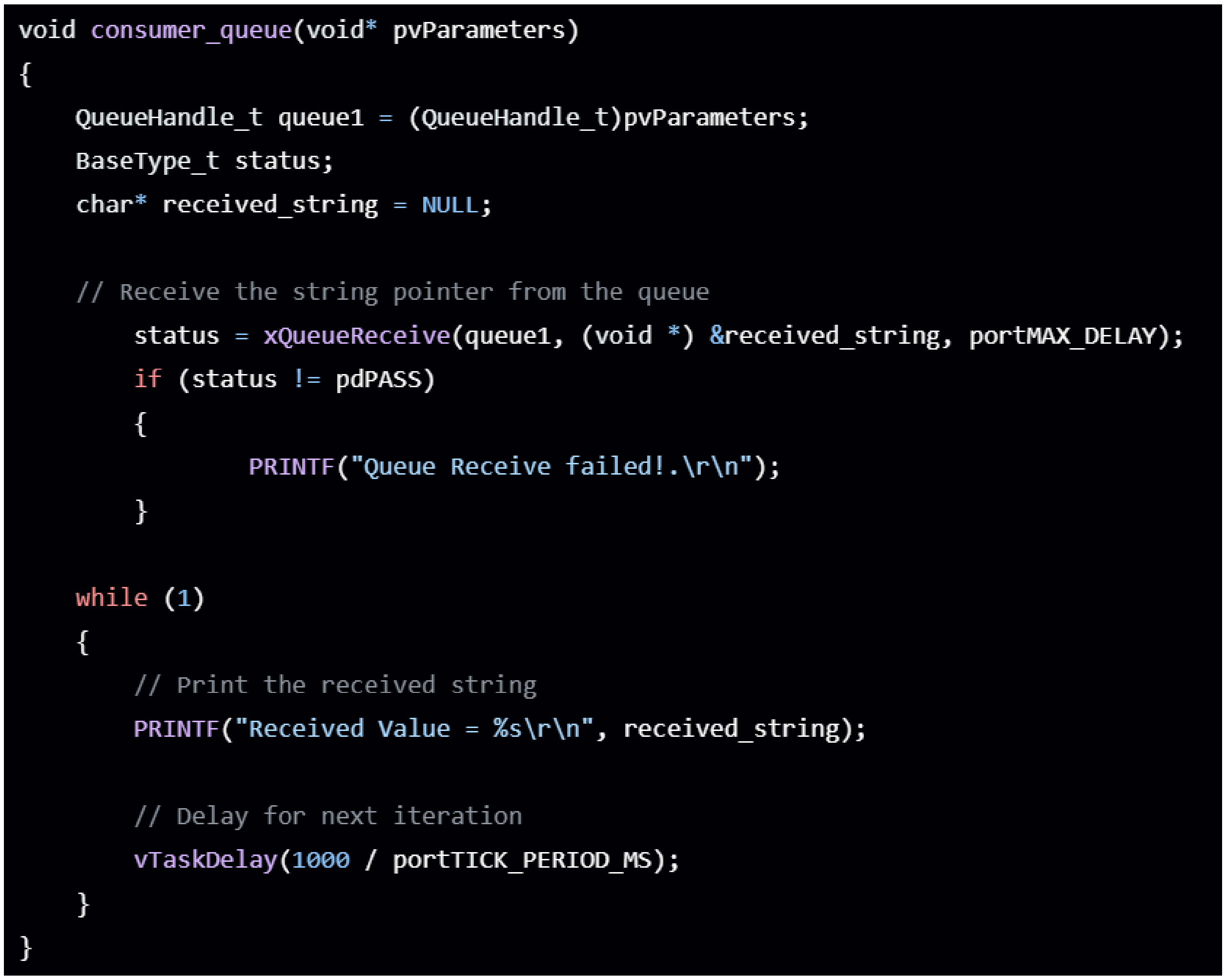


```c
        {
            PRINTF("Queue Send failed!.\r\n");
            vTaskDelete(NULL);
        }

        // Don't forget to free the memory after sending the data, if necessary
        // In this case, you may want to free it after the consumer task uses it
        vTaskDelete(NULL);
    }
}

```

Next, we have the consumer task which is the consumer_queue function. This task receives the string from the queue and prints it every second. It waits indefinitely, for a string from the queue using xQueueReceive, and stores the received point in received_string. See below for the consumer_queue function.



```
void consumer_queue(void* pvParameters)
{
    QueueHandle_t queue1 = (QueueHandle_t)pvParameters;
    BaseType_t status;
    char* received_string = NULL;

    // Receive the string pointer from the queue
        status = xQueueReceive(queue1, (void *) &received_string, portMAX_DELAY);
        if (status != pdPASS)
        {
                PRINTF("Queue Receive failed!.\r\n");
        }

    while (1)
    {
        // Print the received string
        PRINTF("Received Value = %s\r\n", received_string);

        // Delay for next iteration
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

Problem 3

1. Can we use a single counting producer semaphore for the previous application? Explain

your answer.

No, we cannot use a single counting producer semaphore for the previous application. The reason is that the two consumer tasks (consumer1_sem and consumer2_sem) need to synchronize with the producer task (producer_sem) independently. If we use a single counting semaphore for both consumers, they would not be able to distinguish between signals intended for each other.

In the current implementation, the producer task increments a shared counter and signals both consumer tasks using two separate semaphores (producer1_semaphore and producer2_semaphore). This allows each consumer task to independently wait for its own signal and process the counter value. If we were to use a single counting semaphore, both consumer tasks would be waiting on the same semaphore, and it would be impossible to ensure that each consumer task processes the counter value correctly without interfering with the other.

2. Modify Problem 1 by adding a third task with priority 3 that prints the string with

capital letters. Synchronize the three tasks using semaphores

The code implements a FreeRTOS-based application with three tasks that work together to take user input, print it, and print a capitalized version of the input. The tasks are synchronized using semaphores to ensure proper execution order. The first task, hello_task, is responsible for taking user input and storing it in a shared buffer (str). It also copies the input to another buffer (strCAP) for capitalization. After taking input, it signals the input_semaphore to notify the second task, hello_task2, that input is ready. hello_task2 waits for this semaphore, prints the received string, and then signals the capitalization_semaphore to notify the third task, hello_task3, that it can proceed with capitalization. hello_task3 capitalizes the string in strCAP, prints it, and signals the print_semaphore to allow hello_task2 to continue. This synchronization ensures that the tasks execute in the correct sequence: input , print, capitalize, and print. The hello_task deletes itself after completing its job, while hello_task2 and hello_task3 run in a loop, continuously processing new input. The use of semaphores ensures that no task proceeds until the previous task has completed its work, maintaining proper synchronization and order of operations. The application demonstrates a simple producer-consumer pattern with task synchronization in FreeRTOS.

Problem 4

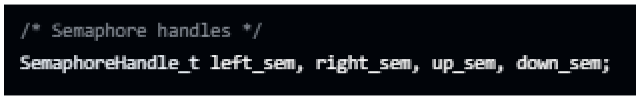
1. Can the priority of the producer task be similar or higher than the priority of the

consumer task? Explain your answer.

In this experiment, the priority of the producer task can indeed be similar to or higher than that of the consumer task, but the system's behaviour will vary depending on the synchronization mechanism and task design. The provided code uses an event group for synchronization, where the producer task (producer_event) sets event bits based on user input, and the consumer task (consumer_event) waits for and processes these bits. If the producer and consumer have the same priority, FreeRTOS schedules them in a round-robin fashion, allowing both to share CPU time. However, if the producer has a higher priority, it may set event bits faster than the consumer can process them, leading to delayed processing by the consumer. Despite this, the event group ensures no data loss, as it retains the bits until they are cleared. In the given implementation, the consumer has a higher priority (3) than the producer (2), ensuring prompt processing of event bits. This design prevents the event group from accumulating too many bits and avoids priority inversion, as both tasks use blocking operations (scanf for the producer and xEventGroupWaitBits for the consumer) that yield the CPU when waiting. Thus, the system functions efficiently, demonstrating the importance of proper task prioritization and synchronization in real-time systems.

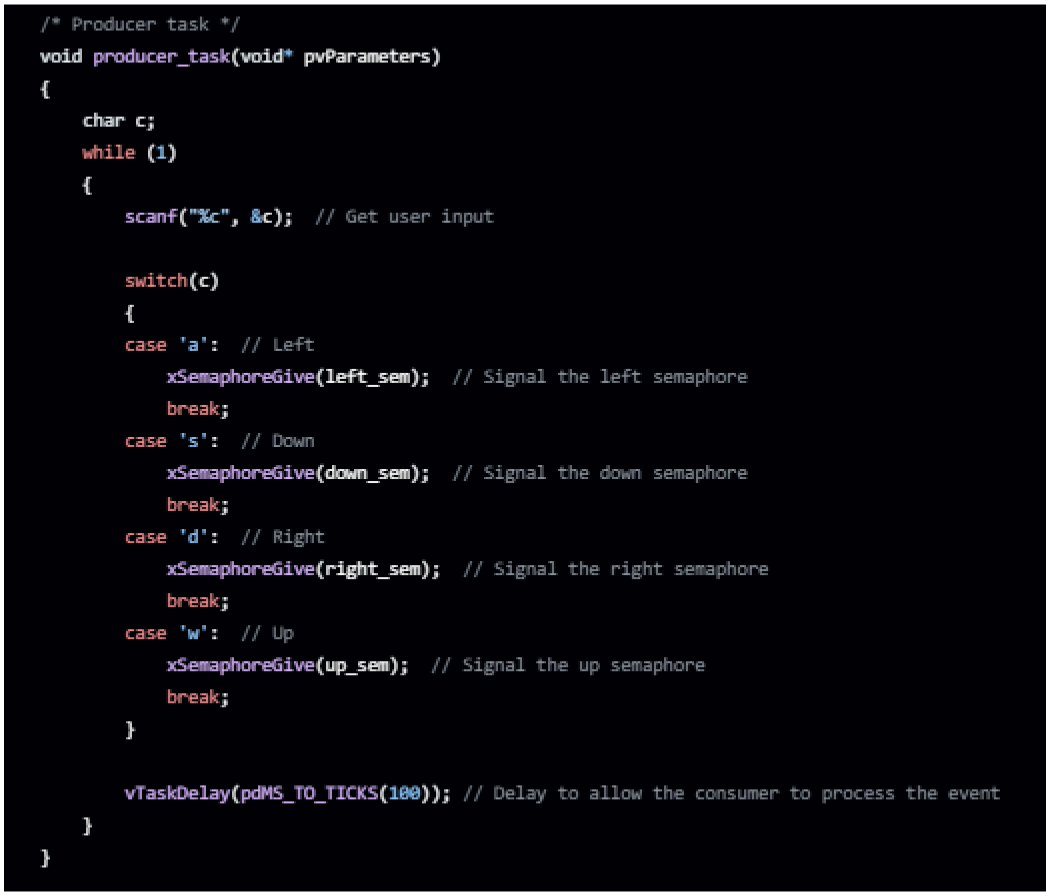
2. Repeat the experiment's application by using semaphores instead of event groups.

Global variables are not allowed for this exercise.



```
/* Semaphore handles */
SemaphoreHandle_t left_sem, right_sem, up_sem, down_sem;
```

Four binary semaphores are declared to represent the four directions: left, right, up, and down. These semaphores are used for synchronization between the producer and consumer tasks.

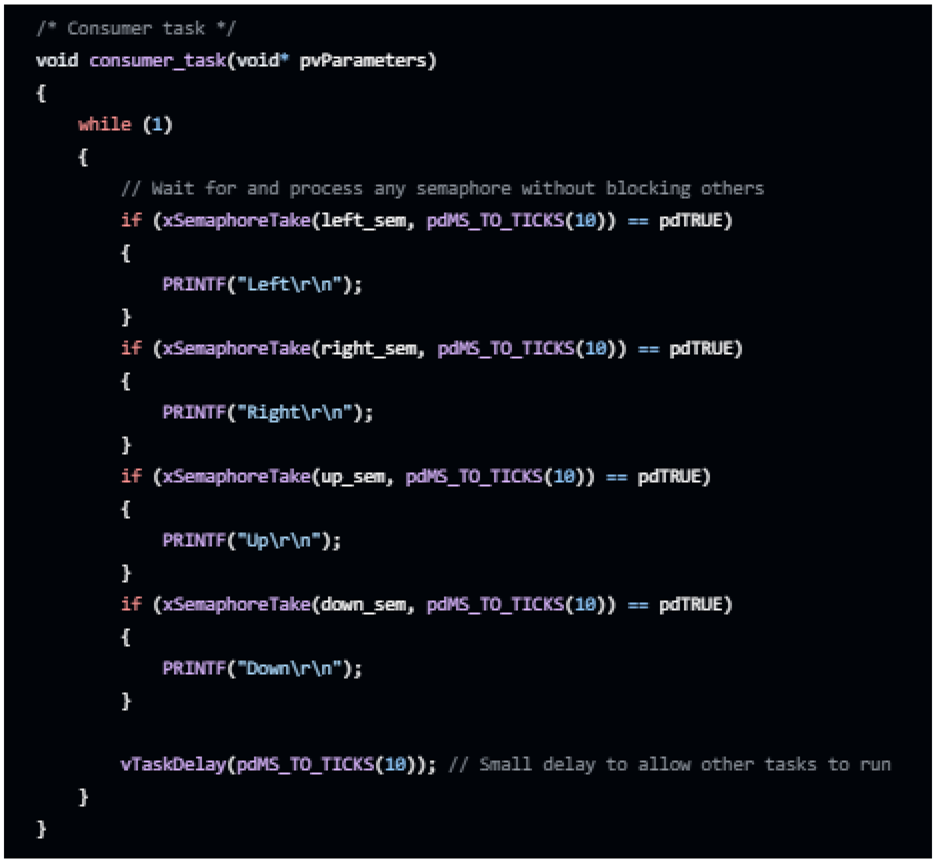


```
/* Producer task */
void producer_task(void* pvParameters)
{
    char c;
    while (1)
    {
        scanf("%c", &c);  // Get user input

        switch(c)
        {
        case 'a':  // Left
            xSemaphoreGive(left_sem);  // Signal the left semaphore
            break;
        case 's':  // Down
            xSemaphoreGive(down_sem);  // Signal the down semaphore
            break;
        case 'd':  // Right
            xSemaphoreGive(right_sem);  // Signal the right semaphore
            break;
        case 'w':  // Up
            xSemaphoreGive(up_sem);  // Signal the up semaphore
            break;
        }

        vTaskDelay(pdMS_TO_TICKS(100)); // Delay to allow the consumer to process the event
    }
}
```

This producer task reads the user input and signals the corresponding semaphore based on the input and adds a delay to allow the consumer task to process the event.



```
/* Consumer task */
void consumer_task(void* pvParameters)
{
    while (1)
    {
        // Wait for and process any semaphore without blocking others
        if (xSemaphoreTake(left_sem, pdMS_TO_TICKS(10)) == pdTRUE)
        {
            PRINTF("Left\r\n");
        }
        if (xSemaphoreTake(right_sem, pdMS_TO_TICKS(10)) == pdTRUE)
        {
            PRINTF("Right\r\n");
        }
        if (xSemaphoreTake(up_sem, pdMS_TO_TICKS(10)) == pdTRUE)
        {
            PRINTF("Up\r\n");
        }
        if (xSemaphoreTake(down_sem, pdMS_TO_TICKS(10)) == pdTRUE)
        {
            PRINTF("Down\r\n");
        }

        vTaskDelay(pdMS_TO_TICKS(10)); // Small delay to allow other tasks to run
    }
}
```

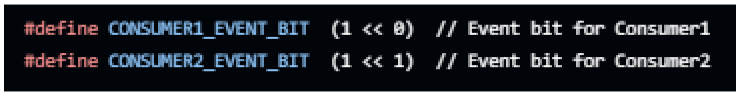
The consumer task is like the producer task but waits for the semaphore signals and processes them by printing the corresponding direction.

3. Repeat the application in Part B, that contains a producer and two consumers, and use

one event group instead of the semaphores.

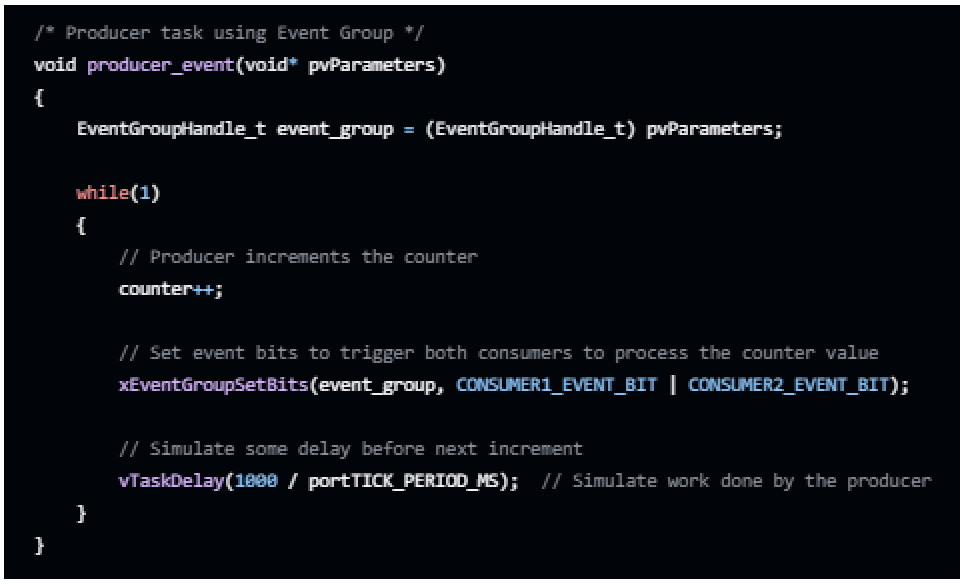
This problem consists of a producer task and two consumer tasks (Consumer1 and Consumer2). The producer task increments a shared counter and signals the consumer tasks to process the counter value using event bits. The consumer tasks wait for their respective event bits to be set, process the counter value, and then reset the event bits. We use an event group to manage synchronization between the producer and consumer tasks. Event groups allow tasks to set and wait for specific bits, which can represent different events or conditions.

We defined these event bits using bitwise shifts.



```
#define CONSUMER1_EVENT_BIT  (1 << 0)  // Event bit for Consumer1
#define CONSUMER2_EVENT_BIT  (1 << 1)  // Event bit for Consumer2
```

The producer task (producer_event) is responsible for incrementing a shared counter and signalling the consumer tasks to process the counter value. The producer increments the counter variable, and then it sets both the consumer 1 and 2 event bits in the event group using xEventGroupSetBits. This notifies both consumer tasks that new data is available. The producer then delays for 1 second to simulate work being done before the next increment.



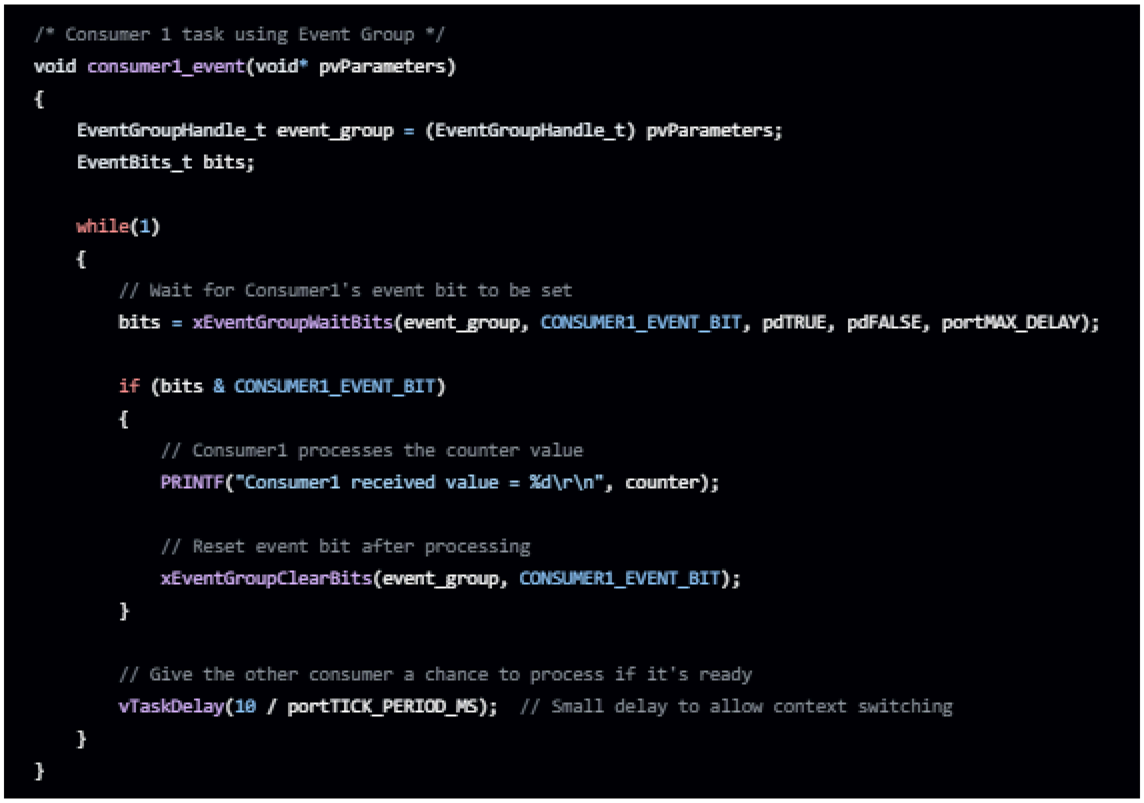
```
/* Producer task using Event Group */
void producer_event(void* pvParameters)
{
    EventGroupHandle_t event_group = (EventGroupHandle_t) pvParameters;

    while(1)
    {
        // Producer increments the counter
        counter++;

        // Set event bits to trigger both consumers to process the counter value
        xEventGroupSetBits(event_group, CONSUMER1_EVENT_BIT | CONSUMER2_EVENT_BIT);

        // Simulate some delay before next increment
        vTaskDelay(1000 / portTICK_PERIOD_MS);  // Simulate work done by the producer
    }
}
```

There are two consumer tasks: Consumer1 and Consumer2. Both tasks wait for their respective event bits to be set, process the counter value, and then reset the event bits. Consumer 1 waits for the event bit to be set and blocks until the bit is set.Once the bit is set, it prints the counter value and then resets the consumer 1 event bit to indicate that it has processed the event.



```
/* Consumer 1 task using Event Group */
void consumer1_event(void* pvParameters)
{
    EventGroupHandle_t event_group = (EventGroupHandle_t) pvParameters;
    EventBits_t bits;

    while(1)
    {
        // Wait for Consumer1's event bit to be set
        bits = xEventGroupWaitBits(event_group, CONSUMER1_EVENT_BIT, pdTRUE, pdFALSE, portMAX_DELAY);

        if (bits & CONSUMER1_EVENT_BIT)
        {
            // Consumer1 processes the counter value
            PRINTF("Consumer1 received value = %d\r\n", counter);

            // Reset event bit after processing
            xEventGroupClearBits(event_group, CONSUMER1_EVENT_BIT);
        }

        // Give the other consumer a chance to process if it's ready
        vTaskDelay(10 / portTICK_PERIOD_MS);  // Small delay to allow context switching
    }
}
```

Problem 5

At the end of the ISR, there are the macro portYIELD FROM ISR and the variable

xHigherPriorityTaskWoken. Search in FreeRTOS documentation and explain why they

are needed.

In FreeRTOS, the portYIELD_FROM_ISR macro and the xHigherPriorityTaskWoken variable are essential for ensuring efficient task scheduling and real-time responsiveness when working with Interrupt Service Routines (ISRs). The xHigherPriorityTaskWoken variable is used to track whether an ISR operation, such as setting an event bit or giving a semaphore, has unblocked a higher-priority task. It is initialized to pdFALSE at the start of the ISR and set to pdTRUE by FreeRTOS API functions (e.g., xEventGroupSetBitsFromISR) if a higher-priority task is unblocked. At the end of the ISR, the portYIELD_FROM_ISR macro is used to request a context switch if xHigherPriorityTaskWoken is pdTRUE, ensuring that the higher-priority task runs immediately. If no higher-priority task is unblocked, portYIELD_FROM_ISR does nothing, avoiding unnecessary context switches. This mechanism is crucial for maintaining real-time performance, as it allows higher-priority tasks to respond promptly to events signaled by the ISR while minimizing overhead.

Problem 6

Create a periodic timer with a 1 second period. The timer's callback function signals a semaphore to a task. The task should keep waiting for the semaphore, and once the signal arrives, it prints something on the console and repeats again.

For this problem, we use semaphores, timers, and tasks to create a periodic "Hello World" printing system. The program initializes a binary semaphore (timerSemaphore) and a periodic timer (periodicTimer) configured to expire every 1000 milliseconds (1 second). When the timer expires, its callback function (timerCallbackFunction2) signals the semaphore using xSemaphoreGive. The hello_task waits indefinitely for the semaphore using xSemaphoreTake, and when the semaphore is signalled, the task prints "Hello World" to the debug console. The timer is set to auto-reload (pdTRUE), ensuring it repeats periodically. The FreeRTOS scheduler manages the execution of the hello_task and the timer, allowing the task to synchronize with the timer's periodic events. This pattern showcases how semaphores can be used to synchronize tasks with timer-driven events in a real-time operating system, ensuring efficient and predictable task execution.

Problem 7

Explain the purpose of the variables "ptr" and "rc values" in the previous code, and how the channels' values are copied to "rc values".

The ptr variable is a pointer of type uint8_t* that is initialized to point to the address of the rc_values structure. This allows the program to treat the rc_values structure as a byte array, enabling byte-level access to its contents. The rc_values variable is an instance of the RC_Values structure, which stores the header and channel values (ch1 to ch8) received from a remote control (RC) system via UART. The purpose of ptr is to facilitate the reading of UART data into the rc_values structure. The program first reads a single byte from UART to check if it matches the expected header byte (0x20). If the header byte is correct, it reads the remaining bytes of the rc_values structure using UART_ReadBlocking, copying the data directly into the structure via the ptr pointer. This ensures that the channel values are correctly populated in the rc_values structure. Once the data is read, the program checks if the full header (0x4020) is valid and then prints the channel values. This approach allows efficient and structured handling of UART data, ensuring that the RC channel values are accurately captured and processed.