Lab 2: Introduction to FreeRTOS Group 10

COMPENG 4DS4: Embedded Systems

Professor: Mohamed Hassan

Due: Wednesday February 28th, 2024

Name: Cameron Beneteau

Email: beneteac@mcmaster.ca

Student Number: 400274092

Name: Shaqif Ahmed

Email: ahmes80@mcmaster.ca

Student Number: 400263303

Declaration of Contributions

Tasks	Contributor(s)
Experiment 1: FreeRTOS Hello World	Cameron Beneteau & Shaqif Ahmed
Experiment Setup: Part A	
Experiment Setup: Part B	
Problem 1	
Experiment 2: Inter-task Communication and Synchronization	
Experiment Setup: Part A - Queues	
Problem 2	
Experiment Setup: Part B - Semaphores	
Problem 3: Part 1	
Problem 3: Part 2	
Experiment Setup: Part C - Event Groups	
Problem 4: Part 1	
Problem 4: Part 2	
Problem 4: Part 3	
Experiment 3: Interrupts	
Problem 5	
Experiment 4: Timers	
Problem 6	
Experiment 5: Radio Controller and Receiver	
Problem 7	

GitHub Repository

https://github.com/COMPENG-4DS4-Winter2024/lab2-freertos-group-10

Overview

This lab introduces FreeRTOS, focusing on creating tasks, understanding inter-task communication, synchronization techniques, configuring interrupts within FreeRTOS, utilizing timers for event generation, and setting up radio controller communication. The practical application of these concepts was demonstrated using the RDDRONE-FMUK66 board, Segger J-Link EDU Mini debugger, and various other components, including telemetry modules and a radio controller with its receiver.

Experiment 1: FreeRTOS Hello World

The experiment began with importing the FreeRTOS "Hello World" project, creating multiple tasks of different priorities. A simple task was set up to print "Hello World" to the console, demonstrating the basic task creation and scheduling in FreeRTOS.

Experiment Setup: Part A

A task was created to print "Hello World" on the console, illustrating the simplicity of task creation and execution in FreeRTOS.

Experiment Setup: Part B

This setup introduced task argument passing by creating another task that prints "Hello 4DS", showcasing how tasks can receive inputs through xTaskCreate.

Problem 1

Developed a program with two tasks where the first task accepts a string input from the user and the second task prints this string periodically every 1 second. The first task deletes itself after input reception, demonstrating dynamic task management and deletion. The priority of the first task is 2 while it is 3 for the second task. Although the priority of the second task is higher than the first, it will not print anything until the string has a non-null value.

```c
char str[MAX_LEN];

void task_1(void *pvParameters)
{
    PRINTF("Input string\r\n");
    scanf("%s", str);
    PRINTF("Input received\r\n");
    vTaskDelete(NULL);
}

void task_2(void *pvParameters)
{
    while (1)
    {
        if (str != NULL)
        {
            PRINTF("%s\r\n", str);
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }

    vTaskDelete(NULL);
}
```

```
frdmk66f_lab_2_problem_1_freertos_hello JLink Debug [GDB SEGGER Interface Debugging]
Executed SetRestartOnClose=1
[MCUXpresso Semihosting Telnet console for 'frdmk66f_lab_2_problem_1_freertos_hello JLink Debug' started on port 59809 @ 127.0.0.1]

SEGGER J-Link GDB Server V7.94b - Terminal output channel

Input string
random
Input received
random
random
random
random
random
random
random
random
random
random
random
random
random
random
random
```

Experiment 2: Inter-task Communication and Synchronization

Experiment Setup: Part A - Queues

Implemented FreeRTOS queues to facilitate data transfer and synchronization between tasks, effectively illustrating queue-based communication.

Problem 2

Modified Problem 1 to utilize queues for inter-task communication in FreeRTOS. In this modified implementation, characters of the input string were enqueued and dequeued individually, rather than the entire sting at once. The size of the queue is one character or a char type which is 1 byte. The characters were sequentially sent and read using the xQueueSendToBack and xQueueReceive functions, which in a way act as queue semaphores.

```
void producer_queue(void *pvParameters)
{
    QueueHandle_t queue1 = (QueueHandle_t)pvParameters;
    BaseType_t status;
    int counter = 0;

    PRINTF("Input string\r\n");
    scanf("%s", str);
    strLen = strlen(str);
    PRINTF("Input received\r\n");

    while (1)
    {
        if (counter < strLen)
        {
            status = xQueueSendToBack(queue1, (void *)&str[counter], portMAX_DELAY);
            counter++;
            if (status != pdPASS)
            {
                PRINTF("Queue Send failed!.\r\n");
                while (1)
                    ;
            }
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }

    vTaskDelete(NULL);
}
```

```c
void consumer_queue(void *pvParameters)
{
    QueueHandle_t queue1 = (QueueHandle_t)pvParameters;
    BaseType_t status;
    int counter = 0;
    char c;

    while (1)
    {
        status = xQueueReceive(queue1, (void *)&c, portMAX_DELAY);
        receivedStr[counter] = c;
        counter++;
        if (status != pdPASS)
        {
            PRINTF("Queue Receive failed!.\r\n");
            while (1)
                ;
        }
        PRINTF("Received Value = %c\r\n", c);

        if (counter == strLen)
        {
            receivedStr[counter] = '\0';

            while (1)
            {
                PRINTF("%s\r\n", receivedStr);
                vTaskDelay(1000 / portTICK_PERIOD_MS);
            }
        }
    }
}
```

```
frdmk66f_lab_2_problem_2_freertos_hello JLink Debug [GDB SEGGER Interface Debugging]
Executed SetRestartOnClose=1
[MCUXpresso Semihosting Telnet console for 'frdmk66f_lab_2_problem_2_freertos_hello JLink Debug' started on port 61148 @ 127.0.0.1]

SEGGER J-Link GDB Server V7.94b - Terminal output channel
Input string
random
Input received
Received Value = r
Received Value = a
Received Value = n
Received Value = d
Received Value = o
Received Value = m
random
random
random
random
random
random
random
random
random
```

Experiment Setup: Part B - Semaphores

Introduced binary and counting semaphores for task synchronization, highlighting their significance in managing task execution order and resource access.

Problem 3

Discussed the feasibility of using a single counting semaphore for synchronizing a producer task with multiple consumer tasks and extended Problem 1's synchronization mechanism using semaphores.

Part 1

A single counting semaphore can be used for the preceding application. The counting semaphore would be initialized with a count of 0, with a maximum value of 2 for the two consumers. The producer, after incrementing the counter, would 'give' the semaphore twice, signalling that new data is available for consumption. Each consumer would 'take' the semaphore once before printing the value, ensuring synchronized access to the shared counter. However, during testing, it was observed that one consumer could potentially 'take' the semaphore twice consecutively before the other had a chance to 'take' it even once. To mitigate this issue, a delay of 100ms was introduced in both consumer tasks, providing a brief pause to allow for more equitable semaphore acquisition between the two consumers. This adjustment ensured that both consumers had the opportunity to access the shared resource in a timely manner.

```
SemaphoreHandle_t *semaphores = (SemaphoreHandle_t *)malloc(2 * sizeof(SemaphoreHandle_t));
semaphores[0] = xSemaphoreCreateCounting(2, 2); // consumer_sem
semaphores[1] = xSemaphoreCreateCounting(2, 0); // producer_sem
```

```
void producer_sem(void *pvParameters)
{
    SemaphoreHandle_t *semaphores = (SemaphoreHandle_t *)pvParameters;
    SemaphoreHandle_t consumer_semaphore = semaphores[0];
    SemaphoreHandle_t producer_semaphore = semaphores[1];
    BaseType_t status1, status2;

    while (1)
    {
        status1 = xSemaphoreTake(consumer_semaphore, portMAX_DELAY);
        status2 = xSemaphoreTake(consumer_semaphore, portMAX_DELAY);
        if (status1 != pdPASS || status2 != pdPASS)
        {
            PRINTF("Failed to acquire consumer_semaphore\r\n");
            while (1)
                ;
        }

        counter++;
        xSemaphoreGive(producer_semaphore);
        xSemaphoreGive(producer_semaphore);

        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

```c
void consumer1_sem(void *pvParameters)
{
    SemaphoreHandle_t *semaphores = (SemaphoreHandle_t *)pvParameters;
    SemaphoreHandle_t consumer_semaphore = semaphores[0];
    SemaphoreHandle_t producer_semaphore = semaphores[1];
    BaseType_t status;

    while (1)
    {
        xSemaphoreGive(consumer_semaphore);

        // Delay so semaphore does not take twice
        vTaskDelay(100 / portTICK_PERIOD_MS);

        status = xSemaphoreTake(producer_semaphore, portMAX_DELAY);
        if (status != pdPASS)
        {
            PRINTF("Failed to acquire producer_semaphore\r\n");
            while (1)
                ;
        }

        PRINTF("Consumer 1 Received Value = %d\r\n", counter);
    }
}
```

```c
void consumer2_sem(void *pvParameters)
{
    SemaphoreHandle_t *semaphores = (SemaphoreHandle_t *)pvParameters;
    SemaphoreHandle_t consumer_semaphore = semaphores[0];
    SemaphoreHandle_t producer_semaphore = semaphores[1];
    BaseType_t status;

    while (1)
    {
        xSemaphoreGive(consumer_semaphore);

        // Delay so semaphore does not take twice
        vTaskDelay(100 / portTICK_PERIOD_MS);

        status = xSemaphoreTake(producer_semaphore, portMAX_DELAY);
        if (status != pdPASS)
        {
            PRINTF("Failed to acquire producer_semaphore\r\n");
            while (1)
                ;
        }

        PRINTF("Consumer 2 Received Value = %d\r\n", counter);
    }
}
```

```
frdmk66f_lab_2_problem_3_part_1_freertos_hello JLink Debug [GDB SEGGER Interface Debugging]
Executed SetRestartOnClose=1
[MCUXpresso Semihosting Telnet console for 'frdmk66f_lab_2_problem_3_part_1_freertos_hello JLink Debug' started on port 64086 @ 127.0.0.1]

SEGGER J-Link GDB Server V7.94b - Terminal output channel
Consumer 1 Received Value = 1
Consumer 2 Received Value = 1
Consumer 1 Received Value = 2
Consumer 2 Received Value = 2
Consumer 1 Received Value = 3
Consumer 2 Received Value = 3
Consumer 1 Received Value = 4
Consumer 2 Received Value = 4
Consumer 1 Received Value = 5
Consumer 2 Received Value = 5
Consumer 1 Received Value = 6
Consumer 2 Received Value = 6
Consumer 1 Received Value = 7
Consumer 2 Received Value = 7
Consumer 1 Received Value = 8
Consumer 2 Received Value = 8
Consumer 1 Received Value = 9
Consumer 2 Received Value = 9
Consumer 1 Received Value = 10
Consumer 2 Received Value = 10
```

Part 2

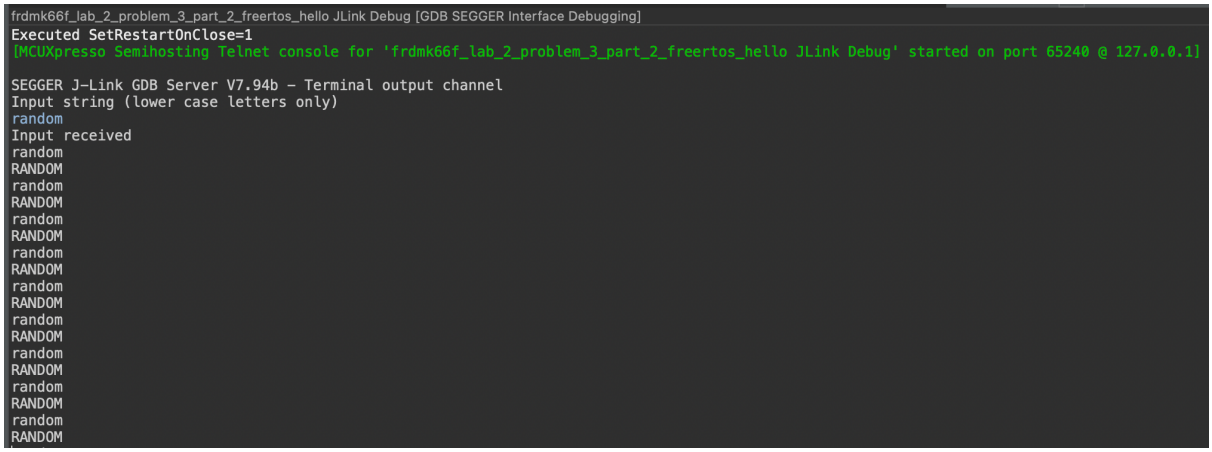
Modified the inter-task communication setup from Problem 1 by integrating an additional task that converts a string to uppercase and synchronize all three tasks using semaphores. The lowercase letters were converted to uppercase by subtracting 32 from their ASCII values. A delay of 1 second was introduced in task 2 and task, providing a brief pause to allow for more equitable semaphore acquisition between the two.

```c
SemaphoreHandle_t *semaphore = (SemaphoreHandle_t *)malloc(sizeof(SemaphoreHandle_t));
semaphore[0] = xSemaphoreCreateBinary();
```

```c
void task_1(void *pvParameters)
{
    SemaphoreHandle_t semaphore = ((SemaphoreHandle_t *)pvParameters)[0];

    PRINTF("Input string (lower case letters only)\r\n");
    scanf("%s", str);
    PRINTF("Input received\r\n");
    xSemaphoreGive(semaphore);

    vTaskDelete(NULL);
}
```

```c
void task_2(void *pvParameters)
{
    SemaphoreHandle_t semaphore = ((SemaphoreHandle_t *)pvParameters)[0];
    BaseType_t status;

    while (1)
    {
        status = xSemaphoreTake(semaphore, portMAX_DELAY);
        if (status != pdPASS)
        {
            PRINTF("Failed to acquire semaphore\r\n");
            while (1)
                ;
        }

        PRINTF("%s\r\n", str);
        xSemaphoreGive(semaphore);

        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }

    vTaskDelete(NULL);
}
```

```c
void task_3(void *pvParameters)
{
    SemaphoreHandle_t semaphore = ((SemaphoreHandle_t *)pvParameters)[0];
    BaseType_t status;

    while (1)
    {
        status = xSemaphoreTake(semaphore, portMAX_DELAY);
        if (status != pdPASS)
        {
            PRINTF("Failed to acquire semaphore\r\n");
            while (1)
                ;
        }

        for (int i = 0; i < strlen(str); i++)
        {
            strCap[i] = str[i] - 32;
        }

        PRINTF("%s\r\n", strCap);
        xSemaphoreGive(semaphore);

        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }

    vTaskDelete(NULL);
}
```



Experiment Setup: Part C - Event Groups

Explored event groups for synchronized task execution based on specific events, offering a flexible method for multi-event synchronization.

Problem 4

Evaluated different synchronization mechanisms, including semaphores and event groups, for their effectiveness in various scenarios, demonstrating advanced synchronization techniques in FreeRTOS.

Part 1

The priority of the producer must be lower than the priority of the consumer. The consumer must be waiting on the event to receive bits from the producer, meaning it must run first. If the producer has a similar or higher priority than the consumer, the producer will be sending events to nothing as the consumer will not be ready and waiting to pick them up.

xEventGroupSetBits does not block its task.

xEventGroupWaitBits does block its task.

```
#define LEFT_BIT (1 << 0)
#define RIGHT_BIT (1 << 1)
#define UP_BIT (1 << 2)
#define DOWN_BIT (1 << 3)
```

```c
void producer_event(void* pvParameters)
{
    EventGroupHandle_t event_group = (EventGroupHandle_t)pvParameters;
    BaseType_t status;
    char c;

    while(1)
    {
        scanf("%c", &c);

        switch(c)
        {
            case 'a':
                xEventGroupSetBits(event_group, LEFT_BIT);
                break;
            case 's':
                xEventGroupSetBits(event_group, DOWN_BIT);
                break;
            case 'd':
                xEventGroupSetBits(event_group, RIGHT_BIT);
                break;
            case 'w':
                xEventGroupSetBits(event_group, UP_BIT);
                break;
        }
    }
}
```

```c
void consumer_event(void* pvParameters)
{
    EventGroupHandle_t event_group = (EventGroupHandle_t)pvParameters;
    EventBits_t bits;

    while(1)
    {
        bits = xEventGroupWaitBits(event_group, LEFT_BIT | RIGHT_BIT | UP_BIT | DOWN_BIT, pdTRUE,
                                   pdFALSE, portMAX_DELAY);

        if((bits & LEFT_BIT) == LEFT_BIT)
        {
            PRINTF("Left\r\n");
        }

        if((bits & RIGHT_BIT) == RIGHT_BIT)
        {
            PRINTF("Right\r\n");
        }

        if((bits & UP_BIT) == UP_BIT)
        {
            PRINTF("Up\r\n");
        }

        if((bits & DOWN_BIT) == DOWN_BIT)
        {
            PRINTF("Down\r\n");
        }
    }
}
```

```
frdmk66f_lab_2_problem_4_part_1 JLink Debug [GDB SEGGER Interface Debugging]
Executed SetRestartOnClose=1
[MCUXpresso Semihosting Telnet console for 'frdmk66f_lab_2_problem_4_part_1 JLink Debug' started on port 51324 @ 127.0.0.1]

SEGGER J-Link GDB Server V7.94b - Terminal output channel
w
Up
d
Right
s
Down
a
Left
```

Part 2

Repeated the experiment's application by using semaphores instead of event groups. As global variables were not allowed for this exercise, four binary semaphores were used. Each consumer (left, right, up, down) interacts with their own corresponding semaphore. Note that only the screenshot of the left consumer is shown below.

```
#define LEFT_BIT (1 << 0)
#define RIGHT_BIT (1 << 1)
#define UP_BIT (1 << 2)
#define DOWN_BIT (1 << 3)
```

```
SemaphoreHandle_t *semaphores = (SemaphoreHandle_t *)malloc(4 * sizeof(SemaphoreHandle_t));
semaphores[0] = xSemaphoreCreateBinary();
semaphores[1] = xSemaphoreCreateBinary();
semaphores[2] = xSemaphoreCreateBinary();
semaphores[3] = xSemaphoreCreateBinary();
```

```c
void producer_event(void *pvParameters)
{

    SemaphoreHandle_t *semaphores = (SemaphoreHandle_t *)pvParameters;
    SemaphoreHandle_t l_semaphore = semaphores[0];
    SemaphoreHandle_t r_semaphore = semaphores[1];
    SemaphoreHandle_t u_semaphore = semaphores[2];
    SemaphoreHandle_t d_semaphore = semaphores[3];
    BaseType_t status;
    char c;

    while (1)
    {
        scanf("%c", &c);

        switch (c)
        {
        case 'a':
            xSemaphoreGive(l_semaphore);
            break;
        case 's':
            xSemaphoreGive(d_semaphore);
            break;
        case 'd':
            xSemaphoreGive(r_semaphore);
            break;
        case 'w':
            xSemaphoreGive(u_semaphore);
            break;
        }
    }
}
```

```c
void consumer_left(void *pvParameters)
{
    SemaphoreHandle_t *semaphores = (SemaphoreHandle_t *)pvParameters;
    SemaphoreHandle_t semaphore = semaphores[0];
    BaseType_t status;

    while (1)
    {
        status = xSemaphoreTake(semaphore, portMAX_DELAY);
        if (status != pdPASS)
        {
            PRINTF("Failed to acquire semaphore\r\n");
            while (1)
                ;
        }

        PRINTF("LEFT\r\n");
    }
}
```

```
frdmk66f_lab_2_experiment_c_freertos_hello JLink Debug [GDB SEGGER Interface Debugging]
Executed SetRestartOnClose=1
[MCUXpresso Semihosting Telnet console for 'frdmk66f_lab_2_experiment_c_freertos_hello JLink Debug' started on port 53351 @ 127.0.0.1]

SEGGER J-Link GDB Server V7.94b - Terminal output channel
w
UP
a
LEFT
s
DOWN
d
RIGHT
```

Part 3

Repeated the application in Experiment 2 Part B, that contains a producer and two consumers, and used one event group instead of the semaphores. Each consumer interacts with their own corresponding producer and consumer bit. They key to this is the fourth parameter in xEventGroupWaitBits of the producer function is set to true, meaning the event must wait for both consumer bits to be set (or given) before continuing.

```
#define C1_BIT (1 << 0)
#define C2_BIT (1 << 1)
#define P1_BIT (1 << 2)
#define P2_BIT (1 << 3)
```

```
void producer_sem(void *pvParameters)
{
    EventGroupHandle_t event_group = (EventGroupHandle_t)pvParameters;
    EventBits_t bits;

    while (1)
    {

        xEventGroupWaitBits(event_group, C1_BIT | C2_BIT, pdTRUE, pdTRUE, portMAX_DELAY);

        counter++;

        xEventGroupSetBits(event_group, P1_BIT);
        xEventGroupSetBits(event_group, P2_BIT);

        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

```c
void consumer1_sem(void *pvParameters)
{
    EventGroupHandle_t event_group = (EventGroupHandle_t)pvParameters;

    while (1)
    {
        xEventGroupSetBits(event_group, C1_BIT);
        xEventGroupWaitBits(event_group, P1_BIT, pdTRUE, pdTRUE, portMAX_DELAY);
        PRINTF("Consumer 1 Received Value = %d\r\n", counter);
    }
}

void consumer2_sem(void *pvParameters)
{
    EventGroupHandle_t event_group = (EventGroupHandle_t)pvParameters;

    while (1)
    {
        xEventGroupSetBits(event_group, C2_BIT);
        xEventGroupWaitBits(event_group, P2_BIT, pdTRUE, pdTRUE, portMAX_DELAY);
        PRINTF("Consumer 2 Received Value = %d\r\n", counter);
    }
}
```

```
frdmk66f_lab_2_problem_4_part_3_freertos_hello JLink Debug [GDB SEGGER Interface Debugging]
Executed SetRestartOnClose=1
[MCUXpresso Semihosting Telnet console for 'frdmk66f_lab_2_problem_4_part_3_freertos_hello JLink Debug' started on port 53595 @ 127.0.0.1]

SEGGER J-Link GDB Server V7.94b - Terminal output channel
Consumer 1 Received Value = 1
Consumer 2 Received Value = 1
Consumer 1 Received Value = 2
Consumer 2 Received Value = 2
Consumer 1 Received Value = 3
Consumer 2 Received Value = 3
Consumer 1 Received Value = 4
Consumer 2 Received Value = 4
Consumer 1 Received Value = 5
Consumer 2 Received Value = 5
Consumer 1 Received Value = 6
Consumer 2 Received Value = 6
Consumer 1 Received Value = 7
Consumer 2 Received Value = 7
Consumer 1 Received Value = 8
Consumer 2 Received Value = 8
Consumer 1 Received Value = 9
Consumer 2 Received Value = 9
```

Experiment 3: Interrupts

Configured UART interrupts within FreeRTOS, showing how to handle external interrupts and their integration into the FreeRTOS environment.

Problem 5

xHigherPriorityTaskWoken is a boolean flag passed into the xEventGroupSetBitsFromISR function. If the task that was interrupted by the ISR has a higher priority than the ISR task, this variable is set to true, if not it stays false. When this value is true, it means a context switch from the ISR to the interrupted higher priority task is required at the end, or exit, of the interrupt. The actual context switch is done, or requested, by the portYIELD_FROM_ISR function at the end of the interrupt when the xHigherPriorityTaskWoken variable is true.

```
#define TARGET_UART UART4

#define LEFT_BIT (1 << 0)
#define RIGHT_BIT (1 << 1)
#define UP_BIT (1 << 2)
#define DOWN_BIT (1 << 3)
```

```
void setupUART()
{
    uart_config_t config;

    UART_GetDefaultConfig(&config);
    config.baudRate_Bps = 57600;
    config.enableTx = true;
    config.enableRx = true;
    config.enableRxRTS = true;
    config.enableTxCTS = true;

    UART_Init(TARGET_UART, &config, CLOCK_GetFreq(kCLOCK_BusClk));

    /******** Enable Interrupts *********/
    UART_EnableInterrupts(TARGET_UART, kUART_RxDataRegFullInterruptEnable);

    NVIC_SetPriority(UART4_RX_TX_IRQn, 2);
    EnableIRQ(UART4_RX_TX_IRQn);
}
```

```
void UART4_RX_TX_IRQHandler()
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    UART_GetStatusFlags(TARGET_UART);
    char ch = UART_ReadByte(TARGET_UART);
    switch (ch)
    {
    case 'a':
        xEventGroupSetBitsFromISR(event_group_global_handler, LEFT_BIT, &xHigherPriorityTaskWoken);
        break;
    case 's':
        xEventGroupSetBitsFromISR(event_group_global_handler, DOWN_BIT, &xHigherPriorityTaskWoken);
        break;
    case 'd':
        xEventGroupSetBitsFromISR(event_group_global_handler, RIGHT_BIT, &xHigherPriorityTaskWoken);
        break;
    case 'w':
        xEventGroupSetBitsFromISR(event_group_global_handler, UP_BIT, &xHigherPriorityTaskWoken);
        break;
    }

    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

```c
void consumer_event(void* pvParameters)
{
    EventGroupHandle_t event_group = (EventGroupHandle_t)pvParameters;
    EventBits_t bits;

    while(1)
    {
        bits = xEventGroupWaitBits(event_group, LEFT_BIT | RIGHT_BIT | UP_BIT | DOWN_BIT, pdTRUE,
                                   pdFALSE, portMAX_DELAY);

        if((bits & LEFT_BIT) == LEFT_BIT)
        {
            PRINTF("Left\r\n");
        }

        if((bits & RIGHT_BIT) == RIGHT_BIT)
        {
            PRINTF("Right\r\n");
        }

        if((bits & UP_BIT) == UP_BIT)
        {
         PRINTF("Up\r\n");
        }

        if((bits & DOWN_BIT) == DOWN_BIT)
        {
            PRINTF("Down\r\n");
        }
    }
}
```

```
frdmk66f_lab_2_problem_5_freertos_hello JLink Debug [GDB SEGGER Interface Debugging]
Executed SetRestartOnClose=1
[MCUXpresso Semihosting Telnet console for 'frdmk66f_lab_2_problem_5_freertos_hello JLink Debug' started on port 60279 @ 127.0.0.1]

SEGGER J-Link GDB Server V7.94b - Terminal output channel
Up
Left
Down
Right
```

Experiment 4: Timers

Demonstrated the use of FreeRTOS timers for generating single-shot and periodic events, showcasing timers as a tool for time-driven task execution.

Problem 6

Implemented a semaphore-based synchronization mechanism triggered by a periodic timer, furthering the understanding of timers and semaphores in task synchronization. The hello task waits on the semaphore to be given before printing the string. This semaphore is given by the period tasks defined in the main function, which gives every second, allowing the string to be printed in the hello task.

```c
SemaphoreHandle_t semaphore;

void timerCallbackFunction(TimerHandle_t timer_handle)
{
    xSemaphoreGive(semaphore);
}

static void hello_task(void *pvParameters)
{
    while (1)
    {
        xSemaphoreTake(semaphore, portMAX_DELAY);
        PRINTF("Hello world.\r\n");
    }
}
```

```c
int main(void)
{
    BaseType_t status;

    /* Init board hardware. */
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();

    semaphore = xSemaphoreCreateBinary();

    status = xTaskCreate(hello_task, "Hello_task", 200, NULL, 2, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1)
            ;
    }

    TimerHandle_t timer_handle = xTimerCreate("Periodic timer", 1000 / portTICK_PERIOD_MS, pdTRUE, NULL, timerCallbackFunction);

    status = xTimerStart(timer_handle, 0);
    if (status != pdPASS)
    {
        PRINTF("Couldn't start the timer!.\r\n");
        while (1)
            ;
    }

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

```
frdmk66f_lab_2_problem_6_freertos_hello JLink Debug [GDB SEGGER Interface Debugging]
Executed SetRestartOnClose=1
[MCUXpresso Semihosting Telnet console for 'frdmk66f_lab_2_problem_6_freertos_hello JLink Debug' started on port 55789 @ 127.0.0.1]

SEGGER J-Link GDB Server V7.94b - Terminal output channel
Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
```

Experiment 5: Radio Controller and Receiver

Set up and configured a radio controller to communicate with the FMUK66 board via UART, interpreting the UART signals from the radio receiver to understand controller inputs.

Problem 7

rc_values is a struct defined by 9 uint16_t members, meaning rc_values has a size of 18 bytes (each uint16_t is 16 bits, or 2 bytes, so 18 bytes in total).

ptr is a pointer to the starting memory address of rc_values.

UART_ReadBlocking reads data from UART and copies it into input the memory address.

UART reads the first byte from the data and stores it in the first byte of rc_values, the memory address pointed to by ptr.

If this value is 0x20, something went wrong, or transmission wasn't yet received.

UART reads the next 17 bytes, size of rc_values minus 1 is 17, from the data and stores it in rc_values starting at the first index (zero-indexed), or 2nd byte.

Then using the indexing of the struct the program prints the values that were stores in rc_values.

For example, rc_values.ch3 points to the 7th byte in rc_values and has a length of 2 bytes, meaning its value is held in bytes 7 and 8 of rc_values. This is the value read by the program when printing.

For example, the streamed data is DC05, using little-endian format, would be stored in the 7th and 8th bytes of rc_values as DC and 05, respectively. Then when read by rc_values.ch3 it sees 0x05DC (which corresponds to 1500 in decimal.

```
typedef struct
{
    uint16_t header;
    uint16_t ch1;
    uint16_t ch2;
    uint16_t ch3;
    uint16_t ch4;
    uint16_t ch5;
    uint16_t ch6;
    uint16_t ch7;
    uint16_t ch8;
} RC_Values;
```

```c
int main(void)
{
    uart_config_t config;

    RC_Values rc_values;
    uint8_t *ptr = (uint8_t *)&rc_values;

    BOARD_InitBootPins();
    BOARD_InitBootClocks();

    UART_GetDefaultConfig(&config);
    config.baudRate_Bps = 115200;
    config.enableTx = false;
    config.enableRx = true;

    UART_Init(UART1, &config, CLOCK_GetFreq(kCLOCK_CoreSysClk));
```

```c
    while (1)
    {
        UART_ReadBlocking(UART1, ptr, 1);

        if (*ptr != 0x20)
            continue;

        UART_ReadBlocking(UART1, &ptr[1], sizeof(rc_values) - 1);

        PRINTF("%d\r\n", 0xDC05);
        PRINTF("%d\r\n", 0x05DC);

        if (rc_values.header == 0x4020)
        {
            printf("Channel 1 = %d\t", rc_values.ch1);
            printf("Channel 2 = %d\t", rc_values.ch2);
            printf("Channel 3 = %d\t", rc_values.ch3);
            printf("Channel 4 = %d\t", rc_values.ch4);
            printf("Channel 5 = %d\t", rc_values.ch5);
            printf("Channel 6 = %d\t", rc_values.ch6);
            printf("Channel 7 = %d\t", rc_values.ch7);
            printf("Channel 8 = %d\r\n", rc_values.ch8);
        }
    }
}
```

```
frdmk66f_lab_2_problem_7_hello_world JLink Debug [GDB SEGGER Interface Debugging]
Executed SetRestartOnClose=1
[MCUXpresso Semihosting Telnet console for 'frdmk66f_lab_2_problem_7_hello_world JLink Debug' started on port 59727 @ 127.0.0.1]

SEGGER J-Link GDB Server V7.94b – Terminal output channel
Channel 1 = 1500    Channel 2 = 1500    Channel 3 = 1071    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1500    Channel 2 = 1500    Channel 3 = 1071    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1500    Channel 2 = 1500    Channel 3 = 1071    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1500    Channel 2 = 1500    Channel 3 = 1071    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1500    Channel 2 = 1500    Channel 3 = 1071    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1500    Channel 2 = 1500    Channel 3 = 1071    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1500    Channel 2 = 1500    Channel 3 = 1071    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1500    Channel 2 = 1500    Channel 3 = 1071    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1500    Channel 2 = 1500    Channel 3 = 1071    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1500    Channel 2 = 1500    Channel 3 = 1071    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1500    Channel 2 = 1500    Channel 3 = 1071    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1500    Channel 2 = 1500    Channel 3 = 1071    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1500    Channel 2 = 1500    Channel 3 = 1071    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1500    Channel 2 = 1500    Channel 3 = 1071    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1500    Channel 2 = 1588    Channel 3 = 1151    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1500
Channel 1 = 1535    Channel 2 = 1797    Channel 3 = 1716    Channel 4 = 1000    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1535
Channel 1 = 2000    Channel 2 = 1735    Channel 3 = 1743    Channel 4 = 1000    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 2000
Channel 1 = 1816    Channel 2 = 1116    Channel 3 = 1715    Channel 4 = 1000    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1816
Channel 1 = 1209    Channel 2 = 1105    Channel 3 = 1578    Channel 4 = 1465    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1209
Channel 1 = 1095    Channel 2 = 1755    Channel 3 = 1219    Channel 4 = 1991    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1095
Channel 1 = 1772    Channel 2 = 2000    Channel 3 = 1000    Channel 4 = 1500    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1772
Channel 1 = 2000    Channel 2 = 1359    Channel 3 = 1353    Channel 4 = 1000    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 2000
Channel 1 = 1542    Channel 2 = 1000    Channel 3 = 1981    Channel 4 = 1091    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1542
Channel 1 = 1000    Channel 2 = 1453    Channel 3 = 1880    Channel 4 = 1743    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1000
Channel 1 = 1199    Channel 2 = 2000    Channel 3 = 1351    Channel 4 = 1960    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1199
Channel 1 = 2000    Channel 2 = 1750    Channel 3 = 1150    Channel 4 = 1300    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 2000
Channel 1 = 1888    Channel 2 = 1129    Channel 3 = 1630    Channel 4 = 1000    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1888
Channel 1 = 1230    Channel 2 = 1051    Channel 3 = 2000    Channel 4 = 1306    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1230
Channel 1 = 1016    Channel 2 = 1935    Channel 3 = 1519    Channel 4 = 1995    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1016
Channel 1 = 1820    Channel 2 = 2000    Channel 3 = 1000    Channel 4 = 1688    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1820
Channel 1 = 2000    Channel 2 = 1374    Channel 3 = 1402    Channel 4 = 1021    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 2000
Channel 1 = 1285    Channel 2 = 1000    Channel 3 = 2000    Channel 4 = 1280    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1285
Channel 1 = 1000    Channel 2 = 1579    Channel 3 = 1788    Channel 4 = 1793    Channel 5 = 1000    Channel 6 = 1000    Channel 7 = 2000    Channel 8 = 1000
```