LAB 2
Introduction to FreeRTOS

---

1 Lab Rules

- You have to stick to your Lab slot assigned to you on Mosaic.
- You have to use the Teams you created in Lab0 on github classroom.
- Prepare a demonstration for all the Lab experiments, and get ready to be asked in any part of the experiments.
- The demonstrations of Lab 0 will be held starting from **Feb 26th at the first hour of each lab slot**.
- All the activities and questions written in **blue** should be documented and answered in your lab report.
- Each team needs to submit one report for all the members, and the first page of the report should contain the team number and the names of its members.
- The submission should be through github classroom at 12pm on the day of your demo. Put the report in a PDF format. submission will be through github classroom.
- The first page (After the title page) of the report must include a **Declaration of Contributions**, where each team member writes his own individual tasks conducted towards the completion of the lab.
- You also need to submit all source files that you modify or add through out the lab.

General Note:

- Make sure to push all your code to the assignment repository from the lab computer before leaving the lab room because your saved work may be deleted after the lab slot.

2 Lab Rules

3 Lab Goals

- Build FreeRTOS and run it on FMUK66.
- Understand and reate tasks in FreeRTOS.
- Learn different techniques for inter-tasks communication and synchronization.
- Configure interrupts inside FreeRTOS.
- Use timers to create single shot and periodic events.

- Setup and use the radio controller.
- Interpret the UART signal returned from the radio receiver.

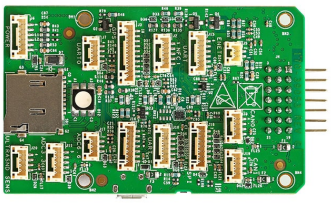
4 Github Classroom

You should accept the lab assignment through this link:
https://classroom.github.com/a/1Ia7lPTx
Again, you have to use the same Team you have created for your group in Lab0.

5 Lab Components

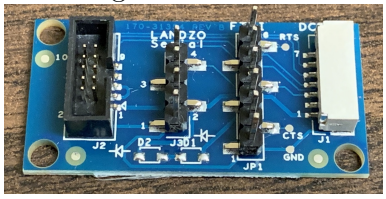
Prepare the following modules before starting the in-lab experiments.

1. 1x RDDRONE-FMUK66 board.



2. 1x Segger J-Link EDU Mini debugger.

3. 3x micro USB cables.
4. 1x Debug breakout board with the 7-wire cable.



5. 2x Telemetry modules.

6. 1x 6-wire cable.

7. 1x FlySky controller (radio transmitter).

8. 1x FlySky receiver (radio receiver).

6 Experiments

Experiment 1: FreeRTOS Hello World

In this experiment, we import the FreeRTOS "Hello World" project from the SDK to the base project for all the following experiments. Then we will create multiple tasks with different priorities.

Experiment Setup: Part A

In this part, we create a task that prints "Hello World" in the console.

1. In the MCUXpresso IDE, select "Import from SDK examples(s)" under the Quickstart Panel.
2. Select "frdmk6ff" and click Next → rtos_examples → check freertos_hello and select semihost → Finish
3. From Miscellaneous under Quickstart Panel, select Quick Settings→SDK Debug Console→Semihost Console.
4. In freertos_hello.c file, clear everything except the includes, and create an empty main function.
5. Make sure the project builds successfully.
6. Add the following code in the main function.

```
1  int main(void)
2  {
3      BaseType_t status;
4
5      /* Init board hardware. */
6      BOARD_InitBootPins();
7      BOARD_InitBootClocks();
8      BOARD_InitDebugConsole();
9
10     status = xTaskCreate(hello_task, "Hello_task", 200, NULL, 2, NULL);
11     if (status != pdPASS)
12     {
13         PRINTF("Task creation failed!.\r\n");
14         while (1);
15     }
16
17     vTaskStartScheduler();
18     while(1)
19     {}
20 }
```

In line 10, xTaskCreate creates a new task, which has the name "Hello_task"

- the second argument to the function. The first argument is a pointer to the task function, which we will define below. The third argument is the required stack size for the task execution, so it should be large enough for the local variables and the function calls

in the task. To pass values to the parameters of the task function, you can provide the address of those values through the fourth argument. If the task does not require any arguments, a NULL can be passed as in our case. The fifth argument is used to specify the priority of the task, where 0 is the lowest priority and 4 is the highest. Finally, the last argument is for returning a handler for the task, and NULL can be passed if the handler is not needed.

You do not have to memorize these inputs. Luckily, FreeRTOS documentation is more than enough to understand its APIs (functions). The documentation is coupled with the header files. For instance, in your workspace, you can find the documentation for xTaskCreate in "freertos → freertos_kernel → include → task.h" along with an example on how to use it. Alternatively, you can also use the FreeRTOS manual (FreeRTOS Reference Manual V10.0.0.pdf) uploaded on Avenue under Labs/Documentation. Refer to Section 2.6 for more details regarding vTaskCreate.

In line 17, vTaskStartScheduler begins the FreeRTOS system scheduler. The tasks will not execute until the scheduler

7. Add the function of "hello_task" above the main. Note that any task function should return void and accept void pointer as an input.

```
void hello_task(void *pvParameters)
{
    while(1)
    {
        PRINTF("Hello World\r\n");
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

This task will print "Hello World" every second. The delay in the task is created using vTaskDelay, which is different from the software delays we used to create in the previous labs. vTaskDelay removes the task from the running state to the waiting state, and leaves the CPU free to execute another task. Also, note that vTaskDelay accepts the delay in terms of number of ticks (FreeRTOS ticks). To convert the ticks to milliseconds, you need to divide by the constant portTICK_PERIOD_MS (Check the documentation in task.h or refer to Section 2.9).

8. Connect the J-link debugger to FMUK66.

9. Compile the code and download it to the board. The console must print "Hello World" every second.

Experiment Setup: Part B

Now, we build over the previous part by creating another task that accepts arguments passed from xTaskCreate.

1. Make the following edits to your existing code..

```
char* str = "4DS";
int main(void)
{
    BaseType_t status;
    /* Init board hardware. */
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();

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

    status = xTaskCreate(hello_task2, "Hello_task2", 200, (void*)str, 2, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1);
    }

    vTaskStartScheduler();
    for (;;);
}
```

The code creates another task, Hello_task2, but unlike the original task this one accepts string through the fourth argument of xTaskCreate.

2. Add the following function, hello_task2, above the main function.

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

The input from xTaskCreate reaches the function through its void pointer.

3. Run the code, and you should see both “Hello World” and “Hello 4DS” appear on the console.

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 **save the string globally**. After receiving the string the task should delete itself. On the other hand, the second task waits until the first the string is available, and then it keeps printing it every second. The priority of first task is 2 while it is 3 for the second task. Refer to Section 2.11 for details on ”vTaskDelete”.

Experiment 2: Inter-task Communication and Synchronization

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

Experiment Setup: Part A - Queues

In this part, we utilize FreeRTOS queues to transfer data between tasks and synchronize them.

1. Use the same project from the previous experiment or create a new "freertos_hello".
2. In freertos_hello.c file, clear everything except the includes and create an empty main function.
3. Include the header file "queue.h". This file also contains the documentation for queues' APIs (You could also refer to Chapter 3 in the manual).
4. Add the following code in the main function.

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

    QueueHandle_t queue1 = xQueueCreate(1, sizeof(int));
    if (queue1 == NULL)
    {
        PRINTF("Queue creation failed!.\r\n");
        while (1);
    }

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

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

This code creates a queue that fits only one item of integer data type's size. It further creates two tasks - producer task and consumer task - and it passes the queue handle to them.

5. Write the producer task's function above the main as follows.

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

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

This task increments an integer counter every second and sends its value to the queue using xQueueSendToBack. Note that if the queue is full while the task needs to send a new value, it will be moved to the waiting state till a place in the queue becomes available. Also, you can specify a maximum waiting time for a vacant location, or choose to wait forever as this task does by passing portMAX_DELAY to xQueueSendtoBack. Thus, the task is blocked till the queue receives a value.

6. Add, also, function for the consumer task as the following code.

```
void consumer_queue(void* pvParameters)
{
    QueueHandle_t queue1 = (QueueHandle_t)pvParameters;
    BaseType_t status;
    int received_counter;

    while(1)
    {
        status = xQueueReceive(queue1, (void *) &received_counter, portMAX_DELAY);
        if (status != pdPASS)
        {
            PRINTF("Queue Receive failed!.\r\n");
            while (1);
        }
        PRINTF("Received Value = %d\r\n", received_counter);
    }
}
```

This task waits for the counter value to be sent to the queue. Until it receives the counter value from the queue, it will be blocked. Once received, it will print the counter value to the console.

7. Run the code, and you should see the counter value is printed every second.

Problem 2

Repeat Problem 1 using queues.

Experiment Setup: Part B: Semaphores

This part introduces binary and counting semaphores and shows how to use them to synchronize the given tasks. A semaphore is essentially a signalling mechanism amongst various tasks. The two important semaphore operations are "Give" and "Take". The task responsible for sending a data/acknowledgement will "Give the semaphore" while the task that requires the data/acknowledgement will "Take the semaphore".

1. Use the same project from the previous part.
2. In the beginning of freertos_hello.c file, include the header file semphr.h. This file also contains the documentation for semaphores' APIs. (You could also refer to chapter 4 in the manual)
3. Modify the main function as the following.

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

    SemaphoreHandle_t* semaphores = (SemaphoreHandle_t*) malloc(2 * sizeof(
    SemaphoreHandle_t));
    semaphores[0] = xSemaphoreCreateBinary(); //Producer semaphore
    semaphores[1] = xSemaphoreCreateBinary(); //Consumer semaphore

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

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

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

This code creates an array of two binary semaphores to send them to the created tasks. The first semaphore is used to signal that a new value is produced, and similarly, the second one is to signal that the value is consumed.

4. Add the producer's function.

```
int counter = 0;

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

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

        counter++;
        xSemaphoreGive(producer_semaphore);

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

In the beginning of the code, a global counter is defined. Inside the producer task, first, it waits for the consumer to send its semaphore, signaling that the consumer is ready to receive the data. Until it receives the acknowledgement, the producer task will be blocked. Then the producer increments the counter and gives the producer semaphore to signal that the data is ready.

5. Add the consumer's function.

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

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

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

First, the consumer task gives the consumer semaphore which the producer is waiting for in the previous task. Then it waits for the producer semaphore. Until it receives the producer semaphore, the task will be blocked. Once received, it will print the counter.

6. Run the code, and you should notice the counter value is printed every second.

7. Now, We will modify the code to have two consumer tasks and implement a counting semaphore.

8. Modify the main function as the following.

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

    SemaphoreHandle_t* semaphores = (SemaphoreHandle_t*) malloc(3 * sizeof(
    SemaphoreHandle_t));
    semaphores[0] = xSemaphoreCreateBinary(); //Producer1_sem
    semaphores[1] = xSemaphoreCreateBinary(); //Producer2_sem
    semaphores[2] = xSemaphoreCreateCounting(2, 2); //consumer_sem

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

    status = xTaskCreate(consumer1_sem, "consumer", 200, (void*)semaphores, 2, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1);
    }

    status = xTaskCreate(consumer2_sem, "consumer", 200, (void*)semaphores, 2, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1);
    }

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

Here, the code creates three semaphores - two binary semaphores for the producer and a counting semaphore for the consumer. Further, it creates an extra consumer task.

9. Modify the producer's function as follows.

```
int counter = 0;

void producer_sem(void* pvParameters)
{
    SemaphoreHandle_t* semaphores = (SemaphoreHandle_t*)pvParameters;
    SemaphoreHandle_t producer1_semaphore = semaphores[0];
    SemaphoreHandle_t producer2_semaphore = semaphores[1];
    SemaphoreHandle_t consumer_semaphore = semaphores[2];
    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(producer1_semaphore);
        xSemaphoreGive(producer2_semaphore);

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

This task waits for the consumer semaphore twice since it is a counting semaphore and the task expects two signals from the two consumer tasks. And similarly, the task signals the two producer semaphores.

10. Add two functions for the consumer tasks as the following code.

```
void consumer1_sem(void* pvParameters)
{
    SemaphoreHandle_t* semaphores = (SemaphoreHandle_t*)pvParameters;
    SemaphoreHandle_t producer1_semaphore = semaphores[0];
    SemaphoreHandle_t consumer_semaphore = semaphores[2];
    BaseType_t status;

    while(1)
    {
        xSemaphoreGive(consumer_semaphore);
        status = xSemaphoreTake(producer1_semaphore, portMAX_DELAY);
        if (status != pdPASS)
        {
            PRINTF("Failed to acquire producer1_semaphore\r\n");
            while (1);
        }
```

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

void consumer2_sem(void* pvParameters)
{
    SemaphoreHandle_t* semaphores = (SemaphoreHandle_t*)pvParameters;
    SemaphoreHandle_t producer2_semaphore = semaphores[1];
    SemaphoreHandle_t consumer_semaphore = semaphores[2];
    BaseType_t status;

    while(1)
    {
        xSemaphoreGive(consumer_semaphore);
        status = xSemaphoreTake(producer2_semaphore, portMAX_DELAY);
        if (status != pdPASS)
        {
            PRINTF("Failed to acquire producer2_semaphore\r\n");
            while (1);
        }

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

This code is quite similar to the previous code of consumer_sem. The only difference is consumer1_sem waits for producer1_semaphore while consumer2_sem waits for producer2_semaphore.

11. Run the code, and you should notice the counter value is printed twice every second.

Problem 3

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

Experiment Setup: Part C - Event Groups

Event groups are popular technique for synchronization. They are similar to binary semaphores, but they are more flexible as they allow synchronization for multiple events using different bits within the group. The working mechanism of event groups will be clear after this experiment.

1. You can use the project of the previous part or create a new one.
2. In the beginning of freertos_hello.c file, include the header file "event_groups.h". This file also contains the documentation for the event groups' APIs. (You can also refer to Chapter 6 of the manual)

3. Write the following code in the main function.

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

    EventGroupHandle_t event_group = xEventGroupCreate();

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

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

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

Simply, this code creates an event group and sends its handle to the tasks.

4. For the producer task, its job will be reading a character from the user, and then it creates events based on the character as shown in the following code.

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

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;
        }

    }
}
```

The code waits for the character 'a', 's', 'd', or 'w', and then it issues left, down, right, or up event respectively. The events are issued by setting a chosen bit in the event group, for instance, the left event is associated with bit 0. Each event group contains 8 bits (from 0 to 7), and you can assign any event to any bit. Accordingly, it is always recommended to give meaningful names to the used bits, similar to the definitions in the beginning of the code, so the code becomes readable. In summary, the producer task generates four events according to the input character, and these events are generated by setting the bits in the event group. Later, the consumer task will act according to the received event.

5. For the consumer task, write the following function.

```
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");
        }
    }
}
```

This task waits for any event (bit) to arrive to the event group, and you should specify which bits the task should wait for in the second input of xEventGroupWaitBits. Also, the function provides the option to clear the bits after reading them or clear them manually using a separate API, and this is determined by passing true or false to the third input of the function. Additionally, the fourth input allows us to choose whether to wait for all the bits combined or just for any of them. Remember you always can return to the documentation in the header files or the pdf manual for more details.

6. Run and test the code. You should verify that the consumer task prints the correct message according to the input character.

Problem 4

1. Can the priority of the producer task be similar or higher than the priority of the consumer task? Explain your answer.
2. Repeat the experiment's application by using semaphores instead of event groups. Global variables are not allowed for this exercise.
3. Repeat the application in Part B, that contains a producer and two consumers, and use one event group instead of the semaphores.

Experiment 3: Interrupts

The goal of this experiment is to show how the MCU peripherals' interrupts work inside FreeRTOS. The experiment uses the telemetry modules to generate UART interrupts similar to Experiment 3 in Lab 1.

1. Connect one telemetry module to FMU board and the other one to the computer.
2. You should open a serial monitor such as PuTTY or RealTerm with baud rate 57600 by following the steps mentioned in lab 1. Both of these applications are already installed on the computers and you can use either of them.
3. In MCUXpresso, use the project of the previous experiment.
4. In freertos_hello.c, include the header of UART, "fsl_uart.h".
5. Make sure to configure the UART pins - UART4 TX, UART4 RX, UART4 CTS B, and UART4 RTS B in BOARD_InitBootPins().
6. For the clocks configuration, change the code inside BOARD_InitBootClocks() to the following code.

```
void BOARD_BootClockRUN(void)
{
    CLOCK_SetSimSafeDivs();

    CLOCK_SetInternalRefClkConfig(kMCG_IrclkEnable, kMCG_IrcFast, 2);
    CLOCK_CONFIG_SetFllExtRefDiv(0);
    CLOCK_SetExternalRefClkConfig(kMCG_OscselIrc);

    CLOCK_SetSimConfig(&simConfig_BOARD_BootClockRUN);
}
```

7. Write the following function to setup UART.

```
#define TARGET_UART                    UART4

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);
}
```

You may notice a new function, NVIC_SetPriority, compared to the setup function used in Lab 1. This function is used to give priorities to the interrupts, and since the MCU allows nested interrupts, it provides priority scheme for the interrupts to govern how interrupts interrupt the execution of others' ISR (i.e., how interrupts interrupt each other). The MCU has 16 levels of priority where 0 is the highest and 15 is the lowest (opposite to tasks' priorities). Previously in Lab 1, we didn't care about the interrupt's priority, and we left it to take the default value which is zero as it was the only interrupt we configured in the system. Now in this experiment, this interrupt runs along with the interrupts defined by FreeRTOS, and FreeRTOS does not allow any user-defined interrupt to have a priority higher than 2. FreeRTOS forces this rules to ensure running safely with the user's code, and for the sake of experimenting, try to use priority 1 and check how the code behaves.

8. In the main function, make the following edits.

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

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

    setupUART();

    event_group_global_handler = xEventGroupCreate();

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

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

You have to declare the variable event_group_global_handler globally before all the functions.

9. Finally, add the following code in the ISR of UART interrupt.

```
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);
}
```

This ISR is similar to the function of the producer function, but it gets the input character from the UART. Also, note that the function used to set the bits of the event group is different from the function used in the task. The reason for this difference is that FreeRTOS provides functions, which have the suffix "FromISR", to be called from ISRs, and these functions are not blocking and require minimal execution time. The execution time of ISRs is crucial and it should be as small as possible because the interrupts can interrupt any running task so their execution times are considered as overhead to the running task.

10. Run and test the code. You should verify that the consumer task prints the correct message according to the input character from the Linux terminal (note the terminal does not show the pressed character and you do not need to press Enter after each one).

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.

Experiment 4: Timers

FreeRTOS timers are used to generate a single shot or a periodic event after a specified time. In this experiment, we will use timers to generate simple events that print information on the console.

1. Create an empty "freertos_hello" project.
2. Include "timers.h" in the beginning of freertos_hello.c.
3. In the main function write the following code.

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

    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("Single Shot timer",
                                 5000 / portTICK_PERIOD_MS,
                                 pdFALSE,
```

```
                                NULL,
                                timerCallbackFunction);

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

    TimerHandle_t timer_handle2 = xTimerCreate("Periodic timer",
                                1000 / portTICK_PERIOD_MS,
                                pdTRUE,
                                NULL,
                                timerCallbackFunction2);

    status = xTimerStart(timer_handle2, 0);
    if(status != pdPASS)
    {
        PRINTF("Couldn't start the timer!.\r\n");
        while (1);
    }

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

This code creates Hello_task which prints "Hello World" every second. Additionally, it creates two timers - one is a single-shot(Observe the third argument of xTimerCreate) it finishes after 5 seconds, while the other timer runs periodically (Again observe the third argument of xTimerCreate) every second. The timers do not start counting automatically. xTimerStart should be called explicitly. Once the timer expires, the callback function is executed. For example, when the single-shot timer expires, the scheduler will call "timerCallbackFunction". Similarly, for the periodic timer, the scheduler will call timerCallbackFunction2.

4. Write "hello_task" function.

5. Add the following code for the callback functions.

```
void timerCallbackFunction(TimerHandle_t timer_handle)
{
    PRINTF("Hello from the single shot timer callback.\r\n");
}

void timerCallbackFunction2(TimerHandle_t timer_handle)
{
    static int counter = 0;

    PRINTF("Hello from the periodic timer callback. Counter = %d\r\n", counter);
    counter++;
    if(counter >= 10)
        xTimerStop(timer_handle, 0);
}
```

The callback function of the periodic timer counts the number of calls, and it stops the timer after ten calls.

6. Run and test the code.

Problem 6

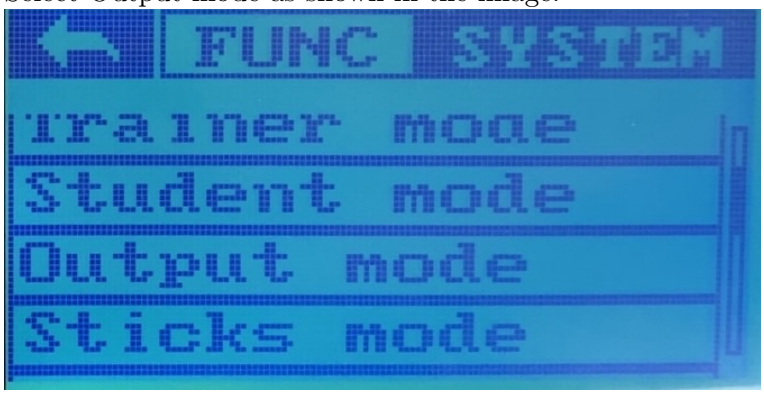
Create a periodic timer with a 1 second period. The timer's callback function signals a semaphore to a task. The task should keep waiting for the semaphore, and once the signal arrives, it prints something on the console and repeats again.

Experiment 5: Radio Controller and Receiver

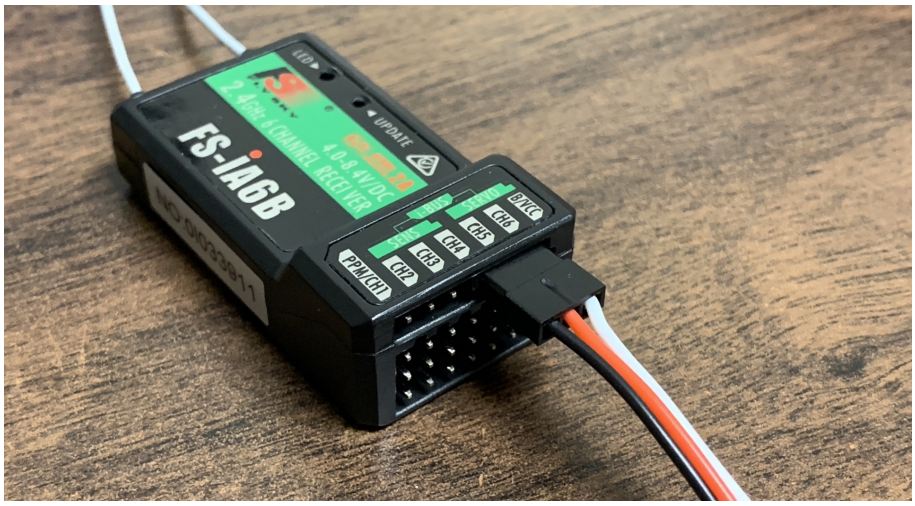
In this experiment, we aim to configure the radio controller to be able to receive its radio signal and read its corresponding UART signal.

Experiment Setup

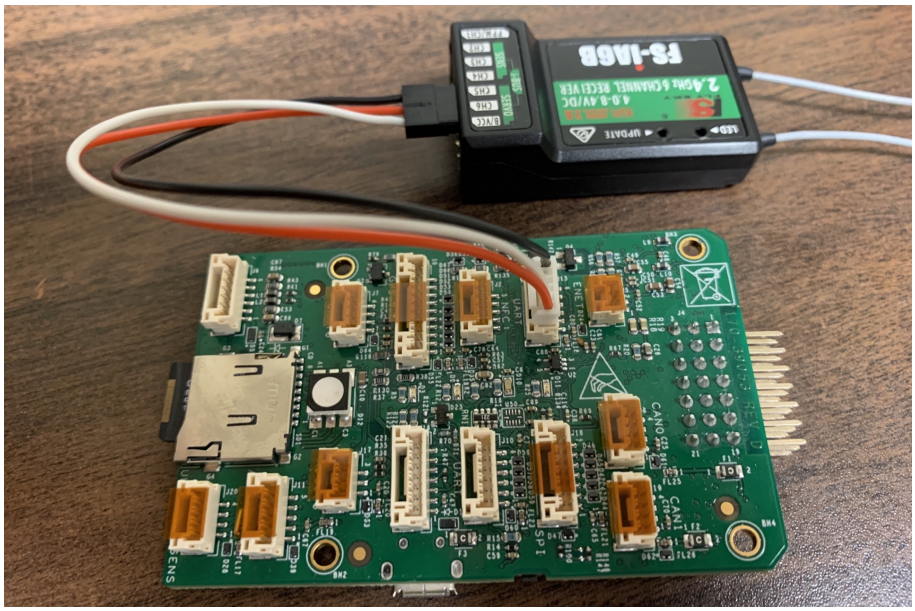
1. Turn on the controller by pressing both power buttons (the buttons on the down right and down left of the controller) in the same time for a couple of seconds. Note the controller requires 4 AA batteries to work.
2. On the controller's screen, press on the lock icon for 2 seconds till the setup icon appears.
3. Press on the setup icon and choose "SYS" from the top bar.
4. Select Output mode as shown in the image.



5. Choose PWM From Output and i-Bus from Serial.
6. Confirm that the 3-wire cable is connected to the receiver as shown in the image below.



7. Make sure that the other end of the cable is connected to J15 connector on the FMU board as shown in the image below.



8. Power on the FMU board and make sure the receiver's LED blinks.

9. Power on the controller as well, and it should connect to the receiver automatically. If the connection is established successfully, the receiver's LED will stop blinking and the controller will show a battery level for the receiver (the battery indicator will not be accurate as the receiver is not connected directly to a battey). If its not connected, make sure the number on the car matches the number printed on the controller.

10. Connect J-link to the FMU board as usual and move to MCUXpresso.
11. Import from the SDK demo_apps→hello_world, and select Semihost for debugging.
12. Select the project from the Project Explorer, and From Miscellaneous under Quickstart Panel, select Quick Settings→SDK Debug Console→Semihost Console.
13. In hello_world.c file, include "fsl_uart.h".
14. Configure the pin PTC3, which is connected to RC_INPUT, to work as UART1_RX.
15. Configure the clocks inside the function "BOARD_InitBootClocks."
16. Add the following code in the main function.

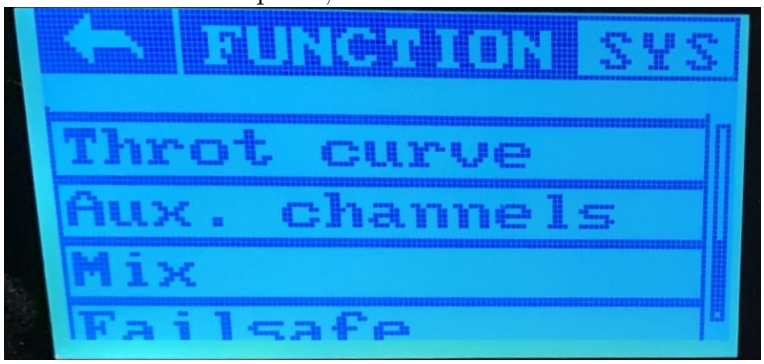
```
typedef struct {
    uint16_t header;
    uint16_t ch1;
    uint16_t ch2;
    uint16_t ch3;
    uint16_t ch4;
    uint16_t ch5;
    uint16_t ch6;
    uint16_t ch7;
    uint16_t ch8;
} RC_Values;

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));

    while (1)
    {
        UART_ReadBlocking(UART1, ptr, 1);
        if(*ptr != 0x20)
            continue;
        UART_ReadBlocking(UART1, &ptr[1], sizeof(rc_values) - 1);
        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);

        }
    }
}
```

The radio receiver sends a stream of data to the FMU via UART, and for that reason the code configures UART1 in the beginning of the main function. The data stream starts by 0x4020 followed by the values for 8 channels, starting from channel 1 to 8. Each channel value is represented by 2 bytes using little-endian formate (least significant byte arrives before the most significant), and the values range from 1000 to 2000.
These channels are mapped to the controller switches/joysticks and their values show the position of the buttons. For instance, let us assume that channel 1 is mapped to the horizontal position of the right joystick, and the joystick is kept in the middle then the channel's value will be 1500. The channels from 1 to 4 are mapped to the two joysticks by default, and the remaining from 5 to 8 are called auxiliary channels and can be assigned to any switch.

17. To change the auxiliary channels assignment, navigate to the setup menu of the controller.

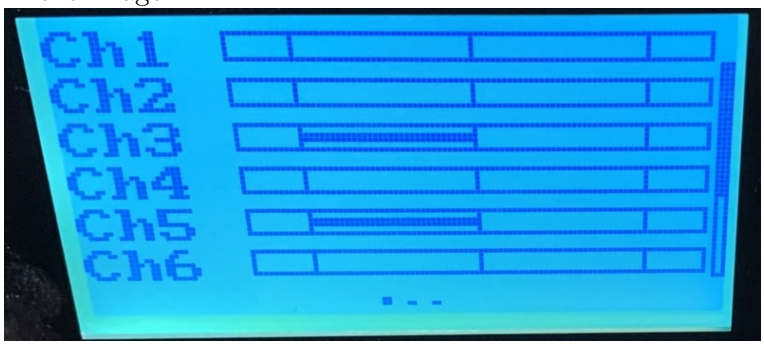
18. Under the function panel, choose "Aux. channels" as shown in the image.



19. Choose any assignment for the channels. For example to assign channel 5 to one of

the two push-buttons behind the controller, you have to tap on the left icon and select "KEY", and choose key1 or key2 to select the left or the right button. Check the image of assigning channel 5 to key1.

20. To check the channels' values. swipe from the left to the right on the home screen. Change the switches' positions and notice the change in the channels' values as shown in the image.



21. After choosing the preferable assignment, build the code and download it to the FMU.

22. Notice the change of the values printed to console according to the change in the position of the switches and joysticks.

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".