# CE 4DS4 - LAB 02: Introduction to FreeRTOS

Professor: Dr. Mohamed Hassan

Section L04

Khaled Hassan, hassak9, 400203796
Nikola Petrevski, petrevsn, 400198379

Due: Thursday, February 29, 2024

As a future member of the engineering profession, the student is responsible for performing the required work in an honest manner, without plagiarism and cheating. Submitting this work with my name and student number is a statement and understanding that this work is my own and adheres to the Academic Integrity Policy of McMaster University and the Code of Conduct of the Professional Engineers of Ontario.

Submitted by: Nikola Petrevski (petrevsn, 400198379) & Khaled Hassan (hassak9, 400203796).

## Declaration of Contributions

We aimed to split the lab's workload as evenly as possible. We worked together to perform the experiments, solve the problems and document our results.

## Problem 1

For the first task, we prompted the user to enter a string, and then stored the string as a global variable. We used a pointer to a char to store the string due to its variable possible size as seen below. We then used vTaskDelete to delete the task after completion.

```
22      char* str;
23
24      void task_1(void *pvParameters)
25      {
26              PRINTF("Please Enter a String: ");
27              scanf("%s", &str);
28              vTaskDelete(NULL);
29      }
```

To ensure that the second task only begins printing after the string has been received, the second task continuously checks if the first index of the string is empty. If it is not, it will begin printing whatever the str variable is pointing to every second using a delay:

```
31      void task_2(void *pvParameters)
32      {
33
34              //if the string is not null
35                      //do the while loop
36              while(1)
37              {
38                      if(&str[0]){
39                              PRINTF("Your string is: %s\n",  &str);
40                      }
41                      vTaskDelay(1000 / portTICK_PERIOD_MS);
42              }
43      }
```

## Problem 2

We first prompt the user to enter a string, and then create the queue "queue1" by using xQueueCreate() to create a queue of 1 element of size of the char data type. We do this because we will be using the queue to pass a single character of the string at a time. This makes handling inputted strings of different sizes more straightforward.

In the producer task, the xQueueSendToBack function passes each element of the string into the queue one at a time. While the queue is full, this function will wait for the consumer task to remove the element from the queue before putting the next one in. After all of the elements in the string have been passed into the queue (when a "\0" is read in the first index), the task deletes itself as it did in the previous problem.

For the consumer task, the xQueueReceive() function waits for the queue to be updated before continuing the execution of the task. In this way, it allows time for the producer task to pass the next element of the string into the queue before using it to print.

```
49    void consumer_queue(void* pvParameters)
50    {
51            QueueHandle_t queue1 = (QueueHandle_t)pvParameters;
52            BaseType_t status;
53            char received_char;
54            while(1)
55            {
56                    status = xQueueReceive(queue1, &received_char, portMAX_DELAY);
57                    if (status != pdPASS)
58                    {
59                            PRINTF("Queue Receive failed!.\r\n");
60                            while (1);
61                    }
62                    PRINTF("%c", received_char);
63            }
64    }
```

**Problem 3**

1. **Can we use a single counting producer semaphore for the previous application(experiment 2B)? Explain your answer.**

No, we cannot use a single counting producer semaphore for the previous application due to the nature of how the consumer semaphores receive their signal that they are ready to print. A producer semaphore would need to take at least two semaphores (given from the consumers) in order for it to know that the counter can be incremented. However, each consumer semaphore needs to take a semaphore before they can print the value of the counter as well. Therefore, one of the consumer semaphores could, in theory, take the semaphore that the other consumer task gave. If this happens, the producer will never receive both semaphores that it needs to increment the counter since one of the semaphores is being taken by a consumer task. In short, there is no way to know for sure which semaphore will be taken by the producer and the consumer at any given time. This is, in fact, likely why the experiment declared three independent semaphore types. The first two were labeled producer1 and producer2 to ensure that they can only be given to consumer1 and consumer2, respectively. Likewise, the consumer tasks gave a third semaphore instance which could only be received by the producer. As soon as a single counting semaphore is used, this vital differentiation vanishes, and the goal of the program cannot be achieved.

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.**

In order to capitalize the string that the user entered, a new function called capitalize() was created, which simply iterates through each index of the string and uses the toupper() function as shown below:

```
57      void capitalize(char * str)
58      {
59              for (int i = 0; str[i] != '\0'; i++){
60                      str[i] = toupper(str[i]);
61              }
62      }
```

The new task which uses this capitalization function to print the string in uppercase works in a similar way to the existing task which prints in lowercase. First, the task takes a binary semaphore, waits for the first index of the string to be non-empty, uses the strcopy() function to create a copy of the string, and then capitalizes it using the previously-defined capitalization function. After this, it prints in uppercase and then gives the semaphore back.

```
65      void task_3(void *pvParameters)
66      {
67              //if the string is not null
68              //do the while loop
69              while(1)
70              {
71                      SemaphoreHandle_t sem = *((SemaphoreHandle_t*)pvParameters);
72                      status_t status = xSemaphoreTake(sem, portMAX_DELAY);
73
74                      if(&str1[0] && status == pdPASS){
75                              strcpy(str2, str1);
76                              capitalize(str2);
77
78                              PRINTF("Your string in uppercase is: %s\n",  str2);
79                              xSemaphoreGive(sem);
80                      }
81
82                      vTaskDelay(1000 / portTICK_PERIOD_MS);
83              }
84      }
```

For this application, only a single binary semaphore was used. This is sufficient due to the delays that are added into the end of each of the two printing tasks (see line 82 in the figure above). This delay allows enough time for the uppercase and lowercase tasks to give each other the semaphore without taking its own semaphore back.

## Problem 4

1. **Can the priority of the producer task be similar or higher than the priority of the consumer task? Explain your answer.**

Yes, the producer task can be of similar or higher priority than a consumer task, depending on the application and the nature of the relationship between the data being produced by the producer and consumed by the consumer task(s). In cases where data produced by the producer must be processed and acted upon immediately, it would make sense for the producer to have a higher priority than the consumer. An example of this would be where an airbag collision sensor is the producer and the consumer is the actuator/airbag inflator. Another instance of this would be if the producer is performing a set of highly complex, time consuming operations. A change in input would mean that a producer of higher priority would interrupt a consumer of lower priority to save time. In our application in this lab, however, that was not the case. Since the scanf() operation is blocking and we required the consumers to operate periodically (our producer also only executes once), it did not make sense for us to give it a higher priority.

2. The code for this problem can be found under the "problem_4/source/freertos_hello.c" directory.

Instead of using event groups, we used 4 binary semaphores to repeat the experiment's application of checking for "w", "a", "s" and "d" key entries and printing "Up", "Left", "Down" and "Right" to the Terminal respectively. After initializing the necessary pins, clocks and debug console, we initialized an array of 4 binary semaphores, one for each key. That array is then shared between the producer and consumer tasks by passing a pointer to this array as a parameter.

```
SemaphoreHandle_t* semaphores = (SemaphoreHandle_t*) malloc(4 * sizeof(SemaphoreHandle_t));
semaphores[0] = xSemaphoreCreateBinary(); // up
semaphores[1] = xSemaphoreCreateBinary(); // down
semaphores[2] = xSemaphoreCreateBinary(); // left
semaphores[3] = xSemaphoreCreateBinary(); // right

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

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

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

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

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

We decided to split up the tasks as such: the producer task takes the input from the user terminal via scanf. A case-switch statement then gives the appropriate semaphore. Meanwhile, 4 consumer tasks of equal priority each wait to take its respective binary semaphore. As soon as it is able to do so, the consumer task will print the respective directional statement to the terminal.

```c
void producer_event(void* pvParameters)
{
    SemaphoreHandle_t* semaphores = (SemaphoreHandle_t*)pvParameters;
    SemaphoreHandle_t up_sem = semaphores[0];
    SemaphoreHandle_t down_sem = semaphores[1];
    SemaphoreHandle_t left_sem = semaphores[2];
    SemaphoreHandle_t right_sem = semaphores[3];

    BaseType_t status;
    char c;

    while(1)
    {
        scanf("%c", &c);
        switch(c)
        {
        case 'a': // give xSemaphoreGive(producer1_semaphore)
            xSemaphoreGive(left_sem);
            break;

        case 's':
            xSemaphoreGive(down_sem);
            break;

        case 'd':
            xSemaphoreGive(right_sem);
            break;

        case 'w':
            xSemaphoreGive(up_sem);
            break;
        }
    }
}
```

```c
void up_event(void* pvParameters)
{
    SemaphoreHandle_t* semaphores = (SemaphoreHandle_t*)pvParameters;
    SemaphoreHandle_t up_sem = semaphores[0];

    while(1)
    {

        if(xSemaphoreTake(up_sem, portMAX_DELAY) == pdPASS)
        {
            PRINTF("Up\r\n");
        }
    }
}
```

3. The code for this problem can be found under the “problem_5/source/freertos_hello.c” directory.

In order to repeat the functionality of binary semaphores using an event group, we defined an event group and shared it between the producer and 2 consumers by passing it to each task as a parameter after initializing the necessary pins, clocks and debug console.

```
75    EventGroupHandle_t event_group = xEventGroupCreate();
76
77    status = xTaskCreate(producer, "producer", 200, (void*)event_group, 2, NULL);
78    if (status != pdPASS)
79    {
80        PRINTF("Task creation failed!.\r\n");
81        while (1);
82    }
83    status = xTaskCreate(consumer1, "consumer", 200, (void*)event_group, 3, NULL);
84    if (status != pdPASS)
85    {
86        PRINTF("Task creation failed!.\r\n");
87        while (1);
88    }
89    status = xTaskCreate(consumer2, "consumer", 200, (void*)event_group, 3, NULL);
90    if (status != pdPASS)
91    {
92        PRINTF("Task creation failed!.\r\n");
93        while (1);
94    }
```

A global counter was initialized to 0, as was done in experiment 2B. The defined producer increments the counter and sets the relevant bits within the event group so that the consumer task may proceed. When a consumer's bit is set, the value of the counter is printed before waiting for 1 second. When checking the value of the bits in the event group, the xClearOnExit attribute of the xEventGroupWaitBits method is set to true. This clears the bits that were set when xEventGroupWaitBits returns such that the producer can then set them again as appropriate. Finally, a bit mask is applied to each consumer's bit within the event group such that we're only checking if the bit relevant to this event has been set by the consumer. Consumer 2 was defined similar to consumer 1.

```
20  void producer(void* pvParameters)
21  {
22      EventGroupHandle_t event_group = (EventGroupHandle_t)pvParameters;
23
24      while(1)
25      {
26          counter++;
27          xEventGroupSetBits(event_group, con1_bit);
28          xEventGroupSetBits(event_group, con2_bit);
29          vTaskDelay(1000 / portTICK_PERIOD_MS);
30      }
31  }
32
33
34  void consumer1(void* pvParameters)
35  {
36      EventGroupHandle_t event_group = (EventGroupHandle_t)pvParameters;
37      EventBits_t bits;
38
39      while(1)
40      {
41          bits = xEventGroupWaitBits(event_group, con1_bit | con2_bit, pdTRUE, pdFALSE,
42          portMAX_DELAY);
43
44          if((bits & con1_bit) == con1_bit){
45              PRINTF("Value Received by Consumer 1 = %d\r\n", counter);
46              vTaskDelay(1000 / portTICK_PERIOD_MS);
47          }
48      }
49  }
```

## 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 our implementation of the ISR, the last line of the IRQ handler, we pass the xHigherPriorityTaskWoken variable to the portYIELD_FROM_ISR macro function. xHigherPriorityTaskWoken is initially set to pdFALSE, but it might be changed to pdTRUE by any op the 4 xEventGroupSetBitsFromISR function calls within the IRQ handler. When xHigherPriorityTaskWoken is set to true, then a task that has a higher priority than the current task was unblocked, and a context switch would occur such that the ISR returns to that newly unblocked task. If xHigherPriorityTaskWoken remained false, the ISR would return to the currently executing task instead.

## Problem 6

The code for this problem can be found under the “problem_6/source/freertos_hello.c” directory. To implement the problem as required, we first initialize a global binary semaphore to share among tasks. 1 task and 1 timer are created. The task waits to take the binary semaphore, before printing a message and the value of a counter. As for the timer, it is a periodic timer with a 1-second period. That means that every second, the timer call back function is called to give the semaphore, which allows the task to execute.

```
71    semaphore = xSemaphoreCreateBinary();
72
73    status = xTaskCreate(periodic_task, "Hello_task", 200, (void*) semaphore, 2, NULL);
74    if (status != pdPASS)
75    {
76        PRINTF("Task creation failed!.\r\n");
77        while (1);
78    }
79
80    TimerHandle_t timer_handle = xTimerCreate("Periodic timer",
81                                              1000 / portTICK_PERIOD_MS,
82                                              pdTRUE,
83                                              NULL,
84                                              timerCallbackFunction2);
85
86    status = xTimerStart(timer_handle, 0);
```

```
42    SemaphoreHandle_t* semaphore;
43
44
45    void timerCallbackFunction2(TimerHandle_t timer_handle)
46    {
47        xSemaphoreGive(semaphore);
48    }
49
50    static void periodic_task(void *pvParameters)
51    {
52        int counter = 0;
53        while(1)
54        {
55            if(xSemaphoreTake(semaphore, portMAX_DELAY) == pdPASS){
56                PRINTF("Hello world, %d.\r\n", counter);
57                counter ++;
58            }
59        }
60    }
```

## 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 rc_values variable is an instance of the RC_Values structure, which contains a header as well as 8 16-bit integer attributes corresponding to 8 channels that map the values of the 2 joysticks (channels 1 - 4) and other switches on the remote controller (auxiliary channels 5 - 8). The ptr variable is a pointer to a type-casted instance of rc_values. This type-casting to unsigned 8-bit integers is done to allow us to read the values of the channels (which are transmitted from the RC as a byte stream in little endian format) as bytes.