COMPENG 4DS4 Lab 2 Report

Aaron Pinto	Raeed Hassan	Jingming Liu	Jeffrey Guo
pintoa9	hassam41	liuj171	guoj69

March 3, 2023

Declaration of Contributions

Problem	Contributions
1	Jeffrey, Jingming
2	Jingming
3	Aaron, Jingming
4	Jingming, Raeed
5	Jeffrey, Raeed
6	Aaron, Raeed
7	Raeed

Problem 1

We write a program that generates two tasks, with the first task getting an input string from the user using `scanf` and then notifying the printing task with `xTaskNotifyGive`. The task will delete itself afterwards. The second task will wait for the notification by blocking on `ulTaskNotifyTake`. The task will then print the input string and wait one second. The main function is shown below in Listing 1. The two tasks are shown in Listings 2 and 3.

Listing 1: Problem 1 main

```c
QueueHandle_t queue1;

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

    usr_str = malloc(sizeof(char) * 100);

    status = xTaskCreate(hello_task, "Hello_task", 200, (void*) usr_str, 2, &t_handle_1);
    if (status != pdPASS) {
        PRINTF("Task creation failed!.\r\n");
        while (1)
            ;
    }

    status = xTaskCreate(hello_task2, "Hello_task2", 200, (void*) usr_str, 3, &t_handle_2);
    if (status != pdPASS) {
        PRINTF("Task creation failed!.\r\n");
        while (1)
            ;
    }

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

Listing 2: Problem 1 Input Task

```c
void hello_task(void *pvParameters) {
    while (1) {
        PRINTF("Enter a string input\n");
        scanf("%s", usr_str);

        vTaskDelay(1000 / portTICK_PERIOD_MS);
        xTaskNotifyGive(t_handle_2);
        vTaskDelete(NULL);
    }
}
```

Listing 3: Problem 1 Print Task

```c
void hello_task2(void *pvParameters) {
    while (1) {
        ulTaskNotifyTake(pdTRUE, 100);

        PRINTF("Hello %s.\r\n", usr_str);

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

Problem 2

When repeating Problem 1 with queues, we have a producer task handling the user input and a consumer task printing the string. The priority of the producer is 2 and the consumer is 3. The queue declared is `queue1`. In the producer task, the user input `usr_str` will be put in the queue by `xQueueSend`. The consumer task is blocked by `xQueueReceive` until producer one sends the input string, which will also be received by `xQueueReceive`. After the string is received on the consumer side, it will be printed to the console.

The main function is shown below in Listing 4. The two tasks are shown in Listings 5 and 6.

Listing 4: Problem 2 main

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

    queue1 = xQueueCreate(10, sizeof(char*));

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

    vTaskStartScheduler();

    while (1) {
    }

}
```

Listing 5: Problem 2 Producer Queue

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

    char *usr_str;
    usr_str = malloc(sizeof(char) * 10);

    printf("please enter a string\n");
    scanf("%s", usr_str);

    while (1) {
```

```
        status = xQueueSend(queue1, (void* ) &usr_str, portMAX_DELAY);
        if (status != pdPASS) {
            PRINTF("Queue Send failed!.\r\n");
            while (1)
                ;
        }

        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }

    vTaskDelete(NULL);
}
```

Listing 6: Problem 2 Consumer Queue

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

    char *receive_str;

    while (1) {
        status = xQueueReceive(queue1, (void*) &receive_str, portMAX_DELAY);

        if (status != pdPASS) {
            PRINTF("Queue Receive failed!.\r\n");

            while (1)
                ;
        }

        PRINTF("Received Value = %s\r\n", receive_str);
    }
}
```

Problem 3

1. It is possible to use a single producer semaphore to control the two consumers with a counting semaphore. This will make sure producers are shared on both consumer sides. But on the consumer side we will need two separate identities of counting semaphores of initial count 0 and max count of 1. This makes sure that by taking the semaphore from the producer. only one of the consumers can access the shared resource at a time.

Listing 7: Problem 3.1

```
producer_semaphore = xSemaphoreCreateCounting(2,2);
consumer1_semaphore = xSemaphoreCreateCounting(1,0);
consumer2_semaphore = xSemaphoreCreateCounting(1,0);
```

2. For this problem, we used a similar structure as the experiment. Two producer semaphores and one consumer counting semaphore which counts to 2 and is initialized to 2. The producer will take consumer semaphore and then give semaphore back to them. The user string is taken from the console as previous questions. When the producer task receives a string, it will expect a signal from both consumer tasks, then will signal both producer semaphores. The first consumer will print the string as it is, and the second consumer will change all the letters to the capital letters with the `toupper` function before printing the string.

The main function is shown below in Listing 8. The tasks are shown in Listings 9 to 11.

Listing 8: Problem 3 main

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

    usr_str = malloc(sizeof(char) * 100);

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

    vTaskStartScheduler();

    while (1) {
    }
}
```

Listing 9: Problem 3 Producer Task

```c
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;

    printf("please enter a string\n");
    scanf("%s", usr_str);

    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)
```

```
                ;
        }

        xSemaphoreGive(producer1_semaphore);
        xSemaphoreGive(producer2_semaphore);
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }

}
```

Listing 10: Problem 3 Consumer 1 Task

```
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 = %s\r\n", usr_str);
    }
}
```

Listing 11: Problem 3 Consumer 2 Task

```
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;

    char *receive_str_cap;

    receive_str_cap = malloc(sizeof(char) * 100);

    while (1) {
        xSemaphoreGive(consumer_semaphore);
        status = xSemaphoreTake(producer2_semaphore, portMAX_DELAY);

        if (status != pdPASS) {
            PRINTF("Failed to acquire producer2_semaphore\r\n");

            while (1)
                ;
        }

        for (int i = 0; i < 100; i++) {
            receive_str_cap[i] = toupper(usr_str[i]);
        }

        PRINTF("Received Value in captal = %s\r\n", receive_str_cap);
    }
}
```

Problem 4

1. The producer task can be similar or higher priority compared to the consumer task. The consumer does not have any urgent or dependent information from the producer.

2. We create a struct containing the semaphore and a counter. We pass the pointer to the struct object to the task parameter in order to pass down the counter. Then the consumer will have a switch case for different counter values to print the respective direction message.

   The main function is shown below in Listing 12. The tasks are shown in Listings 13 and 14.

Listing 12: Problem 4.2 main

```
typedef struct {
SemaphoreHandle_t sem;

int counter;
} wasd_handler;

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

    SemaphoreHandle_t my_sem = xSemaphoreCreateBinary();

    int my_counter = 0;
    wasd_handler my_wasd = { my_sem, my_count };
    wasd_handler *my_wasd_ptr = &my_wasd;

    xSemaphoreGive(my_sem);

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

Listing 13: Problem 4.2 Producer Task

```
void producer_event(void *pvParameters) {
    wasd_handler *my_sem = (wasd_handler *)pvParameters;

    BaseType_t status1,
    char c;

```

```
    while (1) {
        status1 = xSemaphoreTake(my_sem->sem, portMAX_DELAY);


        if (status1 != pdPASS) {
            PRINTF("Failed to acquire consumer_semaphore\r\n");

            while (1)
                ;
        }

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

            switch (c) {
            case 'a':
                xSemaphoreGive(my_sem->sem);
                vTaskDelay(1000 / portTICK_PERIOD_MS);
                my_sem -> counter = 0;
                break;
            case 'd':
                xSemaphoreGive(my_sem->sem);
                vTaskDelay(1000 / portTICK_PERIOD_MS);
                my_sem -> counter = 1;
                break;
            case 'w':
                xSemaphoreGive(my_sem->sem);
                vTaskDelay(1000 / portTICK_PERIOD_MS);
                my_sem -> counter = 2;
                break;
            case 's':
                xSemaphoreGive(pmy_sem->sem);
                vTaskDelay(1000 / portTICK_PERIOD_MS);
                my_sem -> counter = 3;
                break;
            }
        }
    }
}
```

Listing 14: Problem 4.2 Consumer Task

```
void consumer_event(void *pvParameters) {
    wasd_handler *my_sem = (wasd_handler *)pvParameters;
    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)
                ;
        }
        switch (my_sem->counter) {
        case '0':
            PRINTF("left\r\n");
            vTaskDelay(1000 / portTICK_PERIOD_MS);
            break;
        case '1':
            PRINTF("right\r\n");
            vTaskDelay(1000 / portTICK_PERIOD_MS);
            break;
        case '2':
            PRINTF("up\r\n");
            vTaskDelay(1000 / portTICK_PERIOD_MS);
            break;
        case '3':
```

```
            PRINTF("down\r\n");
            vTaskDelay(1000 / portTICK_PERIOD_MS);
            break;
        }
    }
}
```

3. We have one producer and two consumers. The producer increments the counter and sets the receive bit. Then the first consumer takes the receive bit to pass to the echo consumer by setting the echo bit to the event group, and the second consumer takes the echo bit to echo the string to the console.

   The main function is shown below in Listing 15. The tasks are shown in Listings 16 to 18.

Listing 15: Problem 4.3 main

```
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)
            ;
    }
    status = xTaskCreate(consumer2_event, "consumer", 200, (void*) event_group,
            3, NULL);
    if (status != pdPASS) {
        PRINTF("Task creation failed!.\r\n");
        while (1)
            ;
    }
    vTaskStartScheduler();
    while (1) {
    }
}
```

Listing 16: Problem 4.3 Producer Task

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

    while (1) {
        counter++;
        xEventGroupSetBits(event_group, receive_BIT);
        vTaskDelay(1000 / portTICK_PERIOD_MS);

    }
}
```

Listing 17: Problem 4.3 Consumer 1 Task

```c
void consumer_event(void *pvParameters) {
    EventGroupHandle_t event_group = (EventGroupHandle_t) pvParameters;
    EventBits_t bits;
    while (1) {
        bits = xEventGroupWaitBits(event_group,
        receive_BIT,
        pdTRUE,
        pdFALSE,
        portMAX_DELAY);

        if ((bits & receive_BIT) == receive_BIT) {
            xEventGroupSetBits(event_group, echo_BIT);
            vTaskDelay(1000 / portTICK_PERIOD_MS);
        }

    }
}
```

Listing 18: Problem 4.3 Consumer 2 Task

```c
void consumer2_event(void *pvParameters) {
    EventGroupHandle_t event_group = (EventGroupHandle_t) pvParameters;
    EventBits_t bits;
    while (1) {
        bits = xEventGroupWaitBits(event_group,
            echo_BIT,
        pdTRUE,
        pdFALSE,
        portMAX_DELAY);

        if ((bits & echo_BIT) == echo_BIT) {
            PRINTF("Received Value = %d\r\n", counter);
        }

    }
}
```

Problem 5

FreeRTOS provides interrupt service routine (ISR) safe functions as some functions may not operate properly when called by an interrupt.

The `xHigherPriorityTaskWoken` parameter is used when to determine if the interrupt needs to perform a *context switch* (switch the state of a task) due to a higher priority task being unblocked by a FreeRTOS API function. Context switches do not automatically occur inside ISR versions of API functions for several reasons, therefore this parameter is used to indicate if a higher priority task has been unblocked or not.

The `portYIELD FROM ISR` macro is used to request a context switch when inside an ISR. The argument of the macro is the parameter described above, and is used as `portYIELD FROM ISR(xHigherPriorityTaskWoken)`. If `xHigherPriorityTaskWoken` is set to `pdFALSE`, the macro will have no effect and not request any context switch. If `xHigherPriorityTaskWoken` is not set to `pdFALSE` (i.e. the value of the parameter was modified by an ISR safe API function), a context switch is requested and the task that is running may change (e.g. the running task may switch to a higher priority task).

Problem 6

We create a periodic timer with the callback function defined in `timerCallbackFunction2`. The timer callback function signals a semaphore every second. The task will block on `xSemaphoreTake` until it receives a signal from the semaphore, then print a message on the console.

The main function is shown below in Listing 19. The timer callback function is shown in Listing 20. The print task is shown in Listing 21.

Listing 19: Problem 6 Producer Task

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

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

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

Listing 20: Problem 6 Periodic Timer Callback Function

```
void timerCallbackFunction2(TimerHandle_t timer_handle) {
    SemaphoreHandle_t consumer_semaphore = semaphores[0];
    BaseType_t status;

    xSemaphoreGive(consumer_semaphore);
}
```

Listing 21: Problem 6 Print Task

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

    while (1) {
        status = xSemaphoreTake(consumer_semaphore, portMAX_DELAY);
```

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

        PRINTF("Hello World!\n");
    }
}
```

Problem 7

The variable `rc_values` contains a structure `RC_Values`, which stores 9 2-byte integers, one integer for the header and eight integers for each of the eight data channel. This structure is used to store the data received via UART.

The variable `ptr` contains a pointer to the address of `rc_values`, specifically the address to the first byte of `rc_values`. When the main function is run and a stream of data is received via UART, the first byte is checked and stored in `ptr`, or the first byte of `rc_values`. The value of this byte is checked to see if matches `0x20`, which is the first expected byte of data stream received. If this matches, the remaining 17 bytes are read via UART and stored at `&ptr[1]`, which points to the address of the second byte of `header` in the structure.