4DS4 Lab2 Report: Introduction to FreeRTOS

Johnnathan Gershkovich - 400408809
Joshua Umansky - 400234265
Shanzeb Immad - 400382928

Name	Contribution
Johnnathan Gershkovich	Aided in the implementations for Problems 1-7 Wrote the report for Problem 3
Joshua Umansky	Aided in the implementations for Problems 1-7 Wrote the report for Problem 1,2,4,7
Shanzeb Immad	Aided in the implementations for Problems 1,2,3,4. Wrote the report Problem 4,5,6

Problem #1

In order to accomplish this problem, we created two tasks (input_task and hello_task2). The first task would read in a value from the user using scanf, which saves the value globally to my_string, then deletes the task using vTaskDelete. The second task, takes our global variable my_string as a pointer input, and simply prints the value to the console every one second going forward.

```
void input_task(void *pvParameters)
{
	while(1)
	{
		PRINTF("Hello, please enter a string: ");
		SCANF("%s", my_string);
		vTaskDelay(1000 / portTICK_PERIOD_MS);
		vTaskDelete(NULL);

	}
}
```

Input_task Function

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

Hello_task2 Function

Problem #2

Adapting our solution from problem 1, we implemented the solution using a single queue (queue1) using two functions; producer_queue and consumer_queue. Our producer queue would accept a string from the user, which is then passed one character at a time into the queue.

```c
QueueHandle_t queue1 = (QueueHandle_t)pvParameters;
BaseType_t status;
int counter = 0;

PRINTF("ENTER INPUT STRING\r\n");
scanf("%s", pass_string);
StrLen = strlen(pass_string);

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

}
vTaskDelete(NULL);
```

Producer_queue Function

From that point, we use the consumer_queue to control printing the string to the console. Our function receives characters from the queue and appends them one at a time to the variable rec_string, which is ultimately printed once the string length has passed. In our implementation, StrLen is a global variable to ease the complexity of passing as a parameter.

```c
QueueHandle_t queue1 = (QueueHandle_t)pvParameters;
BaseType_t status;
int counter = 0;
char c;
while(1)
{
        status = xQueueReceive(queue1, (void *) &c, portMAX_DELAY);
        rec_string[counter] = c;
        counter++;
        if (status != pdPASS)
        {
                PRINTF("Queue Receive failed!.\r\n");
                while (1);
        }
        if(counter == StrLen+1){
                //rec_string[counter] = '\0';
                while(1){
                        PRINTF("%s\r\n", rec_string);
                        vTaskDelay(1000 / portTICK_PERIOD_MS);
                }
        }
        //PRINTF("Received Value = %s\r\n", received_string);
}
```

Consumer_queue Function

Problem #3

Problem #3.1

The way our group was able to utilize 1 counting producer semaphore was setting the max size to 2. Each count value is assigned to a producer semaphore which allows for each received value to be printed twice. The code snippet below shows the initializations of the semaphores changing semaphore[0] to a counting sem similar to the third one added in the experiments.

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

SemaphoreHandle_t* semaphores = (SemaphoreHandle_t*) malloc(3 * sizeof(
SemaphoreHandle_t));
semaphores[0] = xSemaphoreCreateCounting(2, 2); //Producer semaphore
semaphores[1] = xSemaphoreCreateBinary(); //Consumer semaphore
semaphores[2] = xSemaphoreCreateCounting(2, 2);
```

The next code below shows how the producer and consumer sem were assigned as both the producers are using the new counting semaphore made while the consumer remains the same from the experiment.

```
void producer_sem(void* pvParameters)
{
    SemaphoreHandle_t* semaphores = (SemaphoreHandle_t*)pvParameters;
    SemaphoreHandle_t producer1_semaphore = semaphores[0];
    SemaphoreHandle_t producer2_semaphore = semaphores[0];
    SemaphoreHandle_t consumer_semaphore = semaphores[2];
    BaseType_t status1, status2;
    while(1)
```

After executing this code, we see that the application still functions correctly, as there are no issues handling the multiple producers with a single consumer using this method, as the counting semaphores can act as two individual binary semaphores, as the value of semaphores[0] can simply be incremented, and decremented as required.

Problem #3.2

Adding a third task to print in upper case, all while synchronizing the three tasks using semaphores was accomplished using a single producing function. We started with the base producer function used in the previous problems, and augmented this code to include two semaphores (producer1 and producer2), which are used to signal when a character is ready for the consumer functions to receive a character. In order to synchronize with the consumer functions, we also use a counting semaphore (consumer_semaphore), which is use to lock the consumers while sending a new character.

```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;
	int counter = 0;

	PRINTF("ENTER INPUT STRING\r\n"); //scanf doesnt work w spaces
	scanf("%[^\n]", pass_string);
	StrLen = strlen(pass_string);

	while(counter <= StrLen)
		{
		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);
		}
		//Before is setup, after is get string
		global_char = pass_string[counter];
		counter++;

		//this means string is ready
		xSemaphoreGive(producer1_semaphore);
		xSemaphoreGive(producer2_semaphore);

		while(counter == StrLen+1)

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

Producer1 Function

```c
void consumer1_sem(void* pvParameters)
{
	SemaphoreHandle_t* semaphores = (SemaphoreHandle_t)pvParameters;
	SemaphoreHandle_t producer1_semaphore = semaphores[0];
	SemaphoreHandle_t consumer_semaphore = semaphores[2];
	//PRINTF("Strlen %d\n", StrLen);
	char print_string[StrLen];
	int counter = 0;
	BaseType_t status;
	while(1)
	{
		status = xSemaphoreTake(producer1_semaphore, portMAX_DELAY);

		if (status != pdPASS)
		{
			PRINTF("Failed to acquire producer1_semaphore\r\n");
			while (1);
		}

		print_string[counter] = global_char;
		counter++;

		if(counter == StrLen+1){
			//rec_string[counter] = '\0';
			while(1){
				PRINTF("%s\r", print_string);
				vTaskDelay(1000 / portTICK_PERIOD_MS);
			}
		}

		xSemaphoreGive(consumer_semaphore);
	}
}
```

Consumer1 Function

Problem #4

Problem #4.1

Through experimentation, we determined that if the priority of the consumer is equal to or lower than the priority of the producer, the consumer gets stuck in an infinite waiting cycle. Once the priority of the producer is set to be lower than the consumer, there is then no problem and the producer/consumer handshake occurs how it is expected.

Problem #4.2

In order to implement this problem using semaphores instead of event groups, we created four semaphores to represent the direction entered by the user. The producer event then signals the respective semaphore when the corresponding direction is entered. In this problem, our producer event has a priority of one, while the consumer events have a priority of two.

Four separate consumer functions exist, one for each direction, which waits for the semaphore to trigger, which then prints the direction chosen.

```c
void producer_event(void* pvParameters)
{
	SemaphoreHandle_t* semaphores = (SemaphoreHandle_t*)pvParameters;

	BaseType_t status;
	char c;
	while(1)
	{
		scanf("%c", &c);
		switch(c)
		{
			case 'a':
				xSemaphoreGive(semaphores[2]);
				break;
			case 's':
				xSemaphoreGive(semaphores[1]);
				break;
			case 'd':
				xSemaphoreGive(semaphores[3]);
				break;
			case 'w':
				xSemaphoreGive(semaphores[0]);
				break;
		}
	}
}
```

Producer function

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

	while(1){
		status = xSemaphoreTake(sem, portMAX_DELAY);
		PRINTF("Up\r\n");
	}
}
```

Example of one consume function (Up)

Problem #4.3

To incorporate event groups, we first declared an event group that we would modify in our producer_sem function. In the code snippet below, we see that we wait until the values of the bits equals 0x3 (11 in binary), which then sets the value of 0xC, which in turn triggers the consumer_sem function to then print the counter value. Once the two consumers print the counter value, they reset the total value of bits back to 0x3. Once the producer sees the change, it increments the counter so the process continues but with a new value printed.

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

        xEventGroupSetBits(event_group, 0x3);

        while(1)
        {
                xEventGroupWaitBits(event_group,
                                                        0x3,
                                                        pdTRUE,
                                                        pdTRUE,
                                                        portMAX_DELAY);

                xEventGroupSetBits(event_group, 0xC);

                counter++;

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

Producer Function

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

	while(1)
	{
		xEventGroupWaitBits(event_group,
								1 << 3,
								pdTRUE,
								pdFALSE,
								portMAX_DELAY);

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

		xEventGroupSetBits(event_group, 1 << 1);
	}
}

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


	BaseType_t status;

	while(1)
	{
		xEventGroupWaitBits(event_group,
								1 << 2,
								pdTRUE,
								pdFALSE,
								portMAX_DELAY);

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

		xEventGroupSetBits(event_group, 1 << 0);
	}
}
```

Consumer 1 and 2 Functions

Problem #5

The macro portYIELD_FROM_ISR acts as a request for a context switch at the end of the ISR when the parameter xHigherPriorityTaskWoken is set to true. This means that when the interrupt has finished, it returns to the task with the highest priority to be completed. The variable xHigherPriorityTaskWoken is of type BaseType_t which is initialized to pdFalse, this variable essentially acts as a tracker to see if a higher priority task has been unblocked by the ISR. With experiment 3, if the switch case observed any of the characters activated the corresponding bit is flipped and sets the variable to true. The portYield then checks if the variable is true which switches context to print the appropriate direction from the consumer task.

Problem #6

For this problem, a binary semaphore was implemented so the timers callback signals it to a task. The binary semaphore was initialized to 0 which indicates that the rask/resource is not available. Similar to the experiment, the timerCallBackFunction2 for the periodic timer is the function that gives/signals the semaphore. Once the semaphore has been signaled, the timerTask which was being blocked/waiting indefinitely by the semaphore can freely run and print the required string, this process then runs every second by the periodic timer which continues to signal the semaphore while the task then requests it.

```
void timerCallbackFunction2(TimerHandle_t timer_handle)
{
	xSemaphoreGive(semaphore);

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

timerCallbackFunction2

```
void timer_task(void *pvParameters)
{
	while(1) {
		xSemaphoreTake(semaphore, portMAX_DELAY);
		PRINTF("Yup, the timer and semaphore worked\n");
	}
}
```

Timer task

Problem #7

Both rc_values, and ptr are important for receiving and managing data from the UART channel. Rc_values is a structure of RC_Values that is used to hold the data from the multiple input channels from the controller. We define the structure specifically that when the data is read into the structure, the first 16 bits are read into the header, the next 16 are read into ch1, etc. These values are simply printed everytime a full UART input has been accepted.

```c
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;
```

RC_Values Structure definition

Ptr is initialized to point to the location of rc_values (uint8_t*) which is used to point the UART values being read into the location corresponding in the rc_values array.

```c
uint8_t* ptr = (uint8_t*) &rc_values;
```

Ptr initialization