CG2271 Real Time Operating Systems
Lab 5 – FreeRTOS and Real-Time Scheduling
Answer Book

Name: Deepak Buddha	Matric Number: A0155454A
Name: Anton Chua	Matric Number: A0161811N

Submission Deadline: Friday 27 Oct midnight
Submit under IVLE → Files → Lab4-Submissions → Your Lab Group

Question 1A (1 mark)

It is possible to create two different tasks using the same function as each task has its own stack and its own (virtual) set of processor registers and variables ,which declared in task creation, are placed on the stack. Threads have different stacks, so the task local variables are different for them.

For Task 1and Task 2 since the variables passed in (int 1 or 2) are different and both the tasks are created separately with its own stack, we can create two tasks with same function task1and2 as both tasks execute the same function but with different local variables.

Question 1B (1 mark)

The program is expected to write to the serial port “Task 1” or “Task 2” with a new line each time task 1 and task 2 occur.

Question 1C (2 marks)

There is garbled output as serial port communication takes time and both the tasks (1 & 2) are competing to access the same function and use some shared data. So tasks 1and 2 are executing very quickly and entering the critical selection in the function concurrently at a faster rate than serial port communication occurs resulting in a race condition where both the tasks are accessing the shared data in task1and2 function causing this data to be changed while the other task is still accessing it leading to the garbled output.

Question 2A (2 mark)

Binary semaphores are used for both mutual exclusion and synchronisation purposes.

Binary semaphores and mutexes are very similar but have some differences: Mutexes are binary semaphores that include a priority inheritance mechanism, which only binary semaphores do not. This makes binary semaphores the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.

Question 2B (4 marks)

Refer to //QUESTION 2// in code segment at the end of the report.

Question 3A (4 mark)

Refer to //QUESTION 3// in code segment at the end of the report.

Question 3B (2 marks)

Task 2 is indeed appearing more frequently as much as Task 1 and this is because in FreeRTOS, the highest priority task waiting in the queue is made ready , instead of FIFO order and since Task 2 is of higher priority than Task 1, once a slot becomes available in a full message queue, Task 2 will be unblocked and is able to send taskNum to message queue again instead of Task 1, and hence appears almost twice as much as Task 1 in the serial print monitor.

Question 4A (1 mark)

The special primitive xSemaphoreGiveFromISR() is needed to perform V() operation as the semaphore needs to given based on interrupt routine before the semaphore can be taken in the task which is UNLIKE the usual xSemaphoreGive() primitive that is used to release the semaphore after the semaphore has already been taken and once the task is done executing.

Question 4B (1 mark)

This primitive also contains the parameter *pxHigherPriorityTaskWoken. This allows for tasks to be unblocked based on priority and not FIFO. SemaphoreGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if giving the semaphore caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xSemaphoreGiveFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Question 4C (5 marks)

Refer to //QUESTION 4// in code segment at the end of the report.

Question 5A (5 marks)

Refer to //QUESTION 5// in code segment at the end of the report.

Question 5B (1 marks)

Every time the button is pressed every 5 seconds and the potentiometer is moved the left by a little with each button press, the values printed on the serial port are gradually reduced based on the potentiometer values. This is because with each button press the producer is reading the values of potentiometer at the different times of presses and the consumer is printing this values, hence a direct change or reduction in values are observed.

Question 5C (1 marks)

Every time the button is pressed every 2 seconds and the potentiometer is moved the left by a little with each button press, we can see the values in the serial print are jumping and reducing drastically compared to 5B. This is because the consumer only checks the producer every 5 seconds and the values are removed slower than entered in the buffer hence causing a big jump between those values that are not consumed and printed.

```
////////////////////////// LAB 5 G32 CODESEGMENT ////////////////////////////

#include <Arduino.h>
#include <avr/io.h>
```

```
#include <FreeRTOS.h>
#include <task.h>
#include <semphr.h>
#include <queue.h>

#define STACK_SIZE 200
#define LED_PIN6 6
#define LED_PIN7 7
#define PIN_PTTM 0
#define SIZE 4
#define LONG_TIME 0xffff

int buf[SIZE]; // shared buffer with 4 entries
int in = 0;
int out = 0;
SemaphoreHandle_t xSemaphore0;
SemaphoreHandle_t xSemaphore1;
SemaphoreHandle_t xSemaphoreE;// semaphore with number of empty slots
SemaphoreHandle_t xSemaphoreF; // semaphore with no slots (full)
SemaphoreHandle_t xSemaphoreM; // mutex semaphore
SemaphoreHandle_t xSemaphoreB; // binary semaphire
SemaphoreHandle_t xSemaphore;

QueueHandle_t xQueue1;
unsigned long last_interrupt_time0 = 0;
unsigned long last_interrupt_time1 = 0;
BaseType_t xHigherPriorityTaskWoken;


//////////////////////////// QUESTION 1 ////////////////////////////////
//void task1and2(void *p) {
//      while (1) {
//              int taskNum = (int) p;
//              Serial.print("Task ");
//              Serial.println(taskNum);
//              vTaskDelay(1);
//      } }
//void setup() {
//      Serial.begin(115200);
//}
//void loop() {
//      /* create two tasks one with higher priority than the other */
//      xTaskCreate(task1and2, "Task1", STACK_SIZE, (void * ) 1, 1, NULL);
//      xTaskCreate(task1and2, "Task2", STACK_SIZE, (void * ) 2, 2, NULL); /* start
scheduler */
//      vTaskStartScheduler();
//}

//////////////////////////// QUESTION 2 ////////////////////////////////
void task1and2(void *p) {
        while (1) {
                if( xSemaphore != NULL ) {
                        if (xSemaphoreTake( xSemaphore, ( TickType_t ) 10 ) == pdTRUE )
{
                                int taskNum = (int) p;
                                Serial.print("Task ");
                                Serial.println(taskNum);
                                xSemaphoreGive( xSemaphore );
                                vTaskDelay(1);
                        }
                        else {
                                Serial.print("Semaphore cannot be obtained"); // error
message
```

```
                        }
                }
        }
}
void setup() {
        Serial.begin(115200);
        xSemaphore = xSemaphoreCreateMutex();
}
void loop() {
        /* create two tasks one with higher priority than the other */
        xTaskCreate(task1and2, "Task1", STACK_SIZE, (void * ) 1, 1, NULL);
        xTaskCreate(task1and2, "Task2", STACK_SIZE, (void * ) 2, 2, NULL);
        /* start scheduler */
        vTaskStartScheduler();
}

////////////////////////// QUESTION 3 ////////////////////////////////
void serialPrint(void *) {
        while (1) {
                int taskNum;
                if (xQueueReceive( xQueue1, &(taskNum), ( TickType_t ) 10 )) {
                Serial.print("Task ");
                Serial.println(taskNum);
                }
                else {
                        Serial.print("Task unable to be received from queue"); // error
message
                }
        }
}

void task1and2(void *p) {
      while (1) {
            int taskNum = (int) p;
            if (xQueueSend( xQueue1, ( void * ) &taskNum, ( TickType_t ) 0 ) ==
pdPASS) {
            vTaskDelay(1);
            }
            else {
              Serial.print("Task unable to be sent to queue"); // error message
            }
      }
}

void setup() {
        Serial.begin(115200);
        xQueue1 = xQueueCreate( 2, sizeof(int));
}

void loop() {
        /* create two tasks one with higher priority than the other */
        xTaskCreate(task1and2, "Task1", STACK_SIZE, (void * ) 1, 2, NULL);
        xTaskCreate(task1and2, "Task2", STACK_SIZE, (void * ) 2, 3, NULL);
        xTaskCreate(serialPrint, "Print", STACK_SIZE, NULL,  1 , NULL);
        /* start scheduler */
        vTaskStartScheduler();
}

////////////////////////// QUESTION 4 ////////////////////////////////
//Flashes LED at pin 7: 5 times at 4 Hz
void int0task(void *)
{
        while(1) {
```

```
                if( xSemaphore0 != NULL ) {
                        if( xSemaphoreTake( xSemaphore0, LONG_TIME ) == pdTRUE ) {
                                for (int i=0; i<5; i++) {
                                        digitalWrite(LED_PIN7, HIGH);
                                        delay(125);
                                        digitalWrite(LED_PIN7, LOW);
                                        delay(125);
                                }
                        }
                }
        }
}
// Flashes LED at pin 6: 5 times at 2 Hz
void int1task(void *)
{
        while(1) {
                if( xSemaphore1 != NULL ) {
                        if( xSemaphoreTake( xSemaphore1, LONG_TIME ) == pdTRUE ) {
                                for (int i=0; i<5; i++) {
                                        digitalWrite(LED_PIN6, HIGH);
                                        delay(250);
                                        digitalWrite(LED_PIN6, LOW);
                                        delay(250);
                                }
                        }
                }
        }
}

void int0ISR()
{
        xHigherPriorityTaskWoken = pdFALSE;
        unsigned long interrupt_time0 = millis();
        if (interrupt_time0 - last_interrupt_time0 > 200) { //de-bouncing

                xSemaphoreGiveFromISR(xSemaphore0, &xHigherPriorityTaskWoken );
                if (interrupt_time0 > 200) {
                        last_interrupt_time0 = interrupt_time0;
                }
        }

        if (xHigherPriorityTaskWoken) {
        taskYIELD();
        }
}

void int1ISR()
{
        xHigherPriorityTaskWoken = pdFALSE;
        unsigned long interrupt_time1 = millis();
        if (interrupt_time1 - last_interrupt_time1 > 200) { //de-bouncing
                xSemaphoreGiveFromISR(xSemaphore1, &xHigherPriorityTaskWoken );
                if (interrupt_time1 > 200) {
                        last_interrupt_time1 = interrupt_time1;
                }
        }

        if (xHigherPriorityTaskWoken) {
        taskYIELD();
        }
}

void setup()
```

```
{
        Serial.begin(115200);
        xSemaphore0 = xSemaphoreCreateBinary();
        xSemaphore1 = xSemaphoreCreateBinary();
        pinMode(LED_PIN6, OUTPUT);
        pinMode(LED_PIN7, OUTPUT);
        attachInterrupt(0, int0ISR, RISING);
        attachInterrupt(1, int1ISR, RISING);
}

void loop() {
        /* create two tasks one with higher priority than the other */
    xTaskCreate(int1task, "Task1", STACK_SIZE, NULL, 1, NULL);
        xTaskCreate(int0task, "Task0", STACK_SIZE, NULL, 2, NULL);
        /* start scheduler */
        vTaskStartScheduler();
}


////////////////////////// QUESTION 5 ////////////////////////////////

void producer (void *) {
        while(1) {
                if( xSemaphoreTake( xSemaphoreB, LONG_TIME ) == pdTRUE ) {
                   if (xSemaphoreTake( xSemaphoreE, ( TickType_t ) 10 ) == pdTRUE) {
                        if (xSemaphoreTake( xSemaphoreM, ( TickType_t ) 10 ) == pdTRUE)
{
                                        int data = analogRead(PIN_PTTM);
                                        buf[in] = data;
                                        in = (in+1) % SIZE;
                                }
                        }
                        xSemaphoreGive(xSemaphoreM);
                        xSemaphoreGive(xSemaphoreF);
                }
        }
}

void consumer (void *) {
        TickType_t xLastWakeTime;
        const TickType_t xFrequency = 5000;
        for( ; ;) {
                int c;
                if (xSemaphoreTake( xSemaphoreF, ( TickType_t ) 10 ) == pdTRUE) {
                        if (xSemaphoreTake( xSemaphoreM, ( TickType_t ) 10 ) == pdTRUE)
{
                                c = buf[out];
                                Serial.println(c);
                                out = (out + 1) % SIZE;
                                xSemaphoreGive(xSemaphoreM);
                                xSemaphoreGive(xSemaphoreE);
                        }
                }
                vTaskDelayUntil( &xLastWakeTime, xFrequency );
        }
}

void int0ISR()
{
        xHigherPriorityTaskWoken = pdFALSE;
        unsigned long interrupt_time0 = millis();
        if (interrupt_time0 - last_interrupt_time0 > 200) { //de-bouncing
```

```
                xSemaphoreGiveFromISR(xSemaphoreB, &xHigherPriorityTaskWoken );
                if (interrupt_time0 > 200) {
                        last_interrupt_time0 = interrupt_time0;
                }
        }

        if (xHigherPriorityTaskWoken) {
        taskYIELD();
        }
}

void setup()
{
        Serial.begin(115200);
        xSemaphoreB = xSemaphoreCreateBinary();
        xSemaphoreE = xSemaphoreCreateCounting(4, 0); // semaphore with number of
empty slots
        xSemaphoreF = xSemaphoreCreateCounting(4, 4);// semaphore with no slots (full)
        xSemaphoreM = xSemaphoreCreateMutex();
        attachInterrupt(0, int0ISR, RISING);
}

void loop() {
    xTaskCreate(producer, "Producer", STACK_SIZE, NULL, 1, NULL);
        xTaskCreate(consumer, "Consumer", STACK_SIZE, NULL, 1, NULL);
        /* start scheduler */
        vTaskStartScheduler();
}
```