

Laboratory 2

ENGG4420: Real-Time Systems Design

Instructor:

Dr.Radu Muresan

Group 17: Wed-8:30 Section

Bilal Ayyache: 0988616

Robert Mackenzie Beggs: 0819747

Lab Start Date: October 14th, 2020

Lab End Date: October 23rd, 2020

Contents

1	Introduction	1
1.1	Problem Description	1
1.2	System Requirements	1
2	Background	1
2.1	Benefits of TrueStudio	1
2.2	Semaphores	2
2.3	On Priority and Priority Inversion	2
2.4	Mutual Exclusion Semaphores	2
2.5	Queues	2
2.6	Hardware/Software Implementation Details	2
3	Consumer Producer Implementation	3
3.1	Object Initialization	3
3.1.1	Threads and Ids	3
3.1.2	Binary Semaphore	4
3.1.3	Queues	4
3.2	Consumer Tasks	4
3.2.1	StartTask1	4
3.2.2	StartTask3	5
3.3	Producer Tasks	5
3.3.1	StartTask2	5
4	Example Expansion Implementation	6
4.1	Semaphore Tasks	6
4.1.1	StartTask1	6
4.1.2	StartTask2	6
4.2	Queue Tasks	7
4.2.1	StartTask1	7
4.2.2	StartTask2	7
4.3	Mutex Tasks	8
4.3.1	StartTask1	8
4.3.2	StartTask2	8
5	Conclusion	9

List of Figures

1	The consumer and producer problem illustrated via 3 tasks.	3
2	Priorities were defined for each of the tasks.	4
3	A binary semaphore was used to encourage mutual exclusion.	4
4	Task queues were defined	4
5	Consumer and Producer task progress output	5
6	Semaphore tasks	6
7	Output from the interaction of Semaphore Task 1 and Task 2	6
8	Queue tasks	7
9	Output that is created through Task 1 and Task 2	7
10	Mutex Tasks	8
11	The output that is a product of the interaction of the Mutex StartTask1 and StartTask2 . . .	8

1 Introduction

1.1 Problem Description

One challenge associated with modelling real time systems is that they must adhere to several temporal constraints; the structure of the operating system must support the demands associated with the deadlines, delay and duration characteristics of a given task, all of which are continuously reported. This poses a problem in ensuring the consistency and integrity of data. To accommodate this demand, tasks are often partitioned into foreground and background processes via the aid of a real time operating system (RTOS). Further expanding the concept of foreground and background processes to multiple partitions allows for the support of multiple processes to be implemented. Multitasking allows for separate tasks to be simultaneously created, priority scheduled and halted and resumed on demand. Given a working example of a FreeRTOS and an uC/OS-III application example, the functionality of working example was improved by considering the deadlines, delay and duration characteristics of each task. Finally, a producer and consumer problem was solved through utilizing FreeRTOS.

1.2 System Requirements

To execute multiprocessing in any capacity, synchronization must occur. Synchronization in a real-time environment means utilizing mechanisms such as event flags or counting semaphores to simultaneously update their respective tasks with the corresponding task or interrupt service routine. A working example with code relevant to completing the tasks of initialization, screen display, button and command prompt interaction was provided in FreeRTOS. A complete system, capable of performing two different tasks separately, then in parallel, was required. These tasks were to take the form of a traditional producer/consumer structure in which one task generates an output and the other consumes it. Additionally, the task was displayed via the LCD display. In implementing an accurate model, knowledge of the components necessary for a RTOS was gained; the concepts of queues, priority, mutex and semaphores were required for completion of the system.

2 Background

2.1 Benefits of TrueStudio

Real time systems are in a state of continuous state of interaction with their environment. To model a real time system, both the plant and environment need to be accurately replicated. This degree of constant interaction provides a challenge for debugging as tasks are divided into foreground and background processes and often possess additional time constraints. TrueStudio is a free tool for students and researchers built on Eclipse, CDT, GCC and GDB that features the ability to perform RTOS aware debugging [3].

2.2 Semaphores

A major challenge in RTOS is managing access to shared resources, ensuring that no tasks are simultaneously accessing a given resource; this is the concept of mutual exclusion. Semaphores are kernel objects used to synchronize tasks, thereby controlling access to shared resources. Furthermore, given a task, it is possible to interrupt it in order to alert it that an important state change has occurred; interrupt service routines (ISR), alert a task that an event of interest has occurred. In order to synchronize this alert to a given task, semaphores are used. While counting semaphores are used to track how many times a resource has been accessed, binary semaphores are used to ensure mutual exclusion and synchronize tasks that utilize pooled resources [1].

2.3 On Priority and Priority Inversion

Priority refers to the order in which two tasks that share a given resource should be executed; priority inversion occurs in cases in which lower priority tasks precede higher priority tasks in a given interval.

2.4 Mutual Exclusion Semaphores

Mutual exclusion semaphores (mutex) are a subcategory of binary semaphores that eliminate the possibility of unbounded priority inversion [1].

2.5 Queues

Given a collection of tasks that are prepared to interact with the processor, a queue is formed. A first in first out (FIFO) queue gives the task in the order they appear to the processor. In certain scenarios this is appropriate, however, in the paradigm of RTOS, priority needs to be considered to maintain mutual exclusion [1].

2.6 Hardware/Software Implementation Details

The implementation of this system was completed using TrueStudio as an IDE, and the STM32F429I-DISC1 (STM32f4) as an ST evaluation board. The evaluation board featured the STMSTM32F429 microcontroller. To be able to successfully implement the extensions to the provided examples, familiarity with TrueStudio and the STM32f4 was developed through reading provided documentation. Familiarity with FreeRTOS and uC/OS-III was developed by interacting with the provided examples and engagement with the tutorial. The bulk of the project was completed using FreeRTOS. FreeRTOS was selected over uC/OS-III as it offers synchronization and inter-task communication via queues and semaphores as well as execution trace functionality. Additionally, the syntax was perceived to be more intuitive than that of uC/OS-III.

3 Consumer Producer Implementation

The code in Figure 1 depicts the 3 tasks that were programmed for the consumer/producer problem.

```

//Consumer
void StartTask1(void const * argument){
    osEvent retvalue;

    while(1){
        retvalue = osMessageGet(myQueue01Handle, osWaitForever);
        if( (((data1 *)retvalue.value.p)->Value) == 100){
            osSemaphoreRelease(myBinarySem01Handle);
            osDelay(1000);
        }else{
            myprintf("Percentage Completed :%d\n\r",(((data1 *)retvalue.value.p)->Value));
        }
    }
}

void StartTask3(void const * argument){
    while(1){
        osSemaphoreWait(myBinarySem01Handle, osWaitForever);
        myprintf("DONE\n\r");
        BSP_LCD_SetTextColor(LCD_COLOR_GREEN);
        BSP_LCD_DisplayStringAtLine(1, (uint8_t *)"          SYSTEM          ");
        BSP_LCD_SetTextColor(LCD_COLOR_GREEN);
        BSP_LCD_DisplayStringAtLine(2, (uint8_t *)"          ")";
        BSP_LCD_SetTextColor(LCD_COLOR_GREEN);
        BSP_LCD_DisplayStringAtLine(3, (uint8_t *)"          READY          ");
        BSP_LCD_SetTextColor(LCD_COLOR_GREEN);
        BSP_LCD_DisplayStringAtLine(5, (uint8_t *)"          ");
        osDelay(1000);
    }
}

//Producer
void StartTask2(void const * argument){
    while(1){
        DataToSend.Value = DataToSend.Value + 10;
        osMessagePut(myQueue01Handle, (uint32_t)&DataToSend, 200);
        osDelay(1000);

        if(DataToSend.Value == 100){
            DataToSend.Value = 0;
        }
    }
}
    
```

Figure 1: The consumer and producer problem illustrated via 3 tasks.

3.1 Object Initialization

3.1.1 Threads and Ids

In order to successfully have the tasks operate in real time, priority levels for each of the tasks were defined; this can be seen in Figure 2. 'osThreadDef' takes arguments which define the name of the task, the priority of the task, number of instances and the stack size in bytes [2]. From highest to lowest the priorities are assigned as Task3, Task1, and then Task2. 'osThreadCreate' generates a new thread with a unique thread id given a thread definition. Handles for each thread reference the unique id corresponding to a thread.

```

osThreadDef(Task1, StartTask1, osPriorityAboveNormal, 0, 128);
Task1Handle = osThreadCreate(osThread(Task1), NULL);

osThreadDef(Task2, StartTask2, osPriorityNormal, 0, 128);
Task2Handle = osThreadCreate(osThread(Task2), NULL);

osThreadDef(Task3, StartTask3, osPriorityHigh, 0, 128);
Task3Handle = osThreadCreate(osThread(Task3), NULL);
    
```

Figure 2: Priorities were defined for each of the tasks.

3.1.2 Binary Semaphore

Given the name of a semaphore as an argument, a semaphore object is defined by 'osSemaphoreDef'. 'osSemaphore' provides direct access to the Semaphore object given a reference; this access is required by 'osSemaphoreCreate' in order to initialize a previously defined semaphore object such that it can receive a specified number of system resources; the second argument defines the number of resources it will be allowed to access. Passing 1 as the argument indicates that a binary semaphore should be created [2].

```

osSemaphoreDef(myBinarySem01);
myBinarySem01Handle = osSemaphoreCreate(osSemaphore(myBinarySem01), 1);
    
```

Figure 3: A binary semaphore was used to encourage mutual exclusion.

3.1.3 Queues

By utilizing 'osMessageQDef' a queue is defined according to the specified arguments; name, queue size, and type. Queue size refers to the maximum number of objects in a queue, here it is specified to be 1. Furthermore, the name is specified to be myQueue01 and the type is given as the memory address to the data that is to be sent to a task. Additionally, a communication task queue is also created via the same process.

```

osMessageQDef(myQueue01, 1, &DataToSend);
myQueue01Handle = osMessageCreate(osMessageQ(myQueue01), NULL);

// Communication task queue
osMessageQDef(CommQueue, 1, &DataVCP);
CommQueueHandle = osMessageCreate(osMessageQ(CommQueue), NULL);
//-----
    
```

Figure 4: Task queues were defined

3.2 Consumer Tasks

3.2.1 StartTask1

StartTask1 utilizes several functions from the CMSIS RTOS API to implement a progress bar routine. 'osMessageGet' returns a message from a queue immediately or after waiting a given amount of time. The

first argument represents the id of the queue and the second argument specifies how long the current thread should wait, in milliseconds, for a message to arrive. The argument 'osWaitForever' is a reserved keyword that indicates that the thread should wait indefinitely for a response. The return value is checked to see if it is equal to 100, indicating that the queue is full. If it is, 'osSemaphoreRelease' is passed the id of the Binary Semaphore, and the id is freed; this allows for other processes waiting for this id to move to a ready state [2]. If the queue is not yet full, the current percentage complete is displayed; this is visible in Figure 5.

```
Percentage Completed :10
Percentage Completed :20
Percentage Completed :30
Percentage Completed :40
Percentage Completed :50
Percentage Completed :60
Percentage Completed :70
Percentage Completed :80
Percentage Completed :90
DONE
```

Figure 5: Consumer and Producer task progress output

3.2.2 StartTask3

StartTask3 configures the settings related to the LCD display; the display colour and message is altered when this task is engaged. 'osSemaphoreWait' takes two arguments, the first of which is an id corresponding to a binary semaphore, the second is the time that the semaphore should hang before a semaphore token is obtainable. In this case, it is indicated that the system should wait until a token becomes available. When a token is available, a message indicating that the process is complete is printed. Following this interaction, several STM32 functions are utilized. 'BSP_LCD_SetTextColor()' was used to set the text colour; the input type is a uint16_t representing colour. 'BSP_LCD_DisplayStringAtLine' displays specified characters at a given x position on the screen. Finally, 'osDelay' waits a specified time, in milliseconds, before proceeding; a status code is returned indicating that the allotted time has passed.

3.3 Producer Tasks

3.3.1 StartTask2

The function of StartTask2 is to increment the value to be displayed in StartTask1. While this task is engaged, the data to be sent to other tasks is continually incremented. Given the id of a queue, some data, and an amount of time in milliseconds, 'osMessagePut' will place the specified message into the queue given a priori via the queue id; the time provided indicates a timeout value. The task then waits for 1000 milliseconds. If the process is finished, the value of data is reset to 0.

4 Example Expansion Implementation

4.1 Semaphore Tasks

4.1.1 StartTask1

Task 1 continuously prints a message stating that Task 1 is running, waits for 2000 milliseconds using 'osDelay', prints out a message informing the user that the semaphore for this task has been released before then releasing the semaphore. Given the unique id of a semaphore, 'osSemaphoreRelease' releases a semaphore to the system, increasing the amount of tokens available [2]. The system then waits another 1000 milliseconds. The process is repeated indefinitely; the output is visible in Figure 7.

```
void StartTask1(void const * argument)
{
    //This While loop utilizes the Semaphore
    while(1)
    {
        myprintf ("Task1 running .....\\n\\r");
        osDelay(2000);
        myprintf ("Task1 Release Semaphore\\n\\r");
        osSemaphoreRelease(myBinarySem01Handle);
        osDelay(1000);
    }
}

void StartTask2(void const * argument)
{
    //This While loop utilizes the Semaphore
    while(1)
    {
        osSemaphoreWait(myBinarySem01Handle, osWaitForever);
        myprintf ("Task2 synchronized\\n\\r");
    }
}
```

Figure 6: Semaphore tasks

4.1.2 StartTask2

Task 2 continuously calls 'osSemaphoreWait' to wait indefinitely until a semaphore token becomes available. When a token becomes available, a message indicating that Task2 has synchronized is given to output, as pictured in Figure 7.

```
Task1 running .....
Task1 Release Semaphore
Task2 synchronized
Task1 running .....
Task1 Release Semaphore
Task2 synchronized
Task1 running .....
Task1 Release Semaphore
Task2 synchronized
```

Figure 7: Output from the interaction of Semaphore Task 1 and Task 2

4.2 Queue Tasks

4.2.1 StartTask1

StartTask1 prints a message stating that Task 1 is running, before 'osMessagePut' places the message in a queue given a queue id, the address of the information and a timeout value. Finally, the process is paused for 1000 milliseconds before it repeats indefinitely.

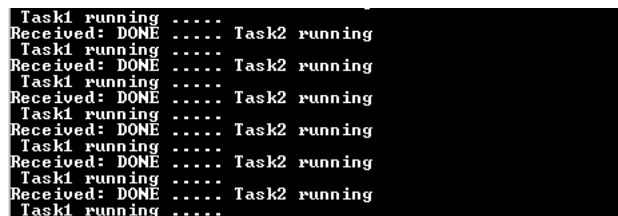
```
void StartTask1(void const * argument)
{
    //This While loop Utilizes the Queue
    while(1)
    {
        myprintf ("Task1 running .....\\n\\r");
        osMessagePut(myQueue01Handle, (uint32_t)&DataToSend, 200);
        osDelay(1000);
    }
}

void StartTask2(void const * argument)
{
    osEvent retvalue;
    //This While loop utilizes the Queue
    while(1)
    {
        retvalue = osMessageGet(myQueue01Handle, osWaitForever);
        myprintf ("Received: %s", (char *)(((data *)retvalue.value.p)->Value));
        myprintf (" ..... Task2 running\\n\\r ");
    }
}
```

Figure 8: Queue tasks

4.2.2 StartTask2

StartTask2 works in tandem with StartTask1 and receives the message from 'osMessagePut' via 'osMessageGet' given a queue id and a timeout period; it is specified that the system should wait indefinitely. When the specified id is received, a message is given to output stating that the id has been received and Task2 is running (see Figure 9).



```
Task1 running .....
Received: DONE ..... Task2 running
Task1 running .....
Received: DONE ..... Task2 running
Task1 running .....
Received: DONE ..... Task2 running
Task1 running .....
Received: DONE ..... Task2 running
Task1 running .....
Received: DONE ..... Task2 running
Task1 running .....
Received: DONE ..... Task2 running
Task1 running .....
Received: DONE ..... Task2 running
Task1 running .....
Received: DONE ..... Task2 running
Task1 running .....
Received: DONE ..... Task2 running
Task1 running .....
Received: DONE ..... Task2 running
```

Figure 9: Output that is created through Task 1 and Task 2

4.3 Mutex Tasks

4.3.1 StartTask1

The syntax for mutex function calls are analogous to semaphore function calls. Given a mutex id, and a time measured in milliseconds, 'osMutexWait' waits until a mutex object becomes available, then a message is sent to output [2]. Finally, the token of the unique identifier has been released via 'osMutexRelease' and the system is delayed for 2000 milliseconds before the process is repeated indefinitely.

```
void StartTask1(void const * argument)
{
    while(1)
    {
        osMutexWait(myMutex01Handle, 1000);
        myprintf ("Task1 ..... Mutex Print\n\r");
        osMutexRelease(myMutex01Handle);
        osDelay(2000);
    }
}

/* StartTask2 function */
void StartTask2(void const * argument)
{
    while(1)
    {
        osMutexWait(myMutex01Handle, 1000);
        myprintf ("..... Task2 Mutex Print\n\r");
        osMutexRelease(myMutex01Handle);
        osDelay(2000);
    }
}
```

Figure 10: Mutex Tasks

4.3.2 StartTask2

StartTask2 is identical to StartTask1, with the exception of the content of the message being printed. The output displayed by the interaction of the mutex tasks is displayed in Figure 11.



```
Task1 ..... Mutex Print
..... Task2 Mutex Print
Task1 ..... Mutex Print
..... Task2 Mutex Print
Task1 ..... Mutex Print
..... Task2 Mutex Print
Task1 ..... Mutex Print
..... Task2 Mutex Print
Task1 ..... Mutex Print
..... Task2 Mutex Print
Task1 ..... Mutex Print
..... Task2 Mutex Print
Task1 ..... Mutex Print
..... Task2 Mutex Print
Task1 ..... Mutex Print
..... Task2 Mutex Print
Task1 ..... Mutex Print
..... Task2 Mutex Print
Task1 ..... Mutex Print
..... Task2 Mutex Print
```

Figure 11: The output that is a product of the interaction of the Mutex StartTask1 and StartTask2

5 Conclusion

Priority inversion and achieving mutual exclusion are common challenges that arise when developing environments which benefit from RTOS. The main purpose of experimentation was to gain familiarity and fluency in FreeRTOS as well as uc/OS-III. This was accomplished via expanding on provided working examples as well as developing a producer/consumer task. All aforementioned FreeRTOS programs were developed by utilizing binary semaphores in conjunction with queues in order to develop effective task priority via the use of TrueStudio and the STM32f4. Development of larger, more complex systems via FreeRTOS is left for future experimentation.

References

- [1] Giorgio C Buttazzo. *Hard real-time computing systems*. 3rd ed. Springer, 2011.
- [2] *CMSIS RTOS Documentation: Real-Time Operating System: API and RTX Reference Implementation*. 2020.
- [3] Radu Muresan and Kevin Dong. *ENGG4420: REAL-TIME SYSTEMS DESIGN - LAB MANUAL*. 2020.