1 FreeRTOS Semaphores and Queues

Many conflicts can occur when using a shared variable or memory areas. Imagine a low priority task clearing a memory area (writing all zeros) that act as a display buffer.

Halfway through the task become preempted by a medium priority task which write bitmap data to the same buffer. When the medium priority task is done the low priority task continue its clearing. Now the first half of the display buffer contain bitmap data and the second half is all zeros.

A third high priority task might output the buffer to a display and the result will not be quite what you expect.

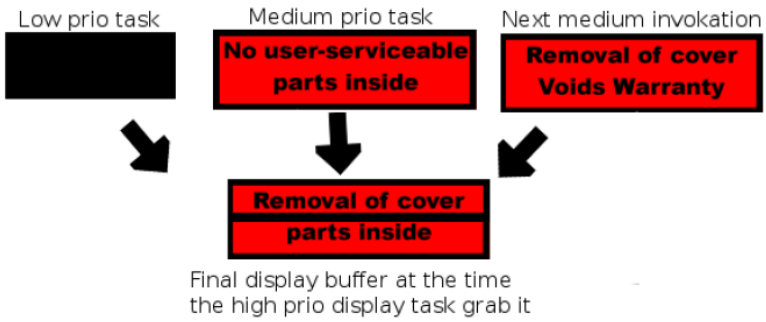


Figure 1: Imaginary Display Buffer

1.1 Semaphores and Critical Sections

To avoid these problems, synchronization is necessary. Semaphores and *"critical sections"* are used for this purpose. Semaphores are a mechanism of telling others that a specific variable or memory area is used by someone else. Critical sections are used to prevent interrupts and preemption, thus making operations in the critical section atomic.

We will look at two different types of semaphores in FreeRTOS:

1. Binary semaphore.
2. Mutex with Priority Inheritance Protocol (PIP).

Binary semaphores and mutexes are similar but have some subtle differences: In FreeRTOS mutexes include a PIP mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronization between tasks and mutexes the better choice for implementing simple mutual exclusion of shared resources.

When using semaphores, the header file *semphr.h* must be included.

1.2 Binary Semaphore

In FreeRTOS semaphores must be created at runtime before they can be used. This is done by the function *vSemaphoreCreateBinary()*:

vSemaphoreCreateBinary(xSemaphoreHandle xSemaphore);

Parameters:

- *xSemaphore* - Handle to the semaphore. It is set to NULL if an error occurred.

1.3 Mutex with PIP

Mutexes in FreeRTOS uses PIP which assumes that only one mutex will be held at a time. The priority of the holding task will be raised to the highest priority of the task attempting access to the same mutex. The priority will be dropped back to the base priority when either mutex is returned. This could result in some priority inversion, but this won't necessarily cause you a problem, and in fact some inversion must have already existed for the inheritance to have been triggered in the first place.

xSemaphoreHandle xSemaphoreCreateMutex(void);

Example:

```
#include "semphr.h"
// The semaphore handle
    xSemaphoreHandle xSemaphore = NULL;
void vTask (void * pvParameters)
{
    // Mutex semaphores cannot be used before a call to
    // xSemaphoreCreateMutex (). The created mutex is returned.
        xSemaphore = xSemaphoreCreateMutex ();
        if( xSemaphore != NULL ) {
            // The semaphore was created successfully.
            ...
            }
}
```

1.4 How to Take a Semaphore

Before accessing a protected variable or memory area the adhering semaphore need to be taken. There's no difference in taking a semaphore or mutex, the same function is used:

xSemaphoreTake(xSemaphoreHandle xSemaphore, portTickType xBlockTime);

Parameters:

- *xSemaphore* - The semaphore handle - which semaphore to take.
- *xBlockTime* - The number of ticks to wait for the semaphore to become available. A block time of zero can be used to poll the semaphore and give up trying immediately if it is busy.

Returns:

- *pdTRUE* if the semaphore was obtained.
- *pdFALSE* if *xBlockTime* expired without the semaphore becoming available.

1.5 How to Give a Semaphore Back

Semaphores must be given back in order to let other tasks use the semaphore. There's no difference in giving back a semaphore or mutex, the same function is used.

xSemaphoreGive(xSemaphoreHandle xSemaphore);

Parameters:

- *xSemaphore* - The semaphore handle.

Returns:

- *pdTRUE* if the semaphore was successfully given back. Otherwise *pdFALSE* is returned.

Example:

```
#include "semphr.h"

xSemaphoreHandle xSemaphore = NULL;
void vTask (void * pvParameters)
{
}
// Create a semaphore
    vSemaphoreCreateBinary (xSemaphore);
// Check if the semaphore creation was successfully
    if (xSemaphore != NULL) {
        ...
        while (1) {
            ...
           // Try to take the semaphore
           while (xSemaphoreTake (xSemaphore, (portTickType) 0) != pdPASS);
           ...
           }
        }
    void vAnotherTask (void * pvParameters)
    {
        ...
        xSemaphoreGive (xSemaphore);
        ...
        }
```

A binary semaphore need not be given back once obtained, so task synchronization can be implemented by one task/interrupt continuously giving the semaphore while another continuously takes the semaphore.

Actually, if you create a semaphore to protect a shared resource there is nothing to prevent the coder from using the shared resource without using the semaphore. But it is considered a logic error to do so.

2 Messages and Queues

Communication between tasks can be done through messages. In FreeRTOS a queue can be created and used for passing messages. A message is typically a struct containing some data, e.g. id, data and timestamp:

Example:

```
struct mesg {
    int id;
    int data [DATALEN];
    portTickType timestamp;
    } ;
```

2.1 Create the Queue

A queue needs to be created before it can be used. This is done with the *xQueue-Create()* function:

```
xQueueHandle xQueueCreate(unsigned portBASE_TYPE uxQueueLength,
                          unsigned portBASE_TYPE uxItemSize);
```

2.2 Send Messages to the Queue

There are several FreeRTOS functions to send messages:

- *xQueueSendToBack()*
- *xQueueSendToFront()*
- *xQueueSendToBackFromISR()*
- *xQueueSendToFrontFromISR()*

Each function adds a message to the queue, given that the queue is not already full. The two last functions are used when the call is made from an interrupt service routine.

2.3 Read Messages from the Queue

There are a few FreeRTOS functions to read messages from a queue:

- *xQueueReceive()*
- *xQueueReceiveFromISR()*
- *xQueuePeek()*

Each function removes a message from the queue, given that the queue is not empty. The *xQueuePeek()* function read the message but doesn't remove it from the queue. Peeking cannot be done from an interrupt service routine.

3 Producer/Consumer

The Producer-Consumer is a classical synchronization problem that we will investigate. The Producer produces bytes and put them in a buffer. If the buffer become full, the Producer stop and goes to sleep for a while. As soon as the Producer start producing, it wakes up a Consumer to come visit every now and then. As long as there are any bytes left in the buffer the Consumer removes them. If there are no bytes in the buffer the Consumer try to wake the Producer and then let itself go to sleep for a while.

See the pseudocode below.

```
Pseudocode
Producer - Consumer
==================
int byteCount
task producer()
{
    byte = produceByte()
    if (byteCount == BUFFER_SIZE)
        sleep()
        putByteIntoBuffer()
        byteCount = byteCount + 1
        if (byteCount == 1)
            wakeup(consumer)
}

task consumer()
{
    if (byteCount == 0)
        sleep()
```

```
        byte = removeByteFromBuffer()
        byteCount = byteCount - 1
        if (byteCount == BUFFER_SIZE - 1)
            wakeup(producer)
            consumeByte(byte)
}
```

4 Assignments (2 bonus points if completed together with the report before the deadline, which is specified in Canvas for each assignment)

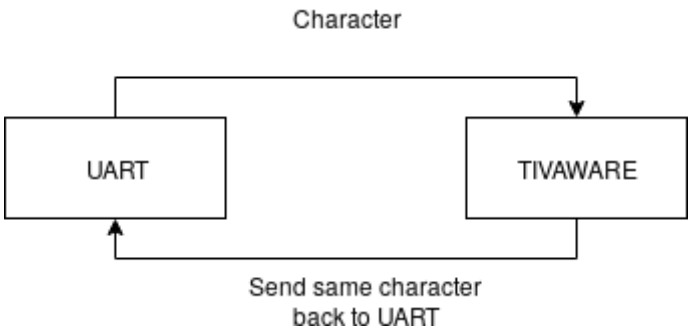
Important Tip: Read the study material provided in Lecture 9.

1. **Producer/Consumer problem:**
   (a) Implement the Producer/Consumer problem in FreeRTOS with a preemptive scheduler and run the code. Visualize in any way you like {LCD, UART, memory dumps}. You are NOT allowed to use queues in the producer-consumer assignment. Can you find any potential problem(s) with your code? How can the problem be solved? The problem should be demonstrated before solving that.
   (b) Use counting semaphores to correct the problem(s) you have encountered. Also correct the problems that you have not encountered but can predict, i.e., what could potentially happen if you have more than one producer and more than one consumer? Note – for an acceptable solution, you are expected to use **both** counting semaphores and binary semaphores, the most simple approach is to use two counting and one binary semaphore .

   Hint: Make two separate projects for a) and b)

2. **UART echo program**

   Write a program using a preemptive scheduler that gets text from the UART and processes the data on the chip and echoes it back to the UART, see figure.



   a) Your output should be 15 characters at a time. Consider the following text consisting of 26 characters as an example: **“Embedded systems is so fun”**.

   The visible text on line 1 on the UART should only be the last 15 characters of the above text, i.e., **“stems is so fun”**

If any key, at any point is pressed, the corresponding character is appended to the text and only last 15 characters of the text are shown. For example, if you press “i”, the output on the serial console program should immediately change from
“stems is so fun”
to:
“tems is so funi”.
b) Consider the task in (a). There should also be a "status task", activated by a push button, displaying how many characters have been received in total. This status text should be on the second row of the serial program you are using and be active for 10 seconds after a button press. If characters are received during status show, the count of characters received should also be updated. The count needs to be updated automatically and should not require an extra button press. To figure out how to format the serial output from the board to your program, check out ASCII escape characters (there are specific characters you can send to clear screen and move your cursor)
The output during button press should look as follows:
tems is so funi
27
If you press “i” five times more, the output should immediately be updated to:
is so funiiiiii
32

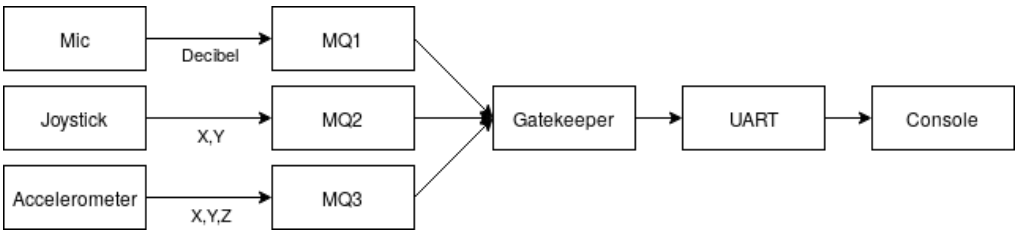
– Assuming that the status task is still active. When the status task becomes inactive, the count variable should disappear.

3. **Sensors and Queues**
   Write a program using FreeRTOS that display values from the sensors (accelerometer, microphone and joystick) on serial port.
   Hint – in order to access the sensors of the booster pack, you should use the analog to digital converter (**ADC**) unit of the TM4C129 chip (you are encouraged to use the functions defined in "driverlib/adc.h"). To get a starting hint on how to use the ADC for our desired peripherals, please consider the following steps:
   1. Enable the ADC using **SysCtlPeripheralEnable**
   2. Loop until ADC is ready using **SysCtlPeripheralReady != 0**
   3. Enable GPIOE port
      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE**);**
   4. Configure the pin that you are after as an ADC function, e.g., for joystick x-values the function looks as follows:
      GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_4)

   The above four steps enables the ADC and configures the GPIO pin as an ADC type, which means you will read analog values from the pin and use the ADC module to convert these values into digital machine readable values.

   a) Your own contribution will be to enable the sample sequencer so that you can read values out of the ADC. Our tested solution used 3 as a sequence number parameter, which you can use as a reference.

b) Implement sensor specific message queues. Each message queue will handle the data given by one sensor only. E.g., messagequeue_1 will contain all accelerometer x,y,z data, messagequeue_2 will contain all joystick x,y data, and messagequeue_3 will contain microphone data. Furthermore, the data should be sent out to the UART in a synchronized order using a gatekeeper task. Refer the figure below:



The output should be presented one line per sensor, so an example output should look as follow:

Microphone: 80db
Joystick: 150, 200
Accelerometer, 80, 50, 20

Note: The value of the microphone should be in (decibel) db

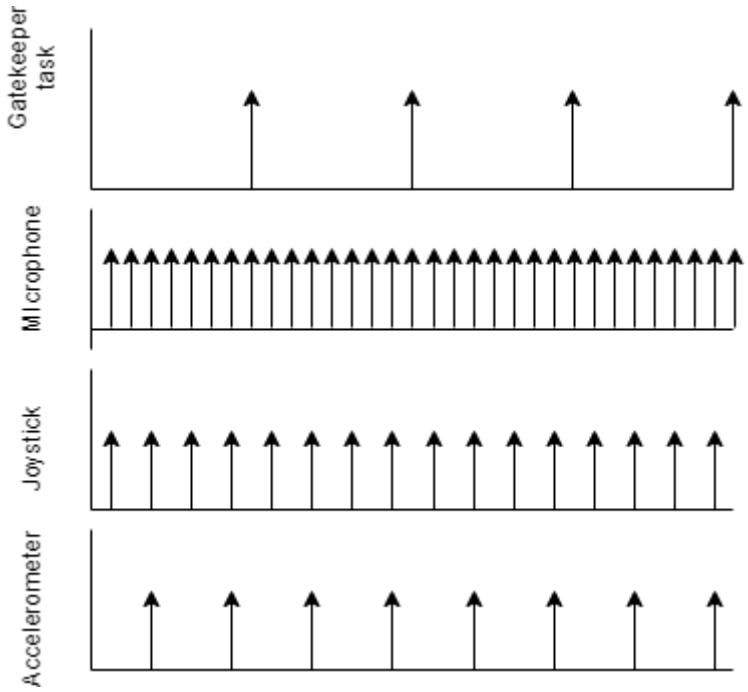
Please keep the output clean, do not continuously print out values, instead print – clear screen – print next value.

c) Lets imagine that one single sensor read is not enough to make an accurate reading of the sensor value. Firstly, there is a risk of debouncing and you might fetch an extra value. Secondly, there is a risk that the sensor value is too jittery. To counter this, you will have to make average readings on each sensor. Follow the specifications in the table below:

Sensor	Required average readings
Microphone	8
Joystick	4
Accelerometer	2

This means, to make an accurate reading of the microphone, you will calculate the average of 8 values for microphone, 4 values for the joystick and 2 values for the accelerometer. The best solution is to let the gatekeeper calculate the average once it has received the feasible amount of values.

Your last and final task will be to create a complete schedule of the three different sensor tasks. An example of a complete system with release times is described in the following graphs:



The easiest way you implement this is to use harmonic periods, i.e., one integer multiple of the shortest period. An example in this case could be 5ms for one sensor task, 10ms for another sensor task and 20ms for the final task. Finally, adjust the gatekeeper periodicity so that it can fetch the required amount of values from each respective queue to perform the average calculations.

The gatekeeper task should **NOT** calculate the moving average, but must calculate the current average. This means, once the average calculation of the queues has been done, the gatekeeper cannot use these values for average calculations anymore.

5 Report

Each group must have a report and it should include the following.

1. Identify and describe the problems you found in the Producer/Consumer assignment without using semaphores.
2. Describe how you fixed the Producer/Consumer problems with semaphores.
3. Describe and motivate your solution for assignment 3. Moreover, present the experiment you have done using your solution.

6 Optional Assignments (4 bonus points if completed before the deadline, which is specified in Canvas for each assignment)

1. The Dining philosopher's problem is a classic example. Five philosophers are gathered around a table with a big plate of rice in the middle. Between each philosopher is a single chopstick. A philosopher needs two chopsticks to eat the rice. The life of a philosopher consists of alternate periods of thinking and eating. When a philosopher wants to eat, she tries to acquire the chopsticks next to her. If successful in acquiring two chopsticks, she eats for a while, then puts down the chopsticks and continues to think. The key issue is that a finite set of tasks (philosophers) are sharing a finite set of resources (chopsticks), and each resource can be used by only one task at a time. Implement the Dining philosopher's problem on the EK-TM4C129EXL. **(2 bonus points)**

2. Write a program with a random number of producers and a random number of consumers. All sharing the same buffer. The assignment must work according to your previous optional assignment, i.e., it is not sufficient to implement a random number of producers and consumers without mutexes and/or semaphores. You must furthermore write a report which explains newly introduced problems when introducing random number of producers and consumers, according to the report template in chapter 5. **(2 bonus points)**

Note: The bonus points collected after successfully completing the compulsory assignments and optional assignments before the corresponding deadlines will be added to your score in the final exam. These bonus points could help you in improving your final grade in the course.