# Department of Engineering

# EE 4710 Lab 6

Title: Semaphore Implementation

Objective: The student should understand how semaphores work and affect the scheduling of tasks in a real-time system.

Parts: 1-C8051FX20-TB Evaluation Board
1-USB Debug Adapter
1-DB-9 Serial cable (USB adapter cable is also ok)

Software: Silicon Laboratories IDE version 3.50.00 or greater. Keil compiler.

Procedure: Write the title and a short description of this lab in your lab book. Make sure the page is numbered and make an entry in the table of contents for this lab.

Starting with your code from lab 4, decide as a team how to modify your code to support semaphores. At a minimum, you will need to provide two functions, one to take a specified semaphore and one to release it. You may also want to write a function to create or initialize the semaphore.

Here are some issues you may wish to consider:

1. When a task fails to take a semaphore, it is suspended or blocked until the semaphore is free. How does the scheduler know that a task is blocked?

2. Releasing a semaphore might involve waking up another task. How is the best way to find the highest priority task that is blocked by a given semaphore?

3. While we're on the subject of priority, should task priorities be fixed or variable? The easiest thing to do is to assign fixed priority based on a task's position in the taskdesc[] array, but that will make inheritance protocols difficult. Which direction do you want to go?

Step 1. Once you have your plan, you will doubtless need to modify the task descriptor in context.h to include additional fields. IT IS IMPORTANT that you also modify the constant taskdesc_size in context.asm to match the size of your new task descriptor. After making these changes, verify that your code originally from lab 4 still works.

Step 2. Write a function to initialize the new fields of the all the task descriptors. (Each task descriptor should be initialized in a blocked or suspended state so it will not be scheduled until its task has been "created".) Name the function something like rtos_init() and have it call context_init(). In main() call rtos_init() instead of context_init(). Compile, run and verify that your code still works.

Step 3. Modify your code in create_task() so that it sets the task's descriptor to a "run" state (so it can be scheduled). Change the scheduler so that any task in the range 1..NUM_TASKS-1 that has the highest priority and is ready (not delayed nor waiting for a semaphore) is scheduled. If no such task exists, schedule task 0. Verify that your code still works.

Step 4. Write a function that takes a semaphore. The code should do the following:

(a) if the semaphore is non-zero, decrement it and return.
(b) otherwise, modify the task descriptor to suspend the task
(c) yield the processor
(d) return (assuming that when it wakes up, it has the semaphore)

You will need to protect some or all of this procedure by disabling interrupts. Once that code compiles, write code to release the semaphore, which should do the following:

(a) increment the semaphore
(b) if the semaphore is not equal to 1, return
(c) otherwise, seek the highest priority task that is waiting for this semaphore.
(d) if such a task is found, alter its task descriptor so it is no longer suspended, then set the semaphore to 0
(e) if such a task is found and it has a higher priority than the current task, yield the processor

Again, you will need to protect some or all of this procedure by disabling interrupts.

If your design calls for a function to create/initialize semaphores, write that function now.

Using lab 5 as a guide, write a high-priority task that uses semaphores to release a medium priority task every 500ms and a low priority task 10ms thereafter. The medium priority task should print the phrase "No, you can't\r\n" and the low priority task should

print the words “Yes, I can\r\n” to the serial port.

Note: you should be able to use a common put_string() function:

```
void put_string(char code *s)
{
    // eventually, you will add code to take a semaphore here
    while ( *s )
    {
        SBUF0 = *s;
        while ( TI0 == 0 ) { }
        TI0 = 0;
        s++;
    }
    // eventually, you will add code to release the semaphore here
}
```

Compile and run your code, then verify it works.

Step 5. Exchange the priorities of the low and medium priority tasks. Run your code again and verify it fails. This is because both tasks are executing code in put_string() at the same time and the low priority task gets stuck forever waiting for TI0 to be set.. To guard against this problem, create a new semaphore that is initialized to 1, then add code in put_string() to take and release the semaphore surrounding access to the serial port (see comments in put_string(), above).

Compile and run your code, then verify it works. Demonstrate your code to the lab instructor.

Affix all your source code to your lab book then write a summary or conclusion. Remember to sign or initial then date each page.