San José State University
Department of Computer Engineering
CMPE 146-03, Real-Time Embedded System Co-Design, Fall 2020

Lab Assignment 8

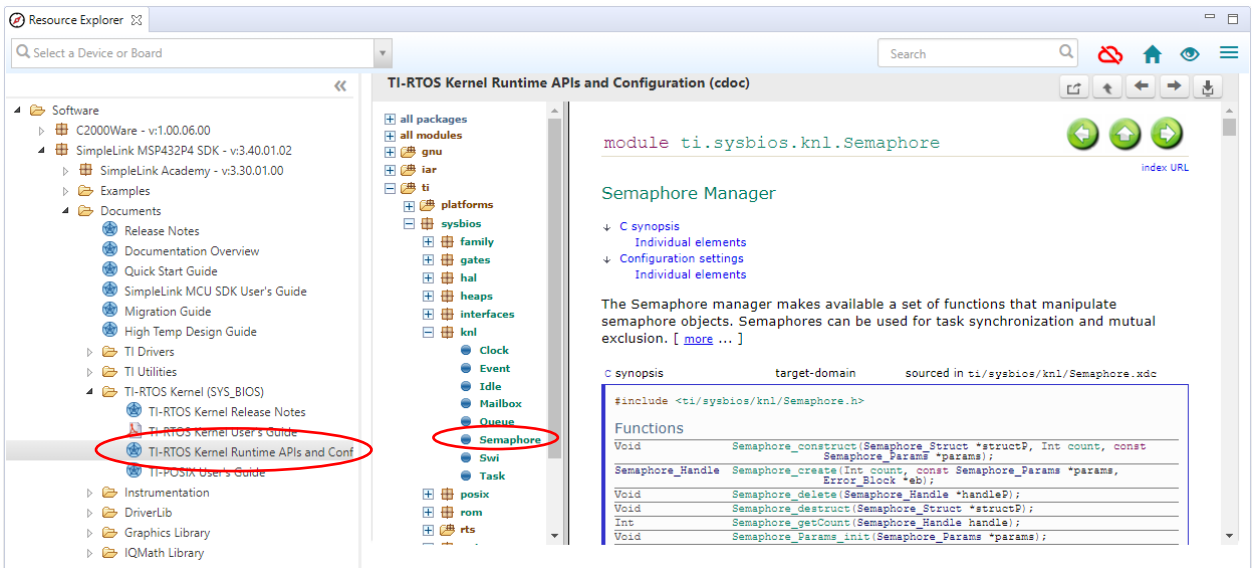
Due date: 12/07/2020, Monday

1. Description

In this assignment, we will be familiarized with the semaphore system object in the TI's real-time operating system, TI-RTOS. We will create tasks to control LEDs using semaphore.

In an embedded system, tasks rarely work all by themselves. There is usually some dependence among the tasks in terms of the flow of operations. We can use semaphores to control the flow to ensure the correct system behaviors.

The semaphore API documentation can be found on the Resource Explorer as shown below. Click on the function name to get more details.



2. Exercise 1

In this exercise, we are going to create two tasks to control one LED. We use one task to create the control timing and the other task to actually turn the LED on or off.

We will import a "skeleton" project, /Software/SimpleLink MSP432P4 SDK…/Examples/Development Tools/MSP432P401R LaunchPad – Red 2.x (Red)/TI-RTOS Kernel (SYS_BIOS)/hello/TI-RTOS/CCS Compiler/hello, and modify the main program file *hello.c* for this exercise. After importing the project, rename the project and main file to something meaningful. Use the C file, *Lab8_template.c*, on Canvas that

comes with this PDF. Replace the contents of the main file of the project with this C file's contents. This template program basically creates two small tasks to print two short messages to the debug console in a particular order.

In the template file, *Lab8_ template.c*, we declare two global data objects as follows.

```
28  Semaphore_Struct semaStruct;
29  Semaphore_Handle semaphoreHandle = NULL;
```

The *semaStruct* is the data structure with which the RTOS manages the semaphore. The semaphore is referenced with a handle (essentially a reference ID), *semaphoreHandle*. Whenever a task wants to do something with the semaphore, it will use the handle as the reference.

In *main()*, we use the following sequence of instructions to create a semaphore.

```
52      Semaphore_Params semaParams;
53      Semaphore_Params_init(&semaParams);
54      Semaphore_construct(&semaStruct, 0, &semaParams);
55      semaphoreHandle = Semaphore_handle(&semaStruct);
```

Line 52 declares a semaphore parameter data structure, *semaParams*.
Line 53 initializes the parameter data structure to contain the default values. By default, it is a counting semaphore.
Line 54 creates the semaphore with the input parameter data structure. Additionally, we initialize the semaphore's *count* value to be 0, i.e., the semaphore is initially not available to any tasks.
Line 55 gets the handle of the semaphore for future referencing.

Each task basically prints a message to the debug console. The order of printing is controlled by the usage of the semaphore.

```
61  Void task1(UArg arg0, UArg arg1)
62  {
63      Semaphore_pend(semaphoreHandle, BIOS_WAIT_FOREVER);
64      printf("Task1\n");
65  }
66
67  Void task2(UArg arg0, UArg arg1)
68  {
69      printf("Task2\n");
70      Semaphore_post(semaphoreHandle);
71  }
```

We are going to print the message from Task 2 first. So, upon entering the task function, Task 1 will try to get the semaphore first by calling *Semaphore_pend()* (equivalent to POSIX's *sema_wait()* we discussed in class); without the semaphore, it won't proceed forward. When the semaphore is not available (its initial condition), the OS will suspend the task. On the other hand, Task 2 prints its message immediately upon entering the task function. Then it calls *Semaphore_post()* (equivalent to POSIX's *sema_post()*) to increase the semaphore's *count* value so that the semaphore will become available for Task 1. When the semaphore becomes available, the OS resumes execution of Task 1. Then Task 1 proceeds to print its message.

Build and run the template program. We should see two messages on the debug console with Task 2's message going first. If you comment out the *Semaphore_pend()* call in *task1()*, you should see that Task 1's

message is printed first. That is because Task 1 is created first in *main()*, so, by default, it will be executed first.

Modify the task functions (do not change anything else) so that one task, Task 1, controls the frequency of blinking an LED at **0.4 Hz** and Task 2 turns on or off the **red LED** (LED1 on the LaunchPad). So, we divide the timing and control functionality into different tasks.

Task 1 will increase the semaphore's count value, by calling *Semaphore_post()*, when the LED needs to change state. The amount of time the LED remains in a certain state can be set by calling *Task_sleep()* with the proper argument. Similar to previous labs, the timing control is done in an infinite loop in the task function; do not use your own or other people's timing or delay function; do not use any hardware timers for the timing purpose. The task function shall have no control or knowledge of the actual device it is influencing. All it does is "sending out a signal" at the proper time, which is the principal function of a semaphore.

Task 2 sets up the GPIO port to drive the LED at the beginning. Then it enters an infinite control loop. It tries to get the semaphore by calling *Semaphore_pend()*. Remember, if the semaphore is not available (its *count* is ≤ 0), the task has nothing to do, and will be suspended by the OS. When the call returns, i.e., the semaphore is available and the OS resumes the task's execution, it shall change the LED's state. After finishing the work, it calls *Semaphore_pend()* again to try to get the semaphore. The task function shall have no knowledge of timing of any kind. It shall use DriverLib functions to control the LED. Essentially, Task 2 spends almost all of its time in *Semaphore_pend()*, waiting for an event to happen without wasting any processor time.

Lab Report Submission

1. List the task functions.
2. Explain how you control the blinking.
3. Provide the program listing in the appendix.

3. Exercise 2

In this exercise, we are going to create multiple tasks with the same task function. That way, we can use exactly the same algorithm for controlling multiple devices of the same type without duplicating the code. When we create a task, we will provide parameters to the new task so that it can control the right device.

Duplicate the project created in the previous exercise. Add a constant *NTASKS* to the main program file to indicate how many tasks we are going to create. Declare two arrays *taskStructs* and *taskStacks* for the task data structures and stacks to replace the original declarations. You can use something like the followings.

```
#define NTASKS 3
Task_Struct taskStructs[NTASKS];
Char taskStacks[NTASKS][TASKSTACKSIZE];
```

The data structure *Task_Params* provides two fields for passing "arguments" to the new task to be created. You can check the task-creation API on how to use them.

For this exercise, create three tasks using the same task function that prints the arguments continually. When creating the tasks, provide different arguments to the parameter data structure so that each task can be easily identified. You should remove the original task creations (in *main()*) and functions from the previous

exercise, but keep the semaphore part for the next exercise. You can use the common task function shown below.

```
Void taski(UArg arg0, UArg arg1)
{
    while(1)
    {
        printf("Task: arg0=%u, arg1=%u\n", (uint32_t)arg0, (uint32_t)arg1);
        Task_sleep(1000);
    }
}
```

Lab Report Submission

1. List the relevant code to create the tasks.
2. Show a screenshot of the console outputs.
3. Provide the program listing in the appendix.

4. Exercise 3

In this exercise, we are going to blink the red LED (LED1 on the LaunchPad), green and blue LEDs (two parts of LED2 on the LaunchPad) in synchronization, with a dedicated task for each LED. Basically, the LEDs are turned on and off at the same time, visually, from three different tasks.

Duplicate and expand the project created in the previous exercise. Create a new task in *main()*, like Task 1 in Exercise 1, to control the timing of blinking with one single counting semaphore. Do not use more than one semaphore. Make use of the count attribute of the single semaphore to control all three LEDs. The ON and OFF times within a period must be the same (50% duty cycle). Same as in Exercise 1, the availability of the semaphore indicates a change of LED state.

Again, the timing task shall provide all the necessary timing. Make the blinking frequency change with time. In each iteration of the timing control loop, it shall use a frequency that is randomly generated. Limit the frequencies to the range of 0.2 Hz to 1 Hz. You can use the standard C library function, *rand()*, to get a random number for generating the necessary timing parameter. Also, you'll need to do something extra with the semaphore in order to be able to provide the proper timing for all three LEDs.

You shall use the common task function to control the three different LEDs. When creating an LED control task, pass the port number and pin number as arguments to the new task. For example, for the red LED, they will be *GPIO_PORT_P1* and *GPIO_PIN0*. In the LED control task, as before, set up the output pin properly and initialize it to a known state, before entering the forever control loop. The LED shall change state when the semaphore is acquired.

Here is one piece of information that you may need to use. Under a multitasking OS, a task would keep on executing, using as much CPU time as possible, until it is suspended by the OS. However, in some situations, a task should voluntarily suspend itself, giving up its CPU time, so that other tasks can run. To suspend itself, a task can call the *Task_yield()* function. If there is a higher-priority or an equal-priority task in the ready queue, the OS will switch to it. Otherwise, the function returns immediately and the task proceeds to execute the next instruction.

Note that since the green and blue LEDs are very close to each other on the board, when they are both on at the same time, the color actually looks bluish white.

Lab Report Submission

1. List the timing task function.
2. List the common LED control task function.
3. Explain briefly how you make sure all three LEDs blink in synchronization.
4. Provide the program listing in the appendix.

5. Submission Requirements

Write a formal report that contains at least what are required to be submitted in the exercises. Submit the report in PDF to Canvas by the deadline. Include source code in your report. As for the report contents, do not use screenshots to show your codes. In the report body, list the relevant codes or screenshots only for the exercise you are discussing. Put the entire program listing in the appendix. Use single-column format.

6. Grading

Lab 8 Rubric

Criteria	Ratings				Pts
Exercise 1	**3 pts** 3 Correct implementation. Program listing in appendix.	**1 pts** 1 Listed task functions. Provided explanation of control.	**0 pts** 0 Not attempted or reported.		3 pts
Exercise 2	**3 pts** 3 Correct implementation. Program listing in appendix.	**2 pts** 2 Listed relevant code to create multiple tasks. Provided correct console outputs.	**0 pts** 0 Not attempted or reported.		3 pts
Exercise 3	**4 pts** 4 Correct implementation. Program listing in appendix.	**2 pts** 2 Listed LED control task. Provided explanation.	**1 pts** 1 Listed timing task. Provided explanation.	**0 pts** 0 Not attempted or reported.	4 pts
				Total Points: 10	

Grading Policy

The lab's grade is determined by the report. If you miss the submission deadline for the report, you will get zero point.