Department of Electrical and Computer Engineering

ECE 5710/6710 Lab 4

Title: Round Robin Scheduling

Objective: The student should be able to implement a round-robin pre-emptive scheduler.

Parts: 1-EFM32GG Evaluation Board

Software: Silicon Laboratories Simplicity Studio.

Preparation: Write the title and a short description of this laboratory exercise in your lab book. Make sure the page is numbered and make a corresponding entry in the table of contents.

You are to write a preemptive round-robin scheduler to generate a feasible schedule for the following four tasks:

Task	Period	Execution Time (e_i)	Relative Deadline (D_i)
A	3+ ms	0.4 ms	3 ms
B	7+ ms	1.2 ms	7 ms
C	10+ ms	1.9 ms	10 ms
D	30+ ms	5.0 ms	30 ms

Tasks have some inter-release jitter, so minimum periods are listed.

Determine a suitable interrupt rate so that all tasks meet the criterion below.

$$\left(\left\lceil\frac{e_i}{\tau}\right\rceil(n-1)+1\right)\tau + e_i \leq D_i \quad \forall i$$

where *n* is the number of tasks and τ is the interrupt period.

Document your calculations and the interrupt rate you select in your lab book.

Create a new (empty C) project using Simplicity Studio and rename the project to something suitable, like Lab4.

You will be configuring the SysTick timer as you did in Lab 3, but instead of writing your own SysTick_Handler(), download the assembly file context.s from the Lab 4 page in canvas and add it to your project. The SysTick handler in this file assumes that the

variable CurrentTask points to a task control block and that the first word of that block contains the stack pointer for a suspended task. Whenever a timer interrupt occurs, SysTick_Handler saves all the registers on the stack then saves the stack pointer into the current task's control block. It then calls scheduler() (which you will write) to select the next task to run.

Creating tasks under this paradigm is a little tricky. One must allocate a stack for the new task and initially set up the stack frame. It is recommended that you download lab4_skeleton.c as a starting point. This code contains a function called CreateTask() that will take care of those pesky details. In fact it should run except that the scheduler needs to be fleshed out first. For now, simply write code in the body of the scheduler that alternately returns a pointer to the second and third elements of the task control block, or TCB. (That would be TCB+1 and TCB+2, or for those who have a hard time with pointer arithmetic, &TCB[1] and &TCB[2].) Build your code and see to it there are no errors.

Procedure: Launch the debugger to download your code. Set breakpoints in the while loops of foo() and bar(), then run your code and verify one of the breakpoints is hit. (Which breakpoint you hit first depends on how you wrote your scheduler.) Remove the breakpoint that was hit and resume execution. Verify the other breakpoint is hit.

Go to the Lab 4 assignment page in canvas and download tasks4.o. Move it into your project folder. Right-click on your project name and select "Properties". Under "C/C++ Build" select Settings. Then under Tool Settings, select Miscellaneous from the GNU ARM Linker. Add "${ProjDirPath}/tasks4.o" to the "Other objects" box and click both "Apply" and "Ok".

You will now need to go back into your main program and add declarations for task_A() thru task_D() as you did in lab 3. You will also need to add the declarations below:

```
int task_A_released(void); // returns true when Task A is released
int task_B_released(void); // returns true when Task B is released
etc…
```

In most real-time systems, tasks are implemented by functions that never return. Modify foo() or write a new function as shown on the next page to handle task A:

```
void Task_A_Loop(void)
{
    while(1)
    {
        while ( ! task_A_released() ) { }
        task_A();
    }
}
```

Write a similar function for each of the other three tasks. Now (a) change NUM_TASKS to 5, (b) call CreateTask in main() for each of the functions you wrote, (c) configure the SysTick_Timer using the interrupt rate you chose when you prepared for this lab, and (d) change your scheduler so it schedules each of the tasks in a round-robin fashion. Don't forget to allocate a separate stack for each task.

Build and download your code. Run it and verify both amber LEDs light up. If not, the debugging tools that you used in Lab 2 are still available (e.g. get_time, early_task, etc.).

The problem with this system is that no time is allotted for the aperiodic or sporadic tasks. This is because each of the task loops wastes processor time waiting for its task to be released. A better approach is to yield the processor to the aperiodic task loop when it is clear a periodic task has no use for it. To do this, we will write a function called Yield() that causes a context switch to the aperiodic task loop and call it when a task is waiting to be released. Yield will simply generate a SVC (service) interrupt whose ISR will do exactly what SysTick_Handler does, except it will always switch to the background task, i.e. TCB[0].

Make a copy of SysTick_Handler in context.s and rename it to SVC_Handler (Hint: you will also need the copy the pseudo instructions starting with .thumb_func). Replace the call to the scheduler with code that loads the address of the TCB (i.e. &TCB[0]) into r0. One way to do this is to use the following instruction in place of the call to the scheduler:

```
    ldr    r0,=TCB
```

Then write a new assembly function called Yield that executes the following instructions:

```
Yield:
    svc   #0   // raise SVC interrupt
    bx    lr   // return from subroutine
```

Add a prototype (external declaration) of Yield() to your main C file and change all task loops to call Yield() when a task is not yet released, e.g.

```
while ( ! task_A_released() )
{
        Yield();
}
```

Build and download your code. Verify that both amber LEDs remain lit and that each time you stop program execution, idle_count is a different number. (Note: if the compiler optimizes away idle_count, you will need to disable compiler optimizations in the project properties.)

When your scheduler works, demonstrate it to your lab instructor. You will ne asked to stop your code several times to demonstrate that idle_count is incrementing.

Print a copy of your code (both files) and affix it into your lab book. Write a short summary or conclusion and submit your lab book to your lab instructor for grading. (Remember to initial and date!)

Points will be assigned according to the rubric below:

Criterion	Points
Lab book properly bound and kept in ink	1
Lab book contains a title and short description	1
Each page initialed and dated, large areas crossed out	1
Pages are legible with no obliterations	1
Lab book contains schedulability calculations	1
Lab book contains scheduler and context switch code	2
Program works	3

Late work is penalized 2 points (20%).