CG2271 Real Time Operating Systems
Term Assignment I

1. Introduction

In this assignment you will be implementing a multitasking kernel for a small RTOS called AVROs. A framework has been provided to you in AVROs.zip, containing:

i. kernel.c: Most of your scheduling stuff goes in here. It also includes skeleton code for routines to create tasks, create atomic sections, etc.
ii. kernel.h: Consists of prototypes for the routines in kernel.c, as well as customization parameters like maximum number of processes, default stack size,
iii. app.c: contains the application that you will write to test AVROs

There are two parts to this assignment. In this first part you will implement a simple time-slicing kernel that can handle an arbitrary number of processes (up to a maximum specified in kernel.h) in a round-robin fashion. In the second part you will implement more advanced scheduling techniques.

2. Deliverables

Both parts, together with the demos, are due in Week 13 during your respective lab sessions. Both part 1 and 2 require reports, and you will submit hardcopies of both reports in Week 13 at your lab session, together with the demo. There is a template report included in the assignment zip file.

Part 2 depends on Part 1 working. Part 2 will be released on 22 October so you will have until then to finish this part.

3. Instructions

We will continue using the same circuit as in Lab 4.

Create a new called "AVROs1" in AVR Studio. This will create an AVROs1.c file, which you can delete from the project. Copy over kernel.c, kernel.h and app.c into the project directory (<workdir>\AVROs1\AVROs1, where <workdir> is the directory that you created the project in), then add kernel.c and app.c to the project (right mouse button over the AVROs1 project name in the solution explorer, then clock Add->Existing Item).

The table below shows what's in the 3 files you've added:

Item Name	Description
kernel.h	
OS_NUM_TASKS	Maximum number of tasks that can be created.
OS_SCHED_TYPE	Type of scheduling algorithm to be used.
Protypes	Prototypes for OSMakeAtomic, OSLeaveAtomic, etc, as defined in kernel.c
kernel.c	
pxCurrentTCB	Contains the stack pointer for the task about to be swapped out using portSAVE_CONTEXT or to be swapped in using portRESTORE_CONTEXT. Refer to lecture to see what this variable does.
portSAVE_CONTEXT	Macro to save the context of the current task.
portRESTORE_CONTEXT	Macro to restore the context of the next task to be run.
TtaskBlock	Task Control Block type definition.
taskTable	Task Control Block, containing information about the tasks. Filled up by OSAddTask. Array of OS_NUM_TASK elements, with OS_NUM_TASK defined in kernel.h.
OSMakeAtomic	Creates an atomic region by disabling interrupts.
OSLeaveAtmoic	Leaves the atomic region.
OSInit	Initializes the operating system.
OSAddTask	Adds in a new task to the Task Control Block.
findNextTask	Applies scheduling algorithm to decide which task to run next. Tasks are numbered from 0-(n-1), where n is the actual number of tasks created through calls to OSAddTask. For the moment implement round-robin scheduling. I.e. with 3 processes, we will run 0->1->2->0->1->2->0->1->…
OSSwapTask	Calling this function causes the currently running task to be swapped out and the next task to be swapped in.
OSRun	Starts the operating system by running the first task.

app.c	
main	Main routine that initializes the OS, creates the tasks and starts them.
task1	Task that generates the buzzing tone. Argument passed in is the base frequency in Hz.
task2	Task that maps the ADC channel 1 readings to an appropriate range for the LED.
task3	Task that maps the ADC channel 0 readings to an appropriate range for the buzzer.
task4	Blinks the LED at a specified rate in Hz.
ISRs	ISRs for the ADC and PWM.

The task allocation for app.c is flexible and you can change it if you like. You can also use fewer or more tasks, but make sure that your design makes sense or you will lose points.

You may add or modify anything in the kernel.h, kernel.c or app.c files, including data structures, etc. However if you modify anything marked "do not modify", make sure you know what you're doing. ;)

PART I

In the first part we will implement a co-operative version of AVROs. This means that every task must call OSSwapTask to switch to another task. So you DO NOT have to implement the ISR for Timer 0 yet! Your tasks in app.c are currently written to call OSSwapTask at the end of the while loops.

Implement all the code in kernel.c, and describe your implementation for kernel.c in your report. To test your kernel, modify app.c to implement task1 to task4 as described in the table above, together with all the necessary ISRs for the ADC and to generate the PWM signal.

PART II

We will now turn AVROs from a cooperative multitasker to a time-slicing multitasker. To do this, modify kernel.c so that OSSwapTask is called every 50 ms. Thus we will swap task 0->1->2->3->0->1->2->3->… every 50 ms.

Since tasks can now be interrupted at unpredictable places resulting in unpredictable performance, you can now use OSMakeAtomic and OSLeaveAtomic to enforce atomic sections. I.e. sections of code that cannot be pre-empted.

Document the changes that you made in your report. You will demonstrate this part of your project during submission.