Assignment 1:
Scheduling and Queues

1DT056: Programming Embedded Systems
Uppsala University

January 26th, 2015

You can achieve a maximum number of 20 points in this assignment. 8 out of the 20 points are required to pass each assignment and you will need to score at least 60 points across all six assignments. Accumulating 80/90/100/110/120 points will result in 1/2/3/4/5 bonus points in the final exam. Assignments are to be solved by students *individually*.

Due to the upload of the solution to the first exercise by mistake, solution to the second exercise will carry the full points for this assignment.

Installation of the development environment

In the exercises and the lab of this course, we are going to use the MDK-ARM developer kit and the μVision IDE for implementing embedded software. Both tools are commercial products by ARM for developing embedded systems on the basis of ARM micro-controllers and other processors. In the scope of the course, we are going to use evaluation versions of the tools that are available free of charge; those versions offer only a restricted feature set compared to the full versions, but are sufficient for our purposes.

μVision is installed centrally on the lab computers in room 1313, in the directory `G:\Programs\Keil`. You can run it from there without any further installation, etc.

If you want to install the software on your own computer, you need to download it from the web page `https://www.keil.com/arm/demo/eval/arm.htm`. The tools are native Windows applications, but can be used without problems on Linux computers with the help of Wine.

To get started with the exercises, download the development project `http://www.it.uu.se/edu/course/homepage/pins/vt15/lab_env.zip`; this project can be used as starting point in all of the following exercise questions. Unpack the archive and open the project `lab_env` using μVision. This project contains definitions and firmware for the STM32F103 processor (an ARM CORTEX M3 controller), as well as the FreeRTOS operating system.

To write, compile, and simulate/debug your own code, you can start by modifying the source file `main.c` of the `lab_env` project.

Further information is available in the following places:

- Description of the STM32F103 processor: `http://www.keil.com/dd/chip/4794.htm`
- Documentation of the libraries provided by MDK-ARM: `http://infocenter.arm.com/help/topic/com.arm.doc.dui0475b/index.html`
- FreeRTOS API: `http://www.freertos.org/a00106.html`
- Books on FreeRTOS (we recommend the Generic Cortex M3 edition): `http://www.freertos.org/Documentation/FreeRTOS-documentation-and-book.html`

Exercise 1 Rate-monotonic scheduling (8p)

Rate-monotonic scheduling is a strategy to choose priorities of tasks in a real-time system with preemptive multi-tasking. Suppose that a system has to execute n periodic tasks, where:

- the instances of task τ_i (for $i \in \{1, \ldots, n\}$) are regularly activated; the time T_i between two activations is called the *period* of τ_i.
- all instances of task τ_i have the worst-case execution time C_i.
- the deadline of each instance of a task τ_i is the point in time when the next instance arrives.
- the tasks do not share any resources, or have any other interdependencies.

In rate-monotonic scheduling, the priorities of tasks are chosen inversely related to the task's period: if $T_1 < T_2$, then the priority of τ_1 will be higher than that of τ_2. If priorities are chosen like this, then preemptive fixed-priority scheduling is guaranteed to meet the deadlines of all tasks if the following inequality holds (the system is *schedulable*):

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \;\leq\; n(\sqrt[n]{2} - 1)$$

The left expression is called the *utilisation factor* U of the set of tasks.

Consider the tasks running in the elevator control system described on the page `http://www.it.uu.se/edu/course/homepage/pins/vt15/elevator`; for sake of simplicity, we ignore commonly used resources of the tasks at this point.

1. Using the result about rate-monotonic scheduling, can we say anything about schedulability of the control system (or parts of it), given the fixed priorities specified on the lab page?
   Reasonable estimates of the worst-case execution times C_i are:
   - Position processor: $\leq 20\mu s$ (per iteration)
   - Button processor: $\leq 50\mu s$
   - Actuator module: $\leq 50\mu s$
   - Safety module: $\leq 30\mu s$
   - Planning module: $\leq 150\mu s$
2. Describe how such execution time estimates can practically be derived through simulation.

Exercise 2 Programming queues (12p)

In this exercise you need to program your own toy version of a queue that can be safely used between multiple tasks in the FreeRTOS. The queue should be dynamically allocated and possess blocking behavior (with infinite waiting time). The goal of the exercise is to implement the following functions:

- *queueHandle createQueue(int elementSize, int queueCapacity)* — Dynamically allocates a new queue of appropriate size using *malloc* and returns a queueHandle (which uniquely identifies it). Creates book-keeping entry for the created queue.
- *int deleteQueue(queueHandle queue)* — Frees the memory allocated for this particular queue. Removes the corresponding book-keeping entry.
- *int enqueue(queueHandle queue, void * content)* — Adds an element to the end of the queue. If the queue is full the task invoking it is blocked until the queue has free space again. The data is copied to the queue byte-by-byte.
- *int dequeue(queueHandle queue, void * buffer)* — Removes the first element of the queue and stores it in the buffer. The data is copied byte-by-byte. It assumes that the buffer has sufficient space to store an element of the queue. If the queue is empty the invoking task should be blocked until there are some elements in the queue.

Since one can use multiple queues at the same time, some kind of book-keeping is necessary. For each queue you need to keep track of its queueHandle, head, tail, elementSize, capacity, queuePointer (this can be wrapped

together with the queueHandle). Work out the details of it for yourself in case that something needs to be added to achieve the requirements of the exercise. It is recommended that you declare a global fixed-size table to store book-keeping information about created queues, but other solutions are allowed.

To demonstrate the use of your queue write a main function which starts a pair of tasks which communicate using the queue you implemented. Your example should demonstrate blocking behavior of the implemented queue.

Hint: *Use semaphores to ensure that the queue implementation meets the requirements. Binary semaphores could be of use to enable concurrent-safe use. Counting semaphores could be of use to implement blocking behavior. Also, keep in mind that a racing condition could occur due to these requirements, it should be prevented. In order to use the semaphores make sure that the following macros are properly defined in the freertosconfig.h:*
#define configUSE_MUTEXES 1
#define configUSE_COUNTING_SEMAPHORES 1

Submission

Solutions to this assignment are to be submitted
via Student Portal at latest on **Friday, February 6th, 2015**.