# Assignment 3: Interrupts, Scheduling, Arithmetic

1DT056: Programming Embedded Systems
Uppsala University

February 2nd, 2012

This assignment is deliberately chosen to be short, in order to provide time for finishing the first lab. The answers for the assignment are due 2 weeks after publication of the assignment (more details on the last page).

## Exercise 1 Interrupts (6p)

Recall the program that you wrote in assignment 1, exercise 5, which was supposed to send seconds and tenths of seconds since controller start-up to the USART 1 device. In this exercise, you are supposed to implement a similar program generating the following sequence of outputs (again, at the correct points in time):

```
0.0000s
0.1001s
0.2002s
0.3003s
0.4004s
```

This means, the program is supposed to send a time stamp to the serial device every $0.1001s$.

1. Implement this program, starting from the project skeleton that is provided in `http://www.it.uu.se/edu/course/homepage/pins/vt11/lab_env.zip`. You are not allow to use the FreeRTOS functions vTaskDelay or vTaskDelayUntil (which would not provide the required resolution anyway). Instead, set up one of the timers of the micro-controller to raise an interrupt every $0.1001s$, and generate the time stamps in the interrupt service routine (ISR).
   Recall that it is not possible (due to FreeRTOS restrictions) to use the function printf in the immediate ISR; you have to use the "deferred interrupt handling" pattern.
2. Discuss (by listing advantages and disadvantages) which of the solutions is preferable: the program developed in assignment 1, exercise 5, or the program written in the present assignment.

## Exercise 2 Rate-monotonic scheduling (4p)

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

For instance, the task described in assignment 1, exercise 5 is a task with period $0.1s$; the worst-case execution time $C_i$ is the maximum time needed in an iteration of the task's main loop.

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 $\tau_1$ will be higher than that of $\tau_2$. 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/vt12/lab`; 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? Note that you have to determine the period of tasks by inspecting the implementation of the elevator control system.
   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 3 Fixed-point arithmetic (10p)

In the lecture on Monday, February 6th, you have seen the definitions for a basic version of signed fixed-point arithmetic, here using a 32-bit significand and four digits after the comma:

```
typedef s32 FIA;          // s32: signed 32bit integers,
                          // defined in stm32f10x_type.h

#define P 4               // binary digits after comma

#define FROMFLOAT(x)  (FIA)(((double)x) * (1 << P))
#define FADD(x, y)    ((x) + (y))
#define FSUB(x, y)    ((x) - (y))
#define FMUL(x, y)    (((x) * (y)) >> P)

void print(FIA x) {
  printf("%f\n", ((double)x) / (1 << P));
}
```

1. Determine the precise value range (the subset of the real numbers $\mathbb{R}$) that is represented by the fixed-point datatype.
2. As discussed in the lecture, the chosen definition of multiplication FMUL has the problem that the intermediate result (x) * (y) can overflow, even if the final result could correctly be represented using the 32-bit fixed-point datatype. E.g., the correct result of the computation

$$1\,000 \cdot 60\,000 \;=\; 60\,000\,000$$

can be represented, but our naive definition FMUL will produce a negative number.

**Define two improved versions of FMUL ...**

1. one using the signed 64-bit integer datatype "**long long**" for intermediate computations,
2. one using the decomposition

$$x \;=\; (\underbrace{(x \gg P)}_{=x_l} \ll P) + \underbrace{(x \;\&\; ((1 \ll P) - 1))}_{=x_r}$$

into the digits $x_l$ before and the digits $x_r$ after the comma. This decomposition is helpful, since the product

$$x \cdot y \quad = \quad ((x_l \ll P) + x_r) \cdot ((y_l \ll P) + y_r)$$

can be simplified using the standard laws of arithmetic (like distributivity) in such a way that problematic intermediate overflows can be avoided, even if all operations are carried out in 32-bit arithmetic. Justify the correctness of your implementation. It is sufficient if your implementation works for $P = 4$.

Because this implementation is likely to be more involved than the other fixed-point operations, it is a good idea to formulate it as an ordinary function

```
FIA FMUL_decomp(FIA x, FIA y) { ... }
```

rather than defining it as a macro.

Test your two implementations to ensure that they produce the same results, and that internal overflows have effectively been prevented.

3. A naive way to add a division operation to our fixed-point datatype is the following definition:

```
#define FDIV(x, y)      (((x) << P) / (y))
```

Recall that we require all fixed-point operations to round downwards: the result produced by a fixed-point operation ($x$ o$\tilde{\text{p}}$ $y$) shall be the greatest fixed-point number not larger than the precise mathematical result ($x$ op $y$). Investigate whether this is the case for FDIV, in particular also taking negative operands into account. Argue that FDIV indeed rounds correctly, or show cases where FDIV produces incorrect results and provide a corrected implementation of FDIV.

You do not have to take internal overflows into account in this question: in particular, you can assume that the expression (x) << P will not overflow.

## Submission

Solutions to this assignment are to be submitted
via email to othmane.rezine@it.uu.se before the exercise session on
**Friday, February 17th, 2012, 08:15** (room 1245).

Make sure that your solution is clearly marked with your name and personal number. No solutions will be accepted after the exercise session.

Please note, the answers are due 2 weeks after publication of the assignment. You should use the free time to already work on the elevator lab. This lab requires quite some time.