Real-Time Systems (IN4343)
- Practical training 2020/2021 -
Hands-on MSP430 lab

Your saga begins here

You are hired by a company as an embedded-system engineer whose first task is to improve a legacy embedded system that has been marketed by the company for a couple of decades. This system is kind-of cursed! Whoever had worked on it had left the company after a week!

Yet here you are! A brave adventurous TUDelft-branded embedded-system engineer who will not only successfully reverse engineer this system but also improve it by reducing its scheduling overhead. Of course in this battle, your knowledge from the *real-time system course* is your Friesian horse that moves you forward, your *brain* is your lance that helps you defeat the problem, and your persistence is your shield that protects you against disappointments! No legacy software beats (or even bites) you as long as you have these three.

Figure 1. A brave adventurous TUDelft-branded embedded-system engineer who is going to fight with a legacy system

Your mission. Break the curse forever! Work on this legacy code and **(i)** understand it (survive), **(ii)** improve its efficiency by re-designing how the tasks are scheduled (fight), and **(iii)** add new features such as faster and more effective scheduling algorithms (win). Task (i) , (ii) and (iii) are mandatory in this assignment. All of these tasks have the same deadline of three weeks.

Reward. Your employer will let you *pass* to the next level to test you with the next monstrous software they have and see if you are worthy of receiving the title of *Real-Time System Engineer* and perhaps an interesting upgrade in your salary.

Survival hints. Here are a few tips for your survival during this mission:

1. Keep in mind that you are a real-time system engineer! Use your knowledge and creativity, but beware of your design choices.
2. Be critical and study the code carefully.
3. Write your answers concisely (briefly).

Setting up the virtual machine and running the hardware emulator

Installing the virtual machine. The legacy system is designed for a microcontroller MSP430. We use a hardware simulator to run the code and analyze the program behavior. Since the simulator has been designed for UBUNTU 9.10 which might be hard to find and install nowadays, we use the virtual machine provided by TU/e that already includes the simulator. You can access this virtual machine in the following address:
`http://www.win.tue.nl/san/education/2IN26/`

Note that you need to install a virtual-machine player such as vmware Workstation on your machine in order to be able to open the virtual machine of TU/e. You can download a vmware Workstation for free in the following address [1]:
`https://my.vmware.com/en/web/vmware/free#desktop_end_user_computing/vmware_workstation_player/15_0`

Creating a shared folder between the virtual machine and your host machine. After opening the virtual machine, you need to connect the virtual machine to the code library of the legacy system (i.e., a set of C files that are provided for each assignment and are supposed to run on the MSP430 microcontroller). The most maintainable way is to share a folder between your host PC/Laptop and the virtual machine. By doing that, you will be able to use your source code editors in your current machine and compile and execute the code on the hardware emulator in the virtual machine. Of course, you can just copy the C files to your virtual machine and do everything there without linking it to your current machine.

In order to share the folder between your host machine and the virtual machine, open the settings of the virtual machine and create a permanent shared folder from your *host* OS. Then you should run the `sudo vmware-config-tools.pl` command inside the virtual machine. Answer all questions with the default answers (press enter). Reboot the *guest* OS. You can reach the shared folder under `/mnt/hgfs`. If this does not work, you can just install VMTools for your virtual machine and then copy/paste files from windows to your virtual machine. This was tested on a windows 10 machine.

Running the simulator within the virtual machine. The following actions will allow you to get started with your experiments:

- *Compile the program, generate a trace, and visualize the result*:
  Execute the following commands.

  ```
  $ make clean
  $ make
  $ wsim-iclbsn2 --ui --trace=sched.trc --mode=time --modearg=5s SchedTest.elf
  $ wtracer --in=sched.trc --out=wsim.vcd-NPBASIC --format=vcd
  $ gtkwave wsim.vcd-NPBASIC
  ```

  The last command opens a GTKWave window; see Figure 2. In case you need to take a similar image from the simulator, you can use the "snippingtool" of Windows (if you work in a windows machine).

- *Set-up the visualizer*:
  By left-clicking `led` in the `STT` window, `blue`, `green`, and `red` are shown in the `Signals` window *below* the `STT window`. Note that `YELLOW` in the code (`Led.h`) is represented by the `blue` signal. By dragging these three signals (one by one)

  - *from* the `Signals` window *below* the `STT window`
  - *to* the `Signals` window *to the right* of the `STT window`,

  additional lines become visible in the `Waves` window.

  Left-click `msp430` in the `STT` window, and drag `intr_num[7:0]`

  - *from* the `Signal` window *below* the `STT window`
  - *to* the `Signal` window *to the right* of the `STT window`.

  Right-click `intr_num[7:0]`, and subsequently select `Data Format`, `Analog`, and `Step`.

[1] Or get the latest version from `vmware.com`

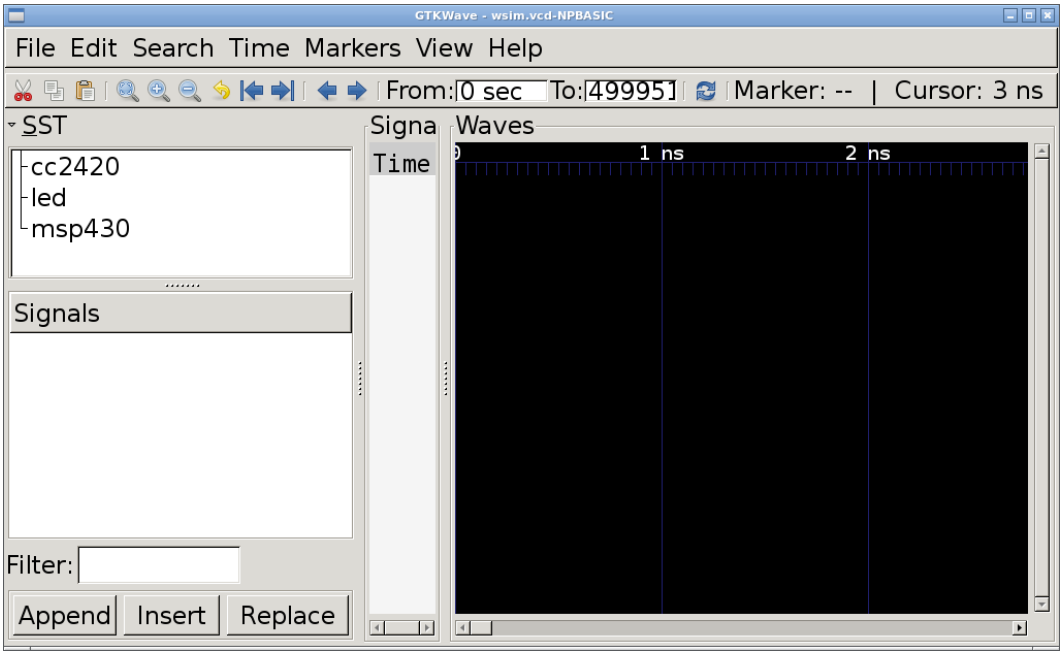

Figure 2. GTKWave window for `wsim.vcd-NPBASIC`.

- *Basic use of the simulator*:
  Repeatedly click "zoom-out", i.e. the icon with a magnifying-glass and an "−", to get a timeline covering at least 4 *ms*.

  By left-clicking on the rising edge of `blue` in the `Waves` window, a red line appears and the `Marker` provides detailed timing information.

  By pressing `Alt+F`, you automatically zoom the windows such that the whole trace is captured.

- *Advanced use of the simulator*:
  Document `Virtual machine for MSP430 development-2016.pdf` contains various hints for using the simulator. The usage of markers (see Section "Trace analysis" in that document) may considerably reduce workload.

Finally, more information regarding interrupt handling can be found in the user guide of the MSP430 hardware, see Section 2.2.3 in `http://www.win.tue.nl/san/education/2IN26/MSP430%20-%20general.pdf` (or `MSP430-general.pdf`).

Part 1: Understand the system

Learning objective: Measurement of different overheads.

Preparation: A package of source files will be available for this assignment. Download the Assignment1_code_part1.zip file for this assignment, which is provided at the course website. Study the distributed code[2].

Main questions

Answer the following questions in a digital format and upload it into BrightSpace. Please do not forget to add your names, student numbers, and the date of submission to the file you upload.

1. **(8 points)** How much time is spent on executing system functionalities (e.g., context switch function, scheduling function, timer interrupt handler, etc.) instead of executing the actual tasks within one hyperperiod? (Briefly justify your answer using plots or tables that show the overheads). Note: do not include the initialization phase (i.e., start counting from time first interrupt flank at 3.45ms). Hint: `intr_num` displays this overhead.

2. **(2 point)** What is the average system overhead per unit of time (or per clock tick)? System overhead is the same as the overhead mentioned in the previous question. Hint: calculate the time that was *not spent* on executing actual tasks for one hyperperiod (start after the initialization phase) and divide it by the length of the hyperperiod.

3. **(10 points)** How does the overhead change with the increase in the number of tasks in the system? To answer this question, draw a plot whose horizontal axis is the number of tasks (e.g., $\{1,2,3,4,5,6,7,8,9,10\}$ tasks) and its vertical axis is the average system overhead per unit of time (i.e., total system overhead per hyperperiod divided by the length of the hyperperiod as you have calculated in Question 2).

   To get the plot, you can simply increase the number of maximum tasks in the systems. This is defined by the `NUMTASKS` global variable defined in `Scheduler.h`.

[2]The MSP430x4xx User's Guide can be found at http://www.win.tue.nl/san/education/2IN26/MSP430%20-%20general.pdf

Part 2 : Improve the system

Learning objective: Improve the efficiency of the system by designing an event-based online scheduler on top of the bare metal hardware. The required code templates for this assignment are provided in Assignment1_code_part2.zip file, which can be downloaded from the course website.

Assessment instructions: You will be graded on a scale from 0-10. You will *pass* this assignment only if you gain at least 6 points *and* you have handed-in your code. You will have to present the code to the TA at any of the lab days after the submission of the assignment.

Instructions for improving the efficiency

As you have noticed in the previous assignment, in the current design, the scheduler imposes a significant amount of overhead to the system (namely, the system spends a lot of time only on scheduling the tasks rather than executing them!). In the second assignment, you are expected to re-design the structure of the scheduler (namely, modify how and when the scheduler is activated) to build an event-based online scheduler.

Recall. An *event-based scheduler* is the one that will be called (activated) only at *certain events*, e.g., *job-release event* and *job-completion event*. An *online scheduler* takes its decisions at runtime based on the current status of the system.

Properties of our desired scheduler. The *scheduler* is called at most only once per job-release event and at most only once per job-completion event. For example, if the system has 2 tasks $\tau_1 = (2,7)$ and $\tau_2 = (8,14)$ (where the numbers denote the execution time and period of each task, respectively), then the job-release events happen at times 0, 7, 14, 21, etc., and job-completion events happen at times 2, 9, 12, etc. In this example, the scheduler is called only at times 0, 2, 7, 9, and 12 (in the first hyperperiod). Figure 3 shows the schedule and the aforementioned events.

Hints on how to modify the code. As you may have noticed in Figure 3, in the desired system, timer interrupt is used to handle the job-release events so that there is no need to check if a new job is released at every timer tick (wasn't that crazy in the legacy system?!). However, a single timer interrupt can only handle one interrupt at a time! Hence, you cannot assign more than one timer interrupt to your hardware timer (since you only have one hardware timer). What is the solution then?

Conceptually, you need to use a queue (e.g., an array) to keep track of the timer events that are going to happen in the future[3]. For example, at time 0 after both τ_1 and τ_2 released their jobs, the next timer event for τ_1 will be at time 7 and for τ_2 will be at 14. However, your hardware timer cannot handle more than one interrupt at a time. So you need to first set the timer to provide you an interrupt at time 7, and then set it again for time 14. Looks good?

After sorting out how the release events are handled, you need to call the scheduler again.

[3] In terms of implementation, you may not really need an actual queue object, instead, you can add the extra information you need to store for the timer to the task data structure.

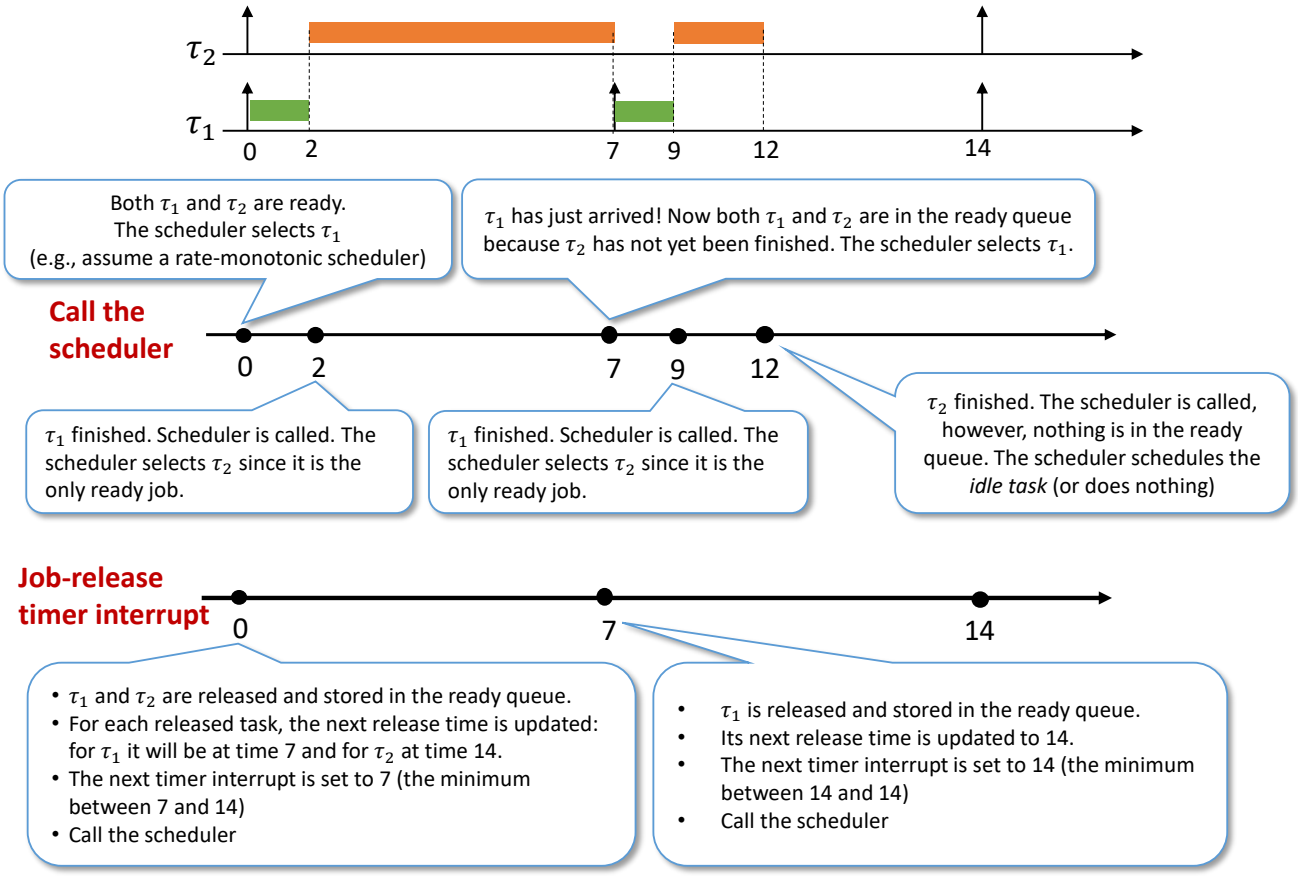

Figure 3. **(i)** The schedule of two periodic tasks τ_1 and τ_2 with periods 7 and 14 and execution times 2 and 8, respectively, using an online scheduling algorithm, **(ii)** the time instants at which the scheduler is called, and **(iii)** the time instants at which the timer-interrup handler is called to release the new jobs of the tasks.

```
static void ExecuteTask (Taskp t)
{
  /* ----------------------- INSERT CODE HERE ----------------------- */

  t->Invoked++;
  t->Taskf(t->ExecutionTime); // execute task

  /* ---------------------------------------------------------------- */

}

void Scheduler_P_FP (Task Tasks[])
{
  /* ----------------------- INSERT CODE HERE ----------------------- */

  /* Super simple, single task example */
  Taskp t = &Tasks[3];
  if (t->Activated != t->Invoked)
  {
    ExecuteTask(t);
  }
  /* End of example*/

  /* ---------------------------------------------------------------- */
}
```

Figure 4. Insert scheduler code in `Scheduler_P_FP.c`

```
interrupt (TIMERA0_VECTOR) TimerIntrpt (void)
{
  ContextSwitch();

  /* ----------------------- INSERT CODE HERE ----------------------- */

  /* Insert timer interrupt logic, what tasks are pending? */
  /* When should the next timer interrupt occur? Note: we only want interrupts at job releases */

  /* Super simple, single task example */
  Taskp t = &Tasks[3];
  t->NextRelease += t->Period; // set next release time
  t->Activated++;
  NextInterruptTime = t->NextRelease;
  /* End of example*/

  /* ---------------------------------------------------------------- */


  TACCR0 = NextInterruptTime;

  CALL_SCHEDULER;

  ResumeContext();
}
```

Figure 5. Insert timer interrupt code in `SchedulerOnline.c`

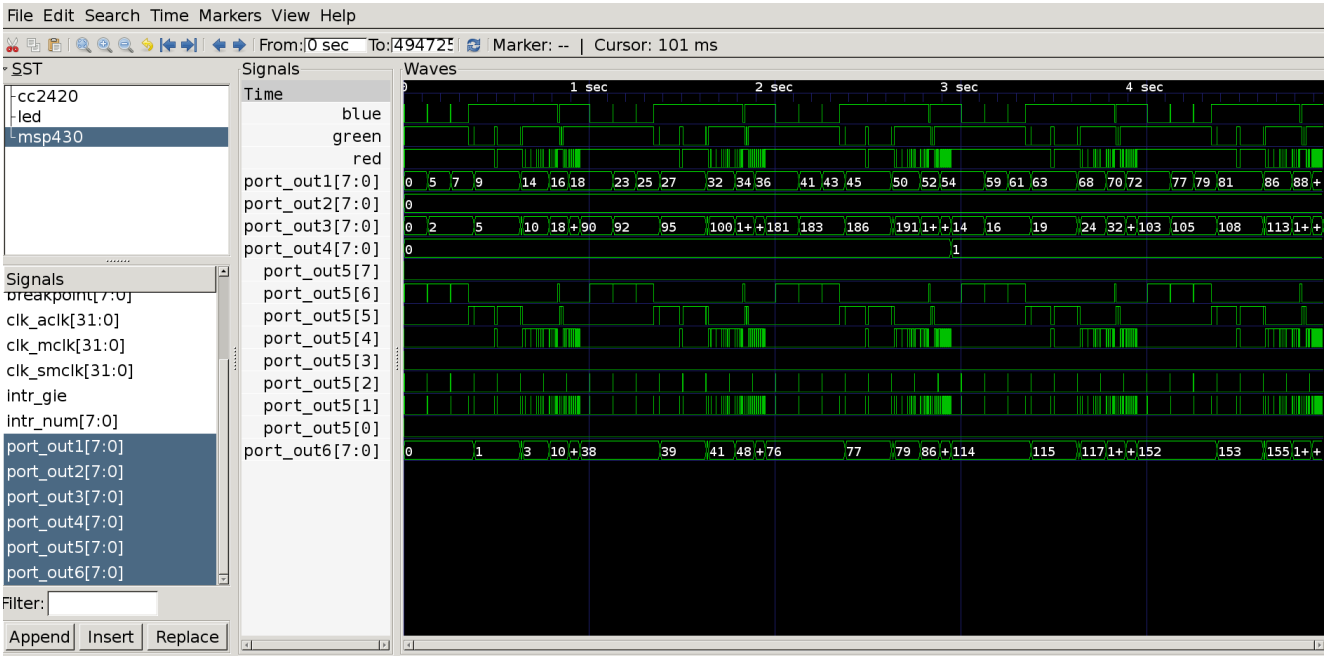

Figure 6. Use the output ports of the microcontroller for debugging

What you should not change. In order to ensure that you focus only on the design of scheduler and task-activation mechanism, please try to not modify any files or functions other than `TimerIntrpt`, `ExecuteTask` and `Scheduler_P_FP`

Hints on how to debug the code. Developing software comes with writing bugs. A common way of debugging is using `printf` to show variables values or following the execution path of the your program. However, this is not supported in this simulator. Instead, you can use the output ports of the microcontroller to log variables or the show execution path. Example output is shown in Figure 6.

The microcontroller has six output ports: `port_out1` to `port_out6`. Note that on `port_out5`, the three LEDs `yellow`, `green` and `red` are placed (inverted compared with the original). You don't have to use these individual signals anymore for this assignment, you can use `port_out5` to show the LEDs. Use `port_out5` for toggling LEDs and use the other ports to log variable values. Note that these output ports are have eight bits. If you want to log a 16-bit variable, you should use two ports to visualize this.

To import the signals into the viewer, select the output ports, hold `shift` and drag + drop them intro the wave viewer. Then select `port_out5` and press `F3` to expand the signal to 8 individual outputs.

If you see that you're simulation is not running up to the default 5 seconds, but gets stuck after say 1 ms, you probably dereferenced a wild pointer or accessed an array out of it's bounds. So be carefull when dereferencing pointers and array elements!

Before you start coding. To verify that your scheduler is correct, have you to compare the result with the correct schedule. So before you start coding, make a sketch of one hyperperiod of task set `Tst1` defined in `SchedTest.c`.

Expected deliverables.

- **(10 points)** Hand-in your source code (only the files that must run on top of the hardware simulator as a zip file via BrightSpace Assignment 1. During the demo, you will be asked to run the example that is provided with the code on your system for at least two hyperperiods. Be prepared for some modifications that is asked from you by the TA on the spot. For example, he may ask you to change some parameters to check if the code works nicely. For this lab you only have to implement the preemptive fixed point scheduler in `Scheduler_P_FP.c`, you can ignore the other scheduler source files.
- **(1 points)** Take a snapshot of the schedule for one hyperperiod of task set `Tst1` defined in `SchedTest.c`. Compare this with the sketch you drew up front. Write a short description to explain how the scheduler and timer interrupt handler work in your code (no more than 500 words).
- **(5 point)** Calculate the system overhead in one hyperperiod. To backup your answer, add a snapshot of your schedule and explain how much time is spent on anything expect executing the tasks (you do not have to consider idle time). Because measuring all scheduling invocations by measuring time in GTKWave is a tedious job, you can make use of `TimeTracking.c`. There you instrument the code to do measurements for you.
- **(2 point)** Does the method of overhead measurement provided in `TimeTracking.c` interfere with the program execution? Justify your answer in at most 100 words.
- **(5 points)** Plot the overhead (in percentages %) as a function of the number of tasks in your system (ranging from 1 task to 10 tasks). For this question, you can use the given task set in `SchedTest.c`, by defining `TstSweep`.
- **(2 points)** How does the chosen task set parameters (periods and execution times) influence the total overhead of the system in a hyperperiod? Justify your answer in at most 100 words.

Part 3: Add new features

Learning objective: Implement several preemptive and non-preemptive real-time scheduling algorithms in the system. Your design must follow an event-based online scheduler.

This assignment asks you to implement a **Preemptive EDF (P-EDF)** scheduling policy. The algorithm must be implemented in the `Scheduler_P_EDF.c` which is provided for you in the package of Part 2 of Assignment 1. You can select the appropriate scheduler in `Scheduler.h`

Expected deliverables

- **(15 points)** Hand-in your source code (only the files that must run on top of the hardware simulator) as a zip file via BrightSpace.
- **(3 points)** Take a snapshot of the schedule for one hyperperiod of task set `Tst1`, `Tst2` and `Tst3` defined in `SchedTest.c` for each of the each of the requested scheduling algorithms. Compare this with the sketch you drew up front.
- **(5 points)** Write a short description to explain what you have done to implement this algorithms (no more than 200 words).
- **(7 points)** Plot the overhead (in percentages %) as a function of the number of tasks in your system (ranging from 1 task to 10 tasks) for each of the scheduling algorithms that you have implemented. Hint: for this experiment, you can use the given task set in `SchedTest.c`, by defining `TstSweep`.

Node: The code of this assignment should be presented to the TA for the evaluation.