Introduction

The focus for the next four weeks of this course will be on real-time task scheduling. We start with a simple cyclic executive and end up using a Real-Time Operating System (RTOS). At the end of week 6 you have to submit a short report of the assignments of week 3 to 5 in a task created for this purpose in Brightspace. In this report, formatted in `.pdf`, you will discuss the code you wrote and answer the questions posed in the assignments.

In these assignments we will not use the LL and HAL APIs from STM because we want to develop software that will run on any Cortex-M4 microcontroller[1].

Assignments week 3 – Cyclic executive and coöperative scheduler

This week you will learn how to:

- manage time by using the SysTick timer with:
  - polling;
  - interrupt.
- implement these methods for managing time:
  - without using an API;
  - by using the CMSIS API.
- save power by putting the processor to sleep when there is nothing to do except to wait until the time passes;
- develop a cyclic executive;
- develop a cooperative scheduler;

Cyclic Executive

3.1 In this assignment you will create (again) a program which will blink the user LEDs of the STM32F411E-DISCO development board. But instead of using a **`for`**-loop to pass time, you will use the so called SysTick timer. This timer is part of the Cortex-M4 core and is therefore present in every Cortex-M4 microcontroller. It is not documented in

[1] The only part which will not, and can not, be portable will be the use of the peripherals.

RM0383 (STM32F411xC/E advanced Arm-based 32-bit MCUs reference manual) but in PM0214 (STM32 Cortex-M4 MCUs and MPUs programming manual).

A Copy the project `opdr_2_1` to `opdr_3_1` in the Project Explorer of STM32CubeIDE. Delete in this *new* project the `Debug` folder and the `.launch` file.

B Read section 4.5 of PM0214. Configure the SysTick timer to set the `COUNTFLAG` in the `STK_CTRL` register every 0.5 s. Replace the **`for`**-loop with the following C code:

```
while ((STK_CTRL & (1 << 16)) == 0);
```

You have to properly define the symbol `STK_CTRL` yourself to make this work. Build and debug the project. If all is well, the user LEDs will blink with a frequency of 1 Hz.

3.2 In this assignment you will repeat assignment 3.1 but this time you will use the CMSIS API.

A Copy the project `opdr_2_2` to `opdr_3_2` in the Project Explorer of STM32CubeIDE. Delete in this *new* project the `Debug` folder and the `.launch` file.

B Configure the SysTick timer to set the `COUNTFLAG` in the `STK_CTRL` register every 0.5 s using the CMSIS API. Replace the **`for`**-loop with the following C code:

```
while ((SysTick->CTRL & (1 << 16)) == 0);
```

The symbol `SysTick` is defined in the CMSIS API. Build and debug the project. If all is well, the user LEDs will blink with a frequency of 1 Hz.

3.3 In this assignment you will repeat assignment 3.1 but instead of polling the `COUNTFLAG` you will use an interrupt.

A Copy the project `opdr_3_1` to `opdr_3_3` in the Project Explorer of STM32CubeIDE. Delete in this *new* project the `Debug` folder and the `.launch` file.

B Configure the SysTick timer to generate an interrupt (also called an exception) every 0.5 s. Read section 2.3 of PM0214. The exception vectors are already defined in the file `startup_stm32f411vetx.s` that you can find in the `Core‣Startup` folder of the project. On line 142 of this file, the SysTick vector is filled with the symbol `SysTick_Handler` which is defined using the `.weak` assembler

directive[2] on line 263. This will define the symbol if it is not already defined. On line 264 this symbol is made an alias for the `Default_Handler` function by using the `.thumb_set` assembler directive[3]. This ensures that the function `Default_Handler` is called when the function `SysTick_Handler` is *not* defined. The function `Default_Handler` is defined on line 111 and contains an infinite loop. So the only thing you have to do, to handle the SysTick interrupt, is to define a function named `SysTick_Handler`.

Define a global boolean flag in `main.c` like this:

```
#include <stdbool.h>
volatile bool flag = false;
```

Define a function `SysTick_Handler` which makes this flag **`true`**.

Replace the **`while`**-loop which polls the `COUNTFLAG` with the following C code:

```
while (!flag);
flag = false;
```

Build and debug the project. If all is well, the user LEDs will blink with a frequency of 1 Hz.

C Why must the `flag` variable be defined as **`volatile`**?

D It seems that there is no good reason to prefer an interrupt above polling in this case. But there is! The Cortex-M4 processor can be put in a low power mode by executing the **`WFI`** assembler instruction. The core resumes execution when it receives an interrupt. Read section 3.11.12 of PM0214.

Replace the **`while`**-loop which waits for the `flag` to become **`true`** with code which will put the processor to sleep until the next SysTick interrupt. To execute an assembly instruction, you can use the macro `__asm__`[4]:

```
__asm__("    ");
```

Make sure to include some space before the instruction itself. E.g.:

[2] https://sourceware.org/binutils/docs/as/Weak.html#Weak

[3] https://sourceware.org/binutils/docs/as/ARM-Directives.html#index-_002ethumb_005fset-directive_002c-ARM

[4] https://gcc.gnu.org/onlinedocs/gcc-12.1.0/gcc/Using-Assembly-Language-with-C.html

```
__asm__("    nop");
```

The variable `flag` is now not needed anymore and can be removed. Build and debug the project. If all is well, the user LEDs will blink with a frequency of 1 Hz.

3.4 In this assignment you will repeat assignment 3.3 but this time you will use the CMSIS API.

A Copy the project `opdr_3_2` to `opdr_3_4` in the Project Explorer of STM32CubeIDE. Delete in this *new* project the `Debug` folder and the `.launch` file.

B Configure the SysTick timer to generate an interrupt every 0.5 s. This can be done by using the function `SysTick_Config` from the CMSIS API. Define an empty function to handle this interrupt. Replace the **`while`**-loop which polls the `COUNTFLAG` with code which will put the processor to sleep until the next SysTick interrupt. There is a function defined in the CMSIS API to accomplish this, see section 2.5.4 of PM0214.

3.5 Now based on project `opdr_3_4` create a rotation loop which simulates a simple traffic light: red (5 seconds), orange (1 second), green (4 seconds). The time each light is on must be easily adjustable with a granularity of 0.5 s. The processor must be put to sleep in between interrupts. Make use of an enumeration construct (**`enum`**) for the colors and a **`switch-case`**-statement for the rotation.

You have just created a simple and efficient scheduler a so-called cyclic executive!

Coöperative Scheduler

You will expand on the simple scheduler by implementing a list to which functions can be added to run at a specific period measured in ticks. A tick is a specific amount of time. In this assignment you have to use ticks of 1 ms. We will thus need a **`struct`** in C to combine a function pointer with a period, a counter and a possible initial delay. This **`struct`** will describe a “task”. The goal is to be able to run a task at a specified period of e.g. 500 ticks.

3.6

- Create a copy of the previous project and rename it to `opdr_3_6`.
- Using the description of this assignment, define a **`struct`** for a “task” and create a global array of 8 tasks.

- Create a function to initialize and add a task to this task list (the array).
- Create 4 functions (tasks) to toggle each led separately.
- Add each function (task) to the task list with periods of: 200 ticks, 500 ticks, 750 ticks, and 300 ticks for green, orange, red, and blue respectively.
- In the SysTick ISR, walk through the task list and decrement each of the task counters.
- Think of, and expand on, the task struct to notify per task whether it is in a `WAITING` or `READY` state. Set the state in the ISR depending on the task counter.
- Create a function `runReadyTasks()` that will walk through the task list and execute any task in the `READY` state. Replace your **`switch-case`** rotation in the function `main` with a call to this function.
- Make use of a logic analyzer to verify the timing of the tasks

3.7 Now add initial delays to your tasks. Use an initial delay of 100, 200, 300, and 400 for green, orange, red, and blue respectively. Make use of a logic analyzer to verify the timing.

Enjoy the show of your “advanced” cooperative scheduler.