Mini RTOS

Real-time systems TP1 and TP2

Lab objective

Simple real-time operating system for Cortex-M microcontrollers.

You can work in teams of **two**.

Tools:
Arm Keil suite (Windows) in simulation mode (using the board is facultative)

Due date: before TP3

Other Materials:

- Lecture 7
- Context Switch on the ARM Cortex-M0
- Joseph Yiu. *The Definitive Guide to ARM Cortex-M0 and Cortex-M0+ Processors*. Chapter 10.
- Joseph Yiu. *The Definitive Guide to the ARM Cortex-M3*.

Lab objectives:

- Use cyclic scheduling (Lecture 4)
- Enable preemptions with context switch
  - Switching exception frame (registers `xPSR`, `PC`, `LR`, `r12`, and `R0-R3`)
  - Using a `PendSV` exception to defer the context switch
- Use privileged access level for kernel and non-privileged for user threads
- Use separate stacks for kernel (`MSP`) and user threads (`PSP`)

Steps:

1 Configure `SysTick` timer to invoke scheduling actions (2pts)

2 Assign stacks to user threads and switch into unprivileged thread mode (2pts)

3 Program context switch in the `SysTick` handler (2pts)

4 Use `PendSV` to defer context switch (2pts)

5 Find the cyclic schedule (2pts)

1 SysTick

`SysTick` is a simple timer inside the processor to perform the function of generating periodic interrupt requests:

- 24-bit down counter,
- reloads automatically after reaching zero and the reload value is programmable,
- when reaching zero, the timer can generate a `SysTick` exception triggering the `SysTick` exception handler.

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	STK_CTRL	Reserved															COUNTFLAG	Reserved													CLKSOURCE	TICK INT	EN ABLE
	Reset Value																0														1	0	0
0x04	STK_LOAD	Reserved							RELOAD[23:0]																								
	Reset Value								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	STK_VAL	Reserved							CURRENT[23:0]																								
	Reset Value								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	STK_CALIB	Reserved							TENMS[23:0]																								
	Reset Value								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

`ENABLE` Counter enable
`TICKINT` `SysTick` exception request enable
`CLKSOURCE` Clock source selection

The `SysTick` registers can be accessed as follows:

```
SysTick->LOAD = myval;
SysTick->VAL  = myval;
SysTick->CTRL = myval;
```

The `SysTick` exception handler can be defined in:

```
void SysTick_Handler(void) {
        ...
}
```

Task 1: Program `SysTick` to generate an interrupt every 1 ms. Consider input clock frequency of 72 MHz. Debug your system and set a breakpoint inside the `SysTick` handler and verify how often the handler is called.

2 Processor modes and privilege levels

Cortex-M processor can run in one of two modes:

Thread mode Used to execute application software. The processor enters *Thread* mode when it comes out of reset.

Handler mode Used to handle exceptions. The processor returns to *Thread* mode when it has finished exception processing.

Thread mode can be configured in one of two privileged levels:

Unprivileged cannot access the system timer, NVIC, or system control block and might have restricted access to memory or peripherals.

Privileged can use all the instructions and has access to all resources.

The processor starts in *Privileged Thread Mode*. It can switch itself into unprivileged access level by programming the `CONTROL` register (see below).

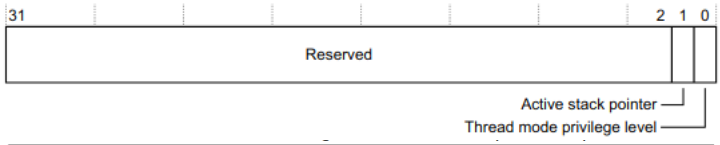
The processor has two separate stack pointers:

Main Stack Pointer (`MSP`) used in *Handler* mode

Process Stack Pointer (`PSP`) used in *Thread* or *Handler* mode.

Processor mode	Used to execute	Privilege level for software execution	Stack used
Thread	Applications	Privileged or unprivileged(1)	Main stack or process stack(1)
Handler	Exception handlers	Always privileged	Main stack

By default, *Thread* mode uses the `MSP`. To switch the stack pointer used in Thread mode to the `PSP`, you need to write to `CONTROL` register.



Bits	Function
Bit 1	**ASPSEL:** Active stack pointer selection Selects the current stack: 0: MSP is the current stack pointer 1: PSP is the current stack pointer. In Handler mode this bit reads as zero and ignores writes.
Bit 0	**TPL:** Thread mode privilege level Defines the Thread mode privilege level. 0: Privileged 1: Unprivileged.

The `CONTROL` register can be read/written as follows:

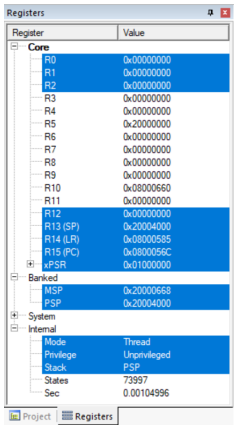
```
__set_CONTROL(ctrl);
ctrl = __get_CONTROL();
```

The `PSP` register can be read/written as follows:

```
__set_PSP(psp_value);
psp_value = __get_PSP();
```

Task 2: Switch to *Unprivileged* mode with *Process Stack*. Use `CONTROL` register.

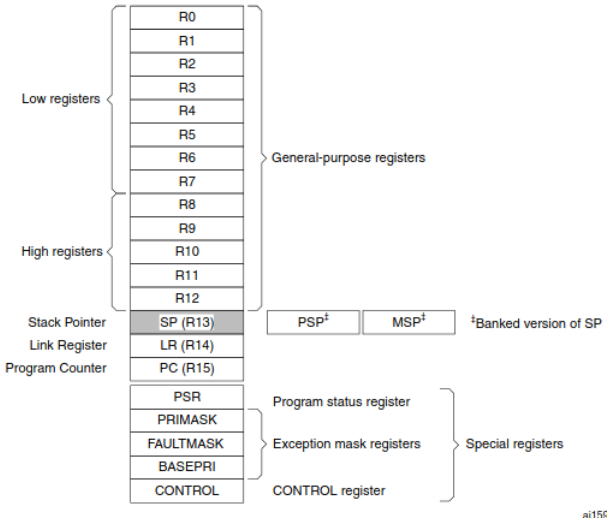
Question: Run debug and see how the internal state of the processor (mode, privilege, and stack) changes when the program enters and leaves `SysTick` handler.



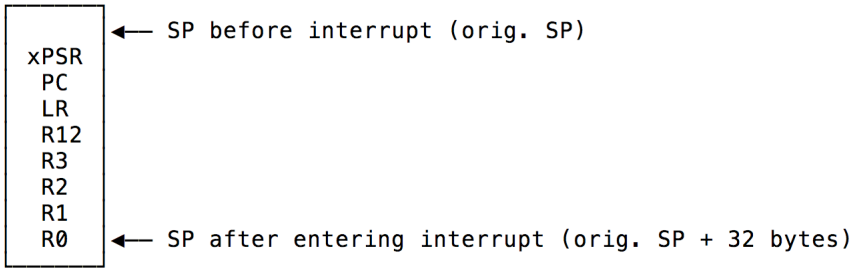
3 Context Switch

Once an interrupt occurs, the NVIC hardware automatically:

- stacks onto *Process Stack* (`PSP`) an exception frame (registers `xPSR`, `PC`, `LR`, `R12`, and `R0-R3`),
- branches to the interrupt handler routine in *Handler* Mode (which uses the *Main Stack* (`MSP`)).



The following figure shows the basic exception stack frame saved by hardware (each register has 32 bits).



The *Program Counter* (`PC`) should point to the memory address where the execution will jump in after unstacking.

Link Register (`LR`) defines exception return behavior:

`0xFFFFFFF1`	go back to *Handler* mode with *Main* stack
`0xFFFFFFF9`	go back to *Thread* mode with *Main* stack
`0xFFFFFFFD`	go back to *Thread* mode with *Process* stack

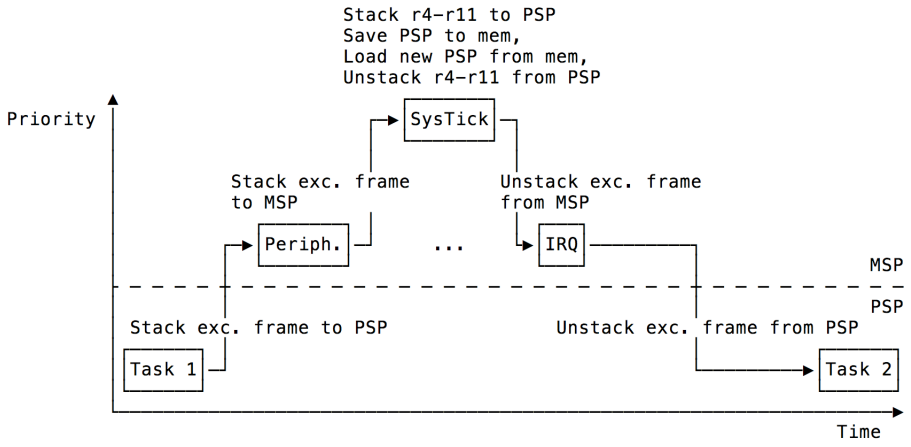
Task 3: a) Write two dummy functions with an infinite loop and no body. **b)** Initialize stack for each function. Consider that RAM starts at `0x20000000` and its size is `0x5000`. Registers `R0-R3` and `R12` should be initialized to `0x00`,

and the register `XPSR` default value is `0x01000000`. **c)** Execute two functions in an alternating round-robin manner using the *SysTick* handler to change the current *Process* stack pointer (`PSP`). **d)** Test your system in Debug mode.

Question: How would the design of a system change if it were non-preemptive?

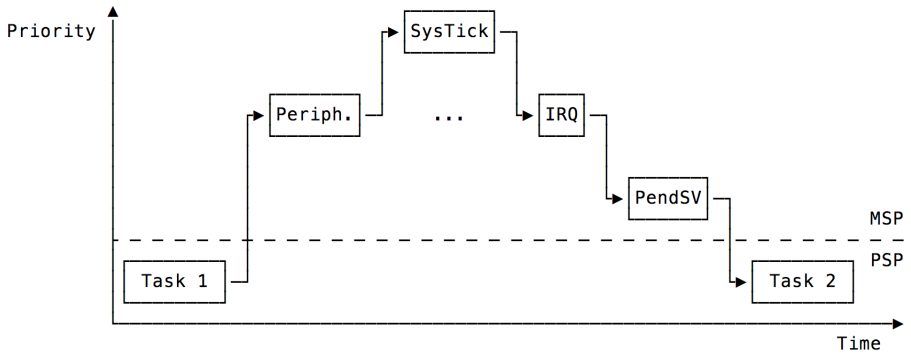
4 PendSV

The processor automatically saves registers `R0-R3`, `R12`, `LR`, `PC`, and `xPSR` but the program might also use other registers, `R4-R11`. These registers need to be saved manually. Typically, during context switch, the registers `R4-R11` are pushed/popped to/from the stack. However, what will happen if a context switch is triggered when the processor is not running a user thread but processing another interrupt? Analyze the situation shown in the next figure. Suppose that the interrupt service routine is using registers `R4-R11`.



(https://www.adamh.cz/blog/2016/07/context-switch-on-the-arm-cortex-m0/)

To avoid this problem, the *Pendable Service Call* (`PendSV`) exception can be used. It permits to postpone the context-switching request until all other IRQ handlers are finished. It must be assigned the lowest priority. The `SysTick` will choose the next task and defer the actual context switch to the `PendSV`.



(https://www.adamh.cz/blog/2016/07/context-switch-on-the-arm-cortex-m0/)

`PendSV` is triggered by writing 1 to the `PENDSVSET` bit in the NVIC *Interrupt Control State Register* (`ICSR`):

```
SCB->ICSR |= SCB_ICSR_PENDSVSET;
```

To set the interrupt priority levels (`0x00` is the highest, `0x03` the lowest) use:

```
NVIC_SetPriority(SysTick_IRQn, systick_prio);
NVIC_SetPriority(PendSV_IRQn,  pendsv_prio);
```

Task 4: Use `PendSV` to prevent context switching in the middle of an interrupt handler. For simplicity, consider that only `R4` must be saved during context switch.

5 Cyclic Scheduling

Task 5: Is the following task set feasible on a single processor? If so, construct a cyclic schedule. How would you integrate its timeline in your scheduler?

	C_i	T_i
τ_1	4	16
τ_2	30	40