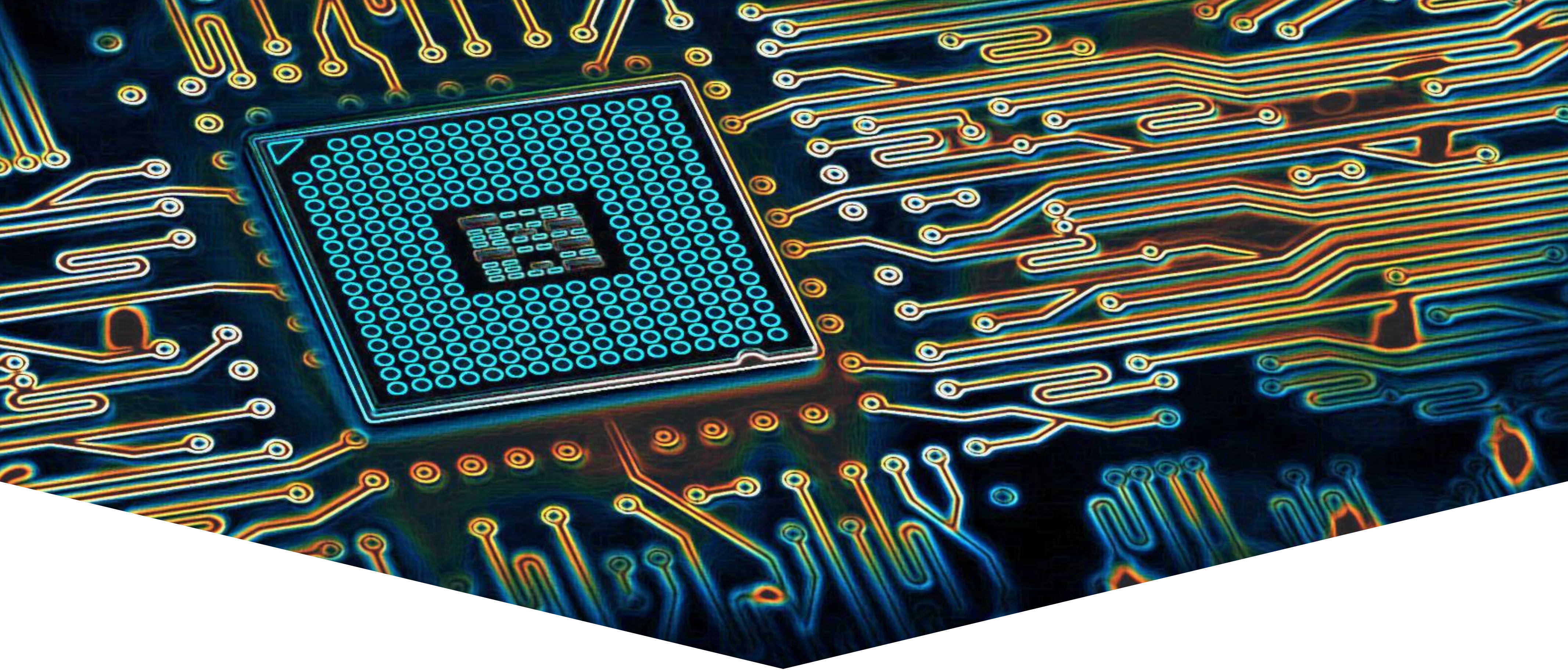


RTS10 Week 4

Planning RTS10

Week 3: Cooperative Scheduling

Week 4: Pre-emptive Scheduling

Week 5: Using an RTOS

Week 6: Schedulability Analyses, Priority Assignment

Week 7/8: Introduction (embedded) Rust

HOGESCHOOL ROTTERDAM

Overview

Scheduling

- Problem
- Goal
- Possible solution

Scheduling

Problem

- Multiple processes require CPU time
  - Some processes need it asap
  - Some processes just need to happen at some point in time
- Multiple processes require bandwidth
  - USB, Serial, SPI ....
  - Prioritization?

Goal

- Create a framework that'll ease (CPU) time management
- Easy to add new processes and to share resources

HOGESCHOOL ROTTERDAM

Demo

Demonstration of pre-emptive project

HOGESCHOOL ROTTERDAM

Cooperative versus Pre-emptive scheduling

Cooperative

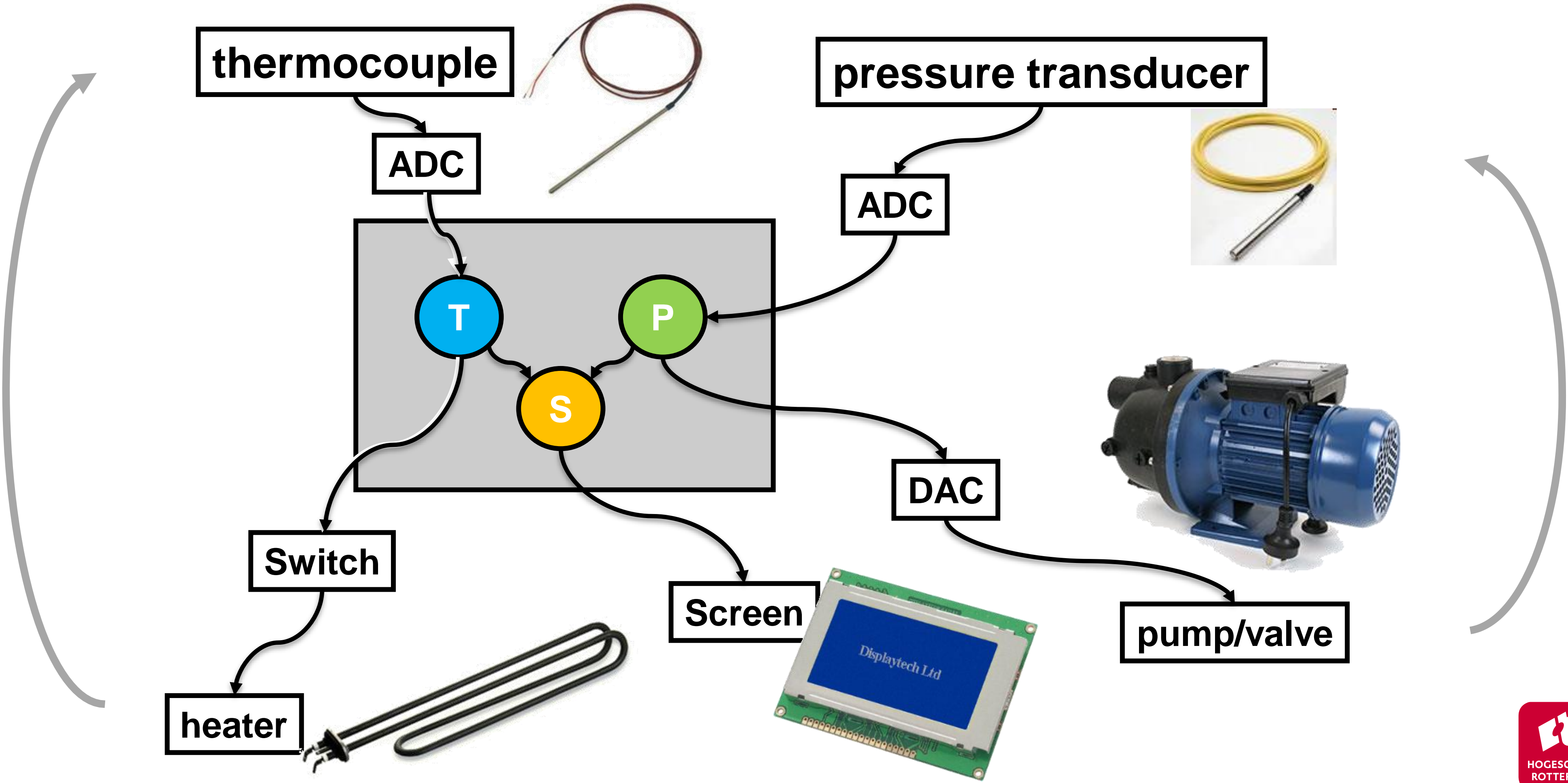
- Tasks run sequentially
- High priority tasks have to wait till last task finishes
- Easy to set up
- Low overhead scheduler

Pre-emptive (Multi-tasking)

- Important tasks always finish first
- Danger of starvation and using hardware concurrently
- More overhead on resources(RAM) and CPU time

HOGESCHOOL ROTTERDAM

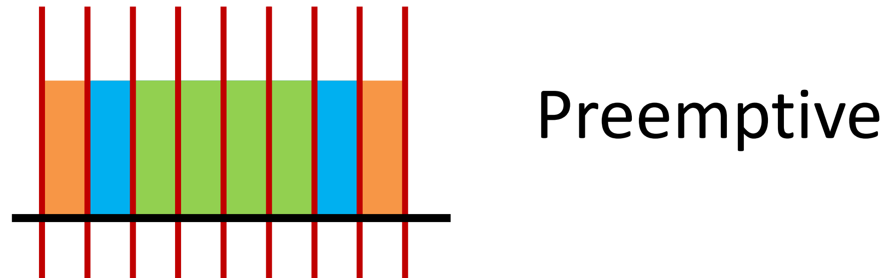
Embedded system example



Pre-emption

- Interrupting a task to execute a different task

Cooperative



Preemptive

- Scheduler decides next task and can interrupt existing
- Context switch, switches the tasks

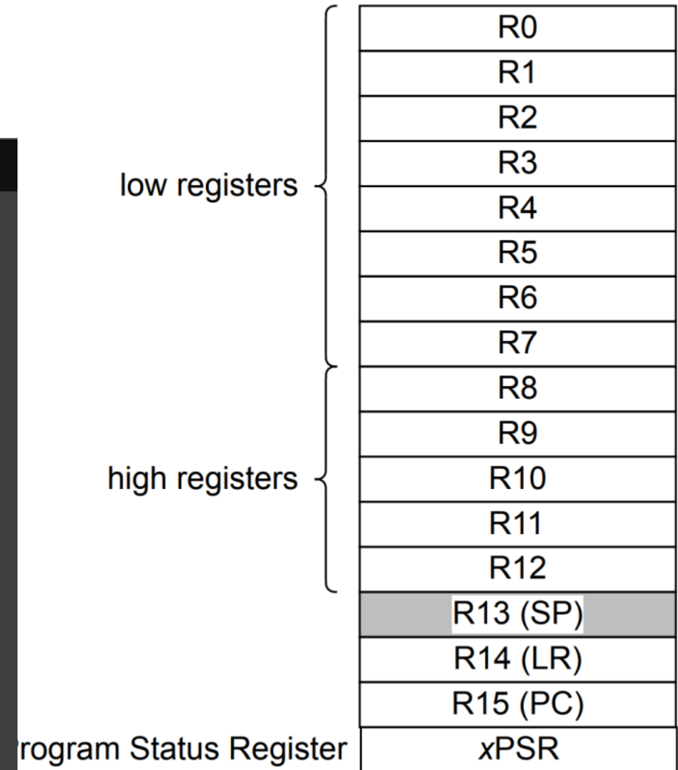
HOGESCHOOL ROTTERDAM

Context switch

- What is context?

```
 1  void * TempControl(void * par)
 2  {
 3      int adcSample, dacSample;
 4                                  //P, I,   D
 5      pidSettings pidSetting = {1, 2.3, 0.04}
 6      while(1)
 7      {
 8          adcRead(&adcSample);
 9          dacSample = pid(100, //setpoint
10                      &pidSetting,
11                      adcToTemp(adcSample)
12                      )
13          dacWrite(dacSample);
14      }
15  }
```

-



OS support in Cortex M4

- Banked stack pointers
- Privileged and non-privileged operation modes
- Advanced interrupt controller (NVIC)
  - Has several interrupts specifically for RT kernels
  - Fault handlers
- Memory Protection Unit (MPU)

HOGESCHOOL ROTTERDAM

Banked stack pointers

- R13 can contain
  - the Process Stack Pointer (PSP)
  - the Main Stack Pointer (MSP)
- MSP is default
- PSP can be used for separating tasks from the OS
  - Normal tasks do not share their stack with the kernel! Much safer.
  - Can only be used outside of interrupts
  - Set using SPSEL bit in CONTROL register
- Thus far all we have used is the MSP

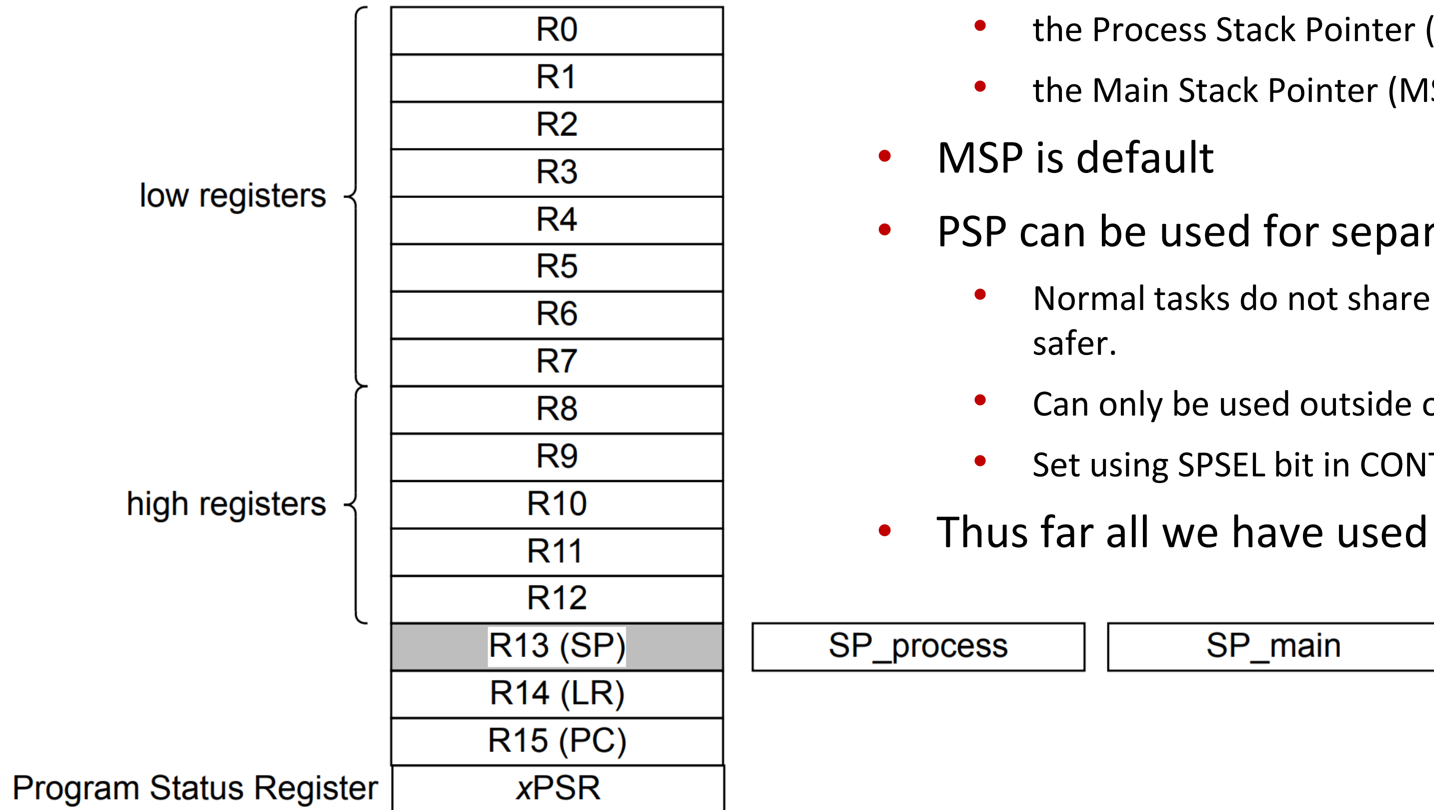
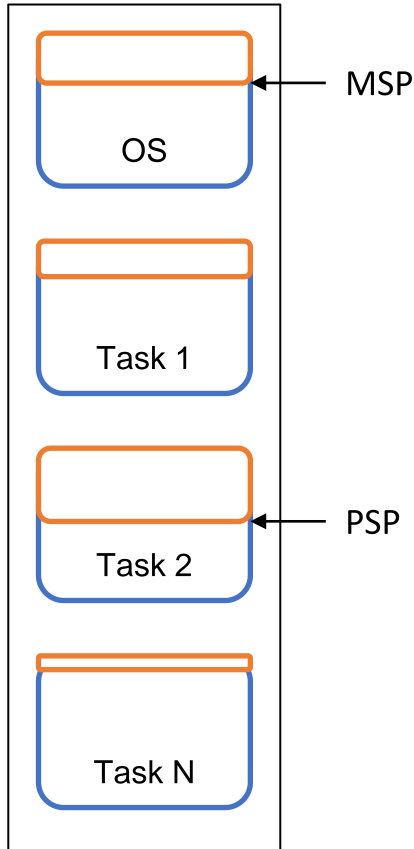


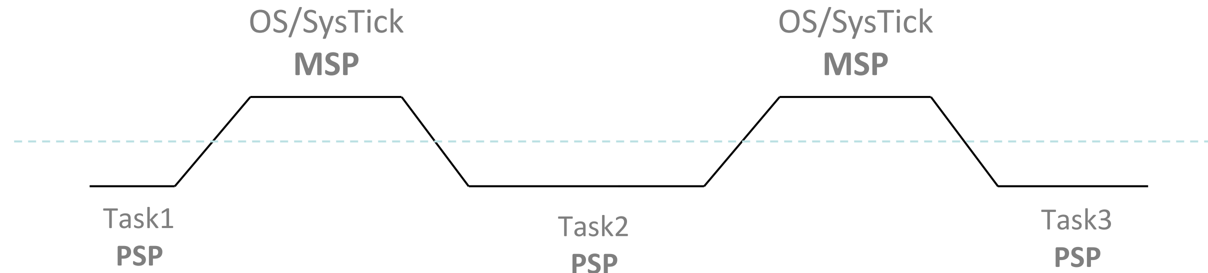
Figure 3-3 Processor register set

HOGESCHOOL ROTTERDAM

Memory usage in a RTOS



- Each task gets own chunk of memory
  - The OS uses the MSP
  - The tasks use the PSP
    - When a new task is selected, the PSP points to the stack of that task.
    - Basically, a form of ‘Time division multiplexing’
    - The scheduling algorithm picks the next task/stack



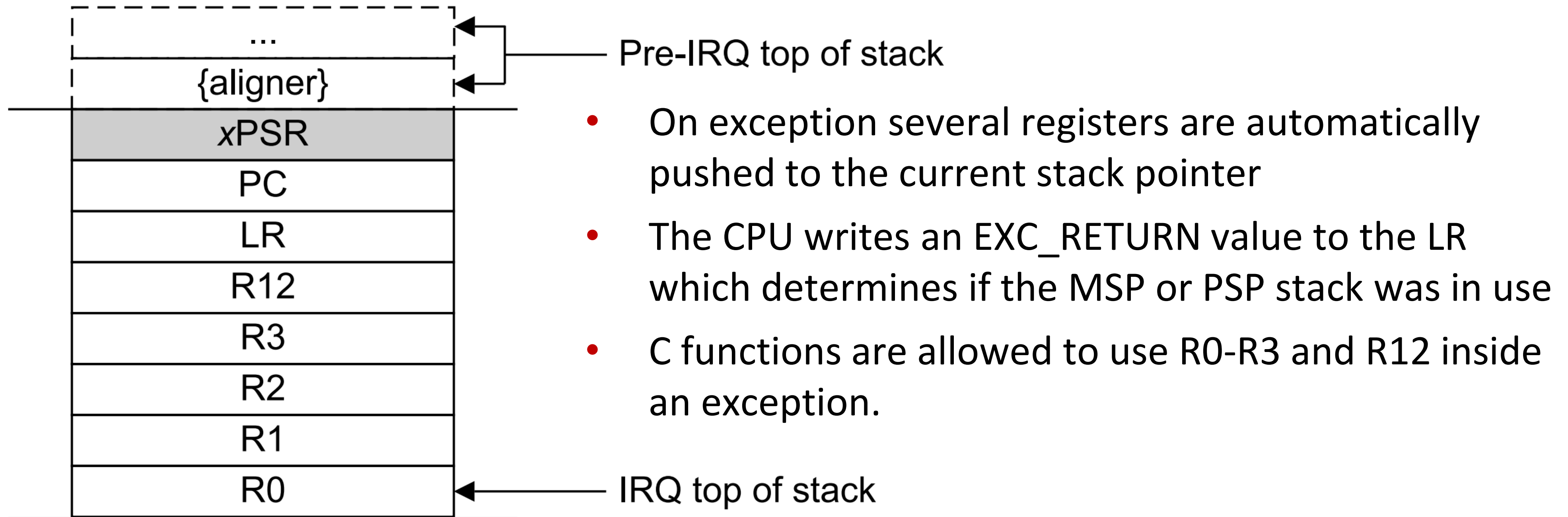
HOGESCHOOL ROTTERDAM

Banked stack pointers

- Two assembly instructions for accessing special registers
  - MRS (move special register value to normal register)
  - E.g. `MRS R0, PSP ; move PSP value to R0`
  - MSR (Move normal register value to special register)
  - E.g. `MSR PSP, R0 ; move R0 value to PSP`
- And CMSIS functions
  - `__get_PSP(void)`
  - `__set_PSP(uint32_t topOfStack)`

HOGESCHOOL ROTTERDAM

Exception entry



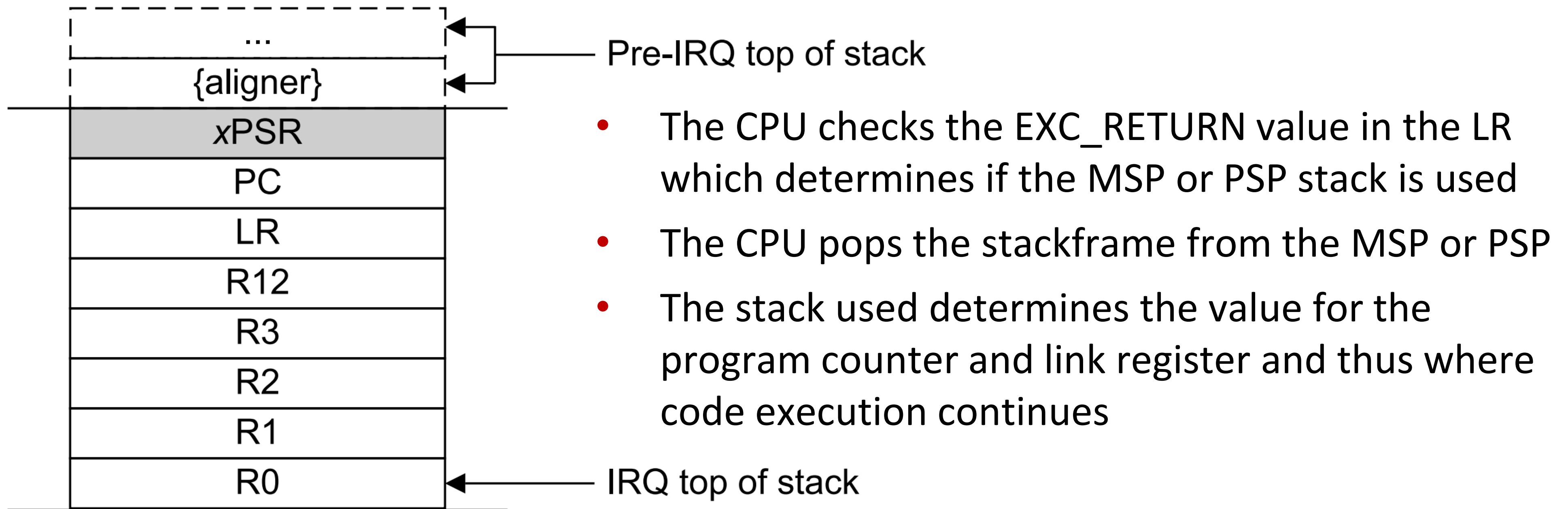
- On exception several registers are automatically pushed to the current stack pointer
- The CPU writes an EXC_RETURN value to the LR which determines if the MSP or PSP stack was in use
- C functions are allowed to use R0-R3 and R12 inside an exception.

Exception frame without floating-point storage

Figure 2-3 Exception stack frame

HOGESCHOOL ROTTERDAM

Exception return



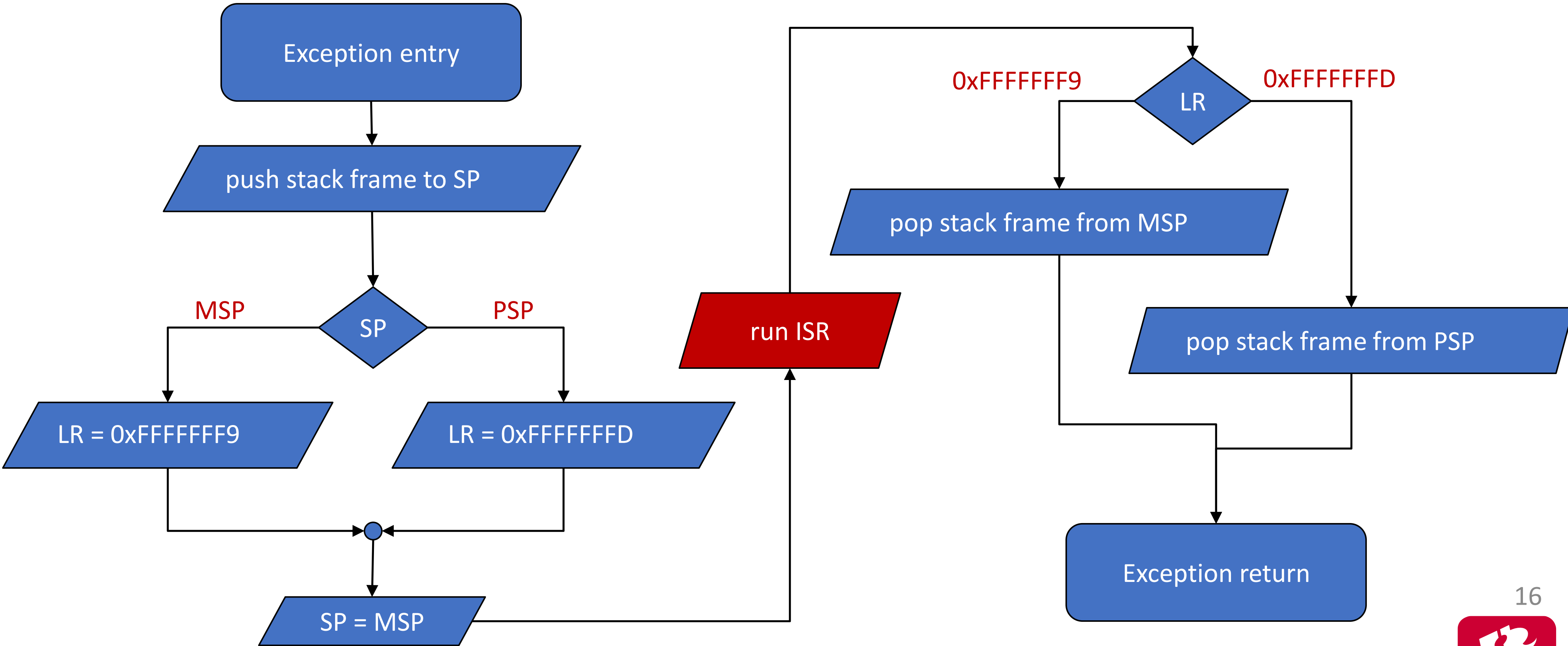
- The CPU checks the EXC_RETURN value in the LR which determines if the MSP or PSP stack is used
- The CPU pops the stackframe from the MSP or PSP
- The stack used determines the value for the program counter and link register and thus where code execution continues

Exception frame without floating-point storage

Figure 2-3 Exception stack frame

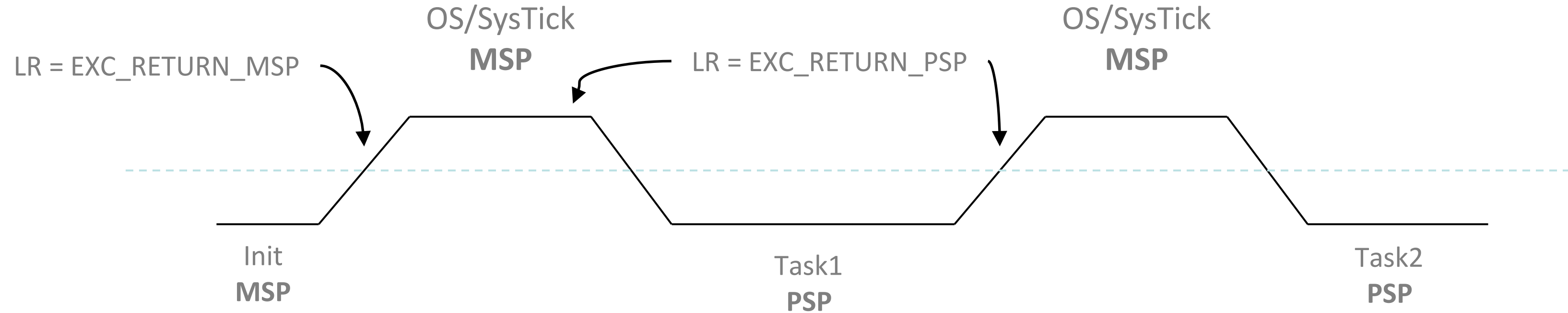
HOGESCHOOL ROTTERDAM

Exception flowchart



HOGESCHOOL ROTTERDAM

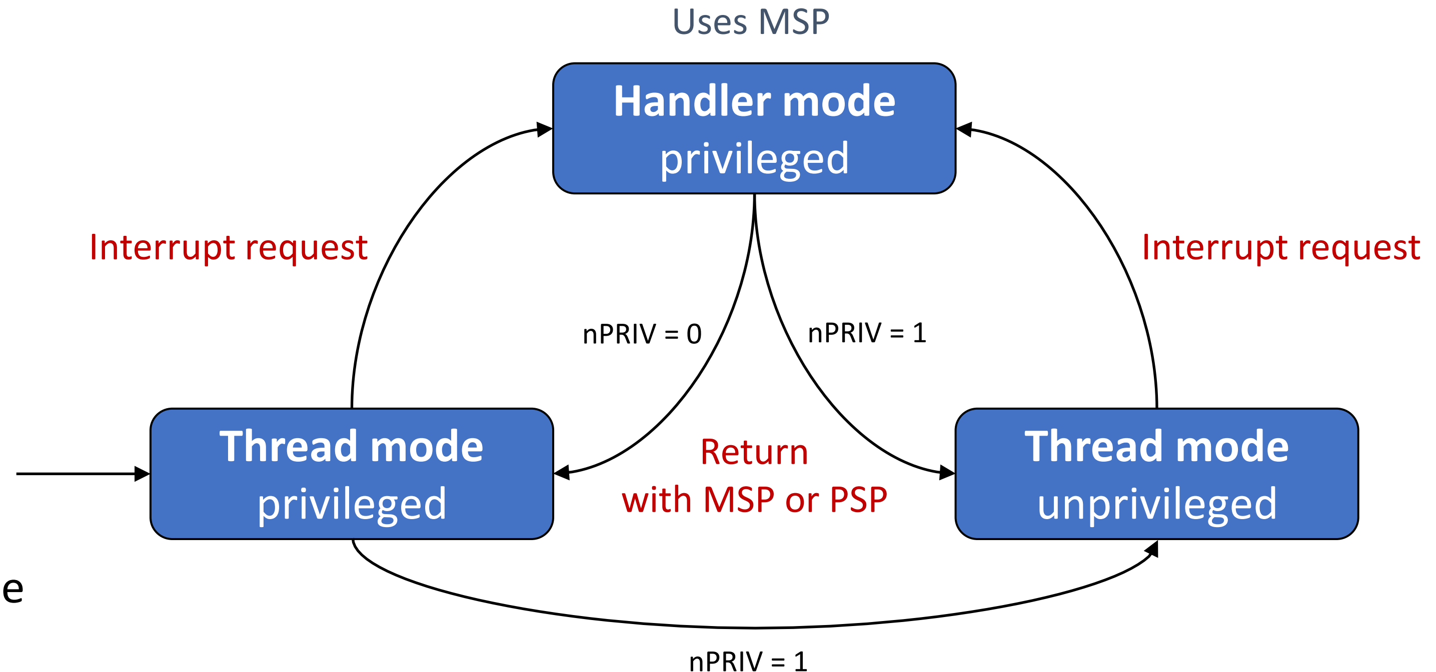
Example OS



HOGESCHOOL ROTTERDAM

Operation modes

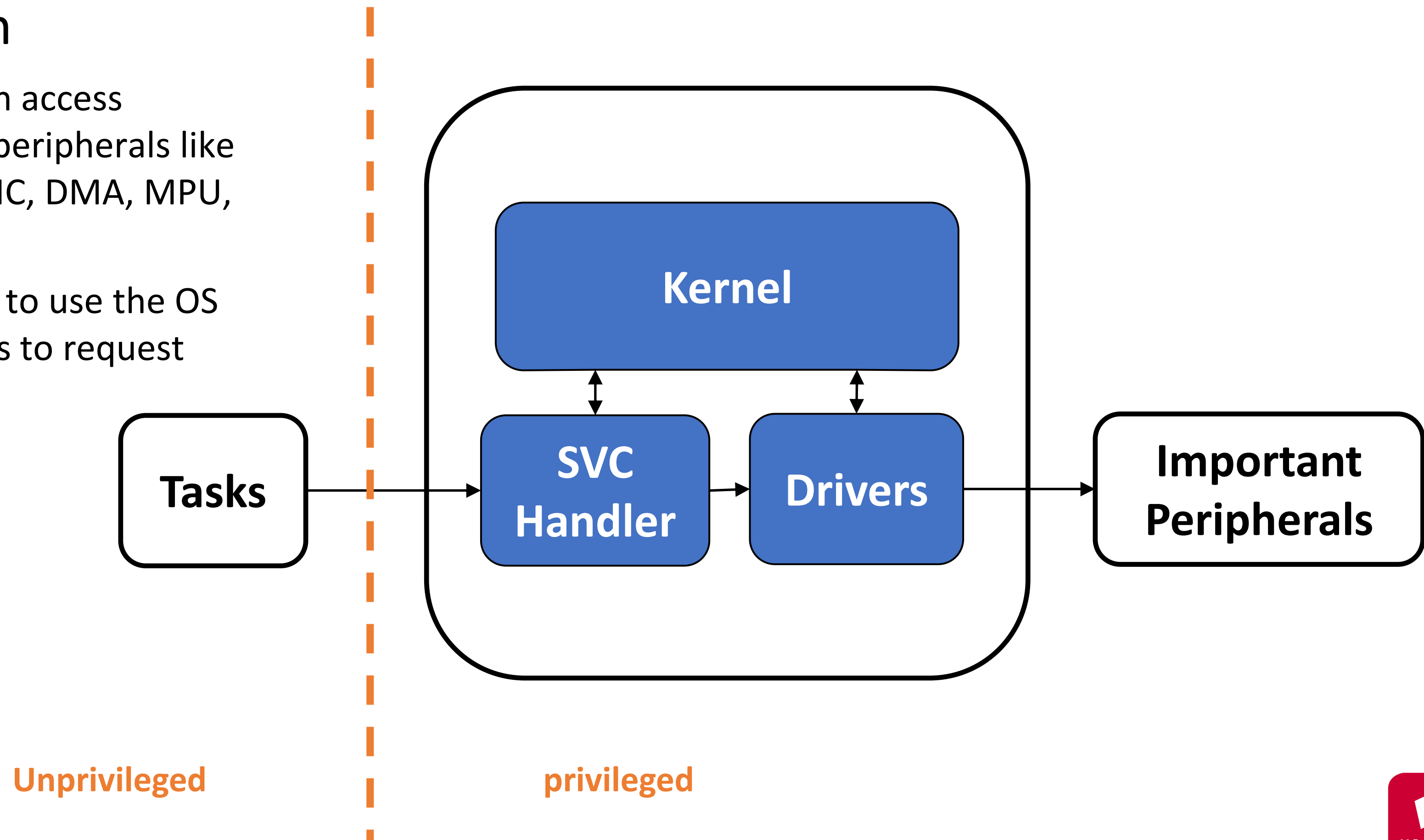
- Security by:
  - Separating stacks
  - Separating access levels between kernel and tasks.
    - nPRIV bit in CONTROL register
- Handler mode
  - Used by exceptions
- Privileged Thread mode
  - OS intialisation
  - OS kernel
- Unprivileged Thread mode
  - Used by tasks
  - Tasks have limited access to memory
  - MPU can be used to change access



HOGESCHOOL ROTTERDAM

Operation modes

- Abstraction
  - Only OS can access important peripherals like Systick, NVIC, DMA, MPU, etc
  - Tasks need to use the OS system calls to request changes



HOGESCHOOL ROTTERDAM

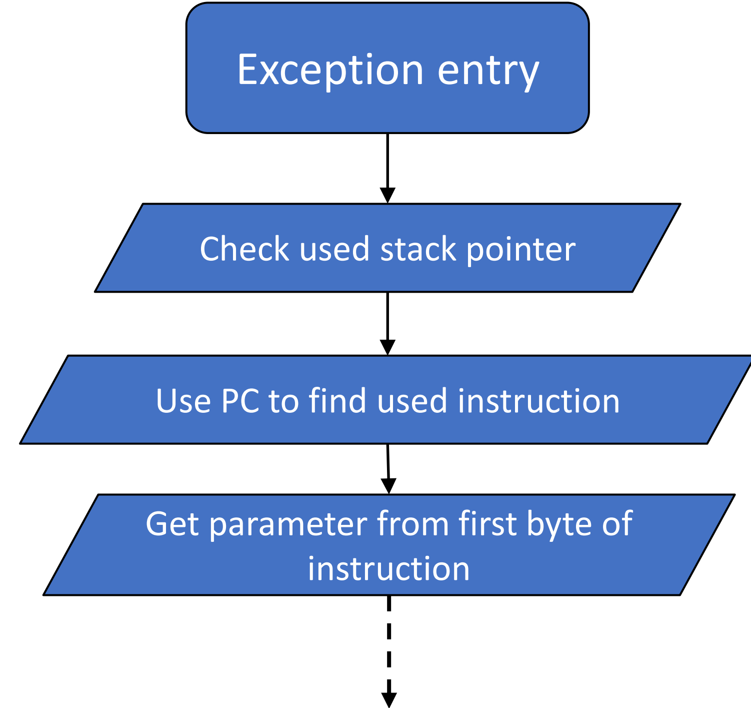
OS Interrupts

- Systick
  - Used by OS to periodically update everything
- PendSV (Pendable Service Call)
  - Used by OS to request a context switch
  - Software bit triggers this interrupt
  - Lowest priority
- SVC (SuperVisor Call)
  - Used by tasks to request a service from the OS

HOGESCHOOL ROTTERDAM

Supervisor call

- Assembly call
  - SVC #x
    - Parameter X defines which action the OS performs, first byte of instruction
    - Interrupt occurs immediately
    - You can get and return parameters with stacked registers like normal function call



HOGESCHOOL ROTTERDAM

PendSV - Context switch

- R0-R3,R12,LR,PC,XPSr get pushed to current stack
- Push {R4-R11} CPU registers to PSP, update stack pointer
- Pick new task and change PSP
- Pop {R4-R11} from new PSP, update stack pointer
- R0-R3,R12,LR,PC,XPSr get popped from current stack

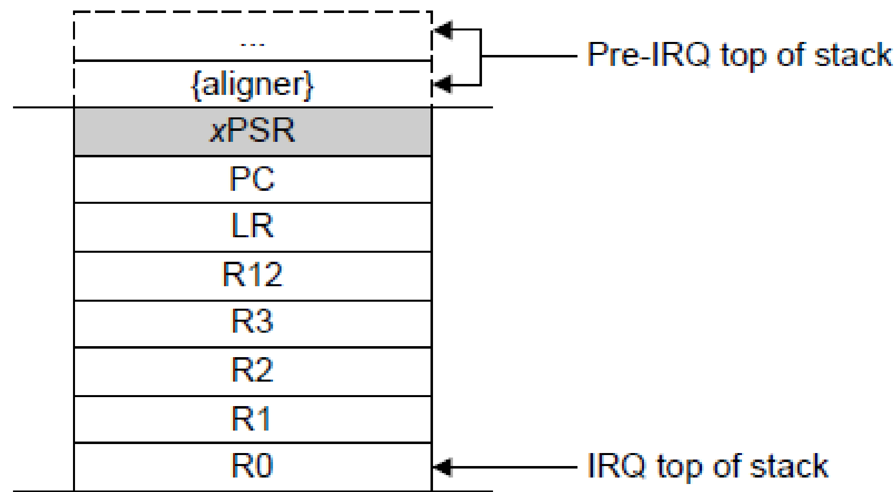
Switching context

```
while (1)
{
    //declare variables
    int a,b,c,d,e,f,g;

    //perform length calculation
    a=b*(c+d/e*f%g);

    //add to output queue
    addToQueue(a);
}
```

1. Enter exception (PendSV)
2. Save context (CPU registers) to stack
3. Switch stack to new task
4. Load context from stack to CPU
5. Leave exception using new

```
1  void task()
2  {
3      float samples[256];
4      float output[256];
5
6      while(1)
7      {
8          performFFT(samples);
9      }
10 }
```

Name	Value	Description
Core Registers		
PC	0x00000936	Program Coun
SP	0x20000640	General Purpo:
LR	0x00000733	General Purpo:
xPSR	0x21000000	Stores the stat
R0	0x00008000	General Purpo:
R1	0xE000E100	General Purpo:
R2	0x200002FC	General Purpo:
R3	0x00000000	General Purpo:
R4	0x00000002	General Purpo:
R5	0x00000000	General Purpo:
R6	0x00000000	General Purpo:
R7	0x20000648	General Purpo:
R8	0x00000000	General Purpo:
R9	0x00000000	General Purpo:
R10	0xA4420001	General Purpo:
R11	0x400FD108	General Purpo:
R12	0x400FD000	General Purpo:
R13	0x20000640	General Purpose Register 13 [Core]
R14	0x00000733	General Purpose Register 14 [Core]



Exception frame without floating-point storage

(x)= Variables | Expressions | Registers | Breakpoints

	Value	Description
Registers		
C	0x00000BC6	Program Counter [Core]
P	0x20000204	General Purpose Register 13 - Stac...
R	0x00000BD7	General Purpose Register 14 - Link...
PSR	0x61000000	Stores the status of interrupt enab...
0	0x00000000	General Purpose Register 0 [Core]
1	0xFFFFFFFF	General Purpose Register 1 [Core]
2	0x00000002	General Purpose Register 2 [Core]
3	0x00000003	General Purpose Register 3 [Core]
4	0x00000002	General Purpose Register 4 [Core]
5	0x00000000	General Purpose Register 5 [Core]
6	0x00000000	General Purpose Register 6 [Core]
7	0x20000648	General Purpose Register 7 [Core]
8	0x00000000	General Purpose Register 8 [Core]
9	0x00000000	General Purpose Register 9 [Core]
10	0xA4420001	General Purpose Register 10 [Core]
11	0x400FD108	General Purpose Register 11 [Core]
12	0x0000000C	General Purpose Register 12 [Core]
R13	0x20000204	General Purpose Register 13 [Core]
R14	0x00000BD7	General Purpose Register 14 [Core]

Memory Protection Unit

- Can set memory locations
  - Inaccessible
  - Read only
  - Non-executable
- Possible use-case in RTOS
  - Limit access to certain peripherals
  - Only allow access to own task memory (and thus stack)
    - Downside?
  - Make RAM segments non-executable to prevent injection-type attacks

[FreeRTOS-MPU - ARM Cortex-M3 and ARM Cortex-M4 Memory Protection Unit support in FreeRTOS]

[Why ARM MPU Has Become an Outcast in Embedded Systems]

HOGESCHOOL ROTTERDAM

Pre-emptive scheduling

Priority based

- Scheduler decides and update states of tasks
- When high priority task comes alive, it interrupts lower priority tasks
- When all tasks are suspended, the idle task can run

Round robin

- Every task gets equal CPU time
- When all tasks are suspended, the idle task can run

Demo

- Instructor demonstrates algorithms

HOGESCHOOL ROTTERDAM

Problems with pre-emptive scheduling

Starvation

- Low priority tasks don’t get cpu time
  - Possible solution: Aging

Sharing resources

- Tasks can’t use hardware ‘simultaneously’
- Waiting for hardware to come available can cause deadlock or priority inversion
  - Next week

HOGESCHOOL ROTTERDAM

Walkthrough VersdOS

HOGESCHOOL ROTTERDAM

Next Week

Free-RTOS

- What is it
- Problems and challenges with
- Threads and IPC (Inter Process Synchronization)
- POSIX API overview

Read assignment 5 before next week's lesson!

HOGESCHOOL ROTTERDAM

Aan de slag!

Aan de slag met [Opdrachten_Week_4.pdf]