EMBEDDED SYSTEMS

BASED ON CORTEX-M4 AND THE RENESAS SYNERGY PLATFORM

2020

PROF. DOUGLAS RENAUX, PHD
PROF. ROBSON LINHARES, DR.
UTFPR / ESYSTECH

RENESAS ELECTRONICS CORPORATION

BIG IDEAS FOR EVERY SPACE RENESAS

13 – CONCURRENT PROGRAMMING

- Tasks
  - Processes vs Threads
  - Context Switching
  - Scheduling
- Inter-task Communications and Synchronization
- Caveats

UNDERSTANDING CONCURRENCY

- Consider a single-person company. Suppose you could specify his activities by writing a program-like script. Wouldn’t it be very complex? Full of interleaved chores that would be hard to specify in a single script?

source: pixabay.com

UNDERSTANDING CONCURRENCY

Now, consider a simple embedded system with:

- 3 serial ports operating at 115 Kbps (aprox. 1 char every 87us);
- USB, requiring processing every 125 us for packets with about 1KBytes;
- IHM: touch screen plus LCD;
- activities implemented in SW:
  - A1: every 2 ms – process data from the touch screen;
  - A2: every 7 ms – process USB data;
  - A3: every 500 us – manage the USB protocol;
  - A4: every 100 ms – manage the menu system.

Wouldn’t it be very hard to program a single sequential code to execute all these tasks, even more if several possible interleavings could occur?

UNDERSTANDING CONCURRENCY

As embedded systems increase constantly in complexity and code size, this complexity can be better managed if a single program (responsible for all activities) could be divided into many smaller programs, each one responsible for a single activity. Each one of these small programs is called a **task**. The multiple tasks that compose a concurrent program execute concurrently and cooperate among them to achieve the desired functionality.

It is TEAM WORK!!

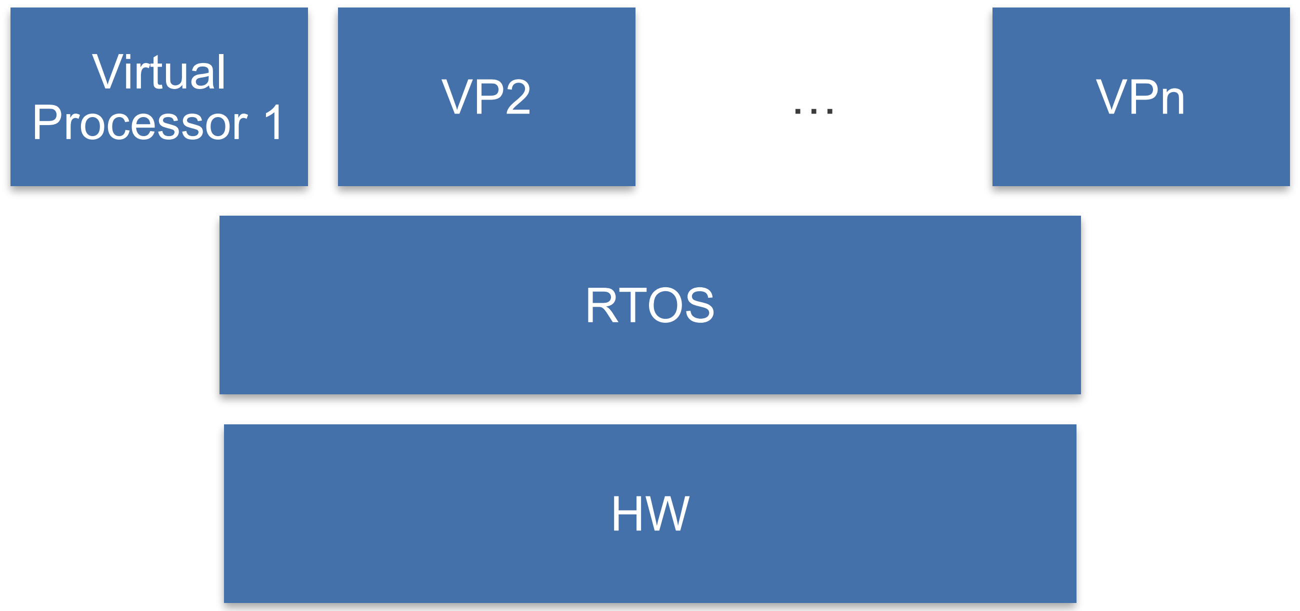
UNDERSTANDING CONCURRENCY

How can multiple tasks execute concurrently on a single processor?

Answer: each task will execute on a “virtual processor”. The combined performance of all virtual processors is about the same as the performance of the actual processor, as the virtual processor **share** the physical resources: processor, memory and peripherals.

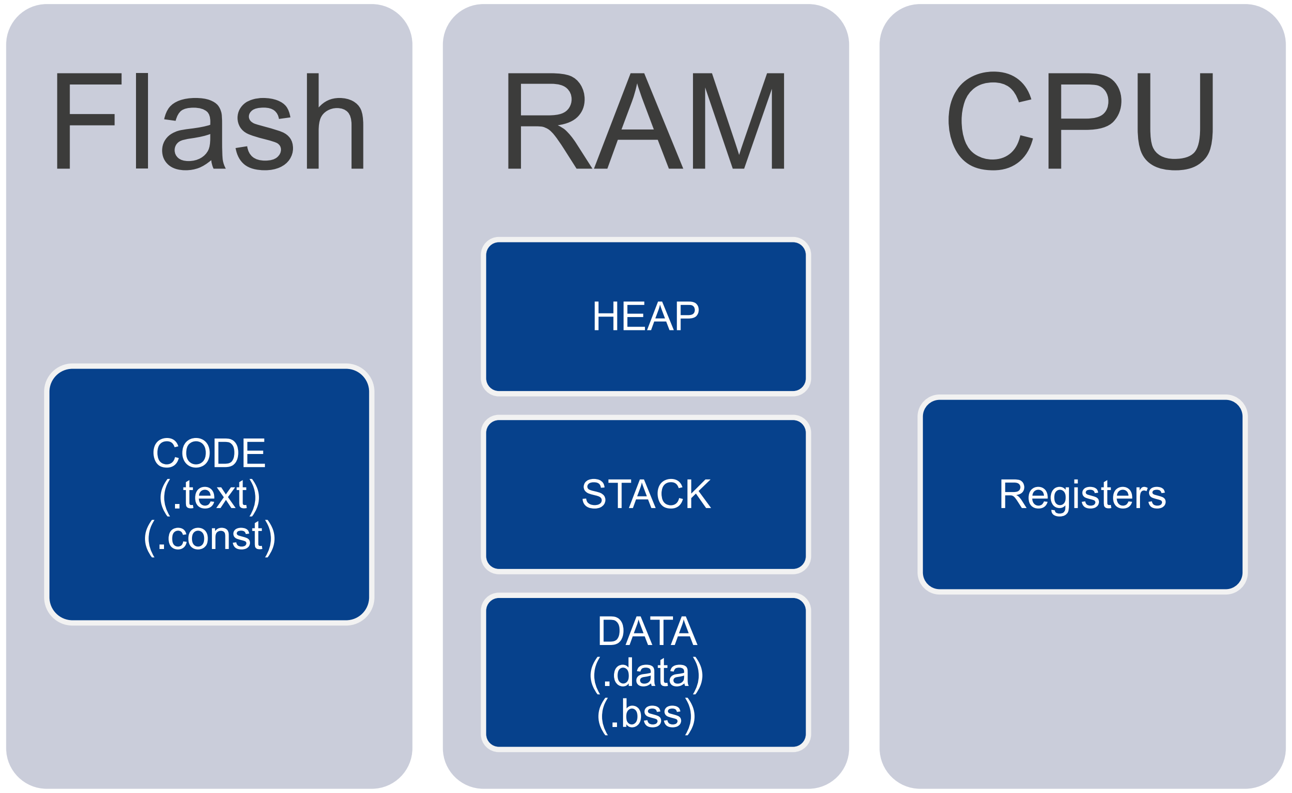
UNDERSTANDING CONCURRENCY

The sharing of the actual hardware (processor, memory, peripherals) is managed by an embedded operating system (or RTOS - Real-Time Operating System).



UNDERSTANDING CONCURRENCY

The storage regions of a single sequential program (e.g a C-program) are:



MULTITHREADING

For simplicity and to reduce the usage of computational resources, RTOS for MCUs typically rely on multithreading to implement concurrency.

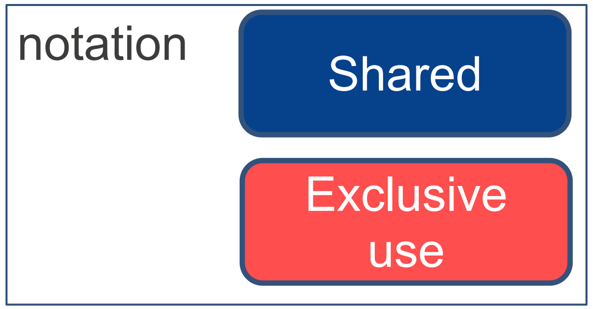
A **thread** is a program segment that executes concurrently to other threads (program segments). Hence, each thread is characterized by its own PC (program counter), its own set of processor registers and its own stack, while sharing the other memory regions with the other threads.

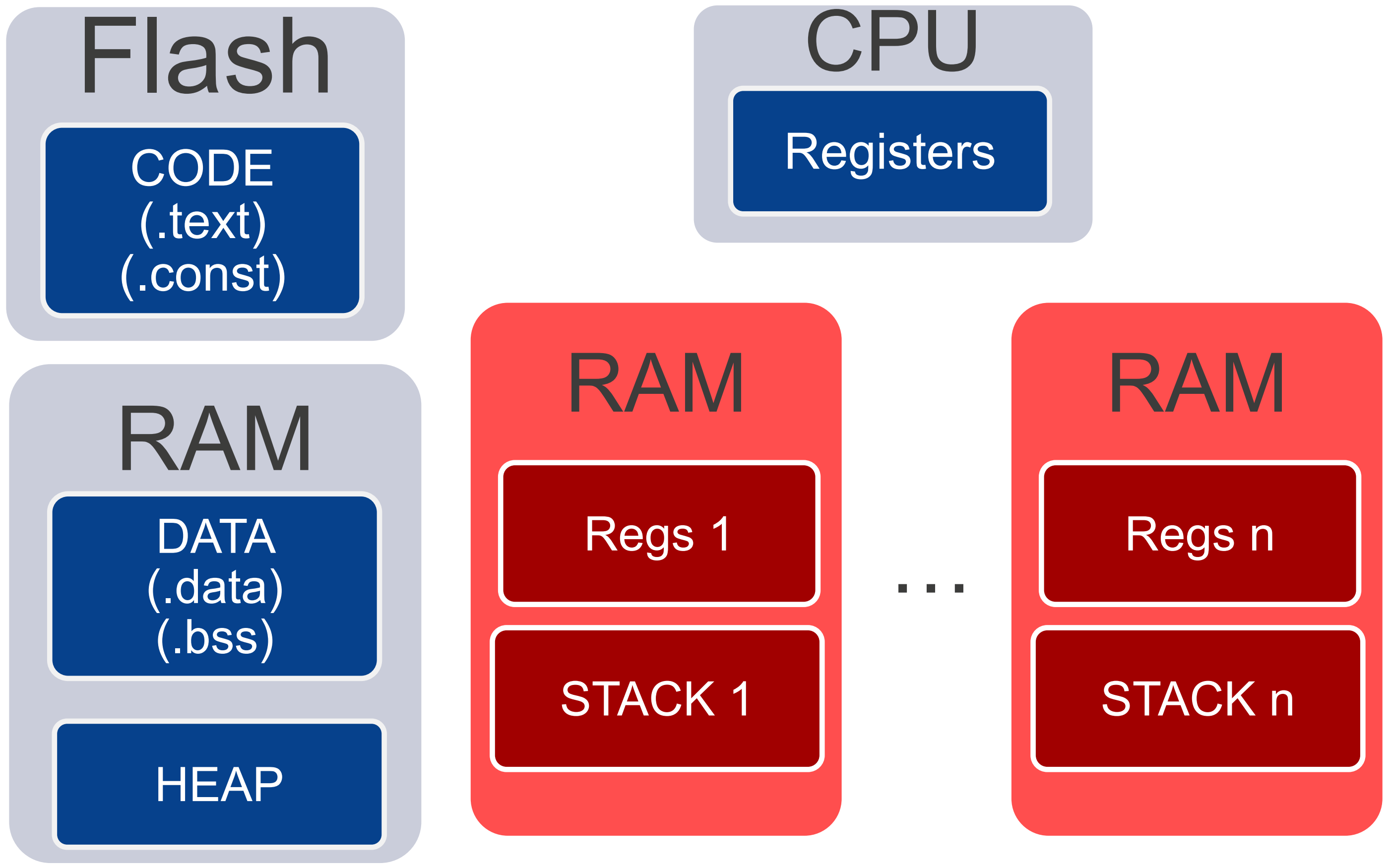
On MCUs, each task (abstract concept) is implement by one thread.

MULTITHREADING

Some sections of RAM (shown in red) are of **exclusive** use of a thread. These sections hold the stacks and a copy of the set of processor registers.

Other sections of Memory (shown in blue) are **shared** among all threads. While this sharing provides efficient access to shared data, it does not provide means of protection among threads.





MULTITHREADING

Differences between threads and processes:

- On processors with MMUs (Memory Management Units), it is possible to create an exclusive addressing space for each task. This type of implementation is called **process**. In a process, it is possible to host several threads, hence, there are single-threaded processes and multi-threaded processes.
- On processors without MMUs, all tasks share the available memory. This type of implementation is called **thread**.
- A **task** is a logical concept that can be implemented by a process or by a thread.

MULTITHREADING – CONCEPTS

Context Switch

How do multiple threads share a single processor?

- An RTOS manages the physical resources (processor, memory, peripherals).
- A context switch consists of saving the state of the processor when one thread is executing and restoring the state of another thread:

  1. Task A is executing;
  2. All CPU registers are saved onto the stack of Task A;
  3. The RTOS executes and selects another task to execute: Task B. The criteria for selecting another task is defined by the scheduling policy;
  4. The state of task B is restored from Task B stack onto the CPU registers. Execution proceeds on Task B's code and using Task B's stack.

MULTITHREADING – CONCEPTS

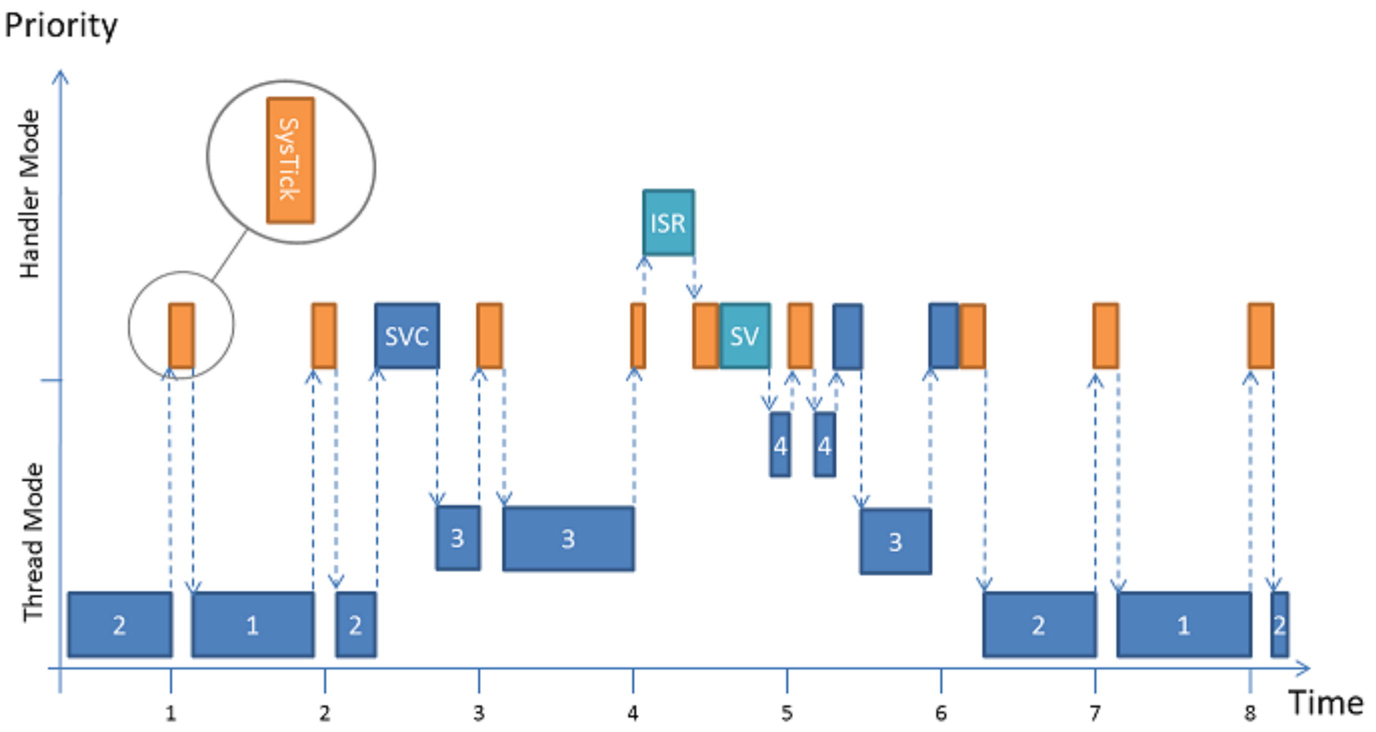
Preemptive vs Non-Preemptive RTOS

- A preemptive RTOS has control of the processor during all times. It releases the processor to a thread and at any time may get back the control of the processor to releases to another thread. This is the case when a higher priority thread becomes ready to run.
- When a non-preemptive RTOS releases control to a thread, it is unable to regain control of the processor until that thread, voluntarily, releases the processor back to the RTOS.

MULTITHREADING – CONCEPTS

Task Priority

- Each task is created with a defined priority level, which typically can be changed during execution. When a task of higher priority than the running task becomes ready, a priority-based preemptive scheduler will interrupt the running task and context switch to the higher priority task. Once this task releases the processor, the preempted task can resume its execution.



source: ARM, CMSIS-RTOS specs
http://www.keil.com/pack/doc/CMSIS_Dev/RTOS2/html/theory_of_operation.html

MULTITHREADING – CAVEATS

If all tasks in a concurrent program are independent, then the only control that is needed is the allocation of the processor to the tasks (**scheduling**).

However, if tasks share resources (memory regions or peripherals) then an adequate control of resource sharing must be performed to guarantee **exclusive access to shared resources** and to avoid **deadlocks** and **priority inversion**.

Renesas.com