Practical Intro to Real Time Operating Systems

Mohamed Abdelazeem

Agenda

1. RTOS vs GPOS
2. Scheduler
3. Memory Management
4. Inter-task communication
5. Task synchronization
6. Exercises

RTOS Definition

What is Real-Time OS?

- A real-time operating system (RTOS) is an operating system with two key features:
- Predictable.
- Deterministic. (No Random Execution Pattern)

- You can define very clearly what the tasks are in the system, what their scheduling priority and requirements are, and then you will get consistent behavior.

General Purpose OS

- Not Deterministic
- Un-predictable response times. *"we'll try to do what you ask"*
- Designed to perform non-time-critical general tasks.
- These systems' scheduling isn't always prioritized.
- A lower-priority process can be executed first. The task scheduler uses a fairness policy, allowing the overall high throughput but not ensuring that high-priority jobs will be executed first.

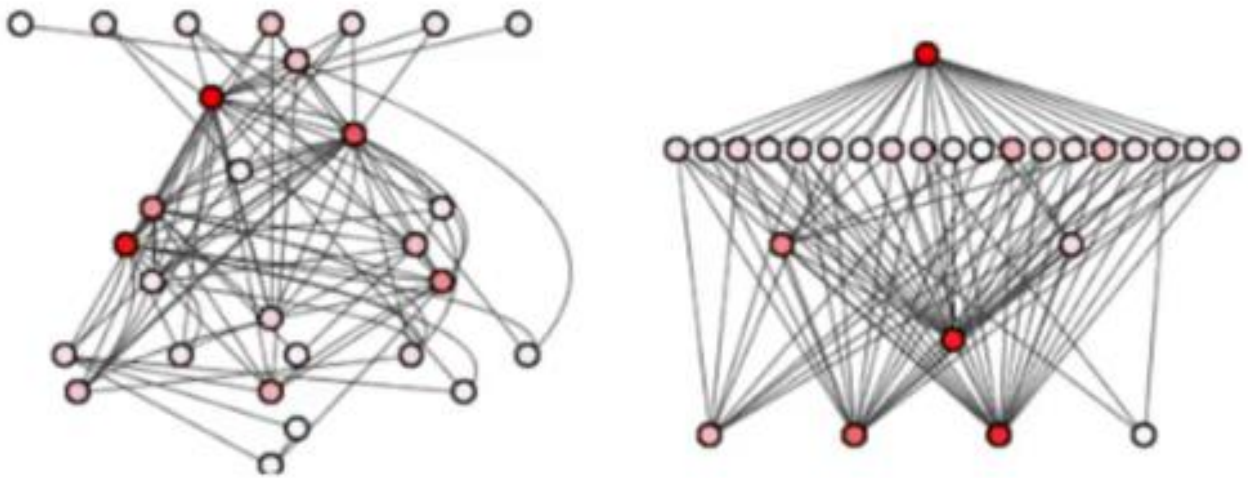
Why Do we need RTOS?

RTOS Advantages:

- Expandability.
- Maintainability.
- Portability.
- Security.

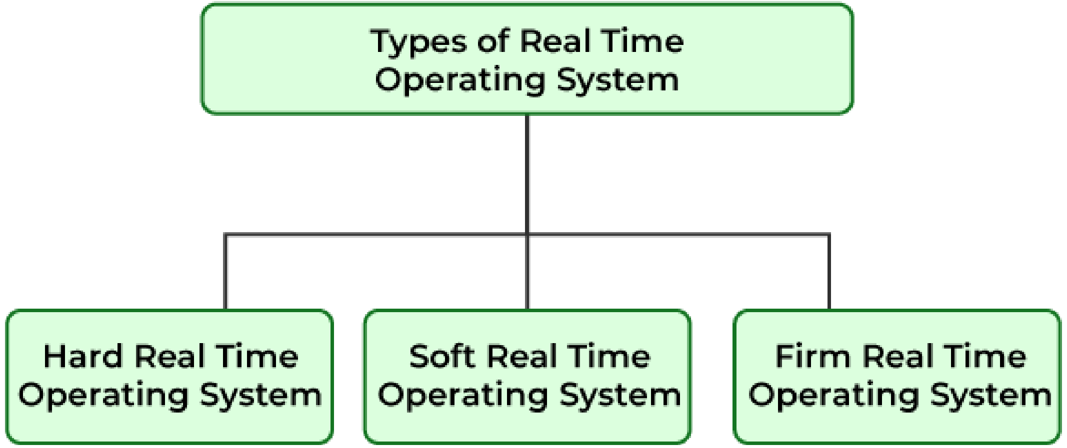
RTOS Disadvantages:

- Increased footprint:
  - There's extra code space and RAM used by the RTOS itself

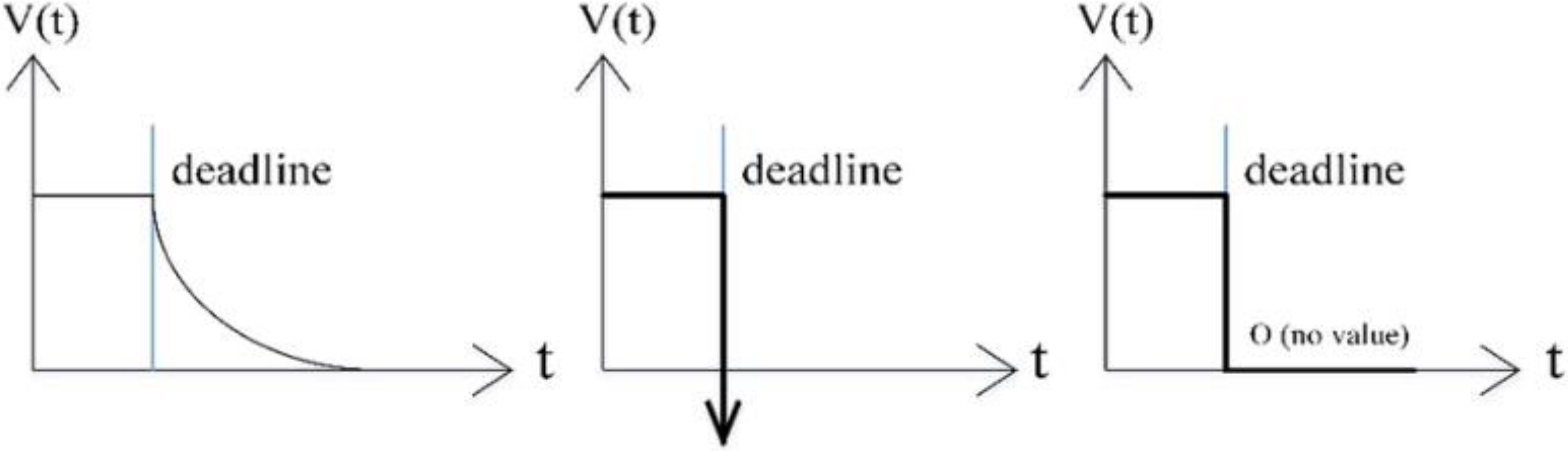


Real-Time Systems

- Hard real-time : Missing a deadline results in failure (e.g., safety-critical systems).
- Firm real-time : Missing a deadline leads to useless results but not system failure (e.g., certain types of data processing).
- Soft real-time: Missing deadlines is acceptable and affects performance but does not lead to failure (e.g., multimedia applications).

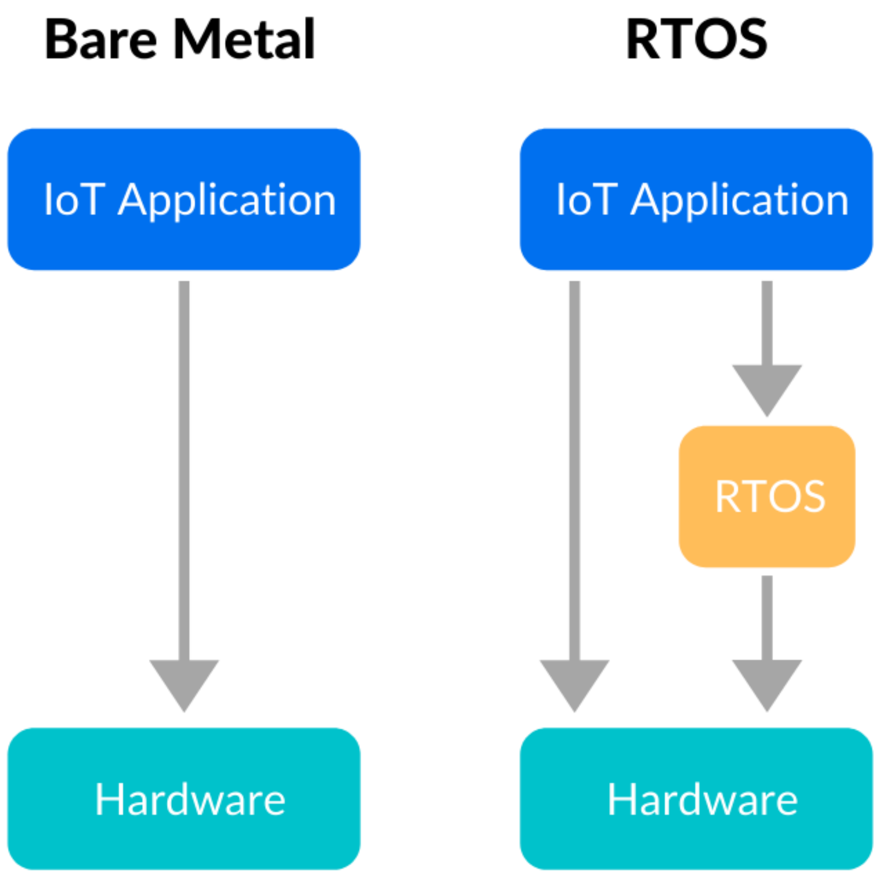


Real-Time Systems

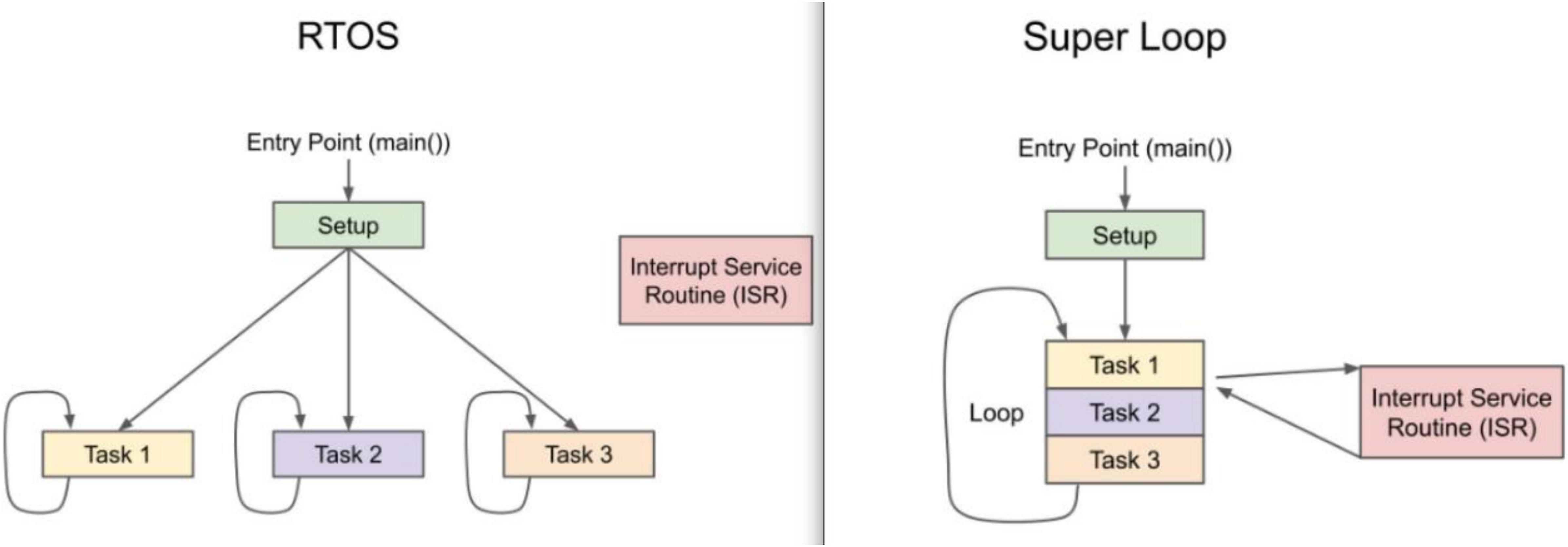


RTOS vs Bare-metal

- Used in resource-constrained systems where RTOSes are not available.
- Advantages:
  - Low overhead.
  - Simplicity.
- Disadvantages
  - Complexity in Multitasking.
  - Maintenance
  - Code is less portable.



RTOS vs Bare-metal

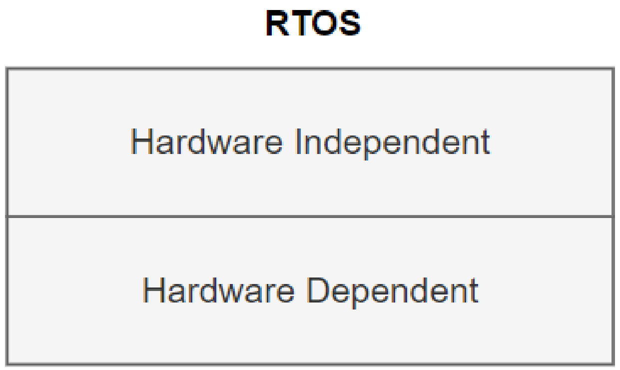


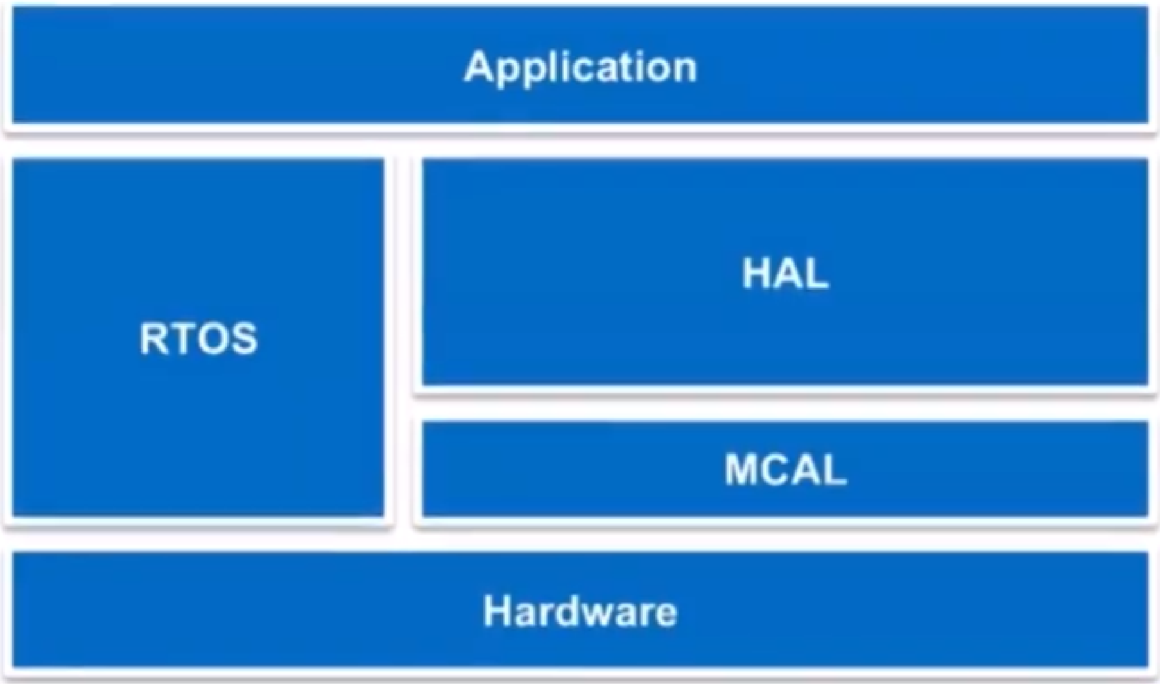
RTOS vs Bare Metal vs GPOS

	Bare Metal	RTOS	OS
Good for small devices	✓	✓	✗
Level of application hardware control	complete	medium	none
Ability to multitask	✗	Capable, but not with 100% separation	✓
Overhead	None	Minimal	High
Efficient memory usage	✓	✓	✗
Community support	✗	✓	✓
Scalable/Portable	✗	Medium	Easily portable

RTOS Architecture

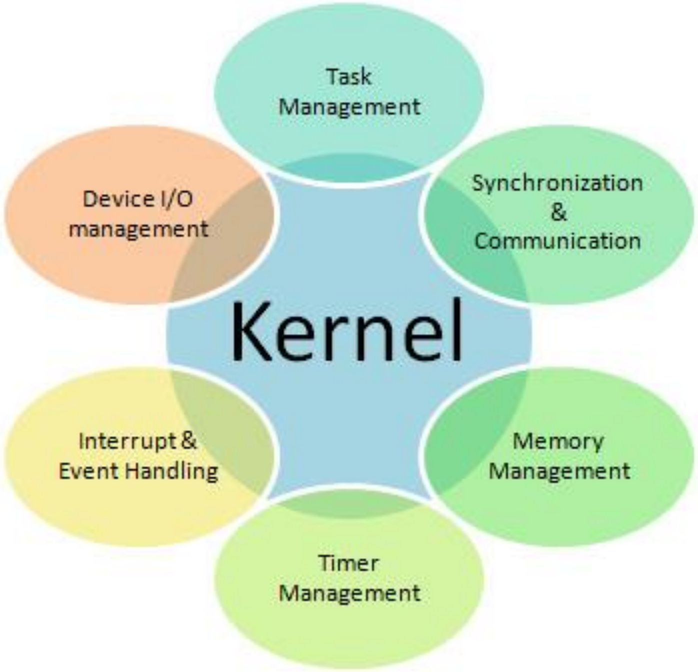
- RTOS needs access Hardware for:
  - Hardware Timers.
  - Hardware registers.





RTOS components

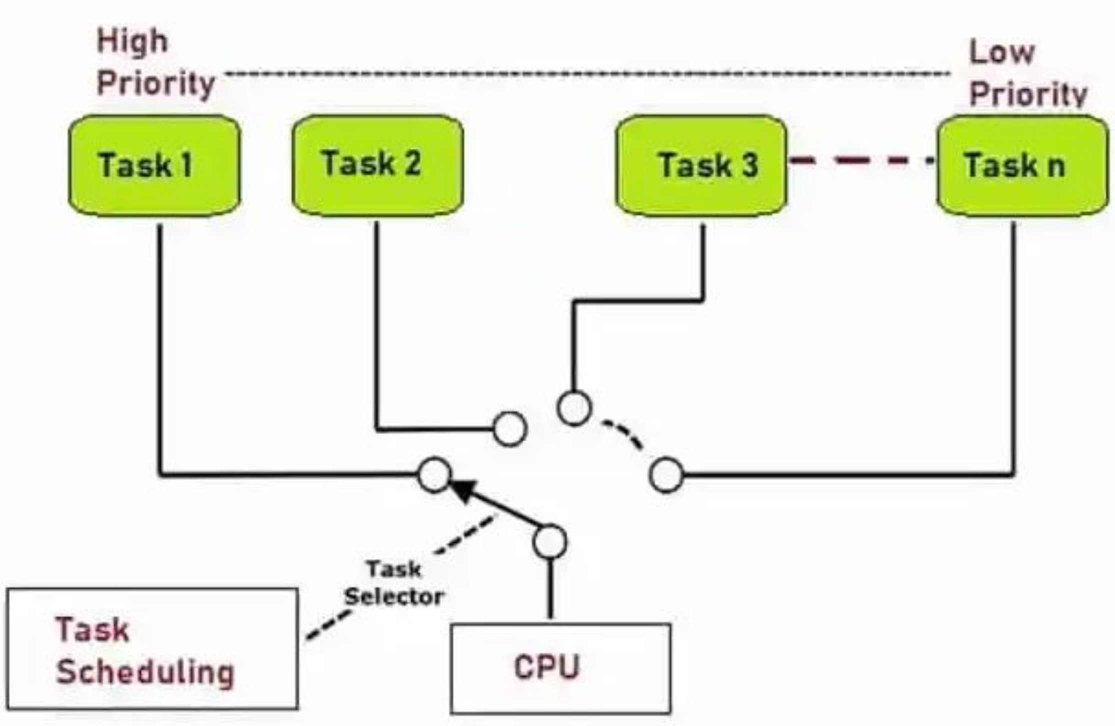
RTOS Components



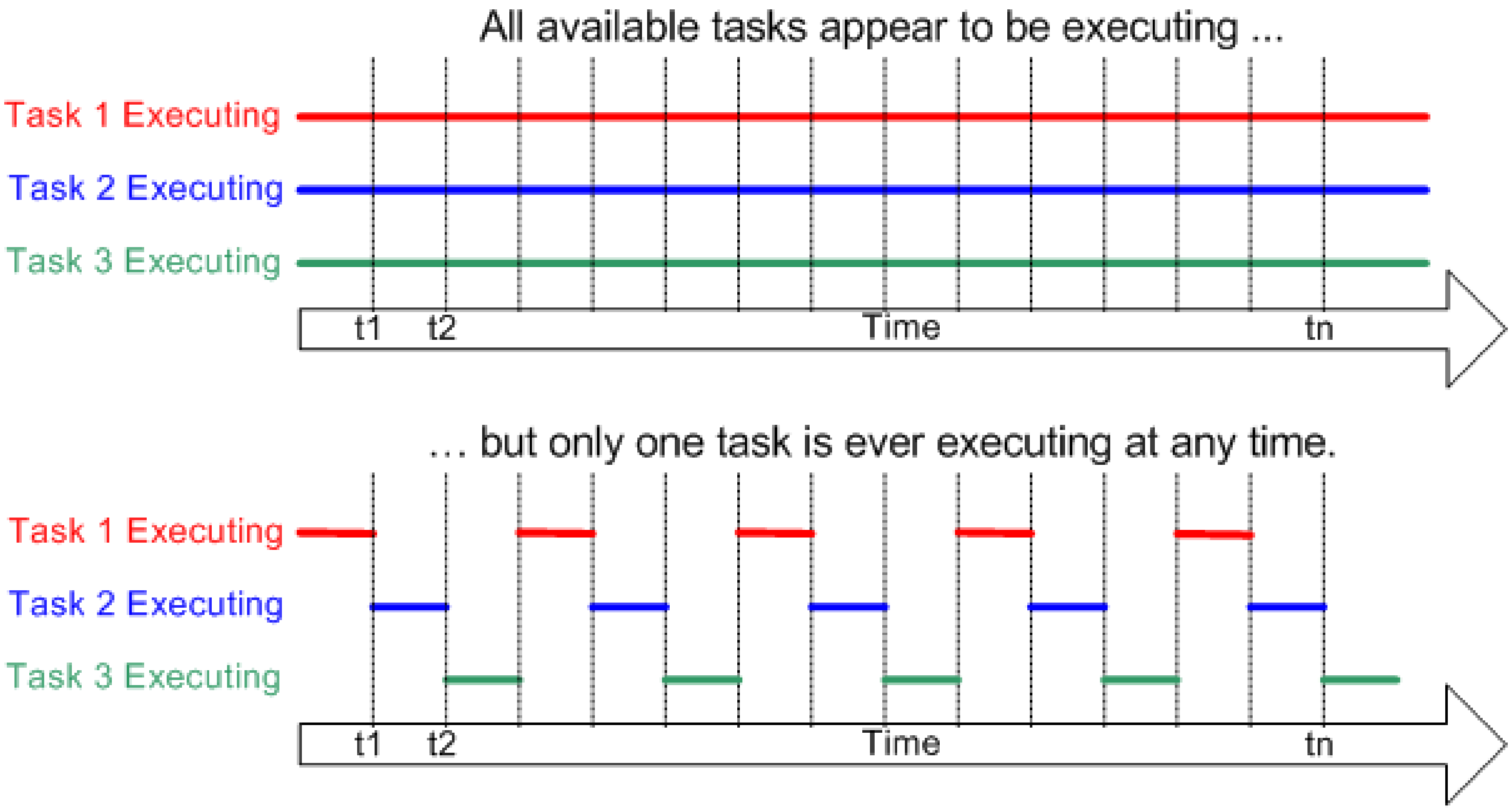
RTOS Scheduler

Scheduler?

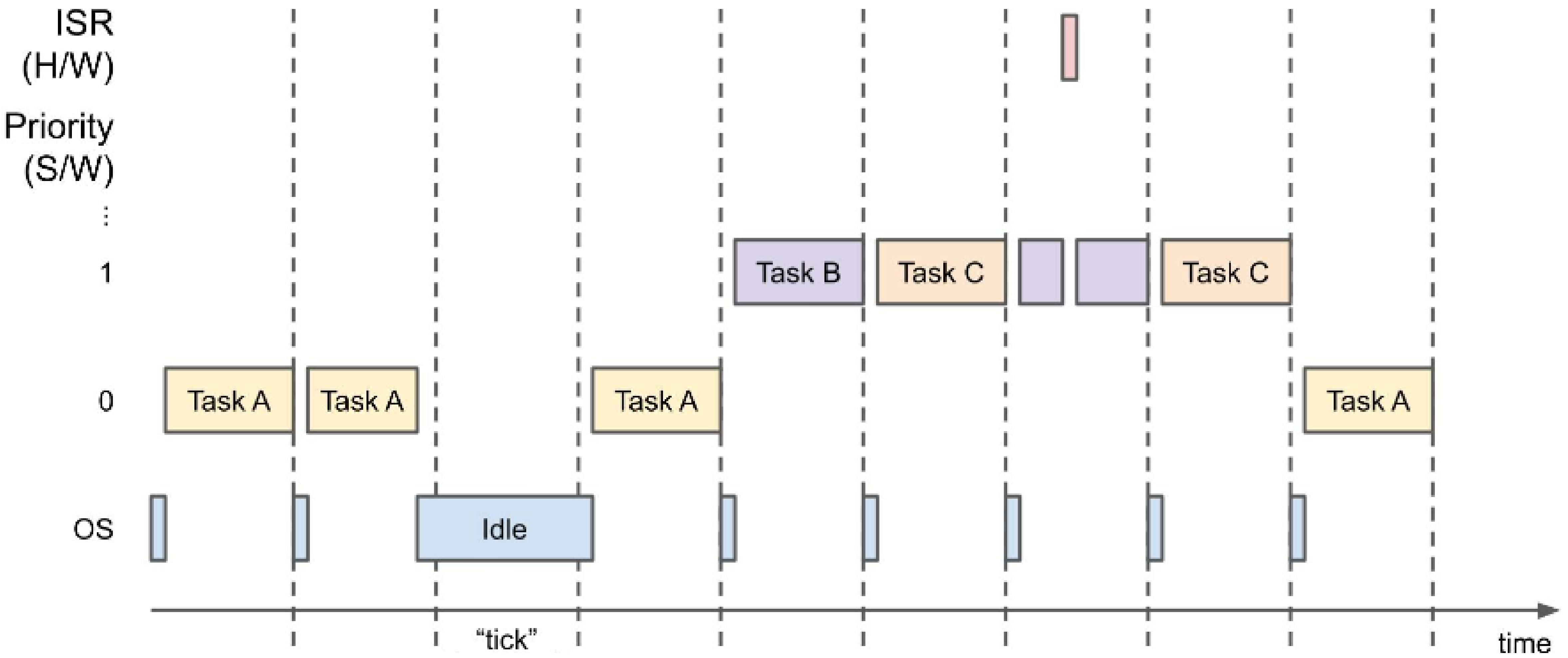
- The **scheduler** is an algorithm determining which task to execute.
- It selects one of the task being ready to be executed (in READY state).



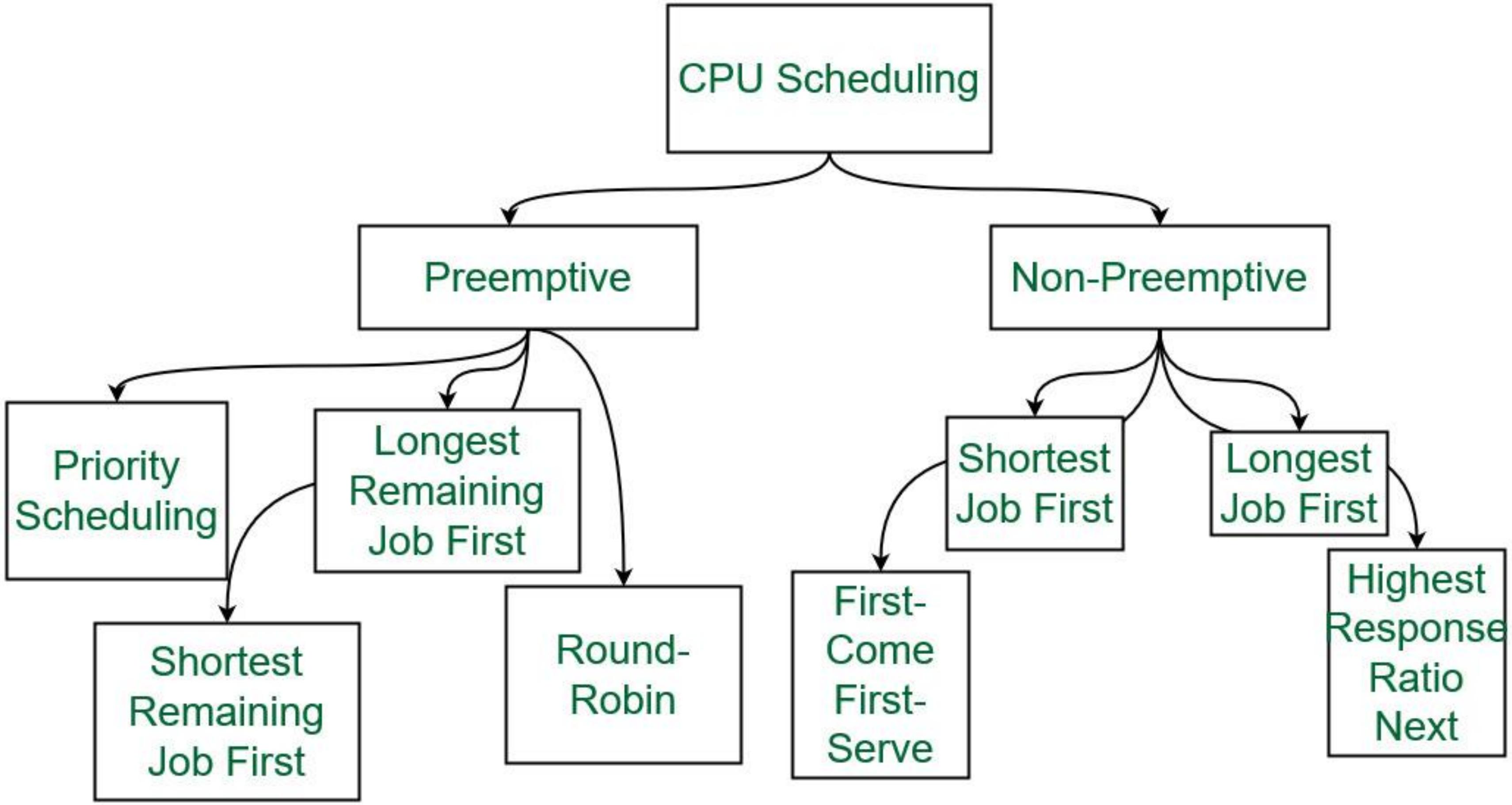
Tasks Execution



Tasks Execution



Scheduling Policies



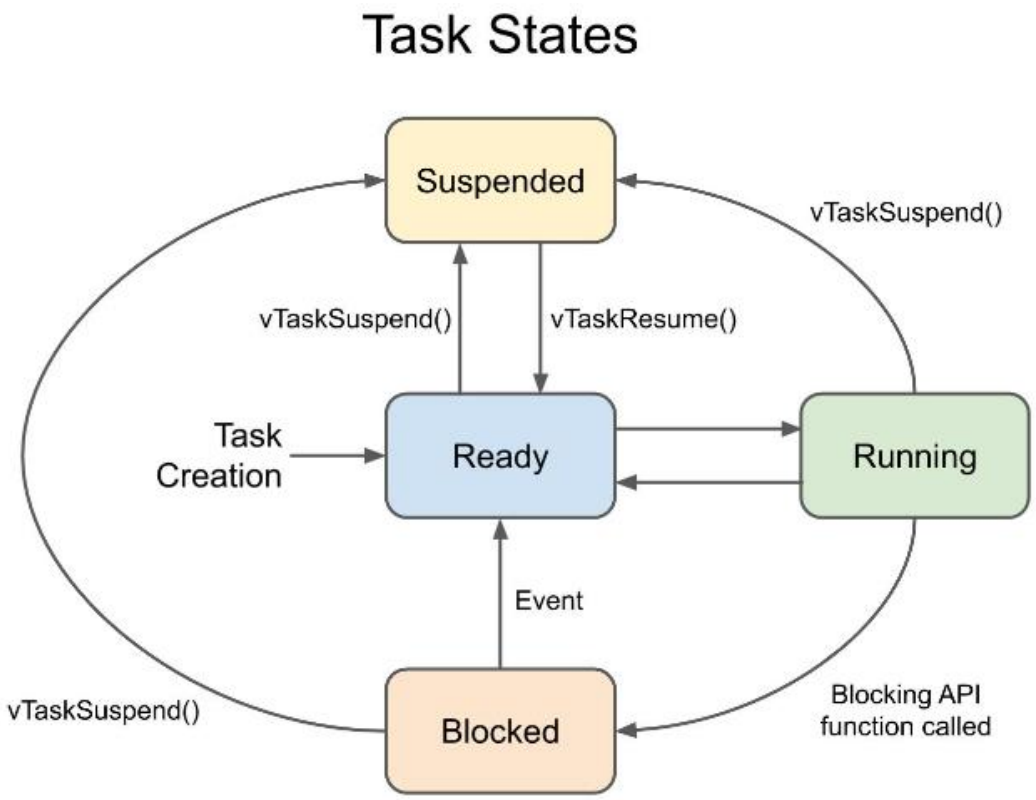
What is a Task?

- It is C function:
  - It should be run within infinite loop, like:

```
for(;;)
{
/* Task code */

}
```

- It is created and deleted by calling API functions.
- It can be in one of 4 states (RUNNING, BLOCKED, SUSPENDED, READY)



Task structure

```
void FirstTask(void const * argument)
{
 /* task initialization */

 for(;;)
 {
  /* Task code */
 }

 /* we should never be here */
```

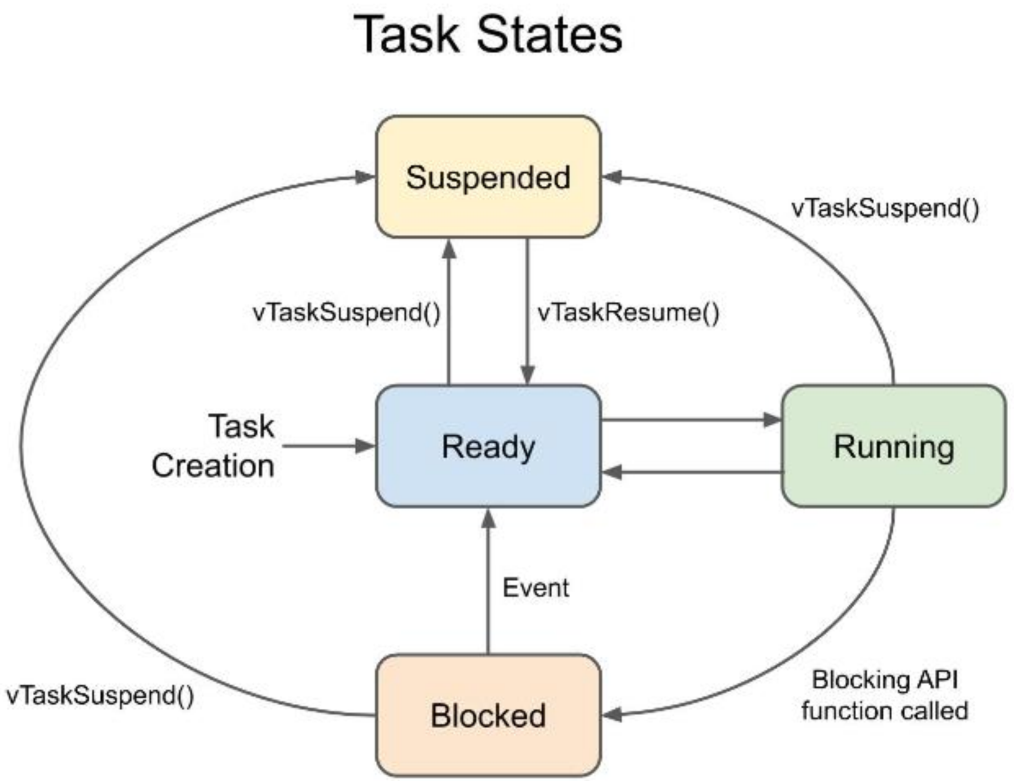
Run once at first run of each task instance

Run when task instance is in RUN mode

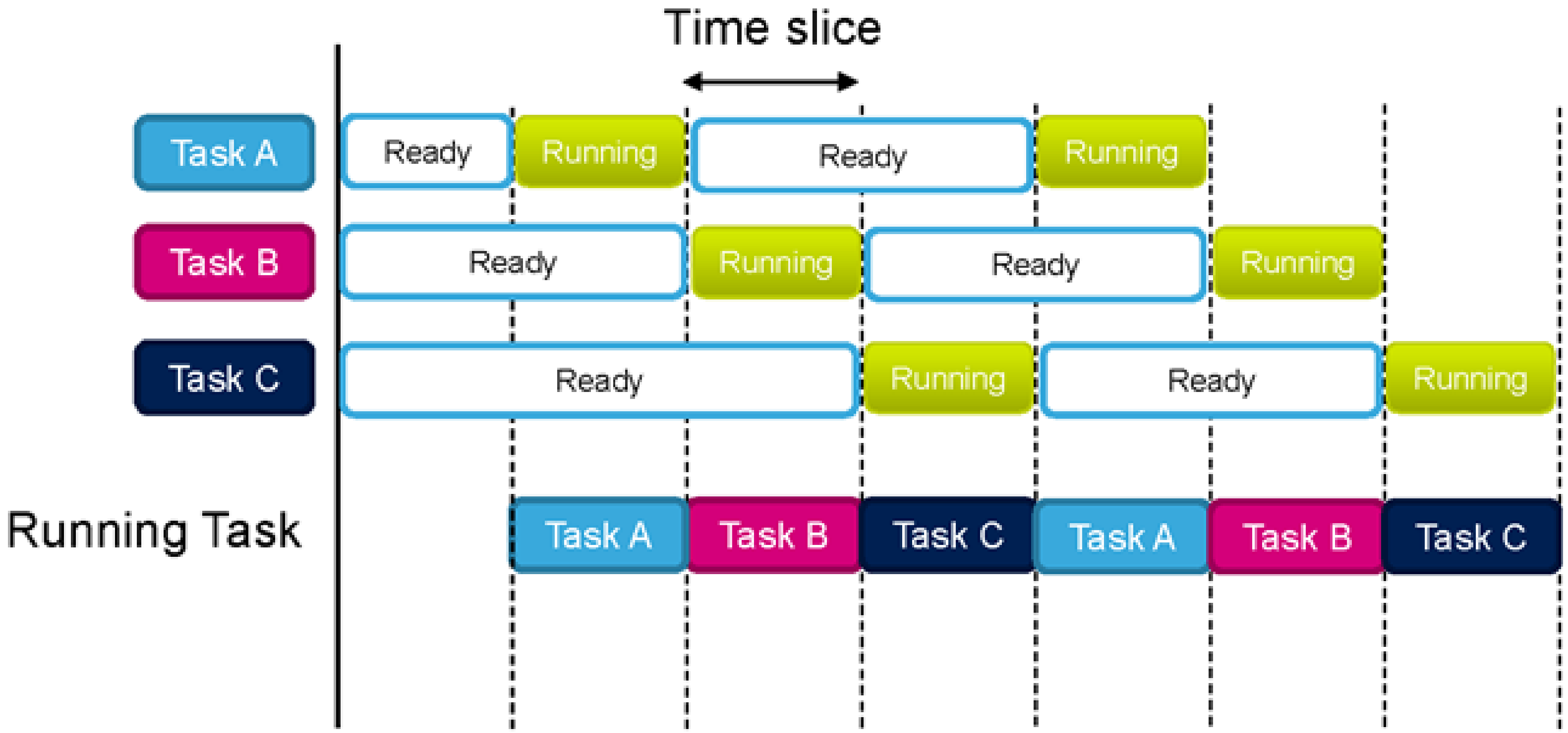
Should be never executed.

What is a Task?

- **Ready**
  • Task is ready to be executed but is not currently executing because a different task with equal or higher priority is running
- **Running**
  • Task is actually running (only one can be in this state at the moment)
- **Blocked**
  • Task is waiting for either a temporal or an external event
- **Suspended**
  • Task not available for scheduling, but still being kept in memory.

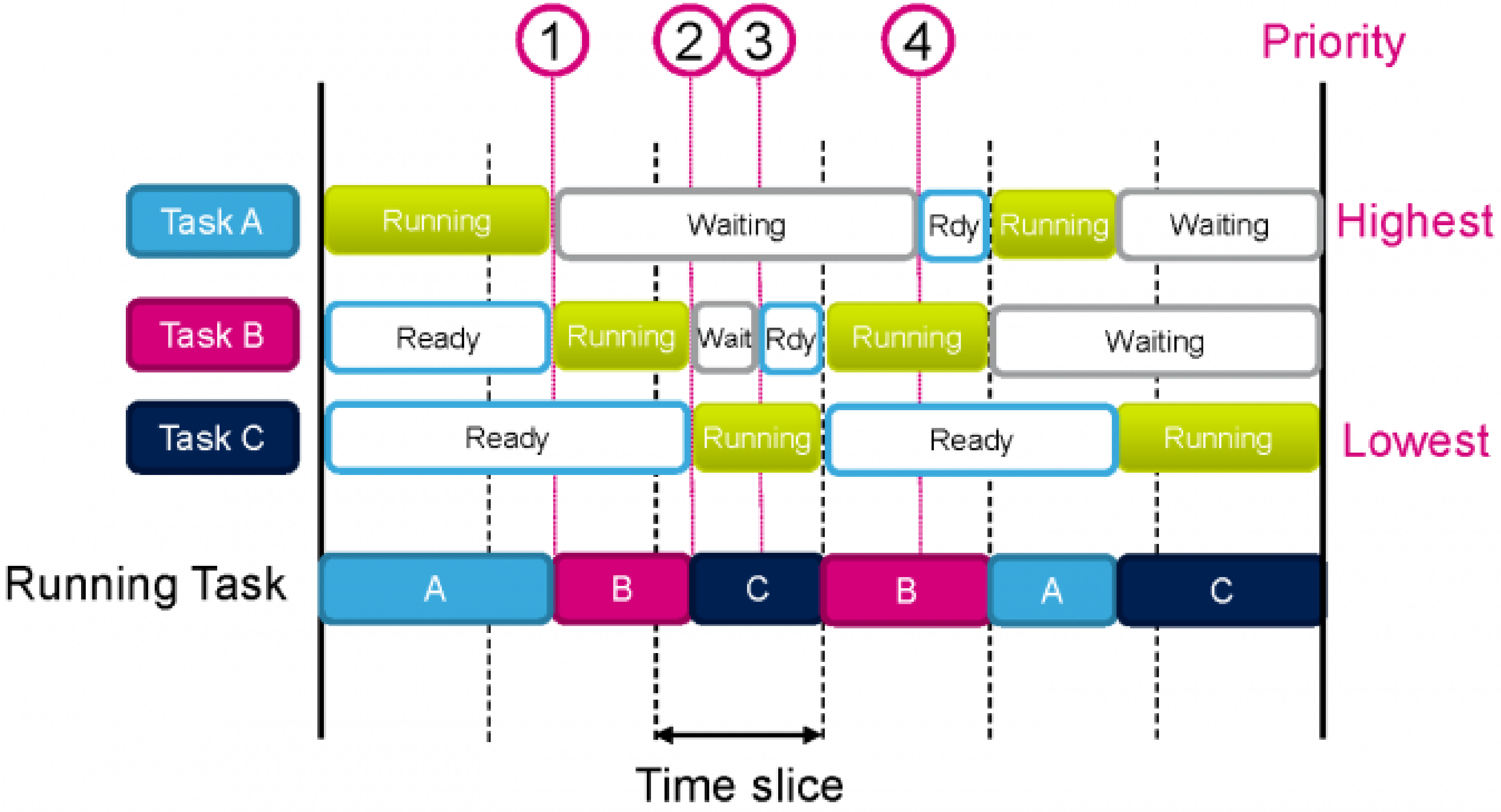


Round-robin Scheduler



Round-robin scheduler

Pre-emptive/ Priority-based Scheduler

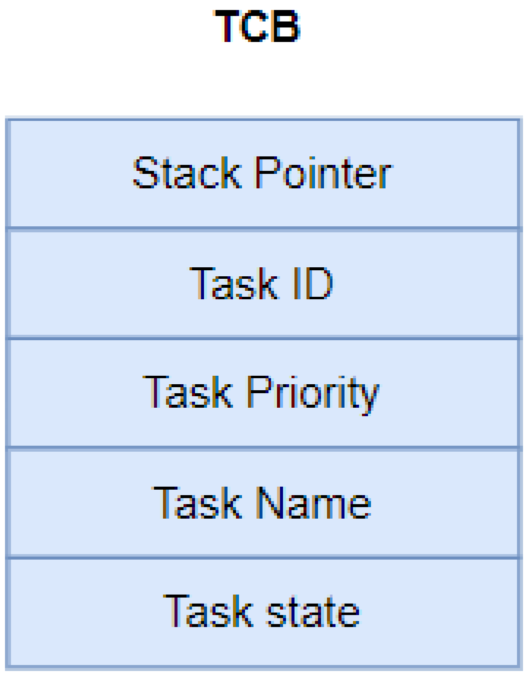


Priority-based scheduler

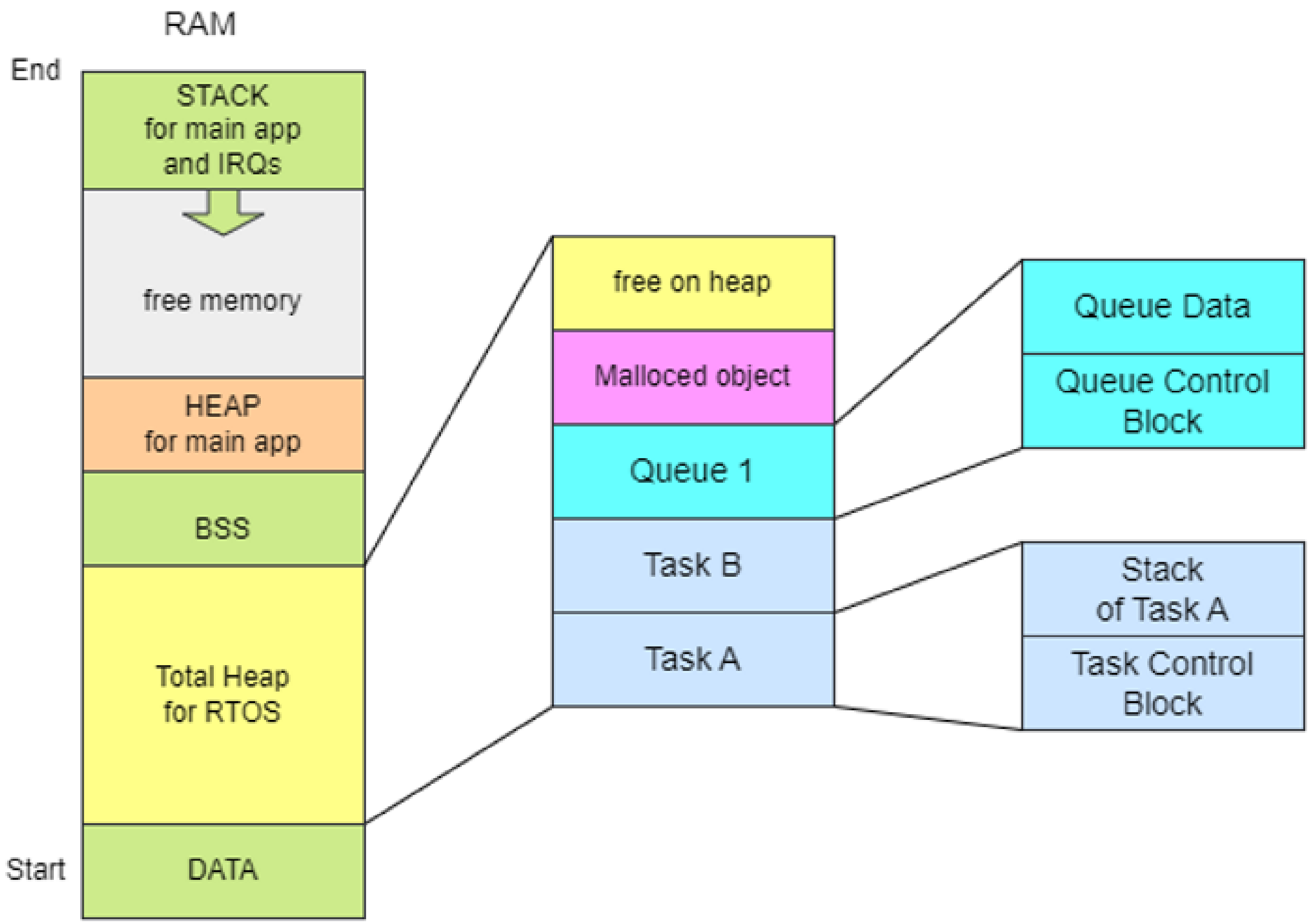
Task Control Block

- How ROTS keep track of the tasks?

- It a data structure used by the kernel to maintain information about a task.

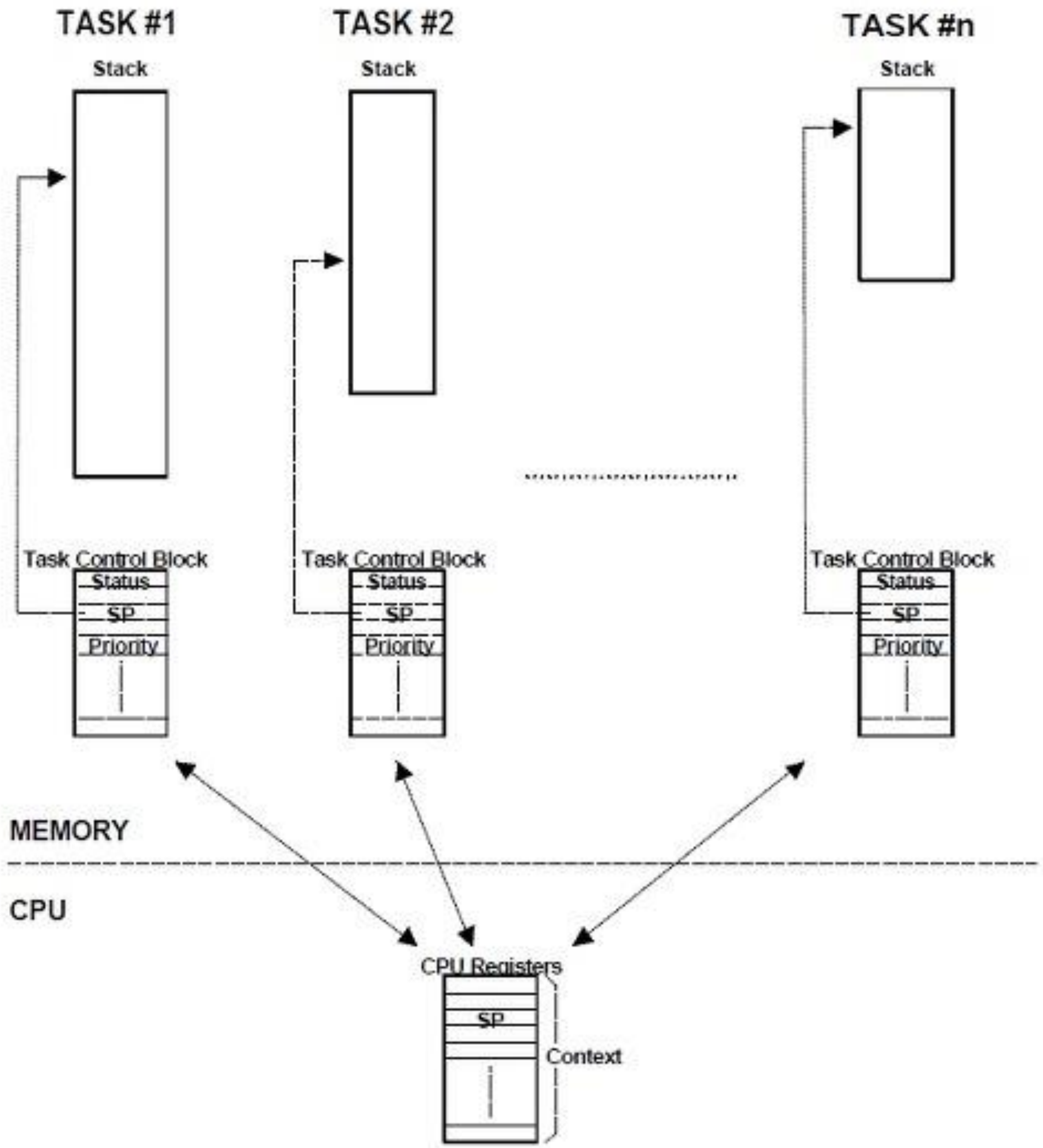


Memory Heap In RTOS

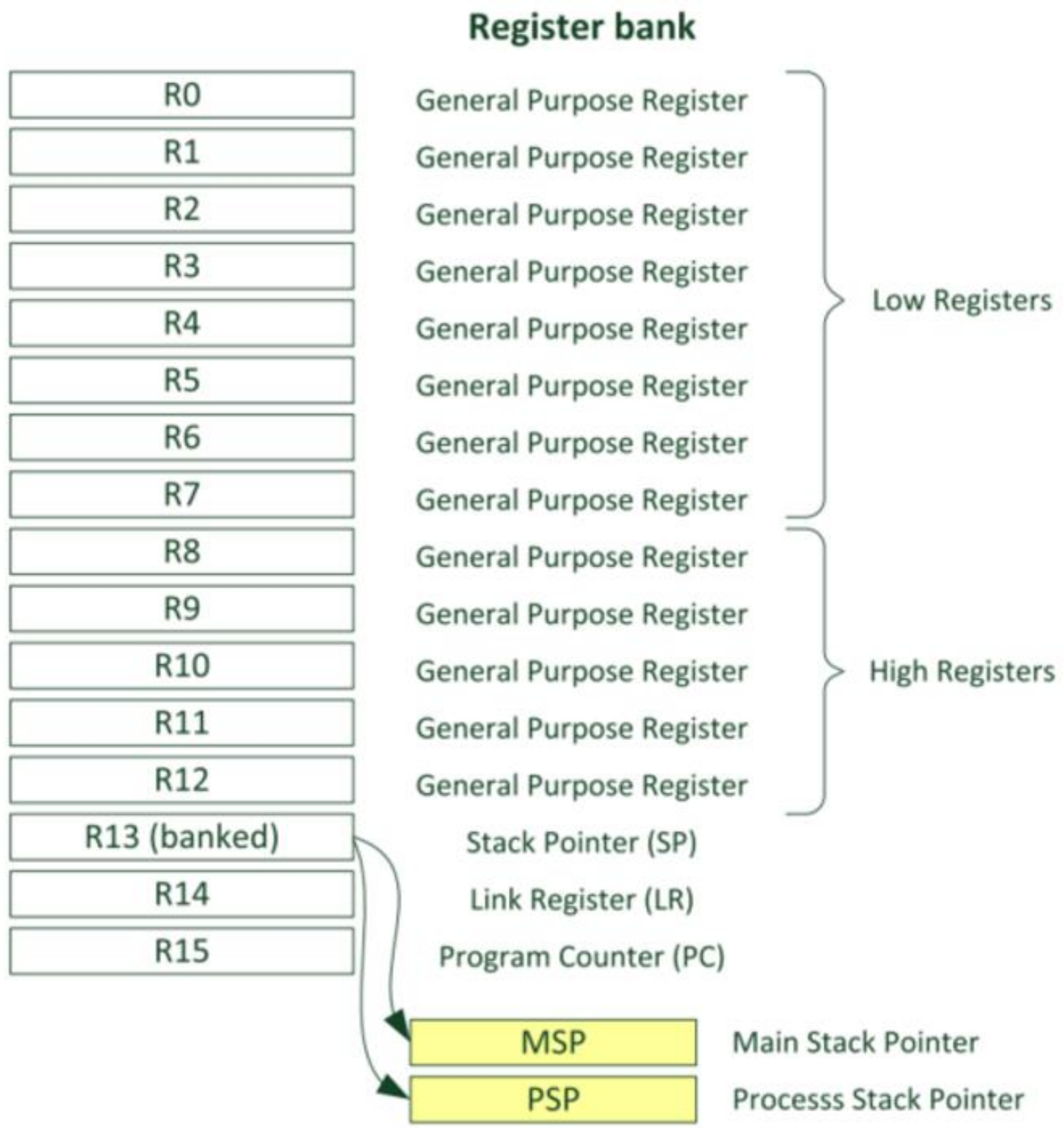


Memory Heap in RTOS

Context Switching, How?



What is context?



Process Stack Pointer

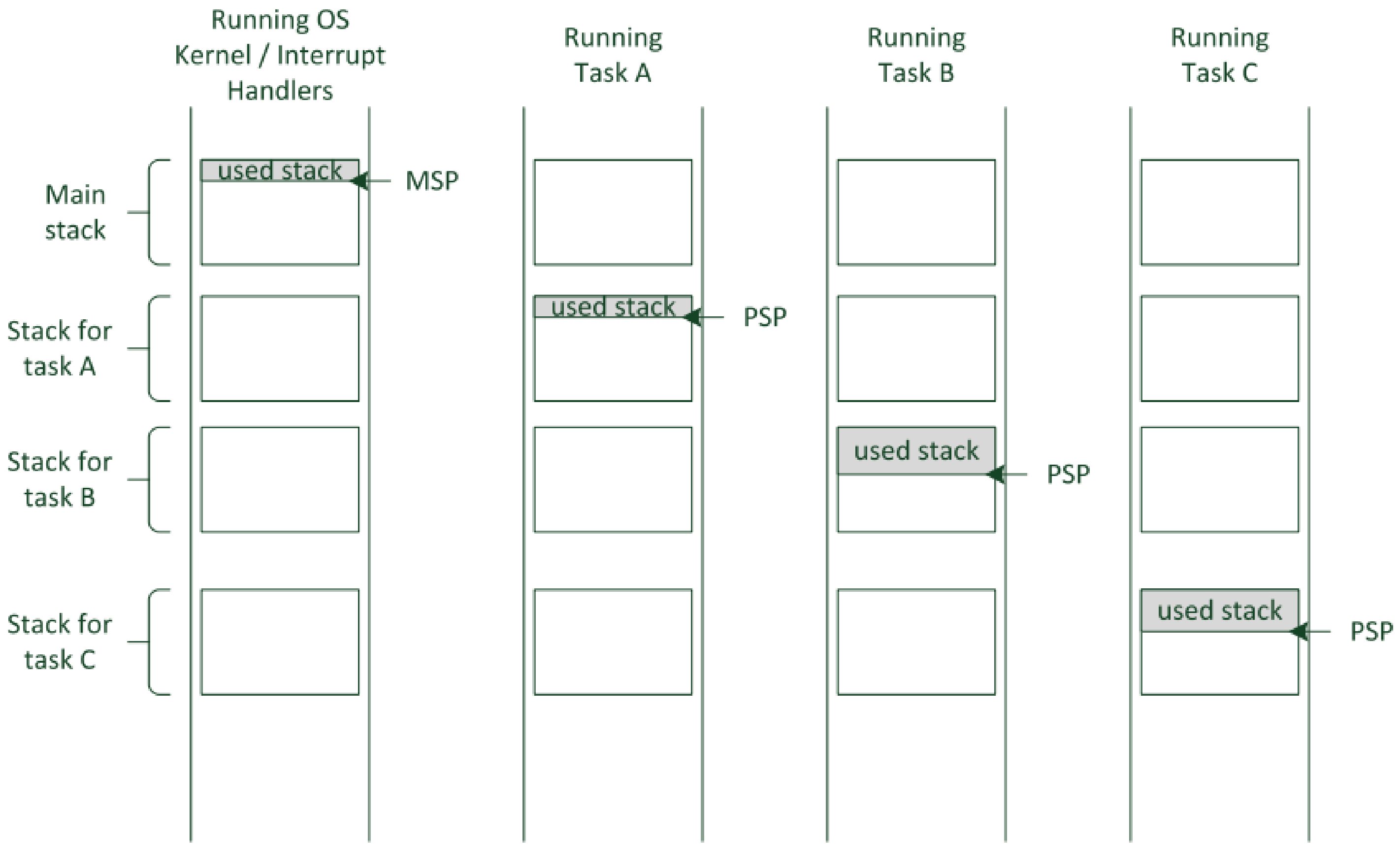


FIGURE 10.1

The stack for each task is separated from the others

Context Switching

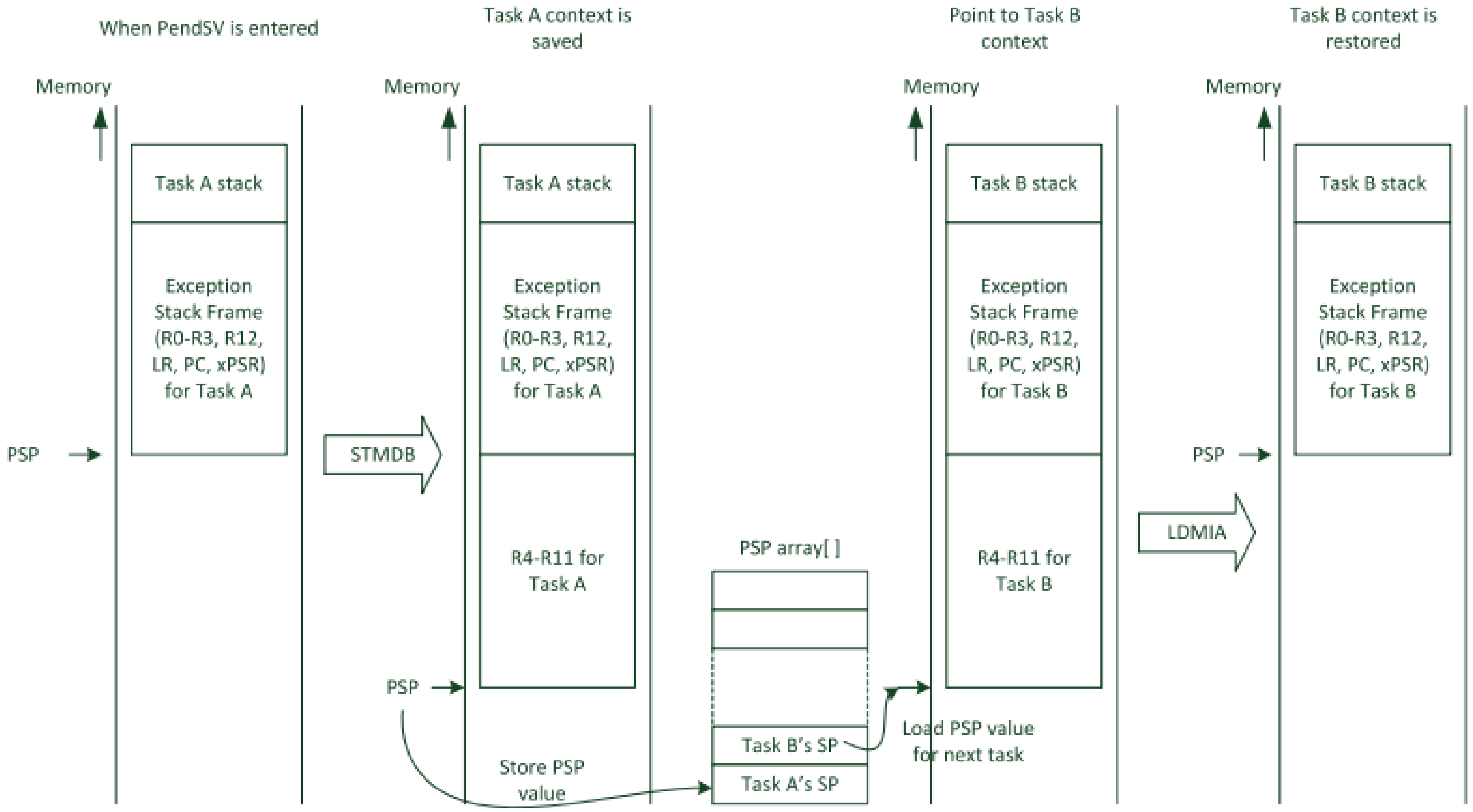
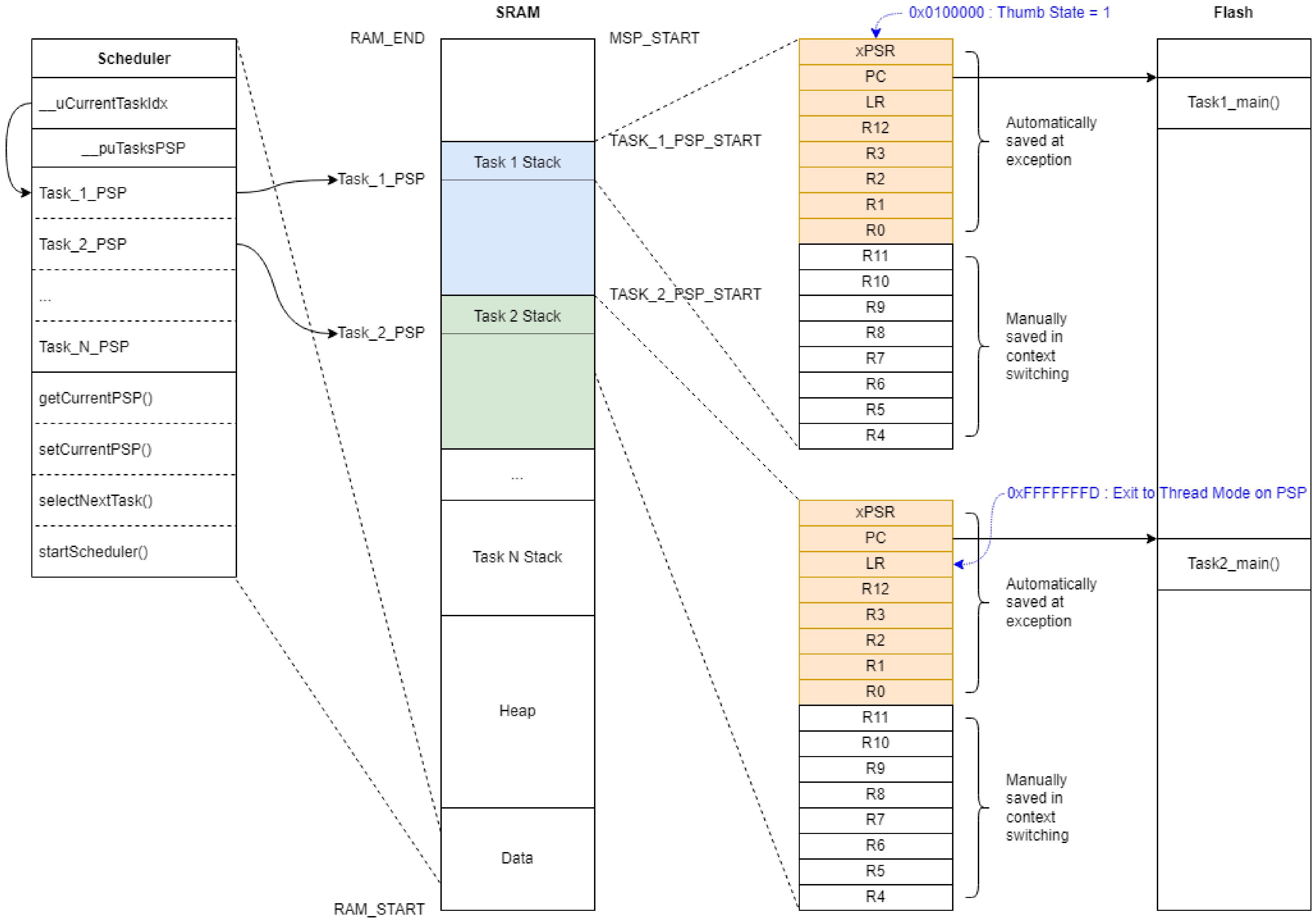
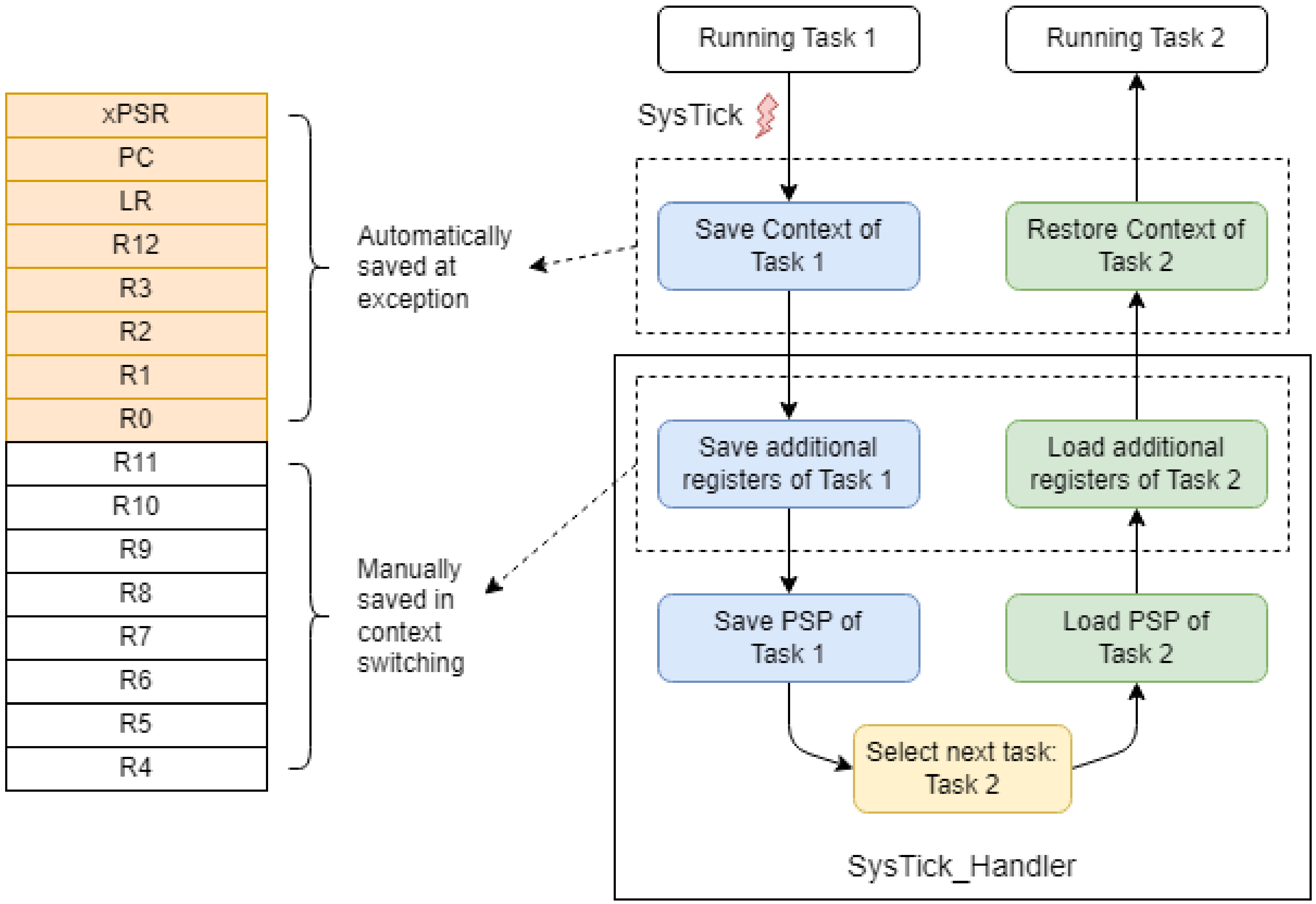


FIGURE 10.10

Context switching


Scheduler
__uCurrentTaskIdx
__puTasksPSP
Task_1_PSP
Task_2_PSP
...
Task_N_PSP
getCurrentPSP()
setCurrentPSP()
selectNextTask()
startScheduler()
SRAM
RAM_END
MSP_START
Task_1_PSP
Task_2_PSP
TASK_1_PSP_START
TASK_2_PSP_START
Task 1 Stack
Task 2 Stack
...
Task N Stack
Heap
Data
RAM_START
0x0100000 : Thumb State = 1
0xFFFFFFFD : Exit to Thread Mode on PSP
xPSR
PC
LR
R12
R3
R2
R1
R0
R11
R10
R9
R8
R7
R6
R5
R4
Automatically saved at exception
Manually saved in context switching
Flash
Task1_main()
Task2_main()

Context Switching



When Context switching happens?

- The execution of an OS kernel context switching can be triggered by:
- Scheduling Points:
  - Execution of SVC instruction from RTOS APIs (Like Delay, Activate Task, Terminate Task, Mutex & semaphoes APIs). For example, when an application task is stalled because it is waiting for some data or event, it can call a system service to swap in another task.
  - Periodic SysTick exception.

RTOS Interrupts

- **SysticTimer** Interrupts:
  - It is integrated as a part of the NVIC.
  - a decrement 24-bit timer.
  - run on processor clock frequency.
  - If you do not need an embedded OS in your application, the SysTick timer can be used as a simple timer peripheral for periodic interrupt generation, delay generation.

Context Switching

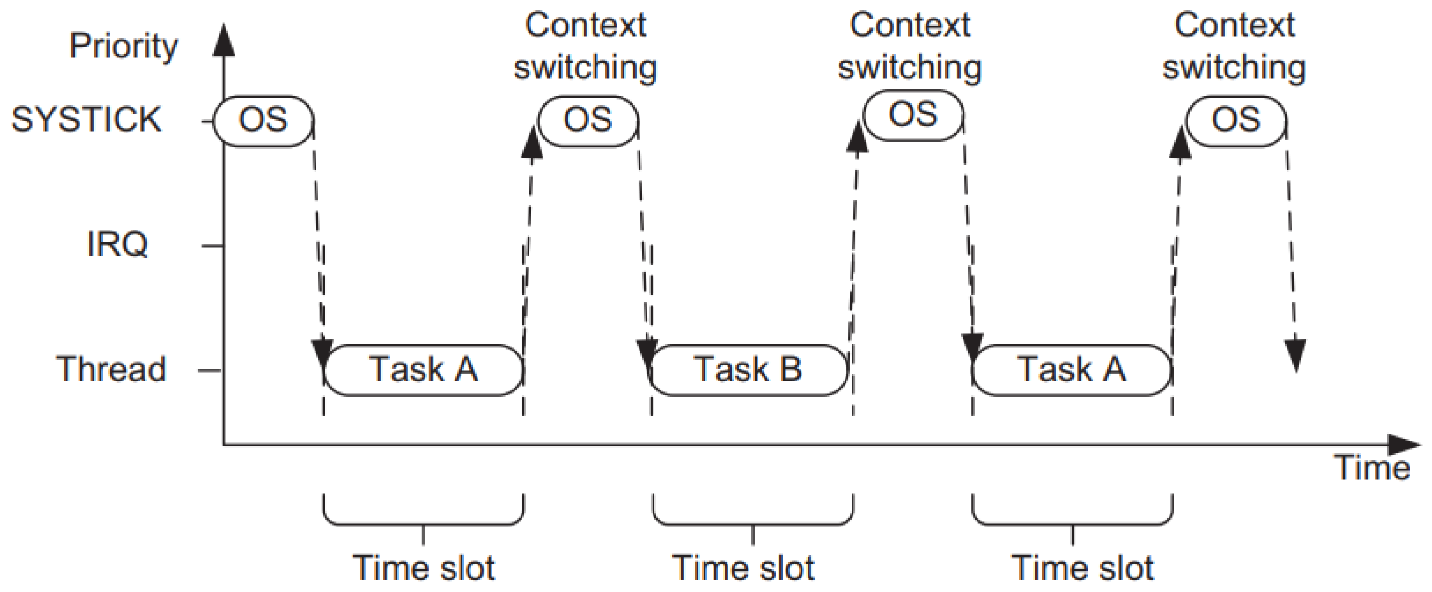


FIGURE 10.6

A simple example of context switching

Context Switching

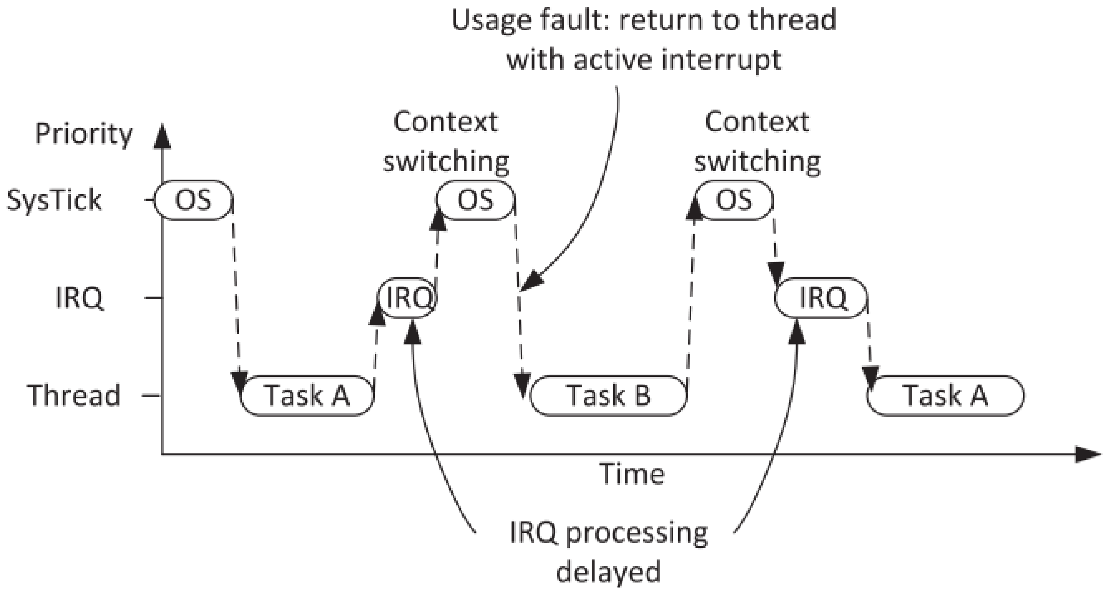


FIGURE 10.7

Context switching during ISR execution can delay interrupt service

RTOS Interrupts

- **PendSV** Exception:
- The context-switching operation is carried out by the PendSV exception handler.
- The PendSV exception delays the context-switching request until all other IRQ handlers have completed their processing.
- The PendSV Exception is triggered by setting its pending status by writing to the Interrupt Control and State Register (ICSR).
- Unlike the SVC exception, it is not precise. So its pending status can be set inside a higher priority exception handler and executed when the higher-priority handler finishes.

Context Switching

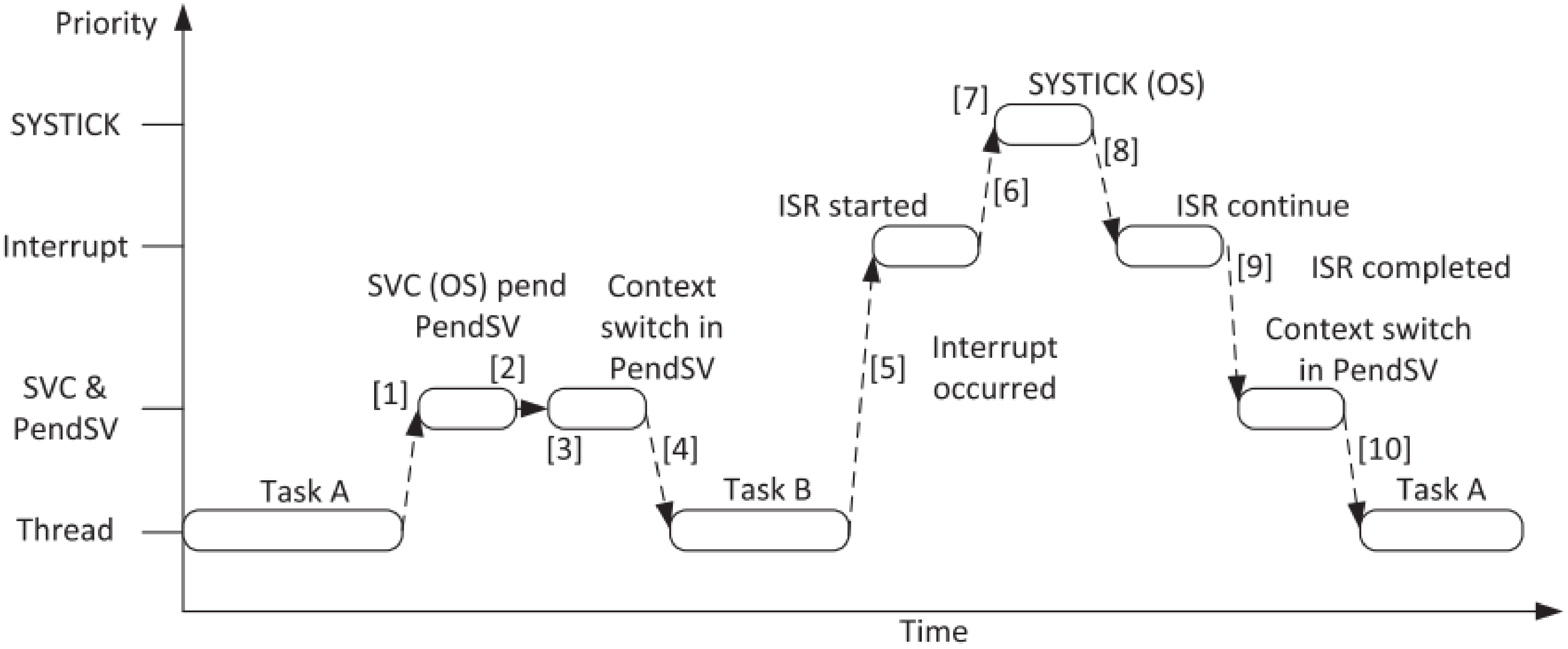


FIGURE 10.8

Example context switching with PendSV

Context Switching

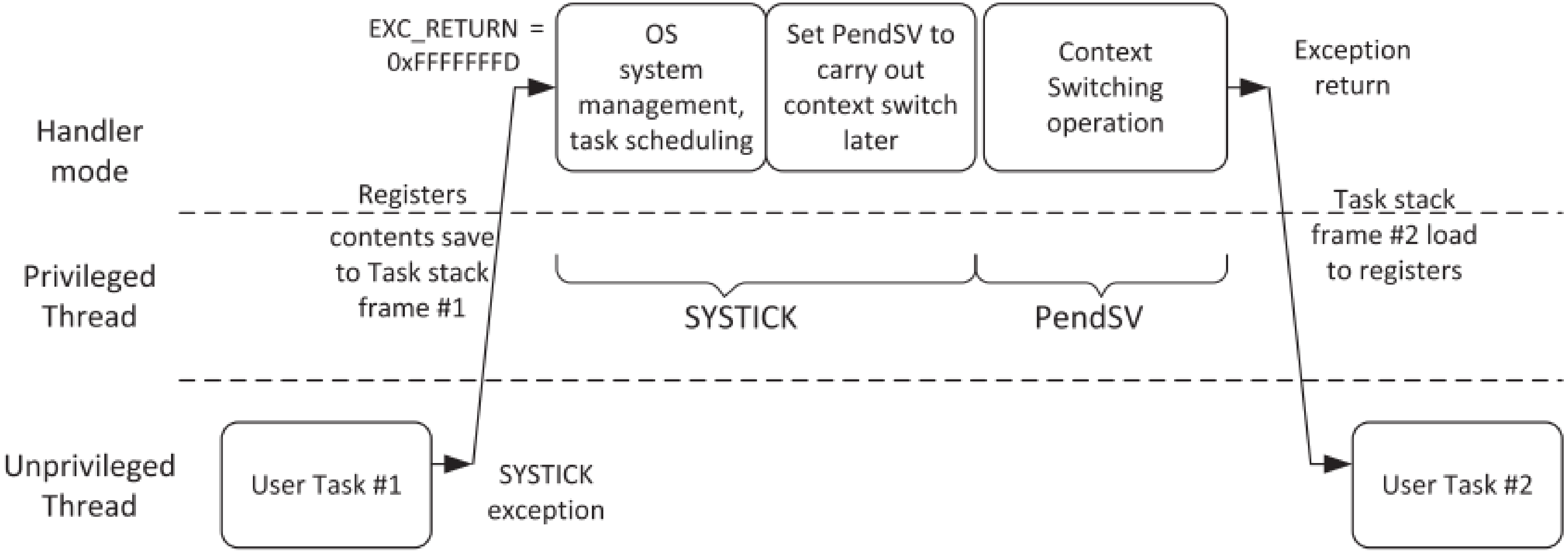


FIGURE 10.3

Concept of context switching

PendSV other Usage

- If there is no embedded OS and there is a time-consuming interrupt.

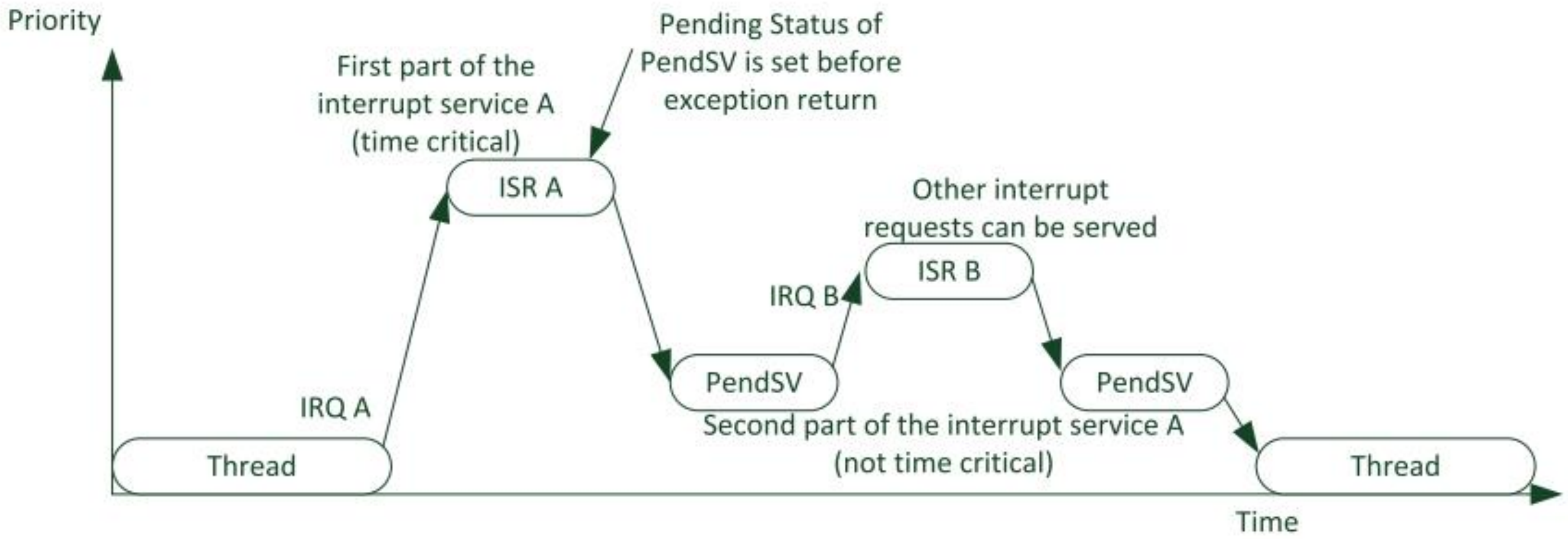
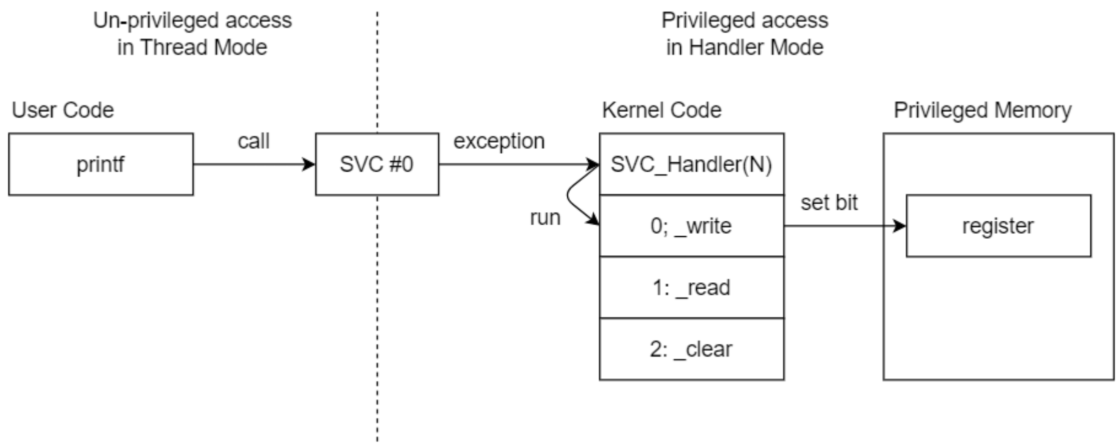


FIGURE 10.9

Using PendSV to partition an interrupt service into two sections

RTOS Interrupts



Example of using Supervisor Call to access to a protected memory

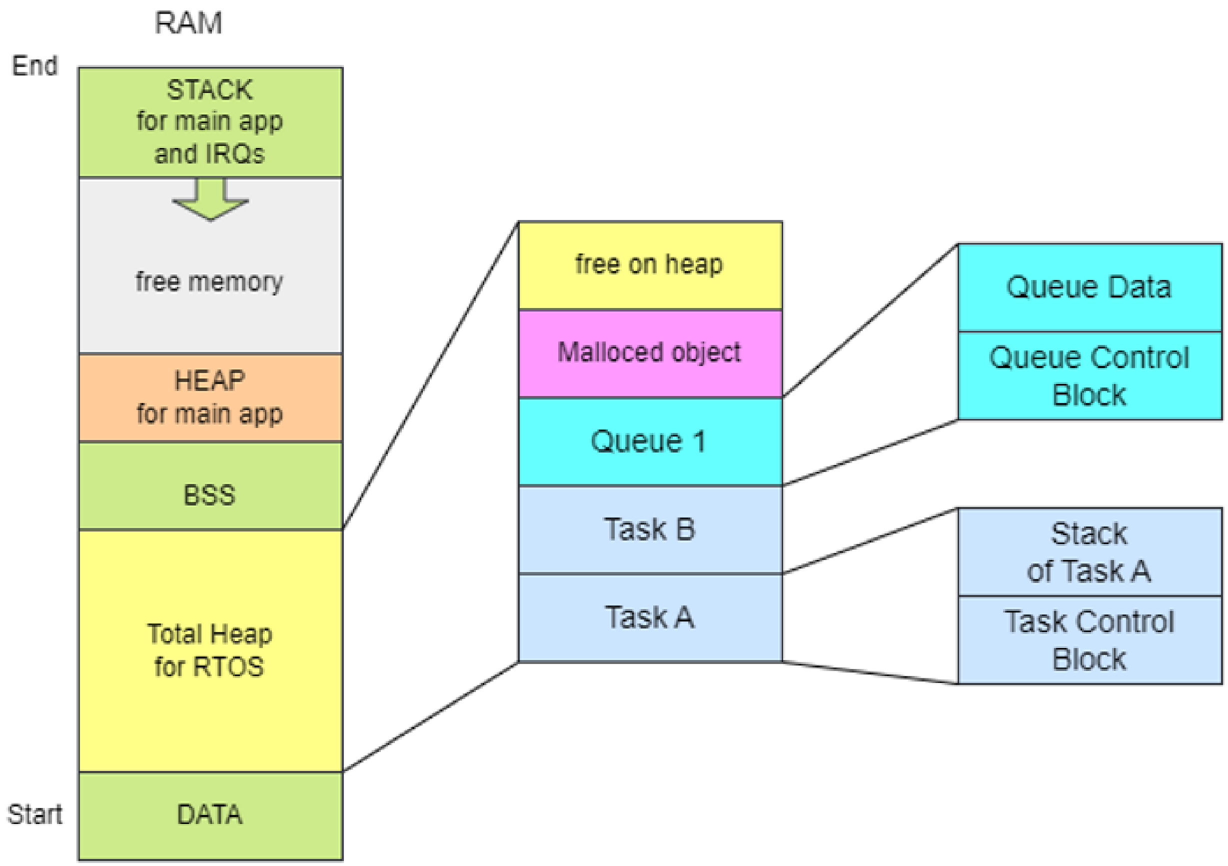
- **SVC** Exception:
- Supervisor Call (SVC) exception allows User Task to request Privileged operations or access to system resource.

FreeRTOS common APIs

- In FreeRTOS documentation ^^
- Create Task API

FreeRTOS Memory Management

Memory Heap In RTOS



Memory Heap in RTOS

Static Vs Dynamic Memory Allocation

- Creating RTOS objects using statically allocated RAM has the benefit of providing the application writer with more control:
- RTOS objects can be placed at specific memory locations.
- The maximum RAM footprint can be determined at link time, rather than run time.

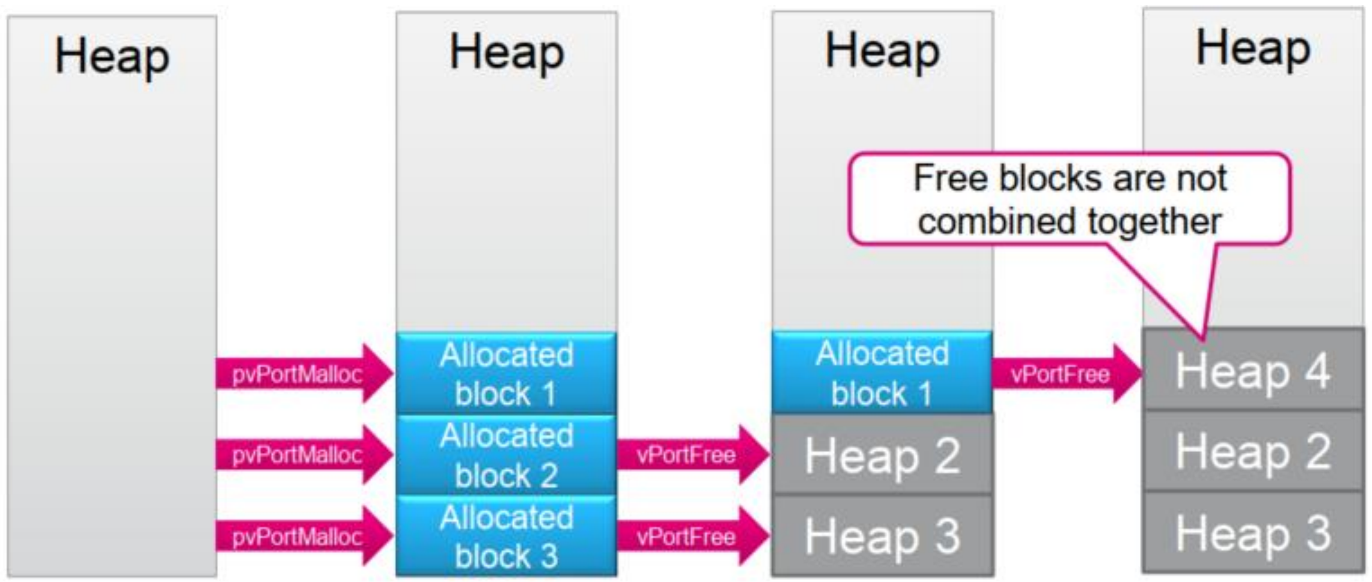
- The memory allocation occurs automatically, within the RTOS API functions.
- The application writer does not need to concern themselves with allocating memory themselves.
- The RAM used by an RTOS object can be re-used if the object is deleted, potentially reducing the application's maximum RAM footprint.

FreeRTOS Dynamic Memory

- FreeRTOS manages own heap for:
  - Tasks
  - Queues
  - Semaphores.
  - Mutexes.
  - Dynamic memory allocation

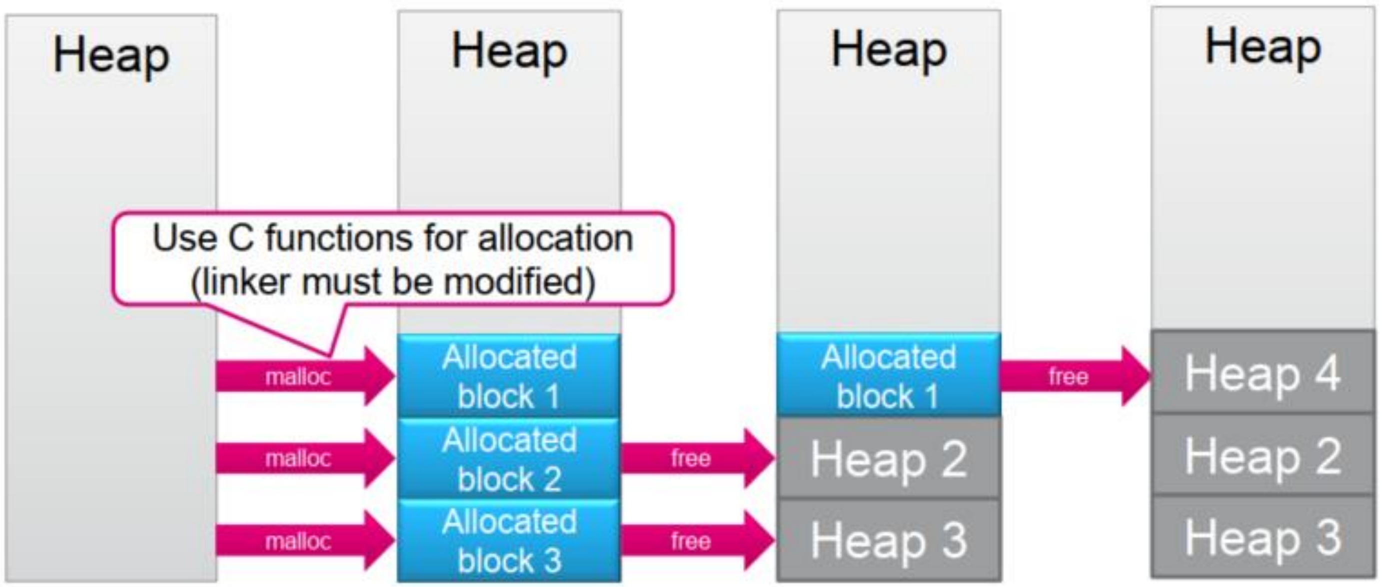
- Configured by (**TOTAL_HEAP_SIZE**)

FreeRTOS Dynamic memory management



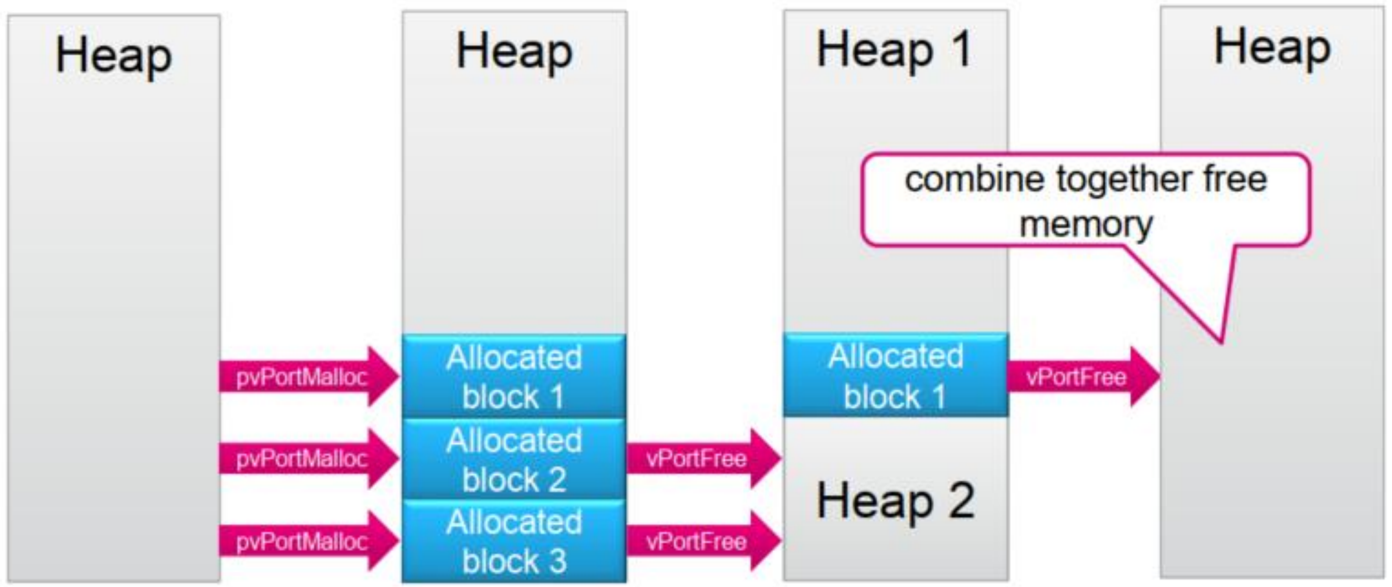
- **Heap_2.c**
  - Not recommended to new projects. Kept due to backward compatibility.
  - Implements the best fit algorithm for allocation
  - Allows memory free() operation but doesn't combine adjacent free blocks
  => risk of fragmentation

FreeRTOS Dynamic memory management



- **Heap_3.c**
  - Implements simple wrapper for standard C library malloc() and free(); wrapper makes these functions thread safe, but makes code increase and not deterministic
  - It uses linker heap region.
  - configTOTAL_HEAP_SIZE setting has no effect when this model is used

FreeRTOS Dynamic memory management



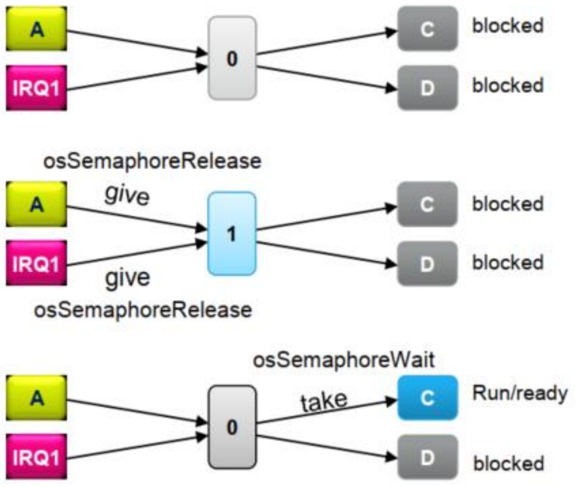
- **Heap_4.c**
  • Uses **first fit algorithm** to allocate memory. It is able to combine adjacent free memory blocks into a single block

RTOS Synchronization

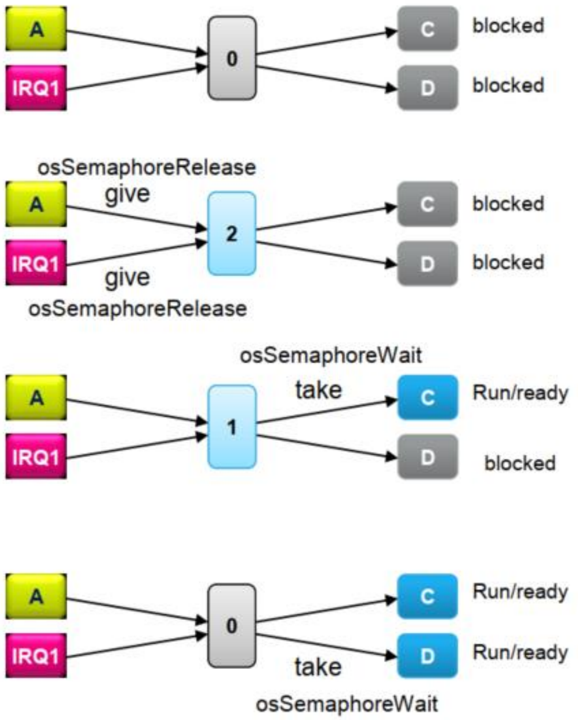
Semaphores

- Semaphores are used to synchronize tasks with other events in the system (especially IRQs).
- Waiting for semaphore is equal to wait() procedure, task is in blocked state not taking CPU time.
- Semaphore should be created before usage
- Counting semaphores are typically used for two purposes:
  - Counting events.
  - Resource management : the count value indicates the number of resources available.

Semaphores: Binary vs Counting



Binary

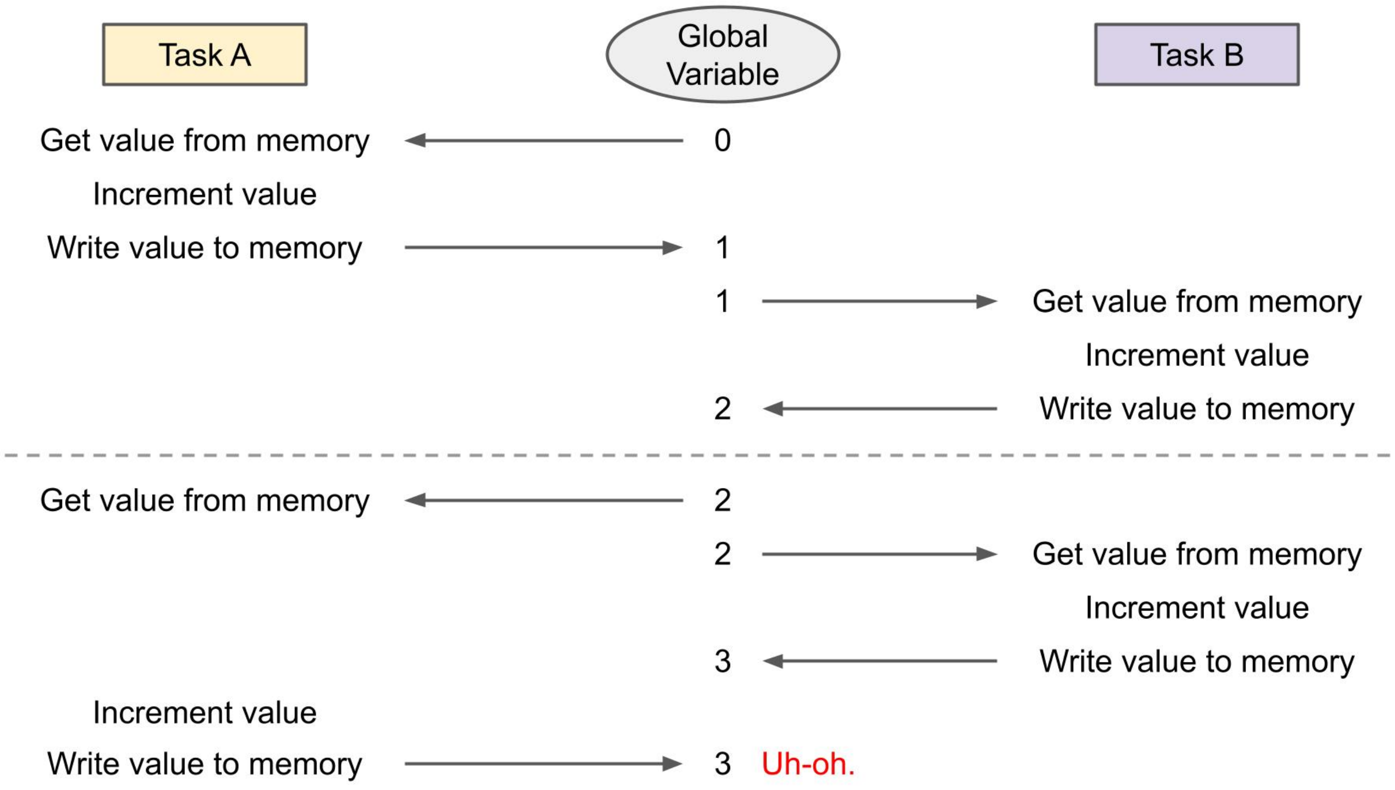


Counting

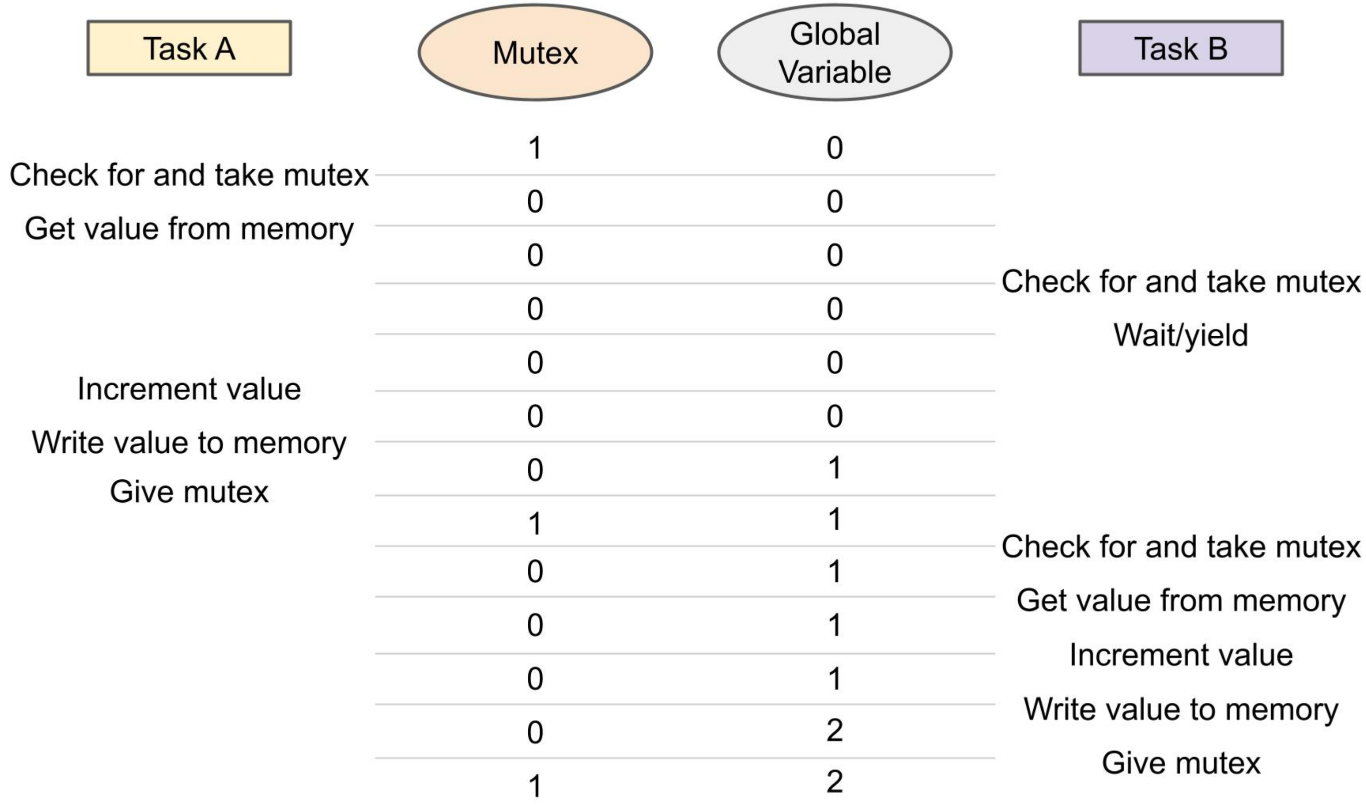
Mutex

- It is a flag or lock used to allow only one thread to access a section of code at a time. It blocks (or locks out) all other threads from accessing the code or resource
- Mutex is a binary semaphore that include a priority inheritance mechanism:
- binary semaphore is the better choice for implementing synchronization (between tasks or between tasks and an interrupt),
- Mutex is the better choice for implementing simple mutual exclusion.
- Unlike binary semaphores however - mutexes employ priority inheritance. This means that if a high priority task is blocked while attempting to obtain a mutex (token) that is currently held by a lower priority task, then the priority of the task holding the token is temporarily raised to that of the blocked task.
- Mutex Management functions cannot be called from interrupt service routines (ISR).
- A task must not be deleted while it is controlling a Mutex. Otherwise, the Mutex resource will be locked out to all other tasks.

Shared Variable Without Mutex

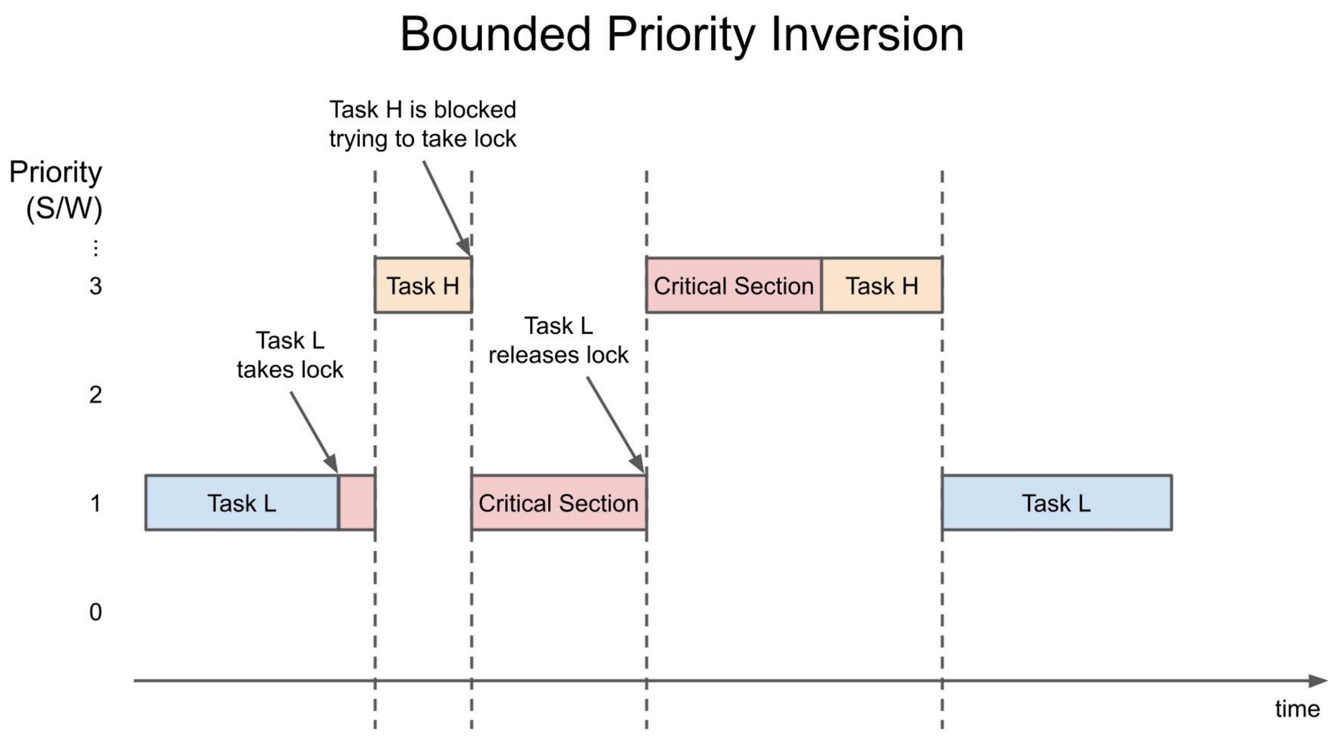


With Mutex



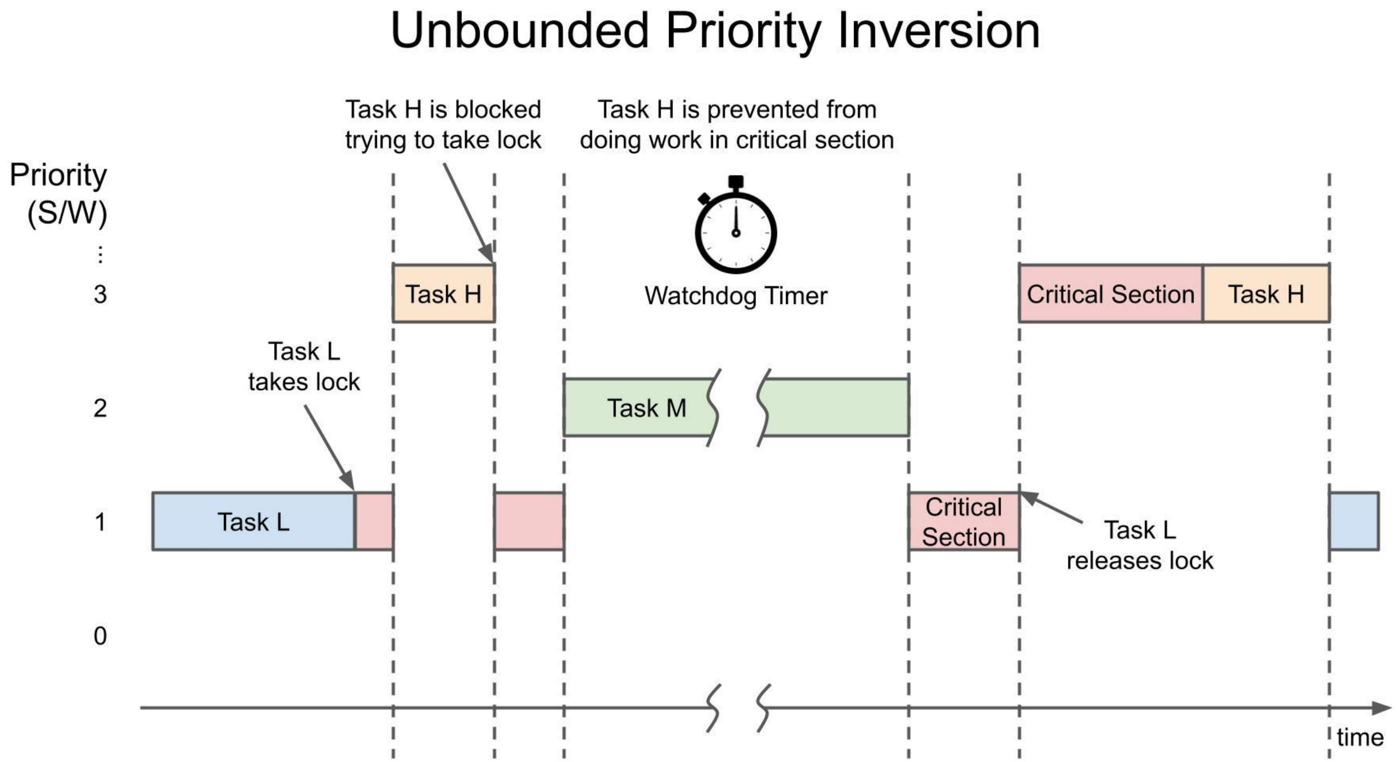
Task A
Mutex
Global Variable
Task B
1
0
Check for and take mutex
0
0
Get value from memory
0
0
Check for and take mutex
0
0
Wait/yield
0
0
Increment value
0
0
Write value to memory
0
1
Give mutex
1
1
Check for and take mutex
0
1
Get value from memory
0
1
Increment value
0
1
Write value to memory
0
2
Give mutex
1
2

Priority inversion

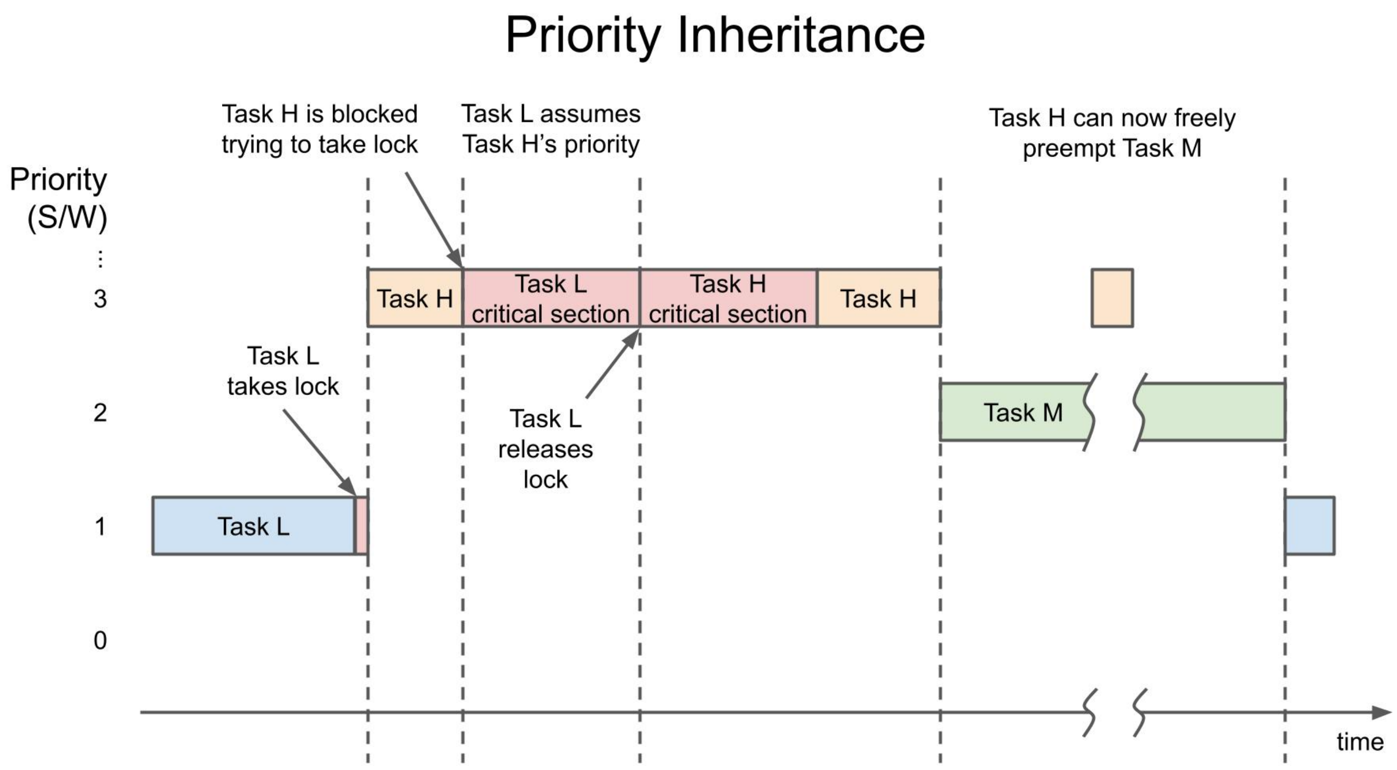


- This is the situation where a higher priority task is waiting for a lower priority task to give a control of the mutex, and low priority task is not able to execute.

Priority inversion

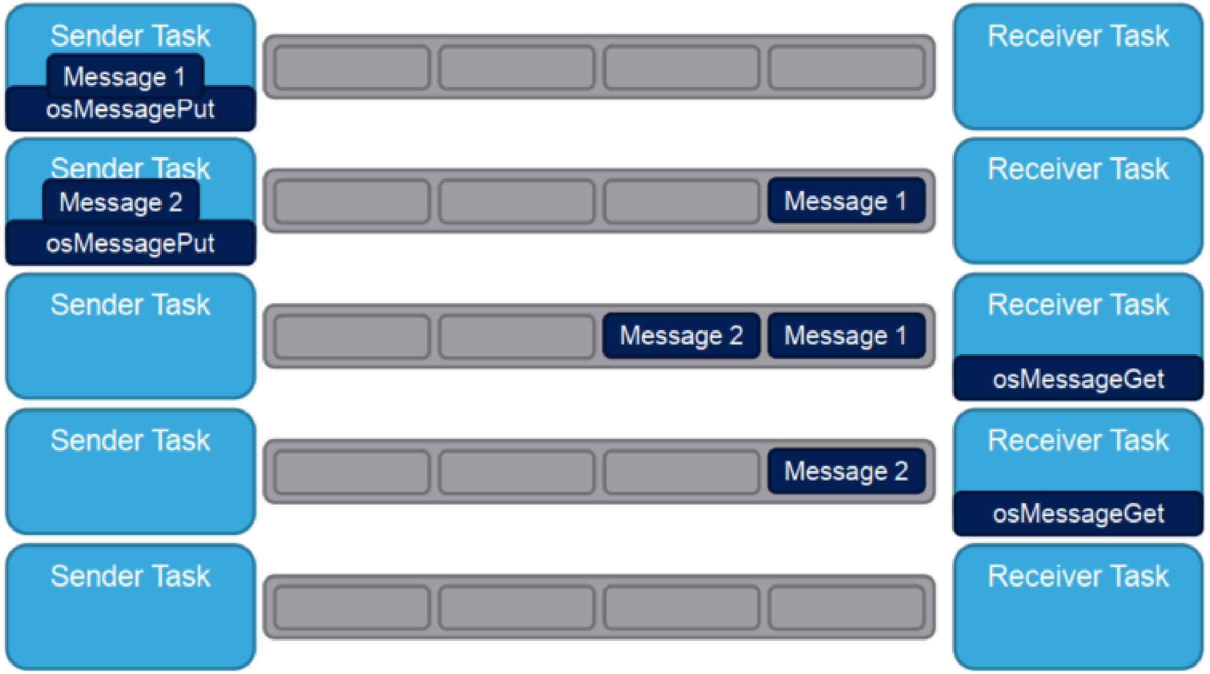


Priority Inheretence

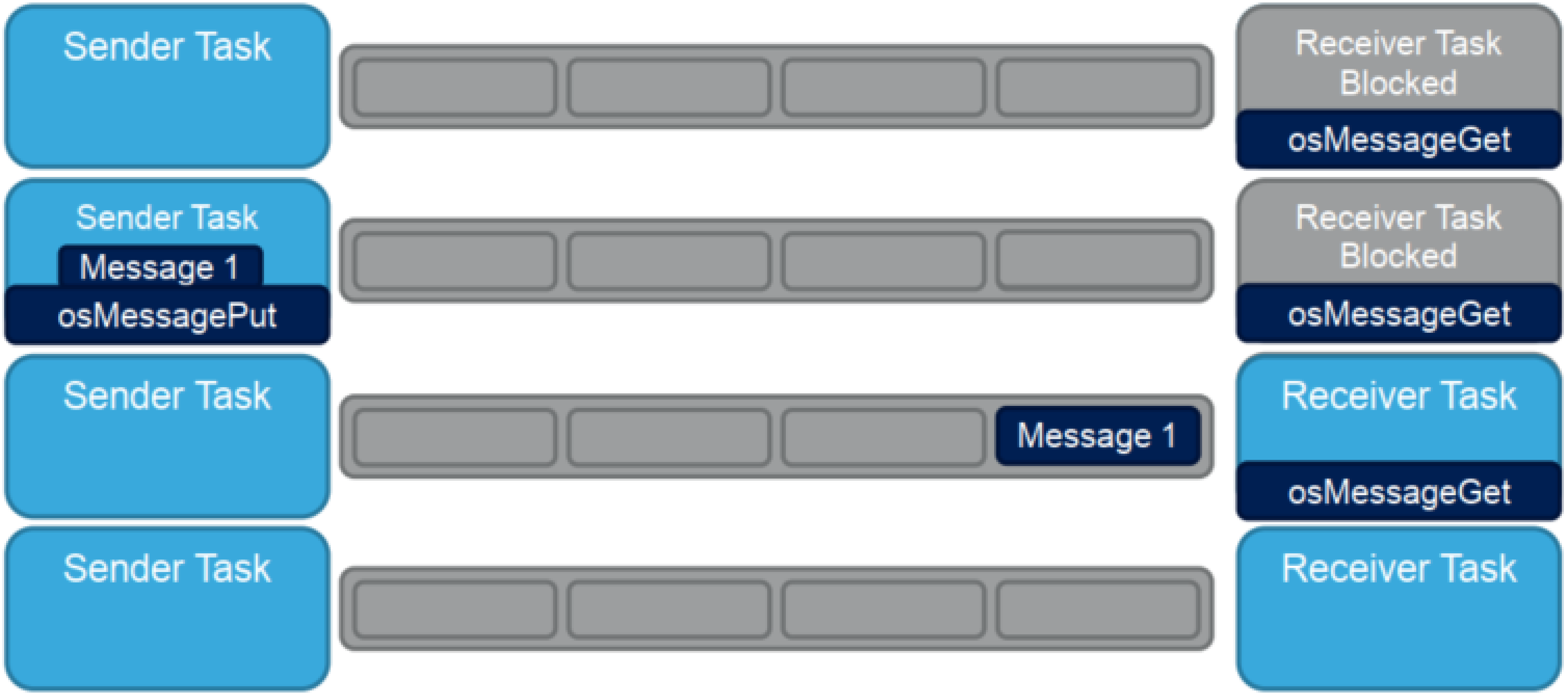


RTOS Inter-task Communication

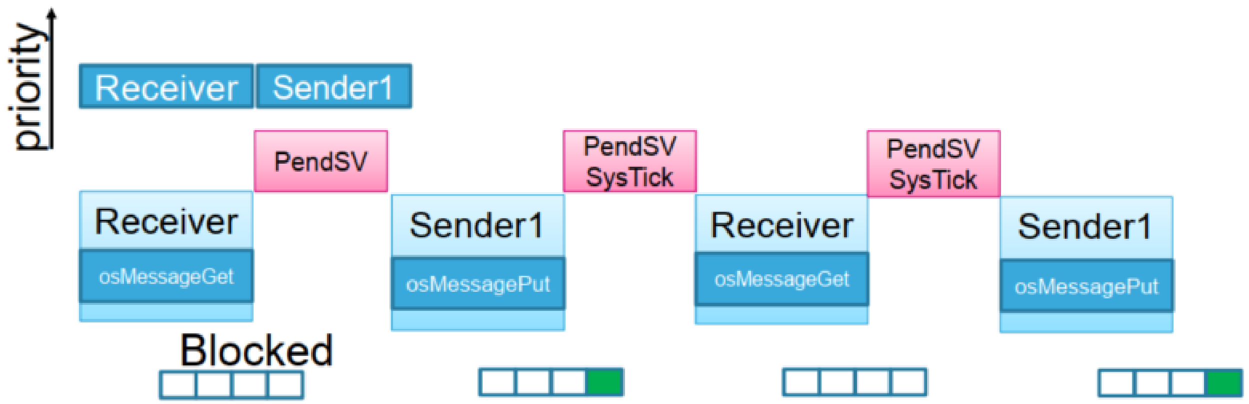
Queues


Sender Task
Message 1
osMessagePut
Receiver Task
Sender Task
Message 2
osMessagePut
Message 1
Receiver Task
Sender Task
Message 2
Message 1
Receiver Task
osMessageGet
Sender Task
Message 2
Receiver Task
osMessageGet
Sender Task
Receiver Task

Queue Blocking



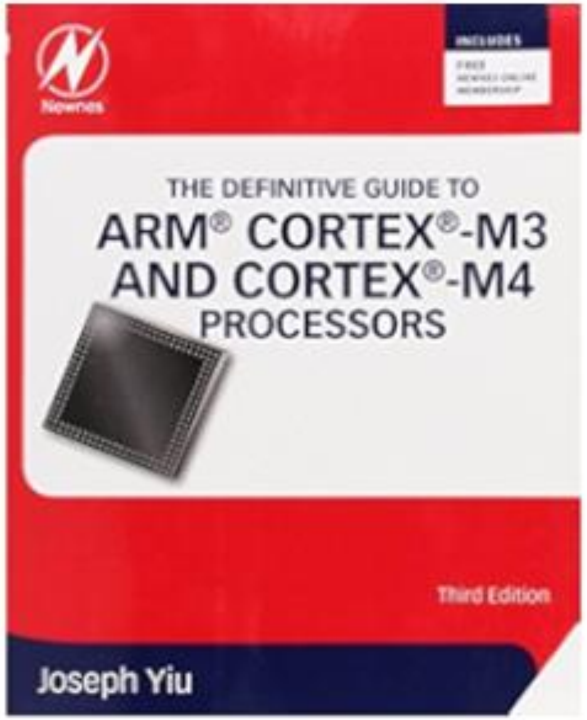
Example

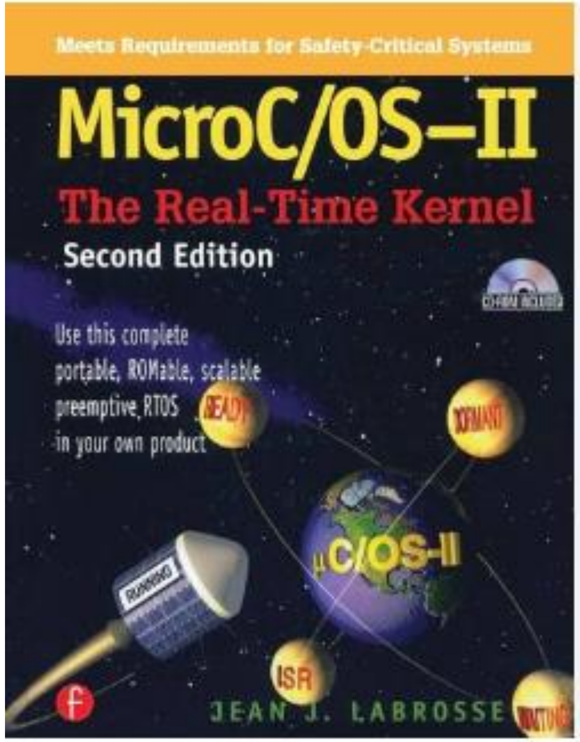

priority
Receiver
Sender1
PendSV
PendSV
SysTick
PendSV
SysTick
Receiver
osMessageGet
Sender1
osMessagePut
Receiver
osMessageGet
Sender1
osMessagePut
Blocked

Queue APIs

- freeRTOS documentation

References


THE DEFINITIVE GUIDE TO
ARM® CORTEX®-M3
AND CORTEX®-M4
PROCESSORS
Third Edition
Joseph Yiu


Meets Requirements for Safety-Critical Systems
MicroC/OS–II
The Real-Time Kernel
Second Edition
Use this complete
portable, ROMable, scalable
preemptive RTOS
in your own product
µC/OS-II
ISR
JEAN J. LABROSSE

STM32WBA Architecture

NUCLEO-WBA55CG is a wireless and ultra-low-power board

STM32WBA

Figure 1. NUCLEO-WBA55CG global view

Figure 3. **NUCLEO-WBA55CG PCB top view**

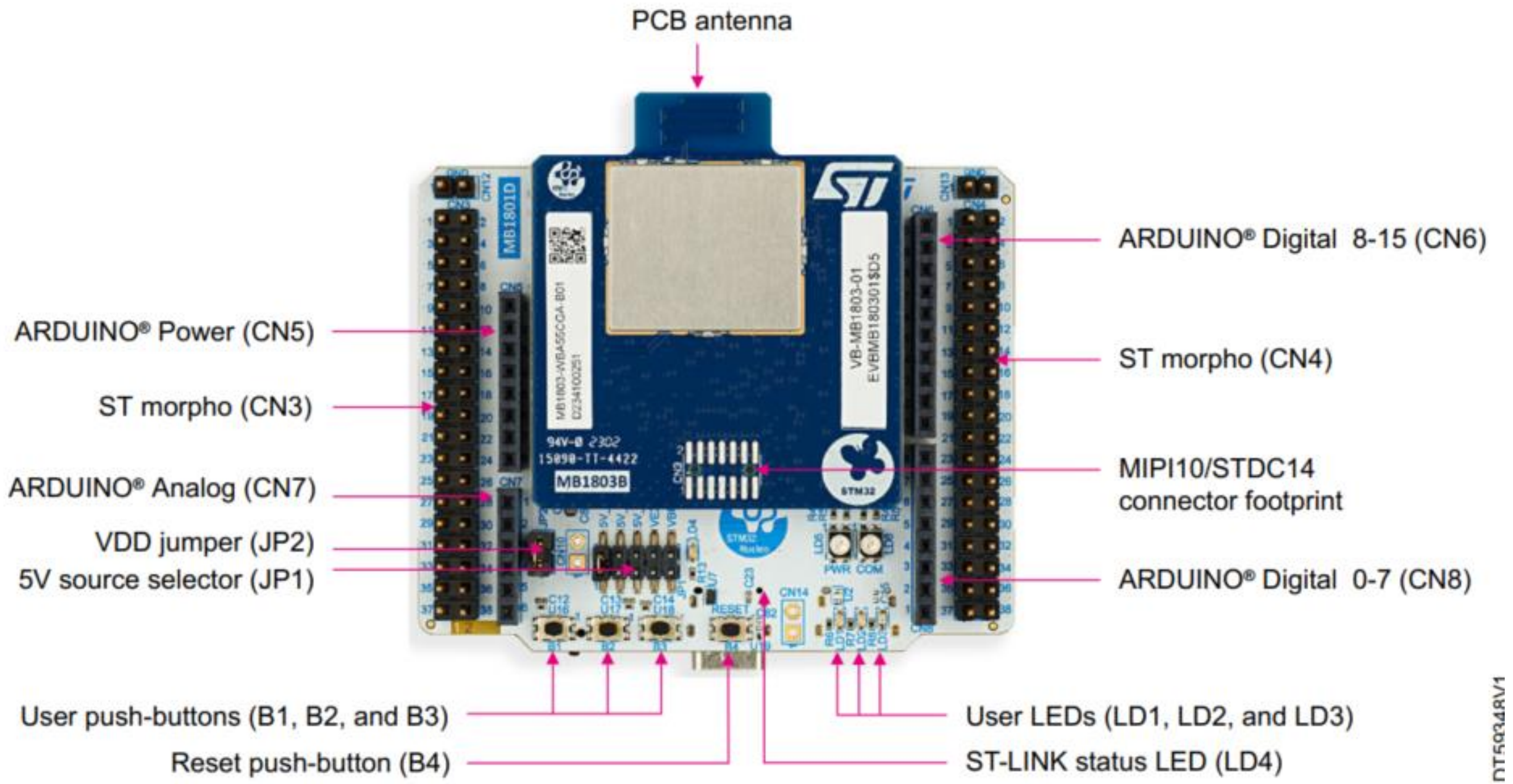


Figure 23. ST morpho connector pinout

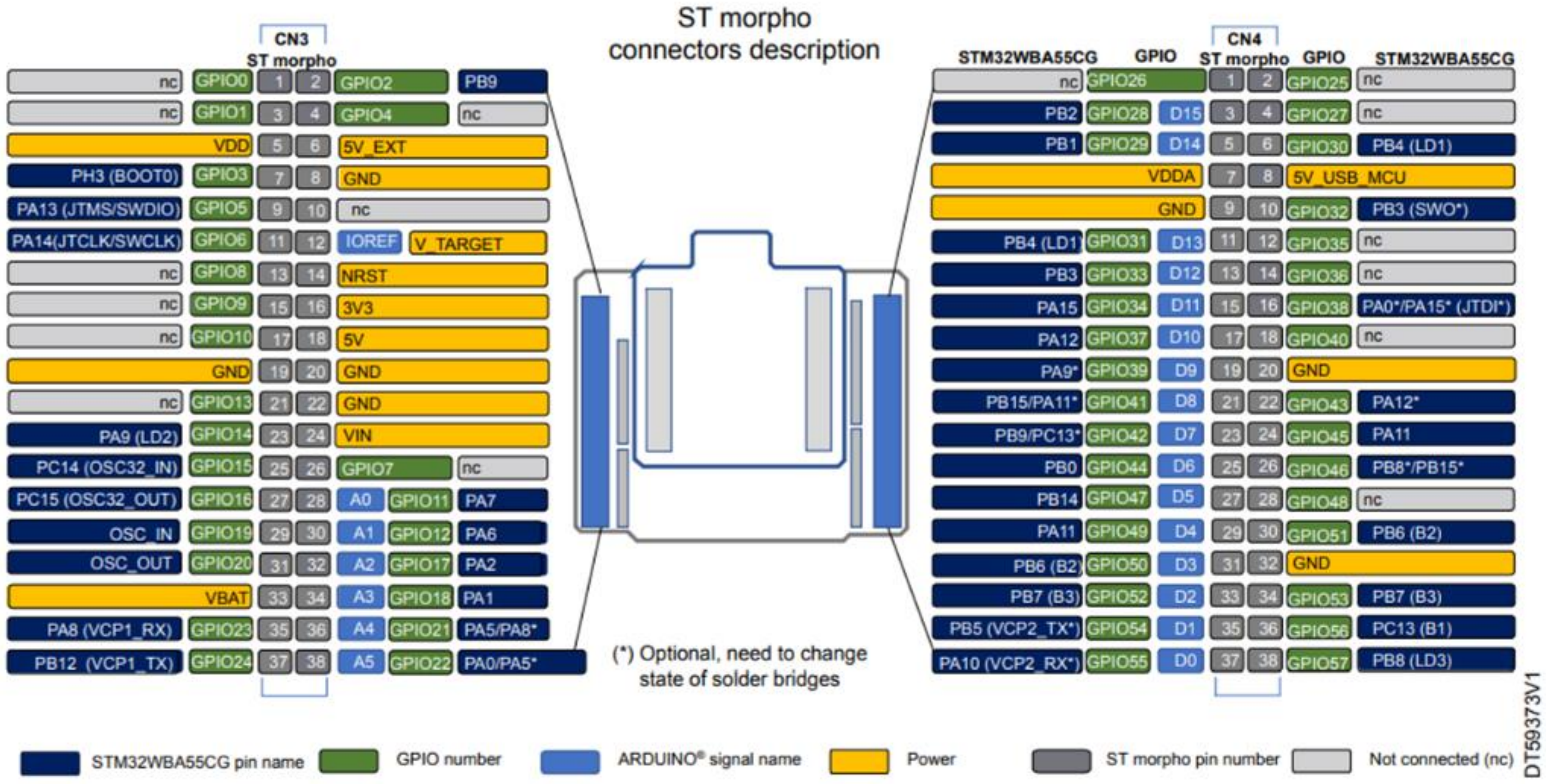


Figure 2. Hardware block diagram