

Mastering RTOS: Hands on FreeRTOS and STM32Fx with Debugging

Learn Running/Porting FreeRTOS Real Time Operating System on
STM32F4x and ARM cortex M based Microcontrollers

Created by :

FastBit Embedded Brain Academy

Visit www.fastbitlab.com

for all online video courses on MCU programming, RTOS and
embedded Linux

About FastBit EBA

FastBit Embedded Brain Academy is an online training wing of Bharati Software.
We leverage the power of the internet to bring online courses at your fingertip in the domain of embedded systems and programming, microcontrollers, real-time operating systems, firmware development, Embedded Linux.

All our online video courses are hosted in Udemy E-learning platform which enables you to exercise 30 days no questions asked money back guarantee.

For more information please visit : www.fastbitlab.com
Email : contact@fastbitlab.com

RTOS Introduction

What is Real Time Application (RTA) ?

Truths and Myths

Myth

The Real-Time Computing is equivalent to fast computing

Truth

The Real-Time Computing is equivalent to Predictable computing

Truths and Myths

Myth

The Real-Time Computing means higher performance

Truth

In Real-Time Computing timeliness is more important than performance.

What is Real time ?

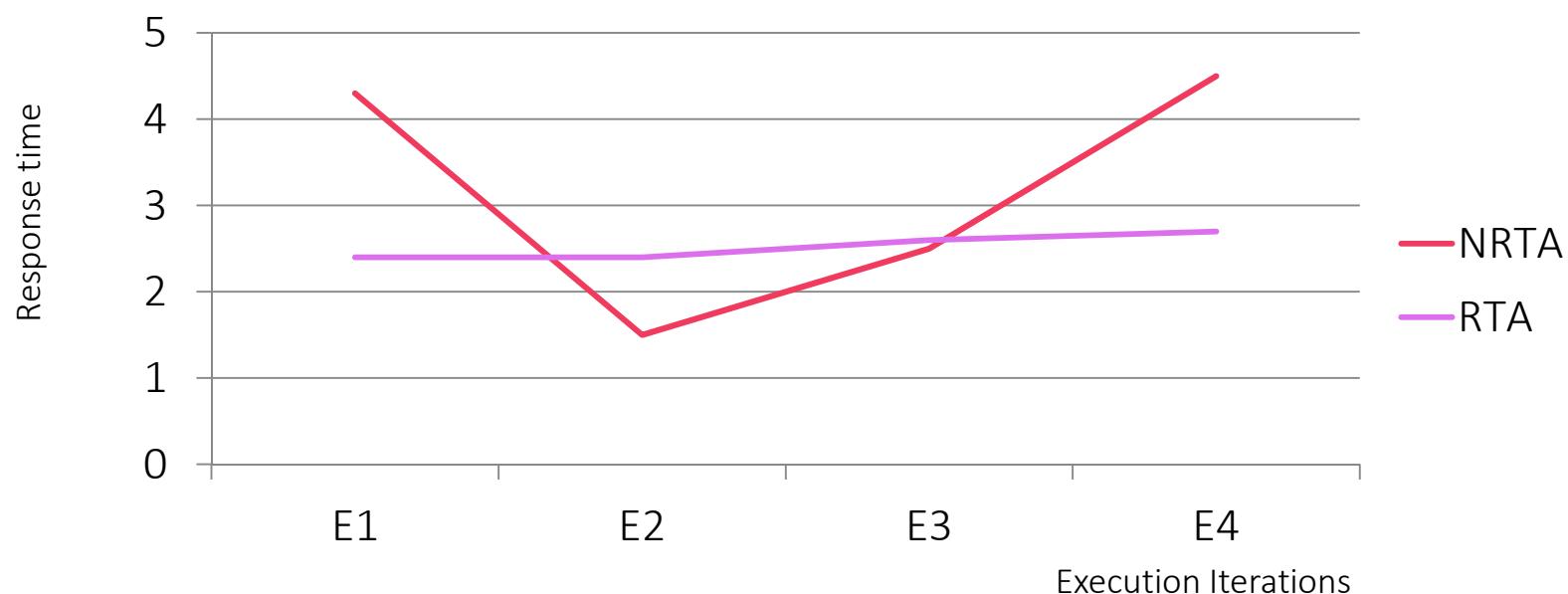
"Real-time deals with *guarantees*, not with *raw speed*.

What is Real time ?

“A Real time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation, but also upon the time at which the result is produced . If the timing constraints are not met, system failure is said to have occurred “

What is Real time ?

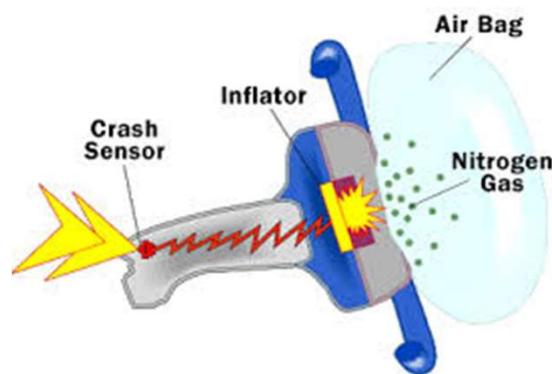
The response time is guaranteed



What are Real time Applications(RTAs) ?

- RTAs are not fast executing applications.
- RTAs are time deterministic applications, that means, their response time to events is almost constant.
- There could be small deviation in RTAs response time, in terms of ms or seconds which will fall into the category of soft real time applications.
- Hard real time functions must complete within a given time limit. Failure to do so will result in absolute failure of the system.

What are Real time Applications(RTAs) ?



Hard-Real Time application



May be soft-real time application ?
Delay is tolerable

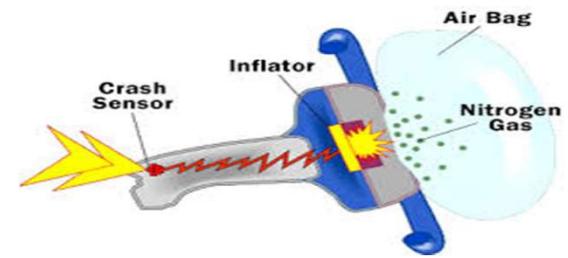
Some Examples of RTAs



Missile guidance and control systems



Anti-Lock Braking System



Airbag Deployment



Stock Market Website

Industry	Hard Real-time	Soft Real-time
Aerospace	Fault Detection, Aircraft Control	Display Screen
Finance	ATMs	Stock Market Websites
Industrial	Robotics, DSP	Temperature Monitoring
Medical	CT Scan, MRI, fMRI	Blood Extraction, Surgical Assist
Communications	QoS	Audio/Video Streaming, Networking, Camcorders

Aerospace

Fault Detection(Hard)
Aircraft Control(Hard)
Display Screen(Soft)

Finance

ATMs(Hard)
Stock Market Websites(Soft)

Medical

CT Scan(Hard)
MRI(Hard)
Blood Extraction (Soft)
Surgical Assist (soft)

Industrial

Robotics(Hard)
DSP (Hard)
Temperature Monitor (soft)

Communications

Audio/Video Streaming(soft)

Real-Time Application

Time Deterministic – Response time to events is always almost constant .

You can always trust RTAs in terms of its timings in responding to events

What is Real Time Operating system?

What is a Real Time OS?

It's a OS , specially designed to run applications with very precise timing and a high degree of reliability.

To be considered as "real-time", an operating system must have a known maximum time for each of the critical operations that it performs. Some of these operations include

- ✓ Handling of interrupts and internal system exceptions
- ✓ Handling of Critical Sections.
- ✓ Scheduling Mechanism , etc.

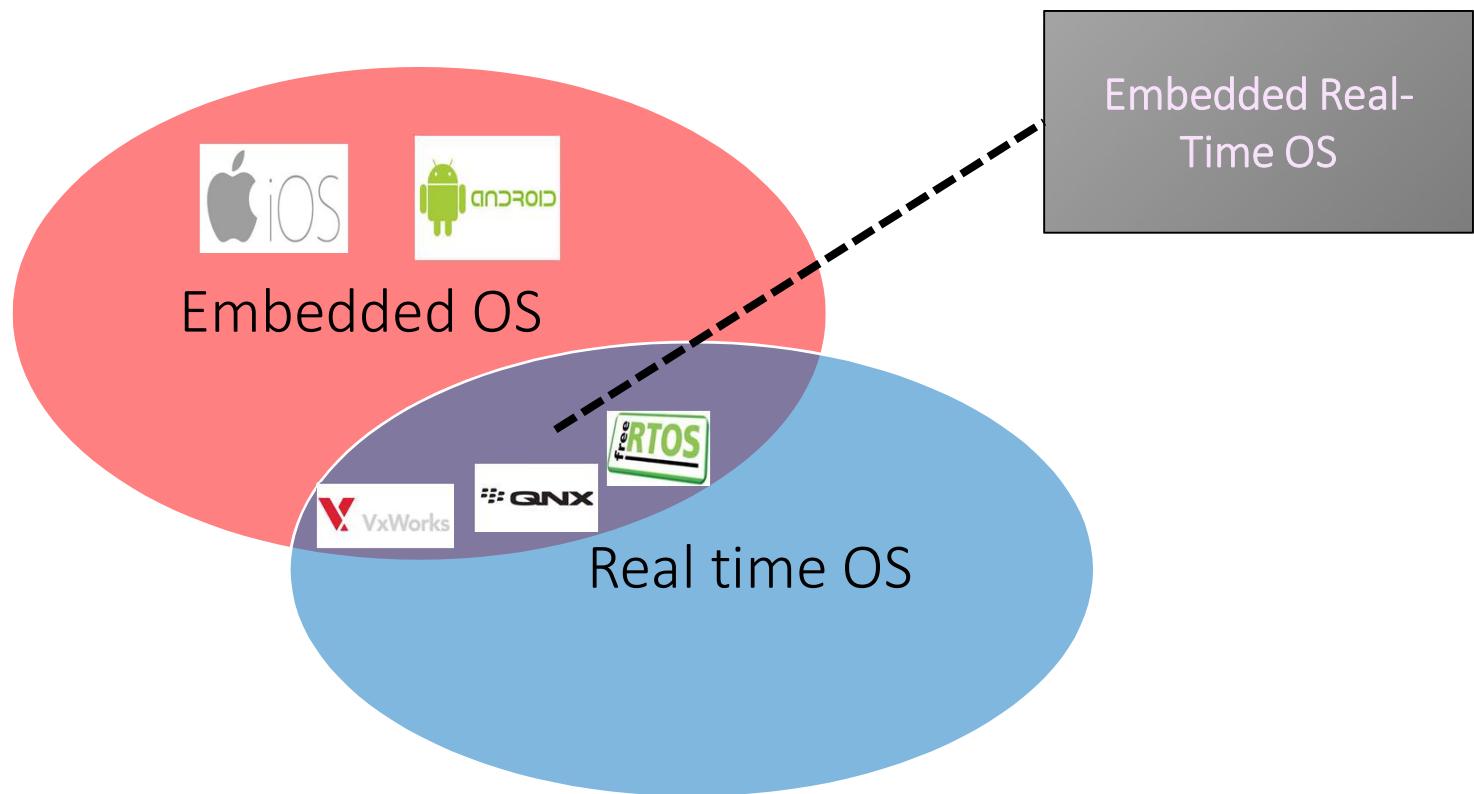
RTOS vs GPOS

GPOS



RTOS





RTOS vs GPOS: Task Scheduling

GPOS

In the case of a GPOS – task scheduling is not based on “priority” always!

RTOS

Where as in RTOS – scheduling is always priority based. Most RTOS use pre-emptive task scheduling method which is based on priority levels.

GPOS- Task Scheduling

GPOS is programmed to handle scheduling in such a way that it manages to achieve high throughput.

Throughput means – the total number of processes that complete their execution per unit time

Some times execution of a high priority process will get delayed inorder to serve 5 or 6 low priority tasks. High throughput is achieved by serving 5 low priority tasks than by serving a single high priority one.

GPOS- Task Scheduling

In a GPOS, the scheduler typically uses a fairness policy to dispatch threads and processes onto the CPU.

Such a policy enables the high overall throughput required by desktop and server applications, but offers no guarantees that high-priority, time critical threads or processes will execute in preference to lower-priority threads.

RTOS- Task Scheduling

On the other hand in RTOS, Threads execute in the order of their priority. If a high-priority thread becomes ready to run, it will take over the CPU from any lower-priority thread that may be executing.

Here a high priority thread gets executed over the low priority ones. All “low priority thread execution” will get paused. A high priority thread execution will get override only if a request comes from an even high priority threads.

RTOS vs GPOS: Task Scheduling

Does that mean , RTOS are very poor in throughput ?

RTOS vs GPOS: Task Scheduling

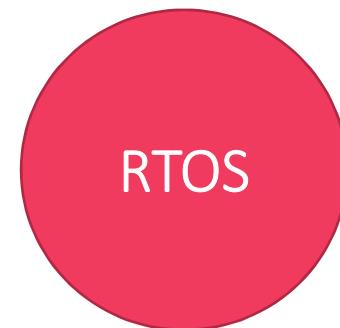
RTOS may yield less throughput than the General Purpose OS, because it always favors the high Priority task to execute first , but that does not mean, it has very poor throughput !

A Quality RTOS will still deliver decent overall throughput but can sacrifice throughput for being deterministic or to achieve time predictability .

RTOS vs GPOS: Task Scheduling



Meet Higher Throughput



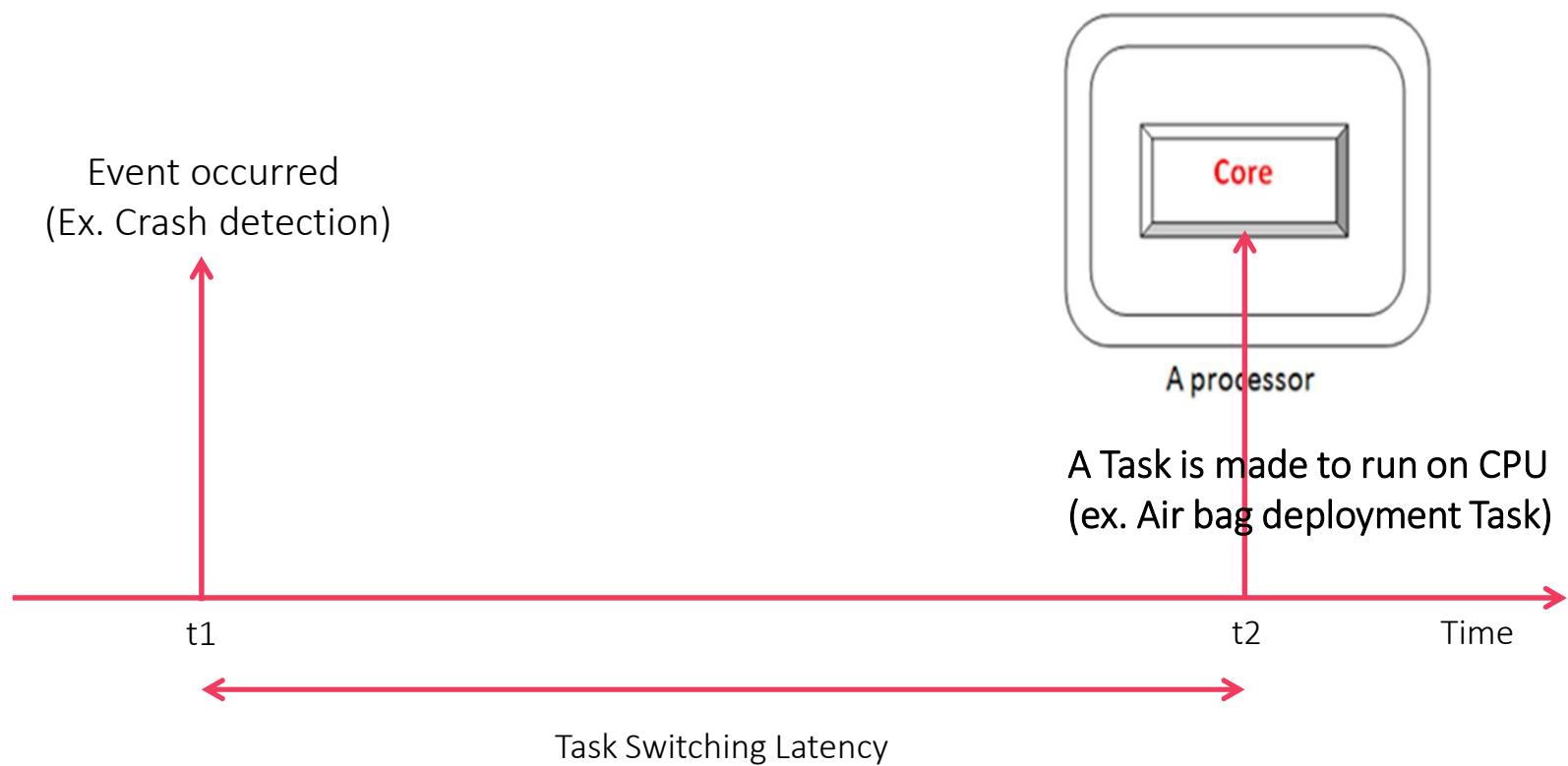
Meet Time Predictability

For the RTOS, achieving predictability or time deterministic nature is more important than throughput, but for the GPOS achieving higher throughput for user convenience is more important

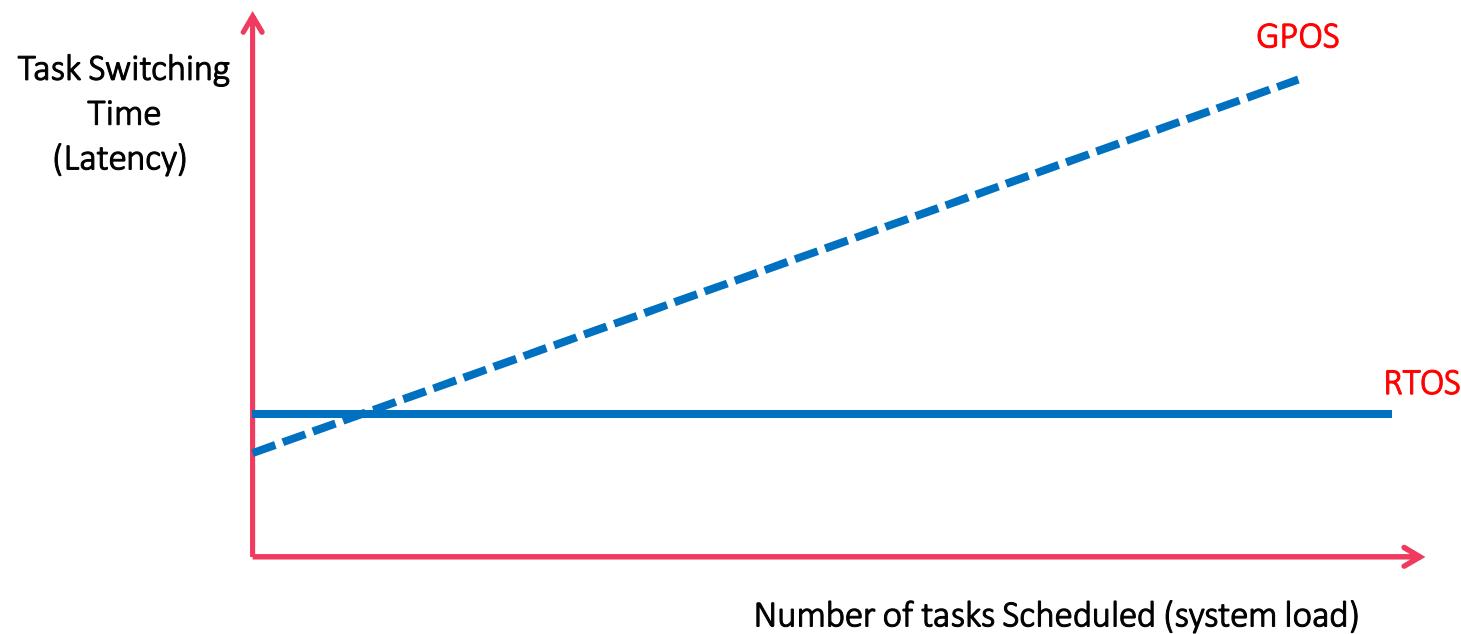
RTOS vs GPOS: Task Switching Latency

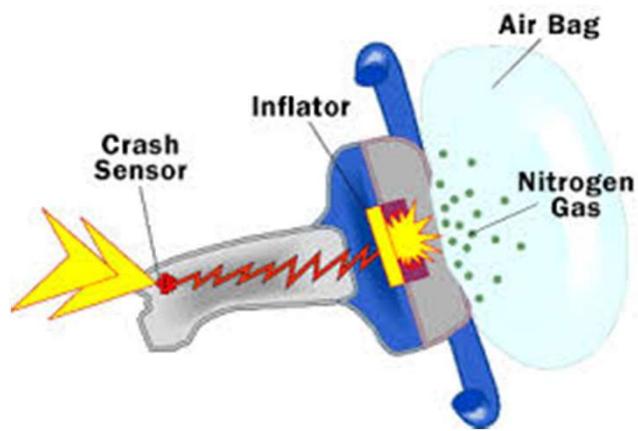
In Computing , Latency means “The time that elapses between a stimulus and the response to it”.

Task switching latency means, that time gap between “A triggering of an event and the time at which the task which takes care of that event is allowed to run on the CPU”

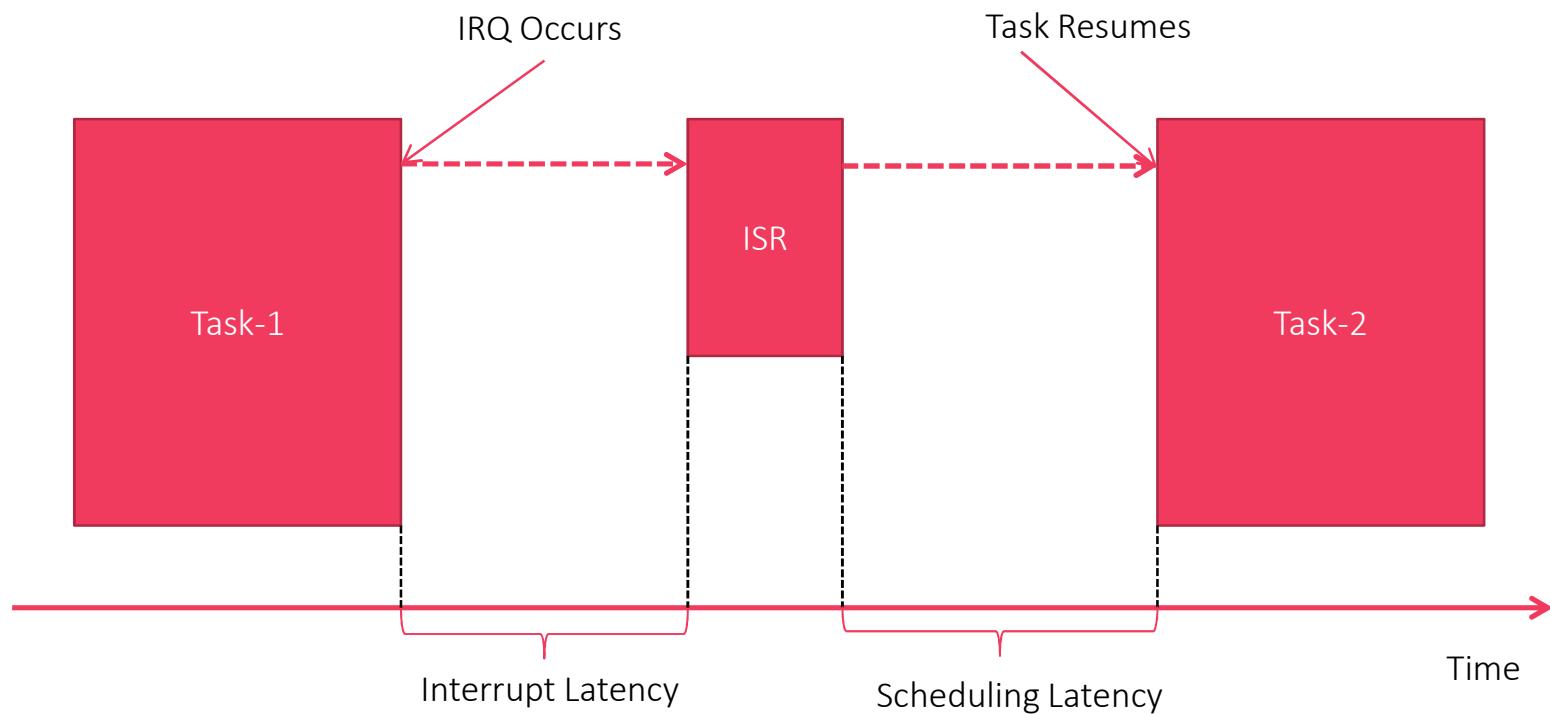


RTOS vs GPOS: Task Switching Latency

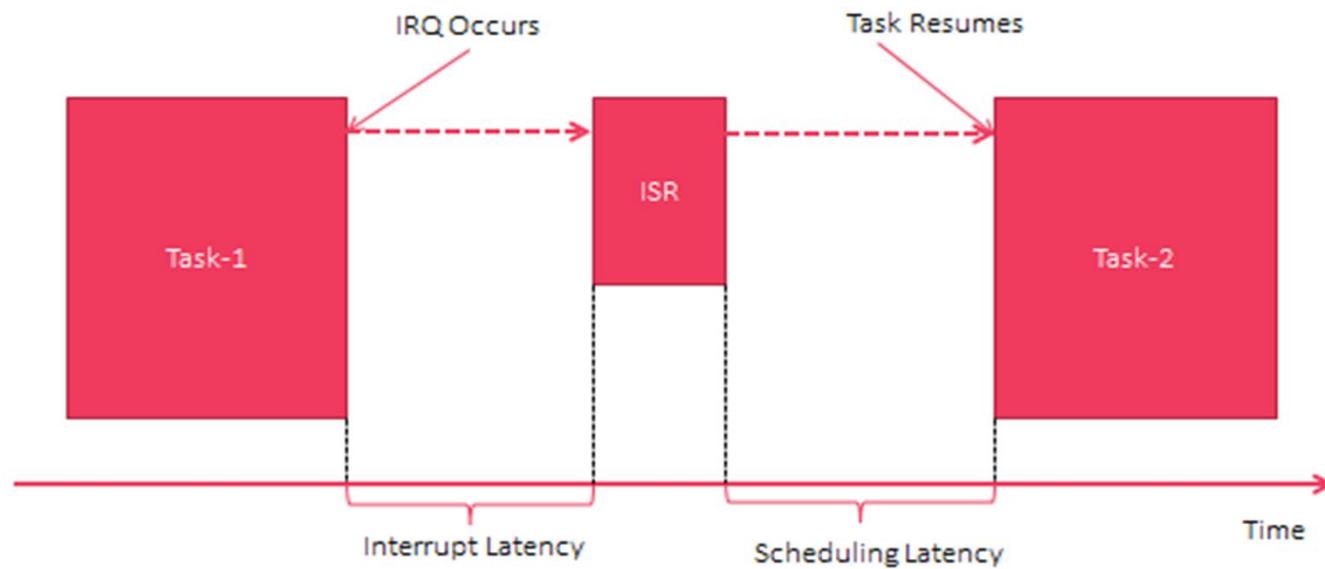




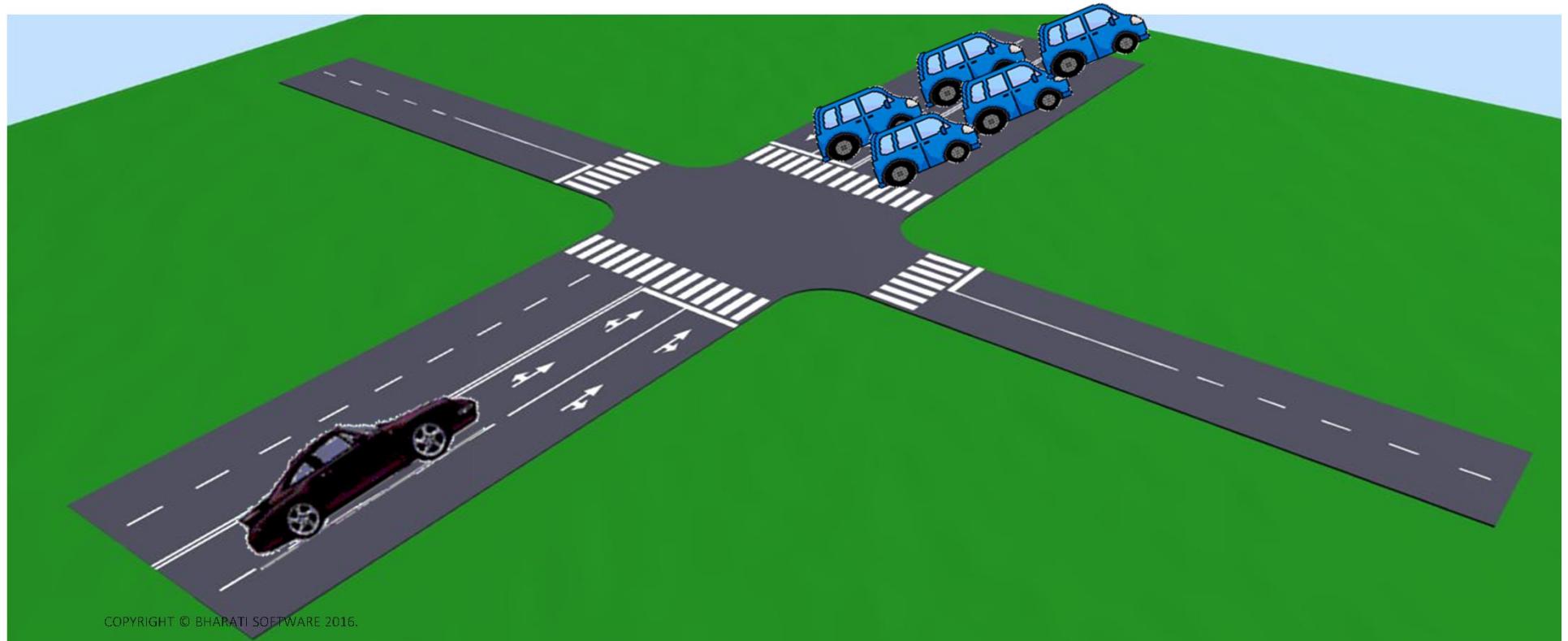
RTOS vs GPOS: Interrupt Latency



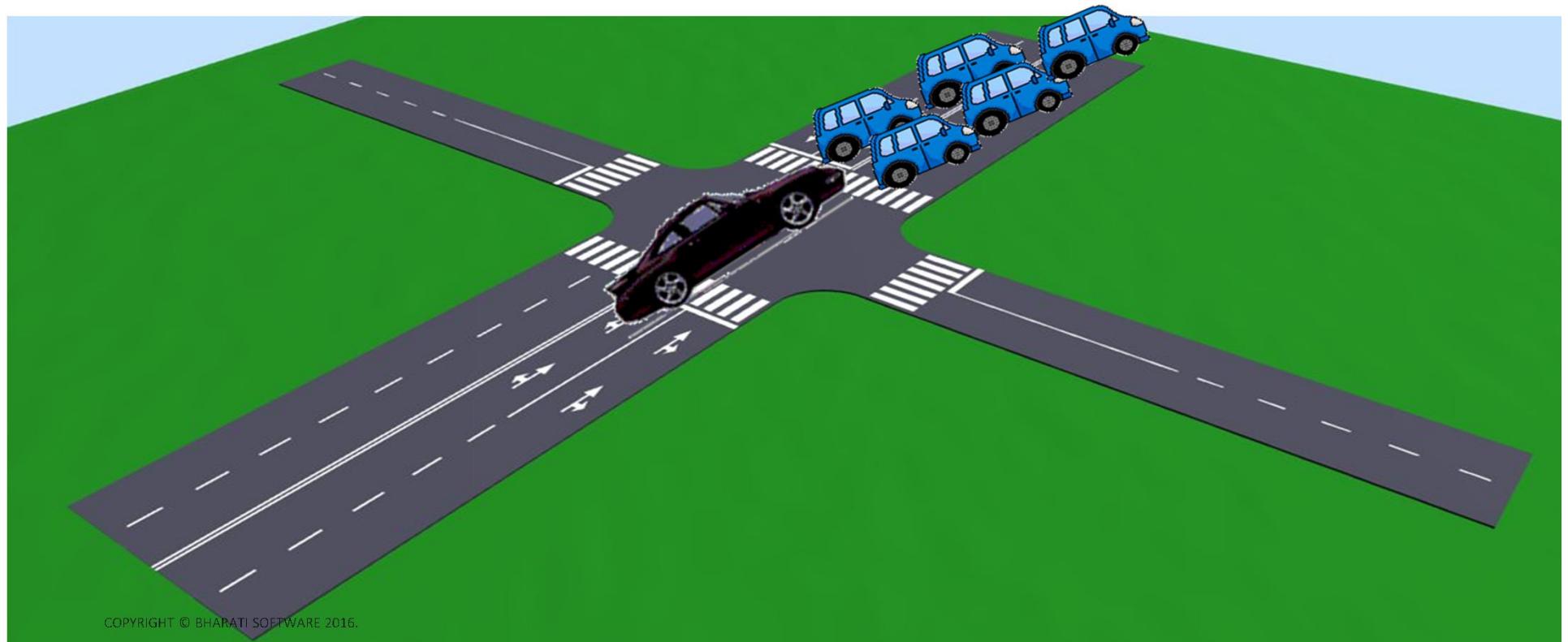
RTOS vs GPOS: Interrupt Latency



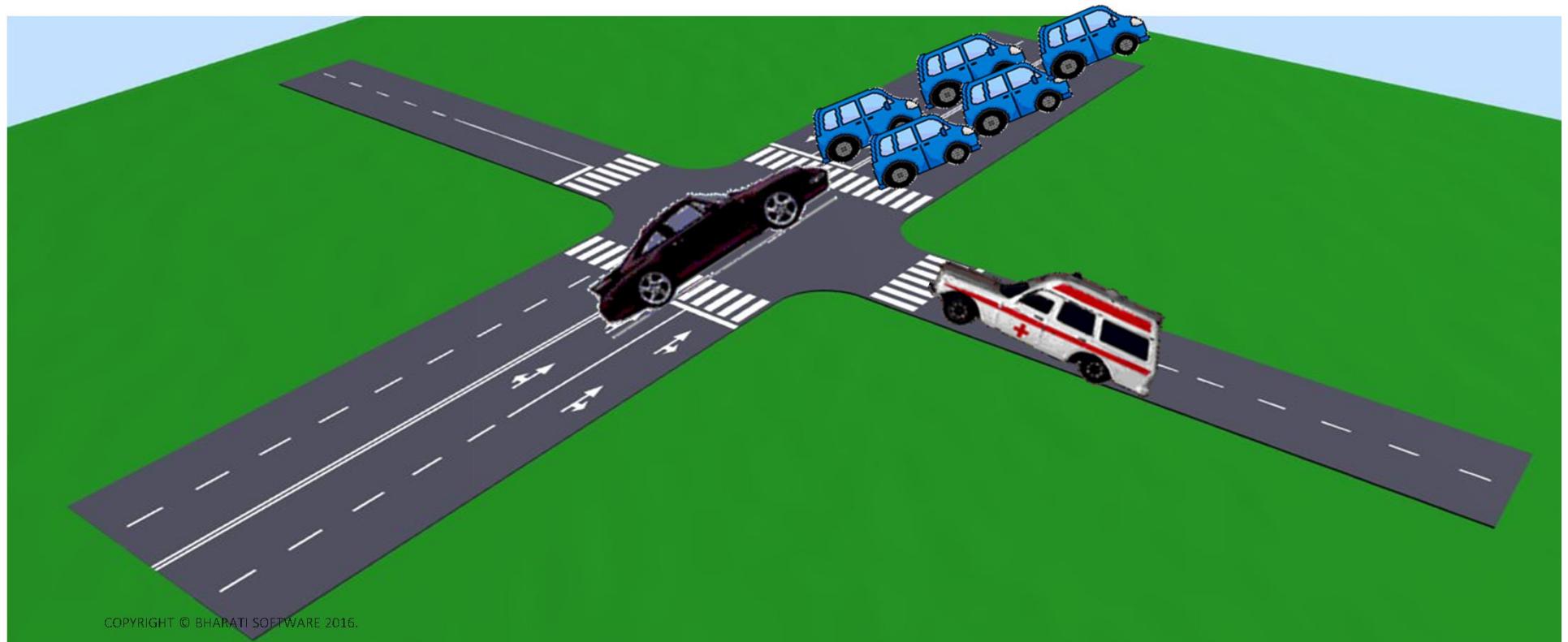
RTOS vs GPOS: Priority Inversion



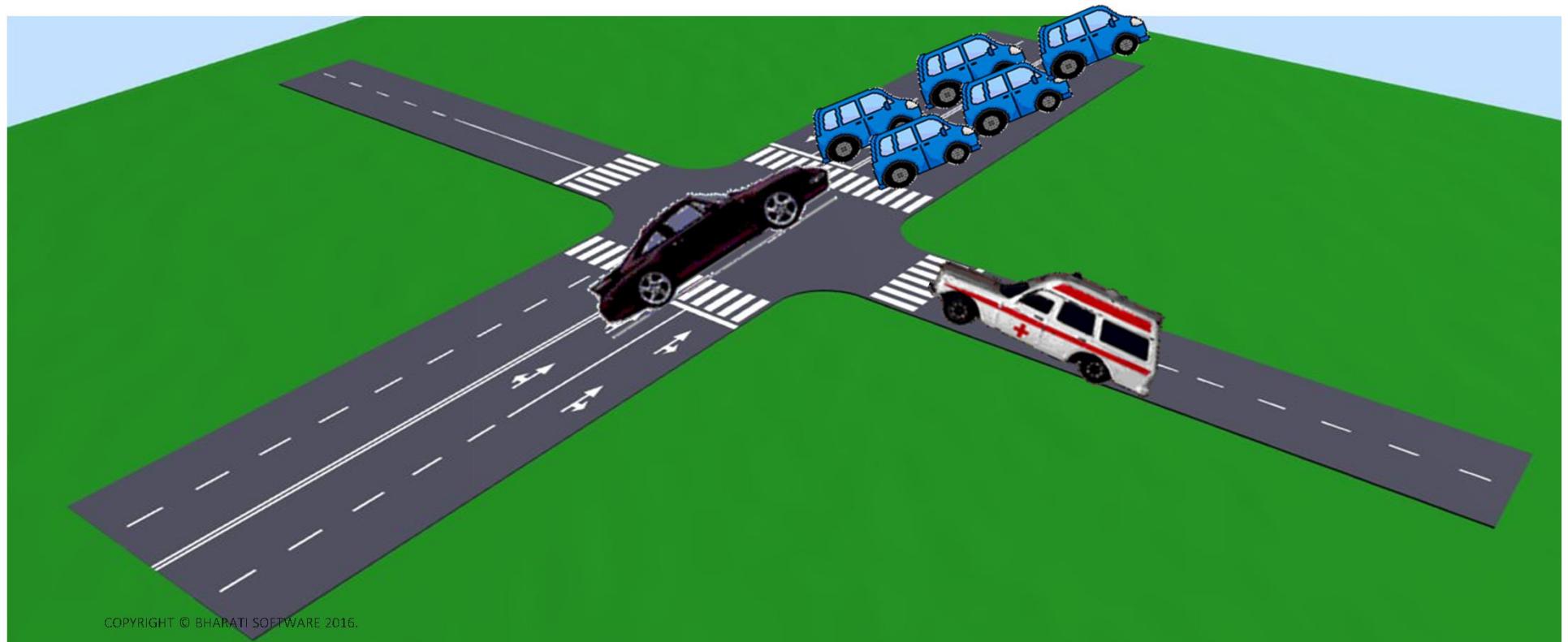
RTOS vs GPOS: Priority Inversion



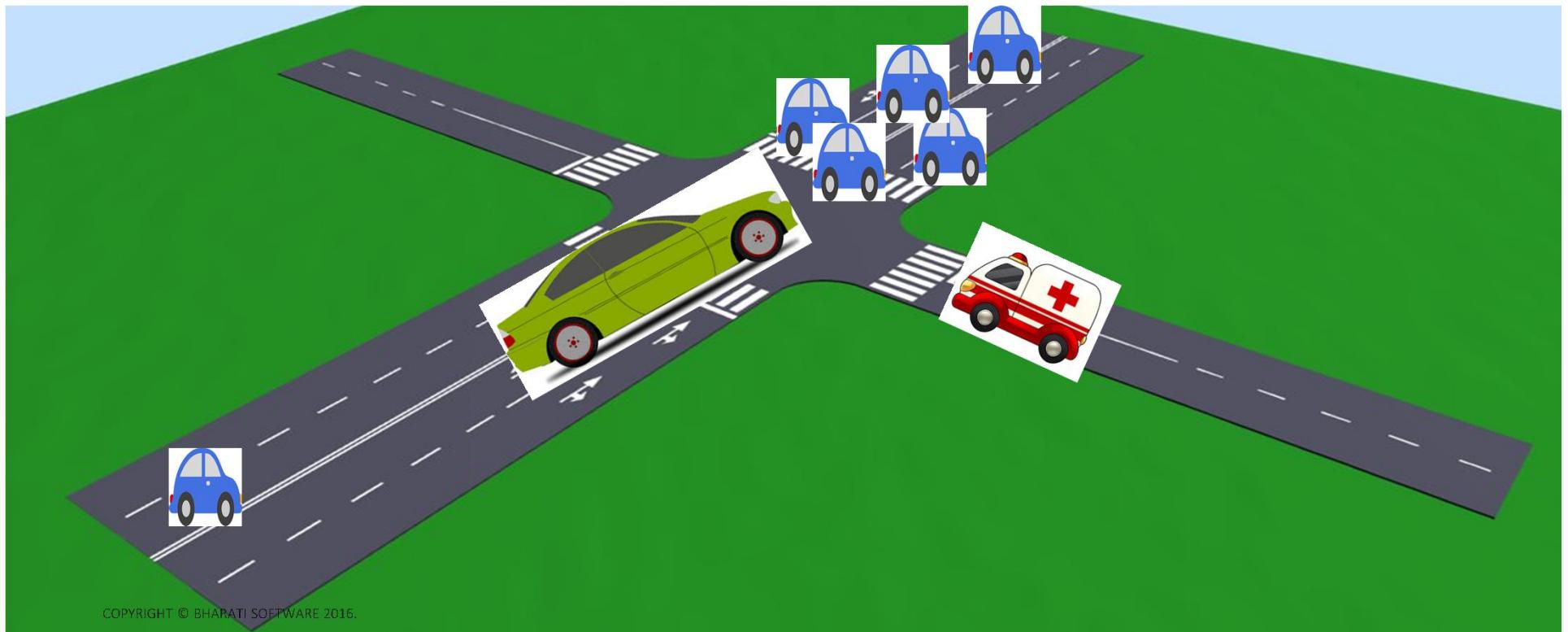
RTOS vs GPOS: Priority Inversion



RTOS vs GPOS: Priority Inversion



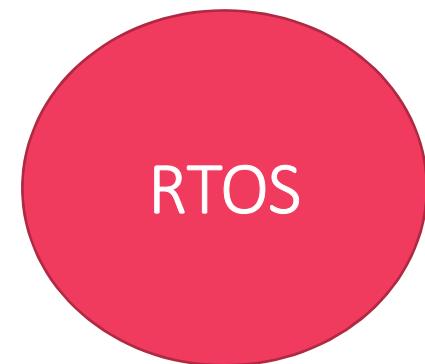
RTOS vs GPOS: Priority Inversion



RTOS vs GPOS: Priority Inversion



Priority inversion effects are in-significant



Priority inversion effects must be solved

RTOS vs GPOS: Kernel Preemption

Preemption

In computing **preemption** is the act of temporarily removing the task from the running state without its co-operation

In RTOS , Threads execute in order of their priority. If a high-priority thread becomes ready to run, it will, within a small and bounded time interval, take over the CPU from any lower-priority thread that may be executing that we call preemption.

The lower priority task will be made to leave CPU, if higher priority task wants to execute.

RTOS vs GPOS: Kernel Preemption

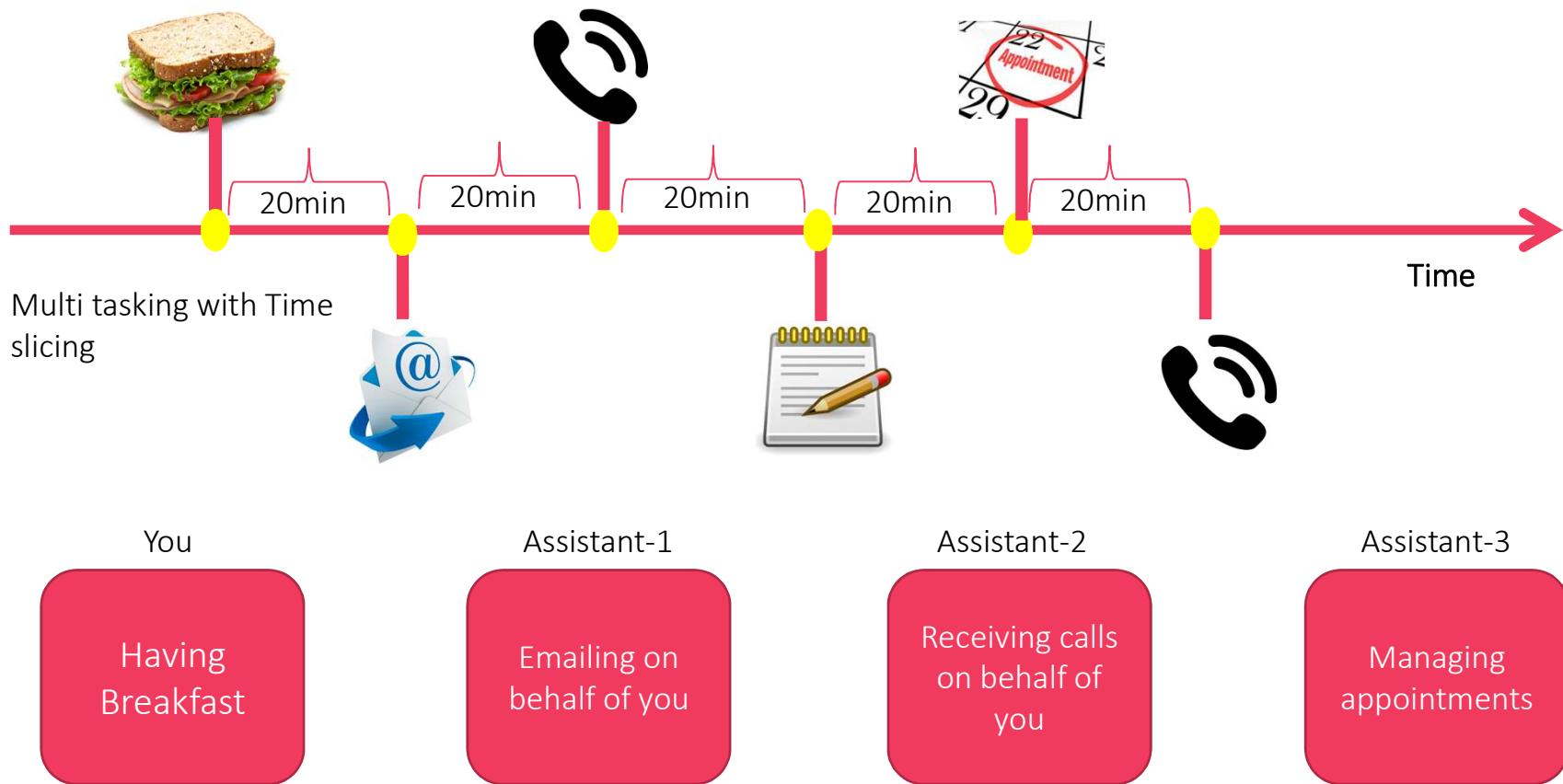
The kernel operations of a RTOS are pre-emptible whereas the kernel operations of a GPOS are not pre-emptible

What are the features that a RTOS has but a GPOS doesn't ?

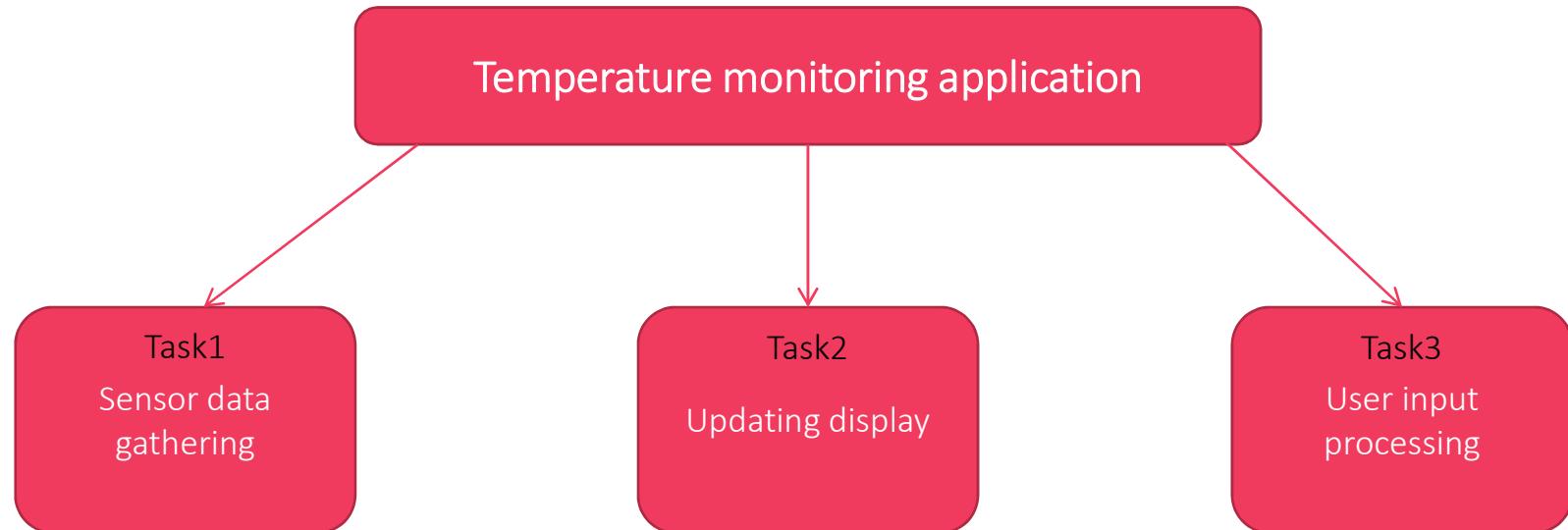
1. Priority based preemptive scheduling mechanism
2. No or very short Critical sections which disables the preemption
3. Priority inversion avoidance
4. Bounded Interrupt latency
5. Bounded Scheduling latency , etc.

What is Multi-Tasking ??

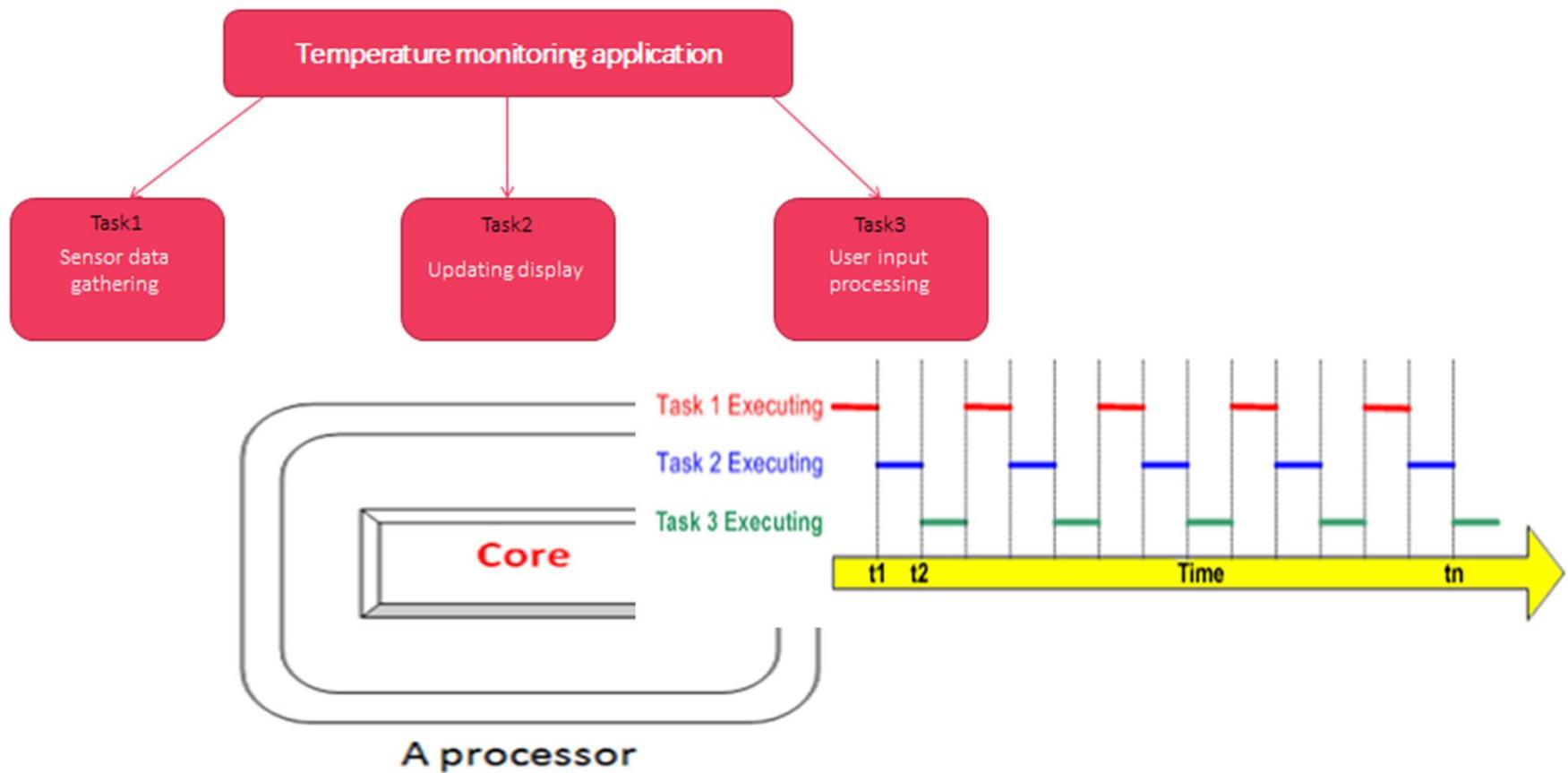




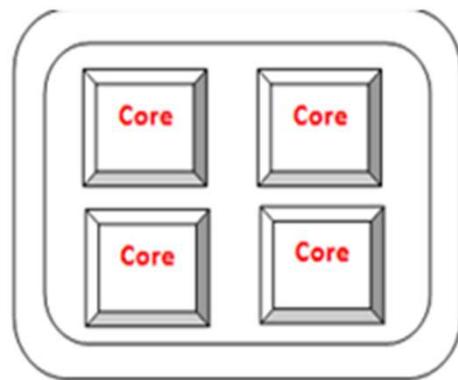
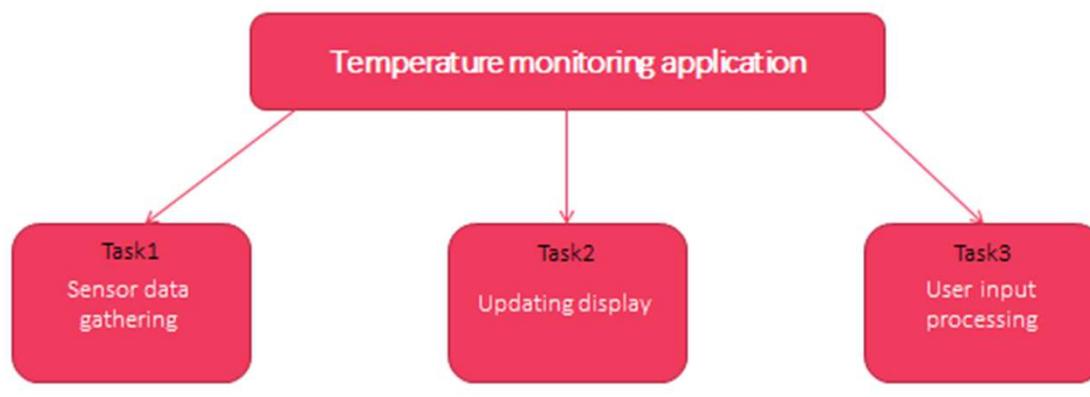
Application and tasks



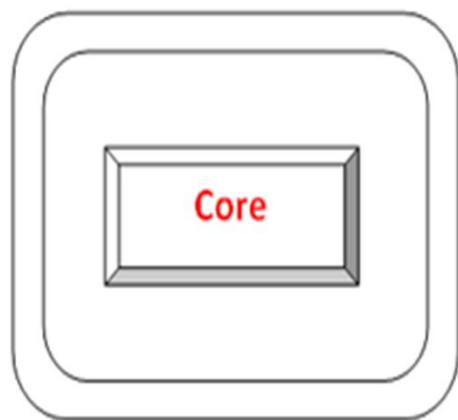
Application and tasks



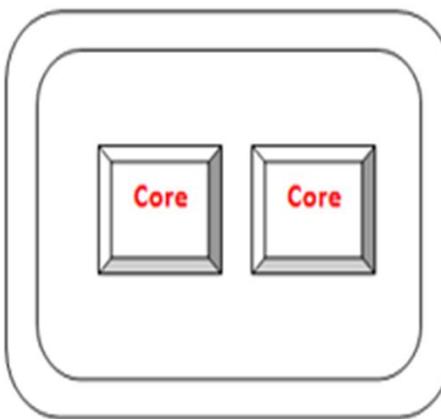
Application and tasks



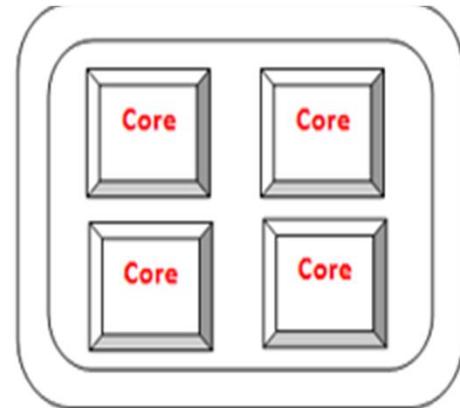
A processor with 4 cores



A processor



A processor with 2 cores



A processor with 4 cores

Any point in time
only 1 task can run

Any point in time 2 or 4
task can run

Scheduler



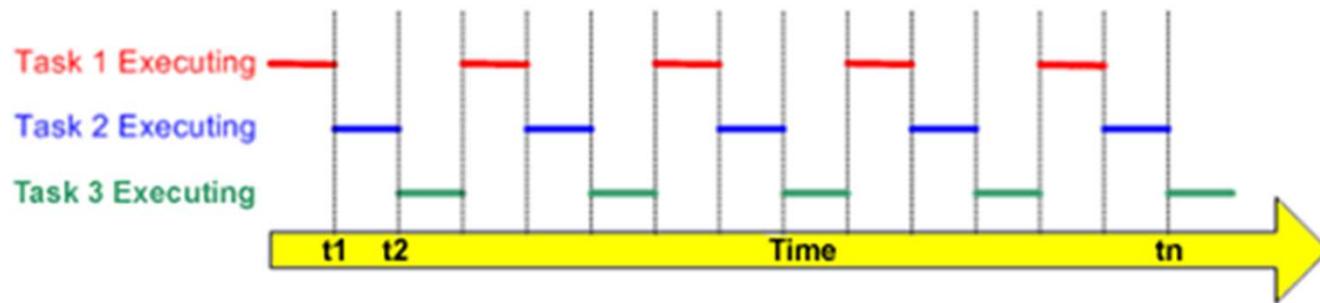
I have my own **scheduling policy** , I act according that .

Task List

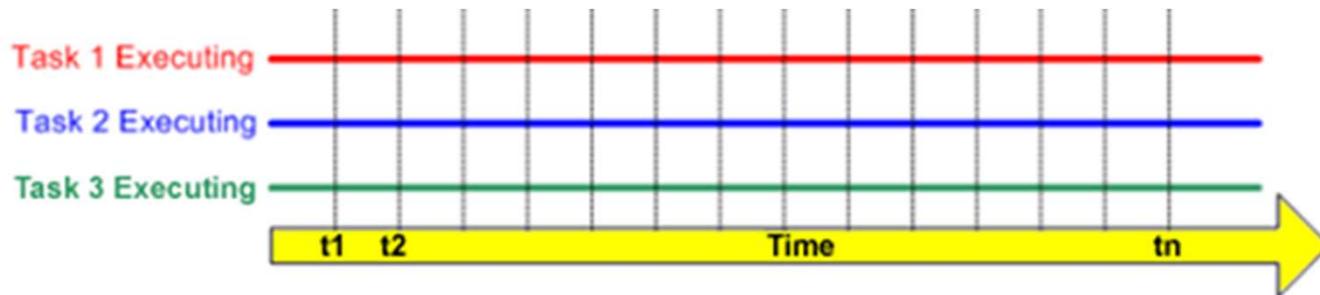
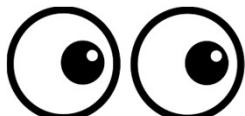
You can configure **my scheduling policy**

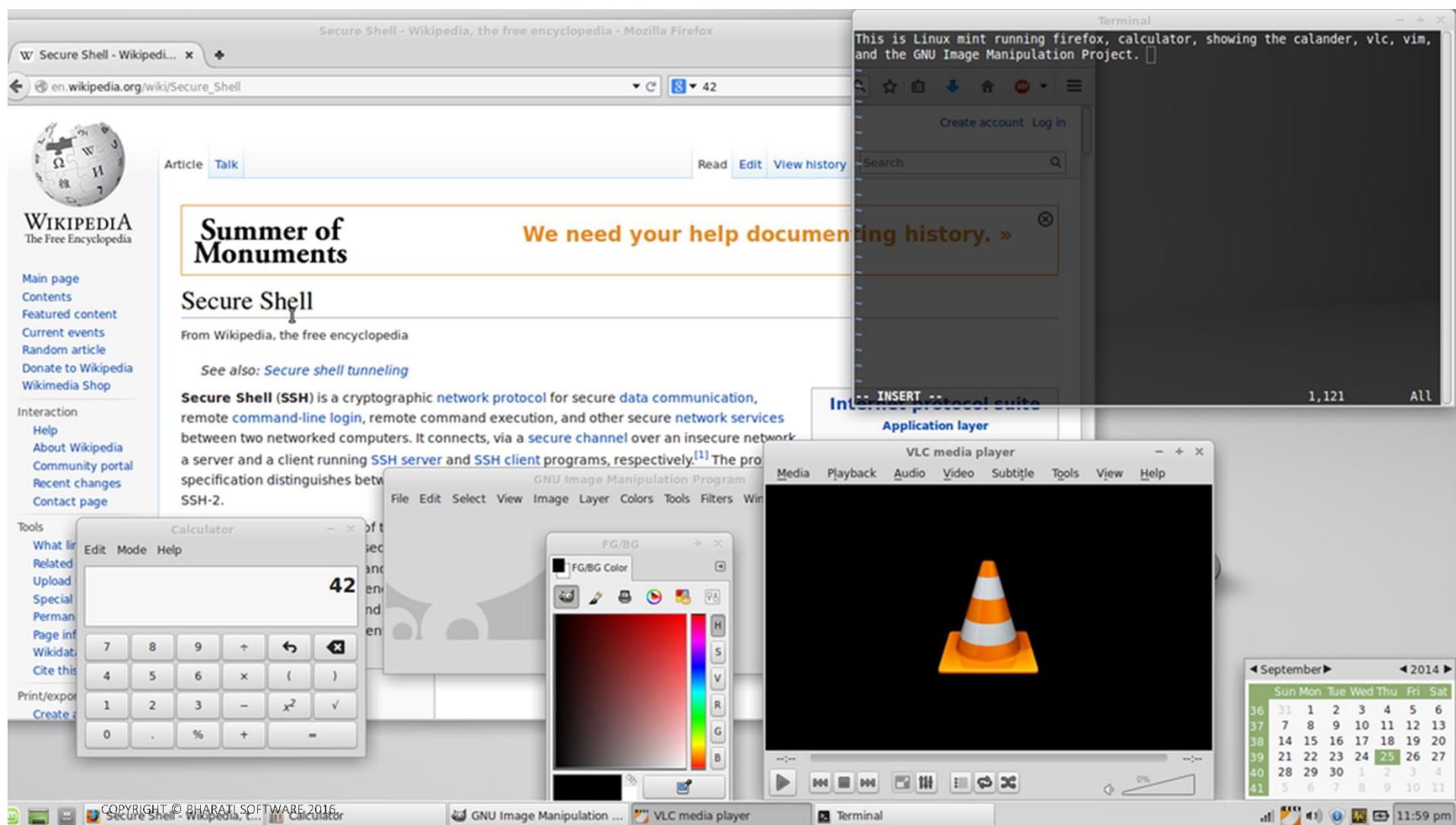


This is how multiple tasks run on CPU



Which gives the illusion that all tasks are executing simultaneously





FreeRTOS Overview

COPYRIGHT © BHARATI SOFTWARE 2016.

About freertos.org

COPYRIGHT © BHARATI SOFTWARE 2016.

Supported architectures [edit]

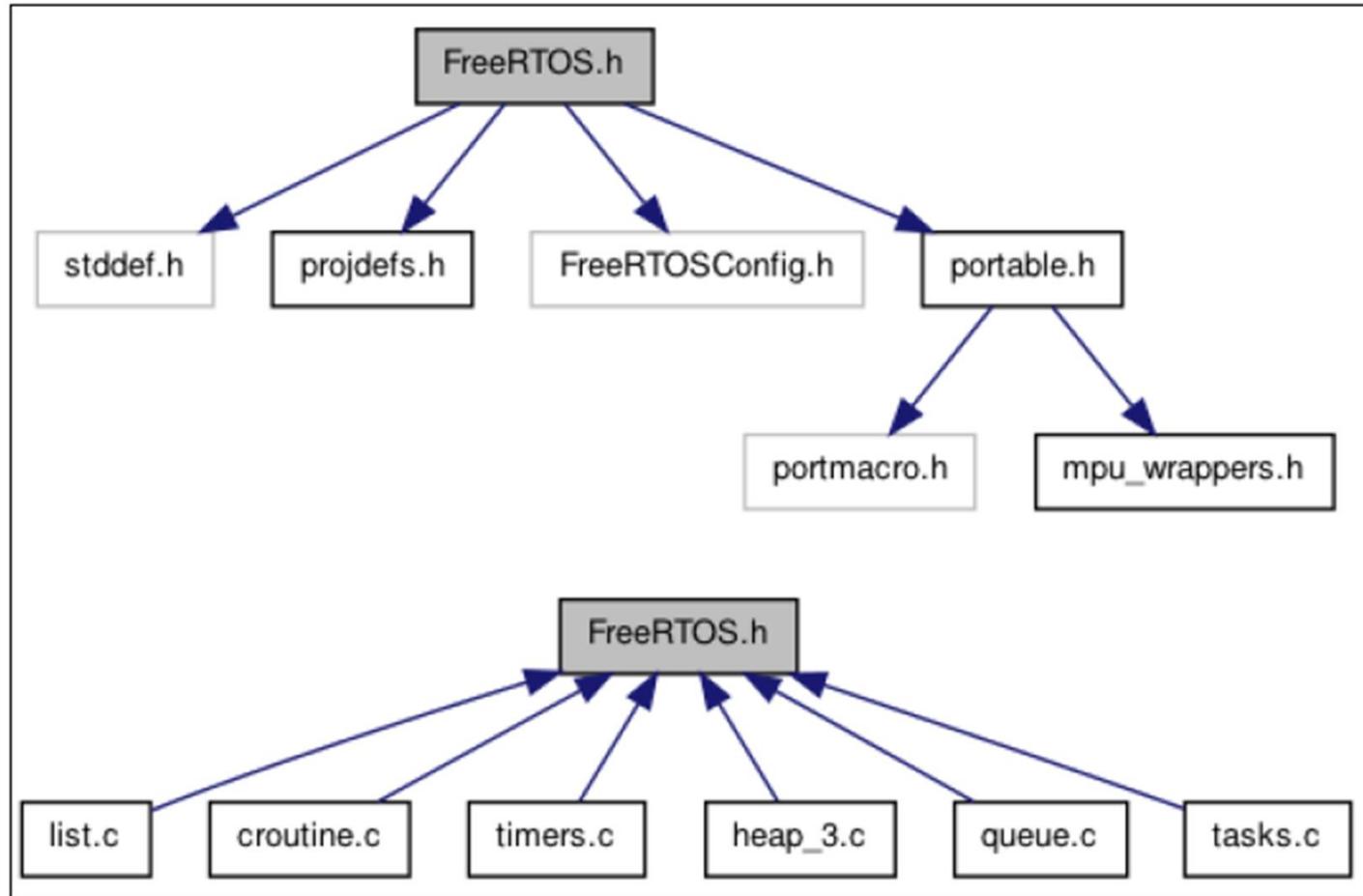
- Altera Nios II
- ARM architecture
 - ARM7
 - ARM9
 - ARM Cortex-M0
 - ARM Cortex-M0+
 - ARM Cortex-M3
 - ARM Cortex-M4
 - ARM Cortex-M7
 - ARM Cortex-A
- Atmel
 - Atmel AVR
 - AVR32
 - SAM3
 - SAM4
 - SAM7
 - SAM9
 - SAM D20
 - SAM L21
- Cortus
 - APS1
 - APS3
 - APS3R
 - APS5
 - FPS6
 - FPS8
 - Cypress
 - PSoC
 - Energy Micro
 - EFM32
 - Fujitsu
 - FM3 series
 - MB91460 series
 - MB96340
 - Freescale
 - Coldfire V1
 - Coldfire V2
 - HCS12
 - Kinetis
- IBM
 - PPC405
 - PPC404
- Infineon
 - TriCore
 - Infineon XMC4000
- Intel
 - x86
 - 8052
- PIC microcontroller
 - PIC18
 - PIC24
 - dsPIC
 - PIC32
- Microsemi
 - SmartFusion
- Multiclet
 - Multiclet P1
- NXP
 - LPC1000
 - LPC2000
 - LPC4300
- Renesas
 - 78K0R
 - RL78
 - H8/S
 - RX600
 - RX200
 - SuperH
 - V850
- STMicroelectronics
 - STM32
 - STR7
- Texas Instruments
 - MSP430
 - Stellaris
 - Hercules (TMS570LS04 & RM42)
- Xilinx
 - MicroBlaze
 - Zynq-7000
- Espressif
 - ESP8266ex

<https://en.wikipedia.org/wiki/FreeRTOS>

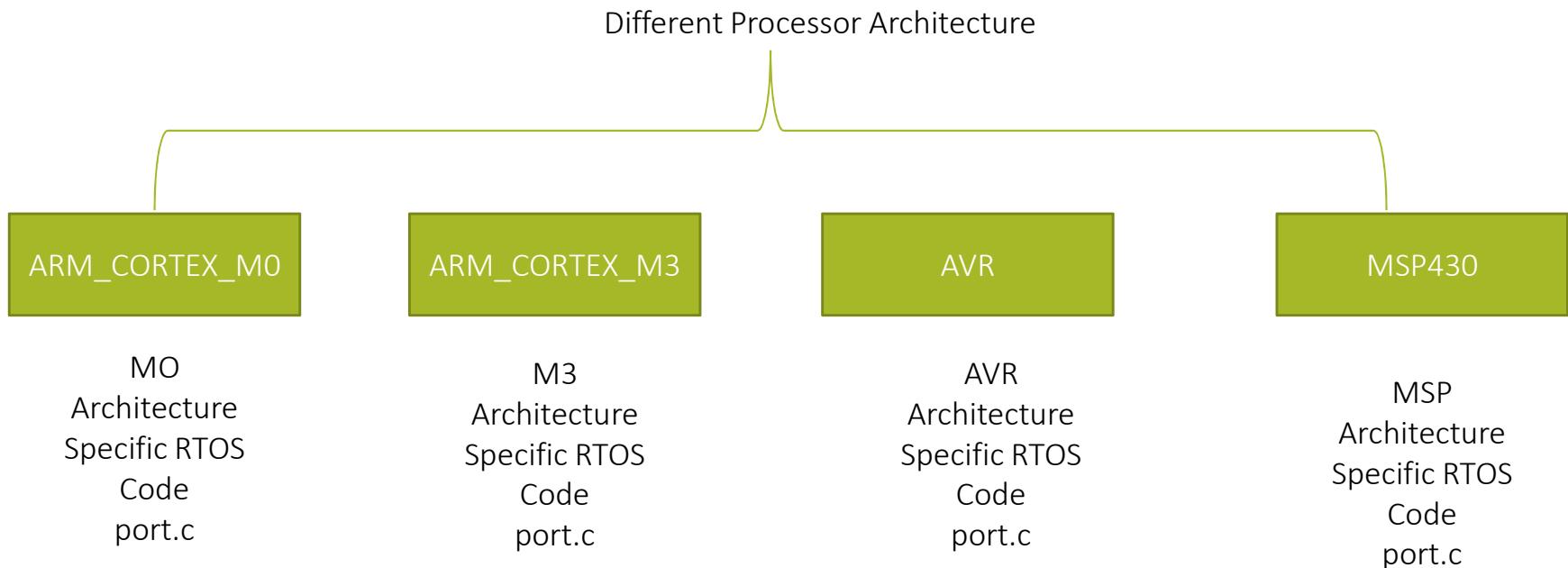
COPYRIGHT © BHARATI SOFTWARE 2016.

Coming Up-Next :

FreeRTOS file structure and Hierarchy

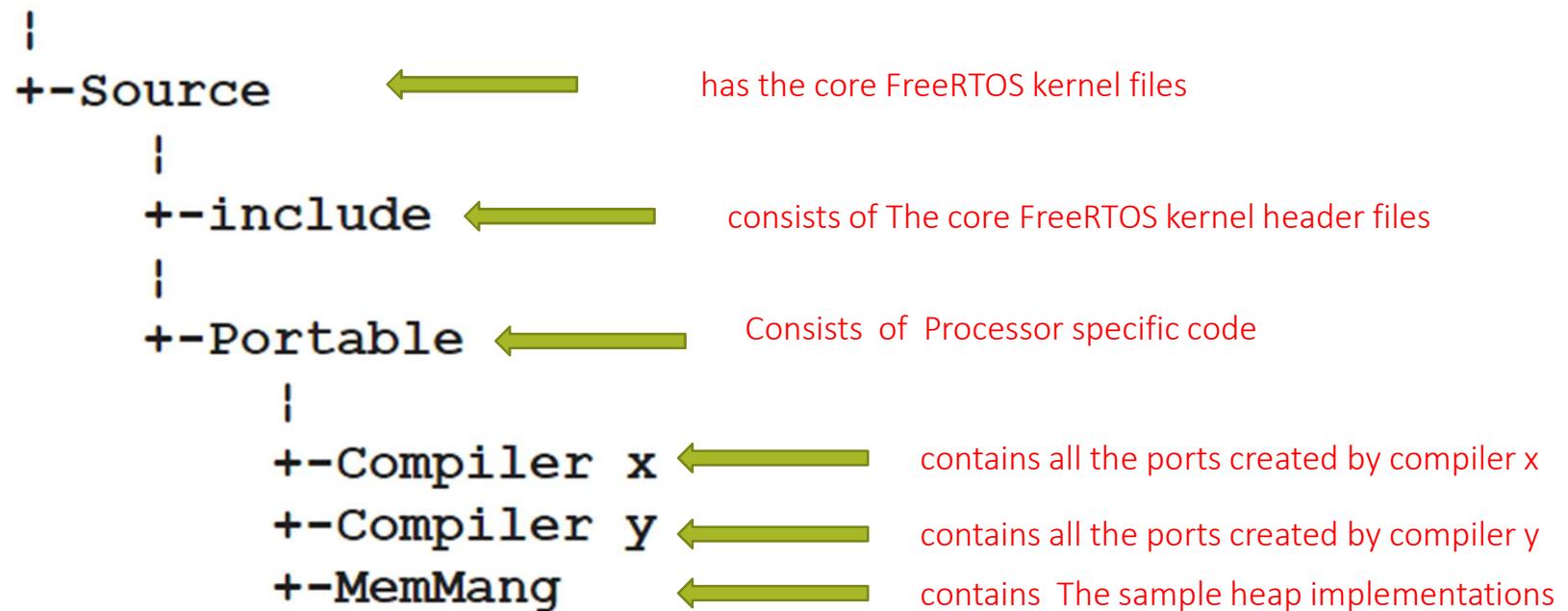


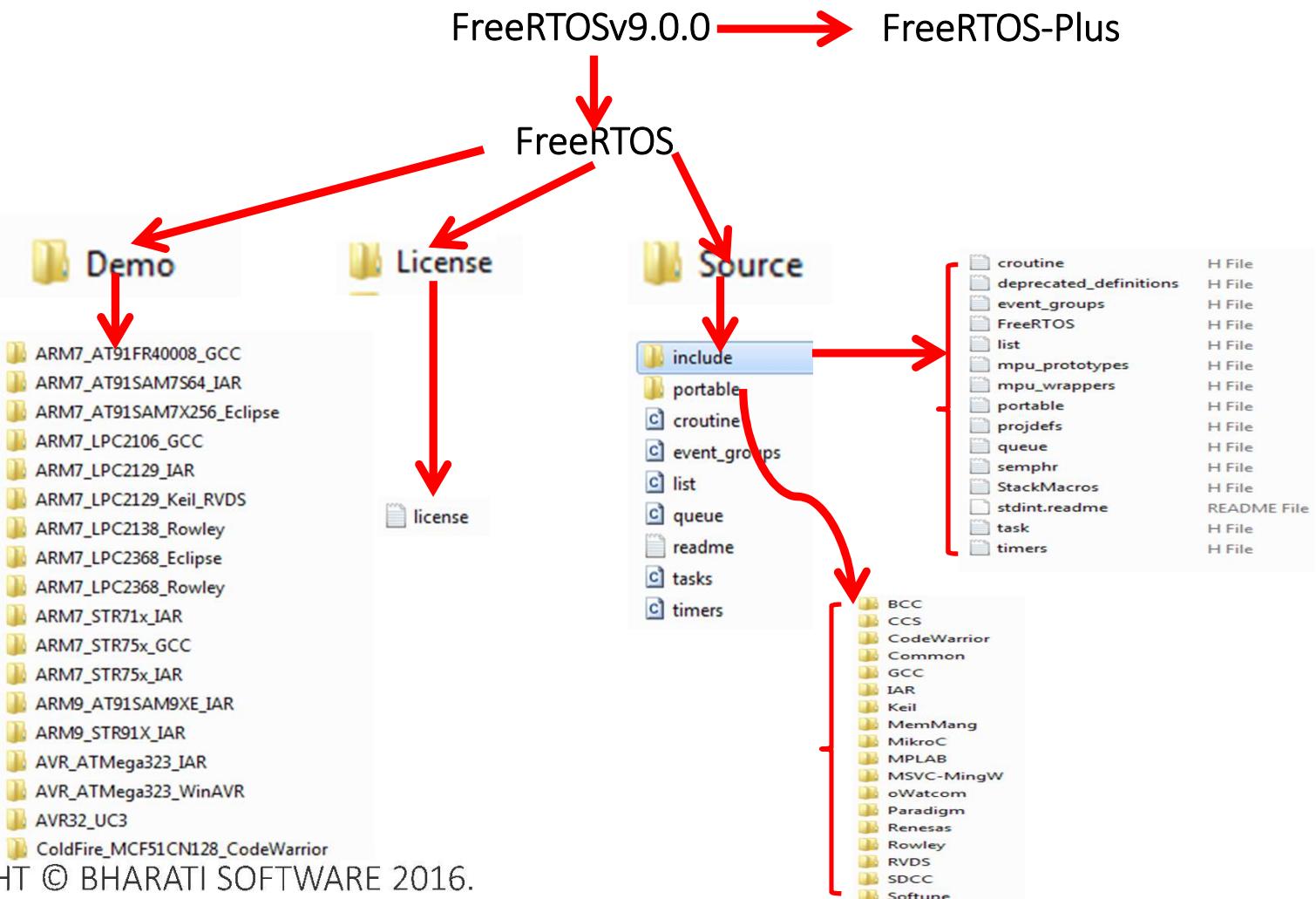
COPYRIGHT © BHARATI SOFTWARE 2016.



COPYRIGHT © BHARATI SOFTWARE 2016.

FreeRTOS





COPYRIGHT © BHARATI SOFTWARE 2016.

FreeRTOS Kernel features

COPYRIGHT © BHARATI SOFTWARE 2016.

FreeRTOS Kernel features

*Pre-emptive or
co-operative
operation*

*Very flexible
task priority
assignment*

Software timers

Queues

*Binary
semaphores*

*Counting
semaphores*

FreeRTOS Kernel features

*Recursive
semaphores*

Mutexes

*Tick hook
functions*

*Idle hook
functions*

*Stack overflow
checking*

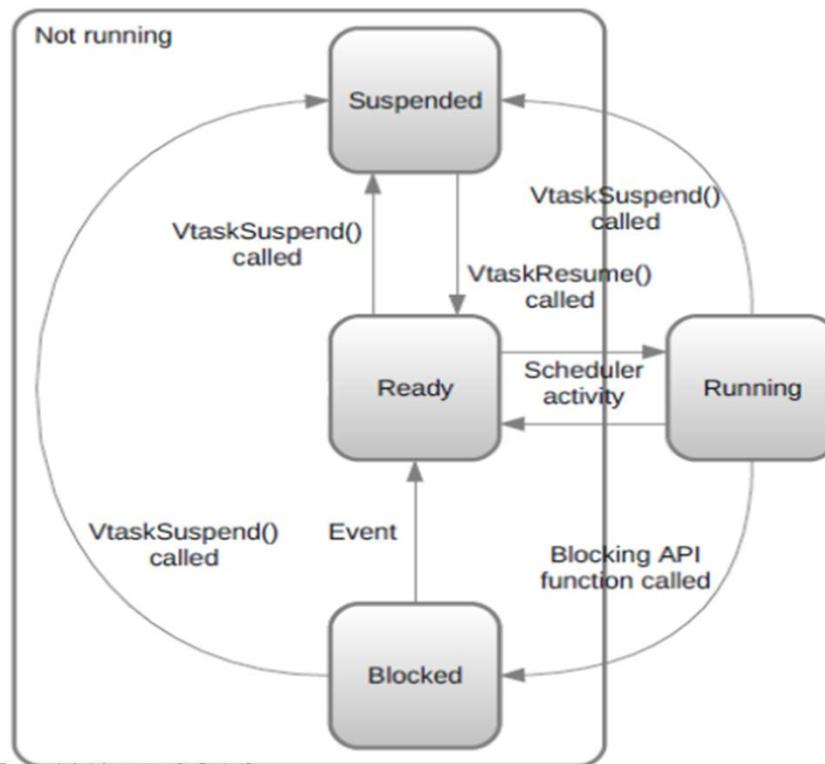
*Trace hook
macros*

Coming Up-Next

Overview of FreeRTOS Task Management

COPYRIGHT © BHARATI SOFTWARE 2016.

Task States

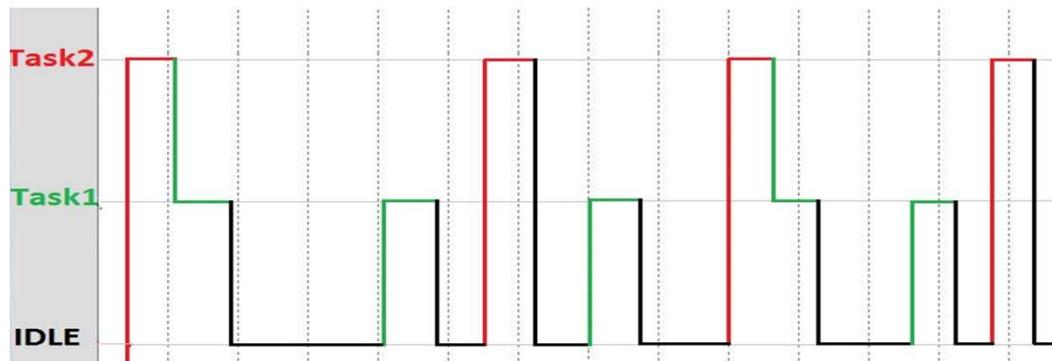


COPYRIGHT © BHARATI SOFTWARE 2016.

FreeRTOS other features

COPYRIGHT © BHARATI SOFTWARE 2016.

Idle Task



The Idle task is created automatically when the RTOS scheduler is started to ensure there is always at least one task that is able to run.

It is created at the lowest possible priority to ensure it does not use any CPU time if there are higher priority application tasks in the ready state.

Some facts about Idle Task

It is a lowest priority task which is automatically created when the scheduler is started

The idle task is responsible for freeing memory allocated by the RTOS to tasks that have been deleted

When there are no tasks running, Idle task will always run on CPU.

You can give an application hook function in the idle task to send the CPU to low power mode when there are no useful tasks are executing.

Idle Task hook function

Idle task hook function implements a callback from idle task to your application

You have to enable the idle task hook function feature by setting this config item
`configUSE_TICK_HOOK` to 1 within FreeRTOSConfig.h

Then implement the below function in your application

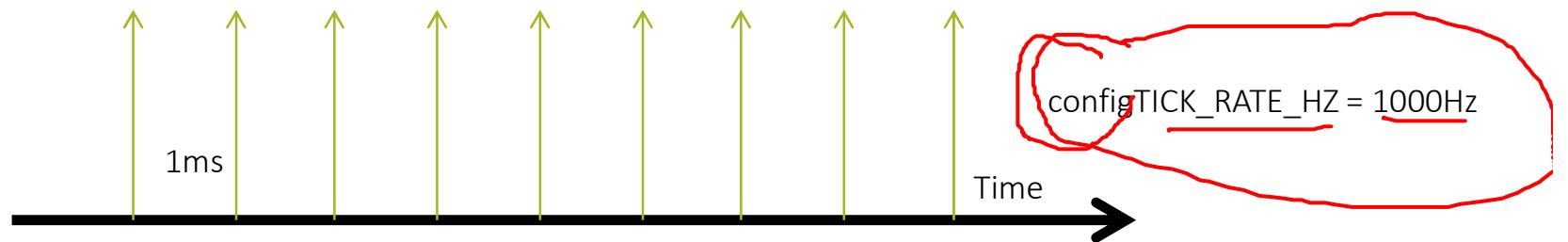
```
void vApplicationIdleHook( void );
```

That's it , whenever idle task is allowed to run, your hook function will get called, where you can do some useful stuffs like sending the MCU to lower mode to save power

When the hook function is called, care must be taken that the idle hook function does not call any API functions that could cause it to block

COPYRIGHT © BHARATI SOFTWARE 2016.

Tick hook function



The tick interrupt can optionally call an application defined hook (or callback) function - the tick hook
1~5

You can use tick hook function to implement timer functionality.

The tick hook will only get called if configUSE_TICK_HOOK is set to 1 within FreeRTOSConfig.h

Implement this function : void vApplicationTickHook(void); in your application

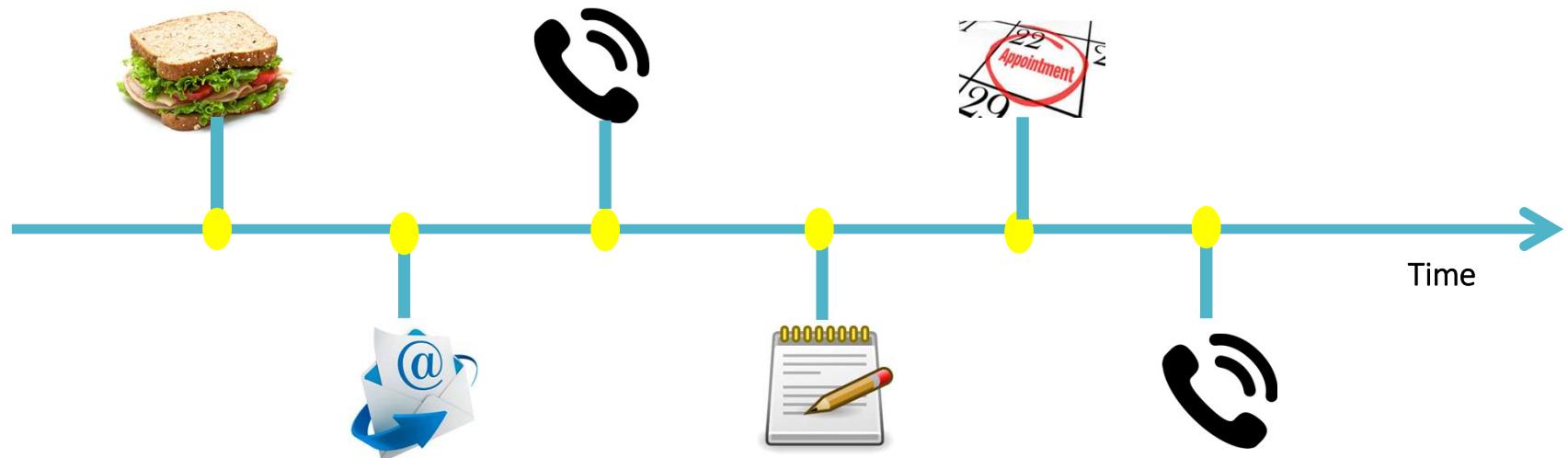
vApplicationTickHook() executes from within an ISR so must be very short, not use much stack, and not call any API functions that don't end in "FromISR"

COPYRIGHT © BHARATI SOFTWARE 2016.

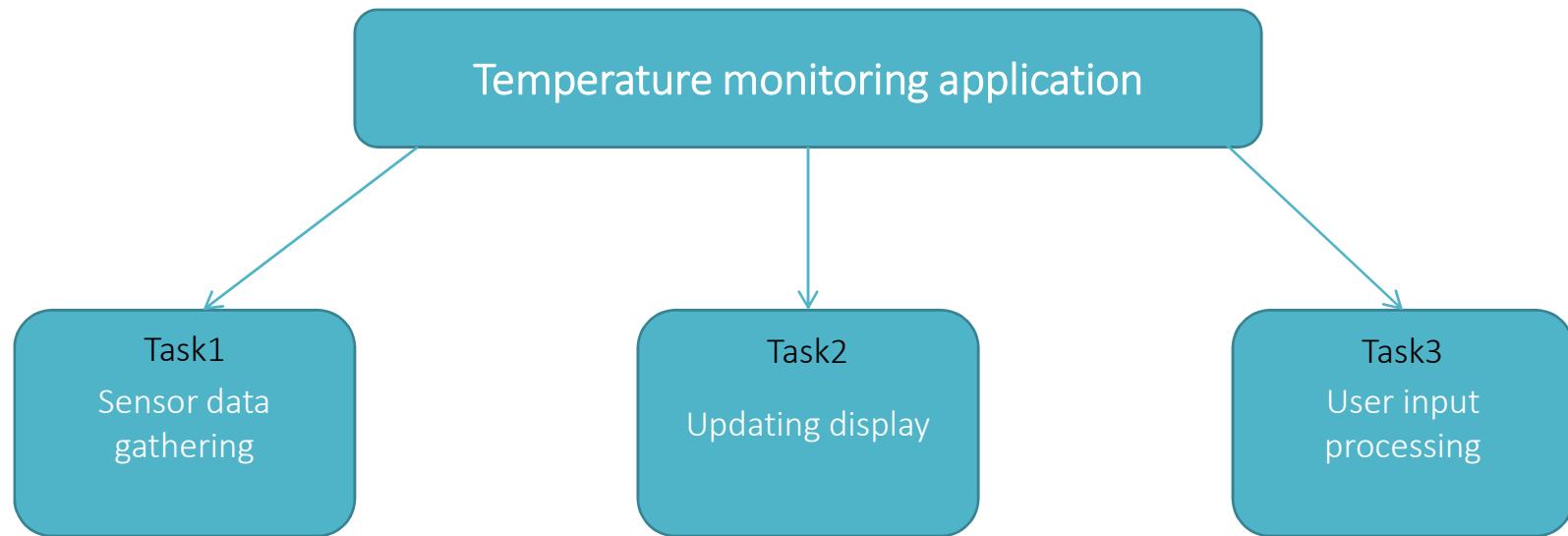
Free RTOS Task Creation

COPYRIGHT © BHARATI SOFTWARE 2016.

Typical Tasks of a day !



Application and tasks



So, how do we Create and implement a task in FreeRTOS ?

Task Creation

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,  
                        const char * const pcName,  
                        unsigned short usStackDepth,  
                        void *pvParameters,  
                        UBaseType_t uxPriority,  
                        TaskHandle_t *pxCreatedTask  
);
```

Task Implementation

```
void vATaskFunction( void *pvParameters )  
{  
    for( ;; )  
    {  
        -- Task application code here. --  
    }  
  
    vTaskDelete( NULL );  
}
```

Task Implementation Function(Task Function)

```
void ATaskFunction( void *pvParameters )
{
    /* Variables can be declared just as per a normal function. Each instance
       of a task created using this function will have its own copy of the
       iVariableExample variable. This would not be true if the variable was
       declared static - in which case only one copy of the variable would exist
       and this copy would be shared by each created instance of the task. */
    int iVariableExample = 0;

    /* A task will normally be implemented as in infinite loop. */
    for( ; ; )
    {
        /* The code to implement the task functionality will go here. */
    }

    /* Should the task implementation ever break out of the above loop
       then the task must be deleted before reaching the end of this function.
       The NULL parameter passed to the vTaskDelete() function indicates that
       the task to be deleted is the calling (this) task. */
    vTaskDelete( NULL );
}
```

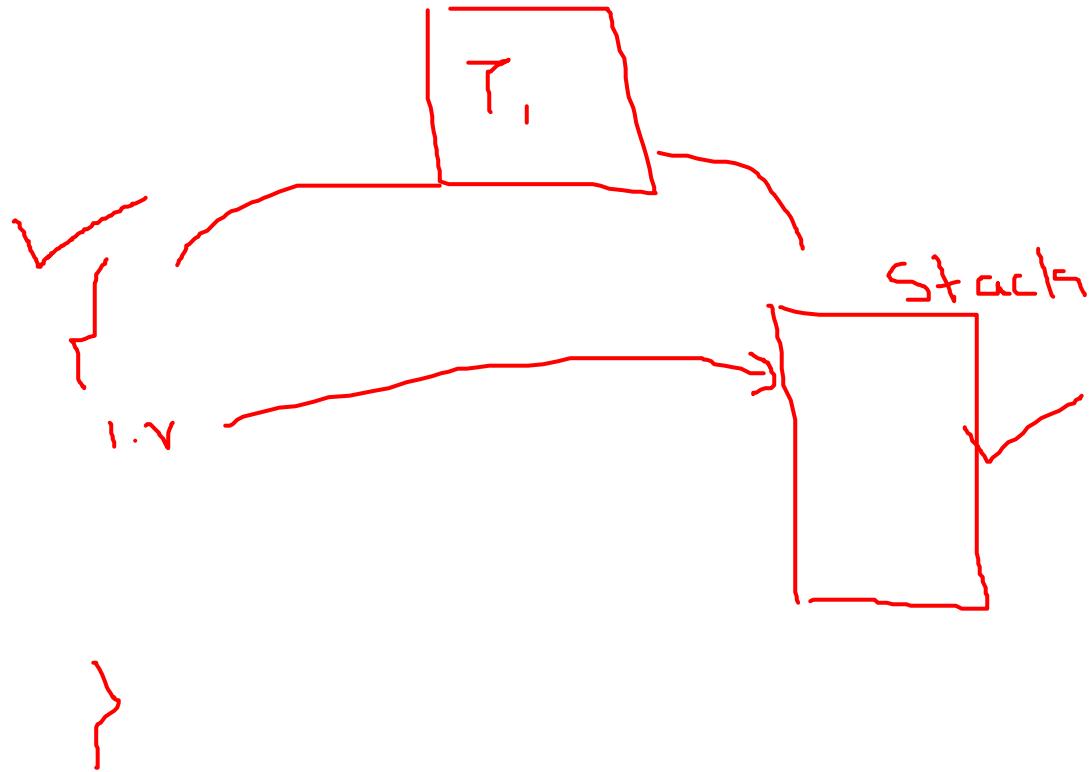
APIs to Create and Schedule a Task

COPYRIGHT © BHARATI SOFTWARE 2016.

API to Create a FreeRTOS Task



```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,  
                        const char * const pcName,  
                        unsigned short usStackDepth,  
                        void *pvParameters,  
                        UBaseType_t uxPriority,  
                        TaskHandle_t *pxCreatedTask );
```

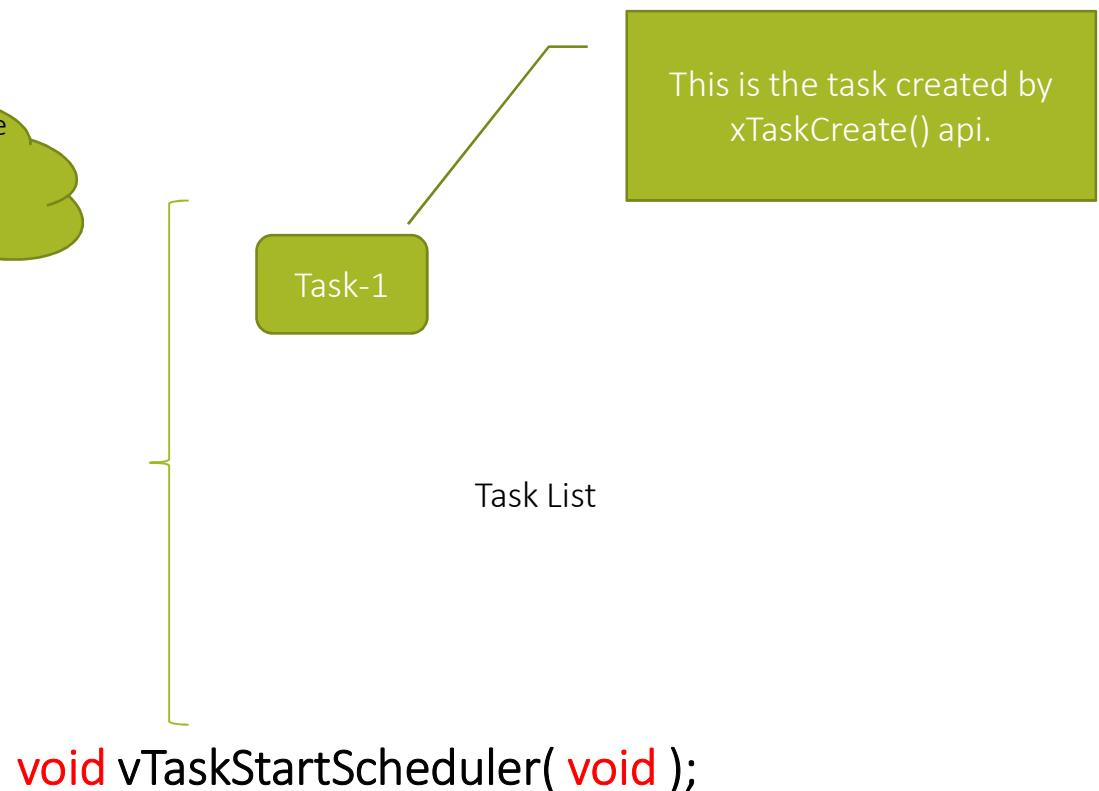
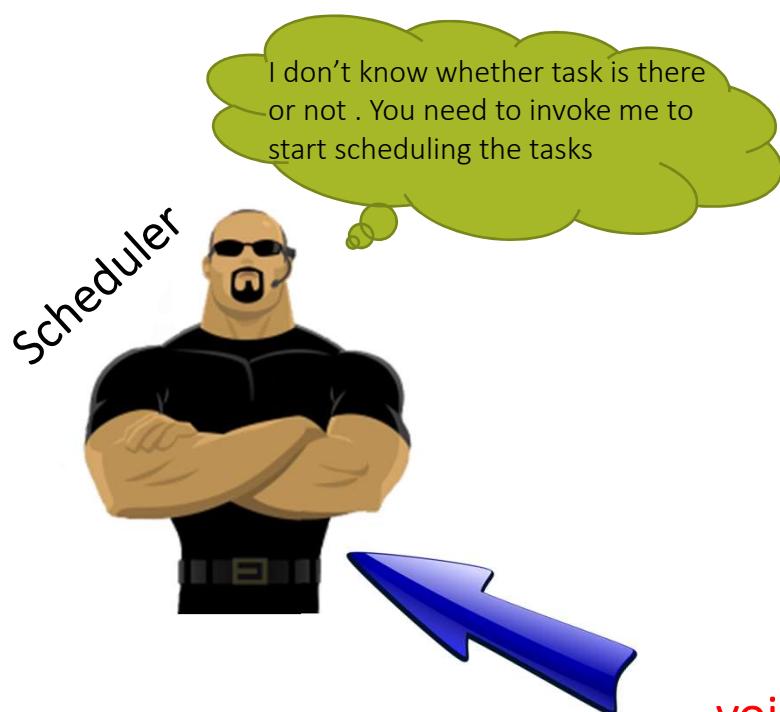


COPYRIGHT © BHARATI SOFTWARE 2016.

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,  
                        const char * const pcName,  
                        unsigned short usStackDepth,  
                        void *pvParameters,  
                        UBaseType_t uxPriority,  
                        TaskHandle_t *pxCreatedTask );
```

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,  
                        const char * const pcName,  
                        unsigned short usStackDepth,  
                        void *pvParameters,  
                        UBaseType_t uxPriority,  
                        TaskHandle_t *pxCreatedTask );
```

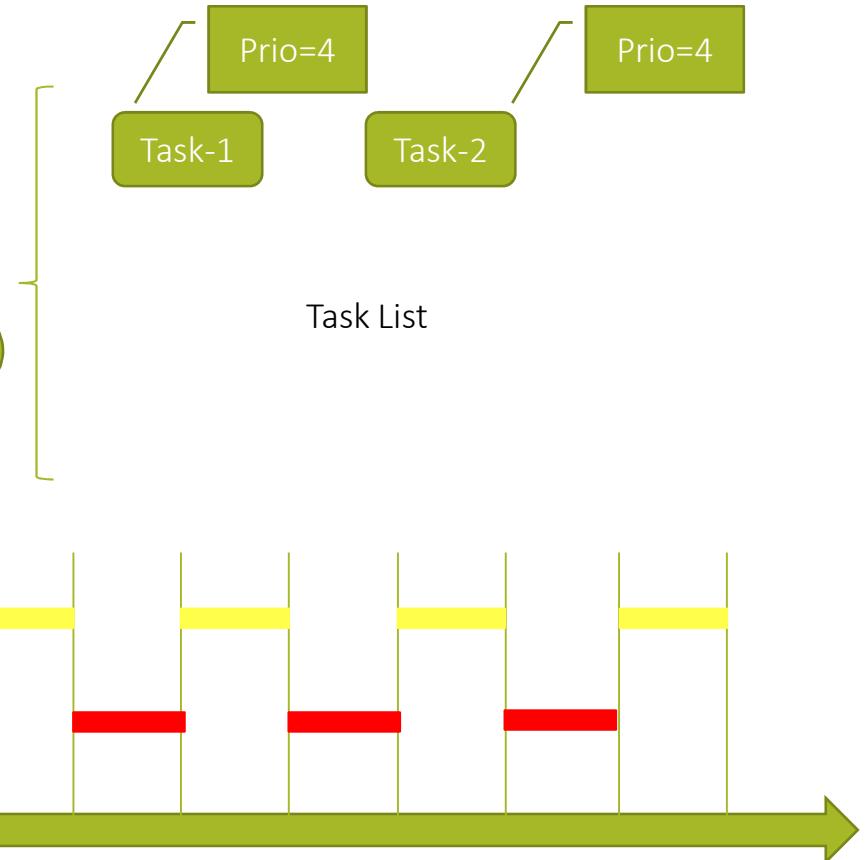
API to Schedule a Task





A cartoon character of a bald man wearing sunglasses and a black t-shirt, with his arms crossed, looking thoughtful. A thought bubble originates from his head.

Oops !! My Secluding policy is priority preemptive and both the task have equal priority . If I don't schedule like this I will be a fool !

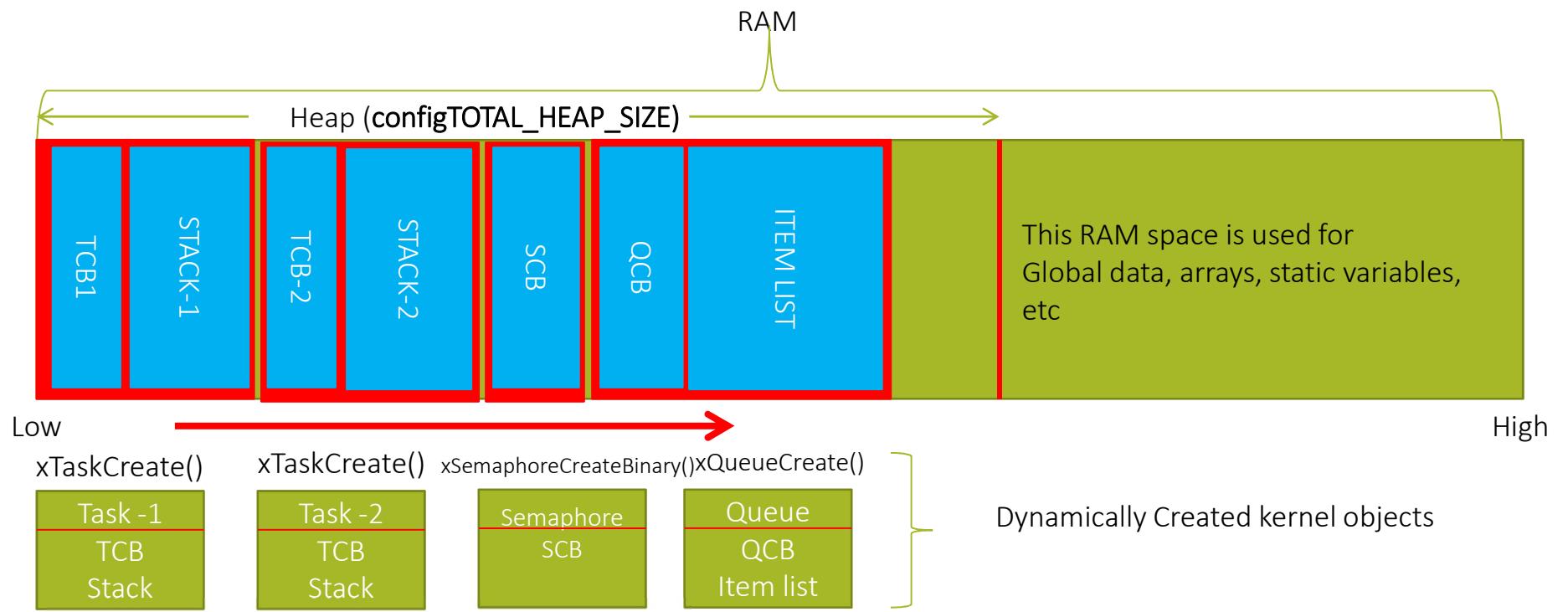


Scheduler
COPYRIGHT © BHARATI SOFTWARE 2016.

FreeRTOS behind the scene TASK management

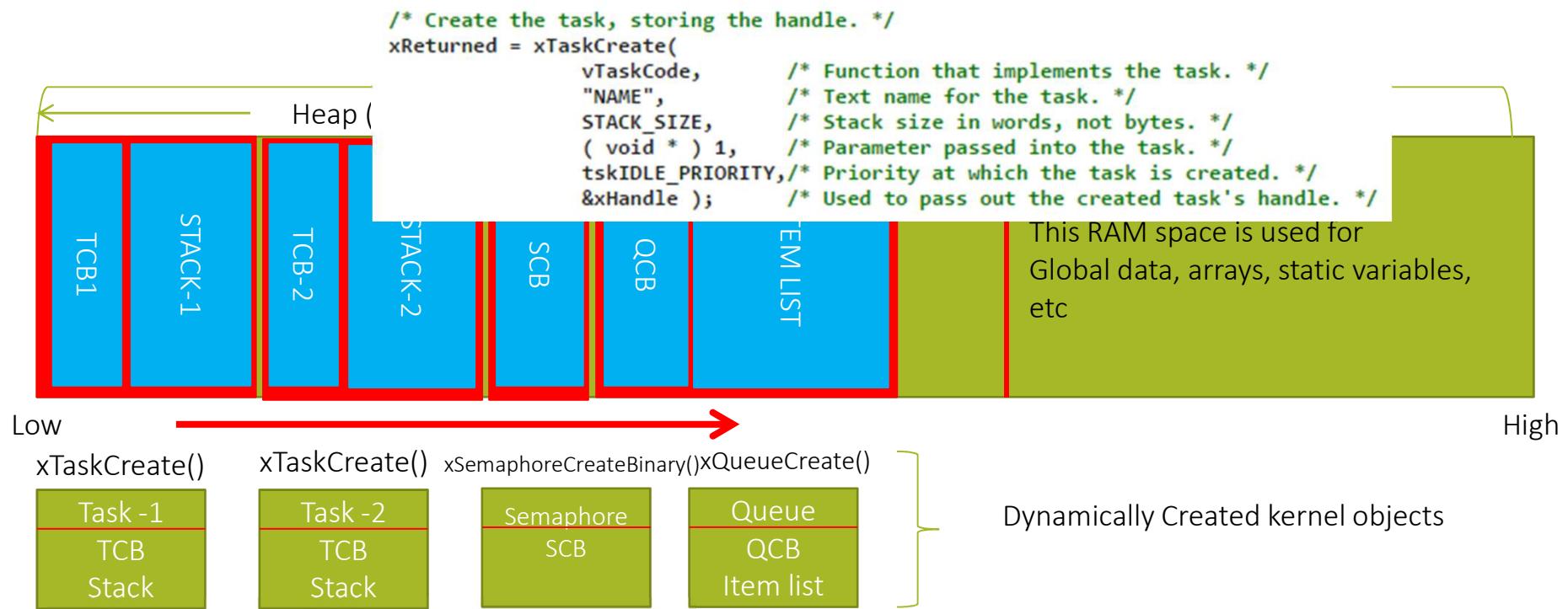
COPYRIGHT © BHARATI SOFTWARE 2016.

What happens when you create a TASK?



COPYRIGHT © BHARATI SOFTWARE 2016.

What happens when you create a TASK?



COPYRIGHT © BHARATI SOFTWARE 2016.

FreeRTOS Hello World Application and Testing on hardware

COPYRIGHT © BHARATI SOFTWARE 2016.

Exercise

Write a program to create 2 tasks Task-1 and Task-2 with same priorities .

When Task-1 executes it should print “*Hello World from Task-1*”

And when Task-2 executes it should print “*Hello World from Task-2*”

Case 1 : Use ARM Semi-hosting feature to print logs on the console

Case 2 : Use UART peripheral of the MCU to print logs

MCU Clock Configuration

COPYRIGHT © BHARATI SOFTWARE 2016.



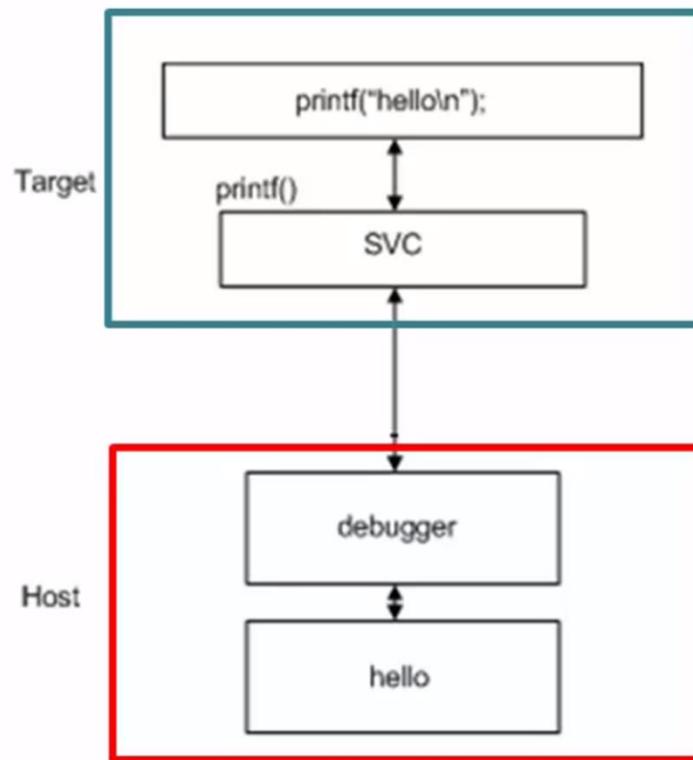
Ok, Now you created the FreeRTOS Project ,
but at what clock frequency your main
system clock of the MCU is running ?

You should know this in any RTOS project .

Semi hosting and UART setup

COPYRIGHT © BHARATI SOFTWARE 2016.

Semihosting Overview



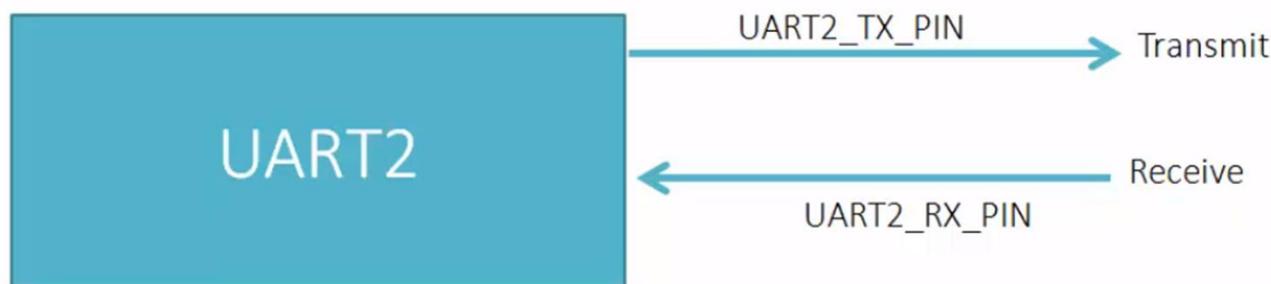
Target code wants to print "Hello"

Minimal library code which converts ***printf*** in to
"CMD" + "ARGS" (example "hello" is argument
here) and sends to debugger

Debugger agent running on HOST catches that
command and argument

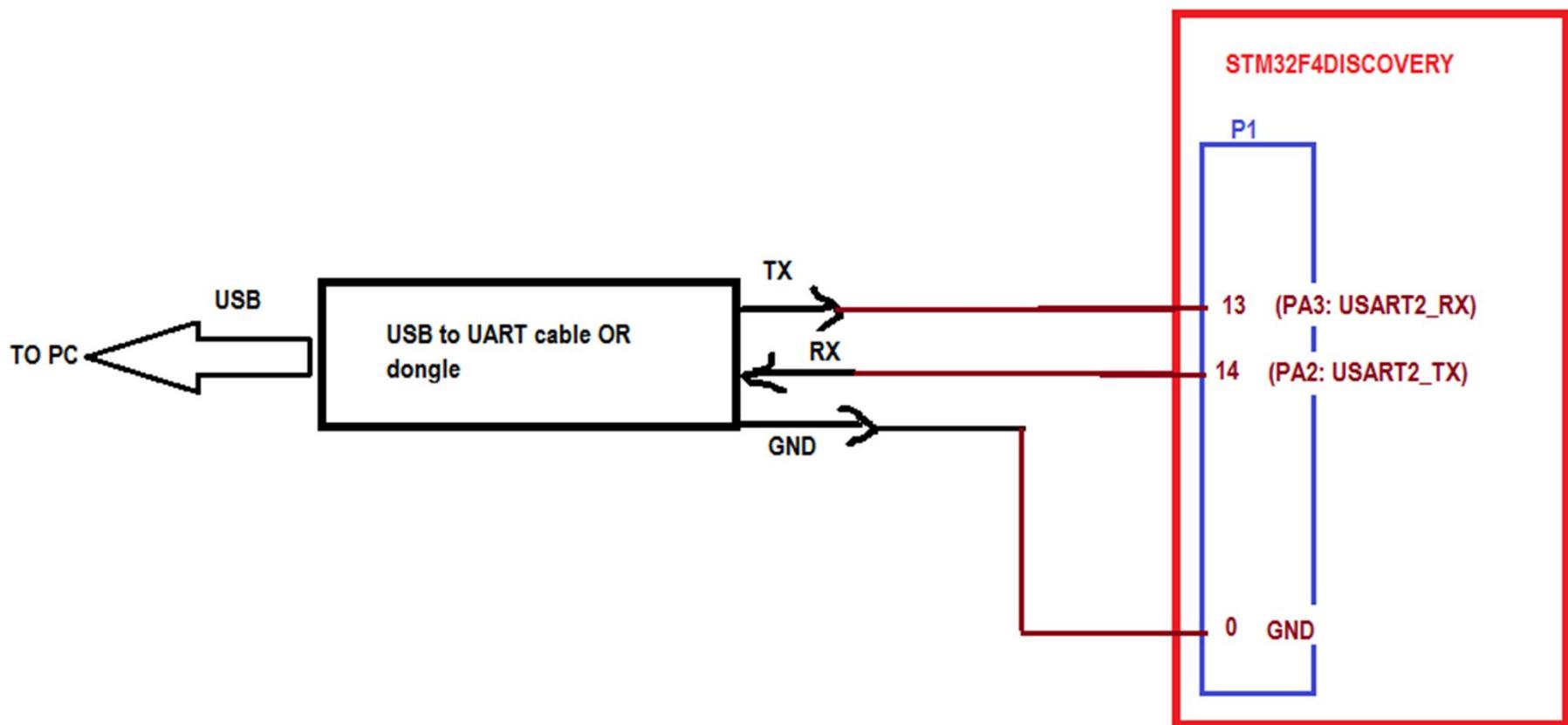
Prints "***hello***" on the HOST debugger's screen.

UART Communication without hardware flow control



This is UART peripheral of the MCU

You need just 2 pins of the MCU to establish full duplex data communications using UART peripheral



COPYRIGHT © BHARATI SOFTWARE 2016.

FreeRTOS app debugging using SEGGER SystemView Tools

Downloading SEGGER SystemView Software

SEGGER SystemView Installation and first look

What is SEGGER SystemView ?

SystemView is a software toolkit which is used to analyze the embedded software behaviour running on your target.

The embedded software may contain embedded OS or RTOS or it could be non-OS based application.

What is SEGGER SystemView ?

The systemView can be used to analyze how your embedded code is behaving on the target .

Example : In the case of FreeRTOS application

- ✓ You can analyze how many tasks are running and how much duration they consume on the CPU
- ✓ ISR entry and exit timings and duration of run on the CPU.
- ✓ You can analyze other behaviour of tasks: like blocking, unblocking, notifying, yielding, etc.
- ✓ You can analyze CPU idle time so that you can think of sending CPU to speed mode
- ✓ Total runtime behaviour of the application

What is SEGGER SystemView ?

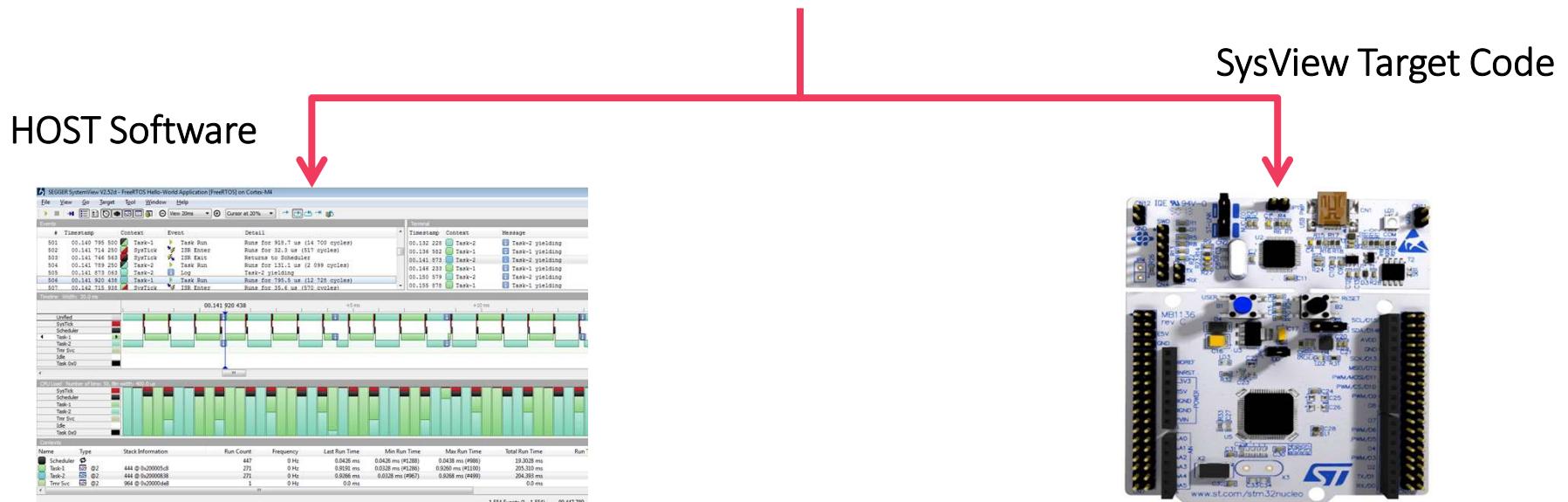
- ✓ It sheds light on what exactly happened in which order, which interrupt has triggered which task switch, which interrupt and task has called which API function of the underlying RTOS
- ✓ SystemView should be used to verify that the embedded system behaves as expected and can be used to find problems and inefficiencies, such as superfluous and spurious interrupts, and unexpected task changes

SEGGER SystemView Toolkit

SystemView toolkit come in 2 parts

- 1) PC visualization software : SystemView Host software
(Windows / Linux/mac)
- 2) SystemView target codes (this is used to collect the target events and sending back to PC visualization software)

SEGGER SystemView Toolkit



SystemView Visualization modes

1. Real time recording (Continuous recording) :

With a SEGGER J-Link and its *Real Time Transfer* (RTT) technology SystemView can continuously record data, and analyze and visualize it in real time.

Real time mode can be achieved via ST-link instead of J-link . For that J-link firmware has to be flashed on ST-link circuitry of STM32 boards. More on this later.

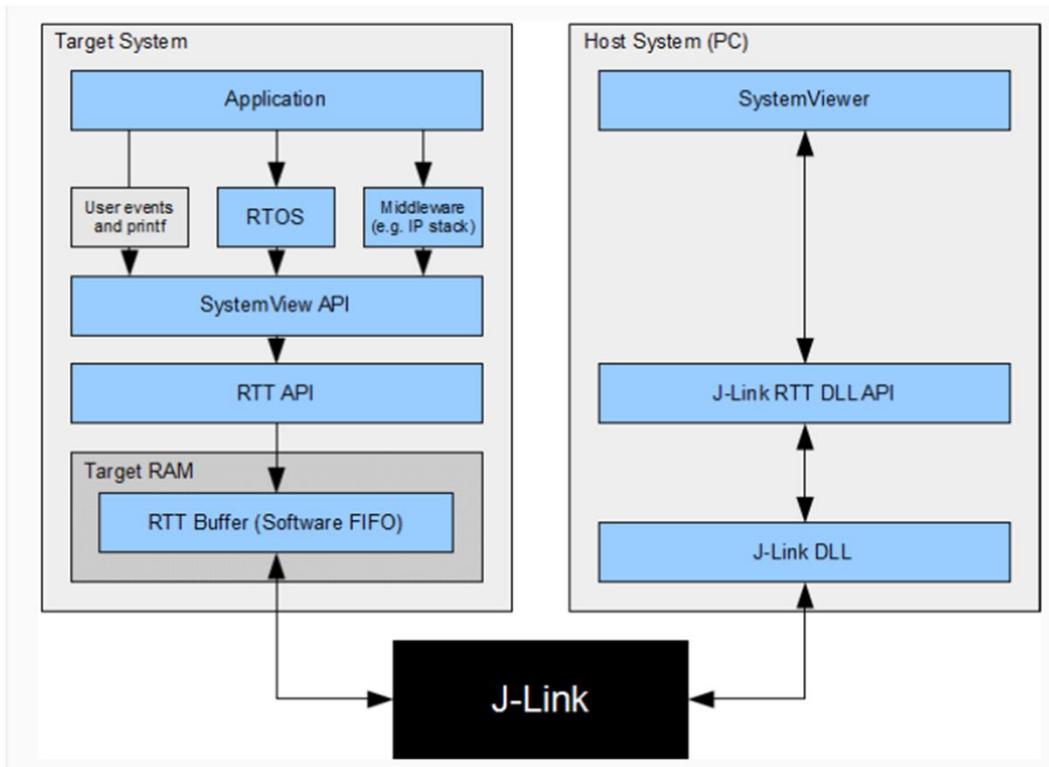
SystemView Visualization modes

2 Single-shot recording:

You need not to have JLINK or STLINK debugger for this.

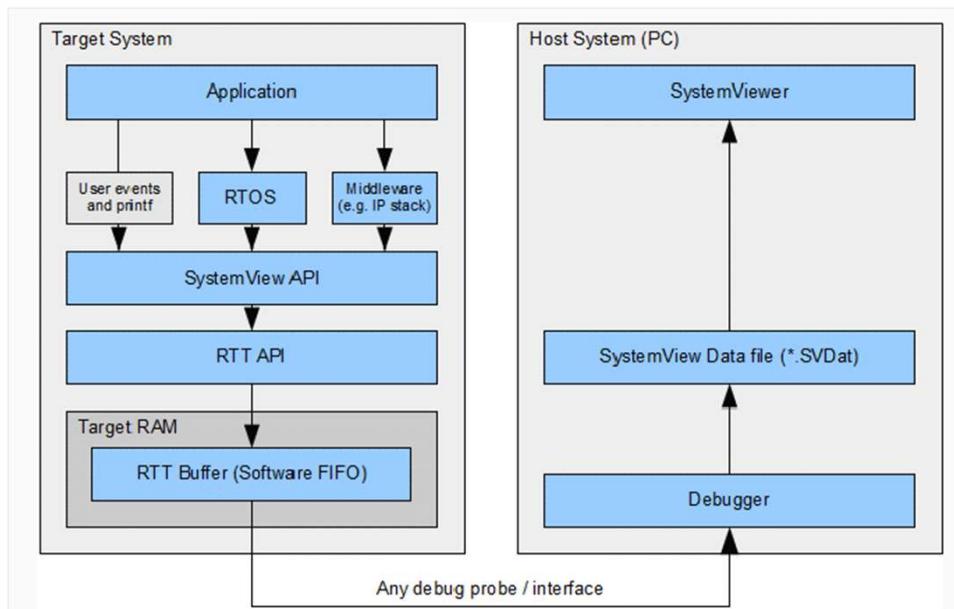
In single-shot mode the recording is started manually in the application, which allows recording only specific parts, which are of interest

Real time recording



COPYRIGHT © BHARATI SOFTWARE 2016.

Single-shot recording

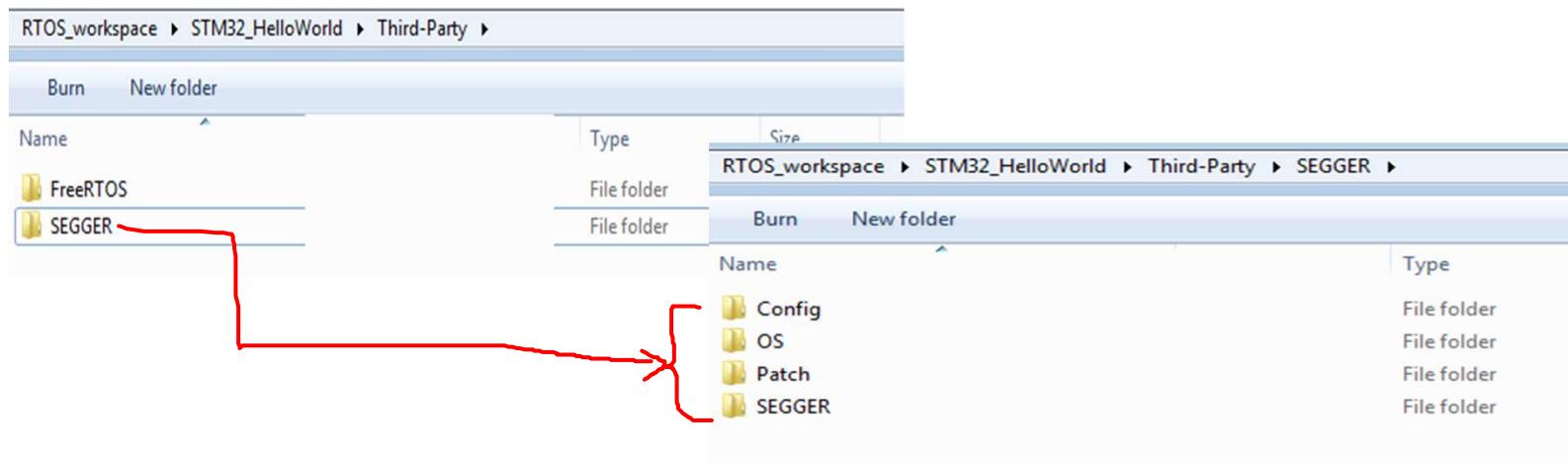


when no J-Link is used, SEGGER SystemView can be used to record data until its target buffer is filled. In single-shot mode the recording is started manually in the application, which allows recording only specific parts, which are of interest.

SEGGER SystemView Target Integration

STEP 1 : Including SEGGER SystemView in the application

- ✓ Download Systemview target sources and extract it
- ✓ Create the folders as below in your FreeRTOS Project.
- ✓ Do the path settings for the include files of SEGGER in Eclipse



COPYRIGHT © BHARATI SOFTWARE 2016.

RTOS_workspace ▶ STM32_HelloWorld ▶ Third-Party ▶ SEGGER ▶ Config		
Burn	New folder	
Name	Type	Size
Global	H File	6 KB
SEGGER_RTT_Conf	H File	20 KB
SEGGER_SYSVIEW_Conf	H File	10 KB
SEGGER_SYSVIEW_Config_FreeRTOS	C File	6 KB

RTOS_workspace ▶ STM32_HelloWorld ▶ Third-Party ▶ SEGGER ▶ OS		
Burn	New folder	
Name	Type	Size
SEGGER_SYSVIEW_FreeRTOS	C File	11 KB
SEGGER_SYSVIEW_FreeRTOS	H File	26 KB

RTOS_workspace ▶ STM32_HelloWorld ▶ Third-Party ▶ SEGGER ▶ Patch ▶

Burn New folder

Name	Date modified	Type
FreeRTOSv10.1.1		File folder

RTOS_workspace ▶ STM32_HelloWorld ▶ Third-Party ▶ SEGGER ▶ Patch ▶ FreeRTOSv10.1.1

Burn New folder

Name	Date modified	Type	Size
FreeRTOSV10_Core.patch		PATCH File	13 KB

RTOS_workspace ▶ STM32_HelloWorld ▶ Third-Party ▶ SEGGER ▶ SEGGER

Burn New folder

Name	Date modified	Type	Size
SEGGER		H File	11 KB
SEGGER_RTT		C File	54 KB
SEGGER_RTT		H File	14 KB
SEGGER_SYSVIEW		C File	93 KB
SEGGER_SYSVIEW		H File	18 KB
SEGGER_SYSVIEW_ConfDefaults		H File	8 KB
SEGGER_SYSVIEW_Int		H File	6 KB

STEP2 : Patching FreeRTOS files

You need to patch some of the FreeRTOS files with patch file given by SEGGER systemView

STEP3: FreeRTOSConfig.h Settings

1. The SEGGER_SYSVIEW_FreeRTOS.h header has to be included at the end of FreeRTOSConfig.h or above every include of FreeRTOS.h. It defines the trace macros to create SYSTEMVIEW events.
2. In freeRTOSConfig.h include the below macros

```
#define INCLUDE_xTaskGetIdleTaskHandle 1  
#define INCLUDE_pxTaskGetStackStart 1
```

STEP4: MCU and Project specific settings

1. Mention which processor core your MCU is using in
SEGGER_SYSVIEW_Conf.h
2. Do SystemView buffer size configuration in SEGGER_SYSVIEW_Conf.h
(SEGGER_SYSVIEW_RTT_BUFFER_SIZE) .
3. Configure the some of the application specific information in
SEGGER_SYSVIEW_Config_FreeRTOS.c

STEP5: Enable the ARM Cortex Mx Cycle Counter

This is required to maintain the time stamp information of application events. SystemView will use the Cycle counter register value to maintain the time stamp information of events.

DWT_CYCCNT register of ARM Cortex Mx processor stores number of clock cycles that have been happened after the reset of the Processor.

By default this register is disabled.

STEP6: Start the recording of events

1. To start the recordings of your FreeTOS application, call the below SEGGER APIs .

SEGGER_SYSVIEW_Conf();

SEGGER_SYSVIEW_Start();

The segger systemview events recording starts only when you call SEGGER_SYSVIEW_Start(). We will call this from main.c

STEP7: Compile , Flash and Debug

1. Compile and flash your FreeRTOS + SystemView application
2. Go to debugging mode using your openSTM32 System Workbench.
3. Hit run and then pause after couple of seconds.

STEP8: Collect the recorded data(RTT buffer)

you can do this via continuous recording or Single-shot recording.

Single-shot recording :

1. Get the SystemView RTT buffer address and the number of bytes used.
(Normally _SEGGER_RTT.aUp[1].pBuffer and _SEGGER_RTT.aUp[1].WrOff).
2. Take the memory dump to a file
3. save the file with .SVDat extension
4. use that file to load in to SystemView HOST software to analyze the events.

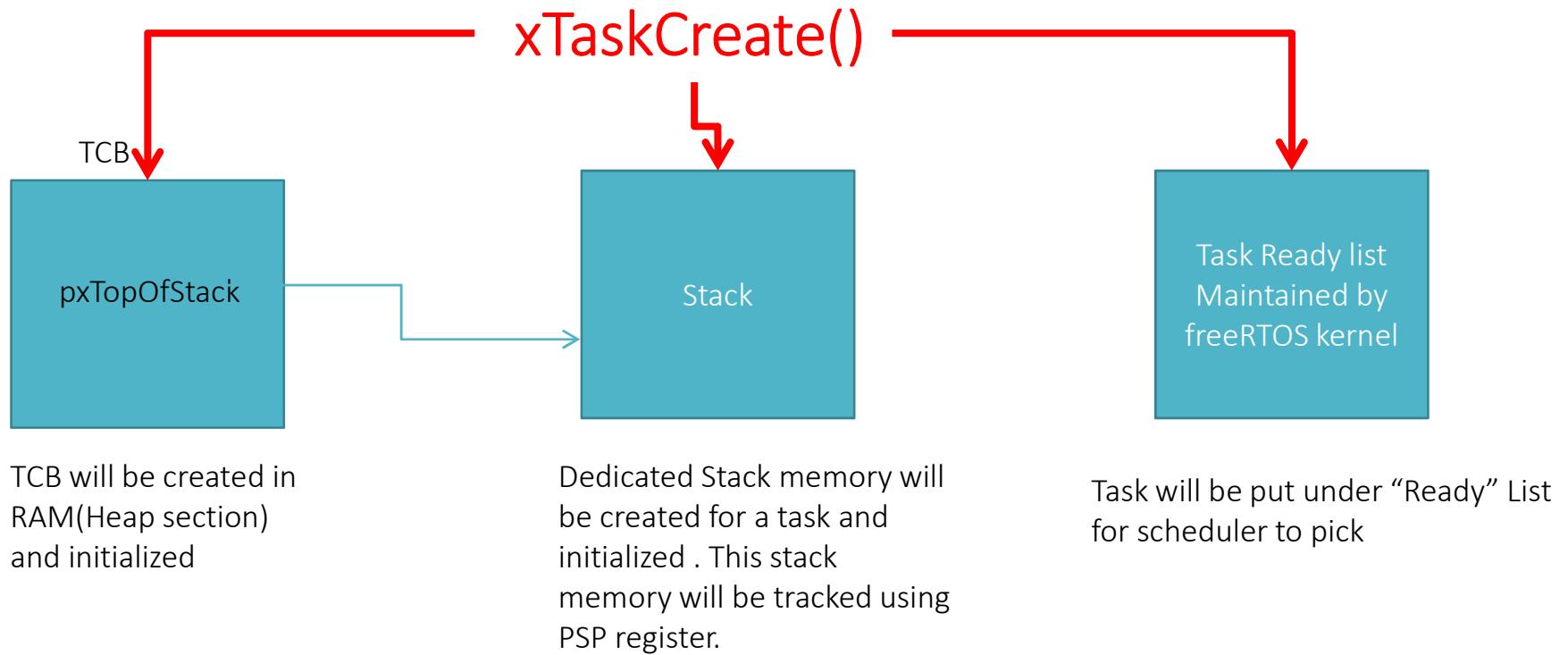
STEP8: Collect the recorded data(RTT buffer)

you can do this via continuous recording or Single-shot recording.

Continues recording on STM32 STLINK based boards

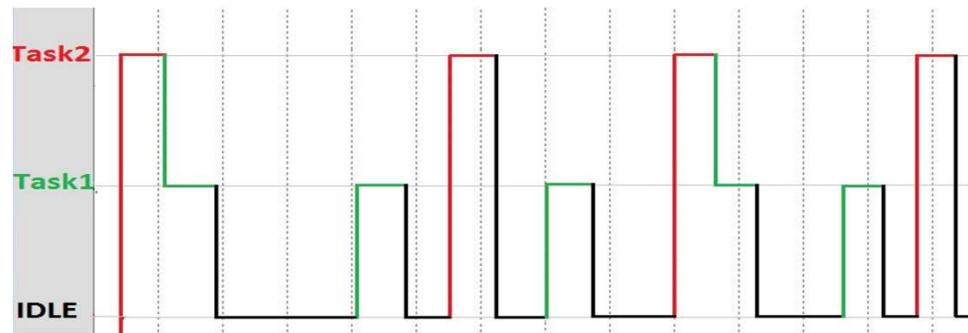
1. Flash the J-link firmware on your board. (this step basically removes the stlink frimware present in your board with Jlink firmware)
2. open the SystemView HOST software
3. Go to Target->Start recording
4. Mention “Target Device” and RTT address details and click “OK
5. Software will record the events and display.

Task Creation



About FreeRTOS idle task and its significance

Idle Task



The Idle task is created automatically when the RTOS scheduler is started to ensure there is always at least one task that is able to run.

It is created at the lowest possible priority to ensure it does not use any CPU time if there are higher priority application tasks in the ready state.

Some facts about Idle Task

It is a lowest priority task which is automatically created when the scheduler is started

The idle task is responsible for freeing memory allocated by the RTOS to tasks that have been deleted

When there are no tasks running, Idle task will always run on the CPU.

You can give an application hook function in the idle task to send the CPU to low power mode when there are no useful tasks are executing.

FreeRTOS Timer Services Task

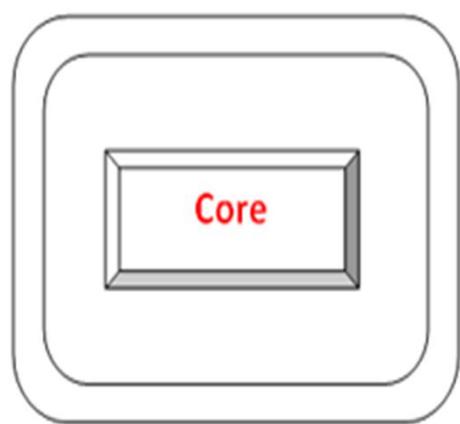
COPYRIGHT © BHARATI SOFTWARE 2016.

Timer Services Task (Timer_svc)

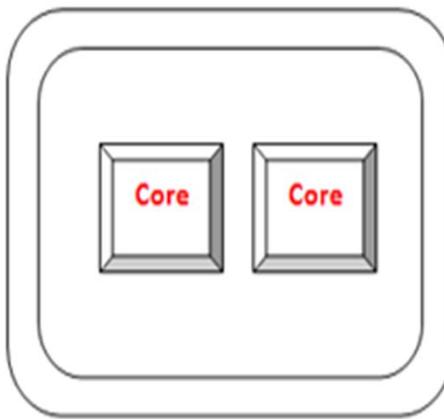
- ✓ This is also called as timer daemon task
- ✓ The timer daemon task deals with “Software timers”
- ✓ This task is created automatically when the scheduler is started and if `configUSE_TIMERS = 1` in FreeRTOSConfig.h
- ✓ The RTOS uses this daemon to manage FreeRTOS software timers and nothing else.
- ✓ If you don't use software timers in your FreeRTOS application then you need to use this Timer daemon task. For that just make `configUSE_TIMERS = 0` in FreeRTOSConfig.h
- ✓ All software timer callback functions execute in the context of the timer daemon task

Overview of FreeRTOS Scheduler

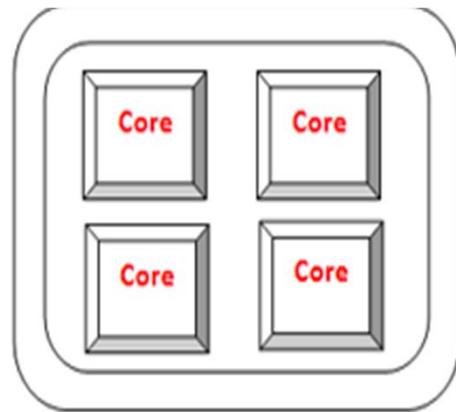
COPYRIGHT © BHARATI SOFTWARE 2016.



A processor



A processor with 2 cores



A processor with 4 cores

Why do we need Scheduler ?

1. It just a piece of code which implements task switching in and Task switching out according to the scheduling policy selected.
2. Scheduler is the reason why multiple tasks run on your system efficiently
3. The basic job of the scheduler is to determine which is the next potential task to run on the CPU
4. Scheduler has the ability to preempt a running task if you configure so

Scheduler



I have my own **scheduling policy** , I act according that .

You can configure **my scheduling policy**

Scheduling Policies (Scheduler types)

1. simple Pre-emptive Scheduling (Round robin)
2. Priority based Pre-Emptive Scheduling
3. Co-operative Scheduling

The **scheduling policy** is the algorithm used by the scheduler to decide which task to execute at any point in time.

FreeRTOS or most of the Real Time OS most likely would be **using Priority based Pre-emptive Scheduling** by default

FreeRTOS Scheduler and xTaskStartScheduler() API

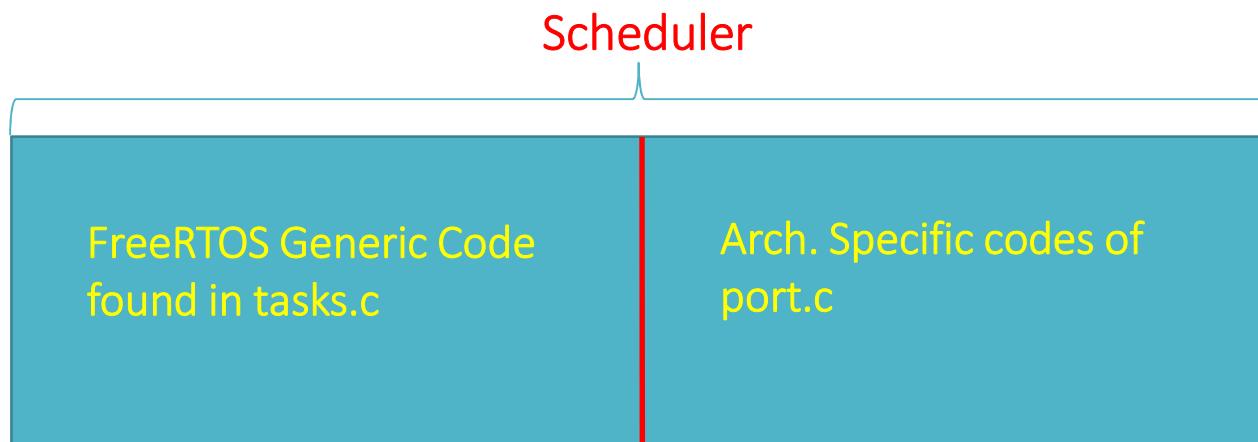
COPYRIGHT © BHARATI SOFTWARE 2016.

FreeRTOS Scheduling

- ✓ Scheduler is a piece of kernel code responsible for deciding which task should be executing at any particular time on the CPU.
- ✓ The **scheduling policy** is the algorithm used by the scheduler to decide which task to execute at any point in time.
- ✓ **configUSE_PREEMPTION** of freeRTOSConfig.h configurable item decides the scheduling policy in freeRTOS.
- ✓ If **configUSE_PREEMPTION** =1, then scheduling policy will be priority based pre-emptive scheduling .
- ✓ If **configUSE_PREEMPTION** =0, then scheduling policy will be cooperative scheduling

FreeRTOS Scheduler Implementation

In FreeRTOS the scheduler code is actually combination of FreeRTOS Generic code + Architecture specific codes



Architecture specific codes responsible to achieve scheduling of tasks.

All architecture specific codes and configurations are implemented in **port.c** and **portmacro.h**

If you are using ARM Cortex Mx processor then you should be able locate the below interrupt handlers in **port.c** which are part of the scheduler implementation of freeRTOS

Three important kernel interrupt handlers responsible for scheduling of tasks

vPortSVCHandler()

Used to launch the very first task.
Triggered by SVC instruction

xPortPendSVHandler()

Used to achieve the context switching between tasks
Triggered by pending the PendSV System exception of ARM

xPortSysTickHandler()

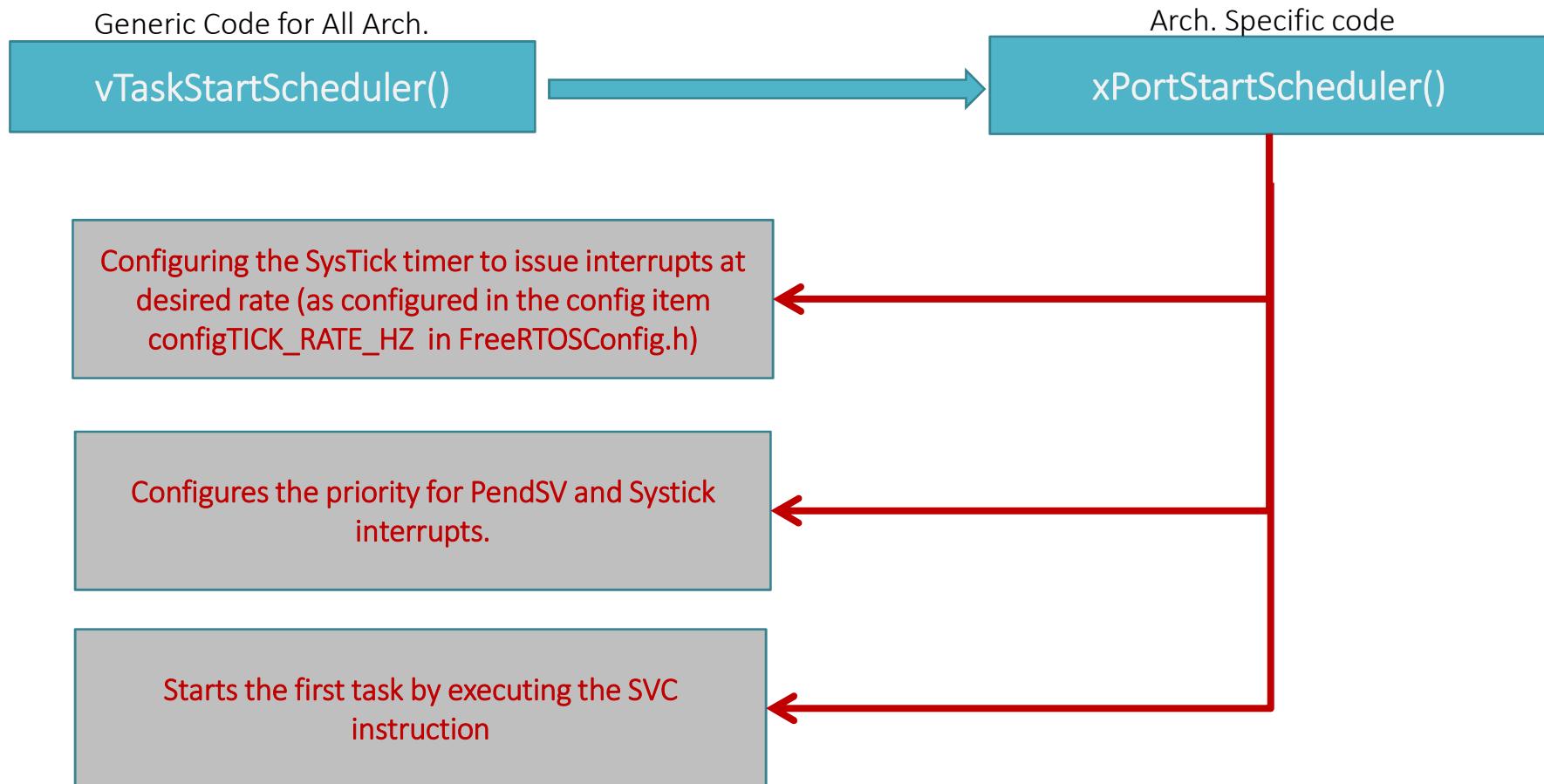
This implements the RTOS Tick management.
Triggered periodically by Systick timer of ARM cortex Mx processor

vTaskStartScheduler()

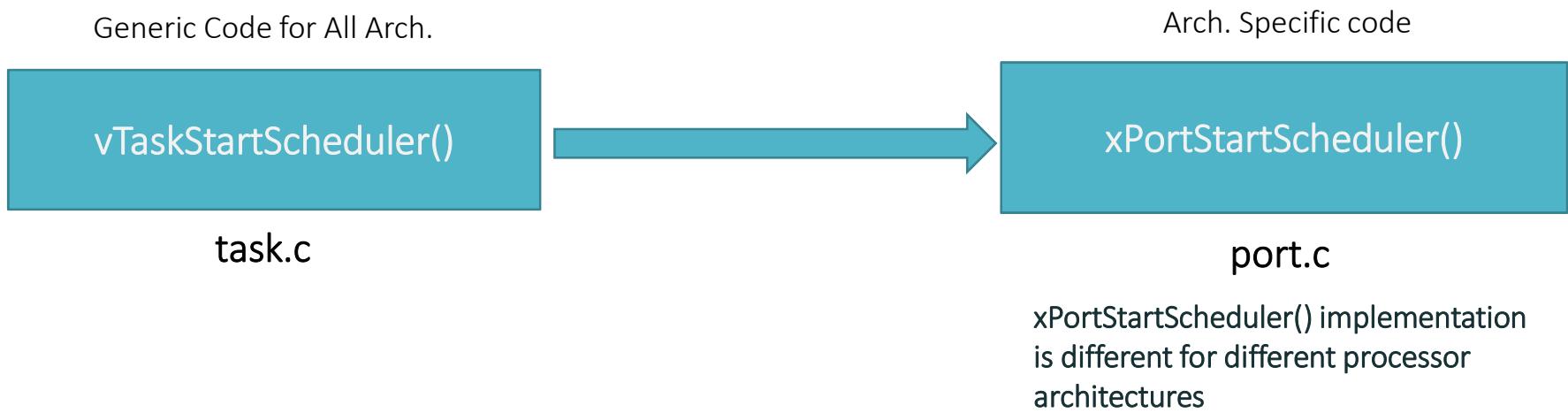
- ✓ This is implemented in tasks.c of FreeRTOS kernel and used to start the RTOS scheduler .
- ✓ Remember that after calling this function only the scheduler code is initialized and all the Arch. Specific interrupts will be activated.
- ✓ This function also creates the idle and Timer daemon task
- ✓ This function calls `xPortStartScheduler()` to do the Arch. Specific Initializations such as
 1. *Configuring the SysTick timer to issue interrupts at desired rate (as configured in the config item configTICK_RATE_HZ in FreeRTOSConfig.h)*
 2. *Configures the priority for PendSV and Systick interrupts.*
 3. *Starts the first task by executing the SVC instruction*
- ✓ So basically this function triggers the scheduler(i.e various Arch specific interrupts aka kernel interrupts) and never returns.

vTaskStartScheduler()

- ✓ This is implemented in tasks.c of FreeRTOS kernel and used to start the RTOS scheduler .
- ✓ Remember that after calling this function only the scheduler code is initialized and all the Arch. Specific interrupts will be activated.
- ✓ This function also creates the idle and Timer daemon task
- ✓ This function calls xPortStartScheduler() to do the Arch. Specific Initializations



vTaskStartScheduler()



FreeRTOS Kernel Interrupts and Scheduling of Tasks

COPYRIGHT © BHARATI SOFTWARE 2016.

FreeRTOS Kernel interrupts

When FreeRTOS runs on ARM Cortex Mx Processor based MCU, below interrupts are used to implement the Scheduling of Tasks.

1. **SVC Interrupt** (SVC handler will be used to launch the very first Task)
2. **PendSV Interrupt** (PendSV handler is used to carry out context switching between tasks)
3. **SysTick Interrupt** (SysTick Handler implements the RTOS Tick Management)

If SysTick interrupt is used for some other purpose in your application, then you may use any other available timer peripheral

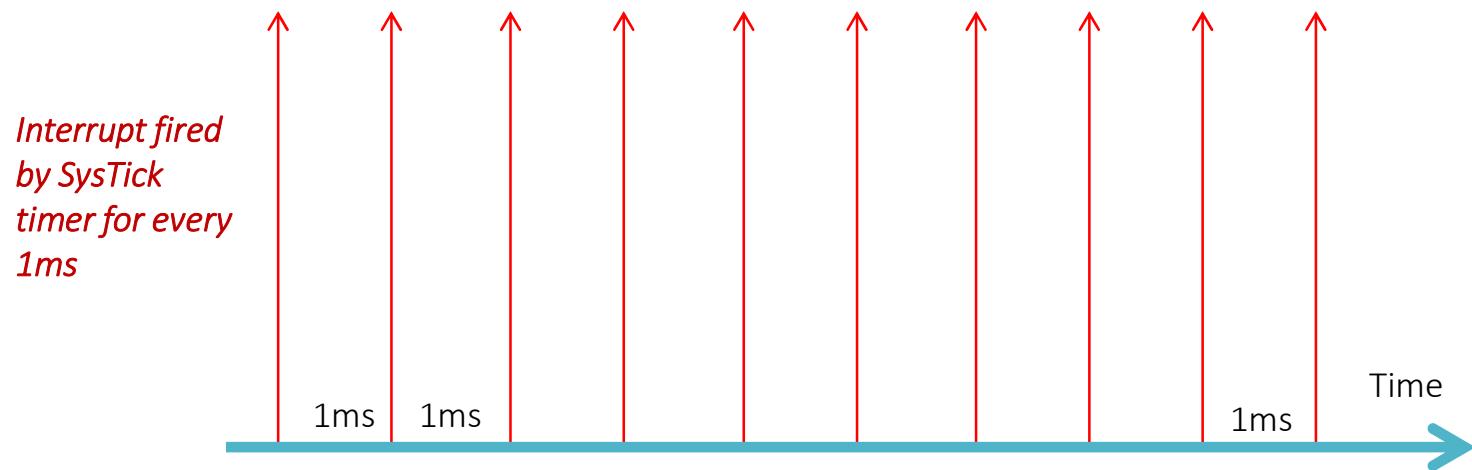
All interrupts are configured at the lowest interrupt priority possible.

RTOS Tick (The heart beat)

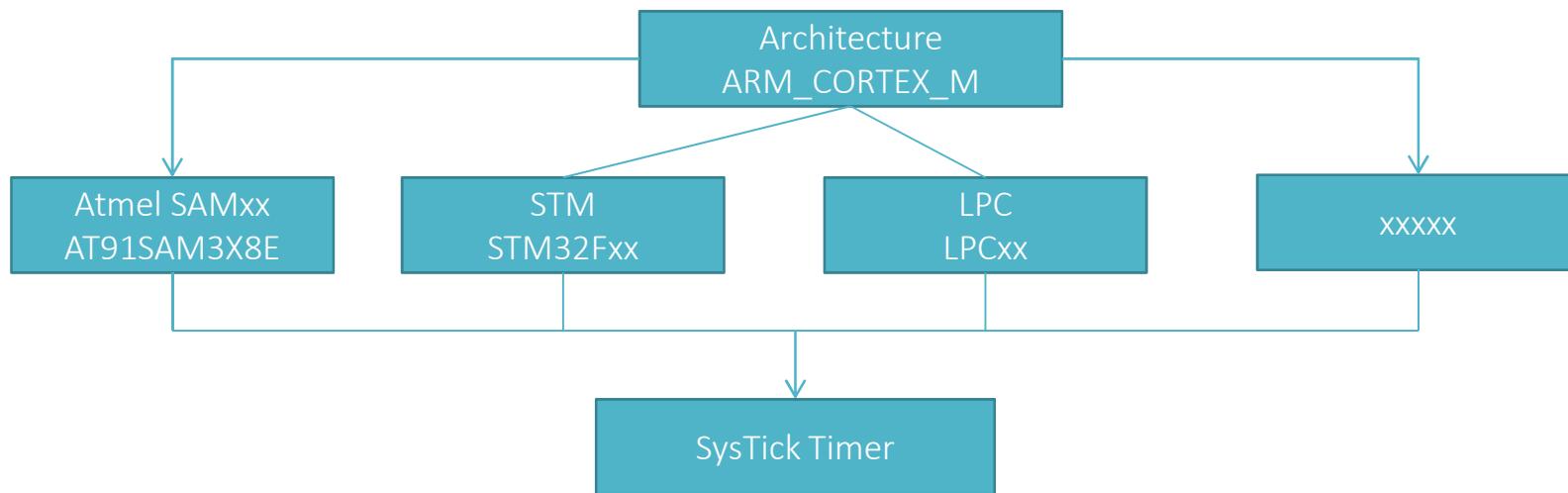


The RTOS Tick

```
#define configTICK_RATE_HZ      ( (portTickType)1000)
```



RTOS Ticking is implemented using timer hardware of the MCU



The RTOS Tick- Why it is needed ?

- ✓ The simple answer is to keep track of time elapsed
- ✓ There is a global variable called “**xTickCount**”, and it is incremented by one whenever tick interrupt occurs
- ✓ RTOS Ticking is implemented using SysTick timer of the ARM Cortex Mx processor.
- ✓ Tick interrupt happens at the rate of `configTICK_RATE_HZ` configured in the `FreeRTOSConfig.h`

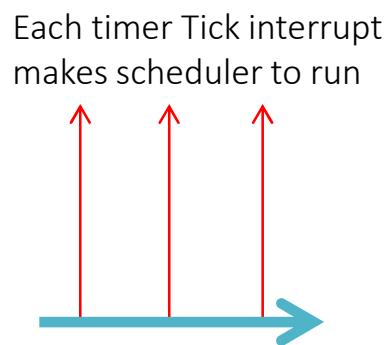
The RTOS Tick- Why it is needed ?



The RTOS Tick- Why it is needed ?

When RTOS Tick interrupt happens its interrupt handler will be called.
i.e **xPortSysTickHandler ()**

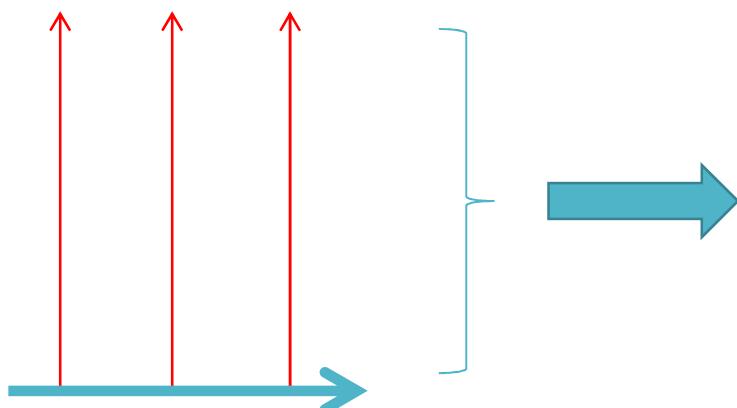
So, we can say that each tick interrupt causes scheduler to run.



The RTOS Tick- Why it is needed ?

Used for Context Switching to the next potential Task

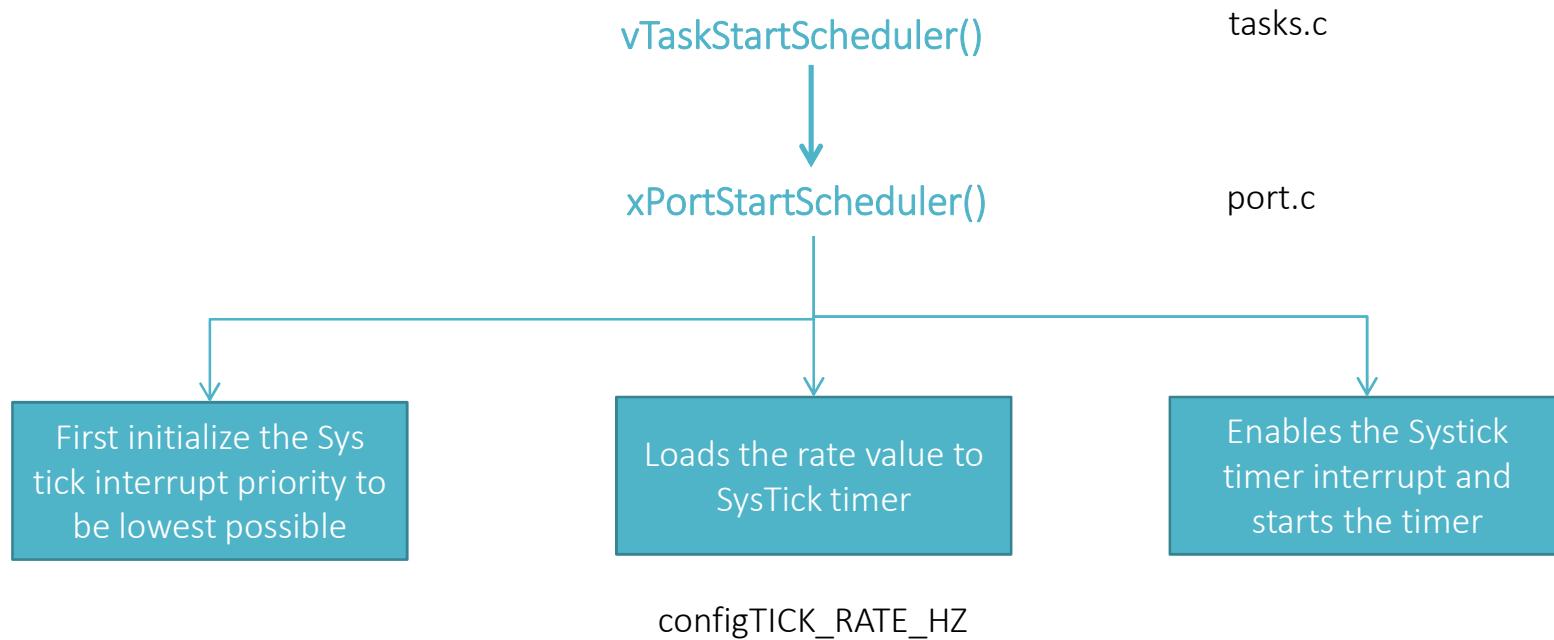
Each timer Tick interrupt makes scheduler to run



1. The tick ISR runs
2. All the ready state tasks are scanned
3. Determines which is the next potential task to run
4. If found, triggers the context switching by pending the PendSV interrupt
5. The PendSV handler takes care of switching out of old task and switching in of new task

Who configures RTOS tick timer (SysTick)?

Who configures RTOS tick Timer ?



The RTOS Tick Configuration

configSYSTICK_CLOCK_HZ = configCPU_CLK_HZ

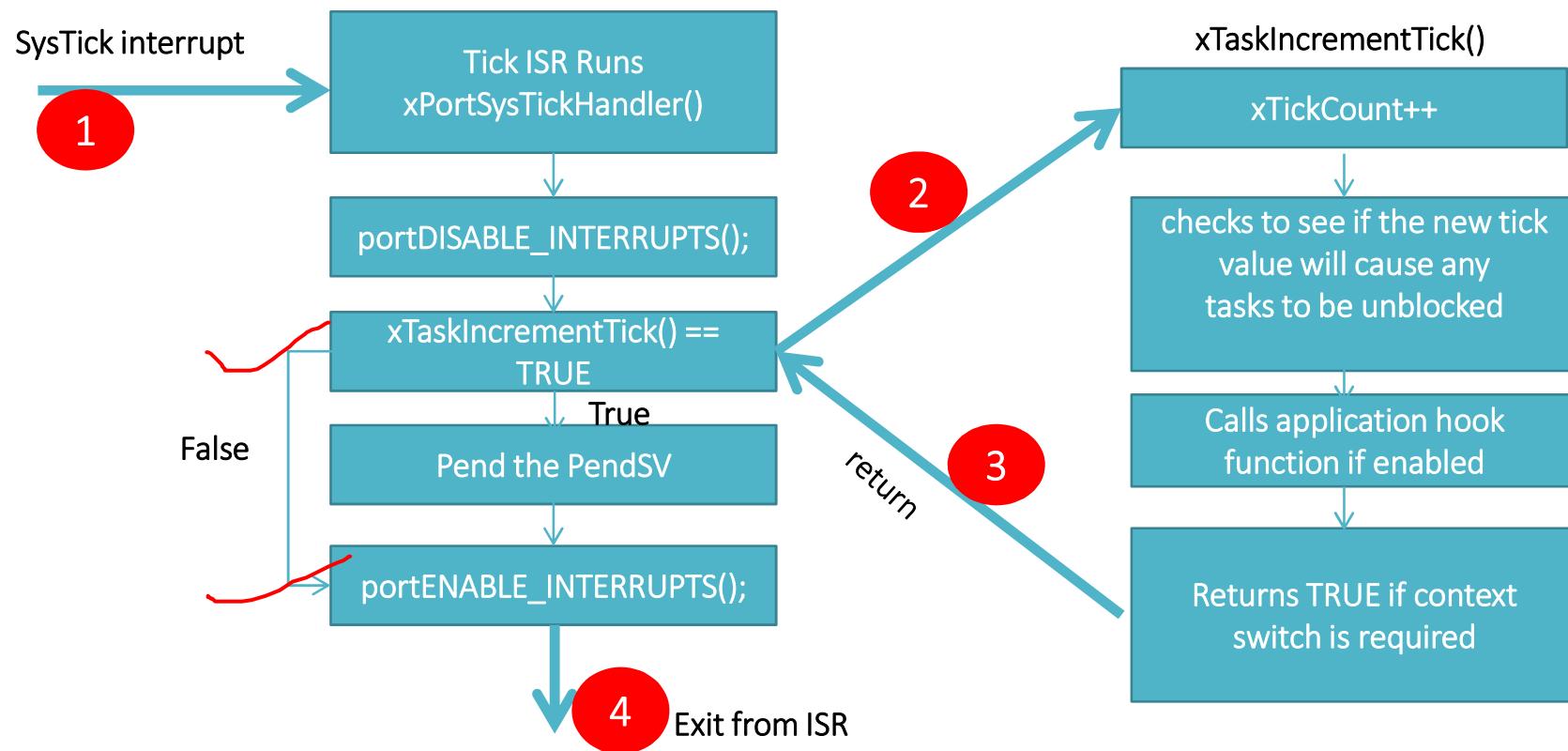
If configCPU_CLK_HZ = 16000000 (i.e 16Mhz)

And configTICK_RATE_HZ = 1000Hz.

Then portSYSTICK_NVIC_LOAD_REG = $(16000000/1000)-1 = 15999$

So, the SysTick timer should count from 0 to 15999 to generate an interrupt for every 1ms.

What RTOS tick ISR does ? : Summary



Context Switching

What is Context Switching ?

- ✓ Context switching is a process of switching out of one task and switching in of another task on the CPU to execute.
- ✓ In RTOS, Context Switching is taken care by the Scheduler .
- ✓ In FreeRTOS Context Switching is taken care by the PendSV Handler found in port.c

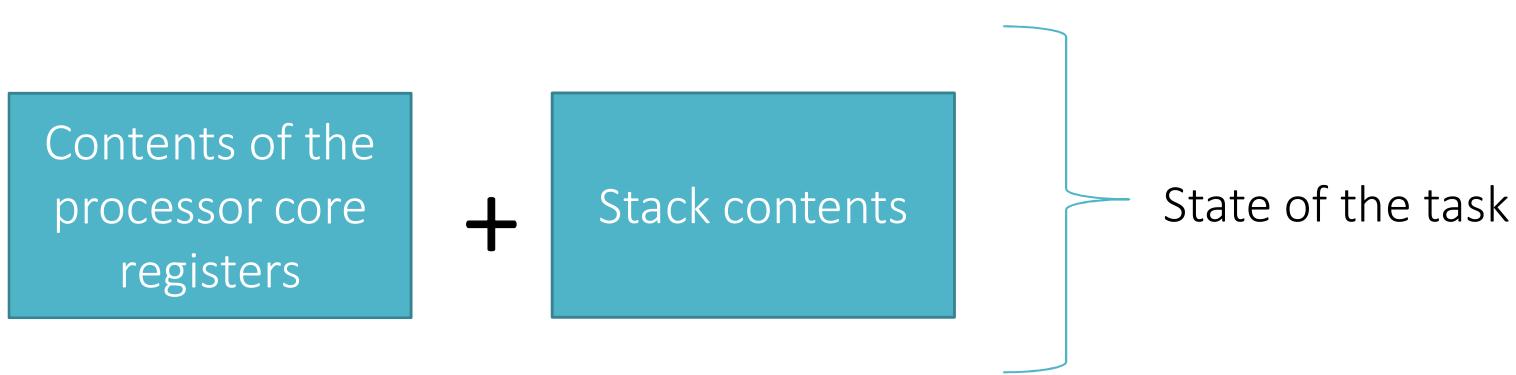
What is Context Switching ?

- ✓ If the scheduler is priority based pre-emptive scheduler , then for every RTOS tick interrupt, the scheduler will compare the priority of the running task with the priority of ready tasks list. If there is any ready task whose priority is higher than the running task then context switch will occur.
- ✓ On FreeRTOS you can also trigger context switch manually using **taskYIELD()** macro
- ✓ Context switch also happens immediately whenever new task unblocks and if its priority is higher than the currently running task.

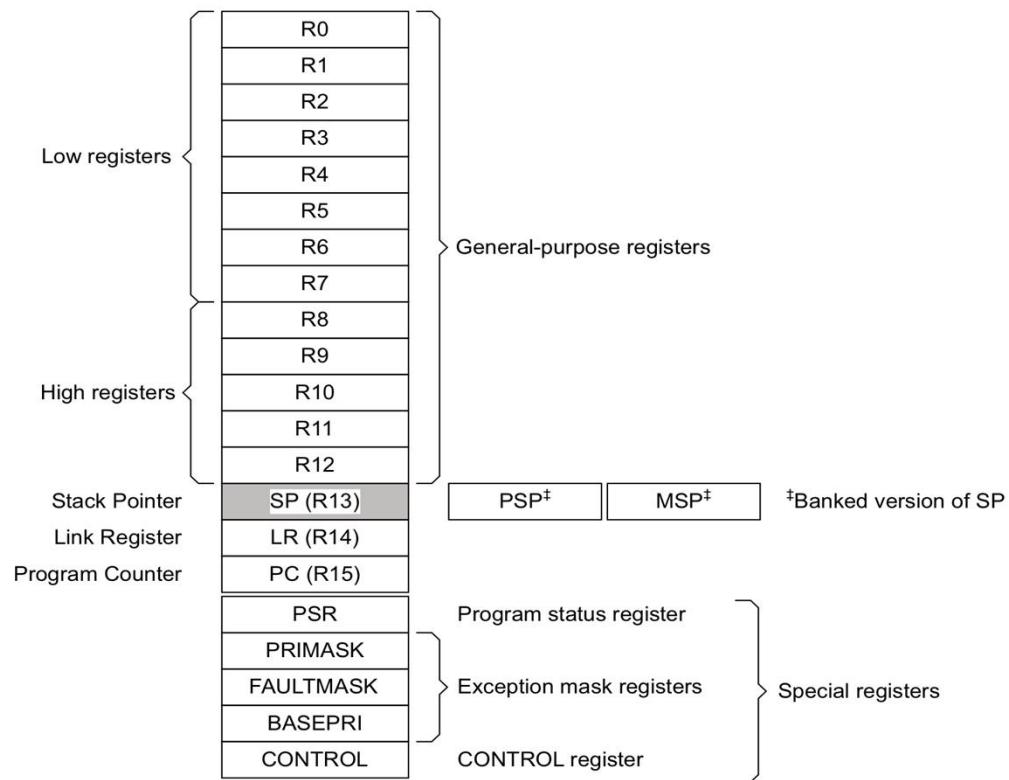
Task State

When a task executes on the Processor it utilizes

- ✓ Processor core registers.
- ✓ If a Task wants to do any push and pop operations (during function call) then it uses its own dedicated stack memory .

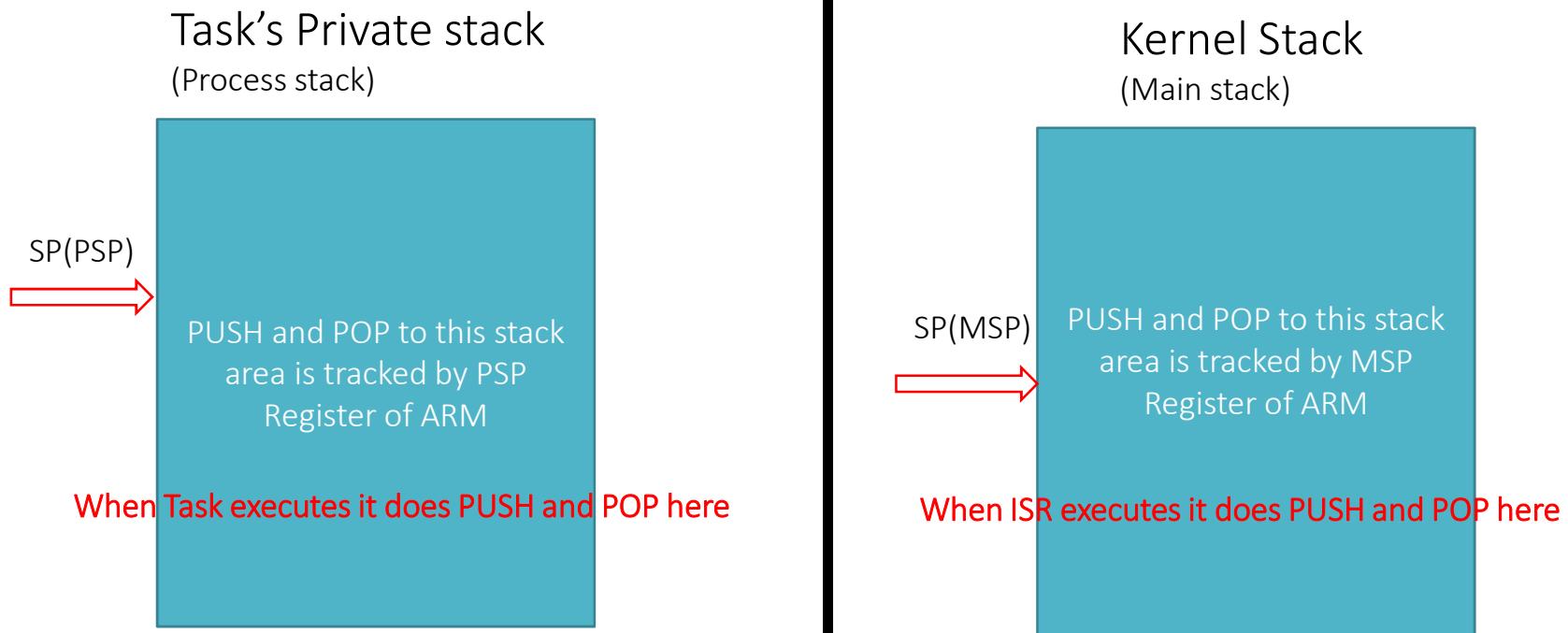


ARM Cortex Mx Core registers



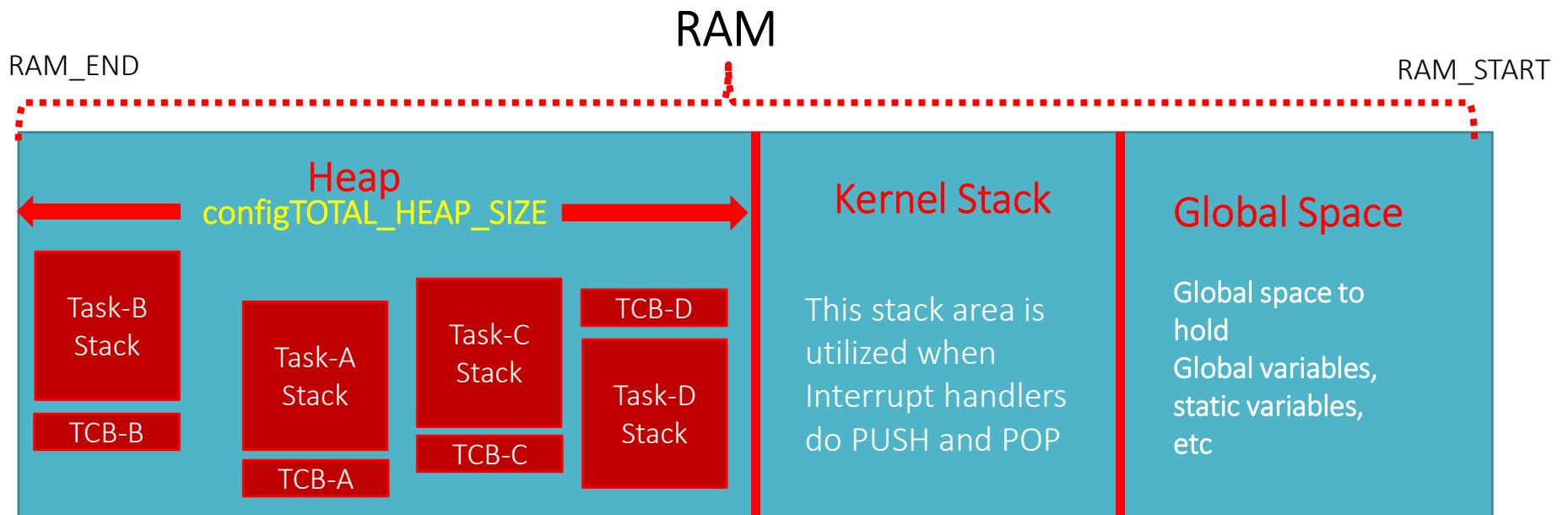
Stacks

There are mainly 2 different Stack Memories during run time of FreeRTOS based application

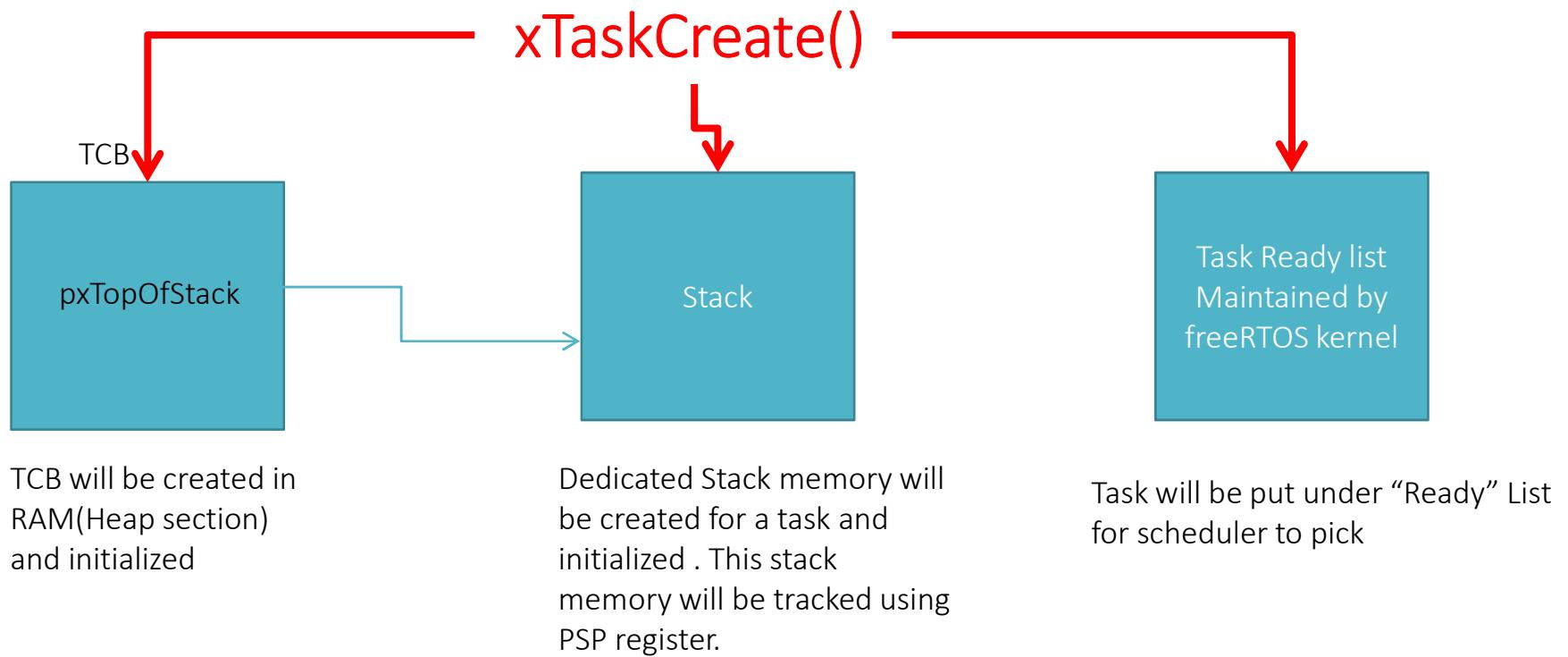


Stacks

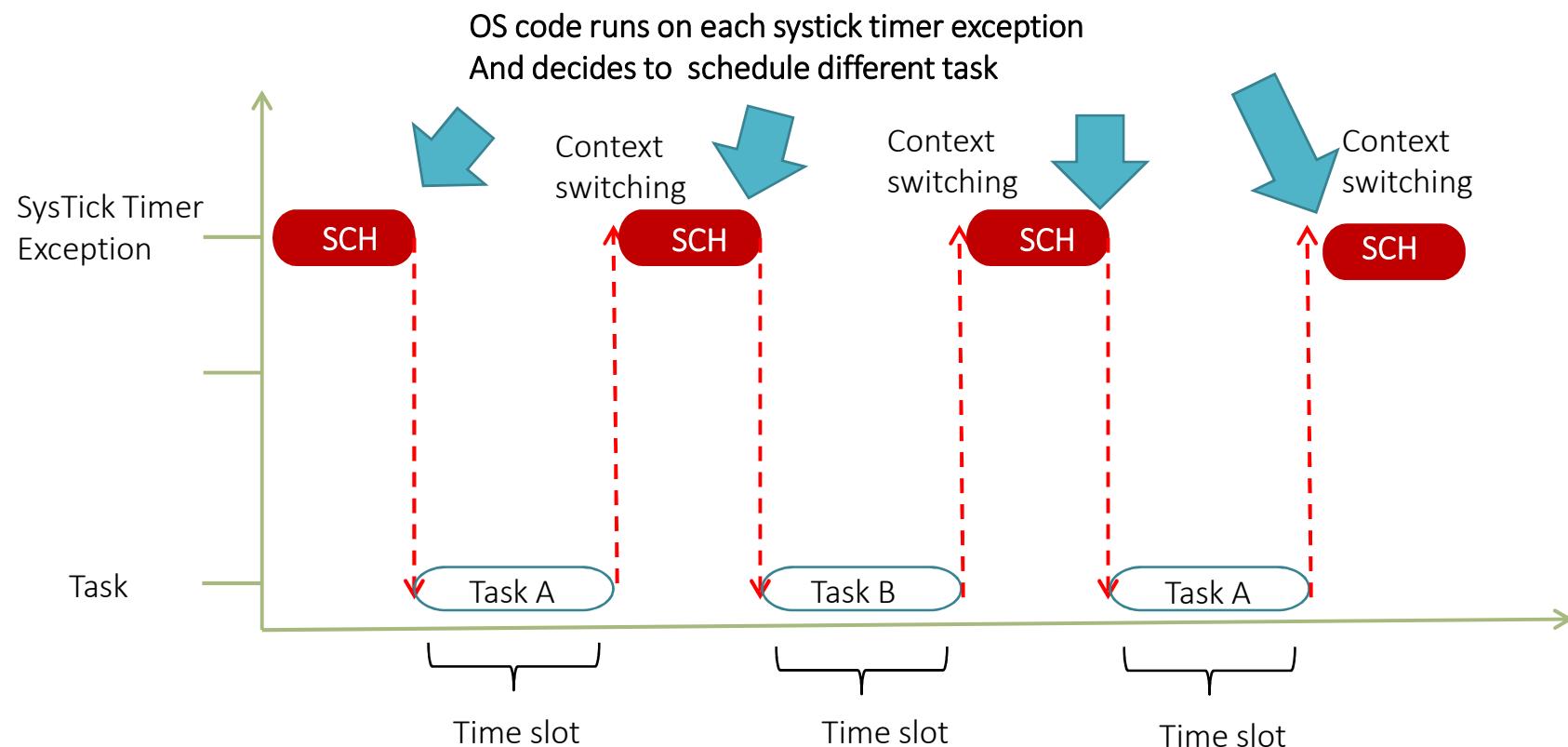
There are mainly 2 different Stack Memories during run time of FreeRTOS based application



Task Creation



Context Switching with animation



Task switching out & switching in procedure

Task switching out procedure

Before task is switched out , following things have to be taken care.

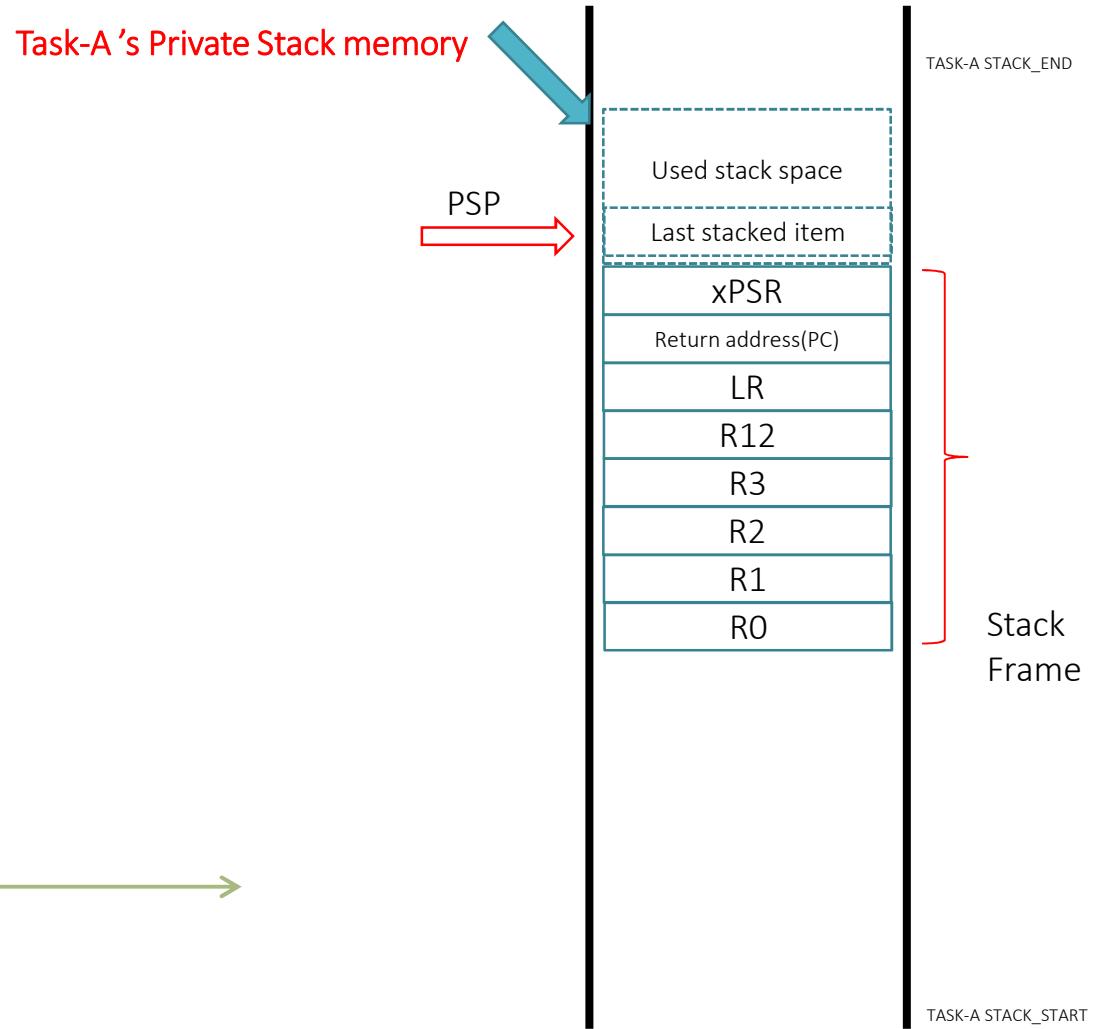
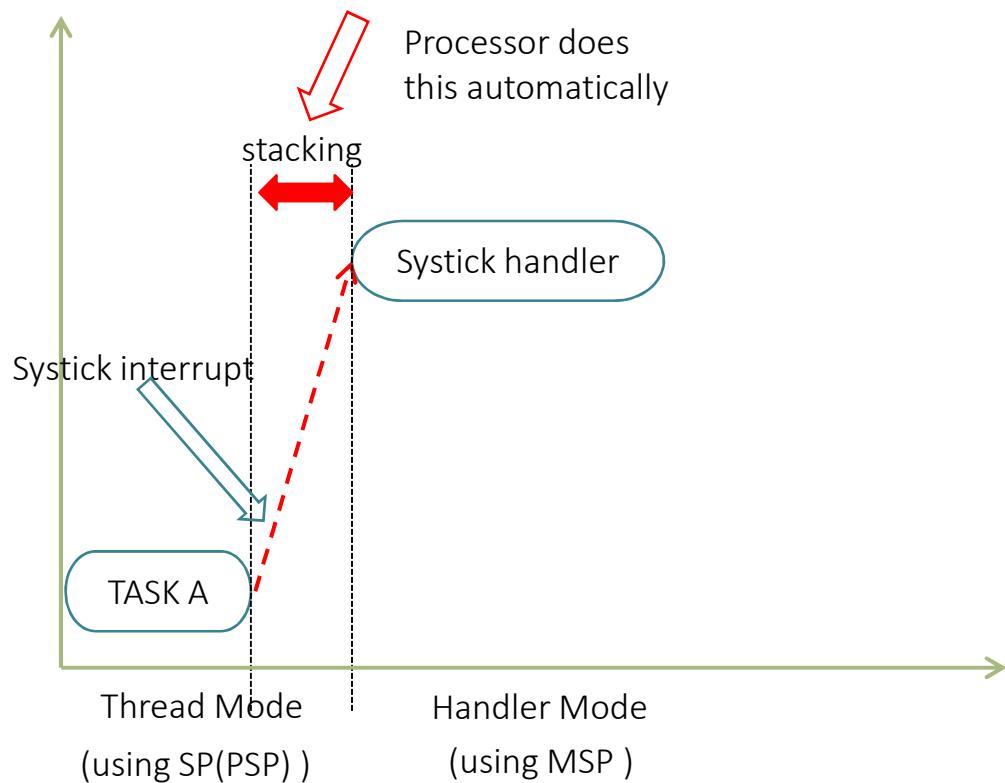
1. Processor core registers R0,R1,R2,R3,R12,LR,PC,xPSR(stack frame) are saved on to the task's private stack automatically by the processor **SysTick interrupt entry sequence**.
2. If Context Switch is required then SysTick timer will pend the PendSV Exception and PendSV handler runs
3. Processor core registers (R4-R11, R14) have to be saved manually on the task's private stack memory (**Saving the context**)

Task switching out procedure

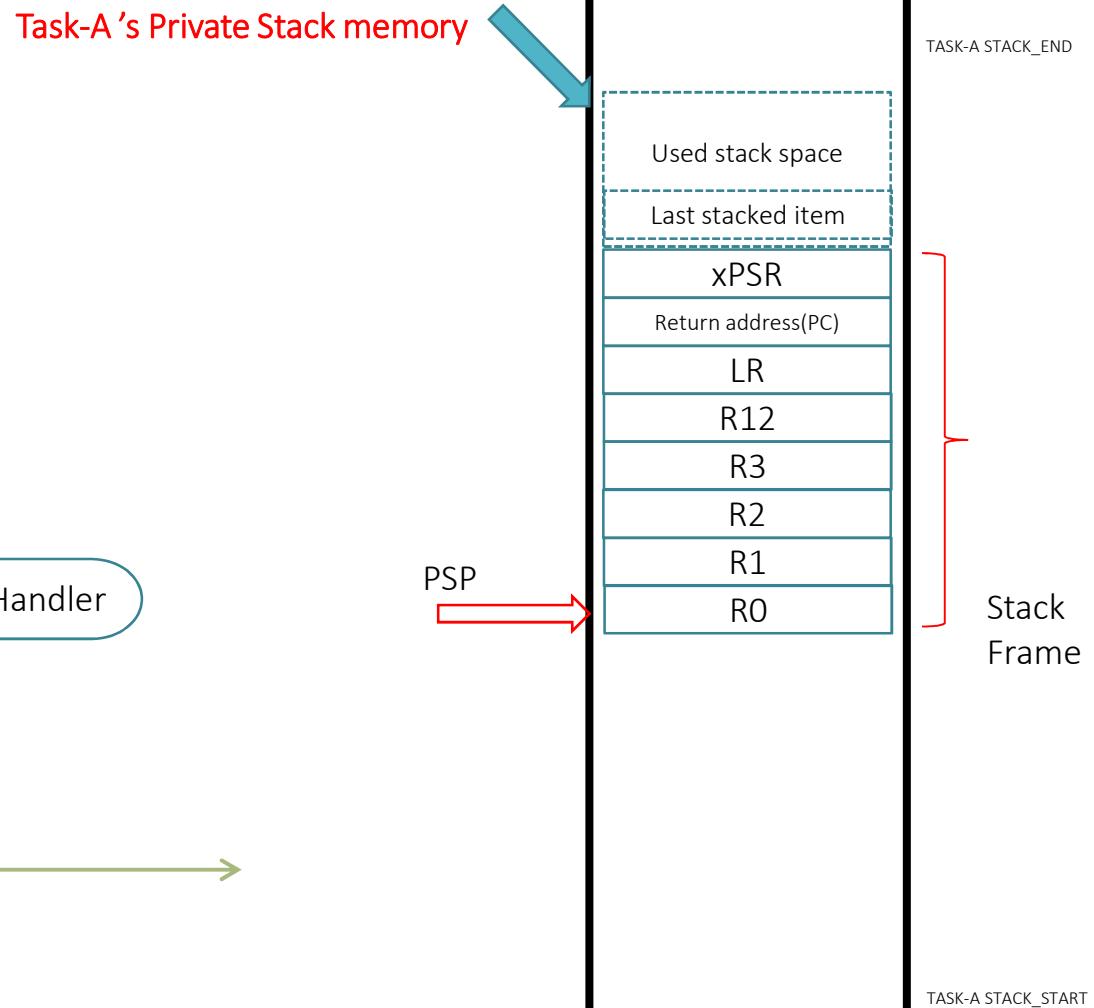
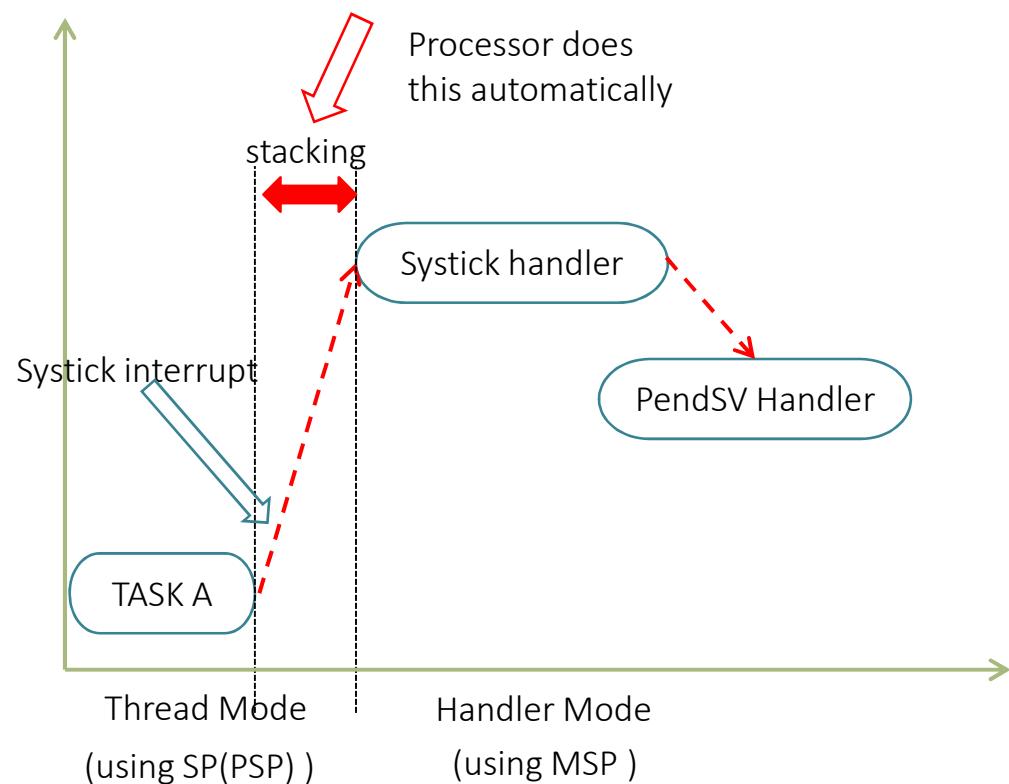
4. Save the new top of stack value (PSP) into first member of the TCB
5. Select the next potential Task to execute on the CPU. Taken care by `vTaskSwitchContext()` implemented in tasks.c

```
/* Select a new task to run using either the generic C or port
optimised asm code. */
taskSELECT_HIGHEST_PRIORITY_TASK();
```

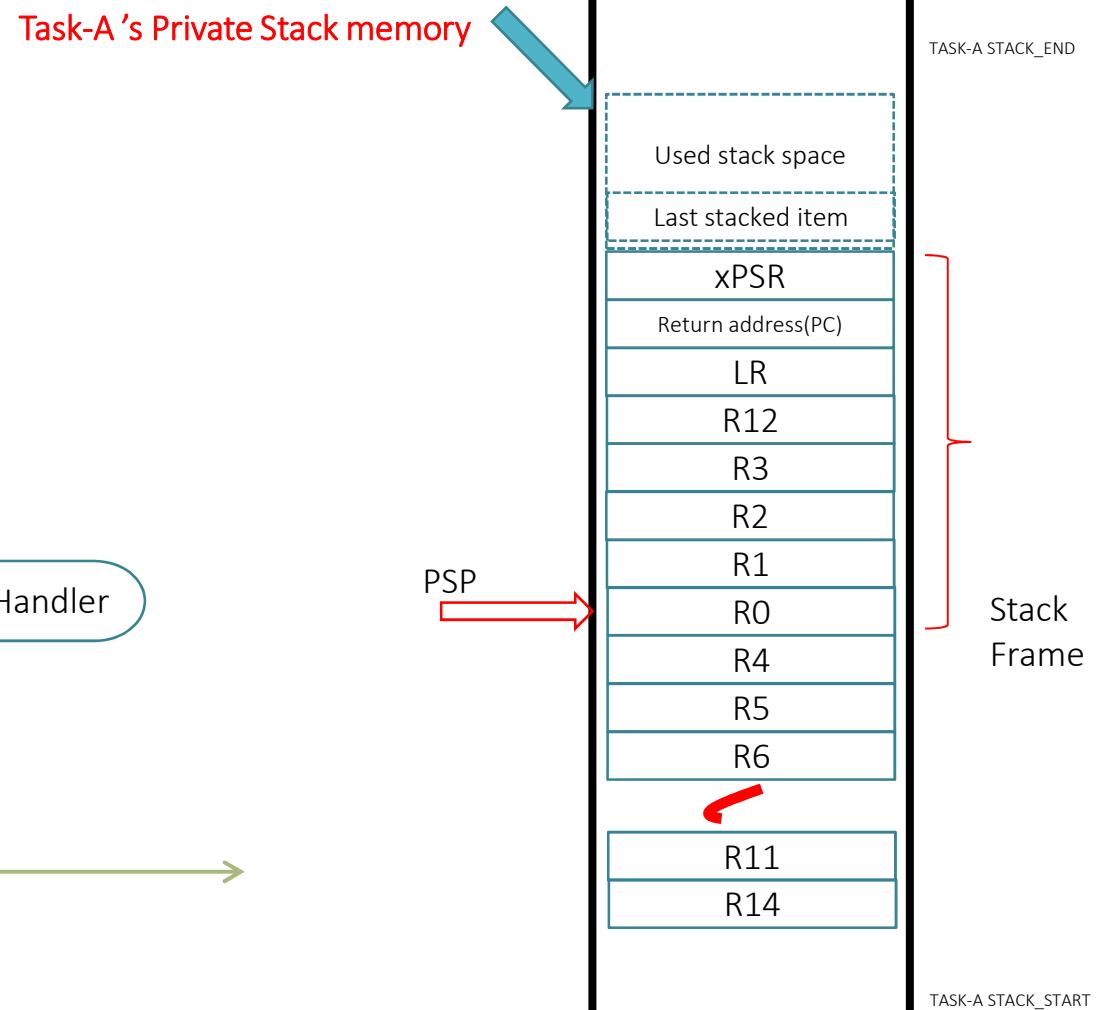
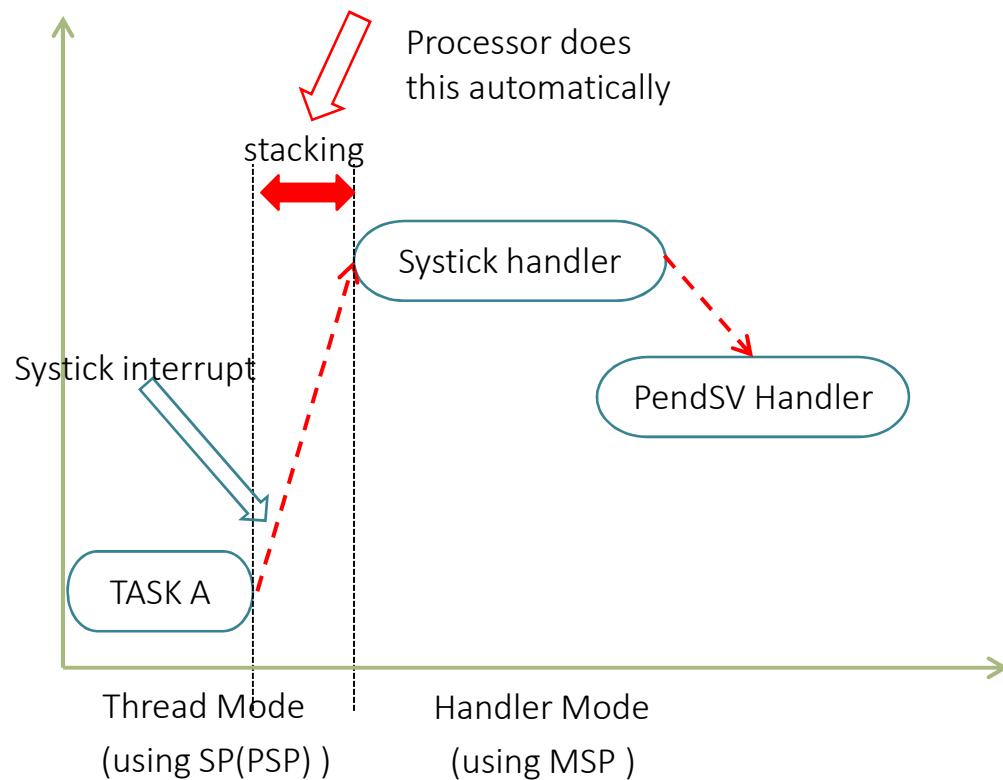
1 Exception Entry



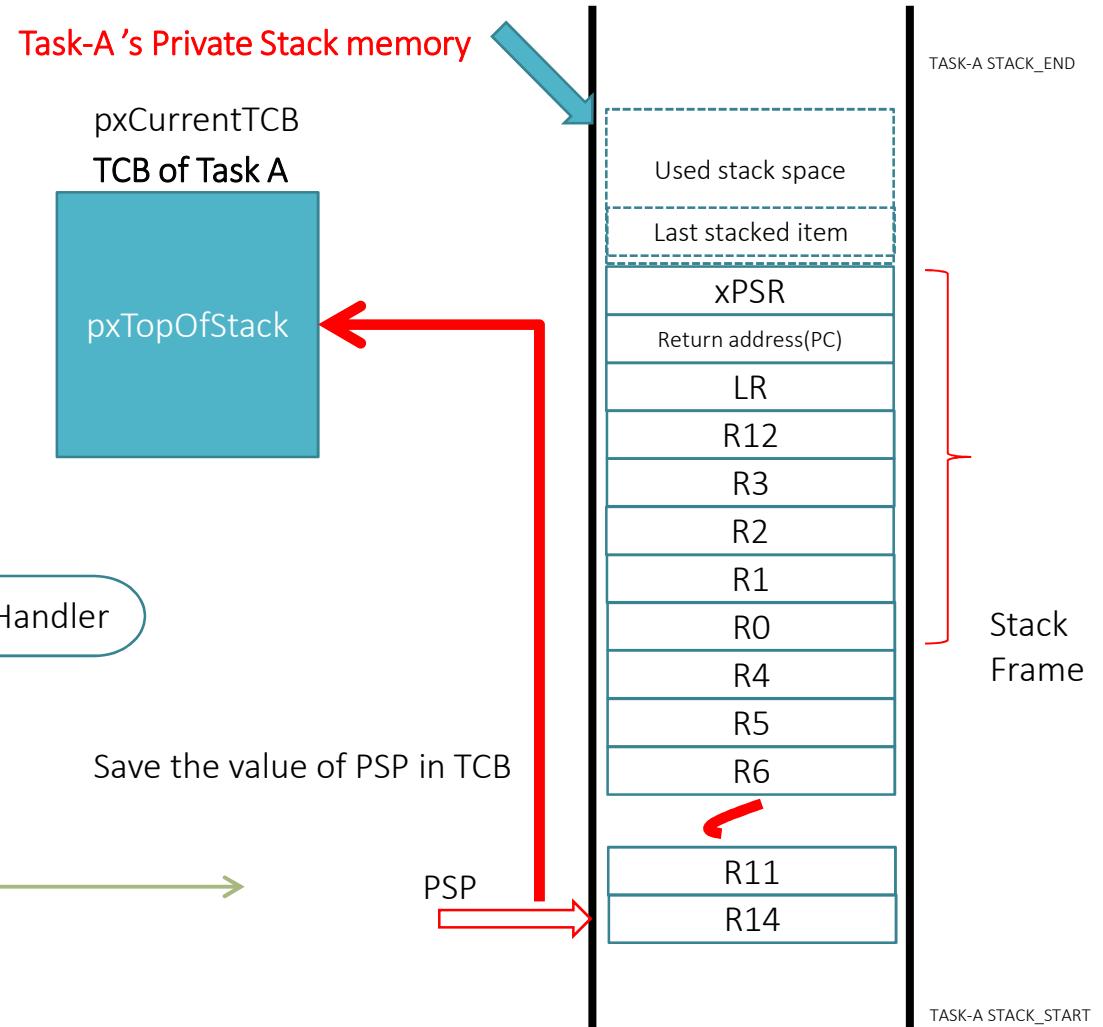
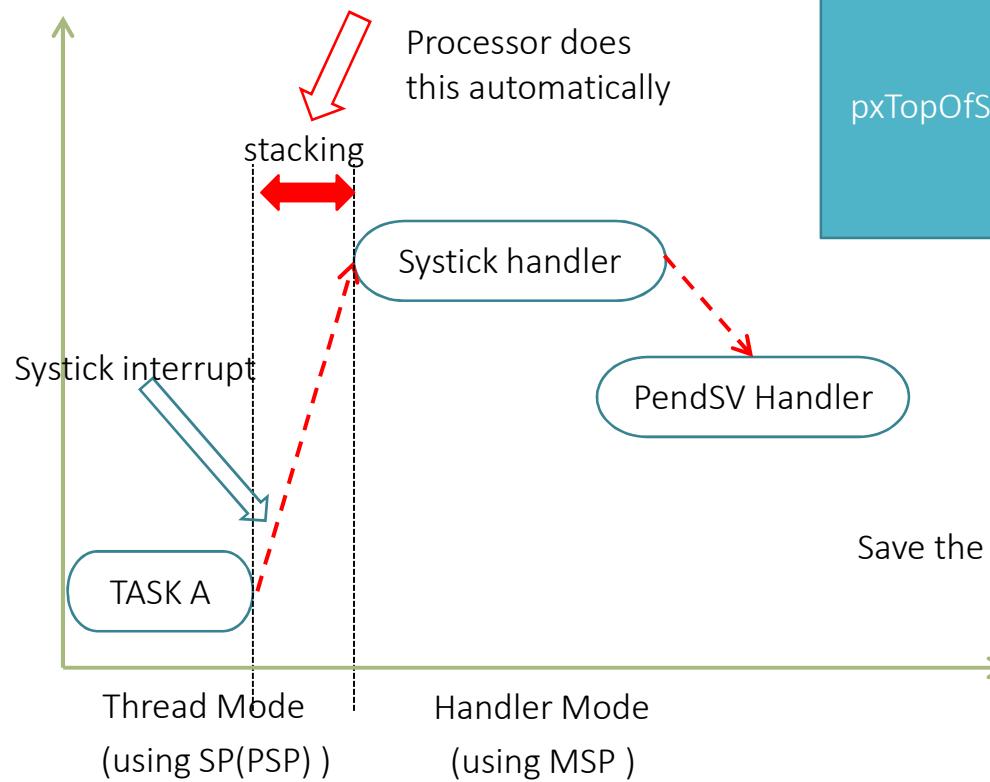
2 PendSV Handler Entry



3 Save core registers



4 Save PSP Into TCB

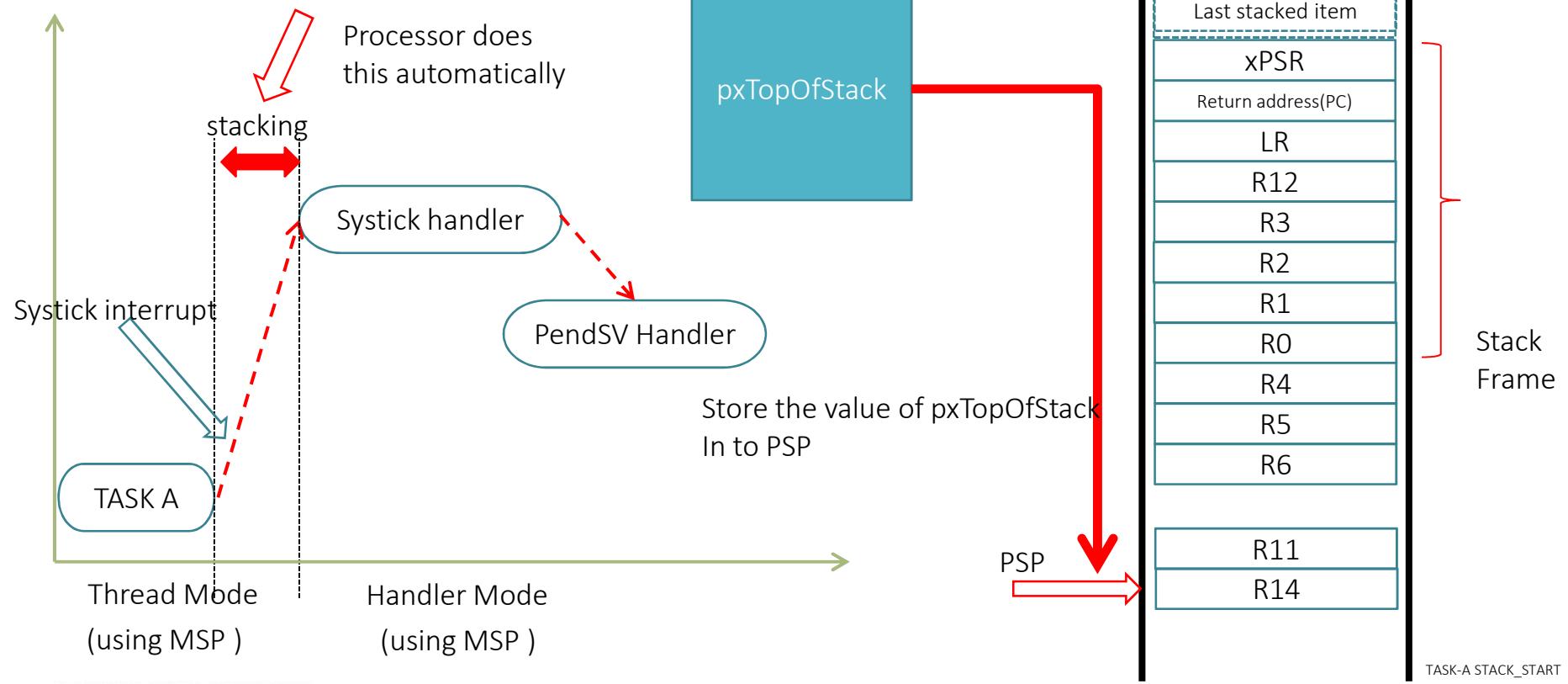


Task Switching In Procedure

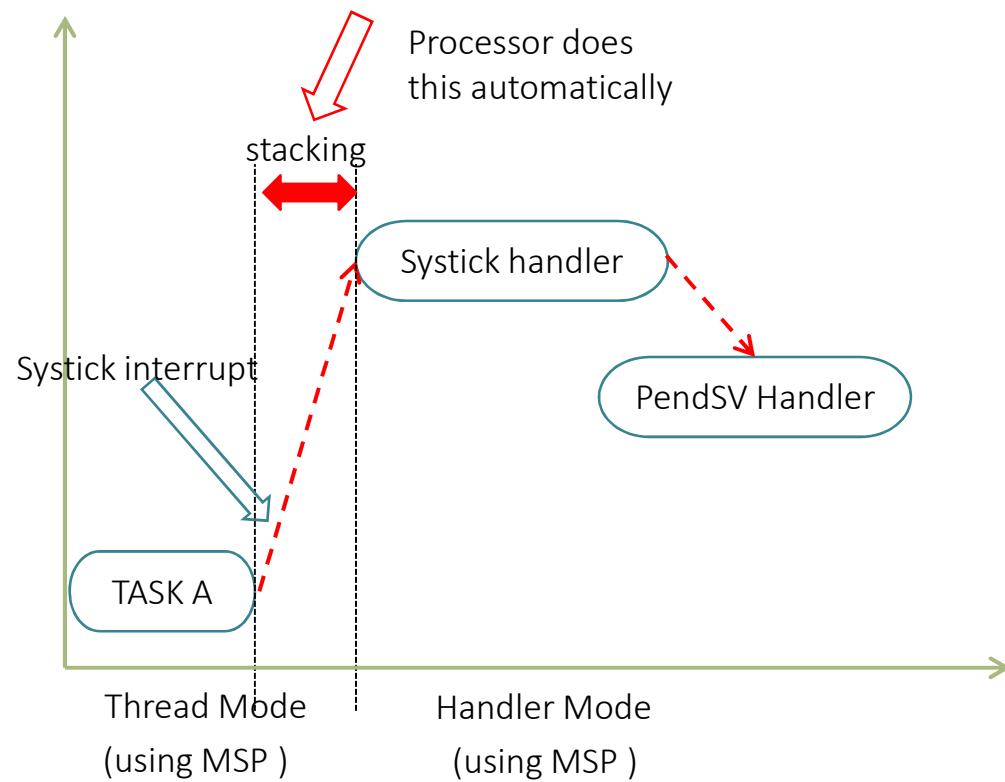
So, at this time, we already know which task(TCB) should be switched in .That means new switchable task's TCB can be accessed by `pxCurrentTCB`

1. First get the address of top of stack. Copy the value of `pxTopOfStack` in to `PSP` register
2. Pop all the registers (`R4-R11, R14`) (Restoring the context)
3. Exception exit : Now `PSP` is pointing to the start address of the stack frame which will be popped out automatically due to exception exit .

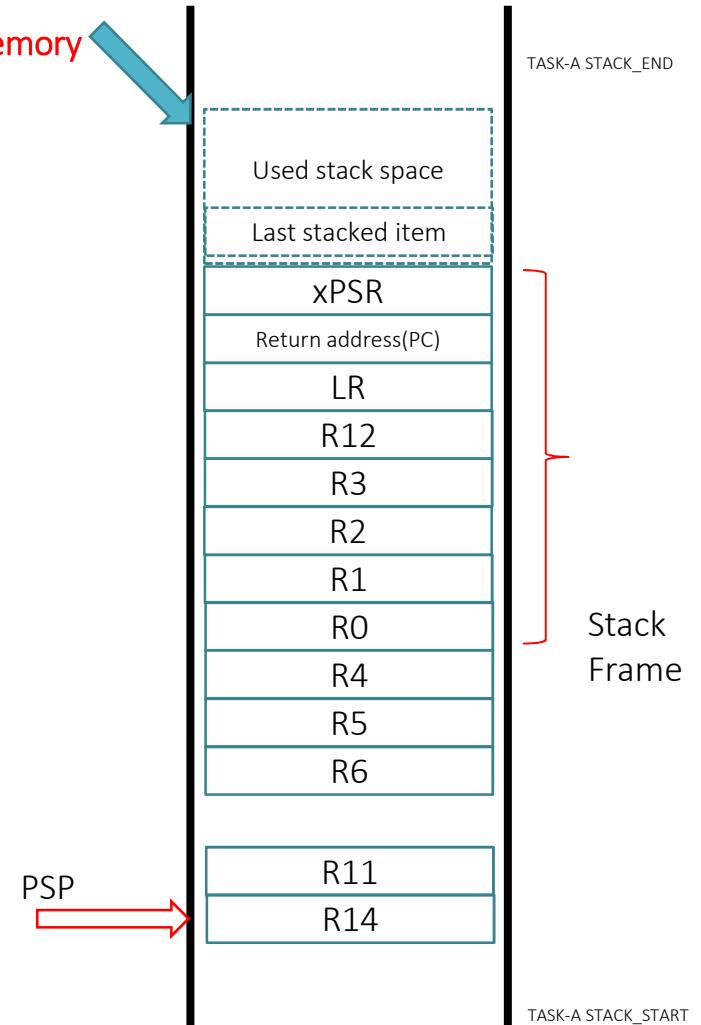
1. Load PSP



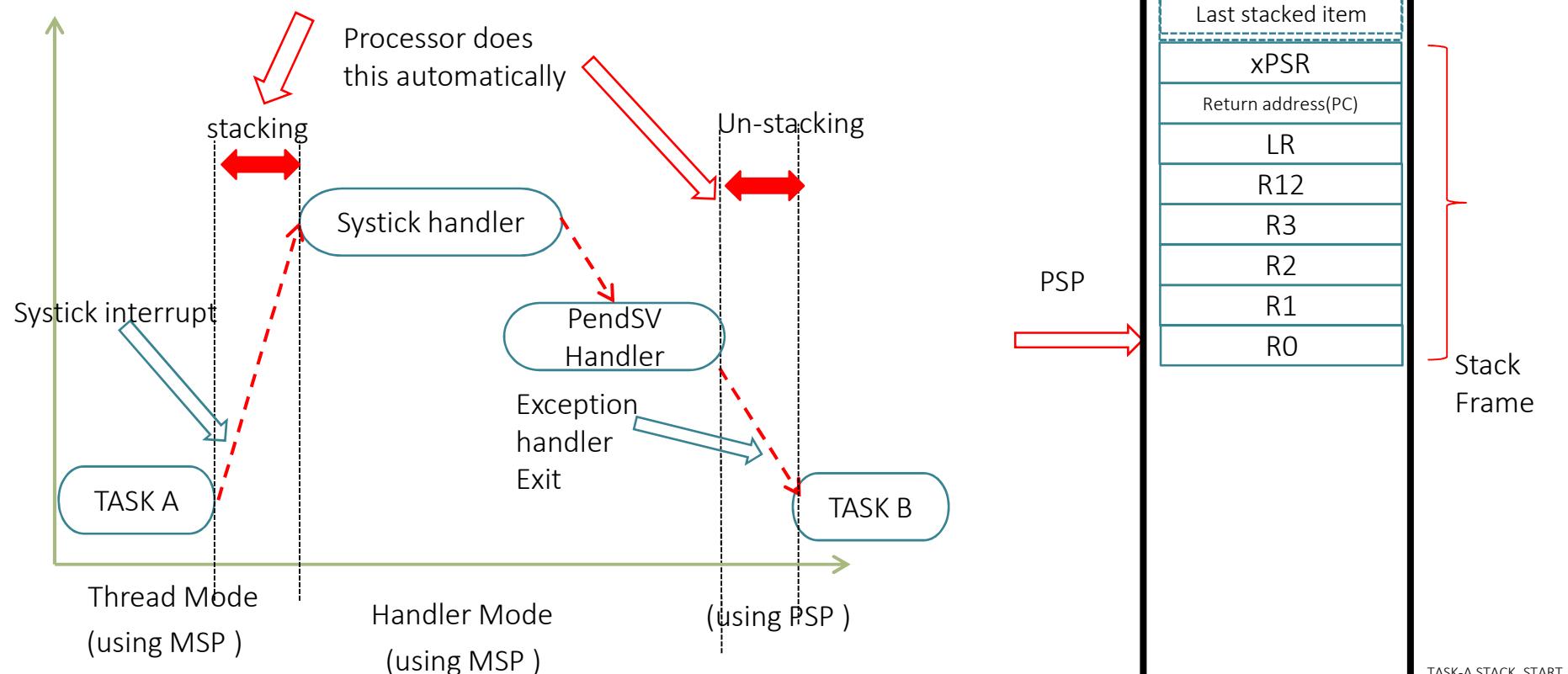
2. POP all Core Registers

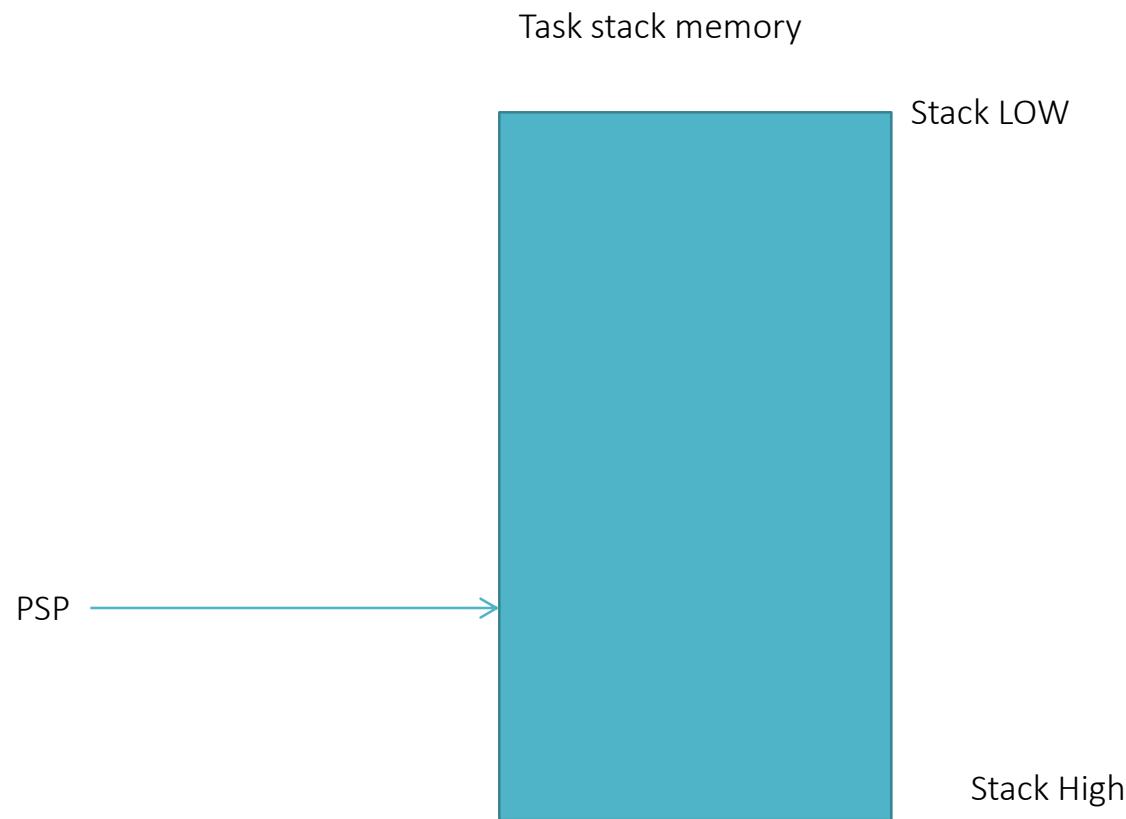


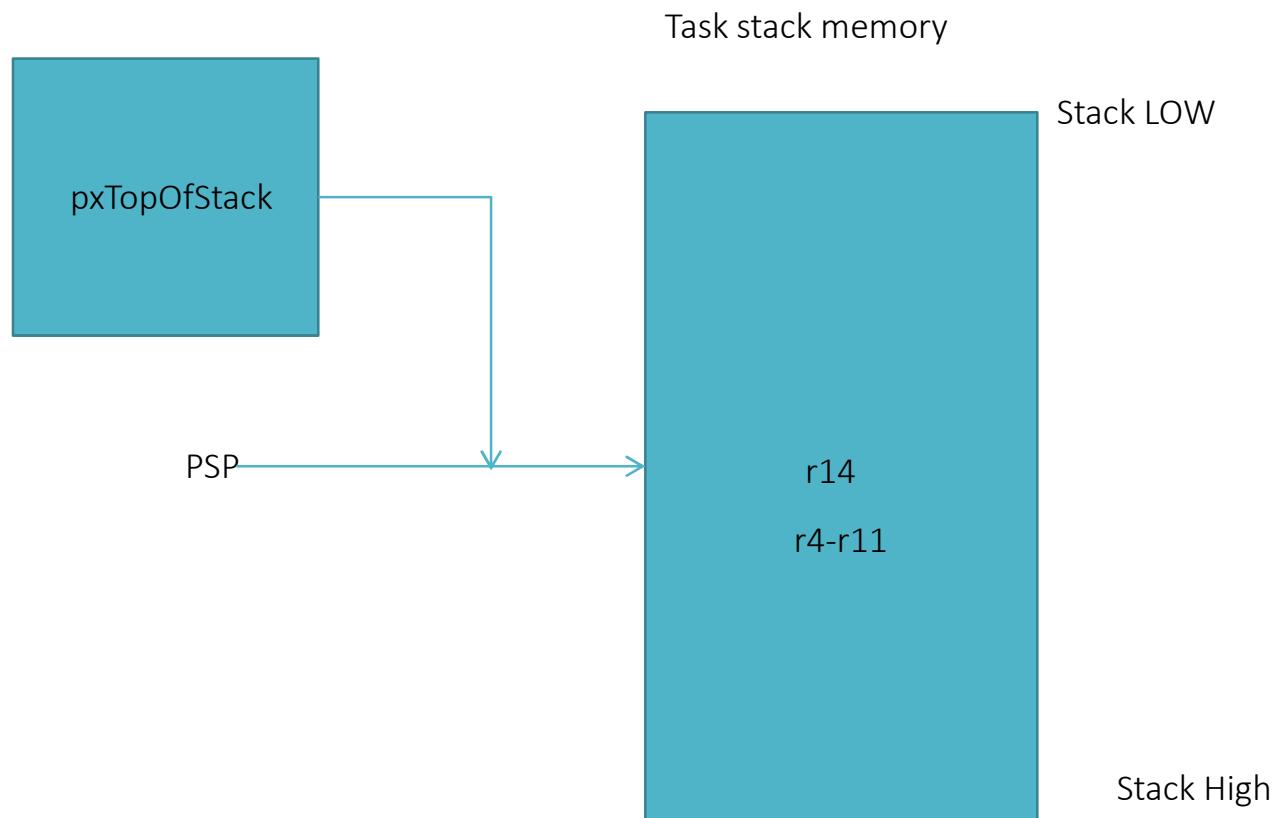
Task-B's Private Stack memory

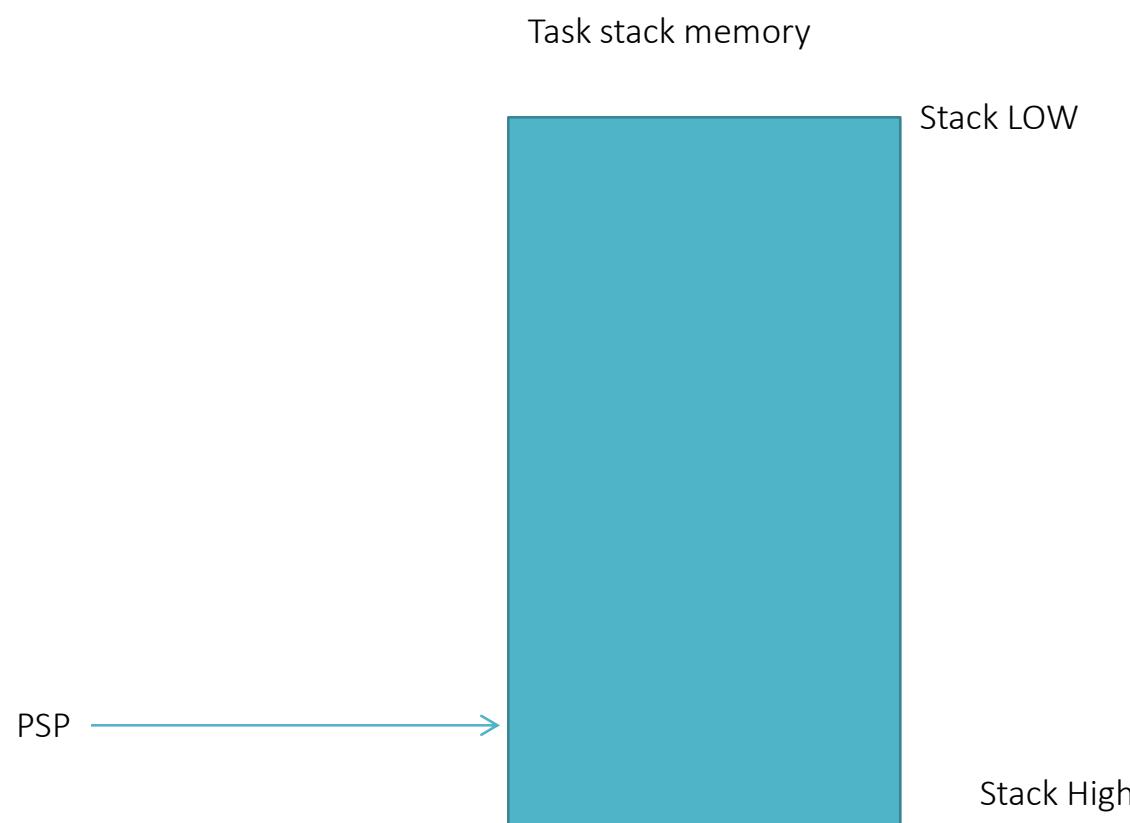


3. Exception Exit









Our embedded system courses are well suited for beginners to intermediate students, job seekers and professionals .

Life time access through udemy

Dedicated support team

Course completion certificate

Accurate Closed captions (subtitles) and transcripts

Step by step course coverage

30 days money back guarantee

Browse all courses on
MCU programming , RTOS,
embedded Linux
@ www.fastbitlab.com

PART-2

Mastering RTOS: Hands on FreeRTOS and STM32Fx with Debugging

Learn Running/Porting FreeRTOS Real Time Operating System on
STM32F4x and ARM cortex M based Microcontrollers

Created by :

FastBit Embedded Brain Academy

Visit www.fastbitlab.com

for all online video courses on MCU programming, RTOS and
embedded Linux

COPYRIGHT © BHARATI SOFTWARE 2016.

About FastBit EBA

FastBit Embedded Brain Academy is an online training wing of Bharati Software. We leverage the power of the internet to bring online courses at your fingertip in the domain of embedded systems and programming, microcontrollers, real-time operating systems, firmware development, Embedded Linux.

All our online video courses are hosted in Udemy E-learning platform which enables you to exercise 30 days no questions asked money back guarantee.

For more information please visit : www.fastbitlab.com

Email : contact@fastbitlab.com

Exercise

Create 2 Tasks in your FreeRTOS application *led_task* and *button_task*.

Button Task should continuously poll the button status of the board and if pressed it should update the flag variable.

Led Task should turn on the LED if button flag is SET, otherwise it should turn off the LED.

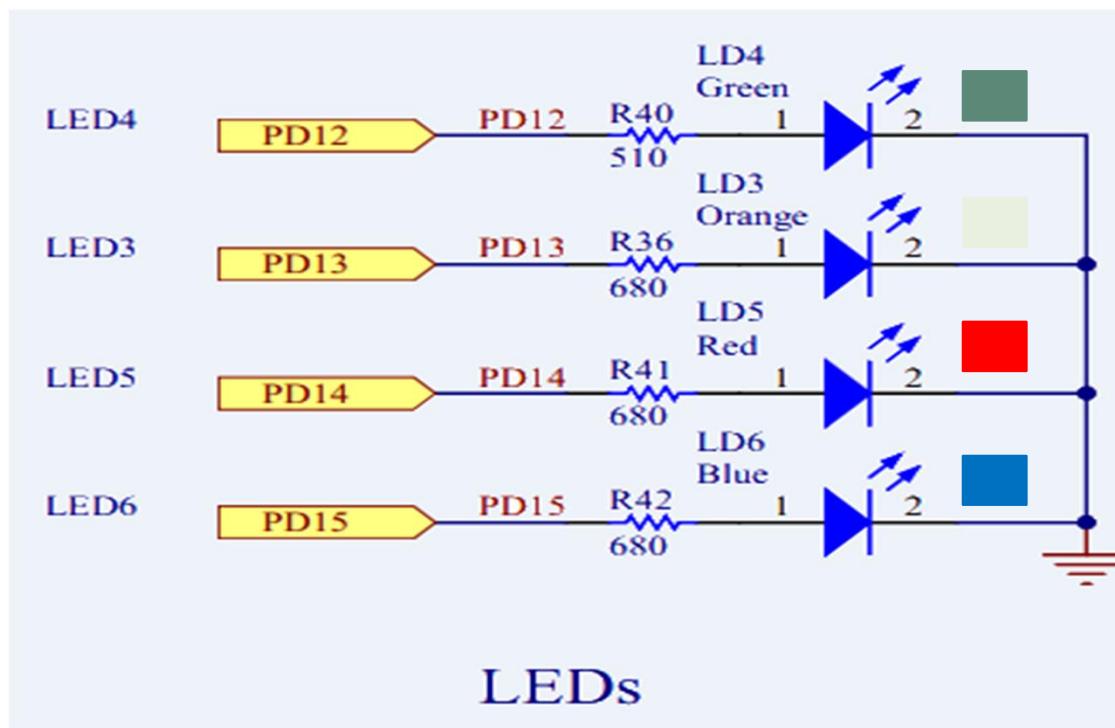
Use same freeRTOS task priorities for both the tasks.

Note :

On nucleo-F446RE board the LED is connected to PA5 pin and button is connected to PC13

If you are using any other board, then please find out where exactly the button and LEDs are connected on your board.

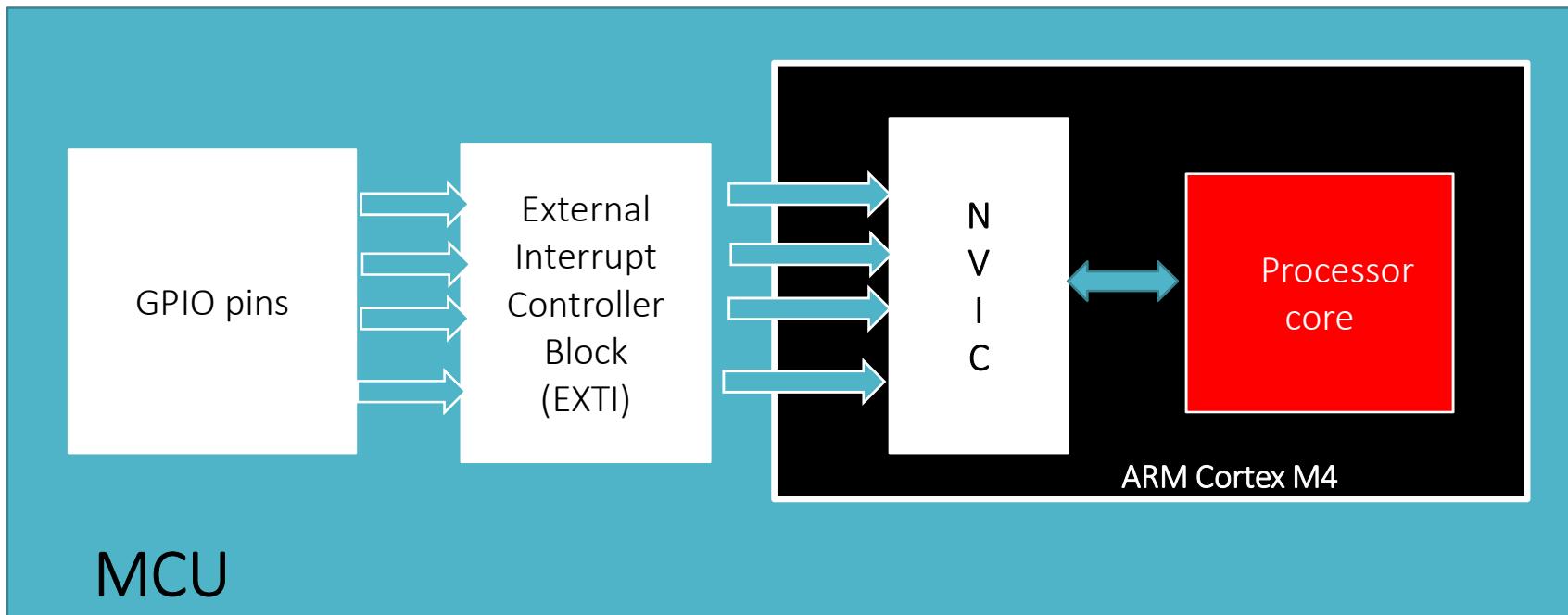
Discovery Board LEDs



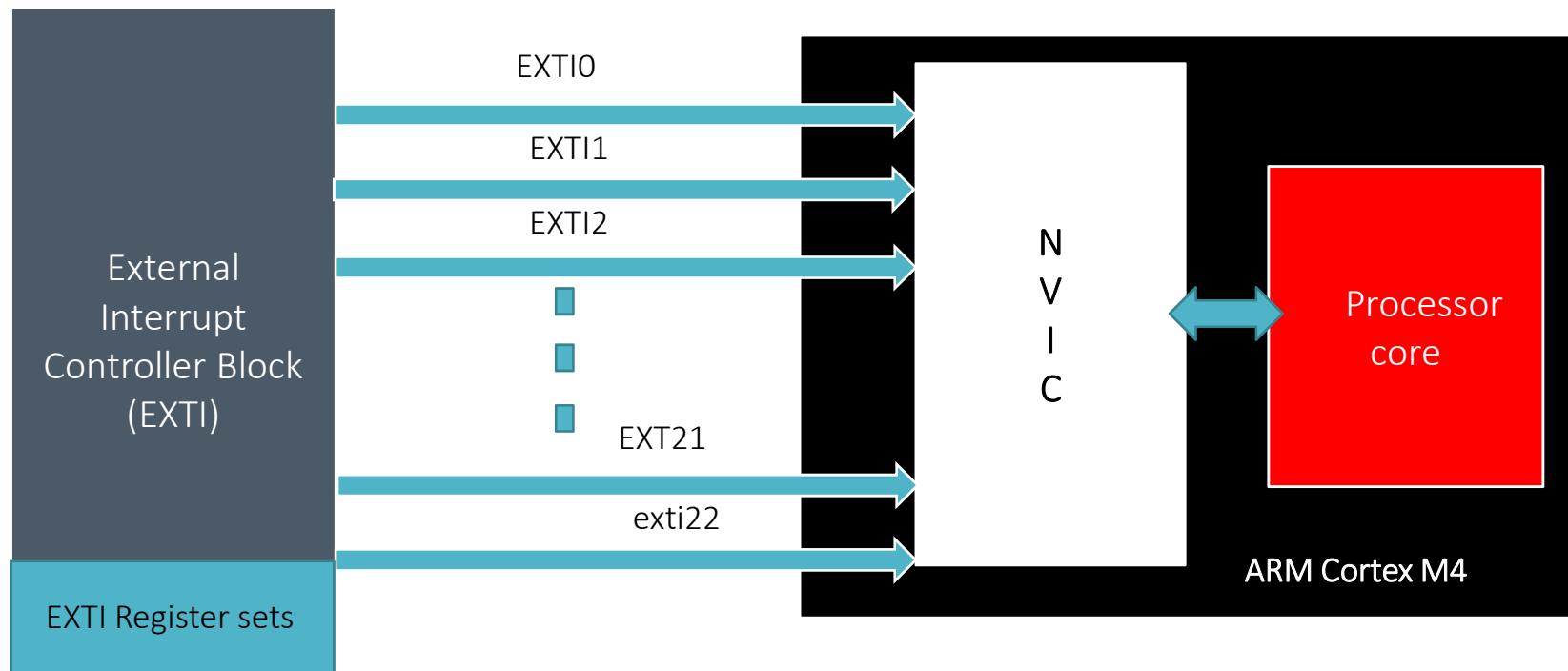
Exercise

Write a FreeRTOS application which creates only 1 task : *led_task* and it should toggle the led when you press the button by checking the button status flag.

The button interrupt handler must update the button status flag.



GPIOs Interrupt delivery to the Processor in STM32 MCUs



Note

Tasks run in “**Thread mode**” of the ARM cortex Mx processor

ISRs run in “**Handler mode**” of the ARM Cortex Mx processor

When interrupt triggers the processor mode changes to “Handler mode” and ISR will be executed.

Once the ISR exits and if there are no “pended” interrupts in the processor then task execution will be resumed.

Task Notification APIs

COPYRIGHT © BHARATI SOFTWARE 2016.

RTOS Task Notification

Each RTOS task has a 32-bit notification value which is initialised to zero when the RTOS task is created

An RTOS task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value in a number of different ways. For example, a notification may overwrite the receiving task's notification value, or just set one or more bits in the receiving task's notification value.

Wait and Notify APIs

xTaskNotifyWait()

xTaskNotify()

xTaskNotifyWait()

If a task calls `xTaskNotifyWait()` , then it waits with an optional timeout until it receives a notification from some other task or interrupt handler.

xTaskNotifyWait() Prototype

```
BaseType_t xTaskNotifyWait( uint32_t  
ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit, uint32_t  
*pulNotificationValue, TickType_t xTicksToWait );
```

**This explanation is taken from <https://www.freertos.org/xTaskNotifyWait.html>*

xTaskNotifyWait() Parameters

ulBitsToClearOnEntry

Any bits set in ulBitsToClearOnEntry will be cleared in the calling RTOS task's notification value on entry to the xTaskNotifyWait() function (before the task waits for a new notification) provided a notification is not already pending when xTaskNotifyWait() is called. For example, if ulBitsToClearOnEntry is 0x01, then bit 0 of the task's notification value will be cleared on entry to the function. Setting ulBitsToClearOnEntry to 0xffffffff (ULONG_MAX) will clear all the bits in the task's notification value, effectively clearing the value to 0.

**This explanation is taken from <https://www.freertos.org/xTaskNotifyWait.html>*

xTaskNotifyWait() Parameters

ulBitsToClearOnExit

Any bits set in ulBitsToClearOnExit will be cleared in the calling RTOS task's notification value before xTaskNotifyWait() function exits if a notification was received. The bits are cleared after the RTOS task's notification value has been saved in *pulNotificationValue (see the description of pulNotificationValue below). For example, if ulBitsToClearOnExit is 0x03, then bit 0 and bit 1 of the task's notification value will be cleared before the function exits. Setting ulBitsToClearOnExit to 0xffffffff (ULONG_MAX) will clear all the bits in the task's notification value, effectively clearing the value to 0.

**This explanation is taken from <https://www.freertos.org/xTaskNotifyWait.html>*

xTaskNotifyWait() Parameters

pulNotificationValue

Used to pass out the RTOS task's notification value. The value copied to *pulNotificationValue is the RTOS task's notification value as it was before any bits were cleared due to the ulBitsToClearOnExit setting. If the notification value is not required then set pulNotificationValue to NULL.

**This explanation is taken from <https://www.freertos.org/xTaskNotifyWait.html>*

xTaskNotifyWait() Parameters

xTicksToWait

The maximum time to wait in the Blocked state for a notification to be received if a notification is not already pending when xTaskNotifyWait() is called. The RTOS task does not consume any CPU time when it is in the Blocked state.

The time is specified in RTOS tick periods. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds into a time specified in ticks.

**This explanation is taken from <https://www.freertos.org/xTaskNotifyWait.html>*

xTaskNotifyWait() Return value

Returns:

pdTRUE if a notification was received, or a notification was already pending when xTaskNotifyWait() was called.

pdFALSE if the call to xTaskNotifyWait() timed out before a notification was received

`xTaskNotify()`

`xTaskNotify()` is used to send an event directly to and potentially unblock an RTOS task, and optionally update the receiving task's notification value in one of the following ways:

- *Write a 32-bit number to the notification value*
- *Add one (increment) the notification value*
- *Set one or more bits in the notification value*
- *Leave the notification value unchanged*

This function must not be called from an interrupt service routine (ISR). Use [`xTaskNotifyFromISR\(\)`](#) instead.

xTaskNotify() Prototype

```
BaseType_t xTaskNotify( TaskHandle_t xTaskToNotify,  
uint32_t ulValue, eNotifyAction eAction );
```

**This explanation is taken from <https://www.freertos.org/xTaskNotify.html>*

xTaskNotify Parameters

xTaskToNotify

The handle of the RTOS task being notified.
This is the *subject task*;

**This explanation is taken from <https://www.freertos.org/xTaskNotify.html>*

xTaskNotify() Parameters

ulValue

Used to update the notification value of the subject task. See the description of the eAction parameter below.

**This explanation is taken from <https://www.freertos.org/xTaskNotify.html>*

xTaskNotify() Parameters

eAction

An enumerated type that can take one of the values documented in the table below in order to perform the associated action

eNoAction

eIncrement

eSetValueWithOverwrite

MS to Ticks conversion

$$xTicksToWait = (xTimeInMs * \text{configTICK_RATE_HZ}) / 1000$$

Example :

If configTICK_RATE_HZ is 500, then Systick interrupt is going to happen for every 2ms.

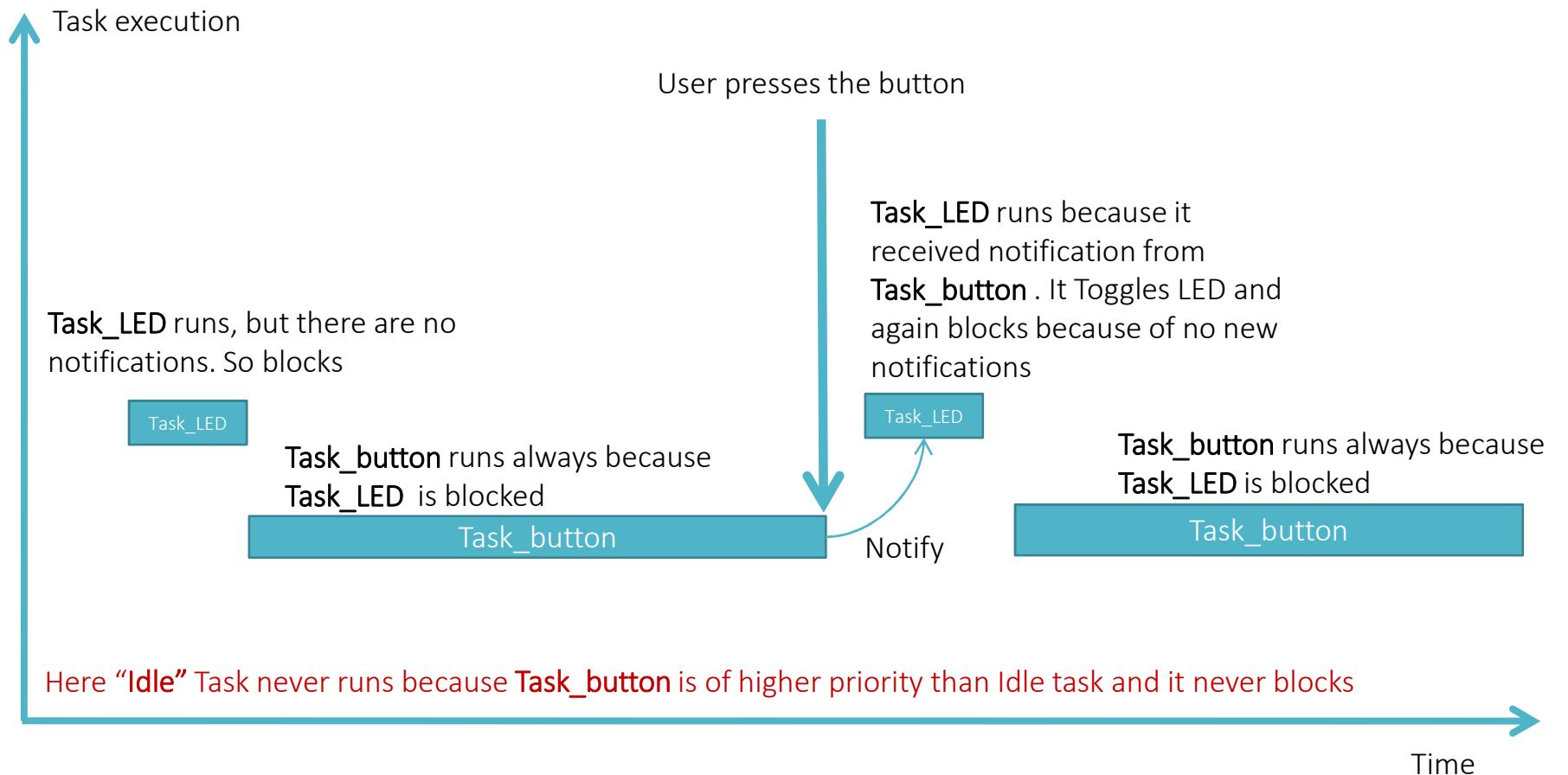
So, 500ms of delay is equivalent to 250 ticks duration

Exercise

Write a program which creates 2 tasks `task_led` and `task_button` with equal priorities.

When button is pressed, `task_button` should notify the `task_led` and `task_led` should run on the CPU to toggle the LED . Also `task_led` should print how many times user has pressed the button so far.

`task_led` should not unnecessarily run on the CPU and it should be in Block mode until it receives the notification from the `task_button` .



FreeRTOS: Licensing

COPYRIGHT © BHARATI SOFTWARE 2016.

FreeRTOS is Free “Don’t worry ☺”

You can use it in your commercial applications “No problem ☺”

No need to give any royalty to freertos.org “Awesome ☺”

Its based on GNU GPL license and you should make open your code changes made to FreeRTOS kernel “That’s Ok ☺”

You need not to open source your applications Written using freeRTOS API

Does it pass any safety standard ? No it doesn't ☹

Is it safe to use freeRTOS in Safety critical applications ? “No No No ☹”

Does freertos.org provide any legal protection ? No it doesn't 😞

Does freeRTOS.org provides any technical support ? No it doesn't 😞

FreeRTOS: Commercial licensing

COPYRIGHT © BHARATI SOFTWARE 2016.

FreeRTOS: Commercial licensing

If you want ,

Legal Support

Technical Support during your Product development

Ensure meeting safety standard

Then you have to go for Commercial Licensing of
freertos.org

FreeRTOS : Commercial licensing

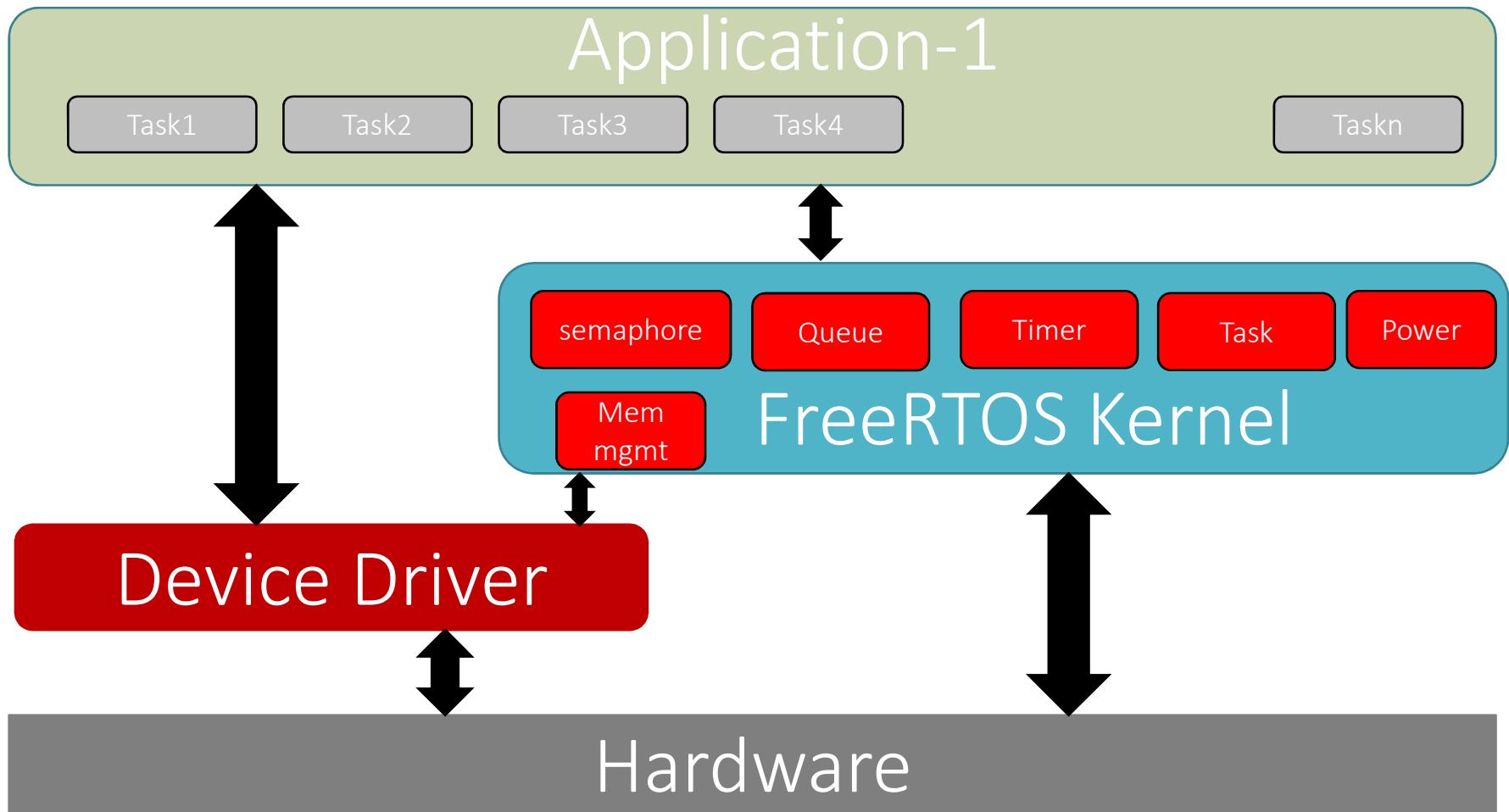
- ✓ SAFERTOSTM is a derivative version of FreeRTOS that has been analyzed, documented and tested to meet the stringent requirements of the IEC 61508 safety standard. This RTOS is audited to verify IEC 61508 SIL 3 conformance.
- ✓ OpenRTOSTM is a commercially licensed version of FreeRTOS. The OpenRTOS license does not contain any references to the GPL

License feature comparison

	FreeRTOS Open Source License	OpenRTOS Commercial License
Is it free?	Yes	No ✓
Can I use it in a commercial application?	Yes	Yes
Is it royalty free?	Yes	Yes
Is a warranty provided?	No ✓	Yes
Can I receive professional technical support on a commercial basis?	No, FreeRTOS is supported by an online community	Yes ✓
Is legal protection provided?	✓ No	Yes, IP infringement protection is provided
Do I have to open source my application code that makes use of the FreeRTOS services?	No	No
Do I have to open source my changes to the RTOS kernel?	✓ Yes	No ✓
Do I have to document that my product uses FreeRTOS?	Yes if you distribute source code	No
Do I have to offer to provide the FreeRTOS code to users of my application?	Yes if you distribute source code	No ✓

FreeRTOS API interface

COPYRIGHT © BHARATI SOFTWARE 2016.



Very important links

Download :

www.freertos.org

FreeRTOS Tutorial Books

http://shop.freertos.org/FreeRTOSTutorialBooksAndReferenceManuals_s/1825.htm

Creating a New FreeRTOS Project

<http://www.freertos.org/Creating-a-new-FreeRTOS-project.html>

FreeRTOS Quick Start Guide.

<http://www.freertos.org/FreeRTOSQuickStartGuide.html>

Books and kits

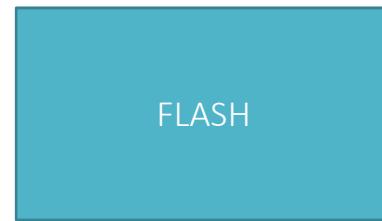
http://shop.freertos.org/RTOSPrimerBooksAndManuals_s/1819.htm

Overview of FreeRTOS Memory Management

RAM and Flash



RAM



FLASH

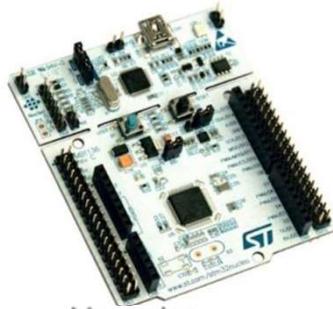
Every Microcontroller Consists of two types of memories : **RAM and Flash**

Usually RAM memory always less than FLASH memory

RAM and Flash



STM32F401RE



- Memories
 - up to 512 Kbytes of Flash memory
 - up to 96 Kbytes of SRAM

RAM and Flash

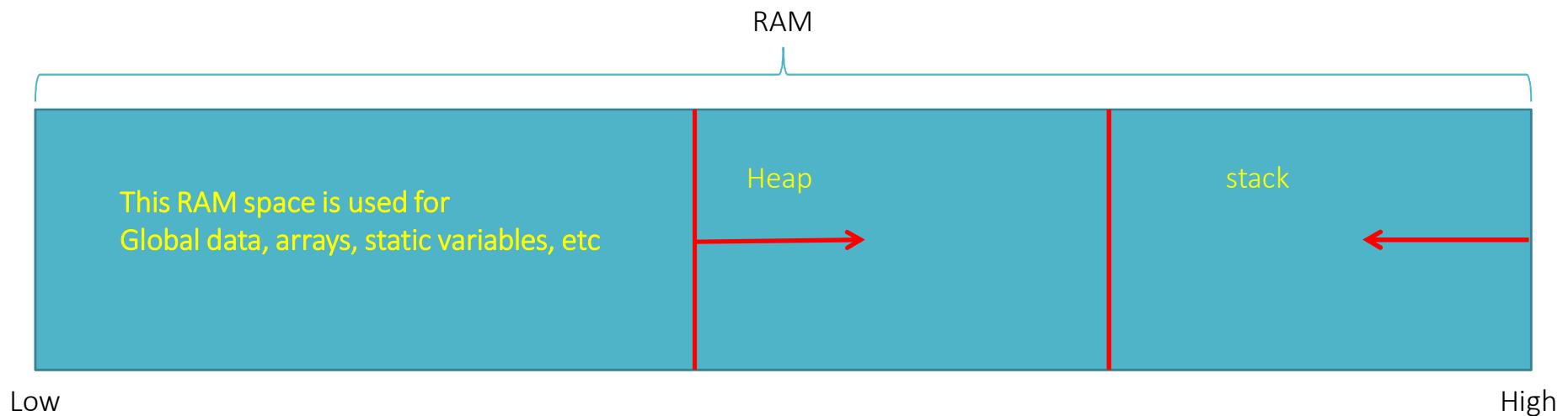
RAM

- 1) To store your application data like global arrays, global variables, etc
- 2) You can download code to RAM and Run (e.g patches)
- 1) A part of RAM is used as STACK to store local variables , function arguments, return address, etc
- 2) A part of RAM is used as HEAP for dynamic memory allocations

FLASH

- 1) Flash is used to hold your application code
- 2) Flash also holds constants like string initialization
- 3) Flash holds the vector table for interrupts and exceptions of the MCU

Stack and Heap in embedded Systems



Stack and Heap in embedded Systems

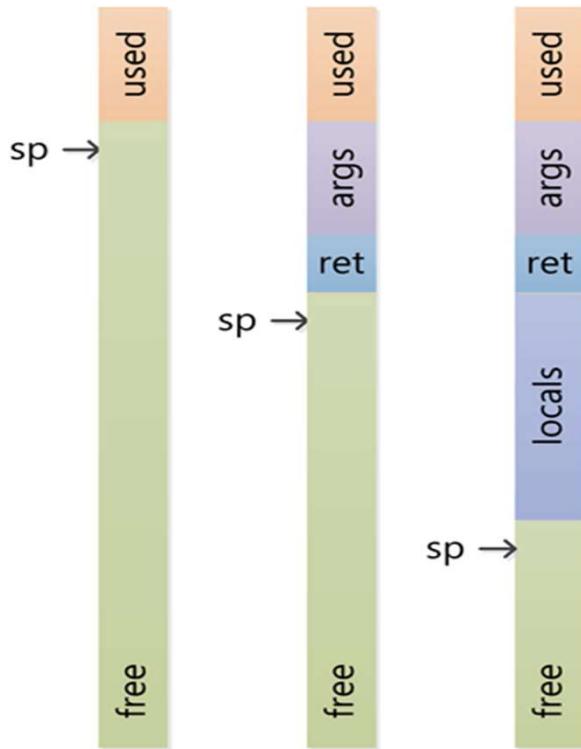


Fist in Last out Order Access



out of order access of memory

Stack



sp : Stack Pointer

used : Unavailable stack (already used)

args : function arguments

ret : return address

locals : local variable

free : available stack

Stack

```
char do_sum( int a , int b, int c )
{
    char x=1;
    char y=2;
    char *ptr;

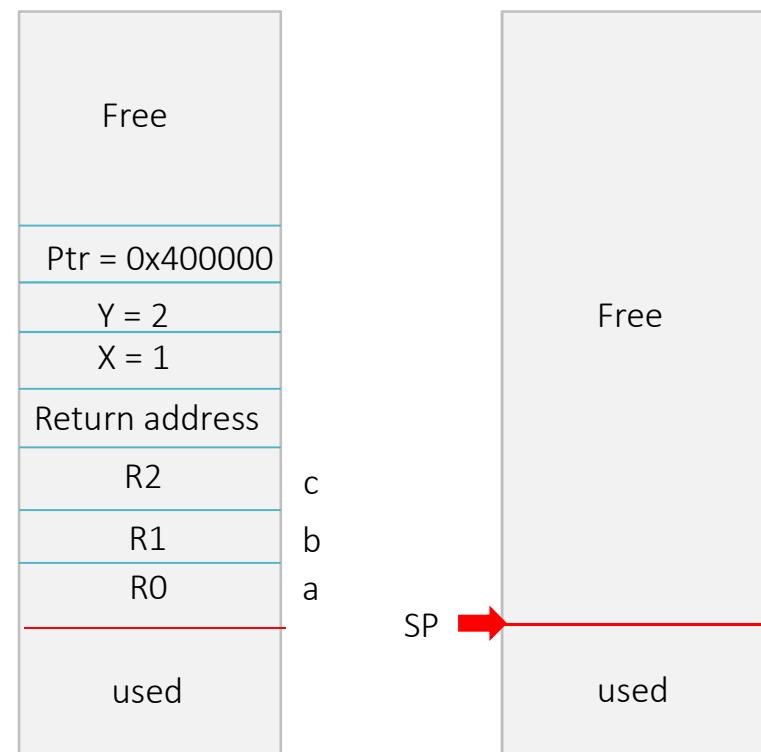
    ptr = malloc(100);
    x = a+b+c;

    return x;
}
```

Local variables }

Some operations }

Exiting [R0 = x] }



Heap



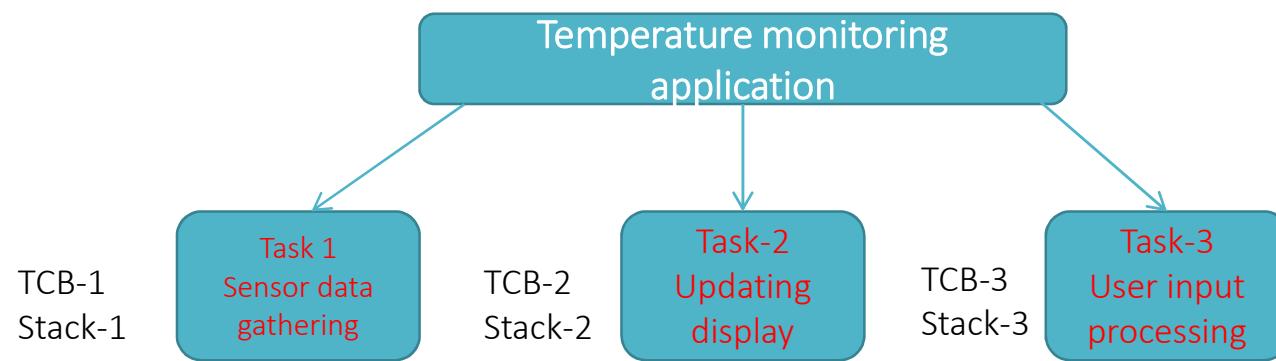
A heap is a general term used for any memory that is allocated dynamically and randomly.

I understand stack is managed by SP and dedicated instructions like PUSH n POP , how Heap is managed ?

How the heap is managed is really up to the runtime environment. C uses **malloc** and C++ uses **new** .

But for embedded systems **malloc** and **free** APIs are not suitable because they eat up large code space , lack of deterministic nature and fragmented over time as blocks of memory are allocated

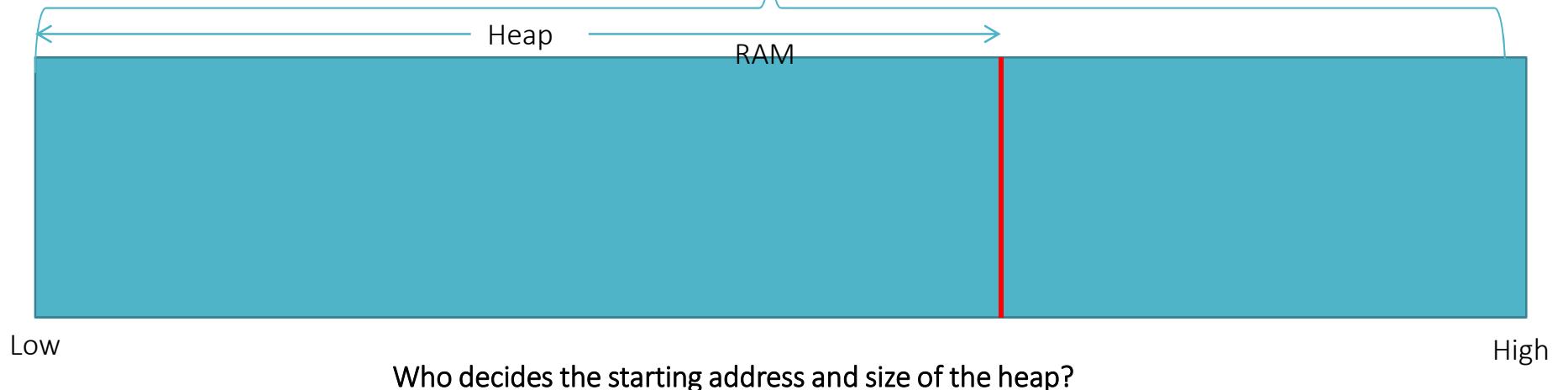
FreeRTOS Stack and heap management



Every task creation will consume some memory in RAM to store its TCB and stack

So , 1 task creation consumes **TCB size + Stack size** in RAM

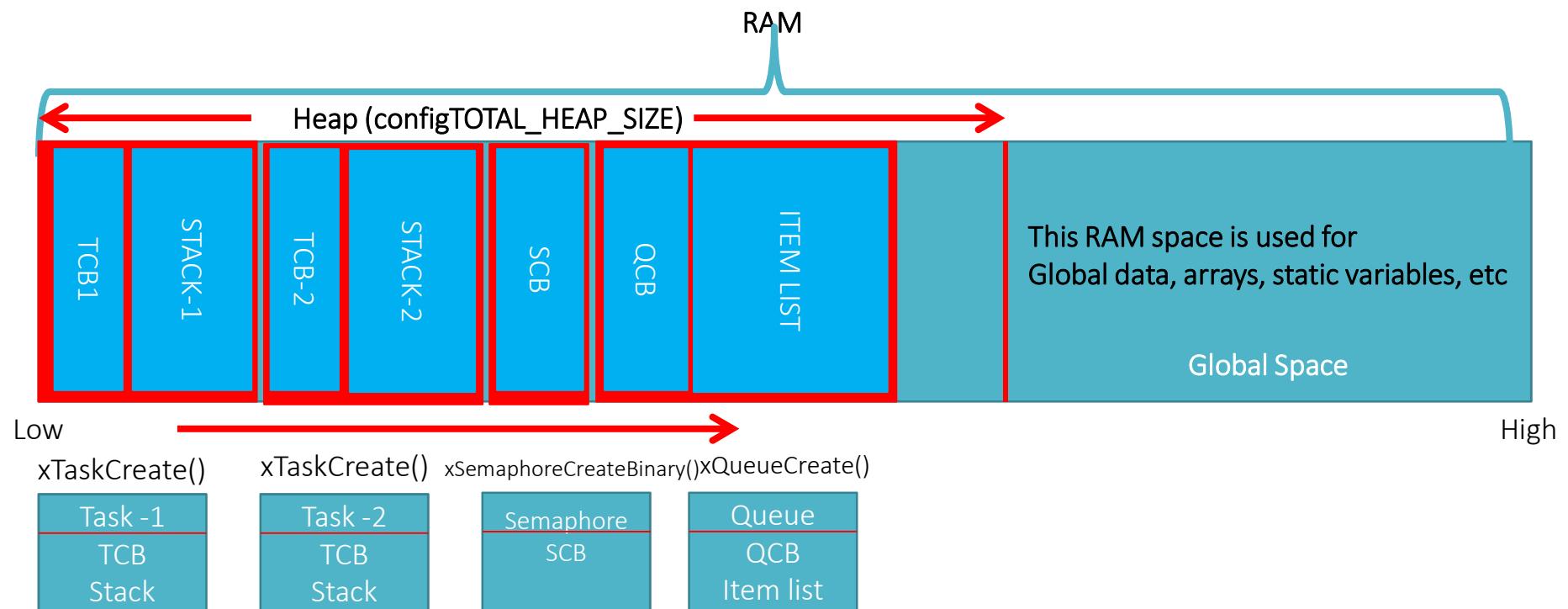
FreeRTOS Stack and heap

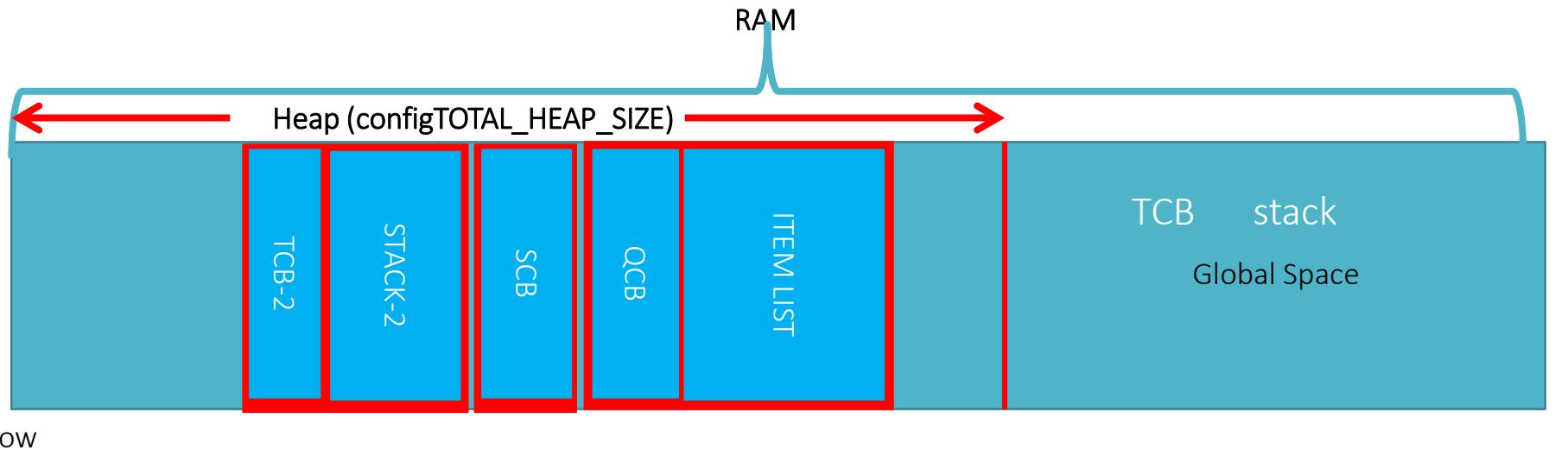


By default the [FreeRTOS heap](#) is declared by FreeRTOS kernel

Setting `configAPPLICATION_ALLOCATED_HEAP` to 1 allows the heap to instead be declared by the application

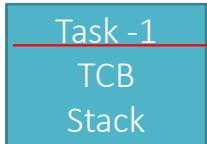
FreeRTOS Stack and heap





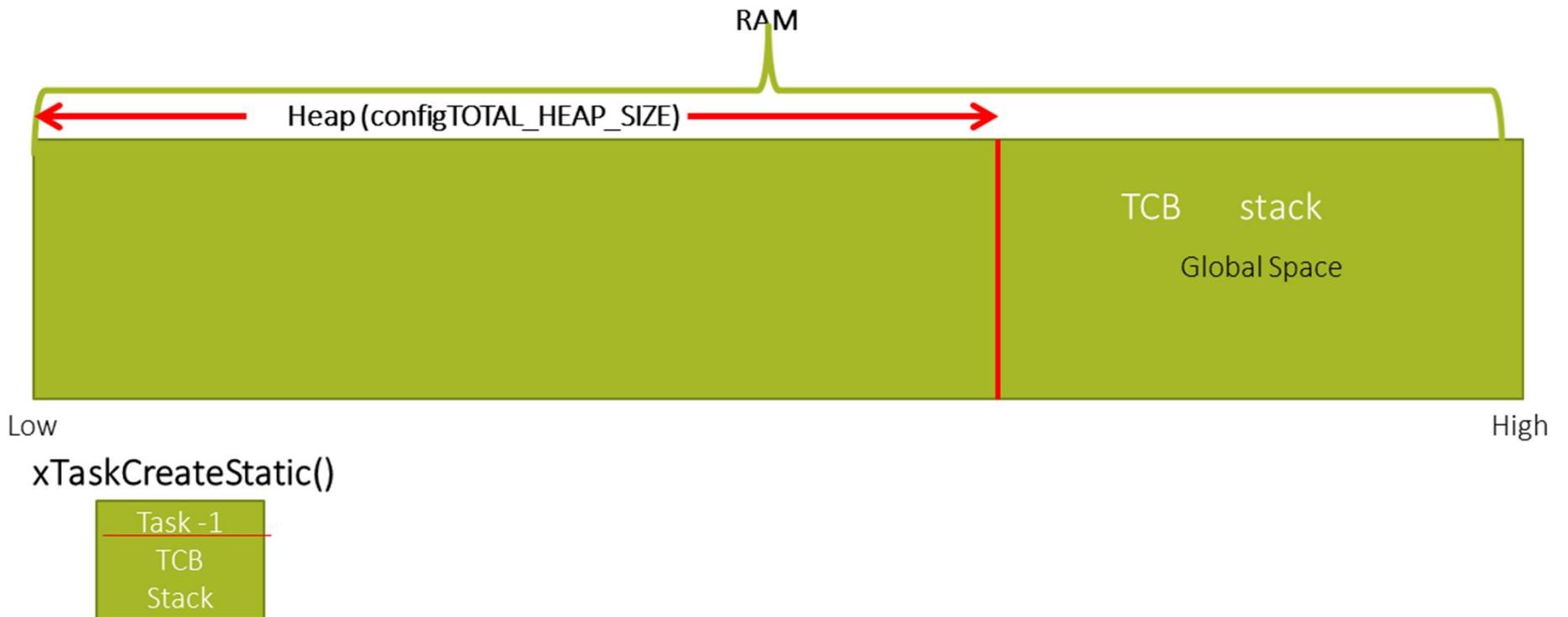
Low

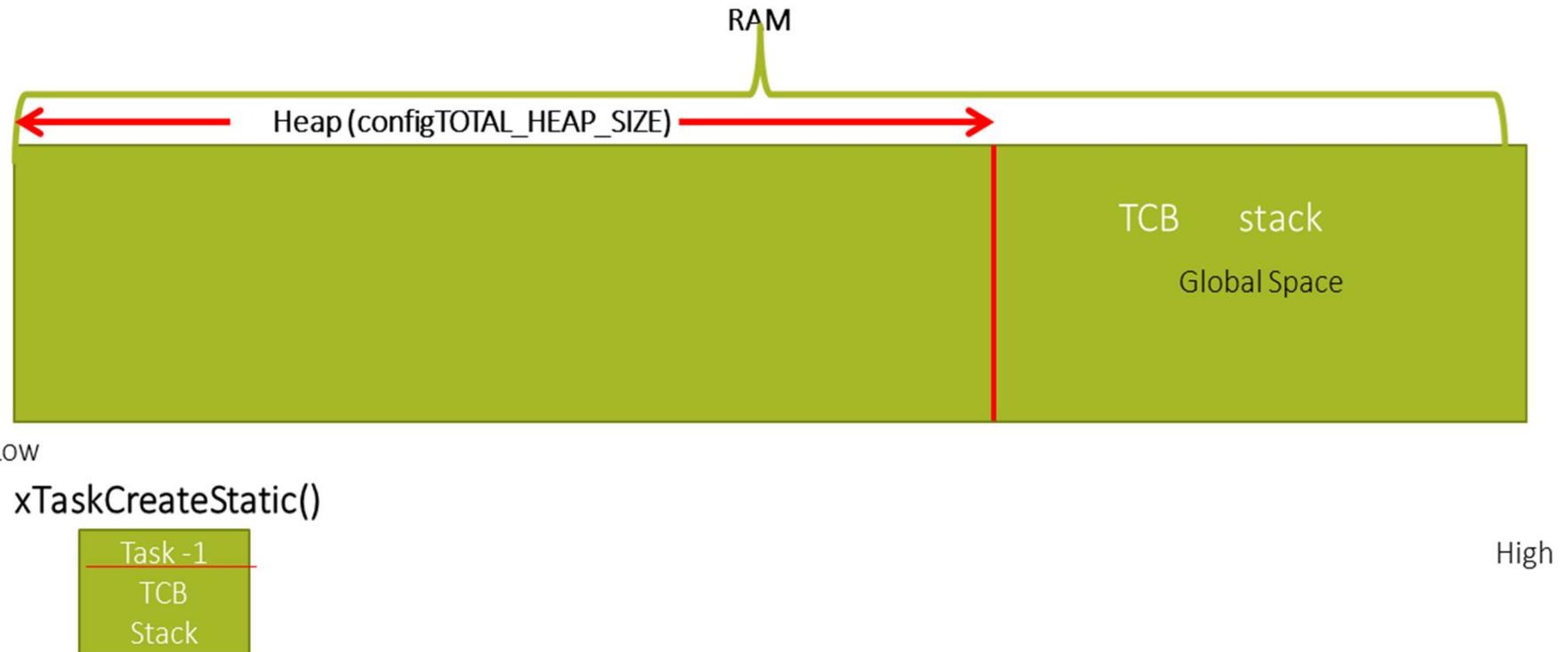
xTaskCreateStatic()



High

FreeRTOS Stack and heap





```

/* Dimensions the buffer that the task being created will use as its stack.
NOTE: This is the number of words the stack will hold, not the number of
bytes. For example, if each stack item is 32-bits, and this is set to 100,
then 400 bytes (100 * 32-bits) will be allocated. */
#define STACK_SIZE 200

/* Structure that will hold the TCB of the task being created. */
StaticTask_t xTaskBuffer;

/* Buffer that the task being created will use as its stack. Note this is
an array of StackType_t variables. The size of StackType_t is dependent on
the RTOS port. */
StackType_t xStack[ STACK_SIZE ];

/* Function that creates a task. */
void vOtherFunction( void )
{
    TaskHandle_t xHandle = NULL;

    /* Create the task without using any dynamic memory allocation. */
    xHandle = xTaskCreateStatic(
        vTaskCode,          /* Function that implements the task. */
        "NAME",            /* Text name for the task. */
        STACK_SIZE,         /* Number of indexes in the xStack array. */
        ( void * ) 1,       /* Parameter passed into the task. */
        tskIDLE_PRIORITY, /* Priority at which the task is created. */
        xStack,             /* Array to use as the task's stack. */
        &xTaskBuffer );   /* Variable to hold the task's data structure. */
}

```

```
/* Dimensions the buffer that the task being created will use as its stack.  
NOTE: This is the number of words the stack will hold, not the number of  
bytes. For example, if each stack item is 32-bits, and this is set to 100,  
then 400 bytes (100 * 32-bits) will be allocated. */  
#define STACK_SIZE 200  
  
/* Structure that will hold the TCB of the task being created. */  
StaticTask_t xTaskBuffer;  
  
/* Buffer that the task being created will use as its stack. Note this is  
an array of StackType_t variables. The size of StackType_t is dependent on  
the RTOS port. */  
StackType_t xStack[ STACK_SIZE ];  
  
/* Function that creates a task. */  
void vOtherFunction( void )  
{  
    TaskHandle_t xHandle = NULL;  
  
    /* Create the task without using any dynamic memory allocation. */  
    xHandle = xTaskCreateStatic(  
        vTaskCode,          /* Function that implements the task. */  
        "NAME",            /* Text name for the task. */  
        STACK_SIZE,         /* Number of indexes in the xStack array. */  
        ( void * ) 1,       /* Parameter passed into the task. */  
        tskIDLE_PRIORITY, /* Priority at which the task is created. */  
        xStack,             /* Array to use as the task's stack. */  
        &xTaskBuffer );   /* Variable to hold the task's data structure. */  
}
```

```
/* Allocate space for the TCB. Where the memory comes from  
depends on the implementation of the port malloc function and  
whether or not static allocation is being used. */  
pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof( TCB_t ) );  
  
if( pxNewTCB != NULL )  
{  
/* Allocate space for the stack used by the task being created.  
The base of the stack memory stored in the TCB so the task can  
be deleted later if required. */  
  
pxNewTCB->pxStack = ( StackType_t * ) pvPortMalloc( ( ( ( size_t )  
usStackDepth ) * sizeof( StackType_t ) ) );
```

FreeRTOS Heap management Schemes

heap_1.c	heap_2.c	heap_3.c	heap_4.c	heap_5.c	Your_own _mem.c
pvPortMalloc() vPortFree()	pvPortMalloc() vPortFree()	pvPortMalloc() vPortFree()	pvPortMalloc() vPortFree()	pvPortMalloc() vPortFree()	pvPortMalloc() vPortFree()

Application uses any one of these Schemes according to its requirements

FreeRTOS APIs and
Applications

Overview of FreeRTOS Synchronization & Mutual exclusion services

Synchronization in computing ?

Synchronization (computer science)

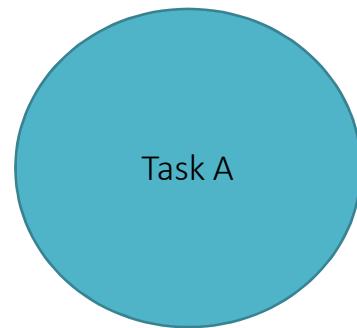
From Wikipedia, the free encyclopedia



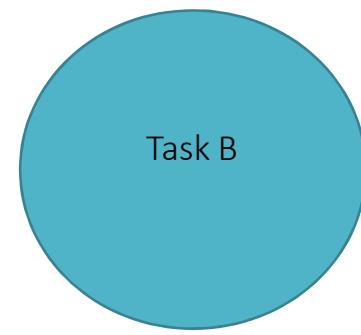
This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. (November 2014) ([Learn how and when to remove this template message](#))

In computer science, **synchronization** refers to one of two distinct but related concepts: synchronization of [processes](#), and synchronization of [data](#). *Process synchronization* refers to the idea that multiple processes are to join up or [handshake](#) at a certain point, in order to reach an agreement or commit to a certain sequence of action. *Data synchronization* refers to the idea of keeping multiple copies of a dataset in coherence with one another, or to maintain [data integrity](#). Process synchronization primitives are commonly used to implement data synchronization.^[1]

Synchronization between Tasks

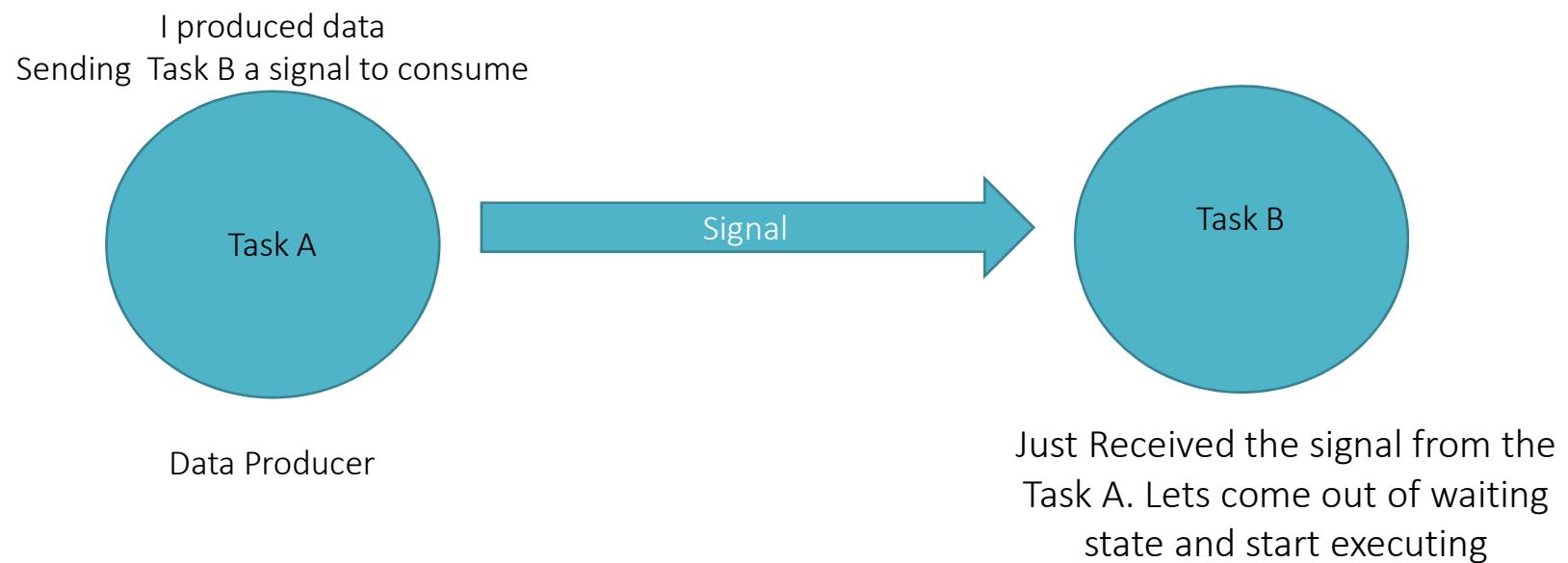


Data Producer



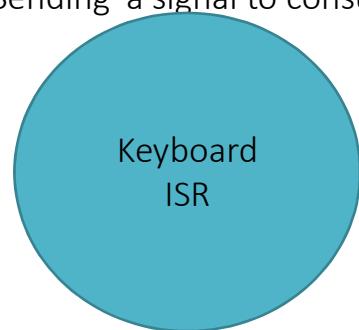
I need some data to consume .
But waiting for **Task A** to produce
some data.

Synchronization between Tasks

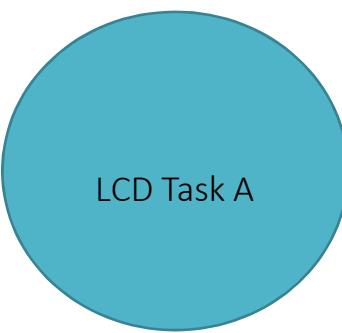


Synchronization between Task and Interrupt

I filled the queue with some data
. Sending a signal to consume data



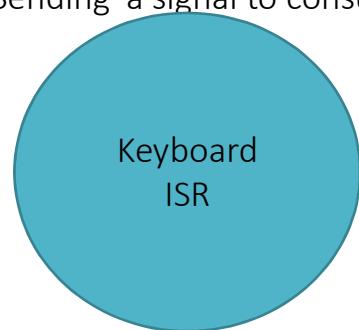
Data Producer



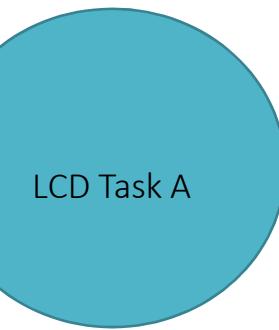
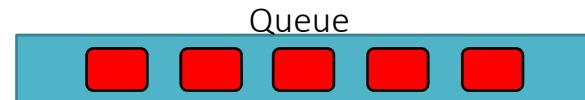
Received signal to wakeup
Looks like Queue has some data, lets
wakeup and display it on LCD

Synchronization between Task and Interrupt

I filled the queue with some data
. Sending a signal to consume data



Data Producer



Received signal to wakeup
Looks like Queue has some data, lets
wakeup and display it on LCD

How to achieve this signaling ?

Events (or Event Flags)

Semaphores (Counting and binary)

Queues and Message Queues

Pipes

Mailboxes

Signals (UNIX like signals)

Mutex



All these software subsystems support signaling hence can be used in Synchronization purposes

You will Learn More :

- 1) How to use Binary semaphore?
- 2) How to use Counting semaphore ?
- 3) How to synchronize between tasks ?
- 4) How to synchronize between a task and interrupt ?
- 5) How to use queues for synchronizations ?
- 6) Code examples.

Mutual Exclusion Services of FreeRTOS

Mutual exclusion

means that only a single thread should be able to access the shared resource at any given point of time. This avoids the race conditions between threads acquiring the resource. Usually you have to lock that resource before using and unlock it after you finish accessing the resource.

Synchronization

means that you synchronize/order the access of multiple threads to the shared resource.

```
int counter = 0;
ptread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void func(void *arg)
{
    int val;

    Pthread_mutex_lock( &mutex );
    val = counter;
    counter = val + 1;
    Pthread_mutex_unlock( &mutex );

    return NULL;
}
```

This code acts on shared item “Counter”
So needs protection

Mutual Exclusion Services of FreeRTOS

Mutex (Very powerful)

Binary Semaphore

```
int counter = 0;
ptread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void func(void *arg)
{
    int val;
    Pthread_mutex_lock( &mutex );
    val = counter;
    counter = val + 1;
    Pthread_mutex_unlock( &mutex );
    return NULL;
}
```

You will Learn More :

- 1) How to use Mutex ?
- 2) How to use binary Semaphore
- 3) Difference between Mutex and binary semaphore
- 4) Code examples.

FreeRTOS Coding Style

COPYRIGHT © BHARATI SOFTWARE 2016.

Variables Convention

COPYRIGHT © BHARATI SOFTWARE 2016.

Variables Convention

Variables of type '*unsigned long*' are prefixed with '*ul*', where the 'u' denotes '*unsigned*' and the 'l' denotes '*long*'.

Variables of type '*unsigned short*' are prefixed with '*us*', where the 'u' denotes '*unsigned*' and the 's' denotes '*short*'

Variables of type '*unsigned char*' are prefixed with '*uc*', where the 'u' denotes '*unsigned*' and the 'c' denotes '*char*'.

Variables of **non stdint** types are prefixed with '*x*'

Unsigned variables of non stdint types have an additional prefix '*u*'

Enumerated variables are prefixed with '*e*'

Variables Convention

Pointers have an additional prefix '*p*', for example a pointer to a uint16_t will have prefix '*pus*'.

In line with MISRA guides, unqualified standard '*char*' types are only permitted to hold ASCII characters and are prefixed with '*c*'.

In line with MISRA guides, variables of type '*char **' are only permitted to hold pointers to ASCII strings and are prefixed '*pc*'

Functions Convention

COPYRIGHT © BHARATI SOFTWARE 2016.

Functions Convention

API functions are prefixed with their return type, as per the convention defined for variables, with the addition of the prefix '*v*' for **void**.

API function names start with the name of the file in which they are defined.
For example vTaskDelete is defined in tasks.c, and has a void return type

File scope static (private) functions are prefixed with '*prv*'.

Macros

COPYRIGHT © BHARATI SOFTWARE 2016.

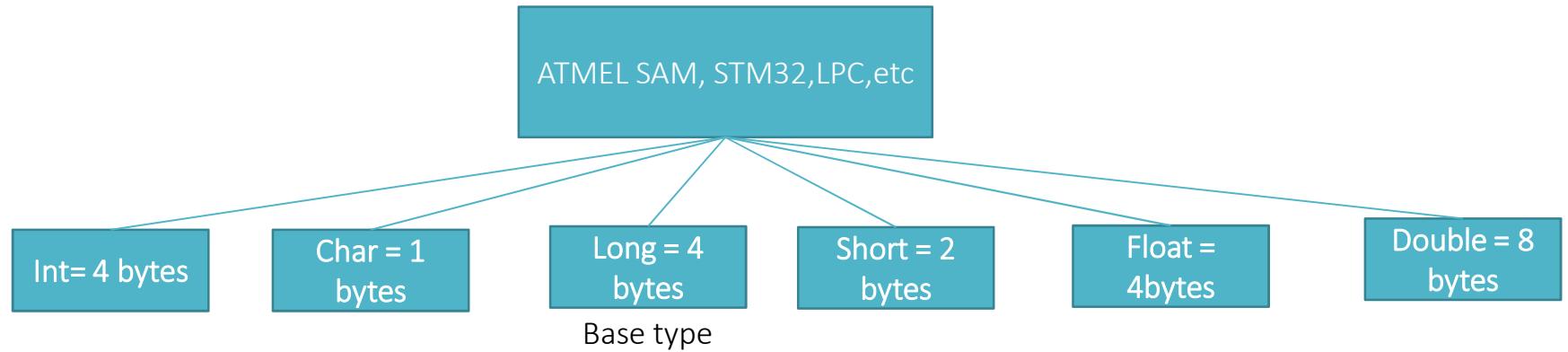
Macros

Macros are pre-fixed with the file in which they are defined. The pre-fix is lower case. For example, configUSE_PREEMPTION is defined in FreeRTOSConfig.h.

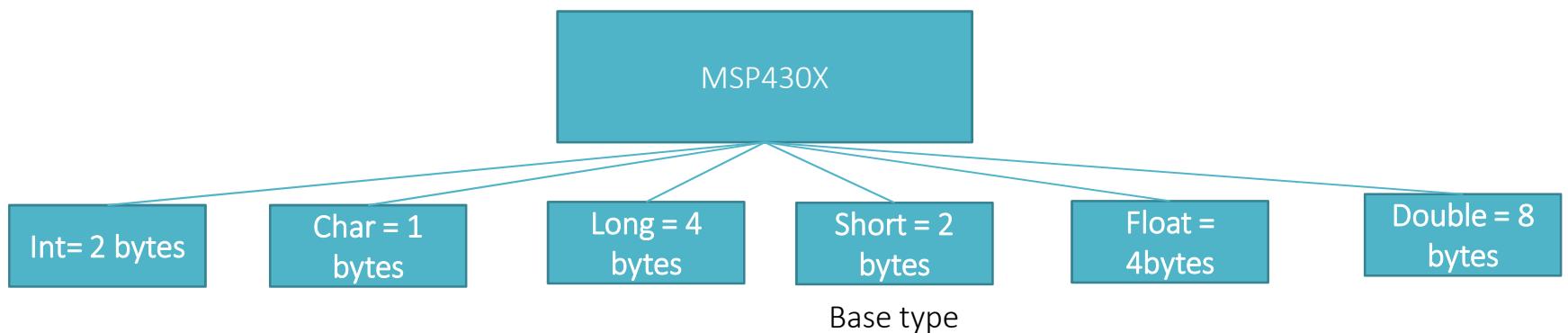
Other than the pre-fix, macros are written in all upper case, and use an underscore to separate words.

Data Types

ARM Cortex M



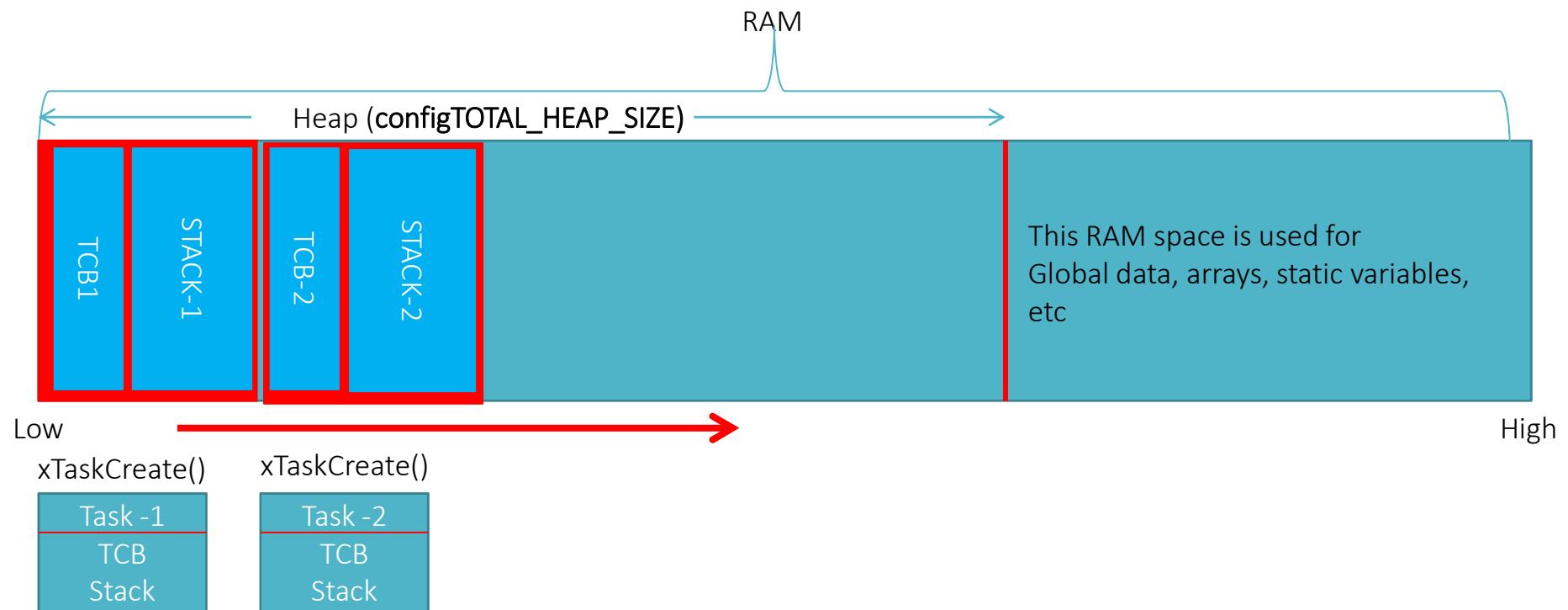
MSP430X



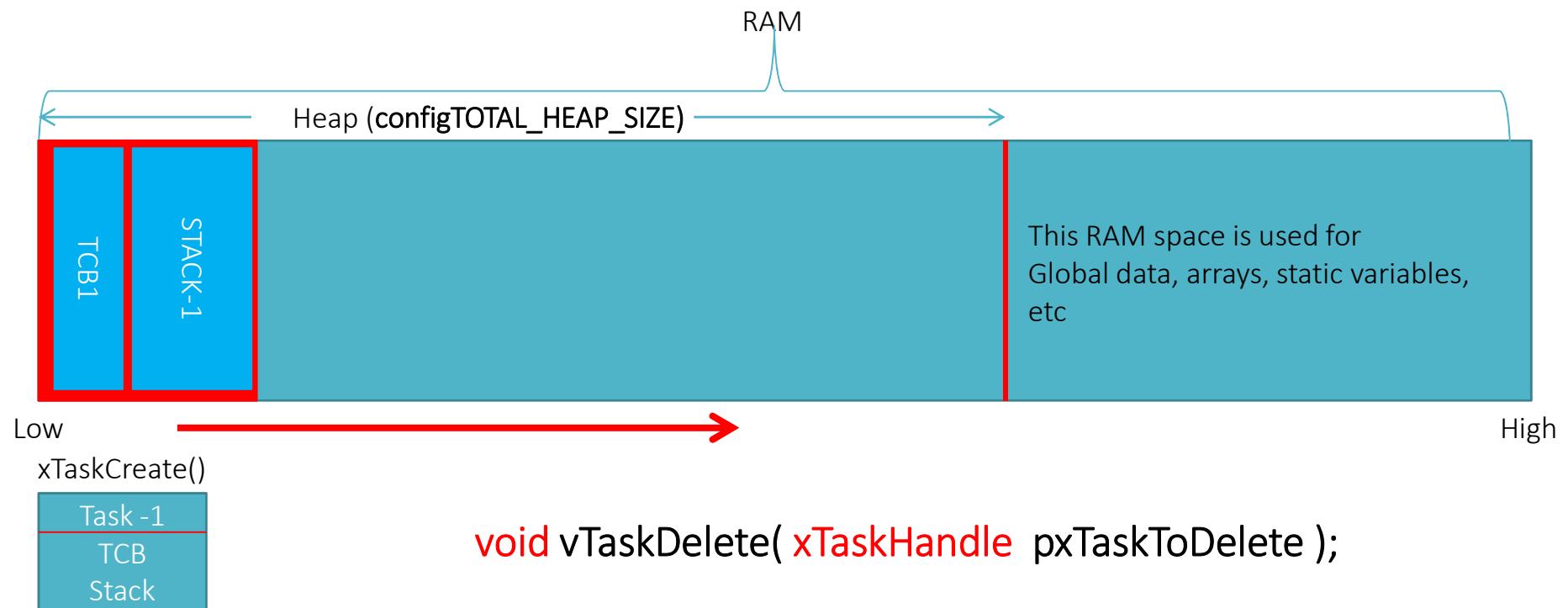
Deleting a Task

COPYRIGHT © BHARATI SOFTWARE 2016.

Deleting a Task



Deleting a Task



Deleting a Task

```
void ATaskFunction( void *pvParameters )
{
    /* Variables can be declared just as per a normal function.  Each instance
       of a task created using this function will have its own copy of the
       iVariableExample variable.  This would not be true if the variable was
       declared static - in which case only one copy of the variable would exist
       and this copy would be shared by each created instance of the task. */
    int iVariableExample = 0;

    /* A task will normally be implemented as in infinite loop. */
    for( ;; )
    {
        /* The code to implement the task functionality will go here. */
    }

    /* Should the task implementation ever break out of the above loop
       then the task must be deleted before reaching the end of this function.
       The NULL parameter passed to the vTaskDelete() function indicates that
       the task to be deleted is the calling (this) task. */
    vTaskDelete( NULL );
}
```

Exercise

Write an application which launches 2 tasks **task1** and **task2**.

task1 priority = 1

task2 priority = 2

task 2 should toggle the LED for every 1 sec and should delete itself when button is pressed by the user.

task1 should toggle the same led for every 200ms.

FreeRTOS Hardware Interrupt Configuration Items

COPYRIGHT © BHARATI SOFTWARE 2016.

FreeRTOS Hardware Interrupt Configuration Items

`configKERNEL_INTERRUPT_PRIORITY`

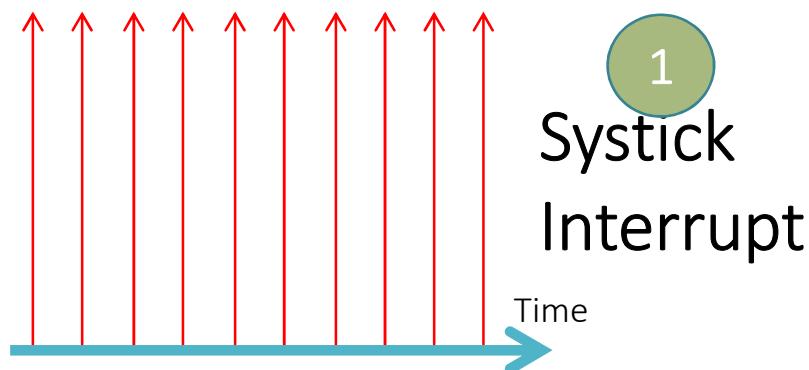
`configMAX_SYSCALL_INTERRUPT_PRIORITY`

configKERNEL_INTERRUPT_PRIORITY

This config item, decides the priority for the kernel Interrupts

What are the kernel interrupts ?

```
#define configTICK_RATE_HZ      ( (portTickType)1000)
```



1
Systick
Interrupt

2
PendSV interrupt

3
SVC interrupt

configKERNEL_INTERRUPT_PRIORITY

What's the lowest priority possible in My MCU which is based on ARM Cortex M PROCESSOR ?

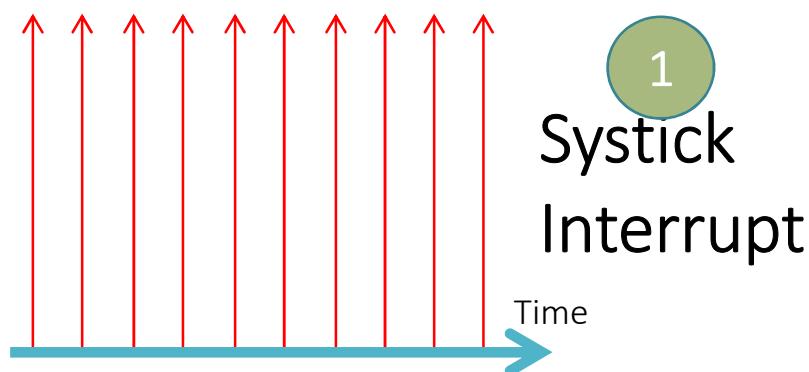
What is the value of __NVIC_PRIO_BITS macro ?

✓ configKERNEL_INTERRUPT_PRIORITY

This config item, decides the priority for kernel Interrupts

What are the kernel interrupts ?

```
#define configTICK_RATE_HZ      ( (portTickType)1000)
```



1
Systick
Interrupt

2
PendSV interrupt

3
SVC interrupt

configMAX_SYSCALL_INTERRUPT_PRIORITY

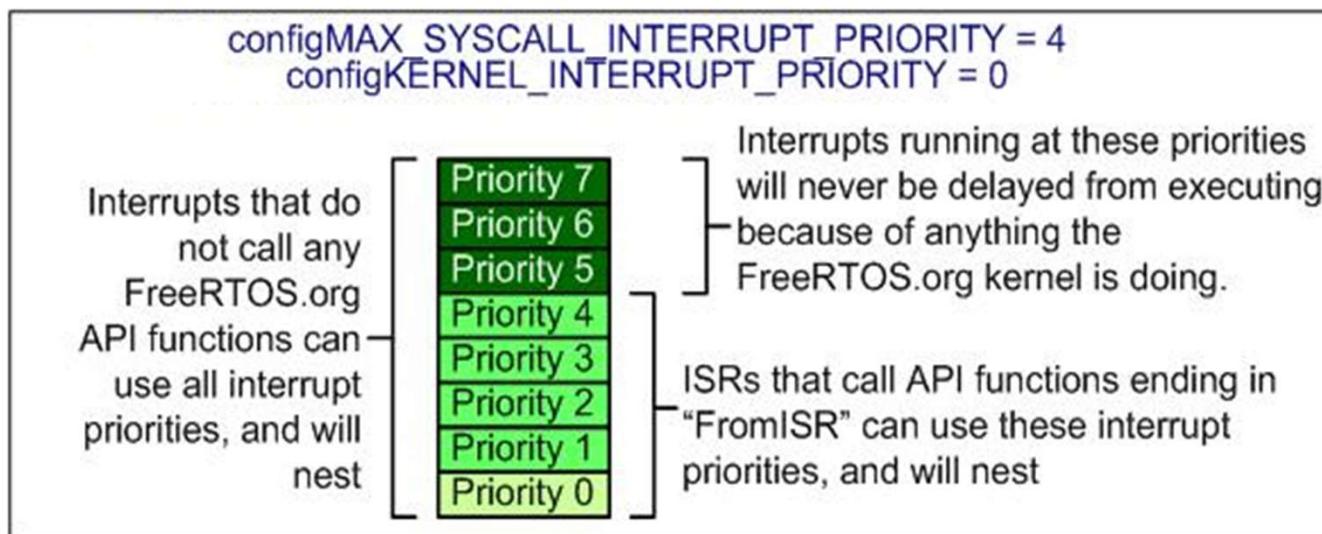
In the newer version of FreeRTOS Port file, its name is changed to
configMAX_API_CALL_INTERRUPT_PRIORITY

This is a threshold priority limit for those interrupts which use freeRTOS APIs which end with “FromISR”

Interrupts which use freeRTOS APIs ending with “FromISR”, should not use priority greater than this value.

Greater priority = less in numeric value

configKERNEL_INTERRUPT_PRIORITY & configMAX_SYSCALL_INTERRUPT_PRIORITY



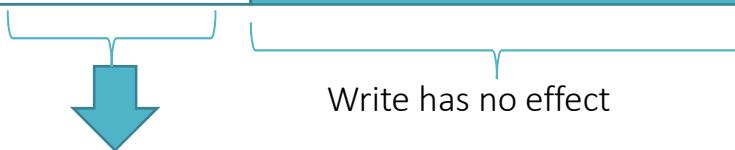
Interrupts that do not call API functions can execute at priorities above configMAX_SYSCALL_INTERRUPT_PRIORITY and therefore never be delayed by the RTOS kernel execution

Priority Register

Microcontroller Vendor XXX

TM4C123G

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented				Not implemented			



8 Levels of Priority level

0x00,0x20,0x40,0x60,
0x80,0xA0,0xC0, 0xE0

Microcontroller Vendor YYY

STM32F4xx

AT91SAM3X8E

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented				Not implemented			



16 Levels of Priority level

0x00,0x10,0x20,0x30,0x40,0x50,
0x60,0x70,0x80,0x90,0xa0,0xb0,0xc0,0xd0,0xe0,0xf0

nted

ffect

Implemented

Not implemented



16 Levels of Priority level

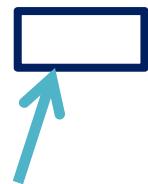


Highest Priority

0x00, 0x10, 0x20, 0x30, 0x40, 0x50,

0x60, 0x70, 0x80, 0x90, 0xa0, 0xb0, 0xc0, 0xd0, 0xe0, 0xf0

Lowest Priority

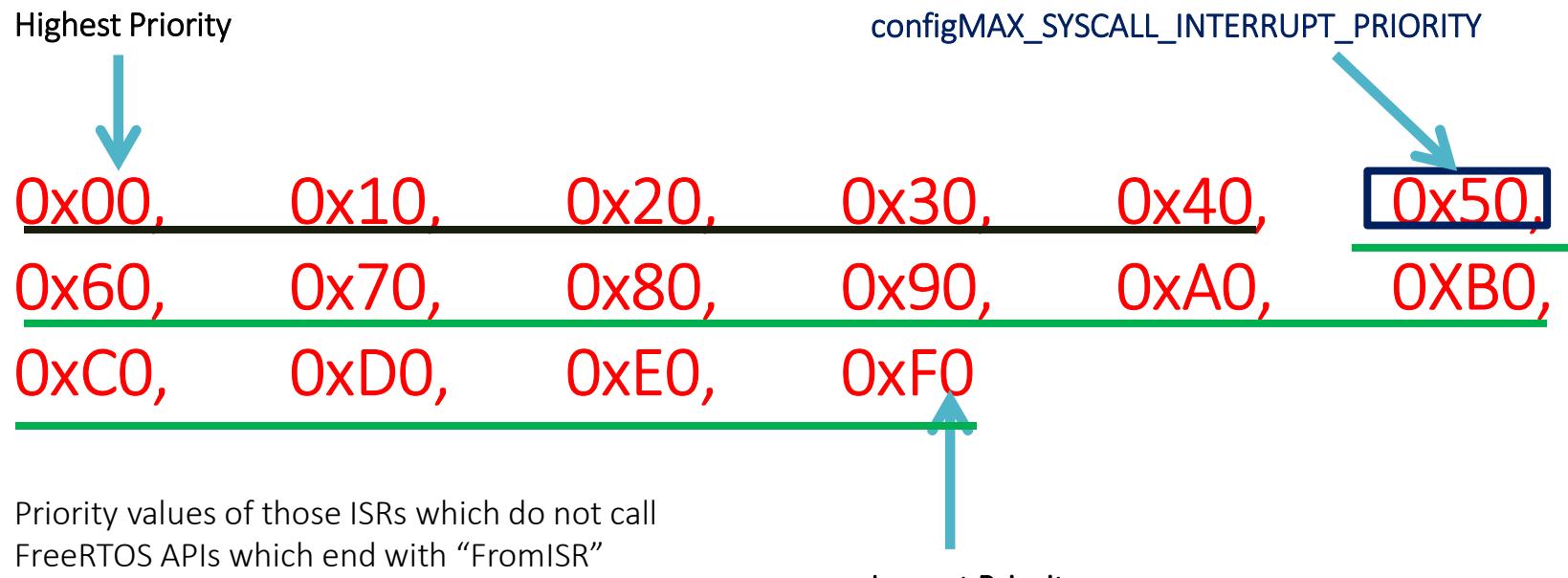


configMAX_SYSCALL_INTERRUPT_PRIORITY

Write has no effect

the interrupt service routine that uses an RTOS API function must have its priority manually set to a value that is numerically equal to or greater than this point.

The interrupt service routine that uses an RTOS API function must have its priority manually set to a value that is numerically equal to or **greater than** this point. Remember in ARM greater prio. Value lesser is the priority(urgency)



Concluding Points

FreeRTOS APIs that end in "**FromISR**" are interrupt safe, but even these APIs should not be called from ISRs that have priority(Urgency) above the priority defined by **configMAX_SYSCALL_INTERRUPT_PRIORITY**

Therefore, any interrupt service routine that uses an RTOS API function must have its priority value manually set to a value that is numerically equal to or **greater than configMAX_SYSCALL_INTERRUPT_PRIORITY** setting

Cortex-M interrupts default to having a priority value of zero. Zero is the highest possible priority value. Therefore, **never leave the priority of an interrupt that uses the interrupt safe RTOS API at its default value.**

Concluding Points

First we learnt there are 2 configuration items

`configKERNEL_INTERRUPT_PRIORITY`

`configMAX_SYSCALL_INTERRUPT_PRIORITY`

`configKERNEL_INTERRUPT_PRIORITY` : The kernel interrupt priority config item actually decides the priority level for the kernel related interrupts like systick, pendsv and svc and it is set to lowest interrupt priority as possible.

`configMAX_SYSCALL_INTERRUPT_PRIORITY` : The max sys call interrupt priority config item actually decides the maximum priority level , that is allowed to use for those interrupts which use freertos APIs ending with “Fromlsr” in their interrupt service routines.

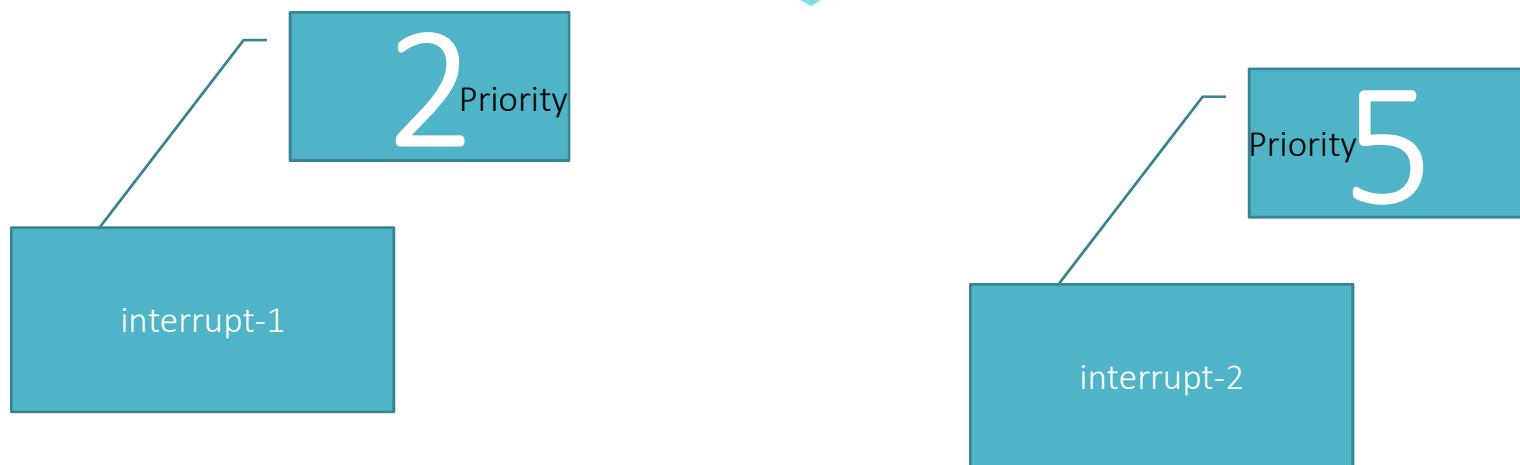
Priority of FreeRTOS Tasks

FreeRTOS Task Priority

Vs

Processor Interrupt/Exception Priority

Lower logical Priority means higher numeric priority value

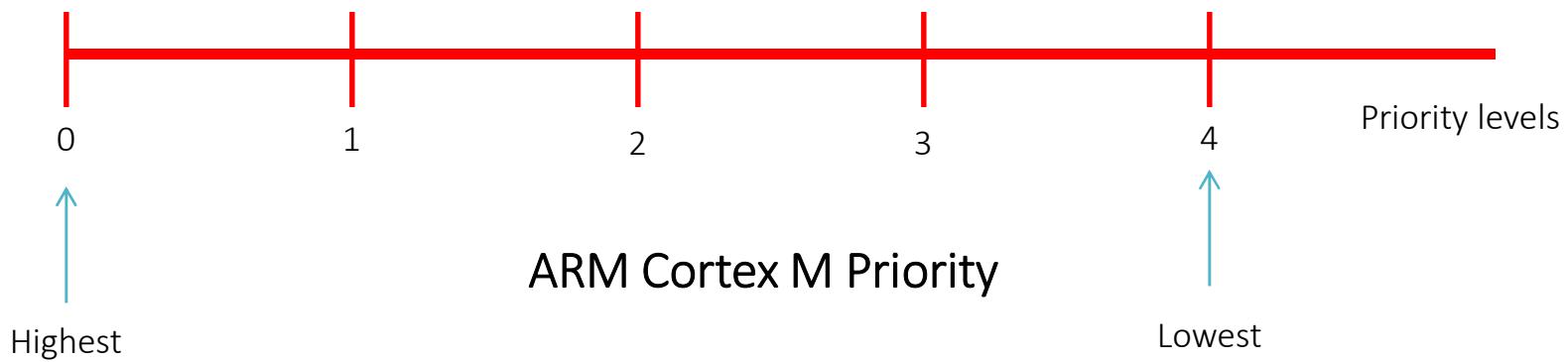
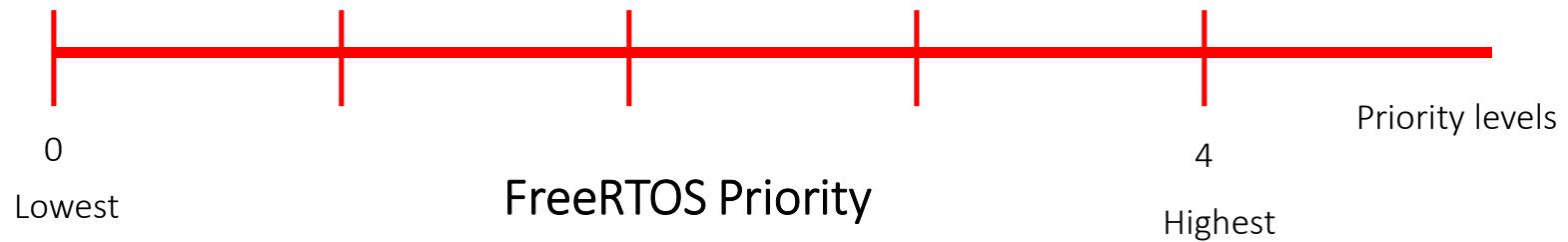


lower priority means higher numeric value





Interrupt-1 is higher priority than the interrupt-2



FreeRTOS Task Priority APIs

COPYRIGHT © BHARATI SOFTWARE 2016.

API to set Priority

```
void vTaskPrioritySet( xTaskHandle pxTask,  
                      unsigned portBASE_TYPE uxNewPriority );
```

API to Get Priority

```
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
```

Exercise

Write an application which creates 2 tasks

task 1 : Priority 2

task 2 : Priority 3

task 2 should toggle the LED at 1 sec duration and task 1 should toggle the led at 100ms duration.

When application receives button interrupt the priority must be reversed in side the task handlers.

Interrupt Safe and Interrupt Un-Safe APIs

Interrupt Un-Safe APIs

2 flavors of freeRTOS APIs

Non-interrupt safe APIs

Interrupt safe APIs

Interrupt Un-Safe APIs

FreeRTOS APIs which don't end with the word “**FromISR**” are called as interrupt unsafe APIs

e.g.

xTaskCreate(),
xQueueSend()
xQueueReceive()
etc

Interrupt Safe and Un-safe APIs

If you want to send a data item in to the queue from the ISR, then use `xQueueSendFromISR()` instead of `xQueueSend()`.

`xQueueSendFromISR()` is an interrupt safe version of `xQueueSend()`.

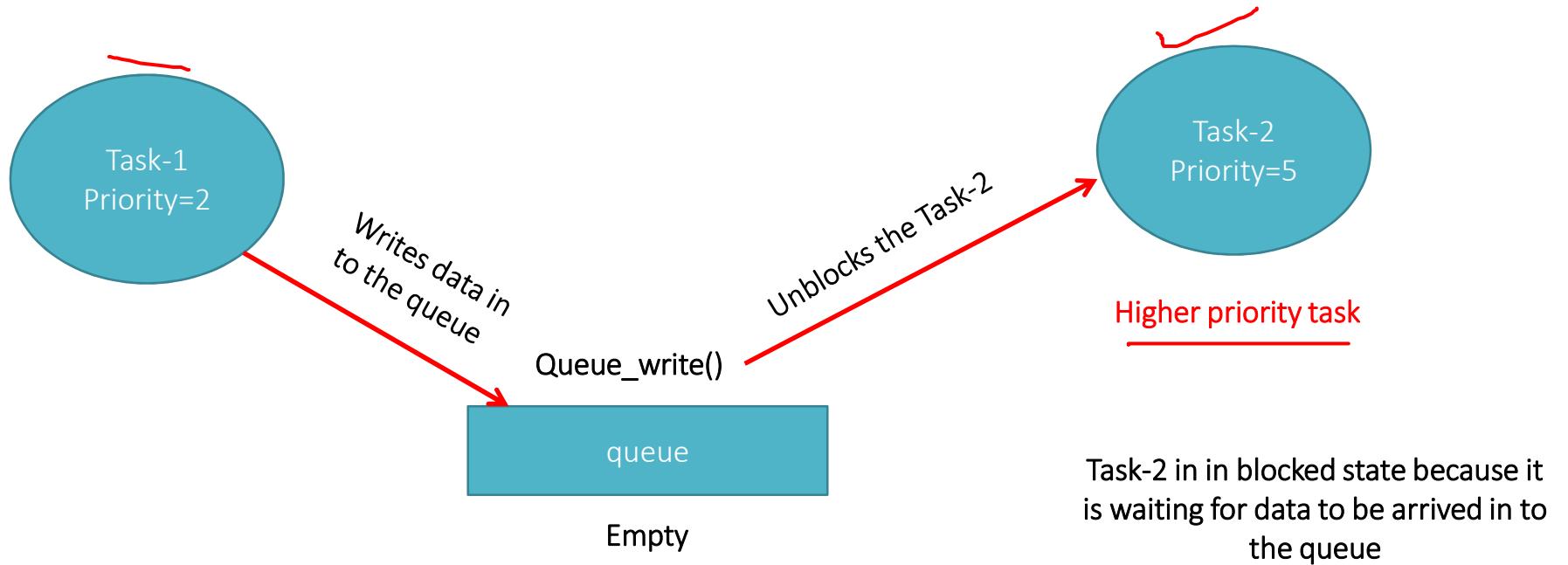
If you want to Read a data item from the queue being in the ISR, then use `xQueueReceiveFromISR()` instead of `xQueueReceive()`.

`xQueueReceiveFromISR()` is an interrupt safe version of `xQueueReceive()`.

`xSemaphoreTakeFromISR()` is an interrupt safe version of `xSemaphoreTake()` : which is used to ‘take’ the semaphore

`xSemaphoreGiveFromISR()` is an interrupt safe version of `xSemaphoreGive()` : which is used to ‘give’ the semaphore

Why separate interrupt Safe APIs?



```
Queue_write(QUEUE *qptr , void * data)
{
    1. write to queue
    2. does write to queue unblock any higher priority task ?
    3. if yes, must do taskYIELD()
    4. if no, continue with the same task 1
}
```

```
QUEUE some_queue;

Task1_fun(void *data)
{
    Queue_write(&some_queue, data);

    next statement 1;
    next statement 2;
    .
    .
    .

}
```

```
QUEUE some_queue;  
  
Task1_fun(void *data)  
{  
    Queue_write(&some_queue, data);  
  
    next statement 1;  
    next statement 2;  
    .  
    .  
    .  
}  
  
Queue_write(QUEUE *qptr, void * data)  
{  
    1. write to queue  
    2. does write to queue unblock any higher priority task ?  
    3. if yes, must do taskYIELD()  
    4. if no, continue with the same task 1  
}
```

Scenario of task calling FreeRTOS API

Why separate interrupt Safe APIs?

Ok ,That's fine but our goal is to understand why the same API **Queue_Write()** can not be called from an ISR ?? Why its ISR flavour **Queue_Write_FromISR()** must be used in FreeRTOS ??

```
QUEUE some_queue;  
  
ISR_Fun(void *data)  
{  
  
    Queue_write(&some_queue, data);  
  
    next statement 1;  
    next statement 2;  
    .  
    .  
    .  
}
```

Queue_write(QUEUE *qptr, void * data)
{

 1. write to queue

 2. does write to queue unblock any higher priority task ?

 3. if yes, must do **taskYIELD()**

 4. if no, continue with the same task 1

}

Scenario of an ISR calling non-interrupt safe API

```
Queue_write_FromISR (QUEUE *qptr , void * data, void *
xHigherPriorityTaskWoken)
{
    1. write to queue

    2. does write to queue unblocks any higher priority task ?

    3. if yes, then set xHigherPriorityTaskWoken = TRUE

    4. if no, then set xHigherPriorityTaskWoken = FALSE

    5. return to ISR
}
```

```
QUEUE some_queue;

ISR_Fun(void *data)
{
    unsigned long xHigherPriorityTaskWoken = FALSE;

    Queue_write_FromISR(&some_queue, data, & xHigherPriorityTaskWoken );

    next statement 1;
    next statement 2;
    .
    .
    .
/* yielding to task happens in ISR Context , no tin API context */
    if(xHigherPriorityTaskWoken )
        portYIELD()
}
```

COPYRIGHT © BHARATI SOFTWARE 2016.

```

QUEUE some_queue;

ISR_Fun(void *data)
{
    unsigned long xHigherPriorityTaskWoken = FALSE;

    Queue_write_FromISR(&some_queue, data, &xHigherPriorityTaskWoken);

    next statement 1;
    next statement 2;
    .
    .
    /* yielding to task happens in ISR Context , no tin API context */
    if(xHigherPriorityTaskWoken)
        portYIELD()
}

```

Queue_write_FromISR (QUEUE *aptr, void * data, void * xHigherPriorityTaskWoken)

1. write to queue
2. does writing to queue unblocks any higher priority task?
3. if yes, then set xHigherPriorityTaskWoken = TRUE
4. if no, then set xHigherPriorityTaskWoken = FALSE
5. return to ISR

Scenario of an ISR calling interrupt safe API

Interrupt Safe APIs: Conclusion

Whenever you want use FreeRTOS API from an ISR its ISR version must be used, which ends with the word “FromISR”

This is for the reason, Being in the interrupt Context (i.e being in the middle of servicing the ISR) you can not return to Task Context (i.e making a task to run by pre-empting the ISR)

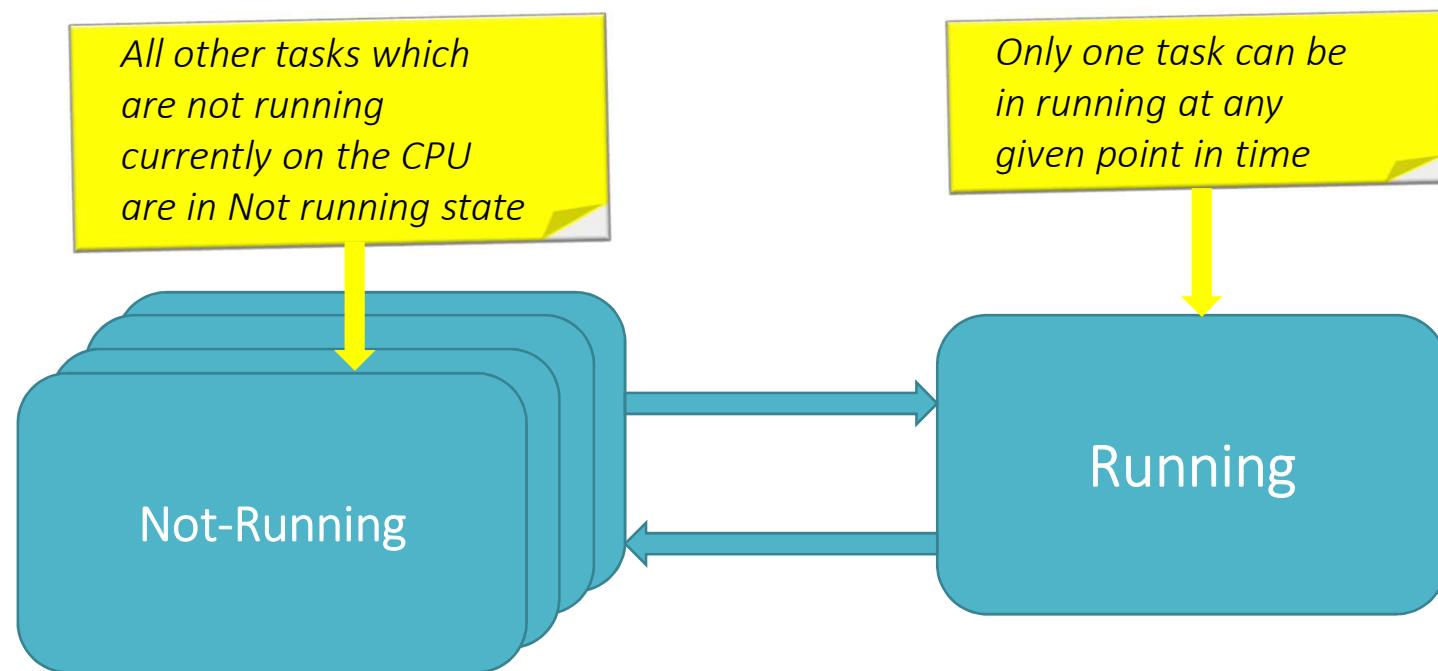
In ARM cortex M based Processors usage exception will be raised by the processor if you return to the task context by preempting ISR .

FreeRTOS Task States

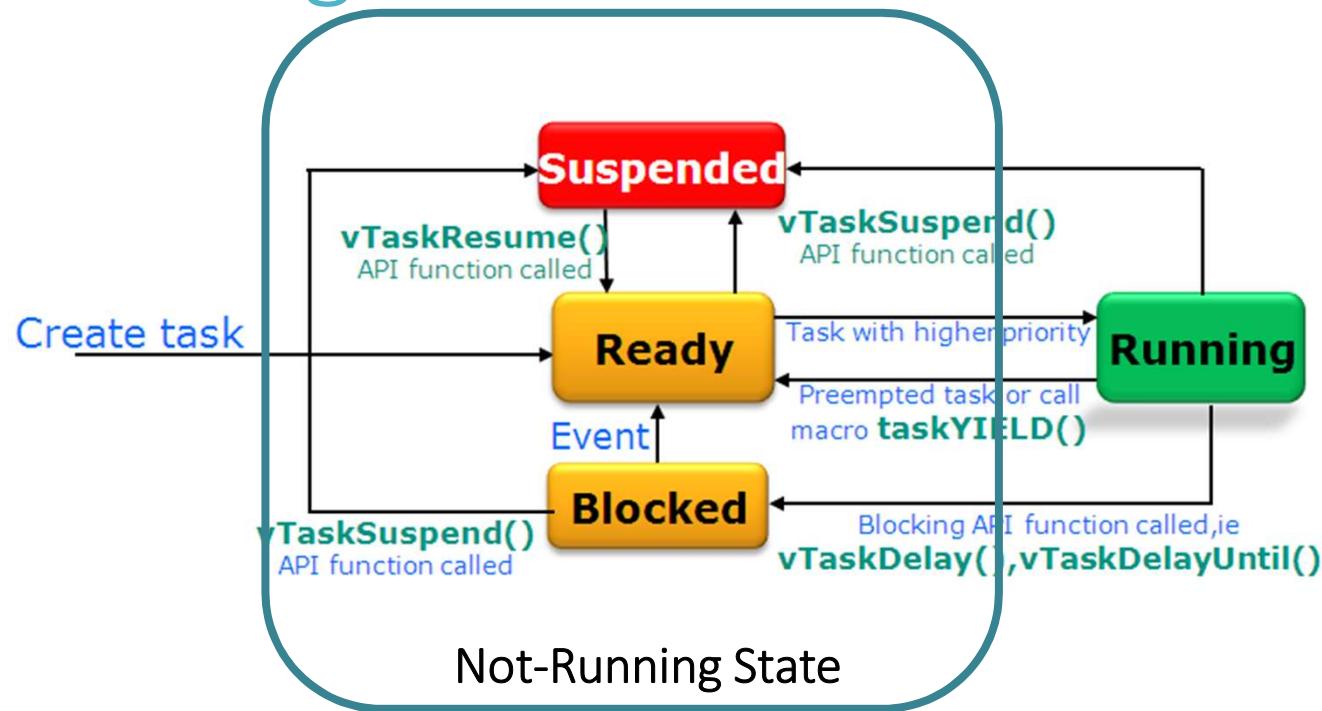
Top Level Task States

Running and Not-Running State of a Task

Top Level Task States- Simplistic Model



Not-Running State



The Blocked state

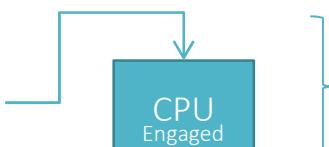
What is blocking of a Task ?



A Task which is Temporarily or permanently chosen not to run on CPU

Generate delay of ~ 10ms

```
for ( int i=0 ; i < 5000 ; i++ );  
      runs for 10ms
```



This code runs on CPU continuously for 10ms, Starving other lower priority tasks. Never use such delay implementations

vTaskDelay(10)

Not runs for 10ms



This is blocking delay API which blocks the task for 10ms. That means for the next 10ms other lower priority tasks can run. After 10ms the task will wake up

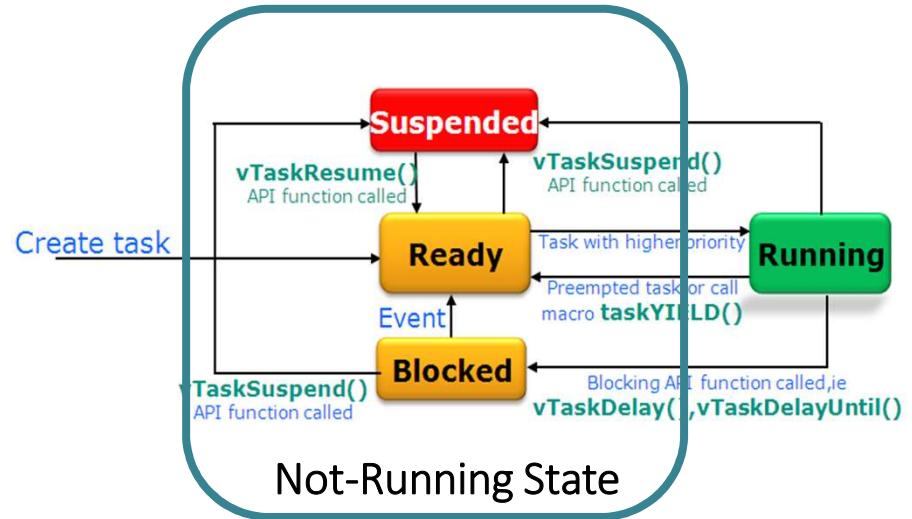
Advantages of blocking:

1. To implement the blocking Delay – For example a task may enter the Blocked state to wait for 10 milliseconds to pass.
2. For Synchronization –For example, a task may enter the Blocked state to wait for data to arrive on a queue. When the another task or interrupt fills up the queue , the blocked task will be unblocked.

FreeRTOS queues, binary semaphores, counting semaphores, recursive semaphores and mutexes can all be used to implement synchronization and thus they support blocking of task.

Blocking Delay APIs

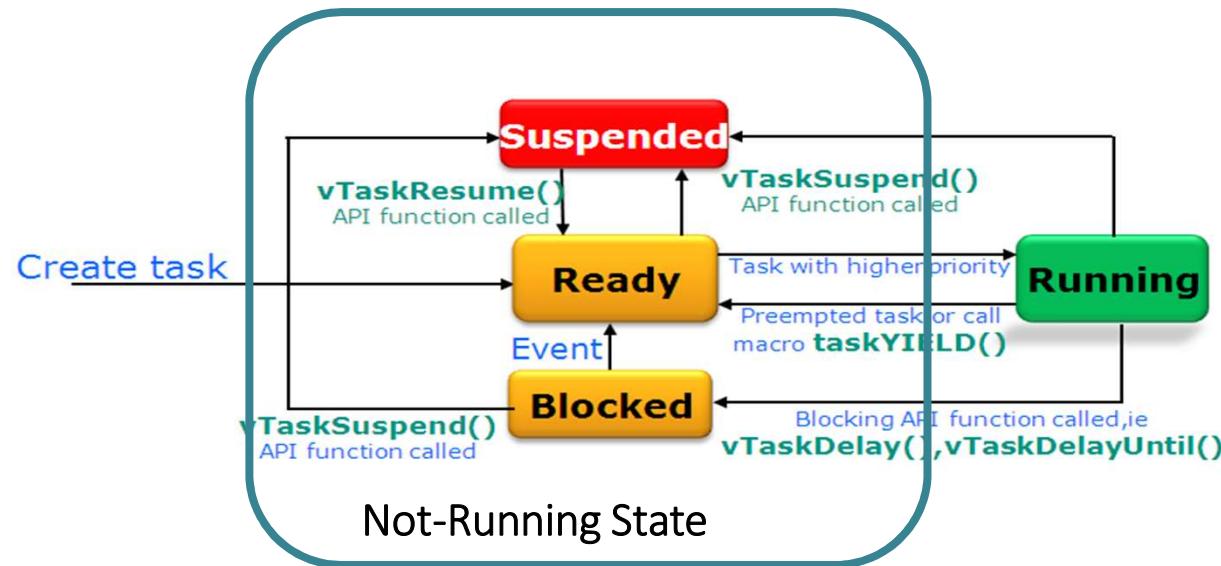
vTaskDelay()
vTaskDelayUntil()



All these kernel objects support APIs which can block a task during operation , which we will explore later in their corresponding sections

The Suspended state

The Suspended state



```
void vAFunction( void )
{
TaskHandle_t xHandle;

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

    // ...

    // Use the handle to suspend the created task.
    vTaskSuspend( xHandle );

    // ...

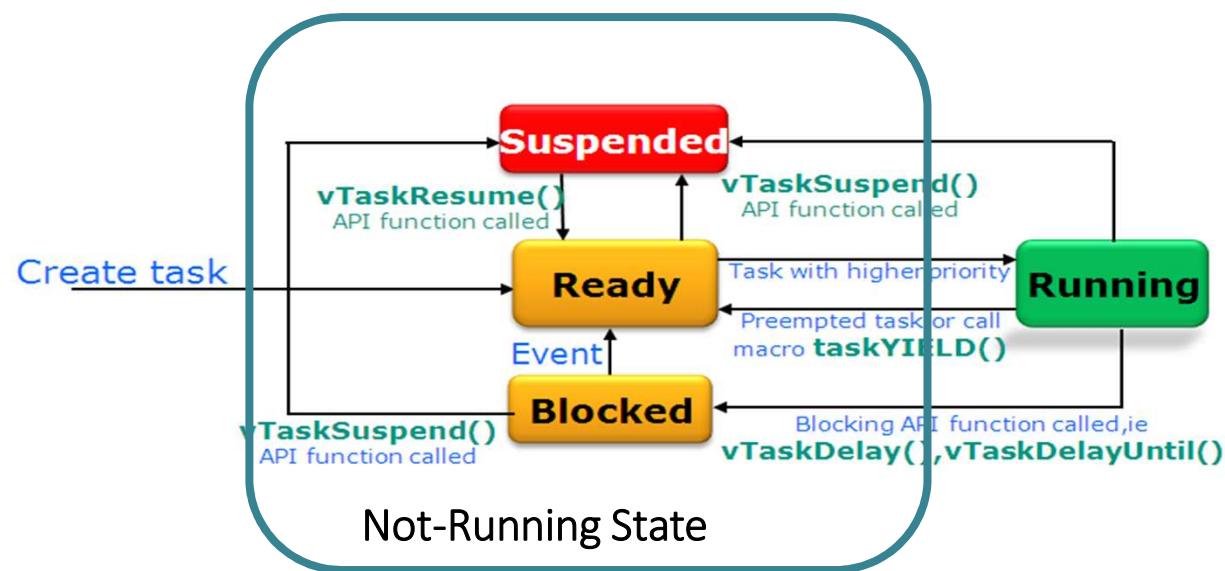
    // The created task will not run during this period, unless
    // another task calls vTaskResume( xHandle ).

    //...
    // Suspend ourselves.
    vTaskSuspend( NULL );
}

    // We cannot get here unless another task calls vTaskResume
    // with our handle as the parameter.
}
```

The Ready state

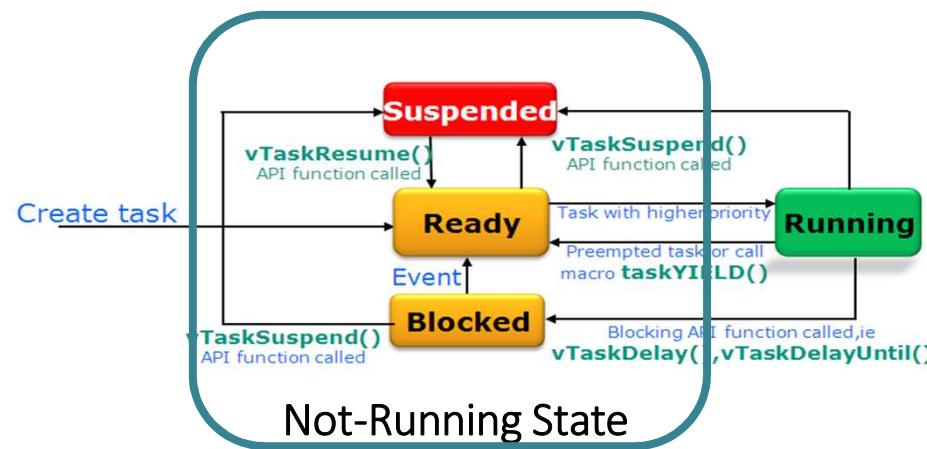
The Ready state



Conclusion

COPYRIGHT © BHARATI SOFTWARE 2016.

Task States : Conclusion



FreeRTOS : Importance of delay

Crude delay Implementation

Crude delay Implementation

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile unsigned long ul;

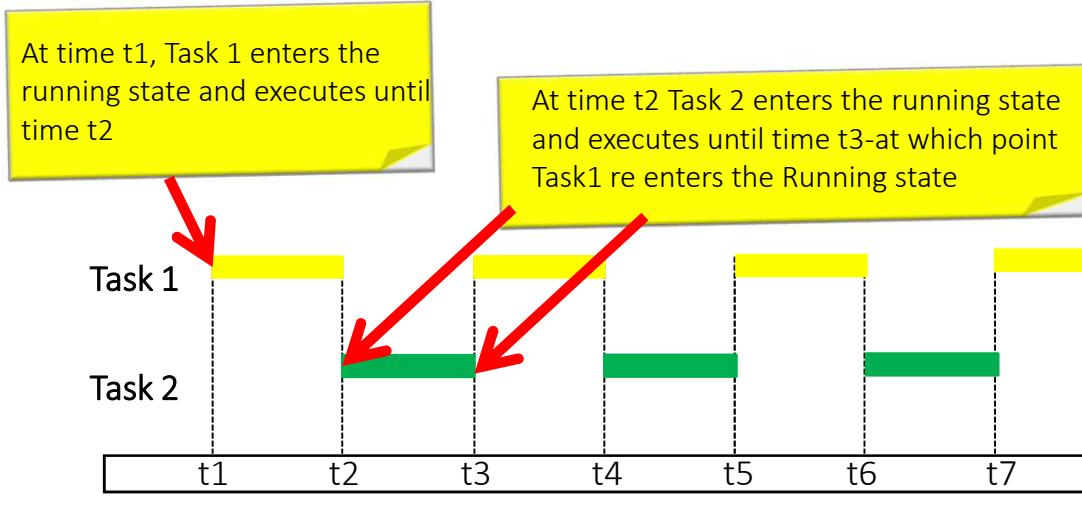
/* As per most tasks, this task is implemented in an infinite loop. */
for( ;; )
{
    /* Print out the name of this task. */
    vPrintString( pcTaskName );

    /* Delay for a period. */
    for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
    {
        /* This loop is just a very crude delay implementation. There is
        nothing to do in here. Later examples will replace this crude
        loop with a proper delay/sleep function. */
    }
}

void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\r\n";
volatile unsigned long ul;

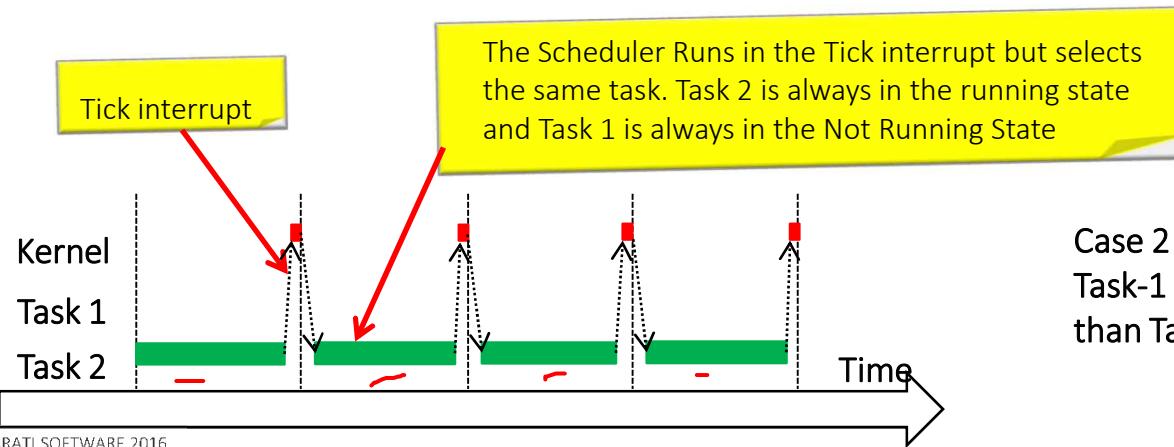
/* As per most tasks, this task is implemented in an infinite loop. */
for( ;; )
{
    /* Print out the name of this task. */
    vPrintString( pcTaskName );

    /* Delay for a period. */
    for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
    {
        /* This loop is just a very crude delay implementation. There is
        nothing to do in here. Later examples will replace this crude
        loop with a proper delay/sleep function. */
    }
}
```



Case 1 :
Task 1 and Task-2 having same priorities

$$P_{T_1} = P_{T_2}$$



Case 2 :
Task-1 is having lower priority than Task-2

FreeRTOS Blocking Delay APIs

```
void vTaskDelay( portTickType xTicksToDelay );  
void vTaskDelayUntil( portTickType xTicksToDelay );
```

Using the Blocking state to Create a delay

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile unsigned long ul;

/* As per most tasks, this task is implemented in an infinite loop. */
for( ;; )
{
    /* Print out the name of this task. */
    vPrintString( pcTaskName );

    /* Delay for a period. */
    for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
    {
        /* This loop is just a very crude delay implementation. There is
           nothing to do in here. Later examples will replace this crude
           loop with a proper delay/sleep function. */
    }
}
}
```

```
void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\r\n";
volatile unsigned long ul;

/* As per most tasks, this task is implemented in an infinite loop. */
for( ;; )
{
    /* Print out the name of this task. */
    vPrintString( pcTaskName );

    /* Delay for a period. */
    for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
    {
        /* This loop is just a very crude delay implementation. There is
           nothing to do in here. Later examples will replace this crude
           loop with a proper delay/sleep function. */
    }
}
}
```

FreeRTOS Blocking Delay APIs

```
void vTaskDelay ( portTickType xTicksToDelay );
```

```
void vTaskDelayUntil (TickType_t *pxPreviousWakeTime,  
                      const TickType_t xTimeIncrement );
```

Conclusion

Never use *for loop* based delay implementation , which doesn't do any genuine work but still consumes the CPU .

Using *for loop* for delay implementation may also prevent any lower priority task to take over the CPU during the delay period.

vTaskDelay()

```
void vTaskDelay( portTickType xTicksToDelay );
```

vTaskDelay()

Example usage:

```
void vTaskFunction( void * pvParameters )
{
    /* Block for 500ms. */
    const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

    for( ;; )
    {
        /* Simply toggle the LED every 500ms, blocking between each toggle. */
        vToggleLED();
        vTaskDelay( xDelay );
    }
}
```

void vTaskDelay(portTickType xTicksToDelay);

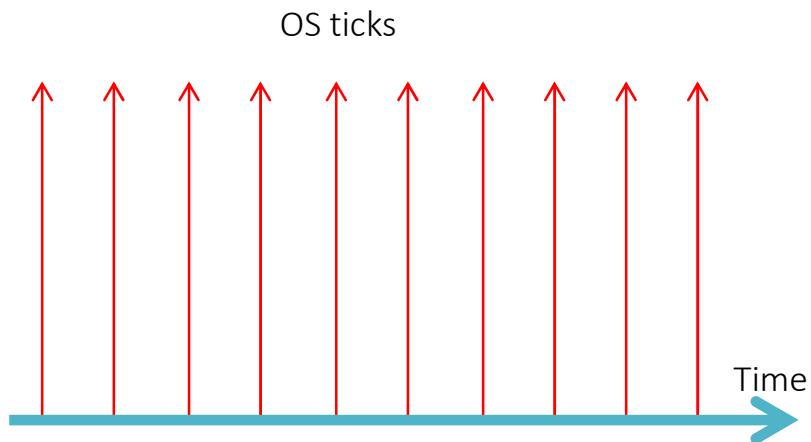
vTaskDelay()

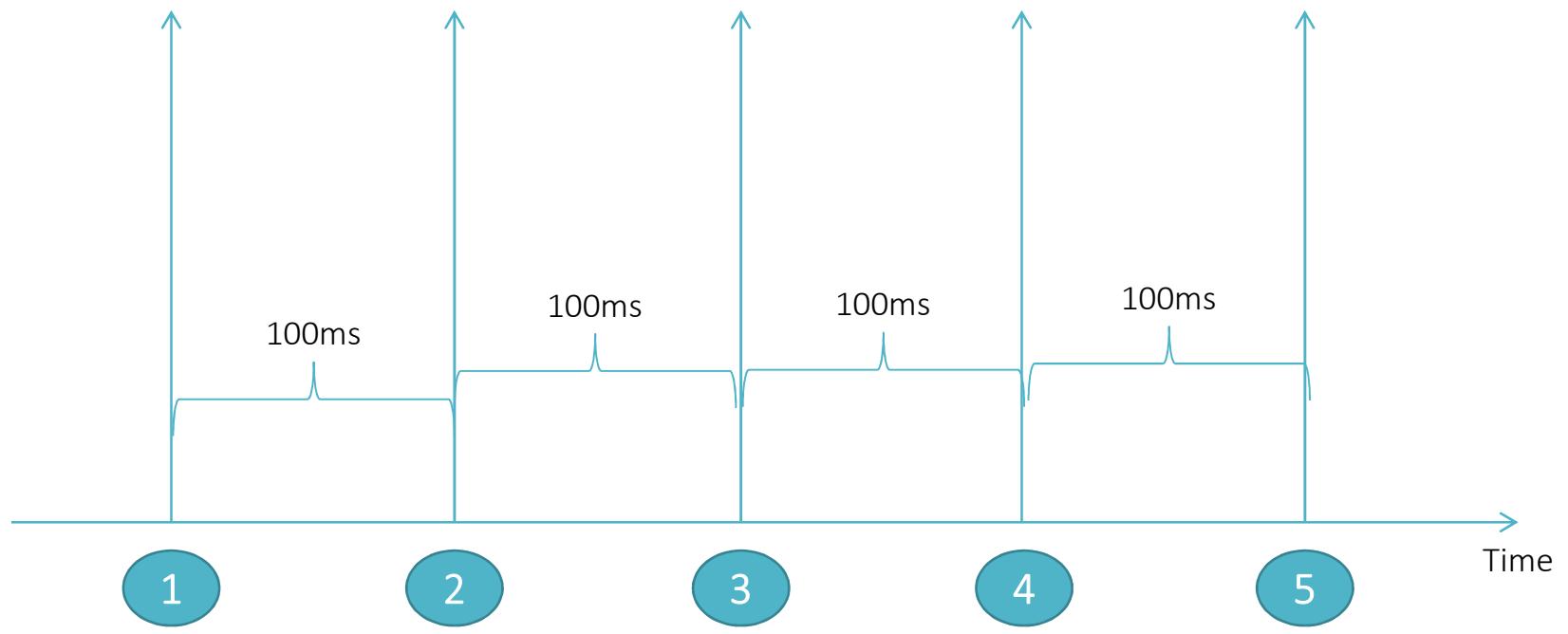
Example usage:

```
void vTaskFunction( void * pvParameters )
{
    /* Block for 500ms. */
    const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

    for( ;; )
    {
        /* Simply toggle the LED every 500ms, blocking between each toggle. */
        vToggleLED();
        vTaskDelay( xDelay );
    }
}
```

void vTaskDelay(portTickType xTicksToDelay);



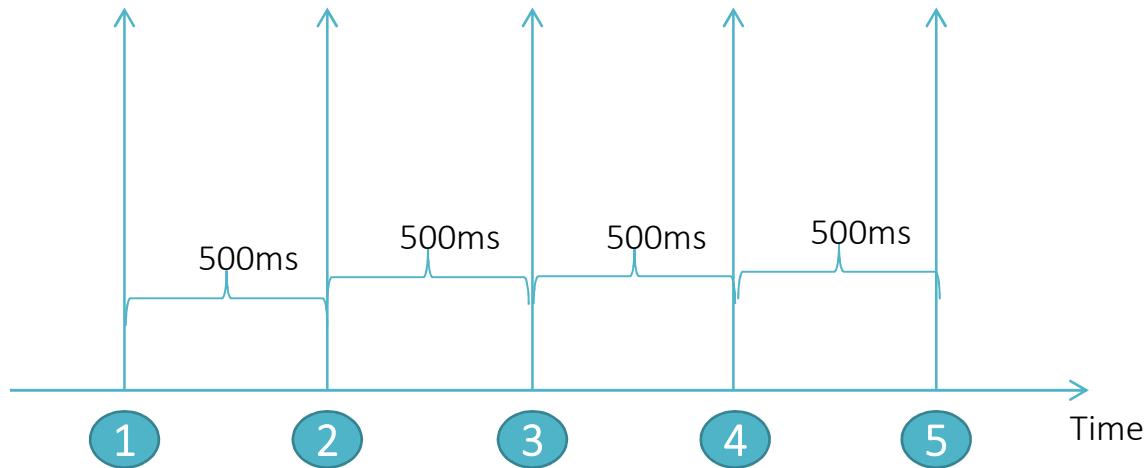


Task 1 executing periodically for every 100ms

Example usage:

```
void vTaskFunction( void * pvParameters )
{
    /* Block for 500ms. */
    const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

    for( ;; )
    {
        /* Simply toggle the LED every 500ms, blocking between each toggle. */
        vToggleLED();
        vTaskDelay( xDelay );
    }
}
```



Task executing periodically for every 500ms

Example usage:

```
void vTaskFunction( void * pvParameters )
{
    /* Block for 500ms. */
    const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

    for( ;; )
    {
        /* Simply toggle the LED every 500ms, blocking between each toggle. */
        vToggleLED();
        vTaskDelay( xDelay );
    }
}
```

COPYRIGHT © BHARATI SOFTWARE 2016.

Task 1 prints status of the LED over UART, then it too enters the Blocked state by calling vTaskDelay(1000)

2

When the delay expires the scheduler moves the tasks back into the ready state, where both execute again before once again calling vTaskDelay() causing them to re-enter the blocked state. Task-2 executes first as it has the higher Priority

4

Task 1

Task 2

Idle

1

Task 2 has the highest priority so runs first. It toggles the LED then calls vTaskDelay(1000) and in so doing enters the blocked state, permitting the lower priority Task 1 to execute.

1000ms

1000ms

tn

Time

3

At this point both applications tasks are in the blocked state-so the idle task runs

FreeRTOS Hook Functions

COPYRIGHT © BHARATI SOFTWARE 2016.

Idle Task hook function

Idle task hook function implements a callback from idle task to your application

You have to enable the idle task hook function feature by setting this config item
`configUSE_IDLE_HOOK` to 1 within `FreeRTOSConfig.h`

Then implement the below function in your application

```
void vApplicationIdleHook( void );
```

That's it , whenever idle task is allowed to run, your hook function will get called, where you can do some useful stuffs like sending the MCU to lower mode to save power

FreeRTOS Hook Functions

- ✓ Idle task hook function
- ✓ RTOS Tick hook function
- ✓ Dynamic memory allocation failed hook function (**Malloc Failed Hook Function**)
- ✓ Stack over flow hook function

These hook functions you can implement in your application code if required

The FreeTOS Kernel will call these hook functions whenever corresponding events happen.

FreeRTOS Hook Functions

Idle task hook function

`configUSE_IDLE_HOOK` should be 1 in `FreeRTOSConfig.h`

and your application source file (`main.c`) should implement the below function

```
void vApplicationIdleHook( void )
{
}
```

FreeRTOS Hook Functions

RTOS Tick hook function

configUSE_TICK_HOOK should be 1 in FreeRTOSConfig.h

and your application source file (main.c) should implement the below function

```
void vApplicationTickHook ( void )
{
}
```

FreeRTOS Hook Functions

Malloc Failed hook function

`configUSE_MALLOC_FAILED_HOOK` should be 1 in `FreeRTOSConfig.h`

and your application source file (`main.c`) should implement the below function

```
void vApplicationMallocFailedHook ( void )  
{  
}  
}
```

FreeRTOS Hook Functions

Stack over flow hook function

`configCHECK_FOR_STACK_OVERFLOW` should be 1 in `FreeRTOSConfig.h`
and your application source file (`main.c`) should implement the below function

```
void vApplicationStackOverflowHook( TaskHandle_t xTask, signed char  
*pcTaskName )  
{  
}  
}
```

Exercise

Write a program to send Microcontroller to sleep mode when Idle task is scheduled to run on the CPU and take the current measurement.

FreeRTOS Scheduling Policies

Important Scheduling Policies

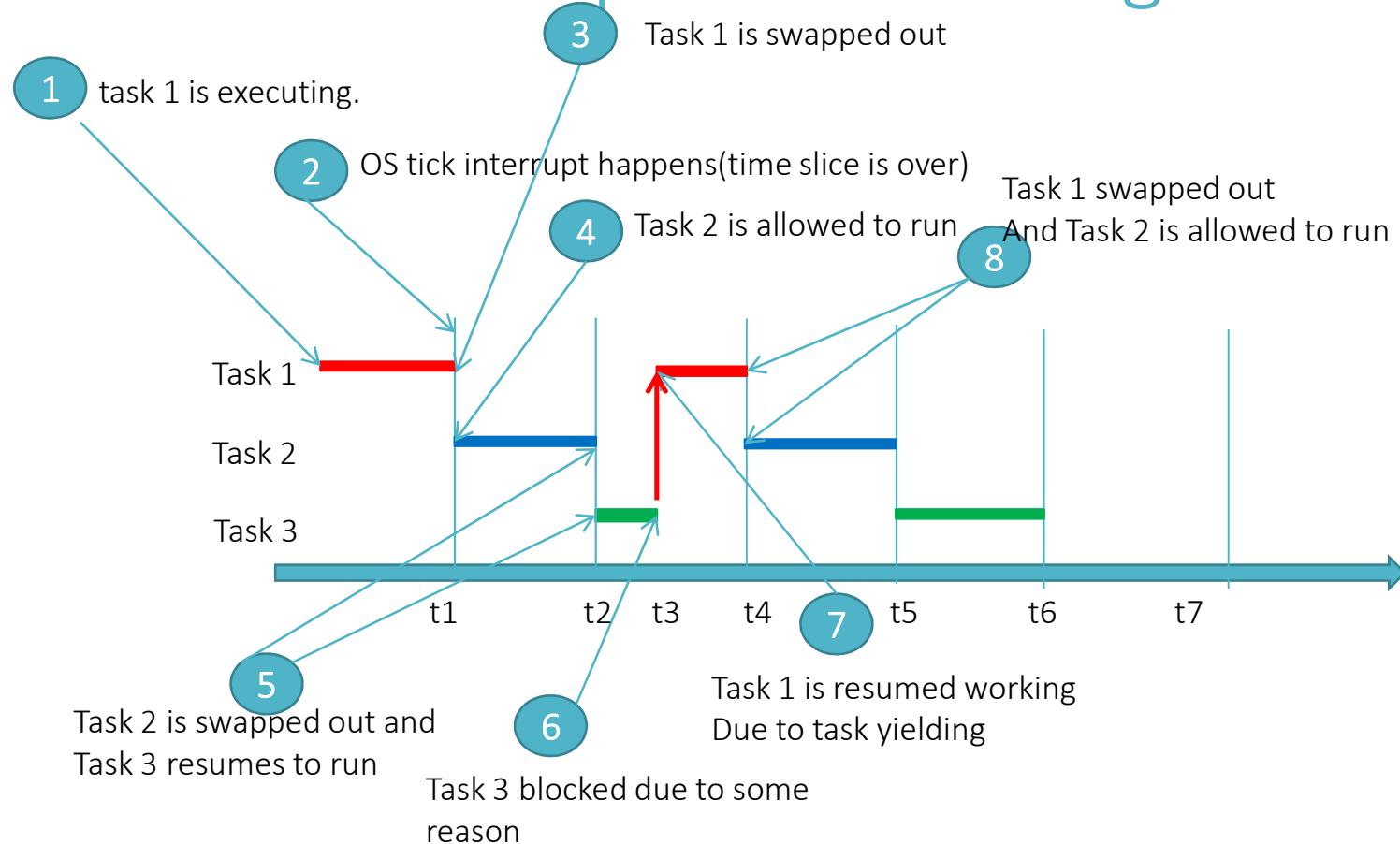
1. Preemptive scheduling
2. Priority based preemptive scheduling
3. co-operative scheduling

Preemption

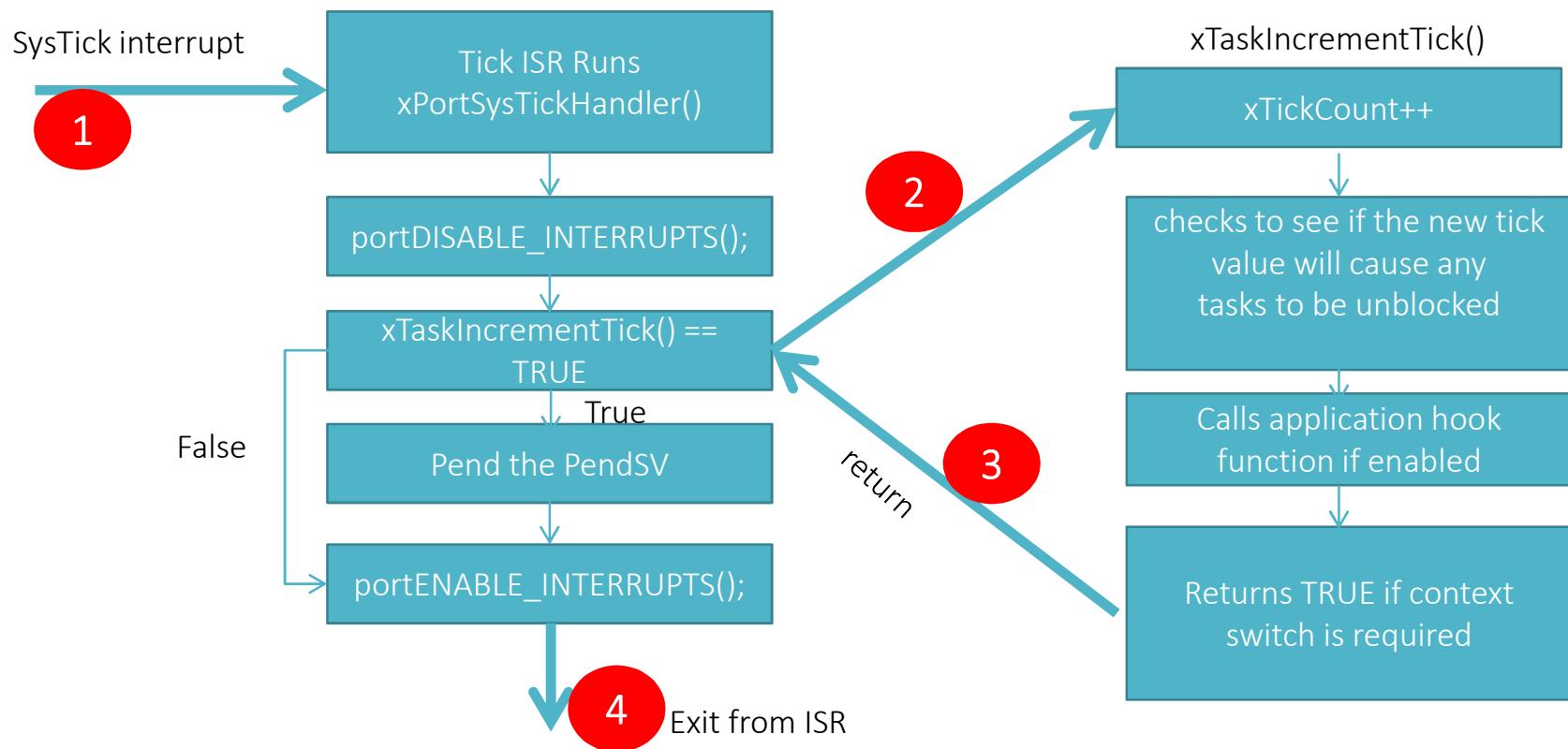
Preemption is the act of temporarily interrupting an already executing task with the intention of removing it from the running state **without its co-operation** .

Pre-emptive Scheduling

Pre-emptive Scheduling

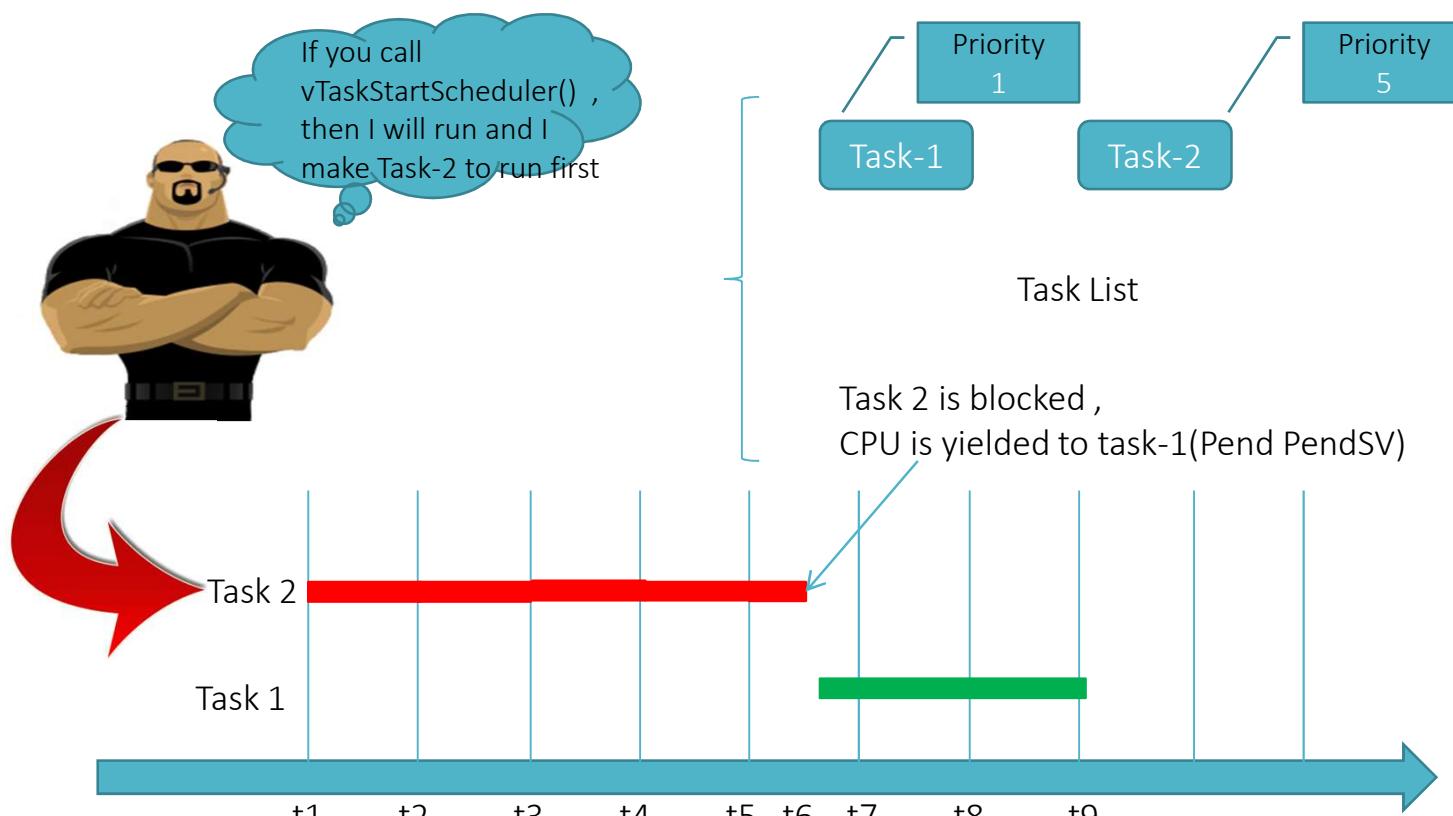


What RTOS tick ISR does ? : Conclusion

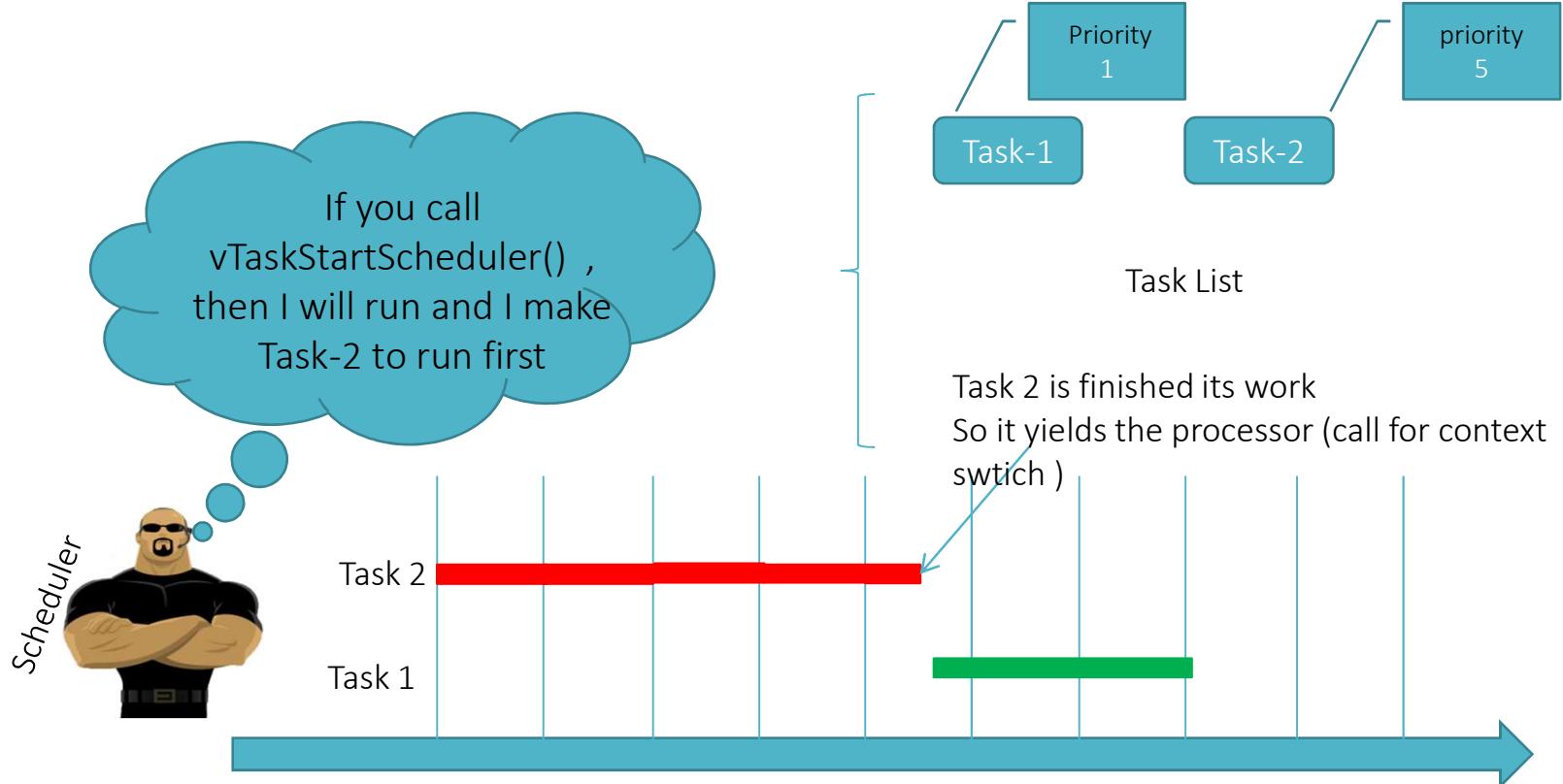


Priority based Pre-Emptive Scheduling

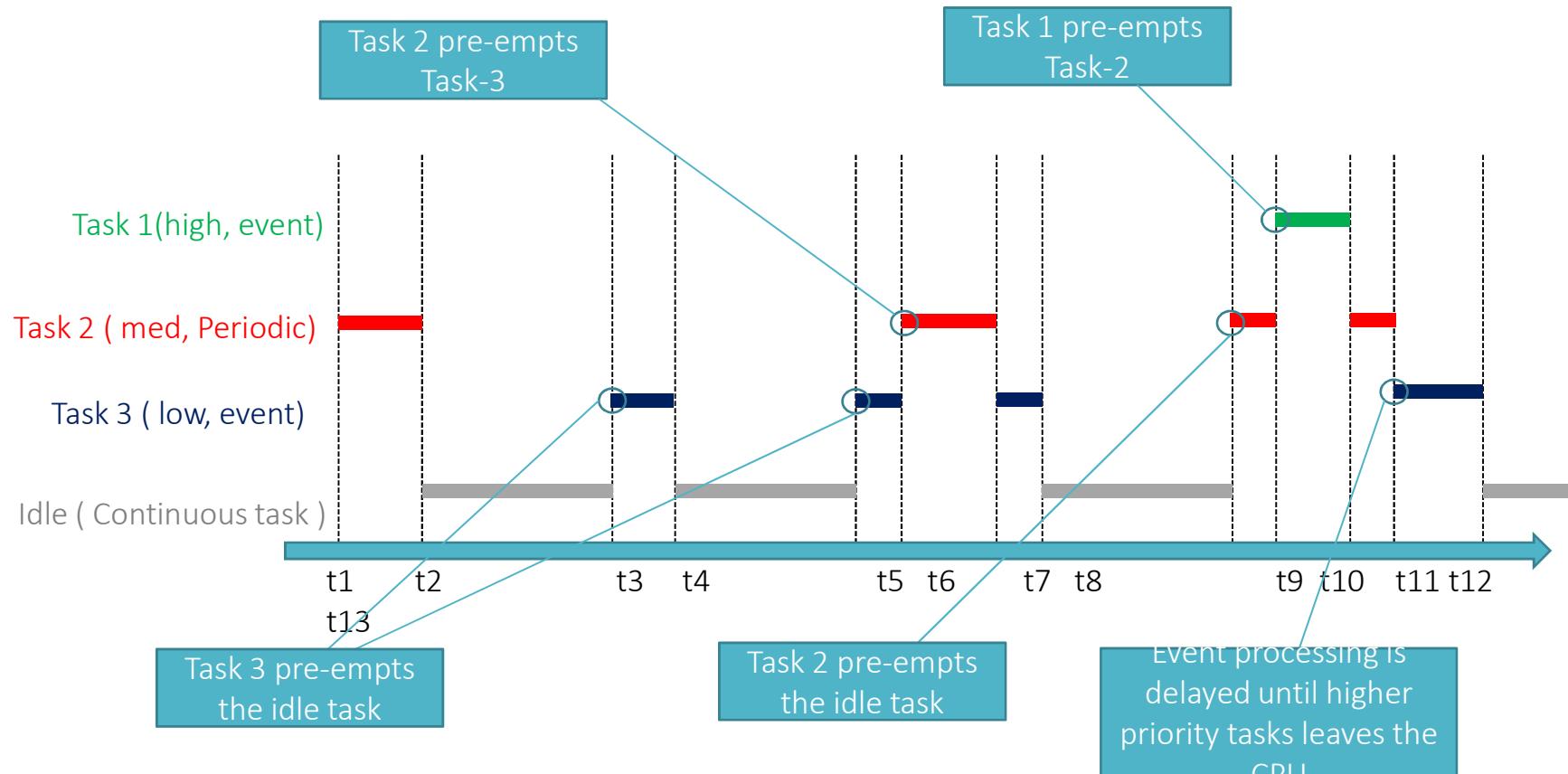
Prioritized Pre-emptive Scheduling



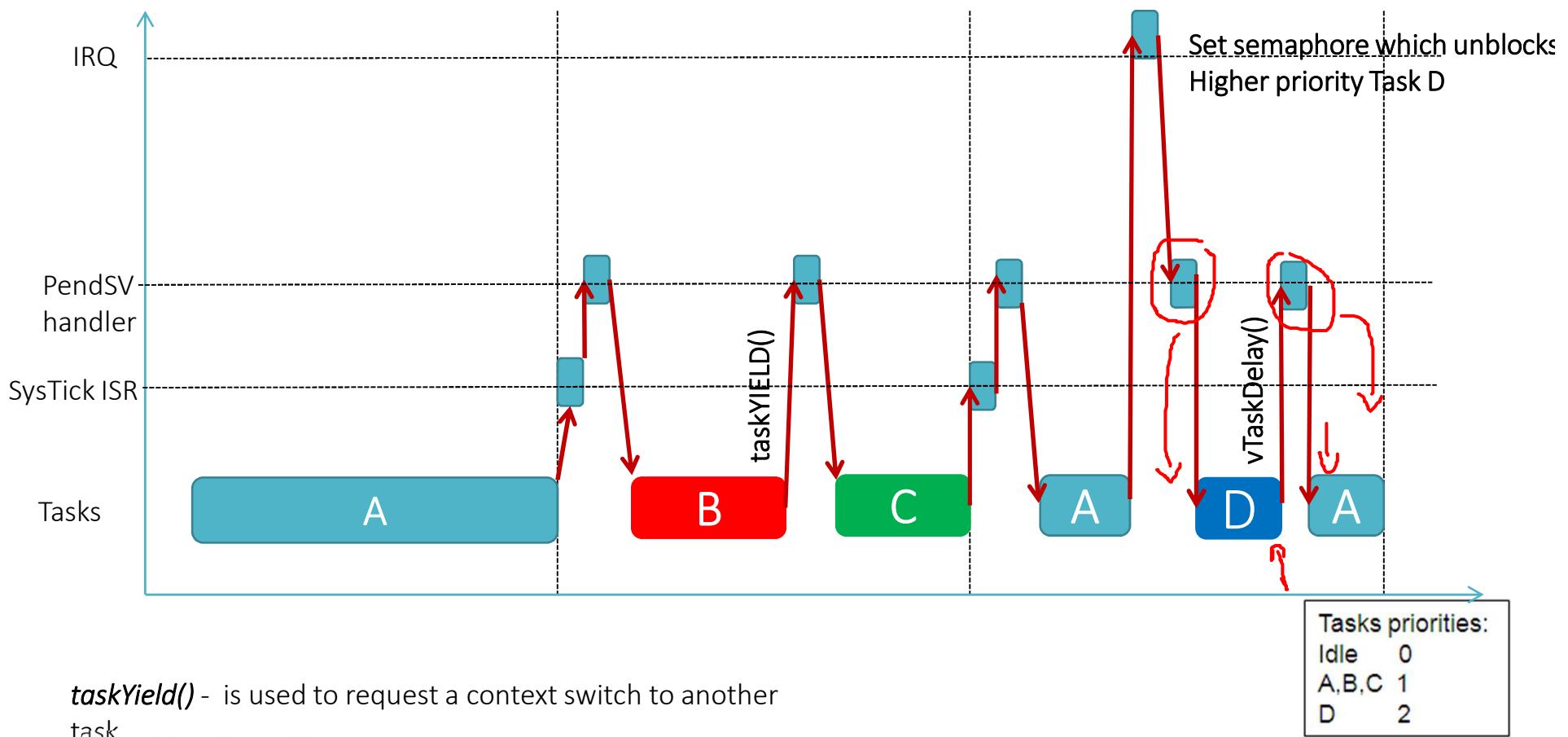
Prioritized Pre-emptive Scheduling



Prioritized Pre-emptive Scheduling



Prioritized Pre-emptive Scheduling

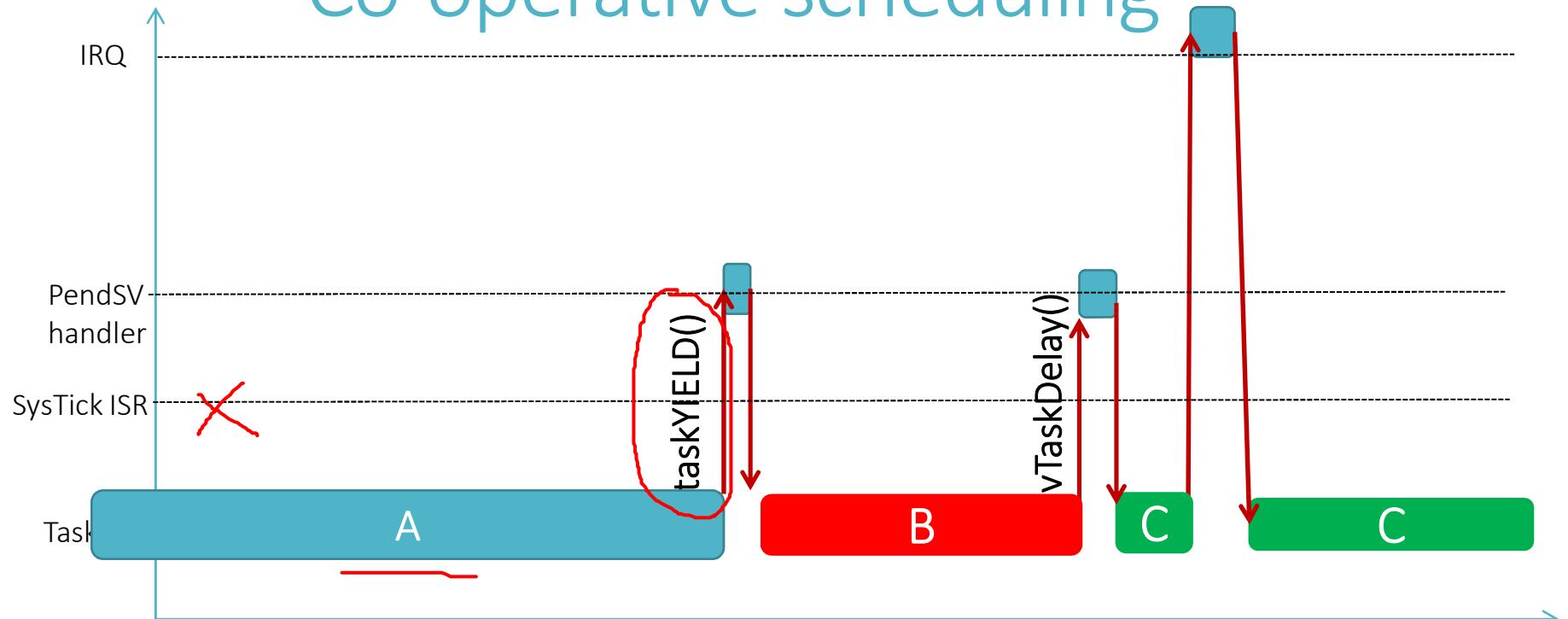


Co-operative scheduling

Co-Operative scheduling

As the name indicates, it is a co-operation based scheduling . That is Co-operation among tasks to run on the CPU.

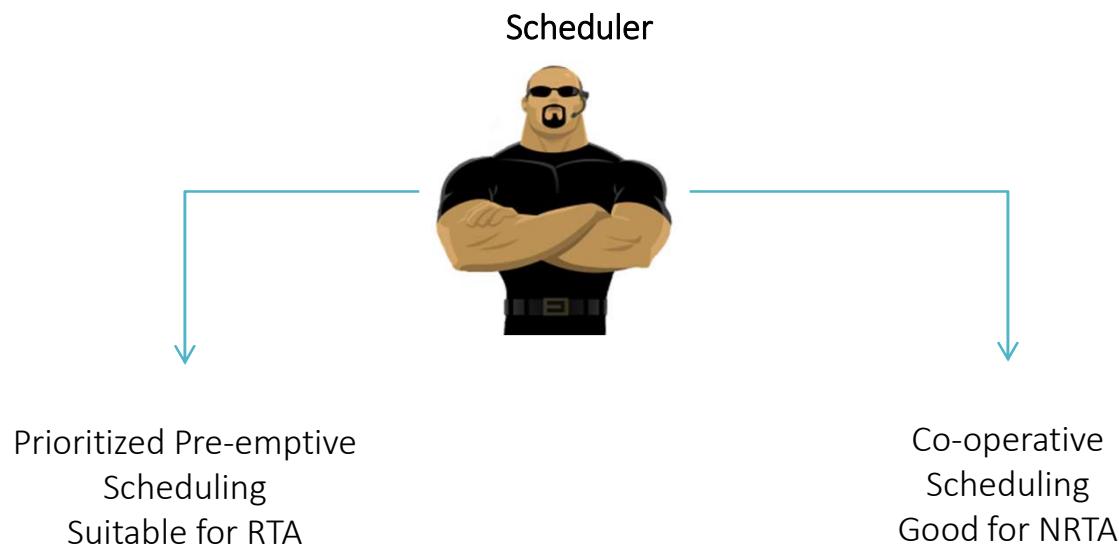
Co-operative scheduling



`taskYIELD()` - is used to request a context switch to another task

Tasks priorities:	
Idle	0
A,B,C	1

Conclusion



Conclusion

First we learnt about simple preemptive scheduling, here priorities of tasks are irrelevant . Equal amount of time slice is given to each ready state tasks.

Where as in priority based preemptive scheduling, the context switch will always happen to the highest priority ready state task.

So, here Priority will play a major role in deciding which task should run next .

And in the co-operative scheduling , the context switch will never happen without the co-operation of the running task. Here, the systick handler will not be used to trigger the context switch.

When the task thinks that it no longer need CPU. then it will call task yield function to give up the cpu for any other ready state tasks.

Our embedded system courses are well suited for beginners to intermediate students, job seekers and professionals .

Life time access through udemy

Dedicated support team

Course completion certificate

Accurate Closed captions (subtitles) and transcripts

Step by step course coverage

30 days money back guarantee

Browse all courses on
MCU programming , RTOS,
embedded Linux
@ www.fastbitlab.com