

# 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](http://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](http://www.fastbitlab.com)

Email : [contact@fastbitlab.com](mailto: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*."

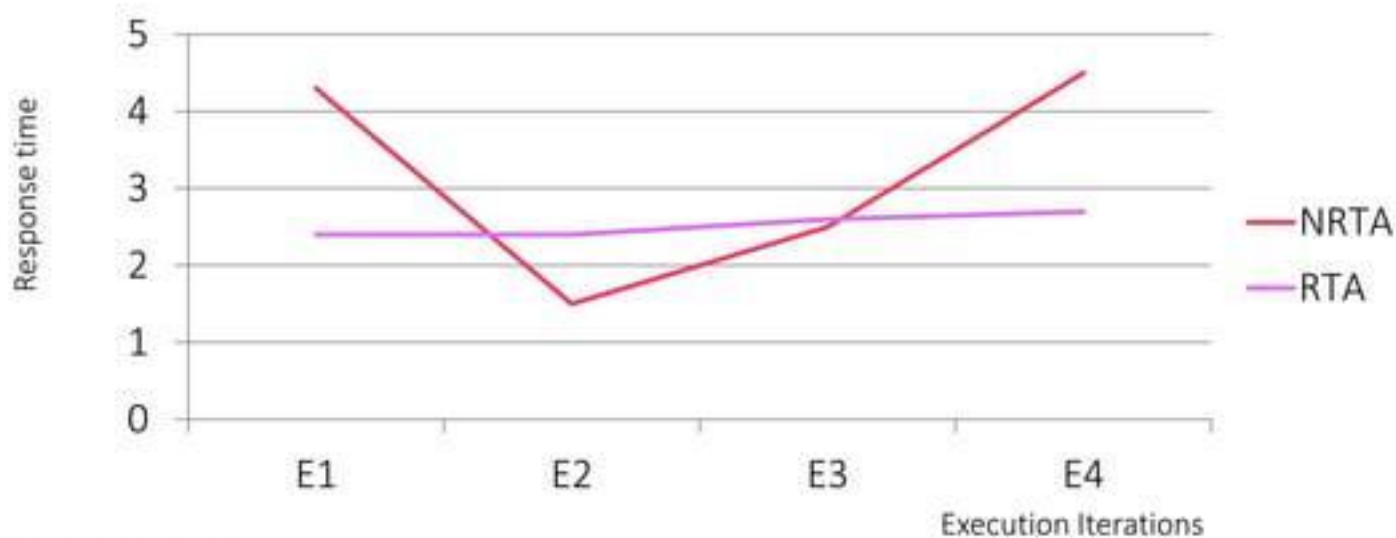
# What is Real time ?

“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 ?

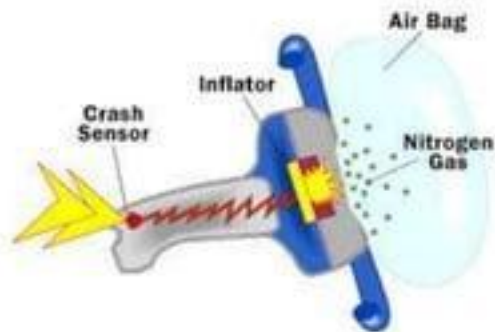
The response time is guaranteed



# 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



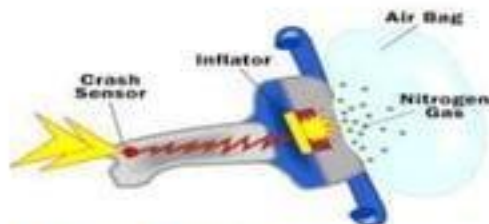
May be soft-real time application ?  
Delay is tolerable

# Some Examples of RTAs

Missile guidance and control systems



Ani-Lock Breaking 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 )

Audio/Video Streaming(soft)

Communications

# 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



Linux



iOS



Android

RTOS



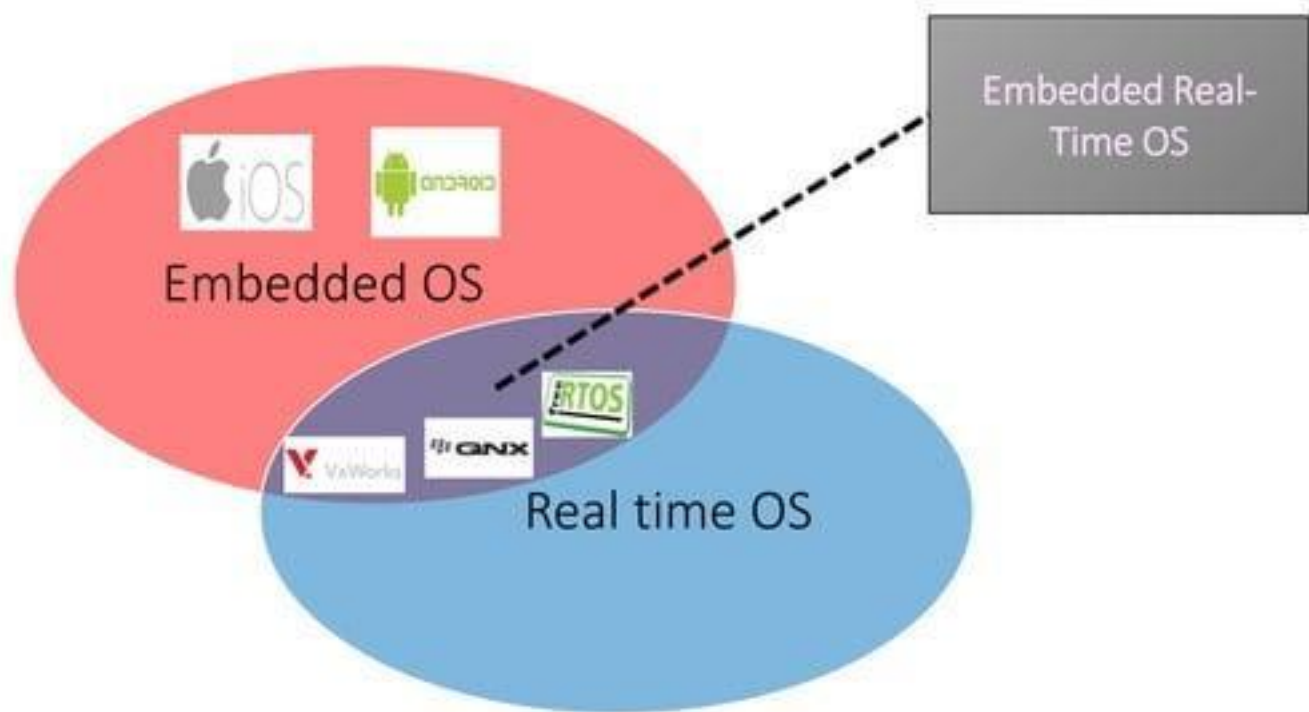
VxWorks



QNX



INTEGRITY



# 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

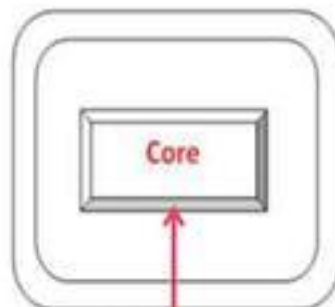
In Computing , Latency means “The time that elapses between a stimulus and the response to it”.

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”

Event occurred  
(Ex. Crash detection)



t1



A processor

A Task is made to run on CPU  
(ex. Air bag deployment Task)

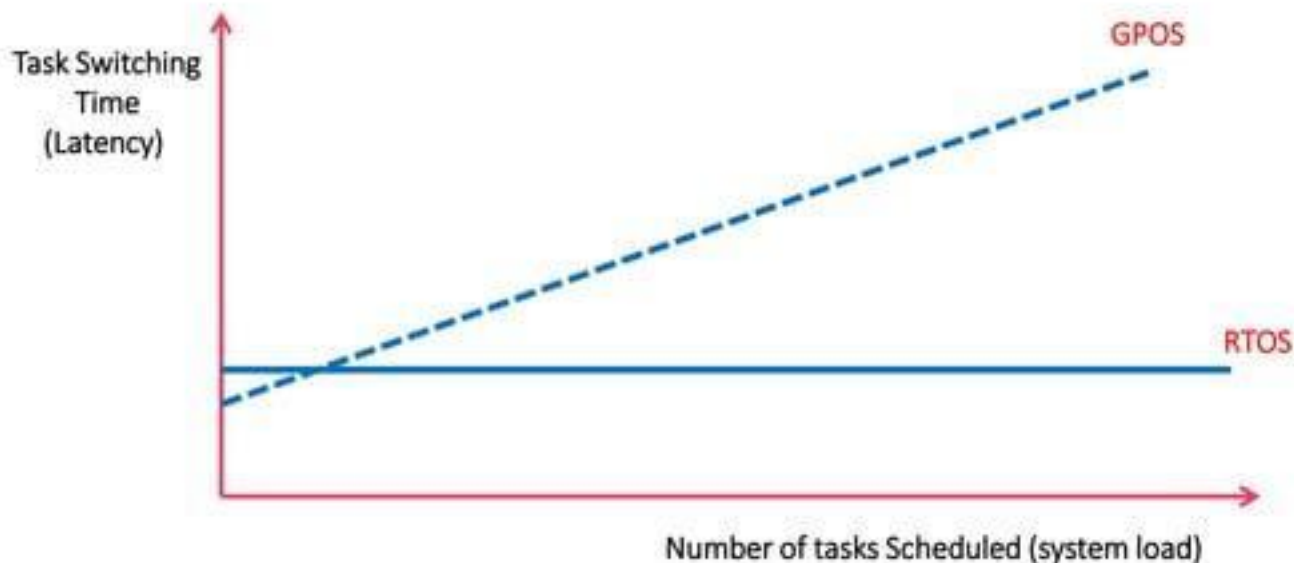
t2

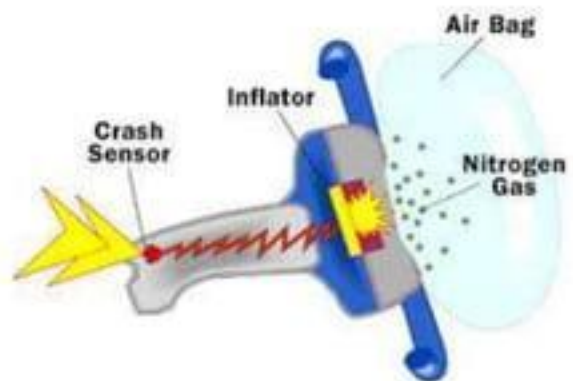
Time



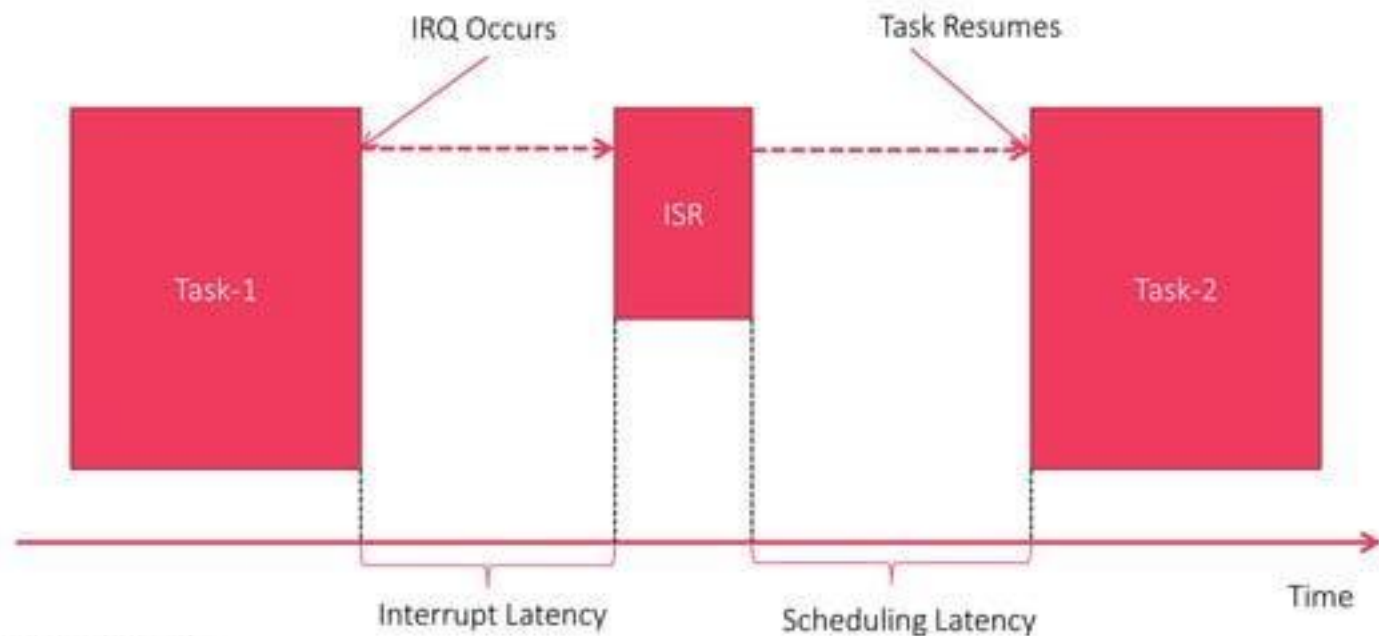
Task Switching Latency

# RTOS vs GPOS: Task Switching Latency

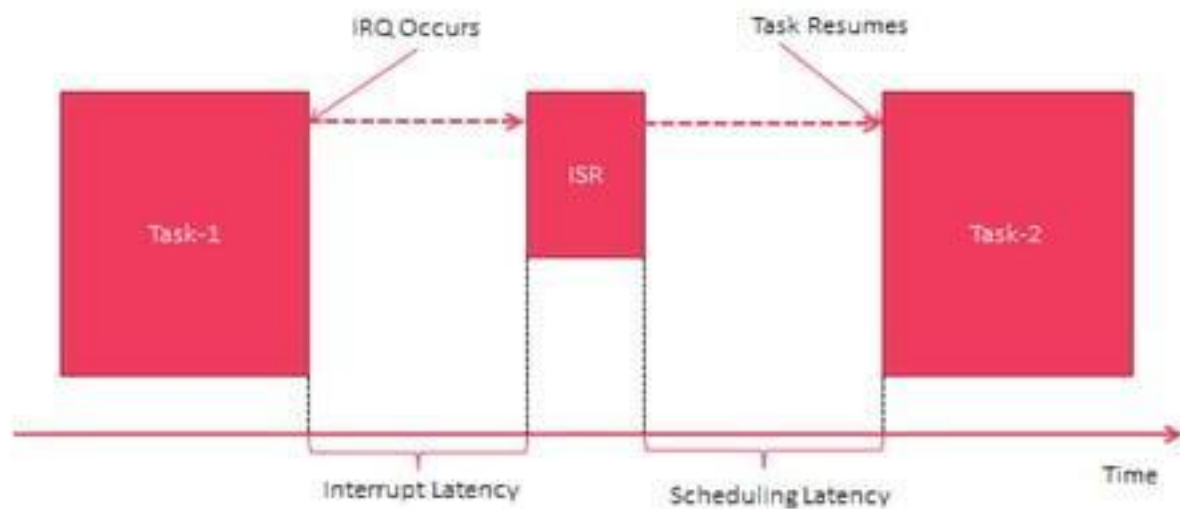




# RTOS vs GPOS: Interrupt Latency

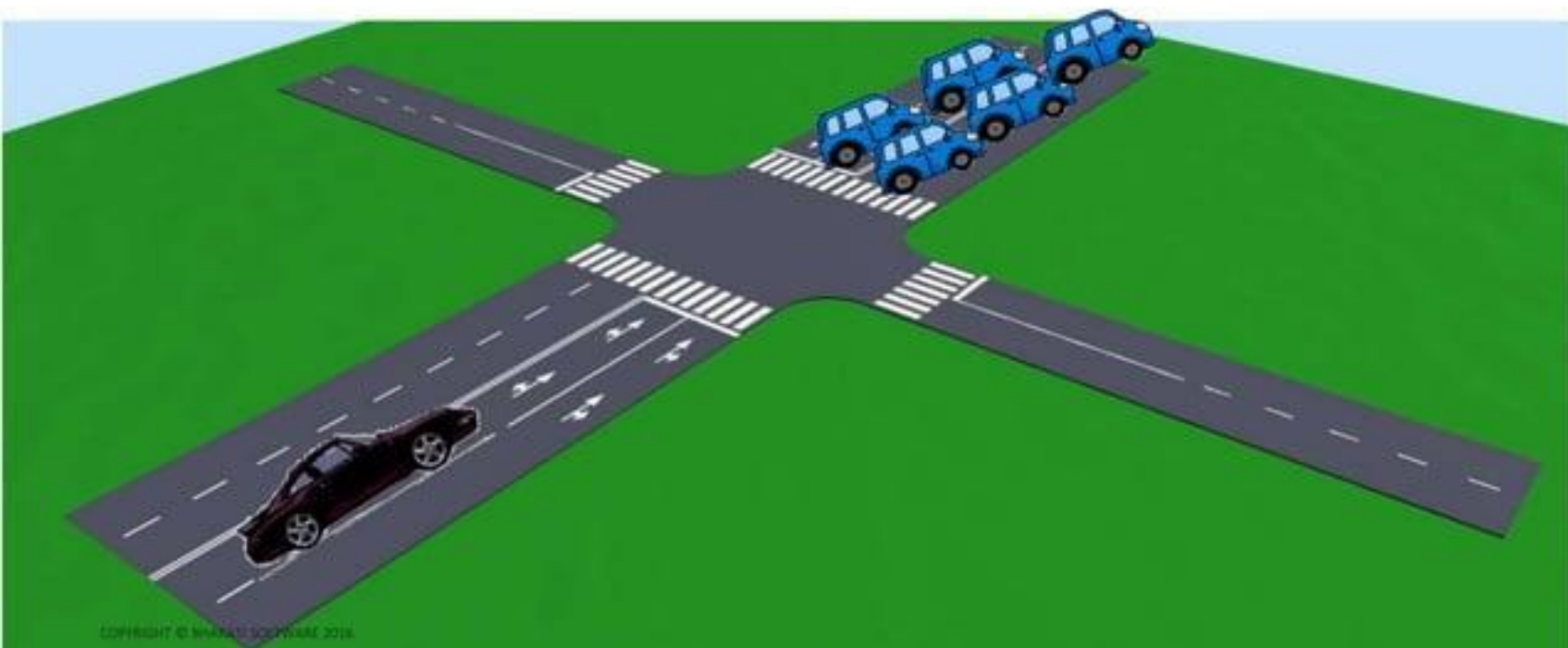


# RTOS vs GPOS: Interrupt Latency

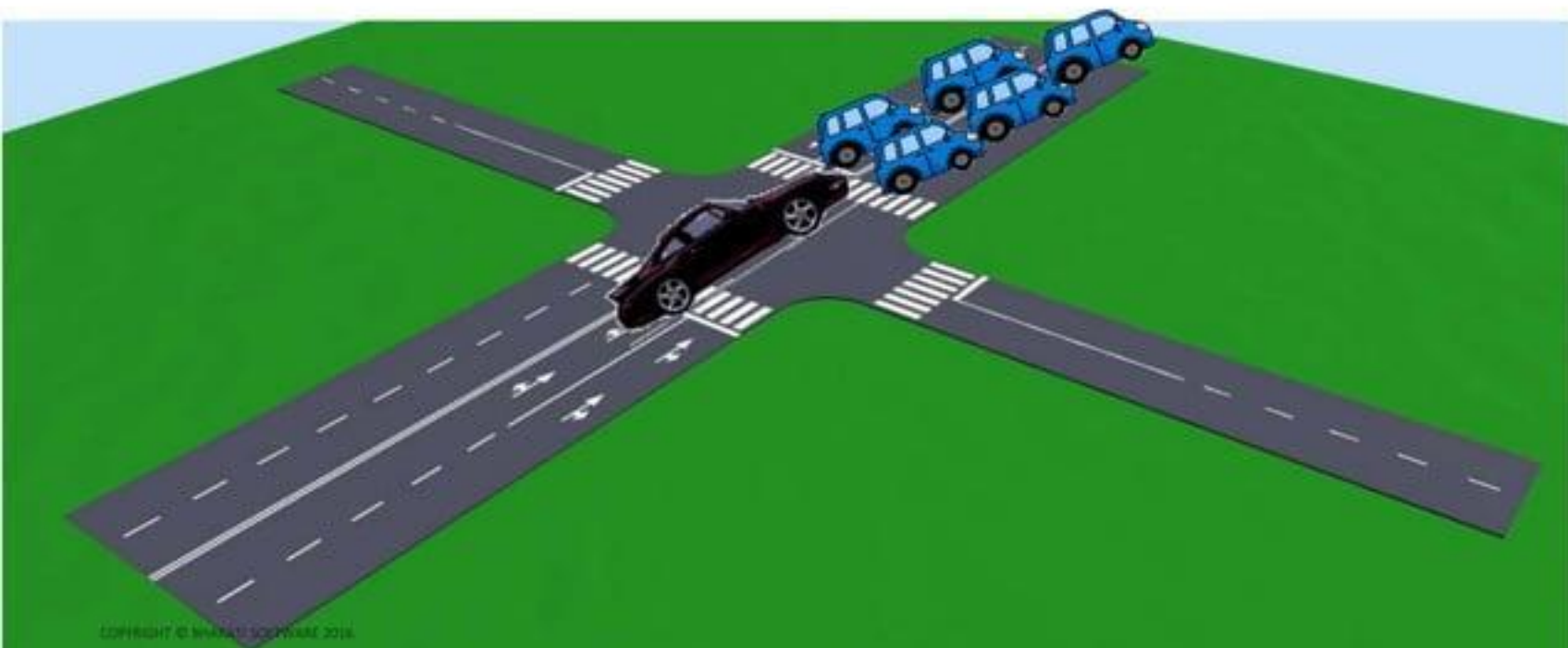




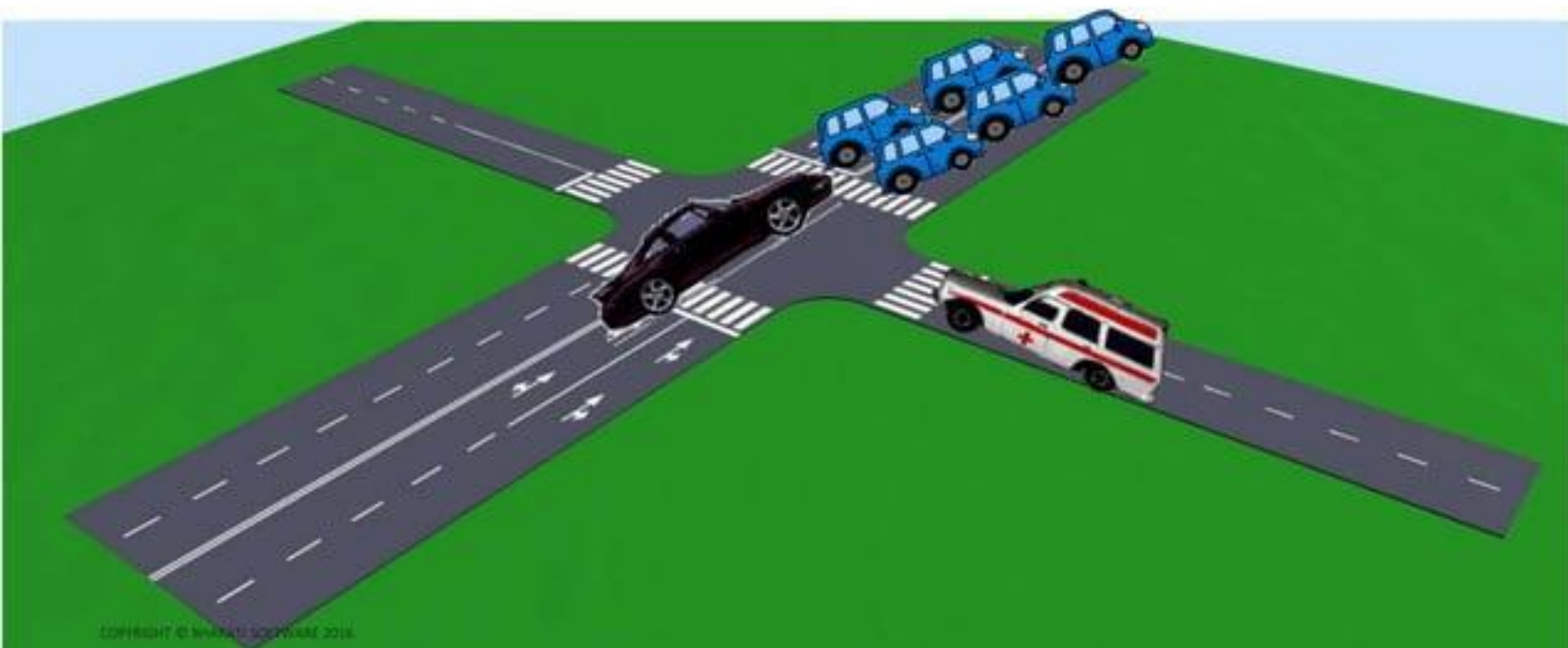
# RTOS vs GPOS: Priority Inversion



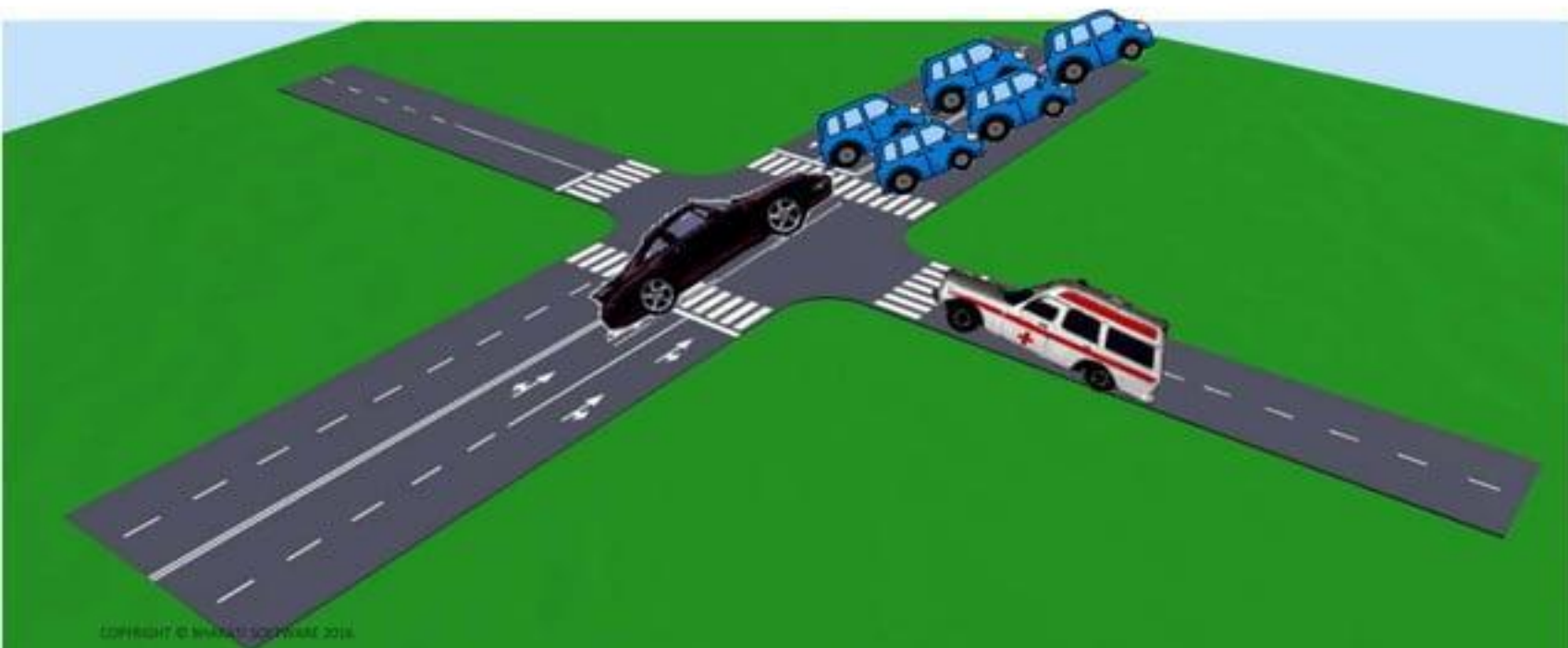
# RTOS vs GPOS: Priority Inversion



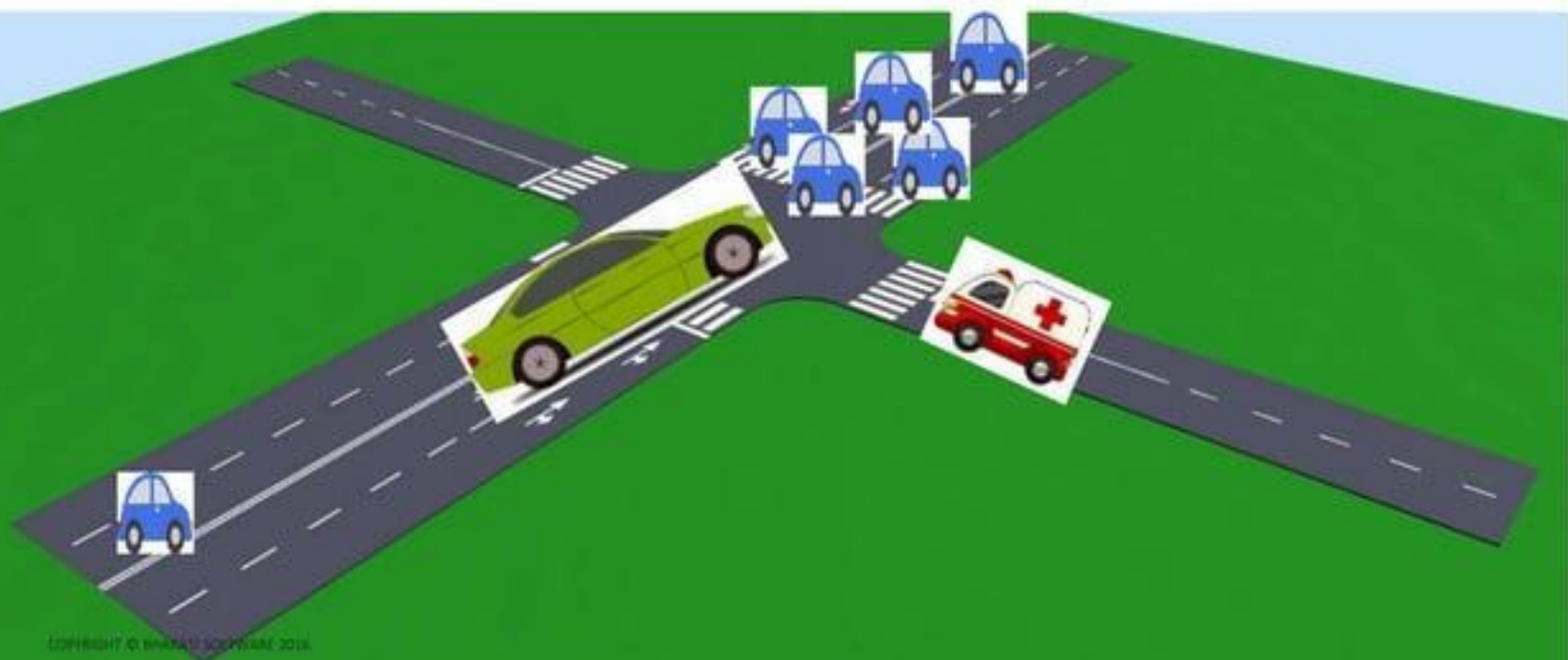
# RTOS vs GPOS: Priority Inversion



# 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 where as the kernel operations of a GPOS are not pre-emptible

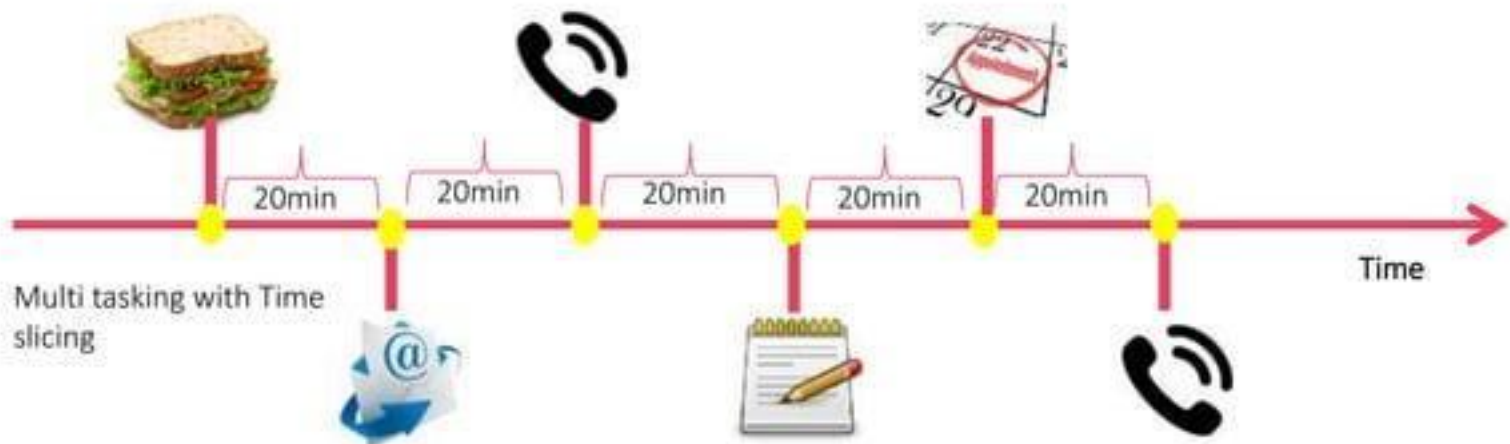


# What are the features that a RTOS has but a GPOS doesn't ?

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 ??





You

Having  
Breakfast

Assistant-1

Emailing on  
behalf of you

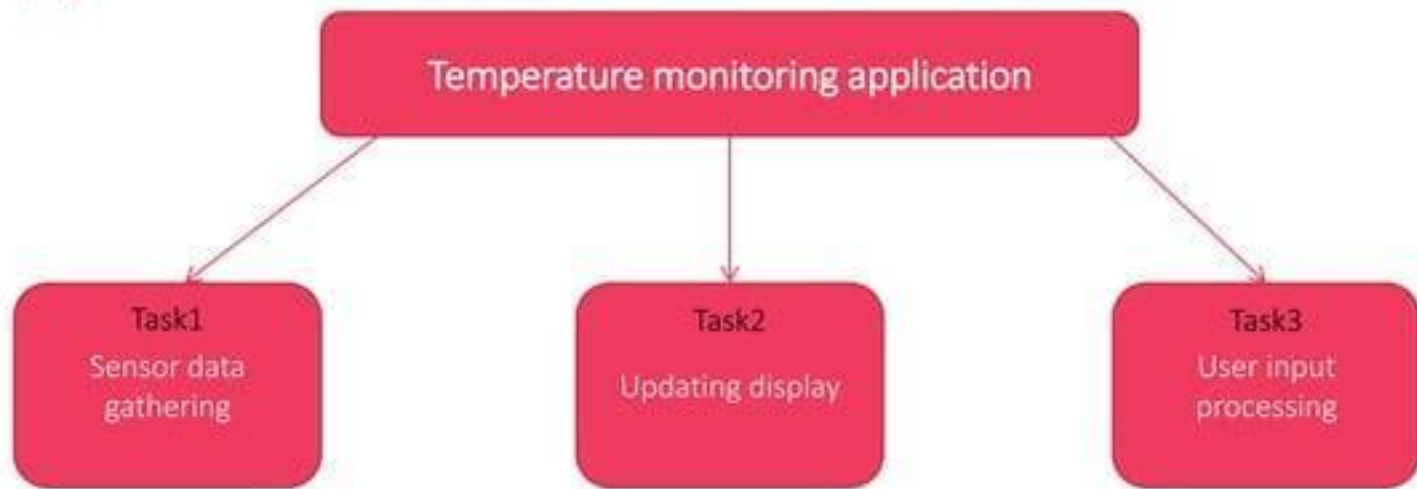
Assistant-2

Receiving calls  
on behalf of  
you

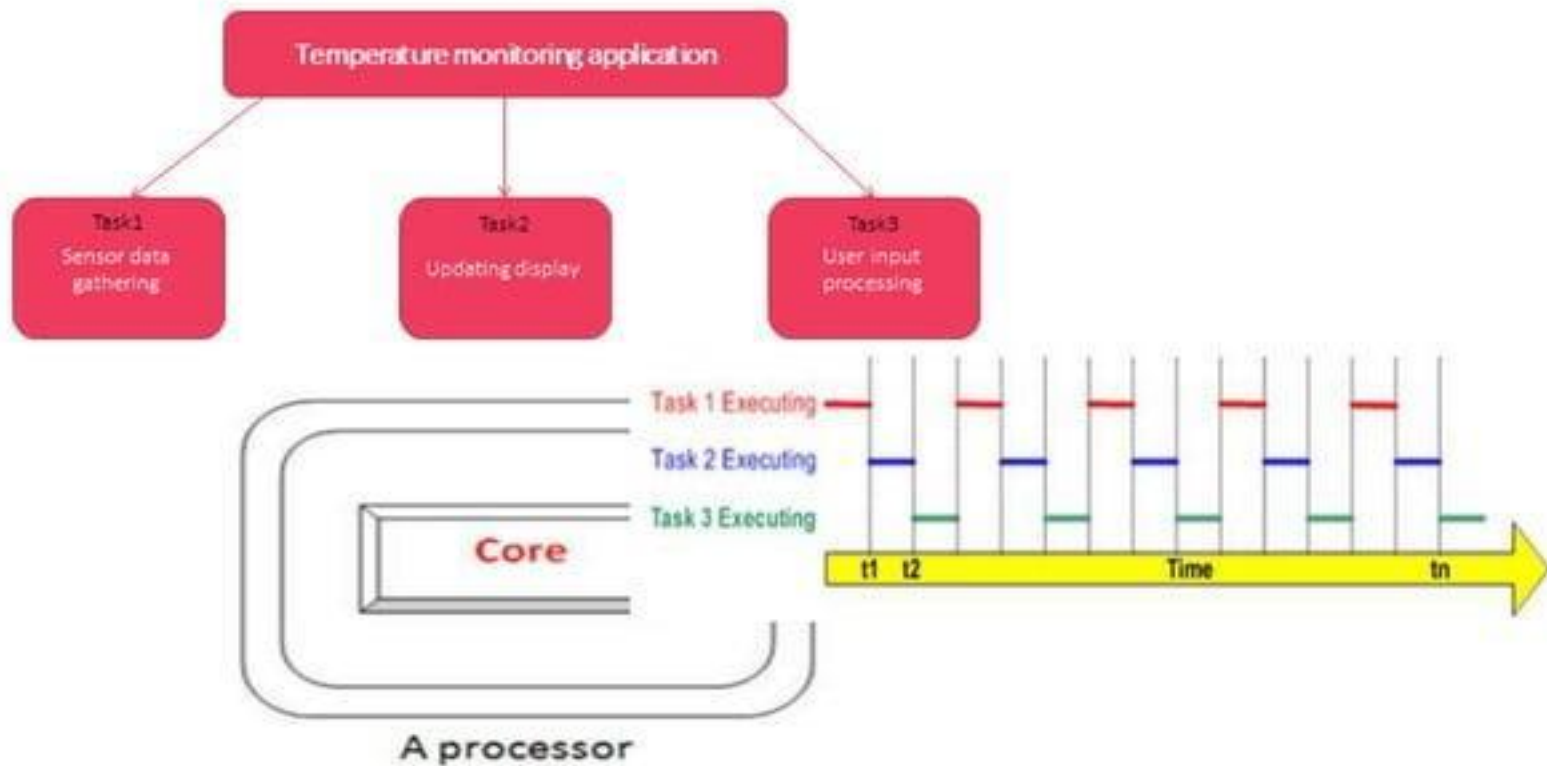
Assistant-3

Managing  
appointments

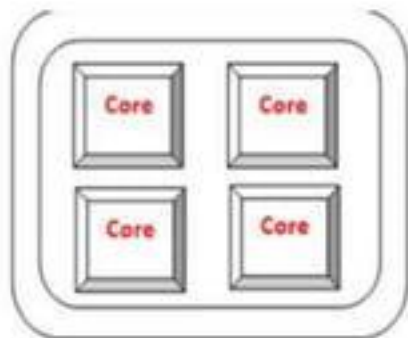
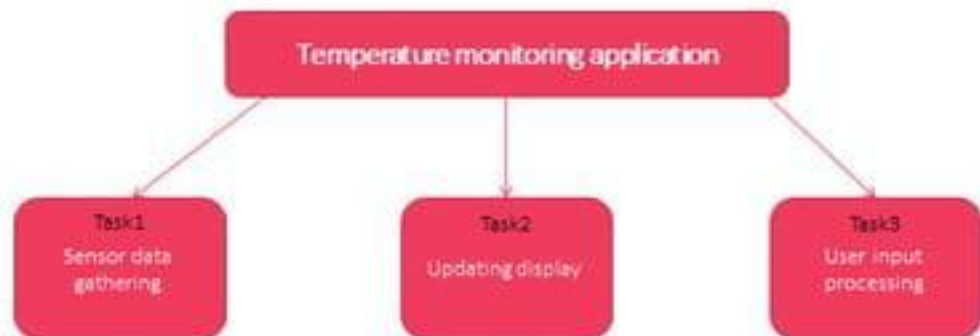
# Application and tasks



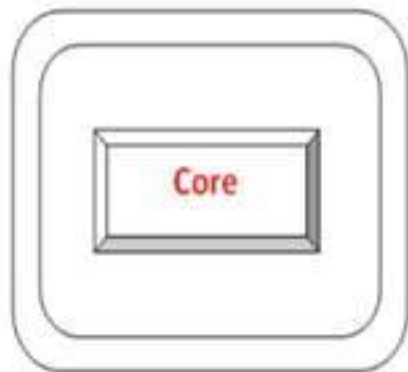
# Application and tasks



# Application and tasks

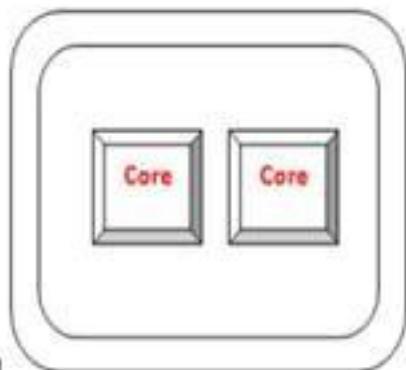


A processor with 4 cores

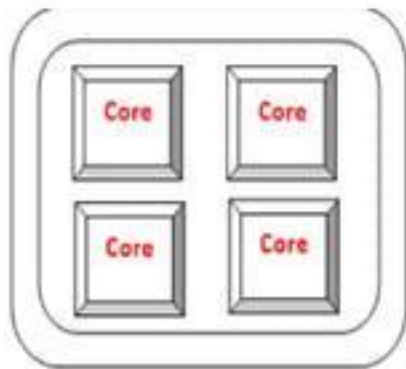


A processor

Any point in time  
only 1 task can run



A processor with 2 cores



A processor with 4 cores

Any point in time 2 or 4  
task can run

## Scheduler



Tasks Don't  
run unless i  
decide



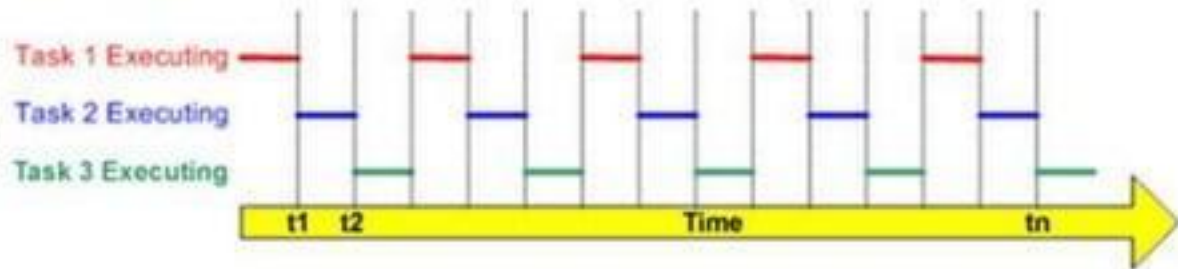
Task List

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

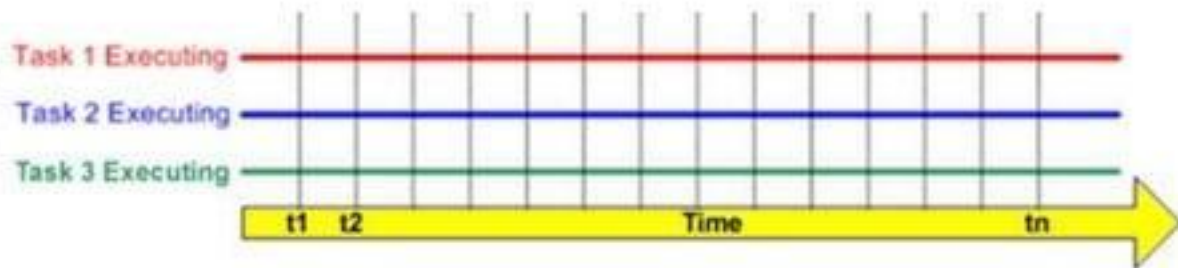
You can configure **my scheduling policy**

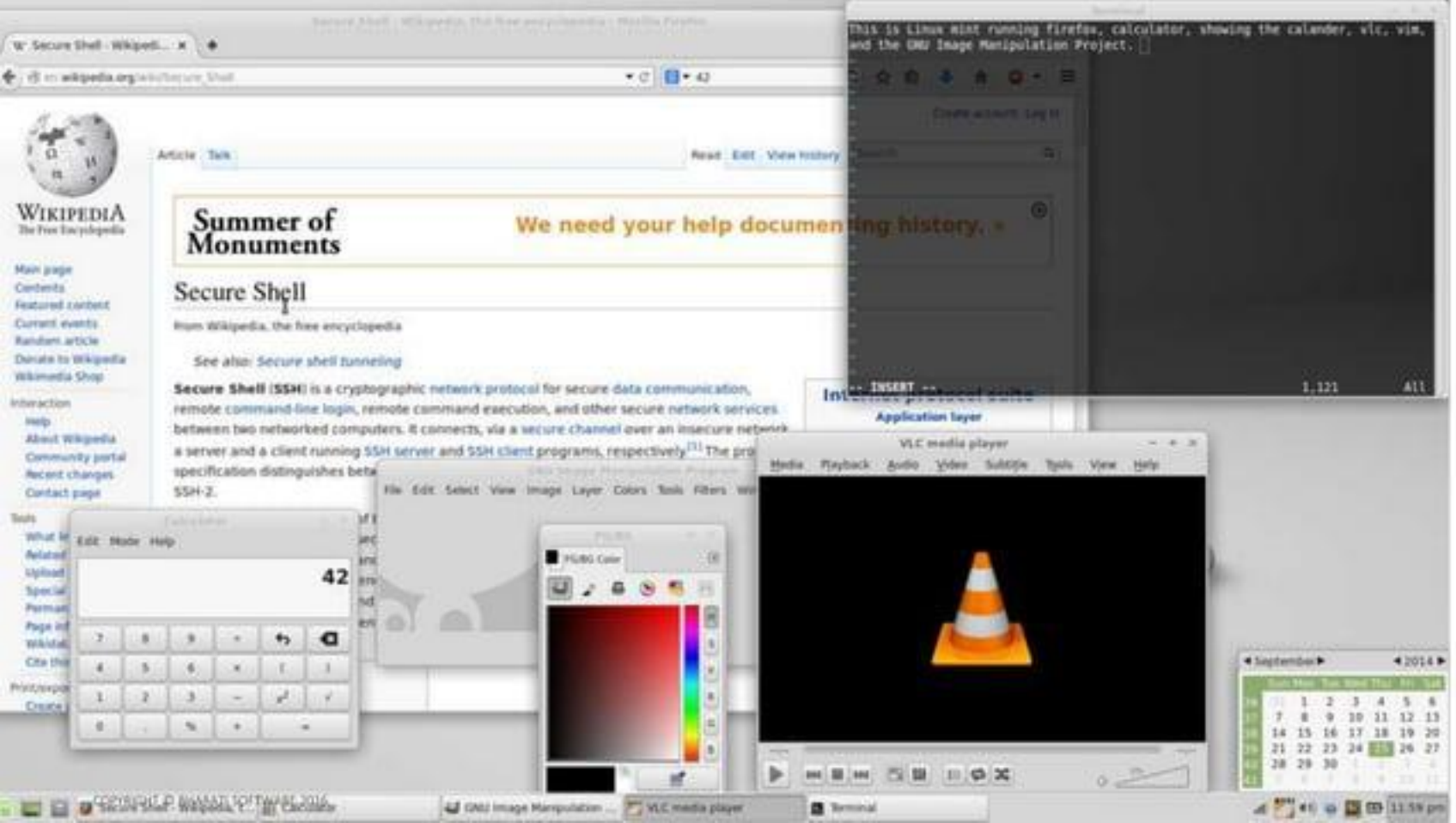


This is how multiple tasks run on CPU



Which gives the illusion that all tasks are executing simultaneously





# FreeRTOS Overview

# About freertos.org

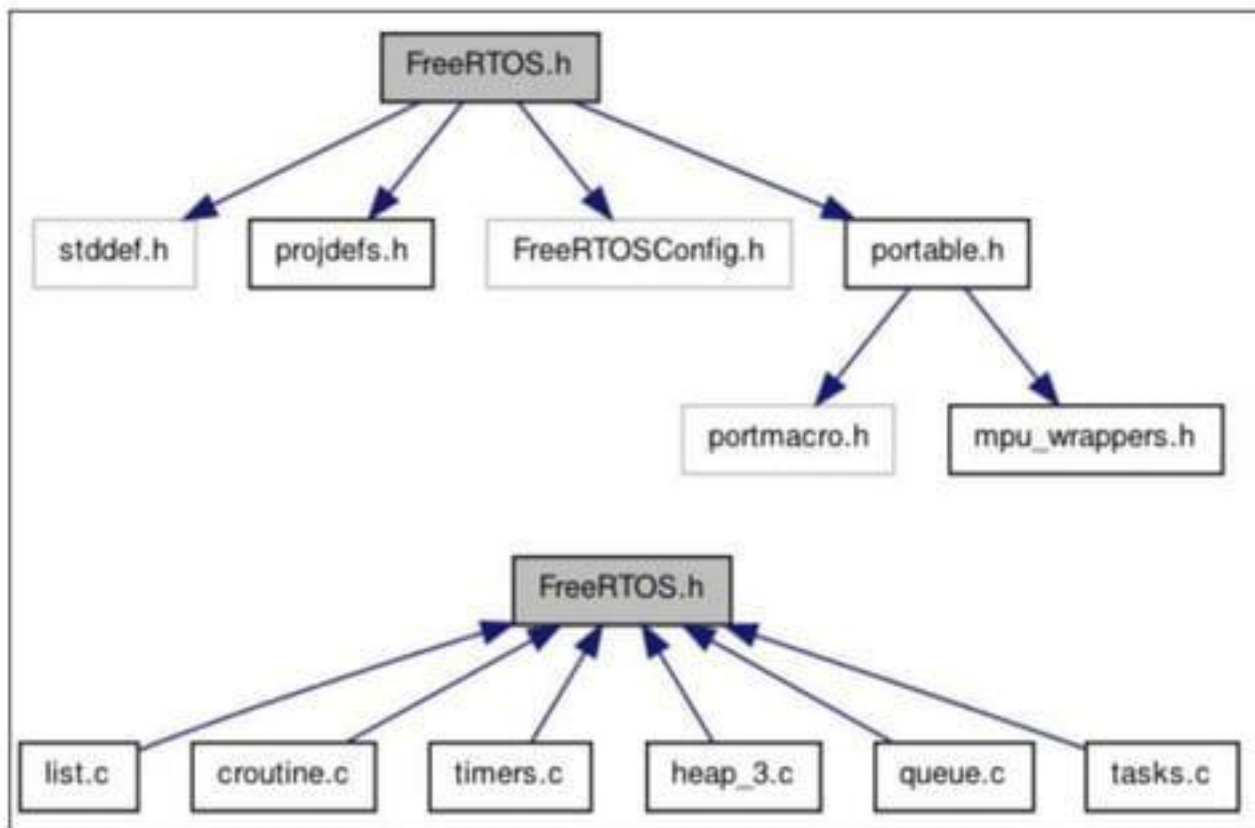
## Supported architectures [ edit ]

- 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
- Multicore
  - Multicore 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

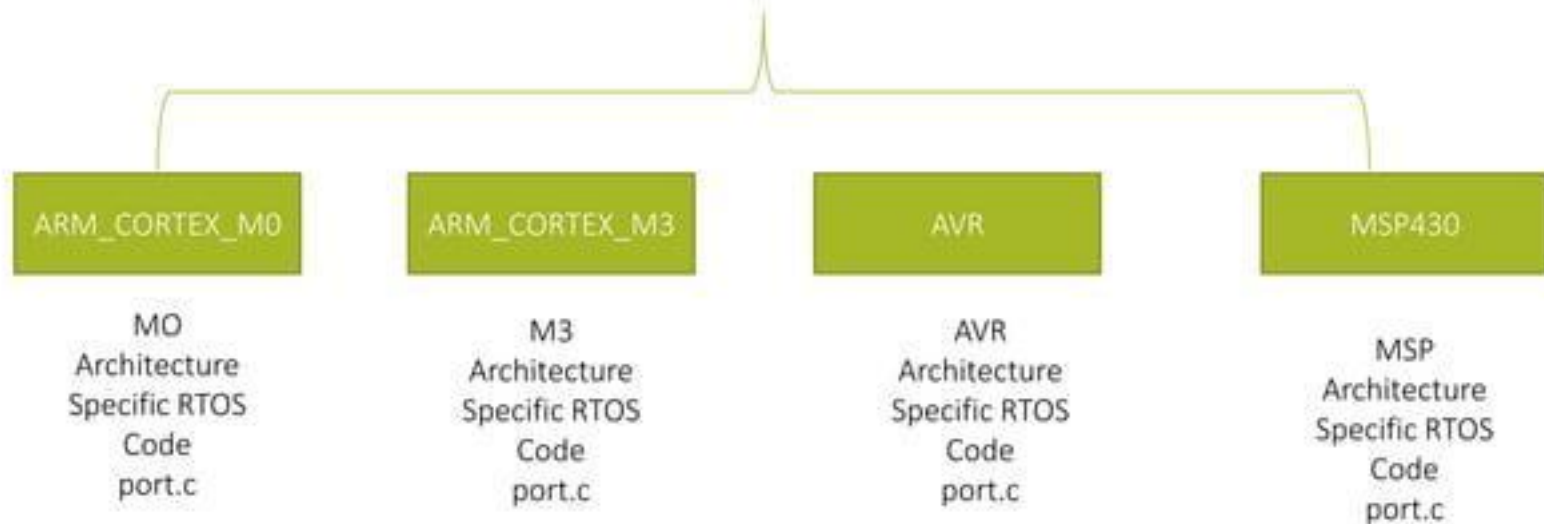
<https://en.wikipedia.org/wiki/FreeRTOS>

Coming Up-Next :

# FreeRTOS file structure and Hierarchy

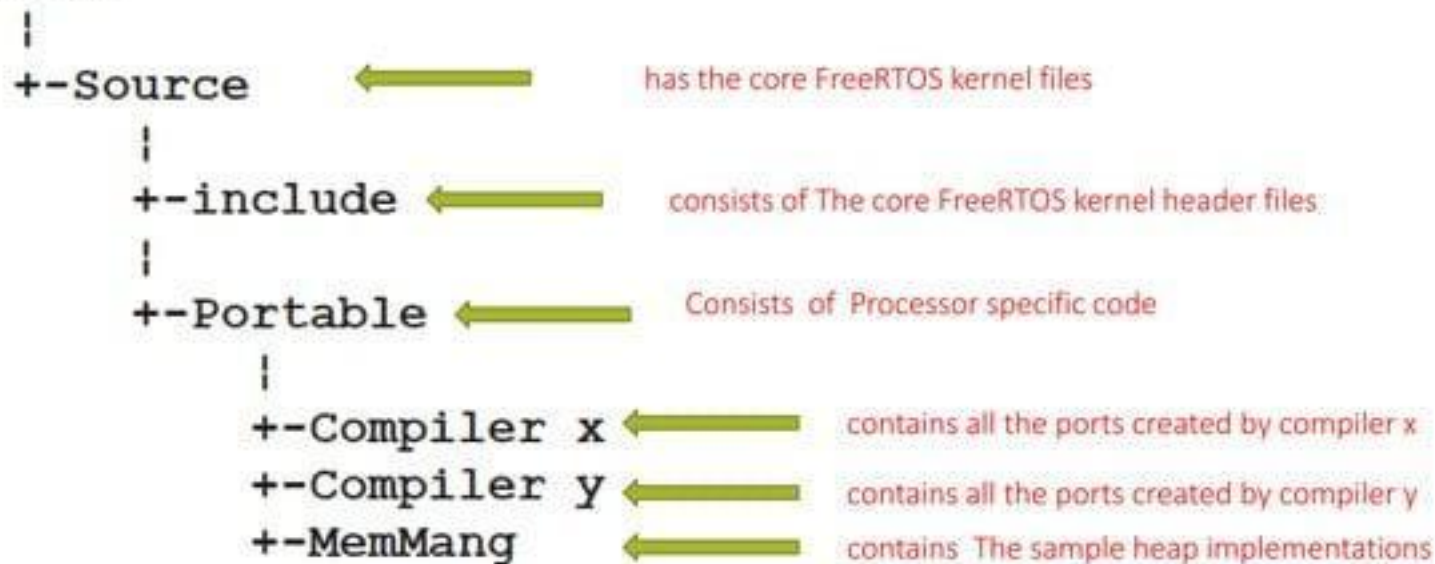


## Different Processor Architecture





# FreeRTOS



FreeRTOSv9.0.0

FreeRTOS-Plus

FreeRTOS

Demo

License

Source

- ARM7\_AT91FR40008\_GCC
- ARM7\_AT91SAM7S64\_IAR
- ARM7\_AT91SAM7X256\_Eclipse
- ARM7\_LPC2106\_GCC
- ARM7\_LPC2129\_IAR
- ARM7\_LPC2129\_Keil\_RVDS
- ARM7\_LPC2138\_Rowley
- ARM7\_LPC2368\_Eclipse
- ARM7\_LPC2368\_Rowley
- ARM7\_STR71x\_IAR
- ARM7\_STR75x\_GCC
- ARM7\_STR75x\_IAR
- ARM9\_AT91SAM9XE\_IAR
- ARM9\_STR91X\_IAR
- AVR\_ATMega323\_IAR
- AVR\_ATMega323\_WinAVR
- AVR32\_UC3
- ColdFire\_MCF51CN128\_CodeWarrior

license

include

portable

croutine

event\_groups

list

queue

readme

tasks

timers

- croutine 11 File
- deprecated\_definitions 11 File
- event\_groups 11 File
- FreeRTOS 11 File
- list 11 File
- mpu\_prototypes 11 File
- mpu\_wrappers 11 File
- portable 11 File
- projects 11 File
- queue 11 File
- sample 11 File
- StackMacros 11 File
- stdout\_readme README File
- task 11 File
- timers 11 File

- BCI
- CCS
- CodeWarrior
- Common
- GLC
- IAR
- Keil
- MicroBlaze
- NetSight
- NetLab
- MSVC - MingW
- uVision
- PowerPC
- Rowley
- RVDS
- SDCC
- Software

# FreeRTOS Kernel features

# 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*

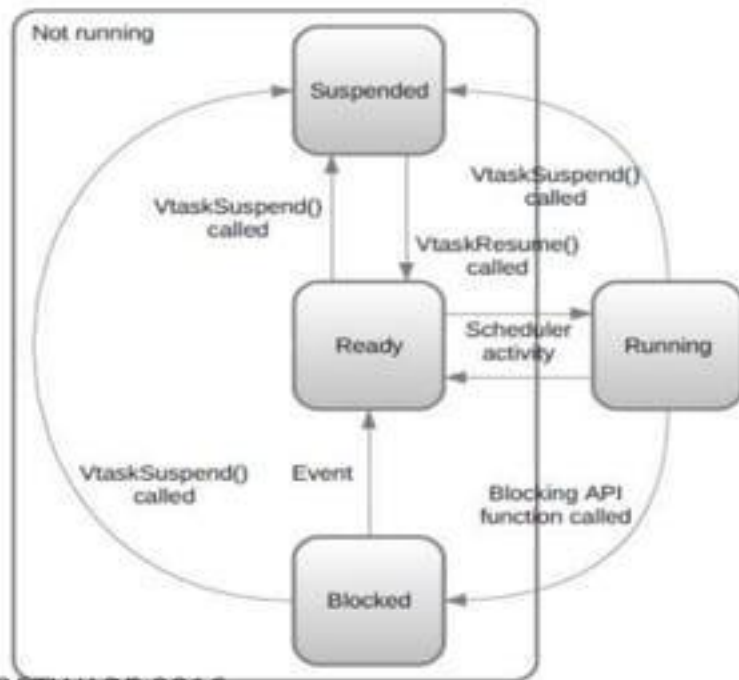


*Trace hook  
macros*

Coming Up-Next

# Overview of FreeRTOS Task Management

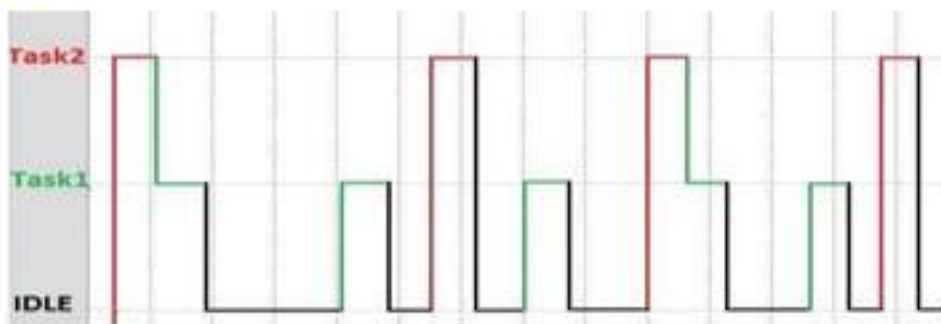
# Task States



# FreeRTOS other features



# 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

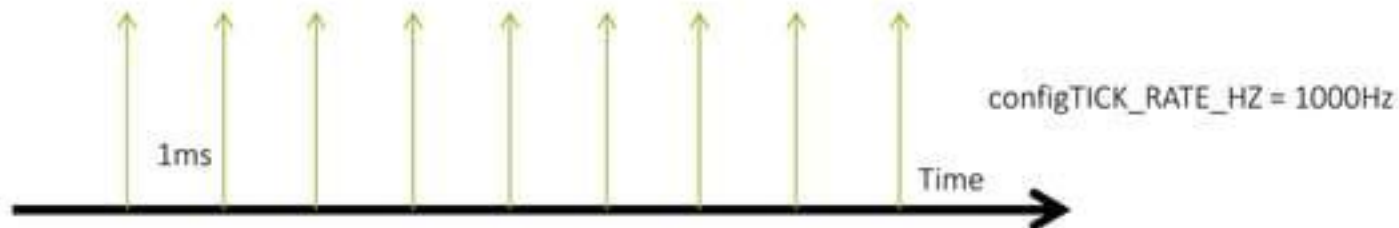
Then implement the below function in your application

```
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

# Tick hook function



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

You can use tick hook function to implement timer functionality.

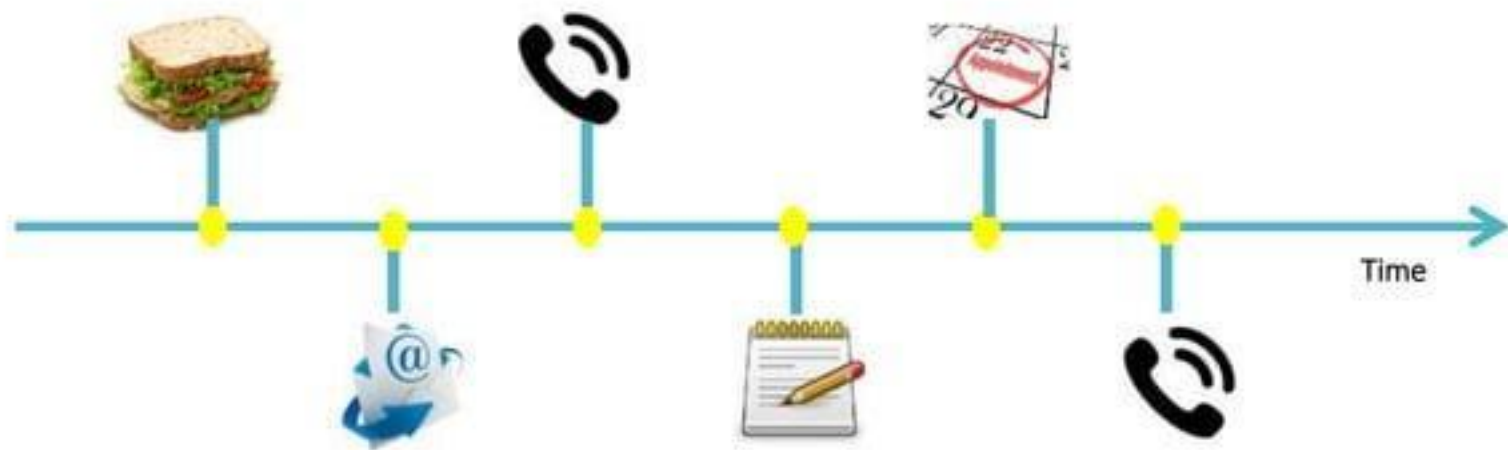
The tick hook will only get called if `configUSE_TICK_HOOK` is set to 1 within `FreeRTOSConfig.h`

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

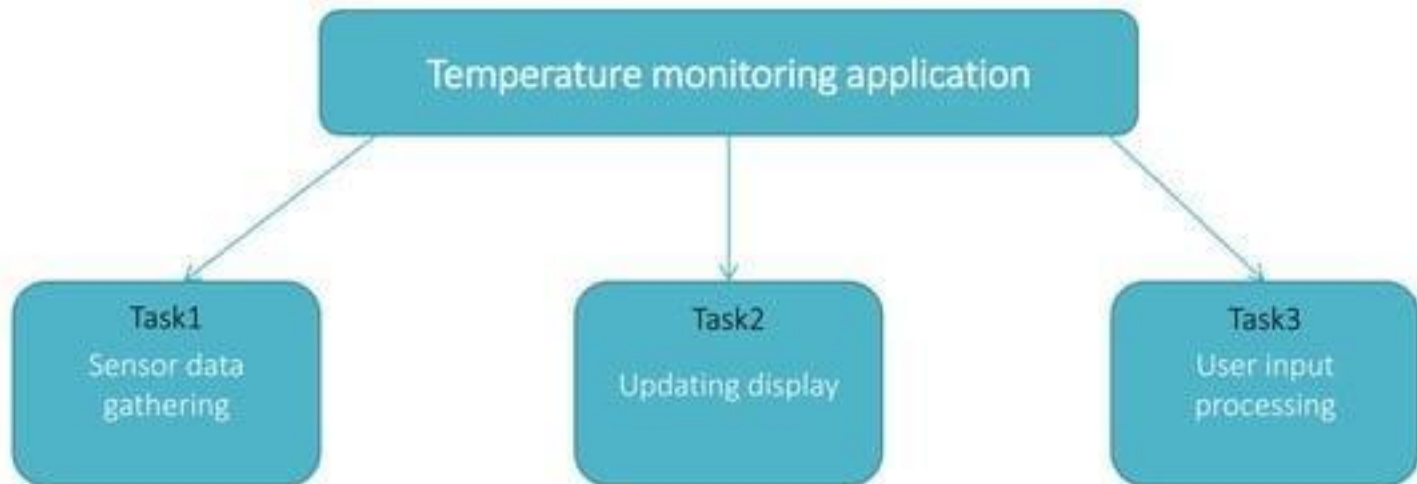
`vApplicationTickHook()` executes from within an ISR so must be very short, not use much stack, and not call any API functions that don't end in "FromISR"

# Free RTOS Task Creation

# 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

# API to Create a FreeRTOS Task



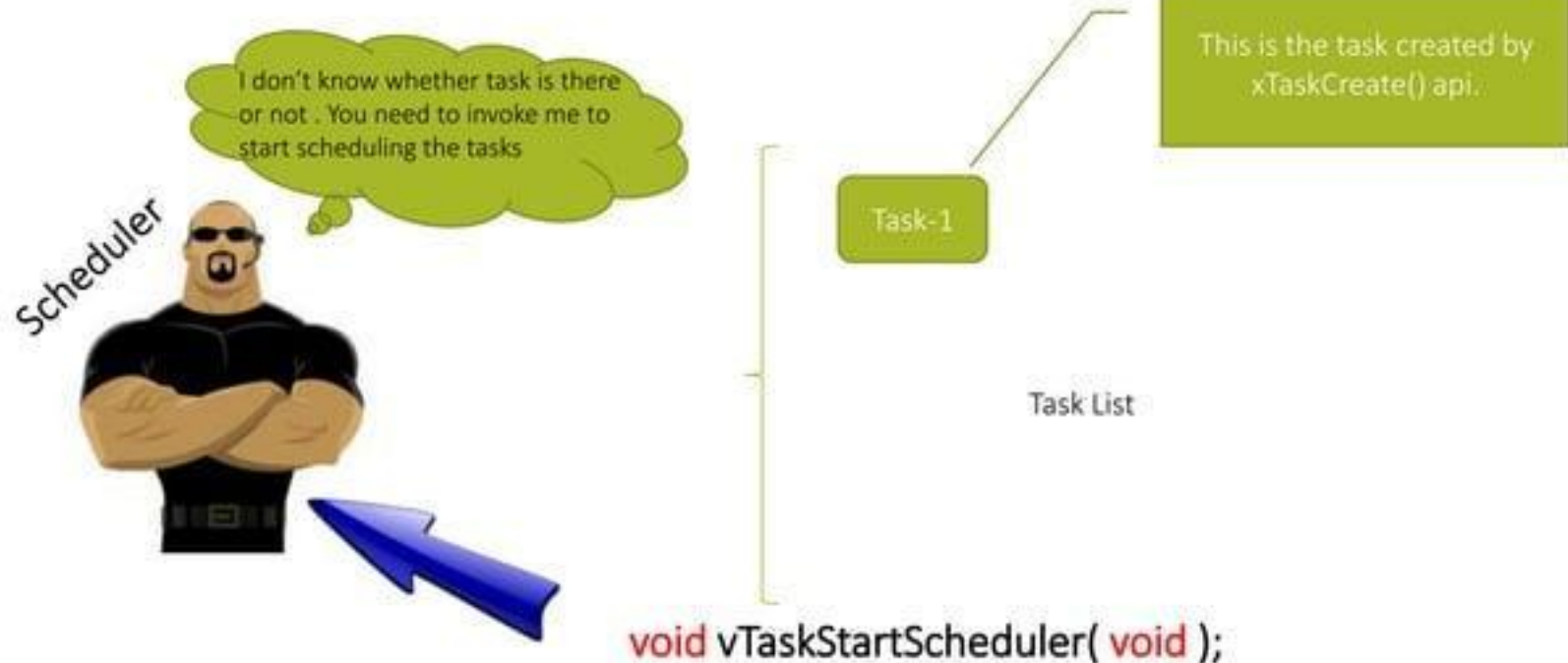
```
BaseType_t xTaskCreate ( TaskFunction_t pvTaskCode,  
                        const char * const pcName,  
                        unsigned short usStackDepth,  
                        void *pvParameters,  
                        UBaseType_t uxPriority,  
                        TaskHandle_t *pxCreatedTask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



```
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 );
```

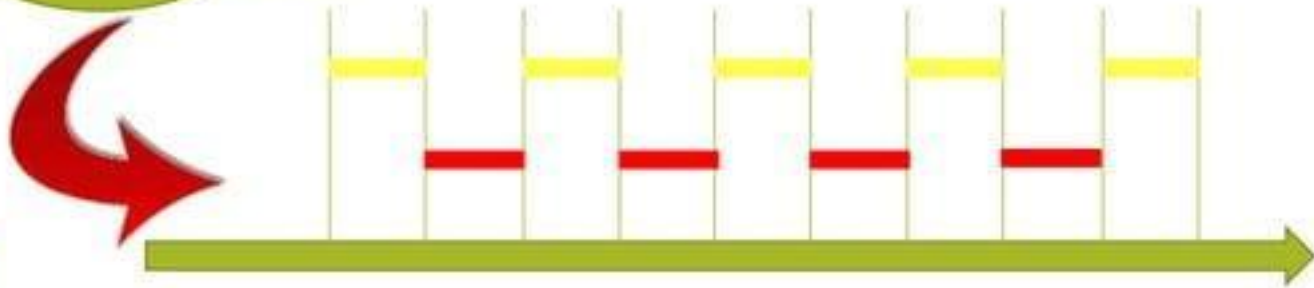
# API to Schedule a Task



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



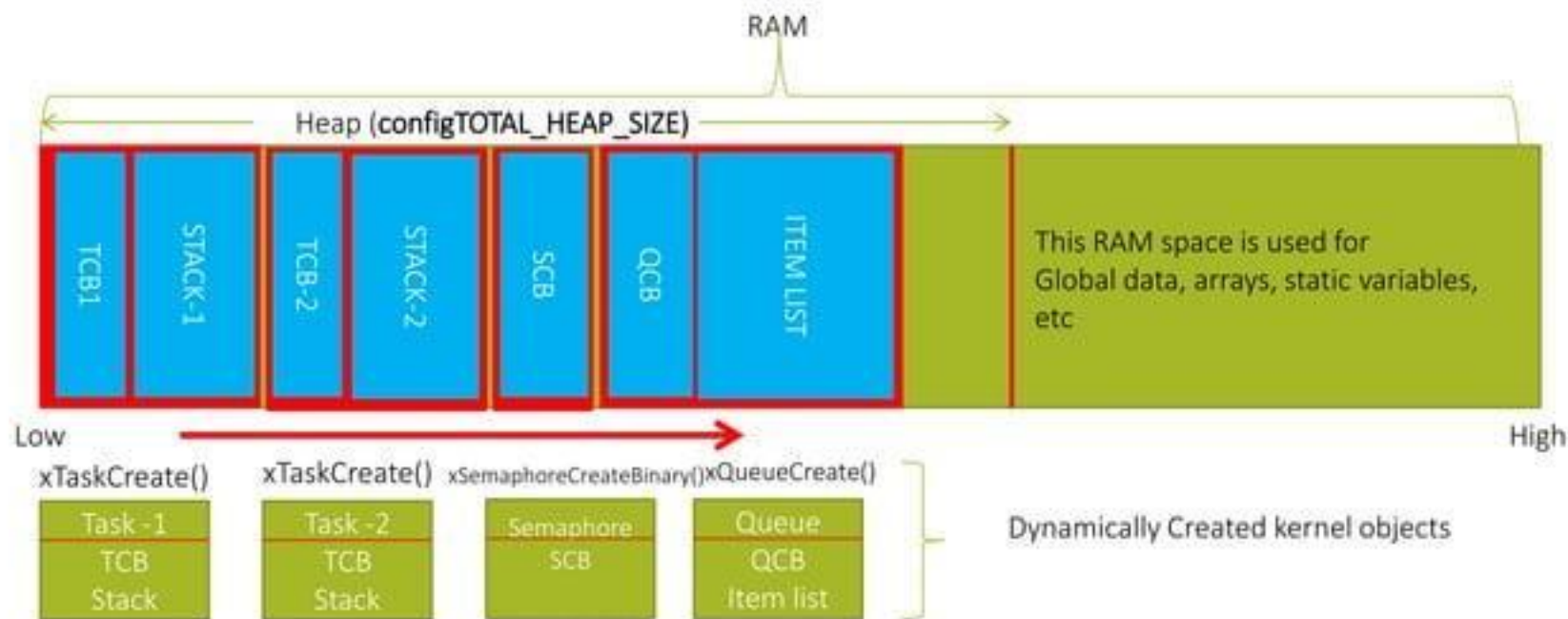
Scheduler





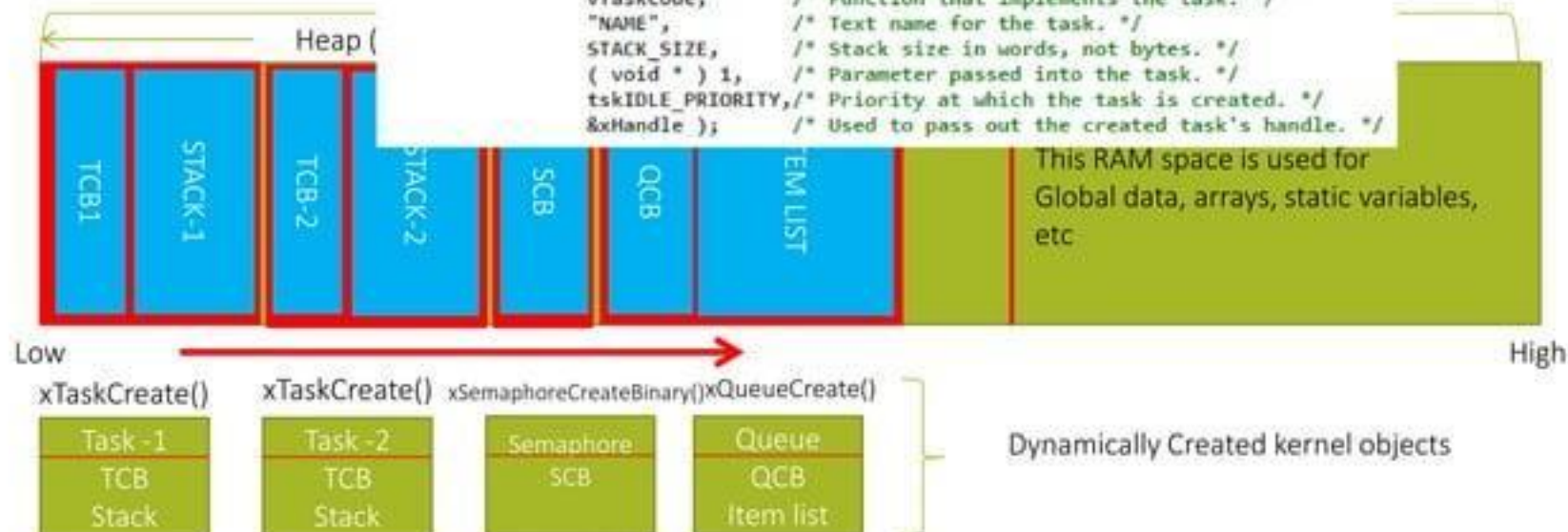
# FreeRTOS behind the scene TASK management

# What happens when you create a TASK?



# What happens when you create a TASK?

```
/* Create the task, storing the handle. */  
xReturned = xTaskCreate(  
    vTaskCode,      /* Function that implements the task. */  
    "NAME",         /* Text name for the task. */  
    STACK_SIZE,     /* Stack size in words, not bytes. */  
    ( void * ) 1,   /* Parameter passed into the task. */  
    tskIDLE_PRIORITY, /* Priority at which the task is created. */  
    &xHandle );      /* Used to pass out the created task's handle. */
```



# FreeRTOS Hello World Application and Testing on hardware

## 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

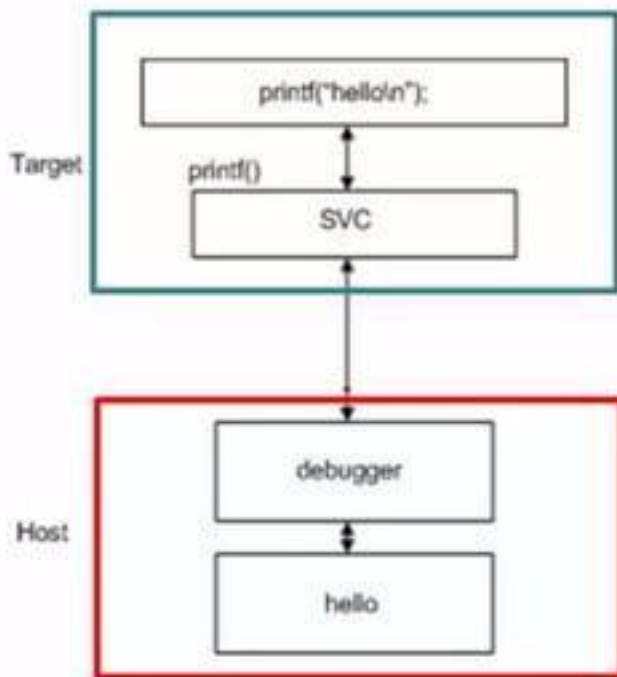
Ok, Now you created the FreeRTOS Project ,  
but at what clock frequency your main  
system clock of the MCU is running ?

You should know this in any RTOS project .

# Semi hosting and UART setup



# Semihosting Overview



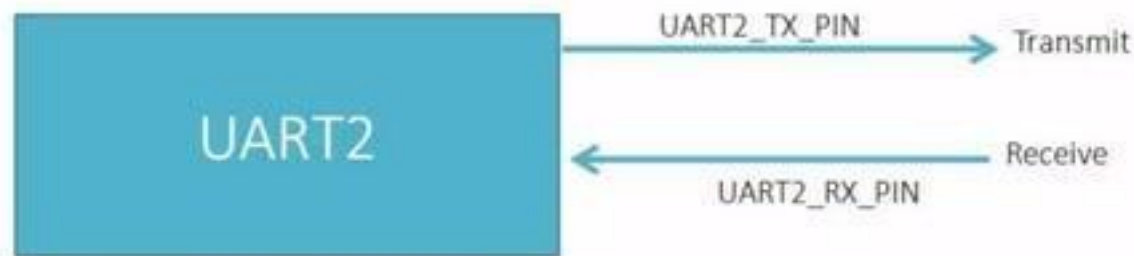
Target code wants to print "Hello"

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

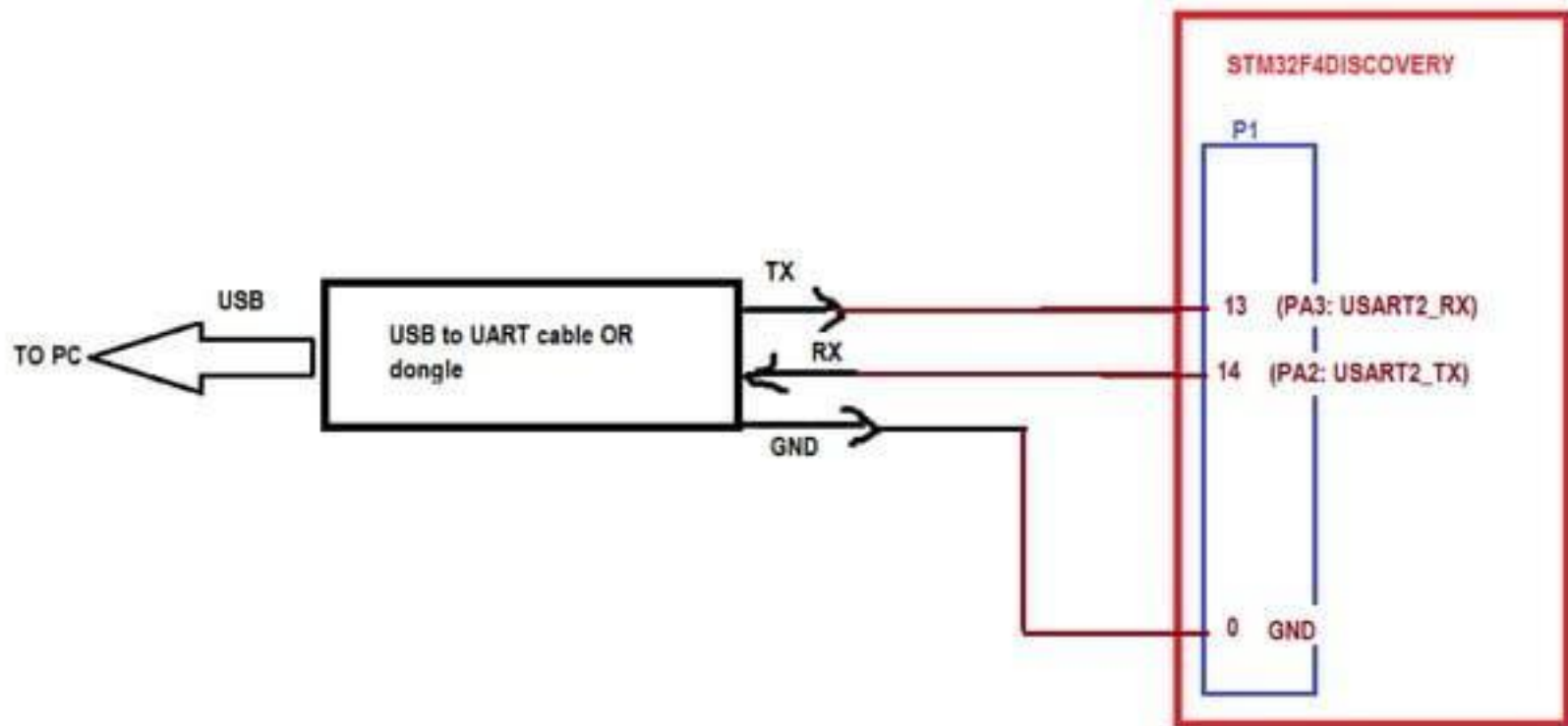
Prints "*hello*" on the HOST debugger's screen.

# 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*



# 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

HOST Software



SysView Target Code



# 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.

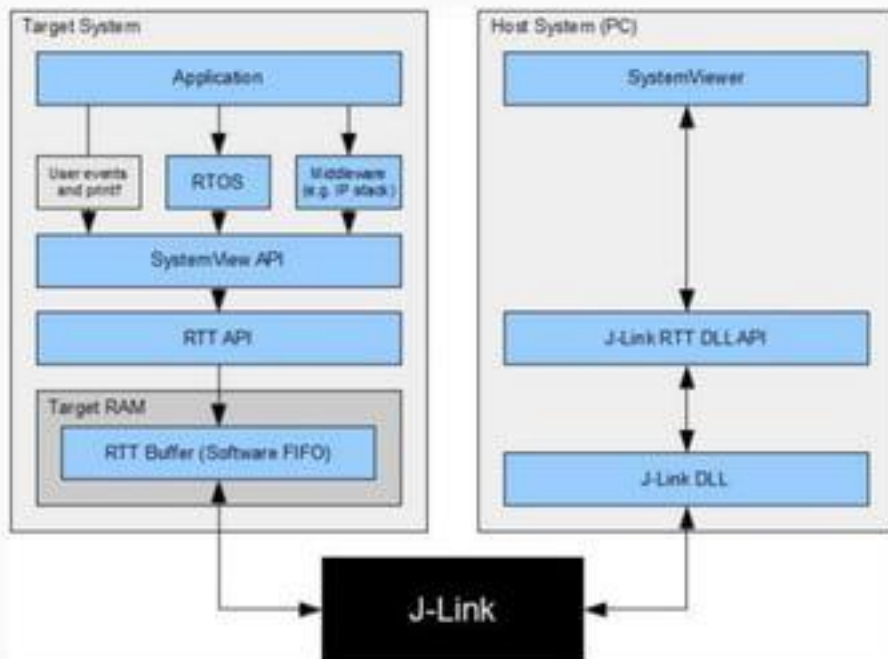
# SystemView Visualization modes

## 2 Single-shot recording:

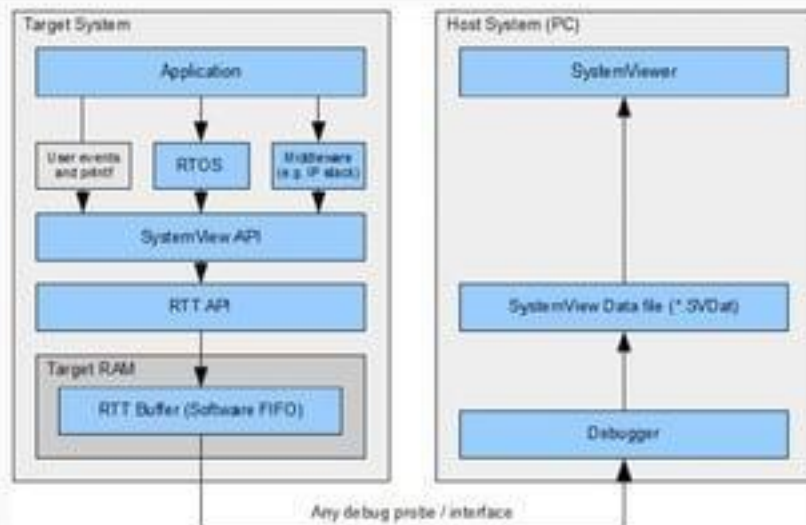
You need not to have JLINK or STLINK debugger for this.

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



# Single-shot recording



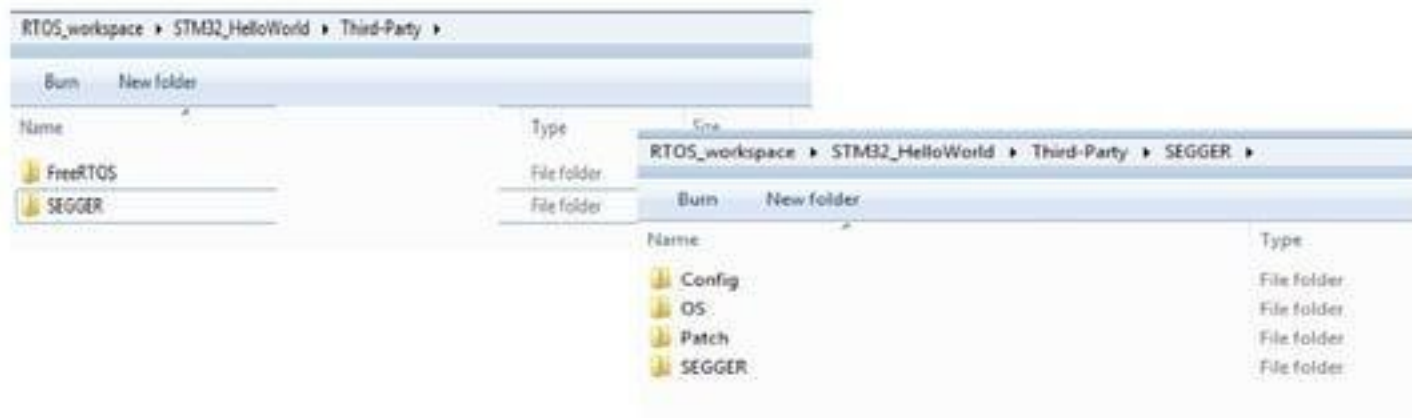
when no J-Link is used, SEGGER SystemView can be used to record data until its target buffer is filled. In single-shot mode the recording is started manually in the application, which allows recording only specific parts, which are of interest.

# 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



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 FreeRTOS 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.

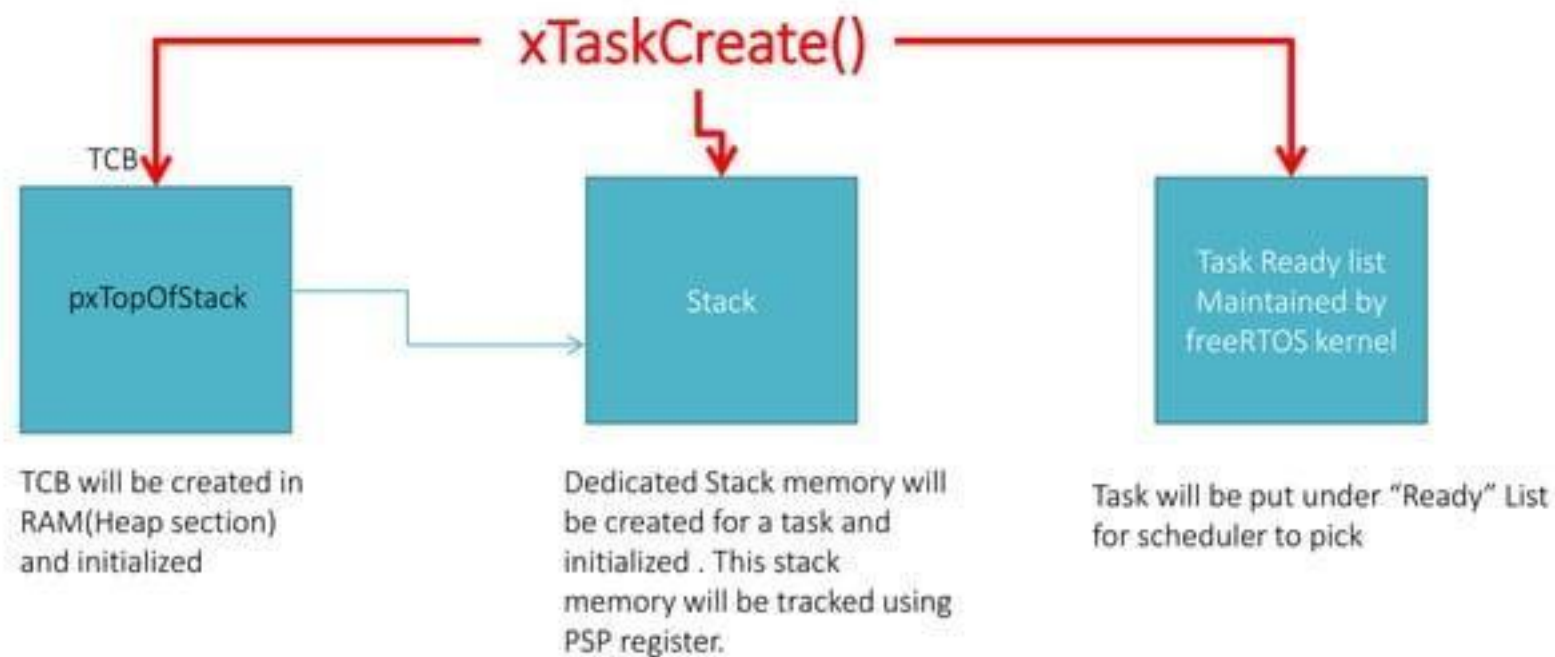
## STEP8: Collect the recorded data(RTT buffer)

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

Continues recording on STM32 STLINK based boards

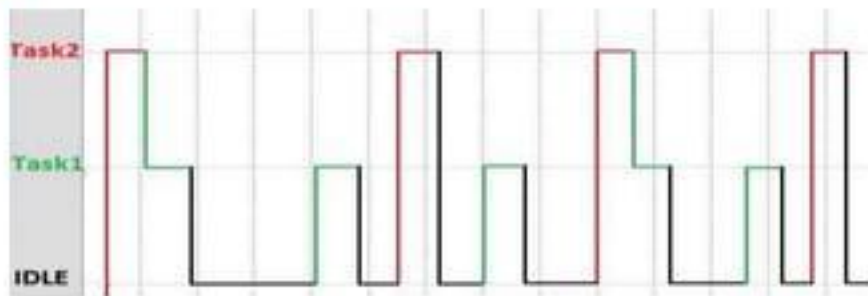
1. Flash the J-link firmware on your board. ( this step basically removes the stlink frimware present in your board with Jlink firmware)
2. open the SystemView HOST software
3. Go to Target->Start recording
4. Mention "Target Device" and RTT address details and click "OK"
5. Software will record the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

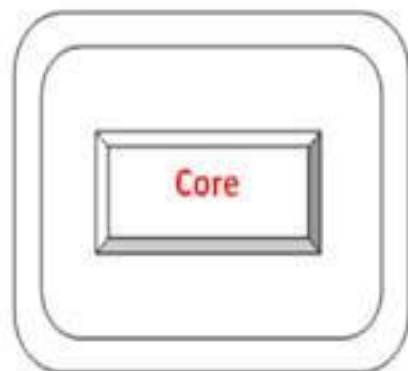


# FreeRTOS Timer Services Task

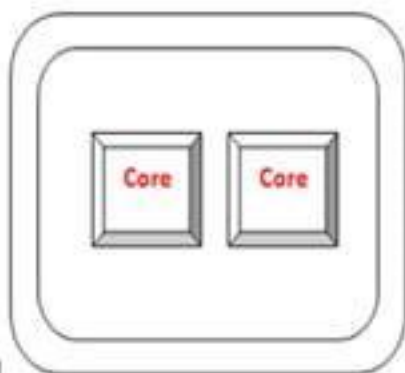
# Timer Services Task (Timer\_svc)

- ✓ 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

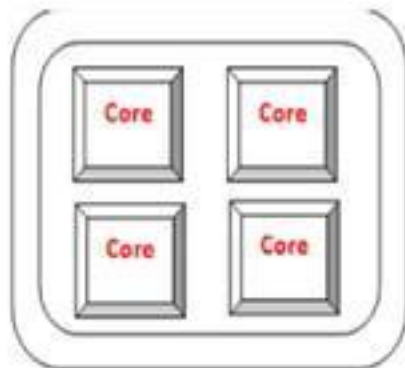
# Overview of FreeRTOS Scheduler



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

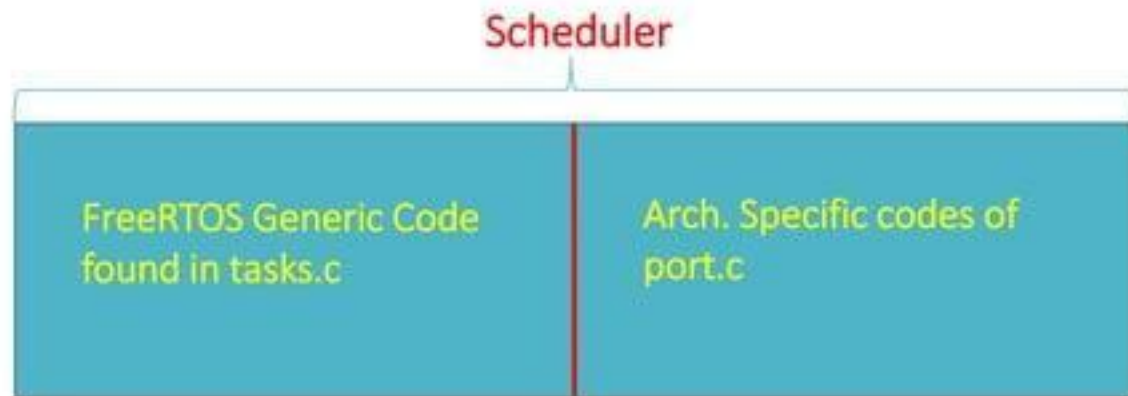


# FreeRTOS Scheduling

- ✓ 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

Generic Code for All Arch.

`vTaskStartScheduler()`

Arch. Specific code

`xPortStartScheduler()`

Configuring the SysTick timer to issue interrupts at desired rate (as configured in the config item `configTICK_RATE_HZ` in `FreeRTOSConfig.h`)

Configures the priority for PendSV and SysTick interrupts.

Starts the first task by executing the SVC instruction

# vTaskStartScheduler()

Generic Code for All Arch.

vTaskStartScheduler()

task.c



Arch. Specific code

xPortStartScheduler()

port.c

xPortStartScheduler() implementation  
is different for different processor  
architectures

# FreeRTOS Kernel Interrupts and Scheduling of Tasks



# 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.

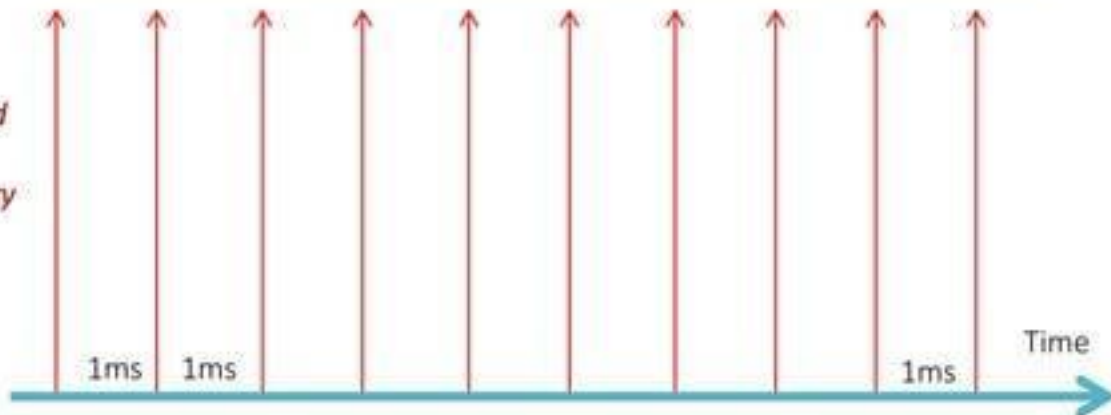
# RTOS Tick ( The heart beat)



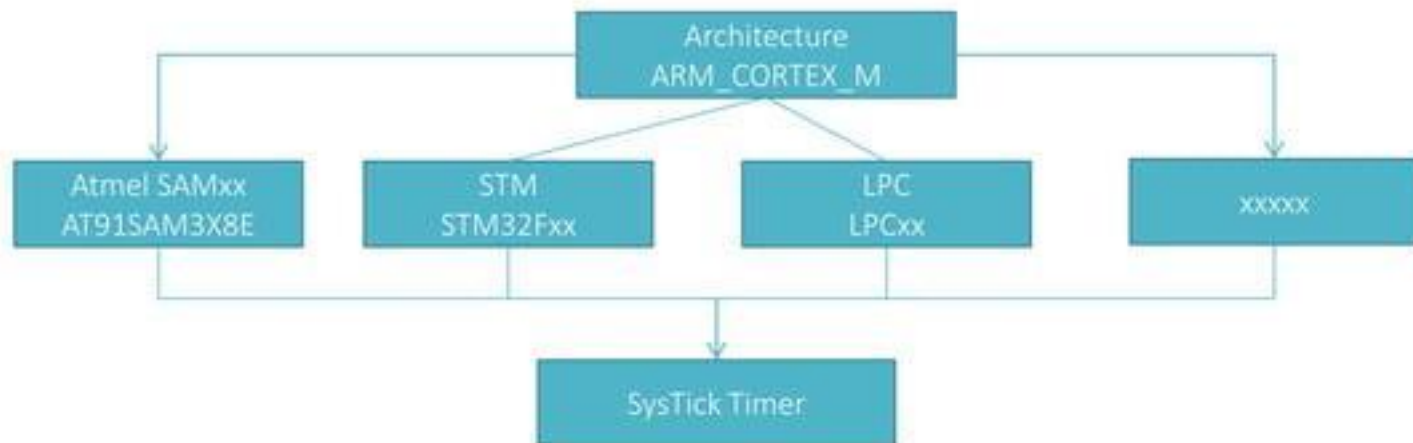
# The RTOS Tick

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

*Interrupt fired  
by SysTick  
timer for every  
1ms*



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 ?

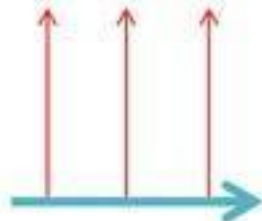


# The RTOS Tick- Why it is needed ?

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.

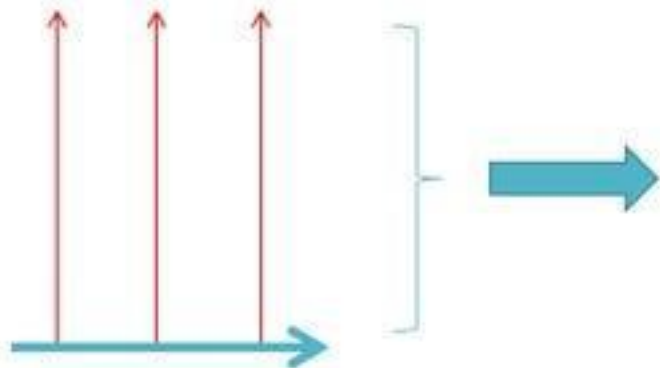
Each timer Tick interrupt  
makes scheduler to run



# The RTOS Tick- Why it is needed ?

Used for Context Switching to the next potential Task

Each timer Tick interrupt makes scheduler to run

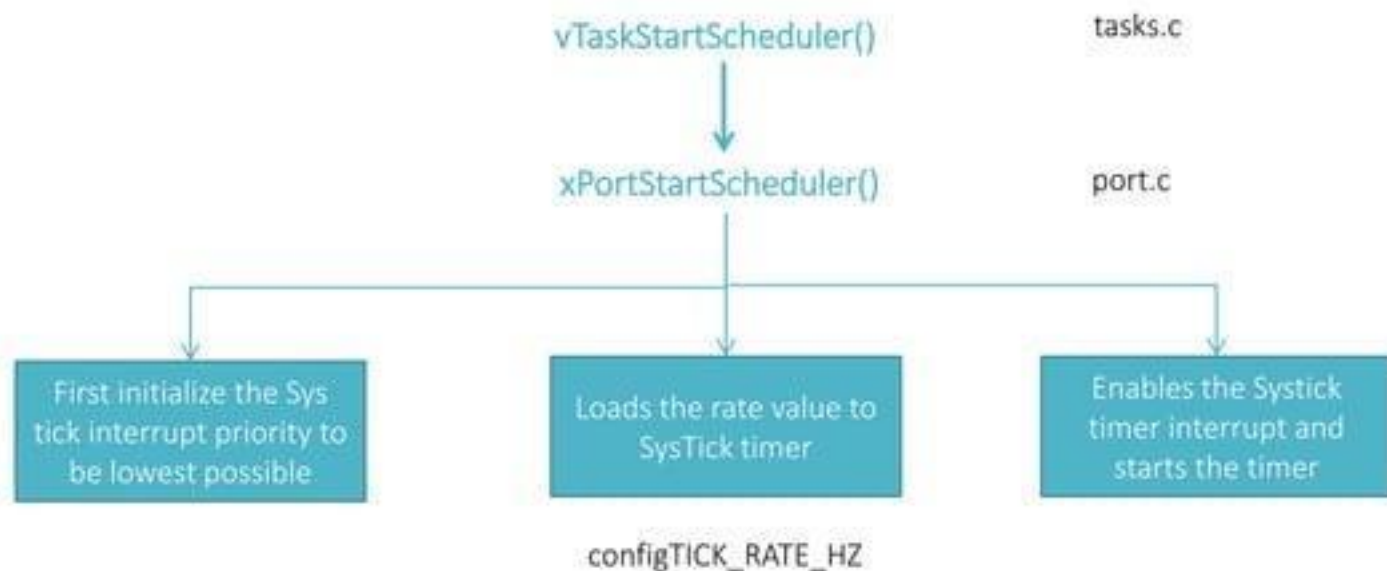


1. The tick ISR runs
2. All the ready state tasks are scanned
3. Determines which is the next potential task to run
4. If found, triggers the context switching by pending the PendSV interrupt
5. The PendSV handler takes care of switching out of old task and switching in of new task



Who configures RTOS  
tick timer (SysTick)?

# Who configures RTOS tick Timer ?



# The RTOS Tick Configuration

`configSYSTICK_CLOCK_HZ = configCPU_CLK_HZ`

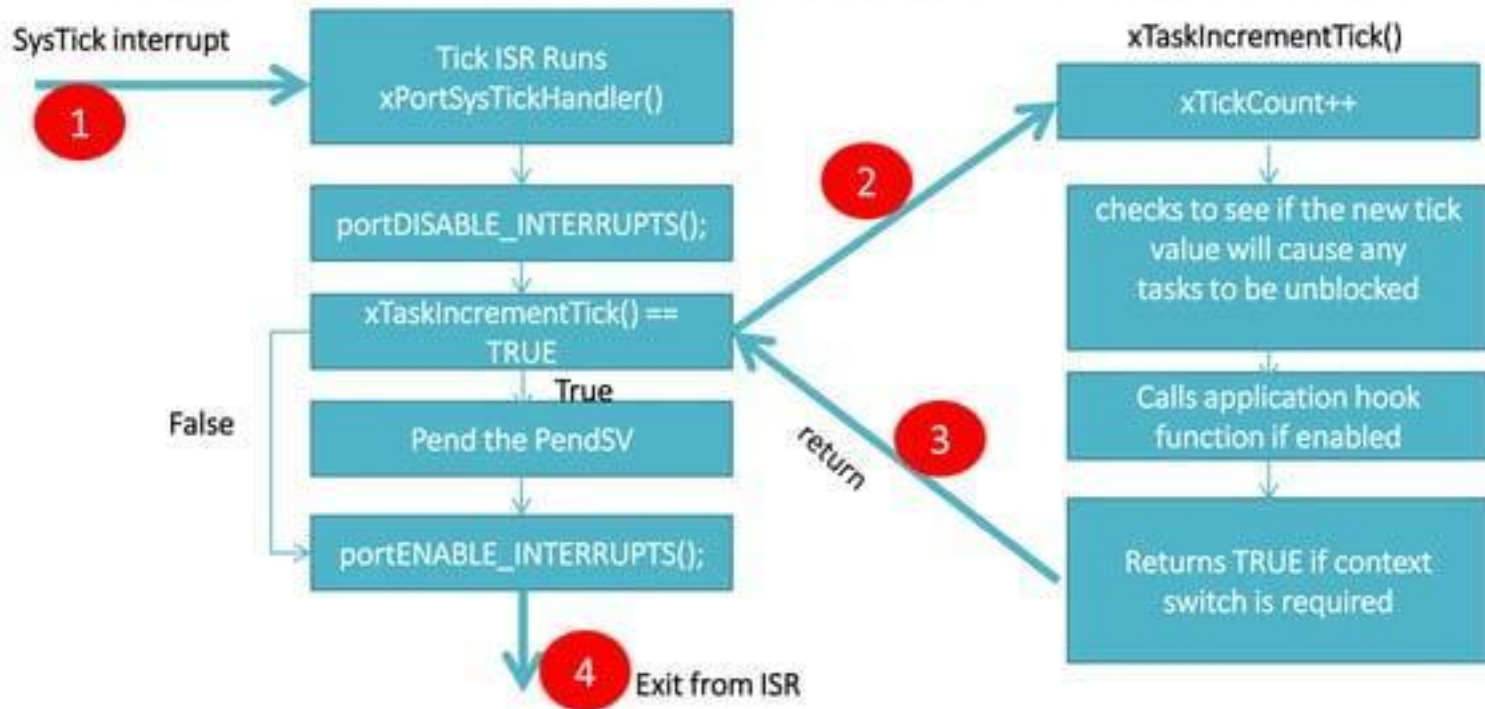
If `configCPU_CLK_HZ = 16000000` ( i.e 16Mhz )

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

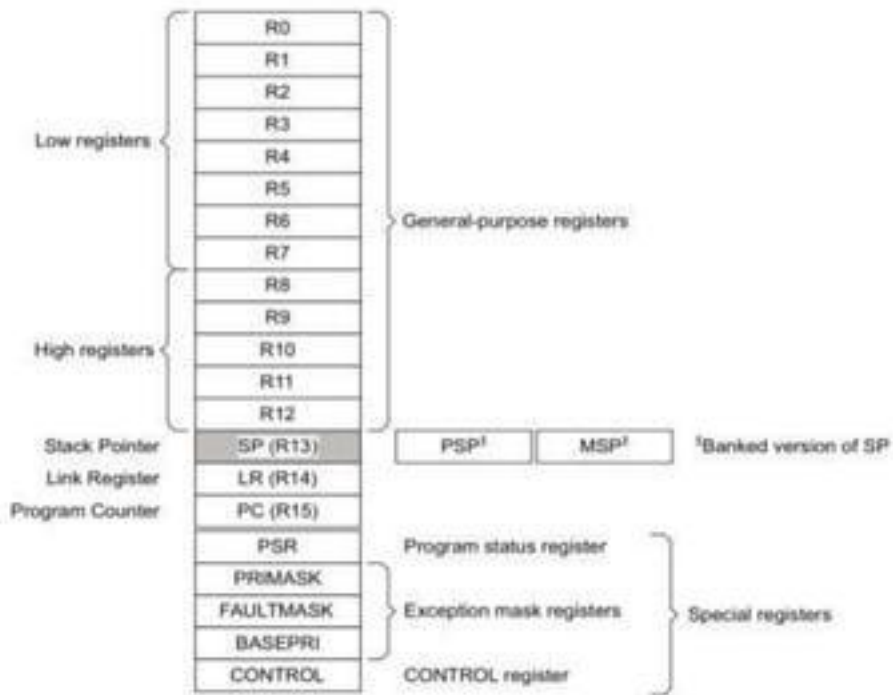
When a task executes on the Processor it utilizes

- ✓ 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

Task's Private stack  
(Process stack)

SP(PSP)  
→

PUSH and POP to this stack  
area is tracked by PSP  
Register of ARM

When Task executes it does PUSH and POP here

Kernel Stack  
(Main stack)

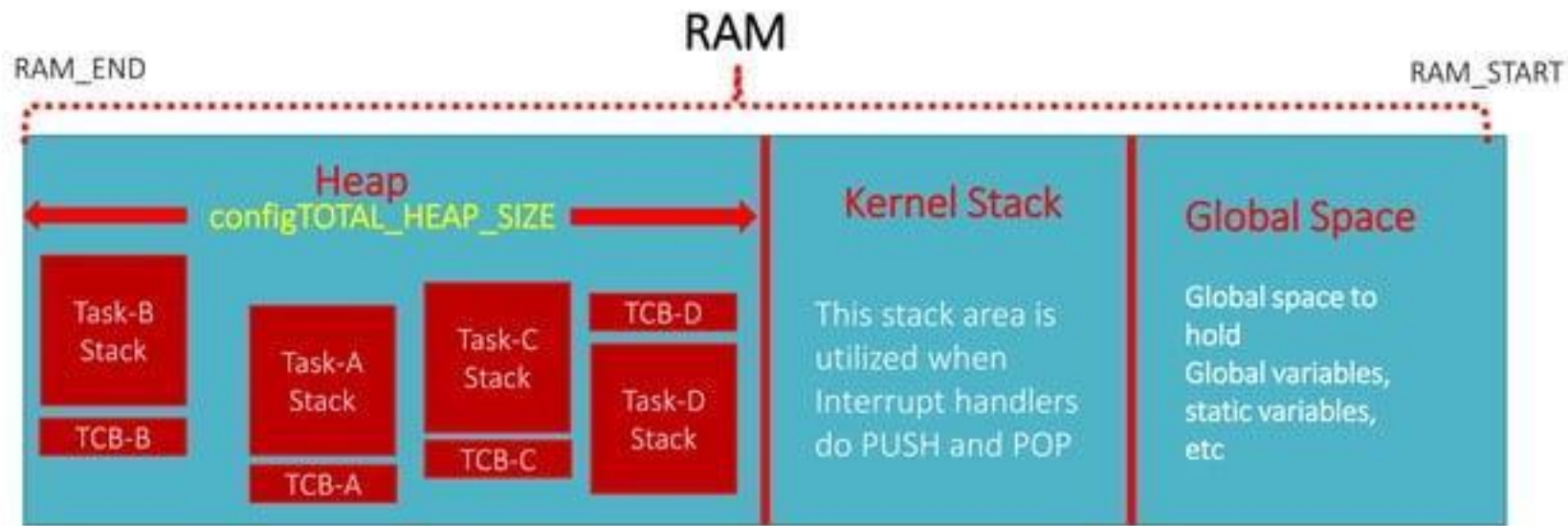
SP(MSP)  
→

PUSH and POP to this stack  
area is tracked by MSP  
Register of ARM

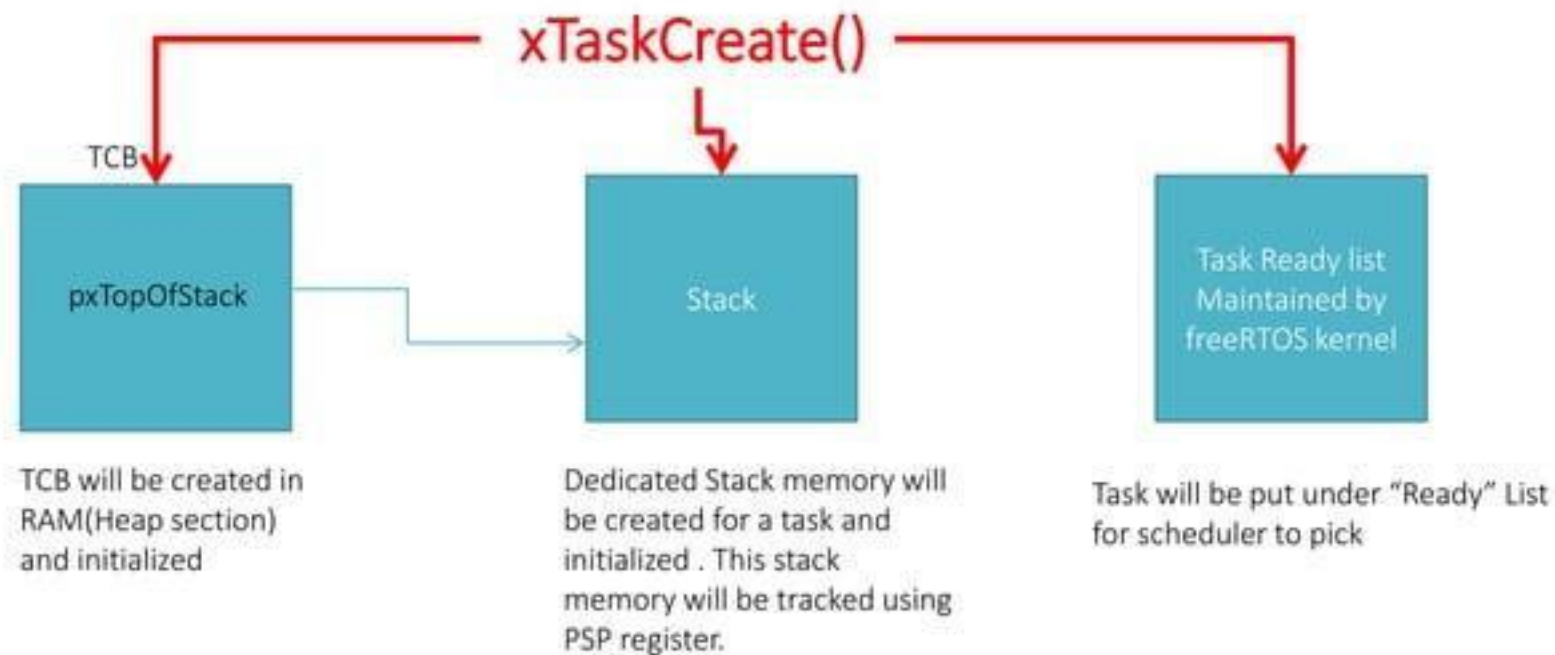
When ISR executes it does PUSH and POP here

# Stacks

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

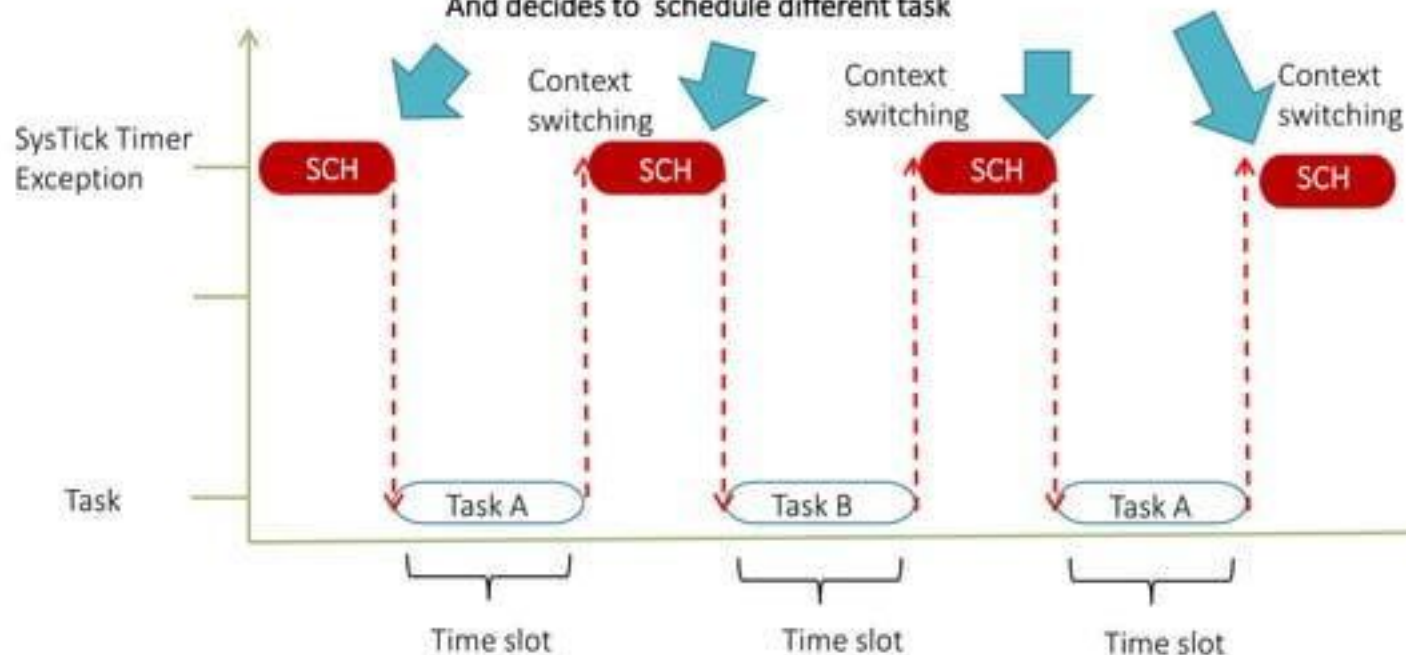


# Task Creation



# Context Switching with animation

OS code runs on each systick timer exception  
And decides to schedule different task



# 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** )

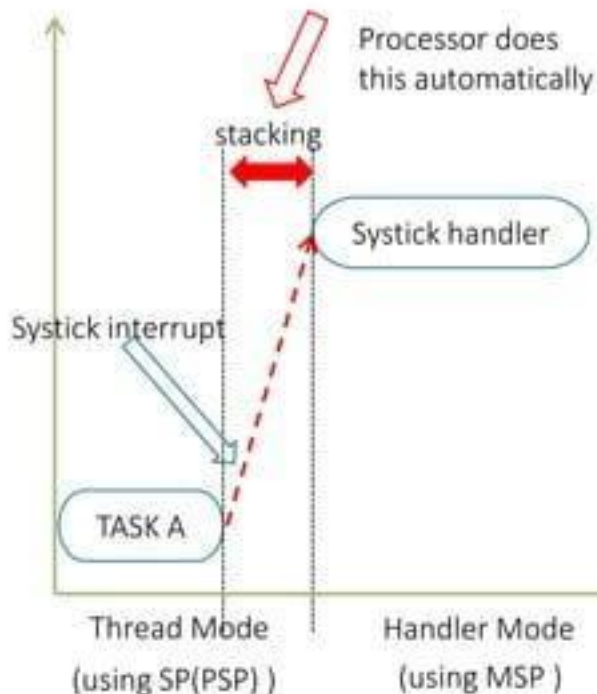
## Task switching out procedure

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



Task-A 's Private Stack memory

PSP

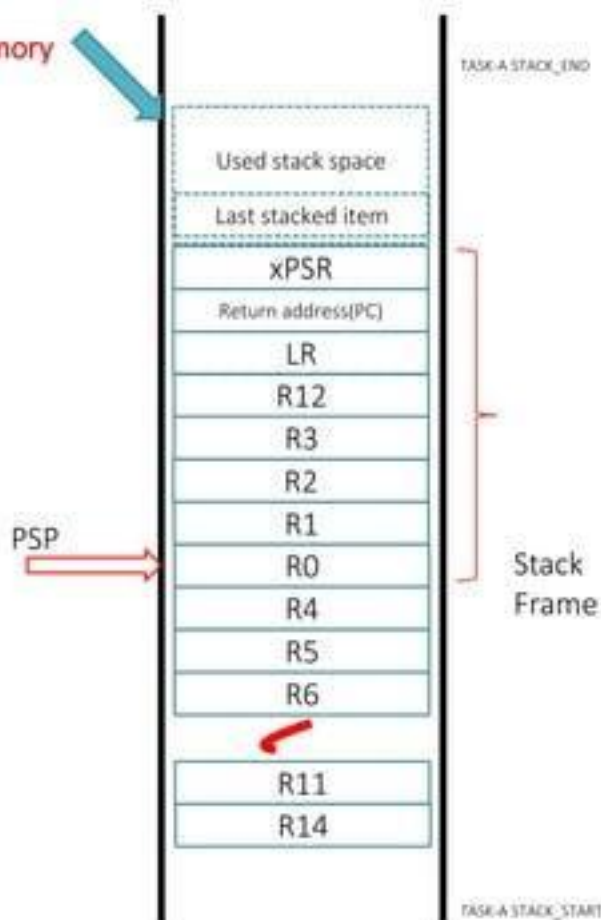
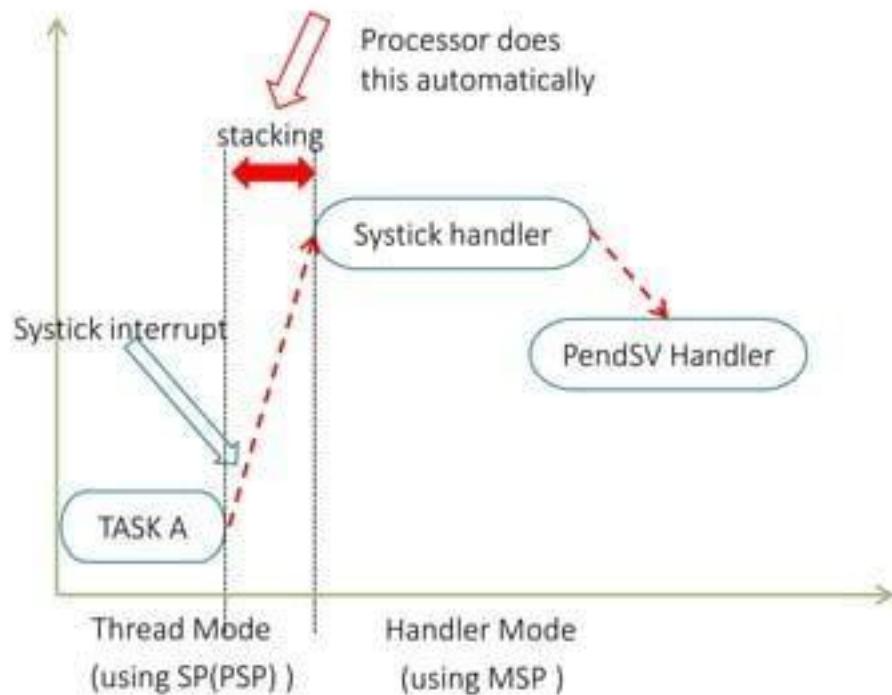


Stack  
Frame

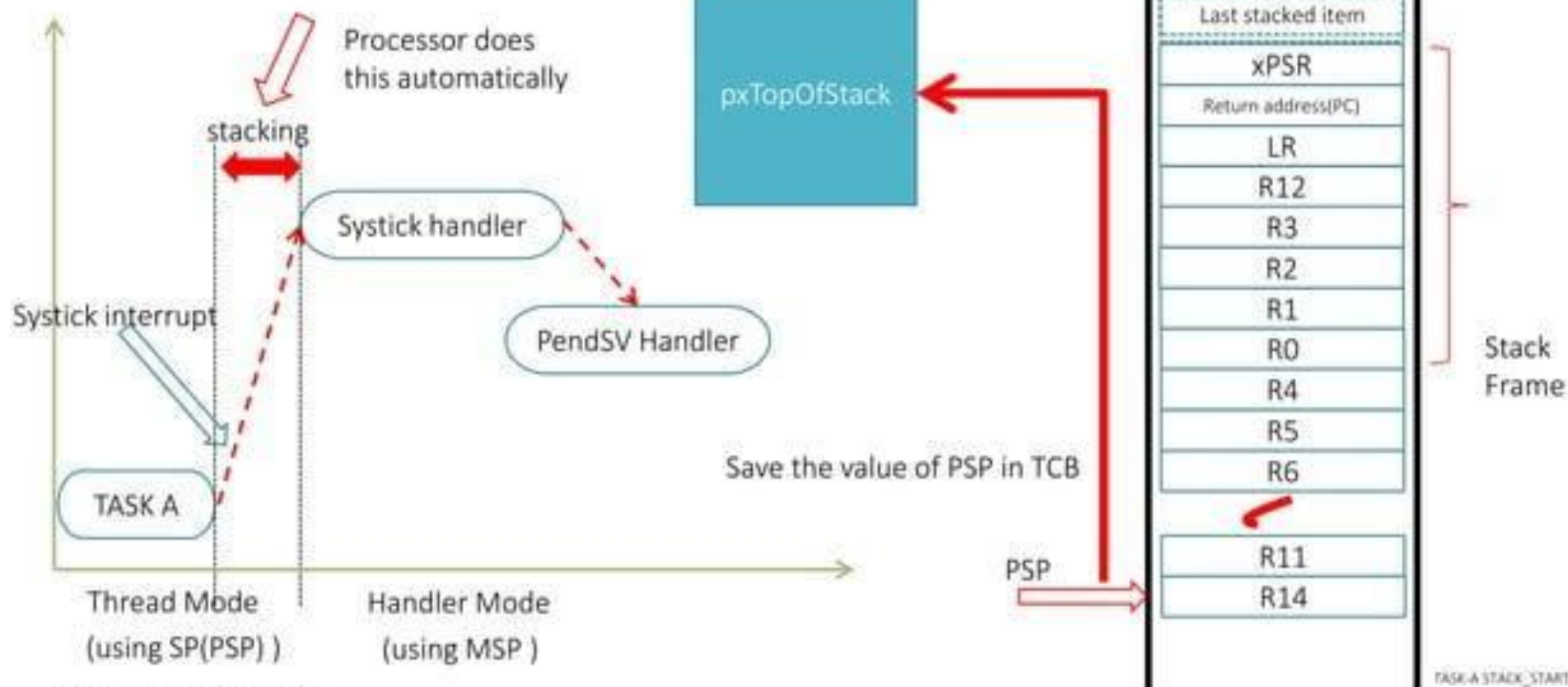


# 3 Save core registers

Task-A's Private Stack memory



# 4 Save PSP Into TCB

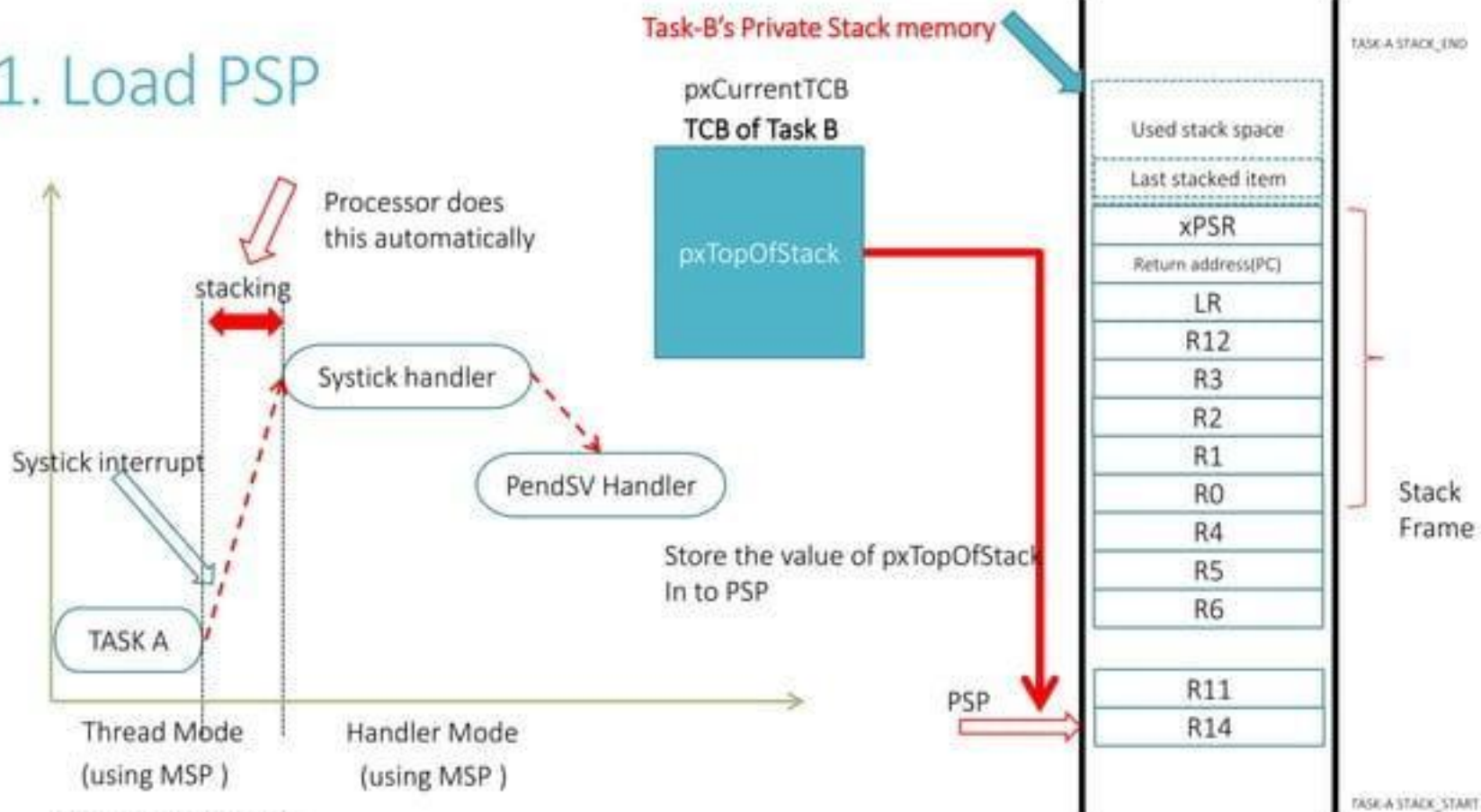


## Task Switching In Procedure

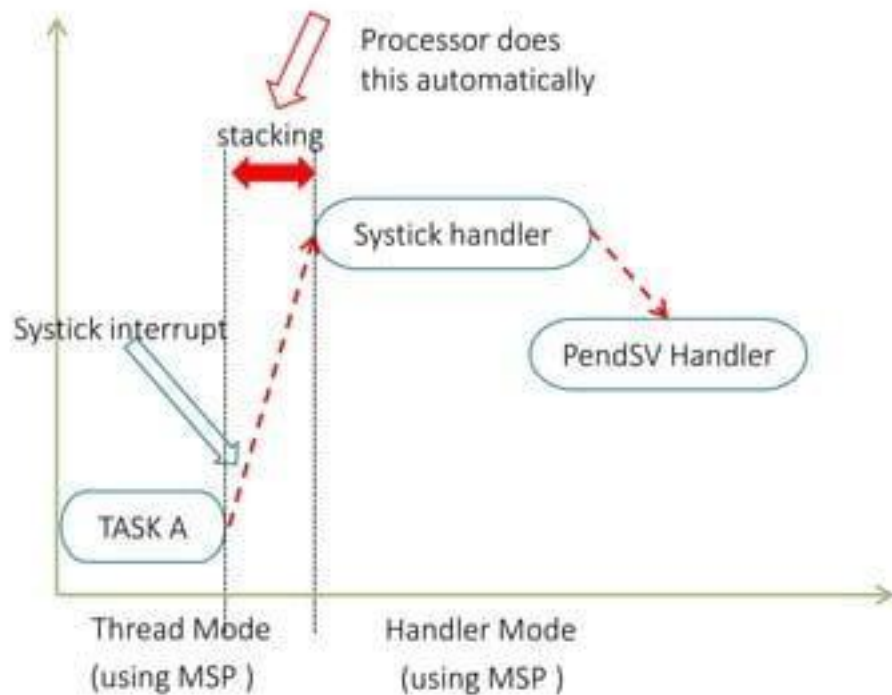
So, at this time, we already know which task(TCB) should be switched in .That means new switchable task's TCB can be accessed by **pxCurrentTCB**

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

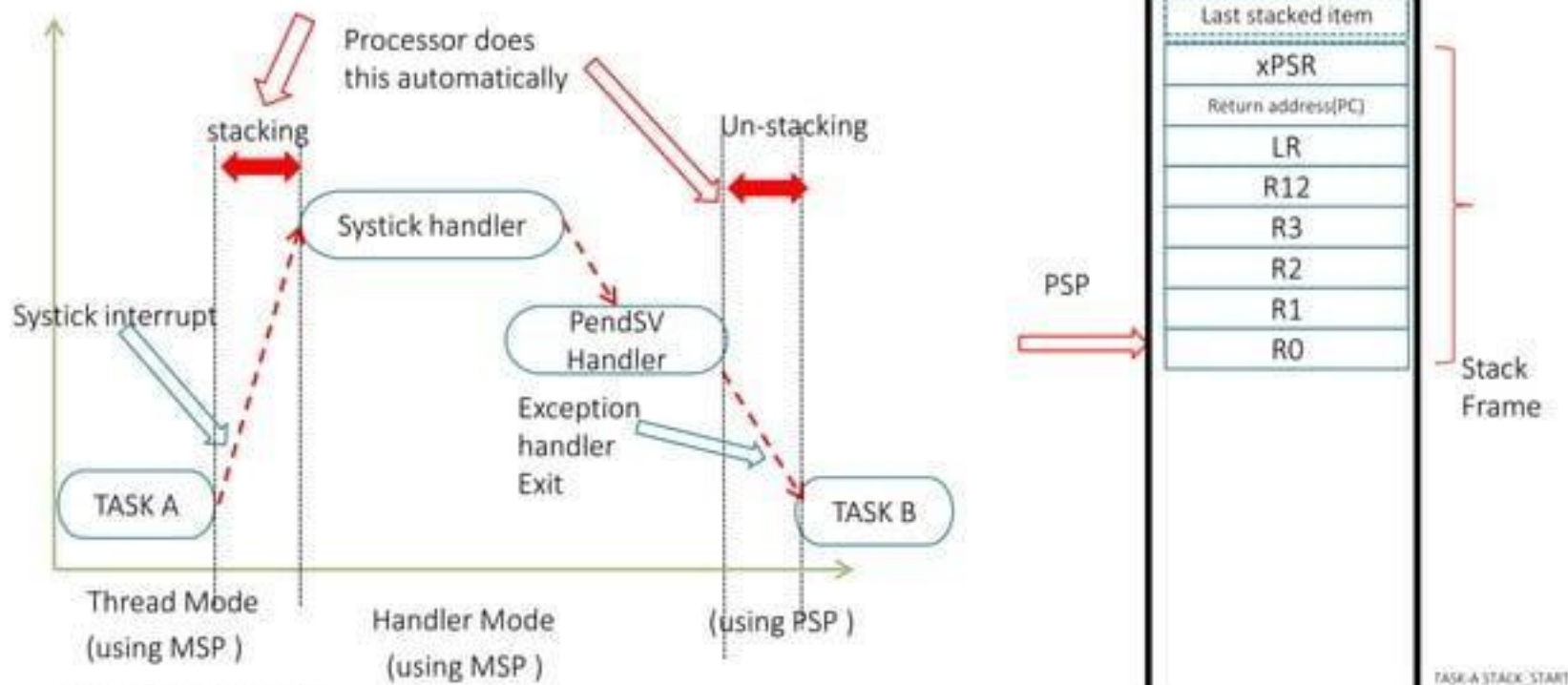


PSP

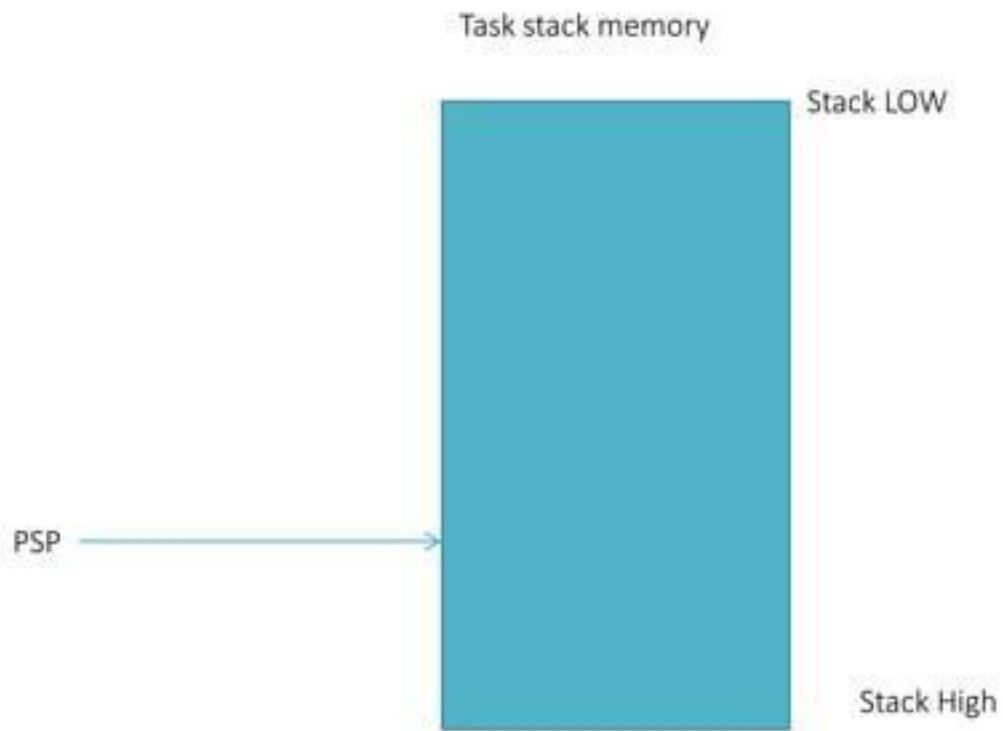
Stack  
Frame

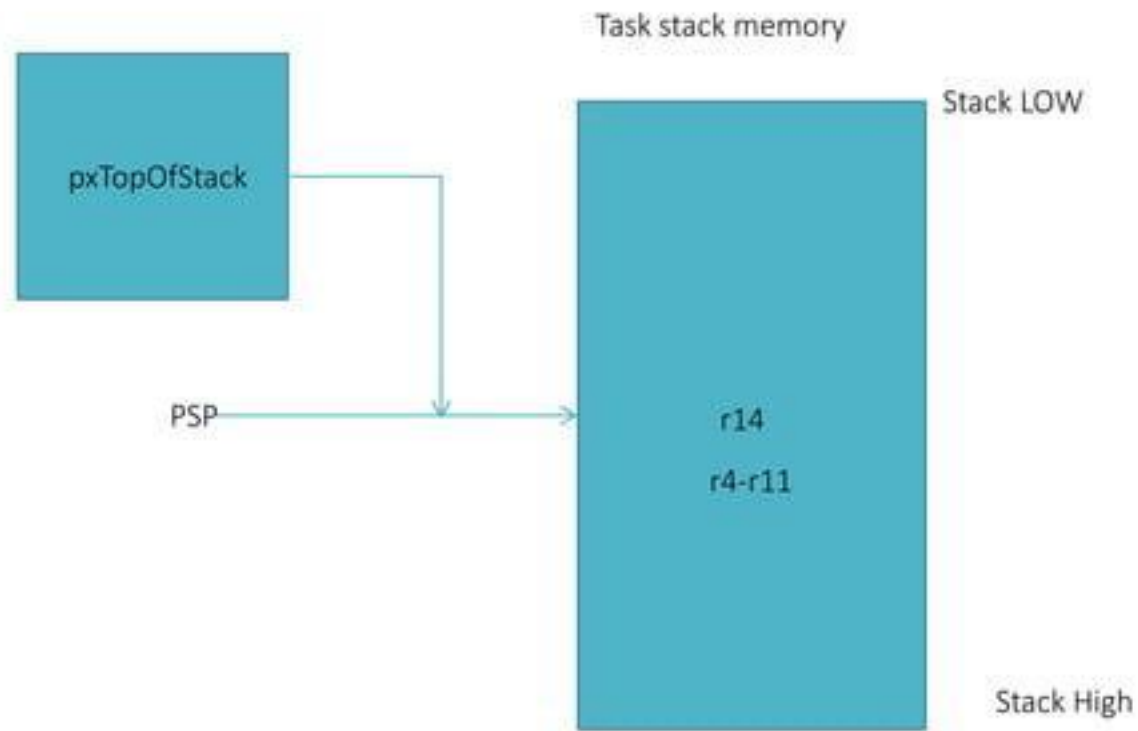


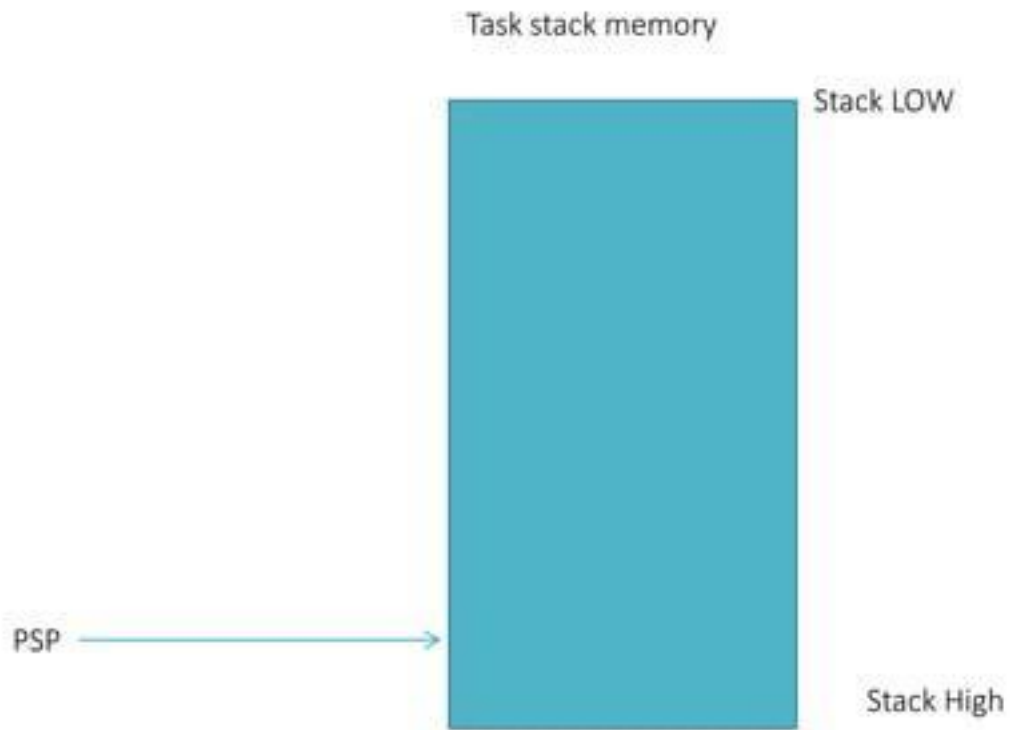
### 3. Exception Exit











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](http://www.fastbitlab.com)