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 Mircocontrollers

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

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.

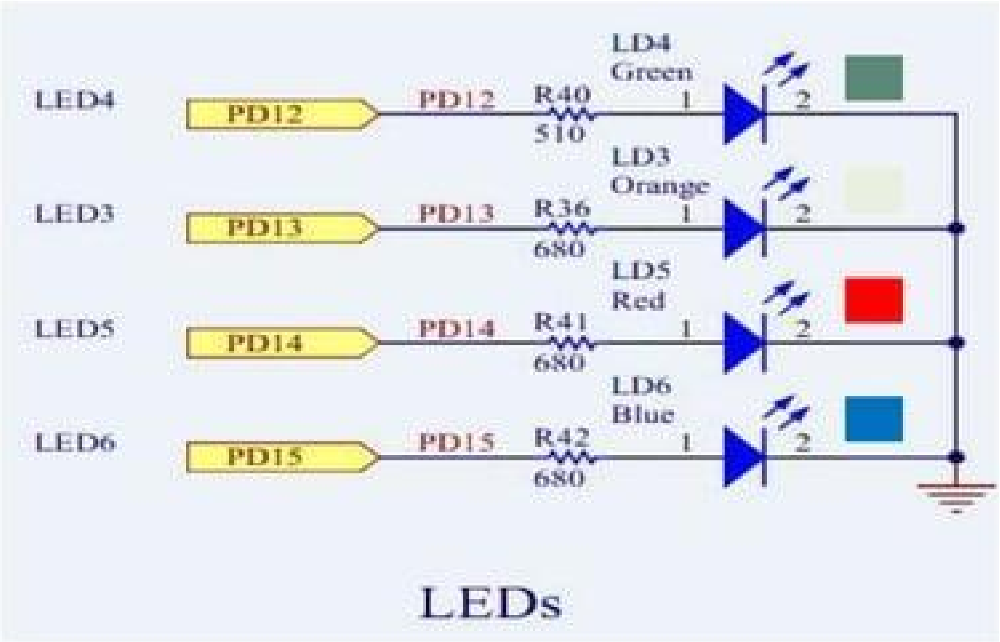
Use same freeRTOS task priorities for both the tasks.

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 exaclty the button and LEDs are connected on your board.

Discovery Board LEDs

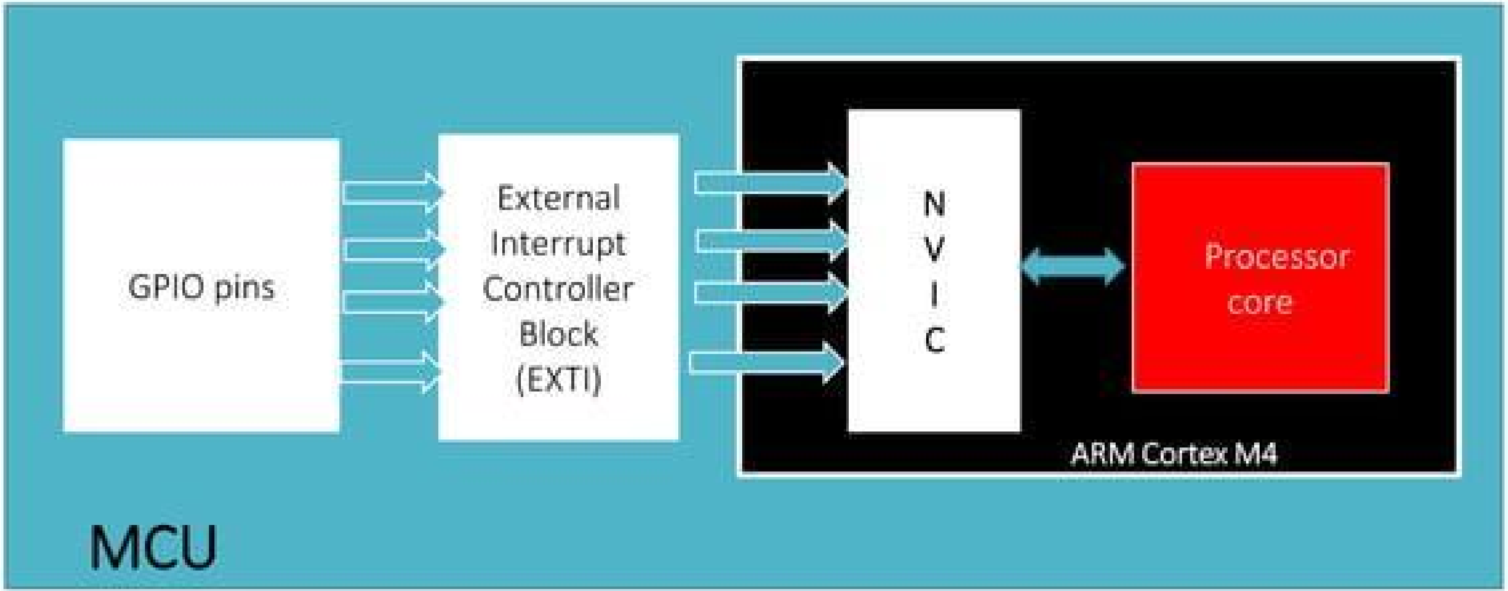


Exercise

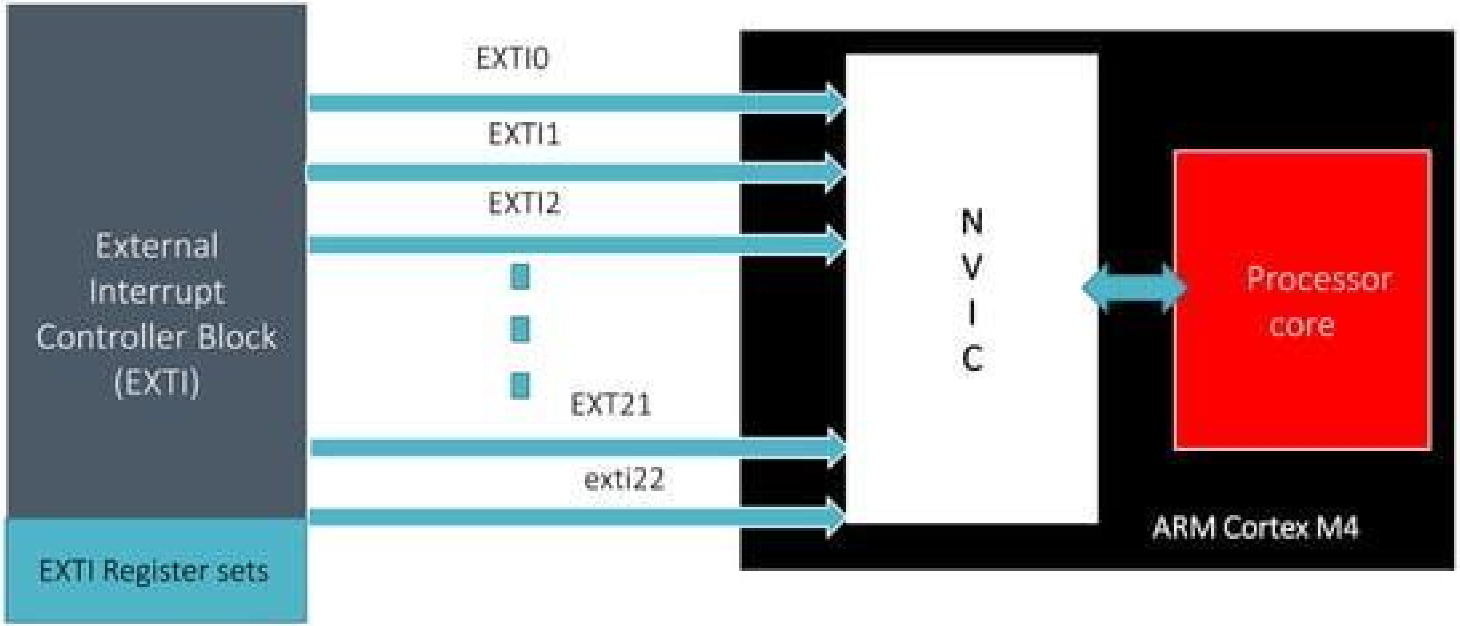
Write a FreeRTOS application which creates only 1 task : *led_task*

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


External Interrupt Controller Block (EXTI)
EXTI Register sets
EXTI0
EXTI1
EXTI2
EXT21
exti22
N V I C
Processor core
ARM Cortex M4

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

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 * configTICK_RATE_HZ) / 1000

Example :

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

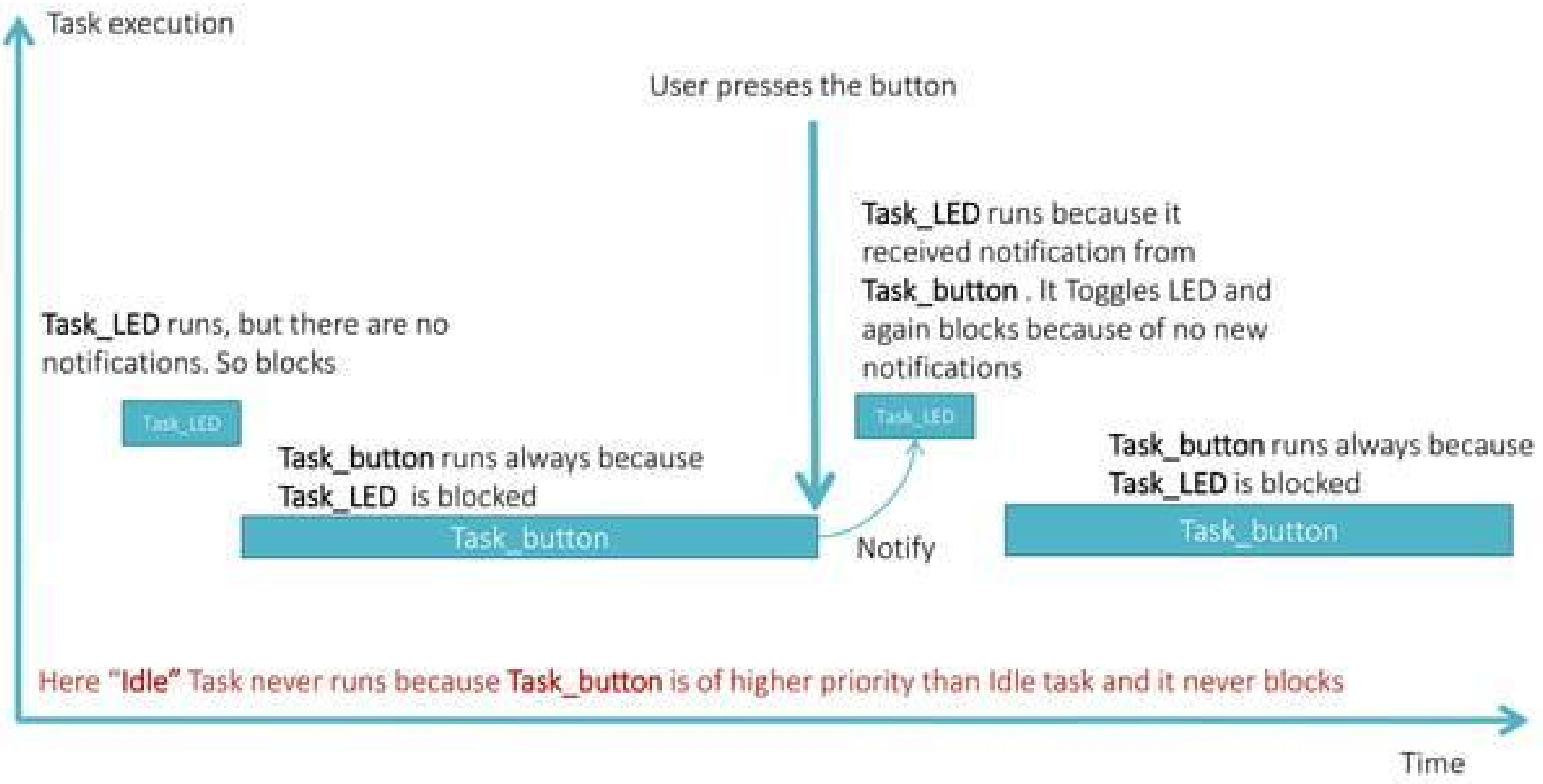
So, 500ms of delay is equivalent to 250 ticks duration

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** .


Task execution
User presses the button
Task_LED runs because it received notification from Task_button . It Toggles LED and again blocks because of no new notifications
Task_LED runs, but there are no notifications. So blocks
Task_LED
Task_LED
Task_button runs always because Task_LED is blocked
Task_button runs always because Task_LED is blocked
Task_button
Notify
Task_button
Here "Idle" Task never runs because Task_button is of higher priority than Idle task and it never blocks
Time

FreeRTOS: Licensing

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

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

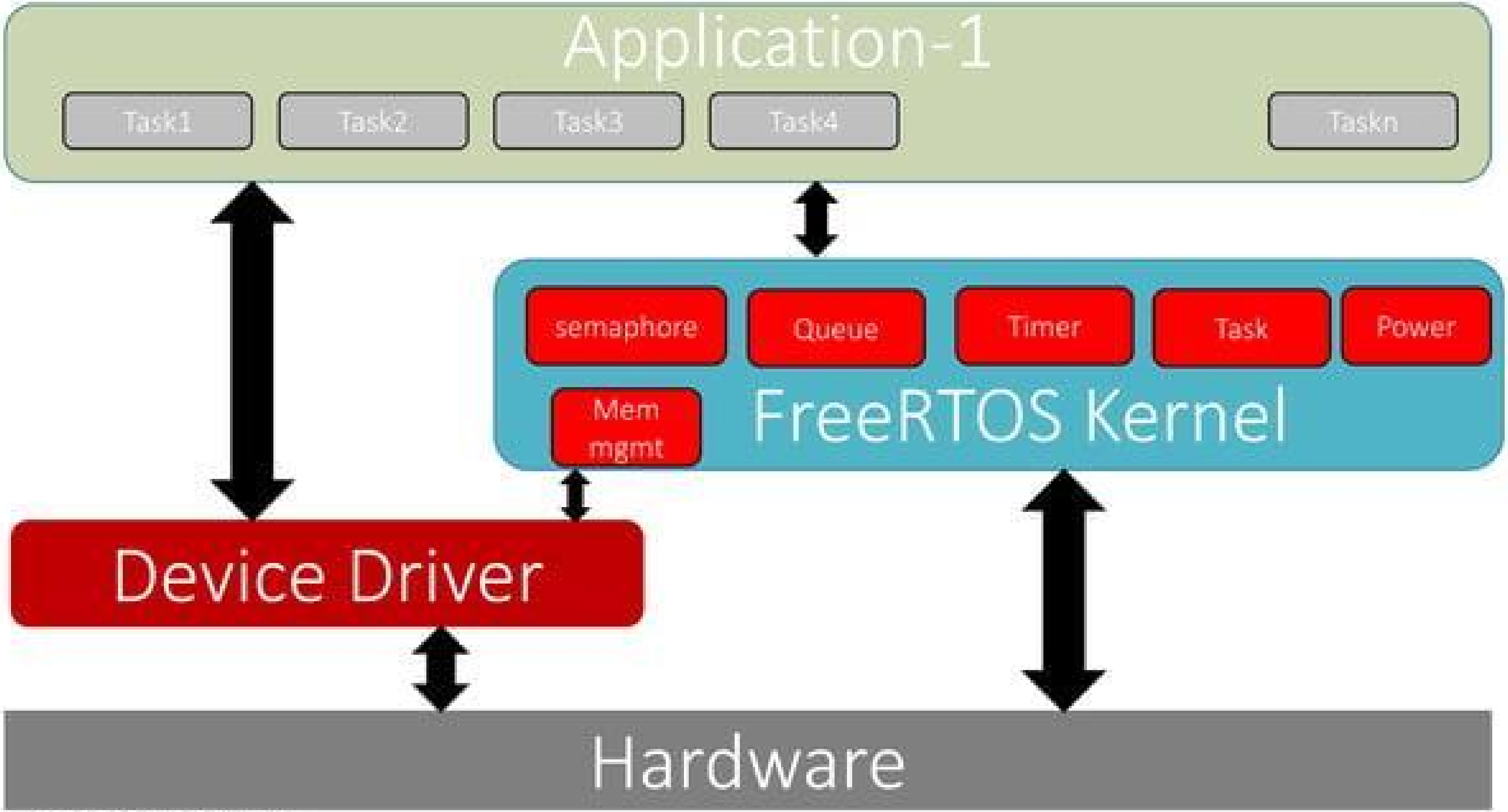
SAFERTOS™ is a derivative version of FreeRTOS that has been analyzed, documented and tested to meet the stringent requirements of the IEC 61508 safety standard. This RTOS is audited to verify IEC 61508 SIL 3 conformance.

OpenRTOS™ 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


Application-1
Task1
Task2
Task3
Task4
Taskn
semaphore
Queue
Timer
Task
Power
Mem
mgmt
FreeRTOS Kernel
Device Driver
Hardware

Very important links

Download :
www.freertos.org

FreeRTOS Tutorial Books
http://shop.freertos.org/FreeRTOS_tutorial_books_and_reference_manuals_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/FreeRTOS-quick-start-guide.html

Books and kits
http://shop.freertos.org/RTOS_primer_books_and_manual_s/1819.htm

Overview of FreeRtos Memory Management

RAM and Flash

Every Microcontroller Consists of two types of memories : RAM and Flash
Usually RAM memory always less than FLASH memory

RAM and Flash

RAM

STM32F401RE

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

FLASH

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

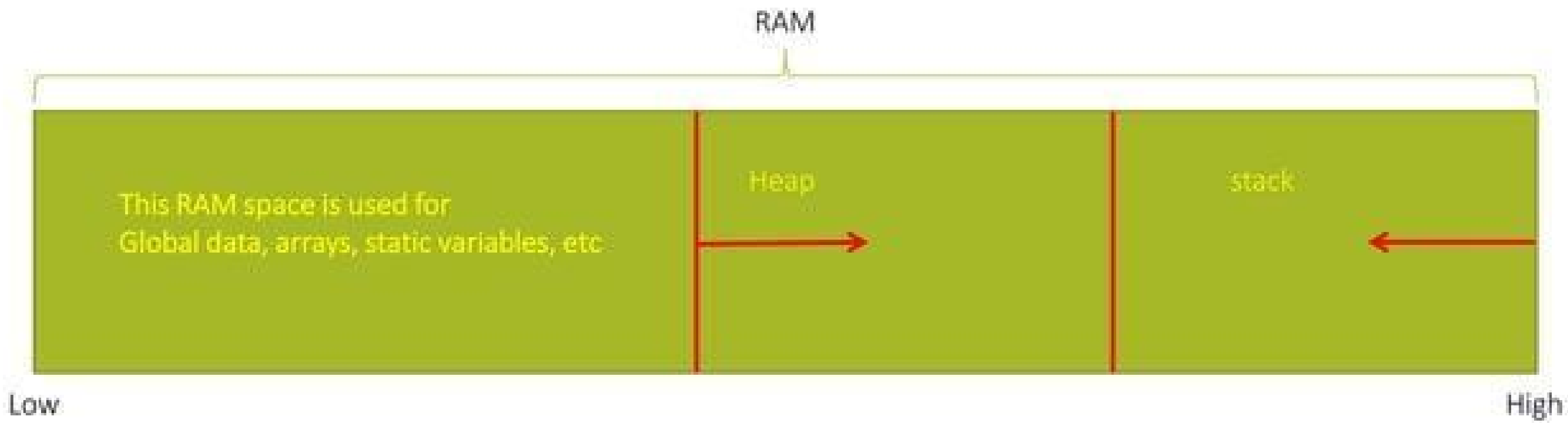
FLASH

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

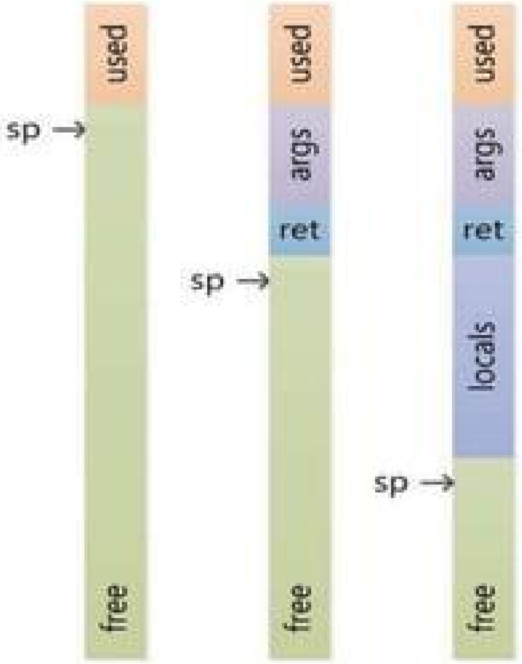


Stack and Heap in embedded Systems

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

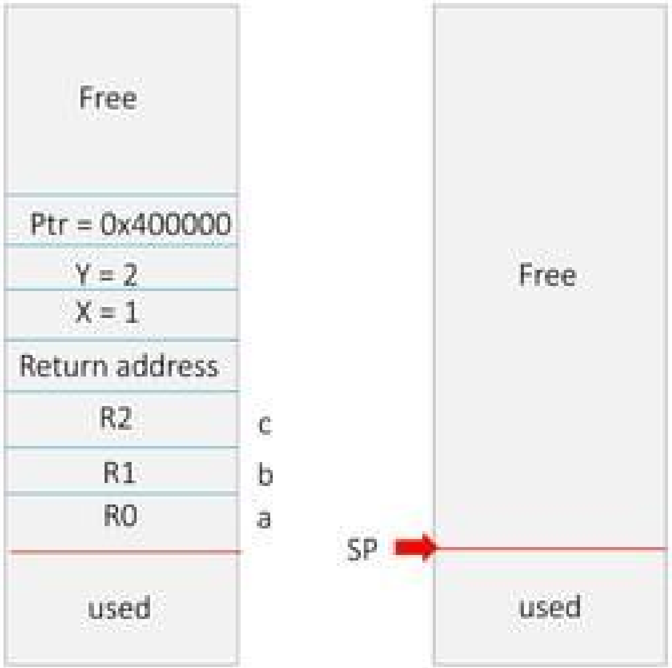
Stack

```
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

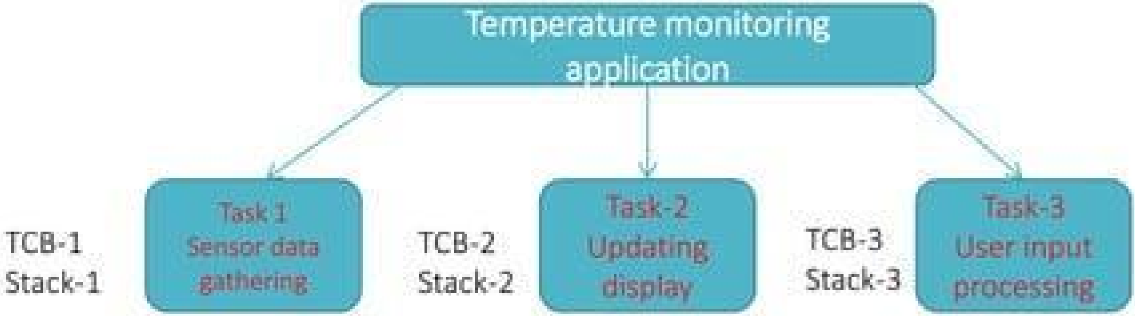
A heap is a general term used for any memory that is allocated dynamically and randomly.

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

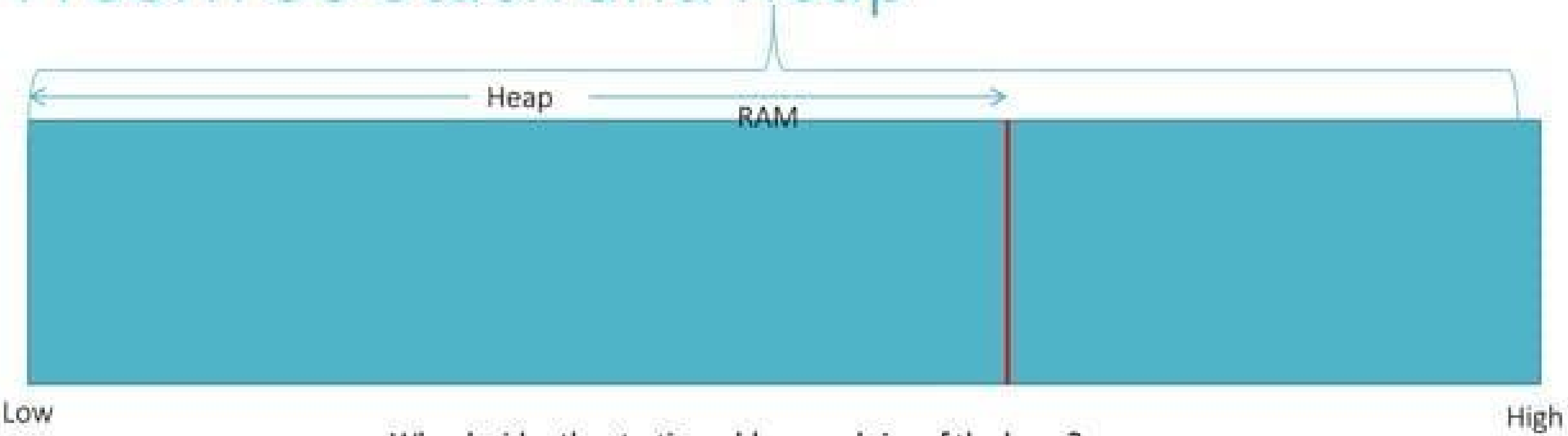
FreeRTOS Stack and heap management



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

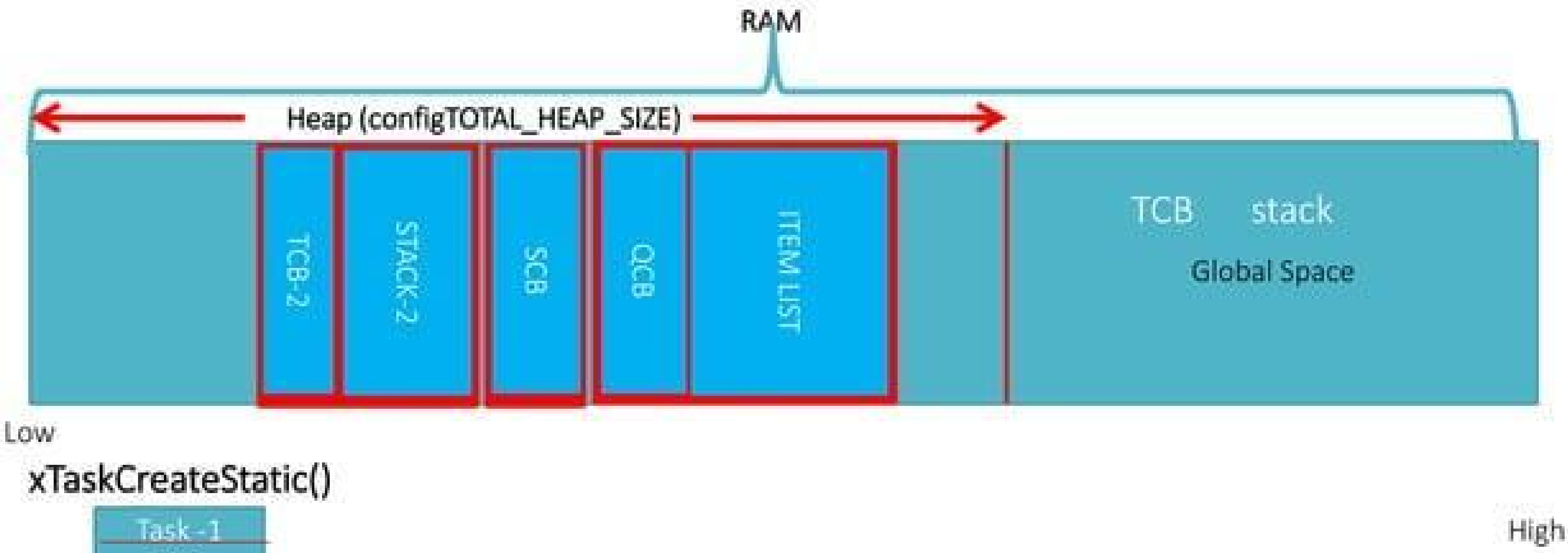
FreeRTOS Stack and heap



Who decides the starting address and size of the heap?

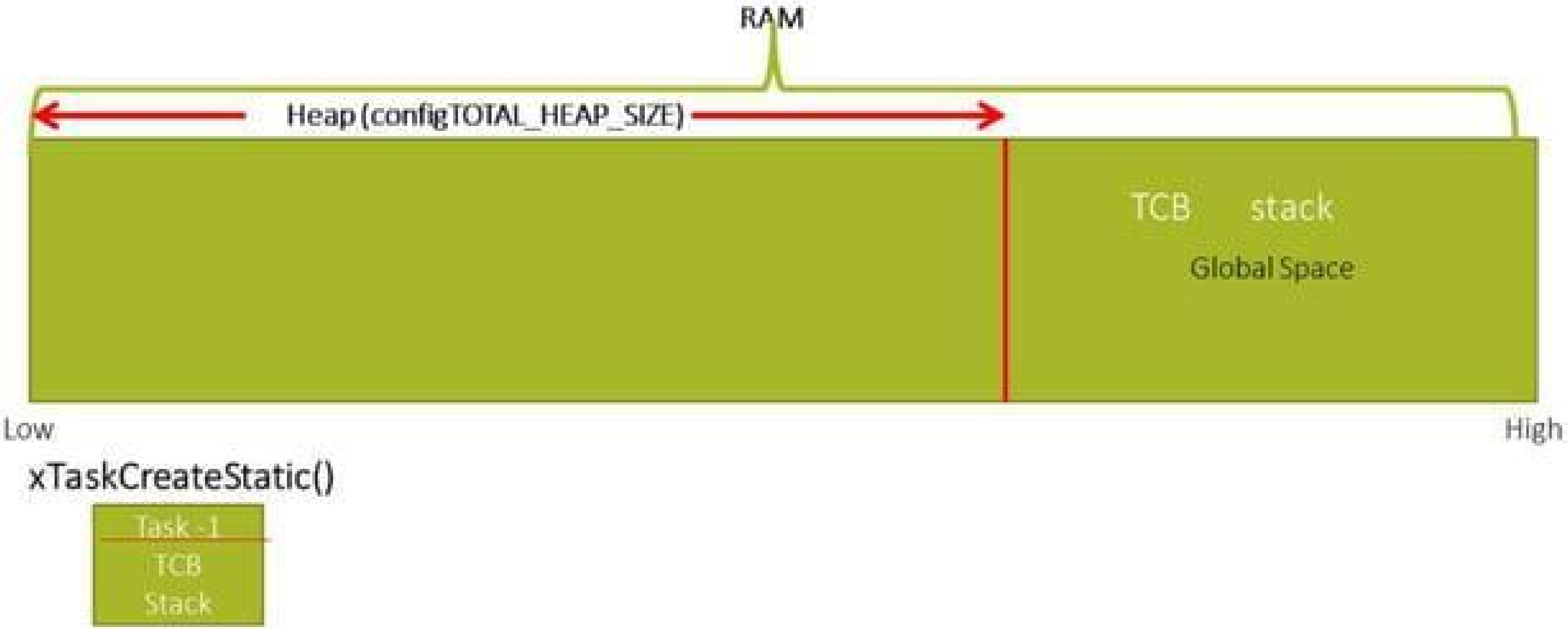
By default the FreeRTOS heap is declared by FreeRTOS kernel

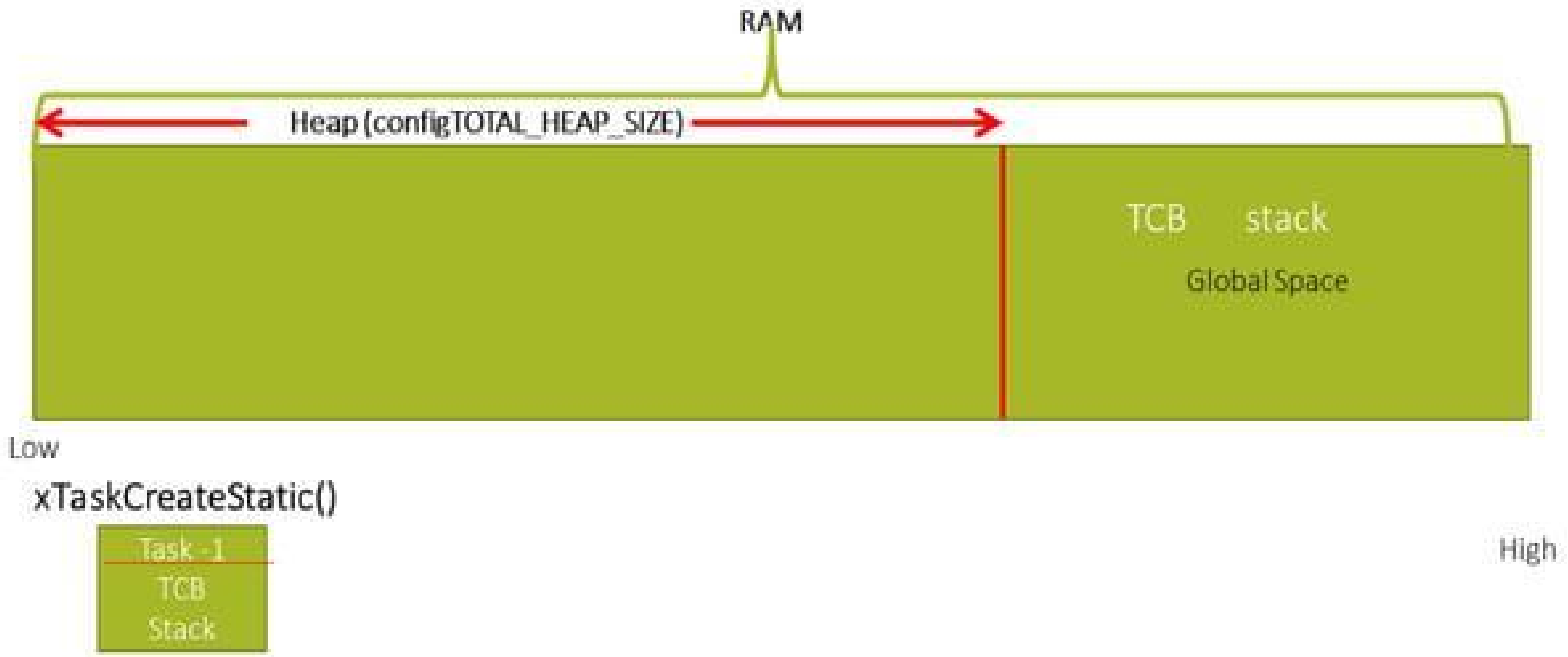
Setting configAPPLICATION_ALLOCATED_HEAP to 1 allows the heap to instead be declared by the application



xTaskCreateStatic()

FreeRTOS Stack and heap



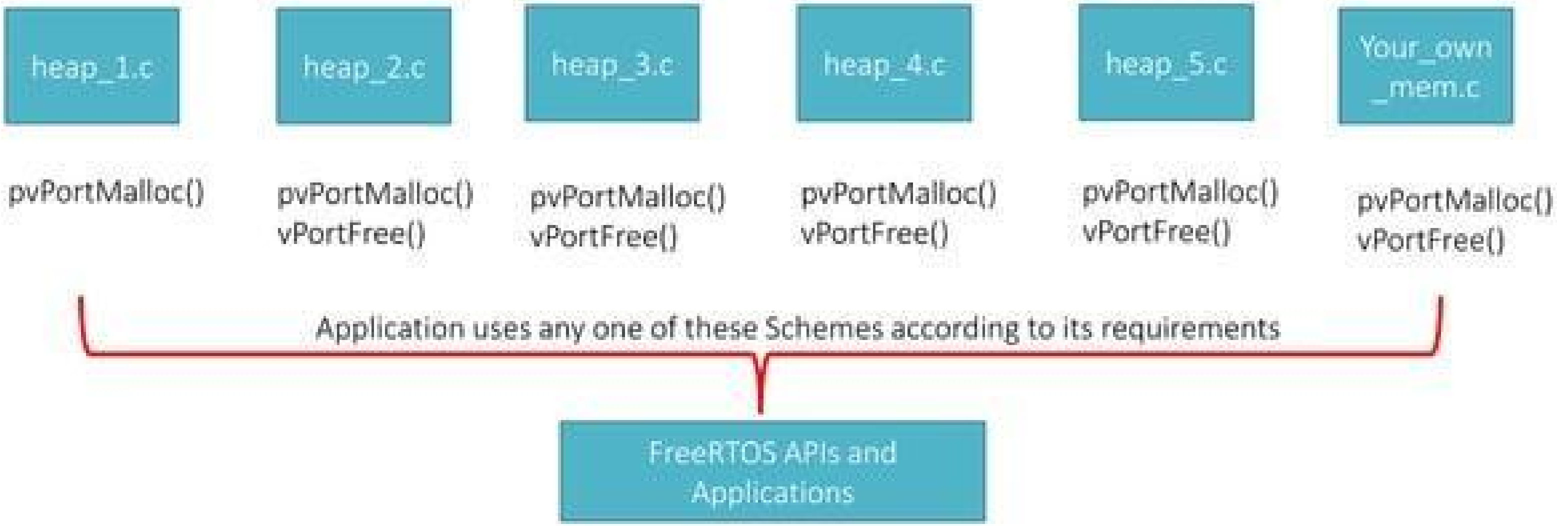

RAM
Heap (configTOTAL_HEAP_SIZE)
TCB
stack
Global Space
Low
xTaskCreateStatic()
Task -1
TCB
Stack
High

```c
/* 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. */
}
```

```c
/* 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



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)*

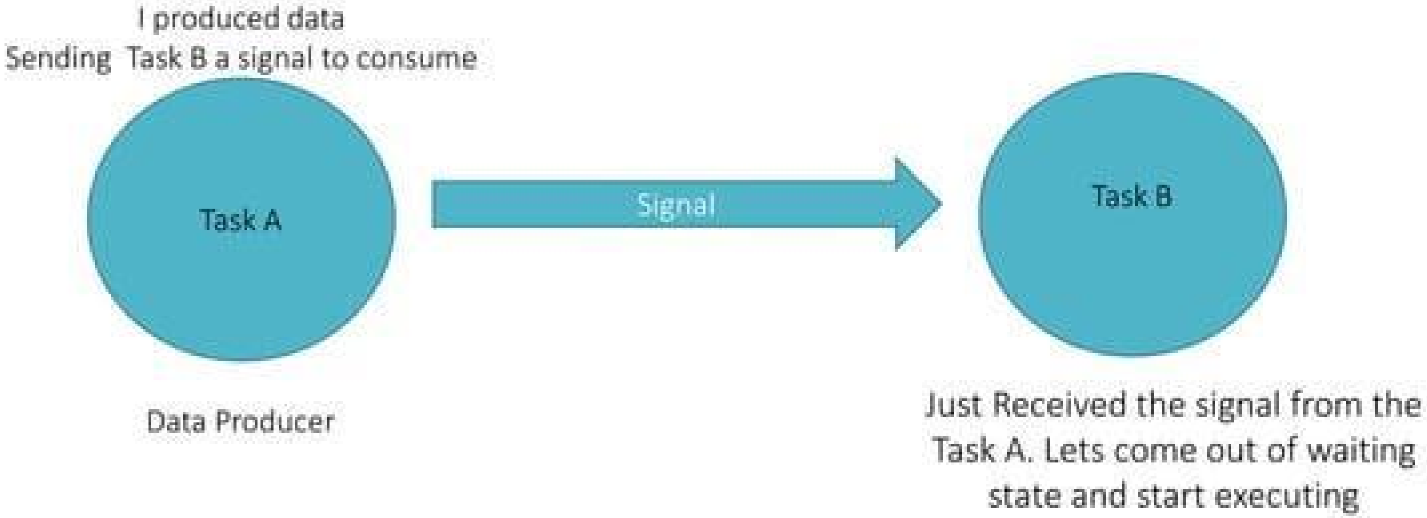
In computer science, **synchronization** refers to one of two distinct but related concepts: synchronization of processes, and synchronization of data. Process synchronization refers to the idea that multiple processes are to join up or handshake at a certain point, in order to reach an agreement or 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



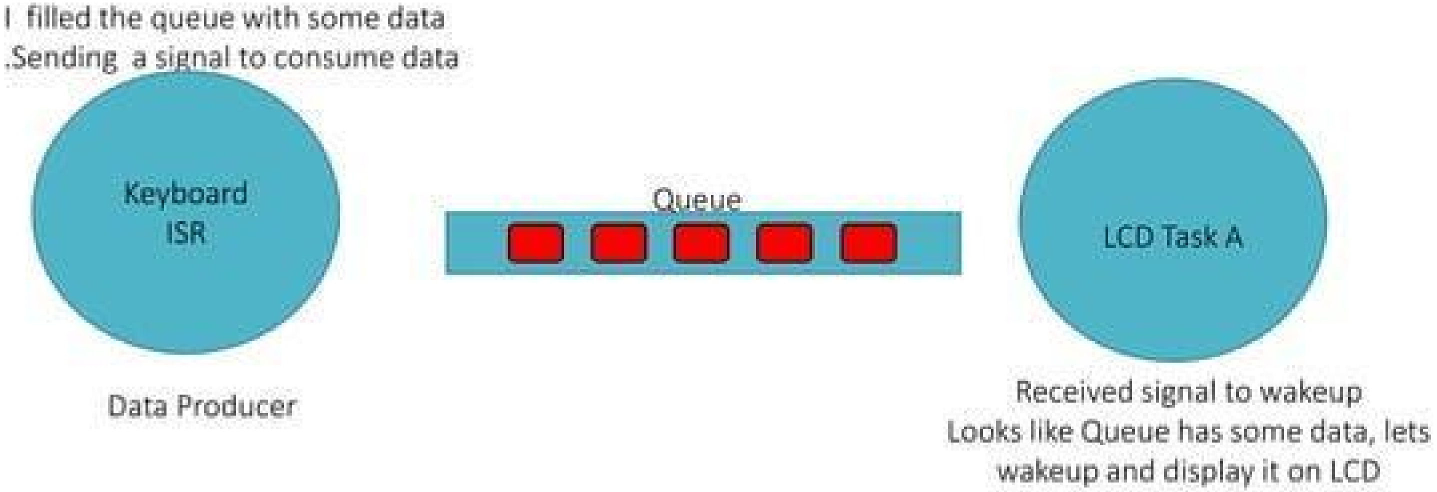
Synchronization between Task and Interrupt

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



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.

```c
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

Variables Convention

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

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

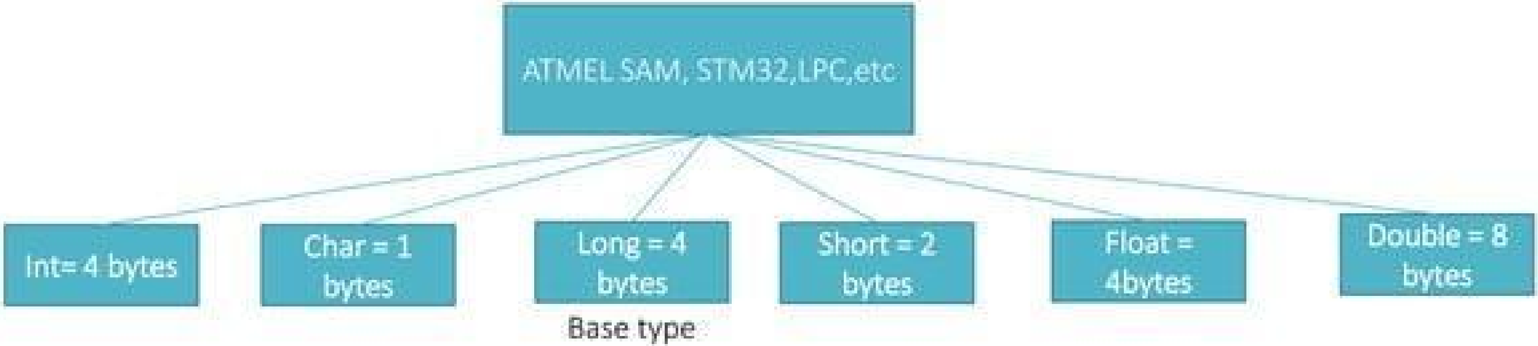
Macros

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.

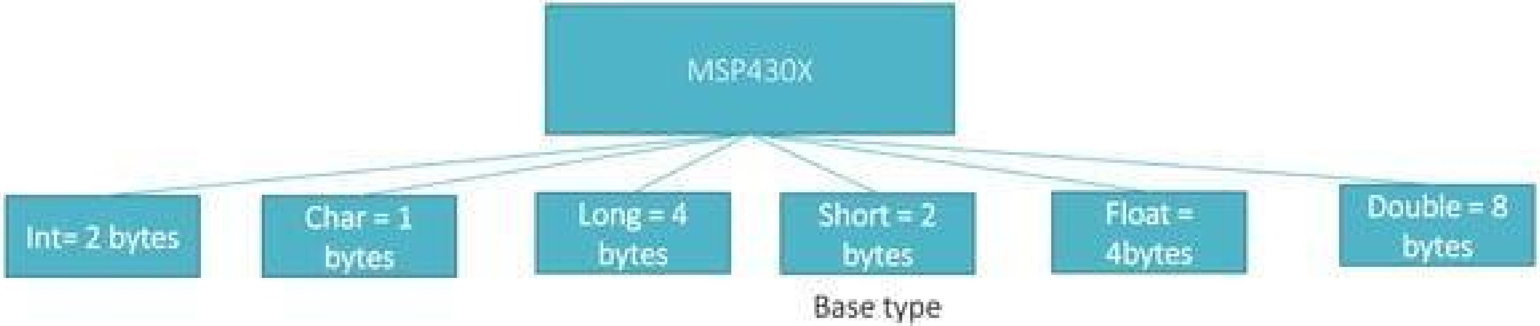
Other than the pre-fix, macros are written in all upper case, and use an underscore to separate words.

Data Types

ARM Cortex M



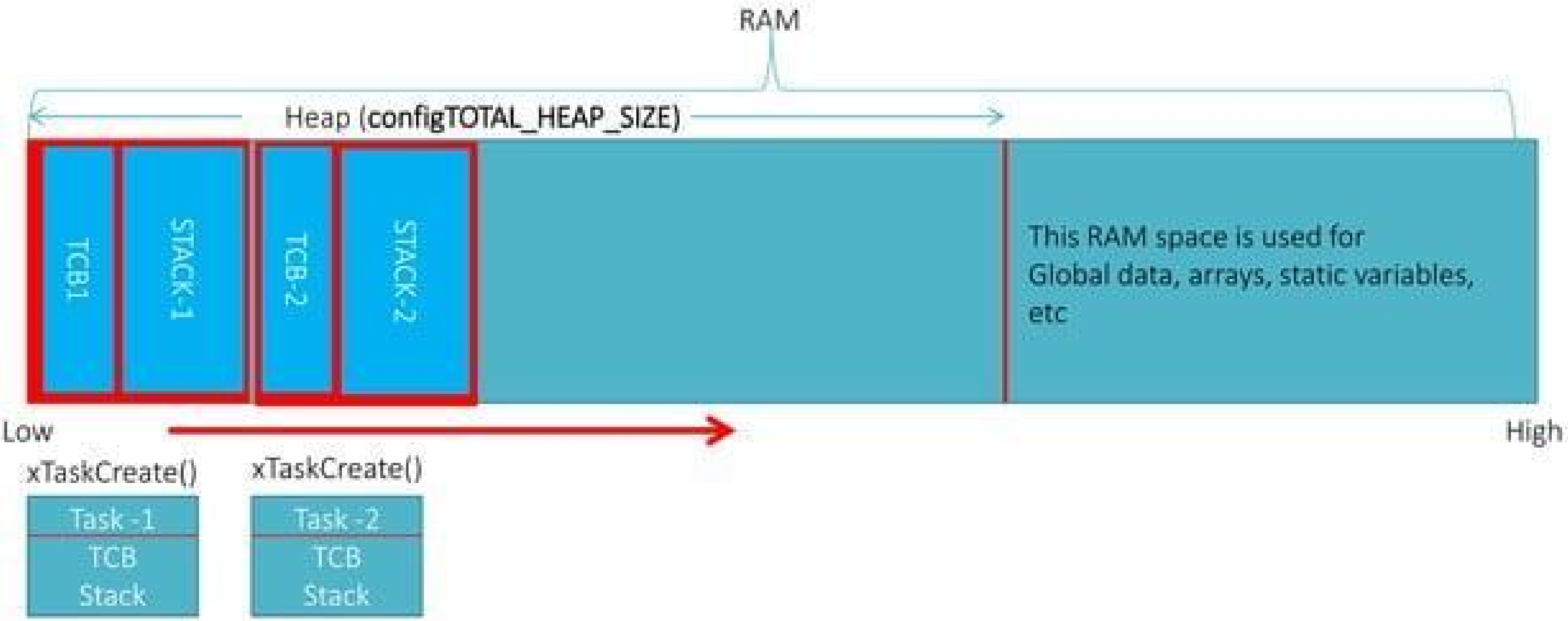
ATMEL SAM, STM32,LPC,etc
Int= 4 bytes
Char = 1 bytes
Long = 4 bytes
Short = 2 bytes
Float = 4bytes
Double = 8 bytes
Base type



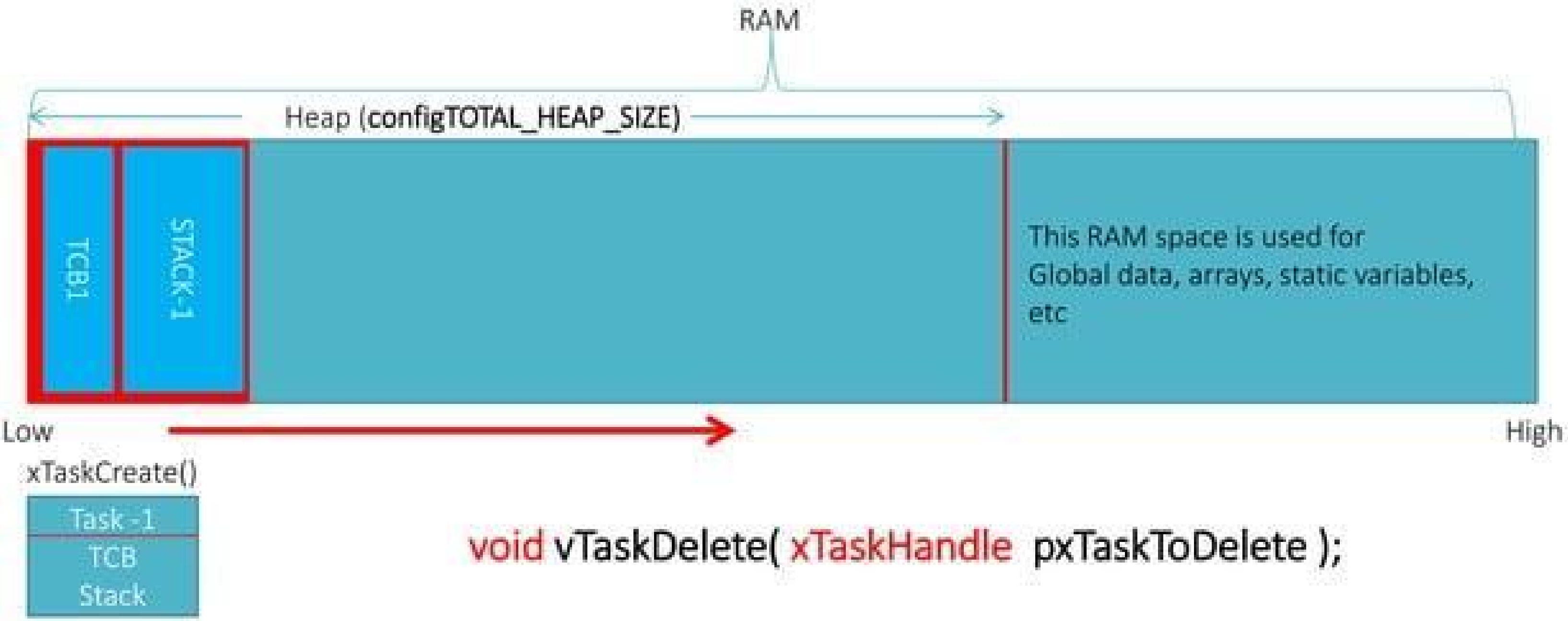
MSP430X
Int= 2 bytes
Char = 1 bytes
Long = 4 bytes
Short = 2 bytes
Float = 4bytes
Double = 8 bytes
Base type

Deleting a Task

Deleting a Task



Deleting a Task



void vTaskDelete(xTaskHandle pxTaskToDelete);

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

FreeRTOS Hardware Interrupt Configuration Items

configKERNEL_INTERRUPT_PRIORITY

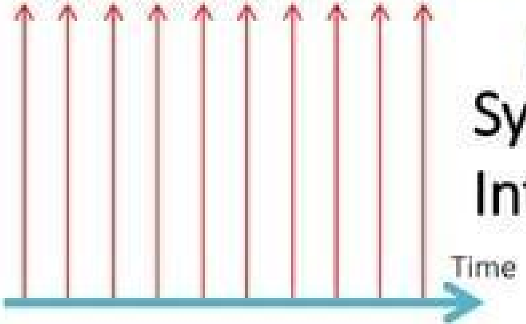
configMAX_SYSCALL_INTERRUPT_PRIORITY

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

SVC interrupt

3

configKERNEL_INTERRUPT_PRIORITY

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

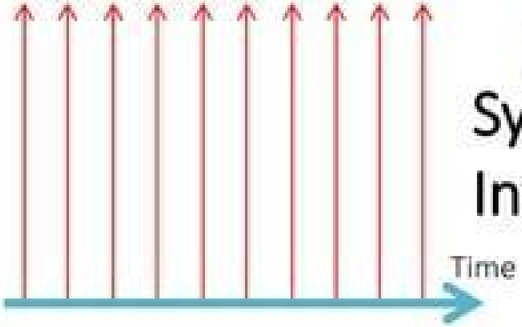
What is the value of __NVIC_PRIO_BITS macro ?

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

SVC interrupt

3

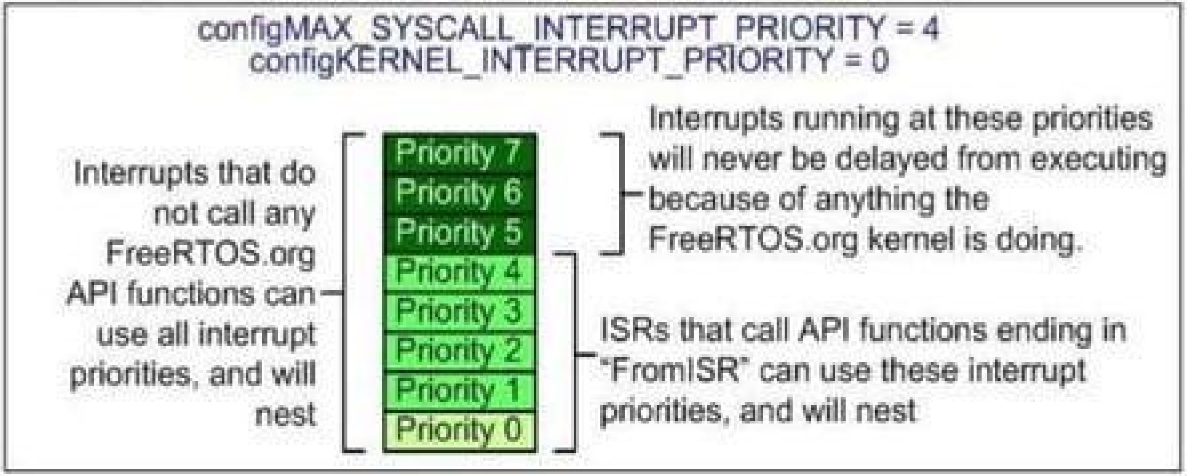
configMAX_SYSCALL_INTERRUPT_PRIORITY

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				

Write has no effect

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			

Write has no effect

16 Levels of Priority level

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

nted | Implemented | Not implemented

ffect

Write has no effect

16 Levels of Priority level

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.

Highest Priority

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

Lowest Priority

configMAX_SYSCALL_INTERRUPT_PRIORITY

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)

Highest Priority

configMAX_SYSCALL_INTERRUPT_PRIORITY

0x00, 0x10, 0x20, 0x30, 0x40, $\boxed{\text{0x50,}}$

0x60, 0x70, 0x80, 0x90, 0xA0, 0XB0,

0xC0, 0xD0, 0xE0, 0xF0

Lowest Priority

Priority values of those ISRs which do not call FreeRTOS APIs which end with "FromISR"

Priority values of those ISRs which call FreeRTOS APIs which end with "FromISR"

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 actully decides the maximum priority level , that is allowed to use for those interrupts which use freertos APIs ending with "FromIsr" 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

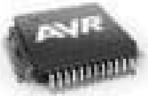
2 Priority

interrupt-1

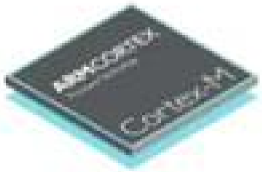
Priority 5

interrupt-2

lower priority means higher numeric value



Non-ARM Cortex M Architectures

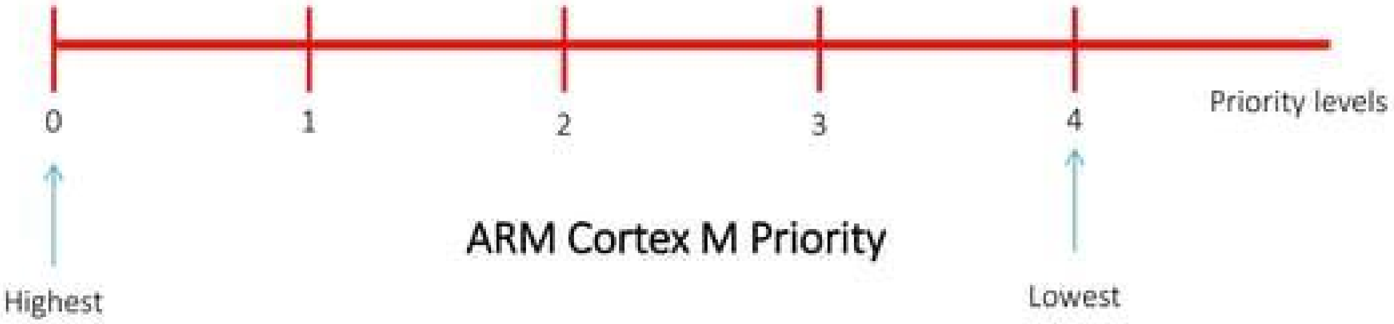


interrupt-1

interrupt-2

Interrupt-1 is higher priority than the interrupt-2


Priority levels
0
Lowest
FreeRTOS Priority
4
Highest


Priority levels
0
1
2
3
4
ARM Cortex M Priority
Highest
Lowest

FreeRTOS Task Priority APIs

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

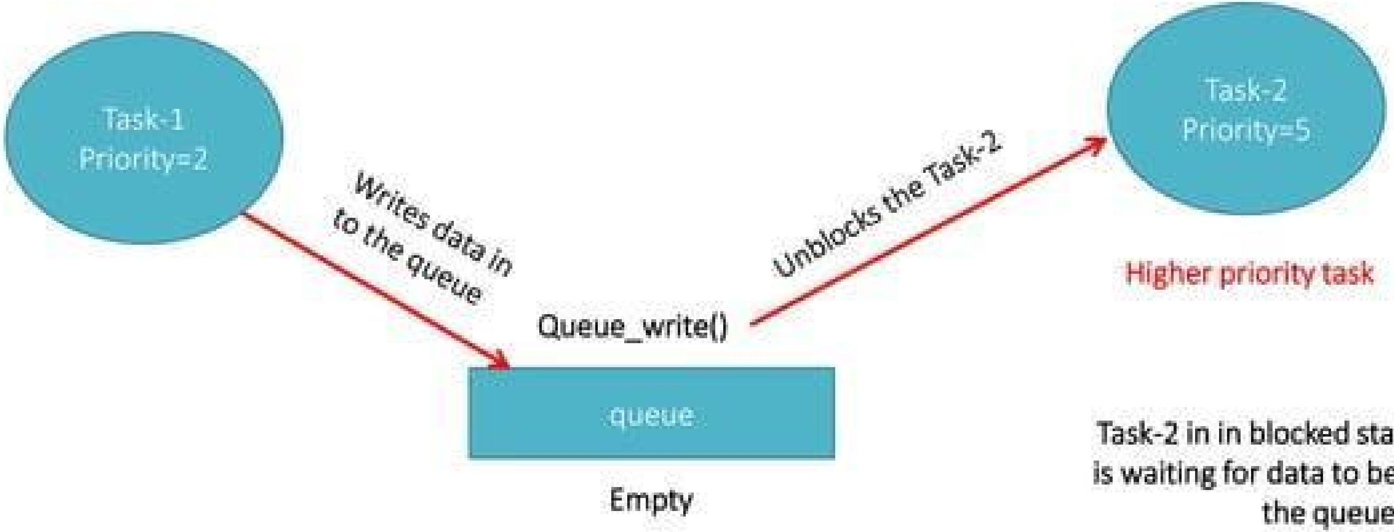
If you want to send a data item in to the queue from the ISR, then use xQueueSendFromISR() instead of xQueueSend().
xQueueSendFromISR() is an interrupt safe version of xQueueSend().

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?



Task-2 in in blocked state because it is waiting for data to be arrived in to the queue

```
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()
   }
```

```
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 *qptr , 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 Contex (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)

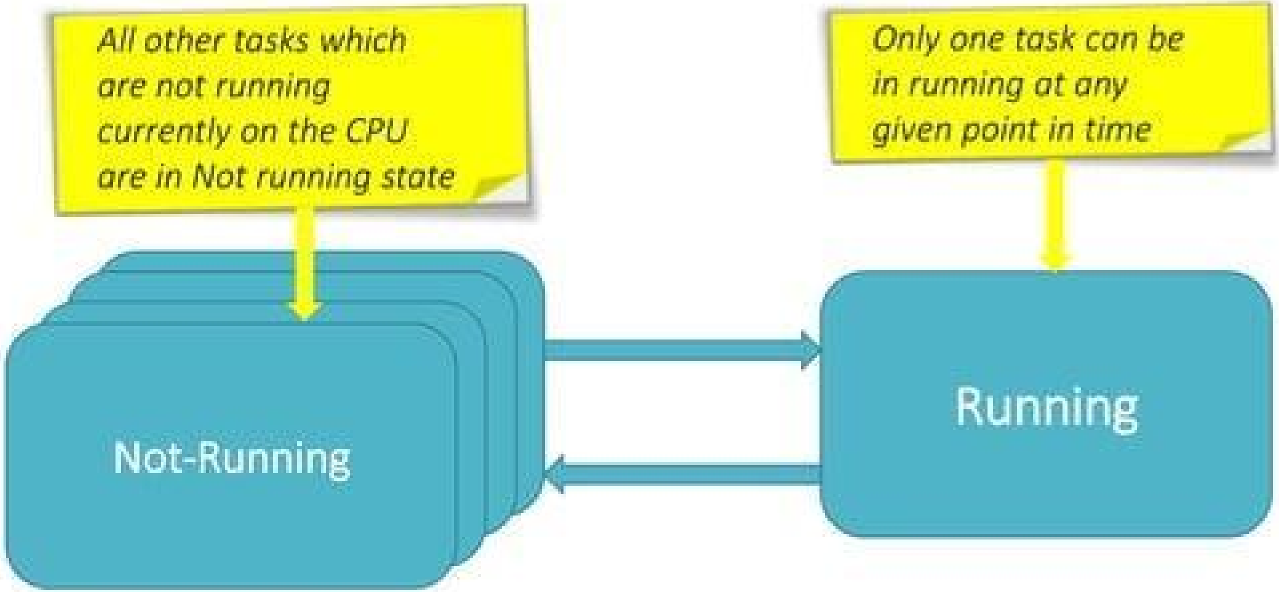
In ARM cortex M based Processors usage exception will be raised by the processor if you return to the task context by preempting ISR .

FreeRTOS Task States

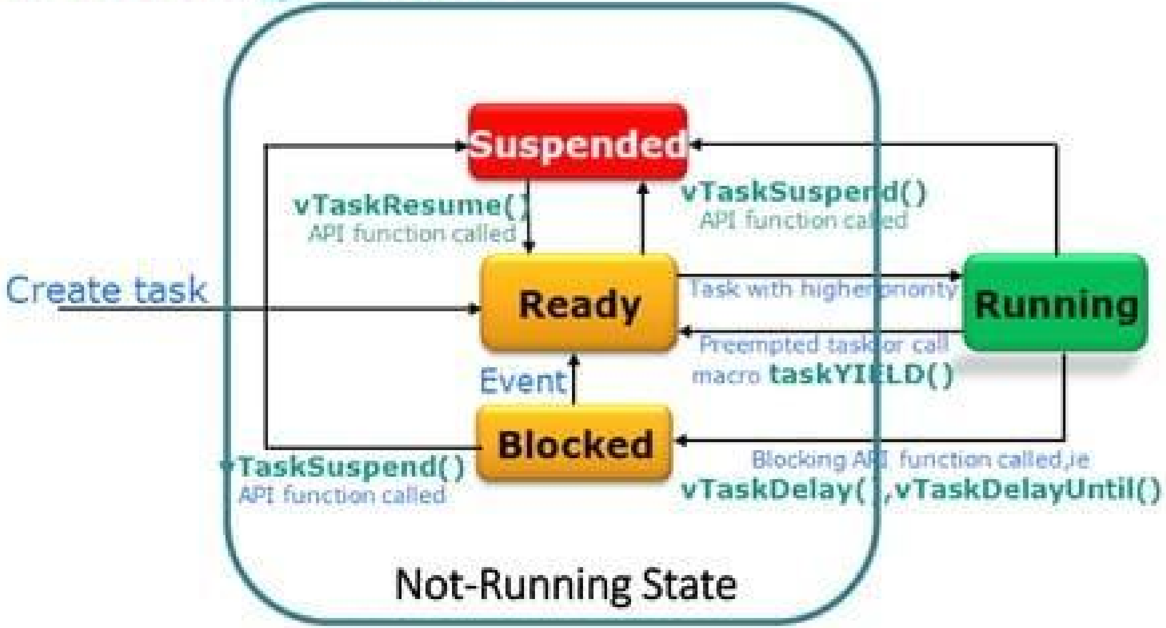
Top Level Task States

Running and Not-Running State of a Task

Top Level Task States- Simplistic Model

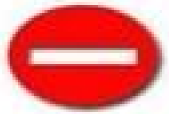


Not-Running State



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

CPU
Engaged

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

```
vTaskDelay(10)
```

Not runs for 10ms

CPU
Not-Engaged

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.

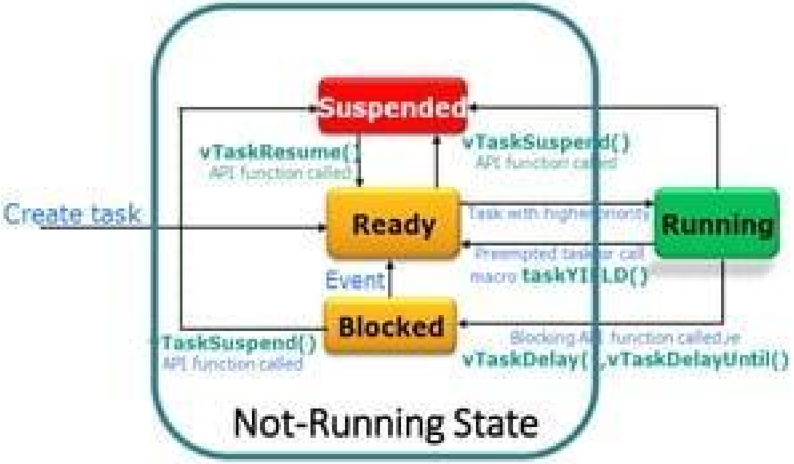
2. For Synchronization –For example, a task may enter the Blocked state to wait for data to arrive on a queue. When the another task or interrupt fills up the queue , the blocked task will be unblocked.

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()

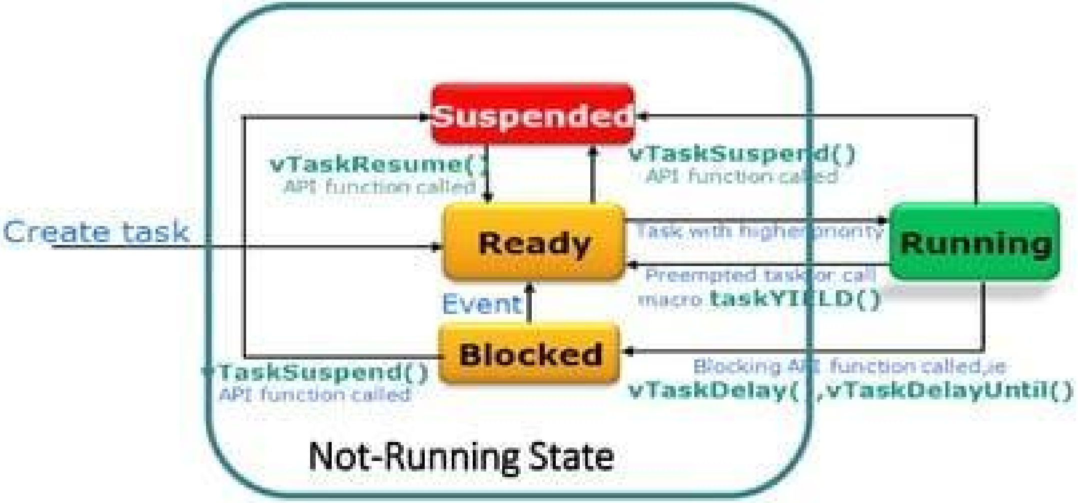


Queues | Semaphores | Mutex

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

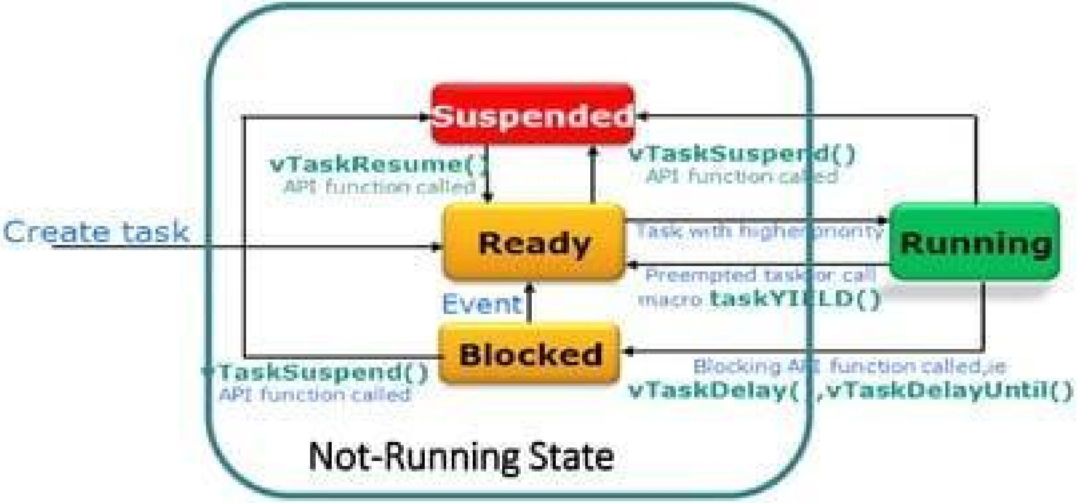
The Suspended state



```
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.
}
```

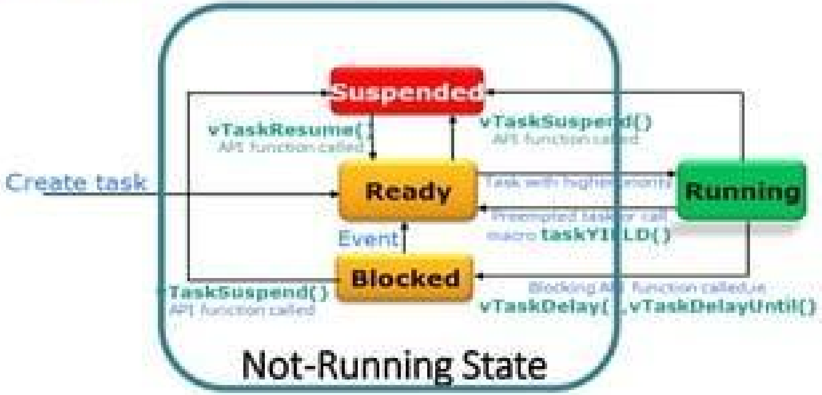
The Ready state

The Ready state



Conclusion

Task States : Conclusion



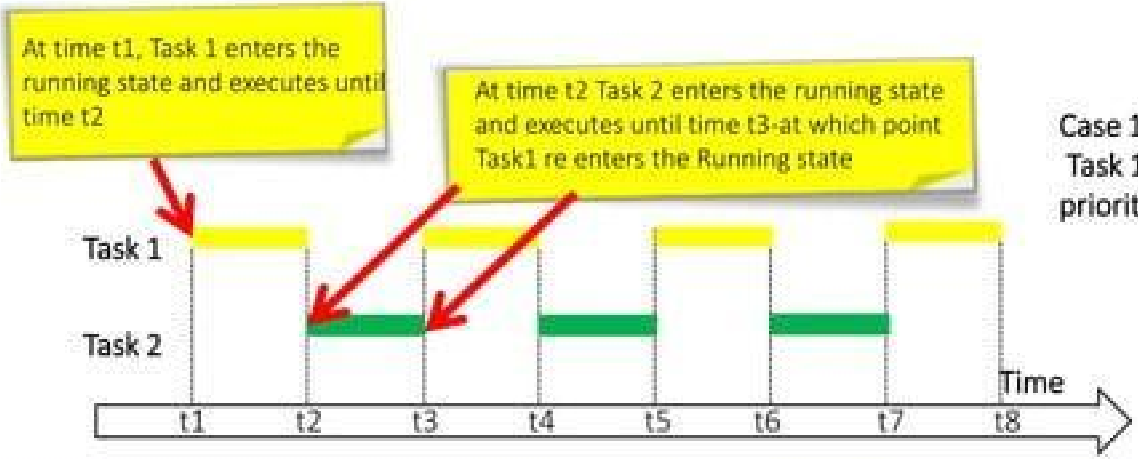
FreeRTOS : Importance of delay

Crude delay Implementation

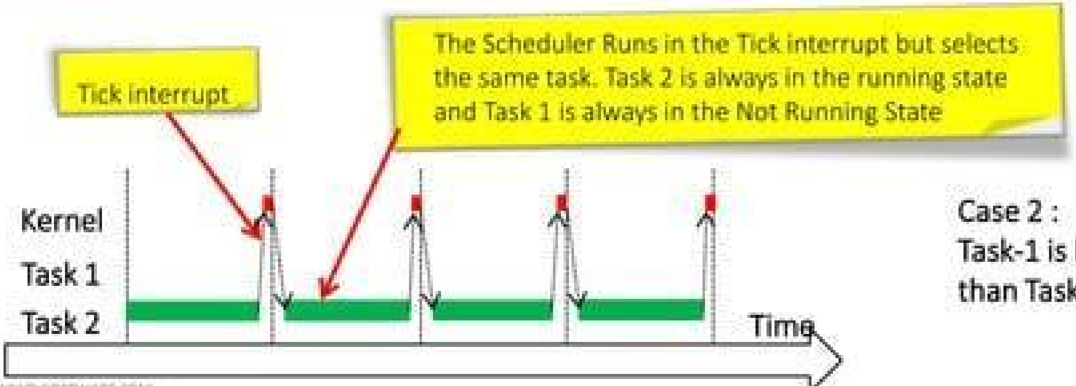
Crude delay Implementation

```
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



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

```c
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. */
        }
    }
}
```

```c
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 );
```

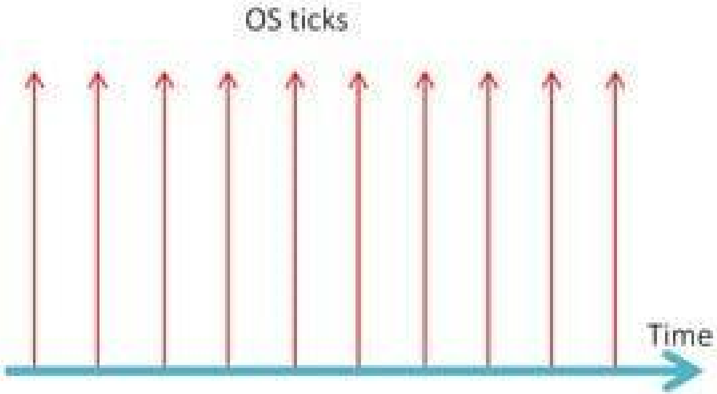
vTaskDelay()

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 );
```

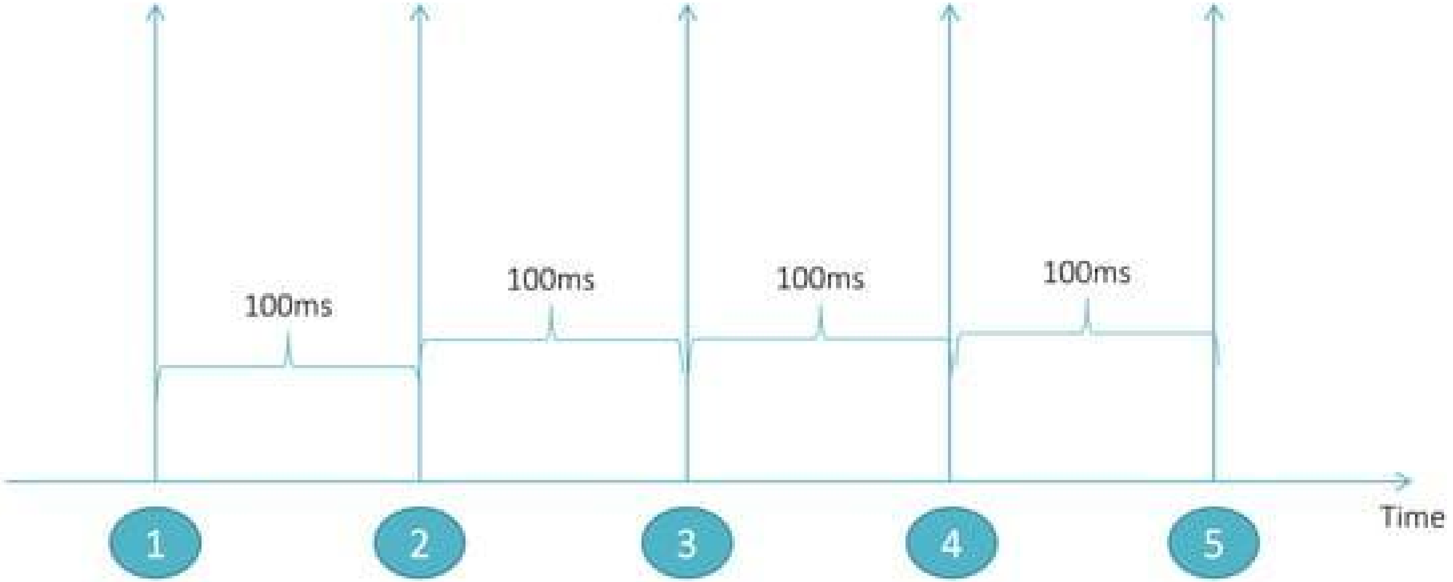
vTaskDelay()



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 );
    }
}
```

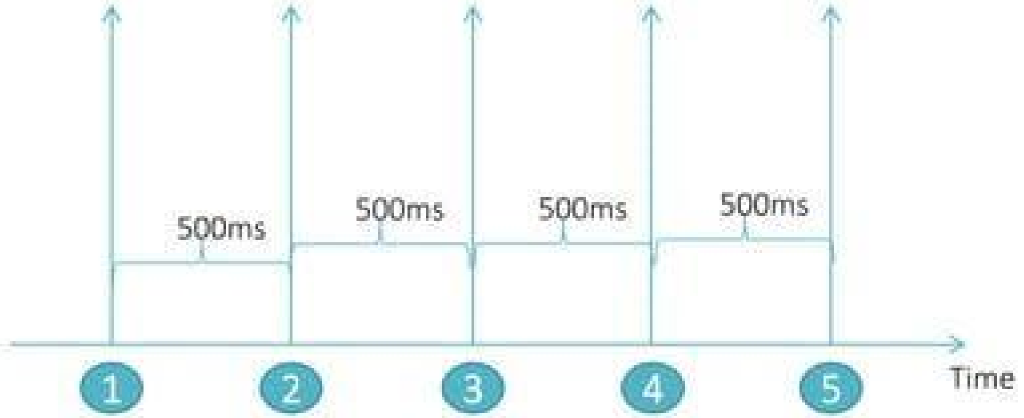
void vTaskDelay(portTickType xTicksToDelay);



Task 1 executing periodically for every 100ms

Example usage:

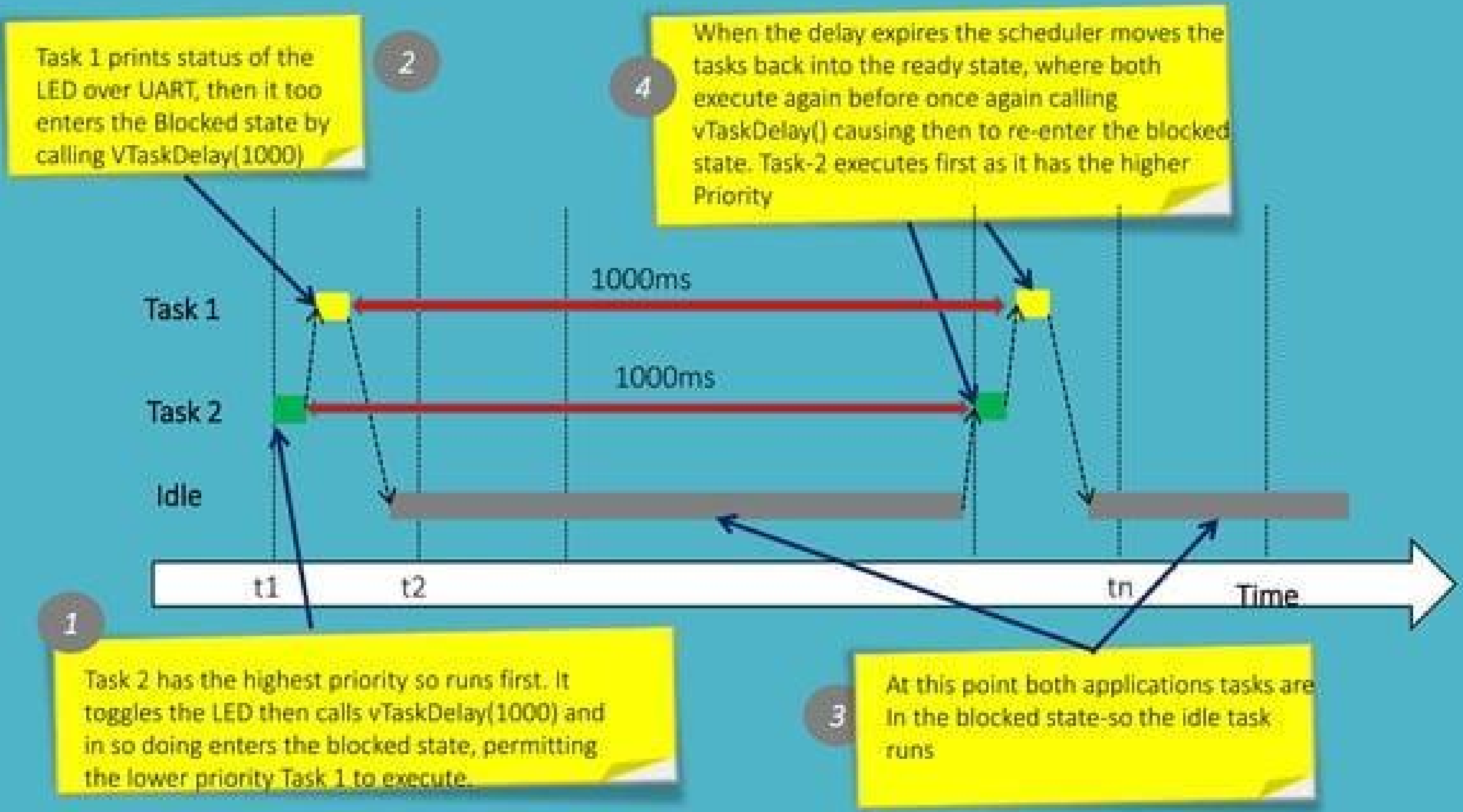
```
void vTaskFunction( void * pvParameters )
{
/* Block for 500ms. */
const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

    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 );
     }
}
```


Task 1 prints status of the LED over UART, then it too enters the Blocked state by calling VTaskDelay(1000)
2
4
When the delay expires the scheduler moves the tasks back into the ready state, where both execute again before once again calling vTaskDelay() causing then to re-enter the blocked state. Task-2 executes first as it has the higher Priority
1000ms
Task 1
1000ms
Task 2
Idle
t1
t2
tn
Time
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.
3
At this point both applications tasks are in the blocked state-so the idle task runs

FreeRTOS Hook Functions

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

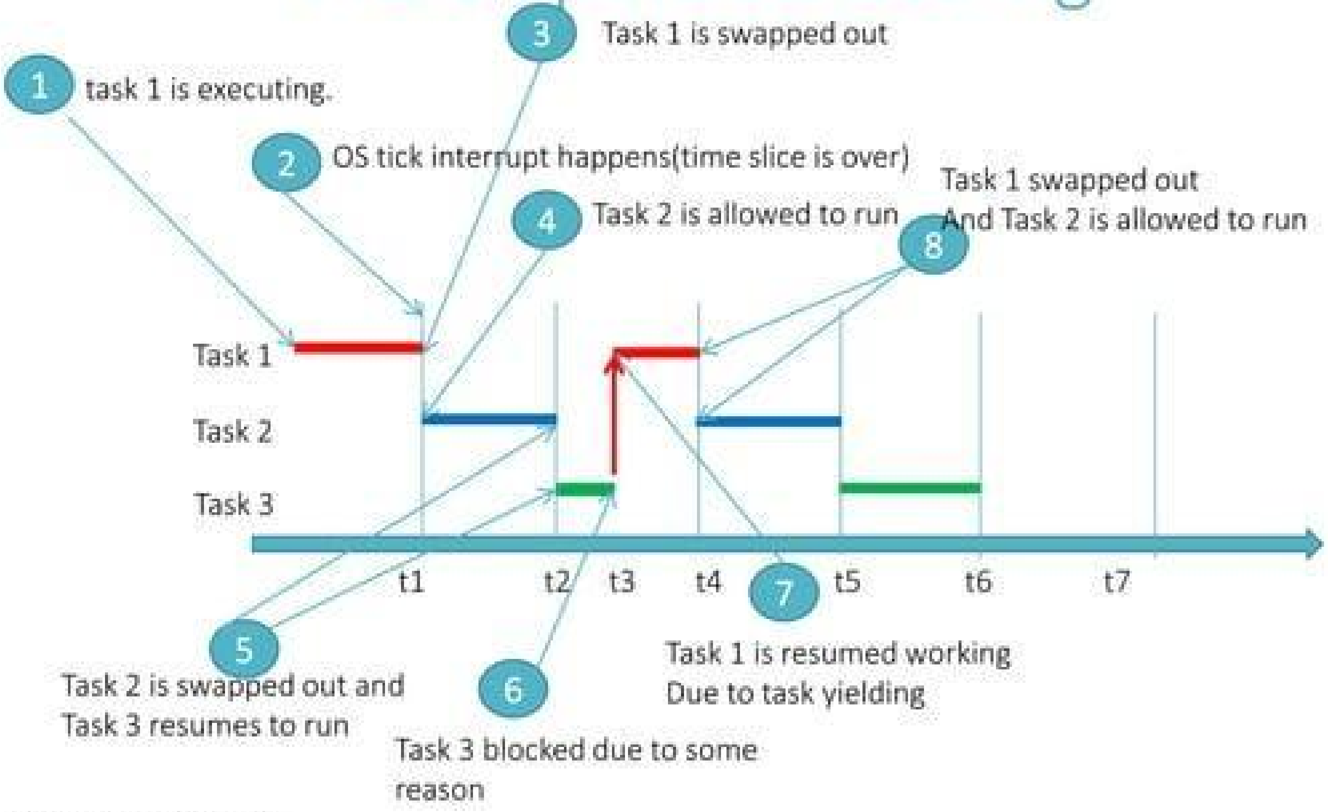
1. Preemptive scheduling
2. Priority based preemptive scheduling
3. co-operative scheduling

Preemption

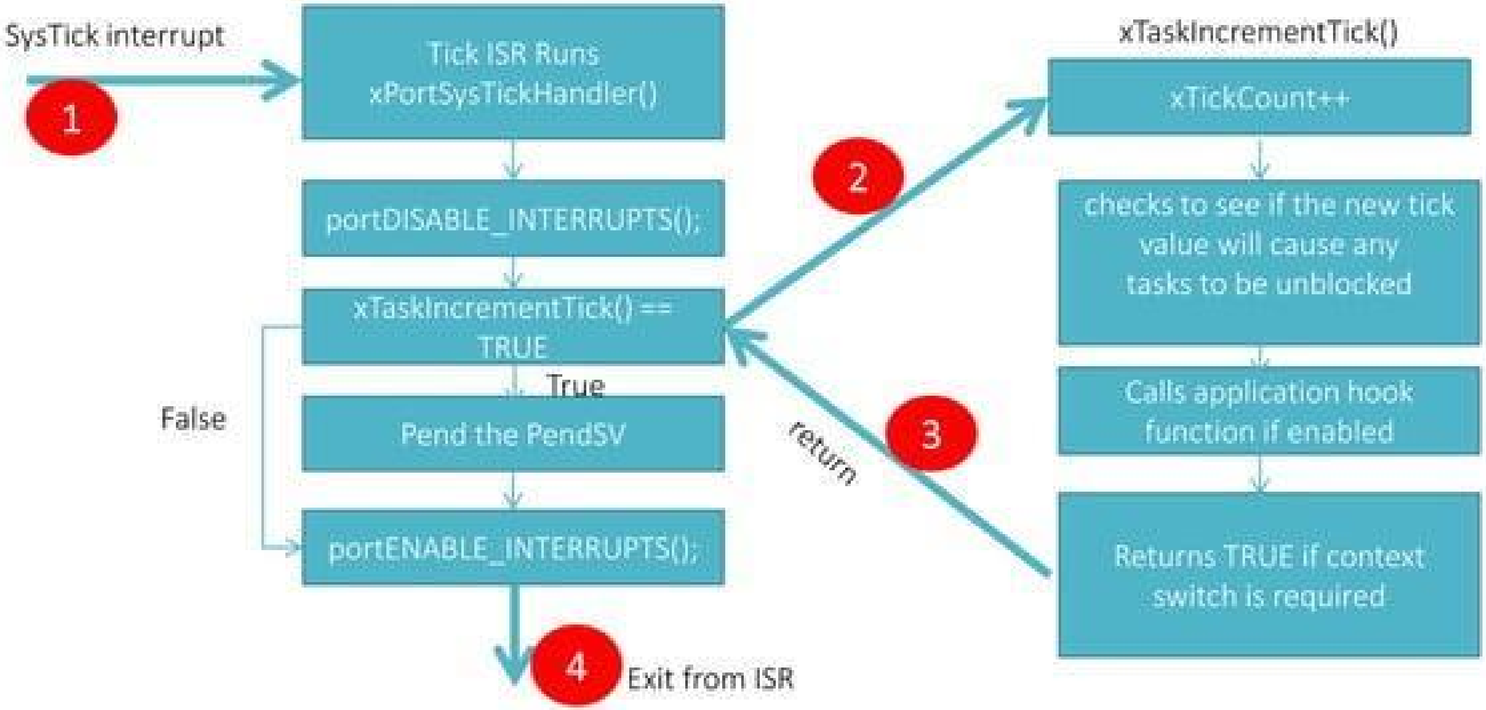
Preemption is the act of temporarily interrupting an already executing task with the intention of removing it from the running state without its co-operation .

Pre-emptive Scheduling

Pre-emptive Scheduling

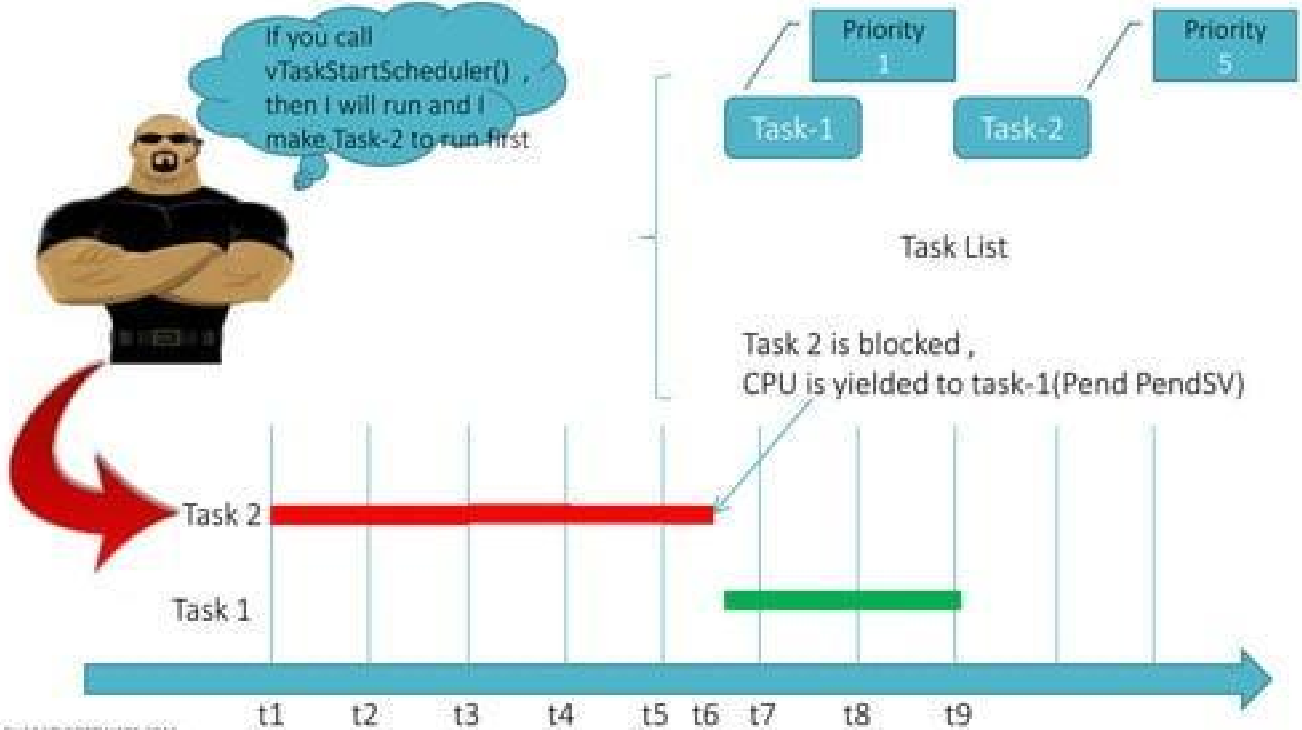


What RTOS tick ISR does ? : Conclusion

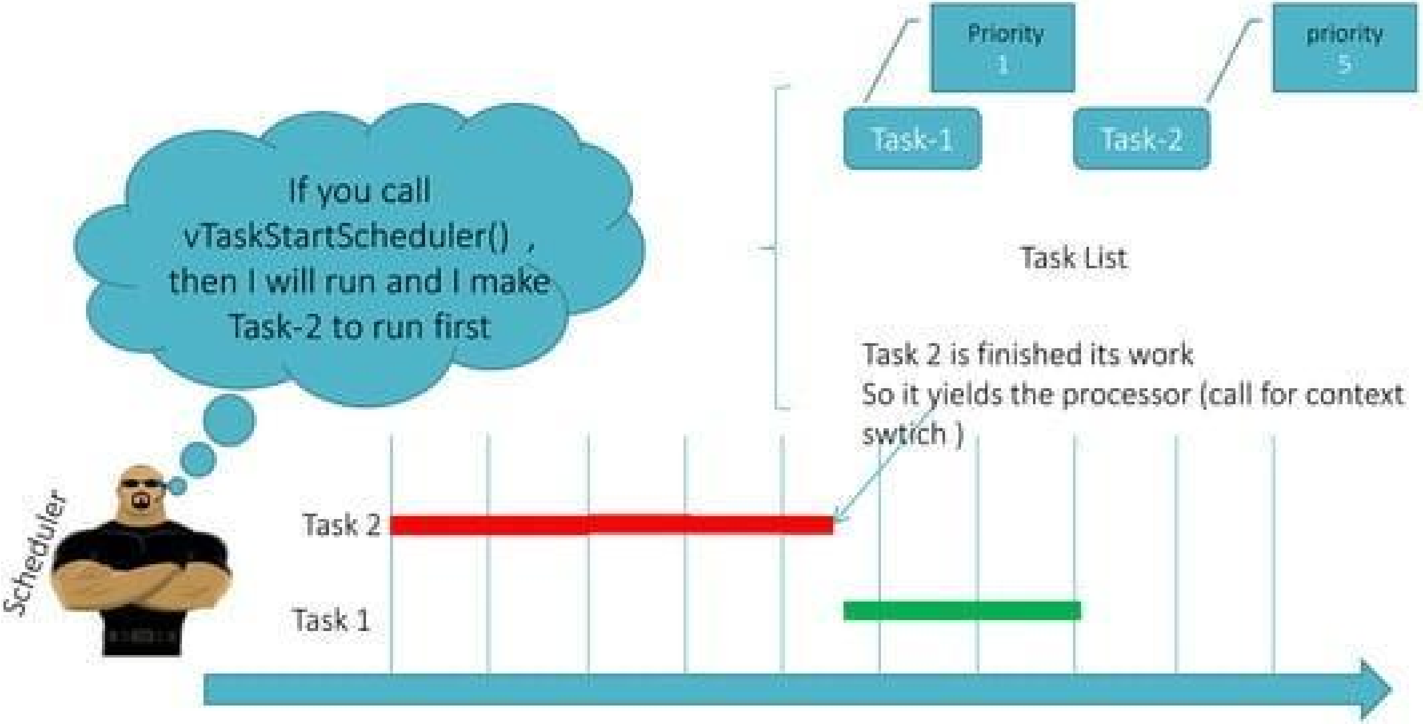


Priority based Pre-Emptive Scheduling

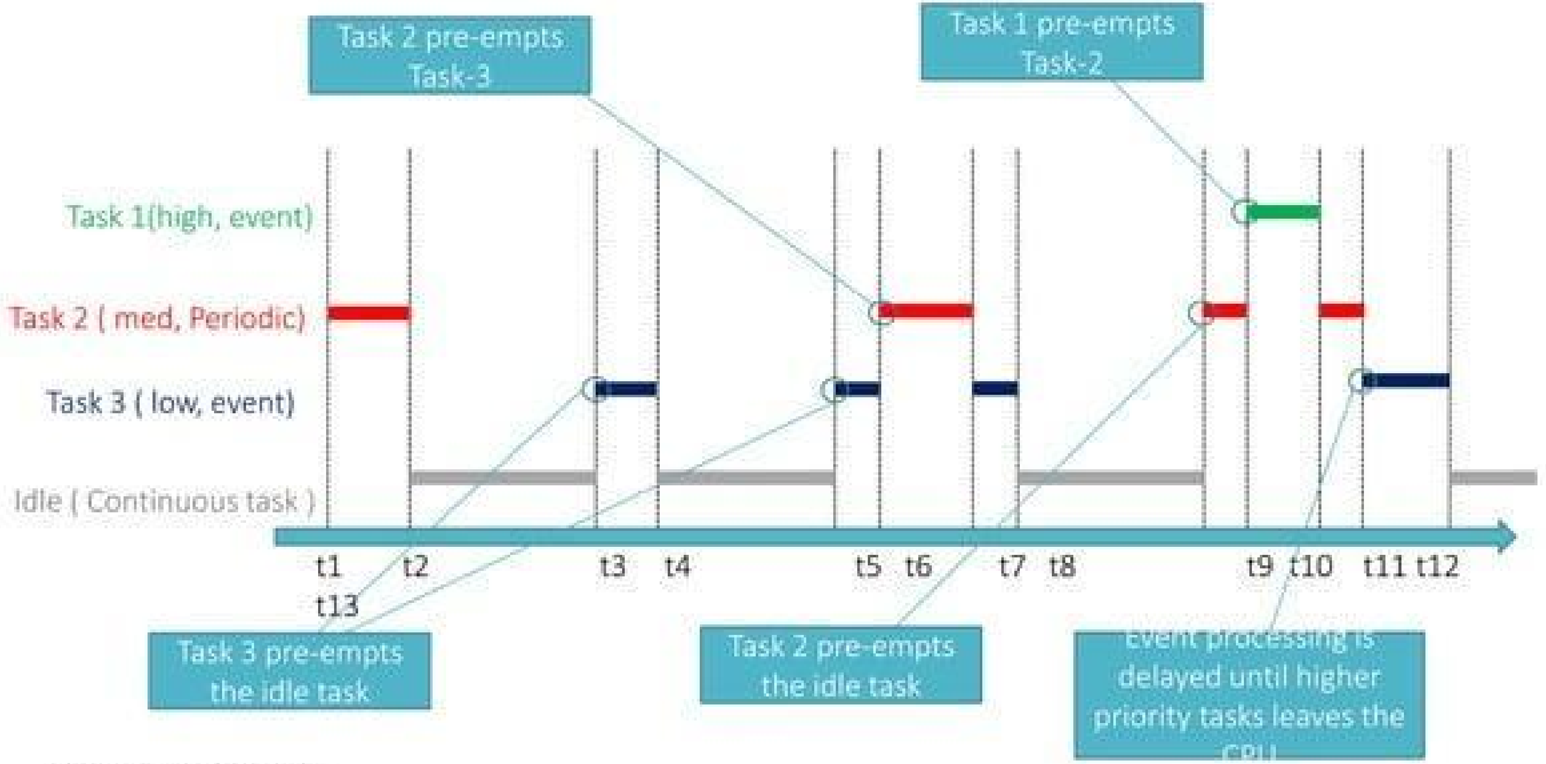
Prioritized Pre-emptive Scheduling



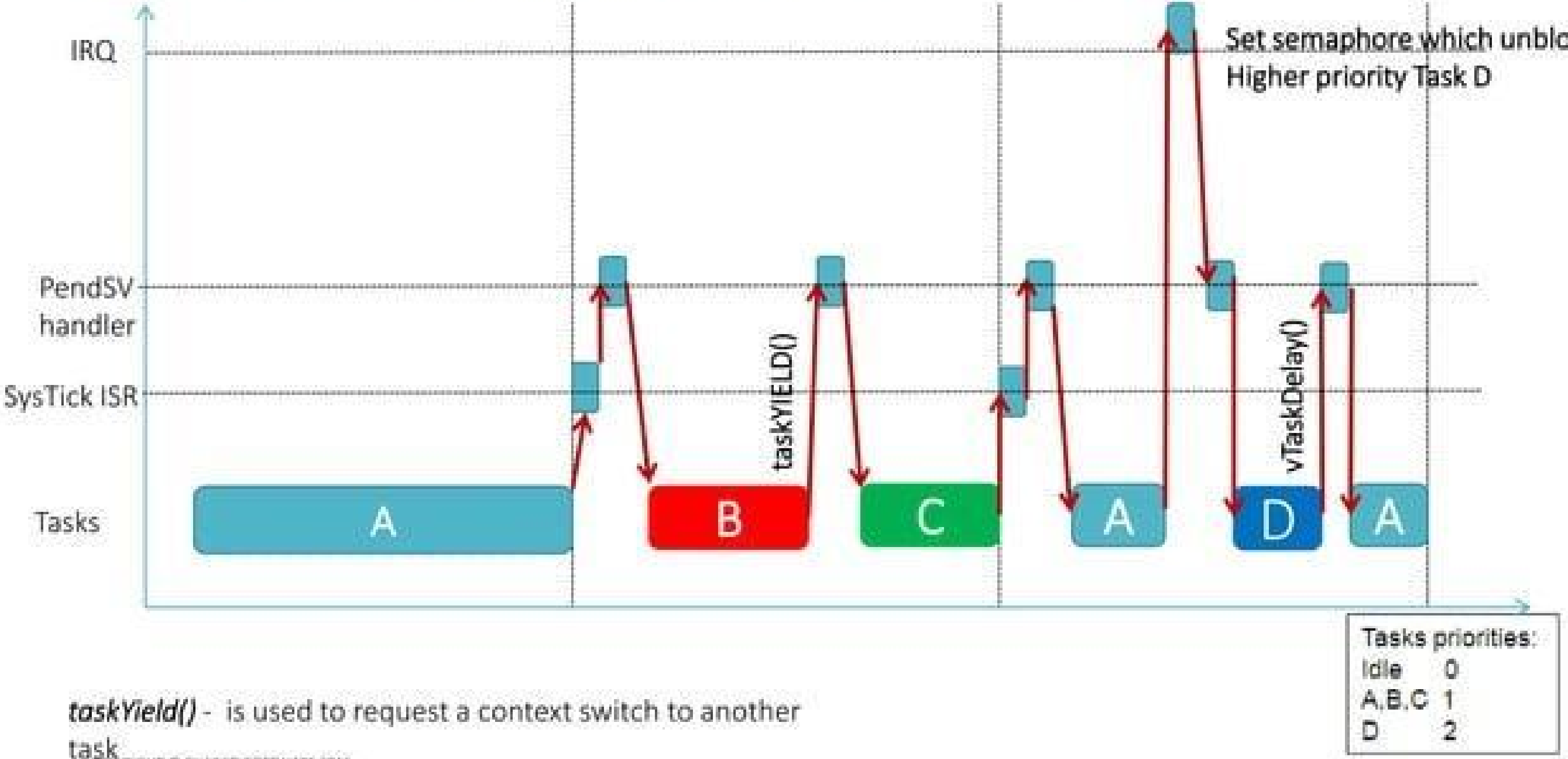
Prioritized Pre-emptive Scheduling



Prioritized Pre-emptive Scheduling



Prioritized Pre-emptive Scheduling



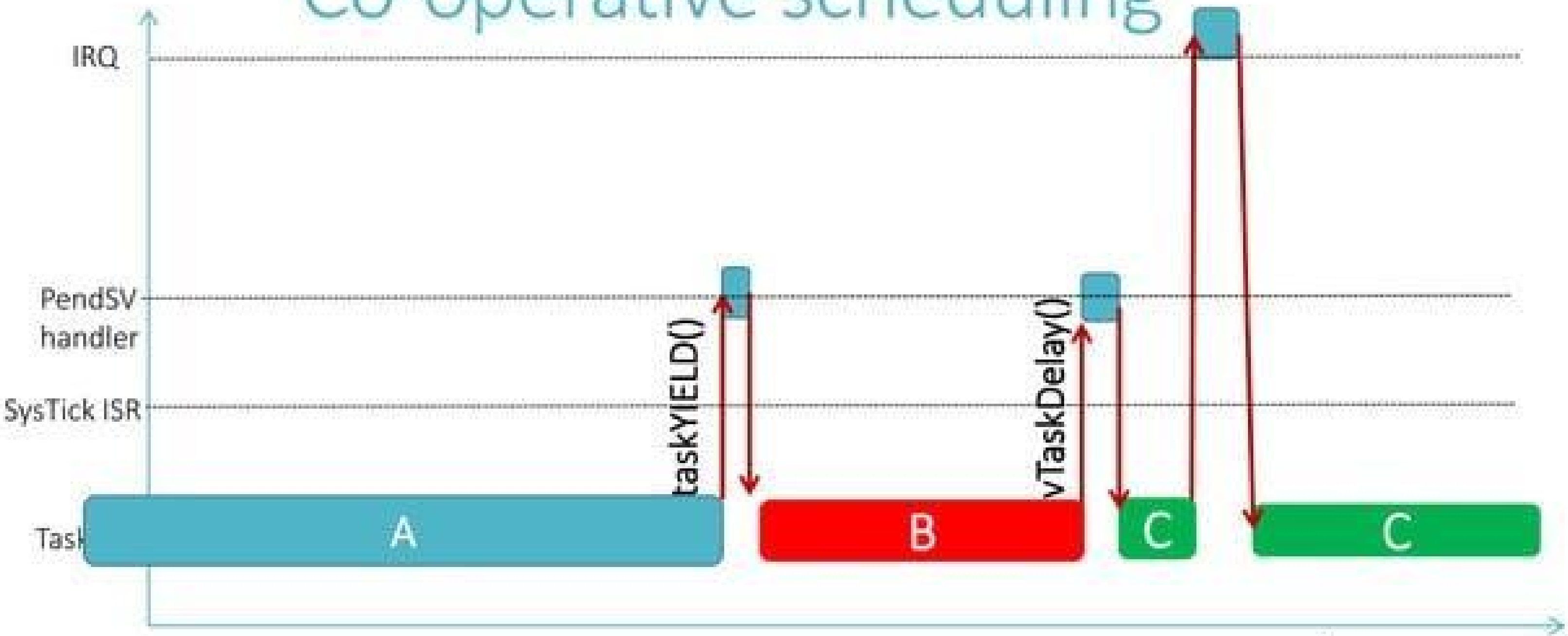
taskYield() - is used to request a context switch to another task

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

Scheduler

Prioritized Pre-emptive
Scheduling
Suitable for RTA

Co-operative
Scheduling
Good for NRTA

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