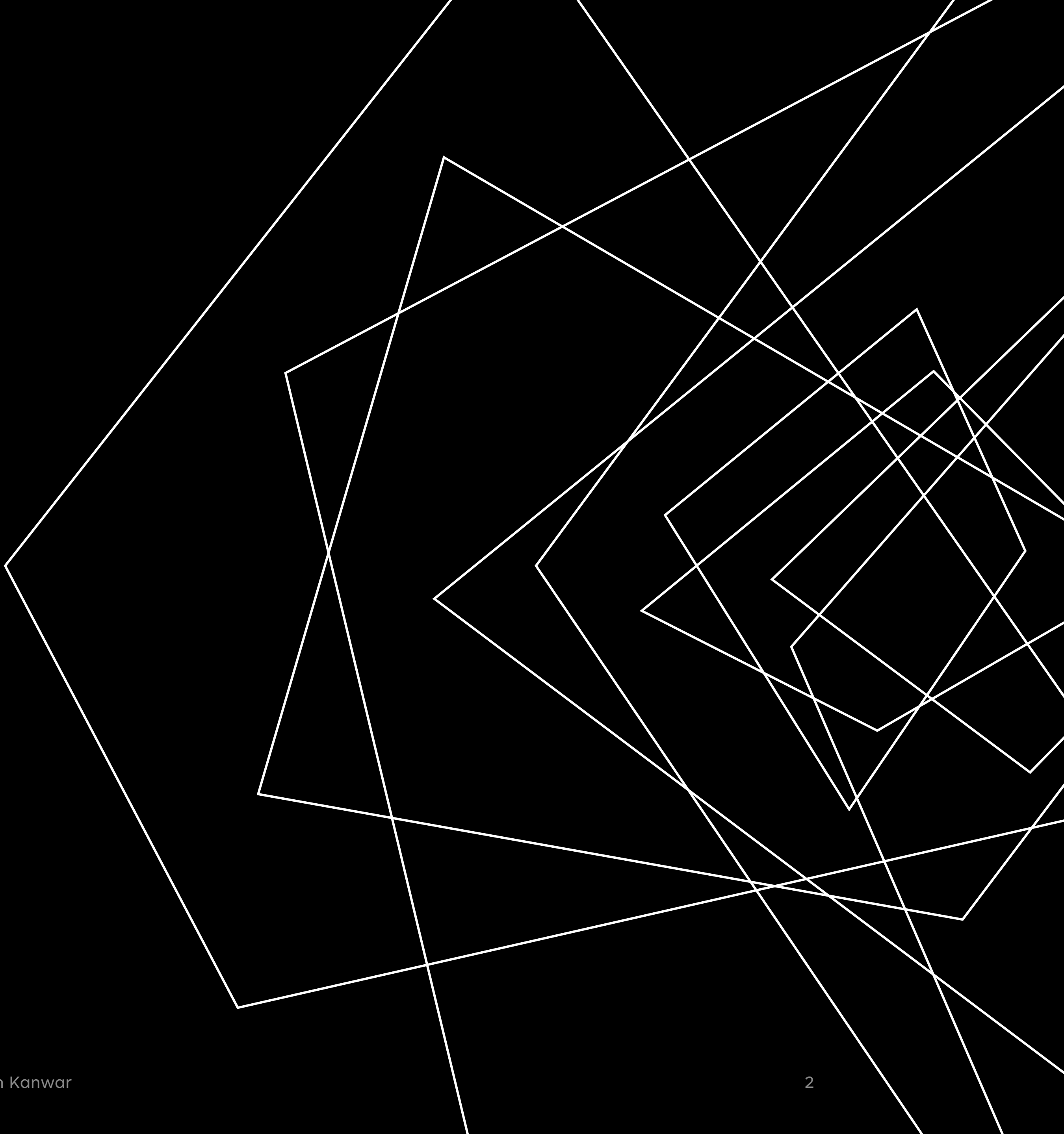
STAY AHEAD OF GAME USING RTOS

Aman Kanwar

AGENDA

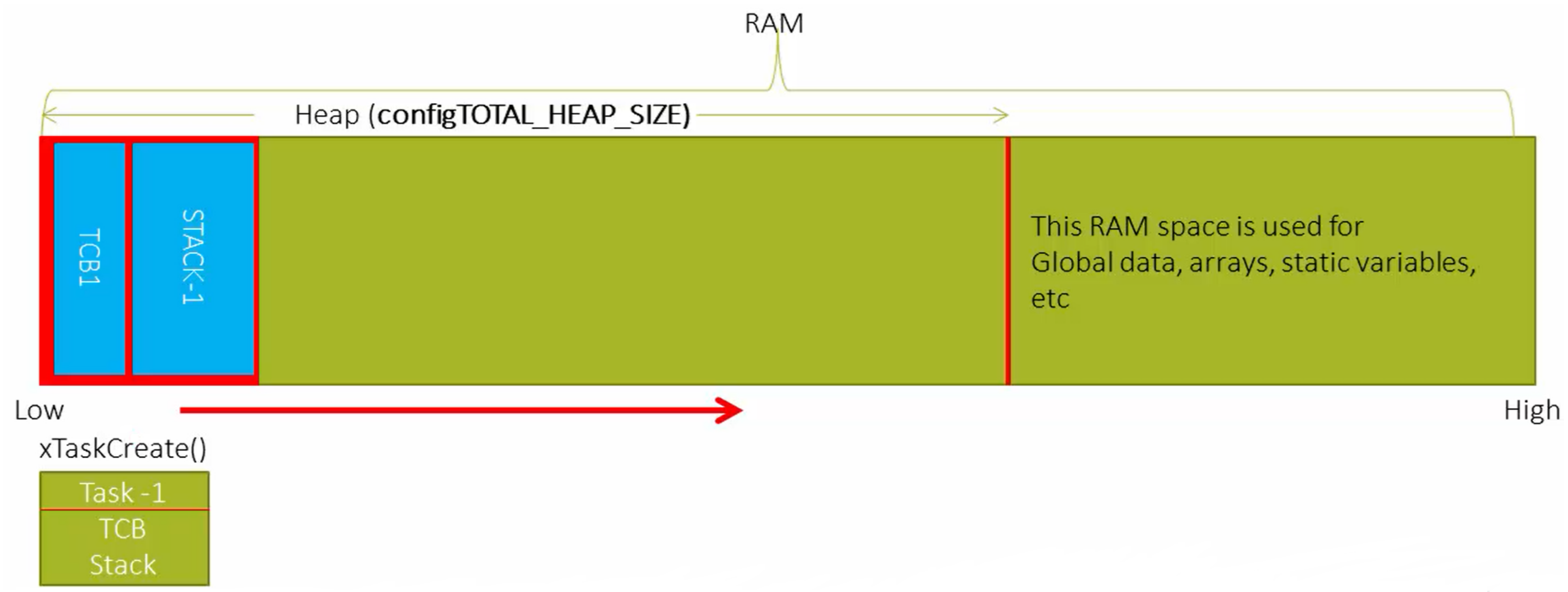
- FreeRTOS Heap
- Task Control block
- Diving Deeper into Task Scheduling
  - SysTick
  - Scheduler
  - SVC
- Task Management
- Quick Test



FreeRTOS HEAP

An Essential Component for Dynamic Memory Allocation

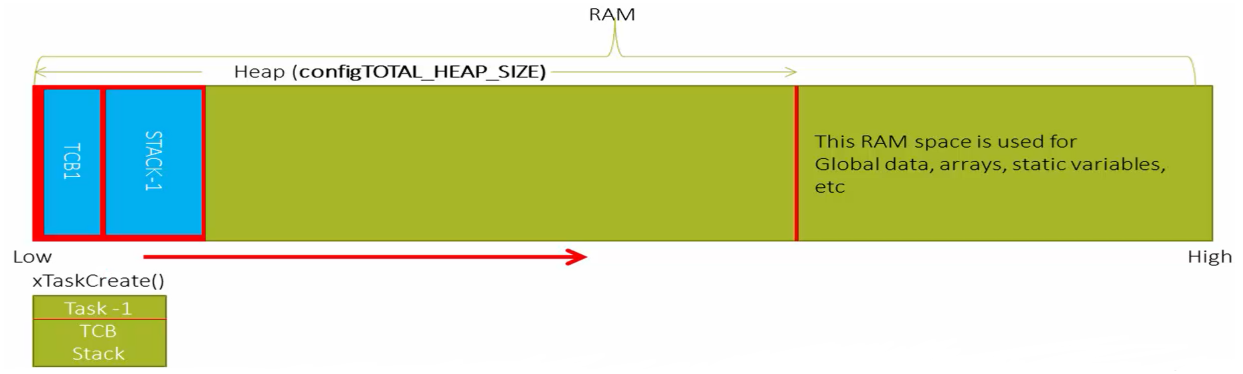
- The FreeRTOS heap plays a critical role in systems utilizing FreeRTOS.
- Allow tasks to allocate and deallocate memory dynamically during runtime.
- Using call to **pvPortMalloc** memory can be allocated by heap manager on heap memory.
- Prevents Fragmentation based on heap configuration used.
- Size of Heap memory is managed by Macro configTOTAL_HEAP_SIZE in **FreeRTOSConfig.h**


RAM
Heap (configTOTAL_HEAP_SIZE)
TCB1
STACK-1
This RAM space is used for Global data, arrays, static variables, etc
Low
High
xTaskCreate()
Task -1
TCB
Stack

FreeRTOS HEAP

An Essential Component for Dynamic Memory Allocation

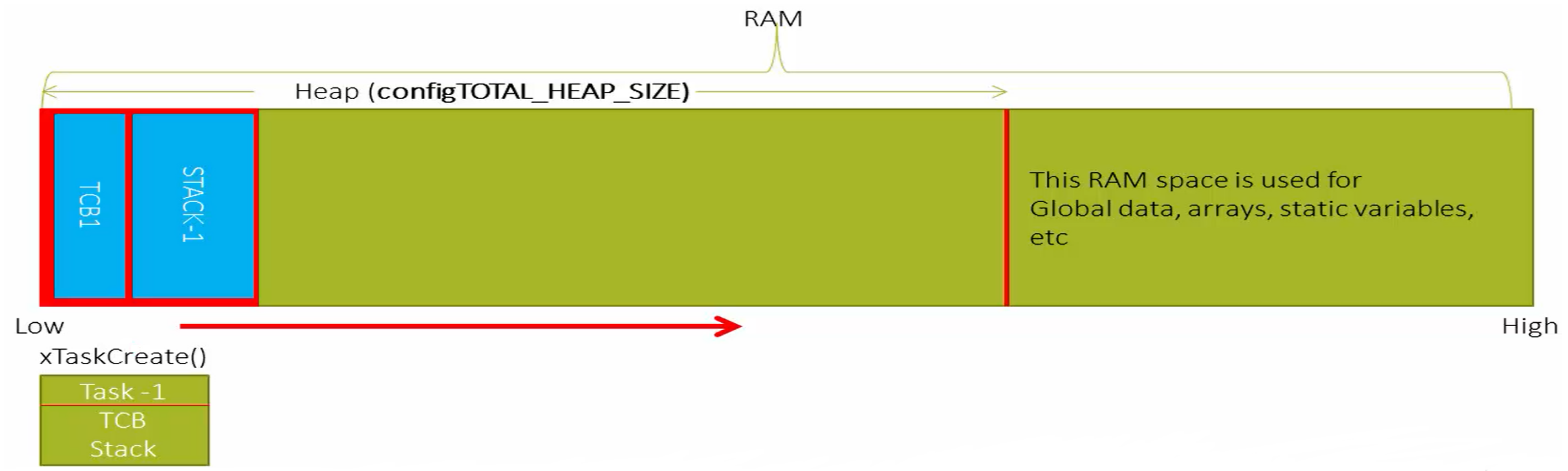
- FreeRTOS memory further divided into RAM space and Heap space.
- This memory is nothing but an array of 1 Byte width created by RTOS Kernel.
- kernel objects like mutexes, semaphores, TCBs those are stored in Heap.
- Please know that the kernel objects are pointed to by variables created in stack space. (Imp.)
- With each Kernel object, the portion of available free memory on RTOS space reduces.
- Heap manager will handle the memory fragmentation problems based on heap manager used.
- Used to dynamically allocate memory for tasks and data.


RAM
Heap (configTOTAL_HEAP_SIZE)
TCB1
STACK-1
This RAM space is used for Global data, arrays, static variables, etc
Low
High
xTaskCreate()
Task -1
TCB
Stack

Heap Initialization?

An Essential Component for Dynamic Memory Allocation

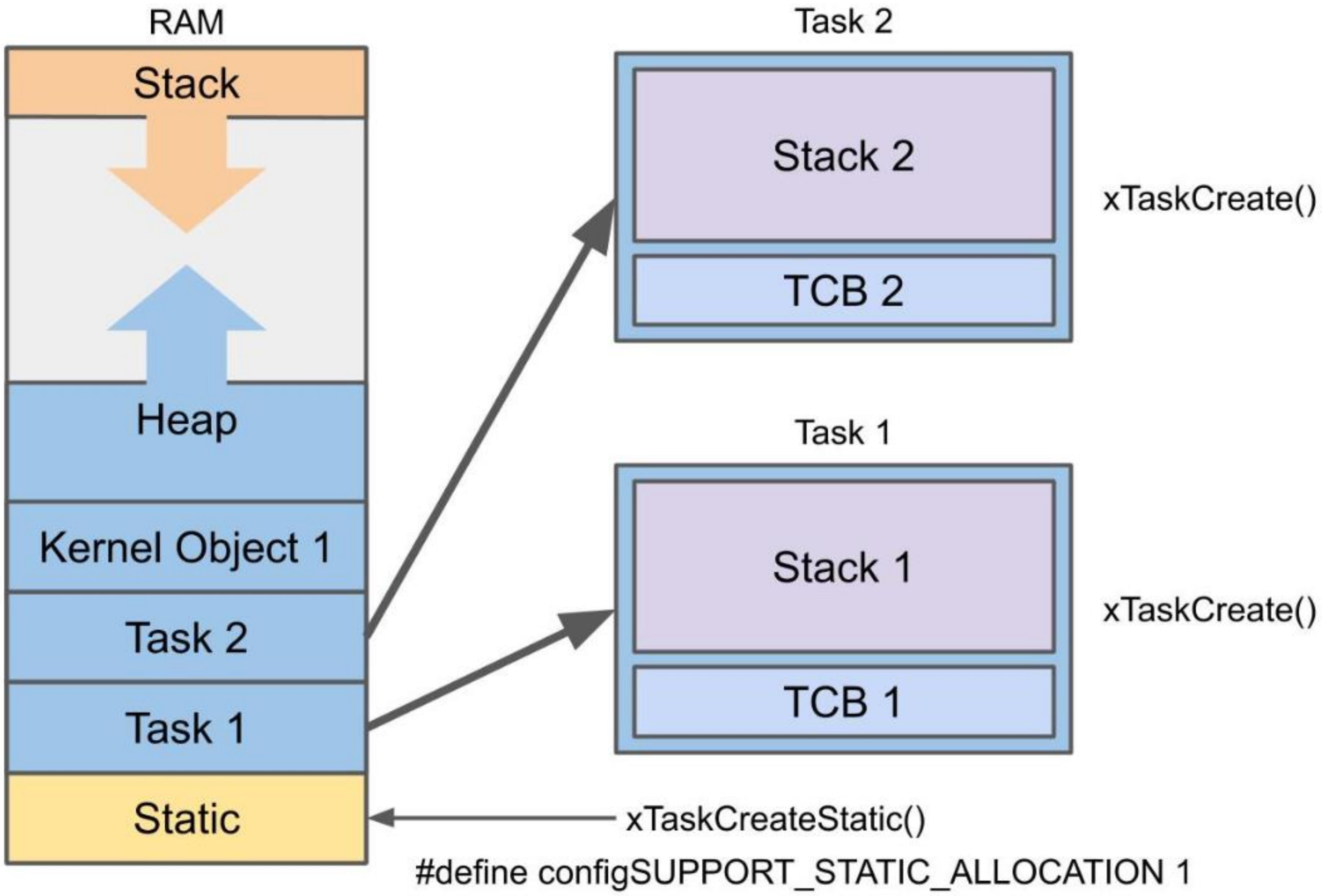
- Where is Heap Memory is being created?
  - Refer to Heap4.c
- How Heap Memory is being implemented?
- Where is Heap Memory is getting initialized?
  - Refer to Heap4.c
- Where is Heap Memory initialization is being done?
- Our Tasks are getting memory statically or dynamically?


RAM
Heap (configTOTAL_HEAP_SIZE)
TCB1
STACK-1
This RAM space is used for Global data, arrays, static variables, etc
Low
High
xTaskCreate()
Task -1
TCB
Stack

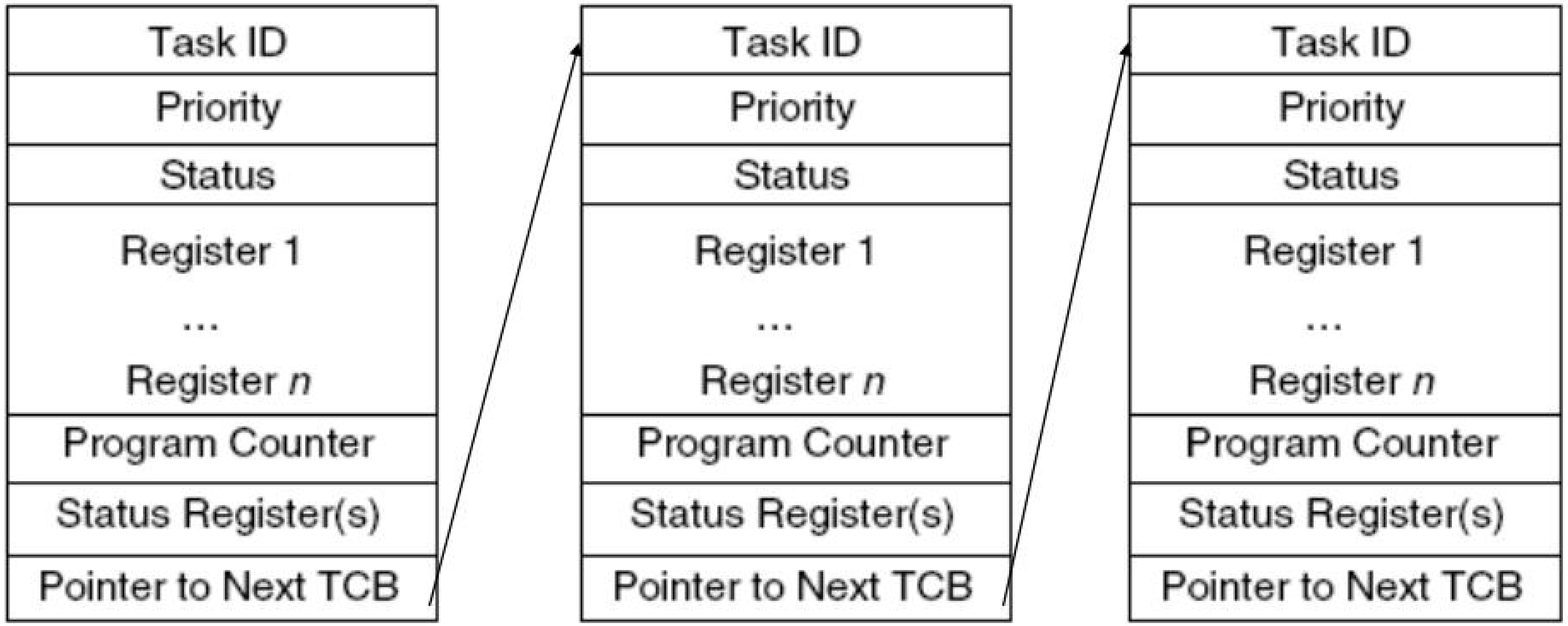
Task Control Block

- The Task Control Block (TCB) is a crucial internal data structure in FreeRTOS.
- TCB is responsible for managing and controlling individual tasks within the real-time operating system.
- The TCB holds essential information about a task, allowing FreeRTOS to effectively schedule, execute, and manage the tasks in a multitasking environment.

Task Control Block (FreeRTOS)



Task Control Block (Generic)



Task Control Block

Task Identification

- Unique task identifier.
- May include a task name or ID

Context Information

- Registers and their values
- Stack pointer information

Task State Management

Current task state (e.g., ready, running, blocked, suspended)

Task Priority

Priority level assigned to the task

Task Control Block

Task Synchronization

Information about synchronization objects (e.g., semaphores, mutexes)

Stack Information

- Stack pointer and size.
- Details for stack management during task execution.

Task Suspension

Flags or information related to task suspension

Task Termination

Exit conditions and cleanup information

FreeRTOS TCB

Refer to **tasks.c** and look for **typedef struct tskTaskControlBlock**

TCB Structure

- A TCB is a data structure that holds information about a task, such as its stack pointer, priority, state, and other context-related data.
- The TCB structure is defined internally in FreeRTOS and is not typically accessible or modifiable by application code.

Automatic TCB Creation

- TCBs are automatically created by the FreeRTOS kernel when you create tasks using the xTaskCreate or xTaskCreateStatic API functions.
- The kernel manages the allocation and initialization of TCBs based on the parameters you provide during task creation.

Task Stack

- Each TCB includes a pointer to the task's stack. The stack is a region of memory allocated for the task's use.
- The stack size is determined by the usStackDepth parameter provided when creating a task.

TASK CONTROL BLOCK

TASK CONTROL BLOCK

Task Priority

- The priority of a task is set during task creation and is stored in the TCB.
- Tasks with higher priorities have a greater chance of being scheduled to run by the FreeRTOS scheduler.

Task State

- The TCB keeps track of the task's state, which can be one of the following: Ready, Running, Blocked, Suspended, or Deleted.
- The FreeRTOS scheduler uses the task state to determine which tasks are eligible to run.

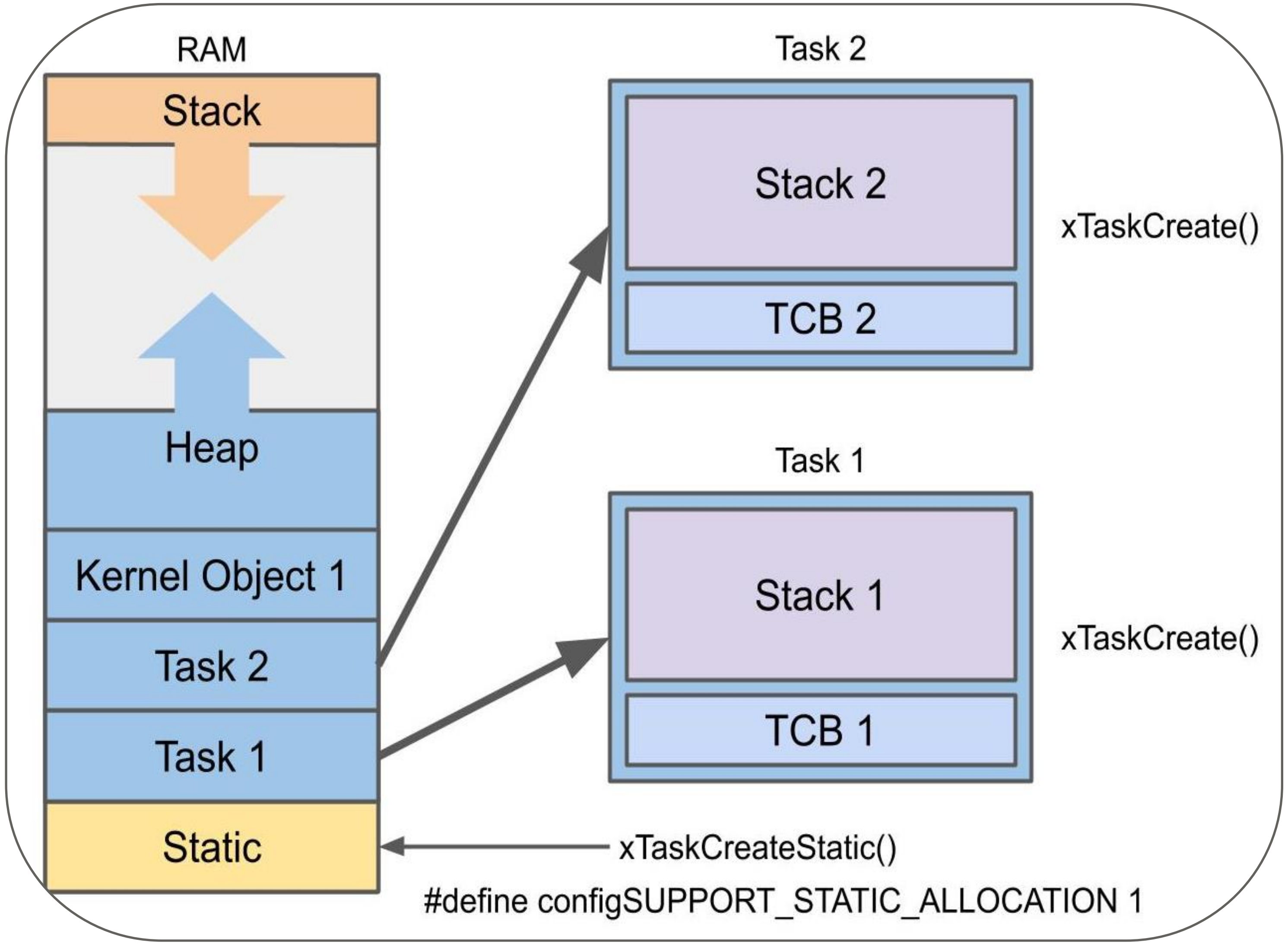
Resource Ownership

- TCBs may include information about resources owned by the task, such as mutexes or semaphores. This helps manage resource ownership and priority inheritance.

Task Control

- FreeRTOS provides various API functions to control and manage tasks after they are created. These functions often require the task handle (which is a pointer to the TCB) to identify and manipulate tasks.

TASK CONTROL BLOCK



SCHEDULER?

- What is a scheduler?
- Is it a Hardware or Software thing?

DIVING DEEP INTO TASK SCHEDULING

- What happens when we create FreeRTOS Tasks?
- Who Invokes the Scheduler?
- How does the Scheduling Work?
- What is a Scheduler Actually?

We need to RECAP following things from ARM Cortex M

- ARM Interrupts and Exceptions
- Priorities
- System Timer Clock

ARM Cortex M

Interrupts and Exceptions

ARM Cortex M4 Interrupts

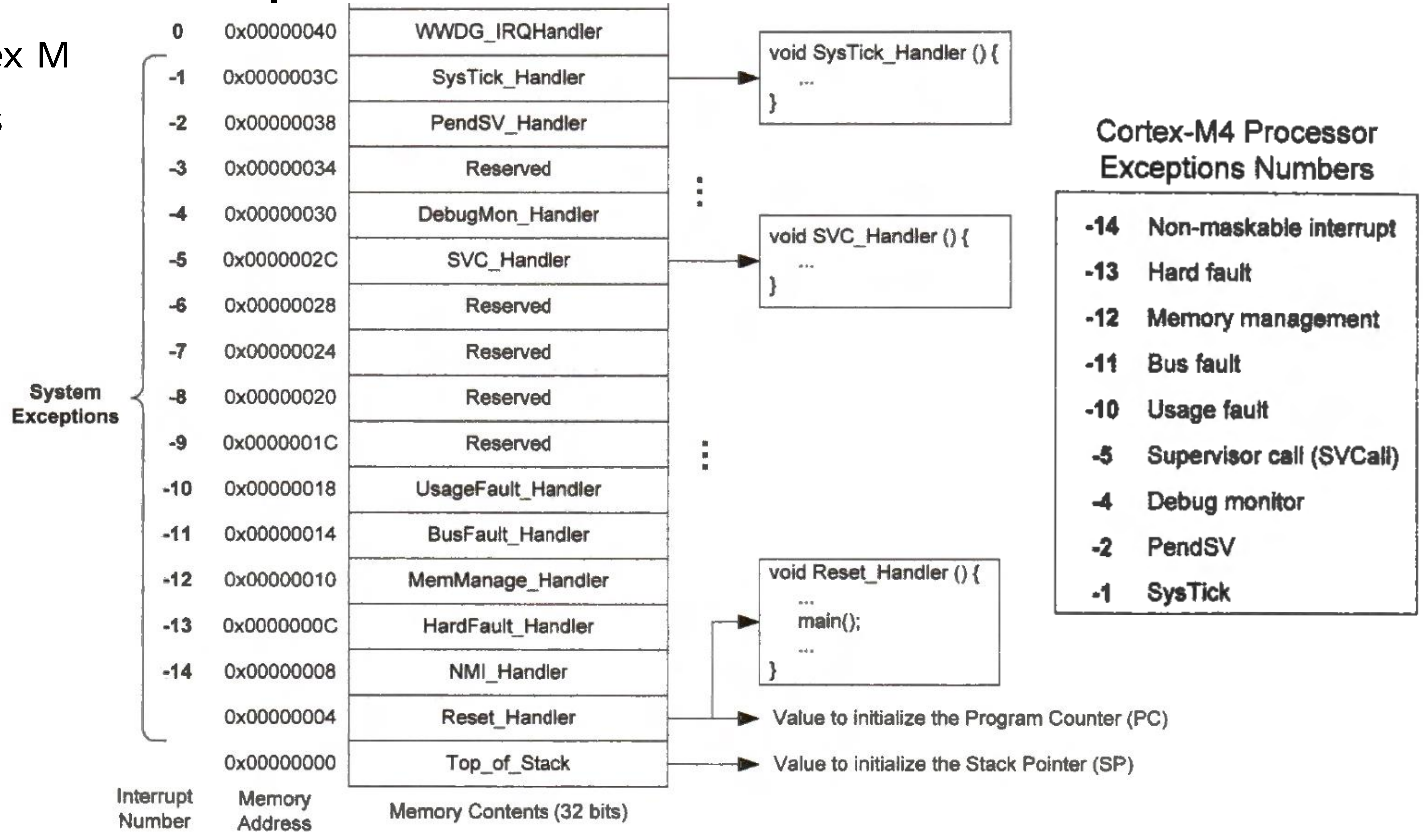
- Cortex-M processors support up to 256 types of interrupts.
- Each interrupt type, excluding the reset interrupt, is identified by a unique number, ranging from -15 to 240.
- Interrupt numbers are defined by ARM and chip manufacturers collectively.
- Interrupt numbers are divided into two groups:
    - **ARM Defined**
        - The first 16 interrupts are system interrupts, also called system exceptions.
        - Exceptions are the interrupts that come from the processor core.
        - The ARM CMSIS library defines all System exceptions by using negative values.
    - **Vendor Defined**
        - The remaining 240 interrupts are peripheral interrupts.
        - The peripheral interrupt numbers start at 0.
        - Peripheral interrupts are defined by chip manufacturers.
- When an interrupt is processed, interrupt number is stored in the program status register (PSR).
- ARM Cortex-M does not store interrupt numbers in two's complement.
- The interrupt number in PSR adds a positive offset of 15 to the CMSIS interrupt number.

Interrupt number in PSR = CMSIS interrupt number+ 15

Recap into ARM Cortex M Core

ARM Cortex M

Exceptions



Interrupt Number	Memory Address	Memory Contents (32 bits)
0	0x00000040	WWDG_IRQHandler
-1	0x0000003C	SysTick_Handler
-2	0x00000038	PendSV_Handler
-3	0x00000034	Reserved
-4	0x00000030	DebugMon_Handler
-5	0x0000002C	SVC_Handler
-6	0x00000028	Reserved
-7	0x00000024	Reserved
-8	0x00000020	Reserved
-9	0x0000001C	Reserved
-10	0x00000018	UsageFault_Handler
-11	0x00000014	BusFault_Handler
-12	0x00000010	MemManage_Handler
-13	0x0000000C	HardFault_Handler
-14	0x00000008	NMI_Handler
	0x00000004	Reset_Handler
	0x00000000	Top_of_Stack

```
void SysTick_Handler () {
    ...
}
```

```
void SVC_Handler () {
    ...
}
```

```
void Reset_Handler () {
    ...
    main();
    ...
}
```

System Exceptions
Value to initialize the Program Counter (PC)
Value to initialize the Stack Pointer (SP)

Cortex-M4 Processor Exceptions Numbers

-14	Non-maskable interrupt
-13	Hard fault
-12	Memory management
-11	Bus fault
-10	Usage fault
-5	Supervisor call (SVCall)
-4	Debug monitor
-2	PendSV
-1	SysTick

Recap into ARM Cortex M Core

ARM Cortex M Exception Handler Address

- There is an interrupt service routine (ISR) associated with each type of interrupt.
- Cortex-M stores the starting memory address of every ISR in a special array called the **interrupt vector table**.
- For a given interrupt number i defined in CMSIS, the memory address of its corresponding ISR is located at the (i + 16)th entry in the interrupt vector table.
- The interrupt vector table is stored at the memory address 0x00000004.
- Each entry in the table represents a memory address, each entry takes four bytes in memory.
- **Address of ISR = InterruptVectorTable[i + 15] or (0x00000004 + 4 x (i+15))**
- **Example SysTick (-1) ISR: (0x00000004 + 4 x (-1+15)) = 0x0000003C**
- **Example Reset (-15) ISR: (0x00000004 + 4 x (-15+15)) = 0x00000004**

Recap into ARM Cortex M Core

ARM Cortex M Exception Handler Address

- Calculate the ISR address for

SVC	(-5)	0x2C
Bus Fault	(-11)	0x14
PendSV	(-2)	0x38
Hard Fault	(-13)	0x0C
SysTick	(-1)	0x3C

SysTick

Heartbeat of ARM Cortex M CPU Core

SysTick

Description

- SysTick timer is a system timer included in many ARM Cortex-M microcontrollers.
- 24-bit down-counter that can be used for various timing and control purposes.

Basic Functionality

- Designed to provide a simple, general-purpose timer for the system.
- Used for generating time delays in software, implementing time-based functions, or serving as a time reference for the system.

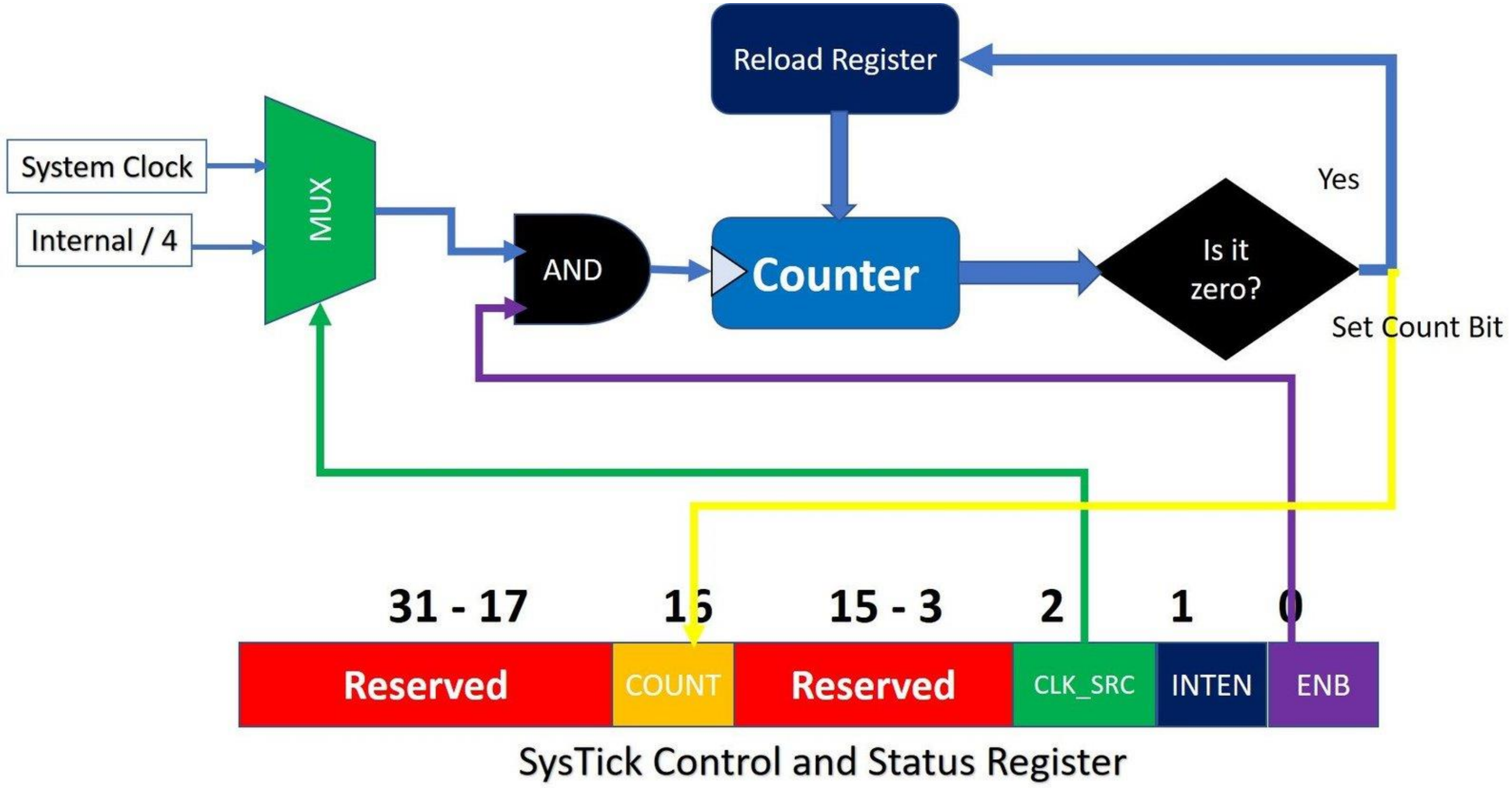
Clock Source

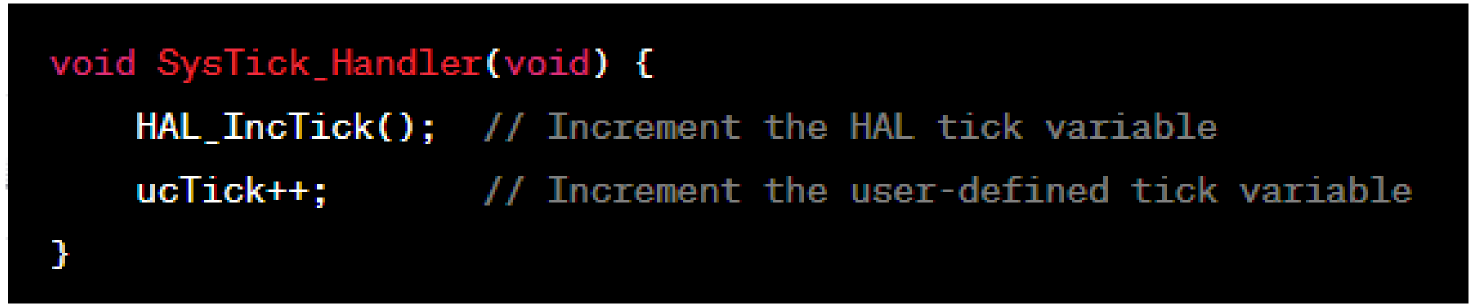
- SysTick can use either processor clock or external clock source as its time base.
- Clock source is determined by the configuration of the STK_CTRL register.

Interrupts

- SysTick can be configured to generate interrupts when the counter reaches zero.
- **This feature is often used in real-time operating systems (RTOS) to implement time slices or time-based task scheduling.**

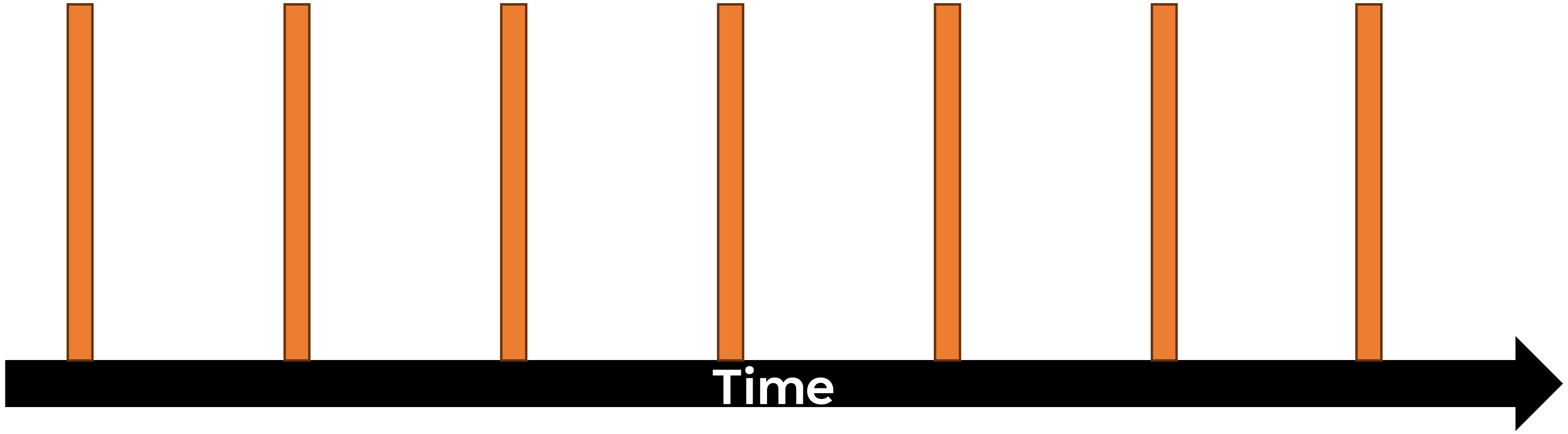
SysTick


Reload Register
System Clock
Internal / 4
MUX
AND
Counter
Is it zero?
Yes
Set Count Bit
31 - 17
16
15 - 3
2
1
0
Reserved
COUNT
Reserved
CLK_SRC
INTEN
ENB
SysTick Control and Status Register



```
void SysTick_Handler(void) {
    HAL_IncTick();  // Increment the HAL tick variable
    ucTick++;       // Increment the user-defined tick variable
}
```

What is ucTick in SysTick_Handler?



SysTick Interrupts

SVC

Base of OS implementation

SVC
SUPERVISOR CALL

Description

Special exception in the ARM Cortex-M architecture that allows the processor to switch from thread mode to handler mode.

Usage in OS

The SVC exception is often used by real-time operating systems (RTOS) to implement system calls. A system call is a request made by a program to the operating system's kernel for a specific operation or service.

SVC Instruction

To trigger the SVC exception, the SVC instruction is used..The SVC instruction is followed by an immediate value, called the SVC number, which corresponds to the specific system call or service requested.

Usage of SVC

- When a task needs a service that requires a context switch, it triggers an SVC call.
- **In case of FreeRTOS and ARM Cortex M, SVC is used to start the First Task.**
- **We can verify the above statement from Segger Trace.**

PednSV

Base of OS implementation

PendSV

Description

PendSV (Pending Supervisor Call) exception is a special exception in ARM Cortex-M microcontrollers that plays a crucial role in context switching and task scheduling in real-time operating systems (RTOS) like FreeRTOS

Usage in OS

- PendSV is specifically designed to facilitate context switching in a multitasking environment.
- Provides a mechanism to trigger context switch at the end of the current task.

Context Switching

- Process of saving the context (registers, stack pointer, etc.) of the currently running task and restoring the context of a different task.

Priority

- PendSV is assigned the lowest priority among exceptions in the NVIC.
- **PendSV can be preempted by other exceptions, allowing higher-priority interrupts to be handled quickly.**
- **PendSV is the actual Scheduler, in case of FreeRTOS port on Cortex M4.**

SCHEDULER IMPLEMENTATION

- Role of SysTick
- Role of SVC
- Role of PendSV

FreeRTOS Scheuler Implementation

- Scheduler is combination of Hardware + Software component.
- Since, Scheduler directly controls what will run on CPU at a given time, the Hardware involved is ARM (Core) specific.
- As discussed before, A given FreeRTOS Port has dependency on the Core, the implementation of FreeRTOS services like Scheduling may have different implementation specific to a given architecture.
- Architecture specific implementation of these services are defined in port.c and portmacro.h
- Scheduler specific implementation for ARM Cortex M4 core is defined in port.c file.
- Port.c file exports generic wrapper functions implemented on top of hardware dependent code.

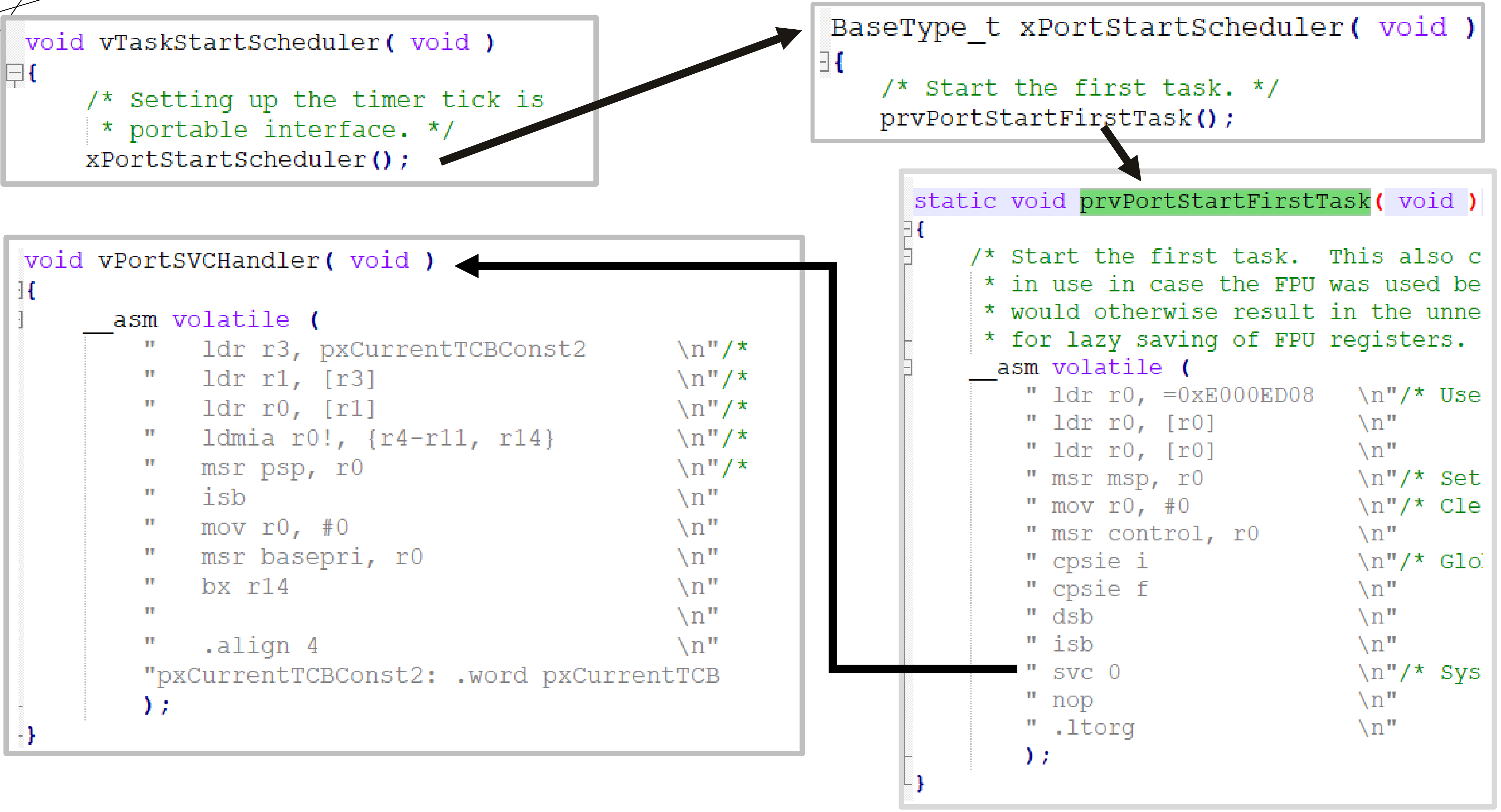
FreeRTOS Scheduler Implementation Contd..

- **SVC handler** is exported as "**vPortSVCHandler()**", implementation is M4 specific.
    - **SVC Handler** is the one which launches the very first task after the scheduler is started in **main()**.
    - SVC called in "**prvPortStartFirstTask()",** and call to "**prvPortStartFirstTask()"** was done by "**vTaskStartScheduler()"** in **main().**

- **PendSV handler** is exported as "**xPortPendSVHandler()**", implementation is M4 specific.4
    - PendSV is the scheduler in case of Cortex M4 FreeRTOS, implements context switching between tasks.
    - PendSV gets triggered periodically by the **SysTick** handler, as **SysTick** Handler sets the pending bit for PendSV.
    - PendSV performs the Context Switching of Tasks by:
        - Getting the TCB of currently running Task. Getting the **Top of Stack pointer** from TCB.
        - Saving the **context** of currently running task on task's stack and updating the **Top of Stack pointer** in TCB**.**
        - Branch to "**vTaskSwitchContext()**", get the TCB info of next **Ready Task**.
        - Get the **Top of Stack pointer** from new Task's TCB.
        - Load the last known context of new Task from it's own stack.
        - Update the Process Stack Pointer

- **SysTick handler** is exported as "xPortSysTickHandler()", implementation is M4 specific.
    - **SysTick handler is the Time-Slice generation unit, manages RTOS Tick.**
    - **SysTick is the service which sets the pending bit for PendSV handler (Scheduler).**
    - **In simple words, Scheduler is invoked by the SysTick Handler, so scheduler is invoked at every Tick interrupt.**

SVC and First Task Launch



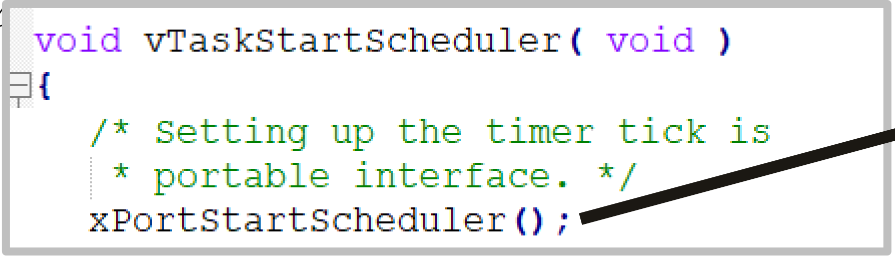
```c
void vTaskStartScheduler( void )
{
    /* Setting up the timer tick is
     * portable interface. */
    xPortStartScheduler();
```

```c
BaseType_t xPortStartScheduler( void )
{
    /* Start the first task. */
    prvPortStartFirstTask();
```

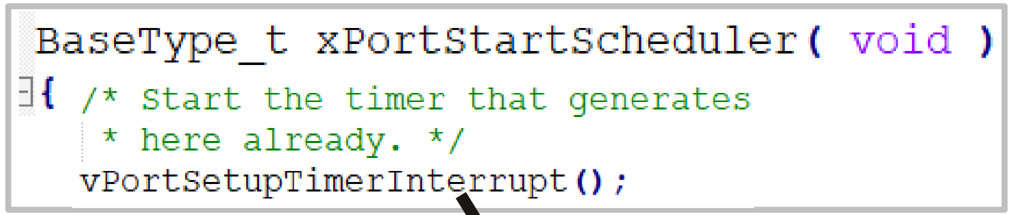
```c
static void prvPortStartFirstTask( void )
{
    /* Start the first task.  This also c
     * in use in case the FPU was used be
     * would otherwise result in the unne
     * for lazy saving of FPU registers.
    __asm volatile (
        " ldr r0, =0xE000ED08   \n"/* Use
        " ldr r0, [r0]          \n"
        " ldr r0, [r0]          \n"
        " msr msp, r0           \n"/* Set
        " mov r0, #0            \n"/* Cle
        " msr control, r0       \n"
        " cpsie i               \n"/* Glo
        " cpsie f               \n"
        " dsb                   \n"
        " isb                   \n"
        " svc 0                 \n"/* Sys
        " nop                   \n"
        " .ltorg                \n"
        );
}
```

```c
void vPortSVCHandler( void )
{
    __asm volatile (
        "   ldr r3, pxCurrentTCBConst2      \n"/*
        "   ldr r1, [r3]                    \n"/*
        "   ldr r0, [r1]                    \n"/*
        "   ldmia r0!, {r4-r11, r14}        \n"/*
        "   msr psp, r0                     \n"/*
        "   isb                             \n"
        "   mov r0, #0                      \n"
        "   msr basepri, r0                 \n"
        "   bx r14                          \n"
        "                                   \n"
        "   .align 4                        \n"
        "pxCurrentTCBConst2: .word pxCurrentTCB
        );
}
```

SysTick and PendSV working

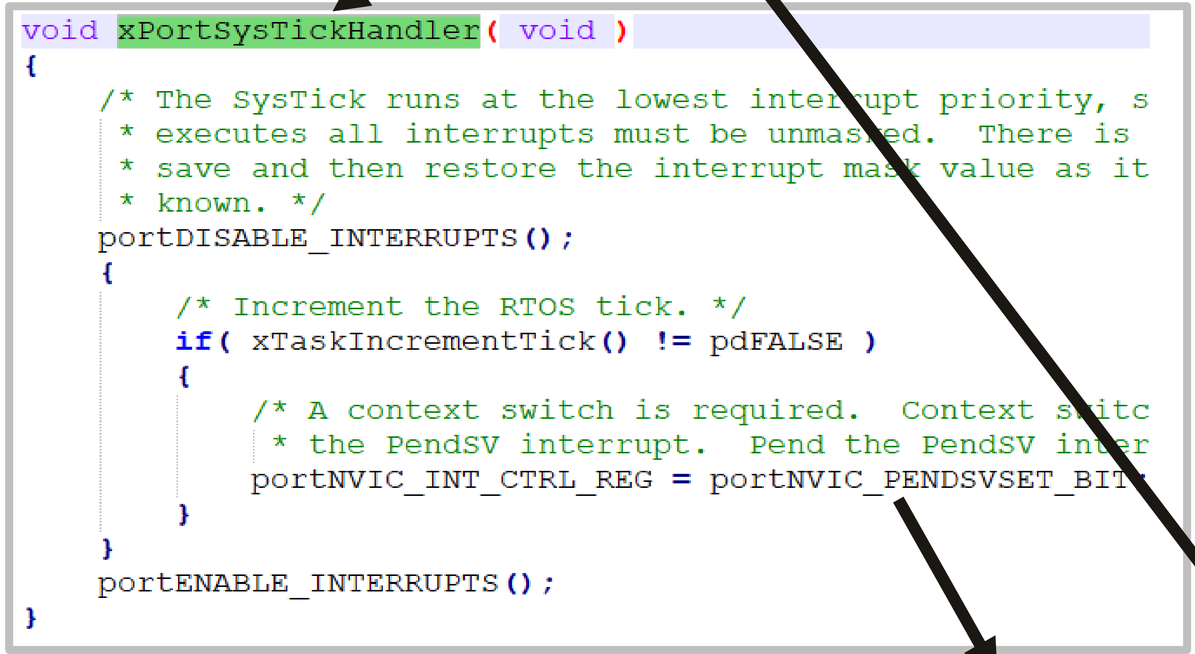


```
void vTaskStartScheduler( void )
{
    /* Setting up the timer tick is
     * portable interface. */
    xPortStartScheduler();
```

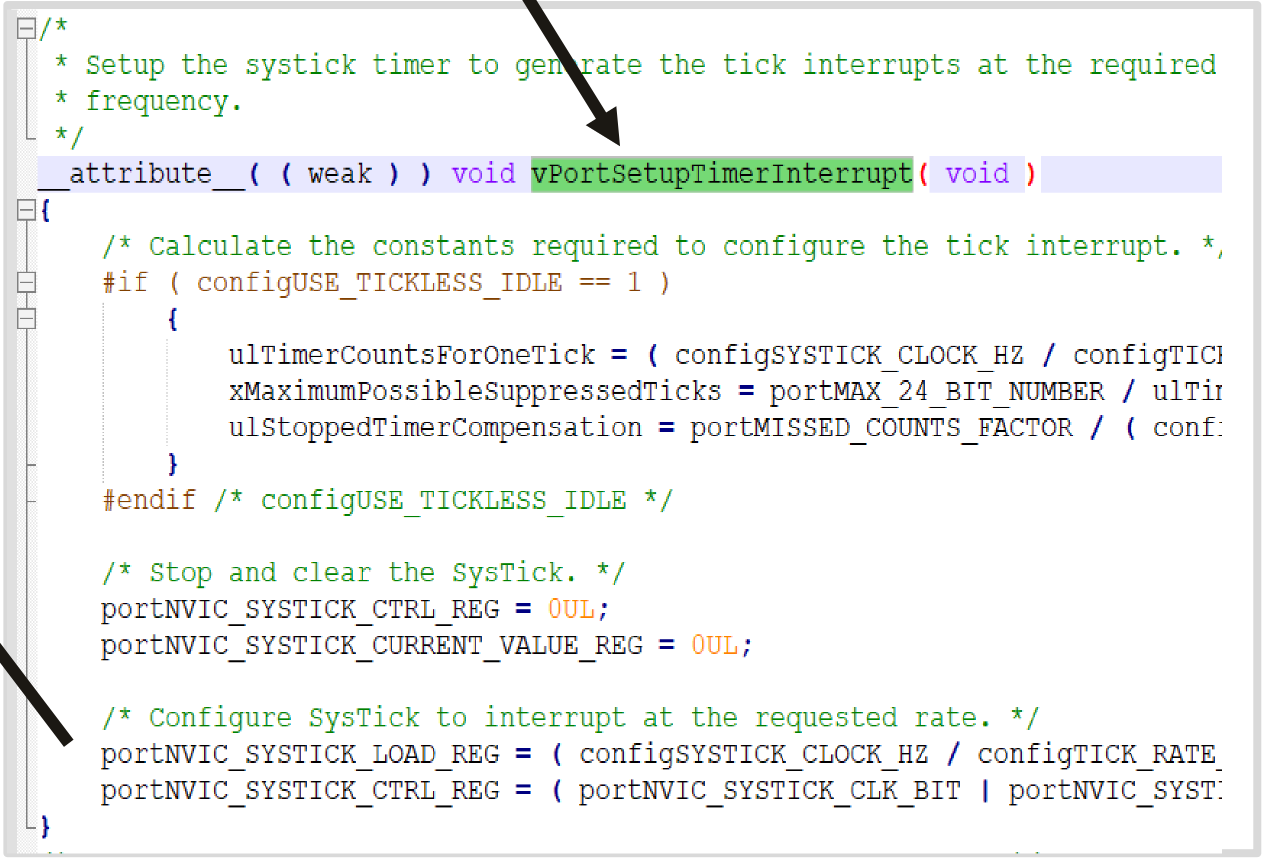


```
BaseType_t xPortStartScheduler( void )
{ /* Start the timer that generates
   * here already. */
  vPortSetupTimerInterrupt();
```

SysTick is now firing interrupts at every Tick Period



```
void xPortSysTickHandler( void )
{
    /* The SysTick runs at the lowest interrupt priority, s
     * executes all interrupts must be unmasked.  There is
     * save and then restore the interrupt mask value as it
     * known. */
    portDISABLE_INTERRUPTS();
    {
        /* Increment the RTOS tick. */
        if( xTaskIncrementTick() != pdFALSE )
        {
            /* A context switch is required.  Context switc
             * the PendSV interrupt.  Pend the PendSV inter
            portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT
        }
    }
    portENABLE_INTERRUPTS();
}
```



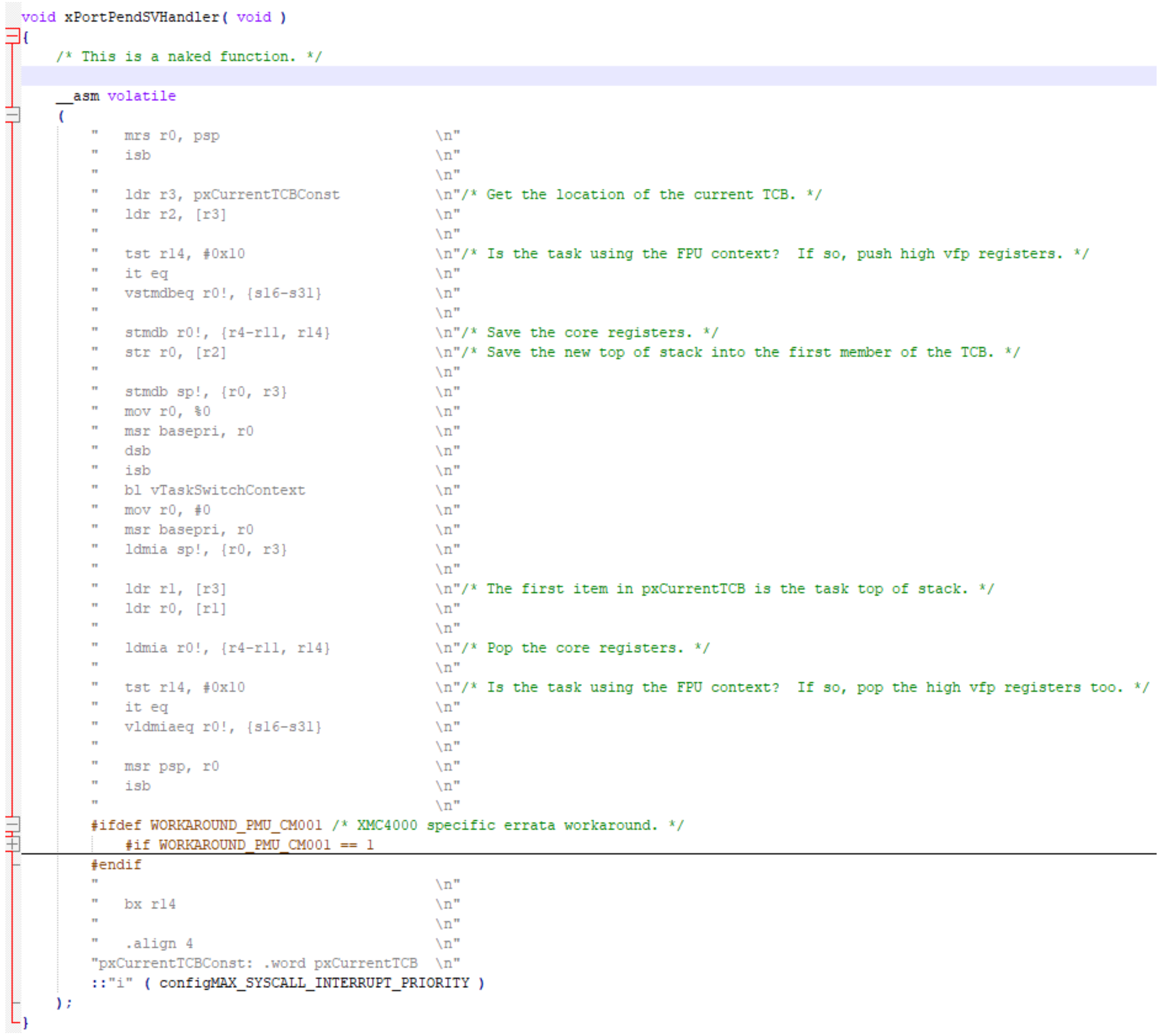
```
/*
 * Setup the systick timer to generate the tick interrupts at the required
 * frequency.
 */
__attribute__( ( weak ) ) void vPortSetupTimerInterrupt( void )
{
    /* Calculate the constants required to configure the tick interrupt. *
    #if ( configUSE_TICKLESS_IDLE == 1 )
        {
            ulTimerCountsForOneTick = ( configSYSTICK_CLOCK_HZ / configTIC
            xMaximumPossibleSuppressedTicks = portMAX_24_BIT_NUMBER / ulTi
            ulStoppedTimerCompensation = portMISSED_COUNTS_FACTOR / ( conf
        }
    #endif /* configUSE_TICKLESS_IDLE */

    /* Stop and clear the SysTick. */
    portNVIC_SYSTICK_CTRL_REG = 0UL;
    portNVIC_SYSTICK_CURRENT_VALUE_REG = 0UL;

    /* Configure SysTick to interrupt at the requested rate. */
    portNVIC_SYSTICK_LOAD_REG = ( configSYSTICK_CLOCK_HZ / configTICK_RATE_
    portNVIC_SYSTICK_CTRL_REG = ( portNVIC_SYSTICK_CLK_BIT | portNVIC_SYST
}
```

Pending Bit for PendSV Exception is set to 1 by SysTick handler, once interrupts are enabled again, PendSV handler will execute

SysTick and PendSV working

Pending Bit for PendSV Exception is set to 1 by SysTick handler, once interrupts are enabled again, PendSV handler will execute

- Get TOP of Stack
- Get Current TCB
- Save Context of Task to it's Stack
- Update Top of Stack
- Get next Task TCB
- Get Top of Stack
- POP registers from Stack to CPU
- Branch to stored PC



```
void xPortPendSVHandler( void )
{
    /* This is a naked function. */

    __asm volatile
    (
        "   mrs r0, psp                         \n"
        "   isb                                 \n"
        "                                       \n"
        "   ldr r3, pxCurrentTCBConst           \n"/* Get the location of the current TCB. */
        "   ldr r2, [r3]                        \n"
        "                                       \n"
        "   tst r14, #0x10                      \n"/* Is the task using the FPU context?  If so, push high vfp registers. */
        "   it eq                               \n"
        "   vstmdbeq r0!, {s16-s31}             \n"
        "                                       \n"
        "   stmdb r0!, {r4-r11, r14}            \n"/* Save the core registers. */
        "   str r0, [r2]                        \n"/* Save the new top of stack into the first member of the TCB. */
        "                                       \n"
        "   stmdb sp!, {r0, r3}                 \n"
        "   mov r0, %0                          \n"
        "   msr basepri, r0                     \n"
        "   dsb                                 \n"
        "   isb                                 \n"
        "   bl vTaskSwitchContext               \n"
        "   mov r0, #0                          \n"
        "   msr basepri, r0                     \n"
        "   ldmia sp!, {r0, r3}                 \n"
        "                                       \n"
        "   ldr r1, [r3]                        \n"/* The first item in pxCurrentTCB is the task top of stack. */
        "   ldr r0, [r1]                        \n"
        "                                       \n"
        "   ldmia r0!, {r4-r11, r14}            \n"/* Pop the core registers. */
        "                                       \n"
        "   tst r14, #0x10                      \n"/* Is the task using the FPU context?  If so, pop the high vfp registers too. */
        "   it eq                               \n"
        "   vldmiaeq r0!, {s16-s31}             \n"
        "                                       \n"
        "   msr psp, r0                         \n"
        "   isb                                 \n"
        "                                       \n"
        #ifdef WORKAROUND_PMU_CM001 /* XMC4000 specific errata workaround. */
            #if WORKAROUND_PMU_CM001 == 1
        #endif
        "                                       \n"
        "   bx r14                              \n"
        "                                       \n"
        "   .align 4                            \n"
        "pxCurrentTCBConst: .word pxCurrentTCB  \n"
        ::"i" ( configMAX_SYSCALL_INTERRUPT_PRIORITY )
    );
}
```

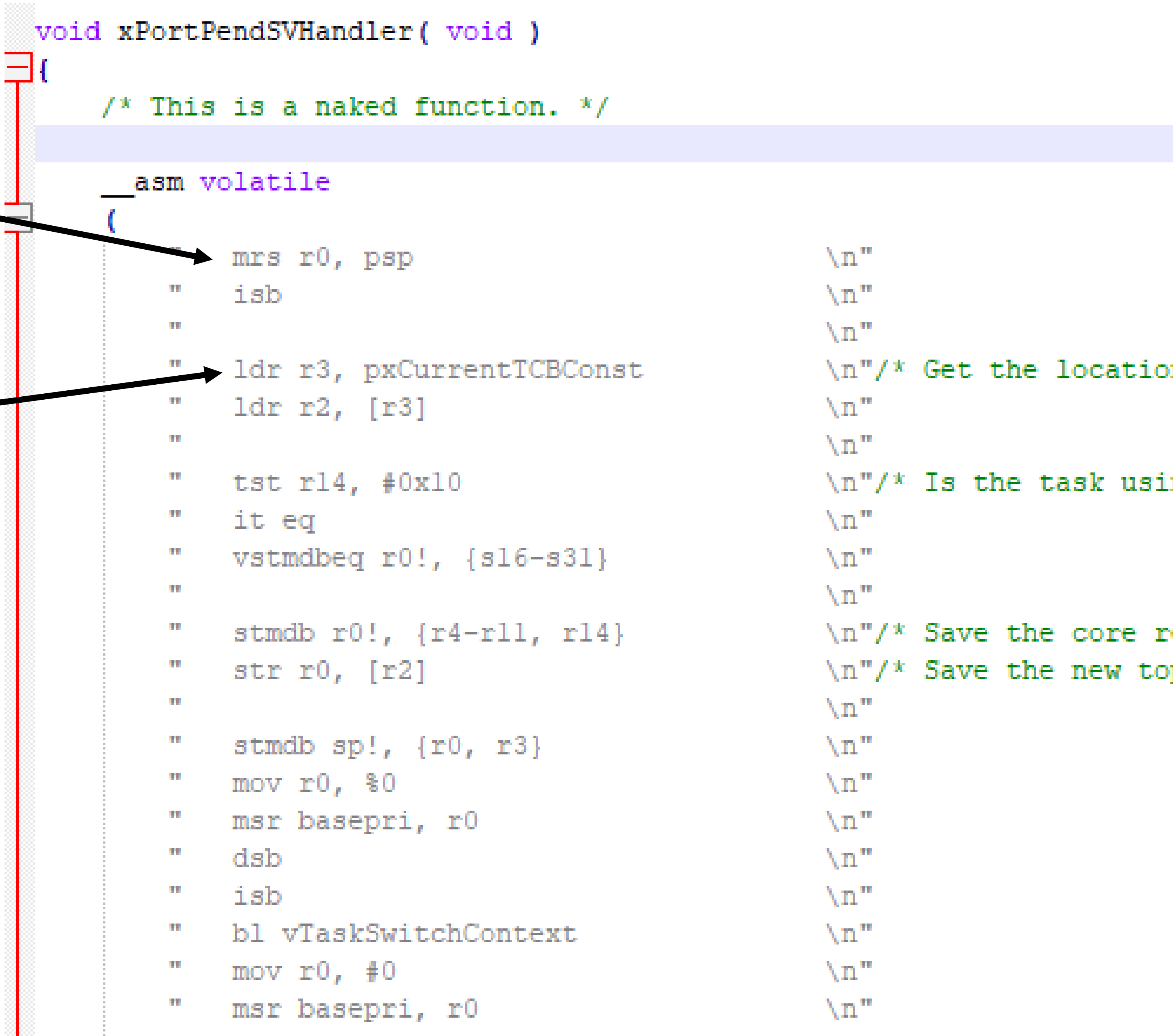
- **Get the <u>Top of Stack</u> from PSP**

Top of Stack is just a pointer to task's private stack. PSP is process stack pointer which points to the task's stack.

- **Get Current Task TCB from pxCurrentTCBConst**

This TCB provides access to the Task Control Block which contains fields to task's stack.

Here the kernel stores the task's context which includes registers, stack pointer and last known program counter.



```
void xPortPendSVHandler( void )
{
    /* This is a naked function. */

    __asm volatile
    (
    "   mrs r0, psp                         \n"
    "   isb                                 \n"
    "                                       \n"
    "   ldr r3, pxCurrentTCBConst           \n"/* Get the locatio
    "   ldr r2, [r3]                        \n"
    "                                       \n"
    "   tst r14, #0x10                      \n"/* Is the task usi
    "   it eq                               \n"
    "   vstmdbeq r0!, {s16-s31}             \n"
    "                                       \n"
    "   stmdb r0!, {r4-r11, r14}            \n"/* Save the core r
    "   str r0, [r2]                        \n"/* Save the new to
    "                                       \n"
    "   stmdb sp!, {r0, r3}                 \n"
    "   mov r0, %0                          \n"
    "   msr basepri, r0                     \n"
    "   dsb                                 \n"
    "   isb                                 \n"
    "   bl vTaskSwitchContext               \n"
    "   mov r0, #0                          \n"
    "   msr basepri, r0                     \n"
```

- **Save Context and update R0 (stack pointer) value**

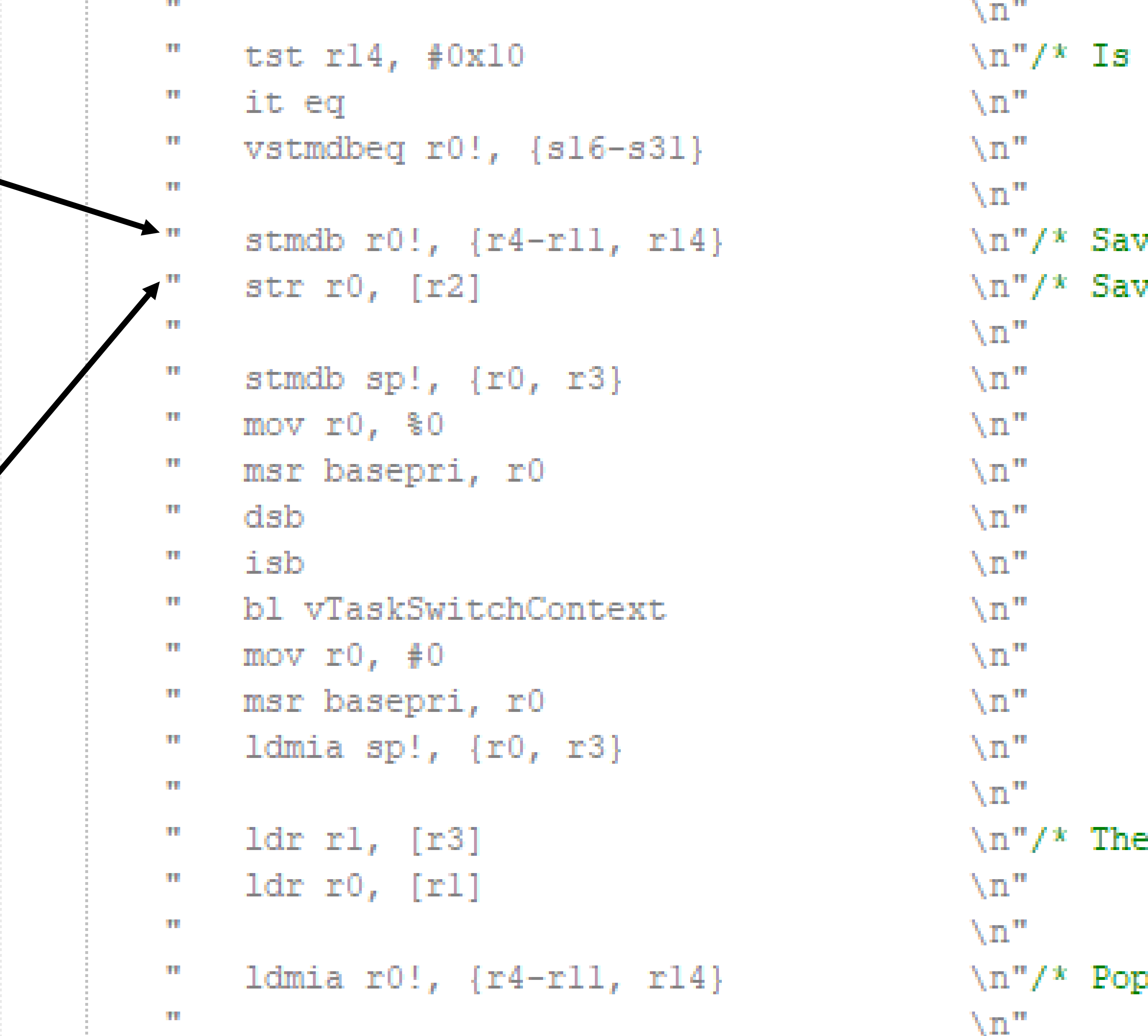
Current Task's context is saved on the Top of Stack.

Registers from r4 till r14 are saved on the task's stack.

r14 contains the PSP for current stack.

- **Update new TOP of Stack by storing stack pointer value from R0**

With new data pushed on the task's stack, the new stack pointer value (r0) is stored in "Top of Stack" location by dereferencing r2.



```
"                                  \n"
"   tst r14, #0x10                 \n"/* Is
"   it eq                          \n"
"   vstmdbeq r0!, {s16-s31}        \n"
"                                  \n"
"   stmdb r0!, {r4-r11, r14}       \n"/* Sav
"   str r0, [r2]                   \n"/* Sav
"                                  \n"
"   stmdb sp!, {r0, r3}            \n"
"   mov r0, %0                     \n"
"   msr basepri, r0                \n"
"   dsb                            \n"
"   isb                            \n"
"   bl vTaskSwitchContext          \n"
"   mov r0, #0                     \n"
"   msr basepri, r0                \n"
"   ldmia sp!, {r0, r3}            \n"
"                                  \n"
"   ldr r1, [r3]                   \n"/* The
"   ldr r0, [r1]                   \n"
"                                  \n"
"   ldmia r0!, {r4-r11, r14}       \n"/* Pop
"                                  \n"
```

- **Push the R0 - R3 registers on MSP (Main Stack Pointer).**

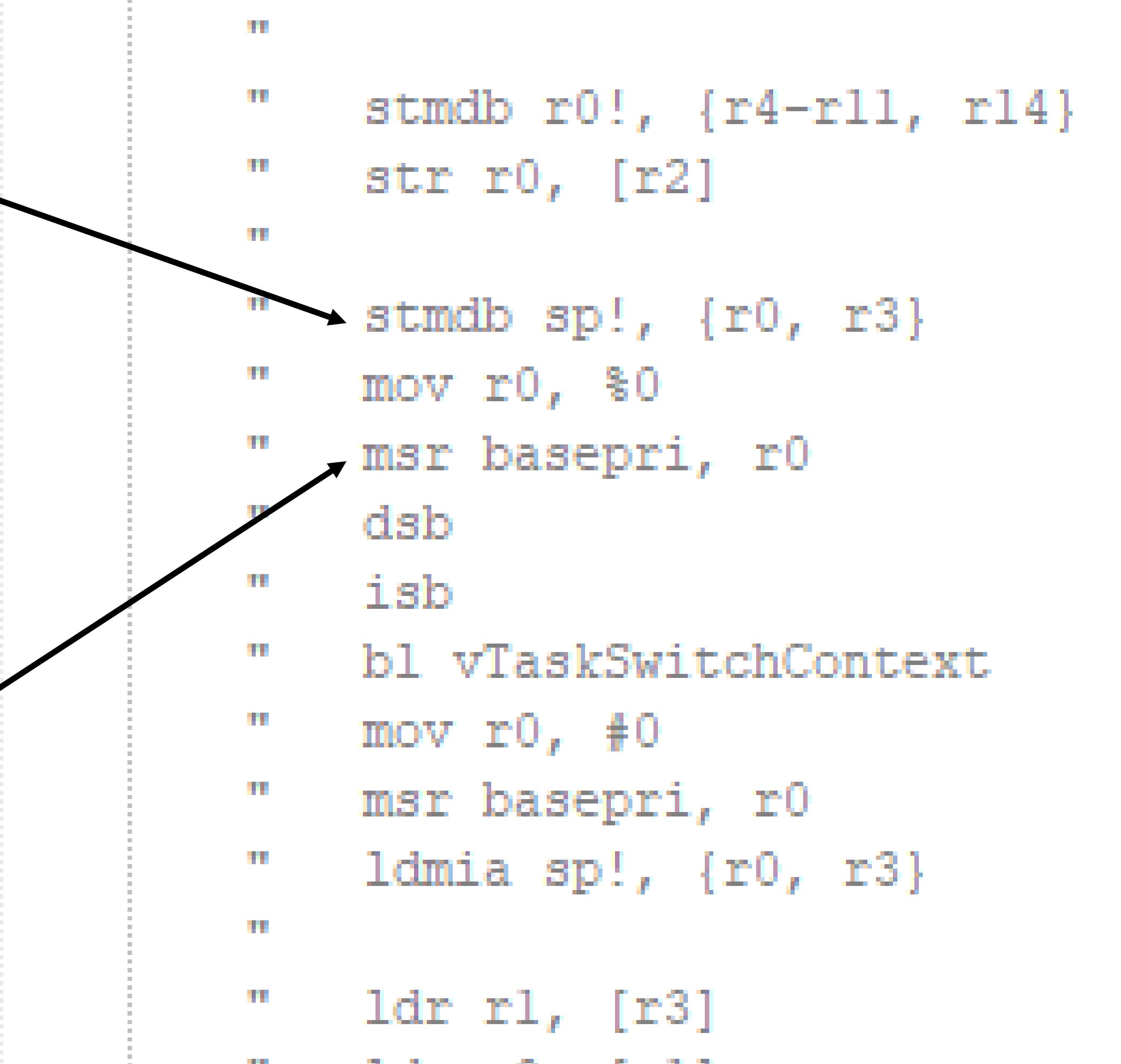
MSP points to kernel stack in handler mode.

This Kernel Stack is used only in case of stacking crucial registers used in context switch.

Since R3 is pointing to symbol pxCurrentTCBConst, the reg is stacked on MSP for later usage.

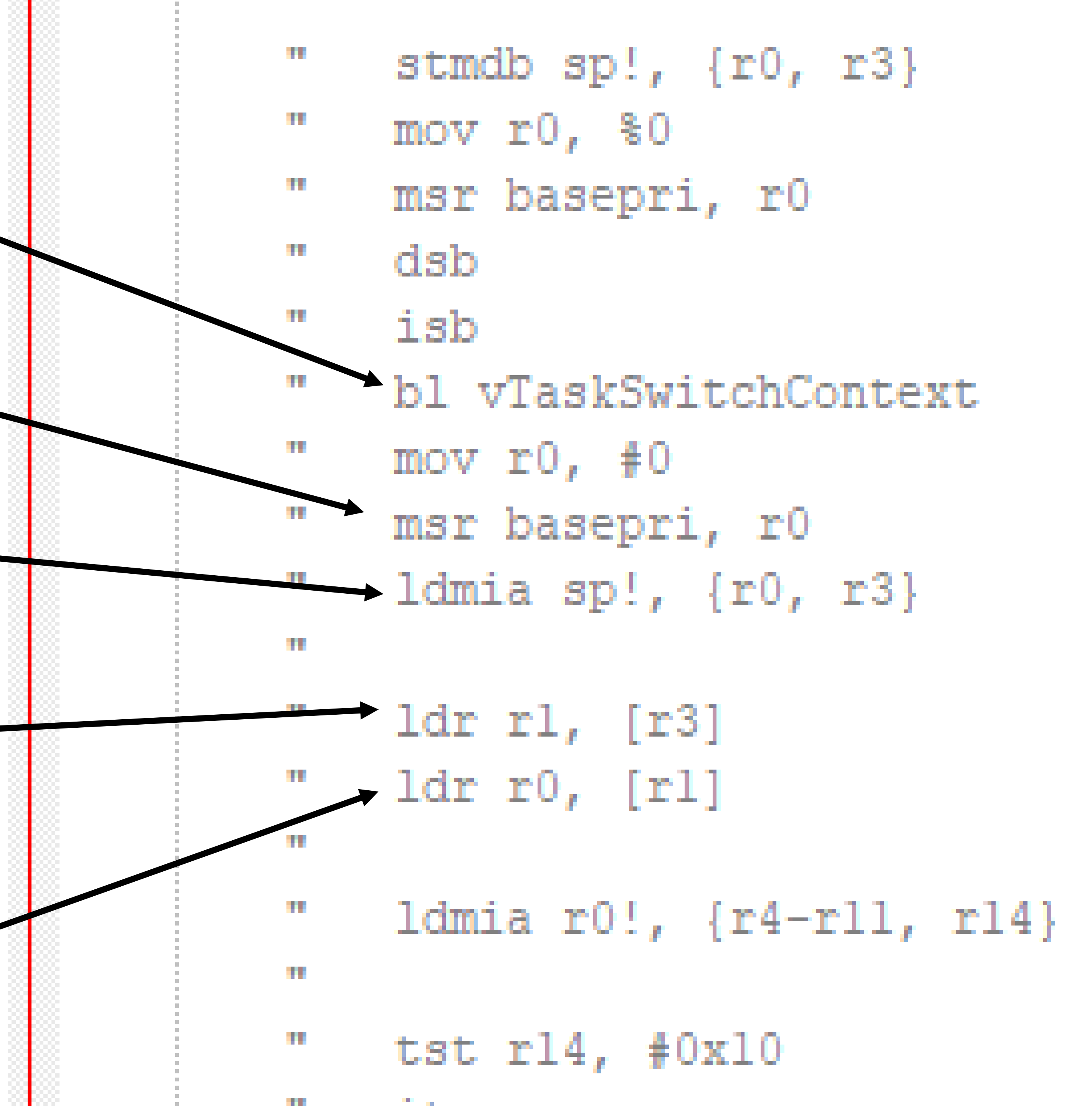
- **Set interrupt priority lower than Sys Call Priority are disabled, so call to vTaskSwitchContext completes without interruption.**

%0 is an argument provided in call to this asm routine.



```
"
"    stmdb r0!, {r4-r11, r14}
"    str r0, [r2]
"
"    stmdb sp!, {r0, r3}
"    mov r0, %0
"    msr basepri, r0
"    dsb
"    isb
"    bl vTaskSwitchContext
"    mov r0, #0
"    msr basepri, r0
"    ldmia sp!, {r0, r3}
"
"    ldr r1, [r3]
```

- **Branch to vTaskSwitchContext, this will update the symbol "pxCurrentTCBConst" with next ready task's TCB.**
- **Enable all interrupt by resetting basepri to 0. (Using Immediate value #0).**
- **Unstack the R0 – R3 registers from Stack currently in use by this handler i.e MSP.**
- **Know that unstacked R3 register still points to symbol "pxCurrentTCBConst" which now have the address to new TCB**
- **Derefrence R3 and get the address of TCB. Further dereference to get address of first variable in TCB i.e "Top of Stack"**



```
"    stmdb sp!, {r0, r3}
"    mov r0, %0
"    msr basepri, r0
"    dsb
"    isb
"    bl vTaskSwitchContext
"    mov r0, #0
"    msr basepri, r0
"    ldmia sp!, {r0, r3}
"
"    ldr r1, [r3]
"    ldr r0, [r1]
"
"    ldmia r0!, {r4-r11, r14}
"
"    tst r14, #0x10
```

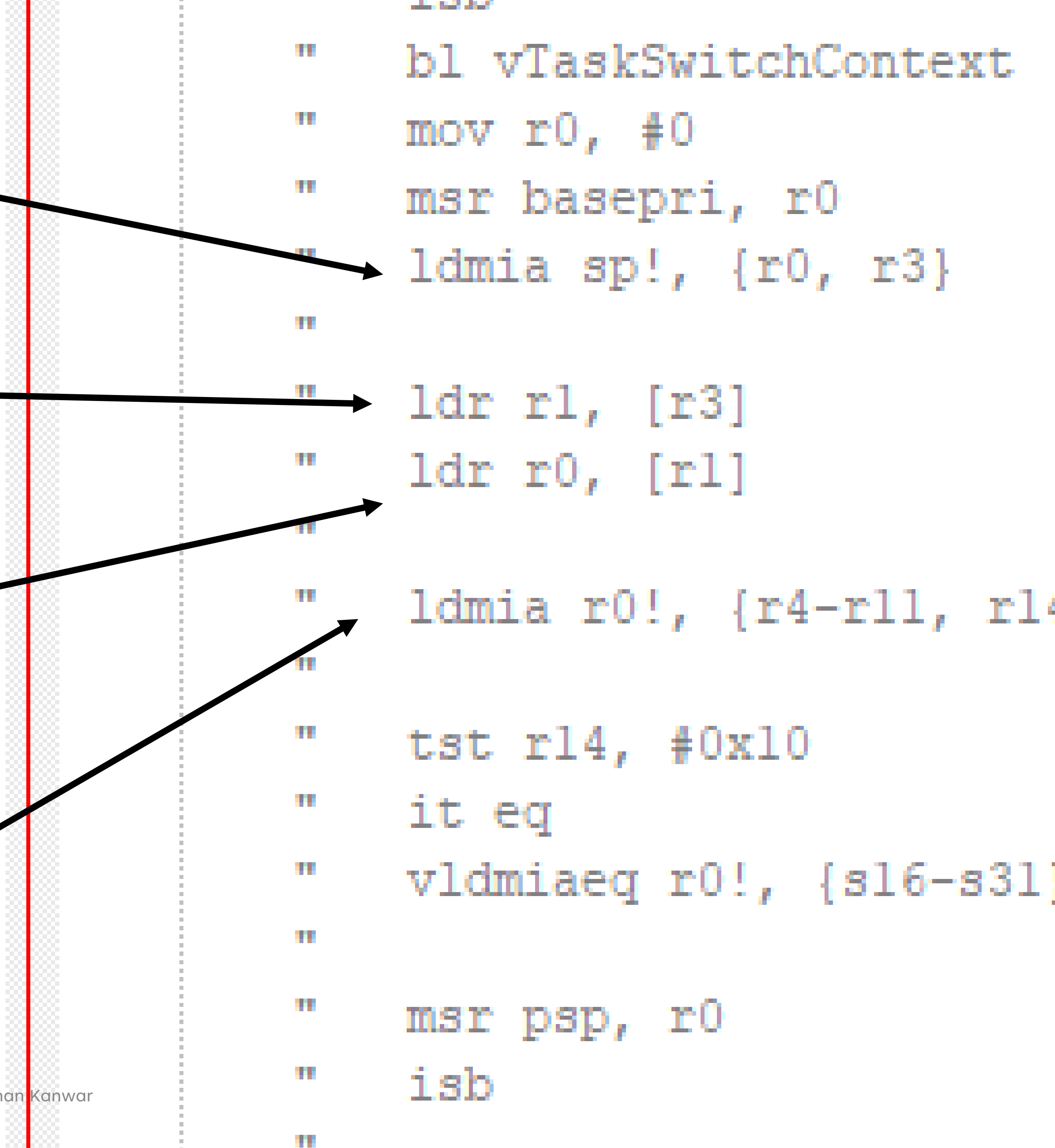
- R3 is having address of pxCurrentTCBConst.

pxCurrentTCBConst have address of TCB.

- Dereferencing R3, will give address of TCB, which is now stored in R1.

- Now as per TCB's structure, first element is "Top of Stack", another dereference will give address of "Top of Stack" now stored in R0

- Now with Stack address stored in R0, using same to unstack the registers of this new tasks R4 – R14 (Context)



```
"   bl vTaskSwitchContext
"   mov r0, #0
"   msr basepri, r0
"   ldmia sp!, {r0, r3}
"
"   ldr r1, [r3]
"   ldr r0, [r1]
"
"   ldmia r0!, {r4-r11, r1
"
"   tst r14, #0x10
"   it eq
"   vldmiaeq r0!, {s16-s31
"
"   msr psp, r0
"   isb
"
```

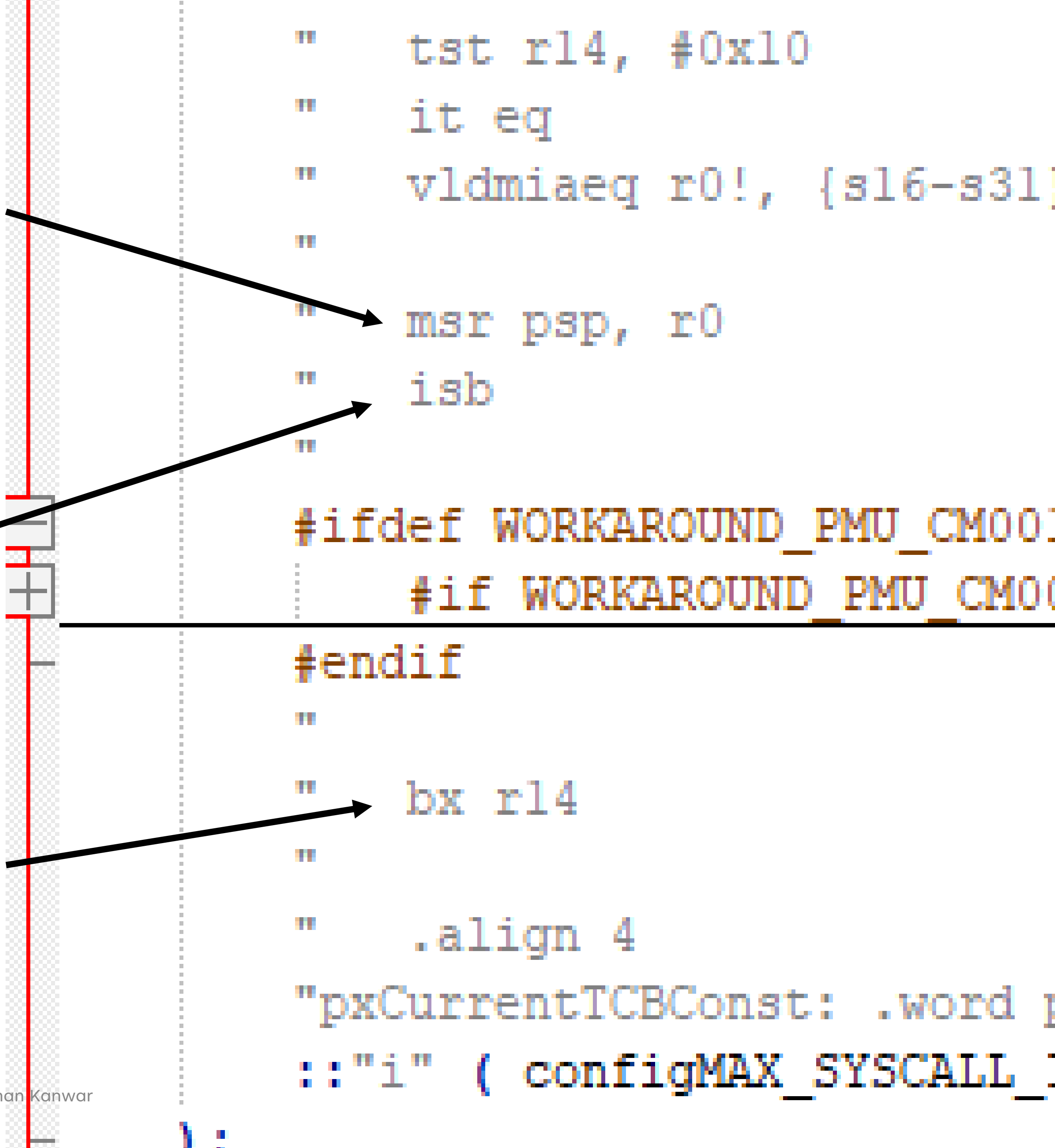
- Finally the updated stack pointer location which is currently pointed by R0 is loaded to Process Stack Pointer (PSP)

Above step is done so that taks's stack can work normally as PSP is the one which should manage stack on any stack operations while task is executing.

- For your information, this "isb" is Instruction Synchronization Barrier, it makes sure that the operation done before should be completed before this isb completes.

- This is the last point, now branching to Link Register (R14) is performed to start execution of new Task.

- Context Switch Completed ☺



```
"   tst r14, #0x10
"   it eq
"   vldmiaeq r0!, {s16-s31
"
"   msr psp, r0
"   isb
"
#ifdef WORKAROUND_PMU_CM00
    #if WORKAROUND_PMU_CM00
#endif
"
"   bx r14
"
"   .align 4
"pxCurrentTCBConst: .word p
::"i" ( configMAX_SYSCALL_
);
```

Piece of cake, wasn't it?

Who's willing to step forward and provide another explanation?

Quick test on what we just discussed

VIRTUAL ASSIGNMENT

Predict the behaviour of following application

PROBLEM 2

Task 1 (Priority 1)

-> Blink LED1, LED2 at every 100mS.

Task 2 (Priority 2)

-> Blink LED3, LED4 at every 500 mS.

What is the purpose of Task 1 then? When it will run?

Types of Tasks

- Periodic Task
- Aperiodic Task
- Idle Task
- Service Tasks

PERIODIC TASKS

Periodi task is
eriod etwe n

- eriod task
ccur ith a

Fi ed ecu

Ji

-
-

PERIODIC TASKS

Description

- Periodic task is a task that runs at regular intervals, typically with a fixed time period between each execution.
- Periodic tasks are often used in real-time systems to perform activities that need to occur with a specific frequency or rhythm.

Fixed Execution Time

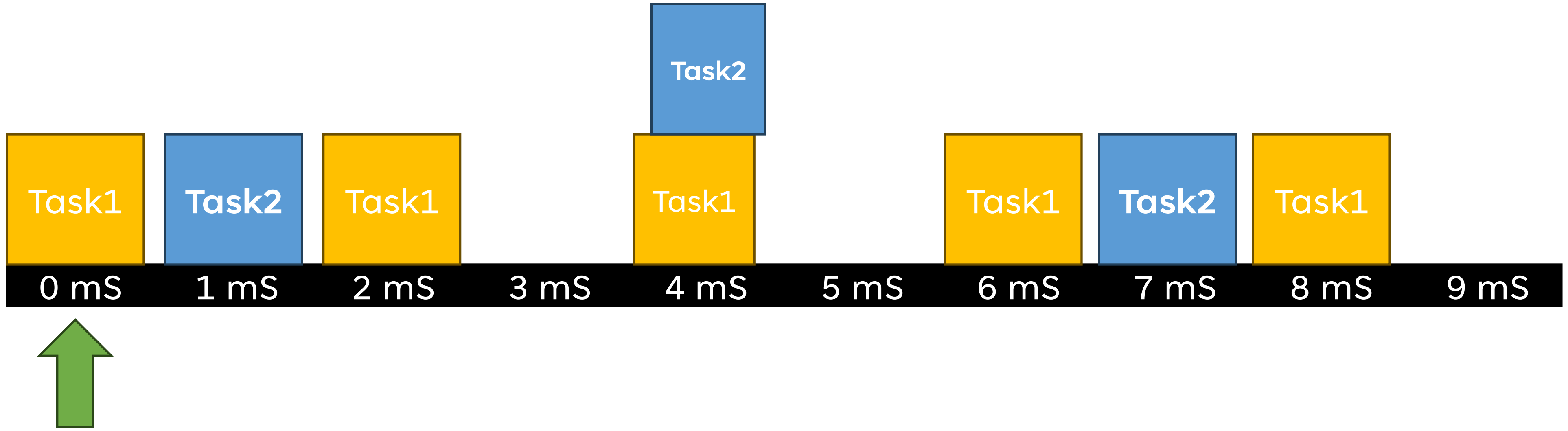
- The time taken by a periodic task to complete its execution should be consistent.
- Predictability ensures that the task meets its deadlines consistently.

Jitter

- Jitter refers to the variability in the start time or execution time of a periodic task.
- Low jitter is desirable for real-time systems, as it contributes to predictability.

Task1
Arrival: 0 mS
Period: 2 mS

Task2
Arrival: 1 mS
Period: 3 mS



PERIODIC TASKS

APERIODIC TASKS

Unlike
ap ric

Irre

- Ap
  ev nts

APERIODIC TASKS

Description

- Aperiodic tasks are tasks that do not follow a regular or fixed time schedule.
- Unlike periodic tasks, which have a predictable and repeating execution pattern, aperiodic tasks are triggered by events or stimuli and can occur at irregular intervals

Irregular Timing

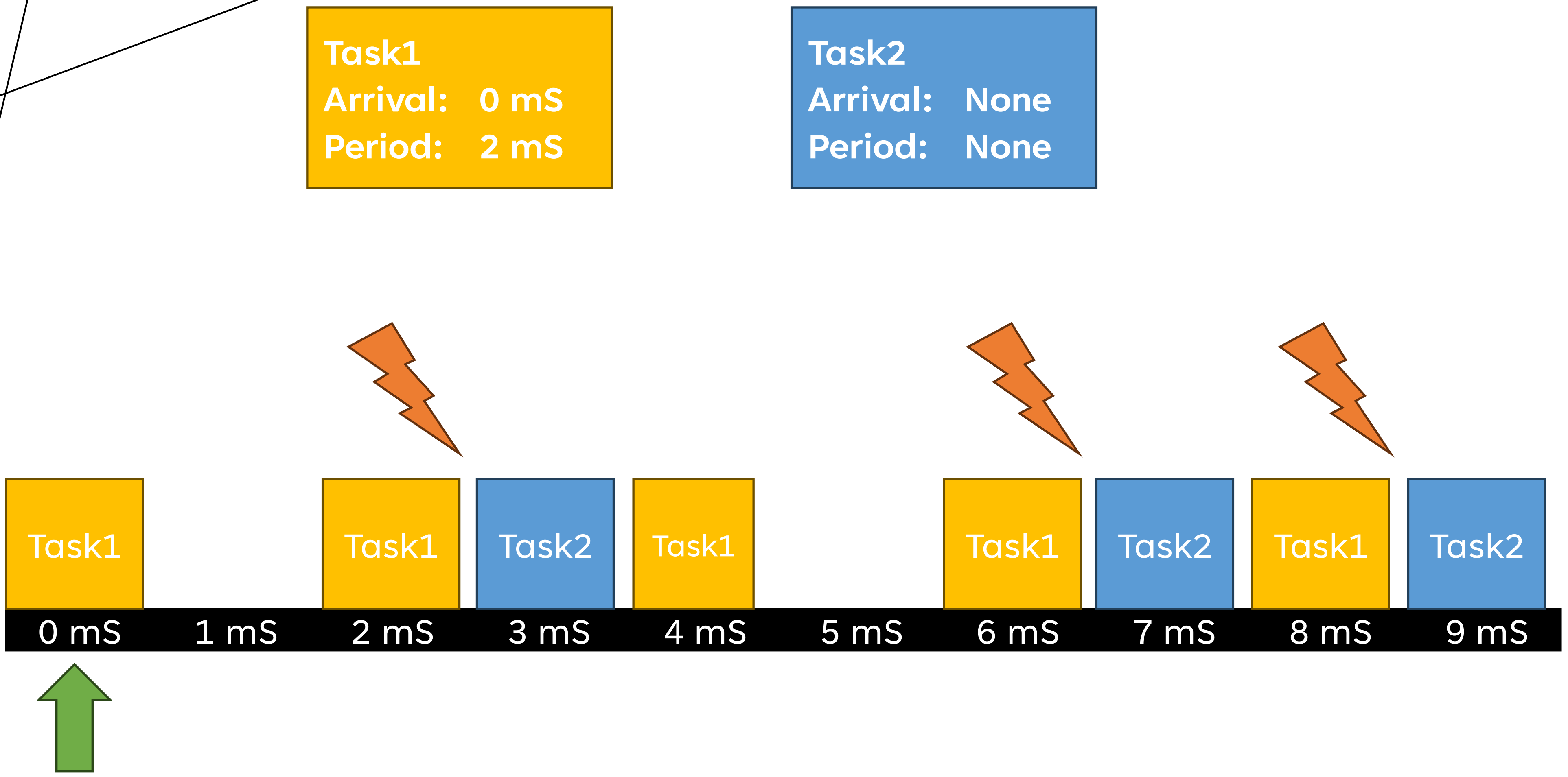
- Aperiodic tasks are often used in real-time systems to handle asynchronous events, interrupts, or other unpredictable occurrences
- Aperiodic tasks are typically associated with specific events, interrupts, or external triggers that prompt their execution.

Dynamic Priority

- Aperiodic tasks may have dynamic priorities, allowing the system to adjust their execution priority based on the urgency of the events they handle.

Critical Event Handling and Variable Execution Time

- Aperiodic tasks may have dynamic priorities, allowing the system to adjust their execution priority based on the urgency of the events they handle.
- The time it takes for an aperiodic task to complete its execution can vary based on the nature of the event it is handling.



APERIODIC TASKS

LOGICAL QUESTION
(STUDENT SPECIFIC)

Come up with 3 examples of a periodic task scenario

1

2

3

Periodic Tasks

- Are we creating Periodic Tasks as of now?
- Remember the definition of Periodic Task.
- Is using Hal_Delay() efficient?
- How Hal_Delay() works?
- Let’s look at the CPU Utilization with HAL_Delay() usage.
- Better Solution?

Hal_Delay()

Implementation

- Normally implemented on top of **SysTick timer** interrupts.
- **SysTick** increments **ucTick** global parameter.
- **ucTick** represents system tick count.

Working

- Compares ucTick (Micro Controller's Tick).
- HAL implements **ucTick** using timer that increments at regular intervals.
- System tick count is usually incremented in an interrupt service routine (ISR) and is used for timekeeping in the software.
- Resolution of Tick Increment depends on configured frequency at which ISR is called.

Delay

- Once call to Hal_Delay(x) is made from a function, Hal_Delay() stores current **ucTick**.
- Hal_Delay(x) compares **ucTick** with input parameter value provided in milliseconds, comparison is done inside a while loop (**Important**).
- Once the delay value matches with **ucTick** value, Hal_Delay(x) returns back to function.
- **In short Hal_Delay() is just a loop which is continuously in execution (IMPORTANT)**

Hal_Delay()

Is this a Good solution for Task Delays.

Has worst affect on CPU Loads.

Idle Task is unable to run due to heavy CPU load.

Idle Task is responsible for freeing the unused kernel objects.

More on Idle Tasks later-on.

Idle task can put CPU in power saving mode and save power.

Hal_Delay()

Tick Rate: 1 mS (1000uS)

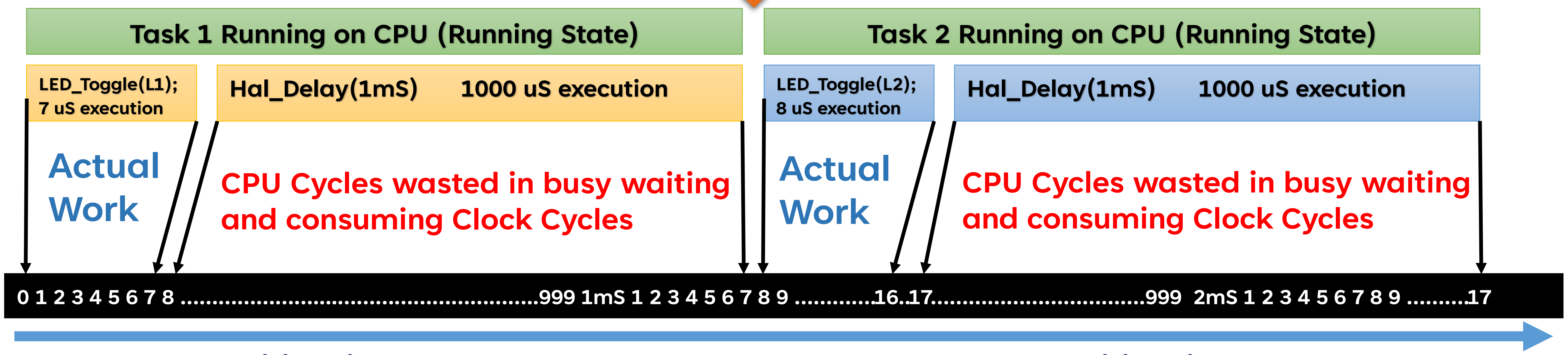
```
Task1 ()
{
 loop()
 {
    LED_Toggle(L1);
    Hal_Delay(1);
 }
}
```

Actual Work?

Task1% = (1000*100)/1000
Task2% = (1000*100)/1000
CPU Load% = 100% per Tick

```
Task2 ()
{
 loop()
 {
    LED_Toggle(L2);
    Hal_Delay(1);
 }
}
```

Context Switch



Task 1 Running on CPU (Running State)
Task 2 Running on CPU (Running State)
LED_Toggle(L1); 7 uS execution
Hal_Delay(1mS) 1000 uS execution
LED_Toggle(L2); 8 uS execution
Hal_Delay(1mS) 1000 uS execution
Actual Work
CPU Cycles wasted in busy waiting and consuming Clock Cycles
Actual Work
CPU Cycles wasted in busy waiting and consuming Clock Cycles
0 1 2 3 4 5 6 7 8 ....999 1mS 1 2 3 4 5 6 7 8 9 ....16..17.................................999 2mS 1 2 3 4 5 6 7 8 9 17

Task1 CPU utilization: 1000 uS

Task2 CPU utilization: 1000 uS

vTaskDelay()

Description

- vTaskDelay is a function in the FreeRTOS (Real-Time Operating System) task API that allows a FreeRTOS task to introduce a time delay in its execution.
- This function is used to temporarily suspend the execution of the calling task, allowing other tasks to run during the delay period.

Functionality

- Introduces a delay in the execution of the calling task
- Puts the task into the Blocked state during the delay period

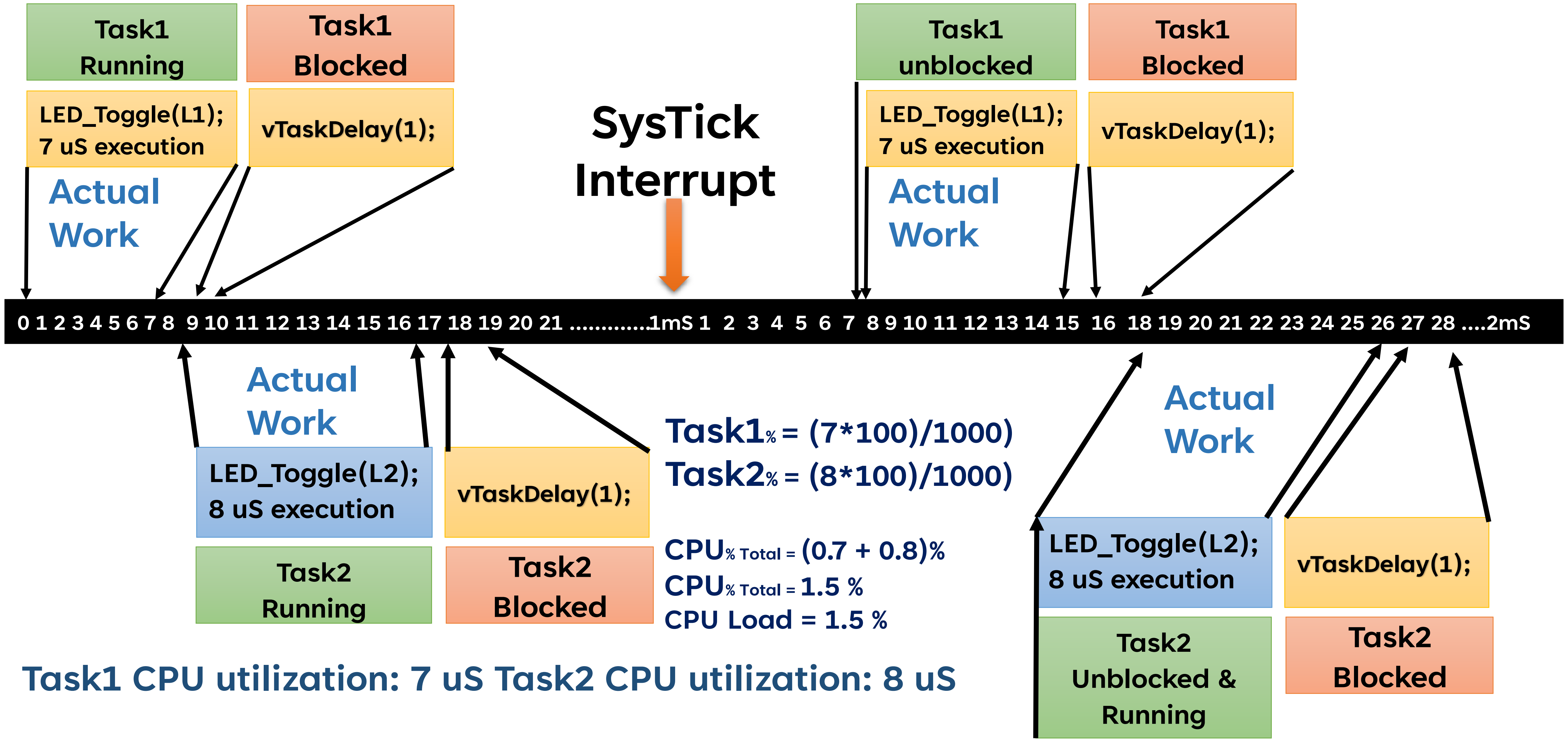
Usage

- Used within FreeRTOS tasks to create time delays.
- Allows other tasks to run during the delay, promoting multitasking.

Periodicicty

- Implemented in Periodic Tasks, also used to intentionally block a task’s execution.
- Used for controlled functionality execution in case of Multiple Tasks with same priority.

vTaskDelay() Tick Rate: 1 mS (1000uS)


Task1 Running
LED_Toggle(L1); 7 uS execution
Task1 Blocked
vTaskDelay(1);
Actual Work
SysTick Interrupt
Task1 unblocked
LED_Toggle(L1); 7 uS execution
Task1 Blocked
vTaskDelay(1);
Actual Work
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 ....1mS 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 18 19 20 21 22 23 24 25 26 27 28 ....2mS
Actual Work
LED_Toggle(L2); 8 uS execution
vTaskDelay(1);
Task2 Running
Task2 Blocked
Actual Work
LED_Toggle(L2); 8 uS execution
vTaskDelay(1);
Task2 Unblocked & Running
Task2 Blocked

$Task1_{\%} = (7*100)/1000$

$Task2_{\%} = (8*100)/1000$

$CPU_{\% \text{ Total}} = (0.7 + 0.8)\%$

$CPU_{\% \text{ Total}} = 1.5\ \%$

CPU Load = 1.5 %

Task1 CPU utilization: 7 uS Task2 CPU utilization: 8 uS

ASSIGNMENT

Write a FreeRTOS Application with following requirements.

Implement Periodic Tasks as follows:

Task 1 (Priority 1)

Toggle LED 1, LED 2 at every 500 mS

Task 2 (Priority 2)

Toggle LED 3, LED 4 at every 1000 mS

Observe and document the behavior, how the outcome is different than using Hal_Delay();

TRY USING TRACING TOOL

Write a FreeRTOS Application with following requirements.

Implement Periodic Tasks as follows:

Task 1 (Priority 1)

Toggle LED 1, LED 2 at every 5 mS

Task 2 (Priority 2)

Toggle LED 3, LED 4 at every 10mS

Observe and document the behavior on document along with Trace Diagram

ASSIGNMENT

Write a FreeRTOS Application with following requirements.

Implement Periodic Tasks as follows:

Create Tasks in Following order: **Task2, Task4, Task 1, Task3**

Toggle LED1, 2, 3, 4 in Task1, 2, 3, 4 respectively without any delay in task's functionality code.

Make sure the Tasks getting executed in following sequence on Trace Tool Graph.

Task1 → Task2 → Task3 → Task 4 → Task1 → Task2 → Task3 → Task4 ......

Aperiodic Task

- Remains inactive/suspended.
- Activated by some event/interrupt.
- Goes back to inactive/suspended state after execution until next event
- Used for urgent functionalities.
- These tasks are created with higher priorities.

ASSIGNMENT

Write an application with following requirements.

Use user-button input

- **-> Task 1 (Periodic) Priority 1**
  - **-> Toggle LED1, LED2 at 100 mS**
- **-> Task 2 (Periodic) Priority 1**
  - **-> Read button, if pressed resume Task3.**
- **-> Task 3 Priority 3 (suspended state)**
  - **-> Toggle LED3, LED4 state**
  - **-> Suspend itself**

Document the behaviour.

IDLE TASKS

Des

Id
o

W

es

eation

The id
schedule

e creatio
duler.

IDLE TASKS

Description

- Special task that runs when no other higher-priority tasks are ready to execute. The idle task serves to consume CPU cycles in a power-efficient manner when there is no other work to be done.
- When higher-priority tasks are ready, the idle task yields control to allow those tasks to execute.

Responsibility

- The primary responsibility of the idle task is to execute when the system is not actively processing any other tasks.
- The idle task performs background activities, such as managing power-saving features or executing low-priority system tasks.

Creation

- The idle task is automatically created during the initialization of the FreeRTOS scheduler.
- The creation and management of the idle task are handled internally by the FreeRTOS scheduler.

Priority

The idle task has the lowest priority in the system, ensuring that it gives up control whenever a task of higher priority becomes ready to run.

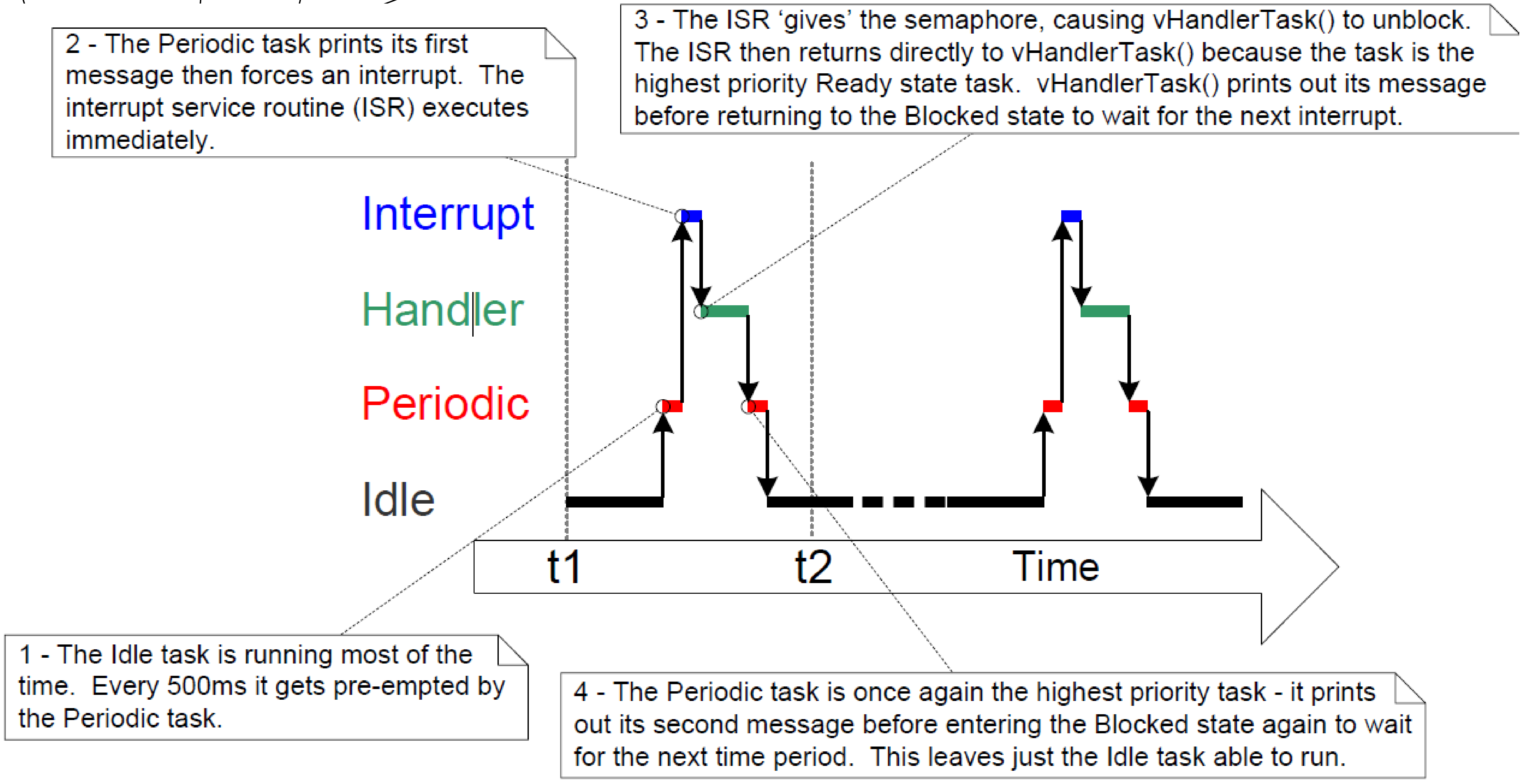
MISC

Idle task running is good for FreeRTOS heap as it serves the purpose of claiming the freed resources. If application is calling Task delete functionality anywhere in program then Idle Task serves the purpose of freeing the memory taken up by freed object.

Points to consider

- Idle tasks also have a callback registered called as application hook.
- Idle Tasks can save CPU Cycles by putting the core into sleep state.
- Tickless kernel approach can also help to save CPU Power consumption.
- Normally Idle Tasks have a priority value 0
- **Refer to File tasks.c line 1989**
- Read section **3.8 “The Idle Task and the Idle Task Hook”**

IDLE TASKS



2 - The Periodic task prints its first message then forces an interrupt. The interrupt service routine (ISR) executes immediately.

3 - The ISR 'gives' the semaphore, causing vHandlerTask() to unblock. The ISR then returns directly to vHandlerTask() because the task is the highest priority Ready state task. vHandlerTask() prints out its message before returning to the Blocked state to wait for the next interrupt.

1 - The Idle task is running most of the time. Every 500ms it gets pre-empted by the Periodic task.

4 - The Periodic task is once again the highest priority task - it prints out its second message before entering the Blocked state again to wait for the next time period. This leaves just the Idle task able to run.

IDLE TASK HOOK

Description

Adding an application specific functionality during the idle task execution is done via Idle Task Hook function, as per the name “Hook” it is hooked every time idle task runs.

Runtime Behaviour

Once enabled, user functionality runs in context of Idle Task and have the run-time properties like priorities, stack etc as per Idle task. Task hook function has a specific prototype and must be crated with same function name and signature.

Points to consider

- Keep the code within vApplicationIdleHook as non-blocking as possible.
- Blocking operations within idle hook can impact the responsiveness of the system.
- Avoid lengthy operations within the idle hook.
- Avoid creating loops within hook that continuously consume CPU cycles.
- **Refer to File tasks.c line 1989**
- Read section **3.8 “The Idle Task and the Idle Task Hook”**

ASSIGNMENT

Write an application with following requirements.

***Use Hal_Delay only for this assignment**

Create Task 1 (Priority 2)

Blink LED 1, 2 at every 500 mS

Create Task 2 (Priority 2)

Blink LED 3, 4 at every 100 mS

Enable and Implement **Application Idle Hook function** and increment a global variable inside it.

After around 10 seconds of program execution, check the current value of that global counter.

Document your conclusions and be ready to given an explaination as per your understanding!

ASSIGNMENT

Write an application with following requirements.

***Use vTaskDelay only for this assignment**

Create Task 1 (Priority 2)

    Blink LED 1, 2 at every 500 mS

Create Task 2 (Priority 2)

    Blink LED 3, 4 at every 100 mS

Enable and Implement **Application Idle Hook function** and increment a global variable inside it.

After around 10 seconds of program execution, check the current value of that global counter.

Document your conclusions and be ready to given an explaination as per your understanding!

Power saving with Idle Tasks?

- Idle Tasks can reduce the power consumption by the core?
- How to verify the same?

Interrupt Management

- How interrupts are handled in RTOS environment.
- Interrupt safe FreeRTOS APIs.
- Rules and tips for interrupt handlers.

ASSIGNMENT

Write an application with following requirements.

Enable External interrupt (Tactile Switch)

- **-> Task 1 (Periodic) Priority 1**
    - **-> Toggle LED1 at 100 mS**
- **-> Task 2 (Periodic) Priority 1**
    - **-> Toggle LED2 at 200 mS**
- **-> External Button Interrupt**
    - **-> Toggle LED3, LED4 state**

INTERRUPTS

Information

- In ARM Cortex M4, lower interrupt priority number means Highest Priority.
- In FreeRTOS tasks with lower interrupt priority number means Lower Priority.

Working

- Many FreeRTOS services are blocking in nature, one should not use blocking APIs for interrupts.
- ISR safe API calls should me used ending “**from_ISR**” in interrupt handlers.
- Interrupt handlers are not tasks and context can’t be scheduled.
- Handlers should be as short as possible to reduce interrupt latencies.

MISC

- RTOS Kernel, scheduler is also running with a specific interrupt priority number.
- Even a Lowest priority H/W interrupt can preempt a Highest Priority running Task.
- Interrupt implementation should be done based on a given RTOS Port and documentation.
- Hardware interrupts must be implemented keeping the priorities as per RTOS port.
- Read **Chapter 6 Interrupt Management** from Mastering the FreeRTOS™ Real Time Kernel.

ASSIGNMENT

Write an application with following requirements.

Enable External interrupt (Tactile Switch)

- **-> Task 1 (Periodic) Priority 1**
  - **-> Initially suspend itself**
  - **-> Toggle LED1 at 100 mS (Inside Functionality)**
- **-> Task 2 (Periodic) Priority 1**
  - **-> Toggle LED2 at 200 mS (Inside Functionality)**
- **-> External Button Interrupt**
  - **-> Resume Task1 once switched is Pressed.**

ASSIGNMENT

Write an application with following requirements.

Enable External interrupt (Tactile Switch)

- -> Task 1 (Periodic) Priority 1
  - -> Toggle LED1 at 100 mS
- -> Task 2 (Periodic) Priority 1
  - -> Toggle LED2 at 200 Priority 1 mS
- -> Task 3 (Aperiodic) Priority 3 (Suspended state)
  - -> Toggle LED3, LED4
  - -> Suspends itself
- -> External Button Interrupt
  - -> Resume Task3

ASSIGNMENT

Write an application with following requirements

Task1 Priority 1

- **Suspended by default**
- **Toggle LED 1, 2, 3, 4 at every 10 mS (Functionality)**

Task2 Priority 2

- **Suspended by default**
- **Resume Task1 once and suspend itself**

Task3 Priority 3

- **Suspended by default**
- **Resume Task2 once and suspend itself**

Task4 Priority 4

- **Suspended by default**
- **Resume Task3 once and suspend itself**

Use ExternalInterrupt0 (Button) and resume Task 4

Observe the trace diagram for the same and document.

ASSIGNMENT

Write an application with following requirements

Task1 Priority 1 (Functionality)

- **Toggle LED 1**
- **Suspend itself**
- **Resume Task 4**

Task2 Priority 2 (Functionality)

- **Toggle LED 2**
- **Suspend itself**

Task3 Priority 3 (Functionality)

- **Toggle LED 3**
- **Suspend itself**

Task4 Priority 4 (Functionality)

- **Toggle LED 4**
- **Suspend itself**

Use ExternalInterrupt0 (Button) and resume Task 1

Observe the trace diagram for the same and document.

ASSIGNMENT (SYNC)

Write an RTOS Application, with following requirements.

Make use of ITM_SendChar(); with following code, and printf call. Print following message without any delay

Task1: “A Linux process is an independent, executing instance of a program”

Task2: “A Linux thread is a lightweight, independently schedulable execution unit within a process”

Observe the behaviour on Trace Data Window and document the same.

AUDIENCE CALL

Any issues with the approach?

What is the actual problem?

Possible solutions?

ASSIGNMENT

Write a FreeRTOS Application with following requirements.

Implement Periodic Tasks as follows:

Create Tasks in Following order: **Task2, Task4, Task 1, Task3**

Toggle LED1, 2, 3, 4 in Task1, 2, 3, 4 respectively without any delay in task’s functionality code.

Make sure the Tasks getting executed in following sequence on Trace Tool Graph.

Task1 → Task2 → Task3 → Task 4 → Task1 → Task2 → Task3 → Task4 ……

TRY USING TRACING TOOL

PROBLEM 1

Observations?

Snapshot of graph

Mention your understanding

PROBLEM 2

Obvervations?

Snapshot of graph

Mention your understanding

PROBLEM 3

Obvervations?

Snapshot of graph

Mention your understanding

ASSIGNMENT QUESTION

Write an RTOS application with two task.

Task1: Use this task to

-> Suspend the Task2 using Task2 handle.

-> Set LED1 high for 10 Seconds

-> Resume Task2 and Delete Task1

Task2: Toggle the LED2 at every 500 mS.

Implementation1: Use global task handles

Implementation2: Use private task handles (passing by parameters).

ASSIGNMENT QUESTION

Write an RTOS application with two task.

Create Task1 Priority 2 and Task2 Priority 1

Task1: Use this task to

-> SET LED1 High for 1 Second.

-> Set the Priority of Task2 as Task1's Priority + 1.

Task2: Use this task to

-> SET LED1 Low for 1 Second.

-> Set the Priority of Task2 as Task1's Priority - 1.

Use private task handles (passing by parameters), observe the behavior and document.

SUMMARY

FreeRTOS has it’s own implementation of scheduler and the low level APIs. Further the periodicity is handled by special blocking calls. Idle task help further in implementing power saving modes and optimize CPU usage.

THANK YOU

Aman Kanwar

QUIZ

Aman Kanwar