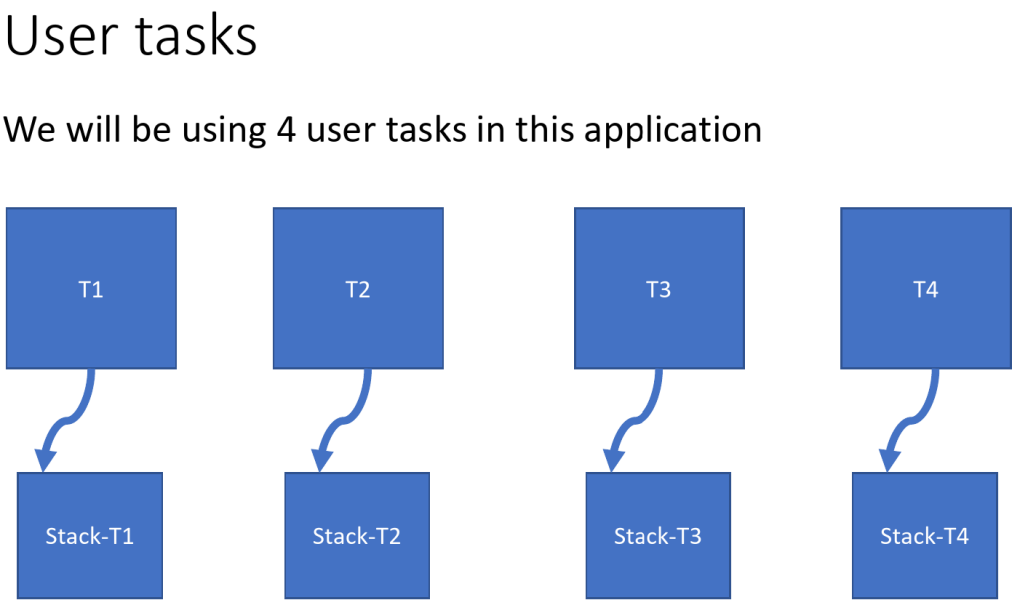
Implementing a scheduler

Implementing a scheduler :

- Let's implement a scheduler which schedules multiple user tasks in a round-robin fashion by carrying out the context switch operation
- Round robin scheduling method is , time slices are assigned to each task in equal portions and in circular order
- First will use systick handler to carry out the context switch operation between multiple tasks
- Later will we change the code using pendSV handler

What is a task ?

- A task is nothing but a piece of code, or you can call it a ('C') function, which does a specific job when it is allowed to run on the CPU.
- A task has its own stack to create its local variables when it runs on the CPU. Also when scheduler decides to remove a task from CPU, scheduler first saves the context(state) of the task in task's private stack
- So, in summary, a piece code or a function is called a task when it is schedulable and never loses its 'state' unless it is deleted permanently.



Current task :

Create 4 user tasks (basically never returning C functions)

```c
#include <stdint.h>
#include <stdio.h>

void task1_handler(void);  // This is task-1
void task2_handler(void);  // This is task-2
void task3_handler(void);  // This is task-3
void task4_handler(void);  // This is task-4 of the application

int main(void)
{
    /* Loop forever */
    for(;;);
}

void task1_handler(void)
{
    while(1)
    {
        printf("This is task-1\n");
    }
}

void task2_handler(void)
{
    while(1)
    {
        printf("This is task-2\n");
    }
}

void task3_handler(void)
{
    while(1)
    {
        printf("This is task-3\n");
    }
}

void task4_handler(void)
{
    while(1)
    {
        printf("This is task-4\n");
    }
}
```

Stack pointer selection :

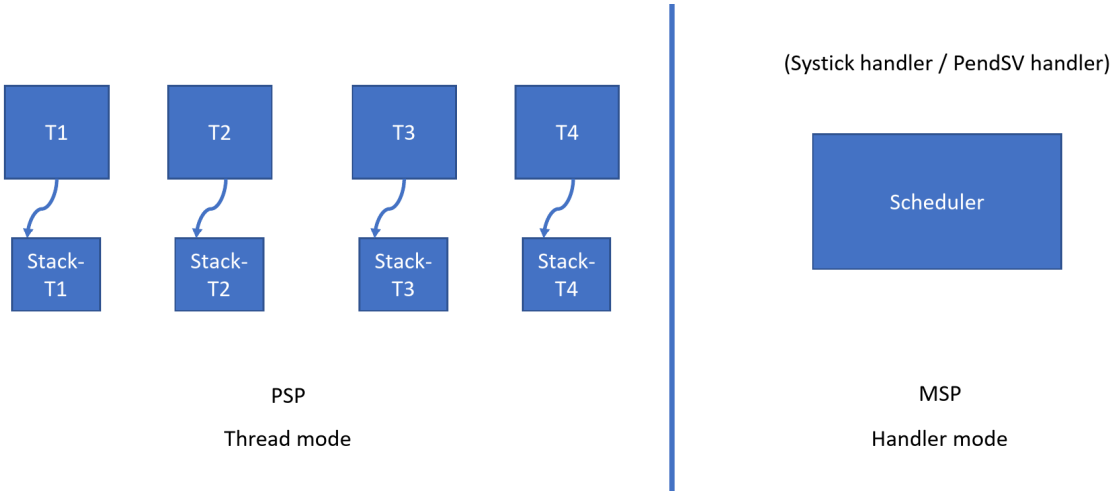
We have two part of the application :

1. Running 4 user tasks

2. Running scheduler

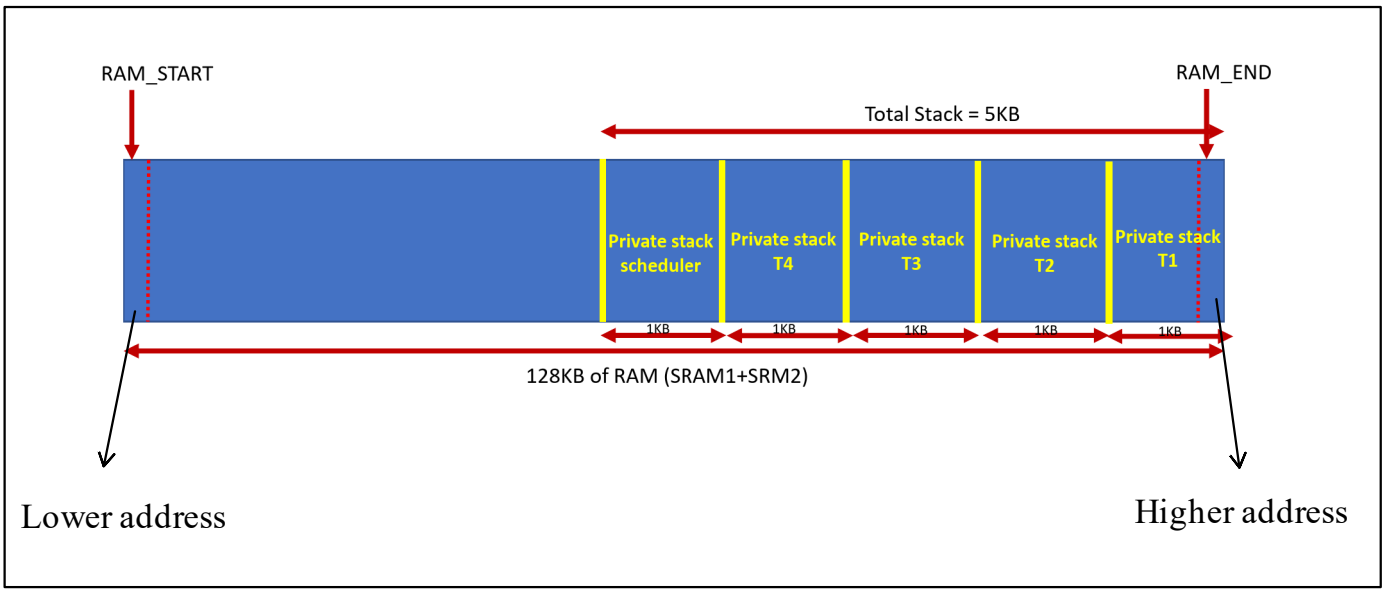
User tasks use PSP as a stack pointer

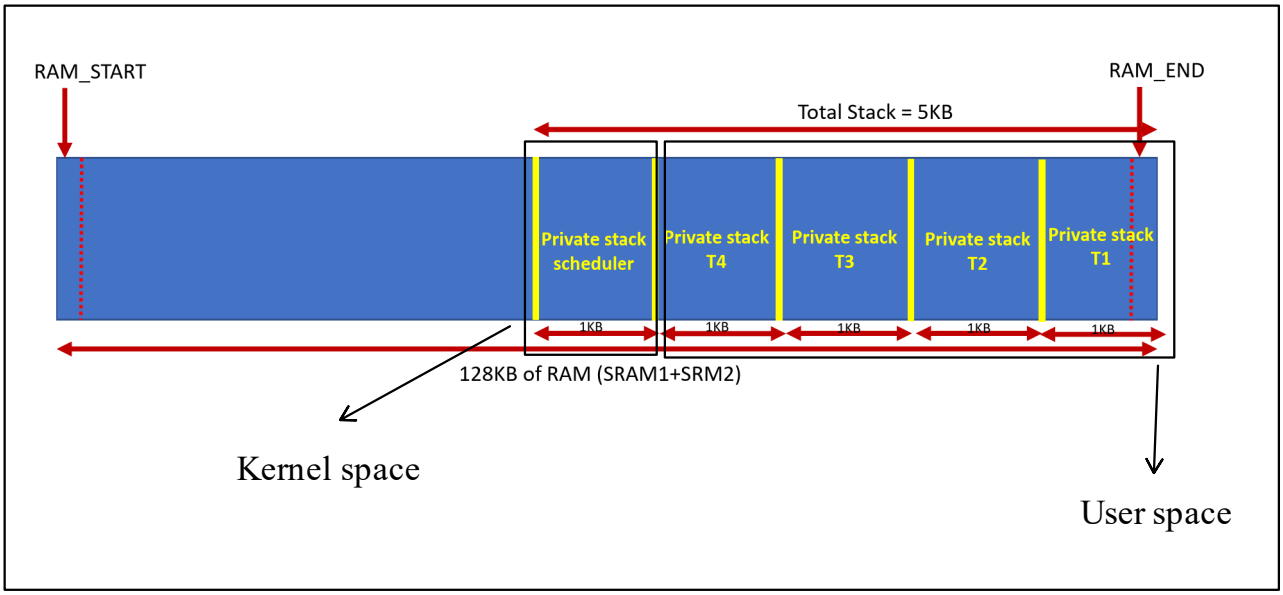
Scheduler use MSP as a stack pointer



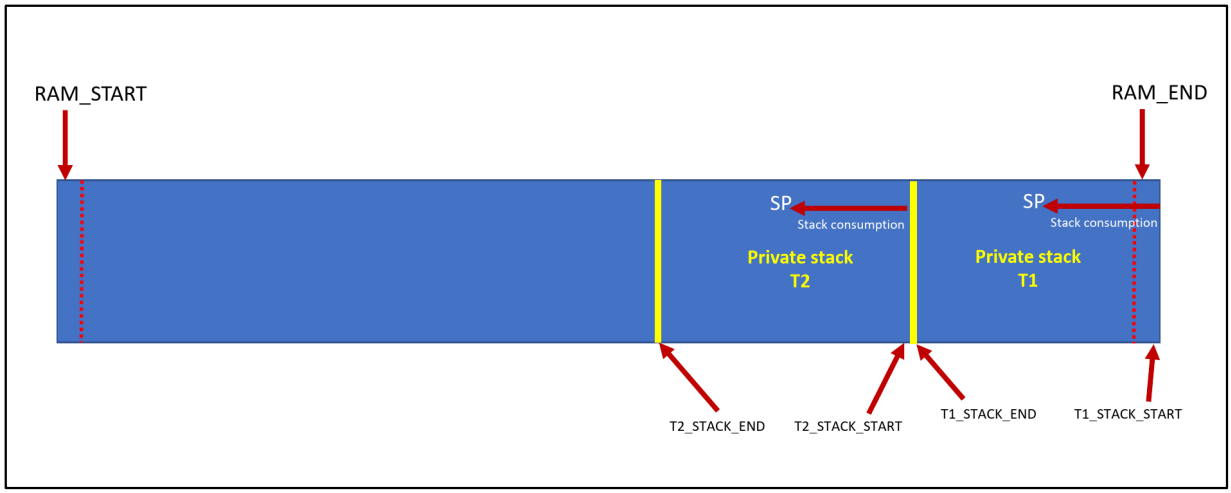
Stack assessment :

Before accessing the stack memory, we need to define proper boundaries for user tasks and scheduler





Our controller has full descending stack model



Current task :

Reserve stack areas for all the tasks and scheduler

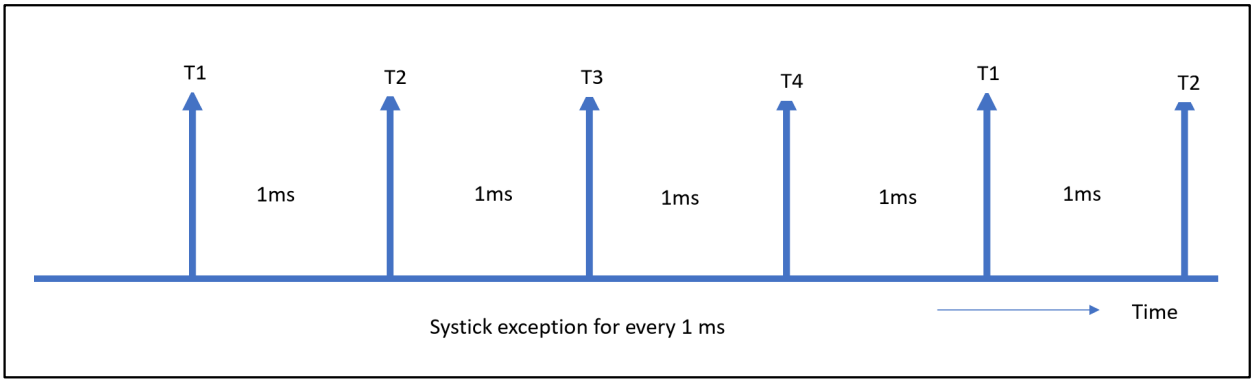
```
/*Stack memory calculations*/

#define SIZE_TASK_STACK            1024U    // 1Kb
#define SIZE_SCHED_STACK           1024U    // 1kb

#define SRAM_START                 0x20000000U
#define SIZE_SRAM                  ( (128) * (1024) )  // 128kb
#define SRAM_END                   ( (SRAM_START) + (SIZE_SRAM) )

#define T1_STACK_START             SRAM_END
#define T2_STACK_START             ( (SRAM_END) - (1 * SIZE_TASK_STACK) )
#define T3_STACK_START             ( (SRAM_END) - (2 * SIZE_TASK_STACK) )
#define T4_STACK_START             ( (SRAM_END) - (3 * SIZE_TASK_STACK) )
#define SCHED_STACK_START          ( (SRAM_END) - (4 * SIZE_TASK_STACK) )
```

Scheduling policy selection :

- We will be using round robin pre-emptive scheduling
- No task priority
- We will use SysTick timer to generate exception for every 1ms to run the scheduler code

Note:

Preemptive means switching out of a running task switching in of the new task or the next task which is in the ready state



What is scheduling ?

- Scheduling is an algorithm which takes the decision of pre-empting a running task from the CPU and takes the decision about which task should run on the CPU next
- The decision could be based on many factors such as system load, the priority of tasks, shared resource access, or a simple round-robin method

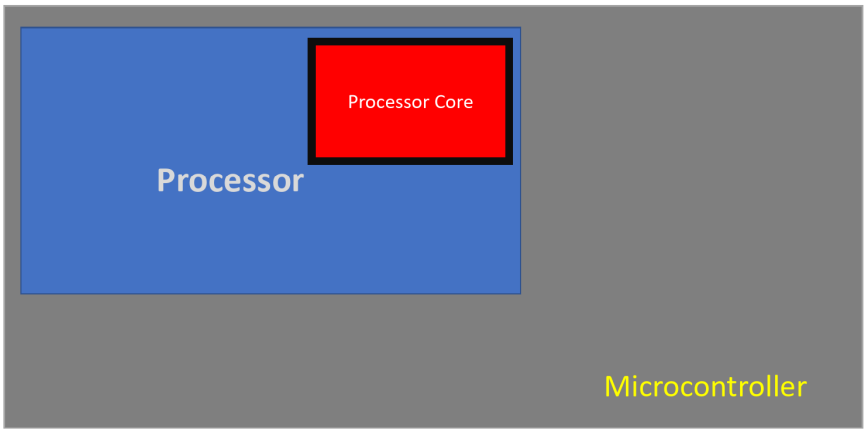
Shared resource access :

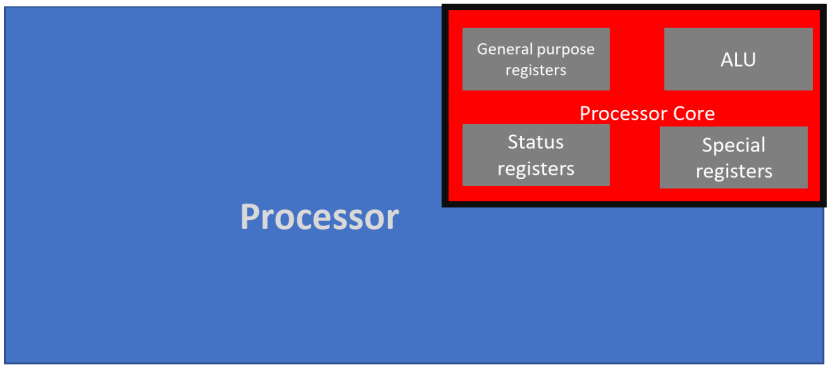
If a task which is getting switched out is holding any key to access the shared resource then the scheduler may take some dynamic decision

What is context switching ?

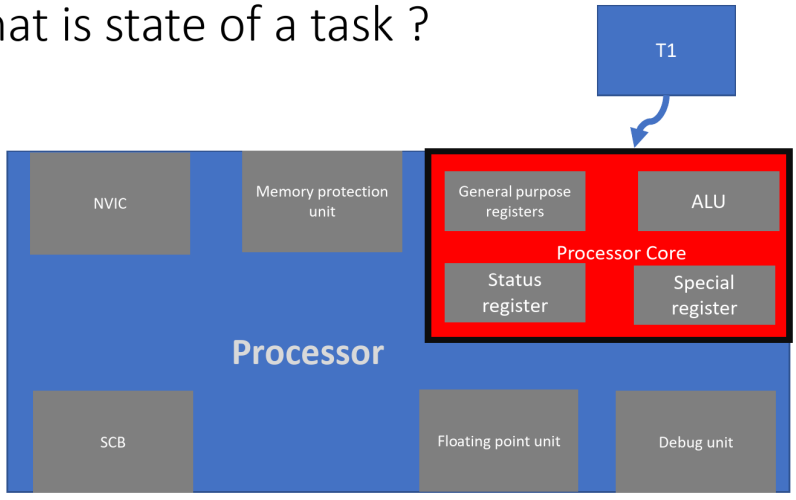
- Context switching is the procedure of switching out of the currently running task from the CPU after saving the task's execution context or state and switching in the next task's to run on the CPU by retrieving the past execution context or state of the task.

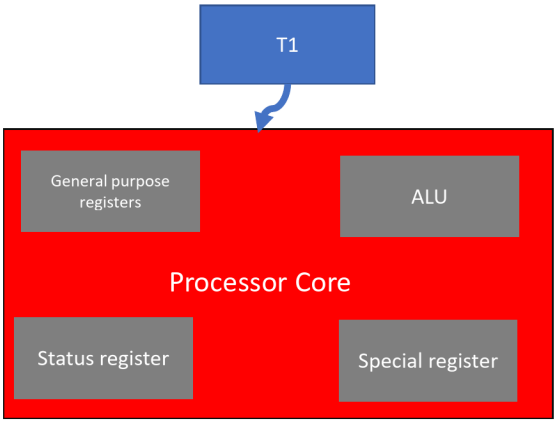
What is execution context or state of a task ?





What is state of a task ?

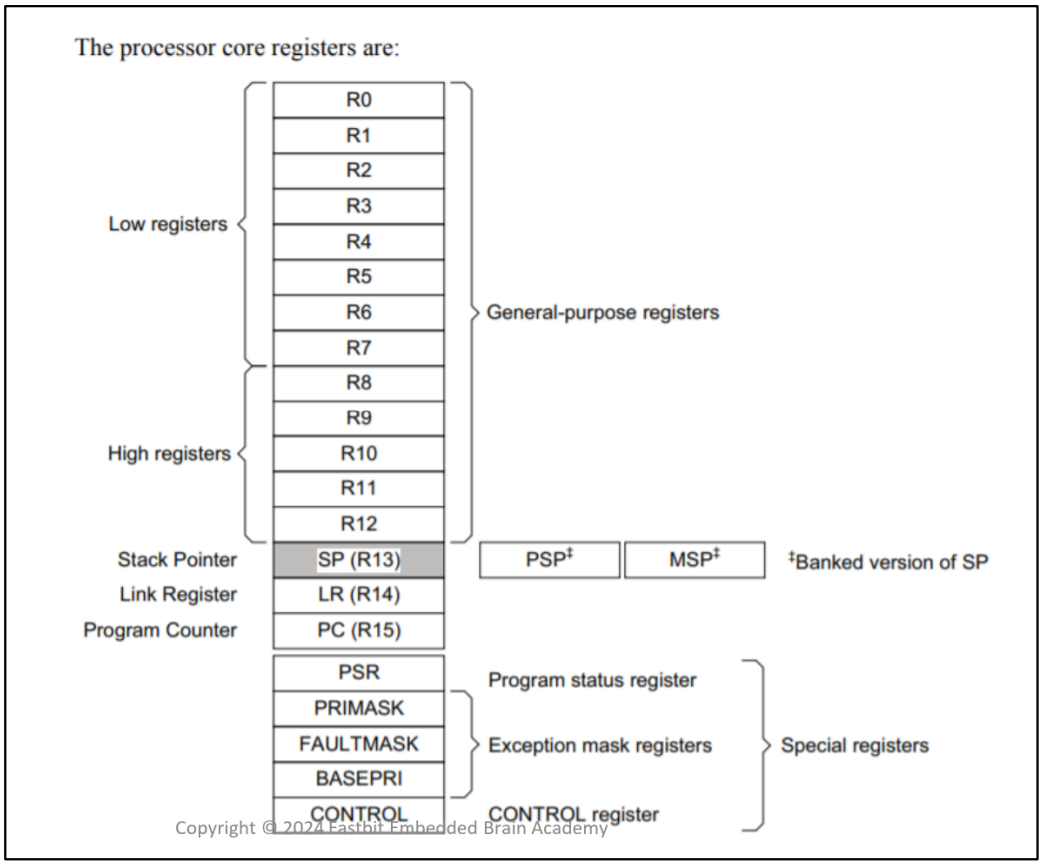




When the scheduler wants to switch out a task, then it needs to save the intermediate results / state of a task included in :

General purpose registers
+
Some special registers
+
Status register

Core registers :



PC :

When the scheduler was pre-empting or switching out a task, the PC will hold address of next instruction of task handler,

when scheduler want to switch in that task some times later, it should know where it should return

LR :

Reveal info about task_handler, stack pointer selection

PSP :

Current (state) value of task private stack is stored in PSP

PSR :

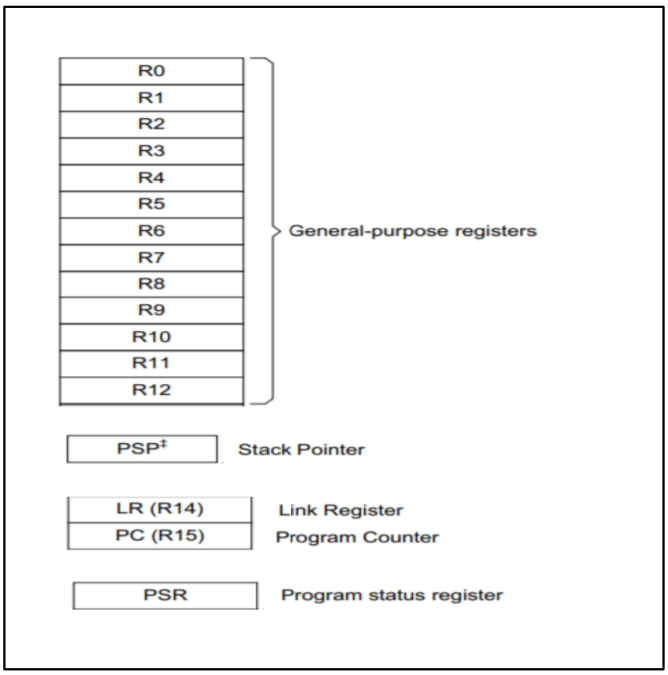
Snapshot of current state, zero flag set or not, negative flag

Summary for state of a task :

When scheduler decides to switch out a task, it should preserve these registers in task private stack.

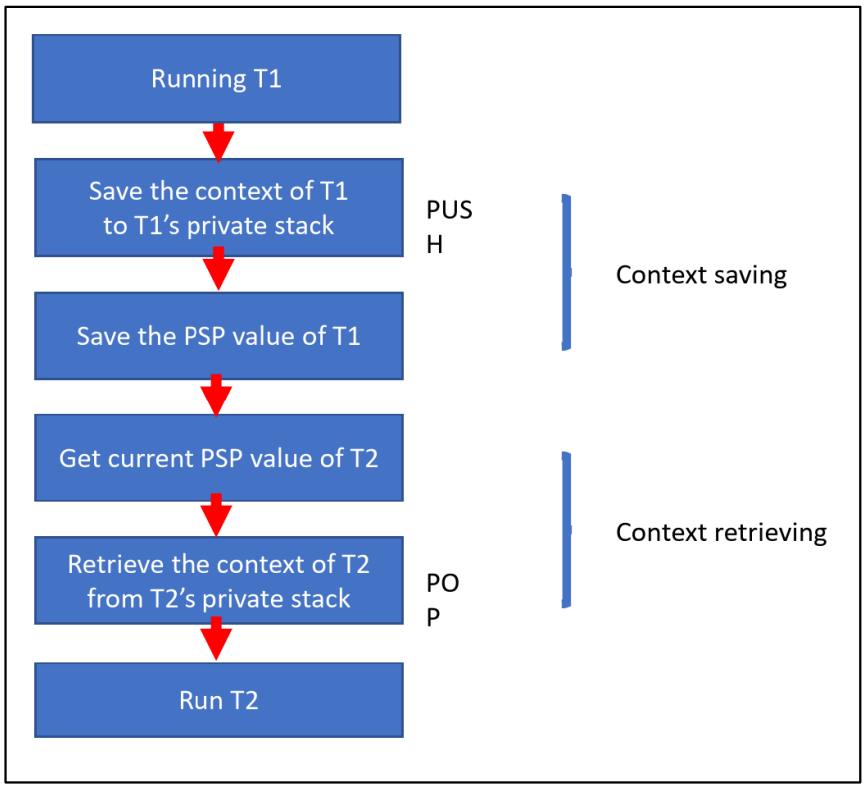
Because these registers collectively represent current execution state of the task.

These state must be retrieved back again when scheduler decides to run this task again in later time.

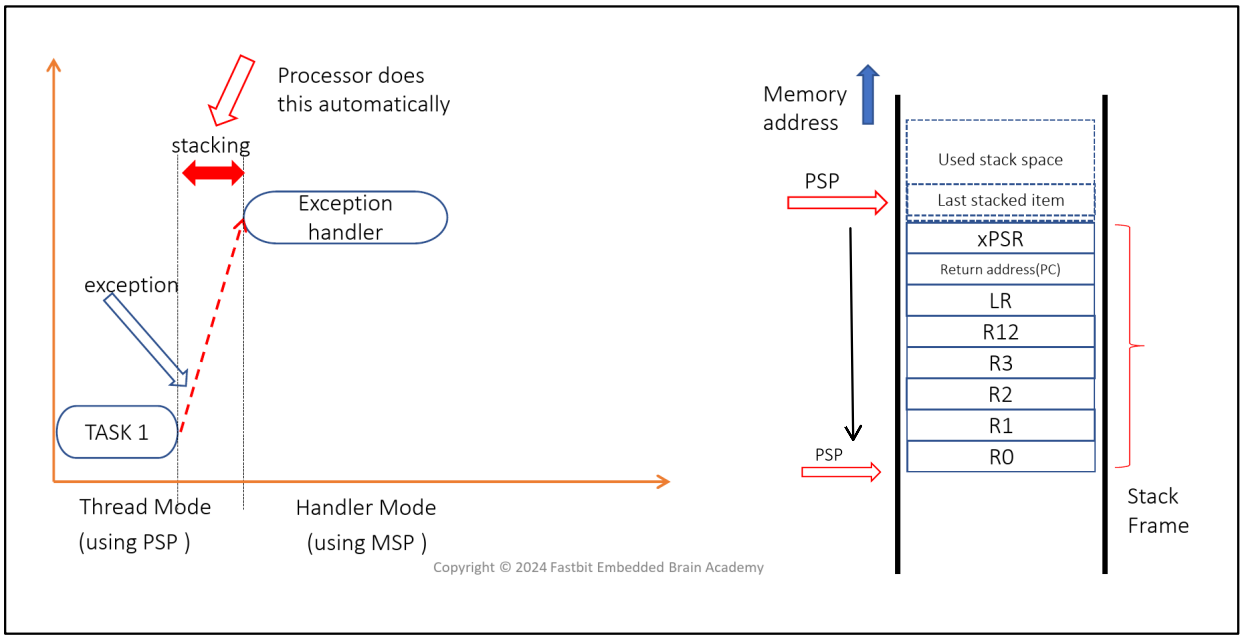


Case study of context switching :

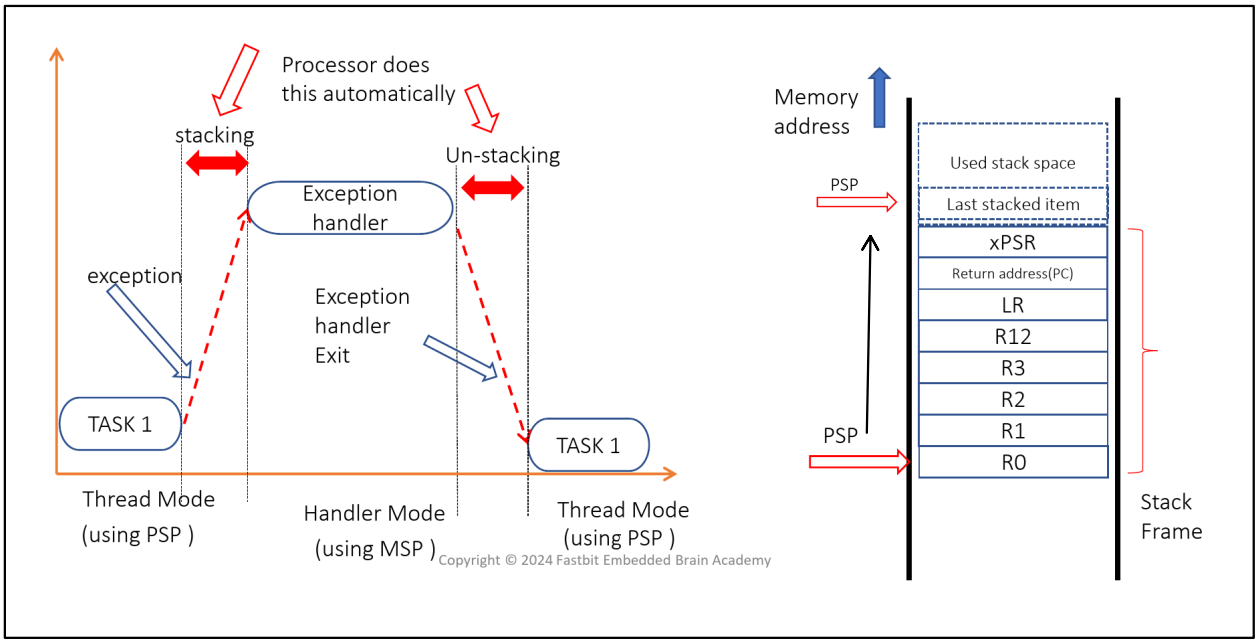
Case of T1 switching out, T2 switching in :



Stacking :



Unstacking :



This is normal exception stacking and unstacking flow

In our case after exception, it should come to task-2

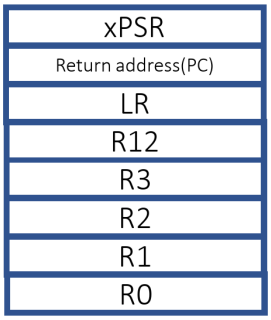
How to do that ?

Whenever we are in the exception handler, we should change the value of PSP to point to task-2's private stack,

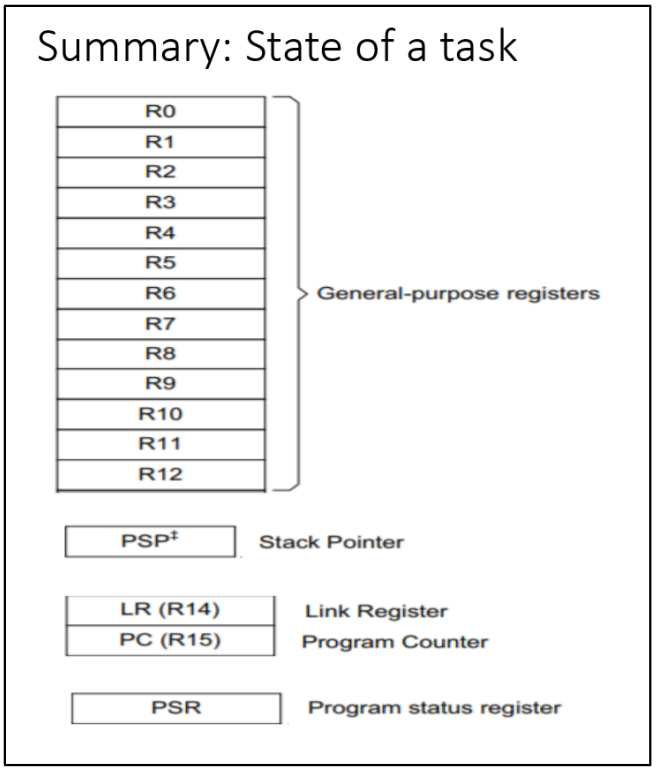
give the return address or address of task-2's handler in PC,

when we make a exit, processor will jump to task-2 handler automatically.

Whenever processor make a exception following frame will be saved in the stack automatically,



Apart from this, we need to store R4 - R11 registers manually in the exception handler (scheduler)



Current task :

- Configure the systick timer to produce exception for every 1ms

Note :

For any timer, you have to decide the count value, upto what point it should count, that value you have to program in any type of timer

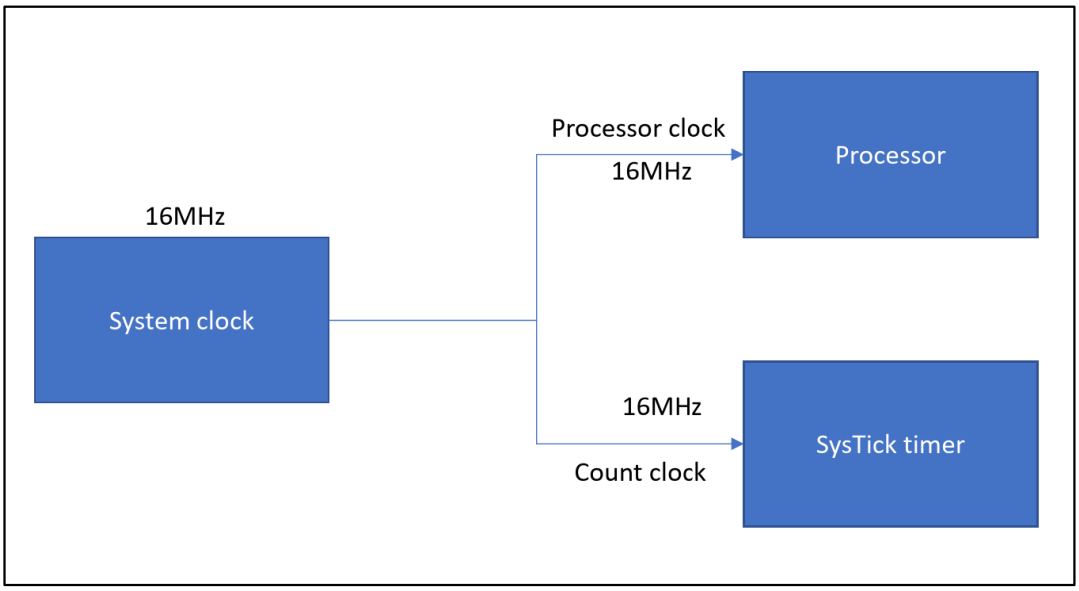
Systick timer makes exception for scheduler for every 1 ms

SysTick count value calculation :

- Processor Clock = 16MHz
- SysTick timer count clock = 16MHz
- 1ms is 1KHz in frequency domain
- So, to bring down SysTick timer count clock from 16MHz to 1KHz use a divisor (reload value)
- Reload value = 16000

$$\frac{16000000\ \text{Hz}}{16000\ \text{count value}} = 1000\ \text{Hz}$$

TICK_HZ
(Desired exception frequency)



Note:

We want 1000 exception (sys tick) in 1 sec. (i.e) In 1 sec, 1000 times context switching will happen

SysTick timer count clock = 16MHz

For 1 count it takes 0.0625 micro seconds

0.0625 µs delay	→	1 count
1 µs delay	→	16 count
1 ms delay	→	16000 count

```c
#define TICK_HZ                 1000U               // 1KHz
#define HSI_CLOCK               16000000U           // 16MHz
#define SYSTICK_TIM_CLK         HSI_CLOCK


int main(void)
{
    init_systick_timer(TICK_HZ);
    for(;;);
}
```

```c
void init_systick_timer(uint32_t tick_hz)
{
    uint32_t count_value = SYSTICK_TIM_CLK / tick_hz;
}
```

How to configure the count value in systick timer ?

4.4 System timer, SysTick

The processor has a 24-bit system timer, SysTick, that counts down from the reload value to zero, reloads, that is wraps to, the value in the SYST_RVR register on the next clock edge, then counts down on subsequent clocks.

Note

When the processor is halted for debugging the counter does not decrement.

The system timer registers are:

Table 4-32 System timer registers summary

Address	Name	Type	Required privilege	Reset value	Description
0xE000E010	SYST_CSR	RW	Privileged	a	*SysTick Control and Status Register*
0xE000E014	SYST_RVR	RW	Privileged	Unknown	*SysTick Reload Value Register* on page 4-34
0xE000E018	SYST_CVR	RW	Privileged	Unknown	*SysTick Current Value Register* on page 4-35
0xE000E01C	SYST_CALIB	RO	Privileged	-a	*SysTick Calibration Value Register* on page 4-35

a. See the register description for more information.

4.4.2 SysTick Reload Value Register

The SYST_RVR register specifies the start value to load into the SYST_CVR register. See the register summary in Table 4-32 on page 4-33 for its attributes. The bit assignments are:

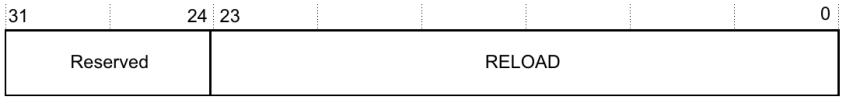


Table 4-34 SYST_RVR register bit assignments

Bits	Name	Function
[31:24]	-	Reserved.
[23:0]	RELOAD	Value to load into the SYST_CVR register when the counter is enabled and when it reaches 0, see *Calculating the RELOAD value*.

4.4.3 SysTick Current Value Register

The SYST_CVR register contains the current value of the SysTick counter. See the register summary in Table 4-32 on page 4-33 for its attributes. The bit assignments are:

Table 4-35 SYST_CVR register bit assignments

Bits	Name	Function
[31:24]	-	Reserved.
[23:0]	CURRENT	Reads return the current value of the SysTick counter. A write of any value clears the field to 0, and also clears the SYST_CSR COUNTFLAG bit to 0.

Calculating the reload value :

Whatever value you calculated, you will program that reload value reg, that value will be copied into current value reg,

you cannot directly modify current value reg, because its a downcounter.

e.g :

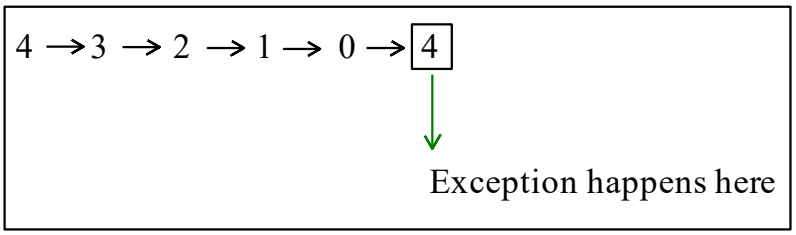
SRVR - 4

SCVR - 4

SCVR starts downcounting 1 digit for each clock cycle,

when it reaches 0 exception don't happen,

it will happen when the value get reloaded from SRVR



So totally it takes 5 clock cycles

But we wanted exception at 4th clock cycle, so always store N - 1 value.

Calculating the RELOAD value

The RELOAD value can be any value in the range `0x00000001-0x00FFFFFF`. A start value of 0 is possible, but has no effect because the SysTick exception request and COUNTFLAG are activated when counting from 1 to 0.

The RELOAD value is calculated according to its use. For example, to generate a multi-shot timer with a period of N processor clock cycles, use a RELOAD value of N-1. If the SysTick interrupt is required every 100 clock pulses, set RELOAD to 99.

```c
void init_systick_timer(uint32_t tick_hz)
{
    uint32_t *pSRVR = (uint32_t *)0xE000E014;
    uint32_t count_value = (SYSTICK_TIM_CLK / tick_hz) - 1;

    // Clear the value of SRVR
    *pSRVR &= ~(0x00FFFFFF);      // First 24 bits

    // Load the value into SRVR
    *pSRVR |= count_value;

    // Do some settings

    // Enable the systick timer

}
```

4.4.1 SysTick Control and Status Register

The SysTick SYST_CSR register enables the SysTick features. The register resets to `0x00000000`, or to `0x00000004` if your device does not implement a reference clock. See the register summary in Table 4-32 for its attributes. The bit assignments are:

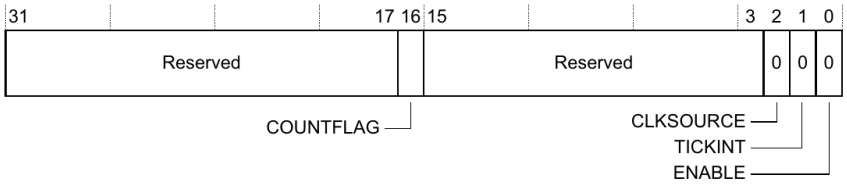


Table 4-33 SysTick SYST_CSR register bit assignments

Bits	Name	Function
[31:17]	-	Reserved.
[16]	COUNTFLAG	Returns 1 if timer counted to 0 since last time this was read.
[15:3]	-	Reserved.

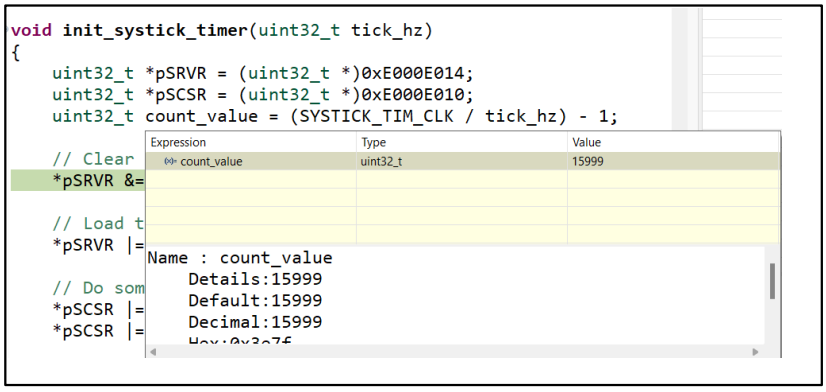
Table 4-33 SysTick SYST_CSR register bit assignments (continued)

Bits	Name	Function
[2]	CLKSOURCE	Indicates the clock source: 0 = external clock 1 = processor clock.
[1]	TICKINT	Enables SysTick exception request: 0 = counting down to zero does not assert the SysTick exception request 1 = counting down to zero asserts the SysTick exception request. Software can use COUNTFLAG to determine if SysTick has ever counted to zero.
[0]	ENABLE	Enables the counter: 0 = counter disabled 1 = counter enabled.

When ENABLE is set to 1, the counter loads the RELOAD value from the SYST_RVR register and then counts down. On reaching 0, it sets the COUNTFLAG to 1 and optionally asserts the SysTick depending on the value of TICKINT. It then loads the RELOAD value again, and begins counting.

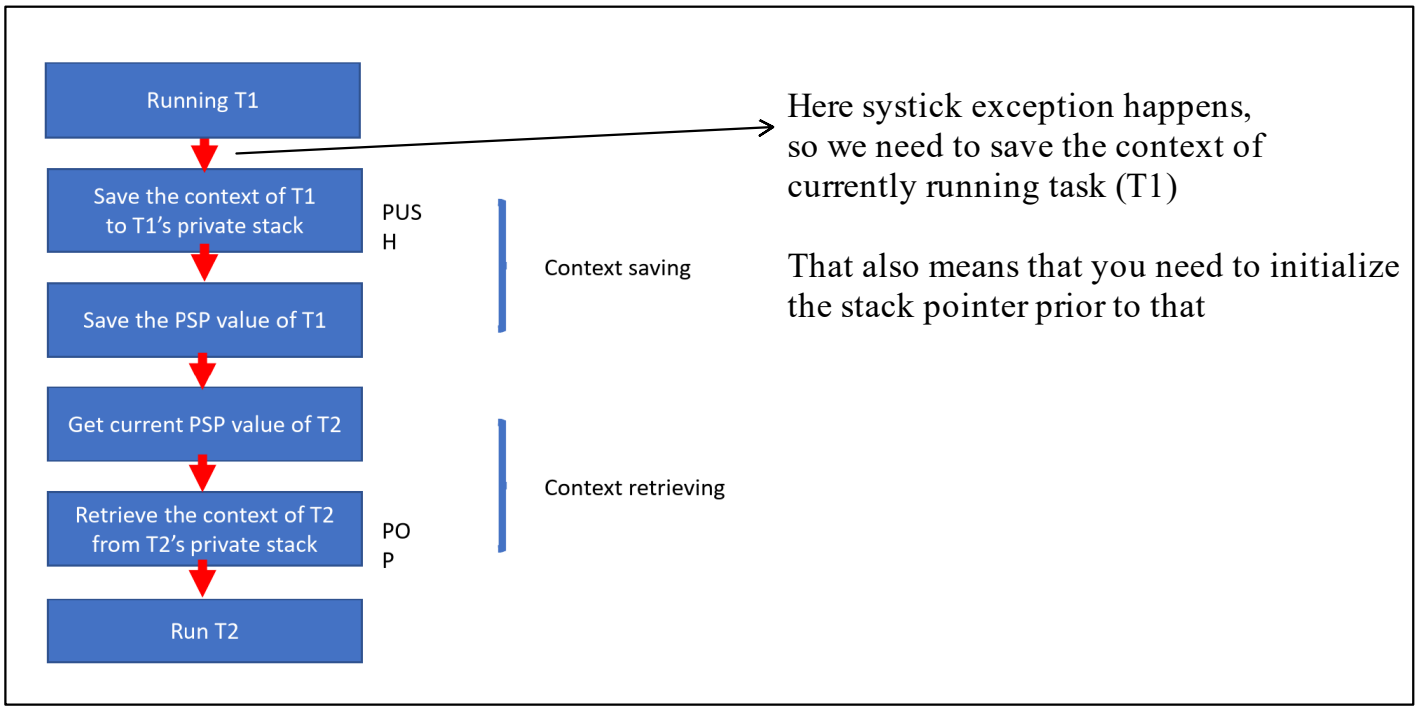
```c
void init_systick_timer(uint32_t tick_hz)
{
    uint32_t *pSRVR = (uint32_t *)0xE000E014;
    uint32_t *pSCSR = (uint32_t *)0xE000E010;
    uint32_t count_value = (SYSTICK_TIM_CLK / tick_hz) - 1;

    // Clear the value of SRVR
    *pSRVR &= ~(0x00FFFFFF);      // First 24 bits

    // Load the value into SRVR
    *pSRVR |= count_value;

    // Do some settings
    *pSCSR |= (1 << 1);          // Enable systick exception request
    *pSCSR |= (1 << 2);          // Indicates the clock source, processor CS

    // Enable the systick timer
    *pSCSR |= (1 << 0);          // Enables the counter
}

void SysTick_Handler(void)
{

}
```

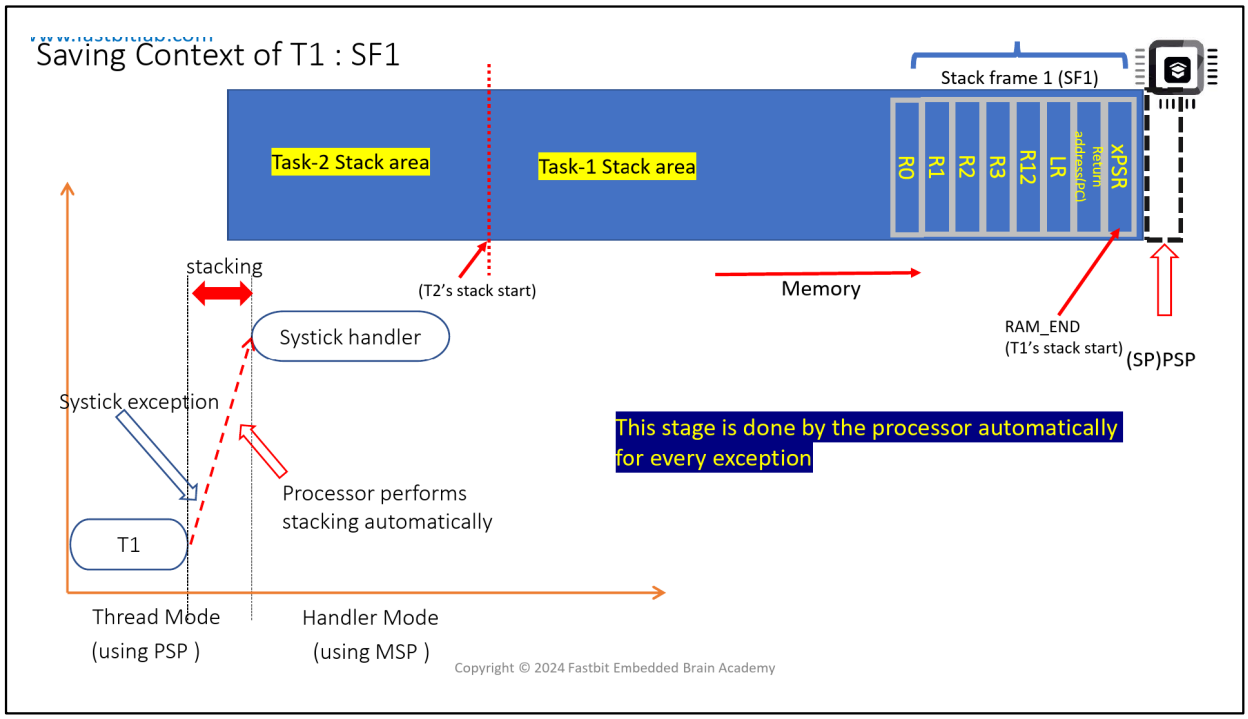
Debugging :



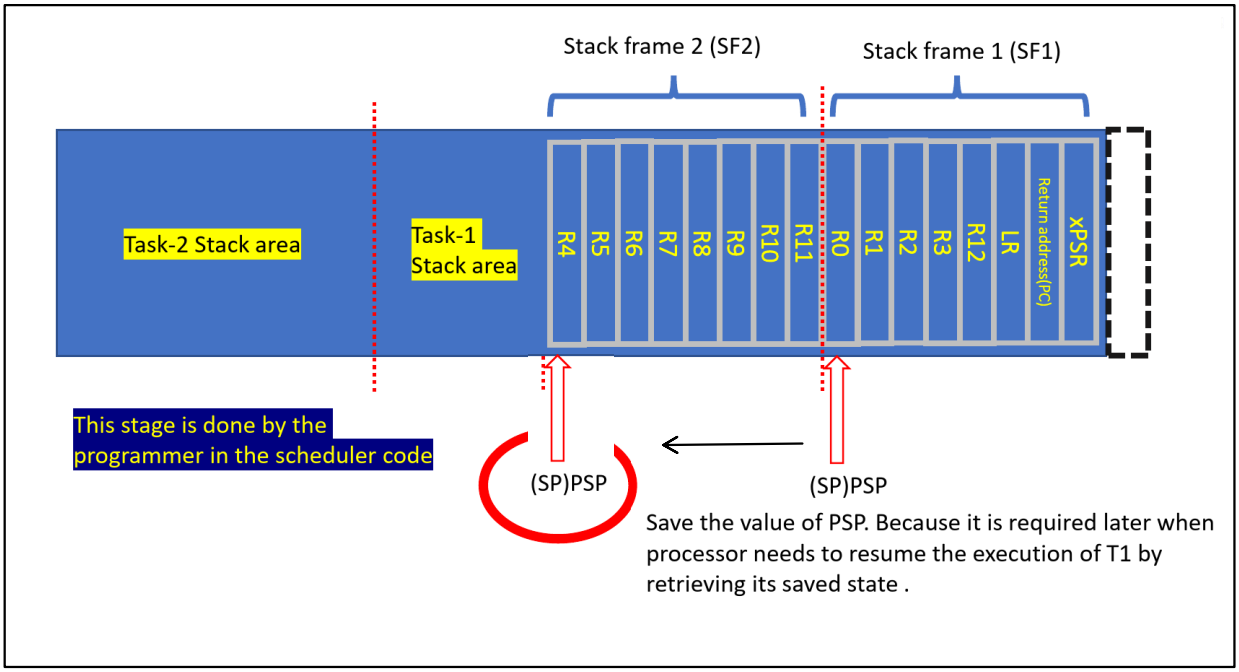
This is our scheduler, this will do the context switch

Context switching case study continued :

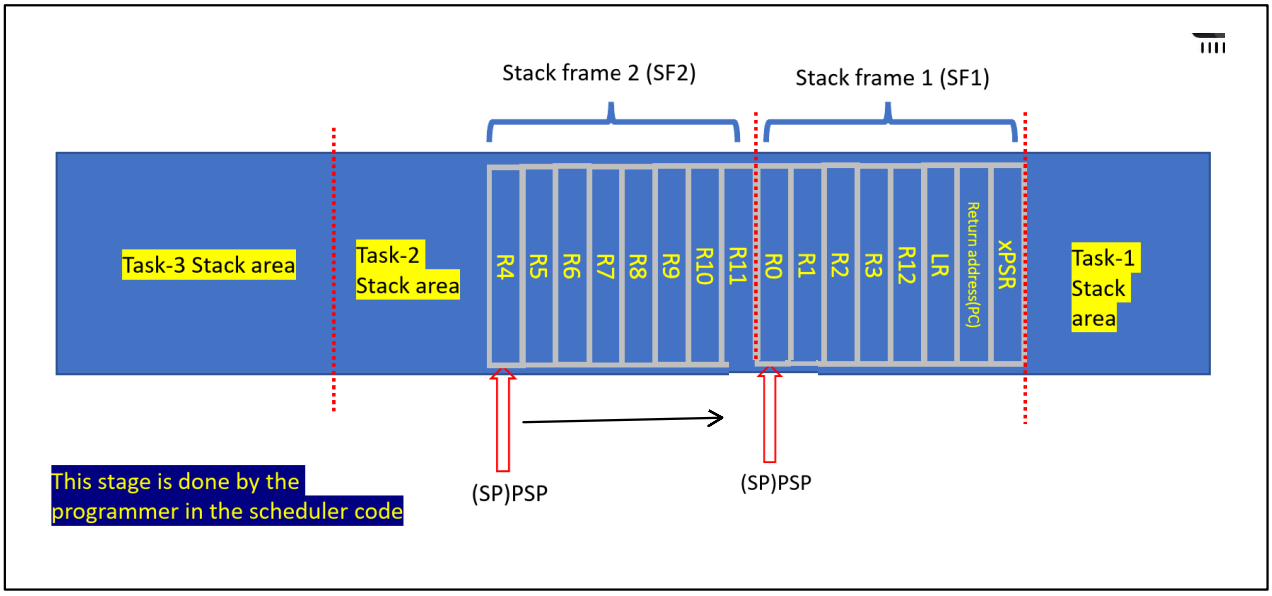




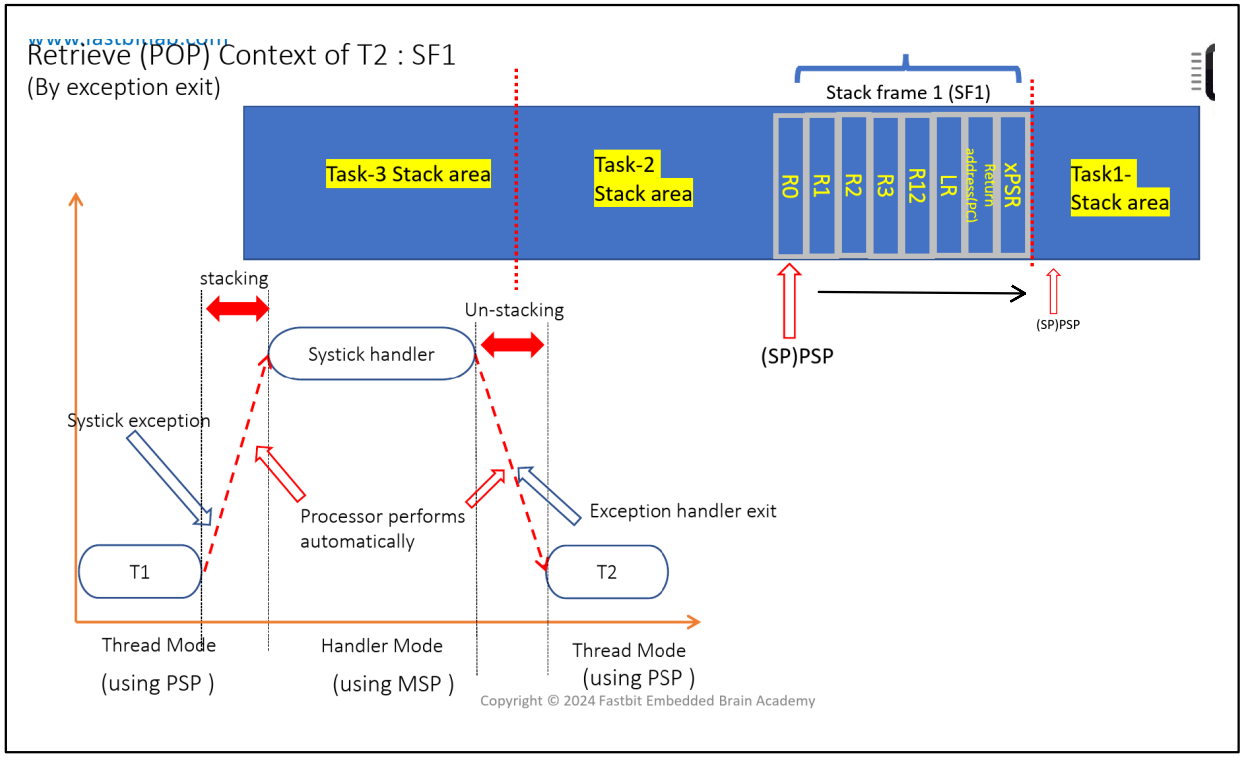
Complete state of a task :



Retrieve (POP) Context of T2 : SF2



After this you should exit the exception handler using exception exit,



When you exit the exception handler, the execution will resume to task-2,

why ? Because return address belongs to task-2 (stack frame of task-2)

What if Task-2 is getting executed for very first time, then its state will not be available in its private stack area ?

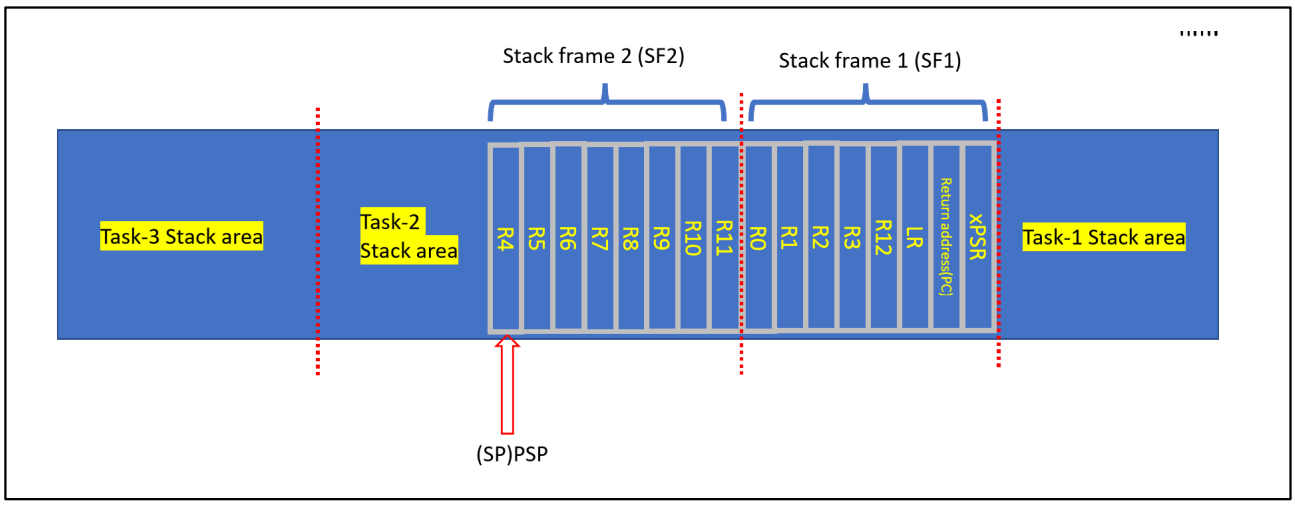
To solve this problem, we can create some dummy stack frames

Task's stack area init and storing of dummy SF :

- Each task can consume a maximum of 1KB of memory as a private stack
- This stack is used to hold tasks local variables and context(SF1+SF2)
- When a Task is getting scheduled for the very first time, it doesn't have any context. So, the programmer should store dummy SF1 and SF2 in Task's stack area as a part of "task initialization" sequence before launching the scheduler.

This dummy initialization should be done for each tasks.

Dummy initial context of a task :



We told dummy task initialization need to be done, before launching scheduler,

so tasks won't have any state yet,

so you can keep all the general purpose registers as 0 (R0 -R11),

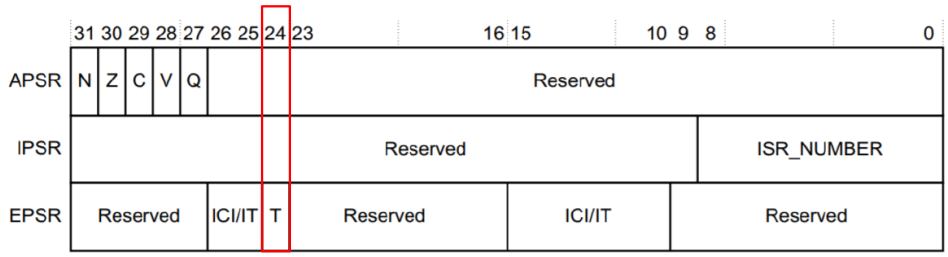
but those 3 special registers are important to store important information.

xPSR :

The *Program Status Register* (PSR) combines:

- *Application Program Status Register* (APSR)
- *Interrupt Program Status Register* (IPSR)
- *Execution Program Status Register* (EPSR).

These registers are mutually exclusive bitfields in the 32-bit PSR. The bit assignments are:



Access these registers individually or as a combination of any two or all three registers, using the register name as an argument to the MSR or MRS instructions. For example:

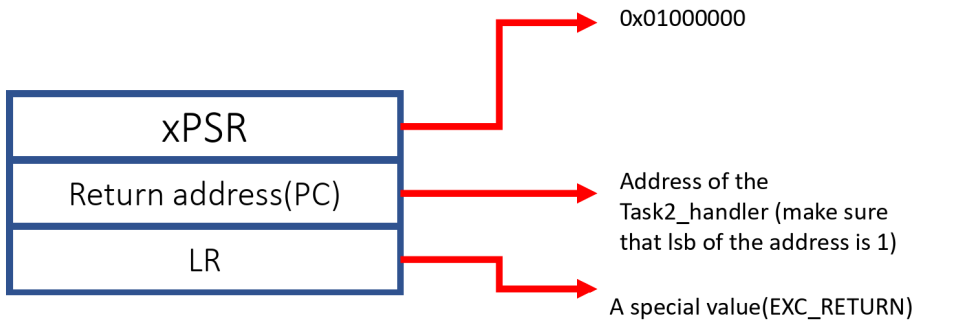
- read all of the registers using PSR with the MRS instruction
- write to the APSR N, Z, C, V, and Q bits using APSR_nzcvq with the MSR instruction.

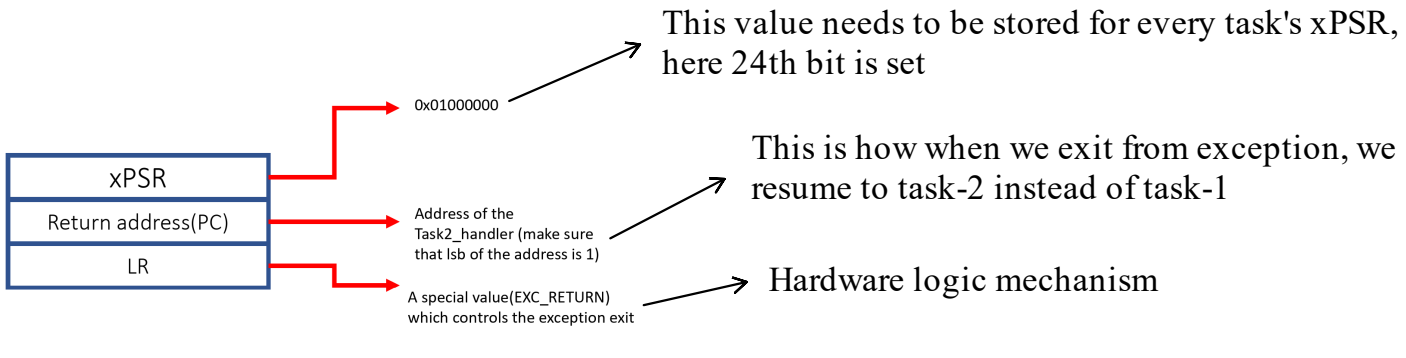
Thumb state

The Cortex-M4 processor only supports execution of instructions in Thumb state. The following can clear the T bit to 0:

- instructions BLX, BX and POP{PC}
- restoration from the stacked xPSR value on an exception return
- bit[0] of the vector value on an exception entry or reset.

Attempting to execute instructions when the T bit is 0 results in a fault or lockup. See *Lockup* on page 2-31 for more information.





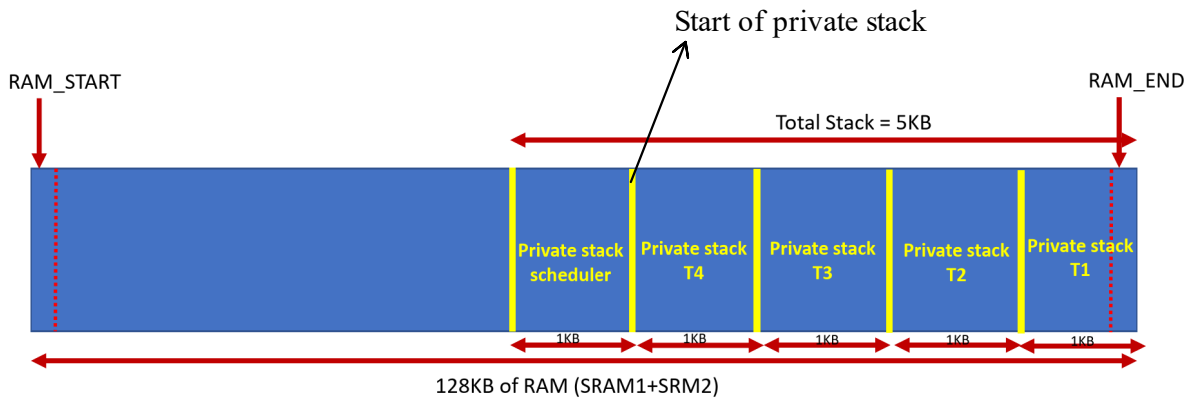
One among the below values need to be considered for LR,

Table 2-17 Exception return behavior

EXC_RETURN[31:0]	Description
0xFFFFFFF1	Return to Handler mode, exception return uses non-floating-point state from the MSP and execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode, exception return uses non-floating-point state from MSP and execution uses MSP after return.
0xFFFFFFFD	Return to Thread mode, exception return uses non-floating-point state from the PSP and execution uses PSP after return.
0xFFFFFFE1	Return to Handler mode, exception return uses floating-point-state from MSP and execution uses MSP after return.
0xFFFFFFE9	Return to Thread mode, exception return uses floating-point state from MSP and execution uses MSP after return.
0xFFFFFFED	Return to Thread mode, exception return uses floating-point state from PSP and execution uses PSP after return.

Initialization of stack :

Initialize Scheduler stack pointer (MSP):




```
int main(void)
{
    init_scheduler_stack(SCHED_STACK_START);

    init_systick_timer(TICK_HZ);

    for(;;);
}
```

We have to now access MSP to change the value of MSP,

MSP is special register of the processor, there is no C function to access that, so we have to use inline assembly.

So our function should be naked. (assembly routine)

```
__attribute__((naked)) init_scheduler_stack(uint32_t sched_top_of_stack)
{

}
```

```
__attribute__((naked)) init_scheduler_stack(uint32_t sched_top_of_stack)
{
    __asm volatile("MSR MSP,R0");
}
```

Our first argument will be stored in R0

or gcc assembly syntax of C variable, you also write something like below :

```
__attribute__((naked)) init_scheduler_stack(uint32_t sched_top_of_stack)
{
    __asm volatile("MSR MSP,%0": : "r" (sched_top_of_stack) : );
}
```

Now we have to go back to main, because this is a assembly routine no prologue epilogue sequence,

```
__attribute__((naked)) init_scheduler_stack(uint32_t sched_top_of_stack)
{
    __asm volatile("MSR MSP,%0": : "r" (sched_top_of_stack) : );
    __asm volatile("BX LR");
}
```

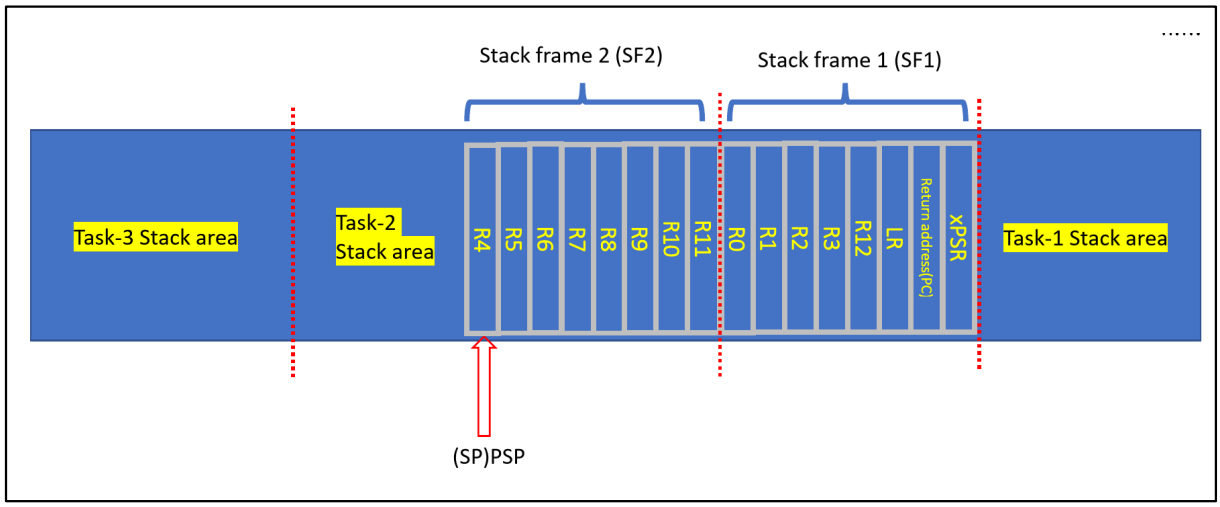
Copies LR(return address-main()) value into PC

Current task :

Init tasks stack memory

- Store dummy SF1 and SF2 in stack memory of each task

Dummy initial context of a Task :



Registers need to be store as it is

```c
uint32_t psp_of_tasks[MAX_TASKS] = {T1_STACK_START, T2_STACK_START, T3_STACK_START, T4_STACK_START};

int main(void)
{
    init_scheduler_stack(SCHED_STACK_START);

    init_tasks_stack();

    init_systick_timer(TICK_HZ);

    for(;;);
}
```

```c
void init_tasks_stack(void)
{
    // Initializing 4 user tasks
    uint32_t *pPSP;

    for(uint32_t i = 0; i < MAX_TASKS; i++)
    {
        pPSP = (uint32_t *)psp_of_tasks[i];
    }
}
```

Stack model is full descending, so decrement first and then store the value,

```c
uint32_t task_handlers[MAX_TASKS];

int main(void)
{
    init_scheduler_stack(SCHED_STACK_START);

    task_handlers[0] = (uint32_t)task1_handler;
    task_handlers[1] = (uint32_t)task2_handler;
    task_handlers[2] = (uint32_t)task3_handler;
    task_handlers[3] = (uint32_t)task4_handler;

    init_tasks_stack();

    init_systick_timer(TICK_HZ);

    for(;;);
}
```

→ Storing addresses of different task_handlers

```c
void init_tasks_stack(void)
{
    // Initializing 4 user tasks
    uint32_t *pPSP;

    for(uint32_t i = 0; i < MAX_TASKS; i++)
    {
        pPSP = (uint32_t *)psp_of_tasks[i];

        pPSP--;                         // xPSR
        *pPSP = DUMMY_XPSR;             // 0x01000000

        pPSP--;                         // PC
        *pPSP = task_handlers[i];       // Address of task handler

        pPSP--;                         // LR
        *pPSP = 0xFFFFFFFD;             // EXC_RETURN

        for(uint32_t j = 0; j < 13; j++)
        {
            pPSP--;
            *pPSP = 0;
        }
        psp_of_tasks[i] = (uint32_t)pPSP;        // Preserving PSP
    }
}
```

Stack pointer setup :

```
int main(void)
{
    /*Initialize MSP*/
    init_scheduler_stack(SCHED_STACK_START);

    /*Addresses of task handlers in array*/
    task_handlers[0] = (uint32_t)task1_handler;
    task_handlers[1] = (uint32_t)task2_handler;
    task_handlers[2] = (uint32_t)task3_handler;
    task_handlers[3] = (uint32_t)task4_handler;

    /*Stack initialization to store dummy frames*/
    init_tasks_stack();

    /*To generate systick timer exception*/
    init_systick_timer(TICK_HZ);

    /*Upto now SP is MSP*/
    /*Now we need to call user tasks, so PSP*/
    switch_sp_to_psp();

    task1_handler();

    for(;;);
}
```

We are dealing with stack memory, we may be touching / doing some illegal activities related to memory / inline assembly, fault can occur while changing from handler to thread mode code,

inorder to trace the fault, we will enable fault handlers,

```
void enable_processor_faults()
{
    // Enable all configurable exceptions like usage fault, mem manage fault and bus fault

    uint32_t *pSHCSR = (uint32_t*)0xE000ED24;

    *pSHCSR |= ( 1 << 16); //mem manage
    *pSHCSR |= ( 1 << 17); //bus fault
    *pSHCSR |= ( 1 << 18); //usage fault
}

// Implement the fault handlers
void HardFault_Handler(void)
{
    printf("Exception : Hardfault\n");
    while(1);
}


void MemManage_Handler(void)
{
    printf("Exception : MemManage\n");
    while(1);
}

void BusFault_Handler(void)
{
    printf("Exception : BusFault\n");
    while(1);
}
```

We can now change SP to PSP using control register,

Before changing to PSP, we should initialize PSP first.

Control register is a special register so we should use naked function.

```
uint8_t current_task = 0;               // Task-1 is running

void get_psp_value()
{
    return psp_of_tasks[current_task];
}

__attribute__((naked)) void switch_sp_to_psp()
{
    // 1. Initialize the PSP with TASK1 stack start

    // 2. Change SP to PSP using CONTROL register
}
```

return value is stored in r0 according to procedure call standard,

(i.e) stack of current task is stored in r0 (initial stack address of current task)

```
uint8_t current_task = 0;               // Task-1 is running

void get_psp_value()
{
    return psp_of_tasks[current_task];
}

__attribute__((naked)) void switch_sp_to_psp(void)
{
    // 1. Initialize the PSP with TASK1 stack start

    // Get the value of PSP of current task
    __asm volatile ("BL get_psp_value");
    __asm volatile ("MSR PSP, R0");

    // 2. Change SP to PSP using CONTROL register
}
```

But in the above code, there is a problem, this first instruction is BL (Branch with Link), after this instruction, the value of LR will be corrupted becuase this function (switch_sp_to_psp( )) is called from main( ),

LR is holding some value previously,

after this function, it has to go back to main, which is done through the value of LR,

in our first instruction we are moving get_psp_value routine, so LR value is corrupted,

so we need to save LR.

```
__attribute__((naked)) void switch_sp_to_psp(void)
{
    // 1. Initialize the PSP with TASK1 stack start

    // Get the value of PSP of current task
    __asm volatile ("PUSH {LR}");
    __asm volatile ("BL get_psp_value");
    __asm volatile ("MSR PSP, R0");
    __asm volatile ("POP {LR}");

    // 2. Change SP to PSP using CONTROL register
}
```

All these instructions use, SP as MSP, because we didn't perform the switch yet.

CONTROL register

The CONTROL register controls the stack used and the privilege level for software execution when the processor is in Thread mode and, if implemented, indicates whether the FPU state is active. See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:

Table 2-10 CONTROL register bit assignments

Bits	Name	Function
[31:3]	-	Reserved.
[2]	FPCA	When floating-point is implemented this bit indicates whether context floating-point is currently active: 0 = no floating-point context active 1 = floating-point context active. The Cortex-M4 uses this bit to determine whether to preserve floating-point state when processing an exception.
[1]	SPSEL	Defines the currently active stack pointer: In Handler mode this bit reads as zero and ignores writes. The Cortex-M4 updates this bit automatically on exception return: 0 = MSP is the current stack pointer 1 = PSP is the current stack pointer.
[0]	nPRIV	Defines the Thread mode privilege level: 0 = privileged 1 = unprivileged.

```
__attribute__((naked)) void switch_sp_to_psp(void)
{
    // 1. Initialize the PSP with TASK1 stack start

    // Get the value of PSP of current task
    __asm volatile ("PUSH {LR}");         // preserve LR which connects back to main()
    __asm volatile ("BL get_psp_value");
    __asm volatile ("MSR PSP,R0");       // Initialize PSP
    __asm volatile ("POP {LR}");          // pops back LR value

    // 2. Change SP to PSP using CONTROL register
    __asm volatile ("MOV R0,#0x02");
    __asm volatile ("MSR CONTROL,R0");
    __asm volatile ("BX LR");
}
```

Going back to main( ), here LR value will be copied to PC.

Implementing systick handler :

```
void SysTick_Handler(void)
{
    /*Save the context of current task*/
    // 1. Get current running task's PSP value
    // 2. Using that PSP value to store SF2 (R4 - R11)
    // 3. Save the current value of PSP


    /*Retrieve the context of next task*/
    // 1. Decide next task to run
    // 2. Get its past PSP value
    // 3. Using that PSP value to retrieve SF2 (R4 - R11)
    // 4. Update PSP and exit
}
```

```
void SysTick_Handler(void)
{
    /*Save the context of current task*/

    // 1. Get current running task's PSP value
    __asm volatile("MRS R0,PSP");
    // 2. Using that PSP value to store SF2 (R4 - R11)

    // 3. Save the current value of PSP


    /*Retrieve the context of next task*/
    // 1. Decide next task to run
    // 2. Get its past PSP value
    // 3. Using that PSP value to retrieve SF2 (R4 - R11)
    // 4. Update PSP and exit
}
```

Here we can't use PUSH instruction, why ?

Because this is a handler, if we use PUSH, MSP will be affected.

In handler mode, processor will always use MSP

Now we have to store the remaining state (R4 - R11) of currently running task using store instruction, we have to simulate PUSH operation,

STMDB, STMEA	Rn{!}, reglist	Store Multiple registers, decrement before	-	page 3-32
STMFD, STMIA	Rn{!}, reglist	Store Multiple registers, increment after	-	page 3-32

Syntax

op{*addr_mode*}{*cond*} *Rn*{!}, *reglist*

where:

op Is one of:
- LDM Load Multiple registers.
- STM Store Multiple registers.

addr_mode Is any one of the following:
- IA Increment address After each access. This is the default.
- DB Decrement address Before each access.

cond Is an optional condition code, see *Conditional execution* on page 3-18.

Rn Specifies the register on which the memory addresses are based.

! Is an optional writeback suffix. If ! is present the final address, that is loaded from or stored to, is written back into *Rn*.

reglist Is a list of one or more registers to be loaded or stored, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range, see *Examples* on page 3-33.

Examples

```
LDM     R8,{R0,R2,R9}      ; LDMIA is a synonym for LDM
STMDB   R1!,{R3-R6,R11,R12}
```

Incorrect examples

```
STM     R5!,{R5,R4,R9} ; Value stored for R5 is unpredictable
LDM     R2, {}         ; There must be at least one register in the list.
```

This is a pointer or address to store

! means after each store R0 will get updated at the current address

```
void SysTick_Handler(void)
{
    /*Save the context of current task*/

    // 1. Get current running task's PSP value
    __asm volatile("MRS R0,PSP");
    // 2. Using that PSP value to store SF2 (R4 - R11)
    __asm volatile("STMDB R0!,{R4-R11}");

    // 3. Save the current value of PSP


    /*Retrieve the context of next task*/
    // 1. Decide next task to run
    // 2. Get its past PSP value
    // 3. Using that PSP value to retrieve SF2 (R4 - R11)
    // 4. Update PSP and exit
}
```

```
void save_psp_value(uint32_t current_psp_value)
{
    psp_of_tasks[current_task] = current_psp_value;
}
```

```
void SysTick_Handler(void)
{
    /*Save the context of current task*/

    // 1. Get current running task's PSP value
    __asm volatile("MRS R0,PSP");
    // 2. Using that PSP value to store SF2 (R4 - R11)
    __asm volatile("STMDB R0!,{R4-R11}");
    // 3. Save the current value of PSP
    __asm volatile("BL save_psp_value");


    /*Retrieve the context of next task*/
    // 1. Decide next task to run
    // 2. Get its past PSP value
    // 3. Using that PSP value to retrieve SF2 (R4 - R11)
    // 4. Update PSP and exit
}
```

In the second instruction, R0 contains the updated value (cuurent psp value), so in third instruction, when you call save_psp_value routine, R0 will be sent as an argument according to the procedure call standard.

```
void update_next_task(void)
{
    current_task++;
    current_task %= MAX_TASKS;
}
```

```
/*Retrieve the context of next task*/

// 1. Decide next task to run
__asm volatile("BL update_next_task");
// 2. Get its past PSP value
__asm volatile("BL get_psp_value");
// 3. Using that PSP value to retrieve SF2 (R4 - R11)
// 4. Update PSP and exit
```

When this called, return value will be stored in R0

Now by using the PSP value, we have to store from stack to register (memory to register)

We have to use Load instruction.

LDMDB, LDMEA	Rn{!}, reglist	Load Multiple registers, decrement before	-	page 3-32
LDMFD, LDMIA	Rn{!}, reglist	Load Multiple registers, increment after	-	page 3-32

Syntax

op{*addr_mode*}{*cond*} *Rn*{!}, *reglist*

where:

op	Is one of:	
	LDM	Load Multiple registers.
	STM	Store Multiple registers.
addr_mode	Is any one of the following:	
	IA	Increment address After each access. This is the default.
	DB	Decrement address Before each access.

IA is default, so LDM is sufficient

```
/*Retrieve the context of next task*/

// 1. Decide next task to run
__asm volatile("BL update_next_task");
// 2. Get its past PSP value
__asm volatile("BL get_psp_value");
// 3. Using that PSP value to retrieve SF2 (R4 - R11)
__asm volatile("LDMIA R0!,{R4-R11}");
// 4. Update PSP and exit
__asm volatile("MSR PSP,R0");
```

Complete systick_handler :

```
void SysTick_Handler(void)
{
    /*Save the context of current task*/

    // 1. Get current running task's PSP value
    __asm volatile("MRS R0,PSP");
    // 2. Using that PSP value to store SF2 (R4 - R11)
    __asm volatile("STMDB R0!,{R4-R11}");
    // 3. Save the current value of PSP
    __asm volatile("BL save_psp_value");


    /*Retrieve the context of next task*/

    // 1. Decide next task to run
    __asm volatile("BL update_next_task");
    // 2. Get its past PSP value
    __asm volatile("BL get_psp_value");
    // 3. Using that PSP value to retrieve SF2 (R4 - R11)
    __asm volatile("LDMIA R0!,{R4-R11}");
    // 4. Update PSP and exit
    __asm volatile("MSR PSP,R0");
}
```

When you exit the systick_handler, usual exception exit sequence will take place automatically by the processor and tries to fetch SF1,

Reference for that SF1 is nothing bu stack address of the next updated task. Thats why we have modified PSP before exiting,

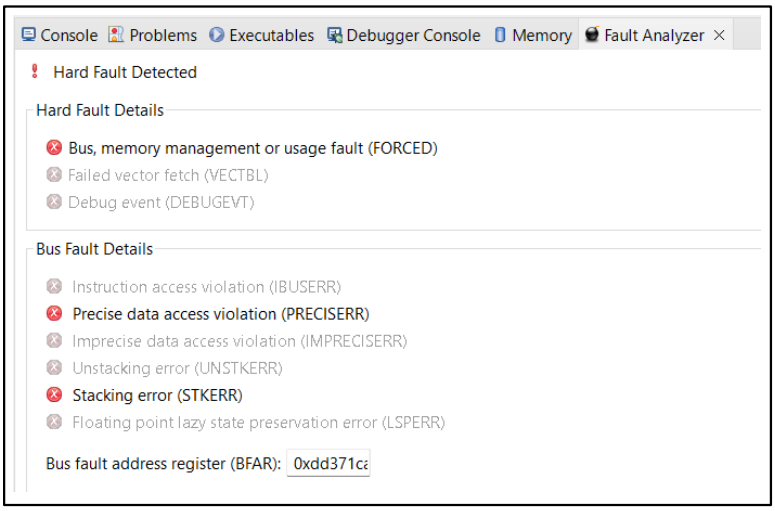
PSP is now pointing to stack area of next updated task, thats why execution will go to next task instead of the same.

Testing :

Lets add breakpoint at task-2, task-3, task-4 and handler and execute the code,

when we debug our application, it result in hardfault :

```
207 // Implement the fault handlers
208 void HardFault_Handler(void)
209 {
210     printf("Exception : Hardfault\n");
211     while(1);
212 }
213
```



something related to stacking error, lets debug step by step,

```
26 int main(void)
27 {
28     enable_processor_faults();
29
30     /*Initialize MSP*/
31     init_scheduler_stack(SCHED_STACK_START);
32
33     /*Addresses of task handlers in array*/
34     task_handlers[0] = (uint32_t)task1_handler;
35     task_handlers[1] = (uint32_t)task2_handler;
36     task_handlers[2] = (uint32_t)task3_handler;
37     task_handlers[3] = (uint32_t)task4_handler;
38
39     /*Stack initialization to store dummy frames*/
40     init_tasks_stack();
41
42     /*To generate systick timer exception*/
43     init_systick_timer(TICK_HZ);
44
45     /*Upto now SP is MSP*/
46     /*Now we need to call user tasks, so PSP*/
47     switch_sp_to_psp();
48
49     task1_handler();
50
51     for(;;);
52 }
```

Upto this, there is no problem, this means that there is a problem in our scheduler

Lets add a breakpoint in systick handler,

```
183 void SysTick_Handler(void)
184 {
185     /*Save the context of current task*/
186
187     // 1. Get current running task's PSP value
188     __asm volatile("MRS R0,PSP");
189     // 2. Using that PSP value to store SF2 (R4 - R11)
190     __asm volatile("STMDB R0!,{R4-R11}");
191     // 3. Save the current value of PSP
192     __asm volatile("BL save_psp_value");
193
194
195     /*Retrieve the context of next task*/
196
197     // 1. Decide next task to run
198     __asm volatile("BL update_next_task");
199     // 2. Get its past PSP value
200     __asm volatile("BL get_psp_value");
201     // 3. Using that PSP value to retrieve SF2 (R4 - R11)
202     __asm volatile("LDMIA R0!,{R4-R11}");
203     // 4. Update PSP and exit
204     __asm volatile("MSR PSP,R0");
205 }
```

Upto here there is no problem at all,

```
195     /*Retrieve the context of next task*/
196
197     // 1. Decide next task to run
198     __asm volatile("BL update_next_task");
199     // 2. Get its past PSP value
200     __asm volatile("BL get_psp_value");
201     // 3. Using that PSP value to retrieve SF2 (R4 - R11)
202     __asm volatile("LDMIA R0!,{R4-R11}");
203     // 4. Update PSP and exit
204     __asm volatile("MSR PSP,R0");
205 }
```

After this, it looping back to systick handler,

```
197     // 1. Decide next task to run
198     __asm volatile("BL update_next_task");
199     // 2. Get its past PSP value
200     __asm volatile("BL get_psp_value");
201     // 3. Using that PSP value to retrieve SF2 (R4 - R11)
202     __asm volatile("LDMIA R0!,{R4-R11}");
203     // 4. Update PSP and exit
204     __asm volatile("MSR PSP,R0");
205 }
```

```
197     // 1. Decide next task to run
198     __asm volatile("BL update_next_task");
199     // 2. Get its past PSP value
200     __asm volatile("BL get_psp_value");
201     // 3. Using that PSP value to retrieve SF2 (R4 - R11)
202     __asm volatile("LDMIA R0!,{R4-R11}");
203     // 4. Update PSP and exit
204     __asm volatile("MSR PSP,R0");
205 }
206
207 // Implement the fault handlers
208 void HardFault_Handler(void)
209 {
210     printf("Exception : Hardfault\n");
211     while(1);
212 }
```

Debugging again :

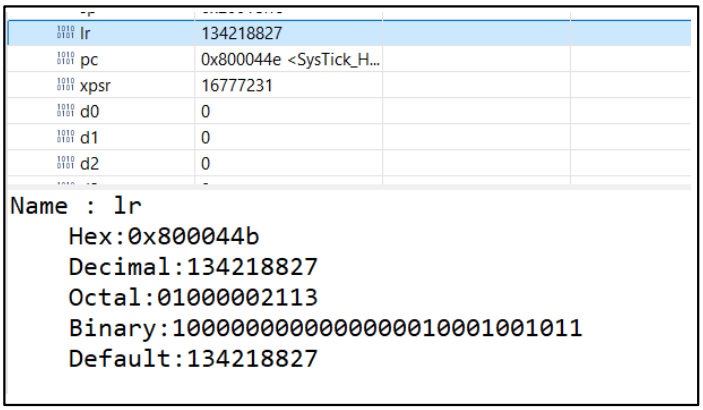
```
183 void SysTick_Handler(void)
184 {
185     /*Save the context of current task*/
186
187     // 1. Get current running task's PSP value
188     __asm volatile("MRS R0,PSP");
189     // 2. Using that PSP value to store SF2 (R4 - R11)
190     __asm volatile("STMDB R0!,{R4-R11}");
191     // 3. Save the current value of PSP
192     __asm volatile("BL save_psp_value");
193
194
195     /*Retrieve the context of next task*/
196
197     // 1. Decide next task to run
198     __asm volatile("BL update_next_task");
199     // 2. Get its past PSP value
200     __asm volatile("BL get_psp_value");
201     // 3. Using that PSP value to retrieve SF2 (R4 - R11)
202     __asm volatile("LDMIA R0!,{R4-R11}");
203     // 4. Update PSP and exit
204     __asm volatile("MSR PSP,R0");
205 }
```

After this line, it needs to exit the systick handler

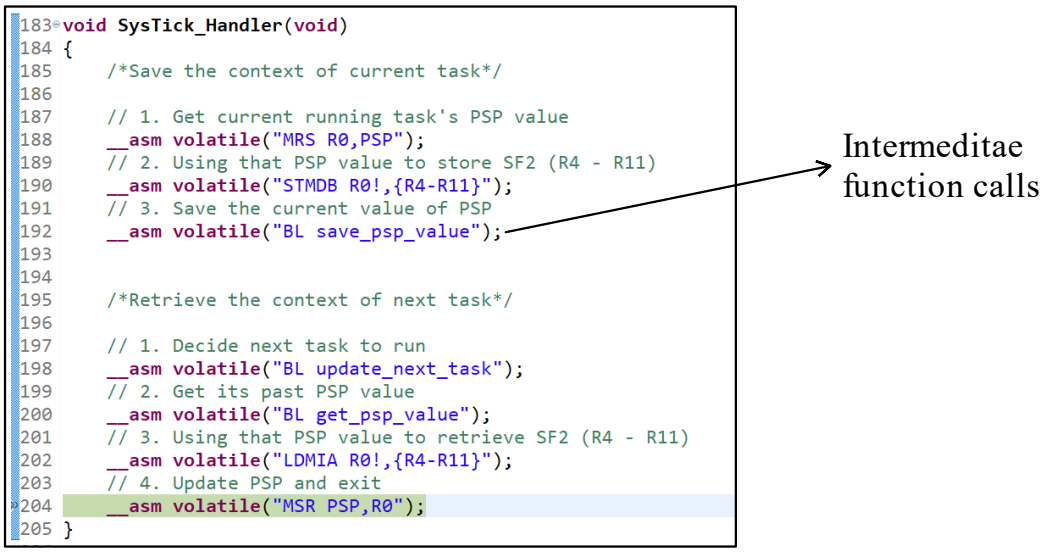
The problem here is LR,

we know after exception exit, LR must contain EXC_RETURN,

it contains a garbage value :



We now see that we have corrupted the LR value because of intermediate function calls,



Before we use BL instruction (manipulating LR), we have to save the content of LR and pop back,

```
__attribute__((naked)) void SysTick_Handler(void)
{
    /*Save the context of current task*/

    // 1. Get current running task's PSP value
    __asm volatile("MRS R0,PSP");
    // 2. Using that PSP value to store SF2 (R4 - R11)
    __asm volatile("STMDB R0!,{R4-R11}");

    __asm volatile("PUSH {LR}");
    // 3. Save the current value of PSP
    __asm volatile("BL save_psp_value");


    /*Retrieve the context of next task*/

    // 1. Decide next task to run
    __asm volatile("BL update_next_task");
    // 2. Get its past PSP value
    __asm volatile("BL get_psp_value");
    // 3. Using that PSP value to retrieve SF2 (R4 - R11)
    __asm volatile("LDMIA R0!,{R4-R11}");
    // 4. Update PSP and exit
    __asm volatile("MSR PSP,R0");

    __asm volatile("POP {LR}");
}
```

Before manipulation

After manipulation

We made this systick handler as naked function because we don't want any epilogue and prologue sequence,

so we need to add manually write instruction for exception exit,

when exception exit happens ? When EXC_RETURN value is copied into PC,

```
__attribute__((naked)) void SysTick_Handler(void)
{
    /*Save the context of current task*/

    // 1. Get current running task's PSP value
    __asm volatile("MRS R0,PSP");
    // 2. Using that PSP value to store SF2 (R4 - R11)
    __asm volatile("STMDB R0!,{R4-R11}");

    __asm volatile("PUSH {LR}");
    // 3. Save the current value of PSP
    __asm volatile("BL save_psp_value");


    /*Retrieve the context of next task*/

    // 1. Decide next task to run
    __asm volatile("BL update_next_task");
    // 2. Get its past PSP value
    __asm volatile("BL get_psp_value");
    // 3. Using that PSP value to retrieve SF2 (R4 - R11)
    __asm volatile("LDMIA R0!,{R4-R11}");
    // 4. Update PSP and exit
    __asm volatile("MSR PSP,R0");

    __asm volatile("POP {LR}");

    __asm volatile("BX LR");
}
```

In inti_tasks_stack( ), the address of task handler needs to be odd, becuase LSB represents T-bit.

```c
void init_tasks_stack(void)
{
    // Initializing 4 user tasks
    uint32_t *pPSP;

    for(uint32_t i = 0; i < MAX_TASKS; i++)
    {
        pPSP = (uint32_t *)psp_of_tasks[i];

        pPSP--;                         // xPSR
        *pPSP = DUMMY_XPSR;             // 0x01000000

        pPSP--;                         // PC
        *pPSP = task_handlers[i];       // Address of task handler

        pPSP--;                         // LR
        *pPSP = 0xFFFFFFFD;             // EXC_RETURN

        for(uint32_t j = 0; j < 13; j++)
        {
            pPSP--;
            *pPSP = 0;
        }
        psp_of_tasks[i] = (uint32_t)pPSP;        // Preserving PSP
    }
}
```

Expression	Type	Value
task_handlers	uint32_t [4]	0x20000088 <task_handl...
task_handlers[0]	uint32_t	134218273
task_handlers[1]	uint32_t	134218289
task_handlers[2]	uint32_t	134218305
task_handlers[3]	uint32_t	134218321
Add new expression		

```
Name : task_handlers[0]
    Details:134218273
    Default:134218273
    Decimal:134218273
    Hex:0x8000221
    Binary:10000000000000000001000100001
    Octal:01000001041
```

Yes it is odd

task1_handler	void (void)	{void (void)} 0x8000220 ...
Add new expression		

```
Name : task1_handler
    Details:{void (void)} 0x8000220 <task1_handler>
    Default:{void (void)} 0x8000220 <task1_handler>
    Decimal:-128
    Hex:0x80
    Binary:10000000
    Octal:0200
```

even address is incremented to represent T-bit.

Debugging again after the changes,

```
62 void task2_handler(void)
63 {
64     while(1)
65     {
66         printf("This is task-2\n");
67     }
68 }
```

control came to task-2 after task-1 execution

```
70 void task3_handler(void)
71 {
72     while(1)
73     {
74         printf("This is task-3\n");
75     }
76 }
```

After task-2, task-3 executed

```
78 void task4_handler(void)
79 {
80     while(1)
81     {
82         printf("This is task-4\n");
83     }
84 }
```

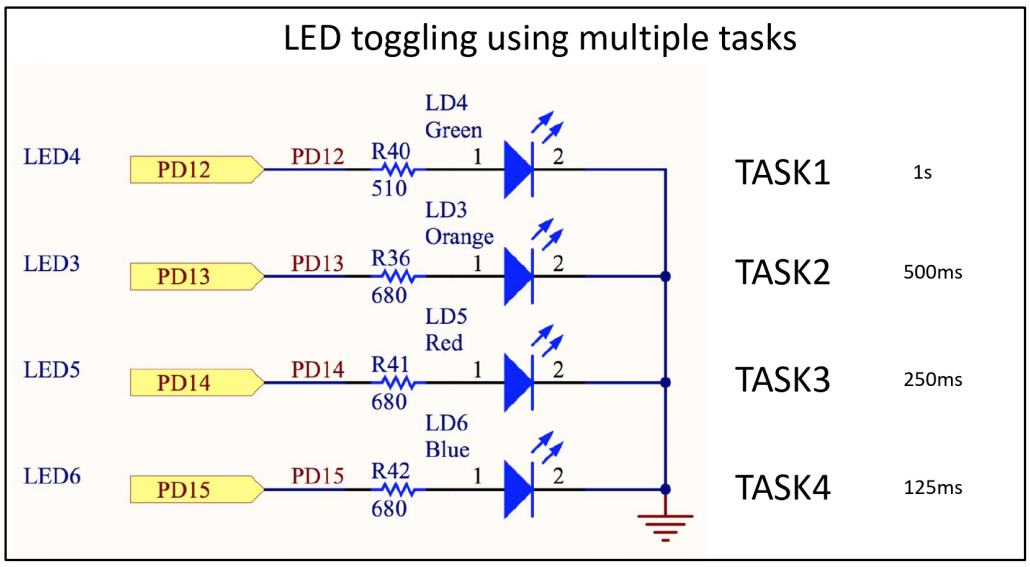
After task-3, task-4 gets executed

```
54 void task1_handler(void)
55 {
56     while(1)
57     {
58         printf("This is task-1\n");
59     }
60 }
```

After task-4, task-1 gets executed

So by our design, context switch is happening successfully in round robin fashion.

Toggling of LED using multiple tasks :



Led.c :

```c
#include<stdint.h>
#include "led.h"

void delay(uint32_t count)
{
  for(uint32_t i = 0 ; i < count ; i++);
}
void led_init_all(void)
{

    uint32_t *pRccAhb1enr = (uint32_t*)0x40023830;
    uint32_t *pGpiodModeReg = (uint32_t*)0x40020C00;


    *pRccAhb1enr |= ( 1 << 3);
    //configure LED_GREEN
    *pGpiodModeReg |= ( 1 << (2 * LED_GREEN));
    *pGpiodModeReg |= ( 1 << (2 * LED_ORANGE));
    *pGpiodModeReg |= ( 1 << (2 * LED_RED));
    *pGpiodModeReg |= ( 1 << (2 * LED_BLUE));

#if 0
    //configure the output type
    *pGpioOpTypeReg |= ( 1 << (2 * LED_GREEN));
    *pGpioOpTypeReg |= ( 1 << (2 * LED_ORANGE));
    *pGpioOpTypeReg |= ( 1 << (2 * LED_RED));
    *pGpioOpTypeReg |= ( 1 << (2 * LED_BLUE));
#endif

    led_off(LED_GREEN);
    led_off(LED_ORANGE);
    led_off(LED_RED);
    led_off(LED_BLUE);
}
void led_on(uint8_t led_no)
{
  uint32_t *pGpiodDataReg = (uint32_t*)0x40020C14;
  *pGpiodDataReg |= ( 1 << led_no);

}

void led_off(uint8_t led_no)
{
      uint32_t *pGpiodDataReg = (uint32_t*)0x40020C14;
      *pGpiodDataReg &= ~( 1 << led_no);

}
```

Initially all LED's will be OFF

Led.h :

```
#ifndef LED_H_
#define LED_H_

#define LED_GREEN  12
#define LED_ORANGE 13
#define LED_RED    14
#define LED_BLUE   15

#define DELAY_COUNT_1MS          1250U
#define DELAY_COUNT_1S          (1000U * DELAY_COUNT_1MS)
#define DELAY_COUNT_500MS       (500U  * DELAY_COUNT_1MS)
#define DELAY_COUNT_250MS       (250U  * DELAY_COUNT_1MS)
#define DELAY_COUNT_125MS       (125U  * DELAY_COUNT_1MS)

void led_init_all(void);
void led_on(uint8_t led_no);
void led_off(uint8_t led_no);
void delay(uint32_t count);

#endif /* LED_H_ */
```

Task handlers :

```
void task1_handler(void)
{
    while(1)
    {
        led_on(LED_GREEN);
        delay(DELAY_COUNT_1S);
        led_off(LED_GREEN);
        delay(DELAY_COUNT_1S);
    }
}

void task2_handler(void)
{
    while(1)
    {
        led_on(LED_ORANGE);
        delay(DELAY_COUNT_500MS);
        led_off(LED_ORANGE);
        delay(DELAY_COUNT_500MS);
    }
}
```

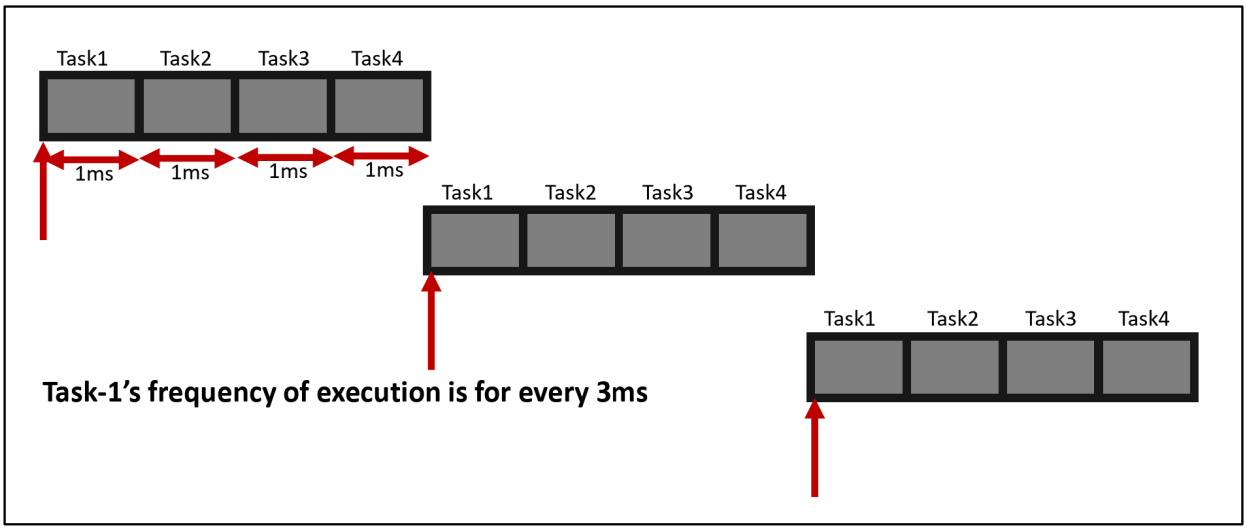
```
void task3_handler(void)
{
    while(1)
    {
        led_on(LED_BLUE);
        delay(DELAY_COUNT_250MS);
        led_off(LED_BLUE);
        delay(DELAY_COUNT_250MS);
    }
}

void task4_handler(void)
{
    while(1)
    {
        led_on(LED_RED);
        delay(DELAY_COUNT_125MS);
        led_off(LED_RED);
        delay(DELAY_COUNT_125MS);
    }
}
```

After compiling and downloading into our target, LED's are toggling successfully,

but they are not toggling the way that we wanted,

led_green is toggling for every 4s not for 1s, means delay is extended by 4 times,

Why is that ?



After turning ON led green at task-1, CPU is trapped in the for loop wasting cycles, so green LED will turn off after every tasks execution (3 ms),

so that only leg green is turning off and on for every 4s.

How to solve this problem ?

Solution is to introduce timer based delay, instead of software based delay,

task should not waste any CPU cycle, if they are done, they should leave the CPU.

Introducing blocking state for tasks :

- When a task has got nothing to do, it should simply call a delay function which should put the task into the blocked state from running state until the specified delay is elapsed
- We should now maintain 2 states for a task. Running and Blocked
- The scheduler should schedule only those tasks which are in Running state
- The scheduler also should unblock the blocked tasks if their blocking period is over and put them back to running state.

TCB(Task Control Block)

```
void init_systick_timer(uint32_t tick_hz);
__attribute__((naked)) void init_scheduler_stack(uint32_t sched_top_of_stack);
void init_tasks_stack(void);
void enable_processor_faults(void);
__attribute__((naked)) void switch_sp_to_psp(void);
uint32_t get_psp_value(void);

uint32_t psp_of_tasks[MAX_TASKS] = {T1_STACK_START,T2_STACK_START,T3_STACK_START,T4_STACK_START};

uint32_t task_handlers[MAX_TASKS];

uint8_t current_task = 0; //task1 is running
```

```
typedef struct{
    uint32_t psp_value;
    uint32_t block_count;
    uint8_t run_state;
    void (*task_handler)(void);
}TCB_t;
```

We are going to replace the global array for user tasks and psp value with a structure →

```
typedef struct
{
    uint32_t psp_value;
    uint32_t block_count;
    uint8_t current_state;
    void (*task_handler)(void);
}TCB_t;

TCB_t user_tasks[MAX_TASKS];
```

```
int main(void)
{
    enable_processor_faults();

    /*Initialize MSP*/
    init_scheduler_stack(SCHED_STACK_START);

    /*Stack initialization to store dummy frames*/
    init_tasks_stack();

    led_init_all();

    /*To generate systick timer exception*/
    init_systick_timer(TICK_HZ);

    /*Upto now SP is MSP*/
    /*Now we need to call user tasks, so PSP*/
    switch_sp_to_psp();

    task1_handler();

    for(;;);
}
```

→ Here itself we can do all the configurations regarding tasks

```c
#define TASK_RUNNING_STATE   0x00
#define TASK_BLOCKED_STATE   0xFF

void init_tasks_stack(void)
{
    // Initially keep all the task in running state
    user_tasks[0].current_state = TASK_RUNNING_STATE;
    user_tasks[1].current_state = TASK_RUNNING_STATE;
    user_tasks[2].current_state = TASK_RUNNING_STATE;
    user_tasks[3].current_state = TASK_RUNNING_STATE;


    // Initializing PSP value
    user_tasks[0].psp_value = T1_STACK_START;
    user_tasks[1].psp_value = T2_STACK_START;
    user_tasks[2].psp_value = T3_STACK_START;
    user_tasks[3].psp_value = T4_STACK_START;

    // Initializing task handler addresses
    user_tasks[0].task_handler = task1_handler;
    user_tasks[1].task_handler = task2_handler;
    user_tasks[2].task_handler = task3_handler;
    user_tasks[3].task_handler = task4_handler;
```

```c
for(uint32_t i = 0; i < MAX_TASKS; i++)
{
    pPSP = (uint32_t *)user_tasks[i].psp_value;

    pPSP--;                         // xPSR
    *pPSP = DUMMY_XPSR;             // 0x01000000

    pPSP--;                                 // PC
    *pPSP = user_tasks[i].task_handler;     // Address of task handler

    pPSP--;                         // LR
    *pPSP = 0xFFFFFFFD;             // EXC_RETURN

    for(uint32_t j = 0; j < 13; j++)
    {
        pPSP--;
        *pPSP = 0;
    }
    user_tasks[i].psp_value = (uint32_t)pPSP;        // Preserving PSP
}
```

```c
uint32_t get_psp_value(void)
{
    return user_tasks[current_task].psp_value;
}

void save_psp_value(uint32_t current_psp_value)
{
    user_tasks[current_task].psp_value = current_psp_value;
}
```

Blocking a task for given number of ticks :

- Let's introduce a function called "task_delay" which puts the calling task to the blocked state for a given number of ticks
- E.g., task_delay(1000); if a task calls this function then task_delay function puts the task into blocked state and allows the next task to run on the CPU
- Here, the number 1000 denotes a block period in terms of ticks, the task who calls this function is going to block for 1000 ticks (systick exceptions), i.e., for 1000ms since each tick happens for every 1ms.
- The scheduler should check elapsed block period of each blocked task and put them back to running state if the block period is over

Idle task

What if all the tasks are blocked? Who is going to run on the CPU?

We will use the idle task to run on the CPU if all the tasks are blocked. The idle task is like user tasks but only runs when all user tasks are blocked, and you can put the CPU to sleep.

Global tick count :

When a task can get blocked ?

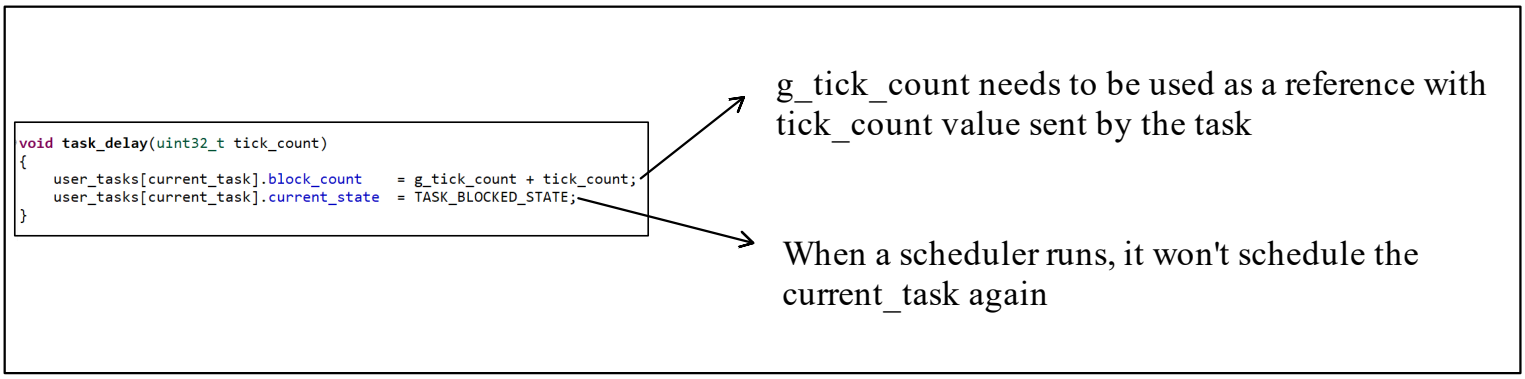
In RTOS or embedded OS, it can get blocked to wait for an event, get blocked calling the delay function, blocked over the semaphore,

In our case, task can get blocked only when it call task_delay routine.

So,

- How does the scheduler decide when to put the blocked state tasks (blocked using task_delay function) back to the running state ?
- It has to compare the task's delay tick count with a global tick count
- So, scheduler should maintain a global tick count and update it for every systick exception

```c
void task_delay(uint32_t tick_count)
{
    user_tasks[current_task].block_count    = g_tick_count + tick_count;
    user_tasks[current_task].current_state  = TASK_BLOCKED_STATE;
}
```




```c
void idle_task(void)
{
    while(1);
}
```

```c
#define MAX_TASKS       5
```

```c
void init_tasks_stack(void)
{
    // Initially keep all the task in running state
    user_tasks[0].current_state = TASK_RUNNING_STATE;
    user_tasks[1].current_state = TASK_RUNNING_STATE;
    user_tasks[2].current_state = TASK_RUNNING_STATE;
    user_tasks[3].current_state = TASK_RUNNING_STATE;
    user_tasks[4].current_state = TASK_RUNNING_STATE;


    // Initializing PSP value
    user_tasks[0].psp_value = IDLE_STACK_START;
    user_tasks[1].psp_value = T1_STACK_START;
    user_tasks[2].psp_value = T2_STACK_START;
    user_tasks[3].psp_value = T3_STACK_START;
    user_tasks[4].psp_value = T4_STACK_START;

    // Initializing task handler addresses
    user_tasks[0].task_handler = idle_task;
    user_tasks[1].task_handler = task1_handler;
    user_tasks[2].task_handler = task2_handler;
    user_tasks[3].task_handler = task3_handler;
    user_tasks[4].task_handler = task4_handler;
```

```c
#define T1_STACK_START              SRAM_END
#define T2_STACK_START              ( (SRAM_END) - (1 * SIZE_TASK_STACK) )
#define T3_STACK_START              ( (SRAM_END) - (2 * SIZE_TASK_STACK) )
#define T4_STACK_START              ( (SRAM_END) - (3 * SIZE_TASK_STACK) )
#define IDLE_STACK_START            ( (SRAM_END) - (4 * SIZE_TASK_STACK) )
#define SCHED_STACK_START           ( (SRAM_END) - (5 * SIZE_TASK_STACK) )
```

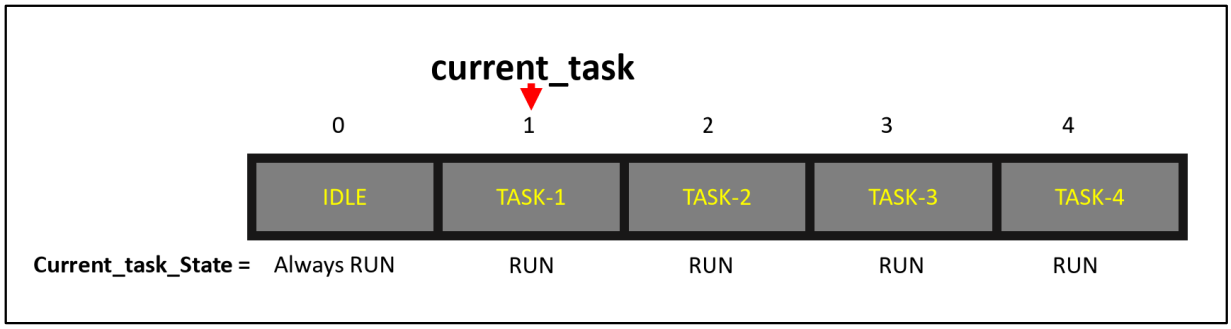
In the systick handler we have to update the global tick count.

```
uint8_t current_task  = 1;              // Task-1 is running;
uint32_t g_tick_count = 0;
```

Deciding next task to run :

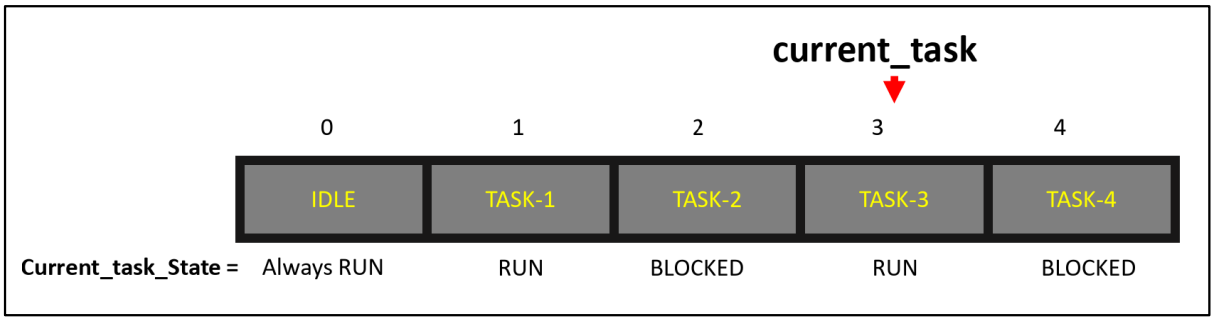
Deciding which task to run next depends on “state” of the next task.

Case 1 :



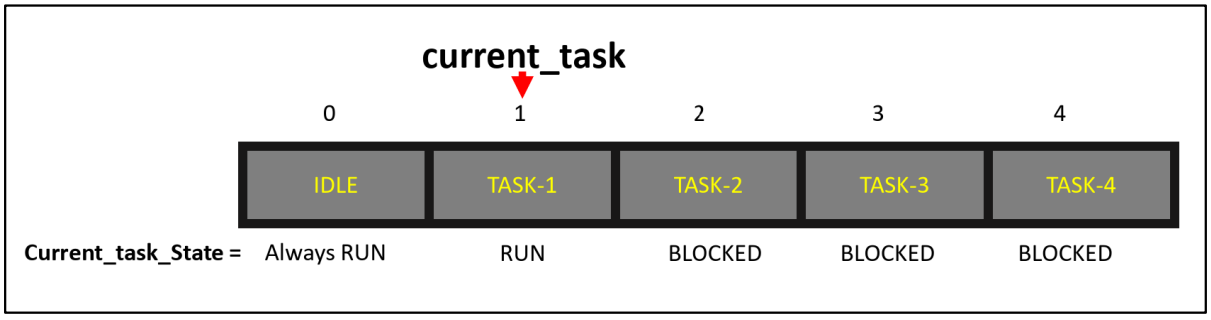
Current task is task-1, if exception happens, then control goes to Task-2 because its in the running state (not blocked state)

Case 2 :



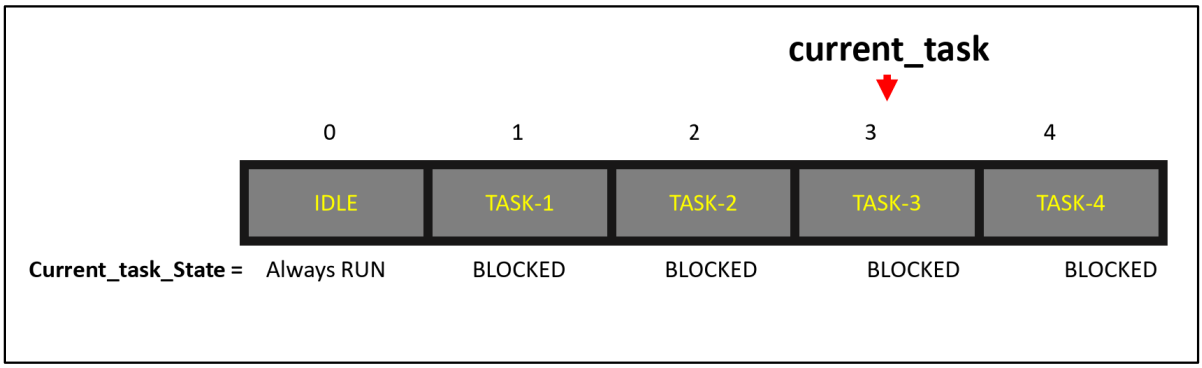
In this case, new task after task-3 is task-1 because task-4 is blocked, idle task should be ignored

Case-3 :



In this case, after exception control will come back to task-1 again.

Case-4 :



After exception, since all tasks are blocked you should decide idle task to run

In the interrupt noisy environment, pendSV is the better option to carry out the context switch,

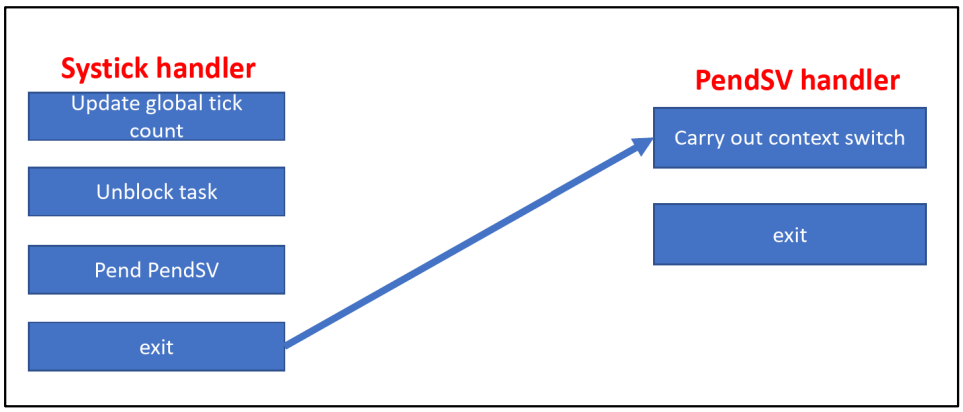
PendSV handler :

- We will use pendSV handler to carry out the context switch operation instead of systick handler

We will offload context switching work from systick to pendSV,

we will implement two handlers, pendSV handler as a naked function, systick handler as a normal C function,

systick will update the global tick count, unblock the task (if any task is qualified for running state from blocking it changes the state of a task), and then pend the pendSV, then exits



```c
void update_global_tick_count(void)
{
    g_tick_count++;
}

void SysTick_Handler(void)
{
    // 1. Update the global tick count
    update_global_tick_count();
    // 2. Unblock the tasks
    unblock_tasks();
}
```

Check any blocked task which can qualify for running state by comparing the block tick count / block period of the blocked task with a global tick count value.

Thats how we can decide whether a task can qualify from blocked state to running state

```c
void unblock_tasks(void)
{
    // Idle task is always in run mode
    for(uint32_t i = 1; i < MAX_TASKS; i++)
    {
        if(user_tasks[i].current_state != TASK_READY_STATE)
        {
            if(user_tasks[i].block_count == g_tick_count)
            {
                user_tasks[i].current_state = TASK_READY_STATE;
            }
        }
    }
}
```

This means that delay is elapsed

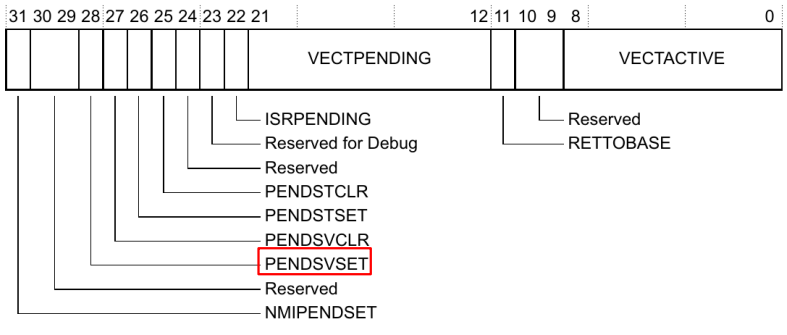
Now we have to pend the pendSV in systick handler,

4.3.3 Interrupt Control and State Register

The ICSR:

- provides:
  - a set-pending bit for the *Non-Maskable Interrupt* (NMI) exception
  - set-pending and clear-pending bits for the PendSV and SysTick exceptions
- indicates:
  - the exception number of the exception being processed
  - whether there are preempted active exceptions
  - the exception number of the highest priority pending exception
  - whether any interrupts are pending.

See the register summary in Table 4-12 on page 4-11, and the Type descriptions in Table 4-15, for the ICSR attributes. The bit assignments are:



[28]	PENDSVSET	RW	PendSV set-pending bit. Write: 0 = no effect 1 = changes PendSV exception state to pending. Read: 0 = PendSV exception is not pending 1 = PendSV exception is pending. Writing 1 to this bit is the only way to set the PendSV exception state to pending.

4.3 System control block

The *System Control Block* (SCB) provides system implementation information, and system control. This includes configuration, control, and reporting of the system exceptions. The system control block registers are:

Table 4-12 Summary of the system control block registers

Address	Name	Type	Required privilege	Reset value	Description
0xE000E008	ACTLR	RW	Privileged	0x00000000	*Auxiliary Control Register*
0xE000ED00	CPUID	RO	Privileged	0x410FC240	*CPUID Base Register* on page 4-13
0xE000ED04	ICSR	RW[a]	Privileged	0x00000000	*Interrupt Control and State Register* on page 4-13

```c
void SysTick_Handler(void)
{
    // 1. Update the global tick count
    update_global_tick_count();
    // 2. Unblock the tasks
    unblock_tasks();
    // 3. pend the pendSV exception
    uint32_t *pICSR = (uint32_t *)0xE000ED04;
    *pICSR = (1 << 28);
}
```

One more modification need to be done in task-delay function,

we should allow other tasks to run,

how to do that ? by triggering the pendSV

```c
void schedule(void)
{
    // pend the pendSV exception
    uint32_t *pICSR = (uint32_t *)0xE000ED04;
    *pICSR = (1 << 28);
}

void task_delay(uint32_t tick_count)
{
    if(current_task)
    {
        user_tasks[current_task].block_count    = g_tick_count + tick_count;
        user_tasks[current_task].current_state  = TASK_BLOCKED_STATE;
        schedule();
    }
}
```

Remember that you should block only user task,

if current task is 0, then it is idle task

Next modification need to be done in update_next_task( ) routine according to the case studies we have studied.

Update next task and testing :

```
void update_next_task(void)
{
    int state = TASK_BLOCKED_STATE;

    for(uint32_t i = 0; i < MAX_TASKS; i++)
    {
        current_task++;
        current_task %= MAX_TASKS;
        state = user_tasks[current_task].current_state;
        if( (state == TASK_READY_STATE) && (current_task != 0) )
        {
            break;
        }
    }

    if(state != TASK_READY_STATE)
    {
        current_task = 0;
    }
}
```

```
void update_next_task(void)
{
    int state = TASK_BLOCKED_STATE;

    for(uint32_t i = 0; i < MAX_TASKS; i++)
    {
        current_task++;
        current_task %= MAX_TASKS;
        state = user_tasks[current_task].current_state;
        if( (state == TASK_READY_STATE) && (current_task != 0) )
        {
            break;
        }
    }

    if(state != TASK_READY_STATE)
    {
        current_task = 0;
    }
}
```

Here we extract schedulable task, (ie.) task which is in ready state

If all the user task is in blocked state, state variable will not get modified, so default blocked state as per the intialization logic

We have used software delay :

```
void delay(uint32_t count)
{
  for(uint32_t i = 0 ; i < count ; i++);
}
```

Our goal is to use task_dealy routine which puts the task into blocked state after capturing the number of ticks for which the task wishes to get blocked and then schedules the next task

```
void task_delay(uint32_t tick_count)
{
    if(current_task)
    {
        user_tasks[current_task].block_count    = g_tick_count + tick_count;
        user_tasks[current_task].current_state  = TASK_BLOCKED_STATE;
        schedule();
    }
}
```

```c
void update_next_task(void)
{
    int state = TASK_BLOCKED_STATE;

    for(uint32_t i = 0; i < MAX_TASKS; i++)
    {
        current_task++;
        current_task %= MAX_TASKS;
        state = user_tasks[current_task].current_state;
        if( (state == TASK_READY_STATE) && (current_task != 0) )
        {
            break;
        }
    }

    if(state != TASK_READY_STATE)
    {
        current_task = 0;
    }
}
```

This will schedule the task in round robin fashion by getting the current state of a task.

```c
__attribute__((naked)) void PendSV_Handler(void)
{

    /*Save the context of current task*/

    // 1. Get current running task's PSP value
    __asm volatile("MRS R0,PSP");
    // 2. Using that PSP value to store SF2 (R4 - R11)
    __asm volatile("STMDB R0!,{R4-R11}");

    __asm volatile("PUSH {LR}");
    // 3. Save the current value of PSP
    __asm volatile("BL save_psp_value");


    /*Retrieve the context of next task*/

    // 1. Decide next task to run
    __asm volatile("BL update_next_task");
    // 2. Get its past PSP value
    __asm volatile("BL get_psp_value");
    // 3. Using that PSP value to retrieve SF2 (R4 - R11)
    __asm volatile("LDMIA R0!,{R4-R11}");
    // 4. Update PSP and exit
    __asm volatile("MSR PSP,R0");

    __asm volatile("POP {LR}");

    __asm volatile("BX LR");
}
```

pendSV handler will do the context switch

```c
void task1_handler(void)
{
    while(1)
    {
        led_on(LED_GREEN);
        task_delay(1000);
        led_off(LED_GREEN);
        task_delay(1000);
    }
}

void task2_handler(void)
{
    while(1)
    {
        led_on(LED_ORANGE);
        task_delay(500);
        led_off(LED_ORANGE);
        task_delay(500);
    }
}
```

```c
void task3_handler(void)
{
    while(1)
    {
        led_on(LED_BLUE);
        task_delay(250);
        led_off(LED_BLUE);
        task_delay(250);
    }
}

void task4_handler(void)
{
    while(1)
    {
        led_on(LED_RED);
        task_delay(125);
        led_off(LED_RED);
        task_delay(125);
    }
}
```

After compiling and downloading the code into the target, our goal of blinking led at the right timings is achieved successfully.

Note :

While accessing global variables, we need to be careful, because they can be accessed by both exception mode and thread mode code,

since exception handlers are asynchronous in nature, there may be chance of race condition

```
void task_delay(uint32_t tick_count)
{
    // Disable Interrupt

    if(current_task)
    {
        user_tasks[current_task].block_count    = g_tick_count + tick_count;
        user_tasks[current_task].current_state  = TASK_BLOCKED_STATE;
        schedule();
    }

    // Enable interrupt

}
```

In above we are accessing global variables,

if we want to disable all the interrupts it should be of global switch, meaning globally disabling all the interrupts,

for this purpose, we can make use of primask registers (special reg)

we can serialize the access to the global data by disabling all interrupts

Priority Mask Register

The PRIMASK register prevents activation of all exceptions with configurable priority. See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:

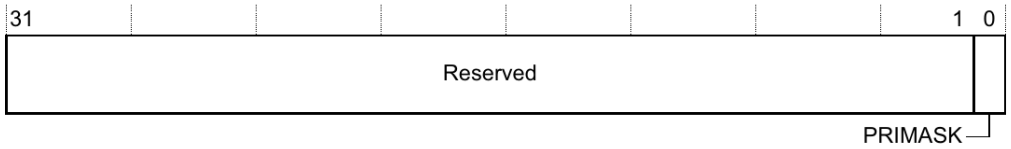


Table 2-7 PRIMASK register bit assignments

Bits	Name	Function
[31:1]	-	Reserved
[0]	PRIMASK	0 = no effect 1 = prevents the activation of all exceptions with configurable priority.

```c
#define INTERRUPT_DISABLE()  do{__asm volatile ("MOV R0,#0x01");  __asm volatile ("MSR PRIMASK,R0");}while(0)

#define INTERRUPT_ENABLE()  do{__asm volatile ("MOV R0,#0x00");  __asm volatile ("MSR PRIMASK,R0");}while(0)

void task_delay(uint32_t tick_count)
{
    // Disable Interrupt
    INTERRUPT_DISABLE();

    if(current_task)
    {
        user_tasks[current_task].block_count    = g_tick_count + tick_count;
        user_tasks[current_task].current_state  = TASK_BLOCKED_STATE;
        schedule();
    }

    // Enable interrupt
    INTERRUPT_ENABLE();
}
```

After compilation and downloading, our program works as expected