1.4 ARM Cortex-M7 exceptions and interrupts

In the terminology ARM uses, all events or conditions that can interrupt the normal program flow and transfer control to a specific handler (service) routine are referred to as exceptions. ARM Cortex-M7 processors support a variety of exceptions, and they are essential for handling events like interrupts, faults, and system calls. In general, exceptions can originate both by the hardware and the software.

1.4.1 ARM Cortex-M7 programmer's model

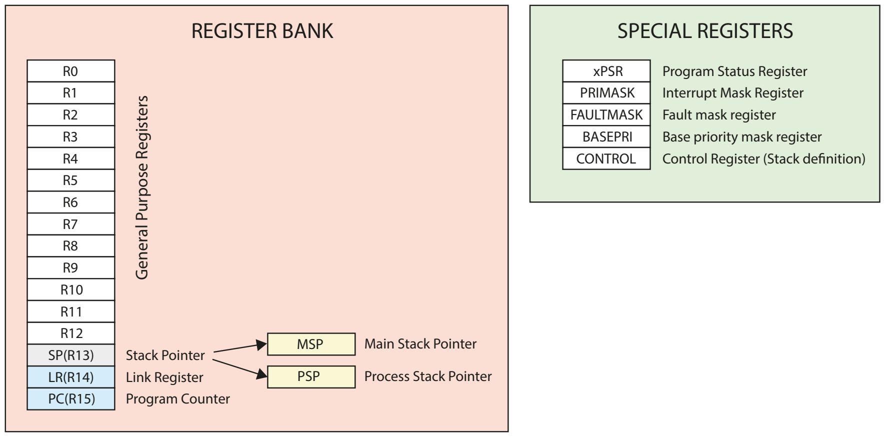


Fig. 1.10: ARM Cortex-M7 core registers.

In this subsection, we will briefly describe the ARM Cortex-M7 programmer's model. The ARM Cortex-M7 processor core features a set of registers used for various purposes in program execution and system control. These registers can be categorized into two groups: register bank and special registers (see Figure 1.10).

1.4.1.1 Register bank

The register bank contains 16 32-bit registers. Thirteen of them are general-purpose registers, and the other three have special uses:

1. Registers R0 to R12 are general-purpose registers for data storage and data operations.

2. R13 is Stack Pointer (SP) for maintaining the stack, typically used for local variables and function call frames. The Cortex-M7 contains two physically different stack pointers for different privilege levels:
   a. The Main Stack Pointer (MSP)is the default Stack Pointer after reset and is mainly used when the processor runs in privileged or system mode.
   b. The Process Stack Pointer (PSP) can only be used in unprivileged or user mode.
3. R14 is Link Register (LR), which stores the return address when calling subroutines or functions. On reset, the processor sets the LR value to 0xFFFFFFFF.
4. R15 is Program Counter (PC), which holds the memory address of the currently executing instruction.

Because the stack pointer register in ARM Cortex-M7 has two physical copies, we say it is **banked**. In the context of ARM Cortex processors, the term ’banked register’ refers to a type of register that has multiple copies or ’banks’, each associated with a specific execution mode or privilege level. These banks allow the processor to maintain separate register sets for different execution contexts, such as user mode, privileged mode, and exception modes. The selection of the stack pointer is determined by a special register called the CONTROL register, which is a part of the special register set.

1.4.1.2 Special registers

Besides the registers in the register bank, there are several special registers. These registers contain the processor status and define the operation states and interrupt/exception masking. The special registers are:

1. xPSR is a 32-bit Program Status Register. Some of the bit fields in the xPSR

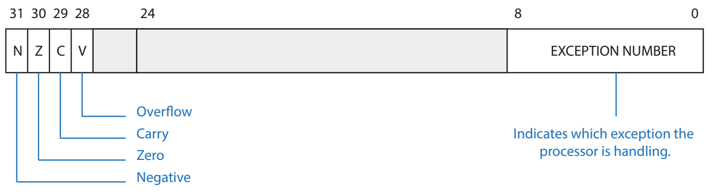


Fig. 1.11: xPSR register.

register are N (negative flag), Z (zero flag), V (overflow flag), C (carry flag), T (Thumb state) and EXCEPTION NUMBER representing the number of the current exception (interrupt).
2. CONTROL register is a 32-bit register that allows the processor to manage privileged and unprivileged execution modes and select the active stack pointer. It

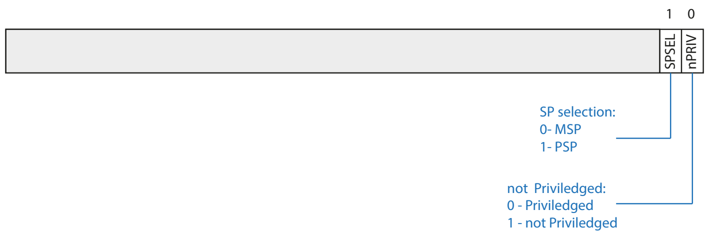


Fig. 1.12: CONTROL register.

includes the following fields: nPRIV (Privilege Level Bit) determines the privilege level of the processor (0 for privileged, 1 for unprivileged), and SPSEL (Stack Pointer Select Bit) selects the active stack pointer (0 for MSP, 1 for PSP).

3. Three exception masking registers:

   a.

4. The PRIMASK register is a 1-bit wide interrupt mask register. When set, it blocks all exceptions (including interrupts) apart from the Non-Maskable Interrupt (NMI) and the HardFault exception. The FAULTMASK register is very similar to PRIMASK, but it also blocks the HardFault exception. The BASEPRI register masks (blocks) exceptions or interrupts based on their priority level

Special registers are not memory mapped and can be accessed using special register access instructions MSR and MRS:
`MRS reg, special_reg`
reads special register into general-purpose register, and
`MSR special_reg, reg`
writes to special register from general-purpose register.

1.4.2 System Control Block

In addition to the registers we have just covered, ARM Cortex-M7 processors maintain another important register bank called System Control Block (SCB). The System Control Block is a crucial part of the processor's control and configuration. The SCB is a memory-mapped register bank that includes several registers and control bits that influence the processor's behaviour, manage exceptions, and provide system-level control. For example, the SCB registers for controlling processor configurations (e.g., low power modes), providing fault status information (fault status registers), relocating the vector table and controlling/obtaining the status of some interrupts. Here, we provide a brief description of only one CSB register related to interruptions and exceptions. This is the Interrupt Control and State Register (ICSR). This register provides bits for setting and clearing two software interrupts, PendSV and SysTick. The ICSR register is memory-mapped at address 0xE000ED04. For example, writing 1 to bit 28 in ICSR will set the PendSV exception to pending.

1.4.3 Exceptions

ARM architecture distinguishes between the two types of exceptions: **interrupts** originate from the external hardware, and **exceptions** originate from the CPU core or software (e.g., access to an invalid memory location or an SVC assembly instruction, which is commonly used as a convenient way to enter the operating system kernel). The following information identifies each ARM Cortex-M7 exception:

1. **Exception Number** - A unique number referencing a particular exception (starting at 1). This number is also used as the offset within the **vector table**, where the address of the handling routine for the exception is stored. This routine is usually referred to as the **exception handler** or **interrupt service routine (ISR)** and is the procedure which runs when an exception is triggered. The ARM hardware will automatically look up this function pointer (address of the exception handler) in the vector table when an exception is triggered and start executing the code. When the CPU is servicing an exception, its exception number is in the lower nine bits of the xPSR register.
2. **Priority Level / Priority Number** - Each exception has a priority associated with it. For most exceptions, this number is configurable. Counter-intuitively, the lower the priority number, the higher the precedence the exception has. So, for example, if two exceptions of priority level 2 and priority level 1 occur simultaneously, the exception with priority level 1 exception will be serviced first. When we say an exception has the “highest priority”, it will have the lowest priority number. If two exceptions have the same priority number, the exception with the lowest exception number will run first.
3. **Synchronous or Asynchronous** - As the name implies, some exceptions will fire immediately after an instruction is executed (e.g. SVCall). These exceptions are referred to as synchronous. Exceptions that do not fire immediately after a particular code path is executed are referred to as asynchronous (e.g. external interrupts).

ARM Cortex-M7 exceptions can be broadly categorised into four main types:

1. **Interrupts** are asynchronous events that can occur anytime and interrupt the normal program execution. They are typically generated by external peripherals (e.g., timers, UARTs, GPIO), and the processor responds to them by temporarily halting the current execution and transferring control to an interrupt service routine (ISR). For instance, a UART may use an interrupt request to indicate that new data have been received. A corresponding exception handler (ISR) is then executed that reads the received data. Interrupts can be divided into two main categories:

   a. **External Interrupts**: These are generated by external peripherals or devices to request the processor's attention. The Cortex-M7 processor supports a set of external interrupts (IRQs) that can be individually configured and prioritized.
   b. **NMI (Non-Maskable Interrupt)**: This is a special type of interrupt that has higher priority than regular interrupts and cannot be disabled or masked. NMIs are typically used for critical system functions. Like ordinary interrupt requests, Non-Maskable Interrupt (NMI) requests can be issued by either hardware or software (e.g. if errors happen in other exception handlers, an NMI will be triggered). The main difference is that their priority is extremely high, namely, the highest in the system below the reset exception.

   Two more exceptions also belong to this category and are generated within the processor rather than from external peripheral devices. They are:

   a. **SysTick** exception, generated periodically by the 24-bit count-down system timer and often used by operating systems to drive time slicing. If needed, the same exception can also be issued by software.
   b. **PendSV** exception can only be triggered by software. Operating systems often use it to indicate that a context switch is due and perform it in the future when no other exceptions are waiting to be handled. The PendSV exception can be triggered by writing 1 to bit 28 in the ICSR (a part of the System Control block), which is memory-mapped at address 0xE000ED04.

2. **Faults** are synchronous events generated due to an abnormal event detected by the processor, either internally or while communicating with memory and other devices. These exceptions are of great interest and concern because they indicate serious hardware or software issues that likely prevent the software itself from continuing with normal activities. The following faults are present in Cortex-M7 processors:

   a. **UsageFault** occurs when the processor detects an issue with the program's execution or when an instruction cannot be executed for various reasons. For instance, the instruction may be undefined or may contain a misaligned address that prevents it from accessing memory correctly. Another reason for raising a UsageFault exception is an attempt to divide by zero. Some of the faults mentioned above (like dividing by zero) can be masked in software, i.e., the processor can be instructed to just ignore them without generating any exception, whereas others (such as undefined instruction) cannot, for obvious reasons.
   b. **BusFault** triggers when an error occurs on the data or instruction bus while accessing memory. In other words, it can be generated as a consequence of an explicit memory access performed by an instruction during its execution and also by fetching an instruction from memory. BusFaults result from issues in memory access, most often as attempting to access a location with no valid memory. As Cortex-M7 is a memory-mapped input-output (I/O) architecture, whenever we refer to a memory address, we actually mean

an address within the processor's address space that may refer to either a memory location or an I/O register.

c. **MemManage** (Memory Management Fault) faults occur when there is a memory access violation, such as accessing restricted memory regions. In other words, this fault occurs when the memory protection mechanism blocks memory access. An optional Memory Protection Unit (MPU) provides a programmable way of protecting memory regions against data read and write operations, as well as instruction fetches. For instance, the processor's MPU can be programmed to forbid instruction fetch from address areas containing I/O registers.
d. **HardFault** is a severe fault that can be generated when an error occurs during exception processing, thus disrupting the normal exception handling flow. HardFaults have a higher priority than any exception with configurable priority. HardFaults are typically unrecoverable, meaning the processor cannot continue the normal program execution from the point of the fault. Usually, the application or CPU must be reset. To prevent HardFaults, developers should follow best practices for writing robust and well-tested code. This includes avoiding undefined instructions, ensuring valid memory accesses, and monitoring stack usage to prevent stack overflows. Additionally, proper fault handling and diagnostics can help identify and address issues before they lead to a HardFault. Hard faults in Cortex-M7 processors are a critical part of system reliability and safety, as they help detect and report severe issues that could otherwise result in unpredictable or incorrect system behaviour.

3. **Supervisor call (SVC)** is a software-initiated exception. It is used to transition from the user or application mode to a more privileged mode, typically for making requests to the operating system or kernel. The execution of an SVC assembly instruction raises this exception. It is commonly used as a convenient way to enter the operating system kernel and request it to perform a function on behalf of the application.
4. **Reset Exception (Reset)** is invoked on power up or a warm reset. The exception model treats reset as a special form of exception. When reset is asserted, the operation of the processor stops, potentially at any point in an instruction. When reset is de-asserted, execution restarts from the address provided by the reset entry in the vector table. It is handled as other exceptions for the most part, except that instruction execution can stop at an arbitrary point.

1.4.4 Exception numbers and priorities

Table 1.4 lists different types of exceptions with their priorities, exception numbers and vector addresses. All exceptions have an associated priority with a lower number value indicating a higher priority. The programmer (software) configures the

Table 1.4: Exception types in Cortex-M7.

Exception Number	**Exception Type**	**Priority**	**Vector Address**	**Activation**
1	Reset	-3 (Highest)	0x00000004	Asynchronous
2	NMI	-2	0x00000008	Asynchronous
3	HardFault	-1	0x0000000C	Synchronous
4	MemManage	Configurable	0x00000010	Synchronous
5	BusFault	Configurable	0x00000014	Synchronous
6	UsageFault	Configurable	0x00000018	Synchronous
7-10	*unused*	-	-	-
11	SVCall	Configurable	0x0000002C	Synchronous
12-13	*unused*	-	-	-
14	PendSV	Configurable	0x00000038	Asynchronous
15	SysTick	Configurable	0x0000003C	Asynchronous
16 and above	Interrupt (IRQ)	Configurable	0x00000040 and above	Asynchronous

priorities for most exceptions, except for Reset, NMI and HardFault. If the software does not configure any priorities, then all exceptions with a configurable priority have a priority of 0. Configurable priority values are in the range 0-15. Here is the rule of **order of execution** of exceptions:

1. If two or more exceptions are pending, the exception with the highest priority runs first.
2. If two or more exceptions with the same priority are pending, the exception with the lowest exception number runs first.
3. When the processor executes an exception handler, the exception handler is preempted if a higher-priority exception occurs. If an exception occurs with the same priority as the exception being handled, the handler is not preempted, irrespective of the exception number. However, the status of the new interrupt remains pending.

The exceptions with exception numbers 1-15 are so-called **built-in exceptions**. The built-in exceptions are a mandatory part of every ARM Cortex-M core. The ARM Cortex-M specifications reserve exception numbers 1-15, inclusive, for built-in exceptions.

1.4.5 Vector table and Exception handlers

The vector table contains the reset value of the stack pointer and the start addresses, also called **exception vectors**, for all exception handlers. On system reset, the vector table is at address 0x00000000. This is the default start address of the vector

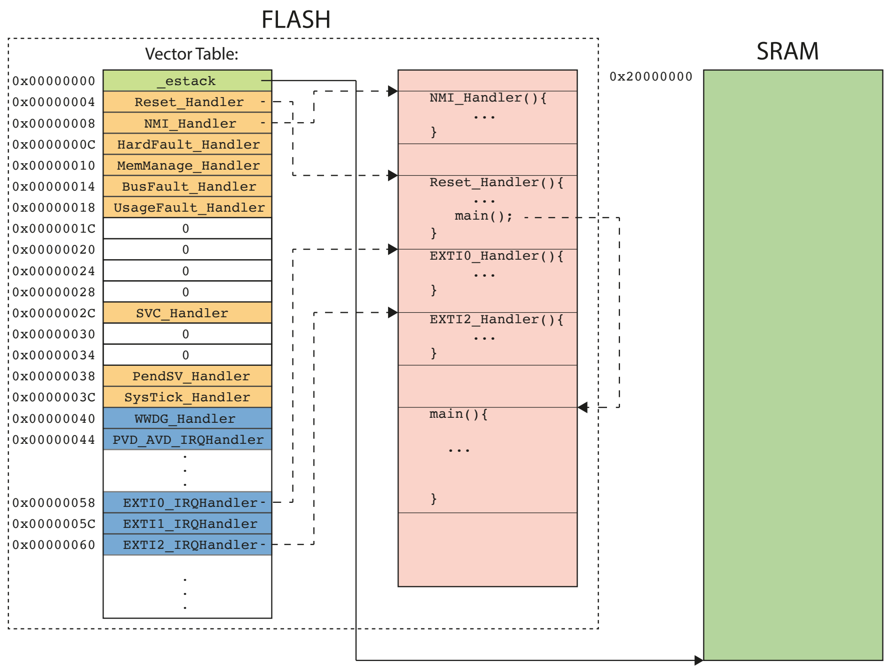

FLASH
Vector Table:
0x00000000 _estack
0x00000004 Reset_Handler
0x00000008 NMI_Handler
0x0000000C HardFault_Handler
0x00000010 MemManage_Handler
0x00000014 BusFault_Handler
0x00000018 UsageFault_Handler
0x0000001C 0
0x00000020 0
0x00000024 0
0x00000028 0
0x0000002C SVC_Handler
0x00000030 0
0x00000034 0
0x00000038 PendSV_Handler
0x0000003C SysTick_Handler
0x00000040 WWDG_Handler
0x00000044 PVD_AVD_IRQHandler
0x00000058 EXTI0_IRQHandler
0x0000005C EXTI1_IRQHandler
0x00000060 EXTI2_IRQHandler
NMI_Handler(){ ... }
Reset_Handler(){ ... main(); }
EXTI0_Handler(){ ... }
EXTI2_Handler(){ ... }
main(){ ... }
0x20000000
SRAM

Fig. 1.13: The memory layout of the vector table and exception handlers in ARM Cortex-M7 cores.

table, where Cortex-M7 expects to find it. This is usually a linker job that places the vector table at the beginning of the binary file we upload to the flash memory. Figure 1.13 shows how the vector table is organized in memory and the order of the exception vectors in the vector table. The first entry of this array is the value of the stack pointer. Note that the programmer is responsible for setting the first value into the stack pointer (which is the address of the beginning of the stack). Usually, this address corresponds to the end of the SRAM, as we often use the stack that expands in the direction of descending addresses. Starting from the second entry of this table, we can find the starting addresses for all exception handlers. This means that the vector table has a length of up to 256 for Cortex-7 and depends on the number of interrupts implemented. The silicon vendor that uses an ARM Cortex-M7 core can implement up to 240 interrupts. The silicon vendor must configure the top range value, which is dependent on the number of interrupts implemented. ARM requires that we always adjust the vector table's size by rounding up to the next power of two. For example, if there are 16 interrupts, the minimum size of the vector table is 32 words, enough for 16 built-in exceptions and up to 16 interrupts. If the user (silicon vendor) requires 21 interrupts, the size of the vector table must be 64 words because the required table size is 37 words, and the next power of two is 64. The name of the exception handlers in Figure 1.13 is just a convention, and we are totally free to rename them if we like a different one. They are just symbols.

Defining a vector table for a Cortex-M7 processor involves setting up a table of exception handler addresses that the processor will jump to when specific exceptions occur. As said before, the vector table must be placed at the beginning of the flash memory, where the processor expects to find it. In ARM Cortex-M microcontroller development, the `.isr_vector` is a special section in the microcontroller's memory where the vector table for exceptions and interrupts is defined. The vector table contains addresses of exception and interrupt service routines (ISRs). The `.isr_vector` section is a label used in the linker script to specify the location of the vector table in memory. Commonly, the vector table is implemented in assembly code in the startup file (e.g. for the Cortex-M7-based STM32H753 microcontroller, the startup file would be startup_stm32h753xx.s) as:

```
.section  .isr_vector

g_pfnVectors:
  .word  _estack
  /* Built-in Exceptions */
  .word  Reset_Handler
  .word  NMI_Handler
  .word  HardFault_Handler
  .word  MemManage_Handler
  .word  BusFault_Handler
  .word  UsageFault_Handler
  .word  0
  .word  0
  .word  0
  .word  0
  .word  SVC_Handler
  .word  DebugMon_Handler
  .word  0
  .word  PendSV_Handler
  .word  SysTick_Handler
  /* External Interrupts */
  .word  WWDG_IRQHandler
  .word  PVD_AVD_IRQHandler
   ...
  .word  EXTI0_IRQHandler
  .word  EXTI1_IRQHandler
  .word  EXTI2_IRQHandler
   ...
  .word  WAKEUP_PIN_IRQHandler
```

Listing 1.9: The vector table for Cortex-M7.

Then, the exception and interrupt handler functions should be implemented in the code. These functions are called when their corresponding exceptions or interrupts occur. The handler function names should match the names of the entries in the vector table for a very obvious reason:

```
void Reset_Handler(void) {
    // Reset handler code
}

void NMI_Handler(void) {
    // NMI handler code
}

void HardFault_Handler(void) {
    // HardFault handler code
}

void EXTI0_IRQHandler (void) {
    // HardFault handler code
}
```

Listing 1.10: Exception handlers in C.

1.4.6 Exception entry and exit

Exception entry and exit in an ARM Cortex-M7 processor is a well-defined process that enables the CPU to handle various exceptions, including interrupts and faults while preserving the state of the currently executing program. This mechanism ensures that the system can respond to events without compromising the integrity of the application code. Here, we provide a detailed description of the exception entry and exit process in a Cortex-M7.

1.4.6.1 Exception entry

The **exception entry** occurs when there is a pending exception with sufficient priority and either:

1. The processor is executing a normal program and the new exception terminates the currently executing program.
2. The processor executes the exception handler, and the new exception is of higher priority than the exception being handled, in which case the new exception preempts the original exception. When one exception preempts another, we say the exceptions are **nested**.

When the processor takes an exception, the processor pushes the **current execution context** onto the current stack. The execution context consists of eight 32-bit words: registers R0 through R3, R12, the link register LR (also accessible as R14), the program counter PC (R15), and the program status register xPSR, for a total of 32 bytes. This operation is referred to as **stacking**, and the structure of eight 32-bit data words is referred to as the **stack frame**. The reason behind automatically saving the execution context is that accepting and handling an exception should not necessarily prevent the processor from returning to its current activity later. This is particularly true for interrupts and other exception requests that occur asynchronously to current processor activities and are most often totally unrelated to them. Thus, the exceptions and interrupts should be transparent with respect to any code executing when they arrive. Figure 1.14 shows the exception stack frame after stacking. Immediately after stacking, the stack pointer indicates the lowest address in the stack frame. The reader will notice that Cortex-M processors use the **full-descending stack** (the stack grows downward in memory, and the stack pointer points to the lowest memory address in use). The stack frame includes the return

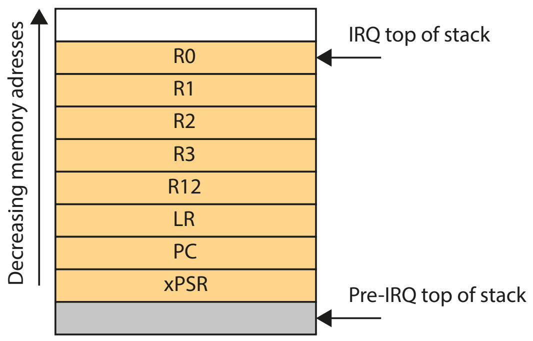


Fig. 1.14: The layout of the stack frame after stacking in ARM Cortex-M7.

address, as the PC is also saved during stacking. This is the address of the next instruction in the interrupted program. This value is restored to the PC at exception return so that the interrupted program resumes.

Here, we have to describe stack pointers and processing modes in ARM Cortex-M processors in more detail. In ARM Cortex-M processors, there are two registers used to access and manipulate stack: the **Main Stack Pointer (MSP)** and the **Process Stack Pointer (PSP)**. These stack pointers are critical in managing the execution context and handling exceptions in the processor. Additionally, the Cortex-M architecture defines two processing modes: **Thread mode** and **Handler mode**, each with distinct purposes and behaviours. The Main Stack Pointer (MSP) and Process Stack Pointer (PSP) can be accessed and manipulated through the stack pointer (SP), also known as register r13. Commonly, operating mode defines which of the two (MSP or PSP) is accessible through SP (i.e. visible as SP).

Thread mode is the typical execution mode for user/application code. The processor often uses the PSP (although it is possible to use MSP in this mode also) as the current stack pointer in this mode. The processor enters Thread mode after a reset or when returning from an exception or interrupt. User-level code runs in Thread mode, and the PSP is often used for function calls and managing thread-specific context. Handler mode is a privileged execution mode used for handling exceptions and interrupts. The processor switches from Thread mode to Handler mode when an exception or interrupt occurs. The processor automatically saves the current context onto the PSP or MSP stack (depending on the operation mode of the interrupted program) before executing the exception handler. The MSP is then used in Handler mode as the stack pointer. Handler mode is reserved for system-level tasks and ensures that critical operations can be carried out even when the application stack is compromised.

In parallel to the stacking operation, the processor writes an **exception return value** (called EXC_RETURN value in the ARM documentation) to the link register (LR). This indicates which stack pointer corresponds to the stack frame and what operation mode the processor was in before the entry occurred. The information provided by the EXEC_RETURN value allows the processor to locate the stack frame to be restored upon returning from an exception, interpret it in the right way, and bring back the processor to the execution mode of the interrupted program. Table 1.5 shows the EXC_RETURN values and their meaning upon returning from an exception.

Table 1.5: Exception return values and their behaviour upon returning from an exception.

EXC_RETURN[31:0]	**Description**
0xFFFFFFF1	Return to Handler mode, exception return uses the exception stack frame from the MSP and execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode, exception return uses the exception stack frame from the MSP and execution uses MSP after return.
0xFFFFFFFD	Return to Thread mode, exception return uses the exception stack frame from the PSP and execution uses PSP after return.

In parallel to the stacking operation, the processor also performs a vector fetch that reads the exception handler start address from the vector table. The processors determines the exception vector to be fetched into the PC by the exception number:

$$\text{PC} \leftarrow \text{M}[\texttt{0x0000\ 0000} + 4 \times (\text{exception\ number})].$$

When stacking is complete, the processor starts executing the exception handler, switching to Handler Mode. Associated with the execution mode switch, the processor may also use a new stack. As mentioned previously, handler mode execution always uses MSP, whereas thread mode execution may use either MSP or PSP, depending on processor configuration. The Reset exception is a deviation from this general rule. The Reset exception is handled in Thread mode instead. Upon reset, execution starts in Thread mode, and the processor is automatically configured to use MSP.

1.4.6.2 Exception return

The **exception return** occurs when the processor is in Handler mode and executes an instruction which loads the EXC_RETURN value into the PC (for example `bx lr`). Recall that EXC_RETURN is the value loaded into the LR on exception entry. The exception mechanism relies on this value to detect when the processor has completed an exception handler. The lowest bits of this value provide information on the return stack and processor mode. When this value is loaded into the PC, it indicates to the processor that the exception is complete, and the processor should initiate the appropriate exception return sequence instead of fetching an instruction.

When an exception return value is loaded into the program counter PC as part of an exception handler epilogue, it directs the processor to initiate an exception handler return sequence instead of simply returning to the caller. In fact, the ARM Architecture Procedure Calling Standard (AAPCS) states that a function call must save into the link register LR the return address before setting the program counter PC to the function entry point. This is typically accomplished by executing a branch and link instruction `bl` with a PC-relative target address. In the epilogue of the called function, it is then possible to return to the caller by storing back into PC the value stored into LR at the time of the call. This can be done, for instance, by means of a branch and exchange instruction `bx`, using LR as argument.

This aspect of the exception return has been architected to permit any AAPCS-compliant function to be used directly as an exception handler. In this way, any AAPCS-compliant function can be used as an exception handler. This is especially important when exception handlers are written in a high-level language like C because compilers are able to generate AAPCS-compliant code by default, and hence, they can also generate exception-handling code without treating it as a special case. The exception handlers for ARM Cortex-M processors are thus implemented as regular C functions and do not require a special function declaration keyword. As a result, an exception handler return performed by hardware is indistinguishable from a regular software-managed function return.

The following code presents the exception handler for an exception triggered by GPIO Pin 13 through EXTI15_10 lines. The exception handler is implemented just as a regular C function without any special function declaration:

```c
void EXTI15_10_IRQHandler(void)
{
  // Check if GPIO_PIN_13 triggered the interrupt:
  if (__HAL_GPIO_EXTI_GET_IT(GPIO_PIN_13) != 0x00U)
  {
    // Your code to handle the GPIO_PIN_13 interrupt goes here

    // Clear the GPIO_PIN_13 interrupt flag
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_13);
  }
}
```

Listing 1.11: The exception handler for EXTI15_10 interrupt implemented as a regular C function.

1.4.7 Case Study: A simple task scheduler on ARM Cortex-M7

In the realm of computer systems and real-time operating systems (RTOS), the concept of context switching is the linchpin of multitasking and responsiveness. It's a finely tuned mechanism that orchestrates the efficient execution of multiple tasks, allowing a processor to handle numerous concurrent operations with precision and determinism. At its core, **context switching is a process by which the processor transitions from executing one task to another**. **The context of each task includes the task's state of the processor—registers, program counter, stack pointer, and system variables**. This transition involves the preservation of the current task's context, the loading of the new task's context, and the seamless continuation of the latter's execution.

Context switching begins with a trigger—typically a timer interrupt signalling the need to switch contexts. The processor diligently saves the current context onto a task's stack and retrieves the context of the next task to be executed from its stack. An RTOS relies on a task scheduler, interrupt handling mechanisms, and precise memory management to orchestrate this performance. The scheduler keeps a record of tasks and manages their execution, while the interrupt system plays a pivotal role in triggering context switches when a timer interrupt occurs.

Understanding the intricacies of context switching is paramount for engineers working with computer systems to create efficient, deterministic, and robust applications. So, let's raise the curtain and delve into the intricacies of context switching, where the processor seamlessly switches tasks, and the computer system transforms into a multitasking maestro.

1.4.7.1 Background

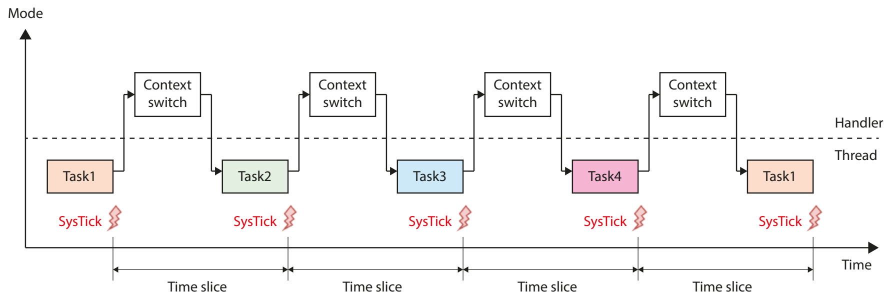


Fig. 1.15: A simple task scheduler.

A simple round-robin task scheduler (Figure 1.15) on Cortex-M7 processors effectively manages multiple tasks or threads in a cooperative multitasking environment. In this scheduler, each task is given a fixed time slice (quantum) during which it can execute. When its time slice expires, the scheduler switches to the next task in the queue. **The task scheduler relies on the interrupts and stacks to achieve context switching**. The SysTick and PendSV interrupts can both be used for context switching. The SysTick peripheral is a 24-bit timer that interrupts the processor each time it counts down to zero. This makes it well-suited to round-robin style context switching, and we are going to use the SysTick to perform a context switch.

When switching contexts, the scheduler needs a way to keep track of which tasks are doing what using a task table. Recall from the previous sections that the ARM Cortex-M7 processor has two separate stack pointers which can be accessed through a banked SP register: Main Stack Pointer (MSP), which is the default one after startup and is used in exception handlers running in the Handler mode, and Process Stack Pointer (PSP), which is often used in regular user procedures running in the Thread mode. In our application, tasks run in the Thread Mode with PSP, and the context-switcher (kernel) runs in the Handler Mode with MSP. This allows stack separation between the kernel and tasks (which simplifies the context switch procedure) and prevents tasks from accessing important registers and affecting the kernel.

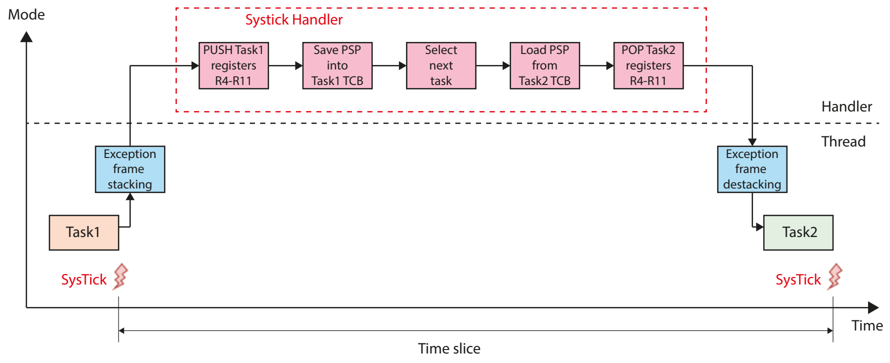


Fig. 1.16: A simple task scheduler.

Figure 1.16 shows the scheduler operations during a context switch in more detail. The scheduler relies on exception entry and exit mechanisms, which automatically save and restore the critical CPU context (registers R0-R3, R12, LR, PC and xPSR) using the exception frame on the stack. When a SysTich exception occurs, the Task1 critical registers are automatically saved into the Task1 exception stack frame. Once in the Systick handler, the scheduler is responsible for pushing the interrupted task Task1 registers R4-R11 onto the task's stack and saving its PSP in the task's TCB. Then, the scheduler selects the next task (Task2) in a round-robin fashion. Before returning from the SysTick handler, the scheduler is responsible for loading the Task2 SP into the PSP register and restoring the Task2 registers R4-

R11 from the Task2 stack. Then, upon exception exit, the Task2 critical registers are restored from its exception stack frame, and the execution returns to the new task.

Usually, three routines are required to implement and run the scheduler: create new tasks, initialize tasks, and perform the context switch. Besides, several data structures are required to implement and manage the stack for each task and represent each task's state. In the following subsections, we provide a step-by-step description of implementing a very simple round-robin scheduler on a Cortex-M7 processor.

1.4.7.2 Tasks

A task is a piece of code or a function that does a specific job when it is allowed to run. Usually, a task is an infinite loop which can repeatedly do multiple steps. In our simple scheduler application, the tasks cannot be finished (they never return) and do not take any arguments. Here is a C implementation of a task:

```c
void task() {
    // init task:
    ...
    // main loop
    while(1) {
        // do things over and over
    }
}
```

Listing 1.12: A task in C. In our application, a task never returns and does not take any arguments.

1.4.7.3 Stacks

In a multitasking environment, where multiple tasks are executed in a time-sharing manner, each task needs to have its own stack. Each task executes within its own context with no coincidental dependency on other tasks within the system or the scheduler itself. Each task's stack provides isolation between tasks. It ensures that local variables and function call frames of one task do not interfere with those of another task. This isolation is crucial for maintaining data integrity and preventing unintended side effects between tasks. Only one task within the application can execute at any point in time, and the scheduler is responsible for deciding which task this should be. As a task does not know of the scheduler activity, it is the scheduler's responsibility to ensure that the processor context (register values, stack contents, etc.) when a task is swapped in is exactly the same as when the same task was swapped out. In other words, each task's stack allows tasks to be reentrant. Reentrancy means that a task can be interrupted while executing and later resume from where it left off without corrupting its state. The stack stores the task's execution context, enabling reentrant behaviour. Besides, each task should be able to make function calls and put arguments on the stack without worrying about function call frames interfering with those of other tasks. Furthermore, allocating a fixed amount of stack space for each task makes it easier to predict memory usage and stack requirements for each task, simplifying system design and analysis.

To achieve this, **each task is provided with its own stack** in our simple task scheduler. The size of each task's stack is 1 kB (256 32-bit words). So, for four tasks we create a memory block that holds all four stacks as follows:

```
unsigned int stackRegion[NTASKS * TASK_STACK_SIZE];
```

Listing 1.13: Memory block for tasks' stacks. NTASKS equals 4 and TASK_STACK_SIZE equals 256.

1.4.7.4 Task control block

A Task Control Block (TCB), also known as a Task Control Structure (TCS), is a data structure used in real-time operating systems (RTOS) and multitasking environments to manage and control individual tasks or threads. The TCB holds essential information about a task's state, allowing the operating system or scheduler to manage and switch between tasks efficiently. The exact contents and structure of a TCB may vary depending on the operating system or RTOS, but it typically includes the following information: task identifier, task state (e.g., ready to run, blocked, suspended, etc.), task priority, stack pointer, task name, and additional task's parameters.

In our implementation, each task will always be ready to run, so we will omit the task state from TCB. Besides, all tasks in our scheduler will have the same priority and will be selected on a round-robin basis, so we will omit the task priority from TCB. Because each task should have its own stack to save its local variable and exception frame, our TCB must include the SP value, which points to the current stack pointer of the task. The scheduler will select the next task in a round-robin fashion and write its SP value into the PSP register. The scheduler will also copy the PSP register of the interrupted task into its SP value. Also, in our implementation, the Task Control Block will contain the start address of the task. Here is a minimal TCB implementation using struct in C:

```
typedef struct{
  unsigned int *sp;
  void (*pTaskFunction)();
} TCB_Type;
```

Listing 1.14: TCB structure.

In our simple implementation, our scheduler will contain only four tasks. It would be easy to add additional tasks later, but for now, we will keep the code as simple as possible. Each of the four tasks should have its TCB. Hence, we create a TCB table as:

```
TCB_Type TCB[NTASKS];
```

Listing 1.15: TCB table. NTASKS is a constant equal to 4.

1.4.7.5 Task creation

The following code presents the function used to create a new task:

```
void CreateTask(TCB_Type* pTCB, unsigned int* pStackBase,
                void (*TaskFunction)()) {
  HWSF_Type* pHWStackFrame;
  SWSF_Type* pSWStackFrame;


  // set pointers to top of HWSF and SWSF:
  pHWStackFrame = (HWSF_Type*)((void*)pStackBase-sizeof(HWSF_Type));
  pSWStackFrame = (SWSF_Type*)((void*)pHWStackFrame-sizeof(SWSF_Type));


  // populate HW Stack Frame
  pHWStackFrame->r0   = 0x0;
  pHWStackFrame->r1   = 0x0;
  pHWStackFrame->r2   = 0x0;
  pHWStackFrame->r3   = 0x0;
  pHWStackFrame->r12  = 0x0;
  pHWStackFrame->lr   = 0xFFFFFFFF; // (reset val)
  pHWStackFrame->pc   = (unsigned int)TaskFunction;
  pHWStackFrame->psr  = 0x01000000; // Set T bit (bit 24) in EPSR.
                                     // The Cortex-M4 processor
                                     // only supports execution of
                                     // instructions in Thumb state.
  // populate SW Stack Frame
  pSWStackFrame->r4   = 0x0;
  pSWStackFrame->r5   = 0x0;
  pSWStackFrame->r6   = 0x0;
  pSWStackFrame->r7   = 0x0;
  pSWStackFrame->r8   = 0x0;
  pSWStackFrame->r9   = 0x0;
  pSWStackFrame->r10  = 0x0;
  pSWStackFrame->r11  = 0x0;

  // Set SP and task's address in task's TCB:
  pTCB->sp             = (unsigned int*)pSWStackFrame;
  pTCB->pTaskFunction  = TaskFunction;
}
```

Listing 1.16: The function `CreateTask()` that creates a new task.

The parameters of the above `CreateTask()` function are:

- `pTCB` - a pointer to a task's TCB,
- `pStackBase` - pointer task's stack block,
- `TaskFunction` - address of a task's function.

The `CreateTask()` function performs the following steps:

1. Initialize pointers to two stack frames that hold the exception stack frame (so-called hardware stack frame) and the so-called software stack frame. The hardware stack frame will hold eight registers saved by the CPU during exception entry. Besides these eight registers, we need to save the remaining eight registers from the task's context (R4-R11). We need to prepare these stack frames for each new task so that when the task switch occurs, both frames will be ready for de-stacking and, hence, entering a new task. To make this task easier, we will abstract the frames with two structures:

```
typedef struct{
  unsigned int r0;
  unsigned int r1;
  unsigned int r2;
  unsigned int r3;
  unsigned int r12;
  unsigned int lr;
  unsigned int pc;
  unsigned int psr;
} HWSF_Type;

typedef struct{
  unsigned int r4;
  unsigned int r5;
  unsigned int r6;
  unsigned int r7;
  unsigned int r8;
  unsigned int r9;
  unsigned int r10;
  unsigned int r11;
} SWSF_Type;
```

Listing 1.17: Structures used to abstract the hardware and software stack frames.

The hardware stack frame resides at the bottom of the task's stack, and the software stack frame resides above the hardware stack frame.

2. Now, as two pointers to stack frames, `pHWStackFrame` and `pHWStackFrame`, are set, we can populate both frames with initial values. The hardware stack frame is populated as follows:

   - PSR = 0x01000000 - this is the default reset value in the program status register,
   - PC = the address of the task,
   - LR = 0xFFFFFFFF - in our case, tasks never finish, so LR=0xFFFFFFFF (reset value),
   - r12, r3-r0 = 0x00000000 - we may also pass the arguments into the task via r0-r3, but this is not the case in our simple scheduler.

3. Finally, it saves the address of the top of the software stack frame into the task's SP enttry and the address of the task's function into the task's TCB.

After these steps, a new task is ready to be executed for the first time when the task switch occurs, and the task is selected for execution. Figure 1.17 illustrates the memory layout and the contents of the task's stack and TCB after creating Task1 using the `CreateTask()` function.

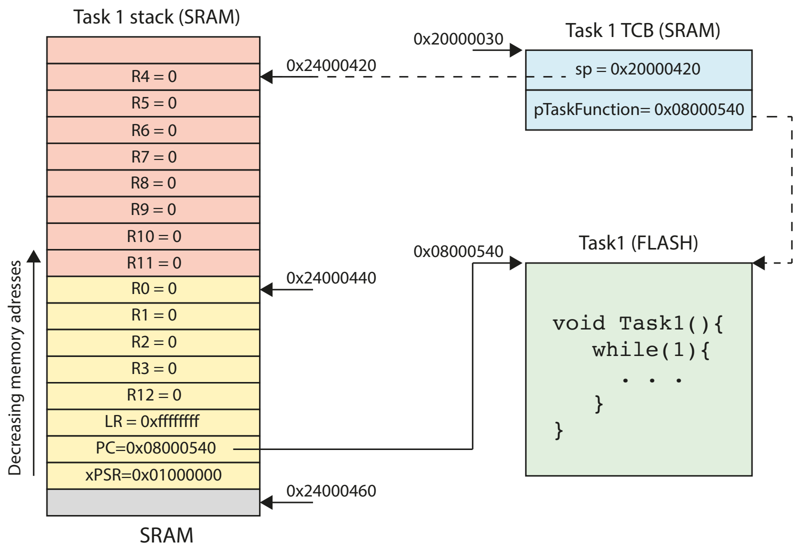


Fig. 1.17: Memory layout and content after calling the `TaskCreate()` function.

1.4.7.6 Task initialisation

The following code presents the function used to initialize all four tasks:

```c
void InitializeTasks(unsigned int* pStackRegion, TCB_Type pTCB[],
                     void (*TaskFunctions[])()){

  unsigned int* pStackBase;

  for(int i=0; i<NTASKS; i++){
        // set the pointer to stack region for a task and
        //     create a task:
    pStackBase = pStackRegion + (i+1)*TASK_STACK_SIZE;
    CreateTask(&pTCB[i], pStackBase, TaskFunctions[i]);
  }

  //saves the top of stack of the first task into PSP:
  __asm__ volatile ( "MSR PSP, %0\n\t"
          :
          : "r" (pTCB->sp)
          );


}
```

Listing 1.18: The function `InitializeTasks()` that initializes all tasks and sets the top of the stack the first task into the PSP register.

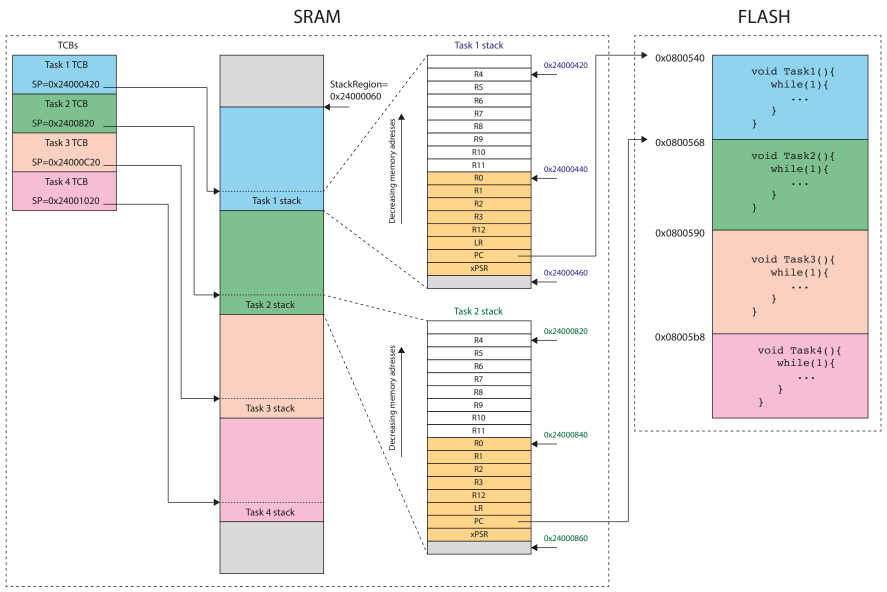
The function `InitializeTasks()` performs the following steps:

1. For each new task:

   a. Set the pointer (`pStackBase`) to the region where the task's stack begins.
   b. Create a new task using the `CreateTask` function and passing `pStackBase` as its argument.

2. Save the top of the stack of the first task into the PSP register.

After these steps, everything is set up for the first context switch on the next timer's interrupt. Figure 1.18 illustrates the memory layout and the task's stack after initial-



```
void Task1(){
   while(1){
      ...
   }
}
```

```
void Task2(){
   while(1){
      ...
   }
}
```

```
void Task3(){
   while(1){
      ...
   }
}
```

```
void Task4(){
   while(1){
      ...
   }
}
```

Fig. 1.18: Memory layout and content after creating four tasks.

izing tasks using the `InitializeTasks()` function.

1.4.7.7 Context switch

Context switching in multitasking environments can be performed using stack pointer (SP) swapping. The process involves saving the current task's context onto its stack and then loading the context of the next task to be executed by swapping the SP. Figure 1.20 shows the process of context switching using stack pointer swapping.

Here's a step-by-step description of how context switching is accomplished using this method:

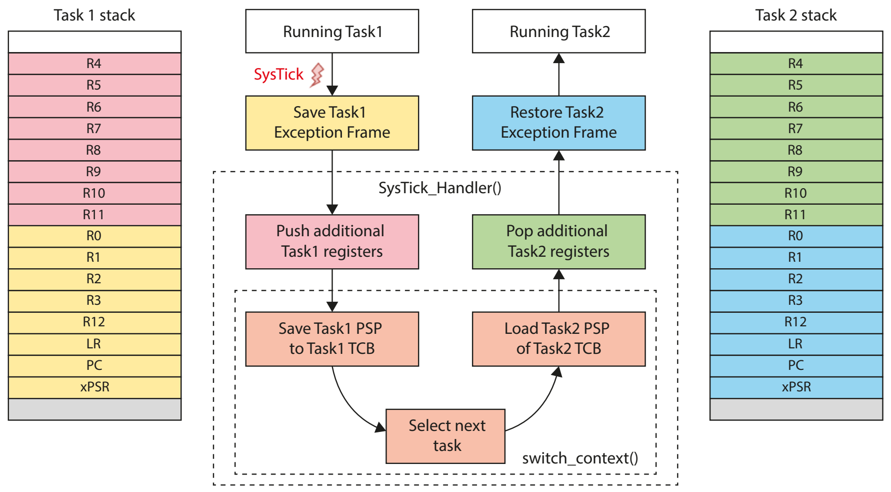


Fig. 1.19: Context switching using stack pointer swapping.

1. When a trigger for context switching occurs (the trigger is a timer interrupt), the CPU saves the exception stack frame onto the Task1 stack using the PSP stack pointer and enters the timer's interrupt handler.
2. The remaining eight registers (R4-R11) are saved the onto the Task1 stack. The context switcher saves the current PSP into the Task1 TCB.
3. The context switcher determines which task should run next. The scheduler considers the round-robin scheduling policy to make this decision.
4. The context switcher retrieves the SP of Task2 from the Task2 TCB and saves it into the PSP register. The PSP now points to the stack where the context of Task2 is saved.
5. The eight registers (R4-R11) of Task2 are popped from stack.
6. The timer handler exits; hence, the de-stacking operation performed by the CPU retrieves the exception frame from the Task2 stack. As the PC of Task2 is part of its exception frame, the CPU returns to Task2

Figure 1.20 shows the chronology of the stack pointer when a context switch happens between Task1 and Task2. The following code presents the function that implements the context switcher:

```
int switch_context (int current_task, TCB_Type pTCB[]) {
  void* current_sp;
    if (current_task != -1) {
       // read the current SP and save it
       //    into TCB of the interrupted task:
     __asm__ volatile ( "MRS %0, PSP\n\t"
                      : "=r" (current_sp)
                      :
                      );

```

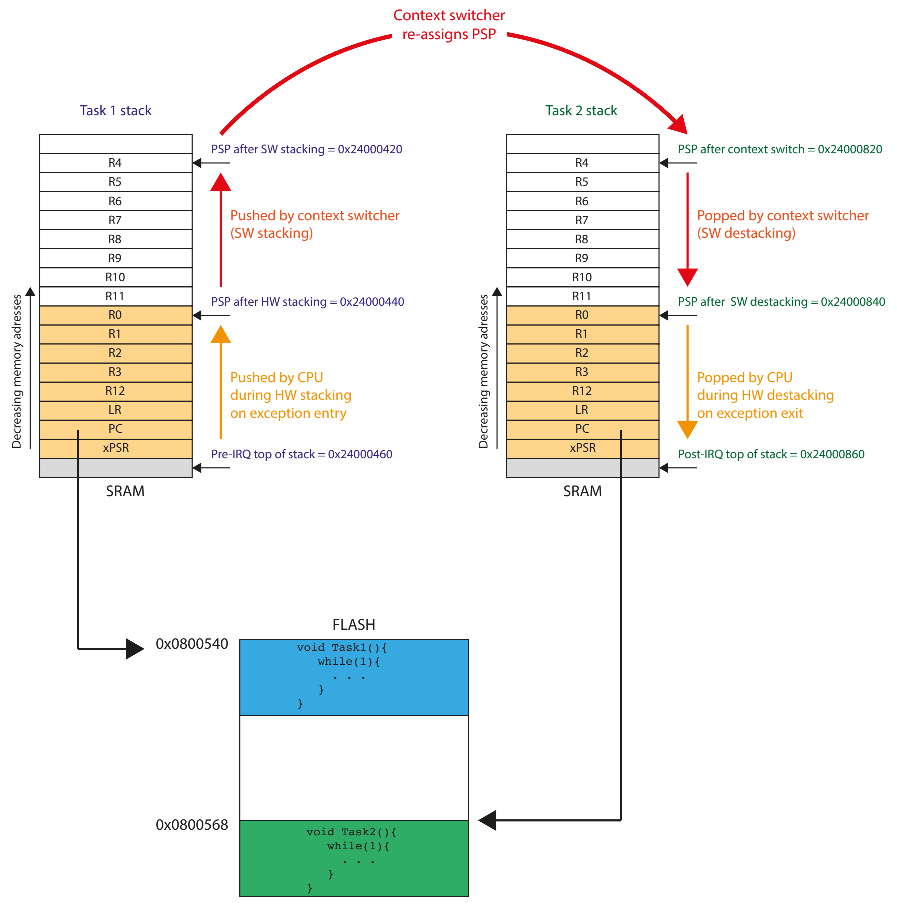

Context switcher
re-assigns PSP
Task 1 stack
Task 2 stack
R4
R5
R6
R7
R8
R9
R10
R11
R0
R1
R2
R3
R12
LR
PC
xPSR
SRAM
Decreasing memory adresses
PSP after SW stacking = 0x24000420
Pushed by context switcher (SW stacking)
PSP after HW stacking = 0x24000440
Pushed by CPU during HW stacking on exception entry
Pre-IRQ top of stack = 0x24000460
PSP after context switch = 0x24000820
Popped by context switcher (SW destacking)
PSP after SW destacking = 0x24000840
Popped by CPU during HW destacking on exception exit
Post-IRQ top of stack = 0x24000860
FLASH
0x0800540
0x0800568
void Task1(){ while(1){ . . . } }
void Task2(){ while(1){ . . . } }

Fig. 1.20: The modification progress of the PSP stack pointer during context switching.

```
        pTCB[current_task].sp = current_sp;
      }

    // select a new task in a round-robin fashion:
    current_task++;
    if (current_task == NTASKS) {
            current_task = 0;
    }

    // set PSP with the SP of the new task:
    __asm__ volatile ( "MSR PSP, %0\n\t"
            :
            : "r" (pTCB[current_task].sp)
           );

    return current_task;
  }
```

Listing 1.19: The function `switch_context()` that implements context switching.

The parameters of the `switch_context` functions are the index of the current task (`current_task`) and the pointer to the TCB table (`pTCB`). The function return the index of a new task.

1.4.7.8 SysTick handler

Finally, we can implement the SysTick handler that will perform task switch:

```c
void SysTick_Handler(void)
{
    void* main_stack_pointer;
    unsigned int tmp;
    // 1. if we interrupted main(), update EXC_RETURN on stack:
    if (current_task == -1) {
        __asm__ volatile ( "MRS %0, MSP\n\t"
                 : "=r" (main_stack_pointer)
                 :
     );
        *((uint32_t*)main_stack_pointer + 1) = 0xFFFFFFFD;
    }

    // 2. Save context of interrupted task:

    if (current_task != -1) {
        __asm__ volatile ( "MRS %0, psp\n\t"
                           "STMFD %0!, {r4-r11}\n\t"
                           "MSR psp, %0\n\t"
                 : "=r" (tmp)
                 );
    }

    // 3: Switch Context:
    current_task = switch_context(current_task, TCB);

    // 4. Restore context of the next task:
    asm volatile ( "MRS %0, psp\n\t"
                   "LDMFD %0!, {r4-r11}\n\t"
                   "MSR psp, %0\n\t" : "=r" (tmp) );
}
```

Listing 1.20: The SysTick handler used to perform task switch.

The SysTick handler performs the following steps:

1. Check whether the Systick interrupted the main function. This happens only once on the first SystTick interrupt. When the main function is interrupted, its exception frame is saved onto the main stack using MSP. The LR register is set to 0xFFFFFFF9, which instructs the CPE that MSP should be for de-stacking upon return from exception. In order to switch to the first task, we should use PSP for de-stacking, so we update the LR value, which the SysTick handler has pushed onto the main stack, with 0xFFFFFFFD.
2. Save the context (R4-R11) of the interrupted task on the task's stack using PSP. This is only performed if the interrupted task is not the main function.

3. Switch context (swap stack pointers) using the texttttswitch_context() function.
4. Restore the context (R4-R11) of the new task from its stack using PSP.
5. Return from interrupt and restore the exception frame of the new task from its stack.

1.4.7.9 Using PendSV for context switching