Contents

Chapter 1
Interrupts and interrupt handling

CHAPTER GOALS

Have you ever wondered how computer components demand and get attention from the CPU? How do they tell the CPU or operating system that something important has just happened in the computer system, which requires an immediate response from the CPU, e.g., new data has just arrived at an I/O interface and should be processed immediately? This is done using so-called interrupts. This chapter will cover the theory and practice of interrupts and their handling. An interrupt is a hardware-initiated procedure that interrupts whatever program (CPU) is currently executing and requests that the CPU immediately start running another program that is written to service the particular interrupt request.
Upon completion of this chapter, you will be able to:

- Distinguish between interrupts and exceptions.
- Explain the operation of the interrupt signals.
- Explain the interrupt and exception handling.
- Explain the function of interrupt vectors and vector tabels.
- Explain the function of an interrupt controller.
- Explain the interrupts and interrupt handling in the Intel and ARM family of processors.

1.1 Introduction

During my childhood, there were two powerful military blocs in Europe and the world: the Eastern (Soviet) Block and the Western (USA) Block. That was a period of geopolitical tension between the Soviet Union and the United States and their respective allies, the Eastern Bloc and the Western Bloc. The country where I grew up, former Yugoslavia, was not part of any of these military blocks, though politically, it

was closer to the eastern block. In the 1970s, former Yugoslav air force purchased a number of Soviet MIG-21 fighter aircraft from the USSR. The MIG-21 aircraft sold to Yugoslav air force had virtually no modern electronic devices, and the military of Yugoslavia wanted to install missile sensors in the planes. However, the USA and its allies have imposed an embargo on the purchase of electronic and computer components against Yugoslavia. Among all the universities in Yugoslavia, only the University of Ljubljana was allowed to purchase a few pieces (up to 20) of each chip that would be used only in the educational process. That's why the Yugoslav Army approached the University of Ljubljana to buy all the necessary electronic and computer components and develop a system that would be installed on the aircraft and would detect missiles. The system at the time had to be based on the modern Motorola 6800 microprocessors from the US. At its core, the system had a microcomputer built on the Motorola 6800 processor and a missile sensor. In addition to detecting missiles, the microcomputer had to do other things, also. If the missile sensor detected a rocket, the computer system had to immediately stop whatever it was currently doing and alert the pilot to the approaching missile. But how would a missile sensor be able to communicate to the CPU if the CPU could do nothing but fetch and execute instructions from memory? Remember that the CPU fetches and executes instructions every clock cycle. That's all it is able to do. So there must be some mechanism by which the CPU can be immediately interrupted and required to start another program. In our case, the CPU would run another program (e.g., display the current altitude and speed of the aircraft). In the event that the sensor detects a missile, it must, in some way, immediately suspend the currently running program and require the CPU to execute a program to flash the warning lights and alert the pilot. So, the CPU must have some mechanism in place to immediately stop the execution of one program and start another program. This mechanism is called **interrupts**, and the program that the CPU starts running in the response to an is called **interrupt service program (ISP)** or **interrupt handler**.

Interrupts and interrupt handling must be **transparent**. This means that the stopped (interrupted) program must not know that it has been stopped and must continue after the termination of the interrupt service program as if it had not been interrupted at all.

In the following chapters, we will learn about the interrupt mechanism and interrupt handling.

1.2 Interrupts

As we said in the Introduction, we want to have to ability to service external interrupts. This is useful if a device external to the processor needs attention. Figure 1.1 illustrates a simplified system with a CPU and a peripheral device. To be able to respond to interrupt requests from a peripheral device, a CPU usually has at least one interrupt request (IRQ) pin and one interrupt acknowledge (INTA) pin. The IRQ pin is the input used by a peripheral device to interrupt the processor (i.e., to interrupt

the normal program flow in the CPU.). Since the CPU should finish executing the current instruction(s) before servicing any external interrupts, the peripheral device may have to wait for several clock cycles before the CPU responds to the interrupt request. The INTA pin is the output used to signal the peripheral device, which has requested an interrupt via the IRQ signal, that the CPU has started servicing the interrupt request and that the IRQ signal can be deactivated. Both pins in Figure 1.1, IRQ and INTA, are active low. Two resistors are used to establish a logic one on both signals IRQ and INTA (i.e., both signals are deactivated) when no one drives them.

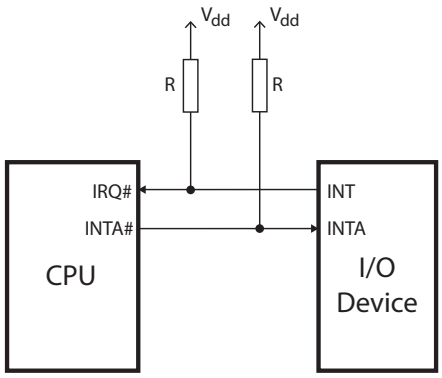

Fig. 1.1: A simplified block diagram of a computer system with interrupt controlling signals.

In general, CPUs can respond to interrupts in two different ways: in either an **edge-sensitive** or **level-sensitive** manner. In an edge-sensitive manner, the interrupt signal input is designed to be triggered by a particular signal edge (level transition): either a falling edge (high to low) or a rising edge (low to high). In a level-sensitive manner, the interrupt signal input is designed to be triggered by a logic signal level. A peripheral device invokes a level-triggered interrupt by driving the signal to and holding it at the active level. We refer to this operation as **asserting the signal**. It de-asserts the signal when the processor signals it to do so. One advantage of level-triggered interrupt inputs is that they allow multiple devices to share a common interrupt signal. Most often, the active level of an interrupt input signal is LOW. In such a case, the interrupt signal is tied to the HIGH voltage level using a pull-up resistor. When multiple peripheral devices share one level-triggered interrupt input signal, the device that wants to assert the interrupt request simply connects the signal to the ground (pulls thew signal LOW). The system in Figure 1.1 uses level-sensitive interrupt signals.

Summary: Assering and de-asserting a signal

Some signals are active high, and some signals are active low. To avoid the problem of high vs. low and the fact that for some signals, active means high and for some signals active means low, we just say asserted (activated) vs. de-asserted (deactivated).

When the device needs the attention from the CPU, it activates (asserts) the IRQ pin on the CPU. During the normal flow of execution through a program, the program counter increases sequentially through the address space, with branches to nearby labels or branches and links to subroutines. The CPU checks the status of the IRQ pin every time before a new instruction pointed to by the program counter is fetched from memory. When a peripheral device requests the interrupt, it is necessary to preserve the previous processor status while handling the interrupt, so that execution of the program that was running when the interrupt request occurred can resume when the appropriate interrupt handler has completed. We say that the interrupts must be 100% transparent. So, when an interrupt request occurs, the CPU

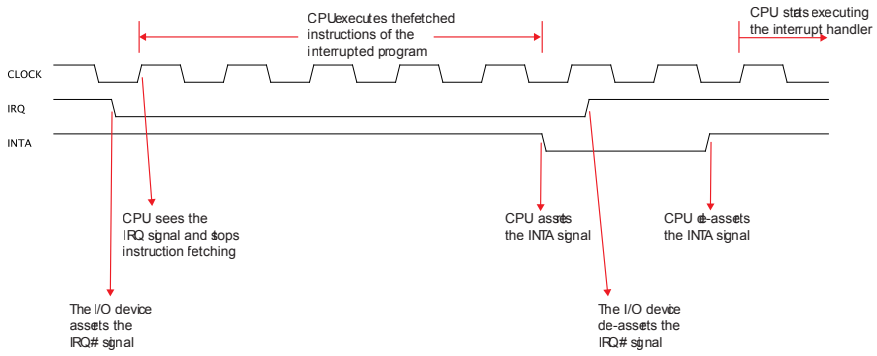

Fig. 1.2: A timing diagram for an external interrupt request.

completes the current instruction and asserts the INTA signal. When a peripheral device sees the INTA signal, it de-asserts the IRQ signal. Figure 1.2 shows the timing diagram for an external interrupt request for the simple system from Figure 1.1.

Then the CPU saves the part of the context of the interrupted program in the stack. A context is a state of the program counter, status register, stack pointer, and all other program-visible CPU registers. Some CPUs save the whole context in the stack, while others save only a part of the context in the stack. Since interrupts can happen at any time, there is no way for the active programs to prepare for the interrupt (e.g., by saving registers that the interrupt handler might write to). It is important to note that calling conventions do not apply when handling interrupts: the interrupt is not being "called" by the active program; it is interrupting the active program. Thus, the interrupt handler code must preserve the content ensure that it

does not overwrite any registers that the program may be using before their content is saved. After the CPU has saved the context, the CPU automatically loads the address of the interrupt handler into the program counter. The interrupt handler is a program written by the user and depends on the peripheral device's functionality. Depending on how much of the context is automatically saved by the CPU, the interrupt handler must first save every register it intends to use in the stack or somewhere in memory.

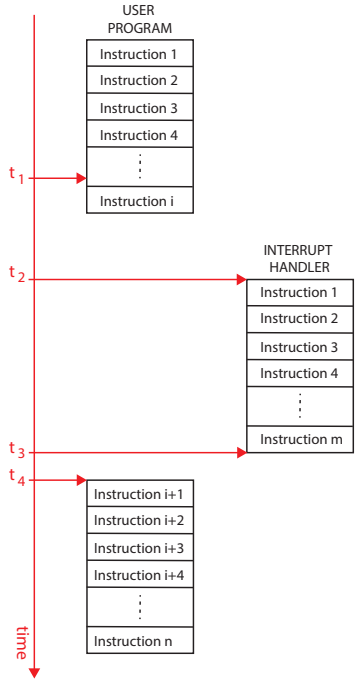

Fig. 1.3: The procedure involved in interrupts.

Figure 1.3 shows the procedure involved in interrupts: the CPU executes the sequence of instructions from a user program until an interrupt request occurs at the time t_1. When the IRQ signal is asserted, the CPU stops executing the user code and starts executing the interrupt handler. But before executing the interrupt handler at time t_2, the CPU must finish the execution of already fetched instructions, save the (part of) context, and obtain the address of the interrupt handler. The time $t_2 - t_1$ required for this procedure is called **interrupt latency**. In general, interrupt latency is the time that elapses from when the IRQ signal is asserted to when the CPU starts to execute the interrupt handler. Interrupt latency duration is usually not predetermined and depends on how many instructions are already in the CPU's pipeline, on how CPU saves the context and on whether any new interrupt requests are temporarily

disabled. Once the CPU completes the execution of the interrupt handler at time t_3, it returns back to the execution of the user code at time t_4. Before returning to user code, the CPU must automatically restore the previously saved context.

1.2.1 Types of interrupts

There are typically three types of interrupts regarding the source of the interrupt: external interrupts (or simply interrupts), traps or exceptions, and software interrupts. External interrupts are triggered by an external device by activating the interrupt request pin on the CPU. Traps or exceptions are activated internally in the CPU, usually as a result of some exceptional condition caused by instruction. For example, traps are caused when illegal or undefined instruction is fetched, or when the CPU attempts to execute an instruction that was not fetched because the address was illegal. A special instruction triggers software interrupts. Such instructions function similarly to subroutine calls, but the subroutine, in this case, the interrupt handler, is not being "called", but an interrupt-like sequence occurs. These software-interrupt instructions are useful when the user program does not know or is not allowed to know the address of the routine which it would like to "call", e.g., they are usually used for requesting operating system services and routines.

External interrupts are divided into two types: maskable and non-maskable interrupts. Maskable interrupts can be enabled or disabled by setting a bit in the CPU's control register or by executing a special instruction. For example, Intel has the CLI instruction to mask the interrupts, and ARM has CPSID instruction for this purpose. Non-maskable interrupts have a higher priority than maskable interrupts. That means that if both maskable and non-maskable interrupts are activated at the same time, the CPU will service the non-maskable interrupt first.

1.2.2 Handling interrupts

In a situation where multiple types of interrupts and exceptions can occur, there must be a mechanism in place where different handler code can be executed for different types of events. In general, there are two methods for handling this problem: polled interrupts and vectored interrupts.

In polled interrupts, the processor branches to a specific address that begins a sequence of instructions that check the cause of the interrupt or exception and branch to handler code for the type of interrupt/exception encountered. This is also called polled interrupt/exception handling.

In vectored interrupts, the processor branches to a different address for each type of interrupt or exception. Each exception address is separated by only one word, and these addresses form a table called **interrupt vector table**. Each entry of the interrupt vector table is called **interrupt vector**, and it is the address of an interrupt

handler. Hence, the vector table contains the start addresses, called interrupt vectors, for all exception handlers. This method is called **vectored interrupt handling**. This concept is common across many processor architectures, although interrupt vector tables may be implemented in other architecture-specific fashions. For example, another common concept is to place a jump instruction (instead of vectors) at each entry in the table. Each of these jump instructions forces the processor to jump to the handler code for each type of interrupt/exception. In this case, the address of each table entry is considered as an interrupt vector.

1.3 RISC-V interrupts

RISC-V architecture defines different privilege modes that determine the level of access and control a program or process has over the system's resources. A privileged mode in a CPU refers to a specific operating mode in which the CPU has access to various system resources. Privileged modes are often used in modern computer architectures to ensure the proper operation, security, and control of the system. Privileged modes are crucial in separating user-level programs from system-level operations and for managing system security, isolation, and resource allocation. For example, a modern CPU restricts a user program from accessing system critical resources (e.g. special CPU registers, memory regions, special instructions, etc.), while the system programs may access all system resources. Privileged modes are the mechanism to achieve this differentiation between user-level and system-level programs. Modern CPUs usually have a separate set of control and status registers (CSRs) for each privileged mode and a special control register that tells which privileged mode the CPU is currently running. Depending on the status of this special control register (i.e. current privileged mode), the CPU can access the corresponding set of CSRs and execute only the instructions allowed in the current privileged level. For example, if the CPU is currently running in a user-privileged mode, it can execute only the standard instruction set. At the same time, executing some special instructions that can alter critical system resources is prohibited. Besides, programs running in user-privileged mode can never alter the content of this special control register and thus switch between privileged modes. But wait, how can we change a privileged mode once the CPU runs in user-privileged mode? Well, it depends on the current privileged mode:

1. If the CPU runs in user-level privileged mode, the only way to switch to a system-level privileged mode is through exceptions (traps or interrupts). Exceptions can trigger mode transitions. When an exception (a trap or an interrupt) occurs, the CPU automatically switches to system-level privileged mode, and the exception handling routine executes in the system-level privileged mode. Upon exiting the exception handler, the CPU automatically switches to the previous (e.g., user-level) privileged mode.
2. If the CPU runs in system-level privileged mode, the CPU can switch to a user-level privileged mode simply by executing a special instruction that alters the content of the special control register and, hence, changes the current system-level privileged mode to user-level privileged mode. CPUs have specific instructions that are used to initiate mode transitions. These instructions are often called privileged and can only be executed when the CPU is in a system-level privileged mode.

1.3.1 RISC-V Privileged Modes

In order to be able to understand interrupts and interrupts handling in RISC-V, we'll briefly describe and explain the privileged modes in RISC-V. Privileged modes are a fundamental part of RISC-V's flexibility, as they enable various operating systems, hypervisors, and security models to be implemented on the same instruction set architecture. Here is a brief description and explanation of three basic privileged modes in RISC-V:

1. **User Mode (U**): User mode is the lowest privilege mode in RISC-V. In this mode, a user-level application or program runs with restricted access to system resources. User mode provides the least privilege and is suitable for application-level code. In user mode, applications can execute most instructions but have limited access to privileged instructions and control registers. User mode can execute basic instructions, access memory, and perform arithmetic operations. However, it cannot directly manipulate control and status registers (CSRs) related to exception handling or interrupt control.
2. **Supervisor Mode (S):** Supervisor mode is a privilege level above user mode. It is designed for operating system kernel code, which needs greater control over system resources and privilege to perform tasks like context switching and managing hardware devices. Supervisor mode has more access to control registers and instructions compared to user mode. It can perform operations related to exception handling, interrupt control, and system management. S-mode can execute privileged instructions that deal with system control and exception handling. It can access and modify most control and status registers (CSRs), including those related to interrupts and exceptions.
3. **Machine Mode (M):** Machine mode is the highest privilege mode in RISC-V. It provides complete control over the system, including access to all resources and system-wide configuration. M-mode has full access to all instructions, control registers, and hardware resources, making it suitable for tasks such as system initialization, low-level device control, and platform management. M-mode can execute all RISC-V instructions, including those reserved for privileged and system-level operations. It can access and modify all control and status registers (CSRs), and it has control over exceptions and interrupts across all privilege levels. Upon reset, RISC-V enters machine mode.

The E31 RISC-V core in FE-310 SoC supports only Machine and User privilege modes. The transition between privilege modes in E31 RISC-V is typically controlled by changing specific bits in control and status registers (CSRs). The machine mode handles these transitions, ensuring that the processor switches between user and machine modes appropriately. Additionally, exceptions and interrupts may trigger mode transitions, allowing the processor to respond to exceptional conditions or external events. As all exceptions (traps and interrupts) execute in Machine mode, we will restrict the description of exceptions only to this privilege mode.

1.3.2 RISC-V Machine Modes Exceptions

According to the RISC-V Privileged Architecture [**?**], the E31 RISC-V CPU comprises five control and status registers for Machine privilege mode:

1. **mstatus:** In RISC-V, the **mstatus** (Machine Status) register is a critical control and status register (CSR) used to manage and store various information related to the Machine privilege mode. The **mstatus** register plays a central role in controlling exception handling, interrupt handling, and the overall operation of the processor in machine mode. The **mstatus** register keeps track of and controls the CPU's current operating state, including whether or not interrupts are enabled. A summary of the **mstatus** bits related to interrupts in the E31 RISC-V CPU is provided in Figure 1.4. Note that this is not a complete

Fig. 1.4: The **mstatus** register.

description of **mstatus** as it contains fields unrelated to interrupts. For the full description of **mstatus**, please consult the RISC-V Instruction Set Manual, Volume II: Privileged Architecture. The **mstatus** register contains the following exception-related bits:

   a. **MIE** (Machine Interrupt Enable): This bit controls whether machine-level interrupts are globally enabled or disabled. When MIE is set, the CPU can process machine-level interrupts; when it is cleared, machine-level interrupts are disabled.
   b. **MPIE** (Machine Previous Interrupt Enable): This bit stores the previous state of MIE before it was modified due to an interrupt. It helps manage interrupt nesting by preserving the previous interrupt-enable state.
   c. **MPP** (Machine Previous Privilege Mode): This two-bit field stores the previous privilege mode before the CPU entered machine mode due to an interrupt. It is used during return from interrupt to return to the appropriate privilege mode after processing an interrupt.

2. **mie:** The **mie** (Machine Interrupt Enable) register is responsible for enabling or disabling various types of interrupts that can interrupt the execution of the CPU in machine mode. Individual interrupts are enabled by setting the appropriate bit in the **mie** register. The **mie** register is depicted in Figure 1.5. The **mie** register contains the following bits:

   a. **MSIE** (Machine Software Interrupt Enable): This bit controls whether machine-level software interrupts are enabled or disabled. When MSIE

Fig. 1.5: The **mie** register.

is set, the CPU can process machine-level software interrupts; otherwise, machine-level software interrupts are disabled.

b. **MTIE** (Machine Timer Interrupt Enable): This bit controls whether machine-level timer interrupts are enabled or disabled. When MTIE is set, the CPU can process machine-level timer interrupts.
c. **MEIE** (Machine External Interrupt Enable): This bit controls whether machine-level external interrupts are enabled or disabled. When MEIE is set, the CPU can process machine-level external interrupts.

3. **mip:** The **mip** (Machine Interrupt Pending) register indicates which interrupts are currently pending. The **mip** register is depicted in Figure 1.6. When an

Fig. 1.6: The **mip** register.

interrupt occurs, the corresponding bit in **mip** is set to 1. When the CPU takes an interrupt, the corresponding bit in **mip** is cleared. The **mip** register contains the following bits:

a. **MSIP** (Machine Software Interrupt Pending): When MSIP is set, the Machine Software Interrupt is pending.
b. **MTIP** (Machine Timer Interrupt Pending): When MTIP is set, the Machine Timer Interrupt is pending.
c. **MEIP** (Machine External Interrupt Pending): When MEIP is set, the MachineExternal Interrupt is pending.

If more than one interrupt is pending, the RISC-V CPU prioritizes the interrupts as follows, in decreasing order of priority: Machine External Interrupts (highest priority), Machine Software Interrupts, and Machine Timer Interrupts (lowest priority).

4. **mcause:** In RISC-V architecture, the **mcause** register is a control and status register (CSR) that is used to provide information about the cause of an exception or interrupt that occurred in machine mode. A summary of the **mcause** bits related to interrupts in the E31 RISC-V CPU is provided in Figure 1.7. When a trap is taken in machine mode, the most significant bit in **mcause** (bit INT) is 0,

Fig. 1.7: The **mcause** register.

and the ten least-significant bits (EXCEPTION CAUSE field) are written with a code indicating the event that caused the trap. When an interrupt is taken, the most significant bit of **mcause** (bit INT) is set to 1, and the ten least-significant bits (EXCEPTION CAUSE field) contain the interrupt number, using the same encoding as the bit positions in the **mip** register. Table 1.1 lists exception codes and their description. For example, a Machine Timer Interrupt causes **mcause** to be set to `0x80000007`.

Table 1.1: **mcause** Exception Codes and their description.

INT	EXCEPTION CODE	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
1	3	Machine software interrupt
1	7	Machine timer interrupt
1	11	Machine external interrupt

5. **mtvec:** The **mtvec** register has two main functions. Firstly, it specifies the base address for the vector table, which contains the addresses of exception handlers. Secondly, it sets the mode by which the E31 CPU will process exceptions. The RISC-V CPU can process exceptions in two modes: direct and vectored. In **direct mode**, the **mtvec** register holds the address of a single global exception handler. The processor directly jumps to this global handler's address when a trap or interrupt occurs. In direct mode, we might use a single handler for all exceptions, simplifying the exception-handling process. However, it may not be suitable for systems requiring fine-grained control over exception handling. In **vectored mode**, the **mtvec** register holds the base address of the vector table. In this mode, the processor uses a 10-bit field in the In **mcause register** to index the vector table and find the appropriate handler for the specific trap or interrupt that occurred. The vectored mode allows more flexibility in handling various exceptions and interrupts with different routines. In vectored mode, we can have multiple handlers for different exceptions and interrupts, allowing us to handle

each type of exception or interrupt differently. This is often the preferred way of interrupt handling. The **mtvec** register is depicted in Figure 1.8. The **mtvec**

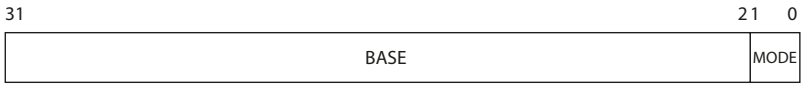

Fig. 1.8: The **mtvec** register.

register contains the following bit fields:

a. **MODE:** This 2-bit field sets the interrupt processing mode (00-Direct, 01-Vectored).
b. **BASE:** This 30-bit field contains the vector table base address. This field requires 64-byte alignment.

Table 1.2 describes how an address of the exception handler is computed in two different interrupt processing modes. For example, suppose the global and ma-

Table 1.2: **mtvec** Modes and Address of Exception Handler Encoding.

MODE	Interrupt Processing Mode	Address of Exception Handler
0	Direct	`PC = BASE`
1	Vectored	`PC = BASE + 4 x mcause[EXCEPTION CODE]` *NOTE: BASE must be 64-byte aligned. This is to avoid an adder in the above computation*

chine timer interrupts are enabled, and the BASE is set to 0x20011500. If the vectored mode is selected and a machine timer interrupt occurs, the EXCEPTION CODE in the **mcause** register will be 0x07. Then, the address of the interrupt handler that processes the machine timer interrupts will be 0x20011500 + 4 x 0x07 = 0x20011500 + 0x1C = 0x2001151C. Hence, when the interrupt is taken, the PC is set to 0x2001151C, and the first instruction of the interrupt handler should be at this address.

Configuring these five Control and Status Registers registers correctly is crucial for proper exception handling in RISC-V systems, as they dictate where the processor should jump when an exception occurs and how exceptions are managed. These CSRs are not memory-mapped and can only be accessed through special privileged instructions: **csrr** and **csrw** for read and write, respectively. Hence, To work with these CSRs, developers must use assembly language instructions to read and modify these registers as needed.

1.3.3 FE-310 Interrupts

The SiFive Freedom E310, also known as FE310, is a microcontroller based on the RISC-V architecture. It's designed for embedded and IoT applications and is notable for being one of the early implementations of the RISC-V ISA. Let us have a deeper view of interrupts supported in SiFive Freedom E310. The FE310 SoC supports two

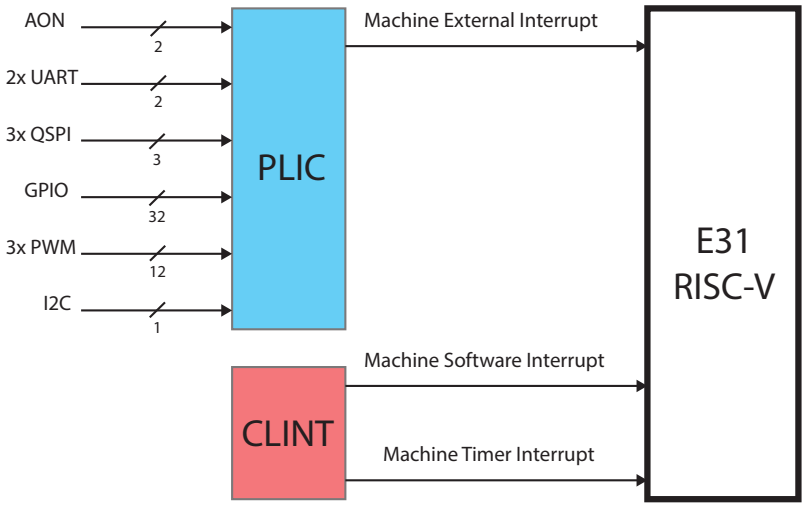

Fig. 1.9: FE310 Interrupt Architecture Block Diagram.

types of RISC-V interrupts: local and global. Local interrupts are signalled directly to the RISC-V E31 CPU with a dedicated interrupt line for each local interrupt. The RISC-V E31 CPU has three interrupt lines for external, software and timer interrupts (Figure 1.9). Software and timer interrupts are local interrupts generated by the Core-Local Interruptor (CLINT). Besides software and timer interrupts, various I/O devices (e.g., UART, GPIO, etc.) can use global interrupts to activate the external interrupt line and to interrupt the CPU. Global interrupts from I/O devices are routed through a Platform-Level Interrupt Controller (PLIC), which will be described later.

The CLINT is a mandatory component in RISC-V processor systems. It's responsible for managing timer-related and software-generated interrupts at the core level. The CLINT generates two interrupts:

1. Machine Timer Interrupts: The CLINT contains a timer called the Machine Timer, which can generate timer interrupts for various purposes, including timekeeping, scheduling, and triggering tasks at specific intervals.
2. Machine Software Interrupts: In RISC-V, the software can generate software interrupts to communicate with the operating system. In general, the program running in user mode is not allowed to call operating system procedures. Hence,

the only way a user program makes a system call is by generating a software interrupt. The software interrupt handler running in machine mode then calls an operating system procedure. The CLINT can be used to handle these software-generated interrupts.

The CLINT comprises memory-mapped control and status registers related to software and timer interrupts. Table 1.3 shows the memory map for CLINT on SiFive FE310.

Table 1.3: Memory map for CLINT registers on SiFive FE310 SoC.

Address	Width	Register
0x02000000	4B	**msip**
0x02004000	8B	**mtimecmp**
0x0200BFF8	8B	**mtime**

1.3.3.1 Machine Software Interrupts

A machine software interrupt is an interrupt generated by software running in machine mode to request attention from the processor for specific tasks or events. Machine software interrupts are generated by writing '1' to the **msip** register within CLINT. The **msip** register is a 32-bit memory-mapped register where the upper 31 bits are hardwired to zero. The least significant bit of the **msip** register is reflected in the MSIP bit of the **mip** register. On reset, the **msip** register is cleared to zero.

1.3.3.2 Machine Timer Interrupts

CLINT, which is a mandatory part of RISC-V architecture, provides a 64-bit real-time counter, which monotonically increases at a clock speed, and its content is visible as a memory-mapped register **mtime**. In the FE310 SoC, CLINT is responsible for providing the real-time counter. Machine timer interrupt is a local interrupt, which can be generated by using two architecturally defined timer registers: **mtime** and **mtimecmp**:

1. **mtime** register: The 64-bit **mtime** register stores the current value of the 64-bit timer counter. The software can read this register to determine the current time.
2. **mtimecmp** register: The **mtimecmp** register holds a value that is compared with the **mtime** register. When **mtime** reaches the value stored in **mtimecmp**, it triggers a timer interrupt. This register is used to set up timer interrupts for specific time intervals.

In summary, the machine timer generates timer interrupts when the **mtime** matches or exceeds the value stored in the **mtimecmp** register. This feature is crucial for

implementing preemptive multitasking, where the processor can switch between tasks at predefined time intervals.

1.3.4 Interrupt Entry and Exit

Interrupt entry and exit refer to the processes by which a RISC-V processor handles interrupts. These processes involve transitioning from regular program execution to an interrupt handler and returning to regular program execution after the interrupt is serviced. In the following subsections, we describe and explain interrupt entry and exit in RISC-V.

1.3.4.1 Interrupt Entry

When a machine interrupt occurs:

1. The value of the MIE bit in **mstatus** is copied into the MPIE bit in **mstatus**, and then MIE is cleared, effectively disabling interrupts.
2. The privilege mode prior to the interrupt is saved in the MPP field in **mstatus**.
3. The cause of the interrupt is encoded into EXCEPTION CODE in **mcause**.
4. The current PC is copied into the **mepc** register, and then the PC is set to the value specified by **mtvec** as described in Table 1.2.

At this point, control is handed over to software in the interrupt handler with interrupts disabled. Interrupts can be re-enabled by explicitly setting the MIE bit in **mstatus** or by executing an `mret` instruction to exit the handler.

1.3.4.2 Interrupt Exit

To exit from a machine interrupt, the `mret` instruction must be executed at the end of the interrupt handler. When a `mret` instruction is executed, the following occurs:

1. The privilege mode is set to the value encoded in the MPP field in **mstatus**.
2. In the **mstatus** register, the MIE bit is set to the value of MPIE.
3. The PC is set to the value of **mepc**, hence pointing to the instruction, which was interrupted.

At this point, control is handed over to the previously interrupted program.

1.3.5 Implementing Vector Table and Handlers

Implementing a vector table and handlers in assembly language for RISC-V involves setting up a program structure to store the addresses of exception handlers

and configuring the system to use this table when exceptions occur to jump to the interrupt-specific handler. Below are the steps to implement an exception table and handlers in RISC-V assembly:

1. **Define the Vector Table**: Create a program structure that serves as the vector table. As we have learned, the address of the first instruction of an interrupt handler is calculated using the BASE address of the vector table and the exception cause (Table 1.2). Each entry in the vector table occupies exactly 4 bytes, and there is only room for one instruction per handler in the vector table. Therefore, the only instructions in the exception table should be the jump instructions that transfer control to an interrupt-specific handler. An example of the vector table is as follows:
   The vector table is populated with jump instructions to transfer control to interrupt-specific handlers. For example, the jump instruction (`j _mtim_interrupt_handler`) that causes the jump to the timer interrupt handler is placed at the offset 7 x 4 = 0x1C from the beginning of the vector table. So when a machine timer interrupt occurs, the PC is set to BASE + 0x1C and the CPU will execute the `j _mtim_interrupt_handler` instruction.

```
# ---------------------------------
#
#  V E C T O R   T A B L E
#
#    must be 64-byte aligned.
# ---------------------------------

.balign 64
.global _vector_table
_vector_table:                  # BASE
    j _default_handler
    j _default_handler
    j _default_handler
    # ------------------------------
    j _msw_interrupt_handler    # 3
    # ------------------------------
    j _default_handler
    j _default_handler
    j _default_handler
    # ------------------------------
    j _mtim_interrupt_handler   # 7
    # ------------------------------
    j _default_handler
    j _default_handler
    j _default_handler
    # ------------------------------
    j _mext_interrupt_handler   # 11
    # ------------------------------
```

Listing 1.1: A vector table for E31 RISC-V.

We can see from Listing 1.1 that besides the jump instructions to exception handlers for software, timer and external interrupts, there is also a jump instruction to `_default_handler` in all other entries in the vector table. We have already learned that there are only three interrupt sources in FE310 SOC (software, timer and external), so why do we need the fourth interrupt handler `_de-`

`fault_handler`? This is to ensure that in case of a trap (INT=0 in **mcause**), the CPU executes `_default_handler`.

2. **Register the Base Vector Table Address:** We should configure the **mtvec** register to point to the exception table. Also, we should set the preferred interrupt processing mode in mtvec. Listing 1.2 presents the RISC-V assembly code to register the base address and to select the vectored mode:

```
#---------------------------------------
#    Register the base address for vector table
#      in mtvec
#
#@arguments:
#    # a0 - interrupt vector table base address
#    # a1 - interrupt processing mode
#           (0x0 - direct, 0x1 - vectored)
#---------------------------------------
.balign 4
.global register_handler
.type register_handler, @function
register_handler:
    # prologue:
    addi sp, sp, -16     # Allocate the routine
                         #   stack frame
    sw ra, 12(sp)        # Save the return address
    sw fp, 8(sp)         # Save the frame pointer
    sw s1, 4(sp)
    sw s2, 0(sp)
    addi fp, sp, 16      # Set the framepointer

    or a0, a0, a1        # OR base address with mode
    csrw mtvec, a0       # and save into mtvec

    # epilogue:
    lw s2, 0(sp)
    lw s1, 4(sp)
    lw fp, 8(sp)         # restore the frame pointer
    lw ra, 12(sp)        # restore the return address
    addi sp, sp, 16      # de-allocate the routine
                         #   stack frame
    ret
```

Listing 1.2: Assembly function for registering the vector table base addreess.

3.

4. **Define Exception Handler:** Write the exception handler routines in assembly language. Each handler should be a separate section of code that corresponds to a specific exception type and ends with the `mret` instruction. The prologue of an interrupt handler usually begins with saving the registers onto the stack to avoid overwriting the contents of the saved registers (s0-s11). After the body of the exception handler executes, the epilogue of an interrupt handler restores the saved registers from the stack. Finally, the handler returns with `mret`, an instruction unique to machine mode. The `mret` instruction restores the PC from **mepc**, the previous interrupt-enable setting, and the privilege mode as described in Subsection 1.3.4.2. For example, the following code (Listing 1.3) presents the RISC-V assembly code for a machine timer interrupt handler:

```
#----------------------------------------
#   Machine Timer Interrupt Handler
#----------------------------------------
.balign 4
.global _mtim_interrupt_handler
_mtim_interrupt_handler:

# Prologue :
#    save 16 ABI caller registers
#    (ra, t0-t6, a0-a7)
addi sp, sp, -16*4   # Allocate the routine stack frame
sw t0, 0*4(sp)
sw t1, 1*4(sp)
sw t2, 2*4(sp)
sw t3, 3*4(sp)
sw t4, 4*4(sp)
sw t5, 5*4(sp)
sw t6, 6*4(sp)
sw a0, 7*4(sp)
sw a1, 8*4(sp)
sw a2, 9*4(sp)
sw a3, 10*4(sp)
sw a4, 11*4(sp)
sw a5, 12*4(sp)
sw a6, 13*4(sp)
sw a7, 14*4(sp)
sw ra, 15*4(sp)

# Decode interrupt cause
csrr t0, mcause      # read exception cause
bgez t0, 1f          # exit if not an interrupt

# Increment timer compare by 1000 cycles
li t0, 0x0200BFF8    # load the mtime address
lw t1, 0(t0)         # load mtime (LO)
lw t2, 4(t0)         # load mtime (HI)
li t3, 1000          # load 1000 cycles
add t3, t1, t3       # increment lower bits by 1000
sltu t1, t3, t1      # generate carry-out
add t2, t2, t1       # increment upper bits with carry

li t0, 0x02004000    # load the mtimecmp address
sw t3, 0(t0)         # update mtimecmp (LO)
sw t2, 4(t0)         # update mtimecmp (HI)

1:
# Epilogue: restore ABI caller regs
lw t0, 0*4(sp)
lw t1, 1*4(sp)
lw t2, 2*4(sp)
lw t3, 3*4(sp)
lw t4, 4*4(sp)
lw t5, 5*4(sp)
lw t6, 6*4(sp)
lw a0, 7*4(sp)
lw a1, 8*4(sp)
lw a2, 9*4(sp)
lw a3, 10*4(sp)
lw a4, 11*4(sp)
lw a5, 12*4(sp)
lw a6, 13*4(sp)
lw a7, 14*4(sp)
lw ra, 15*4(sp)
addi sp, sp, 16*4    # de-allocate the routine stack frame
mret
```

Listing 1.3: Assembly code for the machine timer interrupt.

The code in Listing 1.3 assumes that interrupts are globally enabled in **mstatus** (MIE=1), that timer interrupts have been enabled in **mie**, and that **mtvec** has been set to the base address of the vector table with the interrupt processing mode set to vectored. The prologue preserves 16 registers according to RISC-V ABI (Application Binary Interface). You may find this a little odd — why waste 16 instructions and 64 bytes in memory to save these registers? Well, it turns out there is a very good reason we do this. When writing an interrupt handler in RISC-V assembly language, it's essential to save and restore the necessary registers to ensure the proper operation of the interrupted program. The specific registers that should be saved onto the stack can vary depending on the RISC-V privilege mode, the interrupt source, and the calling conventions of the platform. However, here's a general guideline for which registers we should consider saving:

a. **ra** register stores the return address for function calls. Saving and restoring this register ensures that control can return correctly to the interrupted program.
b. **Caller-Saved Registers t0-t6** can be freely modified by the caller (interrupted program) without the caller being responsible for saving their original values. If the interrupt handler modifies any of these registers, we should save and restore them to maintain the integrity of the interrupted program.
c. **Stack Pointer** when the interrupt handler needs additional stack space. In such a case, we need to save and restore the stack pointer to ensure that stack operations do not interfere with the interrupted program's stack.
d. **Other Registers Used by the Interrupt Handler**. Depending on the specific needs of the interrupt handler, we may use additional registers for temporary storage or calculations or for passing arguments. If these registers are modified, we should save and restore them.

After the prologue, the handler decodes the exception cause by examining **mcause**: interrupt if **mcause** < 0, trap otherwise. Then, it simply increments the time comparator so that the next timer interrupt occurs about 1000 timer cycles in the future. The handler is not preemptible, as it keeps interrupts disabled throughout the handler. Finally, the epilogue restores saved registers and returns with `mret`.
We can also write interrupt handlers in C. To write an interrupt handler in C for a RISC-V-based system, we typically need to use a combination of assembly language and C code. For example, reading and writing CSRs (e.g., **mcause**) is only possible with the special `csrr`, `csrw` instructions; hence, we are forced to use assembly language for such operations. The exact details of how to implement interrupt handlers in C can vary depending on your platform and compiler, but we will give a general outline of how to write an interrupt handler in C for a RISC-V system:

a. **Mark the Function as an Interrupt Handler**: Usually, we use compiler-specific attributes or pragmas to mark the function as an interrupt handler. This attribute is crucial for the compiler to generate prologue and epilogue sequences for an interrupt handler and to put the `mret` instruction at the end of the generated code. The following C code presents how to mark a function as an interrupt handler:

```c
/*
 * Use "interrupt" attribute to indicate that the specified
 * function is an interrupt handler.
 * The compiler generates function entry and exit
 * sequences suitable for use in an interrupt handler
 * when this attribute is present.
*/

__attribute__((interrupt)) void interrupt_handler(void) {
    // Interrupt handling code
}
```

Listing 1.4: Interrupt handler function in C.

b. **Use inline assembly for accessing CSRs:** To read/write the CSRs registers in RISC-V, we should use inline assembly. The exact details of how to use inline assembly depend on the compiler, so we should always consult the compiler manual. Here is an example of how to write inline assembly to read the **mcause** register in C:

```c
unsigned int mcause_value;

// Inline assembly to read mcause
asm volatile(
    "csrr %0, mcause"  // Read mcause into %0
    : "=r" (mcause_value)  // Output : mcause_value
);
```

Listing 1.5: Inline assembly to read **mcause**.

The `volatile` qualifier is necessary as GCC optimizers sometimes discard `asm` statements if they determine there is no need for the output variables. Using the `volatile` qualifier disables these optimizations.

Listing 1.6) presents the machine timer interrupt handler.

```c
unsigned int *pMTime  = (unsigned int *)0x0200bff8;
unsigned int *pMTimeCmp = (unsigned int *)0x02004000;

__attribute__ ((interrupt)) void mtime__handler (void) {

  unsigneg int mcause_value;
  // Decode interrupt cause:
  // Non memory-mapped CSR registers can only be accessed
  // using special CSR instructions. Hence, we should use
  // inline assembly:
  __asm__ volatile ("csrr %0, mcause"
                    : "=r" (mcause_value) /* output */
                    : /* input : none */
```

```
                        : /* clobbers: none */
    );

  if (mcause_value & 0x8000007) { // mtime interrupt!
    // Increment timer compare by 500 ms:
    *pMTimeCmp = *pMTime + 16384;
  }
}
```

Listing 1.6: Machine timer interrupt handler in C.

5. **Enable Global Interrupts:** To enable machine-level interrupts, we should set the MIE bit in the **mstatus** register. The following code (Listing 1.7) presents the RISC-V assembly code to enable global machine-level interrupts :

```

.equ MSTATUS_MIE_BIT_MASK,  0x00000008  # bit 3

#---------------------------------------
#    Enable global interrupts in mstatus
#---------------------------------------
.balign 4
.global enable_global_interrupts
.type enable_global_interrupts, @function
enable_global_interrupts:
    # prologue:
    addi sp, sp, -16     # Allocate the routine
                         #      stack frame
    sw ra, 12(sp)        # Save the return address
    sw fp, 8(sp)         # Save the frame pointer
    sw s1, 4(sp)
    sw s2, 0(sp)
    addi fp, sp, 16      # Set the framepointer

    li t0, MSTATUS_MIE_BIT_MASK
    csrs mstatus, t0     #  set the MIE bit in mstatus

    # epilogue:
    lw s2, 0(sp)
    lw s1, 4(sp)
    lw fp, 8(sp)         # restore the frame pointer
    lw ra, 12(sp)        # restore the return address
    addi sp, sp, 16      # de-allocate the routine
                         #      stack frame
    ret
```

Listing 1.7: Assembly function for enabling global interrupts in the **mstatus** register.

6. **Enable Particular Interrupt:** Depending on what particular interrupt (software, timer or external) we would like to enable, we should set an appropriate bit in the textbfmie register. Listing 1.8 presents the RISC-V assembly code to enable the machine timer interrupt:

```

.equ MIE_MTIE_BIT_MASK,     0x00000080  # bit 7

#---------------------------------------
#    Enable machine timer interrupt in mie
#---------------------------------------

.balign 4
```

```
.global enable_mtimer_interrupt
.type enable_mtimer_interrupt, @function
enable_mtimer_interrupt:
    # prologue:
    addi sp, sp, -16     # Allocate the routine
                         #   stack frame
    sw ra, 12(sp)        # Save the return address
    sw fp, 8(sp)         # Save the frame pointer
    sw s1, 4(sp)
    sw s2, 0(sp)
    addi fp, sp, 16      # Set the framepointer

    li t0, MIE_MTIE_BIT_MASK
    csrs mie, t0         # set MTIE in mie

    # epilogue :
    lw s2, 0(sp)
    lw s1, 4(sp)
    lw fp, 8(sp)         # restore the frame pointer
    lw ra, 12(sp)        # restore the return address
    addi sp, sp, 16      # de-allocate the routine
                         #   stack frame
    ret
```

Listing 1.8: Assembly function for enabling the machine timer interrup in the **mie** register.

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

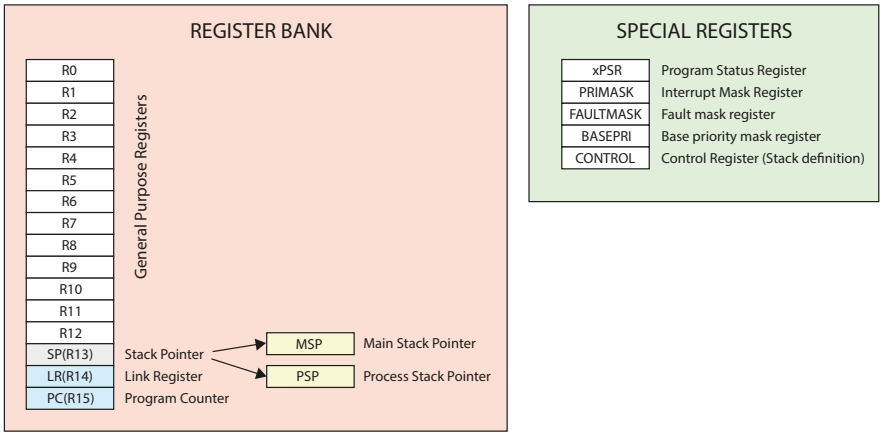

Fig. 1.10: ARM Cortex-M7 core registers.

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

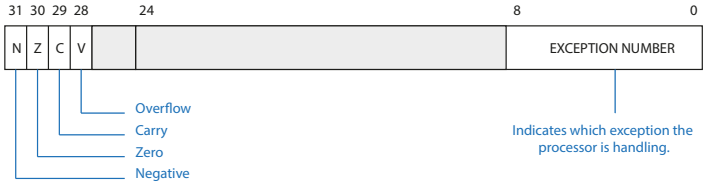

Fig. 1.11: xPSR register.

   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

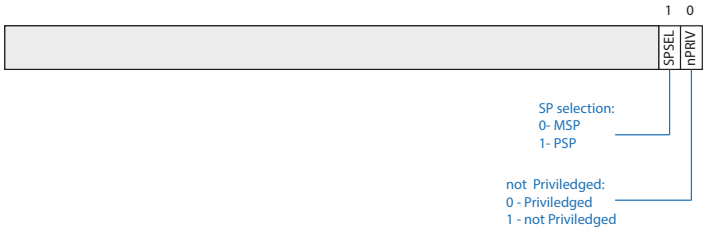

Fig. 1.12: CONTROL register.

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

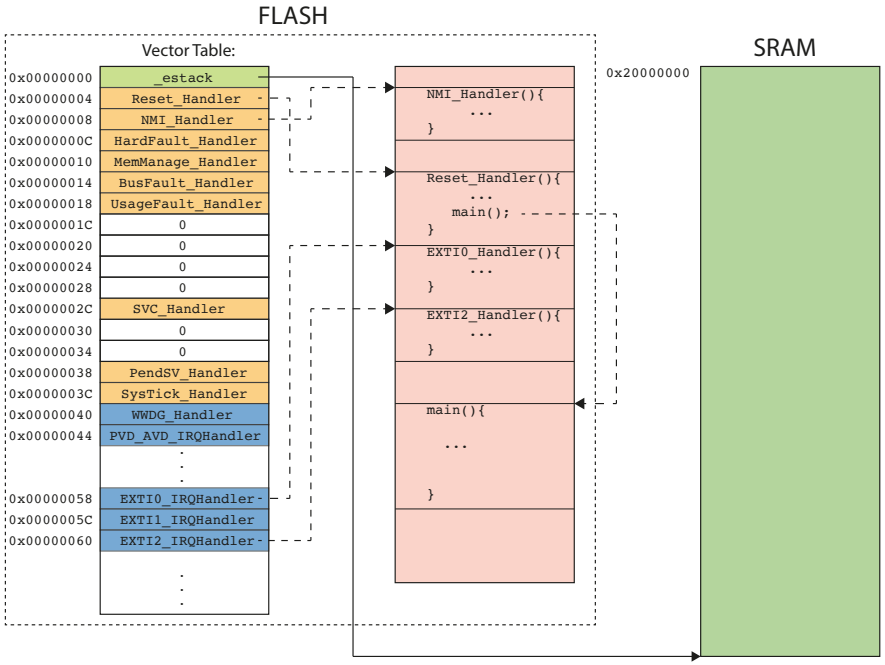

Fig. 1.13: The memory layout of the vector table and exception handlers in ARM Cortex-M7 cores.

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

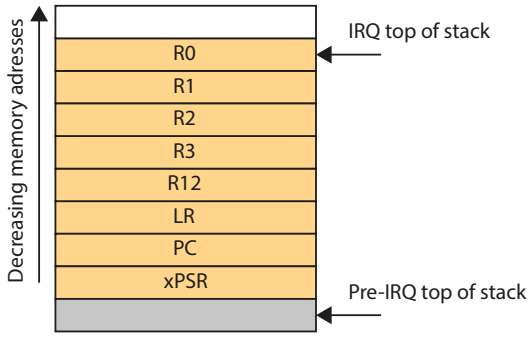

Fig. 1.14: The layout of the stack frame after stacking in ARM Cortex-M7.

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 en-

sures 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:

```
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

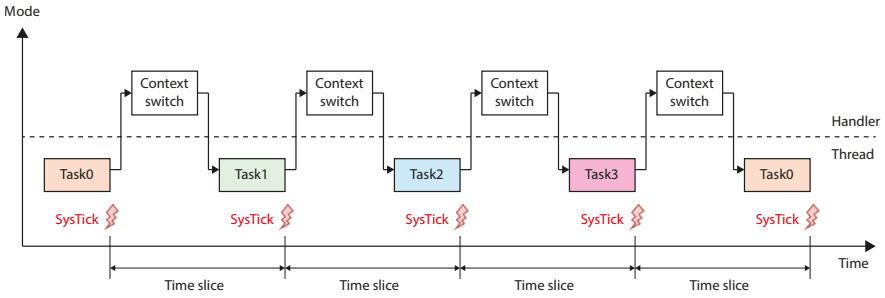

Fig. 1.15: A simple task scheduler.

A simple round-robin task scheduler (Figure 1.15) on Cortex-M7 processors effectively manages multiple tasks or threads in a cooperative multitasking environ-

ment. 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.

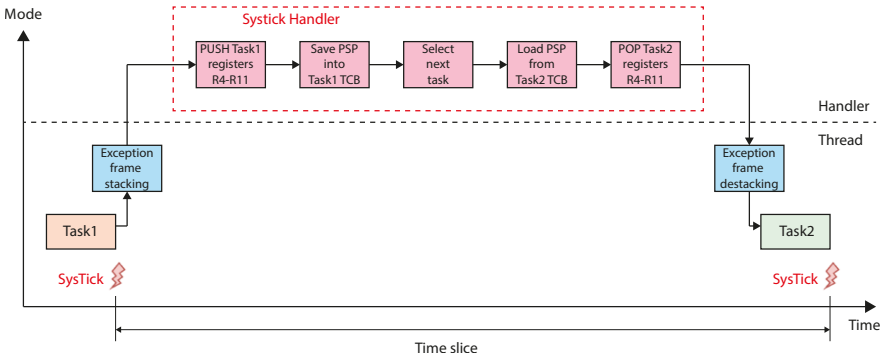

Fig. 1.16: A simple task scheduler.

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:

```
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 `TaskCreate()` function saves the address of the task's stack and the address of the task's function into the task's TCB.

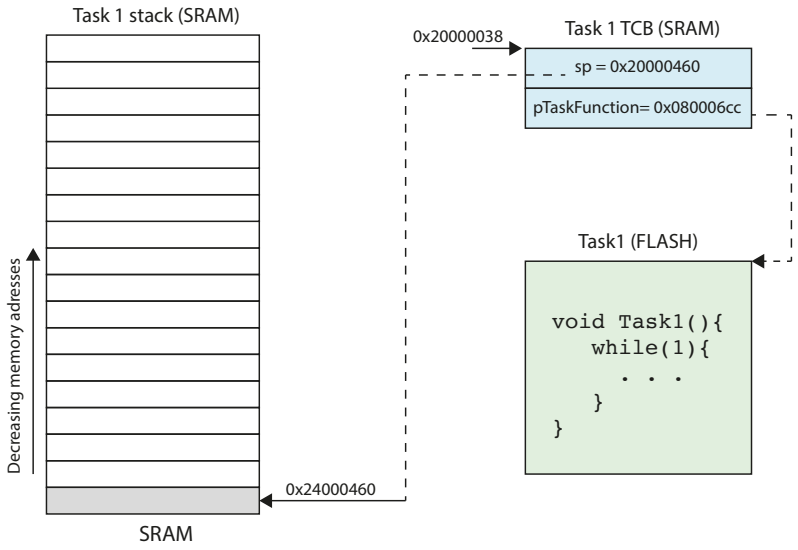

Fig. 1.17: Memory layout and content after calling the `TaskCreate()` function.

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

```
1 void TaskCreate(TCB_Type* pTCB,
                  unsigned int* pTaskStackBase,
3                 void (*TaskFunction)()){

5   pTCB->sp       = (unsigned int*) pTaskStackBase;
    pTCB->pTaskFunction = TaskFunction;
7 }
```

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

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

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

Figure 1.17 illustrates the memory layout and the contents of the task's stack and TCB after creating Task1 using the `TaskCreate()` function.

1.4.7.6 Task initialisation

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

```
void TaskInit(TCB_Type* pTCB){
  HWSF_Type* pHWStackFrame;
  SWSF_Type* pSWStackFrame;


  // Set pointers to HWSF and SWSF:
  pHWStackFrame = (HWSF_Type*)((void*)pTCB->sp - sizeof(HWSF_Type));
  pSWStackFrame = (SWSF_Type*)((void*)pHWStackFrame
                                    - sizeof(SWSF_Type));


  // populate HW Stack Frame
  pHWStackFrame->r0   = 0;
  pHWStackFrame->r1   = 0;
  pHWStackFrame->r2   = 0;
  pHWStackFrame->r3   = 0;
  pHWStackFrame->r12  = 0;
  pHWStackFrame->lr   = 0xFFFFFFFF; // (reset val - task never exits)
  pHWStackFrame->pc   = (unsigned int) (pTCB->pTaskFunction);
  pHWStackFrame->psr  = 0x01000000;   // Set T bit (bit 24) in EPSR.
                                      // The Cortex-M4 processor only
                                      // supports execution of
                                      // instructions in Thumb state.
                                      // Attempting to execute
                                      // instructions when the T bit
                                      // is 0 (Debug state)
                                      //    results in a fault.
  // populate SW Stack Frame
  pSWStackFrame->r4 = 0x04040404;
  pSWStackFrame->r5 = 0x05050505;
  pSWStackFrame->r6 = 0x06060606;
  pSWStackFrame->r7 = 0x07070707;
  pSWStackFrame->r8 = 0x08080808;
  pSWStackFrame->r9 = 0x09090909;
  pSWStackFrame->r10  = 0x0a0a0a0a;
  pSWStackFrame->r11  = 0x0b0b0b0b;

  // Set task's stack pointer in the TCB to point at the top
    // of the task's SW stack frame
  pTCB->sp      = (unsigned int*) pSWStackFrame;

}
```

Listing 1.17: The function `TaskInit()` that creates a new task.

The only parameter of the above `TaskInit()` function is a pointer to a task's TCB. The `TaskInit()` 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 1.18: 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 in 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 1.18 illustrates the memory layout and the contents of the task's stack and TCB after creating Task1 using the `TaskInit()` function.

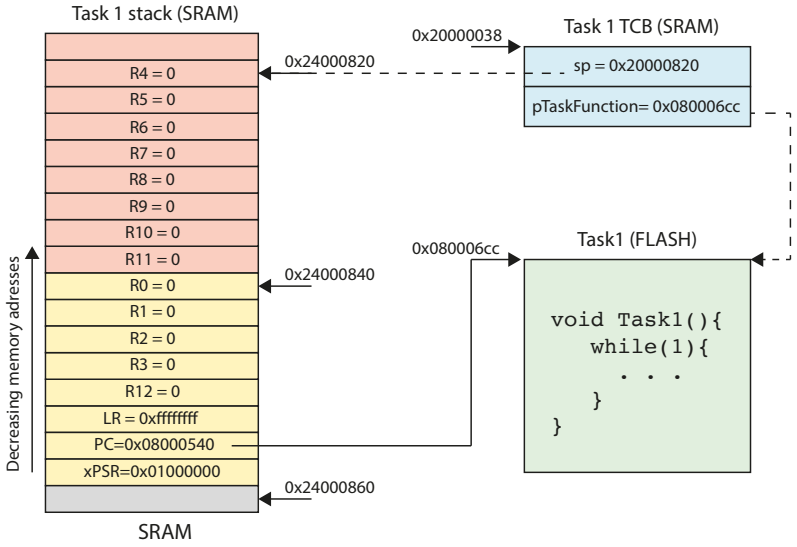

Fig. 1.18: Memory layout and content after calling the `TaskInit()` function.

1.4.7.7 Scheduler initialisation

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

```
void InitScheduler(unsigned int* pStackRegion,
                   TCB_Type pTCB[],
                   void (*TaskFunctions[])()){
  unsigned int* pTaskStackBase;

  // 1. create all tasks:
  for(int i=0; i<NTASKS; i++){
    pTaskStackBase = pStackRegion + (i+1)*TASK_STACK_SIZE;
    TaskCreate(&pTCB[i], pTaskStackBase, TaskFunctions[i]);
  }
  // 2. initialize all tasks except the Task0.
  //    Task0 will be called by main()
  //    and will be the first task interrupted.
  //    Its HWSF and SWSF will be created upon
  //    interrupt/contecxt switch
  for(int i=1; i<NTASKS; i++){
    TaskInit(&pTCB[i]);
  }
  // set PSP to Task0.SP:
  __set_PSP((unsigned int)pTCB[0].sp);
}
```

Listing 1.19: The function `InitScheduler()` creates all tasks and initializes all tasks except the firt one (Task0). At the end, it sets the top of the stack of the first task (Task0) into the PSP register.

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

1. Creates all tasks.
2. Initializes all tasks except the first one (Task0). Task0 will be called from the main function and will be the first task interrupted by the SysTick timer. Hence, its stack frames will be populated during the context switch.
3. Saves the top of the stack of the first task (Task0) into the PSP register.

To read or write the PSP register, which is not memory-mapped, requires the usage of special CPU instructions `MSR` and `MRS`. Hence, in order to access the PSP register, we are forced to use assembly. To make programming easier, the above code relies on the `__set_PSP` function defined in the Cortex Microcontroller Software Interface Standard (CMSIS) library to write intio the PSP register. CMSIS is a vendor-independent hardware abstraction layer (HAL) for ARM Cortex-M processors. It simplifies software development for a wide range of microcontroller devices, promoting code portability and reusability across various microcontroller families and manufacturers. CMSIS defines two inline assembly functions to read or write the PSP register:

```
/**
  \brief   Set Process Stack Pointer
  \details Assigns the given value to the Process Stack Pointer (PSP)
  \param [in]    topOfProcStack  Process Stack Pointer value to set
 */
__attribute__((always_inline))
static inline void __set_PSP(uint32_t topOfProcStack)
{
  __asm volatile ("MSR psp, %0" : : "r" (topOfProcStack) : );
}

/**
  \brief   Get Process Stack Pointer
  \details Returns the current value of the Process Stack Pointer (PSP)
  \return  PSP Register value
 */
__attribute__((always_inline))
static inline void uint32_t __get_PSP(void)
{
  uint32_t result;

  __asm volatile ("MRS %0, psp"  : "=r" (result) );
  return(result);
}
```

Listing 1.20: The CMSIS definition of inline assembly functions for accessing the PSP register.

After these steps, everything is set up for the first context switch. Figure 1.19 illustrates the memory layout and the task's stack after initializing scheduler using the `InitScheduler()` function.

1.4.7.8 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

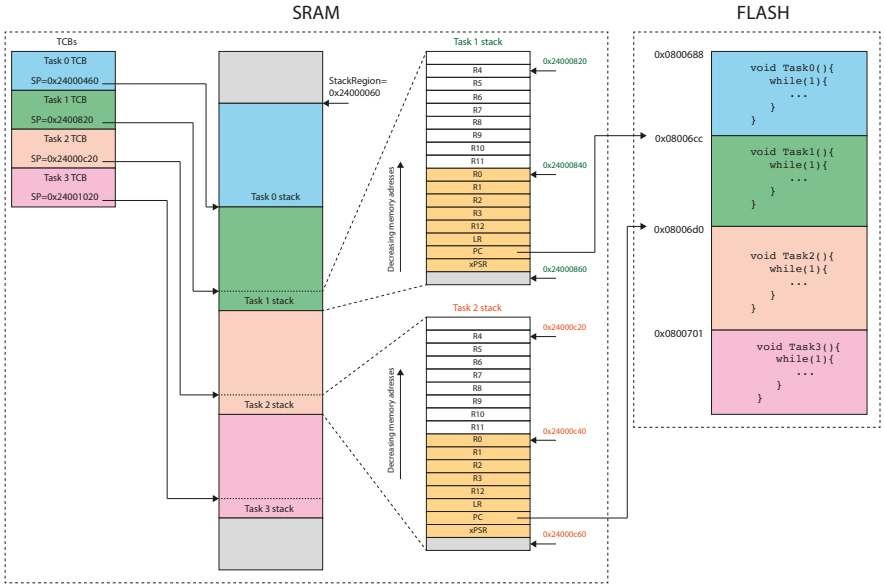

Fig. 1.19: Memory layout and content after creating four tasks.

its stack and then loading the context of the next task to be executed by swapping the SP. Figure 1.21 shows the process of context switching using stack pointer swapping. Here's a step-by-step description of how context switching is accomplished

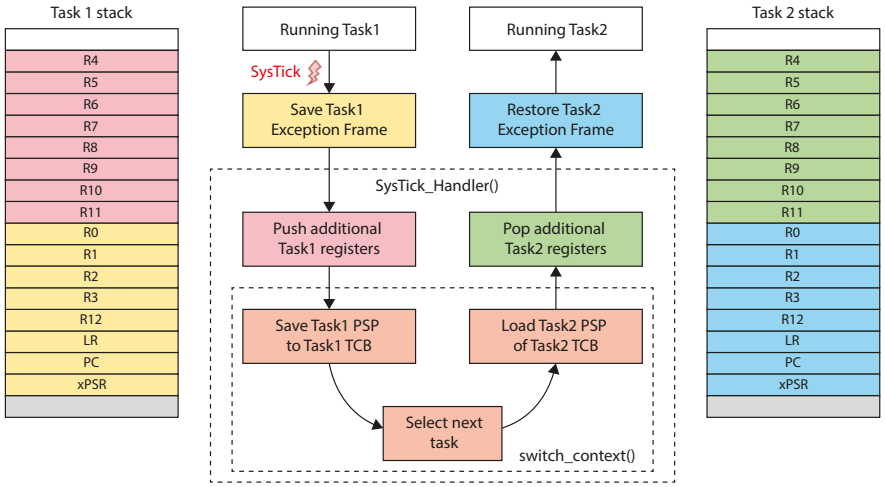

Fig. 1.20: Context switching using stack pointer swapping.

using this method:

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 1.21 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 ContextSwitch(int current_task, TCB_Type pTCB[]){
  volatile int new_task;

  pTCB[current_task].sp = (unsigned int*) __get_PSP();

  // select next task in round-robin fashion
  new_task = current_task + 1;
  if (new_task == NTASKS) new_task = 0;

  __set_PSP((unsigned int)pTCB[new_task].sp);

  return new_task;
}
```

Listing 1.21: The functions `ContextSwitch()` that implements context switching.

The parameters of the `ContextSwitch` 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.

1.4.7.9 SysTick handler

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

```
void SysTick_Handler(void)
{
  unsigned int tmp;
  // 1. Save context of the interrupted task:
  if (current_task != -1){
    __asm__ volatile ( "MRS %0, psp\n\t"
```

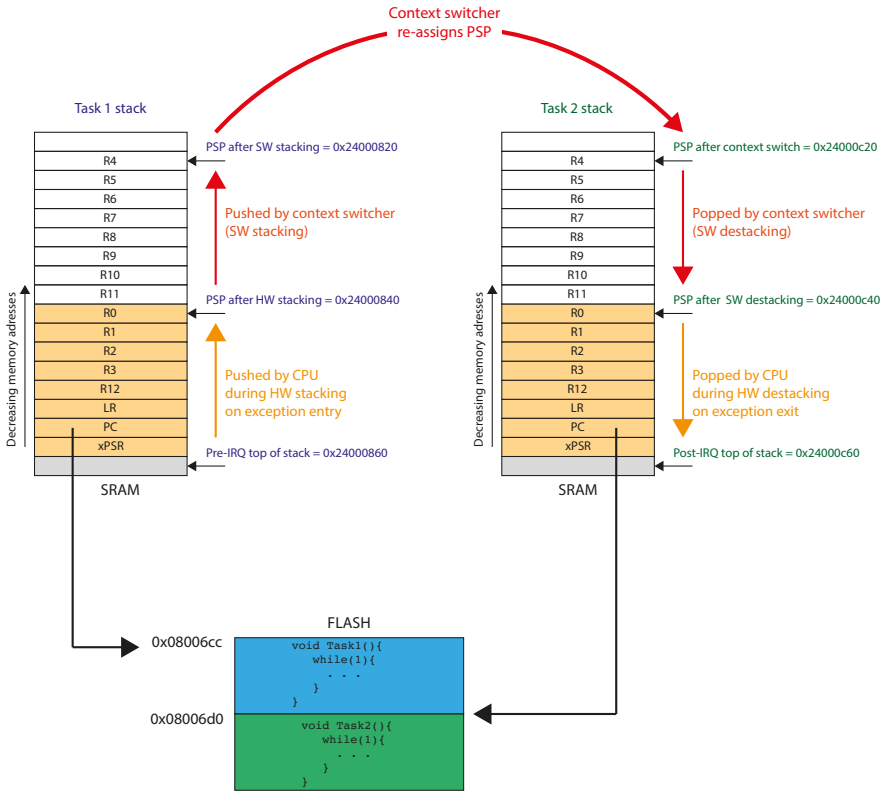

Fig. 1.21: The modification progress of the PSP stack pointer during context switching.

```
7               "STMFD %0!, {r4-r11}\n\t"
                "MSR psp, %0\n\t" : "=r" (tmp) );
9
11      // 2. Switch context:
        current_task = ContextSwitch(current_task, TCB);
13
        // 3. restore context of the new task:
15      __asm__ volatile ( "MRS %0, psp\n\t"
                           "LDMFD %0!, {r4-r11}\n\t"
17                         "MSR psp, %0\n\t" : "=r" (tmp) );
    }
19  }
```

Listing 1.22: The SysTick handler used to perform task switch.

The SysTick handler performs the following steps:

1. Saves the context (R4-R11) of the interrupted task on the task's stack using PSP.
2. Switch context (swap stack pointers) using the textttswitch_context() function.

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

1.4.7.10 Starting the scheduler

Finally, we are ready to start our scheduler within the main function. To do so, we need to:

1. Initialize scheduler.
2. Switch to NOT PRIVILEGED mode with PSP as the stack pointer by setting the last two bits in the CONTROL register.
3. Call Task0.
4. Within Task0, wait for the first SysTick interrupt.

The following code shows how to start the scheduler:

```
1
unsigned int stackRegion[NTASKS * TASK_STACK_SIZE];
3 TCB_Type TCB[NTASKS];       // Array of Tasks Control Blocks
void (*TaskFunctions[NTASKS])();  // Tabela naslovov funkcij opravil
5 int current_task = -1;

7 void Task0(){
  while(1) {}
9 }
void Task1(){
11   while(1) {}
}
13 void Task2(){
  while(1) {}
15 }
void Task3(){
17   while(1) {}
}
19

21 int main(void)
{
23   TaskFunctions[0] = Task0;
  TaskFunctions[1] = Task1;
25   TaskFunctions[2] = Task2;
  TaskFunctions[3] = Task3;
27   // Init scheduler:
  InitScheduler(stackRegion, TCB, TaskFunctions);
29   current_task = 0;
  // Start SysTick timer with the highest priority:
31   HAL_InitTick(0);
  // Switch to NOT PRIVILEDGED with PSP:
33   __set_CONTROL(0x00000003);
  // Call the first task:
35   Task0();  // never return!
  while(1){}
37 }
```

Listing 1.23: Starting the scheduler.

To write into the CONTROL register (which is not memory-mapped), the above code uses the `__set_CONTROL` function defined in the CMSIS library as:

```
/**
  \brief   Set Control Register
  \details Writes the given value to the Control Register.
  \param [in]    control  Control Register value to set
 */
__STATIC_FORCEINLINE void __set_CONTROL(uint32_t control)
{
  __ASM volatile ("MSR control, %0" : : "r" (control) : "memory");
}
```

Listing 1.24: The CMSIS definition of inline assembly function for writing into the CONTROL register.

1.4.7.11 Using PendSV for context switching

The approach with the SysTick handler used to perform the context switching would, however, not work with other interrupts (peripheral interrupts, for example). The SysTick handler would interrupt IRQ handlers as well, and stack registers affected by the peripheral IRQ handler and unstack task's registers, resulting in undefined behaviour of both tasks and peripheral interrupt handler. This would undoubtedly result in the hard fault.

The PendSV (Pending Supervisor Call) interrupt is commonly used for context switching in ARM Cortex-M microcontrollers due to several advantages and characteristics that make it well-suited for this purpose. The PendSV interrupt has the lowest possible priority among all exceptions and interrupts. This makes it an ideal choice for context switching, as it doesn't interfere with other higher-priority interrupts or exceptions. The PendSV exception will interrupt only the non-priority tasks and certainly not any exception handler. The low-priority nature of PendSV ensures that it doesn't preempt other exceptions or interrupts, providing predictable and deterministic behaviour during context switches. This predictability is essential in real-time systems. PendSV can be triggered explicitly through software by setting the PendSV bit in the ICSR register within the System Control Block. This allows for precise control over when context switches occur. Typically, the PendSV interrupt is set pending from the SysTick handler.

Figure 1.22 shows the solution to this problem with the PendSV interrupt. Usually, the SysTick interrupt has the highest priority among all exceptions and interrupts with configurable priority. If an interrupt request (IRQ) takes place before the SysTick exception, the SysTick exception might preempt the IRQ handler. In this case, we should not carry out the context switching. The PendSV exception solves the problem by delaying the context-switching request until all other IRQ handlers have completed their processing. To do this, the PendSV is programmed as the lowest-priority exception. The Systick handler sets the pending status of the

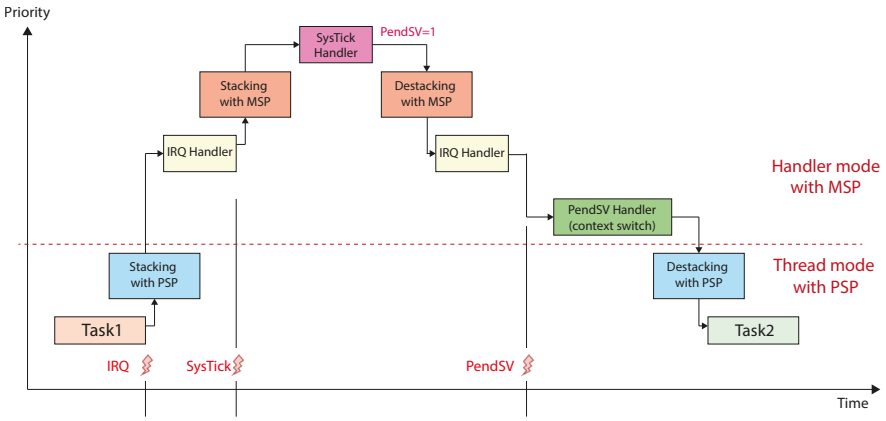

Fig. 1.22: A simple task scheduler based on PendSV interrupts.

PendSV, and the context switching is carried out within the PendSV exception. Let us describe the solution in Figure 1.22:

1. Task1 is preempted by an IRQ interrupt request.
2. Task1's hardware stack frame is stacked on the process stack using the PSP.
3. The IRQ handler executes.
4. The SysTick exception eventually preempts the IRQ handler.
5. The ardware stack frame of the IRQ handler is stacked on the main stack using the MSP register.
6. The SysTick handler sets the PendSV bit. Hence, PendSV interrupt is pending.
7. The SysTick exits, and the hardware stack frame of the IRQ handler is popped from the main stack.
8. The IRQ handler continues its execution.
9. The IRQ handler exits, and the PendSV interrupt is taken immediately.
10. PendSV handler performs the context switching.
11. PendSV handler exits and the Task2's hardware stack frame is popped from the process stack using the PSP.
12. Task2 executes.

Hence, the solution to implement a scheduler based on the SysTick and PendSV exceptions is simple. Firstly, we move the code for context switching from the SysTick handler into the PendSV handler:

```
void PendSV_Handler(void)
{
  unsigned int tmp;
  // 1. Save context of the 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) );
```

```
9
11      // 2. Switch context:
        current_task = ContextSwitch(current_task, TCB);
13
        // 3. restore context of the new task:
15      __asm__ volatile ( "MRS %0, psp\n\t"
                           "LDMFD %0!, {r4-r11}\n\t"
17                         "MSR psp, %0\n\t" : "=r" (tmp) );
    }
19  }
```

Listing 1.25: Starting the scheduler.

Secondly, the SysTick handler only sets PendSV pending in the ICSR register:

```
1   void SysTick_Handler(void)
    {
3     // Set the PendSV Pending bit in  ICSR:
      SCB->ICSR |= (unsigned long)0x01 << 28;
5   }
```

Listing 1.26: The SysTick handler only sets PendSV pending.

The code for the scheduler can be found here:
https://github.com/bulicp/ContextSwitchM7-book.git.

1.5 ARM 9 exceptions and interrupts

The ARM9 supports the following six types of interrupts and exceptions:

- Fast interrupt Request,
- Interrupt Request,
- Data and Prefetched abort exceptions,
- Undefined instruction exception, and
- Software interrupt, and
- Reset.

The interrupt instruction SWI raises the software interrupts. The software interrupts allow a program running in the user mode to request privileged operations such as OS functions. The Prefetch abort exception occurs when the CPU fetches an instruction from an illegal address. The Data abort exception occurs when a data transfer instruction attempts to load or store data at an illegal address. The Undefined instruction exception occurs when the processor cannot recognize the currently fetched instruction. The Interrupt request occurs when the processor's external interrupt request pin (IRQ) is asserted (LOW), and the interrupt mask bit (I) in the current program status register (CPSR) is cleared (interrupts enabled). The Fast interrupt request occurs when the processor's external fast interrupt request pin (FIQ) is asserted (LOW), and the interrupt mask bit (F) in the current program status register (CPSR) is cleared (fast interrupts enabled). The Reset interrupt occurs when the processor's reset pin is asserted.

1.5.1 Vector table and interrupt priorities

Table 1.6: ARM9 vector table.

Interrupt/Exception	**Vector Table Address**	**Priority (1-High, 6-Low)**
Reset	0x00000000	1
Undefined Instruction	0x00000004	6
Software Interrupt	0x00000008	6
Prefetch Abort	0x0000000C	5
Data Abort	0x00000010	2
Interrupt Request	0x00000018	4
Fast Interrupt Request	0x0000001C	3

ARM9 processors use the vectored interrupt handling method. Each interrupt/exception has its own entry in the vector table. Each entry in the vector table has only

32 bits, which is not enough to contain the full code for a handler; hence, each entry commonly contains a branch instruction or load pc instruction to the actual handler. Table 1.6 shows the interrupt/exception, its address in the vector table, and its priority. As interrupts/exceptions can coincide, the CPU has to use a priority mechanism to handle the most important interrupt/exception. For example, the Reset interrupt has the highest priority, and it takes precedence over all other interrupts/exceptions. All interrupts/exceptions disable further interrupts/exceptions by setting the I bit in the CPSR register. The Reset and Fast Interrupt Request also set the F bit in the CPSR register and thus mask the Fast interrupt request. Listing 1.27 shows a typical method of implementing a vector table for ARM9 processors.

```
        .org 0x00000000
Vector_Table:
        b Reset_Handler
        b Undefined_Handler
        b SWI_Handler
        b Prefetch_Handler
        b Abort_Handler
        nop                     // never used
        b IRQ_Handler
        b FIQ_HAndler


Reset_Handler:
        <handler instructions>
Undefined_Handler:
        <handler instructions>
SWI_Handler:
        <handler instructions>
Prefetch_Handler:
        <handler instructions>
Abort_Handler:
        <handler instructions>
IRQ_Handler:
        <handler instructions>
FIQ_Handler:
        <handler instructions>
```

Listing 1.27: ARM vector table and interrupt handlers.

Listing 1.27 shows a typical method of implementing a vector table for ARM9 processors. The vector table starts at the address 0x00000000. Each entry in the vector table is 32 bits long and contains a branch instruction (B) to the interrupt handler. When, for example, a Data Abort exception occurs, the CPU stops the execution of the current running program, saves the program context, and moves the vector `0x00000010` into the program counter. This way, the `b Abort_Handler` instruction is fetched, and the CPU jumps to `Abort_Handler`.

As we already said, the Reset interrupt is the highest priority interrupt and is always taken whenever the Reset pin is asserted. The reset handler is responsible for initializing the system and other interrupt sources, and to set the stack pointer. So the Reset interrupt masks automatically all other interrupts before their sources are initialized. Only then the reset handler enables other interrupts. Hence, during the first few instructions of the reset handler, we should avoid SWI, undefined instructions, and memory accesses that can cause the Data and Prefetch aborts.

The Fast Interrupt Request (FIQ) occurs when a peripheral asserts the processor's FIQ pin. The peripheral device mus hold the FIQ input low until the processor acknowledges the interrupt request. As a response to FIQ, the CPU disables both Interrupt and Fast Interrupt requests. Hence, no external device can interrupt the CPU unless the IRQ and FIQ interrupts are re-enabled by software. The Fast Interrupt Request reduces the execution time of the exception handler relative to a normal interrupt by removing the requirement for register saving (minimizing the overhead of context switching).

The Interrupt Request (IRQ) is a normal interrupt that occurs when a peripheral device asserts the IRQ pin. The peripheral device mus hold the IRQ input pin low until the processor acknowledges the interrupt request. An IRQ has a lower priority than the FIQ and Data Abort and is masked on entry to an FIQ or Data Abort sequence. On entry to the IRQ handler, the further IRQ interrupts are disabled and should remain disabled until the current interrupt source has been acknowledged, and the IRQ pin has been de-asserted.

We can notice from Table 1.6 that both Software Interrupt and Undefined Instruction have the same level of a priority since they cannot occur at the same time.

1.5.2 ARM9 interrupt handling

ARM9 processors are 5-stage pipelined machines with Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM) and Write-Back (WB) stages. In a pipelined machine, an instruction is executed step by step and is not completed for several clock cycles. An external interrupt can occur at any time during the execution of an instruction. Also, other instructions in the pipeline can raise exceptions that may force the machine to abort the instructions in the pipeline before they have been completed. One of the problems with interrupts in the pipelined CPUs is when to halt instruction in the pipeline. In the case of external interrupts, one possible solution would be to execute all fetched instructions before handling the interrupt request. But the problem with this approach would be a long interrupt latency. The other solution would be to halt the execution of all fetched instructions and fetch them again upon returning from the interrupt handler. This way, we would have minimal interrupt latency. Obviously, this is not a good idea because some instructions, such as STORE instructions, can modify the content in memory and should not be stoped and executed again. Also, arithmetic instructions might have already changed the content of the status register (usually in the Execution stage), and should not be dismissed. The most common solution to the problem is to execute all instructions that have been issued into the execution stage. In the case of an external interrupt in ARM9, the CPU executes all instructions in the stages EX, MEM nad WB, while dismissing two instructions in the stages IF and ID.

To resume, in the case of an external interrupt, the CPU has to let all instructions that were issued for execution complete and flush all succeeding instructions from the pipeline. In the case of an exception caused by an instruction, the CPU should

stop executing the offending instruction, let all preceding instructions complete and flush all succeeding instructions from the pipeline. Only then can CPU start saving the context and fetching the instruction pointed by the interrupt vector (the first instruction in the interrupt handler).

Let us now look at how ARM9 handles the IRQ interrupts. When an IRQ interrupt occurs, the ARM 9 processor executes the three instructions that are issued for execution and will flush the last two fetched instructions. The last two fetched instructions are from the addresses PC (the instruction currently in the IF stage) and PC-4 (the instruction currently in the ID stage). The instruction in the EX stage is from the address PC-8. This is very important to notice because the last executed instruction before entering the interrupt handler was from the address PC-8, but the program counter contains the address PC. The first instruction to execute upon returning from the interrupt handler is one that was in the ID stage when the interrupt request occurs. Hence, the address of the instruction that should be fetched upon returning from the interrupt handler is PC-4.

When an IRQ interrupt occurs, the ARM9 processor executes the instructions that are issued for execution. Then, the following hardware procedure is executed:

- the CPU saves the Current Program Status (CPSR) register into the Saved Program Status (SPSR) register; hence the processor automatically saves the status of the interrupted program. The CPSR register is a special purpose register in ARM9 processors that contains arithmetic flags and interrupt masks,
- the CPU automatically disables interrupts by setting the I bit in the CPSR register,
- the CPU saves the current program counter (PC) into the link register (LR). This way, the LR register holds the return address. It is important to note that the CPU saves the address of the last fetched instruction and does not automatically correct this value to point to the instruction that was in the ID stage when the interrupt occurs. Hence, it is the programmer's responsibility to adjust the value in PC upon returning from the interrupt handler, and
- the CPU fetches the instruction from the interrupt vector 0x00000018.

Now, the interrupt handler starts. The above procedure is hard-wired in the CPU and does not involve any instruction fetch and execution. When an interrupt handler has completed, it must move both the return value in the LR register minus 4 to the PC and the SPSR to the CPSR. This action restores both the PC and the CPSR and returns to the interrupted program. Listing 1.28 shows a typical method of returning from an IRQ interrupt handler.

```
1 IRQ_Handler:
2         <handler instructions>
3         ...
4         ...
5         subs pc, lr, #4      //  pc <- lr-4
```

Listing 1.28: A typical IRQ interrupt handler

Many instructions in ARM9 can have an "s" suffix. The "s" suffix ensures that when the program counter is the destination register, the CPSR register is automatically restored from the SPSR register. The same holds for the `subs` instruction in Listing 1.28. Hence, the instruction `subs pc,lr,#4` firstly saves the LR-4 into the program counter (remember that the programmer is responsible to correctly restore the return address into the program counter upon returning from the handler) and then restores CPSR from SPSR.

It is important to stress that not all interrupt/exception handlers use the same instruction to return. For example, the Data abort exception occurs in the MEM stage. Hence, only the instruction in WB stage is executed, while the instructions from IF, ID, and EX stages are flushed. When the Data abort exception occurs, the instruction in the EX stage is from the address PC-8. Thus, the Data abort handler uses `subs pc,lr,#8` to return:

```
IRQ_Handler:
        <handler instructions>
        ...
        ...
        subs pc, lr, #8     //  pc <- lr-8  !!!!
```

Listing 1.29: A typical Data abort exception handler

1.5.3 Interrupt handlers in C

Interrupt handlers can be written in assembler or in a high-level language like C. Usually, we want to avoid the assembly language as much as possible and to program in our favorite high-level language. Remember that an interrupt handler is called directly by the CPU, and the protocol for calling an interrupt handler differs from calling a C function. Most importantly, an ISR has to end with some "interrupt return" opcode, whereas usual C functions end with ordinary "return" opcode. We have seen previously that the ARM interrupt handlers should return with SUBS opcode, which is used to restores the PC from LR-4 and CPSR from SPSR. In the case of an ordinary subroutine, the return opcode for ARM would be MOV PC, LR (restores PC from LR). A programmer could be tempted to write an interrupt handler like this:

```
/* How NOT to write an interrupt handler */
void my_interrupt_handler(void)
{
    /* do something */
}
```

Listing 1.30: How not to write an interrupt handler.

This simply cannot work. The compiler doesn't understand that this is to be an interrupt handler and that the SUBS PC,LR,#4 instruction should be the last instruc-

tion used to return. The compiler will simply use the MOV PC, LR instruction to return.

Some compilers, such as GCC, Clang, and ARMCC, to name a few, have directives like #pragma or special function attributes, allowing you to declare a routine interrupt. For example, the *interrupt* function attribute in GCC indicates that the specified function is an interrupt handler. The compiler then generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

The correct (GCC) way of implementing an interrupt handler in C is:

```
/* GCC style interrupt handler */
__attribute__((interrupt)) void my_interrupt_handler()
{
    /* do something */
}
```

Listing 1.31: GCC style interrupt handler.

The ARMCC compiler offers the `__irq` function declaration keyword to write C interrupt handlers. The `__irq` keyword preserves all registers used by the interrupt handler and exits the handler by setting the PC to (LR–4) and restoring the CPSR to its original value from SPSR. Also, if the kernel calls a subroutine, `__irq` preserves the link register (LR), which is corrupted by the subroutine call.

```
/* ARMCC style interrupt handler */
__irq void my_interrupt_handler()
{
    /* do something */
}
```

Listing 1.32: ARMCC style interrupt handler.

But it is not only the directive or function qualifier that designates the interrupt handlers. Often, compilers require that the handler declaration contains a special function argument, which specifies the kind of interrupt (for example, IRQ or Abort). The compiler uses this special argument to restore the PC from LR correctly (for example, LR-4 for IRQ or LR-8 for Data abort). All these attributes and arguments defined and used by a particular compiler prevent the handler code from being portable.

1.6 Intel interrupts

Intel processors have two external pins for external interrupts:

- INTR pin - it is used to signal for normal (maskable) interrupts.
- NMI pin - it is used to signal nonmaskable interrupts

Besides interrupts, Intel processors can detect exceptions from two sources:

- Processor exception - triggered form processor as a result of some exceptional conditions within the processor (e.g., divide by zero). These exceptions are further classified as faults, traps, and aborts.
- Software interrupts - triggered with the processor instruction INT.

Exceptions are classified as:

- Faults are either detected before the instruction begins to execute or during the execution of the instruction. A fault is an exception that can generally be corrected, and that, once corrected, allows the program to be restarted with no loss of continuity. The return address for the fault handler points to the faulting instruction, rather than to the instruction following the faulting instruction.
- A trap is an exception that is reported immediately following the execution of the instruction INT. Traps allow the execution of a program or task to be continued without loss of program continuity. The return address for the trap handler points to the instruction to be executed after the trapping instruction.
- An abort is an exception that does not allow a restart of the program or task that caused the exception. Aborts are used to report severe errors.

The Intel processor services interrupts and exceptions only between the end of one instruction and the beginning of the next. This is referred to as the instruction boundary. Certain conditions and flag settings cause the processor to inhibit certain interrupts and exceptions at instruction boundaries. The IF (interrupt-enable flag) bit in the FLAGS register (this is the status register in Intel x86 microprocessors that contains the current state of the processor.) controls the acceptance of external interrupts signaled via the INTR pin. When IF=0, INTR interrupts are masked; when IF=1, INTR interrupts are enabled. The Intel processor instructions CLI (Clear Interrupt-Enable Flag) and STI (Set Interrupt-Enable Flag) are used to clear/set the IF flag.

If more than one interrupt or exception is pending at an instruction boundary, the processor services one of them at a time according to their priority. In general, aborts have the highest priority, followed by traps, NMI, and INTR. The faults have the lowest priority.

Each architecturally defined exception and interrupt in Intel processors is assigned a unique identification number, called a **vector number**. The processor uses the vector number assigned to an interrupt as an index into the interrupt vector table. The allowable range for vector numbers is 0 to 255. The Intel architecture reserves vector numbers in the range 0 through 31 for architecture-defined exceptions and

interrupts. Vector numbers in the range 32 to 255 are designated as user-defined interrupts and are assigned to external I/O devices to enable those devices to send interrupts. One characteristic of Intel processors, which distinguish them from ARM processors, is that the peripheral device that caused an interrupt must provide the vector number to the CPU. Table 1.7 shows vector number assignments and exception types for architecturally defined exceptions and interrupts.

Table 1.7: Intel Exceptions and Interrupts. Only a few exceptions and interrupts are shown.

Vector Number	**Description**	**Type**
0	Division by zero	Fault
1	Debug	Fault
2	NMI	Interrupt
3	Breakpoint	Trap
...	...	...
	...	...
14	Page Fault	Fault
...	...	...
32-255	External interrupts on INTR	Interrupt

In the older Intel processors (before 80386), the interrupt table is called IVT (interrupt vector table). The IVT is an array of 32-bit interrupt vectors stored consecutively in memory and indexed by an interrupt vector. The IVT always resides at the same location in memory, ranging from 0x0000 to 0x03ff, and consists of 256 four-byte interrupt vectors (i.e. pointers to the interrupt/exception handles). When responding to an exception or interrupt, the processor multiplies the vector number by four to form the address of the entry in the IVT.

In modern Intel processors, the interrupt table is called IDT (interrupt descriptor table). The IDT is an array of 8-byte descriptors stored consecutively in memory and indexed by an interrupt vector. Each descriptor holds the information that describes how to access the interrupt/exception handler. The IDT may reside anywhere in physical memory. The processor has a special register (IDTR) to store both the physical base address and the length in bytes of the IDT. When an interrupt occurs, the processor multiplies the interrupt vector by eight and adds the result to the IDT base address. With the help of the IDT length, the resulting memory address is then verified to be within the table; if it is too large, an exception is generated. If everything is okay, the 8-byte descriptor stored at the calculated memory location is loaded, and actions are taken according to the descriptor's contents. As said, the interrupt descriptor table (IDT) associates each vector number with a descriptor for the instructions that service the associated event. Because there are only 256 vector numbers, the IDT contains up to 256 descriptors. It can contain fewer than 256 entries; entries are required only for vector numbers that are actually used.

The interrupt handling procedure in the Intel processor is rather complicated. Here, we omit all the details and give only the basic concepts. When responding to