Contents

1 Interrupts and interrupt handling 1
- 1.1 Introduction 1
- 1.2 Interrupts 2
  - 1.2.1 Types of interrupts 6
  - 1.2.2 Handling interrupts 6
- 1.3 ARM Cortex-M7 exceptions and interrupts 8
  - 1.3.1 ARM Cortex-M7 programmer's model 8
  - 1.3.2 System Control Block 10
  - 1.3.3 Exceptions 11
  - 1.3.4 Exception numbers and priorities 13
  - 1.3.5 Vector table and Exception handlers 15
  - 1.3.6 Exception entry and exit 17
  - 1.3.7 Case Study: A simple task scheduler on ARM Cortex-M7 . . 22
- 1.4 RISC-V interrupts and exceptions 41
  - 1.4.1 RISC-V Privileged Modes 42
  - 1.4.2 RISC-V Machine Modes Exceptions 43
  - 1.4.3 FE-310 Interrupts 47
  - 1.4.4 Interrupt Entry and Exit 49
  - 1.4.5 Implementing Vector Table and Handlers 50
  - 1.4.6 Case study: A simple task scheduler on RISC-V based FE310 57
- 1.5 ARM 9 exceptions and interrupts 69
  - 1.5.1 Vector table and interrupt priorities 69
  - 1.5.2 ARM9 interrupt handling 71
  - 1.5.3 Interrupt handlers in C 73
- 1.6 Intel interrupts 75
- 1.7 Interrupt controllers 77
  - 1.7.1 ARM Advanced Interrupt Controller 81
  - 1.7.2 RISC-V Platform-Level Interrupt Controller in FE310 84
  - 1.7.3 ARM Cortex-M Nested Vectored Interrupt Controller 89

1.7.4 Case study: External Interrupts in STM32H7xx Microcontrollers 92
1.7.5 Intel 8259A Programmable Interrupt Controler 99
1.7.6 8259A PIC Cascading 102
1.7.7 Intel Advanced Programmable Interrupt Controler 106
1.8 PCI interrupts 113
1.8.1 PCI Legacy interrupts 113
1.8.2 PCI interrupts routing 115
1.8.3 Message Signaled Interrupts 118

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 previous section, 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.

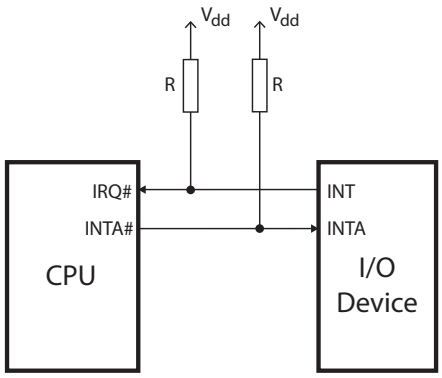

Fig. 1.1: A simplified block diagram of a computer system with interrupt controlling signals.

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

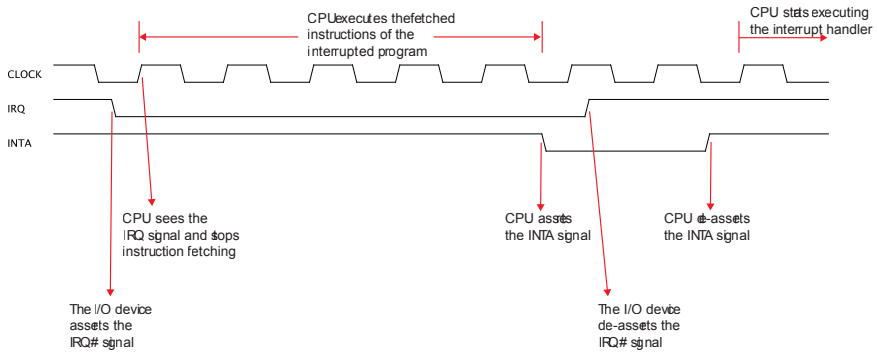

Fig. 1.2: A timing diagram for an external interrupt request.

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.

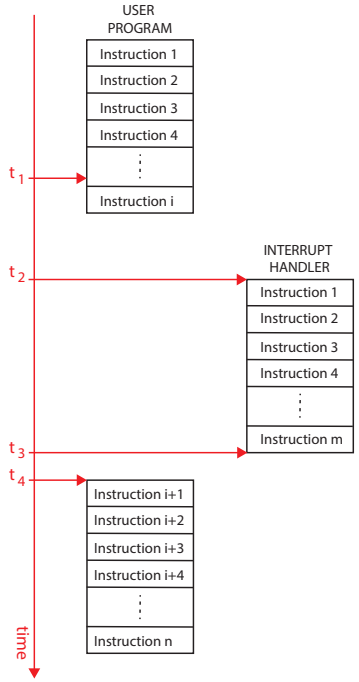

Fig. 1.3: The procedure involved in interrupts.

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 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.

1.3.1 ARM Cortex-M7 programmer's model

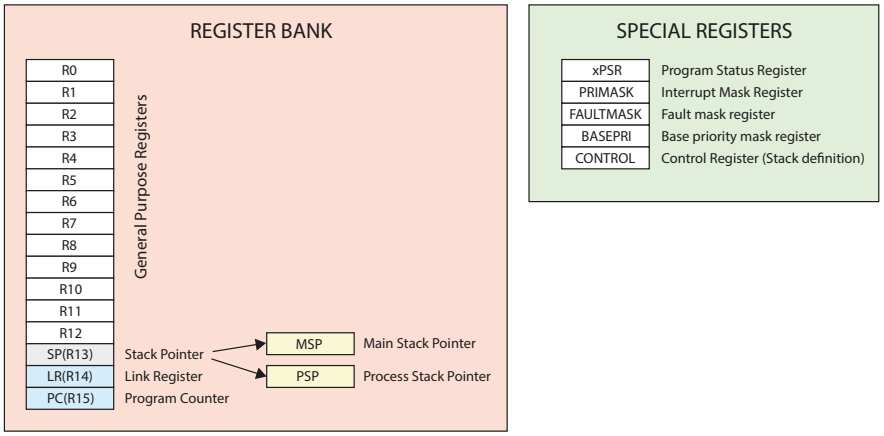

Fig. 1.4: ARM Cortex-M7 core registers.

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 1.4).

1.3.1.1 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.

1.3.1.2 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

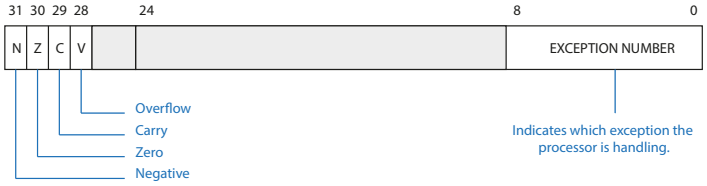

Fig. 1.5: xPSR register.

   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

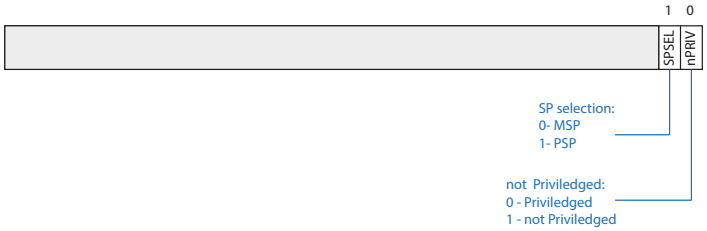

Fig. 1.6: CONTROL register.

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. 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.
   b. The FAULTMASK register is very similar to PRIMASK, but it also blocks the HardFault exception.
   c. 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.

1.3.2 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.

1.3.3 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.

1.3.4 Exception numbers and priorities

Table 1.1 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 1.1: 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.

ARM Cortex-M7 processors support a fixed-priority scheme where each interrupt source (or exception) can have a unique priority level assigned to it. **Each priority is associated with its priority value, where a lower priority value indicates a higher exception priority**. Cortex-M7 processors support up to 16 priority levels, where the value 0 represents the highest priority, and the value 15 represents the lowest priority. If the software does not configure any priorities, then all the exceptions with a configurable priority have a priority value of 0. A higher-priority (smaller priority level) exception can preempt a lower-priority (larger priority level)

exception. Some exceptions (reset, NMI, and HardFault) have fixed priority levels. Their priority levels are represented with negative values to indicate that they are of higher priority than other exceptions. The BASEPRI (Base Priority) register (Figure 1.7), which is a part of special registers in the ARM Cortex-M7 core registers block, provides a mechanism to set a threshold for exception priorities, allowing the processor to temporarily restrict the servicing of specific exceptions to prevent lower-priority interrupts from preempting critical tasks. The 4-bit BASEPRI field in the BASEPRI register defines a priority mask. The processor does not process any exception with a priority value greater than or equal to the value in the BASEPRI field.

Fig. 1.7: BASEPRI register.

1.3.5 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 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 1.8 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

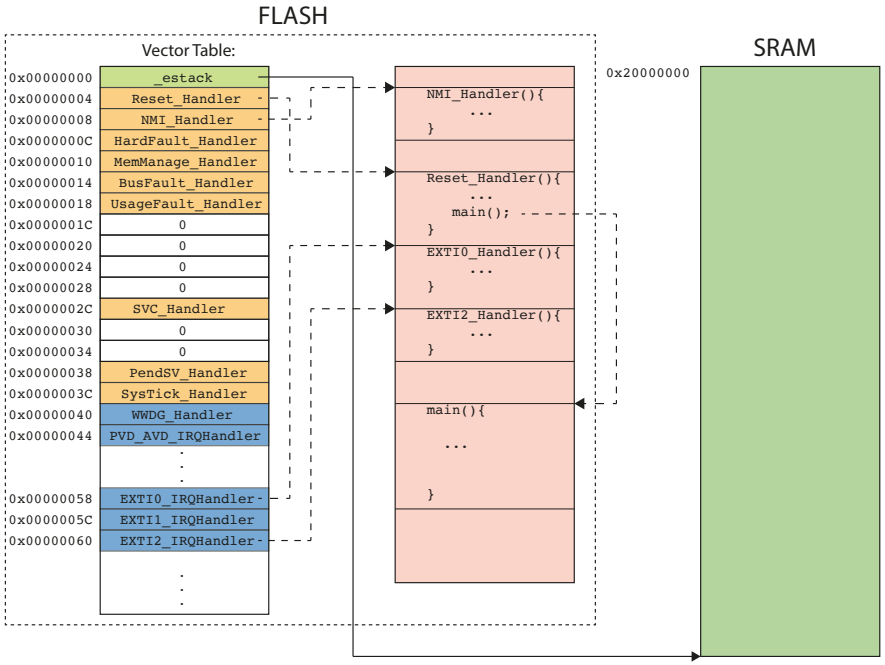

Fig. 1.8: The memory layout of the vector table and exception handlers in ARM Cortex-M7 cores.

name of the exception handlers in Figure 1.8 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 1.1: 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 1.2: Exception handlers in C.

1.3.6 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.

1.3.6.1 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 1.9 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 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).

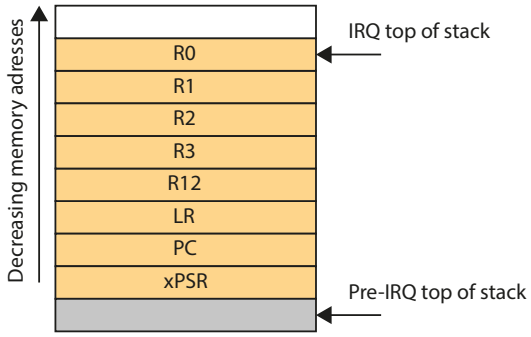

Fig. 1.9: The layout of the stack frame after stacking in ARM Cortex-M7.

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

In parallel to the stacking operation, the processor writes an **exception return value** (called EXC_RETURN value in the ARM documentation) to the link register (LR). This indicates which stack pointer corresponds to the stack frame and what operation mode the processor was in before the entry occurred. The information provided by the EXEC_RETURN value allows the processor to locate the stack frame to be restored upon returning from an exception, interpret it in the right way, and bring back the processor to the execution mode of the interrupted program. Table 1.2 shows the EXC_RETURN values and their meaning upon returning from an exception.

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

Table 1.2: 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.

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.

1.3.6.2 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 1.3: The exception handler for EXTI15_10 interrupt implemented as a regular C function.

1.3.7 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. The ability to seamlessly transition between multiple tasks, known as **context switching**, lies at the heart of efficient and responsive systems. Context switching refers to the process where the state of one task is saved, allowing another task to take precedence and execute. These tasks may be threads of a single application or various concurrent applications. At its core, context switching is a process by which the processor transitions from executing one task to another. 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. The context of each task includes the task's state of the processor—registers, program counter, stack pointer, and system variables.

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. A successful context switch involves the preservation of the current execution context and restoration of a new execution context, enabling the next task to resume from precisely where it left off. This process demands meticulous stack management and the precise handling of interrupts and exceptions.

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.

1.3.7.1 Background

A simple round-robin task scheduler (Figure 1.10) on Cortex-M7 processors effectively manages multiple tasks or threads in a cooperative multitasking environment. In this scheduler, each task is given a fixed time slice (quantum) during which it can execute. When its time slice expires, the scheduler switches to the next task in the queue. **The task scheduler relies on the interrupts and stacks to achieve context switching**. The SysTick and PendSV interrupts can both be used for context switching. The SysTick peripheral is a 24-bit timer that interrupts the processor each time it counts down to zero. This makes it well-suited to round-robin style context switching, and we are going to use the SysTick to perform a context switch.

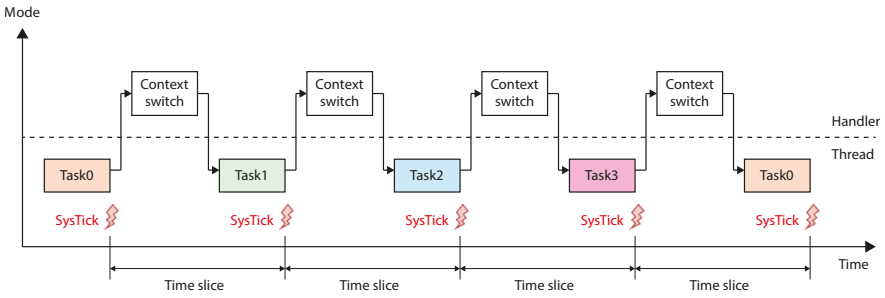

Fig. 1.10: A simple task scheduler.

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.

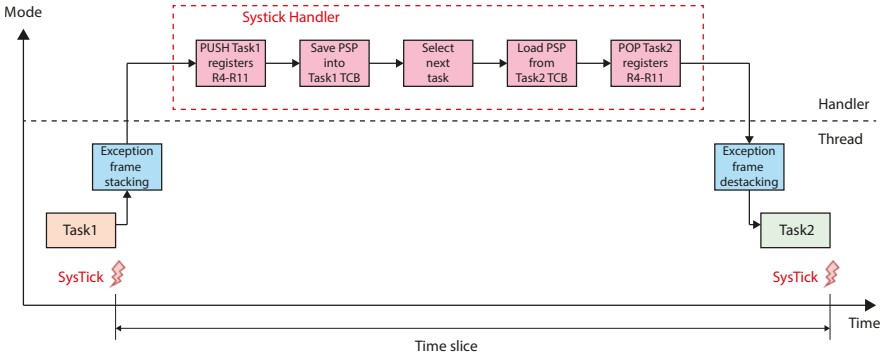

Fig. 1.11: A simple task scheduler.

Figure 1.11 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 SysTick 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.

1.3.7.2 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 1.4: A task in C. In our application, a task never returns and does not take any arguments.

1.3.7.3 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 1.5: Memory block for tasks' stacks. NTASKS equals 4 and TASK_STACK_SIZE equals 256.

1.3.7.4 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:

```
1 typedef struct{
    unsigned int *sp;
3   void (*pTaskFunction)();
```

```
} TCB_Type;
```

Listing 1.6: 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 1.7: TCB table. NTASKS is a constant equal to 4.

1.3.7.5 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.

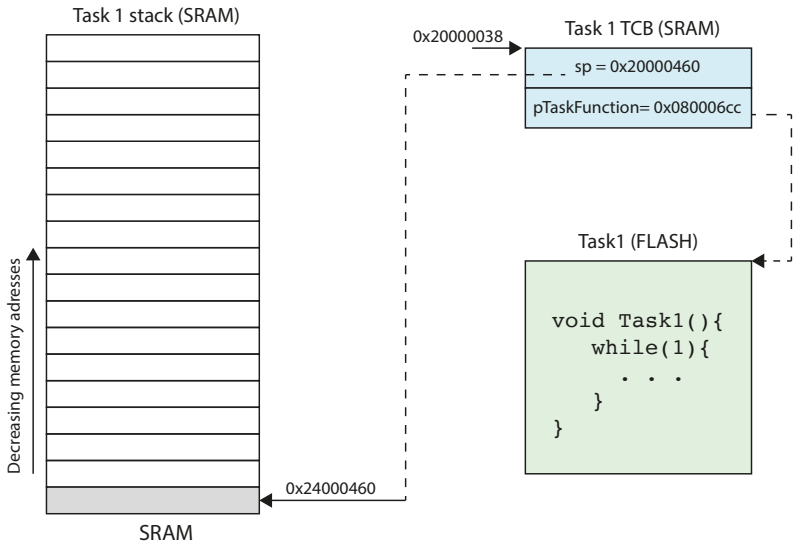

Fig. 1.12: Memory layout and content after calling the `TaskCreate()` function.

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

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

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

Listing 1.8: 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 1.12 illustrates the memory layout and the contents of the task's stack and TCB after creating Task1 using the `TaskCreate()` function.

1.3.7.6 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;
```

```
41 }
```

Listing 1.9: 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 1.10: 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 entry 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 1.13 illustrates the memory layout and the contents of the task's stack and TCB after creating Task1 using the `TaskInit()` function.

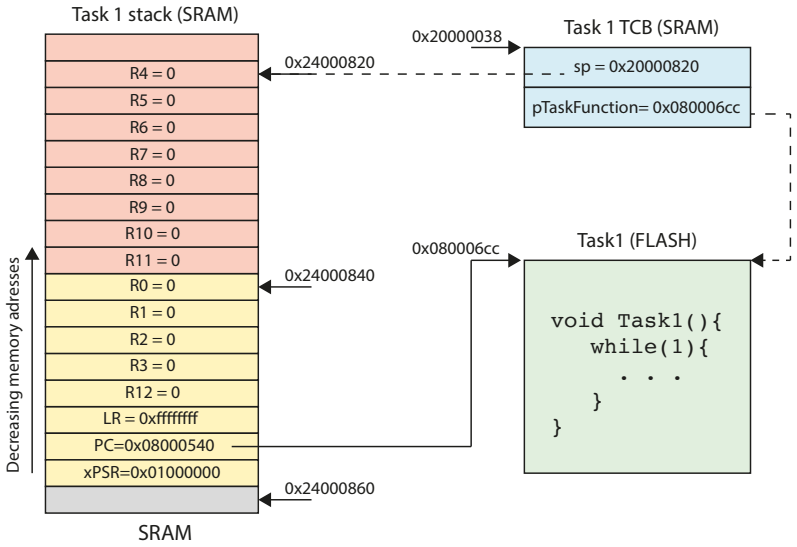

Fig. 1.13: Memory layout and content after calling the `TaskInit()` function.

1.3.7.7 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
```

```
15   //    interrupt/contecxt switch
     for(int i=1; i<NTASKS; i++){
17     TaskInit(&pTCB[i]);
     }
19   // set PSP to Task0.SP:
     __set_PSP((unsigned int)pTCB[0].sp);
21 }
```

Listing 1.11: The function `InitScheduler()` creates all tasks and initializes all tasks except the first 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 into 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:

```
1  /**
     \brief   Set Process Stack Pointer
3    \details Assigns the given value to the Process Stack Pointer (PSP)
     \param [in]    topOfProcStack  Process Stack Pointer value to set
5   */
  __attribute__((always_inline))
7  static inline void __set_PSP(uint32_t topOfProcStack)
  {
9    __asm volatile ("MSR psp, %0" : : "r" (topOfProcStack) : );
  }
11
  /**
13   \brief   Get Process Stack Pointer
     \details Returns the current value of the Process Stack Pointer (PSP)
15   \return  PSP Register value
   */
17 __attribute__((always_inline))
  static inline void uint32_t __get_PSP(void)
19 {
     uint32_t result;
21
     __asm volatile ("MRS %0, psp"  : "=r" (result) );
23   return(result);
  }
```

Listing 1.12: 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 1.14

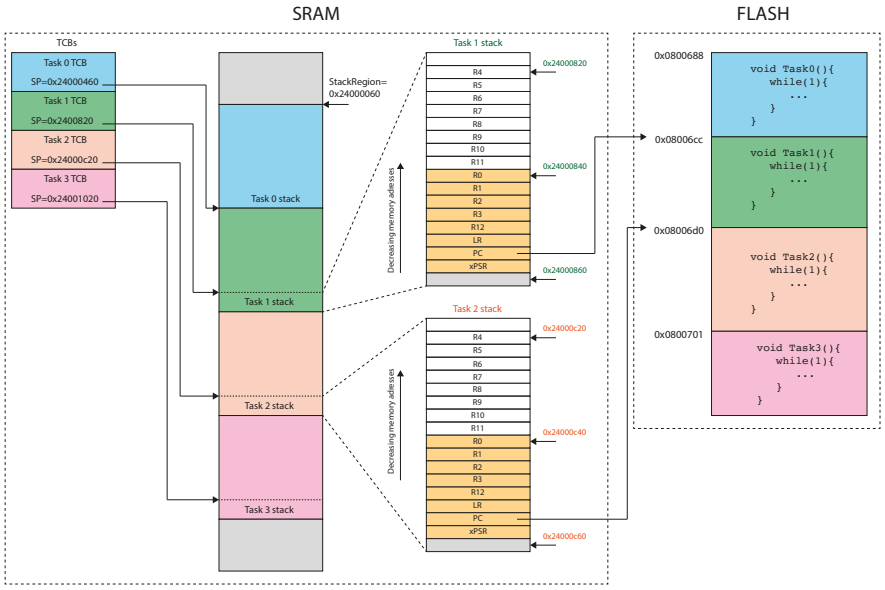

Fig. 1.14: Memory layout and content after initializing four tasks during the scheduler initialization.

illustrates the memory layout and the task's stack after initializing the scheduler using the `InitScheduler()` function.

1.3.7.8 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 its stack and then loading the context of the next task to be executed by swapping the SP. Figure 1.16 shows the process of context switching using stack pointer swapping. Here's a step-by-step description of how context switching is accomplished using this method:

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.

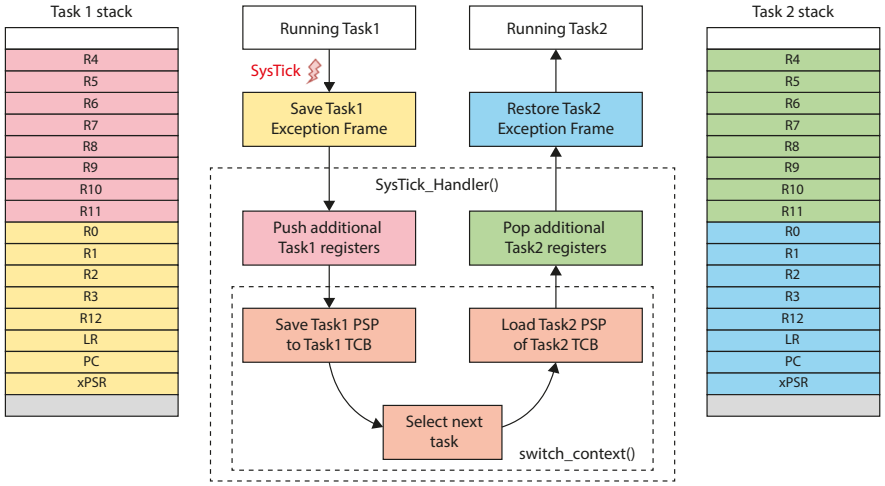

Fig. 1.15: Context switching using stack pointer swapping.

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 1.16 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;
}
```

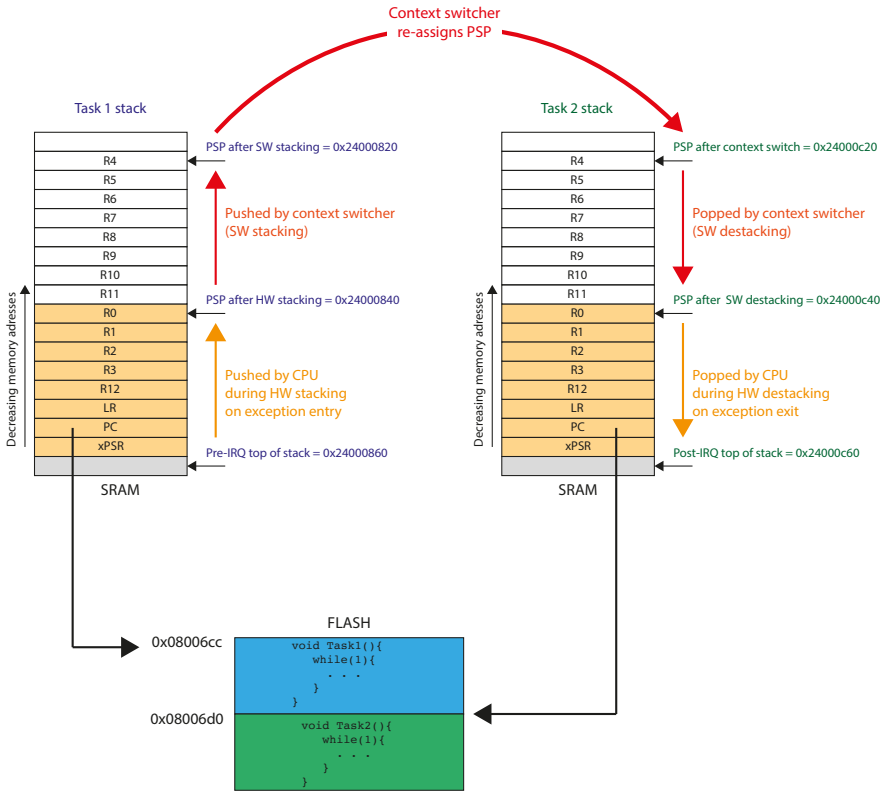

Fig. 1.16: The modification progress of the PSP stack pointer during context switching.

Listing 1.13: 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.

1.3.7.9 SysTick handler

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

```
1 void SysTick_Handler(void)
{
3   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) );


    // 2. Switch context:
    current_task = ContextSwitch(current_task, TCB);

    // 3. restore context of the new task:
    __asm__ volatile ( "MRS %0, psp\n\t"
                       "LDMFD %0!, {r4-r11}\n\t"
                       "MSR psp, %0\n\t" : "=r" (tmp) );
  }
}
```

Listing 1.14: 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.

1.3.7.10 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:

```

unsigned int stackRegion[NTASKS * TASK_STACK_SIZE];
TCB_Type TCB[NTASKS];
void (*TaskFunctions[NTASKS])();
int current_task = -1;

void Task0(){
  while(1) {}
}
void Task1(){
  while(1) {}
}
void Task2(){
  while(1) {}
}
void Task3(){
  while(1) {}
```

```
}

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

Listing 1.15: 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 1.16: The CMSIS definition of inline assembly function for writing into the CONTROL register.

1.3.7.11 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.

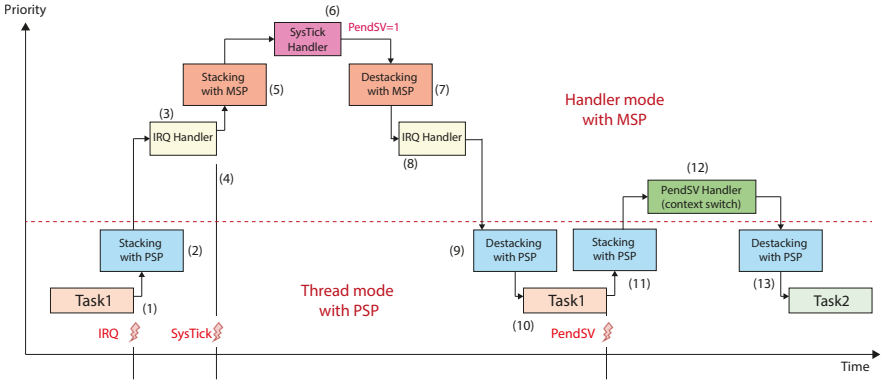

Fig. 1.17: A simple task scheduler based on PendSV interrupts.

Figure 1.17 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 PendSV, and the context switching is carried out within the PendSV exception. Let us describe the solution in Figure 1.17:

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 hardware stack frame of the Task1 is popped from the process stack.
10. The Task1 continues its execution.
11. PendSV is fired and the Task1's hardware stack frame is stacked on the process stack using the PSP.
12. PendSV handler performs the context switching.
13. PendSV handler exits and the Task2's hardware stack frame is popped from the process stack using the PSP.
14. 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)
{
  volatile unsigned int tmp1=0;
  volatile unsigned int tmp2=0;
  // 1. Save context of the interrupted task:
  __asm__ volatile ( "MRS %0, psp\n\t"
           "STMFD %0!, {r4-r11}\n\t"
           "MSR psp, %0\n\t" : "=r" (tmp1) );


  // 2. Switch context:
  current_task = ContextSwitch(current_task, TCB);

  // 3. restore context of the new task:
  __asm__ volatile ( "MRS %0, psp\n\t"
                     "LDMFD %0!, {r4-r11}\n\t"
                     "MSR psp, %0\n\t" : "=r" (tmp2) );
}
```

Listing 1.17: Starting the scheduler.

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

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

Listing 1.18: The SysTick handler only sets PendSV pending.

1.3.7.12 Using the Supervisor call (SVC) exception to start the scheduler

Instead of directly calling the first task (Task0) from the main function, the first task should be initialized and started in the same way as the others. In other words, the scheduler should rely on the exception return to start the first task. For this purpose, we can use the Supervisor Call (SVC) exception. Recall that the SVC instruction

triggers the SVC exception. Due to the interrupt priority behaviour of the Cortex-M processors, the SVC instruction can only be used in thread mode or exception handlers that have a lower priority than the SVC itself. Otherwise, a HardFault exception would be generated. The SVC instruction is a privileged operation that allows a task in an unprivileged mode to request a service from the operating system (or kernel) running in a privileged mode. This separation of privilege levels ensures that only trusted code can initiate scheduling or other system-related operations. Figure 1.18 shows the process of starting and running the scheduler using the SVC

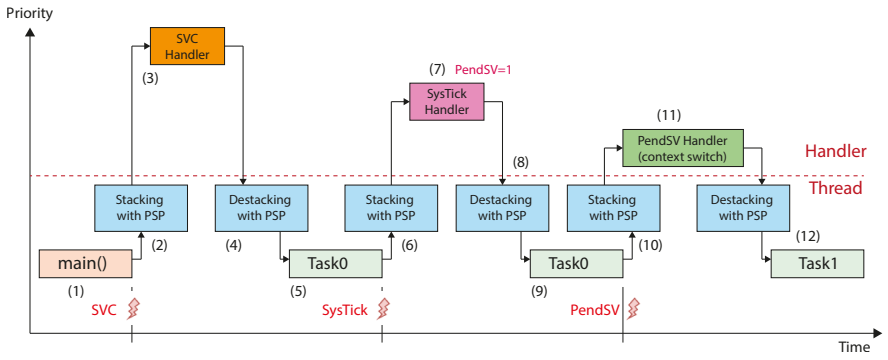

Fig. 1.18: Starting the scheduler with the SVC exception.

exception. Let us describe the solution in Figure 1.18:

1. The `main()` function initializes the scheduler (i.e., initializes all tasks) and eventually executes the SVC instruction.
2. The `main()` function is preempted by the SVC exception, and its hardware stack frame is stacked on the process stack using the PSP
3. The SVC handler sets the PSP to point to the top of the Task0 stack, restores the context (R4-R11) of the Task0 and exits.
4. Upon exception exit, the hardware stack frame of Task0 is restored, therefore returning control to Task0.
5. Task0 executes until the end of its time slot.
6. The SysTick exception preempts Task0, saving its hardware stack frame onto its stack.
7. The SysTick handler sets the PendSV bit. Hence, the PendSV interrupt is pending.
8. The SysTick exits, and the hardware stack frame of Task0 is popped from the main stack.
9. Task0 continues its execution.
10. PendSV is fired, and the Task0 hardware stack frame is stacked on the process stack using the PSP.
11. PendSV handler performs the context switching.

12. PendSV handler exits, and Task1's hardware stack frame is popped from the process stack using the PSP.
13. Task1 now executes.

To initialize the scheduler that uses the SVC exception to start the first task, we use the following function:

```
void InitSchedulerSVC(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
  //    The main() and will be first interrupted by SVC.
  //    Task0 will be entered from SVC Handler
  for(int i=0; i<NTASKS; i++){
    TaskInit(&pTCB[i]);
  }

  // set PSP to Task0.SP:
  __set_PSP((unsigned int)pTCB[0].sp);
}
```

Listing 1.19: The function `InitSchedulerSVC()` creates all tasks and initializes the stack frames for all tasks.

Contrary to the function `InitScheduler()`, the function `InitSchedulerSVC()` initializes the stack and both stack frames of all tasks. The SVC handler simply sets the PSP to point to the top of Task0's stack and restores the context (R4-R11) of the first task:

```
void SVC_Handler(void)
{
  /* We are here, because main() called SVC. As we interrupted main(),
   * there is no need to save its context.
   * We should never return to main()!!
   * The SVC_Handler should start the first task - Task0
   * The Task 0 is started by restoring its SW context and
   * its HW context upon the exception return.
   */
  // set PSP to Task0.SP:
  __set_PSP((unsigned int)TCB[0].sp);
  current_task = 0;
  // Restore the context of the Task 0:
  __RESTORE_CONTEXT();
}
```

Listing 1.20: The SVC Handler.

The following code shows how to start the scheduler:

```

```

```
unsigned int stackRegion[NTASKS * TASK_STACK_SIZE];
TCB_Type TCB[NTASKS];
void (*TaskFunctions[NTASKS])();
int current_task = -1;

void Task0(){
  while(1) {}
}
void Task1(){
  while(1) {}
}
void Task2(){
  while(1) {}
}
void Task3(){
  while(1) {}
}


int main(void)
{
  TaskFunctions[0] = Task0;
  TaskFunctions[1] = Task1;
  TaskFunctions[2] = Task2;
  TaskFunctions[3] = Task3;
  // Init scheduler:
  InitSchedulerSVC(stackRegion, TCB, TaskFunctions);
  // Start SysTick timer:
  HAL_InitTick(0);
  // Switch to NOT PRIVILEDGED with PSP:
  __set_CONTROL(0x00000003);
  // Start the scheduler:
  __asm volatile("svc 0");
  while(1){}
}
```

Listing 1.21: Starting the scheduler using the SVC exception.

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

1.4 RISC-V interrupts and exceptions

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.

1.4.1 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.

1.4.2 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 1.19. Note that this is not a complete description

Fig. 1.19: The **mstatus** register.

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 1.20. 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. 1.20: 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 1.21. When an

Fig. 1.21: 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 1.22. When a trap is taken in machine mode, the most significant bit in **mcause** (bit INT) is

Fig. 1.22: The **mcause** register.

0, and the ten least-significant bits (EXCEPTION CODE 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 CODE field) contain the interrupt number, using the same encoding as the bit positions in the **mip** register. Table 1.3 lists exception codes and their description. For example, a Machine Timer Interrupt causes **mcause** to be set to `0x80000007`.

Table 1.3: **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
0	8	Environment call from U-mode
0	11	Environment call from M-mode
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 1.23. The **mtvec**

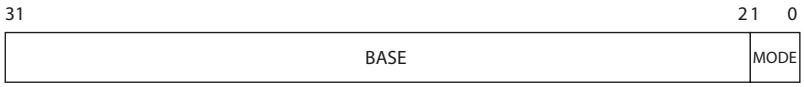

Fig. 1.23: The **mtvec** register.

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 1.4 describes how an address of the exception handler is computed in two different interrupt processing modes. **Direct mode** means all interrupts and exceptions trap to the same handler, and there is no vector table implemented. It is the handler's responsibility to execute code to figure out which interrupt occurred. The handler in direct mode should first read the bit 31 in **mcause** to determine if an interrupt or exception occurred. It should then execute appropriate code based on the EXCEPTION CODE field in **mcause**, which contains the respective interrupt or exception code. **Exceptions always use the direct mode**. Hence, all exceptions trap to the same handler. For example, suppose the BASE is set to 0x20011500. When an exception occurs, the PC is set to 0x20011500, and the first instruction of the exception handler should be at this address. **Vectored mode** allows for creating a vector table that hardware uses

Table 1.4: **mtvec** Modes and Address of Exception Handler Encoding.

MODE	Interrupt Processing Mode	Address of Exception Handler
0	Direct	`PC = BASE` *NOTE: Exceptions are not vectored. All exceptions trap to the same handler. The handler executes code to figure out which exception occurred.*
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*

for lower interrupt handling latency. **Only interrupts can use the vectored**

mode. When an interrupt occurs in vectored mode, the PC will get assigned by the hardware to the address of the vector table index corresponding to the interrupt ID. From the vector table index, a subsequent jump will occur from there to service the interrupt. The interrupt handler offset is calculated by `PC = BASE + 4 x mcause[EXCEPTION CODE]`. The vectored mode does not require the software overhead to determine which interrupt occurred. In this mode, when an interrupt occurs, the execution jumps directly to the vector table offset for the corresponding interrupt. For example, suppose the global and machine 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.

1.4.3 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 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 1.24). 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:

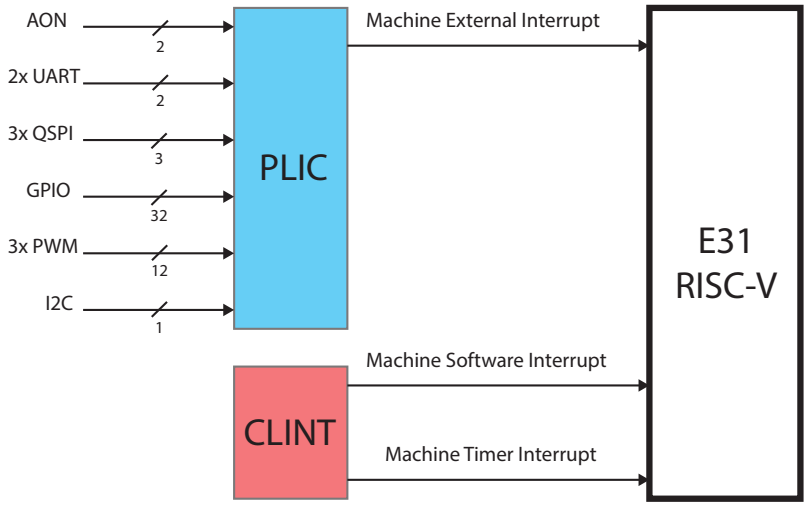

Fig. 1.24: FE310 Interrupt Architecture Block Diagram.

1. Machine Timer Interrupts: The CLINT contains a timer called the Machine Timer, which can generate timer interrupts for various purposes, including time-keeping, 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 1.5 shows the memory map for CLINT on SiFive FE310.

Table 1.5: Memory map for CLINT registers on SiFive FE310 SoC.

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

1.4.3.1 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.

1.4.3.2 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.

1.4.4 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.

1.4.4.1 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 1.4.

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.

1.4.4.2 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.

1.4.5 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 1.4). 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 1.22: A vector table for E31 RISC-V.

We can see from Listing 1.22 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 `_default_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 1.23 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 1.23: Assembly function for registering the vector table base addreess.

3.

4. **Define Exception/Interrupt Handler:** Write the exception/interrupt 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 1.4.4.2. For example, the following code (Listing 1.24) 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)
```

```
27 sw ra, 15*4(sp)
28
29 # Decode interrupt cause
30 csrr t0, mcause      # read exception cause
31 bgez t0, 1f          # exit if not an interrupt
32
33 # Increment timer compare by 1000 cycles
34 li t0, 0x0200BFF8    # load the mtime address
35 lw t1, 0(t0)         # load mtime (LO)
36 lw t2, 4(t0)         # load mtime (HI)
37 li t3, 1000          # load 1000 cycles
38 add t3, t1, t3       # increment lower bits by 1000
39 sltu t1, t3, t1      # generate carry-out
40 add t2, t2, t1       # increment upper bits with carry
41
42 li t0, 0x02004000    # load the mtimecmp address
43 sw t3, 0(t0)         # update mtimecmp (LO)
44 sw t2, 4(t0)         # update mtimecmp (HI)
45
46 1:
47 # Epilogue: restore ABI caller regs
48 lw t0, 0*4(sp)
49 lw t1, 1*4(sp)
50 lw t2, 2*4(sp)
51 lw t3, 3*4(sp)
52 lw t4, 4*4(sp)
53 lw t5, 5*4(sp)
54 lw t6, 6*4(sp)
55 lw a0, 7*4(sp)
56 lw a1, 8*4(sp)
57 lw a2, 9*4(sp)
58 lw a3, 10*4(sp)
59 lw a4, 11*4(sp)
60 lw a5, 12*4(sp)
61 lw a6, 13*4(sp)
62 lw a7, 14*4(sp)
63 lw ra, 15*4(sp)
64 addi sp, sp, 16*4   # de-allocate the routine stack frame
65 mret
```

Listing 1.24: Assembly code for the machine timer interrupt.

The code in Listing 1.24 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 1.25: 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:

```
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 1.26: 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 1.27) presents the machine timer interrupt handler.

```
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 1.27: 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 1.28) 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 1.28: 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 **mie** register. Listing 1.29 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 1.29: Assembly function for enabling the machine timer interrup in the **mie** register.

1.4.6 Case study: A simple task scheduler on RISC-V based FE310

Having delved into the intricacies of context switching on the ARM Cortex-M7 architecture, we now embark on a compelling journey to explore the analogous process on the RISC-V architecture. Our exploration of context switching on ARM Cortex-M7 processors has given us insights into the nuanced dance of saving and restoring task states, managing interrupts, and orchestrating seamless transitions between tasks. Now, with the backdrop of ARM's methodologies, we set our sights on RISC-V—a modular and versatile architecture that captivates developers and researchers alike with its openness and adaptability. In our previous foray into ARM Cortex-M7, we uncovered the distinctive features of the ARM Cortex-M7 architecture. We navigated the intricacies of saving and restoring register states, mitigating interruptions, and steering the efficient flow of tasks. Armed with this knowledge, we are ready to apply these principles to the RISC-V. By understanding the parallels and distinctions between ARM Cortex-M7 and RISC-V, we are poised to master the art of crafting efficient and tailored context-switching routines for diverse computing environments.

Recall that context switching, a pivotal aspect of modern computing, is a process that allows a system to seamlessly transition between multiple tasks, ensuring responsiveness and the efficient use of computational resources. Exploring context switching in the RISC-V architecture unveils a journey through the intricacies of multitasking and efficient resource utilization on RISC-V, characterized by its simplicity, modularity, and open design philosophy, which provides a flexible canvas for implementing context-switching mechanisms.

Contrasting RISC-V's approach to context switching with the ARM Cortex-M7, notable differences emerge. We have learned that the ARM Cortex-M7 employs a specific set of registers (e.g. banked stack pointer) and a dedicated interrupt-handling mechanism to facilitate context switching. Its unique stack frame format and the presence of two different stack pointers, one for tasks (PSP) and one for interrupt handlers (MSP), contribute to efficient task switching. Another key distinction lies in the register sets used during context switching. While both architectures involve saving and restoring register values, the specific registers and their organization differ. Understanding the intricacies of register usage in each architecture is crucial for crafting efficient context-switching routines.

This case study explores the intricacies of context switching in the RISC-V architecture, delving into its underlying mechanisms and the challenges involved in orchestrating efficient task transitions.

1.4.6.1 Background

A simple round-robin task scheduler (Figure 1.25) on RISC-V-based FE310 processors effectively manages multiple tasks or threads in a cooperative multitasking environment. In this scheduler, each task is given a fixed time slice (quantum) during which it can execute. When its time slice expires, the scheduler switches to the

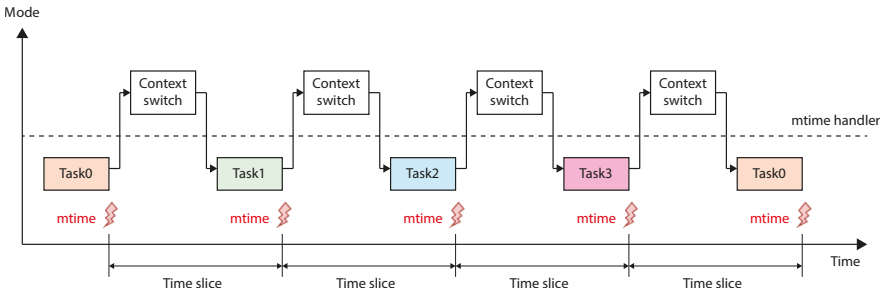

Fig. 1.25: A simple task scheduler on RISC-V based FE310.

next task in the queue. **The task scheduler relies on the interrupts and stacks to achieve context switching**. The machine timer (**mtime**) interrupts will be used for context switching.

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, allowing stack separation between the kernel and tasks, which in turn simplifies the context switch procedure. RISC-V-based FE-310 has only one stack pointer, which slightly complicates the context switching and forces us to carefully manage the stack within the interrupt handler. Besides, both tasks and kernel will run in machine mode.

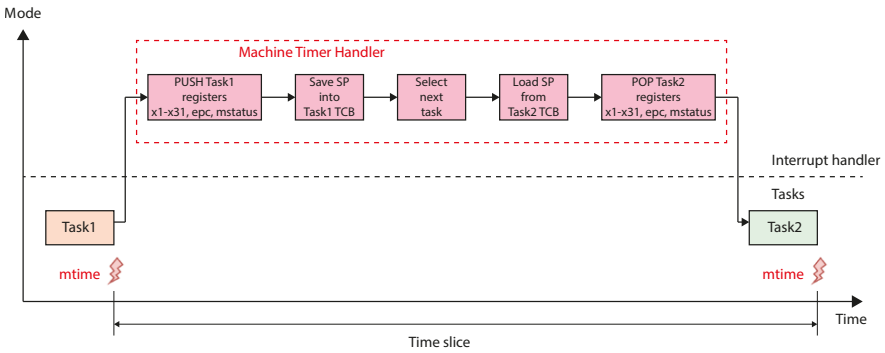

Fig. 1.26: A simple task scheduler.

Figure 1.26 shows the scheduler operations during a context switch in more detail. When a Machine Timer interrupt occurs, the execution switches to the machine timer interrupt handler. Once in the machine timer interrupt handler, the scheduler pushes the interrupted Task1 registers **x1** (return address), **x5-x31**, **epc**, and **mstatus** onto the task's stack and saves its SP in the task's TCB. Contrary to ARM-Cortex

M7, RISC-V does not automatically save critical registers (i.e. it does not implement hardware stacking). Upon interrupt entry, RISC-V only saves the return address into **epc** and the status of the interrupted procedure into **mstatus**. Hence, the interrupt handler is responsible for saving the complete context of the interrupted task: registers **x1**, **x5-x31**, return address contained in **epc** and the processor status before the interrupt occurred (contained in **mstatus**).

Then, the scheduler selects the next task (Task2) in a round-robin fashion. Before returning from the machine timer interrupt handler, the scheduler is responsible for loading the Task2 SP and restoring the Task2 context (**x1**, **x5-x31**, **epc**, and **mstatus**) from the Task2 stack. Finally, upon interrupt exit, **mepc** is copied to **pc** and the execution returns to the new task, Task2.

As we have already learned, 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. The stack region used to implement the tasks' stacks, and the task control block are the same as in the Subsection 1.3.7. The following subsections provide a step-by-step description of implementing a simple round-robin scheduler on a RISC-V-based FE310 processor.

1.4.6.2 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.

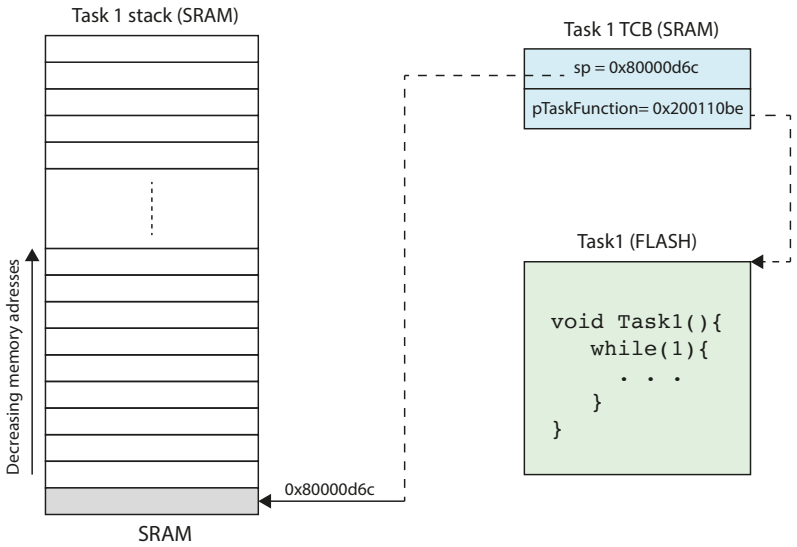

Fig. 1.27: Memory layout and content after calling the `TaskCreate()` function.

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

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

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

Listing 1.30: The function `TaskCreate()` that creates a new task.

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

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

Figure 1.27 illustrates the memory layout and the contents of the task's stack and TCB after creating Task1 using the `TaskCreate()` function.

1.4.6.3 Task initialisation

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

```
void TaskInit(TCB_Type* pTCB){
  Context_TypeDef* pStackFrame;

  // Make the room for the stack frame and
  // set pointer to the top of stack frame:
  pStackFrame = (Context_TypeDef*)((void*)pTCB->sp -
                    sizeof(Context_TypeDef));

  // populate Stack Frame
  pStackFrame->mepc     = (unsigned int) (pTCB->pTaskFunction);
  pStackFrame->x1       = 0xFFFFFFFF; // (ra: task never exits)
  pStackFrame->mstatus  = (0x03 << 11) | (0x01 << 7); // value 0x1880

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

Listing 1.31: 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. Initialise the pointer to the stack frame. The stack frame will hold the task's context. We need to prepare the stack frame for each new task so that when the task switch occurs, the frame will be ready for de-stacking and, hence, entering a new task. To make this task easier, we will abstract the stack frame with the following C structure:

```
/*
 * The RISC-V context is saved in the following stack frame,
 * where the global(tp) and thread (tp) pointers
 * are currently assumed to be constant so are not saved:
 */
typedef struct{
  unsigned int mepc;  // (sp+0)
  unsigned int x1;    // (sp+1)
  unsigned int t5;    // (sp+2)
  unsigned int x6;    // (sp+3)
  unsigned int x7;    // (sp+4)
  unsigned int x8;    // (sp+5)
  unsigned int x9;    // (sp+6)
  unsigned int x10;   // (sp+7)
  unsigned int x11;   // (sp+8)
  unsigned int x12;   // (sp+9)
  unsigned int x13;   // (sp+10)
  unsigned int x14;   // (sp+11)
  unsigned int x15;   // (sp+12)
  unsigned int x16;   // (sp+13)
  unsigned int x17;   // (sp+14)
  unsigned int x18;   // (sp+15)
  unsigned int x19;   // (sp+16)
  unsigned int x20;   // (sp+17)
  unsigned int x21;   // (sp+18)
  unsigned int x22;   // (sp+19)
  unsigned int x23;   // (sp+20)
  unsigned int x24;   // (sp+21)
  unsigned int x25;   // (sp+22)
  unsigned int x26;   // (sp+23)
  unsigned int x27;   // (sp+24)
  unsigned int x28;   // (sp+25)
  unsigned int x29;   // (sp+26)
  unsigned int x30;   // (sp+27)
  unsigned int x31;   // (sp+28)
  unsigned int mstatus; // (sp+29)
  unsigned int unused1; // (sp+30)
  unsigned int unused2; // (sp+31)
} Context_TypeDef;
```

Listing 1.32: A C structure used to abstract the task's stack frame.

2. Now, as the pointer to the stack frame, `pStackFrame`, is set, we can populate the frames with initial values. The stack frame is populated as follows:
   - **mstatus** = 0x00001880. We set the task's previous privilege level (field MPP in **mstatus**) to '11' (machine mode), and the task's previous interrupt flag MPIE to '1'.
   - **mepc** = the address of the task,
   - **x1 (ra)** = 0xFFFFFFFF - in our case, tasks never finish, so the return address is set to 0xFFFFFFFF.
3. Finally, it saves the address of the top of the stack frame into the task's SP entry 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 1.28 illustrates the memory layout and the contents of the task's stack and TCB after creating Task1 using the `TaskInit()` function.

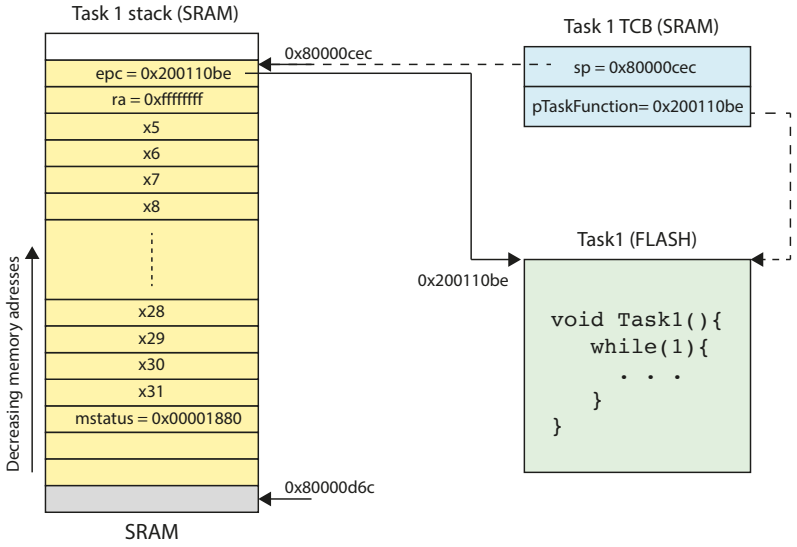

Fig. 1.28: Memory layout and content after calling the `TaskInit()` function.

1.4.6.4 Scheduler initialisation

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

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

5    // 1. create all tasks:
     for(int i=0; i<NTASKS; i++){
7      pTaskStackBase = pStackRegion +
                        (i+1)*TASK_STACK_SIZE;
9      TaskCreate(&pTCB[i], pTaskStackBase,
                  TaskFunctions[i]);
11   }

13   // 2. initialize all tasks:
     for(int i=0; i<NTASKS; i++){
15     TaskInit(&pTCB[i]);
     }
17 }
```

Listing 1.33: The function `InitScheduler()` creates and initializes all tasks.

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

1. Creates all tasks.
2. Initializes all tasks

After these steps, everything is set up for the first context switch. Figure 1.29 illus-

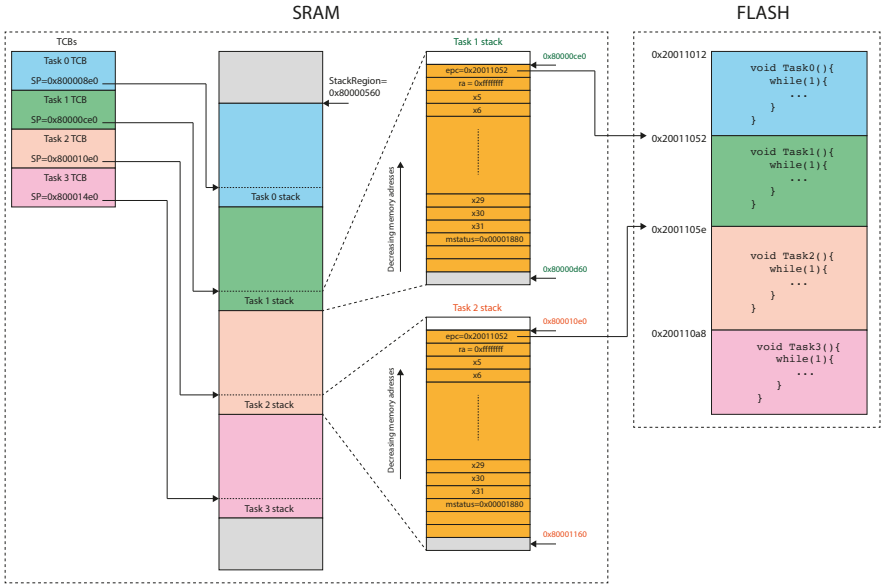

Fig. 1.29: Memory layout and content after initializing four tasks during the scheduler initialization.

trates the memory layout and the task's stack after initializing the scheduler using the `InitScheduler()` function.

1.4.6.5 Machine timer interrupt handler

Finally, we can implement the machine timer interrupt handler that will perform the task switch. Contrary to ARM Cortex-M7, RISC-V does not have two separate stack pointers for handlers and user programs. Hence, interrupt handlers on RISC-V-based FE310 use the same stack pointer as interrupted tasks. If we implemented the machine timer interrupt handler in C, the stack pointer would become corrupted by the interrupt handler itself because the C compiler would generate the prologue code according to the calling convention. On the other hand, assembly language enables precise management of the stack pointer, allowing for the preservation of the current task's context and the restoration of the new task's context. Therefore, an interrupt handler used for context switching on a RISC-V-based processor should be written in assembly language. Here is the machine interrupt handler used for context switching:

```
1 /*---------------------------------------
2     Machine Timer Interrupt Handler
3 ---------------------------------------*/
```

```
.balign 4, 0
.global _mtim_interrupt_handler
_mtim_interrupt_handler:
    # Save context:
    __macro_SAVE_CONTEXT

    # Increment time slice (tick)
    __macro_INCREMENT_TICK

    # switch context
    __macro_SWITCH_CONTEXT

2:
    # Restore context
    __macro_RESTORE_CONTEXT

    mret
```

Listing 1.34: The machine timer interrupt handler used to perform task switch.

The machine timer interrupt handler performs the following steps:

1. Saves the context of the interrupted task on the task's stack using the `__macro_SAVE_CONTEXT` macro, defined as:

```
.macro  __macro_SAVE_CONTEXT
    addi sp, sp, -CONTEXT_SIZE
    sw x1,      1*WORD_SIZE(sp)
    sw x5,      2*WORD_SIZE(sp)
    sw x6,      3*WORD_SIZE(sp)
    sw x7,      4*WORD_SIZE(sp)
    sw x8,      5*WORD_SIZE(sp)
    sw x9,      6*WORD_SIZE(sp)
    sw x10,     7*WORD_SIZE(sp)
    sw x11,     8*WORD_SIZE(sp)
    sw x12,     9*WORD_SIZE(sp)
    sw x13,     10*WORD_SIZE(sp)
    sw x14,     11*WORD_SIZE(sp)
    sw x15,     12*WORD_SIZE(sp)
    sw x16,     13*WORD_SIZE(sp)
    sw x17,     14*WORD_SIZE(sp)
    sw x18,     15*WORD_SIZE(sp)
    sw x19,     16*WORD_SIZE(sp)
    sw x20,     17*WORD_SIZE(sp)
    sw x21,     18*WORD_SIZE(sp)
    sw x22,     19*WORD_SIZE(sp)
    sw x23,     20*WORD_SIZE(sp)
    sw x24,     21*WORD_SIZE(sp)
    sw x25,     22*WORD_SIZE(sp)
    sw x26,     23*WORD_SIZE(sp)
    sw x27,     24*WORD_SIZE(sp)
    sw x28,     25*WORD_SIZE(sp)
    sw x29,     26*WORD_SIZE(sp)
    sw x30,     27*WORD_SIZE(sp)
    sw x31,     28*WORD_SIZE(sp)

    csrr t0, mepc
    sw t0,      0*WORD_SIZE(sp)
    csrr t0, mstatus
    sw t0,      29*WORD_SIZE(sp)
.endm
```

Listing 1.35: `__macro_SAVE_CONTEXT` macro.

2. Increments the time slice using the `__macro_INCREMENT_TICK` macro:

```
.macro __macro_INCREMENT_TICK
    # Increment timer compare by TIME_SLICE cycles
    la t0, CLINT_MTIME      # load the mtime address
    lw t1, 0(t0)            # load mtime (LO)
    lw t2, 4(t0)            # load mtime (HI)
    li t3, TIME_SLICE
    add t3, t1, t3          # increment lower bits
                            # by TIME_SLICE cycles
    sltu t1, t3, t1         # generate carry-out
    add t2, t2, t1          # add carry to upper bits
    la t0, CLINT_MTIME_CMP
    sw t3, 0(t0)            # update mtimecmp (LO)
    sw t2, 4(t0)            # update mtimecmp (HI)
.endm
```

Listing 1.36: `__macro_INCREMENT_TICK` macro.

3. Switch context (swap stack pointers) using the `__macro_SWITCH_CONTEXT`:

```
.macro __macro_SWITCH_CONTEXT
    la t0, current_task
    lw t1, 0(t0)            # t1 holds current_task
    sll t4, t1, 3           # t4 = t1 * 8
    la t5, TCB              # t5 <- &TCB[0]
    add t5, t5, t4          # t5 <- &TCB[current_task]
    sw sp, 0(t5)            # save sp of the current
                            #   task

    # select a new task in round-robin:
    addi t1, t1, 1
    li t2, NTASKS
    bne t1, t2, 1f
    li t1, 0
1:  sw t1, 0(t0)

    sll t4, t1, 3           # t4 = t1 * 8
    la t5, TCB              # t5 <- &TCB[0]
    add t5, t5, t4          # t5 <- &TCB[current_task]
    lw sp, 0(t5)            # load sp of the current ↩
     task
.endm
```

Listing 1.37: `__macro_SWITCH_CONTEXT` macro.

4. Restores the context of the new task from its stack using the `__macro_RESTORE_CONTEXT` macro:

```
.macro  __macro_RESTORE_CONTEXT
    lw t0,      0*WORD_SIZE(sp)
    csrw mepc, t0
    lw t0,      29*WORD_SIZE(sp)
    csrw mstatus, t0

    lw x1,      1*WORD_SIZE(sp)
    lw x5,      2*WORD_SIZE(sp)
    lw x6,      3*WORD_SIZE(sp)
    lw x7,      4*WORD_SIZE(sp)
    lw x8,      5*WORD_SIZE(sp)
    lw x9,      6*WORD_SIZE(sp)
```

```
13      lw  x10,      7*WORD_SIZE(sp)
14      lw  x11,      8*WORD_SIZE(sp)
15      lw  x12,      9*WORD_SIZE(sp)
16      lw  x13,      10*WORD_SIZE(sp)
17      lw  x14,      11*WORD_SIZE(sp)
18      lw  x15,      12*WORD_SIZE(sp)
19      lw  x16,      13*WORD_SIZE(sp)
20      lw  x17,      14*WORD_SIZE(sp)
21      lw  x18,      15*WORD_SIZE(sp)
22      lw  x19,      16*WORD_SIZE(sp)
23      lw  x20,      17*WORD_SIZE(sp)
24      lw  x21,      18*WORD_SIZE(sp)
25      lw  x22,      19*WORD_SIZE(sp)
26      lw  x23,      20*WORD_SIZE(sp)
27      lw  x24,      21*WORD_SIZE(sp)
28      lw  x25,      22*WORD_SIZE(sp)
29      lw  x26,      23*WORD_SIZE(sp)
30      lw  x27,      24*WORD_SIZE(sp)
31      lw  x28,      25*WORD_SIZE(sp)
32      lw  x29,      26*WORD_SIZE(sp)
33      lw  x30,      27*WORD_SIZE(sp)
34      lw  x31,      28*WORD_SIZE(sp)
35      addi  sp, sp, CONTEXT_SIZE
36  .endm
```

Listing 1.38: `__macro_RESTORE_CONTEXT` macro.

1.4.6.6 Using the environment call (ecall) exception to start the scheduler

We have already learned that instead of directly calling the first task (Task0) from the main function, the scheduler should rely on the exception return to start the first task. For this purpose, we can use the environment call exception. In the RISC-V architecture, environment calls, often abbreviated as "ecalls," are a mechanism by which a user-level program can request services or system functions from the kernel. The `ecall` instruction initiates an environment call. This instruction triggers an exception, which is trapped at the exception handler on the address stored in the BASE field of the **mtvec** register. Figure 1.30 shows the process of starting and

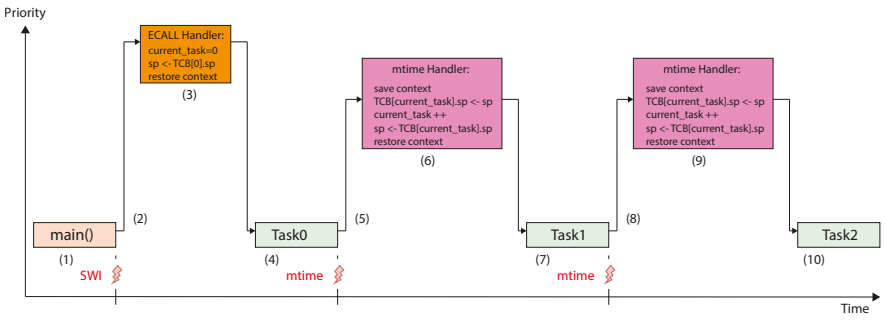

Fig. 1.30: Starting the scheduler with the SVC exception.

running the scheduler using the environment call exception.

The environment call handler simply sets the SP to point to the top of Task0's stack, initializes the first tick, enables the machine timer interrupt and restores the context of the first task:

```
.balign 4, 0
.global _exception_handler
_exception_handler:
    # Decode exception cause:
    csrr t0, mcause      # read exception cause
    bltz t0, 2f          # exit if not an exception

    # Check if ECALL:
1:  li t1, 0xB           # ecall from M-mode
    bne t1, t0, 2f

    /*
      ECALL:
      1. load the SP of the first task (SP <- TCB[0].sp)
      2. increment tick to set the first timer interrupt
      3. enable MTIME interrupt
      4. restore context of the first task
      5. Upon return, the first task is executed and
         the scheduler is running
    */
    /* 1. Load SP of the first task: */
    la t1, TCB       // load the address of TCB[0]
    lw sp, 0(t1)     // load sp from TCB[0].sp

    # 2. Increment time slice (tick)
    __macro_INCREMENT_TICK

    /* 3. Enable MTIME interrupt */
    li t0, 0x00000080
    csrs mie, t0

    # 4. Restore the context of Task0
    __macro_RESTORE_CONTEXT
2:
    mret
```

Listing 1.39: The environemt call exception handler.

Finally, here is the main function which initializes and starts the scheduler:

```
int main() {

  // Set the task functions:
  TaskFunctions[0] = Task0;
  TaskFunctions[1] = Task1;
  TaskFunctions[2] = Task2;
  TaskFunctions[3] = Task3;

  // Init scheduler:
  InitScheduler(stackRegion, TCB, TaskFunctions);
  current_task = 0;

  // Set up vectored interrupts and enable CPU's interrupts
  _register_handler(_vector_table, INT_MODE_VECTORED);
  _enable_global_interrupts();

  //Environment call - start the scheduler:
```

```
    __asm__ volatile("ecall");

    // We should never return here...
    while(1){}
    return 0;
}
```

Listing 1.40: Initializing and starting the the scheduler.

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 1.41 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 1.41: ARM vector table and interrupt handlers.

Listing 1.41 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 1.42 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 1.42: 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 1.42. 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 1.43: 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 1.44: 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 1.45: 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 1.46: 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

an exception or interrupt, the processor first saves the current state of the interrupted program or task (the status FLAGS register and the program counter) on the stack. Each entry in the IDT (or IVT) holds the start address of the interrupt handler. The processor thus reads the start address of the handler from the IDT (or IVT) into the program counter and starts the execution of the handler. To return from an exception- or interrupt-handler handler, the handler uses the IRET instruction. The IRET instruction is similar to the RET instruction used to return from normal procedures except that it restores the saved status register FLAGS.

1.7 Interrupt controllers

We have seen that the interrupt line from a peripheral device should be connected to the CPU's interrupt input signal. In such a way, a peripheral device can interrupt the CPU and require its attention. The CPU will sense this interrupt input signal at every instruction fetch and know that the peripheral device needs attention. But what should we do if there is more than one peripheral device that would like to interrupt the CPU? What if there are tens of external peripheral devices, which is often the case in real computer systems? Should we add an interrupt input pin to the CPU for every external peripheral device? A large number of interrupt input pins on the CPU for every external device would make the CPU interfacing very complicated and increase the error probability.

One possible solution to solve this problem would be to have one level-sensitive interrupt input pin (IRQ) on the CPU shared by all external peripheral devices. This solution is illustrated in Figure 1.31. Whenever an interrupt is asserted, the CPU branches to the interrupt handler associated with the IRQ pin. This interrupt handler would poll each and every I/O peripheral device to determine which device asserted the interrupt line. So, the CPU will handle the interrupt request on the IRQ pin as vectored interrupts, but will also, within the interrupt handler, use the polled interrupts method to check the interrupt's cause. Every modern I/O peripheral device has an addressable (memory-mapped) status register. There is usually one bit in this status register, referred to as an interrupt-pending bit, which is set internally by the interrupting device when the device asserts the interrupt line. This interrupt-pending bit in the status register can be read by the CPU to determine the interrupting device. If the interrupt-pending bit is set, the processor branches to a specific device-service routine. Within this device-specific routine, the interrupt handling bit is cleared and the interrupt request is serviced. Listing 1.47 shows pseudocode for the IRQ interrupt handler that uses polling to determine which device has requested the interrupt. The downside to this technique is that it is time-consuming.

```
1 /*
  * Polling Handler
3 */
  __attribute__((interrupt)) void polling_IRQ_handler () {
5
```

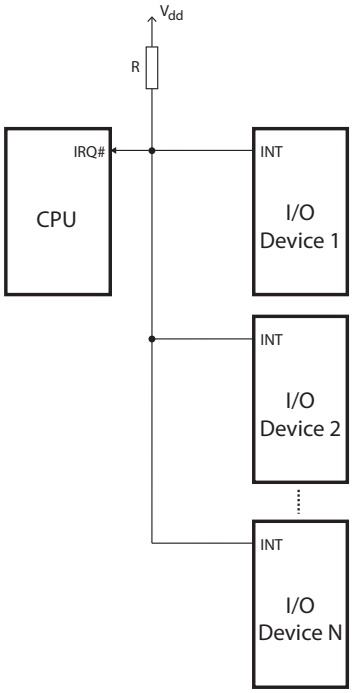

Fig. 1.31: Example system with several I/O devices sharing one interrupt input signal.

```
    /* Check the interrupt pending bit in the I/O device 1 */
    if (IO1_status_reg & (1<<INT_PEND_BIT)) {
    /* I/O Device 1 Code */
    IO1_status_reg &= ~(1<<INT_PEND_BIT); // clear int pending bit

    /* Do something */

    }

    /* Check the interrupt pending bit in the I/O device 2 */
    if (IO2_status_reg & (1<<INT_PEND_BIT)) {
    /* I/O Device 2 Code */
    IO2_status_reg &= ~(1<<INT_PEND_BIT); // clear int pending bit

    /* Do something */

    }

    ...

    /* Check the interrupt pending bit in the I/O device N */
    if (ION_status_reg & (1<<INT_PEND_BIT)) {
    /* I/O Device N Code */
    ION_status_reg &= ~(1<<INT_PEND_BIT); // clear int pending bit

    /* Do something */
```

```
33      }
}
```

Listing 1.47: IRQ interrupt handler with polling.

A better solution to this problem would be to use a special chip — an **interrupt controller**. The interrupt controller is a special device, which:

- combine all external interrupt requests onto one CPU IRQ line,
- prioritizes them (decides which interrupt request will be routed to the CPU when more than one I/O device has requested the interrupt),
- routes the selected interrupt request to the CPU's IRQ input signal, and
- most importantly, it provides the CPU with the information which device has requested the interrupt. Commonly, it provides the CPU with the interrupt vector; thus, the CPU does not have to poll I/O peripheral devices.

Figure 1.32 illustrates the structure of a system that uses an interrupt controller. All potential external interrupt sources are routed through the interrupt controller. In the case of one or more interrupt requests, the interrupt controller prioritizes the interrupt inputs, it transfers the interrupt request with the highest priority to the CPU, along with interrupt vector. This sequence is performed by hardware in the interrupt controller and not by software in the CPU, hence the interrupt controller provides a much faster response to an interrupt request.

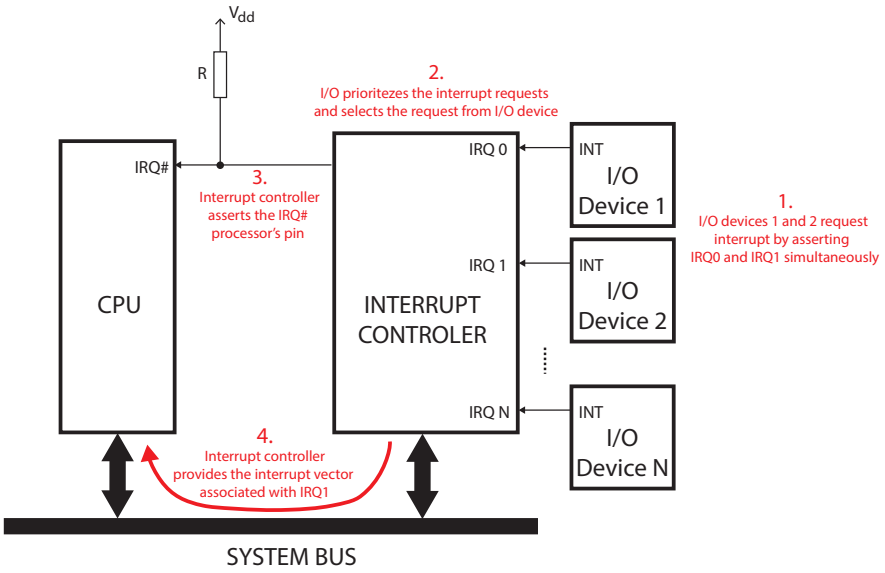

Fig. 1.32: A system with an interrupt controller.

In summary, interrupt controllers are essential in modern computer systems as they facilitate the handling of diverse interrupt requests generated by various hardware components. By allowing the CPU to respond to events as they occur (in contrast to continuous polling of hardware), interrupt controllers optimize CPU resources, ensuring that the processor only performs work when necessary. In systems where multiple interrupts can occur simultaneously, interrupt controllers manage the nesting of interrupts. They ensure that an interrupt can be interrupted by a higher-priority one while maintaining the correct order of execution. They also manage interrupt priorities, ensure timely response to critical events, and optimize system resources by allowing the CPU to handle events as they occur, all of which are fundamental for efficient and responsive system operation.

Although the operations in an interrupt controller are performed by hardware, interrupt controllers are programmable. It means that they typically have a common set of addressable (memory-mapped) registers, which enable the system programmer to set the priorities and interrupt vectors for each interrupt source before the interrupt controller is being used. In the following sections, we will cover a few real-world interrupt controllers used with ARM and Intel processors.

1.7.1 ARM Advanced Interrupt Controller

The Advanced Interrupt Controller (AIC) is an 8-level priority vectored interrupt controller, providing handling of up to thirty-two interrupt sources. It is used with ARM9 processors. Figure 1.33 illustrates the block diagram of an ARM9 based system with AIC. The AIC drives the FIQ# (fast interrupt request) and the IRQ# (standard interrupt request) inputs of an ARM9 processor. Inputs of the AIC are external interrupts coming from the peripheral I/O devices.

The Interrupt Source 0 (IS 0) is always connected to the FIQ processor's input. The interrupt sources 1 to 31 (IS 1 to IS 31) can be connected to the interrupt outputs of peripheral devices. An 8-level priority controller drives the IRQ line of the processor. Each interrupt source has a programmable priority level of 7 (the highest priority) to 0 (the lowest priority).

As soon as an interrupt request occurs on an interrupt source, the IRQ# line is asserted. If several interrupt sources have asserted the interrupt request, the priority controller determines the interrupt source with the highest priority, which will be serviced. If several interrupt sources of equal priority are pending, the interrupt with the lowest interrupt source number is serviced first. If an interrupt request happens during the interrupt service in progress, it is delayed until the software indicates to the AIC the end of the current service. Figure 1.33 illustrates the simplified internal structure of AIC. AIC employs an interrupt vectoring scheme. The interrupt handler

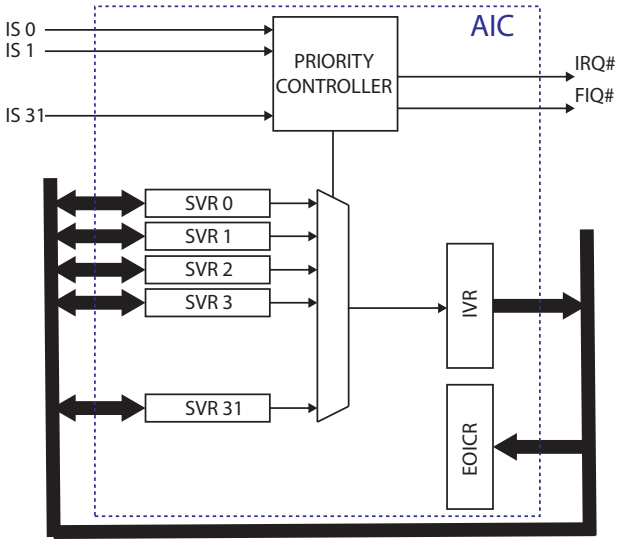

Fig. 1.33: Simplified internal structure of AIC.

addresses (interrupt vectors) corresponding to each interrupt source can be stored

in the AIC's registers SVR1 to SVR31 (Source Vector Register 1 to 31). When one or more interrupt requests occur, the content of the SVR corresponding to the interrupt source with the highest priority is automatically transferred to the Interrupt Vector Register (IVR). To obtain the start address of the interrupt handler, the CPU must read the IVR register. In the ARM9 based systems, the IVR register is always mapped at the absolute address 0xFFFFF100. Remember that the interrupt vector for IRQ interrupt is 0x00000018. Hence, the IVR accessible from the ARM interrupt vector at address 0x00000018 through the following instruction:

```
ldr pc,[pc,#-0XF20]
```

When the processor executes this instruction, it loads the value in IVR into its program counter, thus branching the execution on the correct interrupt handler. Besides, reading the IVR also de-asserts the IRQ# line on the processor. But from where the value -0xF20 comes in the above instruction? Recall that the instructions are executed in the EX stage. By the time the above instruction is issued into the EX stage, the PC has already been increased by 8 and is equal to 0x00000020. This is because the CPU has fetched two more instructions. Hence, we have to subtract 0x2F0 from 0x00000020 to obtain 0xFFFF F100.

Before returning, the interrupt handler must indicate to the AIC the end of the current service by dummy writing to the EOICR register (End Of Interrupt Command Register). This will re-enable the further interrupts in AIC. The return from the interrupt handler is, as we have already learned, performed by the `subs pc,lr,#4` instruction. This has the effect of returning from the interrupt to whatever was being executed before, and of restoring the CPSR from the SPSR.

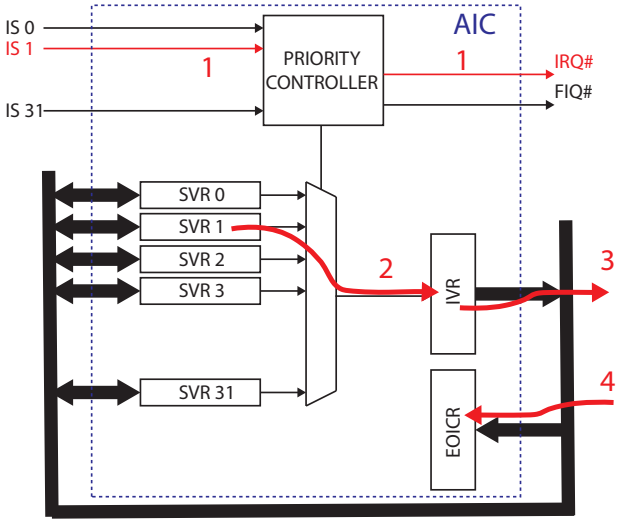

Fig. 1.34: Simplified internal operation of AIC.

An example of the procedure of obtaining the interrupt vector is in Figure 1.34. Let us assume that a peripheral device asserts the interrupt request at the IS 1 line of AIC (step 1). Assuming that no other IS line has been asserted and that the CPU services no interrupt, the priority controller in AIC immediately asserts the CPU's IRQ# signal (step 1). Then, the priority controller selects the SVR1 register, and its content is transferred into the IVR register (step 2). The CPU detects that IRQ# has been asserted, stops the instruction execution, and saves the context of the interrupted program. It then fetches the instruction `ldr pc,[pc,#-0XF20]` from the IRQ interrupt vector (0x00000018). This instruction moves the content of the IVR register into the program counter (step 3) and CPU branches on the IRQ handler. Before returning from the IRQ handler, the CPU dummy writes into the EOICR (step 4).

As we have seen, when AIC is used to route external interrupt requests from peripheral devices to the CPU, the instruction at the interrupt vector 0x00000018 is not a branch instruction (B) to the interrupt handler, but the instruction that loads the IVR into PC (which also acts as a branch). The same holds for the FIQ vector. Hence, we should change the interrupt vector table from Listing 1.41, accordingly. Also, the interrupt handlers for interrupt sources IS1 to IS31 should dummy write to EOICR before returning. Listing 1.48 shows the updated interrupt vector table and pseudocode for an ISx interrupt handler.

```
        .org 0x00000000
Vector_Table:
        b Reset_Handler
        b Undefined_Handler
        b SWI_Handler
        b Prefetch_Handler
        b Abort_Handler
        nop
        lr pc, [pc, #-0xF20]    // load IVR into PC
        lr pc, [pc, #-0xF20]    // load IVR into PC


ISx_Handler:
        <handler instructions>
        ...
        <write to EOICR>
        subs pc,lr,#4
```

Listing 1.48: ARM vector table and ISx handler when AIC is present in the system.

1.7.2 RISC-V Platform-Level Interrupt Controller in FE310

In Subsection 1.4.3, we learned that SiFive FE310 SoC contains two interrupt controllers: The Core Local Interruptor (CLINT) and the Platform Level Interrupt Controller. The Core Local Interruptor (CLINT) is a mandatory component in RISC-V-based systems, which provides two local interrupts (software and timer) to the RISC-V core. The PLIC is another interrupt controller in the SiFive FE310s. It is responsible for managing global interrupts from various IO devices in the system and distributing them to the RISC-V through the Machine External Interrupt line.

The FE310 SoC has multiple peripherals (timers, GPIO pins, UARTs, etc.) that can generate interrupts. These peripheral devices generate (drive) 52 interrupt sources. The PLIC aggregates these interrupt sources and generates the interrupt request over the Machine External Interrupt line. Table 1.8 lists peripheral devices and associated interrupt sources. For example, each GPIO pin can generate one interrupt

Table 1.8: Peripheral devices and their associated interrupt sources in FE310 PLIC.

Device	Interrupt source IDs
WDT	1
RTC	2
UART0	3
UART1	4
QSPI0	5
SPI1	6
SPI2	7
GPIO	8-39
PWM0	40-43
PWM1	44-47
PWM2	48-51
I2C	52

source; hence, the GPIO interface generates 32 interrupt sources.

The PLIC supports multiple priority levels for interrupts, allowing us to prioritize critical events over less critical ones. Priority levels are configurable. If two or more interrupt sources generate interrupt requests, the PLIC will select the source with the highest priority level. Each PLIC interrupt source can be assigned a priority by writing to its 32-bit memory-mapped priority register **priority**. The memory addresses of 52 **priority** registers are `0x0C000000 + 4 x SourceID`. For example, the address of the UART0's **priority** register is `0x0C00000C`. The FE310-G003 supports seven (7) levels of priority. A priority value of 0 means "never interrupt" and disables the interrupt for the source. Priority 1 is the lowest active priority, and priority 7 is the highest. Besides, global interrupts with the lowest source ID have the highest priority. In such a way, if two or more global interrupts with the same priority level are triggered, the PLIC will service first the one with the lowest source ID. The **priority** register is depicted in Figure 1.35. The three least significant bits in the **priority** encode the priority level.

Fig. 1.35: The **priority** register.

Besides priority levels, PLIC enables Per-Source Interrupt Control. Each global interrupt source connected to the PLIC can be individually enabled or disabled by setting the corresponding bit in two registers: **enable1** and **enable2**. This feature allows fine-grained control over which sources can generate interrupts. The **enable1** and **enable2** are memory-mapped and can be accessed as a contiguous array of two memory words at addresses `0x0C002000` (**enable1**) and `0x0C002004` (**enable2**). The enable bit for interrupt source ID is stored in the bit (ID mod 32) of the word (ID/32). For example, the enable bit of the interrupt source 3 (UART0) is stored in the bit (3 mod 32)=3 of the word (3/32)=0, which is accessible as the register **enable1**. Similarly, the enable bit of the interrupt source 39 (GPIO pin 31) is stored in the bit (39 mod 32)=7 of the word (39/32)=1, which is accessible as the register **enable2**. Bit 0 of **enable1** represents the non-existent interrupt source ID 0 and is hardwired to 0.

When one or more interrupt sources trigger the interrupt request to PLIC, PLIC will select the interrupt source with the highest priority and trigger the interrupt on the Machine External Interrupt line of the RISC-V CPU. At the same time, PLIC will write the ID of the highest priority interrupt source into its 32-bit **claim** register, memory-mapped at `0x0C200004`. The RISC-V will execute the Machine Internal Interrupt handler. This handler should then read the **claim** register. This read will return the ID of the highest-priority pending interrupt or zero if there is no pending interrupt. In such a way, the CPU will recognize which interrupt source has triggered the interrupt request. This step informs the PLIC that we're handling the interrupt and prevents it from reasserting the same interrupt while we're servicing it. After appropriately servicing the interrupt source, the Machine Internal Interrupt handler should write the interrupt ID it received from the **claim** register back to the **claim** register.

1.7.2.1 Implementing PLIC Vector Table and Handlers

Here, we will try to provide a complete code example for using the Platform-Level Interrupt Controller (PLIC) in the SiFive FE310 microcontroller. The code snippets in this subsection will hopefully demonstrate to you how to set up the vector table for PLIC interrupt sources, initialize the PLIC, handle a specific interrupt source, and acknowledge (complete) the interrupt.

Implementing a vector table, interrupt handlers and basic routines for the Platform-Level Interrupt Controller (PLIC) in the SiFive FE310 microcontroller in assembly and C language involves defining the vector table, writing assembly code

for each interrupt handler, and writing other routines for PLIC in C. Below, we provide a step-by-step guide to implement this:

1. **Define the Vector Table**. In assembly, we define the interrupt vector table for PLIC as a table (an array) of jump instructions to interrupt handlers. Each jump instruction in the vector table corresponds to a specific interrupt source. We can place the vector table at an arbitrary memory location, providing it is correctly aligned:

```
# ---------------------------------------------------
#
#  P L I C   V E C T O R   T A B L E
#
# ---------------------------------------------------
.balign 4
.global _plic_ext_vector_table
_plic_ext_vector_table:
    j _panic_handler                # PLIC src 0
    j _aon_wdt_handler              # PLIC src 1
    j _aon_rtc_handler              # PLIC src 2
    j _uart0_handler                # PLIC src 3
    j _uart1_handler                # PLIC src 4
    j _qspi0_handler                # PLIC src 5
    j _spi1_handler                 # PLIC src 6
    j _spi2_handler                 # PLIC src 7
    j _gpio0_handler                # PLIC src 8
    j _gpio1_handler                # PLIC src 9
    j _gpio2_handler                # PLIC src 10
    j _gpio3_handler                # PLIC src 11
    j _gpio4_handler                # PLIC src 12
    j _gpio5_handler                # PLIC src 13
    j _gpio6_handler                # PLIC src 14
    j _gpio7_handler                # PLIC src 15
    j _gpio8_handler                # PLIC src 16
    j _gpio9_handler                # PLIC src 17
    j _gpio10_handler               # PLIC src 18
    j _gpio11_handler               # PLIC src 19
    j _gpio12_handler               # PLIC src 20
    j _gpio13_handler               # PLIC src 21
    j _gpio14_handler               # PLIC src 22
    j _gpio15_handler               # PLIC src 23
    j _gpio16_handler               # PLIC src 24
    j _gpio17_handler               # PLIC src 25
    j _gpio18_handler               # PLIC src 26
    j _gpio19_handler               # PLIC src 27
    j _gpio20_handler               # PLIC src 28
    j _gpio21_handler               # PLIC src 29
    j _gpio22_handler               # PLIC src 30
    j _gpio23_handler               # PLIC src 31
    j _gpio24_handler               # PLIC src 32
    j _gpio25_handler               # PLIC src 33
    j _gpio26_handler               # PLIC src 34
    j _gpio27_handler               # PLIC src 35
    j _gpio28_handler               # PLIC src 36
    j _gpio29_handler               # PLIC src 37
    j _gpio30_handler               # PLIC src 38
    j _gpio31_handler               # PLIC src 39
    j _pwm0_handler                 # PLIC src 40
    j _pwm0_handler                 # PLIC src 41
    j _pwm0_handler                 # PLIC src 42
    j _pwm0_handler                 # PLIC src 43
    j _pwm1_handler                 # PLIC src 44
    j _pwm1_handler                 # PLIC src 45
```

```
    j _pwm1_handler                 # PLIC src 46
    j _pwm1_handler                 # PLIC src 47
    j _pwm2_handler                 # PLIC src 48
    j _pwm2_handler                 # PLIC src 49
    j _pwm2_handler                 # PLIC src 50
    j _pwm2_handler                 # PLIC src 51
    j _i2c_handler                  # PLIC src 52
```

Listing 1.49: The PLIC interrupt vector table.

2. **Define Interrupt Handler Routines**. Write the assembly code for each interrupt handler. These routines should handle the specific interrupt and include any necessary operations. Here, we provide only the the basic code for GPIO13 interrupt handler:

```
.balign 4
.weak _gpio13_handler
_gpio13_handler:
    # Your code goes here:
    ...
    ret
```

Listing 1.50: Assembly code for the GPIO13 (PLIC source 21) interrupt handler.

3. textbfWrite the Machine External Interrupt handler. This handler is invoked when the external interrupt is asserted:

```
/*---------------------------------------
    Machine External Interrupt Handler
---------------------------------------*/
.balign 4
.global _mext_interrupt_handler
.type _mext_interrupt_handler, @function
_mext_interrupt_handler:
    # Prologue : save 16 ABI caller registers
    ...

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

    # Claim the interrupt - read CLAIM
    #   A non-zero read contains the ID of
    #   the highest pending interrupt.
    la t0, PLIC_CLAIM   # load the address of CLAIM reg
    lw t1, 0(t0)        # read CLAIM
    slli t2, t1, 2      # id*4 to obtain the offset

    # load the address of the PLIC
    #   external interrupt vector table
    la t3, _plic_ext_vector_table
    add t3, t3, t2      # ext_vector_table + 4*id
    jalr t3             # call interrupt handler

1:
    # epilogue: restore ABI caller regs
    ...

    mret
```

Listing 1.51: Assembly code for the machine external interrupt handler.

The machine external interrupt handler:

a. decodes the interrupt cause (same as in the machine time handler),
b. reads the interrupt source ID from the **claim** register in PLIC,
c. calculates the address of the interrupt handler by adding 4xID to the base address of the PLIC vector table, and
d. calls the interrupt handler.

4. **Set PLIC priorities**. Write the C function to set the interrupt priorities if needed:

```
#define PLIC_INT_PRIORITY_BASE      0x0C000000

/* Set interrupt priority
 *
 * Interrupt source id: 1-52
    * Interrupt priority levels 7
    * Bits 2:0
    * 0 - never interrupt/disables interrupt
    * 1 - lowest active priority
    * 7 - highest priority */

void plic_set_priority(unsigned int source, unsigned int priority){

  *((unsigned int *)PLIC_INT_PRIORITY_BASE + source) = priority;
}
```

Listing 1.52: C function for setting PLIC interrupt priority for a given source.

5. **Enable PLIC source**. Write the C function to enable the specific interrupt source:

```
#define PLIC_INT_ENABLE1            0x0C002000

/*
 * Enable interrupt source in enable registers
 */
void plic_enable_source(unsigned int source){
  unsigned int bit_position = source | 32;
  unsigned int enable_reg = source / 32;

  *((unsigned int *)PLIC_INT_ENABLE1 + enable_reg) |= (1 << ↩
    bit_position);
}
```

Listing 1.53: C function for enabling a PLIC interrupt source.

1.7.3 ARM Cortex-M Nested Vectored Interrupt Controller

The Nested Vectored Interrupt Controller (NVIC) is a crucial component of ARM Cortex-M microcontrollers, including the Cortex-M7. It serves as the central hub for managing and controlling interrupts and exceptions in these processors. Figure 1.36 shows the relation between the NVIC unit, the Processor Core and peripherals. The NVIC supports up to 240 interrupts (IRQ1 to IRQ240), each with up to 256 priority levels (0-255), with a higher level corresponding to a lower priority. The interrupts/exceptions can originate from various sources, such as external peripherals, internal hardware, or system events. The NVIC manages the prioritization of interrupts, allowing the system to handle multiple interrupt requests simultaneously and determine the order in which these interrupts are serviced based on their assigned priority levels.

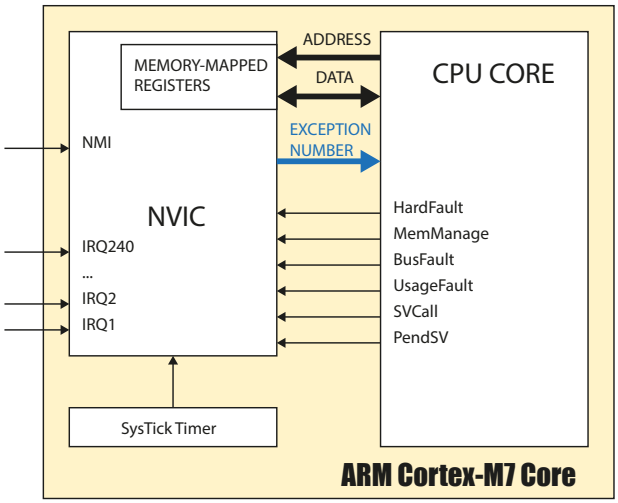

Fig. 1.36: The NVIC controller in the Cortex-M7 core.

One of the unique features of the NVIC is its ability to handle nested interrupts. It allows higher-priority interrupts to preempt the processing of lower-priority interrupts, maintaining the integrity of the system's operation. Moreover, it provides control over enabling and disabling interrupts, allowing the software to manage which interrupt sources are active or inactive. This capability is crucial for managing critical sections of code and ensuring the system's responsiveness.

The NVIC manages the routing of interrupts, determining which interrupt handler (function) should be executed when a specific interrupt occurs. The processor knows where exception handlers are located in memory thanks to exception vectors (i.e. addresses in memory of exception handlers) inside the vector table. The NVIC is responsible for sending exception numbers, which are used as indices of the ex-

ception vectors in the vector table. It enables the CPU to find the interrupt vector and immediately jump to the associated interrupt handler rather than polling interrupt sources to determine which one requested the interrupt. By allowing the CPU to respond to events as they occur (rather than continuous polling of hardware), the NVIC optimizes CPU resources and reduces power consumption.

The processor core interacts with the NVIC through a set of memory-mapped registers, which provide control over enabling and disabling interrupts, allowing the software to manage which interrupt sources are active or inactive.

Developers interact with the NVIC through the CMSIS (Cortex Microcontroller Software Interface Standard). CMSIS is a software interface, which allows programmers to configure interrupts and manage interrupt handling in an efficient and standardized manner. For instance, using the CMSIS software interface, developers can easily assign different priorities to interrupts, enable or disable interrupts, and configure interrupt vectors.

Table 1.9: NVIC registers.

Address	Name	Description
0xE000E100-0xE000E11C	NVIC_ISER0 - NVIC_ISER7	Interrupt Set-Enable Registers
0xE000E180-0xE000E19C	NVIC_ICER0 - NVIC_ICER7	Interrupt Clear-Enable Registers
0xE000E200-0xE000E21C	NVIC_ISPR0 - NVIC_ISPR7	Interrupt Set-Pending Registers
0xE000E280-0xE000E29C	NVIC_ICPR0 - NVIC_ICPR7	Interrupt Clear-Pending Registers
0xE000E300-0xE000E31C	NVIC_IABR0 - NVIC_IABR7	Interrupt Active Bit Register
0xE000E400-0xE000E4EC	NVIC_IPR0 - NVIC_IPR59	Interrupt Priority Register

As mentioned, NVIC comprises several registers that facilitate interrupt configuration, prioritization, and handling. Below are some key registers commonly found in the NVIC of ARM Cortex-M microcontrollers:

1. Eight **Interrupt Set Enable Registers (NVIC_ISER0 - NVIC_ISER7)**, which enable interrupts and show which interrupts are enabled. Each bit in these registers corresponds to a specific interrupt source, allowing individual interrupt control. If a pending interrupt is enabled, the NVIC activates the interrupt based on its priority. If an interrupt is not enabled, asserting its interrupt signal changes the interrupt state to pending, but the NVIC never activates the interrupt, regardless of its priority.
2. Eight **Interrupt Clear Enable Registers (NVIC_ICER0 - NVIC_ICER7)** registers disable interrupts and show which interrupts are disabled. Each bit in these registers corresponds to a specific interrupt source, allowing individual interrupt control.
3. Eight **Interrupt Set Pending Registers (NVIC_ISPR0 - NVIC_ISPR7)** registers force interrupts into the pending state and show which interrupts are pending. They control whether an interrupt is marked as pending or not. Each bit in these registers corresponds to a specific interrupt source, allowing individual interrupt control.

4. Eight **Interrupt Clear Pending Registers (NVIC_ICPR0 - NCVIC_ICPR7)** registers remove the pending status of an interrupt. Each bit in these registers corresponds to a specific interrupt source, allowing individual interrupt control.
5. Eight **Interrupt Active Bit Registers (NVIC_IABR0 - NVIC_IABR7)** registers indicate which interrupts are active. A bit is read as one if the status of the corresponding interrupt is active or active and pending.
6. 60 **Interrupt Priority Registers (NVIC_IPR0 - NVIC_IPR59)** registers provide an 8-bit priority field for each interrupt. These registers are byte-accessible, and there is a total of 240 8-bit priority fields in 60 IPRs.

Table 1.9 lists the memory-mapped NVIC registers and their addresses. The availability and configuration of these registers may vary slightly across different Cortex-M microcontroller variants, so the specific functionalities and register names may differ in certain models.

1.7.3.1 Interrupt Priority Levels

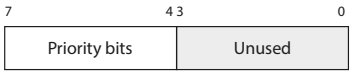

Fig. 1.37: The 8-bit priority field in an Interrupt Priority Register.

In Cortex-M7/ cores, the priority of each interrupt is defined through the corresponding 8-bit Priority field in the IPR register. This 8-bit field allows up to 255 different priority levels. However, in practice, only the four upper bits of this field are used in order to decrease the complexity of NVIC and to lower the power consumption. Figure 1.37 shows how the content of IPR is interpreted. This means that we have only sixteen maximum priority levels. The lower this number is, the higher the priority is. The four priority bits can be further logically subdivided into two parts: a series of bits defining the **preemption priority** and a series of bits defining the **sub-priority**. The preemption priority level rules the preemption priorities between exceptions. If an exception has a priority higher than another one, it will preempt the execution of the lower-priority exception in case it fires. The sub-priority determines which exception handler will be executed first in case of multiple pending exceptions with the same preempt priority, and it will not act on preemption. The way the 4-bit priority field is logically subdivided is called a **priority grouping** and is defined in the ARM Cortex-M7 System Control Block. Once defined, a priority grouping is common to all interrupts used in the system.

Figure 1.38 shows five possible priority groupings in ARM Cortex-M7 processors. In each priority grouping scheme, the most significant bits within the overall priority level represent the preemption priority, which determines the priority between different exceptions and their ability to preempt each other. The least signifi-

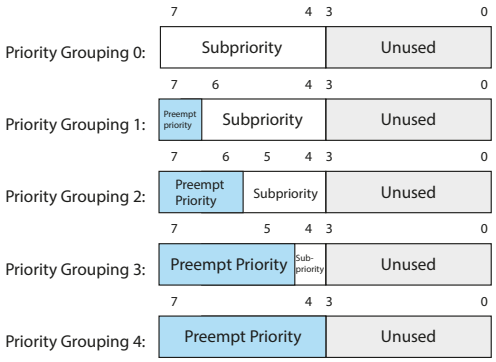

Fig. 1.38: Priority grouping in ARM Cortex-M7.

cant bits within the priority level represent the sub-priority. They manage the order of handling interrupts with the same preemption priority, allowing fine-grained control over which interrupt is serviced first among those with the same preempt bits. The choice of grouping determines the balance between high-priority preemption and the finer-grained management of interrupts with the same high-priority level. For instance, a system configured with three preemption priority bits and one subpriority bit (Priority Grouping 3) allows for eight levels of preemption and two subpriority levels within each preemption level. The priority grouping concept provides a means to tailor the interrupt handling scheme to the specific requirements of an application, allowing for more precise control over the order in which interrupts are processed and handled within the Cortex-M7 architecture.

1.7.4 Case study: External Interrupts in STM32H7xx Microcontrollers

Figure 1.39 shows the block diagram of the interrupt circuitry block connected to the processor core in an STM32H7xx microcontroller. The interrupt circuitry consists of

1. The NVIC unit tightly coupled to the processor core within the ARM-Cortex M7. The internal peripherals within STM32H7xx (e.g. UARTs, timers, etc.) are connected to IRQ lines of the NVIC.
2. An additional dedicated interrupt controller, named Extended Interrupt and Event Controller (EXTI), responsible for the interconnection between the external I/O interrupt signals and the NVIC controller, as we will see next.

The Clock Security System (CSS) in STM32H7xx microcontrollers is a feature designed to enhance the reliability and robustness of clock sources used within the microcontroller. It provides a safeguard against potential failures in the clock system to

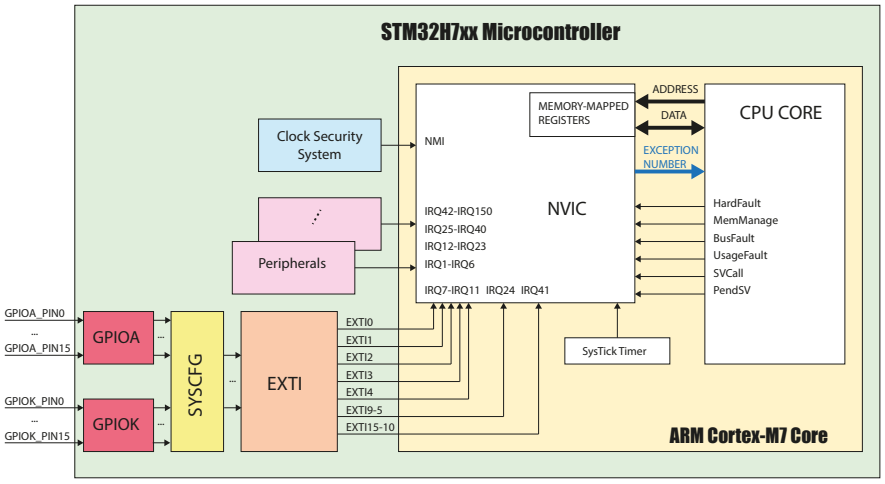

Fig. 1.39: The NVIC and EXTI controllers in the STM32H7xx Microcontrollers.

ensure the proper functioning of the microcontroller in various operating conditions. The CSS is connected to the NMI input of the NVIC controller. The STM32H7xx microcontroller contains 11 16-bit GPIOs named GPIOA through GPIOK. In total, there are 176 GPIO pins which can be used to generate external interrupt requests. In STM32X7xx microcontrollers, the EXTI is used to generate interrupts from GPIO pins. The EXTI is a peripheral that enables interrupt requests based on specific GPIO pin events, such as a rising or falling edge, allowing external events to trigger interrupt requests.

1.7.4.1 Extended Interrupt and Event Controller (EXTI)

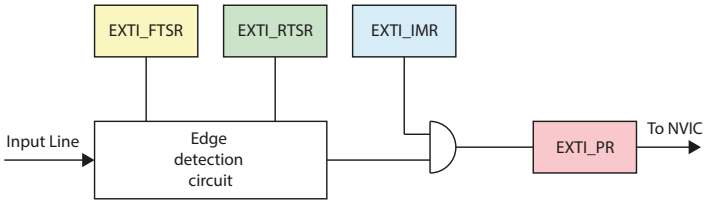

Fig. 1.40: EXTI block diagram.

Figure 1.40 shows the block diagram of the EXTI controller. The main features of the EXTI controller are an independent trigger and mask on each interrupt/event

line and a dedicated pending (status) bit for each interrupt line. The EXTI controller manages 25 input interrupt lines in total. The EXTI includes memory-mapped registers that allow the programmer to set interrupt trigger conditions (rising edge, falling edge or both), and enable interrupts on specific EXTI lines. Some of the EXTI registers, which we are interested in in this subsection, are:

1. **Interrupt mask register (EXTI_IMR)** manages the interrupt mask status for each EXTI line. Setting a bit in this register enables the interrupt from that line.
2. **Rising trigger selection register (EXTI_RTSR)** enables/disables the rising trigger for each EXTI input line. When enabled, the EXTI generates an interrupt request when a rising edge is detected on an input line.
3. **Falling trigger selection register (EXTI_FTSR)** enables/disables falling trigger for each EXTI input line. When enabled, the EXTI generates an interrupt request when a falling edge is detected on an input line.
4. **Pending register (EXTI_PR)** indicates the pending status of the interrupt for each EXTI line. Reading a bit in this register shows if an interrupt request is pending on that line. We should write '1' to the bit in the interrupt handler to clear the pending state of the corresponding interrupt.

The EXTI controller has internal interrupt control logic that monitors the GPIO pins' status, and triggers interrupt requests when the configured events occur. To generate the interrupt, the interrupt line should be configured and enabled. This is done by programming the two trigger registers (EXTI_RTSR and EXTI_FTSR) with the desired edge detection and by enabling the interrupt request by writing a '1' to the corresponding bit in the interrupt mask register (EXTI_IMR). When the selected edge occurs on the external interrupt line, and the interrupt on that line is enabled in the interrupt mask register (EXTI_IMR), the pending bit corresponding to the interrupt line is set in the pending register (EXTI_PR). Also, EXTI generates an interrupt request to the NVIC. Once the interrupt handler is executed, this request must be reset by writing a '1' in the pending register (EXTI_PR) from the interrupt handler. EXTI interrupts can be individually prioritized using the NVIC, allowing different external events to have different levels of priority. To configure a line as an interrupt source, we use the following procedure:

1. Set the corresponding mask bit (EXTI_IMR)
2. Configure the trigger selection bits of the interrupt lines (in the EXTI_RTSR and EXTI_FTSR registers)
3. Configure the enable and mask bits that control the NVIC IRQ channel mapped to the external interrupt controller (EXTI) so that an interrupt coming from one of the 25 lines can be correctly acknowledged.

1.7.4.2 System configuration controller (SYSCFG)

As we already said, there are 176 GPIO pins in STM32H7xx microcontrollers, which can generate external interrupt requests. All these pins are routed to 16 input lines in the EXTI controller using the System Configuration controller. In

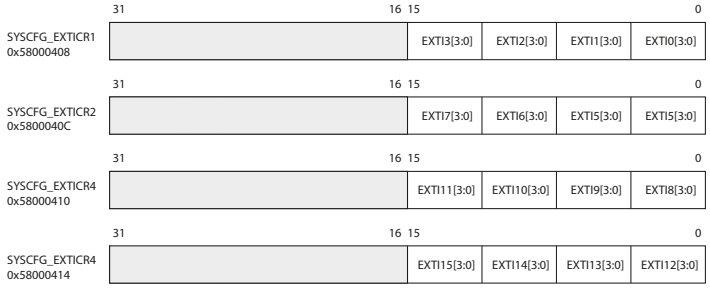

Fig. 1.41: The SYSCFG external interrupt configuration registers and their addresses.

STM32 microcontrollers, including the STM32F7 series, the System Configuration (SYSCFG) controller is crucial in routing GPIO pins to the External Interrupt (EXTI) lines. This routing process involves four memory-mapped specific registers within the SYSCFG (SYSCFG_EXTICR1 to SYSCFG_EXTICR4) and 16 multiplexors in the SYSCFG. Figure 1.41 shows the EXTI configuration registers available in the SYSCFG module. Four SYSCFG registers, SYSCFG_EXTICR1 to SYSCFG_EXTICR4, contain 16 groups of four select bits for 16 multiplexers, allowing the configuration of which of 16 GPIO pins is connected to a specific EXTI line. The EXTI line multiplexing functionality provided by SYSCFG enables the routing of GPIO pins to EXTI lines, as presented in Figure 1.42.

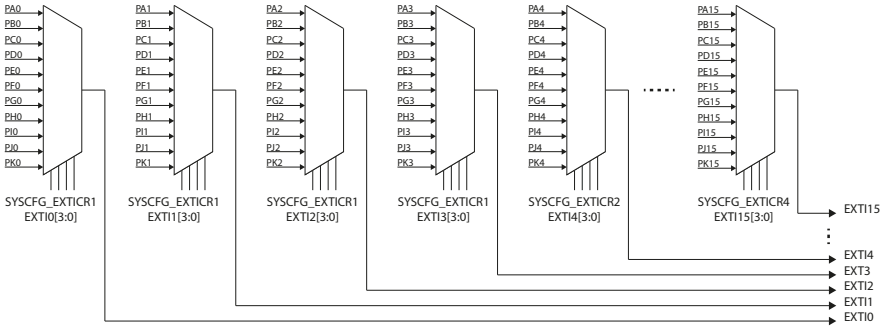

Fig. 1.42: The mapping of GPIO pins to EXTI lines in the System Configuration (SYSCFG) module in an STM32H7 MCU.

The 16 EXTI controller lines, EXTI0 to EXTI15, are connected to the NVIC controller using only 7 IRQ inputs. The EXTI0, EXTI1, EXTI2, EXTI3 and EXTI4 lines are connected to their dedicated NVIC IRQ inputs IRQ7 to IRQ11 (see Figure 1.39). The EXTI lines, EXTI5 to EXTI9, share the NVIC IRQ24 input, and the

EXTI lines EXTI10 to EXTI5 share the NVIC IRQ41 input (see Figure 1.39). Figure 1.43 illustrates how EXTI lines, EXTI0 to EXTI15, are mapped to exception handlers in ARM Cortex-M7 cores.

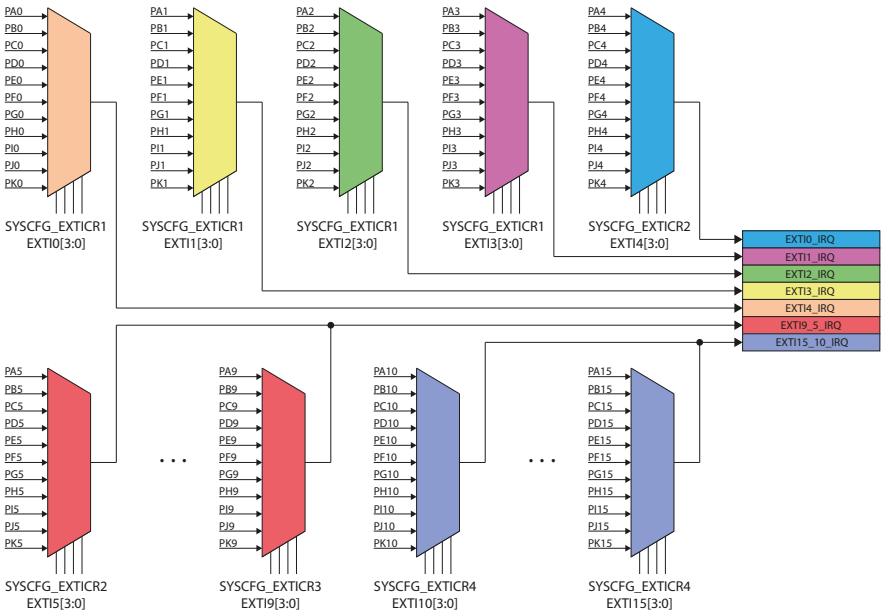

Fig. 1.43: The mapping of the EXTI lines to exception numbers (handlers).

1.7.4.3 Triggering interrupts on the GPIO pins

Suppose we want to trigger interrupts when the rising edge is detected on GPIOC Pin 13. The process of setting up the STM32H7xx system involves the following steps:

1. **GPIO Pin Configuration:** Configure GPIOC Pin13 as input.
2. **SYSCFG Configuration:** After configuring the GPIOC Pin13, we need to map the pin 13 to the EXTI13 line. This involves configuring the associated multiplexor in the SYSCFG controller. To configure the multiplexor which maps GPIOC Pin13 to EXTI13, we need to write '0010' to the 4-bit field EXTI13[3:0] in the SYSCFG_EXTICR4 register.
3. **EXTI Configuration:** Set the triggering conditions for the EXTI line 13 associated with GPIOC Pin13. As we want the interrupt to be triggered by a rising edge, we should set the bit associated with EXTI13 in the Rising trigger selection register (EXTI_RTSR). We should also enable the interrupt associated with

EXTI13 by setting the appropriate bit in the interrupt mask register (EXTI_IMR).

4. **NVIC Configuration:** Configure the NVIC controller. This involves enabling the IRQ41 in the Interrupt Set Enable Register NVIC_ISER1 and setting the priority level for IRQ41 in the priority register NVIC_IPR11.
5. **Exception Handler Implementation:** Implement the exception handler `EXTI15_10_IRQHandler()` for the NVIC IRQ line associated with EXTI13. When the configured event occurs on the GPIOC Pin13, the EXTI generates an interrupt request on NVIC IRQ41, and the corresponding handler is called. Within the ISR, perform the necessary actions in response to the external event. The mandatory part of the handler is to clear the peripheral pending bit (in the EXTI_PR register). The peripheral pending bit will be held high until it is cleared by the application code. If the peripheral pending bit is not cleared, the interrupt will be fired again, and the handler will run again.

Figure 1.44 illustrates how the interrupt request on the GPIOC Pin13 is routed through SYSFIG, EXTI and NVIC to the CPU core after performing the above-listed configuration steps.

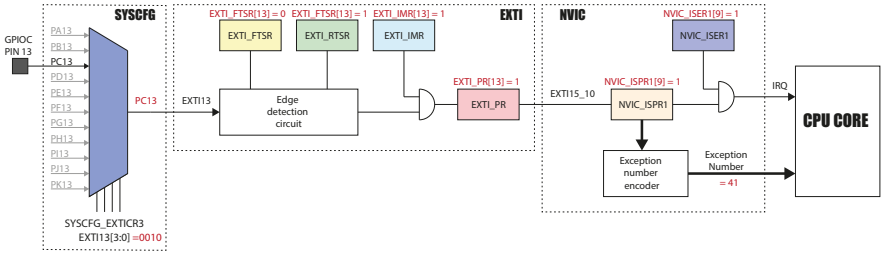

Fig. 1.44: Routing the interrupt request on the GPIOC Pin13 through SYSFIG, EXTI and NVIC to the CPU core.

To enable an external interrupt on GPIOC Pin13 in the STM32H7xx microcontroller, we typically use the STM32Cube HAL (Hardware Abstraction Layer) library provided by STMicroelectronics, which simplifies configuring and using the microcontroller's peripherals. Below are the general steps to enable an external interrupt on GPIOC Pin13:

1. Configure GPIOC Pin13, SYSCFG and EXTI using `HAL_GPIO_Init()` function:

```
GPIO_InitTypeDef GPIO_InitStruct = {0};

__HAL_RCC_GPIOC_CLK_ENABLE();

GPIO_InitStruct.Pin = GPIO_PIN_13;
GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
```

```
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
```

Listing 1.54: GPIO, SYSCFG and EXTI configuration.

The `HAL_GPIO_Init()` function sets the GPIO Pin13 as input, configures the SYSCFG controller to route GPIOC Pin13 to the EXTI13 line, and configures the EXTI controller to fire an interrupt when the rising edge occurs on the EXTI13 line.

2. Configure NVIC controller:

```
HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
```

Listing 1.55: HAL functions used to configure NVIC.

3. Implement the `EXTI15_10_IRQHandler()` handler:

```
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 EXTI13 pending bit in EXTI pending register
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_13);
  }
}
```

Listing 1.56: EXTI15_10_IRQ Handler.

1.7.5 Intel 8259A Programmable Interrupt Controler

Intel processors also have only a single interrupt input. As a personal computer has several peripheral devices that can raise interrupts, the Intel Programmable Interrupt Controller (PIC) 8259A is used to manage them. The 8259A PIC is a special interrupt controller designed particularly for Intel processors. It is connected between the interrupt requesting peripheral device and the Intel processor. This means that the interrupt requests from peripheral devices are first transferred to the PIC, which in turn asserts the processor's interrupt input. Figure 1.45 illustrates the system with the 8259A PIC.

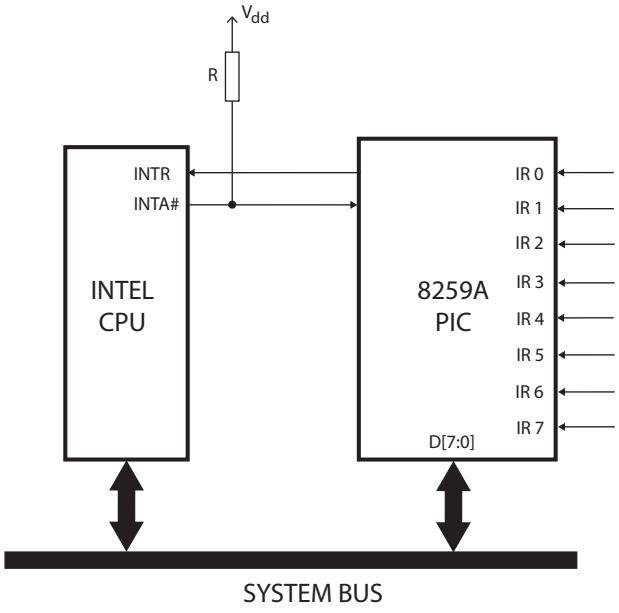

Fig. 1.45: A system with the 8259A PIC.

The 8259A was introduced in the early 1980s and was used in personal computers until the 1990s. It is still used in some Intel-based embedded systems. While not anymore a separate chip, the 8259A interface is still provided by the chipset on modern x86 motherboards. Although someone could say it is obsolete, its functioning will help us to understand the evolution of interrupt controllers in the Intel-based computer systems.

The Intel 8259A Programmable Interrupt Controller handles up to eight vectored priority interrupts for the CPU. It is cascadable for up to 64 vectored priority interrupts without additional circuitry. The interrupt inputs have fixed priority based on

its number, and the interrupts on its inputs may be either edge-triggered or level-triggered.

The 8592A PIC has the following set of registers: Interrupt Request Register (IRR), In-Service Register (ISR), and Interrupt Mask Register (IMR). The IRR register specifies which interrupts are pending. The ISR register specifies which interrupts have been acknowledged, and the IMR specifies which interrupts are to be ignored and not acknowledged. Figure 1.46 illustrates the simplified internal structure of the 8259A PIC.

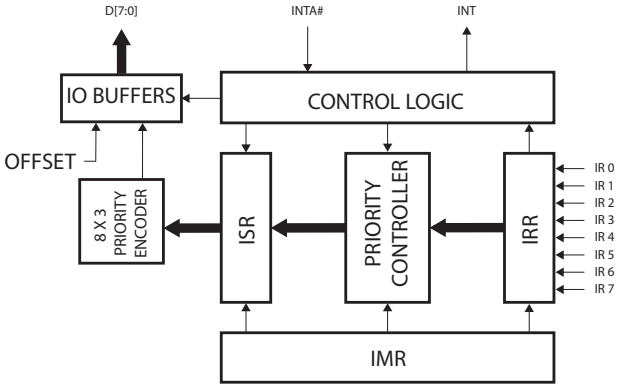

Fig. 1.46: Simplified internal structure of the 8259A PIC.

The peripheral device that wishes to issue an interrupt request asserts one of the pins IR0 to IR7. If the interrupt is not masked in the IMR register, the 8259A PIC will set the corresponding bit in the interrupt request register (IRR). The IRR register remembers all the pending interrupt requests. As more peripheral devices can issue the interrupt request simultaneously, several bits may be set in the IRR register at the same time. At the same time, the 8259A sends an INT to the CPU. When the 8259A PIC asserts the interrupt request on the processor's INTR input, the processor recognizes this request on the next instruction fetch. It then stops the instruction fetch and automatically saves the program context onto the stack. The CPU then starts the so-called **interrupt-acknowledge cycle**.

Interrupt-acknowledge cycles are special bus cycles that enable the PIC to output an interrupt vector onto the data bus. This vector is fetched by the CPU and transferred into the program counter during the interrupt-acknowledge cycle. The value read during the interrupt-acknowledge cycle is then multiplied by 4 and used to load an interrupt vector from this address in memory. The Intel processors perform two back-to-back interrupt-acknowledge cycles in response to an active INTR input:

1. Firstly, the CPU responds by asserting the first INTA pulse. Upon receiving an INTA from the CPU, the priority controller in the 8259 passes the highest priority bit from IRR to the In-Service Register (ISR), and the corresponding

IRR bit is reset. The set bit in the ISR indicates which interrupt request is being serviced.

2. Secondly, the processor asserts the second INTA pulse to instruct the 8259A to release an 8-bit interrupt number onto the Data Bus (D0-D7). This ends the interrupt-acknowledge sequence.

The CPU now reads the 8-bit interrupt number (n) and multiplies it by 4. This value represents the address of the memory location that holds the start address of the interrupt handler. Hence, the CPU executes in hardware the following operation:

```
PC <-  Mem[n x 4]
```

The structure of the 8-bit vector number returned from the 8259A PIC is shown in Figure 1.47. The lower three bits are the binary-coded number of the bit that was set in the ISR. The higher five bits are the offset that can be programmed during the 8259A PIC initialization. Recall, that the Intel stores its IVT table at the address 0x0000 and that the interrupt numbers 32 through 255 are reserved for external interrupts signaled on the INTR pin. If we want to map the IRQ interrupts from 8259A PIC at the address 0x0080 (=32x4) in the IVT, the offset returned in the 8-bit vector number should be 00100. In the case of the IR0 interrupt, the returned vector number is 0x20, which maps to 0x0080; in the case of the IR1 interrupt, the returned vector number is 0x21, which maps to 0x0084, etc.

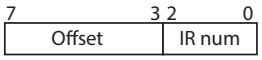

Fig. 1.47: The 8-bit vector number returned by the 8259A PIC. The lower three bits are the binary-coded number of the bit that was set in the ISR. The higher five bits are the offset that can be programmed during the 8259A PIC initialization.

To reset the bit in the ISR register, the interrupt handler should issue an End-Of-Interrupt (EOI) command to the 8259A PIC. The set bit is therefore deleted manually. The 8259A is now ready to process the next pending hardware interrupt request in IRR. The priority controller passes the highest priority bit form IRR to ISR, and the above sequence is repeated. Figure 1.48 illustrates the operation of the 8259A PIC when IR2 and IR4 are issued simultaneously, and IR2 has a higher priority than IR4.

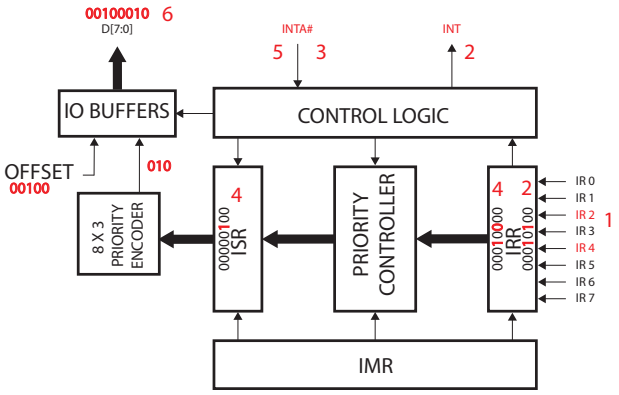

Fig. 1.48: the operation of the 8259A PIC when IR2 and IR4 are issued simultaneously, and IR2 has a higher priority than IR4. (1) Two peripheral devices assert the pins IR2 and IR4. (2) Assuming the interrupts are not masked in the IMR register, the 8259A PIC will set the corresponding bits in the interrupt request register (IRR). At the same time, the 8259A sends an INT to the CPU. (3) The CPU responds by asserting the first INTA pulse. (4) Upon receiving an INTA from the CPU, the priority controller in the 8259 passes the highest priority bit from IRR to ISR, and the corresponding IRR bit is reset. (5) The processor asserts the second INTA pulse. (6) The 8259A controller releases an 8-bit interrupt number onto the data bus (D0-D7).

Summary: AIC vs. 8259A

Besides being designed for different processors, the main difference between the ARM AIC and Intel 8259A PIC is how the interrupt vector is obtained. In ARM AIC, the CPU reads the interrupt vector from an AIC's memory-mapped register using a LOAD instruction, while in 8259A PIC, the CPU reads the vector from the data bus, without executing any instruction.

The former is considered faster (recall that instructions in ARM9 are executed in 5 clock cycles) but requires additional signaling between interrupt controller and CPU (INTA) and a special interrupt-acknowledge cycle.

1.7.6 8259A PIC Cascading

With one 8259A PIC, eight interrupt sources could be managed. But soon, eight interrupt lines weren't enough. The 8259A PIC has the capability for **two-level cascading**. The first level is made up of one **master** PIC, and the second level is formed

from up to eight **slave** PICs. Such a configuration can manage up to 64 peripheral interrupt requests. But in practice, only two PCs are used: one is the master, and the other is the slave. The PCs included two 8259A PICs chained together, and this setup became de facto standard for the x86 platform. This scheme is referred to as the dual-PIC system. Figure 1.49 illustrates the dual-PIC system.

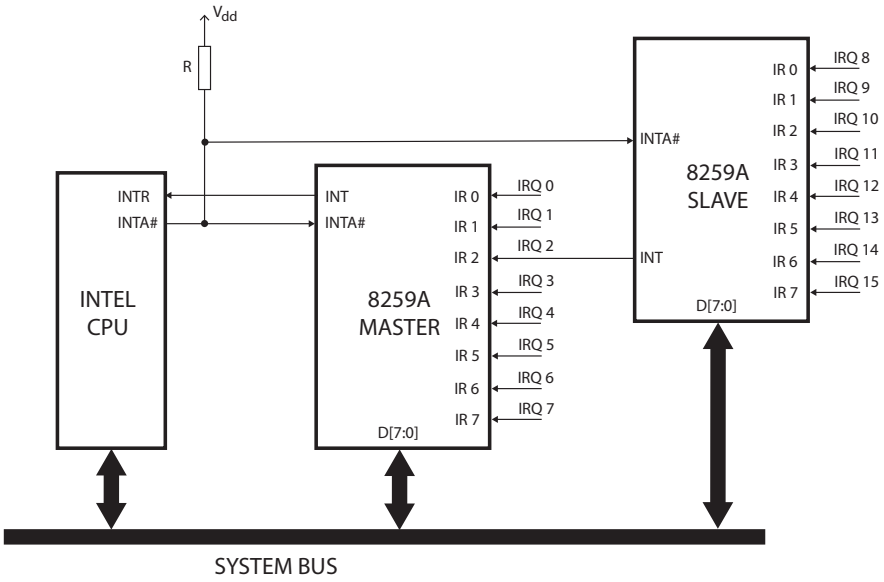

Fig. 1.49: A dual-PIC system.

In the dual-PIC system, the slave's INT output is connected to the master's IR2 input. PC documentation established the following naming convention: IRQs 0 through IRQ7 are processed with the first Intel 8259 PIC (master), and IRQs from 8 to 15 are processed with the second Intel 8259 PIC (slave). Therefore, the slave's INT output is connected to the master's IRQ2 input. Only the master's INT output is connected to the CPU's INTR input and can signal about the incoming interrupts. INTA and D0 through D7 signals of both PICs are connected to the CPU and data bus as in the single-PIC configuration. Note that IRQ 2 is not available for device interrupts, and there are only fifteen interrupt inputs available for peripheral device interrupts.

The way in which an interrupt request is processed depends on whether the request is asserted on slave's or master's IRQ inputs. If the request is asserted on IRQs form 0 to 7 (master), it is processed in the same way as in the single-PIC configuration. Otherwise, the following steps are required: when an interrupt request is placed on lines IRQ 8 through 15, the corresponding bit is set in the slave's IRR register. The slave asserts its INT output and signals the IRQ interrupt to the master PIC,

when master PIC receives the interrupt request on IRQ2, it sets the bit 2 in its IRR and asserts its INT output to signal the INTR request to the processor, the CPE starts the interrupt-acknowledge sequence and sends the first INTA pulse. Upon receipt of the first INTA pulse, the highest priority bits in the master's and slave's IRRs are cleared, and the corresponding bits in both ISRs are set, the CPE outputs the second INTA pulse and causes the slave PIC to output its 8-bit vector number. The interrupt handler must send two EOI commands to clear both ISR bits.

But wait! How can we set two different vector numbers for interrupts signaled at the master's IR3 (IRQ3) input and at the slave's IR3 input (IRQ11)? Recall, that the 8-bit vector number returned from the 8259A PIC contains a programmable offset in its higher five bits. Hence, the master and slave PICs should be initialized with different offsets. For example, we can set the master's offset to 00100 and the slave's offset to 00110. In this case, the interrupts from the master PIC will be mapped to the IVT addresses 0x0080-x009F, and the slave's interrupts will be mapped to the IVT addresses 0x00C0-0x00DF.

The original PCs used the ISA bus for its I/O devices. The interrupts on the ISA bus are edge-triggered. An I/O device asserts an interrupt by raising the signal from low to high. Edge-triggered interrupts inhibit the sharing of ISA interrupts by multiple devices, so each ISA device requires a dedicated interrupt input on the 8259As. A typical interrupt configuration at that time is presented in Table 1.10.

Table 1.10: ISA Interrupt Assignments.

IRQ	Assignment
0	System timer
1	Keyboard controller
2	interrupt from slave controller
3	Serial ports COM 2 / COM 4
4	Serial ports COM 1 / COM 3
5	Sound card
6	Floppy disk controller
7	Parallel port 1 (Printer)
8	Real-time clock
9	ACPI
12	PS/2 mouse controller
13	Math (floating point) co-processor
14	ATA channel 1 (Primary IDE)
15	ATA channel 1 (Secondary IDE)

The Intel 8259 PIC has several limitations to interrupt servicing in modern computer systems:

1. a limited number of interrupt lines necessitates the sharing of interrupts. Shared interrupts require the OS to poll multiple IO devices to determine who actually generated the interrupt,
2. interrupt priority is fixed based on IR number,

3. PIC does not support multiple CPUs.

At the time when the main bus for external devices was the ISA bus, this 8929A based architecture was sufficient. It was only necessary that different peripheral devices did not connect to the same IRQ inputs since ISA uses the edge-triggered interrupts, which are not shareable. But the PCI bus later replaced the ISA bus, and interrupts in the PCI bus can be shared. Also, the PCI bus has been replaced by the serial message-based PCI Express (PCIe) bus, and more CPU cores are added to the computer system. The following sections will cover the evolution of the interrupt controller used in the Intel-based computer systems, and we will learn how to handle the interrupts on the PCI bus where the number of peripheral devices exceeds the number 15, how to share the interrupt lines on the PCI bus, and finally how to handle interrupts in the modern multi-core PCIe-based systems.

1.7.7 Intel Advanced Programmable Interrupt Controler

By nature, the 8259A PIC can only send interrupts to one CPU, and in a multiprocessor system, it is desired to load CPUs in a balanced way. The solution to this problem was the new APIC (Advanced PIC) architecture. This architecture addressed many of the limitations of the older PIC-based architecture. The most apparent is the support for multiple CPUs.

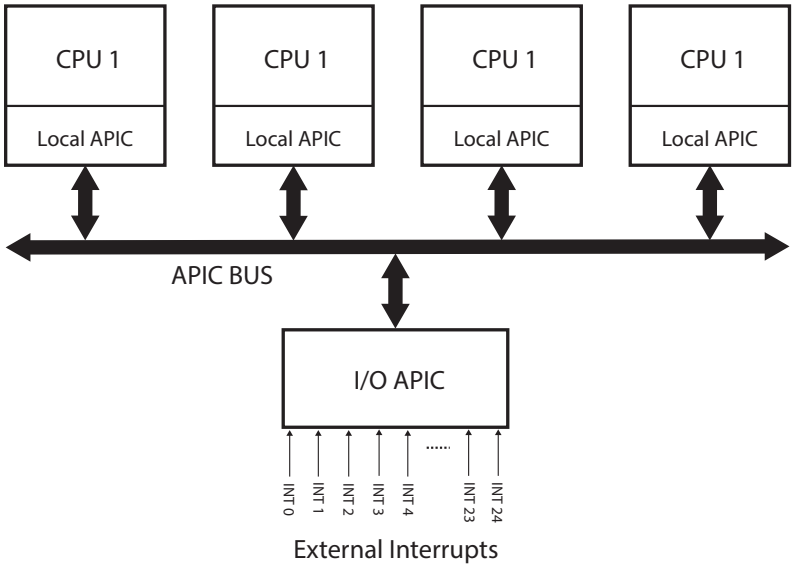

Fig. 1.50: An APIC based computer system.

At the system level, APIC consists of two parts (Figure 1.50). One part resides in the I/O subsystem and is called the I/O APIC. It is responsible for routing interrupts from external devices to the other part, the Local APIC (LAPIC), which resides in each CPU. The local APIC and the I/O APIC communicate over a dedicated 3-wire serial APIC bus. The IOAPIC bus interface consists of two bi-directional data signals (APICD[1:0]) and a clock input (APICCLK). The modern systems may use a standard system bus instead of a separate APIC bus for this task. It is worth noting that it is possible to have several I/O APIC controllers in the system. For example, one for 24 interrupts in a southbridge, and the other one for 32 interrupts in a northbridge. The CPU's Local APIC contains the necessary intelligence to determine whether or not its processor should accept interrupts broadcast on the APIC bus. The Local APIC also provides local pending of interrupts and handles all interactions with its local processor (e.g., the interrupt acknowledge sequence). Additionally, each I/O APIC has 24 interrupt lines and allows the priority of each interrupt to be set independently. The I/O APIC sends an interrupt vector to the local APIC,

and, as a result, the OS does not have to interact with the I/O APIC until it sends the end of interrupt notification.

Summary: APIC

In the APIC-based systems, each CPU includes a local APIC which receives interrupt messages and uses them to assert interrupts on the CPU. The chipset includes one or more I/O APICs which are responsible for converting device interrupt signals into messages that are delivered to one or more local APICs.

1.7.7.1 Local APIC

The Local Advanced Programmable Interrupt Controller (LAPIC) was introduced into the Pentium processor and is included in more recent Intel processor families. The local APIC performs two primary functions for the processor: it receives interrupts from the processor's interrupt pins and an external I/O APIC. It sends these to the processor core for handling. In multiple-processor systems, it sends and receives interprocessor interrupt (IPI) messages to and from other processors on the system bus. IPI messages are used to distribute interrupts among the processors in the system. When a local APIC sends an interrupt to its processor core (by asserting the processor's INTR line) for handling, the processor uses the interrupt and exception handling mechanism described in Section 1.6.

The LAPIC receives interrupts from several sources:

- Locally connected I/O devices: these interrupts are asserted by an I/O device connected directly to the processor's local interrupt pins (LINT0 and LINT1).
- Inter-processor interrupts (IPIs): an Intel processor can use the IPI messages to interrupt another processor or group of processors on the system bus.
- Externally connected I/O devices: these interrupts are asserted by an I/O device connected to the interrupt input pins of an I/O APIC. Interrupts are sent as IPI messages from the I/O APIC to one or more LAPICs in the system.

Figure 1.51 illustrates a simplified internal structure of a Local APIC. The heart of the LAPIC is very similar to the 8259A: it contains the IRR and ISR registers and a priority controller. Besides, it contains two registers that form the local vector table LVT and the interrupt command register (ICR). In fact, the LAPI is a rather complicated device containing a large set of addressable registers, timers, and other control logic. Figure 1.51 shows only its vital parts that are necessary to understand its interrupt handling. The Protocol Transition Logic block receives the IRI messages from the APIC bus. If the LAPIC is the destination, the Protocol Transition Logic block forwards the destination mode and the vector number from the IRI message to the Acceptance Logic block, which decodes the 8-bit vector number and forwards the bit from the decoded 256-bit word into the IRR register. The rest of the internal logic is described for each interrupt source in the text below.

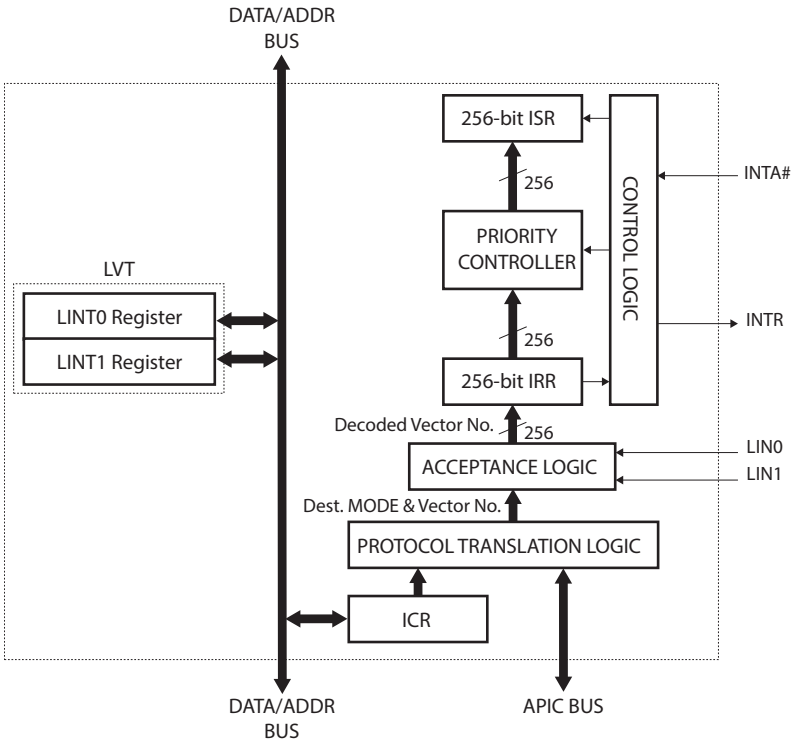

Fig. 1.51: Simplified internal structure of a Local APIC.

Interrupts from locally connected I/O devices. Upon receiving a signal from the processor's LINT0 and LINT1 pins, the local APIC delivers the interrupt to the processor core using a group of APIC registers called the **local vector table**. A separate entry (i.e., a separate register) is provided in the local vector table for each local interrupt pin (LINT0 and LINT1). For example, if the LINT1 pin is going to be used as an NMI pin, the LINT1 entry in the local vector table can be set up to deliver an interrupt with vector number 2 (NMI interrupt in Table 1.7) to the processor core. The LVT consists of two 32-bit registers: LINT0 Register (specifies the interrupt number when an interrupt is signaled at the LINT0 pin) and LINT1 Register (specifies interrupt number when an interrupt is signaled at the LINT0 pin). An interrupt number is an 8-bit number stored in the bits 0 through 7 in each LIN register.

Inter-processor interrupts (IPIs). A processor generates IPIs by writing to a special LAPIC register called the interrupt command register (ICR) in its local APIC. The act of writing to the ICR causes an IPI message to be generated and issued on the system bus or the APIC bus. An IPI message includes the processor destination number, the vector number, and trigger mode (edge or level). When the target pro-

cessor receives an IPI message, its local APIC handles the interrupt request automatically using information included in the message such as vector number and trigger mode. The IPI mechanism is used in multi-processor systems to send or forward interrupts for a specific vector number. For example, a local APIC can use an IPI to forward an interrupt to another processor for servicing. Also, the IPI mechanism is used by I/O APIC to send an interrupt for a specific vector number that originates from an I/O device connected to I/O APIC. The interrupt command register (ICR) is

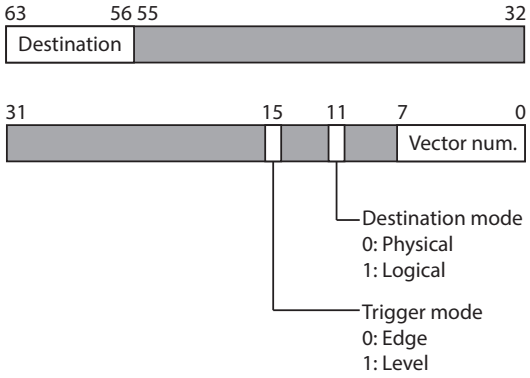

Fig. 1.52: The LAPIC ICR register.

a 64-bit local APIC register that allows software running on the processor to specify and send interprocessor interrupts (IPIs) to other processors in the system. The act of writing to the low 32 bits of the ICR causes the IPI message to be sent. Figure 1.52 illustrates the ICR register (only the bits that are important for understanding are specified/show). The 8-bit Destination field specifies the target processor. The Destination Mode bit further specifies if the destination is either physical (0) or logical (1) processor. The Trigger Mode bit selects the trigger mode: edge (0) or level (1).

Externally connected I/O devices. The local APIC can also receive interrupts from externally connected devices through the I/O APIC (see Figure 1.50). The I/O APIC is responsible for receiving interrupts generated by system hardware and I/O devices and forwarding them to the local APIC as IPI messages. Each individual pin on the I/O APIC can be programmed to generate a specific interrupt vector when asserted. This vector is then sent to LAPIC as a part of an IPI message.

The local APIC handles the interrupts as follows:

1. if it receives a message on the APIC bus, it determines if it is the specified destination. If it is the specified destination, it accepts the message; otherwise, it discards the message,

2. if the local APIC determines that it is the designated destination for the interrupt, the local APIC sets the appropriate bit in the IRR,
3. when interrupts are pending in the IRR register, the local APIC sends them to the processor one at a time, based on their priority, similarly as in the 8259A. The processor responds with the interrupt acknowledge sequence. During the first INTA pulse, the LAPIC moves the highest priority bit form the IRR to ISR. During the second INTA pulse, the LAPIC puts the interrupt vector on the data bus. If the interrupt request comes from a locally connected I/O device (at the LINT0 or LINT1 pins), the interrupt number is stored in the corresponding LVT entry in the LAPIC. If the interrupt request comes from a message, the interrupt number is contained in the the message.

Completing the handler routine is indicated by instruction in the instruction handler code that writes to the end-of-interrupt (EOI) register in the local APIC. The act of writing to the EOI register causes the local APIC to delete the interrupt from its ISR.

1.7.7.2 I/O APIC

The I/O Advanced Programmable Interrupt Controller (IOAPIC) provides multiprocessor interrupt management and incorporates interrupt distribution across all processors. In systems with multiple I/O subsystems, each subsystem can have its own set of interrupts. Each interrupt pin is individually programmable as either edge or level triggered. The interrupt vector can be specified per interrupt input.

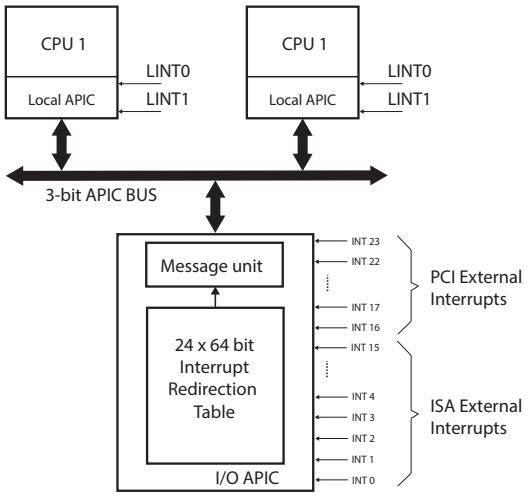

Fig. 1.53: The LAPIC ICR register.

The I/O APIC (Figure 1.53) consists of 24 interrupt input lines, a 24-entry Interrupt Redirection Table (IRT) with 64-bit entries, programmable registers, and a message unit for sending and receiving messages over the APIC bus. I/O devices signals interrupt requests by asserting one of the interrupt lines to the I/O APIC. The I/O APIC selects the corresponding entry in the IRT and uses the information in that entry to format an interrupt request message. Each entry in the IRT contains:

- a bit that indicates edge/level sensitive interrupt,
- the interrupt vector and priority, and
- the destination processor.

The information in the IRT entry is used to form and transmit an IRI message to other LAPICs via the APIC bus.

When an external interrupt request is signaled on the I/O APIC interrupt input, the I/O APIC controller will send an interrupt message to the LAPIC of one of the system CPUs. In this way, the I/O APIC controller helps balance interrupt load between processors.

APIC messages come in several formats and different lengths. Here we present only two types of APIC messages: EOI Message and the so-called Short Message. Local APICs use EOI messages to send an end-of-interrupt (EOI) occurring for a level-triggered interrupt to an I/O APIC. This message is needed, so the I/O APIC knows when an interrupt has been serviced. In this way, the I/O APIC can differentiate between a new interrupt on the interrupt line versus the same interrupt on the interrupt line. I/O APICs use Short Messages for the delivery of external interrupts to local APICS.

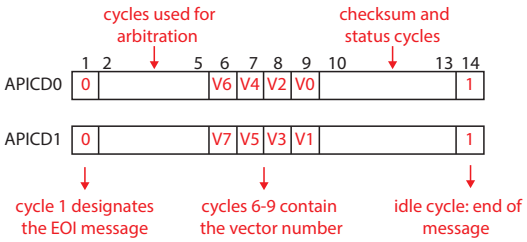

Fig. 1.54: The format of the EOI message.

The format of the EOI message is presented in Figure 1.54. All EOI messages are 14 bits long and take 14 cycles on the APIC bus to transmit. Local APICs send 14-cycle EOI messages to the I/O APIC to indicate that a level triggered interrupt has been accepted by the processor. This message is a result of software writing into the EOI register of the local APIC. The first cycle is used to designate an EOI message. The vector number is sent in cycles 9 through 12. The local APIC gives the target of the EOI message by transmitting the interrupt vector number (V7 through V0). When this message is received, the I/O APIC resets the IRR bit for that interrupt. If

the interrupt signal is still active after the IRR bit is reset, the I/O APIC treats it as a new interrupt. The last cycle in which both data lines are set high is used to signal end-of-message.

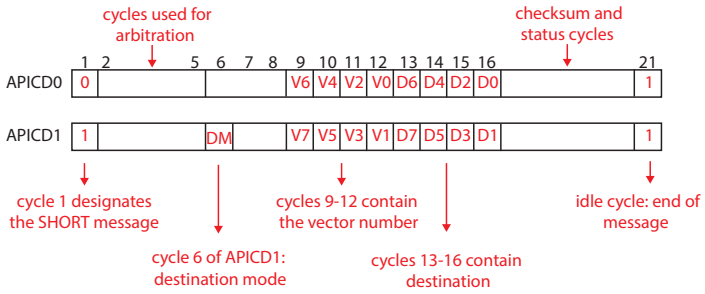

Fig. 1.55: The format of the SHORT message.

The format of a SHORT message is presented in Figure 1.55. All SHORT messages are 21 bits long and take 21 cycles on the APIC bus to transmit. Short messages are used by I/O APICS for sending external interrupts to local APICs. The first cycle is used to designate a SHORT message. The vector number is sent in cycles 9 through 12. If Destination Mode (DM) is 0, then cycles 15 and 16 are the local APIC ID, and cycles 13 and 14 are sent as 1. In this case, the message is sent to a physical processor. If DM is 1, then cycles 13 through 16 are the 8-bit Destination field that selects the logical processor. The last cycle in which both data lines are set high is used to signal end-of-message.

The I/O APIC with 24 input interrupt lines was used in the systems with the PCI bus. The APIC architecture could support up to 16 CPUs. The I/O APIC provided backward compatibility with the older 8259A PIC-based systems. Interrupts 0-15 were used for old ISA interrupts for compatibility with older systems, and interrupts 16-23 were meant for all the PCI devices. With this delimitation, all conflicts between ISA edge-triggered and PCI level-triggered interrupts could be easily avoided. This assignment of interrupts 0-15 provided only eight additional interrupts, which forced the sharing of PCI interrupts - two ore more devices on the PCI bus were forced to share the same I/O APIC's input interrupt line. We will cover the PCI interrupts sharing and routine in the following sections.

One of the biggest differences between the 8259A PICs and I/O APICs is that the pins on I/O APICs are completely independent. With the 8259A PICs, the eight input pins are mapped to eight consecutive vectors in IDT (or IVT), and all of the interrupts are sent to the same CPU. In I/O APICs, on the other hand, each pin is programmed independently. Each pin is assigned its own vector by the operating system and can be mapped to one or more CPUs.

1.8 PCI interrupts

The PCI bus was added to the PCs in the mid-1990s. In the first years, both buses, PCI and ISA, coexisted in the systems. In PCI, the term **device** refers to a piece of hardware that is plugged-in into the PCI slot and contains from one to eight **functions**. A multi-function PCI device is a physical PCI expansion board that embodies between two and eight PCI functions. For example, a single PCI device may include several USB controllers as functions. Another example would be a PCI card with a high-speed communications port and a parallel port. Hence, a PCI expansion card inserted in the PCI expansion slot is a PCI device, and a single expansion card may contain up to eight functions. However, from the operating system's perspective, each function on a PCI device is a logical operating device.

PCI allows devices to assert interrupts in two different ways:

1. The first way uses dedicated interrupt signals (lines) and is known as **Legacy INTx interrupts**.
2. The second way uses special memory writes that are sent over the data bus, just like APIC messages, and is known as **Message Signaled Interrupts (MSI)**. First, we will cover Legacy INTx interrupts. Later, we are going to cover the MSI interrupts.

1.8.1 PCI Legacy interrupts

A PCI card in a slot may have up to eight functions on it, but there are only 4 PCI interrupt pins: INTA#, INTB#, INTC#, and INTD#. PCI legacy interrupts are level-triggered; hence, they may be shared by multiple functions. Each function within the device is only permitted to use one of these interrupt pins to generate requests. If a device contains only one function that uses only one interrupt pin, it must be called INTA#. If a device contains more than one function, all functions within a device may be bonded to the same pin, INTA#, or each may be bonded to a dedicated pin (this would be true for a device with up to four functions embodied within it). Also, a group of functions within the package may share the same interrupt pin. In the most simple (and most common) case, a PCI device has only one function with its interrupt going to the lane INTA#.

Figure 1.56 illustrates a simple interrupt model with two peripheral devices on the PCI bus; hence, both are required to INTA# . Each peripheral device embodies only one function that generates PCI interrupts. The first device is an ethernet card in the PCI slot 0 that generates interrupts on INT#A line, while the second device is a sound card in the PCI slot 3 that also generates interrupts on the INT#A line. Both devices share the same interrupt request signal trace on the system board, which is routed to the IRQ5 input on the Intel 8259A programmable interrupt controller (PIC). Indeed, the PCI standard does not limit the interrupt controller used to route

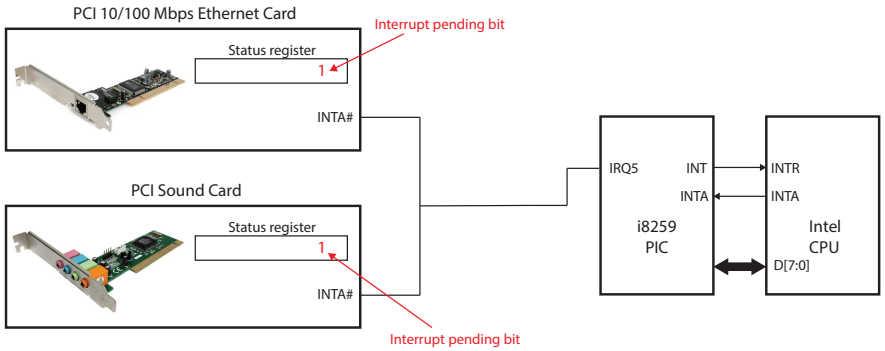

Fig. 1.56: Example system for shared interrupts on the PCI bus.

PCI interrupts to the CPU as long it supports level-triggered interrupts; hence, in this example, we assume that the 8259A PIC is used for this purpose.

Let us assume that both devices assert the interrupt request and that the sound card asserted the interrupt first. The interrupts are asserted by driving the interrupt line LOW. In addition to asserting the interrupt request, both devices set an interrupt pending bit in their memory-mapped status registers so that interrupt handlers can access both status registers. Let us also assume that no other device in the system had asserted an interrupt request before the sound card and ethernet card.

When the Intel 8259A PIC detects the interrupt request on its IRQ5 input, it asserts the interrupt request on the processor's INTR input. The processor will recognize this request on the next instruction fetch. It then stops the instruction fetch and automatically saves the program context onto the stack. The CPU then starts the interrupt acknowledge sequence: the CPU responds by asserting the first INTA pulse, the 8259 PIC prioritizes the pending interrupt requests by setting the bit 5 in its ISR register and clearing the bit 5 in its IRR register, the CPU outputs the second INTA pulse to instruct the 8259A PIC to release the 8-bit pointer onto the data bus (D0 to D7) where it is read by the CPU. Now the processor has received the interrupt vector number associated with IRQ5. Let assume that the interrupt vector number is `0x07`. The processor multiplies this value by 4, which yields the address of the interrupt vector entry, `0x0000001C`. The processor now reads the content of the memory location `0x0000001C` to obtain the start address of the interrupt handler.

As both devices, the ethernet card and the sound card, share the same interrupt line IRQ5, the interrupt handler should contain the code to handle the interrupt requests from both devices. Let us assume that the interrupt handler contains both codes, the "ethernet" handler, and the "sound card" handler. The simplified structure of the IRQ5 interrupt handler is presented in Listing 1.57. The interrupt handler first checks which device has asserted the interrupt request by checking the interrupt pending bit in the corresponding status register.

```
/*
* IRQ5 Handler
*/
__attribute__((interrupt)) void irq5handler () {

    /* Check the interrupt pending bit in the Ethernet status reg */
    if (eth_status_reg & (1<<INT_PEND_BIT)) {
    /* Ethernet Handler Code */
    eth_status_reg &= ~(1<<INT_PEND_BIT); // clear int pending bit
    ...
    ...
    ...
    }

    /* Check the interrupt pending bit in the Sound card status reg */
    if (snd_status_reg & (1<<INT_PEND_BIT)) {
    /* Sound Card Handler Code */
    snd_status_reg &= ~(1<<INT_PEND_BIT); // clear int pending bit
    ...
    ...
    ...
    }

    /* return from interrupt */
}
```

Listing 1.57: IRQ5 interrupt handler

Listing 1.57 shows that the interrupt handler checks the pending bit in the ethernet card's status register first. As this bit is asserted, the ethernet handler is executed first. The ethernet handler clears the pending bit in the status register and processes the interrupt request. Then the IRQ5 handler proceeds with the sound card handler. This handler checks the sound card's interrupt pending bit to determine if it requires servicing. Since the pending bit is set, the main body of the sound card handler is executed. It clears the interrupt pending bit and services the interrupt request. As both devices have their pending bit clear, the interrupt line is de-asserted.

Hence, the system in Figure 1.56 relies on the vectored interrupt handling to determine which interrupt request input in the PIC 8259A has been asserted, but use the interrupt polling to determine which PCI device has asserted the interrupt request. The sequence of polling determines the interrupt priority. This is why the ethernet card is serviced first, although the sound card had asserted the interrupt request a few moments before the ethernet card.

1.8.2 PCI interrupts routing

Each PCI device (function) that needs an interrupt comes with a fixed PCI interrupt that can't be changed. But this PCI legacy interrupts signal (lane) can be mapped (routed or redirected) to any APIC interrupt input. Thus, at one end, we have a PCI legacy interrupt lines (INTA# through INTD#) being signaled by a PCI function that needs attention. At the other end, we have a CPU receiving an IDT vector. In the middle is an interrupt controller (most commonly, this would be an APIC pair: I/O APIC and LAPIC). Whenever any of the PCI legacy interrupt pins is asserted, the

I/O APIC module supplies the vector associated with that input to the processor's embedded local APIC module. We have already learned that IR16-IR23 input I/O APIC pins are devoted to PCI. The upper four (IR20-IR23) are devoted to PCI functions embedded in the chipset, while the lower four are devoted to PCI legacy interrupt pins. The I/O APIC inputs IR20 through IR23 are called PIRQA through PIRQH when used for PCI interrupts. The acronym PIRQ stands for PCI interrupt request. We have also learned that several PCI devices or functions can use the same PCI legacy interrupt signal to assert interrupts. So, the question is, how the PCI legacy interrupt signals INTA# through INTD# are routed to the I/O APIC inputs PIRQA through PIRQD? The best scenario is pictured in Figure 1.57 where each of the individual PCI interrupt lines is routed to an interrupt controller as a separate input. But such a solution is possible only if there are only up to four PCI devices because the I/O APIC has only four available interrupt inputs.

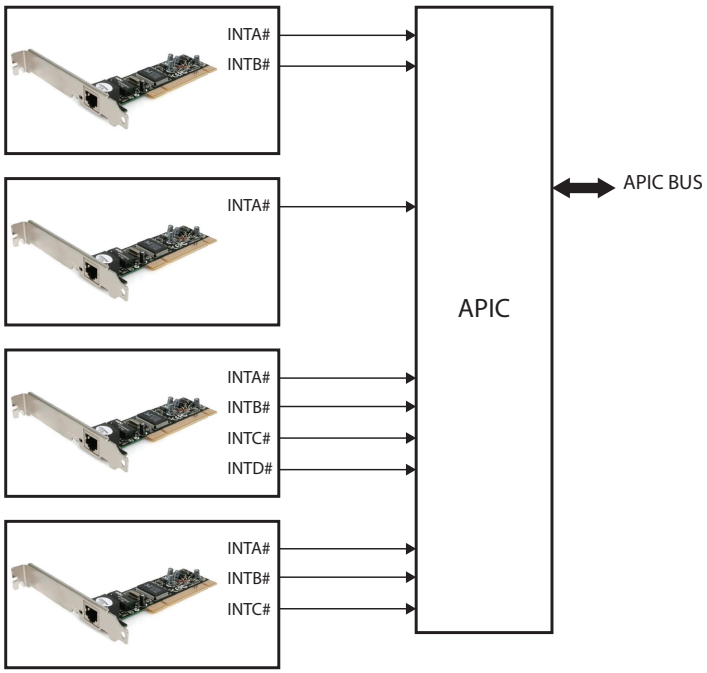

Fig. 1.57: Ideal routing of PCI interrupt lanes.

PCI interrupt signals can be routed to I/O APIC interrupt pins (PIRQs) in several different ways. One simple method of connecting (hardwiring) these lines from PCI devices to the PIRQs would be to connect all INTA# interrupts to PIRQA, all INTB# interrupts to PIRQB, all INTC# interrupts to PIRQC, and all INTD# interrupts to PIRQD. Figure 1.58 illustrates this method of PCI interrupts routing. As we've already said above, the most common case is when a PCI device has only

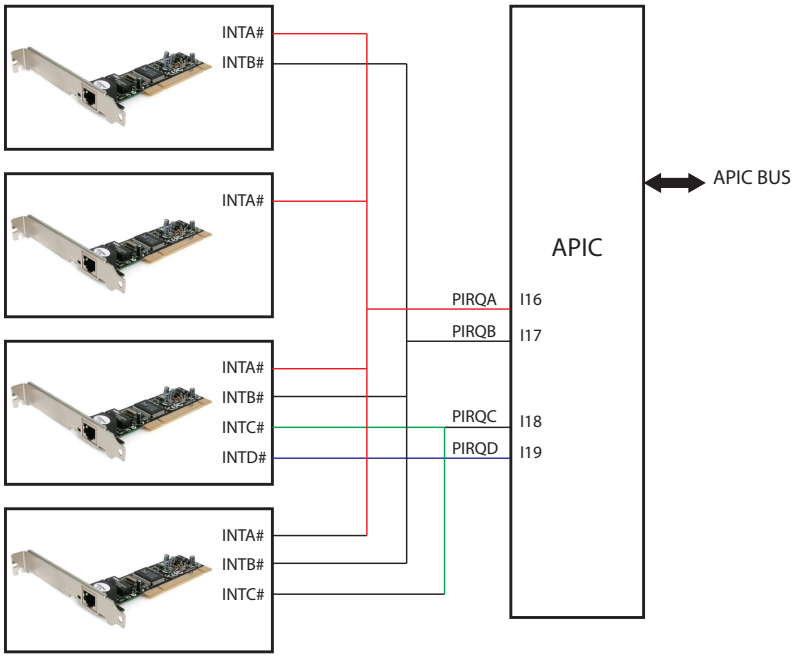

Fig. 1.58: Unbalanced routing of PCI interrupt lanes.

one function, and its interrupt must be connected to the INTA# pin. Therefore if we decided to route all PCI interrupt lanes as we've written, almost all of the devices in a system would share interrupt input PIRQA. As you can see in Figure 1.58 the PIRQA request line is heavily-weighted (four PCI devices). Suppose this lane is connected to the IRQ16 of the APIC. This way, every time a processor has a signal that there is an interrupt on the IRQ16 input, it has to poll all of the device drivers of the PCI devices connected to that IRQ16 line (PIRQA) if they have asserted an interrupt. If there are many of those devices, it will surely decrease system response to the interrupt. And in this case, lanes PIRQB-PIRQD would stand idle most of the time.

The optimal way of PCI interrupts routing should take into account that each PIRQ should have fairly the same number of PCI functions connected to it. We should also consider that some functions trigger interrupts very rarely and some almost constantly (e.g., Ethernet controller). Hence, we may connect the PIRQs in a more random way so that each of them will share about the same number of actual PCI legacy interrupts.

One method of doing this is illustrated in Figure 1.59. This illustration shows how legacy interrupt traces are physically routed across the PCI slots. Although the physical interrupt lines wire each PIRQ to every slot, each PIRQ connects differently to the pins in each slot. Here, wire PIRQA share interrupts INTA# in the PCI slot

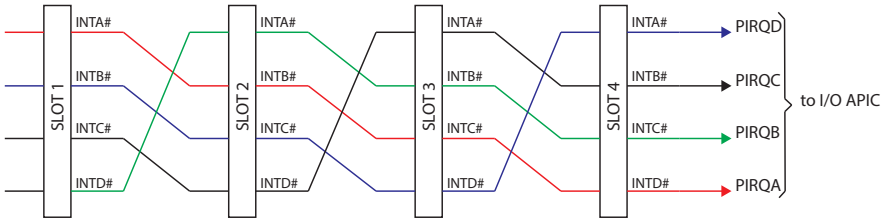

Fig. 1.59: Round-robin routing of PCI legacy interrupt traces (lanes).

1, INTB# in the PCI slot 2, INTC# in the PCI slot 3, and INTD# in the PCI slot 4. Likewise, wire PIRQB share interrupts INTB# in the PCI slot 1, INTC# in the PCI slot 2, INTD# in the PCI slot 3, and INTA# in the PCI slot 4, etc.

Figure 1.60 shows a 4-slot PCI system with the following cards installed:

1. Card 1 installed in Slot 1. This card includes two PCI functions of which one generates interrupts on IRQA#, and the other generates interrupts on IRQD#. The IRQA# pin of this card connects to PIRQA.
2. Card 2 installed in Slot 2. This card includes four PCI functions, thus generating interrupts on IRQA# through IRQD#. The IRQA# pin of this card connects to PIRQB.
3. Card 3 installed in Slot 3. This card includes three PCI functions, which generate interrupts on IRQA# through IRQC#. The IRQA# pin of this card connects to PIRQC.
4. Card 4 installed in Slot 4This card includes four PCI functions, thus generating interrupts on IRQA# through IRQD#. The IRQA# pin of this card connects to PIRQD.

A practical use for this is that one may change the interrupt routing of a PCI card by inserting it in a different slot. In the above example, INTA# of a PCI card will be connected to wire PIRQA if the card is inserted into slot 1, but INTA# will be connected to wire PIRQB when inserted into slot 4.

1.8.3 Message Signaled Interrupts

As we described in the previous section, a PCI device can use up to four dedicated interrupt pins to signal an interrupt request to the I/O APIC. This method is referred to as Legacy INTx interrupts. Each PCI device can have up to four PCI legacy interrupts. After the I/O APIC has received an interrupt request, it forwards it to LAPICs by the mean of the APIC messages. But why not implement this "interrupt message" functionality into a PCI device itself? This way, we could eliminate the need for interrupt traces and interrupt sharing. This method is referred to as **Message Signaled Interrupts (MSI)** and is described in this section. Message Signaled In-

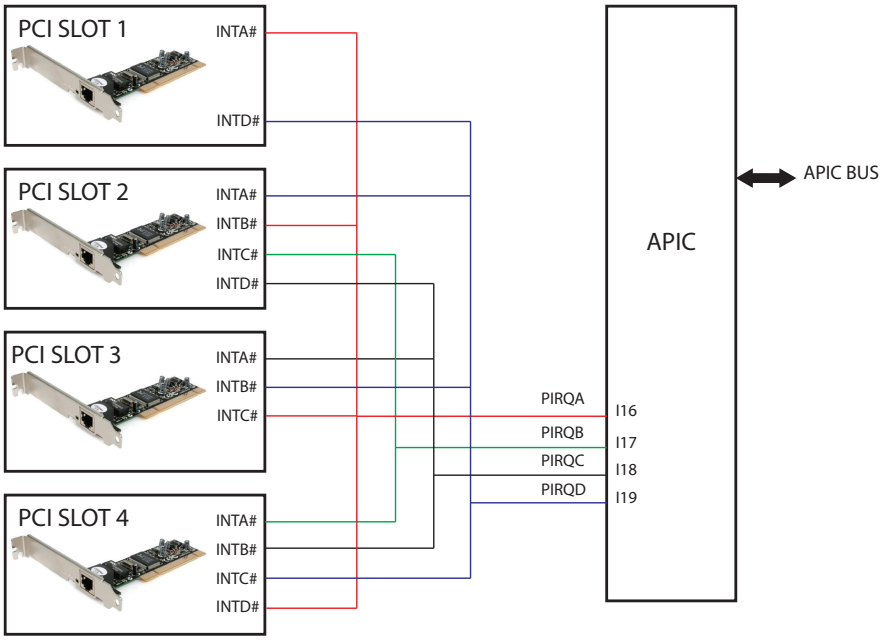

Fig. 1.60: Common round-robin routing of PCI interrupt lanes.

terrupts are special memory writes that are sent over the system bus. In essence, the MSI interrupts do not differ from ordinary PCI memory write transactions. But they are recognized by LAPICs from the address they write to. The data sent in these transactions contain an interrupt vector.

Using a separate signal for PCI INTx interrupts raises several issues. First, on many x86 systems, this requires separate physical traces on the motherboard to connect the signals to interrupt controller input pins. Second, the interrupt signals should be routed in a clever way, so that interrupt requests are evenly distributed across the PIRQ lines. But the largest issue can arise when a device writes data to memory and raises a pin-based interrupt to signal the CPU that the data has been written. It is possible that the interrupt arrives before all the data has arrived in memory. In order to ensure that all the data has arrived in memory, the interrupt handler must read a special register on the PCI device, which raised the interrupt. This read will not be completed until any pending transactions in-between the CPU and the PCI device complete. PCI transaction ordering rules require that all the data arrive in memory before the value may be returned from this special register. Thus, this dummy read guarantees that all the effects of the event that triggered the interrupt will be visible to the CPU. But this dummy read adds extra latency and work to the interrupt handler. Using message signaled interrupts avoids this problem as the interrupt message is a memory write transaction. Hence, it cannot pass the data writes,

so when the interrupt is raised, the interrupt handler knows that all the data has been successfully written into memory.

To summarize, the advantages of MSI interrupts versus the legacy interrupts are:

- the MSI interrupts eliminate the need for interrupt traces between PCI devices and the I/O APIC,
- the MSI interrupts eliminate multiple PCI functions sharing the same PIRQ,
- the MSI interrupts eliminate the need for device polling in the interrupt handlers,
- the MSI interrupts eliminate the need to perform a read from a device's register o force all posted memory writes to be flushed to memory.

When a PCI function supports MSI, it generates an interrupt request to the processor's LAPIC by writing a predefined data item to a predefined memory address in the LAPIC. This PCI write transaction that contains a predefined data and a predefined address is referred to as an interrupt message. MSI was introduced in revision 2.2 of the PCI spec in 1999 as an optional component. However, with the introduction of the PCIe specification in 2004, implementation of MSI became mandatory from a hardware standpoint. It is worthwhile to notice that MSI interrupts can't work without LAPIC, but MSI eliminates the devices' need to use the IO-APIC, allowing every device to write directly to the CPU's LAPIC.

Each PCI function that generates MSI must contain two addressable registers (Message Data register and Message Address register) where BIOS or OS stores this predefined data and addresses during the initialization. When a PCI function asserts an interrupt using MSI, it performs a PCI write operation that writes the content of the Message Data register to the address specified in the Message Address register.

The following is the sequence for MSI delivery and servicing:

1. A device needing servicing from the CPU generates an MSI, writing the interrupt vector number directly into the Local-APIC of the CPU servicing it.
2. The interrupted CPU begins running the handler associated with the interrupt vector number it received. The device is serviced without any need to check and clear an IRQ pending bit

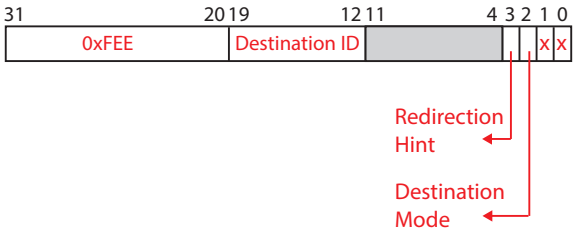

Fig. 1.61: Layout of the MSI Memory Address register.

The format of the Message Address register is presented in Figure 1.61. Fields in the Message Address Register are as follows:

- Bits 31-20 contain a fixed value for interrupt messages (0xFEE). This value locates interrupts at the 1-MByte area with a base address of 0xFEE00000. All accesses to this region are directed as interrupt messages.
- Destination ID — This field contains an 8-bit destination ID. It identifies the target LAPIC.
- Redirection hint (RH) — When this bit is set, the message is directed to the processor with the lowest interrupt priority among processors that can receive the interrupt. When this bit is reset, the interrupt is directed to the processor listed in the Destination ID field.
- Destination mode (DM) — This bit indicates whether the Destination ID field should be interpreted as logical or physical LAPIC for delivery of the lowest priority interrupt. If RH is 0, then the DM bit is ignored. If RH is 1 and DM is 0, only the processor listed in the Destination ID field is considered for delivery of that interrupt (this means no redirection). If RH is 1 and DM is 1, the redirection is limited to only those processors that are part of the logical group of processors based on the Destination ID field in the message.

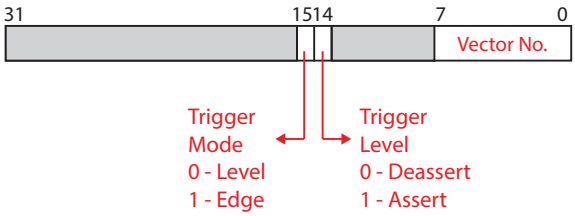

Fig. 1.62: Layout of the MSI Memory Data register.

Figure 1.62 illustrates the format of the Message Data register. The fields in the Message Data Register are:

- Vector number. This 8-bit field contains the interrupt vector number associated with the message. Values range from 0x10 to 0xFE. The software must guarantee that the field is not programmed with vector 0x00 to 0x0F as this are reserved for non-external interrupts and exceptions.
- Trigger Mode. If this bit is 0, the interrupt is edge-triggered. If this bit is 1, the interrupt is level-triggered.
- Level. For edge-triggered interrupts this field is ignored. For level-triggered interrupts, this bit reflects the active state of the interrupt input.