Introduction

Interrupt and exceptions

Interrupts can originate both by the hardware and the software itself. ARM architecture distinguishes between the two types: interrupts originate by the hardware, exceptions by the software (e.g., an access to invalid memory location). In ARM terminology, an interrupt is a type of exception

NVIC

Cortex-M processors provide a unit dedicated to exceptions management. This is called Nested Vectored Interrupt Controller (NVIC)

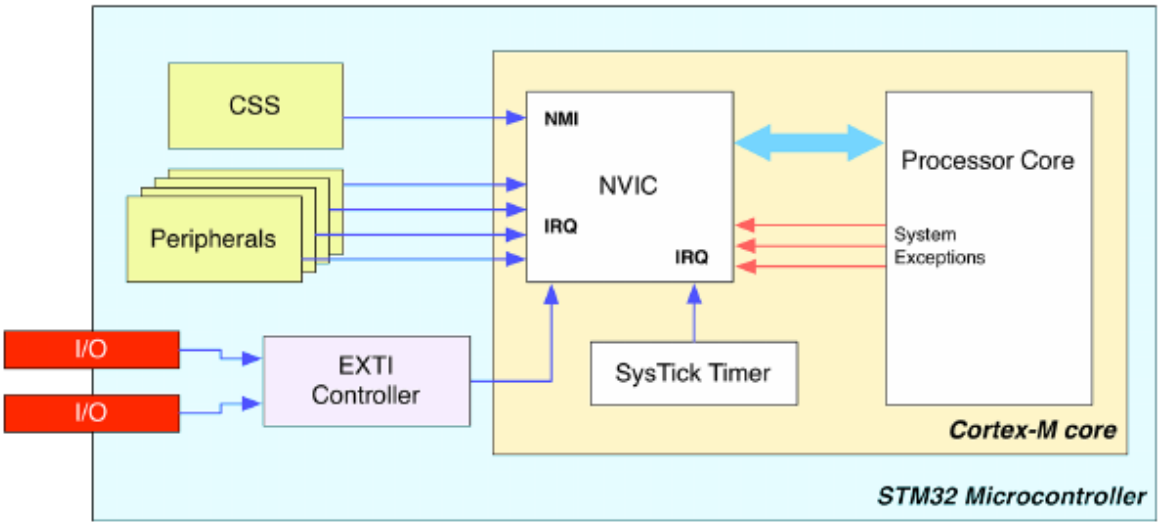

Figure 1: NVIC

Two source of interrupt

- internal: timers, adc, serial buses
- external

The source of the interrupts coming from the last kind of peripherals are the MCU I/O, which can be both configured as general purpose I/O (e.g. a tactile switch connected to a pin configured as input) or to drive an external advanced peripheral.

As stated before, ARM distinguishes between system exceptions, which originate inside the CPU core, and hardware exceptions coming from external peripherals, also called Interrupt Requests (IRQ).

Vector Table in STM32

- Reset: this exception is raised just after the CPU resets. Its handler is the real entry point of the running firmware. In an STM32 application all starts from this exception. The handler contains some assembly-coded functions designed to initialize the execution environment, such as the main stack, the .bss area, etc.

- NMI: this is a special exception, which has the highest priority after the Reset one. Like the Reset exception, it cannot be masked (that is disabled), and it can be associated to critical and non-deferrable activities. In STM32 microcontrollers it is linked to the Clock Security System (CSS). CSS is a self-diagnostic peripheral that detects the failure of the HSE. If this happens, HSE is automatically disabled (this means that the internal HSI is automatically enabled) and a NMI interrupt is raised to inform the software that something is wrong with the HSE. More about this feature in Chapter 10.
- Hard Fault: is the generic fault exception, and hence related to software interrupts. When the other fault exceptions are disabled, it acts as a collector for all types of exceptions (e.g., a memory access to an invalid location raised the Hard Fault exceptions if the Bus Fault one is not enabled).
- Memory Management Fault1 : it occurs when executing code attempts to access an illegal location or violates a rule of the Memory Protection Unit (MPU).
- Bus Fault1: it occurs when AHB interface receives an error response from a bus slave (also called prefetch abort if it is an instruction fetch, or data abort if it is a data access). Can also be caused by other illegal accesses (e.g. an access to a non existent SRAM memory location).
- Usage Fault1 : it occurs when there is a program error such as an illegal instruction, alignment problem, or attempt to access a non-existent co-processor.
- SVCCall: this is not a fault condition, and it is raised when the Supervisor Call (SVC) instructions is called. This is used by Real Time Operating Systems to execute instructions in privileged state (a task needing to execute privileged operations executes the SVC instruction, and the OS performs the requested operations - this is the same behavior of a system call in other OS).
- Debug Monitor1: this exception is raised when a software debug event occurs while the processor core is in Monitor Debug-Mode. It is also used as exception for debug events like breakpoints and watchpoints when software based debug solution is used.
- PendSV: this is another exception related to RTOS. Unlike the SVCall exception, which is executed immediately after a SVC instruction is executed, the PendSV can be delayed. This allows the RTOS to complete tasks with higher priorities.
- SysTick: this exception is also usually related to RTOS activities. Every RTOS needs a timer to periodically interrupt the execution of current code and to switch to another task. All STM32 microcontrollers provide a SysTick timer, internal to the Cortex-M core. Even if every other timer may be used to schedule system activities, the presence of a dedicated timer ensures portability among all STM32 families (due to optimization reasons related to the internal die of the MCU, not all timers could be available as external peripheral). Moreover, even if we aren't using an RTOS in our firmware, it is important to keep in mind that the ST CubeHAL uses the SysTick timer to perform internal time-related activities (and it also assumes that the SysTick timer is configured to generate an interrupt every 1ms).

The remaining exceptions that can be defined for a given MCU are related to IRQ handling. Cortex-M0/0+ cores allows up to 32 external interrupts (See Reference Manual for STM32 L4+).

Even if the vector table contains the addresses of the handler routines, the Cortex-M core needs a way to find the vector table inside memory. By convention, the vector table starts at the hardware address 0x0000 0000 in all Cortex-M based processors. If the vector table resides in the internal flash memory (this is what usually happens), and since the flash in all STM32 MCUs is mapped from 0x0800 0000 address, it is placed starting from the 0x0800 0000 address, which is aliased to 0x0000 0000 when the CPU boots up

Enabling interrupt

When an STM32 MCU boots up, only Reset, NMI and Hard Fault exceptions are enabled by default. The rest of exceptions and peripheral interrupts are disabled, and they have to be enabled on request. To enable an IRQ, the CubeHAL provides the following function:

```
void HAL_NVIC_EnableIRQ(IRQn_Type IRQn);
void HAL_NVIC_DisableIRQ(IRQn_Type IRQn);
```

External Lines and NVIC

GPIO are connected to the EXTI lines, and it is possible to enable interrupts for every MCU GPIO, even if the most of them share the same interrupt line. For example, for an STM32F4 MCU, up to 114 GPIOs are connected to 16 EXTI lines. However, only 7 of these lines have an independent interrupt associated with them.

- Only one PxY pin can be a source of interrupt. For example, we cannot define both PA0 and PB0 as input interrupt pins;
- For EXTI lines sharing the same IRQ inside the NVIC controller, we have to code the corresponding ISR so that we must be able to discriminate which lines generated the interrupt;

What to do to configure a gpio to rise interrupts

1. Configure the desired GPIO to fire an interrupt every time it goes from the low level to the high one. This is accomplished setting GPIO .Mode to be equal to `GPIO_MODE_IT_RISING`
2. Enable the interrupt, calling, for example, the function

```
HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
```

slide ST regarding the interrupt on STM32 microcontrollers

How to write an ISR

Different ways:

1. Defining a function `*_IRQHandler(void)`, for example `EXTI15_10_IRQHandler(void)` if you want to handle an interrupt on the lines 10-15 of the EXTI.

```
void EXTI15_10_IRQHandler(void) {
    // clear the interrupt flag
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_13);
    // toggle the gpio pin A5
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
}
```

2. Using the callback system
3. Enabling Interrupts With CubeMX

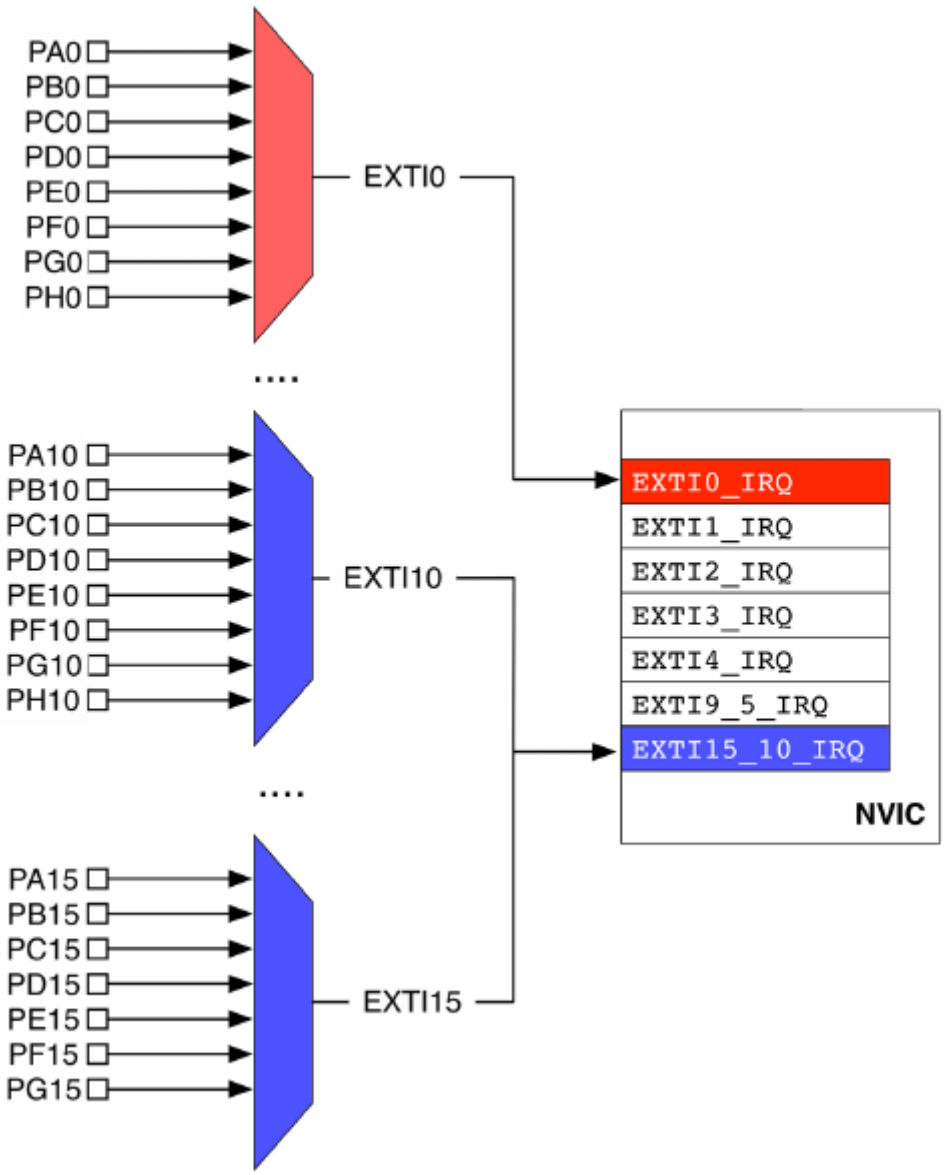

Figure 2: EXTI

Interrupt lifecycle

An interrupt can:

1. either be disabled (default behavior) or enabled;
    - we enable/disable it calling the `HAL_NVIC_EnableIRQ()`/`HAL_NVIC_DisableIRQ()` func- tion;
2. either be pending (a request is waiting to be served) or not pending;
3. either be in an active (being served) or inactive state.

To see if an interrupt is pending (that is, fired but not running), we can use the HAL function:

```
uint32_t HAL_NVIC_GetPendingIRQ(IRQn_Type IRQn);
```

To programmatically set the pending bit of an IRQ we can use the HAL function:

```
void HAL_NVIC_SetPendingIRQ(IRQn_Type IRQn);
```

- This will cause the interrupt to fire, as it would be generated by the hardware.
- It is possible to programmatically fire an interrupt inside the ISR routine of another interrupt

To clear pending interrupts

```
void HAL_NVIC_ClearPendingIRQ(IRQn_Type IRQn);
```

To check if an ISR is active (IRQ being serviced), we can use the function:

```
uint32_t HAL_NVIC_GetActive(IRQn_Type IRQn);
```

which returns 1 if the IRQ is active, 0 otherwise.

Interrupt priority levels

The CubeHAL provides the following function to assign a priority to an IRQ:

```
void HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t SubPriority);
```