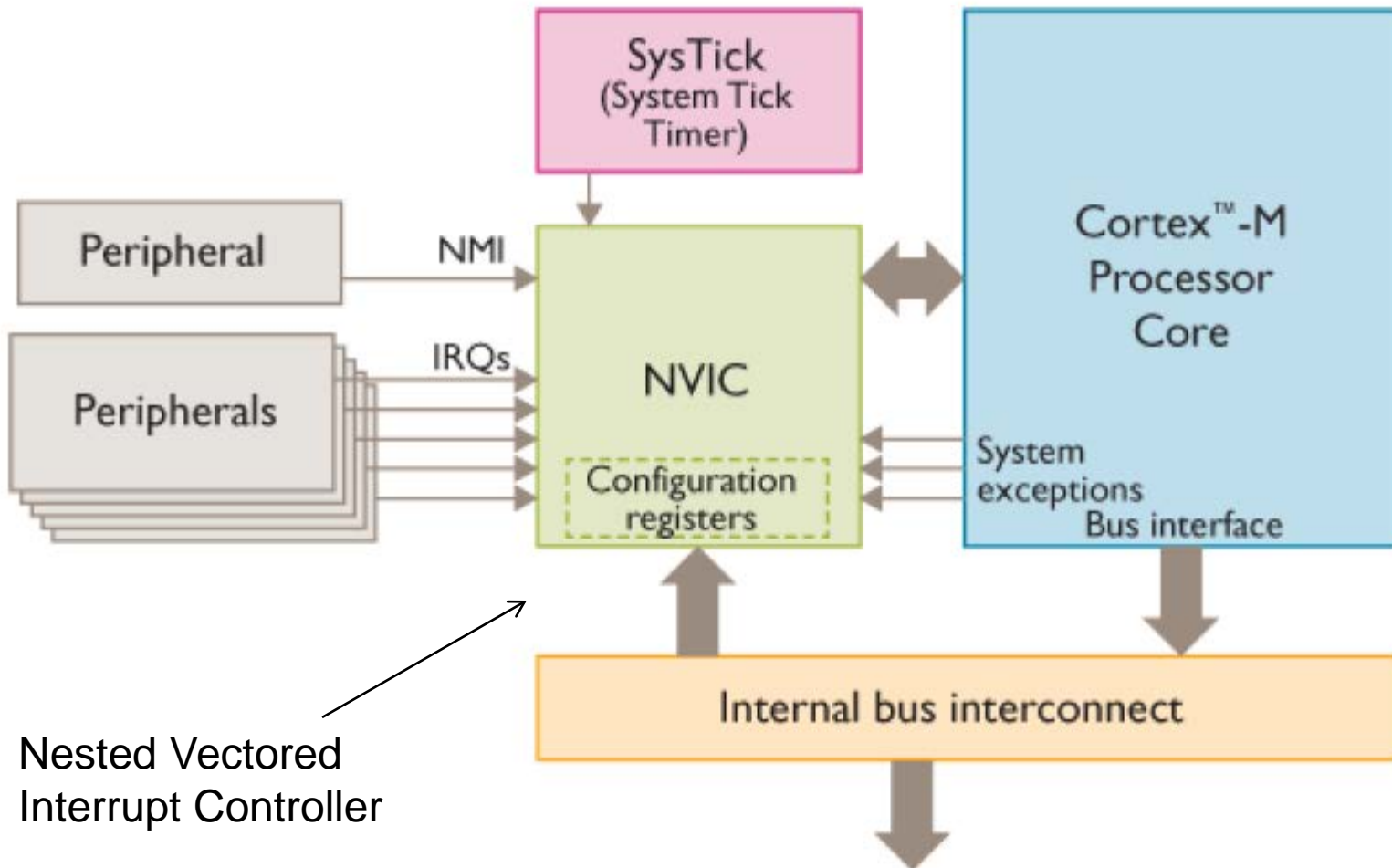


# ARM and STM32F4xx Operating Modes & Interrupt Handling

ARM Cortex-M4 User Guide (Interrupts, exceptions, NVIC)  
STM32F4xx Microcontrollers Technical Reference Manual

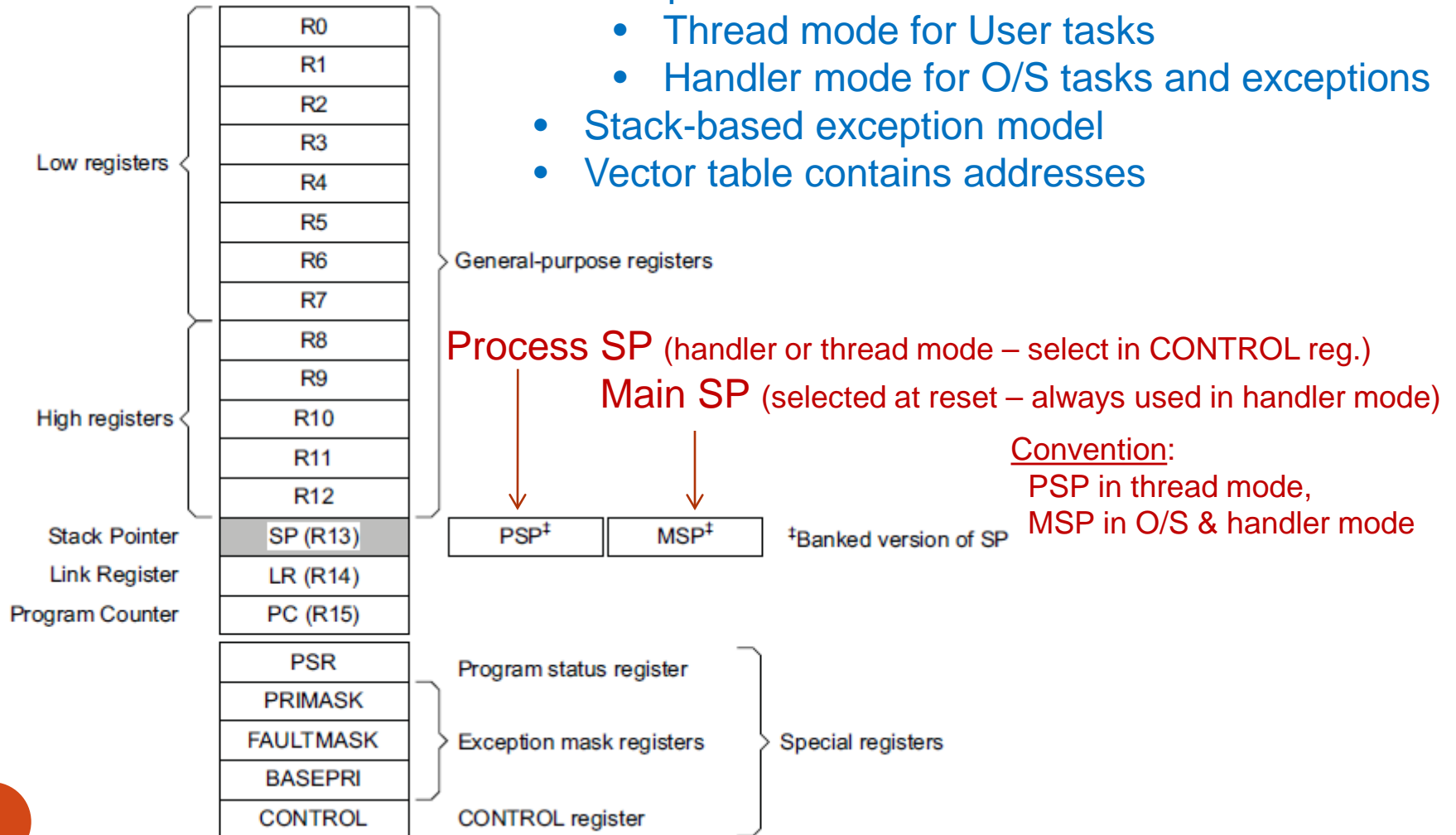
# Cortex-M structure



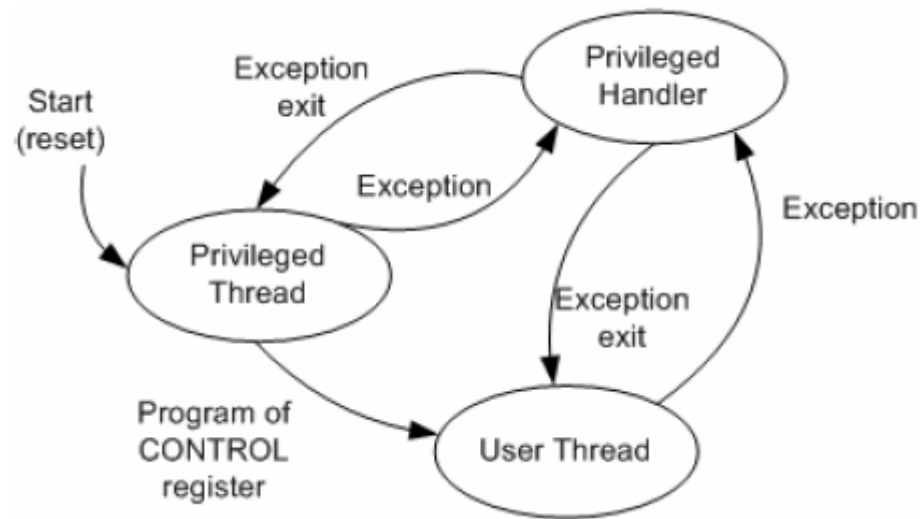
Nested Vectored  
Interrupt Controller

# Cortex CPU core registers

- Two processor modes:
  - Thread mode for User tasks
  - Handler mode for O/S tasks and exceptions
- Stack-based exception model
- Vector table contains addresses



# Cortex-M4 processor operating modes



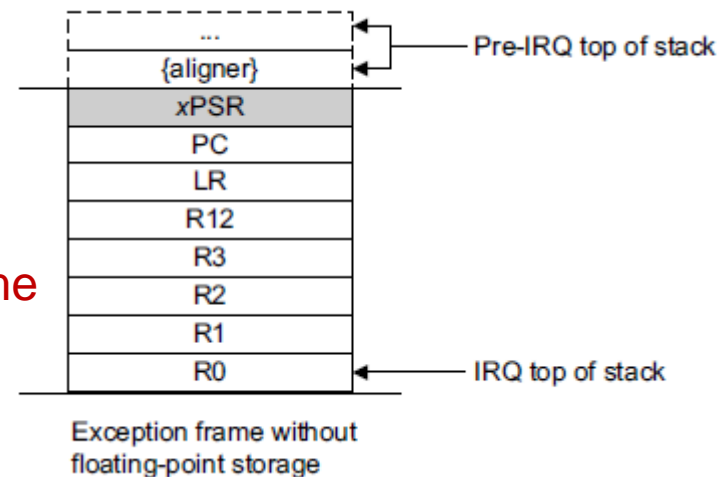
- **Thread** mode – normal processing
- **Handler** mode – interrupt/exception processing
- Privilege levels = **User** and **Privileged**
  - Supports basic “security” & memory access protection
  - Supervisor/operating system usually privileged

# Cortex-M4 interrupts/exceptions

- Interrupts/exceptions managed by *Nested Vectored Interrupt Controller (NVIC)*
- CPU state/context (subset of registers) *saved on the stack*

R0-R3, R12, LR, PC, PSR

Exception  
stack frame



- PC loaded from a vector table, located at 0x0000\_0000
  - Vector fetched (Flash memory) **while** saving state (SRAM)
  - Typical latency = 12 cycles

# Exception states

- Each exception is in one of the following states:
  - **Inactive:** The exception is not active and not pending.
  - **Pending:** The exception is waiting to be serviced by the processor.
  - **Active:** The exception is being serviced by the processor but has not completed.
  - **Active and pending** - The exception is being serviced by the processor and there is a pending exception from the same source.
- An interrupt request from a peripheral or from software can change the state of the corresponding interrupt to pending.
- An exception handler can interrupt (**preempt**) the execution of another exception handler. In this case both exceptions are in the active state.

# Cortex-M CPU and peripheral exceptions

	Priority <sup>1</sup>	IRQ# <sup>2</sup>	Notes
Reset	-3		Power-up or warm reset
NMI	-2	-14	Non-maskable interrupt from peripheral or software
HardFault	-1	-13	Error during exception processing or no other handler
MemManage	Config	-12	Memory protection fault (MPU-detected)
BusFault	Config	-11	AHB data/prefetch aborts
UsageFault	Config	-10	Instruction execution fault - undefined instruction, illegal unaligned access
SVCcall	Config	-5	System service call (SVC) instruction
DebugMonitor	Config		Break points/watch points/etc.
PendSV	Config	-2	Interrupt-driven request for system service
SysTick	Config	-1	System tick timer reaches 0
IRQ0	Config	0	Signaled by peripheral or by software request
IRQ1 (etc.)	Config	1	Signaled by peripheral or by software request

CPU Exceptions

Vendor peripheral interrupts  
IRQ0 .. IRQ44

<sup>1</sup> Lowest priority # = highest priority  
<sup>2</sup> IRQ# used in CMSIS function calls

# Vector table

- 32-bit vector(handler address) loaded into PC, while saving CPU context.
- Reset vector includes initial stack pointer
- Peripherals use positive IRQ #s
- CPU exceptions use negative IRQ #s
- IRQ # used in CMSIS function calls
- Cortex-M4 allows up to 240 IRQs
- IRQ priorities user-programmable
- NMI & HardFault priorities fixed

Exception number	IRQ number	Offset	Vector
16+n	n	0x0040+4n	IRQn
.	.	.	.
.	.	.	.
.	.	.	.
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5	0x002C	SVCcall
10			Reserved
9			Reserved
8			Reserved
7			Reserved
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value



STM32F4  
Vector  
Table  
(partial)

Tech. Ref.  
Table 61

(Refer to  
Startup  
Code)

Position	Priority	Type of priority	Acronym	Description	Address
	-	-	-	Reserved	0x0000 0000
	-3	fixed	Reset	Reset	0x0000 0004
	6	settable	SysTick	System tick timer	0x0000 003C
0	7	settable	WWDG	Window Watchdog interrupt	0x0000 0040
1	8	settable	PVD	PVD through EXTI line detection interrupt	0x0000 0044
2	9	settable	TAMP_STAMP	Tamper and TimeStamp interrupts through the EXTI line	0x0000 0048
3	10	settable	RTC_WKUP	RTC Wakeup interrupt through the EXTI line	0x0000 004C
4	11	settable	FLASH	Flash global interrupt	0x0000 0050
5	12	settable	RCC	RCC global interrupt	0x0000 0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000 0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000 005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000 0060
9	16	settable	EXTI3	EXTI Line3 interrupt	0x0000 0064
10	17	settable	EXTI4	EXTI Line4 interrupt	0x0000 0068
11	18	settable	DMA1_Stream0	DMA1 Stream0 global interrupt	0x0000 006C
12	19	settable	DMA1_Stream1	DMA1 Stream1 global interrupt	0x0000 0070
13	20	settable	DMA1_Stream2	DMA1 Stream2 global interrupt	0x0000 0074

# STM32F4 vector table from startup code (partial)

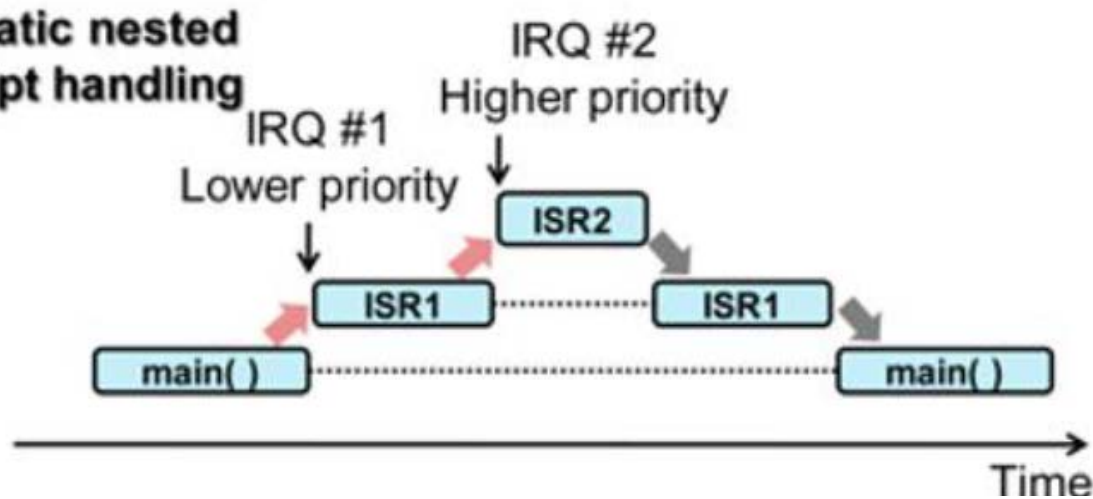
```
__Vectors      DCD    __initial_sp      ; Top of Stack
               DCD    Reset_Handler    ; Reset Handler
               DCD    NMI_Handler      ; NMI Handler
               .....
               DCD    SVC_Handler      ; SVCcall Handler
               DCD    DebugMon_Handler ; Debug Monitor Handler
               DCD    0                 ; Reserved
               DCD    PendSV_Handler   ; PendSV Handler
               DCD    SysTick_Handler  ; SysTick Handler

; External Interrupts
DCD    WWDG_IRQHandler    ; Window WatchDog
DCD    PVD_IRQHandler    ; PVD via EXTI Line detection
DCD    TAMP_STAMP_IRQHandler ; Tamper/TimeStamps via EXTI
DCD    RTC_WKUP_IRQHandler ; RTC Wakeup via EXTI line
DCD    FLASH_IRQHandler  ; FLASH
DCD    RCC_IRQHandler    ; RCC
DCD    EXTI0_IRQHandler  ; EXTI Line0
DCD    EXTI1_IRQHandler  ; EXTI Line1
DCD    EXTI2_IRQHandler  ; EXTI Line2
```

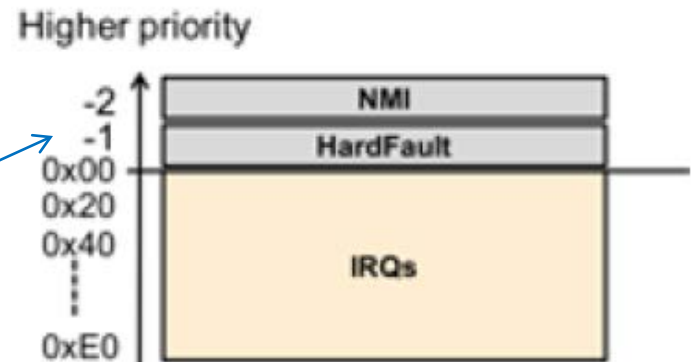


# Prioritized interrupts

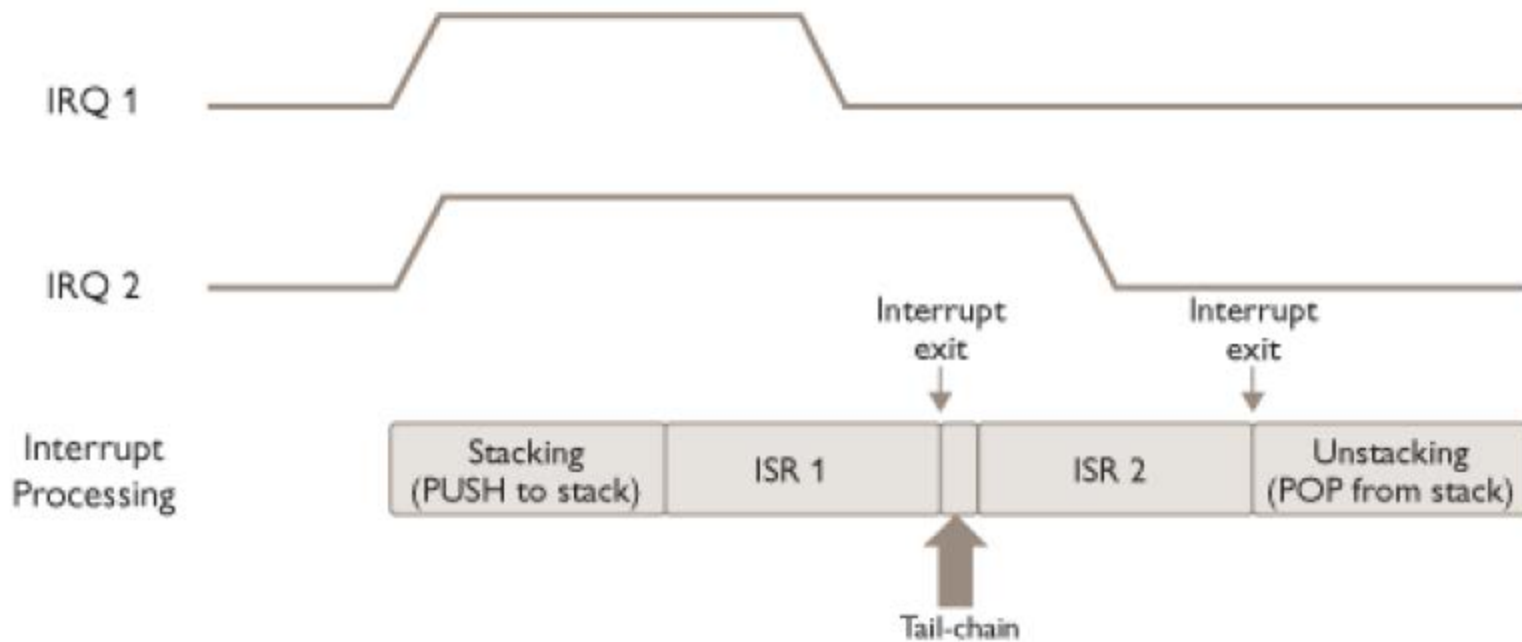
## Automatic nested interrupt handling



- Up to 256 priority levels
  - 8-bit priority value
  - Implementations may use fewer bits
- STM32F4xx uses upper 4 bits of each priority byte => 16 levels
- STM32F4xx uses 4 bits => 16 levels
- NMI & HardFault priorities are fixed



# “Tail-chaining” interrupts



- NVIC does not unstack registers and then stack them again, if going directly to another ISR.
- NVIC can halt stacking (and remember its place) if a new IRQ is received.

# Exception return

- The exception mechanism detects when the processor has completed an exception handler.
- Exception return occurs when:
  1. Processor is in Handler mode
  2. EXC\_RETURN loaded to PC
  3. Processor executes one of these instructions:
    - LDM or POP that loads the PC
    - LDR with PC as the destination
    - BX using any register
- EXC\_RETURN value loaded into LR on exception entry (after stacking original LR)
  - Lowest 5 bits of EXC\_RETURN provide information on the return stack and processor mode.

# Interrupt signal: from device to CPU

## In each peripheral device:

- Each potential interrupt source has a separate **arm (enable)** bit
  - Set for devices from which interrupts, are to be accepted
  - Clear to prevent the peripheral from interrupting the CPU
- Each potential interrupt source has a separate **flag** bit
  - hardware sets the flag when an “event” occurs
  - Interrupt request = (flag & enable)
  - ISR software must clear the flag to acknowledge the request
  - **test flags in software if interrupts not desired**

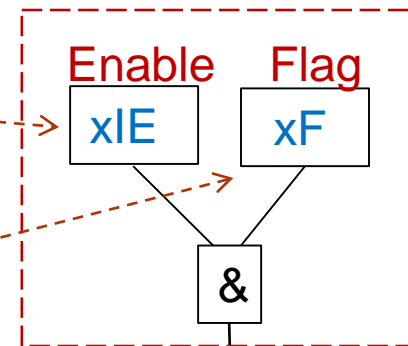
## Nested Vectored Interrupt Controller (NVIC)

- Receives all interrupt requests
- Each has an enable bit and a priority within the VIC
- Highest priority enabled interrupt sent to the CPU

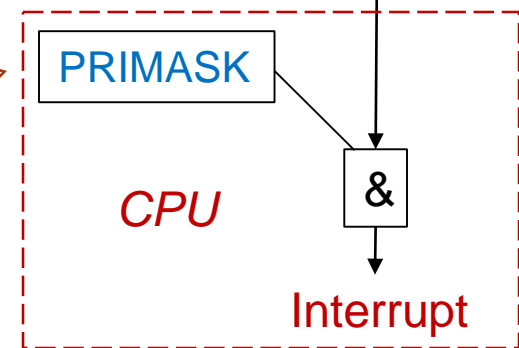
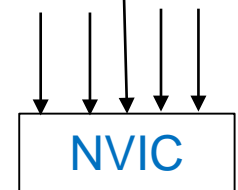
## Within the CPU:

- Global interrupt enable bit in PRIMASK register
- Interrupt if priority of IRQ < that of current thread
- Access interrupt vector table with IRQ#

*Peripheral Device Registers:*

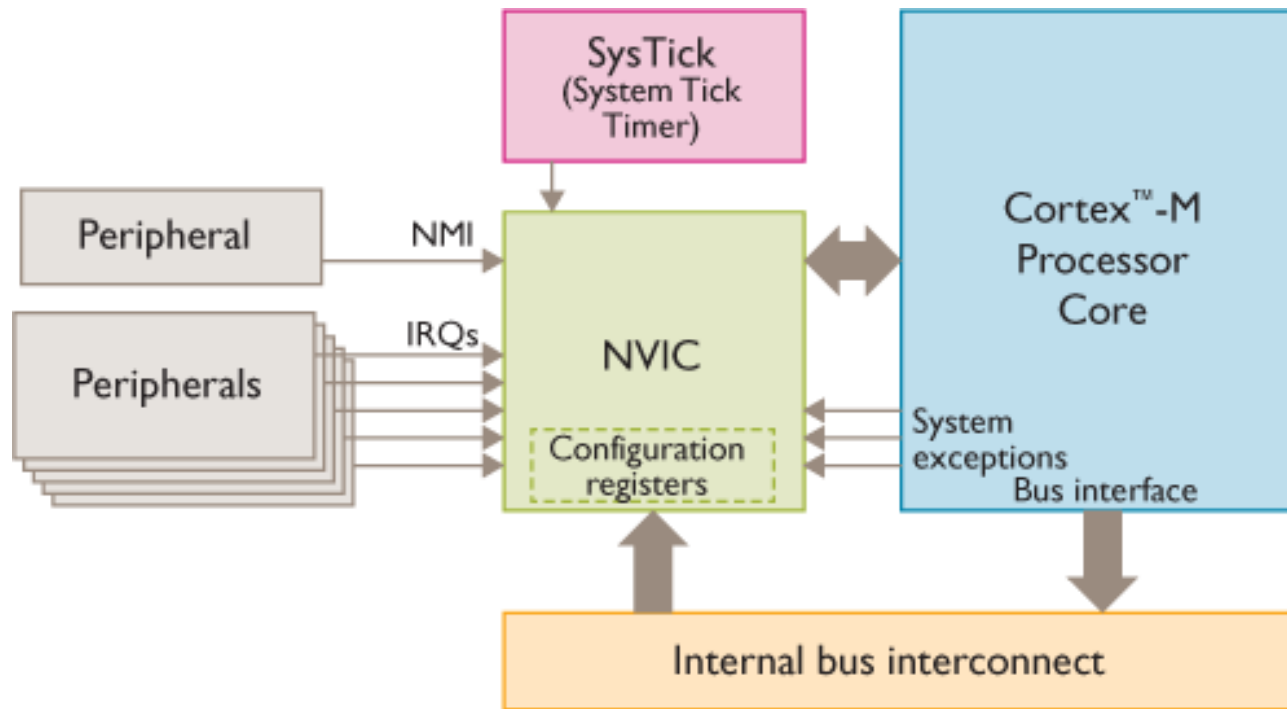


*Peripheral IRQn*



# Nested Vectored Interrupt Controller

- NVIC manages and prioritizes external interrupts in Cortex-M
  - 82 IRQ sources from STM32F4xx peripherals
- NVIC interrupts CPU with IRQ# of highest-priority IRQ signal
  - CPU uses IRQ# to access the vector table & get intr. handler start address





# NVIC registers (one bit for each IRQ#)

- NVIC\_ISER<sub>x</sub>/NVIC\_ICER<sub>x</sub>
  - Each IRQ has its own **enable** bit within NVIC
  - Interrupt Set/Clear Enable Register
  - 1 = Set (enable) interrupt/Clear (disable) interrupt
- NVIC\_ISPR<sub>x</sub>/NVIC\_ICPR<sub>x</sub>
  - Interrupt Set/Clear Pending Register
  - Read 1 from ISPR if interrupt in pending state
  - Write 1 to set interrupt to pending or clear from pending state
- NVIC\_IABR<sub>x</sub> – Interrupt Active Bit Register
  - Read 1 if interrupt in active state

EnableK

PendK

PriorityK

**x = 0..7** for each register type, with 32 bits per register, to support up to 240 IRQs (82 in STM32F4xx)

- Each bit controls one interrupt, identified by its IRQ# (0..239)
- Register#  $x = \text{IRQ\#} \text{ DIV } 32$
- Bit n in the register =  $\text{IRQ\#} \text{ MOD } 32$

# NVIC registers (continued)

- NVIC\_IPRx (x=0..59) – Interrupt Priority Registers
  - Supports up to 240 interrupts: 0..239 (82 in STM32F4)
  - 8-bit priority field for each interrupts (4-bit field in STM32F4)
    - 4 priority values per register (STM32F4 – upper 4 bits of each byte)
    - 0 = highest priority
    - Register# x = IRQ# DIV 4
    - Byte offset within the register = IRQ# MOD 4
    - Ex. IRQ85:
      - $85/4 = 21$  with remainder 1 (register 21, byte offset 1)  
Write  $\text{priority} \ll 8$  to NVIC\_IPR2
      - $85/32 = 2$  with remainder 21: write  $1 \ll 21$  to NVIC\_SER2
- STIR – Software Trigger Interrupt Register
  - Write IRQ# (0..239) to trigger that interrupt from software
  - Unprivileged access to this register enabled in system control register (SCR)

# NVIC example (assembly language)

NVIC\_ISER0/1/2 = 0xE000E100/104/108

NVIC\_ICER0/1/2 = 0xE000E180/184/188

NVIC\_IPR0/1/2/.../20 = 0xE00E400/404/408/40C/..../500

**;Example – Enable EXTI0 with priority 5 (EXTI0 = IRQ6)**

NVIC\_ISER0 EQU 0xE000E100 ;bit 6 enables EXTI0

NVIC\_IPR1 EQU 0xE000E404 ;3<sup>rd</sup> byte = EXTI0 priority

```
ldr    r0,=NVIC_ISER0
```

```
mov    r1,#0x0040 ;Set bit 6 of ISER0 for EXTI0
```

```
str    r1,[r0]
```

```
ldr    r0,=NVIC_IPR1 ;IRQ6 priority in IPR1[23:16]
```

```
ldr    r1,[r0] ;Read IPR1
```

```
bic    r1,#0x00FF0000 ;Clear [23:16] for IRQ6
```

```
orr    r1,#0x00500000 ;Bits [23:20] = 5
```

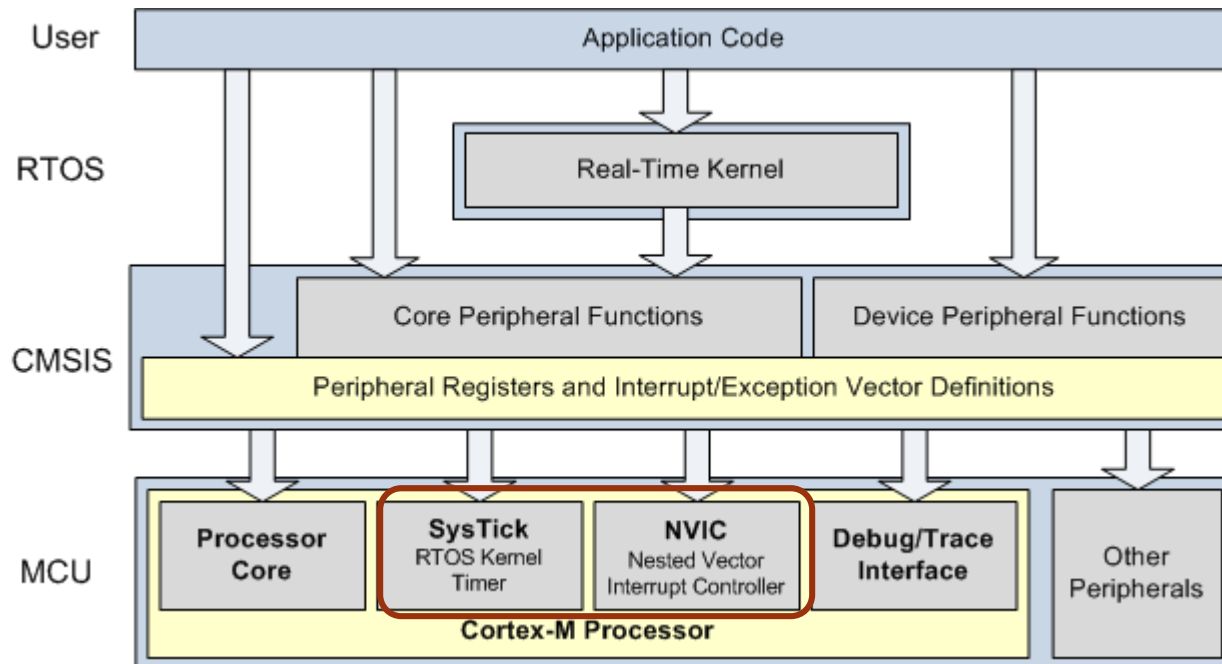
```
str    r1,[r0] ;Upper 4 bits of byte = priority
```

# CMSIS: Cortex Microcontroller Software Interface Standard

Vendor-independent hardware abstraction layer for Cortex-M

(Facilitates software reuse)

- **Core Peripheral Access Layer** provides name definitions, address definitions, and helper functions to access core registers and core peripherals.
- **Device Peripheral Access Layer (MCU specific)** offers name definitions, address definitions, and driver code to access peripherals.
- **Access Functions for Peripherals (MCU specific and optional)** implements additional helper functions for peripherals.



# CMSIS functions

Available when CMSIS Core is included in the project

- NVIC\_EnableIRQ(IRQn\_Type IRQn)
- NVIC\_DisableIRQ(IRQn\_Type IRQn)
  - Set bit IRQn in NVIC\_ISERx/NVIC\_ICERx
- NVIC\_SetPendingIRQ(IRQn\_Type IRQn)
- NVIC\_ClearPendingIRQ(IRQn\_Type IRQn)
- NVIC\_GetPendingIRQ(IRQn\_Type IRQn)
  - Set/read/read bit IRQn in NVIC\_ISPRx/NVIC\_ICPRx
- NVIC\_SetPriority(IRQn\_Type IRQn, unit32\_t priority)
- NVIC\_GetPriority(IRQn\_Type IRQn)
  - Set/get IRQn priority in NVIC\_IPRxI

# NVIC CMSIS example: enable interrupts

- *Interrupt Set Enable Register*: each bit **enables** one interrupt  
`NVIC_EnableIRQ(n); //set bit to enable IRQn`
- *Interrupt Clear Enable Register*: each bit **disables** one interrupt  
`NVIC_DisableIRQ(n); //set bit to disable IRQn`

- For convenience, *stm32f4xx.h* defines a symbol for each IRQn

Examples: `EXTI0_IRQn = 6; //External interrupt EXTI0 is IRQ #6`

`TIM3_IRQn = 29; //Timer TIM3 interrupt is IRQ #29`

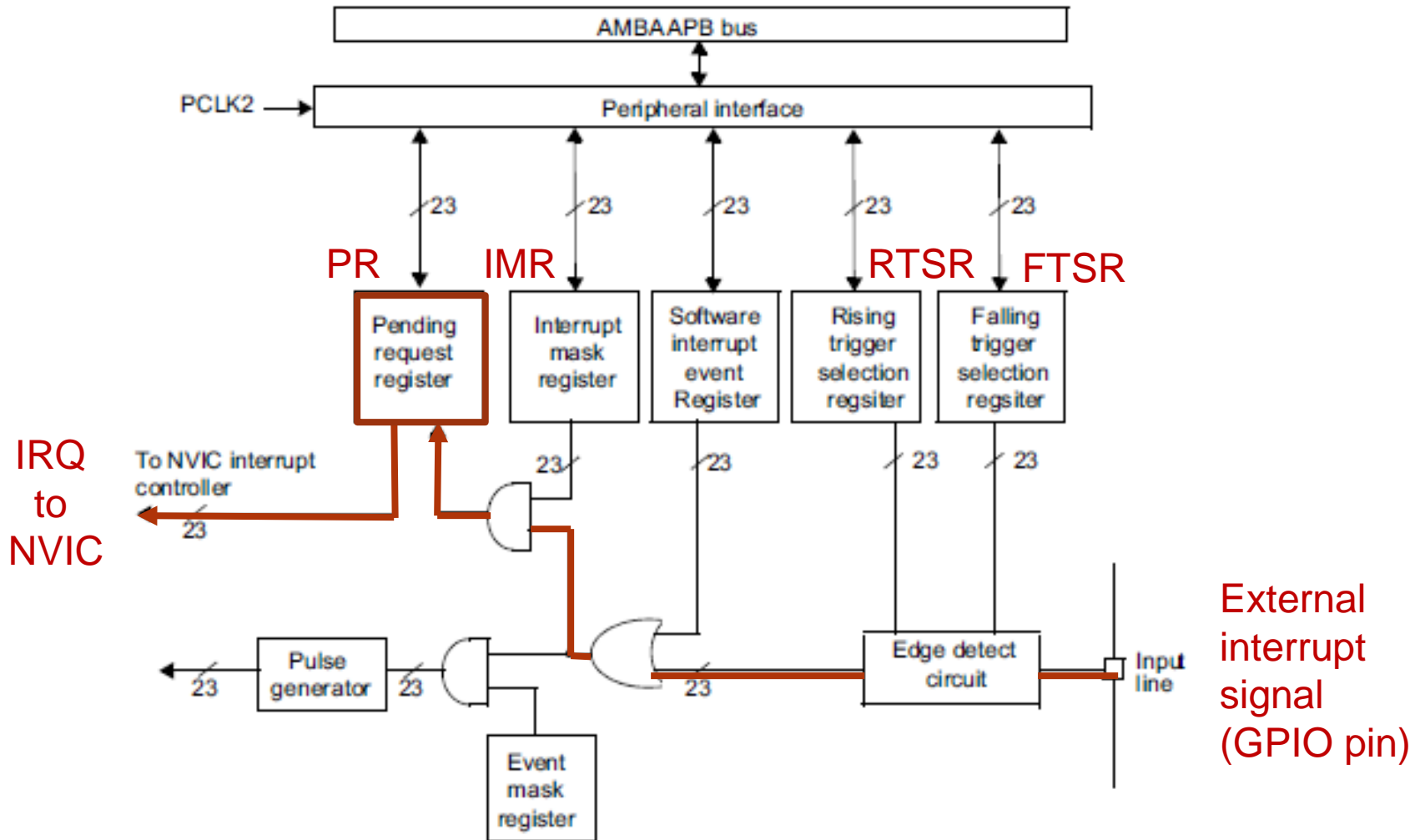
Usage:

`NVIC_EnableIRQ(EXTI0_IRQn); //enable external interrupt EXTI0`

`NVIC_DisableIRQ(TIM3_IRQn); //disable interrupt from timer TIM3`

# STM32F4 external interrupt/event controller

23 edge detectors to trigger events and interrupts signaled by 240 GPIO pins and 7 internal events.

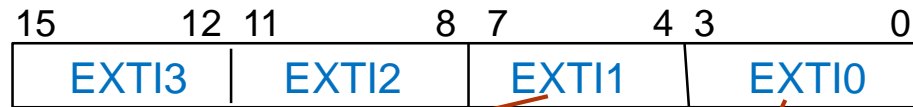


# STM32F4xx external interrupt sources

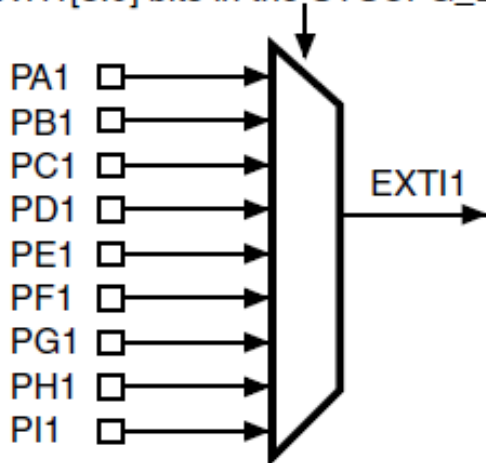
(select in System Configuration Module – SYSCFG)

- 16 multiplexers select GPIO pins as external interrupts EXTI0..EXTI15
- Mux inputs selected via 4-bit fields of EXTICR[k] registers (k=0..3)
  - $EXTIx = 0$  selects PAx, 1 selects PBx, 2 selects PCx, etc.
  - $EXTICR[0]$  selects EXTI3-EXTI0;  $EXTICR[1]$  selects EXTI7-EXTI4, etc

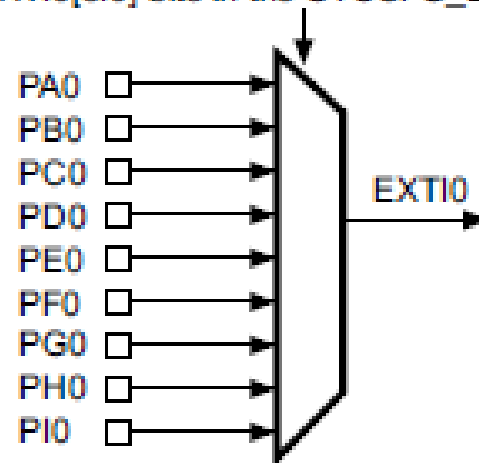
*SYSCFG\_EXTICR1 is  
SYSCFG->EXTICR[0]*



EXTI1[3:0] bits in the SYSCFG\_EXTICR1 register



EXTI0[3:0] bits in the SYSCFG\_EXTICR1 register



Example: Select pin PC2 as external interrupt EXTI2

```
SYSCFG->EXTICR[0] &= 0xF0FF; //clear EXTI2 bit field
```

```
SYSCFG->EXTICR[0] |= 0x0200; //set EXTI2 = 2 to select PC2
```



# STM32F4 external interrupt sources

## Sixteen external interrupts

EXTI0 – EXTI15

## Seven “event” triggers:

EXTI16 = PVD output

EXTI17 = RTC Alarm event

EXTI18 = USB OTG FS Wakeup event

EXTI19 = Ethernet Wakeup event

EXTI20 = USB OTG HS Wakeup event

EXTI21 = RTC Tamper and TimeStamp events

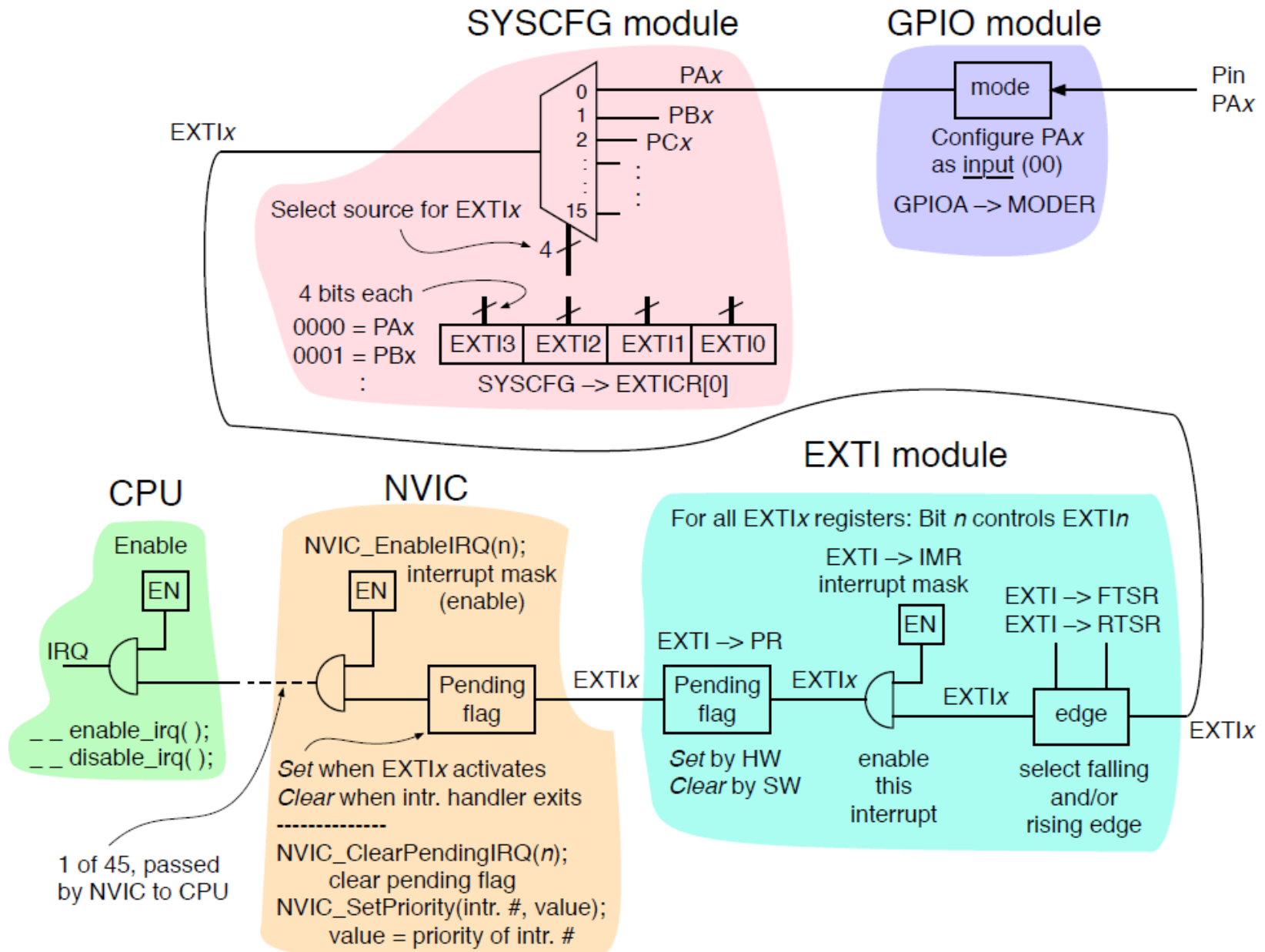
EXTI22 = RTC Wakeup event

# STM32F4 EXTI Registers

23 bits per register - control 23 interrupts/events

- EXTI\_IMR – interrupt **mask** register
  - 0 masks (disables) the interrupt
  - 1 unmask (enables) the interrupt
- EXTI\_RTSR/FTSR – **rising/falling trigger** selection register
  - 1 to enable rising/falling edge to trigger the interrupt/event
  - 0 to ignore the rising/falling edge
- EXTI\_PR – interrupt/event **pending** register
  - read 1 if interrupt/event occurred
  - clear bit by writing 1 (*writing 0 has no effect*)
  - **write 1 to this bit in the interrupt handler to clear the pending state of the interrupt**
- EXTI\_SWIER – **software interrupt** event register
  - 1 to set the pending bit in the PR register
  - Triggers interrupt if not masked

# Signal flow/setup for External Interrupt EXT<sub>Ix</sub>, x = 0...15



# Project setup for interrupt-driven applications

- Write the interrupt handler for each peripheral
  - Clear the flag that requested the interrupt (acknowledge the intr. request)
  - Perform the desired actions, communicating with other functions via shared global variables
  - Use function names from the vector table  
Example: *void EXTI4\_IRQHandler () { statements }*
- Perform all initialization for each peripheral device:
  - Initialize the device, “arm” its interrupt, and clear its “flag”  
Example: External interrupt EXTIIn
    - Configure GPIO pin as a digital input
    - Select the pin as the EXTIIn source (in SYSCFG module)
    - Enable interrupt to be requested when a flag is set by the desired event (rising/falling edge)
    - Clear the pending flag (to ignore any previous events)
  - NVIC
    - Enable interrupt: *NVIC\_EnableIRQ (IRQn);*
    - Set priority: *NVIC\_SetPriority (IRQn, priority);*
    - Clear pending status: *NVIC\_ClearPendingIRQ (IRQn);*
- Initialize counters, pointers, global variables, etc.
- Enable CPU Interrupts: *\_\_enable\_irq ( ) ;*

# Example: Enable EXTI0 as rising-edge triggered

## **;System Configuration Registers**

```
SYSCFG EQU 0x40013800
EXTICR1 EQU 0x08
```

## **;External Interrupt Registers**

```
EXTI EQU 0x40013C00
IMR EQU 0x00 ;Interrupt Mask Register
RTSR EQU 0x08 ;Rising Trigger Select
FTSR EQU 0x0C ;Falling Trigger Select
PR EQU 0x14 ;Pending Register
```

## **;select PC0 as EXTI0**

```
ldr r1,=SYSCFG ;SYSCFG selects EXTI sources
ldrh r2,[r1,#EXTICR1] ;EXTICR1 = sources for EXTI0 - EXTI3
bic r2,#0x000f ;Clear EXTICR1[3-0] for EXTI0 source
orr r2,#0x0002 ;EXTICR1[3-0] = 2 to select PC0 as EXTI0 source
strh r2,[r1,#EXTICR1] ;Write to select PC0 as EXTI0
```

## **;configure EXTI0 as rising-edge triggered**

```
ldr r1,=EXTI ;EXTI register block
mov r2,#1 ;bit #0 for EXTI0 in each of the following registers
str r2,[r1,#RTSR] ;Select rising-edge trigger for EXTI0
str r2,[r1,#PR] ;Clear any pending event on EXTI0
str r2,[r1,#IMR] ;Enable EXTI0
```

## EXTI example – accessing registers directly (in C)

```
#include "STM32F4xx.h"
/*-----
   Initialize the GPIO and the external interrupt
   *-----*/
void Init_Switch(void){

    //Enable the clock for GPIO
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;

    //Pull-up pin 0
    GPIOA->PUPDR |= GPIO_PUPDR_PUPDR0_1;

    //Connect the portA pin0 to external interrupt line0
    SYSCFG->EXTICR[0] &= SYSCFG_EXTICR1_EXTI0_PA;

    //Interrupt Mask
    EXTI->IMR |= (1<<0);

    //Falling trigger selection
    EXTI->FTSR |= (1<<0);

    //Enable interrupt
    __enable_irq();

    //Set the priority
    NVIC_SetPriority(EXTI0_IRQn,0);

    //Clear the pending bit
    NVIC_ClearPendingIRQ(EXTI0_IRQn);

    //Enable EXTI0
    NVIC_EnableIRQ(EXTI0_IRQn);}

/*-----
   Interrupt Handler – count button presses
   *-----*/
void EXTI0_IRQHandler(void) {

    //Make sure the Button is really pressed
    if (!(GPIOA->IDR & (1<<0)) )
    {
        count++;
    }

    //Clear the EXTI pending bits
    NVIC_ClearPendingIRQ(EXTI3_IRQn);
    EXTI->PR|=(1<<0);
}
```

# Supervisor Call Instruction (SVC)

- Access system resources from O/S (“privileged operations”)
- **SVC\_Handler** is defined in the interrupt vector table
- SVC interrupt handler written as a C function:

```
void SVC_Handler()  
{ your code }
```

- SVC interrupt handler as an assembly language function:

```
EXTERN SVC_Handler  
  
SVC_Handler  
    your code  
    bx lr
```

# Supervisor Call instruction (SVC)

- To execute **SVC\_Handler** as a software interrupt
  - Assembly language syntax: **SVC #imm**
  - C syntax: **\_\_svc (imm)**
- **imm** is an “SVC number” (0-255), which indicates a particular “service” to be performed by the handler
  - **imm** is encoded into the instruction, but ignored by the CPU
  - Handler can retrieve **imm** by using stacked PC to read the SVC instruction code (examples provided later)
- Since this is an “interrupt”, R0-R3 are pushed onto the stack:
  - Arguments can be passed to the handler in R0-R3
  - SVC handler can retrieve the arguments from the stack
  - SVC handler can also return results by replacing R0-R3 values in the stack, which will be restored to R0-R3 on return from interrupt.



# Access SVC arguments in C

```
// Stack contains eight 32-bit values:  
//  r0, r1, r2, r3, r12, r14, return address, xPSR  
//  1st argument = r0 = svc_args[0]  
//  2nd argument = r1 = svc_args[1]  
//  7th argument = return address = svc_args[6]
```

```
void SVC_Handler(unsigned int * svc_args) {  
    int a,b,c;  
  
    a = svc_args[0];           //get first argument from stack  
    b = svc_args[1];           //get second argument from stack  
    c = a + b;  
    svc_args[0] = c;           //replace R0 value in stack with result to "return" result in R0  
  
}  
}
```

# Access SVC arguments in assembly language

; Stack contains: r0, r1, r2, r3, r12, r14, return address, xPSR

; The saved r0 is the top entry in the stack

## **EXPORT SVC\_Handler**

### **SVC\_Handler**

TST	LR,#0x04	;Test bit 2 of EXC_RETURN
ITE	EQ	;Which stack pointer was used?
MRSEQ	R4,MSP	;Copy Main SP to R4
MRSNE	R4,PSP	;Copy Process SP to R4
LDR	R1,[R4]	;Retrieve saved R0 from top of stack
LDR	R2,[R4,#4]	;Retrieve saved R1 from stack
....		
STR	R1,[R4]	;Replace saved R0 value in stack
BX	LR	;Return and restore registers from stack

# SVC in C programs

- May associate “standard” function name with the `__svc (imm)` function call

- May pass up to four integer arguments
- May return up to four results in a “value\_in\_regs” structure
- Syntax:

*`__svc(int svc_num) return-type function-name(argument-list)`*

- *`svc_num`* (8-bit constant) = immediate value in SVC instruction
- *“return-type function-name(argument-list)”* = C function prototype
- Call the function via: `function-name(argument-list);`  
(examples on next slide)

# Example: SVC call from C code

```
/*-----  
    Set up SVC "calls" to "SVC_Handler"  
    SVC_Handler function must be defined elsewhere  
*-----*/  
#define SVC_00 0x00  
#define SVC_01 0x01  
  
/* define function "svc_zero" as SVC #0, passing pointer in R0 */  
/* define function "svc_one" as SVC #1, passing pointer in R0 */  
void __svc(SVC_00) svc_zero(const char *string);  
void __svc(SVC_01) svc_one(const char *string);  
  
int call_system_func(void) {  
    svc_zero("String to pass to SVC handler zero");    //Execute SVC #0  
    svc_one("String to pass to a different OS function"); //Execute SVC #1  
}
```

Reference: *ARM Compiler toolchain Developing Software for ARM Processors*: Supervisor Calls, Example 56

# SVC\_Handler with SVC #imm operand (example in *MDK-ARM Help*)

```
// Stack contains eight 32-bit values:  
// r0, r1, r2, r3, r12, r14, return address, xPSR  
// 1st argument = r0 = svc_args[0]  
// 2nd argument = r1 = svc_args[1]  
// 7th argument = return address = svc_args[6]
```

```
void SVC_Handler(unsigned int * svc_args) {  
    unsigned int svc_number;
```

```
    //Read SVC# byte from SVC instruction code  
    svc_number = ((char *)svc_args[6])[-2];
```

```
    //Execute code for each SVC #
```

```
    switch(svc_number) {  
        case SVC_00: /* Handle SVC 00 */  
            break;  
        case SVC_01: /* Handle SVC 01 */  
            break;  
        default: /* Unknown SVC */  
            break;
```

```
    }
```

*Ignore SVC#  
if only one  
“service” in  
the handler*

# Access SVC immediate operand in assembly language

; Parameters in R0-R3 were pushed onto the stack

## **EXPORT SVC\_Handler**

### **SVC\_Handler**

```
TST      LR,#0x04           ;Test bit 2 of EXC_RETURN
ITE      EQ                 ;Which stack pointer was used?
MRSEQ   R0,MSP             ;Copy Main SP to R0
MRSNE   R0,PSP            ;Copy Process SP to R0
LDR     R1,[R0,#24]        ;Retrieve stacked PC from stack
LDRB    R0,[R1,#-2]        ;Get #N from SVC instruction in program
ADR     R1,SVC_Table       ;SVC Vector Table address
LDR     PC,[R1,R0,SLL #2]  ;Branch to Nth routine
```

....

```
SVC_TABLE          ;Table of function addresses
```

```
DCD SVC0_Function
```

```
DCD SVC1_Function
```

```
DCD SVC2_Function
```

# System tick timer interrupts

- **SysTick Timer** is a simple 24-bit down counter
  - Interrupt on count down from 1 -> 0
  - Counter rolls over from 0 to 24-bit “reload” value (determines interrupt period)
  - User provides interrupt handler: `SysTick_Handler(void)`
- Control register bits:
  - 0: enable
  - 1: interrupt enable
  - 2: clock source
    - FCLK = free-running internal core clock (default)
    - STCLK = external clock signal
  - 16: rollover flag (set on count down from 1->0)
- CMSIS function starts timer, enables interrupt, selects clock source and sets reload value:

```
#include “core_cm4.h”
```

```
SysTick_Config(numberOfTicks); //Ex. #ticks = SystemCoreClock/1000
```