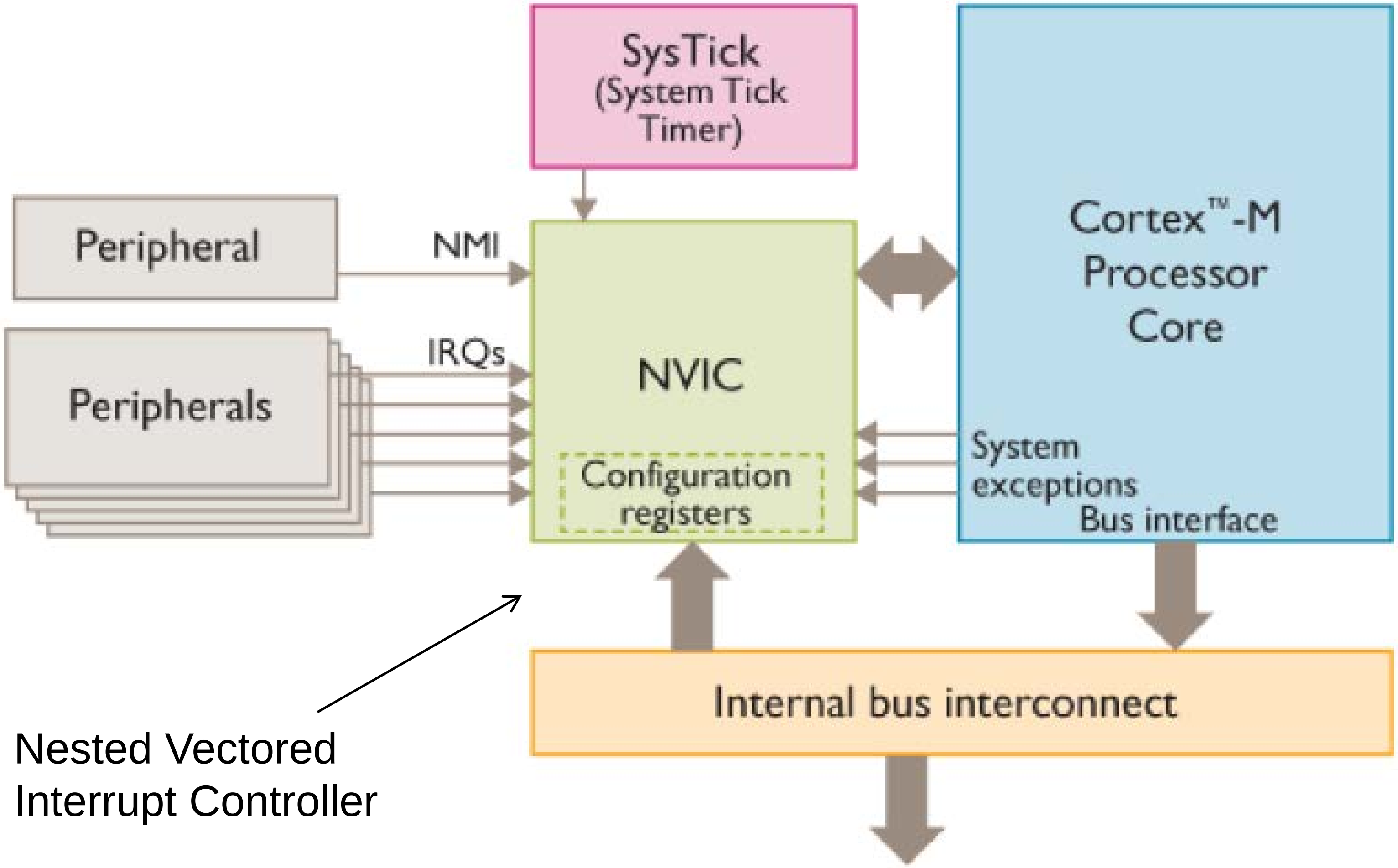
ARM and STM32L4xx
Operating Modes & Interrupt Handling

ARM Cortex-M4 User Guide (Interrupts, exceptions, NVIC)

STM32L4xx Microcontrollers Technical Reference Manual

Cortex-M structure

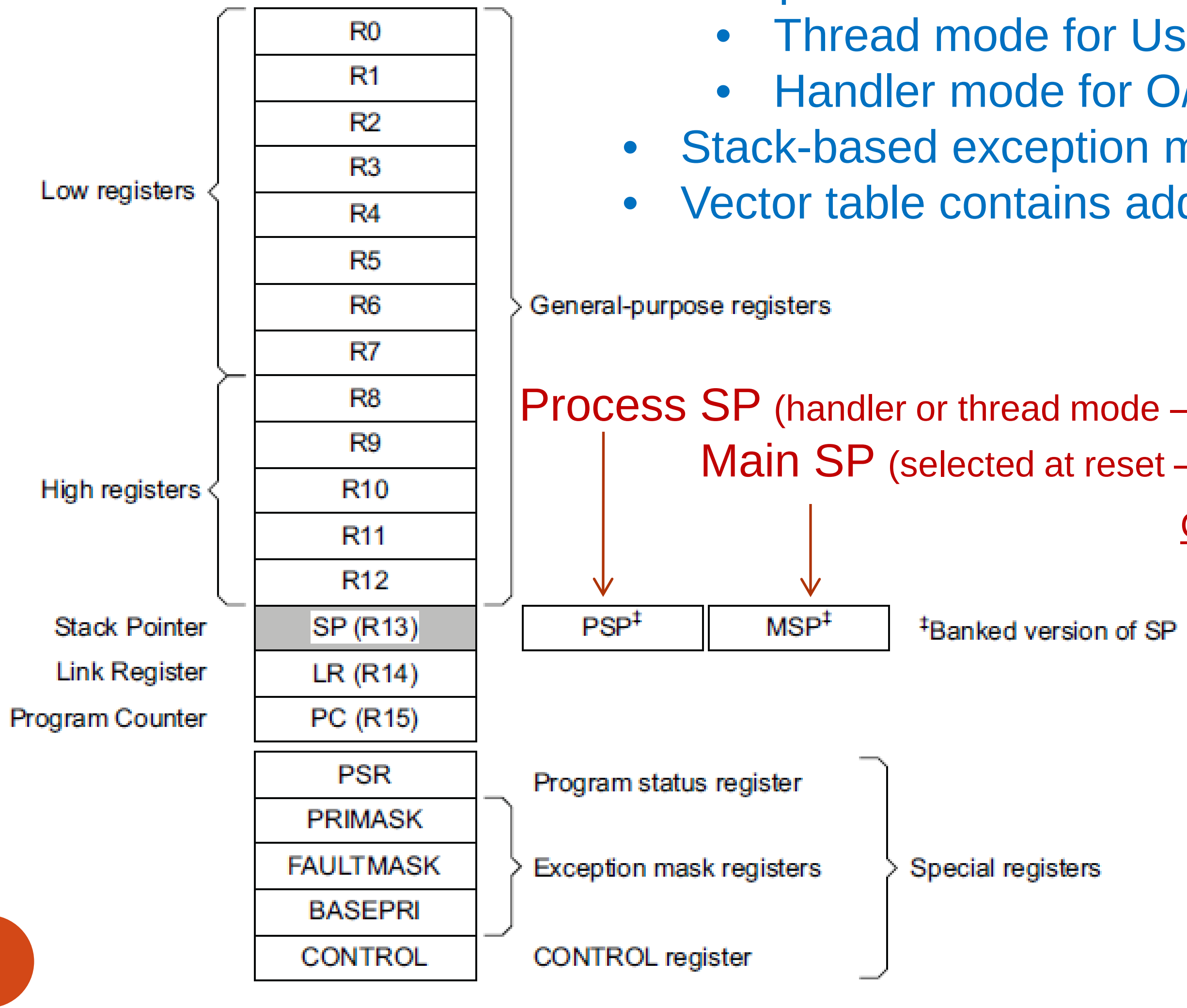


Nested Vectored
Interrupt Controller

CMSIS = Cortex Microcontroller Software Interface Standard

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



Low registers

R0

R1

R2

R3

R4

R5

R6

R7

High registers

R8

R9

R10

R11

R12

General-purpose registers

Stack Pointer SP (R13)

Link Register LR (R14)

Program Counter PC (R15)

PSP‡ MSP‡

‡Banked version of SP

PSR — Program status register

PRIMASK

FAULTMASK

BASEPRI

Exception mask registers

CONTROL — CONTROL register

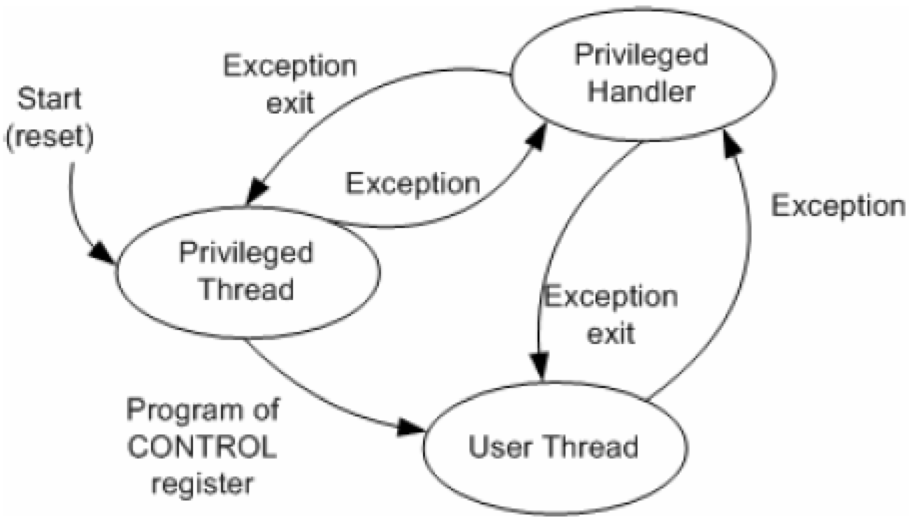
Special registers

Process SP (handler or thread mode – select in CONTROL reg.)

Main SP (selected at reset – always used in handler mode)

Convention:
PSP in thread mode,
MSP in O/S & handler mode

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

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.

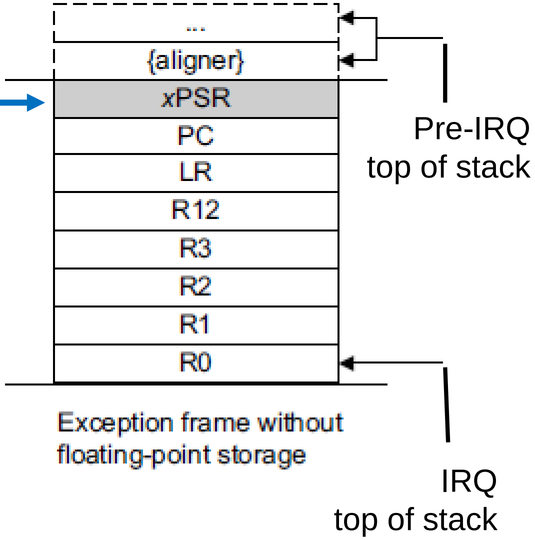
- An interrupt request from a peripheral or from software can change the state of the corresponding interrupt to pending.

- An exception handler can interrupt (preempt) the execution of another exception handler. In this case both exceptions are in the active state.

Cortex-M Interrupt Process

(much of this is transparent when using C)

1. Interrupt signal detected by CPU
2. Suspend main program execution
   - finish current instruction
   - save CPU state (push registers onto stack)
   - set LR to 0xFFFFFFF9 (indicates interrupt return)
   - set IPSR to interrupt number
   - load PC with ISR address from vector table
3. Execute interrupt service routine (ISR)
   - save other registers to be used [1]
   - clear the "flag" that requested the interrupt
   - perform the requested service
   - communicate with other routines via global variables
   - restore any registers saved by the ISR [1]
4. Return to and resume main program by executing ***BX LR***
   - saved state is restored from the stack, including PC



Exception frame without floating-point storage

[1] C compiler takes care of saving/restoring registers

Cortex-M CPU and peripheral exceptions

CPU Exceptions

	Priority[1]	IRQ#[2]	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

Vendor peripheral interrupts
IRQ0 .. IRQ90

[1] *Lowest priority # = highest priority*

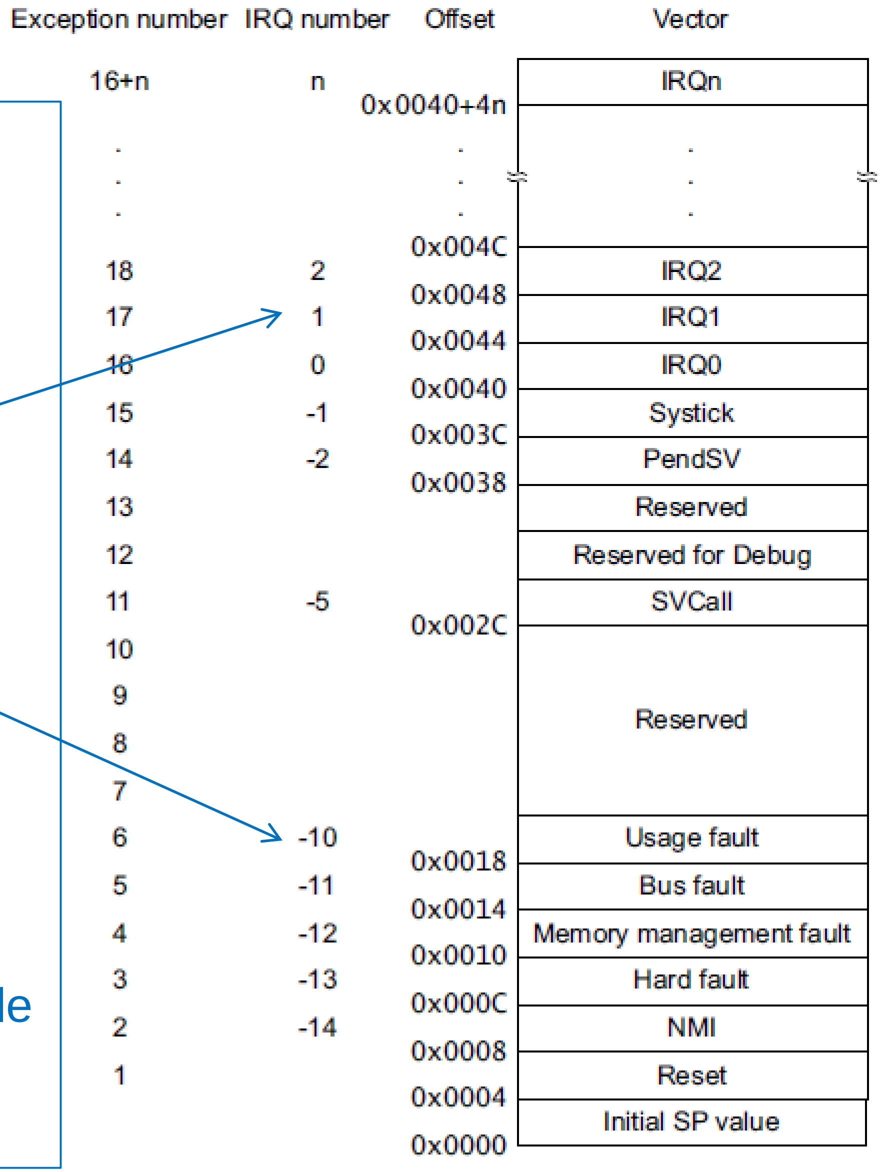
[2] *IRQ# used in CMSIS function calls*

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

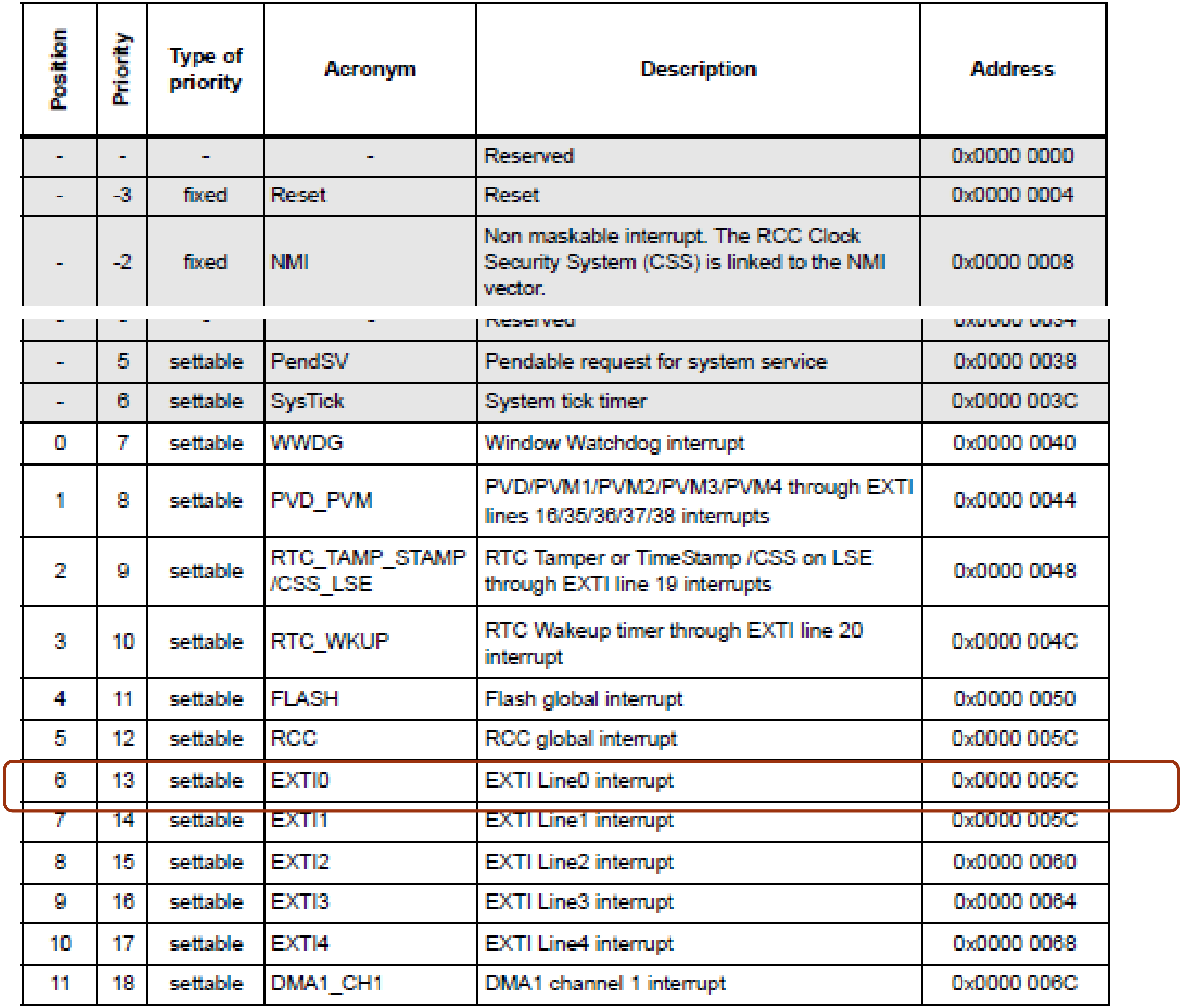


Exception number	IRQ number	Offset	Vector
16+n	n	0x0040+4n	IRQn
.		.	.
.		.	.
.		.	.
		0x004C	
18	2	0x0048	IRQ2
17	1	0x0044	IRQ1
16	0	0x0040	IRQ0
15	-1	0x003C	Systick
14	-2	0x0038	PendSV
13			Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-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

STM32L4 Vector Table (partial)

Tech. Ref. Table 57

(Refer to Startup Code)



Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000 0000
-	-3	fixed	Reset	Reset	0x0000 0004
-	-2	fixed	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000 0008
-	-	-	-	Reserved	0x0000 0034
-	5	settable	PendSV	Pendable request for system service	0x0000 0038
-	6	settable	SysTick	System tick timer	0x0000 003C
0	7	settable	WWDG	Window Watchdog interrupt	0x0000 0040
1	8	settable	PVD_PVM	PVD/PVM1/PVM2/PVM3/PVM4 through EXTI lines 16/35/36/37/38 interrupts	0x0000 0044
2	9	settable	RTC_TAMP_STAMP /CSS_LSE	RTC Tamper or TimeStamp /CSS on LSE through EXTI line 19 interrupts	0x0000 0048
3	10	settable	RTC_WKUP	RTC Wakeup timer through EXTI line 20 interrupt	0x0000 004C
4	11	settable	FLASH	Flash global interrupt	0x0000 0050
5	12	settable	RCC	RCC global interrupt	0x0000 005C
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000 005C
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_CH1	DMA1 channel 1 interrupt	0x0000 006C

STM32L4 vector table from **startup code** (partial)

```
              AREA    RESET, DATA, READONLY

__Vectors     DCD     __initial_sp                ; Top of Stack
              DCD     Reset_Handler               ; Reset Handler
              DCD     NMI_Handler                 ; NMI Handler
                  ......
              DCD     SVC_Handler                 ; SVCall 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_PVM_IRQHandler          ; PVD/PVM… 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
```

Special CPU registers

ARM instructions to “access special registers”

```
MRS    Rd,spec    ;move from special register (other than R0-R15) to Rd
MSR    spec,Rs    ;move from register Rs to special register
```

Use CMSIS[1] functions to clear/set PRIMASK

__enable_irq(); //enable interrupts (set PRIMASK=0)

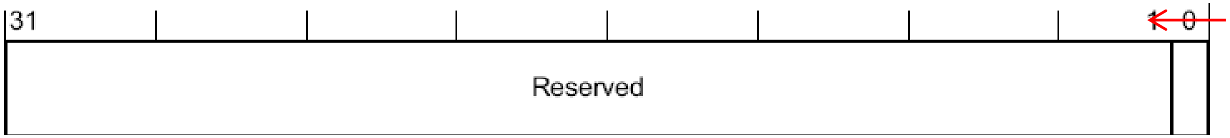
__disable_irq(); //disable interrupts (set PRIMASK=1)

(double-underscore at beginning)

Special Cortex-M Assembly Language Instructions

```
CPSIE  I     ;Change Processor State/Enable Interrupts (sets PRIMASK = 0)
CPSID  I     ;Change Processor State/Disable Interrupts (sets PRIMASK = 1)
```

Prioritized Interrupts Mask Register (PRIMASK)

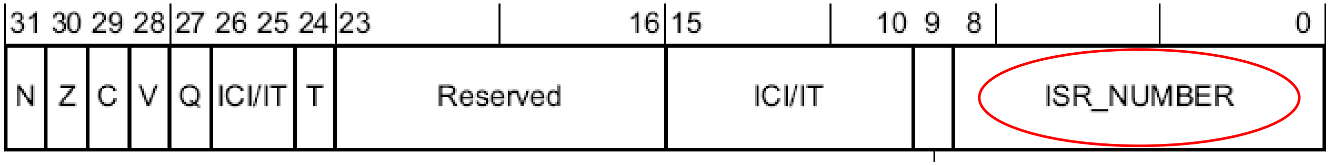


31 … 1	0
Reserved	

PRIMASK

PRIMASK = 1 prevents (masks) activation of all exceptions with configurable priority
PRIMASK = 0 permits (enables) exceptions

Processor Status Register (PSR)

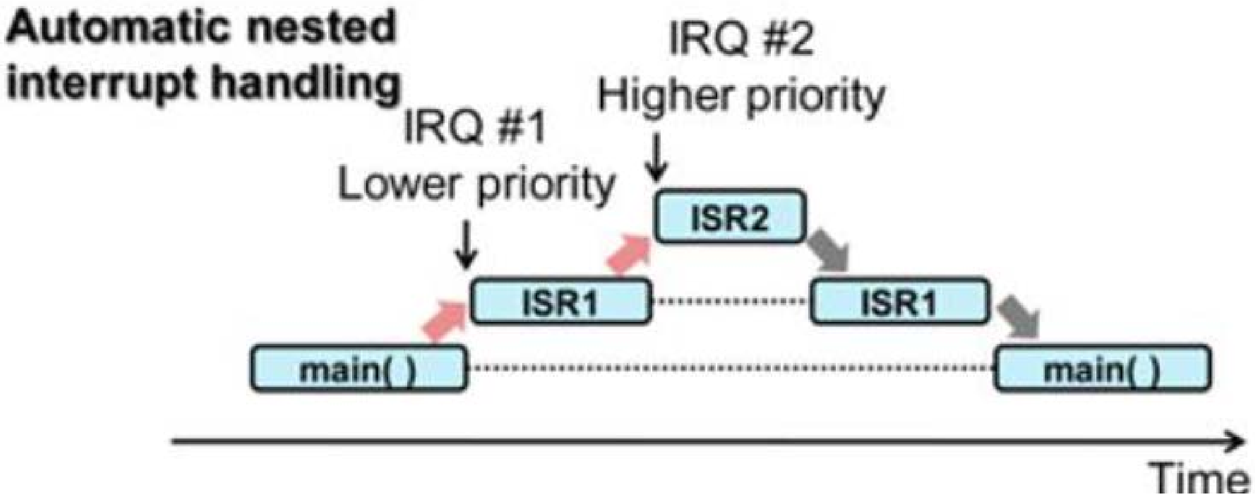


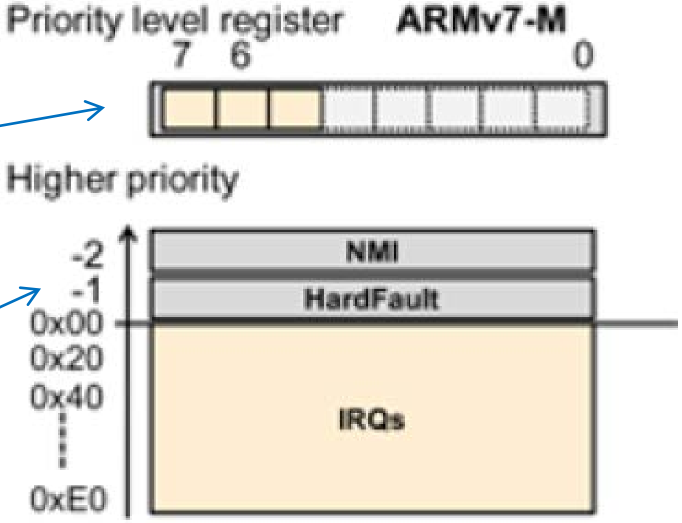
31	30	29	28	27	26 25	24	23 … 16	15 … 10	9	8 … 0
N	Z	C	V	Q	ICI/IT	T	Reserved	ICI/IT		ISR_NUMBER

# of current exception (lower priority cannot interrupt)

[1] **Cortex Microcontroller Software Interface Standard** – Functions for all ARM Cortex-M CPUs, defined in project header files: *core_cmFunc.h, core_cm3.h*

Prioritized interrupts

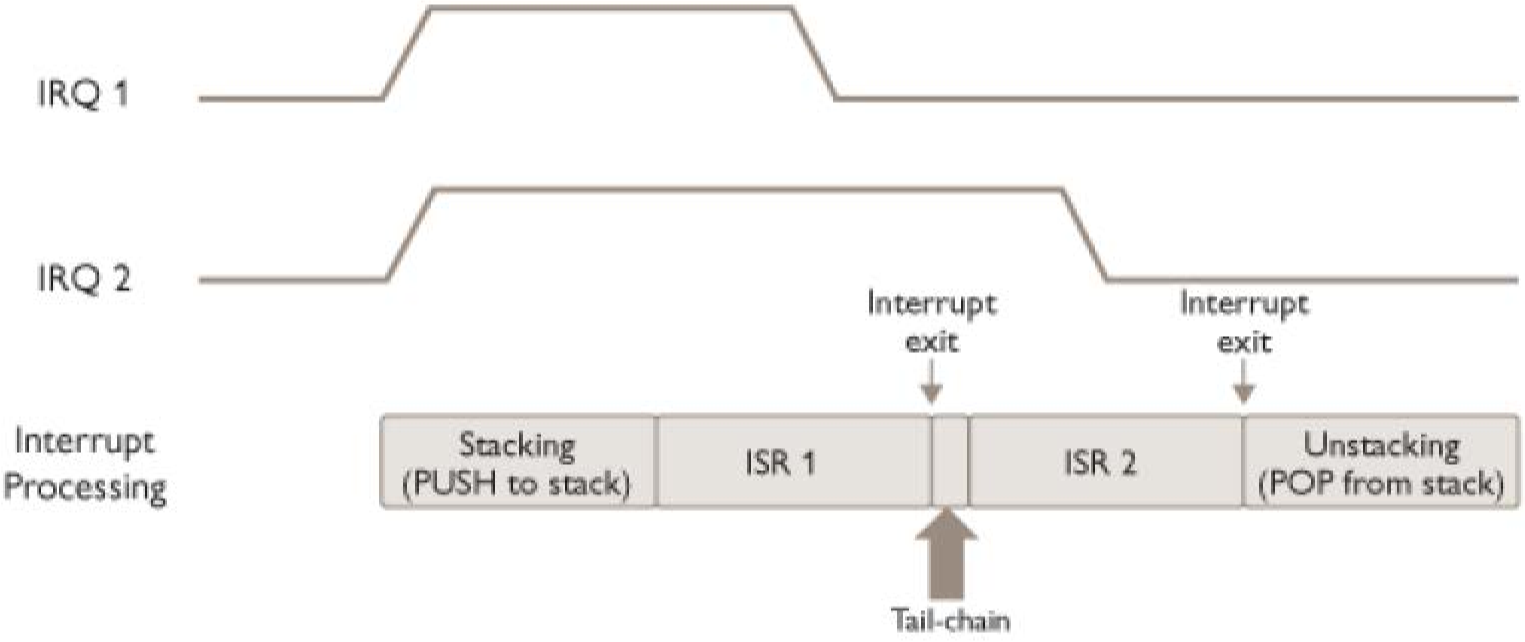




- Up to 256 priority levels
  - 8-bit priority value
  - Implementations may use fewer bits

    STM32L4xx uses upper 4 bits of each priority byte => 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:

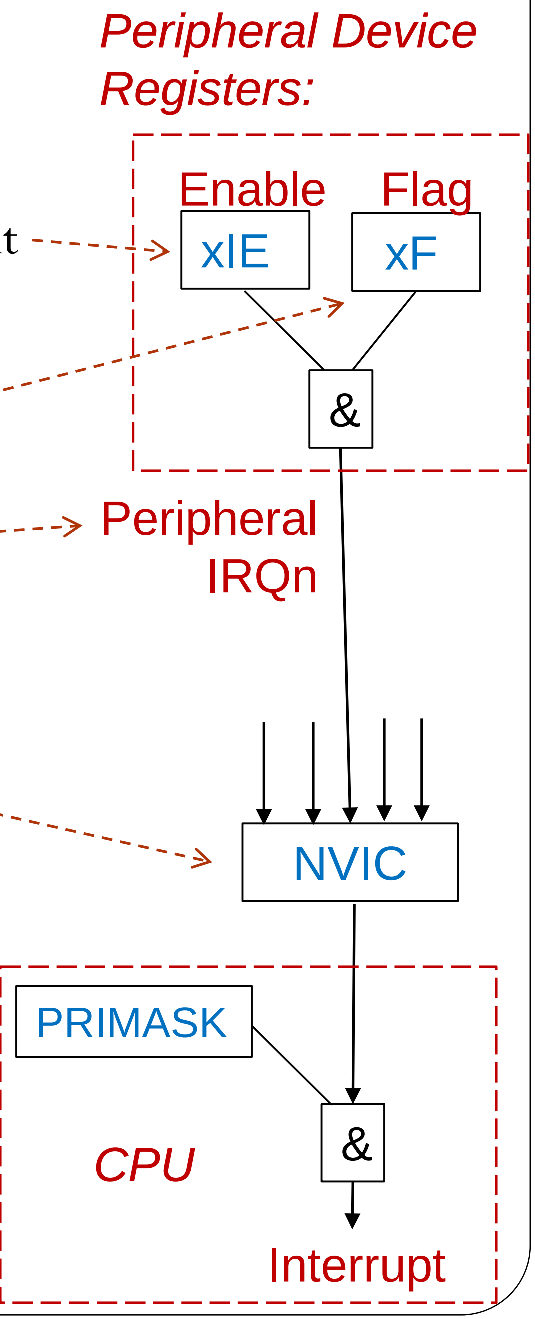
- Each potential interrupt source has a separate **arm (enable)** bit
  - Set for devices from which interrupts, are to be accepted
  - Clear to prevent the peripheral from interrupting the CPU
- Each potential interrupt source has a separate **flag** bit
  - hardware sets the flag when an "event" occurs
  - 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

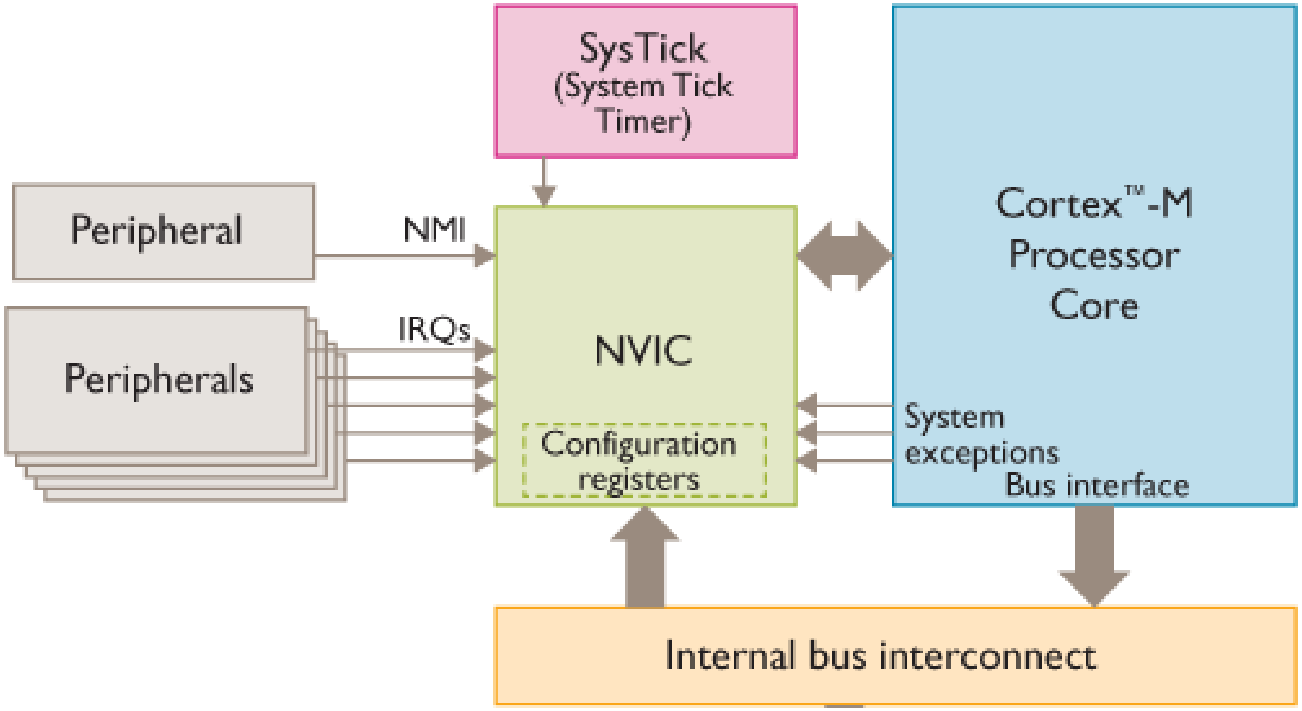
Within the CPU:

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



Nested Vectored Interrupt Controller

- NVIC manages and prioritizes external interrupts in Cortex-M
  - 90 IRQ sources from STM32L4xx 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_ISERx/NVIC_ICERx
  - Each IRQ has its own **enable** bit within NVIC
  - Interrupt Set/Clear Enable Register
  - 1 = Set (enable) interrupt/Clear (disable) interrupt

EnableK

- NVIC_ISPRx/NVIC_ICPRx
  - 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

PendK

- NVIC_IABRx – Interrupt Active Bit Register
  - Read 1 if interrupt in active state

x = 0..7 for each register type, with 32 bits per register, to support up to 240 IRQs (90 in STM32L4xx)

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

NVIC registers (continued)

- NVIC_IPRx (x=0..59) – Interrupt Priority Registers
  - Supports up to 240 interrupts: 0..239 (90 in STM32L4)
  - 8-bit priority field for each interrupts (4-bit field in STM32L4)
    - 4 priority values per register (STM32L4 – upper 4 bits of each byte)
    - 0 = highest priority
    - Register# x = IRQ# DIV 4
    - Byte offset within the register = IRQ# MOD 4
    - Ex. IRQ85:
      - o 85/4 = 21 with remainder 1 (register 21, byte offset 1)
        Write priority<<8 to NVIC_IPR2
      - o 85/32 = 2 with remainder 21: write 1<<21 to NVIC_SER2

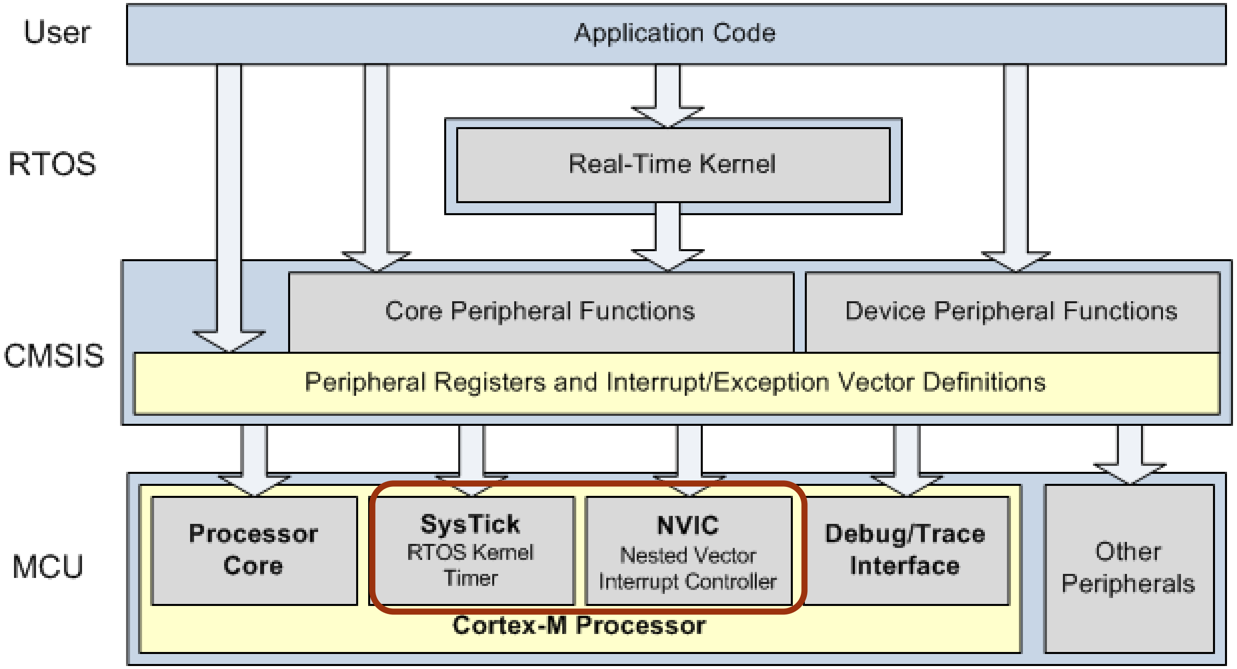
Priority

- 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)

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.


User
Application Code
RTOS
Real-Time Kernel
CMSIS
Core Peripheral Functions
Device Peripheral Functions
Peripheral Registers and Interrupt/Exception Vector Definitions
MCU
Processor Core
SysTick
RTOS Kernel Timer
NVIC
Nested Vector Interrupt Controller
Debug/Trace Interface
Cortex-M Processor
Other Peripherals

NVIC CMSIS functions: enable/disable 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, *stm32l476xx.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
  ```

NVIC CMSIS functions: interrupt pending flags

- NVIC **interrupt pending flag** for each IRQ
  - NVIC sets pending flag when it detects IRQn request
    - IRQn status changes to "pending"
    - IRQn status changes to "active" when its interrupt handler is entered
  - NVIC clears pending flag when handler exited
    - IRQn status changes to "inactive"
- CMSIS functions to set/clear/get IRQn pending status
  - Write 1 to NVIC_ICPRx to clear IRQn from pending state

    NVIC_ClearPendingIRQ(IRQn);
  - Write 1 to NVIC_ISPRx to force IRQn into pending state

    NVIC_SetPendingIRQ(IRQn); //simulates IRQn request
  - Read 1 from ISPR if IRQn in pending state

    NVIC_GetPendingIRQ(IRQn);

  - If IRQn still active when exiting handler, or IRQn reactivates while executing the handler, the pending flag remains set and **triggers another interrupt**
  - **Avoid duplicate service** by clearing IRQn pending flag in software:

NVIC CMSIS functions: interrupt priorities

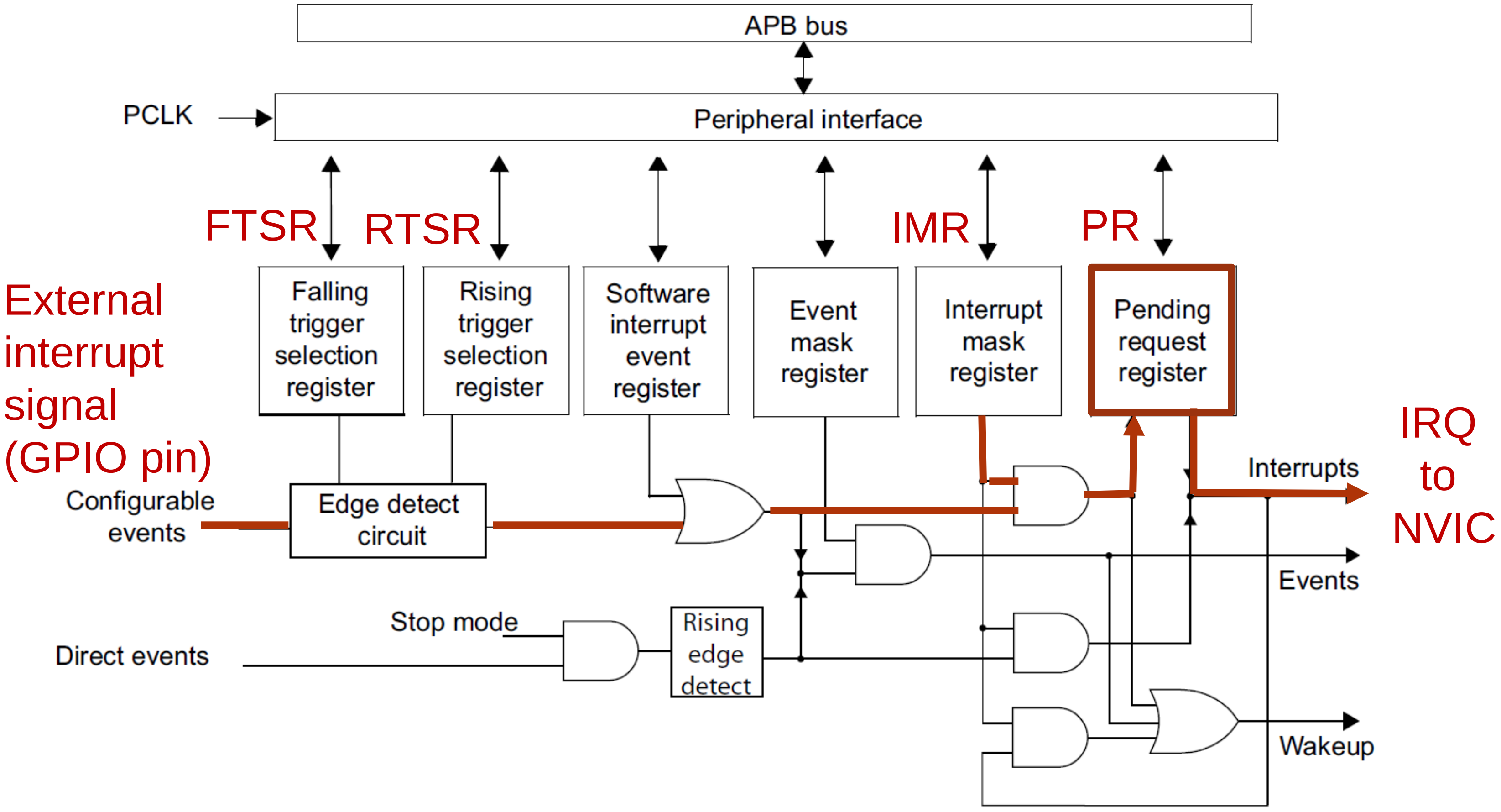
- Each IRQn assigned a **priority** within the NVIC
- NVIC selects highest-priority pending IRQ to send to CPU
  - Lower priority# = higher priority (default value = 0)
    - If equal priorities, lower IRQ# selected
    - STM32L4xx uses 4-bit priority value (0..15)

    (NVIC registers allocate 8 bits per IRQ#, but vendors may use fewer bits)
  - Higher priority IRQ can interrupt lower priority one
  - Lower priority IRQ not sent to CPU until higher priority IRQ service completed
- **Set** IRQn priority via CMSIS function: ***NVIC_SetPriority(IRQn, priority);***

  Ex: *NVIC_SetPriority(EXTI0_IRQn, 1);* //set ext. intr. EXTI0 priority = 1
- **Get** IRQn priority via CMSIS function: ***NVIC_GetPriority(IRQn_Type IRQn)***

STM32L4 external interrupt/event controller

- 26 configurable event/interrupt requests
  - external interrupts (via GPIO) and some peripherals
- 14 direct event/interrupt requests
  - from peripherals to generate wakeup from stop event or interrupt



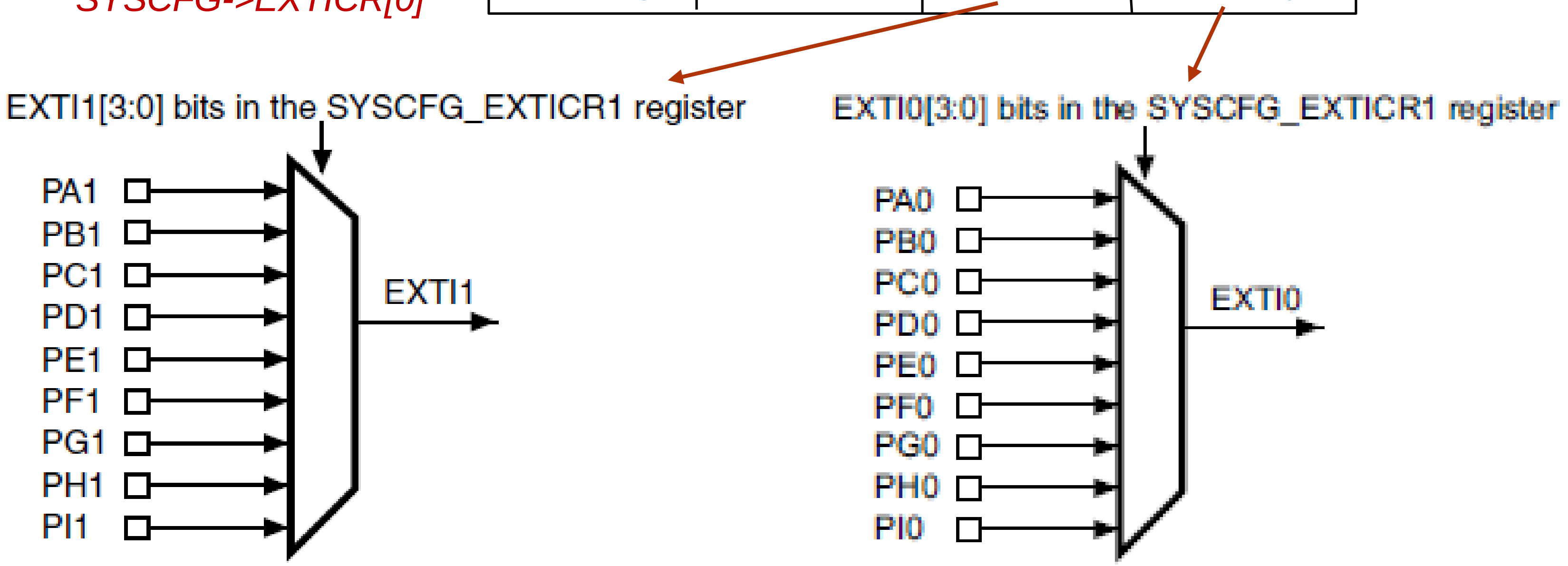
STM32L4xx external interrupt sources

(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]

15 12	11 8	7 4	3 0
EXTI3	EXTI2	EXTI1	EXTI0


EXTI1[3:0] bits in the SYSCFG_EXTICR1 register
PA1
PB1
PC1
PD1
PE1
PF1
PG1
PH1
PI1
EXTI1
EXTI0[3:0] bits in the SYSCFG_EXTICR1 register
PA0
PB0
PC0
PD0
PE0
PF0
PG0
PH0
PI0
EXTI0

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
```

STM32L476 EXTI interrupts and events.

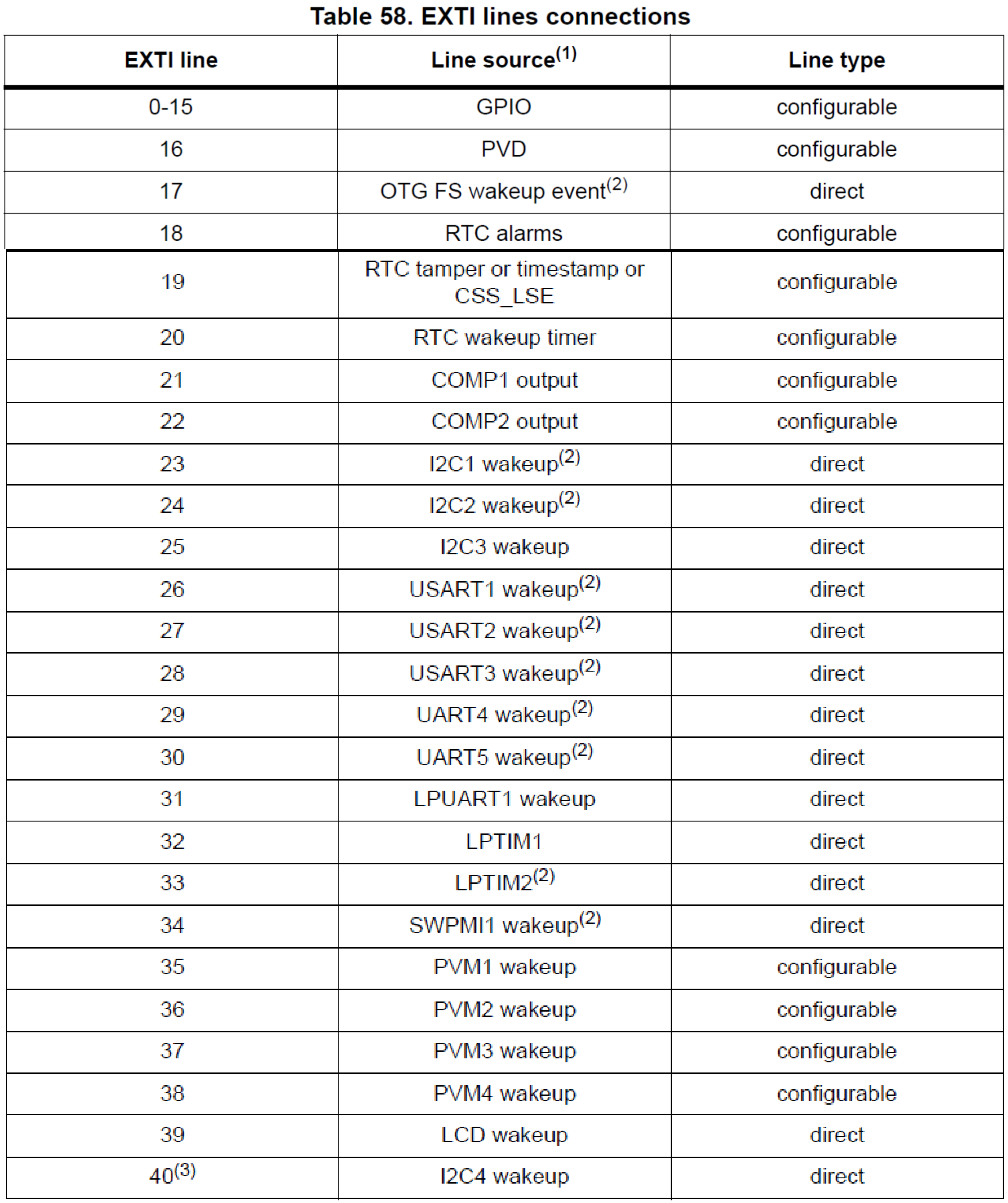


Table 58. EXTI lines connections

EXTI line	Line source[1]	Line type
0-15	GPIO	configurable
16	PVD	configurable
17	OTG FS wakeup event[2]	direct
18	RTC alarms	configurable
19	RTC tamper or timestamp or CSS_LSE	configurable
20	RTC wakeup timer	configurable
21	COMP1 output	configurable
22	COMP2 output	configurable
23	I2C1 wakeup[2]	direct
24	I2C2 wakeup[2]	direct
25	I2C3 wakeup	direct
26	USART1 wakeup[2]	direct
27	USART2 wakeup[2]	direct
28	USART3 wakeup[2]	direct
29	UART4 wakeup[2]	direct
30	UART5 wakeup[2]	direct
31	LPUART1 wakeup	direct
32	LPTIM1	direct
33	LPTIM2[2]	direct
34	SWPMI1 wakeup[2]	direct
35	PVM1 wakeup	configurable
36	PVM2 wakeup	configurable
37	PVM3 wakeup	configurable
38	PVM4 wakeup	configurable
39	LCD wakeup	direct
40[3]	I2C4 wakeup	direct

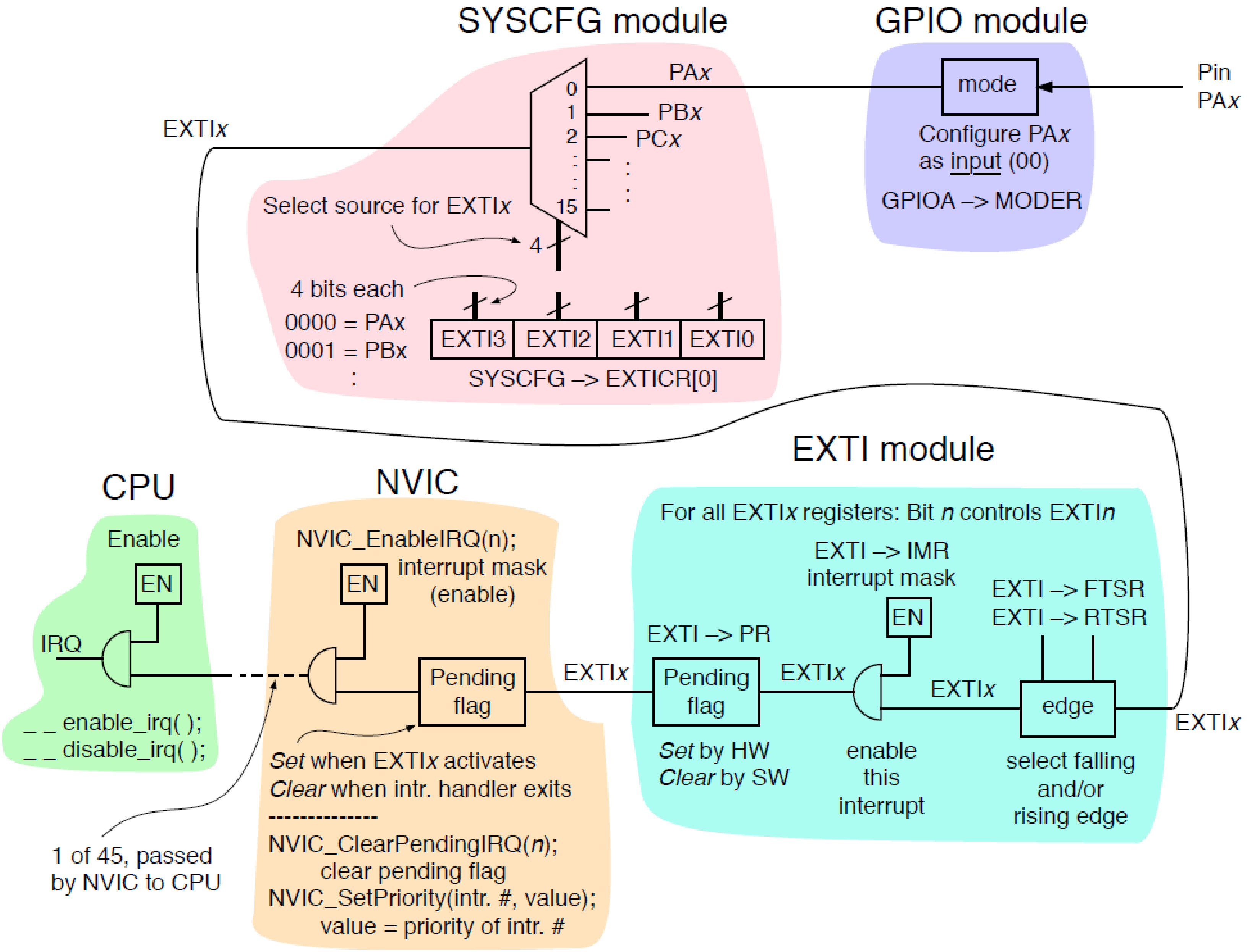
1. All the lines can wake up from the Stop 0 and Stop 1 modes. All the lines, except the ones mentioned above, can wake up from the Stop 2 mode.
2. This line source cannot wake up from the Stop 2 mode.

STM32L4 EXTI Registers

23 bits per register (configurable lines)

- EXTI_IMR – interrupt mask register
  - 0 masks (disables) the interrupt
  - 1 unmasks (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 EXTI*x*, *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 EXTIn
    - Configure GPIO pin as a digital input
    - Select the pin as the EXTIn 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*( );

(diagram on next slide)

EXTI example – accessing registers directly (in C)

```
#include "STM32L476xx.h"
/*----------------------------------------------------------------------------
	Intialize the GPIO and the external interrupt
*----------------------------------------------------------------------------*/
void Init_Switch(void){

	//Enable the clock for GPIO
	RCC->AHB2ENR |= 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);
}
```

System tick timer interrupts

- **SysTick Timer** is a 24-bit down counter
  - Interrupt on count from 1 -> 0
  - Count 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 & reload value:

```
#include "core_cm4.h"
SystemCoreClockUpdate();                     /* Get Core Clock Frequency   */
if (SysTick_Config(SystemCoreClock / 1000)) { /* SysTick 1 msec interrupts  */
   while (1);                                 /* Capture error             */
  }
```

Supervisor Call Instruction (SVC)

- Use a software-triggered interrupt to access system resources from O/S ("privileged operations")
- Interrupt vector SVC_Handler is defined in the 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)

ARM Compiler toolchain Compiler Reference: __svc

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
```