

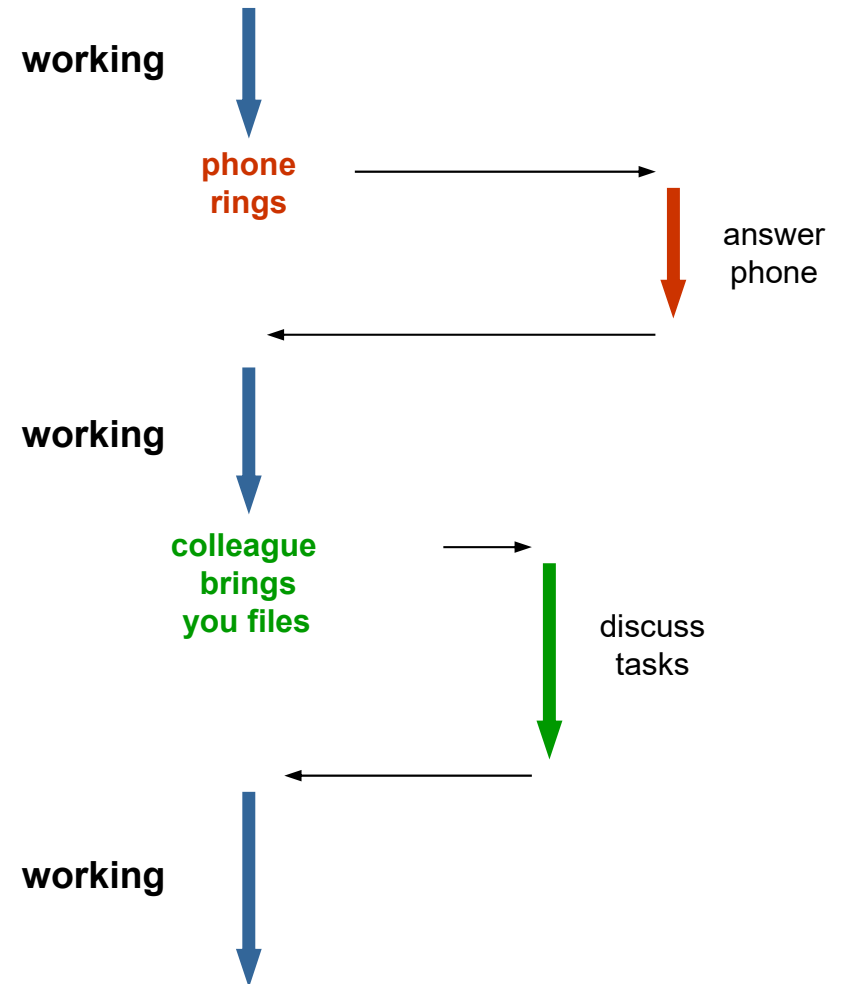
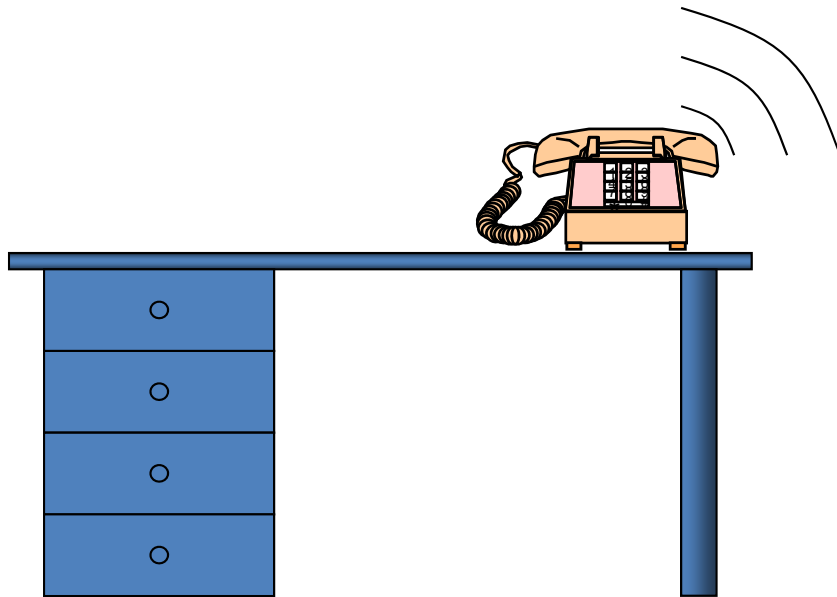
Exceptional Control Flow

Computer Engineering 1

**CT Team: A. Gieriet, J. Gruber, B. Koch, M. Loeser, M. Meli,
M. Rosenthal, M. Ostertag, A. Rüst, J. Scheier, T. Welti**

Motivation

■ You are working at your desk!



- **Computers work concurrently on several tasks**
 - Often wait until result is available, e.g. network request
 - Fill idle time with useful tasks → efficient use of processing power
 - Interrupts signal, that result of a request is available
 - Requires switching between programs
- **Unexpected events – exceptions**
 - Division by 0, unaligned accesses, etc.
 - Must be caught and corresponding actions need to be triggered
- **Time critical events with high priority**
 - Interrupt running program → process event

- **Polling**
- **Interrupt-Driven I/O**
- **Exceptions Cortex-M3/M4**
- **Interrupt-System Cortex-M3/M4**
- **Vector Table**
- **Storing the Context**
- **External Interrupt Pins**
- **Interrupt Control**
- **Nested Exceptions**
- **Special Interrupt Situations**
- **CMSIS Functions**
- **Data Consistency**

The lecture does not cover all the features of Cortex-M3/4 exception processing, in particular the following simplifications apply:

- No distinction between handler mode and thread mode
- No distinction between preempt priority and sub-priority fields
- Tail chaining and late arrival are not covered
- Instructions that are abandoned and restarted/resumed after interrupt service are not covered (e.g. LDM)

Learning Objectives

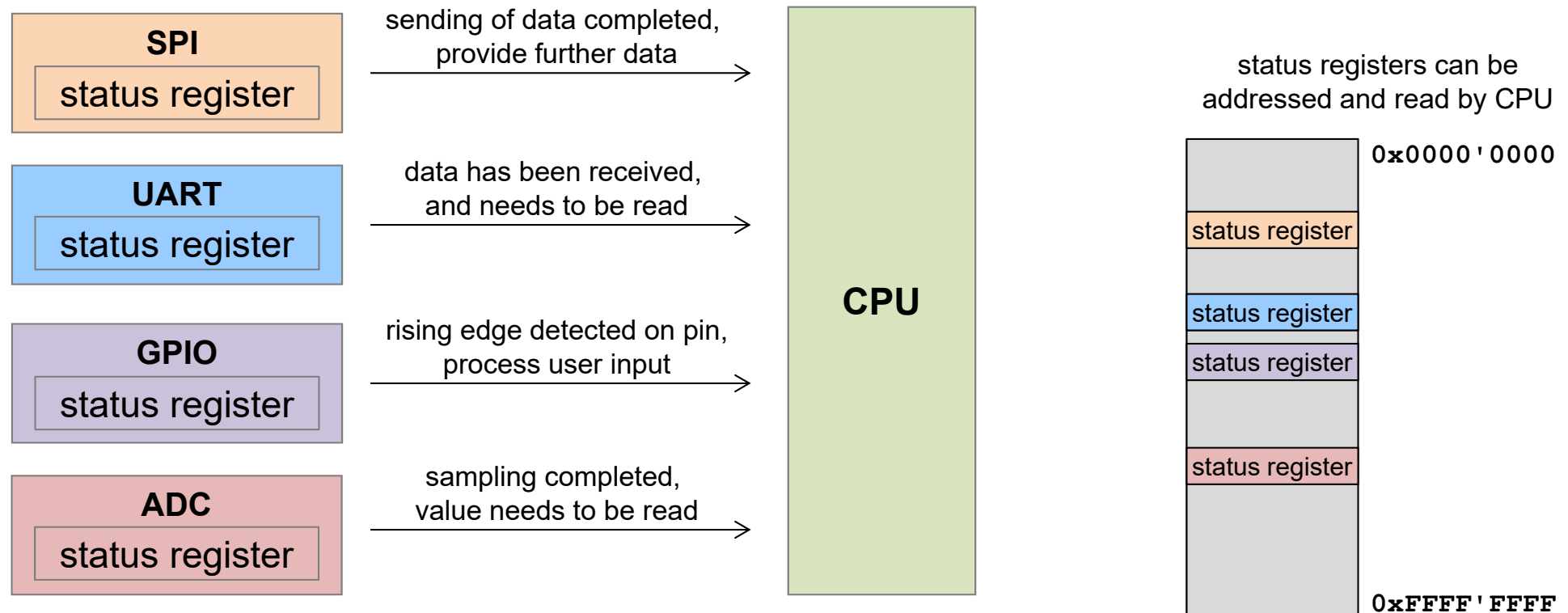
At the end of this lesson you will be able

- to explain advantages and disadvantages of polling and interrupt-driven I/O
- to distinguish the different types of exceptions on a Cortex-M3/M4
- to explain how the Cortex-M3/M4 recognizes and processes exceptions
- to explain the vector table of the Cortex-M3/M4
- to understand the basic functionality of the Nested Vectored Interrupt Controller (NVIC)
 - to enable and disable interrupts
 - to set and clear interrupts by software
 - to prioritize exceptions
 - to know how programmed priorities influence preemption of service routines
 - to explain how simultaneously pending interrupts are processed
- to implement a simple interrupt service routine in Cortex-M assembly
- to explain potential data consistency issues due to interrupts and to give potential examples

CPU Has to Act on Events

■ Peripherals signal events to CPU

- Something happened that requires a CPU service → call a subroutine

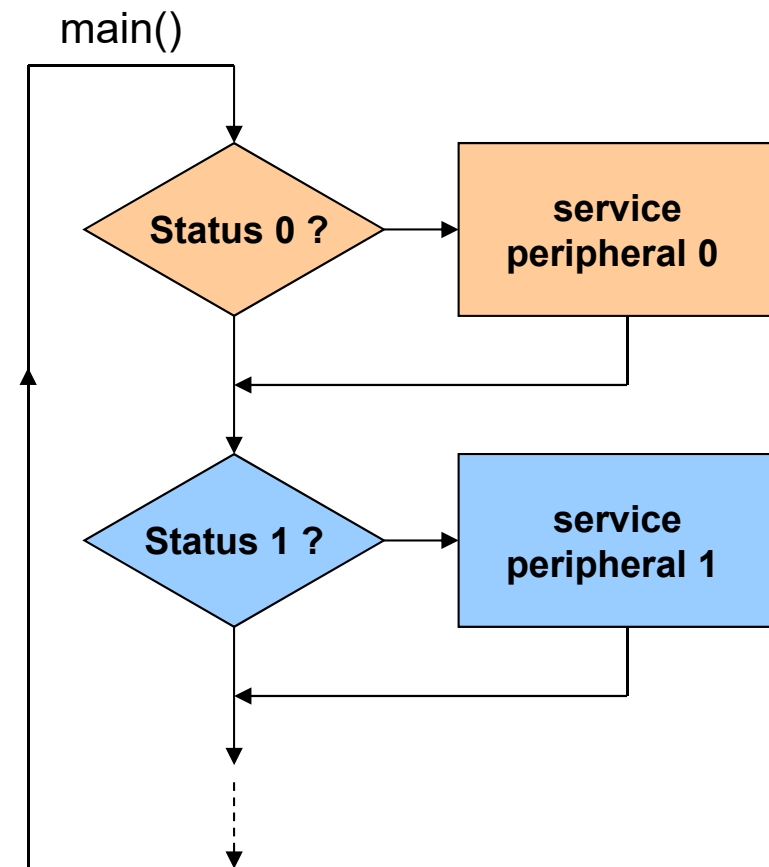


SPI Serial Peripheral Interface
UART Universal Asynchronous Receiver Transmitter
GPIO General Purpose Input Output
ADC Analog-Digital Converter

■ Periodic Query of Status Information

- Reading of status registers in loop
- Synchronous with main program
- Advantages
 - Simple and straightforward
 - Implicit synchronization
 - Deterministic
 - No additional interrupt logic required
- Disadvantages
 - Busy wait → wastes CPU time
 - Reduced throughput
 - Long reaction times

in case of many I/O devices or
if the CPU is working on other tasks



1) to poll → abfragen

■ Architecture

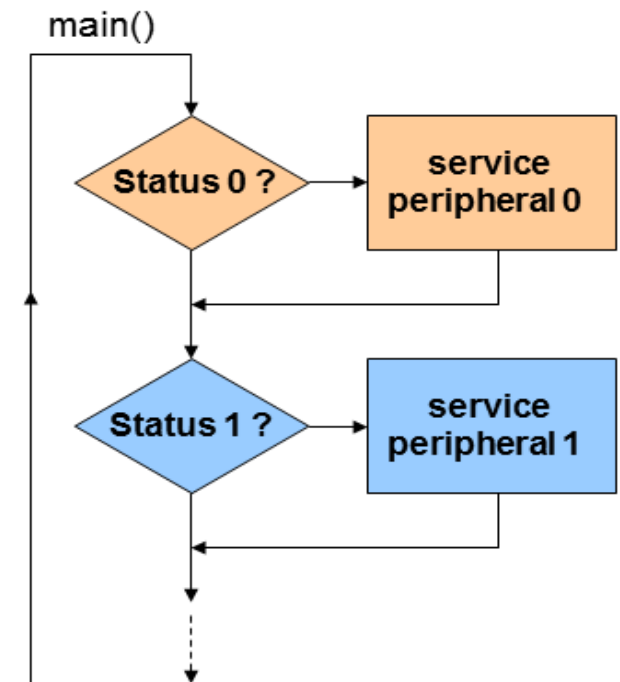
```
while(1)
{
    if ( read_byte(ADDR_BUTTON_A) ) {
        execute_task_A();
    }

    if ( read_byte(ADDR_BUTTON_B) ) {
        execute_task_B();
    }

    if ( read_byte(ADDR_BUTTON_C) ) {
        execute_task_C();
    }

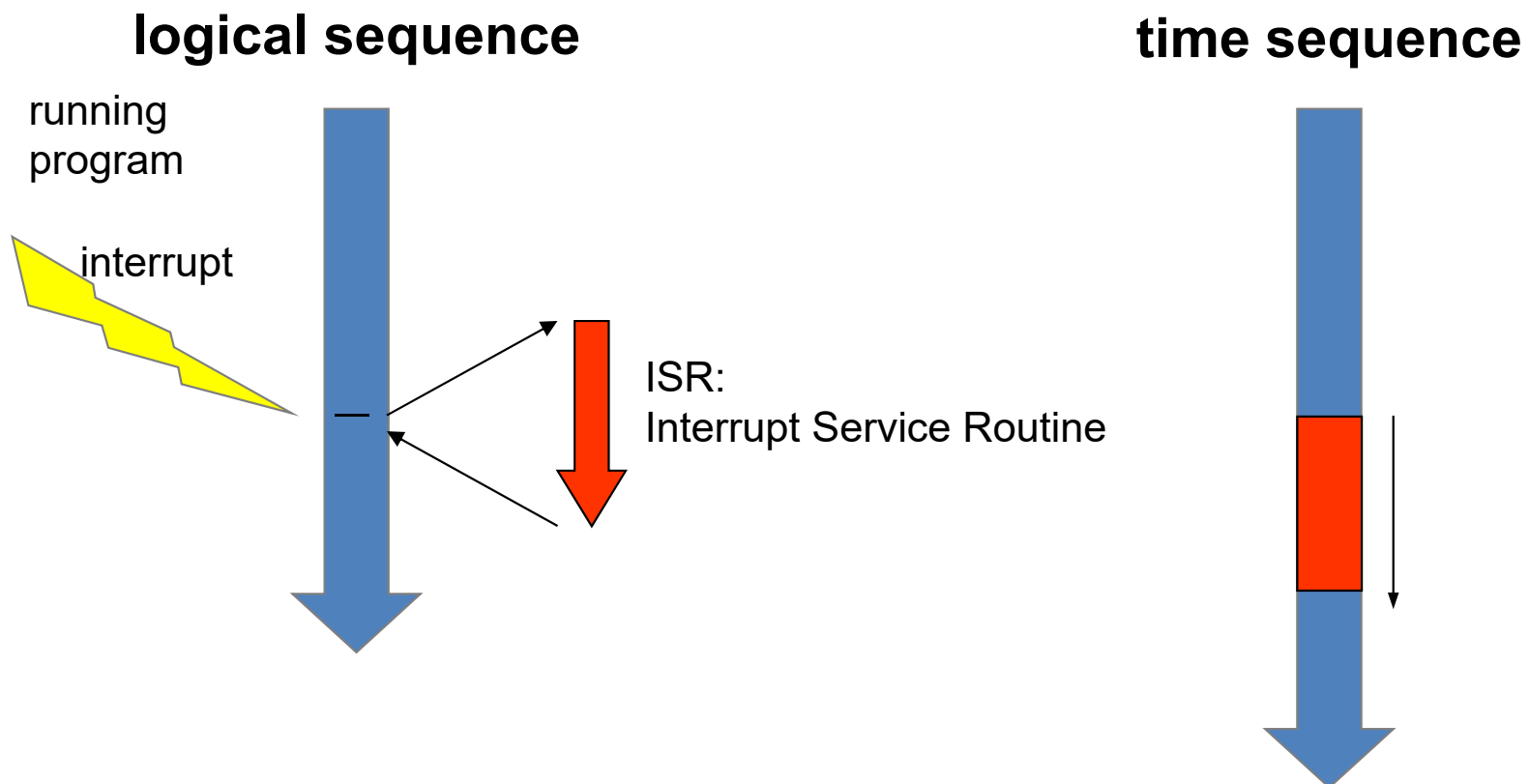
    ...

}
}
```



■ Alternative to polling

Interrupt: Sudden change of program flow due to an event



Interrupt-Driven I/O

■ Main program

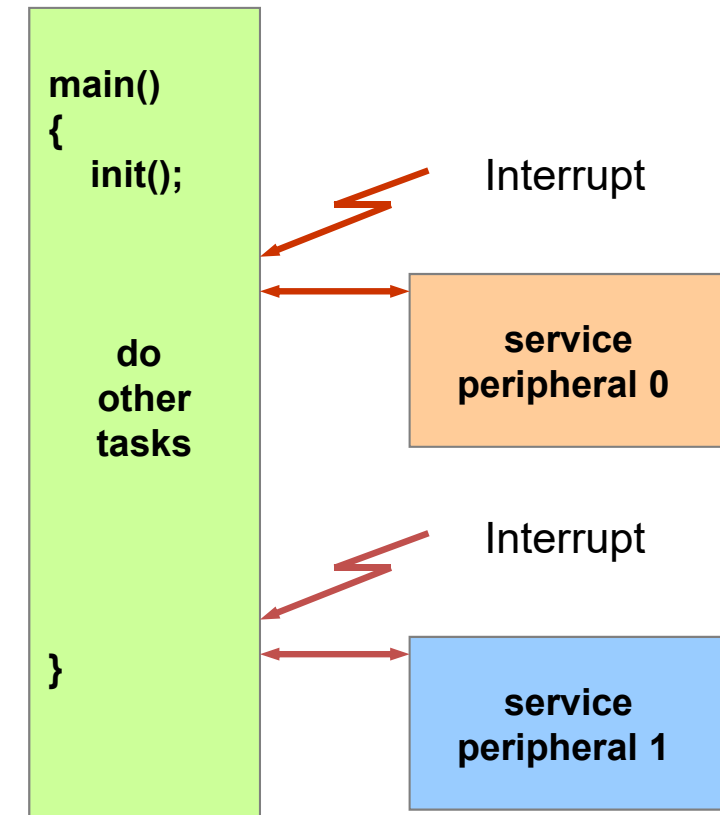
- Initializes peripherals
- Afterwards it executes other tasks
- Peripherals signal when they require software attention (phone call analogy)
- Events interrupt program execution

■ Advantage

- No busy wait → better use of CPU time
- Short reaction times

■ Disadvantages

- No synchronization (between main program and ISRs)
- Difficult debugging



■ Interrupt sources: IRQ0 – IRQ239

- Peripherals¹⁾ signal to CPU that an event needs immediate attention
- Can alternatively be generated by software request
- Asynchronous to instruction execution

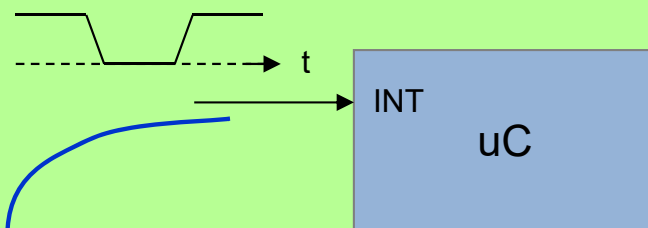
■ System exceptions

- Reset
 - Restart of processor
- NMI – Non-maskable Interrupt
 - Condition that cannot be ignored, e.g. a critical hardware error
- Faults
 - E.g. undefined instructions, unaligned accesses, etc.
- System Level Calls
 - Operating system (OS) calls – Instruction SVC and PendSV

¹⁾ GPIO (general purpose input/output), timer, serial interfaces, etc

■ Examples for Exceptions

interrupts → IRQ0 – IRQ239



```
isr_x  PUSH { ... }  
...  
      POP { ... }  
      BX LR
```

system exceptions

```
...  
SVC    #0x34  
...  
LDR     R0, =0x20001111  
LDR     R1, [R0]  
...
```

SVC
supervisor call

usage fault
e.g. unaligned
memory access

```
isr_y  PUSH { ... }  
...  
      POP { ... }  
      BX LR
```

```
isr_z  PUSH { ... }  
...  
      POP { ... }  
      BX LR
```

Exceptions Cortex-M3/M4

■ System Exceptions

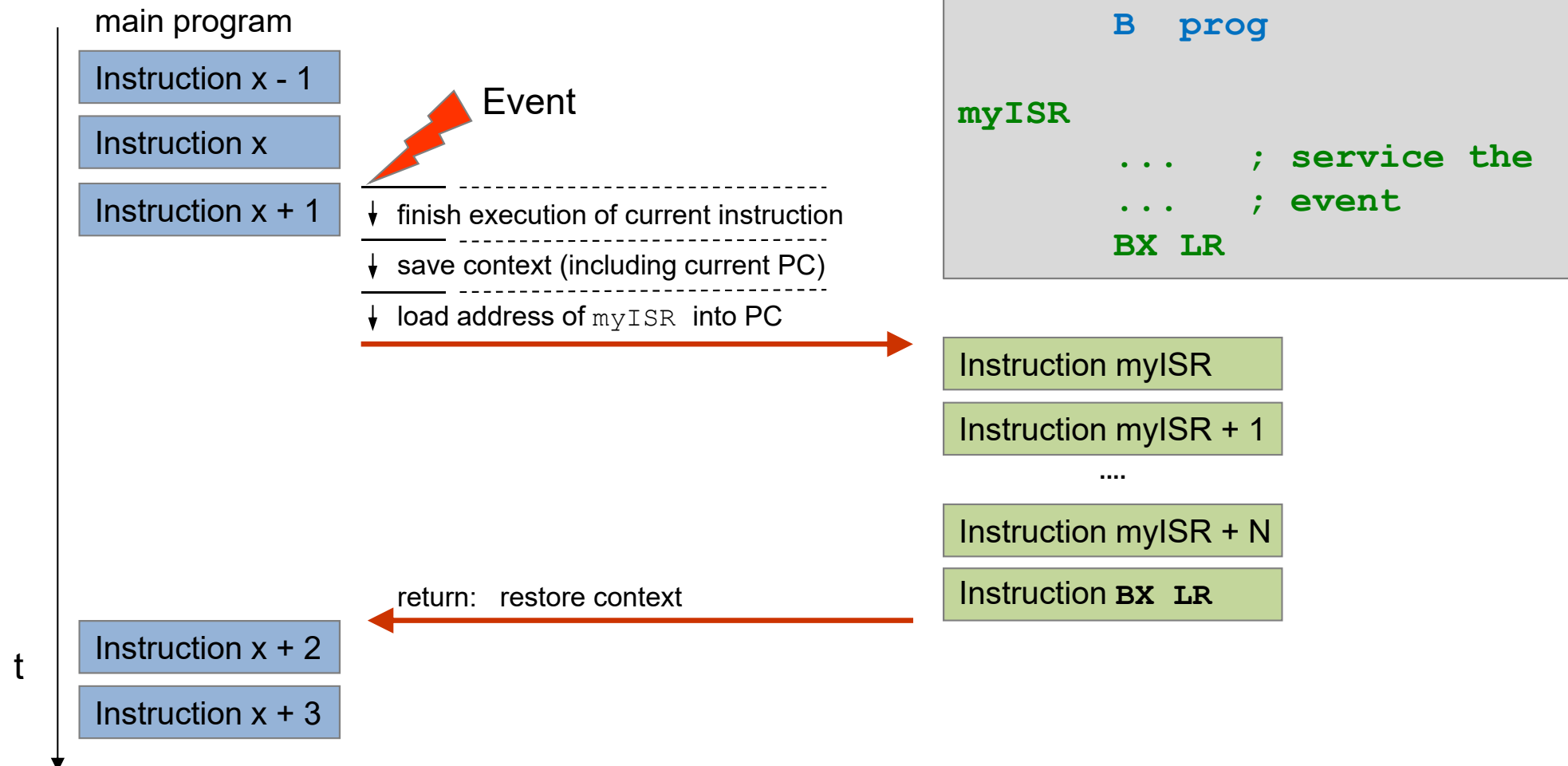
Exception Number	Exception Type	Priority	Description
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt (external NMI input)
3	Hard Fault	−1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage Fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus Fault	Programmable	Bus error; occurs when AHB interface receives an error response from a bus slave (also called <i>prefetch abort</i> if it is an instruction fetch or <i>data abort</i> if it is a data access)
6	Usage Fault	Programmable	Exceptions due to program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7–10	Reserved	NA	–
11	SVCall	Programmable	System Service call
12	Debug Monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	–
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System Tick Timer

source: "The definitive Guide to the Cortex-M3"

Exceptions Cortex-M3/M4

■ Interrupts: Event calls ISR

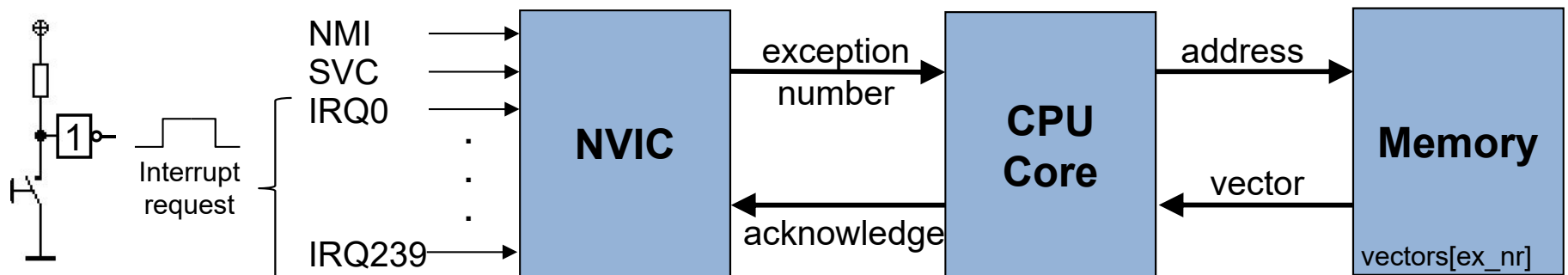
- Change of program flow



Interrupt-System Cortex-M3/M4

■ Nested Vectored Interrupt Controller (NVIC)

- 240 sources can trigger exception → high level signal on IRQx
- Forwards respective exception number to CPU



■ CPU

- Calculates vector table address based on exception number
- Uses address to read vector from memory
- Stores context on stack
- Loads vector into PC → branch to ISR

Simultaneous exceptions from different sources will be covered later

Vector Table (Cortex-M3/M4)

■ Which ISR Shall the Processor Call?

- Each exception has a different ISR

Memory Addr.	31	0	Exception Nr.	
0x0000'0000	Top of Stack	0		System Exceptions 1 – 15
0x0000'0004	Reset	1		
0x0000'0008	NMI	2		
0x0000'000C	Hard Fault	3		
.		
0x0000'002C	SVC	11		
.		
0x0000'0038	PendSV	14		
0x0000'003C	SysTick	15		
0x0000'0040	IRQ0	16		Interrupts 0 – 239 IRQn → Exception Nr. (n + 16)
0x0000'0044	IRQ1	17		
.		
0x0000'03FC	IRQ239	255		

Example:
IRQ3 → Exception Nr. 19

Vector Table (Cortex-M3/M4)

■ Initialization

```
; Vector Table Mapped to Address 0 at Reset
        AREA  RESET, DATA, READONLY

0  __Vectors    DCD    __initial_sp        ; Top of Stack
1              DCD    Reset_Handler       ; Reset Handler
2              DCD    NMI_Handler         ; NMI Handler
3              DCD    HardFault_Handler   ; Hard Fault Handler
...          DCD    ...
...          DCD    ...

        ; Interrupts
16         DCD    IRQ0_Handler            ; ISR for IRQ0
17         DCD    IRQ1_Handler            ; ISR for IRQ1
...         DCD    ...
```

System Exceptions
15 vectors

Interrupts

```
        AREA  SOURCE_CODE, CODE, READONLY

IRQ0_Handler  PUSH { ... }

                ...                ; interrupt service

                POP { ... }
                BX  LR
```

Storing the Context

■ Interrupt event can take place at any time

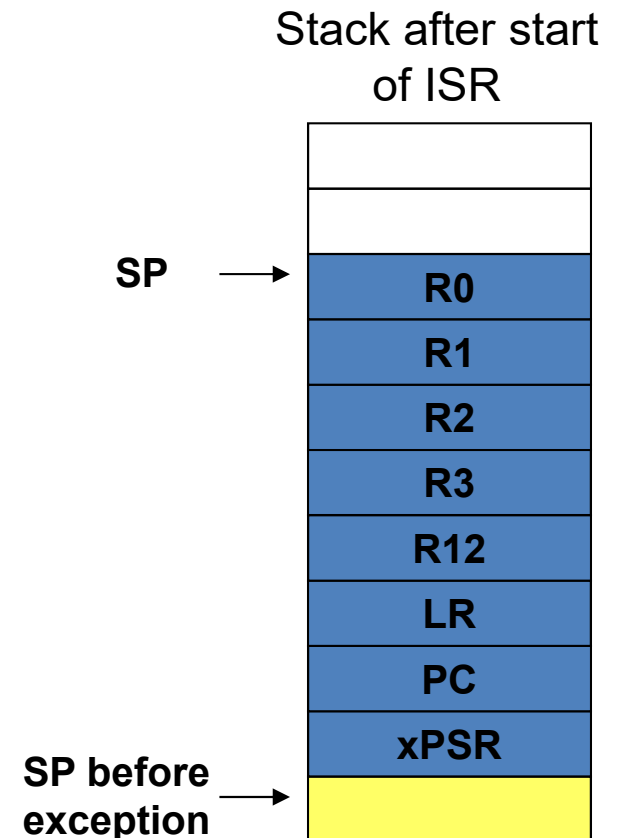
- E.g. between **TST** and **BEQ** instructions
 - ISR call requires automatic save of flags and caller saved registers

■ ISR Call

- Stores xPSR, PC, LR, R12, R0 – R3 on stack
- Stores EXC_RETURN¹⁾ to LR

■ ISR Return

- Use **BX LR** or **POP {..., PC}**²⁾
- Loading EXC_RETURN¹⁾ into PC
 - restores R0 – R3, R12, LR, PC and xPSR from stack



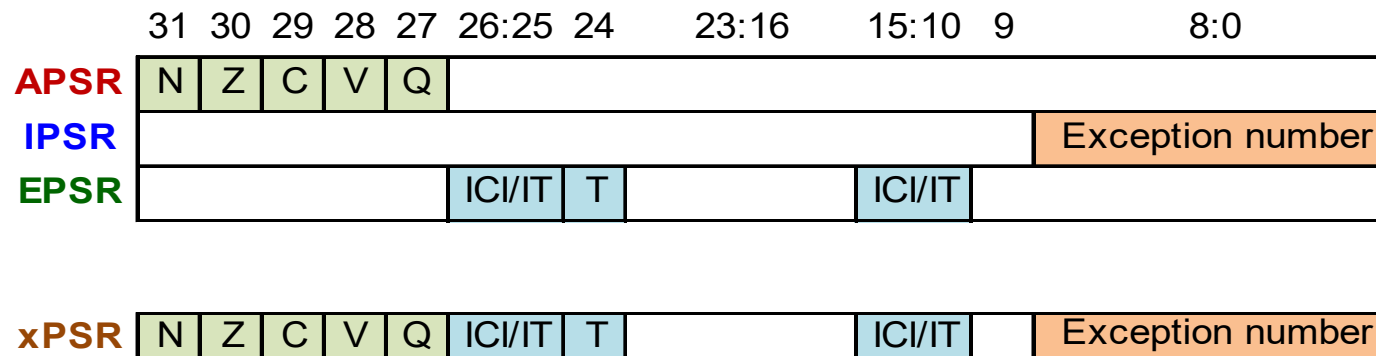
¹⁾ EXC_RETURN = 0xFFFF'FFF9

²⁾ if LR has been saved with PUSH {..., LR} in ISR

Storing the Context

■ Program Status Registers (PSRs)

- **APSR** Application Program Status Register
- **IPSR** Interrupt Program Status Register
- **EPSR** Execution Program Status Register

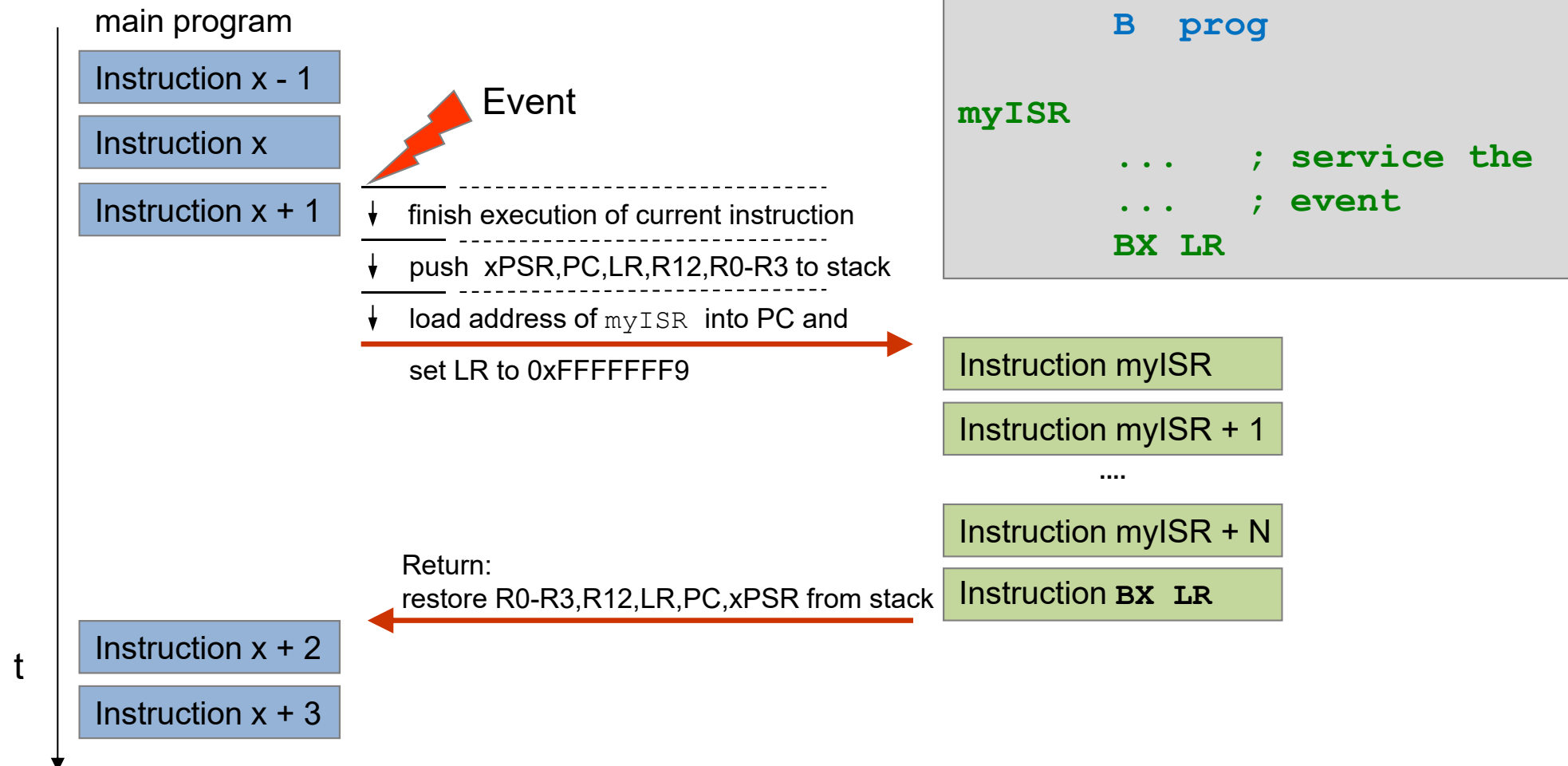


- **xPSR** Combination of all three PSRs

Exception number indicates which exception the processor is handling
ICI/IT Bits used in case of Interrupt-Continuable Instructions e.g. LDM/STM
or conditional execution of instructions (IF-THEN)
T Thumb mode. Always one

Storing the Context

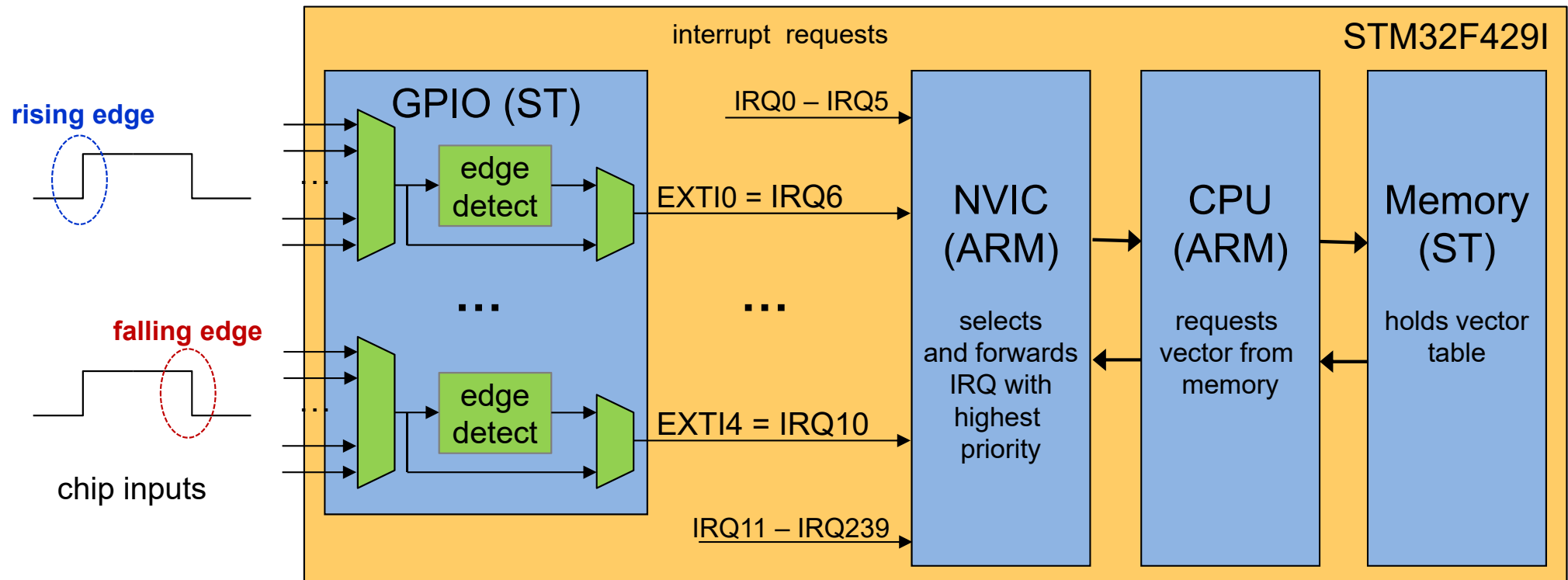
■ Asynchronous event calls ISR



External Interrupt Pins

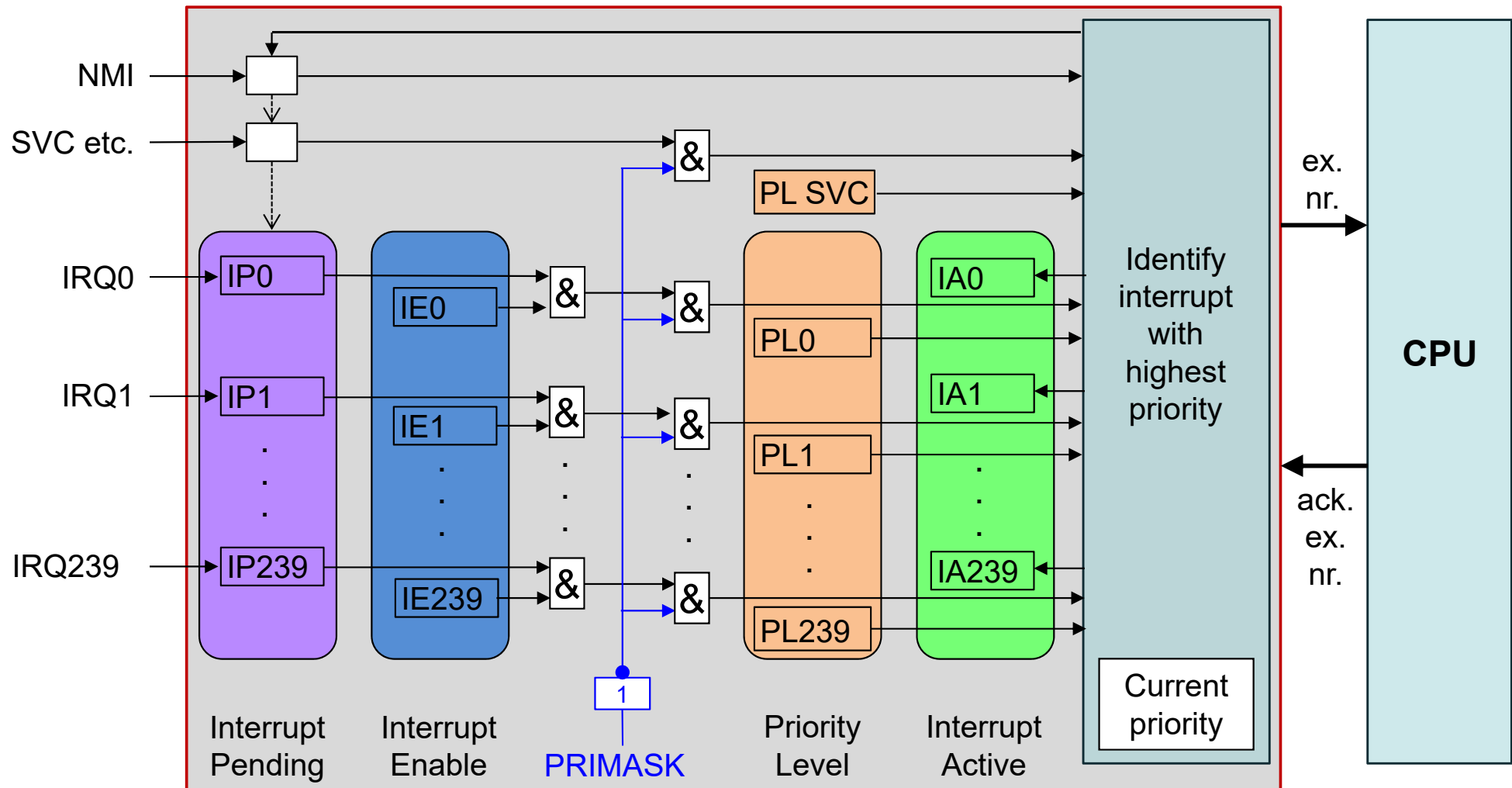
■ Chip Vendor (ST) adds GPIO logic

- EXTI0 through EXTI4 connected to IRQ6 through IRQ10
 - IRQ0 – IRQ5 / IRQ11 – IRQ239 used for other sources e.g. SPI, UART, ADC
- Select chip input for each EXTIx line
- Select level or edge



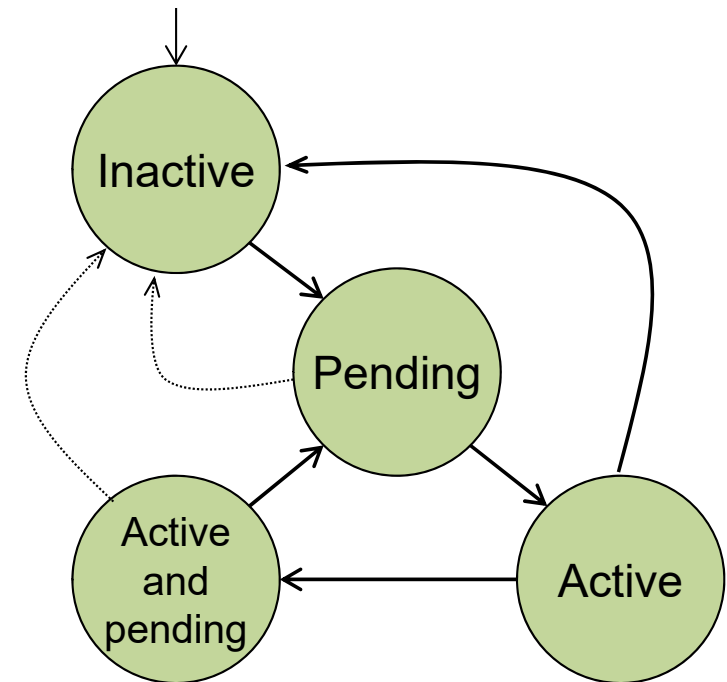
Interrupt Control

■ Nested Vectored Interrupt Controller (NVIC)

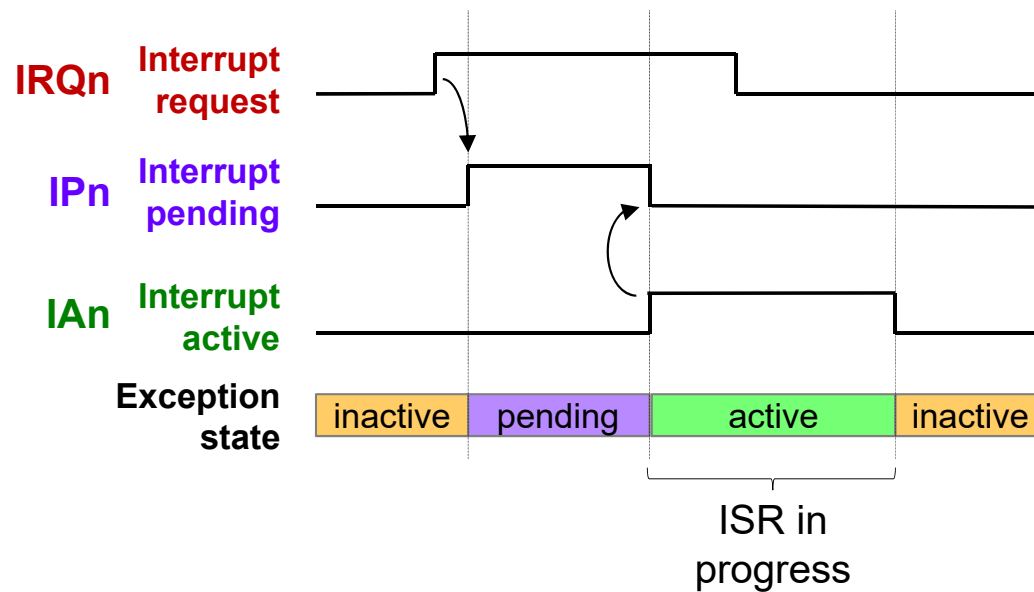


■ Exception States (for each interrupt source, e.g. IRQ17)

- Inactive
 - Exception is not active and not pending
- Pending
 - Exception is waiting to be serviced by CPU
 - E.g. an interrupt event occurred ($IRQn = 1$) but interrupts are disabled (PRIMASK)
- Active
 - Exception is being serviced by the CPU but has not completed
- Active and pending
 - Exception is being serviced by the CPU and there is a pending exception from the same source



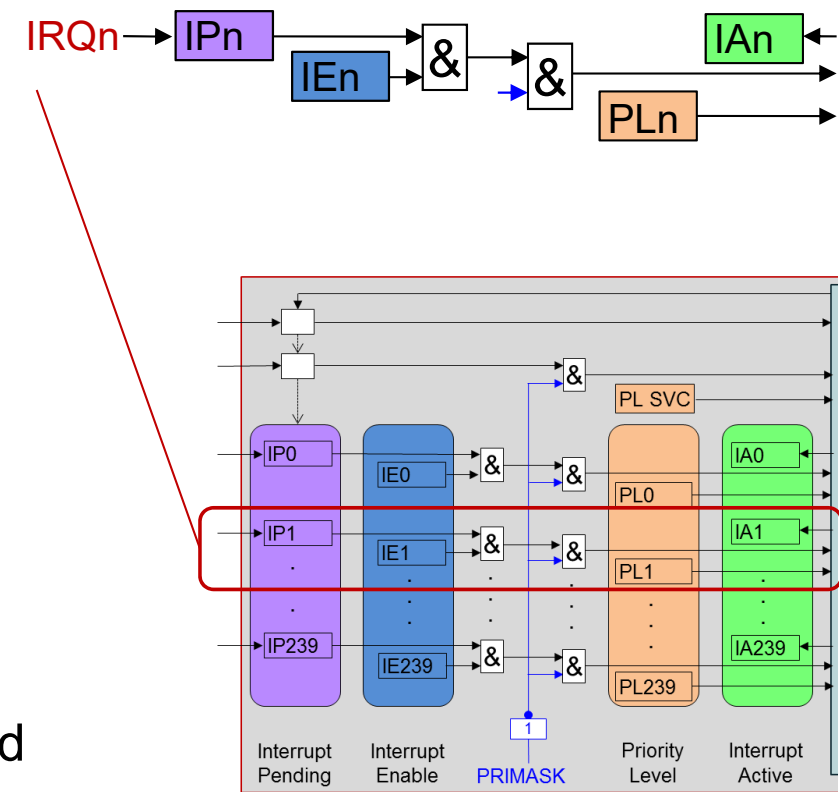
■ IRQ Inputs and Pending Behavior



- High level on IRQn sets pending bit (IPn)
- Active Bit (IAn)
 - Set as soon as exception is being serviced
 - Resets pending bit

¹⁾ IRQn is set by hardware and usually reset by software

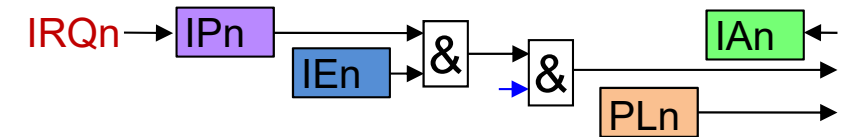
Same logic exists for each interrupt $n = 0 \dots 239$



Interrupt Control

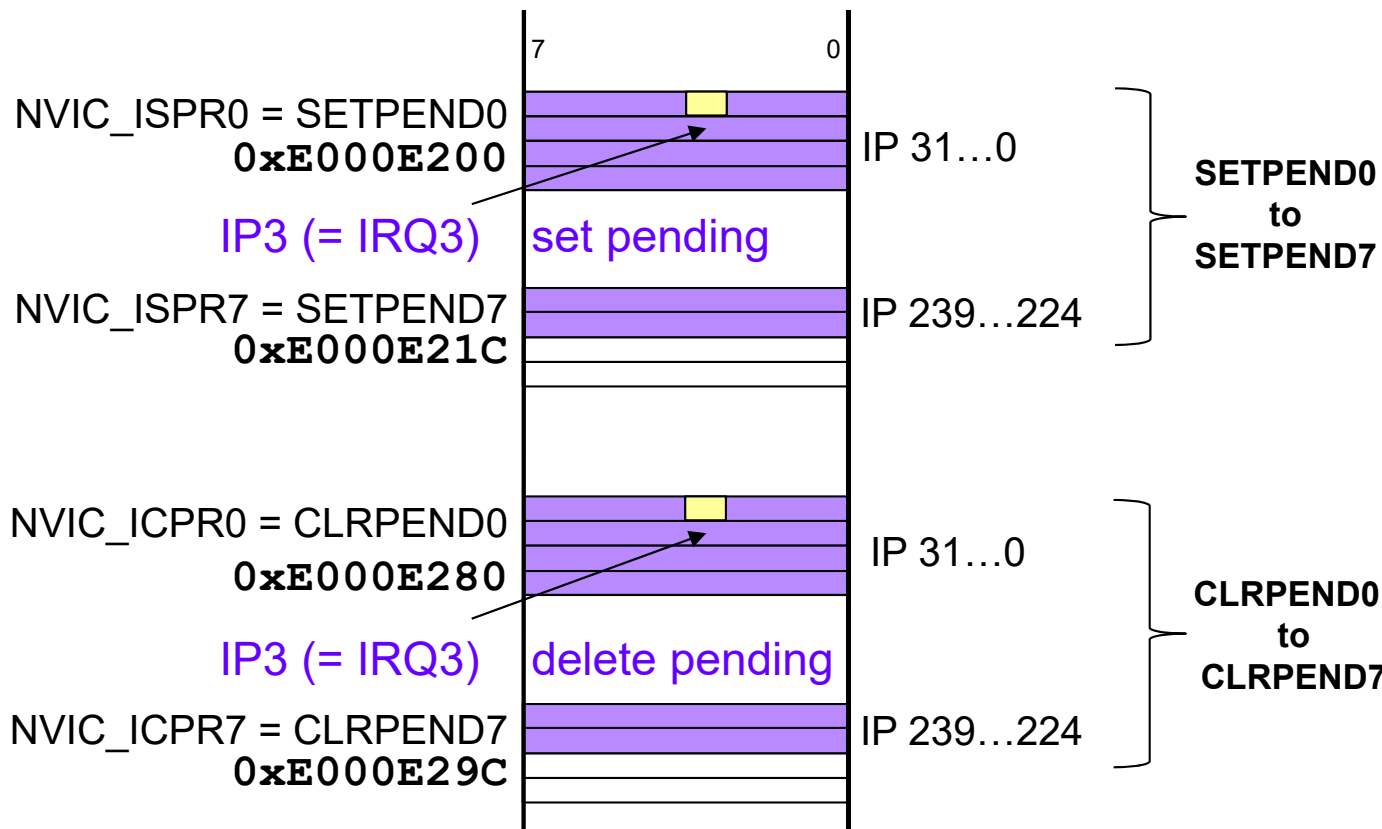
■ Interrupt Pending Registers

- Trigger hardware interrupt by software
- Cancel a pending interrupt



→ set pending bit

→ clear pending bit



Trigger IRQ3 by Software

```

SETPEND0 EQU 0xE000E200
...

LDR R0,=SETPEND0
MOVS R1,#0x08
STR R1,[R0]
  
```

Delete Pending IRQ3

```

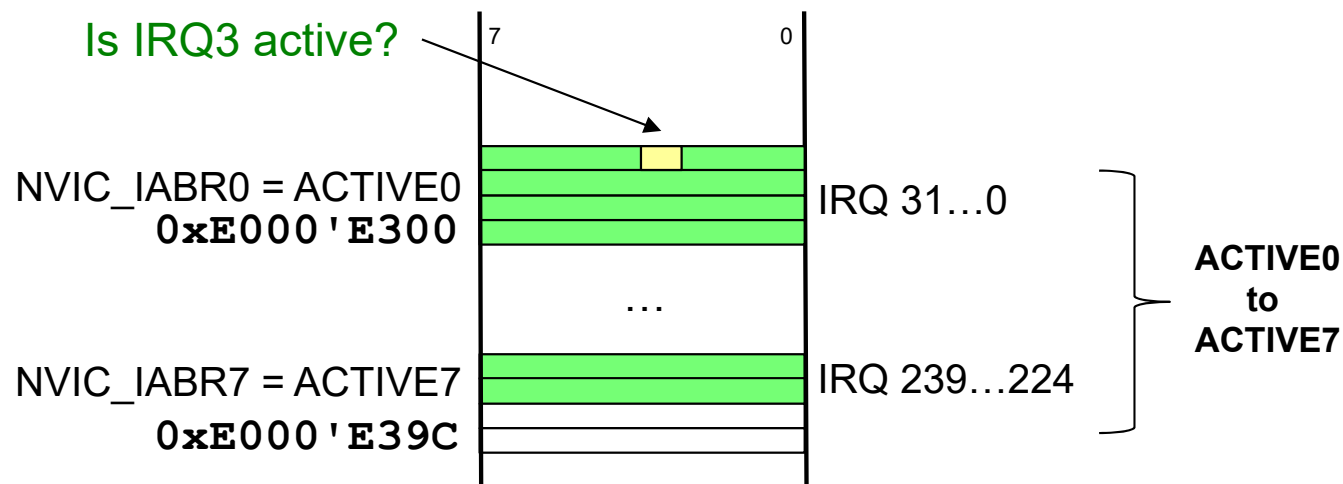
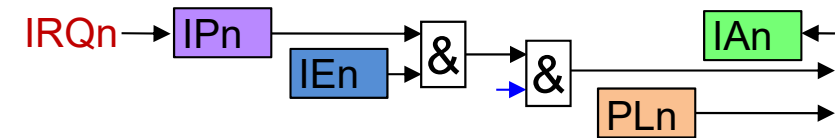
CLRPEND0 EQU 0xE000E280
...

LDR R0,=CLRPEND0
MOVS R1,#0x08
STR R1,[R0]
  
```

ARM and ST use different names for the same register

■ Interrupt Active Status Registers

- Read-only
- Corresponding bit is set when ISR starts
- Corresponding bit is cleared when interrupt return is executed



Test if IRQ3 is active

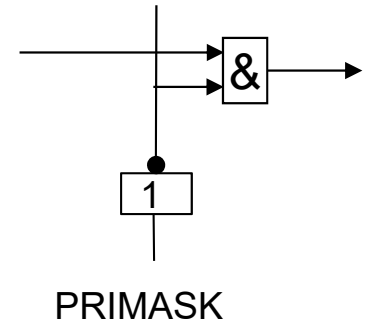
```

ACTIVE0 EQU 0xE000E300
...

LDR R0, =ACTIVE0
MOVS R1, #0x08
LDR R2, [R0]
TST R1, R2
BEQ ...
  
```

■ General Masking of Interrupts

- PRIMASK
 - Single bit controlling all maskable interrupts



- Disable set PRIMASK
- Enable clear PRIMASK

Assembly

```
CPSID1) i  
CPSIE1) i
```

C

```
__disable_irq();  
__enable_irq();
```

- On reset PRIMASK = 0 → enabled

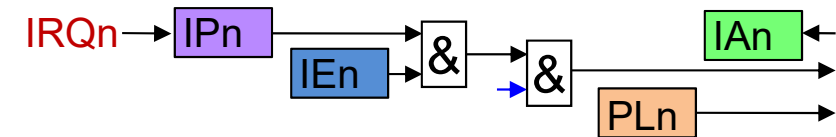
■ Non-maskable Interrupt (NMI)

- Power-fail, emergency button, watchdog, ...

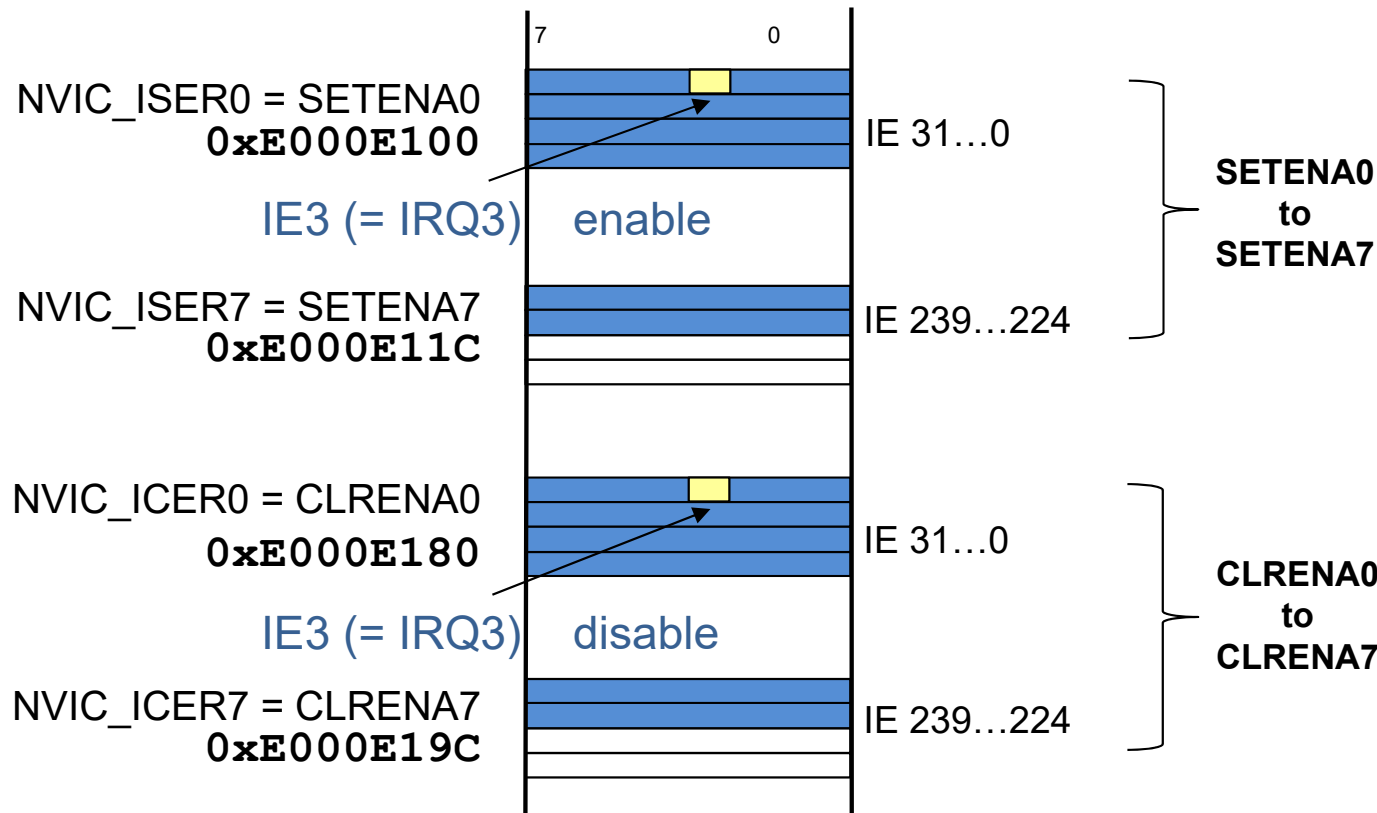
¹ CPSIE/D = Change Processor State Interrupt Enable / Disable

■ Interrupt Enable Registers

- Individual masking of interrupt sources
 - IEn cleared pending bit not forwarded
 - IEn set interrupt enabled



CLRENA and SETENA registers can be read by software and will return the settings of the IE bits



Enable IRQ3

```
SETENA0 EQU    0xE000E100
...

LDR    R0, =SETENA0
MOVS   R1, #0x08
STR    R1, [R0]
```

Disable IRQ3

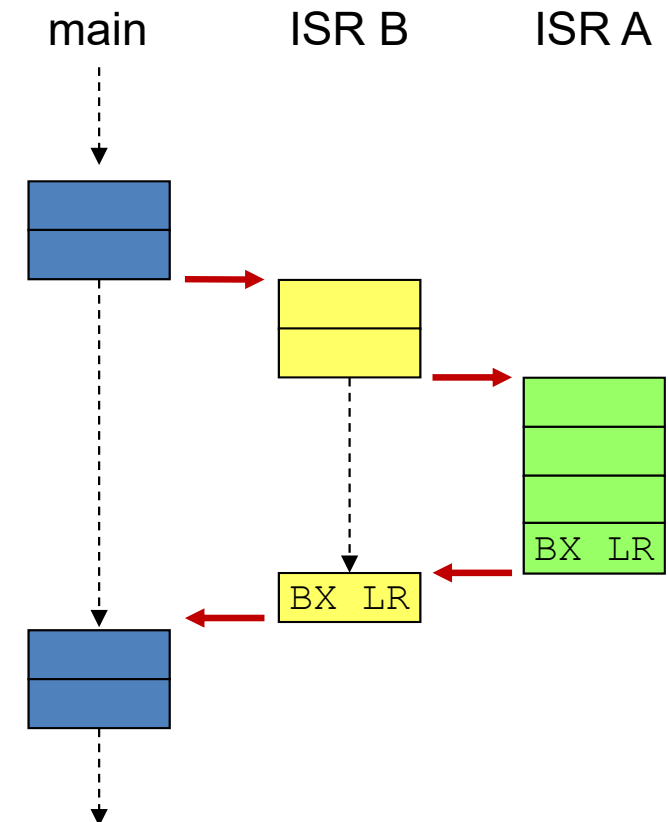
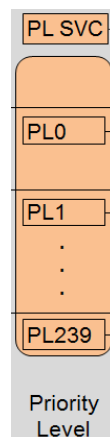
```
CLRENA0 EQU    0xE000E180
...

LDR    R0, =CLRENA0
MOVS   R1, #0x08
STR    R1, [R0]
```

Nested Exceptions

■ Preemption

- Service routine A temporarily interrupts service routine B
- Assigned priority level for each exception
 - Levels define whether A can preempt B
- Fixed priorities
 - Reset (-3), NMI (-2), hard fault (-1)
- All other priorities are programmable

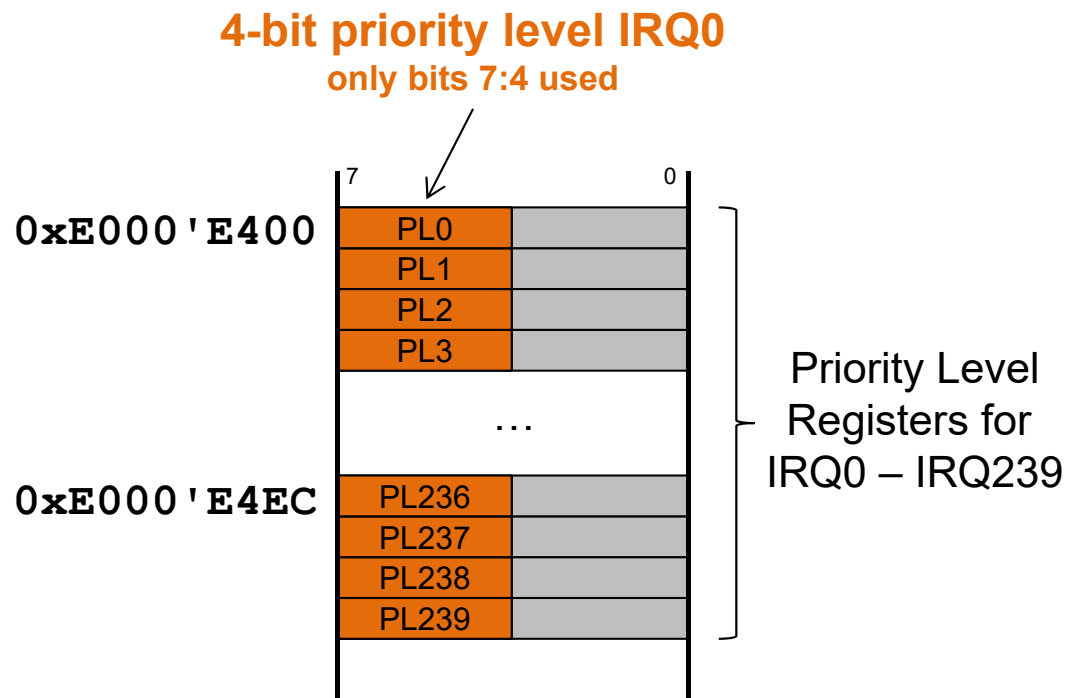


if A has been programmed to higher priority than B

Nested Exceptions

■ Priority Level Registers

- The lower a priority level, the greater the priority
- 4-bit priority level ¹⁾ 0x0 – 0xF



Set priority of IRQ3 to 5

```
PL_IRQ3 EQU 0xE000E403
...
LDR R0, =PL_REG_IRQ3
MOVS R1, #0x50
STRB R1, [R0]
```

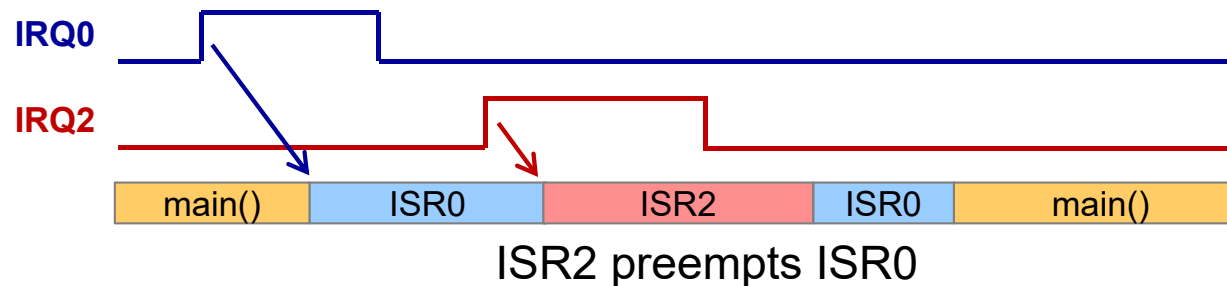
priority level registers for system exceptions
0xE000'ED18 – 0xE000'ED23

1) Cortex-M3/4 can handle up to 8 bits but STM uses only 4
Registers are called NVIC_IPRx on ST

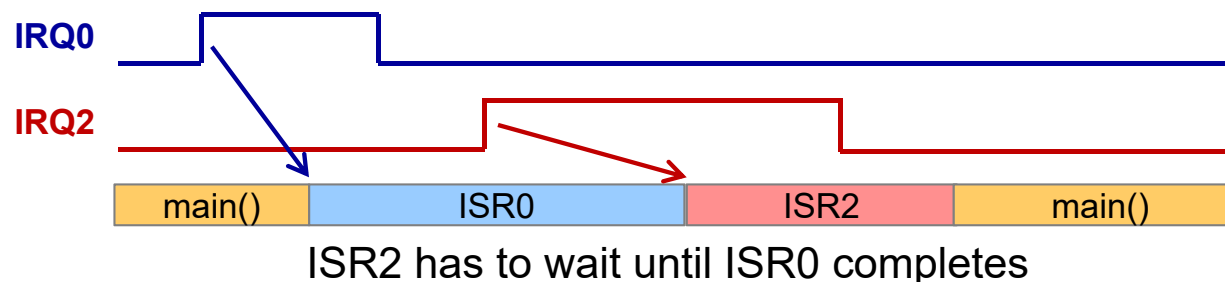
Nested Exceptions

■ IRQ request while other ISR is running

- ISR0 is executing (triggered by IRQ0)
- Interrupt request IRQ2 arrives
- $PL2 < PL0 \rightarrow$ IRQ2 has higher priority than IRQ0



- $PL2 \geq PL0 \rightarrow$ IRQ2 has lower or same priority than IRQ0



■ Two or more pending interrupts

- (1) IRQ requests arrive simultaneously
- (2) More than one request has been waiting for a higher priority ISR to complete
- NVIC will evaluate the priorities of the pending interrupts
 - Selects IRQx with highest priority i.e.
 - ▶ with lowest priority level value (PL)
 - ▶ with lowest number (in case of identical PL values)
 - Other interrupts remain pending

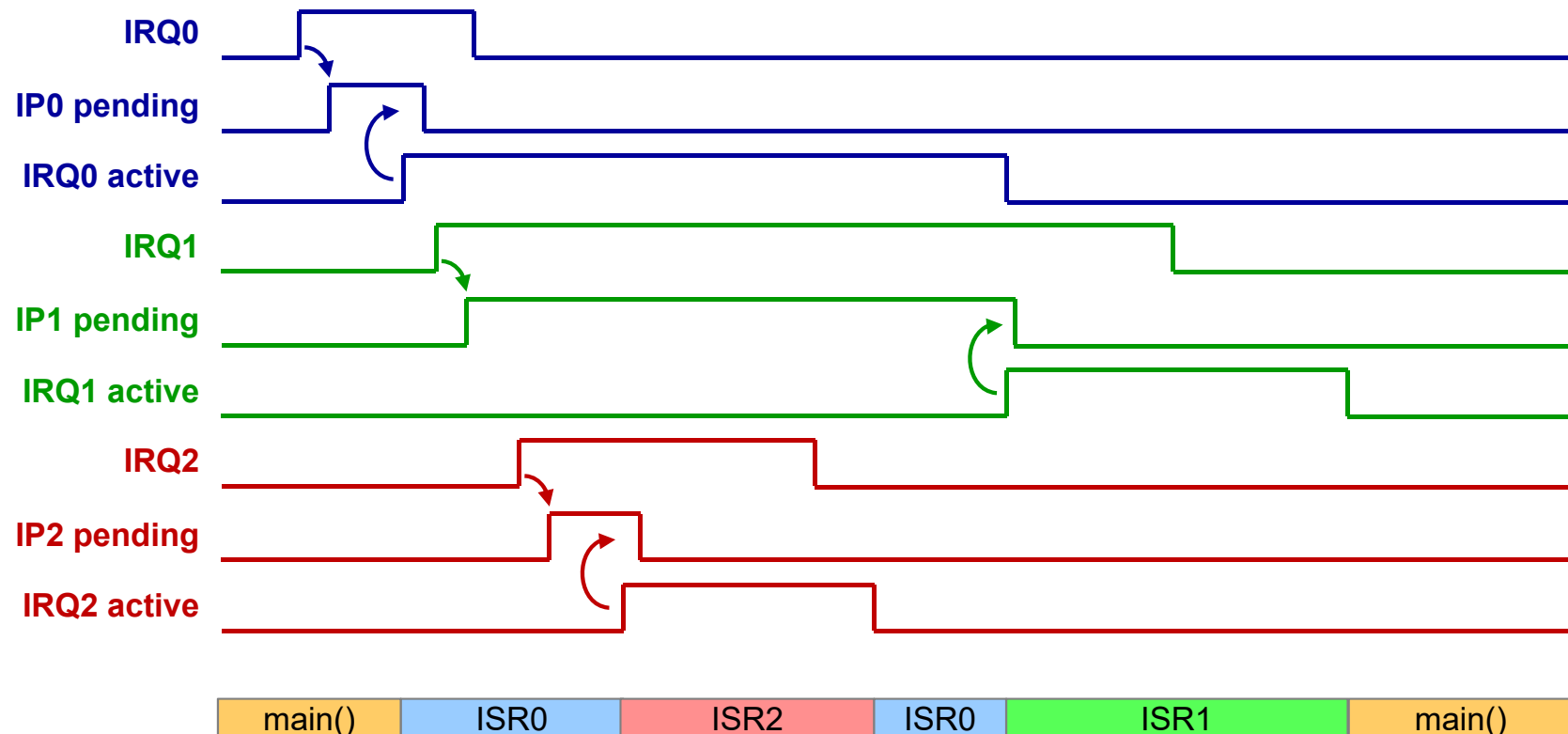
Nested Exceptions

■ Example Priorities

- ISR1 does not preempt ISR0
- ISR2 preempts ISR0

assuming

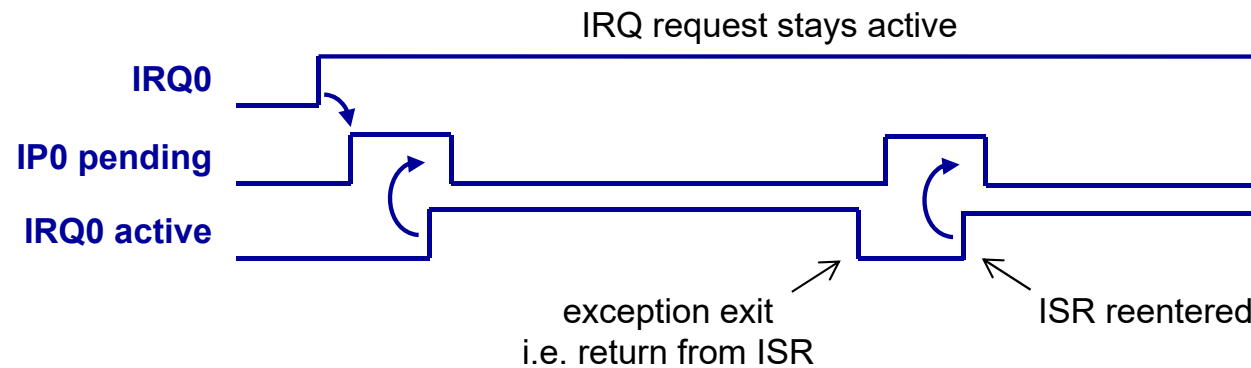
IRQ0	PL0 = 0x2	medium priority
IRQ1	PL1 = 0x3	lowest priority
IRQ2	PL2 = 0x1	highest priority



Special Interrupt Situations

■ IRQ request stays active

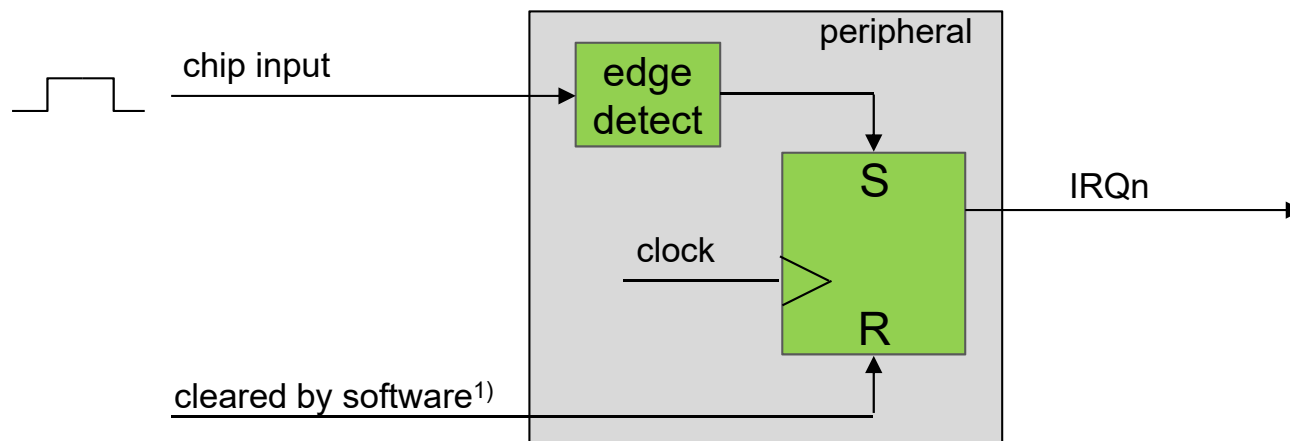
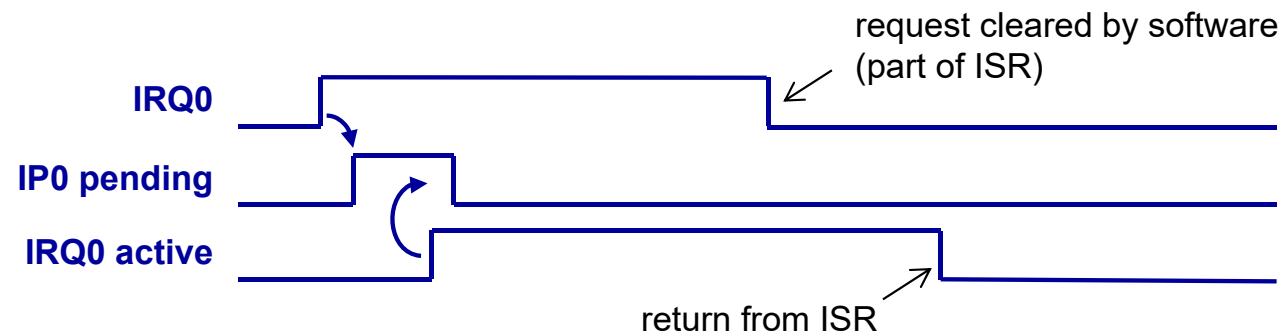
- Interrupt becomes pending again



Special Interrupt Situations

■ Common Configuration on Microcontrollers

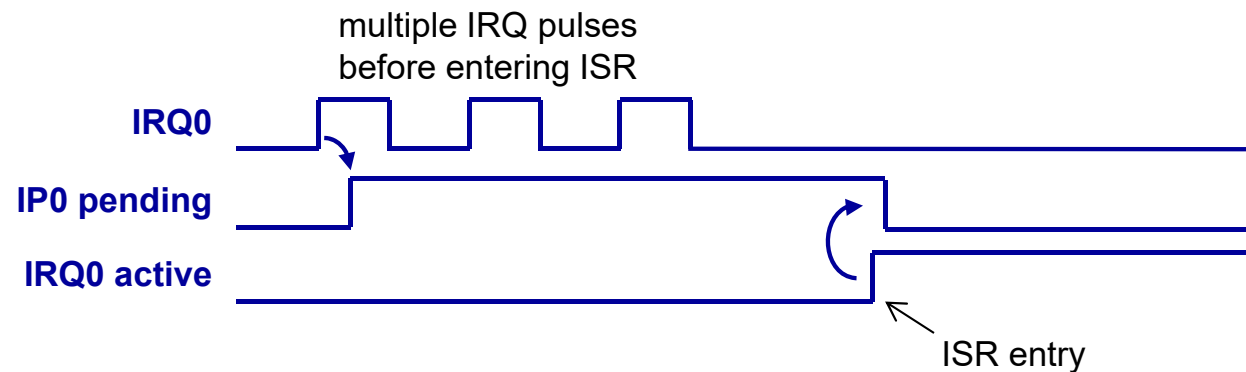
- IRQ Set by Hardware – Cleared by Software



1) The software clears the bit at the correct position at the defined register address

■ Multiple IRQ request pulses before entering ISR

- Treated as single interrupt
- Events are lost



■ What is CMSIS?

- Cortex Microcontroller Software Interface Standard
- Vendor-independent hardware abstraction layer for Cortex-M
- Defines generic tool interfaces
- ISO/IEC C code cannot directly access some Cortex-M3 instructions
 - intrinsic functions required

■ NVIC Control

<code>void NVIC_EnableIRQ (IRQn_t IRQn)</code>	Enable IRQn
<code>void NVIC_DisableIRQ (IRQn_t IRQn)</code>	Disable IRQn
<code>uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)</code>	Return true (IRQ-Number) if IRQn is pending
<code>void NVIC_SetPendingIRQ (IRQn_t IRQn)</code>	Set IRQn pending
<code>void NVIC_ClearPendingIRQ (IRQn_t IRQn)</code>	Clear IRQn pending status
<code>uint32_t NVIC_GetActive (IRQn_t IRQn)</code>	Return '1' if active bit of IRQn is set. '0' otherwise
<code>void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority)</code>	Set priority for IRQn
<code>uint32_t NVIC_GetPriority (IRQn_t IRQn)</code>	Read priority of IRQn
<code>void NVIC_SystemReset (void)</code>	Reset the system

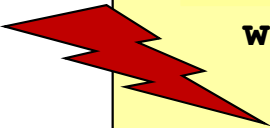
Data Consistency

■ Example

```
typedef struct {
    uint8_t minutes;
    uint8_t seconds;
} time_t;

static time_t time = { 0, 0 };

int main(void)
{
    while (1) {
        write_byte(ADDR_LED_7_0,
                   time.seconds);
        write_byte(ADDR_LED_15_8,
                   time.minutes);
    }
}
```



```
void IRQ1_Handler(void)
{
    time.seconds++;
    if (time.seconds > 59) {
        time.seconds = 0;
        time.minutes++;
    }
}
```

time = { 15, 59 }
1) Output 59 → LED_7_0
2) Interrupt → time = { 16, 0 }
3) Output 16 → LED_15_8
→ display: 16 59 !!!

- Data structure must not be changed during output
- Disable Interrupts during output
- **Multitasking problem**

■ Polling vs. Interrupt-driven I/O

■ Exceptions

- System exceptions: Reset, NMI, Faults, System Level Calls
- Interrupts IRQ0 – IRQ239

■ Nested Vectored Interrupt Controller

- Vector Table holds addresses of interrupt service routines (ISR)
- Save context including xPSR
- Masking of interrupts
- Pending and active bits
- Priority levels
- IRQn: Often hardware set; software reset

■ Data Consistency Issues