EECS 373
Introduction to Embedded System Design

Robert Dick
University of Michigan

Lecture 6: Declarations and Interrupts.

6 February 2024

On a bus stall, does PENABLE immediately go low?

- Immediately?
- Only after next rising clock edge?
- Not until later, when the stall is finished?

Outline

- **Assembler directives review**
- Volatile, const, and function pointers
- PWM
- Interrupts

Assembler directives

- Assembler directions don't necessary generate any instructions.
- Convenience to allow more modular and organized code, e.g., .equ .
  - Generates no code.
  - Acts like a proceprocessor macro (#define) in C.
- Provide information about data to include, e.g., .word .
- Tell assembler which symbols are global, e.g., .global .
- Indicate where in memory things (code and data) should sit, e.g., .text

Assembler directives example

```
@ “#define”-like
.equ    STACK_TOP, 0x20000800
.equ    SYSREG_SOFT_RST_CR, 0xE0042030

@ Make _start externally visible (to ld).
.global  _start

@ “a”: allocatable
@ %progbits: section contains data
@ .int_vector: section name. link.ld uses this.
.section .int_vector, "a", %progbits

_start:
        .word   STACK_TOP, main
```

Outline

- Assembler directives review
- **Volatile, const, and function pointers**
- Interrupts

Volatile keyword

Needed for safe C code that plays with IO devices.

Definition: this value may be changed by something outside this linkage unit (object).

Examples

- #define LED_ADDR ((volatile const unsigned *)(8))
- volatile const unsigned *led_addr = 0x8;

Otherwise, compiler might optimize away accesses.

Const keyword

- Value (accessed via this pointer) won’t be modified (through this pointer).
- Makes debugging easier.
- Get right the first time to avoid type conflicts.
- Default to const pointers.

Pointers and function pointers

- Function pointers to pass code around dynamically among functions and build vector tables in C.
- Pointers
  - Type-safe addresses.
  - Avoid void * unless really needed.
  - When would you use this?
  - The type of the object cannot be known at compile time.

Void *, a short illustrative script

Compiler: Excuse me, sir. May I suggest using a round peg?

Programmer: Shut up! I don't care! Just do it!

Compiler: As you wish, sir.

OS: Here’s a 64 GB core dump file?

Function pointers

```
// Can use for generic functions.
int apple_checker(const void *x);

int orange_checker(const void *x);

int check_stuff(void *stuff_array,
        int (*checker)(const void *);

// Can use for jump tables.
int (*func_ptr[3])(const void *) = {func1, func2, func3};

// If function pointers differ, then use void pointers.
void (*func_ptr[3]) = {func1, func2, func3};
```

Cdecl

- http://cdecl.org/ is great!
- int (*f)(const int *)
  - declare f as pointer to function (pointer to const int) returning int
- int (*f)[3](const int *)
  - declare f as pointer to array 3 of function (pointer to const int) returning int (which is not valid in C)
- int (*f)(const int *)[4]
  - declare f as pointer to function (pointer to const int) returning array 4 of int

Outline

- Assembler directives review
- Volatile, const, and function pointers
- **Interrupts**

Interrupts summary

- Exceptions happen when something outside the normal flow of the program occurs.
- Interrupts are a type of exception generally triggered by hardware, not the program.
- Interrupt vector allows specification of ISRs for particular interrupts.
- Separate processor mode and stack for interrupts with some registers duplicated and aliased.
- Can disable interrupts or prioritize responses to later interrupts.
- No need to change processor mode when going from one ISR to another.
- Shouldn’t allow time-critical sections of code to be interrupted.
- Should get through time-critical sections of code ASAP.

Interrupts

Why do these matter?

- Informs a program of (usually) external event.
- Interrupts execution flow.
- Enables event-driven system design!!!
  - Low-power.
  - Often simpler.

Key questions:

- Where do interrupts come from?
- How to save state for later continuation?
- How to ignore interrupts?
- How to prioritize interrupts?
- How to share interrupts?

I/O data transfer

Two key questions to determine how data are transferred to/from non-trivial I/O device.

1. How does the CPU know when data is available?
   a. Polling
   b. Interrupts

2. How is data transferred into and out of the device?
   a. Programmed I/O
   b. Direct Memory Access (DMA)

Interrupts

Interrupt (a.k.a. exception or trap)

- Stop executing program.
- Execute interrupt handler / service routine (ISR).
- Resume program.

Similar to procedure calls, but

- Occur between any two instructions.
- Transparent to the running program (usually).
- Not generally explicitly called by program.
- Pick procedure (ISR) based on interrupt #.

Instruction interrupts

- TLB miss.
- Illegal/unimplemented instruction.
- Divide by 0.
- Names: trap, exception.

External interrupts

- Reset button.
- Timer expires.
- Power failure.
- System error.
- Names: interrupt, external interrupt.

Interrupt process

- Something tells the processor.
  - E.g., input pin.
- Processor transfers control to ISR.
  - Use interrupt vector or jump table.
- ISR executes.
- Resumes prior program at same location.
- Doing this right is complex.

Interrupts complicate processor design

- Which ISR to call?
- How to resume program when done?
  - Instruction pointer? Other state?
- What about partially executed instructions in the pipeline?
- What if we get an interrupt while we are processing our interrupt?
  - What if we are in a “critical section?”

Where

- If you know the interrupt source.
  - Interrupt vector (ARM).
  - Jump table.
- If not.
  - Must poll all sources to find out.

Returning

- Need to store the return address somewhere.
  - Stack would involve a load/store that might cause another interrupt.
  - Dedicated register in || with R14 (link register).
    - What if there is another interrupt?
    - Turns off interrupts.
    - If manually reenabled, another interrupt can clobber return address.
    - System mode (not IRQ mode) fixes this.
  - Interrupt mode duplicates some registers, e.g., PC and R14 (R8-R14).
  - Pushes/pops all eight registers on stack.

Implications of architectural optimizations

- Out-of-order execution
  - If any state of a “too fast” instruction made its way out of the processor before an interrupt, system state corrupted.
- Need to clean things up before/in ISR.
- Generally a concern for microarchitect, not you.

Nested interrupts

- What if it is handled immediately?
  - If a dedicated interrupt return IP register is being used, how many do we need?
  - What if the ISR is half-way through a precisely timed bus transaction?
- Ignore it: Bad if it is important.
- Prioritize.
  - Take more important interrupts.
  - Ignore the rest
  - Still have dedicated register problems.
  - Have to consider possibility of ISR failing due to timing problems.
- In reality, good designers get through critical portions fast, and if there are more time consuming operations, defer them to allow interrupts to be turned on again.
- Implies switching back to user mode to protect link register and emptying the IRQ stack.

Critical section

- Ignore less important interrupts.
- Take more important interrupts.
- Avoid causing exceptions in interrupt code.
- Keep as short as possible.
  - E.g., write a value to memory that informs the program of something.
  - Program deals with it at a good time.

Example: generally bad

```
void isr(void) {
    Do something complex/slow.
}
```

Example: generally good

```
volatile int button_pressed;

void isr(void) {
    ++(*button_pressed);
}

int superloop(void) {
    while (1) {
        if (*button_pressed) {
            --(*button_pressed);
            button_service();
        }
        Do other stuff, like AI.
        Could also sleep.
    }
}
```

Our processor

- Over 100 interrupt sources
  - E.g., power-on reset, bus errors, I/O pins changing state, data on a serial bus.
- Need a great deal of control
  - Ability to enable and disable interrupt sources
  - Ability to control where to branch to for each interrupt
  - Ability to set interrupt priorities
    - Who wins in case of a tie
    - Can interrupt **A** interrupt the ISR for interrupt **B**?
      - If so, **A** can “preempt” **B**.
- All that control will involve memory mapped I/O.

Table 7.1 List of 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; Memory Protection Unit (MPU) violation or access to illegal locations
5	Bus fault	Programmable	Bus error; occurs when Advanced High-Performance Bus (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)
6	Usage fault	Programmable	Exceptions resulting from program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7–10	Reserved	NA	—
11	SVC	Programmable	Supervisor Call
12	Debug monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	—
14	PendSV	Programmable	Pendable Service Call
15	SYSTICK	Programmable	System Tick Timer

Table 7.2 List of External Interrupts

Exception Number	Exception Type	Priority
16	External Interrupt #0	Programmable
17	External Interrupt #1	Programmable
…	…	…
255	External Interrupt #239	Programmable

Interrupt vectors

```
g_pfnVectors:
    .word  _estack
    .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
    .word  WdogWakeup_IRQHandler
    .word  BrownOut_1_5V_IRQHandler
    .word  BrownOut_3_3V_IRQHandler
.............. (they continue)
```

Table 7.1 List of 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; Memory Protection Unit (MPU) violation or access to illegal locations
5	Bus fault	Programmable	Bus error; occurs when Advanced High-Performance Bus (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)
6	Usage fault	Programmable	Exceptions resulting from program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7–10	Reserved	NA	—
11	SVC	Programmable	Supervisor Call
12	Debug monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	—
14	PendSV	Programmable	Pendable Service Call
15	SYSTICK	Programmable	System Tick Timer

Table 7.2 List of External Interrupts

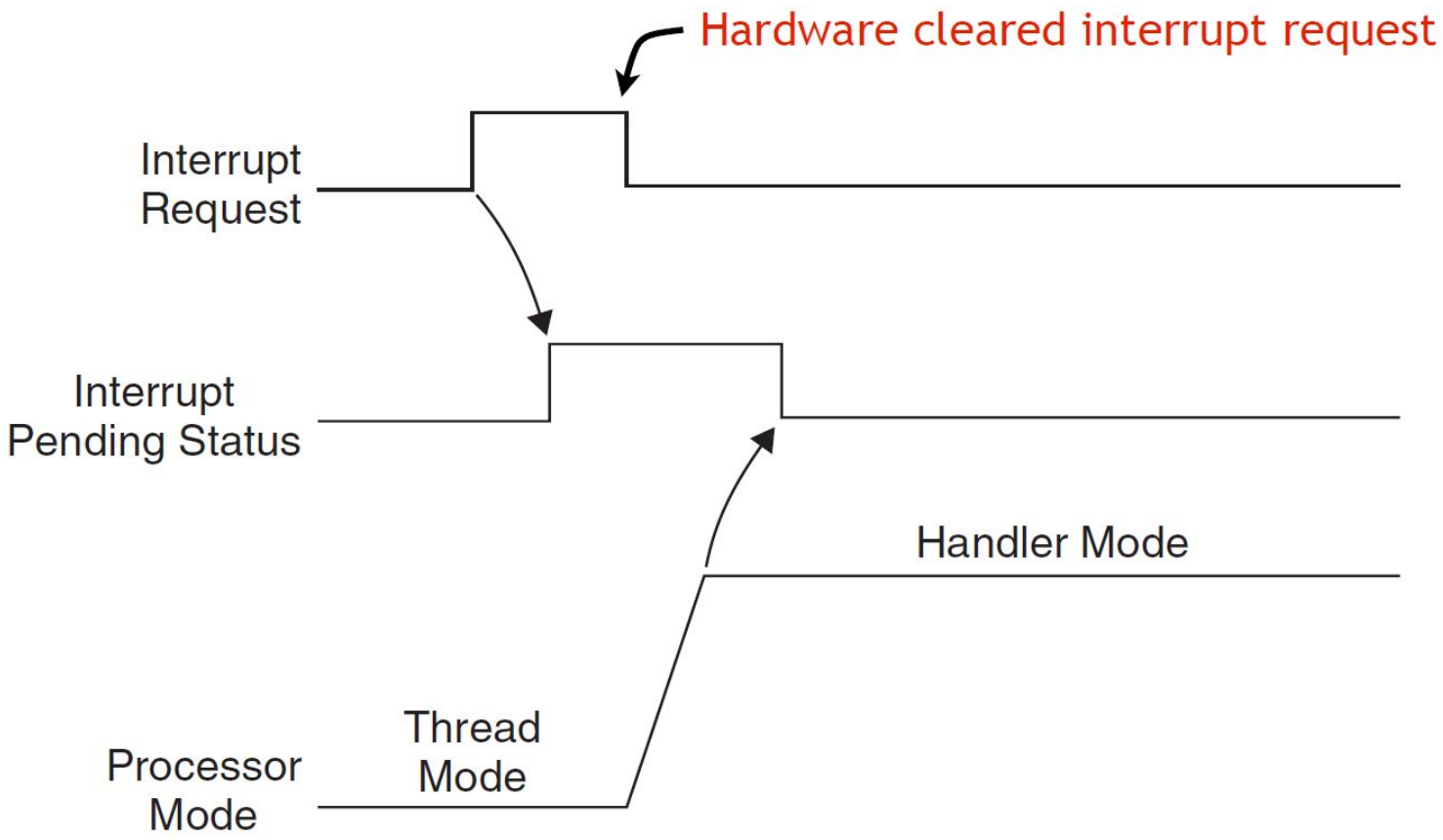
Exception Number	Exception Type	Priority
16	External Interrupt #0	Programmable
17	External Interrupt #1	Programmable
…	…	…
255	External Interrupt #239	Programmable

Interrupt handlers

```
23g_pfnVectors:
24    .word  _estack
25    .word  Reset_Handler
26    .word  NMI_Handler
27    .word  HardFault_Handler
28    .word  MemManage_Handler
29    .word  BusFault_Handler
30    .word  UsageFault_Handler
31    .word  0
32    .word  0
```

```
192/*=========================================
193 * Reset_Handler
194 */
195    .global Reset_Handler
196    .type   Reset_Handler, %function
197Reset_Handler:
198_start:
```

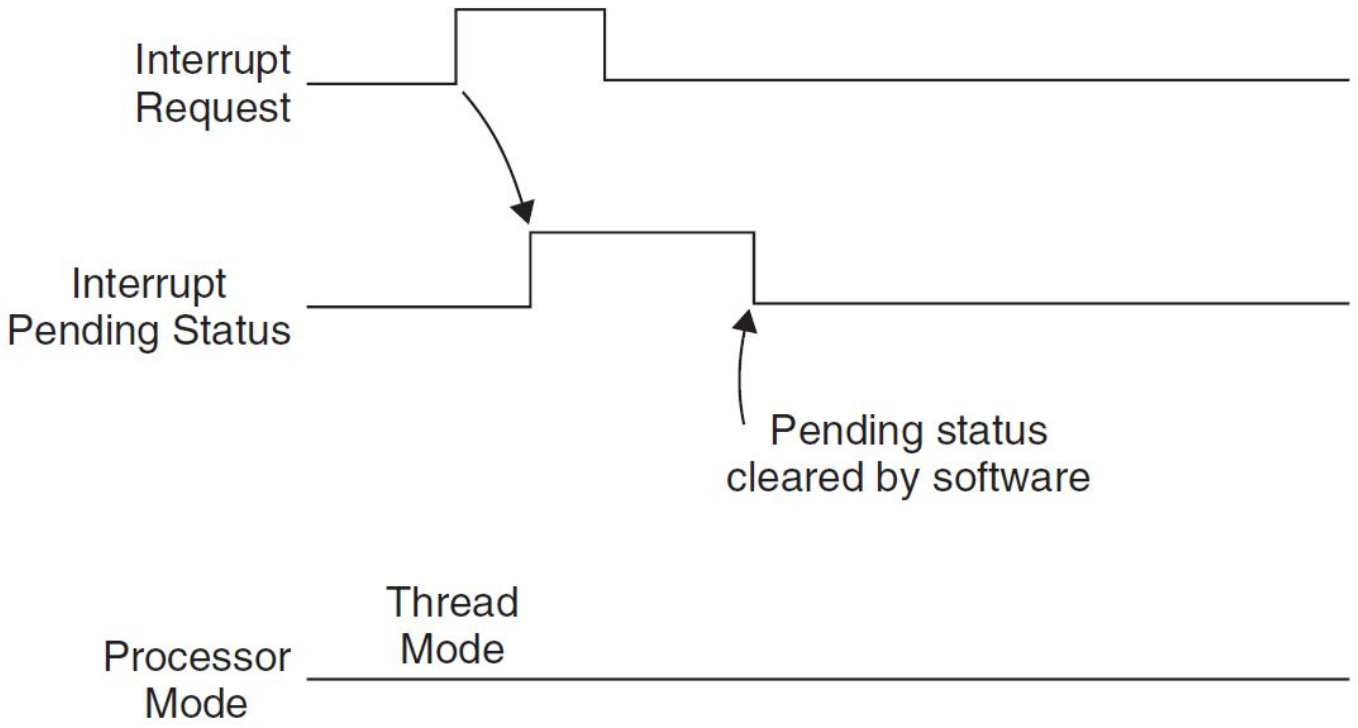
Pending interrupts



The normal case. Once Interrupt request is seen, processor puts it in "pending" state even if hardware drops the request.
IPS is cleared by the hardware once we jump to the ISR.

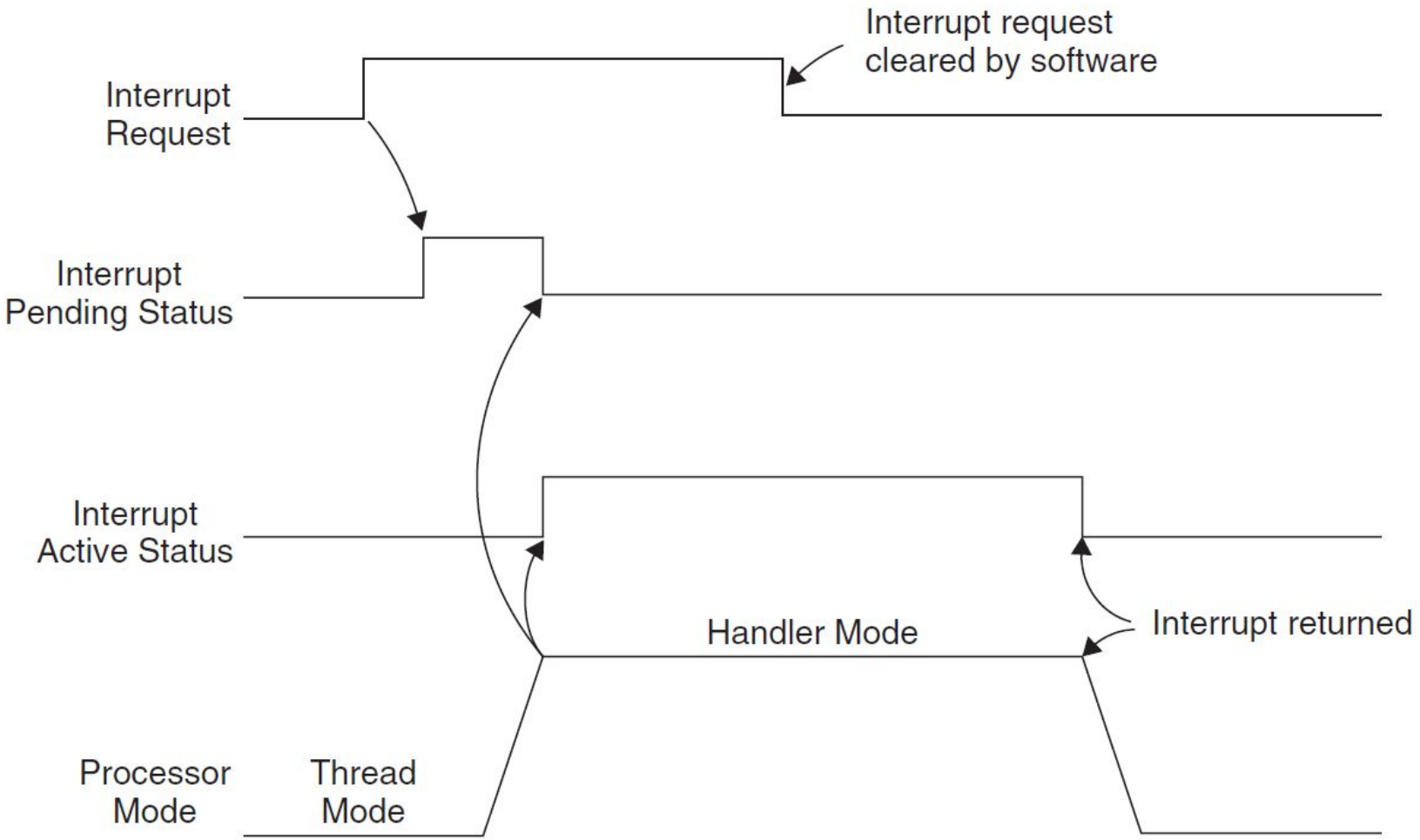
From *The Definitive Guide to the ARM Cortex-M3, Section 7.4*

Untaken interrupts

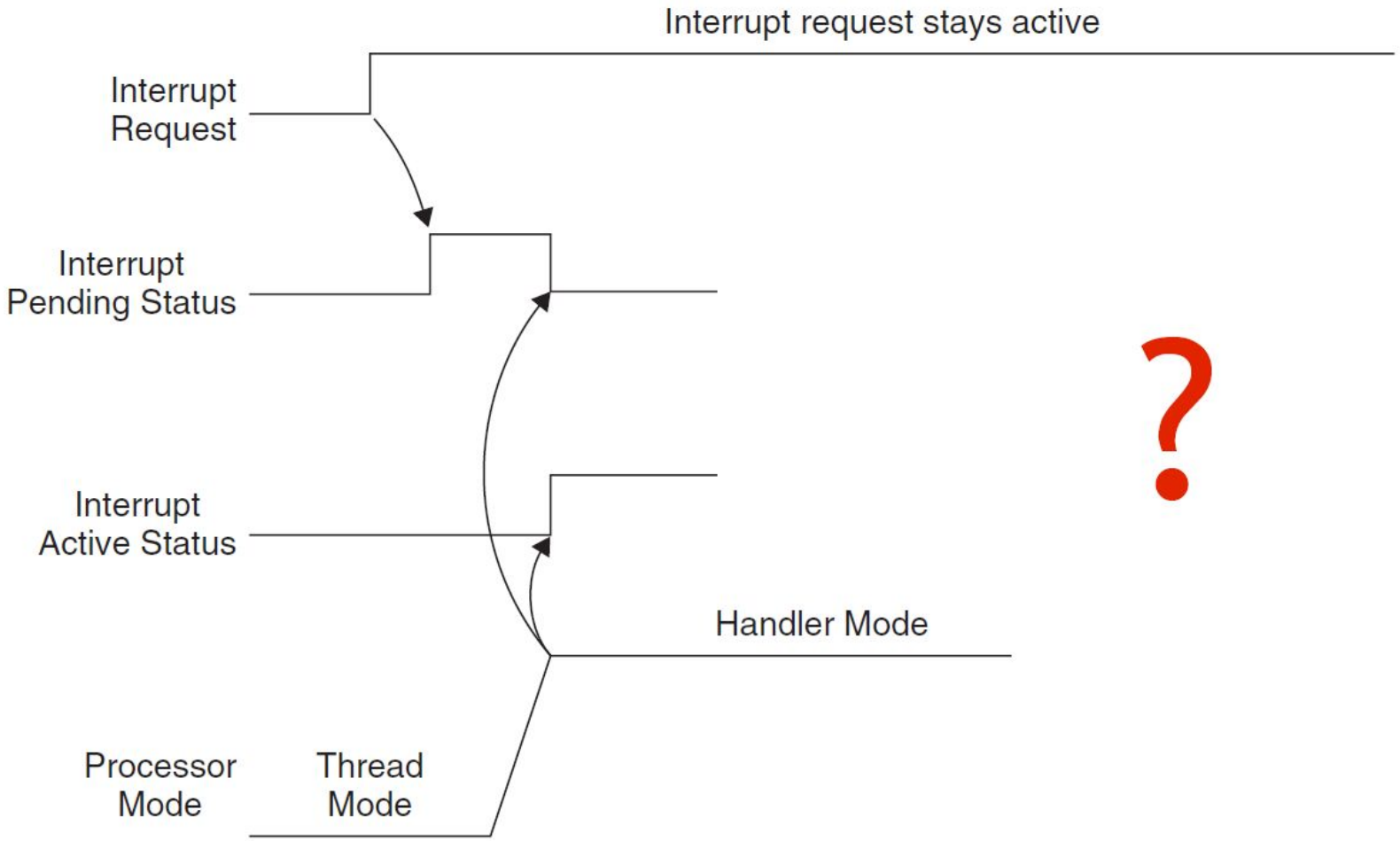


In this case, the processor never took the interrupt because we cleared the IPS by hand (via a memory-mapped I/O register)

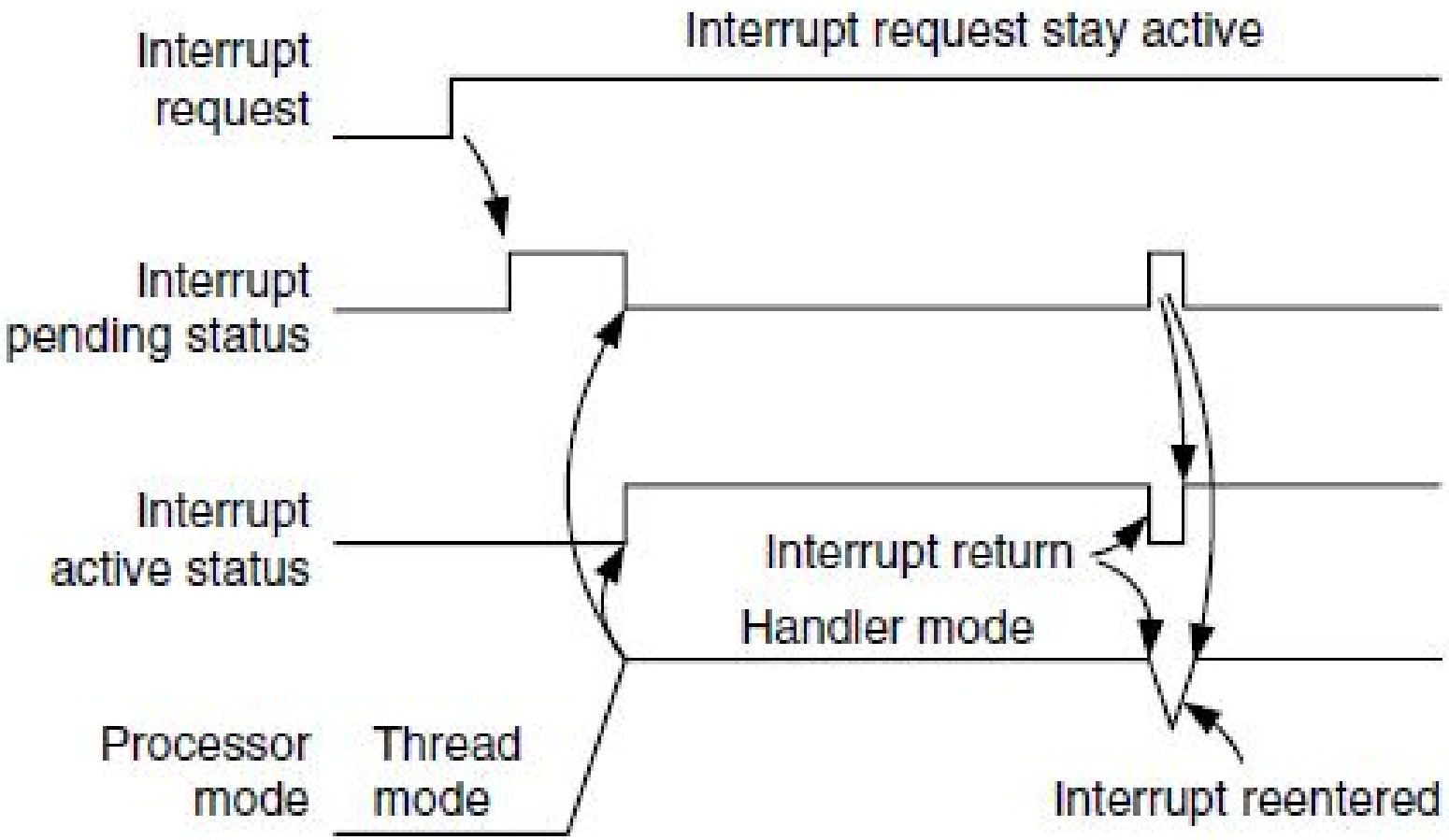
Active status set during handler execution



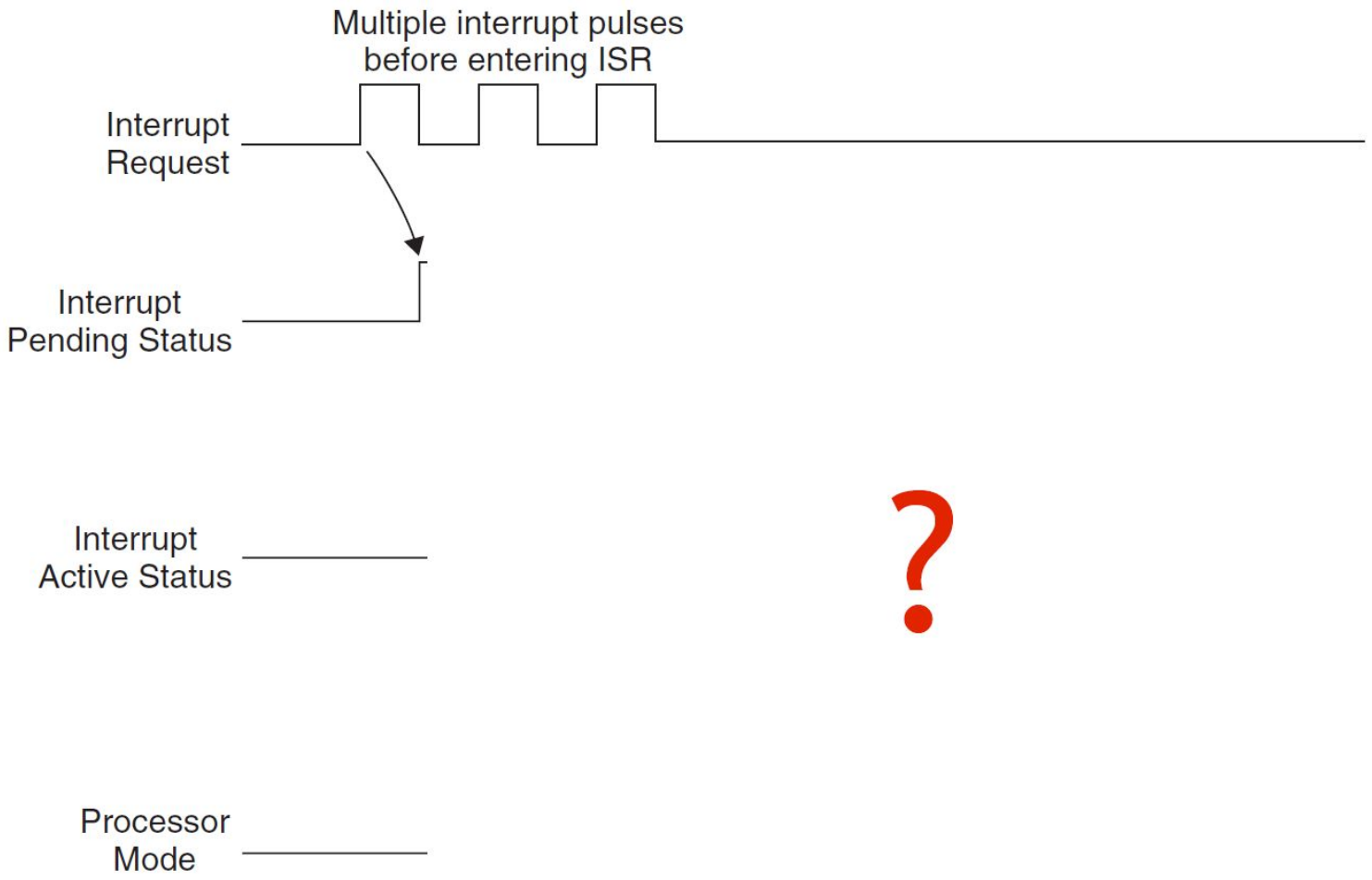
Interrupt request not cleared



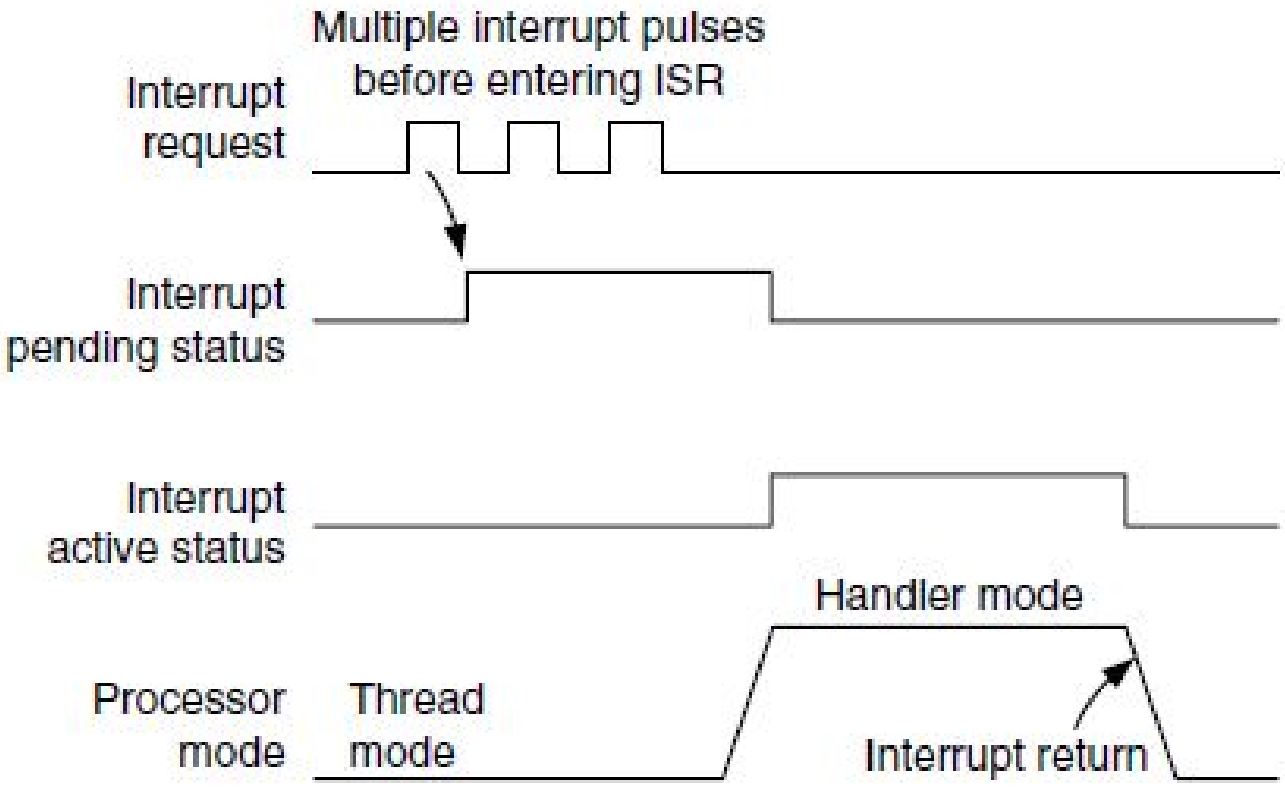
Answer


Interrupt request stay active
Interrupt request
Interrupt pending status
Interrupt active status
Interrupt return
Handler mode
Processor mode
Thread mode
Interrupt reentered

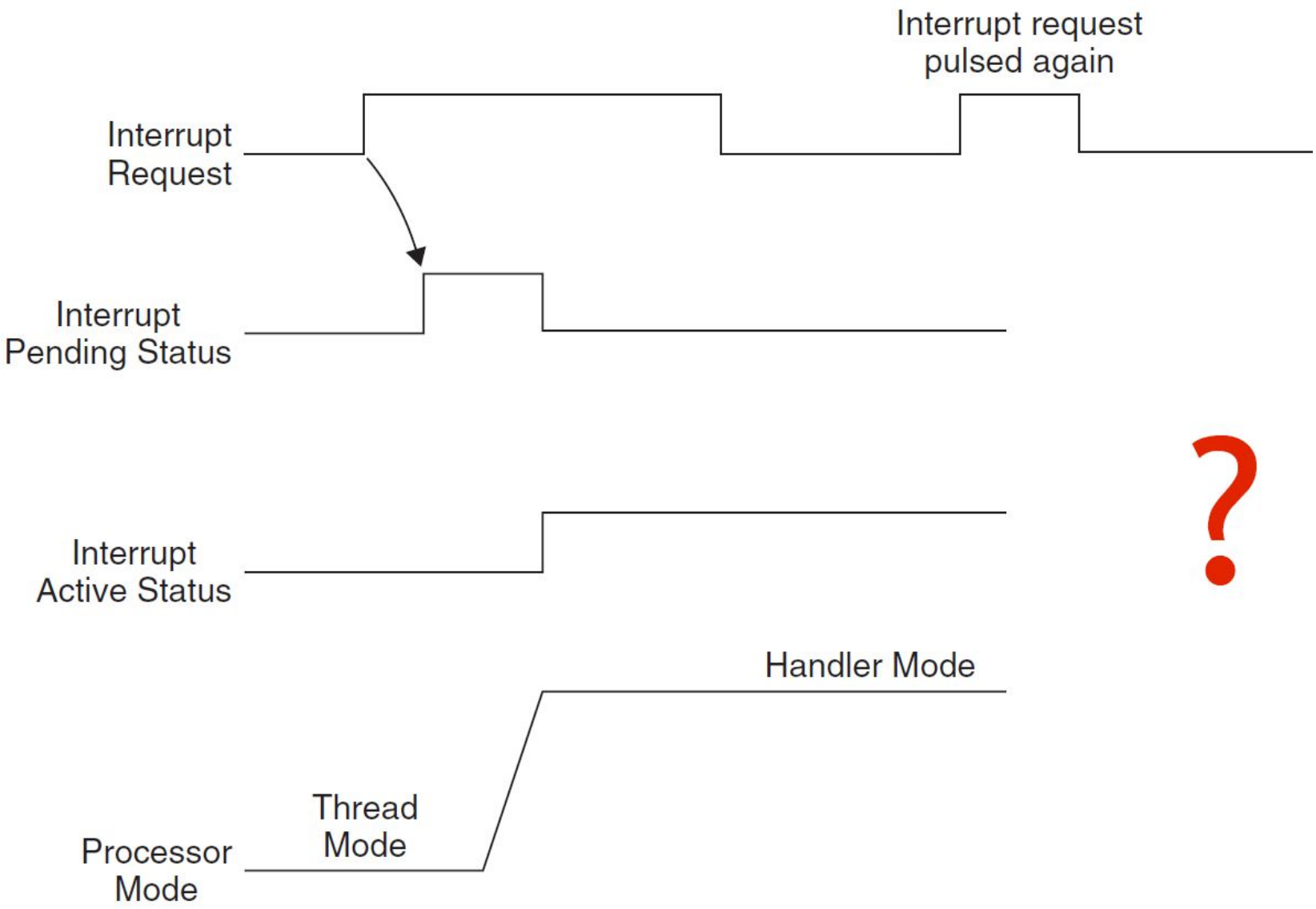
Interrupt pulses before entering ISR



Answer


Multiple interrupt pulses
before entering ISR
Interrupt
request
Interrupt
pending status
Interrupt
active status
Handler mode
Processor
mode
Thread
mode
Interrupt return

New interrupt request after pending cleared



Tail chaining

- Processor can serve multiple interrupts without returning.
- Improves response latency.
- No need for state save/restore.

Configuring the NVIC

- Interrupt Set Enable and Clear Enable
  - 0xE000E100-0xE000E11C, 0xE000E180-0xE000E19C

0xE000E100	SETENA0	R/W	0	Enable for external interrupt #0–31 bit[0] for interrupt #0 (exception #16) bit[1] for interrupt #1 (exception #17) ... bit[31] for interrupt #31 (exception #47) Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status
0xE000E180	CLRENA0	R/W	0	Clear enable for external interrupt #0–31 bit[0] for interrupt #0 bit[1] for interrupt #1 ... bit[31] for interrupt #31 Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current enable status

Configuring the NVIC (2)

- Set Pending & Clear Pending
  - 0xE000E200-0xE000E21C, 0xE000E280-0xE000E29C

0xE000E200	SETPEND0	R/W	0	Pending for external interrupt #0–31 bit[0] for interrupt #0 (exception #16) bit[1] for interrupt #1 (exception #17) ... bit[31] for interrupt #31 (exception #47) Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status
0xE000E280	CLRPEND0	R/W	0	Clear pending for external interrupt #0–31 bit[0] for interrupt #0 (exception #16) bit[1] for interrupt #1 (exception #17) ... bit[31] for interrupt #31 (exception #47) Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current pending status

Configuring the NVIC (3)

- Interrupt Active Status Register
  - 0xE000E300-0xE000E31C

Address	Name	Type	Reset Value	Description
0xE000E300	ACTIVE0	R	0	Active status for external interrupt #0–31 bit[0] for interrupt #0 bit[1] for interrupt #1 … bit[31] for interrupt #31
0xE000E304	ACTIVE1	R	0	Active status for external interrupt #32–63
…	–	–	–	–

Interrupt priorities

- If multiple interrupts arrive at same time, prioritize.
- 3 fixed highest priorities.
- Up to 256 programmable priorities and 128 preemption levels.
- Particular processors support a subset of priorities.
- Higher priorities preempt lower.
- Priority can be sub-divided into groups.
  - Splits register into preempt priority and subpriority.
  - Subpriority used if two interrupts with same preempt priority arrive at same time.

Interrupt priorities (2)

Interrupt Priority (2)

7

- Interrupt Priority Level Registers
  - 0xE000E400-0xE000E4EF

Address	Name	Type	Reset Value	Description
0xE000E400	PRI_0	R/W	0 (8-bit)	Priority-level external interrupt #0
0xE000E401	PRI_1	R/W	0 (8-bit)	Priority-level external interrupt #1
…	–	–	–	–
0xE000E41F	PRI_31	R/W	0 (8-bit)	Priority-level external interrupt #31
…	–	–	–	–

Preemption priority and subpriority

Priority Group	Preempt Priority Field	Subpriority Field
0	Bit [7:1]	Bit [0]
1	Bit [7:2]	Bit [1:0]
2	Bit [7:3]	Bit [2:0]
3	Bit [7:4]	Bit [3:0]
4	Bit [7:5]	Bit [4:0]
5	Bit [7:6]	Bit [5:0]
6	Bit [7]	Bit [6:0]
7	None	Bit [7:0]

Use PRIGROUP field to control split.

Application Interrupt and Reset Control Register (Address 0xE000ED0C)

Bits	Name	Type	Reset Value	Description
31:16	VECTKEY	R/W	–	Access key; 0x05FA must be written to this field to write to this register, otherwise the write will be ignored; the read-back value of the upper half word is 0xFA05
15	ENDIANNESS	R	–	Indicates endianness for data: 1 for big endian (BE8) and 0 for little endian; this can only change after a reset
10:8	PRIGROUP	R/W	0	Priority group
2	SYSRESETREQ	W	–	Requests chip control logic to generate a reset
1	VECTCLRACTIVE	W	–	Clears all active state information for exceptions; typically used in debug or OS to allow system to recover from system error (Reset is safer)
0	VECTRESET	W	–	Resets the Cortex-M3 processor (except debug logic), but this will not reset circuits outside the processor

PRIMASK, FAULTMASK, and BASEPRI

- Quickly disable all interrupts.
- Write 1 into PRIMASK to disable everything but hard fault and NMI.
  - mov r0, #1
  - msr PRIMASK, ro
- FAULTMASK for everything but NMI.

- Disable everything below P.
- Write P into BASEPRI.

Masking

B1.4.3 The special-purpose mask registers

There are three special-purpose registers which are used for the purpose of priority boosting. Their function is explained in detail in *Execution priority and priority boosting within the core* on page B1-18:

- the exception mask register (PRIMASK) which has a 1-bit value
- the base priority mask (BASEPRI) which has an 8-bit value
- the fault mask (FAULTMASK) which has a 1-bit value.

All mask registers are cleared on reset. All unprivileged writes are ignored.

The formats of the mask registers are illustrated in Table B1-4.

Table B1-4 The special-purpose mask registers

	31 ... 8	7 ... 1	0
PRIMASK	RESERVED		PM
FAULTMASK	RESERVED		FM
BASEPRI	RESERVED	BASEPRI	

Interrupt service routines

1. Automatic saving of registers upon exception
   - PC, PSR, R0-R3, R12, LR pushed on the stack
2. While bus busy, fetch exception vector
3. Update SP to new location
4. Update IPSR (low part of PSR) with new exception number
5. Set PC to vector handler
6. Update LR to special value EXC_RETURN

- Several other NVIC registers get updated
- Latency: as short as 12 cycles

The xPSR register layout

The APSR, IPSR and EPSR registers are allocated as mutually exclusive bitfields within a 32-bit register. The combination of the APSR, IPSR and EPSR registers is referred to as the xPSR register.

Table B1-2 The xPSR register layout

	31	30	29	28	27	26	25	24	23 … 16	15 … 10	9	8 … 0
APSR	N	Z	C	V	Q							
IPSR												0 or Exception Number
EPSR						ICI/IT	T			ICI/IT	a	

WFI: Wait For Interrupt

- Puts processor in low-power mode and waits for interrupt.
- Why?

Two stacks? MSP and PSP

- OS always uses MSP.
- Can configure processor so program uses PSP.
- Makes it harder for application code to corrupt OS/superloop state.

EXC_RETURN

- Why is my LR 0xfffffff9?
- Part magic number.
- Part source context status.
- Real return address at SP-24.

Interrupts summary

- Exceptions happen when something outside the normal flow of the program occurs.
- Interrupts are a type of exception generally triggered by hardware, not the program.
- Interrupt vector allows specification of ISRs for particular interrupts.
- Separate processor mode and stack for interrupts with some registers duplicated and aliased.
- Can disable interrupts or prioritize responses to later interrupts.
- No need to change processor mode when going from one ISR to another.
- Shouldn’t allow time-critical sections of code to be interrupted.
- Should get through time-critical sections of code ASAP.

Done.