EE-379 Embedded Systems and Applications
Interrupts (1)

Cristinel Ababei
Department of Electrical Engineering, University at Buffalo
Spring 2013
Note: This course is offered as EE 459/500 in Spring 2013

Outline

- Introduction
- NVIC and Interrupt Control
- Interrupt Pending
- Examples
- Interrupt Service Routines

How does it work?

- Something tells the processor core (which is running the main execution flow) there is an interrupt/exception
- Core transfers control to code that needs to be executed to address the interrupt
- Said code "returns" to the main (old) program

Some questions

- How do you figure out where to branch/jump to?
  - If you know number the possible interrupt cases, and an interrupt comes in, you can just branch to a location, using that number as an offset
- How to you ensure that you can get back to where you started?
  - Store return address to stack or dedicated register?
- Don't we have a pipeline? What about partially executed instructions?
  - Complex architectures
- What if we get an interrupt while we are already "processing" an interrupt?
  - Nested interrupts: handle directly, ignore, prioritize
- What if we are in a "critical section?"
  - Prioritization

Interrupts

- An interrupt is the automatic transfer of software execution in response to a hardware event that is asynchronous with the current software execution
- This hardware event is called a trigger and it breaks the execution flow of the main thread of the program
- The event causes the CPU to stop executing the current program and begin executing a special piece of code called an interrupt handler or interrupt service routine (ISR)
- Typically, the ISR does some work and then resumes the interrupted program

Interrupts

- The hardware event can either be:

  1) A **busy-to-ready transition** in an external I/O device. Caused by the external world
  - Peripheral/device, e.g., UART input/output device
  - Reset button, Timer expires, Power failure, System error
  - Names: exception, interrupt, external interrupt

  2) An **internal event**
  - Bus fault, memory fault
  - A periodic timer
  - Div. by zero, illegal/unsupported instruction
  - Names : exception, trap, system exception
- When the hardware needs service, signified by a busy to ready state transition, it will request an interrupt by setting its trigger flag

Cortex-M3 Interrupts

- Exceptions:
  - System exceptions: numbered 1 to 15
  - External interrupt inputs: numbered from 16 up
- Different numbers of external interrupt inputs (from 1 to 240) and different numbers of priority levels
- Value of the current running exception is indicated by:
  - The special register Interrupt Program Status Register (IPSR) or
  - From the NVIC's Interrupt Control State Register (the VECTACTIVE field)

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; 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 *prefetch abort* if it is an instruction fetch or *data abort* 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

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 Programming

- To arm (disarm) a device/peripheral means to enable (shut off) the source of interrupts. Each potential interrupting trigger has a separate "arm" bit. One arms (disarms) a trigger if one is (is not) interested in interrupts from this source.
- To enable (disable) means to allow interrupts at this time (postponing interrupts until a later time). On the ARM Coretx-M3 processor, there is one interrupt enable bit for the entire interrupt system. In particular, to disable interrupts we set the interrupt mask bit, I, in PRIMASK register.

Interrupt Programming

- Interrupts on the Cortex-M3 are controlled by the Nested Vectored Interrupt Controller (NVIC)
- To activate an "interrupt source" we need to set its priority and enable that source in the NVIC:

  Activate = Set priority + Enable source in NVIC

- This activation is in addition to the "arm" and "enable" steps discussed earlier

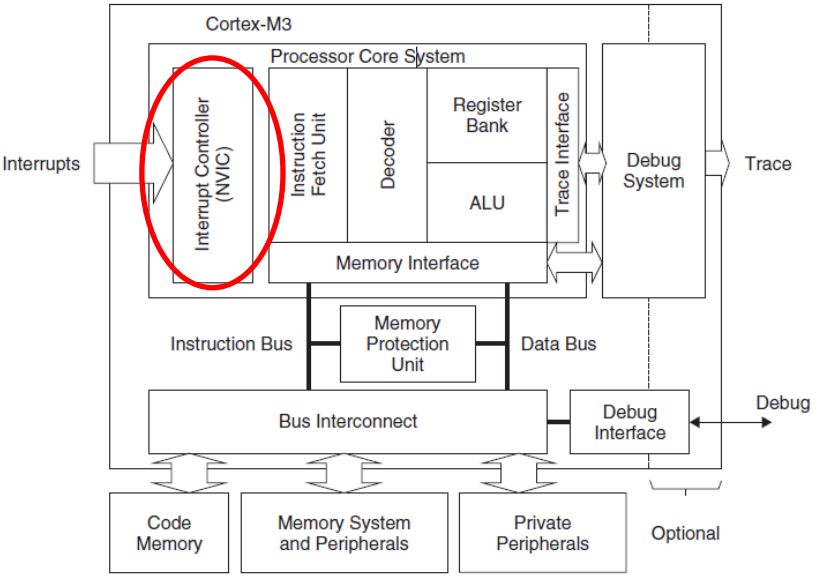
Outline

- Introduction
- NVIC and Interrupt Control
- Interrupt Pending
- Examples
- Interrupt Service Routines

Nested Vectored Interrupt Controller **(NVIC)**

- Interrupts on the Cortex-M3 are controlled by the Nested Vector Interrupt Controller (NVIC)
- NVIC supports 1 to 240 external interrupt inputs (commonly known as IRQs)
- NVIC control registers are accessible as memory-mapped devices
- NVIC can be accessed as memory location **0xE000E000**
- NVIC contains:
  - control registers and control logic for interrupt processing
  - registers for the MPU
  - SYSTICK Timer
  - debugging controls
- In the LPC17xx, the NVIC supports 35 vectored interrupts

Simplified Cortex-M3 Architecture



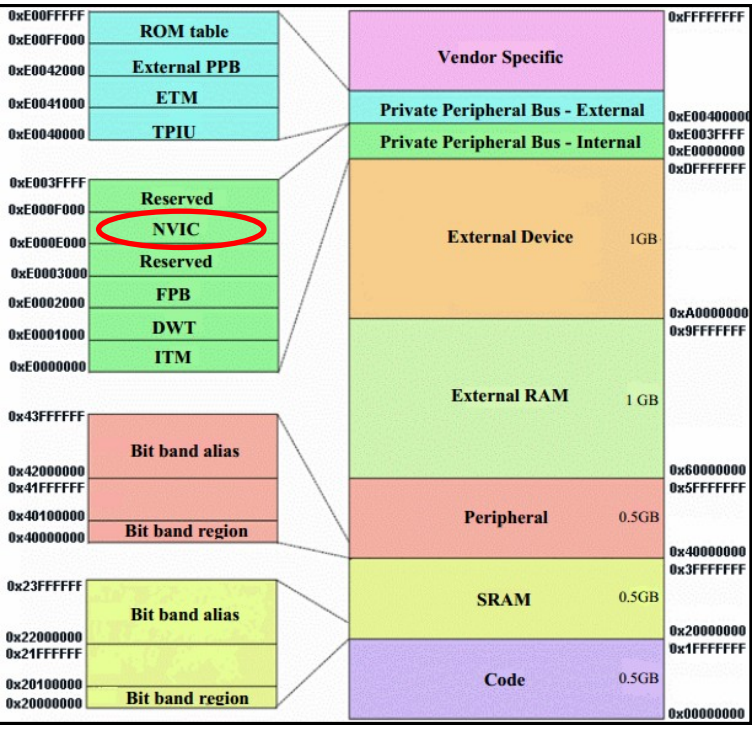
NVIC Programmers Model

Table 6-1 NVIC registers

Address	Name	Type	Reset	Description
0xE000E004	ICTR	RO	-	*Interrupt Controller Type Register, ICTR*
0xE000E100 - 0xE000E11C	NVIC_ISER0 - NVIC_ISER7	RW	0x00000000	Interrupt Set-Enable Registers
0xE000E180 - 0E000xE19C	NVIC_ICER0 - NVIC_ICER7	RW	0x00000000	Interrupt Clear-Enable Registers
0xE000E200 - 0xE000E21C	NVIC_ISPR0 - NVIC_ISPR7	RW	0x00000000	Interrupt Set-Pending Registers
0xE000E280 - 0xE000E29C	NVIC_ICPR0 - NVIC_ICPR7	RW	0x00000000	Interrupt Clear-Pending Registers
0xE000E300 - 0xE000E31C	NVIC_IABR0 - NVIC_IABR7	RO	0x00000000	Interrupt Active Bit Register
0xE000E400 - 0xE000E4EC	NVIC_IPR0 - NVIC_IPR59	RW	0x00000000	Interrupt Priority Register

From Cortex-M3 Technical Reference Manual

Memory Map



Basic Interrupt Configuration

- Each external interrupt has several registers associated with it:
  - Enable and clear enable registers
  - Set-pending and clear-pending registers
  - Active status
  - Priority level
- In addition, a number of other registers can also affect the interrupt processing:
  - Exception-masking registers (PRIMASK, FAULTMASK, and BASEPRI)
  - Vector Table Offset register
  - Software Trigger Interrupt register
  - Priority Group

Interrupt Enable and Clear Enable

- The Interrupt Enable register is programmed via two addresses
  - To set the enable bit, we write to the SETENA register address
  - To clear the enable bit, you need to write to the CLRENA register address
    - 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

SETENA0/1 is a general name to refer to this register. Consult the MCU User Manual for your particular MCU to see what they are actually called. For example, in the case of LPC17xx these registers are called ISER0 and ISER1. See page 77-78 of LPC17xx user manual

See page 77-80 of LPC17xx user manual for description of ISER0,ISER1 and ICER0,ICER1!

Interrupt Pending and Clear Pending

- If an interrupt takes place but cannot be executed immediately (e.g., if another higher-priority interrupt handler is running), it will be pended
- The interrupt pending status can be accessed through the Interrupt Set Pending (SETPEND) and Interrupt Clear Pending (CLRPEND) registers
  - 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

See page 81-84 of LPC17xx user manual for description of ISPR0,ISPR1 and ICPR0,ICPR1!

Active Status

- Each external interrupt has an active status bit.
- When the processor starts the interrupt handler, the bit is set to 1 and cleared when the interrupt return is executed.
- **Interrupt Active Bit Status registers**
  - **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
...	–	–	–	–

See page 85-86 of LPC17xx user manual for description of IABR0,IABR1!

Priority Levels

- Each external interrupt has an associated priority-level register, which has a maximum width of 8 bits and a minimum width of 3 bits
- **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
...	-	-	-	-

See page 87-89 of LPC17xx user manual for description of IPR0..IPR8!

Interrupt Priority

- An exception can be carried out can be affected by the priority of the exception
- A higher-priority (smaller number in priority level) exception can preempt a lower-priority (larger number in priority level) exception
- Cortex-M3 supports three fixed highest-priority levels and up to 256 levels of programmable priority (a maximum of 128 levels of preemption
- Most Cortex-M3 chips have fewer supported levels - for example, 8, 16, 32, ...

Levels of priority

- Reduction of levels is implemented by cutting out the LSB part of the **priority configuration registers**. Example of 3-bit implemented:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented			Not implemented, read as zero				

- In this example, we have possible priority levels:
  - 0x00 (high priority), 0x20, 0x40, 0x60, 0x80, 0xA0, 0xC0, and 0xE0 (the lowest)
- LPC17xx has 32 programmable interrupt priority levels

Interrupt priority

- Priority can be sub-divided into priority groups
- Splits priority register into two halves:
  - Preempt priority – indicates if an interrupts can preempt another
  - Sub priority – used if 2 interrupts of the same group arrive at the same time

Vector Tables

- When an exception takes place and is being handled by the Cortex-M3, the processor will need to locate the starting address of the exception handler
- This information is stored in the vector table
- Each exception has an associated 32-bit vector that points to the memory location where the ISR that handles the exception is located
- Vectors are stored in ROM at the beginning of the memory

Vector Table

- Exception vector table after power-up is located at address 0x00000000:

Address	Exception Number	Value (Word Size)
0x00000000	–	MSP initial value
0x00000004	1	Reset vector (program counter initial value)
0x00000008	2	NMI handler starting address
0x0000000C	3	Hard fault handler starting address
...	...	Other handler starting address

- ROM location 0x00000000 has the initial stack pointer
- Location 0x00000004 contains the initial program counter (PC), which is called the reset vector
- Reset vector points to a function called reset handler, which is the first thing executed following reset
- Vector table can be relocated to change interrupt handlers at runtime (vector table offset register)

Vector Table

- Example of a few vectors as defined inside **startup_LPC17xx.s**:

```
__Vectors
    DCD     __initial_sp              ; Top of Stack
    DCD     Reset_Handler             ; Reset Handler
    DCD     NMI_Handler               ; NMI Handler
    DCD     HardFault_Handler         ; Hard Fault Handler
    ...
    ; External Interrupts
    DCD     WDT_IRQHandler            ; 16: Watchdog Timer
    DCD     TIMER0_IRQHandler         ; 17: Timer0
    ...
    DCD     UART0_IRQHandler          ; 21: UART0
```

Special registers: PRIMASK, FAULTMASK, and BASEPRI

- What if we quickly want to disable all interrupts?
- Write 1 into PRIMASK to disable all interrupt except NMI
  - MOV R0, #1
  - MSR PRIMASK, R0
- Write 0 into PRIMASK to enable all interrupts
- FAULTMASK is the same as PRIMASK, but also blocks hard fault (priority -1)
- What if we want to disable all interrupts below a certain priority?
- Write priority into BASEPRI
  - MOV R0, #0x60
  - MSR BASEPRI, R0

Software interrupts

- Software interrupts can be generated in two ways:
  - Use the SETPEND register
  - Use the Software Trigger Interrupt Register (STIR)

Software Trigger Interrupt Register (0xE000EF00)

Bits	Name	Type	Reset Value	Description
8:0	INTID	W	-	Writing the interrupt number sets the pending bit of the interrupt; for example, write 0 to pend external interrupt #0

See page 90 of LPC17xx user manual for description of STIR!

The SYSTICK Timer

- Often a hardware timer is used:
  - To generate interrupts so that the OS can carry out task management
  - As an alarm timer, for timing measurement, etc.
- Cortex-M3 processor includes a simple timer: 24-bit down counter
- The SYSTICK Timer is integrated with the NVIC and can be used to generate a SYSTICK exception (exception type #15)
- SYSTICK Timer is controlled by four registers

SYSTICK Timer Control and Status Regs

SYSTICK Control and Status Register (0xE000E010)

Bits	Name	Type	Reset Value	Description
16	COUNTFLAG	R	0	Read as 1 if counter reaches 0 since last time this register is read; clear to 0 automatically when read or when current counter value is cleared
2	CLKSOURCE	R/W	0	0 = External reference clock (STCLK) 1 = Use core clock
1	TICKINT	R/W	0	1 = Enable SYSTICK interrupt generation when SYSTICK timer reaches 0 0 = Do not generate interrupt
0	ENABLE	R/W	0	SYSTICK timer enable

See page 505 of LPC17xx user manual for description !

SYSTICK Reload Value Register (0xE000E014)

Bits	Name	Type	Reset Value	Description
23:0	RELOAD	R/W	0	Reload value when timer reaches 0

SYSTICK Current Value Register (0xE000E018)

Bits	Name	Type	Reset Value	Description
23:0	CURRENT	R/Wc	0	Read to return current value of the timer. Write to clear counter to 0. Clearing of current value also clears COUNTFLAG in SYSTICK Control and Status Register

SYSTICK Calibration Value Register (0xE000E01C)

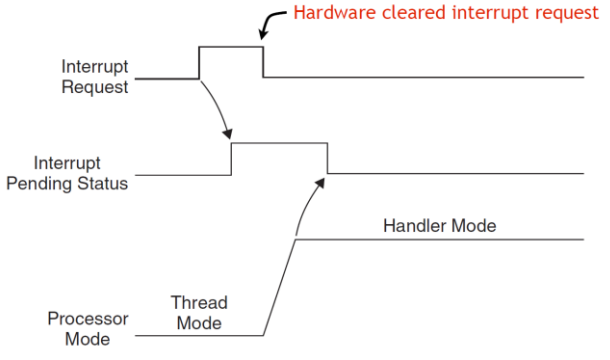
Bits	Name	Type	Reset Value	Description
31	NOREF	R	–	1 = No external reference clock (STCLK not available) 0 = External reference clock available
30	SKEW	R	–	1 = Calibration value is not exactly 10 ms 0 = Calibration value is accurate
23:0	TENMS	R/W	0	Calibration value for 10 ms.; chip designer should provide this value via Cortex-M3 input signals. If this value is read as 0, calibration value is not available

Outline

- Introduction
- NVIC and Interrupt Control
- Interrupt Pending
- Examples
- Interrupt Service Routines

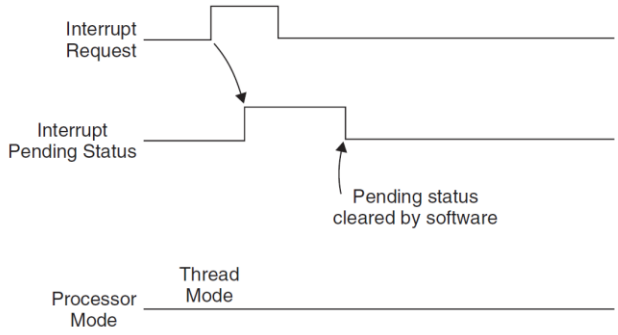
Interrupt Pending

- 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



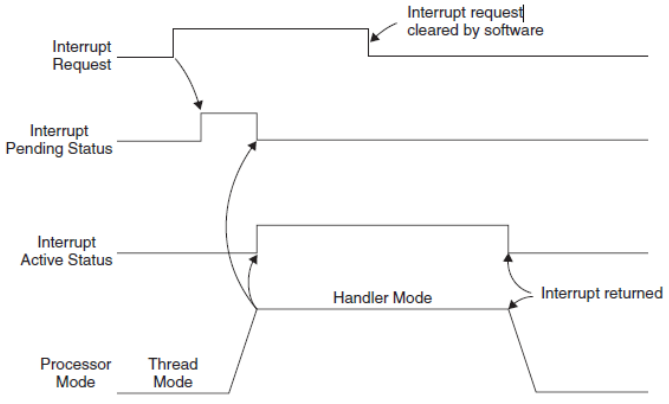
Interrupt pending

- If the pending status is cleared before the processor starts responding to the pended interrupt (e.g., because pending status register is cleared while PRIMASK/FAULTMASK is set to 1), the interrupt can be canceled
- The pending status of the interrupt can be accessed in the NVIC and is writable, so you can clear a pending interrupt or use software to pend a new interrupt by setting the pending register



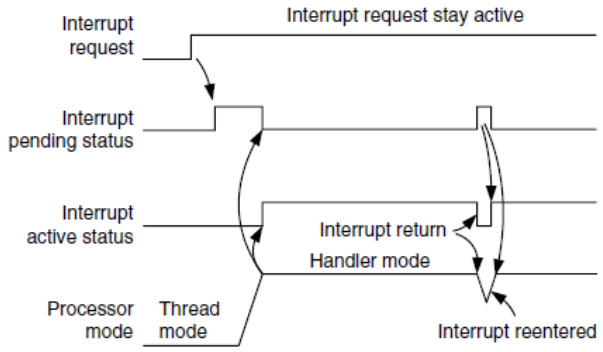
Active status during interrupt handling

- When the processor starts to execute an interrupt, the interrupt becomes active and the pending bit will be cleared automatically



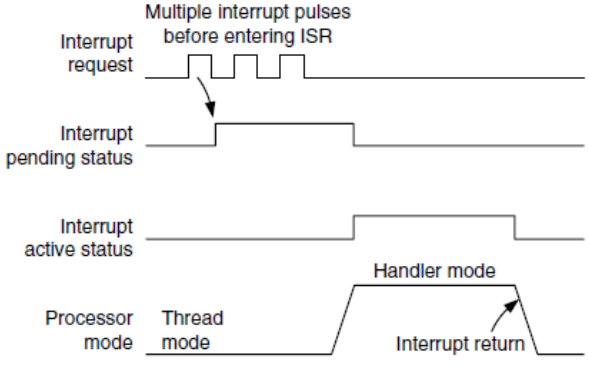
Interrupt source continues to hold

- If an interrupt source continues to hold the interrupt request signal active, the interrupt will be pended again at the end of the interrupt service routine



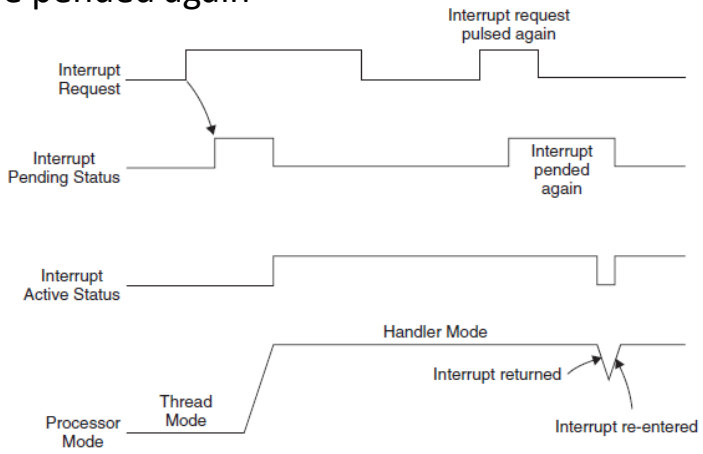
Interrupt is pulsed several times

- If an interrupt is pulsed several times before the processor starts processing it, it will be treated as one single interrupt request



Interrupt de-asserted, then pulsed again

- If an interrupt is de-asserted and then pulsed again during the interrupt service routine, it will be pended again



Outline

- Introduction
- NVIC and Interrupt Control
- Interrupt Pending
- Examples
- Interrupt Service Routines

Procedure for setting up an interrupt

1) When the system boots up, the priority group register might need to be set up
   - By default the priority group 0 is used (bit[7:1] of priority level is the preemption level and bit[0] is the subpriority level)
2) Copy the hard fault and NMI handlers to a new vector table location if vector table relocation is required
3) The Vector Table Offset register should also be set up to get the vector table ready (optional)

Procedure for setting up an interrupt

4) Set up the interrupt vector for the interrupt
   - Since the vector table could have been relocated, we might need to read the Vector Table Offset register, then calculate the correct memory location for your interrupt handler
   - This step might not be needed if the vector is hardcoded in ROM
5) Set up the priority level for the interrupt
6) Enable the interrupt

Example 1: The program in assembly (1)

```
LDR   R0, =0xE000ED0C     ; Application Interrupt and Reset
                          ; Control Register
LDR   R1, =0x05FA0500     ; Priority Group 5 (2/6)
STR   R1, [R0]            ; Set Priority Group
...
MOV   R4,#8               ; Vector Table in ROM
LDR   R5,=(NEW_VECT_TABLE+8)
LDMIA R4!,{R0-R1}         ; Read vectors address for NMI and
                          ; Hard Fault
STMIA R5!,{R0-R1}         ; Copy vectors to new vector table
...
LDR   R0,=0xE000ED08      ; Vector Table Offset Register
LDR   R1,=NEW_VECT_TABLE
STR   R1,[R0]             ; Set vector table to new location
...
```

Example 1: The program in assembly (2)

```
...
LDR   R0,=IRQ7_Handler    ; Get starting address of IRQ#7 handler
LDR   R1,=0xE000ED08      ; Vector Table Offset Register
LDR   R1,[R1]
ADD   R1, R1, #(4*(7+16)); Calculate IRQ#7 handler vector
                          ; address
STR   R0,[R1]             ; Setup vector for IRQ#7
...
LDR   R0,=0xE000E400      ; External IRQ priority base
MOV   R1, #0xC0
STRB  R1,[R0,#7]          ; Set IRQ#7 priority to 0xC0
...
LDR   R0,=0xE000E100      ; SETEN register
MOV   R1,#(1<<7)          ; IRQ#7 enable bit (value 0x1 shifted
                          ; by 7 bits)
STR   R1,[R0]             ; Enable the interrupt
```

Simplified procedure for setting up an interrupt

- If the application is stored in ROM and there is no need to change the exception handlers, we can have the whole vector table coded in the beginning of ROM in the Code region (0x00000000)
- This way, the vector table offset will always be 0 and the interrupt vector is already in ROM
- The only steps required to set up an interrupt are:
  1) Set up the priority group, if needed
  2) Set up the priority of the interrupt
  3) Enable the interrupt

Example 2: Blink LED

```
#include "LPC17xx.h"

int main (void)
{
  // (1) Timer 0 configuration (see page 490 of user manual)
  LPC_SC->PCONP |= 1 << 1; // Power up Timer 0 (see page 63 of user manual)
  LPC_SC->PCLKSEL0 |= 1 << 2; // Clock for timer = CCLK, i.e., CPU Clock (see page 56 of user manual)
  LPC_TIM0->MR0 = 1 << 23; // Give a value suitable for the LED blinking
                           // frequency based on the clock frequency (see page 492 and 496 of user manual)
  LPC_TIM0->MCR |= 1 << 0; // Interrupt on Match 0 compare (see page 492 and 496 of user manual)
  LPC_TIM0->MCR |= 1 << 1; // Reset timer on Match 0 (see page 492 and 496 of user manual)
  LPC_TIM0->TCR |= 1 << 1; // Manually Reset Timer 0 (forced); (see page 492 and 494 of user manual)
  LPC_TIM0->TCR &= ~(1 << 1); // Stop resetting the timer (see page 492 and 494 of user manual)
  // (2) Enable timer interrupt; TIMER0_IRQn is 1, see lpc17xx.h and page 73 of user manual
  NVIC_EnableIRQ(TIMER0_IRQn); // see core_cm3.h header file

  LPC_TIM0->TCR |= 1 << 0; // Start timer (see page 492 and 494 of user manual)
  LPC_SC->PCONP |= ( 1 << 15 ); // Power up GPIO (see lab1)
  LPC_GPIO1->FIODIR |= 1 << 29; // Put P1.29 into output mode. LED is connected to P1.29

  while (1) // Why do we need this?
  {
    // do nothing
  }
  return 0;
}
```

Example 2: Blink LED

```
// Here, we describe what should be done when the interrupt on Timer 0 is handled;
// We do that by writing this function, whose address is "recorded" in the vector table
// from file startup_LPC17xx.s under the name TIMER0_IRQHandler;
void TIMER0_IRQHandler(void)
{
  if ( (LPC_TIM0->IR & 0x01) == 0x01 ) // if MR0 interrupt
  {
    LPC_TIM0->IR |= 1 << 0; // Clear MR0 interrupt flag (see page 492 and 493 of user manual)
    LPC_GPIO1->FIOPIN ^= 1 << 29; // Toggle the LED (see lab1)
  }
}
```

Brief description of Example 2

- Set up Timer 0 to run off the CPU Clock (CCLK)
- Match 0 is set to 2^23
- Ask Timer 0 to be reset on Match 0 and also an interrupt to be generated when Match 0 occurs
- The timer starts, counts from 0 to 2^23. At this point, match occurs. The timer is reset and the interrupt occurs
- Inside the interrupt handler, we check for the source of the interrupt (Timer 0 can produce interrupts from many sources like Mat0, Mat1 etc.) and then toggle the LED
- *Note: Because the start up code gets the chip running at 100Mhz by default, 1 tick or period of the timer = 1/100Mhz = 10 ns. Hence (2^23 + 1) ticks = 0.08388609 seconds. You should see the LED blinking every other 0.083 s.*
- *Assignment: Create a uVision project and verify this example during the next lab session*

Example 3: Blink LED

```
#include "LPC17xx.h“

volatile uint32_t del;
void my_software_delay(uint32_t delay);

int main (void)
{
  NVIC_EnableIRQ(TIMER0_IRQn); // Enable Timer 0 interrupt
  LPC_SC->PCONP |= ( 1 << 15 ); // Power up GPIO
  LPC_GPIO1->FIODIR |= 1 << 29; // Put P1.29 into output mode. LED is connected to P1.29

  while(1)
  {
    my_software_delay(1 << 24); // Wait for about 1 second
    // This is a “software interrupt” as we “call” it from within
    // the program; It is not triggered from the outside;
    NVIC_SetPendingIRQ(TIMER0_IRQn); // Software interrupt
  }
  return 0;
}
```

Example 3: Blink LED

```
void TIMER0_IRQHandler (void)
{
  LPC_GPIO1->FIOPIN ^= 1 << 29; // Toggle the LED
}
void my_software_delay(uint32_t delay)
{
  uint32_t i;
  for (i = 0; i < delay; i++) {
    del = i; // do this so that the compiler does not optimize away the loop;
  }
}
```

Outline

- Introduction
- NVIC and Interrupt Control
- Interrupt Pending
- Examples
- Interrupt Service Routines

Interrupt Service Routines (ISRs)

- When an interrupt/exception takes place, a number of things happen:

1. Stacking (automatic pushing of eight registers' contents to stack)
   - PC, PSR, R0–R3, R12, and LR
2. Vector fetch (reading the exception handler starting address from the vector table)
3. Exception vector starts to execute. On the entry of the exception handler, a number of regs are updated:
   - stack pointer (SP) to new location
   - IPSR (low part of PSR) with new exception number
   - program counter (PC) to vector handler
   - link register (LR) to special value EXC_RETURN

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

Interrupt/Exception Exits

- At the end of the exception handler, an exception exit (a.k.a interrupt return in some processors) is required to restore the system status so that the interrupted program can resume normal execution
- There are three ways to trigger the interrupt return sequence; all of them use the special value stored in the LR in the beginning of the handler:

Instructions that Can be Used for Triggering Exception Return

Return Instruction	Description
BX <*reg*>	If the EXC_RETURN value is still in LR, we can use the *BX LR* instruction to perform the interrupt return.
POP {PC}, or POP {...., PC}	Very often the value of LR is pushed to the stack after entering the exception handler. We can use the POP instruction, either a single POP or multiple POPs, to put the EXC_RETURN value to the program counter. This will cause the processor to perform the interrupt return.
LDR, or LDM	It is possible to produce an interrupt return using the LDR instruction with PC as the destination register.

Credits and references

- Joseph Jiu, The Definitive guide to the ARM Cortex-M3, 2007 (Chapters 7,8,9 and Appendices C,D)
- LPC17xx User's Manual (Chapters 6,23)