Lesson 11
Interrupt

---

1. Overview

In this lesson, we introduce the general concepts about interrupts supported by the Cortex-M3 processor. We will then discuss a specific interrupt example that utilizes an external triggered signal.

2. General Concepts of an Interrupt

One of the methods to read an input from an I/O port is using the polling technique as you did in lab 1. Polling is a technique to monitor an I/O port and to trigger an appropriate action when a change is detected. The general idea is that the processor periodically reads the I/O port and determines if there is a change. If there is a change, the processor will execute the subset of code to deal specifically with this change. Otherwise, the processor will continue with normal operations.

Another (more efficient?) way to read and input for an I/O port is using an interrupt. Interrupt-based program allows the processor to continue processing the main task without periodically checking for a change at an I/O pin. Interrupt is an exception caused by an explicit request signal from a peripheral or hardware device. An interrupt cause the automatic transfer of software execution outside of the normal programmed sequence. (e.g. to provide service to the peripheral). When a peripheral or a hardware device needs service form the processor, typically:

- It asserts an interrupt request to the processor,
- The processor completes the current instruction then it suspends the current task and jumps to an Interrupt Service Routine (ISR) to service the peripheral,
- Then, the processor resumes the previously suspended task.

The Cortex-M3 processor supports vector interrupts. It means that when an interrupt occurs, the program jumps to a specific memory location indicated by the Nested Vectored Interrupt Controller (NVIC). This controller provides an efficient way to handle different exceptions. Exceptions numbers 1-15 are system exceptions as shown in the table below.

Exceptions of number 16 or above are peripheral driven. We will mainly discuss these types of exceptions in class. Parts of the NVIC are discussed in this lesson.

In order for the processor to recognize an interrupt request form a peripheral, the peripheral and processor must be initialized and configured properly to enable the interrupt triggering mechanism. In general, the following conditions must be true for an interrupt to occur:

1: The peripheral is configured properly. This step varies based on the peripheral. We will discuss a specific example later in this lesson (*EINT0*).

2: The interrupt enable bit for the peripheral in the Interrupt Set-Enable Registers (*ISERn* registers) is set. **By default, all interrupt enable bits are cleared (disabled).**

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

From *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*, Joseph Yiu, Elsevier, 3nd ed, 2014.

3: The priority level for the peripheral is configured properly in the Interrupt Priority Registers (*IPRn* registers). In order for this interrupt to occur, this priority level must be higher than or the same as the priority level set in *BASEPRI* register. Note that the lowest number is the highest priority. **By default, the priority level is 0 in *BASEPRI* register. Also, by default the priority level is zero for all peripherals.** It means that default values for these registers are ok to use in your program without an modifications.

4: The global interrupt bit is enabled (bit 0 in *PRIMASK* register = 0). **By default bit 0 of *PRIMASK* is 0 (enabled)**. In general, you won't need to modify this register, but if necessary, you can use the following instructions to enable or disable the global interrupt bit:

You can also use the MRS instructions to enable/disable disable the global interrupt bit. For example,

or

5: The peripheral asserts the interrupt request which sets the interrupt flag.

Once an interrupt request has been asserted by the peripheral and recognized by the processor, the processor needs to service the peripheral which causes the following conditions:

1: Suspension of the main program
- Current instruction is completed
- Suspend execution and push 8 registers (R0-R3, R12, LR, PC, PCR) on the stack
- LR set to 0xFFFFFFF9 which indicates interrupt return
- IPSR set to interrupt number
- PC set to ISR address

2: The ISR is executed
- Process the interrupt request by the peripheral
- Clears the flag that requested the interrupt
- Exit ISR by executing BX LR

3: Resume normal operation
- Pulls 8 registers (R0-R3, R12, LR, PC, PCR) from the stack
- Return to the next instruction in the previously suspended task.

Interrupt Service Routine (ISR) is a subroutine that is executed when an interrupt request occurs. Generally, each potential source of interrupt would have a specific ISR. In most cases (except for the SysTick interrupt), the ISR must clear the flag that caused the interrupt. Failing to clear the flag will trigger continuous ISR execution (endless loop). After the ISR provides the necessary service, it will return to the main program by executing the BX LR instruction.

In the vector interrupt system, each source (or each peripheral or hardware device) of interrupt has an associated 32-bit vector that points to the address of the first instruction in the ISR that handles the exception. These vectors are stored at the beginning of the ROM. **Table 50** (shown below) in the *LPC17xx User Manual* contains the interrupt vector location for different interrupt source/peripheral. Note that the *Vector Offset* value is the offset number from the beginning of the memory space (ROM starts at address 0x00000000). When an interrupt occurs, the processor determines which exception number is activated, then calculates the starting address of the ISR. The address is then used to update the PC. For example, if an *External Interrupt 0* (*EINT0*) requests an interrupt, the processor calculates the PC as:

Interrupt ID	Exception Number	Vector Offset	Function	Flag(s)
0	16	0x40	WDT	Watchdog Interrupt (WDINT)
1	17	0x44	Timer 0	Match 0 - 1 (MR0, MR1) Capture 0 - 1 (CR0, CR1)
2	18	0x48	Timer 1	Match 0 - 2 (MR0, MR1, MR2) Capture 0 - 1 (CR0, CR1)
3	19	0x4C	Timer 2	Match 0-3 Capture 0-1
4	20	0x50	Timer 3	Match 0-3 Capture 0-1
5	21	0x54	UART0	Rx Line Status (RLS) Transmit Holding Register Empty (THRE) Rx Data Available (RDA) Character Time-out Indicator (CTI) End of Auto-Baud (ABEO) Auto-Baud Time-Out (ABTO)
6	22	0x58	UART1	Rx Line Status (RLS) Transmit Holding Register Empty (THRE) Rx Data Available (RDA) Character Time-out Indicator (CTI) Modem Control Change End of Auto-Baud (ABEO) Auto-Baud Time-Out (ABTO)
7	23	0x5C	UART 2	Rx Line Status (RLS) Transmit Holding Register Empty (THRE) Rx Data Available (RDA) Character Time-out Indicator (CTI) End of Auto-Baud (ABEO) Auto-Baud Time-Out (ABTO)
8	24	0x60	UART 3	Rx Line Status (RLS) Transmit Holding Register Empty (THRE) Rx Data Available (RDA) Character Time-out Indicator (CTI) End of Auto-Baud (ABEO) Auto-Baud Time-Out (ABTO)
9	25	0x64	PWM1	Match 0 - 6 of PWM1 Capture 0-1 of PWM1
10	26	0x68	I^2C0	SI (state change)
11	27	0x6C	I^2C1	SI (state change)
12	28	0x70	I^2C2	SI (state change)
13	29	0x74	SPI	SPI Interrupt Flag (SPIF) Mode Fault (MODF)

From *Table 50 in the LPC17xx User manual*, NXP Semiconductors, 2010.

14	30	0x78	SSP0	Tx FIFO half empty of SSP0 Rx FIFO half full of SSP0 Rx Timeout of SSP0 Rx Overrun of SSP0
15	31	0x7C	SSP 1	Tx FIFO half empty Rx FIFO half full Rx Timeout Rx Overrun
16	32	0x80	PLL0 (Main PLL)	PLL0 Lock (PLOCK0)
17	33	0x84	RTC	Counter Increment (RTCCIF) Alarm (RTCALF)
18	34	0x88	External Interrupt	External Interrupt 0 (EINT0)
19	35	0x8C	External Interrupt	External Interrupt 1 (EINT1)
20	36	0x90	External Interrupt	External Interrupt 2 (EINT2)
21	37	0x94	External Interrupt	External Interrupt 3 (EINT3). **Note:** EINT3 channel is shared with GPIO interrupts
22	38	0x98	ADC	A/D Converter end of conversion
23	39	0x9C	BOD	Brown Out detect
24	40	0xA0	USB	USB_INT_REQ_LP, USB_INT_REQ_HP, USB_INT_REQ_DMA
25	41	0xA4	CAN	CAN Common, CAN 0 Tx, CAN 0 Rx, CAN 1 Tx, CAN 1 Rx
26	42	0xA8	GPDMA	IntStatus of DMA channel 0, IntStatus of DMA channel 1
27	43	0xAC	I^2S	irq, dmareq1, dmareq2
28	44	0xB0	Ethernet	WakeupInt, SoftInt, TxDoneInt, TxFinishedInt, TxErrorInt, TxUnderrunInt, RxDoneInt, RxFinishedInt, RxErrorInt, RxOverrunInt.
29	45	0xB4	Repetitive Interrupt Timer	RITINT
30	46	0xB8	Motor Control PWM	IPER[2:0], IPW[2:0], ICAP[2:0], FES
31	47	0xBC	Quadrature Encoder	INX_Int, TIM_Int, VELC_Int, DIR_Int, ERR_Int, ENCLK_Int, POS0_Int, POS1_Int, POS2_Int, REV_Int, POS0REV_Int, POS1REV_Int, POS2REV_Int
32	48	0xC0	PLL1 (USB PLL)	PLL1 Lock (PLOCK1)
33	49	0xC4	USB Activity Interrupt	USB_NEED_CLK
34	50	0xC8	CAN Activity Interrupt	CAN1WAKE, CAN2WAKE

From *Table 50 in the LPC17xx User manual*, NXP Semiconductors, 2010.

Note that the ISR can reside anywhere in the code memory (ROM), so how do we manage the memory space so that the processor can point to the right address for the correct ISR? From a programming perspective, we can write the ISR as regular subroutine but with specific names. These predefined ISR names are provided in the *startupLPC17xx.s* file provided with the Keil uvision software. A screenshot of the *startupLPC17xx.s* showing some of the predefined ISR names is shown in the figure below. For example, if we want to create an ISR for EITN0, then the name for the ISR should be *EINT0_IRQHandler*. The compiler will place the memory address of this ISR in the appropriate offset in the vector table (0x88 in this case).

startup_LPC17xx.s

```
53
54  ; Vector Table Mapped to Address 0 at Reset
55
56                  AREA    RESET, DATA, READONLY
57                  EXPORT  __Vectors
58
59  __Vectors       DCD     __initial_sp              ; Top of Stack
60                  DCD     Reset_Handler             ; Reset Handler
61                  DCD     NMI_Handler               ; NMI Handler
62                  DCD     HardFault_Handler         ; Hard Fault Handler
63                  DCD     MemManage_Handler         ; MPU Fault Handler
64                  DCD     BusFault_Handler          ; Bus Fault Handler
65                  DCD     UsageFault_Handler        ; Usage Fault Handler
66                  DCD     0                         ; Reserved
67                  DCD     0                         ; Reserved
68                  DCD     0                         ; Reserved
69                  DCD     0                         ; Reserved
70                  DCD     SVC_Handler               ; SVCall Handler
71                  DCD     DebugMon_Handler          ; Debug Monitor Handler
72                  DCD     0                         ; Reserved
73                  DCD     PendSV_Handler            ; PendSV Handler
74                  DCD     SysTick_Handler           ; SysTick Handler
75
76                  ; External Interrupts
77                  DCD     WDT_IRQHandler            ; 16: Watchdog Timer
78                  DCD     TIMER0_IRQHandler         ; 17: Timer0
79                  DCD     TIMER1_IRQHandler         ; 18: Timer1
80                  DCD     TIMER2_IRQHandler         ; 19: Timer2
81                  DCD     TIMER3_IRQHandler         ; 20: Timer3
82                  DCD     UART0_IRQHandler          ; 21: UART0
83                  DCD     UART1_IRQHandler          ; 22: UART1
84                  DCD     UART2_IRQHandler          ; 23: UART2
85                  DCD     UART3_IRQHandler          ; 24: UART3
86                  DCD     PWM1_IRQHandler           ; 25: PWM1
87                  DCD     I2C0_IRQHandler           ; 26: I2C0
88                  DCD     I2C1_IRQHandler           ; 27: I2C1
89                  DCD     I2C2_IRQHandler           ; 28: I2C2
90                  DCD     SPI_IRQHandler            ; 29: SPI
91                  DCD     SSP0_IRQHandler           ; 30: SSP0
92                  DCD     SSP1_IRQHandler           ; 31: SSP1
93                  DCD     PLL0_IRQHandler           ; 32: PLL0 Lock (Main PLL)
94                  DCD     RTC_IRQHandler            ; 33: Real Time Clock
95                  DCD     EINT0_IRQHandler          ; 34: External Interrupt 0
96                  DCD     EINT1_IRQHandler          ; 35: External Interrupt 1
97                  DCD     EINT2_IRQHandler          ; 36: External Interrupt 2
98                  DCD     EINT3_IRQHandler          ; 37: External Interrupt 3
99                  DCD     ADC_IRQHandler            ; 38: A/D Converter
100                 DCD     BOD_IRQHandler            ; 39: Brown-Out Detect
```

3. Example of an Interrupt – EINT0 (External Interrupt)

Let's look at an example of an interrupt. The peripheral that we will use in this example is the External Interrupt 0 (EINT0).

Step 1: Setup the peripheral (EINT0)

Since this peripheral shares the same pin with other functions (e.g. GPIO P2.10), we will first need to set it up to function as an External Interrupt 0 Pin. This is done by

Next, we will need to set the pin to detect rising edge changes (for this example) at the pin. These edges will cause interrupts. This can be achieved by

Step 2: Enable EINT0 Interrupt

By default, all external interrupts are disabled. So, we need to enable specific interrupt so that the processor can service the peripheral when requested. We can enable EINT0 interrupt by

Step 3: Set the Priority Level for EINT0 (optional)

By default, all interrupts are set at level 0 which is the highest priority. This default priority is good to use in programs, so you don't have to do anything. However, if you want to set the priority of EINT0 to another level, you will need to

Step 4: Enable the Global Interrupt Bit (optional)

By default, the processor is configured to recognize any interrupts that are enabled (e.g. in step 2). It means that you don't have to do anything to enable this bit. This is indicated by bit 0 of the PRIMASK register. If this bit = 0, all interrupts can be activated if enabled in step 2. If this bit is 1, all exceptions (except NMI) are disabled.

Recall that you can use the following instructions to enable or disable the global interrupt bit:

Step 5: Develop an Interrupt Service Routine (ISR) for EINT0

In order for the processor to jump to right code for EINT0 interrupt, the ISR must be named as:

The first task the ISR should perform is to save the contents of all the registers (r4-r12) that will be used in this subroutine. For example, if your subroutine uses r4 and r5 as temporary storage, you should save these registers by

The next step is to process the interrupt request. This is an application dependent step. For example, we can write an application to keep a count of the number the INT0 pushbutton is pressed.

The next step is very important: clearing the interrupt flag. If your ISR returns without clearing the interrupt flag, the interrupt will endlessly occur (infinite loop). For EINT0, we can clear the interrupt flag by

Before returning to the main program, your subroutine should restore all the temporary registers (r4-r12) that are used in the ISR. For example, if registers r4 and r5 are pushed on the stack at the beginning of the ISR, these registers should be popped as:

The last instruction of the should be

Exercise: Write code to setup EINT0 interrupt. Keep a count of the number of interrupt occurrences.

Notes about debugging with interrupts:

- Cannot use single step
- Use breakpoints and run option instead

4. References

[1]. Joseph Yiu, *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*, Elsevier, 3rd ed, 2014.

[2]. Jonathan Valvano, *Introduction to ARM Cortex-M Microcontroller*, 4nd ed, 2013.

[3]. *ARMv7-M Architecture Reference Manual*, ARM Limited, 2010.

[4]. *LPC17xx User manual*, NXP Semiconductors, 2010.

[5]. *Cortex-M3 Technical Reference Manual*, ARM Limited, 2010.

```
/*----------------------------------------------------------------
    Lesson 11: EINT0 interrupt example
 *----------------------------------------------------------------*/

#include "LPC17xx.h"                    // Device header
#include "LED.h"

void __asm EINT0_IRQHandler(void);
void __asm EINT0_Init(void);

           // Reserve a 32-bit word at address0x10000000

/*----------------------------------------------------------
  Main: Initialize
 *----------------------------------------------------------*/
int main (void) {

     count = 0;

     LED_Init();     /* Initialize LEDs                  */
     EINT0_Init ();  /* Initialize EINT0 interrupt       */

                // endless loop -- do nothing, wait for interrupt

}
void __asm EINT0_Init(void)
{
// ------------------- Step 1 ----------------------------------
     //Setup pin to be EINT0 in PINSEL4 register
                          // R0= OR mask to set bit 20, other bits unchanged
                          // R0= AND mask to clear bit 21, other bits unchanged
                          // R2= Address of PINSEL register

                           // set bit 20
                           // clear bit 21
                           // write back to PINSEL4 register

     //Setup Mode to be edge in EXTMODE register
                          // R2= Address of EXTMODE register

                           // force bit 0 = 1 for edge
                           // write back to EXTMODE register

     //Setup Polarity to be rising edge in EXTPOLAR register
                          // R2= Address of EXTPOLAR register

                           // force bit 0 = 1 for rising edge
                           // write back to EXTPOLAR register
```

```
// ------ Step 2: Enable interrupt for peripheral (EINT0) in ISER0 reigster
                            // R0= OR mask to set bit 18, other bits unchanged
                            // R2= Address of ISER0 register
                           // force bit 18 = 1 to enable interrupt for EINT0
                           // write back to ISER0 register

// ----- Step 3: Setup priority for peripheral (EINT0) -- default ok

// ----- Step 4: Enable global interrupt bit -- default ok

                           // return
}

// ----- Step 5: ISR for EINT0-------------------------------------
void __asm EINT0_IRQHandler(void){

     // Save registers (r4-r12) if used in ISR

     // Processing: increment a count in count
                           // R0= address of count in data RAM

                            // increment count


     // Clear EINT0 interrupt flag in EXTINT register
                           // R0= address of EXTINT register

                           // clear EINT0 flag by writing 1 to bit 0

     // Pull registers (r4-r12) if used in ISR

                          // Return from ISR
}

/*-------------------------------------------------------------
 * end of file
 *-------------------------------------------------------------*/
```