10.Interrupts and ports

The interrupt mechanism is very important for an effective use of processor time and resources, and is implemented in every microcontroller. A simple example of the use of interrupts is given here.

10.1. The hardware – NVIC, the Nested Vectored Interrupt Controller

The processor executes a program line after line as it was written by the programmer. Jumps from one part of the program to another part are possible, and are preprogrammed or/and used to follow decisions based on the values of variables; they are predictable at the time of writing of the program.

Many times an important event outside of the processor requires immediate attention and action of the processor while it is executing a program. The situation where the execution of the program is suspended and the processor is forced to take care of the important event is called an interrupt. Such situation is initiated by an electrical signal called interrupt request. The interrupt request can be issued by a hardware built into the microcontroller, but can also be issued by an external hardware connected to a pin of a port. This example will show how to program a microcontroller to respond to the interrupt request caused by pressing a pushbutton connected to port E, pin 3.

A special block of hardware is built into the microcontroller in order to release the processor from complex tasks involved in handling interrupt requests. This hardware is called interrupt controller, and is capable of receiving an interrupt request signal, interpreting it as a justified reason to stop the processor from executing the regular program and actually forcing the processor to jump to and execute a special piece of code called interrupt function. It is the task of the processor to return to the regular program after the execution of the interrupt function and continue its execution as if nothing had happened.

In STM32F4xx series microcontrollers the interrupt controller is called Nested Vectored Interrupt Controller (NVIC, reference manual RM0090, chapter 12, page 369, and programming manual PM0214, chapters 2.3 and 4.3.7). The controller NVIC can handle interrupt requests caused by most hardware (like DMA requests, interrupt requests from communication channels USART, CAN, I2C, and timers)

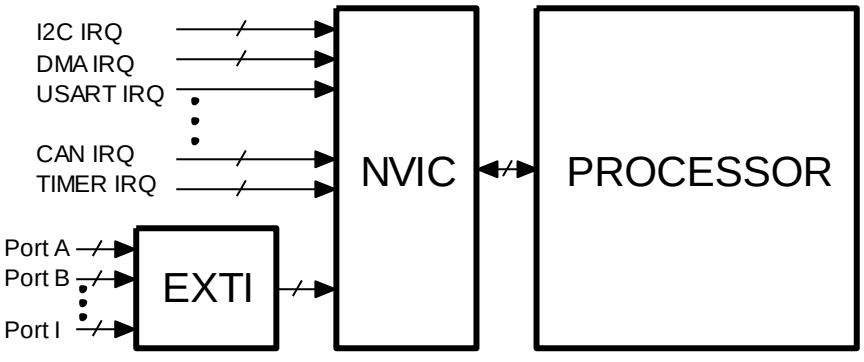

Figure 10.1: The chain involved in the processing of interrupt requests - simplified

built into the microcontroller alone, and also receives interrupt requests from ports through an additional hardware called External interrupt/event Controller (EXTI). A simplified block diagram for the processing of interrupt request signals in shown in Fig. 10.1.

In order to utilize the interrupt capabilities of the microcontroller the following must be fulfilled:

- The interrupt function must be prepared. This means that the programmer must know what to do in case of an interrupt request, and must write the code (interrupt function) to be executed due to the interrupt. The interrupt function must be written in a special way not to harm the execution of the regular program and the compiler must be notified that this is the function to be executed in case of a certain interrupt request on a certain event. The interrupt function cannot receive or return any arguments as normal functions do; the arguments used within the interrupt function must be declared as global at the beginning of the regular program.
- The interrupt controller NVIC must be configured and enabled. Since changing of the settings within the interrupt controller is a delicate task and the registers involved into configuration of this controller can be accessed only from a privileged state of the processor, a set of functions to change/configure the controller NVIC is included in a CMSIS library within the compiler package (“core_cm3.h”, “misc.c”, “misc.h”).
- The controller EXTI must be configured and enabled to sort-out the signals at ports and interpret them as interrupt requests.

Once the controller NVIC receives the interrupt request it validates it and notifies the processor. The processor then requests the code of the interrupt request source and uses it to find the pointer to the corresponding interrupt function from the interrupt vector table. The table is located in the memory of the microcontroller, and has more than 80 entries for vectors. The complete list of vectors is given in Table 62, RM0090, pg. 368. The interrupt requests have priorities, some are more important than others. The default priority for interrupts is given in column 2, but can be changed from software. From this table one can see that there are seven vectors reserved for interrupt requests from ports; these vectors are named EXTIx, where x stands for the number (EXTI0 as table entry number 6, priority 13, …). These seven vectors can serve 16 interrupt requests issued through ports; some vectors are shared by more than one interrupt request.

The vector table itself as used by the compiler can be seen in assembly language source file “startup_stm32f4xx.s” from line 57 on. Individual vectors to point to corresponding interrupt functions are placed into this table during the compilation process, and the functions to be executed on individual interrupt requests must have the same names as listed in this table. As an example, the interrupt function to be executed on external interrupt EXTI3 must have the name “EXTI3_IRQHandler”. The required names of interrupt functions for other interrupt request can be obtained from this vector table.

As stated the controller NVIC can be handled only in privileged mode, and a set of functions to change registers of the controller has been made available in CMSIS library to ease the setting of the controller. These are:

- NVIC_EnableIRQ (IRQ#): Enables the handling for the given interrupt request (IRQ#) as specified in Table 62, RM0090, page 3768, first column. The number IRQ# can also be replaced by a human-friendly name defined in file »stm32f4xx.h«, line 160 and on.
- NVIC_DisableIRQ (IRQ#): Disables the handling for the given interrupt request (IRQ#).
- NVIC_SetPendingIRQ (IRQ#): Sets the status of an interrupt request as pending.
- NVIC_ClearPendingIRQ (IRQ#): Clears the status of an interrupt request from being pending.

- NVIC_GetPendingIRQ (IRQ#): Inquires about the pending status of a given interrupt request (IRQ#) and returns a non-zero value if IRQ# is pending.
- NVIC_SetPriority (IRQ#, priority#): Sets the priority for the given interrupt request (IRQ#) with configurable priority level to value stated by priority#.
- NVIC_GetPriority (IRQ#): Reads the priority for the given interrupt request with configurable priority level.

The controller EXTI handles the processing of interrupt request signals from all pins of ports (144 of them, 16 pins by 8 ports in this microcontroller family) and concentrates the requests into 16 (23 including some extra interrupt request sources, see RM0090, chapter 12, for details) lines to be passed on to controller NVIC. The controller EXTI consists of 16 multiplexors and additional hardware, see fig. 10.2. (only three multiplexors are shown). Multiplexors concentrate interrupt request signals from pins into fewer lines EXTIx. Signals from pins 0 from all ports enter a multiplexor at the left-top, and one of them can be used as an EXTI0 interrupt request signal. Each multiplexer is controlled by four bits of register contained in system configuration hardware, which must be clocked in order to work. Signals from pins 1 from all ports enter the next multiplexer, and one of them can be used as an EXTI1 interrupt request signal. The same pattern repeats for other pins.

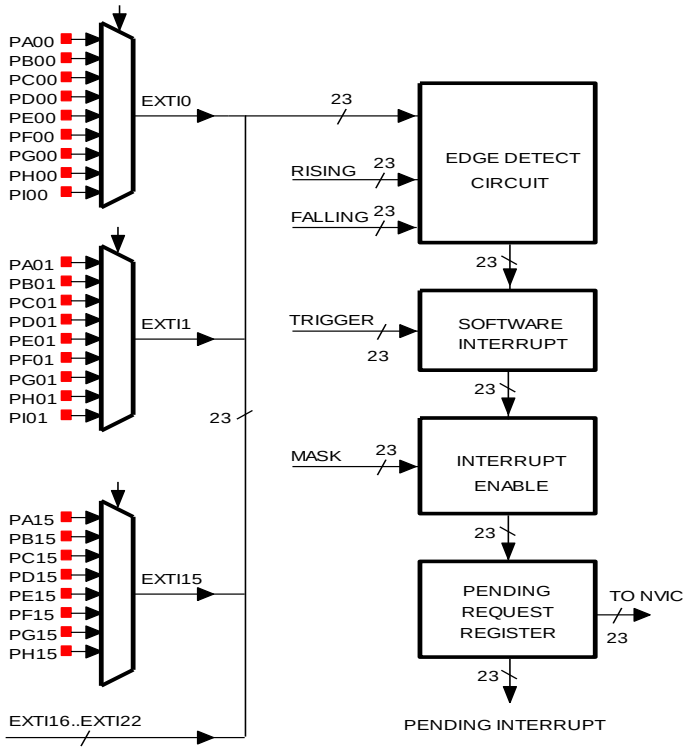

Figure 10.2: The controller EXTI, red dots represent input pins

The resulting bus EXTI is 23-bits wide and enters a circuit for edge detection; an interrupt can be issued on rising and/or falling edge of a signal, and the appropriate edge for each of the 23 signals is selected individually by setting signals 'RISING' and 'FALLING'.

An interrupt request can be also triggered by software, as indicated by the next block in the chain; setting a bit TRIGGER initiates the interrupt processing as if it would be initiated by hardware. However, not all interrupt request signals can actually trigger the interrupt. A mask to enable individual interrupt request signals in implemented in the next block. Only enabled interrupt requests can pass to the last block to be memorized before entering the interrupt controller NVIC.

Interrupt requests are prioritized; when processor is executing a high priority interrupt function other, lower priority requests, must wait. Those are called "pending interrupt requests". The software can check about such pending requests by investigating the 'PENDING INTERRUPT' signals exiting the last block in the chain. The same register also memorizes an interrupt request, and this register must be cleared from the interrupt function to avoid the repeated execution of the same interrupt function for one interrupt request.

10.2. The software to test the interrupt issued by pressing a pushbutton S370

A short program is presented to confirm the operation of interrupt hardware. In the main program the content of a variable is be continuously written to the LCD screen. When a user presses a pushbutton, a variable is incremented within the interrupt function. The pushbutton is connected to port E, pin 3.

The use of the interrupt mechanism is, as far as programming is concerned, split into two parts: function for the configuration of the interrupt hardware, and the actual interrupt function.

- The function for the configuration of the hardware is named “IRQinit_PE3()” in this example and is given below.

```
void IRQinit_PE3 (void) {
EXTI_InitTypeDef        EXTI_InitStructure;

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);                              // 4
  SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOE, EXTI_PinSource3);                       // 5

  EXTI_InitStructure.EXTI_Line    = EXTI_Line3;                                       // 7
  EXTI_InitStructure.EXTI_Mode    = EXTI_Mode_Interrupt;                              // 8
  EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;                              // 9
  EXTI_InitStructure.EXTI_LineCmd = ENABLE;                                           // 10
  EXTI_Init(&EXTI_InitStructure);                                                     // 11

  NVIC_EnableIRQ(EXTI3_IRQn);                 // Enable IRQ for ext. signals, lines 3         // 13
}
```

The registers holding the configuration for multiplexors are configured first, and those registers are located in microcontroller block ‘SYSCFG’; this block needs a clock signal first; it is enabled in the fourth line of the function. Once the clock is present the multiplexor for signal EXTI3 (for pins 3, not shown in Fig. 10.2) can be configured to pass the signal from Port E. This is done by function call in the fifth line of the function, where ‘EXTI_PortSourceGPIOE’ is selects the ‘PE’ input of the multiplexor for EXTI3. This finishes the configuration of the multiplexor.

Other four blocks are best configured using a CMSIS function “EXTI_Init()”, described in the source file “stm32f4xx_exti.c”, and having its data structures and definitions described in the header file “stm32f4xx.exti.h”. Lines 7 to 10 are used to initialize the members of the data structure ‘EXTI_InitStructure’, which is declared in the second line of the function. The pointer to this data structure is the argument of the function call in the 11th line:

- The first member ‘.EXTI_Line’ selects one of the EXTIx signals to be affected by the call to the “EXTI_Init()” function.
- The second member ‘EXTI_Mode’ selects the configuration of interrupt or event structure, see the RM0090 for details.
- The third member ‘EXTI_Trigger’ selects the edge to trigger the interrupt request. It can be either ‘EXTI_Trigger_Rising’, ‘EXTI_Trigger_Falling’ or ‘EXTI_Trigger_Rising_Falling’.
- The last member ‘.LineCmd’ sets the mask to be used and actually enables or disables the EXTI line.

The NVIC controller is configured next. Since the only interrupt request is coming from the pushbutton, there is no need to set its priority, and we only need to enable it. This is done in line 13 with a function call “NVIC_EnableIRQ()” with an argument ‘EXTI3>_IRQn’.

The interrupt function to be executed on the press of a pushbutton is given in the listing bellow:

```
void EXTI3_IRQHandler (void)   {
  IRQcounter++;                              // do the actual work                          // 2
  EXTI_ClearITPendingBit(EXTI_Line3);                                                    // 3
  EXTI_ClearFlag(EXTI_Line3);                                                            // 4
}
```

The function must not receive or return any arguments, and it is declared as 'void' for both. Its name must be as defined in the vector table located in the file "startup_stm32f4xx.s". The body of the interrupt function starts in line 2, where the counter variable 'IRQcounter' is incremented. Remember: this must be a globally declared variable. Two bits in the last register from Fig. 10.2 must still be cleared to confirm the execution of the interrupt function for this interrupt request. The two bits are reset by calling two CMSIS functions at lines 3 and 4.

In order to use the pushbuttons the port E must be declared as input with pull down resistors at lines PE3 to PE7. A function "SWITCHinit()" to do this has been presented already, and must be included in the final listing, given below.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_exti.c"
#include "stm32f4xx_syscfg.c"
#include "LCD2x16.c"

int IRQcounter = 0;                          // declare & initialize IRQ counter            // 8

void main (void){

  LCD_init();                                // init LCD                                    // 12
  LCD_string("IRQs at PE3:", 0x00);          // display title string                        // 13

  SWITCHinit();                                                                          // 15
  IRQinit_PE3();                                                                         // 16

  while (1) {
    LCD_uInt16(IRQcounter,0x40,1);           // write IRQ count to LCD                      // 19
    for (int i = 0; i<100000; i++) {};       // waste some time                             // 20
  };
}
```

The listing commences with several include commands. Note the "stm32f4xx_exti.c" and "stm32f4xx_syscfg.c" for CMSIS function related to the newly used peripherals. A global variable 'IRQcounter' is declared and initialized in line 8. The main part of the program starts with a call to the initializing function for the LCD and by writing of the introductory string to the LCD, lines 12 and 13. Next port E is initialized for pushbuttons, and interrupts are initialized and enabled. This part is followed by an infinite 'while' loop, where the content of the variable 'IRQcounter' is periodically written to the LCD in line 19. A 'for' loop is added to slow down the execution of the program.