

EasyElectronics.ru Community

[All](#) [Collective](#) [Personal](#) [TOP](#)
[Good](#) [Bad](#)

Поиск

8L-Course, Part 3 - Interruptions, EXTI

STM8

[← Part 2 — GPIO Contents](#) [Part 4 — Clocking →](#)

We already know how to work with GPIO. Set the required levels and read the states of pins configured for input. But what if you need to not just read the state of a pin, but quickly respond to a level change? And what if there are several such pins? Here the external interrupt system — **EXTI (EXTERNAL Interrupts)** — comes to our aid. It is quite flexible in configuration and allows you to receive an interrupt from any pin of the microcontroller, which is incredibly convenient — you do not need to adjust the board layout for a pair of special INT pins, as was the case in most AVR.

Before we deal with external interrupts, we need to understand how the interrupt subsystem in STM8 works.

The interrupt controller (**ITC**) in STM8 supports nested interrupts and divides them by priorities. This means that if we are currently executing a handler for some not very important interrupt (for example, from a timer), and suddenly an interrupt with a higher priority occurs, then the interrupt from the timer will be interrupted and the MCU will immediately rush to process another one. But if the priorities are the same (or the priority of the new one is lower than the one currently being processed), the new interrupt will only start to be processed after the previous one has completed.

But what happens if two interrupts occur simultaneously? Or if several interrupts occur while a more important one is being processed? Then the one with the higher priority will be processed first. But what if the priorities are the same? - the meticulous reader will ask. Then the interrupt that is closer to the beginning in the interrupt table will be executed first.

Live

[Comments](#) [Publications](#)

penzet → [Sprint Layout in OS X 18](#) → [Software for electronics engineer](#)

Vga → [EmBitz 6](#) → [Software for electronics engineer](#)

Vga → [Rail-to-rail: ideal operational amplifier or a clever marketing ploy? 1](#) → [Theory, measurements and calculations](#)

Flint → [A little more about 1-wire + UART 56](#) → [Hardware connection to the computer.](#)

penzet → [Cross-platform terminal - SerIO 3.x.25](#) → [Software for electronics engineer](#)

From Gorn → [PIP regulator 2](#) → [Algorithms and software solutions](#)

whoin → [CC1101, Treatise on tracing 89](#) → [Blog im. Khomin](#)

OlegG → [Clock on Bluetooth LE module 8](#) → [Cypress PSoC](#)

Technicum505SU → [Nuances of PWM control of a DC motor by a microcontroller 3](#) → [Theory, measurements and calculations](#)

sunjob → [DDS synthesizer AD9833 88](#) → [Blog named after grand1987](#)

dihalt → [W801, LCD screen and tar spoon 2](#) → [Detail](#)

nictrace → [New Arduino-compatible board. 20](#) → [FPGA](#)

sunjob → [Connecting the ARM GCC compiler to CLion 1](#) → [Software for electronics engineers](#)

Vga → [CRC32: on STM32 as on PC or on PC as on STM32. 58](#) → [STM32](#)

dmitrij999 → [Capturing images from a USB camera using STM32 6](#) → [STM32](#)

Gilak → [Expanding the capabilities of a simple MC up to an ADC on 2 or 1 pin. 8](#) → [Theory, measurements and calculations](#)

sva_omsk → [Lithium ECAD - Russian PCB CAD 40](#) → [Software for electronics engineer](#)

x893 → [W801 - budget controller with Wi-Fi 4](#) → [Details](#)

podkassetnik → [Changing the standard instrument cluster lighting of Logan-like cars 5](#) → [Automotive electronics](#)

Table 10. Interrupt mapping

IRQ No.	Source block	Description	Wakeup from Halt mode	Wakeup from Active-halt mode	Wakeup from Wait (WFI mode)	Wakeup from Wait (WFE mode) ⁽¹⁾	Vector address
	RESET	Reset	Yes	Yes	Yes	Yes	0x0
	TRAP	Software interrupt	-	-	-	-	0x0
0	Reserved						0x0
1	FLASH	EOP/WR_PG_DIS	-	-	Yes	Yes ⁽²⁾	0x0
2	DMA1 0/1	DMA1 channels 0/1	-	-	Yes	Yes ⁽²⁾	0x0
3	DMA1 2/3	DMA1 channels 2/3	-	-	Yes	Yes ⁽²⁾	0x0
4	RTC	RTC alarm interrupt	Yes	Yes	Yes	Yes	0x0
5	EXTI E/F/PVD ⁽³⁾	PortE/F interrupt/PVD interrupt	Yes	Yes	Yes	Yes ⁽²⁾	0x0
6	EXTIB	External interrupt port B	Yes	Yes	Yes	Yes ⁽²⁾	0x0
7	EXTID	External interrupt port D	Yes	Yes	Yes	Yes ⁽²⁾	0x0
8	EXTI0	External interrupt 0	Yes	Yes	Yes	Yes ⁽²⁾	0x0
9	EXTI1	External interrupt 1	Yes	Yes	Yes	Yes ⁽²⁾	0x0
10	EXTI2	External interrupt 2	Yes	Yes	Yes	Yes ⁽²⁾	0x0
11	EXTI3	External interrupt 3	Yes	Yes	Yes	Yes ⁽²⁾	0x0
12	EXTI4	External interrupt 4	Yes	Yes	Yes	Yes ⁽²⁾	0x0
13	EXTI5	External interrupt 5	Yes	Yes	Yes	Yes ⁽²⁾	0x0
14	EXTI6	External interrupt 6	Yes	Yes	Yes	Yes ⁽²⁾	0x0
15	EXTI7	External interrupt 7	Yes	Yes	Yes	Yes ⁽²⁾	0x0
16	LCD	LCD interrupt	-	-	Yes	Yes	0x0
17	CLK/TIM1/DAC	System clock switch/CSS interrupt/TIM1 Break/DAC	-	-	Yes	Yes	0x0
18	COMP/ADC1	Comparator interrupt/ADC1	Yes	Yes	Yes	Yes ⁽²⁾	0x0
19	TIM2	Update /Overflow/Trigger/Break	-	-	Yes	Yes ⁽²⁾	0x0
20	TIM2	Capture/Compare	-	-	Yes	Yes ⁽²⁾	0x0
21	TIM3	Update /Overflow/Trigger/Break	-	-	Yes	Yes ⁽²⁾	0x0
22	TIM3	Capture/Compare	-	-	Yes	Yes ⁽²⁾	0x0
23	TIM1	Update /Overflow/Trigger/COM	-	-	-	Yes ⁽²⁾	0x0
24	TIM1	Capture/Compare	-	-	-	Yes ⁽²⁾	0x0
25	TIM4	Update/overflow/trigger	-	-	Yes	Yes ⁽²⁾	0x0
26	SPI1	End of Transfer	Yes	Yes	Yes	Yes ⁽²⁾	0x0
27	USART 1	Transmission complete/transmit data register empty	-	-	Yes	Yes ⁽²⁾	0x0
28	USART 1	Receive Register Data full/overflow/idle line detected/parity error	-	-	Yes	Yes ⁽²⁾	0x0
29	I ² C1	I ² C1 interrupt ⁽⁴⁾	Yes	Yes	Yes	Yes ⁽²⁾	0x0

There are 29 interrupt vectors in total + two without a number: RESET — MCU reset, and TRAP — software interrupt (called by the TRAP assembler command). Strictly speaking, there are two types of priorities — software (which we can configure for each interrupt) and hardware — this is exactly the order of vectors in the interrupt table. The table shows that the FLASH memory interrupt has the highest hardware priority, and the I2C interrupt is generally neglected, being at the very end of the table. However, no one forbids making it the most important by setting a high software priority — it is more important than the hardware one. The

TRAP interrupt does not care about any software priorities at all. Regardless of the priority of the current interrupt, executing the TRAP command immediately takes us to the handler.

And for other interrupts, you can configure the priority through the ITC_SPR1 — ITC_SPR8 registers.

Vga → [ROPS \(Rem Object Pascal Script\) - embedded interpreter of the Pascal language. Plugin PSImport_Classes 3 → Algorithms and software solutions](#)

[Full broadcast](#) | [RSS](#)

1-Wire The other arduino ARM
Assembler Atmel AVR C++ compel
DIY enc28j60 ethernet FPGA gcc I2C
AND KEIL LaunchPad LCD led linux
LPCXpresso MSP430 npx PCB PIC
pinboard2 RS-485 RTOS STM32
STM8 STM8L OF UART USB
algorithm assembler ADC the library
power unit detail display an idea tool
competition competition 2 ANGRY
microcontrollers for beginners review
Debug board soldering iron
printed circuit board salary FPGA crafts
purchases programmer programming
Light-emitting diode software scheme
circuit design Technologies
smart House photoresist it's free crap
Times humor

Blogs

Group

AVR	38.98
STM8	37.92
Garbage truck 🗑️	29.53
STM32	28.46
Detail	24.63
Connection of hardware to the computer.	24.04
Circuit design	18.15
Smart House	17.75
MSP430	17.13
LPC1xxx	14.79

[All blogs](#)

	7	6	5	4	3	2
ITC_SPR1	VECT1SPR[1:0]	VECT2SPR[1:0]	VECT1SPR[1:0]			
ITC_SPR2	VECT7SPR[1:0]	VECT6SPR[1:0]	VECT5SPR[1:0]			
ITC_SPR3	VECT11SPR[1:0]	VECT10SPR[1:0]	VECT9SPR[1:0]			
ITC_SPR4	VECT15SPR[1:0]	VECT14SPR[1:0]	VECT13SPR[1:0]			
ITC_SPR5	VECT19SPR[1:0]	VECT18SPR[1:0]	VECT17SPR[1:0]			
ITC_SPR6	VECT23SPR[1:0]	VECT22SPR[1:0]	VECT21SPR[1:0]			
ITC_SPR7	VECT27SPR[1:0]	VECT26SPR[1:0]	VECT25SPR[1:0]			
ITC_SPR8	Reserved			VECT29SPR[1:0]		
	rw			rw	rw	

Each vector is allocated two bits here, which gives 4 possible values. But in fact, there are only 3 software priorities:

- 01 - Самый низкий
- 00 - Средний
- 11 - Самый высокий

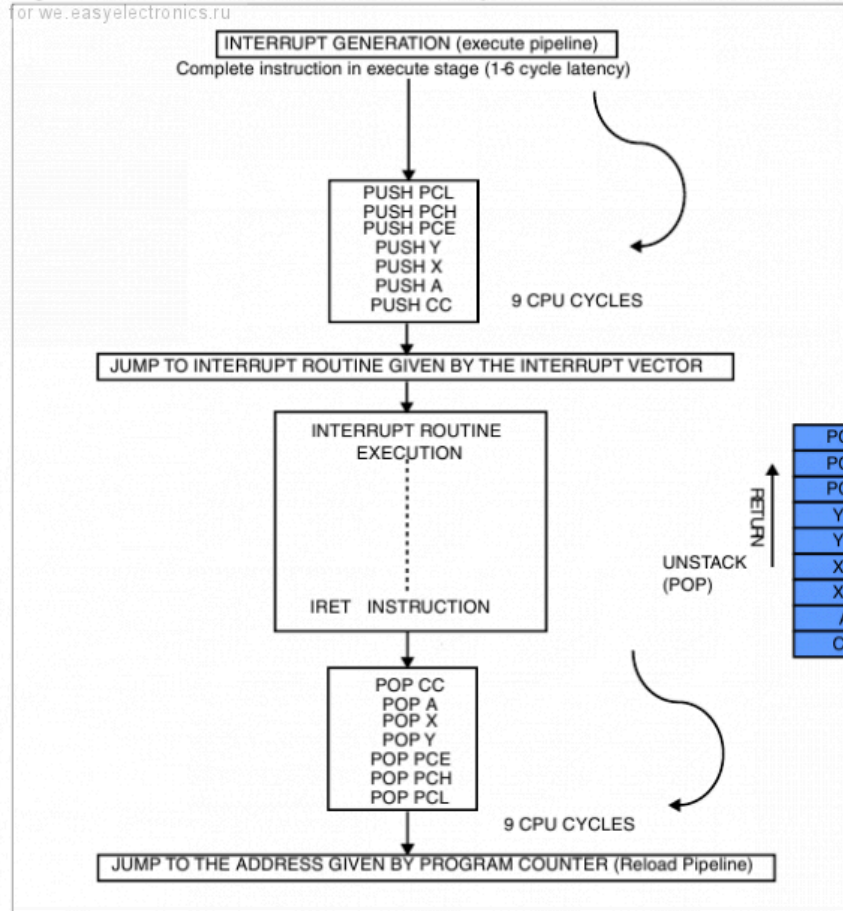
The value 10, which is not in the list, corresponds to the priority of the main program, and cannot be set. And it makes no sense, if you remember how the controller handles the interrupt queue: an interrupt with the same priority as the main program will never interrupt it.

After the necessary interrupts are enabled and configured, **interrupt processing must be globally enabled**. In STM8, this is done by the RIM assembler command (and SIM for interrupt disabling). And they work in a very interesting way. Instead of setting some flag to enable/disable interrupts (as was the case in AVR, for example), here they change the priority of the code that is currently being executed. The RIM command sets a low priority (10, the same one that cannot be set for interrupts), and now any interrupt can interrupt program execution. And the SIM command sets the current code to the highest priority, and no interrupt can interfere with the execution of the code.

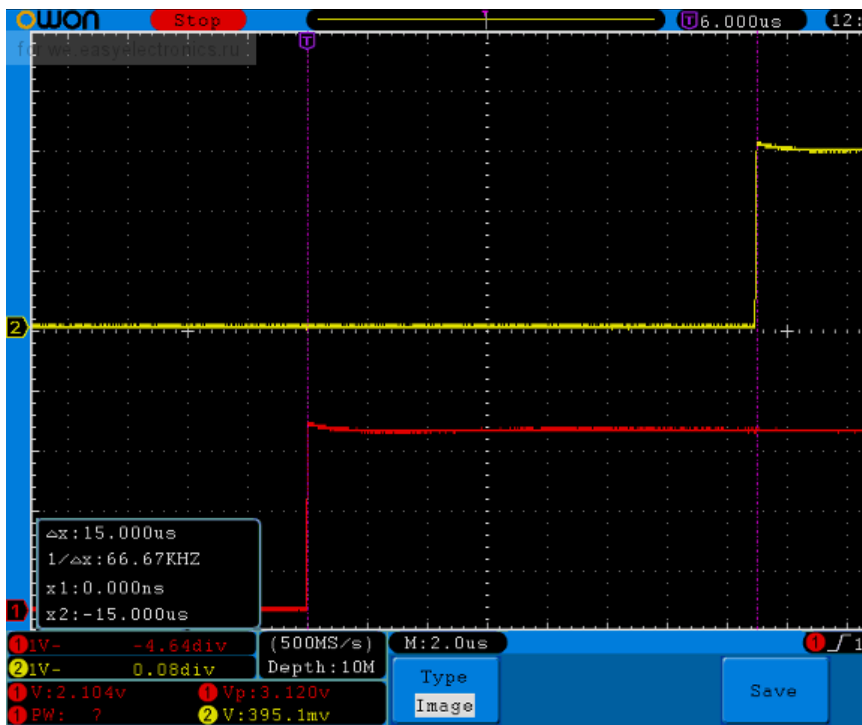
Each interrupt has its own flag, which indicates that the event has happened and it's time to run to the handler. In STM8, these flags, as a rule, are not reset by themselves and have to be lowered manually. Otherwise, after exiting the handler, we will get into it again. So what? The flag is raised, which means that the interrupt must be processed.

However, resetting the flags is organized differently in different cases. Sometimes you need to reset the flag yourself, sometimes you need to read a certain register (for example, in an interrupt from an ADC, you need to read the measurement result). Further, when describing different peripherals, I will provide handler templates and indicate how exactly the interrupt flags are reset there.

By the way, how quickly does the MC jump to the interrupt handler?
Here it is:

Figure 2. Context save/restore for interrupts

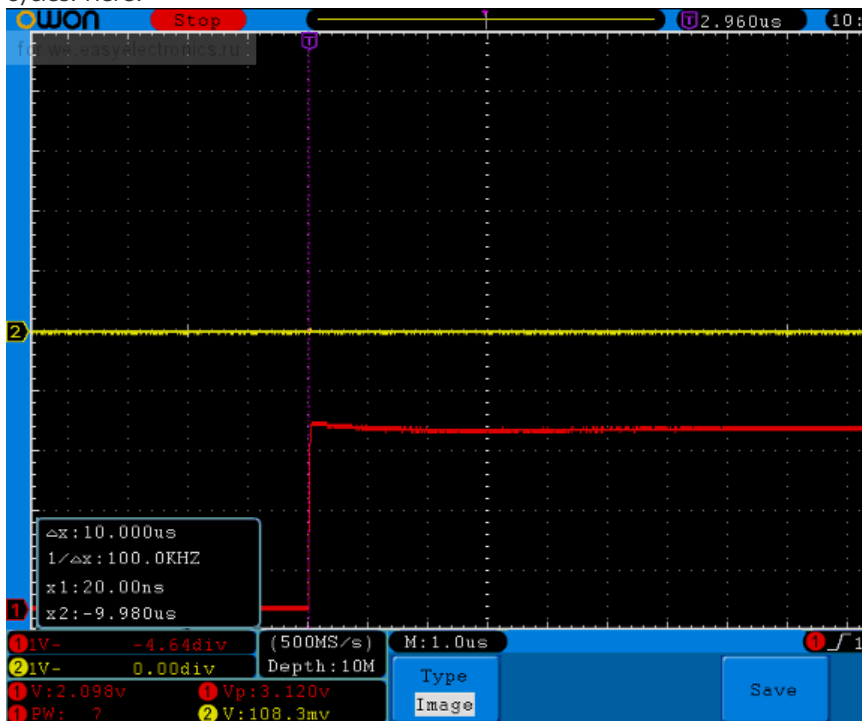
According to this scenario, an interrupt is processed. Any. First, the core completes the execution of the current command. This requires from 1 to 6 clock cycles. Then the address for returning from the interrupt and the processor registers are pushed onto the stack so that after exiting the handler, their original value is returned. This takes another 9 clock cycles. After which the transition to the address stored in the interrupt table occurs. The documentation does not say how many clock cycles this takes. But if you measure the time from the occurrence of an event that causes an interrupt to the transition to the handler, you get this:



Here, the red channel is attached to the pin that is raised by the event from the timer, and the yellow one is attached to the pin that is raised in the interrupt by this event (the very first command). The microcontroller operates at a frequency of 1 MHz, and a delay of 15 μ s corresponds to 15 clock cycles.

The interrupt entry time can be reduced by using the WFI mode — Wait For Interrupt. This is a power-saving mode that is designed to replace interrupt wait delays. In WFI, the core is turned off, while the rest of the peripherals and the clock generator continue to work (the enabled clock generator allows you to quickly exit sleep mode — no time is wasted on starting it). At the same time, before turning off the core, the current address and processor registers are saved in the stack — preparation for entering the interrupt.

When an interrupt occurs in WFI mode, the MCU goes to the handler without unnecessary delays. In practice, the interrupt entry time is reduced by 5 clock cycles. Here:



To enter WFI, you need to execute the WFI assembler command (Captain Obvious actively helped write the article)

```
asm("WFI");
```

Now the core is stopped until the next interrupt. Just don't forget to configure and enable this interrupt, otherwise you might not wake up :)

Well, now we have some idea of how interrupts work in STM8. Let's smoothly move on to **EXTI** - *EXT*ernal *I*nterrupts, that is, external interrupts.

If you look at the interrupt table, you can see that 11 vectors are allocated to EXTI.

Three of them have a letter in their name that suspiciously resembles the port name (EXTIE/F, EXTIB...). And the remaining 8 have a number that is no less suspiciously similar to the pin number. RM0031 dispels all our doubts and confirms our suspicions - due to the fact that there simply aren't enough interrupt vectors for all the pins, the system is organized in a very intricate way. Let

's start analyzing it from the very beginning. For each pin, you can individually enable or disable an interrupt. This is done, if you remember the previous part, through the Px_CR2 register, where x is the letter of the port (before this, you need to configure the pin to be an input). This way, you can enable interrupts only for the necessary pins, and the rest will not interfere in any way.

Next, for each half of the port (that is, for 4 junior pins and 4 senior pins), we can choose where the signal will go - to interrupts of pins (EXTI0, EXTI1 ...), or ports (EXTIB, EXTID ...). In the first case, we get an individual interrupt for each pin. But interrupts from other ports will come here too. In the second case, we will have one handler for the signal from all four pins (if we enable interrupts from all of them, of course), but an individual one for each port. All this is configured through the EXTI_CONF1 and EXTI_CONF2 registers:

12.9.9 External interrupt port select register (EXTI_CONF1)

for we.easyelectronics.ru

Address offset: 0x05

Reset value: 0x00

7	6	5	4	3	2	1
PFES	PFLIS	PEHIS	PELIS	PDHIS	PDLIS	PBHIS
rw	rw	rw	rw	rw	rw	rw

Each bit here corresponds to the senior or junior half of a port. For example, PDLIS (Port D Low Interrupt select) is responsible for pins 0..3 of port D.

Setting the bit to one means that this half of the port will give one interrupt to all pins (for example, EXTID). And zero, accordingly, means that each of the four pins will give its own interrupt from the EXTI0..EXTI7 group.

In addition, for each vector (namely the vector, and not a separate pin), you can configure the edge at which the interrupt is triggered. This is done through the EXTI_CR1 - EXTI_CR4 registers.

12.9.3 External interrupt control register 1 (EXTI_CR1)

for we.easyelectronics.ru

Address offset: 0x00

Reset value: 0x00

7	6	5	4	3	2	1
P3IS[1:0]	P2IS[1:0]	P1IS[1:0]	P0IS[1:0]			
rw	rw	rw	rw			

Each interrupt is allocated two bits here. Accordingly, 4 interrupt operation modes are available:

00 - Triggered by a transition from 1 to 0 (falling edge) and by a low level. That is, the handler will be called constantly while the pin is low.

- 01 - By transition from 0 to 1 (rising edge)
- 10 - Again by the falling edge, but without a low level
- 11 - By both edges. In other words, it is triggered by any change in level.

The names for the bits are formed simply. P1IS — Pin 1 Interrupt Sensitivity, PBIS — Port B Interrupt Sensitivity, and so on.

The interrupt flags (which I wrote above that they need to be reset) are in the EXTI_SR1 and EXTI_SR2 registers.

12.9.7 External interrupt status register 1 (EXTI_SR1)

for we.easyelectronics.ru

Address offset: 0x03
Reset value: 0x00

7	6	5	4	3	2	1
P7F	P6F	P5F	P4F	P3F	P2F	P1F
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

12.9.8 External interrupt status register 2 (EXTI_SR2)

for we.easyelectronics.ru

Address offset: 0x04
Reset value: 0x00

7	6	5	4	3	2	1
Reserved		PHF ⁽¹⁾	PGF ⁽¹⁾	PFF	PEF	PDF
		rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

One bit per vector. And they need to be reset by writing a one (and not a zero, as Captain Obvious might think).

This is such a tricky system. At first, it may seem that it is not very convenient to work with. But in fact, this inconvenience manifests itself only if you need to catch interrupts from a whole bunch of pins at once (and this is not often needed). In ordinary tasks, no special inconveniences arise.

As for **the STM8S**, there is not such a confusing situation with interrupts (but there are fewer options). In fact, there are only interrupts from ports. Like PCINT in AVR. Signals from each of the pins of a given port converge on one vector. You can enable interrupts individually for each pin, but the front is configured only for each vector - i.e. for the entire port.

Let's make up another useless example to understand how it works in practice. Let's take the example from the previous part (where there was an indicator) as a hardware basis and screw another button to it. Each button will have an interrupt hanging on it.

In the handler of the first interrupt, we will switch the numbers on the indicator in a cycle (yes, I'm strange). And in the handler of the second (which we will set a higher priority), we will blink the LED.

If you hold down the first button, the MCU will hang in the interrupt handler and start sorting through the numbers. But when you press the second button, the first interrupt will be interrupted by a more important one, and the LED will start blinking. If you release the second button, the indicator will continue to sort through the numbers from the one it stopped on - the handler of the first interrupt will continue to run. Simple, clear.

First, you need to configure the pins. Let the first button hang on D6, and the second on D7 (these pins are most conveniently connected to the buttons on the Pinboard).

```
PD_CR1_bit.C16 = 1; //На вход..
PD_CR2_bit.C26 = 1; //Прерывание разрешено

PD_CR1_bit.C17 = 1;
PD_CR2_bit.C27 = 1;
```

We do not touch the DDR register - it is already zeroed at the start of the MC.

In the EXTI_CONF1 register (which distributes signals to the interrupts of pins/ports) at the start there are also zeros, which means that each pin will give us a separate interrupt. We will leave it like this.

We also need to configure the front, which will trigger the interrupt. When the buttons are pressed, they are closed to ground, which means that we must catch the trailing edge, which corresponds to the value 2 (10):

```
EXTI_CR2_bit.P6IS = 2;
EXTI_CR2_bit.P7IS = 2;
```

And finally, we need to lower the priority of the interrupt from the first button (D6), so that the interrupt from the second can interrupt the handler of the first:

```
ITC_SPR4_bit.VECT14SPR = 0;
```

Look at the interrupt table - the EXTI6 vector is number 14 and it corresponds to the VECT14SPR group. The value 0, let me remind you, means "medium" priority.

The indicator and LED settings have migrated from previous examples, so I will not provide them here - look at the source codes.

All that remains is to globally enable interrupts. This is done with the RIM assembler command

```
asm("RIM");
```

At this point we are done with main() and move on to interrupt handlers. In IAR, a handler is declared in this tricky way:

```
#pragma vector=[Номер вектора]
__interrupt void [Название обработчика](void)
```

To avoid having to look in the datasheet for the number of the vector you need, you can take the defines from iostm8l151k6.h. All vectors are written at the very end of the file. Here is a snippet for example:

```
#define EXTI6_vector      0x10
#define EXTI7_vector      0x11
#define CLK_CSS_vector    0x13
#define CLK_SWITCH_vector 0x13
#define TIM1_BIF_vector    0x13
#define COMP_EF1_vector    0x14
#define COMP_EF2_vector    0x14
#define TIM2_OVR_UIF_vector 0x15
```

Oh, and why does the EXTI6 vector correspond to number 0x10 (16), and not 14, as in the table from the datasheet? It's just that the first two interrupts are taken into account here - RESET and TRAP. And in the table they are listed without numbers.

And the handler name is just the name of the function, and there are no special requirements for it.

It turns out that we can designate the EXTI6 interrupt handler as follows:

```
#pragma vector=EXTI6_vector
__interrupt void Pin6_interrupt(void)
{
    ...
}
```


Those who like brevity can, if they wish, collapse this into a defin:

```
#define STRINGVECTOR(x) #x
#define ISR( a, b ) \
    _Pragma( STRINGVECTOR( vector = (b) ) ) \
    __interrupt void (a)( void )
```

Then interrupts will be designated as follows:

```
ISR(Pin6_interrupt, EXTI6_vector)
{
    ...
};
```

Which, you must admit, looks nicer.

The interrupt handlers in our example will be almost identical (except that in the first one the numbers are switched, and in the second one the diode is blinking). Therefore, I will show only the handler of the first button:

```
ISR(Pin6_interrupt, EXTI6_vector)
{
    while (PD_IDR_bit.IDR6 == 0)
    {
        value++; //Инкрементируем счетчик
        if (value==10) value=0; //Проверяем - не ушел ли он за предел
        PB_ODR = numbers[value]; //Выводим число на индикатор
        SomeDelay();
    };

    EXTI_SR1_bit.P6F = 1; //Перед выходом из прерывания необходимо вручную
    //Иначе тут-же вернемся обратно в обработчик.
};
```

I think everything should be clear here. In the second handler, we blink the LED on D5 until a high level appears on pin D7.

That seems to be all about external interrupts.



Finally, I'll tell you about an interesting feature called Activation Level, which is also related to interrupts.

Quite often it turns out that the entire program (except for the initial setup) is scattered across interrupt handlers. And in main there is one pathetic endless loop-stopper. We have just such a case. And if our device is battery-powered

and needs to save energy, then between interrupts it will be in sleep mode and exit it only to execute handlers.

The guys from STM decided to make our life easier when building such algorithms and sawed off the AL bit. It is located in the CFG_GCR register (Configuration — General Configuration Register).

1.3.3 Description of global configuration register (CFG_GCR)

Address offset: 0x00
Reset value: 0x00

7	6	5	4	3	2	1
Reserved						AL
						rw

If you write a one to it and go into sleep mode, then after the MC wakes up on interrupt and executes the handler, it will not get to main() but will immediately fly back into sleep mode. We don't really know anything about sleep modes yet, but we want to try this feature.

Let's replace the stub cycle at the end with this construction:

```
CPU_CFG_GCR_bit.AL = 1;  
  
asm("halt");
```

HALT is the deepest power saving mode, in which the clock generator is disabled, and wake-up is possible only by an external interrupt, an interrupt from a pair of interfaces or RTC.

After launch, the program's behavior does not change in any way, but there is no longer any delay cycle at the end of main() and the core does not waste energy. Immediately after the handler completes, the MCU goes into sleep mode.

Example sources

← Part 2 — GPIO Contents Part 4 — Clocking →

STM8 , STM8L, get out interruptions

+1109 February 2013, 14:35dcdoder1


Files in topic: 3 GPIO Interrupts.zip

Comments (4)

RSS Collapse / Expand

Oh, why does the EXTI6 vector correspond to number 0x10 (16), and not 14, as in the table from the datasheet? It's just that the first two interrupts are taken into account here - RESET and TRAP. And in the table they are listed without numbers.

when I first came across this, I cursed a lot... What is the secret meaning of this? Another thing that kills me is that in the IAR definitions, many vectors are named in such a way that it's hard to find the interrupt you need. For ADC, for example. It's easier to declare it yourself, or just write a magic number.

DOOMSDAY09 February 2013, 14:44

Then "the first one up gets the slippers"

Wrong phrase.

and the I2C interrupt is completely ignored

Yeah, it's not in your table at all :)

At first it may seem that it is not very convenient to work with.

At first it seemed to me that she was very stoned. Then too, however.

00

https://we.easyelectronics.ru/STM8/8l-kurs-chast-3-preryvaniya-exti.html

10/11

PS: It should also be added that they should not try to repeat this on the STM8S series :)

**angel5a**

11 February 2013, 17:17

I fixed the table (there was a continuation on another page, I didn't grab it right away), wrote about STM8S, even removed the slippers :)

0

And the system looks strange, yeah. It looks like they took the basis from STM8S and tried to expand it, but they did it in a very bizarre way. Well, at least we have an interrupt from any leg

**dcoder**

11 February 2013, 17:35

Suddenly added a movie

0

**dcoder**

23 March 2013, 03:32

Only registered and authorized users can leave comments.