Using hardware and software to make new stuff

Search

type, hit enter

Categories

- 3D Printing (1)
- Affinity Photo (1)
- Aide-memoir (1)
- Analog (1)
- Ansible (3)
- Dates to Remember (3)
- Docker (1)
- Electronics (144)
- ESP8266 (12)
- FPGA (4)
- Garden (1)
- General (8)
- Home Built CPU (6)
- Informatica (1)
- Internet of Things (8)
- iOS (2)
- KiCad (4)
- MSP430 (2)
- Netduino (49)
- NuttX (9)
- Photography (3)
- Pico (10)
- Raspberry Pi (17)
- Silverlight (6)
- Software Development (88)
- STM32 (5)
- STM8 (54)
- Tips (5)

Archives

- July 2024
- June 2024
- May 2024
- April 2024
- March 2024
- February 2024
- January 2024
- December 2023
- November 2023
- October 2023

- February 2020
- January 2020
- July 2019
- February 2018
- July 2017
- June 2017
- April 2017
- March 2017
- February 2017
- October 2016
- September 2016
- July 2016
- June 2016
- May 2016
- April 2016
- March 2016
- January 2016
- December 2015
- November 2015
- October 2015
- August 2015
- June 2015
- May 2015
- April 2015
- March 2015
- February 2015
- January 2015
- December 2014
- October 2014
- September 2014
- August 2014
- July 2014
- June 2014
- May 2014
- March 2014
- February 2014
- January 2014
- December 2013
- November 2013
- October 2013
- September 2013
- August 2013
- July 2013
- June 2013
- May 2013
- April 2013
- March 2013
- January 2013
- November 2012
- October 2012
- September 2012
- August 2012
- July 2012
- June 2012
- May 2012

Interrupts on the STM8S

A while ago I wrote about using interrupts on the STM8S (see External Interrupts on the STM8S and hinted there that I would come back to the topic and here we are. In this article we will cover the following:

- List of interrupts available and the interrupt table
- Writing your own Interrupt Service Routine (ISR)
- Setting/Changing interrupt priorities

It is probably a good time to remind you that the STM8S Reference Manual (document number RM0016), available from ST's web site is handy, to keep available for reference. This post is really meant for the application developer who wants to know which interrupts are available and how to use them.

Interrupt Table

The STM8S holds a list of interrupt vectors in a table in memory. As a programmer you are able to add your own ISRs to your application. These are really just conventional methods with a slightly different entry and exit mechanism. You don't have to worry about how this mechanism works as you can make the compiler do all of the work for you. We shall see how later. The compiler is also be good enough to ensure that the table of interrupt vectors is also updated to point to your code as part of the application start up.

The following table lists the interrupts which are available on the STM8S103F3 microcontroller:

Vector Number	Abbreviation	Description
1 (0x01)	TRAP	TRAP
2 (0x02)	TLI	Top Level Interrupt
3 (0x03)	AWU	Auto Wake Up
4 (0x04)	CLK	Clock
5 (0x05)	EXTI_PORTA	External Interrupts for Port A (Pins 2 through 6 inclusive)
6 (0x06)	EXTI_PORTB	External Interrupts for Port B (All pins)
7 (0x07)	EXTI_PORTC	External Interrupts for Port C (All pins)
8 (0x08)	EXTI_PORTD	External Interrupts for Port D (Pins 0 through 6 inclusive)
9 (0x09)	N/A	Not used
10 (0x0a)	N/A	Not used
11 (0x0b)	N/A	Not used
12 (0x0c)	SPI	SPI
13 (0x0d)	TIM1_UPD_OVF_TRG_BRK	Timer 1 Update/Overflow/Trigger/Break
14 (0x0e)	TIM1_CAP_COM	Timer 1 Capture/Compare
15 (0x0f)	TIM2_UPD_OVF_BRK	Timer 2 Update/Overflow/Break

20 (0x14)	UART_TX	UART Tx
21 (0x15)	I2C	I^2C
22 (0x16)	UART2_RX	UART 2 Rx
23 (0x17)	UART2_TX	UART 2 Tx
24 (0x18)	ADC1	ADC1
25 (0x19)	TIM4_UPD_OVF	Timer 1 Update/Overflow
26 (0x1a)	EEPROM_EEC	EEPROM EEC

One thing to remember is that while the interrupt vector numbers remain unchanged the availability of the interrupt vectors will change depending upon the chip you are using. For instance, vector 10 is not used here but on the STM8S208 this is available to process one of the CAN interrupts. The simplest way to find out if an interrupt is available is to look at the header file for your chip. So let’s

So how do you know which interrupts are available for your microcontroller?

The first thing to note is that the vector numbers for interrupts 1-9 are usually the same as these features are available on most of the controllers. Note that whilst interrupt 9 is not available on my chip it is the EXTI_PORTE vector. I have not been able to locate any standard definitions (i.e. #define’s etc) for these interrupt vectors in any of the header files supplied with the compiler.

For the rest of the vectors we will have to start to look through the header files for the microcontroller. Opening up <iostm8s103f3.h> and going to the very end of the file we find a section with the comment *Interrupt vector numbers*. There should be one or more definitions for each of the features which allow the use of interrupts and which are available on the microcontroller.

One of the things to note about the list of available interrupts is that there are more than one *#define* statements for each feature. Consider the following extract:

```
/*-------------------------------------------------------------------------
 *      Interrupt vector numbers
 *-----------------------------------------------------------------------*/
#define SPI_TXE_vector                       0x0C
#define SPI_RXNE_vector                      0x0C
#define SPI_WKUP_vector                      0x0C
#define SPI_MODF_vector                      0x0C
#define SPI_CRCERR_vector                    0x0C
#define SPI_OVR_vector                       0x0C
#define TIM1_CAPCOM_TIF_vector               0x0D
#define TIM1_CAPCOM_BIF_vector               0x0D
#define TIM1_OVR_UIF_vector                  0x0D
```

If we look at the *SPI* definitions we can see that all of the vectors map to the same interrupt number. This is because there is only one ISR for this (SPI) feature. So the important point to take away is that your ISR must work out which condition it is working with.

Consider the example from the earlier post External Interrupts on the STM8S. This used the following ISR:

```
#pragma vector = 8
__interrupt void EXTI_PORTD_IRQHandler(void)
{
    PD_ODR_ODR3 = !PD_ODR_ODR3;     //  Toggle Port D, pin 3.
}
```

application should interrogate the status registers in order to work out why the ISR has been called.

Writing your own Interrupt Service Routine (ISR)

Writing your own ISR is really no different from writing any other method in C. There are a few extra rules which need to be followed but the majority of the techniques are the same. So let us return to the external interrupt code example:

```
#pragma vector = 8
__interrupt void EXTI_PORTD_IRQHandler(void)
{
    PD_ODR_ODR3 = !PD_ODR_ODR3;     //  Toggle Port D, pin 3.
}
```

The first thing you notice is the addition of the *#pragma vector = 8* statement. This tells the compiler which interrupt vector we are going to be writing. In this case it is vector 8 which is the EXTI_PORTD interrupt vector (see the table above). You can also use the values found in the header file for you microcontroller. So you could write the following:

```
#pragma vector = SPI_TXE_vector
```

instead of:

```
#pragma vector = 0x0c
```

If you are using the same method for multiple vectors then you can provide the list of vector numbers as a comma separated list thus:

```
#pragma vector = SPI_TXE_vector, TIM1_OVR_UIF_vector
```

The next thing to notice is the *__interrupt* decoration which has been applied to the method. This tells the compiler that the method is to be used as an ISR. Knowing this, the compiler will ensure that the preamble and exit from the method are set up correctly as these are different from those of any other method which you might call.

Something which is not obvious from the above code is the fact the an ISR cannot take any parameters nor can it return a value. Hence the method definition takes a void parameter list and returns void.

You should also note that it is possible to write an interrupt method (i.e. decorate a method with *__interrupt*) without providing a *#pragma vector* statement. In this case the compiler will not generate an entry in the interrupt vector table.

You should also consider how fast the ISR needs to be. In this case, a single button press, we do not need any real efficiency as the microcontroller is not really doing anything overly complex. This may not be the case in non-trivial applications. In which case you will need to analyse your system to determine how fast the ISR needs to be. You also need to take into account the fact that one ISR may be interrupted by a higher priority ISR.

Setting/Changing interrupt priorities

Interrupts can be assigned a software priority from 0 to 3. This allows the interrupts to be nested ensuring that the most important interrupts receive attention over less important interrupts.

1	01
2	00
3	11

Where the priority increases from the top of the table towards the bottom.

By default, all of the interrupts are assigned the same software priority, 3. This effectively means that the priority is disabled and all interrupts are treated as equal by the software. When the software priority is disabled or when two or more interrupts have the same priority then the interrupts are queued for processing

The software interrupt priority for the interrupt vectors (and hence your ISR) is stored in a group of registers. Each register (ITC_SPR1 through ITC_SPR8) holds the priority for four interrupt vectors. At reset all of these priorities are reset to 11 – software priority disabled. The register mapping is given by the following table:

Register	Bits 7:6	Bits 5:4	Bits 3:2	Bits 1:0
ITC_SPR1	VECT3SPR	VECT2SPR	VECT1SPR	VECT0SPR
ITC_SPR2	VECT7SPR	VECT6SPR	VECT5SPR	VECT4SPR
ITC_SPR3	VECT11SPR	VECT10SPR	VECT9SPR	VECT8SPR
ITC_SPR4	VECT15SPR	VECT14SPR	VECT13SPR	VECT12SPR
ITC_SPR5	VECT19SPR	VECT18SPR	VECT17SPR	VECT16SPR
ITC_SPR6	VECT23SPR	VECT22SPR	VECT21SPR	VECT20SPR
ITC_SPR7	VECT27SPR	VECT26SPR	VECT25SPR	VECT24SPR
ITC_SPR8	–	–	VECT29SPR	VECT28SPR

Changing the priority is a simple matter of identifying the vector and changing the appropriate register (ITC_SPR1_VECT1SPR for vector 1, ITC_SPR1_VECT2SPR for vector 2 etc.).

Conclusion

Much of the information we have covered here is adequate for the average user intending to develop a module for the Netduino GO!. For more advanced topics such as what happens in low power modes, how interrupts are handled, nested interrupt handling etc. then you should refer to Chapter 6 in RM0016.

Tags: Software Development, STM8

Sunday, September 2nd, 2012 at 7:48 pm • Software Development, STM8 • RSS 2.0 feed Both comments and pings are currently closed.

Comments are closed.

Pages

- About
- Making a Netduino GO! Module
- The Way of the Register
- Making an IR Remote Control
- Weather Station Project
- NuttX and Raspberry Pi PicoW

Support This Site

Currently Reading