INTERRUPTS

WHAT IS AN INTERRUPT?:

Programming is sequential by nature and that's all right when we deal with computer programming however, micro controllers interact with the physical world. The physical world is unpredictable, events happen at random intervals and sometimes they have to be dealt with right away to prevent hardware damage or data loss.

Say you have a program that bakes cookies and calculates the value of pie. So we write a program that calculates a digit, then checks to see if the cookies are done, and repeats over and over again. Because each digit takes longer and longer to calculate your cookies will probably come out burned.

The good news is that AVR's something called Interrupts, which basically allows you to interrupt the current running code in order to perform another action, and once done resume the function that it interrupted. So using our previous example, the "cookies are done" interrupt would trigger, dropping us out of the pie digit calculation function and provide us with melty, chewy cookie goodness.

I will cover the basics theory of interrupts in this section, the interrupts that deal with specific system will be covered under those systems (ie. timer interrupts will be covered under the timers section .. etc)

THEORY OF OPERATION:

If you look at the AVR data sheet you will notice a table to interrupt vectors. This is basically a priority list, the lowest number on the list will be executed first. As you can see the reset is the highest priority interrupt, and Store Program Memory Ready is the lowest priority. If the program calls for both at the same time, the reset is going to win.

Interrupts are off by default and must be enabled using the sei() function. They could also be stopped using the cli() function (CLear interrupts).

In order to use interrupts we use the sei() function, this Enables the Global Interrupt Enable (I-bit) in the Status Register (SREG). When an interrupt occurs, the Global Interrupt Enable (I-bit) in the is Status Register (SREG) is cleared in order to disable other interrupt calls. The user software can write logic 1 to the I-bit (I in SREG register) to enable nested interrupts(ie to allow other interrupts to interrupt the interrupt). The I-bit is automatically set when the program exits the interrupt routine. When the AVR exits from an interrupt, it will always return to the main program and execute one more instruction before any pending interrupt is served.

> Note: main program means main() or any function called by main(); so if main() calls a function deathray(), and an interrupt is called, the interrupt will be executed and the program will jump back to deathray(). If another interrupt is pending, only 1 instruction of deathray() has to be executed before that interrupt can fire.

SOFTWARE:

Lets just jump into it and explain later:

ATmega8 & ATmega168/328 Code:

```
#include <avr/io.h>
#include <avr/interrupt.h>


int main(void)
{
    sei();          //enable interrupts

    while (1)       // main loop
    {
        ;
    }
}
```

```
ISR (INT0_vect)    // INT0 interrupt function
{
    /* enter code to execute here */
}


ISR (BADISR_vect)  // special function, to execute if a bad interrupt is called
{
    /* enter code to execute here */
}
```

Lets examine the code, io.h is always included in order to setup your special registers. interrupt.h is included because it needed to run your interrupts stuff. sei() starts our interrupt (stands for SEt Interrupt). ISR (***_vect) is the function that is called when the *** vector is called. *** is basicly any name found on the table in the interrupt secion of the datasheet (substitute spaces with understores "_"). BADISR is not on the list, however, it has a neat function, ... in the event that your code calls a vector that does not exist it will execute this code.

Interrupt Disable:

As mentioned above interrupts can be enabled and disabled as needed. In some cases you might have a chunk of code which cannot be interrupted such as a time sensitive output signal to another device.

ATmega8 & ATmega168/328 Code:

```
#include <avr/io.h>

#include <avr/interrupt.h>


int main(void)
{
    sei();          //enable interrupts

    while (1)       // main loop
    {

        /* code here can be interrupted */

        cli();      // turn off interrupts

        /* all code executed here cannot be interrupted */

        sei();      // enable interrupt

        /* code here can be interrupted */

        ;
    }
}


ISR (INT0_vect)    // INT0 interrupt function
{
    /* enter code to execute here */
}


ISR (BADISR_vect)  // special function, to execute if a bad interrupt is called
{
    /* enter code to execute here */
}
```

Passing Variables Into ISP():

ATmega8 & ATmega168/328 Code:

```
#include <avr/io.h>

#include <avr/interrupt.h>


volatile uint8_t test;


int main(void)
{
```

```
    sei();          //enable interrupts

    while (1)       // main loop
    {

        /* code here can be interrupted */

        cli();      // turn off interrupts

        /* all code executed here cannot be interrupted */

        sei();      // enable interrupt

        /* code here can be interrupted */

        ;
    }
}


ISR (INT0_vect)     // INT0 interrupt function
{
    /* enter code to execute here */
}


ISR (BADISR_vect)  // special function, to execute if a bad interrupt is called
{
    /* enter code to execute here */
}
```

Because there is no way to pass data to ISR other then a global variable it is a great idea to make the variable "volatile". Basically, this will prevent the compiler from optimizing the code in such a way as to make the variable no update properly by the ISR. If you like to know more, please read **Do you volatile? should you?** by Dr. Kevin P. Dankwardt.

That's it for the general stuff.

Cheers
Q

Comments? Questions? Problems with content? q.qeewiki@gmail.com

Comments

You do not have permission to add comments.