# EXTERNAL INTERRUPTS ON THE ATmega168/328

**INTRODUCTION:**

In the previous section I talked about the basics of interrupts. In this section, we will talk about the first type of device interrupts called external interrupts. These interrupts are basically called on a given status change on the INTn pin. This is essentially an input interrupt and is great to use for applications when you might need to react quickly to an outside source, such as a bumper of your robot hitting the wall or to detect a blown fuse.

**HARDWARE:**

Hardware wise there is not difference between External Interrupts and Inputs so don't be afraid to reread the Digital Input Tutorial if you need a refresher.

If you look at the AVR pinout diagram you will see the INTx which are used for External Interrupts and PCINTx pins that are used for Pin Change Interrupts.

**EXTERNAL INTERRUPTS:**


*Figure 1: ATmega168/328 – External Interrupt Pins*

The ATmega8 and the ATmega88/168/328 are backwards compatible when it comes to the pinouts however, they are programmed slightly different and while external interrupts work the same way on both types of micro controllers they do require different code to run.

External interrupts are fairly powerful, they can be configured to trigger on one of 4 states. Low level will trigger whenever the pin senses a LOW (GND) signal. Any Logic Change trigger at the moment the pin changes from HIGH (Vcc) to LOW (GND) or from LOW (GND) to HIGH(Vcc). On Falling Edge will trigger at the moment the pin goes from HIGH (Vcc) to LOW (GND). On Rising Edge will trigger at the moment the pin goes from LOW (GND) to HIGH (Vcc). The best part is that you can configure each INTx independently.

External interrupts use the below 3 registers. Which you could find under the "External Interrupts section of the datasheet.

| | 7 bit | 6 bit | 5 bit | 4 bit | 3 bit | 2 bit | 1 bit | 0 bit |
|---|---|---|---|---|---|---|---|---|
| **EICRA** | - | - | - | - | ISC11 | ISC10 | ISC01 | ISC00 |

*External Interrupt Control Register A*

| ISCx1 | ISCx0 | DESCRIPTION |
|---|---|---|
| 0 | 0 | Low level of INTx generates an interrupt request |
| 0 | 1 | Any logic change on INTx generates an interrupt request |
| 1 | 0 | The falling edge of INTx generates an interrupt request |
| 1 | 1 | The rising edge of INTx generates an interrupt request |

*ISC Bit Settings*

| | 7 bit | 6 bit | 5 bit | 4 bit | 3 bit | 2 bit | 1 bit | 0 bit |
|---|---|---|---|---|---|---|---|---|
| **EIMSK** | - | - | - | - | - | - | INT1 | INT0 |

*External Interrupt Mask Register*

| | 7 bit | 6 bit | 5 bit | 4 bit | 3 bit | 2 bit | 1 bit | 0 bit |
|---|---|---|---|---|---|---|---|---|
| **EIFR** | - | - | - | - | - | - | INTF1 | INTF0 |

*External Interrupt Flag Register*

*ATmega168/328 Code:*

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


int main(void)
{
    DDRD &= ~(1 << DDD2);     // Clear the PD2 pin
    // PD2 (PCINT0 pin) is now an input

    PORTD |= (1 << PORTD2);    // turn On the Pull-up
    // PD2 is now an input with pull-up enabled


    EICRA |= (1 << ISC00);    // set INT0 to trigger on ANY logic change
    EIMSK |= (1 << INT0);     // Turns on INT0

    sei();                    // turn on interrupts

    while(1)
    {
        /*main program loop here */
    }
}


ISR (INT0_vect)
{
    /* interrupt code here */
}
```

**PIN CHANGE INTERRUPTS:**

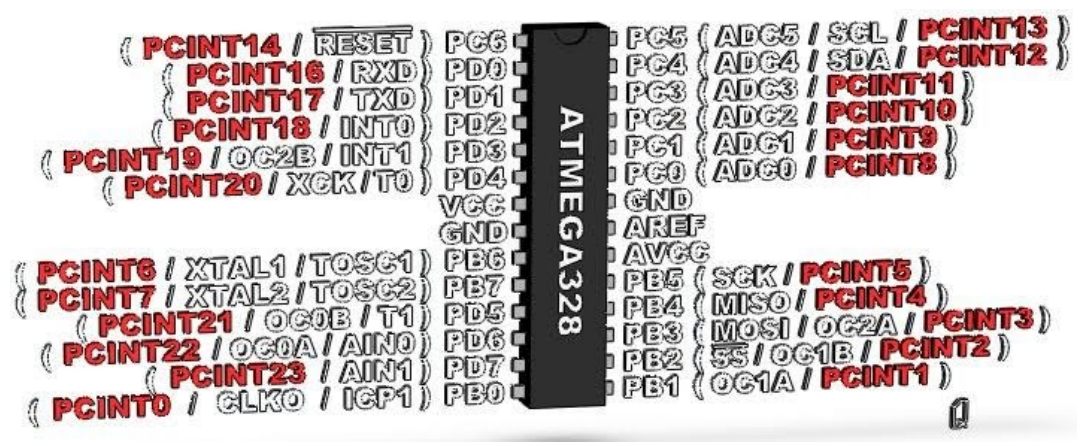


*Figure 2: ATmega168/328 - Pin Change Interrupt Pins*

One important thing to note, on the older ATmega8 does not have any PCINT pints, therefore, this section of the tutorial only applies to ATmega88 through ATmega328.

| | 7 bit | 6 bit | 5 bit | 4 bit | 3 bit | 2 bit | 1 bit | 0 bit |
|---|---|---|---|---|---|---|---|---|
| **PCICR** | - | - | - | - | - | PCIE2 | PCIE1 | PCIE0 |

*Pin Change Interrupt Control Register*

| | 7 bit | 6 bit | 5 bit | 4 bit | 3 bit | 2 bit | 1 bit | 0 bit |
|---|---|---|---|---|---|---|---|---|
| **PCIFR** | - | - | - | - | - | PCIF2 | PCIF1 | PCIF0 |

*Pin Change Interrupt Flag Register*

| | 7 bit | 6 bit | 5 bit | 4 bit | 3 bit | 2 bit | 1 bit | 0 bit |
|---|---|---|---|---|---|---|---|---|
| **PCMSK0** | PCINT7 | PCINT6 | PCINT5 | PCINT4 | PCINT3 | PCINT2 | PCINT1 | PCINT0 |

*Pin Change Mask Register 0*

| | 7 bit | 6 bit | 5 bit | 4 bit | 3 bit | 2 bit | 1 bit | 0 bit |
|---|---|---|---|---|---|---|---|---|
| **PCMSK2** | PCINT23 | PCINT22 | PCINT21 | PCINT20 | PCINT19 | PCINT18 | PCINT17 | PCINT16 |

*Pin Change Mask Register 2*

| | 7 bit | 6 bit | 5 bit | 4 bit | 3 bit | 2 bit | 1 bit | 0 bit |
|---|---|---|---|---|---|---|---|---|
| **PCMSK1** | - | PCINT14 | PCINT13 | PCINT12 | PCINT11 | PCINT10 | PCINT9 | PCINT8 |

*Pin Change Mask Register 1*

| | 7 bit | 6 bit | 5 bit | 4 bit | 3 bit | 2 bit | 1 bit | 0 bit |
|---|---|---|---|---|---|---|---|---|
| **PCMSK0** | PCINT7 | PCINT6 | PCINT5 | PCINT4 | PCINT3 | PCINT2 | PCINT1 | PCINT0 |

*Pin Change Mask Register 0*

The PCIEx bits in the PCICR registers enable External Interrupts and tells the MCU to check PCMSKx on a pin change state. When a pin changes states (HIGH to LOW, or LOW to HIGH) and the corresponding PCINTx bit in the PCMSKx register is HIGH the corresponding PCIFx bit in the PCIFR register is set to HIGH and the MCU jumps to the corresponding Interrupt vector.

*ATmega168/328 Code:*

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


int main(void)
{
    DDRB &= ~(1 << DDB0);     // Clear the PB0 pin
    // PB0 (PCINT0 pin) is now an input

    PORTB |= (1 << PORTB0);    // turn On the Pull-up
    // PB0 is now an input with pull-up enabled


    PCICR |= (1 << PCIE0);    // set PCIE0 to enable PCMSK0 scan
    PCMSK0 |= (1 << PCINT0);  // set PCINT0 to trigger an interrupt on state change

    sei();                    // turn on interrupts

    while(1)
    {
        /*main program loop here */
    }
}



ISR (PCINT0_vect)
{
    /* interrupt code here */
}
```

Now there are 2 limits to the PCINTx interrupt. The first is that ANY change to the pin state will trigger in interrupt, if you remember you can control what state change triggers an INTx interrupt (rising pulse, falling pulse, Low level or any level change). Since a pin has to change states in order to trigger the interrupt we can read the state of the pin and if it's currently HIGH(Vcc) we know that a rising edge event triggered the interrupt, like wise if the pin state is currently LOW(GND) we know that a falling edge interrupt triggered the interrupt.

*ATmega168/328 Code:*

```
#include <avr/io.h>
#include <avr/interrupt.h>    // Needed to use interrupts


int main(void)
{
    DDRB &= ~(1 << DDB0);          // Clear the PB0 pin
    // PB0 (PCINT0 pin) is now an input

    PORTB |= (1 << PORTB0);         // turn On the Pull-up
    // PB0 is now an input with pull-up enabled

    PCICR |= (1 << PCIE0);      // set PCIE0 to enable PCMSK0 scan
    PCMSK0 |= (1 << PCINT0);    // set PCINT0 to trigger an interrupt on state change

    sei();                      // turn on interrupts

    while(1)
    {
        /*main program loop here */
    }
}



ISR (PCINT0_vect)
{
    if( (PINB & (1 << PINB0)) == 1 )
    {
        /* LOW to HIGH pin change */
    }
    else
    {
        /* HIGH to LOW pin change */
    }
}
```

Now for the 2nd problem is that up to 8 pins share the same PCINTx vector. So when the interrupt fires you will have detect what event triggered the change. So what we might need to do is take a snapshot of the Input states and compare them to the state from the previous time the interrupt triggered. If you take a look at the datasheet pinouts PCMSK0 matches up with PORTB, PCMSK1 with PORTC and PCMSK2 with PORTD.

*ATmega168/328 Code:*

```
#include <avr/io.h>
#include <stdint.h>            // has to be added to use uint8_t

#include <avr/interrupt.h>    // Needed to use interrupts


volatile uint8_t portbhistory = 0xFF;     // default is high because the pull-up


int main(void)
{
    DDRB &= ~((1 << DDB0) | (1 << DDB1) | (1 << DDB2)); // Clear the PB0, PB1, PB2 pin
    // PB0,PB1,PB2 (PCINT0, PCINT1, PCINT2 pin) are now inputs

    PORTB |= ((1 << PORTB0) | (1 << PORTB1) | (1 << PORTB2)); // turn On the Pull-up
    // PB0, PB1 and PB2 are now inputs with pull-up enabled

    PCICR |= (1 << PCIE0);      // set PCIE0 to enable PCMSK0 scan
    PCMSK0 |= (1 << PCINT0);    // set PCINT0 to trigger an interrupt on state change

    sei();                      // turn on interrupts

    while(1)
    {
        /*main program loop here */
    }
}



ISR (PCINT0_vect)
{
    uint8_t changedbits;


    changedbits = PINB ^ portbhistory;
    portbhistory = PINB;


    if(changedbits & (1 << PINB0))
    {
        /* PCINT0 changed */
    }

    if(changedbits & (1 << PINB1))
    {
        /* PCINT1 changed */
```

```
    }

    if(changedbits & (1 << PINB2))
    {
        /* PCINT2 changed */
    }

}
```

Ok ok, I know, the bit math is a bit funny. So I'll be nice and explain it. When we XOR ( ^ ) the PORTB register with the portbhistory register we will get a 0's on the bits that are the same, and 1's in the bits that are different (yes a practical use for the XOR operation). In the IF statements we AND ( & ) the operation with a bitmask that we created using the bit shift register (1 << PBx) in order to isolate specific bit. Lastly, notice how I defined porthistory as volatile? Like I said before, if you want to pass a global variable to an Interrupt make it volatile so that it doesn't cause obsolete data due to compiler optimization.

Now finally, what if you wanted to put both together? Well I got to leave a bit of fun for you guys.

Cheers
Q

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

## Comentarios

No tienes permiso para añadir comentarios.