A Short Hardware Interrupt Tutorial

An hardware interrupt is a signal that stops the current program forcing it to execute another program immediately. The interrupt does this without waiting for the current program to finish. It is unconditional and immediate which is why it is called an interrupt - it interrupts the current action of the processor.

A software interrupt, on the other hand is as its name suggests nothing to do with hardware causing an interrupt it has everything to do with making software easier to write.

You will often find software interrupts used in x86 BIOS routines and they make it easier to update the software since the interrupt routine will always be in the located in the same place e.g. for plotting a pixel a software interrupt is executed and the actual code that plots the pixel is defined by the manufacturer.

Hardware Interrupt Example:

Reading a keypad is made far easier using interrupts (especially on PIC devices) as PORTB has an **interrupt-on-change** feature for PortB pins. Attaching each pin to a push-to-make buton and enabling the internal pullups on these pins gives you an easy way to read button presses.

You can arrange that whenever the push button is pressed an interrupt is generated that forces the interrupt routine to go and read (and store) the state of that input.

An interrupt occurs independently of main code operation. So the processor can be doing other things before and after the interrupt. During the interrupt, main code operation is suspended and all registers are saved so that after the interrupt main code starts up exactly where it was.

It is as if the main code operates as usual but the interrupt operates without the main code noticing!

Polling

When you first start creating a program it is easy to read an input directly from an input pin using a simple read command. As you develop the program you will want to know when the input has changed and to do that you will put the read command into a loop and wait until the input pin has changed state.

The following pseudocode illustrates the idea:

```
int pin = 0;

set_input(3,PORTB);

pin = read(3,PORTB)

for(;;) {
   if ( read(3,PORTB)!=pin ) {
      pin = read(3,PORTB);
      break;
   }
}

// pin has changed so take action
```

This is an example of polling and you can see that the problem with polling is that the processor can do nothing else while it is waiting for the input pin to change state.

That is the advantage of interrupts - a program can be doing something completely different and does not waste processing power in a useless loop.

Another benefit of using interrupts is that in some processors you can use a **wake-from-sleep interrupt**. This lets the processor go into a low power mode, where only the interrupt hardware is active, which is useful if the system is running on batteries.

Hardware interrupt Common terms

Terms you might hear associated with hardware interrupts are ISR, interrupt mask, non maskable interrupt, an asynchronous event, interrupt vector and context switching.

ISR	`Interrupt Service Routine.`
Interrupt vector	`The address that holds the location of the ISR.`
Interrupt mask	`Controls which interrupts are active.`
NMI	`Non Maskable Interrupt - an interrupt that is always active.`
Asynchronous event	`An event that could happen at any time.`
Context switching	`Saving/restoring data before & after the ISR.`

ISR

An ISR is simply another program function. It is no different to any other function except that it does context switching (saving/restoring processor registers) and at the end of the function it re-enables interrupts.

After the ISR finishes program execution returns to the original program and it continues exactly from where it was interrupted. The original program will have no idea that this has happened.

Hardware Interrupt vector

This is a fixed address that contains the location of your ISR – for a PIC micro it is usually address 4. In other micros there may be an interrupt vector for each vector – you have to make an interrupt routine for each one that you want to use. For PIC micros you have just one interrupt and you have to detect which interrupt triggered by examining the interrupt flag register(s).

You program the interrupt address with the address of your interrupt routine. Whenever the interrupt is triggered (and if the interrupt is unmasked) program operation jumps to the location of your interrupt routine.

*Note: high level language compilers take care of all of this for you - in 'C' you just declare the function using the keyword **interrupt** (as the type returned from the function). It then puts the address of this routine in the interrupt vector.*

NMI

The NMI is exactly the same as a normal interrupt except that you can not control whether or not it is active - it's always active. It is more commonly found on older/larger processors as a dedicated input pin. In this case it is more than likely fed with a brown-out-detector (BOD) circuit i.e a PSU voltage dip detector circuit - for detecting a condition that should never be ignored!

You don't need it in a microcontroller as you can achieve exactly the same functionality using a programmable interrupt and many microcontrollers have built-in BODs.

Asynchronous event

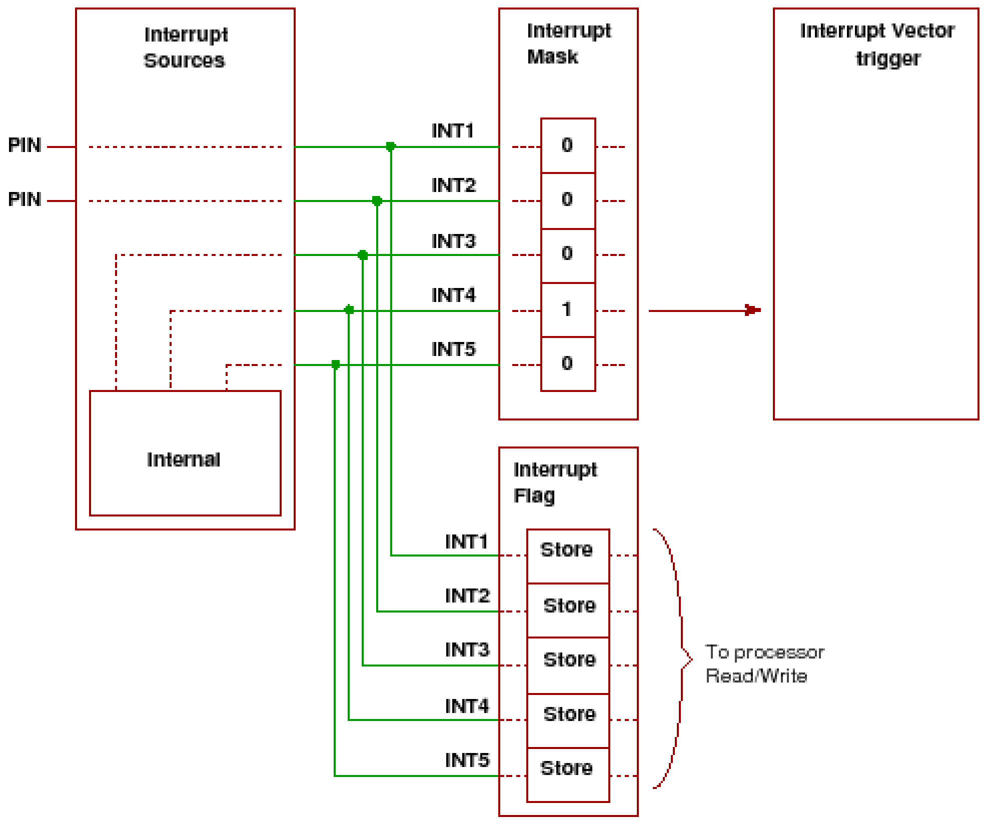
Context switching

This means that the register state is preserved at the start of an ISR and restored at the end of an ISR. It ensures that the function that was interrupted is not affected by the ISR.

A Basic Interrupt System

This is a general description of an interrupt system (biased slightly to PIC micros) which is true for any interrupt module and it is useful for understanding how to control and use interrupts.

Hardware Interrupt Block Diagram



Input sources

These are signals that start an interrupt. They can be external pins (where a rising or falling edge of an input triggers the interrupt) or internal peripheral interrupts e.g. ADC completed, Serial Rx data received, timer overflow etc. (The 16F877 has 15 interrupt sources).

Note: you can usually control which edge (rising or falling) is used by setting bits in another control register.

Interrupt flags.

Each hardware interrupt source has an associated interrupt flag - whenever the interrupt is triggered the corresponding interrupt flag is set. These flags are usually stored as bits within an interrupt register.

The processor can read from and write to the interrupt register, reading from it to find out which interrupts occurred and writing to it to clear the interrupt flags.

Interrupt mask

The interrupt mask has a set of bits identical to those in the interrupt register. Setting any bit (or unmasking) lets the corresponding signal source generate an interrupt - causing the processor to execute the ISR.

Note: When a bit in the mask is clear (masked) the ISR is not activated for that signal source but the interrupt register is still set by the signal source. So you can still detect what the hardware is doing by polling the interrupt register.

Hardware interrupt vector

The interrupt vector is a location in memory that you program with the address of your interrupt service routine (ISR). Whenever an unmasked interrupt occurs program execution starts from the address contained in the interrupt vector.

For PIC micros the hardware interrupt vector address is usually 0004.

This is simply an event that is not synchronised to the processor clock. It is an event that the processor can not predict e.g. A button press.