INTERRUPTS in microprocessor systems

Microcontroller

Microcontroller – all components of computer system in one an integrated circuit (chip)
What are Interrupts?

- One of the most fundamental and useful principles of modern embedded processors are interrupts.
- There seems to be a lot of confusion about interrupts; what are they, and how do we use them?
- In short, an interrupt is a way for an external (or, sometimes, internal) event to pause the current processor’s activity, so that it can complete a brief task before resuming execution where it left off.

What can cause interrupts?

- In systems programming, an interrupt is a signal to the processor or an instruction in software usually indicating an event that needs immediate attention.
- An interrupt signals the processor of a high-priority condition requiring the interruption of the current code the processor is executing, the current thread.
- The processor responds by suspending its current activities, saving its state, and executing a small program called an interrupt handler (interrupt service routine, ISR) to deal with the event.
- This interruption is temporary, and after the interrupt handler finishes, the processor resumes execution of the previous thread.
PC computer - what are Interrupts?

- A signal informing a program that an event has occurred.
- When a program receives an interrupt signal, it takes a specified action (which can be to ignore the signal).
- Interrupt signals can cause a program to suspend itself temporarily to service the interrupt.
- Interrupt signals can come from a variety of sources. For example, every keystroke generates an interrupt signal. Intermittents can also be generated by other devices, such as a printer, to indicate that some event has occurred. These are called hardware interrupts.
- Interrupt signals initiated by programs are called software interrupts. A software interrupt is also called a trap or an exception.
- PCs support 256 types of software interrupts and 15 hardware interrupts. Each type of software interrupt is associated with an interrupt handler -- a routine that takes control when the interrupt occurs.
- For example, when you press a key on your keyboard, this triggers a specific interrupt handler. The complete list of interrupts and associated interrupt handlers is stored in a table called the interrupt vector table, which resides in the first 1 K of addressable memory.

Types of interrupts - hardware interrupt

There are two main types of interrupt:

- A hardware interrupt is an electronic alerting signal to the processor from an external device, either a part of the computer itself such as a disk controller or an external peripheral.
- For example, pressing a key on the keyboard or moving the mouse triggers hardware interrupts that cause the processor to read the keystroke or mouse position.
- Unlike the software type, hardware interrupts are asynchronous and can occur in the middle of instruction execution, requiring additional care in programming.
- The act of initiating a hardware interrupt is referred to as an interrupt request (IRQ).
Types of interrupts - software interrupt

- A **software interrupt** is caused either by an exceptional condition in the processor itself, or a special instruction in the instruction set which causes an interrupt when it is executed.
- The former is often called a trap or exception and is used for errors or events occurring during program execution that are exceptional enough that they cannot be handled within the program itself.
- For example, if the processor's arithmetic logic unit is commanded to divide a number by zero, this impossible demand will cause a *divide-by-zero exception*, perhaps causing the computer to abandon the calculation or display an error message.
- Software interrupt instructions function similarly to subroutine calls and are used for a variety of purposes, such as to request services from low level system software such as device drivers.
- For example, computers often use software interrupt instructions to communicate with the disk controller to request data be read or written to the disk.

What can cause interrupts?

- Hardware interrupts were introduced as a **way to avoid wasting the processor's valuable time in polling loops**, waiting for external events.
- They may be implemented in hardware as a distinct system with control lines, or they may be integrated into the memory subsystem.
- If implemented in hardware, an interrupt controller circuit such as the IBM PC's Programmable Interrupt Controller (PIC) may be connected between the interrupting device and the processor's interrupt pin to multiplex several sources of interrupt onto the one or two CPU lines typically available.
- If implemented as part of the memory controller, interrupts are mapped into the system's memory address space.
**Types of interrupts**

Interrupts can be categorized into these different types:

- **Maskable interrupt** (IRQ) is a hardware interrupt that may be ignored by setting a bit in an interrupt mask register's (IMR) bit-mask.

- **Non-maskable interrupt** (NMI) is a hardware interrupt that lacks an associated bit-mask, so that it can never be ignored. NMIs are used for the highest priority tasks such as timers, especially watchdog timers.

- **Inter-processor interrupt** (IPI) is a special case of interrupt that is generated by one processor to interrupt another processor in a multiprocessor system.

- **Software interrupt** is an interrupt generated within a processor by executing an instruction. Software interrupts are often used to implement system calls because they implement a subroutine call with a CPU ring level change.

- **Spurious interrupt** is a hardware interrupt that is unwanted. They are typically generated by system conditions such as electrical interference on an interrupt line or through incorrectly designed hardware.

**Maskable interrupt**

- Processors typically have an internal *interrupt mask* which allows software to ignore all external hardware interrupts while it is set.

- This mask may offer faster access than accessing an interrupt mask register (IMR) in a PIC, or disabling interrupts in the device itself.

- In some cases, such as the x86 architecture, disabling and enabling interrupts on the processor itself act as a memory barrier, however it may actually be slower.
**Interrupt priority level**

- The **interrupt priority level (IPL)** is a part of the current system interrupt state, which indicates the interrupt requests that will currently be accepted.
- The IPL may be indicated in hardware by the registers in a Programmable Interrupt Controller, or in software by a bitmask or integer value.

**Interrupt priority**

- Each hardware interrupt has:
  - an interrupt level (source),
  - trigger,
  - interrupt priority.

- **Interrupt Level**
  The interrupt level defines the source of the interrupt and is often referred to as the *interrupt source*.

- **Interrupt Trigger**
  There are two types of trigger mechanisms, level-triggered interrupts and edge-triggered interrupts.

- **Interrupt Priorities**
  The interrupt priority defines which of a set of pending interrupts is serviced first.
INTERRUPTS in microcontroller

- Probably you won’t be able to find a microcontroller without interrupt capability. These are essential attributes of any modern microcontroller or processor.
- Interrupt capability may seem confusing and tricky at first glance, but during the time you will find out that normal MCU operation is impossible without interrupts.
- Microcontrollers usually have multiple interrupt sources available. For instance Atmega32 has 21 of them. So program flow can be interrupted by various events like external pin go low, timer overflow or ADC conversion complete.
- If set properly these events will take over program flow once occur and give back resources once they’ve been serviced. As we mentioned microcontroller can have multiple interrupt sources.
- AVR is equipped only with hardware interrupts. Other microcontrollers may have software interrupts as well. Hardware interrupts mean that they can only be generated by hardware events like pin change or other.
- If you look in to AVR datasheet you will always find a table of interrupt vectors like this:

Erasmus 2015/2016, WIEK- PK
This table is programmed into microcontroller memory along with your program as it is a reference table containing address of interrupt service routines – a special functions that will be called once interrupt will occur.

As you can see in table there is a column “Program Address” these are locations in flash where a interrupt service routine is located in program memory.

For instance at the very beginning of flash (0x0000) there is a reset address stored. This means when MCU is powered program will look at this location and jumps to place where your main() program starts.

When programming in C – interrupt table is created automatically during compilation, while ASM programmers have to set this table by themselves.

Main sources of interrupts

- The 8-bit AVRs lack software interrupts, which are usually used for special operating system tasks like switching between user and kernel space, or for handling exceptions.
- Because of this, we’ll only be looking at hardware interrupts.
- Each AVR model contains a different set of interrupt sources.
- We can find out which interrupts our chosen model has by looking in the “Interrupts” chapter of the AVR’s datasheet.
- This contains a complete list of all the possible interrupts in our AVR, giving each interrupt’s Vector table address, Source Interrupt Name, and Source Interrupt Description.
- Take a look at your AVR’s list - see the variety of interrupt sources, from all sorts of on-chip peripherals of the AVR.
- Hardware interrupts can be useful to process sparingly occurring events, such as buttons presses or alarm inputs.
- They can also be useful for situations where a low, fixed latency is required, for example when processing inputs from a rotary encoder.
Simple interrupt handling sequence

- In the image you can see simple interrupt handling sequence.
- Let’s say our program is doing some tasks in Main program flow.
- When interrupt signal occurs at any given moment of program flow it stops at current location, remembers next operation address and then loads program counter with ISR address (1) stored in ISR vector table.
- From this moment interrupt handling function is performed (2). Once it’s complete program counter is loaded with next interrupted program operation (3).
- So main program actually doesn’t know that interrupt has occurred and continues normally. This is very simplified illustration of interrupt as we didn’t mention preserving of register values during interrupt and so on.

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.

Interrupt priority
- If only one interrupt request is issued, the pending interrupt will be handled
- If several interrupts occur at the same time, selecting the highest prioritized interrupt is done in hardware.
Interrupts in AVR

• By default, on the AVR devices, interrupts are themselves not interruptable. When an interrupt fires, the AVR CPU will automatically disable the Global Interrupt Enable bit, to prevent the ISR from being itself interrupted.
• This is to prevent stack overflows from too many interrupts occurring at once and to prevent the ISRs from running too long, as most uses of interrupts are to have minimal latency when processing an event.
• It’s perfectly possible to set the Global Interrupt Enable bit again as part of the ISR so that nested interrupts can occur, but this is highly not recommended as it is dangerous.

So how to we define an ISR?

For an ISR to be called, we need three conditions to be true:
• Firstly, the AVR’s global Interrupts Enable bit (I) must be set in the MCU control register SREG. This allows the AVR’s core to process interrupts via ISRs when set, and prevents them from running when cleared. It defaults to being cleared on power up, so we need to set it.
• Secondly, the individual interrupt source’s enable bit must be set. Each interrupt source has a separate interrupt enable bit in the related peripheral’s control registers, which turns on the ISR for that interrupt. This must also be set, so that when the interrupt event occurs the processor runs the associated ISR.
• Thirdly, The condition for the interrupt must be met - for example, for the USART Receive Complete (USART RX) interrupt, a character must have been received.
• When all three conditions are met, the AVR will fire our ISR each time the interrupt event occurs. Again, the method used to define an ISR differs between languages and compilers.
Simple interrupt handling sequence

Reset and Interrupt Handling

- The Atmel AVR provides several different interrupt sources. These interrupts and the separate reset vector each have a separate program vector in the program memory space.
- All interrupts are assigned individual enable bits which must be written logic one together with the Global Interrupt Enable bit in the Status Register in order to enable the interrupt.
- Depending on the Program Counter value, interrupts may be automatically disabled when Boot Lock bits BLB02 or BLB12 are programmed. This feature improves software security.
Simple interrupt handling sequence

- Without going into details it is worth mentioning that interrupt vectors can be stored in boot section depending on fuse settings – so bootloader could use benefits of interrupts. But leave this for some time later.
Interrupts in AVR – C language

- The Avr C-library provides default interrupt routines, which get used unless overridden.
- If you wish to provide an interrupt routine for a particular signal you must, in addition to any required AVR setup, create the function using one of the SIGNAL() or INTERRUPT() macros, along with the appropriate signal names.

There are a number of preset signal names, such as:
- ISR_ADC (ADC conversion done)
- ISR_EEPROM_READY
- ISR_INTERRUPT0..7 (external interrupts 0 to 7)
- ISR_OVERFLOW0..3 (timer/counter overflow)
- ISR_UART0_DATA, SIG_UART0_RECV,
- ISR_UART0_TRANS (UART empty/receive/transmit interrupts)

To create an interrupt routine, select a macro-signal combination and:

```c
#include <avr/signal.h>
ISR(SIG_INTERRUPT0)
{
    /* Your interrupt handler routine */
}
```

Example

```c
#define F_CPU 16000000ul  // definition of fxtal for CPU in Hertz, ul - unsigned long
#include <avr/io.h>        // Standard AVR header
#include <avr/interrupt.h> // is a header file with interrupt service functions

int main(void)
{
    DDRA = 0b10000000;     // Configure PORTA.7 as output
    DDRD = 0b00100000;     // Configure PORTD.5 as output
    TIMSK=(1<<TOIE0) | (1<<TOIE1); // enable timer overflow interrupt for both Timer0 and Timer1
    TCNT0=0x00;            // set Timer0 counter initial value to 0
    TCNT1=0x0000;          // set Timer1 counter initial value to 0
    TCCR0 = (1<<CS02) | (1<<CS00);  // Normal Mode, start 8-bit Timer0 with 1024 prescaler
    sei();                 // global enable interrupts

    while(1) // main loop
    {
        PORTC ^= _BV(6);   // Toggle bit PORTC.6 only
    }
}
```

```c
ISR(TIMER1_OVF_vect){  // Timer1 overflow
    PORTD ^= _BV(5);    // Toggle bit PORTD.5 only
}
ISR(TIMER0_OVF_vect){  // Timer0 overflow
    PORTA ^= _BV(7);    // Toggle bit PORTA.7 only
}
```

Erasmus 2015/2016, WIEK- PK