Interrupts.

Introduction

Interrupts are a relatively advanced topic in microprocessor programming. They can be very useful in control applications particularly when the microprocessor must perform two tasks apparently at the same time, or when critical timing of program execution is required. Ordinary subroutines are called using the CALL (assembler/machine code) instruction. Interrupts are very much like hardware generated CALL's.

When an interrupt occurs (and certain other conditions are met), the microprocessor saves the current state of its program counter and status register on the stack and jumps to a section of the program which is just like a subroutine, written for the purposes of servicing (responding to) the interrupt.

When this interrupt service routine is complete it executes a Return From Interrupt (RETI) instruction which restores the saved microprocessor registers to their original state and the interrupted program carries on from where it left off.

Interrupts are typically generated by a transition on a particular pin on the microprocessor. For example a pin might go from logic 0 to logic 1 – a so called “positive edge” or from a logic 1 to a logic zero – a negative edge. The transition (not the level) is what causes the interrupt. Some interrupt sources can also be level triggered e.g. keeping a pin at logic zero causes repeated interrupts to take place. Interrupts can also be caused by circuits within the microprocessor itself.

The MSP430 recognises interrupts from many sources and of different priorities. Details are quite model specific however, MSP430G2231 (Launchpad chip) supports the interrupts shown in Table 1 (taken from TI datasheet).

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-Up External Reset Watchdog Timer+ Flash key violation PC out-of-range(1)	PORIFG RSTIFG WDTIFG KEYV(2)	Reset	0FFFEh	31, highest
NMI Oscillator fault Flash memory access violation	NMIIFG OFIFG ACCVIFG(2) (3)	(non)-maskable (non)-maskable (non)-maskable	0FFFCh	30
			0FFFAh	29
			0FFF8h	28
			0FFF6h	27
Watchdog Timer+	WDTIFG	maskable	0FFF4h	26
Timer_A2	TACCR0 CCIFG(4)	maskable	0FFF2h	25
Timer_A2	TACCR1 CCIFG, TAIFG(2)(4)	maskable	0FFF0h	24
			0FFEEh	23
			0FFECh	22
ADC10(5)	ADC10IFG(4)(5)	maskable	0FFEAh	21
USI	USIIFG, USISTTIFG(2)(4)	maskable	0FFE8h	20
I/O Port P2 (two flags)	P2IFG.6 to P2IFG.7(2)(4)	maskable	0FFE6h	19
I/O Port P1 (eight flags)	P1IFG.0 to P1IFG.7(2)(4)	maskable	0FFE4h	18
			0FFE2h	17
			0FFE0h	16
See (6)			0FFDEh to 0FFC0h	15 to 0, lowest

(1) A reset is generated if the CPU tries to fetch instructions from within the module register memory address range (0h to 01FFh) or from within unused address ranges.
(2) Multiple source flags
(3) (non)-maskable: the individual interrupt-enable bit can disable an interrupt event, but the general interrupt enable cannot.
(4) Interrupt flags are located in the module.
(5) MSP430G2x31 only
(6) The interrupt vectors at addresses 0FFDEh to 0FFC0h are not used in this device and can be used for regular program code if necessary.

Table 1. Interrupt sources for the MSP430G2231 (Launchpad chip)

When do you use interrupts?

Consider the following scenario: A microprocessor is to be used to maintain a user interface while at the same time checking to see if a user presses a particular button. This could be achieved by inserting instructions to read the status of some input port register throughout the controlling program. This is not a great solution however as it requires the programmer to be very careful how they write their code – if the user interface enters a time consuming function, a button press might occur when the processor was not paying attention to the input port. Interrupts provide a more elegant way of achieving the same effect. Using interrupts, the programmer writes code to handle the user interface without worrying about the button press event. Hardware permitting, the transition of a port bit can be used to generate an interrupt. The processor will briefly stop what it is doing, note the button press and then return to the user interface. Computers do this sort of "multitasking" all the time. Another useful feature of interrupts, particularly with the MSP430 is that they can be used to save power (great for battery applications). The processor spends most of its time in a low power state, only waking upon receipt of an interrupt when some event takes place.

e.g. The MSP430G2231 consumes around 300μA when active. This would drain a 1000mAh phone battery in about 3000 hours (did I mention it was a low power chip?!)

It gets better, if the chip manages to spend most of its time in low power mode it will drain about

1µA. Theoretically, this would allow the device to run for 114 years! (The battery would decay long before that).

Interrupt handling

Figure 1 below shows how an interrupt might be handled in a simplifed, fictitious processor. The processor contains a communications device (e.g. ethernet network interface) and a digital input port. Both the communications device and input port are capable of generating interrupts. If a communications event takes place (e.g. data received), the communications interface generates an interrupt request (IRQ) in an effort to capture the CPU core's attention. The programmer may or may not allow this interupt to pass by setting or clearing the Interrupt Enable (IE) bit in the communications interface. Even if this interrupt is allowed to pass, it still must be pass into the CPU core if it is to be acted upon. The programmer may or may not allow this by setting or clearing the Global Interrupt Enable (GIE) bit (sometimes, the CPU needs to do tasks without being disturbed. Clearing the GIE bit prevents any devices from interrupting the CPU).

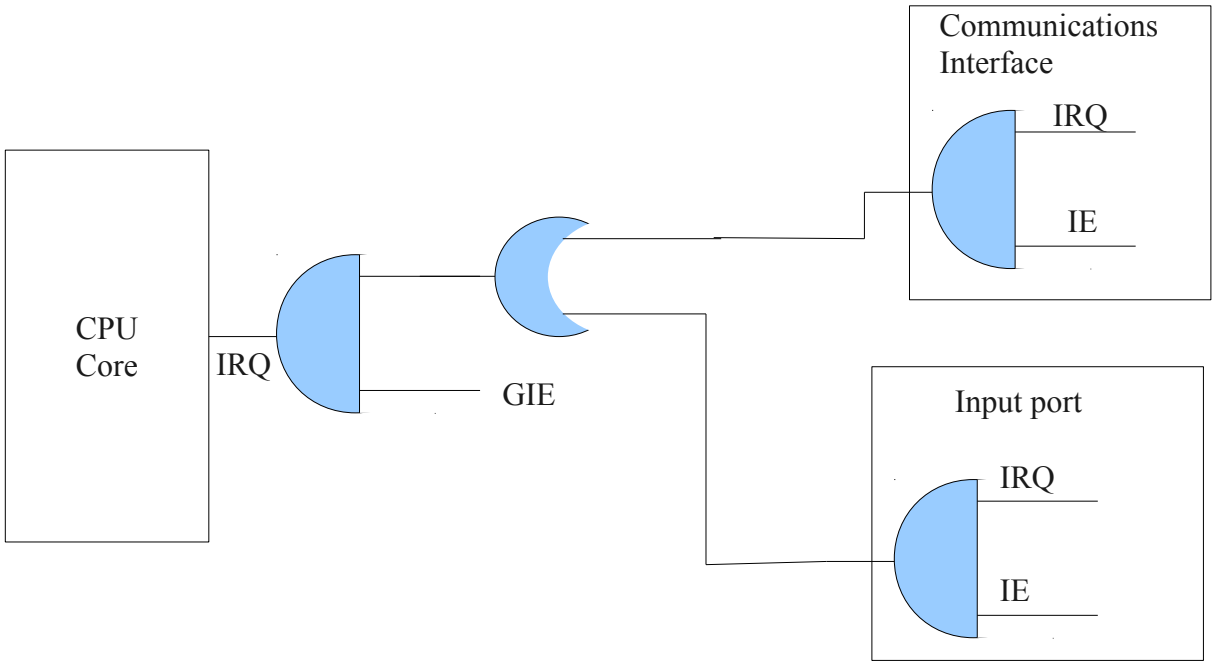

Figure 1. Simplified interrupt handling

Assuming the IE bits and GIE bits are set, interrupt requests (IRQ's) may arrive at the CPU. What then? As you can see in Figure 1, there is just one IRQ line into the CPU core. How does it know which device is interrupting it and hence how does it know how to react? Additional interrupt handling hardware (not shown in Figure 1) does the following when an interrupt happens:

1) IRQ line is driven low (“asserted”)
2) CPU saves some/all of its registers on the stack
3) Interrupt hardware identifies the interrupt source and depending upon the source forces the

CPU to begin executing code from a specific memory location – this code is the "interrupt service routine" (ISR)

4) Once the ISR is complete, the CPU restores its registers from the stack and resumes execution of the interrupted program.

Different processors have different ways of handling step 3 above. Some operate just as stated in 3. Each interrupt source has an interrupt service routine stored at a particular memory location. Other processor introduce a layer of indirection (a pointer) to add flexibility. When an interrupt happens in this type of processor, the CPU fetches an address from a specific memory location (called an interrupt vector) and then jumps to that address to execute the ISR.

Interrupt handling on the MSP430G2231 (Launchpad chip)

The following example is taken from Texas Instruments Launchpad example set. In this example, Timer A is used to generate a periodic interrupt. The LED on the Launchpad board is turned toggled at each interrupt.

```
//*******************************************************************************
//  MSP430F20xx Demo - Timer_A, Toggle P1.0, CCR0 Cont. Mode ISR, DCO SMCLK
//
//  Description: Toggle P1.0 using software and TA_0 ISR. Toggles every
//  50000 SMCLK cycles. SMCLK provides clock source for TACLK.
//  During the TA_0 ISR, P1.0 is toggled and 50000 clock cycles are added to
//  CCR0. TA_0 ISR is triggered every 50000 cycles. CPU is normally off and
//  used only during TA_ISR.
//  ACLK = n/a, MCLK = SMCLK = TACLK = default DCO
//
//           MSP430F20xx
//         ---------------
//     /|\|              XIN|-
//      | |               |
//      --|RST        XOUT|-
//        |               |
//        |           P1.0|-->LED
//
//  M. Buccini / L. Westlund
//  Texas Instruments Inc.
//  October 2005
//  Built with CCE Version: 3.2.0 and IAR Embedded Workbench Version: 3.40A
//*******************************************************************************
```

```
#include <msp430x20x3.h>

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT
  P1DIR |= 0x01;                            // P1.0 output
  CCTL0 = CCIE;                             // CCR0 interrupt enabled
  CCR0 = 50000;
  TACTL = TASSEL_2 + MC_2;                  // SMCLK, contmode

  _BIS_SR(LPM0_bits + GIE);                 // Enter LPM0 w/ interrupt
}

// Timer A0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  P1OUT ^= 0x01;                            // Toggle P1.0
  CCR0 += 50000;                            // Add Offset to CCR0
}
```