MSP430 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 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 (logic 0 to 1 or 1 to 0) on a particular pin on the microprocessor or by a particular level (0 or 1) on a pin. They can also be caused by circuits within the microprocessor itself. The MSP430 recognises the interrupts from many sources and of different priorities. Details are quite model specific however, the 430x44x range support the interrupts shown in Table 1(taken from TI datasheet).

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 wait for data to arrive from some serially connected device. This could be achieved by inserting instructions to read the status of some UART register throughout the controlling program. This is not a great solution however as it requires the programmer to be very disciplined (one false move and data could be lost). 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 arrival of data over the serial link. Hardware permitting, the arrival of a byte of data can be used to generate an interrupt. The processor will briefly stop what it is doing, grab the newly arrived byte (perhaps process it) 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.

Interrupt Source	Interrupt Flag	Maskable	Vector Address	Priority
Power-up, external reset, Watchdog Timer, Flash memory	WDTIFG KEYV	No (reset)	0FFFEh	15 (highest)
NMI, Oscillator Fault, Flash Memory Access Violation	NMIIFG, OFIFG, ACCVIFG	Not globally	0FFFCh	14
Timer_B7	TBCCR0 CCIFG	Yes	0FFFAh	13
Timer_B7	TBCCR1 to TBCCR6 CCIFG's TBIFG	Yes	0FFF8h	12
Comparator_A	CAIFG	Yes	0FFF6h	11
Watchdog Timer	WDTIFG	Yes	0FFF4h	10
USART0 Receive	URXIFG0	Yes	0FFF2h	9
USART0 Transmit	UTXIFG0	Yes	0FFF0h	8
ADC12	ADC12IFG	Yes	0FFEEh	7
Timer_A3	TACCR0 CCIFG	Yes	0FFECh	6
Timer_A3	TACCR1 and TACCR2 CCIFG's, TAIFG	Yes	0FFEAh	5
I/O Port P1 (8 flags)	P1IFG.0 to P1IFG.7	Yes	0FFE8h	4
USART1 Receive	URXIFG1	Yes	0FFE6h	3
USART1 Transmit	UTXIFG1	Yes	0FFE4h	2
I/O Port P2 (8 flags)	P2IFG.0 to P2IFG.7	Yes	0FFE2h	1
Basic Timer1	BTIFG	Yes	0FFE0h	0

Table 1: MSG430x44X Interrupts.

Example:
Using the basic timer interrupt. The C-Source code that follows implements a simple real time clock (apologies in advance if the basic time control register setting is a bit off).

```
//*****************************************************************************
// This program demonstrates how the Basic Timer may be used to implement a
// simple time of day clock.
//*****************************************************************************

#include  <msp430x44x.h>
int seconds;
int minutes;
int hours;
int days;
int ticks;
// This will drift and will require calibration – ok for testing though.

#define TICKS_PER_SECOND 2500
void main(void)
{

  WDTCTL = WDTPW + WDTHOLD;             // Stop WDT

  ticks = seconds = minutes = hours = days = 0; // start clock at 0 run time

  IE2 |= BTIE;                          // Enable BT interrupt
  BTCTL = BTSSEL+BTIP2+BTIP1+BTIP0;     // Divide System clock by 256
                                        // The default system clock is approx 640kHz
                                        // so this should give a tick rate of 2500Hz
  _EINT();                              // Enable interrupts

  for (;;)
  {
    _BIS_SR(CPUOFF);                    // Enter LPM0
    _NOP();                             // Required only for C-spy
  }
}

// Basic Timer interrupt service routine
interrupt[BASICTIMER_VECTOR] void basic_timer(void)
{
  ticks++;
  if (ticks >= TICKS_PER_SECOND) {
    ticks = 0;
    seconds++;
    if (seconds > 59) {
      seconds = 0;
      minutes++;
      if (minutes > 59) {
        minutes = 0;
        hours++;
        if (hours > 23) {
          hours = 0;
          days++; // day of week, calender etc. not implemented.
        }
      }
    }
  }
}
```

The details of the basic timer will be left to a later note. What is at issue here is the way in which interrupt handling is coded in C. The first step is to enable the specific interrupt involved via the interrupt enable register. This is followed by an enabling of the global interrupt flag using the _EINT() “function” - this is translated to an “eint” machine instruction. The other part of the process of using interrupts it to set the interrupt vector. The IAR C-compiler does

this work for you. The interrupt service routine must be declared as returning no values and accepting no parameters (after all, it is not called in the traditional manner). The declaration of the function must be modified using the “interrupt” modifier. This along with an argument has two effects:

1) It replaces the normal subroutine return instruction with a “reti” (return from interrupt instruction)
2) It inserts the address of the function at the specified offset within the interrupt vector table. (`BASICTIMER_VECTOR` in this case).

When using interrupts care should be taken to avoid “collisions” between sections of code that attempt concurrently use a shared resource (e.g. a port or global variable). Figure 1a shows how problems of this sort might arise arise. Figure 1b illustrates a mechanism for overcoming this problem.

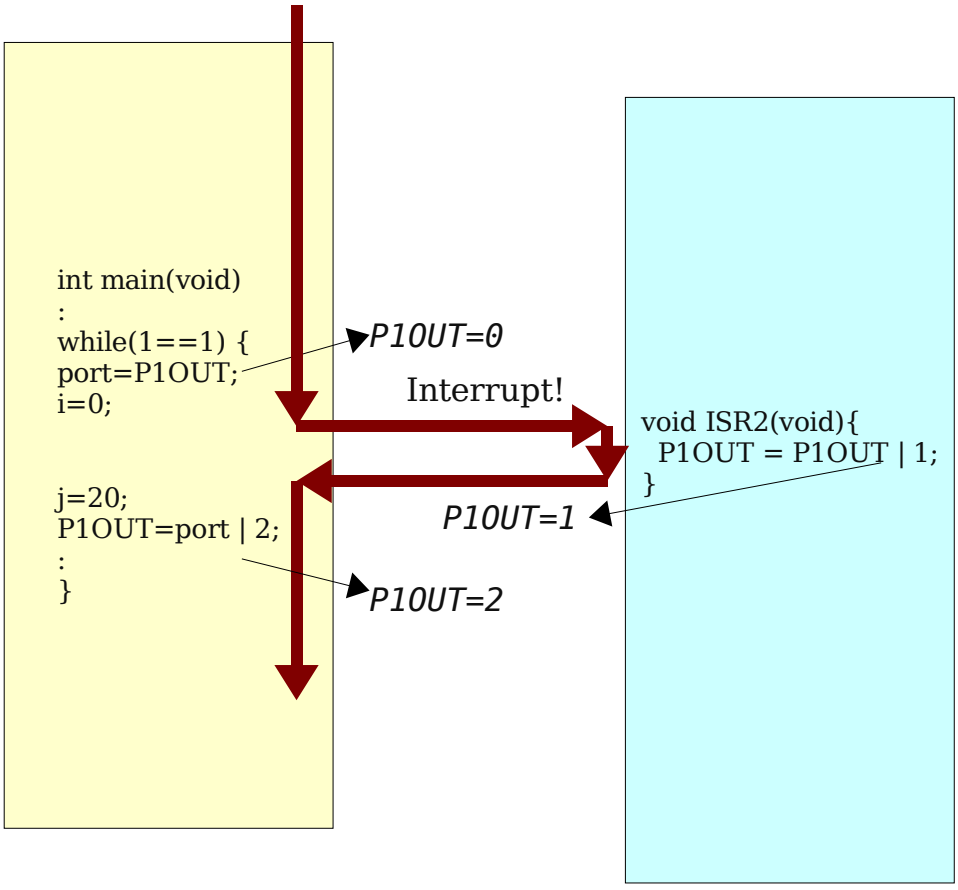

Figure 1a. The main program loop wants to set b1 of Port 1. It does this by copying the current byte of Port 1, updates the copy, and then writing the copy back. However, just after copying the Port contents, an interrupt takes place which modifies the port (sets b0). The main routine knows nothing of this and overwrites it with a stale copy. Port1.0 and Port1.1 should both be 1 after this but only Port1.1 is.

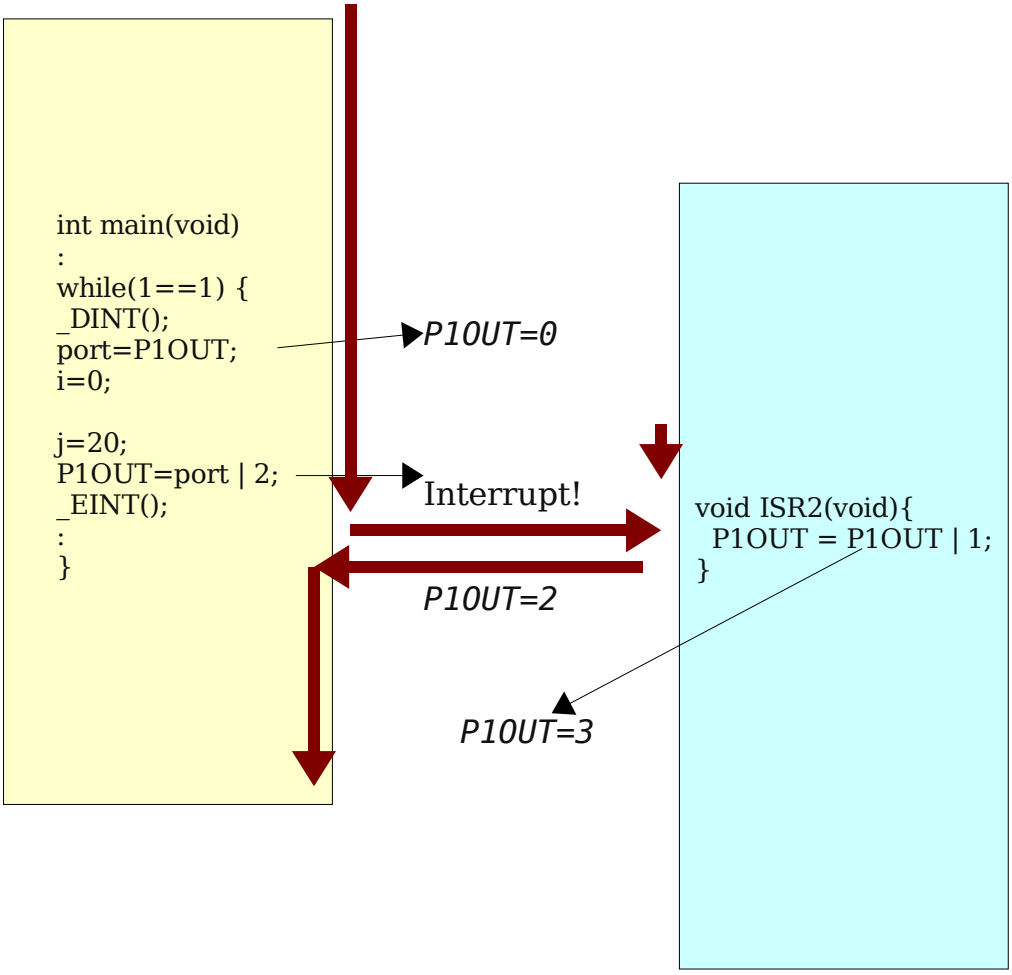

Figure 1b. In this case, the main loop disables interrupts while it is updating the port contents. This prevents the local copy of the port contents from becoming stale however it does run the risk of interrupts being lost. Some texts would describe the section of code that performs the update as a "critical section" i.e. it must not be interrupted. It is generally a good idea to keep critical sections as short as possible. In some cases other techniques such as locking (using a global/shared variable) may be more appropriate.