Introduction to Embedded System Design

Interrupts in MSP430

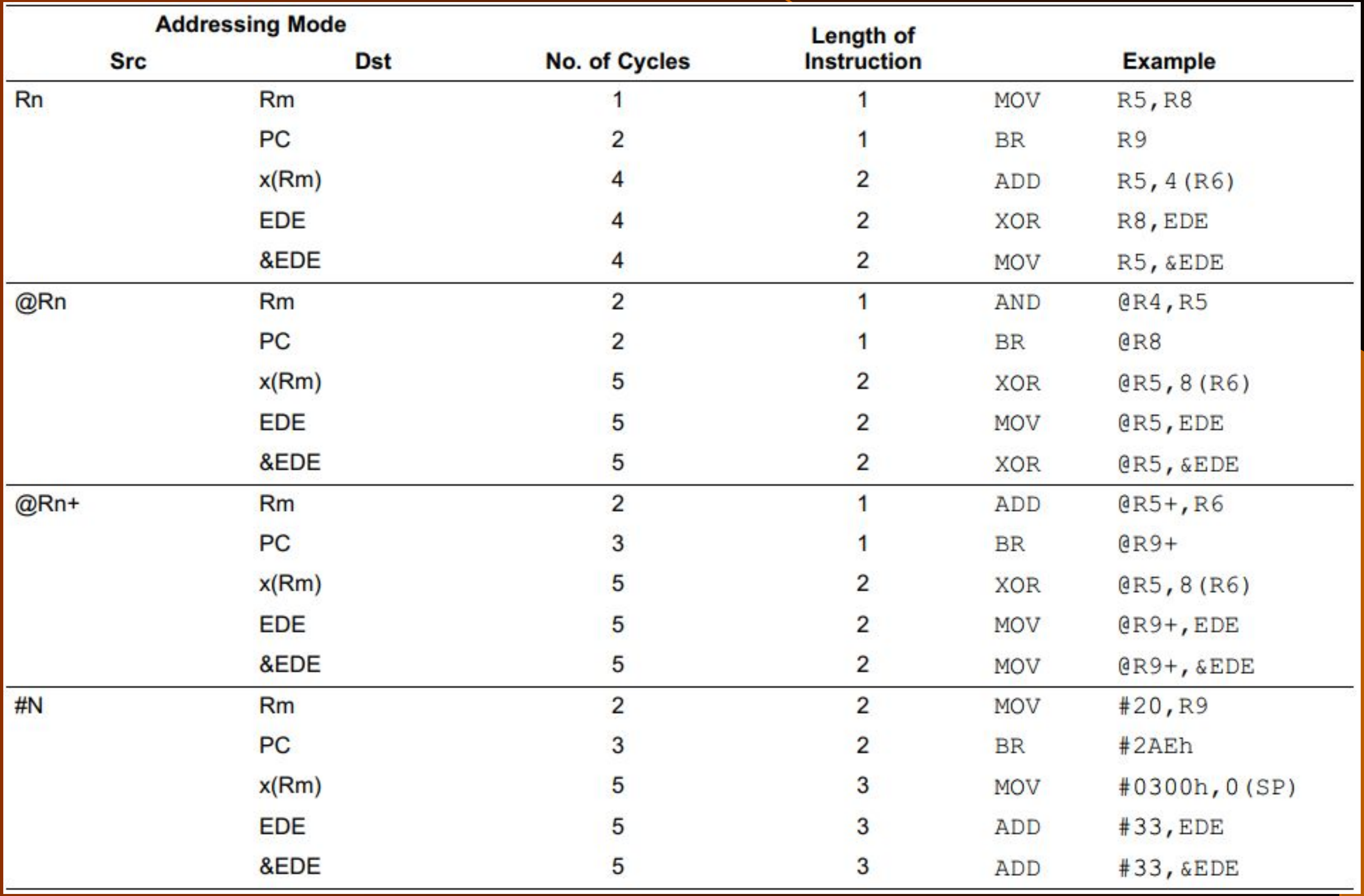
Dhananjay V. Gadre
Associate Professor
ECE Division
Netaji Subhas University of Technology, New Delhi

Badri Subudhi
Assistant Professor
Electrical Engineering Department
Indian Institute of Technology, Jammu

Why Interrupts?

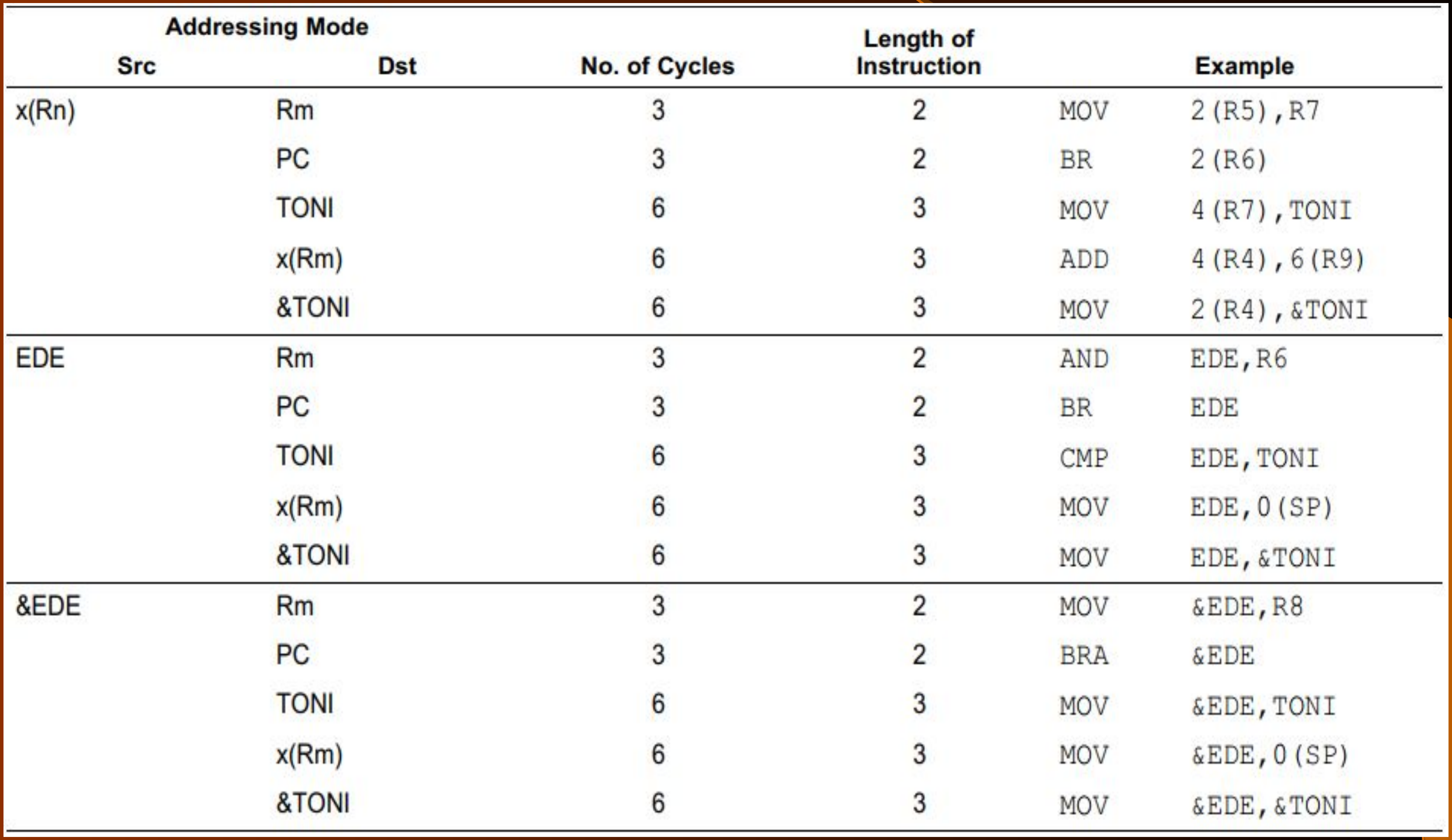
- Polling external events to fulfill computational requirements.
- Multiple inputs require round robin querying method
- Good for small number of inputs.

Format-I (Double Operand) Instruction Cycles and Lengths



Addressing Mode			Length of	Example	
Src	Dst	No. of Cycles	Instruction		
Rn	Rm	1	1	MOV	R5,R8
	PC	2	1	BR	R9
	x(Rm)	4	2	ADD	R5,4(R6)
	EDE	4	2	XOR	R8,EDE
	&EDE	4	2	MOV	R5,&EDE
@Rn	Rm	2	1	AND	@R4,R5
	PC	2	1	BR	@R8
	x(Rm)	5	2	XOR	@R5,8(R6)
	EDE	5	2	MOV	@R5,EDE
	&EDE	5	2	XOR	@R5,&EDE
@Rn+	Rm	2	1	ADD	@R5+,R6
	PC	3	1	BR	@R9+
	x(Rm)	5	2	XOR	@R5,8(R6)
	EDE	5	2	MOV	@R9+,EDE
	&EDE	5	2	MOV	@R9+,&EDE
#N	Rm	2	2	MOV	#20,R9
	PC	3	2	BR	#2AEh
	x(Rm)	5	3	MOV	#0300h,0(SP)
	EDE	5	3	ADD	#33,EDE
	&EDE	5	3	ADD	#33,&EDE

Format-I (Double Operand) Instruction Cycles and Lengths



Addressing Mode					
Src	Dst	No. of Cycles	Length of Instruction	Example	
x(Rn)	Rm	3	2	MOV	2(R5),R7
	PC	3	2	BR	2(R6)
	TONI	6	3	MOV	4(R7),TONI
	x(Rm)	6	3	ADD	4(R4),6(R9)
	&TONI	6	3	MOV	2(R4),&TONI
EDE	Rm	3	2	AND	EDE,R6
	PC	3	2	BR	EDE
	TONI	6	3	CMP	EDE,TONI
	x(Rm)	6	3	MOV	EDE,0(SP)
	&TONI	6	3	MOV	EDE,&TONI
&EDE	Rm	3	2	MOV	&EDE,R8
	PC	3	2	BRA	&EDE
	TONI	6	3	MOV	&EDE,TONI
	x(Rm)	6	3	MOV	&EDE,0(SP)
	&TONI	6	3	MOV	&EDE,&TONI

Introduction to Interrupts

An interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request or event.

Where are Interrupts Used?

- Urgent tasks that must be executed promptly at higher priority than main code as well as concurrently with the main code.
- Infrequent tasks, such as handling slow input from humans. This saves overhead of polling.
  - For example, if you continuously poll (read) an input to wait for a switch to be pressed, your CPU Load is very high. On the other hand, in case the switch is connected to a pin which is configured for an interrupt, your CPU is free to do other tasks or maybe even go to sleep!

- Waking the CPU from sleep. Particularly important for MSP430 which typically spends much of its time in LPM and can be awakened only by interrupt.

Interrupt Versus Function/Subroutine

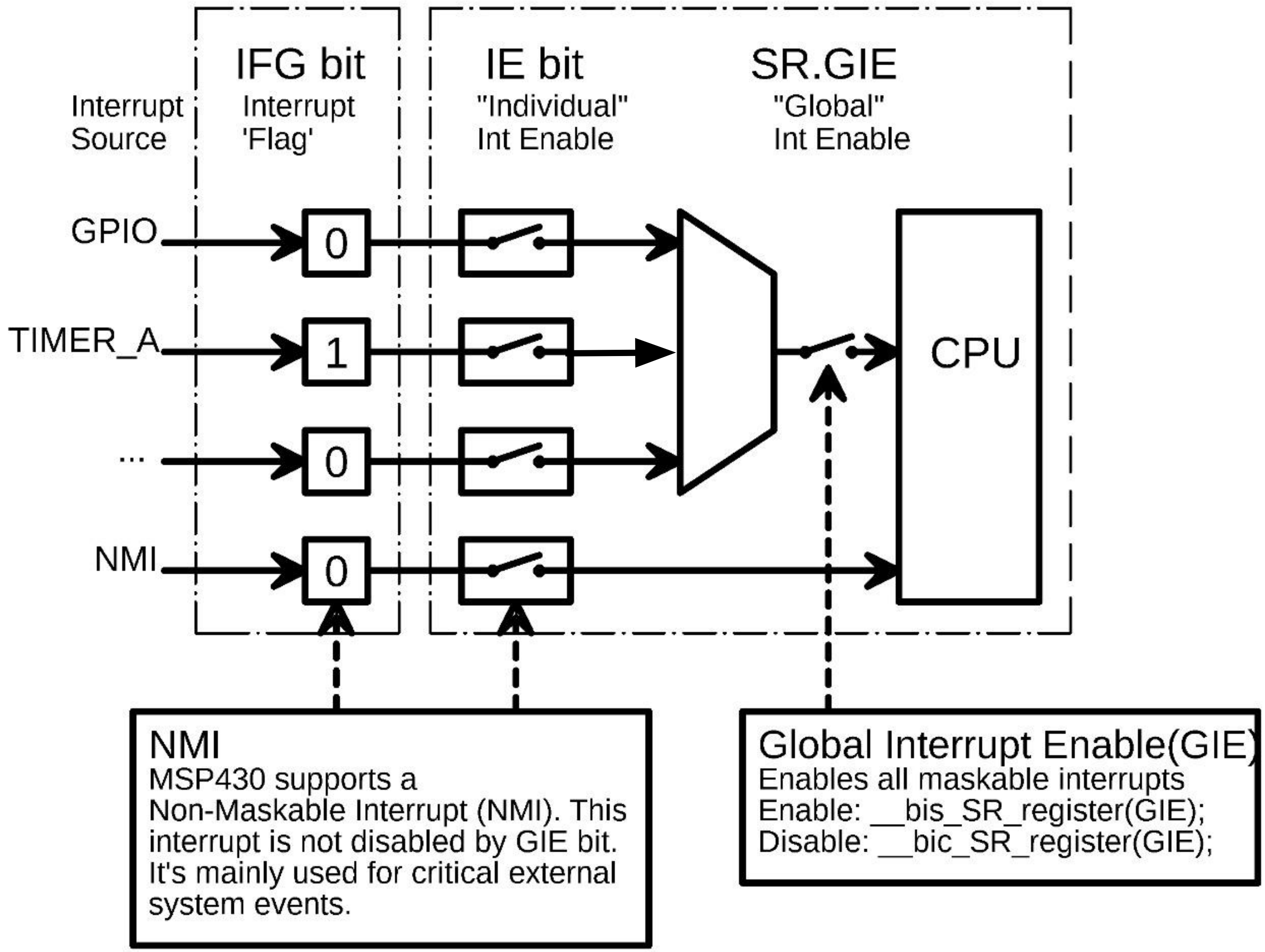
- An interrupt is initiated by an external/internal signal rather than from execution of an instruction (function call).
- An interrupt procedure usually stores all information necessary to define the state of CPU rather than storing only the program counter.
- The address of the interrupt service program is determined by hardware.

Interrupts

- The code which handles an interrupt is called an Interrupt Handler or Interrupt Service Routine (ISR).
- Interrupts can be requested by most peripheral modules like Timers, GPIOs and some in the core of the MCU, such as clock generators.
- Each interrupt has a flag, which is raised (set) when condition of interrupt occurs.
- There are three types of interrupts in MSP430:
    - System Reset
    - Non Maskable Interrupts
    - Maskable Interrupts

Maskable vs Non-Maskable Interrupts

- Most interrupts are maskable (i.e. they can be suspended).
- They are effective only if general interrupt enable (GIE) is set in Status Register (SR).
- Non-maskable interrupts cannot be suppressed by GIE, but are enabled by individual interrupt enable bits.



NMI
MSP430 supports a Non-Maskable Interrupt (NMI). This interrupt is not disabled by GIE bit. It's mainly used for critical external system events.

Global Interrupt Enable(GIE)
Enables all maskable interrupts
Enable: __bis_SR_register(GIE);
Disable: __bic_SR_register(GIE);

Non Maskable Interrupts

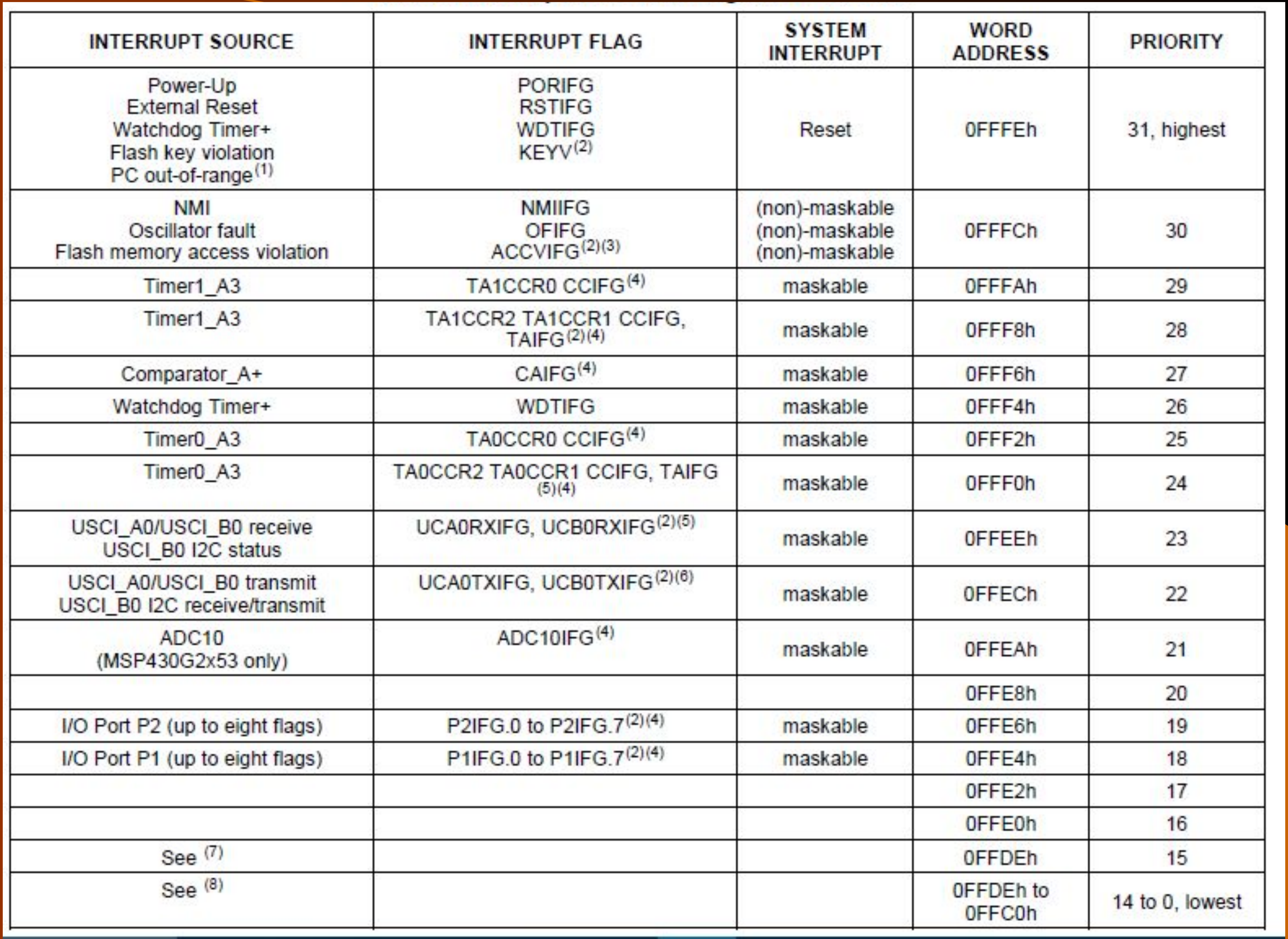
- A Non-maskable NMI interrupt can be generated by three sources:
  - Oscillator Fault, OFIFG.
  - Access violation to memory.
  - NMI (non maskable interrupt) pin if it has been configured for interrupt rather than reset.

RST/NMI Pin

- At power-up, the RST/NMI pin is configured as the reset pin.
- The function of the RST/NMI pins is selected in the watchdog control register WDTCTL.
- If the RST/NMI pin is set to the reset function, the CPU is held in the reset state as long as the RST/NMI pin is held low. After the input changes to a high state, the CPU starts program execution at the word address stored in the reset vector, 0FFFEh, and the RSTIFG flag is set.
- If the RST/NMI pin is configured by user software as NMI, a signal edge selected by the WDTNMIES bit generates an NMI interrupt if the NMIIE bit is set. The RST/NMI flag NMIIFG is also set.

Vectored Interrupts

- MSP430 uses vectored interrupt.
- In a vectored interrupt, the source that interrupts supplies the branch information to the computer.
- Vector address is stored in the vector table.
- Each interrupt vector has a distinct priority, which is used to select which vector is taken if more than 1 interrupt is active.
- Priority is fixed: cannot be changed by user. A higher address means higher priority.
- Reset vector has highest address 0xFFFE.



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
Timer1_A3	TA1CCR0 CCIFG(4)	maskable	0FFFAh	29
Timer1_A3	TA1CCR2 TA1CCR1 CCIFG, TAIFG(2)(4)	maskable	0FFF8h	28
Comparator_A+	CAIFG(4)	maskable	0FFF6h	27
Watchdog Timer+	WDTIFG	maskable	0FFF4h	26
Timer0_A3	TA0CCR0 CCIFG(4)	maskable	0FFF2h	25
Timer0_A3	TA0CCR2 TA0CCR1 CCIFG, TAIFG (5)(4)	maskable	0FFF0h	24
USCI_A0/USCI_B0 receive USCI_B0 I2C status	UCA0RXIFG, UCB0RXIFG(2)(5)	maskable	0FFEEh	23
USCI_A0/USCI_B0 transmit USCI_B0 I2C receive/transmit	UCA0TXIFG, UCB0TXIFG(2)(6)	maskable	0FFECh	22
ADC10 (MSP430G2x53 only)	ADC10IFG(4)	maskable	0FFEAh	21
			0FFE8h	20
I/O Port P2 (up to eight flags)	P2IFG.0 to P2IFG.7(2)(4)	maskable	0FFE6h	19
I/O Port P1 (up to eight flags)	P1IFG.0 to P1IFG.7(2)(4)	maskable	0FFE4h	18
			0FFE2h	17
			0FFE0h	16
See (7)			0FFDEh	15
See (8)			0FFDEh to 0FFC0h	14 to 0, lowest

Interrupt Flag

- The interrupt flag is cleared automatically for vectors that have a single source.
- Flags remain set for servicing by software if the vector has multiple sources.

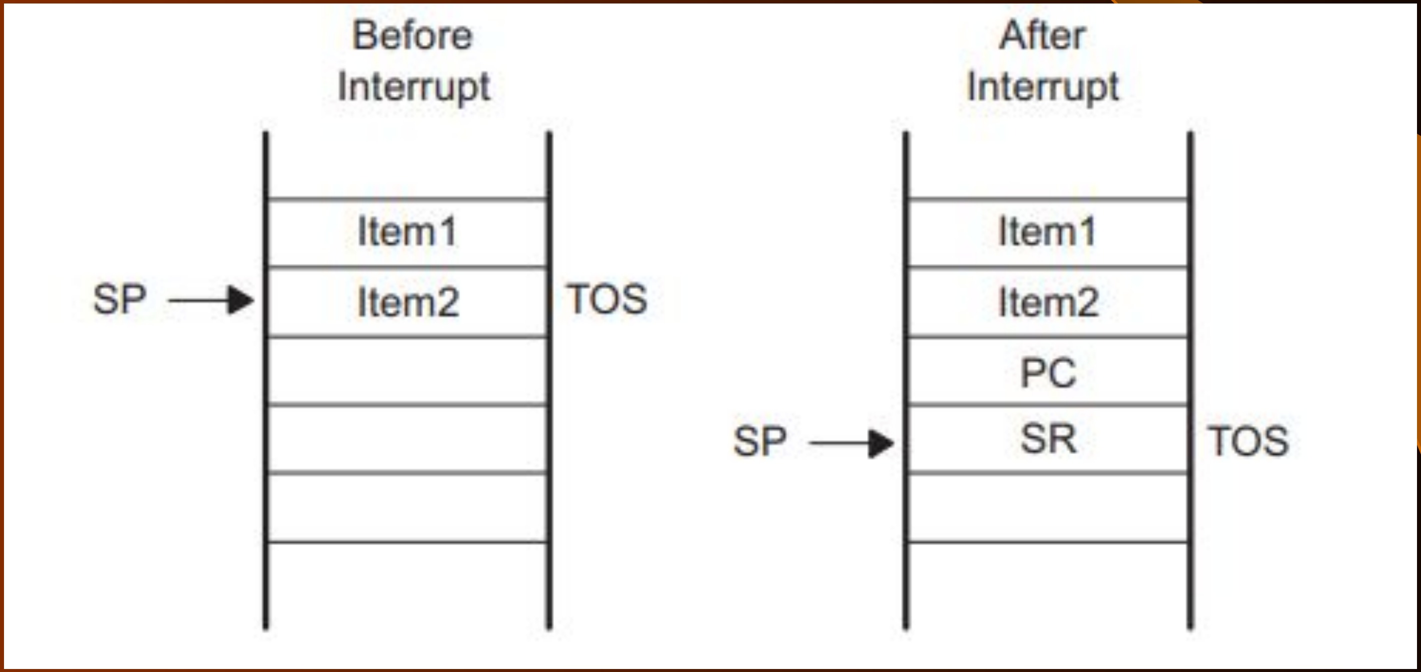
Useful points to remember:

- Keep interrupts short. Avoid delay functions.
- While servicing ISR, other urgent ISRs are disabled unless nested interrupts are allowed.
- If lengthy processing is needed, do minimum in ISR and send a message to main function, by setting a flag.

Interrupt Acceptance

1. Any currently executing instruction is completed.
2. The PC, which points to the next instruction, is pushed onto the stack.
3. The SR(Status Register) is pushed onto the stack.
4. The interrupt with the highest priority is selected if multiple interrupts occurred during the last instruction and are pending for service.
5. The interrupt flag resets automatically on single-source flags. Multiple source flags remain set for servicing by software.
6. The SR is cleared(for use within the ISR). This terminates any low-power mode. Because the GIE bit is cleared, further interrupts are disabled.
7. The content of the interrupt vector is loaded into the PC: the program continues with the interrupt service routine at that address.

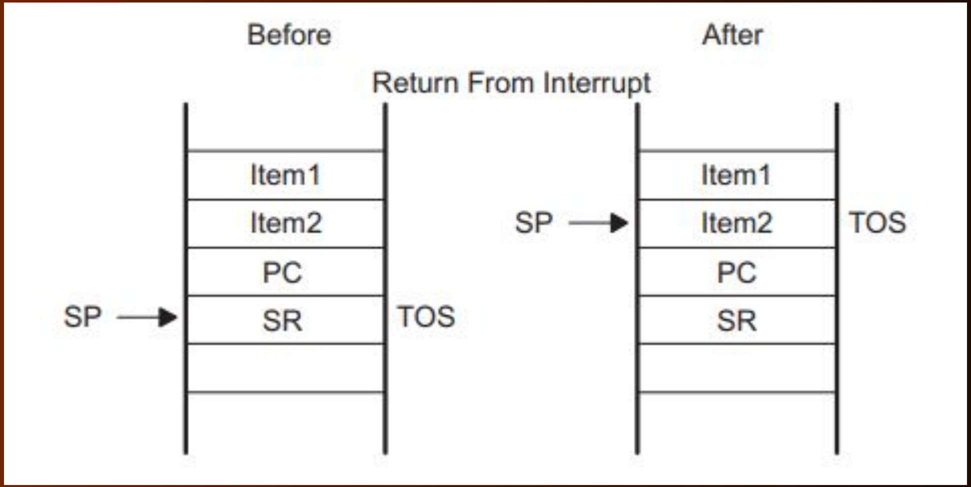
Interrupt Acceptance (State of the RAM)



Returning from an Interrupt

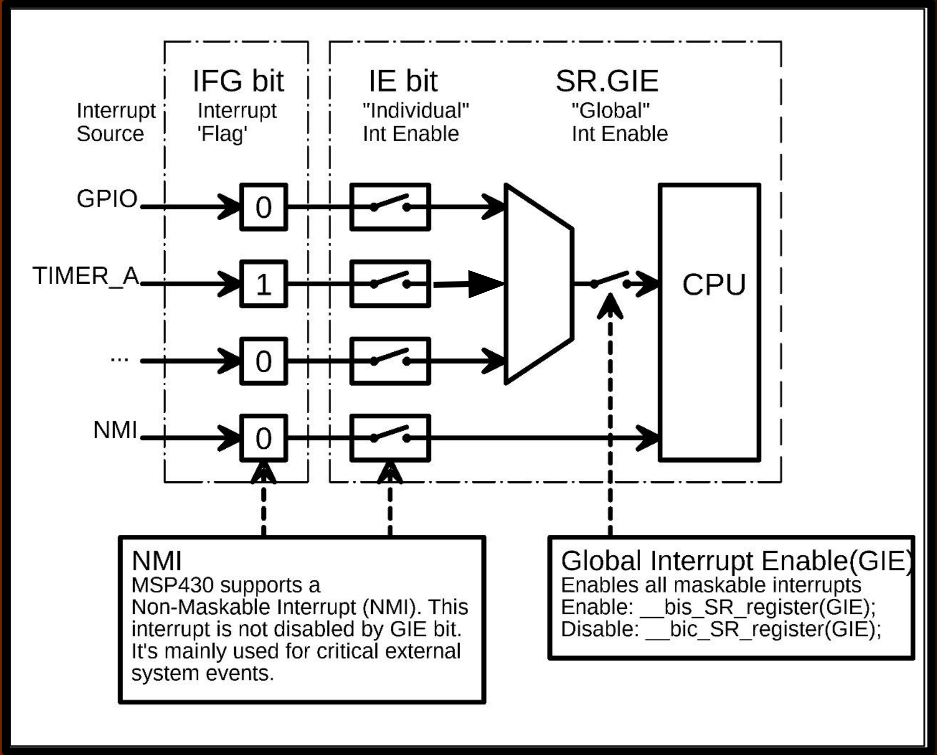
1. The SR with all previous settings pops from the stack. All previous settings of GIE, CPUOFF, etc. are now in effect, regardless of the settings used during the interrupt service routine.

2. The PC pops from the stack and begins execution at the point where it was interrupted.



Interrupt Registers for GPIO

- All these registers are 8-bit registers.
- PxIE - Interrupt enable register
- PxIES - Interrupt edge selection register
- PxIFG - Interrupt flag register



NMI
MSP430 supports a Non-Maskable Interrupt (NMI). This interrupt is not disabled by GIE bit. It's mainly used for critical external system events.

Global Interrupt Enable(GIE)
Enables all maskable interrupts
Enable: __bis_SR_register(GIE);
Disable: __bic_SR_register(GIE);

PxIE

- PxIE - Interrupt enable register
- 8 Bit Register corresponding to 8 pins on each port.
- Setting any bit as ‘0’ in this register disables the interrupt for the corresponding pin.
- All the bits in this register are 0 by default.
- Setting any bit as ‘1’ in the register enables the interrupt for the corresponding pin.

PxIES

- PxIES - Interrupt edge selection register
- 8 bit register corresponding to 8 pins on each port.
- The bits of this register select the interrupt edge transition type on the corresponding pins.
- Setting a bit as ‘0’ in the register enables interrupt flag when transition is from low to high.
- Setting the bit as ‘1’ in the register enables interrupt flag when transition is from high to low.

PxIFG

- PxIFG - Interrupt flag register
- 8 bit register corresponding to 8 pins on each port.
- A bit of the interrupt flag register is set when the selected interrupt edge occurs on the corresponding pin.
- Using software PxIFG bits can be set or reset.
- PxIFG bits must be reset after the interrupt via software. Setting the PxIFG via software can generate software initiated interrupts.

Interrupt Code Flow

ISR in C

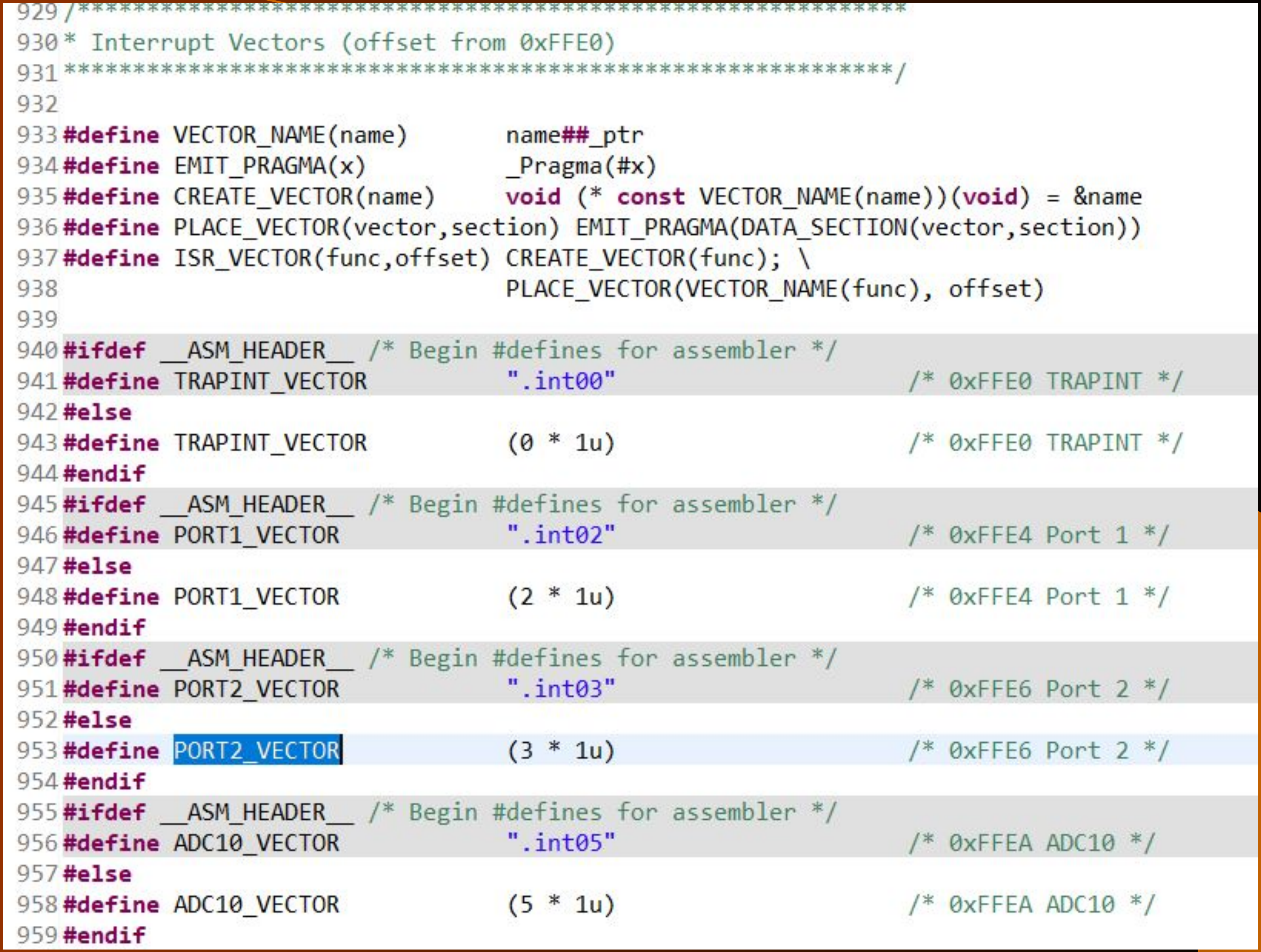
For example:

```
#pragma vector = TIMER0_A0_VECTOR
__interrupt void CCR0_ISR(void)
```

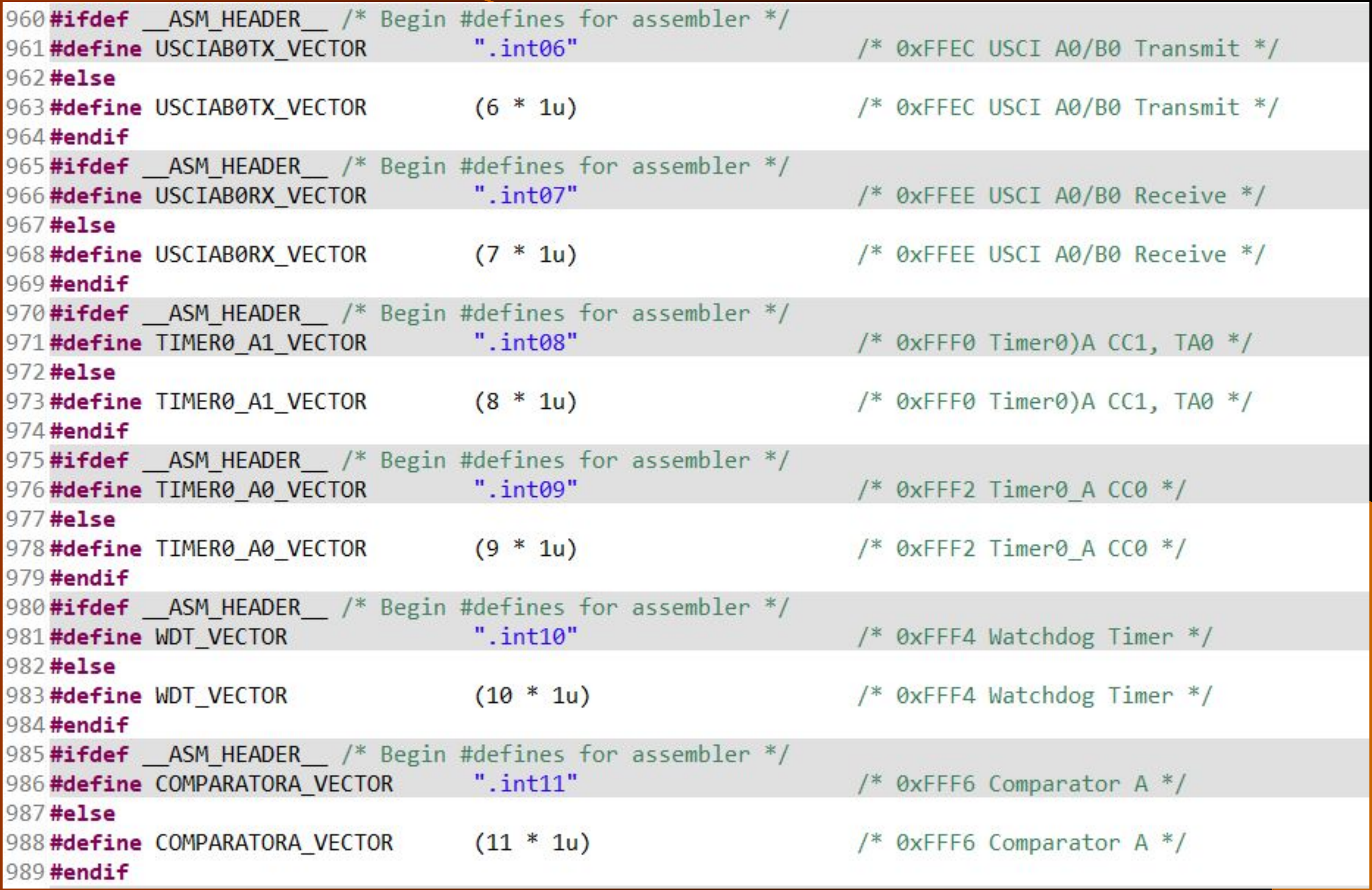
Some extensions are needed by compiler to differentiate a standard function and an ISR.

- *#pragma* associates the function with a particular interrupt vector. This directive is a special-purpose directive that you can use to turn on or off certain features.
- __interrupt keyword in the beginning of the next line that names the function.
- No significance to the name of the function; it is the name of the vector that matters. Name of function is not used in program.
- __enable_interrupt in main program sets the GIE bit and turns on interrupts.

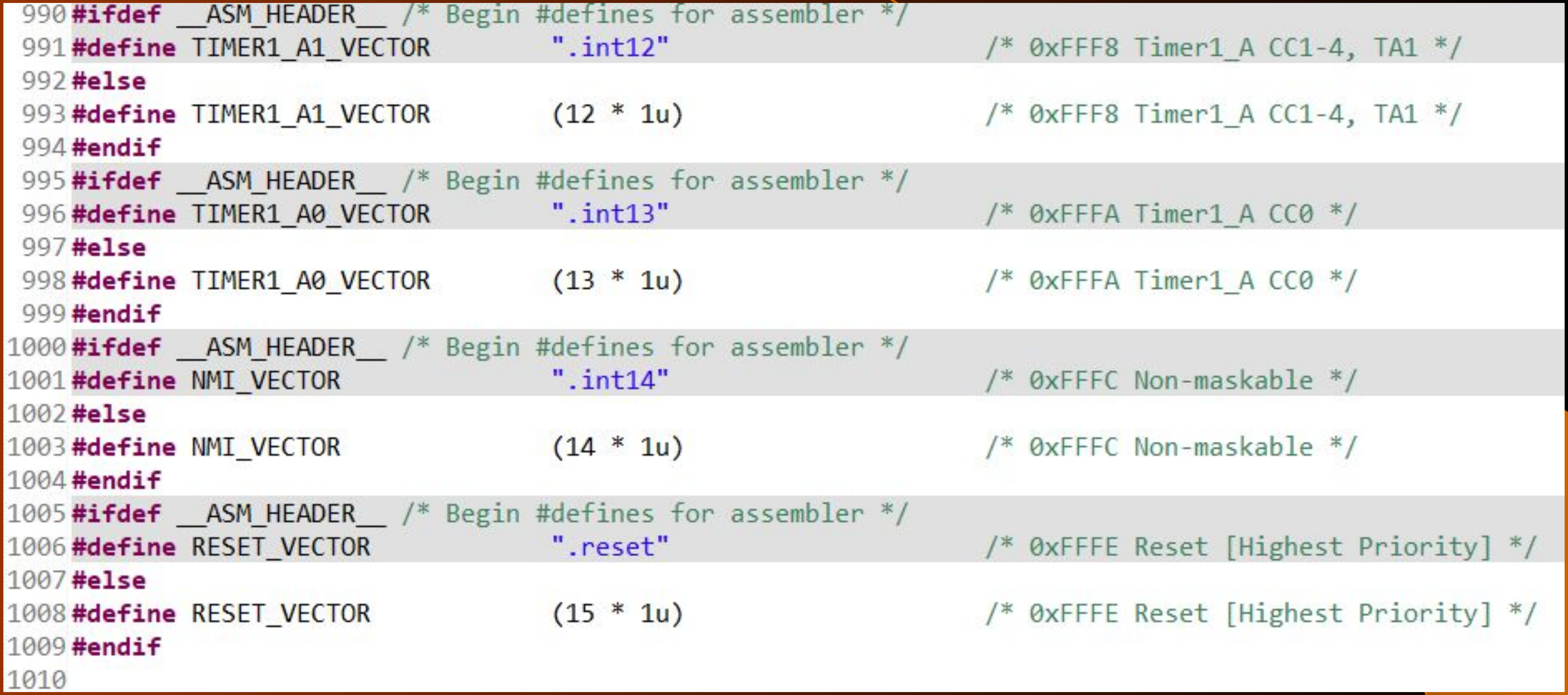
ISR names present in CCS for MSP430 (In MSP430.h header file)



```
/************************************************************
* Interrupt Vectors (offset from 0xFFE0)
************************************************************/

#define VECTOR_NAME(name)       name##_ptr
#define EMIT_PRAGMA(x)          _Pragma(#x)
#define CREATE_VECTOR(name)     void (* const VECTOR_NAME(name))(void) = &name
#define PLACE_VECTOR(vector,section) EMIT_PRAGMA(DATA_SECTION(vector,section))
#define ISR_VECTOR(func,offset) CREATE_VECTOR(func); \
                                PLACE_VECTOR(VECTOR_NAME(func), offset)

#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define TRAPINT_VECTOR          ".int00"                     /* 0xFFE0 TRAPINT */
#else
#define TRAPINT_VECTOR          (0 * 1u)                     /* 0xFFE0 TRAPINT */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define PORT1_VECTOR            ".int02"                     /* 0xFFE4 Port 1 */
#else
#define PORT1_VECTOR            (2 * 1u)                     /* 0xFFE4 Port 1 */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define PORT2_VECTOR            ".int03"                     /* 0xFFE6 Port 2 */
#else
#define PORT2_VECTOR            (3 * 1u)                     /* 0xFFE6 Port 2 */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define ADC10_VECTOR            ".int05"                     /* 0xFFEA ADC10 */
#else
#define ADC10_VECTOR            (5 * 1u)                     /* 0xFFEA ADC10 */
#endif
```



```
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define USCIAB0TX_VECTOR        ".int06"                     /* 0xFFEC USCI A0/B0 Transmit */
#else
#define USCIAB0TX_VECTOR        (6 * 1u)                     /* 0xFFEC USCI A0/B0 Transmit */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define USCIAB0RX_VECTOR        ".int07"                     /* 0xFFEE USCI A0/B0 Receive */
#else
#define USCIAB0RX_VECTOR        (7 * 1u)                     /* 0xFFEE USCI A0/B0 Receive */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define TIMER0_A1_VECTOR        ".int08"                     /* 0xFFF0 Timer0)A CC1, TA0 */
#else
#define TIMER0_A1_VECTOR        (8 * 1u)                     /* 0xFFF0 Timer0)A CC1, TA0 */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define TIMER0_A0_VECTOR        ".int09"                     /* 0xFFF2 Timer0_A CC0 */
#else
#define TIMER0_A0_VECTOR        (9 * 1u)                     /* 0xFFF2 Timer0_A CC0 */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define WDT_VECTOR              ".int10"                     /* 0xFFF4 Watchdog Timer */
#else
#define WDT_VECTOR              (10 * 1u)                    /* 0xFFF4 Watchdog Timer */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define COMPARATORA_VECTOR      ".int11"                     /* 0xFFF6 Comparator A */
#else
#define COMPARATORA_VECTOR      (11 * 1u)                    /* 0xFFF6 Comparator A */
#endif
```



```
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define TIMER1_A1_VECTOR        ".int12"                   /* 0xFFF8 Timer1_A CC1-4, TA1 */
#else
#define TIMER1_A1_VECTOR        (12 * 1u)                  /* 0xFFF8 Timer1_A CC1-4, TA1 */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define TIMER1_A0_VECTOR        ".int13"                   /* 0xFFFA Timer1_A CC0 */
#else
#define TIMER1_A0_VECTOR        (13 * 1u)                  /* 0xFFFA Timer1_A CC0 */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define NMI_VECTOR              ".int14"                   /* 0xFFFC Non-maskable */
#else
#define NMI_VECTOR              (14 * 1u)                  /* 0xFFFC Non-maskable */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define RESET_VECTOR            ".reset"                   /* 0xFFFE Reset [Highest Priority] */
#else
#define RESET_VECTOR            (15 * 1u)                  /* 0xFFFE Reset [Highest Priority] */
#endif
```

What should be done in an ISR

- Save system context (done automatically)
- Re-enable interrupts if required.
- The interrupt code
- If multi-source interrupt, then read associated register to determine source and clear the flag.
- Restore system context (done automatically)
- Return

Variables for Interrupts

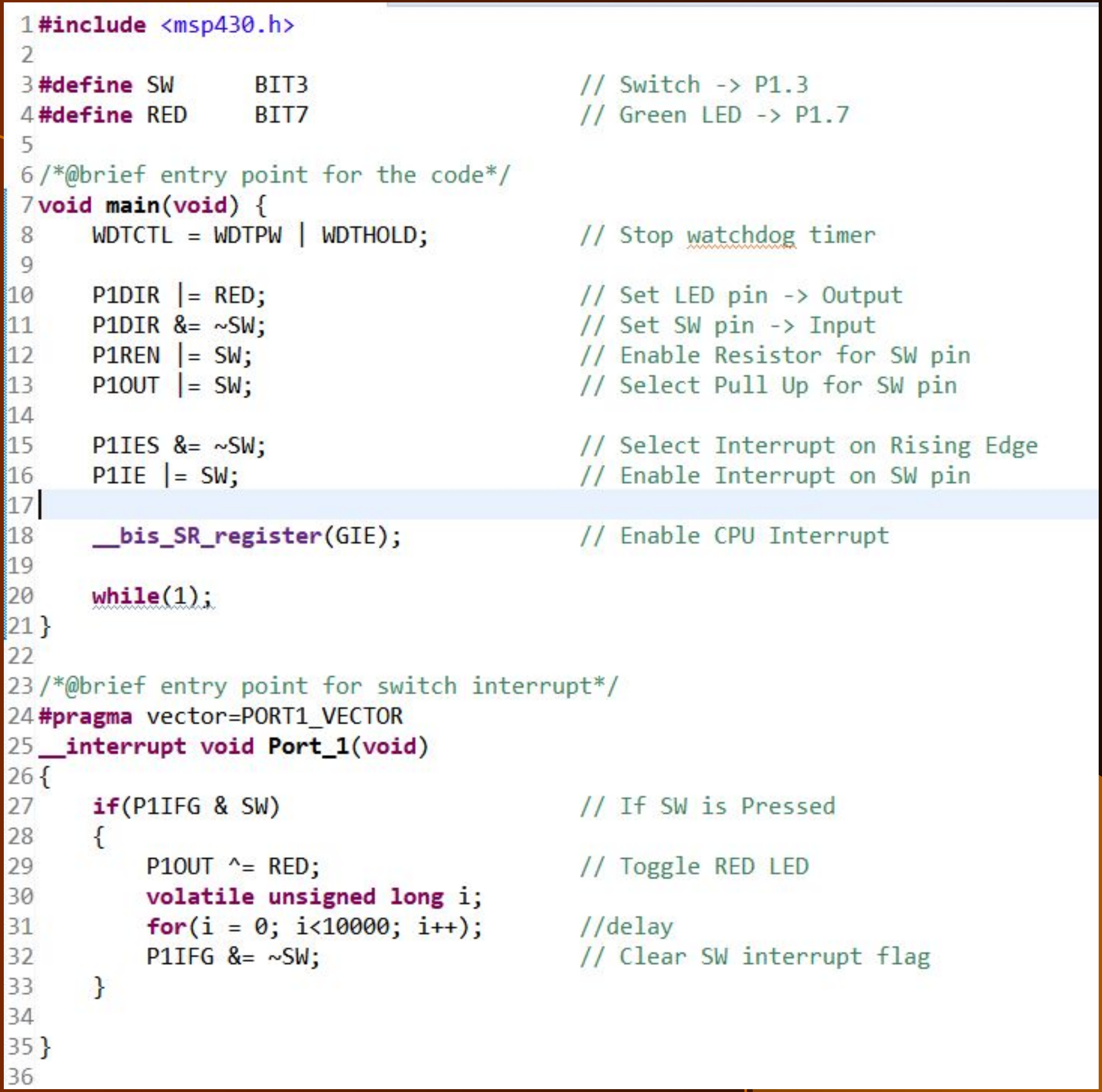
- Volatile: By declaring a variable as volatile, the compiler gets to know that the value of the variable is susceptible to frequent changes (with no direct action of the program) and hence the compiler does not keep a copy of the variable in a register (like cache). This prevents the program to misinterpret the value of this variable.
- As a thumb rule: variables associated with input ports and interrupts should be declared as volatile.

Check List

- Enabled interrupts both in their module and generally (GIE bit)?
- Provided ISR for all the enabled interrupts?
- Included code to acknowledge interrupt that share a vector even if only 1 source is active?
- Are the variables declared as volatile?

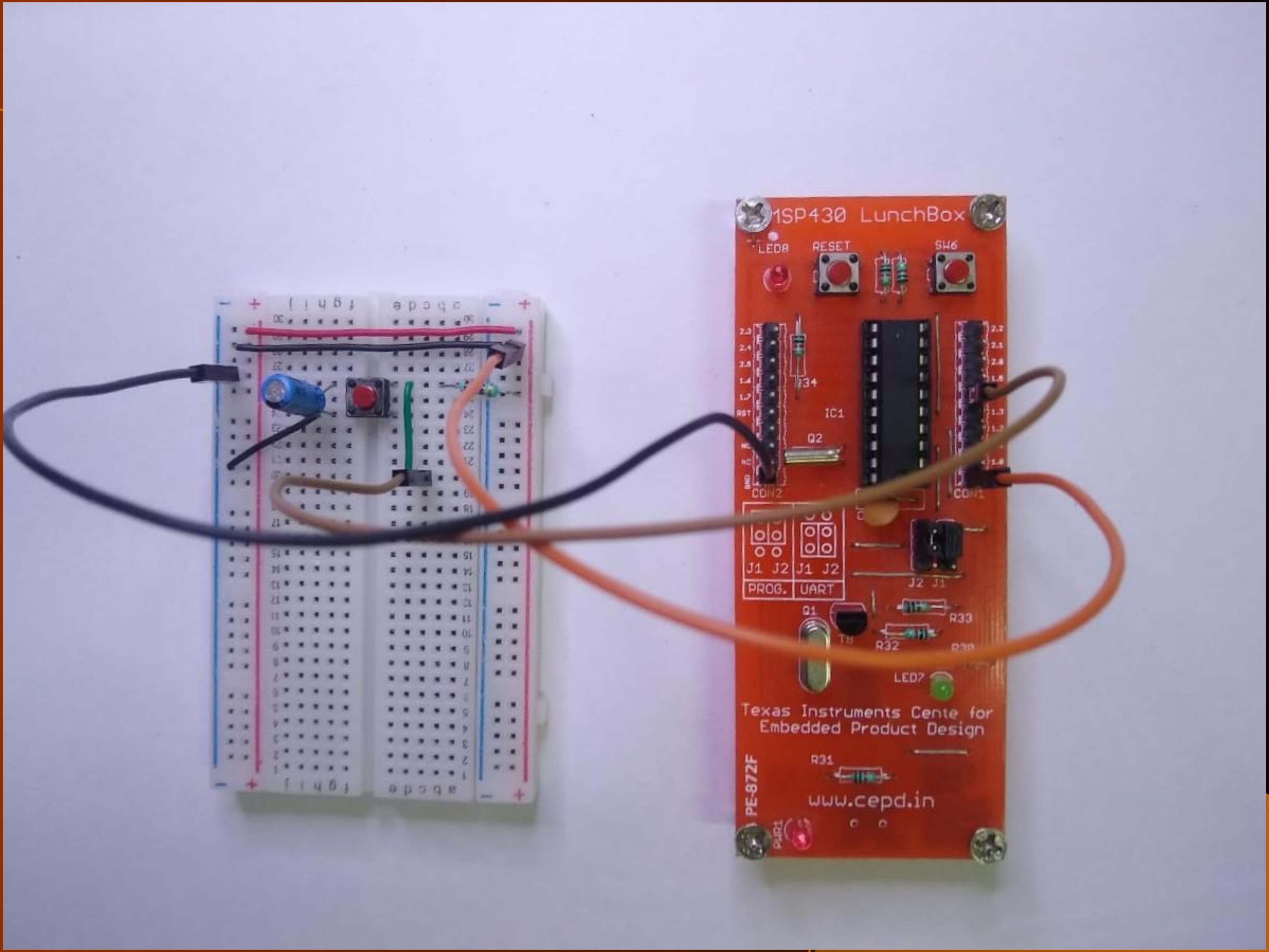
Switch Interfacing using Interrupts

Example Code 1(Bad ISR): HelloInterrupt

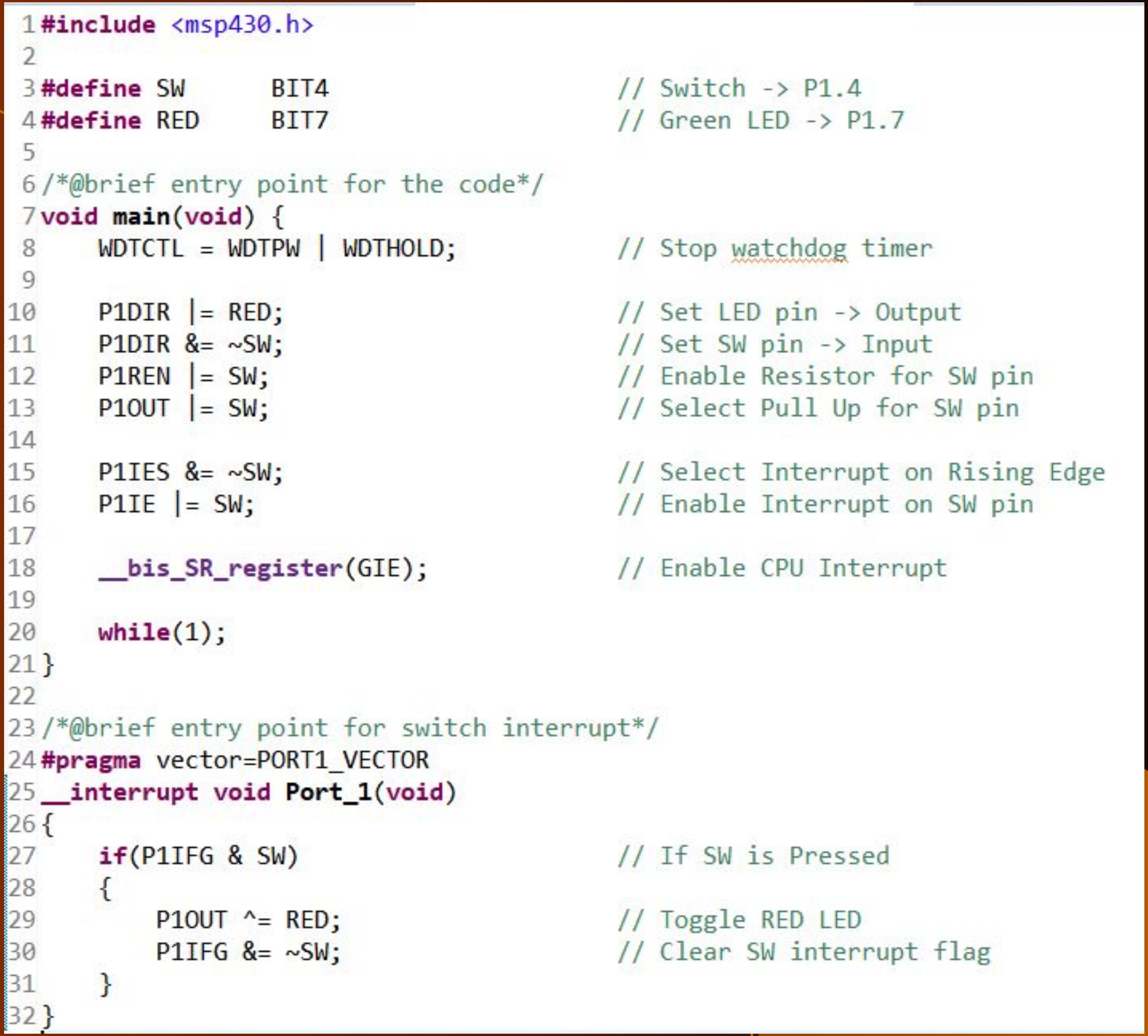


```c
#include <msp430.h>

#define SW      BIT3                    // Switch -> P1.3
#define RED     BIT7                    // Green LED -> P1.7

/*@brief entry point for the code*/
void main(void) {
    WDTCTL = WDTPW | WDTHOLD;           // Stop watchdog timer

    P1DIR |= RED;                       // Set LED pin -> Output
    P1DIR &= ~SW;                       // Set SW pin -> Input
    P1REN |= SW;                        // Enable Resistor for SW pin
    P1OUT |= SW;                        // Select Pull Up for SW pin

    P1IES &= ~SW;                       // Select Interrupt on Rising Edge
    P1IE |= SW;                         // Enable Interrupt on SW pin

    __bis_SR_register(GIE);             // Enable CPU Interrupt

    while(1);
}

/*@brief entry point for switch interrupt*/
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
    if(P1IFG & SW)                      // If SW is Pressed
    {
        P1OUT ^= RED;                   // Toggle RED LED
        volatile unsigned long i;
        for(i = 0; i<10000; i++);       //delay
        P1IFG &= ~SW;                   // Clear SW interrupt flag
    }

}

```

External Switch

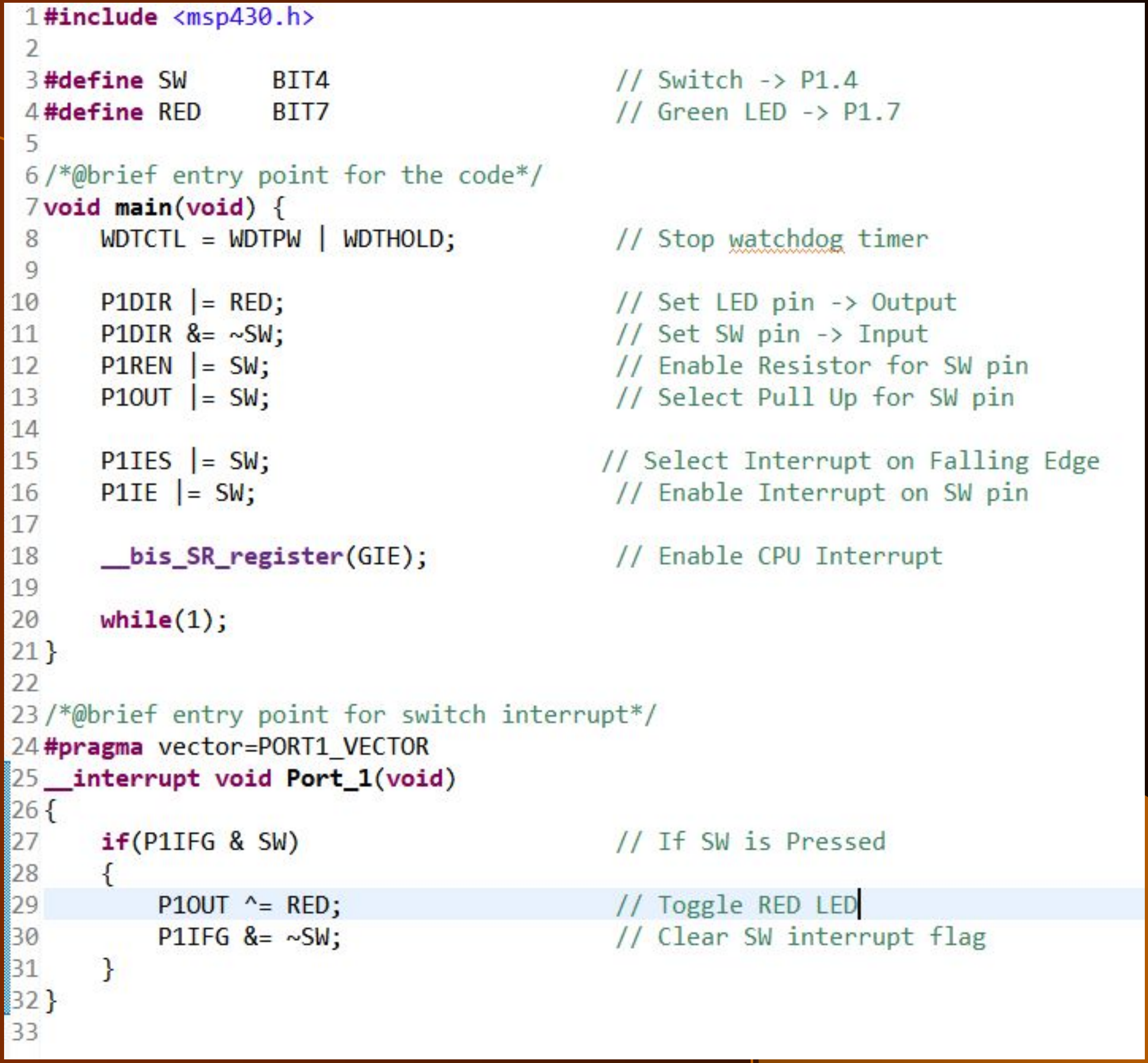


Example Code 2(Better ISR): HelloInterrupt_Rising



```c
#include <msp430.h>

#define SW      BIT4                    // Switch -> P1.4
#define RED     BIT7                    // Green LED -> P1.7

/*@brief entry point for the code*/
void main(void) {
    WDTCTL = WDTPW | WDTHOLD;           // Stop watchdog timer

    P1DIR |= RED;                       // Set LED pin -> Output
    P1DIR &= ~SW;                       // Set SW pin -> Input
    P1REN |= SW;                        // Enable Resistor for SW pin
    P1OUT |= SW;                        // Select Pull Up for SW pin

    P1IES &= ~SW;                       // Select Interrupt on Rising Edge
    P1IE |= SW;                         // Enable Interrupt on SW pin

    __bis_SR_register(GIE);             // Enable CPU Interrupt

    while(1);
}

/*@brief entry point for switch interrupt*/
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
    if(P1IFG & SW)                      // If SW is Pressed
    {
        P1OUT ^= RED;                   // Toggle RED LED
        P1IFG &= ~SW;                   // Clear SW interrupt flag
    }
}
```

Example Code 3: HelloInterrupt_Falling



```c
#include <msp430.h>

#define SW      BIT4                     // Switch -> P1.4
#define RED     BIT7                     // Green LED -> P1.7

/*@brief entry point for the code*/
void main(void) {
    WDTCTL = WDTPW | WDTHOLD;            // Stop watchdog timer

    P1DIR |= RED;                        // Set LED pin -> Output
    P1DIR &= ~SW;                        // Set SW pin -> Input
    P1REN |= SW;                         // Enable Resistor for SW pin
    P1OUT |= SW;                         // Select Pull Up for SW pin

    P1IES |= SW;                        // Select Interrupt on Falling Edge
    P1IE |= SW;                          // Enable Interrupt on SW pin

    __bis_SR_register(GIE);              // Enable CPU Interrupt

    while(1);
}

/*@brief entry point for switch interrupt*/
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
    if(P1IFG & SW)                       // If SW is Pressed
    {
        P1OUT ^= RED;                    // Toggle RED LED
        P1IFG &= ~SW;                    // Clear SW interrupt flag
    }
}

```

Exercise

Modify the existing code to toggle LED1 when SW1 is pressed and to toggle LED2 when SW2 is pressed.

Thank you!