MSP430 Teaching Materials

Week 5

Functions, Subroutines and Interrupts

Hacettepe University

Functions and Subroutines

- ❑ It is good practice to break programs into short functions or subroutines, for reasons that are explained in any textbook on programming:
- ❑ It makes programs easier to write and more reliable to test and maintain.
- ❑ Functions are obviously useful for code that is called from more than one place but should be used much more widely, to encapsulate every distinct function.
- ❑ They hide the detailed implementation of an activity from the high-level, strategic level of the software.
- ❑ Functions can readily be reused and incorporated into libraries, provided that their documentation is clear.

What Happens when a Subroutine Is Called?

- The call instruction first causes the return address, which is the current value in the PC, to be pushed on to the stack.
- The address of the subroutine is then loaded into the PC and execution continues from there.
- At the end of the subroutine the ret instruction pops the return address off the stack into the PC so that execution resumes with the instruction following the call of the subroutine.

- Nothing else is done automatically when a subroutine is called. This means that the subroutine inherits the existing contents of the CPU registers, including the status register.
- The behavior is different for interrupts, as we see shortly. Therefore a subroutine and its calling routine must agree on whether the contents of these registers should be preserved or may be overwritten.
- This must be made clear in the documentation. Such as:

```
call #DelayTenths ; Call subroutine: don't forget #
;-----------------------------------------------------------------------
; Subroutine to give delay of R12 *0.1s
; Parameter is passed in R12 and destroyed
; R4 is used for loop counter but is not saved and restored
; Works correctly if R12 = 0: the test executed first as in while (){}
;-----------------------------------------------------------------------
```

- It is always wise to follow a convention for the use of registers and this becomes essential when assembly code is mixed with C.
  - The scratch registers **R12 to R15** are used for **parameter passing** and hence are **not normally preserved** across the call.
  - The other general-purpose registers, **R4 to R11**, are used mainly for **register variables and temporary results** and must be preserved across a call. This means that you must save the contents of any register that you wish to use and restore its original contents at the end.

Storage for Local Variables

- Most functions need local variables and there are three ways in which space for these can be allocated:
  - CPU registers are simple and fast. The convention is that registers R4–R11 may be used and their values should be preserved. This is done by pushing their values on to the stack.

Storage for Local Variables

- A second approach is to use a fixed location in RAM; There are two serious disadvantages to this approach. The first is that the space in RAM is reserved permanently, even when the function is not being called, which is wasteful. Second, the function is not *reentrant*. This means that the function cannot call a second copy of itself, either directly or with nested functions between. Both copies would try to use the same variable and interfere with each other, unless the intention was for them to share the variable. This would obviously wreck a software delay loop, for instance.

Storage for Local Variables

- The third approach is to allocate variables on the stack and is generally used when a program has run out of CPU registers. This can be slow in older designs of processors, but the MSP430 can address variables on the stack with its general indexed and indirect modes. Of course it is still faster to use registers in the CPU when these are available.

R4 used for loop counter , stacked and restored

```
DelayTenths:
push.w R4 ; Stack R4: will be overwritten in prog.
jmp LoopTest ; Start with test in case R12 = 0 (passed param)
OuterLoop:
    mov.w #DELAYLOOPS ,R4 ; Initialize loop counter
    DelayLoop: ; [clock cycles in brackets]
          dec.w R4 ; Decrement loop counter [1]
    jnz DelayLoop ; Repeat loop if not zero [2]
    dec.w R12 ; Decrement number of 0.1s delays (parameter)
    LoopTest:
    cmp.w #0,R12 ; Finished number of 0.1s delays?
jnz OuterLoop ; No: go around delay loop again
pop.w R4 ; Yes: restore R4 before returning
ret ; Return to caller
```

Operation of Stack

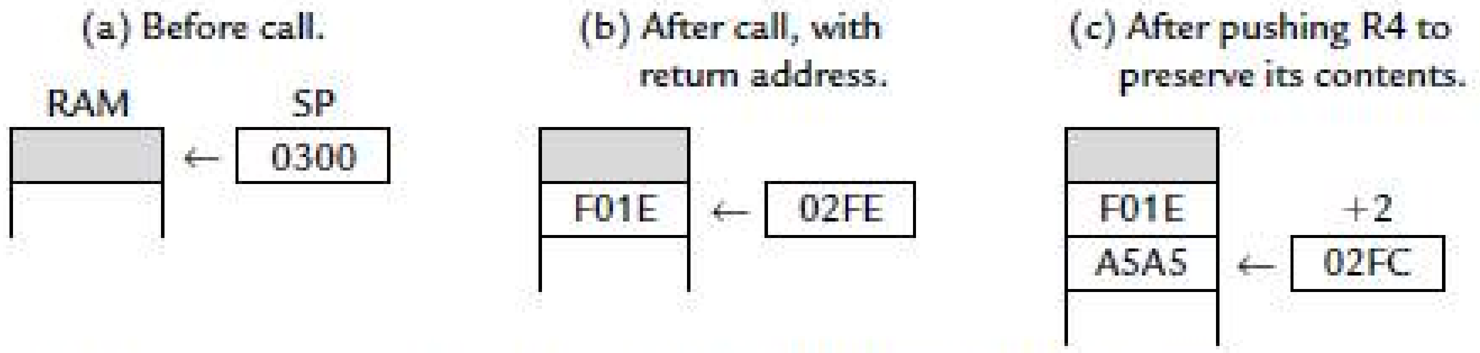

(a) Before call.
RAM
SP
0300
(b) After call, with return address.
F01E
02FE
(c) After pushing R4 to preserve its contents.
F01E
A5A5
+2
02FC

Figure 6.1: Operation of the stack in the MSP430F1121A for the subroutine in Listing 6.1. The stack is shown in words, all values are in hexadecimal, and the top of RAM is at 0x02FF in this device. Register R4 held 0xA5A5 on entry to the subroutine.

Stack as variable storage

```
InfLoop: ; Loop forever
push.w #5 ; Push delay parameter on to stack
call #DelayTenths ; Call subroutine: don't forget the #!
incd.w SP ; Release space used for parameter
xor.b #LED1 ,& P2OUT ; Toggle LED
jmp InfLoop ; Back around infinite loop
; Iterations of delay loop for about 0.1s (6 cycles/iteration ):
BIGLOOPS EQU 130
LITTLELOOPS EQU 100
;-----------------------------------------------------------
DelayTenths:
sub.w #4,SP ; Allocate 2 words (4 bytes) on stack
jmp LoopTest ; Start with test
OuterLoop:
mov.w #BIGLOOPS ,2(SP) ; Initialize big loop counter
BigLoop:
    mov.w #LITTLELOOPS ,0(SP) ; Initialize little loop counter
    LittleLoop: ; [clock cycles in brackets]
            dec.w 0(SP) ; Decrement little loop counter [4]
    jnz LittleLoop ; Repeat loop if not zero [2]
    dec.w 2(SP) ; Decrement big loop counter [4]
jnz BigLoop ; Repeat loop if not zero [4]
dec.w 6(SP) ; Decrement number of 0.1s delays
LoopTest:
cmp.w #0,6(SP) ; Finished number of 0.1s delays?
jnz OuterLoop ; No: go around delay loop again
add.w #4,SP ; Yes: finished , release space on stack
ret ; Return to caller
```

*; Subroutine to give delay of n*0.1s*
; Parameter n is passed on stack
; Space for two loop counters created on stack. After this:
; 0(SP) is innermost (little) loop counter; 2(SP) is big loop counter
; 4(SP) is return address ; 6(SP) is parameter n passed on stack

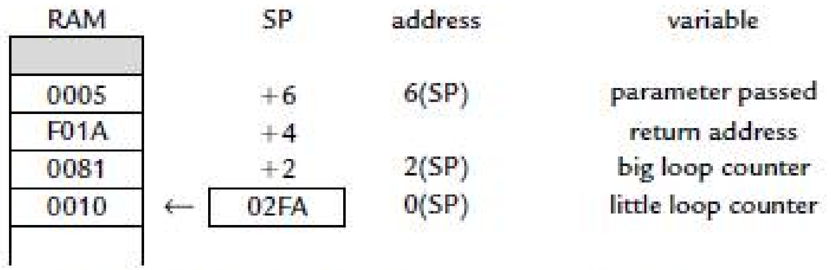


Figure 6.3: Stack in the MSP430F1121A for the subroutine in Listing 6.2, showing the parameter passed, return address and two local variables.

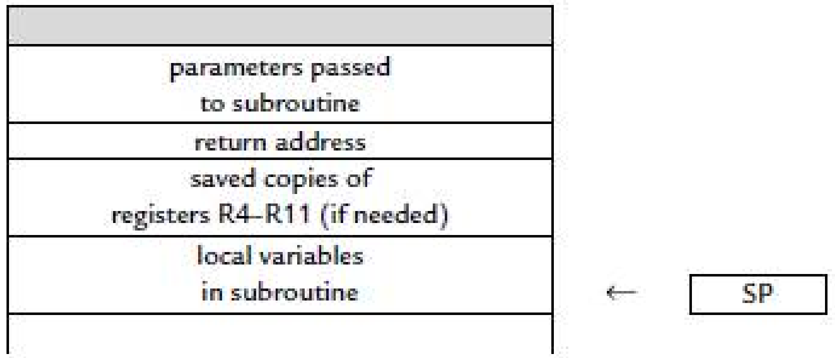

parameters passed to subroutine
return address
saved copies of registers R4–R11 (if needed)
local variables in subroutine
SP

Figure 6.4: Complete stack frame for a subroutine with parameters passed, return address, saved copies of registers, and local variables.

Stack Processes

(a) Stack after initialization.

MSB $n+1$	LSB n	address		SP
▨	▨	0x0280	←	0280
		0x027E		

(b) Stack after `push.w` #0x1234.

▨	▨	0x0280		
12	34	0x027E	←	027E
		0x027C		

(c) Stack after `push.b` #0x56 followed by `push.w` #0x789A.

▨	▨	0x0280		
12	34	0x027E		
▨	56	0x027C		
78	9A	0x027A	←	027A
		0x0278		

(d) Stack after `pop.w` R15.

▨	▨	0x0280		
12	34	0x027E		
▨	56	0x027C	←	027C
78	9A	0x027A		
		0x0278		

Mixing C and Assembly Language

- Check first to see whether an intrinsic function is available to do the job without leaving C. Many of these are declared in the header file intrinsics.h to perform functions that are not possible in standard C. For example, the _ _swap_bytes() intrinsic function calls the swapb instruction.
- Another possibility, when only a line or two of assembly language is needed, use:

  asm("mov.b &P1IN,&dest").
- The third method is to write a complete subroutine in assembly language and call it from C.

WHAT IS AN INTERRUPT?

- A signal indicating the occurrence of an event that needs immediate CPU attention
- Provides for a more efficient event handling than using polling
- Less CPU cycles wasted
- Advantages
  - Compact & Modular Code
  - Allow for Reduced Energy Consumption
  - Faster Multi-event Response Time

Interrupts

- They are like functions but with the critical distinction that they are requested by **hardware** at **unpredictable times** rather than called by **software** in an **orderly manner**

Interrupts

- Interrupts are commonly used for a range of applications:
  - **Urgent tasks** that must be executed promptly at higher priority than the main code. However, it is even faster to execute a task directly by hardware if this is possible.
  - **Infrequent tasks**, such as handling slow input from humans. This saves the overhead of regular polling.
  - **Waking the CPU from sleep**. This is particularly important in the MSP430, which typically spends much of its time in a **low-power mode** and can be awakened only by an interrupt.
  - **Calls to an operating system.** These are often processed through a trap or software interrupt instruction but the MSP430 does not have one.

A computer has 2 basic ways to react to inputs:

- **1)polling: The processor regularly looks at the input and reacts as appropriate.**
- **+easy to implement and debug**
- **-processor intensive**
  - if event is rare, you waste a lot of time checking
  - processor can't go into low power (slow or stopped) modes pgp(pp)
- **2)interrupts: The processor is "interrupted" by an event.**
- **+very efficient time-wise: no time wasted looking for an event that hasn't occurred.**
- **+very efficient energy-wise: processor can be asleep most of the time.**
- **-can be hard to debug**

This program sets P1.0 based on state of P1.4.

```
#include  <msp430x20x3.h>

void main(void)
{
  WDTCTL=WDTPW+WDTHOLD; // Stop watchdog
  P1DIR = 0x01;         // P1.0 output
  P1OUT =  0x10;        // P1.4 hi (pullup)
  P1REN |= 0x10;        // P1.4 pullup

  while (1)  {
    // Test P1.4
    if (0x10 & P1IN) P1OUT |= 0x01;
    else P1OUT &= ~0x01;
  }
}
```

This program toggles P1.0 on each push of P1.4.

```
#include  <msp430x20x3.h>

void main(void) {
  WDTCTL = WDTPW + WDTHOLD; // Stop watchdog
  P1DIR = 0x01;             // P1.0 output
  P1OUT =  0x10;            // P1.4 hi (pullup)
  P1REN |= 0x10;            // P1.4 pullup
  P1IE |= 0x10;             // P1.4 IRQ enabled
  P1IES |= 0x10;            // P1.4 Hi/lo edge
  P1IFG &= ~0x10;           // P1.4 IFG cleared

  _BIS_SR(LPM4_bits + GIE); // Enter LPM4
}

// Port 1 interrupt service routine
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void) {
  P1OUT ^= 0x01;            // P1.0 = toggle
  P1IFG &= ~0x10;           // P1.4 IFG cleared
}
```

INTERRUPT IDENTIFICATION METHODS

- Non-vectored Systems
  - Single, multidrop interrupt request line
  - Single ISR for all devices
  - CPU identifies source by polling service request (SRQ) flags

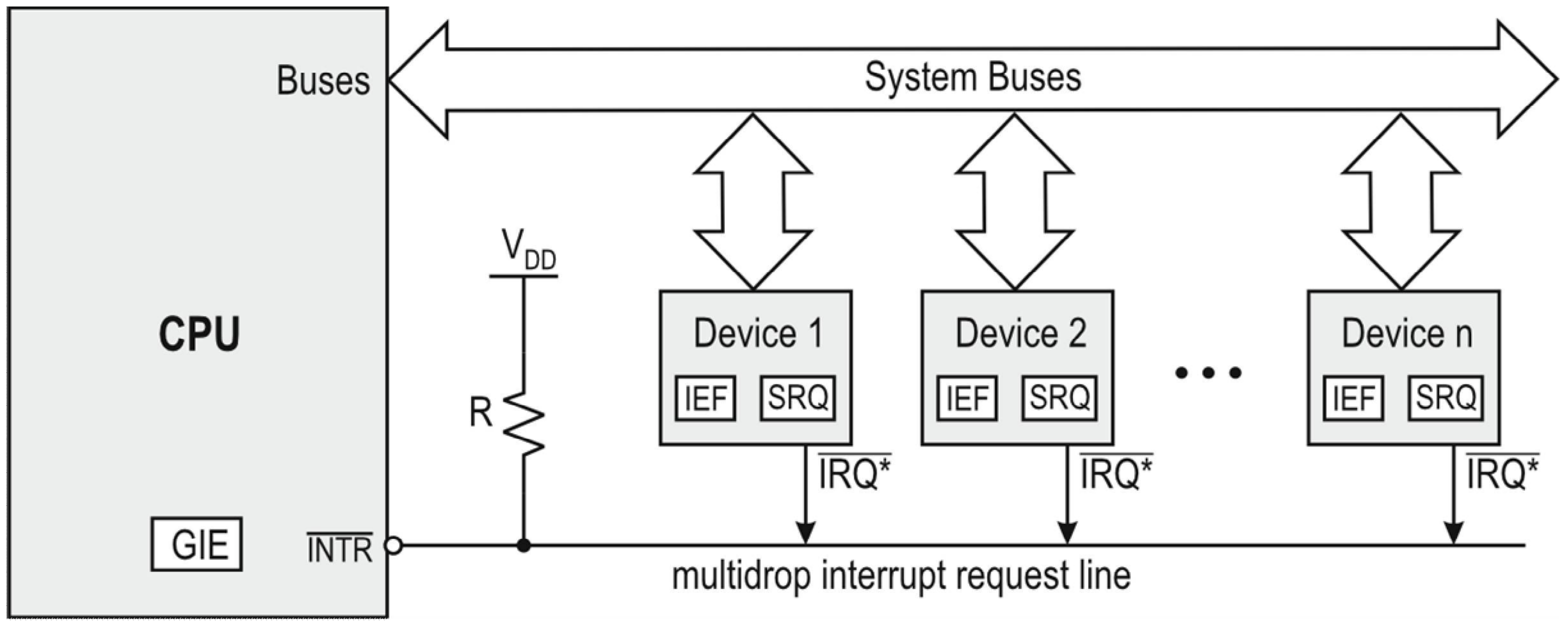


Fig. 7.2 Device interfaces in a system managing non-vectored interrupts

INTERRUPT IDENTIFICATION METHODS

- Vectored Interrupts
  - Require an Interrupt Acknowledgment (INTA) cycle
  - Interfaces generate an ID number (vector) upon INTA
  - ID number allows calculating ISR location
- Auto-vectored Interrupts
  - Each device has a fixed vector or fixed ISR address
  - No INTA cycle or vector issuing required
  - CPU loads direct ISR address into PC to execute ISR
  - Method used in MSP430 MCUs

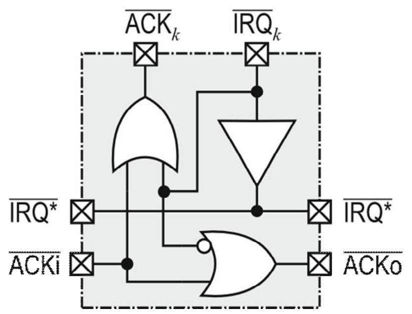
PRIORITY HANDLING (1/3)

- Interrupt Priority Management
  - Strategy to resolve multiple, simultaneous interrupt requests
  - Priority scheme decides which one is served first
- Non-vectored Systems
  - Polling order of SRQ flags decides priority
- Vectored & Auto-vectored Systems
  - Hardware supported
  - Daisy Chain-based
  - Interrupt Controller-based

PRIORITY HANDLING (2/3)

- Daisy Chain-based Arbitration
  - Devices linked by a daisy-chain element
    - Simple to implement
  - Hardwired priorities
    - The closer the device to the CPU the higher the priority

Fig. 7.4 Logic structure of a daisy chain link



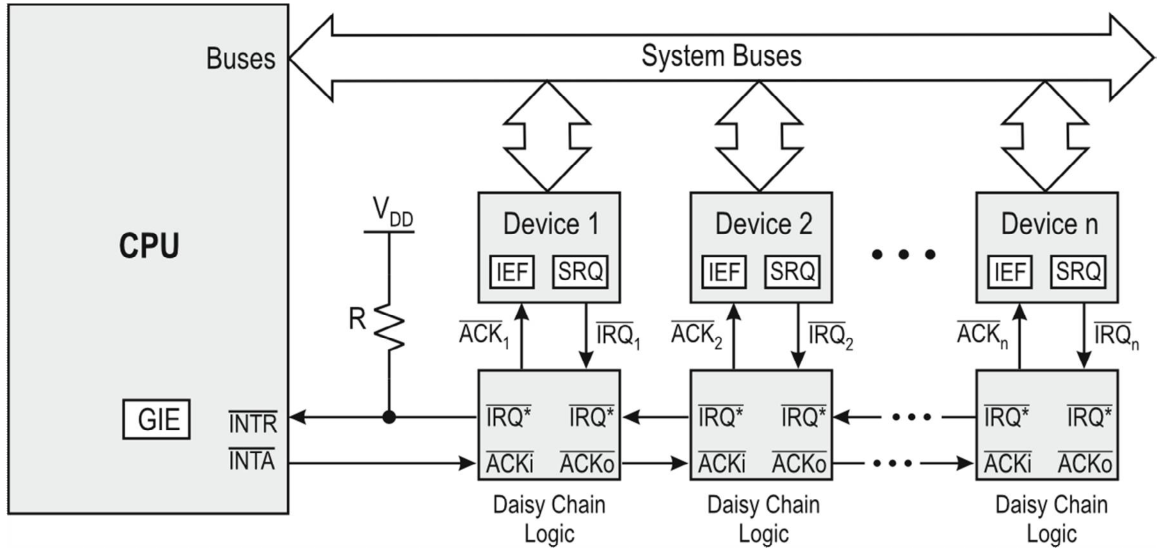


Fig. 7.3 Device interfaces connected in a daisy chain

PRIORITY HANDLING (3/3)

- ❑ Interrupt Controller-based Arbitration
  - ❑ Uses central arbiter for resolving priorities
  - ❑ Reduces interface overhead for vectored systems
  - ❑ Allows for configuring the priority scheme

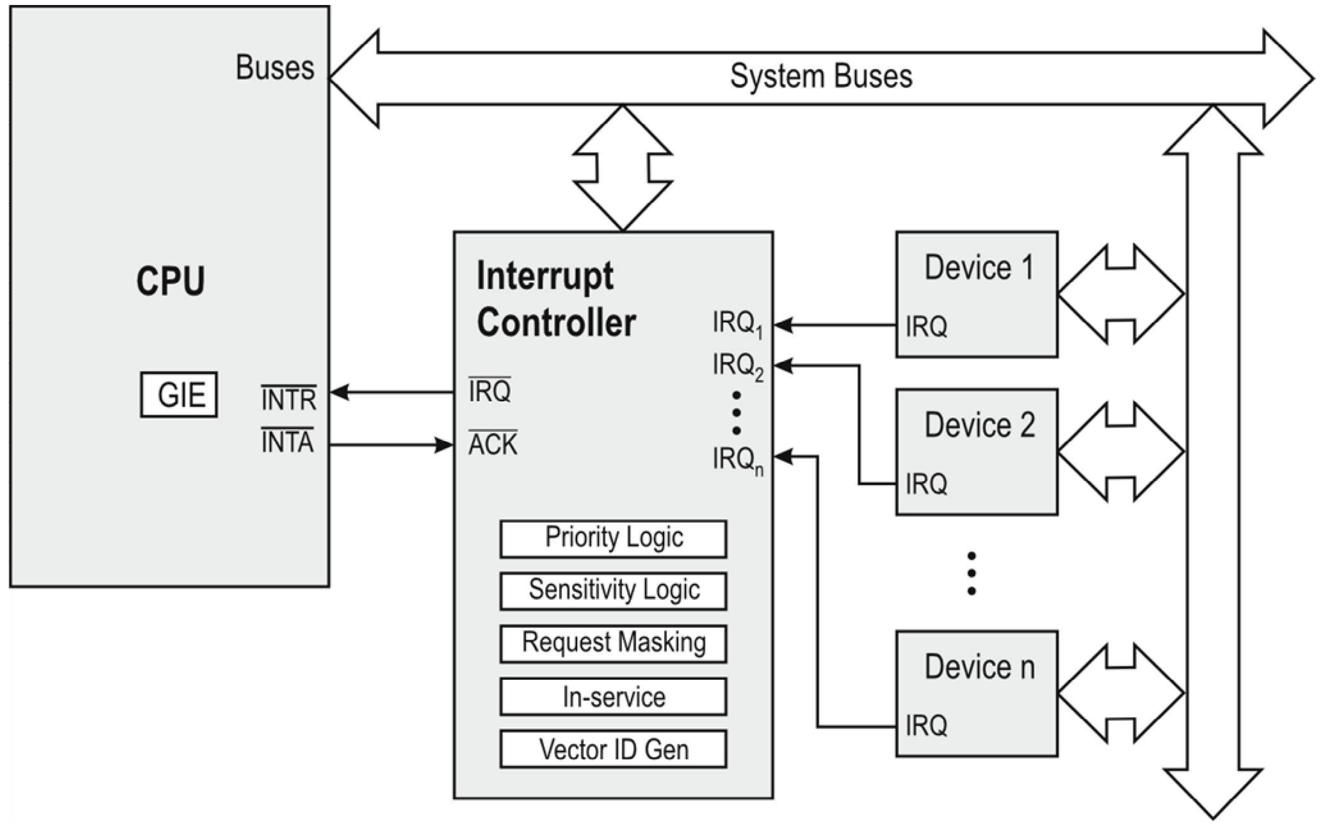


Fig. 7.5 Interrupt management using a programmable interrupt controller

MSP430 Interrupts

- All Internal Peripherals are Interrupt Capable
  - Use an internal daisy-chain scheme
  - Support low-power operation
- Auto-vectored Approach
  - Fixed interrupt table
  - Vector addresses depend on MCU family member
- Types of MSP430 Interrupts
  - System Resets
  - Non-maskable
  - Maskable

SYSTEM RESETS

- ❑ Have the highest system priority
- ❑ Truly non-maskable
- ❑ Triggered by different events (multi-sourced)
  - Power-up,
  - External Reset,
  - Watchdog Timer,
  - FLASH key violation, etc.
- ❑ Vector at address 0FFFEh
- ❑ Saves no PC and no PSW
  - Non returning ISR
  - Calls bootstrap sequence

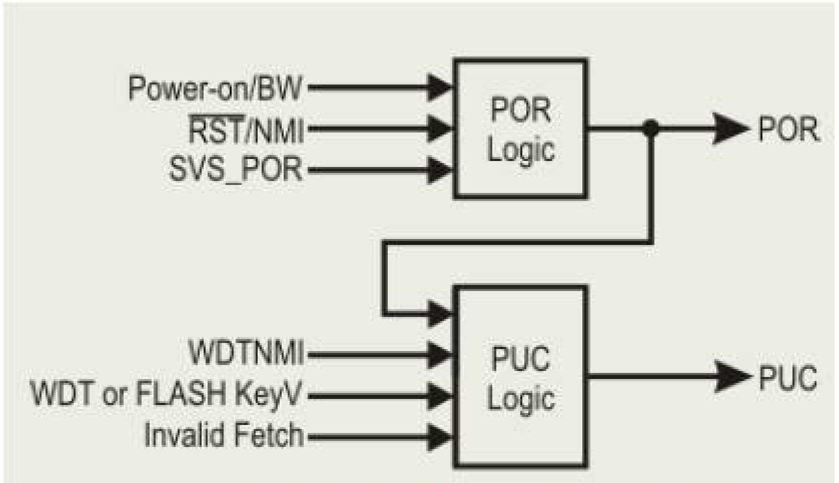


Fig. 6.28 Basic reset module in MSP430 devices

MASKABLE VS. NON-MASKABLE

- Maskable Interrupts
  - Can be blocked through a flag
    - Global Interrupt Flag (GIE)
    - Local Flags in Peripheral Interfaces
  - Most common type of interrupt
  - Disabled upon RESET
  - By default disabled upon entrance into an ISR
- Non-maskable Interrupts (NMI)
  - Cannot be masked, thus are always served
  - Reserved for system critical events

Interrupts

- The code to handle an interrupt is called an *interrupt handler* or *interrupt service routine* (ISR).
- The feature that interrupts arise at unpredictable times means that an ISR must carry out its action and clean up thoroughly so that the main code can be resumed without error—it should not be able to tell that an interrupt has occurred

Interrupts

- Interrupts can be requested by most peripheral modules and some in the core of the MCU, such as the clock generator.
- Each interrupt has a flag, which is raised (set) when the condition for the interrupt occurs. For example, Timer_A sets the **TAIFG flag** in the **TACTL register** when the counter TAR returns to 0. We polled this earlier!

Interrupts

- Most interrupts are *maskable*, which means that they are effective only if the general interrupt enable (GIE) bit is set in the status register (SR).
- They are ignored if GIE is clear.
- Therefore both the enable bit in the module and GIE must be set for interrupts to be generated.
- The (non)maskable interrupts cannot be suppressed by clearing GIE. BUT These interrupts also require bits to be set in special function or peripheral registers to enable them. Thus even the nonmaskable interrupts can be disabled and indeed all are disabled by default.

Interrupts

- The MSP430 uses *vectored* interrupts, which means that the address of each ISR—its vector—is stored in a *vector table* at a defined address in memory
- Each interrupt vector has a distinct priority, which is used to select which vector is taken if more than one interrupt is active when the vector is fetched. The priorities are fixed in hardware and cannot be changed by the user
- They a re given simply by the address of the vector: A higher address means a higher priority. The reset vector has address 0xFFFE, which gives it the top priority, followed by 0xFFFC for the single nonmaskable interrupt vector.

Memory	Size	Address	Description	Access
Flash	32KB	0xFFFF 0xFFC0	Interrupt Vector Table	Word
		0xFFBF 0x8000	Program Code	Word/Byte
SRAM	1KB	0x05FF 0x0200	Stack	Word/Byte
	256	0x01FF 0x0100	16-bit Peripherals Modules	Word
	240	0x00FF 0x0010	8-bit Peripherals Modules	Byte
	16	0x000F 0x0000	8-bit Special Function Registers	Byte

The interrupt vector table is mapped at the very highest end of memory space (upper 16 words of Flash/ROM), in locations 0FFE0h through to 0FFFEh (see the device-specific datasheets).
The priority of the interrupt vector increases with the word address.
The push button, Port 1 or P1.3, the interrupt vector address 0xFFE4 contains the address of the interrupt service routine (ISR) for the push button.
The prototype defining a Port 1 ISR that includes the P1.3 push button:

```
#pragma vector=PORT1_VECTOR
   __interrupt void Port_1(void)
```

Vector Address	MSP430G2553
0xFFE4	`Port_1`

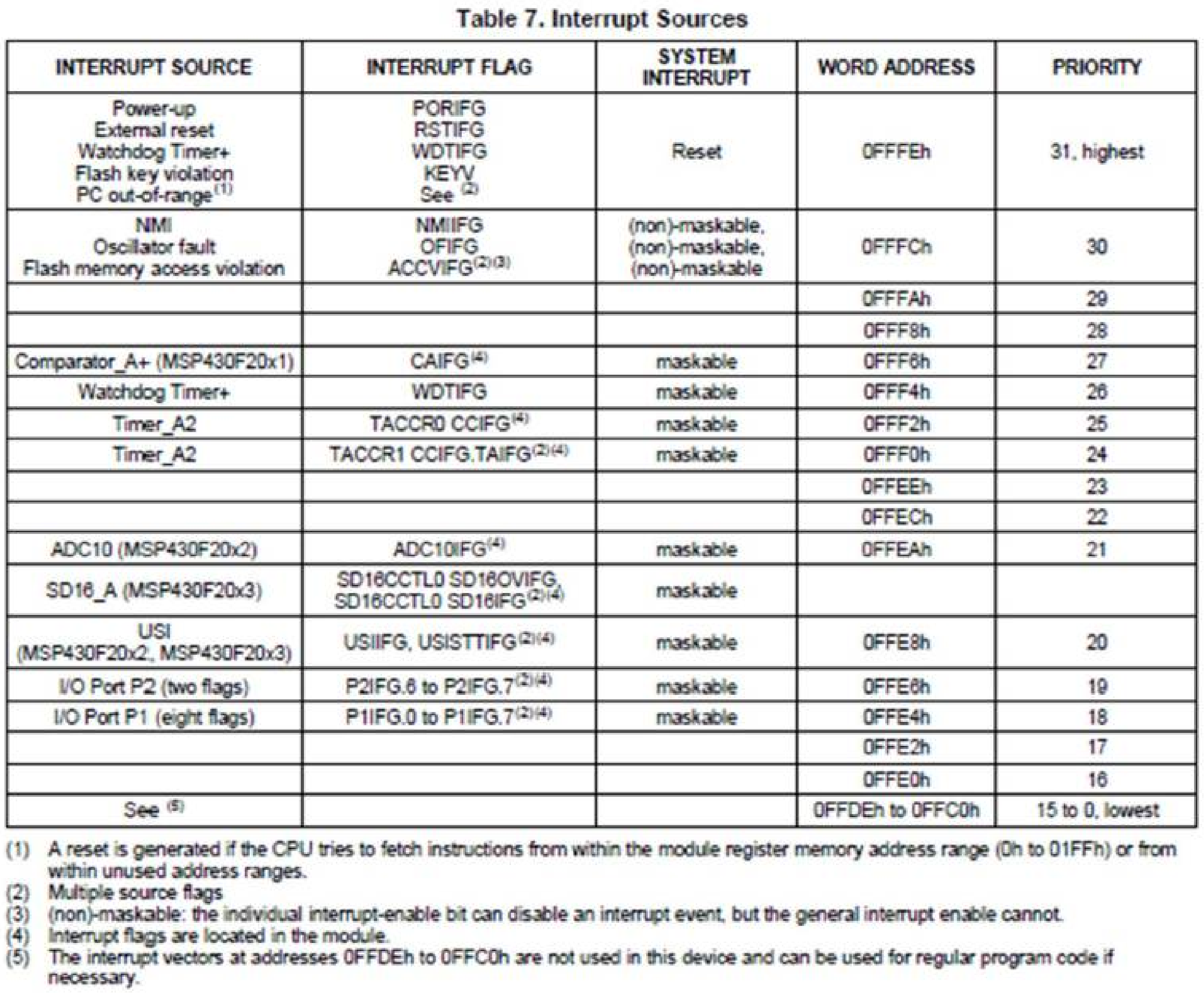


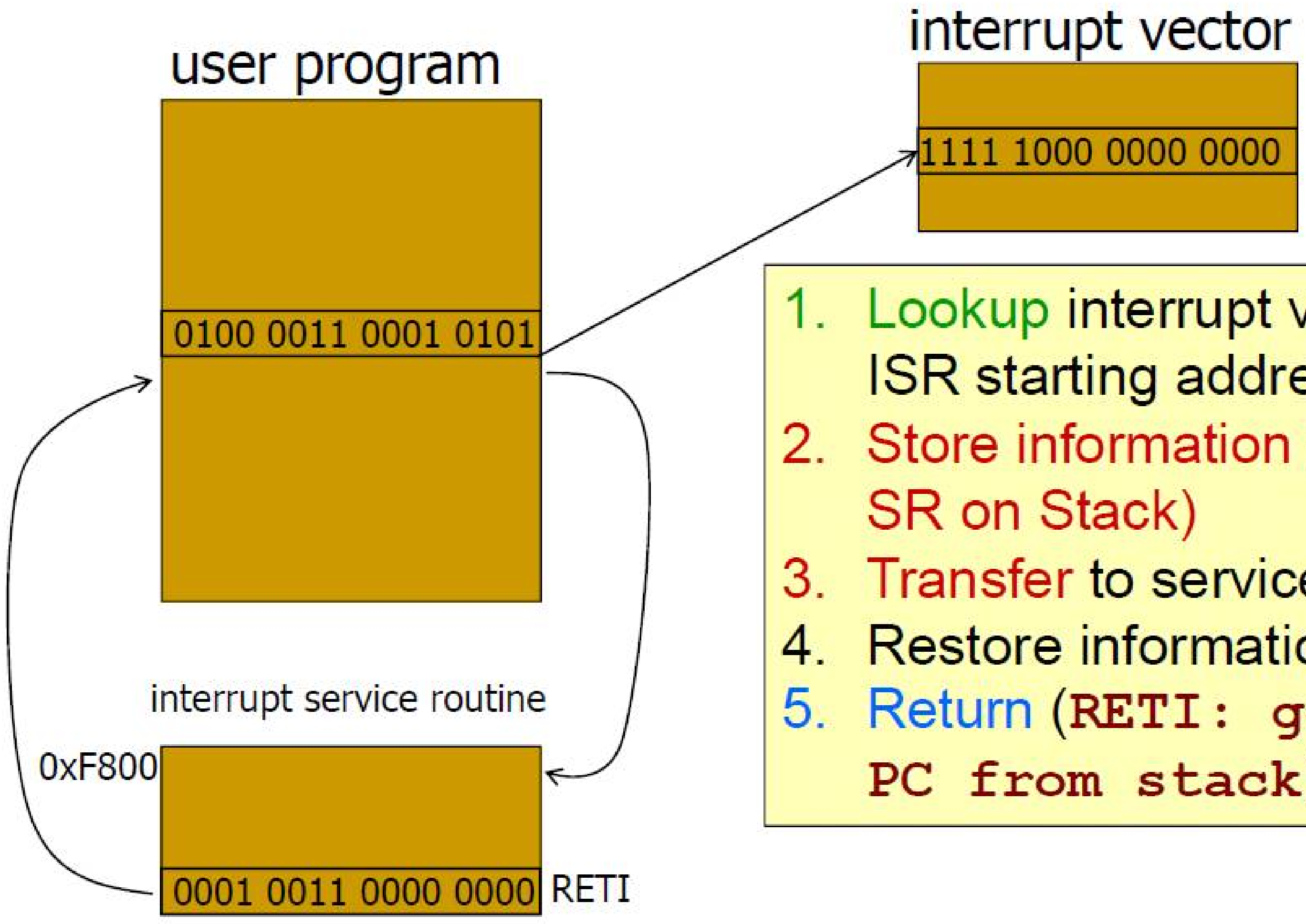
Table 7. Interrupt Sources

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 See (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
Comparator_A+ (MSP430F20x1)	CAIFG (4)	maskable	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 (MSP430F20x2)	ADC10IFG (4)	maskable	0FFEAh	21
SD16_A (MSP430F20x3)	SD16CCTL0 SD16OVIFG, SD16CCTL0 SD16IFG (2)(4)	maskable		
USI (MSP430F20x2, MSP430F20x3)	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 (5)			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) The interrupt vectors at addresses 0FFDEh to 0FFC0h are not used in this device and can be used for regular program code if necessary.

Interrupts

- The vectors for the maskable interrupts depend on the peripherals in a particular device and are listed in a table of *Interrupt Vector Addresses* in the data sheet
- The MSP430 stacks both the return address and the status register. The SR gets this privileged treatment because it controls the low-power modes and the MCU must return to full power while it processes the interrupt
- The other registers must be saved on the stack (or RAM) and restored if their contents are modified in the ISR.



1. Lookup interrupt vector for ISR starting address.
2. Store information (PC and SR on Stack)
3. Transfer to service routine.
4. Restore information
5. Return (**`RETI: get old PC from stack`**).

What Happens when an Interrupt Is Requested?

- A lengthy chain of operations lies between the cause of a maskable interrupt and the start of its ISR.
- It starts when a flag bit is set in the module when the condition for an interrupt occurs. For example, TAIFG is set when the counter TAR returns to 0. This is passed to the logic that controls interrupts if the corresponding enable bit is also set, TAIE in this case. The request for an interrupt is finally passed to the CPU if the GIE bit is set.

What Happens when a Subroutine Is Called?

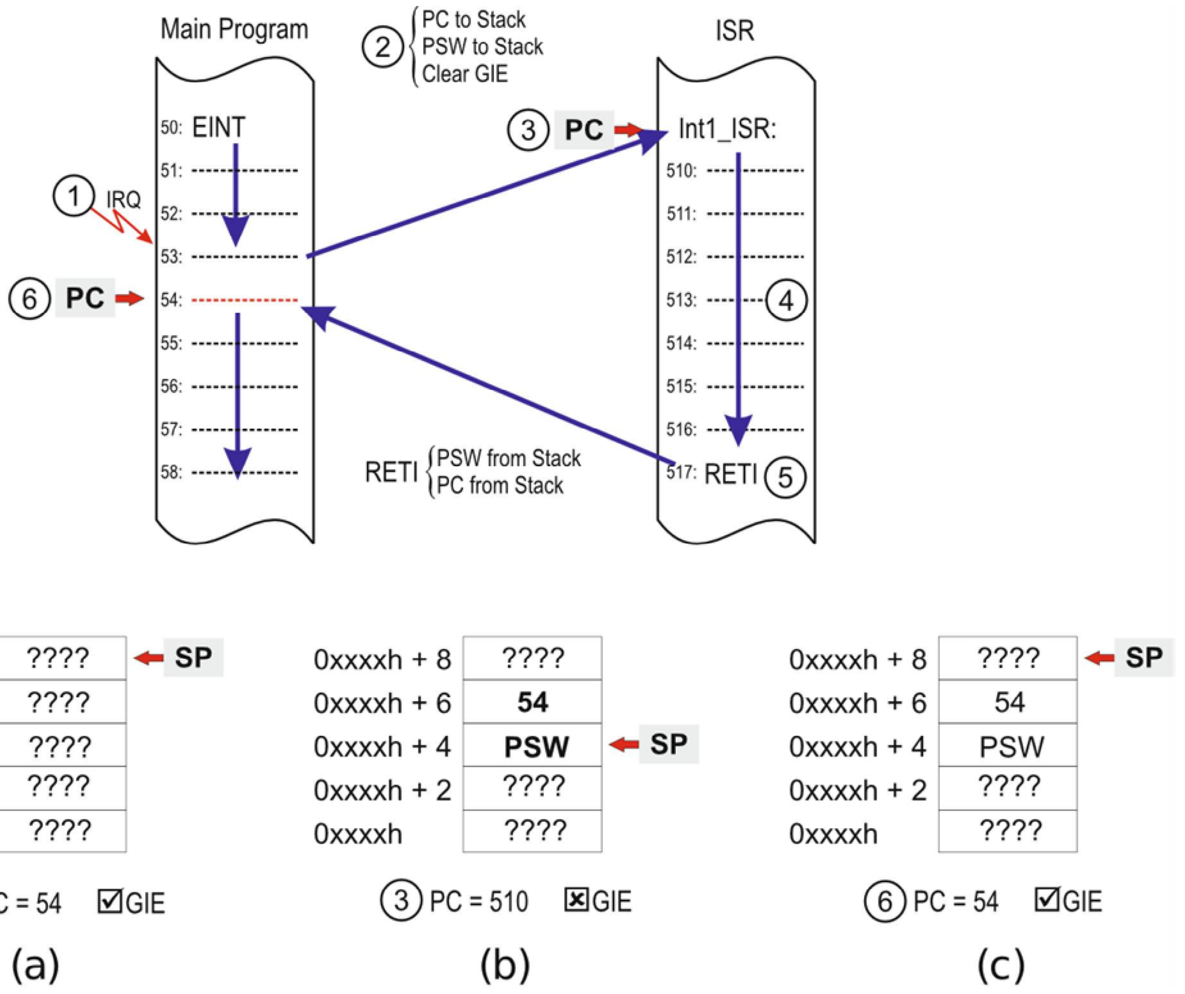


Fig. 7.1 Sequence of events in the CPU upon accepting an interrupt request. **a** Stack before IRQ, **b** Stack after IRQ, **c** Stack after RETI

Hardware then performs the following steps to launch the ISR:

1. Any currently executing instruction is completed if the CPU was active when the interrupt was requested. **MCLK is started if the CPU was off**.
2. The **PC**, which points to the next instruction, is **pushed onto the stack**.
3. The **SR** is **pushed onto the stack**.
4. The **interrupt with the highest priority is selected if multiple interrupts are waiting for service**.

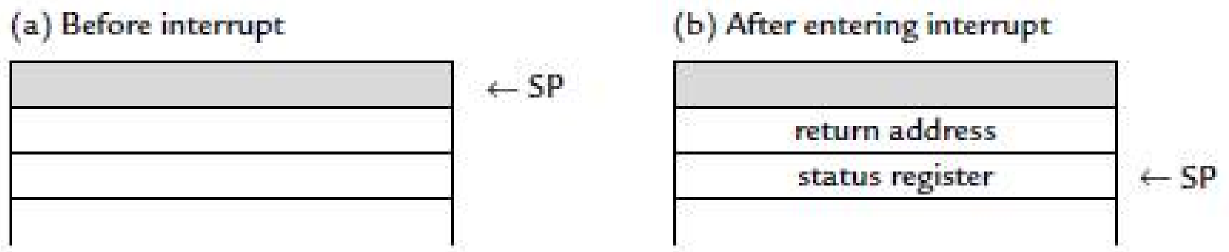


Figure 6.5: Stack before and after entering an interrupt service routine. The return address (PC) and status register (SR) have been saved, with SR on the top of the stack.

Hardware then performs the following steps to launch the ISR:

5. The interrupt request flag is **cleared automatically** for vectors that have a **single source**. **Flags remain set for** servicing by software **if the vector has multiple sources**, which applies to the example of TAIFG.
6. The **SR is cleared**, which has two effects. First, further maskable interrupts are disabled because the GIE bit is cleared; nonmaskable interrupts remain active. Second, it terminates any low-power mode.
7. The **interrupt vector is loaded into the PC** and the CPU starts to execute the interrupt service routine at that address.
8. This sequence takes six clock cycles in the MSP430 before the ISR commences.

ISR

- An ISR looks almost **identical** to a subroutine but with two distinctions:
  - The **address of the subroutine,** for which we can use its name (a label on its first line), must be **stored in the appropriate interrupt vector.**
  - **The routine must end with RETI** rather than RET so that the correct sequence of actions takes place when it returns.

How to return from ISR

- An interrupt service routine must always finish with the special *return from interrupt* instruction RETI, which has the following actions:

  1. The SR pops from the stack. All previous settings of GIE and the mode control bits are now in effect, regardless of the settings used during the interrupt service routine. In particular, this reenables maskable interrupts and restores the previous low-power mode of operation if there was one.
  2. The PC pops from the stack and execution resumes at the point where it was interrupted. Alternatively, the CPU stops and the device reverts to its low-power mode before the interrupt.

Interrupt Programming Requirements

- ❑ 1) Stack Allocation
  - Is where CPU saves the SR and PC
  - Automatically allocated by C-compiler
- ❑ 2) Vector Entry Setup
  - Specify entry in vector table
- ❑ 3) Provide the Actual ISR Code
  - Short and quick
  - Register transparent (in ASM)
  - Do not use parameter passing or value return
  - In ASM, always end with RETI
- ❑ 4) Enable Interrupt Flags
  - Both GIE and local enable

Interrupts in Assembly

```
; Code assumes push-button in P1.3 is hardware debounced and wired to
; produce a high-to-low transition when depressed.
;-----------------------------------------------------------------------------
#include "msp430g2231.h"
;-----------------------------------------------------------------------------
RSEG CSTACK ; Stack declaration <------(1)
RSEG CODE ; Executable code begins
;-----------------------------------------------------------------------------
Init MOV.W #SFE(CSTACK),SP ; Initialize stack pointer <--(1)
MOV.W #WDTPW+WDTHOLD,&WDTCTL ; Stop the WDT
;--------------------------- Port1 Setup --------------------------------
BIS.B #0F7h,&P1DIR ; All P1 pins but P1.3 as output
BIS.B #BIT3,&P1REN ; P1.3 Resistor enabled
BIS.B #BIT3,&P1OUT ; Set P1.3 resistor as pull-up
BIS.B #BIT3,&P1IES ; Edge sensitivity now H->L
BIC.B #BIT3,&P1IFG ; Clears any P1.3 pending IRQ
Port1IE BIS.B #BIT3,&P1IE ; Enable P1.3 interrupt <--(4)
Main BIS.W #CPUOFF+GIE,SR ; CPU off and set GIE <---(4)
NOP ; Debugger breakpoint
;-----------------------------------------------------------------------------
PORT1_ISR ; Begin ISR <--------------(3)
;-----------------------------------------------------------------------------
BIC.C #BIT3,&P1IFG ; Reset P1.3 Interrupt Flag
XOR.B #BIT2,&P1OUT ; Toggle LED in P1.2
RETI ; Return from interrupt <---(3)
;-----------------------------------------------------------------------------
; Reset and Interrupt Vector Allocation
;-----------------------------------------------------------------------------
ORG RESET_VECTOR ; MSP430 Reset Vector
DW Init ;
ORG PORT1_VECTOR ; Port.1 Interrupt Vector
DW PORT1_ISR ; <-------(2)
END
```

Interrupts in C

```
//==================================================================
=#include <msp430g2231.h>
//-----------------------------------------------------------------------
void main(void)
{
WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
P1DIR |= 0xF7; // All P1 pins as out but P1.3
P1REN |= 0x08; // P1.3 Resistor enabled
P1OUT |= 0x08; // P1.3 Resistor as pull-up
P1IES |= 0x08; // P1.3 Hi->Lo edge selected
P1IFG &= 0x08; // P1.3 Clear any pending P1.3 IRQ
P1IE |= 0x08; // P1.3 interrupt enabled
//
_ _bis_SR_register(LPM4_bits + GIE); // Enter LPM4 w/GIE enabled <---(4)
}
//
//-----------------------------------------------------------------------
// Port 1 interrupt service routine
#pragma vector = PORT1_VECTOR // Port 1 vector configured <---(2)
_ _interrupt void Port_1(void) // The ISR code <----(3)
{
P1OUT ˆ= 0x04; // Toggle LED in P1.2
P1IFG &= ˜0x08; // Clear P1.3 IFG
}
//========================================
```

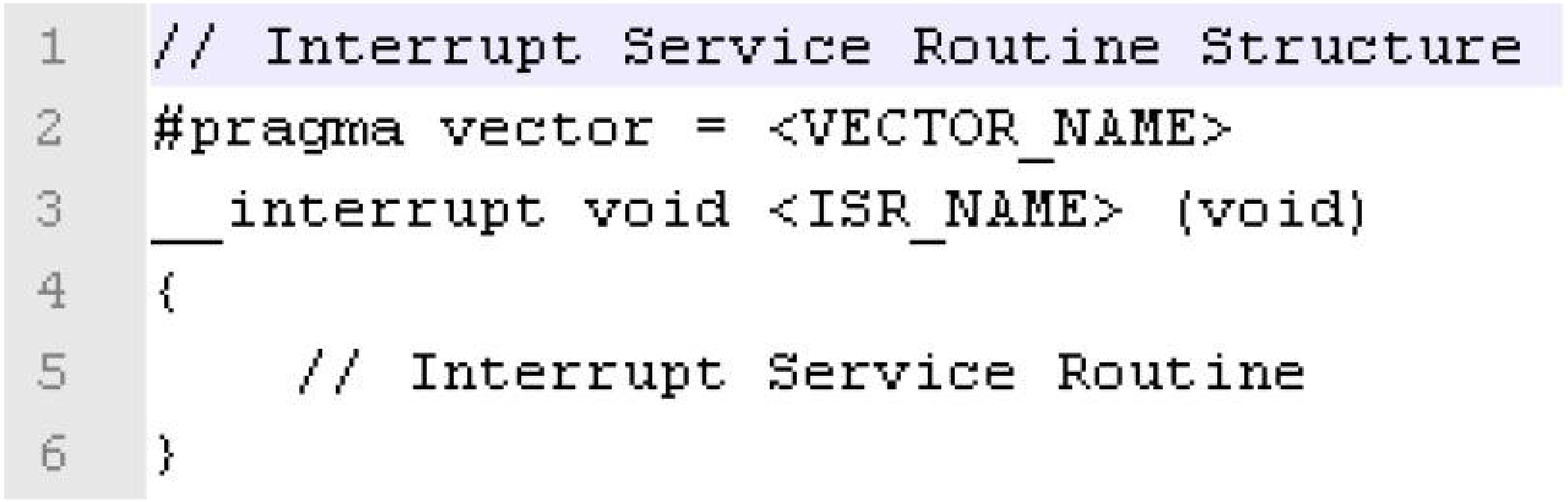
Interrupt Service Routines in C

- ❑ An interrupt service routine cannot be written entirely in standard C because there is no way to identify it as an ISR rather than an ordinary function. In fact it would appear that the function was dead code, meaning that it could never be called, so the compiler would optimize it away. Some extensions are therefore needed and these inevitably differ between compilers.

Interrupt Service Routines in C

- First is the #pragma line, which associates the function with a particular interrupt vector.
- The second is the _ _interrupt keyword at the beginning of the line that names the function. This ensures that the address of the function is stored in the vector and that the function ends with reti rather than ret. Again there is no significance to the name of the function; it is the name of the vector that matters.

- ❑ An intrinsic function is needed to set the GIE bit and turn on interrupts. This is called _ _enable_interrupt.
- ❑ It is declared in intrinsics.h, which must be included.



```
// Interrupt Service Routine Structure
#pragma vector = <VECTOR_NAME>
__interrupt void <ISR_NAME> (void)
{
    // Interrupt Service Routine
}
```

Flow for Multisource Interrupt

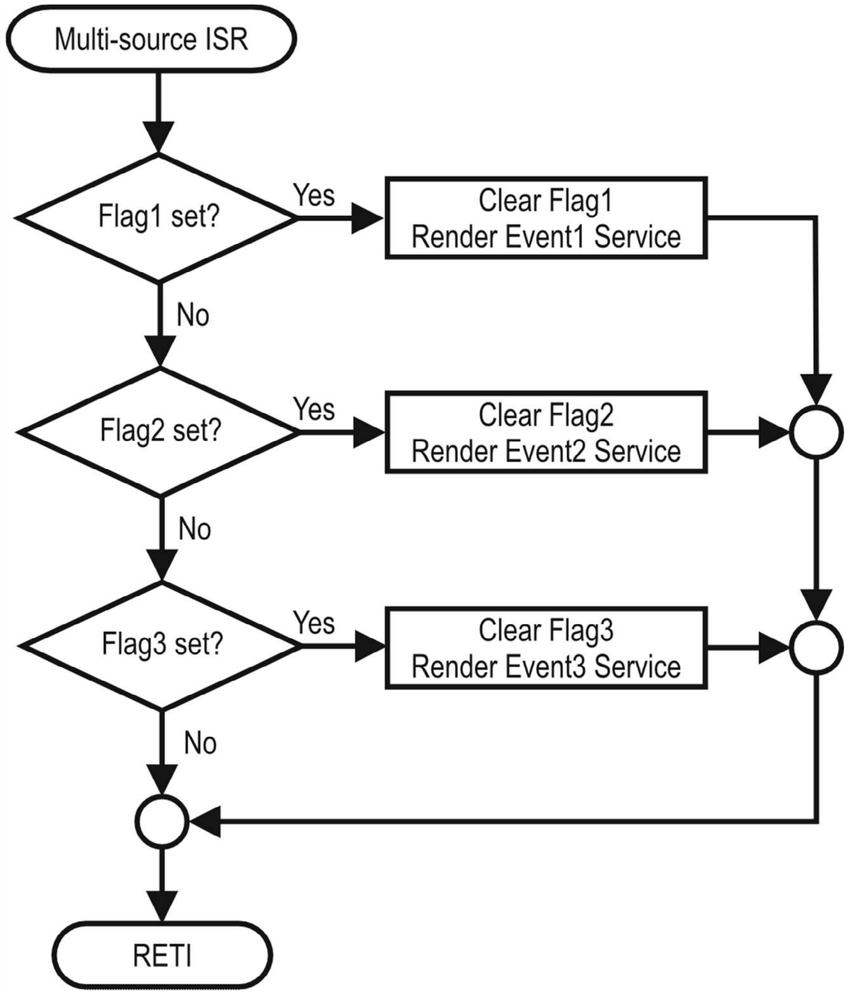


Fig. 7.7 Flowchart of a service routine for a multi-source interrupt vector

MULTI-SOURCE INTERRUPT

- Multiple Events Served by a Single Vector
  - Single ISR for all events
  - Specific event flags need explicit clear
  - ISR code must identify actual trigger source
  - Source Identification
    - Polling ALL interrupt request flags
    - Via calculated branching

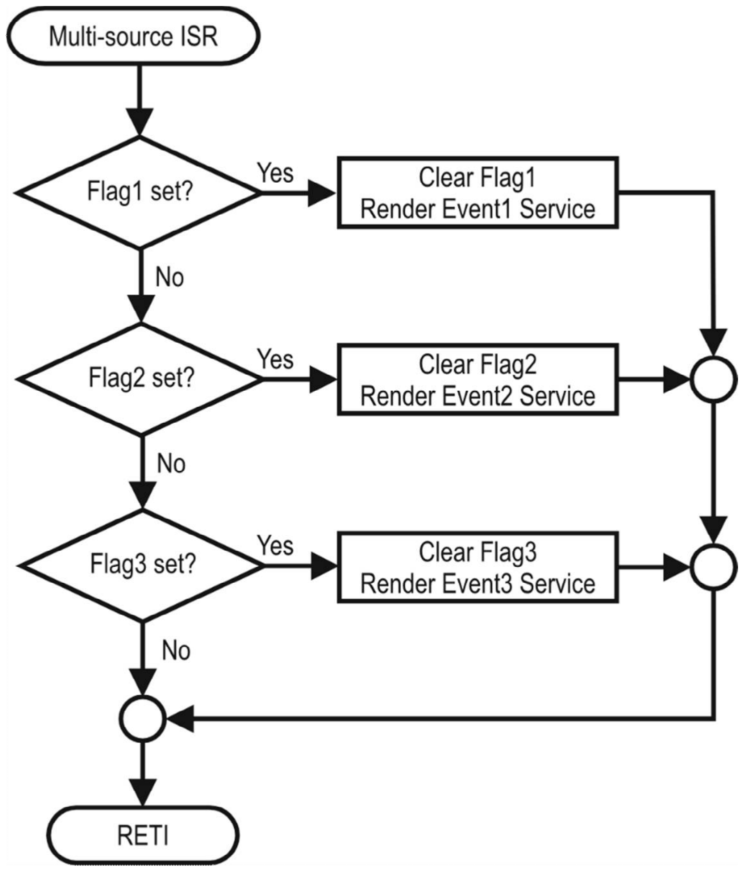


Fig. 7.7 Flowchart of a service routine for a multi-source interrupt vector

Using Calculated Branching

```
;==========================================================================
; Pseudo code for Multi source Interrupt Service Routine for four events
; Assumes FlagReg contains prioritized, encoded IRQ flags organized as:
; 0000 - No IRQ     0004 - Event2     0008 - Event4
; 0002 - Event 1    0006 - Event3
;--------------------------------------------------------------------------
#include "headers.h"           ; Header file for target MCU
...                            ; Preliminary declarations and code
MultiSrcISR
           ADD &FlagReq,PC     ; Adds Interrupt Flag Register contents to PC
           JMP Exit            ; Flags = 0: No interrupt
           JMP Event1          ; Flags = 2: Event1
           JMP Event2          ; Flags = 4: Event2
           JMP Event3          ; Flags = 6: Event3
Event4     Task 4 starts here ; Flags = 8: Event4
           ...
           JMP Exit            ; Exit ISR

Event1     Task 1 starts here ; Vector 2
           ...                 ; Task 1 starts here
           JMP Exit            ; Exit ISR

Event2     Task 2 starts here ; Vector 4
           ...                 ; Task 2 starts here
           JMP Exit            ; Exit ISR

Event2     Task 3 starts here ; Vector 4
           ...                 ; Task 3 starts here
Exit       RETI; Return
```

- Calculation Methods
  - Look-up table-based
  - Via encoded flags (shown)

Port 1 and 2 interrupts

- Only transitions (low to hi or hi to low) cause interrupts
- P1IFG & P2IFG (Port 1 & 2 Interrupt FlaG registers)
  - Bit 0: no interrupt pending
  - Bit 1: interrupt pending
- P1IES & P2IES (Port 1 & 2 Interrupt Edge Select reg)
  - Bit 0: PxIFG is set on low to high transition
  - Bit 1: PxIFG is set on high to low transition
- P1IE & P2IE (Port 1 & 2 Interrupt Enable reg)
  - Bit 0: interrupt disabled
  - Bit 1: interrupt enabled

Programming the MSP430 for Push Button Interrupt

Each device must be initialized to generate an interrupt and the
CPU must be enabled to respond to a maskable interrupt.
1. A mask setting determines which devices (e.g. push button) can generate an interrupt.
Interrupt mask, bit 3 (0x08) of P1.3 is the push button
; 1 enables/0 disables. Must be 1 for the device to trigger an interrupt.

```
P1IE |= 0x08;           // Enables P1.3 to generate interrupt
```

2. Enable the push button P1.3

```
P1SEL &= ~0x08;    // Select Port 1 P1.3 (push button)
P1DIR &= ~0x08;    // Port 1 P1.3 (push button) as input, 0 is input
P1REN |= 0x08;     // Enable Port P1.3 (push button) pull-up resistor
```

Programming the MSP430 for Push Button Interrupt

3. Enable CPU interrupt response.

```
_BIS_SR(GIE);       // Enable interrupts
```

4. Interrupt service routine (ISR).

The push button is part of port 1, a PORT1_VECTOR interrupt handler is executed when a port 1 interrupt handler occurs.

`__interrupt` defines *Port_1* procedure to be an ISR.

```
#pragma vector=PORT1_VECTOR
      __interrupt void Port_1(void)
```

When interrupt occurs, the corresponding flag (bit) is set to 1.

An interrupt handler should normally disable the interrupt by setting flag to 0, allowing another interrupt to occur.

`P1IFG &= ~0x08;` Clears the push button interrupt.

Programming the MSP430 for Push Button Interrupt

Push button Interrupt
When the push button is pressed *down,* a counter is set to 0.
Press `RESET` button to clear any lingering effects of previous programs.
Set a breakpoint at: `i = 0;`

Programming the MSP430 for Push Button Interrupt

```
#include <msp430g2553.h>
long i=0;
void main(void)
{
WDTCTL = WDTPW + WDTHOLD;
// Stop watchdog timer P1SEL &= ~0x08;
// Select Port 1 P1.3 (push button)
P1DIR &= ~0x08;
// Port 1 P1.3 (push button) as input, 0 is input
P1REN |= 0x08;
// Enable Port P1.3 (push button) pull-up resistor
P1IE |= 0x08;
// Port 1 Interrupt Enable P1.3 (push button)
P1IFG &= ~0x08;
// Clear interrupt flag
_BIS_SR(GIE);
// Enable interrupts
while(1)
i++;
// Execute some useful computation
}
```

Port 1 interrupt service routine

```
// Port 1 interrupt service routine
#pragma vector=PORT1_VECTOR
  __interrupt void Port_1(void)
     {
     P1IFG &= ~0x08;
     // P1.3 Interrupt Flag cleared
     i = 0;
     // Reset count
}
```

Watchdog Timer Interrupt

- The Watchdog Timer (WDT) is typically used to trigger a system reset after a certain amount of time. In most examples, the timer is stopped during the first line of code.
- The WDT counts down from a specified value and either resets or interrupts when count overflow is reached. A way to use this timer is to periodically service it by resetting its counter so that the system “knows” that everything is all right and there is no reset required. In this case, this module is configured as an interval timer to generate interrupts at selected time intervals.
- A computer watchdog is a hardware timer used to trigger a system reset if software neglects to regularly service the watchdog (often referred to as "petting", "kicking", or "feeding" the dog). In a watchdog mode, the watchdog timer can be used to protect the system against software failure, such as when a program becomes trapped in an unintended, infinite loop.

Watchdog Timer Interrupt

- The watchdog can also be configured as an interval timer instead of a timeout device by setting the WDTTMSEL bit in the watchdog control register (WDTCTL). When the timer reaches its limit in timer mode, the counter restarts from 0.
- As an interval timer, the watchdog has its own interrupt vector (which is not the same as the reset vector). An interrupt is generated only if the WDTIE bit in the special function register IE1 and the GIE bit in the status register are set. The WDTIFG flag is automatically cleared when the interrupt is serviced. The watchdog can be polled if interrupts are not used (we did this!).
- Applications needing a periodic "tick" may find the watchdog interval mode ideal for generating a regular interrupt. The disadvantage is the limited selection of interval times - only 4 intervals are available and they are dependent upon the clock assigned to the watchdog.

Watchdog Timer Interrupt

Assuming the MSP430 system clock has a frequency of 1Mhz and the watchdog timer is clocked by the sub-master clock (SMCLK), the following intervals are available as an example
WDTCTL = WDT_MDLY_0_5
Sets the WDT to a 0.5ms interval or 2000 intervals/second.

Constant	Interval	Clocks/Interval (@1Mhz)	Intervals/Second (@1Mhz)
WDT_MDLY_32	32ms (default)	32000	1000000/32000 = 31.25
WDT_MDLY_8	8ms	8000	1000000/8000 = 125
WDT_MDLY_0_5	0.5ms	500	1000000/500 = 2000
WDT_MDLY_0_064	0.064ms	64	1000000/64 = 15625

WDT Int Example

- sets the WDT to an time interval of approximately 32 ms.
- enables WDT interrupts
- enables the red LED P1.0 for output
- enables interrupts.
- when a WDT interrupt, executes procedure __interrupt watchdog_timer(void)
- if 10*32 ms. passed, toggle red LED and reset counter, otherwise increment counter.

WDT Int Example

```
#include <msp430g2553.h>
unsigned int counter = 0;
void main(void){
   WDTCTL = WDT_MDLY_32;        // Watchdog Timer interval to ≈32ms
   IE1 |= WDTIE;                // Enable WDT interrupt
   P1DIR |= BIT0;               // Set red LED P1.0 to output direction
   _BIS_SR(GIE);                // Enable interrupts
   while(1);
}
#pragma vector=WDT_VECTOR       // Watchdog Timer interrupt service routine
 __interrupt void watchdog_timer(void)
           if(counter == 10){          // 10 * 32 ms = 320 ms, ≈.3 s
           P1OUT ^= BIT0;           // P1.0 toggle, red LED
           counter = 0;
           }
           else
           counter++;
}
```

Clock Interrupt

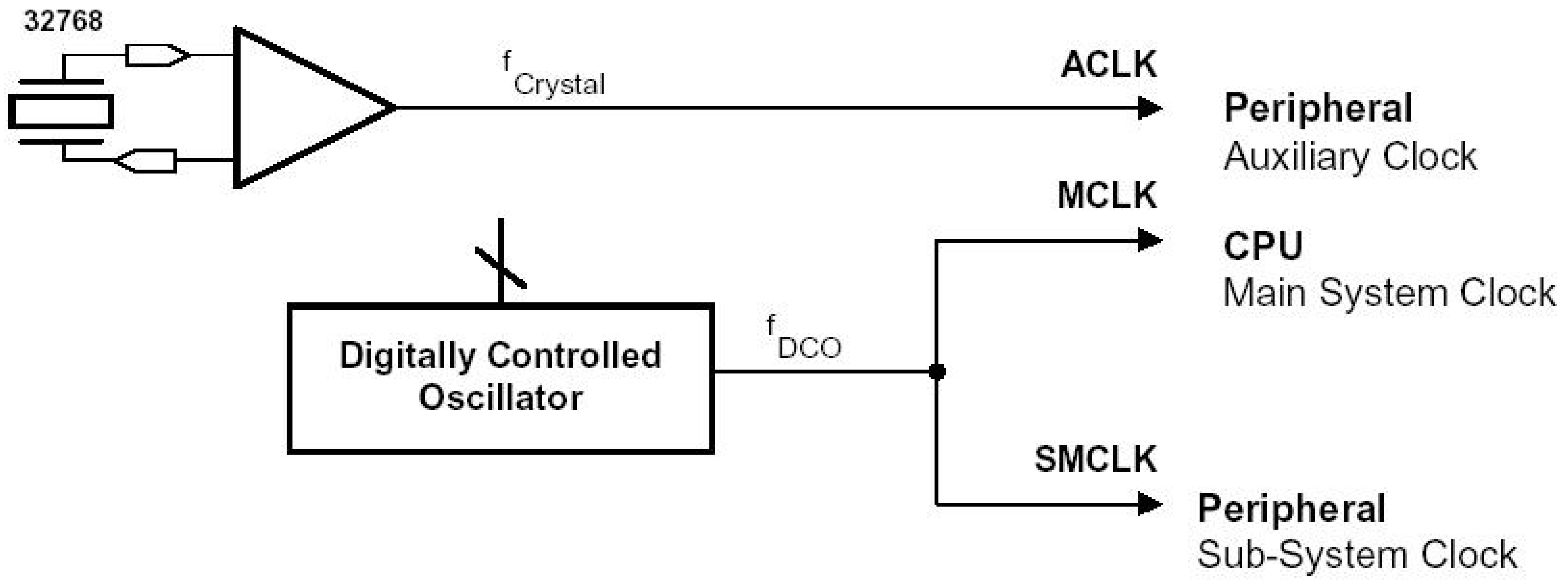
MSP430 CPU and other system devices use three internal clocks:
1.Master clock, MCLK, is used by the CPU and a few peripherals.

2.Subsystem master clock, SMCLK, is distributed to peripherals.

3.Auxiliary clock, ACLK, is also distributed to peripherals.

Typically SMCLK runs at the same frequency as MCLK, both in the megahertz range.
ACLK is often derived from a watch crystal and therefore runs at a much lower frequency. Most peripherals can select their clock from either SMCLK or ACLK.
For the MSP430 processor, both the MCLK and SMCLK clocks are supplied by an internal digitally controlled oscillator (DCO), which runs at about 1.1 MHz.

Clock Source review



DCO - Digitally Controlled Oscillator internal runs at about 1Mhz.
The frequency of the DCO is controlled through sets of bits in the module's registers at three levels. There calibrated frequencies of 1, 8, 12, and 16 MHz.
To change frequency simply copy values into the clock module registers.

Timers/Counters

- Timers are fundamentally counters driven by a clock signal, commonly incrementing or decrementing the counter on each clock tick.

- When the counter reaches some predefined value (e.g. 0), the timer can generate an interrupt. The result is a reasonably accurate time base for executing functionality such as maintaining a reference clock (seconds, minutes, etc.) or performing some operation on a regular basis (blink a LED every second).

- The count of a 16-bit timer in continuous mode below repeats counting from 0 to FFFFh. Using UP mode, other limits can be specified.

Figure 12–4. Continuous Mode

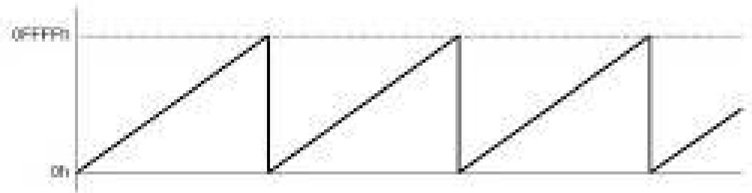


Figure 12–2. Up Mode

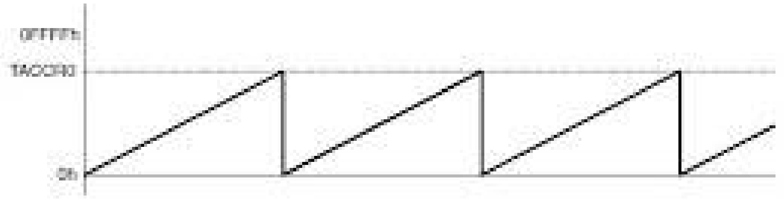
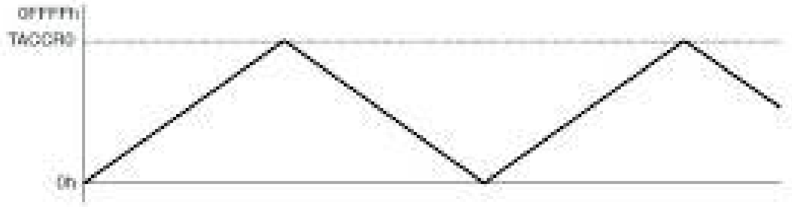
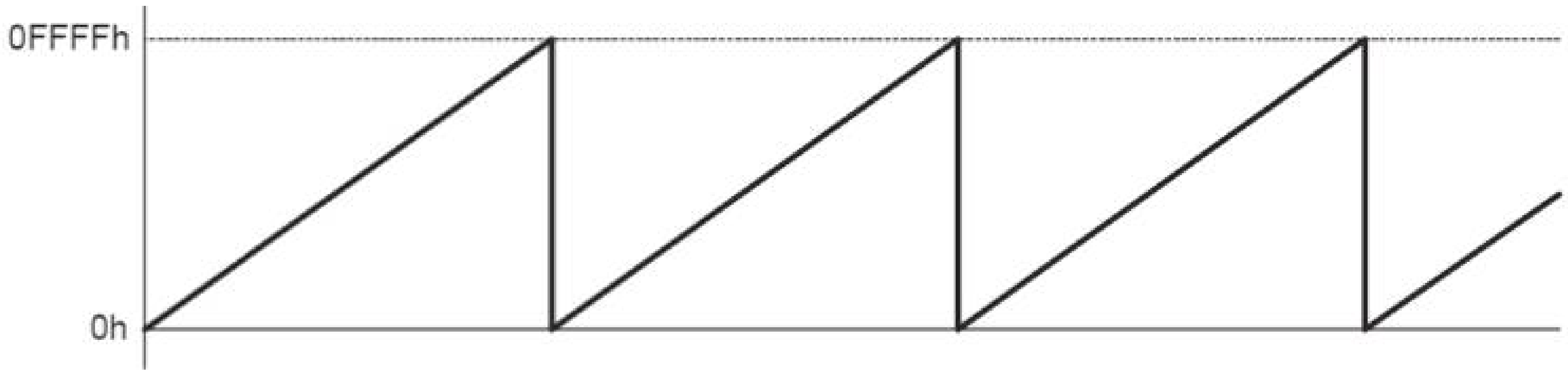


Figure 12–7. Up/Down Mode



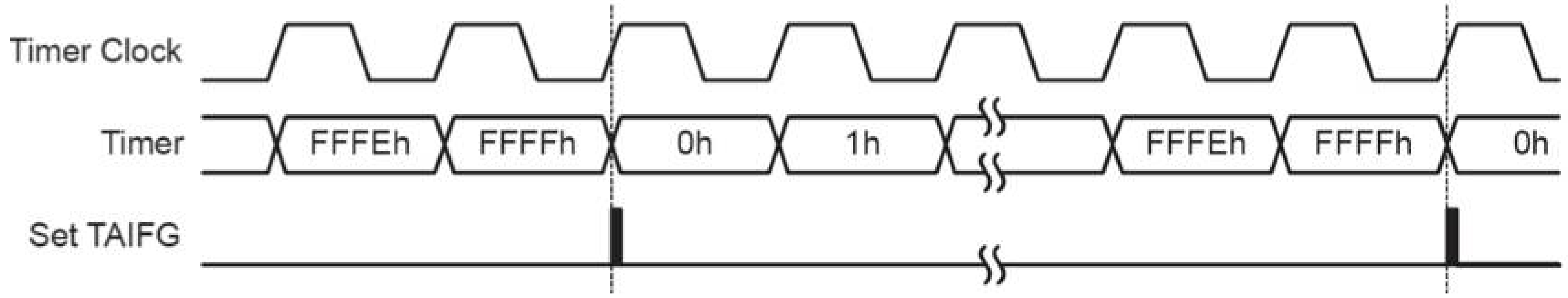
Timers/Counters



The clock tick drives the counter, with each clock tick incrementing the counter.

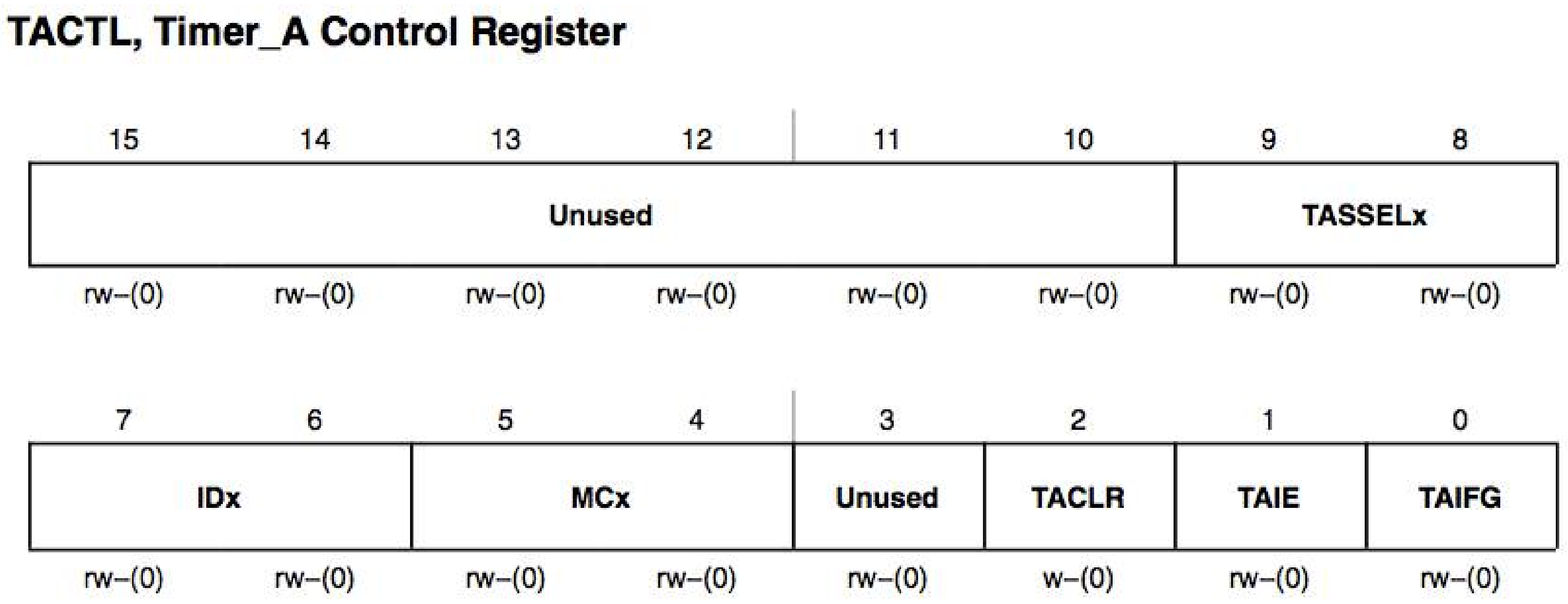
MSP430 timers can interrupt (TAIFG) when the timer reaches a specified limit, serving to perform accurately timed operations.

For example, a timer clock of FFFFh Hz and a count limit of FFFFh could generate an interrupt every second by setting the TAIFG.


Timer Clock
Timer
FFFEh
FFFFh
0h
1h
FFFEh
FFFFh
0h
Set TAIFG

Timer

TA0CTL Timer A0 control register.



TACTL, Timer_A Control Register

15	14	13	12	11	10	9	8
Unused						TASSELx	
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)

7	6	5	4	3	2	1	0
IDx		MCx		Unused	TACLR	TAIE	TAIFG
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	w–(0)	rw–(0)	rw–(0)

TAIE and TAIFG (bits 0 and 1) control the ability of the timer to trigger interrupts

TACCTLx -- The Timer_A Capture/Compare Control Registers correspond to the TACCRx registers. These set the behavior of how the CCR's are used. CCIE and CCIFG (bits 4 and 0) are interrupts associated with the CCR's.

TACCRx Holds the 16-bit count value. TACCRx -- The Timer_A Capture/Compare Registers, of which there are two (TACCR0 and TACCR1) are where specific values to use are stored.

In compare mode, the timer signals an event on the values. Particularly, TACCR0 stores the value that Timer_A counts in up and up/down mode. In capture mode, the processor will record the value of TAR when the MSP430 is signaled to do so.

One important point is that timers can run independently to the CPU clock, allowing the CPU to be turned off and then automatically turned on when an interrupt occurs.

- **TAIV -- The Timer_A Interrupt Vector Register; since there are multiple types of interrupts that can be flagged by Timer_A, this register holds details on what interrupts have been flagged.**
  - The only bits used here are bits 1-3 (TAIVx), which show the type of interrupt that has happened, allowing us to take different actions to resolve the different types of interrupts.

```
 #include <msp430g2553.h>
void main(void) {
  WDTCTL = WDTPW + WDTHOLD;        // Stop watchdog timer

  P1DIR |= BIT0;                  // Set P1.0 to output direction
  P1OUT &= ~BIT0;                   // Set the red LED off

  P1DIR |= BIT6;                  // Set P1.6 to output direction
  P1OUT &= ~BIT6;                   // Set the green LED off

  TA0CCR0 = 12000;                  // Count limit (16 bit)
  TA0CCTL0 = 0x10;                 // Enable Timer A0 interrupts, bit 4=1
  TA0CTL = TASSEL_1 + MC_1;        // Timer A0 with ACLK, count UP

  TA1CCR0 = 24000;                  // Count limit (16 bit)
  TA1CCTL0 = 0x10;                 // Enable Timer A1 interrupts, bit 4=1
  TA1CTL = TASSEL_1 + MC_1;        // Timer A1 with ACLK, count UP

  //_BIS_SR(LPM0_bits + GIE);        // LPM0 (low power mode) interrupts enabled
  _BIS_SR(GIE);        // interrupts enabled
  while (1) {};
}
```

```
#pragma vector=TIMER1_A0_VECTOR    // Timer1 A0 interrupt service
routine

  __interrupt void Timer1_A0 (void) {

   P1OUT ^= BIT0;                    // Toggle red LED
}

#pragma vector=TIMER0_A0_VECTOR    // Timer0 A0 interrupt service
routine

  __interrupt void Timer0_A0 (void) {

   P1OUT ^= BIT6;                    // Toggle green LED
}
```

A Timer Example in C

```
// timintC1.c - toggles LEDs with period of about 1.0s
// Toggle LEDs in ISR using interrupts from timer A CCR0
// in Up mode with period of about 0.5s
// Timer clock is SMCLK divided by 8, up mode , period 50000
// Olimex 1121STK , LED1 ,2 active low on P2.3,4
// J H Davies , 2006 -10 -11; IAR Kickstart version 3.41A
// ----------------------------------------------------------------------
#include <io430x11x1.h> // Specific device
#include <intrinsics.h> // Intrinsic functions
// ----------------------------------------------------------------------
// Pins for LEDs
#define LED1 BIT3
#define LED2 BIT4
// ----------------------------------------------------------------------
void main (void)
{
      WDTCTL = WDTPW|WDTHOLD; // Stop watchdog timer
      P2OUT = ~LED1; // Preload LED1 on , LED2 off
      P2DIR = LED1|LED2; // Set pins with LED1 ,2 to output
      TACCR0 = 49999; // Upper limit of count for TAR
      TACCTL0 = CCIE; // Enable interrupts on Compare 0
      TACTL = MC_1|ID_3|TASSEL_2|TACLR; // Set up and start
      Timer A
      // "Up to CCR0" mode , divide clock by 8, clock from SMCLK ,
      clear timer
      __enable _interrupt (); // Enable interrupts (intrinsic)
      for (;;) { // Loop forever doing nothing
      } // Interrupts do the work
}
// ----------------------------------------------------------------------
// Interrupt service routine for Timer A channel 0
```

```
// ----------------------------------------------------------------------
#pragma vector = TIMERA0_VECTOR
__interrupt void TA0_ISR (void)
{
                P2OUT ^= LED1|LED2;
                // Toggle LEDs
}
```

Assembly version

```
; timrint1.s43 - toggles LEDs with period of about 1s
; TACCR0 interrupts from timer A with period of about 0.5s
; Timer clock is SMCLK divided by 8, up mode , period 50000
; Olimex 1121STK , LED1 ,2 active low on P2.3,4
;-----------------------------------------------------------------------
#include <msp430x11x1.h> ; Header file for this device
;-----------------------------------------------------------------------
; Pins for LED on port 2
LED1 EQU BIT3
LED2 EQU BIT4
;-----------------------------------------------------------------------
RSEG CSTACK ; Create stack (in RAM)
;-----------------------------------------------------------------------
RSEG CODE ; Program goes in code memory
Reset: ; Execution starts here
mov.w #SFE(CSTACK),SP ; Initialize stack pointer
main: ; Equivalent to start of main() in C
mov.w #WDTPW|WDTHOLD ,& WDTCTL ; Stop watchdog timer
mov.b #LED2 ,& P2OUT ; Preload LED1 on , LED2 off
bis.b #LED1|LED2 ,& P2DIR ; Set pins with LED1 ,2 to output
mov.w #49999 ,& TACCR0 ; Period for up mode
mov.w #CCIE ,& TACCTL0 ; Enable interrupts on Compare 0
mov.w #MC_1|ID_3|TASSEL_2|TACLR ,& TACTL ; Set up Timer A
; Up mode , divide clock by 8, clock from SMCLK , clear TAR
bis.w #GIE ,SR ; Enable interrupts (just TACCR0)
jmp $ ; Loop forever; interrupts do all
```

```
;-----------------------------------------------------------------------
; Interrupt service routine for TACCR0 ,
; called when TAR = TACCR0
; No need to acknowledge interrupt explicitly
;- done automatically
TA0_ISR: ; ISR for TACCR0 CCIFG
xor.b #LED1|LED2 ,& P2OUT ; Toggle LEDs

reti ; That's all: return from interrupt
;-----------------------------------------------------------------------
COMMON INTVEC ; Segment for vectors (in Flash)
ORG TIMERA0_VECTOR
DW TA0_ISR ; ISR for TA0 interrupt
ORG RESET_VECTOR
DW Reset ; Address to start execution
```

DEALING WITH FALSE TRIGGERS

- Interrupt Signals Triggered by Unwanted Events
  - Cause undesirable effects
- Causes of False Interrupt Triggers
  - Power glitches
  - Electromagnetic Interference (EMI)
  - Electrostatic Discharges
  - Other noise manifestations
- Mitigation Mechanisms
  - Provide dummy ISR for unused sources
    - Write a “False ISR Handler”
  - Use Watchdog Timers
  - Consider the use of polling

INTERRUPT LATENCY HANDLING

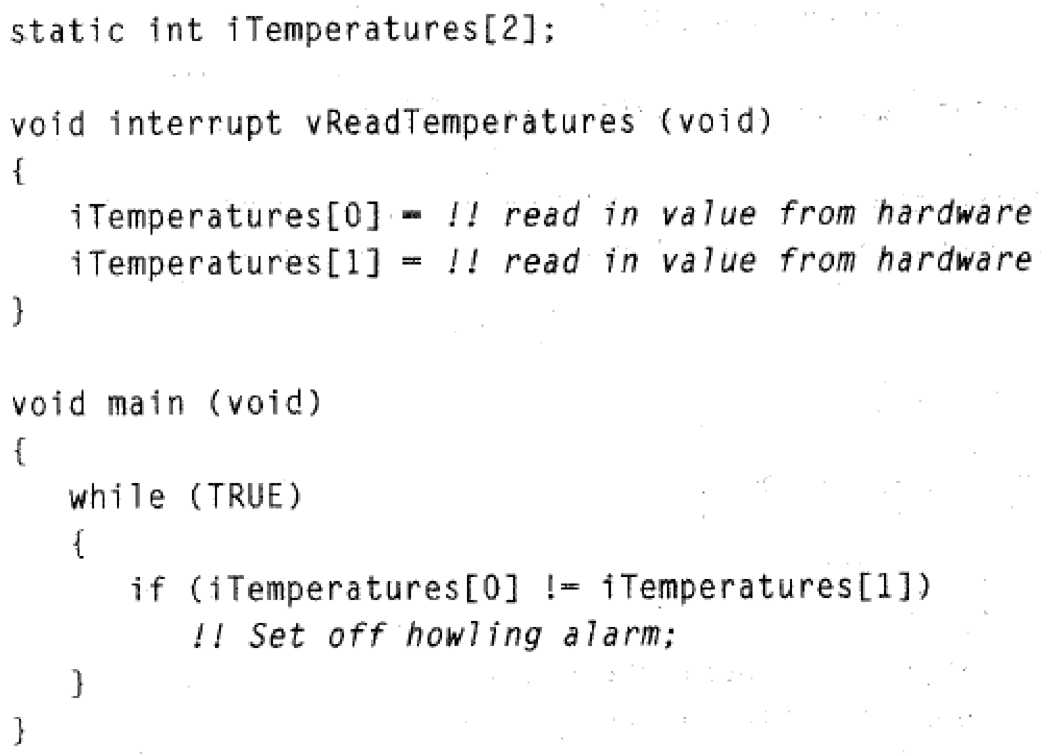
- Interrupt Latency
  - Amount of time from IRQ to fetch of first ISR instruction
  - Negligible in most applications
    - An issue in fast, time sensitive real-time systems
  - Affected by Hardware & Software factors
- Hardware Factors
  - Propagation delays in IRQ and acknowledgment paths
  - Source identification method
- Software Factors
  - Scheduling & Priority Scheme
  - ISR coding style
- Recommendations
  - Minimize number of stages in IRQ & ACK path (if possible)
  - Use vectored schemes with in-service hardware tracking
  - Keep ISRs short & quick
  - Avoid service monopolization (prioritization Vs. nesting)

INTERRUPT NESTING

- Achieved by re-enabling the GIE within an ISR
- Use only when strictly necessary
  - Most applications don’t need it
- Recommendations
  - Establish a strict prioritization
  - Exert SW stack depth control
  - Whenever possible, avoid re-entrancy
  - Avoid static variables – self-modification risk
  - Do not use self-modifying code

The Shared-Data Problem

- In many cases the ISRs need to communicate with the task codes through shared variables.
- Example:
  - Task code monitors 2 temperatures and alarm if they differ
  - An ISR reads temperatures from hardware



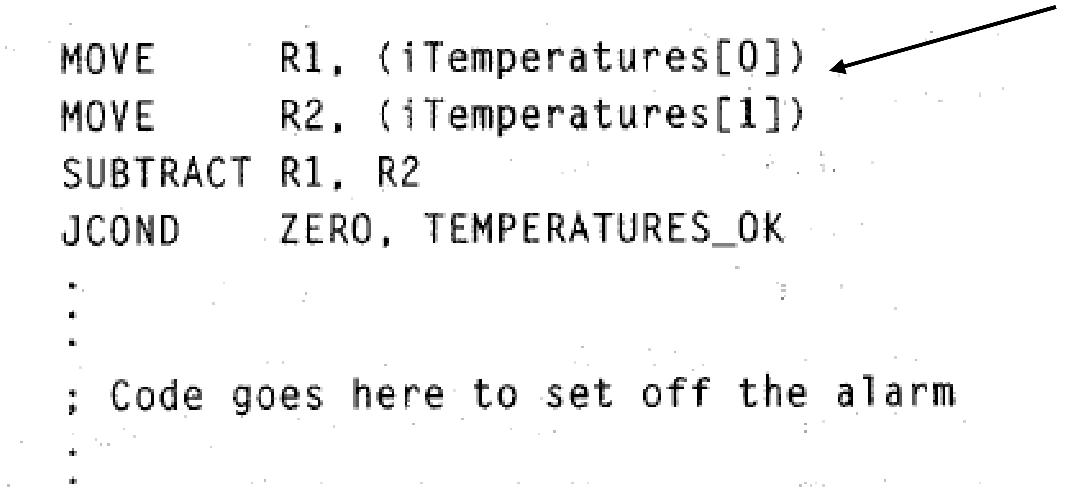
```
static int iTemperatures[2];

void interrupt vReadTemperatures (void)
{
   iTemperatures[0] = !! read in value from hardware
   iTemperatures[1] = !! read in value from hardware
}

void main (void)
{
   while (TRUE)
   {
      if (iTemperatures[0] != iTemperatures[1])
         !! Set off howling alarm;
   }
}
```

The Shared-Data Problem

- **Now, consider the assembly code:**
  - When temperatures are 70 degrees and an interrupt occurs between the two MOVES
  - The temperatures now become 75 degrees
  - On returning from ISR, iTemp[1] will be assigned 75 and an alarm will be set off even though the temperatures were the same



```
MOVE     R1, (iTemperatures[0])
MOVE     R2, (iTemperatures[1])
SUBTRACT R1, R2
JCOND    ZERO, TEMPERATURES_OK
.
.
.
; Code goes here to set off the alarm
.
.
```

The Shared-Data Problem

- ❑ Problem is due to shared array iTemperatures.
- ❑ These bugs are very difficult to find as they occur only when the interrupt occurs in between the first 2 MOVE instructions, other than which code works perfectly.

Solving Shared-Data Problem

- Disable interrupts during instructions that use the shared variable and re-enabling them later

```
 while (TRUE)
{
  disable();        // Disable interrupts
  iTemp0 = iTemperatures[0];
  iTemp1 = iTemperatures[1];
  enable();         // Re-enable interrupts
  ...
 }
```

Solving Shared-Data Problem

- **"Atomic" and "Critical Section"**
  - A part of a program that should not be interrupted
- **Another Example:**
  - An ISR that updates iHours, iMinutes and iSeconds every second through a hardware timer interrupt:

```
long iSecondsSinceMidnight (void) {
  long lReturnVal;
  disable();
  lReturnVal = (((iHours*60)+iMinutes)*60)+iSeconds;
  enable();
  return (lReturnVal);
}
```

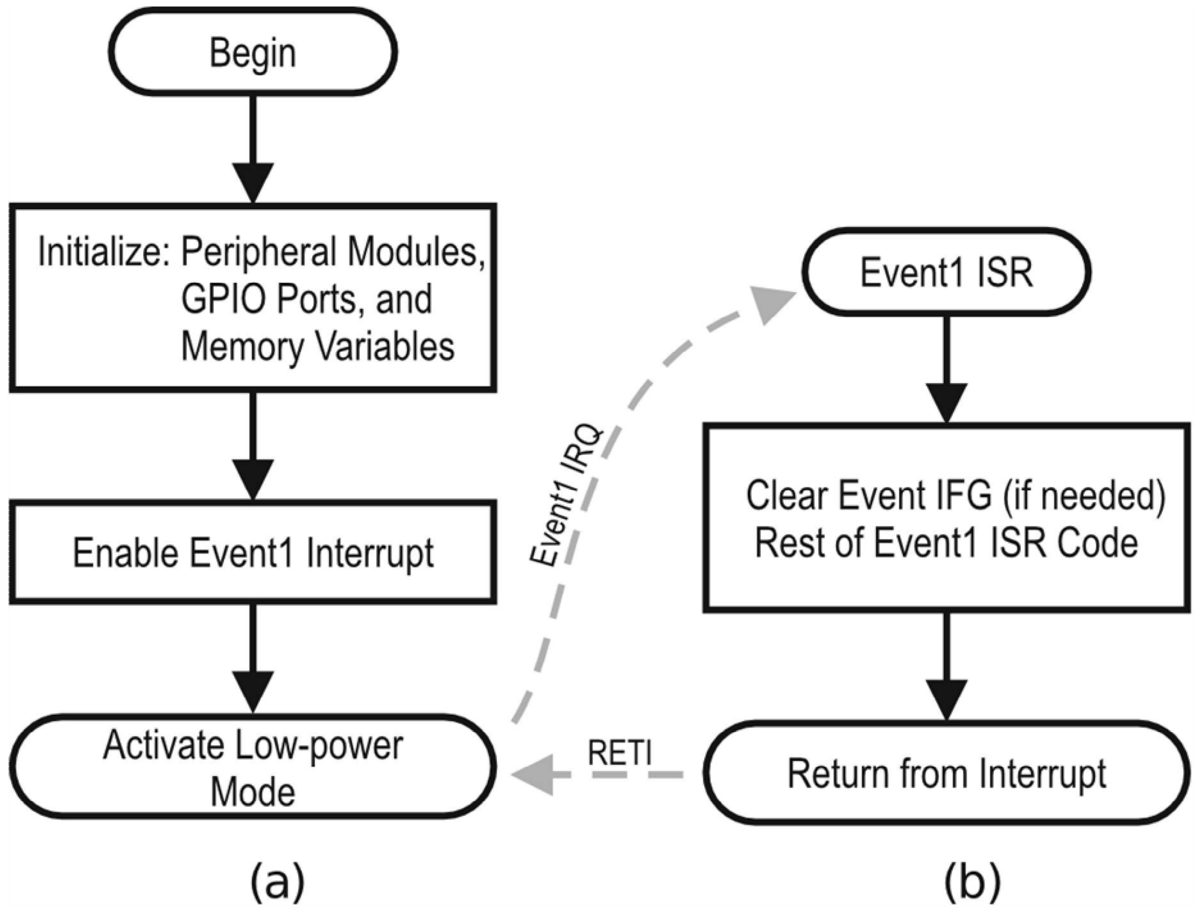

Begin
Initialize: Peripheral Modules, GPIO Ports, and Memory Variables
Enable Event1 Interrupt
Activate Low-power Mode
Event1 IRQ
Event1 ISR
Clear Event IFG (if needed) Rest of Event1 ISR Code
Return from Interrupt
RETI
(a)
(b)

Fig. 7.8 Flowchart of a main program using a low-power mode and a single event ISR. **a** Main program, **b** Interrupt service routine