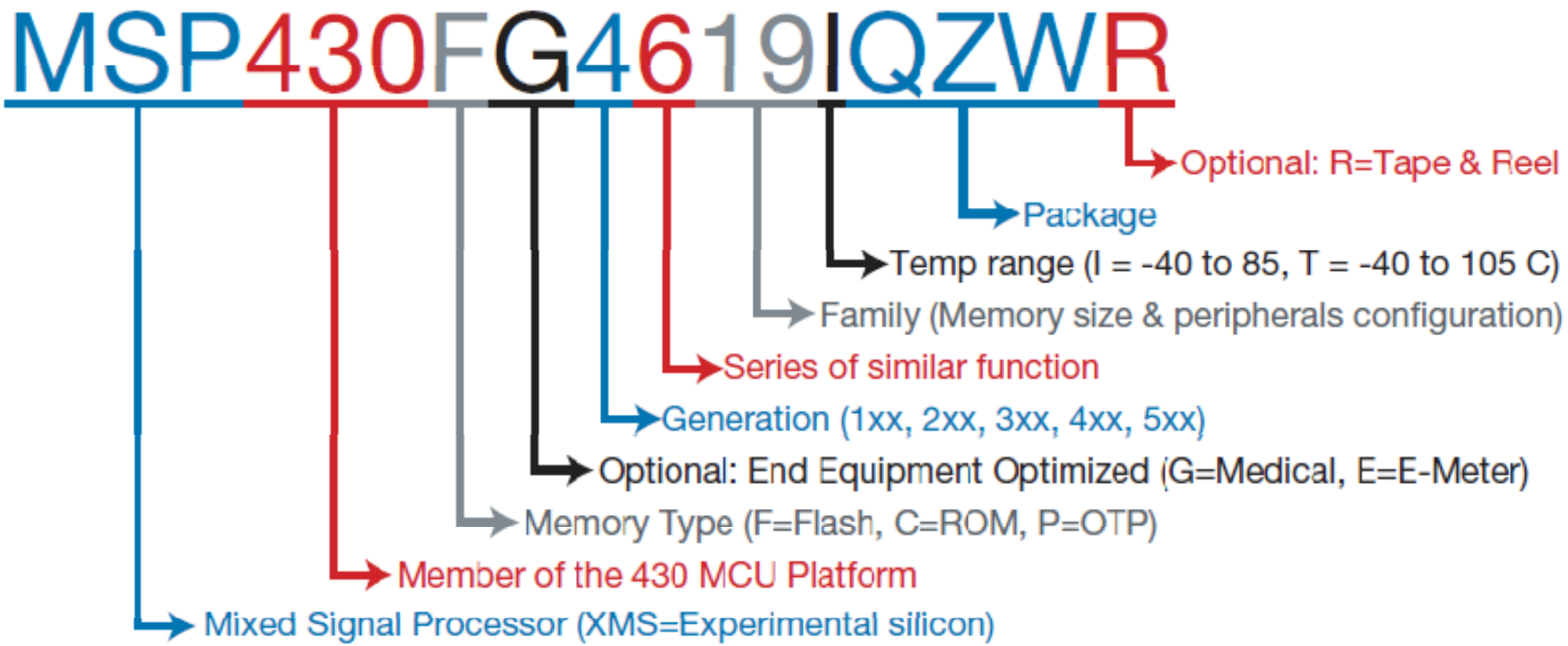
E91

Interrupts, Low Power Modes and Timer A



Interrupts

(Chapter 6 in text)

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
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

Polling vs Interrupt

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
}
```

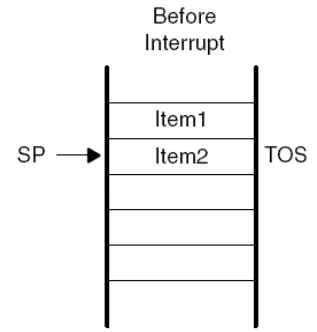
The details are not important now, we will come back to the interrupt version later and go over it line-by-line, bit-by-bit.

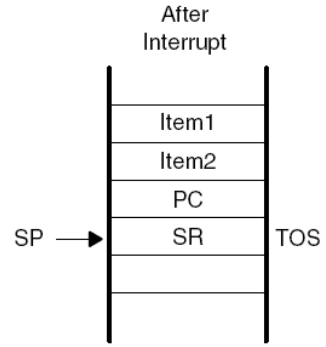
What happens on interrupt?

Interrupt Acceptance

The interrupt latency is 6 cycles (CPU), from the acceptance of an interrupt request to the start of execution of the interrupt-service routine. The interrupt logic executes the following:

1. Any currently executing instruction is completed.
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 occurred during the last instruction and are pending for service.
5. The interrupt request flag resets automatically on single-source flags. Multiple source flags remain set for servicing by software.
6. The SR is cleared. 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.



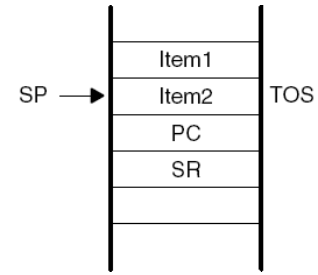


Return From Interrupt

The interrupt handling routine terminates with instruction: `RETI` (return from ISR)

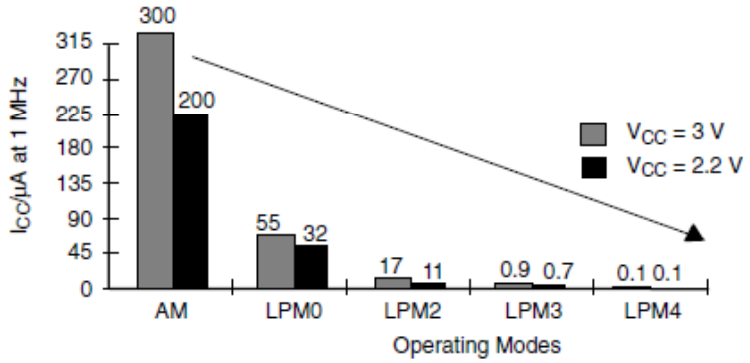
The return from the interrupt takes 5 cycles (CPU) or 3 cycles (CPUx) to execute the following actions.

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.

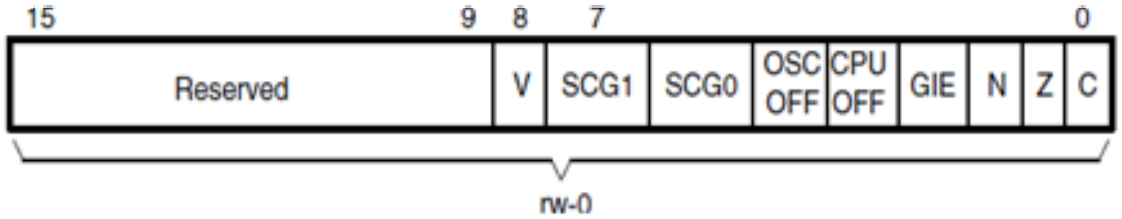


Low power modes

Figure 2–8. Typical Current Consumption of 21x1 Devices vs Operating Modes



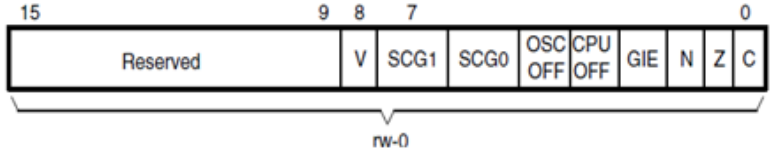
SCG1	SCG0	OSCOFF	CPUOFF	Mode	CPU and Clocks Status
0	0	0	0	Active	CPU is active, all enabled clocks are active
0	0	0	1	LPM0	CPU, MCLK are disabled SMCLK , ACLK are active
0	1	0	1	LPM1	CPU, MCLK are disabled, DCO and DC generator are disabled if the DCO is not used for SMCLK. ACLK is active
1	0	0	1	LPM2	CPU, MCLK, SMCLK, DCO are disabled DC generator remains enabled ACLK is active
1	1	0	1	LPM3	CPU, MCLK, SMCLK, DCO are disabled DC generator disabled ACLK is active
1	1	1	1	LPM4	CPU and all clocks disabled



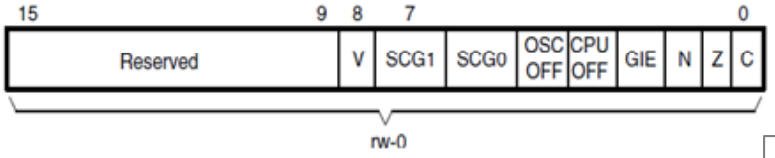
Getting into and out of LPM

An enabled interrupt event wakes the MSP430 from any of the low-power operating modes. The program flow is:

- Enter interrupt service routine:
  - The PC and SR are stored on the stack
  - The CPUOFF, SCG1, and OSCOFF bits are automatically reset
- Options for returning from the interrupt service routine:
  - The original SR is popped from the stack, restoring the previous operating mode.
  - The SR bits stored on the stack can be modified within the interrupt service routine returning to a different operating mode when the `RETI` instruction is executed.



Getting into and out of LPM



ASM Example

```
; Enter LPM0 Example
   BIS    #GIE+CPUOFF,SR           ; Enter LPM0
;  ...                             ; Program stops here
;
; Exit LPM0 Interrupt Service Routine
   BIC    #CPUOFF,0(SP)            ; Exit LPM0 on RETI
   RETI
```

In main routine (enter LPM mode)

In ISR (exit LPM when returning to main program).

Using C

```
__bis_SR_register(CPUOFF + GIE);      // LPM0, ADC10_ISR will force exit

// . . .

// ADC10 interrupt service routine
#pragma vector=ADC10_VECTOR
__interrupt void ADC10_ISR(void) {
  __bic_SR_register_on_exit(CPUOFF);   // Clear CPUOFF bit from 0(SR)
}
```

In main routine (enter LPM mode)

In ISR (exit LPM when returning to main program).

```
_bis_SR_register(unsigned short mask);                    //BIS mask, SR
```

```
_bic_SR_register_on_exit(unsigned short mask);      //BIC mask, saved_SR
```

Registers that effect interrupts on P1

Interrupt Flag	P1IFG	023h	Read/write	Reset with PUC
Interrupt Edge Select	P1IES	024h	Read/write	Unchanged
Interrupt Enable	P1IE	025h	Read/write	Reset with PUC

If a bit in PIES=0, the corresponding bit in P1IFG is set on rising edge on corresponding input pin (P1IN). If PIES=1, P1IFG is set on falling edge.

If PIES=1, P1IFG is set on falling edge of P1IN.

If interrupt enable bit is set in (P1IE), and Global Interrupts are enabled (GIE in Status Register), an interrupt is requested when the corresponding interrupt flag is set (P1IFG).

Note: Writing to PxIESx

Writing to P1IES, or P2IES can result in setting the corresponding interrupt flags.

PxIESx	PxINx	PxIFGx
0 → 1	0	May be set
0 → 1	1	Unchanged
1 → 0	0	Unchanged
1 → 0	1	May be set

Using interrupts on Port 1

Toggles P1.0 on each push of P1.4.

```
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
}
```

P1.0 is output

P1.4 resistor is enabled

P1.4 resistor is connected to logic 1

Enable interrupt on P1.4 (GIE is still cleared)

Set sensitivity to falling edge.

Clear Interrupt flag (just in case).

Enter LPM4 and enable interrupts

Tell compiler to fill in interrupt vector with address of this function

Tell compiler to return from function with “`iret`” (as opposed to “`ret`”)

Toggle P1.0

Clear interrupt flag. (Some interrupts do this automatically, check manual, or example code)

Keep ISR's short!

It is important to keep interrupt service routines short. Since interrupts are disable globally during an ISR, you might miss something important.

If you need to do a lot of processing, have the ISR set a flag, and have main routine act on it.

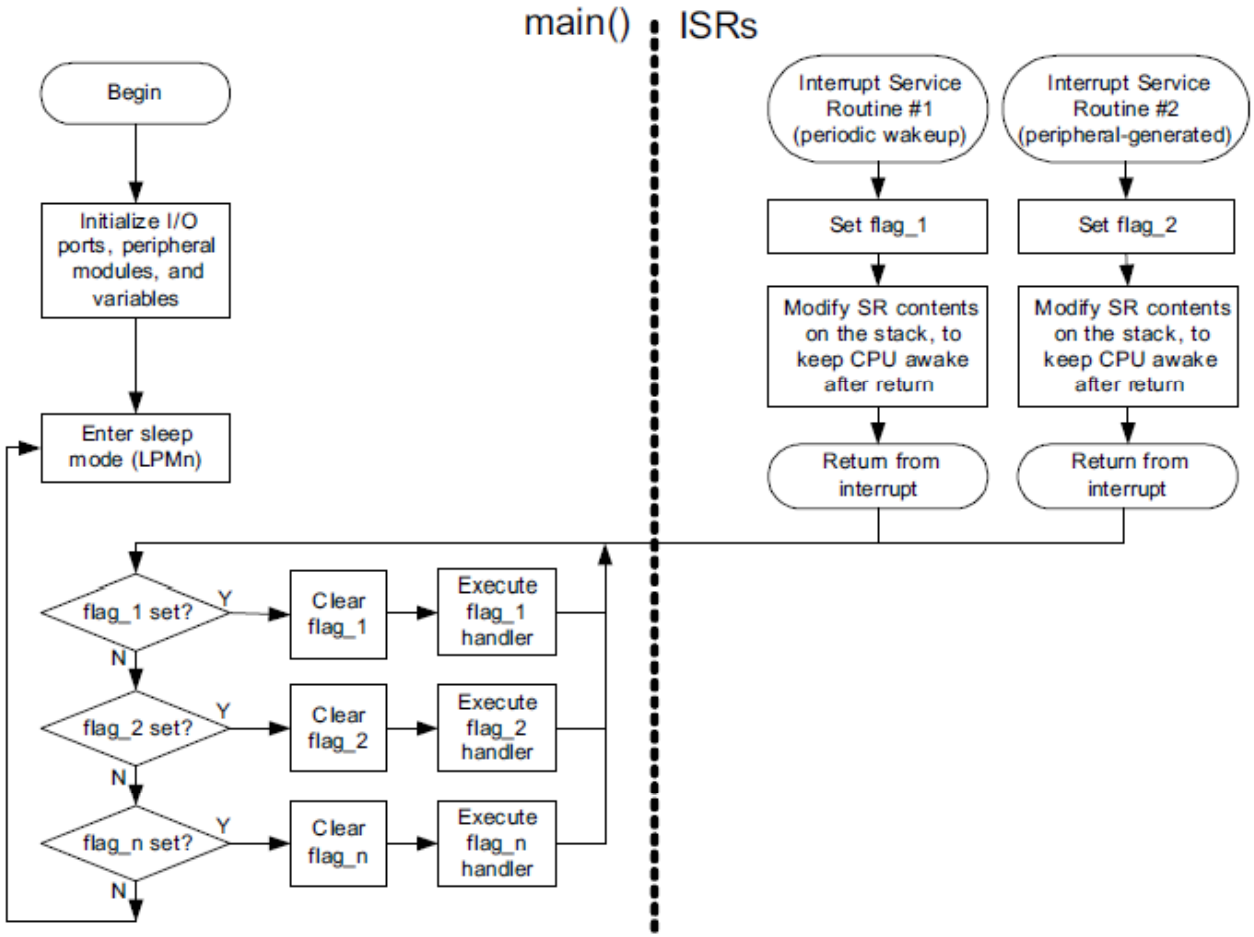


Figure 1. MSP430 Top-Level Code Flow

C → ASM

```
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
}
```

```
0xF800:   40B2 5A80 0120      MOV.W   #0x5a80,&Watchdog_Timer_WDTCTL
0xF806:   43D2 0022           MOV.B   #1,&Port_1_2_P1DIR
0xF80A:   40F2 0010 0021      MOV.B   #0x0010,&Port_1_2_P1OUT
0xF810:   D0F2 0010 0027      BIS.B   #0x0010,&Port_1_2_P1REN
0xF816:   D0F2 0010 0025      BIS.B   #0x0010,&Port_1_2_P1IE
0xF81C:   D0F2 0010 0024      BIS.B   #0x0010,&Port_1_2_P1IES
0xF822:   F0F2 00EF 0023      AND.B   #0x00ef,&Port_1_2_P1IFG
0xF828:   D032 00F8           BIS.W   #0x00f8,ST
0xF82C:   4130                RET
        c_int00, _c_int00_noinit_noexit:
0xF82E:   4031 027E           MOV.W   #0x027e,SP
0xF832:   40B2 F860 0200

                    ( other stuff )

        Port_1:
0xF84C:   E3D2 0021           XOR.B   #1,&Port_1_2_P1OUT
0xF850:   F0F2 00EF 0023      AND.B   #0x00ef,&Port_1_2_P1IFG
0xF856:   1300                RETI

                    ( other stuff )

0xFFDE:   FFFF FFFF           AND.B   @R15+,0xffff(R15)
0xFFE2:   FFFF F84C           AND.B   @R15+,0xf84c(R15)
0xFFE6:   FFFF FFFF           AND.B   @R15+,0xffff(R15)

                    ( other stuff )

0xFFFA:   FFFF FFFF           AND.B   @R15+,0xffff(R15)
        reset_vector:
0xFFFE:   F82E                AND.W   @R8,R14
```

Memory location FFE4,FFE5 contains F84C (the location of the interrupt routine)

Memory location FFFE,FFFF contains F82E (the location of the interrupt routine)

interrupt vector addresses

The interrupt vectors and the power-up starting address are located in the address range of 0FFFFh–0FFC0h. The vector contains the 16-bit address of the appropriate interrupt handler instruction sequence.

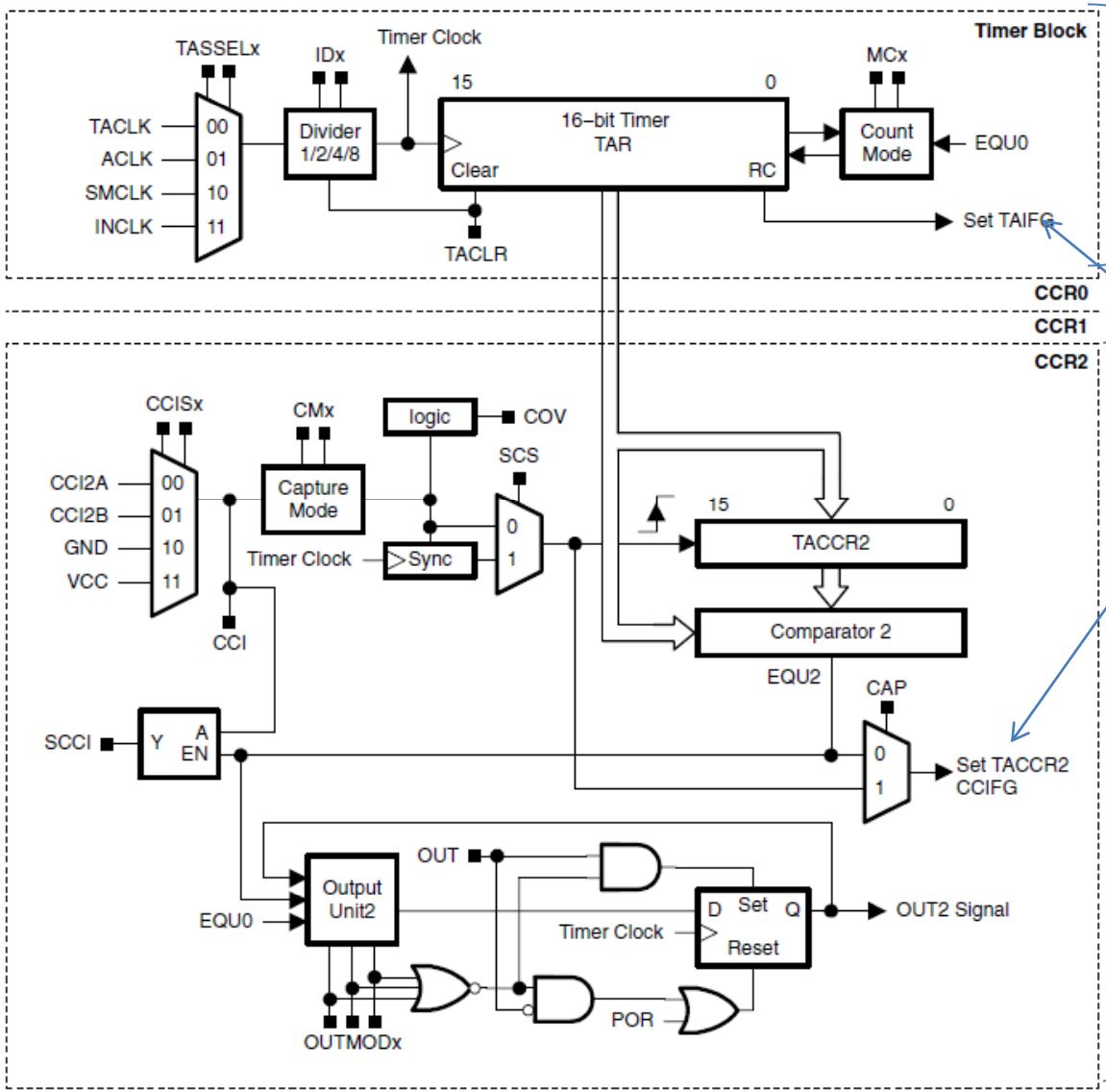
If the reset vector (located at address 0FFFEh) contains 0FFFFh (e.g., flash is not programmed) the CPU will go into LPM4 immediately after power-up.

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-up External reset Watchdog Timer+ Flash key violation PC out-of-range (see Note 1)	PORIFG RSTIFG WDTIFG KEYV (see Note 2)	Reset	0FFFEh	31, highest
NMI Oscillator fault Flash memory access violation	NMIIFG OFIFG ACCVIFG (see Notes 2 and 4)	(non)-maskable, (non)-maskable, (non)-maskable	0FFFCh	30
			0FFFAh	29
			0FFF8h	28
Comparator_A+ (MSP430x20x1 only)	CAIFG (see Note 3)	maskable	0FFF6h	27
Watchdog Timer+	WDTIFG	maskable	0FFF4h	26
Timer_A2	TACCR0 CCIFG (see Note 3)	maskable	0FFF2h	25
Timer_A2	TACCR1 CCIFG. TAIFG (see Notes 2 and 3)	maskable	0FFF0h	24
			0FFEEh	23
			0FFECh	22
ADC10 (MSP430x20x2 only)	ADC10IFG (see Note 3)	maskable	0FFEAh	21
SD16_A (MSP430x20x3 only)	SD16CCTL0 SD16OVIFG, SD16CCTL0 SD16IFG (see Notes 2 and 3)	maskable		
USI (MSP430x20x2, MSP430x20x3 only)	USIIFG, USISTTIFG (see Notes 2 and 3)	maskable	0FFE8h	20
I/O Port P2 (two flags)	P2IFG.6 to P2IFG.7 (see Notes 2 and 3)	maskable	0FFE6h	19
I/O Port P1 (eight flags)	P1IFG.0 to P1IFG.7 (see Notes 2 and 3)	maskable	0FFE4h	18
			0FFE2h	17
			0FFE0h	16
(see Note 5)			0FFDEh ... 0FFC0h	15 ... 0, lowest

NOTES:
1. A reset is generated if the CPU tries to fetch instructions from within the module register memory address range (0h–01FFh) or from within unused address ranges.
2. Multiple source flags
3. Interrupt flags are located in the module
4. (non)-maskable: the individual interrupt-enable bit can disable an interrupt event, but the general interrupt enable cannot.
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.

TIMER A
(very similar to TIMER B)

(Chapter 8 in text)

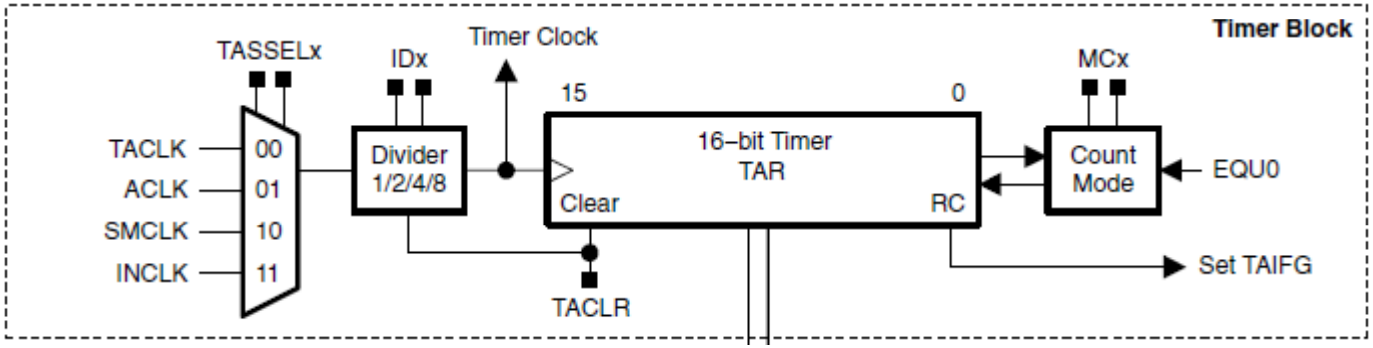


Timer A
Timer / Counter

Note interrupt flags

Capture Compare Registers
(2013 only has
CCR0 and CCR1)

Timer / Counter



- The timer register (TAR) can be read and written and can generate an interrupt on overflow. It can be cleared with TACLR.
- TASSELA selects on of four inputs
- IDA chooses one of four divider
- MCA chooses one of four counting modes

MCx	Mode	Description
00	Stop	The timer is halted.
01	Up	The timer repeatedly counts from zero to the value of TACCR0.
10	Continuous	The timer repeatedly counts from zero to 0FFFFh.
11	Up/down	The timer repeatedly counts from zero up to the value of TACCR0 and back down to zero.

See previous page for definition

TIMER A Registers

Table 12–3. Timer_A Registers

Register	Short Form	Register Type	Address	Initial State
Timer_A control	TACTL	Read/write	0160h	Reset with POR
Timer_A counter	TAR	Read/write	0170h	Reset with POR
Timer_A capture/compare control 0	TACCTL0	Read/write	0162h	Reset with POR
Timer_A capture/compare 0	TACCR0	Read/write	0172h	Reset with POR
Timer_A capture/compare control 1	TACCTL1	Read/write	0164h	Reset with POR
Timer_A capture/compare 1	TACCR1	Read/write	0174h	Reset with POR

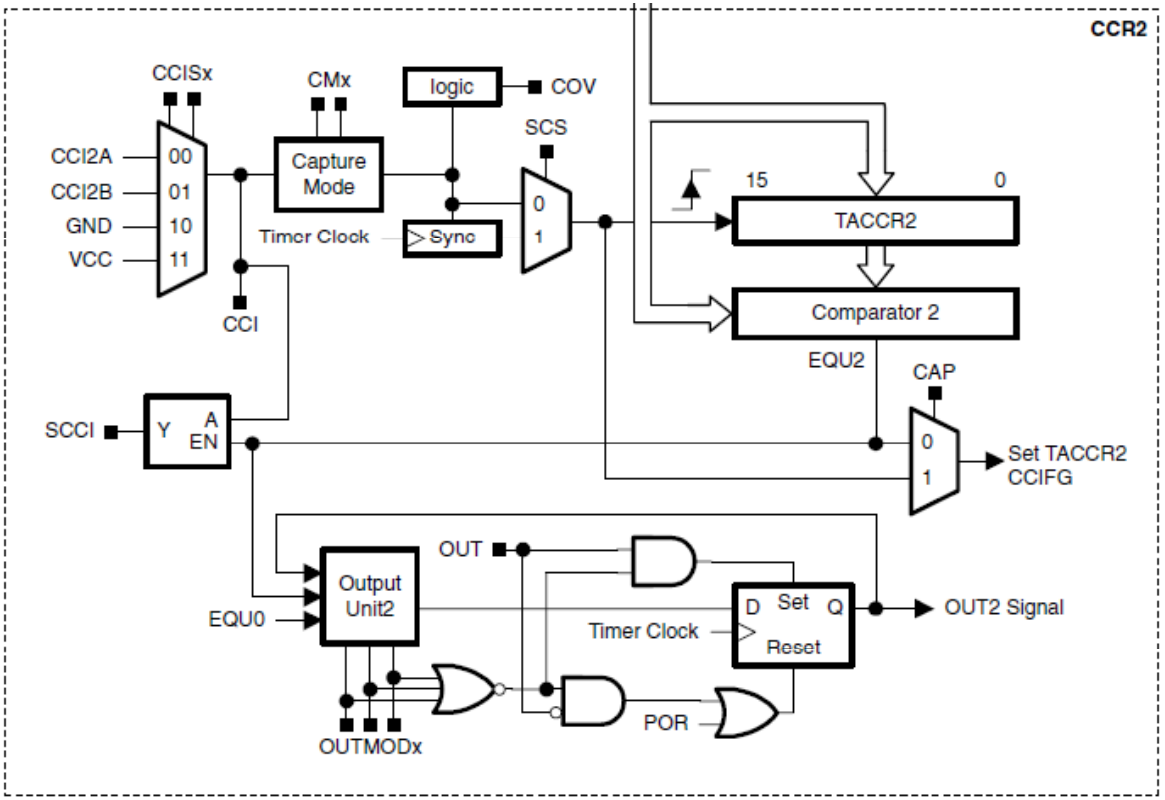
TACTL, Timer_A Control Register

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)

If we wanted to use TAIFG for a periodic interrupt, the ISR would have to set the value of TAR to 0xffff-(desired delay count – 1).

Capture Mode



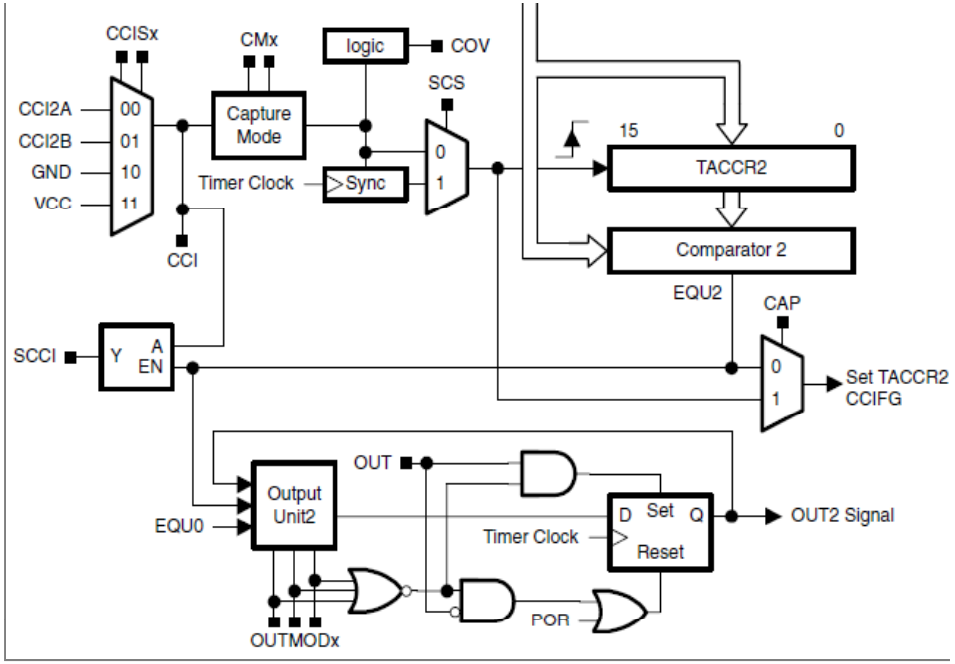
Capture mode (CAP=1) is used to time events on CCIxA or CCIxB (where x is the CCR register).

On a rising edge, falling edge, or both (as determined by CMx) the value of TAR is copied into TACCRx, and the CCIFG flag is set.

Compare Mode

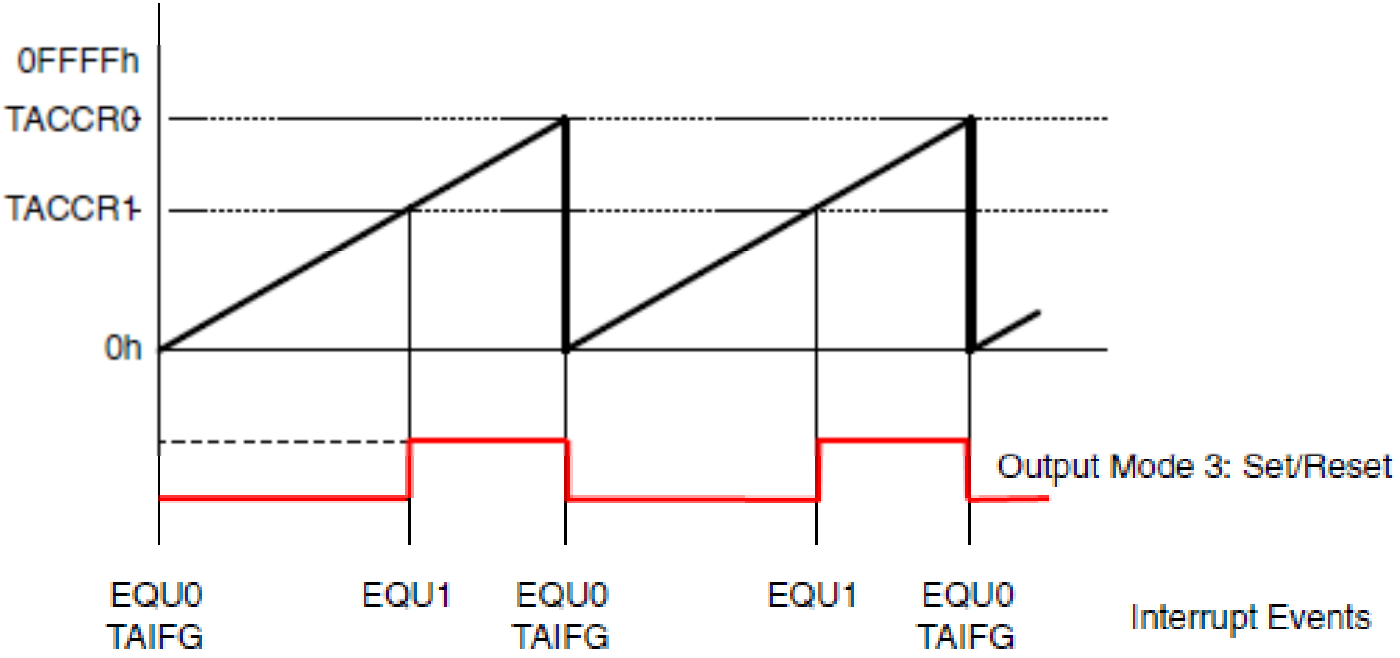
Compare Mode used to generate periodic signals of whose frequency and duty cycles can be altered.

Exact behavior is set by bit EQUx and OUTMODx.



As one example:

- TAR counts to TACCR0 and resets (i.e., TACCR0 determines frequency (along with TAR input frequency))
- Output OUT1 is high when TAR>TACCR1 (i.e., TACCR1 determines pulse width)



PWM

If you need PWM, you need to choose the mode you need:

Table 12–2. Output Modes

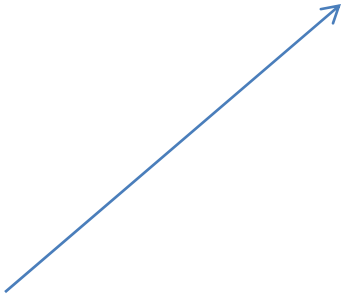
OUTMODx	Mode	Description
000	Output	The output signal OUTx is defined by the OUTx bit. The OUTx signal updates immediately when OUTx is updated.
001	Set	The output is set when the timer *counts* to the TACCRx value. It remains set until a reset of the timer, or until another output mode is selected and affects the output.
010	Toggle/Reset	The output is toggled when the timer *counts* to the TACCRx value. It is reset when the timer *counts* to the TACCR0 value.
011	Set/Reset	The output is set when the timer *counts* to the TACCRx value. It is reset when the timer *counts* to the TACCR0 value.
100	Toggle	The output is toggled when the timer *counts* to the TACCRx value. The output period is double the timer period.
101	Reset	The output is reset when the timer *counts* to the TACCRx value. It remains reset until another output mode is selected and affects the output.
110	Toggle/Set	The output is toggled when the timer *counts* to the TACCRx value. It is set when the timer *counts* to the TACCR0 value.
111	Reset/Set	The output is reset when the timer *counts* to the TACCRx value. It is set when the timer *counts* to the TACCR0 value.

Timer B

13.1.1 Similarities and Differences From Timer_A

Timer_B is identical to Timer_A with the following exceptions:

- The length of Timer_B is programmable to be 8, 10, 12, or 16 bits.
- Timer_B TBCCRx registers are double-buffered and can be grouped.
- All Timer_B outputs can be put into a high-impedance state.
- The SCCI bit function is not implemented in Timer_B.



Grouping is important when PWM's must be synchronized (as with H-bridges – but don't worry if you don't know what an H-bridge is).

References Used

- http://focus.ti.com/lit/an/slaa294a/slaa294a.pdf MSP430 Software Coding Techniques
- http://focus.ti.com/mcu/docs/mcuprodcodeexamples.tsp?sectionId=96&tabId=1468 MSP430 example code
- http://focus.ti.com/lit/ug/slau132e/slau132e.pdf MSP430 Optimizing C/C++ Compiler v 3.3