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
}

#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

<table>
<thead>
<tr>
<th>SCG1</th>
<th>SCG0</th>
<th>OSCOFF</th>
<th>CPUOFF</th>
<th>Mode</th>
<th>CPU and Clocks Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Active</td>
<td>CPU is active, all enabled clocks are active</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>LPM0</td>
<td>CPU, MCLK are disabled, SMCLK, ACLK are active</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>LPM1</td>
<td>CPU, MCLK are disabled, DCO and DC generator are disabled if the DCO is not used for SMCLK, ACLK is active</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>LPM2</td>
<td>CPU, MCLK, SMCLK, DCO are disabled, ACLK remains enabled, ACLK is active</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>LPM3</td>
<td>CPU, MCLK, SMCLK, DCO are disabled, ACLK is active</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>LPM4</td>
<td>CPU and all clocks disabled</td>
</tr>
</tbody>
</table>

![Register Diagram]
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.

![Diagram of register bits](image)
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)
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 vec=ADC10_VECTOR
__interrupt void ADC10_ISR(void) {
  __bic_SR_register_on_exit(CPUOFF);    // Clear CPUOFF bit from 0(SR)
}

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

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

In main routine (enter LPM mode)

In ISR (exit LPM when returning to main program).
 Registers that effect interrupts on P1

<table>
<thead>
<tr>
<th>Interrupt Flag</th>
<th>P1IFG</th>
<th>023h</th>
<th>Read/write</th>
<th>Reset with PUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt Edge Select</td>
<td>P1IES</td>
<td>024h</td>
<td>Read/write</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Interrupt Enable</td>
<td>P1IE</td>
<td>025h</td>
<td>Read/write</td>
<td>Reset with PUC</td>
</tr>
</tbody>
</table>

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 P1IESx

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

<table>
<thead>
<tr>
<th>P1IESx</th>
<th>P1INx</th>
<th>P1IFGx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 → 1</td>
<td>0</td>
<td>May be set</td>
</tr>
<tr>
<td>0 → 1</td>
<td>1</td>
<td>Unchanged</td>
</tr>
<tr>
<td>1 → 0</td>
<td>0</td>
<td>Unchanged</td>
</tr>
<tr>
<td>1 → 0</td>
<td>1</td>
<td>May be set</td>
</tr>
</tbody>
</table>
Using interrupts on Port 1

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

```c
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.
C → ASM

```c
void main(void) {
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog
    P1DIR = 0x01;              // P1.0 output
    P1OUT = 0x10;              // P1.4 hi (pullup)
P1REN |= 0x10;              // P1.4 pullup
    PIE |= 0x10;               // P1.4 IRQ enabled
    PIES |= 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
    P1IFG &= ~0x10;            // P1.4 IFG cleared
    P1IFG &= ~0x10;            // P1.4 IFG cleared

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

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 0FF00h–0FFC0h. The vector contains the 16-bit address of the appropriate interrupt handler or 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.

<table>
<thead>
<tr>
<th>INTERRUP TORCE</th>
<th>INTERRUPT FLAG</th>
<th>SYSTEM INTERRUPT</th>
<th>WORD ADDRESS</th>
<th>PRIORITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power-up</td>
<td>PORIFG</td>
<td>Root</td>
<td>0FFFEh</td>
<td>31, highest</td>
</tr>
<tr>
<td></td>
<td>RSTIFG</td>
<td></td>
<td>0FFFEh</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WDTIFQ</td>
<td></td>
<td>0FFFEh</td>
<td></td>
</tr>
<tr>
<td></td>
<td>KEYV</td>
<td></td>
<td>0FFFEh</td>
<td></td>
</tr>
<tr>
<td>NMI</td>
<td>NMIIFG</td>
<td>(non)-maskable,</td>
<td>0FFFCf</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>OFIFG</td>
<td>(non)-maskable,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACOVIFG</td>
<td>(non)-maskable</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0FFFAh</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0FFF8h</td>
<td>28</td>
</tr>
<tr>
<td>Comparator_A+</td>
<td>CAIFG (see Note 3)</td>
<td>maskable</td>
<td>0FFF6h</td>
<td>27</td>
</tr>
<tr>
<td>(MSP430x20x1 only)</td>
<td></td>
<td></td>
<td>0FFF4h</td>
<td>26</td>
</tr>
<tr>
<td>Watchdog Timer+</td>
<td>WDITFG</td>
<td></td>
<td>0FFF3h</td>
<td>25</td>
</tr>
<tr>
<td>Timer_A2</td>
<td>TACC0 TCGIFG</td>
<td>maskable</td>
<td>0FFF2h</td>
<td>24</td>
</tr>
<tr>
<td>Timer_A2</td>
<td>TACC1 TCGIFG</td>
<td>maskable</td>
<td>0FFF0h</td>
<td>23</td>
</tr>
<tr>
<td>ADC10 (MSP430x20x2 only)</td>
<td>ADC10IFG</td>
<td>maskable</td>
<td>0FFFFFFh</td>
<td>22</td>
</tr>
<tr>
<td>SD16_A (MSP430x20x3 only)</td>
<td>SD16CCTL0, SD16OVIFG, SD16CCTL0, SD16IFG (see Notes 2 and 3)</td>
<td>maskable</td>
<td>0FFFEh</td>
<td>21</td>
</tr>
<tr>
<td>USI</td>
<td>USIIIFG, USISTIFG (see Notes 2 and 3)</td>
<td>maskable</td>
<td>0FFFAh</td>
<td>20</td>
</tr>
<tr>
<td>I/O Port F2</td>
<td>P2IFG.6 to P2IFG.7 (see Notes 2 and 3)</td>
<td>maskable</td>
<td>0FFFAh</td>
<td>19</td>
</tr>
<tr>
<td>(two flags)</td>
<td></td>
<td></td>
<td>0FFF8h</td>
<td>18</td>
</tr>
<tr>
<td>I/O Port P1</td>
<td>P1IFG.0 to P1IFG.7 (see Notes 2 and 3)</td>
<td>maskable</td>
<td>0FFF2h</td>
<td>17</td>
</tr>
<tr>
<td>(eight flags)</td>
<td></td>
<td></td>
<td>0FFF0h</td>
<td>16</td>
</tr>
<tr>
<td>(see Note 5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

1. A reset occurs if the CPU tries to fetch instructions from within the module high memory address range (0-0 FFH) or from an unused address range.
2. Multiple vector flags are located in the module that enable bit can disable an interrupt event, but the general interrupt enable cannot.
3. The interrupt vectors at addresses from 0FF00h 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 one of four inputs
- IDA chooses one of four divider modes
- MCA chooses one of four counting modes

<table>
<thead>
<tr>
<th>MCx</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Stop</td>
<td>The timer is halted.</td>
</tr>
<tr>
<td>01</td>
<td>Up</td>
<td>The timer repeatedly counts from zero to the value of TACCR0.</td>
</tr>
<tr>
<td>10</td>
<td>Continuous</td>
<td>The timer repeatedly counts from zero to 0FFFFh.</td>
</tr>
<tr>
<td>11</td>
<td>Up/down</td>
<td>The timer repeatedly counts from zero up to the value of TACCR0 and back down to zero.</td>
</tr>
</tbody>
</table>
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 (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)
If you need PWM, you need to choose the mode you need:

**Table 12–2. Output Modes**

<table>
<thead>
<tr>
<th>OUTMODx</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Output</td>
<td>The output signal OUTx is defined by the OUTx bit. The OUTx signal updates immediately when OUTx is updated.</td>
</tr>
<tr>
<td>001</td>
<td>Set</td>
<td>The output is set when the timer <em>counts</em> to the TACCRx value. It remains set until a reset of the timer, or until another output mode is selected and affects the output.</td>
</tr>
<tr>
<td>010</td>
<td>Toggle/Reset</td>
<td>The output is toggled when the timer <em>counts</em> to the TACCRx value. It is reset when the timer <em>counts</em> to the TACCR0 value.</td>
</tr>
<tr>
<td>011</td>
<td>Set/Reset</td>
<td>The output is set when the timer <em>counts</em> to the TACCRx value. It is reset when the timer <em>counts</em> to the TACCR0 value.</td>
</tr>
<tr>
<td>100</td>
<td>Toggle</td>
<td>The output is toggled when the timer <em>counts</em> to the TACCRx value. The output period is double the timer period.</td>
</tr>
<tr>
<td>101</td>
<td>Reset</td>
<td>The output is reset when the timer <em>counts</em> to the TACCRx value. It remains reset until another output mode is selected and affects the output.</td>
</tr>
<tr>
<td>110</td>
<td>Toggle/Set</td>
<td>The output is toggled when the timer <em>counts</em> to the TACCRx value. It is set when the timer <em>counts</em> to the TACCR0 value.</td>
</tr>
<tr>
<td>111</td>
<td>Reset/Set</td>
<td>The output is reset when the timer <em>counts</em> to the TACCRx value. It is set when the timer <em>counts</em> to the TACCR0 value.</td>
</tr>
</tbody>
</table>
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/mcu/docs/mcuprodcodeexamples.tsp?sectionId=96&tabId=1468](http://focus.ti.com/mcu/docs/mcuprodcodeexamples.tsp?sectionId=96&tabId=1468)  MSP430 example code