Chapter 9 – Interrupts
Topics to Cover…

- Interrupts
- Interrupt Service Routines (ISR’s)
- Watchdog Timer
  - Example 9.4 – Watchdog ISR
- Energy Consumption
- Processor Clocks
  - Example 9.2 – Master Clock
- Low Power Modes
  - Example 9.3 – Low-Power Mode
- Timers
  - Example 9.4 – Interrupts w/Timer_A
- Pulse Width Modulation (PWM)
  - Example 9.5 – LED PWM w/Timer_A
- Speaker (Transducer)
  - Example 9.6 – Speaker PWM w/Watchdog
Interrupt Service Routines

Interrupt Service Routine (asynchronous)
Interrupts

- Execution of a program normally proceeds predictably, with *interrupts* being the exception.
- An *interrupt* is an asynchronous signal indicating something needs attention.
  - Some event has occurred
  - Some event has completed
- The processing of an interrupt subroutine uses the stack.
  - Processor stops with it is doing,
  - stores enough information on the stack to later resume,
  - executes an *interrupt service routine* (ISR),
  - restores saved information from stack (*RETI*),
  - and then resumes execution at the point where the processor was executing before the interrupt.
Interrupt Flags

- Each interrupt has a flag that is raised (set) when the interrupt is pending.
- Each interrupt flag has a corresponding enable bit – setting this bit allows a hardware module to request an interrupt.
- Most interrupts are **maskable**, which means they can only interrupt if
  1) Individually enabled and
  2) general interrupt enable (GIE) bit is set in the status register (SR).
- Reset and Non-Maskable Interrupts (NMI) are reserved for system interrupts such as power-up (PUC), external reset, oscillator fault, illegal flash access, watchdog, and illegal instruction fetch.
Interrupt Vectors

- The CPU must know where to fetch the next instruction following an interrupt.
- The address of an ISR is defined in an *interrupt vector*.
- The MSP430 uses *vectored interrupts* where each ISR has its own vector stored in a *vector table* located at the end of program memory.
- Note: The *vector table* is at a fixed location (defined by the processor data sheet), but the ISRs can be located anywhere in memory.
# MSP430 Interrupt Vectors

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Interrupt Flag</th>
<th>Address</th>
<th>Section</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power-up</td>
<td>PORIFG</td>
<td>0xFFFE</td>
<td>.reset</td>
<td>15, highest</td>
</tr>
<tr>
<td>External reset</td>
<td>RSTIFG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Watchdog</td>
<td>WDTIFG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NMI</td>
<td>NMIIFG</td>
<td>0xFFFC</td>
<td>.int14</td>
<td>14</td>
</tr>
<tr>
<td>Oscillator fault</td>
<td>OFIFG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash memory violation</td>
<td>ACCDVIFG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timer_B3</td>
<td>TBCCRO CCIFG</td>
<td>0xFFFF8</td>
<td>.int12</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>TBCCR1 CCIFG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TBCCR2 CCIFG,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TBIFG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Watchdog Timer</td>
<td>WDTIFG</td>
<td>0xFFF4</td>
<td>.int10</td>
<td>10</td>
</tr>
<tr>
<td>Timer_A3</td>
<td>TACCR0 CCIFG</td>
<td>0xFFF2</td>
<td>.int09</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>TACCR1 CCIFG,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TACCR2 CCIFG,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TAIFG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USCI_A0/USCI_B0 Rx</td>
<td>UCA0RXIFG, USB0RXIFG</td>
<td>0xFFF0</td>
<td>.int08</td>
<td>8</td>
</tr>
<tr>
<td>USCI_Z0/USCI_B0 Rx</td>
<td>UCA0TXIFG, UCB0TXIFG</td>
<td>0xFFF6</td>
<td>.int07</td>
<td>7</td>
</tr>
<tr>
<td>ADC10</td>
<td>ADC10IFG</td>
<td>0xFFF8</td>
<td>.int06</td>
<td>6</td>
</tr>
<tr>
<td>I/O Port P2</td>
<td>P2IFG.0 – P2IFG.7</td>
<td>0xFFF4</td>
<td>.int05</td>
<td>5</td>
</tr>
<tr>
<td>I/O Port P1</td>
<td>P1IFG.0 – P1IFG.7</td>
<td>0xFFF2</td>
<td>.int04</td>
<td>4</td>
</tr>
</tbody>
</table>

- **Non-Maskable Interrupts**
- **Timers**
- **Ports**
Processing an Interrupt...

1. Processor completes execution of current instruction.
2. Master Clock (MCLK) started (if CPU was off).
3. Processor pushes Program Counter (PC) on stack.
4. Processor pushes Status Register (SR) on stack.
5. Interrupt w/highest priority is selected.
6. Interrupt request flag cleared (if single sourced).
7. Status Register is cleared:
   - Disables further maskable interrupts (GIE cleared)
   - Terminates low-power mode
8. Processor fetches interrupt vector and stores it in the program counter.
9. User ISR must do the rest!
## Return From Interrupt

### Single operand instructions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH (.B or .W) src</td>
<td>SP-2→SP, src→@SP</td>
<td>Push byte/word source on stack</td>
</tr>
<tr>
<td>CALL dst</td>
<td>dst→tmp, SP-2→SP, PC→@SP, tmp→PC</td>
<td>Subroutine call to destination</td>
</tr>
<tr>
<td>RETI</td>
<td>TOS→SR, SP+2→SP, TOS→PC, SP+2→SP</td>
<td>Return from interrupt</td>
</tr>
</tbody>
</table>

### Emulated instructions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Emulation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RET</td>
<td>@SP→PC</td>
<td>MOV @SP+,PC</td>
<td>Return from subroutine</td>
</tr>
<tr>
<td>POP (.B or .W) dst</td>
<td>@SP→temp, SP+2→SP, temp→dst</td>
<td>MOV (.B or .W) @SP+, dst</td>
<td>Pop byte/word from stack to destination</td>
</tr>
</tbody>
</table>
Interrupt Stack

Prior to Interrupt

SP

PC

Interrupt (hardware)
- Program Counter pushed on stack
- Status Register pushed on stack
- Interrupt vector moved to PC
- Further interrupts disabled
- Interrupt flag cleared

SP

PC

Execute Interrupt Service Routine (ISR)

SP

PC

Return from Interrupt (reti)
- Status Register popped from stack
- Program Counter popped from stack

SP

PC

add.w r4,r7
jnc $+4
add.w #1,r6
add.w r5,r6
xor.b #1,&P1OUT
reti

add.w r4,r7
jnc $+4
add.w #1,r6
add.w r5,r6
Interrupt Latency

- The time between the interrupt request and the start of the ISR is called latency
  - MSP430 requires 6 clock cycles before the ISR begins executing
  - An ISR may be interrupted if interrupts are enabled in the ISR

Well-written ISRs:
- Should be short and fast – get in and get out
- Require a balance between doing very little – thereby leaving the background code with lots of processing – and doing a lot and leaving the background code with nothing to do

Applications that use interrupts should:
- Disable interrupts as little as possible
- Respond to interrupts as quickly as possible
- Communicate w/ISR only through global variables (never through registers!!!)
Watchdog
Watchdog Timer

- The MSP430 watchdog can be configured as a COP (computer operating properly) device or as a timer.
- The primary function of the watchdog timer (WDT+) module is to perform a controlled system restart after a software problem occurs.
  - After a power-up cycle (PUC), the WDT+ module is automatically configured in watchdog mode with an initial 32768 clock cycle reset interval using the DCOCLK.
  - The user must setup or halt the WDT+ prior to the expiration of the initial reset interval, else an unmasked system reset is generated.
- If the watchdog function is not needed in an application, the module can be configured as an interval timer and can generate interrupts at selected time intervals.
### WDT+ Control Register (WDTCTL)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value 1</th>
<th>Value 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-8</td>
<td>WDTPW WDT+ password:</td>
<td>0x69 ₹</td>
<td>0x5A ₹</td>
</tr>
<tr>
<td>7</td>
<td>WDTHOLD WDT+ hold:</td>
<td>0 ⇒ Enabled</td>
<td>1 ⇒ Stopped</td>
</tr>
<tr>
<td>6</td>
<td>WDTNMIES WDT+ NMI edge select:</td>
<td>0 ⇒ NMI on rising edge</td>
<td>1 ⇒ NMI on falling edge</td>
</tr>
<tr>
<td>5</td>
<td>WDTNMI WDT+ NMI select:</td>
<td>0 ⇒ Reset function</td>
<td>1 ⇒ NMI function</td>
</tr>
<tr>
<td>4</td>
<td>WDTTMSEL WDT+ mode select:</td>
<td>0 ⇒ Watchdog mode</td>
<td>1 ⇒ Interval timer mode</td>
</tr>
<tr>
<td>3</td>
<td>WDTCNTCL WDT+ counter clear:</td>
<td>0 ⇒ No action</td>
<td>1 ⇒ WDTCNT = 0x0000</td>
</tr>
<tr>
<td>2</td>
<td>WDTISSEL WDT+ clock source select:</td>
<td>0 ⇒ SMCLK</td>
<td>1 ⇒ ACLK</td>
</tr>
<tr>
<td>1-0</td>
<td>WDTISx WDT+ interval select:</td>
<td>0 ⇒ WD clock source / 32768</td>
<td>1 ⇒ WD clock source / 8192</td>
</tr>
</tbody>
</table>
Example 9.1 – Watchdog ISR

```
.dcecls C,"msp430.h"

SMCLK .equ 1200000 ; 1.2 Mhz clock
WDT_CTL .equ WDT_MDLY_8 ; WDT SMCLK, 8 ms (@1 MHz)
WDT_CPS .equ SMCLK/8000 ; WD clocks / second count

; Data Section -------------------------------------------------------------
.bss WDTSecCnt,2 ; WDT second counter

; Code Section -------------------------------------------------------------
.text
.start:      mov.w #0x400,SP               ; initialize stack pointer
            mov.w #WDT_CTL,&WDTCTL        ; set WD timer interval
            mov.w #WDT_CPS,&WDTSecCnt ; initialize 1 sec WD counter
            bis.b ...      ; enable WDT interrupt
            bis.b #0x01,&P1DIR            ; P1.0 output
            xor.b #0x01,&P1OUT            ; toggle P1.0

loop:       ;<< program >>
            jmp loop                    ; loop indefinitely

; Watchdog ISR -------------------------------------------------------------
WDT_ISR:    dec.w &WDTSecCnt ; decrement counter, 0?
            jne WDT_02 ; n
            mov.w #WDT_CPS,&WDTSecCnt ; y, re-initialize
            xor.b #0x01,&P1OUT ; toggle P1.0

WDT_02:     reti ; return from interrupt

; Interrupt Vectors --------------------------------------------------------
.set   ".int10" ; Watchdog Vector
.word   WDT_ISR ; Watchdog ISR
.set   ".reset" ; PUC Vector
.word   start ; RESET ISR
```

8 ms (@1 MHz SMCLK)

WDT_CPS = clocks/second

Configure Watchdog as a timer and enable it to interrupt

Watchdog Interrupt Service Routine
Quiz 9.1

1. What conditions must be met before a device can interrupt the computer?

2. What is saved on the stack when an interrupt occurs?

3. Where are interrupt vectors located in memory? ISRs?

4. (T or F) Interrupts are predictable asynchronous events.
Energy Consumption
U.S. Energy Consumption

- The United States is the 2nd largest energy consumer in terms of total use in 2010\(^1\) in the world.
- Not getting better – everyone must do their part.
- How could I lower home energy consumption?
  - Turn off lights
  - Turn down thermostat
  - Unplug appliances not in use
  - Use slower motor speeds
  - Heat/cool at a slower rate
  - Improve insulation
  - Purchase energy efficient appliances
  - Eliminate unnecessary living space
- How could I make computers more energy efficient?

Computers occupy a small but growing percentage of annual U.S. electricity consumption.
- Estimates vary for 3% to 13% of entire U.S. supply.
- Hidden costs – every 100 watts consumed by computers requires 50 watts of cooling.
- Computers are ubiquitous – found in every energy sector.

Contemporary processors require more electricity than their predecessors.

Server farms, supercomputers, web hosting, scientific simulations, 3D rendering, search engines,…
- Google uses enough energy to continuously power 200,000 homes (260 million watts – ¼ output of a nuclear power plant).
- One Google search is equal to turning on a 60W light bulb for 17 seconds.
Clocks
Processor Clock Speeds

- Often, the most important factor for reducing power consumption is slowing the clock down.
  - Faster clock = Higher performance, more power required
  - Slower clock = Lower performance, less power required

```
; Set ACLK to ≈12kHz
mov.w #LFXT1S_2,&BCSCTL3

; Set DCO to 8 MHz:
mov.b #CALBC1_8MHZ,&BCSCTL1
mov.b #CALDCO_8MHZ,&DCOCTL
```

**Power Consumption**

- 10% loss of power
- 60% loss of power
Example 9.2 – Clock Speed

```assembly
.cdecls C,"msp430.h"

SMCLK .equ 1200000 ; 1.2 MHz clock
WDT_CTL .equ WDT_MDLY_8 ; WDT SMCLK, 8 ms (@1 MHz)
WDT_CPS .equ SMCLK/8000 ; WDT clocks / set and flag
MHz8 .set 1 ; 8 MHz flag

; Code Section --------------------------------------------

.text

start:           mov.w #0x0400,SP              ; init stack pointer
                 mov.w #WDTPW|WDTHOLD,&WDTCTL  ; stop WDT
                 .if     MHz8                    ; set DCO to 8 MHz
                 mov.b #CALBC1_8MHZ,&BCSCTL1   ; set range
                 mov.b #CALDCO_8MHZ,&DCOCTL    ; set DCO step + modulation
                 .endif

                 bis.b #0x01,&P1DIR            ; set P1.0 as output

                 mainloop:  xor.b #0x01,&P1OUT            ; toggle P1.0
                             mov.w #8,r14                  ; use R14 as outer loop counter

                 delaylp1: mov.w #0,r15                  ; use R15 as delay counter
                             delaylp2:   dec.w r15                     ; delay over?
                                               jne delaylp2                ; n
                                               dec.w r14                     ; y, outer loop done?
                                               jne delaylp1                ; n
                                               jmp mainloop ; y, toggle led

; Interrupt Vectors --------------------------------------

;sect ".reset"                ; MSP430 RESET Vector
.word   start                   ; start address
.end
```

Conditionally assemble setting new DCO constants (for 8 MHz)

Nested delay loop to blink LED
Low-Power Mode
Another method to reduce power consumption is to turn off some (or all) of the system clocks.

A device is said to be *sleeping* when in low-power mode; *waking* refers to returning to active mode.
### MSP430 Clock Settings

<table>
<thead>
<tr>
<th>Mode</th>
<th>V</th>
<th>SCG1</th>
<th>SCG0</th>
<th>OSC Off</th>
<th>CPU Off</th>
<th>GIE</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>Current Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Active Mode</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>~ 250 µA</td>
</tr>
<tr>
<td><strong>LPM0</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>~ 35 µA</td>
</tr>
<tr>
<td><strong>LPM3</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>~ 0.8 µA</td>
</tr>
<tr>
<td><strong>LPM4</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>~ 0.1 µA</td>
</tr>
</tbody>
</table>

**SMCLK and ACLK Active**

**Only ACLK Active**

**Sleep Modes**

No Clocks!

<table>
<thead>
<tr>
<th>; enable interrupts / enter low-power mode 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bis.w #LPM0+GIE,SR ; LPM0 w/interrupts</td>
</tr>
</tbody>
</table>
Example 9.3 – Low Power

```assembly
.cdecls C,"msp430.h"

SMCLK .equ 1200000 ; 1.2 Mhz clock
WDT_CTL .equ WDT_MDLY_8 ; WDT SMCLK, 8 ms (@1 Mhz)
WDT_CPS .equ SMCLK/8000 ; WDT clocks / second count
STACK .equ 0x0400 ; top of stack

; Data Section ---------------------------------------------------------------------
.bss WDTSecCnt,2 ; WDT second counter

; Code Section ---------------------------------------------------------------------
.text
start:  mov.w #STACK,SP               ; initialize stack pointer
        mov.w #WDT_CTL,&WDTCTL        ; set WD timer interval
        mov.w #WDT_CPS,&WDTSecCnt ; initialize 1 sec WD counter
        bis.b #WDTIE,&IE1             ; enable WDT interrupt
        bis.b #0x01,&P1DIR            ; P1.0 output

loop:   bis.w #LPM0|GIE,SR            ; sleep/enable interrupts
        xor.b #0x01,&P1OUT            ; toggle P1.0
        jmp loop                    ; loop indefinitely

; Watchdog ISR ---------------------------------------------------------------------
WDT_ISR: dec.w &WDTSecCnt ; decrement counter
        jnz WDT_02                  ; y, re-initialize
        mov.w #WDT_CPS,&WDTSecCnt ; y, re-initialize
        bic.b #LPM0,0(SP)             ; wakeup processor
        reti ; return from interrupt

; Interrupt Vectors ---------------------------------------------------------------
.sect   "int10"                ; Watchdog Vector
.word   WDT_ISR                 ; Watchdog ISR
.sect   "reset"                ; PUC Vector
.word   start                   ; RESET ISR
.end

Low-Power Mode
1. Enable interrupts
2. Goto Sleep (Low-power Mode 0)
3. Blink LED when awakened

300µA 1µA
 Activity Profile
1. Reset counter every second
2. Set Active Mode in saved SR
3. Wakeup processor on RETI
Lower Power Savings

- Finally, powering your system with lower voltages means lower power consumption as well.

$$P = CV^2F$$
where
- $P$ = power consumption
- $C$ = load capacitance
- $V$ = operating voltage
- $F$ = frequency.
Principles of Low-Power Apps

- Maximize the time in low-power modes.
- Sleep as long as possible and use interrupts to wakeup.
- Use the slowest clock while still meeting processing needs.
- Switch on peripherals only when needed (i.e., switch off peripherals when not needed).
- Use low-power integrated computer peripherals.
  - Timers: Timer_A and Timer_B for PWM
  - A/D convertors, flash, LCD’s
- Use faster software algorithms / programming techniques
  - Calculated branches instead of flag polling.
  - Fast table look-ups instead of iterative calculations.
  - Use in-line code instead of frequent subroutine / function calls.
  - More single-cycle CPU register usage.
Quiz 9.2

1. Why is low-power usage an important issue?

2. What power mode is used by ISRs?

3. Name 3 ways to reduce power consumption.
   1. 
   2. 
   3. 

4. Approximately what percentage less power is consumed in a given time period by sleeping?
Timers
Timers

- System timing is fundamental for real-time applications
- The main applications of timers are to:
  - generate events of fixed time-period
  - allow periodic wakeup from sleep
  - count transitional signal edges
  - replace delay loops allowing the CPU to sleep between operations, consuming less power
  - maintain synchronization clocks
  - debounce mechanical devices
  - real-time clocks
  - control simulations
  - measure rates
  - Pulse Width Modulation
Timer_A/B

Timer_A is a 16-bit timer/counter with three capture/compare registers:

- Capture external signals
- Compare PWM mode
- SCCI latch for asynchronous communication
## Timer Registers

### Description

<table>
<thead>
<tr>
<th>Timer_B (Not available in F2013)</th>
<th>Description</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capture/compare register</td>
<td>TBCCR2</td>
<td>0x0196</td>
</tr>
<tr>
<td>Capture/compare register</td>
<td>TBCCR1</td>
<td>0x0194</td>
</tr>
<tr>
<td>Capture/compare register</td>
<td>TBCCR0</td>
<td>0x0192</td>
</tr>
<tr>
<td>Capture/compare control</td>
<td>TBR</td>
<td>0x0190</td>
</tr>
<tr>
<td>Capture/compare control</td>
<td>TBCCTL2</td>
<td>0x0186</td>
</tr>
<tr>
<td>Capture/compare control</td>
<td>TBCCTL1</td>
<td>0x0184</td>
</tr>
<tr>
<td>Capture/compare control</td>
<td>TBCCTL0</td>
<td>0x0182</td>
</tr>
<tr>
<td>Timer_B register</td>
<td>TBCTL</td>
<td>0x0180</td>
</tr>
<tr>
<td>Timer_B control</td>
<td>TBIV</td>
<td>0x011E</td>
</tr>
<tr>
<td>Timer_B interrupt vector</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timer_A</th>
<th>Description</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capture/compare register</td>
<td>TACCR2</td>
<td>0x0176</td>
</tr>
<tr>
<td>Capture/compare register</td>
<td>TACCR1</td>
<td>0x0174</td>
</tr>
<tr>
<td>Capture/compare register</td>
<td>TACCR0</td>
<td>0x0172</td>
</tr>
<tr>
<td>Capture/compare control</td>
<td>TAR</td>
<td>0x0170</td>
</tr>
<tr>
<td>Capture/compare control</td>
<td>TACCTL2</td>
<td>0x0166</td>
</tr>
<tr>
<td>Capture/compare control</td>
<td>TACCTL1</td>
<td>0x0164</td>
</tr>
<tr>
<td>Capture/compare control</td>
<td>TACCTL0</td>
<td>0x0162</td>
</tr>
<tr>
<td>Timer_A control</td>
<td>TACTL</td>
<td>0x0160</td>
</tr>
<tr>
<td>Timer_A interrupt vector</td>
<td>TAIIV</td>
<td>0x012E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Watchdog Timer+</th>
<th>Description</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>WD Control</td>
<td>WDTCTL</td>
<td>0x0120</td>
</tr>
</tbody>
</table>

### Diagram

- **Interrupt Vector Table**
  - Offset: 0xFFFF
  - Range: 0xFFFC0 - 0xFFC0

- **Program Code**
  - Offset: 0xFFBF
  - Range: 0xFC00 - 0xFFBF

- **Stack**
  - Offset: 0x03FF
  - Range: 0x0200 - 0x03FF

- **16-bit Peripherals Modules**
  - Offset: 0x01FF
  - Range: 0x0100 - 0x01FF

- **8-bit Peripherals Modules**
  - Offset: 0x00FF
  - Range: 0x0000 - 0x00FF

- **8-bit Special Function Registers**
  - Offset: 0x000F
  - Range: 0x0000 - 0x000F
### TXCTL Control Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9-8</td>
<td>Timer_x clock source:</td>
<td>0 0 ⇒ TxCLK, 0 1 ⇒ ACLK, 1 0 ⇒ SMCLK, 1 1 ⇒ INCLK</td>
</tr>
<tr>
<td>7-6</td>
<td>Clock signal divider:</td>
<td>0 0 ⇒ /1, 0 1 ⇒ /2, 1 0 ⇒ /4, 1 1 ⇒ /8</td>
</tr>
<tr>
<td>5-4</td>
<td>Clock timer operating mode:</td>
<td>0 0 ⇒ Stop mode, 0 1 ⇒ Up mode, 1 0 ⇒ Continuous mode, 1 1 ⇒ Up/down mode</td>
</tr>
<tr>
<td>2</td>
<td>Timer_x clear when TxCLR = 1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Timer_x interrupt enable when TxIE = 1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Timer_x interrupt pending when TxIFG = 1</td>
<td></td>
</tr>
</tbody>
</table>

(Used only by Timer_B)
Timer Modes

- **01 = Up Mode**
  - The timer repeatedly counts from 0x0000 to the value in the TxCCR0 register.

- **10 = Continuous Mode**
  - The timer repeatedly counts to 0xFFFF.

- **11 = Up/Down Mode**
  - The timer repeatedly counts from 0x0000 to the value in the TxCCR0 register and back down to zero.
Example 9.4 – Timer_A

Example 9.4 – Timer_A

```assembly
.cdecls C,"msp430.h" ; MSP430
TA_CTL       .equ TASSEL_2|ID_3|MC_1|TAIE ; 000000 10 11 01 000 1 = SMCLK,/8,UP,IE
TA_FREQ      .equ 0xffff                 ; clocks
STACK        .equ 0x0400                  ; top of stack

; Code Section -------------------------------------------------------------
.text                           ; beginning of executable code
start:      mov.w #STACK,SP               ; init stack pointer
            mov.w #WDTPW|WDTHOLD,&WDTCTL  ; stop WDT
            bis.b #0x01,&P1DIR            ; set p1.0 as output
            clr.w &TAR                    ; reset timerA
            mov.w #TA_CTL,&TACCR0        ; set interval (frequency)
            mov.w #TA_FREQ,&TACCR0       ; set interval (frequency)
            bis.w #LPM0|GIE,SR            ; enter LPM0 w/interrupts
error:      jmp $                       ; SHOULD NEVER GET HERE!!!!!!! !!!!

; Timer A ISR -------------------------------------------------------------
TA_isr:     bic.w #TAIFG,&TACTL           ; acknowledge interrupt
            xor.b #0x01,&P1OUT            ; toggle p1.0
            reti

; Interrupt Vectors -------------------------------------------------------
.sect   ".int08"                ; timer A section
.word   TA_isr                  ; timer A isr
.sect   ".reset"                ; MSP430 RESET Vector
.word   start                   ; start address
.end
```

**Blinky Example**

- **TASSEL_2 = SMCLK**
- **TA_CTL = SMCLK, /8, UP,IE**
- **MC_1 = UP Mode**
- **ID_3 = /8**
- **Enable Interrupt**
- **Put the processor to sleep!**
- **Never Execute!!!**
- **Timer A ISR**
Quiz 9.3

1. How could I speed up the blink?

2. What happens if the Timer_A ISR doesn’t acknowledge the interrupt?
Pulse Width Modulation
Pulse Width Modulation (PWM)

- PWM is a technique of digitally generating analog signals.

The average value of voltage (and current) is generated by switching the supply on and off at a fast pace.

Longer “ON” periods compared “OFF” periods results in higher power.

Longer “OFF” periods compared “ON” periods results in lower power.

Given a sufficient bandwidth, any analog value can be encoded with PWM.

The PWM signal is still digital because, at any given instant of time, the full DC supply is either fully on or fully off.
Examples of PWM Machines

Pulse Width Modulation
Pulse Width Modulation

PWM – Frequency/Duty Cycle

<table>
<thead>
<tr>
<th>Device</th>
<th>Frequency</th>
<th>Duty Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speaker</td>
<td>Pitch</td>
<td>Volume</td>
</tr>
<tr>
<td>LED</td>
<td>Flicker</td>
<td>Brightness</td>
</tr>
<tr>
<td>Motor</td>
<td>Speed</td>
<td>Speed</td>
</tr>
<tr>
<td>Heater</td>
<td>Steadiness</td>
<td>Heat</td>
</tr>
</tbody>
</table>
Example 9.5 – PWM w/Timer_A

```
.cdecls C,"msp430.h"

TA_CTL .equ TASSEL_2|ID_0|MC_1|TAIE ; SMCLK, /1, UP, IE
TA_FREQ .equ 120 ; FREQ / SMCLK = 0.0001 = 100 us
STACK .equ 0x0400 ; top of stack

.bss pwm_duty,2 ; PWM duty cycle counter
.bss pwm_cnt,2 ; PWM frequency counter

; Code Section -------------------------------------------------------------
.text
start: mov.w #STACK,SP ; init stack pointer
        mov.w #WDTPW|WDTHOLD,&WDTCTL ; stop watchdog
        bis.b #0x41,&P1DIR ; set P1.0,6 as output
        mov.b #0x40,&P1OUT
        clr.w &TAR ; reset timerA
        mov.w #TA_CTL,&TACTL ; set timerA control reg
        mov.w #TA_FREQ,&TACCR0 ; set interval (frequency)
        clr.w &pwm_duty ; init PWM counters
        clr.w &pwm_cnt

loop:  bis.w #LPM0|GIE,SR ; red -> green, enter LPM0 w/interrupts
        inc.w &pwm_duty ; increment &
        cmp.w #100,&pwm_duty ; 100% (full on)?
        jlo loop ; n

loop02: bis.w #LPM0|GIE,SR ; green -> red, enter LPM0 w/interrupts
        cmp.w &pwm_duty ; decrement % (full off)?
        jne loop02 ; n
        jmp loop ; y, repeat

; Timer A ISR ---------------------------------------------------------------
TA_isr: bic.w #TAIFG,&TACTL ; acknowledge interrupt
        cmp.w &pwm_duty,&pwm_cnt ; in duty cycle?
        jne TA_isr2 ; n
        xor.b #0x41,&P1OUT ; y, turn on LEDs

TA_isr2: inc.w &pwm_cnt ; increment %
        cmp.w #100,&pwm_cnt ; 100%?
        jlo TA_isr4 ; n
        mov.b #0x40,&P1OUT ; y, reset LEDs state
        clr.w &pwm_cnt ; clear counter
        bic.w #LPM0,0(SP) ; wakeup processor

TA_isr4: reti ; return from interrupt

; Interrupt Vectors --------------------------------------------------------
.int08 .word   TA_isr ; timerA isr
.reset ; PUC reset

; Interrupt Vectors --------------------------------------------------------

LED Intensity

Use Timer A ISR to PWM LEDs

Timer A Setup

Adjust duty cycle in main program. (Sleep during cycle)
A speaker or magnetic transducer consists of a iron core, wound coil, yoke plate, permanent magnet, and vibrating diaphragm with a movable iron piece.

A positive AC signal produces a fluctuating magnetic field, which causes the diaphragm to vibrate up and down, thus vibrating air.

Use PWM on P1.1/P1.2 to produce tones.
Example 9.6 – Watchdog PWM

```
Example 9.6 – Watchdog PWM

cdecls C,"msp430.h" ; include c header

WDT_CPS .equ 1200000/500 ; WD clocks / second count
STACK .equ 0x0400 ; stack

.bss WDTSecCnt,2 ; WDT second counter
.bss buzzON,1 ; buzzer on flag

; Code Section -------------------------------------------------------------
.text ; program section
.start: mov.w #STACK,SP ; initiali ze stack pointer
        mov.w #WDT_MDLY_0_5,&WDTCTL ; set WD timer interval to 0.5 ms
        mov.w #WDT_CPS,&WDTSecCnt ; initialize 1 sec WD counter
        mov.b #WDTIE,&IE1 ; enable WDT interrupt
        mov.b #0x07,&P1DIR ; P1.0 (LED) P1.1-2 (speaker)
        mov.b #0x04,&P1OUT ; set P1.1 & P1.2 to toggle
        clr.b buzzON ; turn off buzzer
        bis.w #LPM0|GIE,SR ; enable interrupts / sleep
        jmp $ ; (should never get here !)

; Watchdog ISR -------------------------------------------------------------
WDT_ISR: tst.b buzzON ; buzzer on?
         jeq WDT_02 ; n
         xor.b #0x06,&P1OUT ; y, use 50% PWM

WDT_02: dec.w &WDTSecCnt ; decrement counter
        jne WDT_04 ; y
        mov.w #WDT_CPS,&WDTSecCnt ; y, re-initialize
        xor.b #0x01,&P1OUT ; toggle led
        xor.b #0xff,buzzON ; toggle buzzer on

WDT_04: reti ; return from WDT

; sect ".int10"
.word WDT_ISR ; Watchdog ISR
.sect ".reset"
.word start ; RESET ISR
.end
```

- **PWM speaker (toggle P1.1/P1.2) when buzzON is non-zero (50% duty cycle)**
- **Toggle buzzON every second**
Summary

- By coding efficiently you can run multiple peripherals at high speeds on the MSP430
- Polling is to be avoided – use interrupts to deal with each peripheral only when attention is required
- Allocate processes to peripherals based on existing (fixed) interrupt priorities - certain peripherals can tolerate substantial latency
- Use GIE when it’s shown to be most efficient and the application can tolerate it – otherwise, control individual IE bits to minimize system interrupt latency.
- An interrupt-based approach eases the handling of asynchronous events
Event Driven Programming Model
Programming Paradigms

- **Imperative Programming**
  - computation in terms of statements that change a program state

- **Functional Programming**
  - computation as the evaluation of mathematical functions and avoids state and mutable data.

- **Procedural / Structured Programming**
  - specifying the steps the program must take to reach the desired state

- **Object Oriented Programming (OOP)**
  - uses "objects" – data structures consisting of datafields and methods together with their interactions – to design applications and computer programs.

- **Declarative Programming**
  - expresses the logic of a computation without describing its control flow

- **Automata-based Programming**
  - the program models a finite state machine or any other formal automata.

- **Event Driven Programming**
  - the flow of the program is determined by events, i.e., sensor outputs, user actions (mouse clicks, key presses), messages from other programs or threads.
Interrupt Events (asynchronous)

Event Handlers (synchronous)
Event Driven Programming

- System events
  - sensor outputs (completion interrupts)
  - internal generated events (timers)
  - user actions (mouse clicks, key presses)
  - messages from other programs or threads.

- Program has two sections:
  - event selection.
  - event handling.

- Main loop
  - constantly running main loop, or
  - sleep w/interrupts (preferred)
Example 9.7 – EDP Model

```
.cdecls C,"msp430.h"
TA_CTL    .equ TASSEL_2+ID_3+MC_1+TAIE ; SMCLK,/8,UP,IE
TA_FREQ   .equ 0xffff                  ; timerA frequency
WDT_CPS   .equ 1200000/32000          ; WDT clocks/second
STACK     .equ 0x0400                  ; tos
WDT_EVENT .equ 0x0001                  ; WDT event
TA_EVENT  .equ 0x0002                  ; timerA event

; Data Section -----------------------------------------------------
.bss WDTSecCnt,2             ; WDT second counter
.bss sys_event,2             ; system events
; Code Section -----------------------------------------------------
.text
start:      mov.w #STACK,SP               ; init stack pointer
            mov.w #WDT_MDLY_32,&WDTCTL    ; WDT interval
            mov.w #WDT_CPS,&WDTSecCnt ; WDT 1s counter
            bis.b #WDTIE,&IE1             ; enable WDT interrupt
            clr.w &TAR                    ; reset timerA
            mov.w #TA_CTL,&TACTL          ; timerA control register
            mov.w #TA_FREQ,&TACCR0        ; interval (frequency)
            bis.b #0x41,&P1DIR            ; set P1.0,6 as output
            clr.w &sys_event ; clear pending events

; Event Loop -------------------------------------------------------
loop:      bic.w #GIE,SR                 ; disable interrupts
            cmp.w #0,&sys_event           ; interrupt pending?
            jne loop02                  ; y
            bic.w #GIE|LPM0,SR            ; n, enable/sleep

loop02:    cmp.w #WDT_EVENT,&sys_event ; WDT event?
            jne loop04                  ; n
            bic.w #WDT_EVENT,&sys_event ; y, clear event
            xor.b #0x01,4&1OUT          ; toggle red LED

loop04:    cmp.w #TA_EVENT,&sys_event ; timerA event?
            jne loop06                  ; n
            bic.w #TA_EVENT,&sys_event ; y, clear event
            xor.b #0x40,4&1OUT          ; toggle green LED

loop06:    jmp loop ; loop indefinitely

; Event Driven Program
; Watchdog ISR ----------------------------------------------------
WDT_ISR:    dec.w &WDTSecCnt ; 1 second?
            jne WDT_02                  ; n
            mov.w #WDT_CPS,&WDTSecCnt ; y, reset counter
            bis.w #WDT_EVENT,&sys_event ; schedule WDT event
            bic.w #GIE|LPM0,0(SP)         ; wakeup processor
            reti ; exit ISR

WDT_02:     reti ; exit ISR

; Timer A ISR -----------------------------------------------------
TA_isr:     bic.w #TAIFG,&TACTL           ; ack interrupt
            bis.w #TA_EVENT,&sys_event ; schedule timerA event
            bic.w #GIE|LPM0,0(SP)         ; wakeup processor
            reti

; Interrupt Vectors -----------------------------------------------
.int08 : .sect   " .int08 "                ; timerA section .word   TA_isr ; timerA isr .sect   " .int10 "                ; Watchdog .sect   " .reset "                ; PUC Vector .word   start                   ; RESET ISR .end
```

Event Driven Program

**Event Loop**
- Disable interrupts
- Check for WDT event
- Clear WDT event
- Toggle red LED

**Watchdog Event**
- Decrement WDT second counter
- Clear WDT event
- Schedule WDT event
- Wakeup processor
- Exit ISR

**TimerA Event**
- Clear TimerA event
- Schedule TimerA event
- Wakeup processor
- Exit ISR

**Activity Profile**
- **Active** (300μA)
- **Sleep** (1μA)

**300μA**
- Active

**1μA**
- Sleep

Average: **0.6 μA**