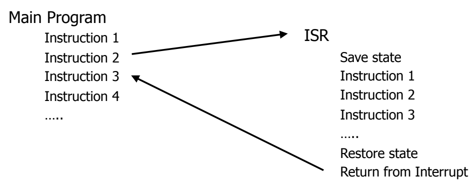
MSP430 Interrupts

Interrupts

- Fundamental concept in computation
- Interrupt execution of a program to “handle” an event
  - Don’t have to rely on program relinquishing control
  - Can code program without worrying about others
- Issues
  - What can interrupt and when?
  - Where is the code that knows what to do?
  - How long does it take to handle interruption?
  - Can an interruption be, in turn, interrupted?
  - How does the interrupt handling code communicate its results?
  - How is data shared between interrupt handlers and programs?

What is an Interrupt?

- Reaction to something in I/O (human, comm link)
- Usually asynchronous to processor activities
- “interrupt handler” or “interrupt service routine” (ISR) invoked to take care of condition causing interrupt
  - Change value of internal variable (count)
  - Read a data value (sensor, receive)
  - Write a data value (actuator, send)



Interrupts

- Code sample that does not interrupt

```
char SPI_SlaveReceive(void)
{
/* Wait for reception complete */
while(!(SPSR & (1<<SPIF)))
;
/* Return data register */
return SPDR;
}
```

- Instead of busy waiting until a byte is received the processor can generate an interrupt when it sets SPIF

```
SIGNAL(SIG_SPI) {
    RX_Byte = SPDR
}
```

Saving and Restoring Context

- Processor and compiler dependent
- Where to find ISR code?
  - Different interrupts have separate ISRs
- Who does dispatching?
  - Direct
    - Different address for each interrupt type
    - Supported directly by processor architecture
  - Indirect
    - One top-level ISR
    - Switch statement on interrupt type
  - A mix of these two extremes?

Saving and Restoring Context

- How much context to save?
  - Registers, flags, program counter, etc.
  - Save all or part?
  - Agreement needed between ISR and program
- Where should it be saved?
  - Stack, special memory locations, shadow registers, etc.
  - How much room will be needed on the stack?
  - Nested interrupts may make stack reach its limit – what then?
- Restore context when ISR completes

Ignoring Interrupts

- Can interrupts be ignored?
  - It depends on the cause of the interrupt
  - No, for nuclear power plant temperature warning
  - Yes, for keypad on cell phone (human timescale is long)
- When servicing another interrupt
  - Ignore others until done
  - Can't take too long – keep ISRs as short as possible
    - Just do a quick count, or read, or write – not a long computation
- Interrupt disabling
  - Will ignored interrupt "stick"?
    - Rising edge sets a flip-flop
  - Or will it be gone when you get to it?
    - Level changes again and its as if it never happened
  - Don't forget to re-enable

Prioritizing Interrupts

- When multiple interrupts happen simultaneously
  - Which is serviced first?
  - Fixed or flexible priority?
- Priority interrupts
  - Higher priority can interrupt
  - Lower priority can't
- Maskable interrupts
  - "don't bother me with that right now"
  - Not all interrupts are maskable, some are non-maskable

Interrupts in the MSP430

- External interrupts
  - From I/O pins of microcontroller
- Internal interrupts
  - Timers
    - Output compare
    - Input capture
    - Overflow
  - Communication units
    - Receiving something
    - Done sending
  - ADC
    - Completed conversion

Chain of Events on Interrupt

- Finish executing current instruction
- Disable all interrupts

Automatic

- Push program counter on to stack
- Jump to interrupt vector table
- Jump to start of complete ISR

Compiler

- Save any context that ISR may otherwise change
  - Registers and flags must be saved within ISR and restored before it returns – **<u>this is very important!</u>**
- Re-enable interrupts if nested interrupts are ok
- Complete ISR's code
- Re-enable interrupts upon return
- Jump back to next instruction before interruption

RETI

Shared Data Problem

- When you use interrupts you create the opportunity for multiple sections of code to update a variable.
- This might cause a problems in your logic if an interrupt updates a variable between two lines of code that are directly dependent on each other (e.g. if statement)
- One solution is to create critical sections where you disable the interrupts for a short period of time while you complete your logic on the shared variable

```
clear (GIE) bit ;
…..critical section code goes here…..
set (GIE) bit ;
```

Interrupts

- Execution of a program proceeds predictably, with *interrupts* being the exception
- Interrupts are usually generated by hardware
  - Processor stops what it is doing,
  - Stores enough information to later resume,
  - Executes an *interrupt service routine* (ISR),
  - Restores saved information,
  - Resumes execution.
- An interrupt is an asynchronous signal indicating the need for attention

Interrupts

- Interrupts *preempt* normal code execution
  - Interrupt code runs in the *foreground*
  - Normal (e.g. `main()`) code runs in the *background*
- Interrupts can be *enabled* and *disabled*
  - *Globally*
  - *Individually* on a per-peripheral basis
  - *Non-Maskable* Interrupt (NMI)
- The occurrence of each interrupt is *unpredictable*
  - *When* an interrupt occurs
  - *Where* an interrupt occurs
- Interrupts are associated with a variety of on-chip and off-chip peripherals.
  - Timers, Watchdog, D/A, Accelerometer
  - NMI, change-on-pin (Switch)

Interrupts

- Interrupts commonly used for
  - Urgent tasks w/higher priority than main code
  - Infrequent tasks to save polling overhead
  - Waking the CPU from sleep
  - Call to an operating system (software interrupt).
- Event-driven programming
  - The flow of the program is determined by events—i.e., sensor outputs or user actions (mouse clicks, key presses) or messages from other programs or threads.
  - The application has a main loop with event detection and event handlers.

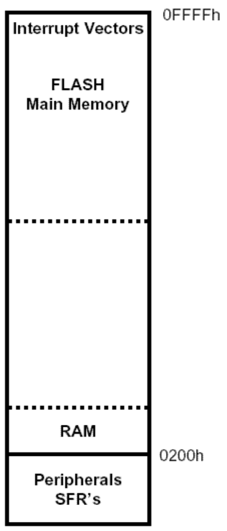
Interrupt Flags

- Each interrupt has a flag that is raised (set) when the interrupt occurs.
- 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) enabled and
  2) the general interrupt enable (GIE) bit is set in the status register (SR).

Interrupt Vectors

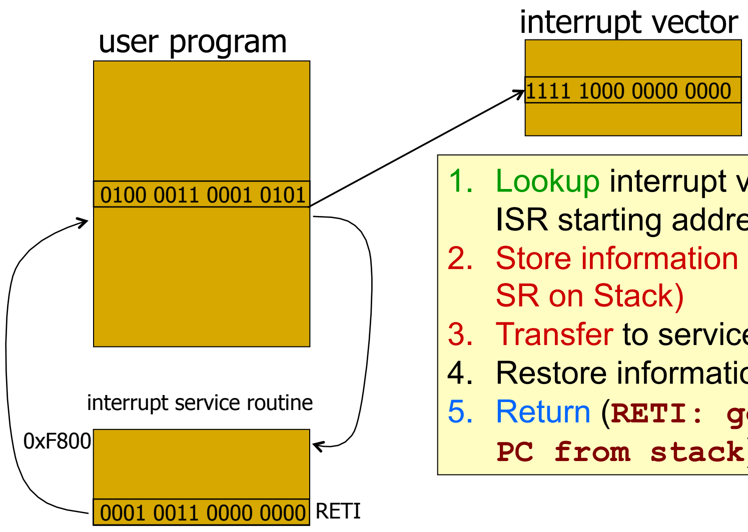
- The CPU must know where to fetch the next instruction following an interrupt.
- The address of an ISR is defined in an *interrupt vector*.
- The MSP430 uses *vectored interrupts* where each ISR has its own vector stored in a *vector table* located at the end of program memory.
- Note: The *vector table*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

MSP430 Memory



- Unified 64KB continuous memory map
- Same instructions for data and peripherals
- Program and data in Flash or RAM with no restrictions
- Designed for modern programming techniques such as pointers and fast look-up tables

Serving Interrupt Request



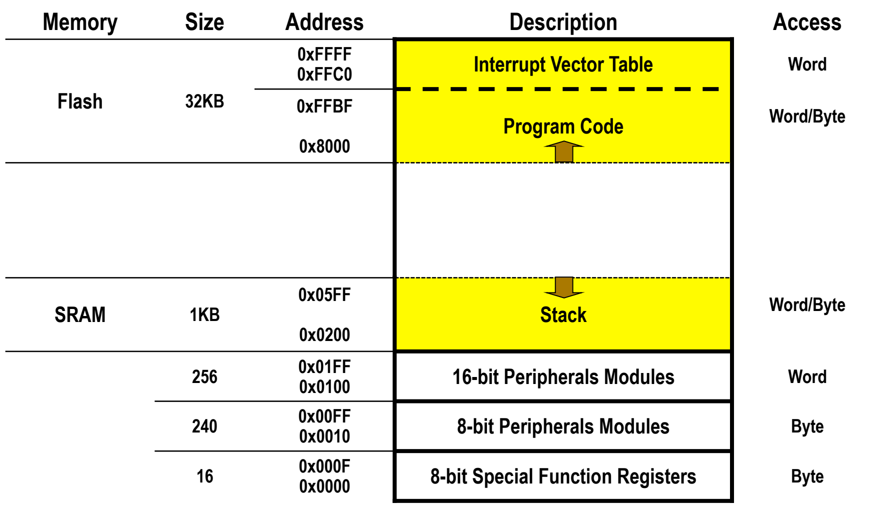
1. Lookup interrupt vector for ISR starting address.
2. Store information (PC and SR on Stack)
3. Transfer to service routine.
4. Restore information
5. Return (**`RETI: get old PC from stack`**).

MSP430x2xx Interrupt Vectors

Higher address = higher priority

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	ADDRESS	SECTION	PRIORITY
Power-up External reset Watchdog	PORIFG RSTIFG WDTIFG	Reset	0xFFFE	.reset	15, highest
NMI Oscillator fault Flash memory violation	NMIIFG OFIFG ACCDVIFG	Non-maskable	0xFFFC	.int14	14
Timer_B3	TBCCR0 CCIFG	Maskable	0xFFFA	.int13	13
Timer_B3	TBCCR1 CCIFG TBCCR2 CCIFG, TBIFG	Maskable	0xFFF8	.int12	12
			0xFFF6	.int11	11
Watchdog Timer	WDTIFG	Maskable	0xFFF4	.int10	10
Timer_A3	TACCR0 CCIFG	Maskable	0xFFF2	.int09	9
Timer_A3	TACCR1 CCIFG, TACCR2 CCIFG, TAIFG	Maskable	0xFFF0	.int08	8
USCI_A0/USCI_B0 Rx	UCA0RXIFG, USB0RXIFG	Maskable	0xFFEE	.int07	7
USCI_Z0/USCI_B0 Tx	UCA0TXIFG, UCB0TXIFG	Maskable	0xFFEC	.int06	6
ADC10	ADC10IFG	Maskable	0xFFEA	.int05	5
			0xFFE8	.int04	4
I/O Port P2	P2IFG.0 – P2IFG.7	Maskable	0xFFE6	.int03	3
I/O Port P1	P1IFG.0 – P1IFG.7	Maskable	0xFFE4	.int02	2
			0xFFE2	.int01	1
			0xFFE0	.int00	0

MSP430F2274 Address Space

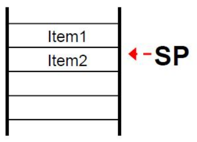


Memory	Size	Address	Description	Access
Flash	32KB	0xFFFF 0xFFC0	Interrupt Vector Table	Word
		0xFFBF 0x8000	Program Code	Word/Byte
SRAM	1KB	0x05FF 0x0200	Stack	Word/Byte
	256	0x01FF 0x0100	16-bit Peripherals Modules	Word
	240	0x00FF 0x0010	8-bit Peripherals Modules	Byte
	16	0x000F 0x0000	8-bit Special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

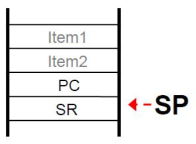
Processing an Interrupt…

1) Current instruction completed
2) MCLK started if CPU was off
3) Processor pushes program counter on stack
4) Processor pushes status register 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!

Interrupt Stack

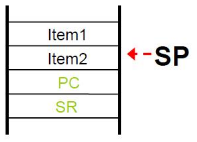


Prior to Interrupt Service Routine (=ISR)



ISR hardware - automatically

- Program Counter (= PC) pushed
- Status Register (= SR) pushed
- Interrupt vector moved to PC
- **GIE, CPUOFF, OSCOFF and SCG1 cleared**
- IFG flag cleared on single source flags



reti - automatically

- SR popped - *original*
- PC popped

Interrupt Service Routines

- Look superficially like a subroutine.
- However, unlike subroutines
  - ISR’s can execute at unpredictable times.
  - Must carry out action and thoroughly clean up.
  - Must be concerned with shared variables.
  - Must return using ***reti*** rather than ***ret***.
- ISR must handle interrupt in such a way that the interrupted code can be resumed without error
  - Copies of all registers used in the ISR must be saved (preferably on the stack)

Interrupt Service Routines

- Well-written ISRs:
  - Should be *short* and *fast*
  - Should affect the rest of the system *as little as possible*
  - 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

Interrupt Service Routines

- Interrupt-related runtime problems can be exceptionally hard to debug
- Common interrupt-related errors include:
  - Failing to *protect global variables*
  - Forgetting to actually *include the ISR* - no linker error!
  - Not testing or validating thoroughly
  - *Stack overflow*
  - Running out of *CPU horsepower*
  - Interrupting critical code
  - Trying to *outsmart the compiler*

Returning from ISR

- MSP430 requires 6 clock cycles before the ISR begins executing
  - The time between the interrupt request and the start of the ISR is called ***latency (plus time to complete the current instruction, 6 cycles, the worst case)***
- An ISR always finishes with the return from interrupt instruction (***reti***) requiring 5 cycles
  - The SR is popped from the stack
    - Re-enables maskable interrupts
    - Restores previous low-power mode of operation
  - The PC is popped from the stack
  - Note: if waking up the processor with an ISR, the new power mode must be set in the stack saved SR

Return From Interrupt

- Single operand instructions:

Mnemonic	Operation	Description
`PUSH`(.B or .W) `src`	SP-2→SP, src→@SP	Push byte/word source on stack
`CALL` `dst`	SP-2→SP, PC+2→@SP dst→PC	Subroutine call to destination
`RETI`	TOS→SR, SP+2→SP TOS→PC, SP+2→SP	Return from interrupt

- Emulated instructions:

Mnemonic	Operation	Emulation	Description
`RET`	@SP→PC SP+2→SP	MOV @SP+,PC	Return from subroutine
`POP`(.B or .W) `dst`	@SP→temp SP+2→SP temp→dst	MOV(.B or .W) @SP +,dst	Pop byte/word from stack to destination

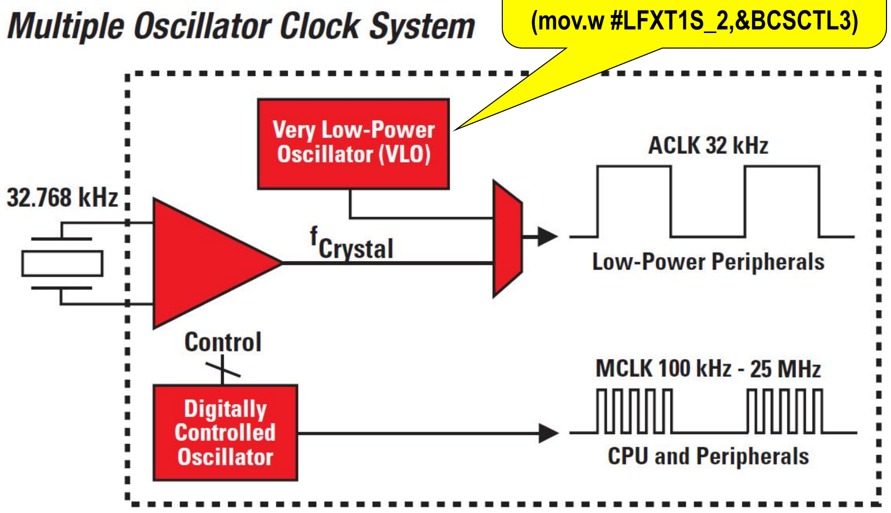
Summary

- 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

```
//  MSP430F20x3 Demo - SD16A, Sample A1+ Continuously, Set P1.0 if > 0.3V
#include  <msp430x20x3.h>

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                   // Stop watchdog timer
  P1DIR |= 0x01;                              // Set P1.0 to output direction
  SD16CTL = SD16REFON + SD16SSEL_1;           // 1.2V ref, SMCLK
  SD16INCTL0 = SD16INCH_1;                    // A1+/-
  SD16CCTL0 =  SD16UNI + SD16IE;              // 256OSR, unipolar, interrupt enable
  SD16AE = SD16AE2;                           // P1.1 A1+, A1- = VSS
  SD16CCTL0 |= SD16SC;                        // Set bit to start conversion

  _BIS_SR(LPM0_bits + GIE);
}

#pragma vector = SD16_VECTOR
__interrupt void SD16ISR(void)
{
  if (SD16MEM0 < 0x7FFF)                      // SD16MEM0 > 0.3V?, clears IFG
    P1OUT &= ~0x01;
  else
    P1OUT |= 0x01;
}
```

Multiple Clocks

No crystal on eZ430 tools
Use VLO for ACLK
(mov.w #LFXT1S_2,&BCSCTL3)



Processor Clock Speeds

- Often, the most important factor for reducing power consumption is slowing the clock down
  - Faster clock = Higher performance, more power
  - Slower clock = Lower performance, less power
- Using assembly code:

```
;   MSP430 Clock - Set DCO to 8 MHz:
    mov.b   #CALBC1_8MHZ,&BCSCTL1    ; Set range
    mov.b   #CALDCO_8MHZ,&DCOCTL     ; Set DCO step + modulation
```

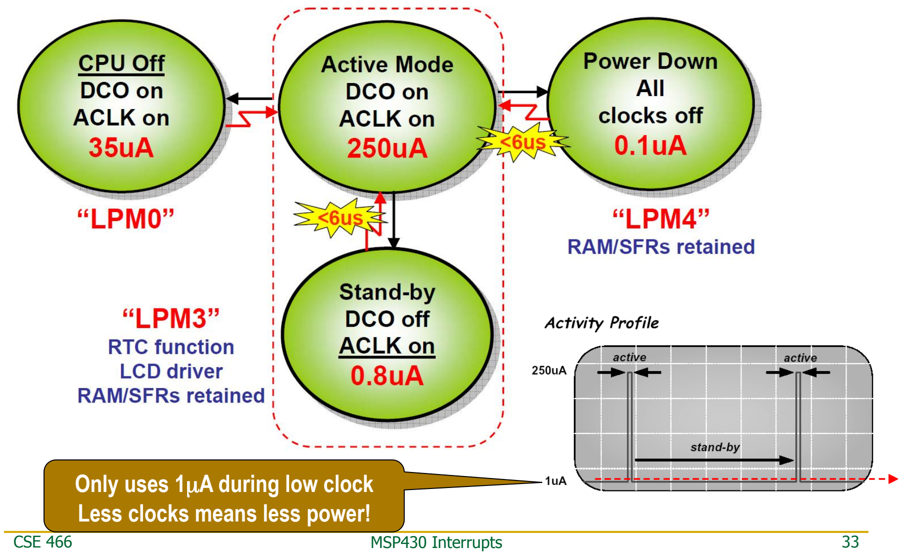
- Using C code:

```
//  MSP430 Clock - Set DCO to 8 MHz:
    BCSCTL1 = CALBC1_8MHZ;           // Set range 8MHz
    DCOCTL = CALDCO_8MHZ;            // Set DCO step + modulation
```

Processor Clock Speeds

- Another method to reduce power consumption is to turn off some (or all) of the system clocks
  - Active Mode (AM): CPU, all clocks, and enabled modules are active (≈300 μA)
  - LPM0: CPU and MCLK are disabled, SMCLK and ACLK remain active (≈85 μA)
  - LPM3: CPU, MCLK, SMCLK, and DCO are disabled; only ACLK remains active (≈1 μA)
  - LPM4: CPU and all clocks disabled, RAM is retained (≈0.1 μA)
- A device is said to be ***sleeping*** when in low-power mode; ***waking*** refers to returning to active mode

MSP430 Clock Modes



Clocks Off Power Savings

R2/SR	Reserved	V	SCG1	SCG0	OSC OFF	CPU OFF	GIE	N	Z	C
Active Mode			0	0	0	0	~ 250uA			
LPM0			0	0	0	1	~ 35uA			
LPM3			1	1	0	1	~ 0.8uA			
LPM4			1	1	1	1	~ 0.1uA			

SMCLK and ACLK Active

Only ACLK Active

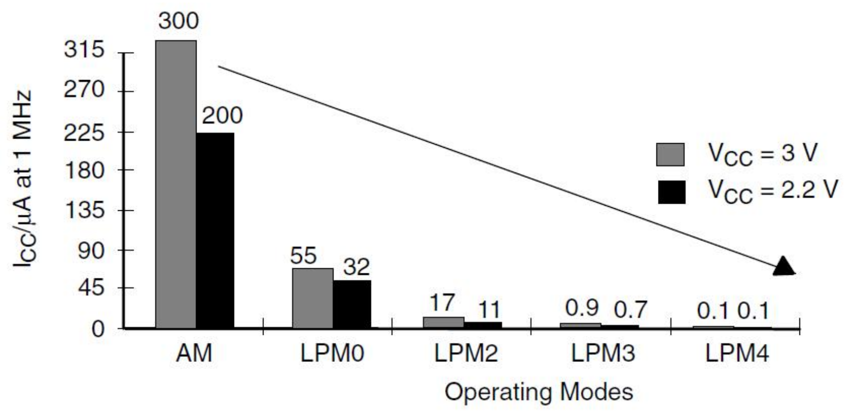
No Clocks!

Sleep Modes

```
bis.w    #CPUOFF,SR           ; LPM0
```

Lower Power Savings

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



Principles of Low-Power Apps

- Maximize the time in LPM3 mode
- Use interrupts to wake the processor
- Switch on peripherals only when needed
- Use low-power integrated peripherals
  - Timer_A and Timer_B for PWM
- Calculated branches instead of flag polling
- Fast table look-ups instead of calculations
- Avoid frequent subroutine and function calls
- Longer software routines should use single-cycle CPU registers

Setting Low-Power Modes

- Setting low-power mode puts the microcontroller "to sleep" – so usually, interrupts would need to be enabled as well.
- Enter LPM3 and enable interrupts using assembly code:

```
;   enable interrupts / enter low-power mode 3
    bis.b   #LPM3+GIE,SR        ; LPM3 w/interrupts
```

- Enter LPM3 and enable interrupts using C code:

```
//  enable interrupts / enter low-power mode 3
    __bis_SR_register(LPM3_bits + GIE);
```

Timers

- System timing is fundamental for real-time applications
- The MSP430F2274 has 2 timers, namely Timer_A and Timer_B
- The timers may be triggered by internal or external clocks
- Timer_A and Timer_B also include multiple independent capture/compare blocks that are used for applications such as timed events and Pulse Width Modulation (PWM)

Timers

- The main applications of timers are to:
  - generate events of fixed time-period
  - allow periodic wakeup from sleep of the device
  - count transitional signal edges
  - replace delay loops allowing the CPU to sleep between operations, consuming less power
  - maintain synchronization clocks

TxCTL Control Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
(Used by Timer_B)						TxSSELx		IDx		MCx		-	TxCLR	TxIE	TxIFG

Bit		Description
9-8	TxSSELx	Timer_x clock source: 0 0 $\Rightarrow$ TxCLK 0 1 $\Rightarrow$ ACLK 1 0 $\Rightarrow$ SMCLK 1 1 $\Rightarrow$ INCLK
7-6	IDx	Clock signal divider: 0 0 $\Rightarrow$ / 1 0 1 $\Rightarrow$ / 2 1 0 $\Rightarrow$ / 4 1 1 $\Rightarrow$ / 8
5-4	MCx	Clock timer operating mode: 0 0 $\Rightarrow$ Stop mode 0 1 $\Rightarrow$ Up mode 1 0 $\Rightarrow$ Continuous mode 1 1 $\Rightarrow$ Up/down mode
2	TxCLR	Timer_x clear when TxCLR = 1
1	TxIE	Timer_x interrupt enable when TxIE = 1
0	TxIFG	Timer_x interrupt pending when TxIFG = 1

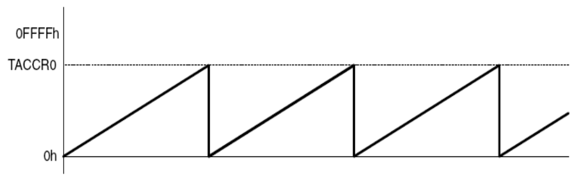
4 Modes of Operation

- Timer reset by writing a 0 to TxR
- Clock timer operating modes:

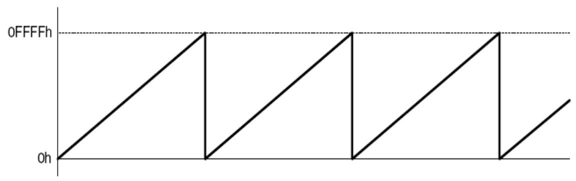
MCx	Mode	Description
0 0	Stop	The timer is halted.
0 1	Up	The timer repeatedly counts from 0x0000 to the value in the TxCCR0 register.
1 0	Continuous	The timer repeatedly counts from 0x0000 to 0xFFFF.
1 1	Up/down	The timer repeatedly counts from 0x0000 to the value in the TxCCR0 register and back down to zero.

Timer Modes

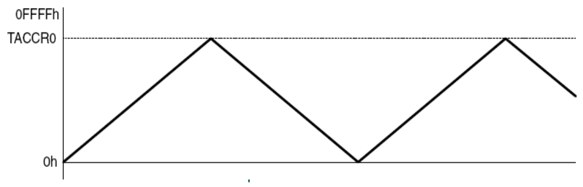
- Up Mode



- Continuous Mode



- Up/Down Mode



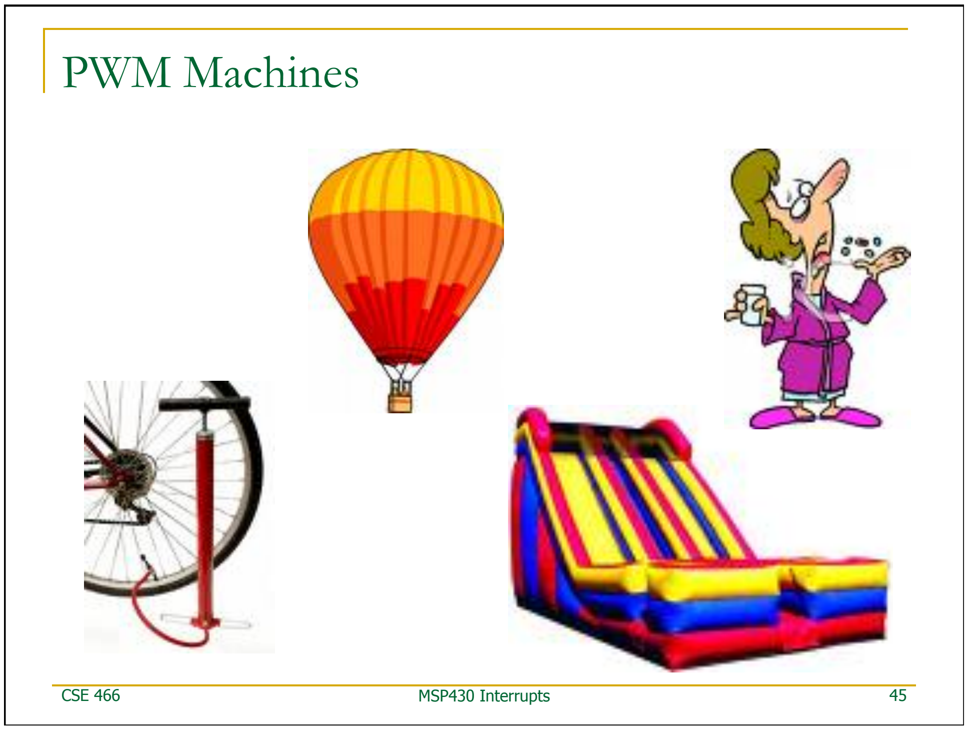
Timer_A Example

- Use Timer A to interrupt every 1 ms

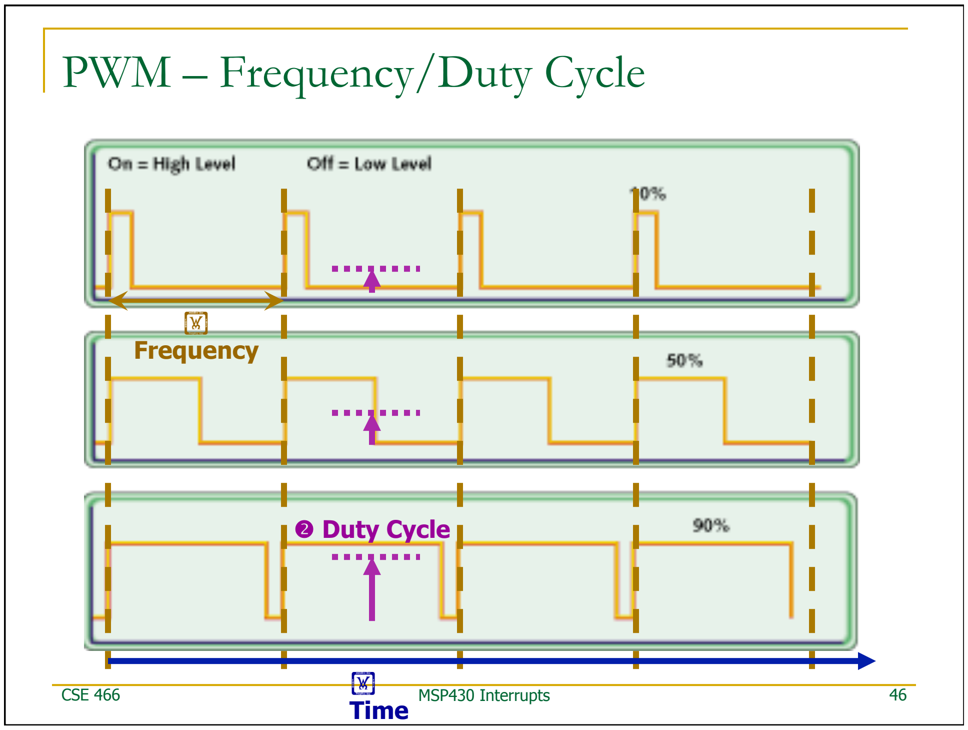
```
SMCLK    .set  1200000         ; 1200000 clocks / second
TIME_1MS .set  1000            ; 1 ms = 1/1000 s

TA_CTL   .set  TASSEL_2+ID_0+MC_1+TAIE ; SMCLK, /1, UP, IE
TA_FREQ  .set  SMCLK/TIME_1MS  ; clocks / 1 ms

    clr.w   &TAR               ; reset timerA
    mov.w   #TA_CTL,&TACTL     ; set timerA control reg
    mov.w   #TA_FREQ,&TACCR0   ; set interval (frequency)
    bis.w   #LPM0+GIE,SR       ; enter LPM0 w/interrupts
    jmp     $                  ; will never get here! (CPU is off

 TA_isr:                        ; timer A ISR
     bic.w   #TAIFG,&TACTL      ; acknowledge interrupt
 ;   <<add interrupt code here>>
     reti

     .sect   ".int08"           ; timer A section
     .word   TA_isr             ; timer A isr
```

Pulse Width Modulation (PWM)

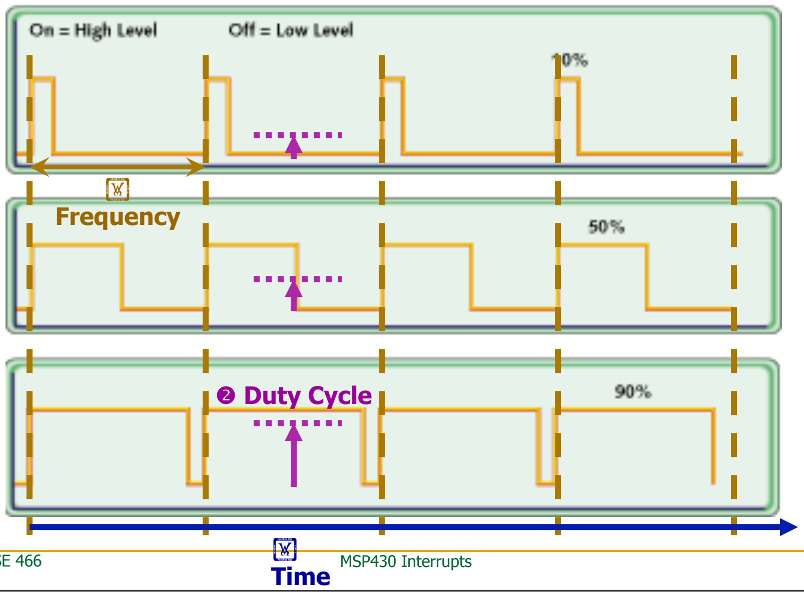
- Pulse width modulation (PWM) is used to control analog circuits with a processor's digital outputs
- PWM is a technique of digitally encoding analog signal levels
  - The duty cycle of a square wave is modulated to encode a specific analog signal level
  - The PWM signal is still digital because, at any given instant of time, the full DC supply is either fully on or fully off
- The voltage or current source is supplied to the analog load by means of a repeating series of on and off pulses
- Given a sufficient bandwidth, any analog value can be encoded with PWM.



PWM Machines



PWM – Frequency/Duty Cycle



Watchdog Timer

- The primary function of the watchdog timer+ (WDT+) module is to perform a controlled system restart after a software problem occurs.
- If the selected time interval expires, a 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.

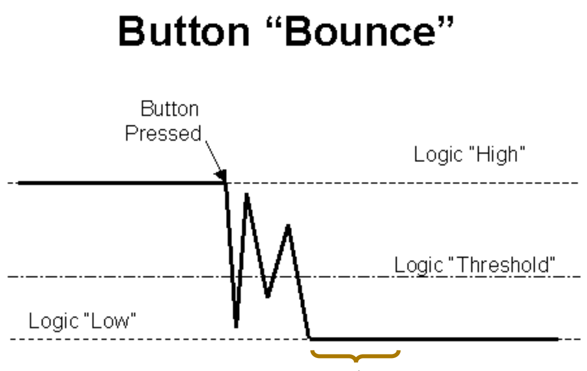
Watchdog Timer

- Features of the watchdog timer+ module include:
  - Four software-selectable time intervals
  - Watchdog mode
  - Interval mode
  - Access to WDT+ control register is password protected
  - Control of RST/NMI pin function
  - Selectable clock source
  - Can be stopped to conserve power
  - Clock fail-safe feature

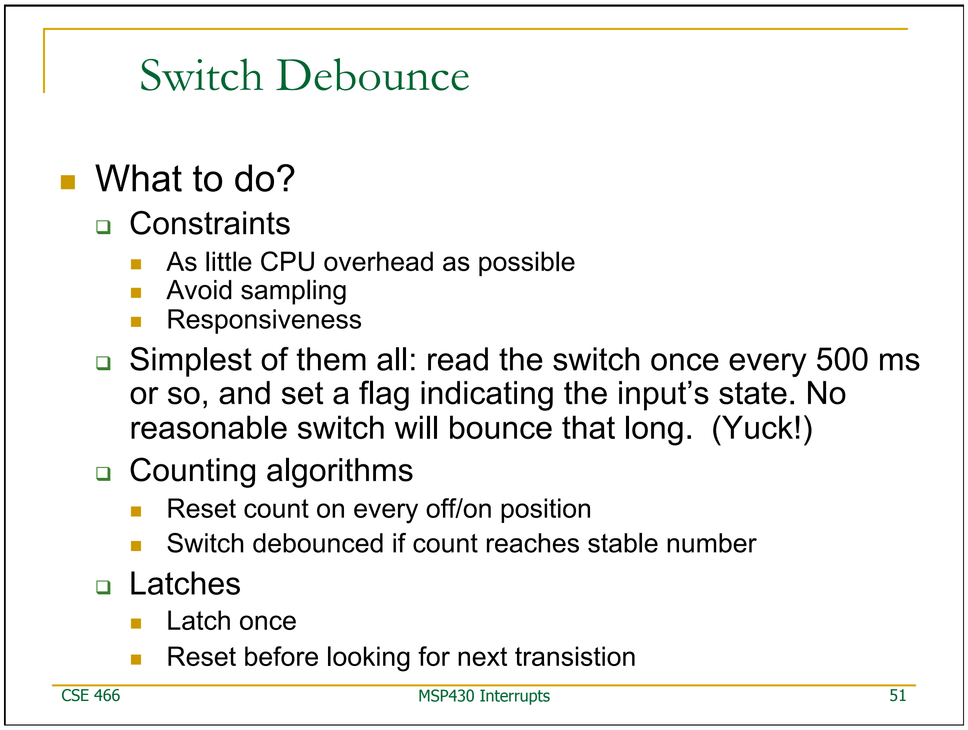
Watchdog Power-up

- After a power-up cycle (PUC), the WDT+ module is automatically configured in the 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.

Switch Debounce



Switch "debounded" after signal remains low/high for specified time



Switch Debounce

- What to do?
  - Constraints
    - As little CPU overhead as possible
    - Avoid sampling
    - Responsiveness
  - Simplest of them all: read the switch once every 500 ms or so, and set a flag indicating the input's state. No reasonable switch will bounce that long. (Yuck!)
  - Counting algorithms
    - Reset count on every off/on position
    - Switch debounced if count reaches stable number
  - Latches
    - Latch once
    - Reset before looking for next transistion