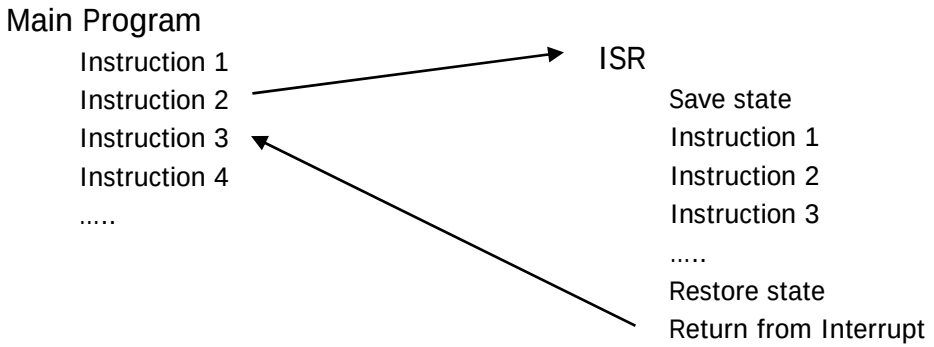
MSP430 Interrupts

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

- 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).

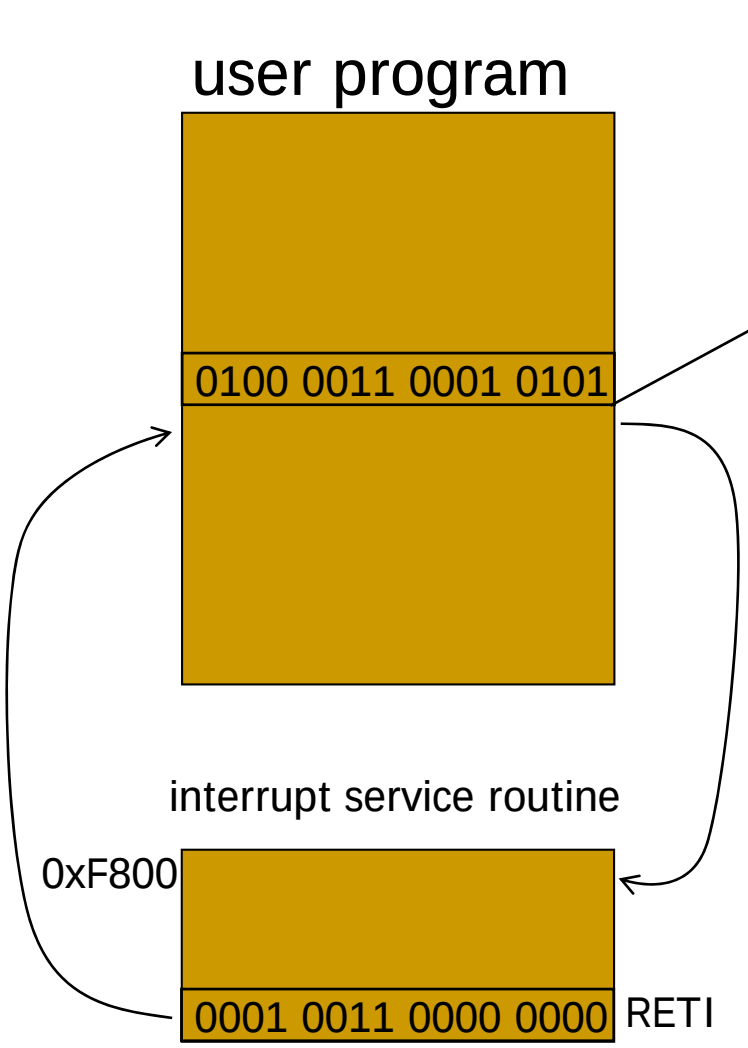
Interrupt Vectors

- 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

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

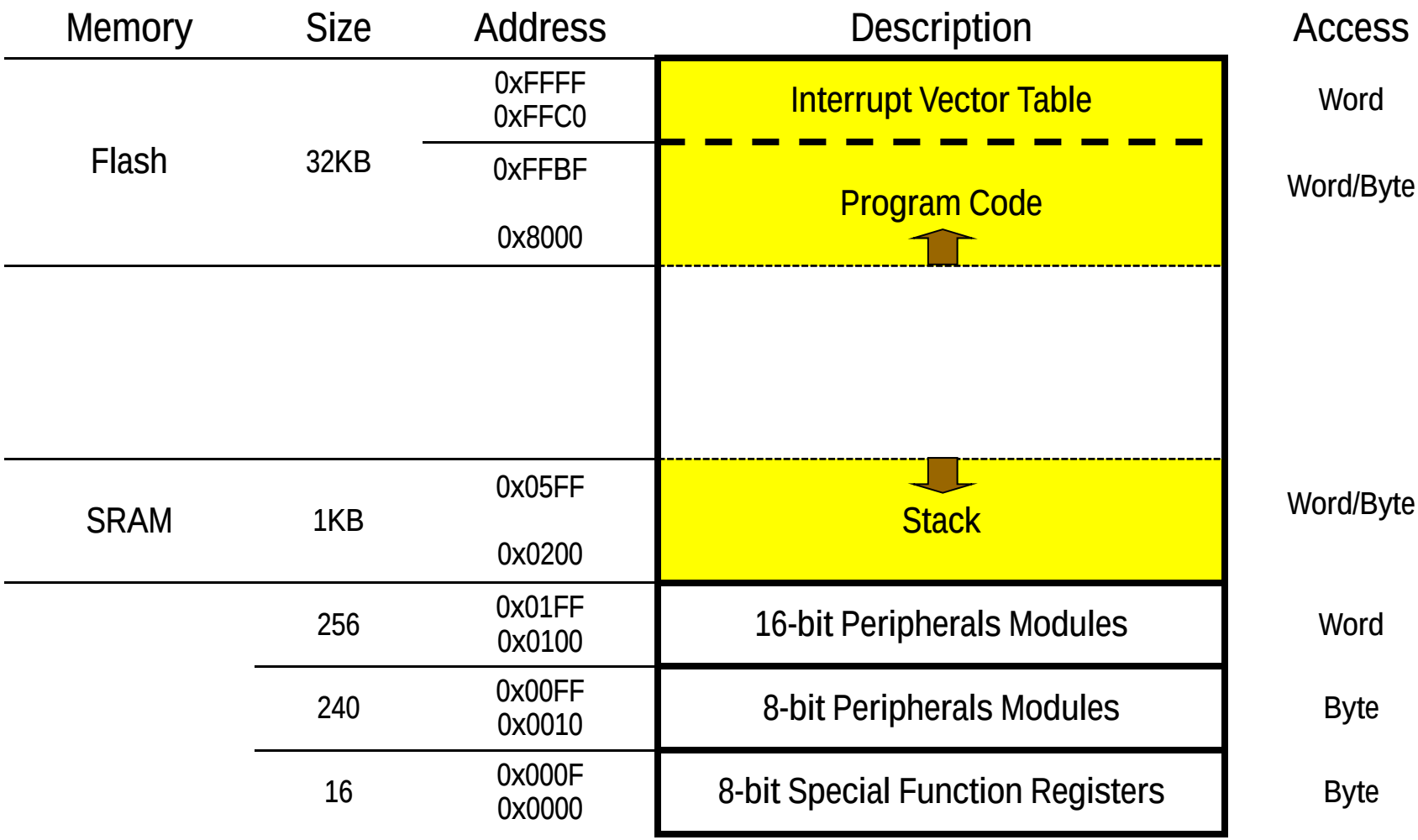
Higher address➔
higher priority

Table 7. Interrupt Sources

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-up External reset Watchdog Timer+ Flash key violation PC out-of-range(1)	PORIFG RSTIFG WDTIFG KEYV See (2)	Reset	0FFFEh	31, highest
NMI Oscillator fault Flash memory access violation	NMIIFG OFIFG ACCVIFG(2)(3)	(non)-maskable, (non)-maskable, (non)-maskable	0FFFCh	30
			0FFFAh	29
			0FFF8h	28
Comparator_A+ (MSP430F20x1)	CAIFG(4)	maskable	0FFF6h	27
Watchdog Timer+	WDTIFG	maskable	0FFF4h	26
Timer_A2	TACCR0 CCIFG(4)	maskable	0FFF2h	25
Timer_A2	TACCR1 CCIFG.TAIFG(2)(4)	maskable	0FFF0h	24
			0FFEEh	23
			0FFECh	22
ADC10 (MSP430F20x2)	ADC10IFG(4)	maskable	0FFEAh	21
SD16_A (MSP430F20x3)	SD16CCTL0 SD16OVIFG SD16CCTL0 SD16IFG(2)(4)	maskable		
USI (MSP430F20x2, MSP430F20x3)	USIIFG, USISTTIFG(2)(4)	maskable	0FFE8h	20
I/O Port P2 (two flags)	P2IFG.6 to P2IFG.7(2)(4)	maskable	0FFE6h	19
I/O Port P1 (eight flags)	P1IFG.0 to P1IFG.7(2)(4)	maskable	0FFE4h	18
			0FFE2h	17
			0FFE0h	16
See (5)			0FFDEh to 0FFC0h	15 to 0, lowest

(1) A reset is generated if the CPU tries to fetch instructions from within the module register memory address range (0h to 01FFh) or from within unused address ranges.
(2) Multiple source flags
(3) (non)-maskable: the individual interrupt-enable bit can disable an interrupt event, but the general interrupt enable cannot.
(4) Interrupt flags are located in the module.
(5) The interrupt vectors at addresses 0FFDEh to 0FFC0h are not used in this device and can be used for regular program code if necessary.

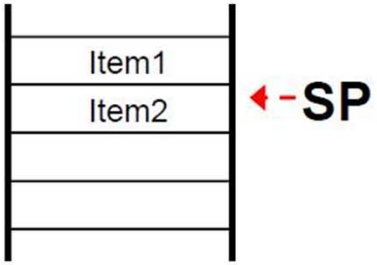
MSP430F2274 Address Space



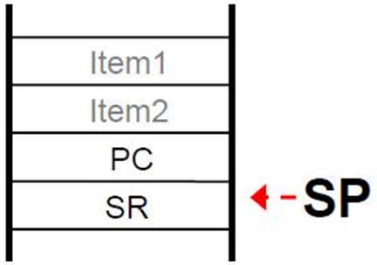
Processing an Interrupt…

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

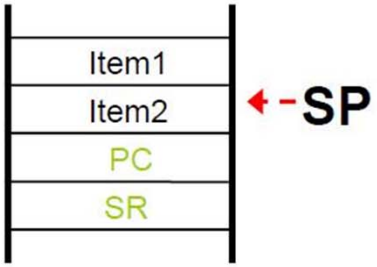


Prior to Interrupt Service Routine (=ISR)



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

P1 and P2 interrupts

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

Example P1 interrupt msp430x20x3_P1_02.c

```
#include  <msp430x20x3.h>
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;   // Stop watchdog timer
  P1DIR |= 0x01;              // Set P1.0 to output direction
  P1IE |= 0x10;               // P1.4 interrupt enabled
  P1IES |= 0x10;              // P1.4 Hi/lo edge
  P1IFG &= ~0x10;             // P1.4 IFG cleared

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

Ex: Timer interrupt: msp430x20x3_ta_03.c

```
#include <msp430x20x3.h>
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT
  P1DIR |= 0x01;                            // P1.0 output
  TACTL = TASSEL_2 + MC_2 + TAIE;           // SMCLK, contmode, interrupt

  _BIS_SR(LPM0_bits + GIE);                 // Enter LPM0 w/ interrupt
}
// Timer_A3 Interrupt Vector (TAIV) handler
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A(void)
{
 switch( TAIV )
 {
   case  2: break;                          // CCR1 not used
   case  4: break;                          // CCR2 not used
   case 10: P1OUT ^= 0x01;                  // overflow
            break;
 }
}
```

Example

we stepped through the following code in class with the debugger

Msp430x20x3_ta_06.c (modified, part 1)

Demo: Samples 8

```
#include  <msp430x20x3.h>
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;// Stop WDT
  P1DIR |= 0x01;           // P1.0 output
  CCTL1 = CCIE;            // CCR1 interrupt enabled
  CCR1 = 0xA000;
  TACTL = TASSEL_2 + MC_2; // SMCLK, Contmode
  _BIS_SR(LPM0_bits + GIE);// Enter LPM0 w/ int.
}
```

Servicing a timer interrupt; toggling pin in ISR

Msp430x20x3_ta_06.c (modified, part 2)

Demo: Samples 8

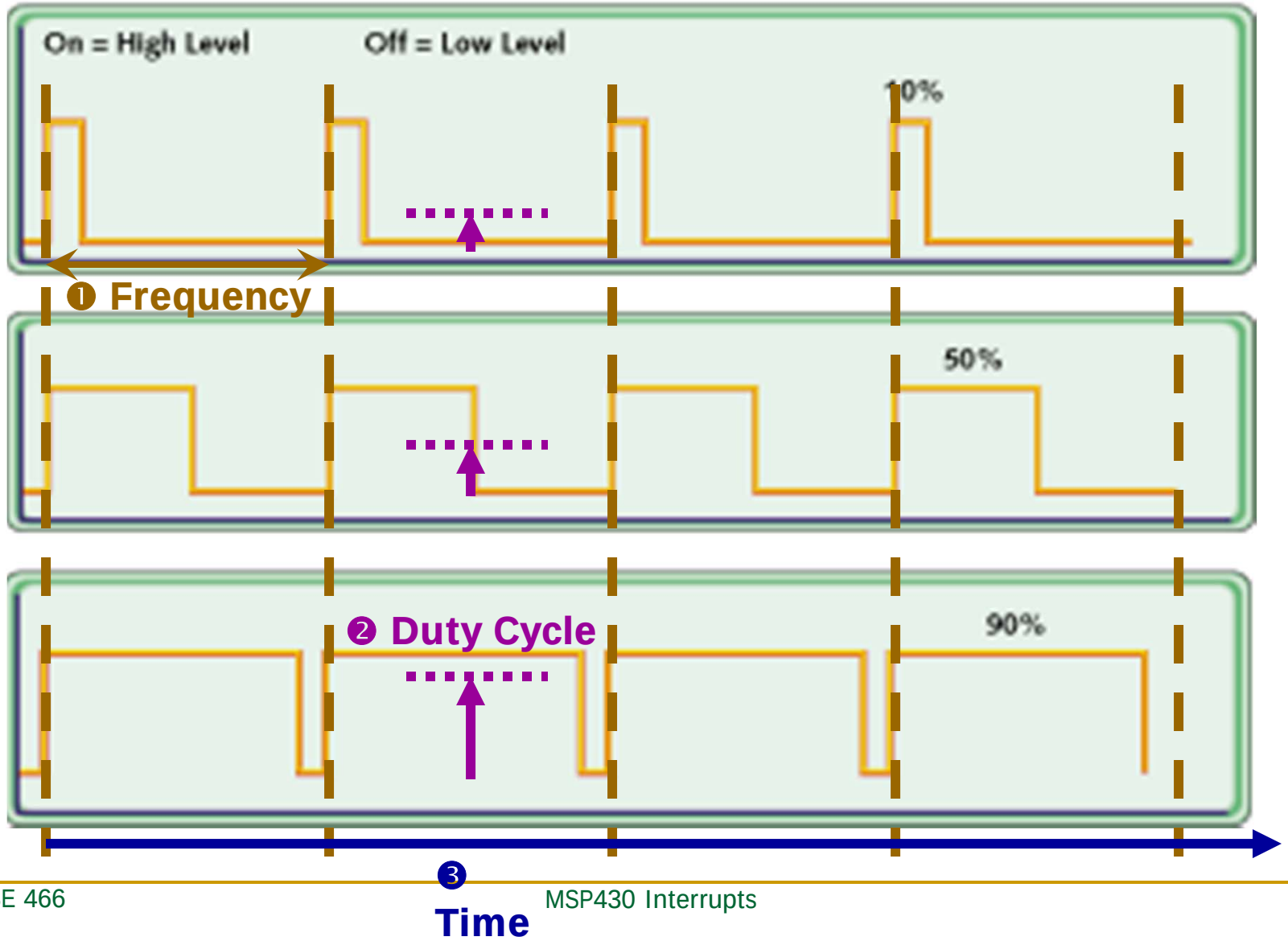
```
// Timer_A3 Interrupt Vector (TAIV) handler
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A(void)
{
  switch( TAIV )
  {
  case  2:           // CCR1
    {
    P1OUT ^= 0x01;  // Toggle P1.0
    CCR1 += 0xA000; // Add Offset to CCR1 == 0xA000
    }
          break;
  case  4: break;   // CCR2 not used
  case 10: break;   // overflow not used
 }
}
```

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.

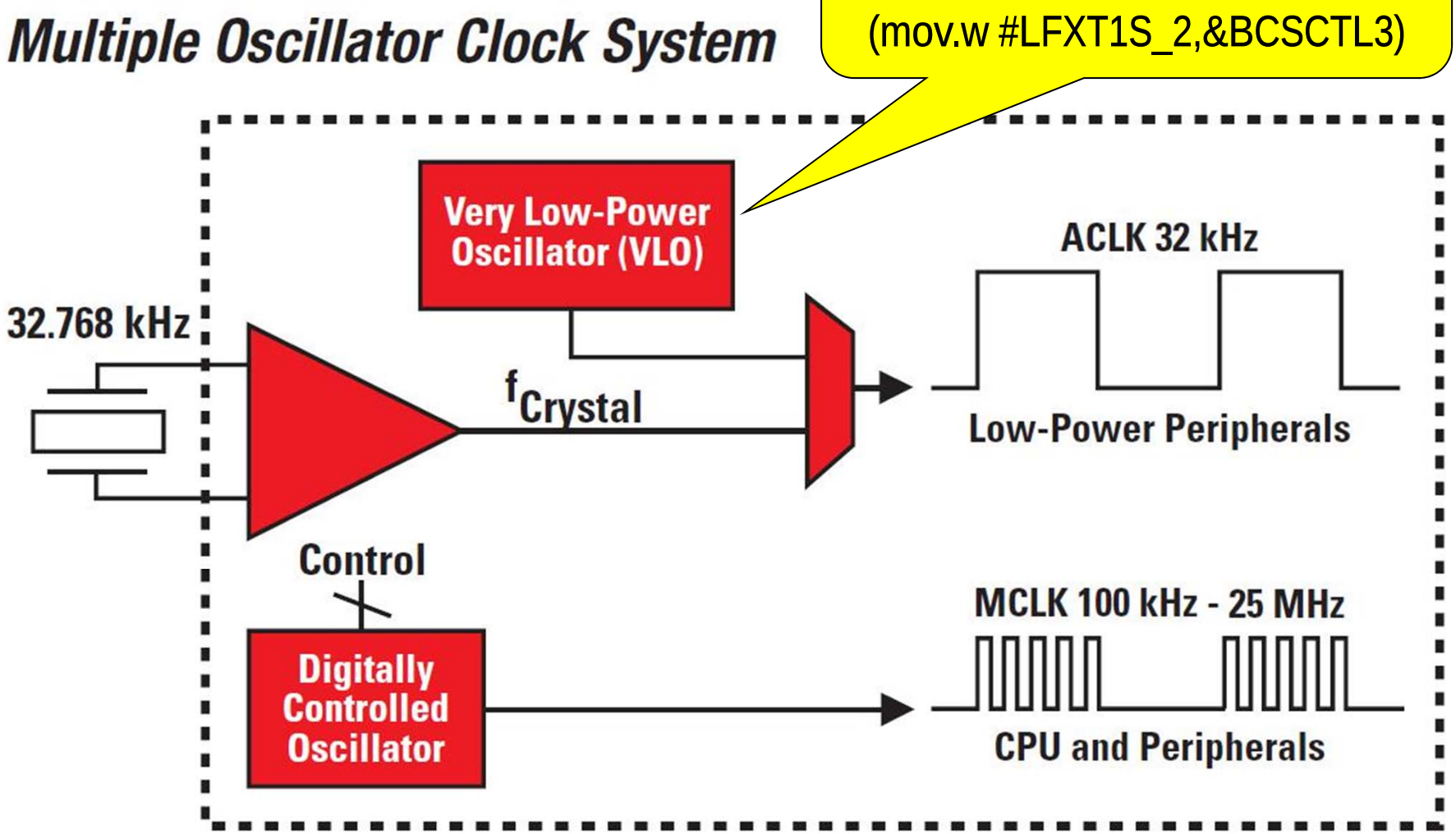
PWM Machines

PWM – Frequency/Duty Cycle



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
```

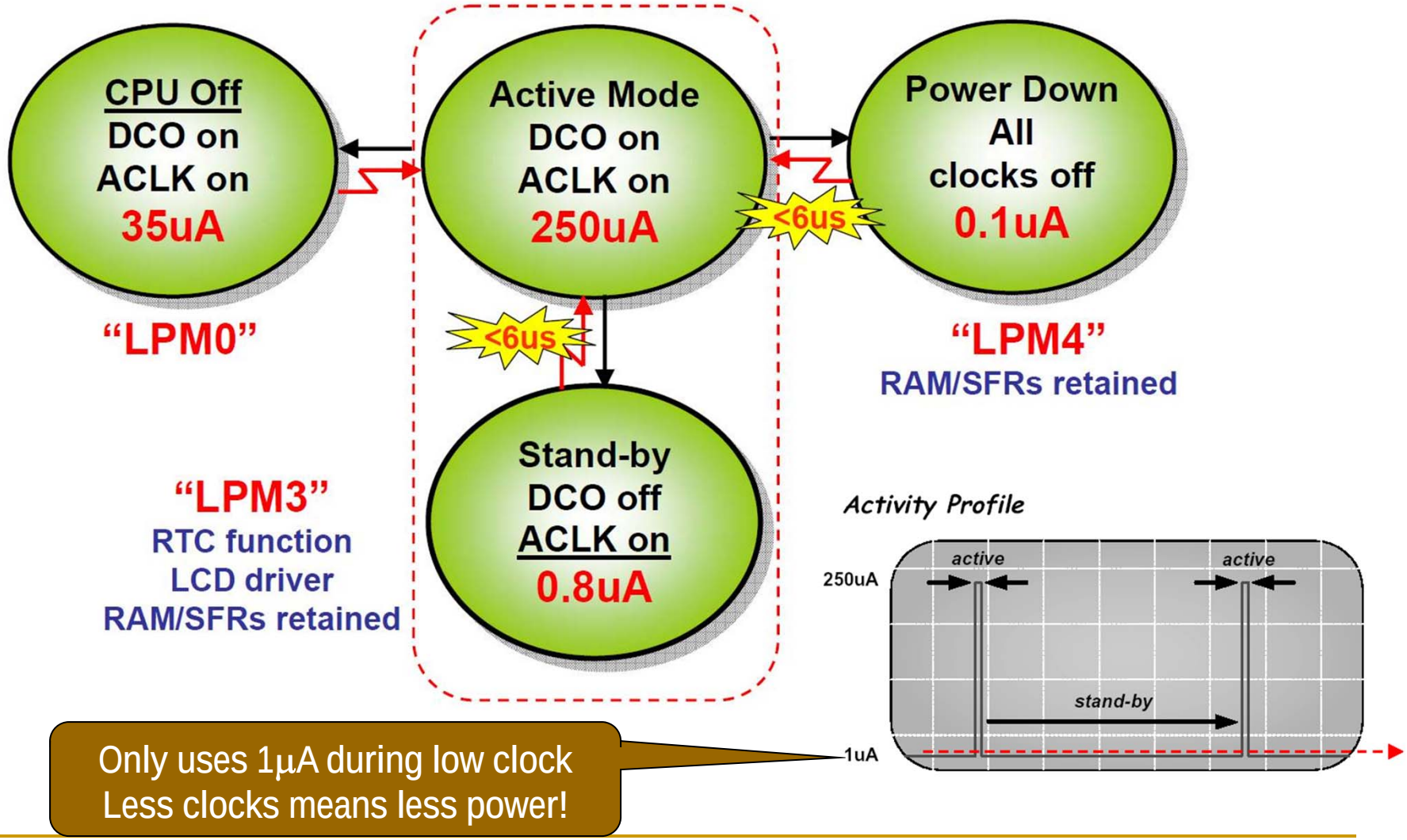
- Using C code:

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

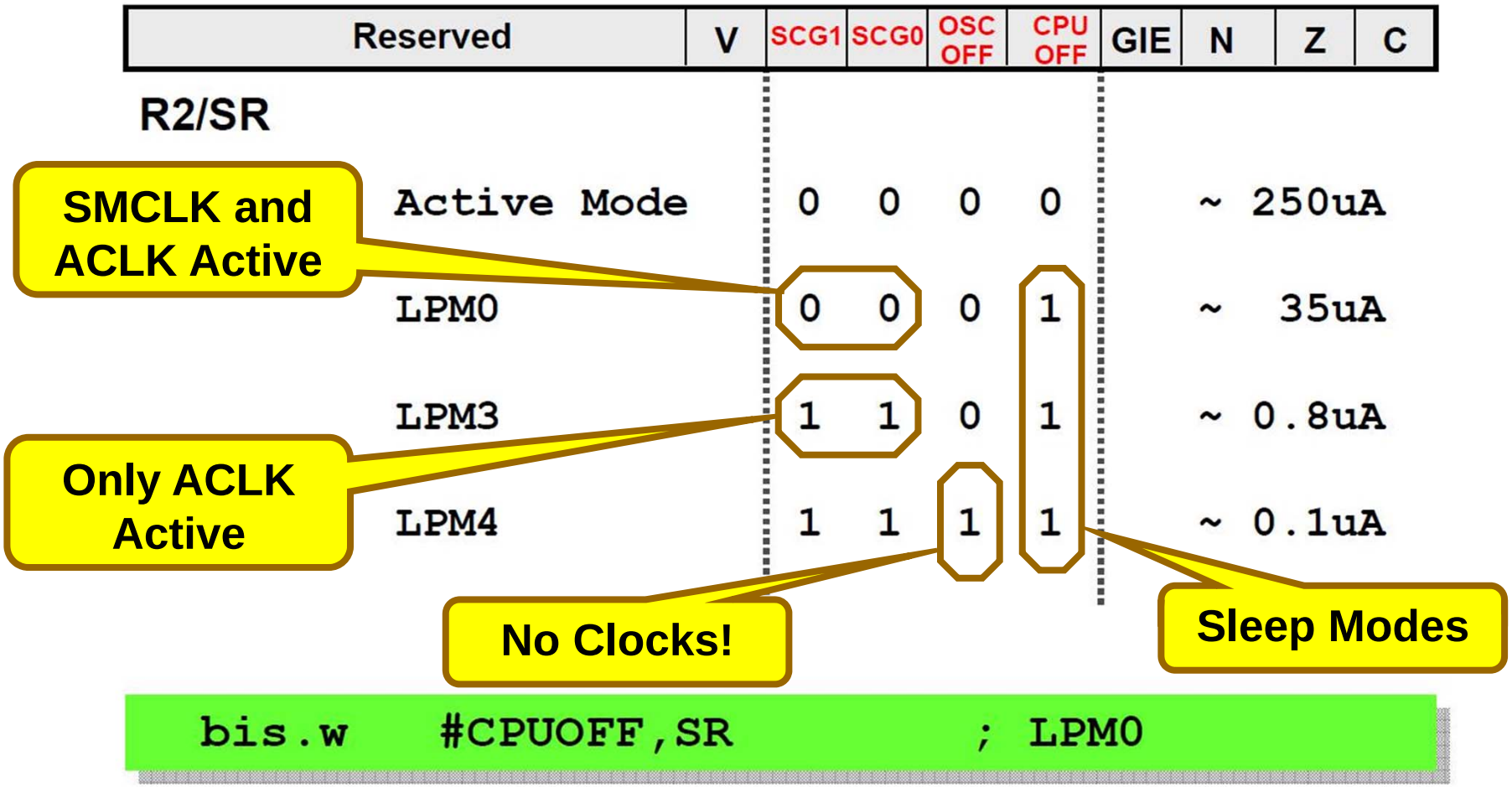
Processor Clock Speeds

- 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

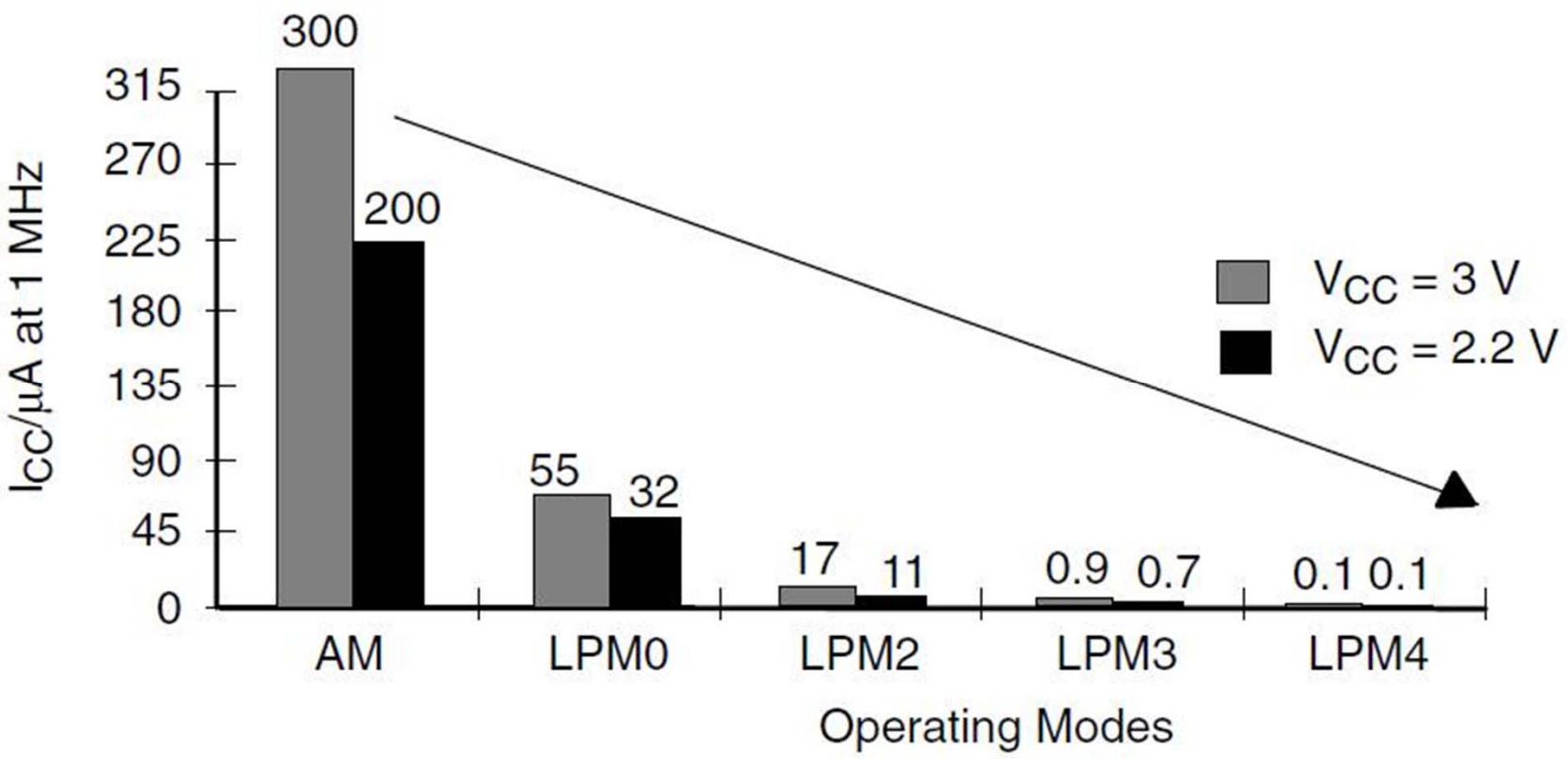


Clocks Off Power Savings



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 ⇒ TxCLK 0 1 ⇒ ACLK 1 0 ⇒ SMCLK 1 1 ⇒ INCLK
7-6	IDx	Clock signal divider:	0 0 ⇒ / 1 0 1 ⇒ / 2 1 0 ⇒ / 4 1 1 ⇒ / 8
5-4	MCx	Clock timer operating mode:	0 0 ⇒ Stop mode 0 1 ⇒ Up mode 1 0 ⇒ Continuous mode 1 1 ⇒ 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	

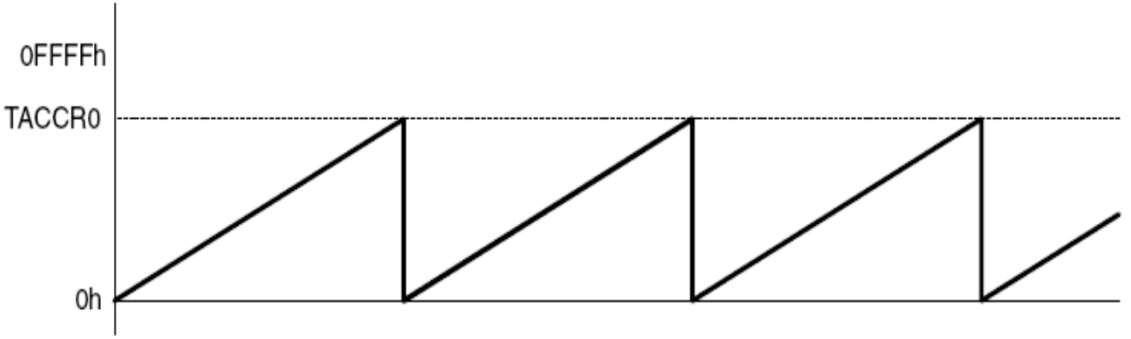
4 Modes of Operation

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

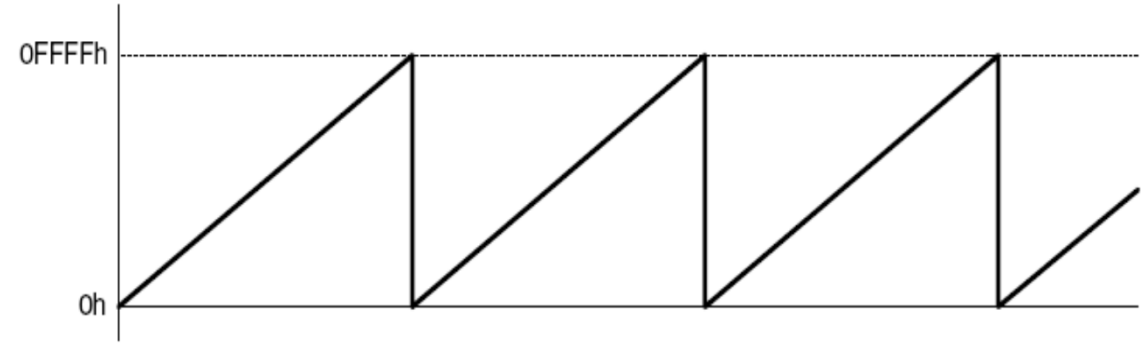
MCx	Mode	Description
0 0	Stop	The timer is halted.
0 1	Up	The timer repeatedly counts from 0x0000 to the value in the TxCCR0 register.
1 0	Continuous	The timer repeatedly counts from 0x0000 to 0xFFFF.
1 1	Up/down	The timer repeatedly counts from 0x0000 to the value in the TxCCR0 register and back down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

Timer Modes

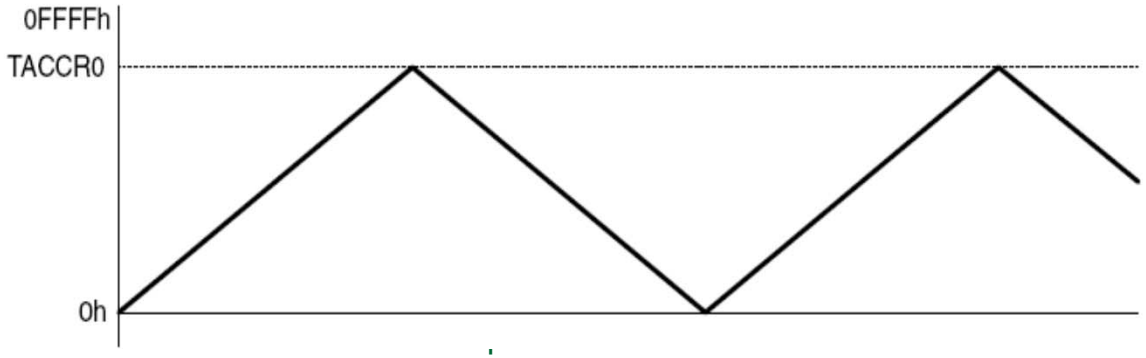
- Up Mode



- Continuous Mode



- Up/Down Mode



TACTL

12.3.1 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)	rw-(0)	rw-(0)	rw-(0)

Unused	Bits 15-10	Unused
TASSELx	Bits 9-8	Timer_A clock source select
		00 TACLK
		01 ACLK
		10 SMCLK
		11 INCLK (INCLK is device-specific and is often assigned to the inverted TBCLK) (see the device-specific data sheet)
IDx	Bits 7-6	Input divider. These bits select the divider for the input clock.
		00 /1
		01 /2
		10 /4
		11 /8
MCx	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power.
		00 Stop mode: the timer is halted.
		01 Up mode: the timer counts up to TACCR0.
		10 Continuous mode: the timer counts up to 0FFFFh.
		11 Up/down mode: the timer counts up to TACCR0 then down to 0000h.
Unused	Bit 3	Unused
TACLR	Bit 2	Timer_A clear. Setting this bit resets TAR, the clock divider, and the count direction. The TACLR bit is automatically reset and is always read as zero.
TAIE	Bit 1	Timer_A interrupt enable. This bit enables the TAIFG interrupt request.
		0 Interrupt disabled
		1 Interrupt enabled
TAIFG	Bit 0	Timer_A interrupt flag
		0 No interrupt pending
		1 Interrupt pending

TAR & TACCRx

12.3.2 TAR, Timer_A Register

15	14	13	12	11	10	9	8
TARx							
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
TARx							
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

TARx	Bits 15-0	Timer_A register. The TAR register is the count of Timer_A.

12.3.3 TACCRx, Timer_A Capture/Compare Register x

15	14	13	12	11	10	9	8
TACCRx							
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
TACCRx							
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

TACCRx	Bits 15-0	Timer_A capture/compare register. Compare mode: TACCRx holds the data for the comparison to the timer value in the Timer_A Register, TAR. Capture mode: The Timer_A Register, TAR, is copied into the TACCRx register when a capture is performed.

TACCTLx

12.3.4 TACCTLx, Capture/Compare Control Register

15	14	13	12	11	10	9	8
CMx		CCISx		SCS	SCCI	Unused	CAP
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	r0	rw-(0)
7	6	5	4	3	2	1	0
OUTMODx			CCIE	CCI	OUT	COV	CCIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	rw-(0)

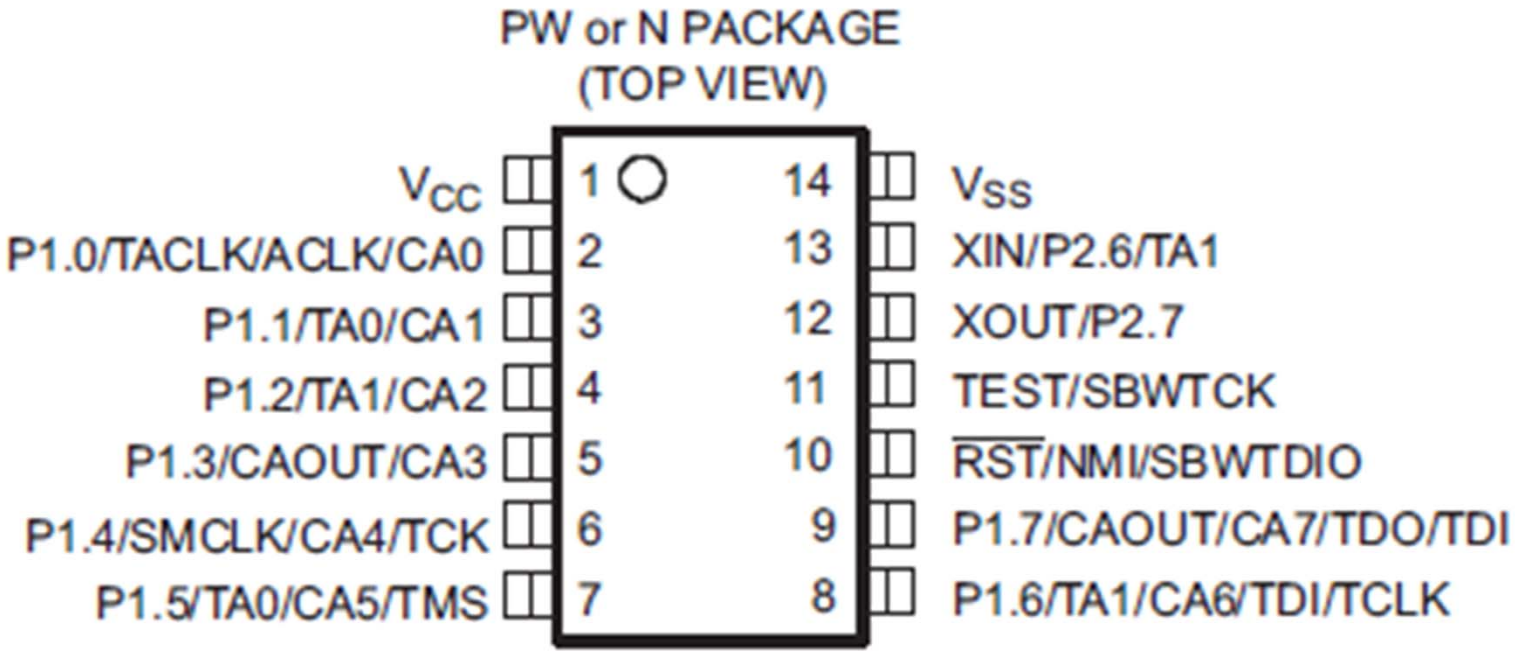
CMx	Bit 15-14	Capture mode 00 No capture 01 Capture on rising edge 10 Capture on falling edge 11 Capture on both rising and falling edges
CCISx	Bit 13-12	Capture/compare input select. These bits select the TACCRx input signal. See the device-specific data sheet for specific signal connections. 00 CCIxA 01 CCIxB 10 GND 11 V_{CC}
SCS	Bit 11	Synchronize capture source. This bit is used to synchronize the capture input signal with the timer clock. 0 Asynchronous capture 1 Synchronous capture
SCCI	Bit 10	Synchronized capture/compare input. The selected CCI input signal is latched with the EQUx signal and can be read via this bit
Unused	Bit 9	Unused. Read only. Always read as 0.
CAP	Bit 8	Capture mode 0 Compare mode 1 Capture mode
OUTMODx	Bits 7-5	Output mode. Modes 2, 3, 6, and 7 are not useful for TACCR0, because EQUx = EQU0. 000 OUT bit value 001 Set 010 Toggle/reset 011 Set/reset 100 Toggle 101 Reset 110 Toggle/set 111 Reset/set
CCIE	Bit 4	Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag. 0 Interrupt disabled 1 Interrupt enabled
CCI	Bit 3	Capture/compare input. The selected input signal can be read by this bit.
OUT	Bit 2	Output. For output mode 0, this bit directly controls the state of the output. 0 Output low 1 Output high
COV	Bit 1	Capture overflow. This bit indicates a capture overflow occurred. COV must be reset with software. 0 No capture overflow occurred 1 Capture overflow occurred
CCIFG	Bit 0	Capture/compare interrupt flag 0 No interrupt pending 1 Interrupt pending

OUTMOD

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.

Configuring PWM



PWM can be configured to appear on TA1 pins

PxSEL.x that chooses which pin TA1 connects to

TAIV

12.3.5 TAIV, Timer_A Interrupt Vector Register

15	14	13	12	11	10	9	8
0	0	0	0	0	0	0	0
r0	r0	r0	r0	r0	r0	r0	r0

7	6	5	4	3	2	1	0
0	0	0	0	TAIVx			0
r0	r0	r0	r0	r-(0)	r-(0)	r-(0)	r0

TAIVx Bits 15-0 Timer_A interrupt vector value

TAIV Contents	Interrupt Source	Interrupt Flag	Interrupt Priority
00h	No interrupt pending	-	
02h	Capture/compare 1	TACCR1 CCIFG	Highest
04h	Capture/compare 2 (1)	TACCR2 CCIFG	
06h	Reserved	-	
08h	Reserved	-	
0Ah	Timer overflow	TAIFG	
0Ch	Reserved	-	
0Eh	Reserved	-	Lowest

(1) Not implemented in MSP430x20xx devices

Msp430x20x3_ta_16.c

PWM without the processor!

```
#include  <msp430x20x3.h>

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT
  P1DIR |= 0x0C;                            // P1.2 and P1.3 output
  P1SEL |= 0x0C;                            // P1.2 and P1.3 TA1/2 options
  CCR0 = 512-1;                             // PWM Period
  CCTL1 = OUTMOD_7;                         // CCR1 reset/set
  CCR1 = 384;                               // CCR1 PWM duty cycle
  TACTL = TASSEL_2 + MC_1;                  // SMCLK, up mode

  _BIS_SR(CPUOFF);                          // Enter LPM0
}
```

End of lecture

Bonus example

```
//  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;
}
```