

M3V2A

Drawing the 'Big Picture'

**Define the problem to
Understand the solution**

Complicated last couple of videos

Don't get lost in the details

Learn the 'Big Picture'

Then the details will fall into place

Big Picture of the Videos so far:

1) Processor Architecture

2) Low Power Modes – Sleep

3) Wake up – Interrupts

4) GPIO

1) Processor Architecture CPU

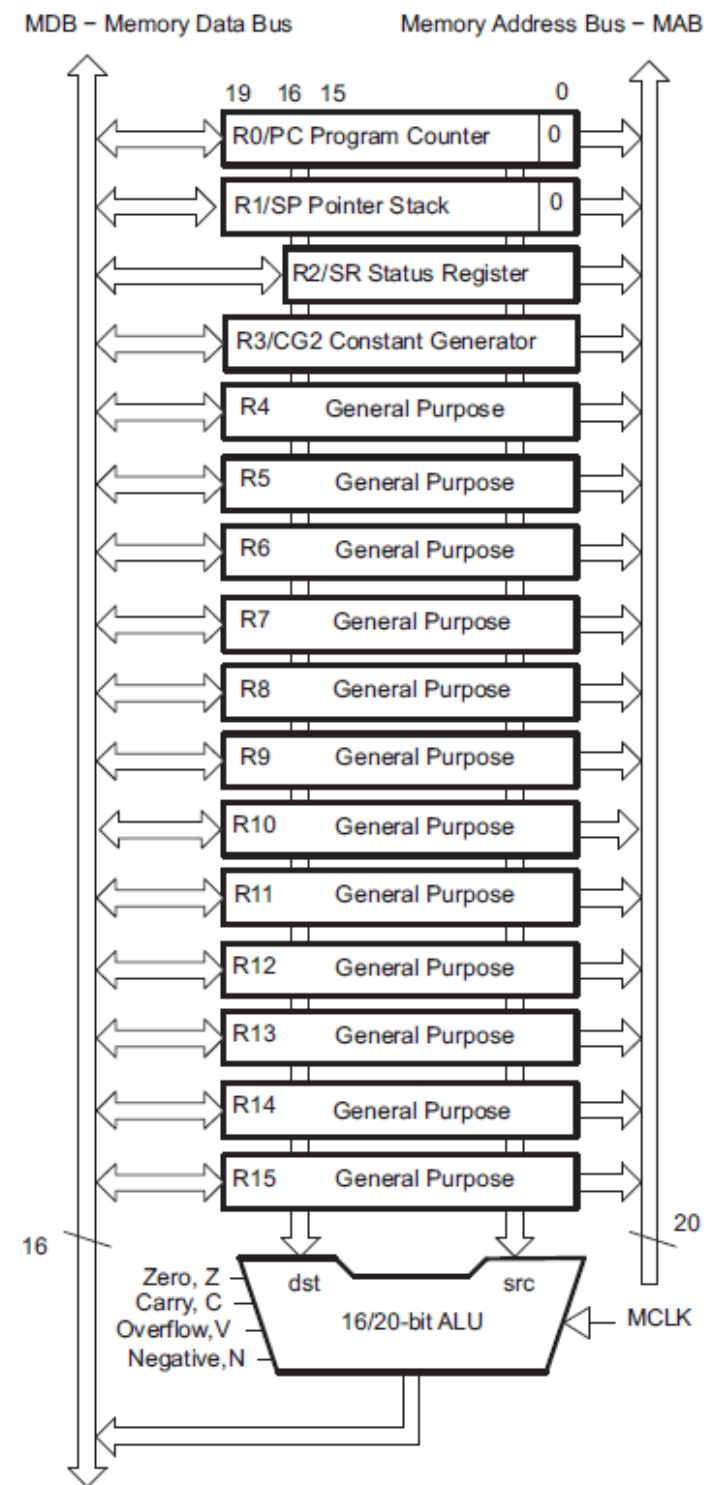
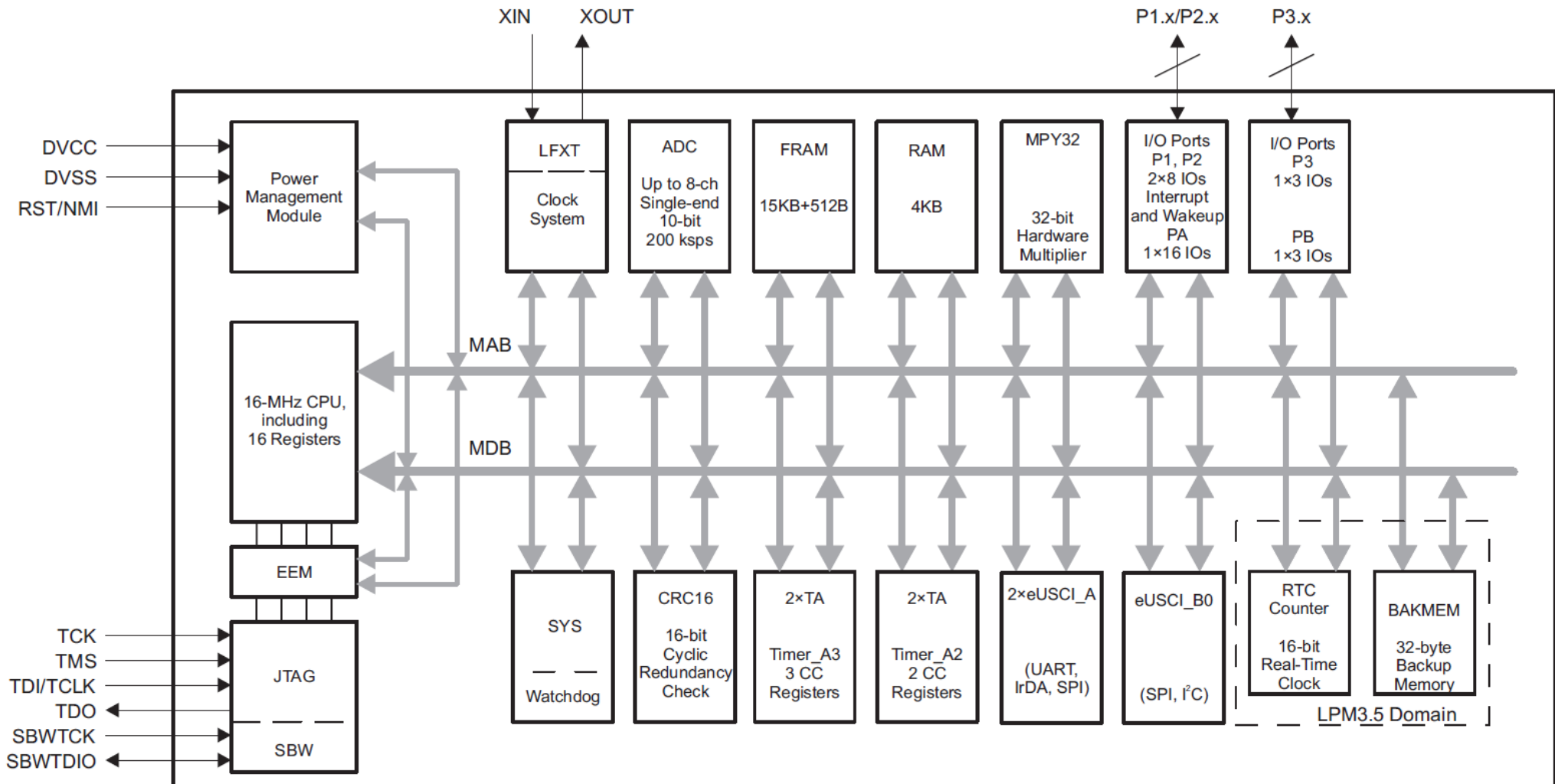


Figure 4-1. MSP430X CPU Block Diagram

1) Processor Architecture

Functional Elements

Block Diagram MSP430FR2433 – Functional Elements SLASE59B



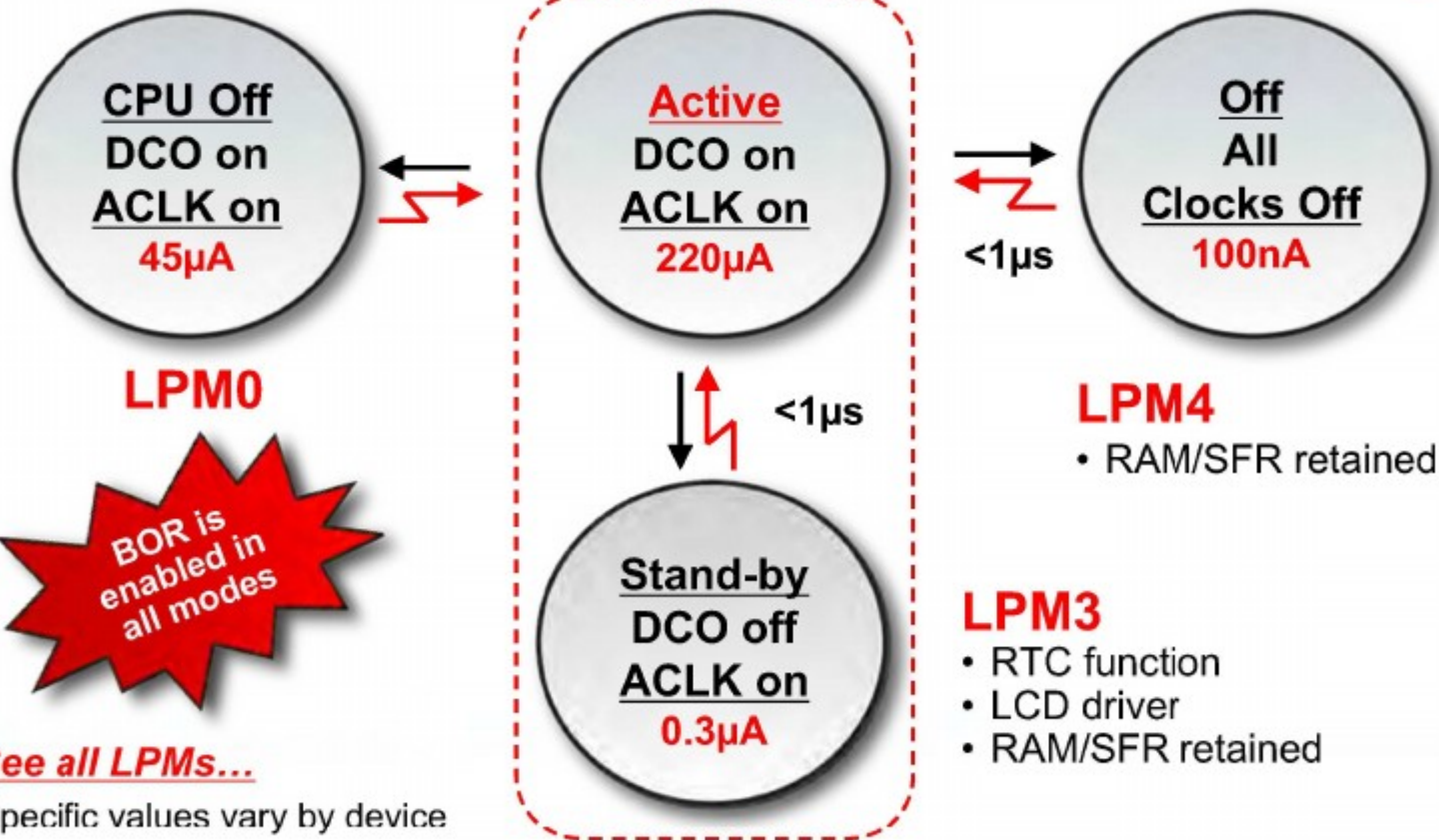
2) Low Power Modes

Low-Power Modes

Operating Mode	CPU (MCLK)	SMCLK	ACLK	RAM Retention	BOR	Self Wakeup	Interrupt Sources
Active	☒	☒	☒	☒	☒	☒	Timers, ADC, DMA, WDT, I/O, External Interrupt, COMP, Serial, RTC, other...
LPM0		☒	☒	☒	☒	☒	
LPM1		☒	☒	☒	☒	☒	
LPM2			☒	☒	☒	☒	
LPM3			☒	☒	☒	☒	
LPM3.5					☒	☒	External Interrupt, RTC
LPM4				☒	☒		External Interrupt
LPM4.5					☒		External Interrupt

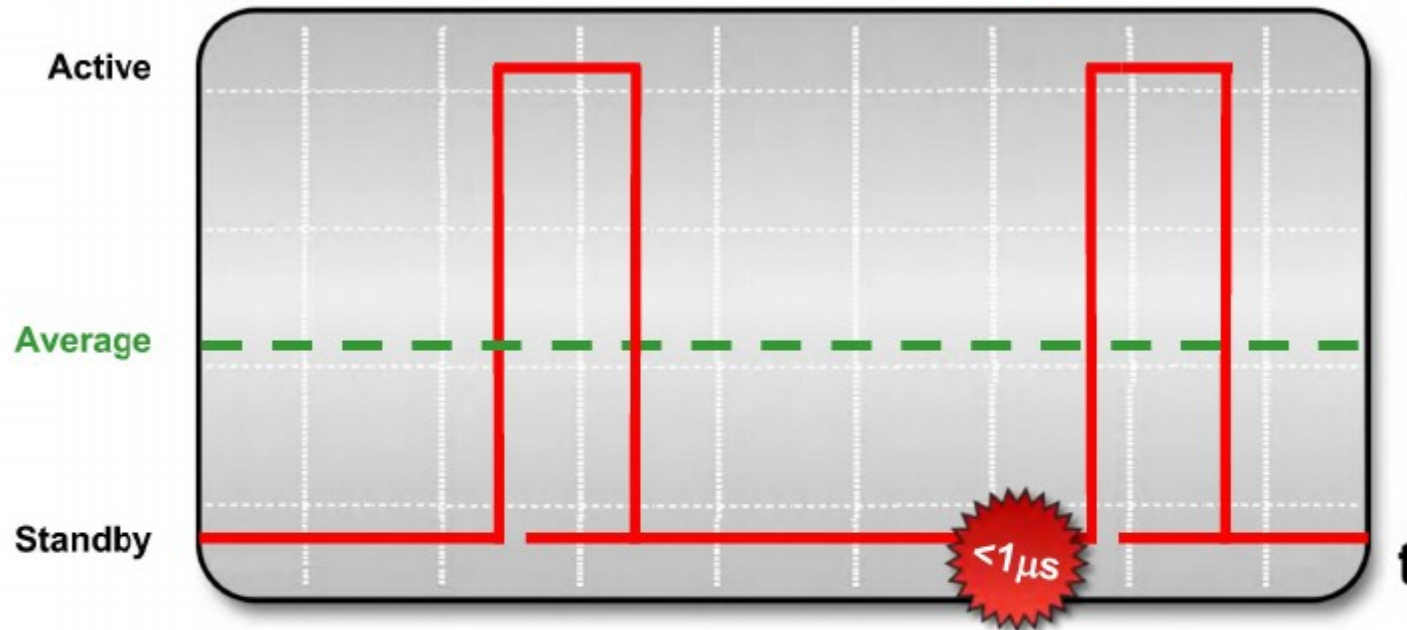
Low Power Modes

MSP430 Low Power Modes



3)Wake up – Interrupts

Ultra-Low Power Activity Profile



- Minimize active time
- Maximize time in **Low Power Modes**
- Interrupt driven performance on-demand with **$<1\mu\text{s}$ wakeup time**
- Always-On, Zero-Power **Brownout Reset (BOR)**

3)Wake up – Interrupts

Waiting for an Event: Family vacation



Polling

Interrupts

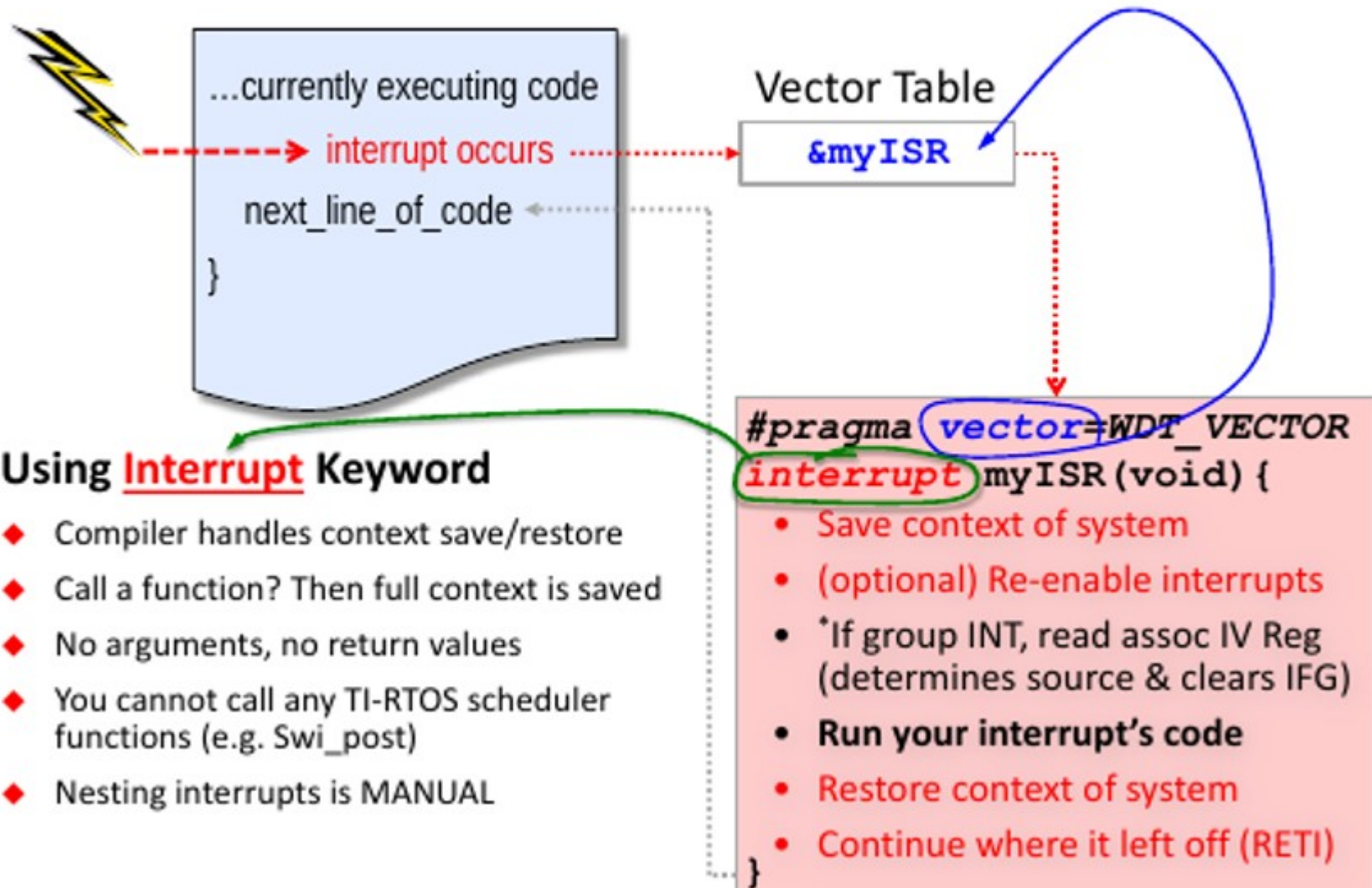
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?

Wake me up when we get there...

Both methods signal that we have arrived at our destination. In most cases, though, the use of Interrupts tends to be much more efficient. For example, in the case of the MSP430, we often want to sleep the processor while waiting for an event. When the event happens and signals us with an interrupt, we can wake up, handle the event and then return to sleep waiting for the next event.

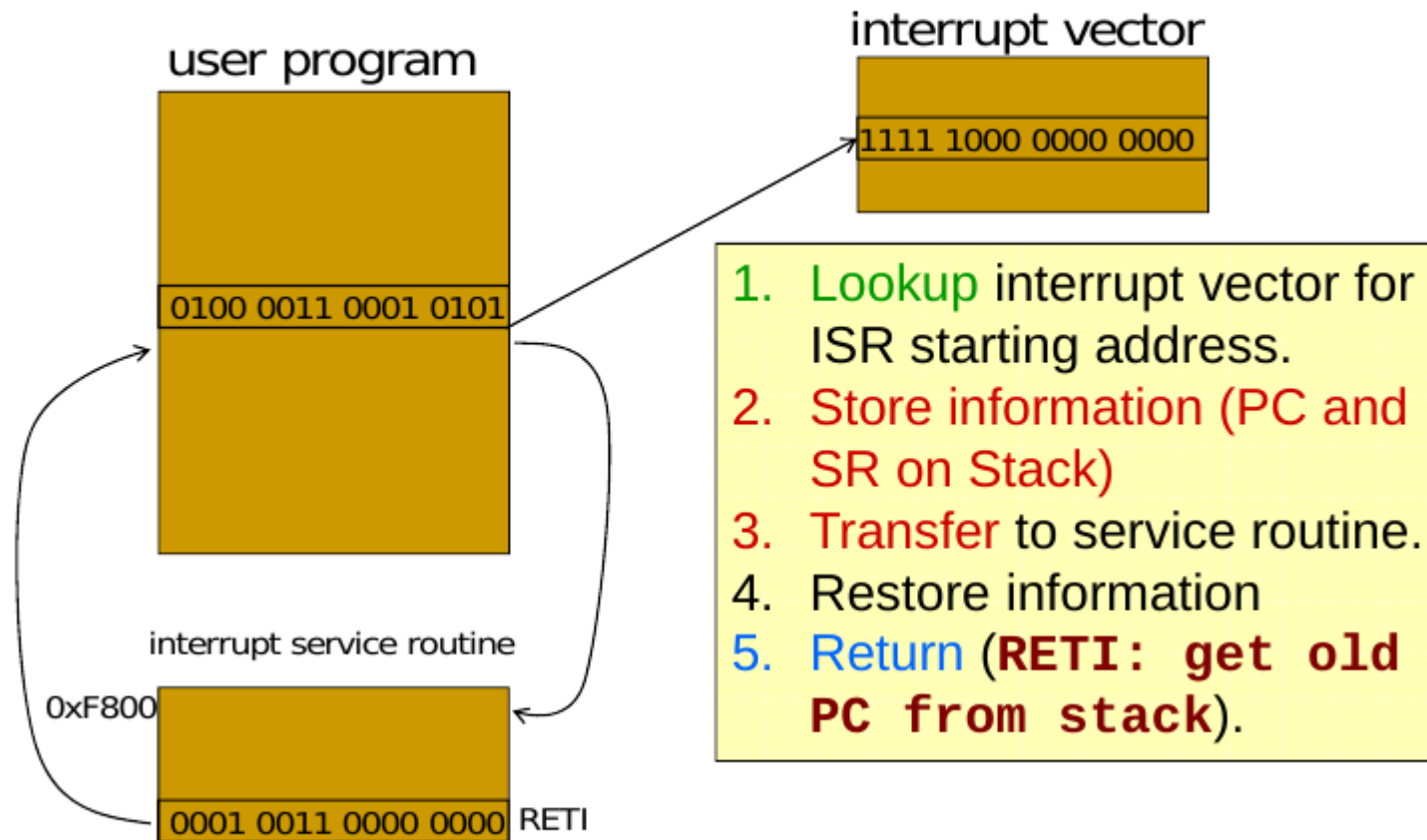
3)Wake up – Interrupts

□ Interrupt Service Routine (ISR)



3)Wake up – Interrupts

Serving Interrupt Request



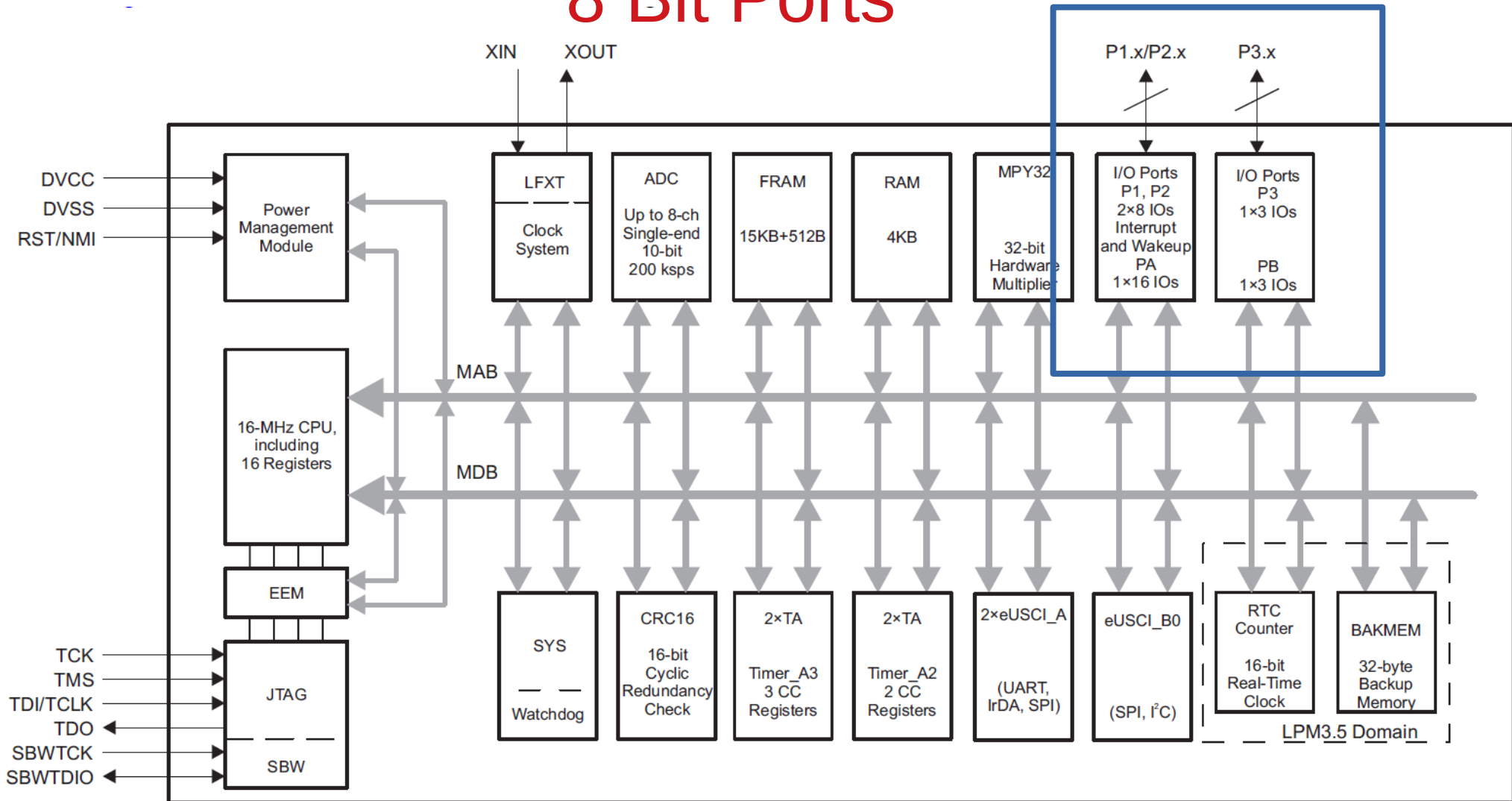
3)Wake up – Interrupts

```
23 #include <msp430.h>
24
25 int main(void)
26 {
27     WDTCTL = WDTPW | WDTHOLD;           // Stop WDT
28
29     // Configure GPIO
30     P1DIR |= BIT0;                     // P1.0 output
31     P1OUT |= BIT0;                     // P1.0 high
32
33     // Disable the GPIO power-on default high-impedance mode to activate
34     // previously configured port settings
35     PMSCTL0 &= ~LOCKLPM5;
36
37     TA0CTL0 |= CCIE;                   // TACCR0 interrupt enabled
38     TA0CCR0 = 50000;
39     TA0CTL |= TASSEL__SMCLK | MC__UP;  // SMCLK, Up mode
40
41     __bis_SR_register(LPM3_bits | GIE); // Go to Sleep: Enter LPM3 w/ interrupts
42     __no_operation();                 // For debug
43 }
44
45 // Timer A0 interrupt service routine
46 #pragma vector = TIMER0_A0_VECTOR
47 __interrupt void Timer_A (void)
48 {
49 {
50     P1OUT ^= BIT0;
51 }
52 }
```

4)GPIO

General Purpose Input/Output

8 Bit Ports



4)GPIO

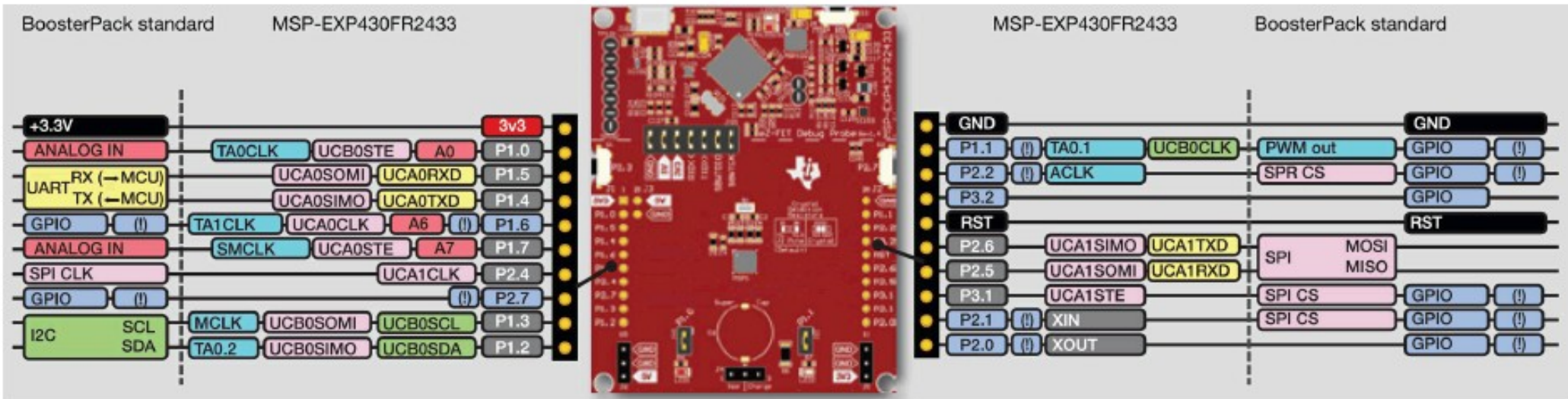


Figure 11. LaunchPad Kit to BoosterPack Module Connector Pinout

Ports: Each bit has Multiplexed functions

4) GPIO

Input Mode

PxDir

0 PxIn

1 PxOut

PxREN

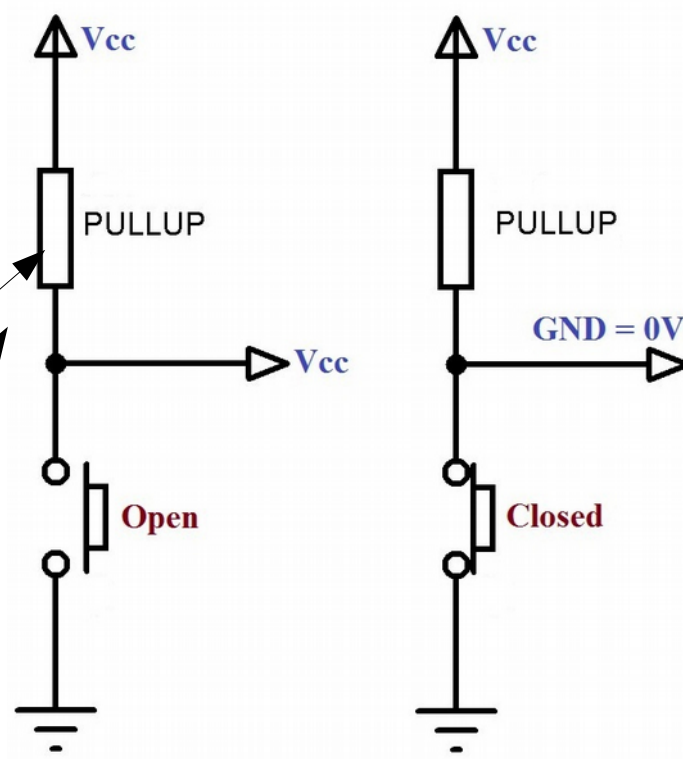
(input mode)

PxOUT

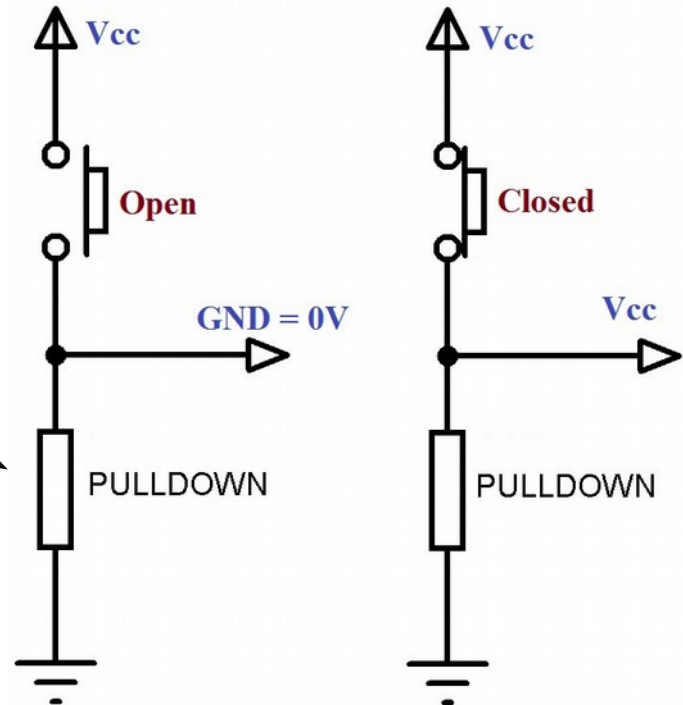
0 – pull down

1 – pull up

Each Port Bit



PxREN = 1
PxOUT=1

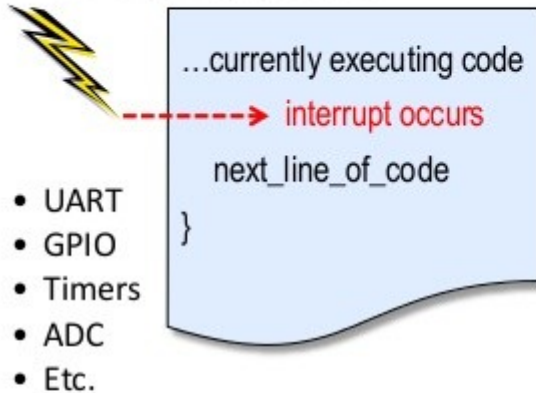


PxREN = 1
PxOUT=0

4)GPIO

How do Interrupts Work?

1. An interrupt occurs



3. CPU acknowledges INT by...

- Current instruction completes
- Saves return-to location on stack
- Saves 'Status Reg' (SR) to the stack
- Clears most of SR, which turns off interrupts globally (SR.GIE=0)
- Determines INT source (or group)
- Clears non-grouped flag* (IFG=0)
- Reads interrupt vector & calls ISR

2. Sets a flag bit (IFG) in register



Each 8 Bit Port Bit

PxIE – Interrupt Enable Pin

PxIES – Rising or falling edge

PxIFG – Interrupt Flag (status)

Plus the whole CPU

GIE – General Interrupt Enable

Define the Example program Goals

What are we trying to do?

Single Sentence:

Blink the LED with an Interrupt created by pressing a Button

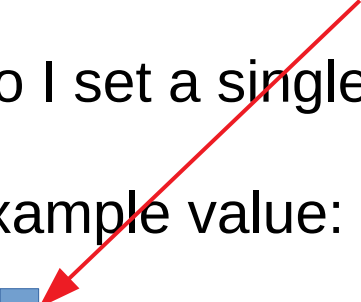
Program Elements: Setup

Red LED is connected to P1.0 - Port 1 Bit 0
Set as Output bit

Push Button SW1 is connected to P2.3 -
Port 2 Bit 3 - Set as:
Input Bit
Pull up Resistor
Interrupt Flag cleared
Interrupt Enable
Rising Edge Trigger

How do I set a single bit as part of an 8 Bit Port?

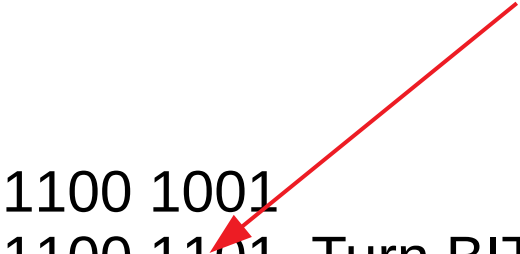
Port example value:



7	6	5	4	3	2	1	0	Bit #
1	1	0	0	1	0	0	1	Value

How to turn BIT2 on without destroying all other values?

EASY!

1. Read the port byte: 1100 1001
 2. Modify the single bit: 1100 1101 Turn BIT2 On
 3. Write the port byte back: 1100 1101
- 

How modify single bit?

Use C Language Bitwise Operators

BitWise Operators – C language review

Red LED is connected to P1.0 (Port 1 BIT0)

Want P1.0 = 1; Turn LED on

Want P1.0 = 0; Turn LED off

Can't write `P1OUT = 1` because the write will output all 8 bits

Therefore writing `P1OUT` would force

`P1OUT = 00000001`

Suppose P1.2 is 'on' (Port 1 BIT2)

`00000101`

Then `P1OUT=1` would turn 'off' P1.2

Use BitWise Operators to modify single bits

YouTube: <https://youtu.be/d0AwjSpNXR0>

BitWise Operators – C language review

Turn Red LED BIT On

'OR' BIT0 with existing Port 1 Output value

P1OUT |= BIT0

C shorthand for P1OUT=BIT0 | P1OUT; // 'OR'

Turn Red LED BIT Off

'AND' the complement of BIT0

with existing Port 1 Output value

P1OUT &=~ BIT0

Toggle Red LED BIT On/Off

'XOR' BIT0 with existing Port 1 Output value

P1OUT ^= BIT0

After SetUp, then program will put the processor to sleep with General Interrupts Enabled

Program Code:

```
// LPM4 - shut down - 0.49uA - OFF Section 6.3 SLASE59B  
__bis_SR_register(LPM4_bits + GIE ); // LPM4 with interrupts enabled
```

Then code the Interrupt Service Routine

```
// Port 2 interrupt service routine
#pragma vector=PORT2_VECTOR
__interrupt void Port_2(void)
{
    // Code to do the Service Task
    // No Calling or return values
    // Brief and specific
}
```

Next Video,

Lets build the actual program!!!!

Single Sentence:

Blink the LED with an Interrupt created by pressing a Button