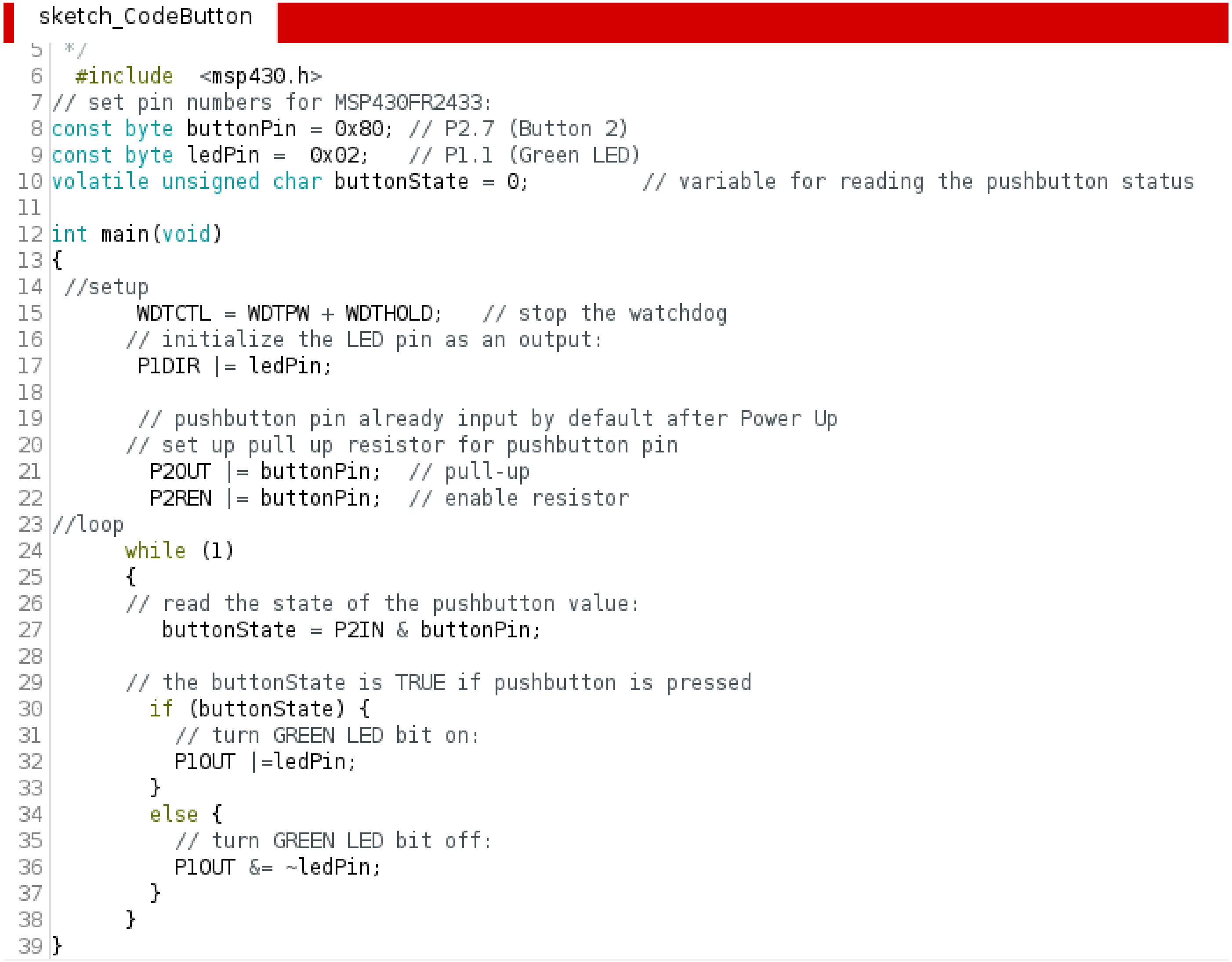
M2v2 CDA 3104

Low Power Modes

Interrupts (Wake up)

How can this be modified to run for 1 year on a battery?

From the last lecture:



```
 */
  #include  <msp430.h>
// set pin numbers for MSP430FR2433:
const byte buttonPin = 0x80; // P2.7 (Button 2)
const byte ledPin =  0x02;   // P1.1 (Green LED)
volatile unsigned char buttonState = 0;         // variable for reading the pushbutton status

int main(void)
{
 //setup
       WDTCTL = WDTPW + WDTHOLD;   // stop the watchdog
      // initialize the LED pin as an output:
       P1DIR |= ledPin;

       // pushbutton pin already input by default after Power Up
      // set up pull up resistor for pushbutton pin
        P2OUT |= buttonPin;  // pull-up
        P2REN |= buttonPin;  // enable resistor
//loop
      while (1)
      {
      // read the state of the pushbutton value:
         buttonState = P2IN & buttonPin;

      // the buttonState is TRUE if pushbutton is pressed
        if (buttonState) {
          // turn GREEN LED bit on:
          P1OUT |=ledPin;
        }
        else {
          // turn GREEN LED bit off:
          P1OUT &= ~ledPin;
        }
      }
}
```

GOING LOW POWER

delay() *vs.* sleep()

The loop() blinks the LED twice, enters delay() state for 10 seconds, then blinks once, and goes to sleep() for 10 seconds.

With **sleep()** and **sleepSeconds()**, the MCU enters LPM3.

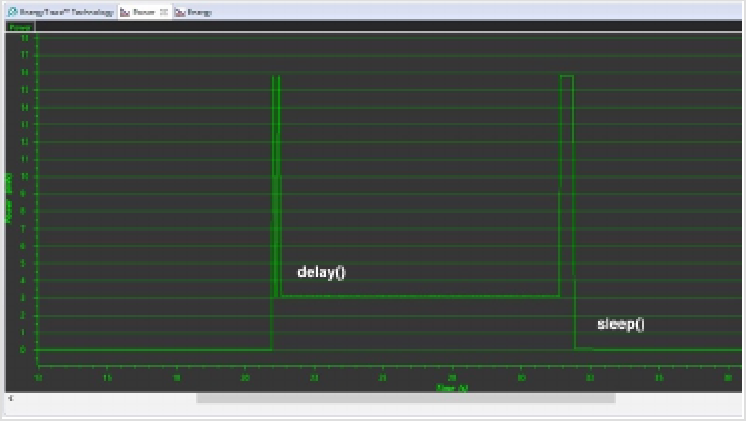
- CPU is disabled.
- MCLK and SMCLK are disabled.
- DCO's dc generator is disabled.
- ACLK remains active.

As a consequence of SMCLK being disabled, background processes such as Serial transmit and receive will halt or get scrambled.

sleep() applies for milliseconds and **sleepSeconds()** for seconds.

```
const uint8_t myLED = RED_LED;
const uint8_t myButton = PUSH2;

void setup()
{
    pinMode(myLED, OUTPUT);
}

void flash()
{
    digitalWrite(myLED, HIGH);
    delay(100);
    digitalWrite(myLED, LOW);
}

void loop()
{
    flash();
    delay(100);
    flash();
    delay(10000);

    flash();
    sleepSeconds(10);
}
```

1



2

3

Notice the difference of power consumption between delay() and sleep().

- **delay()** requires 3 mW or 1 mA
- **sleep()** requires less than 0.1 mW.

The peaks correspond to the LED blinking twice before delay(), and once before sleep().

Use Hardware Interrupt (PUSH2) to WakeUp a processor put to sleep

suspend() *and* wakeup()

With **suspend()**, the MCU enters LPM4.

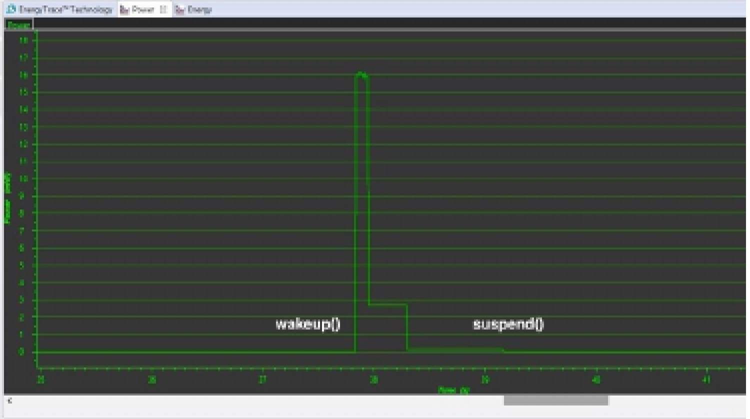
- CPU is disabled.
- ACLK is disabled.
- MCLK and SMCLK are disabled.
- DCO's dc generator is disabled.
- Crystal oscillator is stopped.

The MCU can only react to a hardware interrupt, triggered here by PUSH2. The interrupt calls the buttonISR() routine and launches **wakeup()** to return to active mode.

This example is based on Frank Milburn's code (June 2015), which is derived from @spirilis at 43oh.com

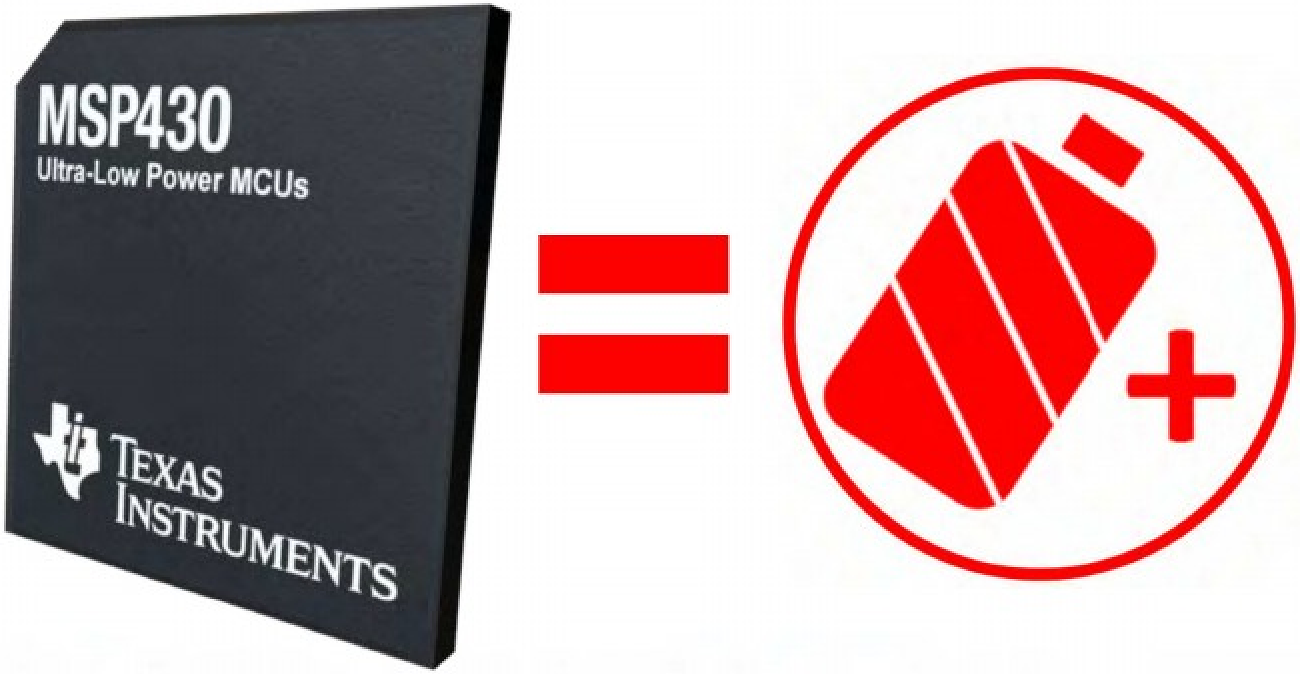
```
const uint8_t myLED = RED_LED;
const uint8_t myButton = PUSH2;

void buttonISR()
{
    wakeup();
}

void setup()
{
    pinMode(myLED, OUTPUT);
    pinMode(myButton, INPUT_PULLUP);
    attachInterrupt(myButton, buttonISR, FALLING);
}

void loop()
{
    digitalWrite(myLED, HIGH);
    delay(100);
    digitalWrite(myLED, LOW);

    suspend();
}
```

The peak corresponds to the LED turned on and happens just after PUSH2 is pressed.

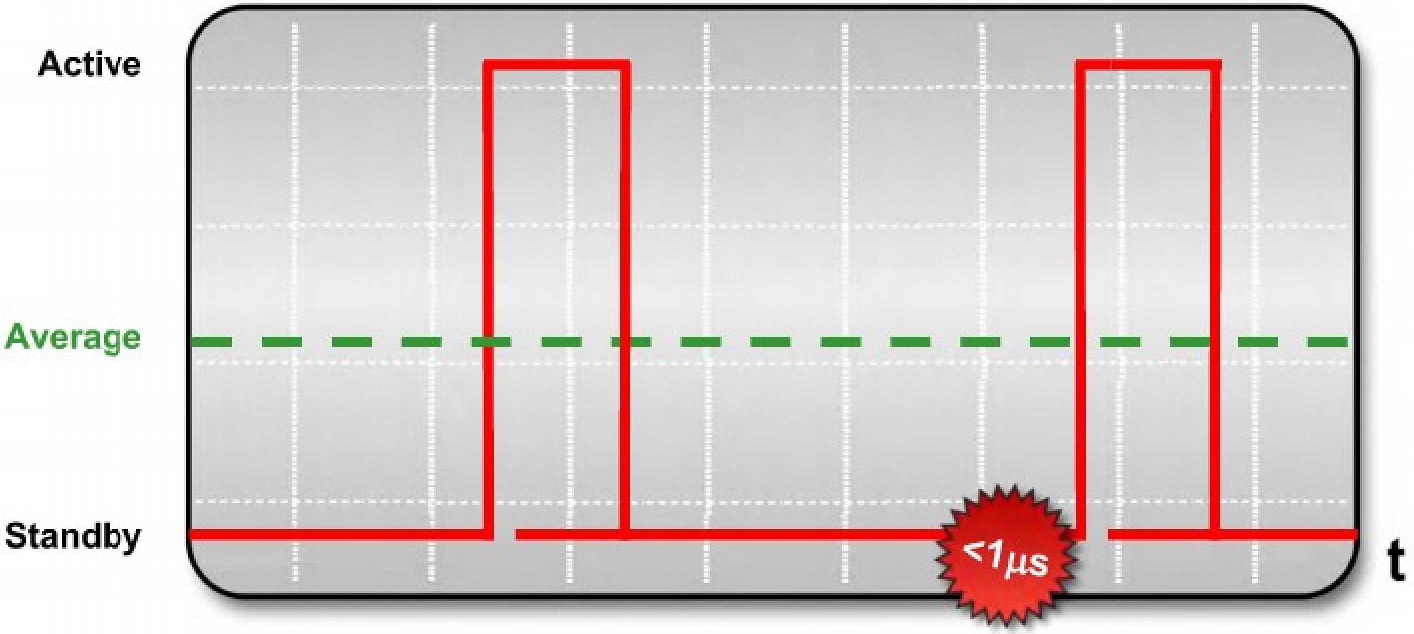


https://embeddedcomputing.weebly.com/ultra-low-power-with-msp430.html

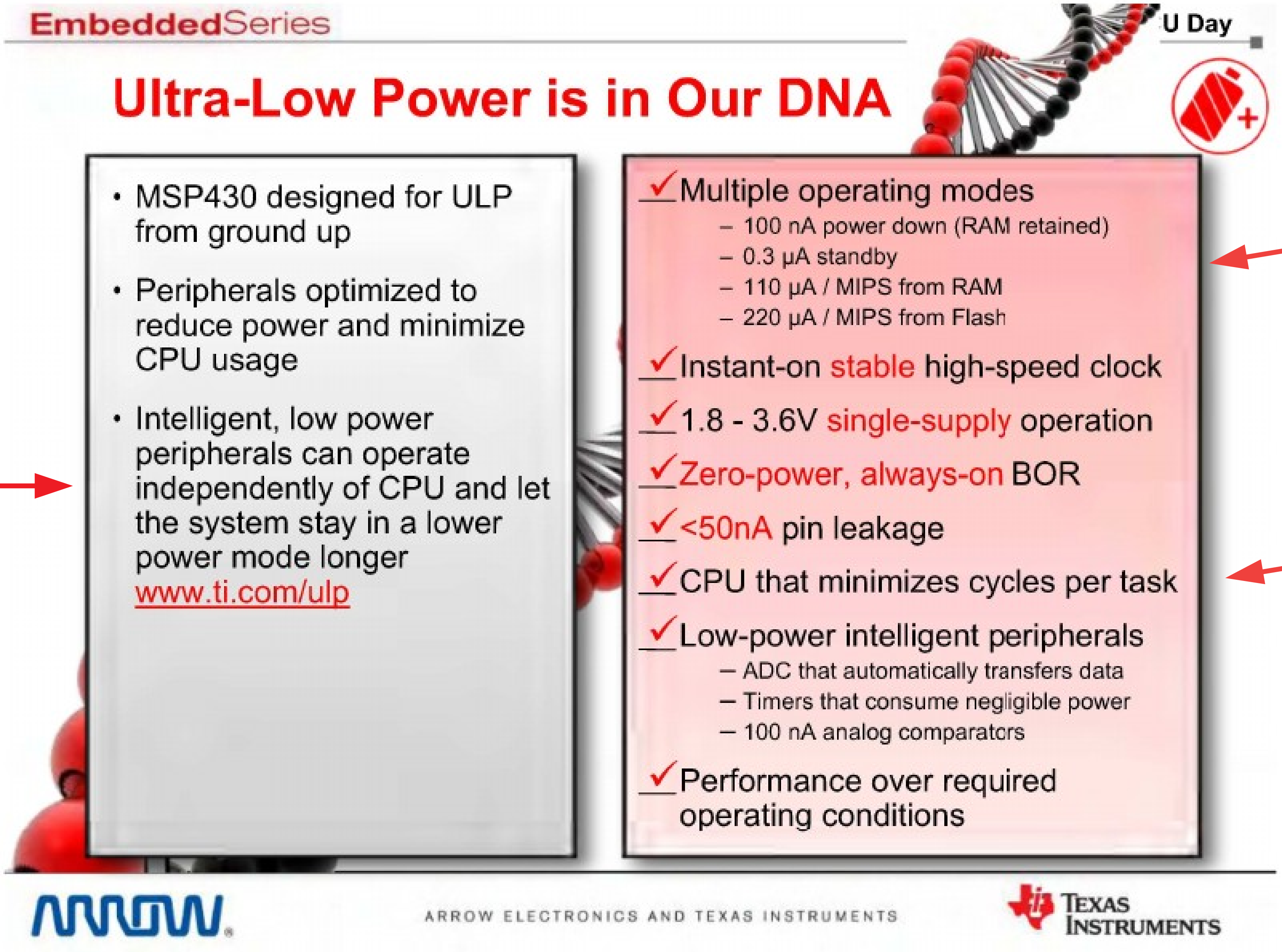
MSP430 is Ultra-Low Power + Performance



Ultra-Low Power Activity Profile



- Minimize active time
- Maximize time in Low Power Modes
- Interrupt driven performance on-demand with <1µs wakeup time
- Always-On, Zero-Power Brownout Reset (BOR)



Ultra-Low Power is in Our DNA

- MSP430 designed for ULP from ground up
- Peripherals optimized to reduce power and minimize CPU usage
- Intelligent, low power peripherals can operate independently of CPU and let the system stay in a lower power mode longer
  www.ti.com/ulp

✓ Multiple operating modes
  - 100 nA power down (RAM retained)
  - 0.3 µA standby
  - 110 µA / MIPS from RAM
  - 220 µA / MIPS from Flash

✓ Instant-on stable high-speed clock

✓ 1.8 - 3.6V single-supply operation

✓ Zero-power, always-on BOR

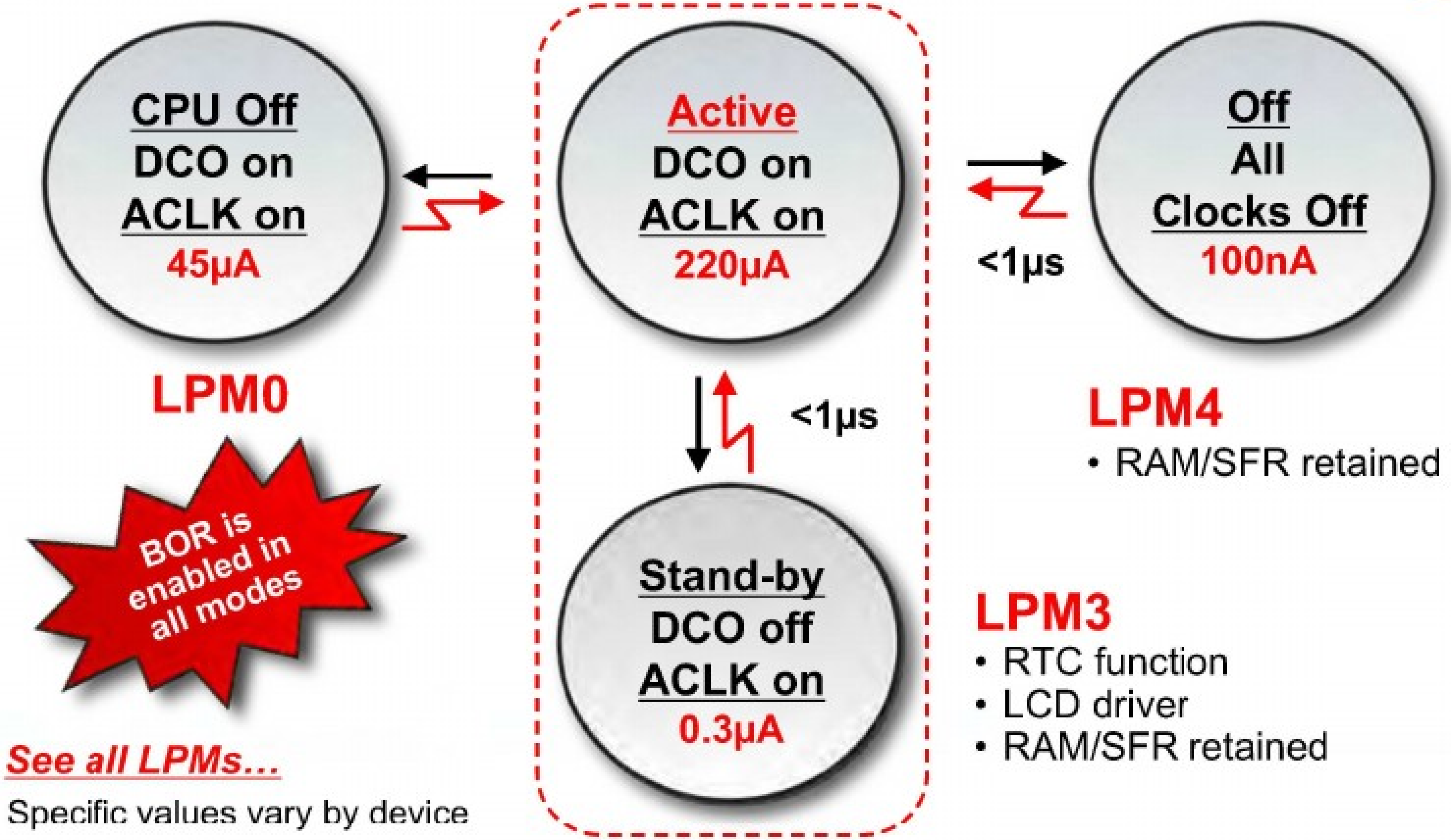
✓ <50nA pin leakage

✓ CPU that minimizes cycles per task

✓ Low-power intelligent peripherals
  - ADC that automatically transfers data
  - Timers that consume negligible power
  - 100 nA analog comparators

✓ Performance over required operating conditions

MSP430 Low Power Modes



CPU Off
DCO on
ACLK on
45µA

LPM0

BOR is enabled in all modes

Active
DCO on
ACLK on
220µA

<1µs

Stand-by
DCO off
ACLK on
0.3µA

<1µs

Off
All
Clocks Off
100nA

LPM4
- RAM/SFR retained

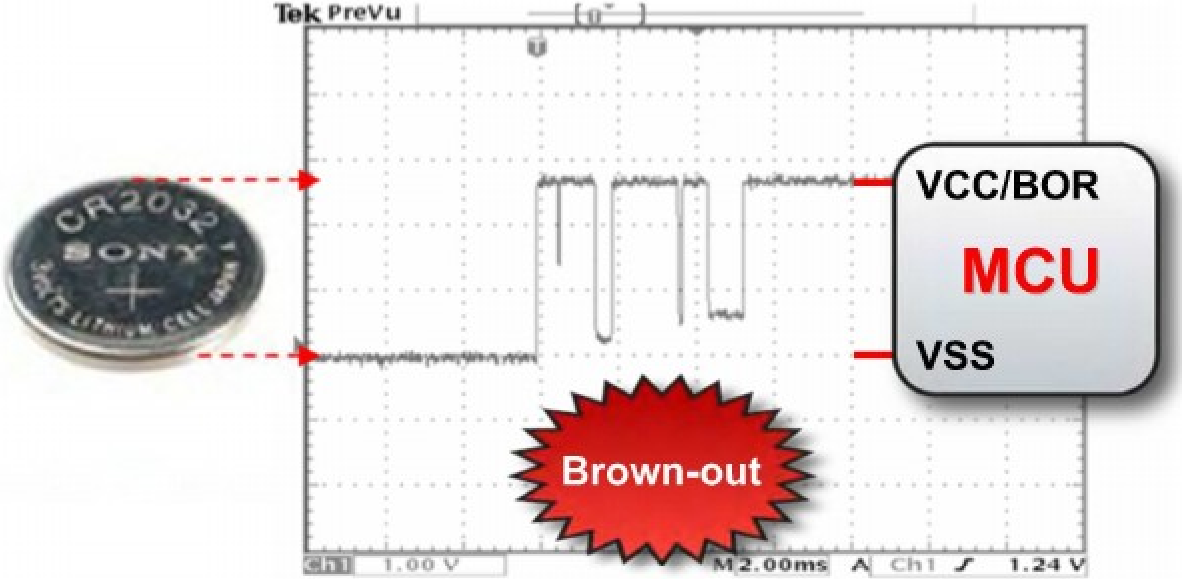
LPM3
- RTC function
- LCD driver
- RAM/SFR retained

See all LPMs…

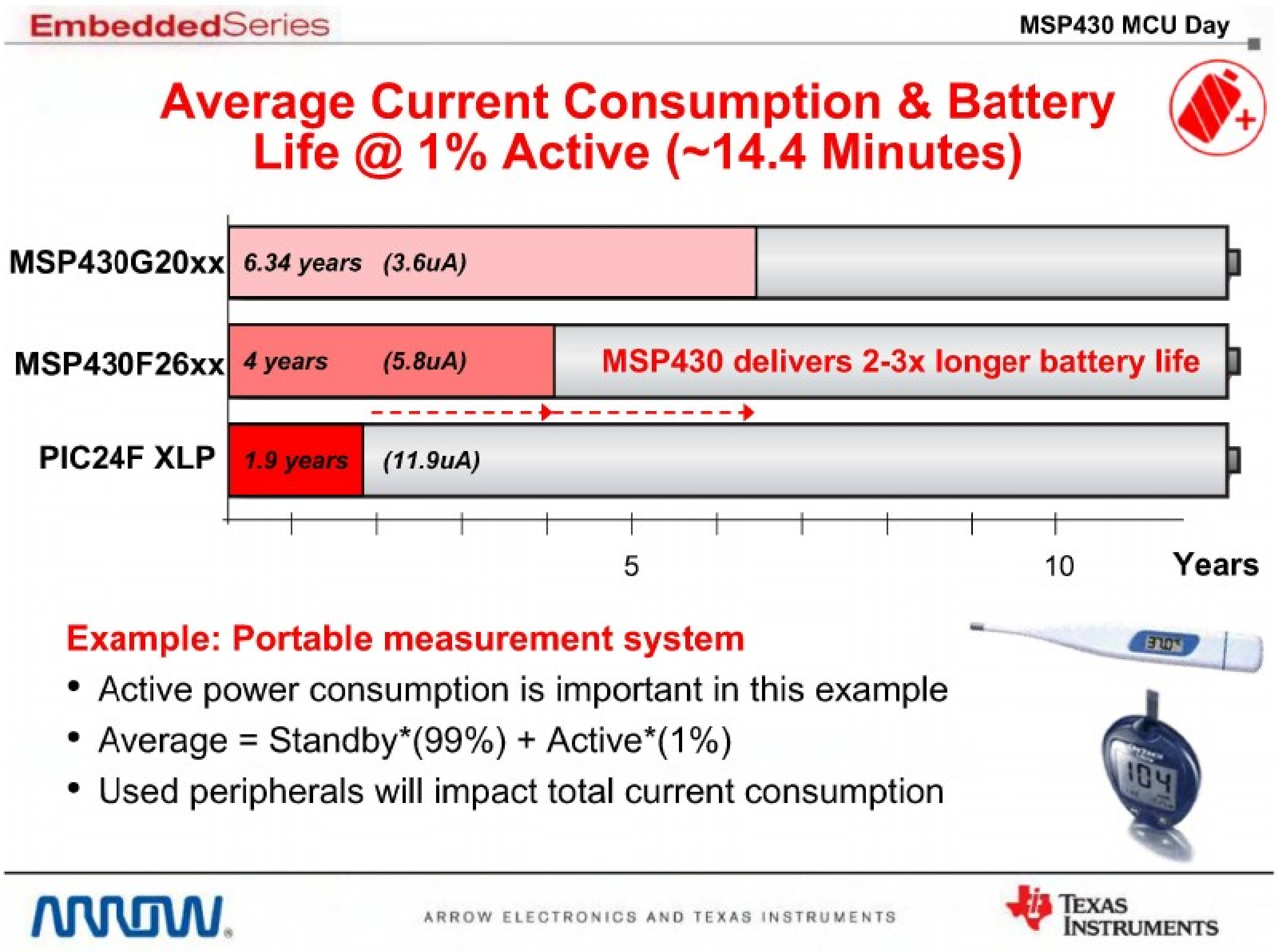
Specific values vary by device

Always-on Brownout Reset

- Brown-out reset (BOR) forces the MCU to reset both on power-up/down
  - When V_{CC} rises and when V_{CC} falls below normal operating range, a POR is triggered.
  - Zero-power Brown Out Reset
  - **Always-on and active in all modes of operation.**



1% active per day is approximately 14.4 minutes operation in 24 hours



EmbeddedSeries MSP430 MCU Day

Average Current Consumption & Battery Life @ 1% Active (~14.4 Minutes)

MSP430G20xx *6.34 years* *(3.6uA)*

MSP430F26xx *4 years* *(5.8uA)* MSP430 delivers 2-3x longer battery life

PIC24F XLP *1.9 years* *(11.9uA)*

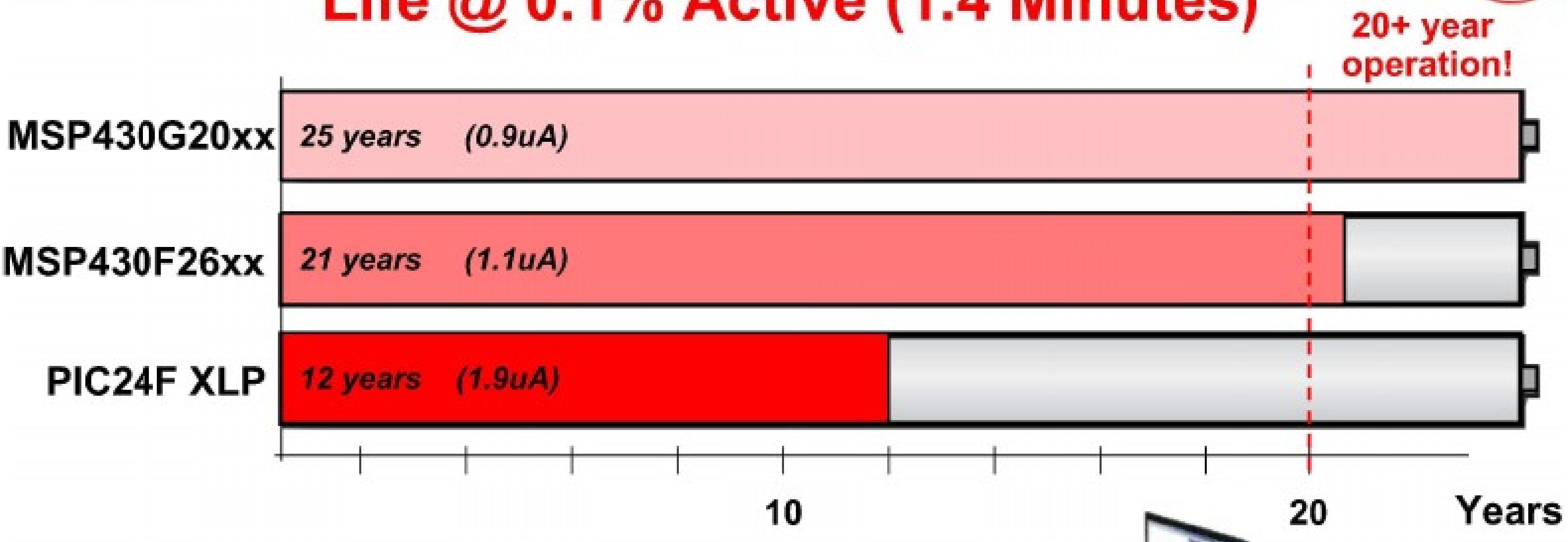
5 10 **Years**

Example: Portable measurement system

- Active power consumption is important in this example
- Average = Standby*(99%) + Active*(1%)
- Used peripherals will impact total current consumption

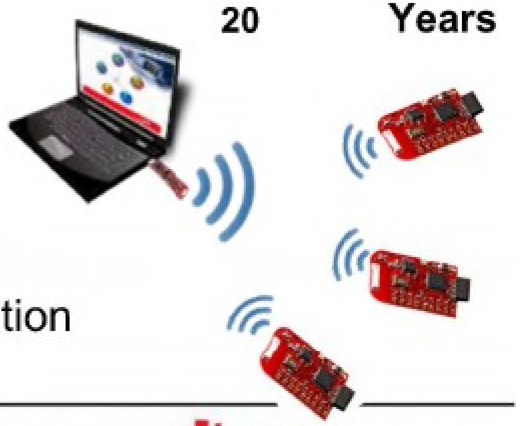
ARROW ELECTRONICS AND TEXAS INSTRUMENTS

Average Current Consumption & Battery Life @ 0.1% Active (1.4 Minutes)



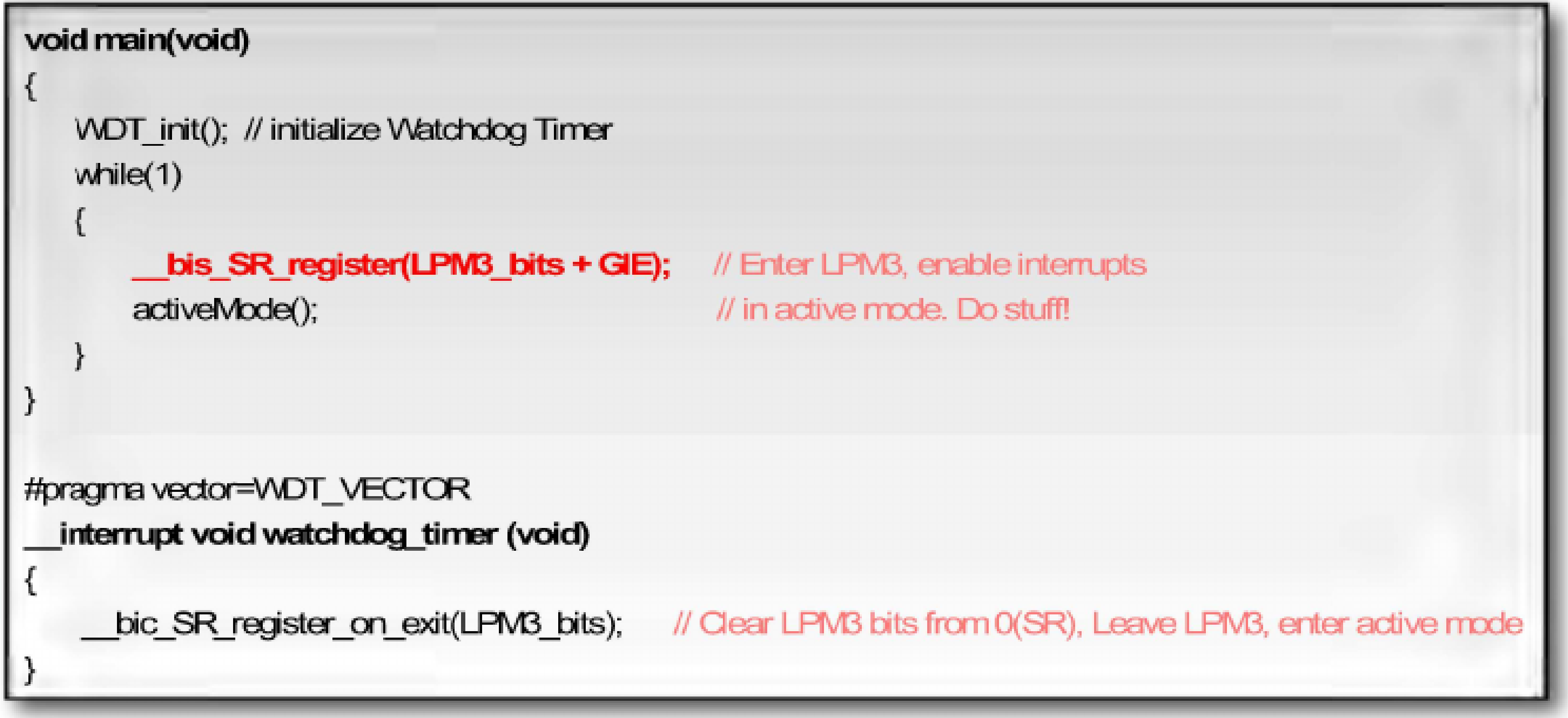
Example: Wireless sensor network

- Standby & Active power are equally important
- Average = Standby*(99.9%) + Active*(0.1%)
- Used peripherals will impact total current consumption



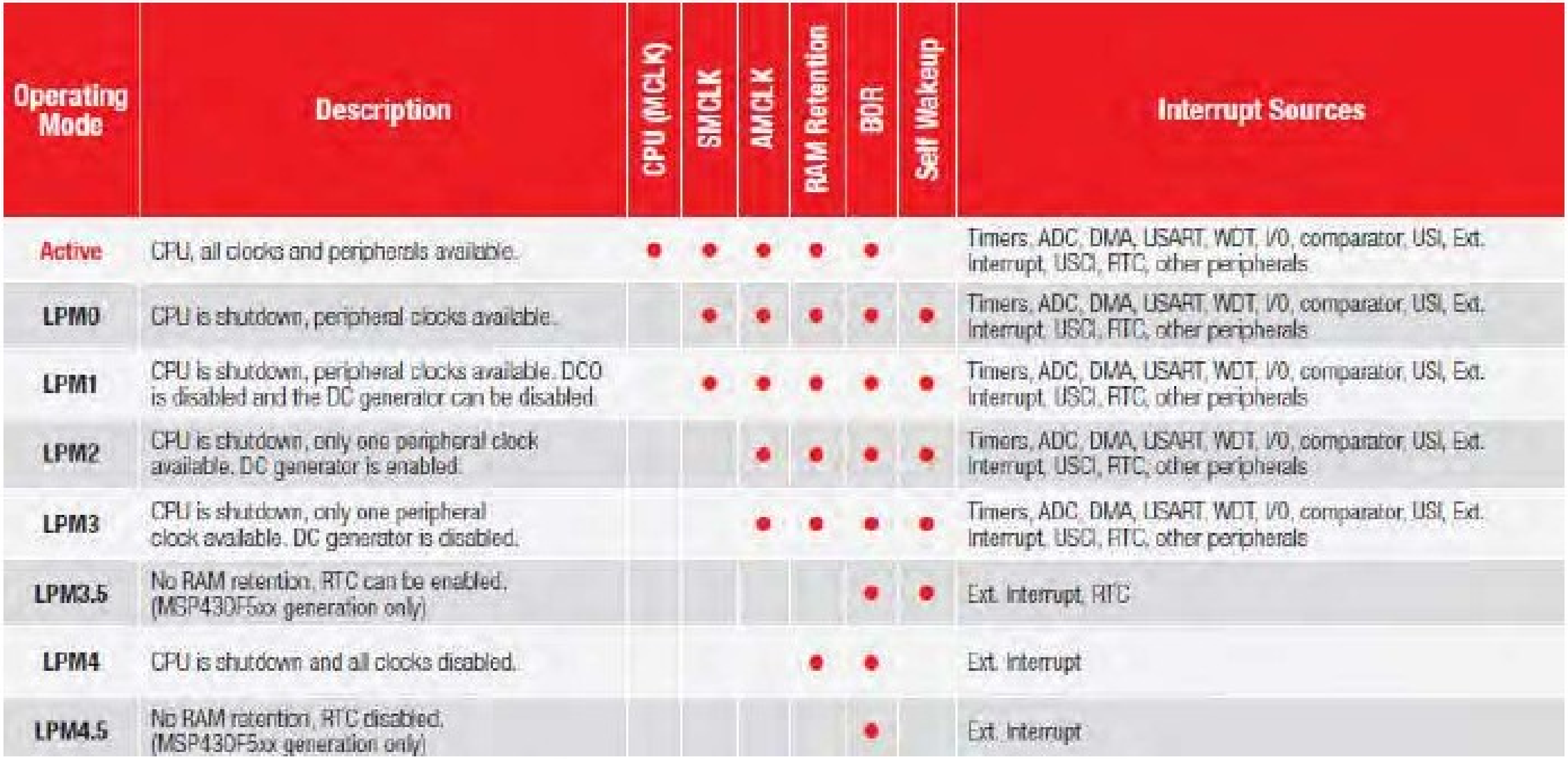
ULP is Easy!

- Using our Low Power Modes are easy
- Enter low power mode with *1 line of code!*



```
void main(void)
{
    WDT_init();  // initialize Watchdog Timer
    while(1)
    {
        __bis_SR_register(LPM3_bits + GIE);     // Enter LPM3, enable interrupts
        activeMode();                            // in active mode. Do stuff!
    }
}

#pragma vector=WDT_VECTOR
__interrupt void watchdog_timer (void)
{
   __bic_SR_register_on_exit(LPM3_bits);      // Clear LPM3 bits from 0(SR), Leave LPM3, enter active mode
}
```

Low Power Mode Overview



Operating Mode	Description	CPU (MCLK)	SMCLK	AMCLK	RAM Retention	BOR	Self Wakeup	Interrupt Sources
Active	CPU, all clocks and peripherals available.	●	●	●	●	●		Timers, ADC, DMA, USART, WDT, I/0, comparator, USI, Ext. Interrupt, USCI, RTC, other peripherals
LPM0	CPU is shutdown, peripheral clocks available.		●	●	●	●	●	Timers, ADC, DMA, USART, WDT, I/0, comparator, USI, Ext. Interrupt, USCI, RTC, other peripherals
LPM1	CPU is shutdown, peripheral clocks available. DCO is disabled and the DC generator can be disabled.		●	●	●	●	●	Timers, ADC, DMA, USART, WDT, I/0, comparator, USI, Ext. Interrupt, USCI, RTC, other peripherals
LPM2	CPU is shutdown, only one peripheral clock available. DC generator is enabled.			●	●	●	●	Timers, ADC, DMA, USART, WDT, I/0, comparator, USI, Ext. Interrupt, USCI, RTC, other peripherals
LPM3	CPU is shutdown, only one peripheral clock available. DC generator is disabled.			●	●	●	●	Timers, ADC, DMA, USART, WDT, I/0, comparator, USI, Ext. Interrupt, USCI, RTC, other peripherals
LPM3.5	No RAM retention, RTC can be enabled. (MSP430F5xx generation only)					●	●	Ext. Interrupt, RTC
LPM4	CPU is shutdown and all clocks disabled.				●	●		Ext. Interrupt
LPM4.5	No RAM retention, RTC disabled. (MSP430F5xx generation only)					●		Ext. Interrupt

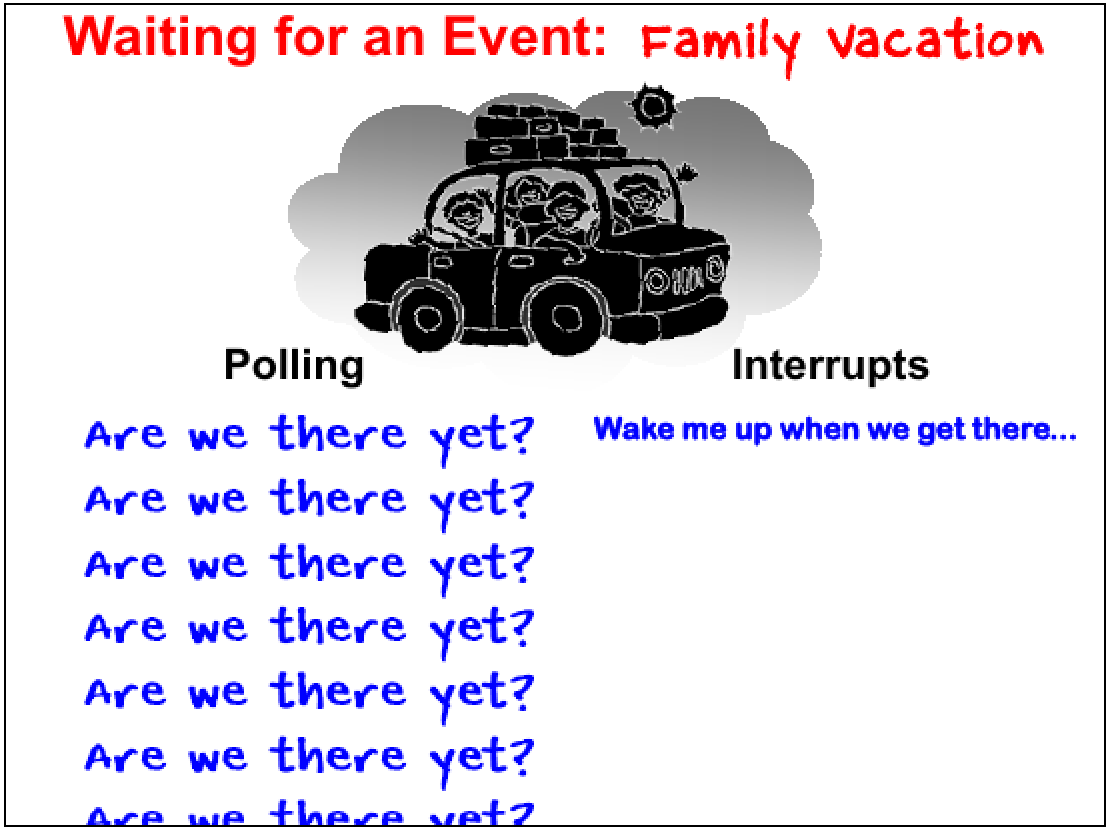
Ok, so we enter a Low Power Mode which shuts down parts of the processor.

So how do we wake up from the Low Power Mode?

Easy, use a hardware Interrupt to restart the processor and execute the Interrupt Service Routine – ISR

Interrupts hardware are built directly into the processor – each device block has this hardware

What is an Interrupt?



Waiting for an Event: Family Vacation

Polling

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?

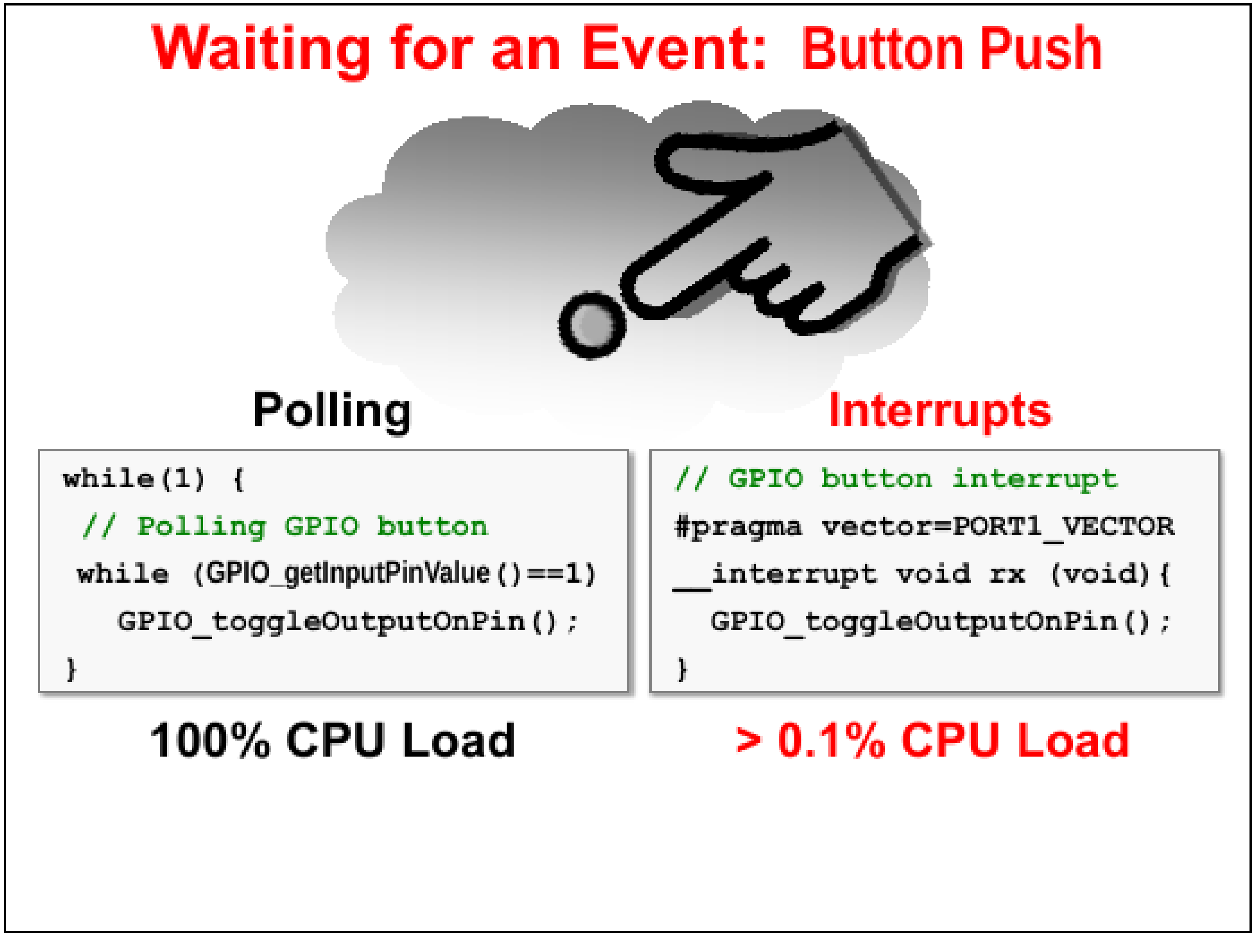
Interrupts

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.

http://processors.wiki.ti.com/index.php/MSP430_Design_Workshop

It is common to see “simple” example code utilize **Polling**. As you can see from the left-side example below, this can simply consist of a while{} loop that keeps repeating until a button-push is detected. The big downfall here, though, is that the processor is constantly running– asking the question, “Has the button been pushed, yet?”



Waiting for an Event: Button Push

Polling

```
while(1) {
 // Polling GPIO button
 while (GPIO_getInputPinValue()==1)
   GPIO_toggleOutputOnPin();
}
```

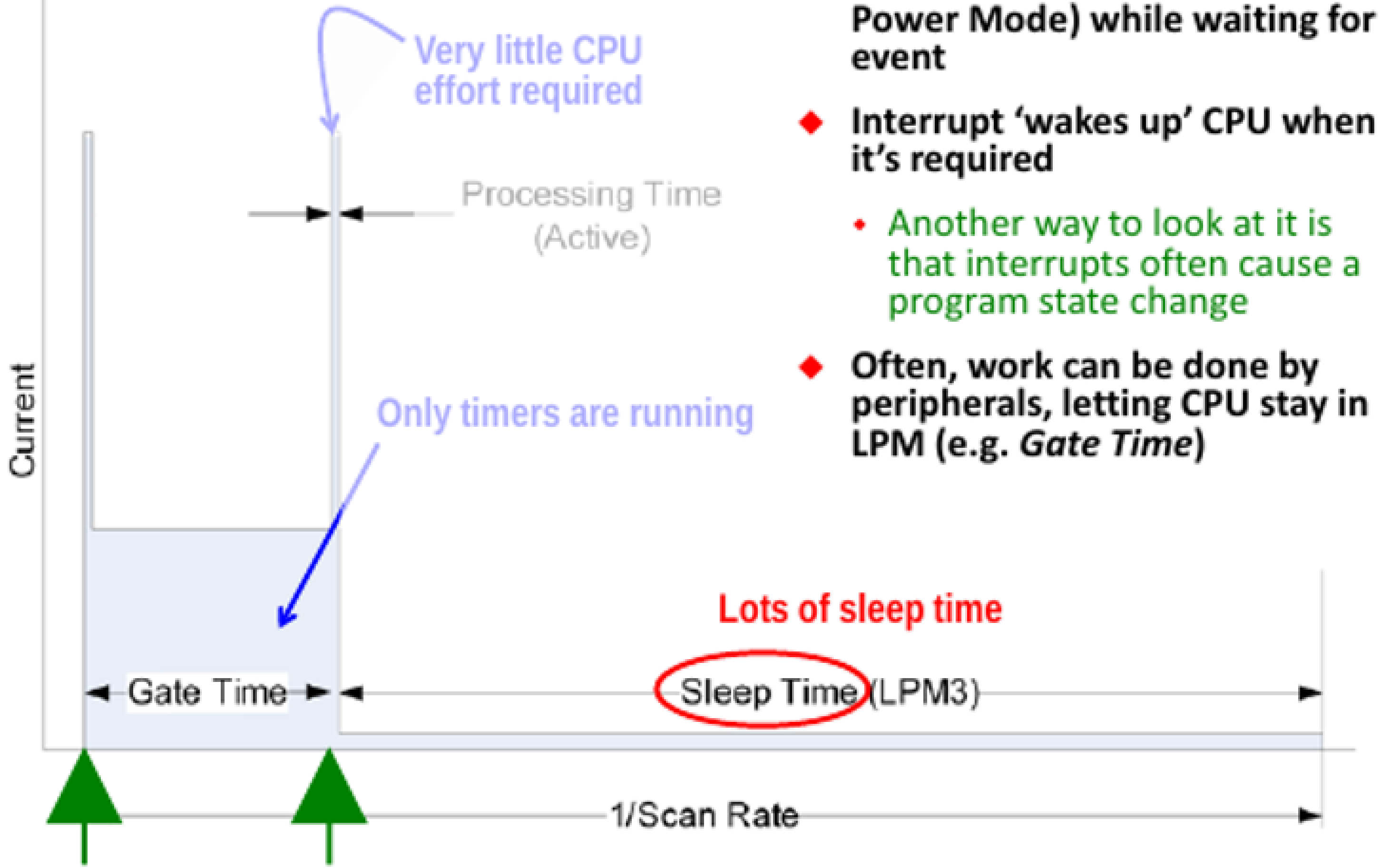
100% CPU Load

Interrupts

```
// GPIO button interrupt
#pragma vector=PORT1_VECTOR
__interrupt void rx (void){
  GPIO_toggleOutputOnPin();
}
```

> 0.1% CPU Load

Interrupts Help Support Ultra Low Power



- Keep CPU asleep (i.e. in Low Power Mode) while waiting for event
- Interrupt 'wakes up' CPU when it's required
  - Another way to look at it is that interrupts often cause a program state change
- Often, work can be done by peripherals, letting CPU stay in LPM (e.g. *Gate Time*)

Foreground / Background Scheduling

```
main() {
    //Init
    initPMM();
    initClocks();
    ...

    while(1){
       background
       or LPMx
    }
}
```

```
ISR1
   get data
   process
```

```
ISR2
   set a flag
```

System Initialization

- The beginning part of main() is usually dedicated to setting up your system (Chapters 3 and 4)

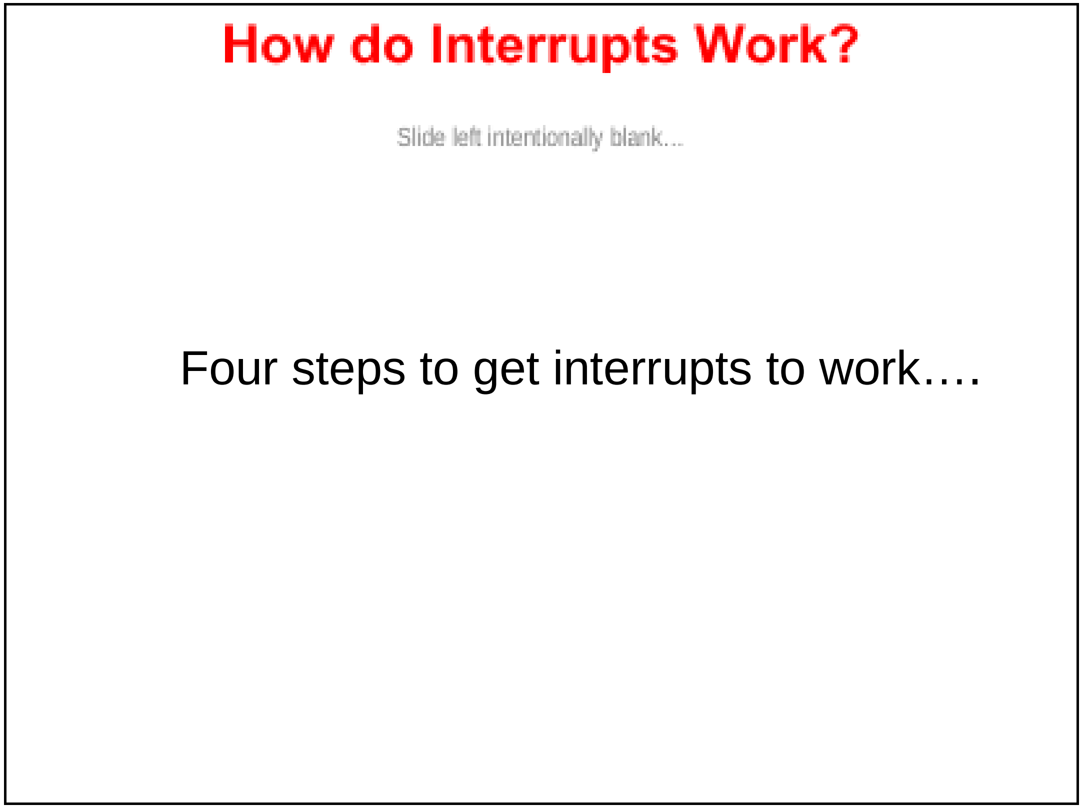
Background

- Most systems have an endless loop that runs 'forever' in the background
- In this case, 'Background' implies that it runs at a lower priority than 'Foreground'
- In MSP430 systems, the background loop often contains a Low Power Mode (LPMx) command – this sleeps the CPU/System until an interrupt event wakes it up

Foreground

- Interrupt Service Routine (ISR) runs in response to enabled hardware interrupt
- These events may change modes in Background – such as waking the CPU out of low-power mode
- ISR's, by default, are not interruptible
- Some processing may be done in ISR, but it's usually best to keep them short

Now that we have a rough understanding of what interrupts are used for, let's discuss what mechanics are needed to make them work. Hint, there are 4 steps to getting interrupts to work…



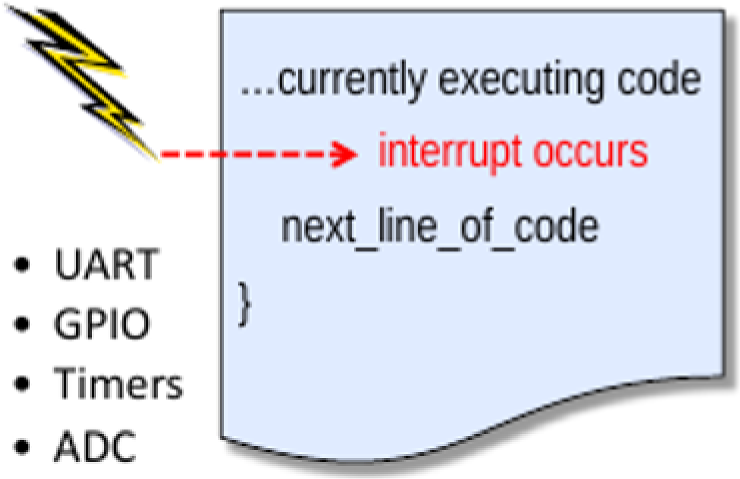
How do Interrupts Work?

Slide left intentionally blank…

Four steps to get interrupts to work….

How do Interrupts Work?

1. **An interrupt occurs**

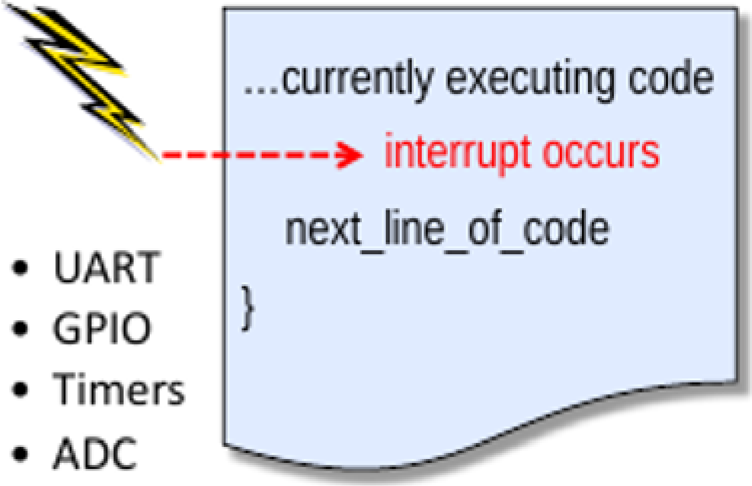


```
...currently executing code
      interrupt occurs
   next_line_of_code
}
```

- UART
- GPIO
- Timers
- ADC
- Etc.

How do Interrupts Work?

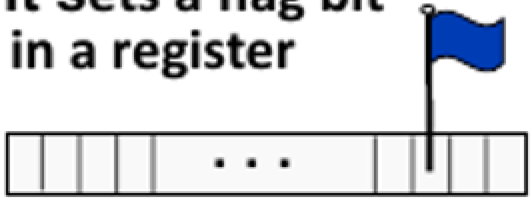
1. **An interrupt occurs**



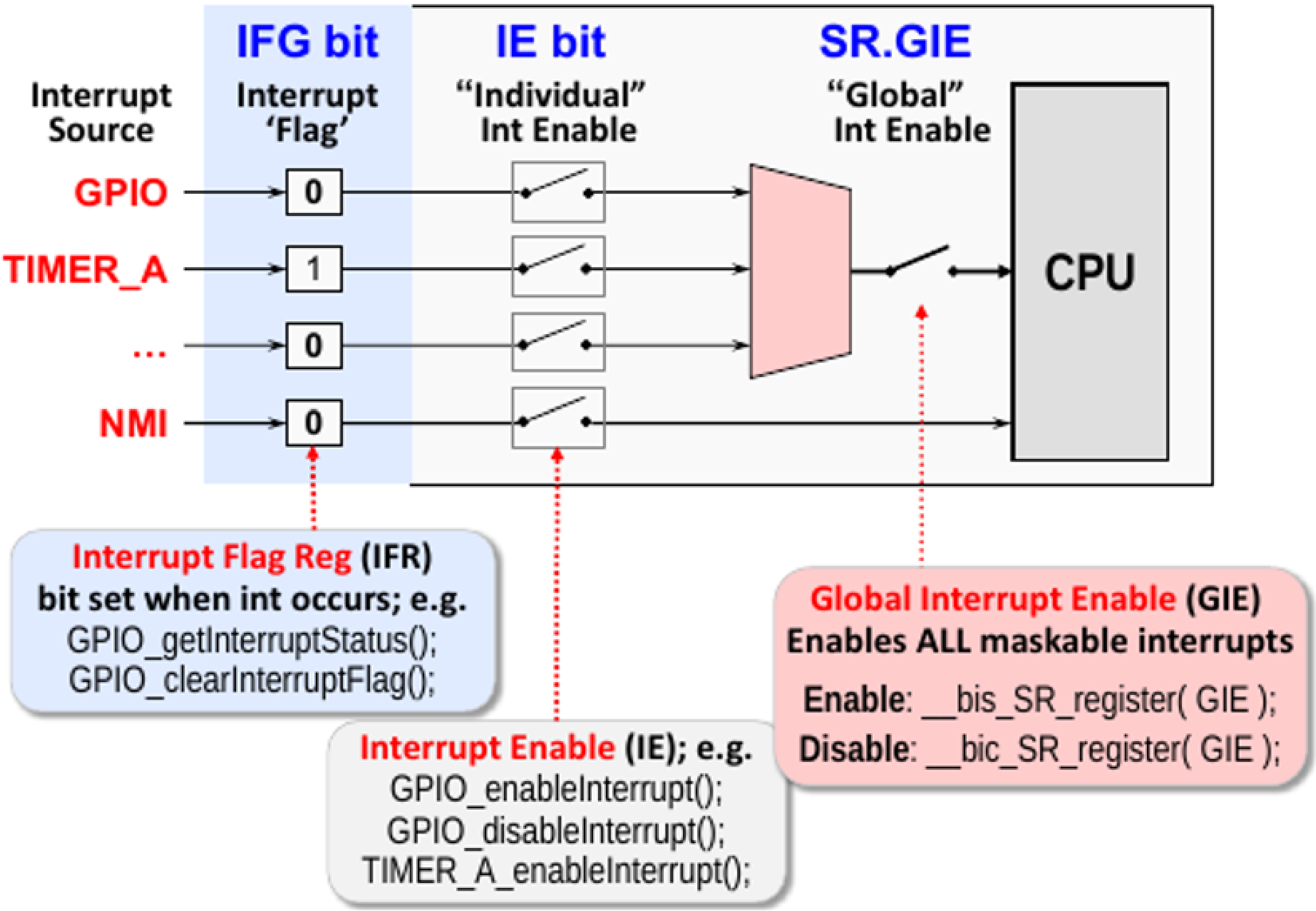
```
...currently executing code
   interrupt occurs
   next_line_of_code
}
```

- UART
- GPIO
- Timers
- ADC
- Etc.

2. **It sets a flag bit in a register**



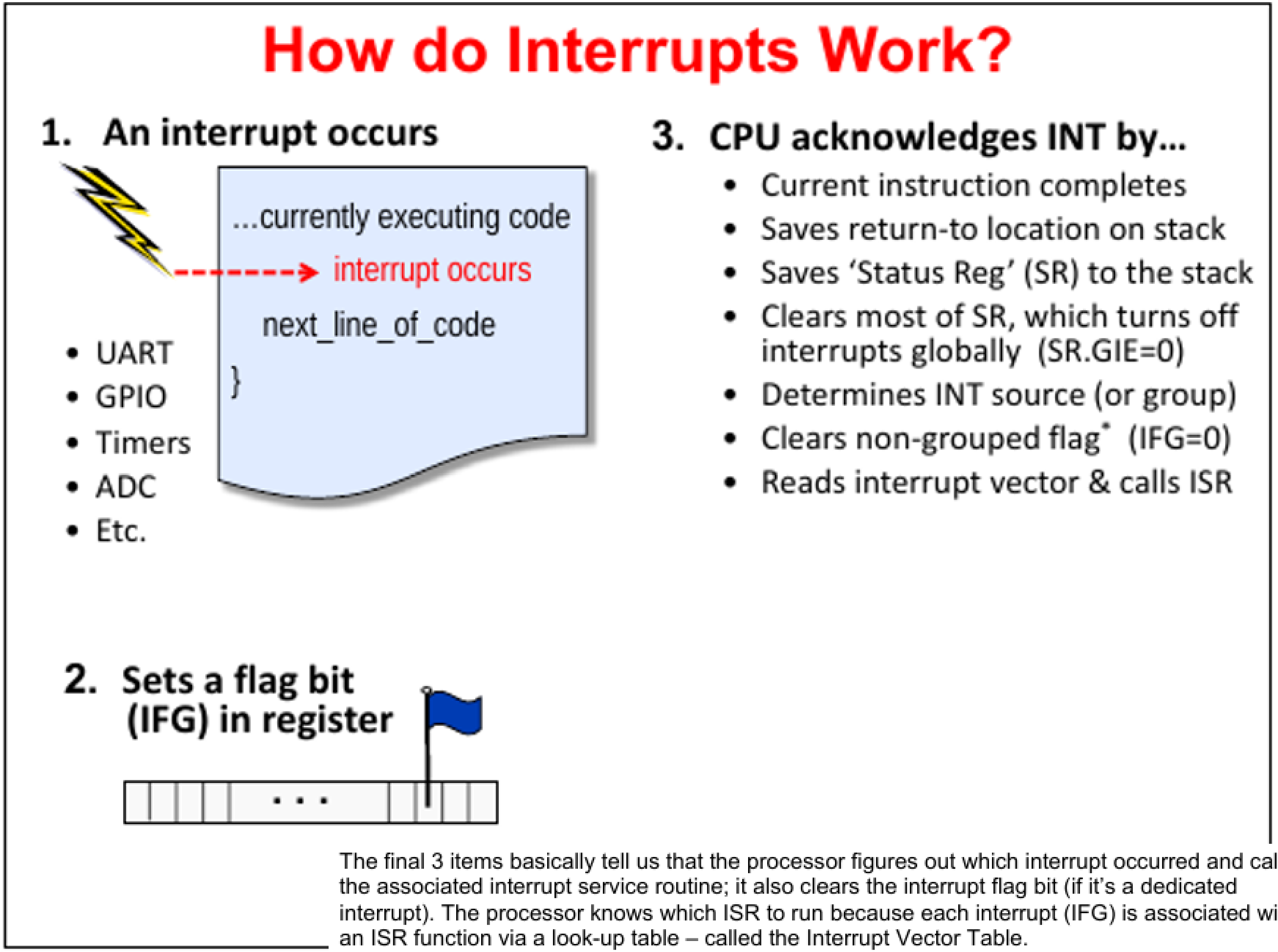
Interrupt Flow



Interrupt Flag Reg (IFR)
bit set when int occurs; e.g.
GPIO_getInterruptStatus();
GPIO_clearInterruptFlag();

Interrupt Enable (IE); e.g.
GPIO_enableInterrupt();
GPIO_disableInterrupt();
TIMER_A_enableInterrupt();

Global Interrupt Enable (GIE)
Enables ALL maskable interrupts
Enable: __bis_SR_register(GIE);
Disable: __bic_SR_register(GIE);



How do Interrupts Work?

1. An interrupt occurs

```
...currently executing code
   interrupt occurs
   next_line_of_code
}
```

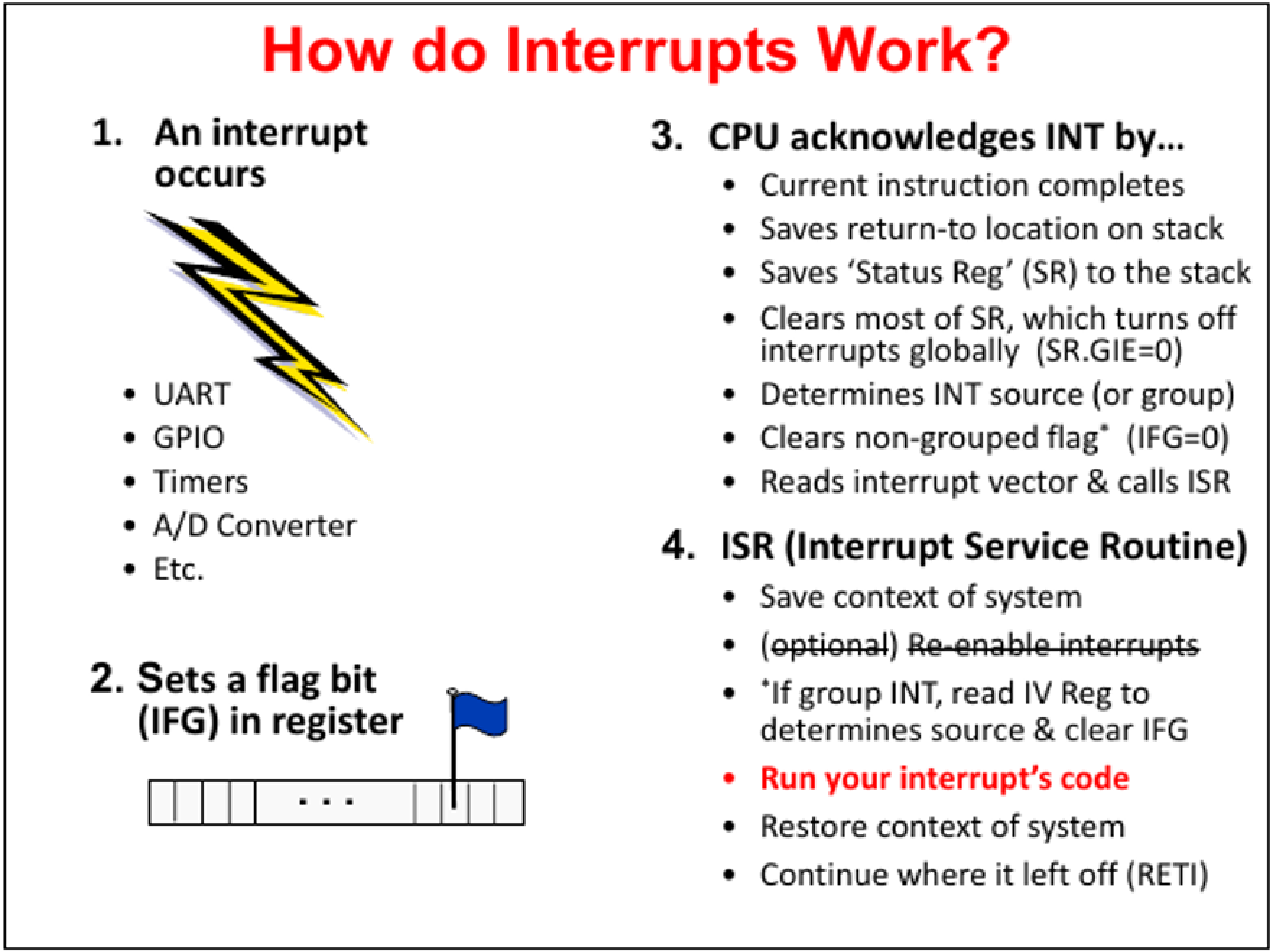
- UART
- GPIO
- Timers
- ADC
- Etc.

2. Sets a flag bit (IFG) in register

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

The final 3 items basically tell us that the processor figures out which interrupt occurred and calls the associated interrupt service routine; it also clears the interrupt flag bit (if it's a dedicated interrupt). The processor knows which ISR to run because each interrupt (IFG) is associated with an ISR function via a look-up table – called the Interrupt Vector Table.



How do Interrupts Work?

1. An interrupt occurs

- UART
- GPIO
- Timers
- A/D Converter
- Etc.

2. Sets a flag bit (IFG) in register

3. CPU acknowledges INT by…

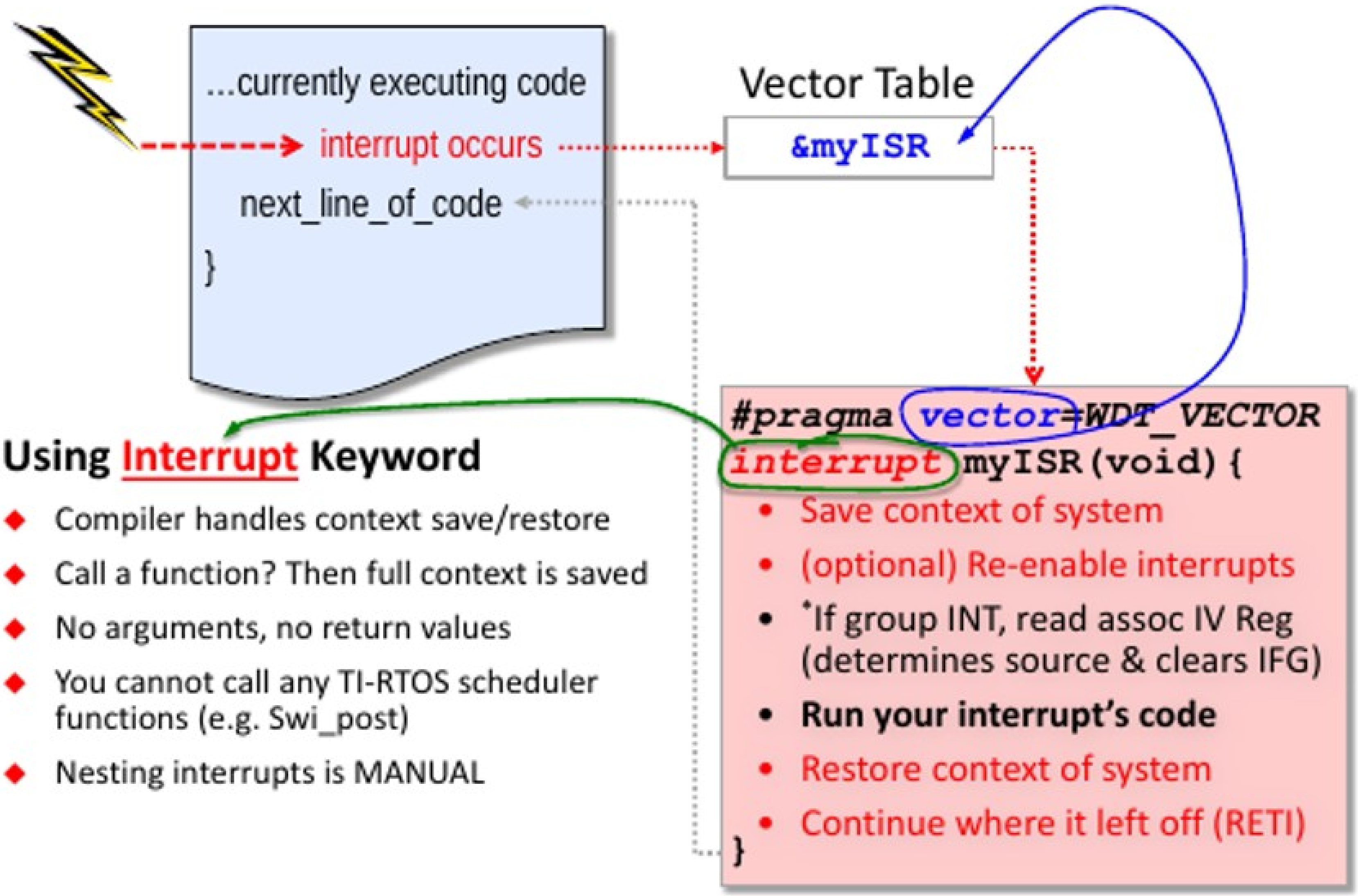
- Current instruction completes
- Saves return-to location on stack
- Saves 'Status Reg' (SR) to the stack
- Clears most of SR, which turns off interrupts globally (SR.GIE=0)
- Determines INT source (or group)
- Clears non-grouped flag* (IFG=0)
- Reads interrupt vector & calls ISR

4. ISR (Interrupt Service Routine)

- Save context of system
- (~~optional~~) ~~Re-enable interrupts~~
- *If group INT, read IV Reg to determines source & clear IFG
- Run your interrupt's code
- Restore context of system
- Continue where it left off (RETI)

An *interrupt service routine* (ISR), also called an *interrupt handler*, is the code you write that will be run when a hardware interrupt occurs. Your ISR code must perform whatever task you want to execute in response to the interrupt, but without adversely affecting the threads (i.e. code) already running in the system.

4. Interrupt Service Routine (ISR)



```
...currently executing code
  interrupt occurs
  next_line_of_code
}
```

Vector Table

`&myISR`

```
#pragma vector=WDT_VECTOR
interrupt myISR(void){
```

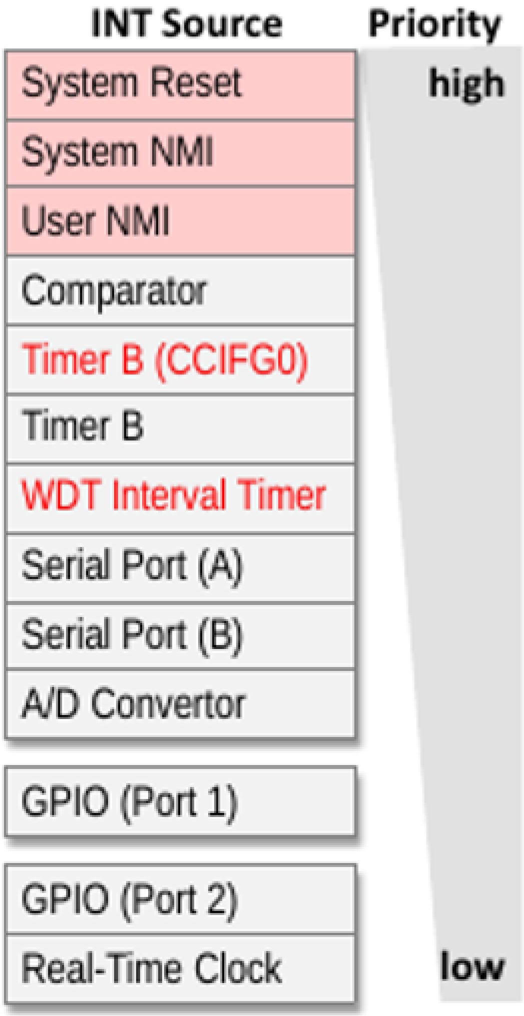
- Save context of system
- (optional) Re-enable interrupts
- *If group INT, read assoc IV Reg (determines source & clears IFG)
- **Run your interrupt's code**
- Restore context of system
- Continue where it left off (RETI)

```
}
```

Using Interrupt Keyword

- Compiler handles context save/restore
- Call a function? Then full context is saved
- No arguments, no return values
- You cannot call any TI-RTOS scheduler functions (e.g. Swi_post)
- Nesting interrupts is MANUAL

Interrupt Priorities (F5529)



INT Source	Priority
System Reset	high
System NMI	
User NMI	
Comparator	
Timer B (CCIFG0)	
Timer B	
WDT Interval Timer	
Serial Port (A)	
Serial Port (B)	
A/D Convertor	
GPIO (Port 1)	
GPIO (Port 2)	
Real-Time Clock	low

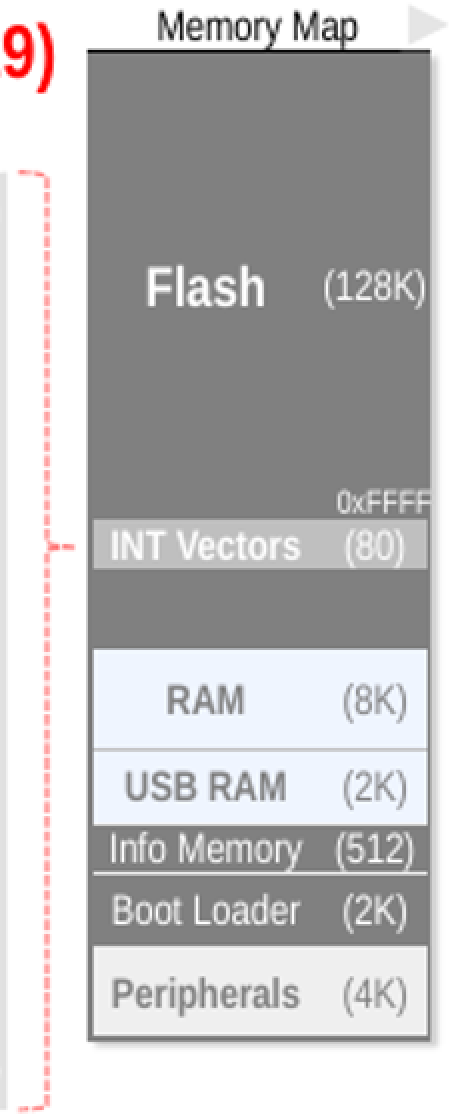
- **There are 23 interrupts**
  (partially shown here)
- **If multiple interrupts (of the 23) are pending, the highest priority is responded to first**
- **By default, interrupts are not nested …**
  - That is, unless you re-enable INT's during your ISR, other interrupts will be held off until it completes
  - It doesn't matter if the new INT is a higher priority
  - As already recommended, you should keep your ISR's short
- **Most of these represent 'groups' of interrupt source flags**
  - 145 IFG's map into these 23 interrupts

Interrupt Vectors & Priorities (F5529)

INT Source	IV Register	Vector Address	Loc'n	Priority
System Reset	SYSRSTIV	RESET_VECTOR	63	high
System NMI	SYSSNIV	SYSNMI_VECTOR	62	
User NMI	SYSUNIV	UNMI_VECTOR	61	
Comparator	CBIV	COMP_B_VECTOR	60	
Timer B (CCIFG0)	CCIFG0	TIMER0_B0_VECTOR	59	
Timer B	TB0IV	TIMER0_B1_VECTOR	58	
WDT Interval Timer	WDTIFG	WDT_VECTOR	57	
Serial Port (A)	UCA0IV	USCI_A0_VECTOR	56	
Serial Port (B)	UCB0IV	USCI_B0_VECTOR	55	
A/D Convertor	ADC12IV	ADC12_VECTOR	54	
GPIO (Port 1)	P1IV	PORT1_VECTOR	47	
GPIO (Port 2)	P12V	PORT2_VECTOR	42	
Real-Time Clock	RTCIV	RTC_VECTOR	41	low

Legend:

Non-maskable	Group'd IFG bits
Maskable	Dedicated IFG bits

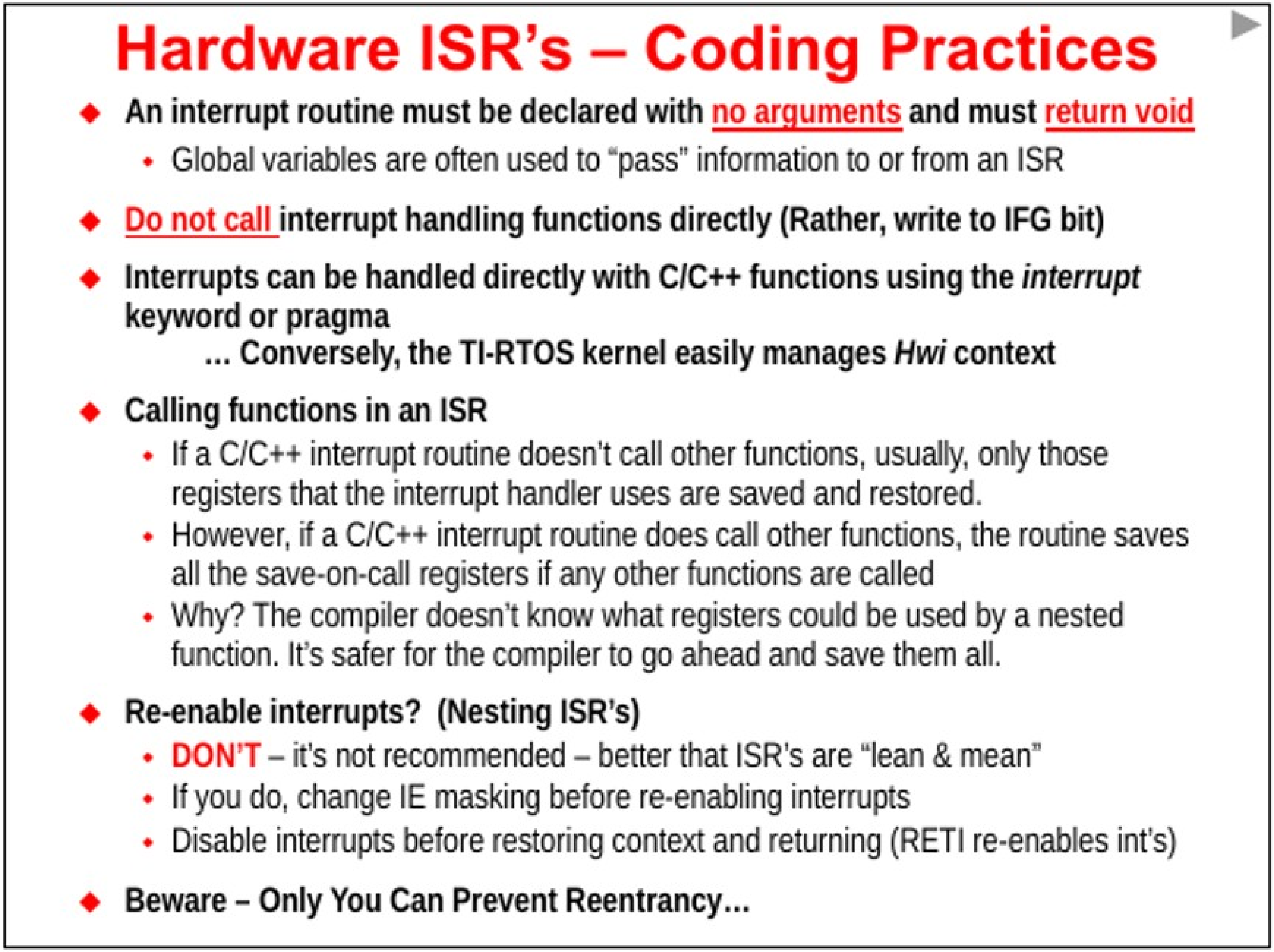


Interrupt Service Routine (Dedicated INT)

INT Source	IV Register	Vector Address	Loc'n
WDT Interval Timer	WDTIFG	WDT_VECTOR	57

- **#pragma vector** assigns **'myISR'** to correct location in vector table
- **__interrupt** keyword tells compiler to save/restore context and `RETI`
- For a dedicated interrupt, the MSP430 CPU auto clears the WDTIFG flag

```
#pragma vector=WDT_VECTOR
__interrupt void myWdtISR(void) {

   GPIO_toggleOutputOnPin( ... );

}
```



Hardware ISR's – Coding Practices

- **An interrupt routine must be declared with no arguments and must return void**
  - Global variables are often used to “pass" information to or from an ISR
- **Do not call interrupt handling functions directly (Rather, write to IFG bit)**
- **Interrupts can be handled directly with C/C++ functions using the *interrupt* keyword or pragma**

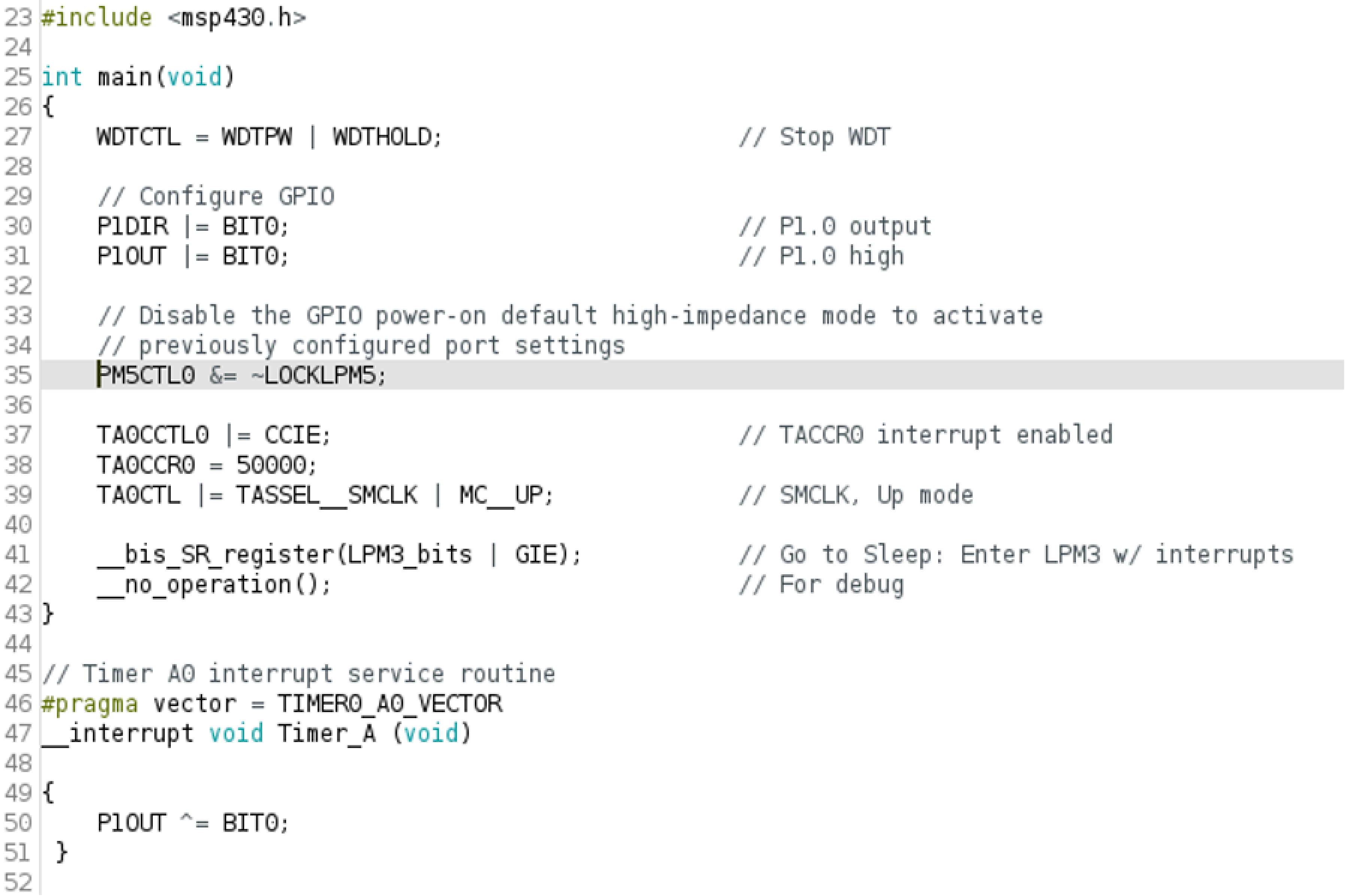
  ... Conversely, the TI-RTOS kernel easily manages *Hwi* context
- **Calling functions in an ISR**
  - If a C/C++ interrupt routine doesn't call other functions, usually, only those registers that the interrupt handler uses are saved and restored.
  - However, if a C/C++ interrupt routine does call other functions, the routine saves all the save-on-call registers if any other functions are called
  - Why? The compiler doesn't know what registers could be used by a nested function. It's safer for the compiler to go ahead and save them all.
- **Re-enable interrupts? (Nesting ISR's)**
  - **DON'T** – it's not recommended – better that ISR's are “lean & mean”
  - If you do, change IE masking before re-enabling interrupts
  - Disable interrupts before restoring context and returning (RETI re-enables int's)
- **Beware – Only You Can Prevent Reentrancy…**

The Code is simpler than all the details

Take a look at blinking the LED using a timer

How simple can it get?

Toggle LED using Timer Interrupt Service Routine



```c
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;                     // Stop WDT

    // Configure GPIO
    P1DIR |= BIT0;                                // P1.0 output
    P1OUT |= BIT0;                                // P1.0 high

    // Disable the GPIO power-on default high-impedance mode to activate
    // previously configured port settings
    PM5CTL0 &= ~LOCKLPM5;

    TA0CCTL0 |= CCIE;                             // TACCR0 interrupt enabled
    TA0CCR0 = 50000;
    TA0CTL |= TASSEL__SMCLK | MC__UP;             // SMCLK, Up mode

    __bis_SR_register(LPM3_bits | GIE);           // Go to Sleep: Enter LPM3 w/ interrupts
    __no_operation();                             // For debug
}

// Timer A0 interrupt service routine
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A (void)

{
    P1OUT ^= BIT0;
 }

```

Video of loading and running this code: