

# MSP430 Programming

---

## Introduction

If you've programmed most any mainstream microcontroller, this module shouldn't be too much of a surprise. What may surprise you is just how little power the MSP430 can draw when the programmer makes some informed decisions. Most microcontrollers operate in a real-time environment and respond to either interrupts or timers. The MSP430 is ideally suited to just this programming model. It can wake quickly, compute quickly and sleep deeply to save power and has a wide selection of highly capable and low power peripherals.

## Learning Objectives

- MSP430 Products
- Power Efficient Applications
- Operating Modes
- Clock Management
- Peripherals
- Timer
- Coding Tips

\*\*\* always wear sunscreen \*\*\*

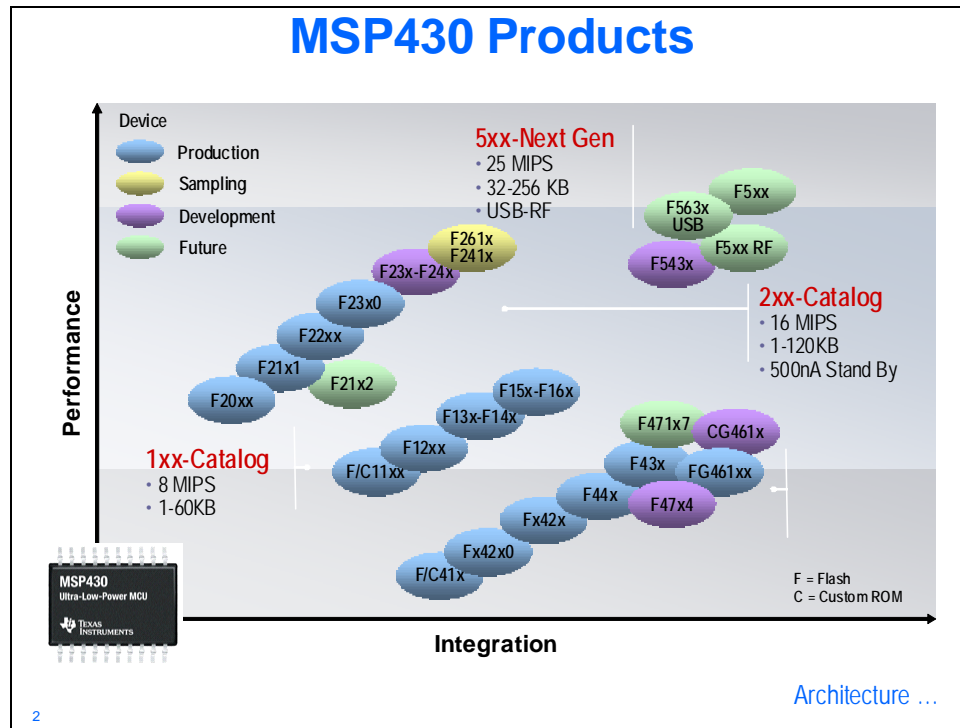
---

# Module Topics

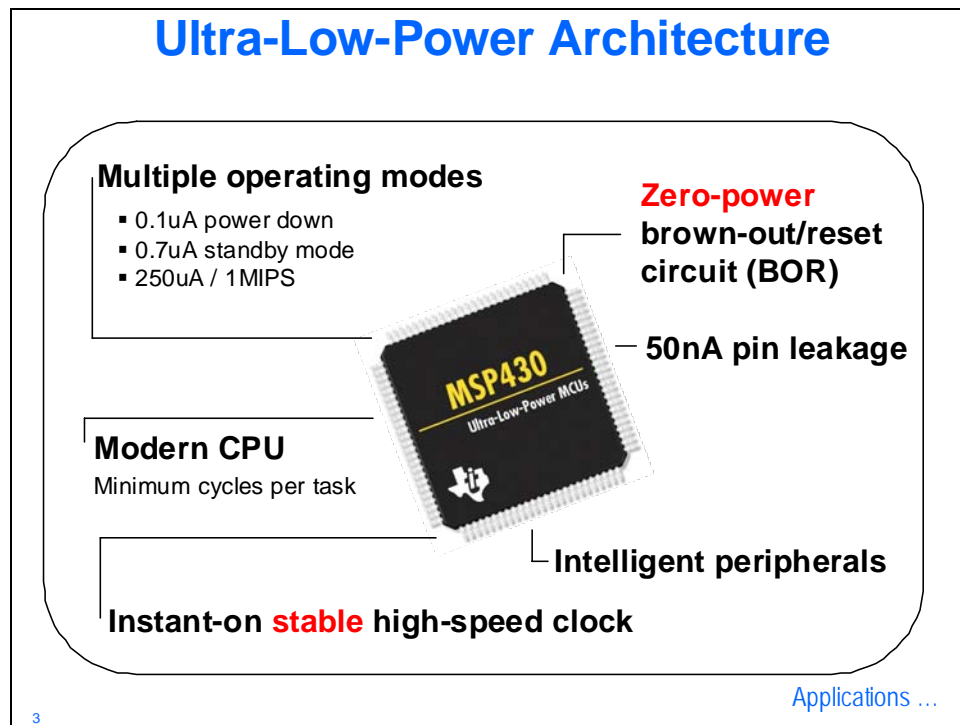
<b>MSP430 Programming.....</b>	<b>3-1</b>
<i>Module Topics.....</i>	<i>3-3</i>
<i>MSP430 Products and Architecture.....</i>	<i>3-5</i>
<i>Power Efficient Applications and Power Modes.....</i>	<i>3-6</i>
<i>Clock Management .....</i>	<i>3-7</i>
<i>Hints and Tips.....</i>	<i>3-8</i>
<i>Timers .....</i>	<i>3-11</i>
<i>Summary .....</i>	<i>3-13</i>
<i>Lab3 – MSP430 Programming .....</i>	<i>3-15</i>
Description: .....	3-15
Hardware list: .....	3-16
Software list:.....	3-16
Procedure.....	3-17

\*\*\* I've seen an MSP430 run on a couple of grapes \*\*\*

# MSP430 Products and Architecture



2

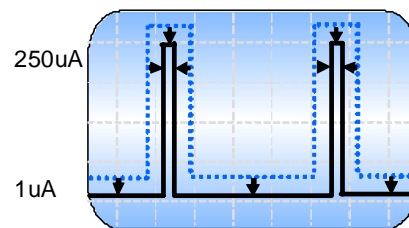
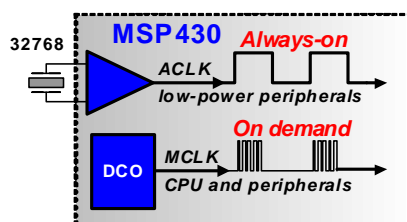


3

# Power Efficient Applications and Power Modes

## Design Power-Efficient Applications

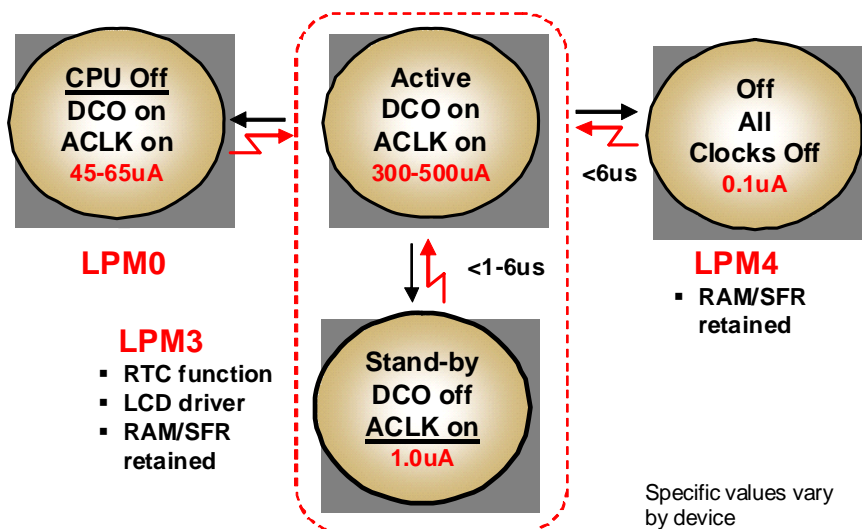
- ◆ Power-efficient MSP430 apps:
  - ◆ Minimize the instantaneous current draw
  - ◆ Maximize the time spent in low-power modes
- ◆ The MSP430 is inherently low-power, but your design has a big impact on power efficiency
- ◆ Proper low-power design techniques make the difference



Operating modes ...

4

## Operating Modes

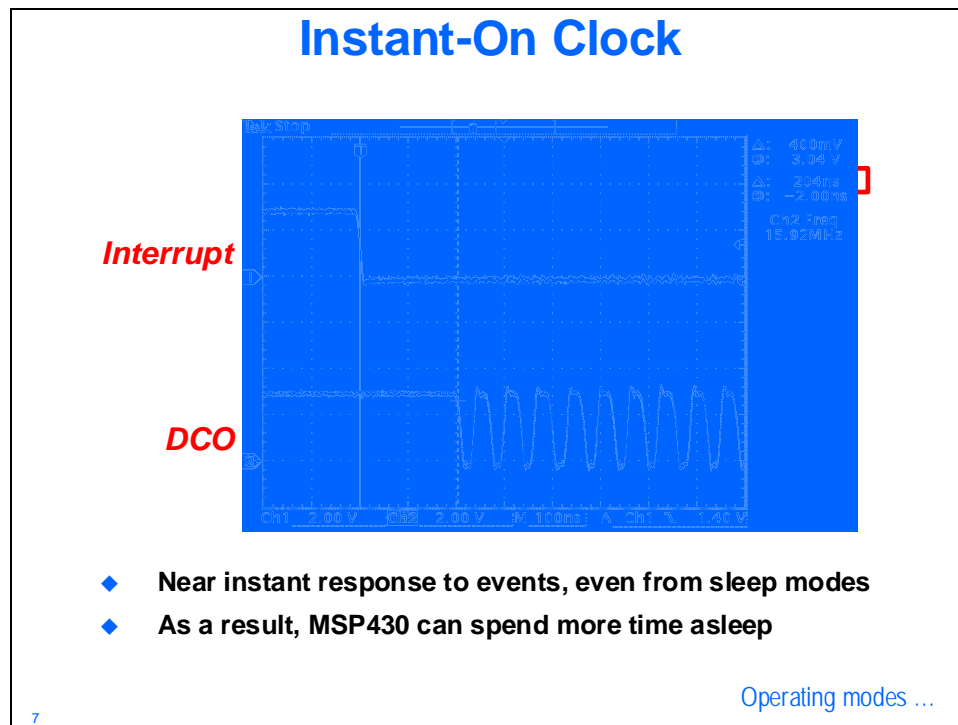
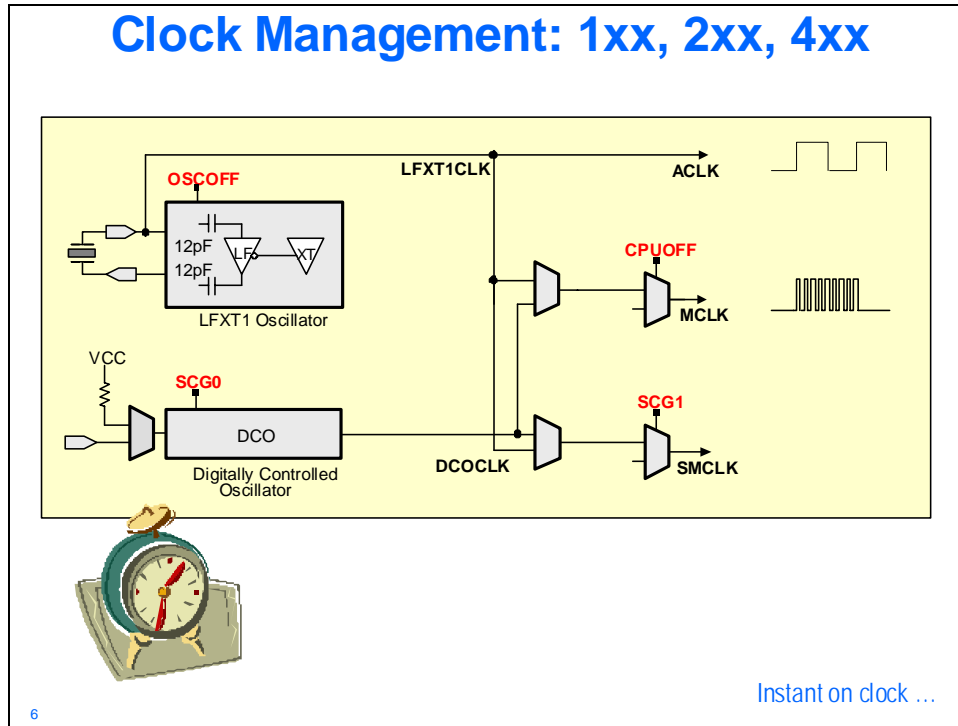


Specific values vary by device

Clock Management ...

5

# Clock Management



## Hints and Tips

### Using MSP430 Operating Modes

- ◆ Maximize time spent in low-power modes
  - Set up interrupt handling and then go to sleep
- ◆ Use ACLK for peripherals
  - Allows use of LPM3 instead of LPM0
- ◆ Only activate peripherals while used, disable when finished

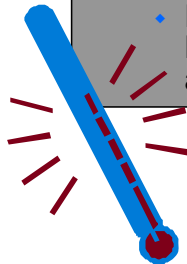


Effects of temperature ...

8

### Effects of Vcc / MCLK / Temperature

- ◆ Power draw increases with...
  - Vcc
  - CPU clock speed (MCLK)
  - Temperature
- ◆ Slowing MCLK reduces instantaneous power, but usually increases active duty cycle
  - Power savings nullified – best to use default MCLK (or increase it if required for application performance)

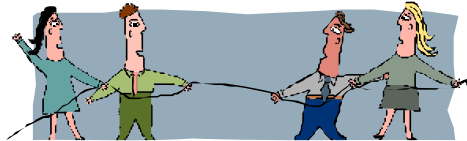


Unused I/Os ...

9

## Configuring Unused I/Os

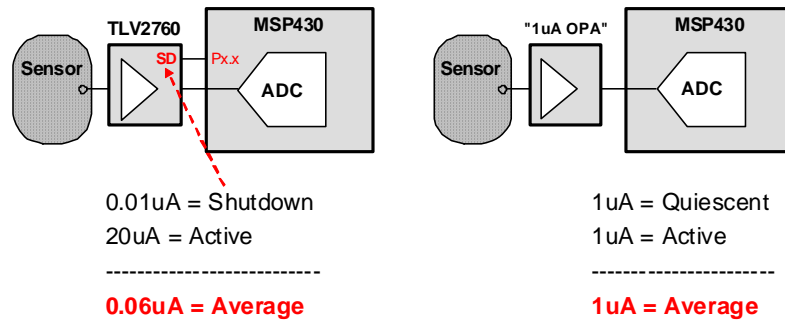
- ◆ Port I/Os should be...
  - ◆ Driven as outputs
  - ◆ Be driven at Vcc/ground by an external device
  - ◆ Have a pull-up/down resistor



External devices ...

10

## Power Manage External Devices



- ◆ External op amp with shutdown can be 20x lower total power

Move functions to peripherals ...

11

## Move Functions to Peripherals

- ◆ Peripherals use less current than CPU
- ◆ Delegating to them allows CPU to shut down, saving system power
- ◆ “Intelligent” peripherals are more capable, providing more opportunity for CPU shutoff
- ◆ Use DMA for repetitive data handling rather than CPU load/store

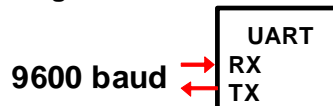


Reduce Cycles ...

12

## Reduce Cycles

- ◆ CPU active time is a direct function of how many cycles need to be executed
- ◆ Reducing cycles is key to maximizing the use of low-power modes
- ◆ Many ways to do this, but an important one is interrupt-driven coding



```
// Polling UART Receive
For(;;)
{
  while (!(IFG2&URXIFG0));
  TXBUF0 = RXBUF0;
}
```

100% CPU Load

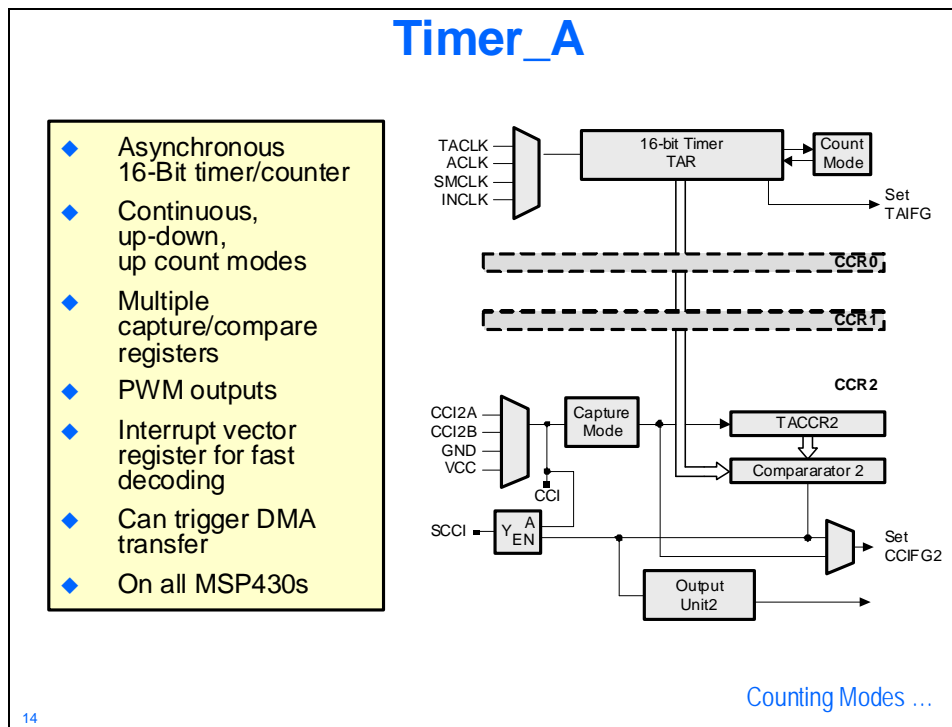
```
// UART Receive Interrupt
#pragma vector=UART_VECTOR
__interrupt void rx (void)
{
  TXBUF0 = RXBUF0;
}
```

0.1% CPU Load

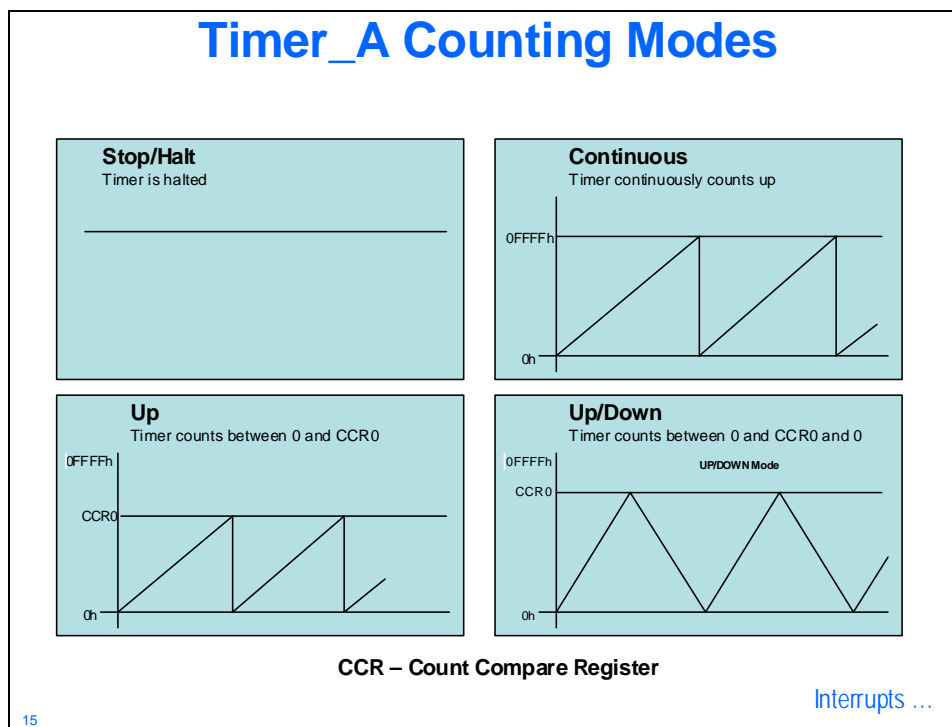
Timer\_A ...

13

# Timers



14



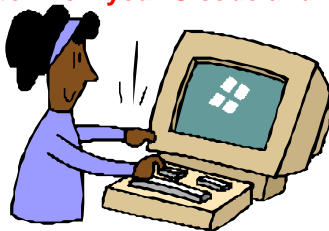
15



## Summary

### C Coding Tips

- ◆ Use local variable as much as possible. Local variables use CPU registers where global variables use RAM.
- ◆ Use bit mask instead of bit fields for unsigned int and unsigned char.
- ◆ Use unsigned data types where possible
- ◆ Use pointers to access structures and unions
- ◆ Use “static const” class to avoid run-time copying of structures, unions, and arrays.
- ◆ Avoid modulo
- ◆ Count down “for” loops
- ◆ **Get to know your C code and its disassembly**



18

[Summary ...](#)

### Summary

- ◆ Proper system design is necessary for the best low-power performance
- ◆ Maximize power efficiency by minimizing program duty cycle
- ◆ Make good use of all power modes
- ◆ Reduce program cycles



19

[Lab Time ...](#)

\*\*\* this page isn't really blank, you know \*\*\*




## Lab3 – MSP430 Programming

### Description:

Let's take our previous, energy-inefficient lab from Lab2 and see what we can do to it to make it draw less power.

### Lab3 – MSP430 Programming

- ◆ **Measure current draw of application**
- ◆ **Apply low power design techniques to reduce current**



20

## Hardware list:

- ✓ 3 eZ430-RF2500 Target Boards
- ✓ 2 Battery Modules
- ✓ 4 AAA Batteries
- ✓ 1 eZ430-RF2500 Emulator Board
- ✓ 1 USB Extender Cable
- ✓ 1 Volt-Ohm-Milliampmeter
- ✓ 1 set of Banana plug to Mini-Clip test leads

## Software list:

- ✓ IAR Embedded Workbench for MSP430 version 4.11D

(You will find shortcuts for the above application on the desktop)

## Procedure

### Measure the Current

#### 1. Baseline

**Reconnect** the **Power jumper** on the battery module (the one we used at the end of the last lab). The LEDs on the attached target board should be flashing again.

Take your Volt-Ohm-Milliameter (**VOM**) and connect the **black** banana plug to the **COM** input and the **red** banana plug to the **VΩmA** input. Remove the Power jumper from the battery module and put it where you'll be able to find it again.

On the VOM, select **20m** under **DCA**. Connect the **black** mini-clip at the other end of the test lead to the battery module pin **nearest** the batteries. Connect the **red** mini-clip to the pin **furthest** away from the batteries. If you manage to get this backwards, the current will merely be negative.

The eZ430-RF2500 target board doesn't just have a MSP430 on it. When connected to the battery, the CC2500 is also powered. When powered up, the CC2500 goes into an idle state and draws about 1.5mA. It is possible to send the device into a sleep state in which it would draw nanoamps, but that would require us to add most of the SimpliciTI software to the project. In the interest of doing things simply, we'll just subtract 1.5mA from our measurements to get the CPU current.

My measurement was about **6.5mA – 1.5mA = 5mA**. Fill in yours below.

If my memory from my 9<sup>th</sup> grade electronics class is still accurate,

**Power (P) = Voltage (E) x Current (I)**

The Voltage is **2.9V** (measure it yourself across the two batteries), so I calculate **14.5mW**. You will be awarded demerits for writing down any values below tenths of a mW.

	Measured mA – 1.5mA	Calculated Power (mW)
S/W loop with LEDs		
S/W loop w/o LEDs		
LPM0 w/o LEDs		
LPM3 w/o LEDs		

**Table 1**

Some of this current is for the LEDs. Let's get rid of that in the next steps.

## 2. Lose those LEDs



With the LEDs blinking, a substantial amount of the current draw could be from them alone. Let's turn them off and see what the MCU draws by itself.

**Start** IAR Embedded Workbench and open the existing workspace **Lab3.eww** in **C:\Texas Instruments\SimplificTI-1.0.6\Projects\Examples\Peer applications\ez430rf\Lab3**. This is simply the project from Lab2 copied over into a new folder. Open main.c for editing.

Comment **out** the three LED control statements and replace each of them with **BSP\_TURN\_OFF\_LED2 ( ) ;** as shown below. This will execute exactly the same number of instructions, but will not light the LEDs.

```
BSP_Init();
// BSP_TURN_ON_LED2();
BSP_TURN_OFF_LED2();
while(1)
{
//   BSP_TOGGLE_LED1();
//   BSP_TOGGLE_LED2();
  BSP_TURN_OFF_LED2();
  BSP_TURN_OFF_LED2();}
```


## 3. Build/Load and Measure

**Build and load** the program by clicking the **Debug** button . When the download is complete, click the **Stop Debugging** button . **Remove** the target board, **attach** it to the battery module and measure the current. I got **4.3mA – 1.5mA = 2.8mA** and calculated **8.1mW**. Enter your values into the table in step 1.

## 4. Using a Timer

Using CPU cycles to provide a delay time is not only wasteful of CPU cycles and power, it's the wrong way to program an MSP430. As an example of how a programmer can use the peripherals to save power, we'll use a timer to provide us with a delay. Every MSP430 has a Timer\_A peripheral, so let's use that one.

I didn't write the code that follows from scratch; I stole it from **slac123**, which is a downloadable set of code example for the MSP430F2274 and others. [www.ti.com/msp430](http://www.ti.com/msp430) is where you can find a ton of example code. Save yourself some time and check it out before you start coding your project at home.

I dropped the important code pieces into a file named **Lab3 Code.txt** so you wouldn't have to type them in. Click the **Open**  button on the menu, navigate to: **C:\Texas Instruments\SimplificTI-1.0.6\Projects\Examples\Peer applications\ez430rf\Lab3**, select **Lab3 Code.txt** and click **Open** to open it for editing.

## 5. LPM0 and Cut/Paste

Add **LPM0**; statements before and after the LED control statements inside the **while()** loop, like below:

Cut/paste the **top** portion of the code from **Lab3 Code.txt** into **main.c** just above the **while(1)** statement. Cut/paste the **middle** portion from **Lab3 Code.txt** into **main.c** at the **end** of the file, like below:

```
volatile unsigned int i;           // volatile to prevent optimization

void main(void)
{
    BSP_Init();                   // Always initialize BSP first
    BSP_TURN_ON_LED2();           // Turn on green LED
    // BSP_TURN_OFF_LED2();

    // Set up Timer A and Enable Interrupts
    TACCTLO = CCIE;               // TACCR0 interrupt enabled
    TACCRO = 65535;               // maximum time between interrupts
    TACTL = TASSEL_2 + MC_2;      // Select MCLK, contmode
    _EINT();                       // Enable Global Interrupts

    while(1)
    {
        LPM0;
        BSP_TOGGLE_LED1();        // Toggle red LED
        BSP_TOGGLE_LED2();        // Toggle green LED
        // BSP_TURN_OFF_LED2();
        // BSP_TURN_OFF_LED2();
        LPM0;
    }
}

//*****
// Timer A interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A(void)
{
    TACCRO += 65535;              // Add Offset to TACCRO
    LPM0_EXIT;                    // Exit LPM0 on RETI
}
//*****
```

Want to find out more about programming MSP430 timers? Search the TI website for the **MSP430x2xx User's Guide**.

## Re-enable the LEDs

The code won't be very interesting if we can't see it work. Comment/un-comment the LED control statements like below:

```

BSP_Init();
BSP_TURN_ON_LED2();
// BSP_TURN_OFF_LED2();

// Set up Timer A and Enable Interrupts
TACCTL0 = CCIE;
TACCR0 = 65535;
TACTL = TASSEL_2 + MC_2;
_EINT();

while(1)
{
    LPM0;
    BSP_TOGGLE_LED1();
    BSP_TOGGLE_LED2();
//    BSP_TURN_OFF_LED2();
//    BSP_TURN_OFF_LED2();
    LPM0;
}

```

## 6. Build/Load/Run

Take a **target** board and carefully **insert** it into the emulator, then **build** the project. Make sure that it builds without error, and then click the **Go** button. The LEDs should be blinking quite a bit faster than before, but it will do for the purposes of this exercise.

### Warning: Code Explanation Ahead

The code that we added controls Timer\_A and its interrupt response. In the code right above `while()`:

<code>TACCTL0 0 = CCIE;</code>	enables Timer_A to generate an interrupt
<code>TACCR0 = 65535;</code>	loads the maximum possible value into the 16-bit count register
<code>TACTL = TASSEL_2 +MC_2;</code>	selects MCLK as the timer clock and continuous as the count mode
<code>_EINT();</code>	enables global interrupts

The code we added at the bottom of `main.c` is the Timer\_A interrupt service routine.

<code>#pragma vector=TIMERA0_VECTOR</code>	places the ISR correctly in memory
<code>__interrupt void Timer_A(void)</code>	tells the compiler this is an ISR
<code>TACCR0 += 65535;</code>	adds offset to count register
<code>LPM0_EXIT;</code>	returns to low power mode

By the way, `LPM0;` drops the MSP430 into this operating mode with the DCO **on**.

## 7. Turn off the LEDs

Click the **Stop Debugging** button and **undo** the LED code changes we just made:

```
BSP_Init();
// BSP_TURN_ON_LED2();
BSP_TURN_OFF_LED2();

// Set up Timer A and Enable Interrupts
TACTL0 = CCIE;
TACCR0 = 65535;
TACTL = TASSEL_2 + MC_2;
_EINT();

while(1)
{
    LPM0;
//    BSP_TOGGLE_LED1();
//    BSP_TOGGLE_LED2();
    BSP_TURN_OFF_LED2();
    BSP_TURN_OFF_LED2();
    LPM0;
```

**Build** and **load** the project to the target, then click the **Stop Debugging** button.

## 8. Measure Current

**Connect** the target board to the battery module and **measure** the current. I got **2.1mA – 1.5mA = 0.6mA** or **1.7mW**. Write your values into the table.

## 9. LPM3

In the existing code, we're running the MSP430 on the DCO at 8MHz. For this code, that's overkill. We can easily run it on the VLO (Very Low frequency oscillator) that is internal to the MSP430. But if we simply change out LPM0; statements to LPM3; the code simply won't work; the VLO needs to be setup and the timer needs to be changed to clock on the VLO.

Start out by changing both **LPM0;** statements to **LPM3;** .

## 10. Clock setup Code

**Delete** the **four** lines of code just above the **while()** statement and replace them with the **six** lines of code at the bottom of **Lab3 Code.txt** .

**Add #include "VLO\_Library.h"** as the third include file at the top of **main.c**.

Add **VLO\_Library.s43** from **C:\Texas Instruments\SimpliciTI-1.0.6\Projects\Examples\Peer applications\Z430RF\Lab3** to the **Source group** in the **Workspace** window.

Delete **TACCR0 += 65535;** from the ISR at the **bottom** of **main.c**.

Finally, add **unsigned int dco\_delta;** right below the line defining **i**.

The screenshot shows the IAR Embedded Workbench interface. On the left, the 'Workspace' window displays a project tree for 'Lab3 - Debug'. The 'Source' folder contains several files, including 'main.c'. On the right, the 'main.c' file is open, showing the following code:

```

volatile unsigned int i;           // volatile to prevent optimization
unsigned int dco_delta;

void main(void)
{
    BSP_Init();                    // Always initialize BSP first
    BSP_TURN_ON_LED2();           // Turn on green LED
    // BSP_TURN_OFF_LED2();

    // Set up Timer A to VLO and enable interrupts
    dco_delta = TI_measureVLO();   // # of 8MHz cycles in 1 ACLK cycle

    TACCTL0 = CCIE;                // TACCR0 interrupt enabled
    TACCR0 = (80000 / dco_delta);  // Set the wakeup period to 1 sec
    TACTL = (TASSEL_1 | MC_1);     // Select ACLK source, count UP mode

    BCSCCTL3 |= LFX1S_2;          // Set ACLK = VLO
    _EINT();                       // Enable Global Interupts

    while(1)
    {
        LPM3;
        BSP_TOGGLE_LED1();        // Toggle red LED
        BSP_TOGGLE_LED2();        // Toggle green LED
        // BSP_TURN_OFF_LED2();
        // BSP_TURN_OFF_LED2();
        LPM3;
    }

    //*****
    // Timer A interrupt service routine
    #pragma vector=TIMERAO_VECTOR
    __interrupt void Timer_A(void)
    {
        LPM3_EXIT;                // Exit LPM3 on RETI
    }
}
    
```

### 11. Build/Load/Run

Before testing the code, **swap** the LED comments back so you can see the code operating. **Build/load/run**. The numbers I selected for the timer should look about like what we had before. Click the **Stop Debugging** button, **swap** the LED comments back so the LEDs stay off and **Build/Load** the project. Click the **Stop Debugging** button.

### 12. Measure

**Connect** this target board to the battery module and **measure** the current. I got **1.7mA – 1.5mA = 0.2mA** or **0.6mW**. Write your values into the table.

### 13. Shut Down

Shut the **VOM off**, wrap the test leads around it and give it to your instructor; we won't need it again in this workshop. Shut down **IAR Embedded Workbench**, **disconnect** the eZ430-RF2500 hardware and extender cable and put them aside. **Make sure to get that jumper and place it on one of the battery pins for safekeeping.**



You're done