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

MSP430 Programming

Module Topics

MSP430 Products and Architecture

Power Efficient Applications and Power Modes

Clock Management

Hints and Tips

Timers

Summary

Lab3 – MSP430 Programming

Description: 

Hardware list:

Software list:

Procedure:
*** I’ve seen an MSP430 run on a couple of grapes ***
MSP430 Products and Architecture

MSP430 Products

1xx-Catalog
- 8 MIPS
- 1-60KB

2xx-Catalog
- 16 MIPS
- 1-120KB
- 500nA Stand By

5xx-Next Gen
- 25 MIPS
- 32-256 KB
- USB-RF

Ultra-Low-Power Architecture

Multiple operating modes
- 0.1uA power down
- 0.7uA standby mode
- 250uA / 1MIPS

Zero-power brown-out/reset circuit (BOR)
- 50nA pin leakage

Modern CPU
Minimum cycles per task

Instant-on stable high-speed clock

Intelligent peripherals

Applications ...
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

- LPM0
  - CPU Off
  - DCO on
  - ACLK on
  - 45-65uA

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

- Active
  - DCO on
  - ACLK on
  - 300-500uA

- Stand-by
  - DCO off
  - ACLK on
  - 1.0uA

- Off
  - All
  - Clocks Off
  - 0.1uA

Specific values vary by device

Clock Management ...
Clock Management

Clock Management: 1xx, 2xx, 4xx

Instant-on clock ...

Instant-On Clock

Near instant response to events, even from sleep modes
As a result, MSP430 can spend more time asleep

Operating modes ...
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 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)
Hints and Tips

**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

**Power Manage External Devices**

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

**TLV2760**  
0.01uA = Shutdown  
20uA = Active  
0.06uA = Average

**MSP430**  
“1uA OPA”  
1uA = Quiescent  
1uA = Active  
1uA = Average

Move functions to peripherals …
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

- 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;
}

// UART Receive Interrupt
#pragma vector=UART_VECTOR
__interrupt void rx (void) {

  TXBUF0 = RXBUF0;
}

100% CPU Load

9600 baud

0.1% CPU Load
Timers

Timer_A

- Asynchronous 16-Bit timer/counter
- Continuous, up-down, up count modes
- Multiple capture/compare registers
- PWM outputs
- Interrupt vector register for fast decoding
- Can trigger DMA transfer
- On all MSP430s

Timer_A Counting Modes

Stop/Halt
Timer is halted

Continuous
Timer continuously counts up

Up
Timer counts between 0 and CCR0

Up/Down
Timer counts between 0 and CCR0 and 0

CCR – Count Compare Register

Interruption...
Timers

**Timer_A Interrupts**

The Timer_A Capture/Comparison Register 0 Interrupt Flag (TACCR0) generates a single interrupt vector:

```
TACCR0 CCIFG  TIMERA0_VECTOR
```

No handler required

TACCR1, 2 and TA interrupt flags are prioritized and combined using the Timer_A Interrupt Vector Register (TAIV) into another interrupt vector

```
TACCR1 CCIFG
TACCR2 CCIFG   TAIV  TIMERA1_VECTOR
TAIFG
```

Your code must contain a handler to determine which Timer_A1 interrupt triggered

```
#pragma vector = TIMERA1_VECTOR
__interrupt void TIMERA1_ISR(void)
{
    switch(__even_in_range(TAIV,10))
    {
        case 2 :      // TACCR1 CCIFG
            P1OUT ^= 0x04; break;
        case 4 :      // TACCR2 CCIFG
            P1OUT ^= 0x02; break;
        case 10 :     // TAIFG
            P1OUT ^= 0x01; break;
    }
}
```

**TAIV Handler Example**

```

Source TAIV Contents
No interrupt pending 0
TACCR1 CCIFG 02h
TACCR2 CCIFG 04h
Reserved 06h
Reserved 08h
TAIFG 0Ah
Reserved 0Ch
Reserved 0 Eh
```

C Coding Tips …
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

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
*** 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.

- Measure current draw of application
- Apply low power design techniques to reduce current
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-Milliammeter (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 9th 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.

<table>
<thead>
<tr>
<th></th>
<th>Measured mA – 1.5mA</th>
<th>Calculated Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S/W loop with LEDs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S/W loop w/o LEDs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LPM0 w/o LEDs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LPM3 w/o LEDs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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\SimpliciTI-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.

```c
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\SimpliciTI-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:

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

#include "bsp.h"

void main(void) {
  BSF_Init(); // Always initialize BSP first
  BSF_TURN_ON_LED2(); // Turn on green LED
  BSF_TURN_OFF_LED2();

  // Set up Timer A and Enable Interrupts
  IACCL0 = CCIE;  // IACCR0 interrupt enabled
  TACR0 = 65535; // maximum time between interrupts
  TACL = TASSEL_2 + MC_2; // Select NCLK, contmode
  _EINT(); // Enable Global Interrupts

  while(1) {
    LPM0;
    BSF_TOGGLE_LED1(); // Toggle red LED
    BSF_TOGGLE_LED2(); // Toggle green LED
    BSF_TURN_OFF_LED2();
    BSF_TURN_OFF_LED2();
    LPM0;
  }
}

/************************************************************
// Timer A interrupt service routine
#pragma vector-TIMER0_VECTOR
__interrupt void Timer_A(void) {
  IACCR0 = 65535; // Add Offset to IACCR0
  LPM0_EXIT; // Exit LPM0 on RSTI
}
/************************************************************
```

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:

```c
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(1)`:

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

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

```c
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A(void)
TACCR0 += 65535;
LPM0_EXIT;
```

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 **un-do** the LED code changes we just made:

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

// Set up Timer A and Enable Interrupts
TACCTL0 = CCIE;
TACCR0 = 65535;
TACTL = TACSEL_2 + MC_2;
_PINT();

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 set up 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\SimpliciiTI-1.0.6\Projects\Examples\Peer applications\eZ430RF\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`.
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.