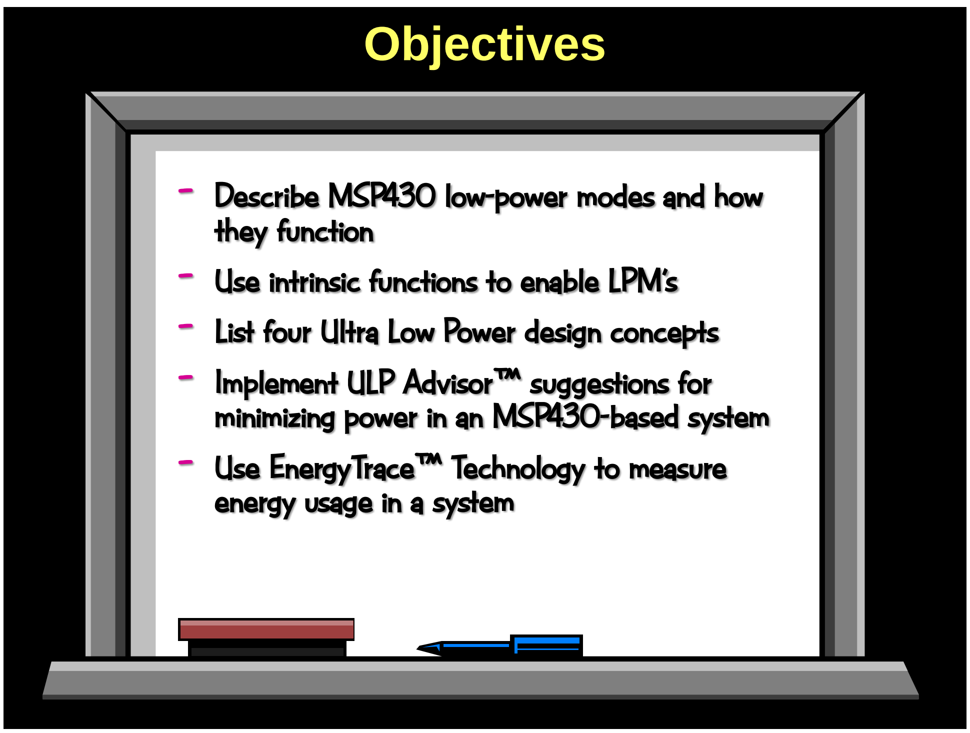
Introduction

Ultra-low power is in our DNA.

The MSP430 is inherently low-power by design. But there's more to it than that. As a system designer and programmer, you need to utilize the low-power modes and features to extract the most from the least. This chapter introduces us to a number of these ultra-low power (ULP) capabilities; including the many tools TI provides to help you achieve your ULP target.

Learning Objectives



Chapter Topics

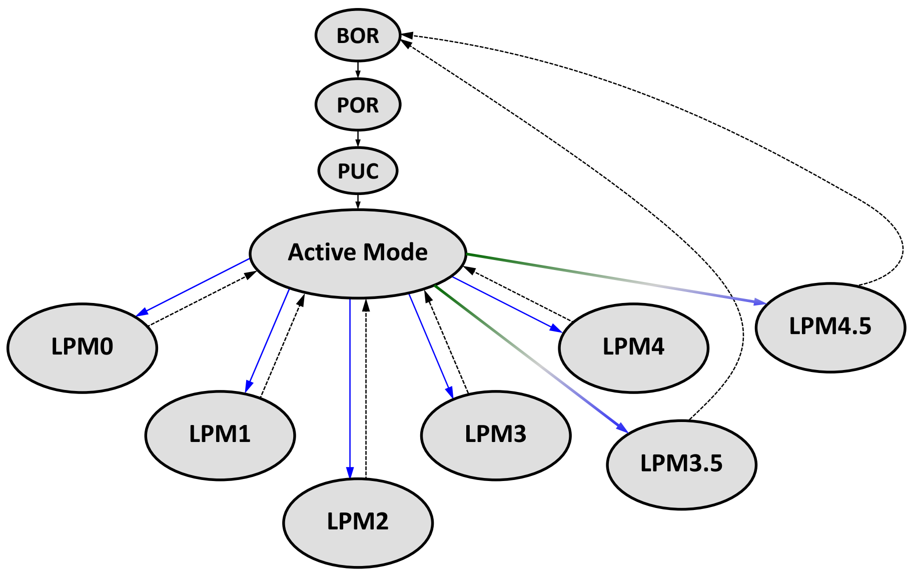
Prerequisites and Tools

Prerequisites & Tools

- **Skills** — **Chapter**
  - Creating a CCS Project for MSP430 Launchpad(s) (Ch 2 & 3)
  - Basic knowledge of:
    - C language
    - Setting up MSP430 clocks (Ch 4)
    - Using interrupts (setup and ISR's) (Ch 5)
    - Timer usage and configuration (Ch 6)
- **Hardware**
  - EnergyTrace™ capable hardware (one of the following)
    - MSP-EXP430FR5969 Launchpad
    - MSP-FET emulation tool (plus 4 jumper wires)
  - Windows 7 (and 8) PC with available USB port
  - MSP430F5529 Launchpad or MSP430FR5969 Launchpad (with included USB micro cable)
  - One jumper wire (female to female)
- **Software**
  - CCSv6
  - MSP430ware_1_90_xx_xx

Low Power Modes (LPM)

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/0, 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 (Bit Settings)

Operating Mode	CPU (MCLK)	SMCLK	ACLK	Vcore	RAM Retention	FRAM Retention	Status Register (SR): CPUOFF	Status Register (SR): OSCOFF	Status Register (SR): SCG0	Status Register (SR): SCG1	PMMCTL0. PMMREGOFF
Active	☒	☒	☒	☒	☒	☒	0	0	0	0	0
LPM0		☒	☒	☒	☒	☒	1	0	0	0	0
LPM1		☒	☒	☒	☒	☒	1	0	1	0	0
LPM2			☒	☒	☒	☒	1	0	0	1	0
LPM3			☒	☒	☒	☒	1	0	1	1	0
LPM3.5						☒	1	1	1	1	1
LPM4				☒	☒	☒	1	1	1	1	0
LPM4.5						☒	1	1	1	1	1

* SCG = System Clock Generator

MSP430™ Series Comparison

Mode		G2xx	F5xx	FR57xx	FR58xx FR59xx
Performance (max)		16 MHz	25 MHz	24 MHz (FRAM at 8MHz)	16 MHz (FRAM at 8MHz)
Flex Unified Memory		No	No	FRAM (16K)	FRAM (64K)
Active	AM	230 µA (1MHz)	180 µA/MHz	100 µA/MHz	<100 µA/MHz
Standby RTC	LPM3 LPM3.5	0.7 µA	1.9 µA 2.1 µA	6.3 µA 1.5 µA	0.7 µA 0.4 µA
Off	LPM4 LPM4.5	0.1 µA	1.1 µA 0.2 µA	5.9 µA 0.3 µA	0.6 µA 0.1 µA
Wake-up from	Standby	1.5 µs	3.5 µs or 150 µs	78 µs	<10 µs
	Off	-	2000 µs	310 µs	150 µs

Using Low Power Modes

Entering Low Power Modes

Enter LPMx	C Compiler Intrinsic	Writing to SR with Intrinsic
LPM0	_low_power_mode_0();	_bis_SR_register(GIE + LPM0_bits);
LPM1	_low_power_mode_1();	_bis_SR_register(GIE + LPM1_bits);
LPM2	_low_power_mode_2();	_bis_SR_register(GIE + LPM2_bits);
LPM3	_low_power_mode_3();	_bis_SR_register(GIE + LPM3_bits);
LPM4	_low_power_mode_4();	_bis_SR_register(GIE + LPM4_bits);

- As written, both intrinsic functions *enable interrupts* and associated *low-power mode*
- bis (and bic) instructions mimic assembly language:
  - bis = bit set
  - bic = bit clear
- bis/bic intrisics allows greater flexibility in selecting bits to set/clear

Automatically Re-entering LPM (after ISR)

```
main()
{
  initGpio();
  initClocks();
  initTimers();

  _low_power_mode_3();

  //while(1);
}
```

LPM3

- Executing LPM3 function puts the processor standby
- Unless an interrupt occurs, CPU will stay asleep
- No while{} loop is needed

```
#pragma vector = TIMER1_A0
__interrupt ISR()
{
  GPIO_toggleOutputOnPin()

} // Return from interrupt (RETI)
```

- An interrupt wakes the CPU
- Status Register (SR) is saved to stack (including the LPM setting)
- Exiting ISR routine:
  - Compiler uses RETI instruction which restores SR from stack
  - Restoring SR places CPU back into low-power mode

Leaving LPM (after ISR)

```
main()
{
  initGpio();
  initClocks();
  initTimers();
  while(1){
     _low_power_mode_3();
     filter();
  }
}

#pragma vector = TIMER1_A0
__interrupt ISR()
{
  getSample();
  _low_power_mode_off_on_exit();

}  // Return from interrupt (RETI)
```

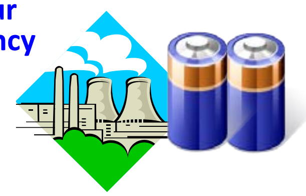
LPM3

- Executing LPM3 function puts the processor standby
- Unless an interrupt occurs, CPU will stay asleep
- Since ISR exits from LPM, we need additional code (such as a while{} loop)
- An interrupt wakes the CPU
- Status Register (SR) is saved to stack (including LPM bits)
- Exiting ISR routine:
  - 'exit' fcn modifies saved SR (clearing LPM) before restore
  - RETI instruction restores SR from stack
  - With LPM "off", CPU returns to instruction after LPM intrinsic; e.g. filter()

Low Power Concepts

Principles For ULP Applications

- MSP430 is inherently low-power, but your design has a big impact on power efficiency
- Even wall powered devices can become "greener"

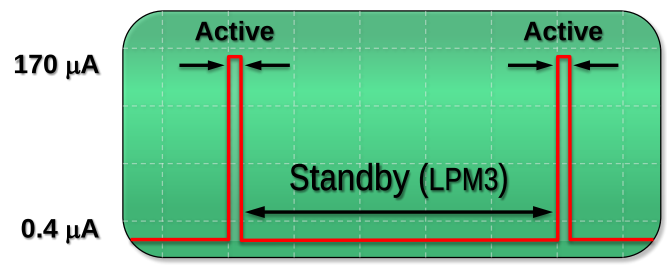


- Use interrupts to control program flow
- Maximize the time in LPM3
- Replace software with peripherals
- Configure unused pins properly
- Power manage external devices
- Efficient code makes a difference

  Every unnecessary instruction executed is a portion of the battery that's wasted and gone forever

Use Interrupts and Low-Power Modes

Use Interrupts & Maximize LPM3



Leave On the Slow Clock

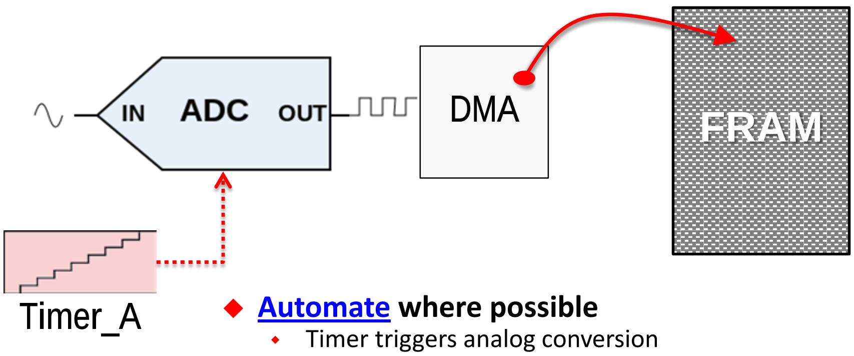
- Low power clock and peripherals interrupt CPU only for processing

On-Demand CPU Clock

- DCO starts immediately
- CPU processes data and quickly returns to Low Power Mode

Replace Software with Peripherals

Replace Software With Peripherals

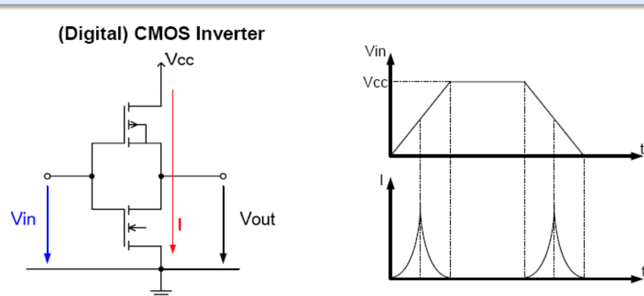


- **Automate where possible**
  - Timer triggers analog conversion
  - ADC triggers DMA to move result to memory
- **Saves power since CPU and high-speed clock can be turned off**
- **Higher precision and less latency for analog sampling since timer directly triggers conversion**
- **Faster results since peripherals are optimized to perform operations more quickly than the CPU**

Configure Unused Pins

Configure Unused Pins

- **Digital input pins subject to shoot-through current**
  - Input voltages between V_{IL} and V_{IH} cause shoot-through if input is allowed to "float" (left disconnected)
- **Port I/O's should either:**
  1. Be driven to V_{cc} or ground by an external device
  2. Set as an input using the pull-up/down resistor
  3. Driven as an output



Efficient Code Makes a Difference

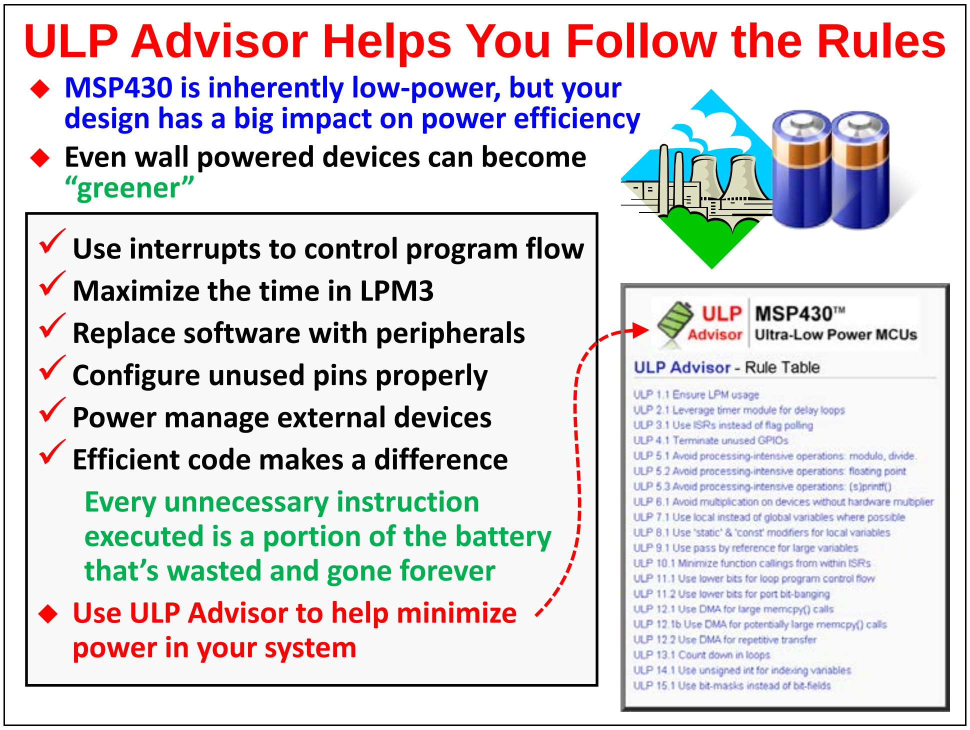
ULP "Sweet Spot"

- **Power dissipation increases with…**
  - CPU clock speed (MCLK)
  - Input voltage (V_{CC})
  - Temperature
- **Slowing MCLK reduces instantaneous power, but often increases active duty-cycle (how long the CPU stays on)**
  - Look for ULP 'sweet spot' to maximize performance with minimum current consumption per MIPS
    - ☞ Usually 8 MHz MCLK is the best tradeoff of power/performance
- **Use lowest input voltage possible**
  - 'F5529 lets you lower core voltage if full-speed operation is not required
  - 'FR5969 operates at full speed down to 1.8V
  - On some MSP430 devices, you need to take into consideration minimum Vcc for flash programming, etc.

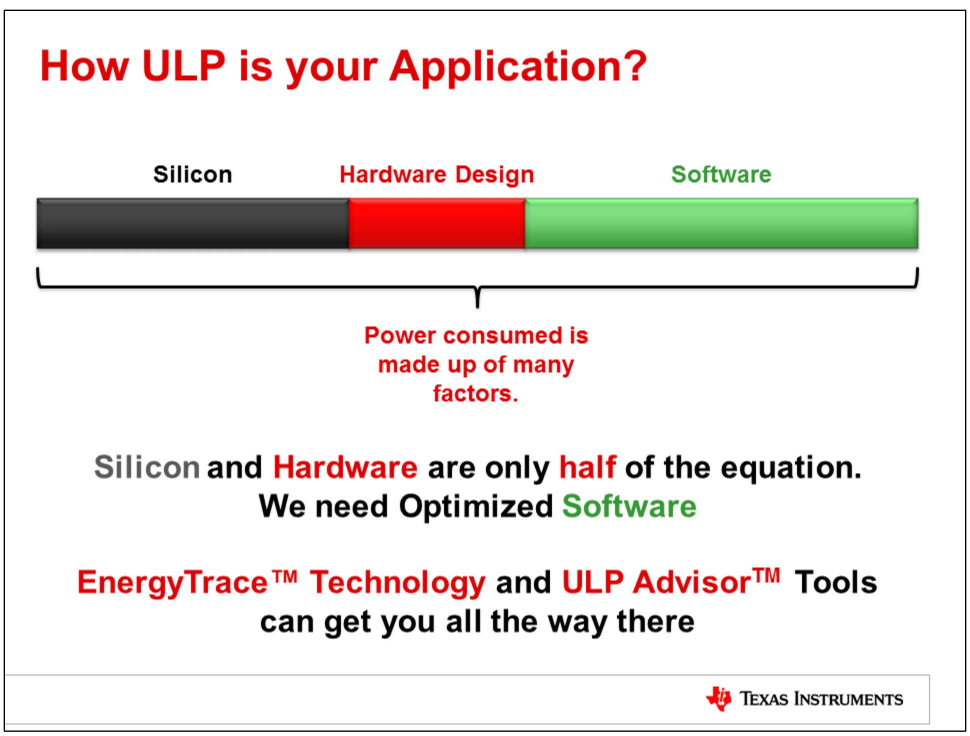
Optimize Performance

- **Use Hardwired Accelerators, where available**
  - MPY32
  - AES256
  - CRC16
  - DMA
- **Optimize Code (saves code size and wasted cycles)**
  - CCS "Release" configuration with -O, -O3, or -O4
  - Use –mf option to set tradeoff between code size/speed
  - Optimization Advisor
- **Optimized Libraries (faster and easier)**
  - MSPMathLib (floating-point math)
  - IQmath and Qmath (fixed-point math)
  - Energy calculations
  - Capacitive Touch

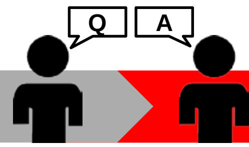
Follow the Rules (ULP Advisor™)



About ULP Advisor™



ULP Advisor™ benefits all experience levels



Beginning ULP developers

- Teaching tool for new MSP430 users
- Practical introduction to ULP techniques
- Immediate coding feedback
- Wiki provides quick solution and detailed background info
- Learn more from the community & E2E

Experienced ULP developers

- Not everybody remembers all the rules all the time
- New rules might come in
- Saves time vs. manually going through a large project or library to check for ULP
- Helpful when developers inherit code from other sources
- ULP Advisor should always be used regardless of the application or target device.
- Contribute to wiki & E2E

Texas Instruments

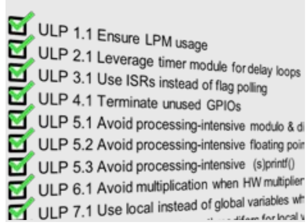
ULP Advisor™ Software: Turning MCU developers into Ultra-Low-Power experts

ULP Advisor analyzes all MSP430 C code line-by-line.

- Supports all MSP430 devices and can benefit any application
- Checks all code within a project at build time
- Enabled by default
- Parses code line-by-line

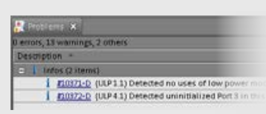
Checks against a thorough Ultra-Low-Power checklist.

- List of 15 Ultra-Low-Power best practices
- Compilation of ULP tips & tricks from the well-known to the more obscure
- Combines decades of MSP430 and ULP development experience



Highlights areas of improvement within code.

- Identify key areas of improvement
- Presented as a “remark” within “Problems” window
- Includes a link to more information



Texas Instruments

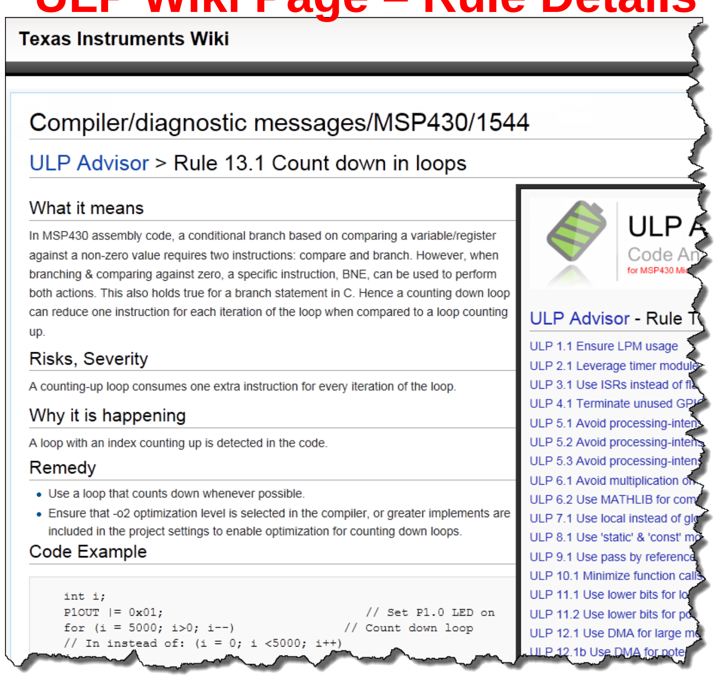
The List … of ULP Rules

ULP Advisor™
Code Analysis Tool
for MSP430 Microcontrollers

ULP Advisor Rules

Basic	ULP 1.1	Ensure LPM usage
	ULP 2.1	Leverage timer module for delay loops
	ULP 3.1	Use ISRs instead of flag polling
	ULP 4.1	Terminate unused GPIOs
Math	ULP 5.1	Avoid processing-intensive operations: modulo, divide
	ULP 5.2	Avoid processing-intensive operations: floating point
	ULP 5.3	Avoid processing-intensive operations: (s)printf()
	ULP 6.1	Avoid multiplication on devices without hardware multiplier
	ULP 6.2	Use MATHLIB for complex math operations
Coding Details	ULP 7.1	Use local instead of global variables where possible
	ULP 8.1	Use 'static' & 'const' modifiers for local variables
	ULP 9.1	Use pass by reference for large variables
	ULP 10.1	Minimize function calls from within ISRs
	ULP 11.1	Use lower bits for loop program control flow
	ULP 11.2	Use lower bits for port bit-banging
DMA	ULP 12.1	Use DMA for large memcpy() calls
	ULP 12.1b	Use DMA for potentially large memcpy() calls
	ULP 12.2	Use DMA for repetitive transfer
Counts, Indexes, Masks	ULP 13.1	Count down in loops
	ULP 14.1	Use unsigned variables for indexing
	ULP 15.1	Use bit-masks instead of bit-fields

ULP Wiki Page – Rule Details



How Do You Enable ULP Advisor™?

Easily access ULP Advisor™ software, supporting all MSP430 development environments

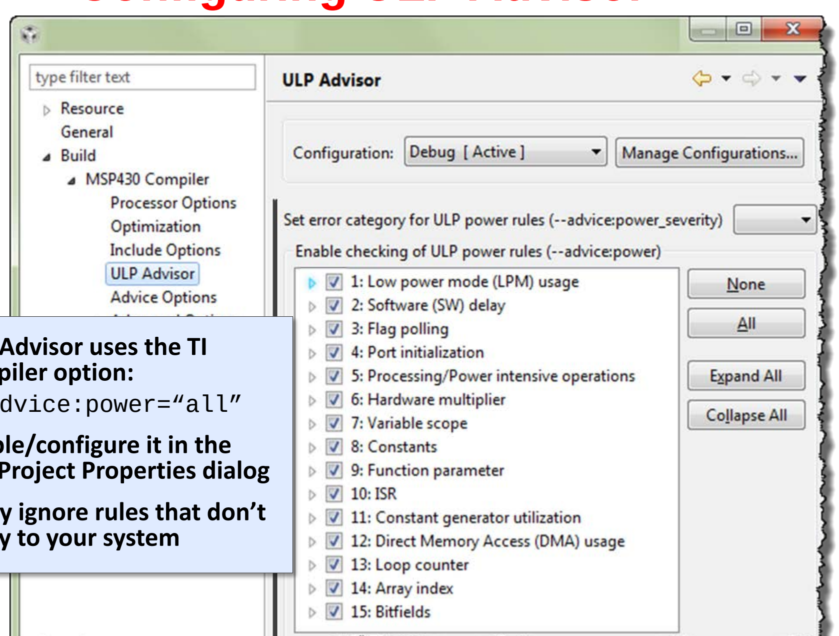
- Integrated into popular MSP430 IDEs for seamless operation
- ULP Advisor is automatically enabled & checks all code at build time
- Joins differentiated MSP430 software tools integrated into CCS, including MSP430Ware & Grace

Integrated into TI's Code Composer Studio

Integrated into IAR Embedded Workbench

Stand-alone version available for Open Source GCC & other compilers

Configuring ULP Advisor



- ULP Advisor uses the TI compiler option: `--advice:power="all"`
- Enable/configure it in the CCS Project Properties dialog
- Easily ignore rules that don't apply to your system

EnergyTrace™

Energy Aware Debugging

MSP-EXP430FR5969 Launchpad with on-board MSP-FET

MSP-FET
- Available: June 2014
- System power must come from FET

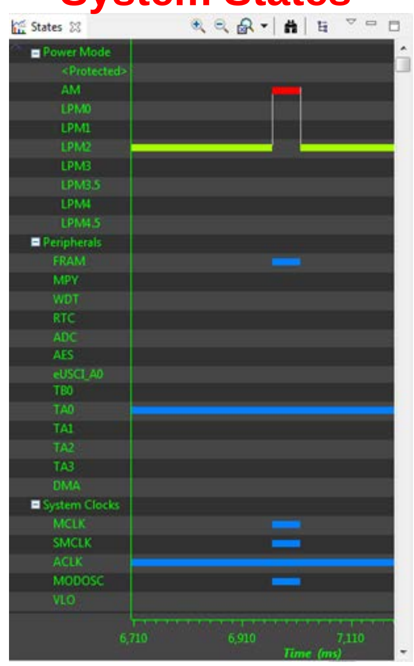
- **Two levels of EnergyTrace™**
  1. EnergyTrace: Measures energy usage in the system
  2. EnergyTrace++: Energy, Power Modes, Clocks and Peripherals
- **Devices supported by EnergyTrace (using MSP-FET):**
  - ‘FR59xx and ‘FR69xx devices support EnergyTrace++
  - All MSP430 devices support EnergyTrace

EnergyTrace Profile

EnergyTrace++™ Profile

Name	Runtime (%)	Energy (%)
System	100	83.5
CPU		
Low Power Mode	94.3	67.2
LPM2	94.3	67.2
LPM0	0.0	0.0
Active Mode	5.7	32.8
TIMER0_A0_ISR	5.5	32.6
main	0.2	0.2
Peripherals		
TA0	99.8	
FRAM	5.7	
TA2	0.0	
TA1	0.0	
MPY	0.0	
TA3	0.0	
eUSCI_A0	0.0	
TB0	0.0	
eUSCI_A1	0.0	
RTC	0.0	
eUSCI_B0	0.0	
AES	0.0	
DMA	0.0	
COMP	0.0	
WDT	0.0	
ADC	0.0	
REF	0.0	
System Clocks		
ACLK	100.0	
SMCLK	5.7	
MCLK	5.7	
MODOSC	5.7	
VLO	0.0	

System States



Power & Energy Graphs

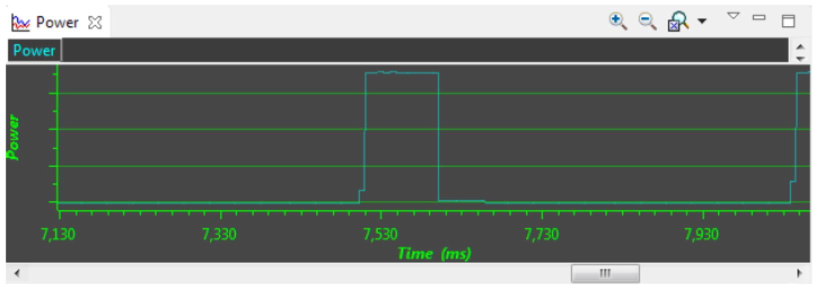


Figure 3-9. Power Window

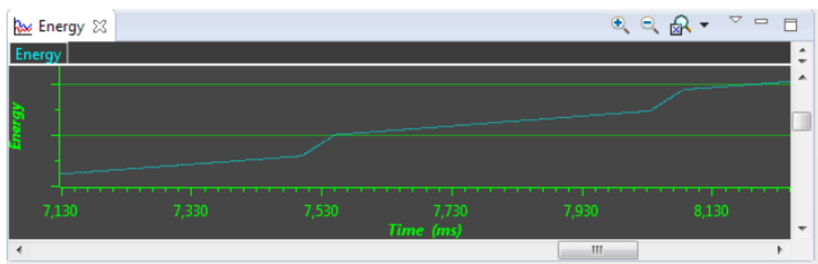
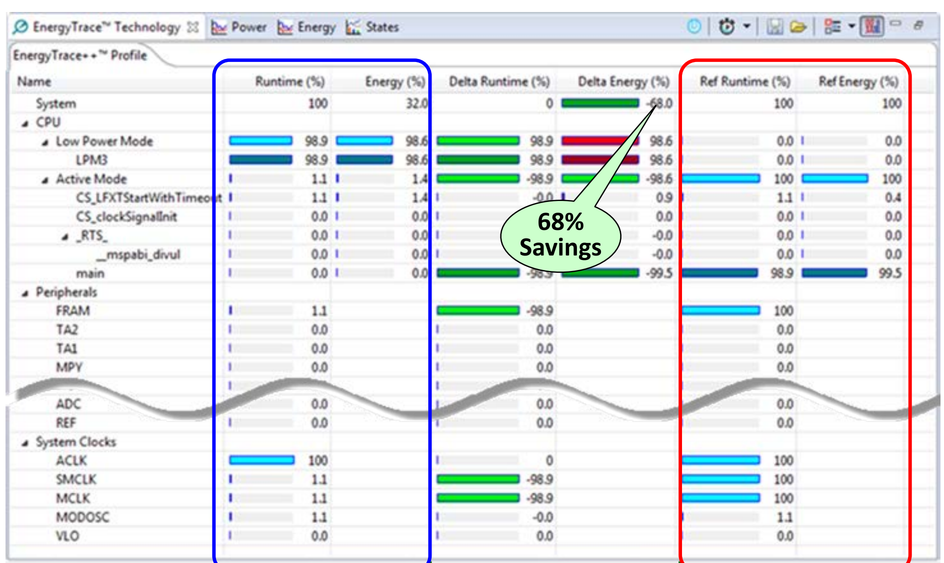


Figure 3-10. Energy Window

EnergyTrace Profile Comparison

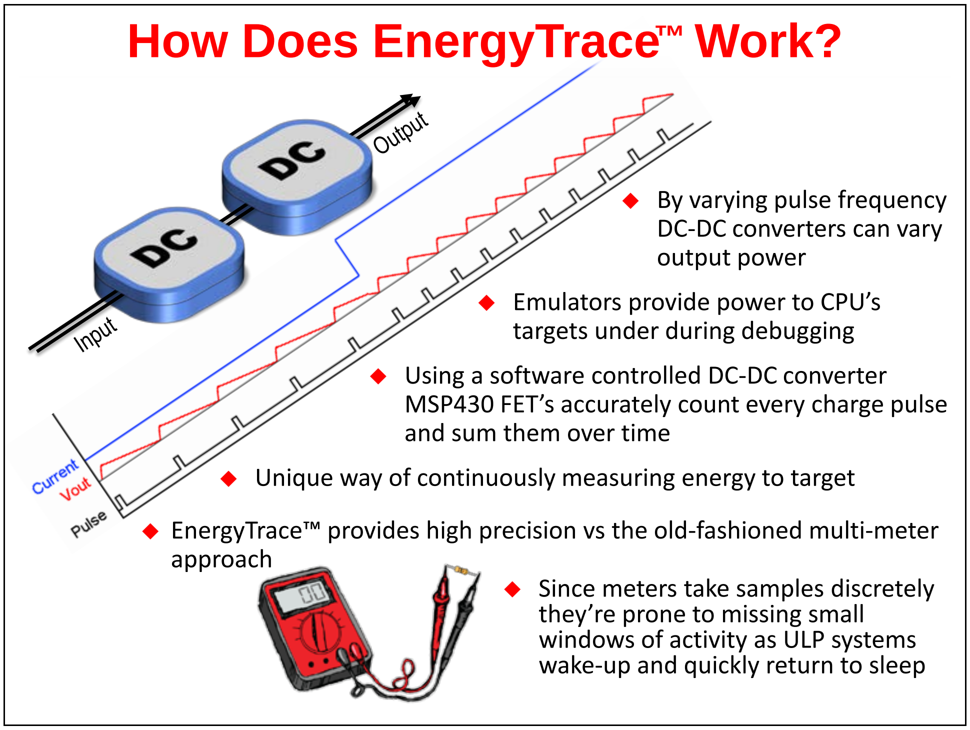


EnergyTrace™ Technology | Power | Energy | States

EnergyTrace++™ Profile

Name	Runtime (%)	Energy (%)	Delta Runtime (%)	Delta Energy (%)	Ref Runtime (%)	Ref Energy (%)
System	100	32.0	0	-68.0	100	100
CPU						
Low Power Mode	98.9	98.6	98.9	98.6	0.0	0.0
LPM3	98.9	98.6	98.9	98.6	0.0	0.0
Active Mode	1.1	1.4	-98.9	-98.6	100	100
CS_LFXTStartWithTimeout	1.1	1.4	-0.0	0.9	1.1	0.4
CS_clockSignalInit	0.0	0.0	[illegible]	0.0	0.0	0.0
RTS	0.0	0.0	[illegible]	-0.0	0.0	0.0
__mspabi_divul	0.0	0.0	[illegible]	-0.0	0.0	0.0
main	0.0	0.0	[illegible]	-99.5	98.9	99.5
Peripherals						
FRAM	1.1		-98.9		100	
TA2	0.0		0.0		0.0	
TA1	0.0		0.0		0.0	
MPY	0.0		0.0		0.0	
ADC	0.0		0.0		0.0	
REF	0.0		0.0		0.0	
System Clocks						
ACLK	100		0		100	
SMCLK	1.1		-98.9		100	
MCLK	1.1		-98.9		100	
MODOSC	1.1		-0.0		1.1	
VLO	0.0		0.0		0.0	

How does EnergyTrace Work?



Lab 7 – Low Power Optimization

Abstract

This lab exercise introduces us to many of the techniques used for measuring and reducing power dissipation in a MSP430 based design.

We begin by learning how to use EnergyTrace™ to measure energy consumption in our programs. Using this (or more crudely, using a multi-meter) we can now judge the affects our low-power optimizations have on our system.

Lab 7 – Optimizing for Low-Power

A. Getting Started with EnergyTrace™

Explore tools by comparing Lab4a & Lab4c

- **Enable EnergyTrace**
- **Capture EnergyTrace profile**
- **Compare EnergyTrace profiles**
- **‘FR5969 users can explore EnergyTrace++**

B. Using ULP Advisor, Interrupts and LPM3

Improve power using Lab4c & Lab6b

- **Enable ULP Advisor**
- **Replace delay() function with Timer**
- **Make use of Low Power Mode 3 (LPM3)**

C. Does Initializing GPIO Ports Make a Difference?

- **Taking Lab4c, replace LED toggle with LPM3**
- **Initialize ALL pins as Outputs after reset**
- **Then, check if setting pins as Inputs makes a difference to power optimization**

In part B of the lab, we use ULP Advisor to point out where our code might be improved, from a power perspective. In this part of the lab, we go on to replace __delay_cycles() with a timer; as well as implement low power mode 3 (LPM3).

Finally, in part C, we examine what – if any – affect uninitialized GPIO can have on an microcontroller design. The results may surprise you…

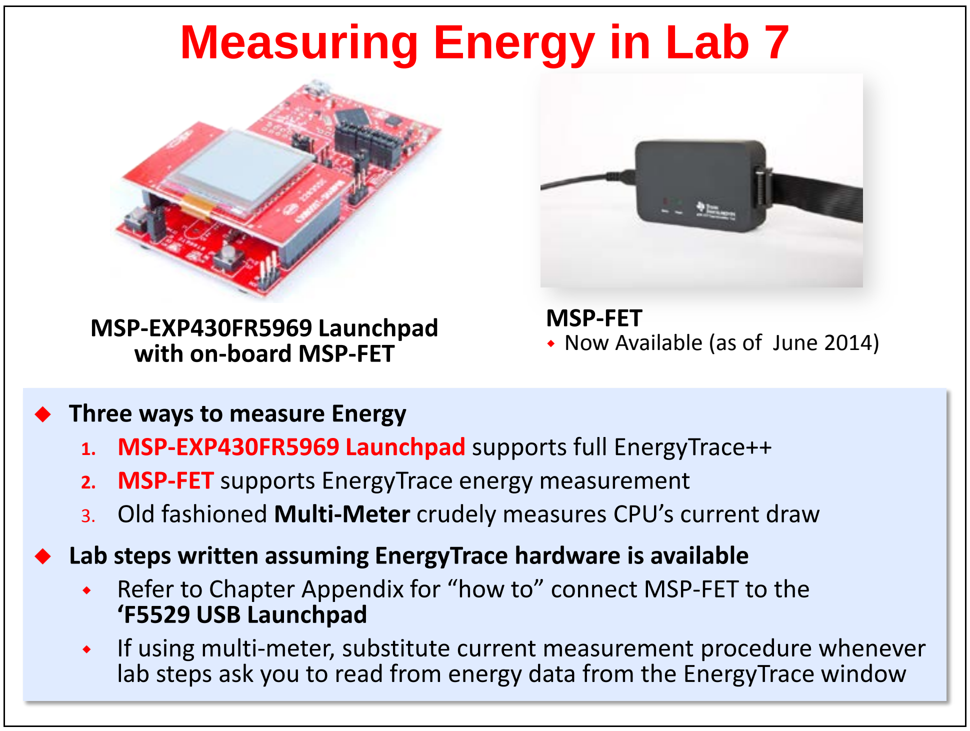
Chapter Topics

Notice - Measuring Energy in Lab 7

How to Measure Energy

There are three ways you can measure energy for the exercises found in this chapter:

1. The ‘FR5969 FRAM Launchpad supports the full EnergyTrace++ feature set – which includes energy measurement as well as tracing the CPU modes and peripheral states.
2. The new MSP-FET (Flash Emulation Tool) – supports measurement of energy with the EnergyTrace feature for all MSP430 devices.
3. If you do not have either tool which supports TI’s EnergyTrace, you will need to measure it the old fashioned way – using a multi-meter to determine the current being drawn by the MSP430 CPU. We refer you to Section 2.3 of the *MSP-EXP430F5529 Launchpad User’s Guide* (slau533b.pdf) for a detailed procedure on how this can be done.



Lab Exercise Energy Measurement Recommendations

As written, all Lab 7 exercises assume that you hardware (items #1 and #2 above) which implements EnergyTrace.

FR5969

‘FR5969 FRAM Launchpad

If you are using the ‘FR5969 FRAM Launchpad, no hardware configuration is required; the Launchpad (and ‘FR5969 silicon) has been designed to support these features.

F5529

‘F5529 USB Launchpad

If you are using the ‘F5529 USB Launchpad (or any other MSP430 board, for that matter), we suggest that you obtain the new MSP-FET tool. This will give you access to the new energy measurement feature. *(For live workshops held in North America, we provide MSP-FET tools that you may borrow to complete these lab exercises.)*

Normally, the MSP-FET connects to a target system via a 14-pin connector that follows TI’s emulation pinout standard. Since the ‘F5529 Launchpad does not ship with this connector populated on the Launchpad, you will need to use 4 jumper wires to connect the appropriate MSP-FET pins to the emulation-target isolation jumpers. Please see topic the topic *“Connecting MSP-FET to ‘F5529 USB Launchpad”* (page 7-46) for details on how to make these connections.

Bottom Line

To reiterate, these lab directions assume that you have hardware which supports EnergyTrace.

If you are using the ‘FR5969 Launchpad, you will have additional visibility into the CPU, but in either case, EnergyTrace provides highly accurate energy measurement.

Using a Multi-Meter

On the other hand, if you are using a multi-meter, you should substitute recording the current (μA/mA) for those lab steps where we direct users to view the EnergyTrace display. If you have any previous multi-meter experience, this shouldn’t be a difficult substitution to make. Comparing current values should be enough to evaluate ULP optimizations. Of course, you can always calculate the approximate energy values from the current and voltage (DVCC) values.

Note: Be warned… once you’ve used EnergyTrace, you’ll find it difficult going back to using a multi-meter; if not for the ease-of-use, for the increased measurement accuracy.

Lab 7a – Getting Started with Low-Power Optimization

This first lab exercise introduces us to measuring power – or energy – using EnergyTrace. *(If you don't have hardware that supports EnergyTrace, please refer to the note on the previous page.)*

We won't actually write much code in this exercise; rather, we will compare the solutions for a couple of our previous lab exercises – spending most of the time learning how to use the tools in the process.

Prelab Worksheet

1. What is the difference between EnergyTrace and EnergyTrace++?

   ____

   ____

   Which devices support EnergyTrace++? ________________________________

2. What hardware options are available that supports EnergyTrace? ______________

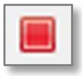
   ____

3. How can you calculate energy without EnergyTrace? ______________________

   ____

   What is the downside to this method? ________________________________

Configure CCS and Project for EnergyTrace

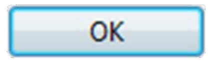


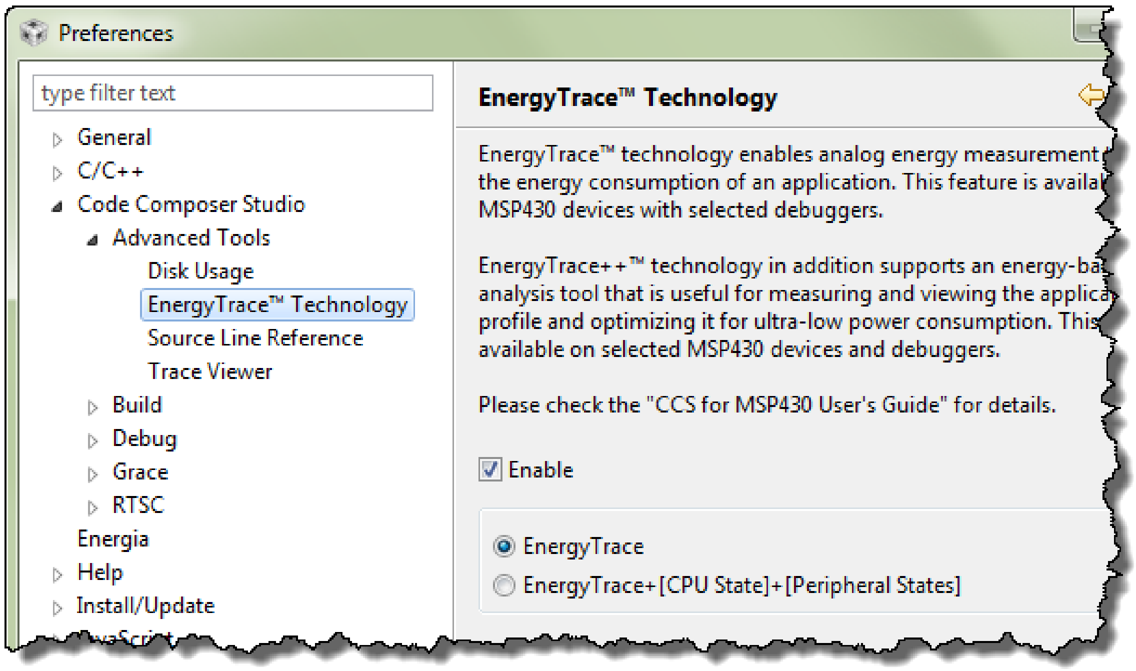
1. **Terminate the debugger if it's still open and close all projects and files that may be open in your CCS workspace.**

2. **Enable EnergyTrace profiling.**

   `Window → Preferences`

   `Code Composer Studio → Advanced Tools → EnergyTrace™ Technology`

   ☑ Enable   ⦿ EnergyTrace 



Note: 'FR5969 users, we'll look at the +States mode later on in the lab exercise.

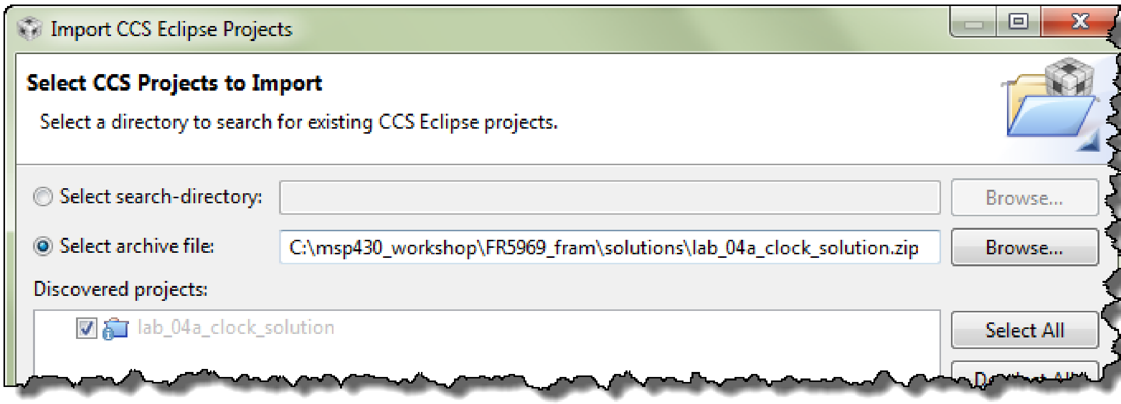
---

3. **Import the previous lab exercise: `lab_04a_clock_solution.zip`**

   `Project → Import CCS Projects`

   Then select either project (based upon the board you're using) and click OK.

   ```
   C:\msp430_workshop\F5529_usb\solutions\lab_04a_clock_solution.zip
   C:\msp430_workshop\FR5969_fram\solutions\lab_04a_clock_solution.zip
   ```

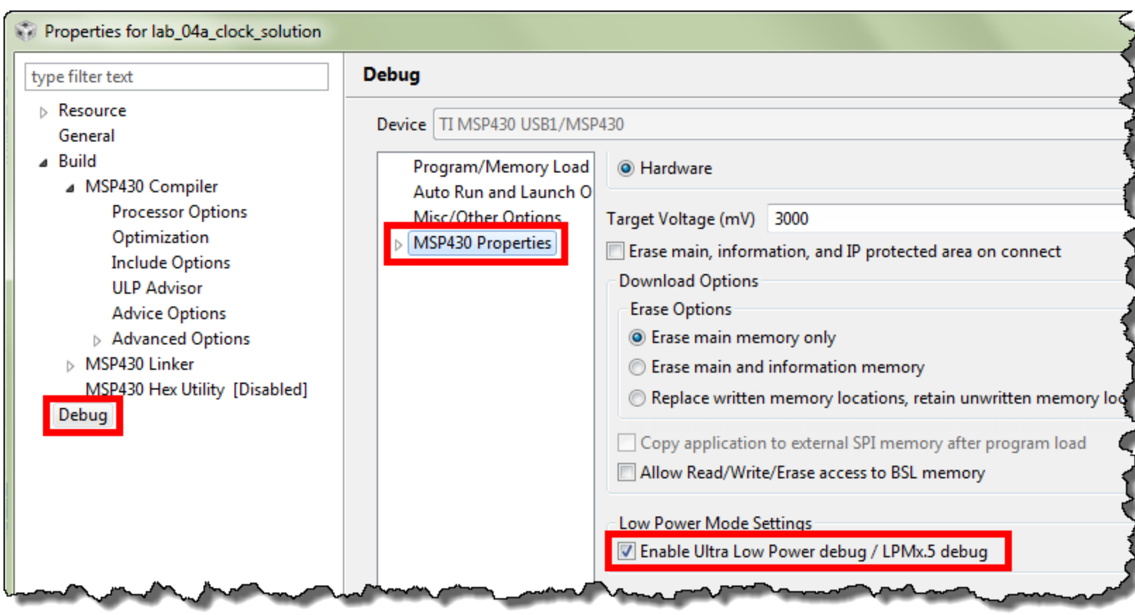


4. **('FR5969 only) Verify debugger is enabled for low-power (LPMx.5) modes.**

FR5969

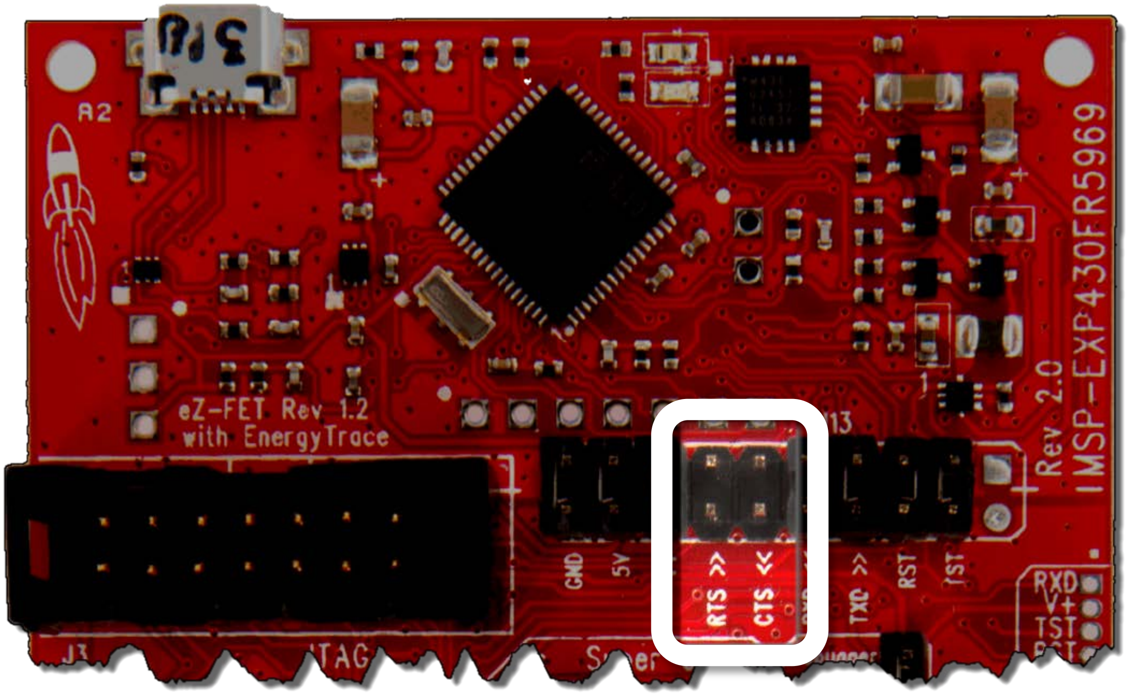
   `Right-Click on project → Properties → Debug → MSP430 Properties`

   Scroll-down and make sure the following is enabled, then click OK.



5. **If connected, remove the jumpers on the Launchpad for RTS and CTS in the emulator/target isolation connector.**

   This code does not use these UART signals, and keeping them connected draws slightly more power. (By default, these signals are usually disconnected.)



   Shown above is the 'FR5969 Launchpad, but you've find the same signals on the 'F5529 Launchpad connector.

Build Project and Run with EnergyTrace

6. **Build the project.**

   At this point, we shouldn't see any advice from ULP Advisor since we disabled this when building our previous lab projects. In a few minutes we'll turn this on and examine the results.

7. **Start the debugger.**

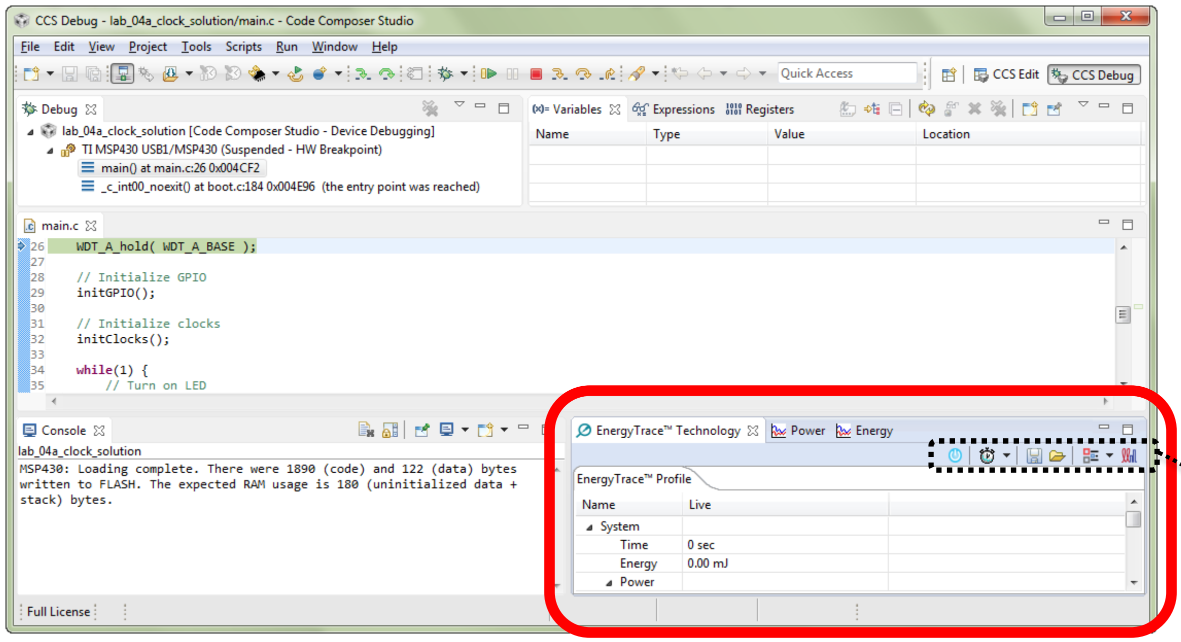
8. **Briefly examine the EnergyTrace window.**

   Notice that there's an extra window that opens in your debugger..

   If the EnergyTrace window did not open:

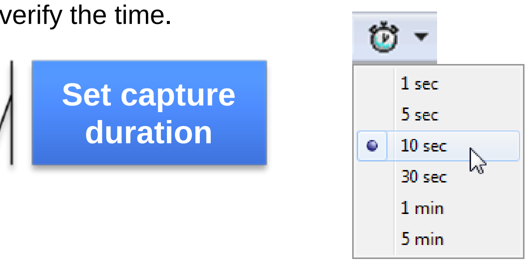
   – Double-check EnergyTrace is enabled.

   – `Window → Show View → Other… → MSP430-EnergyTrace`

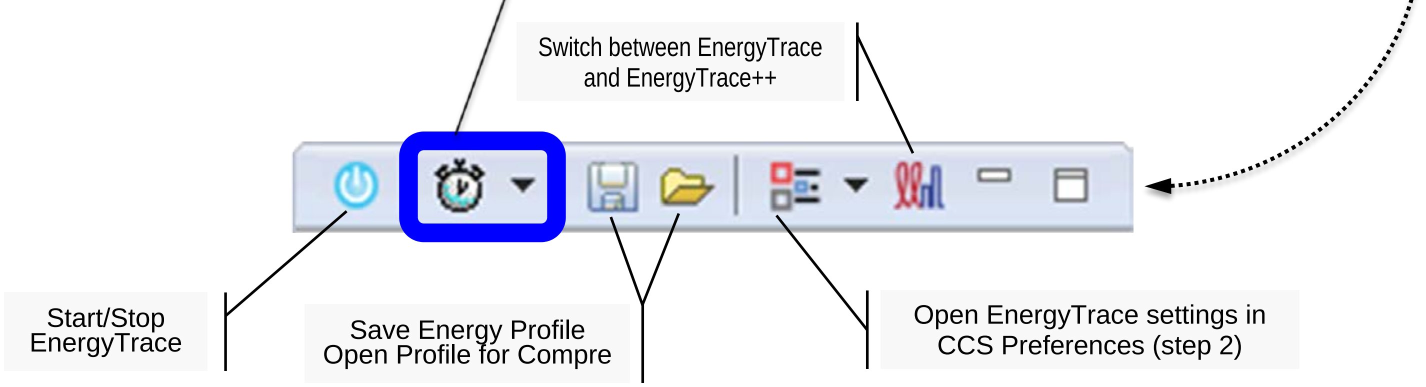


9. **Set the EnergyTrace capture duration to 10 seconds.**

   EnergyTrace captures data for a set period of time, and then displays those results. We can easily choose the capture period using the provided EnergyTrace toolbar button. It defaults to 10 seconds, but it doesn't hurt to verify the time.



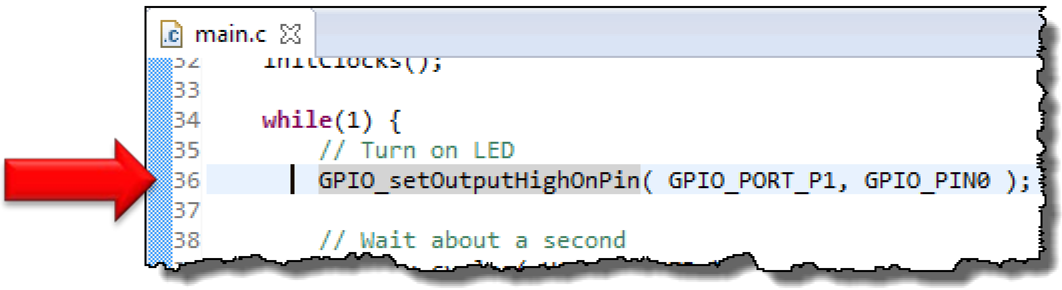
   While we're looking at the toolbar, please note some of its other buttons.



10. Set the cursor on the first line of code in the while loop.

In most systems, we care more about "continuous" power usage rather than "initialization" power usage. Because of this, we want to run past our initialization code before we start collecting energy data.

Instead of setting a breakpoint, it's often easier to place your cursor on the line you want to stop at, and then run to that cursor. Let's start the action by placing our cursor on the first line of the while loop.

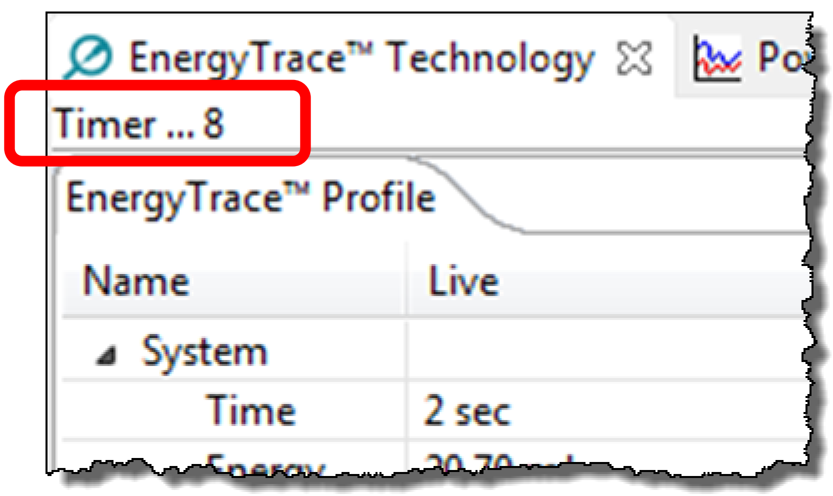


11. Run to the cursor

`Run → Run to Line` or better yet use: 

12. Click Resume and watch the duration count down.

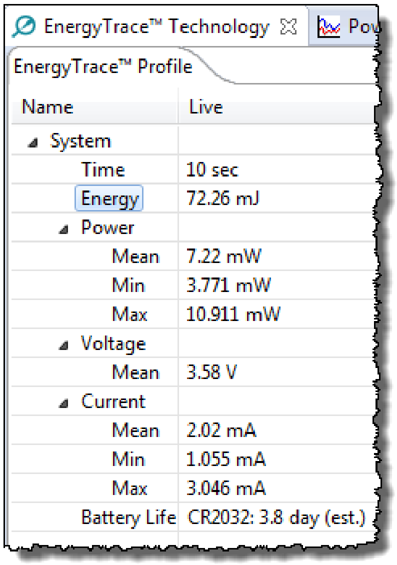
When we begin running the code it will execute the while{} loop and capture the energy data for 10 seconds.



13. Suspend your program after count reaches zero.

EnergyTrace doesn't require that we halt the program, but we don't need to keep it running either.

14. Expand EnergyTrace window to view the energy profile you just created.



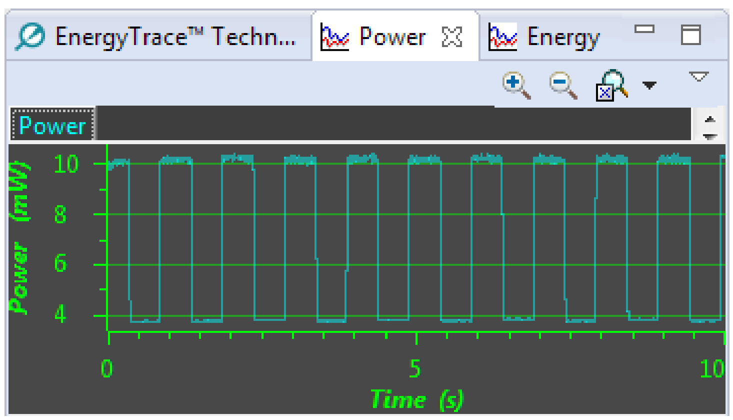
We see that our processor consumed 72.26mJ in the 10 second capture period.

For many reasons, ***your numbers may differ*** from that shown here:

- You may be using a different Launchpad.
- You start/end capture locations were different than ours
- Your compiler version or code was slightly different

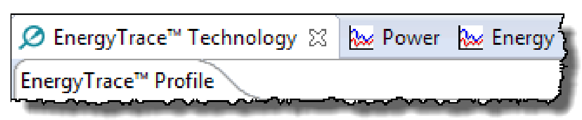
Finally, note that we have not yet optimized for power and the LED's that we are blinking (driven from our GPIO pins) are consuming quite a bit of energy.

15. Switch to the *Power* tab and see power consumption over time.



You might also want to check out the *Energy* tab. It shows running energy usage ofer timer.

16. Save the energy profile – naming it "Lab04a".



To view the EnergyTrace toolbar again, click back on the "EnergyTrace™ Technology" profile tab.

Then click the "Save Profile" EnergyTrace toolbar button and provide the name. (Use the default save-to directory.)

EnergyTrace with Free Run

Not surprisingly, the device hardware that supports many debugging features – such as breakpoints – requires energy to operate. Let's disable that hardware and capture another energy profile.

17. Make sure your program is suspended.

18. Set the cursor at the first line in the while{} and run to that line.

If you need a reminder how to do this, check back to steps 10-11 (on page 7-26).

19. Verify the EnergyTrace Capture duration is 10 seconds, then "Run Free".

This time, rather than hitting the Resume button, we want to run our target FREE of any emulation.

```
Run → Run Free
```

20. Watch the EnergyTrace count down to zero and then suspend the program again.

If you remember your program's previous energy consumption you may notice a reduction. But, we'll do a more accurate comparison in the next few steps.

21. Save the new EnergyTrace profile – give it the name `Lab4a_free_run`.

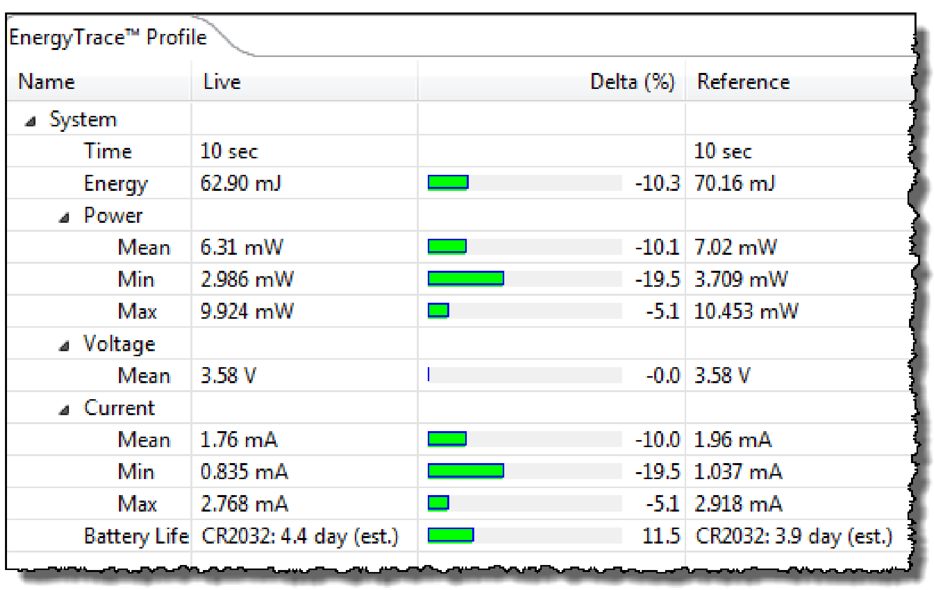
This isn't required, but it allows us to reference this information in a later comparison.

Compare EnergyTrace Profiles

22. Click on the Open button in the EnergyTrace toolbar.

Choose your first EnergyTrace profile: `Lab4a.profxml`

23. View the EnergyTrace profile comparison that opens.



EnergyTrace™ Profile

Name	Live	Delta (%)	Reference
System			
Time	10 sec		10 sec
Energy	62.90 mJ	-10.3	70.16 mJ
Power			
Mean	6.31 mW	-10.1	7.02 mW
Min	2.986 mW	-19.5	3.709 mW
Max	9.924 mW	-5.1	10.453 mW
Voltage			
Mean	3.58 V	-0.0	3.58 V
Current			
Mean	1.76 mA	-10.0	1.96 mA
Min	0.835 mA	-19.5	1.037 mA
Max	2.768 mA	-5.1	2.918 mA
Battery Life	CR2032: 4.4 day (est.)	11.5	CR2032: 3.9 day (est.)

This comparison shows that turning off the emulation features – using Run Free – saved more than 10mJ.

24. Write down the energy used for Lab4a_free_run profile: ____________________ mJ

25. Terminate the debug session.

26. Close the lab_04a_clock_solution project.

Create Energy Profile for lab_04c_crystals

27. Import the `lab_04c_crystals_solution.zip` into your workspace.

If you need a reminder on how to do this, please check back to Step 3 (page 7-23).

28. Build the project and start the debugger.

29. Run past the initialization code to the first line of the while{} loop.

For a reminder on how to do this, check back to steps 10-11 (on page 7-26).

30. Verify the EnergyTrace Capture duration is 10 seconds, then "Run Free".

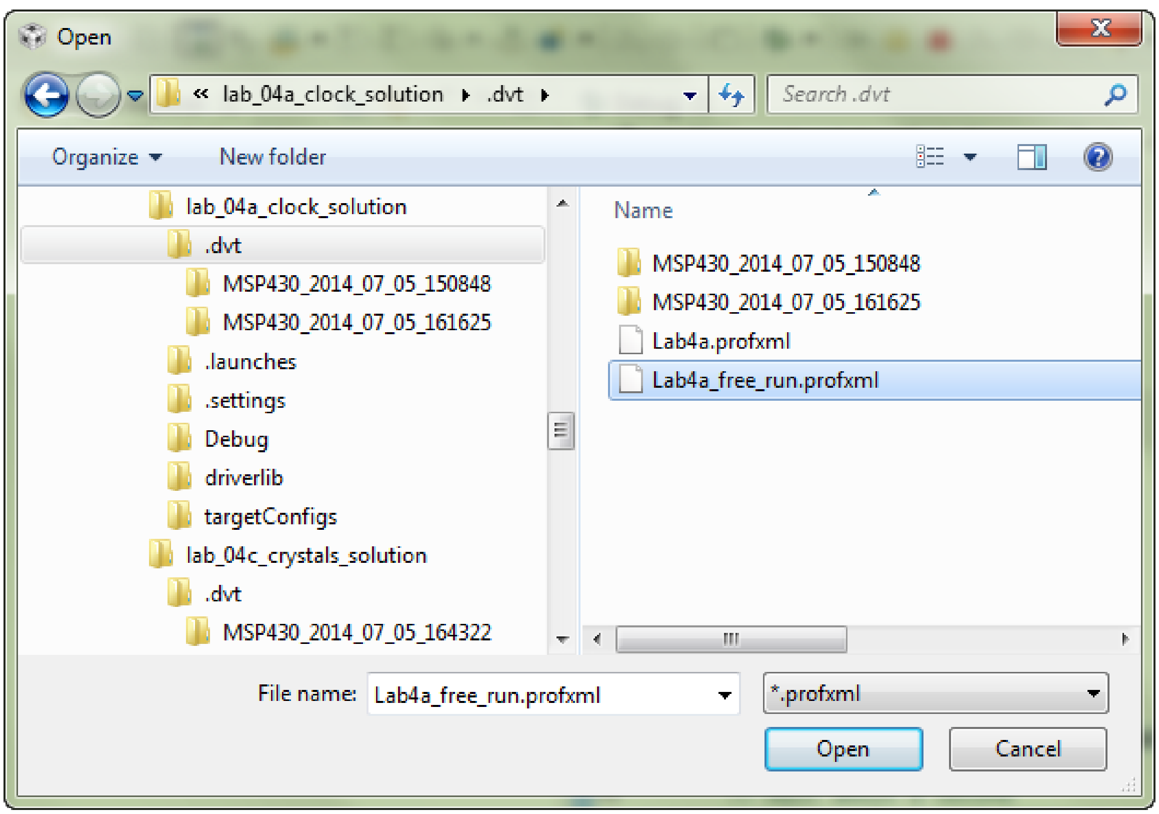
This time, rather than hitting the Resume button, we want to run our target FREE of any emulation.

```
Run → Run Free
```

31. Watch the EnergyTrace count down to zero and then suspend the program again.

32. Save the new EnergyTrace profile – give it the name `Lab4c_free_run`.

33. Open the the `Lab4a_free_run.profxml` energy profile to compare against Lab4c.



34. How do the two profiles compare?

Add your values to the chart below.

(Hint: You can copy the value for the Lab4a_free_run from step 24 (page 7-29).

Project Energy Profile	Time	Energy
Lab4a_free_run	10 sec	
Lab4c_free_run	10 sec	

Which version consumed less energy? ______________________________________

Why? ____

____

Hint: During the exercise steps for both Lab 4a and 4c we set breakpoints and recorded the values of three variables. What variables did we track … and how did they differ between Lab 4a and Lab 4c?

35. Terminate the debug session.

What have we learned in Lab7a?

- ☑ How to open archived project solutions
- ☑ *Enable* EnergyTrace
- ☑ Enable low-power debugging in projects.
- ☑ *Capture* and *Save* energy profiles
- ☑ Using *“Run Free”* to increase accuracy of energy capture profile
- ☑ Compare energy profiles

FR5969

(Optional) Viewing ‘FR5969 EnergyTrace++ States

Remember that the ‘FR58xx and ‘FR59xx devices support additional tracing of their internal CPU and peripheral states. Let’s examine this great new capability.

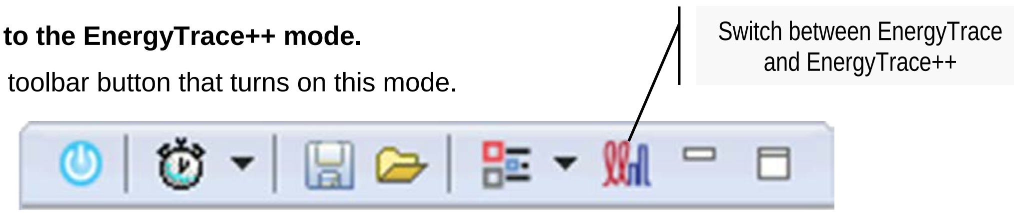
36. Open `lab_4c_crystal_solution` for debugging.

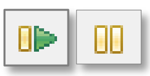
37. Verify that EnergyTrace is enabled.

You can do this via the CCS Preferences, though, it’s easier to simply check if the EnergyTrace window is open and the Start/Stop icon is “on” (that is, it should be blue).

38. Change to the EnergyTrace++ mode.

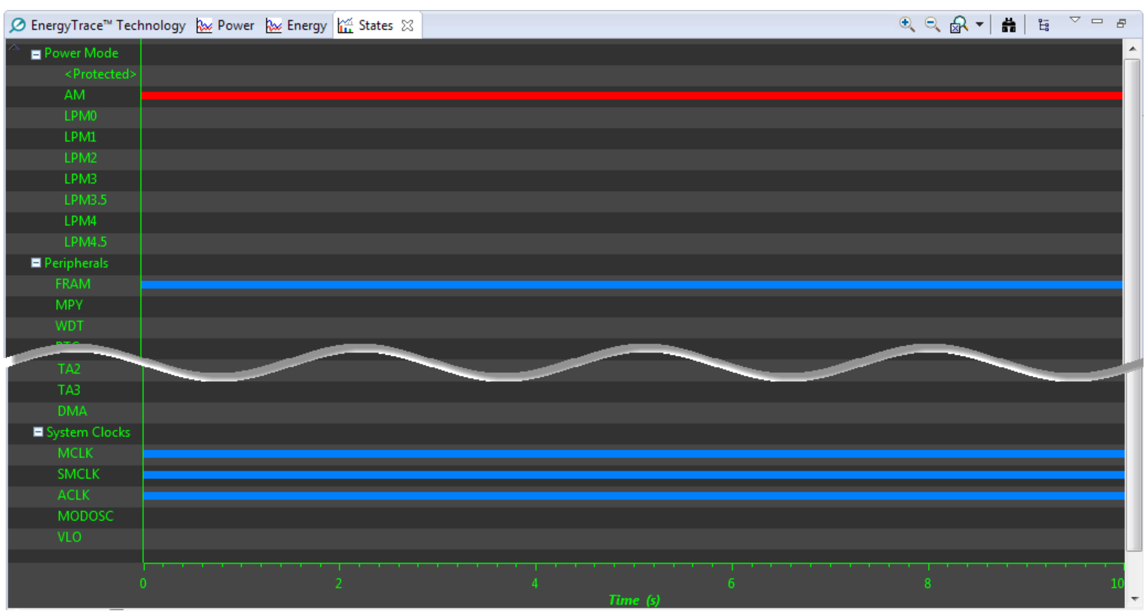
Click the toolbar button that turns on this mode.





39. Resume your program while letting EnergyTrace profile your code. Suspend when the EnergyTrace has finished counting down.

View the various tabs in the EnergyTrace window – note that a new one has been added showing the processor’s *“States”*.



Notice the following:

- We’re in Active Mode (AM) for the duration of the capture.
- Also, the FRAM is being accessed and all three clocks are running (MCLK, SMCLK, and ACLK).

Admittedly, this information becomes more interesting once we begin using the low-power modes and peripherals. But it’s fascinating to see how the processor is running internally.

Lab 7b – Reducing Power with ULP Advisor, LPM's and Interrupts

This exercise will start with the code we used from Lab 7a (which we imported from Lab 4c). Rather than just measuring power, though, we'll start to explore ways to reduce the program's power consumption.

Get Suggestions from ULP Advisor

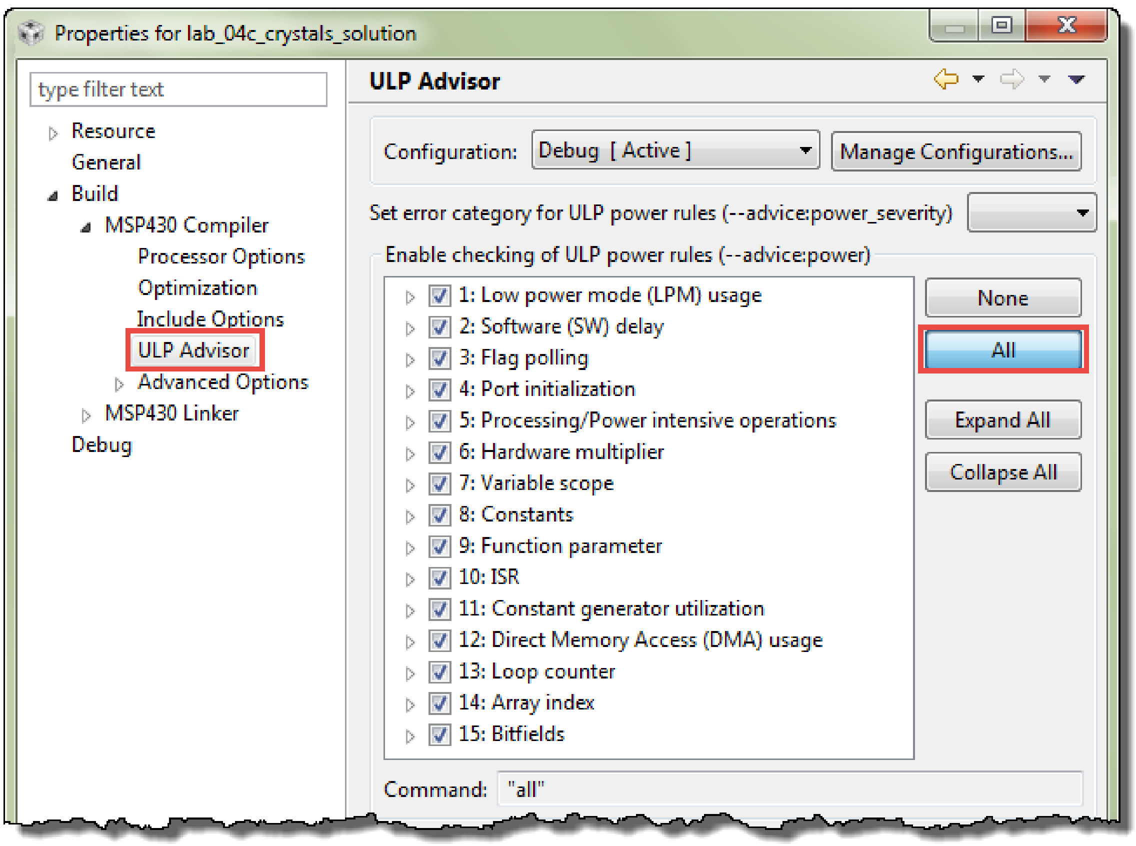
1. **Just to verify, all projects should be closed except `lab_4c_crystals_solution`; that is, the project we were just working with.**

2. **Turn on all of the ULP Advisor rules.**

   Select the project `lab_4c_crystals_solution`

   `Press the key combination` Alt — Enter

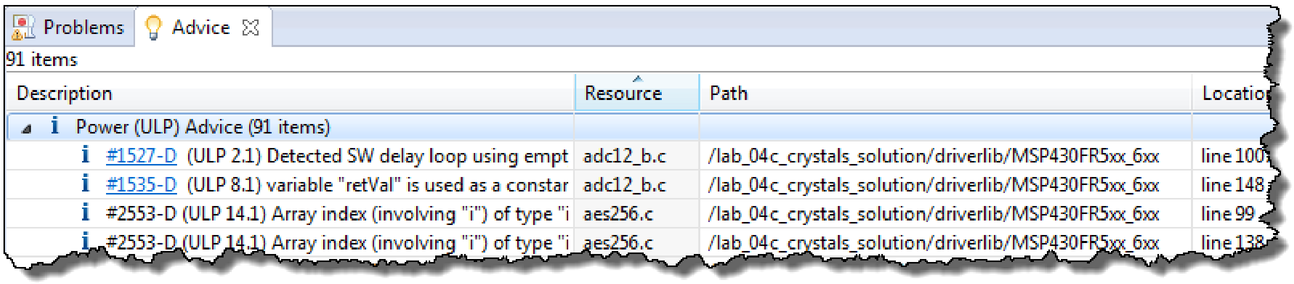
   And select *All* the rules, as shown below:



3. Build the project and then open the Advice window.

The Advice window is available by default in the standard CCS window; if not, open it with:

```
View → Advice
```

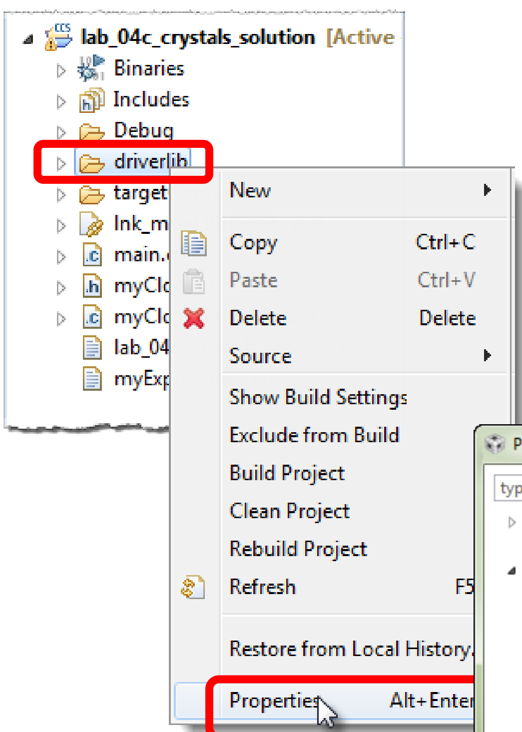


You results may vary based upon which processor you are using, but running with ULP Advisor, we received 91 items of advice. You may notice that most of the items relate to DriverLib code … further, most of them are related to peripheral source code that we're not even using in our program. (Thus, the linker will remove this from the final binary program.)

With some experience you will find that there will be times that ULP Advisor notes an item that you will want to ignore – maybe it's providing a false-positive, where you know that an item in your program just cannot be changed. Sometimes you will just choose to ignore the item, but often we can use CCS build options to filter them out (as we will do in the next step).

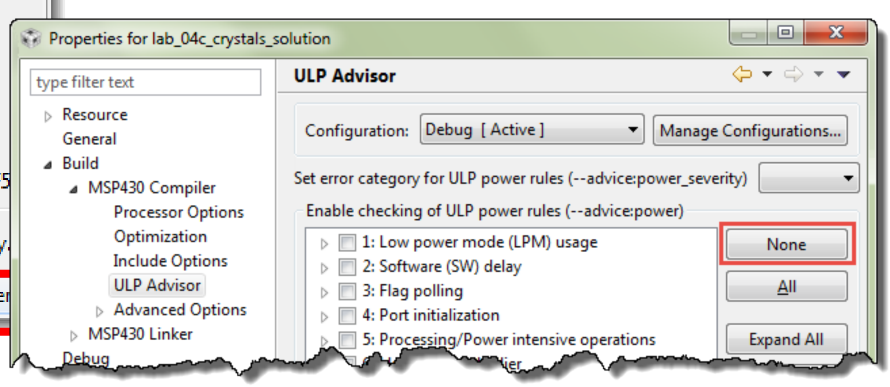
4. Modify the project options to focus ULP Advisor on our source code.

In other words, let's tell CCS not to rule ULP Advisor on MSP430ware DriverLib code. This can be done with ***file-specific*** project options.



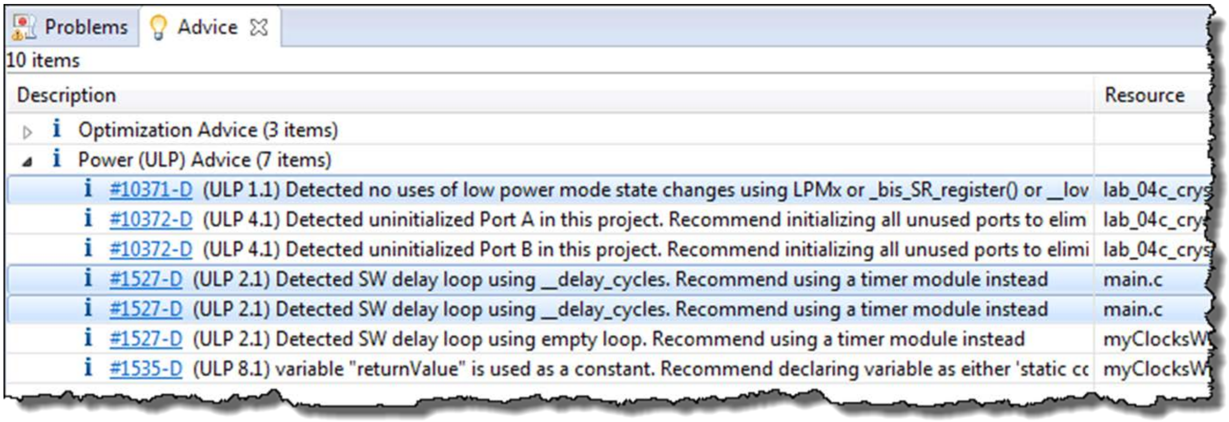

```
Right-click on the 'driverlib' folder
Select Properties
Click None
Click OK
```

This turns off the ULP Advisor option for all of the files in the 'driverlib' folder. In fact, you can use this feature to modify most all compiler option for any file or files.



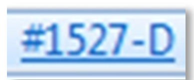
5. **Build the project again.**

Looking at *Power (ULP) Advice* for just our code, the list becomes more manageable.

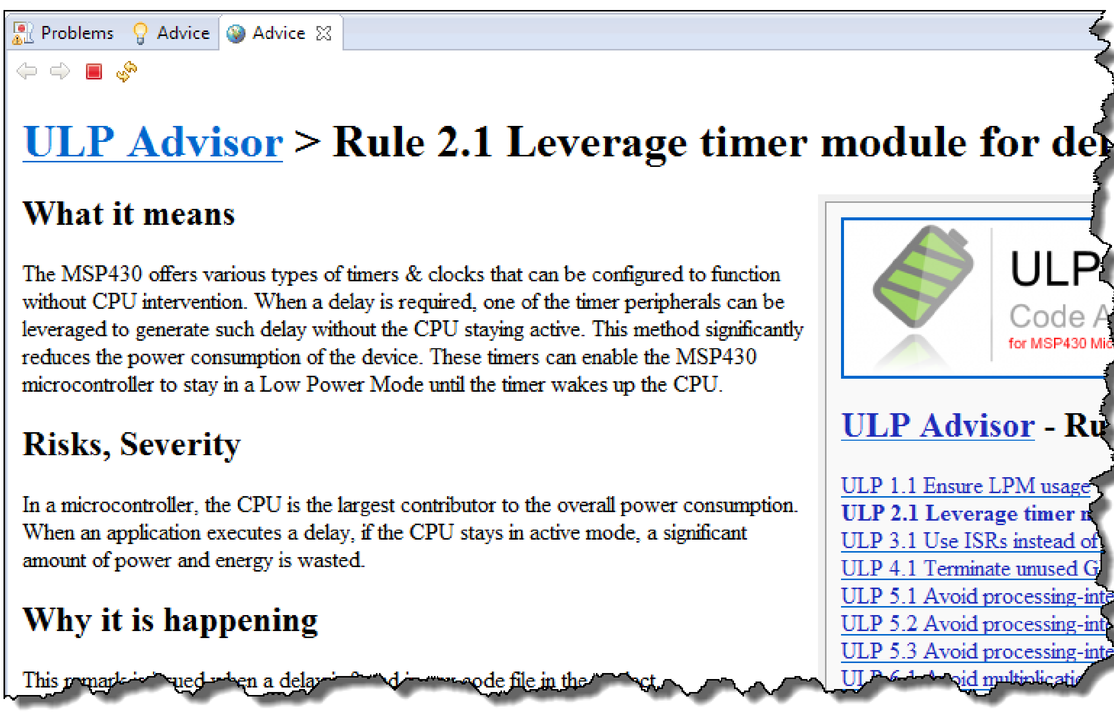


In Lab7b, we're plan to improve upon the items highlighted above; i.e. rules ULP 1.1 and 2.1.

6. **(Optional) If you have internet access, you can get more information for each rule by clicking on its link.**

`For example, clicking`  `takes you to...`

The wiki page which provides more information regarding rule ULP 2.1. This page explains the rule and tries to give you suggestions for improving your code.



Essentially, this rule is telling us that using the __delay_cycles() intrinsic is very power inefficient. *(This reinforces our warnings in previous lab projects where we admit that the code we asked to write was inefficient.)*

Replace __delay_cycles()

Let's begin by following the ULP 2.1 rule which tells us to replace __delay_cycles() by using a timer. This provides the advantage of letting the timer interrupt us, rather than the having the CPU count cycles in this inefficient intrinsic.

Also, using a timer will allow us (in the next section) to utilze one of the MSP430's low-power modes (LPMx).

7. **Complete the table of lab exercises (from Chapters 1 - 7) in this workshop which combined a timer with blinking an LED?**

Lab Exercise	Timer Module Used
lab_05b_wdtBlink	
lab_06a_timer	
lab_06b_upTimer	
lab_06c_timerDirectDriveLed	
lab_06d_simplePWM	'F5529: TimerA0 'FR5969: Timer_A1

In other words, we have already accomplished the task of swapping out __delay_cycles() with a timer. Rather than re-creating this code, we will import and use a previous solution.

8. **Close the `lab_04c_crystals_solution` project.**

9. **Import `lab_06b_upTimer_solution` into your workspace.**

   (Hint: If you need a reminder on how to do this, please check back to Step 3 on page 7-23.)

   We chose this exercise because:

   - The *Watchdog Timer* example was not implemented with the same LED blink rate, which will affect the energy comparisons.
   - TimerA's *Up* mode is more flexible than the *Continuous* mode (found in lab_06a_timer).
   - We're going to look at the 'DirectDrive' example a little bit later.
   - The PWM example was fancier than we needed for this exercise.

10. Rename the project to lab_07b_lpm_timer.

```
Right-click on the project → Rename
lab_07b_lpm_timer
```

11. Turn on ULP Advisor for the project. Turn it off for the '`driverlib`' folder.

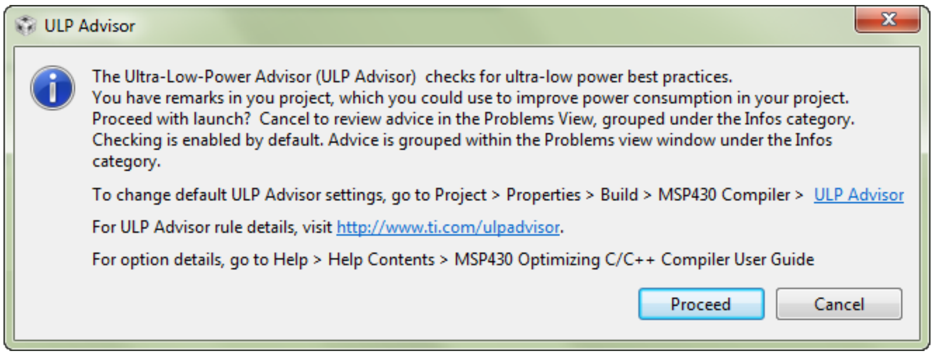
(Hint: If you need a reminder, look at Steps 2-4 (page 7-32) for how this was done.)

12. Build the project and examine the ULP Advisor suggestions.

Notice that the __delay_cycles() recommendations for `main.c` are now gone.

13. Start the debugger and load the program.

If you see this dialog, just click *Proceed*.



14. Verify that EnergyTrace is still enabled and set for a 10 second capture duration.

FR5969

15. ('FR5969 only) Verify that you are using the EnergyTrace mode (and not EnergyTrace++).

If you performed the optional exercise at the end of Lab 7a, your preferences may be set to EnergyTrace++ mode. While this provides additional States visibility, the emulator's use of power prevents us from getting accurate energy measurements.

Please go ahead and run the example with EnergyTrace++ mode. You should see that the TA1 peripheral is now active.

After trying ++ mode, though, please return to the EnergyTrace (non++) mode for the next part of the exercise.

16. Set your cursor in the while{} loop and "Run to Line".

Set your cursor on the __no_operation() intrinsic function and then run to that point – as we did earlier in the lab.

```
Run → Run to Line
```

Run your code with the Free Run command. After EnergyTrace captures the data (for 10 sec), suspend the program.

```
Run → Free Run
```

17. **Save the new energy profile as: `Lab7b_original.profxml`**

18. **Compare to the energy profile from `Lab4c_free_run.profxml`.**

    (Hint: Check back to Step 33 on page 7-29 for a reminder on how this was done.)

19. **Record the energy usage for each of these projects.**

Project Energy Profile	Time	Energy
Lab4c_free_run	10 sec	
Lab7b_original	10 sec	

Which project uses more power? ______________________________________

Why would our new project take more power after following the advice from ULP Advisor? What could account for the extra power it’s requiring?

(Hint: Let your lab_07b_lpm_timer project. Run it again… and watch the LED’s.)

____

____

20. **Terminate your debugging session.**

21. **Comment out the toggling of LED1.**

    Hopefully you figured out that our new Lab 7b project was toggling both LEDs, whereas the Lab4 project only toggled one LED. In this case, it isn’t the toggling function that draws too much power, but rather that we’re expending energy to drive both LEDs.

    To provide a fair comparison, we need to comment out one of the LED toggle functions. As an example, we arbitrarily choose to comment out the LED1 function.

    Open up the `myTimer.c` file and comment out the *GPIO_toggleOutputPin()* as shown here:

myTimers.c

```
//****** Interrupt Service Routines ********************
#pragma vector=TIMER1_A0_VECTOR
__interrupt void ccr0_ISR (void)
{
    // 4. Timer ISR and vector

    // Toggle LED1 on/off
   // GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN6 );
}
```

> **Note**
>
> Shown here to the left is the ‘FR5969 code.
>
> If using the ‘F5529, you’ll be using Timer0 and LED1 uses a different Port/Pin.

22. **Build your project and fix any syntax errors.**

23. **Start the debugger and then run to the *__no_operation()* inside the *while{}* loop.**

Control R

24. Free Run your program and then click suspend when the EnergyTrace timer finishes counting down from 10 seconds.

25. Save the new energy profile as: `Lab7b_one_led`

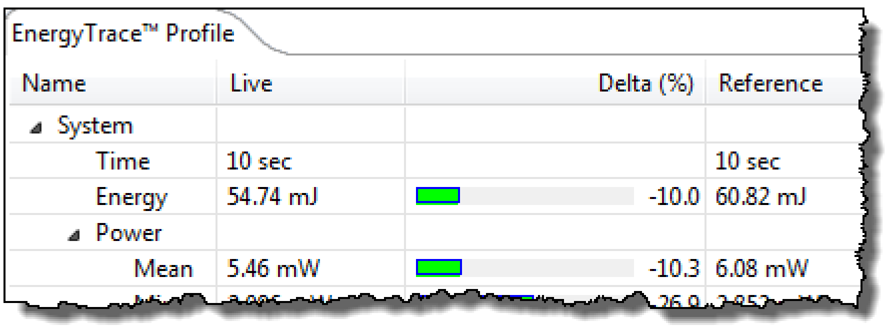
Once again, compare this to the Lab4c energy profile.

Project Energy Profile	Time	Energy
Lab4c_free_run	10 sec	
Lab7b_one_led	10 sec	

Which project uses more power? ________________________________

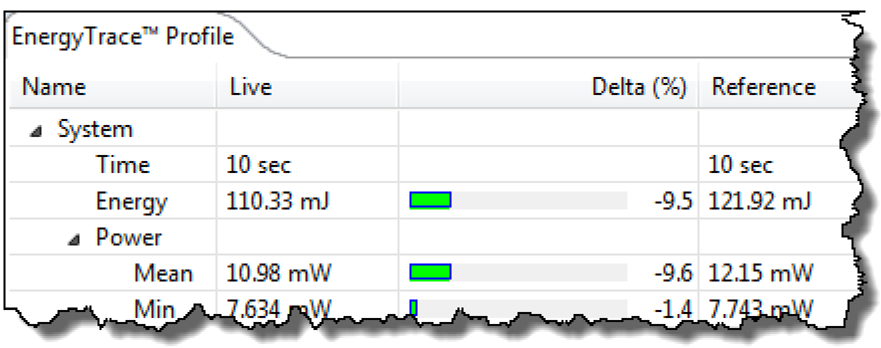
Here's the comparison we found for the 'FR5969 at the time of writing this exercise. As you can see below, using the timer (versus the CPU running __delay_cycles) saved us 10% of our energy.

FR5969


EnergyTrace™ Profile

Name	Live	Delta (%)	Reference
System			
Time	10 sec		10 sec
Energy	54.74 mJ	-10.0	60.82 mJ
Power			
Mean	5.46 mW	-10.3	6.08 mW

F5529


EnergyTrace™ Profile

Name	Live	Delta (%)	Reference
System			
Time	10 sec		10 sec
Energy	110.33 mJ	-9.5	121.92 mJ
Power			
Mean	10.98 mW	-9.6	12.15 mW
Min	7.634 mW	-1.4	7.743 mW

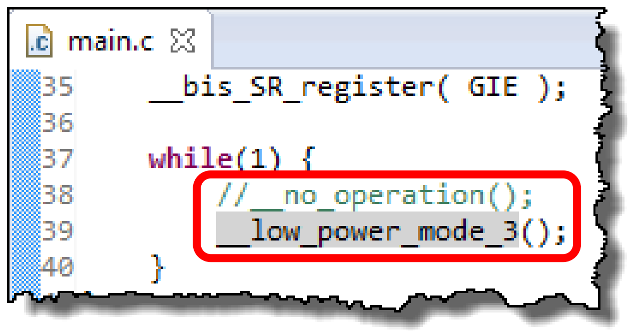
Using Low-Power Mode (LPM3)

Once you've built your program to be interrupt-driven, it's often quite easy to utilize the MSP430 low-power modes.

We chose to use Low-Power Mode 3 (LPM3) because it provides a very low standby power, keeps ACLK running (which we're using to clock Timer_A), and makes it easy to return to Active Mode when an interrupt occurs.

26. Modify `lab_07b_lpm_timer` to use LPM3.

In the program, you only need to replace __no_operation() with __low_power_mode_3().



As we learned during the Chapter 7 discussion:

– Executing the _low_power_mode_3() function changes a few bits in the Status Register (SR), therefore putting the CPU into LPM3.

– The processor remains in that state until an interrupt occurs.

– Interrupt ISR's automatically save and restore the SR context; therefore, unless we alter the normal ISR flow, the CPU will automatically return to LPM3 upon exiting the ISR.

This means, we don't need the while(1){} loop anymore, but it doesn't hurt to leave it there.

27. Build your code and fix any syntax errors.

28. Start the debugger.

Control R **29. Set your cursor on the `__low_power_mode_3()` function and then run to that line.**

30. *Free Run* your code and then *Suspend* after the EnergyTrace capture duration.

31. Save the new energy profile as: `Lab7b_lpm`

32. Compare the current energy profile to your previous one.

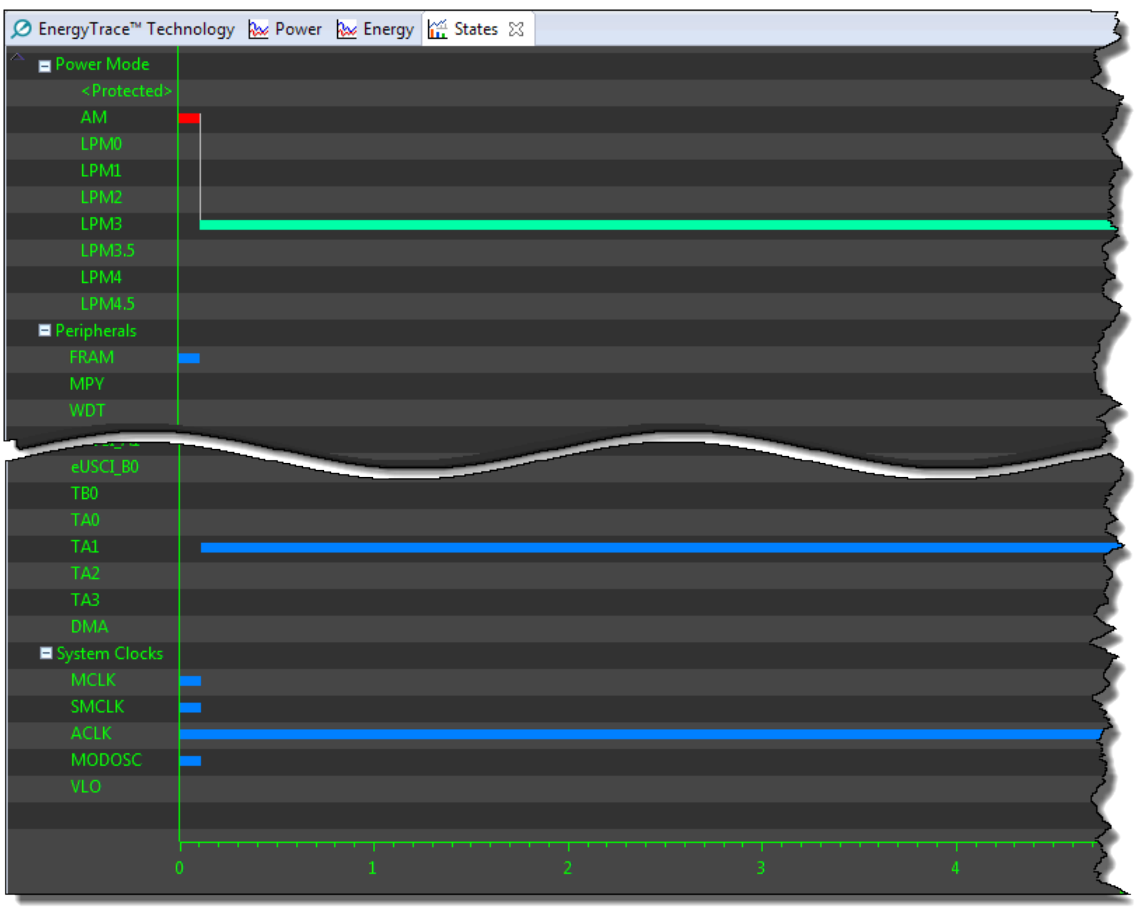
Project Energy Profile	Time	Energy
Lab7b_one_led	10 sec	
Lab7b_lpm	10 sec	

Which profile uses less power? ____

Our 'FR6969 results show another 20% savings in energy by utilizing LPM3; while the 'F5529 LPM3 results in amost 70% savings.

(Optional) Viewing 'FR5969 EnergyTrace++ States

If you are using the "FR5969, try running EnergyTrace++ again with the `lab_07b_lpm_timer` project. The States is now more interesting since you can see the changes in the clocks and CPU modes.



(Optional) Directly Driving the LED from Timer_A

Note: We suggest that you skip this option lab exercise and continue on to Lab 7c. Then, if you still have time after completing Lab 7c, you can try out this experiment.

Another interesting energy comparison would be a comparison between, effectively, a comparison between lab_06b_upTimer and lab_06c_timerDirectDrive. In other words, can you reduce power if you take away the CPU interrupt service routine and let the timer drive the LED directly.

Rather than provide detailed, step-by-step directions for this optional exercise, we've written down a few notes and will let you work through the details on your own.

Rough lab exercise procedural

- Import `lab_06c_ledDirectDrive_solution.zip` into CCS and rename imported project to `lab_07b_timerDirectDrive`.
- As with our previous exercise, change the following two lines of code:
  - Comment out code that toggles LED2 in timer ISR
  - Replace __no_operation() function with LPM3 function call.
- Build and profile the energy usage

  By the way, don't forget to connect LED1 to the timer output pin using a jumper wire. Please see Lab 6c, if you have questions about how to connect the jumper wire.
- Compare to lab_07b_lpm_timer energy profile results

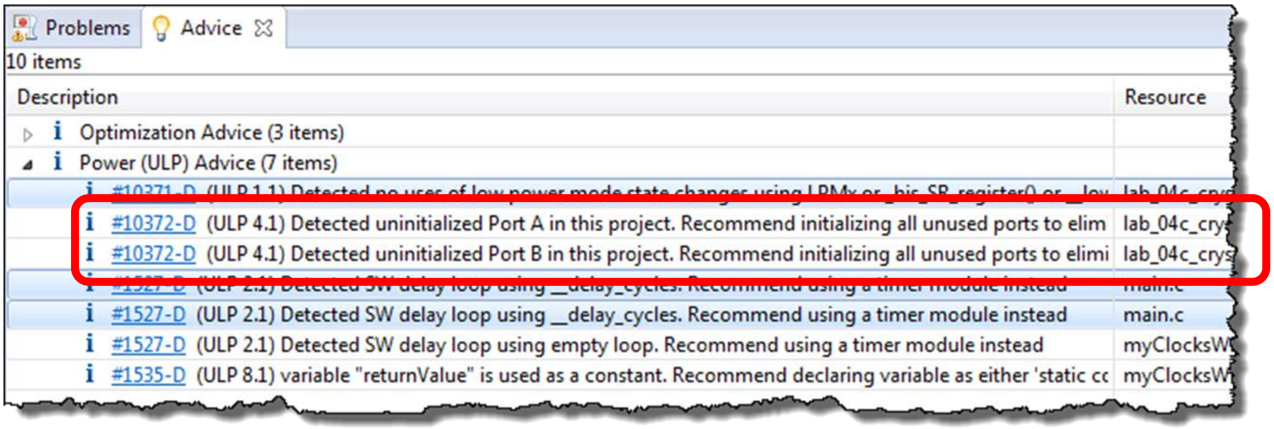
  When we did this, we found that (using the 'FR5969 Launchpad) the directly driven LED project took quite a bit more energy … these results shocked us.

  The key to our understanding this was to look at the Power graph differences between the projects. We noted that the LED for one project consumed a lot more energy than for the other project.
- Go back to `lab_07b_lpm_timer` and redo that lab exercise driving the other LED. In other words, we wanted to make sure both labs are driving the same LED to get a better apples-to-apples comparison.

  When we did this, we found that directly driving the LED save a minute amount of energy.

Lab 7c – Configuring Ports for Lowest Power

One of the other items ULP Advisor remarked was that our GPIO ports had not been properly initialized. Referring back to Lab 7b Step 5 (on page 7-34), it's listed as rule ULP 4.1.



Once again, we're going to start with lab_04c_crystals and explore what affect GPIO initialization might have on our system.

Import and Modify Program

1. **Terminate the debugger if it running and close all open projects and files.**

2. **Open project: `lab_04c_crystals_solution`**

3. **Copy the project `lab_04c_crystals_solution` and rename it `lab_07c_initPorts`.**

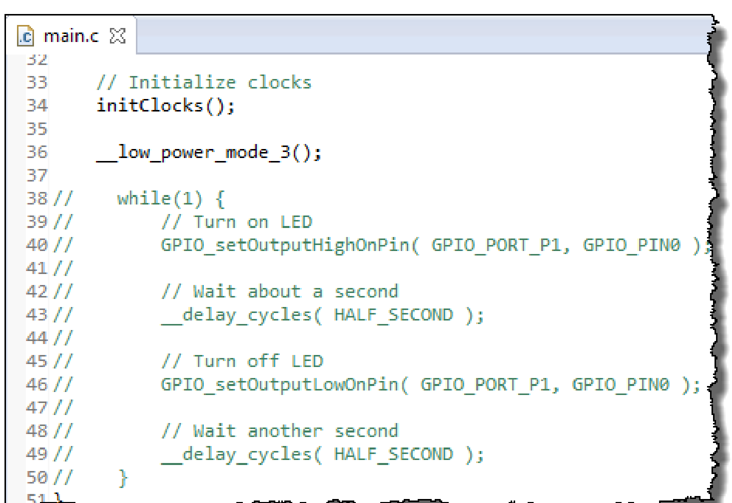
   a) In CCS Project Explorer, `right-click and copy lab_04c_crystals_solution`

   b) Then `right-click and paste it`

   c) Enter the new name `lab_07c_initPorts` when CCS requests it

4. **Replace the while{} loop with LPM3.**

   To focus specifically on the affects of GPIO initialization, we suggest removing the code that blinks the LED – replacing it with a call to *__low_power_mode_3()*.



Capture Baseline Reference

5. **Build the project. Once any errors are fixed, launch the debugger.**

6. **Run the code until you reach the LPM3 function.**

   Set the cursor on the __low_power_optimization() function and then press Control R

7. **Free Run the program until the EnergyTrace capture has completed. Save the energy profile as `Lab7c_noinit.profxml` and record the energy data.**

   We'll fill in the 2nd and 3rd rows of this table in upcoming lab steps.

Project Energy Profile	Capture Duration Time	Energy (mJ)	Battery Life (Days)
Lab7c_noinit	10 sec		
Lab7c _initPortsAsOutputs	10 sec		
Lab7c_initPortsAsInputs	10 sec		

Add GPIO Port Initialization Code

Rather than ask you to type the same functions over and over again, we have already created a port initialization file for you. The functions were the same ones discussed in Chapter 3, although we utilized #ifdef statements to allow the same file to be used for most any MSP430 device.

8. **Terminate your debug session if it's running.**

9. **Add three new files to your project.**

```
Right-click on the project → Properties
Add Files…
```

   Navigate to the appropriate directory for you processor:

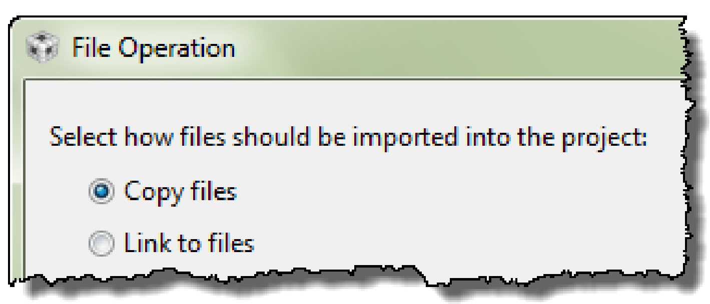
```
C:\msp430_workshop\<target\lab_07c_ports
```

   Select the following three files and click *Open*.

```
initPortsAsOutputs.c
initPorts.h
lab_07c_initPorts_readme.txt
```

   When the Copy/Link dialog appears, select "*Copy*" and click OK.

   You can delete the old readme file, if you'd like.



10. Open and examine the `initPortsAsOutputs.c` function.

Notice that each port, if found for that device, is set so that all of the GPIO pins are set as outputs in a low state.

11. Add *initPorts()* function call to `main.c`.

While we've added the files to the project, we haven't add the call to the initPorts() function, yet. Immediately after the Watchdog hold function, add the new function to your program.

```
// Initialize I/O Ports
initPorts();
```

Make sure you the new *initPorts()* function comes **<u>before</u>** the call to *initGPIO()*. We wrote the *initPorts()* function to be a generic initialization routine, whereas the *initGPIO()* function sets only the specific GPIO pins we need for our program.

While we could combine these files, it is often useful – especially during development – to use a baseline initialization routine at the beginning of your program.

Your main() function should now look like this:

```
*main.c
// ------------------------------------------------------------------
// main.c  (for lab_07c_ports project)
// ------------------------------------------------------------------

//***** Header Files ***************************************
#include <driverlib.h>
#include "myClocks.h"
#include "initPorts.h"

//***** Prototypes *****************************************
void initGPIO(void);


//***** Main Function **************************************
void main (void)
{
    // Stop watchdog timer
    WDT_A_hold( WDT_A_BASE );

    // Initialize I/O Ports
    initPorts();

    // Initialize GPIO
    initGPIO();

    // Initialize clocks
    initClocks();

    __low_power_mode_3();

//    while(1) {
//        // Turn on LED
//        GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
//
```

12. Build the project. Once any errors are fixed, launch the debugger.

13. Run the code until you reach the LPM3 function.

Set the cursor on the __low_power_optimization() function and then press Control R

14. Free Run the program until the EnergyTrace capture has completed. Save the energy profile as `Lab7c_initPortsAsOutputs.profxml` and record the energy data.

Fill in the 2nd row of the table found in Step 7 on page 7-43.

Does initializing the I/O ports make much of a difference to energy consumption?

Improve on GPIO Port Initialization

While working on this lab exercise we found that our port initialization routine could be improved upon. This last part of the exercise quickly examines this.

15. Add one more file to your project: `initPorts.c`

Follow the same steps as before to add this file – making sure you "*Copy*" the file into your project

16. Open and briefly examine `initPorts.c`.

This file includes the same *initPorts()* function, although it configures GPIO in a different mode. Rather than setting the GPIO pins as outputs, how does this new routine configure them?

---

17. Exclude from build...

If you were to try and build the project right now, you should get an error. The *initPorts()* function is defined twice. Rather than deleting one copy, we suggest that you just exclude one file from being built.

`Right-Click on the file` *`initPortsAsOutputs.c`* `→ Exclude From Build`

Now, when we click *Build*, CCS will ignore this file.

18. Build the project. Once any errors are fixed, launch the debugger.

19. Run the code until you reach the LPM3 function.

Set the cursor on the __low_power_optimization() function and then press Control R

20. Free Run the program until the EnergyTrace capture has completed. Save the energy profile as `Lab7c_initPortsAsInputs.profxml` and record the energy data.

Fill in the 3rd row of the table found in Step 7 on page 7-43.

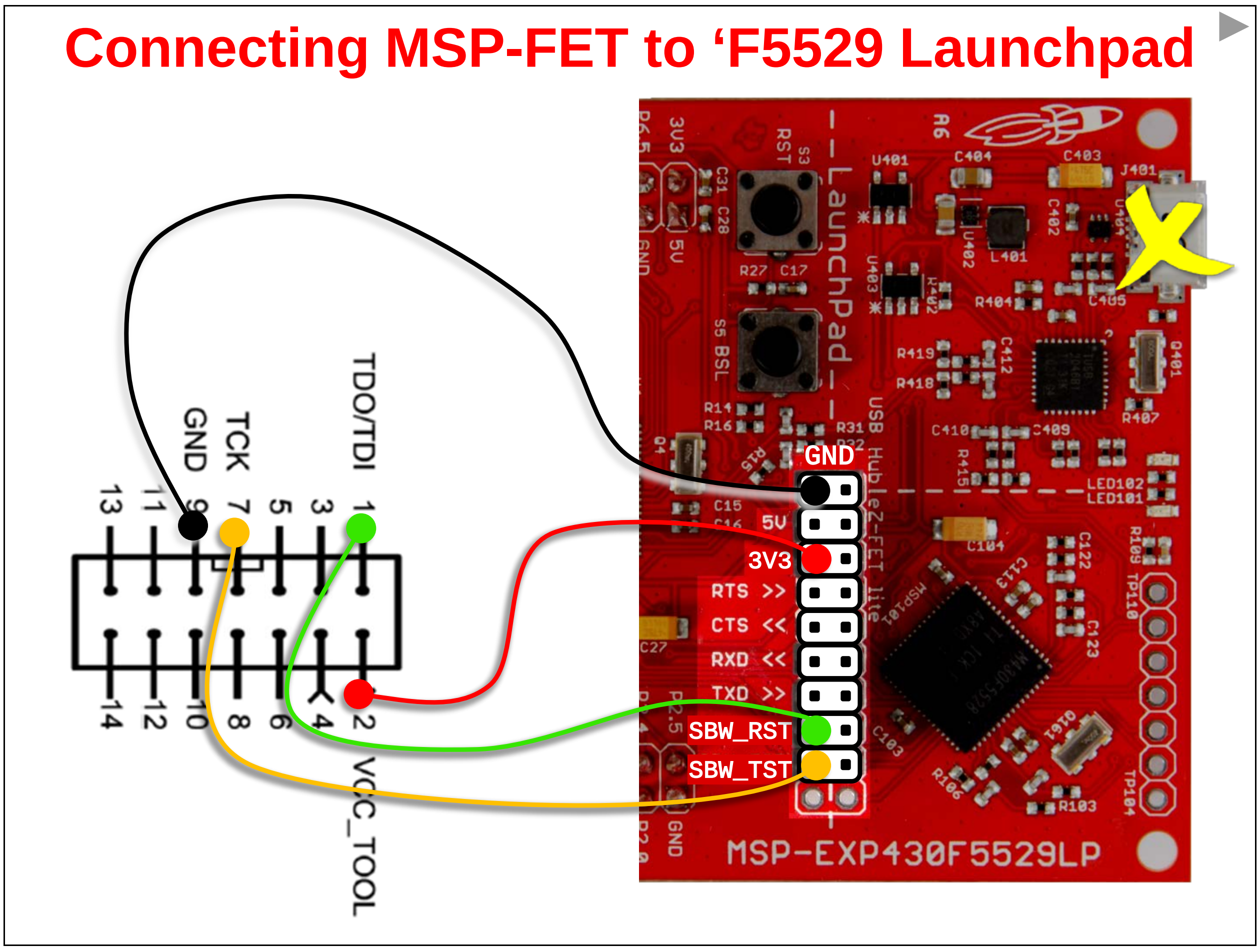
Does initializing the I/O ports as inputs (with a pulldown resistor) make much of a difference?

Chapter 7 Appendix

Connecting MSP-FET to ‘F5529 USB Launchpad

Using the following two User’s Guide, we determined that you can connect the MSP-FET flash emulation tool to the MSP-EXP430F5529 Launchpad’s isolation connector.

- MSP-EXP430F5529 Launchpad User's Guide (slau533b.pdf)
- MSP430 Hardware Tools User’s Guide (slau278r.pdf)



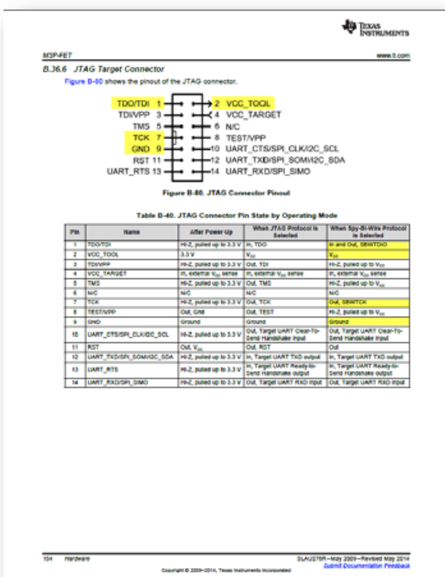
MSP-FET to 'F5529 Launchpad Summary of Pin Connections

MSP-FET Signal	MSP-FET Pin		'F5529 Launchpad (Isolation Jumper Block) Signal	'F5529 Launchpad (Isolation Jumper Block) Pin
GND	9	→	GND	JP3
VCC_TOOL	2	→	3V3	JP2
TDO/TDI	1	→	SBW_RST	JP4.2
TCK	7	→	SBW_TST	JP4.1

MSP430 Hardware Tools User's Guide (SLAU287r.PDF)
B.36.6 MSP-FET JTAG Target Connector (pg 154)
Table B-40: JTAG Connector Pin State by Operating Mode

MSP-EXP430F5529 Launchpad User's Guide (SLAU533b .PDF)
2.2.7 Emulator and Target Isolation Jumper Block
Table 3: Isolation Block Connections (pg 19)

User Guide Reference Pages


TEXAS INSTRUMENTS

MSP-FET www.ti.com

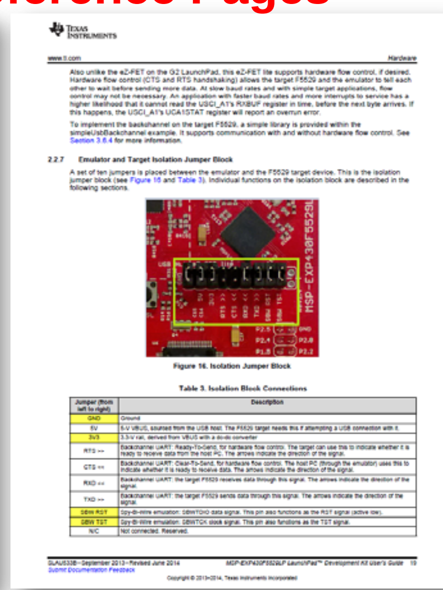
B.36.6 JTAG Target Connector

Figure B-80 shows the pinout of the JTAG connector.

Figure B-80. JTAG Connector Pinout

Table B-40. JTAG Connector Pin State by Operating Mode

Pin	Name	After Power Up	When JTAG Protocol is Selected	When Spy-Bi-Wire Protocol is Selected
1	TDO/TDI	Hi-Z, pulled up to 3.3 V	In, TDO	In and Out, SBWTDIO
2	VCC_TOOL	3.3 V	V_{CC}	V_{CC}
3	TDI/VPP	Hi-Z, pulled up to 3.3 V	Out, TDI	Hi-Z, pulled up to V_{CC}
4	VCC_TARGET	In, external V_{CC} sense	In, external V_{CC} sense	In, external V_{CC} sense
5	TMS	Hi-Z, pulled up to 3.3 V	Out, TMS	Hi-Z, pulled up to V_{CC}
6	N/C	N/C	N/C	N/C
7	TCK	Hi-Z, pulled up to 3.3 V	Out, TCK	Out, SBWTCK
8	TEST/VPP	Out, GND	Out, TEST	Hi-Z, pulled up to V_{CC}
9	GND	Ground	Ground	Ground
10	UART_CTS/SPI_CLK/I2C_SCL	Hi-Z, pulled up to 3.3 V	Out, Target UART Clear-To-Send Handshake input	Out, Target UART Clear-To-Send Handshake input
11	RST	Out, V_{CC}	Out, RST	Out
12	UART_TXD/SPI_SOMI/I2C_SDA	Hi-Z, pulled up to 3.3 V	In, Target UART TXD output	In, Target UART TXD output
13	UART_RTS	Hi-Z, pulled up to 3.3 V	In, Target UART Ready-to-Send Handshake output	In, Target UART Ready-to-Send Handshake output
14	UART_RXD/SPI_SIMO	Hi-Z, pulled up to 3.3 V	Out, Target UART RXD input	Out, Target UART RXD input


TEXAS INSTRUMENTS

www.ti.com Hardware

Also unlike the eZ-FET on the G2 LaunchPad, this eZ-FET lite supports hardware flow control, if desired. Hardware flow control (CTS and RTS handshaking) allows the target F5529 and the emulator to tell each other to wait before sending more data. At slow baud rates and with simple target applications, flow control may not be necessary. An application with faster baud rates and more interrupts to service has a higher likelihood that it cannot read the USCI_A1's RXBUF register in time, before the next byte arrives. If this happens, the USCI_A1's UCA1STAT register will report an overrun error.

To implement the backchannel on the target F5529, a simple library is provided within the simpleUsbBackchannel example. It supports communication with and without hardware flow control. See Section 3.6.4 for more information.

2.2.7 Emulator and Target Isolation Jumper Block

A set of ten jumpers is placed between the emulator and the F5529 target device. This is the isolation jumper block (see Figure 16 and Table 3). Individual functions on the isolation block are described in the following sections.

Figure 16. Isolation Jumper Block

Table 3. Isolation Block Connections

Jumper (from left to right)	Description
GND	Ground
5V	5-V VBUS, sourced from the USB host. The F5529 target needs this if attempting a USB connection with it.
3V3	3.3-V rail, derived from VBUS with a dc-dc converter
RTS >>	Backchannel UART: Ready-To-Send, for hardware flow control. The target can use this to indicate whether it is ready to receive data from the host PC. The arrows indicate the direction of the signal.
CTS <<	Backchannel UART: Clear-To-Send, for hardware flow control. The host PC (through the emulator) uses this to indicate whether it is ready to receive data. The arrows indicate the direction of the signal.
RXD <<	Backchannel UART: the target F5529 receives data through this signal. The arrows indicate the direction of the signal.
TXD >>	Backchannel UART: the target F5529 sends data through this signal. The arrows indicate the direction of the signal.
SBW RST	Spy-Bi-Wire emulation: SBWTDIO data signal. This pin also functions as the RST signal (active low).
SBW TST	Spy-Bi-Wire emulation: SBWTCK clock signal. This pin also functions as the TST signal.
N/C	Not connected. Reserved.

SLAU533B–September 2013–Revised June 2014 MSP-EXP430F5529LP LaunchPad™ Development Kit User's Guide 19
Submit Documentation Feedback
Copyright © 2013–2014, Texas Instruments Incorporated

MSP430 Hardware Tools User's Guide (SLAU287r.PDF)
B.36.6 MSP-FET JTAG Target Connector (pg 154)
Table B-40: JTAG Connector Pin State by Operating Mode

MSP-EXP430F5529 Launchpad User's Guide (SLAU533b .PDF)
2.2.7 Emulator and Target Isolation Jumper Block
Table 3: Isolation Block Connections (pg 19)

Notes

Lab 7 Debrief and Solutions

Lab 7a - Worksheet

1. What is the difference between Energy Trace and Energy Trace++?

   Both support energy measurement; EnergyTrace++ also supports tracing CPU and peripheral states

   Which devices support Energy Trace++? MSP430FR5xxx devices

2. What hardware options are available that supports Energy Trace?

   'FR5969 Launchpad and any MSP430 connected to MSP-FET

3. How can you calculate energy without Energy Trace? Use a multi-meter to measure current drawn by CPU multiplied by voltage and time

   What is the downside to this method? Not as accurate as EnergyTrace

Lab 7a – Debrief ('FR5969)

34. How do the two profiles compare?

Add your values to the chart below.

(Hint: You can copy the value for the Lab4a_free_run from step 24 (page 7-16).

Project Energy Profile	Time	Energy
Lab4a_free_run	10 sec	62.90 mJ
Lab4c_free_run	10 sec	54.01 mJ

Which version consumed less energy? Lab4c

Why? The MSP430 clocks in `lab_04c_crystals` were running at a lower frequency, which consumes less power

Hint: During the exercise steps for both Lab 4a and 4c we set breakpoints and recorded the values of three variables. What variables did we track … and how did they differ between Lab 4a and Lab 4c?

Lab 7a – Debrief ('F5529)

34. How do the two profiles compare?

Add your values to the chart below.

(Hint: You can copy the value for the Lab4a_free_run from step 24 (page 7-16).

Project Energy Profile	Time	Energy
Lab4a_free_run	10 sec	118.28 mJ
Lab4c_free_run	10 sec	121.92 mJ

Which version consumed less energy? Very close, but Lab4a is slightly less

Why? The two are essentially equal; the differences in clock speed (4a to 4c) are less than they are for the 'FR5969 solutions.

Hint: During the exercise steps for both Lab 4a and 4c we set breakpoints and recorded the values of three variables. What variables did we track … and how did they differ between Lab 4a and Lab 4c?

Lab 7b

7. **Complete the table of lab exercises (from Chapters 1 - 7) in this workshop which combined a timer with blinking an LED?**

Lab Exercise	Timer Module Used
lab_05b_wdtBlink	Watchdog (Interval Timer mode)
lab_06a_timer	'F5529: TimerA0 'FR5969: Timer_A1
lab_06b_upTimer	'F5529: TimerA0 'FR5969: Timer_A1
lab_06c_timerDirectDriveLed	'F5529: TimerA0 'FR5969: Timer_A1
lab_06d_simplePWM	'F5529: TimerA0 'FR5969: Timer_A1

Lab 7b

'F5529 values are shown here

19. Record the energy usage for each of these projects.

Project Energy Profile	Time	Energy
Lab4c_free_run	10 sec	121.92 mJ
Lab7b_original	10 sec	146.26 mJ

Which project uses more power? The timer code (Lab7b)

Why would our new project take more power after following the advice from ULP Advisor? What could account for the extra power it's requiring?

(Hint: Let your lab_07b_lpm_timer project. Run it again… and watch the LED's.)

Watching Lab7b run, you might notice that both LEDs are blinking – whereas in Lab4c, only one is blinking

Lab 7b

'F5529 values are shown here

32. Compare the current energy profile to your previous one.

Project Energy Profile	Time	Energy
Lab7b_one_led	10 sec	110.33 mJ
Lab7b_lpm	10 sec	34.81 mJ

Which profile uses less power? Lab7b_lpm is much better

Our 'FR6969 results show another 20% savings in energy by utilizing LPM3; while the 'F5529 LPM3 results in amost 70% savings.

Lab 7c ('FR5969)

7. **Free Run the program until the EnergyTrace capture has completed. Save the energy profile as `Lab7c_noinit.profxml` and record the energy data.**

We'll fill in the 2nd and 3rd rows of this table in an upcoming lab step.

Project Energy Profile	Capture Duration Time	Energy (mJ)	Battery Life (Days)
Lab7c_noinit	10 sec	11.28	24.4
Lab7c _initPortsAsOutputs	10 sec	0.14	1920.4
Lab7c_initPortsAsInputs	10 sec	0.01	24553.6

Steps 13/19 asked if initializing the GPIO (and init as inputs) made much of a different to energy usage… Absolutely YES!

Lab 7c ('F5529)

7. **Free Run the program until the EnergyTrace capture has completed. Save the energy profile as `Lab7c_noinit.profxml` and record the energy data.**

We'll fill in the 2nd and 3rd rows of this table in an upcoming lab step.

Project Energy Profile	Capture Duration Time	Energy (mJ)	Battery Life (Days)
Lab7c_noinit	10 sec	8.03	34.2
Lab7c _initPortsAsOutputs	10 sec	7.47	36.8
Lab7c_initPortsAsInputs	10 sec	7.47	36.8

Steps 13/19 asked if initializing the GPIO made much of a different to energy usage… a little bit. On the 'F5529, though, no noticeable difference if GPIO was set as outputs or inputs (unlike the 'FR5969).