

MSP430 Workshop

STUDENT GUIDE



*MSP430
Workshop
Revision 3.10
Feb/March 2014*

Important Notice

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Copyright © 2013 Texas Instruments Incorporated

Revision History

July 2013 – Revision 2.22 (based on MSP430G2553 Value-Line Launchpad)

October 2013 – Revision 3.0 (based on MSP430F5529 USB Launchpad)

November 2013 – Revision 3.01

January 2014 – Revision 3.02

February 2014 – Revision 3.10 (based on MSP430F5529 & MSPFR5969 Launchpad's)

Mailing Address

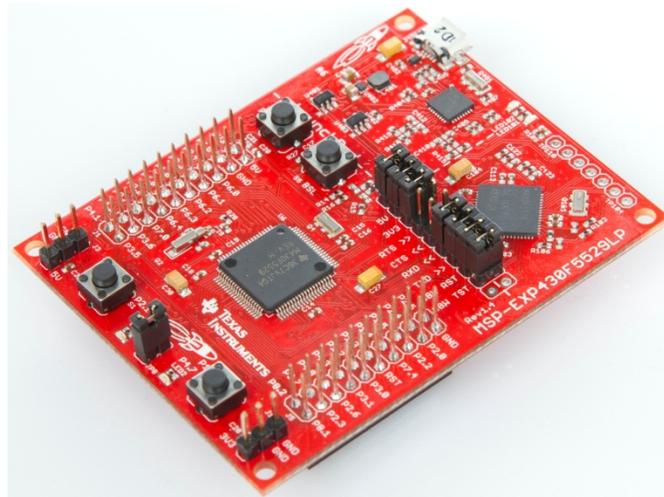
Texas Instruments
Training Technical Organization
6500 Chase Oaks Blvd Building 2
M/S 8437
Plano, Texas 75023

Introduction to MSP430

Introduction

Welcome to the MSP430 Workshop. This workshop covers the fundamental skills needed when designing a system based on the Texas Instruments (TI) MSP430™ microcontroller (MCU). This workshop utilizes TI's integrated development environment (IDE) which is named Code Composer Studio™ (CCS). It will also introduce you to many of the libraries provided by TI for rapid development of microcontroller projects, such as MSP430ware™.

Whether you are a fan of the MSP430 for its low-power DNA, appreciate its simple RISC-like approach to processing, or are just trying to keep your system's cost to a minimum ... we hope you'll enjoy working through this material as you learn how to use this nifty little MCU.



MSP-EXP430F5529LP Launchpad



MSP-BNDL-FR5969LCD

Chapter Topics

Introduction to MSP430	1-1
<i>Administrative Topics</i>	1-3
<i>Workshop Agenda</i>	1-4
<i>TI Products</i>	1-6
TI's Entire Portfolio	1-6
Wireless Products	1-7
<i>TI's Embedded Processors</i>	1-8
<i>MSP430 Family</i>	1-10
<i>MSP430 CPU</i>	1-13
<i>MSP430 Memory</i>	1-17
Memory Map	1-17
FRAM	1-20
<i>MSP430 Peripherals</i>	1-23
GPIO	1-23
Timers	1-24
Clocking and Power Management	1-25
Analog	1-26
Communications (Serial ports, USB, Radio)	1-27
Hardware Accelerators	1-28
Summary	1-29
<i>ULP</i>	1-30
Profile Your Activities	1-31
<i>Launchpad's</i>	1-35
MSP-EXP-430F5529LP	1-35
MSP-BNDL-FR5969LCD	1-36
<i>Lab 1 – Out-of-Box User Experience Lab</i>	1-37

Administrative Topics

A few important details, if you're taking the class live. If not, we hope you already know where your own bathroom is located.

Administrative Topics

- ◆ **Tools Install & Labs**
- ◆ **Start & End Times**
- ◆ **Lunch**
- ◆ **Course Materials**
- ◆ **Name Tags**
- ◆ **Restrooms**
- ◆ **Mobile Communications**
- ◆ **Questions & Dialogue (the key to learning)**



Workshop Agenda

Here's the outline of chapters in this workshop.

Workshop Agenda

1. Introduction to MSP430
2. Code Composer Studio
3. GPIO and MSP430ware
4. Clocking and System Init
5. Interrupts
6. Timers (A & B)
7. Non-Volatile Memory: Flash & FRAM
8. USB
9. Using Energia (Arduino)

MSP430 Workshop (v3.1)

Chapter 1: *“Intro”* Provides a quick introduction to TI, TI’s Embedded Processors, as well as the MSP430 Family of devices.

Chapter 2: *“CCS”* introduces TI’s development ecosystem. This includes:

- Code Composer Studio (CCSV5)
- Target software, such as MSP430ware and TI-RTOS
- TI’s support infrastructure, including the embedded processors [wiki](#) and Engineer-to-Engineer ([e2e](#)) forums.

Chapter 3: *“GPIO”* This is our introduction to programming with MSP430ware; specifically, the DriverLib (i.e. driver library) part of MSP430ware. We start out by using it to program GPIO to blink an LED (often called the “embedded systems version of ‘Hello World’”). The second part of the lab reads a Launchpad pushbutton.

Chapter 4: *“Clocks”* This chapter starts at reset – in fact, all three resets found on the MSP430. We then progress to examining the rich and robust clocking options provided in the MSP430. This is followed by the power management features found on many of the ‘430 devices. The chapter finishes up by reviewing the other required system initialization tasks ... such as configuring (or turning off) the watchdog timer peripheral.

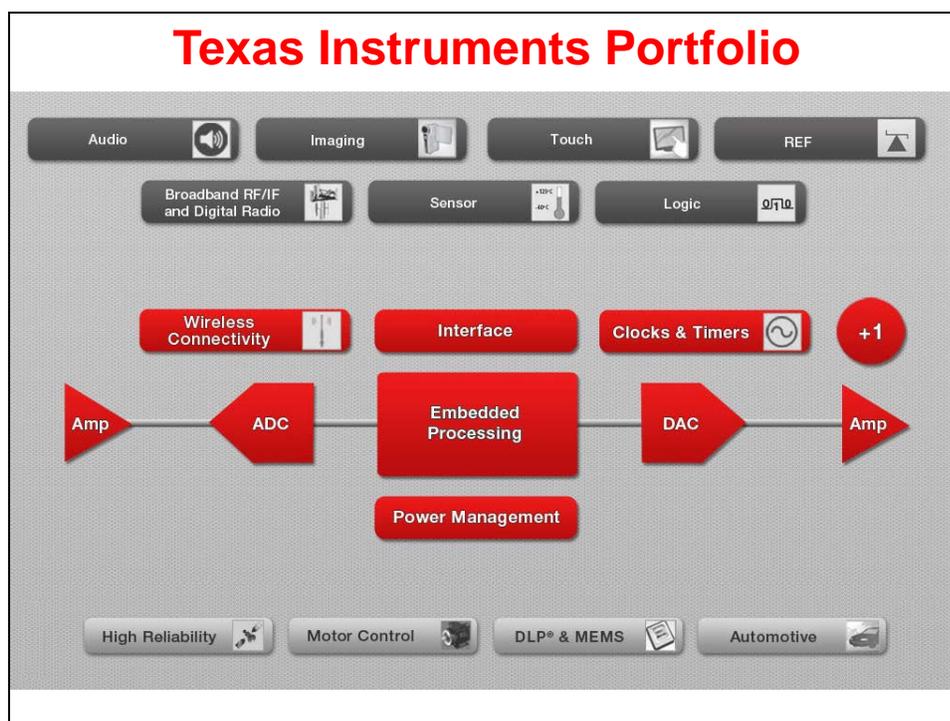
- Chapter 5:** *Interrupts* ... do you use interrupts? Yep, they're one of the most fundamental parts of embedded system designs. This is especially true when your processor is known as the king of low-power. We examine the sources, how to enable, and what to do in response to ... interrupts.
- Chapter 6:** *Timers* are often thought of as the lifeblood of a microcontroller program. We use them to generate periodic events, as one-shot delays, or just to wake ourselves up every once in a while to read a sensor value. This chapter focuses on Timer_A – the primary timer module found in the MSP430.
- Chapter 7:** *USB* – Universal Serial Bus is an ideal way to communicate with host computers. This is especially true as most PC's have done away with dedicated serial and parallel ports. We attempt to explain how USB works as well as how to build an application around it. What you'll find is that the MSP430 team has done an excellent job of making USB simple.
- Chapter 8:** *Energia* is also known by the name "Arduino". *Energia* was the name given to Arduino as it was ported to the TI MCU's by the open-source community. Look up the definition of *Energia* – and let it 'propel' your application right off the Launchpad.

TI Products

TI's Entire Portfolio

It's very difficult to summarize the entire breadth of TI's semiconductor products – it's so far reaching. But, maybe that's not to be unexpected from the company who invented the integrated circuit.

Whether you are looking for embedded processors (the heart of following diagram) or all the components that sit alongside – such as power management, standard logic, op amps, data conversion, display drivers, or ... so much more – you'll find them at TI.



Before taking a closer look at embedded processors, we'll glance at one of the hottest growing product categories ... TI's extensive portfolio of wireless connectivity.

Wireless Products

Wireless devices let us talk through the air. *Look ma, no wires.*

What protocol or frequency resonates with you and your end-customers? Whether it's: near-field communications (NFC); radio-frequency ID (RFID); the long range, low-power sub 1-GHz; ZigBee®; 6LoPan; Bluetooth® or Bluetooth Low Energy® (BLE); ANT®; or just good old Wi-Fi – TI's got you covered.

The industry's broadest wireless connectivity portfolio

Supported Standards					
134.2K-13.56MHz	Sub 1GHz	2.4GHz to 5GHz			
RFID, NFC ISO14443A/B ISO15693	SimpliciTI 6LoWPAN W-MBus	SimpliciTI PurePath Wireless	ZigBee® 6LoWPAN RF4CE	Bluetooth® BLE ANT	Wi-Fi
Example Applications					
					
Product Lineup					
TMS37157 TRF796x TRF7970	CC1110 CC1190 CC11xL CC430 CC112X CC120X CC1180	CC2500 CC2543/4/5 CC2590/91 CC8520/21 CC2530/31	CC2530 CC2530ZNP CC2531 CC2533 CC2520	CC2560/4 CC2540/1 CC2570/1	WL1271/3 WL 18xx CC3000
				Red = SimpleLink family	

Many low-end, low-cost MCU designers have longed for a way to connect wirelessly to the rest of the world. TI's wireless devices and modules make this possible. No longer do you need a gigahertz processor to run the various networking stacks required to talk to the outside world – the TI SimpleLink line handles this for you ... meaning that any processor that can communicate via a serial port can be networked. Drop a CC3000 module into your design and you've enabled it to join the *Internet of Things* revolution.

Check out TI's inexpensive, low-power and innovative wireless lineup!

TI's Embedded Processors

Whether you are looking for the MSP430, which is the lowest power microcontroller (MCU) in the world today ... or the some of the highest performance single-chip microprocessors (MPU) ever designed (check out Multicore) ... or something in between ... TI has your needs covered.

Microcontrollers (MCU)				Application (MPU)		
MSP430	C2000	Tiva C	Hercules	Sitara	DSP	Multicore
16-bit Ultra Low Power & Cost	32-bit Real-time	32-bit All-around MCU	32-bit Safety	32-bit Linux Android	16/32-bit All-around DSP	32-bit Massive Performance
MSP430 ULP RISC MCU	• Real-time C28x MCU • ARM M3+C28	ARM Cortex-M4F	ARM Cortex-M3 Cortex-R4	ARM Cortex-A8 Cortex-A9	DSP C5000 C6000	• C66 + C66 • A15 + C66 • A8 + C64 • ARM9 + C674
• Low Pwr Mode = 0.1 µA = 0.5 µA (RTC) • Analog I/F • USB and RF	• Motor Control • Digital Power • Precision Timers/PWM	• 32-bit Float • Nested Vector IntCtrl (NVIC) • Ethernet (MAC+PHY)	• Lock step Dual-core R4 • ECC Memory • SIL3 Certified	• \$5 Linux CPU • 3D Graphics • PRU-ICSS industrial subsys	• C5000 Low Power DSP • 32-bit fix/float C6000 DSP	• Fix or Float • Up to 12 cores 4 A15 + 8 C66x • DSP MMAC's: 352,000
TI-RTOS	TI-RTOS (k)	TI-RTOS	3rd Party (only)	Linux, Android, TI-RTOS Kernel	C5x: DSP/BIOS C6x: TI-RTOS (k)	Linux TI-RTOS (k)
Flash: 512K FRAM: 64K	512K Flash	512K Flash	256K to 3M Flash	L1: 32K x 2 L2: 256K	L1: 32K x 2 L2: 256K	L1: 32K x 2 L2: 1M + 4M
25 MHz	300 MHz	80 MHz	220 MHz	1.35 GHz	800 MHz	1.4 GHz
\$0.25 to \$9.00	\$1.85 to \$20.00	\$1.00 to \$8.00	\$5.00 to \$30.00	\$5.00 to \$25.00	\$2.00 to \$25.00	\$30.00 to \$225.00

To start with, look at the **Blue/Red** row about 1/3 the way down the slide. The columns with **Red** signify devices utilizing ARM processor cores. If you didn't think TI embraces the ARM lineup of processors, think again. TI is one of the leaders in ARM development, manufacturing and sales.

Jumping to the 3rd column, the **Tiva C** (Tiva Connected) processors are probably the best all-around MCU's in use today. The 32-bit floating point ARM Cortex-M4F core can be connected to the real-world by a dizzying array of peripherals. They provide a near-perfect balance of performance, power, and connectivity.

On the other hand, if you're building safety critical applications, the **Hercules** family of processors is what you should key in on. Whether your customers appreciate the safety of dual-core, lockstep processing or the SIL3 certification, these processors are a unique mix of ARM Cortex-R4 performance combined with TI's vast SafeTI[®] knowledge.

Moving up to what ARM calls their 'Application' series of processors, TI set the processing world on fire (figuratively) when they introduced the **Sitara** AM335x. That you could get a \$5 processor which runs Linux, Android or other high-level operating systems was jaw-dropping. We probably didn't make some PC manufacturers happy – we've seen many of our customers replace bulky, power-hungry embedded PC's with small, low-power BeagleBoard-like replacements. This device was the inflection point – it's started a new direction for embedding high-level host systems.

And if you're looking for the high-end **ARM Cortex-A15**, we've got that too. Take your pick: do you want one ... or up to 4 A15 cores on a single device? And these multi-core devices also pack the number crunching of TI's C66x line of DSP cores. When high-end performance processing is critical to your systems, look no further than TI Multicore.

But as one student asked, “If ARM is so great, why do you make other types of processors?”

While ARM is probably thought of today as the best all-around set of processor cores, there are areas where it can be improved upon.

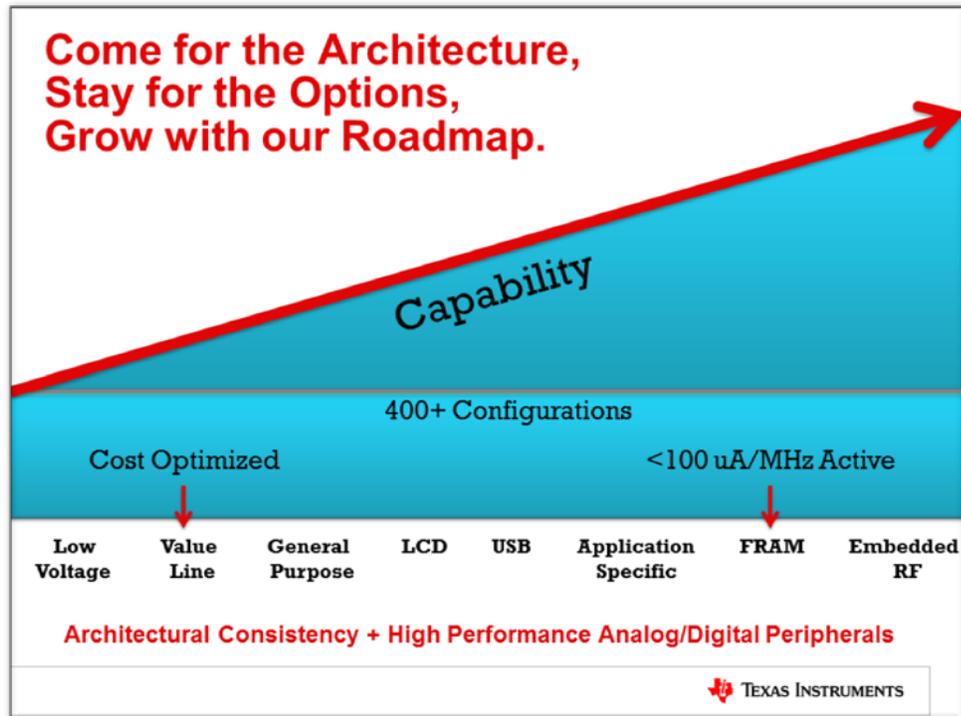
Driving to the *lowest-power dissipation* is one of those areas. In the end, the venerable **MSP430** is not to be outdone on the low end. As the MSP430 teams says, Ultra Low-Power (ULP) is “in our DNA”. You know you’re doing something right when the 10-year shelf-life of the battery ends up self-dissipating before you run it dry with your MSP430 design. It’s just hard to beat an MCU designed from the ground up as a low-power CPU. That said, it’s also hard to beat the MSP430’s simple, inexpensive, high-performance RISC engine.

The **C2000** family has set the standard for control applications. Whether it’s digital motor control, power control or one of the many other control-oriented MCU applications, this CPU really crunches the data. You might also see a little **Red** in this column. That’s to indicate that even a good DSP-based microcontroller can use a little bit of ARM to get a leg-up in the industry. We’ve coupled an ARM Cortex-M3 along with the C28x core to make a stellar processing duo. Use the ARM to run your networking and USB stacks – all the while the C28x core is taking care of your system’s real-time processing needs. Sure, you could buy two chips to implement your systems (we’ll happily sell you a C28x along with Tiva C), but these devices integrate them both into a singular device.

Finally, TI is known by many as the center of **DSP** excellence. While these CPUs often get lost in all the hoopla surrounding ARM today, when it comes to real-time systems, a good DSP is hard to beat. Whether you’re implementing a low-power system (look to **C5000** DSP’s) or need the number crunching performance of the **C6000**, these devices still cannot be bested in the world of hard real-time, low-latency, highly deterministic applications. As mentioned earlier, the highest performing C6000 DSP cores have been combined into the awesome performance of Multicore. You can get up to 8 CPU’s on a single device; make them all C66x DSPs – or match four C66x CPU’s up with four of ARM’s stunning Cortex-A15’s for a performance knock-out punch.

MSP430 Family

As stated, low-power is 'in our DNA'. Though, it's not all the MSP430 is known for.



One vector of new products has continued to integrate a wide range of low-power peripherals into the MSP430 platform. Look for the products in the MSP430 F5xx, F6xx and FR5xxx families. Also, the CC430 family adds the unique touch of on-chip integrated RF radios.

A second vector of development is driving the cost out of your designs. Look no further than the Gxxx Value Line series of devices. The goal is to provide highly integrated, low-power, 16-bit performance in an inexpensive device – giving you a new choice versus those old 8-bit micros.

And finally, the new MSP430 Wolverine series of devices is once again setting new standards for low-power processing. Sure, we're only topping our own products, but who else is better suited to enable your lowest power processing needs? Utilizing the FRAM memory technology, the FR5xxx Wolverine devices combine the lowest power dissipation with a rich integration of peripherals.

MSP430 Families

Series	Low Voltage	Value Line	1 Series	2 Series	4 Series	5 Series	6 Series	FRAM Series	RF SoC
Part Number	L092	G2xxx	F1xx	F2xx	F4xx	F5xx	F6xx	FR5xxx	CC430
Max speed (MHz)	4	16	8	16	16	25	25	24	20
NVM (max KB)	0	56	60	120	120	512	512	64	32
SRAM (max KB)	2	4	10	8	8	66	66	2	4
GPIO	11	4-32	14-48	10-48	14-80	29-87	72-90	17-40	30-44
Comparator	•	•	•	•	•	•	•	•	•
Timer	•	•	•	•	•	•	•	•	•
ADC	•	•	•	•	•	•	•	•	•
DAC	•	•	•	•	•	•	•	•	•
UART	•	•	•	•	•	•	•	•	•
I ² C	•	•	•	•	•	•	•	•	•
SPI	•	•	•	•	•	•	•	•	•
Capacitive touch	•	•	•	•	•	•	•	•	•
Multiplier	•	•	•	•	•	•	•	•	•
DMA	•	•	•	•	•	•	•	•	•
Op amps	•	•	•	•	•	•	•	•	•
LCD	•	•	•	•	•	•	•	•	•
RTC	•	•	•	•	•	•	•	•	•
PMM	•	•	•	•	•	•	•	•	•
1.8-V I/O	•	•	•	•	•	•	•	•	•
CRC	•	•	•	•	•	•	•	•	•
High-resolution timer	•	•	•	•	•	•	•	•	•
USB	•	•	•	•	•	•	•	•	•
Hardware encrypt (AES)	•	•	•	•	•	•	•	•	•
FRAM	•	•	•	•	•	•	•	•	•
RF	•	•	•	•	•	•	•	•	•

Higher Integration

Low Cost
Ultra Low Power

Here's a quick overview of the device we'll be using in this workshop. The MSP430F5529 is part of the F5xx series of devices and is found on the new 'F5529 USB Launchpad.

F5xx Key Features

Ultra-Low Power

- ◆ 160 μ A/MIPS
- ◆ 2.5 μ A standby mode
- ◆ Integrated LDO, BOR, WDT+, RTC
- ◆ 12 MHz @ 1.8V
- ◆ Wake up from standby in <5 μ s

Increased Performance

- ◆ Up to 25 MHz
- ◆ 1.8V ISP Flash erase and write
- ◆ Fail-safe, flexible clocking system
- ◆ User-defined Bootstrap Loader
- ◆ Up to 1MB linear memory addressing

Innovative Features

- ◆ Multi-channel DMA supports data movement in standby mode
- ◆ Industry leading code density
- ◆ More design options including USB, RF, encryption, LCD interface

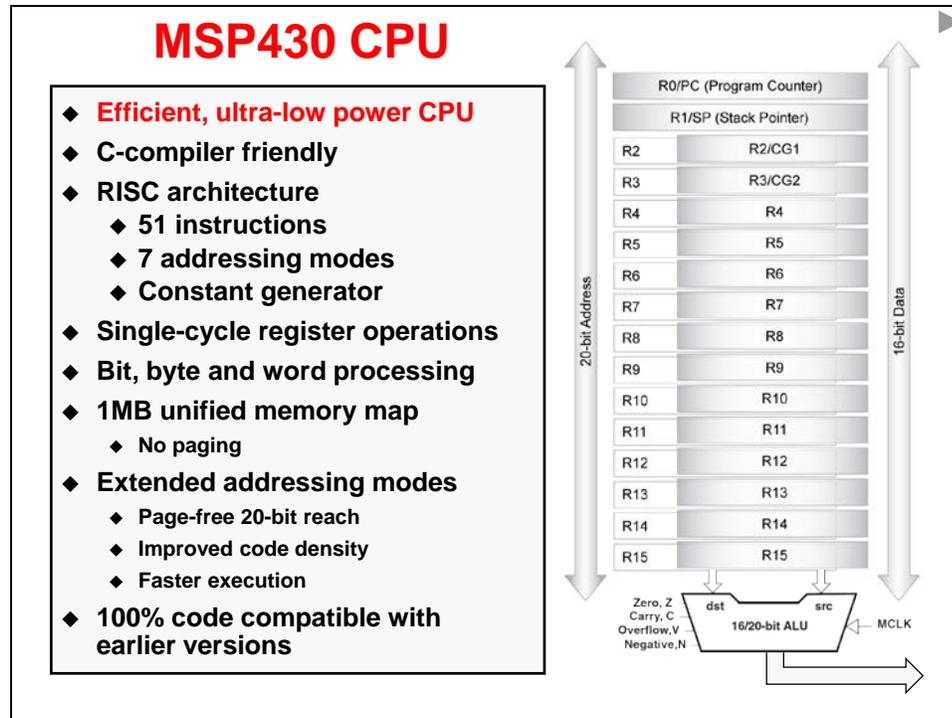
MSP430F5xx Block Diagram

Computation	Timing and Control	Signal Chain	Communication	I/O & Display
Hardware 32x32 Multiplier	General Purpose Timers Capture/Compare PWM Outputs	Comparators	Universal Serial Communication Interfaces (SPI, UART, I2C)	General Purpose I/O, Pull-Up/Down, High Drive
CRC	Basic Timer + RTC	ADC	USB 2.0 (Full Speed) Engine + PHY	Segmented LCD, Static, Muxed
AEC		DAC	RF Transceivers	
		Operational Amplifiers		

This is one of TI's line-up of MSP430 devices featuring highly integrated set of peripherals. We will be exploring quite a bit more about this device as we progress through the workshop.

MSP430 CPU

As stated earlier, the MSP430 is an efficient, simple 16-bit low power CPU. Its orthogonal architecture and register set make it C-compiler friendly.



The original MSP430 devices were true 16-bit processors. While 16-bits are quite ideal from a data perspective, it's limited from an addressing perspective. With 16-bit addresses, you're limited to only 64K of memory – and that really isn't acceptable in many of today's applications.

As early as the second generation of MSP430 devices, the CPU was expanded to provide full 20-bits of addressing space – which provides 1M of address reach. The new CPU cores that support these enhancements were called CPUX (for eXtended addressing). Thankfully, the extended versions of the CPU maintained backward compatibility with the earlier devices.

In this course, we don't dwell on these CPU features for two reasons:

- This change was made long enough to go that all the processors engineers choose today include the enhanced CPU.
- With the prevalence of C coded applications in world of MSP 430, and embedded processing in general, these variations fall below our radar. The compiler, handily, manages low-level details such as this.

There are many touches to the MSP430 CPU which make it idea for low-power and microcontroller applications, such as the ability to manage bytes, as well as 16-bit words.

Bytes, Words And CPU Registers

16-bit addition		Code/Cycles
5405	add.w R4,R5	; 1/1
529202000202	add.w &0200,&0202	; 3/6
8-bit addition		
5445	add.b R4,R5	; 1/1
52D202000202	add.b &0200,&0202	; 3/6

- ◆ Use CPU registers for calculations and dedicated variables
- ◆ Same code size for word or byte
- ◆ Use word operations when possible



Note: If you see a 'gray' slide like the one above and below were placed into the workbook, but has been hidden in the slide set, so the instructor may not present it during class.

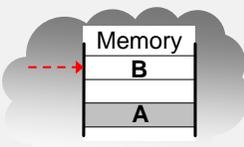
Seven Addressing Modes

Mode	Example	Notes
Register	mov.w R10,R11	Single cycle
Indexed	mov.w 2(R5),6(R6)	Table processing
Symbolic	mov.w EDE,TONI	Easy to read code, PC relative
Absolute	mov.w &EDE,&TONI	Directly access any memory
Indirect Register	mov.w @R10,0(R11)	Access memory with pointers
Indirect Autoincrement	mov.w @R10+,0(R11)	Table processing
Immediate	mov.w #45h,&TONI	Unrestricted constant values



A rich set of addressing modes lets the compiler create efficient, small-footprint programs. And, features like 'atomic' addressing are critical for real-world embedded processing.

Atomic Addressing

$B=B+A$


<pre> ; Pure RISC push R5 ld R5,A add R5,B st B,R5 pop R5 </pre>	<pre> ; MSP430 add A,B </pre>
---	---------------------------------

- ◆ Non-interruptible memory-to-memory operations
- ◆ Useable with complete instruction set

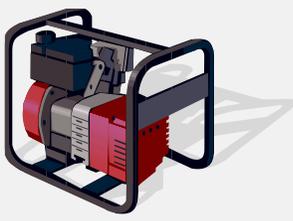


The little bit of genius that is the Constant Generator minimizes code size and runtime cycle count. These ideas save you money while helping to reduce power dissipation.

Constant Generator

<u>4</u> 314	mov.w	#0002h,R4	; With CG
40 <u>3</u> 41234	mov.w	#1234h,R4	; Without CG

- ◆ Immediate values -1,0,1,2,4,8 generated in hardware
- ◆ Reduces code size and cycles
- ◆ Completely automatic



A low number of instructions are at the heart of Reduced Instruction Set Computers (RISC). RISC lowers complexity, cost and power ... while, surprisingly, maintaining performance.

51 Total Assembly Instructions

Format I Src, Dest	Format II Single Operand	Format III +/- 9bit Offset	Support
add(.b)	br	jmp	clrc
addc(.b)	call	jc	setc
and(.b)	swpb	jnc	clrz
bic(.b)	sxt	jeq	setz
bis(.b)	push(.b)	jne	clrn
bit(.b)	pop(.b)	jge	setn
cmp(.b)	rra(.b)	jl	dint
dadd(.b)	rrc(.b)	jn	eint
mov(.b)	inv(.b)		nop
sub(.b)	inc(.b)		ret
subc(.b)	incd(.b)		reti
xor(.b)	dec(.b)		
	decd(.b)		
	adc(.b)		
	sbc(.b)		
	clr(.b)		
	dadc(.b)		
	rla(.b)		
	rlc(.b)		
	tst(.b)		

Bold type denotes emulated instructions

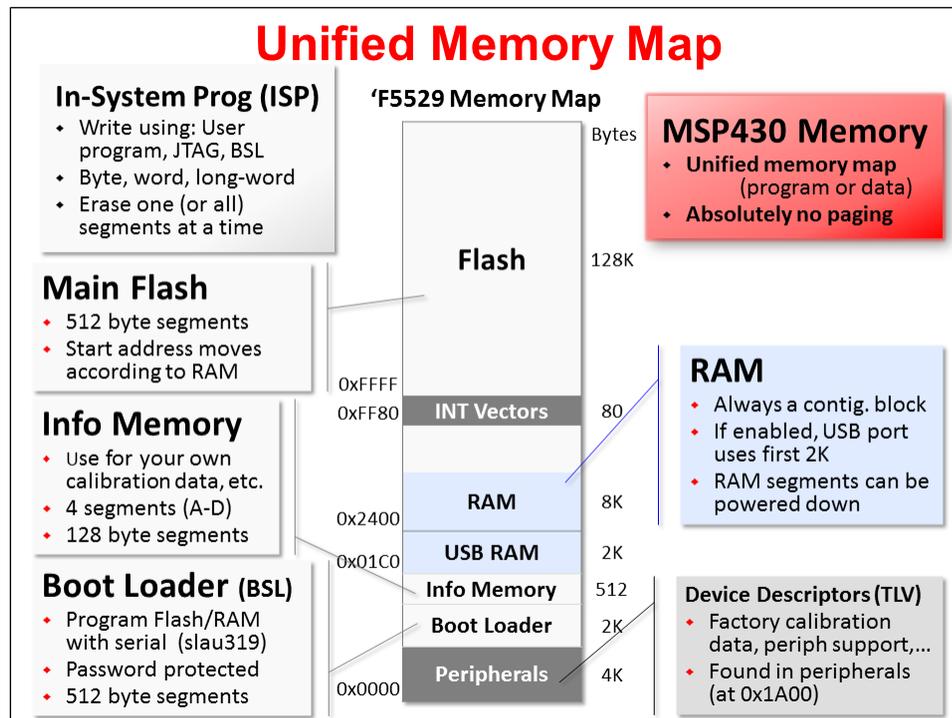
MSP430 Memory

Memory Map

We present the MSP430F5529 memory map as an example of what you find on most MSP430's. It's certainly what we'll see as we work through the lab exercises in this workshop.

A couple of important – and beneficial – points about MSP430's memory map:

- The MSP430 defines a **unified** memory map. This means that, technically speaking, data and program code can be located anywhere in the available memory space. (*This doesn't mean it's practical to locate global variables in flash memory, but the architecture does not prevent you from doing so.*)
- The MSP430, as stated earlier (see page 1-13), is implemented using 20-bit addressing; therefore, the MSP430 can directly address the full 1M memory map without resorting to paging schemes. (If you have ever had to deal with paging, we expect you might be cheering at this point.)



Flash

Like most MCU's nowadays, the processor is dominated by non-volatile memory. In this case, Flash technology provides us with the means to store information into the device – which retains its contents, even when power is removed. (As we'll see next, some of the latest MSP430 devices use FRAM technology rather than Flash.)

The flash memory is In-System Programmable (ISP), which means we can reprogram the memory without taking the chip off of our boards or using difficult bed-of-nails methods. In fact, you can program the flash using:

- An IDE, such as CCS or IAR. These debugging tools utilize the 4-wire JTAG or 2-wire SPI-biwire emulation connections.
- The MSP430 Boot-Strap Loader supports a variety of connections and options. For example, you can use the serial (or USB) interfaces to reprogram your devices. These interfaces are popular on many manufacturing work flows.
- Finally, you can reprogram all – or part – of the flash memory via your own program running on the device itself. Check out the MSP430ware FLASH DriverLib functions.

On the 'F5529, as with most MSP430 devices, the Flash actually consists of 3 regions.

Main consists of the bulk of flash memory. This is where our programs are written to when using the default project settings. Main flash consists of one contiguous memory; although, the Interrupt Vectors are located inside of it at 0xFF80. If your device has more than 64K of flash, then some will exist above and below the vectors – as shown in the diagram for the 'F5529 (which has 128K of flash).

Info Memory can be thought of as user data flash. Again, there are not any limitations on what you store here, but these four segments are commonly used to hold calibration data or other non-program items you want to store in non-volatile memory.

Boot Loader (BSL) holds the aforementioned boot loader code. This code, in turn, is used to load new programs into Main flash. Please be aware that the BSL is handled differently amongst the various generations of MSP430. In some cases, as with the 'F5529, it is stored in its own region of flash memory. On other devices, it may be hard-coded into the device.

RAM

RAM (Static Random Access Memory – SRAM) is found on every MSP430 device. Like flash, though, the amount of RAM varies from device to device; and the amount of RAM memory is often directly proportional to the cost of the device.

RAM is where most of the data is stored: everything from global variables, to stacks and heaps. It is often thought of as the 'working' memory on the device. Even so, due to the 'unified' nature of the MSP430 architecture, you can also move program code into RAM and run from this space.

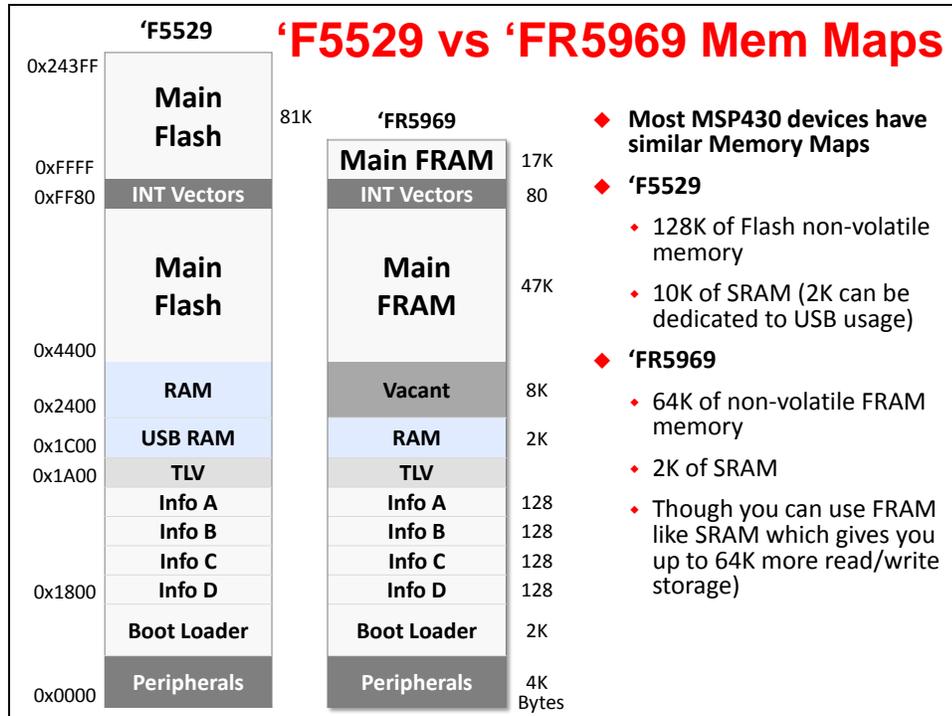
The 'F5529 has one aspect that is common among MSP430 devices which include the USB peripheral. These devices have an extra 2KB of RAM; this RAM is dedicated to the USB peripheral when it is in use, but available to your programs when the USB port is not being used. Please refer to the USB Developers Package documentation to learn more about how the USB protocol stack uses this RAM.

TLV

Although not 'memory', the **Device Descriptors (TVL)** does appear within the memory map. This segment contains a tag-length-value (TLV) data structure that comprises a hierarchical description (or on older devices, flat file description) of information such as: the device ID, die revisions, firmware revisions, and other manufacturer and tool related information. Additionally, these descriptors may contain information about the available peripherals, their subtypes and addresses. This info may prove useful if building adaptive hardware drivers for operating systems. (Note that some of the Value Line devices may not contain all of this information; and, their factory supplied calibration data may reside in Info Memory A.)

Comparing Memory Maps

Most MSP430 devices have fairly similar Memory Maps; the primary differences ends up coming down to how much memory a specific device contains. Please check the datasheet for the specific details on each device.



The two devices shown here have one other major differentiating factor, the 'F5529 uses *Flash* technology while the 'FR5969 uses *FRAM* technology to store its non-volatile information.

We briefly compare these two technologies in the next section, though you may want to refer to the *Non-Volatile Memory (Flash & FRAM)* chapter for more details.

FRAM

Some of the latest MSP430 devices from TI now use FRAM in place of Flash for their non-volatile memory storage. For example, you will find the Wolverine (FR58xx, FR59xx) devices utilize this new technology.

FRAM: The Future of MCU Memory

- ◆ **Non-volatile, Reliable Storage**
 - ◆ Over 100 Trillion write/read cycles
 - ◆ Write Guarantee in case of power loss
- ◆ **Fast write times like SRAM**
 - ◆ ~50ns per byte or word
 - ◆ 1,000x faster than Flash/EEPROM
- ◆ **Low Power**
 - ◆ Only 1.5v to write & erase
 - ◆ >10-14v for Flash/EEPROM
- ◆ **Universal Memory**

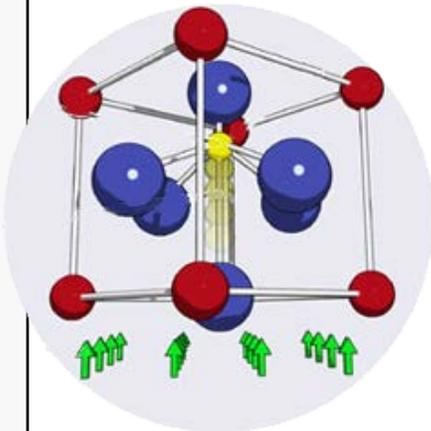


Photo: Ramtron Corporation

Actually, FRAM is not a brand new technology. It has been available in stand-alone memory chips for nearly a decade. It is quite new, though, to find it used within micros.

In brief, the MSP430 FRAM provides some exciting new features in our MCUs:

- FRAM memory is a nonvolatile memory that reads and writes like standard SRAM
- It supports Byte or word write access
- A nearly limitless re-write capability – ‘we haven’t worn it out yet’
- Very fast write cycles – much faster than Flash or EEPROM
- Very low power – unlike Flash memory, it only takes 1.5V to write and erase FRAM (really ideal for low-power data logging applications)
- Error Correction Code with bit error correction, extended bit error detection and flag indicators
- Power control for disabling FRAM if it is not used – and due to non-volatile nature, it naturally does not lose its contents in the process of powering down

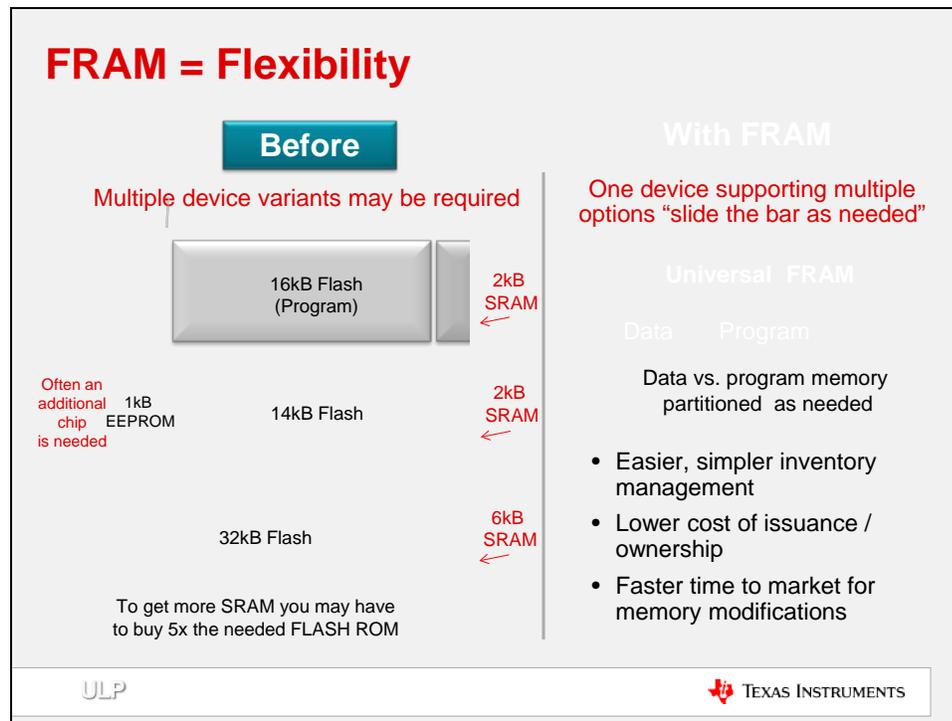
As stated above, FRAM can be read and written in a similar fashion to SRAM and needs no special requirements. This provides a big value in letting you choose how to use your memory; in other words, if your system needs “a little bit more RAM”, this can be accomplished by locating your data in FRAM.

The downside, of course, is that your program could be just as easily overwritten in the same fashion. (We shouldn't have code that writes to program addresses – but accidents occur.) To this end, the FRAM based devices provide a memory protection unit (MPU) that lets you create 1 to 3 segments of FRAM. Often, these segments are set for: Execute only, Read only, and Read/Write.

The other two caveats to FRAM are that reads are a bit slower than Flash and their density is not as great as we can build using flash technology. On the other hand, the benefits are an outstanding fit for many MSP430 types of applications.

FRAM MCU Delivers Max Benefits				
	FRAM	SRAM	EEPROM	Flash
Non-volatile Retains data without power	Yes	No	Yes	Yes
Write speeds	10 ms	<10 ms	2 secs	1 sec
Average active Power (μ A/MHz)	110	<60	50mA+	230
Write endurance	100 Trillion +	Unlimited	100,000	10,000
Dynamic Bit-wise programmable	Yes	Yes	No	No
Unified memory Flexible code/data partitioning	Yes	No	No	No

This graphic speaks to the earlier comment about the trade-offs between Flash and RAM. We have seen users who are forced into purchasing a larger, more expensive MCU just to get a little bit more RAM. The flexibility of FRAM allows your programs to use the non-volatile storage for things like variables and buffers. This flexibility often ends up lowering your overall system costs.



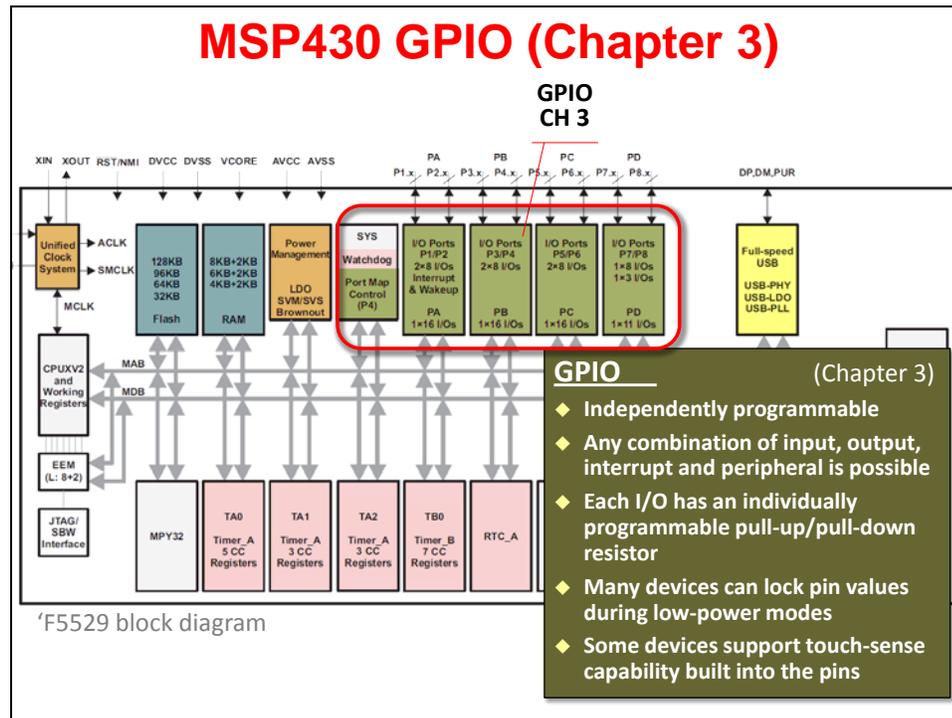
MSP430 Peripherals

This section provides a high-level overview of the various categories of MSP430 peripherals.

GPIO

MSP430 devices contain many I/O ports. The largest limitation is usually the package selection – a lesser pin-count package means less General Purpose bit I/O.

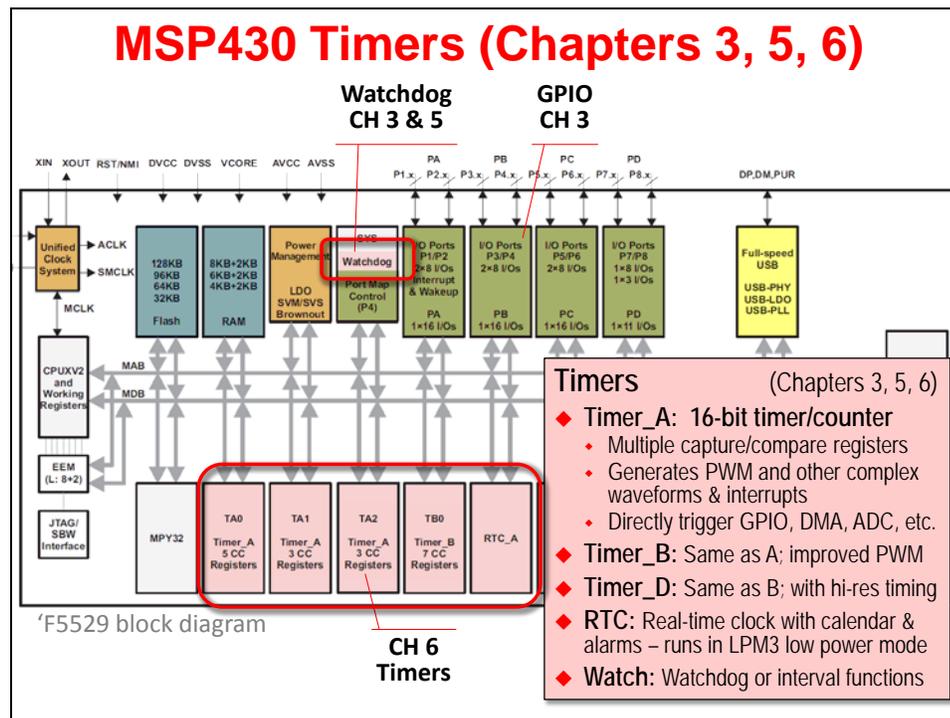
Like most current day microcontrollers, the pins on our devices are heavily multiplexed. That is, you often have one of several choices of signals that can be output to a given pin. The MSP430 makes each signal independently programmable, which affords maximum flexibility.



Other handy GPIO features include:

- I/O ports 1 and 2 can generate interrupts to the CPU. (Some devices support interrupts on additional I/O ports.)
- Pull-up and Pull-down resistors are available as part of the I/O port, simplifying your board design.
- Many devices can lock the state of the pins when going into the lowest power modes, which again saves the effort, power, and cost of adding external transceivers to accomplish this purpose.
- Finally, many I/O ports include 'touch' circuitry. This additional circuitry makes it easy to implement capacitive touch based interfaces in your systems – all without having to add extra hardware.

Timers



As stated earlier, timers are often thought of as the heartbeat of an embedded system. The MSP430 contains a number of different timers that can assist you with different system needs.

Timer_A (covered in detail in Chapter 6) is the original timer found across all MSP430 generations. And there is a reason for that, it is quite powerful, as well as flexible.

These 16-bit timers contain anywhere from 2 to 7 capture/compare registers (CCR). Each CCR can capture a time value when triggered (capture mode). Alternatively, each CCR could be used to generate an interrupt or signal (internal or external via a pin) when the timer's counter (TAR) matches the value in the CCR (compare mode). Oh, and each CCR is independently programmable – thus some could be used for capture while others for compare.

Using the CCR feature, it is easy to create a host of complex waveforms – for example, they could be used to generate PWM outputs. (Something we'll explore in Lab 6.)

Timer_B is nearly similar to **Timer_A**. It provides the ability to use the internal counter in 8/10/12 or 16-bit modes. This affords it a bit more flexibility. Additionally, double-buffered CCR registers, as well as the ability to put the timer outputs into high-impedance, provide a couple of additional advantages when driving H-bridges and such.

Timer_D takes **Timer_B** and adds a higher resolution capability. (BTW, we're not sure what happened to **Timer_C**...)

RTC (real-time clock) peripherals not only provide a time base, but their calendar and alarm modes make them ideal for clock/calendar types of activities. More importantly, they have been designed to run with extremely low power. This means they can provide a heartbeat while the rest of your system is asleep.

Watchdog timers provide two different functions. In their namesake mode, they act as failsafe's for the system. If your code does not reset them before their counter reaches the end, they reset the system. This functionality is ALWAYS enabled at boot. You can also choose to use them as an interval timer.

Clocking and Power Management

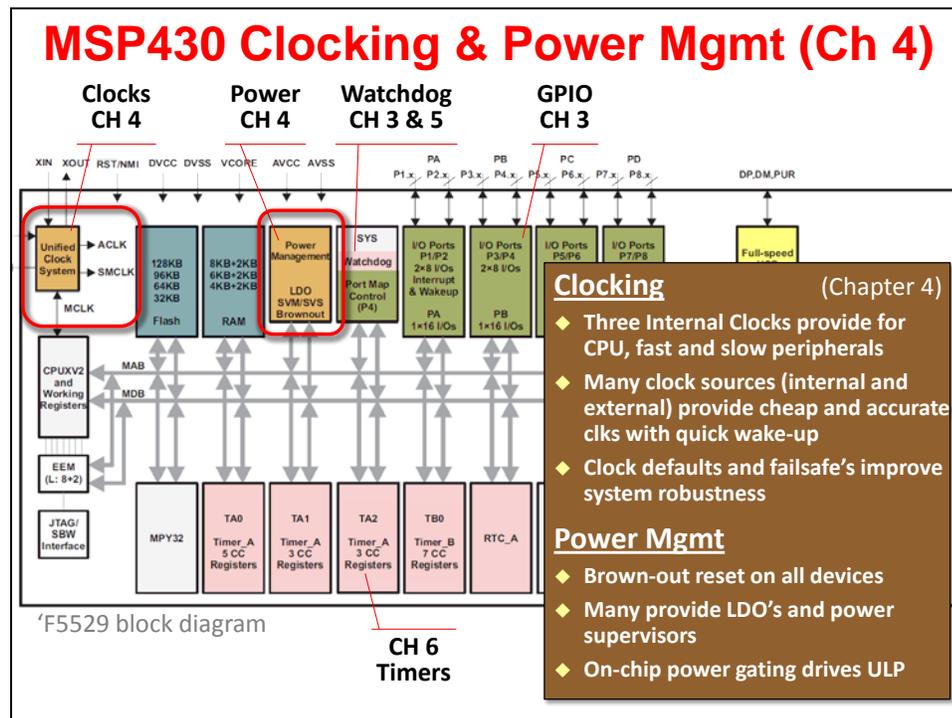
MSP430 Clocks (Chapter 4)

The MSP430 devices provide a rich, robust set of clocking options.

Rich in that they provide a great number of on- and off-chip clock sources. Further, there are three internal clocks routed to the CPU and various peripherals. Why three? Simply, there's a clock for the CPU and two clocks for the peripherals - one fast and the other slow - with goal of providing the user a balance of performance and low power. Of course, some of the devices provide more clock choices than others.

Robust clocking in that there are defaults and failsafe's for all of the various clocks. These failsafe clocks choices can be particularly important for some applications. Imagine a crystal oscillator being forcibly removed from the board - or maybe just broken - when your end-product is accidentally damaged in use. It's nice to know there are internal alternatives that let your product continue working in a well-documented state.

Please turn to the Clocking chapter for further information.



Power Management

Power is one of those features that every system needs but doesn't often get highlighted. All of the MSP430 devices provide some level of Power Management. On the most cost-sensitive, it might only be a Brown-Out Reset (BOR) peripheral - which makes sure there is enough power going to the device to assure proper, stable operation. The other notable point is that BOR was designed with extreme sensitivity to low-power system needs.

On other devices you'll find BOR plus an increasing set of power management peripherals. For example, the 'F5529 device adds an LDO (low dropout voltage regulator) which derives a steady CPU voltage from that applied to the device. (Normally, voltage regulation is handled by an extra

device in your system.) The 'F5529 also contains a sophisticated power supervisor to warn (i.e. interrupt) your system when the power is getting close to out-of-spec.

Power gating is another feature found on most of the MSP430 devices. The basic idea is that we want to power-down anything that is not needed.

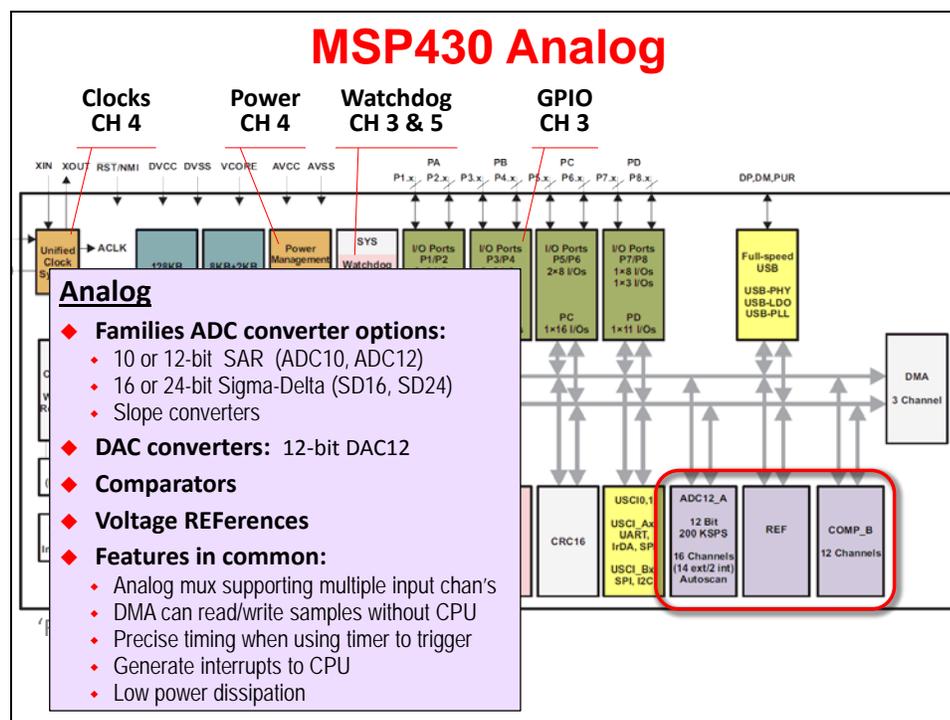
Analog

Bringing high-quality analog components on-chip was a big selling point of the original MSP430 devices - and still is today. Besides providing high-quality analog, they've done it with a low-power footprint, too.

MSP430 analog peripherals cover a wide range of needs. At one end, you'll find most every device contains one or more analog comparators. These signal the processor when an analog input crosses a boundary. (Comparators are often used to build a "poor mans" analog to digital converter.)

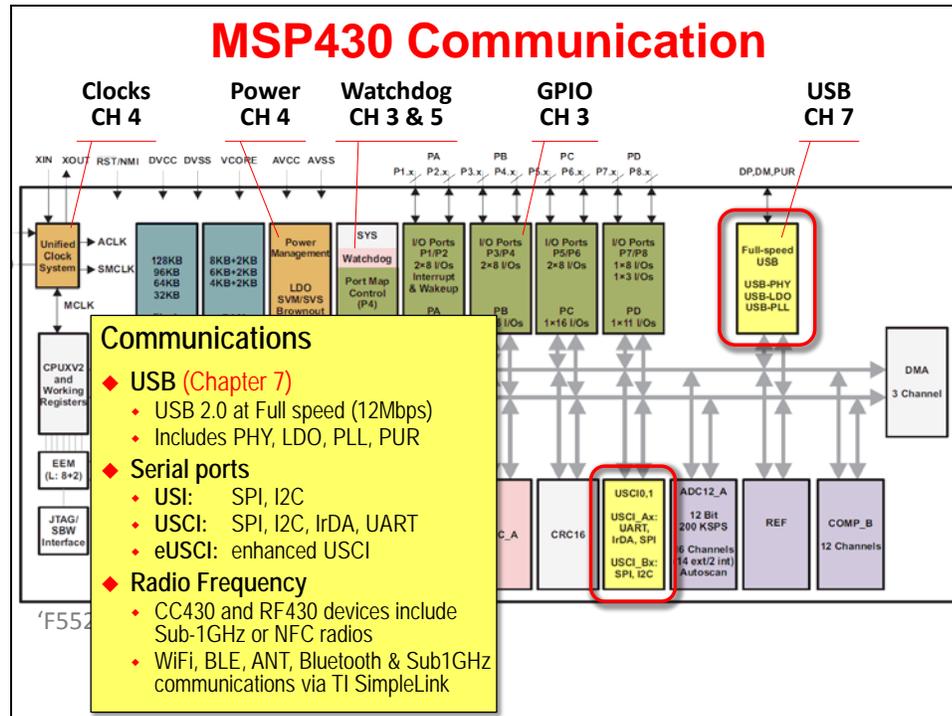
In many systems, though, you will want an actual ADC (analog to digital) converter. The MSP430 family provides a wide variety of options. In fact, some designers select their specific MSP430 device based upon which type of converter they want to use.

Almost regardless of the type of analog component, they have a few key features in common. The ability to generate interrupts is fundamental. Also critical are the ability to trigger conversions based on timers; and couple that with using DMA's to transfer the results to memory sans CPU.



Communications (Serial ports, USB, Radio)

We specifically chose the name "Communications" for this category, rather than the more common "Serial Communications". It's true that most of the communications ports utilize serial connections; this is due to the lower cost and power of using fewer pins. But, in the end, we didn't want to overlook the growing support for wireless communications.



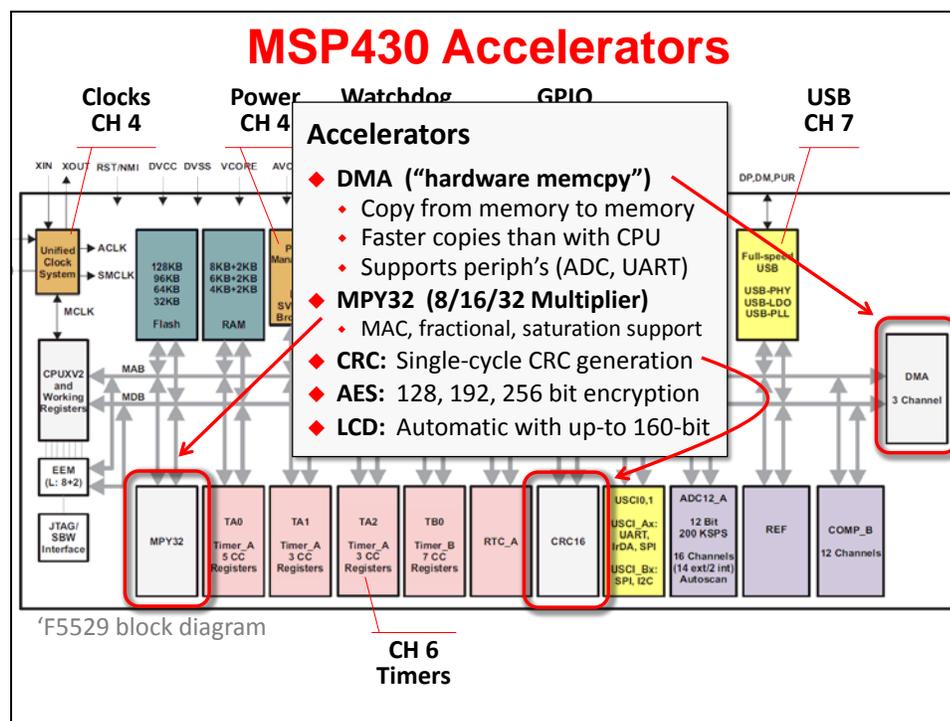
The additional of radios to some MSP430 devices makes them quite unique in the industry. Beyond that, TI has created wireless chips and modules that can be used from any MSP430 device. It's really telling when the cheapest Value Line MSP430 device can actually talk Wi-Fi using TI's CC3000 module. A similar story can be shown across TI's complete portfolio of wireless technologies. In the end, TI is enabling a very low-cost entry point into the "Internet of Things".

Let's not forget the various MSP430 serial ports. They are the workhorses of communications. There are a variety of serial modules, from UART, to SPI, to I2C.

Hardware Accelerators

One question that is often asked, "Why would you put dedicated hardware accelerators onto low-cost, low-power processors?"

It's an interesting question ... with a very practical answer. If a specific functionality is required, accelerators are the most efficient implementation. Take for example, the CRC or AES modules; serial (and wireless) communications are often requiring these functions to make the data transmissions robust and secure. To implement these functions in software is possible, but would actually consume a lot more power. Further, the memory footprint for an algorithm (code and data) often ends up greater than the smaller footprint of the hardwired accelerator. Thus, where it makes sense, you'll see TI adding dedicated hardware modules.

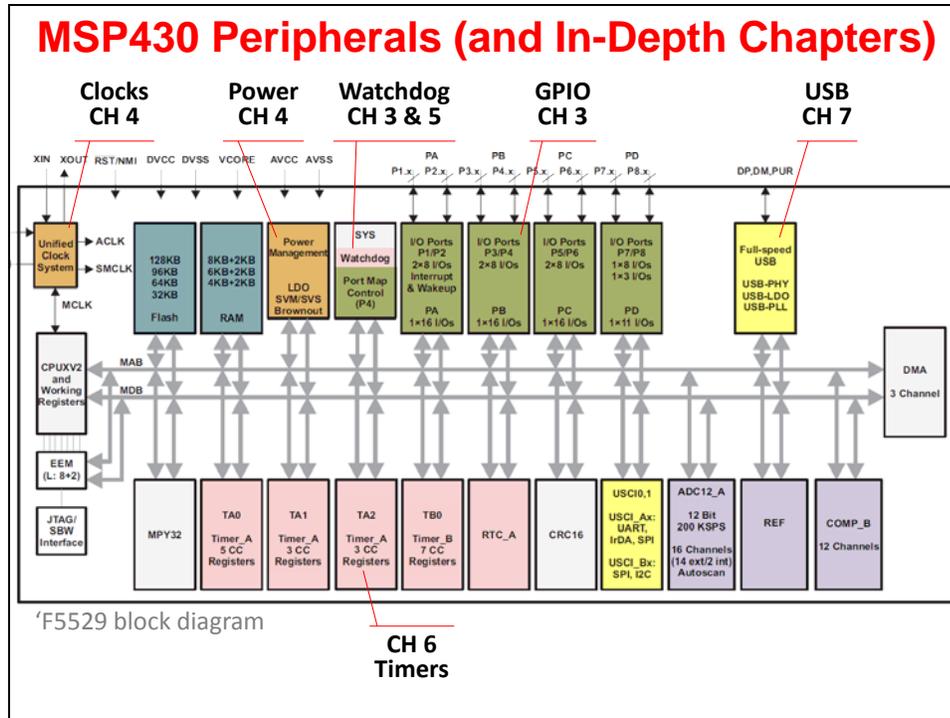


Another example is the multiplier. We can benefit from it without any programming effort, since the compiler automatically uses this hardware, when it's available.

With regards to the Direct Memory Access (DMA) peripheral, we caution you ... if you find yourself using `memcpy()` in your code, you should investigate how the DMA might save you time and power. It also should be utilized in your peripheral driver software whenever and wherever it's available.

Summary

Many of the peripherals we've just outlined are covered - in detail - within their own chapters. Over time, we'll be adding more chapters to the course to cover additional peripherals.



The following comparison table has not been updated for the latest devices; even so, we included it as a quick comparison between some of the MSP430 generations.

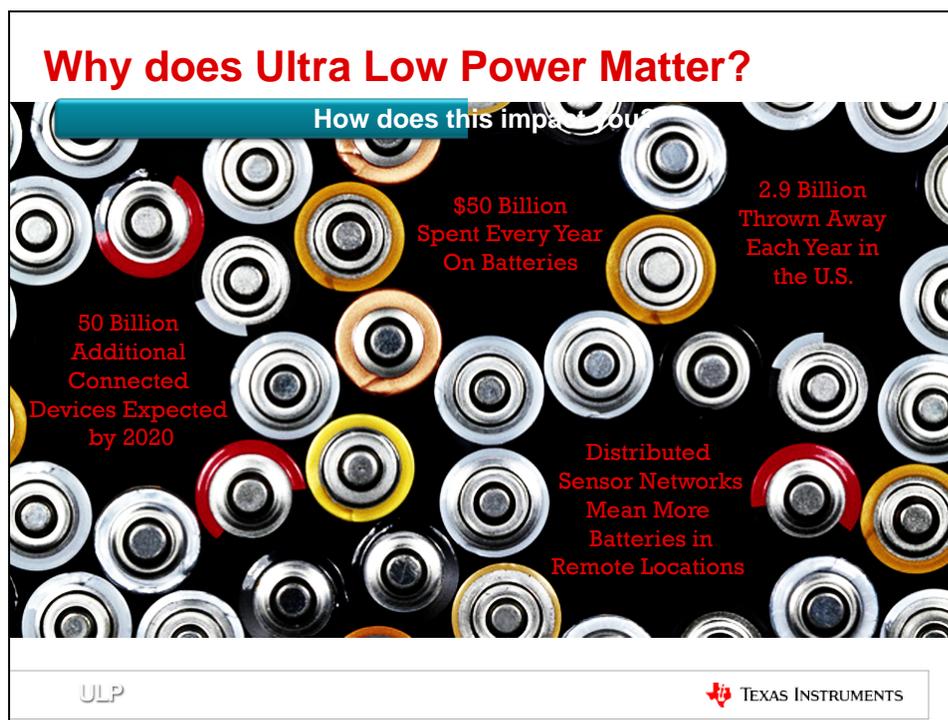
MSP430 Peripheral Overview			
1xx	2xx	4xx	5xx
Basic Clock System	Basic Clock System +	FLL, FLL+	Unified Clock System
Core voltage same as supply voltage (1.8-3.6V)	Core voltage same as supply voltage (1.8-3.6V)	Core voltage same as supply voltage (1.8-3.6V)	Programmable core voltage with integrated PMM (1.8-3.6V)
16-bit CPU	16-bit CPU, CPUX	16-bit CPU, CPUX	16-bit CPUXv2
GPIO	GPIO w/ pull-up and pull-down	GPIO, LCD Controller	GPIO w/pull-up and pull-down, drive strength
N/A	N/A	N/A	CRC16
Software RTC	Software RTC	Software RTC with Basic Timer, Basic Timer + RTC	True 32-bit RTC w/Alarms
USART	USCI, USI	USART, USCI	USCI, USB, RF
DMA up to 3-ch	DMA up to 3-ch	DMA up to 3-ch	DMA up to 8-ch
MPY16	MPY16	MPY16, MPY32	MPY32
ADC10,12	ADC10,12, SD16	ADC12, SD16, OPA	ADC12_A
4-wire JTAG	4-wire JTAG, 2-wire Spy Bi-Wire (Some devices)	4-wire JTAG	4-wire JTAG, 2-wire Spy Bi-Wire

ULP

Does Low Power matter? Our answer is a resounding YES!

Some end-products are only enabled by low-power operation. For example, a wrist watch that cannot make it through a single day would be of little value.

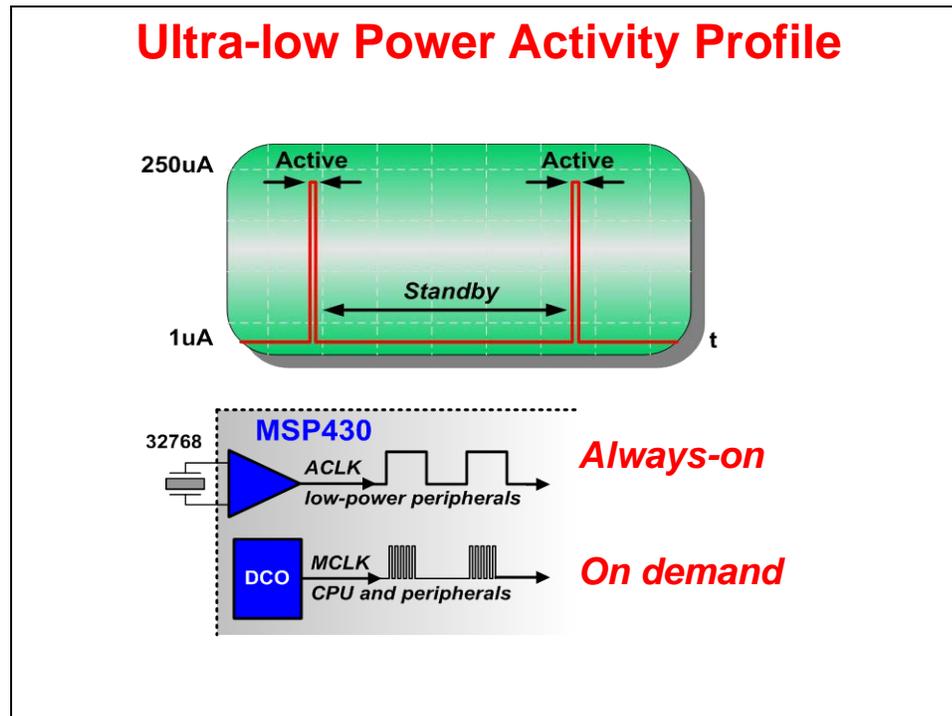
But even when the application does not demand low power, we think it still matters. The trend in electronics over the past few years has been, "Why consume power if you don't have to?" In fact, the MSP430 has found many new applications in the last couple of years where end-users are demanding the reduction of 'phantom load', also known as 'vampire power'. This can be defined as the dissipation of power when electronic products are in standby mode (or even when switched off completely). The MSP430 is a perfect fit for systems trying to prevent these issues.



Profile Your Activities

A fundamental precept of low-power systems is: turn on, do something, then turn off.

The following diagram is a good example of this. One of the low-power modes lets you put the fast components of the system to sleep, while retaining the slow clock running a RTC. Then, as needed, the system wakes up, performs one or more tasks, then goes back into low-power mode.

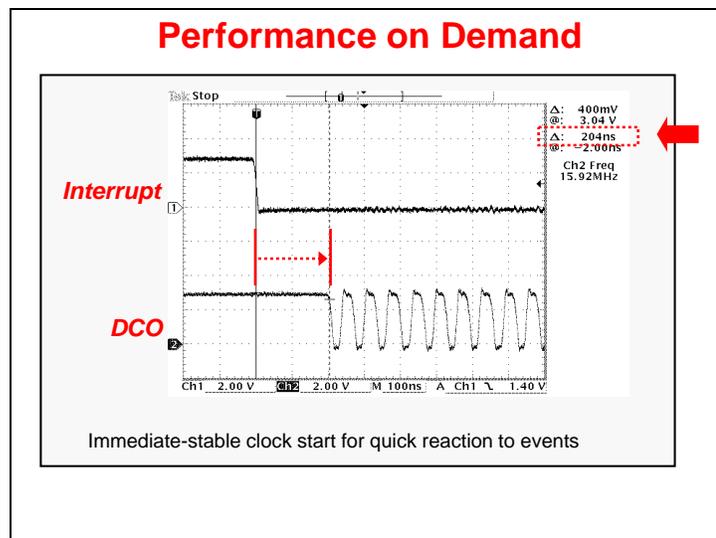


The MSP430 supports this sleep/wake/sleep profile quite well, by providing a variety of low-power modes (LPM). The following chart is an example of the LPM's found on various MSP430 devices, showing which resources are powered down by LP mode. It also broadly indicates what it takes to wake up from a given LPM. (In general, LPM0 and LPM3 are very popular modes.)

Low-Power Modes

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

Almost as important is the 430's ability to wake up quickly from a sleep mode as is demonstrated on the next slide. The DCO (digitally controlled oscillator) is one of the on-chip, high-performance clocks available to the MSP430. The graphic is powerful statement, showing how quickly the clocks and system can be up-and-running after receiving an interrupt.



This slide shows some of the quantitative data for different LPM's across a few different devices. Please, keep in mind that you should always design your system by referencing the datasheet, but this slide does give us a good comparison between the various MSP430 generations.

MSP430™ Series Comparison

Device (mode)		F2xx	F5xx	FR57xx	FR58xx FR59xx
Performance (max)		16MHz	25MHz	24MHz (FRAM at 8MHz)	16MHz (FRAM at 8MHz)
Flex Unified Memory		No	No	FRAM (16K)	FRAM (64K)
Active Mode	AM	>250 μ A/MHz	~180 μ A/MHz	100 μ A/MHz	100 μ A/MHz
RTC Mode	LPM3x	0.8uA	2.5uA	.5uA	0.36uA
Standby Mode	LPM4	0.1uA	1.3uA	5.9uA	0.17uA
	LPM4.5	-	0.18uA	0.32uA	0.01uA
Wake-up from	LPM3	1us	3.5us	78us	7us
	LPM4		150us		
	LPM3.5 LPM4.5	-	2000us	310us	150us

 TEXAS INSTRUMENTS

Much of designing for low-power is common sense; e.g. turn it off when you're not using it. The following slide provides a good set of guidelines (or principles) to use when developing our application.

Principles For ULP Applications

- ◆ Maximize the time in LPM3
- ◆ Use interrupts to control program flow
- ◆ Replace software with peripherals
- ◆ Power manage external devices
- ◆ Configure unused pins properly
- ◆ Efficient code makes a difference
- ◆ Even wall powered devices can be "greener"
- ◆ **Every unnecessary instruction executed is a portion of the battery wasted that will never return**
- ◆ **Use ULP Advisor to help you minimize power in your system**





MSP430™ Ultra-Low Power MCUs

ULP Advisor - Rule Table

- 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
- 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 7.1 Use local instead of global variables where possible
- ULP 8.1 Use 'static' & 'const' modifiers for local variables
- ULP 9.1 Use parts by reference for large variables
- ULP 10.1 Minimize function callings from within ISRs
- ULP 11.1 Use lower bits for loop program control flow
- ULP 11.2 Use lower bits for port bit-banging
- 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
- ULP 13.1 Count down in loops
- ULP 14.1 Use unsigned int for indexing variables
- ULP 15.1 Use bit-masks instead of bit-fields

Many of these guidelines have been distilled into a static code analysis tool that is part of the TI (and IAR) compiler. This tool can help us learn what techniques to apply - or for the more experienced, help us not overlook something we already know.

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 & Ultra-Low-power development experience

- 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
- ULP 5.1 Avoid processing-intensive modulo & div
- ULP 5.2 Avoid processing-intensive floating point
- ULP 5.3 Avoid processing-intensive (s)printf()
- ULP 6.1 Avoid multiplication when HW multiplier
- ULP 7.1 Use local instead of global variables w/

Highlights areas of improvement within code.

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





Launchpad's

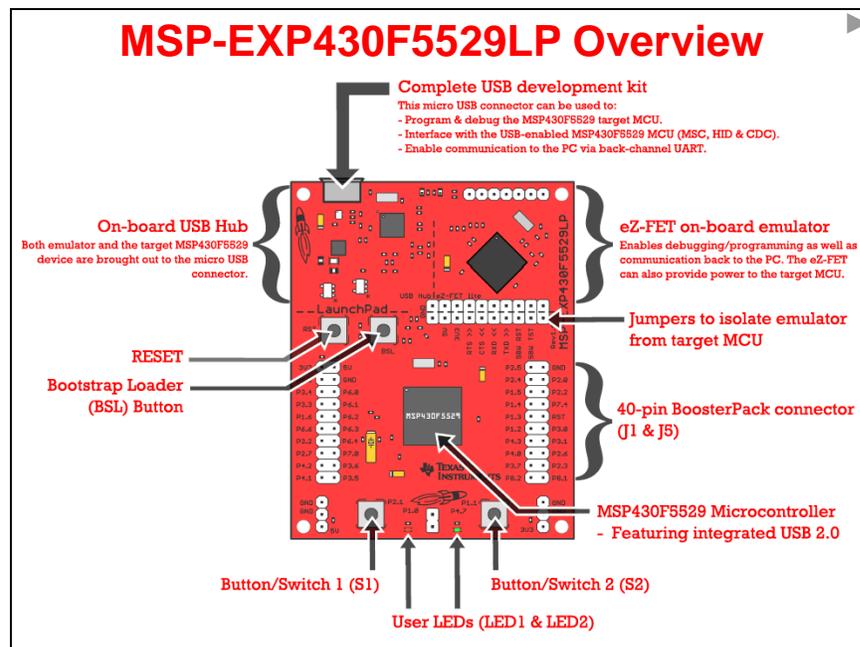
MSP-EXP-430F5529LP

The MSP430F5529 Launchpad is a powerful, low-cost evaluation (and development) tool.



As the diagram shows, the board is really divided into two halves. The top portion (above the ----- line) is an open-source emulator (called eZ-FET lite). This connects our 'target' MSP430 to a PC running a debugging tool, such as Code Composer Studio. You can isolate the emulator from the 'target' processor by pulling the appropriate jumpers (that straddle the dashed line).

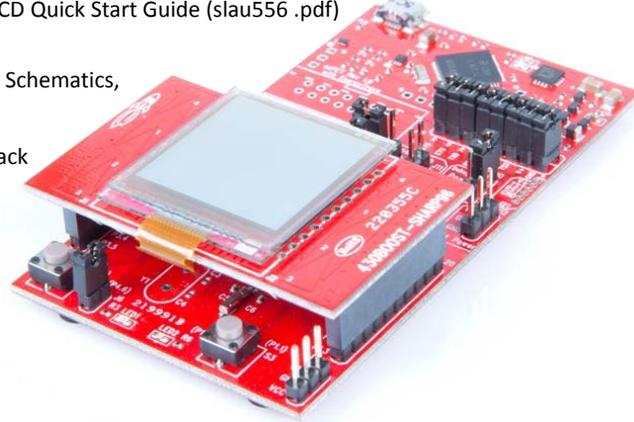
The lower portion of the board provides the target of our application programming. There are LED's, pushbuttons, and pins we can use to let our programs interact with the 'real world'.



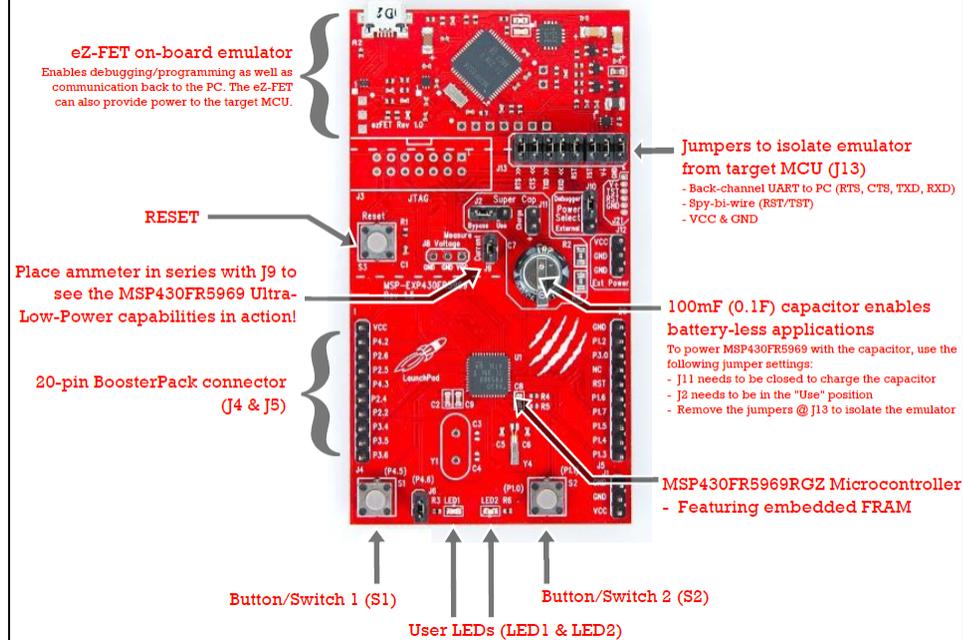
MSP-BNDL-FR5969LCD

MSP-BNDL-FR5969LCD

- ◆ **Wolverine Bundle includes:**
 - MSP-EXP430FR5969 Launchpad
 - 430BOOST-SHARP90 Boosterpack
 - 1x Micro USB cable
- ◆ **Documentation**
 - MSP-EXP430FR5969 Launchpad User's Guide (slau535.pdf)
 - Sharp LCD Boosterpack for Launchpad User's Guide (slau553.pdf)
 - MSP-BNDL-FR5969LCD Quick Start Guide (slau556 .pdf)
- ◆ **Software**
 - MSP-EXP430FR5969 Schematics, BOM, and Software) (slac645.zip)
 - Sharp LCD Boosterpack (slac643.zip)
 - MSP430ware v1.70
 - GraphicsLib v1.20



MSP-EXP430FR5969 Overview

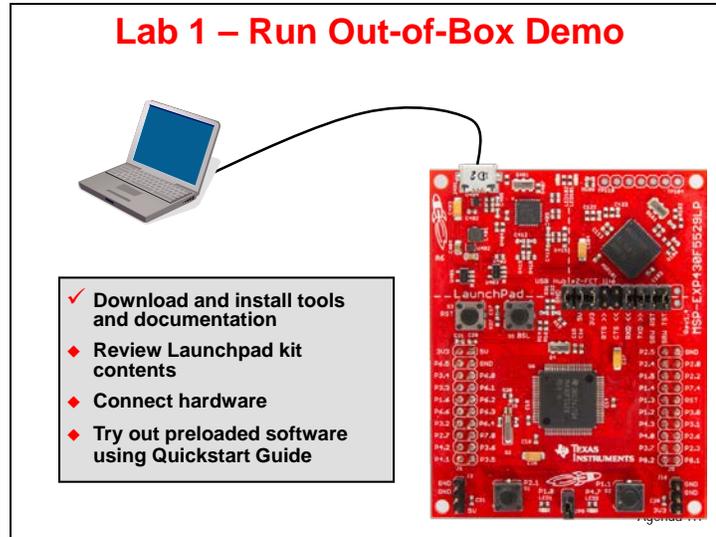


Lab 1a – MSP43F5529 LaunchPad User Experience

'FR5969 Wolverine Launchpad users should jump to [Lab 1b](#) on page [1-39](#).

This lab simply gives us an opportunity to pull the board out of the box and make sure it runs properly. The board arrives with a USB keyboard/memory application burned into the flash memory on the 'F5529.

You can either follow the quick start directions on the card included with the Launchpad, or follow the directions here. We re-created the directions since some folks have a tough time reading the small print of the quick start card.



Examine the LaunchPad Kit Contents

1. Open up your MSP430F5529 LaunchPad box. You should find the following:

- The MSP-EXP430F5529LP LaunchPad Board
- USB cable (A-male to micro-B-male)
- “Meet the MSP430F5529 Launchpad Evaluation Kit” card

2. Initial Board Set-Up

Using the included USB cable, connect the USB emulation connector on your evaluation board to a free USB port on your PC.

A PC's USB port is capable of sourcing up to 500 mA for each attached device, which is sufficient for the evaluation board. If connecting the board through a USB hub, it must usually be a powered hub. The drivers should install automatically.

3. Run the User Experience Application

Your LaunchPad Board came pre-programmed with a User Experience application. This software enumerates as a composite USB device.

- HID (Human Interface device): an emulated keyboard
- MSC (Mass Storage class): an emulated hard drive with FAT volume

The contents of the hard drive can be viewed with a file browser such as Windows Explorer.

4. View the contents of the emulated hard drive

Open Windows Explorer and browse to the emulated hard drive. You should see four files there:

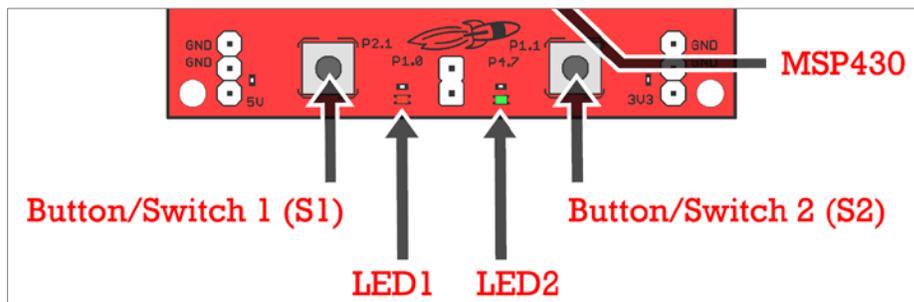
- **Button1.txt** – the contents of this file are "typed out" to the PC, using the emulated keyboard when you press button S1
- **Button2.txt** – the contents of this file are "typed out" to the PC, using the emulated keyboard when you press button S2
- **MSP430 USB LaunchPad.url** – when you double-click, your browser launches the MSP- EXP430F5529LP home page
- **README.txt** – a text file that describes this example

5. Use S1 and S2 buttons to send ASCII strings to the PC

The LaunchPad's buttons S1 and S2 can be used to send ASCII strings to the PC as if they came from a keyboard. These strings that are sent are stored in the files Button1.txt and Button2.txt, respectively; and these files can be modified to change the strings. The text string is limited to 2048 characters, so even though you can make the file contents longer, be aware that the string will be truncated to 2048.

Open Notepad. In the start menu, type "Run", then type "Notepad"

To send the strings to Notepad, press S1.



What do you see? _____

Now press S2. What happens now? _____

The default ASCII strings stored in the two text files are:

- **Button1.txt:** "Hello world"
- **Button2.txt:** an ASCII-art picture of the LaunchPad rocket

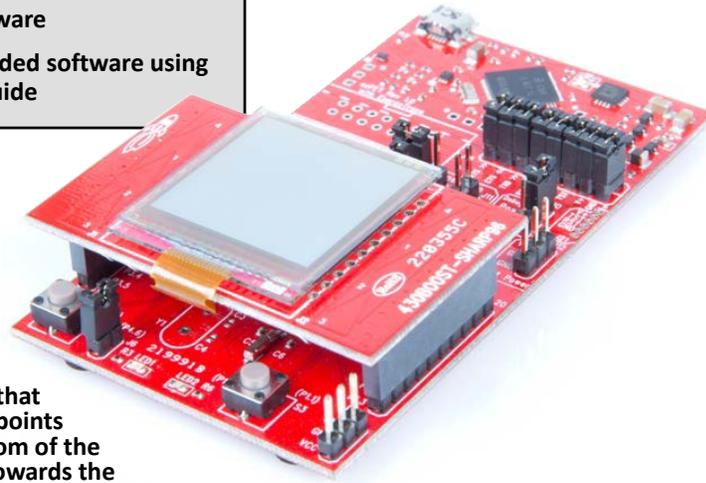
For the rocket picture, please note that the display can be affected by settings of the application receiving the typed characters. On Windows, the basic Notepad.exe is recommended.

Note: If you have an older version of the 'F5529 Launchpad (prior to "Revision 1.5), then your board must enumerate with a USB host before it can receive power. This means USB batteries – which do not contain a USB host – cannot be used as a power source.

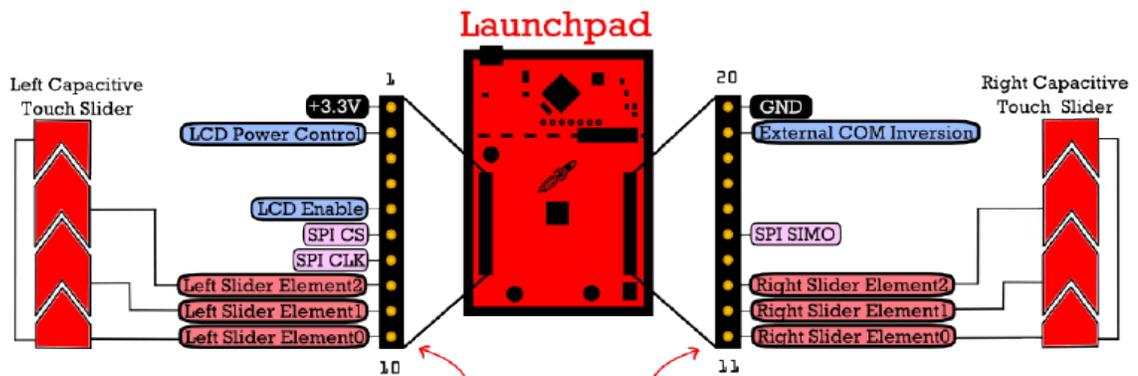
Lab 1b – MSP43FR5969 LaunchPad User Experience

Lab 1 – MSP430FR5969 Launchpad

- ◆ Verify tool installation
- ◆ Review Launchpad kit contents
- ◆ Connect hardware
- ◆ Try out preloaded software using Quick Start Guide



Install display so that the ribbon cable points towards the bottom of the Launchpad (i.e. towards the buttons), as shown.



BoosterPack Connectors

Figure 3. BoosterPack Default Pinout

Quick Start Procedure

The out-of-box demo:

The MSP-EXP430FR5969 LaunchPad features an MSP430FR5969 device that is pre-loaded with some demo functionality.

1. Connect the 430BOOST-SHARP96 Display BoosterPack

The demo code for this LaunchPad requires the 430BOOST-SHARP96 BoosterPack to be connected.

2. Connecting to the computer

Connect the LaunchPad using the included USB cable to a computer. If prompted, install any necessary software. A green power LED should illuminate.

3. It's alive!

When connected to your computer, the LaunchPad will power up and a series of images should cycle on the 430BOOST-SHARP96 BoosterPack. A Red LED (LED1) will also blink during this startup sequence.

4. The MAIN MENU

There are several demo applications. To select one of the modes, simply use the left slider on the 430BOOST-SHARP96 BoosterPack. When the desired mode is highlighted, press button (S2) on the LaunchPad to select it. To exit any of the modes, press button (S1) to return to the MAIN MENU.

Clock

This mode provides an accurate clock using RTC in Low Power Mode 3 (LPM3). Use the slider to change the time settings. Use button (S2) to save your settings.

FRAM Speed

This mode shows the maximum write speed of FRAM on the BoosterPack display. FRAM is written in 1kB blocks. Direct Memory Access (DMA) is used to transfer data and the main clock (MCLK) is set to run @ 8MHz. This application writes data to FRAM @ ~7564 kBps (typical Flash write speeds = 13 kBps). Also shown is the total number of kB written as well as the FRAM write endurance (%).

Battery Free

This mode runs a stopwatch without batteries by leveraging the on-board capacitor. When entering this mode, you are presented with 2 options:

Run App: In this mode, the MSP430FR5969 stays in an ultra-low power LPM3.5 mode consuming ~500 nA. A RTC is available to wake up the MCU once a minute to read the input voltage from the capacitor & stores that data into FRAM. During this time, a stopwatch is continuously updated. When the MCU is asleep in LPM3.5, the display is turned off. To wakeup the MCU to see the remaining charge of the capacitor and the current time on the stopwatch, press button (S2). Press button (S2) again to go back to LPM3.5. Ensure the capacitor is being used by following the jumper settings in the diagram to the left.

Transfer Data: In this mode, the logged voltage readings from a previous "Run App" execution are read from FRAM and sent to a PC via back-channel UART over USB. These readings can be read using any terminal/serial monitor application on your PC.

Active Mode

The active power consumption of the MSP430FR5969 is dependent on three things: the code, data cache hit ratio & clock speed of the CPU. Choose the desired operating frequency of the CPU (1MHz, 2.67MHz, 4MHz or 8MHz). Then, choose your desired cache hit ratio (50%, 66%, 75% or 100%). Pressing button (S2) will allow you to enter/exit the Active Mode code operation. To measure active mode current, remove Jumper J9 & place an ammeter across the J9 terminals.

SliderBall Game

This mode demonstrates the capacitive touch I/O pins available on the MSP430FR5969. Two linear sliders are available on the 430BOOST-SHARP96 BoosterPack, which are used to control two paddles. Move the paddles to keep the ball in play! Your high scores are saved in FRAM & is retained on subsequent power cycles. They are only lost when you re-program the device.

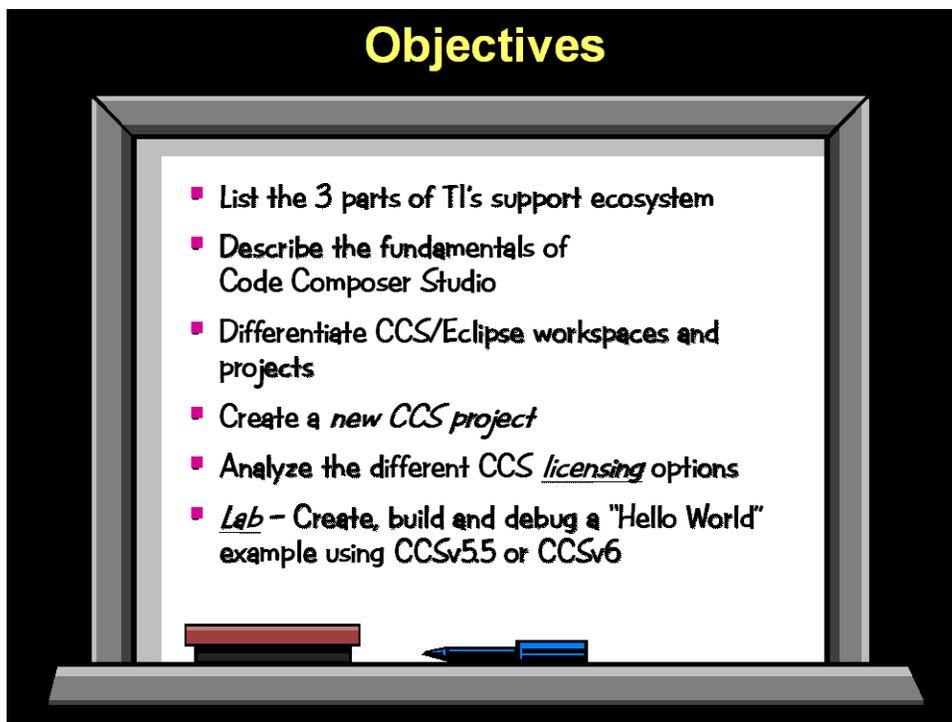
Programming C with CCS

Introduction

This chapter will introduce you to Code Composer Studio (CCS).

In the lab, we will build our first project using CCS and then experiment with some useful debugging features. Even if you have some experience with CCS, we hope that you will find exercise to be a good review – and in fact, that you might even learn a few new things about CCS that you didn't already know.

Learning Objectives

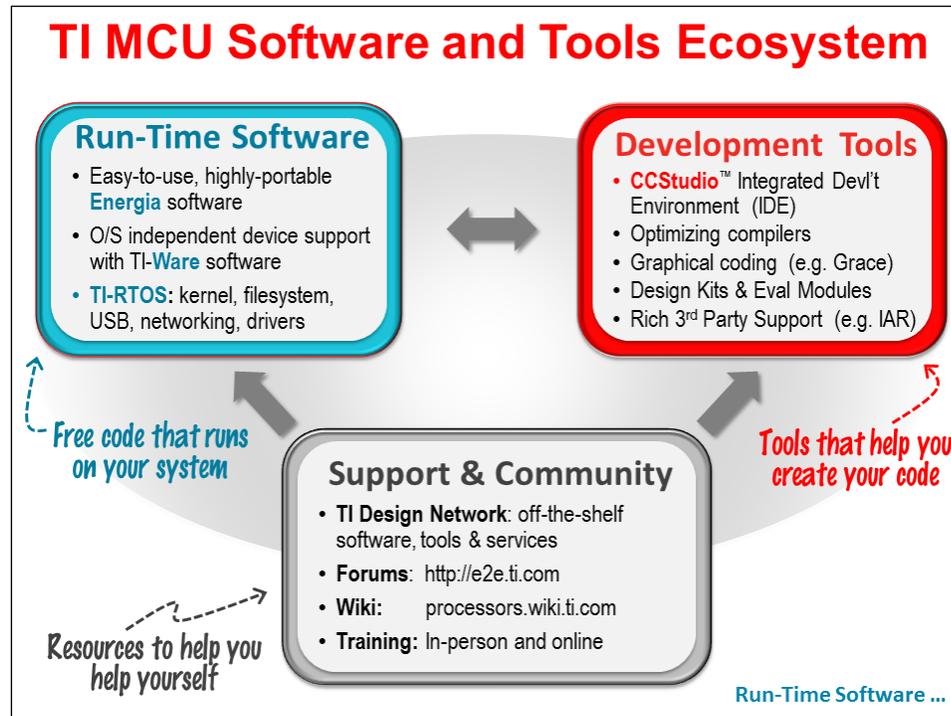


Chapter Topics

Programming C with CCS	2-1
<i>TI Support Ecosystem.....</i>	<i>2-3</i>
Run-Time Software	2-4
Low-level C Header Files	2-4
MSP430ware (DriverLib).....	2-4
Energia	2-5
TI-RTOS	2-5
Development Tools	2-6
Integrated Development Environments (IDE)	2-6
Other MSP430 Tools.....	2-7
Support & Community	2-8
Wiki.....	2-8
Forums	2-9
<i>Examining CCS.....</i>	<i>2-10</i>
Functional Overview.....	2-10
Editing	2-10
Debugging.....	2-12
Target Config & Emulation	2-12
Perspectives.....	2-13
Workspaces & Projects.....	2-14
Some Final Notes about CCS/Eclipse	2-15
Creating a Project	2-16
Adding Files to a project.....	2-17
Portable Projects.....	2-17
Licensing/Pricing	2-18
Changing a CCS User Licence	2-19
<i>Writing MSP430 C Code.....</i>	<i>2-20</i>
Build Config & Options.....	2-20
Processor Options.....	2-20
<u>Debug Options</u>	2-21
<u>Optimize Options</u> (aka Release Options).....	2-21
Build Configurations	2-22
Data Types.....	2-23
Device Specific Files (.h and .cmd).....	2-24
MSP430 Compiler Intrinsic Functions.....	2-26
<i>Lab 2 – CCStudio Projects.....</i>	<i>2-27</i>

TI Support Ecosystem

TI's goal is to provide an entire ecosystem of tools and support. Development tools, like Code Composer Studio are just the starting point; then add in software libraries that run on your target processor as well as wiki's and support forums.

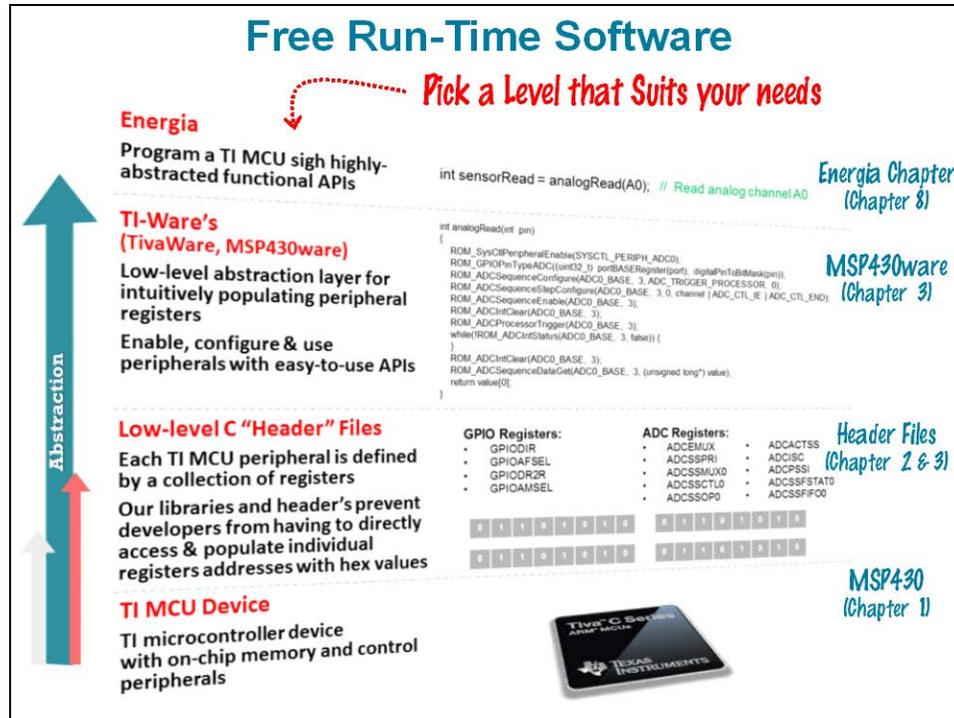


We'll take a brief look at all three parts of the Ecosystem:

- Run-Time Software
- Development Tools
- Support and Community

Run-Time Software

The MSP430, like most of TI's microcontroller (MCU) platforms, is supported by a rich, layered approach to foundational software.



Low-level C Header Files

Working our way up from the bottom, the MSP430 family provides a custom C language header file (and linker command file) for each device. These header files provide symbols that define all the various registers, pointers and bitfields found on 'your' device. Not only do they minimize the number of times you'll need to pour through the user guide and datasheet (to figure out obsequious hex values), but they make your code more readable. We also hope that providing a common set of symbols will make it easier to share and reuse code. Finally, since these files primarily contain 'definitions', they don't add any 'bulk' to your code. *(We'll discuss these files further at the end of this chapter.)*

MSP430ware (DriverLib)

MSP430ware is a collection of libraries, examples, and tools. We'll examine many of these items in the next chapter. What we want to call out here is the MSP430ware Driver Library – also known as "DriverLib".

MSP430ware DriverLib borrows heavily from the stellar TivaWare driver library that ships with TI's ARM Cortex-M4F devices. In each case, DriverLib provides a low-level abstraction layer that makes writing code easier. MSP430ware even builds upon the 'header' file layer making it easier to dig-thru the source code (which is provided) if you ever want to discover how an API is implemented. Furthermore, it means you can easily mix-and-match DriverLib with 'header' file code.

Our main goal is to help you improve the readability and maintenance of your '430 code; that said, we also strive to keep the library as small and efficient as possible.

If you've ever had to return to low-level code a year later – or port it to another device in the same MCU family – you'll really appreciate the convenience and ease-of-use of DriverLib.

Energia

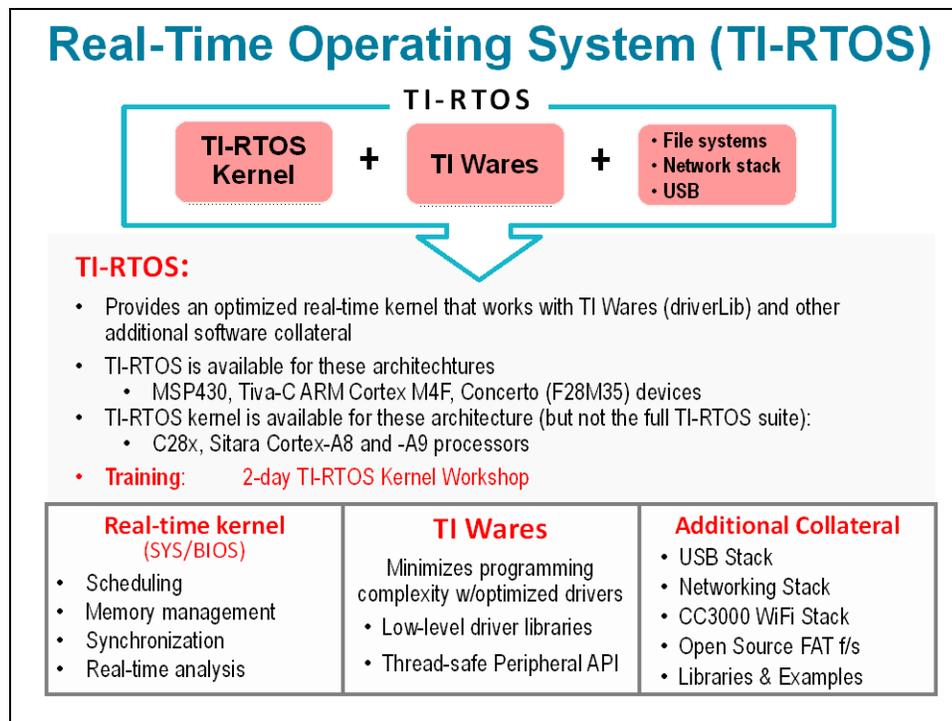
Energia is a community-based port of the ever-popular Arduino. This software makes it easy for users to grab code already available in the Arduino community and put it to good use on TI's MSP430 Launchpads. In other words, it puts the word "rapid" in rapid-prototyping.

In fact, Energia isn't just for prototyping anymore. There are many customers using this in small to midsize production systems. In any case, whether you use it for prototyping or otherwise, you'll find it an easy, fun way to get your ideas into hardware. (With good reason, Arduino helped coin the phrase, "Sketching with hardware".)

(Coming in 2014, look for Arduino support in TI's high-end development tool: Code Composer Studio.)

TI-RTOS

TI's real-time operating system (TI-RTOS) is a highly capable package of system-building software. It's not just enough to package a bunch of software libraries together into a single executable; the TI-RTOS team validates all the components against each other – creating examples that utilize all the various libraries.



The soul of TI-RTOS is the TI-RTOS Kernel (formerly named SYS/BIOS). The kernel provides a broad set of embedded system services, most notably: Threads, Scheduling, Semaphores, Instrumentation, Memory Management, inter-thread communication and so on. It's been built with modularity in mind, so it's easy to take the parts that make sense for your application and exclude the parts that don't.

TI-RTOS includes the kernel plus a number of customized drivers built upon the TI-wares (i.e. MSP430ware DriverLib). They've also thrown in a variety of other O/S level packages, such as: USB Stack, WiFi networking, FatFs. (The list will continue to grow, so keep your eye on the TI-RTOS [webpage](#).)

Development Tools

Integrated Development Environments (IDE)

TI Code Composer Studio is a highly capable integrated development tool (IDE). Built on the popular Eclipse IDE platform, TI has both simplified and extended the Eclipse framework to create a powerful, easy-to-use development platform. In fact, the MSP430 was the first MCU inside TI to get the Eclipse treatment ... but it's come a long way since then.

Development Tools for MSP430				
				
Evaluation License	<input type="checkbox"/> 32KB code-size or 30-day limit <input type="checkbox"/> Upgradeable	<input type="checkbox"/> Full function <input type="checkbox"/> JTAG limited after 90-days	N/A	N/A
Compiler	IAR C/C++	TI C/C++	GCC*	GCC*
Debugger and IDE	<input type="checkbox"/> C-SPY <input type="checkbox"/> Embedded Workbench	<input type="checkbox"/> TI or GDB <input type="checkbox"/> CCSstudio (Eclipse-based)	Energia IDE [†] (Arduino port)	MSPDEBUG (gdb proxy)
Full Upgrade	\$2700	\$445	Free	Free
JTAG Debugger	J-Link \$299	MSP-FET430UIF \$99	No JTAG <input type="checkbox"/> serial.printf() <input type="checkbox"/> LED or scope	MSP-FET430UIF \$99

* GCC*: CCSv6 contains GNU GCC compiler * †CCSv6 allows you to debug Energia projects using full debug toolset
 MSPGCC was available prior to GNU GCC

As highly as we value CCS, we know it may not be for every user. To that end, we work diligently with our 3rd parties and the open-source community to provide MSP430 compatibility in their ecosystems.

IAR Systems, for example, commands a huge fan base among MCU developers. Whenever the MSP430 team creates new tooling, they don't just think about how it can be integrated into CCS, but they also consider how it can be used by our IAR customers as well. With their highly regarded compiler, many of our customers think that the extra cost of IAR is easily worth it.

At the other end of the spectrum, we know that some of our customers cannot even afford the low-cost price-point of CCS. For hobbyists and folks needing to rapid-prototype systems, the Energia open-source port of Arduino is a great option.

If you want to stay in the open-source domain, but step down from the abstraction provided by Energia, you can write C code using the open-source version of the Gnu Compiler (GCC).

It doesn't matter which tool suite you choose, in any case, you'll still have all the other MSP430 ecosystem components at your disposal. For example, MSP430ware DriverLib works in all of these environments.

Other MSP430 Tools

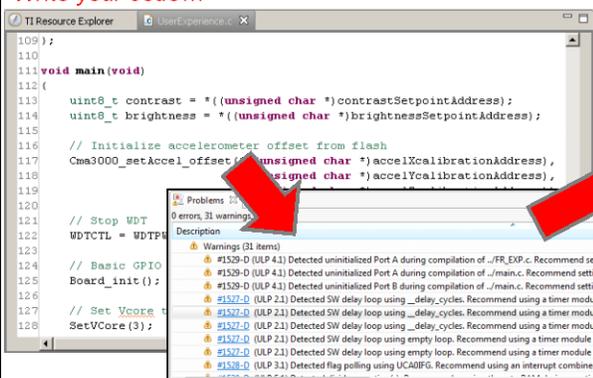
The MSP430 team has created a number of additional tools to support development of MSP430 applications. For example, since low-power designs are a major consideration for MSP430 users, the **ULP Advisor** tool provides static analysis of your code – from a power perspective – every time you compile. Novice and experienced users alike will find something they missed when trying to cut every nano-amp from their system.

ULP (Ultra-Low Power) Advisor

Squeezing out every last nanoAmp

- ◆ Checks your code against an MSP430 ULP Checklist
- ◆ The ULP Advisor wiki includes a description of each rule, proposed remedies, code examples & links to related e2e online forum posts
- ◆ ULP Advisor is *FREE* and is available as a plugin for CCS
- ◆ Standalone command-line tool for use with other IDEs
- ◆ Learn more at www.ti.com/ulpadvisor

Write your code...



ULP Advisor finds areas for code improvement

ULP Advisor - Rule Table

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
ULP 5.1	Avoid processing-intensive operations: modulo, division
ULP 5.2	Avoid processing-intensive operations: floating point
ULP 5.3	Avoid processing-intensive operations: (sprintf)
ULP 6.1	Avoid multiplication on devices without hardware multiplier
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 callings from within ISRs
ULP 11.1	Use lower bits for loop program control flow
ULP 11.2	Use lower bits for loop program control flow

Wiki provides details & remedies

ULP Advisor > Rule 1.1 Ensure LPM Usage

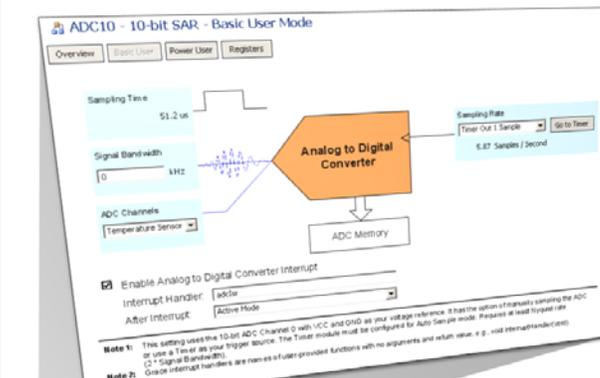
What it means
The MSP430 microcontroller achieves low power consumption by minimizing the time staying in the active mode. To be configured to operate without CPU intervention and CPU only needs to wake up to process critical events.

Why it is happening
The application in active standby will greatly increase power consumption and reduce battery life if it is not configured to enter LPM when appropriate. A warning is issued when no LPM-entering instruction can be found in any code file in the project.

Remedy
Use low power modes in your application when applicable, i.e. while waiting for certain peripheral events. LPM-entering instructions in the code where applicable such as:
`__LPM0();`
`__LPM1();`
`__LPM2();`

Code Example

Grace, on the other hand, provides a graphical development interface for TI's Value-Line and Wolverine series of devices. Just by selecting a few simple choices from the GUI interface, you can quickly build up your system. Grace outputs well commented DriverLib and/or Header file code. Use it to build up a custom set of drivers – or build your entire application – in Grace.



Grace™

- ◆ A free, graphical user interface for use with CCSstudio or IAR
- ◆ Simplifies peripheral configuration
- ◆ Prevents contradicting H/W configurations
- ◆ Generates well-commented source code
- ◆ Currently supports: G2xx (Value Line) and FR5xx (FRAM based) devices

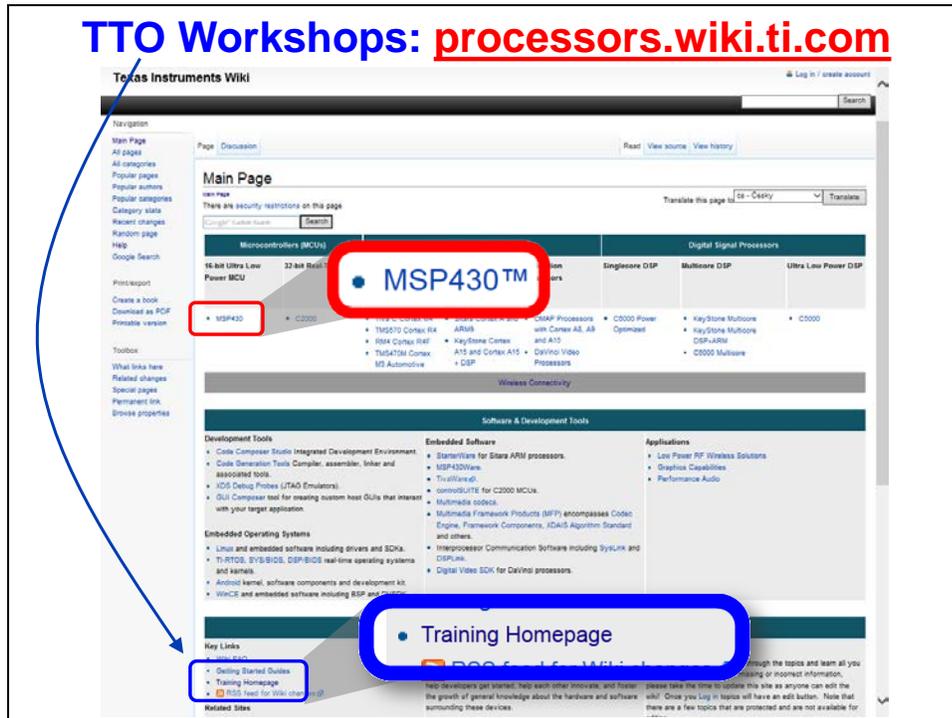
MSP430 Workshop - Programming C with CCS

2 - 7

Support & Community

Wiki

The TI Embedded Processor's wiki provides a wealth of information. Highlighted below you'll find the MSP430 and TTO (Technical Training Organization) links found on the main TI wiki page. Of course, most anything else you might be looking for can be easily found from the Google search box, right under the "Main Page" title.



From the TTO wiki page you'll find a link to this workshop. You most likely already found this page when following our download/installation instructions to get ready for the workshop. You may also want to return here often to access updates to [these workshop materials](#).

This Workshop

Hands-On Training for TI Embedded Processors

Hands-On Training for TI Embedded Processors

TI's Technical Training Organization (TTO) conducts hands-on training for TI embedded processors at various sites, organized by specific processor families. You can also enroll in a live workshop using the links below.

Workshop Descriptions and Materials

[MSP430™ 16-bit Ultra-Low-Power MCU Training](#)

Getting Started with the MSP430™ LaunchPad Workshop - online videos provided

[MSP430™ 5xx One-Day Workshop](#)

[MSP430™ 5xx One-Day Workshop - online videos provided](#)

[MSP430™ FRCT™ Training Workshop](#)

Getting Started with the MSP430 LaunchPad Workshop

Getting Started with the MSP430 LaunchPad Workshop
Version 2.22
July 2013

CapTouch Chapter updated with support for New Tools
Energia (i.e. Arduino) Chapter for MSP430 Launchpad

Introduction

The **Getting Started with the MSP430 LaunchPad Workshop** is an in-depth introduction into MSP430 basics. The LaunchPad is an easy-to-use development tool intended for beginners and experienced users alike for creating microcontroller-based applications.

This on-line, in-depth workshop is free. Additionally, you can sign up to take the class live, with an instructor. You can watch a video of the workshop modules by clicking the 'Links' in the Video column.

The topics for this course include:

#	Chapter	Description	Download	Video
0.	Install Guide	Software Installation Procedures for Workshop Lab Exercises	Installation Guide	N/A
1.	Introduction	An introduction to the MSP430 Value Line series of products	(470KB)	Link
2.	Code Composer Studio (CCS)	Introduction to Code Composer Studio IDE		Link



Forums

There are a wide ranging set of user-to-user forums. Check them out, when you have a ???

Engineer-2-Engineer Forums



Products Applications Tools & Software Support & Community Sample & Buy About TI

TI E2E™ Community

engineer to engineer, solving problems

[Join](#) | [Sign In with my.TI Login](#)

Support Forums Blogs Groups Videos 简体中文

Search Community

TI Home » TI E2E Community

Find out if your question has already been answered

Advanced Search >

Choose a support forum to post a new question

ARM®-based Processors

Amplifiers

DLP® & MEMS

Applications

Digital Signal Processors

Broadband RF/IF & Digital Radio

Interface

Tools & Software

Microcontrollers

Clocks & Timers

Logic

Wireless Connectivity

OMAP™ Applications Processors

Data Converters

Power Management

See all support forums here >

Recent Forum Activity

[swati arora](#) replied to [writing a simple application using cc2540 in Low Power RF Bluetooth® Low Energy & ANT Forum](#)

a few seconds ago

TI E2E Top Contributors

Top Contributors

Top TI Contributors

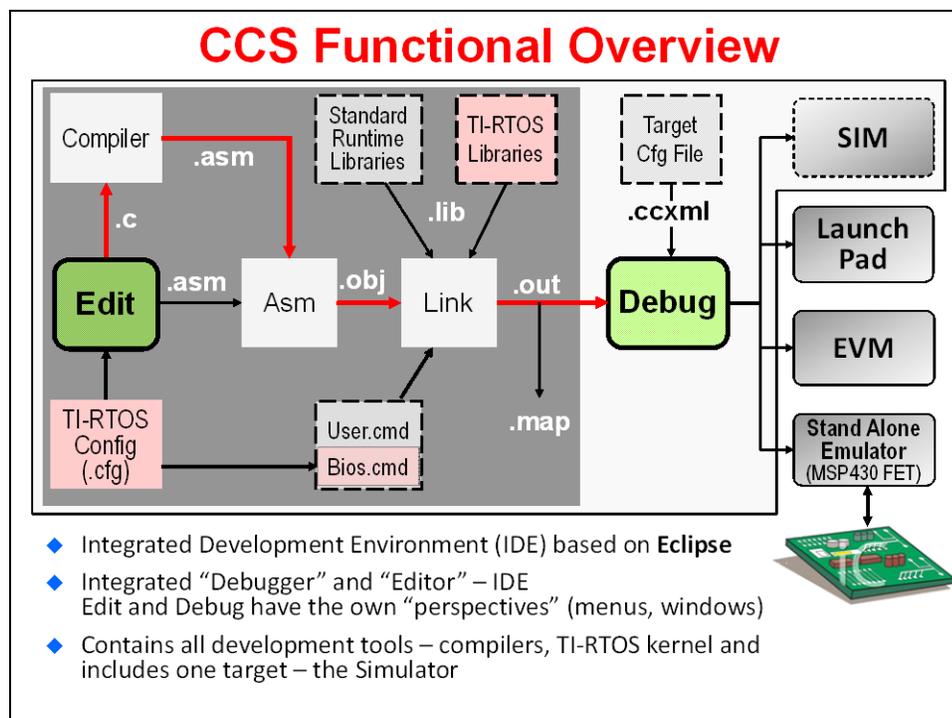
<http://e2e.ti.com>

Examining CCS

Functional Overview

As described earlier, Code Composer Studio is TI's Eclipse based Integrated Development Environment (IDE). You might also think of IDE as meaning, "Integrated Debugger and Editor", since that's really what it provides. CCS is made up of a suite of tools that help you:

- Edit and Build your code
- Debug and Validate your code

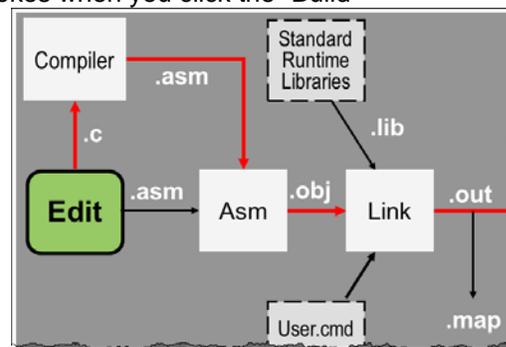


Editing

On the Editing side, you'll find the Compiler → Assembler → Linker tools combine to create the executable output file (.out). These are the tools that CCS invokes when you click the "Build" toolbar button.

Let's do a brief summary of the files shown here:

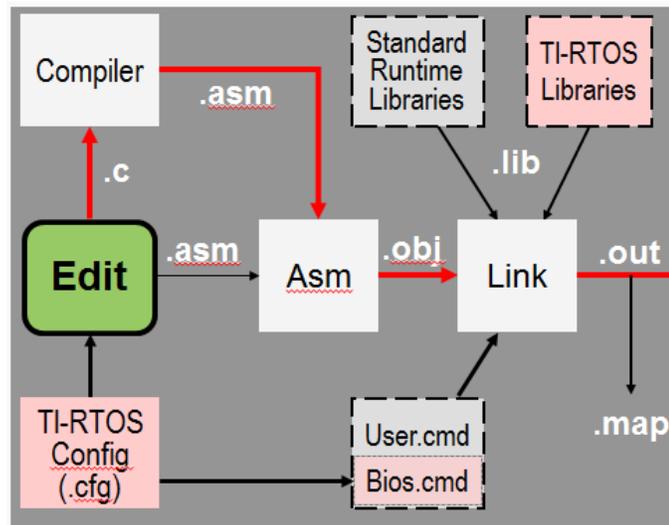
- .c Your C (or C++) source code files
- .asm Assembly files are created by the compiler. By default, they're considered temporary and deleted; though, you can tell CCS to retain them.
- .obj Relocatable object files. Again thought of as temporary and deleted when build is complete.
- .lib Any object library you want to reference in your code.
By default, TI's compiler ships with a run-time support library (RTS) that provides standard C functions. See the compiler user's guide for more information. ([slau132.pdf](#))



- .cmd Linker command files tells the linker how to allocate memory and stitch your code and libraries together. TI provides a default linker command file specific to each MSP430 device; it is automatically added to your project when you create a new project. You can edit it, if needed, though most users get by without ever touching it.
- .out The executable output file. This is the file that is loaded into Flash or FRAM on your MSP430 MCU whenever you click the “Debug” button on your CCS toolbar.
- .map The map file is a report created by the linker describing where all your code and data sections were linked to in memory.

Please refer to the *MSP430 Compiler User’s Guide* ([slau132.pdf](#)) and *MSP430 Assembly Language User’s Guide* ([slau131.pdf](#)) for more information on the TI code generation tools.

The remaining “BUILD” tools shown in our diagram are related to the TI-RTOS kernel.



In essence, the TI-RTOS kernel is composed of many object code libraries. By creating a new project based on the TI-RTOS template, CCS will automatically:

- Link in the required libraries
- Add the TI-RTOS configuration file (.cfg)

The configuration file provides a GUI interface for specifying which parts of the kernel you want to use; helping you to create any static O/S objects that you want in your system; as well as creating a second linker command file that tells the linker where to find all the kernel’s libraries.

While we briefly discuss TI-RTOS scheduling and threads during the Interrupts chapter of this workshop, we recommend you take a look at the [TI-RTOS Kernel Workshop](#)¹ if you want more information.

¹ http://processors.wiki.ti.com/index.php/Introduction_to_the_TI-RTOS_Kernel_Workshop

Debugging

Once again, the “debug” side of the Code Composer Studio lets you download your executable output (.out) file onto your target processor (i.e. MSP430 device on your Launchpad) and then run your code using various debugging tools: breakpoint, single-step, view memory and registers, etc.

You will get a lot more detail and experience with debugging projects when running the upcoming lab exercises on your Launchpad.

Target Config & Emulation

CCS needs to understand how to connect to your target. That is, which target processor do you want to download-to and run your code on?

Going back to older revisions of CCS (versions prior to CCSv4), TI provided a stand-alone tool where you would specify how the target board was connected to CCS. Nowadays, this feature has been integrated into CCS. The Target Configuration File (.ccxml) contains all the information CCS needs to connect and talk to your target (be it a board or a software simulator).

Target Configuration and Emulators

```

graph TD
    TCF[Target Cfg File] --> Debug[Debug]
    Debug --> SIM[SIM]
    Debug --> LP[Launch Pad]
    Debug --> EVM[EVM]
    Debug --> EMU[EMU]
    EMU --> Board[Target Board]
          
```

- ◆ The Target Configuration File specifies
 - Connection to the target (e.g. USB FET)
 - Target device (e.g. MSP430F5529)
 - GEL file (if applicable) for h/w setup

- ◆ EMU Connection Options
 - MSP-FET430 stand-alone FET
 - EZ-FET built into development boards (i.e. Launchpad)
 - (non MSP430) XDS100v1/v2, 200, 510, 560, 560v2

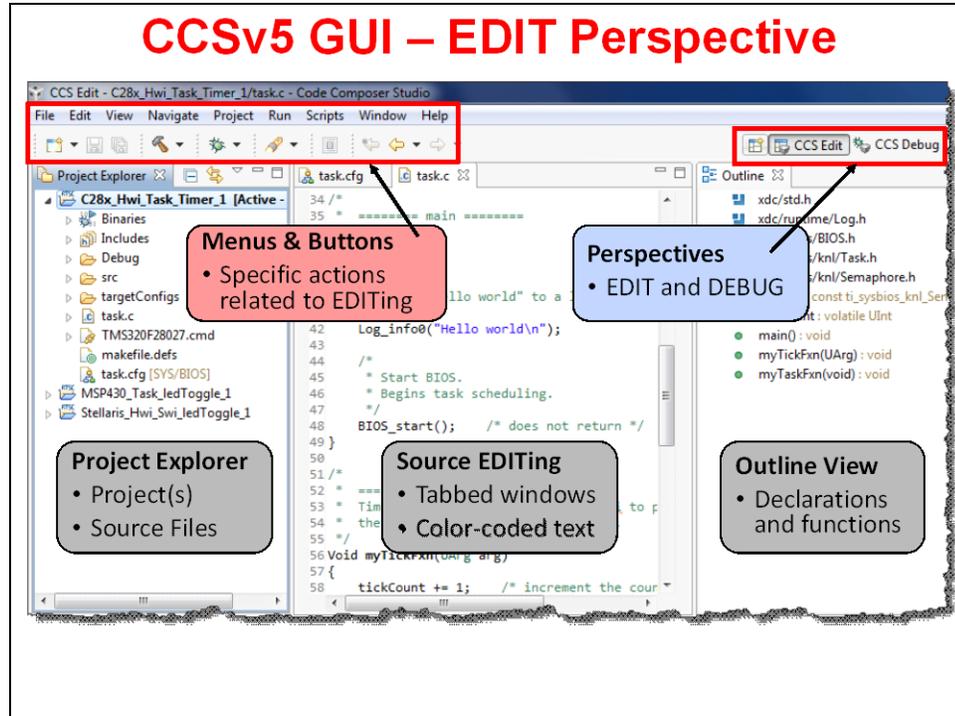
For the MSP430, the CCXML file is automatically created when you create a new project. This file is based on your telling CCS which CPU variant you’ve chosen (i.e. MSP430F5529); as well as which “Connection” you are planning to use for connecting your PC to the target board.

The most common connection that MSP430 users choose is: **TI MSP430 USB1 [Default]**
In fact, this is the connection we’ll be using in the upcoming lab exercises.

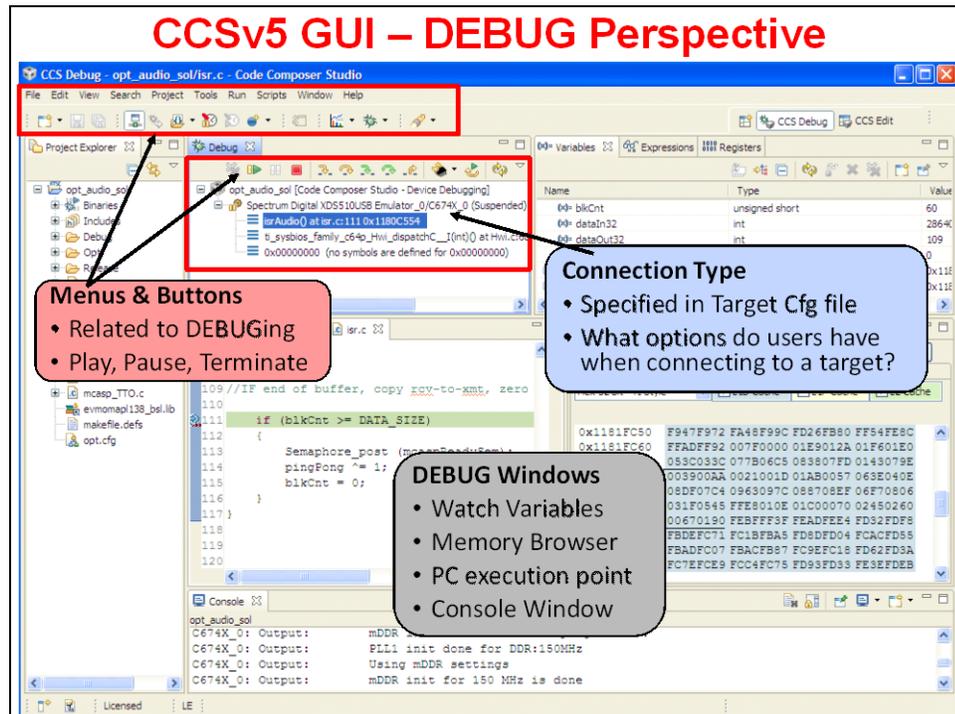
Note: If you ever get an error that indicates CCS doesn’t know how to connect to the target, you probably didn’t specify the “connection” when creating your project. You can easily fix this by editing the project’s properties.

Perspectives

In Eclipse, *Perspectives* describe an arrangement for toolbars and windows. **CCS Edit** and **CCS Debug** are the two perspectives that are used most often. Notice how the perspectives differ for each of the modes shown below.



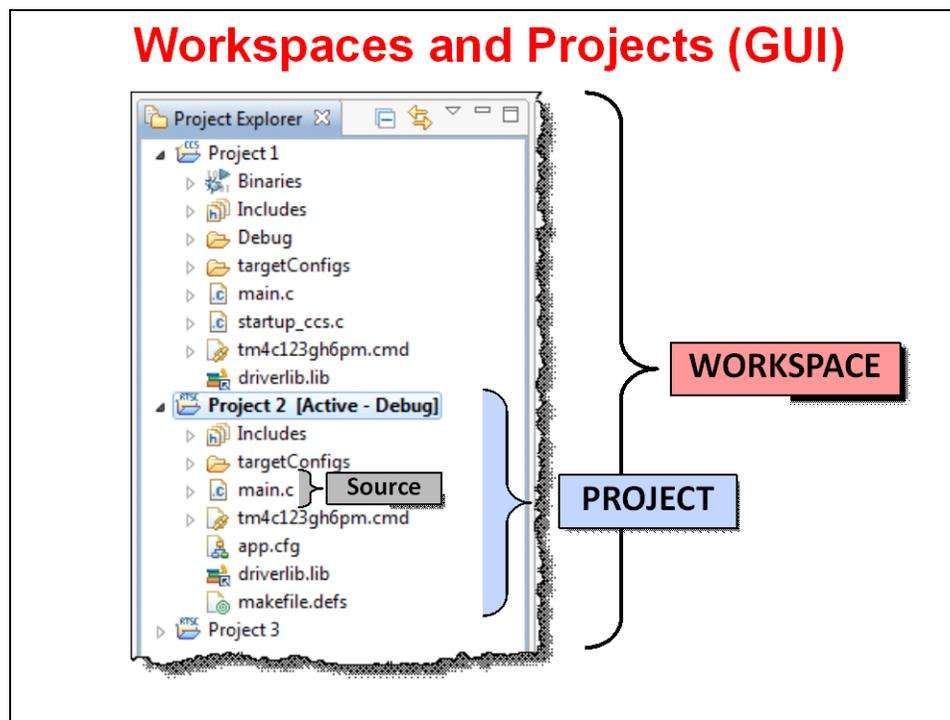
Eclipse even varies the toolbars and menus between perspectives.



Workspaces & Projects

Eclipse based IDE's provide a hierarchy for storing program information. Experienced programmers are familiar with the concept of keeping all their programs source files in a **Project**.

Eclipse goes one step further and also defines a **Workspace**. In fact, whenever you open CCS (or any Eclipse IDE) you are asked to select a *workspace*. In essence, a *Workspace* is just the folder in which your projects reside. In the CCS/Eclipse, you can actually think of the *Project Explorer* window as a visual representation of your *Workspace*.



Every active project in your *workspace* will be displayed in the *Project Explorer* window, whether the project happens to be open or closed.

Some users like to only put only one project per *workspace*; others put every *project* into a single *workspace* – it doesn't matter to Eclipse.

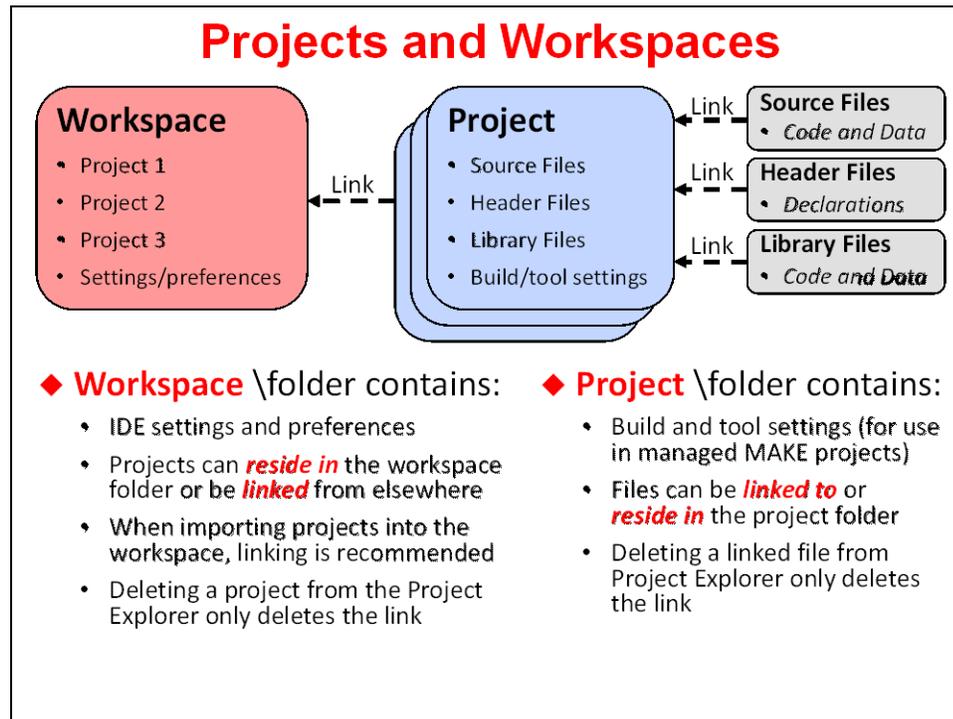
In our workshop, we have chosen to create one *workspace* which will hold all of our lab files. This makes it easy to switch back and forth between exercises, if you should want to do so.

As a final note, this hierarchy reflects how many settings are handled inside of Eclipse. Most settings are modified at the Project level – for example, you can pick the compiler per project.

Some settings, though, can be defined for the whole *Workspace*; for example, you can create path variables to point to library repositories. These almost always can be overridden in a given project, but this means you're not forced to define certain items over-and-over again.

Finally, there are some definitions that are globally setup in the Eclipse/IDE preferences. Unlike pre-Eclipse versions of CCS, they are not stored in the Windows registry. This makes the Linux version of the tools possible; but it also means it's easier to keep multiple versions of CCS on your computer (if you should need to do so).

Let's look at projects & workspaces from another perspective. The following diagram should confirm what we just discussed. **Workspaces** contain **Projects** which contain **Source** files.



Notice how the lines between the various objects are labeled "Link". This represents one way in which they can be connected. Reading the bullets on the above slide tells us that Source files can actually reside "inside" the project folder or be "linked" to the project.

As we'll see in a minute, when you add a file to a project, you have the option of "copying" the file into the project or "linking" it to the project. In other words, you have the option to decide how and where to store your files.

Within Projects, it's most common to see source files reside in the project folder; whereas, libraries are most often linked to the project. This is not a rule, but rather a style adopted by most users.

With regards to Projects and Workspaces: a project folder always resides inside of the workspace. At the very least, this is where Eclipse stores the metadata for each project (in a few different project-related XML files). The remaining project files can reside in a folder outside of the Workspace. Once again, Eclipse provides users with a lot of flexibility in how their files are stored.

Some Final Notes about CCS/Eclipse

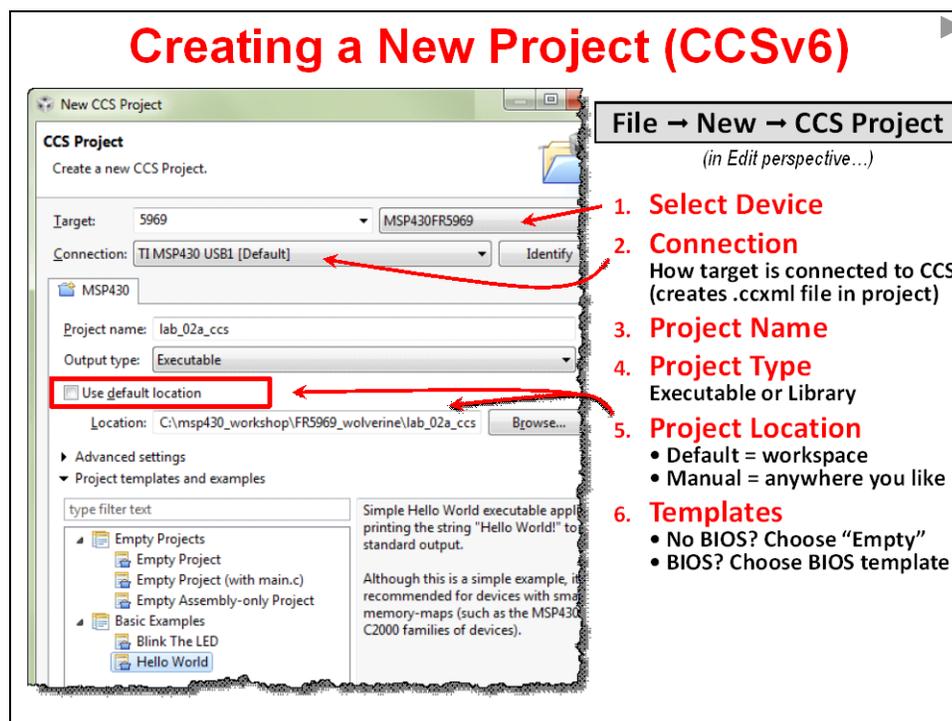
- If you create a new source file in CCS/Eclipse, it will automatically be stored in the project folder.
- If you copy a source file (e.g. C file) into the project folder using the O/S filesystem, it will automatically show up in the project. That is, if you copy a C file into the project folder using Windows explorer, it will be "in the project". Note, though, that CCS does provide a way to "exclude a file from build" – but this is not the default.
- You can export and import projects directly to/from archive (zip) files. Very nice!

Creating a Project

There are many ways to create a new project, the easiest is to select:

File → New → CCS Project

TI defined their own C project type called “CCS Project”. This enhancement condenses the standard Eclipse “new project” wizard from 6 dialogs down to 1. (*Awesome!*)



When creating a new project you need to define:

- *Project Name*
- Are you making an *Executable* program or a *Library*
- Where do you want your project to reside – by default, CCS puts it in the Workspace
- *Processor Family* (i.e. MSP430)
- *Specific device* you’re using
- *Target Connection* (i.e. MSP430 USB 1)
- *Template* – CCS provides a number of project templates. The most common template is probably “Empty”. But some of the others may come in handy. For example, if you are creating a TI-RTOS based project, you will want to choose one of their project templates.

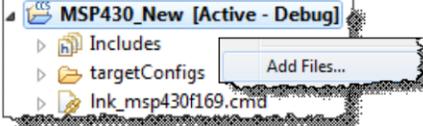
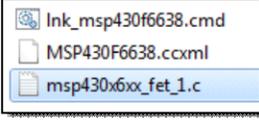
Adding Files to a project

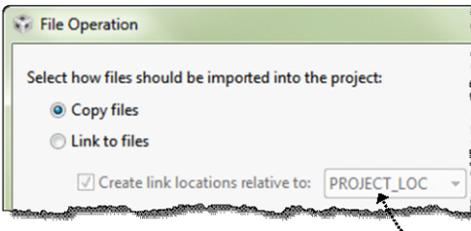
As we described earlier, when adding files to a project, you have the choice of copying them into the project folder or linking them to the project folder.

Copying the files keeps them together inside the project folder. On the other hand, if you're sharing libraries or files between projects (or with other users), it might make more sense to link them.

Adding Files to a Project

- ◆ **Users can ADD (copy or link) files into their project**
 - SOURCE files are typically COPIED
 - LIBRARY files are typically LINKED (referenced)

- ① Right-click on project and select:
 
- ② Select file(s) to add to the project:
 
- ③ Select "Copy" or "Link"



- ◆ **COPY**
 - Copies file from original location to project folder (two copies)
 - ◆ **LINK**
 - References (points to) source file in the *original folder*
 - You can select the "reference" point (default is project's dir)

Portable Projects

This is not an issue for this workshop because the MSP430 team recommends that you add a copy of DriverLib to each project. That said, you will likely run into this issue in the future, so we wanted to bring it to your attention.

The phrase **Portable Projects** signifies that projects can be built in a portable fashion. That is, with a little consideration, it is easy to build projects that can be moved from one user to another – or from one computer environment to another.

When a source file or library is contained inside of a project folder, it is easy for the tools to find and use it. Eclipse automatically knows how to find files inside the project folder.

The biggest headache in moving projects relates to "linked" source files and libraries. When a file is located outside of the project folder, the build will fail unless the person receiving the project user places all the referenced (i.e. linked) files into exactly the same locations inside their filesystem. This is a very common problem!!!

The best solution is to use Eclipse *Path Variables* to point to each directory where you have linked resources. Since this is not a problem encountered in this workshop, we suggest you refer to these locations for more info:

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

You may also want to reference the *Tiva-C Workshop* or the *TI-RTOS Kernel Workshop* for code examples dealing with *Portable Projects*.

Licensing/Pricing

Many users will find that they can use Code Composer Studio free of charge.

For example, there is no charge when using CCS with most of the available TI development boards – with the MSP430, they allow you to use it for free (with any tool), as long as your program is less than 16KB.

Furthermore, TI does not charge for CCS licenses when you are connecting to your target using the low-cost XDS100 JTAG connection.

CCStudio Licensing and Pricing

◆ Licensing

- Wide variety of options (node locked, floating, time based)
- All versions (full, DSK, free tools) use same image
- Updates available online

◆ Pricing

- Reasonable – includes FREE options as noted below
- Annual subscription - \$99 (*\$159 for floating*)



Item	Description	Price	Annual
Platinum Eval Tools	Full tools with 90 day limit (all EMU)	FREE	
Platinum Bundle	XDS100; Simulators; many TI dev'l boards (such as Tiva-C Launchpad); MSP430 when using GNU Compiler	FREE	
Platinum Node Lock	Full tools tied to a machine	\$445*	\$ 99
Platinum Floating	Full tools shared across machines	\$795	\$159

* Download version; \$495 when disc is shipped to you

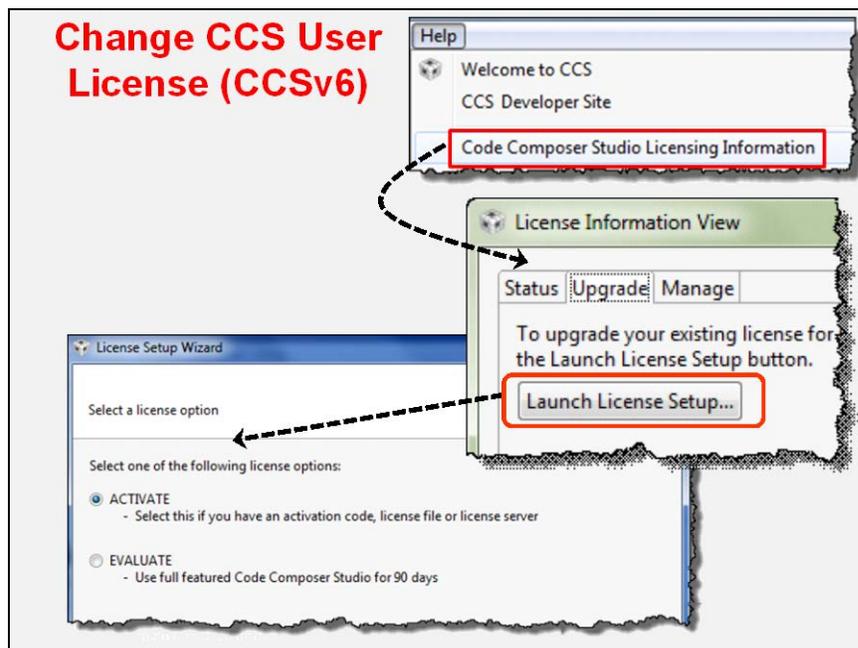
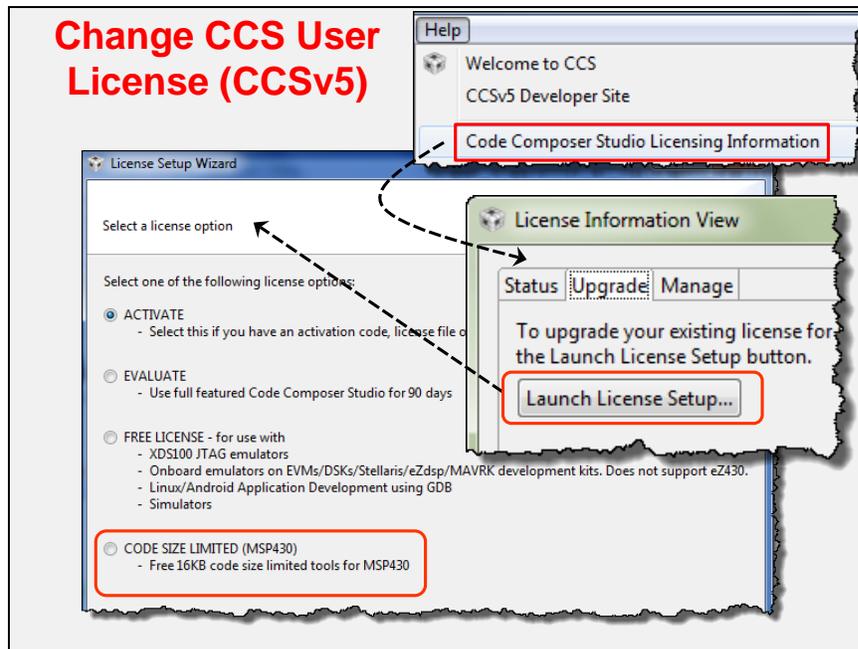
For those cases where you need to use a more sophisticated (i.e. faster) JTAG development connection, TI provides a 90-day free evaluation license. After that, you need to purchase the tool. Thankfully, when you encounter one of these cases, CCS for only costs \$445.

Changing a CCS User Licence

In this workshop, we can use the free license options. For CCSv5 you would choose the “**16K Code Size Limited (MSP430)**” option; you don’t have to do anything for CCSv6, it defaults to the free option.

It is a little bit tricky to change the licensing method. That is, it’s hard to find the following dialog.

As shown, choose *Code Composer Studio Licensing Information* from the *Help* menu. When that dialog appears, choose the *Upgrade* tab, then click the *Launch License Setup...* button.



Writing MSP430 C Code

As part of the prerequisites for the workshop, we stated that you should be familiar with the C language; therefore, in this section we do not plan to cover general C language syntax. Rather, this section is dedicated to implementation details of the MSP430 C Compiler.

Build Config & Options

TI C compilers offer nearly a hundred different build options. We plan to focus on just a few options so that you're aware of the most common ones.

You should find the table below broken into three sets of options:

Processor Options

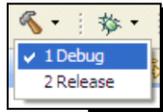
- Early MSP430 CPU's were 16-bit only processors and could only address 64K bytes. Most of the MSP430 devices from the F2xx series onward utilize the MSP430X CPU which contains eXtended 20-bit addressing (up to 1MB).
- The **--silicon_version** option lets you choose which CPU to compile for. CCS chooses this option for you based on the device you select when creating a new project. Nowadays, it's almost always *msp*.
- By default, the **--code_model** option follows the CPU type; therefore, it's most common to see *large* as the common default.
- The **--data_model** defaults to *small*, which constrains data to 64K (addresses to 16-bits); *restricted* means addresses can be 32-bits, but no data objects can be over 64KB; *large* indicates that addresses are 32-bits and there are no restrictions on data objects.

Compiler Build Options

- ◆ Almost 100 compiler options let you tune your code's performance, size, etc.
- ◆ The following table lists the most commonly used options:

	Options	Description		
		16-bit addressing		32-bit addressing
Processor Options	--silicon_version=	msp		msp
	--code_model=	small		large
	--data_model=	small (default)	restricted	large
Debug	-ss	Interlist C statements into assembly listing		
Optimize (Release)	-o3	Invoke optimizer (-o0, -o1, -o2/-o, -o3, -o4)		
	-mf	Speed/code size tradeoff (-mf0 thru -mf5)		
	-k	Keep asm files, but don't interlist		

- ◆ To make things easier, CCS creates two BUILD CONFIGURATIONS:
 - *Debug* (no optimization) which is great for LOGICAL debug
 - *Release* (-o) which is good for PERFORMANCE/Size
 - Users can create their own custom build config's



Debug Options

Until recently, you were required to use the `-g` option when you wanted source-level debugging turned on. The drawback to this option was that it affected the code performance and size. This has changed... since source-level debugging does not affect the optimizer's efficiency, it is always enabled.

On the other hand, if you want to see your C code interlisted with its associated assembly code, then you should use the `-ss` option. Be aware, though, that this does still affect the optimizer – which means that you should turn off this option when you want to minimize the code size and maximize performance such as when building your production code.

Optimize Options (aka Release Options)

We highlight 3 optimization options:

- `-o` turns on the optimizer. In fact, you can enable the optimizer with different levels of aggressiveness; from `-o0` up thru `-o4`. When you get to `-o3`, the compiler is optimizing code across the entire C file. Recently, TI has added the `-o4` level of optimization; this provides link-time optimizations, on top of all those performed in level `-o3`.
- `-mf` lets the compiler know how to tradeoff code size versus speed.
- `-k` does not change the optimizer; rather, it tells the tools to *keep* the assembly file (.asm). By default the asm file is deleted, since it's only an intermediate file. But, it can be handy if you're trying to debug your code and/or want to evaluate how the compiler is interpreting your C code. **Bottom Line:** *When optimizing your code, replace the `-ss` option with the `-k` option!*

Build Configurations

Early in development, most users always use the *Debug* compiler options.

Later in the development cycle, it is common to switch back and forth between *Debug* and *Release* (i.e. optimize) options. It is often important to optimize your code so that it can perform your tasks most efficiently ... and with the smallest code footprint.

Rather than forcing you to continuously tweak options by hand, you can use *Build Configurations*. Think of these as 'groups' of options.

When you create a new project, CCS automatically creates two Build Configurations:

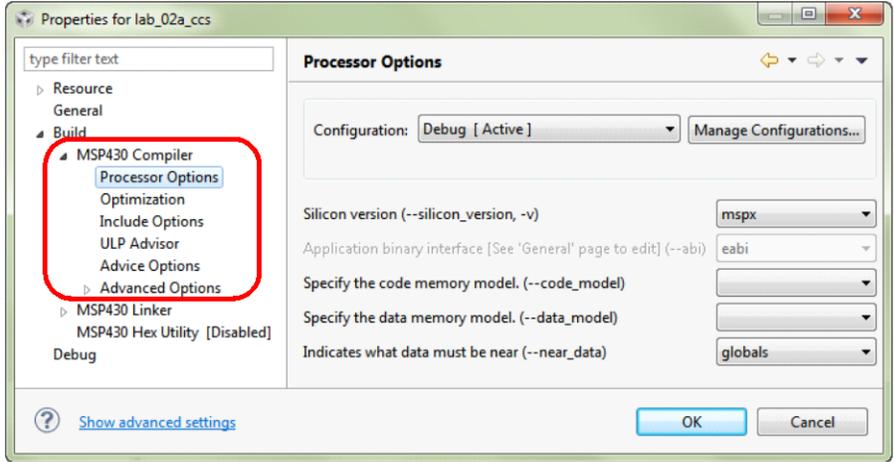
- Debug
- Release

This makes it easy for you to switch back and forth between these two sets of options.

Even further, you can modify each of these option sets ... or create your own.

Modifying Build Configurations

- ◆ Select the build configuration: *Debug or Release*
- ◆ Right-click on the project and select *Properties*
- ◆ Then click "*Processor Options*" or any other category (like *Optimization*):



Hint: If you modify a Project build option, it only affects the active build configuration.

This is a common source of errors. For example, when you add a new library search path to your project options during Debug, it only affects that configuration. This means that it's common to run into errors whenever you switch to the Release build configuration.

CCS is trying to help – and often asks if you want to update both/all configurations. But, this is a new feature and only works for some of the options. This means that when an option should apply to all configurations, you should (manually) change them both at the same time ... or be prepared to tweak the Release build options the first time you use it.

Data Types

The following data types are specified in the C Compiler Users Guide. We've circled the types that best describe this processor.

With the MSP430's ability to perform byte-wide addressing, it follows that *char*'s are 8-bits.

As one might expect, though, being a 16-bit CPU, both the *short* and *int* data types are 16-bits wide.

MSP430 C Data Types (ELF format)

Type	Bits	Representation
char	8	(aligned to 8-bit boundary)
short	16	Binary, 2's complement
int	16	Binary, 2's complement
long	32	Binary, 2's complement
long long	64	Binary, 2's complement
float	32	IEEE 32-bit
double	64	IEEE 64-bit
long double	64	IEEE 64-bit

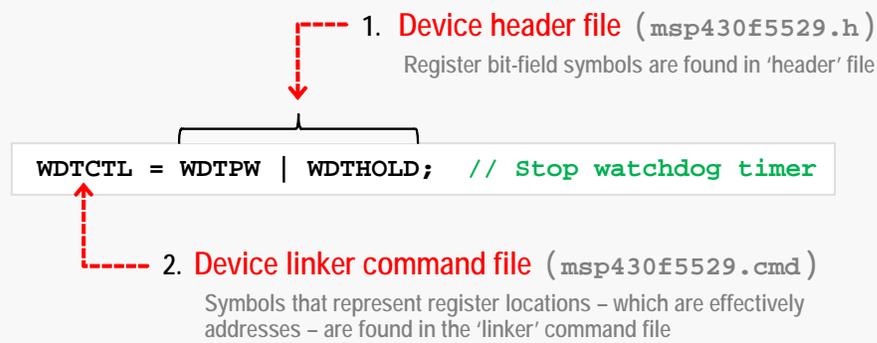
- ◆ Data are aligned to 16-bit address boundary (except where noted)
- ◆ 8-bit values are stored in bits 0-7 of a register
- ◆ 32- and 64-bit types require 2 and 4 registers, respectively

Device Specific Files (.h and .cmd)

TI has created a device-specific header file (.h) and linker command file (.cmd) for each specific MSP430 device. With the MSP430F5529 device as an example, if you look through the files installed with the MSP430 compiler, you'll find: `msp430f5529.h` and `msp430f5529.cmd`

Example: Device Specific 'Header' Files

- ◆ Below is an example of using the MSP430 'header' files.
- ◆ This example will be used in the upcoming lab exercise. It turns off the Watchdog Timer (WDT). We have to setup the WDT in every MSP430 program. (We explain why in Chapter 4 of the workshop.)
- ◆ Notice how "address" values (i.e. register locations) are found in the .cmd file, while all other symbol definitions are found in the .h file.



As described in the above diagram, these two files provide symbolic definitions for all registers and bitfields found in each CPU and its peripherals.

What's the simple key to figure out which file contains a given symbol?

- If the symbol relates to an address, such as the symbol for a memory-mapped register (e.g. `WDTCTL`), you'll find it defined in the .CMD file. This is because the linker (and its associated linker command file) specifies memory allocations and addresses.
- All the other device-specific symbols are described in the header (.h) file, as is common practice for the C language.

To make programming easier for you, CCS automatically adds these two device-specific files to your project.

- You'll find a linker command file added to your project folder; in fact, it should be listed in the Project Explorer window within your project.
- Most new CCS projects include an "empty" `main.c` file. The header file is `#included` at the top of this file.

Device Specific Files (.h/.cmd)

- ◆ New CCS projects automatically contain two files based upon the "Target CPU" selection:
 1. **Device header file** (e.g. `msp430f5529.h`)
 - Symbols defined for bit fields, reg's, etc.
 - Structs/unions also defined for bit fields, if you prefer
 - You shouldn't have to use hard-coded bit locations, etc.
 - Your code should `#include msp430.h`, this points to the device specific .h file
 2. **Device linker command file** (e.g. `msp430f5529.cmd`)
 - Device specific addresses defined in dev specific .cmd file
 - Creating a new CCS project automatically includes a project .cmd file ... which includes the device specific .cmd file
 - You shouldn't have to ever look up the address of a register
 - Default linker command file in your project points to device specific .cmd file
- ◆ You should use these symbols in your code, rather than specifying hard values/addresses
- ◆ MSP430ware also uses these symbolic definitions; that is, these definitions represent the lowest-level abstraction layer for C code

In the next chapter we introduce the MSP430ware Driver Library. It utilizes these device-specific header (and linker command) files, though it is automatically included by including the Driver Library's own header file `<driverlib.h>`.

MSP430 Compiler Intrinsic Functions

Along with the symbols defined in the device specific header & linker files, it's common to see programmers use the compiler's intrinsic functions. Think of these as functions that are "built-in" to the TI compiler. In most cases, intrinsic functions correlate to hardware specific features found in processors.

Intrinsics for MSP430 C Compiler

- ◆ Compiler intrinsic functions are essentially "built-in" C functions
- ◆ They usually provide access to underlying hardware features of a processor; often mapping closely to specific asm instructions
- ◆ We will use some of these in today's workshop:

```

_bcd_add_short();          _disable_interrupt();    _never_executed();
_bcd_add_long();          _enable_interrupt();     _no_operation();
bic_SR_register();        _even_in_range();       _op_code();
bic_SR_register_on_exit();  _get_interrupt_state();  _set_interrupt_state();
_bis_SR_register();       _get_R4_register();      _set_R4_register();
_bis_SR_register_on_exit(); _get_R5_register();      _set_R5_register();
_data16_read_addr();      _get_SP_register();      _set_SP_register();
_data16_write_addr ();    _get_SR_register();      _swap_bytes();
_data20_read_char();      _get_SR_register_on_exit();
_data20_read_long();      _low_power_mode_0();
_data20_read_short();     _low_power_mode_1();
_data20_write_char();     _low_power_mode_2();
_data20_write_long();     _low_power_mode_3();
_data20_write_short();    _low_power_mode_4();
delay_cycles();           _low_power_mode_off_on_exit();

```

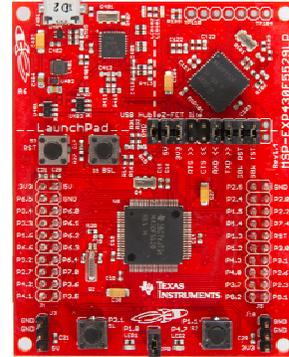
We've circled some of the intrinsic functions we'll use in this class. From setting and/or clearing bits in the Status Register (SR) to putting the processor into low-power modes.

Lab 2 – CCStudio Projects

The objective of this lab is to learn the basic features of Code Composer Studio. In this exercise you will create a new project, build the code, and program the on-chip flash on the MSP430 device.

Lab 2 – Creating CCS Projects

- ◆ **Lab 2a – Hello World**
 - Create a new project
 - Build program, launch debugger, connect to target, and load your program
 - printf() to CCS console
- ◆ **Lab 2b – Blink the LED**
 - Explore basic CCS debug functionality
Restart, Breakpoint, Single-step, Run-to-line
- ◆ **Lab 2c – Restore Demo to Flash**
 - Import CCS project (for original demo)
 - Load program to device's flash memory
 - Verify original demo program works
- ◆ **(Optional) Lab 2d**
 - Create binary TXT file of your program
 - Use MSP430 Flasher to program original demo's binary file to device's flash



Time: 45 minutes

Lab Outline

Programming C with CCS	2-25
<i>Lab 2 – CCStudio Projects.....</i>	<i>2-27</i>
Lab 2a – Creating a New CCS Project	2-29
Intro to Workshop Files	2-29
Start Code Composer Studio and Open a Workspace	2-30
00430 ... Licensed to Develop	2-31
“CCS Edit” Perspective	2-32
Create a New Project	2-33
Build The Code (ignore advice).....	2-36
Debug The Code	2-37
Fix Your Project.....	2-41
Build, Load, Connect and Run ... with the Easy Button	2-42
Lab 2b – My First Blinky.....	2-43
Create and Examine Project	2-43
Build, Load, Run.....	2-44
Restart, Single-Step, Run To Line	2-45
Lab 2c – Putting the OOB back into your device	2-47
(Optional) Lab 2d – MSP430Flasher	2-49
Programming the UE OOB demo using MSP430Flasher.....	2-49
Programming Blinky with MSP430Flasher.....	2-52
Cleanup.....	2-53

Lab 2a – Creating a New CCS Project

In this lab, you create a new CCS project that contains one source file – `hello.c` – which prints “Hello World” to the CCS console window.

The purpose of this lab is to practice creating projects and getting to know the look and feel of CCSv5. If you already have experience with CCSv5 (or the Eclipse) IDE, this lab will be a quick review. The workshop labs start out very basic, but over time, they’ll get a bit more challenging and will contain less “hand holding” instructions.

Hint: In a *real-world* MSP430 program, you would **NOT want to call `printf()`**. This function is slow, requires a great deal of program and data memory, and sucks power – all bad things for any embedded application. (Real-world programs tend to replace `printf()` by sending data to a terminal via the serial port.)

We’re using this function since it’s the common starting point when working with a new processor. Part B of this lab, along with the next chapter, finds us programming what is commonly called, the “embedded” version of “hello world”. This involves blinking an LED on the target board.

Intro to Workshop Files

1. Find the workshop lab folder.

Using Windows Explorer, locate the following folder. In this folder, you will find at least two folders – aptly named for the two launchpads this workshop covers – `F5529_USB`, `FR5969_Wolverine`:

```
C:\msp430_workshop\F5529_USB
C:\msp430_workshop\FR5969_Wolverine
```

Click on YOUR specific target’s folder. Underneath, you’ll find many subfolders

```
C:\msp430_workshop\F5529_USB\lab_02a_ccs
C:\msp430_workshop\F5529_USB\lab_02b_blink
...
C:\msp430_workshop\F5529_USB\solutions
C:\msp430_workshop\F5529_USB\workspace
```

From this point, we will usually refer to the path using the generic `<target>` so that we can refer to whichever target board you may happen to be working with.

e.g. `C:\msp430_workshop\<target>\lab_02a_ccs`

So, when the instructions say “navigate to the Lab2 folder”, this assumes you are in the tree related to YOUR specific target.

Finally, you will usually work within each of the `lab_` folders but if you get stuck, you may opt to import – or examine – a lab’s archived (`.zip`) solution files. These are found in the `\solutions` directory.

Hint: This lab does not contain any “starter” files, rather, we’ll create everything from scratch.

In future labs, though, there may be files already present in the lab folder. If this is the case, we will also include an archive (`_starter.zip`) in case you ever need to refer back to an original file.

Start Code Composer Studio and Open a Workspace

Note: CCS5.5 or v6 should already be installed; if not please refer to the workshop installation guide.

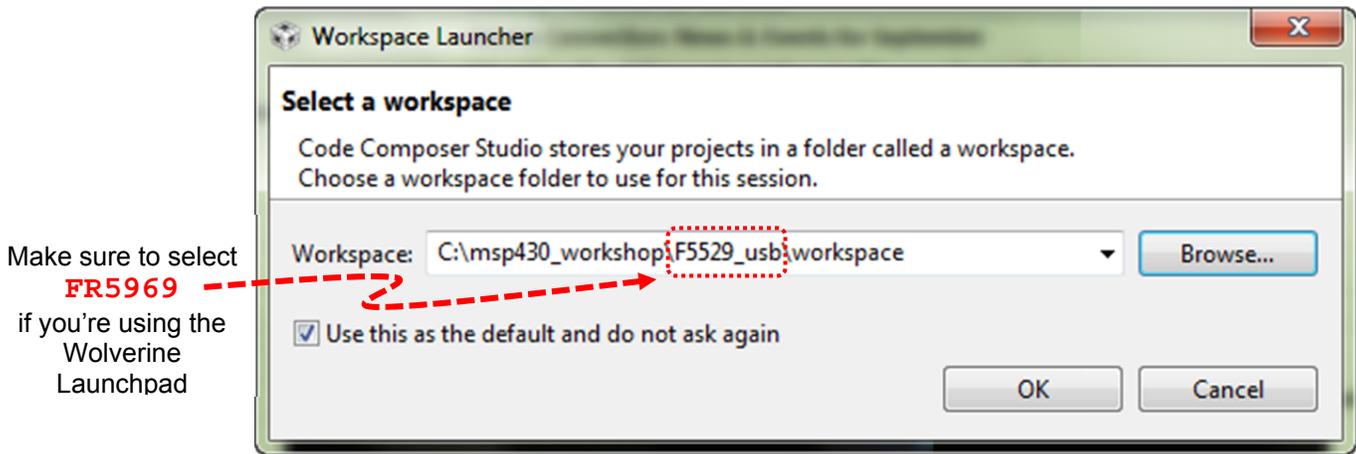
2. Start Code Composer Studio (CCS).

Double clicking the CCStudio icon on the desktop or selecting it from the Windows Start menu.

3. Select a *Workspace* – don't use the default workspace location !!

When CCS starts, a dialog box will prompt you for the location of a workspace folder. We suggest that you select the workspace folder provided in our workshop labs folder. (*This will help your experience to match our lab instructions.*)

Select: C:\msp430_workshop\<<target>\workspace



Most importantly, the workspace provides a location to store your projects ... or links to your projects. In addition to this, the workspace folder also contains many CCS preferences, such as: perspectives, views, and IDE variables. The workspace is saved automatically when CCS is closed.

Hint: If you check the “Use this as the default...” option, you won't be asked to choose a workspace every time you open CCS. At some point, if you need to change the workspace – or create a new one – you can do this from the menu: File → Switch Workspace

4. Click OK (to close workspace dialog). View, then close, *TI Resource Explorer*.

When CCS opens to a new workspace, the *TI Resource Explorer* window is automatically opened and you're greeted with:

Welcome to Code Composer Studio

This *Explorer* is a handy way for you explore the CCSv5 features, such as: examples, libraries (i.e. MSP430ware), and tools, such as Grace™. In this workshop, we'll use many of these features, but we won't necessarily access them from here. Once you close this window, you can always reopen it via: Help → Welcome to CCS

Go ahead and close the *TI Resource Explorer* tab

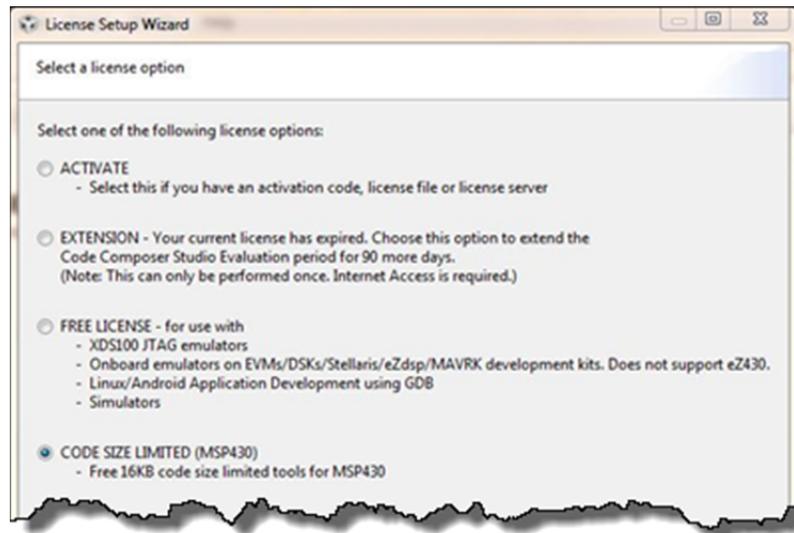
00430 ... Licensed to Develop

5. Set CCSv5 license ... if required. (Not required for CCSv6 as it defaults to the free license.)

The first time CCS opens, the “License Setup Wizard” should appear. In case you need to change the license option, you can open the wizard by clicking:

Help → Code Composer Studio Licensing Information

Then click the Upgrade tab and the Launch License Setup...



If you have a full CCS license, please use that, otherwise we recommend that you select the option:

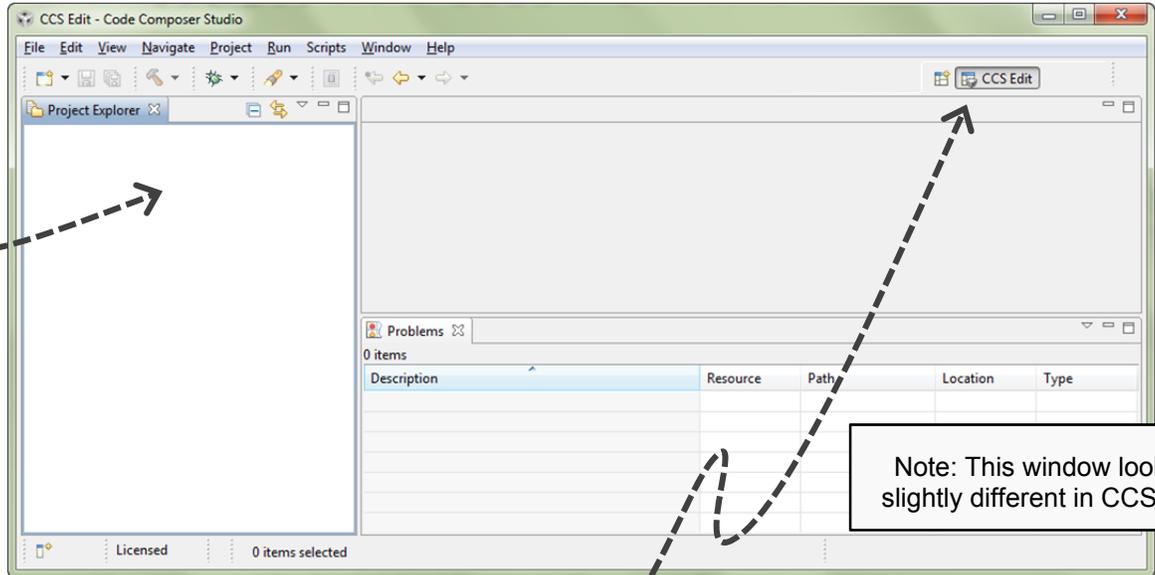
CODE SIZE LIMITED (MSP430)

Hint: If you are attending another workshop in conjunction with this one, like the Tiva-C ARM Cortex-M4F LaunchPad workshop, you can return here and change this to the FREE LICENSE option.

“CCS Edit” Perspective

6. At this point you should see an empty CCS workbench.

The term *workbench* refers to the desktop development environment.



Notice Project Explorer is empty – this matches our empty Workspace folder

Note: This window looks slightly different in CCSv6.

The workbench will open in the “CCS Edit” view.

Maximize CCS to fill your screen

Notice the tab in the upper right-hand corner...

Perspectives define the window layout views of the workbench, toolbars, and menus – as appropriate for a specific type of activity (i.e. editing or debugging). This minimizes clutter of the user interface.

- The “CCS Edit” perspective is used to when creating, editing and building C/C++ projects.
- CCS automatically switches to the “CCS Debug” perspective when a debug session is started.

You can customize the perspectives and save as many as you like.

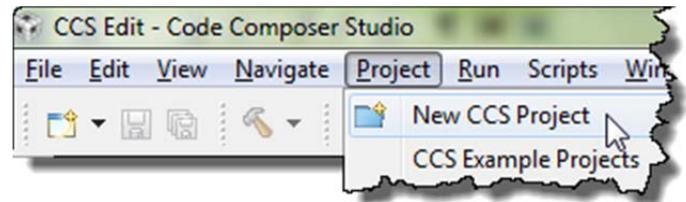
Hint: Most of us find the `Window → Reset Perspective...` handy for those times when we’ve messed our windows up a bit too much.

Create a New Project

7. Select New CCS Project from the menu.

A *project* contains all the files you will need to develop an executable output file (.out) which can be run on the MSP430 hardware. To create a new project click:

File → New → CCS Project



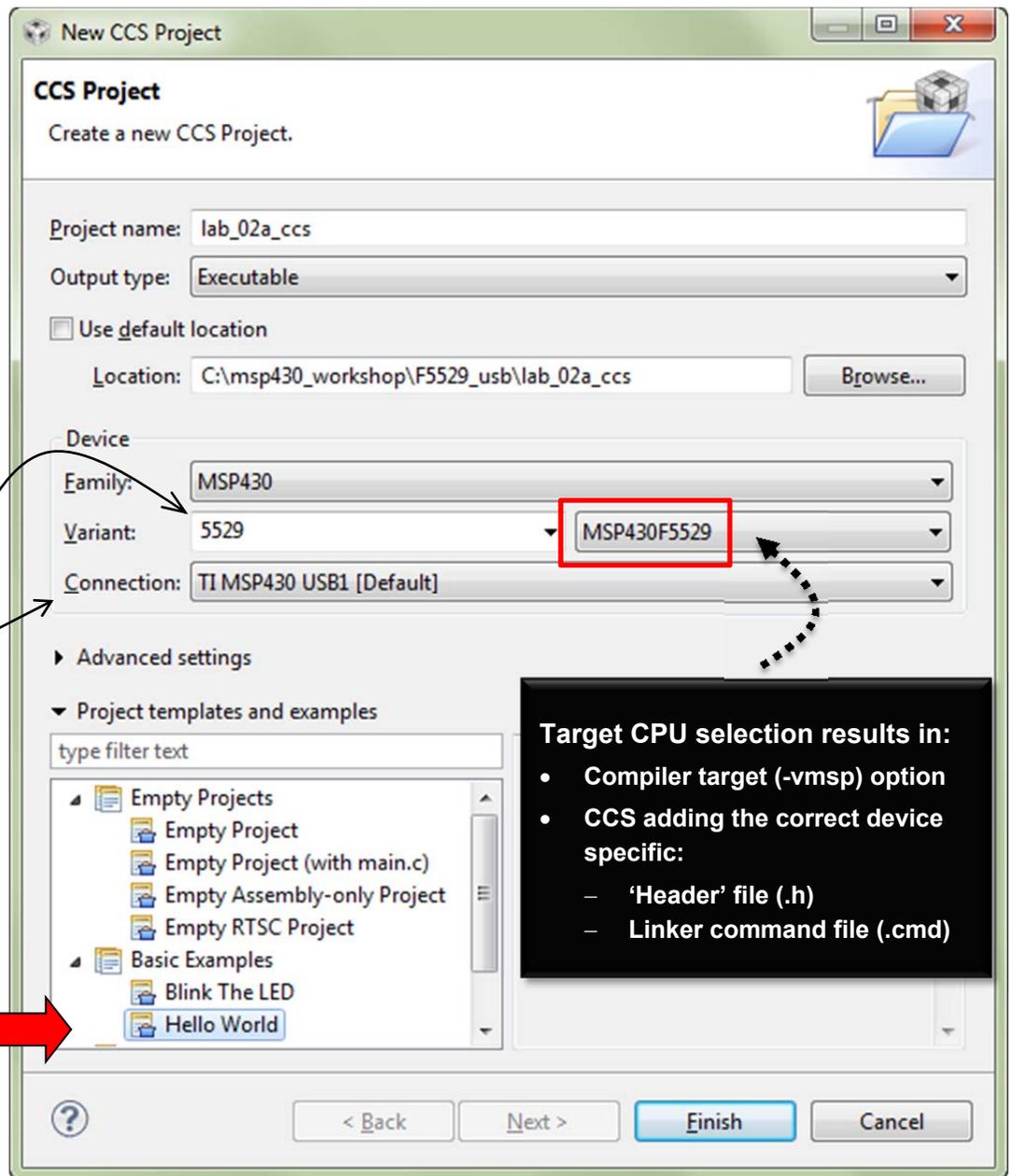
8. Make project choices as shown here:

Note: Your dialog may look slightly different than this one. This is how it looked for CCSv5.5 (build 61).

- a) Name: lab_02a_ccs
- b) **Executable**
- c) Don't use default loc'n
- d) Choose your *target's* lab_02a_ccs folder

- e) Pick **MSP430** family
- f) Type "5529" or "5969" into *variant* to quickly select **Target CPU**
- g) Use *Default* debugger connection (*this creates the .csxml file for you*)

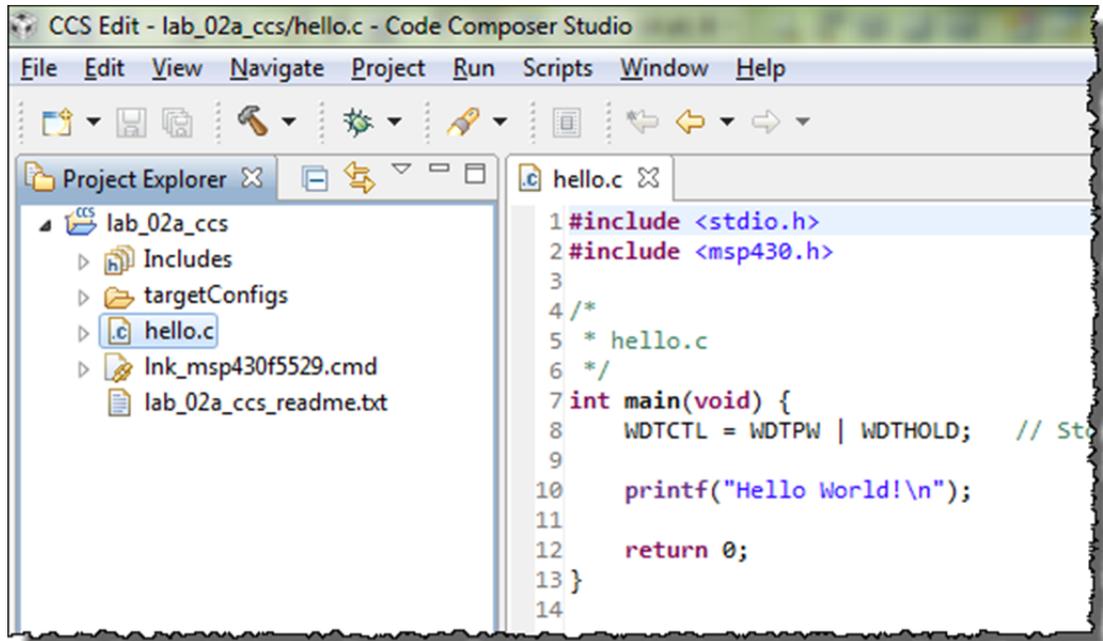
- h) Select template: Hello World



9. Code Composer will add the named project to your workspace.

View the project in the Project Explorer pane.

Click on the ▸ left of the project name to expand the project



CCS includes other items based upon the **Template** selection. These might include source files, libraries, etc.

When choosing the *Hello World* template, CCS adds the file hello.c to the new project.

10. Open and view lab_02a_ccs_readme.txt.

During installation, we placed the readme file into the project folder.

By default, Eclipse (and thus CCS) adds any file it finds within the project folder to the project. This is why the readme text file shows up in project explorer. Go ahead and open it up:

Double-click on lab_02a_ccs_readme.txt

You should see a description of this lab similar to the abstract found in these lab directions.

Hint: Be aware of this Eclipse feature. If – say in Windows Explorer – you absent-mindedly add a C source file to your project folder, it will become part of your program the next time you build.

If you want a file in the project folder, but not in your program, you can exclude files from build:

Right-click on the file → Exclude from Build

11. Explore source code in `hello.c`.

Open the file, if it's not already open.

Double-click on `hello.c` in the Project Explorer window

We hope most of this code is self-explanatory. Except for one line, it's all standard C code:

```
#include <stdio.h>
#include <msp430.h>

/*
 * hello.c
 */
int main(void) {
    WDTCTL = WDTPW | WDTHOLD;    // Stop watchdog timer
    printf("Hello World!\n");
    return 0;
}
```

The only MSP430-specific line is the same one we examined in the chapter discussion:

```
WDTCTL = WDTPW | WDTHOLD;    // Stop watchdog timer
```

As the comment indicates, this turns off the watchdog timer (WDT peripheral). As we'll learn in Chapter 4, the WDT peripheral is always turned on (by default) in MSP430 devices. If we don't turn it off, it will reset the system – which is not what we usually want during development (especially during 'hello world').

Build The Code (ignore advice)

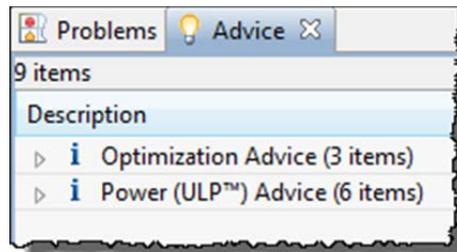
12. Build your project using “the hammer” and check for errors.

At this point, it is a good time to build your code to check for any errors before moving on.

Just click the “hammer” icon:



It should build without any *Problems*, although you should see two sets of Advice: Optimization Advice (new to CCSv5.5) and Power (ULP™) Advice.

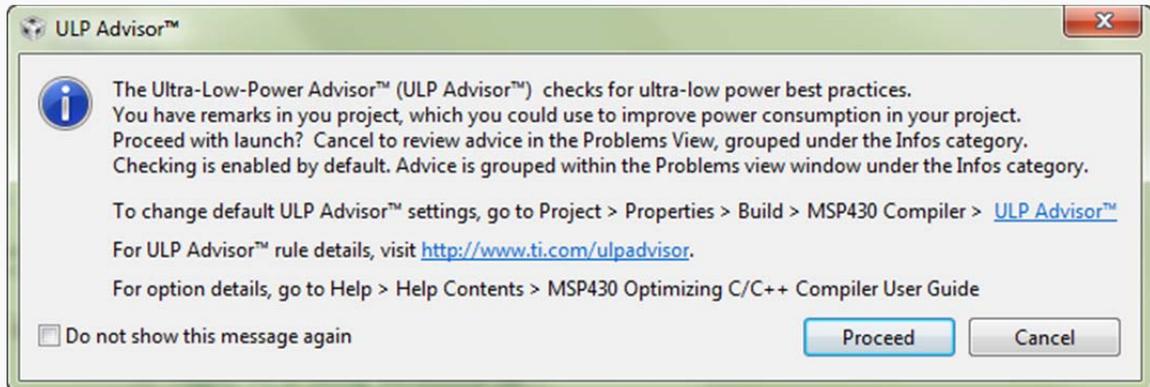


At this point, we’re just going to ignore their advice. It’s better to get code running first. Later, we return and investigate some of these items further.

If the program builds successfully, move to the next page to begin debugging. If you have problems getting it to build, please ask a neighbor, or your instructor for help.

Sidenote: ULP Advisor

Sometime, when you launch the debugger (as we will soon), CCS will warn you that your code could be better optimized for lower power.



While we like the ULP Advisor tool, this usually comes up a long time before we are ready to start optimizing our performance. We recommend that you click the box:

Do not show this message again

As the dialog above indicates, you can always go into your project’s properties and enable or disable this advice. We will do this in a later chapter, when we’re ready to focus on driving our every last Nano amp.

Debug The Code

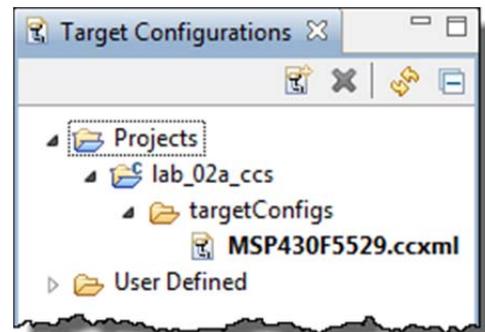
Starting up the debugger is a 3-step process. You could even call it five steps, if you include building and running the code. (In a few minutes, we'll show you a quick shortcut.)

13. Launch a debug session.

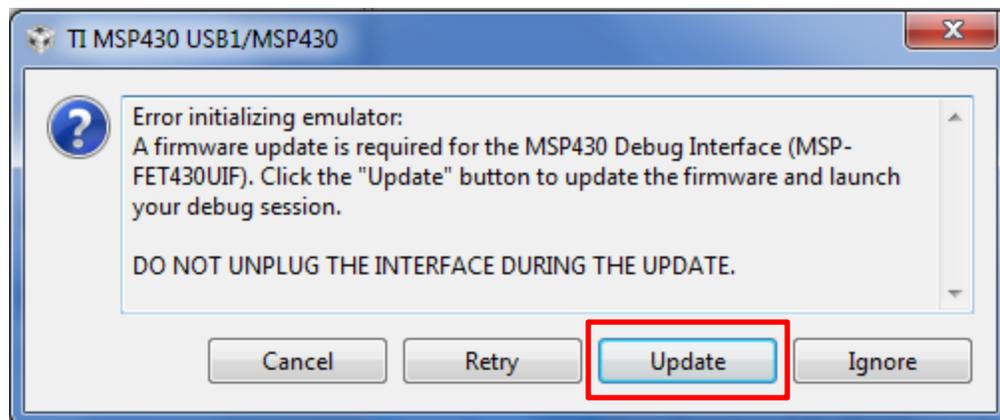
This starts the *CCS Debugger* and then switches to the *Debug* perspective.

- a) Open Target Configurations window
 - View → Target Configurations
- b) Expand hierarchy until you can see your project's .ccxml file
- c) Launch .ccxml file

Right-click .ccxml file → Launch Selected Configuration



Note: The first time you Launch a debugger session, you may encounter the following dialog:

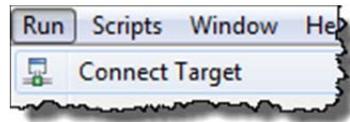


This occurs when CCS finds that the FET firmware – that is, the firmware in your Launchpad's debugger – is out-of-date. We recommend that you choose to update the firmware. Once complete, CCS should finish launching the debugger.

14. Connect to Target.

With your debugger open, you can now connect to your target board.

- Use menu: Run → Connect Target ► Or the *Connect Target* toolbar button:



Connection Problems - Troubleshooting

If the error “*cannot connect to target*” appears, the problem is most likely due to:

- No target configuration (.ccxml) file
- Wrong board/target config file or both – i.e. board does not match the target config file
- Bad USB cable
- Windows USB driver is incorrect – or just didn’t get enumerated correctly

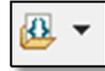
If you run into this, check for each of these possibilities. In the case of the Windows USB driver try:

- Unplugging the USB cable and trying it in a different USB port. (Just changing ports can often get Windows to re-enumerate the device.
- Open Windows Device Manager and verify the board exists and there are no warnings or errors with its driver.
- If all else fails, ask your neighbor (or instructor) for assistance.

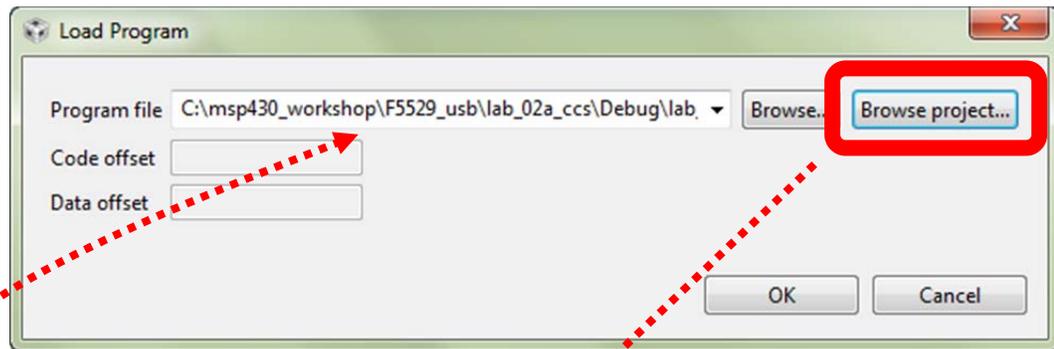
15. Load the code.

We need to load the code to our Launchpad. With this step, CCS actually programs the on-chip Flash memory with your program.

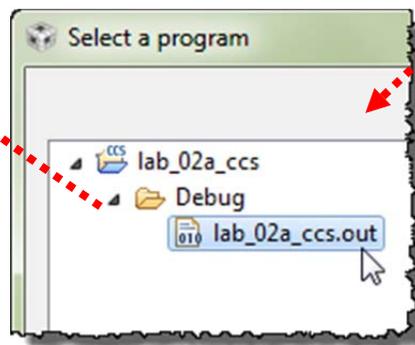
Run → Load → Load Program – or – use the download button:



When the dialog appears, select *Browse Project...*



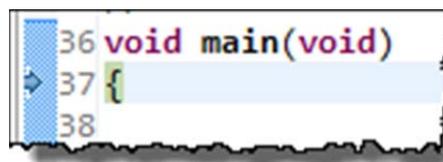
... and navigate to the executable (.out) file in your project:

**Hint:**

Use *Browse Project* to select the .out file.

Often, the default file is NOT the .out file you want. After you have browsed to select it once, it usually provides the correct defaults thereafter.

Your program will now download to the target board and the PC will automatically run until it reaches `main()`, then stop as shown:



16. Run the code.

Now, it's finally time to RUN or "Play". ► Hit the Resume button:



The button is called 'Resume', though we may end up calling it 'Play' since that's what the icon looks like.

17. Pause the code.

To stop your program running, ► click Halt (Pause):

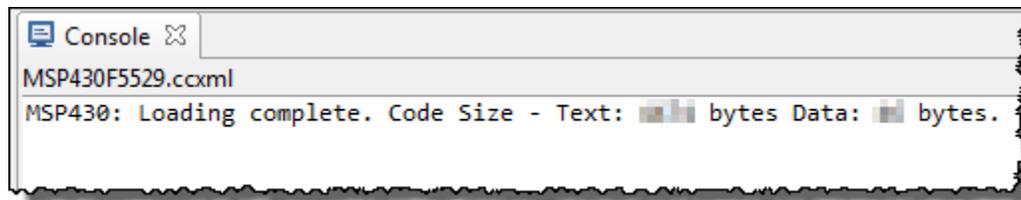


Warning: Pause is different than Terminate !!!

If you click the Terminate button, the debugger – and your connection to the target – will be closed. If you're debugging and just want to view a variable or memory, you will have to open a new debug session all over again. Remember to **pause** and think, before you halting your program.

18. Did printf work?

Did "Hello World!" show up in your console window?



Nope, it didn't show up for us. ☹

19. Let's Terminate the debug session and go fix our project.

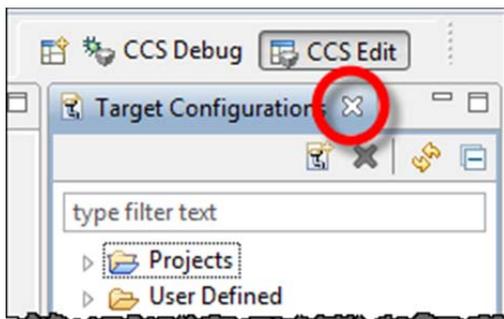
OK, this time we really want to terminate our debug session.

Click the red Terminate button:



This closes the debug session (and Debug Perspective). CCS will switch back to the *Edit* perspective. You are now completely disconnected from the target.

20. Also, if the Target Configurations window is still open, please close it.



Note: Make sure you click the correct "X".

Close the window, don't delete the file!

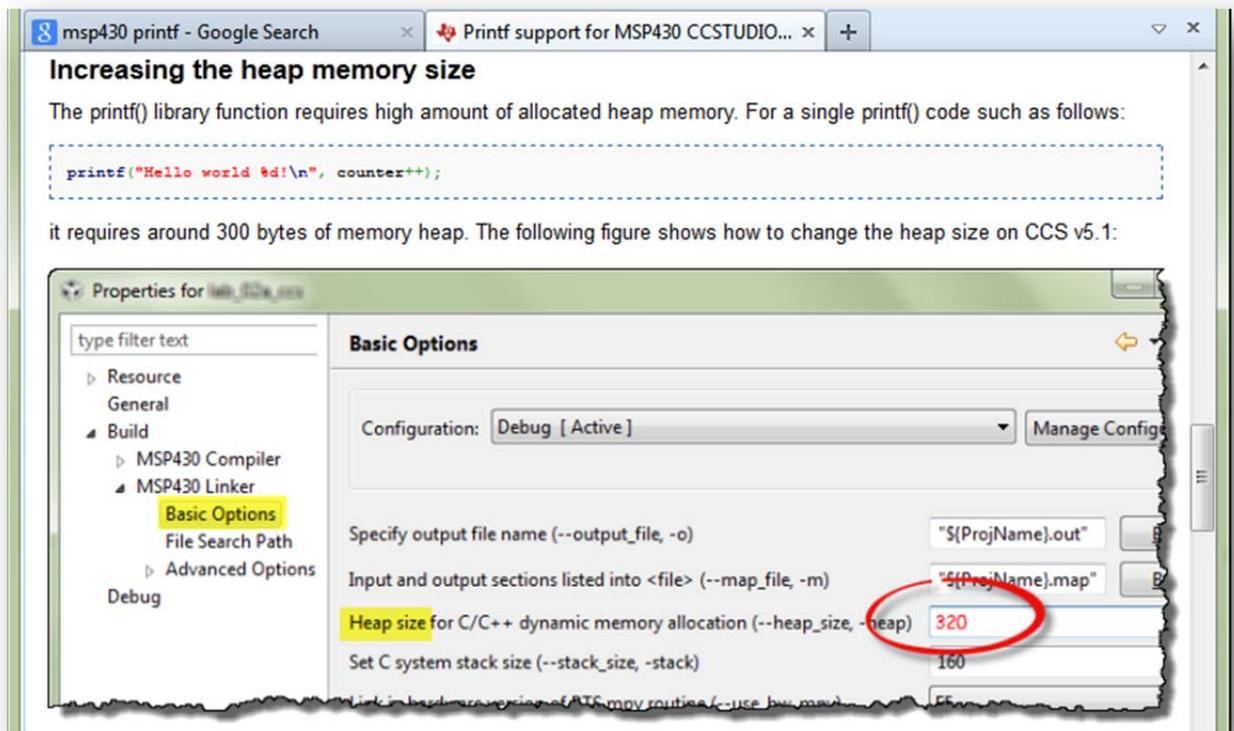
Fix Your Project

21. What is wrong?

We searched the internet for: “msp430 printf” and found a wiki page that demonstrated how to get printf() to work:

http://processors.wiki.ti.com/index.php/Printf_support_for_MSP430_CCSTUDIO_compiler

Since you may not have internet access in the classroom, here's the relevant bit:



22. Increase the heap size.

Per the wiki suggestion, let's increase the heap size to 320 bytes.

Rt-click project → Properties → MSP430 Linker → Basic Options

Increase *Heap size* to: **320**

Hint: As a side note, if you look just below the entry for setting the Heap size, you will see the setting for Stack size. This is where you would change the stack size of you system, if you ever need to do that.

Build, Load, Connect and Run ... with the Easy Button

23. Rebuild and Reload your program – the one-step method.

Here's the “easy button” (i.e. one button) method for debugging your code. First, make sure you terminated your previous debug session and you are in the Edit perspective.

Click the BUG toolbar button:



Clicking this button will: Build the program (if needed); Launch the debugger; Connect to Target; and Load your program



24. Once the program has successfully loaded, ► run it.



25. Close the lab_02a_ccs project.

Terminate the debug session and then close the project. Closing a project is both handy and prevents errors.

Right-click project → Close Project

If your source file (hello.c) was open, notice how closing the project also closes most source files. This can help prevent errors. *(Wait until you've spent an hour editing a file – with it not working – only to find you were editing a file with the same name, but from a different project. Doh!)*

You can quickly reopen the project, when and if you need to.

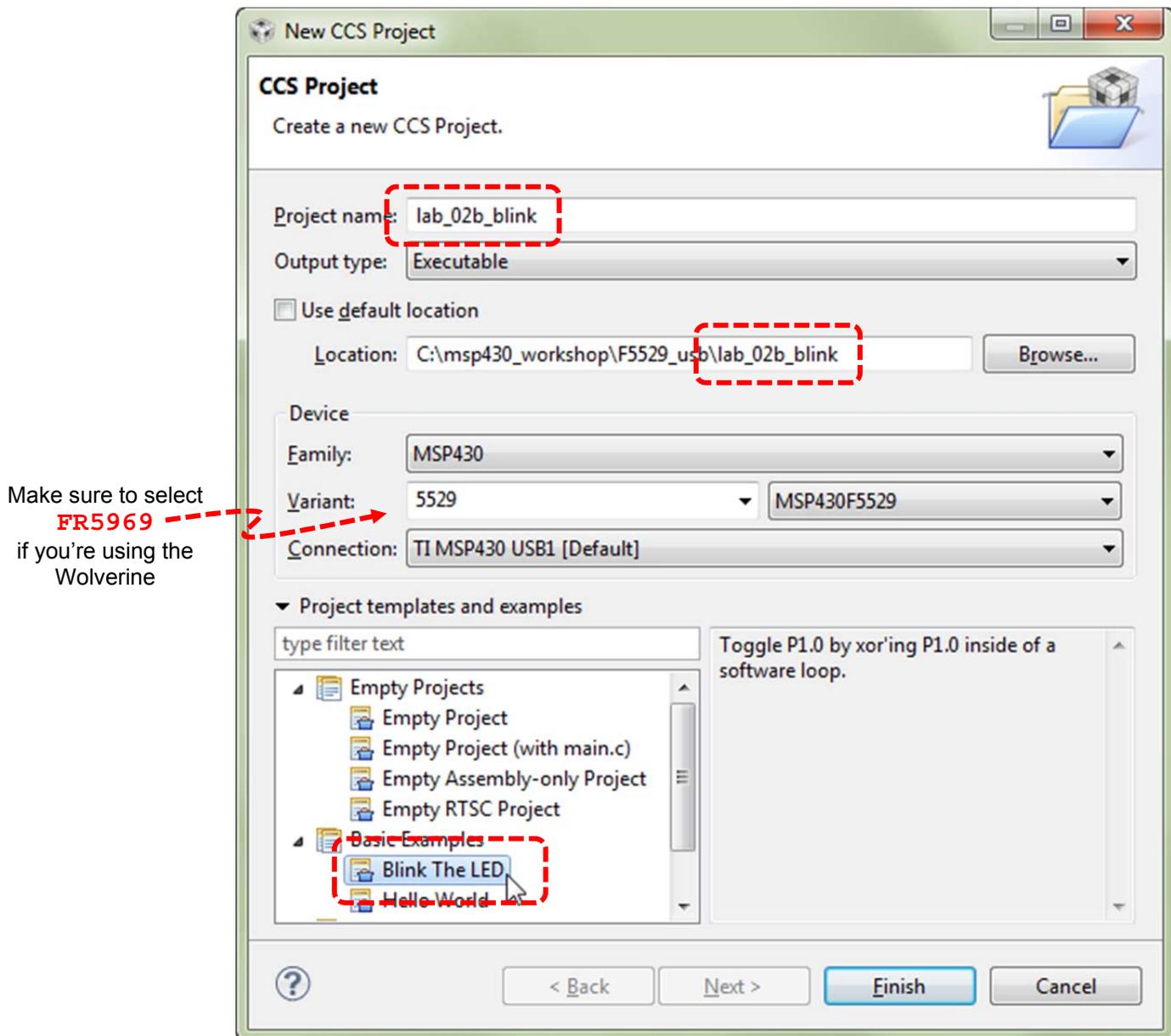
Lab 2b – My First Blinky

We plan to get into all the details of how GPIO (general purpose input/output) works in the next chapter. At that time, we will also introduce the MSP430ware DriverLib library to help you program GPIO, as well as all the other peripherals on the MSP430.

In the lab exercise, we want to teach you a few additional debugging basics – and need some code to work with. To that end, we're going to use the Blink template found in CCS. This is generic, low-level MSP430 code, but it should suite our purposes for now.

Create and Examine Project

1. Create a new project (lab_02b_blink) with the following properties:



2. Let's quickly examine the code that was in the template.

This code simply blinks the LED connected to Port1, Pin0 (often shortened to P1.0).

```
#include <msp430.h>

int main(void) {
    WDCTL = WDTPW | WDT HOLD;    // Stop watchdog timer

    P1DIR |= 0x01;                // Set P1.0 to out-put direction

    for(;;) {
        volatile unsigned int i; // volatile to prevent optimization

        P1OUT ^= 0x01;           // Toggle P1.0 using exclusive-OR

        i = 10000;               // SW Delay
        do i--;
        while(i != 0);
    }
}
```

Other than standard C code which creates an endless loop that repeats every 10,000 counts, there are three MSP430-specific lines of code.

- As we saw earlier, the Watchdog Timer needs to be halted.
- The I/O pin (P1.0) needs to be configured as an output. This is done by writing a “1” to bit 0 of the Port1 direction register (P1DIR).
- Finally, each time thru the for loop, the code toggles the value of the P1.0 pin.
(In this case, it appears the author didn't really care if his LED started in the on or off position; just that it changed each time thru the loop.)

Hint: As we mentioned earlier, we will provide more details about the MSP430 GPIO features, registers, and programming in the next chapter.

Build, Load, Run



3. Build the code. Start the debugger. Load the code.

If you don't remember how to use the easy button (or the long method), please refer back to *lab_02a_ccs*.



4. Let's start by just running the code.

Click the **Run** button on the toolbar (or press **F8**)

You should see the LED toggling on/off.



5. Halt the debugger ... don't terminate!

Restart, Single-Step, Run To Line

6. Restart your program.

Let's get the program counter back to the beginning of our program.

Run → Restart - or - use the Restart toolbar button:



Notice how the arrow, which represents the Program Counter (PC) ends up at main() after your restart your program. This is where your code will start executing next.

In CCS, the default is for execution to stop whenever it reaches the main() routine.

By the way, **Restart** starts running your code from the entry point specified in the executable (.out) file. Most often, this is set to your reset vector. On the other hand, **Reset** will invoke an actual reset. (*Reset will be discussed further in Chapter 4.*)

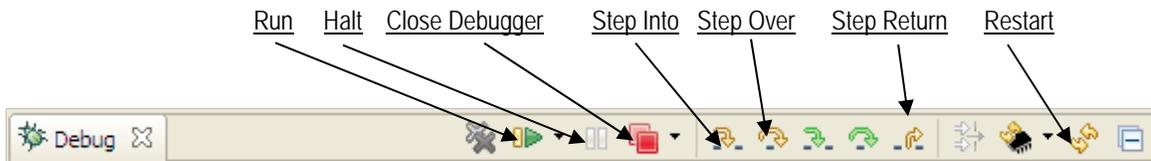
```

21 #include <msp430.h>
22
23 int main(void) {
24     WDTCTL = WDTPW | WDTHOLD;
25     P1DIR |= 0x01;
26

```

7. Single-step your program.

With the program halted, click the **Step Over (F6)** toolbar button (or tap the F6 key):



Notice how one line of code is executed each time you click *Step Over*, in fact, this action treats functions calls as a single point of execution – that is, it steps *over* them. On the other hand *Step Into* will execute a function call step-by-step – go *into* it. Step Return helps to jump back out of any function call you're executing.

Hint: You probably won't see anything happen until you have stepped past the line of code that toggles P1.0.

8. Single-step 10,000 times

Try stepping over-and-over again until the light toggles again...

Hmmm... looking at the count of 10,000; we could be single-stepping for a long time. For this, we have something better...

9. Try the *Run-To-Line* feature.

Click on the line of code that toggles the LED.

Click on the line: `P1OUT ^= 0x01;`

Then Right-click and select **Run To Line** (or hit Ctrl-R)

Single-step once more to toggle the LED

10. Set a breakpoint.

There are many ways to set a breakpoint on a line of code in CCS. You can right-click on a line of code to toggle a Breakpoint. But the easiest is to:

Double-click the blue bar on the line of code

For example, you can see we have just set a breakpoint on our toggle LED line of code:

Once a breakpoint is set, there will be a blue marker that represents it. By **double-clicking** in this location, we can easily add or remove breakpoints.



11. Run to breakpoint.

Run the code again. Notice how it stops at the breakpoint each time the program flow encounters it.

Press F8 (multiple times)

You should see the LED toggling on or off each time you run the code.

12. Terminate your debug session.

When you're done having fun, terminate your debug session.

13. Close the project.

Note: When using beta versions of CCSv6 with the 'FR5969 device, under some circumstances, CCS may corrupt your program in Flash memory if you have more than one breakpoint set. This usually occurs when restarting or resetting your program during debug. The easiest way to visualize this is to view your main() function using the *Disassembly Window*.

The workarounds include:

1. Clear all breakpoints before resetting, restarting or terminating your program.
 2. Load a different program; then load the program that has become corrupted.
-

Lab 2c – Putting the OOB back into your device

Do you want to go back and run the original Out-Of-Box (OOB) demo that came on your Launchpad board?

Unfortunately, we overwrote the Flash memory on our microcontroller as downloaded our code from the previous couple lab exercises. In this part of the lab, we will build and reload the original demo program. Note: sometimes the Out-Of-Box demo is also referred to as the UE (User Experience) demo.

1. Import OOB demo project.

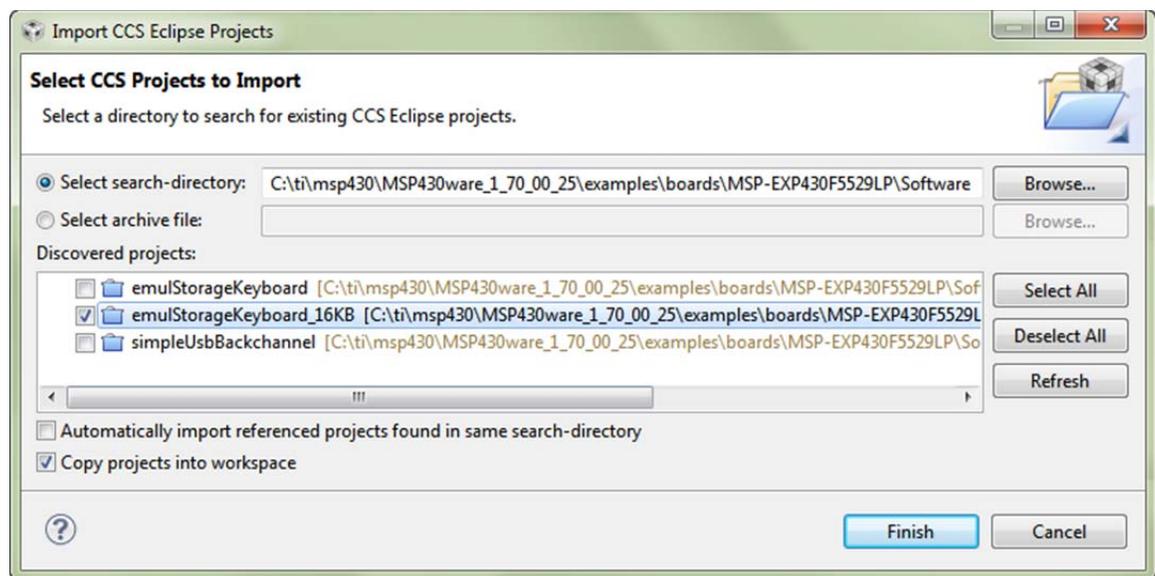
The out-of-box demo can be found in the latest version of MSP430ware.

Project → Import Existing CCS Eclipse Project

F5529

For **'F5529** users, import the project **emulStorageKeyboard_16KB** from the following:

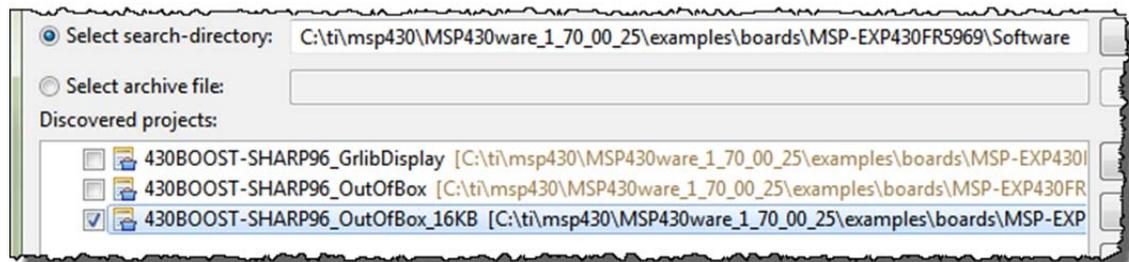
C:\ti\msp430\MSP430ware_1_70_00_28\examples\boards\MSP-EXP430F5529LP\Software\



FR5969

For **'FR5969** users, import the project **430BOOST-SHARP96_OutOfBox_16KB** from:

C:\ti\msp430\MSP430ware_1_70_00_28\examples\boards\MSP-EXP430FR5969\Software\



In both cases, if possible, check “Copy projects into workspace” and then hit the *Finish* button:

Note: For more information on these Launchpad’s, see: ti.com/msp-exp430f5529lp
ti.com/tool/MSP-BNDL-FR5969LCD

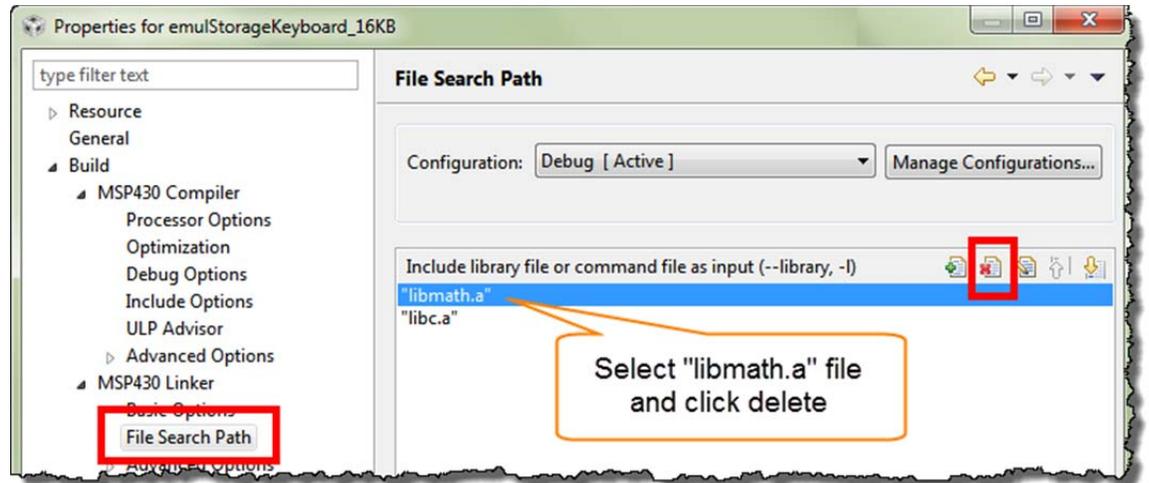
2. Build the out-of-box demo project that you just imported.

Note: If you're using the CCSv6 along with your 'F5529 Launchpad you will most likely get an error stating that CCS cannot find the "libmath.a" library.

F5529

Empirical analysis shows that the project should build and run fine if you just remove this library reference.

Open the project properties: ALT-Enter



3. Click the easy debug button to launch the debugger, and load the program to flash.

Ignore any warnings. In this lab, we're not that interested in running the code within the debugger, rather we're just using the debug button as an easy way to program our device with the demo program. Later labs will explore the various features on display in the demos.

4. Terminate the debugger and close the project. (You can run it within the debugger, but running it outside the debugger 'proves' the program is actually in Flash memory.)

5. Unplug the Launchpad from your PC and plug it back in.

This runs the original demo that was just re-programmed into Flash.
(You can refer back to Lab1 if you have questions.)

(Optional) Lab 2d – MSP430Flasher

The MSP430Flasher utility lets you program a device without the need for Code Composer Studio. It can actually perform quite a few more tasks, but writing binary files to your board is the only feature that we explore in this exercise. The tool is documented at:

http://processors.wiki.ti.com/index.php/MSP430_Flasher_-_Command_Line_Programmer

Note: The MSP430Flasher utility is quite powerful; with that comes the need for caution. With this tool you could – if you are being careless – lock yourself out of the device. This is a feature that is appreciated by many users, but not when doing development. The batch files we provide should not hurt your Launchpad – but we ask that you treat this tool with caution.

Programming the UE OOB demo using MSP430Flasher

1. Verify MSP430Flasher installation.

Where did you install the MSP430Flasher program? Please write down the path here:

_____ /MSP430Flasher.exe

Hint: If you have not installed this executable, either return to the installation guide to do so, or you may skip this lab exercise.

2. Edit / Verify DOS batch program in a text editor.

We created the ue.bat file to allow you to program the User Experience OOB demo to your Launchpad without CCS. Open the following file in a text editor:

```
C:\msp430_workshop\

```

Verify – and modify, if needed – the two directory paths listed in the .bat file. For example:

```
CLS
```

```
C:\ti\MSP430Flasher_1.3.0\MSP430Flasher.exe -n MSP430F5529 -w
"C:\msp430_workshop\F5529_usb\workspace\emulStorageKeyboard_16KB\Debug\emulStorageKeyboard_16KB.txt" -v
```

```
CLS
```

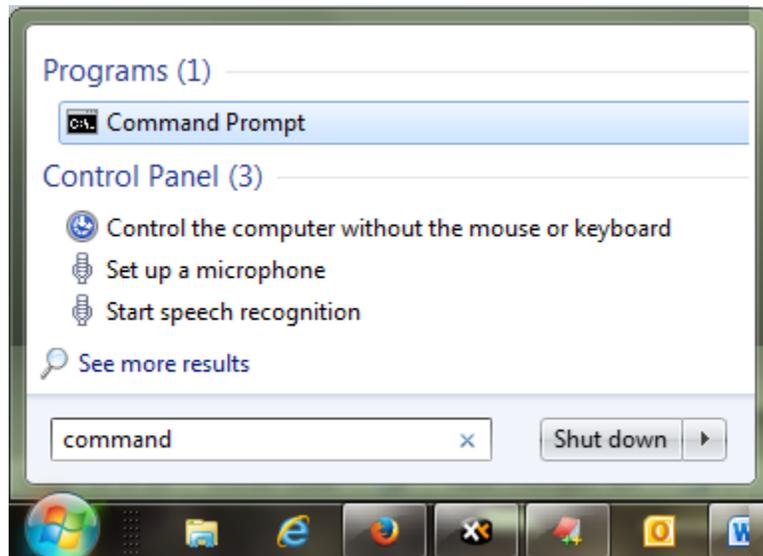
```
C:\ti\MSP430Flasher_1.3.0\MSP430Flasher.exe -n MSP430F5529 -w
C:\msp430_workshop\FR5969_wolverine\workspace\430BOOST-SHARP96_OutOfBox_16KB\Debug\430BOOST-SHARP96_OutOfBox_16KB.txt" -v
```

Where: -n is the name of the processor to be programmed
 -w indicates the binary image
 -v tells the tool to verify the image

We used the default locations for MSP430Flasher and our lab exercises. You will have to change them if you installed these items to other locations on your hard drive.

3. Open up a DOS command window.

One way to do this is by typing “command” in Windows “Start” menu, then hitting Enter.



After starting command, it should open to something similar to this:



4. Navigate to your lab_02d_flasher folder.

The DOS command for changing directories is: “cd”

```
cd C:\msp430_workshop\<target>\lab_02d_flasher\
```

Once there, you should be able to list the directories contents using the *dir* command.

```
dir
```

5. Run the batch file to program the UE out-of-box executable to your board.

```
ue.bat ↵
```

You should see it running ... here's a screen capture we caught mid-programming:

```

C:\msp430_workshop\F5529_usb\lab_02d_flasher>C:\ti\MSP430Flasher_1.2.2\MSP430Flasher.exe -n MSP430F5529 -w "C:\msp430_workshop\F5529_usb\lab_02c_oob\CCS\Debug\MSP-EXP430F5529LP_UE.txt" -v
*-----*
*  /- \  *
*  /- \  *
*  /- \  *
*-----*
* MSP430 Flasher v1.2.2 *
*-----*
* Evaluating triggers...done
* Checking for available FET debuggers:
* Found USB FET @ COM20.
* Initializing interface on TIUSB port...done
* Checking firmware compatibility:
* FET firmware is up to date.
* Reading FW version...done
* Reading HW version...done
* Powering up...done
* Accessing device...done
* Reading device information...done
* Loading file into device...

```

If the information echoed by MSP430Flasher went by too fast on the screen, you can review the log file it created. Just look for the 'log' folder inside the directory where you ran MSP430Flasher.

6. Once again, verify the Launchpad program works.

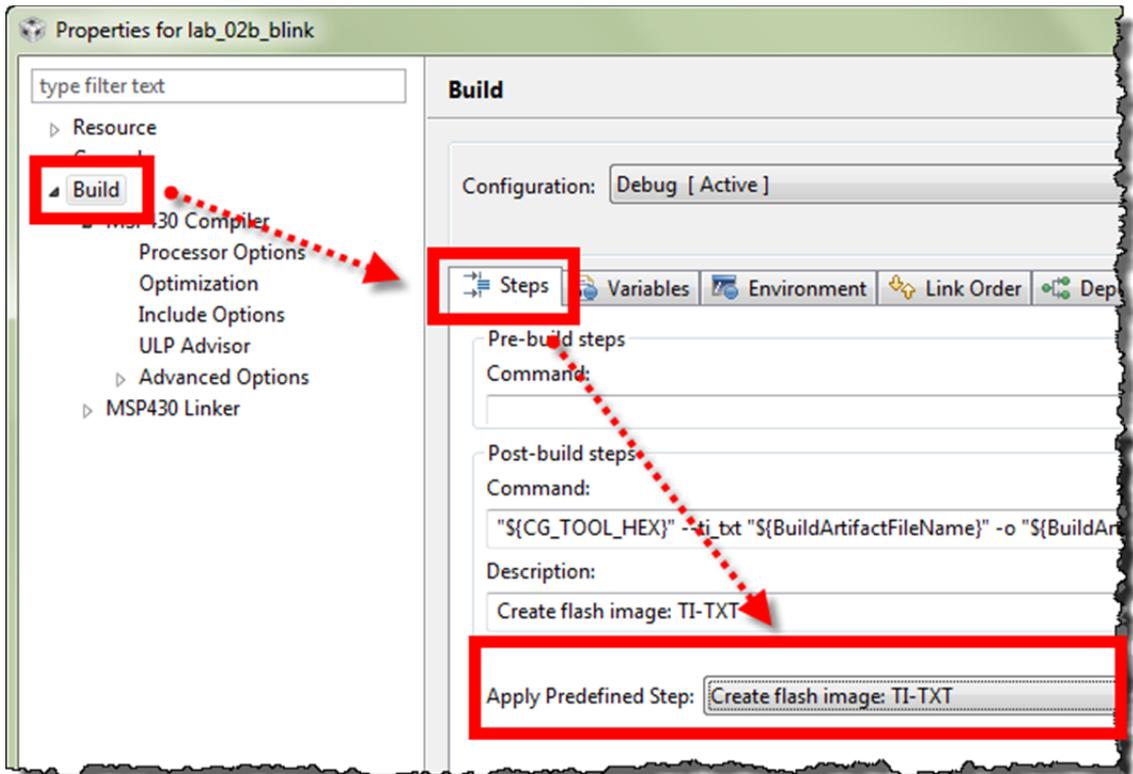
Hint: If you have trouble finding the binary hex file (or in the next section, creating the binary hex file), we created a subdirectory in Lab2c called "local_copy" and placed the two binary files along with their respective .bat files.

Programming Blinky with MSP430Flasher

We can use this same utility to burn other programs to our target. Before we can do that, though, we need to create the binary file of our program. The UE app already did this as part of their build process, but we need to make a quick modification to our project to have it build the correct binary format for the flasher tool.

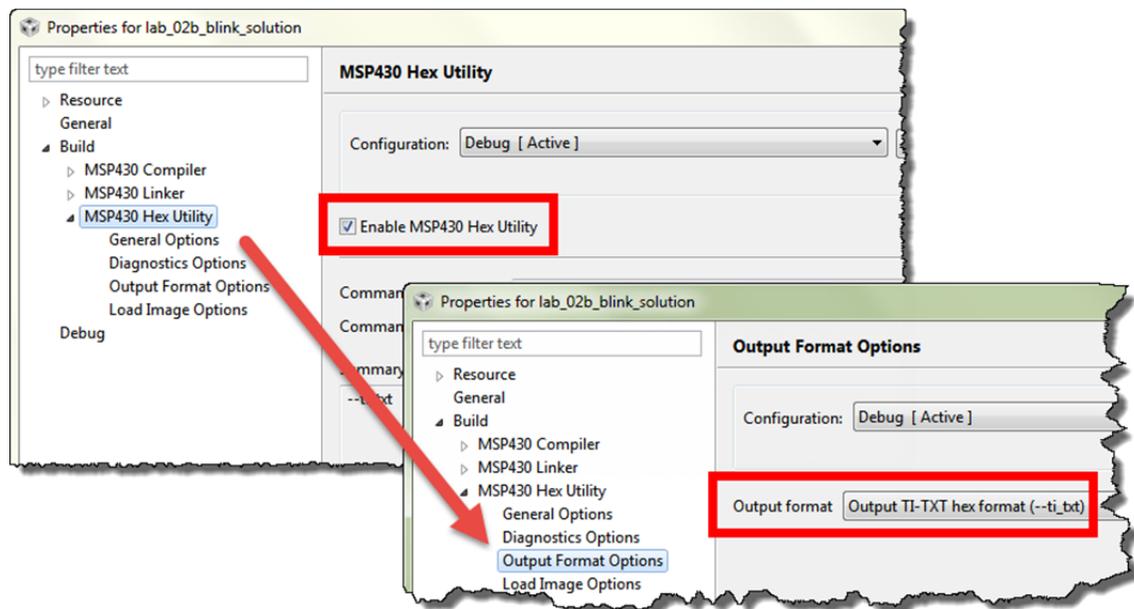
7. **Open your lab_02b_blink project.**
8. **Open the project properties for you project.**
With the project selected, hit *Alt-Enter*.
9. **(CCSv5) Change the required build setting, as shown below.**

(see the CCSv6 steps on the next page)



This is documented at:

http://processors.wiki.ti.com/index.php/Generating_and>Loading_MSP430_Binary_Files

10. (CCSv6) Change the following settings in your project, as shown below:**11. Rebuild the project.**

If you don't rebuild the project, the .txt binary might not be generated if CCS thinks the program is already built.

```
Clean the project
Build the project
```

12. Verify that lab_02b_blink.txt was created in the /Debug directory.

If you're using CCSv6, the method we used to create the binary file causes it to be named:

```
lab_02b_blink.hex
```

13. Open blink.bat with a text editor and verify all the paths are correct.

```
C:\msp430_workshop\

```

Note, you may need to change the name of the file depending on the file extension needed for your program (either .hex or .txt).

14. Run blink.bat from the DOS command window.

When done programming, you should see the LED start blinking.

Cleanup**15. Close your lab_02b_blink project.****16. You can also close the DOS command window, if it's still open.**



Using GPIO with MSP430ware

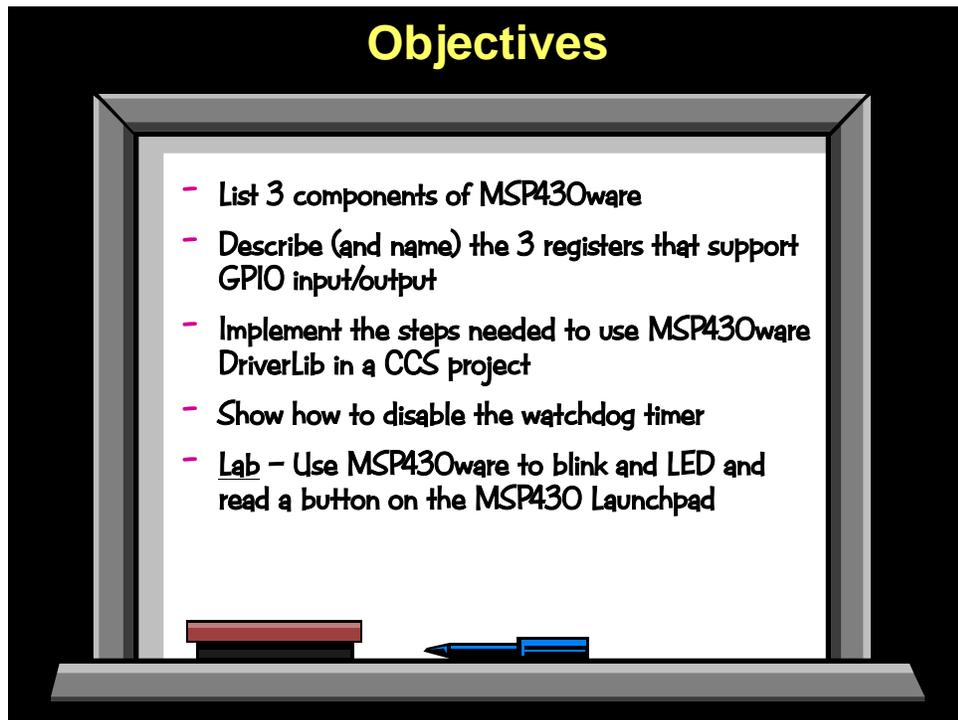
Introduction

In the previous lab exercise, we blinked an LED on the MSP430 Launchpad, but we didn't write the code – we were able to import a generic 'blink' template that ships with CCSstudio.

This chapter explores the GPIO (general purpose bit input/output) features of the MSP430 family. By examining the hardware operation of the I/O pins, as well as the registers that control them, we gain insight into the many ways we can utilize these features.

To make programming easier, we'll use the driver library (DriverLib) component of MSP430ware. While not actually a set of "drivers" in the traditional sense, this library provides us the software tools to quickly build and deploy our own driver code for MSP430 devices.

Learning Objectives

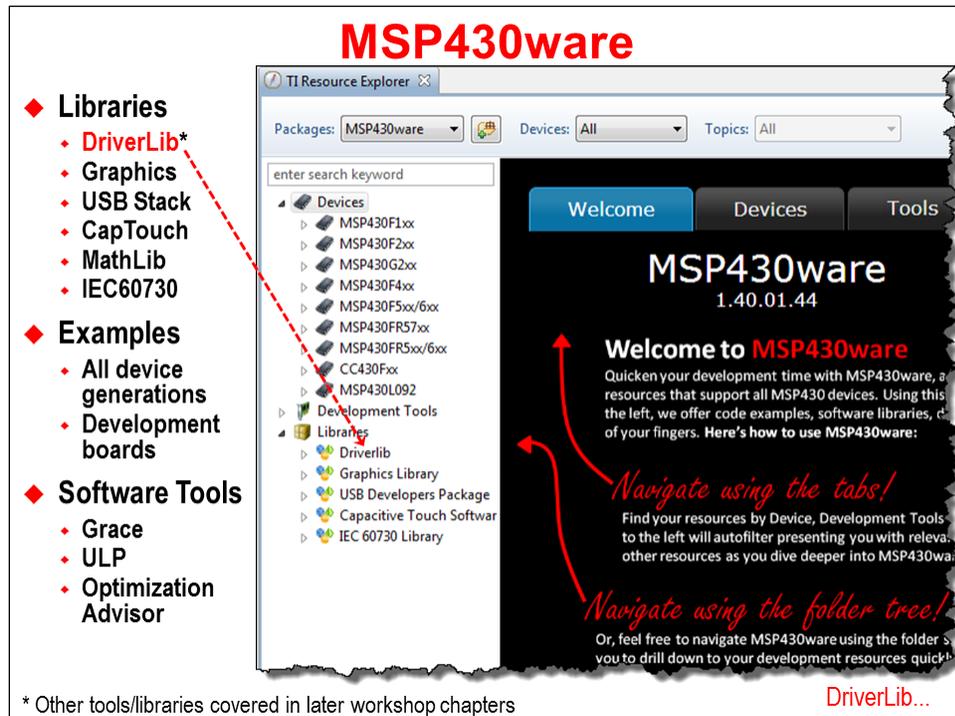


Chapter Topics

Using GPIO with MSP430ware	3-1
<i>MSP430ware (DriverLib)</i>	3-3
Installing MSP430ware	3-3
DriverLib	3-4
DriverLib Modules	3-5
Programming Methods – Summary	3-5
<i>MSP430 GPIO</i>	3-6
GPIO Basics.....	3-6
Input or Output	3-7
GPIO Output	3-8
GPIO Input	3-9
Drive Strength	3-10
Flexible Pin Usage (Muxing)	3-11
Pin Selection	3-12
Port Mapping	3-13
Summary	3-14
<i>Before We Get Started Coding</i>	3-16
1. #Include Files	3-16
2. Disable Watchdog Timer	3-17
3. Pin Unlocking (Wolverine only)	3-18
<i>Lab 3</i>	3-19
Lab 3 Worksheet	3-21
MSP430ware DriverLib	3-21
GPIO Output	3-21
Lab 3a – Blinking an LED.....	3-22
???	3-23
Add MSP430ware DriverLib.....	3-24
Add the Code to <code>main.c</code>	3-26
Debug.....	3-27
Lab 3b – Reading a Push Button	3-29
GPIO Input Worksheet	3-29
File Management	3-30
Add Setup Code (to reference push button)	3-32
Modify Loop.....	3-33
Verify Code.....	3-34
Optional Exercises	3-34
<i>Chapter 3 Appendix</i>	3-35

MSP430ware (DriverLib)

MSP430ware is a bundle of Libraries, Examples and Tools supporting the MSP430 family of microcontrollers. To simplify the installation of all these elements, they have been bundled together into a single (.exe) file.



Installing MSP430ware

When installing Code Composer Studio (CCSv5.5), if you choose to include MSP430 support (or choose to install everything), MSP430ware will be installed by the CCS installer. In this case, you will find MSP430ware along the path:

C:\ti\ccsv5\ccs_base\msp430\msp430ware_1_60_02_09\

The path above assumes that you installed CCS to "c:\ti". Also, the revision numbers (e.g. _1_40_01_44) will depend upon which revision of MSP430ware was available when that CCS installation file was created.

On the other hand, when you install CCSv6 – or download and install MSP430ware directly from the TI website – MSP430ware is installed to a different location. (For example, the installation instructions for this workshop directed you to download MSP430ware directly from the TI website and install it.) When installing MSP430ware in this way, the default path will be slightly different:

C:\ti\msp430\MSP430ware_1_70_00_28\

Note, you can usually get the updated version of MSP430ware by using the auto-update feature of CCS, but we prefer to use the more explicit approach in our workshops since internet access is not always available at workshop locations.

DriverLib

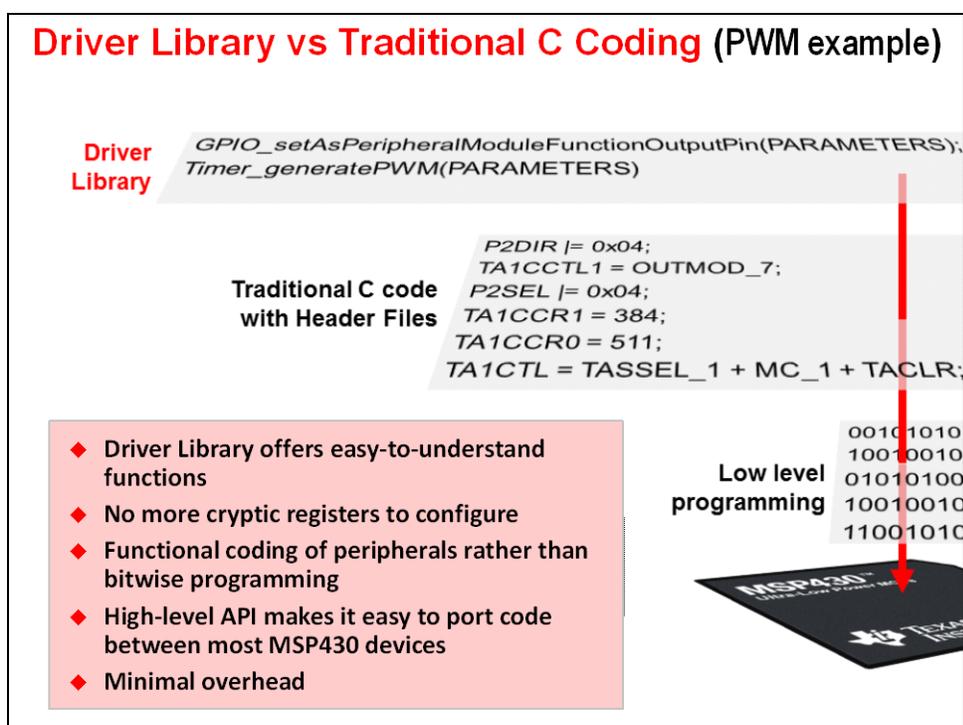
The MSP430ware library used most often in this workshop will be the Driver Library – often called DriverLib.

To quote the DriverLib documentation (we couldn't have said this better ourselves):

The Texas Instruments® MSP430® Peripheral Driver Library is a set of drivers for accessing the peripherals found on the MSP430 5xx/6xx family of microcontrollers. While they are not drivers in the pure operating system sense (that is, they do not have a common interface and do not connect into a global device driver infrastructure), they do provide a mechanism that makes it easy to use the device's peripherals.

While we recommend that you read the entire “Introduction” in the DriverLib users guide (look in the “doc” folder within the DriverLib folder), but this statement does a good job stating the intent of the driver library.

In the following graphic, you can see that the Driver Library provides a convenient way to program the MSP430 peripherals in an easy-to-read (hence easy-to-maintain).



In the previous chapter, we showed you the method of “Traditional C code with Header Files”. In a few rare cases, you might still want to use the Header File symbols; in fact, DriverLib itself utilizes some of these symbols, so they are both compatible with each other.

This said, the convenience of DriverLib's API easily makes it the most desirable method of programming MSP430 peripherals.

On a side note, you might remember a similar diagram (to that above) from the previous chapter. One big difference is that diagram shows an additional, higher-level layer called Energia. Energia (the Arduino port for TI microcontrollers) provides a convenient, portable means of programming MSP430 peripherals – in fact, it's even easier to use than DriverLib. Once again, you can even mix the two programming paradigms. For some, this is a godsend, for others, it's one abstraction layer too much; therefore, most of the chapters in this workshop will focus on DriverLib. Please check out the “Energia” chapter, though, if you're interested in using the Arduino port for rapid prototyping (or production coding).

DriverLib Modules

For the most part, DriverLib is organized by peripheral modules. Each peripheral has its own API specification; this provides good modularity and makes it easy to reuse peripheral code across devices whose peripherals are in common. There are cases where one module may rely on another, but in most cases they are independent API sets.

MSP430ware DriverLib Modules						
Basics & Clocking	Memory	Analog	Power	Timing	Accelerators	I/O
CS	FLASH	ADC10	PMM	TIMER_A	AES	GPIO
USC	FRAM	ADC12	BATT	TIMER_B	CRC	PM
SFR	RAM	SD24	LDO	TIMER_D	DMA	SPI
SYS		COMP		WDT	MPY32	I2C
TLV		REF		RTC		UART
		DAC		TEC		

- Software modules tend to match 1-to-1 with hardware peripherals
- Some of the module names above have been abbreviated
- Not all devices have all hardware (and thus, software) modules
- DriverLib is not currently available for MSP430 ValueLine devices

Used in this chapter

Programming Methods – Summary

Over the past two chapters we have introduced four ways to program the MSP430. They are listed below along with the chapters (and courses) they are discussed in.

Summary	
Name 4 ways to program GPIO:	
1.	<u>Using device specific header & command files (.h/.cmd)</u> Ch2
2.	<u>MSP430ware DriverLib (F5xx/6xx and Wolverine devices)</u> Ch3
3.	<u>Energia</u> Ch8
4.	<u>Grace graphical driverlib tool (Value-line and Wolverine devices)</u> *
*see Chapter 8 in the "G2553 Value-Line Launchpad Workshop"	

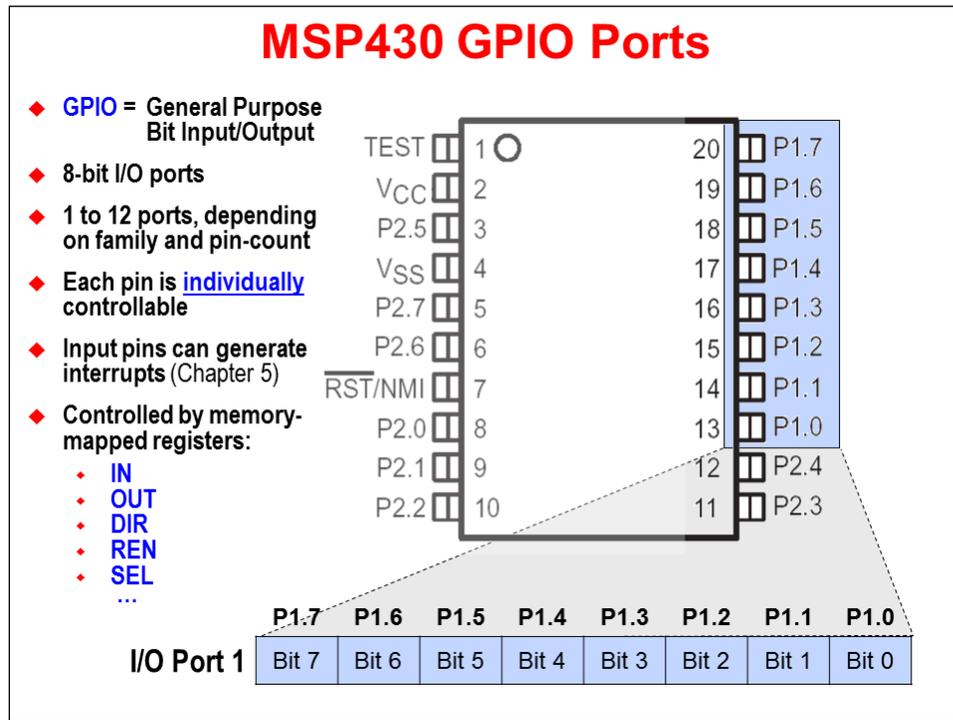
* http://processors.wiki.ti.com/index.php/Getting_Started_with_the_MSP430_LaunchPad_Workshop_v220

MSP430 GPIO

GPIO Basics

General Purpose Bit Input/Output (GPIO) provides a means of controlling – or observing – pin values on the microcontroller. This is the most basic service provided by processors.

The MSP430 provides one or more 8-bit I/O ports. The number of ports is often correlated to the number of pins on the device – more pins, more I/O. The I/O port bits (and their related pins) are enumerated with a Port number, along with the bit/pin number; for example, the first pin of Port 1 is called: P1.0.

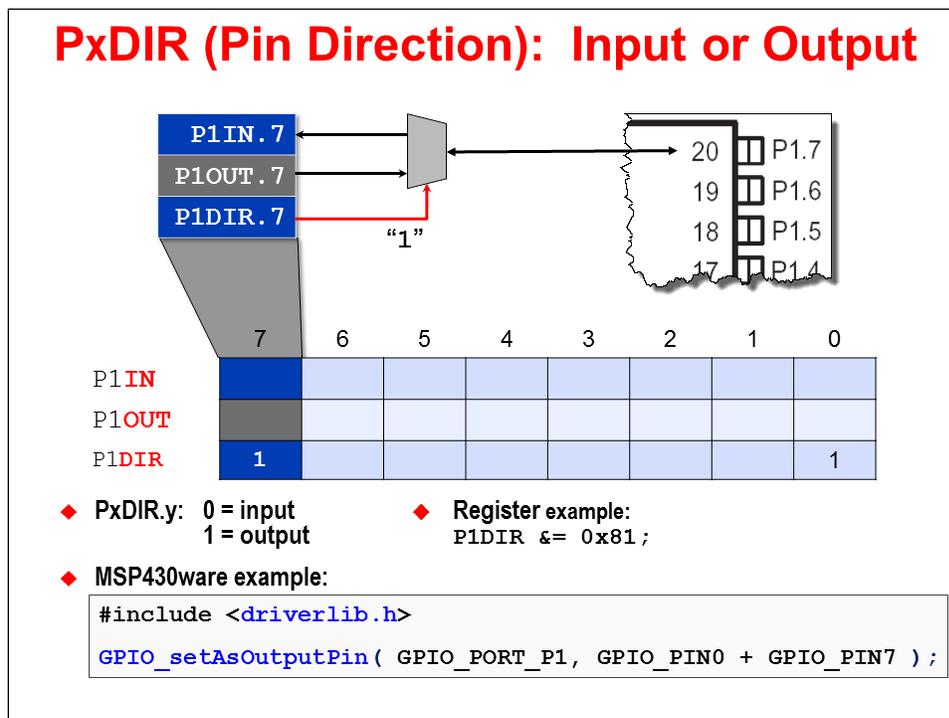


Why did we say pin/bit number? Each I/O pin is individually controllable via CPU registers. For example, if we want to read the input value on P1.0, we can look in bit 0 of the register called P1IN. There are a number of registers used to view and control the I/O pins. In this chapter we'll examine most of them; though, a few – such as those related to interrupts – will be explored in a later chapter.

Note: As mentioned in the previous paragraph, many GPIO pins can be used to trigger interrupts to the CPU. The number of pins that support interrupts depends upon which device you're using. Most devices support interrupts with Ports 1 and 2, but make sure you reference your device's datasheet.

Input or Output

Each GPIO pin is individually controllable – that is, you can configure each pin to be either an input or an output. This is managed via the DIR register; for example, to set P1.7 to be an output you would need to set P1DIR.7 = 1 (as shown below).



Remember that we had multiple programming methodologies? Our graphic above shows us two of them.

- You'll see the "Register example" above uses C code to set the P1DIR register to a given hex value.
- On the other hand, in the "MSP430ware example", the DriverLib function allows you to set one or more pins of a given port as an output. (By the way, to setup the pin as an input, you would use the `GPIO_setAsInputPin()` function.)

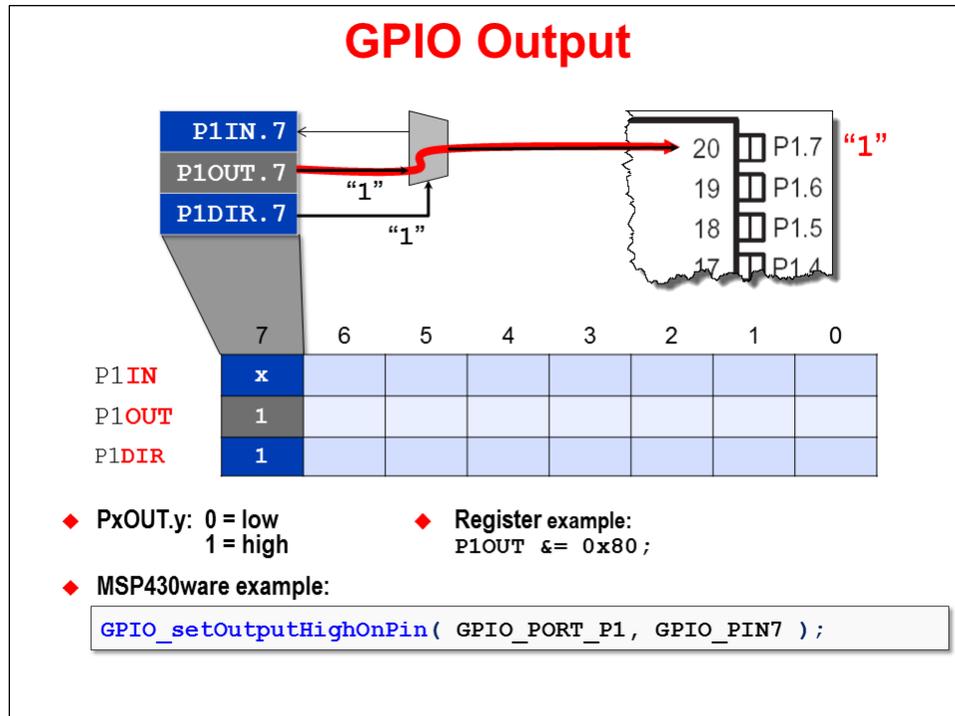
Both methods will end up setting the same registers to the same bit values, though nowadays most teams prefer the more intuitive coding of the DriverLib example. Why? Because you don't really even have to know the register details to understand that pins 0 and 7 are setup as outputs.

Note: As stated earlier in the chapter, the other two programming methods are discussed elsewhere. The Energia method is discussed in its own chapter. Arduino has predefined function names for setting I/O pins. Similarly, the GRACE tool is discussed in its own chapter – which as of this writing is only found in the Value-Line Launchpad version of this workshop.

With the direction configured you will either use the respective IN or OUT register to view or set the pin value (as we'll see on the next couple pages).

GPIO Output

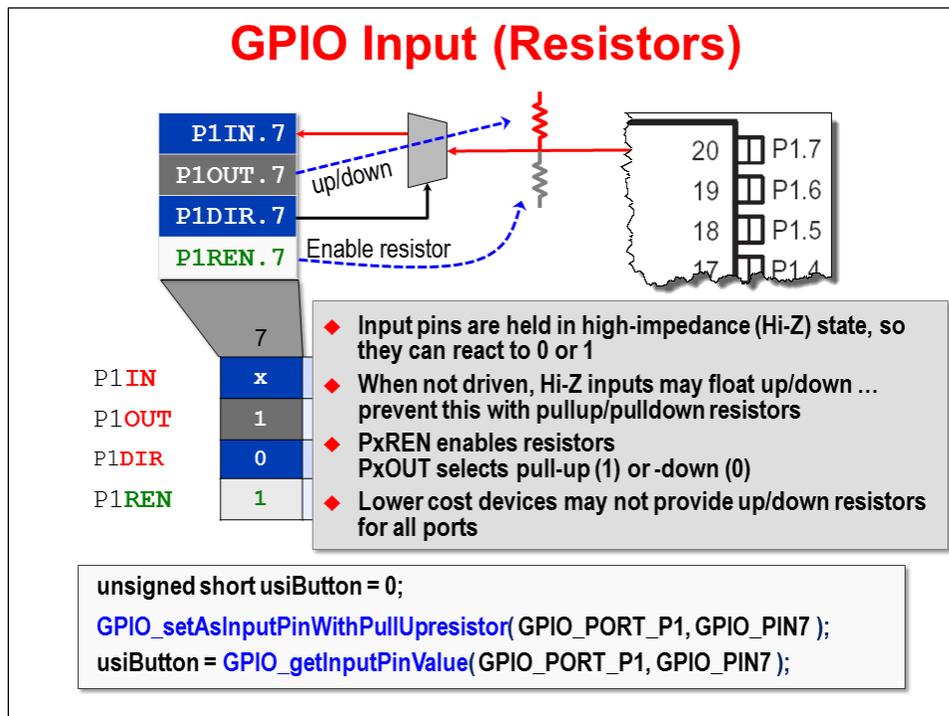
Once you've configured a pin as an output with the PxDIR register, you can set the pins value using the PxOUT register. For P1.7, this would be the P1OUT register.



Once again, the DriverLib `GPIO_setOutputHighOnPin()` or `GPIO_SetOutputLowOnPin()` functions are the easiest way to write to the PxOUT registers. You can set multiple pins/bits by or'ing (+) them together (similar to the P1DIR example on the previous page).

GPIO Input

Reading a pin's value is done by reading the PxIN register. The `GPIO_getInputValue()` DriverLib function returns this value to a variable in your program.



Input pins are slightly more complicated than output pins. While the `PxDIR` function selects whether a pin is used for an input or output, your input pin may need further configuration.

When using a pin as an input, what value does the pin have when it is not being driven by external hardware? Unfortunately, when not being driven, an input pin 'floats' – that is, it can change state arbitrarily. Not only is this undesirable from a logical point of view, but even worse, power is consumed every time the pin changes state. The common solution is to tie the pin high (or low) through a resistor. When driven, the external signal can override the weak pull-up (or pull-down); otherwise the resistor holds the input to a given value.

To minimize system cost and power, most MSP430 I/O ports provide internal pull-up and pull-down resistors. You can enable these resistors via the `PxREN` (Resistor ENable) register bits. When `PxREN` is used to enable these resistors, the associated `PxOUT` bit lets you choose whether the pull-up or pull-down resistor is enabled.

Of course, the easiest way to configure the pull-up or pull-down resistor is to use one of the two GPIO DriverLib functions:

```

GPIO_setAsInputPinWithPullUpresistor()
GPIO_setAsInputPinWithPullDownresistor()

```

Note: Another feature of input pins is their ability to generate CPU interrupts. We won't cover those details in this chapter; rather, we'll save that discussion until the *Interrupts* chapter.

Drive Strength

The F5xx/6xx series of MSP430 devices allow the designer to select whether they want outputs to be driven with lower or higher drive strength. The benefit of this extra feature is that it allows you to tune or power dissipation of your system. You can minimize the extra power usage of outputs when and where it is not needed.

Output Drive Strength (F5xx/6xx only)

	7	6	5	4	3	2	1	0
P1IN	x							
P1OUT	x							
P1DIR	1							
P1REN	x							
P1DS	0/1							

0: Low drive
 1: High drive

- ◆ F5xx (e.g. 'F5529) devices have individually programmable drive strength
- ◆ MSP430ware example:

```

GPIO_setDriveStrength( GPIO_PORT_P1, GPIO_PIN7 );
  
```

Flexible Pin Usage (Muxing)

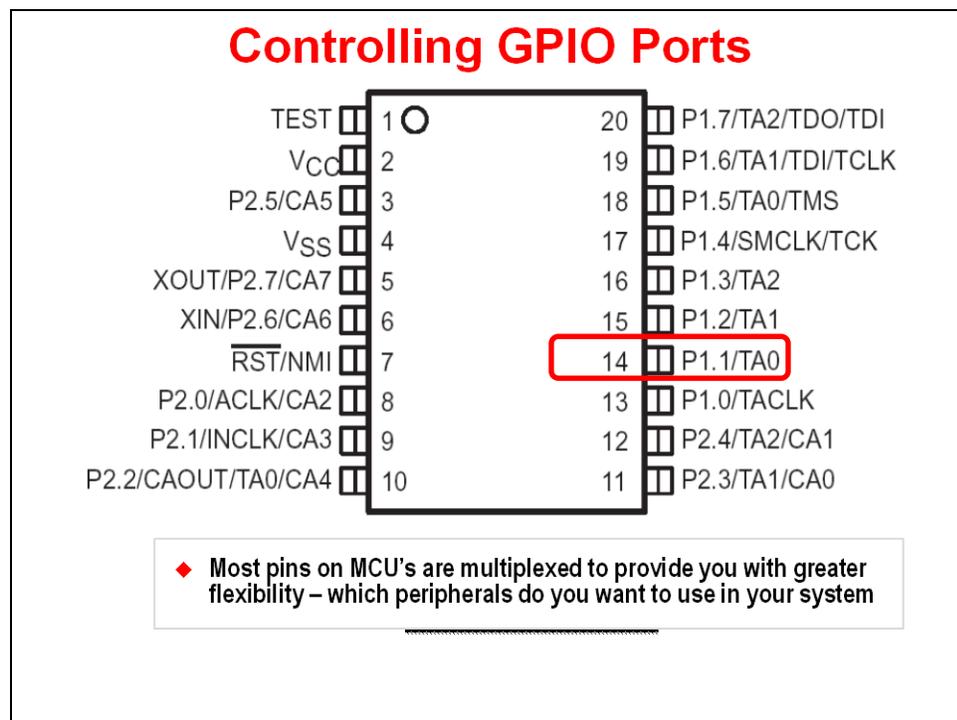
Microcontroller designers have to deal with two conflicting user needs:

More Capability vs Lower Cost

Users want as many features as possible on their processors; the more peripheral options, the better. For example, some users may want 4 serial ports, where others might need 4 I/O ports.

The more pins you add to a device, the greater the cost. (Not only does this make the device more expensive, but it adds to the overall board/system cost.) Therefore, if we added pins for every feature stuffed into our microcontrollers, the cost quickly becomes untenable.

The way this is managed is by 'muxing' different functions onto each pin. In other words, you can select which function you want to use for any given pin on the device. For example, looking at pin 14 in the following diagram, it can be used as either GPIO pin P1.1 or for Timer A0.



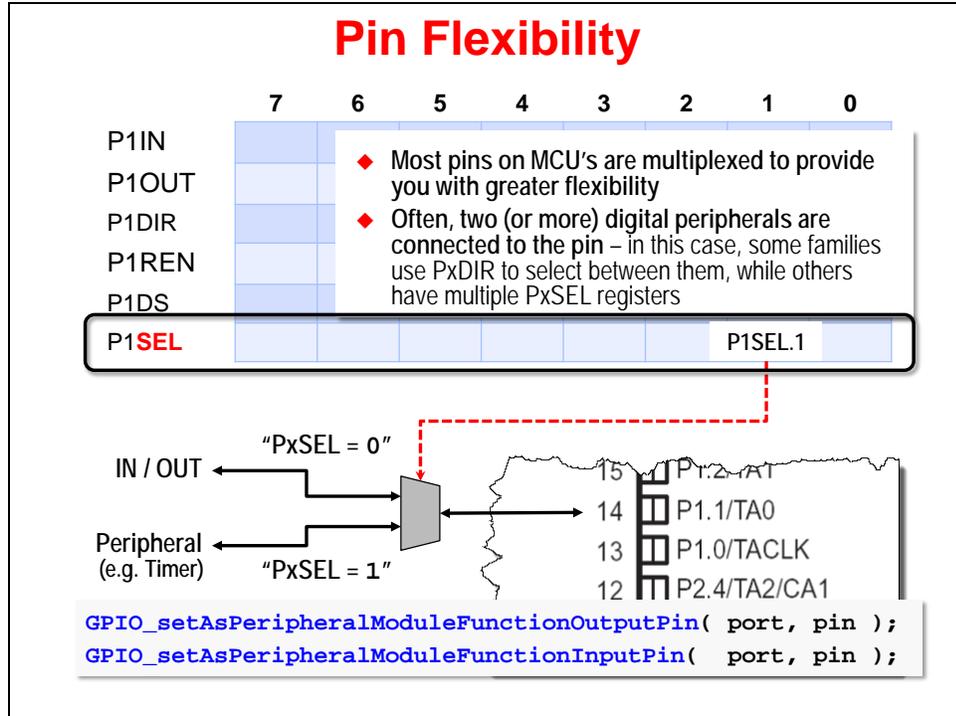
While these pin configurations can be changed at runtime, most users do not find this very useful. The primary reason for this flexibility is so you can choose which features are needed for your specific system.

Note: Please do not select your specific device – or layout your board's hardware – before deciding which features are needed for your system.

If you have done microcontroller system design in the past, this is probably an obvious statement, but it's a mistake we've seen a number of times in the past.

Pin Selection

The PxSEL register lets you choose whether to use a peripheral or GPIO functionality for each pin. As you can see in the diagram below, DriverLib provides functions to specify this functionality.



In some devices, you'll actually find two select registers – which provides more pin mux options.

P1SEL0 & P2SEL1: FR5969 Example

Table 50. Port P1 (P1.0 to P1.2) Pin Functions (From the 'FR5969 datasheet')

PIN NAME (P1.x)	x	FUNCTION	CONTROL BITS/SIGNALS ⁽¹⁾		
			P1DIR.x	P1SEL1.x	P1SEL0.x
P1.0/TA0.1/DMAE0/RTC CLK/A0/CO/VREF-/VREF-	0	P1.0 (I/O)	I: 0; O: 1	0	0
		TA0.CC1A	0	0	1
		TA0.1	1		
		DMAE0	0	1	0
		RTCCLK	1		

Table 8-2. I/O Function Selection (From the 'FR5969 User's Guide')

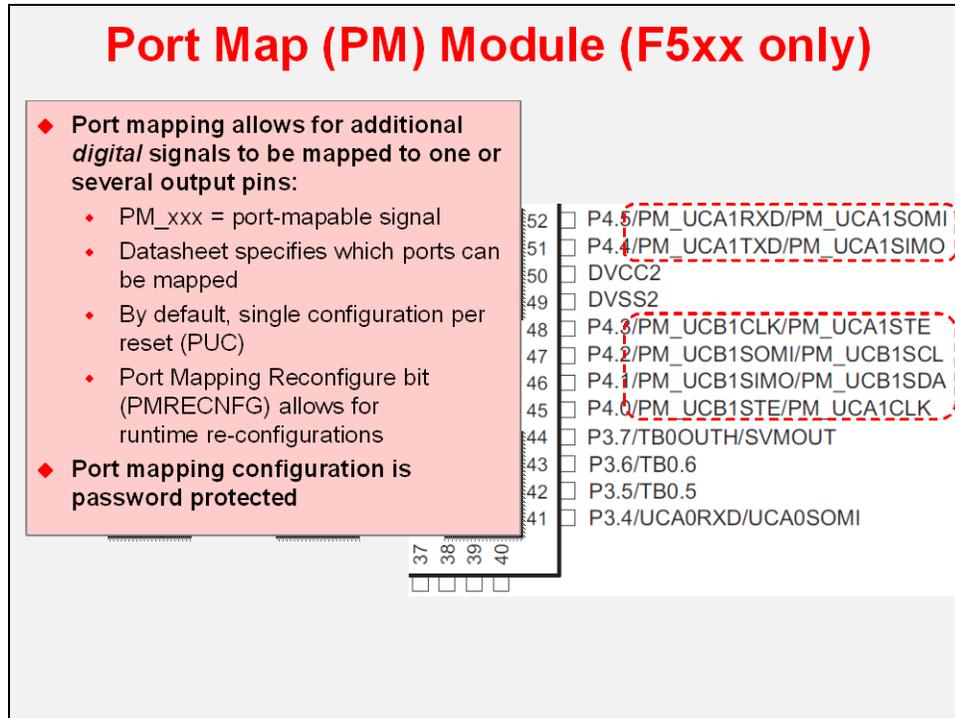
PxSEL1	PxSEL0	I/O Function
0	0	General purpose I/O is selected
0	1	Primary module function is selected
1	0	Secondary module function is selected
1	1	Tertiary module function is selected

- ◆ The User's Guide tells us how to read the datasheet
- ◆ DriverLib uses *I/O Function* name

```
// Set pin P1.3 to output direction, then turn LED off
GPIO_setAsPeripheralModuleFunctionOutputPin(
    GPIO_PORT_P1, // I/O Port number
    GPIO_PIN3,    // pin Number
    GPIO_PRIMARY_MODULE_FUNCTION ); // Which peripheral function
// (primary, secondary, ternary)
```

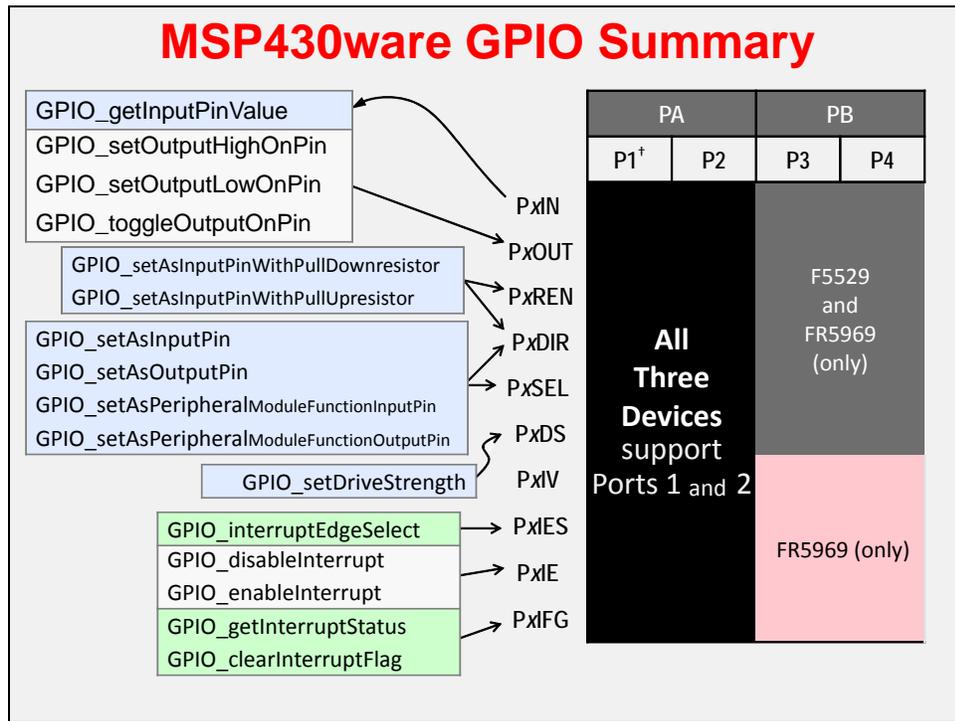
Port Mapping

The MSP430F5xx and 'F6xx devices provide the Port Map module which provides additional flexibility for mapping functions to pins. The signals that can be mapped to the port mapping pins are highlighted with a `PM_` prefix.



On the device shown above, only Port 4 has been designed with the Port Mapping (PM) feature.

The following diagram summarizes the GPIO API found in MSP430ware DriverLib. Not only have we listed the various functions, but we've indicated which GPIO registers they write to (or read from).



Before We Get Started Coding

Getting Your Program Started

We cover system initialization details in Chapter 4, but here are a few items needed for Lab 3:

1. Include required #include files
2. Turn off the Watchdog timer
3. Unlock pins (Wolverine devices)

1. #Include Files

If you've programmed in C for very long, you have probably become accustomed to using Include files. As described in the last chapter, every MSP430 device has a specific .h file created to define its various registers and symbols. When using the "Register" model of programming, you would need to include this header file.

To make programming easier, the DriverLib team combined all their header files into a single "driverlib.h" file; in fact, this header file also pulls in the appropriate .h file for your device.

Include Files

- ◆ Like most C programs, we need to include the required header files
- ◆ Each MSP430 device has its own .h file to define various symbols and registers – include this using `msp430.h`
- ◆ DriverLib defines all peripherals available for each given device – include `hw_memmap.h` (from /inc folder)
- ◆ But to make DriverLib easy, TI created a **single header file** to link in: `driverlib.h`

TIMER_A	AES	GPIO
TIMER_B	CRC	PM
TIMER_D	DMA	SPI
WDT_A	MPY32	I2C

```
#include <driverlib.h>
GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN7);
```

2. Disable Watchdog Timer

The MSP430 watchdog timer is always enabled. If you're just trying to get your first program to run, you won't need this feature, thus you can stop this timer with the DriverLib function shown below.

Disable WatchDog Timer

- ◆ MSP430 watchdog timer is always enabled at reset
- ◆ Watchdog timer requires modification password (0x5A)
- ◆ Easiest solution:
Begin your program with DriverLib (WDT_A) function

```

#include <driverlib.h>

WDT_A_hold(WDT_A_BASE); //Stop watchdog timer

```

Note: We discuss the watchdog timer in more detail during the next chapter.

3. Pin Unlocking (Wolverine only)

Pin locking is a feature that holds the last state of all GPIO pins when a device is put into its lowest power modes – that is, when power is removed from the memory and registers. Without this ‘locking’ feature, the pins would lose their values when these power modes are entered.

The pin-locking feature freezes the state of each pin. That is, the pins are effectively disconnected from their associated register bits (i.e. PxOUT) – you can think of there being a switch along the vertical dashed line shown below.

Pin UnLocking (FR58/59 only)

- ◆ PM5CTL0.**LOCKLPM5** bit disconnects registers from pins – allows pin values to remain constant during low power modes (LPM3.5/4.5)
- ◆ Bit automatically set upon entering LPMx.5 mode (see Low Power Chapter)
- ◆ **FR58/59 Wolverine** devices always power-on with this mode set – you must clear it for pins to respond to your register settings

```
GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN7 );
GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN7 );
PMM_unlockLPM5 ( ) ; //unlock pins after setting all gpio registers
```

Many devices prior to Wolverine, such as the ‘F5529, provided the pin-locking feature – although, it was not enabled by default. The new ‘FR5xxx (Wolverine) devices, though, have this feature enabled by default ... therefore, the pins are always locked at power-up.

When this feature is enabled, there is an additional ‘unlocking’ step required in order for your I/O to respond to the values written to the GPIO control registers.

As shown above, it is suggested that you setup your GPIO registers and then unlock the registers using the `PMM_unlockLPM5 ()` function.

Lab 3

We begin with a short Worksheet to prepare ourselves for coding GPIO using MSP430 DriverLib.

Next you'll implement the blinking LED example using DriverLib, finally adding a test of the push button in the final part of the lab exercise.

Lab 3 – Blink with MSP430ware

- ◆ **Lab Worksheet... a Quiz, of sorts on:**
 - ◆ GPIO
 - ◆ DriverLib
 - ◆ Path Variables
- ◆ **Lab 3a – Embedded 'Hello World'**
 - ◆ Create a MSP430ware DriverLib GPIO project
 - ◆ Use IDE path variables to make your project portable
 - ◆ Write code to enable LED
 - ◆ Use simple (inefficient) delay function to create ½ second LED blinking
 - ◆ Use CCSv5 debugging windows to view registers and memory
- ◆ **Lab 3b – Read Launchpad Push Button**
 - ◆ Test the state of the push button
 - ◆ Only blink LED when button is pushed (again, inefficient, but we'll fix that in Ch4)



Time:

Worksheet – 15 mins

Labs – 30 mins

Lab3 Abstract

Lab 3a – GPIO

This lab creates what is often called, the "Embedded Hello World" program.

Your code will blink the Launchpad's LED example using the MSP430ware DriverLib library. While this is a simple exercise, that's perfect for learning the mechanics of integrating DriverLib.

Part of learning to use a library involves adding it to our project and adding its location the compiler's search path.

Finally, along with single-stepping our program, we will explore the "Registers" window in CCS. This lets us view the CPU registers, watching how they change as we step thru our code.

Note: Our code example is a BAD way to implement a blinking light ... from an efficiency standpoint. The `_delay_cycles()` function is VERY INEFFICIENT. A timer, which we'll learn about in a later chapter, would be a better, lower-power way to implement a delay. For our purposes in this chapter, though, this is an easy function to get started with.

Lab 3b - Button

The goal of this lab is to light the LED when the SW1 button is pushed.

After setting up the two pins we need (one input, one output), the code enters an endless while loop where it checks the state of the push button and lights the LED if the button is pushed down.

Basic Steps:

- Cut/Paste previous project
- Delete/replace previous while loop
- Single-step code to observe behavior
- Run, to watch it work!

Note: "Polling" the button is very inefficient!

We'll improve on this in both the Interrupts and Timers chapters and exercises.

Lab 3 Worksheet

MSP430ware DriverLib

1. Where is your MSP430ware folder located? *(You should have written this down in the Installation Guide)*

2. To use the MSP430ware GPIO and Watchdog API, what header file needs to be included in your source file?

```
#include < _____ >
```

3. How do we turn off the Watchdog timer using a DriverLib function call?

_____ ;

GPIO Output

4. We need to initialize our GPIO output pin. What two GPIO DriverLib functions setup Port 1, Pin 0 (P1.0) as an output and set its value to "1"?

_____ ;

_____ ;

FR5969

For the 'FR5xx devices, what additional function do you need to call for the I/O to work?

_____ ;

5. Using the `_delay_cycles()` intrinsic function (from the last chapter), write the code to blink an LED with a 1 second delay setting the pin (P1.0) high, and then low?

```
#define ONE_SECOND 800000

while (1) {
    //Set pin to "1" (hint, see question 4)
    _____ ;
    _delay_cycles( ONE_SECOND );
    // Set pin to "0"
    _____ ;
    _delay_cycles( ONE_SECOND );
}
```

Check your answers against ours ... see the Chapter 3 Appendix.

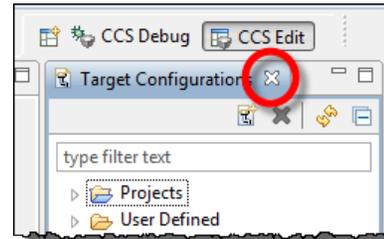
Lab 3a – Blinking an LED

1. Close any open project and file.

This helps to prevent us from accidentally working on the wrong file, which is easy to do when we have multiple lab exercises that use "main.c". If a previous project is open:

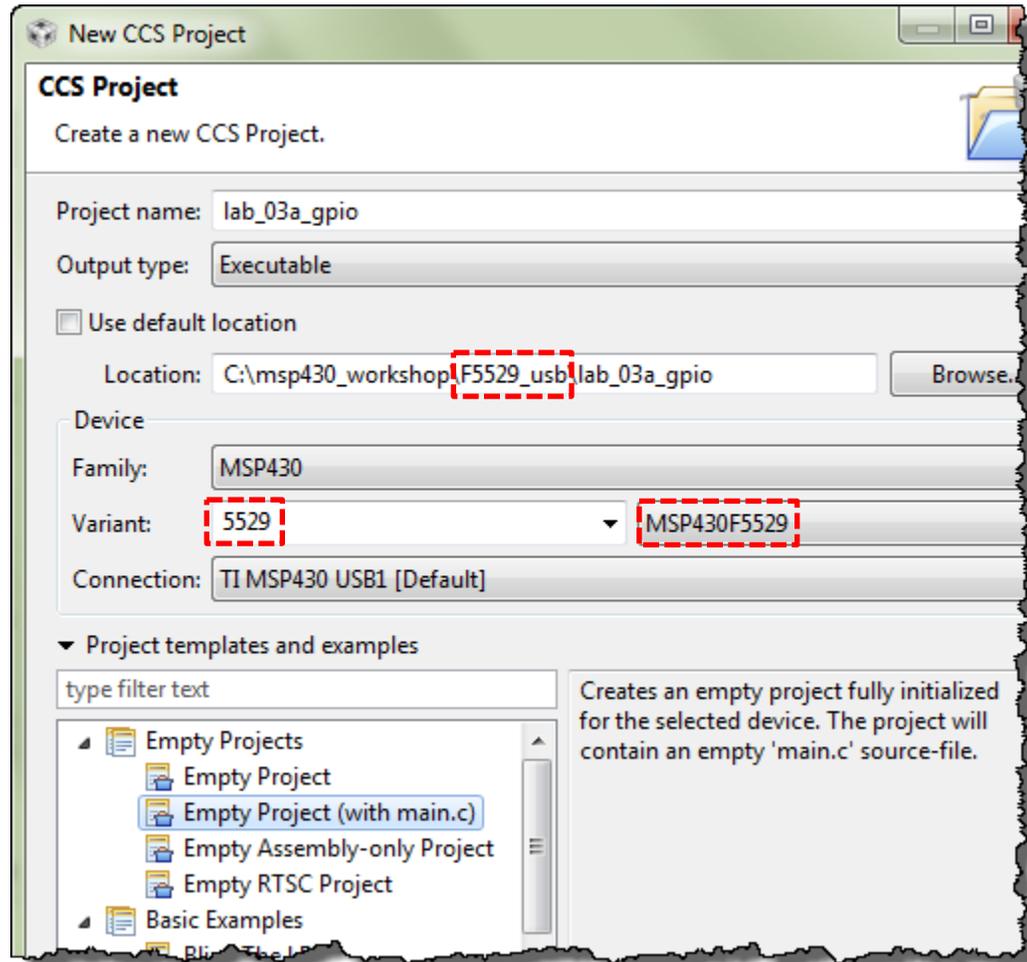
Right-click on the project and select "Close Project"

2. Also, if the Target Configurations window is open, please close it.



3. Create a new project.

Name the new project: lab_03a_gpio



Note: If you're working with the 'FR5969, please replace the 'F5529' references shown above with those required for your Launchpad.

4. Notice that the main() function already turns off the watchdog timer.

Although this is not required, you can replace this “register-based” code with the DriverLib function. Either way works fine. *If you want to use DriverLib, please reference your Worksheet answer #3 (on page 3-21).*

5. Add required header files.

Add the #include header required by MSP430ware DriverLib. (See Worksheet question #2).

Hint: The default main.c created by the new project wizard already has #included <msp430.h>. You can replace this with the DriverLib #include. It's OK to have both of them, but the DriverLib header file already references msp430.h.

???

6. Do you see question marks next to #include statement? What does this mean?

Add MSP430ware DriverLib

Hopefully you answered the last question by saying that we need to add the DriverLib library to our project. The question marks told us that CCS couldn't find the header file.

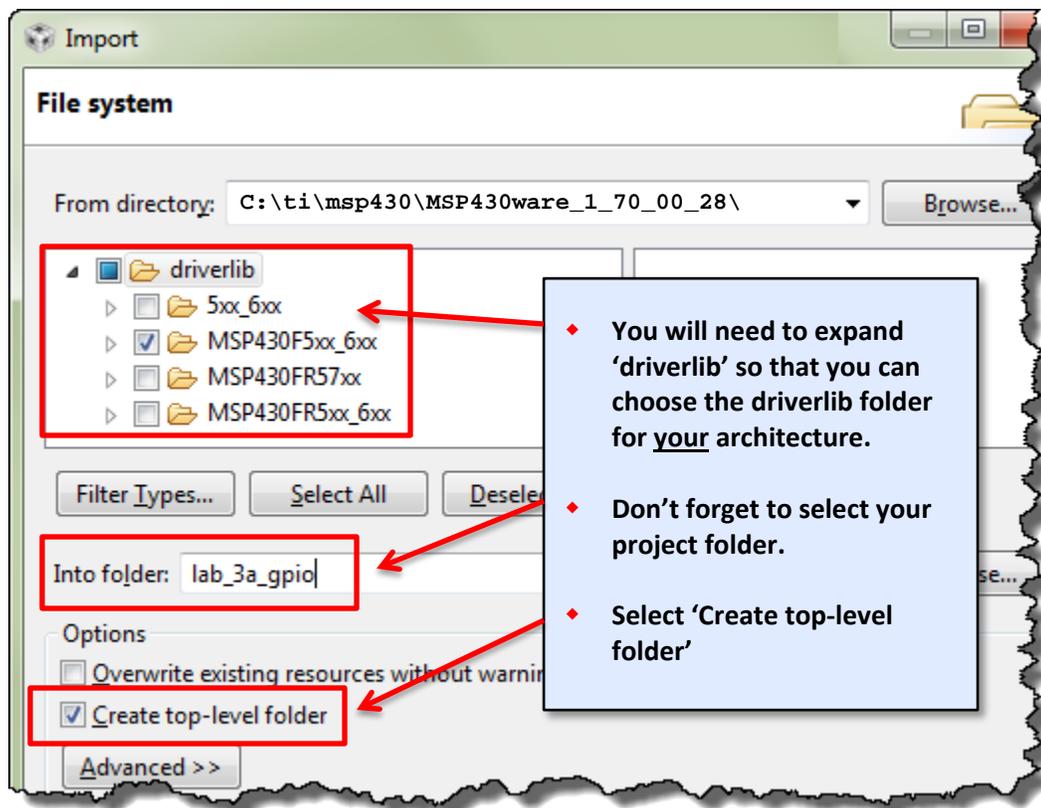
Adding the DriverLib library is a two-step process:

- Import a copy of the library
- Include the location in the CCS build search path

7. Import MSP430ware DriverLib library to your project.

File → Import... → General → File System

Then select the version and path of MSP430ware you are using. Note: Your path may be different than what is shown below. (See Worksheet question #1.)



After clicking *Finish*, you should notice the library folder was added to your project:

▷ driverlib/MSP430F5xx_6xx (or driverlib/MSP430FR5xx_6xx)

Note: The version of MSP430ware you have may vary slightly from what is shown above. If the version is lower (i.e. older), you should update it. If it is later, hopefully it will work without any problems.

8. Update your project's search path with the location of DriverLib header files.

Along with adding the library, we also need to tell the compiler where to find it.

Open the Include Options and add the directory to #include search path:

Right-click project → Properties

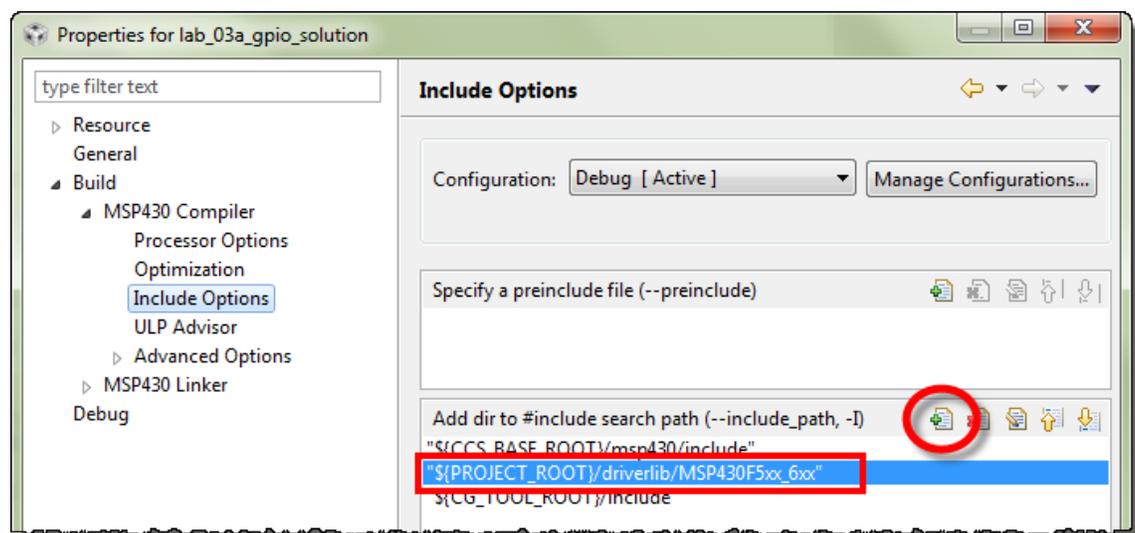
Then select:

Build → MSP430 Compiler → Include Options

and add the appropriate path to the #include search path:

```

    ${PROJECT_ROOT}\driverlib\MSP430F5xx_6xx
    or ${PROJECT_ROOT}\driverlib\MSP430FR5xx_6xx
  
```



With this step done, you should notice the ??? gone from the #include statements.



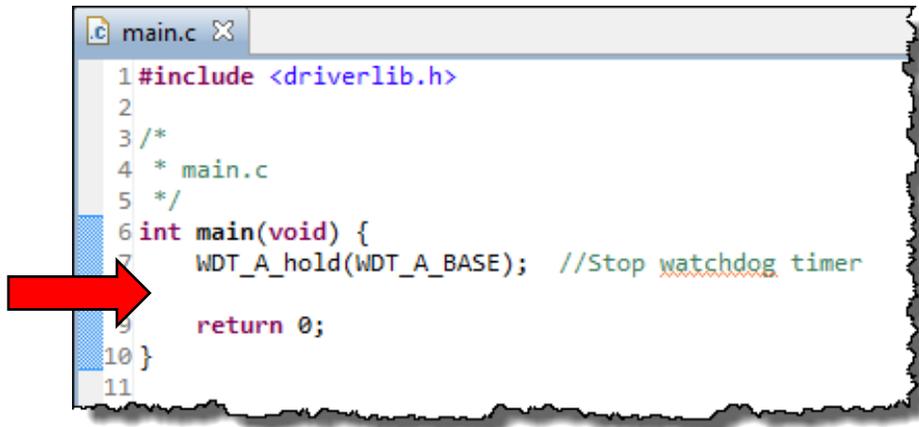
9. Click the build toolbar button to verify that your edits, thus far, are correct.

Add the Code to `main.c`

10. Setup P1.0 as output pin.

Reference Worksheet question #4 (page 3-21).

Begin writing your code after the code that disables the watchdog timer as shown:



```

1 #include <driverlib.h>
2
3 /*
4  * main.c
5  */
6 int main(void) {
7     WDT_A_hold(WDT_A_BASE); //Stop watchdog timer
8
9     return 0;
10 }
11

```

FR5969

Hint: If you're using the 'FR5969 Launchpad, don't forget to add the line of code which unlocks the pins. (Reference Worksheet question 4b (page 3-21)).

11. Create a `while{} loop` that turns LED1 off/on with a 1 second delay.

Reference Worksheet question #0 (page 3-21). Begin the `while{} loop` after the code you wrote in the previous step. Also, don't forget to add the `#define` for "ONE_SECOND" towards the top of the file.



12. Build your program with the Hammer icon.

Make sure your program builds correctly, fixing any syntax mistakes found by the compiler. For now, you can ignore any remarks or advice recommendation, we'll explore this later.



13. Load and Run your program.

Click the easy Debug button to start the debugger and download your program. Then click the Run button.

Does your LED flash? _____

If it doesn't, let's hope following debug steps help you to track down your error.

If it does, hooray! We still think you should perform the following debug steps, if only to better understand some additional features of CCS.



14. Suspend the debugger.

Alt-F8

Debug



15. Restart your program.

16. Open the Registers window and view P1DIR and P1OUT. Then single-step past the GPIO DriverLib functions.

View → Registers

Expand Port_1_2, P1OUT and P1DIR as shown



Then, single-step (i.e. Step Over – F6) until you execute this line:

```
GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );
```

Your register view should now look similar to this:

Name	Value
Port_A	
Port_1_2	
P1IN	0xFF
P1OUT	0x01
P1OUT7	0
P1OUT6	0
P1OUT5	0
P1OUT4	0
P1OUT3	0
P1OUT2	0
P1OUT1	0
P1OUT0	1
P1DIR	0x01
P1DIR7	0
P1DIR6	0
P1DIR5	0
P1DIR4	0
P1DIR3	0
P1DIR2	0
P1DIR1	0
P1DIR0	1
P1REN	0x00

17. Single-step until you reach the `_delay_cycles()` function.

You should see the P1OUT register change as you step over the appropriate function.

Unfortunately, the “Step Over” command doesn’t step over `_delay_cycles()`.

18. Set breakpoints on both `GPIO_setAs...` functions, then *Run* and check values in *Registers* window.

Since it's difficult to step over `_delay_cycles()`, we'll just run past them. Setting the breakpoints on both lines where we change the GPIO pin value, we should see the LED toggle each time you press run.

Set breakpoints as shown below:

```

main.c
12  GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);
13
14  while(1){
15
16      GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN0);
17
18      _delay_cycles (ONE_SECOND);
19
20      GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN0);
21
22      _delay_cycles (ONE_SECOND);
23
24  }
    
```

Then click Run several times stopping at each breakpoint and keeping your eye on the LED.

Note: Following these debugging steps, we ended up finding the problem in our original code. A cut and paste error left us with two lines of code in our loop that both turned off the LED. Oops!

While basic debugging techniques, these steps are powerful tools for finding and fixing errors in your code.

19. If you're using the 'FR5969 Launchpad, you may want to examine the PM5CTL0 register.

FR5969

If you've already run your code, the `PM5CTL0.LOCKLPM5` should already have been cleared by your program. It requires power-cycle to reset to set this to its initial condition. Follow these steps to see your code "unlock" the pins on the device.

- a) If running, suspend your program.

Alt-F8

- b) Open the register window and display the LOCKLPM5 bit.

- c) Perform a *Hard Reset*.

Run → Reset → Hard Reset

- d) Then, restart the program.



- e) Finally, single-step your program until you see LOCKLPM5 value change to 0.

Name	Value	Description
PM5MIFG	0x0200	PMM Interrupt Flag [Mem...
PM5CTL0	0x0001	PMM Power Mode 5 Cont...
LOCKLPM5	1	Lock I/O pin configuration
Port_A		

PM5MIFG	0x0200
PM5CTL0	0x0000
LOCKLPM5	0

the

Lab 3b – Reading a Push Button

GPIO Input Worksheet

1. What three DriverLib functions can setup a GPIO pin for input?

Hint, one place to look would be the MSP430 DriverLib Users Guide found in the MSP430ware folder:

```
\MSP430ware_1_70_00_28\driverlib\doc\MSP430F5xx_6xx\
\MSP430ware_1_70_00_28\driverlib\doc\MSP430FR5xx_6xx\
```

2. What can happen to an input pin that isn't tied high or low?

3. Assuming you need a pull-up resistor for a GPIO input, write the line of code required to setup pin P1.1 for input:

4. Complete the following code to read pin P1.1:

```
volatile unsigned short usiButton1 = 0;
while(1) {
    // Read the pin for push-button S2

    usiButton1 = _____;

    if ( usiButton1 == GPIO_INPUT_PIN_LOW ) {
        // If button is down, turn on LED
        GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
    }
    else {
        // Otherwise, if button is up, turn off LED
        GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
    }
}
```

5. In embedded systems, what is the name given to the way in which we are reading the button? (Hint – it's not an interrupt.)

Check your answers against ours ... see the Chapter 3 Appendix.

File Management

We're going to try another – easier – method of creating a new DriverLib project from scratch.

Import the Empty driverlib example project

1. Import the *emptyProject* from the MSP430 DriverLib examples.

There are a couple different ways to import the example projects, but in this lab we'll utilize the TI Resource Explorer as it provides convenient access to examples from within CCS.

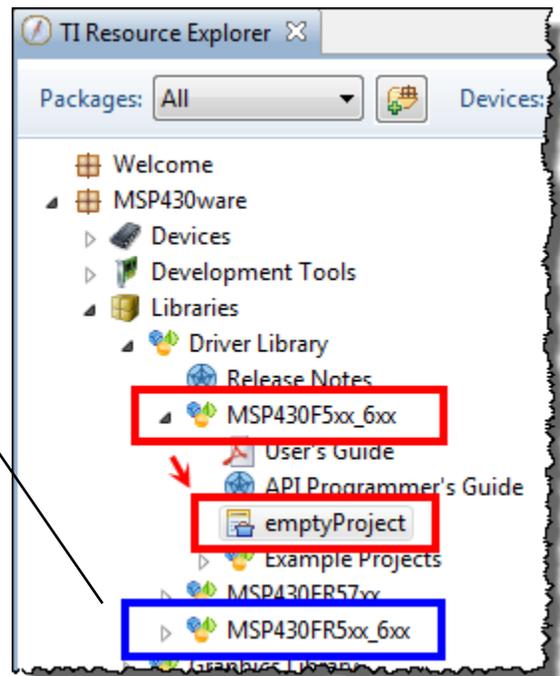
a) Open the TI Resource Explorer window, if it's not already open

View → TI Resource Explorer

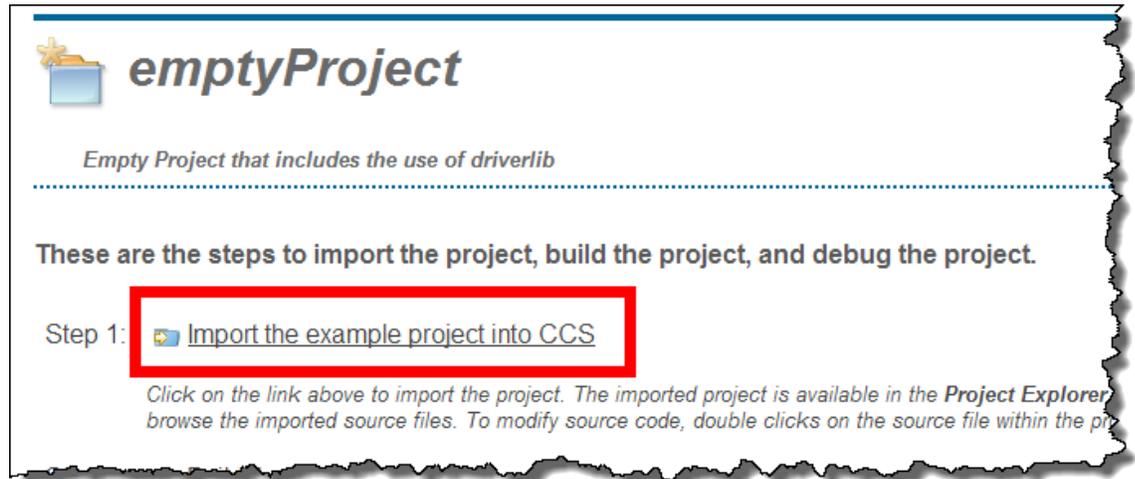
b) Locate the *emptyProject* example.

Look for it as shown here:

If you're using the FR5969, follow the same path starting from the *MSP430FR5xx_6xx* heading.



- c) Click the link to “Import the example project into CCS”.



Once imported you can close the TI Resource Explorer, if you want to get it out of the way.

- d) **Rename the imported project to: lab_03b_button** (Right-click on the project name and select “Rename”)

2. Quickly examine the new lab_03b_button Project.

Looking at this project, you’ll see that it already has the DriverLib library imported into the project. Also, the required #include search path entry has already been added to the project.

Copy our code from the previous project

3. Delete the ‘empty’ main.c from the new project.

4. Copy/Paste main.c from lab_03a_gpio to lab_03b_button.

You can easily copy and paste files right inside the CCS Project Explorer. Simply right-click on the file (main.c) from the previous project and select “Copy” and then right-click on the new project and select “Paste”.

(Alternatively, we could have just copied and pasted the main() function from our previous lab project, but we found it easier to copy the whole file.)

5. Close the previous lab: lab_03a_gpio

As we’ve learned, this should close the .c source files associated with the project, which can help us from accidentally editing the wrong file. (Believe us, this happens a lot.). Right-click on the project and select “Close Project”.



6. Build the new lab, just to make sure everything was copied correctly.

Add Setup Code (to reference push button)

7. Before `main()`, add the global variable: `usiButton1`

```
volatile unsigned short usiButton1 = 0;
```

Let's explain some of our choices:

Global variable: We chose to use a *global* variable because it's in scope all the time. Since it exists all the time (as opposed to a *local* variable), it's just a bit easier to debug the code. Otherwise, local variables are probably a better choice: better programming style, less prone to naming conflicts and more memory efficient.

Volatile: We'll use this variable to hold the state of the switch, after reading it with our `DriverLib` function.

Does this variable change outside the scope of C? _____

Absolutely; its value depends upon the pushbutton's up/down state. That is why we must declare the variable as *volatile*.

unsigned short ... You tell us, why did we pick that? _____

usiButton1: The 'usi' is Hungarian notation for *unsigned short integer*. We added the '1' to 'Button', just in case we want to add a variable for the other button later on. (We could have also used the names 'S1' and 'S2' as they're labeled on the Launchpad, but we liked 'Button' better.)

=0 ... well, that's just good style. You should always initialize your variables. Many embedded processor compilers do not automatically initialize variables for you.

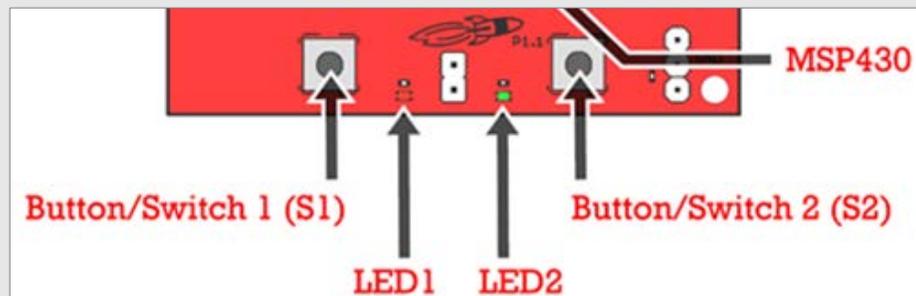
8. In `main()`, add code to setup push button (S2) as an input with pull-up resistor.

This setup code should go before the `while()` loop. (And for the 'FR5969, we recommend placing this code before the `unlock LPM5` function.)

And don't forget, this code was the answer to Worksheet question #3 (page 3-29).

As a reminder – S2 is connected to Port 1, Pin 1)

Hint: We should've have recommended bringing a magnifying glass to read the silk screen on the Launchpad board. It's very hard to see which button is S2 – and the pin it is connected to. It may be easier to reference the Quick Start sheet that came with your Launchpad.



Modify Loop

9. Modify the while loop to light LED when S2 push button is pressed.

Comment out (or delete) LED blinking code and replace it with the code we created in the Worksheet question #4 (page 3-29).

At this point, your `main.c` file should look similar to this:

```
// -----
// main.c (for lab_03b_button project)
// -----

//***** Header Files *****
#include <driverlib.h>

//***** Global Variables *****
volatile unsigned short usiButton1 = 0;

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

    // Set P1.0 to output direction, P1.1 as input with pullup resistor
    GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );
    GPIO_setAsInputPinWithPullUpresistor( GPIO_PORT_P1, GPIO_PIN1 );

    // Unlock pins (required for Wolverine 'FR5xx devices)
    PMM_unlockLPM5();

    while(1) {
        // Read P1.1 pin connected to push button S2
        usiButton1 = GPIO_getInputPinValue ( GPIO_PORT_P1,
                                             GPIO_PIN1 );

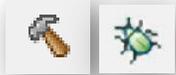
        if ( usiButton1 == GPIO_INPUT_PIN_LOW ) {
            // If button is down, turn on LED
            GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        }
        else {
            // If button is up, turn off LED
            GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        }
    }
}
```

Hint: If you want to minimize your typing errors, you can copy/paste the code from the listing above. We have also placed a copy of this code into the lab's readme file (in the lab folder); just in case the copy/paste doesn't work well from the PDF file.

Copying from PDF will usually mess up the code's indentation. You can fix this by selecting the code inside CCS and telling it to clean-up indentation:

Right-click → Source → Correct Indentation (Ctrl+I)

Verify Code



10. Build & Load program.

11. Add the `usiButton1` variable to the Watch Expression window.

Hint: select the variable name before you right-click on it and add it to the *Watch* window.



12. Single-step project. Watch the LED and variable.

Loop thru `while{}` multiple times with the button pressed (and not pressed), watching the variable (and LED) change value.



13. Run the program.

Go ahead and click the Run toolbar button and revel in your code, as the LED lights whenever you push the button.

Note: This is not efficient code. It would be much better to use the push-button input pin as an interrupt ... which we'll do in Chapter 5.

Optional Exercises

- Try this lab without pull-up (or pull-down) resistor.

Without the resistor, is the pushbutton's value always consistent? (yes / no) _____

- Try using the other LED on the board ...
- ... or the other pushbutton.



Chapter 3 Appendix

Lab3a – Worksheet

1. Where is your MSP430ware folder located?
Most likely: `C:\ti\msp430\MSP430ware_1_70_00_25\`
2. To use the MSP430ware GPIO and Watchdog API, what header file needs to be included in your source file?
`#include < driverlib.h >`
3. How do we turn off the Watchdog timer?
`WDT_A_hold(WDT_A_BASE) ;`
- 4a. We need to initialize our GPIO output pin. What two GPIO DriverLib functions setup Port 1, Pin 0 (P1.0) as an output and set its value to "1"?
`GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0) ;`
`GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN0) ;`
- 4b. For the 'FR5xx devices, what additional function do you need to call for the I/O to work?
`PMM_unlockLPM5() ;`

Lab3a – Worksheet

5. Using the `_delay_cycles()` intrinsic function (from the last chapter), write the code to blink an LED with a 1 second delay setting the pin (P1.0) high, then low?

```
#define ONE_SECOND 800000

while (1) {
    //Set pin to "1" (hint, see question 4)
    GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN0) ;
    _delay_cycles( ONE_SECOND ) ;
    // Set pin to "0"
    GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN0) ;
    _delay_cycles( ONE_SECOND ) ;
}
```

Lab3b – Worksheet

1. What three functions choices are there for setting up a pin for GPIO input?

Hint, one place to look would be the MSP430 Driverlib Users Guide found in the MSP430ware folder:

```
\ MSP430ware_1_70_00_28\driverlib\doc\MSP430F5xx_6xx\
```

GPIO_setAsInputPin()

GPIO_setAsInputPinWithPullDownresistor()

GPIO_setAsInputPinWithPullUpresistor()

2. What can happen to an input pin that isn't tied high or low?

The input pin could end up floating up or down. This uses more power ... and can give you erroneous results.

3. Assuming you need a pull-up resistor for a GPIO input, write line of code required to setup pin P1.1 for input:

GPIO_setAsInputPinWithPullUpresistor (GPIO_PORT_P1, GPIO_PIN1) ;

Lab3b – Worksheet

4. Complete the following code to read pin P1.1:

```
volatile unsigned short usiButton1 = 0;
while(1) {
    // Read the pin for push-button S2
    usiButton1 = GPIO_getInputPinValue ( GPIO_PORT_P1, GPIO_PIN1 );
    if ( usiButton1 == GPIO_INPUT_PIN_LOW ) {
        // If button is down, turn on LED
        GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
    }
    else {
        // Otherwise, if button is up, turn off LED
        GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
    }
}
```

5. In embedded systems, what is the name given to the way in which we are reading the button? (Hint, it's not an interrupt)

"Polling"

MSP430 Clocks & Initialization

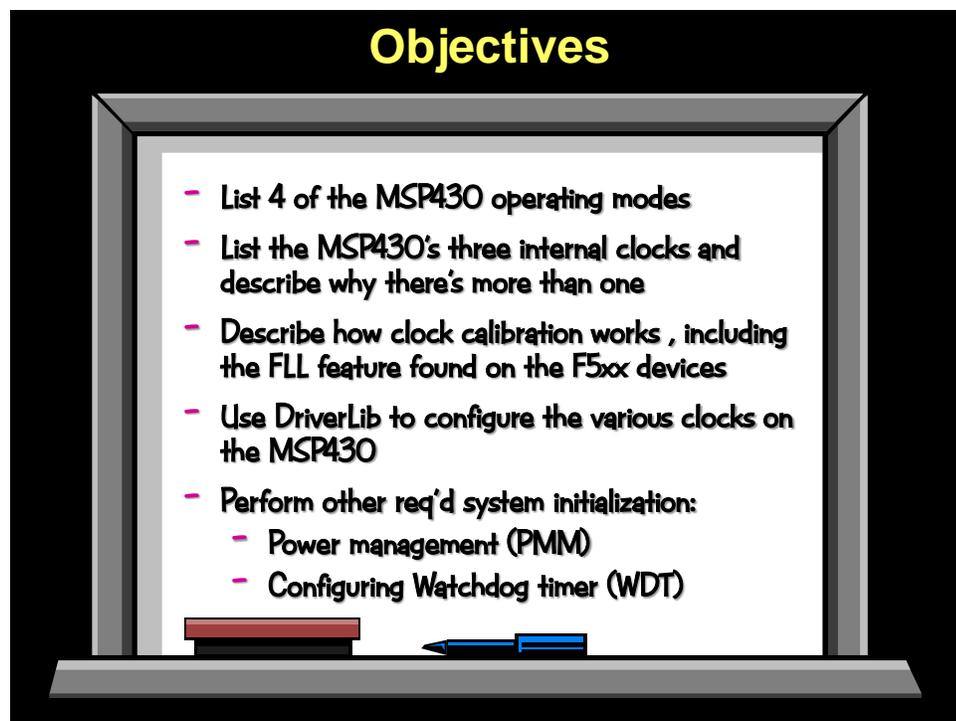
Introduction

A fundamental part of any modern MCU is its clocking. While rarely a flashy part of system design, it provides the heartbeat of the system. It becomes even more important in applications that depend upon precise, or very low-power, timing.

The MSP430 provides a wealth of clock sources; from ultra-low-power, low-cost, on-chip clock sources to high-speed external crystal inputs. All of these can be brought to bear through the use of 3 internal clock signals, which drive the CPU along as well as fast and slow peripherals.

Along with clocking, though, there are a few other items that need to be initialized at system startup. Towards the end of the chapter, we touch on the power management and watchdog features of the MSP430.

Learning Objectives



Chapter Topics

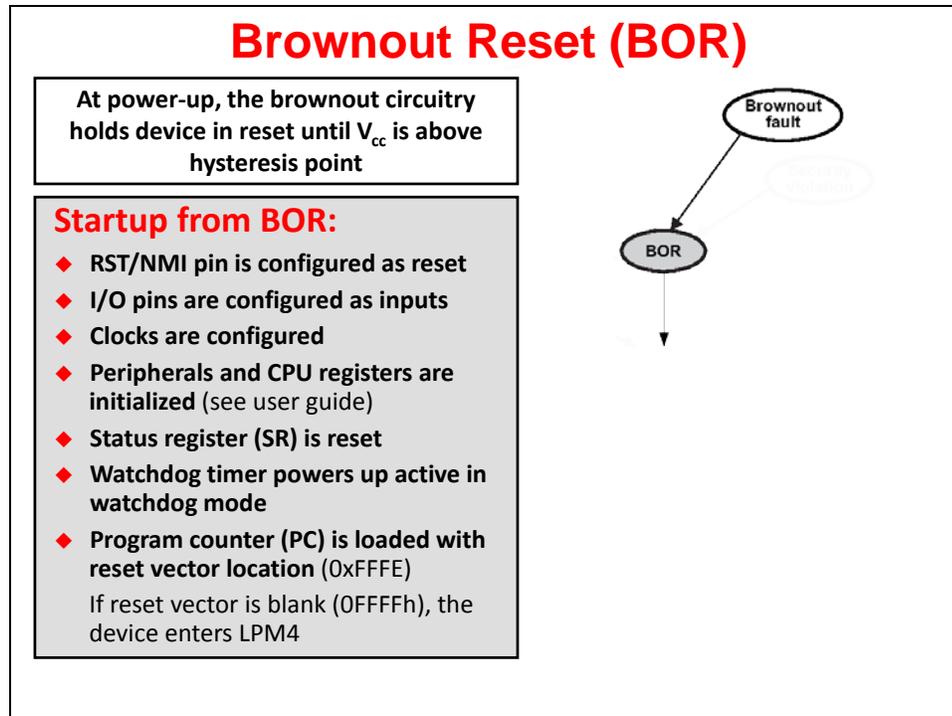
MSP430 Clocks & Initialization	4-1
<i>Operating Modes (Reset → Active)</i>	4-3
BOR.....	4-3
BOR → POR → PUC → Active (AM)	4-4
<i>Clocking</i>	4-6
What Clocks Do You Need?	4-6
MCLK, SMCLK, ACLK	4-8
Oscillators (Clock Sources).....	4-9
Clock Details (by Device Family)	4-11
Using MSP430ware to Configure Clocking.....	4-16
Additional Clock Features	4-18
<i>DCO Setup and Calibration</i>	4-21
How the DCO Works.....	4-22
Factory Calibration (G2xx, FR5xx).....	4-26
Runtime Calibration.....	4-28
VLO 'Calibration'	4-31
<i>Other Initialization (WDT, PMM)</i>	4-32
Watchdog Timer.....	4-33
PMM with LDO, SVM, SVS, and BOR	4-34
Operating Voltages	4-36
Summary	4-37
Initialization Summary (template).....	4-40
<i>Lab Exercise</i>	4-41

Operating Modes (Reset → Active)

The MSP430 has a number of operating modes. In this chapter we explore the modes that take the processor from startup to active. In a future chapter, the low-power modes will be explored.

BOR

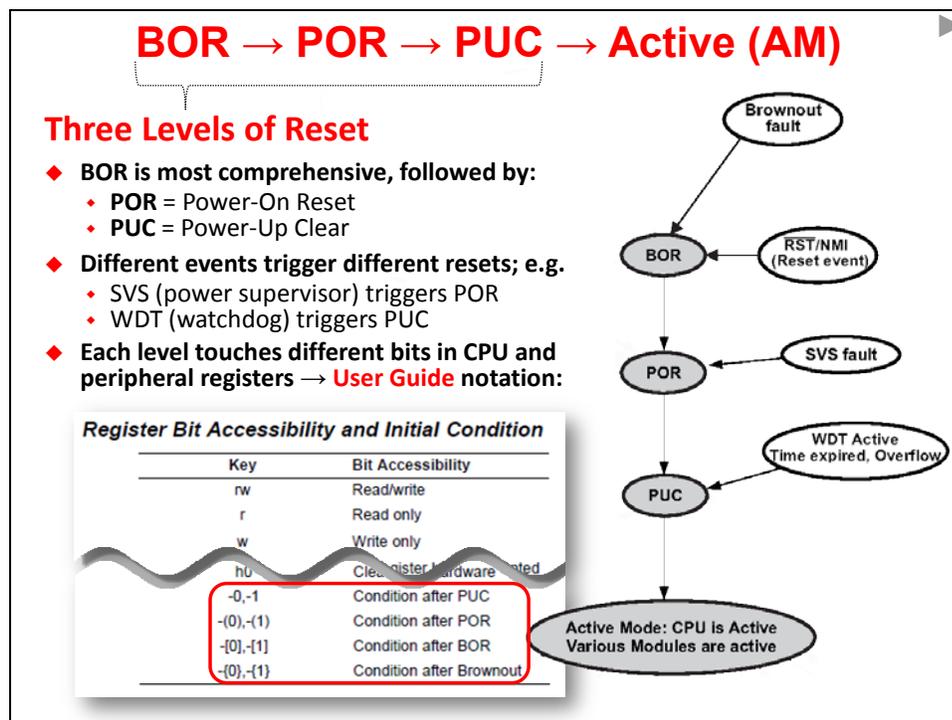
The MSP430 starts out in the Brown-Out Reset (BOR) mode. A Brownout Fault (i.e. not enough power) is the most common event that brings the CPU to this state.



In BOR, a series of items (listed above) are changed to their default states. (As always, the device datasheet and users guide should be the final reference as to what is changed in each of the reset states.)

BOR → POR → PUC → Active (AM)

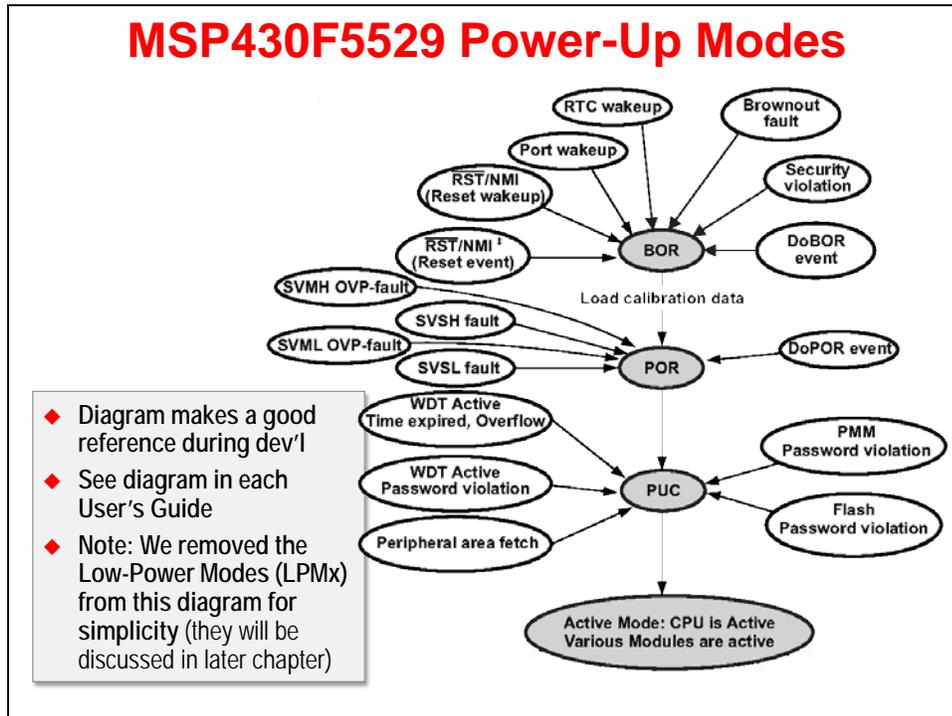
As shown below, BOR is the first of three reset states.



Different reset states, such as BOR, POR and PUC are triggered from different events. For example, upon power-up you may want to do a full system reset; though, this is usually not desired for something like a watchdog timeout event.

The previous page contained a list of actions that occur in the MSP430 hardware when a BOR event occurs. To find these details for all of the reset modes, please refer to the datasheet and users guides; as shown above, there are different nomenclature which represent the reset mode where a given hardware default value is applied.

Here's the full diagram showing the Reset and Active modes for the 'F5529¹. This shows all the various events that direct the MSP430 CPU into its different *Reset* states. You can find a similar diagram for each series of MSP430 processors.



¹ MSP430x5xx and MSP430x6xx Family User's Guide, slau208m.pdf, (Texas Instruments, 2013) pg. 63

Clocking

What Clocks Do You Need?

MSP430 provides a wide range of clocking options. Before choosing and configuring the clocks, though, you need to determine which clock features are most important for your system: Fast, low-power, accurate, etc. At times, choosing these various options may force you to make tradeoffs; hence, it's important to for you consider which of these (or what group of them) are most significant for your end-application.

What Clocks Do You Need?

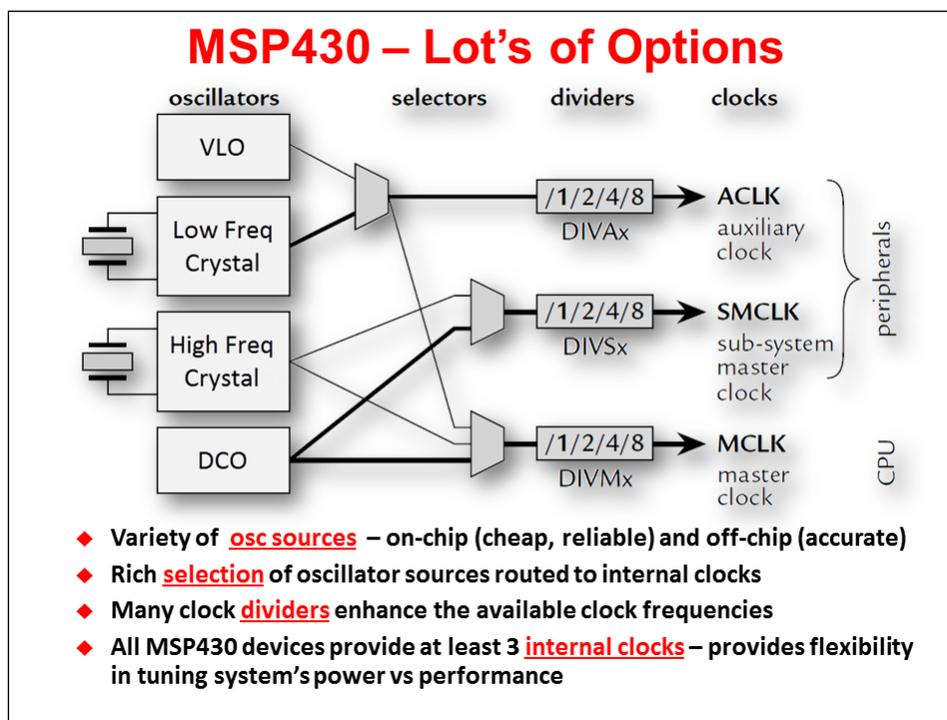
- ◆ **Fast Clocks** CPU, Communications, Burst Processing
- ◆ **Low-power** RTC, Remote, Battery, Energy Harvesting
- ◆ **Accurate** Stable over %V, Communications, RTC, Sensors
- ◆ **Failsafe** Robust—keeps system running in case of failure
- ◆ **Cheap** ... goes without saying ...

... or some combination of these features?

MSP430's rich clock ecosystem provides three internal clocks from a variety of clock sources.

Let's start on the right-side of the following diagram; there are 3 internal clocks which provide a variety of high and/or low speed options.

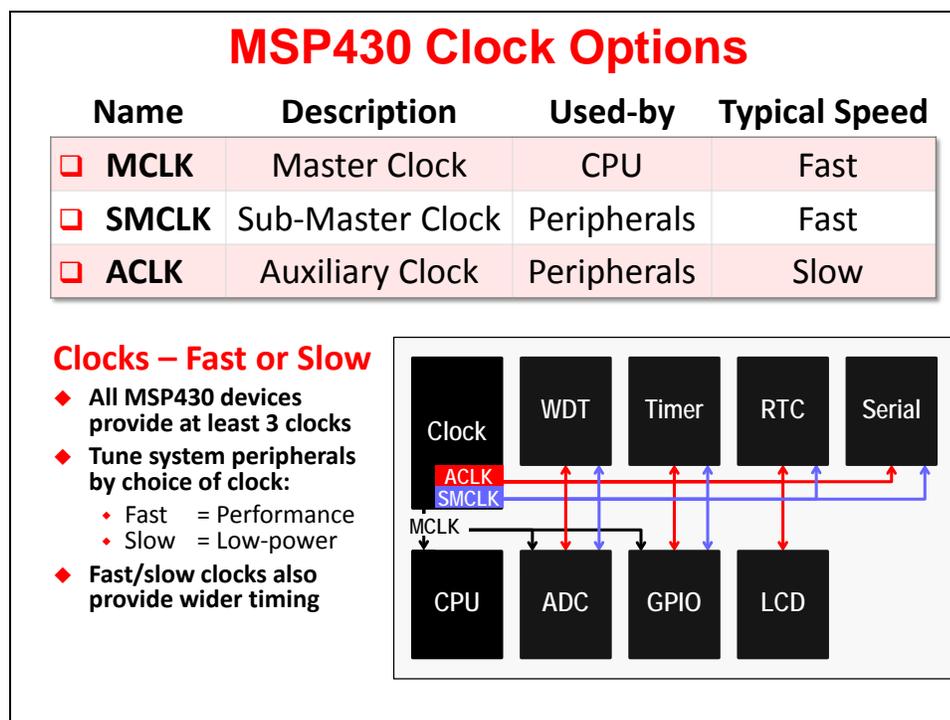
On the left-hand side, there are internal and external oscillators which provide both high and slow speed clock sources.



The next few slides provide further examination of the source oscillators and internal clocks.

MCLK, SMCLK, ACLK

As described in the following graphic, MCLK drives the clock rate of the CPU. It typically runs at a “fast” speed – from 1 MHz up to 16 or 25 MHz (depending upon the upper limit of the given device). MCLK can run slower than this, but it’s more common to see the CPU run in the MHz range in order to get its work done quickly and then go into one of the low-power “sleep” modes.



SMCLK and ACLK are primarily used for clocking peripherals. It’s convenient to have two peripheral clocks – one faster (SMCLK) and another slower (ACLK).

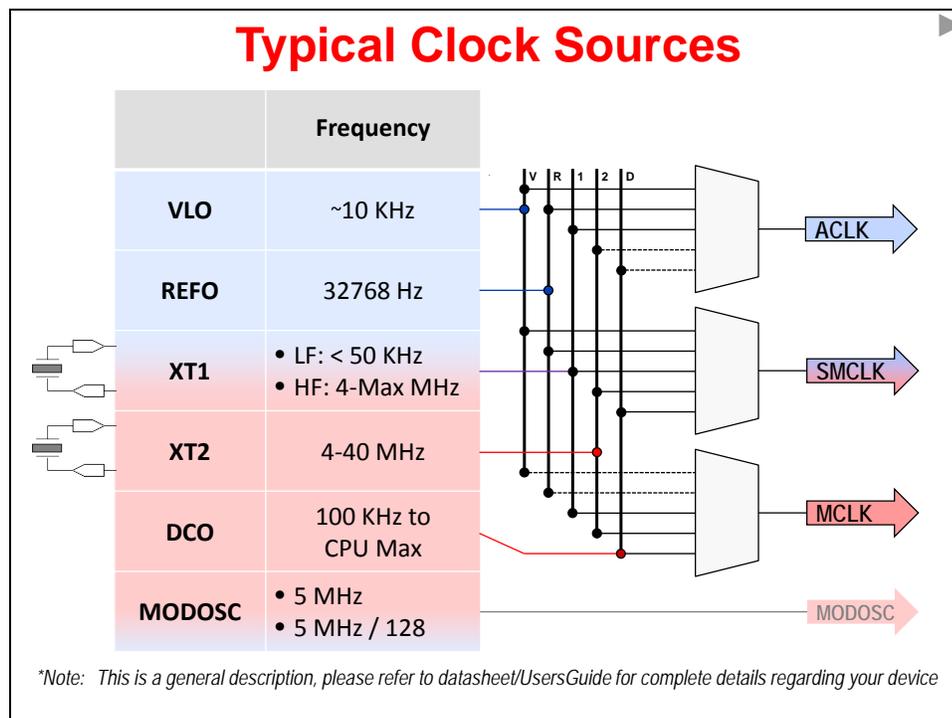
Some peripherals (such as serial ports) often require a fast clock to meet the communication data rate requirements while other peripherals (e.g. timers) may not always need to run as fast. The ability to provide a low-speed clock can provide two advantages:

- As you probably know, higher frequencies beget higher power usage; thus, a lower-speed clock saves power.
- It is often difficult to provide slow-enough timing if you only have a single, fast clock. Two peripheral clocks provide a greater range of performance to the various peripherals on the device.

The preceding graphic shows how one might use these various clocks on the MSP430. Please refer to the datasheet, though, since these vary slightly by device. For example, some devices allow all three clocks (MCLK, SMCLK, ACLK) to drive all of the peripherals while others only allow SMCLK and ACLK.

Oscillators (Clock Sources)

The typical MSP430 device provides a wide range of clock oscillator sources: internal/external, fast/slow, higher precision vs lower cost. Looking at the diagram, we can see that the typical sources are listed in the order from lower to higher frequency. Two slides from now, we'll compare the essential differences between the oscillator clock sources.



Again, we caution you to examine the datasheet carefully to determine which oscillator clock sources are available for your specific device. That said, the following table provides a quick snapshot of what sources are available on each of the three MSP430 Launchpad's.

Typical Clock Sources

	Frequency	'G2553 Value-line	'F5529 USB	'F5969 Wolverine
VLO	~10 KHz	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
REFO	32768 Hz		<input checked="" type="checkbox"/>	
XT1	<ul style="list-style-type: none"> LF: < 50 KHz HF: 4-Max MHz 	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
XT2	4-40 MHz		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
DCO	100 KHz to CPU Max	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
MODOSC	<ul style="list-style-type: none"> 5 MHz 5 MHz / 128 	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

**Note: This is a general description, please refer to datasheet*

Here we see that the typical sources are listed in the order from lower to higher frequency. In this case, we're looking specifically at the clock source options found on the 'F5529.

Clock Source Details ('F5529)

	Frequency	Precision	Current / Startup	Comments
VLO	~10 KHz	Very Low (±40%)	60nA	Use as Ultra Low Power tick
REFO	32768 Hz	Med/High (3.5%)	3µA 25µS	Trimmed to 3.5%
XT1	<ul style="list-style-type: none"> • LF: < 50 KHz • HF: 4-Max MHz 	High	75nA 500-1k mS	Crystal or Ext Clock
XT2	4-40 MHz	High	260µA (12MHz) 400µS	Crystal or Ext Clock
DCO	100 KHz to CPU Max	Low/Med	60µA 200nS	Calibrate with Constant/FLL
MODOSC	<ul style="list-style-type: none"> • 5 MHz • 5 MHz / 128 	Med	N/A	Used by FLASH or ADC

VLO: most MSP430 devices provide a Very Low-frequency Oscillator (VLO). While not a highly accurate clock, this source is extremely low-power. Also, as it is internal to the chip, it ends up being very inexpensive. If you need to wake up the processor every couple seconds to perform a task (i.e. read a sensor), the low-power VLO is a common way to get this done.

REFO: not all devices provide the REference Oscillator (REFO) source, but when available, it's a low-cost, internal source for the common "watch crystal" frequency. This can be a convenient way to drive a real-time clock in your system without requiring an external crystal. While not quite as accurate as some crystals, it's a less-expensive, robust solution.

XT1 and **XT2:** as the graphic demonstrates, XT1 and XT2 provide the eXTernal clock inputs. These sources, along with a couple pins each, provide a means of connecting to external crystal oscillator sources.

- Not all devices provide both clock sources; for example, we saw on the previous page that the 'G2553 only has XT1 (in fact, it's actually called LFX1 on that device).
- Why would you need two external clocks? For those cases when you need very precise low and high frequency clocks. For example, you might use XT1 to drive a real-time clock (RTC) while the 'F5529 uses XT2 to source a high-speed, high-precision clock to the USB peripheral.
- It should also be noted that you can connect a digital oscillator signal directly to these inputs; that is, you don't have to use a crystal if you've already got the necessary frequency on your board.
- Bottom line, the XT inputs provide the highest possible precision, but are a little less robust since crystals can often be one of the most delicate components in a system.

DCO: the Digitally Controlled Oscillator (DCO) provides a fast, low-lost, on-chip oscillator source. It is very common to see this source being used to drive the CPU and many high-speed peripherals. Another great feature is fast start-up time for this source, which is very important in a low-power system (where you might want to sleep the clock to save power). Later in the chapter, we'll explore a variety of methods for 'tuning' the DCO for improved accuracy.

MODOSC: the MODuale OSCillator (MODOSC) is another common high-frequency source. In some devices, it dedicated to the Analog to Digital Converter (ADC) - which can start and stop the source as needed. On other devices, though, the clock can be used to source a variety of peripherals. In any case, this is another on-chip oscillator source.

Clock Details (by Device Family)

The MSP430F5529 specific clock options we just examined are found in the F5xx/F6xx UCS (Unified Clock System) peripheral. As we've stated, various device sub-families provide different clocking features and options. Each "unique" set of options is described by a clock peripheral name – for example, while the 'F5529 has the UCS peripheral, the 'FR5969 Wolverine devices use the CS peripheral.

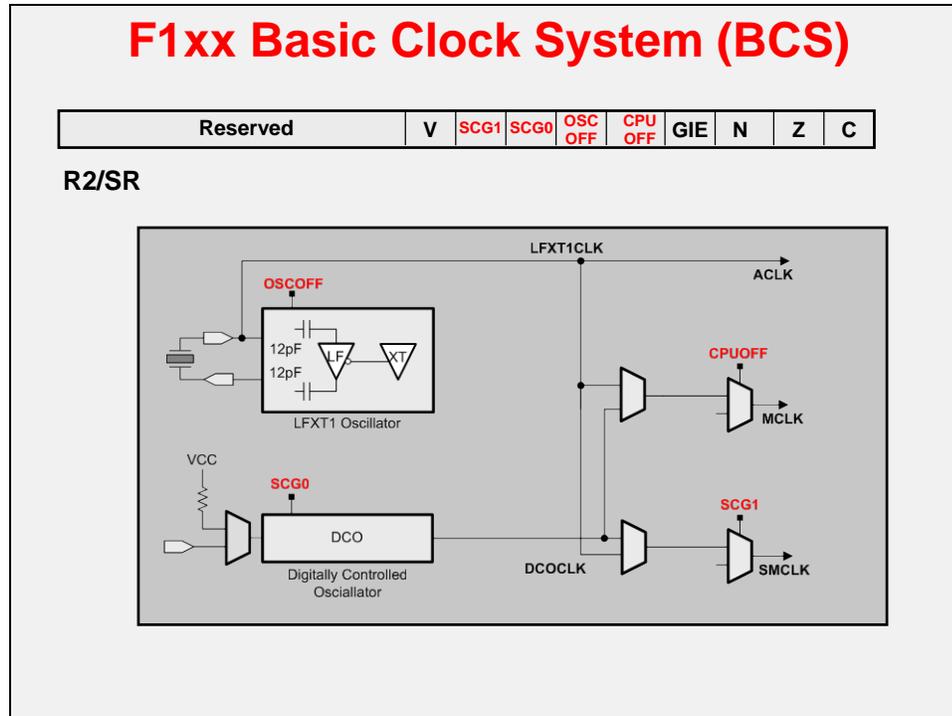
MSP430 Clock Modules		
Module	Clock Module Name	MSP430 Device Family
BCS	Basic Clock System	F1xx / F2xx
BCS+	Basic Clock System +	F2xx / G2xx
FLL+	Frequency Locked Loop +	F4xx
UCS	Unified Clock System	F5xx / F6xx
CS	Clock System	FR5xx
CCS	Compact Clock System	L092

In general, all of these "different" peripherals provide the same basic functionality: that is, they nearly all provide three internal clocks (MCLK, SMCLK, ACLK) from a similar set of oscillator sources.

What differs between them are exactly which sources are provided for a given family, how the DCO frequency is configured and tuned, as well as a number of other miscellaneous clock features. Many of these similarities and differences are described over the next few pages.

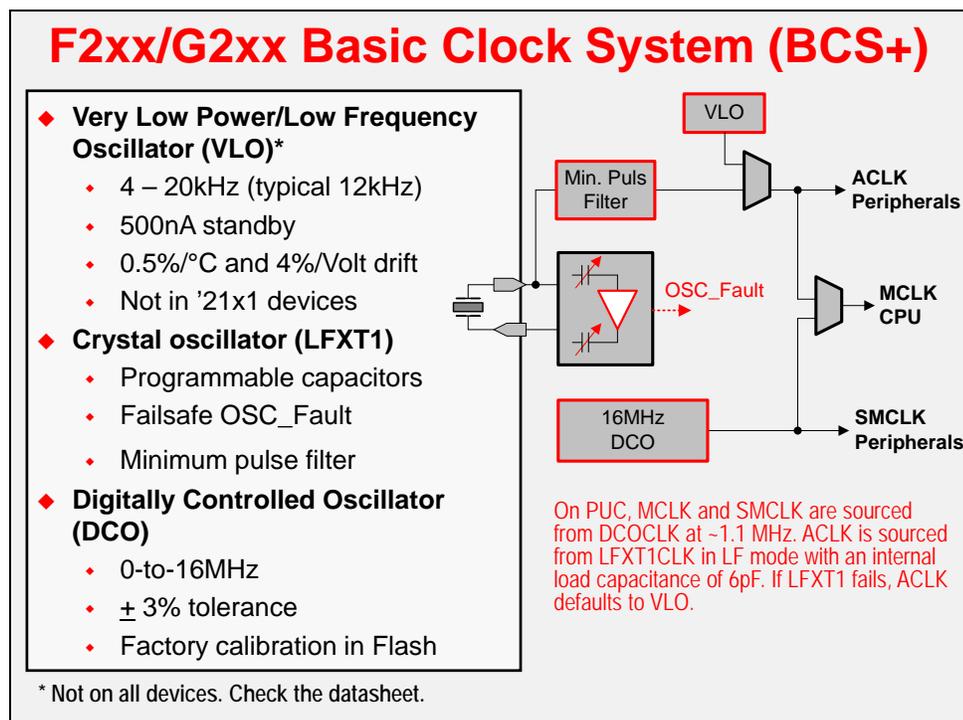
F1xx Clocking (BCS)

These early devices provided the same three internal clocks, but the oscillator sources were quite a bit more limited. Also, the DCO had to be tuned in software if the temperature or voltage changed significantly during operation. (Later devices moved this chore into hardware.)



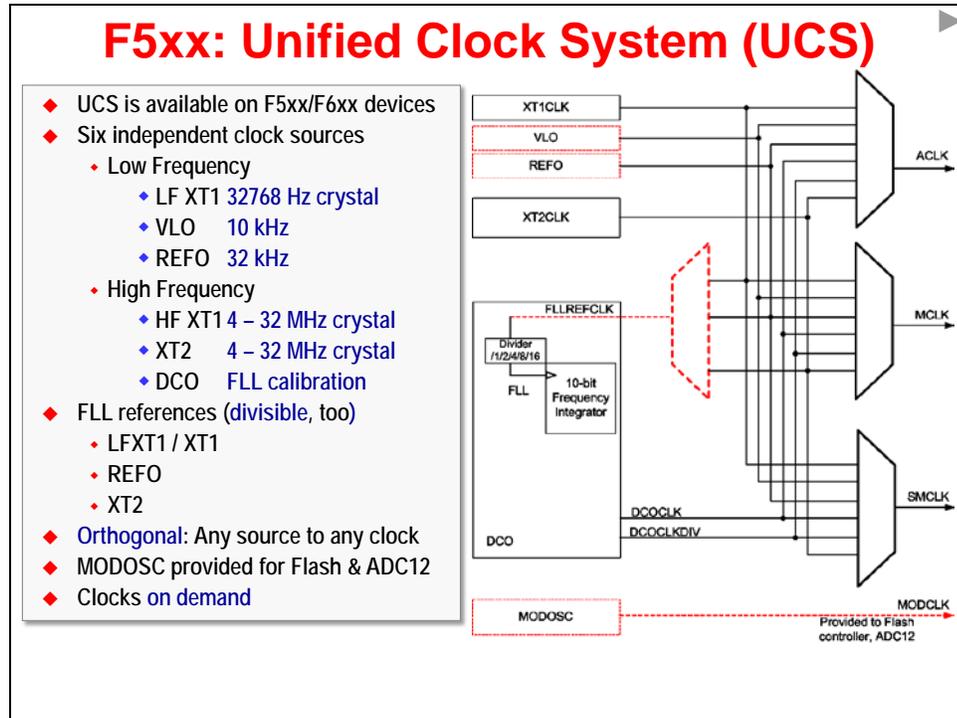
F2xx/G2xx Clocks (BCS+)

Some F2xx devices still utilized the BCS peripheral, but the later devices – as well as the “G” series Value-Line devices – provide users with the enhanced BCS+ peripheral. You’ll find that this clock system has additional source options. Also, the DCO (as well as some other peripherals, such as the ADC) are calibrated during factory testing. Thus, you can get a much higher precision DCO by utilizing the correct calibration values stored in the flash by TI.



F5xx/F6xx Clocks (UCS)

The Unified Clock System is most flexible MSP430 clock peripheral to date. It provides an orthogonal set of clock options – any source can drive any internal clock signal. Additionally, it provides the hardware required to dynamically tune the DCO as needed under varying conditions. (We'll explain later how this works.)



F5xx: Unified Clock System

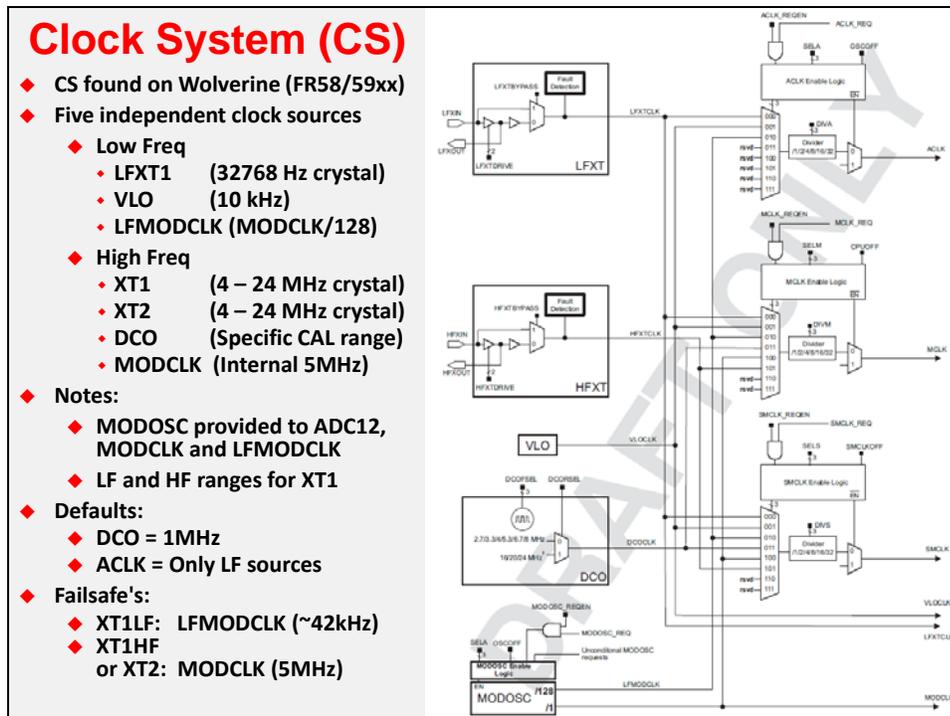
- ◆ **Orthogonal clock system**
 - ◆ Any source can drive any clock signal
- ◆ **2 Integrated clock sources:**
 - ◆ REFO: 32kHz, trimmed osc.
 - ◆ VLO: 12kHz, ultra-low power
- ◆ DCO & FLL provide high frequency accurate timing
- ◆ MODOSC provides bullet proof timing for Flash
- ◆ Crystal pins muxed with I/O function

Main Features:

- ◆ Any OSC can drive any system clock (MCLK, ACLK, SMCLK)
- ◆ Clock divider up to 32 for each system clock
- ◆ Control the CLK in Low Power Modes (stopped or running) and react to module CLK requests
- ◆ OSC enable logic according requests
- ◆ Supporting the FLL as sub-module and providing the control registers
- ◆ MODOSC as Clock source for Flash and ADC

FR58xx/FR59xx - Clock System (CS)

The Clock System (CS) used in the new “Wolverine” devices provides almost as much flexibility as the UCS peripheral, although – as we’ll see later – it’s easier to configure.



Using MSP430ware to Configure Clocking

As we have done with our other peripherals (e.g. GPIO), we can use MSP430ware's DriverLib to configure the clocking options. For example, in the following diagram the UCS_clockSignalInit() function can be used to configure ACLK to use the REFO clock source.

DriverLib – Selecting Clock Sources

```
#include <driverlib.h>

void myClkInit(void) {
    //Set ACLK = REFO
    UCS_clockSignalInit (
        UCS_BASE,
        UCS_ACLK,           // Configure ACLK
        UCS_REFCLK_SELECT, // Set to REFO source
        UCS_CLOCK_DIVIDER_1 // Set clock divider to 1
    );
    ...
}
```

- ◆ Call “clockSignalInit” function for each clock you want to configure
- ◆ Function prefix: UCS_ (F5xx/6xx), CS_ (FR5xx)
- ◆ Exception – we usually configure MCLK for F5xx/6xx using the initFLL function (discussed later)

An earlier clock diagram demonstrated the many places where the clock input frequencies can be divided-down; once again, this provides you with a greater possible clock range. In this code example, we just chose to set the clock divider to 1. Conveniently, the DriverLib API provides an enumeration for each possible field value, including all the various clock divider options. (*DriverLib, with these enumerations, makes the code very easy to read.*)

Using an external clock crystal is a bit more involved than using an internal oscillator source. Before you can configure the clock using the same `UCS_clockSignal_init()` function, you must:

- Setup the XIN/XOUT as clock pins. (On many devices, these pins default to their GPIO modes.)
- The crystal oscillators must be started up before they can be used to source a clock. The clock API provides two start functions: one will not exit until the oscillator has started, while the other one can timeout and return even if the crystal hasn't started running correctly. (If you use the latter, make sure you evaluate its return value.)

DriverLib – Using External Crystal

```
#include <driverlib.h>

//Set XIN (P5.4) and XOUT (P5.5) in Clock mode
GPIO_setAsPeripheralModuleFunctionInputPin(
    GPIO_PORT_P5, GPIO_PIN4);
GPIO_setAsPeripheralModuleFunctionOutputPin(
    GPIO_PORT_P5, GPIO_PIN5 );

//Start the XT1 oscillator, wait until it's running
UCS_LFXT1Start( UCS_BASE, UCS_XT1_DRIVE0, UCS_XCAP_3 );

UCS_clockSignalInit ( UCS_BASE,
    UCS_ACLK,           // Configure ACLK
    UCS_XT1CLK_SELECT, // Set to REFO source
    UCS_CLOCK_DIVIDER_1 ); // Set clock divider to 1
```

- ◆ **Warning:** Verify XIN and XOUT before starting external oscillators! On many devices, these pins are shared with GPIO
- ◆ `UCS_LFXT1StartWithTimeout()` lets the function exit even if the crystal isn't working – make sure you check its return value

As pointed out in the slide, there are two functions that can be used to start each of the crystal oscillator sources: one will continue until the crystal has started (and will run forever); while the other provides a timeout option.

The crystal startup functions provide two arguments for selecting the crystal drive strength and on-chip load capacitance.

- For Low Frequency (LF) crystals, the drive strength option allows you to tune minimize the power needed to drive the crystal; also, you can select on-chip capacitor that meets your crystals requirements. (Additional external capacitors can be added if necessary.)
- For HF crystals, different crystal or resonator ranges are supported by choosing the proper drive settings. In this case, you will need to use external capacitors.

If you choose to use the XT1 (and/or XT2) inputs with an external clock signal on XIN (XT2IN), you need to set them for bypass mode. Conveniently, DriverLib provides UCS functions for putting the interfaces into bypass mode.

The optional lab exercise for this chapter provides a crystal oscillator example for you to explore.

Additional Clock Features

There are a number of additional clock features that are summarized for our three example devices in the following table

Additional Clock Features				
Clock Feature		'G2553 (BCS+)	'F5529 (UCS)	'FR5969 (CS)
Available Clock Sources	MCLK	VLO, LFXT1, XT2, DCO	VLO, REFO, XT1, XT2, DCOCLK, DCOCLKDIV	VLO, LFXT, LFMODCLK, HFXT, MODCLK, DCOCLK
	SMCLK			VLO, LFXT, LFMODCLK
	ACLK	VLO, LFXT1		VLO, LFXT, LFMODCLK
Clock Defaults (at PUC Reset)	MCLK	DCO (1.1MHz)	DCOCLKDIV (1.048 MHz)	DCO (1MHz)
	SMCLK			
	ACLK	LFXT1	XT1CLK (32KHz)	LFXT
External Clk Failsafe		ACLK = VLO S/MCLK = DCO	LF XT1 = REFO HF XT1/XT2 = DCO	LFXT= LFMODCLK (38KHz) HFXT=MODCLK (4.8MHz)
DCO Calibration		Factory Constant	FLL (Run-time)	Factory Trimmed
Password Needed (To change clock settings)		No	No	Yes
Clock Request (Periph can force clk on)		WDT+ only	Yes	Yes

There's quite a bit of information on this table. We'll summarize the features row-by-row.

Available Clock Sources: The various clock oscillator sources were described earlier in this chapter. This table shows which clock sources can be used for MCLK, SMCLK, and ACLK. You might notice that, as we described earlier, the UCS peripheral (found on the 'F5529) allows any source to be used with any of the three clocks.

Clock Defaults: What happens if you do not configure the clock peripheral? As you might expect, at (PUC) reset the three internal clocks default to a specific clock source and rate. These are shown in the table.

External Clock Failsafe: What happens if the external crystal breaks or falls off your board? The MSP430 clocks will default to an internal clock. While this may not be the rate/precision you were expecting to run at, it's better than having the system fail outright. There are clock fault events that indicate if the external clock is not working correctly. (Note: it is expected that the clock will be in a 'fault' state while the crystal is initializing.)

DCO Calibration: As we mentioned earlier – and will discuss in more detail later – different generations of the MSP430 use different methods for calibrating the DCO. The first generation forced you to do this in software; later generations use hardware or pre-calibrated constants.

Password: The latest generation of the MSP430 devices requires a password to modify the clock configuration. The purpose of this is to prevent a software error from accidentally changing the settings.

Clock Request: Some devices, such as the 'F5529, have a "clock request" signal running from their peripherals to the UCS module – these signals *request* that their clock source must remain on. In other words, when this feature is enabled, it prevents you from accidentally turning off a clock that is in use by a peripheral.

For example:

Let's say that you wanted to put the CPU to sleep using Low-Power Mode 3 (LPM3) and wait in that mode until the UART received a byte and created an interrupt.

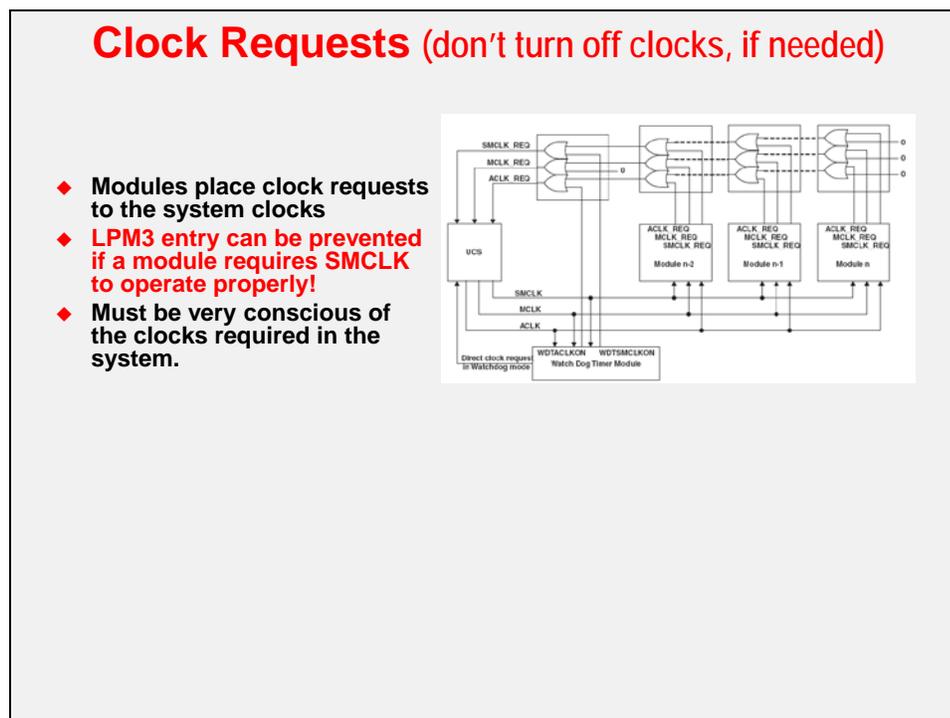
A problem would occur, though, if your UART was being clocked by SMCLK since LPM3 turns off SMCLK. In other words, what happens if the peripheral you were using to wake the processor up just happened to be using that clock, you would never wake up.

The *Clock Request* feature allows a peripheral, such as the UART, to prevent its source clock from being turned off. The CPU will still go into LPM3 mode, but in this case SMCLK would remain on.

The caveat of *Clock Request* is that it affects power dissipation. By preventing a clock from turning off, your processor will consume more power.

On the 'G2553, only the clock being used by the Watchdog (WDT+) cannot be turned off, even if the power mode (LPM) normally turns off that specific clock.

Our other two example devices ('F5529, 'FR5969) use a bit more advanced scheme. That is, additional peripherals can 'request' a clock to remain on, even if a specific LPM normally disables that clock.



Note: While this feature is a handy failsafe, it can also prevent your system from reaching its lowest power state.

Additional Clock Notes/Warnings

Here's an assortment of notes and warnings about the clocks.

Other Clock Notes/Warnings

- ◆ Devices with shared IO's for GPIO and XIN/XOUT:
 - ◆ Configure the XIN/XOUT ports correct, if you forget this the Fault will be still available.
 - ◆ If using a loop or interrupt for clearing the fault flag you will loop forever
- ◆ After clearing the fault flag in the Clock system successfully you need to clear the OFIFG flag inside the SFR as well.
 - ◆ If you don't do this you run always with the failsafe clock. Two stage Fault logic is new for 5xx series
- ◆ If LFXT is disabled when entering into a low-power mode:
 - ◆ It is not fully enabled and stable upon exit from the low-power mode, because its enable time is much longer than the wakeup time.
 - ◆ If the application needs to keep LFXT enabled during a low-power mode, the LFXTOFF bit can be cleared prior to entering the low-power mode which causes LFXT to remain enabled.
 - ◆ Similarly, the HFXTTOFF bit can be cleared prior to entering the low-power mode. This causes HFXT to remain enabled.

DCO Setup and Calibration

Calibrating DCO

Additional Clock Features

Clock Feature		G2553 (BCS+)	F5529 (UCS)	F5969 (CS)
Available Clocks	MCLK	VLO, LFXT1, XT1CLK	VLO, LFXT1, XT1CLK	FXT, LFMODCLK, MODCLK, DCOCLK
Clock Defaults (at PUC reset)	MCLK	DCO (1.1MHz)	DCOCLKDIV (1MHz)	DCO (1MHz)
	ACLK	LFXT1	XT1CLK	LFXT
External Clk Failsafe		ACLK = VLO S/MCLK = DCO	LFXT1 = REFO HFXT1/XT2 = DCO	LFXT = LFMODCLK (42kHz) HFXT = MODCLK (5MHz)
DCO Calibration		Factory Constant	FLL (Run-time)	Factory Trimmed
Password Needed (To change clock settings)		No	No	Yes
Clock Request (Periph can force clk on)		No	Yes	Yes

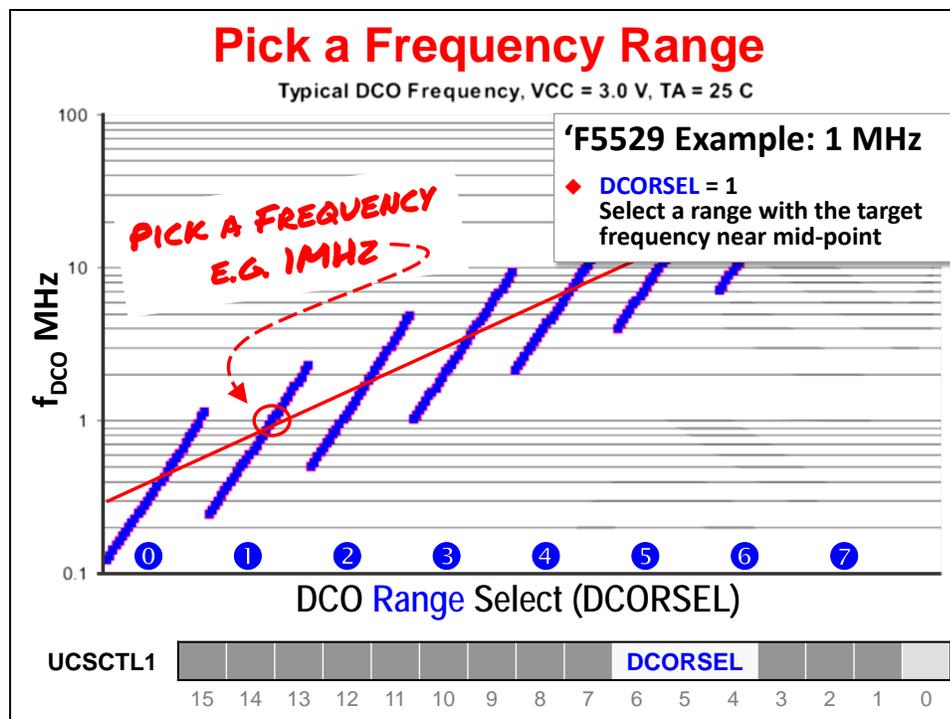
Before we look at the details of calibration, let's start with "How does the DCO work?"

As you can see from our earlier table, the DCO (digitally controlled oscillator) can be calibrated in a variety of different ways, depending upon which generation MSP430 processor you're using. Before discussing these various calibration options, let's first look at how the DCO works.

How the DCO Works

The DCO is configured using three register fields. On most devices they're named: DCORSEL, DCO, and MOD. In the process of discovering how the DCO works, we'll see how each of these fields affects the DCO's output.

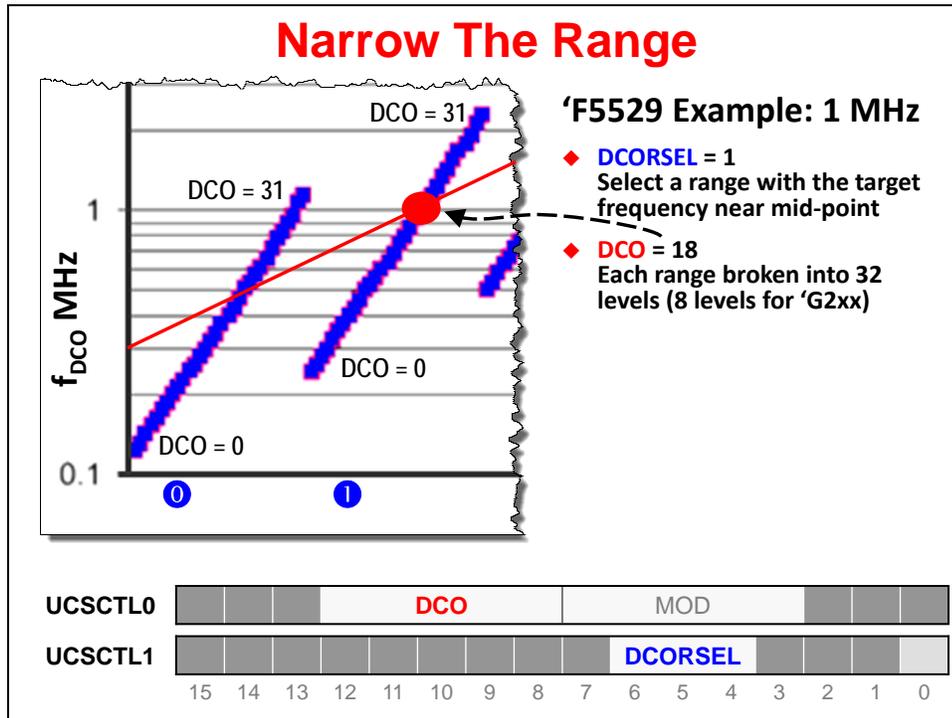
The DCO can operate in a number of different frequency ranges. On the 'F5529, you can select from one of eight different frequency ranges. You might notice that these ranges overlap each other quite a bit. The goal would be to pick a range where your desired frequency sits near the middle. (This is not required, but provides the greatest flexibility - as we'll see in a minute.)



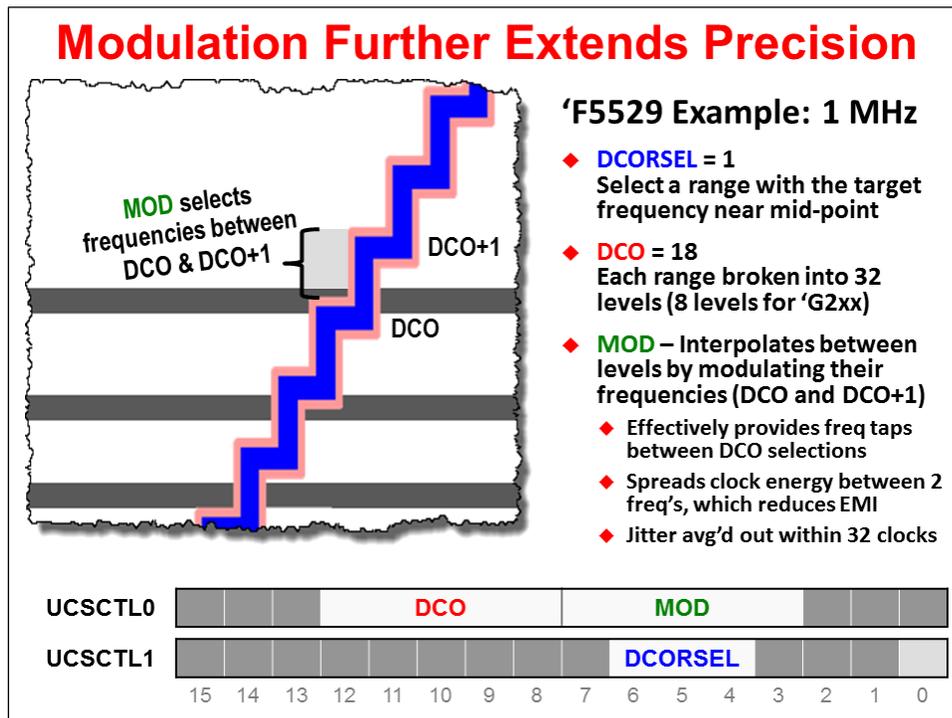
In the diagram above, if we wanted to run at 1 MHz, range "one" happens to be a good choice. Any of the first three would work, but range "1" puts our desired frequency close to the middle of the range.

Notice that the DCORSEL (DCO Range SElect) register field provides a means of selecting which DCO range you want to use.

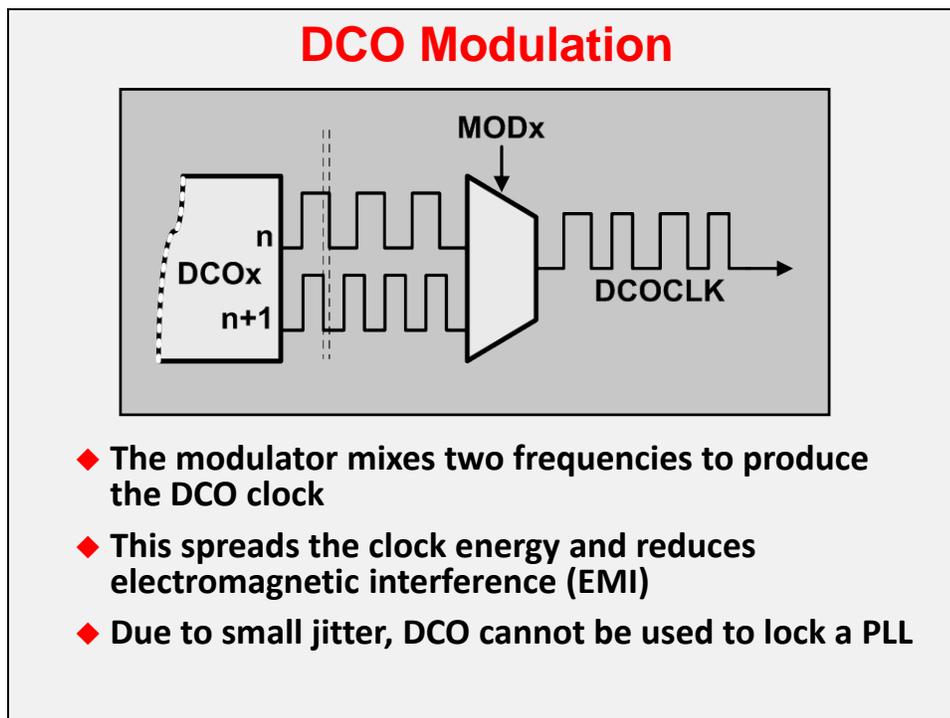
While the DCORSEL allows you to select a range of frequencies, it's the DCO field that allows us to indicate which frequency we desire within that range. On the 'F5529 the DCO field is 5-bits long, which means we're provided 32 different frequency levels in our chosen range.



What happens when the frequency you're interested in falls between two levels specified by the DCO field? In other words, what happens if the granularity of the DCO field is not enough to specify our frequency of interest? (I.E. our frequency falls between a value of DCO and DCO+1.)



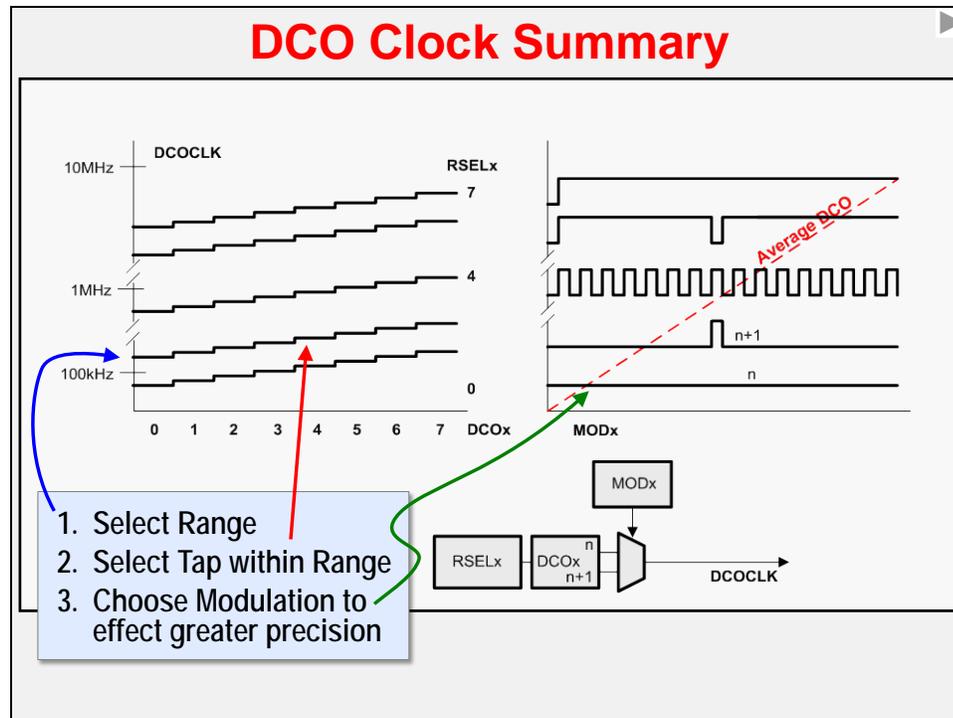
This is where the final field, called MOD, comes into play. MOD lets you tell the MSP430 clock to modulate between two frequency levels: DCO and DCO+1. By mixing these two frequencies you can obtain a very close approximation to your chosen clock frequency.



Naturally, you will probably configure DCO and MOD (and DCORSEL) during system initialization (probably early in your main() function). If the temperature or input voltage varies over time, though, you will likely want to tweak (i.e. tune) DCO and MOD to compensate for your systems changing environment. On older MSP430 devices, these tweaks had to be done in software; on later devices, hardware was added to automate this task for you. We'll look at these tuning options in the next section of the chapter.

DCO Summary

Here's a summary of the DCO features we just discussed – the graphic is just drawn a little differently. In essence, you must: (1) pick a range; (2) select a level within the range; and (3) pick a modulation scheme that allows you to interpolate between adjacent ranges, as needed.



Factory Calibration (G2xx, FR5xx)

The Value-Line ('G2xx) and Wolverine ('FR5xxx) devices use static, pre-calibrated settings, chosen during device testing, to allow your DCO to meet the frequencies and tolerances specified in the device datasheet.

FR5xx DCO – Calibrated Frequencies

	DCORSEL	DCOFSEL	DCO (MHz)
◆ Clock System (CS) module found on FR5xx devices	0 or 1	000	1
◆ DCO (CS module) provides multiple pre-defined & calibrated frequencies	0	001	2.667
	0	010	3.333
	0	011	4
◆ Factory Trimmed Accuracy: ±2% from 0-50C ±3.5% from -40 to 85C	0/1	100/001	5.33
	0/1	101/010	6.67
◆ FR5xx CS module requires psw to write clock reg's	Ex: 0/1	110/011	8
◆ *If DCOCLK = 20 or 24MHz it must be divided down for MCLK	1	100	16
	1	101	20*
	1	110	24*

CSCTL1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
										DCORSEL						

```
// Set DCO to 8MHz
CS_setDCOFreq(CS_BASE, CS_DCORSEL_1, CS_DCOFSEL_3);
```

Configuration of the 'FR5xxx devices is the easiest of all the MSP430 devices. Looking at the table in the datasheet (which has been replicated above), you just need to choose the value of the DCORSEL and DCOFSEL fields to match the frequency you want to run at. The silicon is trimmed at the factory so that the device meets the accuracy specified in the datasheet.

The 'G2xxx Value-Line devices take a slightly different approach. Rather than trimming the silicon, as is done with the 'FR5xxx devices, the factory stores calibration values into each device's Flash memory (INFOA section) during device test.

'G2xxx DCO – Calibration Constants

DCO Calibration Data (provided from factory in flash info memory segment A)			
DCO Frequency	Calibration Register	Size	Address
1 MHz	CALBC1_1MHz	byte	010FFh
	CALDCO_1MHz	byte	010FEh
8 MHz	CALBC1_8MHz	byte	010FDh
	CALDCO_8MHz	byte	010FCh
12 MHz	CALBC1_12MHz	byte	010FBh
	CALDCO_12MHz	byte	010FAh
16 MHz	CALBC1_16MHz	byte	010F9h
	CALDCO_16MHz	byte	010F8h

- ◆ Most G2xx devices provide pre-calibrated clock settings – applying these sets the Range, DCO, and MCO values
- ◆ Clock (and ADC) calibration values are calculated at the factory and stored into Flash memory (INFOA)
- ◆ G2xx1 provide 1MHz calibration; G2xx2/3 provides all 4 frequencies

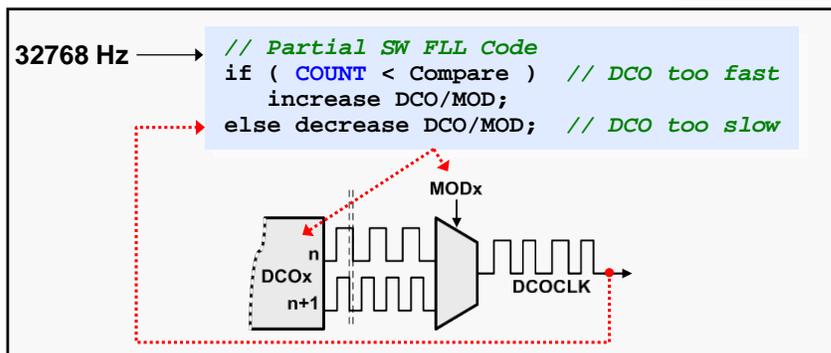
```
// Setting the DCO to 1MHz
if (CALBC1_1MHZ ==0xFF || CALDCO_1MHZ == 0xFF)
while(1); // Erased calibration data? Trap!
BCSCTL1 = CALBC1_1MHZ; // Set range
DCOCTL = CALDCO_1MHZ; // Set DCO step + modulation
```

Basically, the device tester measures the silicon to determine what value of DCO and MOD is required to run the DCO at a set of pre-determined frequencies. These calibration values are stored into INFOA memory by the tester. You can then copy the appropriate calibration constant from Flash into your DCO control register to run the clock at a specified frequency.

Runtime Calibration

The MSP430F5xx series (along with the 'F4xx and 'F6xx) of processors can perform dynamic calibration of the Digitally Controlled Oscillator (DCO) using the Frequency-Locked Loop (FLL) hardware built into the Unified Clock System (UCS).

Dynamic Calibration of DCO in Software

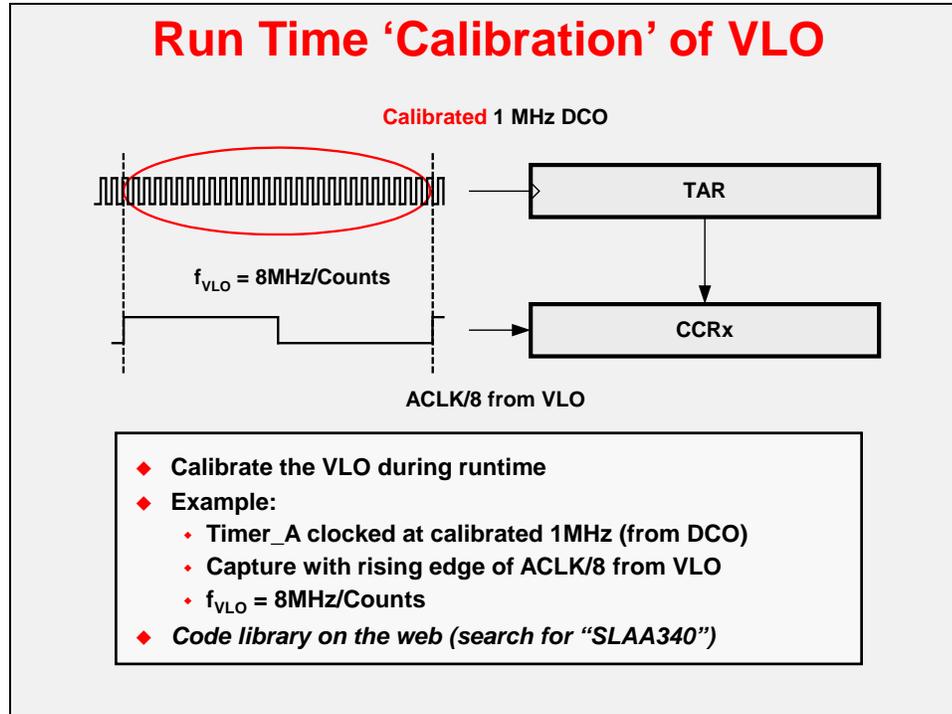


- ◆ **Minimize frequency drift** due to changes in voltage or temperature
 - DCO clock precision is achieved by periodic adjustment in loop
 - Modify settings (DCO, MOD) in loop based upon comparison of DCO to another known/stable freq, such as 32kHz crystal (or 50/60Hz AC power)
- ◆ **Frequency Locked Loop (FLL) – ‘lock’ one frequency to another**
 - Software FLL is the only option available on 'F1xx devices
 - While software FLL could be used for any MSP430 device, the F4xx/5xx/6xx clock modules contain Hardware FLL circuitry

In earlier MSP430 processors, this needed to be handled in software. Using the FLL, the Modulation (MOD) parameter (i.e. field of the DCO control register) is adjusted up or down based upon the count of DCO cycles versus an accurate reference clock (most commonly, a 32KHz watch crystal).

VLO 'Calibration'

The app note and library mentioned on the slide below can be used to calibrate the VLO clock at runtime. While originally not known for its high accuracy, the VLO can be 'calibrated' using another clock. The example shown here uses the DCO and TIMER_A to calibrate the VLO.



Other Initialization (WDT, PMM)

When starting up a system, there are a number of elements that must be initialized. Here's a generic summary detailing these items.

Software Initialization				
	Initialization Step	Required Action?	Who is Responsible	Where Discussed
1.	Initialize the stack pointer (SP)	Yes	Compiler	N/A
2.	Initialize <u>watchdog timer</u> (usually OFF when debugging)	Yes	User	Chapter 4
3.	Setup Power Manager & Supervisors	No	User	Chapter 4
4.	Configure GPIO pins	No	User	Chapter 3
5.	Reconfigure clocks (if desired)	No	User	Chapter 4 (earlier)
6.	Configure peripheral modules	No	User	Later chapters

The **Stack Pointer** must be initialized but the compiler does this for us, which is why we don't directly discuss this in this workshop.

As discussed many times already in this workshop, since the **Watchdog Timer** defaults to "ON", it must be configured. During development and debugging we usually turn it off. The next section discusses the Watchdog in further detail.

Some of the more feature-rich series of the MSP430 devices contain an on-chip **LDO** along with **Power Manager** and **Supervisor** circuitry. If these features exist on your chosen device, you will probably want to configure them. This is discussed later in this chapter.

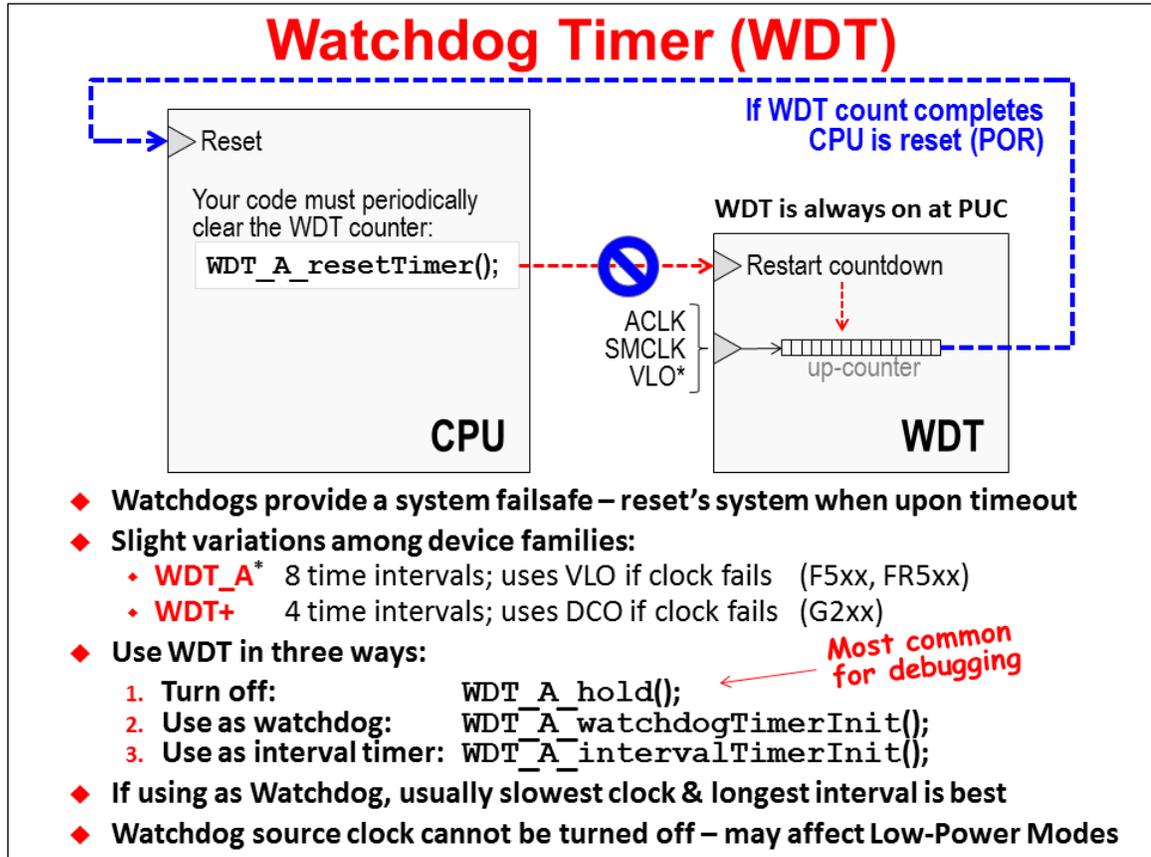
In the last chapter we discussed and used **GPIO pins** (general purpose bit I/O). It highly recommended that you configure all GPIO pins on your device. Obviously, those being used need to be configured, but you should also configure those pins not in use so as to minimize power dissipation.

Earlier in this chapter we discussed the many, varied **clock options** for the MSP430 devices. Unless the default clock options are exactly what you need for your system, these need to be configured.

Finally, you will need to setup and configure the remaining **peripherals** that will be used in your application. We won't try to list them all here – and they vary based upon the selected device – but this is usually handled in `main()` before starting your `while{}` loop.

Watchdog Timer

Watchdog Timers provide a system failsafe; if their counter ever rolls over (back to zero) they reset the processor. To prevent your system from continuously resetting itself, you should clear the counter at regular intervals. The goal here is to prevent your system from locking-up due to some unexpected fault.



As mentioned frequently in this class, the MSP430 watchdog timer is “on” by default. You should always disable the watchdog or configure it as needed.

The preceding slide describes three ways to utilize this peripheral:

1. Turn it off – which is useful while developing or debugging your application. You can use the MSP430ware DriverLib “hold” function to accomplish this.
2. Use the Watchdog for its intended function. Again, the provided DriverLib function can be used to perform this initialization.
3. Finally, if you do not need a watchdog for your system, you could re-purpose the peripheral as a generic interval timer. Used this way, for example, you might setup the timer to create periodic interrupts.

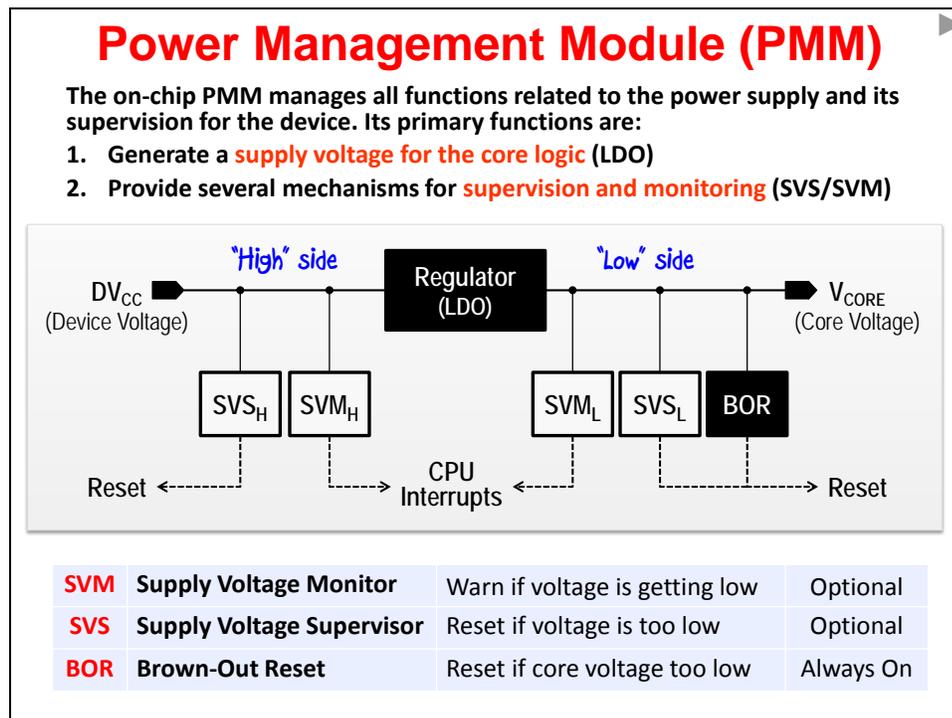
Note: As discussed earlier in this chapter, the clock being actively used by the Watchdog timer cannot be disabled. Keep this in mind as you plan out your system and calculate its power requirements.

PMM with LDO, SVM, SVS, and BOR

The power management module (**PMM**) integrates a number of power supply features that may help you minimize external power supply hardware – and cost.

From the diagram below, you can see that we've drawn the **LDO** (low dropout voltage regulator) right in the center of the diagram. This is to drive home the idea that it's a central feature of the PMM. The LDO will provide a regulated, stable voltage to the CPU core from the device voltage applied to the DV_{CC} pins. The device user's guide defines the following nomenclature (as shown below):

- **High Side:** unregulated voltage
- **Low Side:** regulated voltage

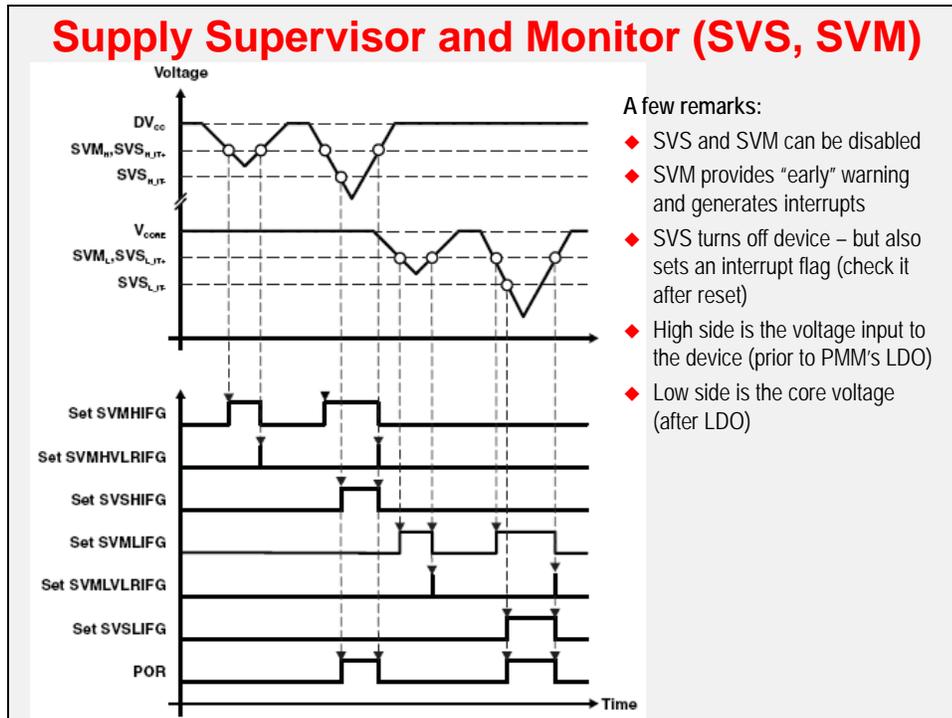


The **SVM** (supply voltage monitor) circuitry is intended to warn you (via interrupts) when the high- or low-side voltages are getting close to their lower limits. You might use this to correct the power supply or prepare for a power error/shutdown. (You can choose not to use this feature if you want to save the small amount of power it consumes.)

The **SVS** (supply voltage supervisor) is another step further in supervision (vs SVM). The SVS actually forces a reset if the high- or low-side voltages fall too low. This helps to prevent possible errors from running the CPU out-of-spec. (You can choose not to use this feature if you want to save the small amount of power it consumes.)

The **BOR** (brown-out reset) circuitry is found on every MSP430 device. You might remember us talking about this hardware at the beginning of the chapter. In a sense, it is redundant to the SVS_L circuitry, although it is always on – and consumes very little power.

The following diagram may help you visualize how the Supply Supervisors work:

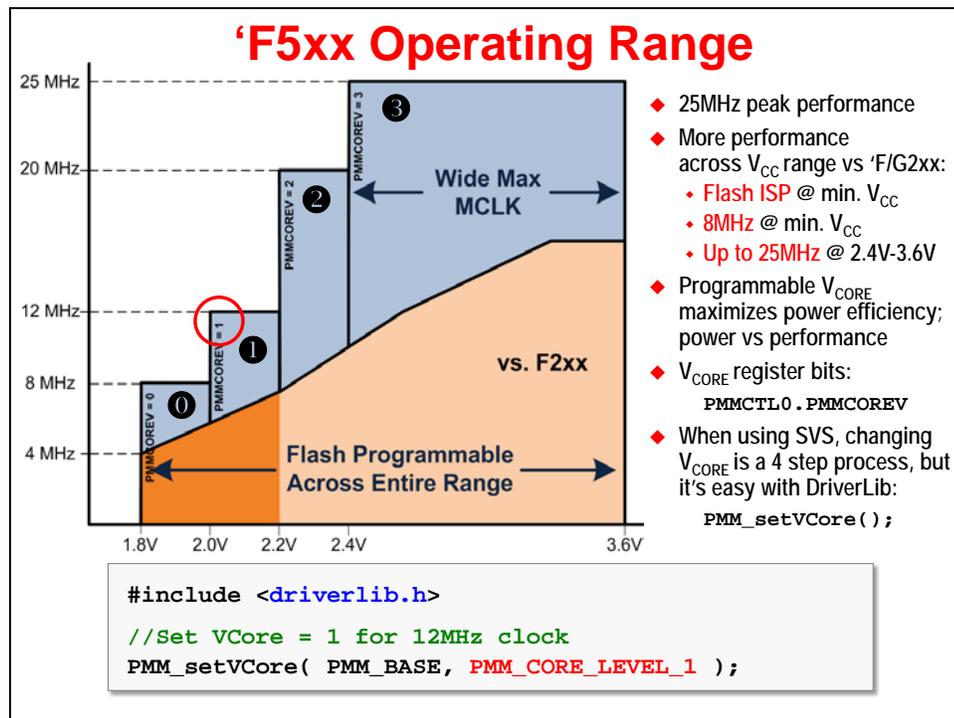


Operating Voltages

For many of the MSP430 devices, their capabilities can vary based upon the input voltage supply. For example, most of the devices do not support in-system Flash programming when running below 2.2V. Another example is that many devices require higher voltages to run at their faster speeds.

Two examples of this are shown below:

- The 'F2xx and 'G2xx devices require 2.2V in order to perform in-system flash programming. Also, their frequency is proportional to the input voltage
- The F5529 can operate at any one of four voltage ranges. You would need to choose the input voltage range appropriate for the speed you want to run. For example, if you want to run at 10MHz you could run at power mode 1, but 25MHz requires power mode 3. On the other hand, the 'F5529 can program its flash memory across the entire input voltage range.



The advantage to running with lower power voltage settings is that you, well, save power. The tradeoff is that you give up capability when you run at the lower settings. Then again, you could always change the Vcore setting on-the-fly, as needed by your application at any given time.

One big advantage of the new Wolverine devices (e.g. 'FR5969) is that they can program their FRAM and run at full speed, even when running at their lowest input voltage. This really helps them to minimize power while providing you with maximum convenience.

Summary

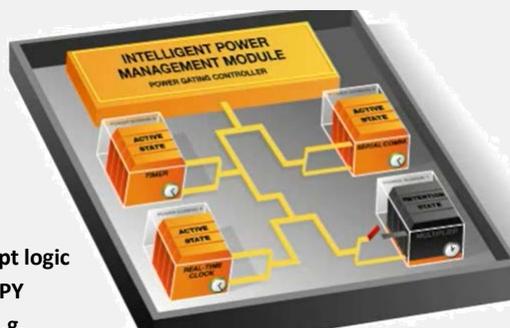
We have summarized three MSP430 devices in the table below. They demonstrate some of the differences between the various series of MSP430: Value-Line, F5xx, and Wolverine FR5xxx.

Power Management Summary			
	G2553	F5529	FR5969
Input Voltage (DV_{CC})	1.8 - 3.6 Volts	1.8 - 3.6 Volts	1.8 - 3.6 Volts
Internal Regulators (LDO)	None	3 LDO's (LP, HP, USB)	4 LDO's (LP, HP, RTC, FRAM)
# of V_{CORE} Levels (Configuration)	N/A	4 Power Levels (Manual)	Intelligent Power (Automatic)
Speed affected by Input Voltage	Yes 1.8V: up to 6MHz 3.3V: up to 16MHz	Yes 1.8V: up to 8MHz 2.4V: up to 25MHz	No All speeds available over entire range
Flash/FRAM Voltage (In-System Programming)	2.2 V and above	Full Range	Full Range
Brown-Out Reset (BOR)	Yes	Yes	Yes
Power Supervisor (SVS)	F2xx (but not G2xx)	Yes	Yes
Power Monitor (SVM)	No	Yes	Yes
I/O protection (LOCKLPM5)	No	Yes	Yes

The following two slides provide backup information. The first shows the advanced power-gating found in the Wolverine devices...

Wolverine Power Gating ('FR58/59)

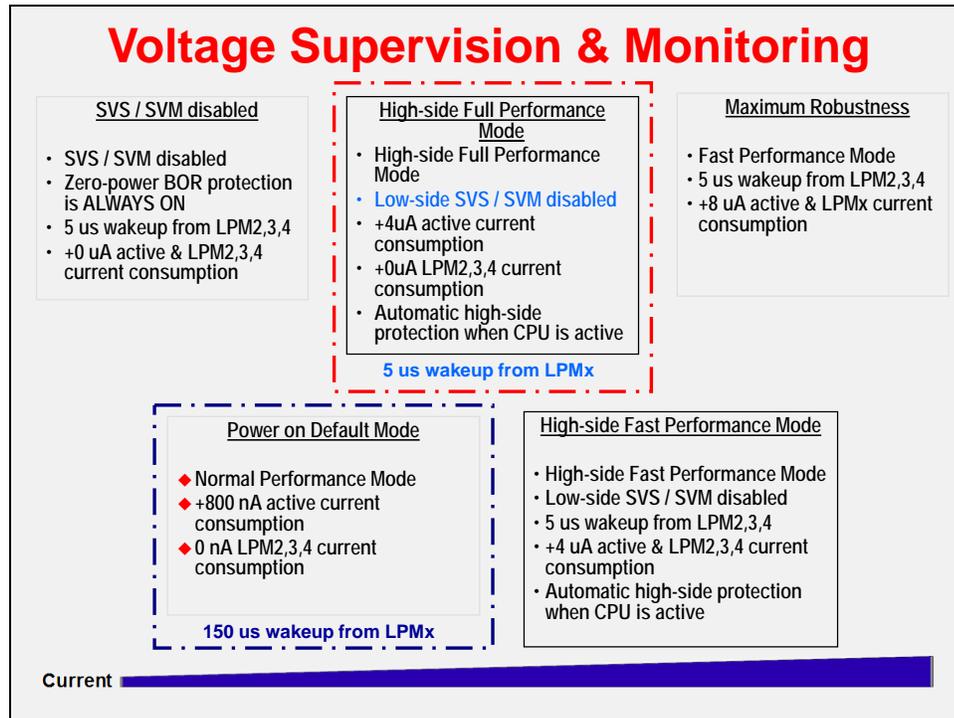
- ◆ Enhanced clock system
- ◆ Each module has a clock enable line
- ◆ If CE line is not in use the domain is powered down



Domain 1: Always ON CPU, Interrupt logic
Domain 2: Always OFF, AES, HW MPY
Domain 3/4: Peripheral Domain for e.g. timers

Completely transparent to the user

This slide shows a bit more information regarding the voltage supervision/monitoring.



Initialization Summary (template)

To some of you the following template may seem obvious, but we thought it might be handy to provide a template, of sorts, for a main() function in an MSP430 program.

Summary: Initializing MSP430

```
#include <driverlib.h>

void main(void) {
    // Setup/Hold Watchdog Timer (WDT+ or WDT_A)
    initWatchdog();

    // Configure Power Manager and Supervisors (PMM)
    initPowerMgmt();

    // Configure GPIO ports/pins
    initGPIO();

    // Setup Clocking: ACLK, SMCLK, MCLK (BCS+, UCS, or CS)
    initClocks();

    //-----
    // Then, configure any other required peripherals and GPIO
    ...

    while(1) {
        ...
    }
}
```

Notice that there are function calls provided for many of the initialization steps discussed in this chapter. Of course, it's up to you to provide the necessary code for each of these functions. The following lab exercises will provide some examples of these functions – which we'll continue to build upon in future chapters.

Lab 4 - Abstract

Lab 4 explores a variety of initialization tasks; the largest one being to setup the clocks for the MSP430.

Lab 4 – Clocks & Init

◆ Initialize the Lab with a Worksheet:

- ◆ Clock setup
- ◆ DCO setup
- ◆ Watchdog configuration

◆ Lab 4a – Program MSP430 Clocks

- ◆ Program MCLK, SMCLK, and ACLK
- ◆ Evaluate using 'get' clock rate functions

Extra Credit:

◆ Lab 4b – Exploring the Watchdog Timer

- ◆ What happens if the WDT times-out?

◆ Lab 4c – Utilizing Crystals

- ◆ Configure SMCLK using the external high-speed crystal
- ◆ Configure ACLK using the off-chip external 'watch' crystal



Time:

Worksheet – 15 mins

Lab 4a – 30 mins

This lab also starts off with a worksheet where we will answer a number of questions (and write a little code) that will be used in the upcoming lab procedure.

Lab 4a – Program MSP430 Clocks

We explore the default clock rates for each of MSP430's three internal clocks; then, set them up with a set of specified clock rates.

(Extra) Lab 4b – Blink LED with Different Clocks

If you have time, this lab provides an opportunity to explore the Watchdog Timer.

(Extra) Lab 4C – Utilizing Crystals as Clock Sources

Once again, if you have time, this lab gives us a chance to configure our system to use the external crystal oscillators found on the Launchpad.

Lab Topics

MSP430 Clocks & Initialization	4-40
<i>Lab 4 - Abstract</i>	4-41
<i>Lab 4 Worksheet</i>	4-43
Hints:	4-43
Reset and Operating Modes & Watchdog Timers	4-43
Power Management	4-43
Clocking.....	4-43
<i>Lab 4a – Program the MSP430 Clocks</i>	4-47
File Management	4-47
Do Clock Code	4-47
Initialization Code - Three more simple changes.....	4-51
Debugging the Clocks.....	4-52
Extra Credit (i.e. Optional Step) – Change the Rate of Blinking.....	4-54
<i>(Optional) Lab 4b – Exploring the Watchdog Timer</i>	4-55
First, a couple of Questions	4-55
Play with last lab exercise	4-55
File Management	4-56
Edit the Source File.....	4-56
Keep it Running.....	4-58
Extra Credit – Try DriverLib’s Watchdog Example (#3).....	4-59
<i>(Optional) Lab 4c – Using Crystal Oscillators</i>	4-60
File Management	4-60
Modify GPIO.....	4-61
Debug.....	4-62
<i>Chapter 04 Appendix</i>	4-63

Lab 4 Worksheet

Hints:

- The MSP430 DriverLib Users Guide will be useful in helping to answer these workshop questions. Find it in your MSP430ware DriverLib doc folder:
e.g. `\MSP430ware_1_70_00_28\driverlib\doc\`

- Maybe even more helpful is to reference the actual DriverLib source code – that is, the .h/.c files for each module you are using. For example:
`\MSP430ware_1_70_00_28\driverlib\driverlib\MSP430F5xx_6xx\ucs.h`

- Finally, we recommend you also reference the DriverLib UCS example #4:

```
\msp430\MSP430ware_1_70_00_28\driverlib\examples\MSP430F5xx_6xx\ucs\ucs_ex4_XTSourcesDCOInternal.c
```

Reset and Operating Modes & Watchdog Timers

1. Name all 3 types of resets:

2. If the Watchdog (WDT) times out, which reset does it invoke?

3. Write the DriverLib function that stops (halts) the watchdog timer:

```
_____ ( WDT_A_BASE );
```

Power Management

F5529

4. (**F5529 Launchpad users only**) Write the DriverLib function that sets the core voltage needed to run MCLK at 8MHz.

```
_____ ( _____ );
```

Clocking

5. Why does MSP430 provide 3 different types of internal clocks?

Name them:

6. What is the speed of the crystal oscillators on your board?

(Hint: look in the Hardware section of the Launchpad Users Guide.)

```
#define LF_CRYSTAL_FREQUENCY_IN_HZ _____
#define HF_CRYSTAL_FREQUENCY_IN_HZ _____
```

7. What function specifies these crystal frequencies to the DriverLib?

(Hint: Look in the MSP430ware DriverLib User's Guide – "UCS or CS chapter".)

```
_____ (
    LF_CRYSTAL_FREQUENCY_IN_HZ ,
    HF_CRYSTAL_FREQUENCY_IN_HZ );
```

8. What speed are the clocks running at? There's an API for that...

Write the code that returns your current clock frequencies:

```
uint32_t myACLK = 0;
uint32_t mySMCLK = 0;
uint32_t myMCLK = 0;

myACLK = _____ ();
mySMCLK = _____ ();
myMCLK = _____ ();
```

Refer to clocking section of
DriverLib User's Guide

9. We didn't setup the clocks (or power level) in our previous labs, how come our code worked?

Don't spend too much time pondering this, but what speed do you think each clock is running at before we configure them?

ACLK: _____ SMCLK: _____ MCLK: _____

10. Setup ACLK:

- Use **REFO** for the F5529 device
- Use **VLO** for the FR5969 device

```
// Setup ACLK
_____ (
    _____ _ACLK, // Clock to setup
    _____, // Source clock
    _____ _CLOCK_DIVIDER_1 );
```

F5529

11. **(F5529 User's only)** Write the code to setup MCLK. It should be running at 8MHz using the DCO+FLL as its oscillator source.

```
#define MCLK_DESIRED_FREQUENCY_IN_KHZ _____

#define MCLK_FLLREF_RATIO _____/(UCS_REFOCLK_FREQUENCY/1024 )

// Set the FLL's clock reference clock to REFO
_____(
    UCS_FLLREF,           // Clock you're configuring
    _____,         // Clock Source
    UCS_CLOCK_DIVIDER_1 );

// Config the FLL's freq, let it settle, and set MCLK & SMCLK to use DCO+FLL as clk source
_____(
    MCLK_DESIRED_FREQUENCY_IN_KHZ,
    _____);
```

Hint: There's a discussion slide very similar to this question

FR5969

12. **(FR5969 Users only)** Write the code to setup MCLK. It should be running at 8MHz using the DCO as its oscillator source.

```
// Set DCO to 8MHz
CS_setDCOFreq(
    _____, // Set Frequency range (DCOR)
    _____ // Set Frequency (DCOF)
);

// Set MCLK to use DCO clock source
_____(
    _____,
    _____,
    UCS_CLOCK_DIVIDER_1 );
```

Check your answers against ours ... see the Chapter 4 Appendix

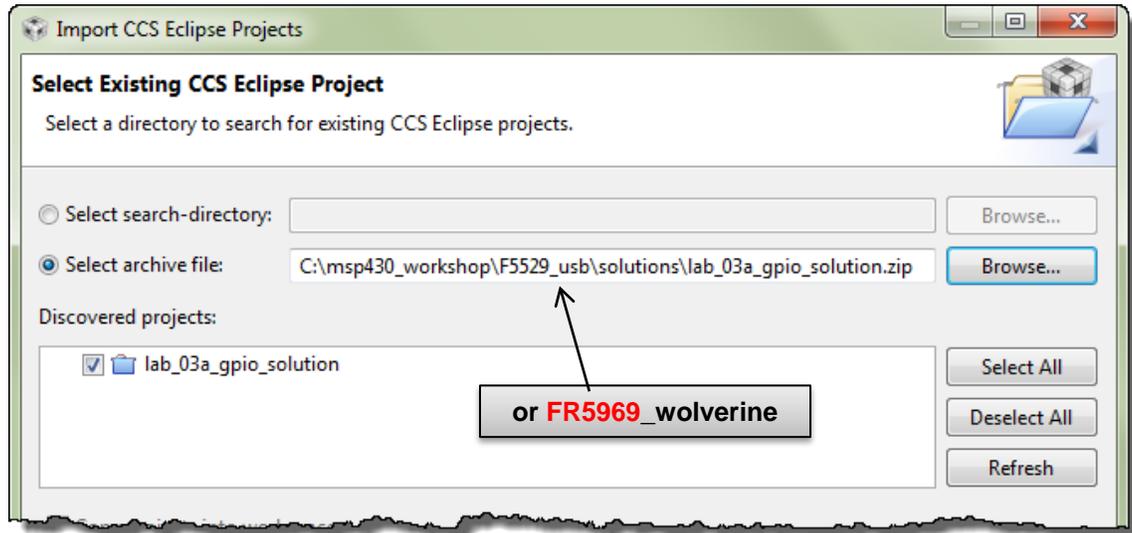
Notes:

Lab 4a – Program the MSP430 Clocks

File Management

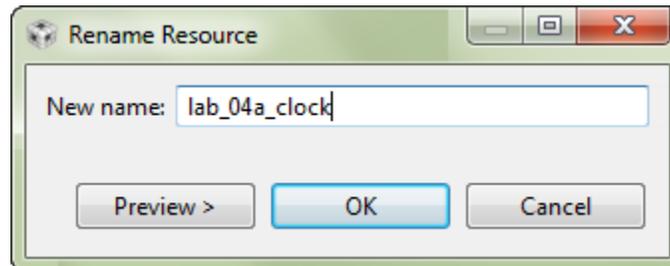
1. Import previous lab_03a_gpio solution.

Project → Import Existing CCS Eclipse Project



2. Rename the project to: lab_04a_clock

Right-Click on Project → Rename



3. Build it, just to make sure the import went without errors.

Do Clock Code

4. Add myclocks.c into the project (from the lab_04a_clock folder).

Since there can be quite a few lines of code (if you setup all the clocks), we decided to place the clock initialization into its own file.

Right-click on project → Add Files...

C:\msp430_workshop*<target>*\lab_04a_clock\myClocks.c

You might notice, the myClocks.c file is missing some code. We'll fix this in the next step...

F5529

5. ('F5529 only) Update `myclocks.c` – adding answers from the worksheet

Fill in the blanks with code you wrote on the worksheet.

Worksheet Question #6

Worksheet Question #11

Worksheet Question #7

Worksheet Question #8

Worksheet Question #10

Worksheet Question #11

```

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

/***** Defines *****/
#define LF_CRYSTAL_FREQUENCY_IN_HZ _____
#define HF_CRYSTAL_FREQUENCY_IN_HZ _____

#define MCLK_DESIRED_FREQUENCY_IN_KHZ _____
#define MCLK_FLLREF_RATIO _____/(UCS_REFOCLK_FREQUENCY/1024)

/***** Global Variables *****/
uint32_t myACLK = 0;
uint32_t mySMCLK = 0;
uint32_t myMCLK = 0;

/***** Functions *****/
void initClocks(void) {

    // Initialize the XT1 and XT2 crystal frequencies being used
    // so driverlib knows how fast they are
    _____
    _____
    _____

    // Verify if the default clock settings are as expected
    myACLK = UCS_getACLK();
    mySMCLK = UCS_getSMCLK();
    myMCLK = UCS_getMCLK();

    // Setup ACLK to use REFO as its oscillator source
    UCS_clockSignalInit(
        UCS_ACLK, // Clock you're configuring
        _____, // Clock source
        UCS_CLOCK_DIVIDER_1 // Divide down clock source
    );

    // Set the FLL's clock reference clock source
    UCS_clockSignalInit(
        UCS_FLLREF, // Clock you're configuring
        _____, // Clock source
        UCS_CLOCK_DIVIDER_1 // Divide down clock source
    );

    // Configure the FLL's frequency and set MCLK & SMCLK to use the FLL
    UCS_initFLLSettle(
        MCLK_DESIRED_FREQUENCY_IN_KHZ, // MCLK frequency
        _____, // Ratio between MCLK and
        // FLL's ref clock source
    );

    // Verify that the modified clock settings are as expected
    myACLK = UCS_getACLK();
    mySMCLK = UCS_getSMCLK();
    myMCLK = UCS_getMCLK();
}

```

FR5969

6. ('FR5969 only) Update myclocks.c – adding answers from the worksheet

Fill in the blanks with code you wrote on the worksheet.

Worksheet
Question #6

Worksheet
Question #7

Worksheet
Question #8

Worksheet
Question #10

Worksheet
Question #12

Worksheet
Question #12

```

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

/***** Defines *****/
#define LF_CRYSTAL_FREQUENCY_IN_HZ _____
#define HF_CRYSTAL_FREQUENCY_IN_HZ 0

/***** Global Variables *****/
uint32_t myACLK = 0;
uint32_t mySMCLK = 0;
uint32_t myMCLK = 0;

/***** Functions *****/
void initClocks(void) {

    // Initialize the LFXT and HFXT crystal frequencies being used
    // so driverlib knows how fast they are
    _____
    _____

    // Verify if the default clock settings are as expected
    myACLK = CS_getACLK();
    mySMCLK = CS_getSMCLK();
    myMCLK = CS_getMCLK();

    // Setup ACLK to use VLO as its oscillator source
    CS_clockSignalInit(
        CS_ACLK, // Clock you're configuring
        _____, // Clock source
        CS_CLOCK_DIVIDER_1 // Divide down clock source
    );

    // Set DCO to 8MHz
    CS_setDCOFreq(
        CS_DCORSEL_1, // Set Frequency range (DCOR)
        CS_DCOFSEL_3 // Set Frequency (DCOF)
    );

    // Set SMCLK to use the DCO clock
    CS_clockSignalInit(
        CS_SMCLK, // Clock you're configuring
        _____, // Clock source
        CS_CLOCK_DIVIDER_1 // Divide down clock source
    );

    // Set MCLK to use the DCO clock
    CS_clockSignalInit(
        CS_MCLK, // Clock you're configuring
        _____, // Clock source
        CS_CLOCK_DIVIDER_1 // Divide down clock source
    );

    // Verify that the modified clock settings are as expected
    myACLK = UCS_getACLK();
    mySMCLK = UCS_getSMCLK();
    myMCLK = UCS_getMCLK();
}
    
```



7. Try building to see if there are any errors.

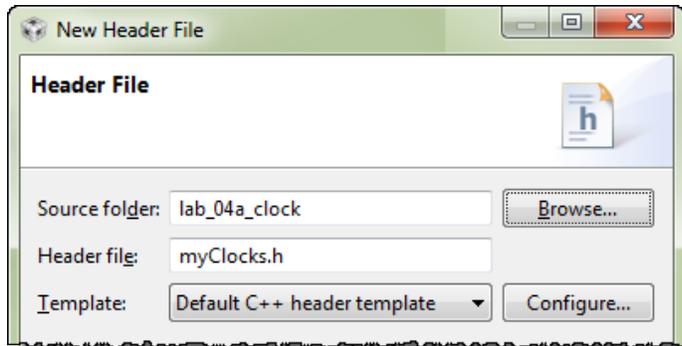
Hopefully you don't have any typographic or syntax errors, but you should see this error:

fatal error #1965: cannot open source file "myClocks.h"

Since we placed the clock function into another file, we should use a header file to provide an external interface for our code.

8. Create a new source file called myclocks.h.

File → New → Header File

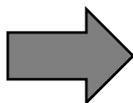


Then click 'Finish'.

9. Add prototype to new header file.

CCS automatically creates a set of #ifndef statements, which are good practice to use inside of your header files. It helps to keep items from accidentally being defined more than once – which the compiler will complain about.

All we really need in the header file is the prototype of our initClocks() function:



```

/*
 * myClocks.h
 */

#ifndef MYCLOCKS_H
#define MYCLOCKS_H

//***** Prototypes *****
void initClocks(void);

#endif /* MYCLOCKS_H_ */
    
```

10. Add reference to myclocks.h to your main.c.

While we're working with this header file, it's a good time to add a #include to it at the top of your main.c. Otherwise, you will get a warning later on.



11. Try building again. Keep fixing errors until they're all gone.

Initialization Code - Three more simple changes

12. Use the simple initialization “template” to organize your setup code.

We’ve outlined the 3 areas you will need to adapt to create a little better code organization.

```
// -----
// main.c (for lab_04a_clock project)
// -----

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

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

//***** Defines *****
#define ONE_SECOND 800000
#define HALF_SECOND 400000

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

    //Initialize Power Management
    initPowerMgmt();

    //Initialize GPIO
    initGPIO();

    //Initialize clocks
    initClocks();

    while(1) {
        // Turn on LED
        GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        // Wait
        _delay_cycles( ONE_SECOND );
        // Turn off LED
        GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        // Wait
        _delay_cycles( ONE_SECOND );
    }
}

//*****
void initGPIO(void) {
    // Set P1.0 to output direction
    GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );
}

void initPowerMgmt(void) {
    // Set core voltage level to handle 8MHz clock rate
    PMM_setVCore( PMM_BASE, _____ );
}
```

Since the setup code is now organized into functions, prototypes need to be included for them

This follows the init code ‘template’ discussed in class

Create GPIO and PowerMgmt functions referenced above

To fill in the blank, refer to Worksheet Question #4

FR5969
The ‘FR5969’ has automatic power management, so it does not need the initPowerMgmt() function.

FR5969

13. ('FR5969 only) Unlock the pins.

Don't forget to add the `PMM_unlockLPM5()` function to `initGPIO()`, if you haven't already done so.



14. Build the code and fix any errors. When no errors exist, launch the debugger.



Debugging the Clocks

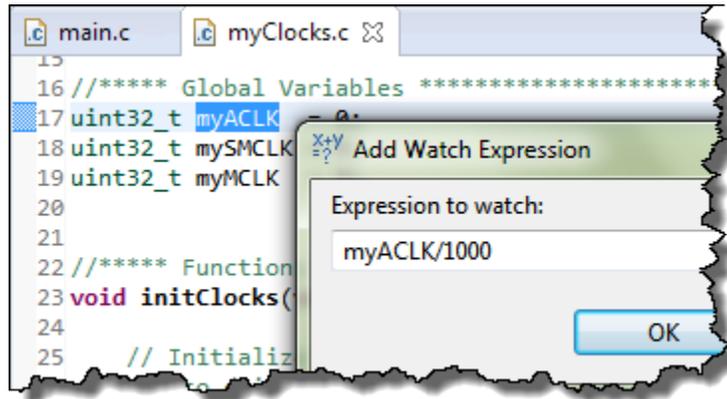
Before running the code, let's set some breakpoints and watch expressions.

15. Open `myClocks.c` in the debugger.

16. Add a watch expression for `myACLK` (in KHz).

Select `myACLK` in your code → Rt-click → Add Watch Expression...

Enter `'myACLK/1000'` into the dialog and hit OK. Upon hitting "OK", the *Expressions* window should open up, if it's not already open.



In a minute, this should give us a value of 32, if `ACLK` is running at 32KHz.

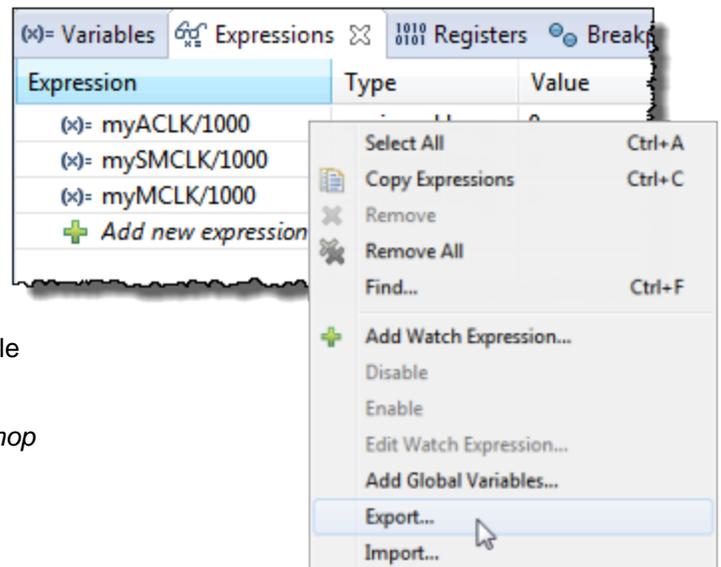
17. Go ahead and create similar watch expressions for `SMCLK` and `MCLK`.

```
mySMCLK/1000
myMCLK/1000
```

18. Export expressions.

CCS lets you export and import expressions. Let's save them so that we can quickly import them later.

- a) Right-click on *Expressions* window
- b) Select *Export...*
- c) And choose a name & location for the file
 - We called it: `myExpressions.txt`
 - and placed it at: `C:\msp430_workshop`



Note: Before you run the code to the first breakpoint, you may see an error in the Expressions window similar to “Error: identifier not found”. This happens when the variable in the expression is out-of-scope. For example, this can happen if you defined the variable as a local, but you were currently executing code in another function. Then again, it will also happen if you delete a variable that you had previously added to the Expression watch window.



19. Run the code to the first breakpoint and write down the Express values:

myACLK/1000: _____

mySMCLK/1000: _____

myMCLK/1000: _____

Are these the values that you expected? _____

(Look back at Worksheet question #9, if you need a reminder.)



20. Run to the next breakpoint – at the end of the initClocks() function.

Check on the values again:

myACLK/1000: _____

mySMCLK/1000: _____

myMCLK/1000: _____

Are these the values we were asked to implement? _____

(Look back at Worksheet questions 10-12.)



21. Let the program run from the breakpoint and watch the blinking LED.

Extra Credit (i.e. Optional Step) – Change the Rate of Blinking



22. Halt the processor and terminate the debugger session.

23. Add a function call to `initClocks()` to force MCLK to use a different oscillator.

- ‘F5529 users, try REFO.
- ‘FR5969 users, try using VLO since you don’t have the REFO oscillator.

We suggest that you copy/paste the function that sets up ACLK... then change the ACLK parameter to MCLK.

The ‘F5529 example is to the right:

As it demonstrates, it sets up MCLK (via the `UCS_initFLLSettle()` function) then changes it again right away ... but that’s OK. No harm done.

```

49 // Configure the FLL's frequency and se
50 UCS_initFLLSettle( UCS_BASE,
51     MCLK_DESIRED_FREQUENCY_IN_KHZ,
52     MCLK_FLLREF_RATIO
53 );
54
55 UCS_clockSignalInit( UCS_BASE,
56     UCS_MCLK,
57     UCS_REFOCLK_SELECT,
58     UCS_CLOCK_DIVIDER_1
59 );
60
61 // Verify that the modified clock setti
62 myACLK = UCS_getACLK( UCS_BASE );
63 mySMCLK = UCS_getSMCLK( UCS_BASE );
    
```

FYI: DriverLib version 1.70 removed the “_BASE” argument from most of the functions.



24. Build your code and launch the debugger.



25. Run the code, stopping at both breakpoints.

Did the value for MCLK change? _____

It should be much slower now that it’s running from REFO or VLO.

26. After the second breakpoint, watch the blinking light.

When the code leaves the `initClocks()` function and starts executing the `while{} loop`, it should take a very loooooong time to run the `_delay_cycles()` functions; our “ONE_SECOND” time was based upon a very fast clock, not one this slow.

If you’re patient enough, you should see the light blink...

(I guess I’m not patient enough to wait for VLO clock... Ed)

(Optional) Lab 4b – Exploring the Watchdog Timer

First, a couple of Questions

1. Complete the code needed to enable the Watchdog Timer using ACLK:

```
WDT_A_watchdogTimerInit(                                     //Initialize the WDT as a watchdog
    WDT_A_BASE,
    _____, //Which clock should WDT use?

    //WDT_A_CLOCKDIVIDER_64 ); //Divide the WDT clock input?
    WDT_A_CLOCKDIVIDER_512 ); //Here are 3 (of 8) different div choices
    //WDT_A_CLOCKDIVIDER_32K );

    _____( WDT_A_BASE ); //Start the watchdog
```

2. Write the code to 'kick the dog'? (Or you can say 'feed the dog' if 'kick sounds too mean)

The purpose of the watchdog is reset the processor if your code doesn't reset it before the count runs out. What driverlib function can you use to reset the timer?

Play with last lab exercise

Before we create a new lab exercise, let's quickly test our old one with regards to the Watchdog.



3. Launch and run the lab_04a_clock project.

If there are any breakpoints set, remove them. Run the program and observe how fast the LED is blinking. (Ours was blinking about 1/sec.)



4. Terminate the Debugger.

5. Edit the source file by commenting out the Watchdog hold function.

```
// WDT_A_hold( WDT_A_BASE );
```



6. Launch the debugger and run the program.

How fast is the LED blinking now? _____

(Ours wasn't blinking at all, after we left the WDT_A running. It must reset the processor before we even get to the while{} loop.)

7. Close the lab_04a_clock project.

File Management

8. Import the solution for lab_02a_ccs.

Project → Import Existing CCS Eclipse Project

Use the archived solution file:

```
C:\msp430_workshop\<target>\solutions\lab_02a_ccs_solution.zip
```

9. Rename the project to: lab_04b_wdt

10. Build the project, just to verify it still works correctly.

11. Import DriverLib into your project and add the appropriate path to the compiler's #include search path setting.

If you need a reminder on how to do this, look back at **Lab3a** under the heading:

“Add MSP430ware Driverlib”

Hint: Don't forget to add the DriverLib directory to the compiler's search path!

12. Build the project, to verify the library was added correctly.

Fix any errors and test until the program builds without any errors.

Edit the Source File

13. First, let's modify the printf() statement.

Next, we want to modify the print statement so that it shows how many times it has been executed.

a) Add a global variable to the program.

```
uint16_t count = 0;
```

b) Replace printf() statement with the following while{} loop:

```
while (1) {  
    count++;  
    printf("I called this %d times\n", count);  
}
```

14. Build the code to make sure it's still error free. Fix any errors it finds.

15. Replace the watchdog hold code with the two WDT_A functions written earlier.

Remember that we didn't actually write this code. It 'holds' the watchdog by using register-based syntax. So, this is the line you want to replace:

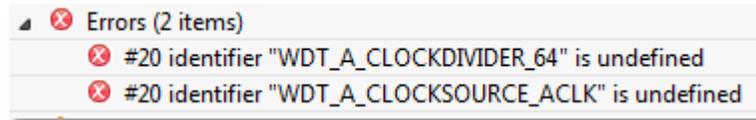
```
WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
```

This new code will initialize the watchdog timer using the clock and divisor of our choice; then start the watchdog timer running. (See question in step #1 on page 4-55.)



16. Build the code to test that is error-free (syntax wise).

Did you get an error? Unless you're really experienced and changed one other item, you should have received an error similar to this:



Where are these values defined? _____

17. Include driverlib.h in your hello.c file.

Yep, when we added the driverlib code, we needed to add the driverlib header file, too. Actually, you can replace the msp430.h file with driverlib.h because the latter references the former.

When complete, your code should look similar to this:

```
#include <stdio.h>
#include <driverlib.h>

uint16_t count = 0;

/*
 * hello.c
 */
int main(void) {
    // WDTCTL = WDTPW | WDT_HOLD; // Stop watchdog timer

    WDT_A_watchdogTimerInit( WDT_A_BASE,
                             WDT_A_CLOCKSOURCE_ACLK,
                             //WDT_A_CLOCKDIVIDER_64 ); //WDT clock input divisor
                             WDT_A_CLOCKDIVIDER_512 ); //Here are 3 (of 8) div choices
                             //WDT_A_CLOCKDIVIDER_32K );

    WDT_A_start( WDT_A_BASE );

    while (1) {
        count++;
        printf("I called this %d times\n", count);
    }
}
```



18. Build the code; fix any errors.



19. Launch the debugger and run the program. Write down the results.

How many times does printf() run before the count restarts? Terminate, change divisor, and retest. (This is why we put 2 commented-out lines in the code.)

Number of times printf() runs before watchdog reset:

WDT_A_CLOCKDIVIDER_64: _____

WDT_A_CLOCKDIVIDER_512: _____

WDT_A_CLOCKDIVIDER_32K: _____

For the watchdog lab using different divisor values, we got the following results:

- 'F5529: 1, 9, 589 (respectively) ... *did you wait all the way to 589 before giving up?*
- 'FR5969: 0, 2, 141

If you're really curious about what is happening under-the-hood, try examining the Watchdog control register. You can see it sets a different value for each of the divisor arguments. For example, on the 'FR5969, the arguments relate to these values:

- ÷ Default: 4 (i.e. ÷32K)
- ÷ 64: 7
- ÷ 512: 6
- ÷ 32K: 4

Keep it Running

20. Add the function call that will keep the CPU running without a watchdog reset.

Add the line of code to the while{} loop – our answer to question # in this lab – that will reset the watchdog and keep the program running.

```
WDT_A_resetTimer( WDT_A_BASE );
```

Hint: You may want to change the clock divisor back to WDT_A_CLOCKDIVER_64 to make it easier to see the change. Then, if the count goes past "1" you'll know the watchdog is being serviced.

21. Build and run the program to observe the watchdog resetting the MSP430.

How many times will it run now? _____

22. When done playing with the program, terminate your debug session close the project.

Extra Credit – Try DriverLib’s Watchdog Example (#3)

The driverlib library contains an example for ‘watching’ the watchdog timer. Give it a test to watch every time the watchdog rolls-over.

23. Import the `wdt_a_ex3_watchdogACLK` project using the CCSv5 Resource Explorer.

If you cannot remember how to import a project using Resource Explorer, please refer back to the beginning of *Lab3b – Reading a Push Button*. We started that lab by importing the EmptyProject example project.

24. Examine the source file in the project.

Notice how they utilize the GPIO pin. Every time the program re-starts it toggles the GPIO pin.

If you look in the User Guide for your MSP430 device, you can see that while the PDIR (pin direction) register is reset after a Power-Up Clear (PUC), the POUT value is left alone. This is the trick used to make the pin toggle after every watchdog reset.

Note, PUC was described during this chapter, while the GPIO pins were discussed in Chapter 3.

25. Build and run the program to observe the watchdog resetting the MSP430.

26. When you’re done, close the project.

(Optional) Lab 4c – Using Crystal Oscillators

File Management

1. Import lab_04a_clock_solution.

If you don't remember how to do this, refer back to lab step 1 (on page 4-47).

2. Rename the project to lab_04c_crystals.

3. Make sure the project builds correctly.

4. Delete two files from the project:

- myClocks.c
- Old readme file (not required, but might make things less confusing later on)

5. Add files to project.

Add the following two files to the project:

- myClocksWithCrystals.c
- lab_04_crystals_readme.txt (again, not required, but helpful)

You'll find them along the path

```
C:\msp430_workshop\<target>\lab_04c_crystals\
```

6. Examine the new C file.

Notice the following:

- We need to “start” the crystal oscillators before selecting them as a clock source.
- Two different ways to “start” a crystal – with and without a timeout.
 - If no timeout is used, then that function will continue until the oscillator is started. That could effectively halt the program indefinitely, if there is a problem with the crystal (say, it breaks, has a solder fault, or has fallen off the board).
 - A better solution might be to specify a timeout ... as long as you check for the result after the function completes. (In our example, we just used an indefinite wait loop, but “in real life” you might choose another clock source based on a failed crystal.)

7. Build to verify the file import was OK.

Modify GPIO

8. Add the following code to the `initGpio()` function in `main.c`.

Rather than having you build and run the project only to find out it doesn't work (like what happened to the course author), we'll give you a hint: connect the clock pins to the crystals.

As you can see, the two different devices are pinned-out differently. Pick the code to match your processor.

F5529

```
// Connect pins to crystal in/out pins
GPIO_setAsPeripheralModuleFunctionInputPin(
    GPIO_PORT_P5,
    GPIO_PIN5 +           // XOUT on P5.5
    GPIO_PIN4 +           // XIN on P5.4
    GPIO_PIN3 +           // XT2OUT on P5.3
    GPIO_PIN2             // XT2IN on P5.2
);
```

and

FR5969

```
// Connect pins to crystal in/out pins
// Note, PJ.6 and PJ.7 not needed as HF crystal is not present
GPIO_setAsPeripheralModuleFunctionInputPin(
    GPIO_PORT_PJ,
    GPIO_PIN4 +           // LFXIN on PJ.4
    GPIO_PIN5,             // LFXOUT on PJ.5
    // GPIO_PIN6 +         // HFXTIN on PJ.6
    // GPIO_PIN7           // HFXTOUT on PJ.7
    GPIO_PRIMARY_MODULE_FUNCTION
);
```

By default – on some MSP430 devices, such as the F5529 and FR5969 – these pins default to GPIO mode. Thus, we have to connect them by reprogramming the GPIO.

One difference between the two processors – besides the port number being used – is that we had to specify “GPIO_PRIMARY_MODULE_FUNCTION” for the ‘FR5969. This device allows multiple Peripheral I/O pin options. (Refer back to Chapter 3 for more details on this topic.)

Note: In our solution, we connected all four pins to their clock functions using the `GPIO_setAsPeripheralModuleFunctionInputPin()`.

In other examples, we saw this done two different ways. One example was similar to ours, the other set the IN pins with the ‘InputPin’ function, while the setting the OUT pins using the `GPIO_setAsPeripheralModuleFunctionOutputPins()` function.

It appears that either of these solutions works. *We chose the solution with less typing.*

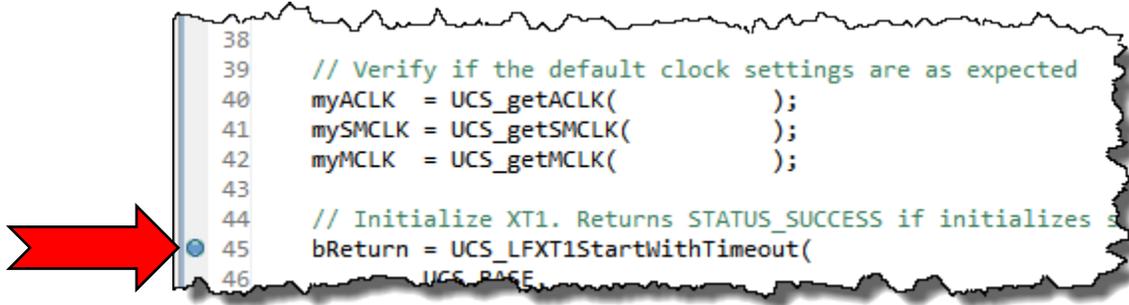
9. Build and launch the debugger.

Debug

10. Set three breakpoints in the `myClocksWithCrystals.c` file.

Set a breakpoint after each instance of the code where we read the clock settings.

For example:



```
38
39 // Verify if the default clock settings are as expected
40 myACLK = UCS_getACLK( );
41 mySMCLK = UCS_getSMCLK( );
42 myMCLK = UCS_getMCLK( );
43
44 // Initialize XT1. Returns STATUS_SUCCESS if initializes s
45 bReturn = UCS_LFXT1startWithTimeout(
46 UCS_BASE,
```

11. Run the code (click 'Resume') three times and record the clock settings:

Because of the way the FLL clock is handled on the 'F5529, we have three places to record the clock values. With the 'FR5969, you only need the first two columns.

Expression	Default Settings	First Clock Get	Second Clock Get
myACLK/1000			
mySMCLK/1000			
myMCLK/1000			

On the 'F5529, why didn't SMCLK get set correctly on the first setup?
We setup SMCLK to use XT2CLK, but it didn't seem to take:

Hint: Read the comments on the code itself. We hope that'll explain what caused this.

12. When done experimenting with this code, terminate the debugger and close the project.

Chapter 04 Appendix

Hints:

Chapter 4 Worksheet (1)

- ◆ The MSP430 DriverLib Users Guide will be useful in helping to answer these workshop questions. Find it in your MSP430ware DriverLib doc folder:
e.g. `\MSP430ware_1_70_00_28\driverlib\doc\`
- ◆ Maybe even more helpful is to reference the actual DriverLib source code – that is, the .h/.c files for each module you are using. For example:
`\MSP430ware_1_70_00_28\driverlib\driverlib\MSP430F5xx_6xx\ucs.h`
- ◆ Finally, we recommend you also reference the DriverLib UCS example #4:
`\msp430\MSP430ware_1_70_00_28\driverlib\examples\MSP430F5xx_6xx\ucs\ucs_ex4_XTSourcesDCOInternal.c`

Reset and Operating Modes & Watchdog Timers

1. Name all 3 types of resets:
BOR, POR, PUC
2. If the Watchdog (WDT) times out, which reset does it invoke?
PUC
3. Write the DriverLib function that stops (halts) the watchdog timer:
WDT_A_hold (WDT_A_BASE);

Chapter 4 Worksheet (2)

Power Management

4. ('F5529 Launchpad users only)
Write the DriverLib function that sets the core voltage needed to run MCLK at 8MHz.
initPowerMgmt (PMM_CORE_LEVEL_1);

Clocking

5. Why does MSP430 provide 3 different types of internal clocks?
To meet the varying demands of performance, accuracy, and power.
One clock runs the CPU, while the other two provide fast and
slow/low-power clocking to the peripherals

Name them:

MCLK SMCLK ACLK

Chapter 4 Worksheet (3)

6. What is the speed of the crystal oscillators on your board?
(Hint: look in the Hardware section of the Launchpad Users Guide.)

```
#define LF_CRYSTAL_FREQUENCY_IN_HZ 32768
#define HF_CRYSTAL_FREQUENCY_IN_HZ 400000
```

(for FR5969: We chose "0" for High Frequency crystal, since the board doesn't ship with one)

7. What function specifies these crystal frequencies to the DriverLib?
Hint: Look in the MSP430ware DriverLib User's Guide – "UCS chapter".

```
UCS_setExternalClockSource (
    (for FR5969: CS_setExternalClock Source) LF_CRYSTAL_FREQUENCY_IN_HZ ,
    HF_CRYSTAL_FREQUENCY_IN_HZ );
```

Chapter 4 Worksheet (4)

8. What speed are the clocks running at? There's an API for that...
Write the code that returns your current clock frequencies:

```
uint32_t myACLK = 0;
uint32_t mySMCLK = 0;
uint32_t myMCLK = 0;

myACLK = UCS_getACLK ();
mySMCLK = UCS_getSMCLK ();
myMCLK = UCS_getMCLK ();
```

F5529 Prefix = 'UCS'
FR5969 Prefix = 'CS'

9. We didn't setup the clocks (or power level) in our previous labs,
how come our code worked?

There are default values provided in hardware for clocks, power, etc.

Don't spend too much time pondering this, but what speed do you
think each clock is running at before we configure them?

```
'F5529 ACLK: 32 KHz SMCLK: 1.048 MHz MCLK: 1.048 MHz
'FR5969 ACLK: 39 KHz SMCLK: 1 MHz MCLK: 1 MHz
```

Chapter 4 Worksheet (5)

10. Setup ACLK:

- Use REFO for the F5529 device
- Use VLO for the FR5969 device

F5529 Prefix = 'UCS'
FR5969 Prefix = 'CS'

F5529

```
// Setup ACLK
UCS_clockSignalInit (
    UCS _ACLK,           // Clock to setup
    UCS_REFOCLK_SELECT, // Source clock
    UCS _CLOCK_DIVIDER_1
);
```

FR5969

```
// Setup ACLK
CS_clockSignalInit (
    CS _ACLK,           // Clock to setup
    CS_VLOCLK_SELECT,  // Source clock
    CS _CLOCK_DIVIDER_1
);
```

Chapter 4 Worksheet (6)

11. (F5529 User's only) Write the code to setup MCLK. It should be running at 8MHz using the DCO as its oscillator source.

```
#define MCLK_DESIRED_FREQUENCY_IN_KHZ    8000

#define MCLK_FLLREF_RATIO MCLK_DESIRED_FREQUENCY_IN_KHZ/(UCS_REFOCLK_FREQUENCY/1024)

// Set the FLL's clock reference clock to REFO
UCS_clockSignalInit (
    UCS_FLLREF,           // Clock you're configuring
    UCS_REFOCLK_SELECT,  // Clock Source
    UCS_CLOCK_DIVIDER_1 );

// Config the FLL's freq, let it settle, and set MCLK & SMCLK to use DCO+FLL as clk source
UCS_initFLLSettle (
    MCLK_DESIRED_FREQUENCY_IN_KHZ,
    MCLK_FLLREF_RATIO );
```

Chapter 4 Worksheet (7)

12. (FR5969 Users only) Write the code to setup MCLK. It should be running at 8MHz using the DCO as its oscillator source.

```
// Set DCO to 8MHz
CS_setDCOFreq(
    CS_DCORSEL_1, // Set Frequency range (DCOR)
    CS_DCOFSEL_3 // Set Frequency (DCOF)
);

// Set MCLK to use DCO clock source
CS_clockSignalInit(
    CS_MCLK,
    CS_DCOCLK_SELECT,
    UCS_CLOCK_DIVIDER_1 );
```

Chapter 4b Worksheet

1. Complete the code needed to enable the Watchdog Timer using ACLK. (Hint: look at the WDT_A section of the DriverLib User's Guide)

```
// Initialize the WDT as a watchdog
WDT_A_watchdogTimerInit(
    WDT_A_BASE,
    WDT_A_CLOCKSOURCE_ACLK; //Which clock should WDT use?
    WDT_A_CLOCKDIVIDER_64 ); //Divide the WDT clock input?
//WDT_A_CLOCKDIVIDER_512 ); //Two other divisor options
//WDT_A_CLOCKDIVIDER_32K );

// Start the watchdog
WDT_A_start ( WDT_A_BASE );
```

2. Write the code to 'kick the dog'?

```
WDT_A_resetTimer ( WDT_A_BASE );
```

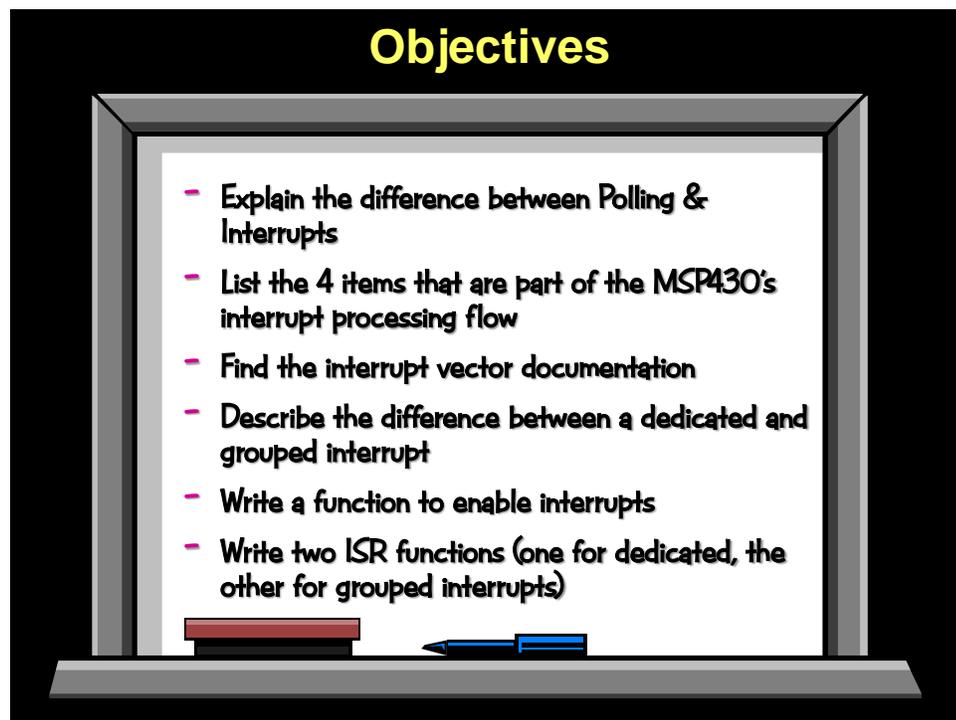
Introduction

What is an embedded system without interrupts?

If you just needed to solve a math problem you would most likely sit down and use a desktop computer. Embedded systems, on the other hand, take inputs from real-world events and then act upon them. These real-world events usually translate into 'interrupts' – asynchronous signals provided to the microcontroller: timers, serial ports, pushbuttons ... and so on.

This chapter discusses how interrupts work; how they are implemented on the MSP430 MCU, and what code we need to write in order to harness their functionality. The lab exercises provided are relatively simple (using a pushbutton to generate an interrupt), but the skills we learn here will apply to all the remaining chapters of this workshop.

Learning Objectives



Chapter Topics

Interrupts	5-1
<i>Interrupts, The Big Picture</i>	<i>5-3</i>
Polling vs Interrupts.....	5-3
Processor States and Interrupts	5-5
Threads: Foreground and Background.....	5-6
<i>How Interrupts Work</i>	<i>5-7</i>
1. Interrupt Must Occur	5-9
2. Interrupt is Flagged (and must be Enabled)	5-10
3. CPU's Hardware Response	5-12
4. Your Software ISR	5-14
<i>Interrupts: Priorities & Vectors</i>	<i>5-17</i>
Interrupts and Priorities.....	5-17
Interrupt Vector (IV) Registers	5-18
Interrupt Vector Table	5-19
<i>Coding Interrupts.....</i>	<i>5-22</i>
Dedicated ISR (Interrupt Service Routine).....	5-22
Grouped ISR (Interrupt Service Routine).....	5-24
Enabling Interrupts	5-26
<i>Miscellaneous Topics.....</i>	<i>5-28</i>
Handling Unused Interrupts	5-28
Interrupt Service Routines – Coding Suggestions	5-29
GPIO Interrupt Summary	5-30
Interrupt Processing Flow	5-30
<i>Interrupts and TI-RTOS Scheduling.....</i>	<i>5-31</i>
Threads – Foreground and Background.....	5-31
TI-RTOS Thread Types.....	5-33
Summary: TI-RTOS Kernel.....	5-36
<i>Lab Exercise</i>	<i>5-37</i>

Interrupts, The Big Picture

While many of you are already familiar with interrupts, they are so fundamental to embedded systems that we wanted to briefly describe what they are all about.

From Wikipedia:

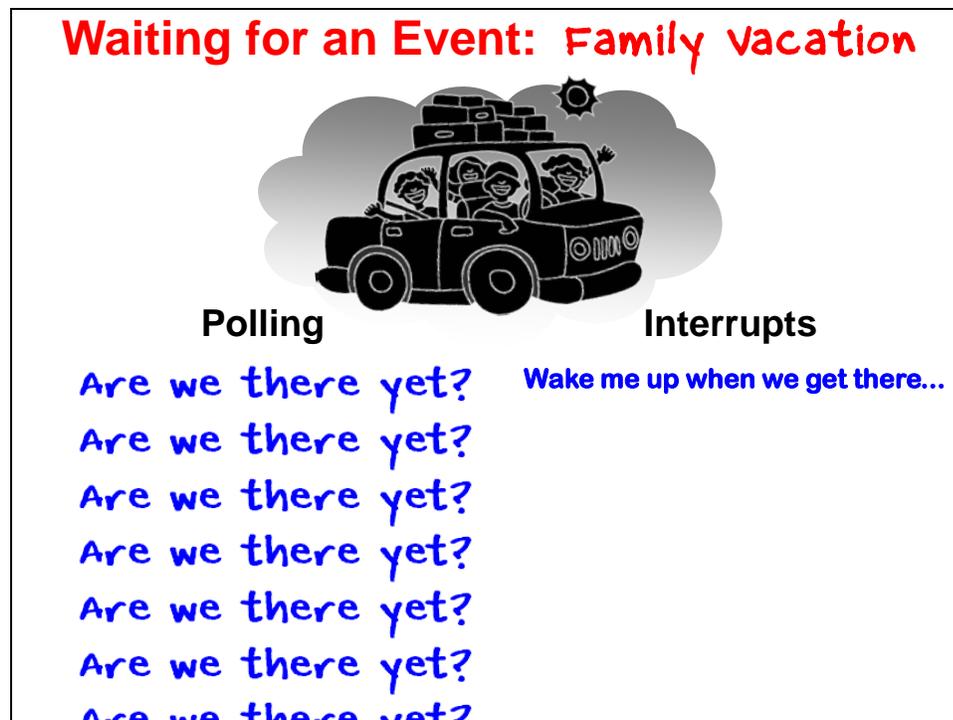
*A **hardware interrupt** is an electronic alerting signal sent to the processor from an external device, either a part of the [device, such as an internal peripheral] or an external peripheral.*

In other words, the interrupt is a signal which notifies the CPU that an event has occurred. If the interrupt is configured, the CPU will respond to it immediately – as described later in this chapter.

Polling vs Interrupts

In reality, though, there are two methods that events can be recognized by the processor. One is called “Polling”; the other is what we just defined, “Interrupts”.

We start with a non-engineering analogy for these two methods. If you’ve ever taken a long family vacation, you’ve probably dealt with the “Are we there yet” question. In fact, kids often ask it over-and-over again. Eventually ... the answer will be, “Yes, we’re there”. The alternative method is when my spouse says, “Wake me up when we get there”.



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

A real-world event might be our system responding to a push-button. Once again, the event could be handled using either **Polling** or **Interrupts**.

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

Waiting for an Event: Button Push



Polling

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

100% CPU Load

Interrupts

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

> 0.1% CPU Load

The example on the right shows an **Interrupt** based solution. Since this code is not constantly running, as in the previous example’s while{} loop, the CPU load is very low.

Why do simple examples often ignore the use of interrupts? Because they are “simple”. Interrupts, on the other hand, require an extra three items to get them running. We show two of them in the right-hand example above.

- The #pragma sets up the interrupt vector. The MSP430 has a handy pragma which makes it easy to configure this item. (Note: we’ll cover the details of all these items later in this chapter.)
- The __interrupt keyword tells the compiler to code this function as an interrupt service routine (ISR). Interrupt functions require a context save and restore of any resources used within them.

While not shown above, we thought we’d mention the third item needed to get interrupts to work. For a CPU to respond to an interrupt, you also need to enable the interrupt. (Oh, and you may also have to setup the interrupt source; for example, we would have to configure our GPIO pin to be used as an interrupt input.)

So, in this chapter we leave the simple and inefficient examples behind and move to the real-world – where real-world embedded systems thrive on interrupts.

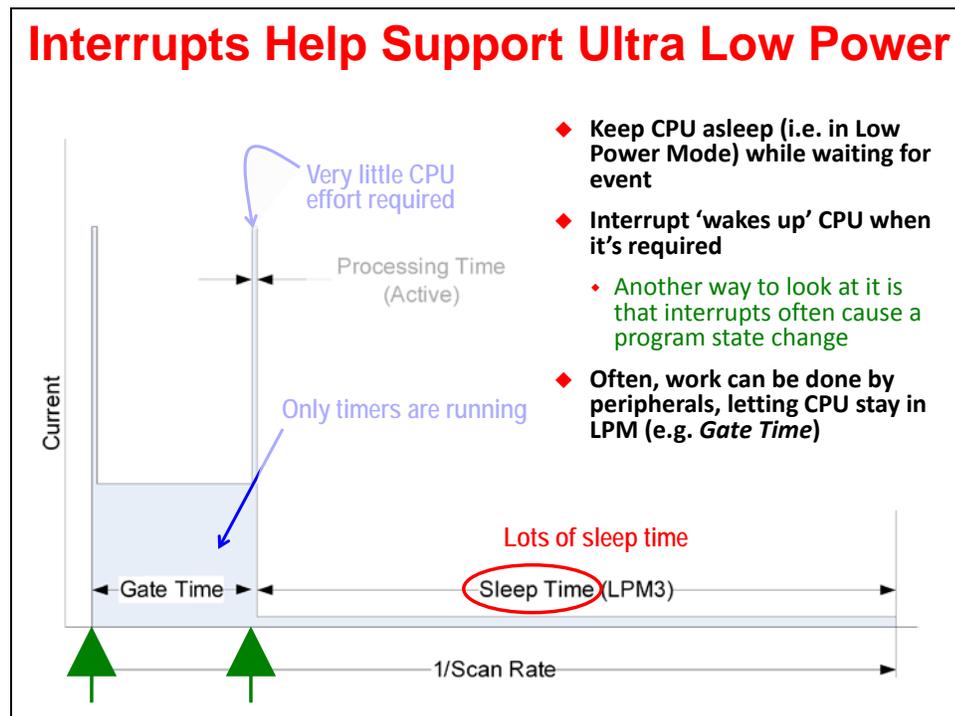
Processor States and Interrupts

In a previous chapter we covered many of the MSP430's processor states. To summarize, the MSP430 CPU can reside in: Reset, Active, or one of many Low-Power Modes (LPM). In many cases, interrupts cause the CPU to change states. For example, when sitting in Low Power Mode, an interrupt can "wake-up" the processor and return it to its active mode.

To help demonstrate this point, we stole the following slide from a discussion about Capacitive Touch. While most of this slide's content is not important for our current topic, we thought the *current vs time graph* was interesting. It tries to visually demonstrate the changing states of the device by charting power usage over time.

Notice the four states shown in this diagram:

- Notice how the current usage goes up at the beginning event – this is when the CPU is woken up so it can start a couple of peripherals (timers) needed to read the CapTouch button.
- The CPU can then go back to sleep while the sensor is being 'read' by the timers.
- When the read is complete (defined by something called "Gate" time, the CPU gets interrupted and wakes up again in order to calculate the CapTouch button's value from the sensor data.
- Finally the CPU (and CapTouch hardware) can go back to sleep and wait for another system wake-up event.



Threads: Foreground and Background

We conclude our Interrupts introduction by defining a few common terms used in interrupt-driven systems: **Thread**, **Foreground** and **Background**.

If you look at the “code” below, you will see that there are three individual – and independent – code segments below: main, ISR1, and ISR2.

We use the word *independent* because, if you were to examine the code in such a system, there are no calls between these three routines. Each one begins and ends execution without calling the others. It is common to call these separate segments of code: “Threads”.

Foreground / Background Scheduling

<pre style="margin: 0;">main() { //Init initPMM(); initClocks(); ... while(1){ background or LPMx } }</pre>	<p>System Initialization</p> <ul style="list-style-type: none"> ◆ The beginning part of main() is usually dedicated to setting up your system (Chapters 3 and 4)
<pre style="margin: 0;">ISR1 get data process</pre>	<p>Background</p> <ul style="list-style-type: none"> ◆ Most systems have an endless loop that runs ‘forever’ in the background ◆ In this case, ‘Background’ implies that it runs at a lower priority than ‘Foreground’ ◆ In MSP430 systems, the background loop often contains a Low Power Mode (LPM) command – this sleeps the CPU/System until an interrupt event wakes it up
<pre style="margin: 0;">ISR2 set a flag</pre>	<p>Foreground</p> <ul style="list-style-type: none"> ◆ Interrupt Service Routine (ISR) runs in response to enabled hardware interrupt ◆ These events may change modes in Background – such as waking the CPU out of low-power mode ◆ ISR’s, by default, are not interruptible ◆ Some processing may be done in ISR, but it’s usually best to keep them short

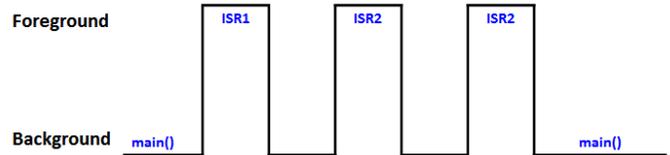
As we’ve seen in the workshop already, it is our main() thread that begins running once the processor has been started. The compiler’s initialization routine calls main() when its work is done. (In fact, this is why all C programs start with a main() function. Every compiler works the same way, in this regard.)

With the main() thread started, since it is coded with a while(1) loop, it will keep running forever. That is, unless a hardware interrupt occurs.

When an enabled interrupt is received by the CPU, it preempts the main() thread and runs the associated ISR routine – for example, ISR1. In other words, the CPU stops running main() temporarily and runs ISR1; when ISR1 completes execution, the CPU goes back to running main().

Here's where the terms **Foreground** and **Background** come into play. We call `main()` the **Background** thread since it is our "default" thread; that is, the program is designed such that we start running `main()` and go back to it whenever we're done with our other threads, such as `ISR1`.

Whenever an interrupt causes another thread to run, we call that a **Foreground** thread. The foreground threads preempt the Background thread, returning to the Background once completed.



The words "**Foreground**" and "**Background**" aren't terribly important. They just try to provide a bit of context that can be visualized in this common way.

It should be noted that it's important to keep your interrupt service routines short and quick. This, again, is common practice for embedded systems.

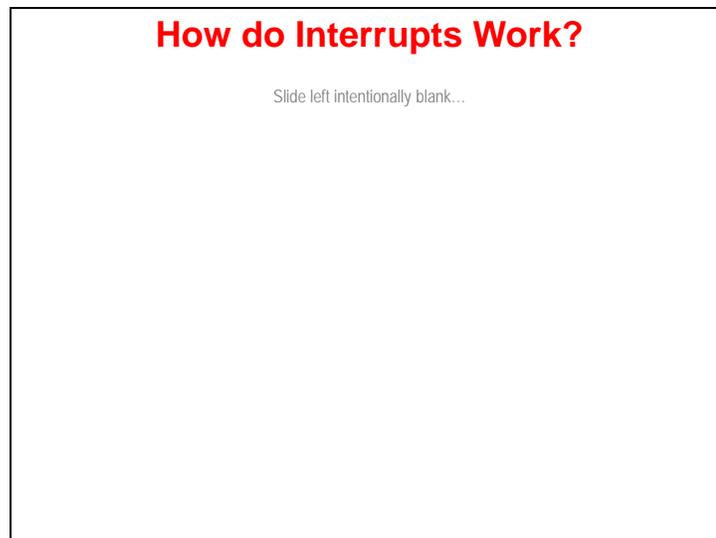
Note: We realize that our earlier definition of "Thread" was a little weak. What we said was true, but not complete. The author's favorite definition for "Thread" is as follows:

"A function or set of functions that operate independently of other code – running within their own context."

The key addition here is that a thread runs within its own context. When switching from one thread to another, the context (register values and other resources) must be saved and restored.

How Interrupts Work

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

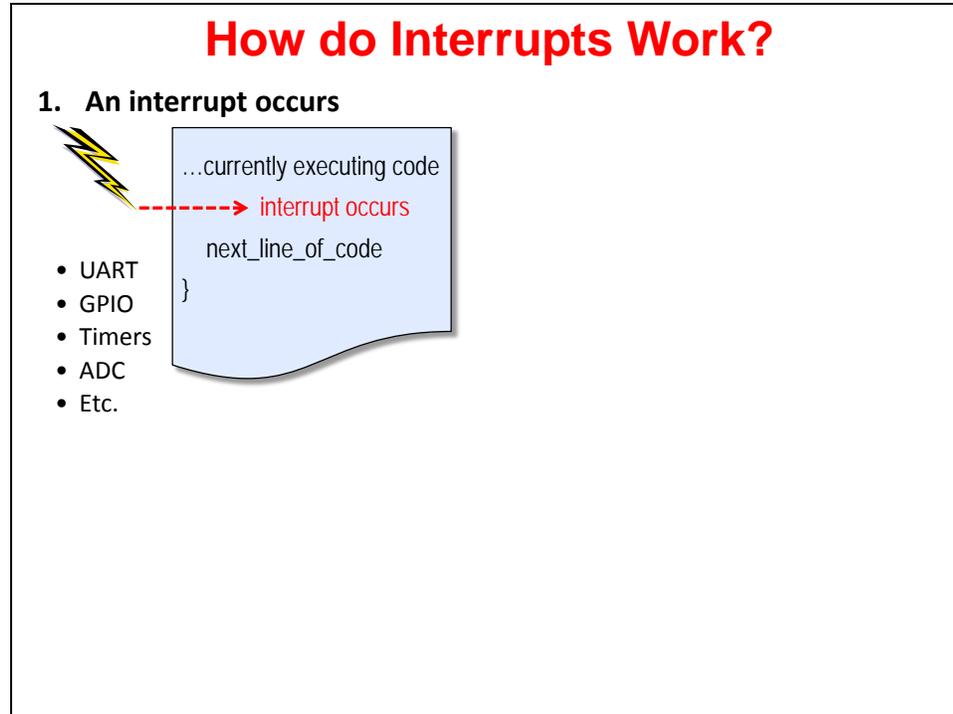


If you've been reading this chapter, you might notice that we've already covered these four items. Over the next few pages we enumerate these steps again, filling-in additional details.

Notes

1. Interrupt Must Occur

For the processor to respond to an interrupt, it must have occurred. There are many possible sources of interrupts. Later in this chapter we will delve into the MSP430 datasheet which lists all of the interrupt sources.



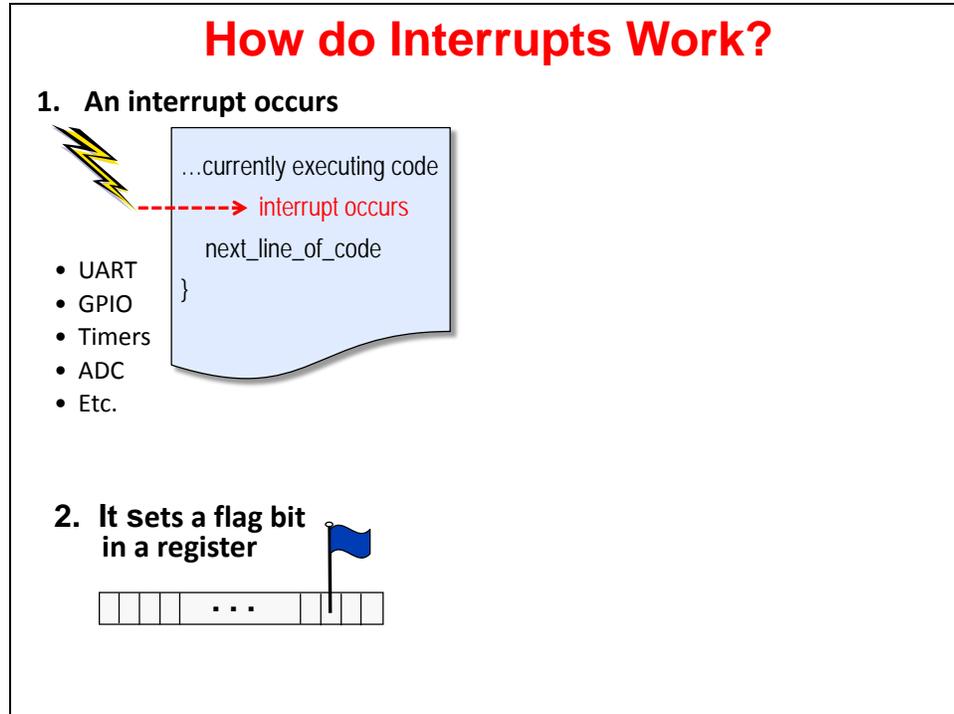
Suffice it to say that most peripherals can generate interrupts to provide status and information to the CPU. Most often, the interrupt indicates that data is available (e.g. serial port) and/or an event has occurred that needs processing (e.g. timer). In some cases, though, an interrupt may be used to indicate an error or exception in a peripheral that the CPU needs to handle.

Interrupts can also be generated from GPIO pins. This is how an external peripheral, or some other controller, can signal the MSP430 CPU. Most MSP430 devices allow the pins from the first two I/O ports (P1 and P2) to be individually configured for interrupt inputs. On the larger devices, there may be additional ports that can be configured for this, as well.

Finally, your software can often generate interrupts. The logic for some interrupts on the processor allow you to manually set a flag bit, thus 'emulating' a hardware interrupt. Not all interrupts provide this feature, but when available, it can be a handy way to test your interrupt service routine.

2. Interrupt is Flagged (and must be Enabled)

When an interrupt signal is received, an interrupt flag (IFG) bit is latched. You can think of this as the processor's "copy" of the signal. As some interrupt sources are only on for a short duration, it is important that the CPU registers the interrupt signal internally.



MSP430 devices are designed with "distributed" interrupt management. That is, most IFG bits are found inside each peripheral's control registers; this is different from most processors which have a common, dedicated set of interrupt registers.

The distributed nature of the interrupts provides a number of benefits in terms of device flexibility and future feature expansion; further, it fits nicely with the low-power nature of the MSP430.

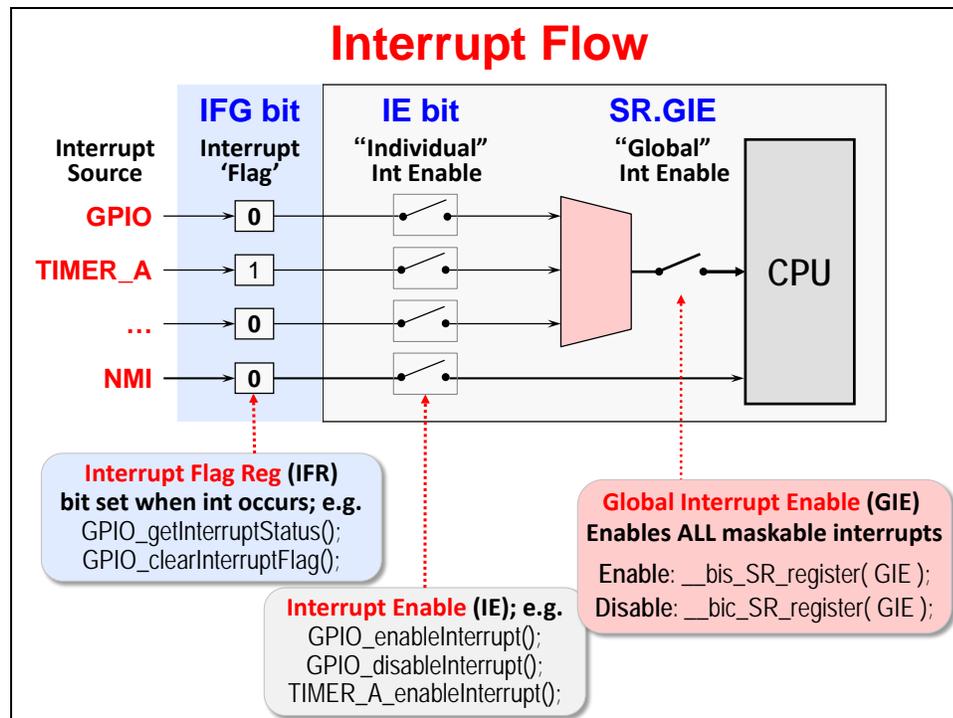
The only 'negative' of distributed interrupts might be that it's different — it's just that many of us older engineers are used to seeing all the interrupts grouped together. Bottom line, though, is that working with interrupts (enabling interrupts, clearing flags, responding to them) is the same whether the hardware is laid out centrally or in a distributed fashion.

Interrupt Flow

How does the interrupt signal reach the CPU?

We've just talked about the interrupt flag (IFG) bit – let's start there. As described on the previous page, when the interrupt source signal is received, the associated IFG bit is set. In fact, DriverLib contains functions to read the status of most IFG bits. (Handy in those few cases where you need to poll an interrupt source.)

When the IFG is set, the MSP430 *device* now sees that the signal has occurred, but the signal hasn't made its way to the *CPU*, yet. For that to happen, the interrupt must be **enabled**.



Interrupt enable bits (IE) exist to protect the CPU ... and thus, your program. Even with so many peripherals and interrupt sources, it's likely that your program will only care about a few of them. The enable bits provide your program with 'switches' that let you ignore all those sources you don't need.

By default, all interrupt bits are disabled (except the Watchdog Timer). It is your program's responsibility to enable those interrupt sources that are needed. To that end, once again, DriverLib provides a set of functions that make it easy for you to set the necessary IE bits.

Finally, there's a "master" switch that turns all interrupts off. This lets you turn off interrupts without having to modify all of the individual IE bits. The MSP430 calls this the global interrupt enable (GIE). It is found in the MSP430 Status Register (SR).

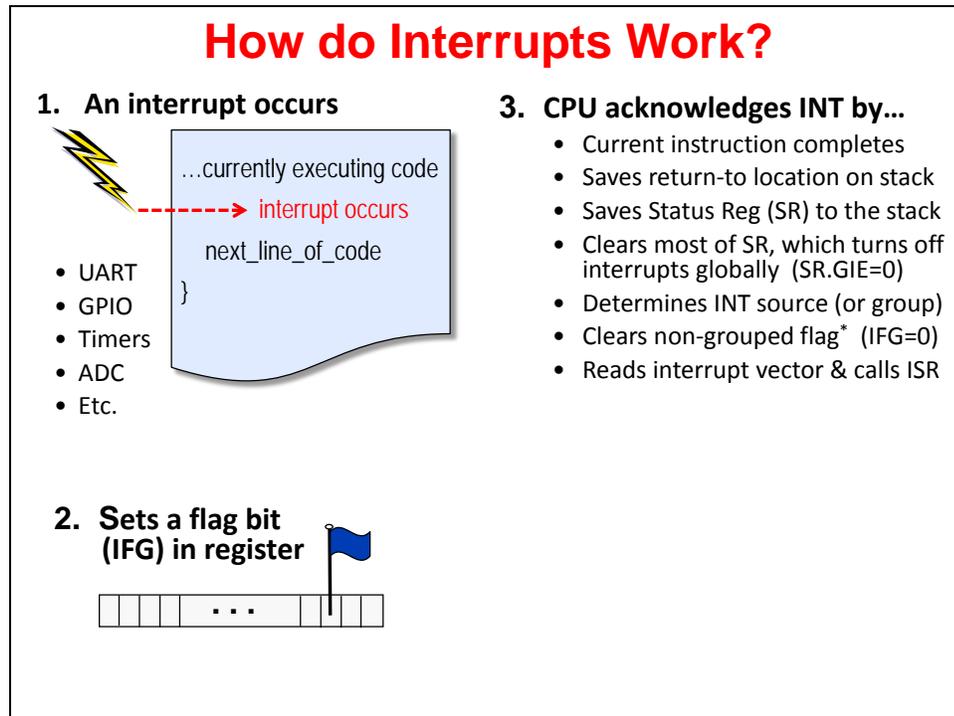
Why would you need a GIE bit? Sometimes your program may need to complete some code *atomically*; that is, your program may need to complete a section of code without the fear that an interrupt could preempt it. For example, if your program shares a global variable between two threads – say between main() and an ISR – it may be important to prevent interrupts while the main code reads and modifies that variable.

Note: There are a few non-maskable interrupts (NMI). These sources bypass the GIE bit. These interrupts are often considered critical events – i.e. 'fatal' events – that could be used to provide a *warm reset* of the CPU.

3. CPU's Hardware Response

At this point, let's assume you have an interrupt that has: occurred; been flagged; and since it was enabled, its signal has reached the CPU. What would the CPU do in response to the interrupt?

Earlier in the chapter we stated: "The interrupt preempts the current thread and starts running the interrupt service routine (ISR)." While this is true, there are actually a number of items performed by the hardware to make this happen – as shown below:



We hope the first 3 items are self-explanatory; the current instruction is completed while the Program Counter (PC) and Status Register (SR) are written to the system stack. (You might remember, the stack was setup for the MSP430 by the compiler's initialization routine. Please refer to the compiler user's guide for more information.)

After saving the context of SR, the interrupt hardware in the CPU clears most of the SR bits. Most significantly, it clears GIE. That means that by default, whenever you enter an ISR function, all maskable interrupts have been turned off. (We'll address the topic of 'nesting' interrupts in the next section.)

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

Interrupt Vector Table – How is it different than other MCU's?

The MSP430 Vector Table is similar and dissimilar to other microcontrollers:

- The MSP430, like most microcontrollers, uses an Interrupt Vector Table. This is an area of memory that specifies a vector (i.e. ISR address) for each interrupt source.
- Some processors provide a unique ISR (and thus, vector) for every interrupt source. Other processors provide only 1 interrupt vector and make the user program figure which interrupt occurred. To maximize flexibility and minimize cost and power, the MSP430 falls in between these two extremes. There are some interrupts which have their own, dedicated interrupt vector – while other interrupts are logically grouped together.
- Where the MSP430 differs from many other processors is that it includes an Interrupt Vector (IV) register for each grouped interrupt; reading this register returns the highest-priority, enabled interrupt for that group of interrupt sources. As we'll see later in this chapter, all you need to do is read this register to quickly determine which specific interrupt to handle.

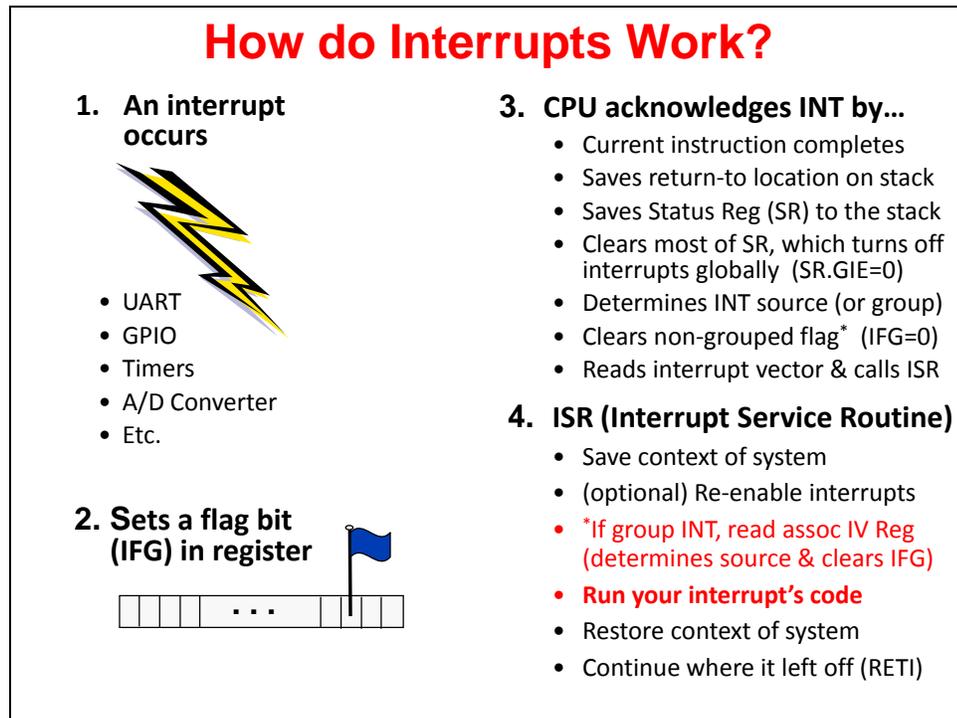
Note: We'll describe Interrupt Vector Table in more detail later in the chapter.

4. Your Software ISR

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

Before we examine the details of the ISR; once again, how did we get to this point?

Looking at the diagram below, we can see that (1) the interrupt must have occurred; (2) the processor flags the incoming interrupt; (3) if enabled, the interrupt flag signal is routed to the CPU where it saves the Status Register and Return-to address and then branches to the ISR's address found in the appropriate location in the vector table. (4) Finally, your ISR is executed.



The crux of the ISR is doing what needs to be done in response to the interrupt; the 4th bullet (listed in red) reads:

- Run your interrupt's code

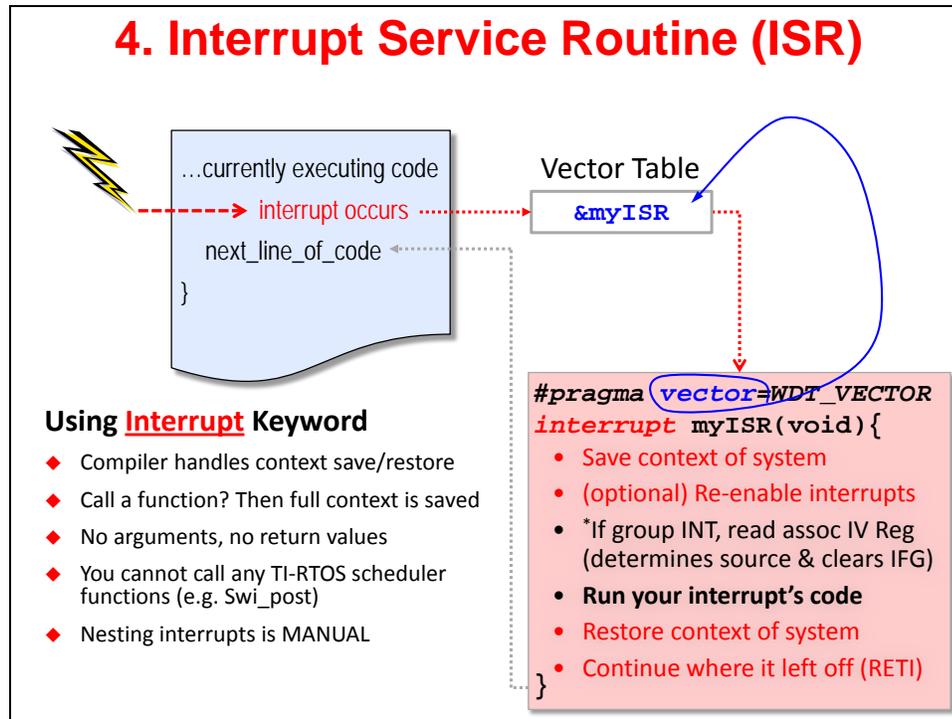
This is meant to describe the code you write to handle the interrupt. For example, if it's a UART interrupt, your code might read an incoming byte of data and write it to memory.

We'll discuss the 2nd (optional) bullet on the next page.

The 3rd bullet indicates that if this is a "grouped" interrupt, you have to add code to figure out which interrupt, in the group, needs to be handled. This is usually done by reading the group's IV register. (This bullet was in red because it is code you need to write.)

The other bullets listed under "4. ISR" are related to saving and restoring the context of the system. This is required so that the condition mentioned earlier can be met: *"without adversely affecting the code threads already running in the system."*

We show the interrupt flow in a slightly different fashion in the following diagram. As you can see, when an enabled interrupt occurs, the processor will look up the ISR's branch-to address from a specific address in memory (called the *interrupt vector*). For the MSP430, this address is defined using the *vector* pragma.



The context of the system – for example, the CPU registers used by the ISR – must be saved before running your code and restored afterwards. Thankfully, the compiler handles this for you when the function is declared as an *interrupt*. (As part of the “context restore”, the compiler will return to running the previous thread of code by using the RETI instruction).

Please note the bullets under “Using the Interrupt Keyword” from the preceding diagram.

Using this keyword, the compiler handles all of the context save/restore for you and knows how to return to your previous code – even restoring the original value for the Status Register (SR).

Hint: If you call a function within your ISR, the compiler will have to save/restore every CPU register, not just the ones that it uses to implement your C code. This is because it doesn't know what resources the function call may end up using.

Since the interrupt occurs asynchronously to the background thread, you cannot pass arguments to and receive return values from the ISR. You must communicate between threads using global variables (or other appropriate data objects).

TI's real-time operating system (TI-RTOS) provides a rich set of scheduling functions that are often used within interrupt service routines. Be aware, though, that some of these functions can only be used with RTOS “managed” interrupts. In fact, it's actually easier to let TI-RTOS manage your interrupts; it automatically handles plugging the interrupt vector as well as context save/restore. (All you have to do is write a standard C function.) But, the details of TI-RTOS are outside the scope of this workshop. While we provide a brief discussion of TI-RTOS at the end of this chapter, please refer to the [Introduction to TI-RTOS Kernel](#) workshop for more details.

Nesting Interrupts (not recommended)

Finally, while the MSP430 allows nesting of interrupts, it is not recommended.

- *Nesting* interrupts means one interrupt can interrupt another interrupt.
- You must manually configure nesting. That is, before running your interrupt handling code you must:
 - Disable any interrupts that you do not want to occur during your ISR. In other words, you must first save, then disable, any IE bit that correlates to an interrupt that you do not want to interrupt your ISR.
 - Then, turn on interrupts globally by setting $GIE = 1$.
 - At this point you can run your code that responds to the original interrupt. It may end up being interrupted by any source that you left enabled.
 - When you've completed your original interrupt code, you need to disable interrupts before returning from the function. That is, set $GIE = 0$. (This is the state GIE was in when entering your ISR code.
 - You can now restore the IE bits that you saved before enabling GIE .
 - At this point, you can return from the ISR and let the compiler's code handle the remaining context save and return branch back to the original thread.
- In general, it's considered better programming practice to keep interrupt service routines very short – i.e. lean-and-mean. Taking this further, with low-power and efficiency in mind, the MSP430 team recommends you follow the no-nesting general principle.

Hint: We encourage you to avoid nesting, if at all possible. Not only is it difficult, and error prone, it often complicates your programs ability to reach low-power modes.

Interrupts: Priorities & Vectors

Interrupts and Priorities

Each MSP430 device datasheet defines the *pending* priority for each of its hardware interrupts. In the case of the MSP430F5529, there are 23 interrupts shown listed below in decreasing priority.

In the previous paragraph we used the phrase “pending priority” deliberately. As you might remember from the last topic in this chapter, interrupts on the MSP430 do not nest within each other by default. This is because the global interrupt (GIE) bit is disabled when the CPU acknowledges and processes an interrupt. Therefore, if an interrupt occurs while an ISR is being executed, it will have to wait for the current ISR to finish before it can be handled ... even if the new interrupt is of higher priority.

On the other hand, if two interrupts occur at the same time – that is, if there are two interrupts currently pending – then the highest priority interrupt is acknowledged and handled first.

Interrupt Priorities (F5529)

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

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

Most of the 23 interrupts on the 'F5529 represent 'groups' of interrupts. There are actually 145 interrupt sources – each with their own interrupt flag (IFG) – that map into these 23 interrupts.

For example, the “Timer B (CCIFG0)” interrupt represents a single interrupt signal. When the CPU acknowledges it, it will clear its single IFG flag.

On the other hand, the next interrupt in line, the “Timer B” interrupt, represents all the rest of the interrupts that can be initiated by Timer0_B. When any one of the interrupts in this group occurs, the ISR will need to determine which specific interrupt source occurred and clear its flag (along with executing whatever code you want to associate with it).

Interrupt Vector (IV) Registers

As has been mentioned a couple of times in this chapter, to make responding to *grouped* interrupts easier to handle, the MSP430 team created the concept of *Interrupt Vector (IV) Registers*. Reading an IV register will return the *highest-priority, pending interrupt* in that group; it will also clear that interrupt's associated flag (IFG) bit.

Interrupt Vector (IV) Registers

- ◆ **IV = Interrupt Vector register**
- ◆ **Most MSP430 interrupts can be caused by more than one source; for example:**
 - ◆ Each 8-bit GPIO port has a single CPU interrupt
- ◆ **IV registers provide an easy way to determine which source(s) actually interrupted the CPU**
- ◆ **The interrupt vector register reflects only 'triggered' interrupt flags whose interrupt enable bits are also set**
- ◆ **Reading the 'IV' register:**
 - ◆ Clears the pending interrupt flag with the highest priority
 - ◆ Provides an address offset associated with the highest priority pending interrupt source
- ◆ **An example is provided in the "Coding Interrupts" section of this chapter**

For grouped interrupts, most users read the IV register at the beginning of the ISR and use the return value to pick the appropriate code to run. This is usually implemented with a Switch/Case statement. (We will explore an example of this code later in the chapter.)

Interrupt Vector Table

We can expand the previous interrupt source & priority listing to include a few more items. First of all, we added a column that provides the IV register associated with each interrupt. (Note, the two names shown in red text represent the IFG bits for dedicated/individual interrupts.)

Additionally, the first 3 rows (highlighted with red background fill) indicate that these interrupt groups are non-maskable; therefore, they bypass the GIE bit.

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

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

The final column in the above diagram hints at the location of each interrupts address vector in the memory map. For example, when using the WDT as an interval timer, you would put the address of your appropriate ISR into location "57". As we saw in a previous topic, this can easily be done using the *vector* pragma.

The MSP430 devices reserve the range 0xFFFF to 0xFF80 for the interrupt vectors. This means that for the 'F5529, the address for the System Reset interrupt service routine will sit at addresses 0xFFFFE – 0xFFFF. (A 16-bit address requires two 8-bit memory locations.) The remaining interrupt vectors step down in memory from this point. The map to the right of the table shows where the interrupt vectors appear within the full MSP430 memory map.

Here's a quick look at the same table showing the MSP430FR5969 interrupt vectors and priorities. The list is very similar to the 'F5529; the main differences stem from the fact that the two devices have a slightly different mix of peripherals.

Interrupt Vectors & Priorities ('FR5969)

INT Source	IV Register	Vector Address	Loc'n	Priority
System Reset	SYSRSTIV	RESET_VECTOR		high
System NMI	SYSSNIV	SYSNMI_VECTOR	54	
User NMI	SYSUNIV	UNMI_VECTOR	53	
Comparator_E	CEIV	COMP_B_VECTOR	52	
Timer B0 (CCIFG0)	CCIFG0	TIMER0_B0_VECTOR	51	
Timer B0	TB0IV	TIMER0_B1_VECTOR	50	
WDT Interval Timer	WDTIFG	WDT_VECTOR	49	
Serial Port (A0)	UCA0IV	USCI_A0_VECTOR	48	
Serial Port (B0)	UCB0IV	USCI_B0_VECTOR	47	
ADC12_B	ADC12IV	TIMER0_B0_VECTOR	46	
GPIO (Port 1)	P1IV	PORT1_VECTOR	39	
Real-Time Clock	RTCIV	RTC_VECTOR	31	
AES256 Accelerator	AESRDYIFG	AES256_VECTOR	30	low

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

The preceding interrupt tables were re-drawn to make them easier to view when projected during a workshop. The following slide was captured from 'F5529 datasheet. This is what you will see if you examine the MSP430 documentation.

'F5529 Vector Table (From Datasheet)

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
System Reset Power-Up External Reset Watchdog Timeout, Password Violation Flash Memory Password Violation	WDTIFG, KEYV (SYSRSTIV) ⁽¹⁾⁽²⁾	Reset	0FFFEh	63, highest
System NMI PMM Vacant Memory Access JTAG Mailbox	SVMLIFG, SVMHIFG, DLYLIFG, DLYHIFG, VLRLLIFG, VLRHIFG, VMAIFG, JMBNIFG, JMBOUTIFG (SYSSNIV) ⁽¹⁾	(Non)maskable	0FFCh	62
User NMI NMI Oscillator Fault Flash Memory Access Violation	NMIIFG, OFIFG, ACCVIFG, BUSIFG (SYSUNIV) ⁽¹⁾⁽²⁾	(Non)maskable	0FFAh	61
Comp B Vector B interrupt flags (CBVIFG)		Maskable	0FFE0h	60
DMA	DMA0IFG, DMA1IFG, DMA2IFG (DMAIV) ⁽¹⁾⁽³⁾	Maskable	0FFE4h	50
TA1	TA1CCR0 CCIFG0 ⁽³⁾	Maskable	0FFE2h	49
TA1	TA1CCR1 CCIFG1 to TA1CCR2 CCIFG2, TA1IFG (TA1IV) ⁽¹⁾⁽³⁾	Maskable	0FE0h	48
I/O Port P1	P1IFG.0 to P1IFG.7 (P1IV) ⁽¹⁾⁽³⁾	Maskable	0FDEh	47
USCI_A1 Receive or Transmit	UCA1RXIFG, UCA1TXIFG (UCA1IV) ⁽¹⁾⁽³⁾	Maskable	0FFDCh	46
USCI_B1 Receive or Transmit	UCB1RXIFG, UCB1TXIFG (UCB1IV) ⁽¹⁾⁽³⁾	Maskable	0FFDAh	45
TA2	TA2CCR0 CCIFG0 ⁽³⁾	Maskable	0FFD8h	44
TA2	TA2CCR1 CCIFG1 to TA2CCR2 CCIFG2, TA2IFG (TA2IV) ⁽¹⁾⁽³⁾	Maskable	0FFD6h	43
I/O Port P2	P2IFG.0 to P2IFG.7 (P2IV) ⁽¹⁾⁽³⁾	Maskable	0FFD4h	42
RTC_A	RTCRDYIFG, RTCTEVIFG, RTCAIFG, RT0PSIFG, RT1PSIFG (RTCIV) ⁽¹⁾⁽³⁾	Maskable	0FFD2h	41
			0FFD0h	40

Each device's datasheet provides a similar *vector table* listing. If you are using the 'G2553 or 'FR5969 devices, for example, you will find a similar table in each of their respective datasheets.

We will use the following code example to demonstrate these three items.

Interrupt Service Routine (Dedicated INT)

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

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

```

#pragma vector=WDT_VECTOR
__interrupt void myWdtISR(void) {
    GPIO_toggleOutputOnPin( ... );
}

```

Plug the Vector Table (#pragma vector)

In our example, the following line of code:

```
#pragma vector=WDT_VECTOR
```

tells the compiler to associate the function (on the following line) with the WDT_VECTOR. Looking in the MSP430F5529 device-specific linker command file, you should find this vector name (“WDT_VECTOR”) associated with vector #57. This matches with the datasheet documentation we looked at earlier in the chapter.

Save/Restore CPU context (__interrupt keyword)

The `__interrupt` keyword tells the compiler that this function is an interrupt service routine and thus it needs to save (and then restore) the context of the processor (i.e. CPU registers) before (and after) executing the function’s code.

Don’t forget, functions using the `__interrupt` keyword cannot accept arguments or return values.

Hint: Empirical analysis shows that “`__interrupt`” and “`interrupt`” are both accepted by the compiler.

Your Interrupt Code

In this example, the output of a GPIO pin is toggled every time the watchdog timer interrupt event occurs. Not all ISR’s will be this short, but we hope this gives you a good starting example to work from.

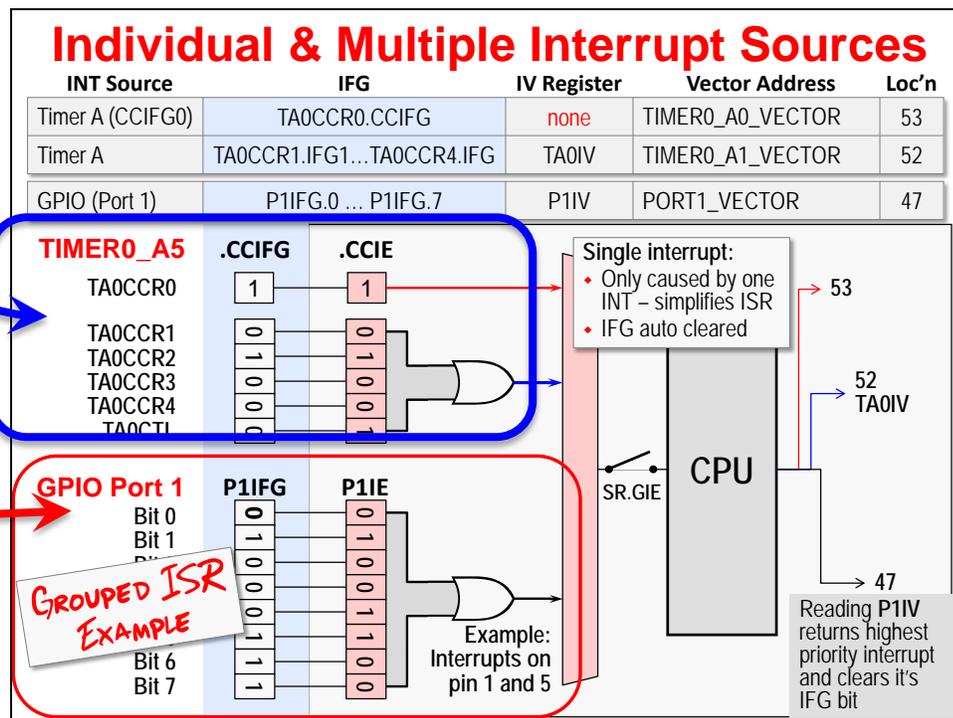
Grouped ISR (Interrupt Service Routine)

Logical Diagram for Grouped Interrupts

Before examining the code for a grouped ISR, let's first examine the grouped interrupt using a logical diagram.

As we briefly mentioned earlier in the chapter (and will discuss in full detail in a later chapter), the Timer_A and Timer_B peripherals are provided with two interrupts. For example, when looking at Timer0_A5, there is a dedicated interrupt for TA0CCR0 (which stands for Timer0_A Capture/Compare Register 0). Notice below how this is routed directly to the GIE input mux.

The remaining five Timer0_A5 interrupts are logically AND'd together; this combination provides a second interrupt signal from Timer0_A5 to the GIE input mux.



This diagram also shows that all of the input pins for GPIO port 1 (P1) share a single, grouped interrupt. This means your GPIO ISR must always verify which pin actually caused an interrupt whenever the ISR is executed.

The interrupt logic within the CPU recognizes each of these interrupt sources, therefore:

- If the first interrupt (TA0CCR0) occurs, it will cause the code at vector address 53 (TIMER_A0_VECTOR) to be executed.
- Similarly, the remaining Timer0 interrupts are associated with vector 52.
- Finally, the GPIO port (P1) was assigned (by the chip designer) to vector 47.

ISR Example for Grouped Interrupts

The code for a grouped ISR begins similar to any MSP430 interrupt service routine; you should use the `#pragma vector` and `__interrupt` keyword syntax.

Interrupt Service Routine (Group INT)

INT Source	IV Register	Vector Address	Loc'n
GPIO (Port 1)	P1IV	PORT1_VECTOR	47

- ◆ `#pragma vector` assigns 'myISR' to correct location in vector table
- ◆ `__interrupt` keyword tells compiler to save/restore context and RETI
- ◆ Reading P1IV register:
 - ◆ Returns value for highest priority INT for the Port 1 'group'
 - ◆ Clears IFG bit
- ◆ Tell compiler to ignore un-needed switch cases by using intrinsics:


```
__even_in_range()
__never_executed()
```

```

#pragma vector=PORT1_VECTOR
__interrupt void myISR(void) {
    switch(__even_in_range( P1IV, 10 )) {
        case 0x00: break; // None
        case 0x02: break; // Pin 0
        case 0x04: break; // Pin 1
        case 0x06: GPIO_toggleOutputOnPin(...); // Pin 2
                   break;
        case 0x08: break; // Pin 3
        case 0x0A: break; // Pin 4
        case 0x0C: break; // Pin 5
        case 0x0E: break; // Pin 6
        case 0x10: break; // Pin 7
        default:  __never_executed();
    }
}

```

For grouped interrupts, though, we also need to determine which specific source caused the CPU to be interrupted. As we've described, the Interrupt Vector (IV) register is an easy way to determine the highest-priority, pending interrupt source. In the case of GPIO port 1, we would read the P1IV register.

It's common to see the IV register read within the context of a switch statement. In the above case, if the P1IV register returns "6", it means that pin 2 was our highest-priority, enabled interrupt on Port 1; therefore, its case statement is executed. (Note, the return values for each IV register are detailed in the *F5xx device Users Guide* and the *F5xx DriverLib User's Guide*. You will find similar documentation for all MSP430 devices..)

If our program was using Pin 2 on Port 1, you should see the code for case 0x06 executed if the GPIO interrupt occurs.

By the way, there are two items in the above code example which help the compiler to produce better, more optimized, code. While these intrinsic functions are not specific to interrupt processing, they are useful in creating optimized ISR's.

- The `__even_in_range()` intrinsic function provides the compiler a bounded range to evaluate. In other words, this function tells the compiler to only worry about even results that are lower or equal to 10.
- Likewise the `__never_executed()` intrinsic tells the compiler that, in this case, "default" will never occur.

Enabling Interrupts

Earlier in the chapter we learned that for the CPU to recognize an interrupt two enable bits must be set:

- **Individual Enable** – one IE bit for each interrupt source
- **Global Interrupt Enable** – GIE is a common “master” enable bit for all interrupts (except those defined as non-maskable)

In the example below we show the code required to setup a GPIO pin as an interrupt. We chose to enable the interrupt, as well as configuring the other GPIO pins, in a function called `initGPIO()`; implementing your code in this way is not required, but it's how we decided to organize our code.

The key DriverLib function which enables the external interrupt is:

```
GPIO_enableInterrupt()
```

You will find that most of the MSP430ware DriverLib interrupt enable functions take a similar form: `<module>_enableInterrupt()`.

Enabling Interrupts – GPIO Example

```
#include <driverlib.h>
...
void main(void) {
    // Setup/Hold Watchdog
    initWatchdog();

    // Configure Power
    initPowerMgmt();

    // Configure GPIO
    initGPIO();

    // Setup Clocking
    initClocks();

    //-----
    // Then, configure
    ...

    __bis_SR_register( GIE );

    while(1) {
        ...
    }
}

void initGPIO() {
    // Set P1.0 as output & turn LED on
    GPIO_setAsOutputPin (
        GPIO_PORT_P1, GPIO_PIN0 );

    GPIO_setOutputLowOnPin (
        GPIO_PORT_P1, GPIO_PIN0 );

    // Set input & enable P1.1 as INT
    GPIO_setAsInputPinWithPullUpresistor (
        GPIO_PORT_P1, GPIO_PIN1 );

    GPIO_clearInterruptFlag (
        GPIO_PORT_P1, GPIO_PIN1 );

    GPIO_enableInterrupt (
        GPIO_PORT_P1, GPIO_PIN1 );
}

```

Within `initGPIO()` we highlighted two other related functions in **Red**:

- **GPIO_setAsInputPinWithPullUpresistor()** is required to configure the pin as an input. On the Launchpad, the hardware requires a pull-up resistor to complete the circuit properly. Effectively, this function configures our interrupt “source”.
- **GPIO_clearInterruptFlag()** clears the IFG bit associated with our pin (e.g. P1.1). This is not required but is commonly used right before a call to “enable” an interrupt. You would clear the IFG before setting IE when you want to ignore any prior interrupt event; in other words, clear the flag first if you only care about interrupts that will occur now – or in the future.

Finally, once you have enabled each individual interrupt, the global interrupt needs to be enabled. This can be done in a variety of ways. The two most common methods utilize compiler intrinsic functions:

- `__bis_SR_register(GIE)` instructs the compiler to set the GIE bit in the Status Register
 - bis = bit set
 - SR = Status Register
 - GIE = which bit to set in the SR
- `__enable_interrupts(void)` tells the compiler to enable interrupts. The compiler uses the EINT assembly instruction which pokes 1 into the GIE bit.

Sidebar – Where in your code should you enable GIE?

The short answer, “Whenever you need to turn on interrupts”.

A better answer, as seen in our code example, is “right before the while{} loop”.

Conceptually, the main() function for most embedded systems consists of two parts:

- Setup
- Loop

That is, the first part of the main() function is where we tend to setup our I/O, peripherals, and other system hardware. In our example, we setup the watchdog timer, power management, GPIO, and finally the system clocks.

The second part of main() usually involves an infinite loop – in our example, we coded this with an endless while{} loop. An infinite loop is found in almost all embedded systems since we want to run forever after the power is turned on.

The most common place to enable interrupts globally (i.e. setting GIE) is right between these two parts of main(). Looking at the previous code example, this is right where we placed our function that sets GIE.

As a product example, think of the A/C power adaptor you use to charge your computer; most of these, today, utilize an inexpensive microcontroller to manage them. (In fact, the MSP430 is very popular for this type of application.) When you plug in your power adapter, we’re guessing that you would like it to run as long as it’s plugged in. In fact, this is what happens; once plugged in, the first part of main() sets up the required hardware and then enters an endless loop which controls the adaptor. What makes the MSP430 such a good fit for this application is: (1) it’s inexpensive; and (2) when a load is not present and nothing needs to be charged, it can turn off the external charging components and put itself to sleep – until a load is inserted and wakes the processor back up.

Miscellaneous Topics

Handling Unused Interrupts

While you are not required to provide interrupt vectors – or ISR’s – for every CPU interrupt, it’s considered good programming practice to do so. To this end, the MSP430 compiler issues a warning whenever there are “unhandled” interrupts.

The following code is an example that you can include in all your projects. Then, as you implement an interrupt and write an ISR for it, just comment the associated `#pragma` line from this file.

Handling Unused Interrupts

- ◆ The MSP430 compiler issues warning whenever all interrupts are not handled (i.e. when you don’t have a vector specified for each interrupt)
- ◆ Here’s a simple example of how this might be handled:

```
// Example for UNUSED_HWI_ISR()
#pragma vector=ADC12_VECTOR
#pragma vector=COMP_B_VECTOR
#pragma vector=DMA_VECTOR
#pragma vector=PORT1_VECTOR
...
#pragma vector=TIMER1_A1_VECTOR
#pragma vector=TIMER2_A0_VECTOR
#pragma vector=TIMER2_A1_VECTOR
#pragma vector=UNMI_VECTOR
#pragma vector=USB_UBM_VECTOR
#pragma vector=WDT_VECTOR
__interrupt void UNUSED_HWI_ISR (void)
{
    __no_operation();
}
```

Note: The TI code generation tools distinguish between “warnings” and “errors”. Both represent issues found during compilation and build, but whereas a *warning* is issued and code building continues ... when an *error* is encountered, an *error* statement is issued and the tools stop before creating a final executable.

Interrupt Service Routines – Coding Suggestions

Listed below are a number of required and/or good coding practices to keep in mind when writing hardware interrupt service routines. Many of these have been discussed elsewhere in this chapter.

Hardware ISR's – Coding Practices

- ◆ An interrupt routine must be declared with no arguments and must return void
- ◆ Do not call interrupt handling functions directly (Rather, write to IFG bit)
- ◆ Interrupts can be handled directly with C/C++ functions by using the the *interrupt* keyword or pragma
... Conversely, the TI-RTOS kernel easily manages *Hwi* context
- ◆ Calling functions in an ISR
 - ◆ If a C/C++ interrupt routine doesn't call other functions, only those registers that the interrupt handler uses are saved and restored.
 - ◆ However, if a C/C++ interrupt routine does call other functions, the routine saves all the save-on-call registers if any other functions are called
 - ◆ Why? The compiler doesn't know what registers could be used by a nested function. It's safer for the compiler to go ahead and save them all.
- ◆ Re-enable interrupts? (Nesting ISR's)
 - ◆ **DON'T** – it's not recommended – better that ISR's are "lean & mean"
 - ◆ If you do, change IE masking before re-enabling interrupts
 - ◆ Disable interrupts before restoring context and returning (RETI re-enables int's)
- ◆ Beware – Only You Can Prevent Reentrancy...

We wrote the last bullet, regarding reentrancy, in a humorous fashion. That said, it speaks to an important point. If you decide to enable interrupt nesting, you need to be careful that you either prevent reentrancy - or that your code is capable of reentrancy.

Wikipedia defines reentrancy as:

*In computing, a computer program or subroutine is called **reentrant** if it can be interrupted in the middle of its execution and then safely called again ("re-entered") before its previous invocations complete execution.*

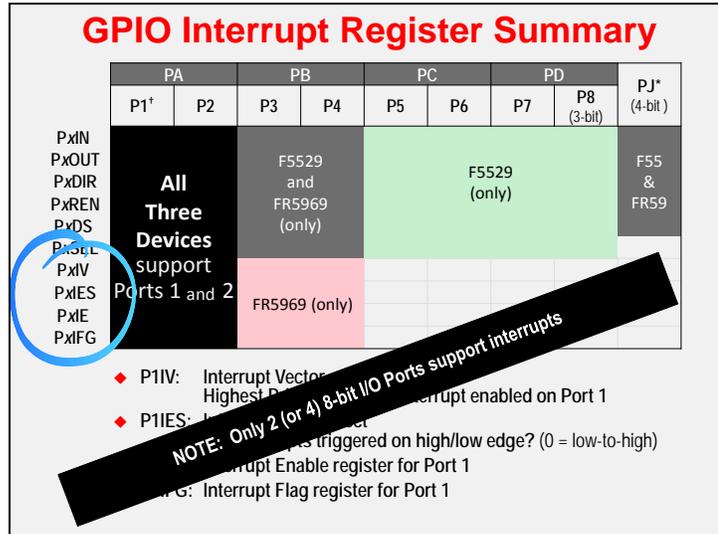
This type of program/system error can be very difficult to debug (i.e. find and fix). This is especially true if you call functions within your interrupt service routines. For example, the C language's `malloc()` function is not reentrant. If you were to call this function from an ISR and it was interrupted, and then it is called again by another ISR, your system would most likely fail – and fail in a way that might be very difficult to detect.

So, we stated this humorously, but it is very true. We recommend that:

- You shouldn't nest interrupts
- If you do, verify the code in your ISR is reentrant
- Never call `malloc()` – or similar functions - from inside an ISR

GPIO Interrupt Summary

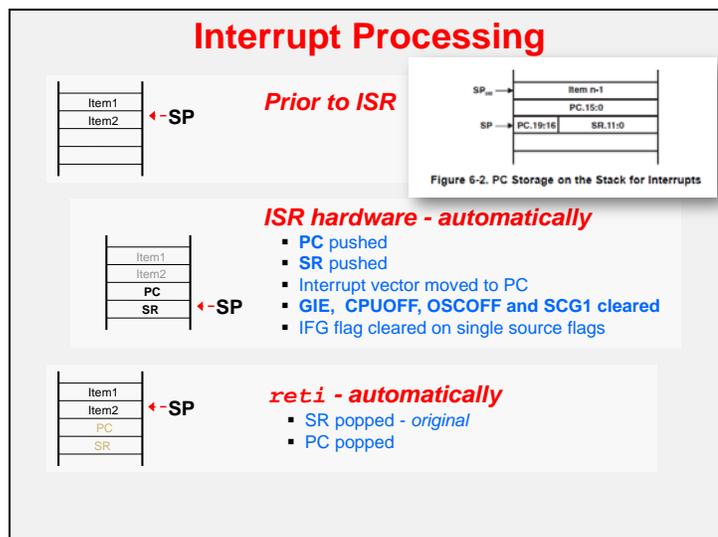
The diagram used to summarize the GPIO control registers in a previous chapter is a good way to visualize the GPIO interrupt capabilities of our devices. From the diagram below we can see that most MSP430 processors allow ports P1 and P2 to be used as external interrupt registers; we see this from the fact that these ports actually have the required port interrupt registers.



There are other devices in the MSP430 family that support interrupts on more than 2 ports, but of the three example processors we use throughout this course, only the FR5969 (Wolverine) devices support interrupt inputs on additional ports (P3 and P4).

Interrupt Processing Flow

The following information was previously covered in this chapter, but since the slide is a good summary of the interrupt processing flow, we have included it anyway.



Interrupts and TI-RTOS Scheduling

When embedded systems start to become more complex – that is, when you need to juggle more than a handful of events – using a Real-Time Operating System (RTOS) can greatly increase your system’s reliability ... while decreasing your time-to-market, frustration and costs.

The Texas Instruments RTOS (TI-RTOS) – also known as SYS/BIOS – provides many functions that you can use within your program; for example, the TI-RTOS kernel includes: *Scheduling*, *Instrumentation*, and *Memory Management*. You can choose which parts of TI-RTOS are needed and discard the rest (to save memory).

Think of TI-RTOS as a library and toolset to help you build and maintain robust systems. If you’re doing just “one” thing, it’s probably overkill. As you end up implementing more and more functionality in your system, though, the tools and code will save you time and headaches.

The only part of TI-RTOS discussed in this chapter is “Scheduling”. We talk about this because it is very much related to the topics covered throughout this chapter – interrupts and threads. In many cases, if you’re using an RTOS, it will manage much of the interrupt processing for you; it will also provide additional options for handling interrupts – such as post-processing of interrupts.

As a final note, we will only touch on the topics of scheduling and RTOS’s. TI provides a 2-day workshop where you can learn all the details of the TI-RTOS kernel. You can view a video version of the TI-RTOS course or take one live. Please check out the following wiki page for more information:

http://processors.wiki.ti.com/index.php/Introduction_to_the_TI-RTOS_Kernel_Workshop

Threads – Foreground and Background

Our quick introduction to TI-RTOS begins with a summary of threads. While we discussed these concepts earlier in the chapter, they are very important to how a RTOS scheduler works.

What is a Thread?

The diagram illustrates the execution flow of a program. It is divided into three main sections:

- main() {**: Contains **init code** and a **while(1) { nonRT Fxn }** loop. A red dashed arrow labeled "Background thread" points to the while loop.
- UART ISR**: Contains **get byte**, **process**, and **output**. A red dashed arrow labeled "Foreground threads" points to this section.
- Timer ISR**: Contains **Scan keyboard**. A red dashed arrow labeled "Foreground threads" points to this section.

- ◆ We all know what a **function()** is...
- ◆ A **thread** is a **function** that runs within a specific context; e.g.
 - Priority
 - Registers/CPU state
 - Stack
- ◆ To retain a thread’s context, we must **save** ---->


```

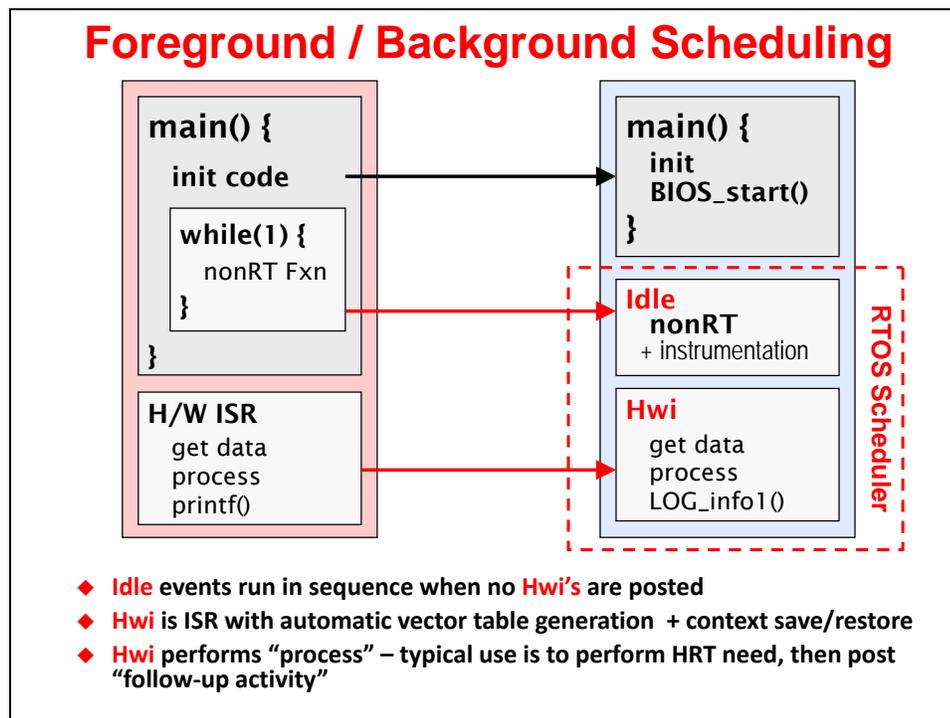
Thread wrapper (C/S)
void my_fxn()
{
  int m, x, b;
  int y;

  y = m*x + b;
  serial = y;
  results += 1;
}
            
```
- then **restore** it ---->


```

Thread wrapper (C/R)
            
```
- ◆ Most common threads in a system are **hardware interrupts**

We also discussed the idea of foreground and background threads as part of the interrupts chapter. In the case shown below (on the left), the endless loop in main() will run forever and be pre-empted by higher-priority hardware interrupts.



TI-RTOS utilizes these same concepts ... only the names and threads change a little bit.

Rather than main() containing both the *setup* and *loop* code as described earlier, TI-RTOS creates an **Idle** thread that operates in place of the while{} loop found previously in main(). In other words, rather than adding your functions to a while{} loop, TI-RTOS has you add them to **Idle**. (TI-RTOS includes a GUI configuration tool that makes this very easy to do.)

Since interrupts are part of the MSP430's hardware, they essentially work the same way when using TI-RTOS. What changes when using RTOS are:

- TI-RTOS calls them **Hwi** threads ... for Hardware Interrupts
- Much of the coding effort is handled automatically for you by TI-RTOS (very nice)

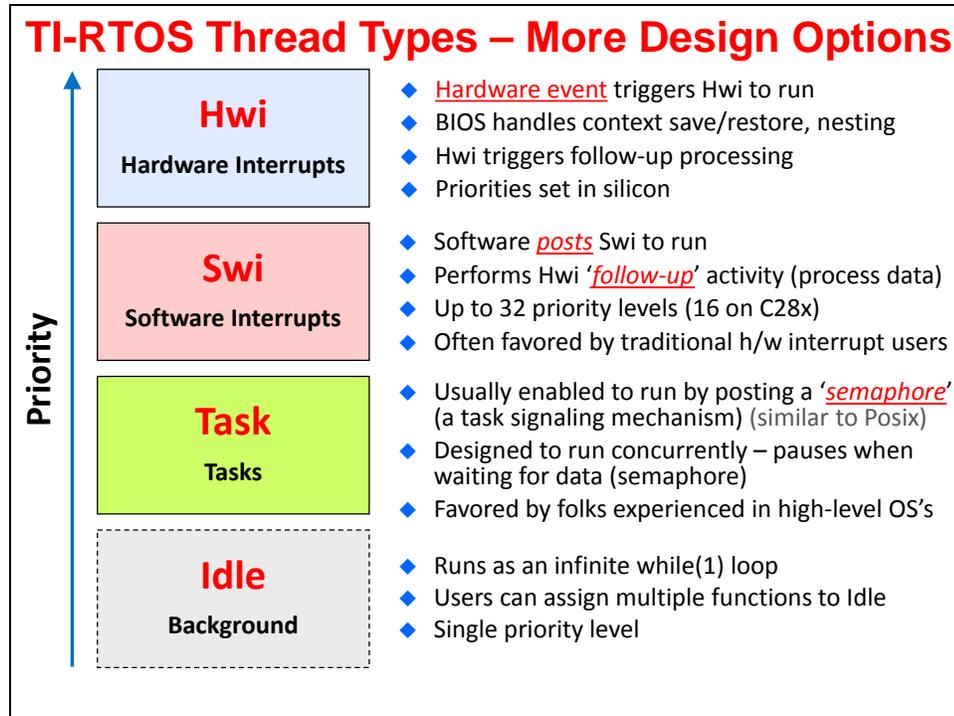
Don't worry, though, you're not locked into anything. You can mix-and-match how you handle interrupts. Let TI-RTOS manage some of your interrupts while handling others in your own code, just as we described in this chapter.

Hint: When using TI-RTOS, the author prefers to let it manage all of the interrupts because it's easier that way. Only

Only in a rare case – like to save a few CPU cycles – would there be a need to managed an interrupt outside of TI-RTOS. Thusfar, the only reason I've actually done this is to provide that it works.

TI-RTOS Thread Types

We already described two types of threads: **Hwi** and **Idle**. Using these two threads is very similar to what we described throughout this chapter.



TI-RTOS provides two additional thread types: **Software Interrupts (Swi)** and **Tasks (Task)**. As you can see above, these thread types fall between **Hwi** and **Idle** in terms of priority.

Each of these threads can be used to extend your system's processing organization.

What do we mean by this?

You might remember that we HIGHLY recommended that you DO NOT nest interrupts. But what happens if you want to run an algorithm based on some interrupt event? For example, you want to run a filter whenever you receive a value from an A/D converter or from the serial port.

Without an RTOS, you would need to organize your main while{} loop to handle all of these interrupt, follow-up tasks. This is not a problem for one or two events; but for lots of events, this can become very complicated – especially when they all run at different rates. This way of scheduling your processing is called a *SuperLoop*.

With an RTOS, we can post follow-up activity to a Swi or Task. A **Swi** acts just like a software triggered interrupt service routine. **Tasks**, on the other hand, run all the time (*have you heard the term multi-tasking before?*) and utilize **Semaphores** to signal when to run or when to *block* (i.e. pause).

In other words, **Swi's** and **Task's** provide two different ways to schedule follow-up processing code. They let us keep our hardware interrupts (Hwi's) very short and simple – for example, all we need to do is read our ADC and then post an associated **Swi** to run.

If all of this sounds complicated, it really isn't. While outside the scope of this course, the TI-RTOS course will have you up-and-running in no time. Once you experience the effective organization provided by an RTOS, you may never build another system without one.

TI-RTOS Details

The following slide provides some “characteristics” of the TI-RTOS kernel. The bottom-line here is that it is a priority-based scheduler. The highest priority thread gets to run, period. (Remember, hardware interrupts are always the highest priority.)

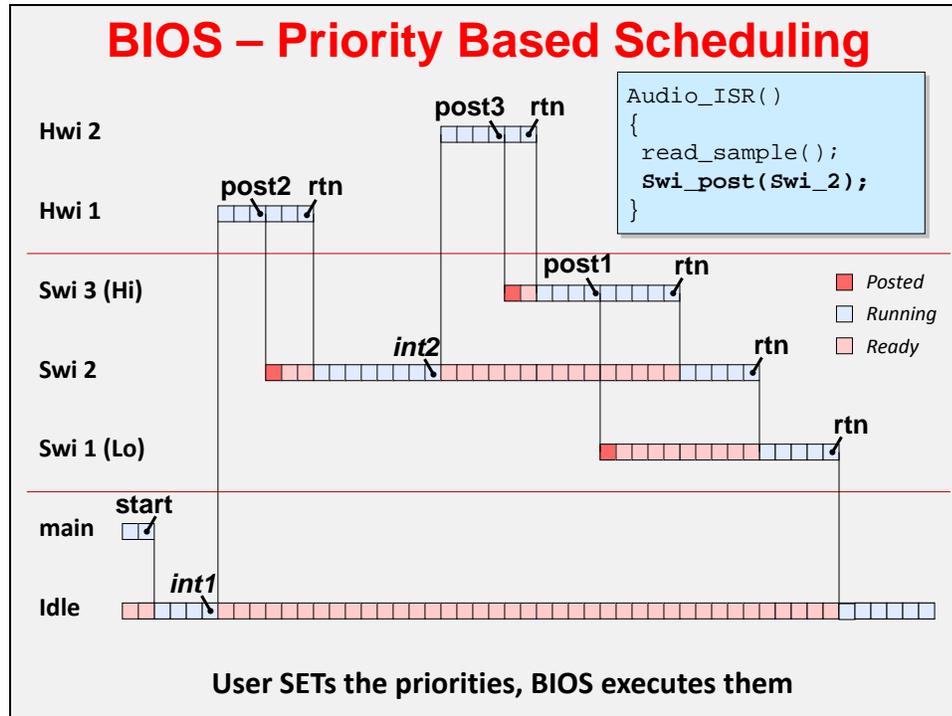
TI-RTOS Kernel – Characteristics

- ◆ RTOS means “Real-time O/S” – so the intent of this O/S is to provide common services to the user WITHOUT disturbing the real-time nature of the system
- ◆ The TI-RTOS Kernel (SYS/BIOS) is a PRE-EMPTIVE scheduler. This means the highest priority thread ALWAYS RUNS FIRST. Time-slicing is not inherently supported.
- ◆ The kernel is EVENT-DRIVEN. Any kernel-configured interrupts or user calls to APIs such as `Swi_post()` will invoke the scheduler. The kernel is NOT time-sliced although threads can be triggered on a time bases if so desired.
- ◆ The kernel is OBJECT BASED. All APIs (methods) operate on self-contained objects. Therefore when you change ONE object, all other objects are unaffected.
- ◆ Being object-based allows most RTOS kernel calls to be DETERMINISTIC. The scheduler works by updating event queues such that all context switches take the same number of cycles.
- ◆ Real-time Analysis APIs (such as Logs) are small and fast – the intent is to LEAVE them in the program – even for production code – yes, they are really that small

While you can construct a time-slicing system using TI-RTOS, this is not commonly done. While time-slicing can be a very effective technique in host operating systems (like Windows or Linux), it is not a common method for scheduling threads in an embedded system.

Hwi – Swi – Idle Scheduling

Here's a simple, visual example of what real-time scheduling might look like in an RTOS based system.



Notice how the system enters Idle from main(). Idle is always ready to run (just as our old while{} loop was always ready to run).

When a hardware interrupt (Hwi) occurs, we leave Idle and execute the Hwi thread's code. Since it appears the Hwi posted a Swi, that's where the TI-RTOS scheduler goes to once the Hwi finishes.

We won't go through the remaining details in this course, though we suspect that you can all follow the diagram. For this slide, and a lot more information, please refer to the TI-RTOS Kernel Workshop.

Summary: TI-RTOS Kernel

The following slide summarizes much of the functionality found in the TI-RTOS kernel. In this chapter we've only touched on the scheduling features.

TI-RTOS Kernel Services

TI-RTOS Kernel (i.e. SYS/BIOS) is a *library of services* that users can add to their system to perform various tasks:

- ◆ **Memory Mgmt** (stack, heap, cache)
- ◆ **Real-time Analysis** (logs, graphs, loads)
- ◆ **Scheduling** (various thread types)
- ◆ **Synchronization** (e.g. semaphores, events)

Attend the 2-day TI RTOS kernel workshop for more information on all of these services

The TI-RTOS product includes the kernel, shown above, along with a number of additional drivers and stacks. Oh, and the kernel comes with complete source code – nothing is hidden from you.

For many, though, one of the compelling features of TI-RTOS is that it's FREE*.

Remember, we make our money selling you devices. Our code and tools are there to help you get your programs put together – and your systems to market – more quickly.

* That is, it's free for use on all Texas Instruments processors.

Lab 5 – Interrupts

This lab introduces you to programming MSP430 interrupts. Using interrupts is generally one of the core skills required when building embedded systems. If nothing else, it will be used extensively in later chapters and lab exercises.

Lab 5 – Button Interrupts

- ◆ **Lab Worksheet... a Quiz, of sorts:**
 - Interrupts
 - Save/Restore Context
 - Vectors and Priorities
- ◆ **Lab 5a – Pushing your Button**
 - Create a CCS project that uses an interrupt to toggle the LED when a button is pushed
 - This requires you to create:
 - Setup code enabling the GPIO interrupt
 - GPIO ISR for pushbutton pin
 - You'll also create code to handle all the interrupt vectors
- ◆ **Optional**
 - **Lab 5b – Use the Watchdog Timer**
Use the WDT in interval mode to blink the an LED

Time:
Worksheet – 15 mins
Labs – 45 mins



The image shows a red MSP430 LaunchPad development board. It features a central MSP430 microcontroller, a pushbutton (SW1), and an LED. The board is populated with various components including resistors, capacitors, and a USB-to-UART bridge. The Texas Instruments logo is visible on the board.

Lab 5a covers all the essential details of interrupts:

- Setup the interrupt vector
- Enable interrupts
- Create an ISR

When complete, you should be able to push the SW1 button and toggle the Red LED on/off.

Lab 5b is listed as optional since, while these skills are valuable, you should know enough at the end of Lab 5a to move on and complete the other labs in the workshop.

Lab Topics

Interrupts	5-36
<i>Lab 5 – Interrupts</i>	<i>5-37</i>
Lab 5 Worksheet	5-39
General Interrupt Questions	5-39
Interrupt Flow	5-39
Interrupt Priorities & Vectors	5-40
ISR's for Group Interrupts	5-41
Lab 5a – Push Your Button	5-42
File Management	5-42
Configure/Enable GPIO Interrupt ... Then Verify it Works.....	5-45
Add a Simple Interrupt Service Routine (ISR)	5-48
Sidebar – Vector Errors.....	5-48
Upgrade Your Interrupt Service Routine (ISR)	5-50
(Optional) Lab 5b – Can You Make a Watchdog Blink?	5-51
Import and Explore the WDT_A Interval Timer Example	5-51
Run the code	5-53
Change the LED blink rate	5-53
Appendix	5-54

Lab 5 Worksheet

General Interrupt Questions

1. When your program is not in an interrupt service routine, what code is it usually executing? And, what 'name' do we give this code?

2. Why keep ISR's short (i.e. Why shouldn't you do a lot of processing in them)?

3. What causes the MSP430 to exit a Low Power Mode (LPMx)?

4. Why are *interrupts* generally preferred over *polling*?

Interrupt Flow

5. Name 4 sources of interrupts? (*Well, we gave you one, so name 3 more.*)

TIMER A

6. What signifies that an interrupt has occurred?

A _____ bit is set

What's the acronym for these types of 'bits' _____

7. Write the code to enable a GPIO interrupt on Port 1, pin1 (aka P1.1)?

_____ // setup pin as input

_____ // clear individual flag

_____ // enable individual interrupt

8. Write the line of code required to turn on interrupts globally:

_____ // enable global interrupts (GIE)

Where, in our programs, is the most common place we see GIE enabled? (*Hint, you can look back at the slides where we showed how to do this.*)

Interrupt Priorities & Vectors

9. Circle the interrupt that has higher priority: GPIO Port 2 or WDT Interval Timer?

Let's say you're CPU is in the middle of the GPIO Port 2 ISR, can it be interrupted by a new WDT interval timer interrupt? If so, is there anything you could do to your code in order for this to happen?

10. Where do you find the name of an "interrupt vector"?

11. How do you write the code to set the interrupt vector? (Hint, we've provided a simple ISR to go with the line of code we're asking you to complete.)

```
// Sets ISR address in the vector for Port 1
```

#pragma

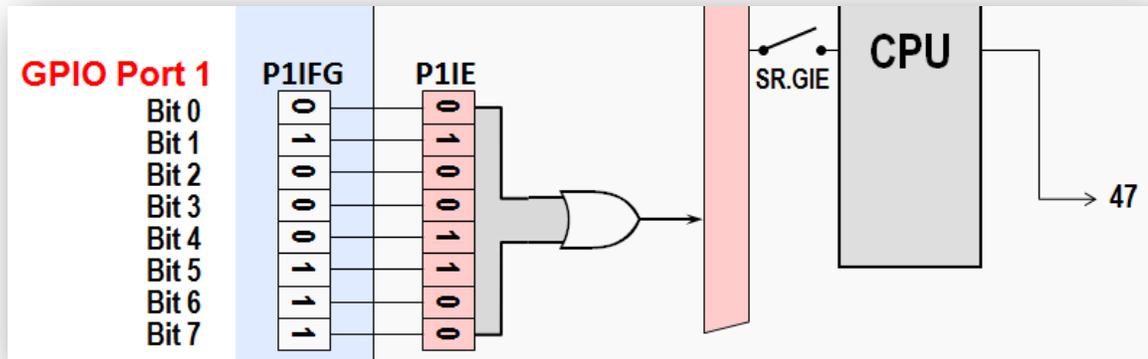
```
__interrupt void pushbutton_ISR (void)
{
    // Toggle the LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```

What is wrong with this GPIO port ISR?

12. How do you pass a value into (or out from) and interrupt service routine (ISR)?

ISR's for Group Interrupts

As we learned earlier, most MSP430 interrupts are grouped. For example, the GPIO port interrupts are all grouped together.



13. For dedicated interrupts (such as WDT interval timer) the CPU clears the IFG flag when responding to the interrupt. How does an IFG bit get cleared for group interrupts?

14. Creating ISR's for grouped interrupts is as easy as following a 'template'. The following code represents a grouped ISR template. Fill in the blanks required for the CPU to toggle the LED (P1.0) on in response to a GPIO pushbutton interrupt (P1.1).

```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {
    switch(__even_in_range( _____, 10 )) {
        case 0x00: break;           // None
        case 0x02: break;           // Pin 0
        case 0x04:                   // Pin 1

        _____

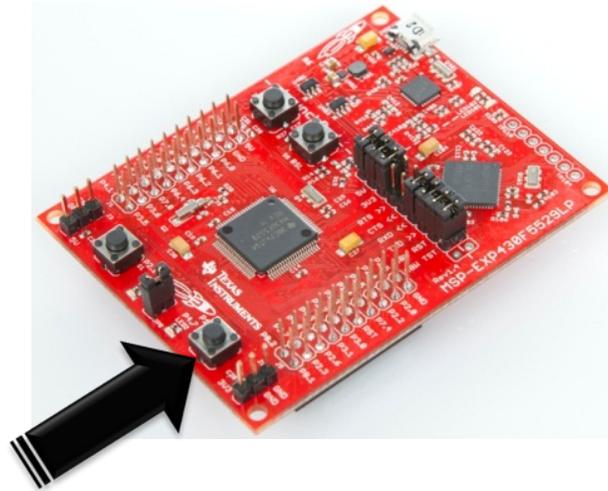
        break;
        case 0x06: break;           // Pin 2
        case 0x08: break;           // Pin 3
        case 0x0A: break;           // Pin 4
        case 0x0C: break;           // Pin 5
        case 0x0E: break;           // Pin 6
        case 0x10: break;           // Pin 7
        default:  __never_executed();
    }
}
```

Lab 5a – Push Your Button

When Lab 5a is complete, you should be able to push the S2 button and toggle the Red LED on/off.

We will begin by importing the solution to Lab 4a. After which we'll need to delete a bit of 'old' code and add the following.

- Setup the interrupt vector
- Enable interrupts
- Create an ISR

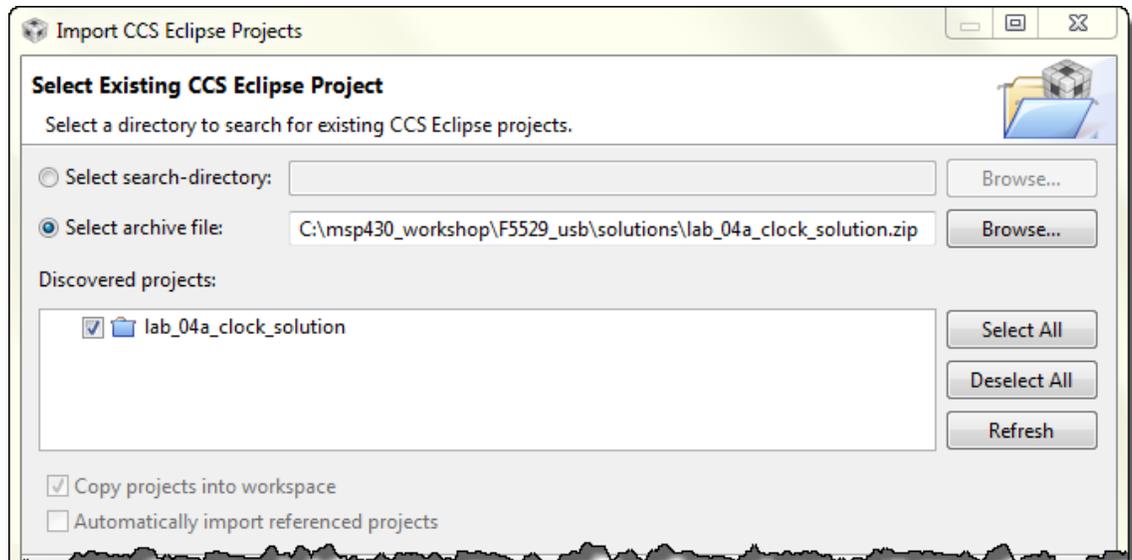


File Management

1. Close all previous projects. Also, close any remaining open files.
2. Import the solution for Lab 4a from: lab_04a_clock_solution

Select import previous CCS project from the *Project* menu:

Project → Import Existing CCS Eclipse Project

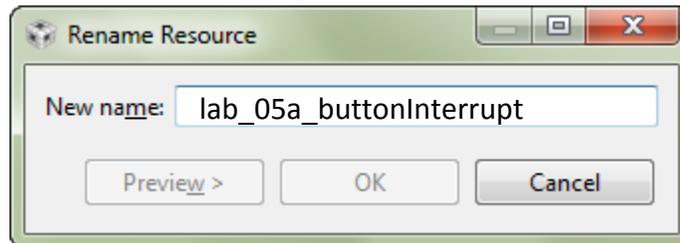


3. Rename the imported project to: lab_05a_buttonInterrupt

You can right-click on the project name and select *Rename*, though the easiest way to rename a project is to:

Select project in Project Explorer → hit 

When the following dialog pops up, fill in the new project name:



4. Verify the project builds and runs.

Before we change the code, let's make sure the original project is working. Build and run the project – you should see the LED flashing once per second.

5. Add unused_interrupts.c file to your project.

To save a lot of typing (and probably typos) we already created this file for you. You'll need to add it to your project.

Right-click project → Add Files...

Find the file in:

```
C:\msp430_workshop\<>target>\lab_05a_buttonInterrupt\unused_interrupts.c
```

You can take a quick look at this file, if you'd like. Notice that we created a single ISR function that is associated with all of the interrupts on your device – since, at this point, all of the interrupts are unused. As you add each interrupt to the project, you will need to modify this file.

6. Before we start adding new code ... delete old code from while{} loop.

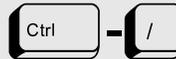
Open `main.c` and comment out – or delete – the code in the `while{} loop`. This is the old code that flashes the LED using the inefficient `__delay_cycles()` function.

```
30 while(1) {
31 //     // Turn on LED
32 //     GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
33 //
34 //     // Wait about a second
35 //     __delay_cycles( HALF_SECOND );
36 //
37 //     // Turn off LED
38 //     GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
39 //
40 //     // Wait another second
41 //     __delay_cycles( HALF_SECOND );
42 }
43 }
```

After commenting out the while code, just double-check for errors by clicking the build button. (Fix any error that pops up.)



Hint: If you are commenting out the code, it's easiest to select all the code and hit the `Ctrl-/` keys:



This toggles the line comments on/off.

Configure/Enable GPIO Interrupt ... Then Verify it Works

Add Code to Enable Interrupts

7. Open `main.c` and modify `initGPIO()` to enable the interrupt for your push-button.

If you need a hint on what three lines are required, refer back to the Lab 5 Worksheet, question number 7 (see page 5-39).

Note that the pin numbers are the same, but the switch names differ for these Launchpads:

- For the 'F5529 Launchpad, we're using pushbutton S2 (P1.1)
- For the 'FR5969 Launchpad, we're using pushbutton S3 (P1.1)

8. Add the line of code needed to enable interrupts globally (i.e GIE).

This line of code should be placed right before the `while()` loop in `main()`. Refer back to the Lab 5 Worksheet, question number 8 (see page 5-40).



9. Build your code.

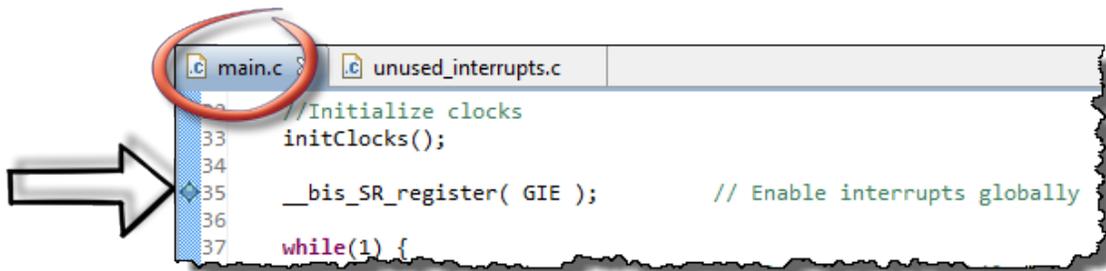
Fix any typos or errors.

Start the Debugger and Set Breakpoints

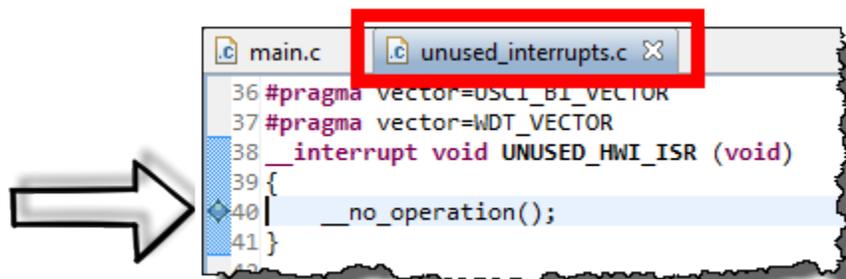
Once the debugger opens, we'll setup two breakpoints. This allows us to verify the interrupts were enabled, as well as trapping the interrupt when it occurs.

10. Launch the debugger.

11. Set a breakpoint on the "enable GIE" line of code in `main.c`.



12. Next, set a breakpoint inside the ISR in the `unused_interrupts.c` file.



Run Code to Verify Interrupts are Enabled



13. Click Run ... the program should stop at your first breakpoint.

14. Open the Registers window in CCS (or show it, if it's already open).

If the Registers window isn't open, do so by:

View → Registers

15. Verify Port1 bits: DIR, OUT, REN, IE, IFG.

The first breakpoint (should have) halted the processor right before setting the GIE bit. We'll look at that in a minute; for now, we want to view the GPIO Port 1 settings. Scroll/expand the registers to verify:

- P1DIR.0 = 1 (pin in output direction)
- P1DIR.1 = 0 (input direction – to be used for generating an interrupt)
- P1REN.1 = 1 (we enabled the resistor for our input pin)
- P1OUT.0 = 0 (we set it low to turn off LED)
- P1IE.1 = 1 (our button interrupt is enabled)
- P1IFG.1 = 0 (at this point, we shouldn't have received an interrupt – unless you already pushed the button...)

Here's a snapshot of the P1IE register as an example ...

Register Name	Value	Description
P1IE	0x02	Port 1 Interrupt Enable
P1IE7	0	P1IE7
P1IE6	0	P1IE6
P1IE5	0	P1IE5
P1IE4	0	P1IE4
P1IE3	0	P1IE3
P1IE2	0	P1IE2
P1IE1	1	P1IE1
P1IE0	0	P1IE0
P1IFG	0x00	Port 1 Interrupt Flag
P1IFG7	0	P1IFG7

16. Next, let's look at the Status Register (SR).

You can find it under the Core Registers at the top of the Registers window.

You should notice that the GIE bit equals 0, since we haven't executed the line of code enabling interrupts globally, yet.

Name	Value	Description
Core Registers		
PC	0x004E1A	Core
SP	0x0043FC	Core
SR	0x0000	Core
V	0	Overflow bit. T
SCG1	0	System clock g
SCG0	0	System clock g
OSCOFF	0	Oscillator Off. T
CDUOFF	0	CPU off. This b
GIE	0	General interr
N	0	Negative bit. Th
Z	0	Zero bit. This b



17. Single-step (i.e. Step-Over) the processor and watch GIE change.

Click the toolbar button or tap the  key. Either way, the *Registers* window should update:

Name	Value	Description
Core Registers		
PC	0x0043FC	Core
SP	0x0043FC	Core
SR	0x0008	Core
V	0	Overflow bit. This bit is set when the res
SCG1	0	System clock generator 1. This bit, when
SCG0	0	System clock generator 0. This bit, when
OSCOFF	0	Oscillator Off. This bit, when set, turns o
CPUOFF	0	CPU off. This bit, when set, turns off the
GIE	1	General interrupt enable. This bit, when
N	0	Negative bit. This bit is set when the res
Z	0	Zero bit. This bit is set when the result o
C	0	Carry bit. This bit is set when the result e

Testing your Interrupt

With everything setup properly, let's have a go at it.



18. Click *Resume* (i.e. Run) ... and nothing should happen.

In fact, if you *Suspend* (i.e. Halt) the processor, you should see that the code is sitting in the `while{}` loop, as expected.



19. Press the appropriate pushbutton (connected to P1.1) on your board.

Did that cause the program to stop at the breakpoint we set in the ISR?

If you hit *Suspend* in the previous step, did you remember to hit *Resume* afterwards?

(If it didn't stop, and you cannot figure out why, ask a neighbor/instructor for help.)

Add a Simple Interrupt Service Routine (ISR)

20. Add your Port 1 (P1.1) ISR to the bottom of main.c.

Here's a simple ISR routine that you can copy/paste into your code.

```
//*****
// Interrupt Service Routines
//*****
#pragma vector=????
__interrupt void pushbutton_ISR (void)
{
    // Toggle the LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```

Don't forget to fill in the ??? with your answer from question 11 from the worksheet (see page 5-40).



21. Build your program to test for any errors.

You should have gotten the error ...

```
./driverlib/MSP430F5xx_6xx/adc10_a.obj" "./unused_interrupts.obj" "./myClocks.obj" "./main.obj" "./lnk_msp430f5529"
error #10056: symbol "__TI_int47" redefined: first defined in "./unused_interrupts.obj"; redefined in "./main.obj"
error #10010: errors encountered during linking; "lab_05a_buttonInterrupt.out" not built
<Linking>
gmake: *** [lab_05a_buttonInterrupt.out] Error 1
gmake: Target `all' not remade because of errors.

>> Compilation failure
```

This error example (from the 'F5529) is telling us that the linker cannot fit all the PORT1_VECTOR is defined twice. (The error is related to INT39 on the 'FR5969 device.)

We just created one of these vectors, where is the other one coming from?

Sidebar – Vector Errors

First, how did we recognize this error?

1. It says, "*errors encountered during linking*". This tells us the compilation was fine, but there was a problem in linking.
2. Next, "*symbol "__TI_int47" redefined*". Oops, too many definitions for this symbol. It also tells us that this symbol was found in both `unused_interrupts.c` as well as `main.c`. (OK, it says that the offensive files were `.obj`, but these were directly created from their `.c` counterparts.)
3. Finally, what's with the name, "`__TI_int47`"? Go back and look at the Interrupt Vector Location (sometimes it's also called Interrupt Priority) in the Interrupt Vector table. You can find this in the chapter discussion or the datasheet. Once you've done so, you should see the correlation with the PORT1_VECTOR.

22. Comment out the PORT1_VECTOR from unused_interrupts.c.

```

17 #pragma vector=COMP_B_VECTOR
18 #pragma vector=DMA_VECTOR
19 // #pragma vector=PORT1_VECTOR
20 #pragma vector=PORT2_VECTOR
21 #pragma vector=RTC_VECTOR
22 #pragma vector=SYNCH_VECTOR

```



23. Try building it again

It should work this time... *our fingers are crossed for you.*



24. Launch the debugger.

25. Remove all breakpoints.

View → Breakpoints then click the Remove All button



26. Set a new breakpoint inside your new ISR.

```

62 #pragma vector=PORT1_VECTOR
63 __interrupt void pushbutton_ISR (void)
64 {
65     // Toggle the LED on/off
66     GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
67 }

```



27. Run your code ... once the code is running, push the button to generate an interrupt.

The processor should stop at your ISR (location shown above). Breakpoints like this can make it easier to see that we reached the interrupt. (A good debugging trick.)



28. Resuming once again, at this point inside the ISR should toggle-on the LED.

If it works, call out “Hooray!”



29. Push the button again.

Hmmm... did you get another interrupt? We didn't appear to.

We didn't see the light toggle-off – and we didn't stop at the breakpoint inside the ISR.

Some of you may have already known this was going to happen. If you're still unsure, go back to Step 14 from our worksheet (page 5-41). We discussed it there.

Upgrade Your Interrupt Service Routine (ISR)

If you hadn't already guess what the problem was, since the IFG bit never got cleared, the CPU never realized that new interrupts were being applied.

For grouped interrupts, if we use the appropriate Interrupt Vector (IV) register, we can easily decipher the highest priority interrupt of the group, as well as getting the CPU to clear the IFG bit.

30. Replace the code inside your ISR with the code that uses the P1IV register.

Once again, we have already created the code as part of the worksheet; refer to the Worksheet, Step 14 (page 5-41).

To make life easier, here's a copy of the original template from the worksheet. You may want to cut/paste this code, then tweak it with answers from your worksheet.

```

//*****
// Interrupt Service Routines
//*****
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {

    switch(__even_in_range( ????, 10 )) {
        case 0x00: break;           // None
        case 0x02: break;           // Pin 0
        case 0x04:                   // Pin 1
            ??????????????????????;
            break;
        case 0x06: break;           // Pin 2
        case 0x08: break;           // Pin 3
        case 0x0A: break;           // Pin 4
        case 0x0C: break;           // Pin 5
        case 0x0E: break;           // Pin 6
        case 0x10: break;           // Pin 7
        default: _never_executed();
    }
}

```

Hint: The syntax indentation often gets messed up when pasting code. If/when this occurs, the CCS editor provides a prettying feature (<ctrl>-I).

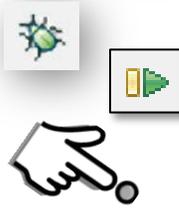
Select the 'ugly' code and press  - 

31. Build the code.



If you correctly inserted the code and replaced all the questions marks, hopefully it built correctly the first time.

32. Launch the debugger. Run. Push the button. Verify the light toggles.



Run the program. Push the button and verify that the interrupt is taken every time you push the button. If the breakpoint in the ISR is still set, you should see the processor stop for each button press (and you'll need to click *Resume*).

You're welcome to explore the code further by single-stepping thru code, using breakpoints, suspending (halting) the processor and exploring the various registers.

(Optional) Lab 5b – Can You Make a Watchdog Blink?

The goal of this lab is to blink the LED. Rather than using a `_delay_cycles()` function, we'll actually use a timer to tell us when to toggle the LED.

In Lab 4 we used the Watchdog timer as a ... well, a watchdog timer. In all other exercises, thus far, we just turned it off with `WDT_A_hold()`.

In this lab exercise, we're going to use it as a standard timer (called 'interval' timer) to generate a periodic interrupt. In the interrupt service routine, we'll toggle the LED.

As we write the ISR code, you may notice that the Watchdog Interval Timer interrupt has a dedicated interrupt vector. (Whereas the GPIO Port interrupt had 8 interrupts that shared one vector.)

Import and Explore the WDT_A Interval Timer Example

1. Import the `wdt_a_ex2_intervalACLK` project from the MSP430 DriverLib examples.

We're going to "cheat" and use the example provided with MSP430ware to get the WDT_A timer up and running.

As we discussed in Chapter 3, there are two ways we can import an example project:

- Use the Project → Import Existing CCS Eclipse Project (as we've done before)
- Utilize the TI Resource Explorer (which is what we'll do again)

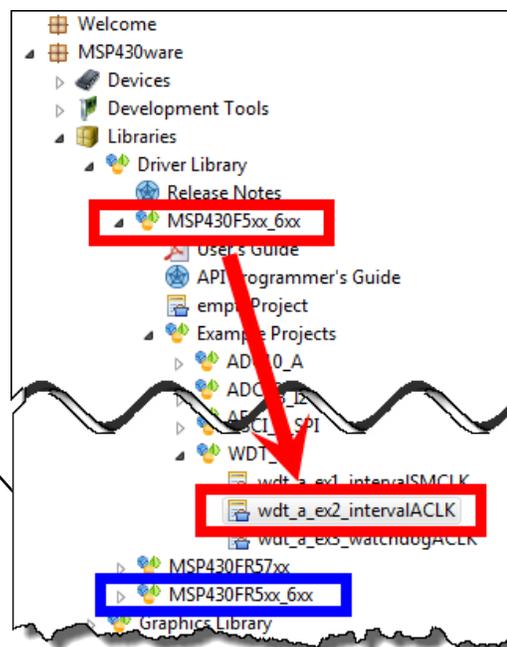
a) Open the TI Resource Explorer window, if it's not already open

View → TI Resource Explorer

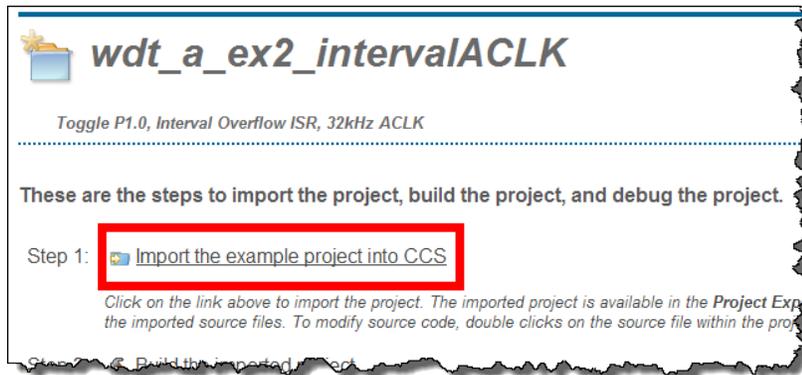
b) Locate the `wdt_a_ex2_intervalACLK` example.

Look for it as shown here under: Example Projects → WDT_A

If you're using the FR5969, follow the same path starting from the `MSP430FR5xx6xx` heading.



c) Click the link to “Import the example project into CCS”.



Once imported you can close the TI Resource Explorer, if you want to get it out of the way.

d) Rename the imported project to: lab_05b_wdtBlink

While not required, this should make it easier to match the project to our lab files later on.

2. Open the lab_05b_wdtBlink.c file. Review the following points:

Notice the DriverLib function that sets up the WDT_A for interval timing.
You can choose which clock to use; we selected ACLK. By the way, what speed is ACLK running at? (This example uses ACLK at the default rate.)
As described, dividing ACLK/8192 gives us an interval of ¼ second.

The WDT_A is a system (SYS) interrupt, so it's IFG and IE bits are in the Special Functions Register. It's always good practice to clear a flag before enabling the interrupt. (Remember, CPU won't be interrupted until we set GIE.)

Along with enabling interrupts globally (GIE=1), this example puts the CPU into low power mode (LPM3).
When the interrupt occurs, the CPU wake up and handles it, then goes back into LPM3. (Low Power modes will be discussed further in a future chapter.)

They got a little bit fancy with the interrupt vector syntax. This code has been designed to work with 3 different compilers:
TI, IAR, and GNU C compiler.

```

main(void)
//Initialize WDT module in timer interval mode,
//with ACLK as source at an interval of 250 ms.
WDT_A_intervalTimerInit(WDT_A_BASE,
                        WDT_A_CLOCKSOURCE_ACLK,
                        WDT_A_CLOCKDIVIDER_8192);

WDT_A_start(WDT_A_BASE);

//Enable Watchdog Interrupt
SFR_clearInterrupt(SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT);
SFR_enableInterrupt(SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT);

//Set P1.0 to output direction
GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );

//Enter LPM3, enable interrupts
__bis_SR_register(LPM3_bits + GIE);
//For debugger
__no_operation();

//Watchdog Timer interrupt service routine
56 #if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
57 #pragma vector = WDT_VECTOR
58 __interrupt
59 #elif defined(__GNUC__)
60 __attribute__((interrupt(WDT_VECTOR)))
61 #endif
62 void WDT_A_ISR(void)
63 {
64     //Toggle P1.0
65     GPIO_toggleOutputOnPin(
66         GPIO_PORT_P1,
67         GPIO_PIN0);
68 }
    
```

These GPIO functions should be familiar by now ...

Since WDT has a dedicated interrupt vector, the code inside the ISR is simple. We do not have to manually clear the IFG bit, or use the IV vector to determine the interrupt source.
MSP430 Workshop - Interrupts

Run the code



3. Build and run the example.

You should see the LED blinking...

Change the LED blink rate

4. Terminate the debug session.

5. Modify the example to blink the LED at 1 second intervals.

Tip: If you want help with selecting and typing function arguments, you can use the autocomplete feature of CCS. Just type part of the test, such as:

```
WDT_A_CLOCKDIVER_
```

and then hit:

```
Control-TAB
```

and a popup box appears providing you with choices – select the one you want. In this case, we suggest you divide by 32K.

The screenshot shows a code editor with the following code:

```
WDT_A_intervalTimerInit( WDT_A_BASE,
    WDT_A_CLOCKDIVER_
    WDT_A_start(WDT_A_BASE);
//Enable Watchdog Interrupt
SFR_clearInterrupt(SFR_BASE,
    WDTIFG);
SFR_enableInterrupt(SFR_BASE,
    WDTIE);
//Set P1.0 to output direction
GPIO_setAsOutputPin(
    GPIO_PORT_P1,
```

An autocomplete popup is visible, listing the following options:

- # WDT_A_CLOCKDIVER_128M
- # WDT_A_CLOCKDIVER_2G
- # WDT_A_CLOCKDIVER_32K
- # WDT_A_CLOCKDIVER_512
- # WDT_A_CLOCKDIVER_512K
- # WDT_A_CLOCKDIVER_64
- # WDT_A_CLOCKDIVER_8192
- # WDT_A_CLOCKDIVER_8192K

The option `# WDT_A_CLOCKDIVER_32K` is highlighted by the mouse cursor.



6. Build and run the example again.

If you want, you can experiment with other clock divider rates to see their affect on the LED.

Appendix

Lab 05 Worksheet (1)

General Interrupt Questions

1. When your program is not in an interrupt service routine, what code is it usually executing? And, what 'name' do we give this code?
main functions while{} loop. We often call this 'background' processing.
2. Why keep ISR's short (i.e. not do a lot of processing in them)?
We don't want to block other interrupts. The other option is nesting interrupts, but this is INEFFICIENT. Do interrupt follow-up processing in while{} loop ... or use TI-RTOS kernel.
3. What causes the MSP430 to exit a Low Power Mode (LPMx)?
Interrupts
4. Why are *interrupts* generally preferred over *polling*?
They are a lot more efficient. Polling ties up the CPU – even worse it consumes power waiting for an event to happen.

Lab 05 Worksheet (2)

Interrupt Flow

5. Name 3 more sources of interrupts?
TIMER A
GPIO
Watchdog Interval Timer
Analog Converter ... and many more
6. What signifies that an interrupt has occurred?
 A flag bit is set
 What's the acronym for these types of 'bits' IFG
7. Write the code to enable a GPIO interrupt on Port 1, pin1 (aka P1.1)?

```
GPIO_setAsInputPinWithPullUpresistor(GPIO_PORT_1, GPIO_PIN1); // setup pin as input
GPIO_clearInterruptFlag(GPIO_PORT_P1, GPIO_PIN1); // clear individual INT
GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN1); // enable individual INT
```

Lab 05 Worksheet (3)

Interrupt Service Routine

8. Write the line of code required to turn on interrupts globally:

`__bis_SR_set(GIE);` // enable global interrupts (GIE)

Where, in our programs, is the most common place we see GIE enabled?
(Hint, you can look back at the slides where we showed how to do this.)

Right before the while{} loop in main().

Interrupt Priorities & Vectors

9. Which interrupt has higher priority: GPIO Port 2 or WDT Interval Timer?

WDT Interval Timer (INT 56 vs GPIO P2 at INT 42)

Let's say you're CPU is in the middle of the GPIO Port 2 ISR, can it be interrupted by a new WDT interval timer interrupt? If so, is there anything you could do to your code in order to allow this to happen?

No, by default, MSP430 interrupts are disabled when running an ISR. To enable this you could setup interrupt nesting (though this isn't recommended)

Lab 05 Worksheet (4)

10. Where do you find the name of an "interrupt vector"?

Interrupt vector table in the datasheet. (It's also defined in the device specific header file (e.g. msp430f5529.h))

11. How do you write the code to set the interrupt vector? (Hint, we've provided a simple ISR to go with the line of code we're asking you to complete.)

```
// Sets ISR address in the vector for Port 1
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void)
{
    // Toggle the LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```

What is wrong with this GPIO port ISR?

GPIO ports are group interrupts, which should read the P1IV register and handle multiple interrupts using a switch/case statement

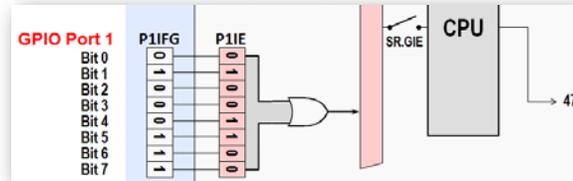
Lab 05 Worksheet (5)

12. How do you pass a value into (or out from) and interrupt service routine (ISR)?

Interrupts cannot pass arguments, we need to use global variables

ISR's for Group Interrupts

As we learned earlier, most MSP430 interrupts are grouped. For example, the GPIO port interrupts are all grouped together.



13. For dedicated interrupts (such as WDT interval timer) the CPU clears the IFG flag when responding to the interrupt. How does an IFG bit get cleared for group interrupts?

Either manually; or when you read the IV register (such as P1IV).

Lab 05 Worksheet (6)

14. Creating ISR's for grouped interrupts is as easy as following a 'template'. The following code represents a grouped ISR template. Fill in the blanks required for the CPU to toggle the LED (P1.0) in response to a GPIO pushbutton interrupt (P1.1).

```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {
    switch(__even_in_range(         P1IV        , 10 )) {
        case 0x00: break;           // None
        case 0x02: break;           // Pin 0
        case 0x04: //break;         // Pin 1
                                GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
                                break;
        case 0x06: break;           // Pin 2
        case 0x08: break;           // Pin 3
        case 0x0A: break;           // Pin 4
        case 0x0C: break;           // Pin 5
        case 0x0E: break;           // Pin 6
        case 0x10: break;           // Pin 7
        default:  _never_executed(); }
}
```

Introduction

Timers are often thought of as the heartbeat of an embedded system.

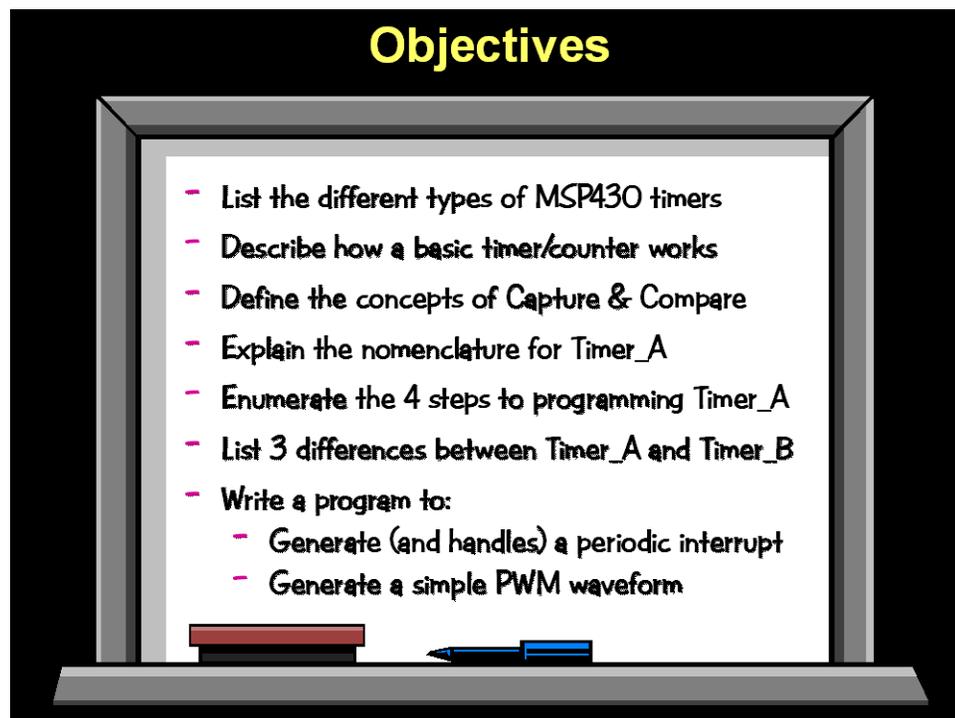
Whether you need a periodic wake-up call, a one-time delay, or need a means to verify that the system is running without failure, Timers are the solution.

This chapter begins with a brief summary of the MSP430 Timers. Most of the chapter, though, is spent digging into the details of the MSP430's TIMER_A module. Not only does it provide rudimentary counting/timing features, but provides sophisticated capture and compare features that allow a variety of complex waveforms – or interrupts – to be generated. In fact, this timer can even generate PWM (pulse width modulation) signals.

Along the way, we examine the MSP430ware DriverLib code required to setup and utilize TIMER_A.

As the chapter nears conclusion, there's a brief summary of the differences between TIMER_A and TIMER_B. Bottom line, if you know how to use TIMER_A, then you can use TIMER_B; but, there are a couple of extra features that TIMER_B provides.

Learning Objectives



Chapter Topics

Timers	6-1
<i>Prerequisites and Tools</i>	<i>6-2</i>
<i>Overview of MSP430 Timers</i>	<i>6-3</i>
TIMER_A/B Nomenclature.....	6-4
Timer Summary.....	6-5
<i>Timer Basics: How Timers Work.....</i>	<i>6-6</i>
Counter.....	6-6
Frequency, Time-Period, Resolution	6-7
Capture.....	6-8
Compare.....	6-9
<i>Timer Details: Configuring TIMER_A</i>	<i>6-12</i>
1. Counter: TIMER_A_configure...().....	6-13
Timer Counting Modes.....	6-14
Summary of Timer Setup Code – Part 1	6-18
2a. Capture: TIMER_A_initCapture().....	6-19
2b. Compare: TIMER_A_initCompare().....	6-21
Summary of Timer Setup Code – Part 2.....	6-23
Output Modes.....	6-24
PWM anyone?.....	6-29
3. Clear Interrupt Flags and TIMER_A_startTimer().....	6-30
4. Interrupt Code (Vector & ISR).....	6-31
<i>TIMER_A DriverLib Summary.....</i>	<i>6-32</i>
<i>Differences between Timer's A and B.....</i>	<i>6-33</i>
<i>Lab Exercise</i>	<i>6-35</i>

Prerequisites and Tools

To get full entitlement from this chapter, we expect that you are already familiar with MSP430ware's DriverLib as well as MSP430 clocking and interrupts. The “extra” piece of hardware required for this chapter is a single jumper wire.

Prerequisites & Tools	
◆ Skills	Chapter
<ul style="list-style-type: none"> • Creating a CCS Project for MSP430 Launchpad(s) • Basic knowledge of: <ul style="list-style-type: none"> • C language • Using a C libraries and header files (MSP430ware DriverLib) • Setting up MSP430 clocks • Using interrupts (setup and ISR's) 	<ul style="list-style-type: none"> (Ch 2) (Ch 3) (Ch 4) (Ch 5)
◆ Hardware	
<ul style="list-style-type: none"> • Windows (XP, 7, 8) PC with available USB port • MSP430F5529 Launchpad (with included USB micro cable) • One (1) jumper wire (female to female) 	
◆ Software	
<ul style="list-style-type: none"> • CCSv5.5 • MSP430ware (v1.60.02.09) 	

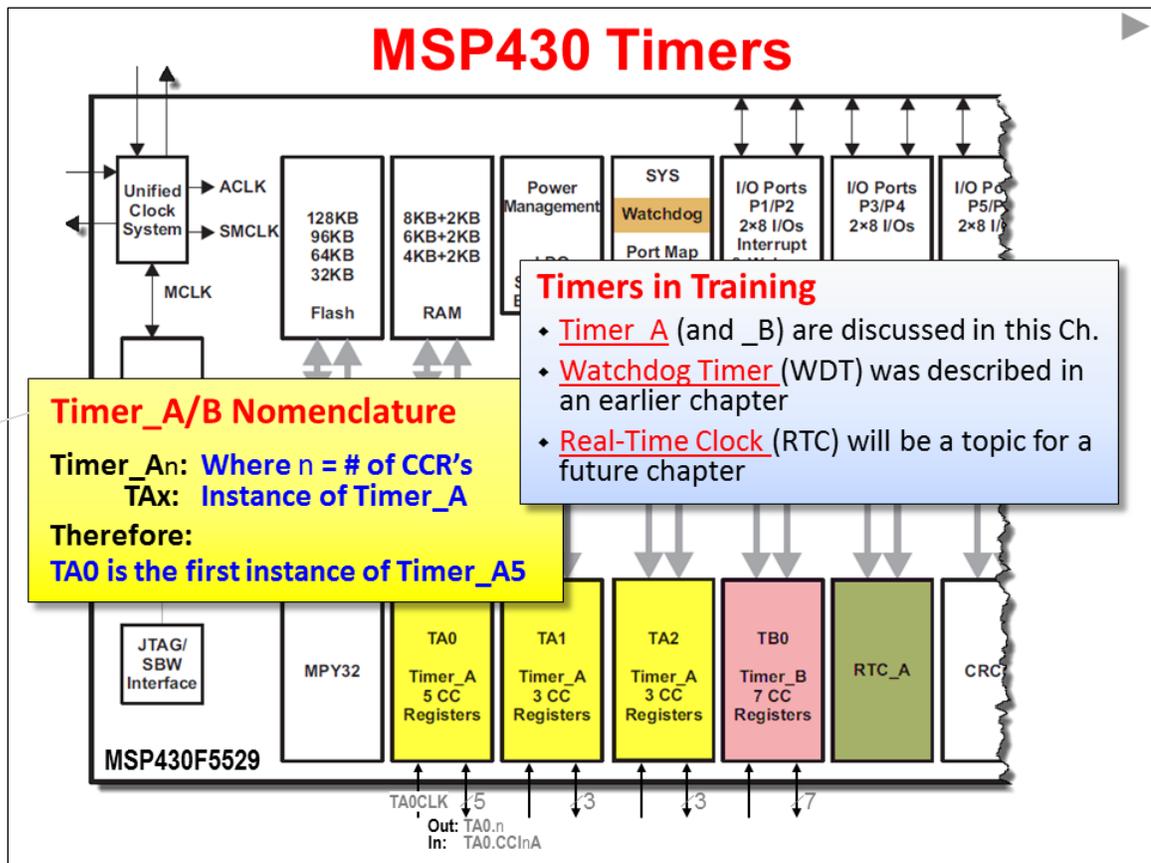
Overview of MSP430 Timers

The MSP430F5529 timers are highlighted in the following block diagram.

- **Yellow** marks the three instances of the TIMER_A module.
- **Pink** was used for TIMER_B.
- **Dark brown** highlights the real-time clock (RTC_A).
- **Light brown** differentiates the Watchdog timer inside the SYS block

The “Timers in Training” callout box describes where the various timers are discussed in this workshop. Timers A and B are covered in this chapter. We have already covered the Watchdog timer in a previous chapter.

The RTC module will be discussed in a future chapter. A brief description of the RTC tells us that it’s a very low-power clock; has built-in calendar functions; and often includes “alarms” that can interrupt the CPU. It is frequently used for keeping a time-base while the CPU is in low-power mode.



Nomenclature is discussed on the next page

TIMER_A/B Nomenclature

The nomenclature of the TIMER_A and _B peripherals is a little unusual. First of all, you may have already noticed that the MSP430 team often adds one of two suffixes to their peripheral names to indicate when features have been added (or modified).

- Some peripherals, such as the Watchdog Timer go from “WDT” to “WDT+”. That is, they add a “+” to indicate the peripheral has been updated (usually with additional features).
- Other peripherals are enumerated with letters. For example, three sophisticated MSP430 timers have been introduced: TIMER_A, TIMER_B, and TIMER_D. *(What happened to _C? Even I don't know that. <ed>)*

The use of a suffix is the generic naming convention found on the MSP430. With the timers, though, there are a couple more naming variations to be discussed.

As we will cover in great detail during this chapter, these timers contain one or more Capture and Compare Registers (CCR); these are useful for creating sophisticated timings, interrupts and waveforms. The more CCR registers a timer contains, the more independent waveforms that can be generated. To this end, the documentation often includes the number of CCR registers when listing the name of the timer. For example, if TIMER_A on a given device has 5 CCR registers, they often name it:

Timer_A5

But wait, that's not all. What happens when a device, such as the 'F5529 has more than one instance of TIMER_A? Each of these instances needs to be enumerated as well. This is done by appending the instance number after the word “Timer”, as in Timer0.

To summarize, here's the long (and short) names for each of the 'F5529 TIMER_A modules:

Instance	Long Name	Short Name
0	Timer0_A5	TA0
1	Timer1_A3	TA1
2	Timer2_A3	TA2

Timer Summary

The 'F5529 contains most of the different types of timers found across the MSP430 family; in fact, the only type of timer not present on this device is the high-resolution TIMER_D.

The following summary provides a snapshot of what timers are found on various MSP430 devices. You'll find our 'F5529 and 'FR5969 devices in the last two columns of the table.

A one-line summary of each type of timer is listed below the table.

MSP430 Timers						
	L092	G2553	FG479	F5172	F5529	FR5969
Timer_A	2 x A3	2 x A3	1 x A3	1 x A3	1 x A5 2 x A3	2 x A3 2 x A2*
Timer_B			1 x B3		1 x B7	1 x B7
Timer_D				2 x D3		
Real-Time Clock			Basic Timer1 with RTC		RTC_A	RTC_B
Watchdog	WDT_A	WDT+	WDT+	WDT_A	WDT_A	WDT_A

Timer_A: 'A3' means it has 3 Capture/Compare Registers (used to generate signals & ints)
Timer_B: Same as A, but improves PWM
Timer_D: Same as B, adding hi-res timing
WDT+: Watchdog or Interval Modes; PSW Protected; Can stop; Select Clk; Clk fail-safe
WDT_A: Same as WDT+, but with 8 timer intervals rather than 4
BT1/RTC: Basic timer has 2x8-bit counters (can use as 1x16-bits) with calendar functions
RTC_A: 32-bit counter with a calendar, flexible programmable alarm, and calibration
RTC_B: Same as RTC_A, but adds switchable battery backup in case main-power fails

Timer Basics: How Timers Work

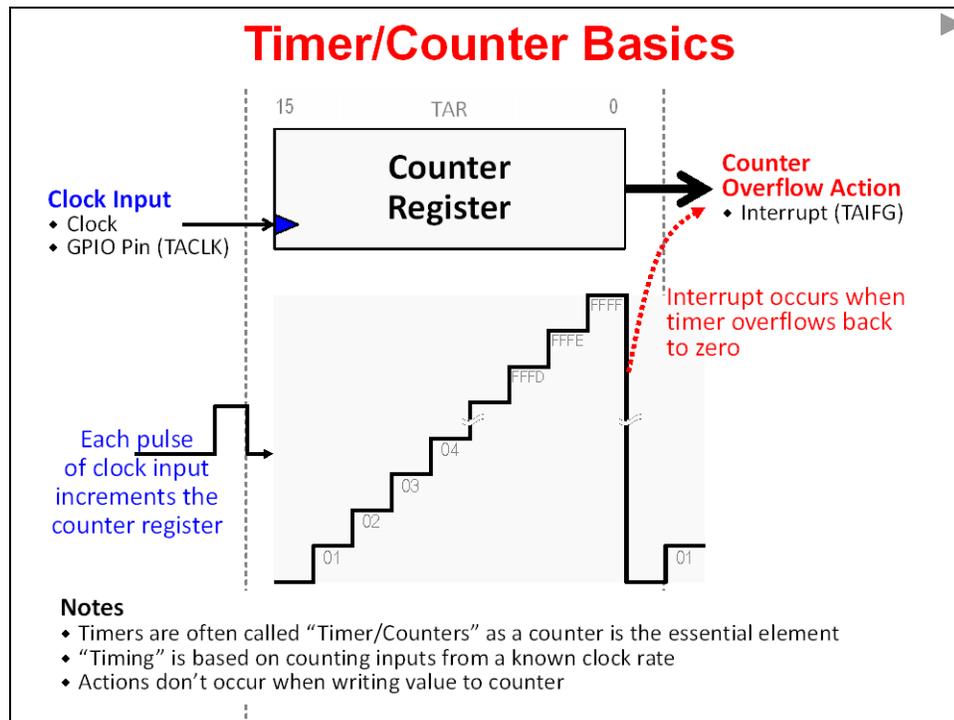
Before we discuss the details of TIMER_A, let's begin with a quick overview describing how timers work. Specifically, we will start by describing how a timer is constructed using a **Counter**. Next, we'll investigate the **Capture** and **Compare** capabilities found in many timers.

Counter

A **counter** is the fundamental hardware element found inside a timer.

The other essential element is a **clock** input. The counter is incremented each time a clock pulse is applied to its clock input. Therefore, a 16-bit timer will count from zero (0x0000) up to 64K (0xFFFF).

When the counter reaches its maximum value, it overflows – that is, it returns to zero and starts counting upward again. Most timer peripherals can generate an interrupt when this overflow event occurs; on TIMER_A, the interrupt flag bit for this event is called TAIFG (TIMER_A Interrupt Flag).



The clock input signal for TIMER_A (named TACLK) can be one of the internal MSP430 clocks or a signal coming from a GPIO pin.

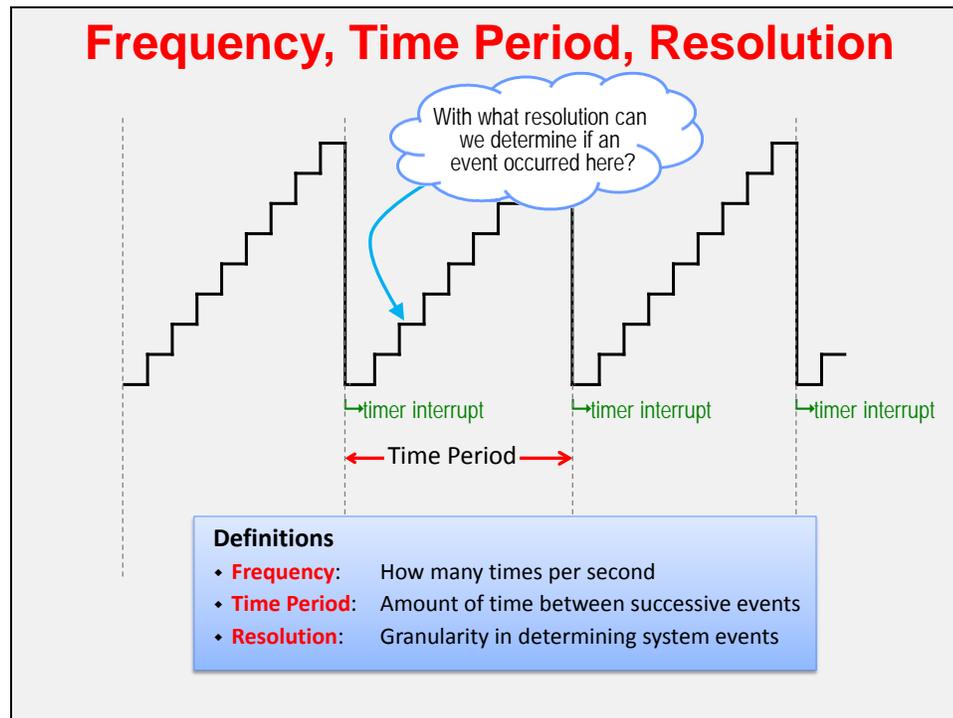
Many engineers call these peripherals “Timer/Counters” as they provide both sets of functionality. They can generate interrupts or waveforms at a specific time-base – or could be used to count external events occurring in your system.

One final note about the MSP430 timers: they do not generate interrupts (or other actions) when you write to the counter register. For example, writing “0” to the counter won’t generate the TAIFG interrupt.

Frequency, Time-Period, Resolution

The Timer's ability to create a consistent, periodic interrupt is quite valuable to system designers. *Frequency* and *Time Period* are two terms that are often used to describe the rate of interrupts.

- How many times per second that a timer creates an interrupt defines its **Frequency**.
- Conversely, the amount of time in-between interrupt events is defined as the **Time Period**.



If a timer only consisted of a single counter, its *resolution* would be limited to the size of the counter.

If some event were to happen in a system – say, a user pushed a button – we could only ascertain if that event occurred within a time period. In other words, we can only determine if it happened between two interrupts.

Looking at the above diagram, we can see that there is “more data” available – that is, if we were to read the actual counter value when the event occurred. Actually, we can do this by setting up a GPIO interrupt; then, having the ISR read the value from the counter register. In this case the resolution would be better, but it is still limited by:

- It takes more hardware (an extra GPIO pin is needed)
- The CPU has to execute code – this consumes power and processing cycles
- The resolution is less deterministic because it's based upon the latency of the interrupt response; in other words, how fast can the CPU get to reading the counter ... and how consistent can this be each time it occurs

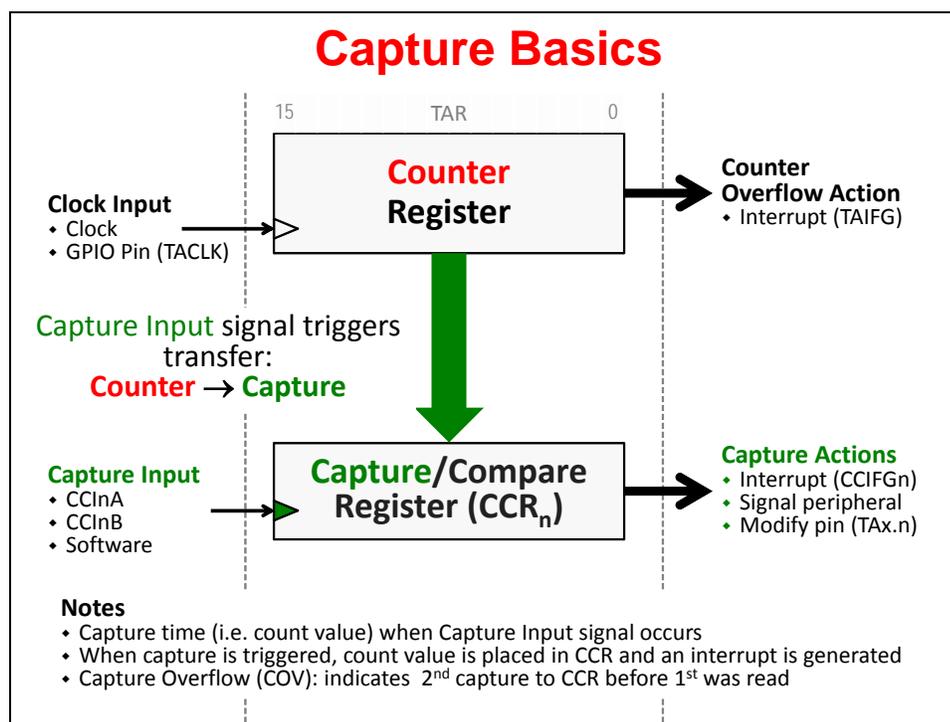
There is a better way to implement this in your system ... turn the page and let's examine the timer's **Capture** feature.

Capture

The **Capture** feature does just that. When a capture input signal occurs, a snapshot of the Counter Register is *captured*; that is, it is copied into a capture register (CCR for Capture and Compare Register). This is ideal since it solves the problems discussed on the previous page; we get the timer counter value captured with no latency and very, very little power used (the CPU isn't even needed, so it can even remain in low-power mode).

The diagram below builds upon our earlier description of the timer. The top part of the diagram is the same; you should see the Counter Register flanked by the Clock Input to the left and TAIFG action to the right.

The bottom portion of the slide is new. In this case, when a Capture Input signal occurs, the value from the Counter Register is copied to a capture register (i.e. CCR).



A few notes about the *capture* feature:

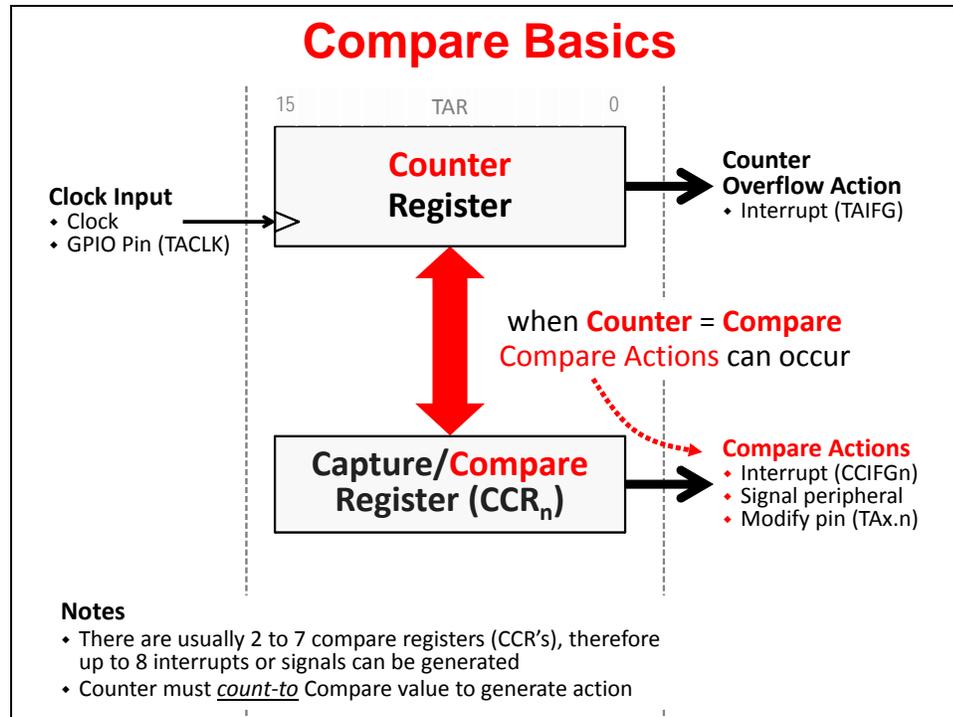
- As we discussed earlier, the MSP430 timers (TIMER_A, TIMER_B, and TIMER_D) have multiple CCR registers; check your datasheet to determine how many are available per timer peripheral. Each CCR, though, has its own capture input signal.
- The Capture Input signal can be connected to a couple of different signals (CCInA, CCInB) or triggered in software
- The Capture Input hardware signals (CCInA, CCInB) are connected differently for each CCR register and device. You need to reference the datasheet to verify what options are available on your specific device.
- When a capture occurs, the CCR can trigger further actions. This “action” signal can generate an interrupt to the CPU, trigger another peripheral, and/or modify the value of a pin.

As we just discussed, the Capture feature provides a deterministic method of capturing the count value when triggered. While handy, there is another important requirement for timers...

Compare

A key feature for timers is the ability to create a consistent, periodic interrupts.

As we know, TIMER_A can do this, but the timer's frequency (i.e. time period) is limited to dividing the input clock by 2^{16} . So, while the timer may be *consistent*, but not very flexible. Thankfully, the **Compare** feature of TIMER_A (TIMER_B & TIMER_D) solves this problem.



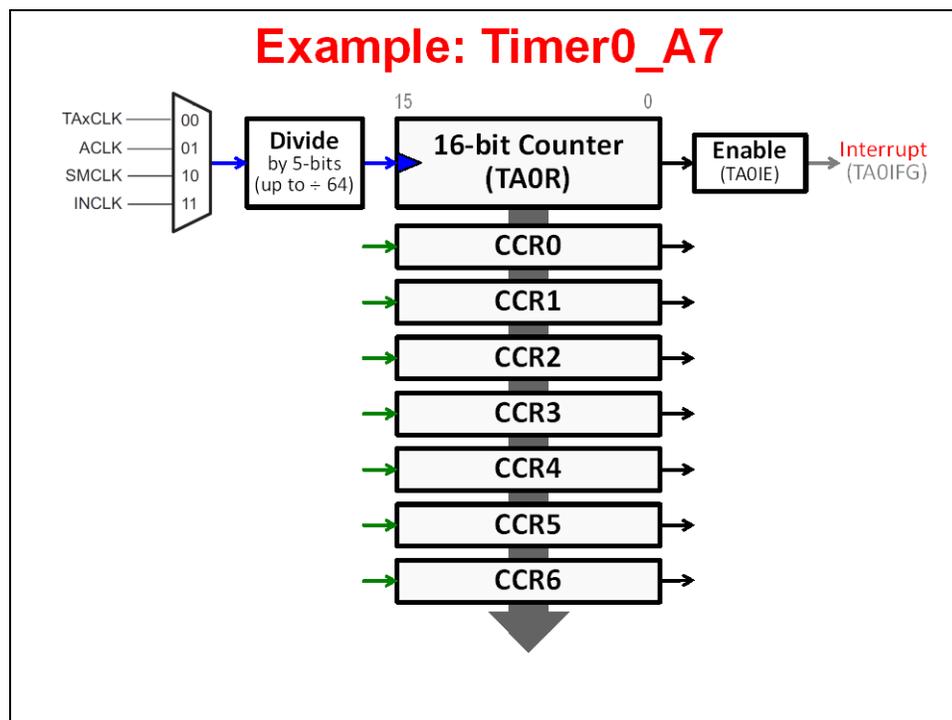
Once again, the top portion of this diagram remains the same (Clock Input + Counter Register).

The bottom portion of the diagram differs from the previous diagrams. In this case, rather than using the CCR register for capture, it's used as a *compare* register. In this mode, whenever a match between the Counter and Compare occurs, a compare action is triggered. The compare actions include generating an interrupt, signaling another peripheral (e.g. triggering an ADC conversion), or changing the state of an external pin.

The "modify pin" action is a very powerful capability. Using the timer's *compare* feature, we can create sophisticated PWM waveforms. (Don't worry, there's more about this later in the chapter.)

Timer Summary – showing multiple CCR's

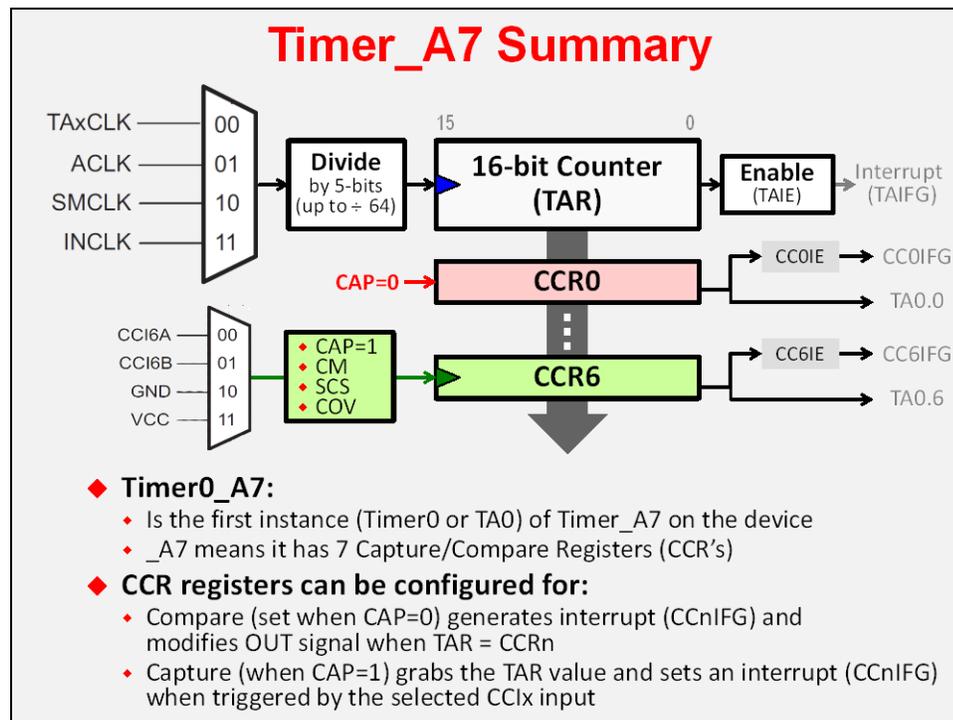
The following example of a Timer0_A7 provides us a way to summarize the timer's hardware.



Remember:

- Timer0 means it's the first instance of Timer_A on the device.
- _A7 means that it's a Timer_A device and has 7 capture/compare registers (CCR's)
- The clock input, in this example, can be driven by a TACLK signal/pin, ACLK, SMCLK or another internal clock called INCLK.
- The clock input can be further divided down by a 5-bit scalar.
- The TA0IE interrupt enable can be used to allow (or prevent) an interrupt (TA0IFG) from reaching the CPU whenever the counter (TA0R) rolls over.

This next diagram allows us to look more closely at the Capture and Compare functions.



Every CCR register has its own control register. Notice above, that the “CAP” bit configures whether the CCR will be used in capture (CAP=1) or compare mode (CAP=0).

You can also see that each CCR has an interrupt flag, enable, and output signal associated with it. The output signal can be routed to a pin or a number of other internal peripherals.

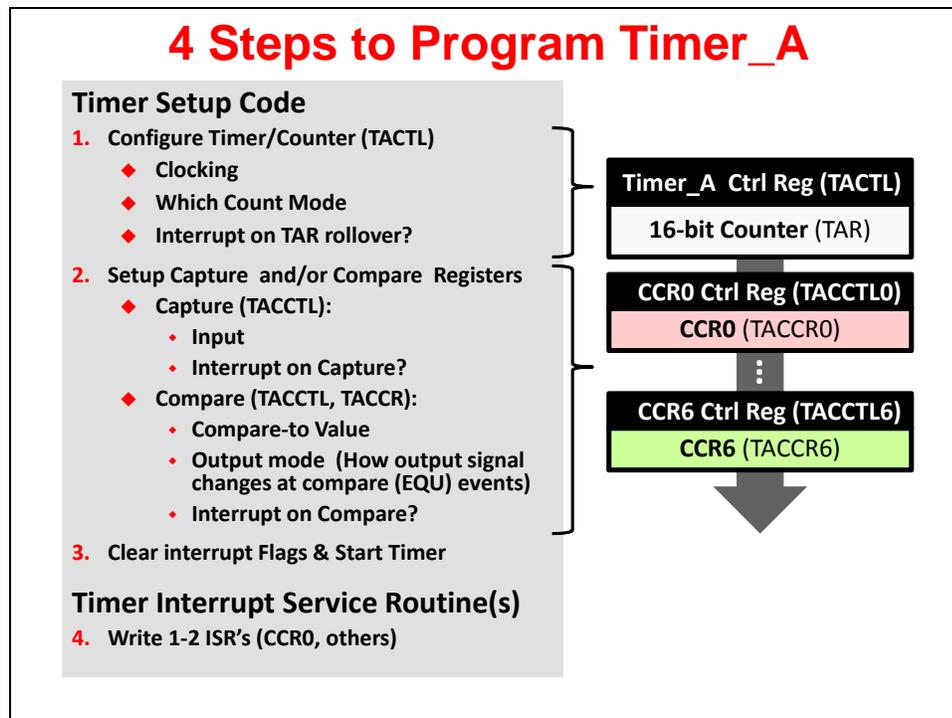
As we go through the rest of this chapter, we’ll examine further details of the CCR registers as well as the various “actions” that the timer generates.

In the next section, we’ll begin examining how to configure the timer using the MSP430ware DriverLib API.

Timer Details: Configuring TIMER_A

There are four steps required to get Timer_A working in your system:

1. Configure the *main Timer/Counter* by programming the TACTL control register.
2. Setup each CCR that is needed for your application. We will examine this step from both the *Capture* and *Compare* perspective.
3. Next, you need to start the timer. (We also listed clearing the timer IFG bits, which is normally done right before starting the timer.)



4. Finally, if your timer is generating interrupts, you need to have an associated ISR for each one. (While interrupts were covered in the last chapter, we briefly summarize this again in context of the Timer_A.)

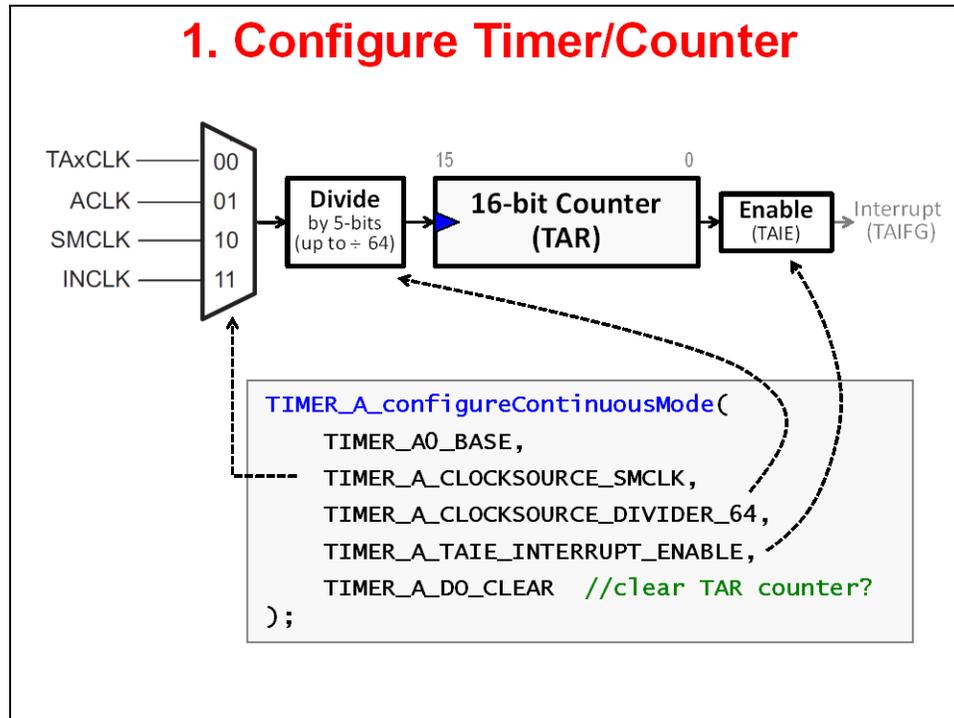
We will intermix how to write code for the timer with further examination of the timer's features.

1. Counter: TIMER_A_configure...()

The first step to using TIMER_A is to program the main timer/counter control register. The MSP430ware Driver Library provides 3 different functions for setting up the main part of the timer:

```
TIMER_A_configureContinuousMode()
TIMER_A_configureUpMode()
TIMER_A_configureUpDownMode()
```

We will address the different modes on the next page. For now, let's choose 'continuous' mode and see how we can configure the timer using the DriverLib function.



From the diagram, we can see that 3 different hardware choices need to be made for our timer configuration; the arrows demonstrate how the function parameters relate to these choices. Let's look at each parameter one-by-one:

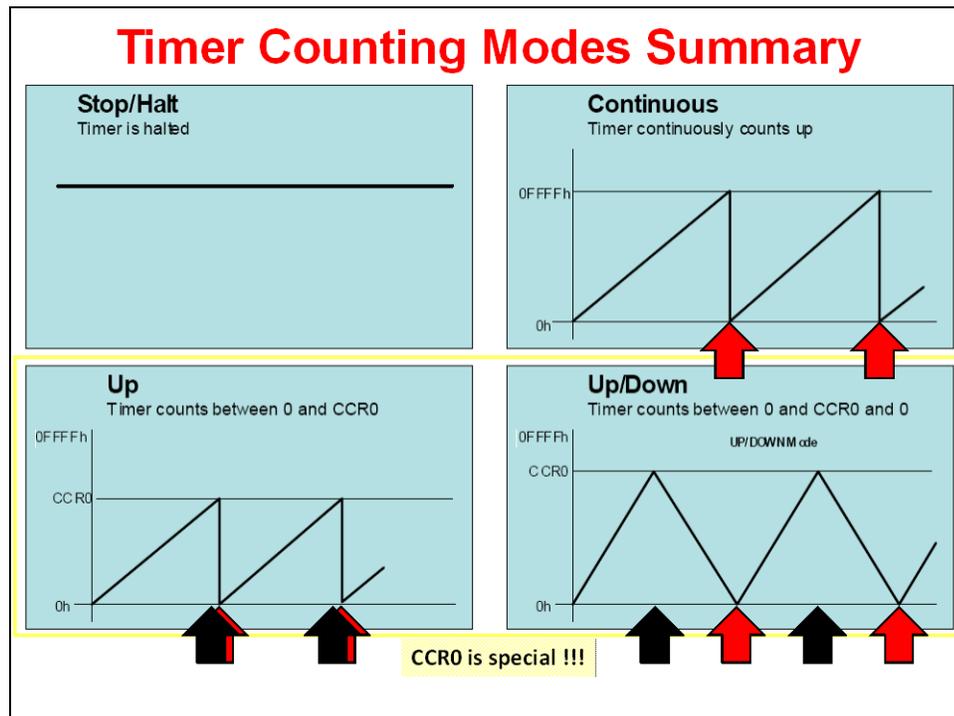
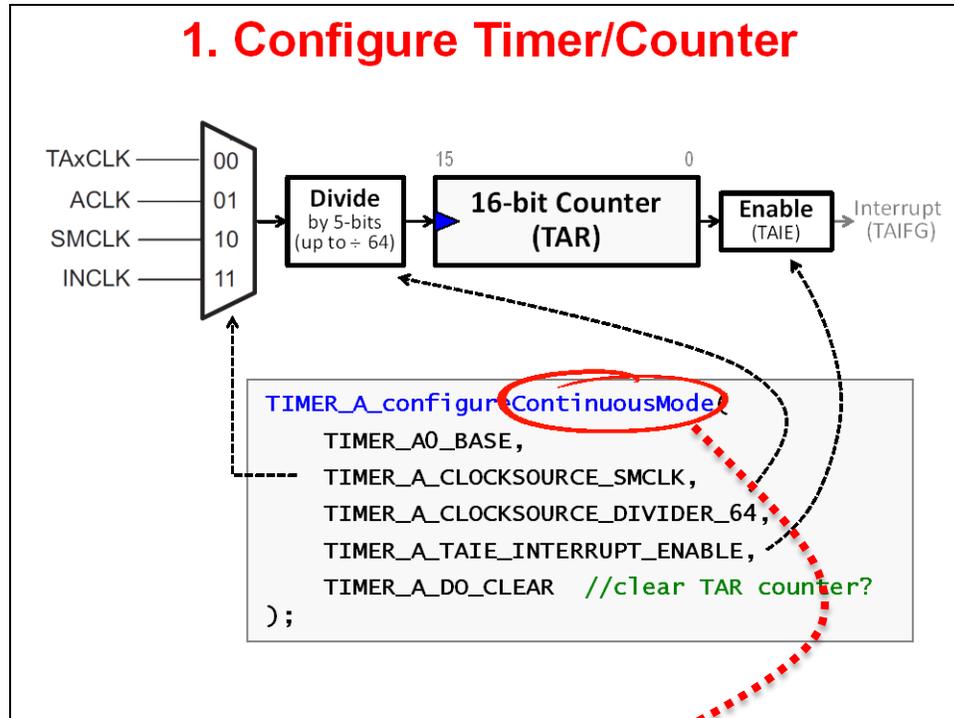
- The first parameter chooses which Timer_A instance you want to program. In our example, we have chosen to program TA0 (i.e. Timer0_A). Conveniently, the DriverLib documentation provides enumerations for all the supported choices. *(This is the same for all DriverLib parameters, so we won't keep repeating this statement. But, this is very handy to know!)*
- The 2nd parameter lets you choose which clock source you want to use. We chose SMCLK.
- The next parameter picks one of the provided clock pre-scale values. The h/w lets you choose from one of 20 different values; we picked ÷ 64.
- Parameter four lets us choose whether to interrupt the processor when the counter (TA0R) rolls over to zero. This parameter ends up setting the TA0IE bit.
- Finally, do you want to have the timer counter register (TA0R) reset when the other parameters are configured?

Remember...

TAR: Timer_A count Register
TA0R: Name for count register when referring to instance "0" (i.e. Timer0_A)

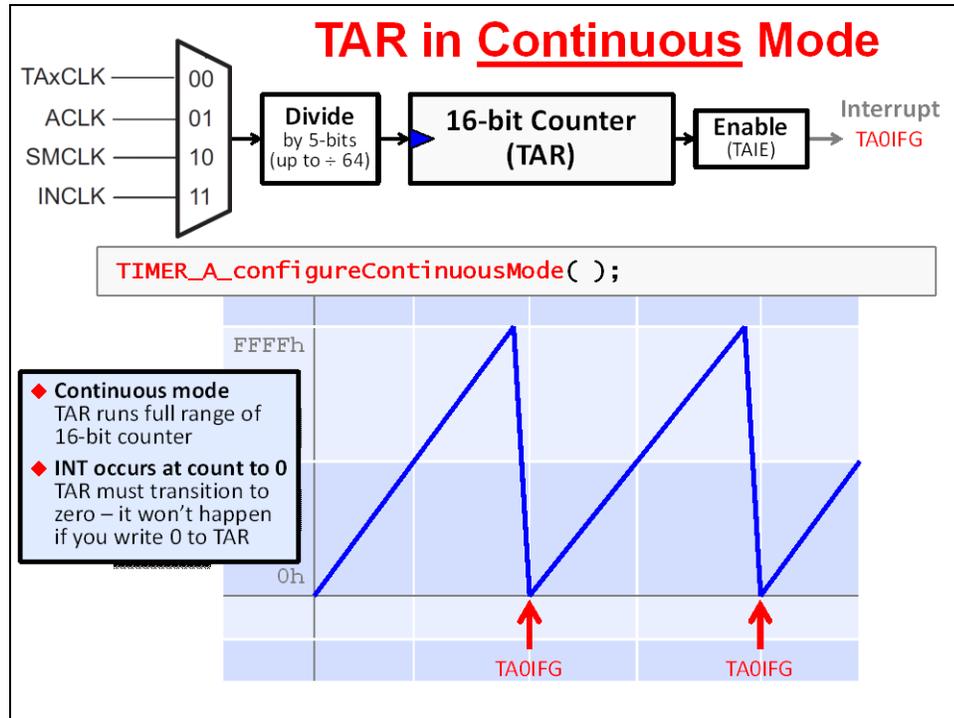
Timer Counting Modes

There are three different ways that the timer counter (TAR) can be incremented. These correlate to the three configuration functions listed on the previous page. This page provides a single-slide summary of the different modes – but we'll examine each one over the following three pages.



Continuous Mode

Thus far we have described the timer's counter operating in the *Continuous* mode; in fact, this was the configuration example we just discussed.



The different counting modes describe how the timer counter register (TAR) is incremented or decremented. For example, in *Continuous* mode, the timer counts from 0x0 up to 0xFFF and then rolls back to 0x0, where it begins counting up again. (This is shown in the diagram above.)

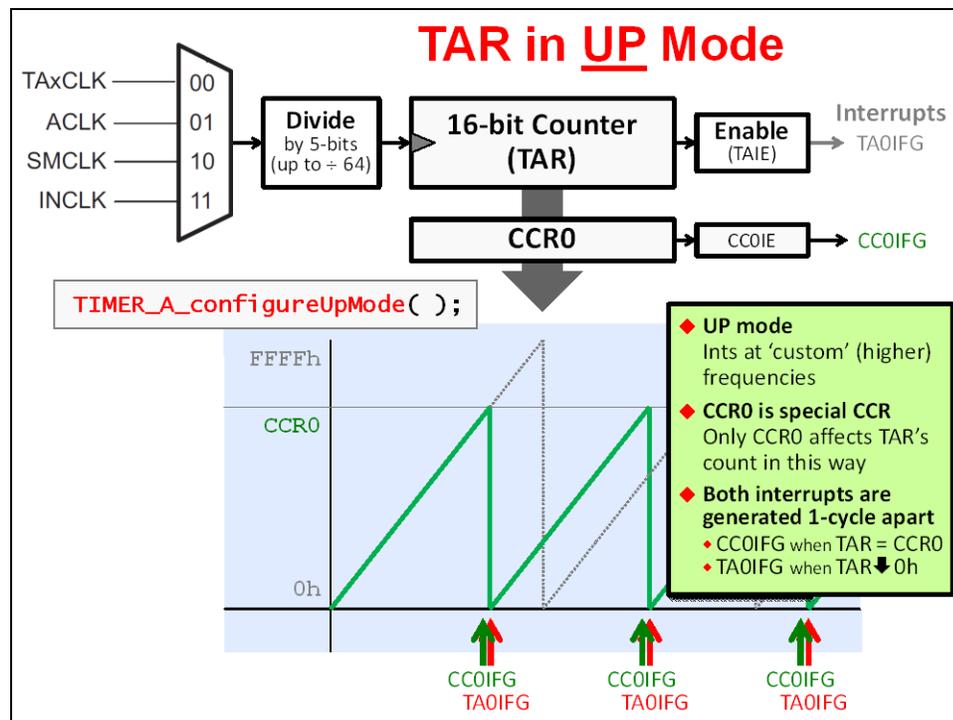
As you can see, every time the counter rolls back to zero, the TAIFG bit gets set; which, if enabled, interrupts the processor every 2^{16} input clocks. (Since our previous example was for Timer0_A, the diagram shows TA0IFG getting set.)

Up Mode

The **Up** counting mode differs from the *Continuous* mode by resetting back to zero whenever the counter matches CCR0 (Capture and Compare Register 0).

You can see the different waveforms compared on the slide below. The green waveform counts **Up** to the value found in CCR0, and then resets back to zero.

On the other hand, the grey dotted waveform shows how, when in *Continuous* mode, the counter goes past CCR0 and all the way to 0xFFFF.



In *Up* mode, since we are using the CCR0 register, the timer can actually generate two interrupts:

- CC0IFG (for Timer0_A, this bit is actually called TA0CC0IFG)
- TAIFG (for Timer0_A, this bit is called TA0IFG)

You're not seeing a color misprint; the two interrupts do not happen at the exact same time, but rather 1 cycle apart. The CC0IFG occurs when there is a compare match, while the TA0IFG interrupt occurs once the counter goes back to zero.

If you compare these two *Up* mode interrupts to the one generated in the *Continuous* mode, you'll see they occur at a more rapid frequency. This is a big advantage of the *Up* mode; your frequency is not limited to 2^{16} counts, but rather can be anywhere within the 16-bit counter's range. (The downside is that you also have to configure the CCR0 registers.)

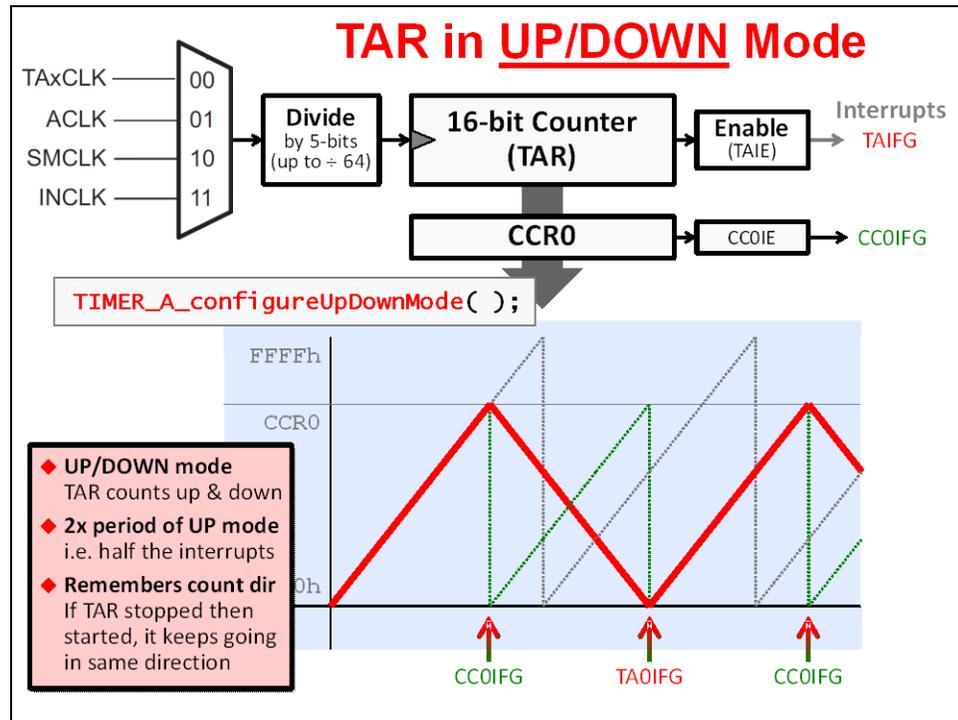
Note: The CCR0 (Capture and Control Register 0) is special. That is, it is special in comparison to the other CCR registers. It is only CCR0 that can be used to define the upper limit of the counter in Up (or UpDown) count mode.

The other special feature of CCR0 is that it provides a dedicated interrupt (CC0IFG). In other words, there is an Interrupt Vector location dedicated to CC0IFG. All the other Timer_A interrupts share a common vector location (i.e. they make up a grouped interrupt).

Up/Down Mode

The **UpDown** count mode is similar to *Up* in that the counter only reaches the value in CCR0 before changing. In this case, though, it actually changes direction and starts counting down rather than resetting immediately back to zero.

Not only does this double the time period (i.e. half the timer's frequency), but it also spreads out the two interrupts. Notice how CCOIFG occurs at the peak of the waveform, while TAOIFG occurs at the base of the waveform.



In our diagram we show all three counter mode waveforms. The **UpDown** mode is shown in red; **Up** is shown in green; and the **Continuous** mode is shown in grey.

Which Count Mode Should I Use?

When using TIMER_A (or TIMER_B), you have a choice as to which counter mode to use. Here are some things to keep in mind.

- Using **Continuous** mode doesn't "tie up" your CCR0 register. It also means you don't have to program the CCR0 register.
- **Up** mode allows you better control of the timer's frequency – that is, you can now control the time period for when the counter resets back to zero.
- On the other hand, the **UpDown** mode not only lets you control the frequency better, but it also allows for lower frequencies – since it effectively halves the frequency of the **Up** mode.
- Two more considerations of **UpDown** mode are:
 - The two interrupts are spaced at ½ the time period from each other.
 - When using multiple CCR registers, you can get two compare interrupts per cycle. (We'll see more on this later.)

Summary of Timer Setup Code – Part 1

Let's summarize Part 1 of the timer setup code – which configures the timer's count options. First of all, as you can see below, we chose to place our timer setup code into its own function. Obviously, this is not a requirement, but it's how we wanted to organize our code examples.

Timer Code Example (Part 1)

```
#include <driverlib.h>

void main(void) {
    // Setup/Hold Watchdog Timer (WDT+ or WDT_A)
    initWatchdog();

    // Configure GPIO ports/pins
    initGPIO();

    // Setup Clocking: ACLK, SMCLK, MCLK (BCS+, UCS, or CS)
    initClocks();

    //-----
    // Then, configure any other required peripherals and GPIO
    initTimers();

    __bis_SR_register( GIE );

    while(1) {
        ...
    }
}
```

Our earlier example for the Timer/Counter setup code demonstrated using the *Continuous* mode. The following example shows using the *Up* mode. Here's a quick comparison between the two functions – notice that the *Up* mode requires two additional parameters.

Parameter	ContinuousMode Function	UpMode Function
Which Timer?	TIMER_A0_BASE	
Clock Source	TIMER_A_CLOCKSOURCE_SMCLK	
Clock Pre-scaler	TIMER_A_CLOCKSOURCE_DIVIDER_xx	
Timer Period	Not applicable	Used to set the CCR0 value
Enable the TAIE interrupt?	TIMER_A_TAIE_INTERRUPT_XXXXXX	
Enable the CCR0 interrupt?	Not applicable	Used to set TAOC0IFG
Clear the counter (TAR) ?	TIMER_A_DO_CLEAR	

Timer Code Example (Part 1)



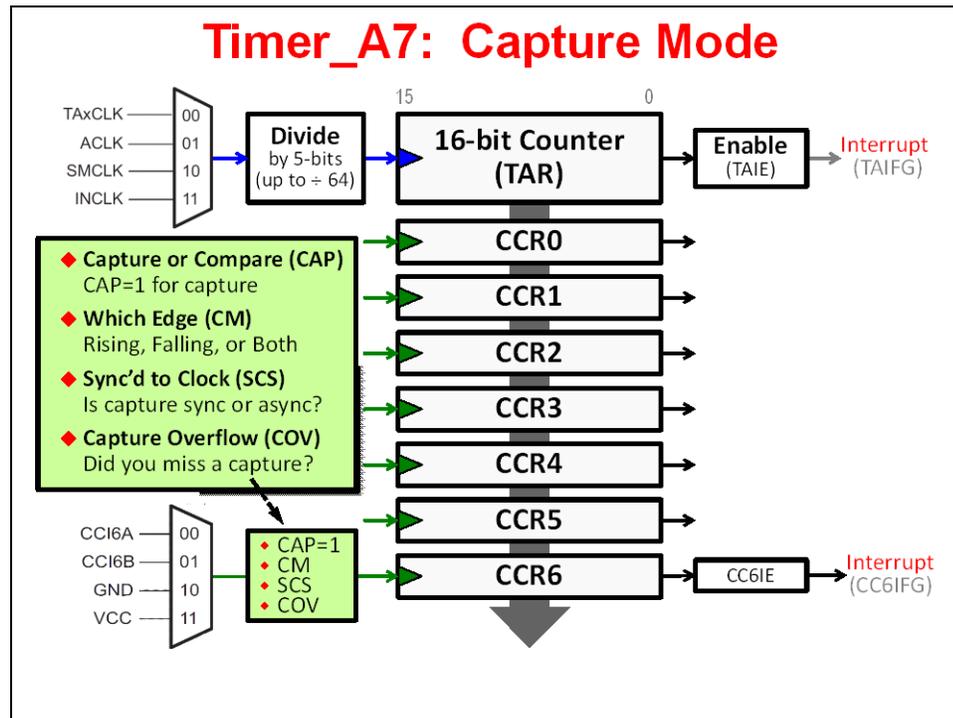
```
#include <driverlib.h>

void initTimerA0(void) {
    // Setup TimerA0 in Up mode
    TIMER_A_configureUpMode( TIMER_A0_BASE,
        TIMER_A_CLOCKSOURCE_SMCLK,
        TIMER_A_CLOCKSOURCE_DIVIDER_1,
        TIMER_PERIOD,
        TIMER_A_TAIE_INTERRUPT_ENABLE,
        TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE,
        TIMER_A_DO_CLEAR );
}
```

2a. Capture: TIMER_A_initCapture()

Before we try writing the code to setup a CCR register for *Capture*, let's first examine the timer's hardware options.

- Most importantly, when wanting to use the *Capture* features, you need to set CAP = 1.
- The CM bit indicates which clock edge to use for the capture input signal.
- Do you want the capture input signal sync'd with the clock input? If so, that's what SCS is for.
- While you don't configure COV, this bit indicates if a capture overflow occurred. In other words, did a 2nd capture occur before you read the captured value from the CCR register?
- Finally, you can select what hardware signal you want to have "trigger" the capture.



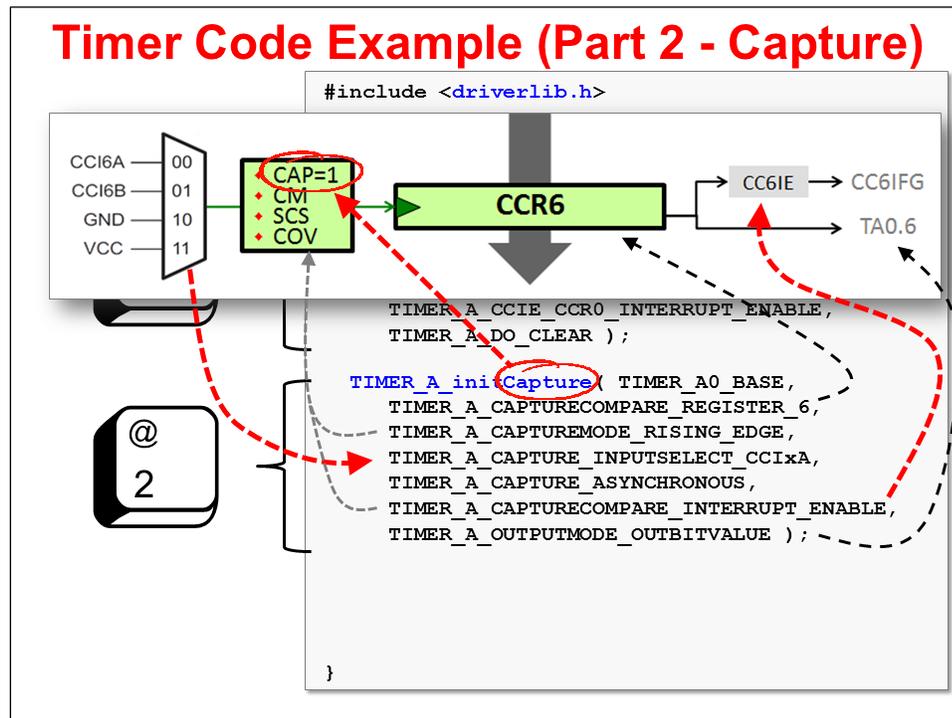
Hint: Each CCR can be configured independently. The flip side to this is that you must configure each one that you want to use; this might involve calling the 'capture' and/or 'compare' configuration functions multiple times.

Use one for capture and the rest for compare. Or, use all for capture. You get to decide how they are used.

Warning: If you are using *Up* or *UpDown* count modes, you should not configure CCR0. Just remember that the `TIMER_A_configureUpMode()` and `TIMER_A_configureUpDownMode()` configuration functions handle this for you.

Capture Code Example

With the Capture mode details in mind, let's examine the code.



To configure a CCR register for Capture mode, use the `TIMER_A_initCapture()` function. Thankfully, when using DriverLib the code is pretty easy to read (and maintain). Hopefully between the diagram and the following table, you can make sense of the parameters.

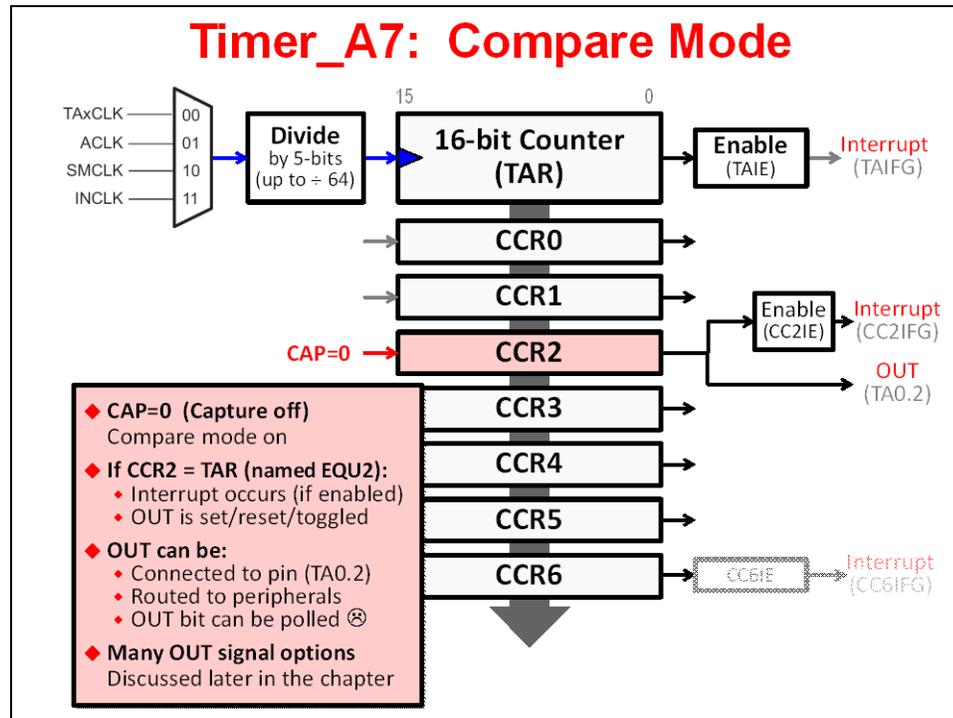
Example's Parameter Value	What is Parameter For?	Value
<code>TIMER_A0_BASE</code>	Which timer are you using?	TA0
<code>TIMER_A_CAPTURECOMPARE_REGISTER_6</code>	Which CCR is being configured?	CCR6
<code>TIMER_A_CAPTUREMODE_RISING_EDGE</code>	Which edge of the capture signal are you using?	Rising
<code>TIMER_A_CAPTURE_INPUTSELECT_CCI6A</code>	The signal used to trigger the capture	CCI6A
<code>TIMER_A_CAPTURE_ASYNCHRONOUS</code>	Sync the signal to the input clock?	No, don't sync
<code>TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE</code>	Enable the CCR interrupt?	CC6IE = 1
<code>TIMER_A_OUTPUTMODE_OUTBITVALUE</code>	How should the output signal be handled?	OUTMOD=0x0

We've briefly talked about every feature (i.e. function parameter) found in this function except *OutputMode*. The "OUTBITVALUE" (for CCR6) indicates that the value of CCR6's IFG bit should be output to CCR6's Output signal. The output signal can be used by other peripherals or routed to the TA0.6 pin.

Note: With regards to OutputMode, this is just the tip-of-the-iceberg. There are actually 8 possible output mode settings. We will take you through them later in the chapter.

2b. Compare: TIMER_A_initCompare()

The other use of CCR is for *comparisons* to the main timer/counter (TAR).

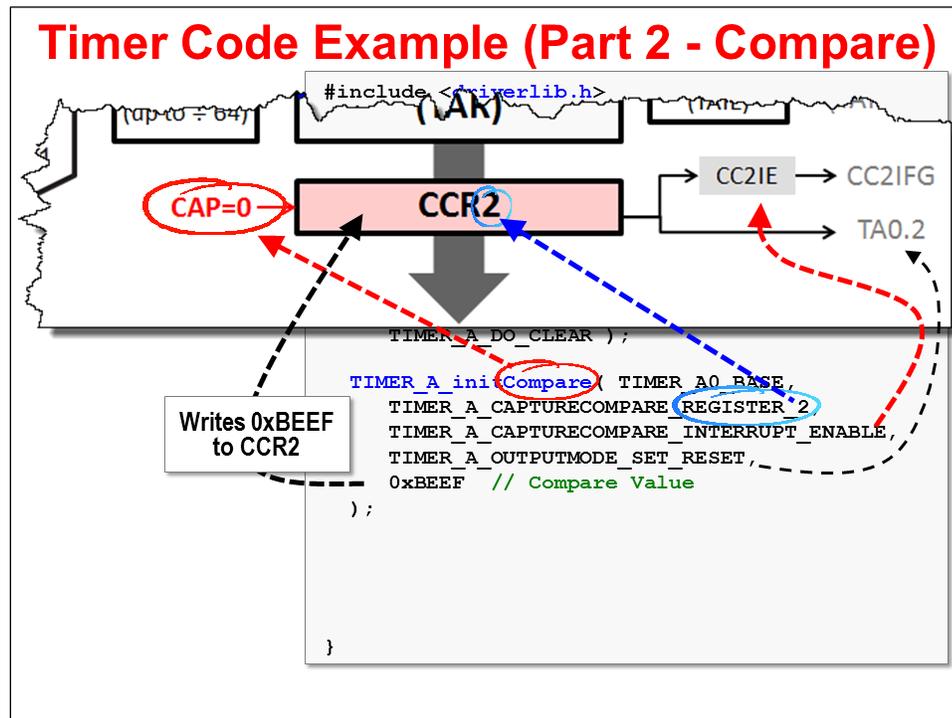


Once again, before we walk through the function that initializes CCR for Compare, let's examine its options:

- Set CAP=0 for the CCR to be configured for Compare Mode. (Opposite from earlier.)
- You must set the CCR2 register to a 16-bit value. When TAR = CCR2:
 - An internal signal called EQU2 is set.
 - If enabled, EQU2 drives the interrupt flag high (CC2IFG).
 - Similar to the Capture mode, the CCR's output signal is modified by EQU2. Again, this signal is available to other internal peripherals and/or routed to a pin (in this case, TA0.2).
 - Again, similar to the Capture mode, there are a variety of possible output modes for the OUT2 signal (which will be discussed shortly).

Compare Code Example

Let's look at the code required to setup CCR2 for use in a Compare operation.



One thing you might notice about the `TIMER_A_initCompare()` function is that it requires fewer parameters than the complementary `initCompare` function.

Example's Parameter Value	What is Parameter For?	Value
<code>TIMER_A0_BASE</code>	Which timer are you using?	TA0
<code>TIMER_A_CAPTURECOMPARE_REGISTER_2</code>	Which CCR is being configured?	CCR2
<code>TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE</code>	Enable the CCR interrupt?	CC2IE = 1
<code>TIMER_A_OUTPUTMODE_SET_RESET</code>	How should the output signal be handled?	OUTMOD=0x3
<code>0xBEEF</code>	What 'compare' value will be written to CCR2?	CCR2 = 0xBEEF

The OutputMode setting will be configured using the "Set/Reset" mode (which correlates to the value 0x3). Once again, with so many different output mode choices, we'll defer the full explanation of this until the next topic.

Summary of Timer Setup Code – Part 2

Here's a summary of the timer setup code we have looked at thus far.

Timer Code Example (Part 2 - Compare)

!
1

@
2

```

#include <driverlib.h>

void initTimerA0(void) {
    // Setup TimerA0 in Up mode with CCR2 compare
    TIMER_A_configureUpMode( TIMER_A0_BASE,
        TIMER_A_CLOCKSOURCE_SMCLK,
        TIMER_A_CLOCKSOURCE_DIVIDER_1,
        TIMER_PERIOD,
        TIMER_A_TAIE_INTERRUPT_ENABLE,
        TIMER_A_CCIE_CCRO_INTERRUPT_ENABLE,
        TIMER_A_DO_CLEAR );

    TIMER_A_initCompare( TIMER_A0_BASE,
        TIMER_A_CAPTURECOMPARE_REGISTER_2,
        TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE,
        TIMER_A_OUTPUTMODE_SET_RESET,
        0xBEEF // Compare Value
    );
}

```

Part 1 of our code configures the timer/counter; i.e. the main element of Timer_A.

Part 2 configures the various Capture/Compare registers (CCR). Due to limited space on the slide we have only included the `initCompare` function for CCR2. In a real application, you might use all of the CCR registers – in which case, our `initTimerA0()` function would become a lot longer.

Before we move onto Part 3 of our timer configuration code, let's spend a few pages explaining the 8 different output mode options available when configuring Capture/Compare Registers.

Output Modes

As you may have already seen, each CCR register has its own associated pin. For CCR1 on Timer0 this pin would be named “TA0.1”. Depending upon which mode you put the CCR into; this pin can be used as an input (for Compare) or an output (for either Capture or Compare).

When the pin is used as an output, its value is determined by the OUT bit-field in its control register. The exact details for this are TA0.1 = TA0CCTL1.OUT. (Sometimes you'll just see this OUT bit abbreviated as OUT1.)

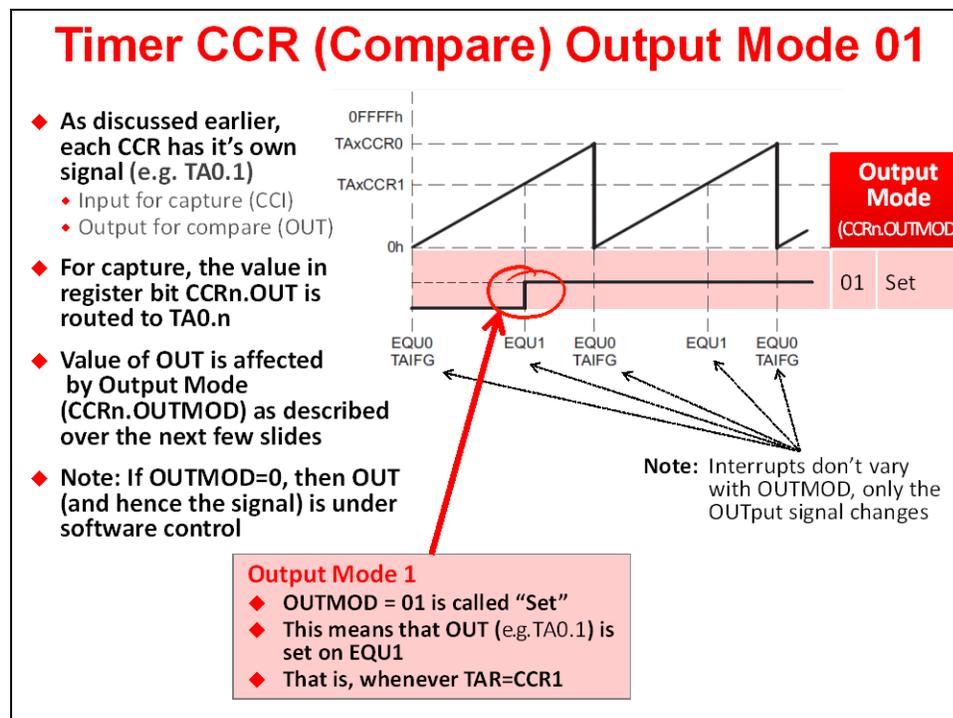
Besides routing the CCR OUT signal to a pin, it can also be used by other MSP430 peripherals. For example, on some devices the A/D converter could be triggered by the timer directly.

So, what is the value of OUT for any given CCR register?

The value of OUT is determined by the OutputMode, as we discussed earlier. (Each CCR control register has its own OUTMOD bit-field). This setting tells the OUT bit how to react as each compare or capture occurs. As previously stated, there are 8 different OutputMode choices.

For example, setting OUTMOD = 0 tells OUT to follow the value of the CCR IFG bit. Whenever the IFG bit gets set, so follows OUT. If IFG is set and you clear it, the pin becomes equal to 0.

What happens to OUT when OUTMOD = 1 (“Set” mode)?



As we can see from the diagram above, when the timer/counter (TAR) counts up to the value in CCR1 (i.e. TAR = CCR1), then a valid comparison is true.

The enumeration for OUTMOD = 1 is called “Set”; whenever TAR=CCR1, then OUT will be “Set” (i.e. OUT = 1). In fact, OUT will remain = 1 until the CCR is reconfigured.

Why use “Set” mode? You might find this mode useful in creating a one-shot type of signal.

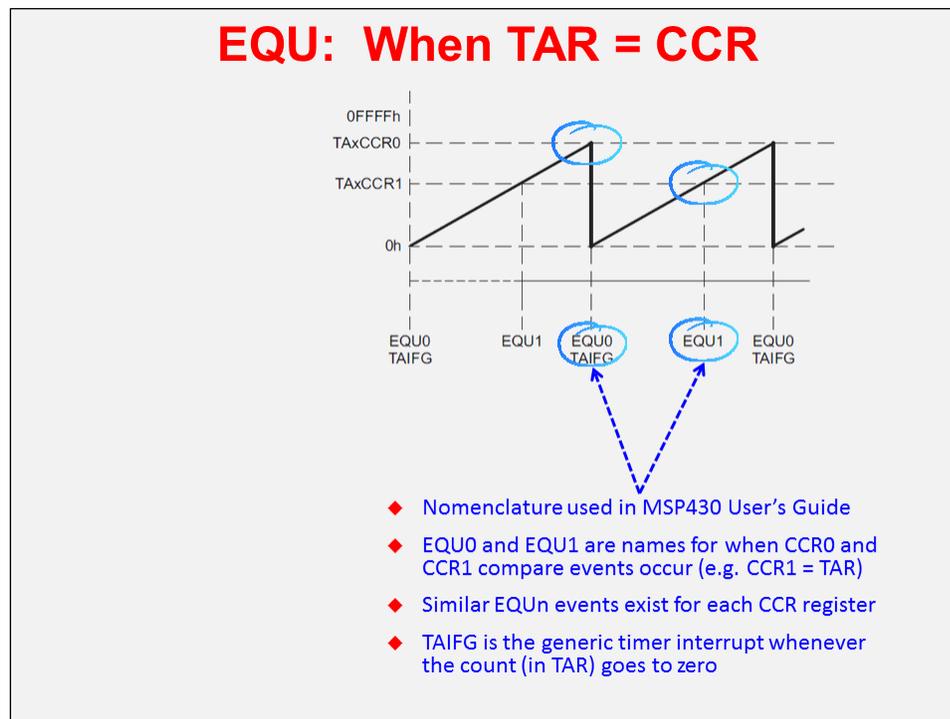
EQU

Before we examine OutputMode 2, let's consider the nomenclature used in the MSP430 User's Guide.

Apparently, there is an EQU (equate) signal inside the timer for each CCR. For example, the equate signal for CCR1 would be called EQU1. While these EQU values cannot be read directly from any of the timer control registers, the documentation makes use of them to describe when a comparison becomes true.

Therefore, when the timer counter (TAR) becomes equal to a compare register (CCR), the associated EQU signal becomes true.

This can be seen in the following diagram captured from the TIMER_A documentation. Notice how EQU0 becomes true when $TAR=CCR0$; likewise, EQU1 becomes true when $TAR=CCR1$.



OUTMOD = 2 (“Toggle/Reset” mode)

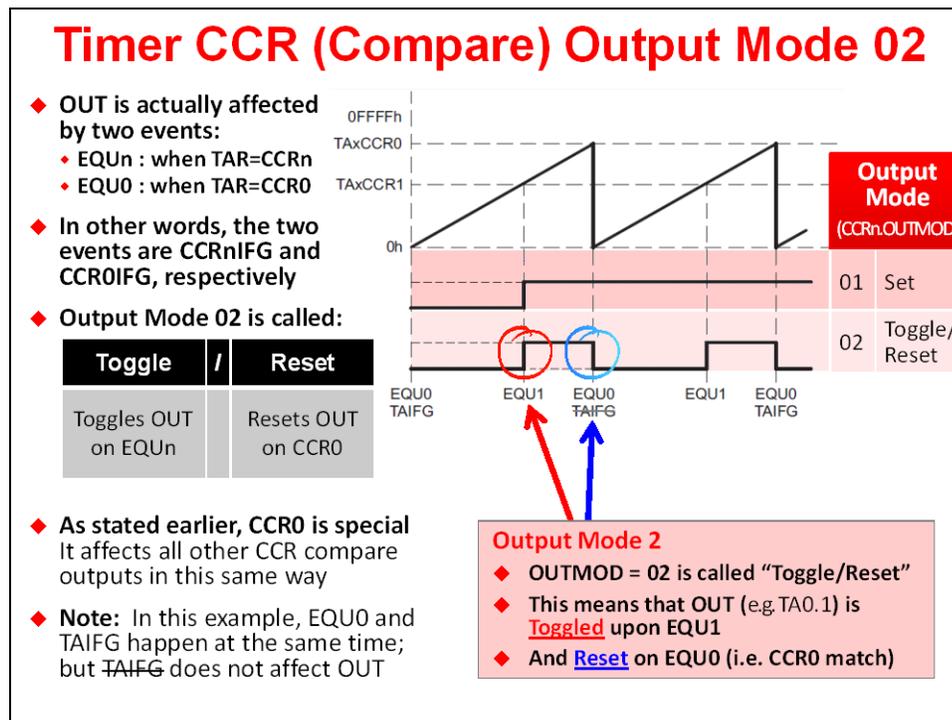
OutputMode 2 is a bit more interesting than the previous output modes. Notice how this mode is called “Toggle/Reset”. Each of these names corresponds to a different event.

- Toggle - This means that OUT_n should be toggled whenever $TAR=CCR_n$
- Reset - This implies that $OUT=0$ (i.e. reset) whenever $TAR=CCR0$

In other words, when the OutputModes are defined by two names, the first one dictates the value of OUT_n whenever the $TAR=CCR_n$ (i.e. whenever EQU_n becomes true). The second name describes what happens to OUT_n whenever $TAR=CCR0$.

Note: Remember what we said earlier, CCR0 is often used in a special way. This is another example of how CCR0 behaves differently than the rest of the CCR’s.

Looking at the diagram below, we can see that in OutputMode 2, the $OUT1$ signal appears to be a pulse whose duty cycle (i.e. width) is proportional to the difference between $CCR0$ and $CCR1$.

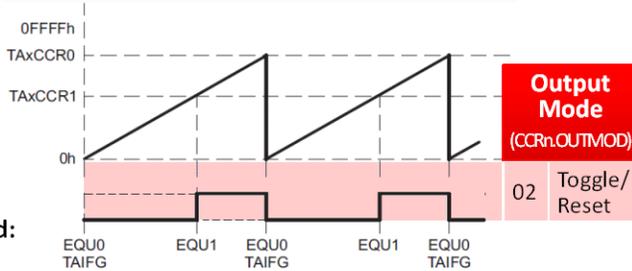


By showing both $OUTMOD=1$ and $OUTMOD=2$ in the same diagram, you can see how the value of OUT_n can be very different depending upon the OutputMode selected.

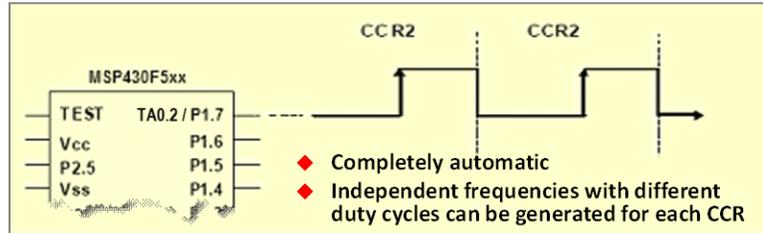
Routing the OUT signal to a pin, as shown here, lets us drive external hardware directly from the output of the timer. (In fact, we'll use this feature to let the timer directly drive an LED during one of the upcoming lab exercises.)

Timer CCR (Compare) Output Mode 02

- ◆ OUT is actually affected by two events:
 - ◆ EQU_n : when TAR=CCR_n
 - ◆ EQU₀ : when TAR=CCR₀
- ◆ In other words, the two events are CCR_nIFG and CCR₀IFG, respectively
- ◆ Output Mode 02 is called: Toggle/Reset

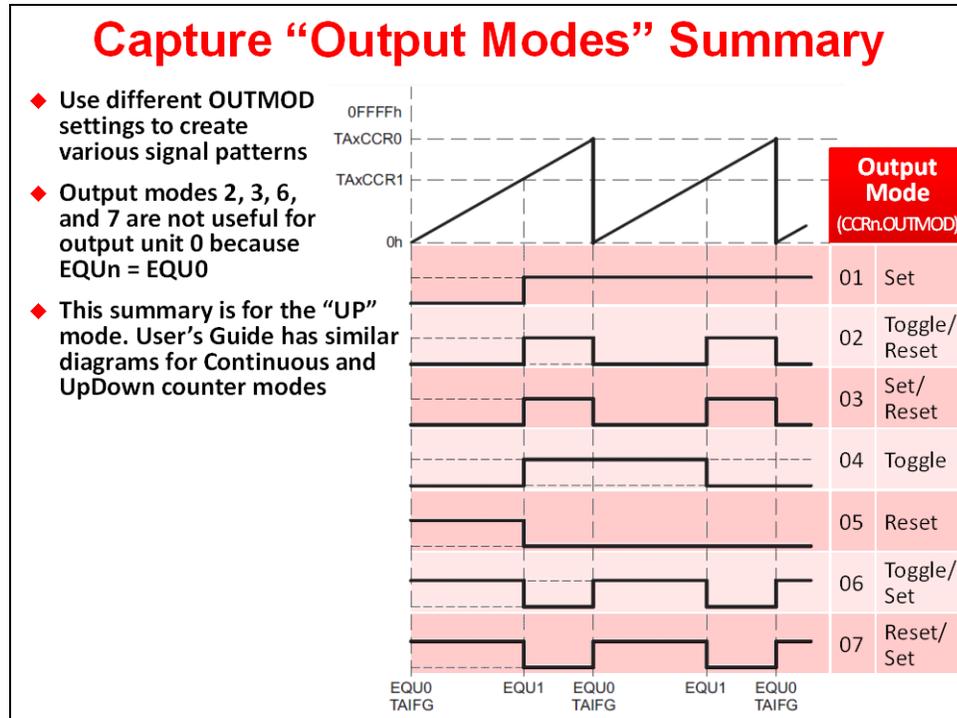


Here's an example of routine TA0.2 (i.e. OUT2) to a GPIO pin:



Summary of Output Modes

While we have only studied a couple of the output modes, we hope you will be able to decipher the remaining modes based on their names. Here is a comparison of all the different OUTPUT waveforms based upon the value of OUTMOD.



Point of Clarification – Only use modes 1, 4, and 5 for CCR0

The second bullet, in the diagram above, states that four of the Output Modes (2, 3, 6, and 7) are not useful when you are working with CCR0.

Why are they not useful?

All four of these OutputModes include two actions:

- One action when: CCR_n=TAR
- A second action when: CCR₀=TAR

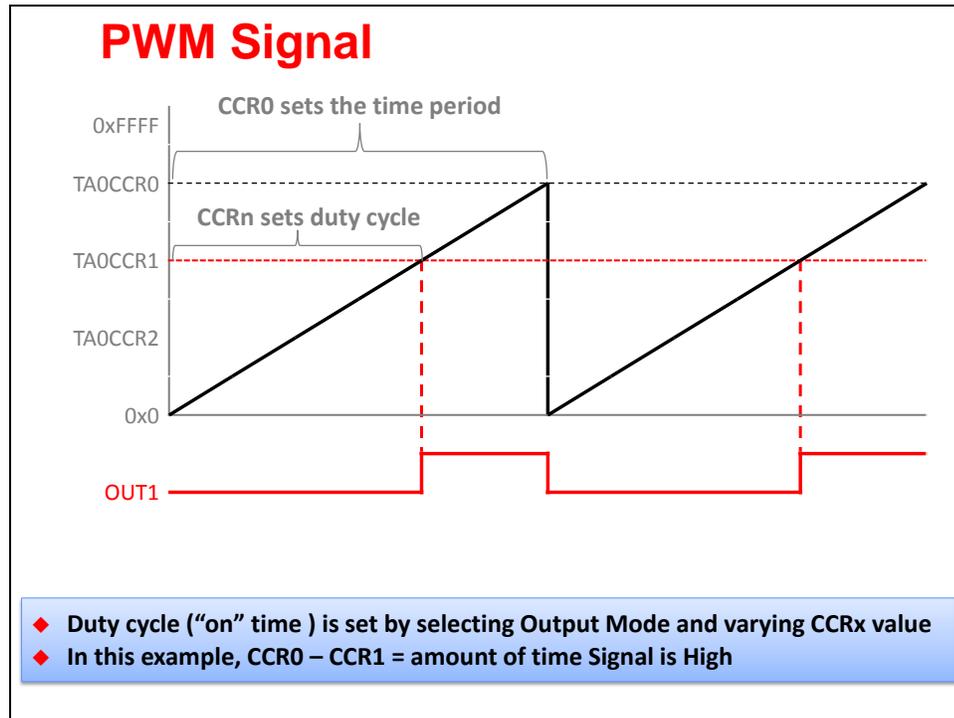
In this case, though, CCR_n = CCR₀. That means these modes could be trying to change OUT₀ in two different ways at the same time.

Bottom Line: When using CCR₀, only set OUTMOD to 0, 1, 4, or 5.

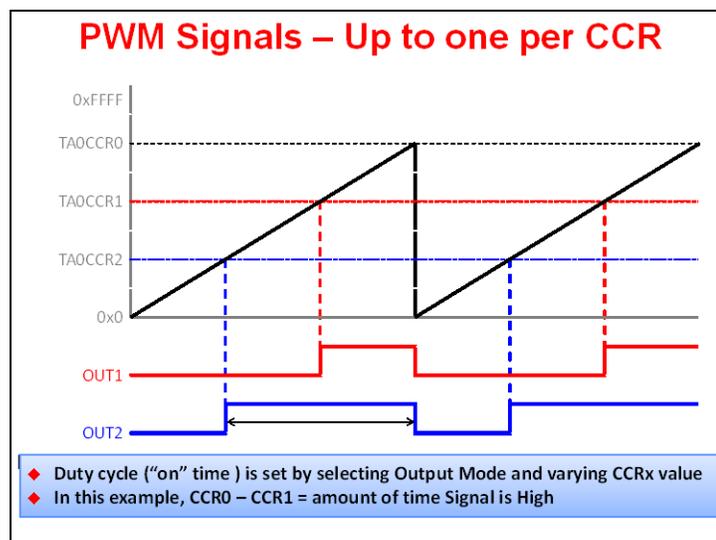
PWM anyone?

PWM, or pulse-width modulation, is commonly used to control the amount of energy going into a system. For example, by making the pulse widths longer, more energy is supplied to the system.

Looking again at the previous example where $OUTMOD = 2$, we can see that by changing the difference between the values of $CCR0$ and $CCRn$ we can set the width of $OUTn$.



In the case of the MSP430, any timer can generate a PWM waveform by configuring the CCR registers appropriately. In fact, if you are using a Timer_A5, you could output 4 or 5 different PWM waveforms.



3. Clear Interrupt Flags and TIMER_A_startTimer()

Part 3 of our timer configuration code is for clearing the interrupt flags and starting the timer.

As described earlier in the workshop, you are not required to clear interrupt flags before enabling an interrupt, but once again, this is common practice. In Part 3 of the example below, we first clear the Timer flag (TA0IFG) using the function call provided by DriverLib. Then, we clear all the CCR interrupts using a single function; notice that the “+” operator tells the function that we want to clear both of these IFG bits.

Timer Code Ex. (Part 3 – Clear IFG’s/Start)

```

#include <driverlib.h>

void initTimerA0(void) {
    // Setup TimerA0 in Up mode with CCR2 Compare
    TIMER_A_configureUpMode( TIMER_A0_BASE,
        TIMER_A_CLOCKSOURCE_SMCLK,
        TIMER_A_CLOCKSOURCE_DIVIDER_1,
        TIMER_PERIOD,
        TIMER_A_TAIE_INTERRUPT_ENABLE,
        TIMER_A_CCIE_CCRO_INTERRUPT_ENABLE,
        TIMER_A_DO_CLEAR );

    TIMER_A_initCompare( TIMER_A0_BASE,
        TIMER_A_CAPTURECOMPARE_REGISTER_2,
        TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE,
        TIMER_A_OUTPUTMODE_SET_RESET,
        0xBEEF ); // Compare Value

    TIMER_A_clearTimerInterruptFlag(
        TIMER_A0_BASE );
    TIMER_A_clearCaptureCompareInterruptFlag(
        TIMER_A0_BASE,
        TIMER_A_CAPTURECOMPARE_REGISTER_0 +
        TIMER_A_CAPTURECOMPARE_REGISTER_2 );
    TIMER_A_startCounter( TIMER_A0_BASE,
        TIMER_A_UP_MODE ); //Make sure this
                          // matches config fxn
}

```

We conclude the code for Part 3 by starting the timer. The start function only has two parameters:

- It's probably obvious that you need to specify which timer that needs to be started.
- The other parameter specifies, once again, the count mode for the timer's counter.

Warning!

Did we get your attention? The timer “start” function ended up being one of the biggest problems during the development of this workshop.

As dumb as it sounds, we missed the fact that you need to set the counter mode (e.g. “UP”) in this function. When we cut/pasted this function from another example, we never thought to change this parameter.

Why, because we thought it had already been specified by using the `TIMER_A_configureUpMode()` function. Well, we found out the hard way that you need to do both. Use the correct function AND specify the correct count mode in the `start` function.

4. Interrupt Code (Vector & ISR)

The last part of our timer code is actually a review since interrupts were covered, in detail, in a previous workshop chapter.

Timer0_A5 Interrupts Review

INT Source	IFG	IV Register	Vector Address	Loc'n
Timer A (CCIFG0)	TA0CCR0.CCIFG	none	TIMER0_A0_VECTOR	53
Timer A	TA0CCR1.IFG1...TA0CCR4.IFG	TA0IV	TIMER0_A1_VECTOR	52

TIMER0_A5

The diagram illustrates the interrupt logic for Timer0_A5. It shows the .CCIFG and .CCIE registers for TA0CCR0 through TA0IFG. The .CCIE bits are connected to an AND gate, which outputs to the SR.GIE pin of the CPU. The CPU has two interrupt vectors: 53 (TIMER0_A0_VECTOR) and 52 (TA0IV).

- ◆ In the interrupts chapter, we learned that most MSP430 interrupts are grouped together and share an interrupt vector, although a few have their own dedicated vector
- ◆ Timers A and B have two vectors: one for CCR0 and the other shared
- ◆ When the CPU responds to TIMER0_A0_VECTOR, the CCR0IFG is auto cleared
- ◆ In the TIMER0_A1_VECTOR ISR, reading **TA0IV** register returns the associated highest priority pending interrupt and clears it's IFG bit

Remember, TIMER_A has two interrupt vectors: one dedicated to CCR0; another shared by TAIFG and all the other CCR's. Below, we provide a simple example of handling both.

Timer Code Example (Part 4 – ISR's)

CCR0
ISR

\$
4

ISR for
CCR2
and TA0IFG

```

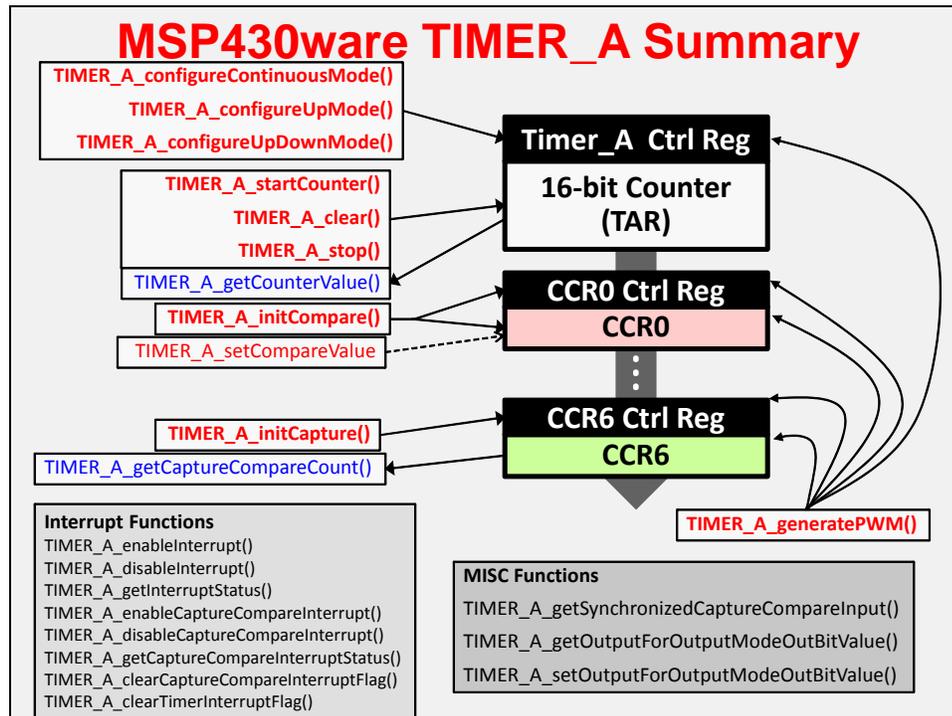
#pragma vector=TIMER0_A0_VECTOR
__interrupt void myISR_TAO_CCR0(void) {
    GPIO_toggleOutputOnPin( ... );
}

#pragma vector=TIMER0_A1_VECTOR
__interrupt void myISR_TAO_Other(void) {
    switch(_even_in_range( TA0IV, 10 )) {
        case 0x00: break;           // None
        case 0x02: break;           // CCR1 IFG
        case 0x04: break;           // CCR2 IFG
        GPIO_toggleOutputOnPin(...);
        break;
        case 0x06: break;           // CCR3 IFG
        case 0x08: break;           // CCR4 IFG
        case 0x0A: break;           // CCR5 IFG
        case 0x0C: break;           // CCR6 IFG
        case 0x0E: break;           // TA0IFG
        GPIO_toggleOutputOnPin(...);
        break;
        default: _never_executed();
    }
}
                
```

TIMER_A DriverLib Summary

This diagram attempts to summarize the functions found in the TIMER_A module of the MSP430ware Driver Library.

Many of the functions have arrows pointed to/from the three main parts of the timer peripheral: TAR (the main timer/counter); CCR (used for Compare); and CCR (used for Capture). The arrows indicate whether the function reads or writes the associated registers.



The bottom of the slide contains two boxes: one summarizes the Interrupt related functions while the other contains three functions that read/write the input and output bit values.

Differences between Timer's A and B

The Timer_A and Timer_B peripherals are very similar. The following slide highlights the few differences between them.

Similarities

Timer_A vs Timer_B

- ◆ Timer_B's default functionality is identical to Timer_A
- ◆ Names are (almost) the same: TAR → TBR, TA0CTL → TB0CTL, etc.

Timer_A specific features

- ◆ "Sampling Mode" acts like a digital sample & hold
 - ◆ Timer_A can latch CCI input (to SCCI) upon compare
 - ◆ Makes it easy to implement software UART's
 - ◆ Timer_B cannot latch CCI directly, but most devices with Timer_B have dedicated communication peripherals

Timer_B specific features

- ◆ Compare (CCRx) registers are double-buffered & can be updated in groups
 - ◆ Preserves PWM "dead time" between driving complementary outputs (H-bridge)
 - ◆ More care is needed when implementing edge-aligned PWM with Timer_A
- ◆ TBR configurable for 8, 10, 12 or 16-bits counter (default is 16-bits)
 - ◆ Provides range of periods when used in 'Continuous' mode
- ◆ Tri-state function from external pin
 - ◆ External TBOUTH pins puts all Timer_B pins into high-impedance
 - ◆ With Timer_A, you would need to reconfigure pins in software

Hint: For a more complete understanding of these differences, we highly recommend that you refer to *MSP430 Microcontroller Basics*. John Davies does a great job of describing the differences between these timers. Furthermore, his discussion of generating PWM waveforms using these timers is extremely good. If you've never heard of the differences between *edge-aligned* and *centered* PWM waveforms, check out his MSP430 book.

MSP430 Microcontroller Basics by John H. Davies, (ISBN-10 0750682760) [Link↗](#)

Notes

Lab 6 – Using Timer_A

Lab 6 – Using Timer_A

◆ Time for the lab prep Worksheet:

- ◆ What time is it?
- ◆ Capture vs Compare
- ◆ 4 steps to timer programming
- ◆ Simple PWM generation

◆ Lab 6a – Simple Timer Interrupt

- ◆ Create a CCR0 interrupt with the timer counting in Continuous Mode
- ◆ ISR toggles LED

◆ Optional Exercises

Lab 6b – Timer using Up Mode

- ◆ Similar to Lab6a, but using Up mode

Lab 6c – Timer with Directly Driven LED

- ◆ Similar to Lab6b, but with the timer directly driving the LED

Lab 6d – Simple PWM Signal

- ◆ Alter the brightness of the LED by changing the PWM duty cycle



Time:

Worksheet – 15 mins

Labs – 30 mins

Note: The solutions exist for all of these exercises, but the instructions for Lab 6d are not yet included. These will appear in a future version of the course.

Lab Topics

<i>Lab 6 – Using Timer_A</i>	6-35
<i>Lab 6a – Simple Timer Interrupt</i>	6-37
Lab 6a Worksheet	6-37
File Management	6-41
Edit <i>myTimers.c</i>	6-42
Debug/Run	6-43
<i>(Extra Credit) Lab 6b – Timer using Up Mode</i>	6-44
Lab 6b Worksheet	6-44
File Management	6-46
Change the Timer Setup Code	6-47
Debug/Run	6-47
Archive the Project	6-48
Timer_B (Optional)	6-49
<i>(Extra Credit) Lab 6c – Timer using Up Mode</i>	6-50
Lab 6c Worksheet	6-50
File Management	6-54
Change the GPIO Setup	6-54
Change the Timer Setup Code	6-55
Debug/Run	6-56
(Optional) Lab 6c – Portable HAL	6-60
<i>(Optional) Lab 6d – Simple PWM</i>	6-61
<i>Chapter 6 Appendix</i>	6-62

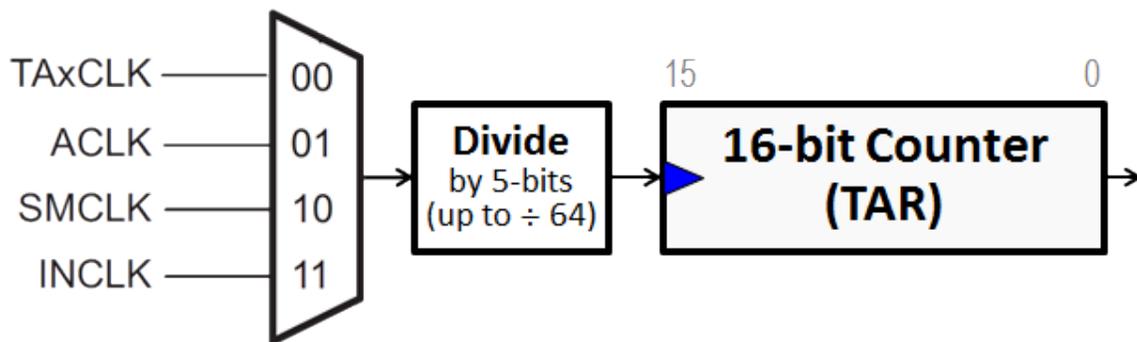
Lab 6a – Simple Timer Interrupt

Similar to `lab_05a_buttonInterrupt`, we want to blink an LED based upon a timer. In this case, though, we'll use `TIMER_A` to generate an interrupt. During the interrupt routine we'll toggle the GPIO value that drives an LED on our Launchpad board.

As we write the ISR code, you should see that `TIMER_A` has two interrupts:

- One is dedicated to `CCR0` (capture and compare register 0).
- The second handles all the other timer interrupts

This first `TIMER_A` lab will use the main timer/counter rollover interrupt (called `TA0IFG`). As with our previous interrupt lab (with GPIO ports), this ISR should read the `TimerA0 IV register (TA0IV)` and decipher the correct response using a switch/case statement.



Lab 6a Worksheet

Goal: Write a function setting up `Timer_A` to generate an interrupt every two seconds.

1. How many clock cycles does it take for the 16-bit `TimerA0` to 'rollover'? (Hint: 16-bit timer)

2. If our goal is to generate a two second interrupt rate, what source and divide by value will get us closest to 2 seconds? (see diagram above)

Clock input: (circle one) **ACLK** **SMCLK**

DriverLib enumeration: `TIMER_A_CLOCKSOURCE_` _____

Divide by: `TIMER_A_CLOCKSOURCE_DIVIDER_` _____

Hint: Since we are interested in 2 seconds, a slow clock might work best.

Another Hint: Look up the arguments for the `TIMER_A_configureContinuousMode()` function in the *MSP430® Peripheral Driver Library User's Guide*.

3. Calculate the Timer frequencies for the clocks & divider values you chose in the previous step.

This lab exercise uses the clock setup from Chapter 4. So you don't have to look it up, we've copied the values into the table below:

Clock	'F5529 Launchpad	'FR5969 Launchpad
ACLK	32 KHz	32 KHz
SMCLK	8 MHz	8 MHz
MCLK	8 MHz	8 MHz

Which clock did you choose in the previous step? Write its frequency below and then calculate the timer rates.

Timer Clock Source: _____

Clock Frequency = _____ cycles/second

Timer Frequency = $\frac{\text{clock frequency}}{\text{timer clock divider}}$ = _____

Timer Output = $\frac{\text{timer frequency}}{\text{counts for timer to rollover}}$ = _____

4. Which Timer do you need to use for this lab exercise?

In a later lab exercise we will output the timer directly to a BoosterPack pin. Unfortunately, the two Launchpad's map different timers to their BoosterPack. (Partly this is due to the 'FR5969 only using the 20-pin BoosterPack layout; versus the 40-pin XL layout for the 'F5529.)

Here are the recommended timers:

Launchpad	Timer	Short Name	Timer's DriverLib Enum
'F5529	TIMER0_A3	TA0	TIMER_A0_BASE
'FR5969	Timer1_A5	TA1	TIMER_A1_BASE

Write down the timer enumeration you need to use: **TIMER_** _____ **_BASE**

5. Write the `TIMER_A_configureContinuousMode()` function.

The first part of our timer code is to setup the Timer control registers (TAR, TACTL). Of course, we'll do this using the following DriverLib function.

```
TIMER_A_configureContinuousMode(
    TIMER_ ______BASE,           // Which timer to setup?
    _____,                       // Timer clock source
    _____,                       // Timer clock divider
    _____,                       // Enable interrupt on TAR counter rollover
    TIMER_A_DO_CLEAR                 // Clear TAR & previous divider state
);
```

Hint: Where do you get help writing this function? We highly recommend the *MSP430ware DriverLib Users Guide*. (See 'docs' folder inside MSP430ware's **driverlib** folder.) Another suggestion would be to examine the header file: (`timer_a.h`).

6. Skip this step ... it's not required.

We outlined 4 steps to setting up a Timer_A. The second step is where you configure the Capture and Compare features. Since this exercise doesn't need to use those features, you can skip this step.

7. Complete the code to for the 3rd part of the "Timer Setup Code".

The third part of the timer setup code includes:

- ~~Enable the interrupt (IE)~~ ... we don't have to do this, since it's done by the `TIMER_A_configureContinuousMode()` function (from question 5 on page 6-39).
- Clear the appropriate interrupt flag (IFG)
- Start the timer

```
// Clear the timer interrupt flag
_____ ( TIMER_____BASE ); // Clear TA0IFG
```

```
// Start the timer
_____ ( _____ // Function to start timer
    TIMER_____BASE, // Which timer?
    _____ // Run in Continuous mode
```

8. Change the following interrupt code to toggle LED2 when Timer_A rolls-over.

a) Fill in the details for your Launchpad.

Port/Pin number for LED2: Port _____, Pin _____

Interrupt Vector: #pragma vector = _____ _VECTOR

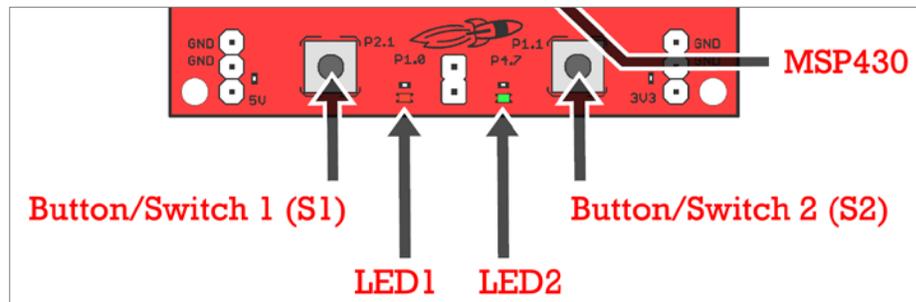
Interrupt Vector register: _____
 (For example, we used P1IV for GPIO Port 1)

b) Here is the interrupt code that exists from a previous exercise, change it as needed:

```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void)
{
    switch( __even_in_range( P1IV , 16 )) {
        case 0: break; // No interrupt
        case 2: break; // Pin 0
        case 4: // Pin 1
            GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
            break;
        case 6: break; // Pin 2
        case 8: break; // Pin 3
        case 10: break; // Pin 4
        case 12: break; // Pin 5
        case 14:
            break; // Pin 6
        case 16: break; // Pin 7
        default: _never_executed();
    }
}
```

Hint:

	'F5529 LP	'FR5969 LP	Color
LED1 (Jumper)	P1.0	P4.6	Red
LED2	P4.7	P1.0	Green
Button 1	P2.1	P4.5	Black
Button 2	P1.1	P1.1	Grey



Please verify your answers before moving onto the lab exercise.

File Management

2. Verify that all projects (and files) in your workspace are closed.

If some are open, we recommend closing them.

3. Import `lab_06a_timer` project.

We have created the initial lab project for you.

```
C:\msp430_workshop\<<target>\lab_06a_timer
```

4. Briefly examine the project files

This project uses code we have written earlier in the workshop, though we have partitioned some of this code into separate files:

- `myGpio.c`
 - The LED pins are configured as outputs and set to Low.
 - For the 'FR5969, the LFXT pins are set as clock inputs; and, the pins are unlocked.
- `myClocks.c`
 - For 'F5529 users, this is the same code you wrote in the *Clocks* chapter except that we moved the `initPowerMgmt()` function into this file since you select the power-level to allow for a given clock frequency.
 - For 'FR5969 users, this file was changed slightly from our earlier exercise. `ACLK` now uses the 32 KHz crystal rather than `VLO`. Also, `MCLK` and `SMCLK` are set to 8MHz.

Edit *myTimers.c*

5. Edit the timer source file.

We want to setup the timer to generate an interrupt every second. The TAIFG interrupt service routine will then toggle LED2 on/off.

**Worksheet
Question #5**

```
void initTimers(void)
{
    // 1. Setup Timer (TAR, TACTL) in Continuous mode using ACLK
    TIMER_A_ _____(
        TIMER__BASE,                // Which timer
        TIMER_A_ _____,        // Which clock
        TIMER_A_ _____,        // Clock divider
        TIMER_A_ _____,        // Enable INT on rollover?
        TIMER_A_DO_CLEAR            // Clear timer counter
    );

    // 2. Setup Capture & Compare features
    // This example does not use these features

    // 3. Clear/enable flags and start timer
    TIMER_A_ _____( TIMER_A1_BASE ); // Clear Timer Flag

    TIMER_A_startCounter(
        TIMER__BASE,
        TIMER_A_ _____        // Which timer mode
    );
}

//***** Interrupt Service Routine *****
#pragma vector=TIMER1_A1_VECTOR
__interrupt void timer1_ISR (void)
{
    // 4. Timer ISR and vector
    switch( __even_in_range( _____, 14 )) { // Read timer IV register
    case 0: break; // None
    case 2: break; // CCR1 IFG
    case 4: break; // CCR2 IFG
    case 6: break; // CCR3 IFG
    case 8: break; // CCR4 IFG
    case 10: break; // CCR5 IFG
    case 12: break; // CCR6 IFG
    case 14: // TAR overflow

        // Toggle the LED2 on/off
        GPIO_toggleOutputOnPin( );
        break;

    default: _never_executed();
    }
}
```

**Worksheet
Question #7**

**Worksheet
Question #8**

6. Modify the *Unused Interrupts* source file.

Since our timer code uses an interrupt, we need to comment out its associated vector from the `unused_interrupts.c` file.

7. Build your code and repair any errors.

Debug/Run

8. Launch the debugger.**9. Set a breakpoint inside the ISR.**

We found it worked well to set a breakpoint on the 'switch' statement.

10. Run your code.

If all worked well, when the counter rolled over to zero, an interrupt occurred ... which resulted in the processor halting at a breakpoint inside the ISR.

11. If the breakpoint occurred, skip to the next step ...

If you did not reach the breakpoint inside your ISR, here are a few things to look for:

- Is the interrupt flag bit (IFG) set?
- Is the interrupt enable bit (IE) set?
- Are interrupts enabled globally?

12. If the breakpoint occurred, then resume running again.

You should always verify that your interrupts work by taking more than 'one' of them. A common cause of problems occurs when the IFG bit is not cleared. This means you take one interrupt, but never get a second one.

In our current example, reading the TA1IV (or TA0IV for 'F5529 users) should clear the flag, so the likelihood of this problem occurring is small, but sometimes the problem still occurs due to a logical error in coding the interrupt routine.

13. Did the LED toggle?

If you are executing the ISR (i.e. hitting the breakpoint) and the LED is not toggling, try single-stepping from the point where the breakpoint occurs. Make sure your program is executing the GPIO instruction.

A common error, in this case, is accidentally putting the "do something" code (in our case, the GPIO toggle function) into the wrong 'case' statement.

(Extra Credit) Lab 6b – Timer using Up Mode

In this timer lab we switch our code from counting in the "Continuous" mode to the "Up" mode. This gives us more flexibility on the frequency of generating interrupts and output signals.

From the discussion you might remember that TIMER_A has two interrupts:

- One is dedicated to CCR0 (capture and compare register 0).
- The second handles all the other timer interrupts

In our previous lab exercise, we created an ISR for the group (non-dedicated) timer ISR. This lab adds an ISR for the dedicated (CCR0 based) interrupt.

Each of our two ISR's will toggle a different colored LED.

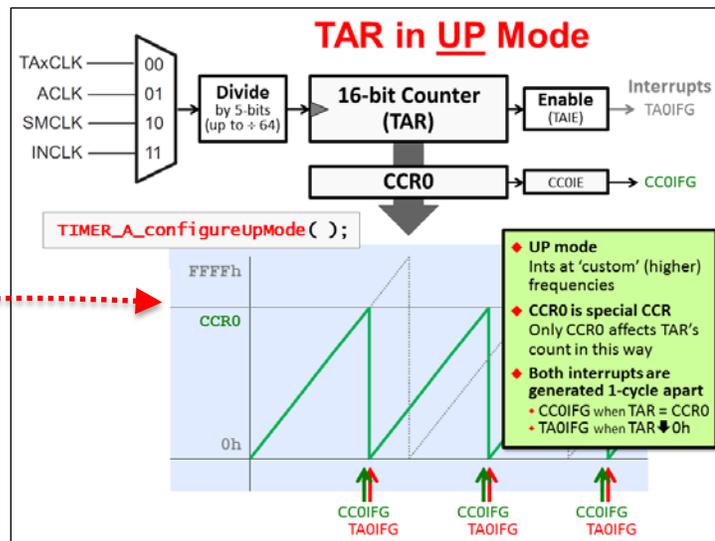
The goal of this part of the lab is to:

```
// TimerA1 in Up mode using ACLK
// Toggle LED1 on/off every second using CCR0IFG
// Toggle LED2 on/off every second using TA1IFG
```

Lab 6b Worksheet

1. Calculate the timer period that will go into CCR0 to set the proper interrupt rate.

Here's a quick review from our discussion.



Timer_A's counter (TAR) will count up until it reaches the value in the CCR0 capture register, then reset back to zero. What value do we need to set CCR0 to get a ½ second interval?

$$\begin{aligned} \text{Timer Frequency} &= \frac{32 \text{ KHz}}{\text{clock frequency}} \div \frac{1}{\text{timer clock divider}} = \frac{32 \text{ KHz}}{\text{timer frequency}} \\ \text{Timer Output} &= \frac{32 \text{ KHz}}{\text{timer frequency}} \div \frac{1}{\text{timer period (i.e. CCR0 value)}} = \frac{1}{2} \text{ SECOND} \end{aligned}$$

2. Complete the `TIMER_A_configureUpMode()` function?

This function will replace the `TIMER_A_configureContinuousMode()` call we made in our previous lab exercise.

Hint: Where to get help for writing this function? Once again, we recommend the MSP430ware DriverLib users guide (“docs” folder inside MPS430ware’s DriverLib).

Another suggestion would be to examine the `timer_a.h` header file.

```
TIMER_A_configureUpMode(  
    TIMER____BASE, // Which timer are you using?  
    TIMER_A_CLOCKSOURCE_ACLK, // Timer clock source  
    TIMER_A_CLOCKSOURCE_DIVIDER_1, // Timer clock divider  
    _____, // Period (calculated in previous question)  
    TIMER_A_TAIE_INTERRUPT_ENABLE, // Enable interrupt on TAR counter rollover  
    _____, // Enable CCR0 compare interrupt  
    TIMER_A_DO_CLEAR // Clear TAR & previous divider state  
);
```

3. Modify your previous code. We need to clear both interrupts and start the timer.

We copied the code from the previous lab into this question. It needs to be modified to meet our new objectives for this lab.

Here are some hints:

- Add an extra line of code to clear the CCR0 flag (we left a blank space below for this)
- Don’t make the mistake we made ... look very carefully at the ‘start’ function. Is there anything that needs to change in that function call?

```
// Clear the timer flag and start the timer  
TIMER_A_clearTimerInterruptFlag( TIMER____BASE ); // Clear TA0IFG  
_____  
_____  
    TIMER____BASE,  
    _____ );  
TIMER_A_startCounter( TIMER____BASE, // Start timer in  
    TIMER_A_____MODE ); // _____ mode
```

4. Add a second ISR to toggle the LED1 whenever the CCR0 interrupt fires.

On your Launchpad, what Port/Pin number does the LED1 use? _____

Here we've given you a bit of code to get you started:

```
#pragma vector= _____
__interrupt void ccr0_ISR (void)
{
    // Toggle the LED1 on/off
    _____
}

```

Please verify your answers before moving onto the lab exercise.

File Management

1. Copy/Paste the lab_06a_timer to lab_06b_upTimer.

- In Project Explorer, right-click on the lab_06a_timer project and select "Copy".
- Then, click in an open area of Project Explorer and select paste.
- Finally, rename the copied project to lab_06b_upTimer.

Note: If you didn't complete lab_06a_timer – or you just want a clean starting solution – you can import the lab_06a_timer archived solution.

2. Close the previous project: lab_06a_timer

3. Delete old, readme file and import the new one.

```
C:\msp430_workshop\<target>\lab_06b_upTimer
```

4. Build the project to verify no errors were introduced.

Change the Timer Setup Code

In this part of Lab 6, we will be setting up TimerA0 in Up Mode.

5. Modify the timer configuration function, configuring it for ‘Up’ mode.

You should have a completed copy of this code in the Lab 6b Worksheet.

Please refer to the Lab Worksheet for assistance. (Question 2, Page 6-45).

6. Modify the rest of the timer setup code, where we clear the interrupt flags, enable the individual interrupts and start the timer.

Please refer to the Lab Worksheet for assistance. (Question 3, Page 6-45).

7. Add the new ISR we wrote in the Lab Worksheet to handle the CCR0 interrupt.

When this step is complete, you should have two ISR’s in your `main.c` file.

Please refer to the Lab Worksheet for assistance. (Question 4, Page 6-46).

8. Don’t forget to modify the “unused” vectors (`unused_interrupts.c`).

Failing to do this will generate a build error. (Most of us saw this error back during the lab exercise for the *Interrupts* chapter.)

9. Build the code to verify that there are no syntax errors; fix any as needed.

Debug/Run

Follow the same basic steps as found in the previous lab for debugging.

10. Launch the debugger and set a breakpoint inside both ISR’s.

11. Run your code.

If all worked well, when the counter rolled over to zero, an interrupt should occur. Actually, two interrupts should occur. Once you reach the first breakpoint, resume running your code and you should reach the other ISR.

Which ISR was reached first? _____

Why? _____

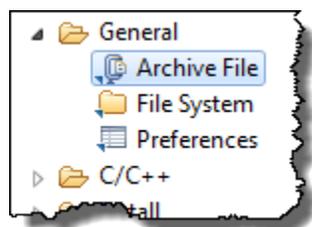
12. Remove the breakpoints and let the code run. Do both LED’s toggle?

Archive the Project

Thus far in this workshop, we have imported many projects from archives ... but we haven't asked you to create an archive, yet. It's not hard, as you'll find out.

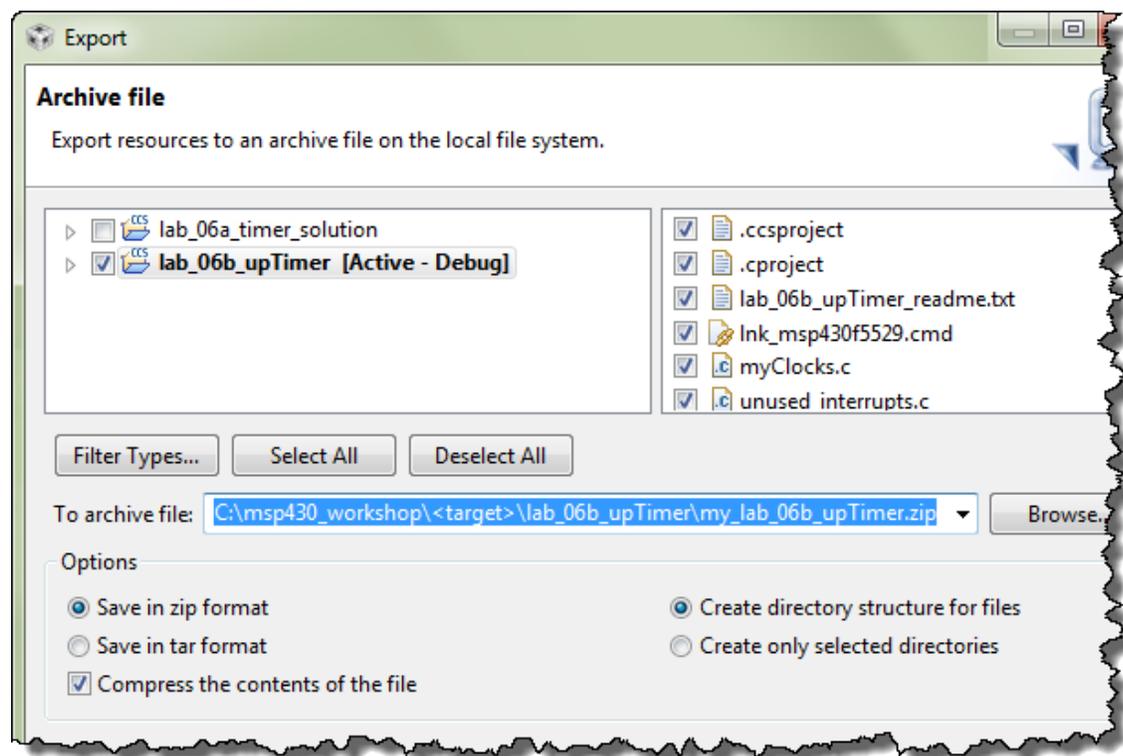
13. Export your project to the lab's file folder.

- Right-click the project and select 'Export'
- Select 'Archive File' for export, then click Next



- Fill out the dialog as shown below, choosing: the 'upTimer' lab; "Save in zip format", "Compress the contents of the file"; and the following destination:

C:\msp430_workshop\<>target>\lab_06b_upTimer\my_lab_06b_upTimer.zip



Timer_B (Optional)

Do you remember during the discussion that we said Timer_A and Timer_B were very similar? In fact, the timer code we have written can be used to operate Timer_B ... with 4 simple changes:

- **It's a different API ... but not really.**

Rather than using the TIMER_A module from DriverLib, you will need to use TIMER_B; unless you're using one of the few unique features of TIMER_B, the rest of the API is the same. In other words, you can carefully search and replace TIMER_A for TIMER_B.

- **Specify a different timer.**

Since you're using a different timer, you need to specify a different timer 'base'. For either the 'F5529 or 'FR5969 you should use TIMER_B0_BASE to specify the timer instance you want to use.

- **You need to use the TIMER_B interrupt vector.**

This changes the #pragma line where we specify the interrupt vector.

- **You need to use the TIMER_B interrupt vector register.**

You need to read the TB0IV register to ascertain which TIMER_B flag interrupted the CPU.

All of these are simple changes. Try implementing TIMER_B on your own.

Note: While we don't provide step-by-step directions, we did create a solution file for this challenge.

(Extra Credit) Lab 6c – Timer using Up Mode

This lab is a minor adaptation of the TIMER_A code in the previous exercise. The main difference is that we'll connect the output of Timer_A CCR2 (TA0.2 or TA1.2) directly to a GPIO pin.

We are still using Up mode, which means that CCR0 is used to reset TAR back to 0. We needed to choose another signal to connect to the external pin... we arbitrarily choose to use CCR2 to generate our output signal for this exercise.

In our case, we want to drive an LED directly from the timer's output signal...

...unfortunately, the Launchpad does not have an LED connected directly to a timer output pin, therefore we'll need to use a jumper in order to make the proper connection. As we alluded to earlier in the chapter, in the case of Timer_A, the Launchpad's route different timer pins to the Boosterpack pin-outs.

Here's an excerpt from the 'F5529 lab solution:

```
// When running this lab exercise, you will need to pull the JP8 jumper and
// use a jumper wire to connect signal from pin ____ (on boosterpack pinouts) to
// JP8.2 (bottom pin) of LED1 jumper ... this lets the TA0.2 signal drive the
// LED1 directly (without having to use interrupts)
```

And a similar statement from the 'FR5969 lab solution:

```
// When running this lab exercise, you will need to pull the J6 jumper and
// use a jumper wire to connect signal from pin ____ (on boosterpack pinouts) to
// J6.2 (bottom pin) of the LED1 jumper ... this lets the TA1.2 signal drive
// LED1 directly (without having to use interrupts)
```

Lab 6c Worksheet

1. Figure out which BoosterPack pin to drive with the timer's output.

We want to choose a BoosterPack pin, as this will make it easy for us to jumper the signal over to LED1. Which BoosterPack pin can our timer output?

Remember, for 'F5529 we're using TA0.2, while 'FR5969 folks are using TA1.2.

There are really two parts to this question:

- a) What GPIO port/pin output is TA0.2 (or TA1.2) combined with?

Hint: *There are a couple places in the datasheet to find this information. We recommend searching your device's datasheet for "TA0.2" or "TA1.2".*

GPIO port/pin: _____

- b) Next, what BoosterPack pin is this GPIO connected to?

This information can be found directly from the Launchpad. Look for the silkscreened labels next to each BoosterPack pin. (If you're getting a little older, you may need a magnifying glass to answer this question...or will need to zoom in on the Launchpad's photo.)

BoosterPack pin: _____

2. Write the function to set this Pin/Port to be used as a timer pin (as opposed to an output pin).

F5529

'F5529 Users, here's the function you need to complete:

```
GPIO_setAs_____ (  
    _____ ,  
    _____ ) ;
```

FR5969

'FR5969 Users, your function requires one more argument:

```
GPIO_setAs_____ (  
    _____ ,  
    _____ ,  
    _____ ) ;
```

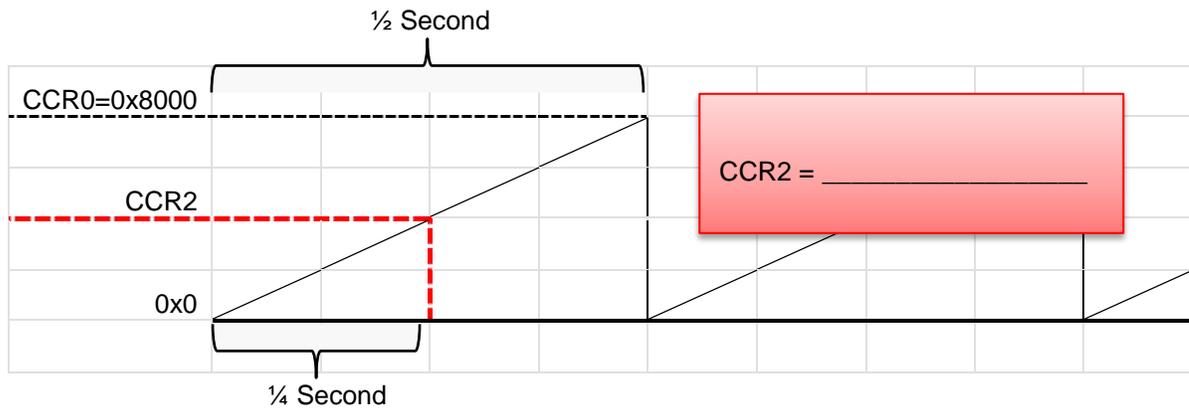
3. Modify the `TIMER_A_configureUpMode()` function?

Here is the code we wrote for the previous lab exercise. We only need to make one change to the code. Since we will drive the signal directly from the timer, we don't need to generate the CCR0 interrupt anymore.

Mark up the code below to disable the interrupt. (We'll bet you can make this change without even looking at the API documentation. Intuitive code is one of the benefits of using DriverLib!)

```
TIMER_A_configureUpMode(  
    TIMER____BASE,           // Which timer are you using  
    TIMER_A_CLOCKSOURCE_ACLK, // Timer clock source  
    TIMER_A_CLOCKSOURCE_DIVIDER_1, // Timer clock divider  
    0xFFFF / 2,             // Period: (0x8000) / 32Khz = 1/2 sec  
    TIMER_A_TAIE_INTERRUPT_ENABLE, // Enable interrupt on TAR counter rollover  
    TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE, // Enable CCR0 compare interrupt  
    TIMER_A_DO_CLEAR        // Clear TAR & previous divider state  
);
```

4. What 'compare' value does CCR2 need to equal in order to toggle the output signal at 1/4 second?



5. Add a new function call to setup Capture and Compare Register 2 (CCR2). This should be added to initTimers().

CCR2 value
calculated above
goes here

```
TIMER_A_init_____(  
    TIMER____BASE, // Which timer are you using?  
    _____, // Select the CCR2 register  
    TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE, // Disable int; since driving LED directly  
    TIMER_A_OUTPUTMODE_TOGGLE, // Toggle mode creates on/off signal  
    _____, // Compare value to toggle at 1/4 second  
);
```

6. Compare your previous code to that below.

What did we change? _____

Note, this is the 'F5529 code example. The 'FR5969 uses a slightly different interrupt vector and interrupt vector register.

```
#pragma vector=TIMER0_A1_VECTOR
__interrupt void timer0_ISR(void)
{
    switch(__even_in_range( TA0IV, 14 )) {
        case 0:  break;                // No interrupt
        case 2:  break;                // CCR1 IFG
        case 4:  break;                // CCR2 IFG
                _no_operation();
                break;
        case 6:  break;                // CCR3 IFG
        case 8:  break;                // CCR4 IFG
        case 10: break;                // CCR5 IFG
        case 12: break;                // CCR6 IFG
        case 14: break;                // TAR overflow
                GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN7 );
                break;
        default: _never_executed();
    }
}
```

During debug, we will ask you to set a breakpoint on 'case 4'.

Why should case 4 not occur, and thus, the breakpoint never reached?

7. Why is better to toggle the LED directly from the timer, as opposed to using an interrupt (as we've done in the previous lab exercises)?

File Management

1. **Copy/Paste the lab_06b_upTimer to lab_06c_timerDirectDriveLed.**
 - a) In Project Explorer, right-click on the lab_06b_upTimer project and select “Copy”.
 - b) Then, click in an open area of Project Explorer and select paste.
 - c) Finally, rename the copied project to lab_06c_timerDirectDriveLed.

Note: If you didn't complete lab_06b_upTimer – or you just want a clean starting solution – you can import the archived solution for it.

2. **Close the previous project: lab_06b_upTimer**
3. **Delete old, readme file.**

Delete the old readme file and import the new one from:

```
C:\msp430_workshop\<target>\lab_06c_timerDirectDriveLed
```
4. **Build the project to verify no errors were introduced.**

Change the GPIO Setup

Similar to the earlier parts of the lab, we will make the changes discussed in the worksheets.

5. **Modify the initGPIO function, defining the appropriate pin to be configured for the timer peripheral function.**

Please refer to the Lab6c Worksheet for assistance. (Step 2, Page 6-51).

Change the Timer Setup Code

6. **Modify the timer configuration function, we are still using 'Up' mode, but not using one of the interrupts anymore.**

Please refer to the Lab Worksheet for assistance. (Step 3, Page 6-51).

7. **Add a call to the TIMER_A function that configures CCR2.**

Please refer to the Lab Worksheet for assistance. (Step 5, Page 6-52).

8. **Delete or comment out the call to clear the CCR0IFG flag.**

We won't need this because the timer will drive the LED directly – that is, no interrupt is required where we need to manually toggle the GPIO with a function call.

```
TIMER_A_clearCaptureCompareInterruptFlag( TIMER_A0_BASE,  
TIMER_A_CAPTURECOMPARE_REGISTER_0 //Clear CCR0IFG  
);
```

Then again, it doesn't hurt anything if you leave it in the code... if so, an unused bit gets cleared.

9. **Make the minor modification to the timer0_isr() as shown in the worksheet.**

Please refer to the Lab Worksheet for assistance. (Step 6, Page 6-53).

'FR5969 users – we only showed the 'F5529 code in the worksheet. Please be careful to not change the interrupt vector or IV register values. That's not what we're asking you to do in this step.

10. **Build the code verifying there are no syntax errors; fix any as needed.**

Debug/Run

11. Launch the debugger and set three breakpoints inside the two ISR's.

- When we run the code, the first breakpoint will indicate if we received the CCR0 interrupt. If we wrote the code properly, we should NOT stop here.
- We should NOT stop at the second breakpoint either. CCR2 was setup to change the Output Signal, not generate an interrupt.
- We should stop at the 3rd breakpoint. We left the timer configured to break whenever TAR rolled-over to zero. (That is, whenever TA0IFG or TA1IFG is set.)

```
97 //*****
98 // Interrupt Service Routines
99 //*****
100 #pragma vector=TIMER0_A0_VECTOR
101 __interrupt void ccr0_ISR (void)
102 {
103     // 4. Timer ISR and vector
104
105     // Toggle the Red LED on/off
106     GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
107 }
108
109 #pragma vector=TIMER0_A1_VECTOR
110 __interrupt void timer0_ISR (void)
111 {
112     // 4. Timer ISR and vector
113
114     switch(__even_in_range( TA0IV, 14 )) {
115         case 0: break;           // None
116         case 2: break;           // CCR1 IFG
117         case 4:                   // CCR2 IFG
118             _no_operation();     // gives us something to set a
119             break;
120         case 6: break;           // CCR3 IFG
121         case 8: break;           // CCR4 IFG
122         case 10: break;          // CCR5 IFG
123         case 12: break;          // CCR6 IFG
124         case 14:                   // TAR overflow
125             // Toggle the Green LED on/off
126             GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN7 );
127             break;
128         default: _never_executed();
129     }
130 }
```

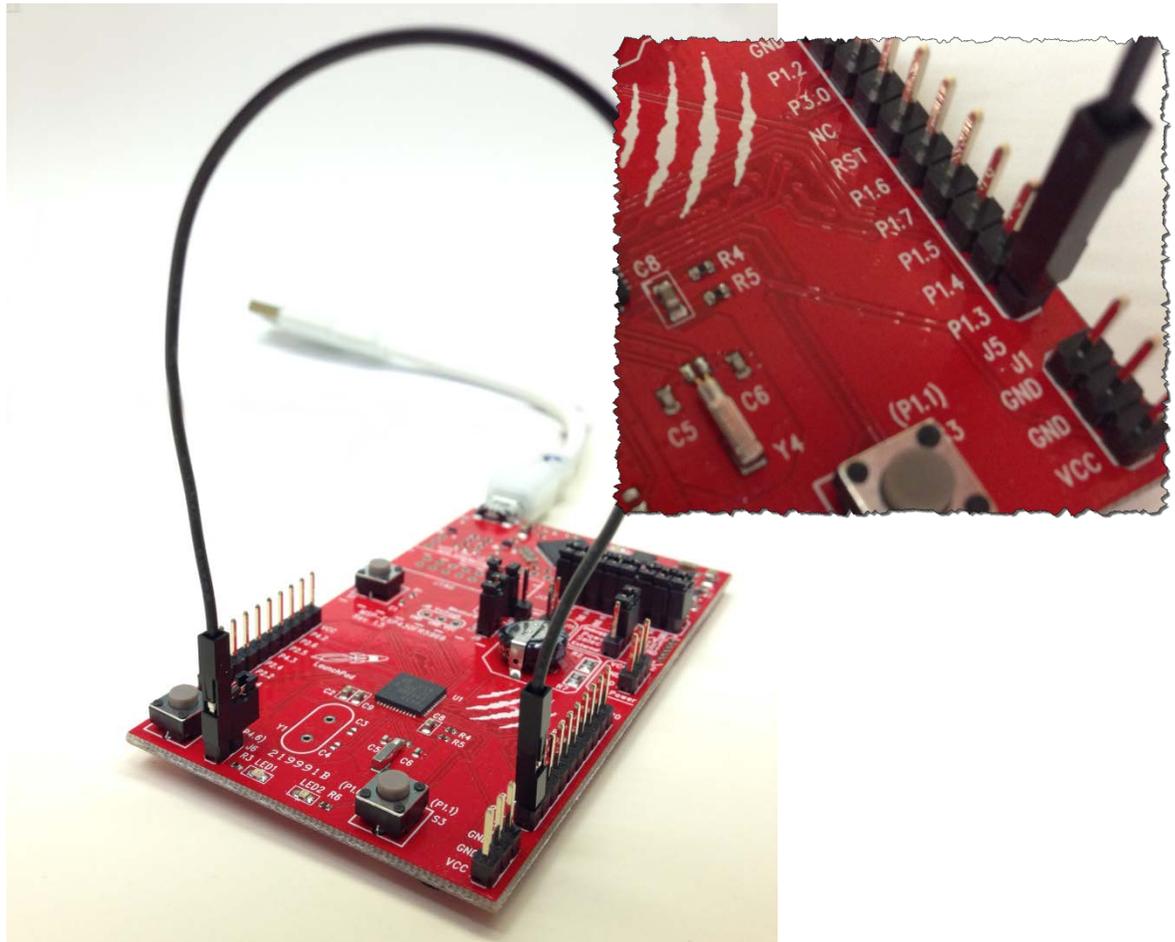


Note: Due to an emulation bug with the beta tools – as we discussed in an earlier lab exercise – terminating, restarting, or resetting the 'FR5969 with two or more breakpoints set may corrupt your program code. If this occurs, load a different program, then reload the current one again.

12. Remove the breakpoints and let the code run. Do both LED's toggle?

Why doesn't the LED1 toggle? _____

The FR5969 looks like this. As you can see below, the connection is very similar to the other Launchpad except the location of the P1.3 pin. In this case, it's in the lower-right-hand corner of the BoosterPack pins.



14. Run your code.

Hopefully both LED's are now blinking. LED1 should toggle first, then the LED2.

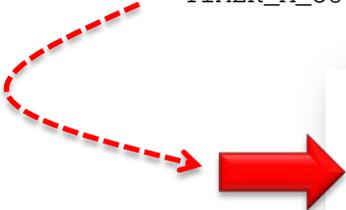
Do they both blink at the same rate? _____

Why is that? _____

15. Terminate the debugger and go back to your `main.c` file.

16. Modify one parameter of the function that configures CCR2, changing it to use the mode:

```
TIMER_A_OUTPUTMODE_SET_RESET
```

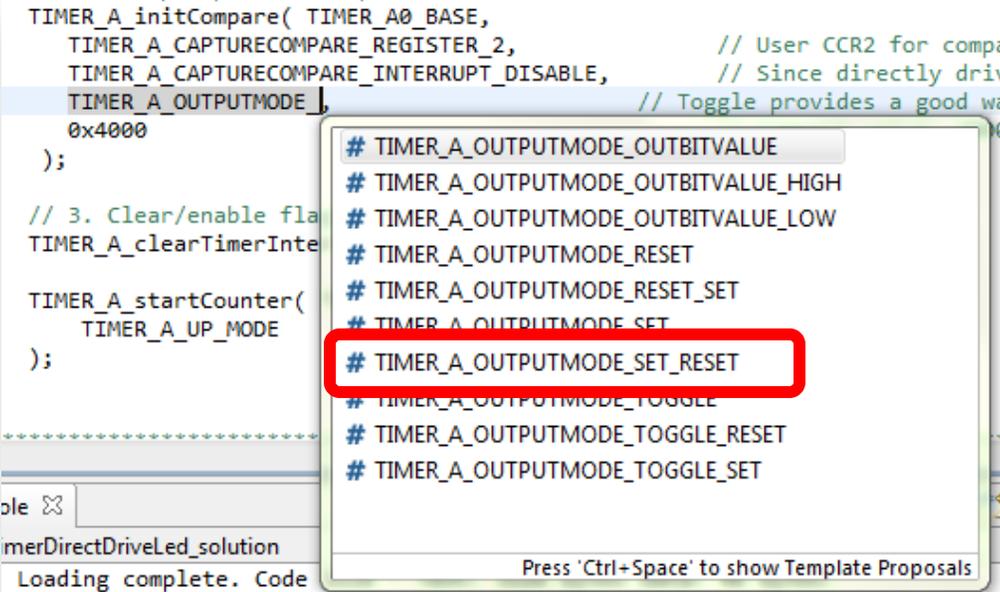


```
TIMER_A_initCompare( TIMER_A0_BASE,  
    TIMER_A_CAPTURECOMPARE_REGISTER_2,  
    TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE,  
    TIMER_A_OUTPUTMODE_SET_RESET,  
    0x4000  
);
```

Hint, if you haven't already tried this trick, delete the last part of the parameter and hit Ctrl_Space:

TIMER_A_OUTPUTMODE_ then hit Control-Space

```
TIMER_A_initCompare( TIMER_A0_BASE,  
    TIMER_A_CAPTURECOMPARE_REGISTER_2,           // User CCR2 for compa  
    TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE,    // Since directly driv  
    TIMER_A_OUTPUTMODE_ ,                        // Toggle provides a good wa  
    0x4000  
);  
  
// 3. Clear/enable fla  
TIMER_A_clearTimerInte  
  
TIMER_A_startCounter(  
    TIMER_A_UP_MODE  
);
```



- # TIMER_A_OUTPUTMODE_OUTBITVALUE
- # TIMER_A_OUTPUTMODE_OUTBITVALUE_HIGH
- # TIMER_A_OUTPUTMODE_OUTBITVALUE_LOW
- # TIMER_A_OUTPUTMODE_RESET
- # TIMER_A_OUTPUTMODE_RESET_SET
- # TIMER_A_OUTPUTMODE_SET
- # **TIMER_A_OUTPUTMODE_SET_RESET**
- # TIMER_A_OUTPUTMODE_TOGGLE
- # TIMER_A_OUTPUTMODE_TOGGLE_RESET
- # TIMER_A_OUTPUTMODE_TOGGLE_SET

Press 'Ctrl+Space' to show Template Proposals

Eclipse will provide the possible variations. Double-click on one (or select one and hit return) to enter it into your code.

17. Build and run your code with the new Output Mode setting.

Do they both blink at the same rate? _____

When using the “TIMER_A_OUTPUTMODE_ **SET_RESET**” output mode ...

If a compare match (TAR = CCR2) causes the output to be SET (i.e. LED goes ON), what causes the RESET (LED going OFF)?

You may want to experiment with a few other output mode settings. It can be fun to see them in action.

18. When done experimenting, close the project.

(Optional) Lab 6c – Portable HAL

Can you create a single timer source file that would work on multiple platforms?

For the most part, “Yes”. This is often done by creating a HAL (hardware abstraction layer). We’ve created a rudimentary HAL version of Lab 6c. You can find this in the solution file:

```
lab_06c_timerHal_solution.zip
```

While the timer file is shared between the two HAL solutions, we didn’t get too carried away. There are a couple of things we didn’t handle; for example, we didn’t do anything with *unused_interrupts.c* and so that needed to be manually edited.

Play with it as you wish...

(Optional) Lab 6d – Simple PWM

While we don't have a complete write-up for our Simple PWM lab exercise, we have created a solution that shows off the `TIMER_A_simplePWM()` DriverLib function. (Hence, the name of the exercise.)

The `lab_06d_simplePWM` project uses the DriverLib function to create a single PWM waveform. As with Lab 6c, the output is routed to LED1 using a jumper wire. By default, it creates a 50% duty cycle ... which means it blinks the light on/off (50% on, 50% off) similar to our previous lab exercises.

We have made one big change, though, by adding two arguments to the `initTimers()` function. These values are the "Period" and "Duty Cycle" values that are passed to the `simplePWM` function. We also rewrote the main `while{}` loop so that it calls `initTimers()` every second.

The purpose of these changes was to allow you the chance to experiment with different Period & Duty Cycle values without having to stop and start the program over-and-over again. The easiest way to do this once the program is running is to:

- Halt (i.e. Suspend) the program
- View the two values in the Expressions watch window
- Change the values, as desired
- Continue running the program – in a second, literally, the values should take effect

If you change the period to something smaller, you won't be able to see the LED going on/off anymore – it will just appear to stay on. At this point, changing the duty cycle will cause the LED to appear bright (or dim).

This is a crude example, using a simple function, but it's something quick and easy to experiment with.

Both `Timer_A` and `Timer_B` peripherals can create a set of complex PWM (pulse-width modulation) waveforms. At this point, this course doesn't go into great details. We highly recommend you review the excellent discussion in John Davies book: ***MSP430 Microcontroller Basics*** by John H. Davies, (ISBN-10 0750682760) [Link](#)

Chapter 6 Appendix

Lab6a Answers

Lab 6a Worksheet (1-2)

Goal: Write a function setting up Timer_A to generate an interrupt every two seconds.

- How many clock cycles does it take for the 16-bit TimerA0 to 'rollover'? (Hint: 16-bit timer)

$2^{16} = 64K$
- If our goal is to generate a two second interrupt rate, what source and divide by value will get us closest to 2 seconds? (see diagram above)

configured for 32KHz / sec

Clock input: (circle one) **ACLK** SMCLK

DriverLib enumeration: TIMER_A_CLOCKSOURCE_ ACLK

Divide by: TIMER_A_CLOCKSOURCE_DIVIDER_ 1 (as we'll see in next)

Hint: Since we are interested in 2 seconds, a slow clock might work best.

Another Hint: Look up the arguments for the `TIMER_A_configureContinuousMode()` function in the *MSP430® Peripheral Driver Library User's Guide*.

Lab 6a Worksheet (3)

- Calculate the Timer frequencies for the clocks & divider values you chose in the previous step.

This lab exercise uses the clock setup from Chapter 4. So you don't have to look it up, we've copied the values into the table below:

Clock	'F5529 Launchpad	'FR5969 Launchpad
ACLK	32 KHz	32 KHz
SMCLK	8 MHz	8 MHz
MCLK	8 MHz	8 MHz

Which clock did you choose in the previous step? Write its frequency below and then calculate the timer rates.

ACLK

Timer Clock Source: _____

Clock Frequency = $\frac{32K / \text{sec}}{\text{cycles/second}}$

Timer Frequency = $\frac{32K / \text{sec}}{\text{clock frequency}} \div \frac{1}{\text{timer clock divider}} = \frac{32KHz / \text{sec}}{\text{timer frequency}}$

Timer Output = $\frac{32K / \text{sec}}{\text{timer frequency}} \div \frac{64K}{\text{counts for timer to rollover}} = \frac{1/2 \text{ sec}}{\text{timer frequency}}$

Lab 6a Worksheet (4-5)

4. Which Timer do you need to use for this lab exercise?

Launchpad	Timer	Short Name	Timer's DriverLib Enum
'F5529	TIMER0_A3	TA0	TIMER_A0_BASE
'FR5969	Timer1_A5	TA1	TIMER_A1_BASE

Pick the one req'd for your board: **AO** or **A1**

Write down the timer enumeration you need to use: **TIMER_** _____ **_BASE**

5. Write the `TIMER_A_configureContinuousMode()` function.

```
TIMER_A_configureContinuousMode(
    TIMER_ ______BASE,           // Which timer to setup?
    TIMER_A_CLOCKSOURCE_ACLK _____, // Timer clock source
    TIMER_A_CLOCKSOURCE_DIVIDER_1 _____, // Timer clock divider
    TIMER_A_TAIE_INTERRUPT_ENABLE _____, // Enable interrupt on TAR counter rollover
    TIMER_A_DO_CLEAR                 // Clear TAR & previous divider state
);
```

Hint: Where do you get help writing this function? We highly recommend the *MSP430ware DriverLib Users Guide*. (See 'docs' folder inside MSP430ware's **driverlib** folder.) Another suggestion would be to examine the header file: (timer_a.h).

Lab 6a Worksheet (7)

7. Complete the code to for the 3rd part of the "Timer Setup Code".

The third part of the timer setup code includes:

- ~~Enable the interrupt (IE) ... we don't have to do this, since it's done by the `TIMER_A_configureContinuousMode()` function (from step5 on page 6-39).~~
- Clear the appropriate interrupt flag (IFG)
- Start the timer

```
// Clear the timer interrupt flag
TIMER_A_clearTimerInterruptFlag ( TIMER_ ______BASE ); //Clear TA0IFG
```

```
// Start the timer
TIMER_A_startCounter _____ ( _____ //Function to start timer
    TIMER_ ______BASE, AO or A1 //Which timer?
    TIMER_A_CONTINUOUS_MODE //Run in Continuous mode
```

Lab 6a Worksheet (8a)

'F5529 Solution

8. Change the following interrupt code to toggle LED2 when Timer_A rolls-over.

a) Fill in the details for your Launchpad.

Port/Pin number for LED2: Port 4, Pin 7

Interrupt Vector: #pragma vector = TIMER0_A1 _VECTOR

Interrupt Vector register: TA0IV
(for example, we used P1IV for GPIO Port 1)

'FR5969 Solution

8. Change the following interrupt code to toggle LED2 when Timer_A rolls-over.

a) Fill in the details for your Launchpad.

Port/Pin number for LED2: Port 1, Pin 0

Interrupt Vector: #pragma vector = TIMER1_A1 _VECTOR

Interrupt Vector register: TA1IV
(for example, we used P1IV for GPIO Port 1)

Lab 6a Worksheet (8b)

b) Here is the interrupt code that exists from a previous exercise, change it as needed:

```

#pragma vector=PORT1_VECTOR TIMER0_A1_VECTOR
__interrupt void pinbutton_ISR timer_ISR (void) or TA0IV (for 'F5529)
{
    switch( __even_in_range( P1IV TA1IV , 14 ) ) {
        case 0: break; // No interrupt
        case 2: break; // Pin 0
        case 4: break; // Pin 1
        GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        break;
        case 6: break; // Pin 2
        case 8: break; // Pin 3
        case 10: break; // Pin 4
        case 12: break; // Pin 5
        case 14: GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        break; // Pin 6
        case 16: break; // Pin 7
        default: _never_executed();
    }
}

```

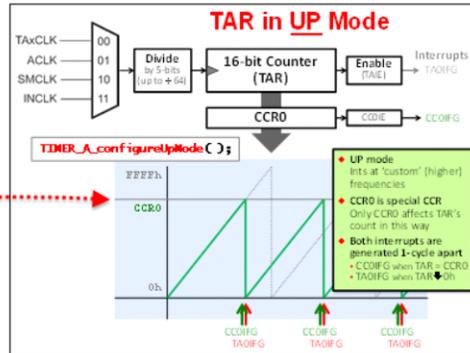
or for the 'F5529:
GPIO_toggleOutputOnPin(GPIO_PORT_P4, GPIO_PIN7);

Lab6b Answers

Lab 6b Worksheet (1)

1. Calculate the timer period that will go into CCR0 to set the proper interrupt rate.

Here's a quick review from our discussion.



Timer_A's counter (TAR) will count up until it reaches the value in the CCR0 capture register, then reset back to zero. What value do we need to set CCR0 to get a 1/2 second interval?

$$\begin{aligned} \text{Timer Frequency} &= \frac{32 \text{ KHz}}{\text{clock frequency}} \div \frac{1}{\text{timer clock divider}} = 32 \text{ KHz} \\ \text{Timer Output} &= \frac{32 \text{ KHz}}{\text{timer frequency}} \div \frac{0x8000}{\text{timer period (i.e. CCR0 value)}} = \frac{1}{2} \text{ SECOND} \end{aligned}$$

0xFFFF / 2

Lab 6b Worksheet (2)

2. Complete the `TIMER_A_configureUpMode()` function?

This function will replace the `TIMER_A_configureContinuousMode()` call we made in our previous lab exercise.

Hint: Where to get help for writing this function? Once again, we recommend the MSP430ware DriverLib users guide ("docs" folder inside MPS430ware's DriverLib).

Another suggestion would be to examine the `timer_a.h` header file.

```
TIMER_A_configureUpMode(
    TIMER__BASE, // AO or A1 // Which timer are you using?
    TIMER_A_CLOCKSOURCE_ACLK, // Timer clock source
    TIMER_A_CLOCKSOURCE_DIVIDER_1, // Timer clock divider
    0xFFFF / 2, // Period (calculated in previous question)
    TIMER_A_TAIE_INTERRUPT_ENABLE, // Enable interrupt on TAR counter rollover
    TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE, // Enable CCR0 compare interrupt
    TIMER_A_DO_CLEAR // Clear TAR & previous divider state
);
```

Lab 6b Worksheet (3)

3. Modify your previous code. We need to clear both interrupts and start the timer.

We copied the code from the previous lab into this question. It needs to be modified to meet our new objectives for this lab.

Here are some hints:

- Add an extra line of code to clear the CCR0 flag (we left a blank space below for this)
- Don't make the mistake we made ... look very carefully at the 'start' function. Is there anything that needs to change in that function call?

```
// Clear the timer flag and start the timer
TIMER_A_clearTimerInterruptFlag( TIMER_  BASE );           // Clear TA0IFG
TIMER_A_clearCaptureCompareInterruptFlag (                // Clear CCR0IFG
    TIMER_  BASE,
    TIMER_A_CAPTURECOMPARE_REGISTER_0 );
TIMER_A_startCounter( TIMER_  BASE,                       // Start timer in
    TIMER_A_  UP  MODE );                                 //  UP  mode
```

AO or A1

Lab 6b Worksheet (4)

4. Add a second ISR to toggle the LED1 whenever the CCR0 interrupt fires.

P4.6 (for 'FR5969')

On your Launchpad, what Port/Pin number does the LED1 use? P1.0 (for 'F5529')

Here we've given you a bit of code to get you started:

```
#pragma vector=  TIMER1_A0_VECTOR  (or TIMER1_A0_VECTOR for 'F5529')
__interrupt void ccr0_ISR (void)
{
    // Toggle the LED1 on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P___, GPIO_PIN___ );
}
```

Reflects the value from above

Lab 6b : Lab Debrief

Debug/Run

Follow the same basic steps as found in the previous lab for debugging.

10. Launch the debugger and set a breakpoint inside the both ISR's.

11. Run your code.

If all worked well, when the counter rolled over to zero, an interrupt should occur. Actually, two interrupts should occur. Once you reach the first breakpoint, resume running your code and you should reach the other ISR.

Which ISR was reached first? **LED1 then LED2**

Why? **Because the CCR0 interrupt occurs before the TAIFG interrupt**

This is shown on the slide entitled "TAR in UP Mode". Since they occur at nearly the same instant in time, you have to set breakpoints in order to see that LED1 happens before LED2.

Lab6c Answers

Lab 6c Worksheet (1)

1. Figure out which BoosterPack pin to drive with the timer's output

a) What GPIO port/pin output is TA0.2 (or TA1.2) combined with?

Hint: There are a couple places in the datasheet to find this info. I recommend searching your device's datasheet for "TA0.2"

'F5529	P1.0/TA0CLK/ACLK	<input type="checkbox"/>	21
	P1.1/TA0.0	<input type="checkbox"/>	22
	P1.2/TA0.1	<input type="checkbox"/>	23
	P1.3/TA0.2	<input checked="" type="checkbox"/>	24
	P1.4/TA0.3	<input type="checkbox"/>	25
	P1.5/TA0.4	<input type="checkbox"/>	26

GPIO port/pin: **P1.3**

b) Next, what BoosterPack pin is this GPIO connected to?

BoosterPack pin: **see photo**

'FR5969

P1.3/TA1.2/UCB0STE/A3/C3	<input type="checkbox"/>	9
--------------------------	--------------------------	---

Lab 6c Worksheet (2)

2. Write the function to set this Pin/Port to be used as a timer pin (as opposed to an output pin).

F5529 'F5529 Users, here's the function you need to complete:

```
GPIO_setAsPeripheralModuleFunctionOutputPin (
    GPIO_PORT_P1,
    GPIO_PIN3
);
```

FR5969 'FR5969 Users, your function requires one more argument:

```
GPIO_setAsPeripheralModuleFunctionOutputPin (
    GPIO_PORT_P1,
    GPIO_PIN3,
    GPIO_PRIMARY_MODULE_FUNCTION
);
```

Lab 6c Worksheet (3)

3. Modify the `TIMER_A_configureUpMode()` function?

Here is the code we wrote for the previous lab exercise. We only need to make one change to the code. Since we will drive the signal directly from the timer, we don't need to generate the CCR0 interrupt anymore.

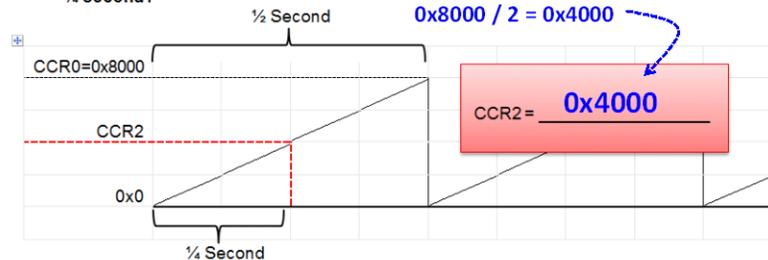
Mark up the code below to disable the interrupt. (We'll bet you can make this change without even looking at the API documentation. Intuitive code is one of the benefits of using *DriverLib!*)

```
TIMER_A_configureUpMode(
    TIMER__BASE, // Which timer are you using
    TIMER_A_CLOCKSOURCE_ACLK, // Timer clock source
    TIMER_A_CLOCKSOURCE_DIVIDER_1, // Timer clock divider
    0xFFFF / 2, // Period: (0x8000) / 32Khz = 1/2 sec
    TIMER_A_TAIE_INTERRUPT_ENABLE, // Enable interrupt on TAR counter rollover
    TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE, // Enable CCR0 compare interrupt
    TIMER_A_DO_CLEAR // Clear TAR & previous divider state
);
```

We changed 'ENABLE' to 'DISABLE'

Lab 6c Worksheet (4-5)

4. What 'compare' value does CCR2 need to equal in order to toggle the output signal at ¼ second?



5. Add a new function call to setup Capture and Compare Register 2 (CCR2). This should be added to `initTimers()`.

CCR2 value calculated above goes here

```
TIMER_A_init Compare (
    TIMER__BASE, // Which timer are you using?
    TIMER_A_CAPTURECOMPARE_REGISTER_2, // Select the CCR2 register
    TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE, // Disable int; since driving LED directly
    TIMER_A_OUTPUTMODE_TOGGLE, // Toggle mode creates on/off signal
    0x4000, // Compare value to toggle at 1/4 second
);
```

Lab 6c Worksheet (6)

6. Compare your previous code to that below.

What did we change? **Added `_no_operation()` – something to breakpoint on**

```
#pragma vector=TIMER0_A1_VECTOR
__interrupt void timer0_ISR(void)
{
    switch(_even_in_range( TA0IV, 14 )) {
        case 0: break;           // No interrupt
        case 2: break;           // CCR1 IFG
        case 4: break;           // CCR2 IFG
                _no_operation();
                break;
        case 6: break;           // CCR3 IFG
        case 8: break;           // CCR4 IFG
        case 10: break;          // CCR5 IFG
        case 12: break;          // CCR6 IFG
        case 14: break;          // TAR overflow
    }
}
```

During debug, we will ask you to set a breakpoint on 'case 4'.

Why should case 4 not occur, and thus, the breakpoint never reached?

def:

TIMER A CAPTURECOMPARE INTERRUPT DISABLE,

We disabled the INT because we're driving the signal directly to the pin

Lab 6c Worksheet (7)

7. Why is better to toggle the LED directly from the timer, as opposed to using an interrupt (as we've done in the previous lab exercises)?

◆ **Lower Power:**

When the Timer drives the pin; no need to wake up the CPU. (Either that, or it leaves the CPU free for other processing.)

◆ **Less Latency:**

When the CPU toggles the pin, there is a slight delay that occurs since the CPU must be interrupted, then go run the ISR.

◆ **More Deterministic:**

The delay caused by generating/responding to the interrupt may vary slightly. This could be due to another interrupt being processed (or a higher priority interrupt occurring simultaneously). Directly driving the output removes the variance and makes it easy to "determine" the time that the output will change!

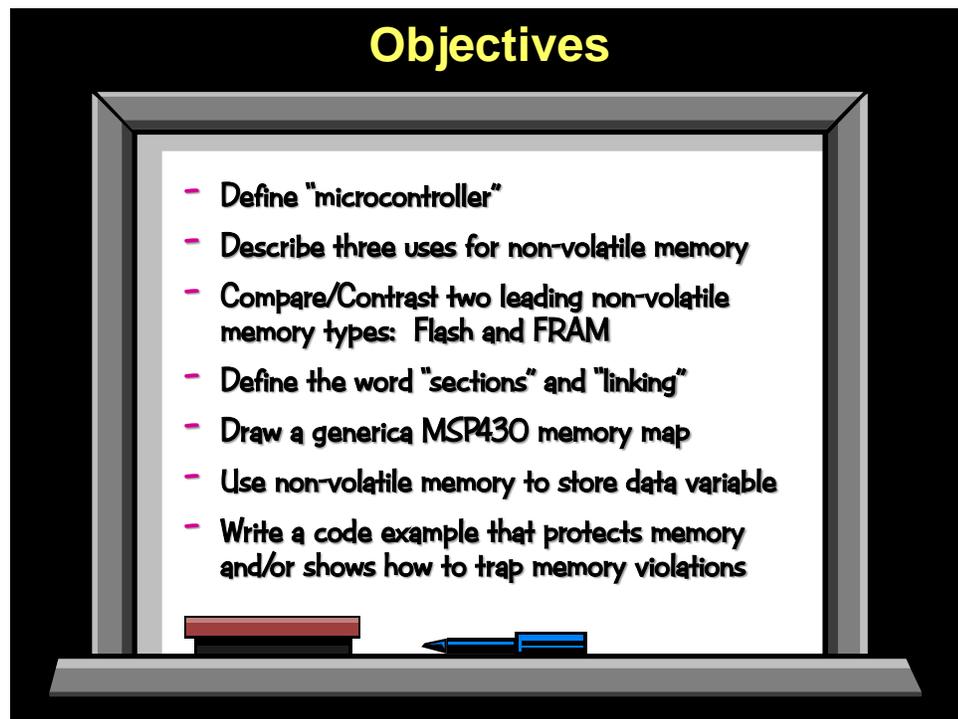
Non-Volatile Memory: Flash & FRAM

Introduction

Replace this text with your introduction

Learning Objectives

Replace this text with module learning objectives (or slide)...



Chapter Topics

Non-Volatile Memory: Flash & FRAM	7-1
<i>What is a Microcontroller?</i>	7-3
<i>Non-Volatile Memory: Flash & FRAM</i>	7-4
<i>Memory Maps & Linking</i>	7-8
Memory Maps	7-8
Sections.....	7-9
Linking.....	7-10
Custom Sections	7-11
<i>Using Flash</i>	7-12
<i>Using FRAM (and the MPU)</i>	7-14
FRAM Controller	7-14
Memory Protection Unit (MPU)	7-16
FRAM Code Example	7-18
<i>Lab Exercises</i>	7-21
lab_07a_infoB	7-22
('F5529 Optional) lab_07a_low_wear_flash	7-26
('FR5969) lab_07b_protection	7-26

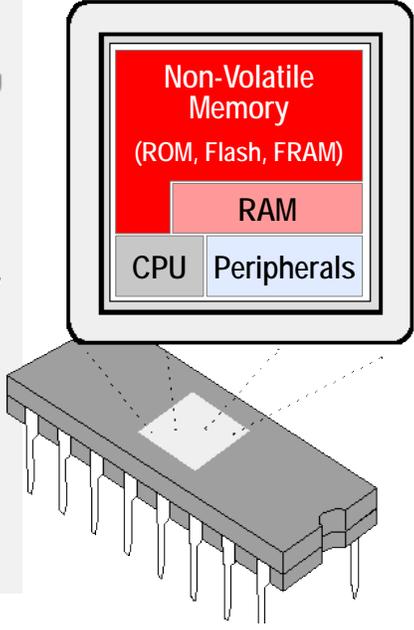
Prerequisites & Tools

◆ Skills	Chapter
◆ Creating a CCS Project for MSP430 Launchpad(s)	(Ch 2)
◆ Basic knowledge of:	
◆ C language	
◆ Using a C libraries and header files (MSP430ware DriverLib)	(Ch 3)
◆ Using printf()	(Ch 2)
◆ GPIO for LED's and Buttons	(Ch 3)
◆ Interrupts	(Ch 5)
◆ Hardware	
◆ Windows (XP, 7, 8) PC with available USB port	
◆ MSP430F5529 Launchpad (with included USB micro cable)	
◆ Software	
◆ CCSv5.5 or CCSv6	
◆ MSP430ware_1_70_00_28	

What is a Microcontroller?

What is a Microcontroller?

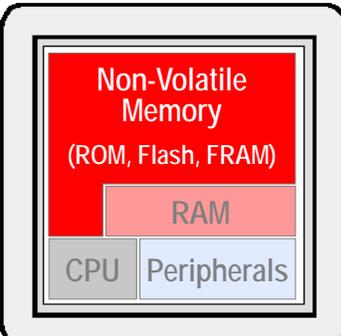
- ◆ Wikipedia defines Microcontroller as:
A microcontroller (μ C, uC or MCU) is a small computer on a single integrated circuit containing a **processor core (CPU)**, **memory**, and **programmable input/output peripherals**[†]
- ◆ By strict definitions...
 - ◆ Microprocessors (MPU) only contain a CPU*
 - ◆ MCU's add the components needed to create a full system on a chip
- ◆ Early MCU's used factory-programmed Read-Only Memory (ROM) to hold program instructions; today's MCU's utilize in-system programmable Flash and FRAM technologies
- ◆ MCU's today are often predominated by memory area – though most development work is centered around programming the CPU
- ◆ U.S. Patent 3,757,306:
Texas Instruments... engineers Gary Boone and Michael Cochran succeeded in creating the first microcontroller... in 1971.*



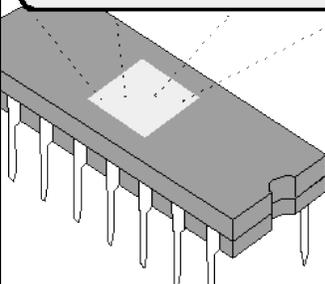
† <http://en.wikipedia.org/wiki/Microprocessor>
 * <http://en.wikipedia.org/wiki/Microcontroller>
 * <http://smithsonianchips.si.edu/augarten/p38.htm>

Non-Volatile Memory: Flash & FRAM

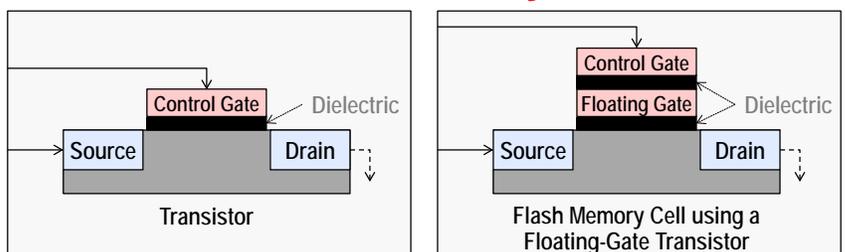
What is Non-Volatile Memory?



- ◆ Non-Volatile Memory (NVM) retains its information when powered down
- ◆ By contrast, Random Access Memory (RAM) needs power to keep its information
- ◆ NVM examples include: (MSP430 devices only use Flash or FRAM)
 - ◆ Flash Memory
 - ◆ FRAM (ferroelectric RAM)
 - ◆ ROM (read-only memory)
 - ◆ EEPROM (electrically erasable ROM)
 - ◆ Hard disk drives
- ◆ Flash & FRAM are in-system programmable, which means a program can rewrite them
- ◆ Typical MCU applications use:
 - ◆ NVM for program and calibration data
 - ◆ RAM for variables, stack and heap



How Flash Memory Works

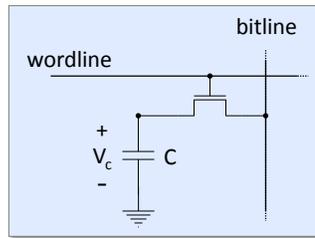


- ◆ In this simplified example of a Flash Memory cell, the addition of a floating gate makes the cell "sticky"
- ◆ Dielectric provides insulation allowing the floating gate to remain charged (or not) for a very, very long time
- ◆ Overcoming the dielectric to erase/charge the floating gate requires a high voltage (~14 Volts); Flash-based processors contain charge pumps to reach these high voltages
- ◆ You must erase a Flash cell before it can be programmed; most Flash memory implementations require an entire block to be erased at one time

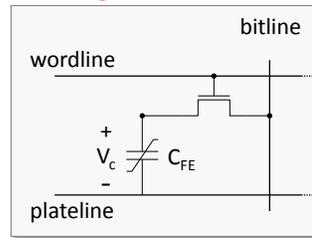
References:

- ◆ *MSP430 Flash Memory Characteristics* by Peter Forstner ([SLAA334A.pdf](http://www.ti.com/lit/slaa334a))
- ◆ EE 216: Principles and Models of Semiconductor Devices by Chintan Hossain at Stanford University <http://www.youtube.com/watch?v=s7JLXs5es7I>

How FRAM Memory Works

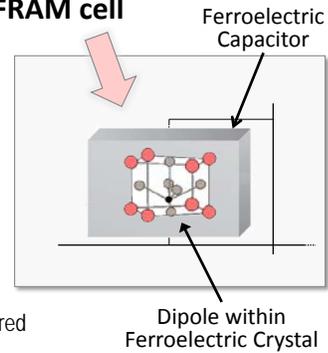


DRAM cell



FRAM cell

- ◆ Ferroelectric RAM (FRAM) is similar to Dynamic RAM (DRAM) – except for using the ferroelectric capacitor which has a crystalline structure with an atom at the center
- ◆ A field applied to the crystal moves the atom in the direction of the field – into an up or down state; the resulting ferroelectric hysteresis loop waveform is key to its operation
- ◆ FRAM (aka Fe-RAM) does not contain element “Fe” (Iron)
- ◆ Also like DRAMs, reads are destructive; though FRAM implementations always write back the original bit values
- ◆ Reads and writes only requires 1.5V – no charge pump required



References:

- FRAM for Dummies - <http://www.edn.com/design/systems-design/4394387/FRAM-MCUs-For-Dummies--Part-1>
- <http://www.radio-electronics.com/info/data/semicond/memory/fram-ferroelectric-random-access-memory-technology-operation-theory.php>

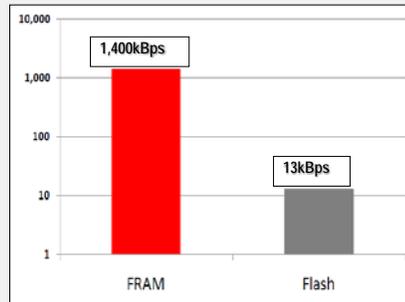
Comparison of Non-Volatile Memory

	FRAM	SRAM	Flash	EEPROM
Non-Volatile Retains data without power	Yes	No	Yes	Yes
Avg Active Power ($\mu\text{A}/\text{MHz}$)	< 100	< 60	260	50,000+
Write Power for 12KB/s	9 μA	N/A	2200 μA	N/A
Write Speeds (13KB)	10 ms	< 10 ms	1 sec	2 secs
Write Endurance	10^{15}	Unlimited	10^5	10^5
Bit-wise Programmable	Yes	Yes	No	No
Data Erase Required	No	No	Segment	Page
Unified: Code and Data	Yes	No	No	No
Read Speeds	8 MHz	up to 25MHz		N/A

FRAM = Ultra-Fast Writes

- Case Example: MSP430FR5739 vs. MSP430F2274
- Both devices use System clock = 8MHz
- Maximum Speed FRAM = 1.4MBps [100x faster]
- Maximum Speed Flash = 13kBps

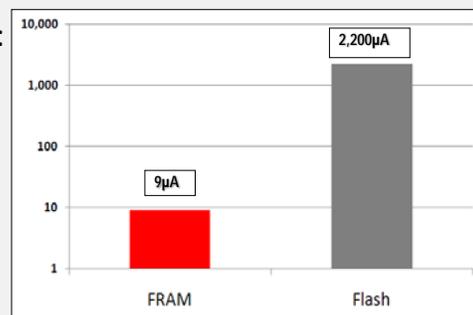
Max. Throughput:



FRAM = Low Active Write Duty Cycle

- Use Case Example: MSP430FR5739 vs. MSP430F2274
- Both devices write to NV memory @ 13kBps
- FRAM remains in standby for 99% of the time
- Power savings: >200x of flash

Consumption @ 13kBps:



FRAM = Ultra-Low Power

- Use Case Example: MSP430FR5739 vs. MSP430F2274
- Average power FRAM = 720µA @ 1400kBps
- Average power Flash = 2200µA @ 13kBps
- 100 times faster using half the power
- Enables more unique energy sources
- FRAM = Non-blocking writes
 - CPU is not held
 - Interrupts allowed

Memory Type	Data Throughput (kBps)	Average Power Consumption (µA)
FRAM	1400	720
Flash	13	2200

FRAM = High Endurance

- Use Case Example: MSP430FR5739 vs. MSP430F2274
- FRAM Endurance >= 100 Trillion [10^{14}]
- Flash Endurance < 100,000 [10^5]
- Comparison: write to a 512 byte memory block @ a speed of 12kBps
 - Flash = 6 minutes
 - FRAM = 100+ years

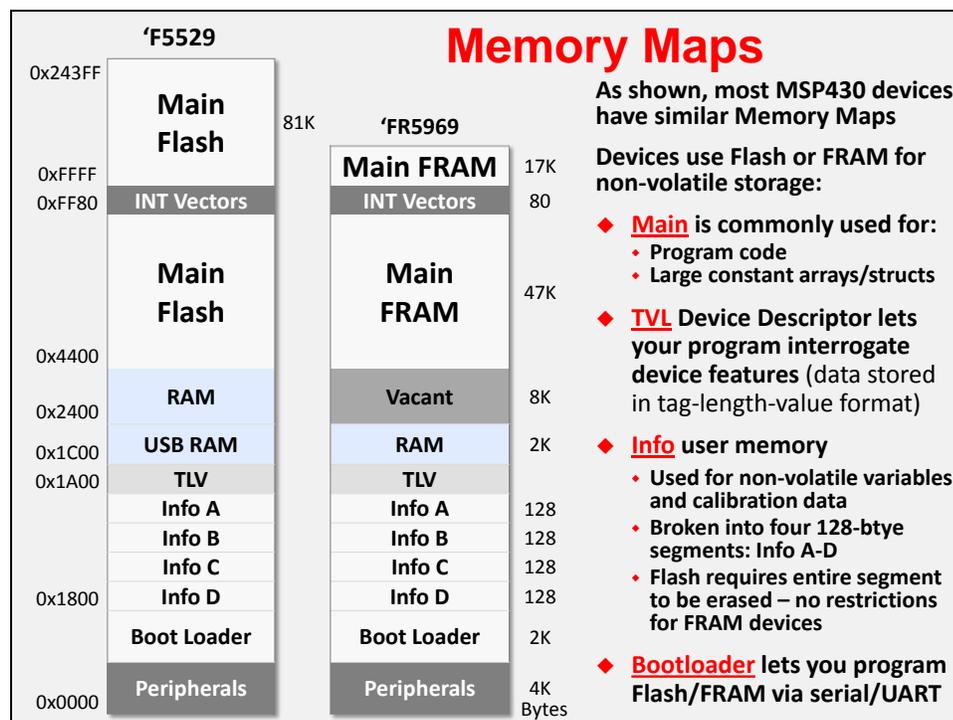
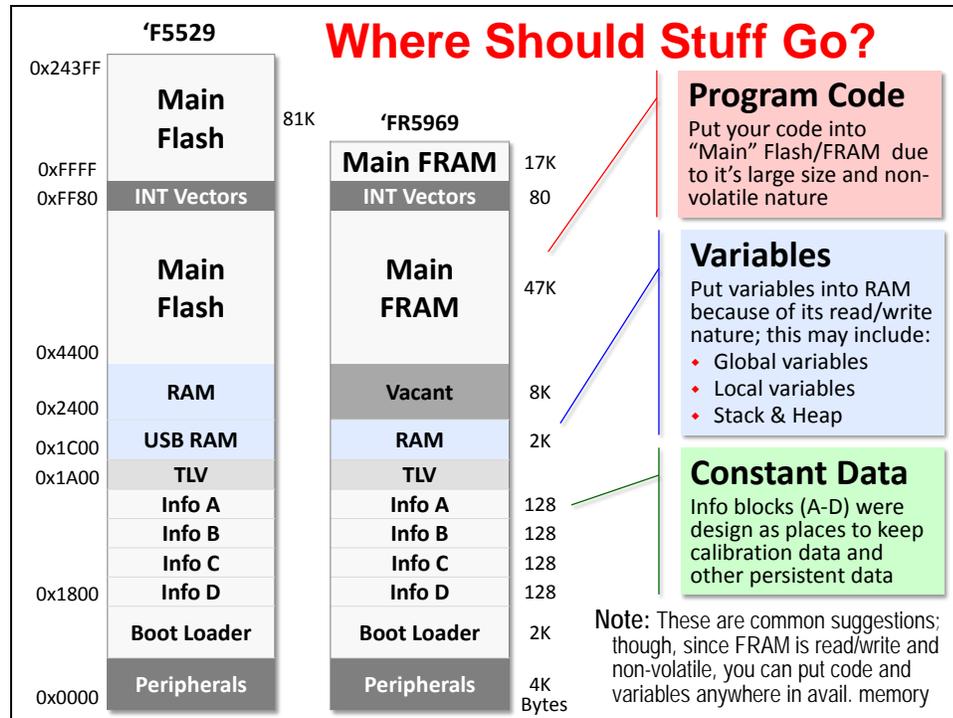
Memory Type	Endurance
FRAM	114,000 years
Flash	6 minutes

FRAM Benefits --- Example App's

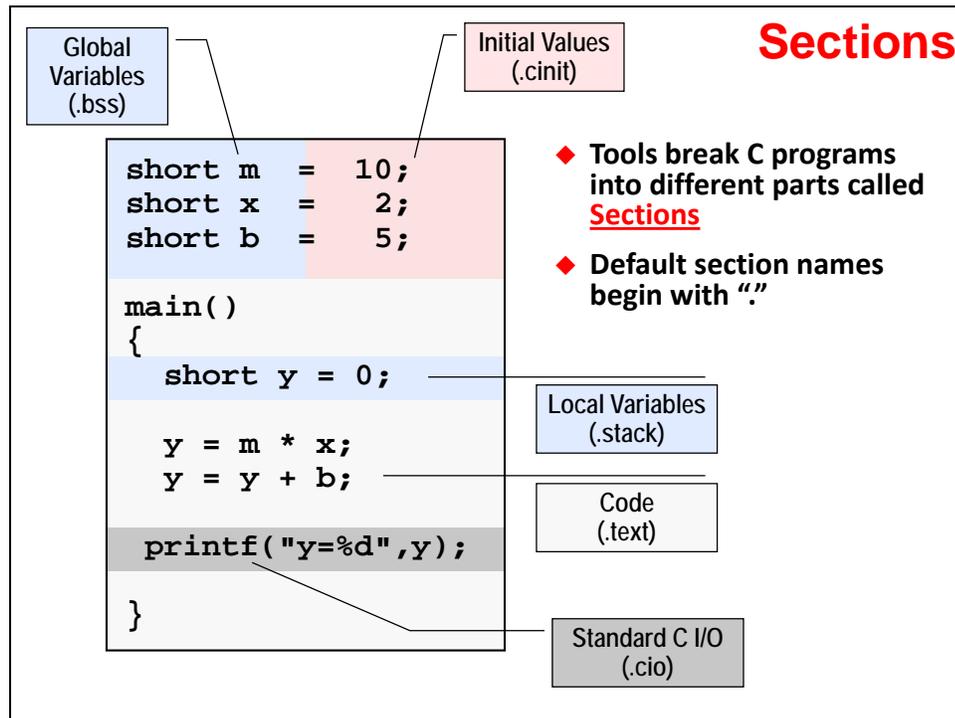
<ul style="list-style-type: none"> ● Non-Volatile <ul style="list-style-type: none"> • Retains data without power ● Fast Write / Update <ul style="list-style-type: none"> • RAM like performance. • Up to ~ 50ns/byte access times today (> 1000x faster than Flash/EEPROM) ● Low Power <ul style="list-style-type: none"> • FRAM only needs 1.5V for writes versus Flash/EEPROM >10-14V • No charge pump needed for FRAM! ● High Write Endurance <ul style="list-style-type: none"> • 100 Trillion read/write cycles ● Superior Data Reliability <ul style="list-style-type: none"> • 'Write Guarantee' in case of power loss 	<ul style="list-style-type: none"> Data logging & remote sensor applications Digital Rights Management (DRM) Low Power Applications (e.g. Mobile & Consumer products) Energy Harvesting (especially wireless) Battery-Backed SRAM Replacement
---	--

Memory Maps & Linking

Memory Maps



Sections

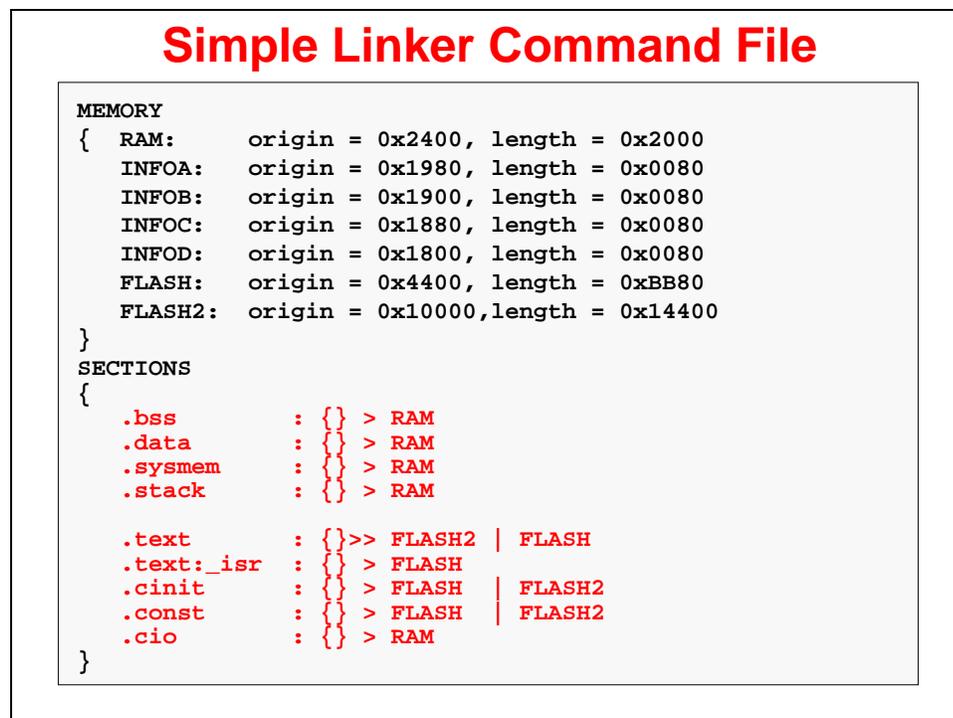
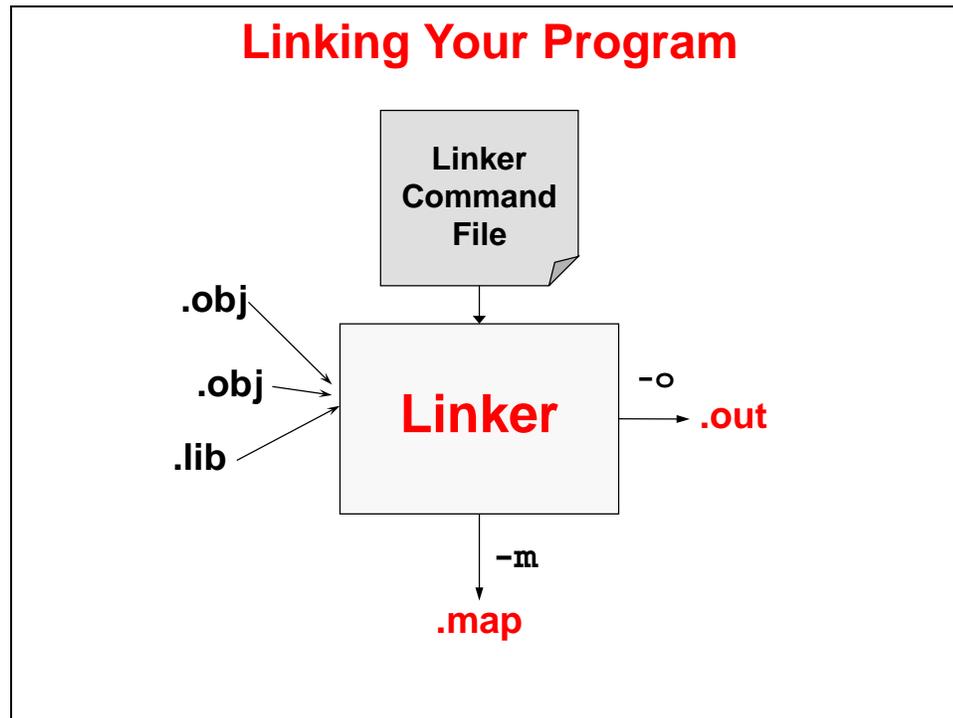


Common Sections Created by TI Compiler

Section Name	Description	Memory Type
.text	Code	Non-Volatile
.data	Global and static non-const variables that are explicitly initialized	Non-Volatile
.rodata	Global and static variables that have const qualifiers	Non-Volatile
.binit	Boot-time copy tables	Non-Volatile
.cinit	Initial values for global/static vars	Non-Volatile
.bss	Global and static variables	Uninitialized
.stack	Stack (local variables)	Uninitialized
.system	Memory for malloc fcns (heap)	Uninitialized
.cio	Buffers for stdio functions	Uninitialized

For more details, see *MSP430 Optimizing C/C++ Compiler User's Guide* (pg. 110 - slau132h.pdf)

Linking



Custom Sections

Create Custom Sections

- ◆ Create custom code section using a pragma:

```
#pragma CODE_SECTION(dotp, "critical");
int dotp(a, x)
```

... or create a sub-section:

```
#pragma CODE_SECTION(ctrl, ".text:_ctrl");
```

- ◆ There's a data section pragma, as well:

```
#pragma DATA_SECTION (x, "myVar");
#pragma DATA_SECTION (y, "myVar");
int x[32];
short y;
```

** Also, look for the SET_CODE_SECTION and SET_DATA_SECTION pragmas in the compiler user's guide*

CMD File with Custom Sections

```
MEMORY
{
  RAM:      origin = 0x2400, length = 0x2000
  INFOA:   origin = 0x1980, length = 0x0080
  INFOB:   origin = 0x1900, length = 0x0080
  INFOC:   origin = 0x1880, length = 0x0080
  INFOD:   origin = 0x1800, length = 0x0080
  FLASH:   origin = 0x4400, length = 0x0080
  FLASH2:  origin = 0x10000, length = 0x0080
}

SECTIONS
{
  dotp      : { } > 0x2400
  .bss     : { } > RAM
  .data    : { } > RAM
  .system : { } > RAM
  .stack   : { } > RAM
  myVar    : { } > FLASH

  .text    : { } >> FLASH2 | FLASH
  .text:_isr : { } > FLASH
  ctrl_code : { } > FLASH
  .cinit   : { } > FLASH | FLASH2
  .const   : { } > FLASH | FLASH2
  .cio     : { } > RAM }
```

- ◆ Custom sections allow you to place code:

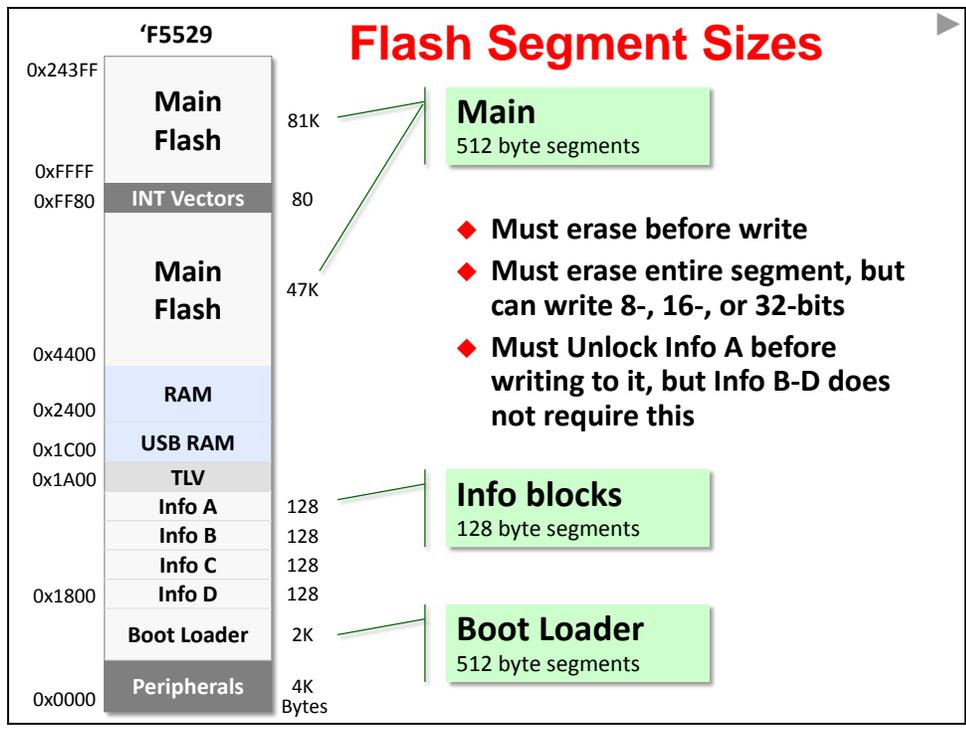
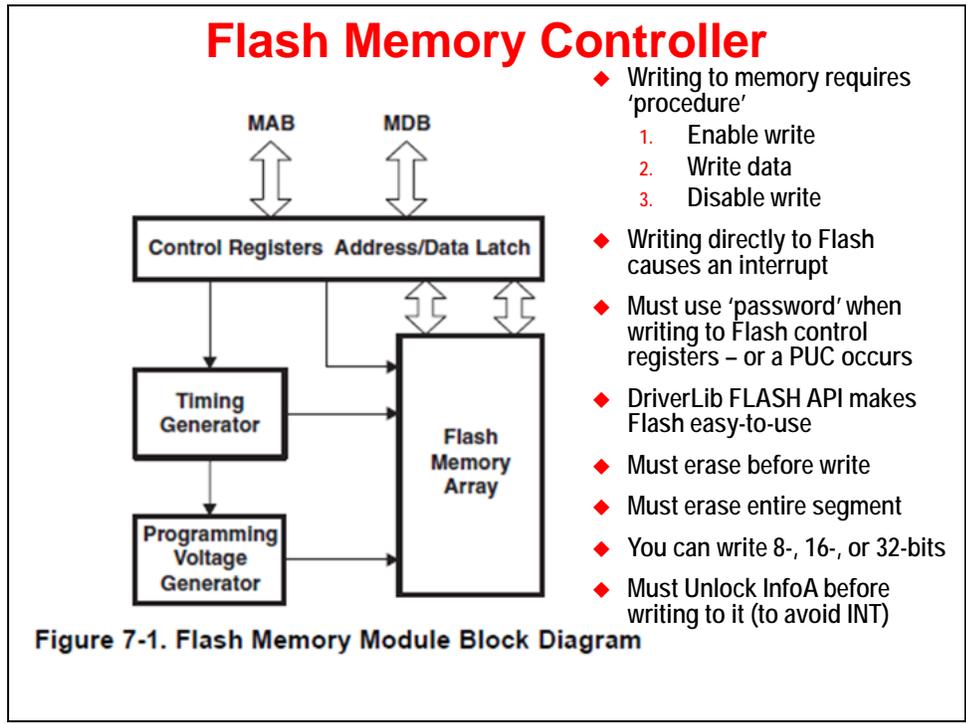
- ◆ At specific locations
- ◆ In a specific order

- ◆ Sub-sections allow you to specify the sub-sect's location

... or if not specified, it is linked along with the full section (i.e. ".text")

- ◆ This is a contrived example to show the mechanism; we'll see 'real' examples later in the chapter

Using Flash



FLASH API

The flash erase operations are managed by

- FLASH_segmentErase()
- FLASH_eraseCheck()
- FLASH_bankErase()

Flash writes are managed by

- FLASH_write8()
- FLASH_write16()
- FLASH_write32()
- FLASH_memoryFill32()

The status is given by

- FLASH_status()
- FLASH_eraseCheck()

The segment A of information memory lock/unlock

- FLASH_lockInfoA()
- FLASH_unlockInfoA()

- ◆ Writing to memory requires 'procedure'
 1. Enable write
 2. Write data
 3. Disable write
- ◆ Writing directly to Flash causes an interrupt
- ◆ Must use 'password' when writing to Flash control registers – or a PUC occurs
- ◆ DriverLib FLASH API makes Flash easy-to-use
- ◆ Must erase before write
- ◆ Must erase entire segment
- ◆ You can write 8-, 16-, or 32-bits
- ◆ Must Unlock InfoA before writing to it (to avoid INT)

Code Example: Writing to “Info A”

```
#pragma DATA_SECTION (calibration_data_char, ".infoA")
uint8_t calibration_data_char[16] = { 0x00,0x01,0x02,...};
uint16_t status;

// Unlock Info Segment A
FLASH_unlockInfoA();

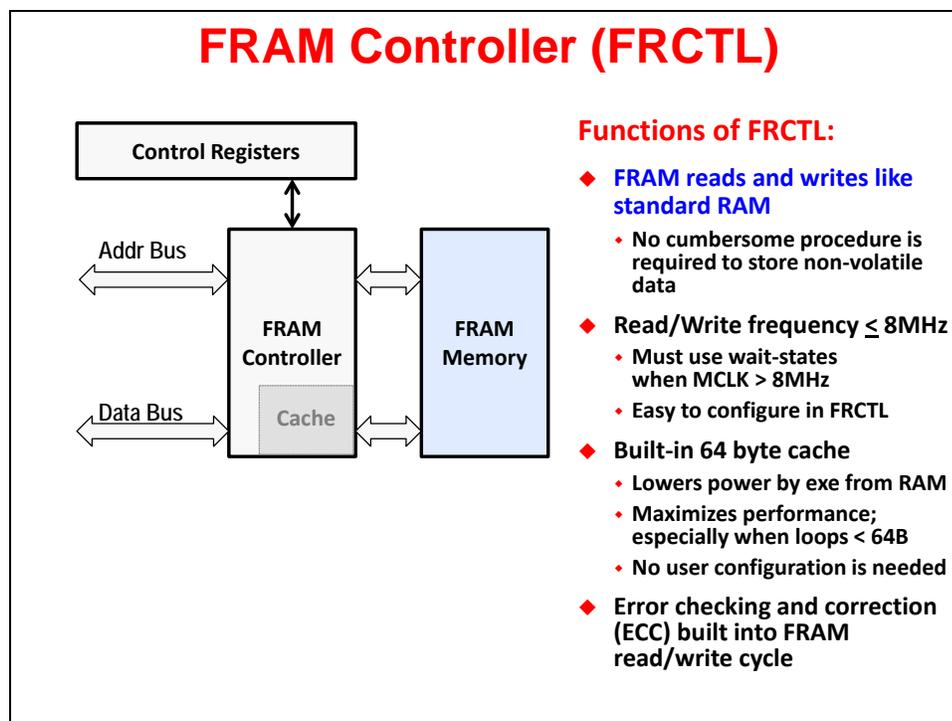
do { // Erase INFOA
    FLASH_segmentErase( (uint8_t*)INFOA_START );
    status = FLASH_eraseCheck( (uint8_t*)INFOA_START, 128 );
} while ( status == STATUS_FAIL );

// Write calibration data to INFOA
FLASH_write8( calibration_data_char,
              (uint8_t*)INFOA_START, 16 );

// Lock Info Segment A
FLASH_lockInfoA();
```

Using FRAM (and the MPU)

FRAM Controller

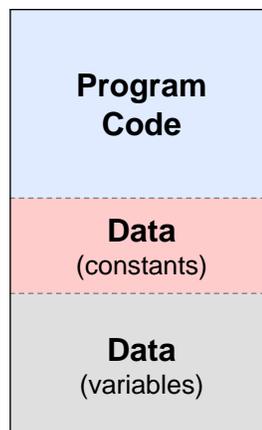


Unified Memory : Code and Data

FRAM Advantages:

- ◆ Mix & Match FRAM for code and/or data
- ◆ Easy to read & write
- ◆ Fast
- ◆ High Endurance
- ◆ Non-volatile
- ◆ Very low power

FRAM



Unified Memory : What Could Happen?

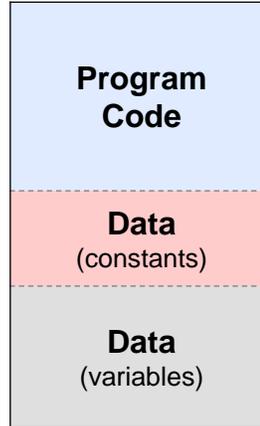
FRAM Advantages:

- Mix & Match FRAM for code and/or data
- Easy to read & write
- Fast
- High Endurance
- Non-volatile
- Very low power

Potential ERROR:

- What if your program code accidentally wrote over itself?
- What if the stack overflows?

FRAM



Unified Memory : What Could Happen?

FRAM Advantages:

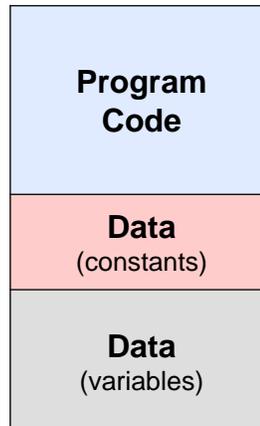
- Mix & Match FRAM for code and/or data
- Easy to read & write
- Fast
- High Endurance
- Non-volatile
- Very low power

Potential ERROR:

- What if your program code accidentally wrote over itself?
- What if the stack overflows?



FRAM

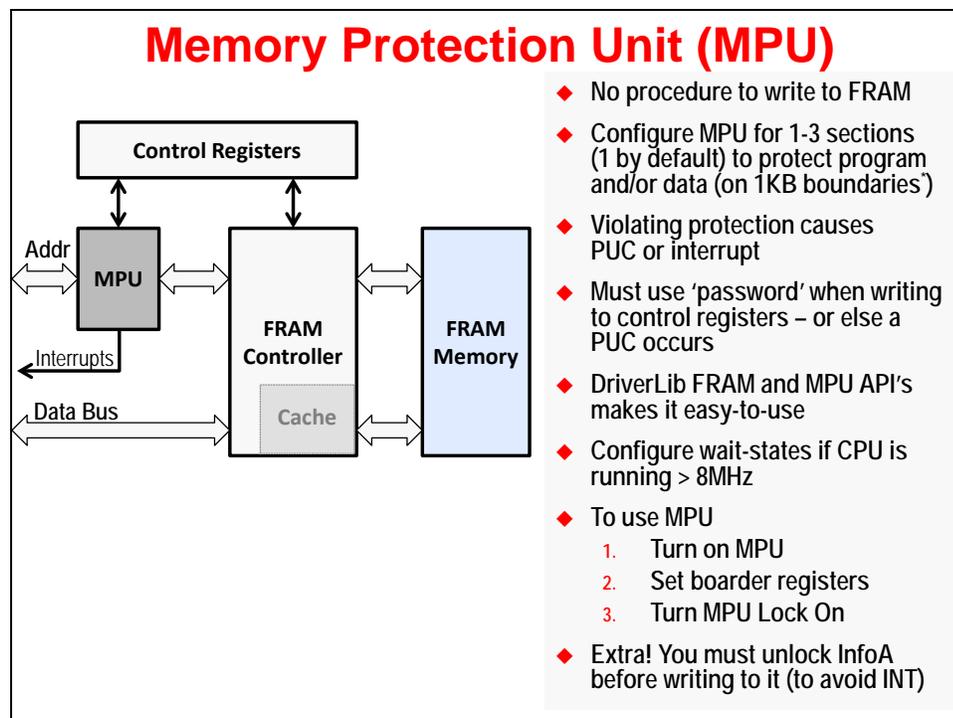
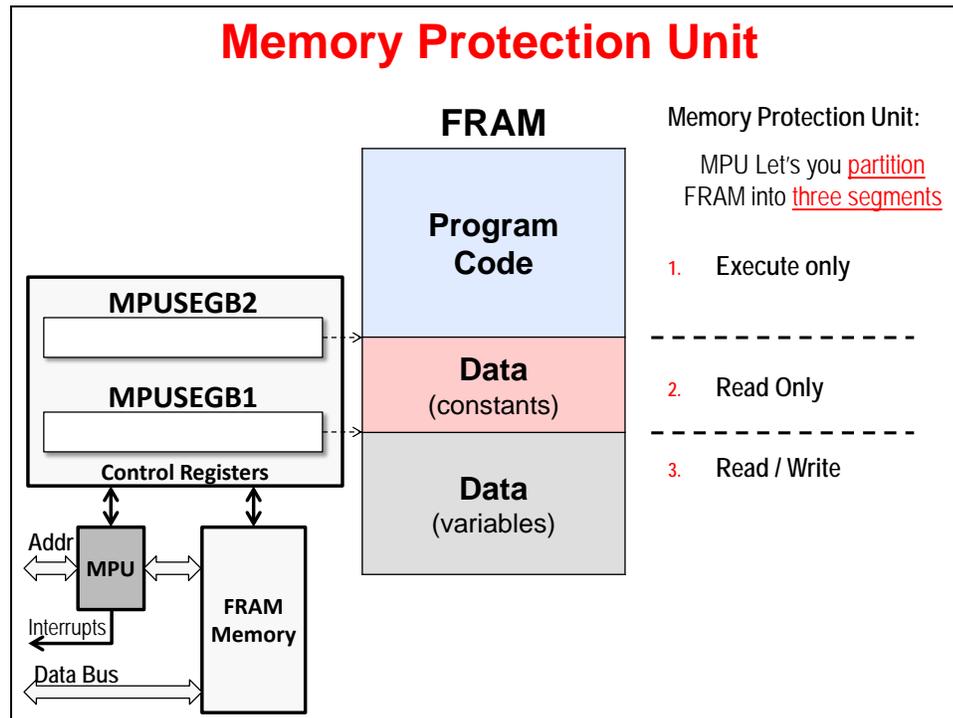


Memory Protection Unit:

MPU Let's you partition FRAM into three segments

1. Execute only
2. Read Only
3. Read / Write

Memory Protection Unit (MPU)



MPU Settings via GUI (CCSv6 only)

Properties for emptyProject

type filter text

Resource
General
Build
MSP430 Compiler
Processor Options

General

Configuration: Debug [Active]

Main MSP430 IPE **MSP430 MPU**

Access Management

Enable Memory Protection Unit (MPU)

Let compiler tools handle memory partitioning, associated access rights management, and the appropriate access rights

Manually specify memory segment boundaries and access rights

Address	Segment	R	W	X	Assert PUC on access rights violation
0x14000	Segment 3	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
0x5400	Segment 2	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
0x04800	Segment 1	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
0x04400	Info Memory	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Lock Configuration
 Enable NMI

- Intuitive way to set segment addresses and assign permissions
- Allows selection of interrupts and/or PUC on violation
- Lock configuration until BOR event

MPU Settings via GUI (CCSv6 only)

Properties for emptyProject

type filter text

Resource
General
Build
MSP430 Compiler
Processor Options

General

Configuration: Debug [Active]

Main MSP430 IPE **MSP430 MPU**

Access Management

Enable Memory Protection Unit (MPU)

Let compiler tools handle memory partitioning, associated access rights management, and the appropriate access rights

Manually specify memory segment boundaries and access rights

Address	Segment	R	W	X	Assert PUC on access rights violation
0x14000	Segment 3	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
0x5400	Segment 2	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
0x04800	Segment 1	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
0x04400	Info Memory	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Lock Configuration
 Enable NMI

- Intuitive way to set segment addresses and assign permissions
- Allows selection of interrupts and/or PUC on violation
- Lock configuration until BOR event
- We recommend: Let the compiler tools manage MPU automatically (explained next)

Linker CMD file is Key to Magical GUI

```

SECTIONS
{
  GROUP(ALL_FRAM)
  {
    GROUP(READ_WRITE_MEMORY)
    {
      .cio      : {}
      .system  : {}
    } ALIGN(0x0400), RUN_START(fram_rw_start)

    GROUP(READ_ONLY_MEMORY)
    {
      .cinit   : {}
      .pinit   : {}
      .init_array : {}
      .const   : {}
    } ALIGN(0x0400), RUN_START(fram_ro_start)

    GROUP(EXECUTABLE_MEMORY)
    {
      .text    : {}
    } ALIGN(0x0400), RUN_START(fram_rx_start)
  } > FRAM | FRAM2
        
```

- ◆ Default linker command file creates **3 groups of sections**:
 - ◆ Read/Write
 - ◆ Read Only
 - ◆ Read/Execute
- ◆ **Address symbol** defined for each group
- ◆ All groups are **allocated** to FRAM
- ◆ GUI assigns:
 - ◆ MPUSEGB2 = fram_rx_start
 - ◆ MPUSEGB1 = fram_ro_start

You can edit CMD file to add new custom sections to any group

FRAM Code Example

DriverLib FRAM & MPU API's

FRAM writes are managed by

- FRAM_write8()
- FRAM_write16()
- FRAM_write32()
- FRAM_memoryFill32()

The FRAM interrupts are handled by

- FRAM_enableInterrupt()
- FRAM_getInterruptStatus()
- FRAM_disableInterrupt()

The status is given by

- void FRAM_configureWaitStateControl
- void FRAM_delayPowerUpFromLPM (u

The MPU initialization function is

- MPU_start()

The MPU memory segmentation and access right definition

- MPU_createTwoSegments()
- MPU_createThreeSegments()

The MPU interrupt handler functions

- MPU_enablePUCOnViolation()
- MPU_disablePUCOnViolation()
- MPU_getInterruptStatus()
- MPU_clearInterruptFlag()
- MPU_clearAllInterruptFlags()
- MPU_enableNMlevent()

The MPU lock function is

- MPU_lockMPU()

Configuring MPU in Software

For CCSv5.5 users or if you want to setup the MPU in your own code:

```
extern const uint16_t fram_ro_start;
extern const uint16_t fram_rx_start;

void initMPU(void)
{
    // Configure MPU Segments
    MPU_createThreeSegments( MPU_BASE,
        &fram_ro_start,           // Boundary between 1 & 2
        &fram_rx_start,         // Boundary between 2 & 3
        MPU_READ | MPU_WRITE | MPU_EXEC, // Seg 1: all access
        MPU_READ,                // Seg 2: read only
        MPU_READ | MPU_EXEC );   // Seg 3: read & exe

    // Disable PUC on segment access violation for segment 2
    MPU_disablePUCOnViolation( MPU_BASE,
        MPU_SECOND_SEG );       // Segment 2

    // Start MPU protection
    MPU_start( MPU_BASE );
}
```

Code Example: Putting Var into “Info B”

```
#pragma DATA_SECTION ( b, ".infoB" )
uint16_t b = 0;

int MyLine( int m, int x )
{
    int y;

    y = (m * x) + b;
    Return ( y );
}
```

- ◆ FRAM makes this easy – as simple to use as RAM
- ◆ Only *exceptional* coding is to assign the variable to a section being allocated to FRAM
- ◆ Default linker command file assigns .infoB to a memory segment called INFOB: `infoB : {} > INFOB`
- ◆ All Info blocks are defined in the CMD file in a similar fashion

Setting FRAM Wait-States

Recommended Operating Conditions			MIN	NOM	MAX	UNIT
V _{CC}	Supply voltage range applied at all DVCC and AVCC pins. ^{(1) (2) (3)}		1.8 ⁽⁴⁾		3.6	V
C _{DVCC}	Recommended capacitor value at DVCC ⁽⁵⁾			1		μF
f _{SYSTEM}	Processor frequency (maximum MCLK frequency) ⁽⁶⁾	No FRAM wait states (NWAITS _x =0)	0		8 ⁽⁷⁾	MHz
		With FRAM wait states (NWAITS _x =1) ⁽⁸⁾	0		16 ⁽⁹⁾	
f _{ACLK}	Maximum ACLK frequency				50	kHz
f _{SMCLK}	Maximum SMCLK frequency				16 ⁽⁹⁾	MHz

```
// If you run the CPU > 8 MHz, you need to set wait-states
FRAM_configureWaitStateControl( FRAM_ACCESS_TIME_CYCLES_1 );
```

Hint: Place this in your initClocks() function – near your MCLK setup code

Lab 7 Exercise - FLASH and FRAM

Lab Exercises

- ◆ **Lab A – Count Power Cycles**
 - ◆ Create non-volatile variable – use it to count the # of power-cycles
 - ◆ Blink LED the # of times there's been a power cycle
 - ◆ printf() to console the # of power cycles
 - ◆ Use custom sections and linker command file to create the non-volatile variable
 - ◆ (Flash only) Use API to write to NVM
 - ◆ Use memory map and memory browser to ascertain where variables were allocated by the linker

- ◆ **Lab B – MPU Configuration (FRAM only ... Flash lab is work-in-progress)**
 - ◆ Configure MPU to use 3 segments
 - ◆ Write to 'read-only' segment of FRAM to cause a memory violation interrupt
 - ◆ (Planned) Show how to setup and handle memory violations when using Flash memory

lab_07a_infoB

This lab uses non-volatile memory to store a data value so that it will be available after a power-cycle.

The value will be stored in Info B, which is one of the four segments of non-volatile memory (NVM) set aside for data information. The 'F5529 uses flash technology to store non-volatile information, while the 'FR5969 uses FRAM.

The code itself will keep track of how many power-cycles (BOR's) have occurred. After power up and initializing the GPIO, the code looks for a count value in NVM, it then increments the count value and:

- Writes the updated value back to Flash or FRAM
- Prints out the # of power-cycle counts with printf()
- Blinks the LED count # of times

To minimize your typing, we have created the project for you. The "hello.c" file in this project is an amalgam of labs:

- lab_03a_gpio for the gpio setup
- lab_04b_wdt for the printf functionality

To this we've added:

- Logic to manage the "count" value
- A function which writes to Flash (Info B) ('F5529 only - FRAM doesn't need it)
- You will need to fill in a few answers from your Lab 7a worksheet

There is no MPU "protection" setup for the 'FR5969 FRAM in this exercise. That is shown in lab_07b_protection.

Worksheet

1. Examine the linker command file (.cmd) and find the name of the memory area that represents the Info B memory.

Name of memory area: _____

Address of Info B: _____

Finish this line of code:

```
#pragma _____ (count, "_____")
static uint16_t count;
```

2. Again, looking at the linker command file, what address symbol is created by the linker to represent the starting address of executable code?

3. (**F5529 only**) What functions are needed to erase and write to Flash?

(Note, we're interested in writing 16-bit integers to Flash.)

```
//Erase INFOB
do {
    _____( (uint8_t*)INFOB_START );
    status = FLASH_eraseCheck(
        (uint8_t*)INFOB_START,
        NUMBER_OF_BYTES );
} while (status == STATUS_FAIL);

//Flash Write
_____ (
    (uint16_t*) value,
    (uint16_t*) flashLocation,
    1
);
```

Lab Procedure:

1. **Import project from lab folder:** lab_07a_infoB

2. **Fill in the blanks in:** hello.c

3. **(FR5969 only) Modify the .infoB setting in linker command file.**

Since FRAM reads/writes like SRAM, the compiler autoinitializes it each time our C program starts ... just like any other global variable. Of course, that's not what we want in this instance – we want to use the non-volatile nature of FRAM to maintain the value of 'count' when the power is off. To make this happen, we can tell the tools to "not initialize" the variables. This can be done by editing one line in the linker command file to add the NOINIT type.

```
.infoB      : { } > INFOB  type=NOINIT
```

We could have limited the scope of our NOINIT modification, but it's an easier edit to set this type for the entire .infoB section.

4. **Build program the program.**

Fix any syntax errors and rebuild until your program compiles successfully.

5. **Open the .map file and answer the questions below.**

The .map file is a report created by the linker which records where memory was allocated. You should be able to find it in the Debug folder of your project.

INFOB memory region: _____

Where was this address specified? _____

.infoB section address: _____

Beginning of code (.text section): _____

Length of program code (.text): _____

count: _____

fram_rw_start: _____

fram_ro_start: _____

fram_rx_start: _____

6. Launch the debugger.**7. Open the memory window**

View → Memory Browser

Try looking at some of the locations used in our code:

```
0x1900
&fram_rx_start
&count
```

From the Memory Browser, what is the address of: &count _____

8. Add variables to watch Expressions for:

```
count
c (for 'F5529 devices)
i (you can also see 'i' in the local Variables window)
```

9. Single-Step thru code to watch it work.

The Memory Browser is interesting because you can see the variable in Flash.

Hint: You can also modify the value in Flash by changing it in the Memory Browser. This is convenient if you want to reset the value back to 0.

10. Restart program.

If you let the program run without a breakpoint, you may need to Suspend it before Restarting it.

11. Step through code again ... hopefully it retained its count value.

You should see the printf() statement count value updated and the LED blink one more time than the previous run.

12. Terminate the debugger and unplug the board – then plug it back in.

Do you see the LED blinking. Again, it should be 1 more time than previously.

13. Reset the Launchpad with the reset button ... does the LED blink 1-more-time each time its reset or power-cycled?

Finally, just clicking the reset button on your board (without unplugging/plugging it) should be enough to restart the program and increment count.

('F5529 Optional) lab_07a_low_wear_flash

'F5529 only -- FRAM parts rarely need to worry about wear issues due to their high endurance.

This example modifies lab_07a_infoB by using the entire infoB segment. In original exercise, we wrote count to the first location in Info B. On the next power-cycle we erased the entire Info B segment and wrote one location; we did this again-and-again on every power-cycle.

This solution provides a simple method of minimizing FLASH wear. Rather than erasing the entire flash on each power-cycle, we use the consecutive location in flash. We keep doing this until we reach the end of Info B; only when we reach the end do we erase the entire segment and start over again.

While there are probably better algorithms to handle these types of flash wear issues, this is a simple example solution to the problem.

Import and explore the lab_07a_low_wear_flash solution

('FR5969) lab_07b_protection

This lab explores the use of the Memory Protection Unit (MPU). We program the MPU using DriverLib and then set about violating the assigned protections by trying to write to read (and execute) only memory. These set up these violations to create NMI (non-maskable interrupt) events.

Import and explore the lab_07b_protection lab

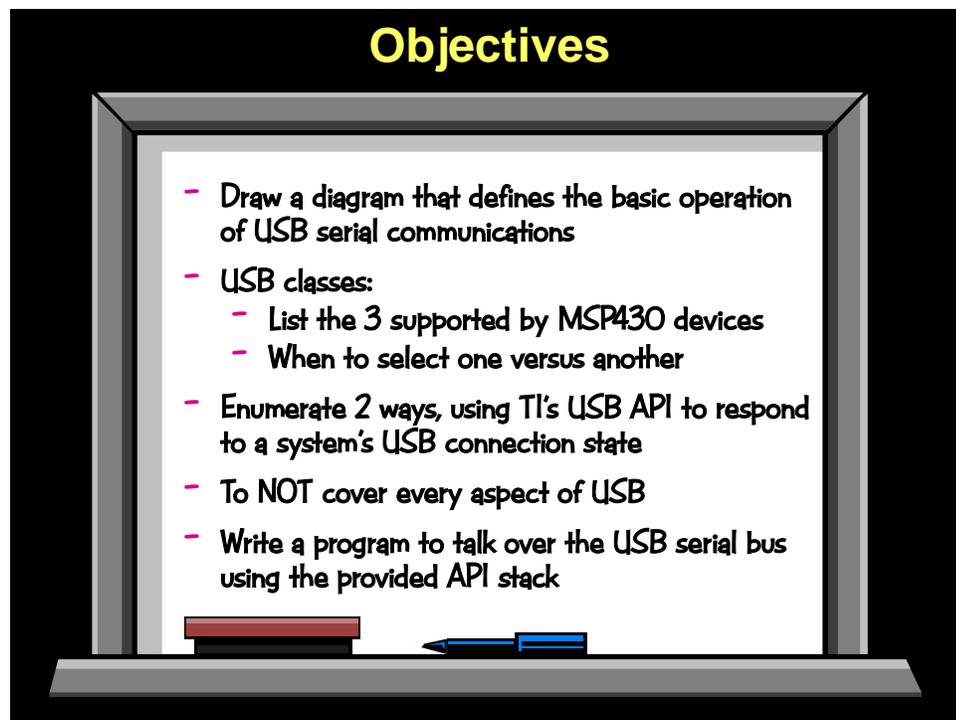
Some things to try:

- Single-step through the MPU setup function and watch as the MPU registers are configured.
- Set breakpoints on the 4 different cases in the NMI interrupt handler that are related to the 4 different FRAM segments. Why don't we get *Info* and *Segment 1* interrupts?

Introduction

The MSP430 makes an ideal USB device: ultra-low power, rich integration of peripherals and it's inexpensive. Do you want to make a Human Interface Device product? Maybe a sensor, such as a barcode reader, that needs to be both low-power (when collecting data), but also capable of 'dumping' its data via USB to a computer. Dream big, we've got the devices, tools, and software to help you make them come true.

Learning Objectives



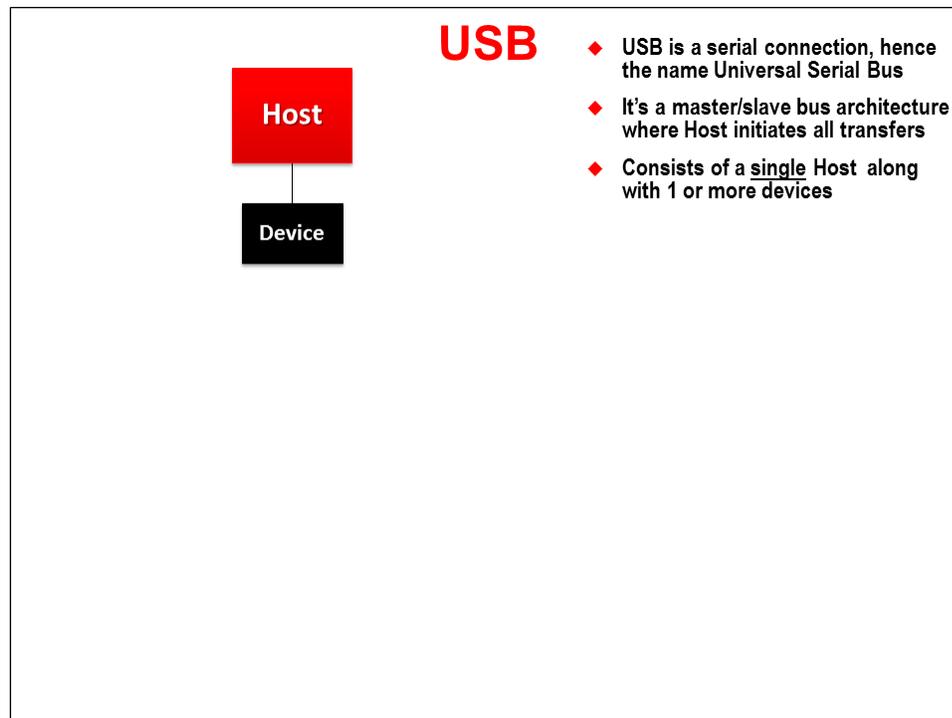
Chapter Topics

USB Devices	8-1
<i>Introduction</i>	<i>8-1</i>
<i>What is USB?</i>	<i>8-3</i>
<i>MSP430's USB Support.....</i>	<i>8-6</i>
<i>How USB Works</i>	<i>8-12</i>
<i>Descriptions and Classes.....</i>	<i>8-15</i>
<i>Quick Overview of MSP430's USB Stack.....</i>	<i>8-20</i>
<i>ABC's of USB.....</i>	<i>8-23</i>
A. Plan Your System	8-23
B. Connect & Enumerate	8-24
C. Managing my App & Transferring Data	8-26
<i>Final Thoughts</i>	<i>8-29</i>
<i>Lab Exercise</i>	<i>8-31</i>

What is USB?

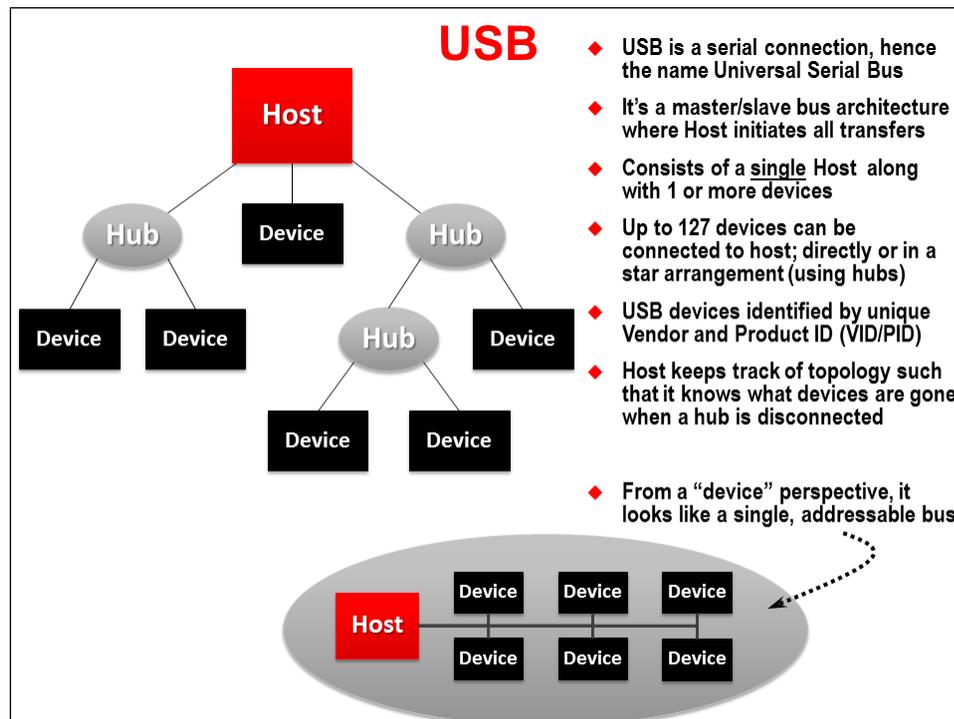
Universal Serial Bus (USB) is just that, a universal serial connection between a “Host” and “Device”. It has taken the place of many synchronous and asynchronous serial bus connections in systems such as personal computers.

In the case of USB, the *host* manages all transfers, whether moving data to or from the *host* – often this is called a master/slave architecture, where the *host* is the bus master. At a minimum, there needs to be one *host* and one *device*.



But, USB supports many more than just a single *device*, the standard can actually support up to 127 different *devices*. Commonly, systems with multiple devices use hubs as interconnection points between the *host* and *devices* – which results in a star arrangement.

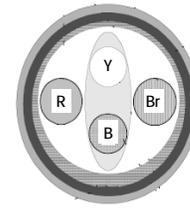
Each type of device is distinguished using Vendor and Product ID (VID/PID). The combination of VID and PID allows a host to identify the type of device that is connected and manage its point-to-point communications with the device – in most cases, this requires the host to load the appropriate drivers to talk with that specific type of device. (We'll discuss this in greater detail later in the chapter.)



USB ... Physical Layer

◆ **Four wires in the cable/connector:**

- ◆ **VBUS** (5V - supplied by host)
- ◆ D+ } for **differential data** signaling
- ◆ D- }
- ◆ Ground



- ◆ Originally only two connector types (host & device), though many additional plugs were defined later
- ◆ USB 2.0 added On-The-Go (OTG) feature, letting devices switch from device to host, as needed
- ◆ USB 3.0 has concurrent bidirectional data transfers, thus cables include four more data lines (backward compatible)
- ◆ USB devices are **hot swappable**



USB Standards

Version	Year	Speeds	Power Available	Notes
USB 1.1	1995	1½ Mbps (Low) 12 Mbps (Full)	–	Host & Device connectors
USB 2.0	2000	1½ Mbps (Low) 12 Mbps (Full) 480 Mbps (High)	500 mA	<ul style="list-style-type: none"> • Backward compatible with USB 1.1 • Added On-the-Go (OTG)
USB 3.0	2008	1½ Mbps (Low) 12 Mbps (Full) 480 Mbps (High) 4.8 Gbps (Super)	900 mA	<ul style="list-style-type: none"> • Backward USB 2.0 compatibility • Full-duplex • Power mgmt features

MSP430 USB Peripheral Supports

- ◆ **USB 2.0 standard**
- ◆ **Full speed USB device (12Mbps)**
- ◆ **Device only**

Note: Look at TI's TivaC processors if you need host, device or OTG support

MSP430's USB Support

MSP430 USB Support

Most comprehensive low power

1. Largest 16-bit portfolio of integrated USB and 512KB memory
2. Proven USB core
3. Optimized for low power operation

F5xx

Ultra-low power MCU with up to 25MHz and integrated USB



F6xx

Ultra-low power MCU with up to 25MHz, integrated USB and LCD



1. Perfect for developers new to USB as well as experienced engineers
2. Code gen tools and proven USB stacks significantly eases development (at no cost to the customer)
3. Availability of a new low price MSP430 USB LaunchPad tool



MSP430 Devices with USB

Product s	Prog (KB)	RAM (KB)	16-Bit Timers	Common Peripherals	ADC	Additional Features
MSP430F663x	up to 256	8 to 16	4	WDT, RTC, DMA(3-6), MPY32, Comp_B, UART, SPI, I2C, PMM (BOR, SVS, SVM, LDO)	12-bit	USB, EDI, DAC12, LCD, Backup battery switch
MSP430F563x	up to 256					USB, EDI, DAC12, Backup battery switch
MSP430F552x	32 - 128	6 to 8			—	USB, 25 MIPS
MSP430F551x	32 - 128	4 to 8				
MSP430F550x	8 - 32	4			10-Bit	

- ◆ Portfolio of devices with more (or less) peripheral/memory integration; this provides basis for different price points
- ◆ USB Launchpad uses the 'F5529 ... found in the middle of the pack

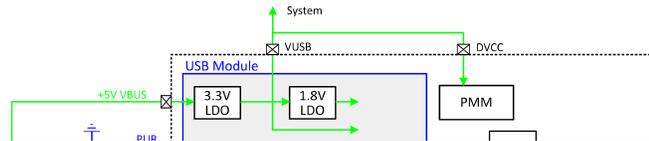
MSP430 USB Module

2.3 MSP430 USB Module

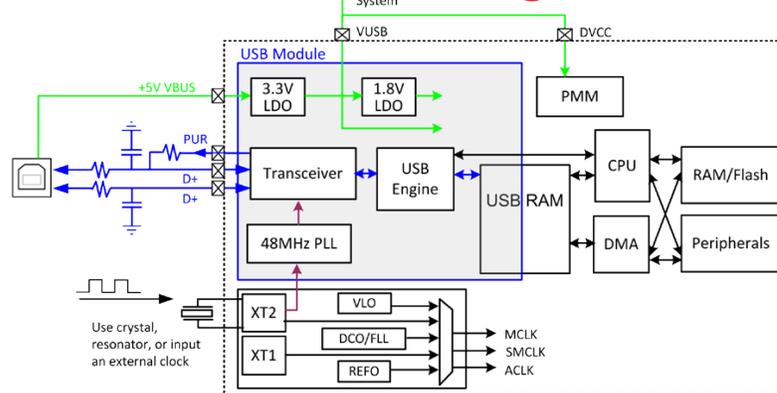
Features of the MSP430 USB module are as follows:

- **Full-speed USB device (12 Mbps).** Full-speed is a great match for a 16-bit MCU. It facilitates communication with a USB host, with simplicity and low system cost. The module does not perform low- or high-speed transfers; it also does not function as a USB host controller.
- **Supports control, interrupt, and bulk transfers.** This enables support of the most popular USB device classes. (Streaming audio using isochronous transfers is not supported.)
- **Eight input and eight output endpoints.** The more endpoints that are supported, the more *USB interfaces* (logical devices) that can be implemented within a *composite USB device*. MSP430 MCUs have enough endpoints for as many as seven interfaces in composite (depending on the ones chosen), which is more than enough for the vast majority of USB applications.
- **An integrated 3.3-V LDO, for operation directly from 5-V VBUS from the host.** In some applications, this eliminates the need for an external LDO, because in addition to sourcing the MCU, the integrated LDO can be used to source the entire system, up to 12 mA. (See the device data sheet for parameters).
- **An integrated D+ pullup.** This pullup is the way in which a USB device tells the host it is ready to be enumerated. In contrast, some USB devices from other vendors require external circuitry to enable the pullup.
- **Programmable PLL.** An integrated PLL generates the 48-MHz clock needed for USB operation. The reference for this PLL comes from the MCU's XT2 oscillator. A wide variety of sources can be used for the reference.
- **Integrated transceiver (PHY).** There is no need to buy one separately.

Figure 1 shows a system block diagram.



USB Block Diagram



MSP430 USB Peripheral Supports

- ◆ USB 2.0 standard
- ◆ Device only (Host not supported— try Tiva-C from TI)
- ◆ Full speed USB device (12Mbps)
- ◆ Includes 16 Endpoints (8-in/8-out)
- ◆ Certified USB module
- ◆ Integrated transceiver (PHY)
- ◆ Integrated 3.3V LDO for direct operation from USB bus
- ◆ Programmable PLL (generates 48MHz USB clock)
- ◆ Integrated D+ pull-up resistor (PUR)

USB Power System (UPS)

- **Integrated LDO**
 - Operates directly from 5V V_{BUS}
- **LDO output is provided externally**
 - Can be used to power device DV_{CC}
 - Can be used to power non-MSP430 components as well (<12mA)
- **DV_{CC} can also be provided through other means**
 - Easy flexibility for bus/self-powered operation, USB battery charging, etc.

System

USB

DV_{CC}

+5V V_{BUS}

V_{BUS}

3.3V LDO

1.8V LDO

Transceiver & PLL

Transceiver

PMM

USB Module

Other MSP430

TEXAS INSTRUMENTS

USB Clocking

- **USB requires a HF clock, supplied by an integrated PLL**
- **PLL needs a reference clock**
 - Apply a crystal to the XT2 oscillator (>4MHz)
 - Or, apply a clock from elsewhere in the system, in bypass mode (>1.5MHz)
- **The PLL is programmable, allowing a wide range of freq's**
 - Choose one already in the system
 - Or choose the cheapest crystal you can find
- **XT2 is reusable for other system functions**

USB

48MHz PLL

VLO

REFO

XT1

XT2

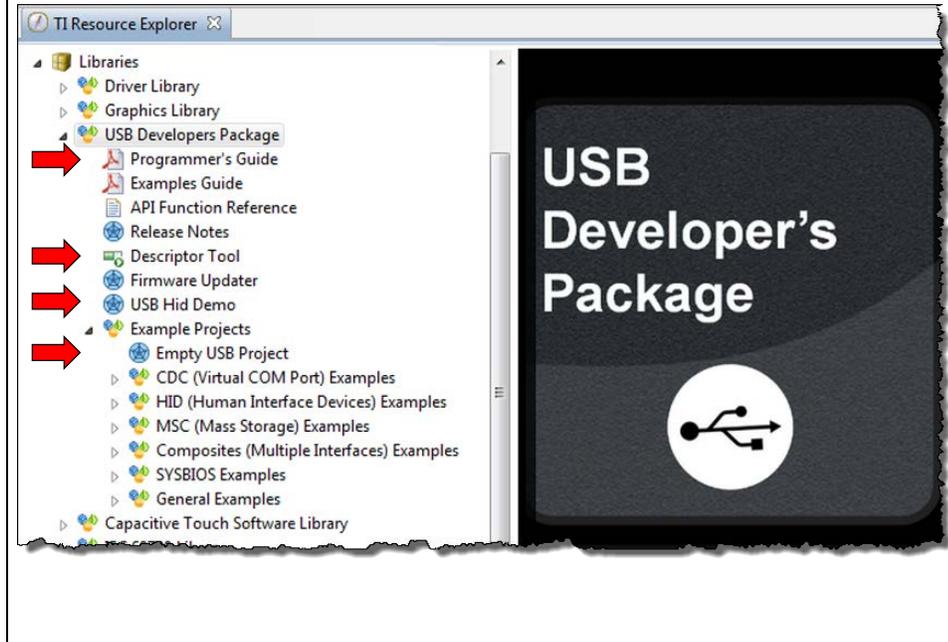
MCLK

SMCLK

ACLK

TEXAS INSTRUMENTS

USB Developers Package



MSP430 USB API Features

1. A finished API
 - Not just example code
 - Increases chance of USB success, because the user doesn't need to modify the USB plumbing; speeds development
 - An API approach makes USB more accessible to USB non-experts
2. Small memory footprint
 - Single-interface CDC or HID: 5K flash / 400 bytes RAM
 - MSC (not including file system / storage volume): 8K flash / 1.4K RAM
3. Can use either DMA or CPU to move data
 - Simply turn the DMA feature 'on' and select the channel
4. Limited resource usage
 - Only uses the USB module, some memory, & a DMA ch; no other resources
5. RTOS-friendly
 - TI will soon provides using it with TI-RTOS

MSP430 USB API Features, cont.

6. Responsiveness
 - No risky blocking calls stuck waiting for the host
 - Data can be transferred “in the background”, for increased system responsiveness and efficiency, even with a busy host/bus
7. Easy data interface (CDC and HID-Datapipe)
 - The function calls are similar to interfacing with a simple COM port
 - You can send/receive data of any size, with a single call -- no packetization required
 - Deep USB knowledge not required
8. Flexibility (MSC)
 - Compatible with any file system software. (We provide the open-source “FatFs” as an example.)
 - Easy multiple-LUN support; just select the number of LUNs
 - No RTOS required – but can be ported to one

USB Fee ... You need a Vendor ID

Fees. The USB-IF provides the USB specification, related documents, software for compliance testing, and much more, all for free on its Web site.

Anyone can develop USB software without paying a licensing fee.

However, anyone who distributes a device with a USB interface must obtain the rights to use a Vendor ID.

- ◆ Vendor ID's (VID) are assigned by the USB Implementers Forum (USB-IF)
- ◆ Obtain VID by:
 - ◆ Joining USB-IF (\$4000 annually)
 - ◆ Get a 2 years license (\$3500)
 - ◆ See <http://www.usb.org/developers/vendor/>
- ◆ Alternatively, TI VID-sharing program licenses VID's to MSP430 customers
 - ◆ For use with the MSP430 VID (0x2047)
 - ◆ License is free, with stipulation it's only used with TI USB devices
 - ◆ Find out more at : <http://www.ti.com/msp430usb>

Clipped from, "USB Complete: The Developer's Guide" by Jan Axelson (ISBN 1931448086)

<http://www.ti.com/msp430usb>

Come here to get up to date for all things related to MSP430 USB!

Microcontrollers (MCU)

• [Design Support](#) • [Getting Started](#) • [Selection Tool](#) • [Training & Events](#) • [Developer Network](#)

MSP430 Applications

Ultra-Low Power
Wireless
Utility Metering
Portable Medical
Security
Energy Harvesting
USB

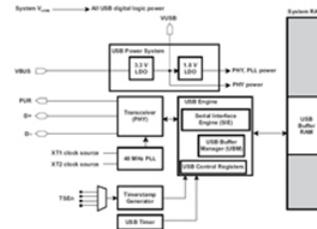
MSP430 + USB

The MSP430 portfolio has been expanded to include a variety of devices integrated with USB, ideal for applications including analog and digital sensor systems, data loggers, and other solutions that require connectivity to various USB hosts. With the MSP430F55xx family of devices, intuitive evaluation tools, and a library of USB software, designers are prepared to implement USB in their projects today!



MSP430's USB Module Features:

- Full speed USB device at 12 Mbps
- Supports control, interrupt, and bulk transfers
- Eight input / Eight output endpoints
- Integrated 3.3V LDO - for direct operation from 5V VBUS
- Integrated D+ pull-up
- Integrated transceiver
- Timestamp generator capable of 62.5 ns resolution



MCU Training

- > Register now for MCU Day
- > TI Technology Days

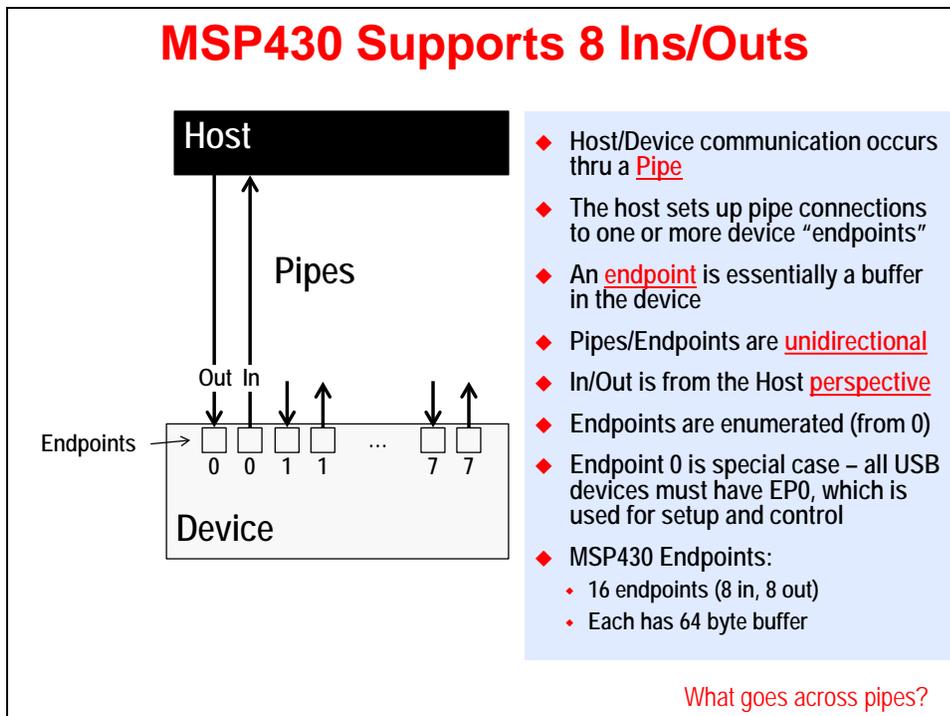
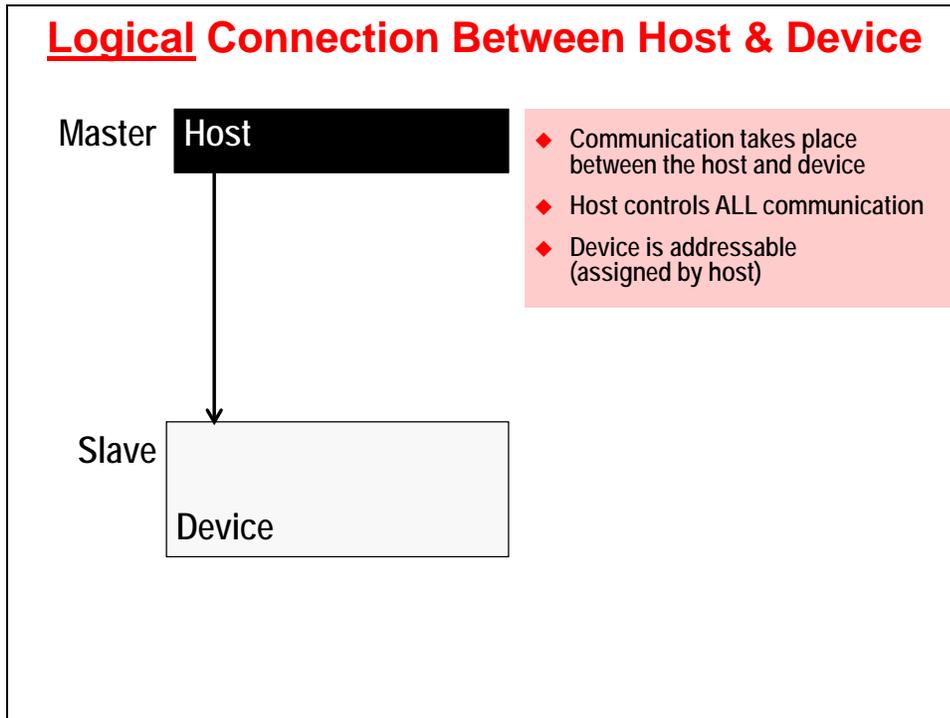
Support

- > TI E2E Community
- > Contact Technical Support
- > MSP430 Discussion Group
- > Third-Party Network

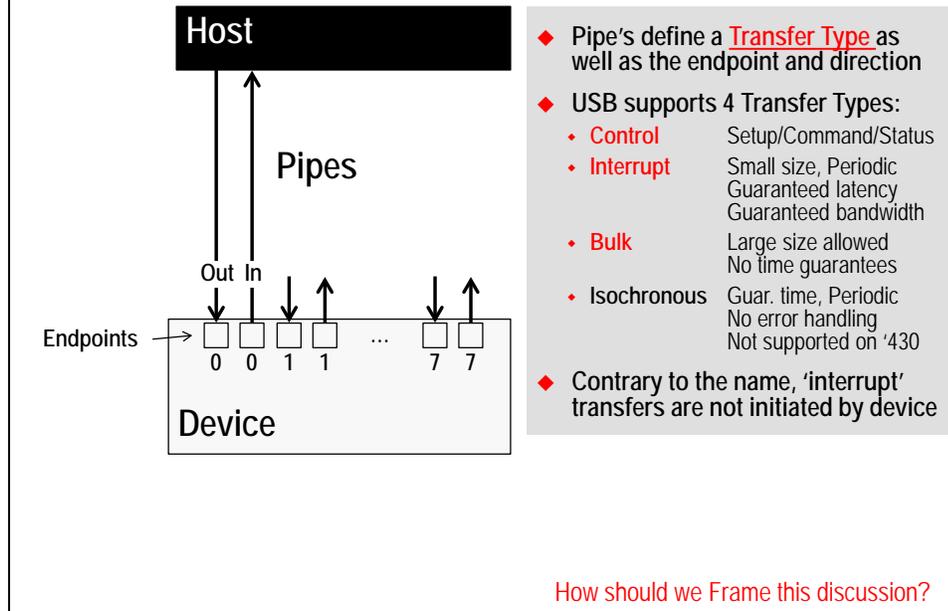
Suggested Reading

- ◆ **"Starting a USB Design Using MSP430™ MCUs"** App Note by Keith Quiring (Sept 2013) (Search ti.com for [SLAA457.pdf](#))
- ◆ **"Programmers_Guide_MSP430_USB_API"** by Texas Instruments (Aug 2013)
Found in the *MSP430 USB Developers Package*
- ◆ **"USB Complete: The Developer's Guide"** by Jan Axelson (ISBN 1931448086)
<http://www.amazon.com/USB-Complete-Developers-Guide-Guides/dp/1931448086>

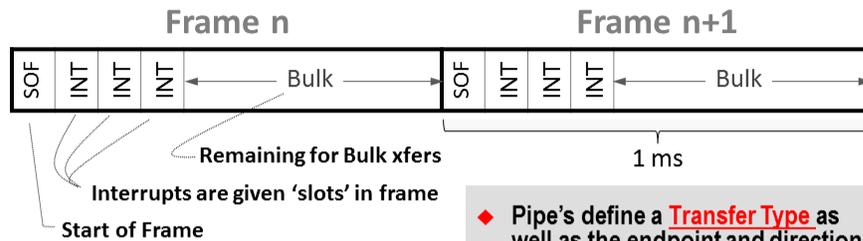
How USB Works



USB Transfers



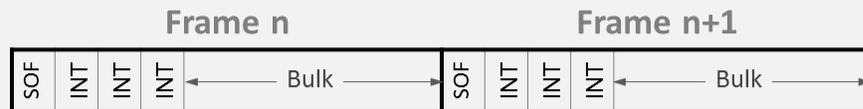
How is Bandwidth Guaranteed?



- ◆ Host won't allocate more than 90% to periodic transfers (e.g. Interrupt)
- ◆ Periodic transfers can be config'd for every 1 to 255 frames
- ◆ Next in priority is Control transfers
- ◆ Last priority are Bulk transfers
 - ◆ Bulk can be the fastest transfer type
 - ◆ But, they have no guarantees

- ◆ Pipe's define a **Transfer Type** as well as the endpoint and direction
- ◆ USB supports 4 Transfer Types:
 - ◆ **Control** Setup/Command/Status
 - ◆ **Interrupt** Small size, Periodic Guaranteed latency Guaranteed bandwidth
 - ◆ **Bulk** Large size allowed No time guarantees
 - ◆ **Isochronous** Guar. time, Periodic No error handling Not supported on '430
- ◆ Contrary to the name, 'interrupt' transfers are not initiated by device

How Do You Fit Large Transfers



- ◆ Transfers are broken down into:
 - ◆ Transactions ... and further into ...
 - ◆ Packets
 - ◆ Whose details are beyond the scope of this presentation
- ◆ Transfers (except Isochronous) are verified using Handshaking, CRC, Data Toggle ... if an error occurs, they are retransmitted
- ◆ Thankfully, the USB hardware (and API stack) takes care of these details

Descriptions and Classes

What Do You Want to Transmit?

- ◆ USB devices describe one (or more) [Interfaces](#) to transmit information
- ◆ Typical interface examples:
 - ◆ Creating a Virtual COM port requires 2-in and 1-out endpoints
 - ◆ Human interface devices (mice/keyboards) require 1-in/1-out
 - ◆ Memory devices also require 1-in/1-out

Device (example shown here is 'composite' device with multiple I/F's)

What Do You Want to Transmit?

- ◆ USB devices describe one (or more) [Interfaces](#) to transmit information
- ◆ Typical interface examples:
 - ◆ Creating a Virtual COM port requires 2-in and 1-out endpoints
 - ◆ Human interface devices (mice/keyboards) require 1-in/1-out
 - ◆ Memory devices also require 1-in/1-out
- ◆ USB devices must describe their themselves using [device descriptors](#)
- ◆ Host must match descriptors (at run time) with host-side device drivers
- ◆ MSP430 supports a [single configuration](#) with [one or more interfaces](#)

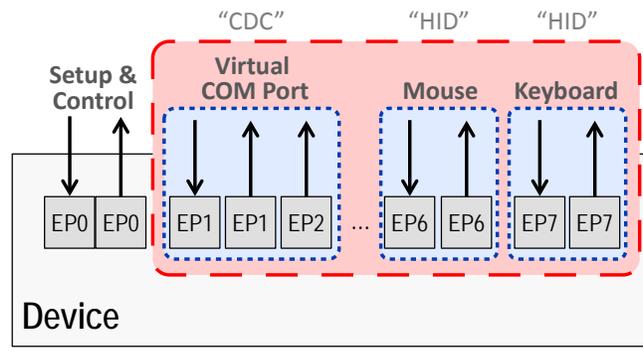
Device (example shown here is 'composite' device with multiple I/F's)

How can we simplify configuration?

USB Classes

USB defines a number of device classes

- ◆ Human Interface Device (HID)
 - ◆ Communications Device (CDC)
 - ◆ Memory Storage Class (MSC)
 - ◆ Printer
 - ◆ Audio
 - ◆ Etc.
- MSP430 Supports 4 classes**
- HID, CDC, MSC (and PHDC)
 - Simplifies specifying interfaces
 - Host O/S can easily match its drivers to known device classes
 - Descriptors take form of:
 - ◆ Device: data-structures
 - ◆ Host: .INF file



Is there an easy way to create USB Descriptors?

MSP430 USB Descriptor Tool

Quick and easy way to create device descriptors and .inf files

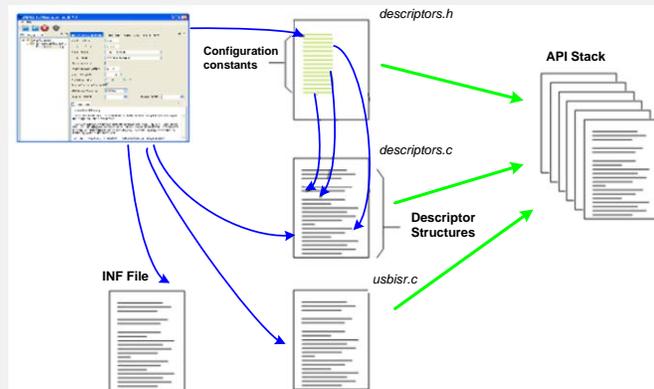
Minimizes error – very common when creating descriptors by hand

Help pane provides useful ‘how to’

Recognized by MSP430’s USB stack ... simply add this tools output to your USB project

Descriptor Tool: API Integration

- The Tool is tightly integrated with the API
- Generates three source files that configure the rest of the stack
- Also generates the INF file (for CDC on Windows)

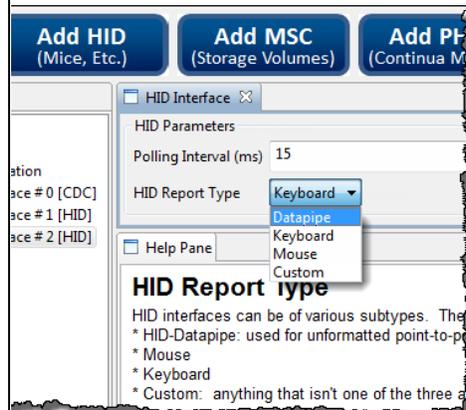


Communications Data Class (CDC)



- ◆ Implements a virtual COM port on PC
- ◆ Simple serial terminal on Host side (e.g. HyperTerm, Putty, Tera Term)
- ◆ The API presents a generic data interface to the application
- ◆ Send/receive data of any size, with a single function call
- ◆ Uses simple calls like:
 - ◆ `USB_connect () ;`
 - ◆ `USB_sendData (buffer, size, intfNum) ;`
 - ◆ `USB_receiveData (buffer, size, intfNum) ;`
- ◆ Can be performed “in the background”
 - ◆ Increases program responsiveness
 - ◆ Improves efficiency

Human Interface Device (HID)

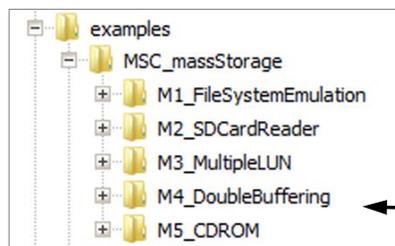
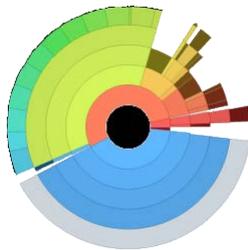


Datapipe mode allows the benefits of HID without some of its downsides

- ◆ **Silent loading** on the host
- ◆ Avoids USB's complex HID report structures
- ◆ Enables a unique value tradeoff

- ◆ HID classes transfers data in 'report' structures
- ◆ MSP430 supports any report type, but are 3 are built-in:
 - ◆ Keyboard (traditional)
 - ◆ Mouse (traditional)
 - ◆ Datapipe (generic)
- ◆ 'Datapipe' presents a generic data interface to the application
 - ◆ Makes it easy to use HID for a CDC-like interface
 - ◆ TI provides a HID host demo tool (which acts like host-side serial terminal for datapipe xfers)
 - ◆ Application code interchangeable with CDC code, for easy migration
- ◆ MSP430 also provides APIs for host-side HID development:
 - ◆ Windows
 - ◆ Mac

Memory Storage Class (MSC)

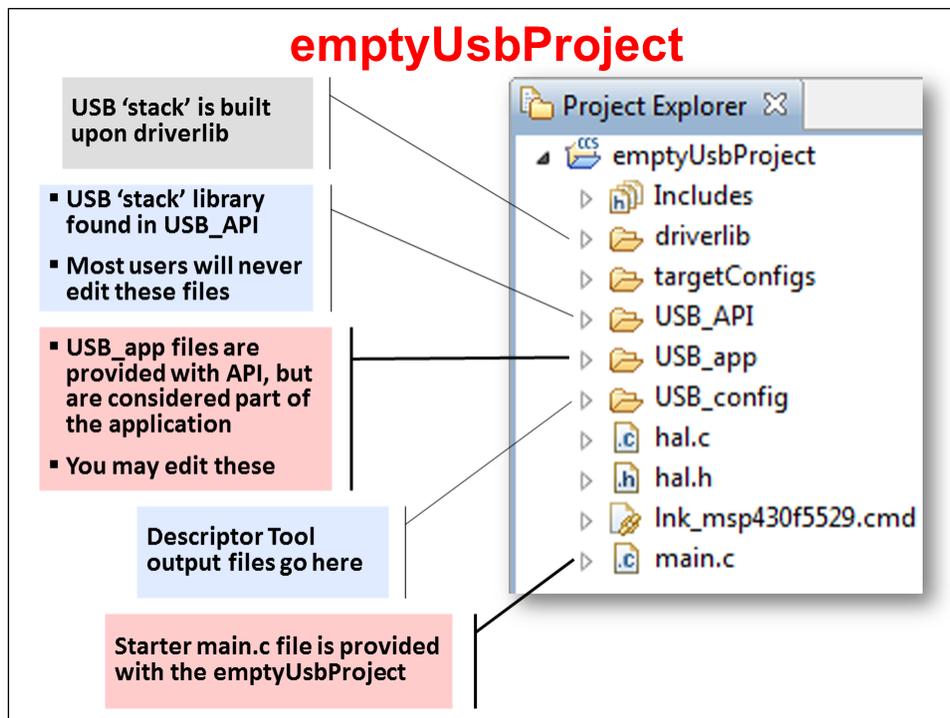
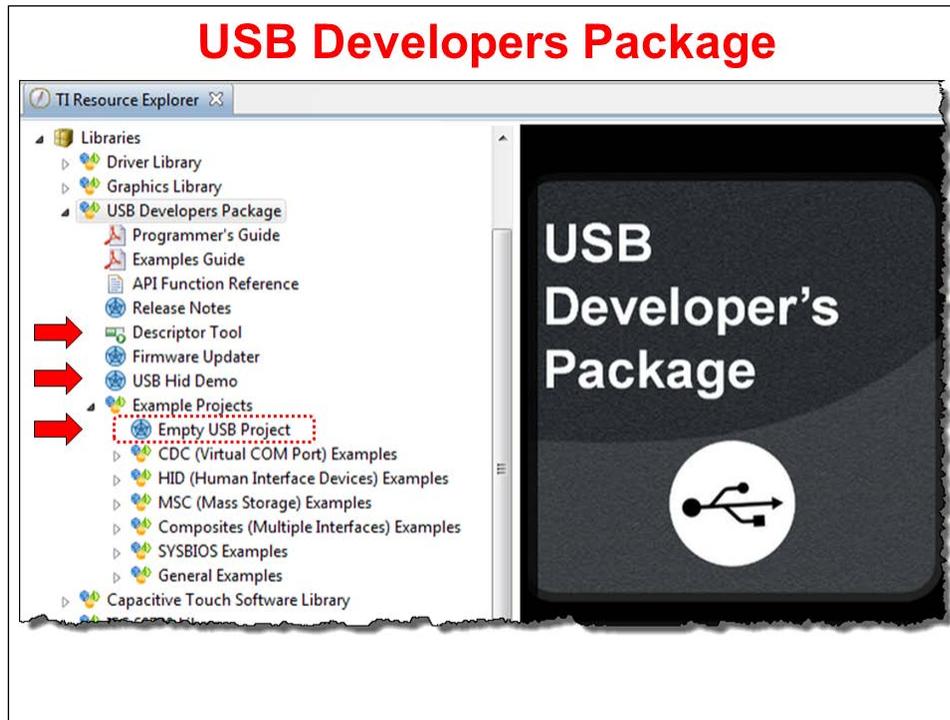


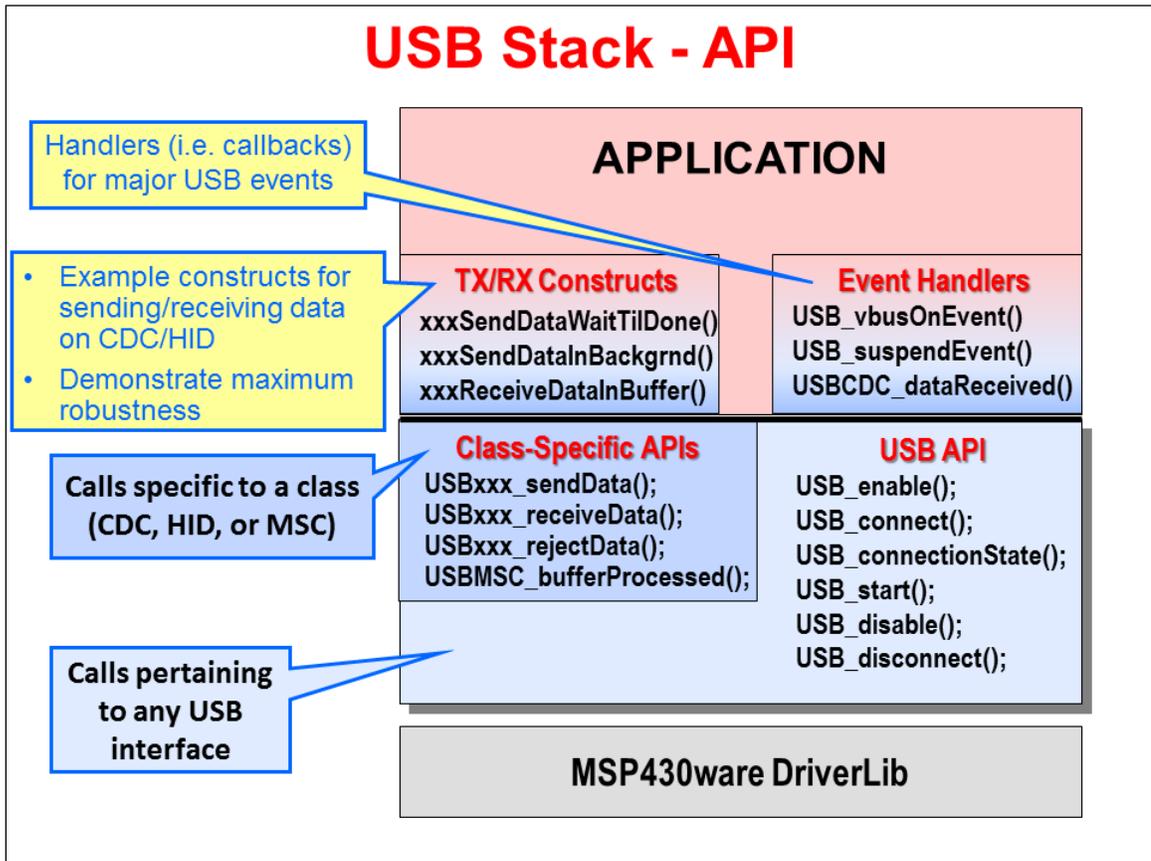
- ◆ Allows easy creation of a USB storage device
- ◆ No RTOS required
 - ◆ But can easily be ported to one
 - ◆ TI-RTOS (coming soon for MSP430) will provide a port with examples
- ◆ USB Developers Package includes a port of the open-source FAT file system (FatFS)
 - ◆ FatFS is provided as an example
 - ◆ USB stack was designed to be compatible with any file system
- ◆ Five demo apps provided

MSC will be covered in more detail in a new chapter under development

Comparison/Summary of Classes			
	CDC	HID	MSC
Host Interface	COM Port	HID device	Storage Volume
Host Driver Required	No (but .inf file required)	No	No
Host Loading	User Intervention	Silent	Silent
Bandwidth	"Hundreds of KB/sec"	62KB/sec	"Hundreds of KB/sec"
Code Size	5K	5K	9K (12-15K w/FS & vol)
Endpoints	2 in 1 out	1 in 1 out	1 in 1 out
Transfer Type	Bulk	Interrupt	Bulk (BOT)
Advantages	<ul style="list-style-type: none"> ▪ Familiar to user ▪ Bulk transport ▪ Common host apps 	<ul style="list-style-type: none"> ▪ Silent loading ▪ Interrupt xfers ▪ Mouse/Keybd 	<ul style="list-style-type: none"> ▪ Familiar to user ▪ Allows storage of data using filesys

Quick Overview of MSP430's USB Stack





Notes

ABC's of USB

ABC's of USB Implementation

Transfer Basics

You can divide USB communication **C** to two categories: **B** communications used in enumerating the device and communications used by the applications that carry out the device's purpose. During enumeration, the host learns about the device and prepares it for exchanging data. Application

- A. Plan Your System**
... and develop the device descriptors
- B. Handling the connection with Host**
 - Support the Host's discovery and setup of the connection (called enumeration – explained shortly)
 - Manage changes to connection state
 - To large part, this is automated by USB stack
- C. Data Communications**
 - Send/receive data - the original purpose of the connection

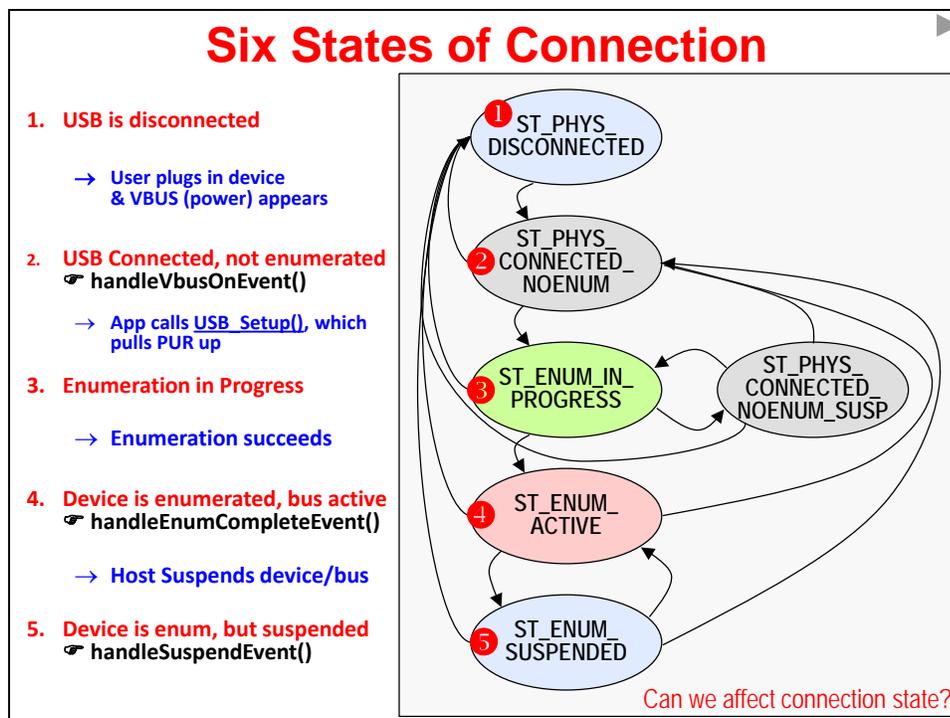
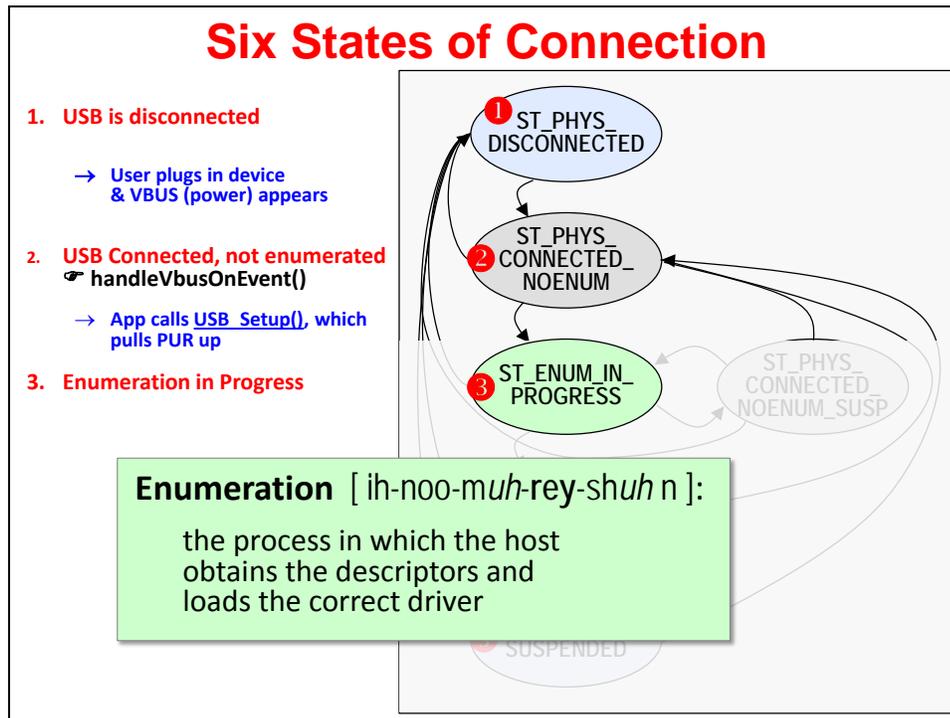
Clipped from, "USB Complete: The Developer's Guide" by Jan Axelson (ISBN 1931448086) ↗

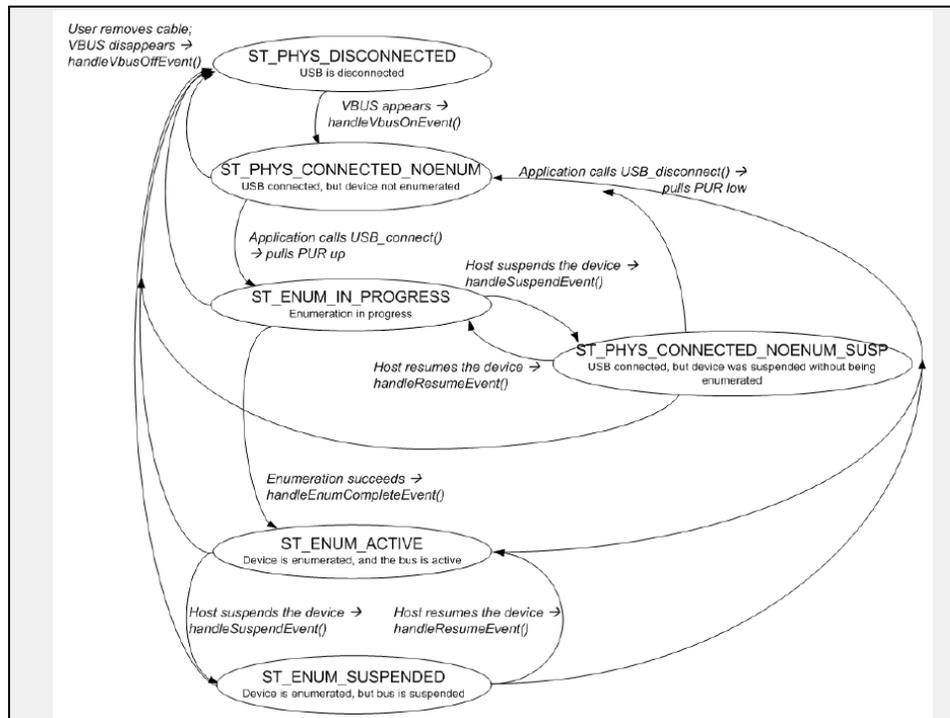
A. Plan Your System

Plan Your System

- 1. What are your requirements?**
 - ♦ How much data needs to transfer ... and how fast?
 - ♦ Is guaranteed bandwidth & timing important?
 - ♦ Are you connecting to Window, Mac, Linux (or all)
 - ♦ What power will be needed?
- 2. From the requirements, decide which class (or classes) will be needed**
- 3. Import EmptyUsbProject (Optional)**
- 4. Run Descriptor Tool**
 - ♦ Provides help & feedback in creating device description
 - ♦ Generates device descriptor files & INF files
 - ♦ If you followed step 3, it automatically drops generated files into the project

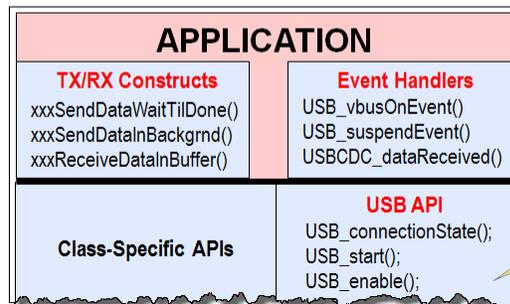
B. Connect & Enumerate





How Can I Modify Connection State?

- ◆ The Host handles most of the Enumeration process
- ◆ The USB stack handles the task of serving up descriptors
- ◆ The application isn't required to do much except call:
 - USB_setup () - To start the USB stack running
- ◆ Additionally, you can elect to disconnect from the USB bus



USB API provides functions to start, disconnect, suspend, resume, force Remote Wakeup, etc.

C. Managing my App & Transferring Data

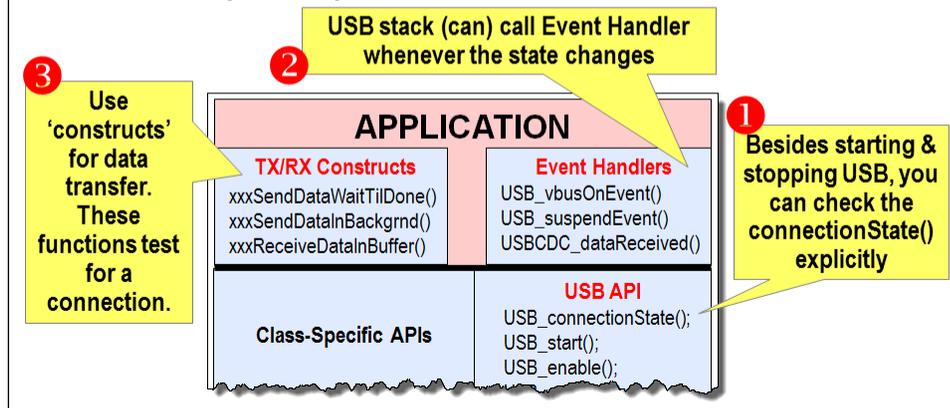
Respond to Connection State (as needed)

- ◆ Most USB programs adjust to connection state

For example:

- ◆ Call USB_Start() after VbusOnEvent
- ◆ Why send data if there isn't a connection?
- ◆ Reduce system power if host suspends USB bus
- ◆ ... to name a few

- ◆ Three ways to respond to connection state:



1 Main Loop USB Framework

```
while(1){
  switch( USB_connectionState() )
  {
    case ST_USB_DISCONNECTED:
      break;
    case ST_ENUM_ACTIVE:
      break;
    case ST_ENUM_SUSPENDED:
      break;
    case ST_ENUM_IN_PROGRESS:
      break;
    case ST_USB_CONNECTED_NO_ENUM:
      break;
    case ST_NOENUM_SUSPENDED:
      break;
    case ST_ERROR:
      break;
    default:;
  }
}
```

These three states are where the application spends most of its time

- ◆ Execution within main loop "forks" depending on the state of USB, creating **alternate main loops**
- ◆ Thus, USB state becomes a central part of managing software flow
- ◆ This framework excels when the device behaves differently in each state!
- ◆ For cases where system only cares about one state, `connectionState()` fxn could be called from `IF{} stmt`
- ◆ Most common non-RTOS solution – it's used in many of the USB examples provided with the API

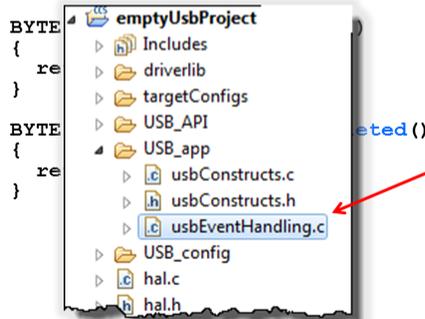
Built-in main() loop framework

2 Respond to 'Stack' Events

```

BYTE USB_handleVbusOnEvent() {
    if (USB_enable() == kUSB_succeed) //Connect when VBUS appears
    {
        USB_start();
    }
    return FALSE;
}

BYTE USB_handleSuspendEvent()
{
    return TRUE;
}
    
```



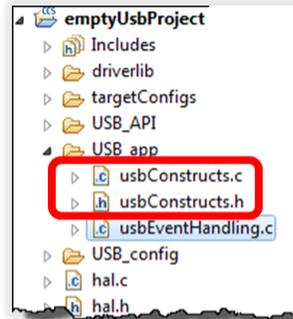
- ◆ The API calls “event handlers” when major events occur
- ◆ These functions are essentially ISR's, as most are called from interrupts
 - ◆ Seven USB-level events
 - ◆ Three CDC events
 - ◆ Three HID events
- ◆ If you're comfortable with the term *callbacks*, these are similar – except we pre-defined the names in the API
- ◆ The app can define behavior here; i.e. you can modify this code as needed – but keep handlers short!
- ◆ If MSP430 was interrupted from LPM:
 - ◆ Return 'TRUE' keeps CPU awake upon returning to main()
 - ◆ Return 'FALSE' allows CPU to return to LPM

3 Construct Functions

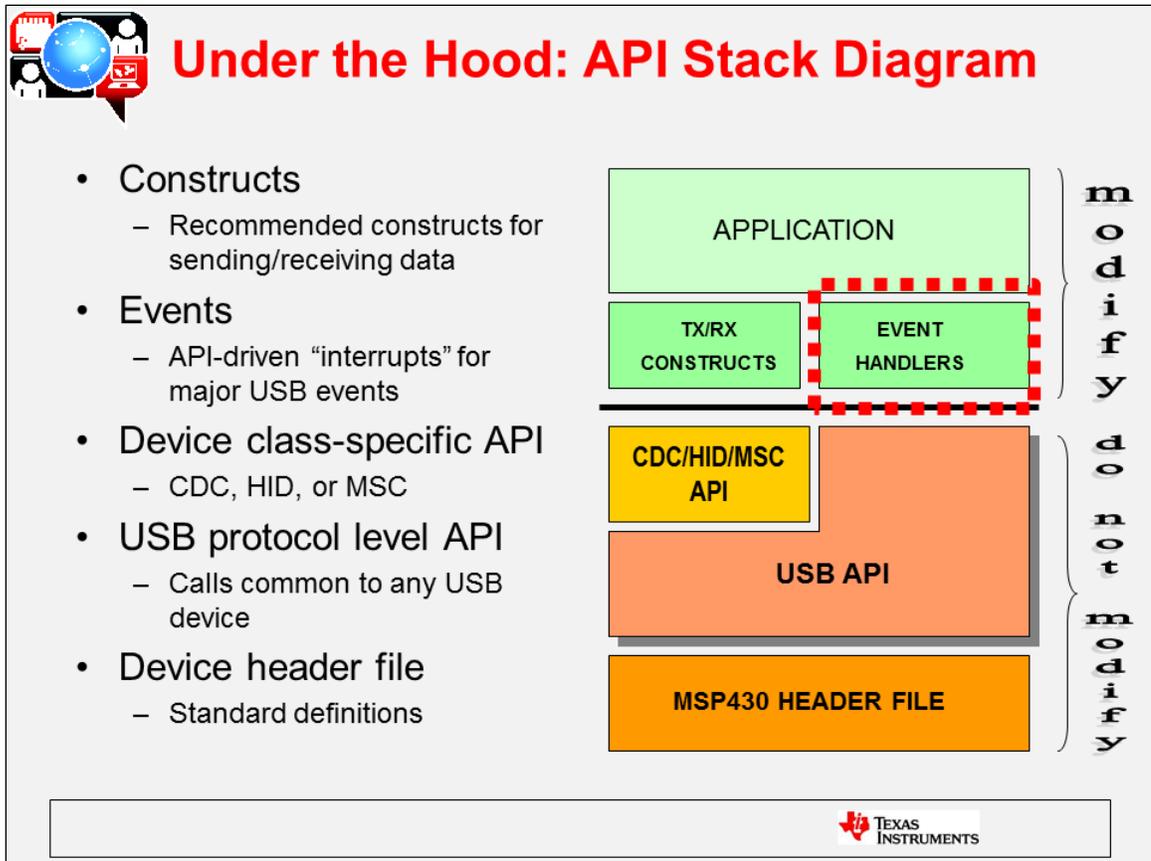
```

// From Example: C0_SimpleSend
convertTimeBinToASCII(timeStr);

if (cdcSendDataInBackground(
    timeStr, //Data to send
    9, //Size is 9 bytes
    CDC0_INTFNUM, //Send to intf#0
    1000)) //Retry 1000 times
{
    // ...
}
    
```



- ◆ Function begins USB send operation and returns immediately, while send occurs in background (i.e. asynchronous function)
- ◆ Retries will be attempted if the previous send hasn't completed
- ◆ If the bus isn't present, it does nothing and simply returns
- ◆ Constructs are defined in usbConstructs.c/h
- ◆ They are example code – you can use and/or modify them



Final Thoughts

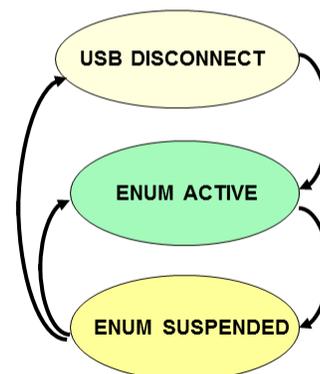
Error! Not a valid link.

How to Get Started with USB

1. **Start with example application from MSP430's USB Developers Package**
 - ♦ Find an example close to your needs and modify it
2. **Begin with the [emptyUsbProject](#) from the Developers Package (method used in Lab 7d)**
 - ♦ Empty project already contains all the needed code & lib's
 - ♦ It also provides a framework (i.e. 'template') to add your code into. This includes the common 'switch' call in main()
3. **Add the USB code to your existing project**
 - ♦ More work required to get app working
 - ♦ USB projects are often structured differently – you may need to re-work some code anyway
 - ♦ Please refer to documentation found in Developers Pkg for further discussion on this topic

Designing an Embedded USB App

- ♦ **Adding USB to existing app may mean re-thinking functionality:**
 - ♦ USB state often has a major impact on device behavior
 - ♦ Does it behave differently when attached to a host vs. not attached?
- ♦ **How does your app respond to the three primary USB states?**
- ♦ **In development, force O/S to reload drivers whenever you change I/F spec**
 - ♦ Delete Windows driver and then connect/disconnect dev to reload driver
 - ♦ Change PID every time you change I/F (e.g. everytime you run Descriptor Tool)
- ♦ **App should stay "fluid" to respond quickly to:**
 - ♦ USB host requests
 - ♦ Changes in bus state
 - ♦ Outside interrupts



Write “Fluid” Apps

- **A USB app should stay “fluid”**
 - Bus state may change at any time
 - While writing your app, always ask “What will happen if the bus is removed here?”
- **Call `USB_connectionState()` often**
 - Gives software a chance to adapt to its new situation
- **Be mindful of API return values**
 - They may indicate a lost bus
 - Otherwise, your code might wait forever for a response that isn't coming
- **Be wary of loops whose exit depends on an available bus**



Lab 8 – Using USB Devices

Lab 8 – USB Devices

- ◆ **Lab 8a – HID LED On/Off Toggle**
 - ◆ Set LED on/off/blinking from Windows PC via the USB serial port using the HID class
 - ◆ Uses HID host demo program supplied with USB Developers Package
- ◆ **Lab 8b – CDC LED On/Off Toggle**
 - ◆ Similar to Lab8a, but using CDC class to transfer the data
 - ◆ Host-side uses CCS serial Terminal (or Putty)
- ◆ **Lab 8c – Send Short Message via CDC**
 - ◆ Example sends a short message (i.e. time) to host via CDC class
 - ◆ Host-side uses CCS serial Terminal (or Putty)
- ◆ **Lab 8d – Send Pushbutton State to Host**
 - ◆ Starts by importing the Empty USB Example
 - ◆ You add code to read the state of the pushbutton and send it to the host (via HID)
 - ◆ Read data on host with serial terminal



Lab Topics

USB Devices	8-29
<i>Lab 8 – Using USB Devices.....</i>	<i>8-31</i>
<i>Lab 8a – LED On/Off HID Example</i>	<i>8-33</i>
<i>Lab 8b – LED On/Off CDC Example.....</i>	<i>8-36</i>
Play with the demo.....	8-39
<i>Lab 8c – CDC ‘Simple Send’ Example</i>	<i>8-41</i>
<i>Lab 8d – Creating a CDC Push Button App.....</i>	<i>8-43</i>
Import Empty USB Project Steps.....	8-43
Use the Descriptor Tool	8-44
Add ‘Custom’ Code to Project.....	8-47

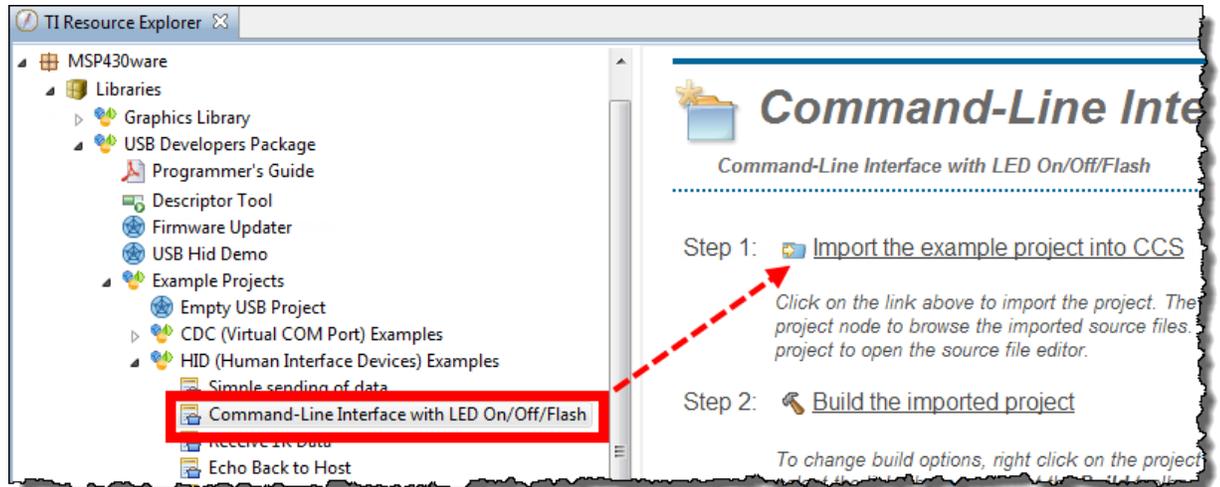
Lab 8a – LED On/Off HID Example

The MSP430 USB Developers Package contains an example which changes the state of an LED based on string commands sent from the USB host.

1. Import the following example into your workspace using TI Resource Explorer.

Help → Welcome to CCS

HID → *Command-Line Interface with LED On/Off/Flash*



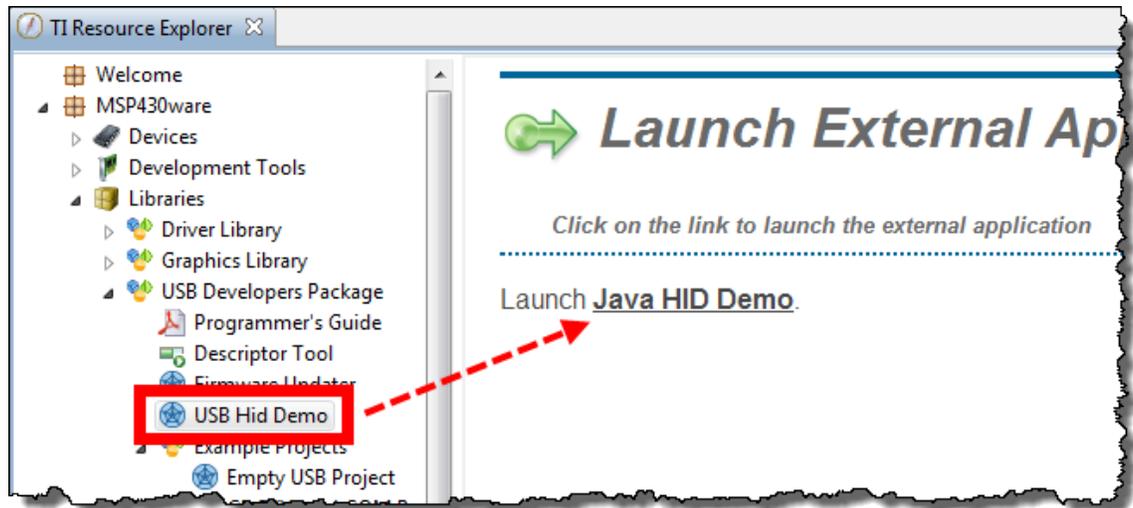
2. Build the project.

3. Launch the debugger and wait for the program to load to flash; then start the program running.

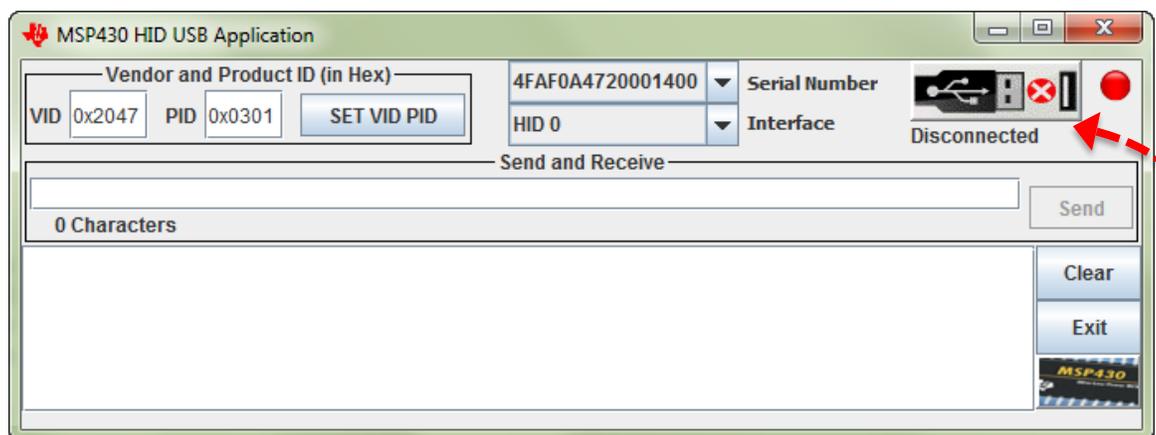
At this point, the MSP430 should start running the USB application. You may see Windows enumerate the USB device (in this case, your Launchpad); this usually appears as a popup message from the system tray saying that a USB device (“USB input device”) was enumerated.

4. Open the *USB HID Demo* program.

TI provides a simple communications utility which can communicate with a USB device implementing the HID-datapipe class. Essentially, this utility allows us to communicate with devices much like a serial terminal lets us talk with CDC (comm port) devices.



When the program opens, it will look like this:



We'll get back to this program in a minute. For now, return to CCS so that we can run the demo code.

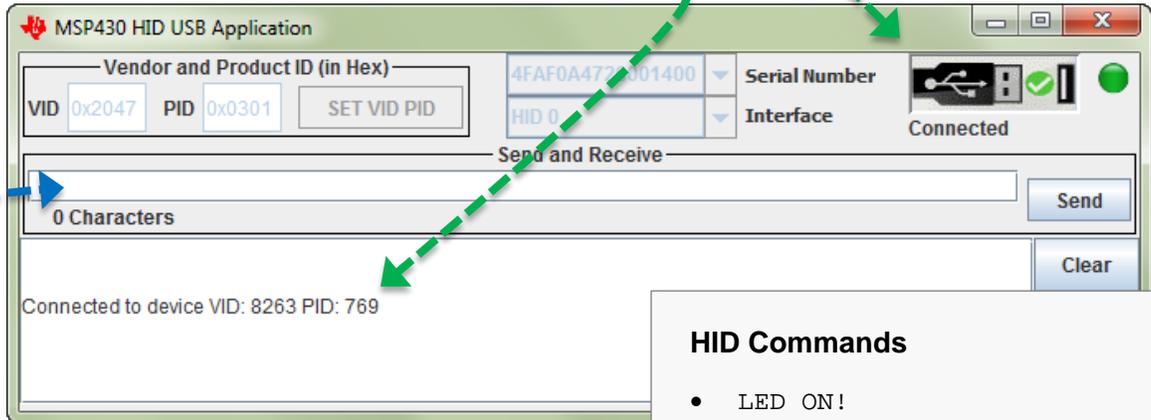
5. Switch back to the USB HID Demo application.

With the USB program running on the Launchpad, let's connect to it and send it commands.

6. Connect to the USB application.

Click the button that tells the HID app to find the USB device with the provided Vendor/Product IDs.

The app should now show **“Connected”** ...
as well as show connected in the log below ...



7. Play with the application.

After getting the device and Windows app running, what does it do? There are 4 commands you can use.

Enter a command and hit **Send**

8. In the HID USB application, disconnect from the USB device; then close the application.

9. Switch back to CCS and *Terminate* the debugger and close the project.

HID Commands

- LED ON!
- LED OFF!
- LED TOGGLE – SLOW!
- LED TOGGLE – FAST!

Don't forget to use the "!". The app uses this as an end-of-string character.

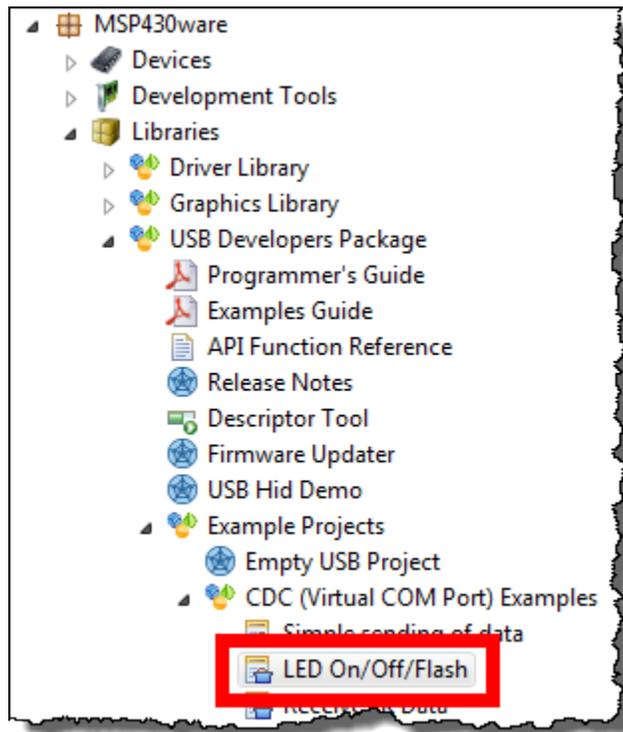
Along with the LED changing, you will see the command repeated back to the log.

Lab 8b – LED On/Off CDC Example

Our next program is another example from the MSP430 USB Developers Package. This program is a near duplicate of the previous lab – that is, it changes the state of an LED based on string commands sent from the USB host. In this example, though, the string commands are sent using the CDC class (versus the HID-datapipe class).

The advantage of the CDC class is that it can communicate with just about any Windows serial terminal application. The disadvantage, as you might remember from the discussion, is that Windows does not automatically load CDC based drivers – whereas Windows did this for us when using an HID class driver.

10. Import the CDC version of the **LED On/Off/Flash** project.



11. Build the project and launch the debugger.

12. Run the program.

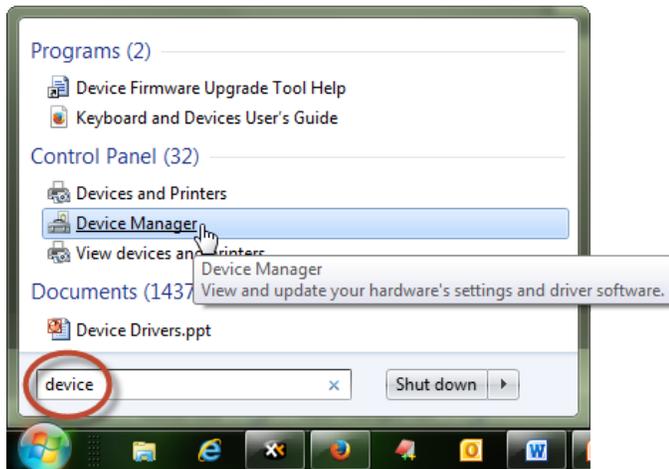


The first time you run the program, Windows may not be able to enumerate the USB CDC driver. You might see an error such as this pop up.

Why does this error occur? _____

13. Open the Windows Device Manager.

For Windows 7, the easiest way to start the device manager is to type “Device” into the Start menu:

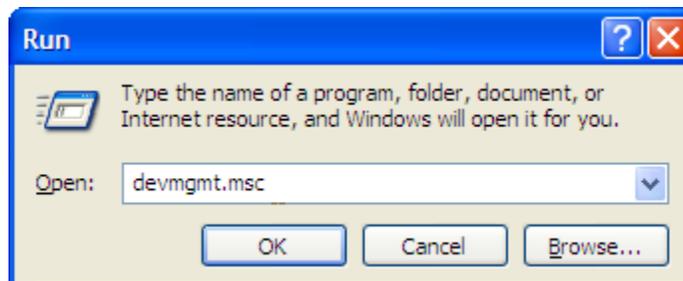


In most versions of Windows, such as Windows XP, you can also run the following program from a command line to start the Device Manager:

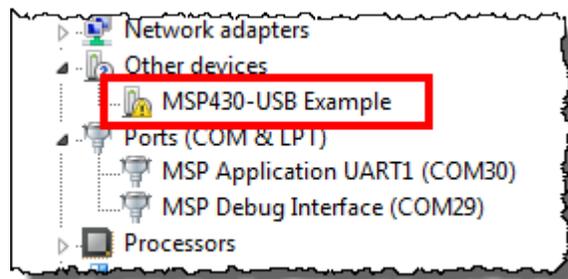
```
devmgmt.msc
```

On Windows XP, you can quickly run the command line from the Start Menu:

Start Menu → Run



You should find the a USB driver with a problem:



14. Update the MSP430-USB Example driver.

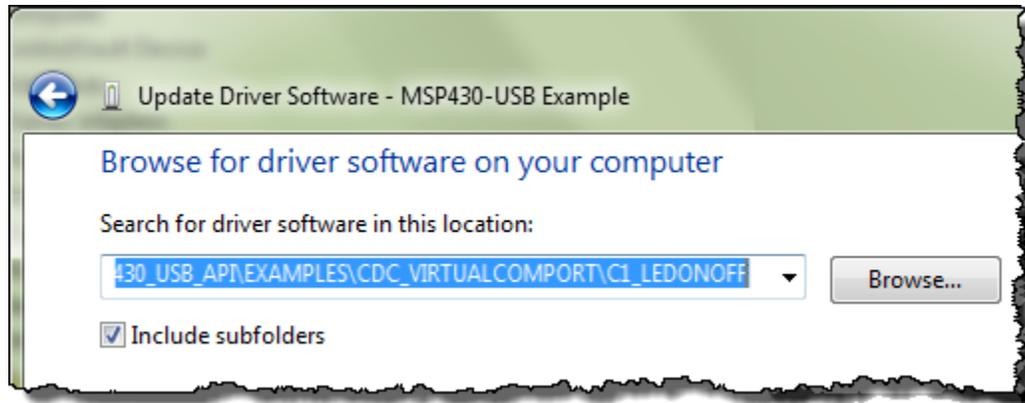
For Windows 7, the steps include:

Right-click on the driver → Update Driver Software...

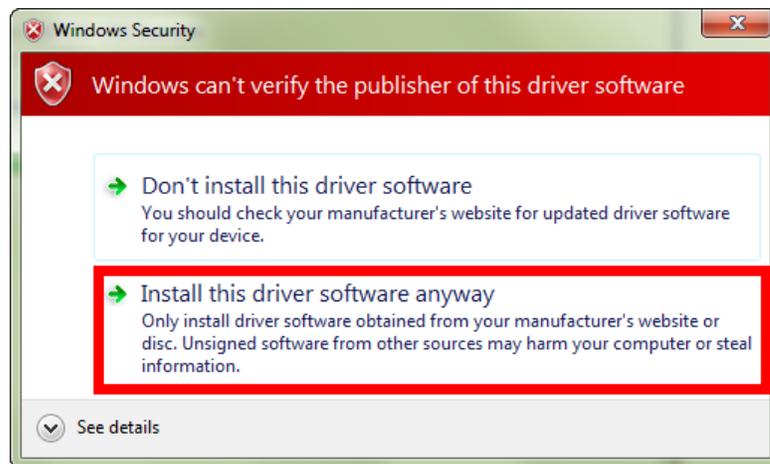
Click Browse my computer for driver software

Select the following (or wherever you installed the USB Developers Package)

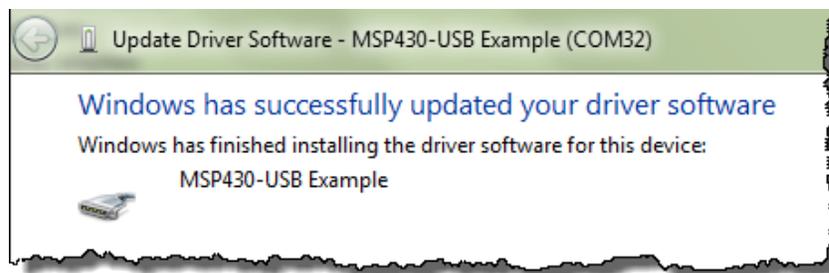
```
C:\TI\MSP430\MSP430USBDEVELOPERSPACKAGE_4_00_02\MSP430_USB_SOFTWARE\MSP430_USB_API\EXAMPLES\CDC_VIRTUALCOMPORT\C1_LEDONOFF
```



During the installation, the following dialog may appear. If so, choose to *Install* the driver.



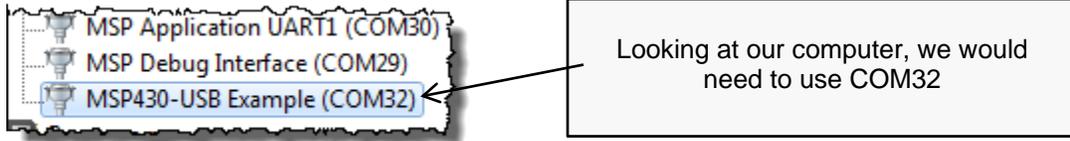
When complete you should see:



Note: The steps to install the USB CDC driver are also documented in the:

Examples_Guide_MSP430_USB.pdf
found in the documentation directory of the USB Developers Package.

15. In the Device Manager, write down the COM port associated with our USB driver:



What is your COM port = _____

Hint: When done, we suggest you minimize the Device Manager; thus, leaving it open in the background. It's quite possible you may need to check the drivers later on during these lab exercises.

Play with the demo

At this point, we should have:

- The USB device application running on the MSP430
- The appropriate Windows CDC driver loaded

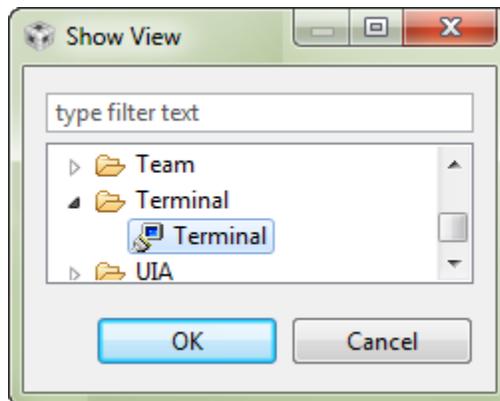
Before we can communicate with the device, though, we also need to open a serial terminal.

16. Open your favorite serial terminal and connect to the MSP430.

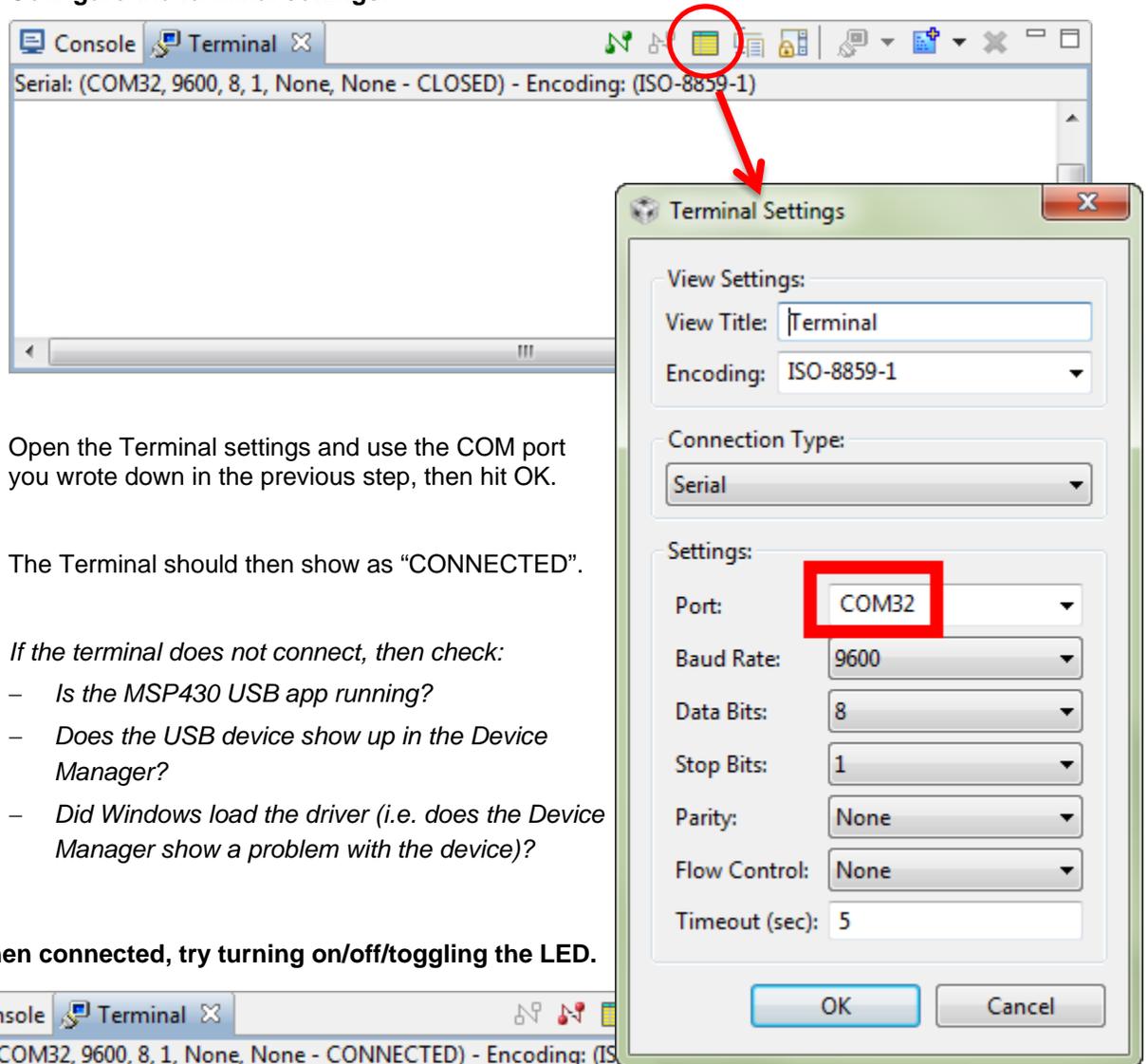
Putty and Tera Term are common favorites, but we'll provide directions for using the Terminal built into CCS.

a) Open the Terminal window.

Window → Show View → Other...



b) Configure the terminal settings:



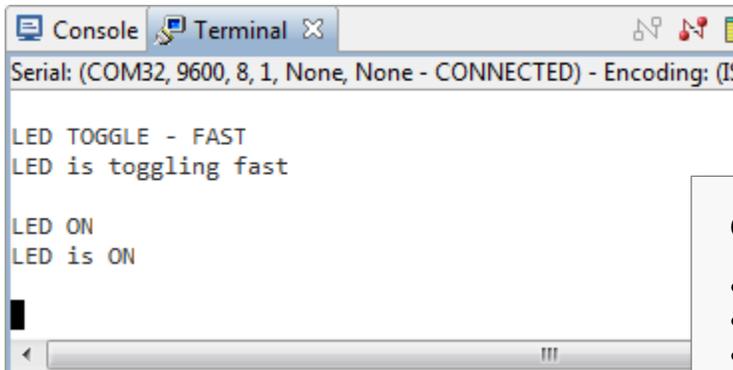
Open the Terminal settings and use the COM port you wrote down in the previous step, then hit OK.

The Terminal should then show as “CONNECTED”.

If the terminal does not connect, then check:

- Is the MSP430 USB app running?
- Does the USB device show up in the Device Manager?
- Did Windows load the driver (i.e. does the Device Manager show a problem with the device)?

17. When connected, try turning on/off/toggling the LED.



CDC Commands

- LED ON
- LED OFF
- LED TOGGLE – SLOW
- LED TOGGLE – FAST

Type one of these strings and then hit the <Enter> key.

Along with the LED changing, you will see the command repeated back to the term.

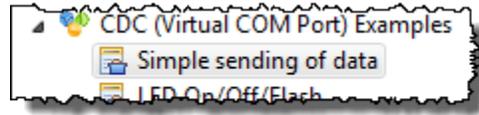
18. When done experimenting...

- Stop the terminal (hit red disconnect button).
- Terminate the debugger.
- Close the project.

Lab 8c – CDC ‘Simple Send’ Example

Let’s try one more simple application example before we build our own. This next example simply sends the time (from MSP430’s Real Time Clock) to a serial terminal.

19. Similar to our previous two examples, import the “Simple Sending of Data” project.



20. Build the project and launch the debugger.

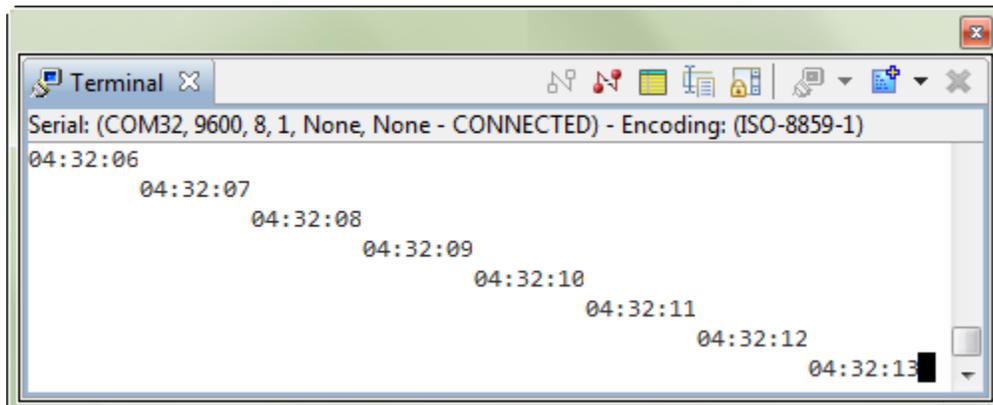
21. Start the program.

22. Wait for the USB device to enumerate.

If you’re not sure that Windows enumerated the device, check the Device Manager. If it does not enumerate, try Terminating the debugger, unplugging the Launchpad, then plugging it back into another USB port on your computer.

23. Once enumerated, start the Terminal again (by hitting the Green Connection button).

You should see the time printed (repeatedly) to the Terminal.



24. Once you are done watch time go by: disconnect the Terminal; Terminate the debugger (if you didn't do it in the last step).

25. (Optional) Review the code in this example. Here's a bit of the code from main.c:

```

VOID main(VOID)
{
    WDT_A_hold(WDT_A_BASE); //Stop watchdog timer

    // Minimum Vcore required for the USB API is PMM_CORE_LEVEL_2
    PMM_setVCore(PMM_BASE, PMM_CORE_LEVEL_2);

    initPorts(); // Config GPIOs for low-power (output low)
    initClocks(8000000); // MCLK=SMCLK=FLL=8MHz; ACLK=REFO=32kHz
    USB_setup(TRUE,TRUE); // Init USB; if a host is present, connect
    initRTC(); // Start the real-time clock

    __enable_interrupt(); // Enable interrupts globally

    while (1)
    {
        // Enter LPM0, which keeps the DCO/FLL active but shuts off the
        // CPU. For USB, you can't go below LPM0!
        __bis_SR_register(LPM0_bits + GIE);

        // If USB is present, send time to host. Flag set every sec.
        if (bSendTimeToHost)
        {
            bSendTimeToHost = FALSE;
            convertTimeBinToASCII(timeStr);

            // This function begins the USB send operation, and immediately
            // returns, while the sending happens in the background.
            // Send timeStr, 9 bytes, to intf #0 (which is enumerated as a
            // COM port). 1000 retries. (Retries will be attempted if the
            // previous send hasn't completed yet). If the bus isn't present,
            // it simply returns and does nothing.
            if (cdcSendDataInBackground(timeStr, 9, CDC0_INTFNUM, 1000))
            {
                _NOP(); // If it fails, it'll end up here. Could happen if
                // the cable was detached after the connectionState()
            } // check, or if somehow the retries failed
        }
    } //while(1)
} //main()

// Convert the binary globals hour/min/sec into a string, of format "hr:mn:sc"
// Assumes str is a nine-byte string.
VOID convertTimeBinToASCII(BYTE* str)
{
    BYTE hourStr[2], minStr[2], secStr[2];

    convertTwoDigBinToASCII(hour, hourStr);
    convertTwoDigBinToASCII(min, minStr);
    convertTwoDigBinToASCII(sec, secStr);

    str[0] = hourStr[0];
    str[1] = hourStr[1];
    str[2] = ':';
    str[3] = minStr[0];
    str[4] = minStr[1];
    str[5] = ':';
    str[6] = secStr[0];
    str[7] = secStr[1];
    str[8] = '\n';
}

```

Lab 8d – Creating a CDC Push Button App

We have experimented with three example USB applications. It's finally time to build one from "scratch". Well, not really from scratch, since we can start with the "Empty USB Example".

The goal of our application is to send the state of the Launchpad button to the PC via USB – using the HID Datapipe interface. Thus, we'll use a HID class driver. This application will borrow from a number of programs we've already written:

GPIO – We will read the push button and light the LED when it is pushed. Also, we'll send "DOWN" when it's down and "UP" when it's up.

Timer – We'll use a timer to generate an interrupt every second. In the Timer ISR we'll set a flag. When the flag is TRUE, we'll read the button and send the proper string to the host.

HID Simple Send Example – we'll borrow a bit of code from the HID example we just ran to 'package' up our string and send it via USB to the host.

Finally, we're going to start by following the first 3 steps provided in TI Resource Explorer for the **Empty USB Example**.

Import Empty USB Project Steps

1. Import the Empty USB Project.

As it states in the Resource Explorer, DO NOT RENAME the project (yet).

The screenshot shows the TI Resource Explorer interface. On the left, a tree view lists various resources, with 'Empty USB Project' under 'Example Projects' highlighted by a red rectangular box. The main content area on the right is titled 'Empty USB project' and contains the following text:

Creates an empty USB project to start development

These are the steps to import the project, use the descriptor tool, build the project

Step 1: [Import the example project into CCS \(Do not rename\)](#)
Click on the link above to import the project. The imported project is available in the Project Explorer. To modify source code, double clicks on the source file within the project.

Step 2: [Launch The Descriptor Tool](#)
Design your USB device in the Descriptor tool and then generate Descriptor Tool files into the project.

Step 3: Rename the project (if needed)
Now that the project is imported and the USB descriptor made, you can rename the project.

Use the Descriptor Tool

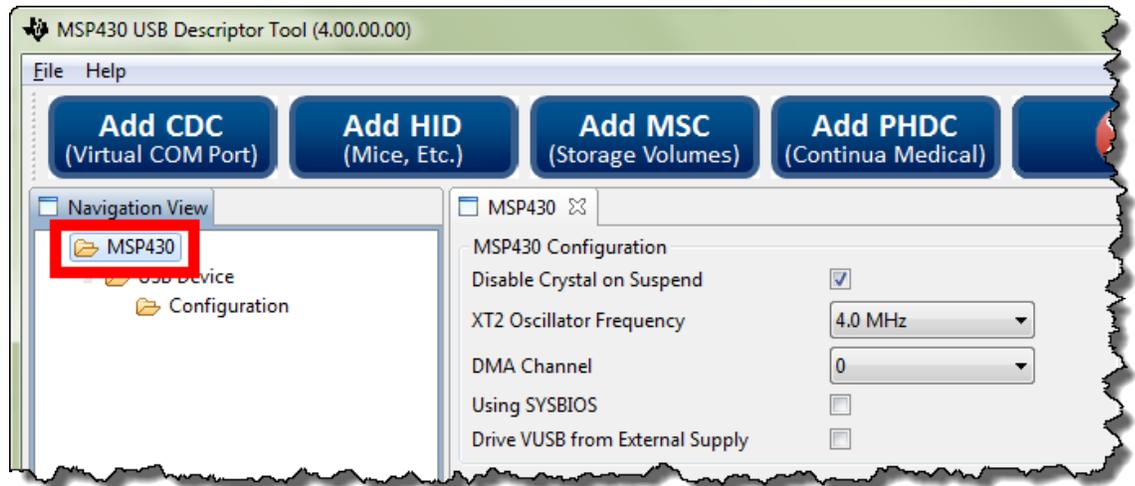
2. Launch the Descriptor Tool.

-  Just as the Resource Explorer directs us, launch the Descriptor Tool. The easiest way to do this is to click the link as shown above.

3. Generate descriptor files using the Descriptor Tool.

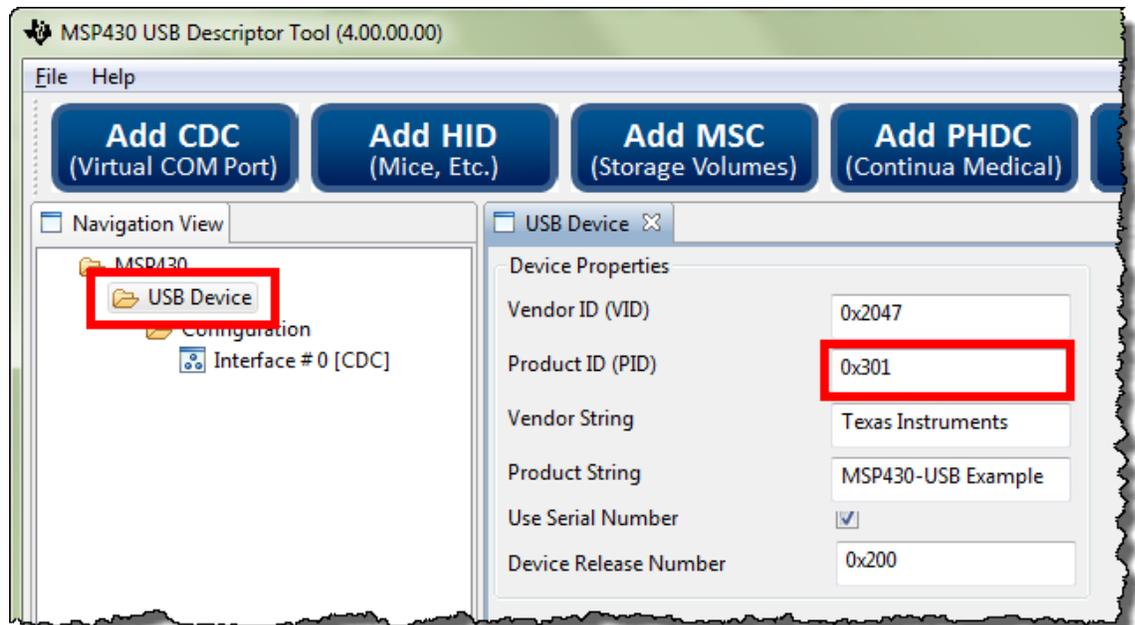
We will take a quick look at the organization levels in the tool. In most cases, we will use the tools defaults.

a) MSP430 level ... use the defaults.



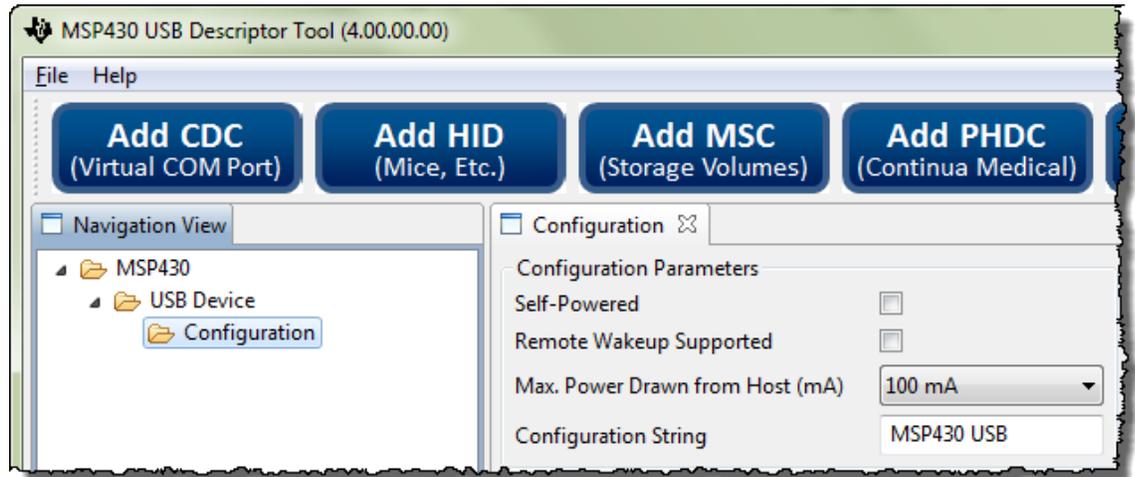
b) USB Device ... MSP430-Button Example

We suggest changing the Product String – so it'll be easier to see that it is different than previous examples. Also, we suggest changing the PID (we picked '301' arbitrarily). For a real design, you might end up purchasing the VID/PID (or obtain a free PID from TI).



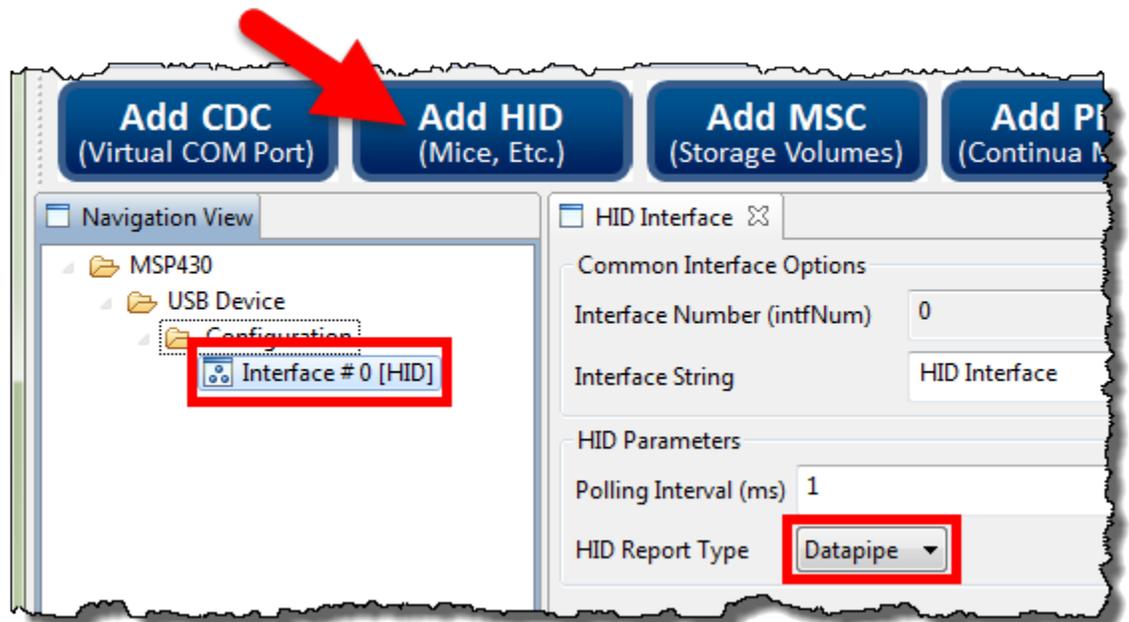
c) Configuration

Nothing to do on the configuration screen.



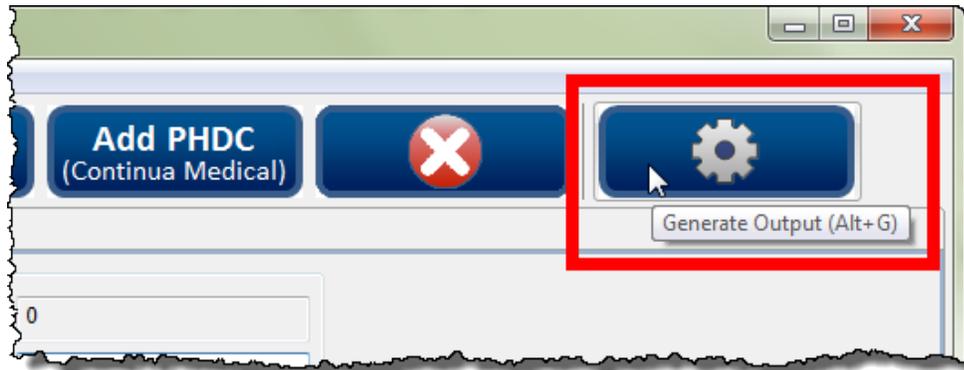
d) Add HID Interface

Once again, we chose to vary the string so that it would be a little bit less generic.



e) Click the button to generate the descriptor files.

Notice they get written to your empty project. (This is the reason we were asked not to change the name until after we had used the Descriptor Tool.)



The files should be saved to our “empty” project ... but if you’re asked where to save them, choose the USB_config folder:

C:\msp430_workshop\F5529_usb\workspace\emptyUsbProject\USB_config\

f) Save the Descriptor Tool settings.

While not required, this is handy if you want to open the tool and view the settings at some later point in time. Notice that ‘Save’ puts the resulting .dat file into the same folder as our descriptor files.



Save to your emptyProject USB_config folder. This is a pretty good place for it, since this is where all of the descriptor files it generates are placed. For example:

C:\msp430_workshop\F5529_usb\workspace\emptyUsbProject\USB_config\

g) You can close the Descriptor Tool.

4. Rename the project to lab_08d_usb.

As you can see, the reason they didn’t want us to rename the project before now was that the descriptor tool generates files to the empty project.

5. Build, just to make sure we’re starting off with a ‘clean’ project.

Add ‘Custom’ Code to Project

6. Copy myTimer.c and myTimer.h (and the readme file) to the project folder.

We’ve already written the timer routine for you. (Look back to our Timer chapter if you want to know the details of how this code was developed.)

Right-click the project → Add Files...

Choose the three files from the location:

C:\msp430_workshop\F5529_usb\lab_08d_usb\

7. Open main.c and add a #include for the myTimer.h.

We suggest doing this somewhere below #include “driverlib.h”.

8. Add global variables.

These are used to capture (and send) the button up/down state.

```
char pbStr[5] = ""; // Stores the string to send
volatile unsigned short usiButton1 = 0; // Stores the button state
```

9. Add additional setup code.

We need to initialize an LED and pushbutton. We also need to call the initTimers() function that was just added to our project in a previous step.

```
GPIO_setAsOutputPin( GPIO_PORT_P4, GPIO_PIN7 );
GPIO_setAsInputPinWithPullUpresistor( GPIO_PORT_P2, GPIO_PIN1 );
initTimers();
```

10. Modify the low-power state of the program.

Search down toward the end of main() until you find the intrinsic that sets the program into low-power mode. Rather than using LPM3, we want to switch this to LPM0.

```
// __bis_SR_register(LPM3_bits + GIE);
__bis_SR_register(LPM0_bits + GIE);
```

Notes:

11. Add code to ST_ENUM_ACTIVE state.

The active state is where we want to put our communication code. (It only makes sense to that we send data to the host when we're actively connected.)

When connected, we will read the pin, set the Launchpad's LED and then construct a string to send to the host. Finally, we send the data to the host in the background; that is, we won't wait for a response – although we do set a timeout in our code below.

Note, it's the timer that wakes us up every second to check the state – and if connected, to through the routine below.

```
// If USB is present, sent the button state to host. Flag set every sec
if (bSend)
{
    bSend = FALSE;

    usiButton1 = GPIO_getInputPinValue ( GPIO_PORT_P2, GPIO_PIN1 );

    if ( usiButton1 == GPIO_INPUT_PIN_LOW ) {
        // If button is down, turn on LED
        GPIO_setOutputHighOnPin( GPIO_PORT_P4, GPIO_PIN7 );
        pbStr[0] = 'D';
        pbStr[1] = 'O';
        pbStr[2] = 'W';
        pbStr[3] = 'N';
        pbStr[4] = '\n';
    }
    else {
        // If button is up, turn off LED
        GPIO_setOutputLowOnPin( GPIO_PORT_P4, GPIO_PIN7 );
        pbStr[0] = 'U';
        pbStr[1] = 'P';
        pbStr[2] = ' ';
        pbStr[3] = ' ';
        pbStr[4] = '\n';
    }

    // This function begins the USB send operation, and immediately
    // returns, while the sending happens in the background.
    // Send pbStr, 5 bytes, to intf #0 (which is enumerated as a
    // HID port). 1000 retries. (Retries will be attempted if the
    // previous send hasn't completed yet). If the bus isn't present,
    // it simply returns and does nothing.
    if (cdcSendDataInBackground((BYTE*)pbStr, 5, HID0_INTFNUM, 1000))
    {
        _NOP(); // If it fails, it'll end up here. Could happen if
                // the cable was detached after the connectionState()
                // check, or if somehow the retries failed
    }
}
```

12. Add #include "USB_app/usbConstructs.h".

We need to use this header file since it supports the `hidSendDataInBackground()` function we are using to send data via USB.

13. Build the program and launch debugger.

14. Start your program and open the USB HID demo tool.

You can either run the program from within the debugger – or – terminate the debugger and unplug and then plug the Launchpad back in. In either case, your USB program should be running.

We need to use the HID tool to view the communications coming from the Launchpad. As we mentioned earlier, it acts as a “terminal” for our HID Datapipe datastream.

If you cannot remember how to open it, please refer back to Step 4 on page 8-34.

Hint: You might have to set the PID depending upon the value you selected while using the Descriptor tool.

15. Verify your program works

Once the the driver is loaded and working properly, open your Terminal, making sure to use the proper comm port. *(As a reminder, all of these steps we discussed earlier in this chapter.)*

At this point:

- The Red LED should be blinking on/off.
- The Green LED should light when Button1 is pushed ...
- ... and the state of the button should be written to the HID Terminal.

Remember that the code only tests the button once per second. So, you will need to hold (or release) it for more than a second for it to take effect.

Using Energia (Arduino)

Introduction



This chapter of the MSP430 workshop explores Energia, the Arduino port for the Texas Instruments Launchpad kits.

After a quick definition and history of Arduino and Energia, we provide a quick introduction to Wiring – the language/library used by Arduino & Energia.

Most of the learning comes from using the Launchpad board along with the Energia IDE to light LED's, read switches and communicate with your PC via the serial connection.

Learning Objectives, Requirements, Prereq's

Prerequisites & Objectives

- ◆ Prerequisites
 - ◆ Basic knowledge of C language
 - ◆ Basic understanding of using a C library and header files
 - ◆ This chapter doesn't explain clock, interrupt, and GPIO features in detail, this is left to the other chapters in the MSP430 workshop
- ◆ Requirements - Tools and Software
 - ◆ Hardware
 - ◆ *Windows (XP, 7, 8) PC with available USB port*
 - ◆ *MSP430F529 Launchpad*
 - ◆ Software
 - ◆ *Energia Download*
 - ◆ *Launchpad drivers*
 - ◆ *(Optional) MSP430ware / Driverlib*
- ◆ Objectives
 - ◆ Define 'Arduino' and describe what it was created for
 - ◆ Define 'Energia' and explain what it is 'forked' from
 - ◆ Install Energia, open and run included example sketches
 - ◆ Use serial communication between the board & PC
 - ◆ Add an external interrupt to an Energia sketch
 - ◆ Modify CPU registers from an Energia sketch

Already installed, if you have installed CCSv5.x

Chapter Topics

Using Energia (Arduino)	9-1
<i>What is Arduino</i>	9-3
<i>Energia</i>	9-4
<i>Programming Energia (and Arduino)</i>	9-7
Programming with 'Wiring'	9-7
Wiring Language/Library Reference	9-8
How Does 'Wiring' Compare?.....	9-9
Hardware pinout.....	9-10
<i>Energia IDE</i>	9-12
Examples, Lots of Examples.....	9-13
<i>Energia/Arduino References</i>	9-14

What is Arduino

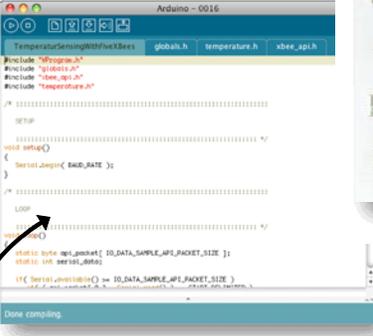
Physical Computing ... Hardware Hacking ... a couple of the names given to Arduino.

- *Our home computers are great at communicating with other computers and (sometimes) with us, but they have no idea what is going on in the world around them. Arduino, on the other hand, is made to be hooked up to sensors which feed it physical information.*¹ These can be as simple as pressing a button, or as complex as using ultrasound to detect distance, or maybe having your garage door tweet every time it's opened.
- *So the Arduino is essentially a simple computer with eyes and ears. Why is it so popular? Because the hardware is cheap, it's easy to program and there is a huge web community, which means that beginners can find help and download myriad programs.*¹

What is Arduino?

Tools

IDE: write, compile, upload



Code

'Wiring' Language includes:

- ◆ C/C++ software
- ◆ Arduino library of functions

Hardware

Open source μ C boards with pins and I/O



- ◆ **Physical Computing**
Software that interacts with the real world
- ◆ **Open-source ecosystem**
Tools, Software, Hardware (Creative Commons)
- ◆ **Popular solution for...**
Open-source programmers, hobbyists, rapid prototyping



- *The idea is to write a few lines of code, connect a few electronic components to the Wiring hardware and observe how a light turns on when person approaches it, write a few more lines, add another sensor, and see how this light changes when the illumination level in a room decreases. This process is called sketching with hardware; explore lots of ideas very quickly, select the more interesting ones, refine and produce prototypes in an iterative process.*²

In the end, Arduino is basically an ecosystem for easy, hardware-oriented, real-world programming. It combines the Tools, Software and Hardware for talking to the world.

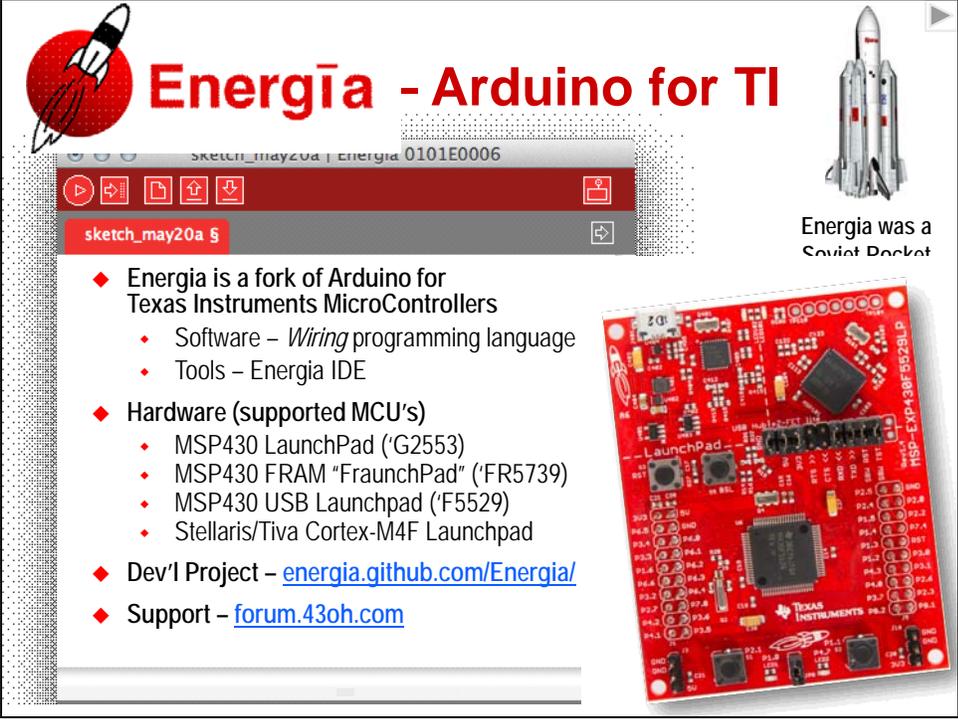
¹ <http://www.wired.com/gadgetlab/2008/04/just-what-is-an/>

² http://en.wikipedia.org/wiki/Wiring_%28development_platform%29

Energia

/ener'gia/ ; e·ner·gi·a

Energia (Russian: Энергия, Energiya, "Energy") was a Soviet rocket that was designed by NPO Energia to serve as a heavy-lift expendable launch system as well as a booster for the Buran spacecraft.³



Energia - Arduino for TI

sketch_may20a §

- ◆ Energia is a fork of Arduino for Texas Instruments MicroControllers
 - ◆ Software – *Wiring* programming language
 - ◆ Tools – Energia IDE
- ◆ Hardware (supported MCU's)
 - ◆ MSP430 LaunchPad ('G2553)
 - ◆ MSP430 FRAM "FraunchPad" ('FR5739)
 - ◆ MSP430 USB Launchpad ('F5529)
 - ◆ Stellaris/Tiva Cortex-M4F Launchpad
- ◆ Dev'l Project – energia.github.com/Energia/
- ◆ Support – forum.43oh.com

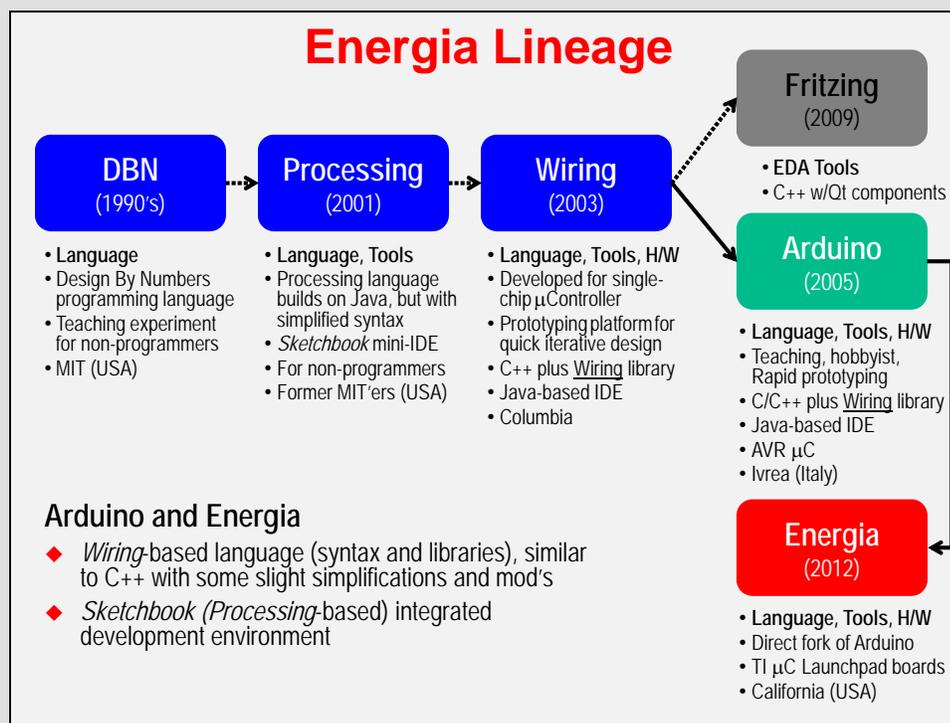
Energia was a Soviet Rocket

Energia is a rapid electronics prototyping platform for the Texas Instruments msp430 LaunchPad. Energia is based on Wiring and Arduino and uses the Processing IDE. It is a fork of the Arduino ecosystem, but centered around the popular TI microcontrollers: MSP430 and ARM Cortex-M4F.

Similar to it's predecessor, it an open-sourced project. It's development is community supported, being hosted on github.com.

³ <http://en.wikipedia.org/wiki/Energia>

Sidebar – Energia Lineage



Design By Numbers (or DBN programming language) was an influential experiment in teaching programming initiated at the MIT Media Lab during the 1990s. Led by John Maeda and his students they created software aimed at allowing designers, artists and other non-programmers to easily start computer programming. The software itself could be run in a browser and published alongside the software was a book and courseware.⁴

Processing (2001) - One of the stated aims of Processing is to act as a tool to get non-programmers started with programming, through the instant gratification of visual feedback.⁵

This process is called sketching with hardware; explore lots of ideas very quickly, select the more interesting ones, refine and produce prototypes in an iterative process.

Wiring (2003)⁶ - The Wiring IDE is a cross-platform application written in Java which is derived from the IDE made for the Processing programming language. It is designed to introduce programming and sketching with electronics to artists and designers. It includes a code editor ... capable of compiling and uploading programs to the board with a single click.

The Wiring IDE comes with a C /C++ library called "Wiring", which makes common input/output operations much easier. Wiring programs are written in C/C++, although users only need to define two functions to make a runnable program: `setup()` and `loop()`.

When the user clicks the "Upload to Wiring hardware" button in the IDE, a copy of the code is written to a temporary file with an extra include header at the top and a very simple `main()` function at the bottom, to make it a valid C++ program.

⁴ http://en.wikipedia.org/wiki/Design_By_Numbers_%28programming_language%29

⁵ [http://en.wikipedia.org/wiki/Processing_\(programming_language\)](http://en.wikipedia.org/wiki/Processing_(programming_language))

⁶ http://en.wikipedia.org/wiki/Wiring_%28development_platform%29

Energia Lineage (cont'd)

Arduino⁷ - In 2005, in Ivrea, Italy, a project was initiated to make a device for controlling student-built interaction design projects with less expense than with other prototyping systems available at the time. Founders Massimo Banzi and David Cuartielles named the project after Arduin of Ivrea, the main historical character of the town.

The Arduino project is a fork of the open source Wiring platform and is programmed using a Wiring-based language (syntax and libraries), similar to C++ with some slight simplifications and modifications, and a Processing-based integrated development environment.

Energia (2012) – As explained in the previous section of this chapter, Energia is a fork of Arduino which utilizes the Texas Instruments microcontroller Launchpad development boards.

Fritzing (2009)⁸ - An open-source initiative to support designers, artists, researchers and hobbyists to take the step from physical prototyping to actual product.

It's essentially an Electronic Design Automation software with a low entry barrier, suited for the needs of designers and artists. It uses the metaphor of the breadboard, so that it is easy to transfer your hardware sketch to the software. From there it is possible to create PCB layouts for turning it into a robust PCB yourself or by help of a manufacturer.

⁷ <http://en.wikipedia.org/wiki/Arduino>

⁸ [http:// Fritzing.org](http://Fritzing.org)

Programming Energia (and Arduino)

Programming with ‘Wiring’

Energia / Arduino Programming

- ◆ Arduino programs are called *sketches*
 - From the idea that we’re...
 - Sketching with hardware*
- ◆ Sketches require only two functions to run cyclically:
 - setup()
 - loop()
- ◆ Are C/C++ programs that can use Arduino’s *Wiring* library
 - Library included with IDE
- ◆ If necessary, you can access H/W specific features of μ C, but that hurts portability
- ◆ Blink is μ C’s ‘Hello World’ ex.
 - ‘Wiring’ makes this simple
 - Like most first examples, it is not optimized

```

sketch_may20a | Energia 0101E0006
sketch_may20a
// Most boards have LED and resistor connected
// between pin 14 and ground (pinout on later slide)
#define LED_PIN 14

void setup () {
    // enable pin 14 for digital output
    pinMode (LED_PIN, OUTPUT);
}

void loop () {
    digitalWrite (LED_PIN, HIGH); // turn on LED
    delay (1000); // wait one second (1000ms)
    digitalWrite (LED_PIN, LOW); // turn off LED
    delay (1000); // wait one second
}

```

Programming in Arduino is relatively easy. Essentially, it is C/C++ programming, but the *Wiring* library simplifies many tasks. As an example, we use the *Blink* sketch (i.e. program) that is one of examples that is included with Arduino (and Energia). In fact, this example is so ubiquitous that most engineers think of it as “*Hello World*” of embedded programming.

How does the ‘Wiring’ library help to make things easier? Let’s examine the Blink code above:

- A sketch only requires two functions:
 - **setup()** – a function run once at the start of a program which can be used to define initial environment settings
 - **loop()** – a function called repeatedly until the board is powered off
- Reading and Writing pins (i.e. General Purpose Input Output – GPIO) is encapsulated in three simple functions: one function defines the I/O pin, the other two let you read or write the pin. In the example above, this allows us to turn on/off the LED connected to a pin on our microcontroller.
- The **delay()** function makes it simple to pause program execution for a given number of microseconds. In fact, in the Energia implementation, the delay() function even utilizes a timer which allows the processor to go into low power mode while waiting.
- Finally, which not shown here, Arduino/Energia makes using the serial port as easy as using printf() in standard C programs.

About the only difference between Arduino and Energia programming is that you might see some hardware specific commands in the sketch. For example, in one of the later lab exercises, you will see how you can change the clock source for the TI MSP430 microcontroller. Changing clocks is often done on the MSP430 so that you can balance processing speed against long battery life.

Wiring Language/Library Reference

What commands are available when programming with 'Wiring' in Arduino and Energia?

Arduino provides a language reference on their website. This defines the operators, controls, and functions needed for programming in Arduino (and Energia).⁹ You will also find a similar HTML reference available in the Energia installation zip file.

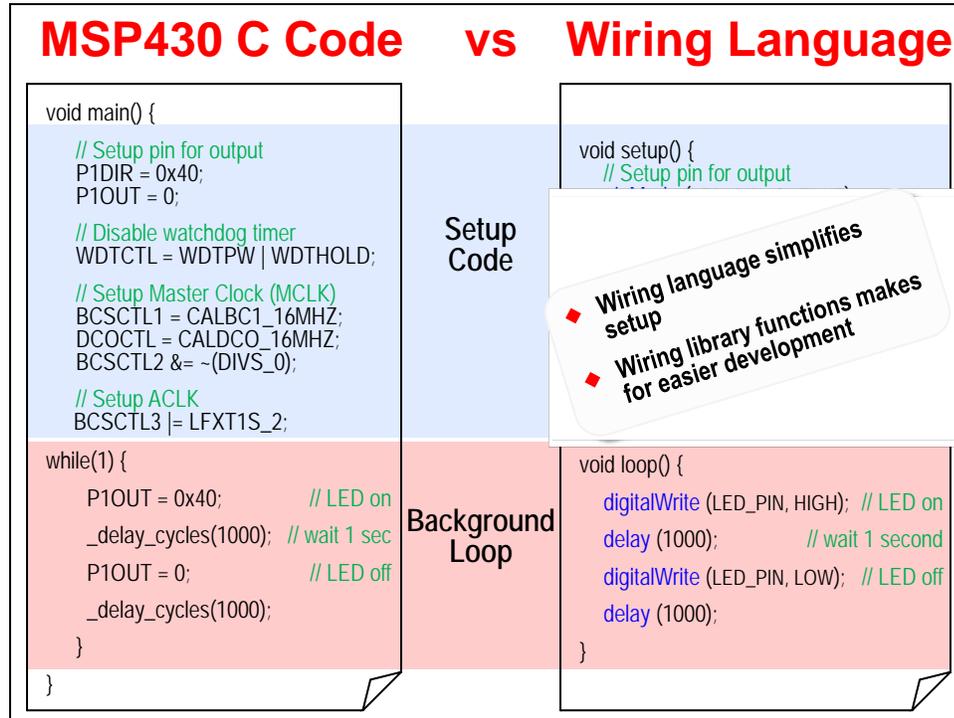
The screenshot shows the 'Wiring Library Reference' page. At the top, there is a navigation bar with links: Home, Download, Getting Started, Reference, Getting Help, FAQ, Projects Using Energia, and Contact Us. The main heading is 'Language Reference'. Below this, a note states: 'Energia programs can be divided in three main parts: *structure*, *values* (variables and constants), and *functions*.' The page is organized into three columns:

- Structure**
 - [setup\(\)](#)
 - [loop\(\)](#)
- Control Structures**
 - [if](#)
 - [if...else](#)
 - [for](#)
 - [switch case](#)
 - [while](#)
 - [do... while](#)
 - [break](#)
 - [continue](#)
 - [return](#)
 - [goto](#)
- Further Syntax**
 - [;](#) (semicolon)
- Variables**
 - Constants**
 - [HIGH](#) | [LOW](#)
 - [INPUT](#) | [OUTPUT](#)
 - [INPUT_PULLUP](#) | [INPUT_PULLDOWN](#)
 - [true](#) | [false](#)
 - [integer constants](#)
 - [floating point constants](#)
 - Data Types**
 - [void](#)
 - [boolean](#)
 - [char](#)
 - [unsigned char](#)
 - [byte](#)
 - [int](#)
 - [unsigned int](#)
- Functions**
 - Digital I/O**
 - [pinMode\(\)](#)
 - [digitalWrite\(\)](#)
 - [digitalRead\(\)](#)
 - Analog I/O**
 - [analogReference\(\)](#)
 - [analogRead\(\)](#)
 - [analogWrite\(\)](#) - *PWM*
 - Advanced I/O**
 - [tone\(\)](#)
 - [noTone\(\)](#)
 - [shiftOut\(\)](#)
 - [shiftIn\(\)](#)

⁹ <http://arduino.cc/en/Reference/HomePage>

How Does 'Wiring' Compare?

How does the 'Wiring' language compare to standard C code?



This comparison helps to demonstrate the simplicity of programming with Energia. As stated before, this can make for very effective rapid prototyping.

Later, during one of the lab exercises, we will examine some of the underpinnings of *Wiring*. Although the language makes programming easier, the same actual code is required for both sides of this diagram. In the case of *Wiring*, this is encapsulated by the language/library. You will see later on where this is done; armed with this knowledge, you can change the default values defined by the folks who ported Arduino over to Energia for the TI microcontrollers.

The remaining colored items show how various pins are used for digital, analog or communications purposes. The color legend on the right side of the diagram demonstrates the meaning of the various colors.

- **Green** indicates that you can use the associated pins with the *digitalRead()* and *digitalWrite()* functions.
- **Purple** is similar to Green, though you can also use the *analogWrite()* function with these pins.
- **Yellow**, **Orange**, and **Red** specify these pins are used for serial communication: UART, I2C, and SPI protocols, respectively.
- Finally, **Blue** demonstrates which pins are connected to the MSP430's ADC (analog to digital converter).

Should you do Pullups or Not?

To reduce power consumption, MSP430 Value-Line Launchpads (version V1.5 and later) are shipped without pull-up resistors on PUSH2 (S2 or P1_3 or pin 5). This saves (77uA) if port P1_3 is driven LOW. (On your LaunchPad just below the "M" in the text "MSP-EXP430G2" see if R34 is missing.) For these newer launchpads, sketches using PUSH2 should enable the internal pull-up resistor in the MSP430. This is a simple change; for example:

```
pinMode(PUSH2, INPUT); now looks like pinMode(PUSH2, INPUT_PULLUP);
```

Hardware Pin References

As stated above, the Energia wiki (<https://github.com/energia/Energia/wiki/Hardware>) and Energia site (http://energia.nu/Guide_MSP430F5529LaunchPad.html) shows these pin mapping diagrams for each of the Energia supported boards. You can also refer to the source code which defines this pin mapping; look for `Energia/hardware/msp430/variants/launchpad/pins_energia.h`. This header file can be found on [github](#), or in the files installed with Energia.

Sidebar

How can some 'pins' be connected to various pieces of hardware? (For example, PUSH2 and A3 (analog input 3) are both mapped to pin 5.)

Well, most processors today have *multiplexed* pins; i.e. each pin can have multiple functionality. While a given 'pin' can only be used for one function at a time, the chip designers give users many options to choose from. In an ideal world, we could just put as many pins as we want on a device; but unfortunately this costs too much, therefore multiplexing is a common cost/functionality tradeoff.

Energia IDE

The Energia IDE (integrated debugger and editor; integrated development environment) has been written in Java. This is how they can provide versions of the tools for multiple host platforms (Windows, Mac, Linux).

Energia Debugger

Verify/Compile
Download

New
Open
Save

- ◆ **Installation**
 - ◆ Simply unzip Energia package
 - ◆ Everything is included: debugger, libraries, board files, compilers
- ◆ **Download** button...
 - ◆ Performs compile and downloads the program to the target
- ◆ **Debugging** – Use common open-src methods
 - ◆ Write values to serial port: `Serial.println()`
 - ◆ Toggle pins & watch with o-scope

Installation of the tools couldn't be much simpler – unzip the package ... that's it. (Though, if you have not already installed TI's Code Composer Studio IDE, you may have to install drivers so that the Energia debugger can talk to the TI Launchpad board.)

Editing code is straightforward. Syntax highlighting, as well as brace matching help to minimize errors.

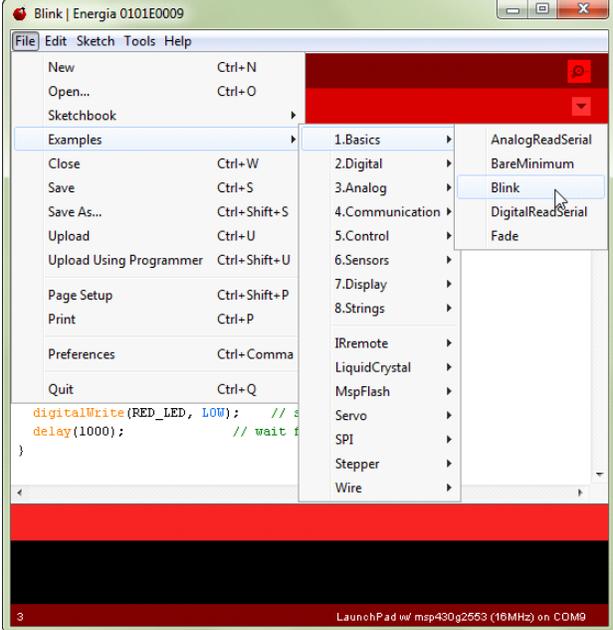
Compiling and **downloading** the program is as simple as clicking the *Download* button.

Debugging code is handled in the common, open-source fashion: `printf()` style. Although, rather than using `printf()`, you can use the Serial print functions to keep track of what is going on with your programs. Similarly, we often use LED's to help indicate status of program execution. And, if you have an oscilloscope or logic analyzer, you can also toggle other GPIO pins to evaluate the runtime state of your program sketches. (*We explore using LED's and serial communications in the upcoming lab exercises.*)

Examples, Lots of Examples

Energia ships with many examples. These are great for getting started with programming – or when trying to learn a new functionality. Our upcoming lab exercises will follow with this tradition of starting from these simple examples.

Energia Sketches (Examples)



```
digitalWrite(LED_RED, LOW); // set
delay(1000); // wait 1 second
}
```

- ◆ Basic Sketches
 - ◆ Blink is the 'hello world' of micro's
 - ◆ BareMinimum is just setup() and loop()
- ◆ Selecting example...
 - ◆ Opens sketch in debugger window
 - ◆ Click download to compile, download and run

Energia/Arduino References

There are many more Arduino references that could possibly be listed here, but this should help get you started.

Where To Go For More Information

◆ Energia

- Home: <http://energia.nu/>
- Download: <http://energia.nu/download/>
- Wiki: <https://github.com/energia/Energia/wiki>
- Getting Started: <https://github.com/energia/Energia/wiki/Getting-Started>
- Support Forum: <http://forum.43oh.com/forum/28-energia/>

◆ Launchpad Boards

- MSP430: <http://www.ti.com/tool/msp-exp430g2> (wiki) (eStore)
- ARM Cortex-M4F: [Launchpad](#) [Wiki](#) [eStore](#)

◆ Arduino:

- Site: <http://www.arduino.cc/>
- Comic book: <http://www.jodyculkin.com/.../arduino-comic-latest3.pdf>

Energia

- Home: <http://energia.nu/>
- Download: <http://energia.nu/download/>
- Wiki: <https://github.com/energia/Energia/wiki>
- Supported Boards: <https://github.com/energia/Energia/wiki/Hardware>
(H/W pin mapping)
- Getting Started: <https://github.com/energia/Energia/wiki/Getting-Started>
- Support Forum: <http://forum.43oh.com/forum/28-energia/>

Arduino

- Site: <http://www.arduino.cc/>
- Comic book: <http://www.jodyculkin.com/.../arduino-comic-latest3.pdf>

Lab 9

This set of lab exercises will give you the chance to start exploring Energia: the included examples, the 'Wiring' language, as well as how Arduino has been adapted for the TI Launchpad boards.

The lab exercises begin with the installation of Energia, then give you the opportunity to try out the basic 'Blink' example included with the Energia package. Then we'll follow this by trying a few more examples – including trying some of our own.

Lab Exercises

Installing Energia

- A.** Blinking the LED
- B.** Pushing the Button
- C.** Serial Communication & Debugging
- D.** PushButton Interrupt
- E.** Timer Interrupt (Uses Non-Energia Code)

Lab Topics

Using Energia (Arduino)	9-14
<i>Lab 9</i>	9-15
Installing Energia.....	9-17
Installing the LaunchPad drivers	9-17
Installing Energia.....	9-17
Starting and Configuring Energia	9-18
Lab 9a – Blink	9-21
Your First Sketch.....	9-21
Modifying Blink	9-24
Lab 9b – Pushing Your button	9-25
Examine the code	9-25
Reverse button/LED action	9-26
Lab 9c – Serial Communication (and Debugging)	9-27
What if the Serial Monitor is blank? (‘G2553 Launchpad Configuration’)	9-28
Blink with Serial Communication.....	9-29
Another Pushbutton/Serial Example	9-29
Lab 9d – Using Interrupts.....	9-30
Adding an Interrupt.....	9-30
Lab 9e – Using TIMER_A	9-32
<i>Appendix – Looking ‘Under the Hood’</i>	9-33
Where, oh where, is Main	9-33
Two ways to change the MSP430 clock source	9-35
Sidebar – initClocks()	9-36
Sidebar Cont’d - Where is <u>F_CPU</u> defined?	9-37
<i>Lab Debrief</i>	9-38
Lab 9a	9-38
Lab 9b	9-39
Lab 9c.....	9-40
Lab 9d	9-42

Installing Energia

If you already installed Energia as part of the workshop prework, then you can skip this step and continue to [Lab 9a – Blink](#).

These installation instructions were adapted from the Energia Getting Started wiki page. See this site for notes on *Mac OSX* and *Linux* installations.

<https://github.com/energia/Energia/wiki/Getting-Started>

Note: If you are attending a workshop, the following files should have been downloaded as part of the workshop's pre-work. If you need them and do not have network access, please check with your instructor.

Installing the LaunchPad drivers

1. To use Energia you will need to have the LaunchPad drivers installed.

For Windows Users

If TI's Code Composer Studio 5.x with MSP430 support is already installed on your computer then the drivers are already installed. Skip to the next step.

- a) Download the LaunchPad drivers for Windows:
[LaunchPad CDC drivers zip file for Windows 32 and 64 bit](#)
- b) Unzip and double click DPinst.exe for Windows 32bit or DPinst64.exe for Windows 64 bit.
- c) Follow the installer instructions.

Installing Energia

2. Download Energia, if you haven't done so already.

The most recent release of Energia can be downloaded from the [download](#) page.

Windows Users

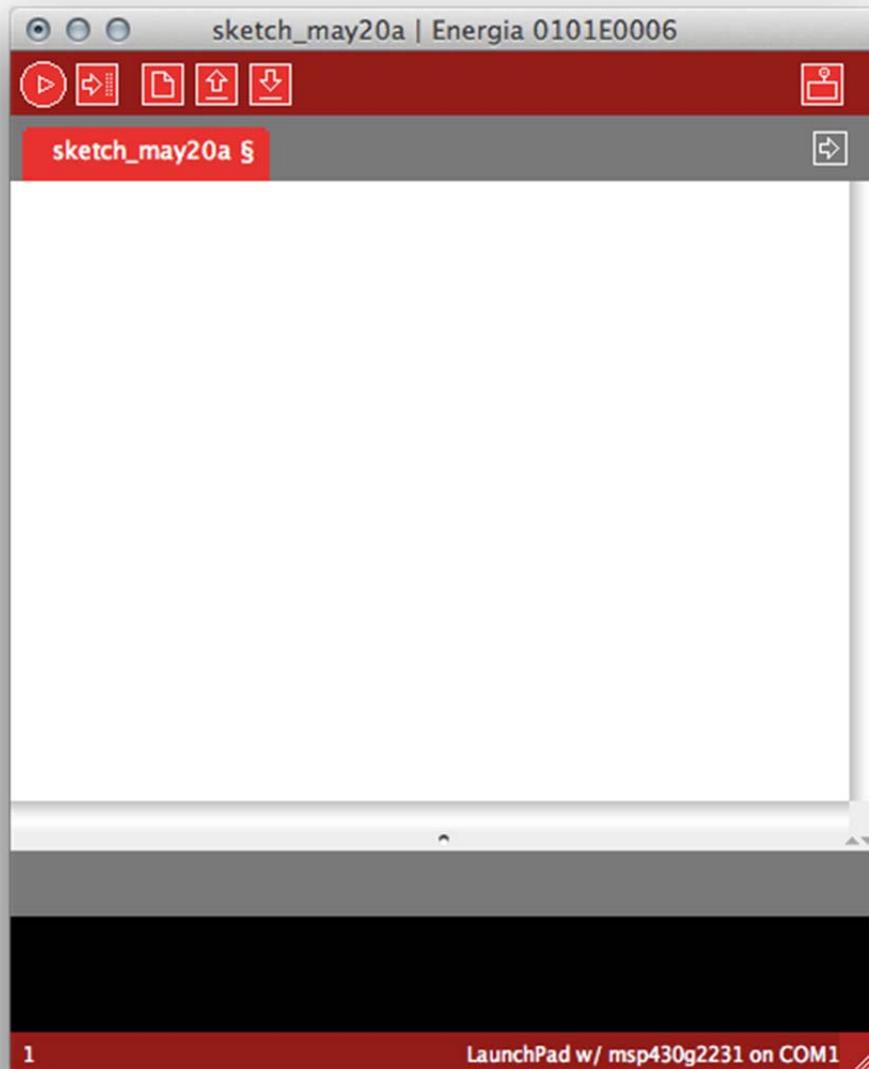
Double click and extract the energia-0101EXXX-windows.zip file to a desired location.

(We recommend unzipping it to: C:\TI\energia-0101E00xx).

Starting and Configuring Energia

3. Double click Energia.exe (Windows users).

Energia will start and an empty Sketch window will appear.



4. Set your *working folder* in Energia.

It makes it easier to save and open files if Energia defaults to the folder where you want to put your sketches.

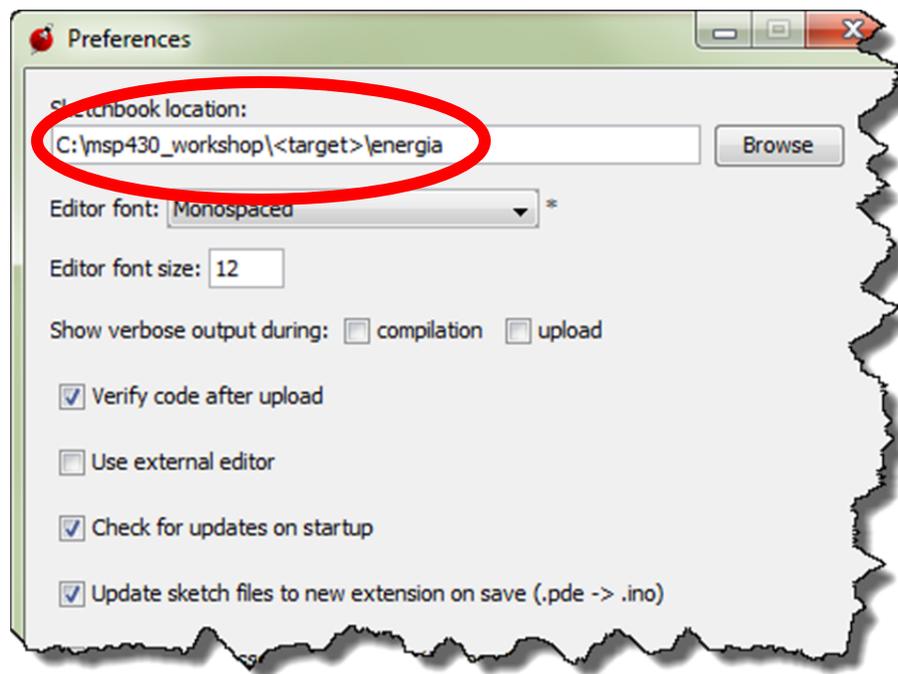
The easiest way to set this locations is via Energia's preferences dialog:

File → Preferences

Then set the *Sketchbook location* to:

C:\msp430_workshop*<target>*\energia

Which opens:



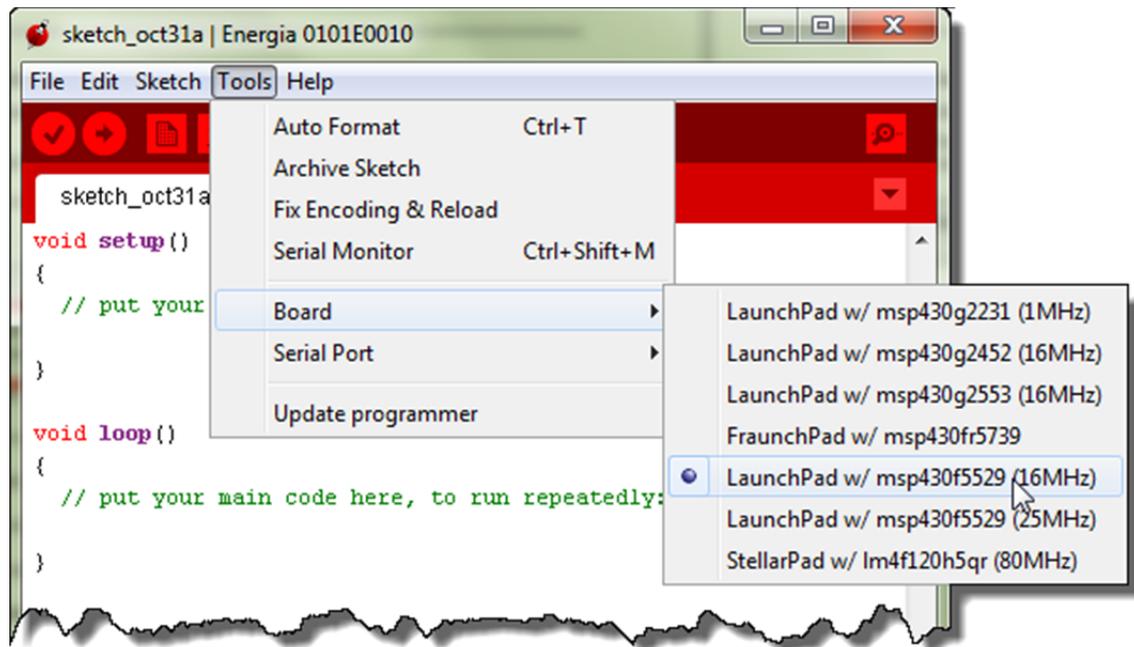
5. Selecting the Serial Port

Select **Serial Port** from the **Tools** menu to view the available serial ports.

For Windows, they will be listed as COMXXX port and usually a higher number is the LaunchPad com port. On Mac OS X they will be listed as /dev/cu.uart-XXXX.

6. Select the board you are using – most likely the msp430f5529 (16MHz).

To select the board or rather the msp430 in your LaunchPad, select **Board** from the **Tools** menu and choose the board that matched the msp430 in the LaunchPad.



Lab 9a – Blink

Don't blink, or this lab will go by without you seeing it. It's a very simple lab exercise – that happens to be one of the many examples included with the Energia package.

As simple as this example is, it's a great way to begin. In fact, if you have followed the flow of this workshop, you may recognize the *Blink* example essentially replicates the lab exercise we created in *Chapter 3* and *4* of this workshop.

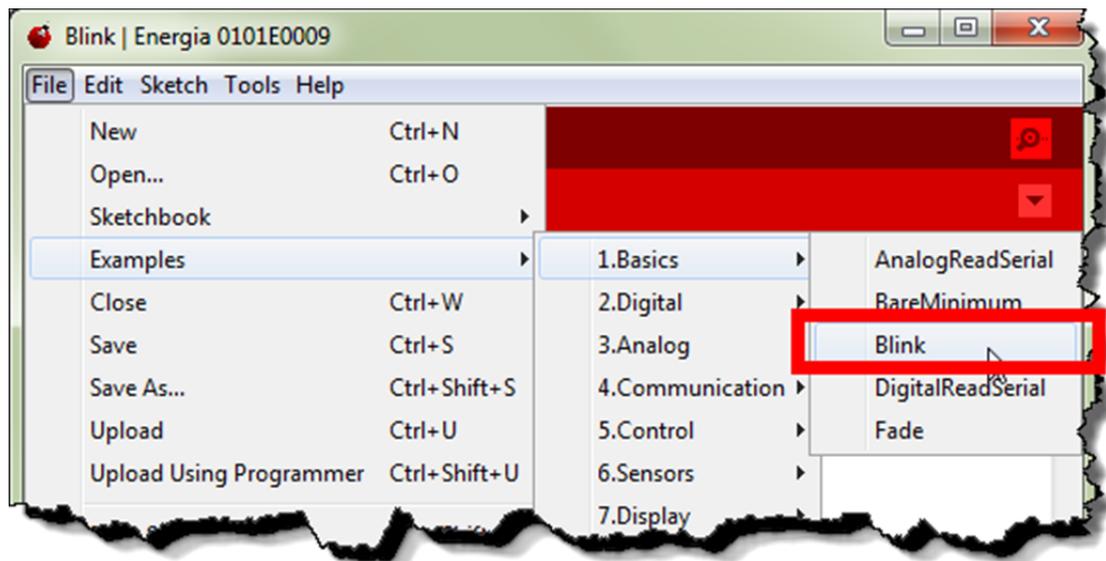
As we pointed out during the *Energia* chapter discussion, the *Wiring* language simplifies the code quite a bit.

Your First Sketch

1. Open the *Blink* sketch (i.e. program).

Load the *Blinky* example into the editor; select ***Blink*** from the *Examples* menu.

File → Examples → 1.Basics → Blink



2. Examine the code.

Looking at the Blink sketch, we see the code we quickly examined during our chapter discussion. This code looks very much like standard C code. (In Lab9d we examine some of the specific differences between this sketch and C code.)

At this point, due to their similarity to standard C language code, we will assume that you recognize most of the elements of this code. By that, we mean you should recognize and understand the following items:

- **#define** – to declare symbols
- **Functions** – what a function is, including: void, () and {}
- **Comments** – declared here using // characters

What we do want to comment on is the names of the two functions defined here:

- **setup()**: happens one time when program starts to run
- **loop()**: repeats over and over again

This is the basic structure of an Energia/Arduino sketch. Every sketch should have – at the very least – these two functions. Of course, if you don't need to setup anything, for example, you can leave it empty.

```
/*
  Blink
  Turns on an LED on for one second, then off for one second,
  repeatedly. This example code is in the public domain.
*/

void setup () {
  // initialize the digital pin as an output.
  // Pin 14 has an LED connected on most Arduino boards:
  pinMode (RED_LED, OUTPUT);
}

void loop () {
  digitalWrite (RED_LED, HIGH); // turn on LED
  delay (1000); // wait one second (1000ms)
  digitalWrite (RED_LED, LOW); // turn off LED
  delay (1000); // wait one second
}
```

3. Compile and upload your program to the board.

To compile and upload the Sketch to the LaunchPad click the  button.



Do you see the LED blinking? What color LED is blinking? _____

What pin is this LED connected to? _____

(Be aware, in the current release of Energia, this could be a trick question.)

Hint: We recommend you check out the Hardware Pin Mapping to answer this last question. There's a copy of it in the presentation. Of course, the original is on the Energia [wiki](#).

Modifying Blink

4. Copy sketch to new file before modification.

We recommend saving the original Blink sketch to a new file before modifying the code.

File → Save As...

Save it to:

C:\msp430_workshop\<<target>\energia\Blink_Green

Hint: This will actually save the file to:

C:\msp430_workshop\<<target>\energia\Blink_Green\Blink_Green.ino

Energia requires the sketch file (.ino) to their to be in a folder named for the project.

5. How can you change which color LED blinks?

Examine the H/W pin mapping for your board to determine what needs to change.

Please describe it here: _____

6. Make the other LED blink.

Change the code, to make the other LED blink.

When you've changed the code, click the **Upload** button to: compile the sketch; upload the program to the processor's Flash memory; and, run the program sketch.

Did it work? _____

(We hope so. Please ask for help if you cannot get it to work.)

Lab 9b – Pushing Your button

Next, let's figure out how to use the button on the Launchpad. It's not very difficult, but since there's already a sketch for that, we'll go ahead and use it.

1. Open the *Button* sketch (i.e. program).

Load the *Button* example into the editor.

File → Examples → 2.Digital → Button

2. Try out the sketch.

Before we even examine the code, let's try it out. (You're probably just like us ... going to try it out right away, too.)

When you push the button the (GREEN or RED) LED goes (ON or OFF)? _____

By the way, you probably know this already from earlier in the workshop, but which button are we using? If you're using the F5529 Launchpad, then the "user" buttons are called PUSH1 and PUSH2; the example uses PUSH2 (the board silkscreen says P1.1) as shown here:



Examine the code

3. The author of this sketch used the LED in a slightly different fashion.

How is the LED defined differently in the Button Sketch versus the Blink sketch?

4. Looking at the pushbutton...

How is the pushbutton created/used differently from the LED? _____

What "Energia" pin is the button connected to? _____

What is the difference between INPUT and INPUT_PULLUP? _____

5. A couple more items to notice...

Just like standard C code, we can create variables. What is the global variable used for in this example?

Finally, this is a very simple way to read and respond to a button. What would be a more efficient way to handle responding to a pushbutton? (And why would this be important to many of us MSP430 users?)

(Note, we will look at this 'more efficient' method in a later part of the lab.)

Reverse button/LED action

Do you find this example to be the reverse of what you expected? Would you prefer the LED to go ON when the button is pushed, rather than the reverse. Let's give that a try.

6. Save the example to sketch new file before modification.

Once again, we recommend saving the original sketch before modification. Save it to:

```
C:\msp430_workshop\<target>\energia\Button_reversed
```

7. Make the LED light only when the button is pressed.

Change the code as needed.

Hint: The changes required are similar to what you would do in C, they are not unique to *Energia/Arduino*.

8. When your changes are finished, upload it to your Launchpad.

Did it work? _____

Lab 9c – Serial Communication (and Debugging)

This lab uses the serial port (UART) to send data back and forth to the PC from the Launchpad.

In and of itself, this is a useful and common thing we do in embedded processing. It's the most common way to talk with other hardware. Beyond that, this is also the most common debugging method in Arduino programming. *Think of this as the “**printf**” for the embedded world of microcontrollers.*

1. Open the *DigitalReadSerial* example.

Once again, we find there's a (very) simple example to get us started.

File → Examples → 1.Basics → DigitalReadSerial

2. Save sketch as `myDigitalReadSerial`.

3. Examine the code.

This is a very simple program, but that's good since it's very easy to see what Energia/Arduino needs to get the serial port working.

```
/* DigitalReadSerial
   Reads a digital input on pin 2, prints the result to the
   serial monitor (This example code is in the public domain) */

void setup() {
  Serial.begin(9600);           // msp430g2231 must use 4800
  pinMode(PUSH2, INPUT_PULLUP);
}

void loop() {
  int sensorValue = digitalRead(PUSH2);
  Serial.println(sensorValue);
}
```

As you can see, serial communication is very simple. Only one function call is needed to setup the serial port: **Serial.begin()**. Then you can start writing to it, as we see here in the **loop()** function.

Note: Why are we limited to 9600 baud (roughly, 9600 bits per second)?

The G2553 Launchpad's onboard emulation (USB to serial bridge) is limited to 9600 baud. It is not a hardware limitation of the MSP430 device. Please refer to the wiki for more info: <https://github.com/energia/Energia/wiki/Serial-Communication>.

If you're using other Launchpads (such as the 'F5529 Launchpad), your serial port can transmit at much higher rates.

4. Download and run the sketch.

With the code downloaded and (automatically) running on the Launchpad, go ahead and push the button.

But, how do we *know* it is running? It doesn't change the LED, it only sends back the current pushbutton value over the serial port.

Hint: After running the sketch and looking at the Serial Monitor (in the next step), you might find that nothing is showing up. Try switching "pin 5" for "PUSH2" in the code. Look at the mapping diagrams between the 'G2553 and 'F5529 Launchpads to see the mismatch.

5. Open the serial monitor.

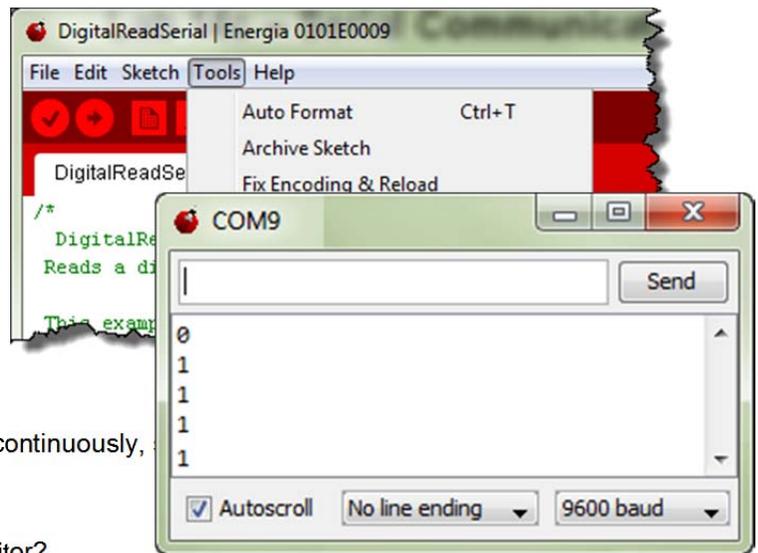
Energia includes a simple serial terminal program. It makes it easy to view (and send) serial streams via your computer.

With the Serial Monitor open, and the sketch running, you should see something like this:

You should see either a "1" or "0" depending upon whether the button is up or down.

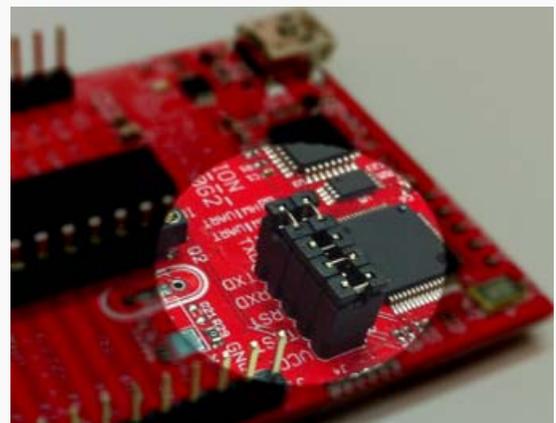
Also, notice that the value is updated continuously, writes it to port in the **loop()** function.

Do you see numbers in the serial monitor?



What if the Serial Monitor is blank? ('G2553 Launchpad Configuration)

If this is the case, your Launchpad is most likely configured incorrectly. For serial communications to work correctly, the J3 jumpers need to be configured differently than how the board is configured out-of-the-box. (This fooled us, too.) Refer to these diagrams for correct operation. (*This does not affect other Launchpads.*)



Blink with Serial Communication

Let's try combining a couple of our previous sketches: *Blink* and *DigitalReadSerial*.

6. Open the *Button* sketch.

Load the *Button* from the *Examples* menu.

File → Examples → 2.Digital → Button

7. Save it to a new file before modification.

Once again, we recommend saving the original sketch before modification. Save it to:

C:\msp430_workshop*<target>*\energia\Serial_Button

8. Add 'serial' code to your *Serial_Button* sketch.

Take the serial communications code from our previous example and add it to your new *Serial_Button* sketch. (Hint, it should only require two lines of code.)

9. Download and test the example.

Did you see the Serial Monitor and LED changing when you push the button?

10. Considerations for debugging...

How you can use both of these items for debugging?

Serial Port; LED (And, what if you didn't have an LED available on your board?):

Another Pushbutton/Serial Example

Before finishing Lab 9C, let's look at one more example.

11. Open the *StateChangeDetection* sketch.

Load the *sketch* from the *Examples* menu.

File → Examples → 2.Digital → StateChangeDetection

12. Examine the sketch, download and run it.

How is this sketch different? What makes it more efficient? _____

How is this (and all our sketches, up to this point) inefficient? _____

Lab 9d – Using Interrupts

Interrupts are a key part of embedded systems. It is responding to external events and peripherals that allow our programs to ‘talk’ to the real world.

Thusfar, we have actually worked with a couple different interrupts without having to know anything about them. Our serial communications involved interrupts, although the Wiring language insulates us from needing to know the details. Also, there is a timer involved in the `delay()` function; thankfully, it is also managed automatically for us.

In this part of the lab exercise, you will setup two different interrupts. The first one will be triggered by the pushbutton; the second, by one of the MSP430 timers.

1. Once again, let’s start with the *Blink* code.

```
File → Examples → 1.Basics → Blink
```

2. Save the sketch to a new file.

```
File → Save As...
```

Save it to:

```
C:\msp430_workshop\<target>\energia\Interrupt_PushButton
```

3. Before we modify the file, run the sketch to make sure it works properly.

4. To `setup()`, configure the `GREEN_LED` and then initialize it to `LOW`.

This requires two lines of code which we have used many times already.

Adding an Interrupt

Adding an interrupt to our Energia sketch requires 3 things:

- An **interrupt source** – what will trigger our interrupt. (We will use the pushbutton.)
- An ISR (**interrupt service routine**) – what to do when the interrupt is triggered.
- The **interruptAttach()** function – this function hooks a trigger to an ISR. In our case, we will tell Energia to run our ISR when the button is pushed.

5. Interrupt Step 1 - Configure the PushButton for input.

Look back to an earlier lab if you don’t remember how to do this.

6. Interrupt Step 2 – Create an ISR.

Add the following function to your sketch; it will be your interrupt service routine. This is about as simple as we could make it.

```
void myISR()  
{  
    digitalWrite(GREEN_LED, HIGH);  
}
```

In our function, all we are going to do is light the `GREEN_LED`. If you push the button and the Green LED turns on, you will know that successfully reached the ISR.

7. Interrupts Step 3 – Connect the pushbutton to our ISR.

You just need to add one more line of code to your *setup()* routine, the *attachInterrupt()* function. But what arguments are needed for this function? Let's look at the Arduino reference to figure it out.

[Help](#) → [Reference](#)

Look up the *attachInterrupt()* function. What three parameters are required?

1. _____
2. _____
3. _____

Once you have figured out the parameters, **add the function** to your *setup()* function.

8. Compile & download your code and test it out.

Does the green RED_LED flash continuously? _____

When you push the button, does the GREEN_LED light? _____

When you push reset, the code should start over again. This should turn off the GREEN_LED, which you can then turn on again by pushing PUSH2.

Note: Did the GREEN_LED fail to light up? If so, that means you are not getting an interrupt.

First, check to make sure you have all three items – button is configured; *attachInterrupt()* function called from *setup()*; ISR routine that lights the GREEN_LED

The most common error involves setting up the push button incorrectly. The button needs to be configured with INPUT_PULLUP. In this way, the button is held high which lets the system detect when the value falls as the button is pressed.

Missing the INPUT_PULLUP is especially common since most Arduino examples – like the one shown on the *attachInterrupt()* reference page only show INPUT. This is because many boards include an external pullup resistor, Since the MSP430 contains an internal pullup, you can save money by using it instead.

Lab 9e – Using TIMER_A

9. Create a new sketch and call it Interrupt_TimerA

File → New

File → Save As...

C:\msp430_workshop*<target>*\energia\Interrupt_TimerA

10. Add the following code to your new sketch.

```
#include <inttypes.h>

uint8_t timerCount = 0;

void setup()
{
    pinMode(RED_LED, OUTPUT);

    TA0CCTL0 = CCIE;
    TA0CTL = TASSEL_2 + MC_2;
}

void loop()
{
    // Nothing to do.
}

__attribute__((interrupt(TIMER0_A0_VECTOR)))
void myTimer_A(void)
{
    timerCount = (timerCount + 1) % 80;
    if(timerCount == 0)
        P1OUT ^= 1;
}
```

In this case, we are not using the `attachInterrupt()` function to setup the interrupt. If you double-check the Energia reference, it states the function is used for ‘external’ interrupts. In this case, the MSP430’s Timer_A is an internal interrupt.

In essence, though, the same three steps are required:

- a) The interrupt source must be setup. In our example, this means setting up TimerA0’s CCTL0 (capture/compare control) and TA0CTL (TimerA0 control) registers.
- b) An ISR function – which, in this case, is named “myTimer_A”.
- c) A means to hook the interrupt source (trigger from TimerA0) to our function. In this case, we need to plug the Interrupt Vector Table ourselves. The GCC compiler uses the `__attribute__((interrupt(TIMER_A0_VECTOR)))` line to plug the Timer_A0 vector.

Note: You might remember that we introduced *Interrupts* in *Chapter 5* and *Timers* in *Chapter 6*. In those labs, the syntax for the interrupt vector was slightly different from what we are using here. This is because the other chapters use the TI compiler. Energia uses the open-source GCC compiler, which uses a slightly different syntax.

Appendix – Looking ‘Under the Hood’

We are going to create three different lab sketches in Lab 9d. All of them will essentially be our first ‘Blink’ sketch, but this time we’re going to vary the system clock – which will affect the rate of blinking. We will help you with the required C code to change the clocks, but if you want to study this further, please refer to *Chapter 3 – Initialization and GPIO*.

Where, oh where, is Main

How does Energia setup the system clock?

Before jumping into how to change the MSP430 system clock rate, let’s explore how Energia sets up the clock in the first place. Thinking about this, our first question might be...

What is the first function in every C program? (This is not meant to be a trick question)

If Energia/Arduino is built around the C language, where is the *main()* function? Once we answer this question, then we will see how the system clock is initialized.

Open main.cpp ...

`C:\TI\energia-0101E0010\hardware\msp430\cores\msp430\main.cpp`

The “`C:\TI\energia-0101E0010`” may be different if you unzipped the Energia to a different location.

When you click the *Download* button, the tools combine your *setup()* and *loop()* functions into the `main.cpp` file included with Energia for your specific hardware. Main should look like this:

main.cpp

C:\TI\energia-0101E0010\hardware\msp430\cores\msp430\



Clicking download combines sketch with main.cpp to create a valid c++ program

```

// main.cpp
#include < Energia.h >
int main(void)
{
  init();
  setup();
  for (;;) {
    loop();
    if (serialEventRun) {
      serialEventRun();
    }
  }
  return 0;
}

```

Energia.h contains the #defines, enums, prototypes, etc.

System initialization is done in **wiring.c** (see next slide)

We have already seen **setup()** and **loop()**. This is how Energia uses them.

Where do you think the MSP430 clocks are initialized? _____

Follow the trail. Open `wiring.c` to find how `init()` is implemented.

`C:\TI\energia-0101E0010\hardware\msp430\cores\msp430\wiring.c`

The `init()` function implements the essential code required to get the MSP430 up and running. If you have already completed *Chapter 4 – Clocking and Initialization*, then you should recognize most of these activities. At reset, you need to perform two essential activities:

- Initialize the clocks (choose which clock source you want use)
- Turn off the Watchdog timer (unless you want to use it, as a watchdog)

The Energia `init()` function takes this three steps further. They also:

- Setup the Watchdog timer as a standard (i.e. interval) timer
- Setup two GPIO pins
- Enable interrupts globally

init() in wiring.c

`C:\TI\energia-0101E0010\hardware\msp430\cores\msp430\`

```

// wiring.c
void init()
{
    disableWatchDog();
    initClocks();
    enableWatchDogIntervalMode();
    // Default to GPIO (P2.6, P2.7)
    P2SEL &= ~(BIT6|BIT7);
    __eint();
}
enableWatchDogIntervalMode()
initClocks()
disableWatchDog()
enableWatchDog()
delayMicroseconds()
delay()
watchdog_isr ()

```

- ◆ `wiring.c` provides the core files for device specific architectures
- ◆ `init()` is where the default initializations are handled
- ◆ As discussed in [Ch 3](#) (Init & GPIO)
 - ◆ Watchdog timer (WDT+) is disabled
 - ◆ Clocks are initialized (DCO 16MHz)
 - ◆ WDT+ set as interval timer

Two ways to change the MSP430 clock source

There are two ways you can change your MSP430 clock source:

- Modify the *initClocks()* function defined in `wiring.c`
- Add the necessary code to your *Setup()* function to modify the clock sources

Advantages

- Do not need to re-modify `wiring.c` after updating to new revision of Energia
- Changes are explicitly shown in your own sketch
- Each sketch sets its own clocking, if it needs to be changed
- In our lab, it allows us to demonstrate that you can modify hardware registers – i.e. processor specific hardware – from within your sketch

Disadvantages

- Code portability – any time you add processor specific code, this is something that will need to be modified whenever you want to port your Arduino/Energia code to another target platform
- A little less efficient in that clocking gets set twice
- You have to change each sketch (if you always want a different clock source/rate)

Sidebar – initClocks()

Here is a snippet of the `initClocks()` function found in `wiring.c` (for the ‘G2553 Launchpad’). We call it a snippet, since we cut out the other CPU speeds that are also available (8 & 12 MHz).

The beginning of this function starts out by setting the calibration constants (that are provided in Flash memory) to their associated clock configuration registers.

(Sidebar): `initClocks()` in `wiring.c`

```
void initClocks(void)
{
  #if (F_CPU >= 16000000L)
    BCCTL1 = CALBC1_16MHZ;
    DCOCTL = CALDCO_16MHZ;
  #elif (F_CPU >= 1000000L)
    BCCTL1 = CALBC1_1MHZ;
    DCOCTL = CALDCO_1MHZ;
  #endif

  BCCTL2 &= ~(DIVS_0);
  BCCTL3 |= LFXT1S_2;

  CSCTL2 &= ~SELM_7;
  CSCTL2 |= SELM_DCOCLK;
  CSCTL3 &= ~(DIVM_3|DIVS_3);

  #if F_CPU >= 16000000L
    CSCTL1 = DCORSEL;
  #elif F_CPU >= 1000000L
    CSCTL1 = DCOFSEL0|DCOFSEL1;
    CSCTL3 |= DIVM_3;
  #endif
}
```

- ◆ F_CPU defined in `boards.txt`
- ◆ Select ‘board’ via: Tools→Boards

Select correct calibration constants based on chosen clock frequency

- ◆ Set SMCLK to F_CPU
- Set ACLK to VLO (12KHz)
- ◆ Clear main clock (MCLK)
- Use DCO for MCLK
- Clear divide clock bits

Set MCLK as per F_CPU

If you work your way through the second and third parts of the code, you can see the BCS (Basic Clock System) control registers being set to configure the clock sources and speeds. Once again, there are more details on this in *Clocking* chapter and its lab exercise.

Sidebar Cont’d - Where is F_CPU defined?

We searched high & low and couldn’t find it. Finally, after reviewing a number of threads in the Energia forum, we found that it is specified in `boards.txt`. This is the file used by the debugger to specify which board (i.e. target) you want to work with. You can see the list from the Tools→Board menu.

```
C:\TI\energia-0101E0010\hardware\msp430\boards.txt
```

```
#####
lpmsp430g2231.name=LaunchPad w/ msp430g2231 (1MHz)
lpmsp430g2231.upload.protocol=rf2500
lpmsp430g2231.upload.maximum_size=2048
lpmsp430g2231.build.mcu=msp430g2231
lpmsp430g2231.build.f_cpu=1000000L
lpmsp430g2231.build.core=msp430
lpmsp430g2231.build.variant=launchpad

#####
#lpmsp430g2231f.name=LaunchPad w/ msp430g2231 (16MHz)
#lpmsp430g2231f.upload.protocol=rf2500
#lpmsp430g2231f.upload.maximum_size=16384
#lpmsp430g2231f.build.mcu=msp430g2231
#lpmsp430g2231f.build.f_cpu=16000000L
#lpmsp430g2231f.build.core=msp430
#lpmsp430g2231f.build.variant=launchpad

#####
lpmsp430g2553.name=LaunchPad w/ msp430g2553 (16MHz)
lpmsp430g2553.upload.protocol=rf2500
lpmsp430g2553.upload.maximum_size=16384
lpmsp430g2553.build.mcu=msp430g2553
lpmsp430g2553.build.f_cpu=16000000L
lpmsp430g2553.build.core=msp430
lpmsp430g2553.build.variant=launchpad

#####
lpmsp430fr5739.name=FraunchPad w/ msp430fr5739
lpmsp430fr5739.upload.protocol=rf2500
lpmsp430fr5739.upload.maximum_size=15872
lpmsp430fr5739.build.mcu=msp430fr5739
lpmsp430fr5739.build.f_cpu=16000000L
lpmsp430fr5739.build.core=msp430
lpmsp430fr5739.build.variant=fraunchpad

#####
```

Lab Debrief

Lab 9a

Q&A: Lab9A (1)

Lab A

3. Do you see the LED blinking? What color LED is blinking? Red
 What pin is this LED connected to? P1_0
 (Code says Pin14, it was RED that blinked)
 (Be aware, in the current release of Energia, this could be a trick question.)



```
void setup() {
  // initialize the digital pin as an output.
  // Pin 14 has an LED connected on most Arduino boards:
  pinMode(RED_LED, OUTPUT);
}
```

Q&A: Lab9A (2)

5. How can you change which color LED blinks?
 Examine the H/W pin mapping for your board to determine what needs to change.
 Please describe it here: Change from P1_0 to P4_7, for the green LED to blink
 (Easier yet, just use the pre-defined symbol: GREEN_LED)
6. Make the other LED blink.
 Did it work? Yes

Lab 9b

Q&A: Lab9B (1)

2. Try out the sketch.

When you push the button the (GREEN or RED) LED goes (ON or OFF)?

Green LED goes OFF

Examine the code

3. How is the LED defined differently in the 'Button' Sketch versus the 'Blink' sketch?

In 'Blink', the LED was #defined (as part of Energia);

in 'Button', it was defined as a const integer. Both work equally well.

4. How is the pushbutton created/used differently from the LED?

In Setup() it is configured as an 'input'; in loop() we use digitalRead()

What "Energia" pin is the button connected to? P1_1

What is the difference between INPUT and INPUT_PULLUP?

INPUT config's the pin as a simple input – e.g. allowing you to read pushbutton.

Using INPUT_PULLUP config's the pin as an input with a series pullup resitor;

(many TI μ C provide these resistors as part of their hardware design).

Q&A: Lab9B (2)

5. Just like standard C code, we can create variables. What is the global variable used for in the 'Button' example?

'buttonState' global variable holds the value of the button returned by digitalRead().

We needed to store the button's value to perform the IF-THEN/ELSE command.

What would be a more efficient way to handle responding to a pushbutton? (And why would this be important to many of us MSP430 users?)

It would be more efficient to let the button 'interrupt' the processor, as opposed to

reading the button over and over again. This is as the processor cannot SLEEP

while polling the pushbutton pin. If using an interrupt, the processor could sleep until

being woken up by a pushbutton interrupt.

(Note, we will look at this later.)

Reverse Button/LED action

8. Did it work? Yes (it should)

```

if (buttonState == HIGH) {
  // turn LED on:
  digitalWrite(ledPin, HIGH);
}
else {
  // turn LED off:
  digitalWrite(ledPin, LOW);
}

```

LOW (with a red arrow pointing to the HIGH value in the code)

HIGH (with a red arrow pointing to the LOW value in the code)

Lab 9c

Q&A: Lab9C (1)

5. Did you see numbers in the serial monitor? Yes

If using 'G2553 LP you might not have seen anything in the Serial Monitor. If so, change:
Change the serial-port jumpers

Note – changing jumpers is only needed for 'G2553 Value-Line Launchpad



Q&A: Lab9C (2)

Blink with Serial Communication (Serial_Button sketch)

9. Did you see the Serial Monitor and LED changing when you push the button?

You (we hope so)

```

void setup() {
  Serial.begin(9600);

  // initialize the LED pin as an output
  pinMode(ledPin, OUTPUT);
}

void loop(){
  // read the state of the pushbutton:
  buttonState = digitalRead(buttonPin);
  Serial.println(buttonState);
}

```

10. Considerations for debugging... How you can use both of these items for debugging? (Serial Port and LED)

Use the serial port to send back info, just as you might use printf() in your C code.

An LED works well to indicate you reached a specific place in code. For example, later on we'll use this to indicate our program has jumped to an ISR (interrupt routine)

Similarly, many folks hook up an oscilloscope or logic analyzer to a pin, similar to using an LED. (Since our boards have more pins than LEDs.)

Q&A: Lab9C (3)

Another Pushbutton/Serial Example (StateChangeDetection sketch)

12. Examine the sketch, download and run it.

How is this sketch different? What makes it more efficient?

It only sends data over the UART whenever the button changes

How is this (and all our sketches, up to this point) inefficient?

Our pushbutton sketches – thusfar – have used polling to determine the state of the button. It would be more efficient to let the processor sleep; then be woken up by an interrupt generated when the pushbutton is depressed.

Lab 9d

Q&A: Lab9D

Interrupt Example (Interrupt_PushButton)

7. Look up the `attachInterrupt()` function. What three parameters are required?

1. Interrupt source – in our case, it's PUSH2
2. ISR function to be called when int is triggered – for our ex, it's "myISR"
3. Mode – what state change to detect; the most common is "FALLING"

8. Compile & download your code and test it out.

Does the green RED_LED flash continuously? _____

When you push the button, does the GREEN_LED light? _____

Notes:

- ◆ Use reset button to start program again and clear GREEN_LED
- ◆ Most common error, not configuring PUSH2 with INPUT_PULLUP.