

MSP430 Workshop

STUDENT GUIDE



*MSP430 Workshop
Revision 3.01
November 2013*

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

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.



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
RAM	1-18
TLV	1-18
FRAM	1-19
<i>MSP430 Peripherals</i>	1-22
GPIO	1-22
Timers	1-23
Clocking and Power Management	1-24
Analog	1-25
Communications (Serial ports, USB, Radio)	1-26
Hardware Accelerators	1-27
Summary	1-28
<i>ULP</i>	1-29
Profile Your Activities	1-30
<i>Launchpad's</i>	1-34
<i>Lab 1 – Out-of-Box User Experience Lab</i>	1-35

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. USB
8. Using Energia (Arduino)

MSP430 Workshop (v3.0)

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.

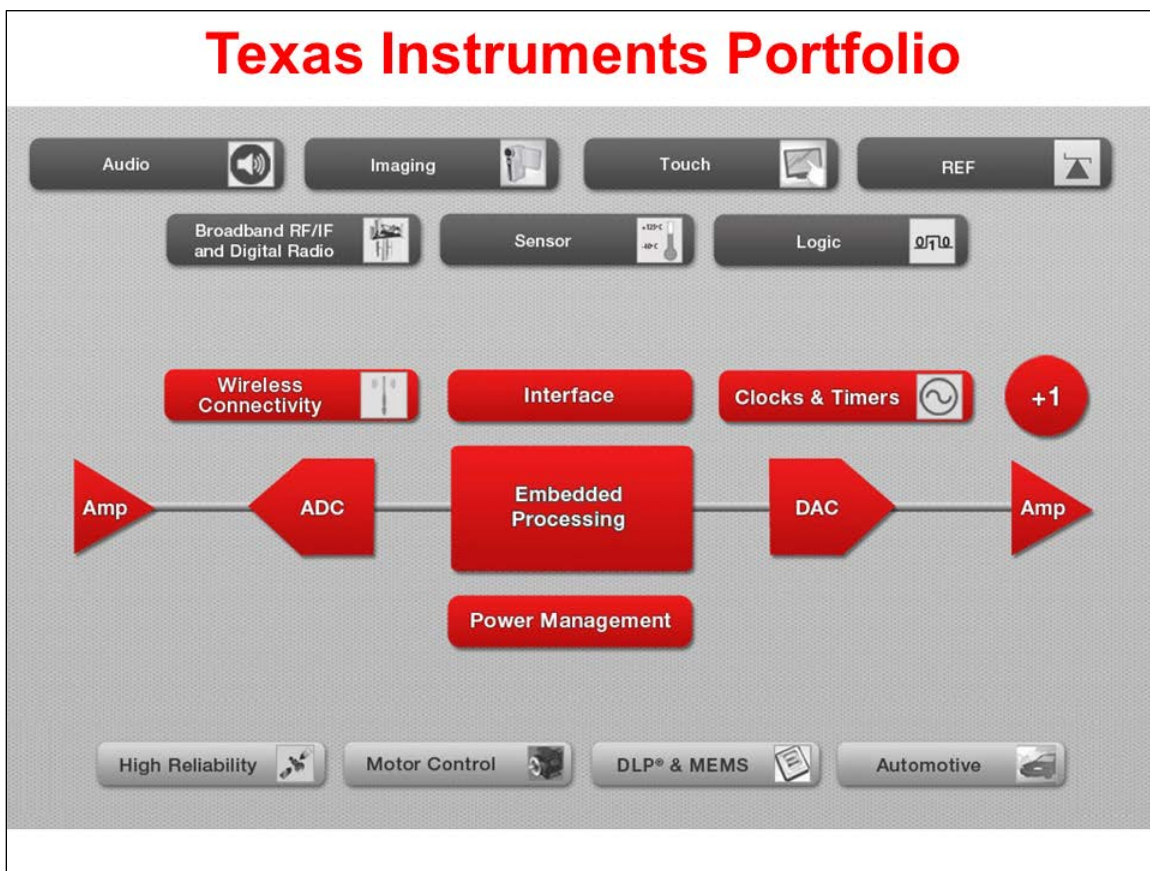
- Chapter 5:** *Interrupts* ... do you use interrupts? Yep, they're one of the most fundamental parts of embedded system designs. This is especially true when your processor is known as the king of low-power. We examine the sources, how to enable, and what to do in response to ... interrupts.
- Chapter 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 manufactures 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.

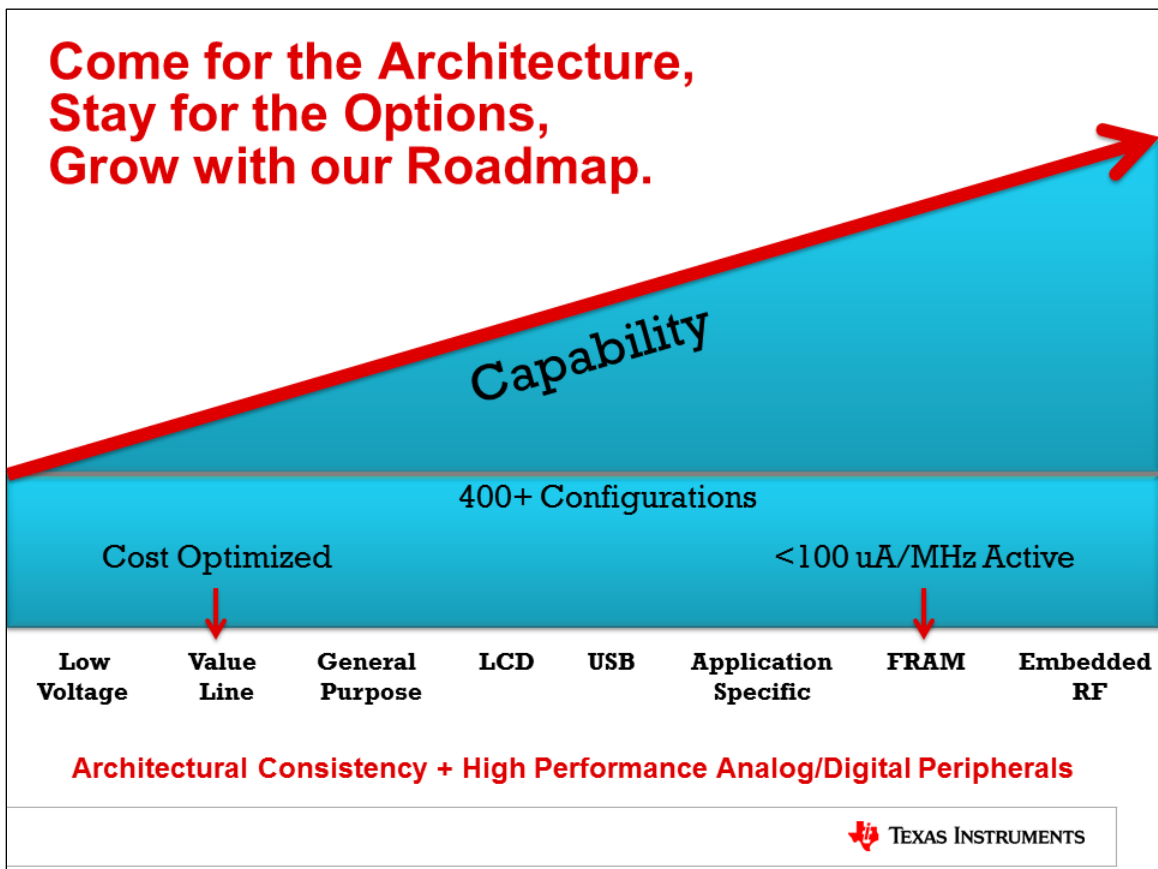
Driving to the *lowest-power dissipation* is one of those areas. In the end, the venerable **MSP430** is not to be outdone on the low end. As the MSP430 teams says, Ultra Low-Power (ULP) is "in our DNA". You know you're doing something right when the 10-year shelf-life of the battery ends up self-dissipating 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'. That said, 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.

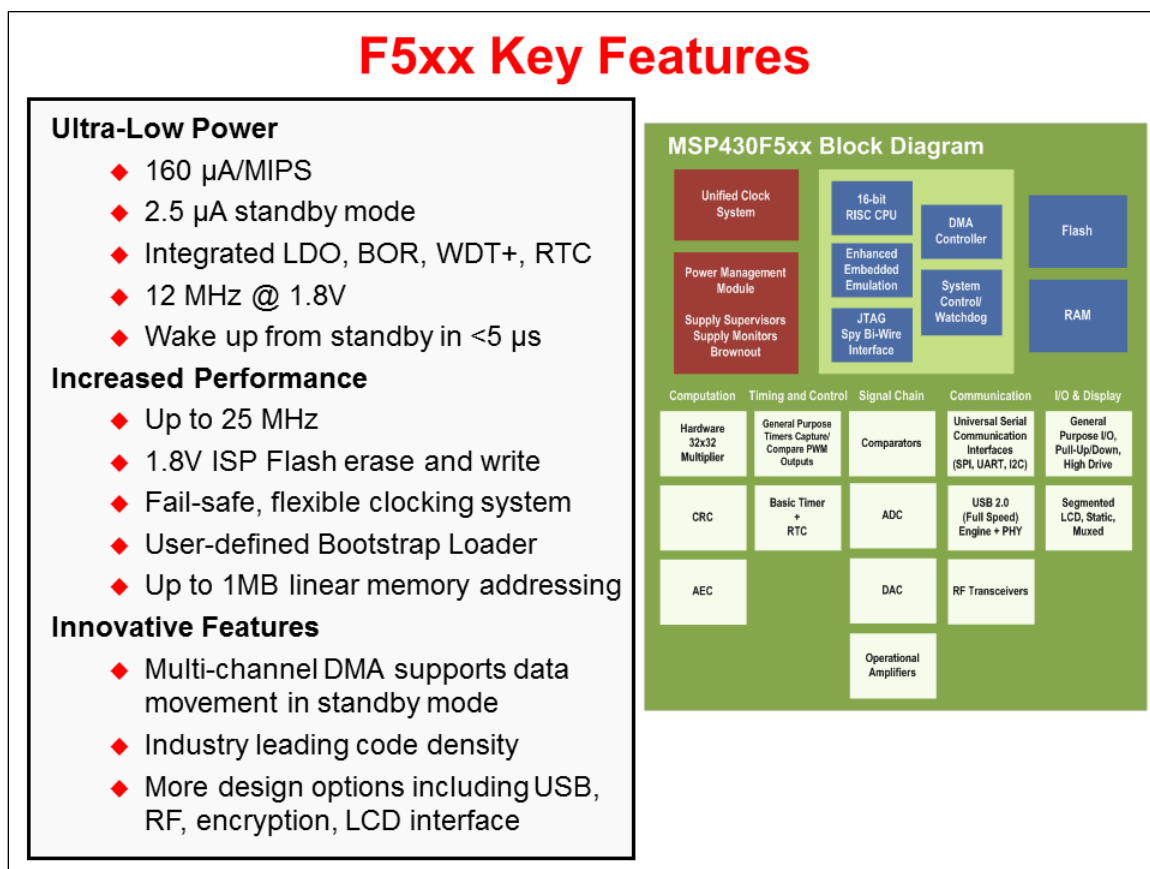
MSP430 Families

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	•	•	•	•	•	•	•	•	•
PC	•	•	•	•	•	•	•	•	•
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

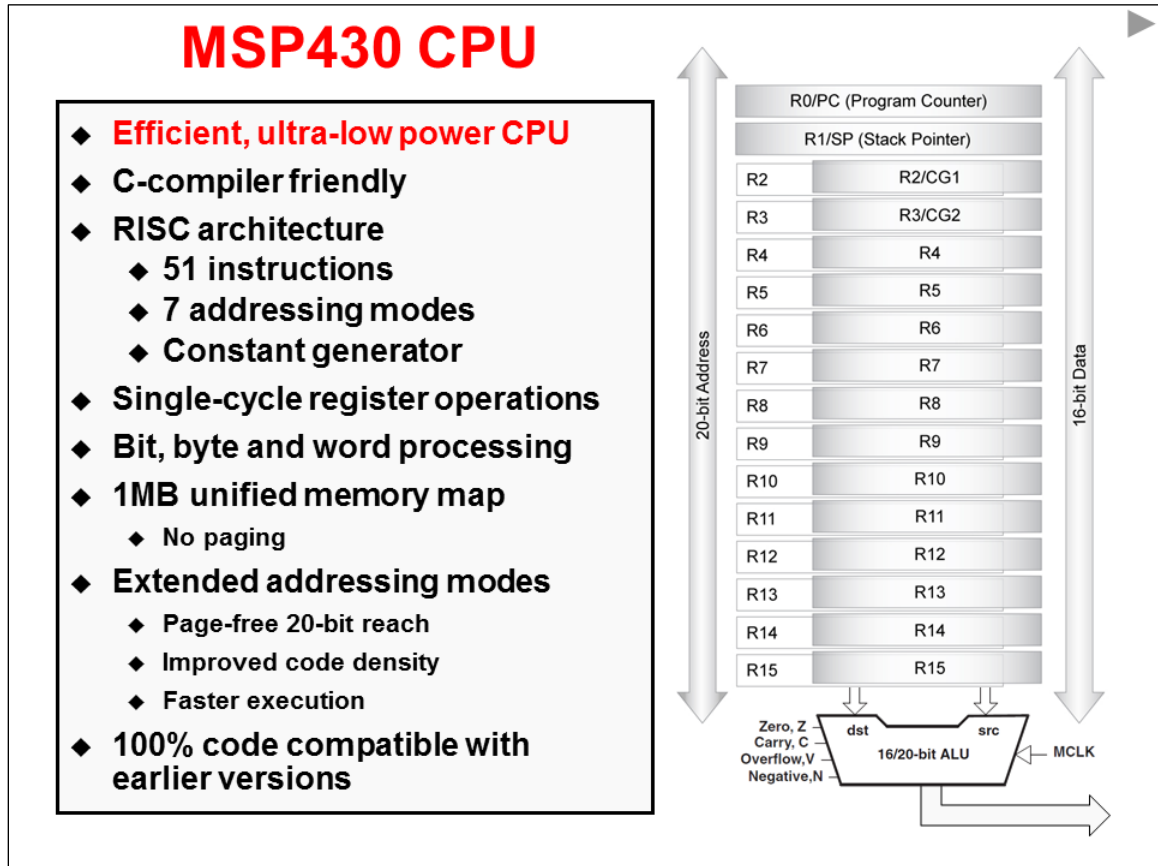
Here's a quick overview of the device we'll be using in this workshop. The MSP430F5529 is part of the F5xx series of devices and is found on the new 'F5529 USB Launchpad.



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		Code/Cycles
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

Atomic

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

<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>4314</u>	mov.w #0002h, R4	; With CG
<u>40341234</u>	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

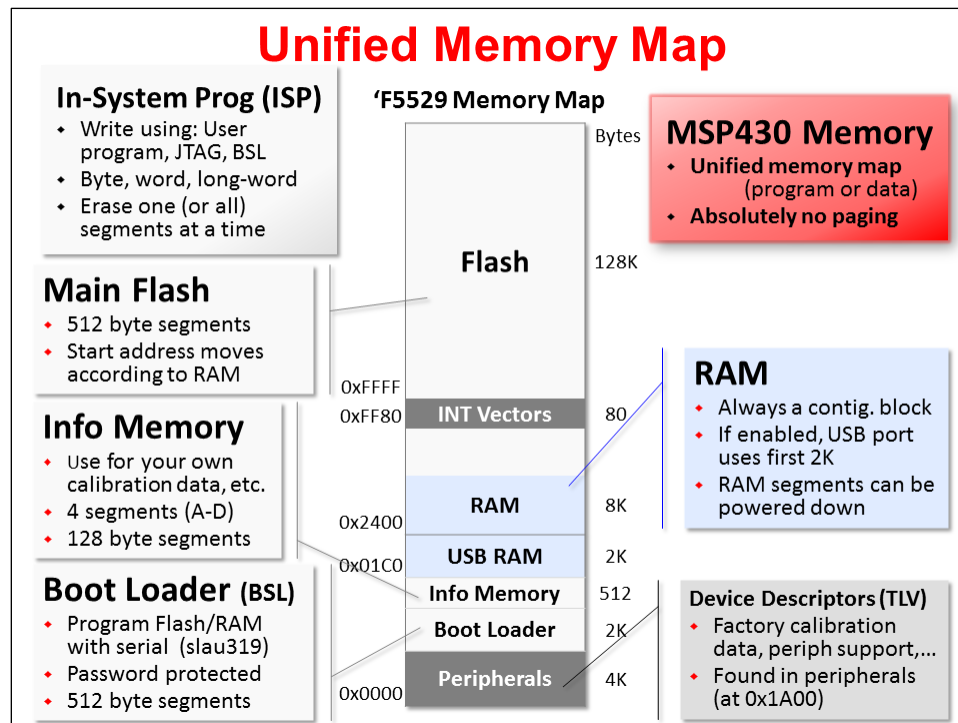
MSP430 Memory

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.)

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**

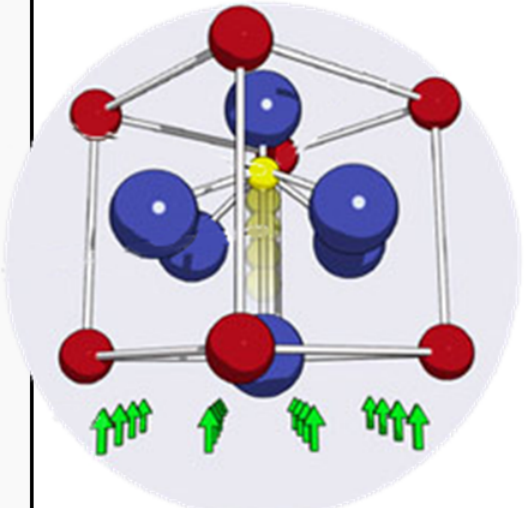


Photo: Ramtron Corporation

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.

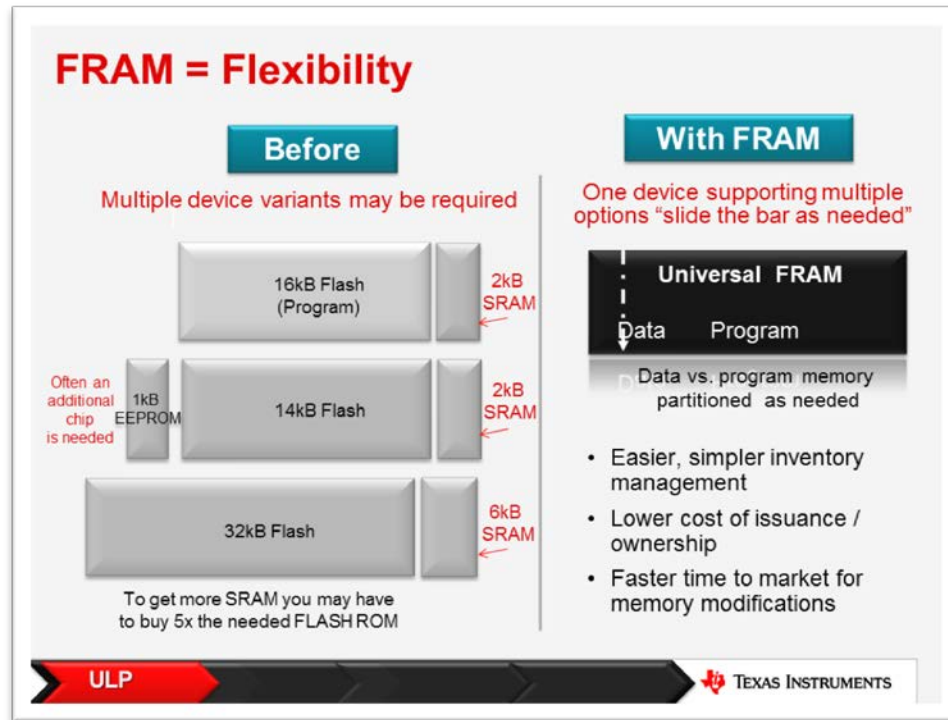
The downside, of course, is that your program could be just as easily overwritten in the same fashion. (We shouldn't have code that writes to program addresses – but accidents occur.) To this end, the FRAM based devices provide a memory protection unit (MPU) that lets you create 1 to 3 segments of FRAM.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



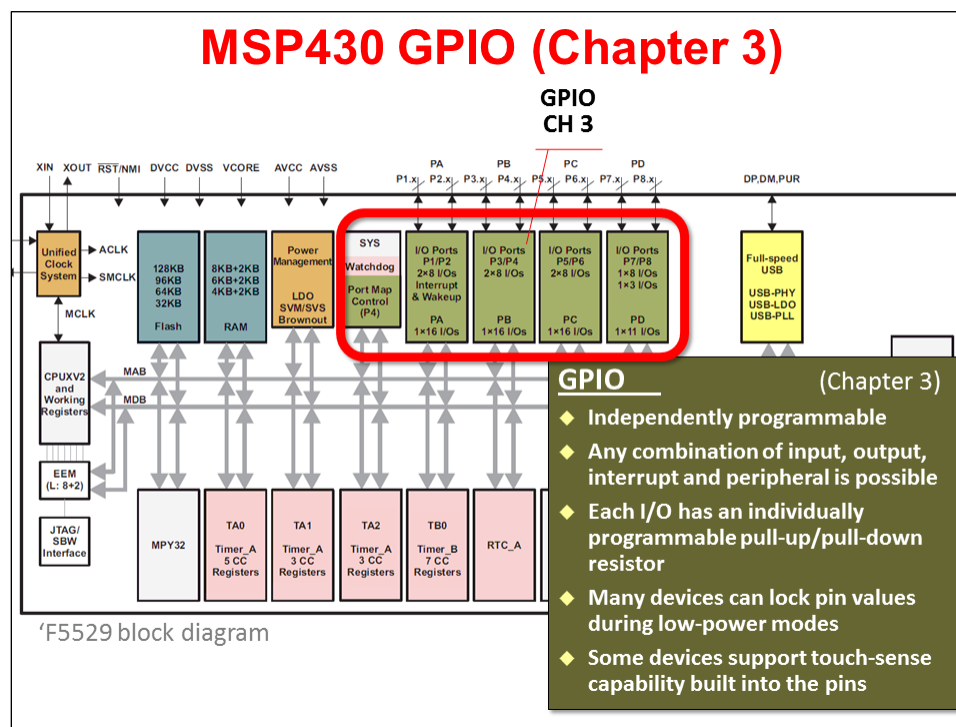
MSP430 Peripherals

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.

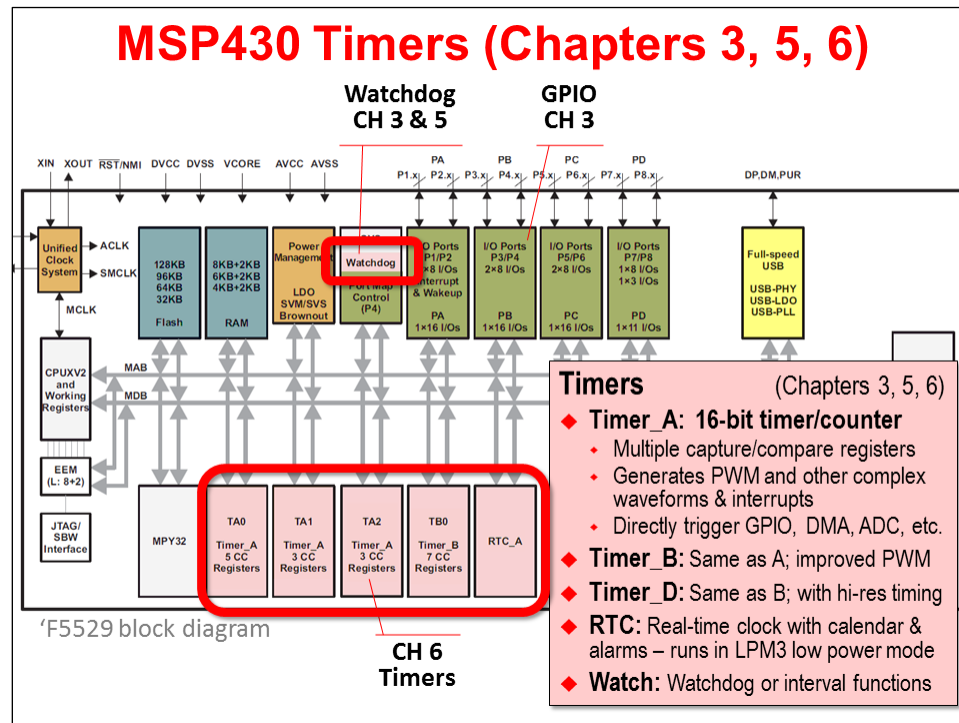
Like most current day microcontrollers, the pins on our devices are heavily multiplexed. That is, you often have one of several choices of signals that can be output to a given pin. The MSP430 makes each signal independently programmable, which affords maximum flexibility.



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

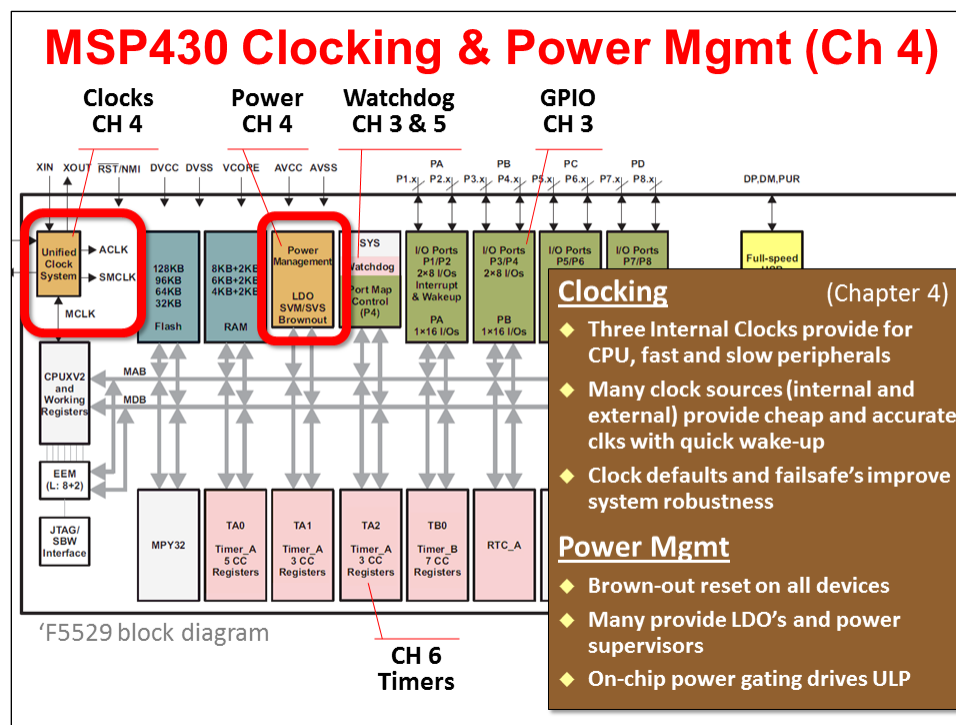
MSP430 Clocks (Chapter 4)

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.

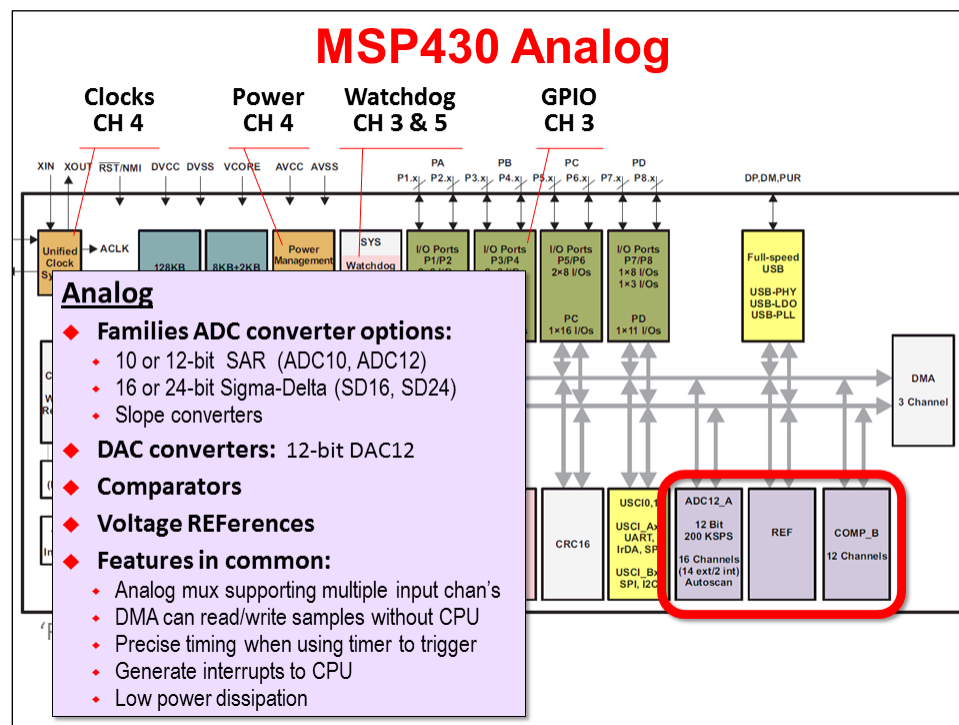
Analog

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.)

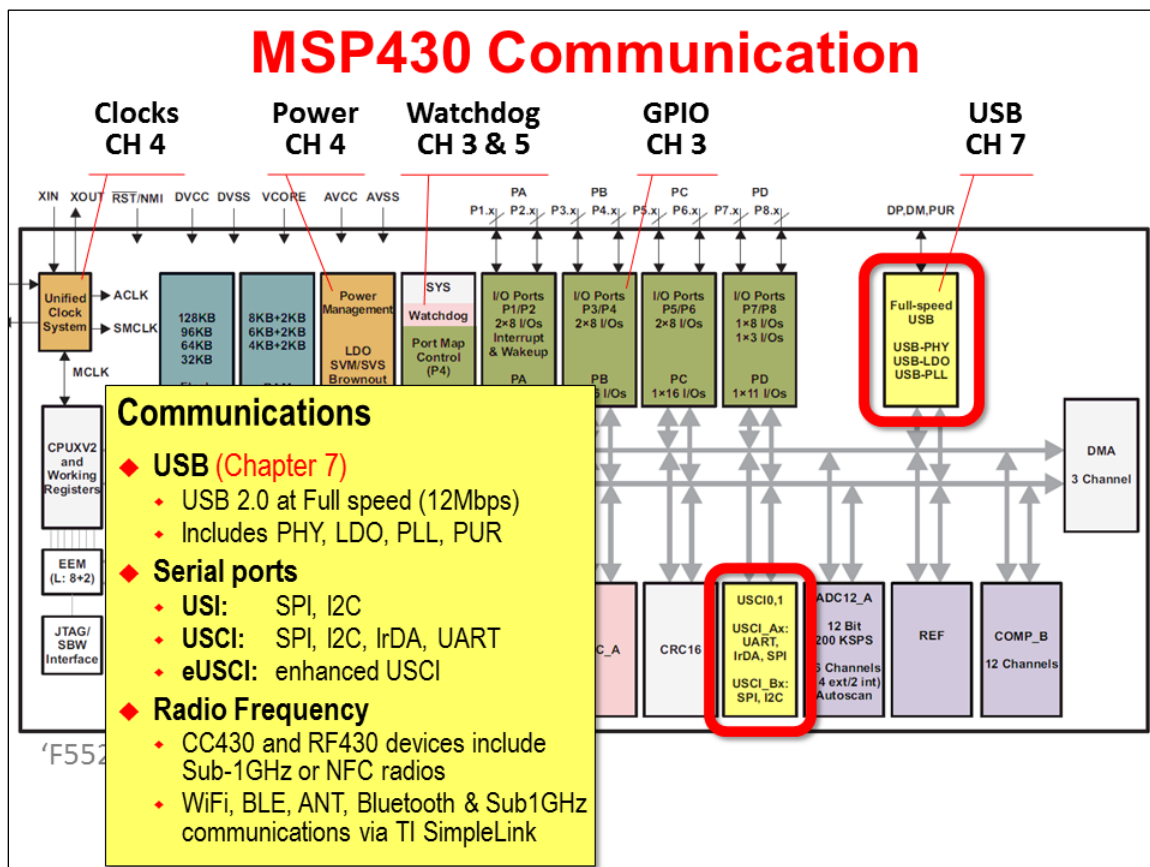
In many systems, though, you will want an actual ADC (analog to digital) converter. The MSP430 family provides a wide variety of options. In fact, some designers select their specific MSP430 device based upon which type of converter they want to use.

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.



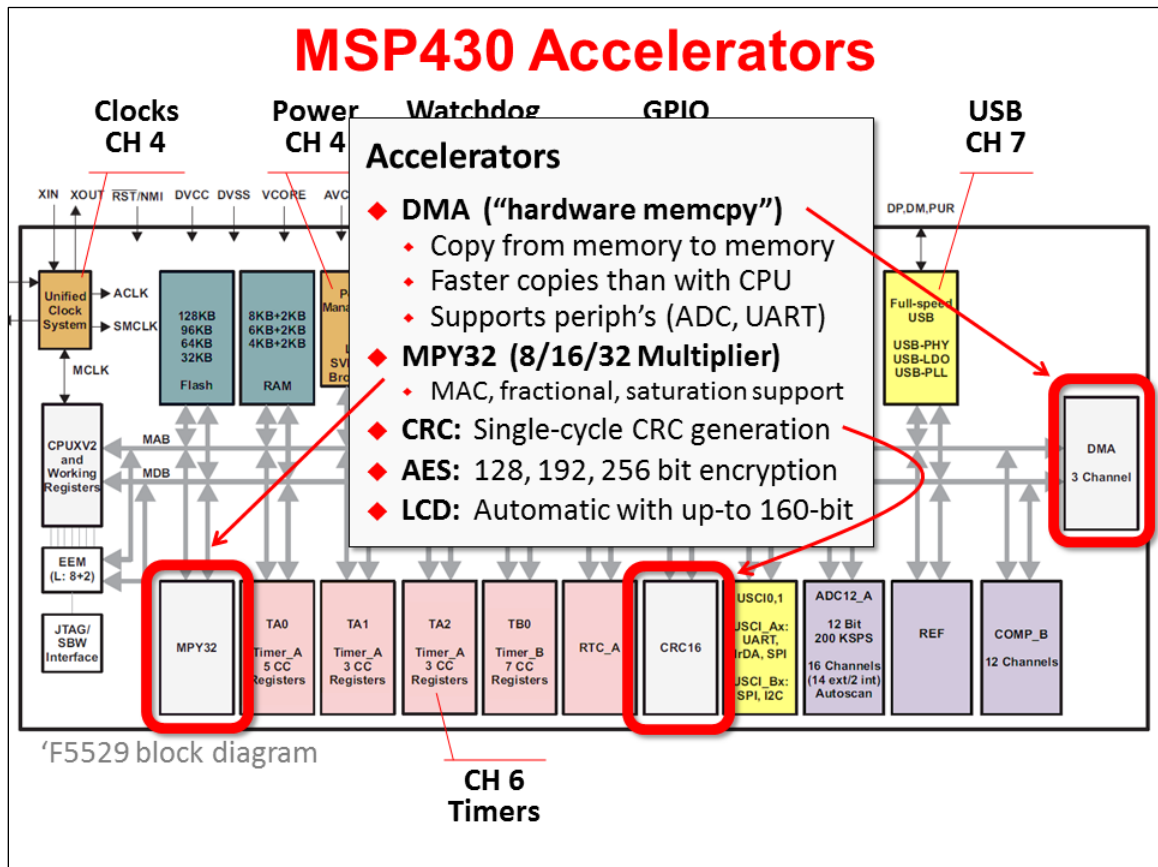
The additional of radios to some MSP430 devices makes them quite unique in the industry. Beyond that, TI has created wireless chips and modules that can be used from any MSP430 device. It's really telling when the cheapest Value Line MSP430 device can actually talk Wi-Fi using TI's CC3000 module. A similar 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.

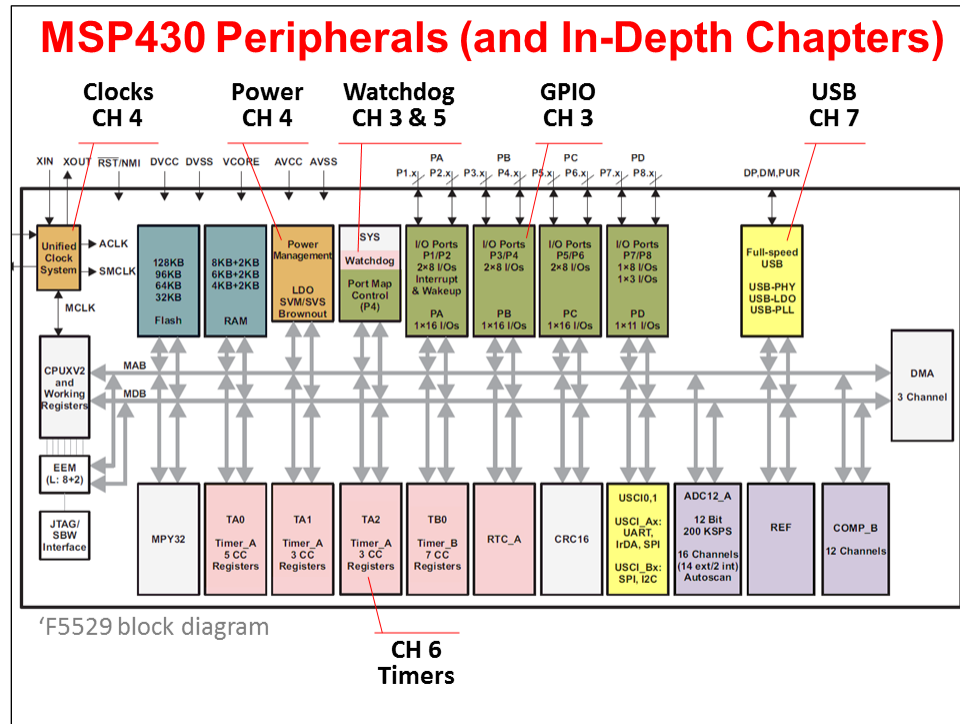


Another example is the multiplier. We can benefit from it without any programming effort, since the compiler automatically uses this hardware, when it's available.

With regards to the Direct Memory Access (DMA) peripheral, we caution you ... if you find yourself using memcopy() 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.

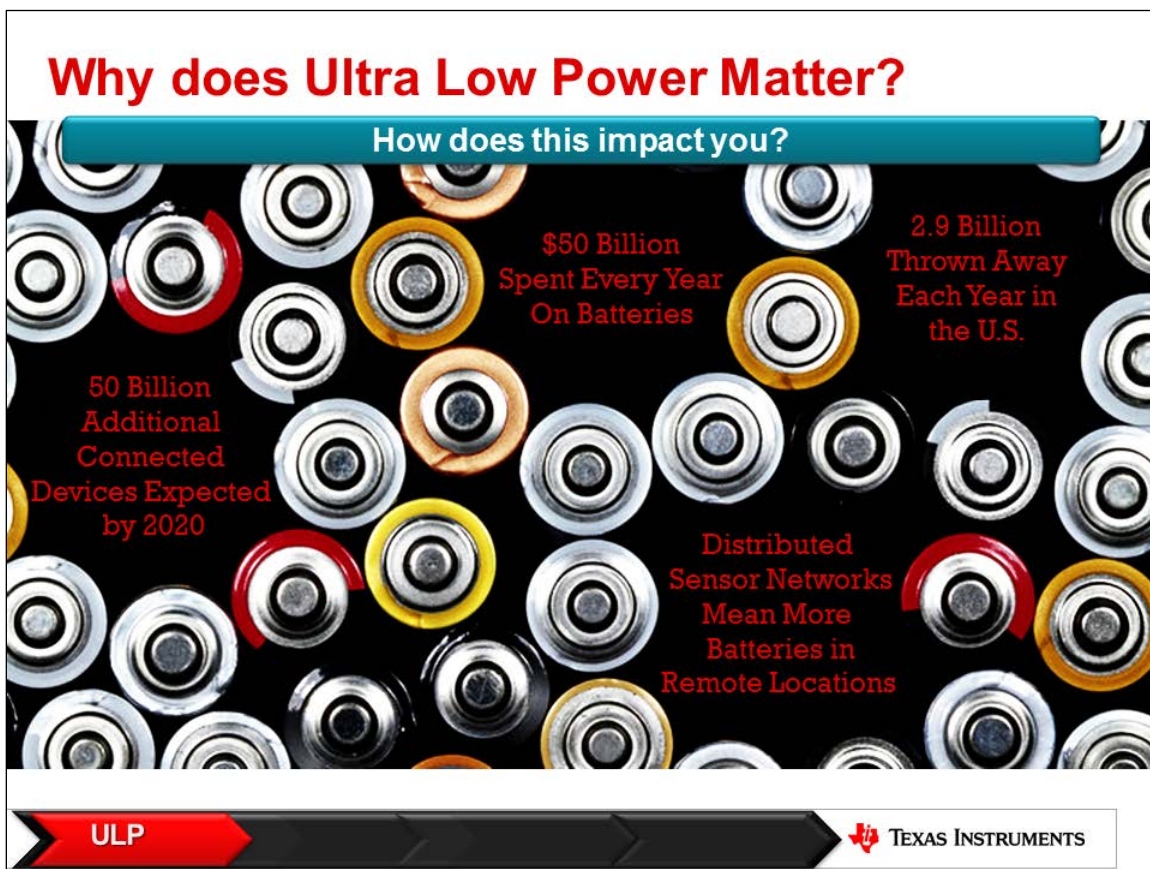


ULP

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.

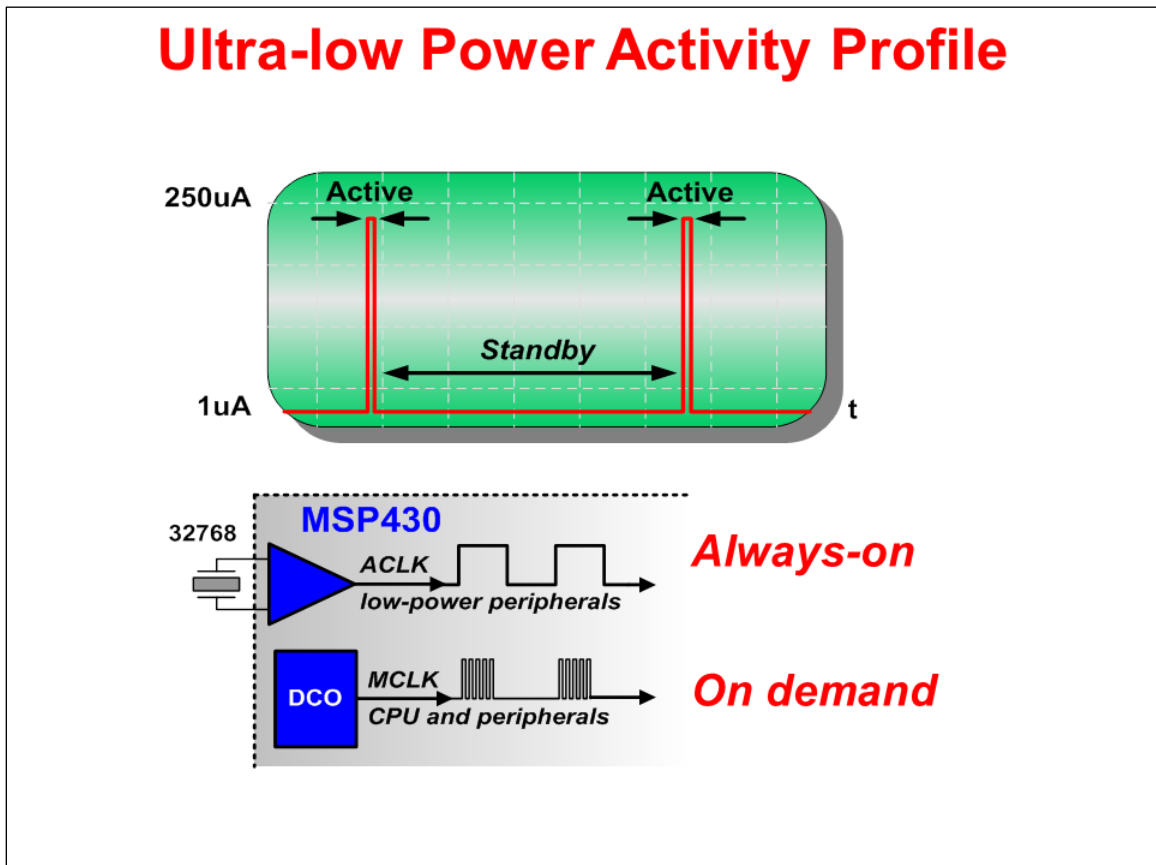
But even when the application does not demand low power, we think it still matters. The trend in electronics over the past few years has been, "Why consume power if you don't have to?" In fact, the MSP430 has found many new applications in the last couple of years where end-users are demanding the reduction 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.



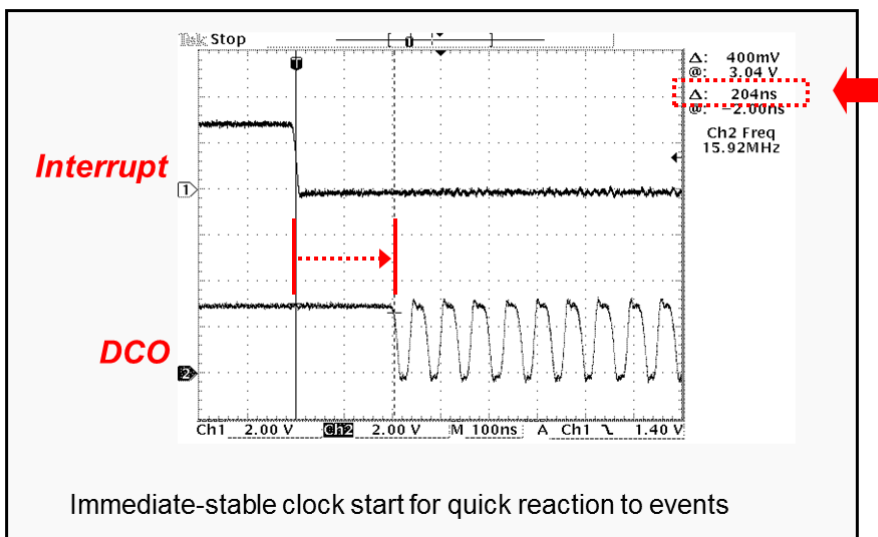
The MSP430 supports this sleep/wake/sleep profile quite well, by providing a variety of low-power modes (LPM). The following chart is an example of the LPM's found on various MSP430 devices, showing which resources are powered down by LP mode. It also broadly indicates what it takes to wake up from a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

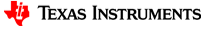
Performance on Demand



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

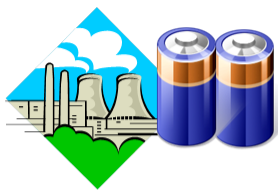
Device (mode)		F2xx	F5xx	FR57xx	FR58xx FR59xx
Performance (max)		16MHz	25MHz	24MHz (FRAM at 8MHz)	16MHz (FRAM at 8MHz)
Flex Unified Memory		No	No	FRAM (16K)	FRAM (64K)
Active Mode	AM	>250 μ A/MHz	~180 μ A/MHz	100 μ A/MHz	100 μ A/MHz
RTC Mode	LPM3x	0.8uA	2.5uA	.5uA	0.36uA
Standby Mode	LPM4	0.1uA	1.3uA	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**



ULP Advisor | 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 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 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



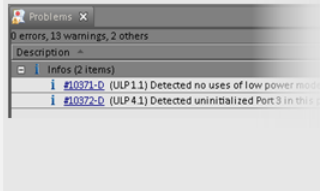
Checks against a thorough Ultra-Low-Power checklist.


- 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 & di
- ULP 5.2 Avoid processing-intensive floating poin
- 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 wh

Highlights areas of improvement within code.

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

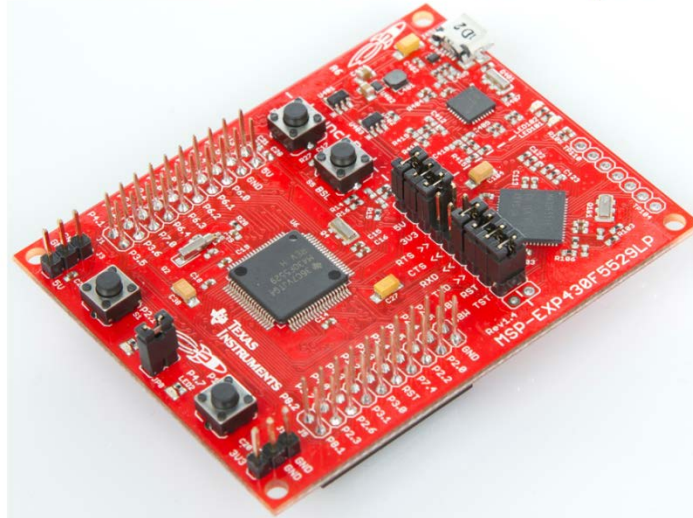




Launchpad's

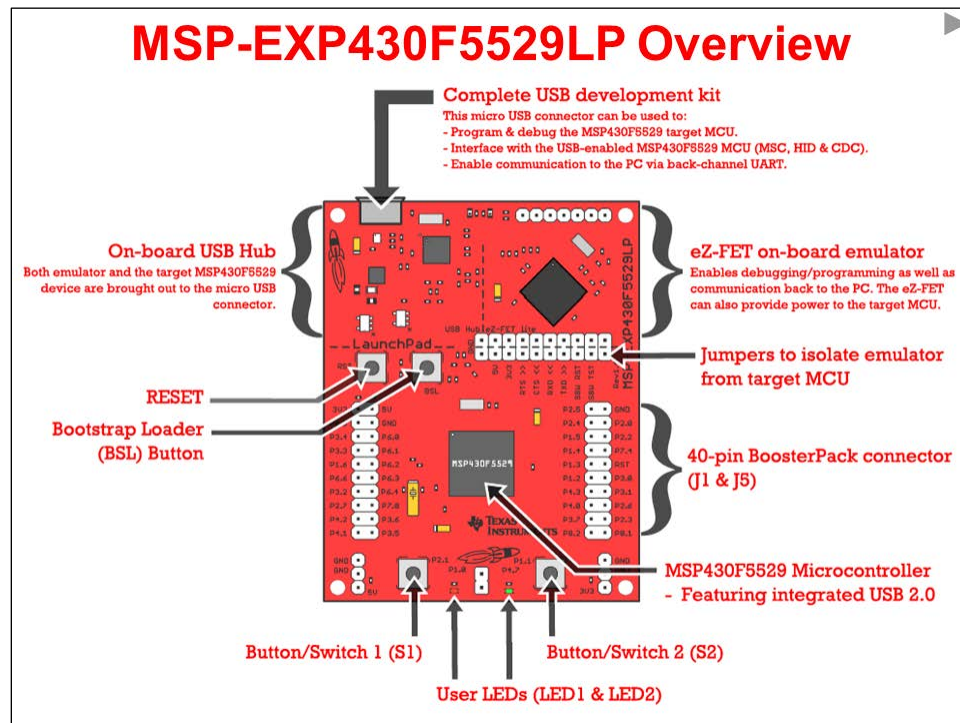
The MSP430F5529 Launchpad is a powerful, low-cost evaluation (and development) tool.

MSP-EXP430F5529LP Launchpad



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).

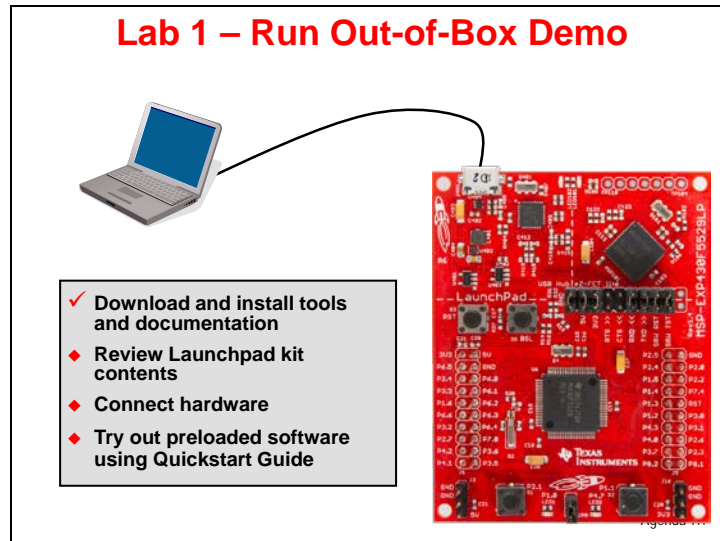
The lower portion of the board provides the target of our application programming. There are LED's, pushbuttons, and pins we can use to let our programs interact with the 'real world'.



Lab 1 – MSP4305529 LaunchPad User Experience

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:

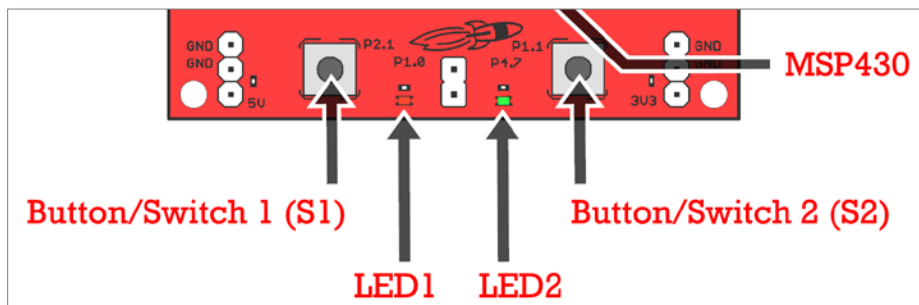
- **Button1.txt** – the contents of this file are "typed out" to the PC, using the emulated keyboard when you press button S1
- **Button2.txt** – the contents of this file are "typed out" to the PC, using the emulated keyboard when you press button S2
- **MSP430 USB LaunchPad.url** – when you double-click, 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 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 powering the current version of the 'F5529 Launchpad via the USB port, the board must enumerate, otherwise it will not power on. This means USB batteries – which do not contain a USB host – cannot be used as a power source, at this time.

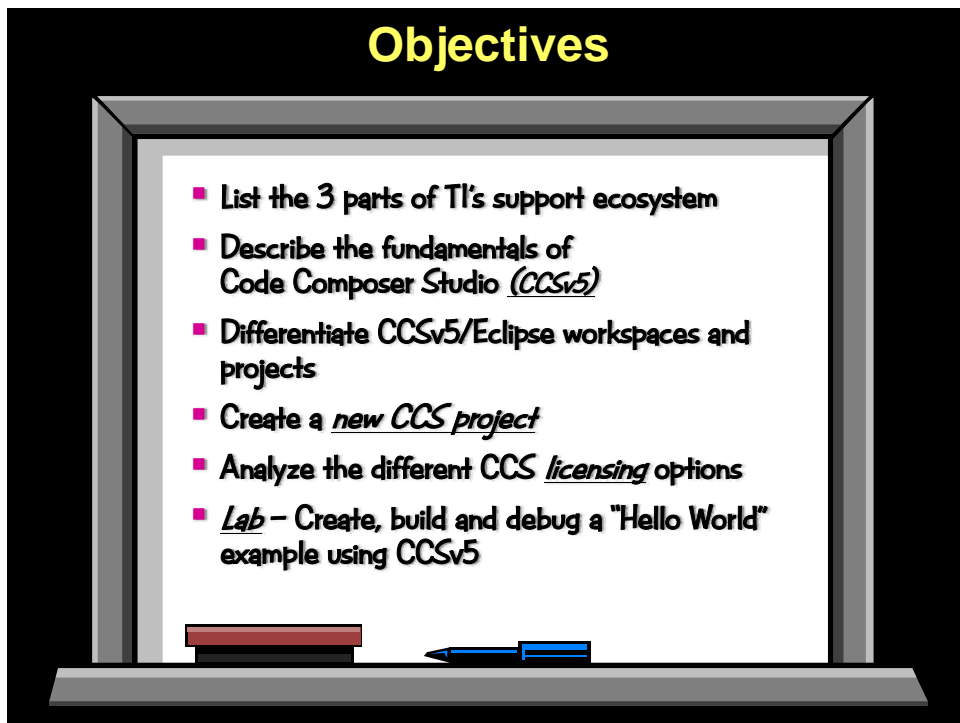
Programming C with CCS

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, this lab is a good review and you will likely learn some new things you don't know.

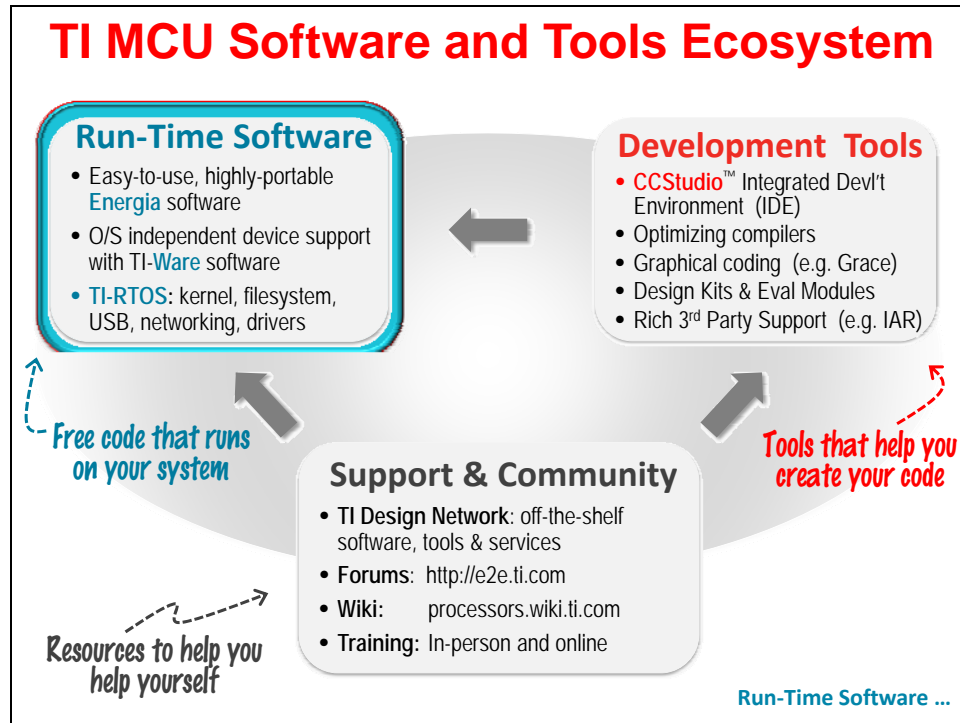
Learning Objectives



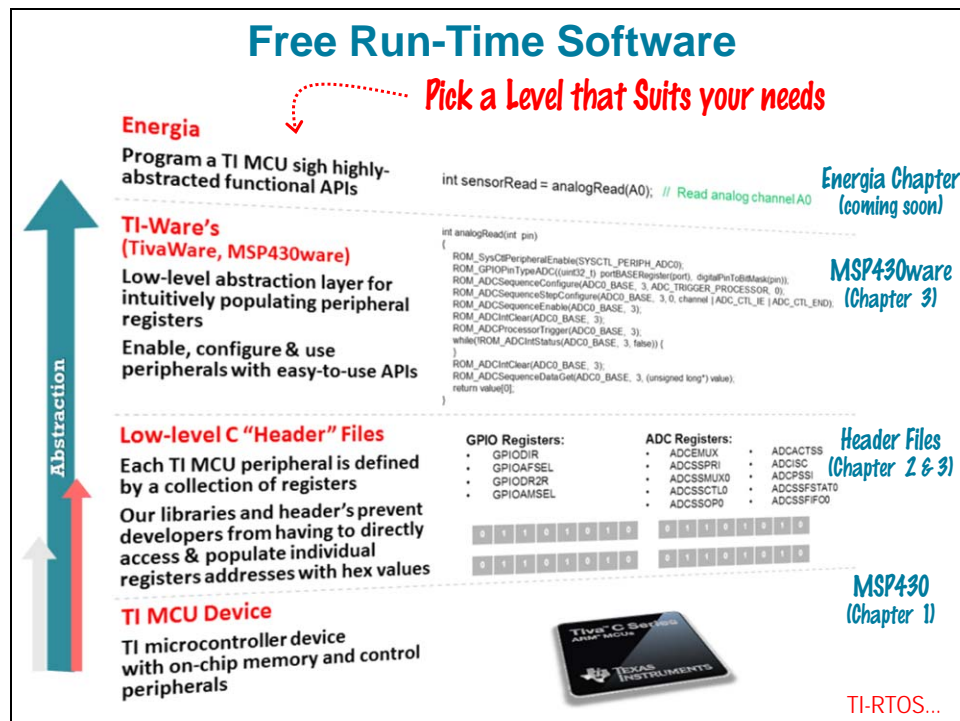
Chapter Topics

Programming C with CCS	2-1
<i>TI Support Ecosystem</i>	2-3
Run-Time Software	2-3
Development Tools	2-4
Support & Community	2-6
<i>Examining CCSv5</i>	2-8
Functional Overview.....	2-8
Perspectives.....	2-9
Target Config & Emulation	2-10
Workspaces & Projects	2-11
Creating a Project	2-12
Licensing/Pricing	2-13
<i>Writing MSP430 C Code</i>	2-14
Build Config & Options	2-14
Data Types	2-15
Device Specific Files (.h and .cmd).....	2-15
MSP430 Compiler Intrinsic Functions	2-16
<i>Lab 2 – CCSv5 Projects</i>	2-17

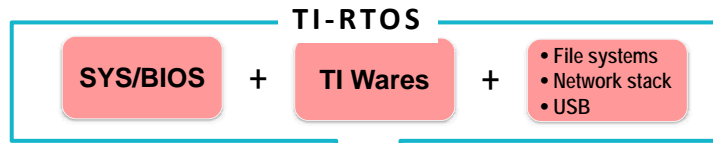
TI Support Ecosystem



Run-Time Software



Real-Time Operating System (TI-RTOS)



TI-RTOS:

- Provides an optimized real-time kernel that works with TI Wares (driverLib) and other additional software collateral
- TI-RTOS Availability *(Note: SYS/BIOS kernel already available for all these)*
 - Now: Tiva-C ARM Cortex M4F, Concerto (F28M35) devices
 - Soon: MSP430
 - Planned: Sitara Cortex-A8 and -A9 processors
- **Training:** 2-day TI-RTOS Kernel Workshop

Real-time kernel (SYS/BIOS)	TI Wares	Additional Collateral
<ul style="list-style-type: none"> • Scheduling • Memory management • Synchronization • Real-time analysis 	Minimizes programming complexity w/optimized drivers <ul style="list-style-type: none"> • Low-level driver libraries • Thread-safe Peripheral API 	<ul style="list-style-type: none"> • USB Stack • Networking Stack • CC3000 WiFi Stack • Open Source FAT f/s • Libraries & Examples

Development Tools

Development Tools for MSP430

Evaluation License	<ul style="list-style-type: none"> ❑ 32KB code-size or 30-day limit ❑ Upgradeable 	<ul style="list-style-type: none"> ❑ Full function ❑ JTAG limited after 90-days 	N/A	N/A
Compiler	IAR C/C++	TI C/C++	MSPGCC*	MSPGCC*
Debugger and IDE	<ul style="list-style-type: none"> ❑ C-SPY ❑ Embedded Workbench 	<ul style="list-style-type: none"> ❑ TI or GDB ❑ CCStudio (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 <ul style="list-style-type: none"> ❑ serial.printf() ❑ LED or scope 	MSP-FET430UIF \$99

MSPGCC*: RedHat GCC compiler in development

ULP...

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: (co)printf
ULP 6.1	Avoid multiplication on devices without hardware multipliers
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

Grace...

Grace™

- ◆ A free, graphical user interface for use with CCStudio 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

Support & Community

TI Wiki: <http://processors.wiki.ti.com>

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

Texas Instruments Wiki

Welcome to the Texas Instruments Wiki

Searching

Google™ Custom Search

- [RSS feed for wiki changes](#)
- Check out the [FAQ](#) section, [GSG](#) category for Getting Started Guides or [Training](#) homepage for online training material.

Embedded Processors

Microcontrollers		ARM Based Processors			Digital Signal Processors	
16-bit ultra low power MCU	32-bit Real-time MCUs	32-bit ARM MCU	32-bit ARM Safety MCU	32-bit ARM MPU Performance	DSP & DSP + ARM	Multicore DSP
MSP430	C2000	Stellaris Cortex-M	Hercules Cortex-R4	Sitara Cortex-A8 and ARM9	C6000 Single Core	C6000 Multicore

Software & Development Tools

Development Tools

- [Code Composer Studio™ IDE](#) Integrated development environment for TI embedded processors.
- [Code Generation Tools](#) Compiler, assembler, linker and associated tools.
- [Emulation XDS JTAG emulators](#)
- [C6000 Base of Development Tools](#)

Technical Training Organization (TTO)

Welcome to the Texas Instruments Wiki

Searching and RSS Feed

- [G](#) Search for an article here:

Google™ Custom Search

- [RSS feed for Wiki changes](#)
- Check out the [FAQ](#) section, [GSG](#) category for Getting Started Guides or [Training](#) homepage for online training material.

Embedded Processors

Microcontrollers		ARM Based Processors		
16-bit ultra low power MCU	32-bit Real-time MCUs	32-bit ARM MCU	32-bit ARM Safety MCU	32-bit ARM MPU Performance
MSP430	C2000	Stellaris Cortex-M	Hercules Cortex-R4	Sitara Cortex-A8 and ARM9

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™ 4xx One-Day Workshop](#)

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

[MSP430™ 6xx One-Day 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
Espressif (i.e. Arduino) Chapter for MSP430 Launchpad

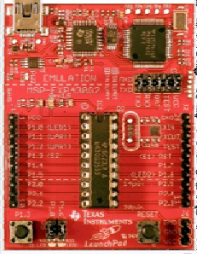
Introduction

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 (170KB)	N/A
1.	Introduction	An introduction to the MSP430 Value Line series of products	Link	Link
2.	Code Composer Studio (CCS)	Introduction to Code Composer Studio IDE.	Link	Link



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 myTI 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

Search through 1,055,110 questions and answers in TI E2E Community [Advanced Search](#)

Choose a support forum to post a new question

ARM®-based Processors

Amplifiers

DLP® & MEMS

Applications

Digital Signal Processors

Broadband RFIF & 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

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

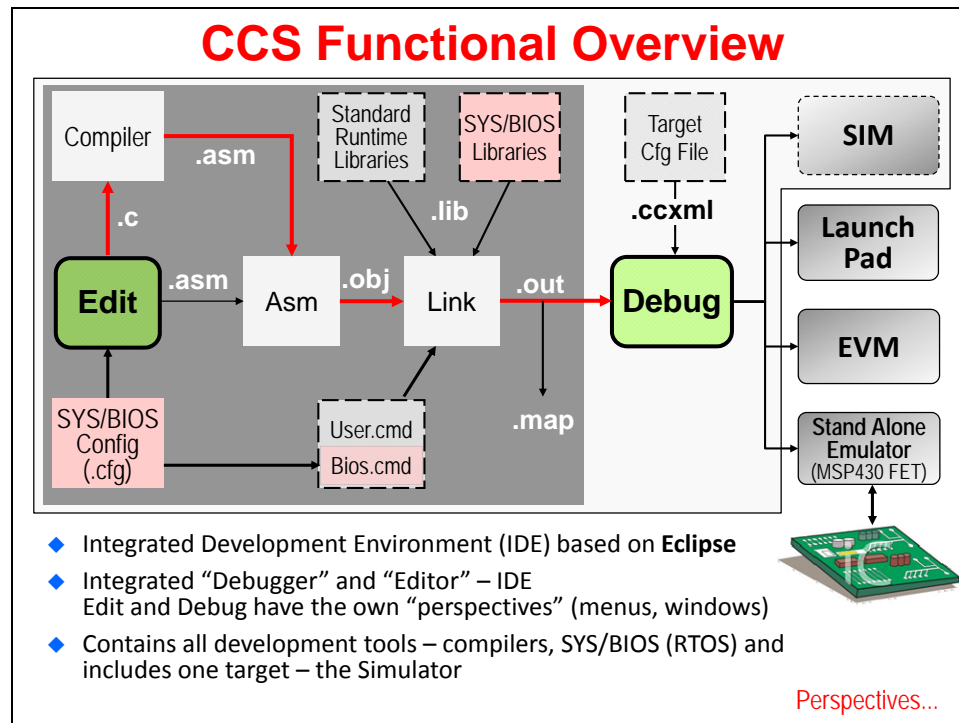
TI E2E Top Contributors

Top Contributors | Top 11 Contributors

<http://e2e.ti.com>

Examining CCSv5

Functional Overview



Perspectives

CCSv5 GUI – EDIT Perspective

Menus & Buttons

- Specific actions related to EDITING

Perspectives

- EDIT and DEBUG

Project Explorer

- Project(s)
- Source Files

Source EDITing

- Tabbed windows
- Color-coded text

Outline View

- Declarations and functions

If you click on the "Debug" perspective, the windows change to...

CCSv5 GUI – DEBUG Perspective

Menus & Buttons

- Related to DEBUGING
- Play, Pause, Terminate

Connection Type

- Specified in Target Cfg file
- What options do users have when connecting to a target?

DEBUG Windows

- Watch Variables
- Memory Browser
- PC execution point
- Console Window

ccxml...

Target Config & Emulation

Target Configuration and Emulators

The diagram illustrates the target configuration process. A 'Target Cfg File' (highlighted with a red box) feeds into a 'Debug' component (green box). From 'Debug', the flow goes to 'SIM', 'Launch Pad', 'EVM', and 'EMU'. The 'EMU' component is connected to a physical development board (shown as a green PCB).

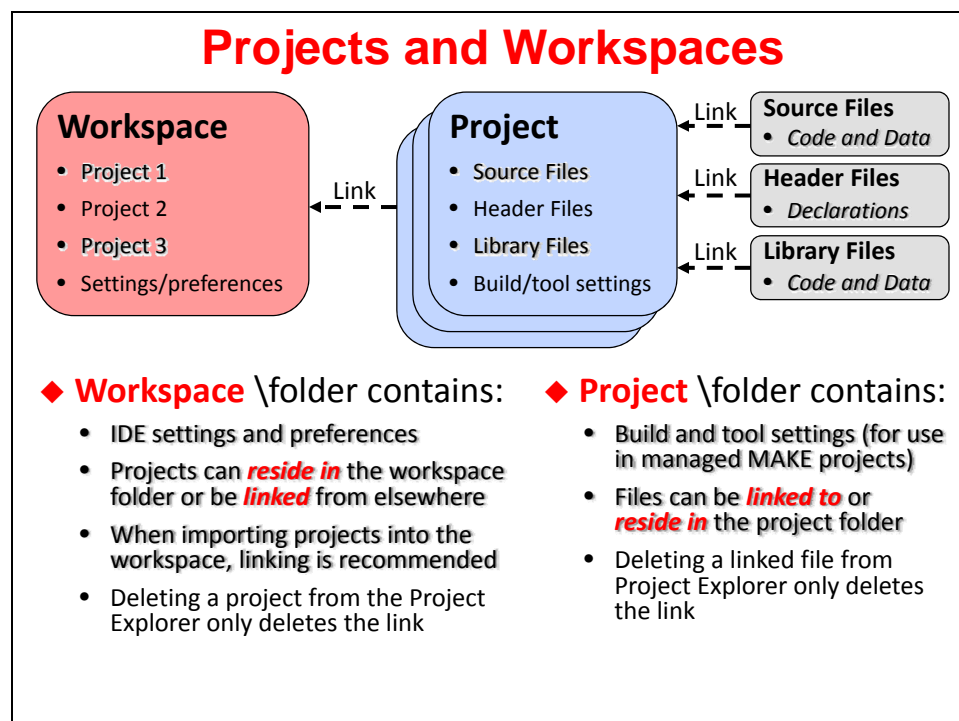
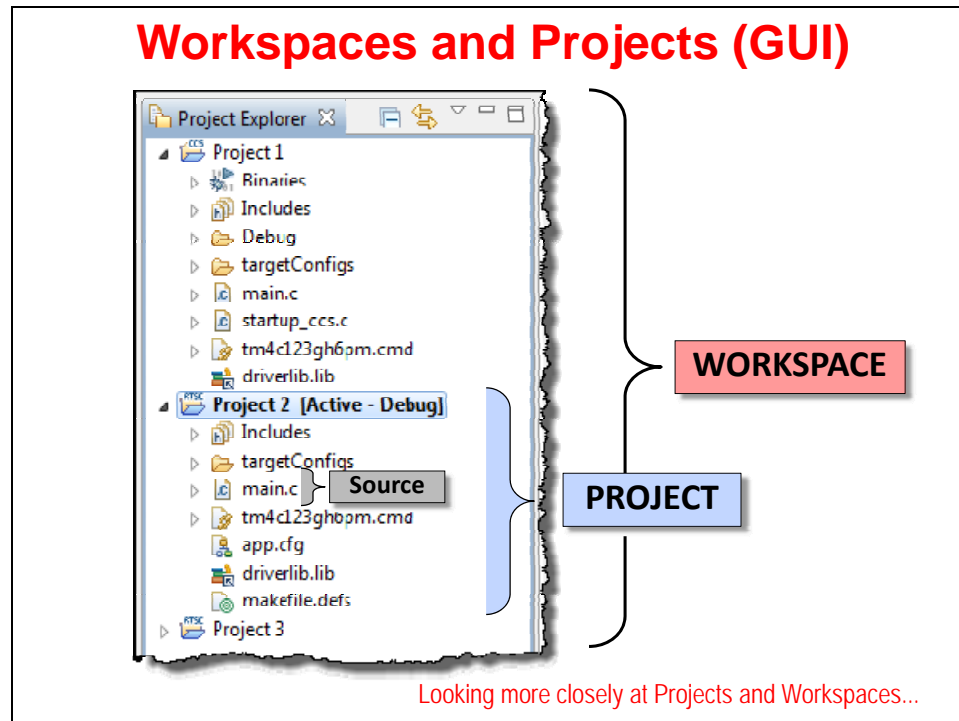
- ◆ The Target Configuration File specifies
 - Connection to the target (e.g. USB FET)
 - Target device (e.g. MSP430F5529)
 - GEL file (if applicable) for h/w setup

The screenshot shows the 'Basic' tab of the Target Configuration File (TTO.ccxml). The 'General Setup' section is visible, with the following options:

- Connection: Stellaris In-Circuit Debug Interface
- Board or Device: type filter text
- Tiva TM4C123GH6PM
- Tiva TM4C123GH6PZ

- ◆ 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

Workspaces & Projects



Creating a Project

Creating a New Project

File → New → CCS Project
(in Edit perspective...)

- ◆ **Project Location**
 - Default = workspace
 - Manual = anywhere you like
- ◆ **Connection**
 - If target is specified, user can choose “connection” (i.e. the target config file)
- ◆ **Templates**
 - No BIOS? Choose “Empty”
 - BIOS? Choose BIOS template

Adding files to the project...

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*
 - Can select a “reference” point (default is project’s directory)

Licensing/Pricing

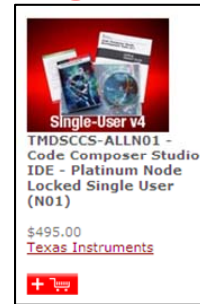
CCSv5 Licensing and Pricing

◆ 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)

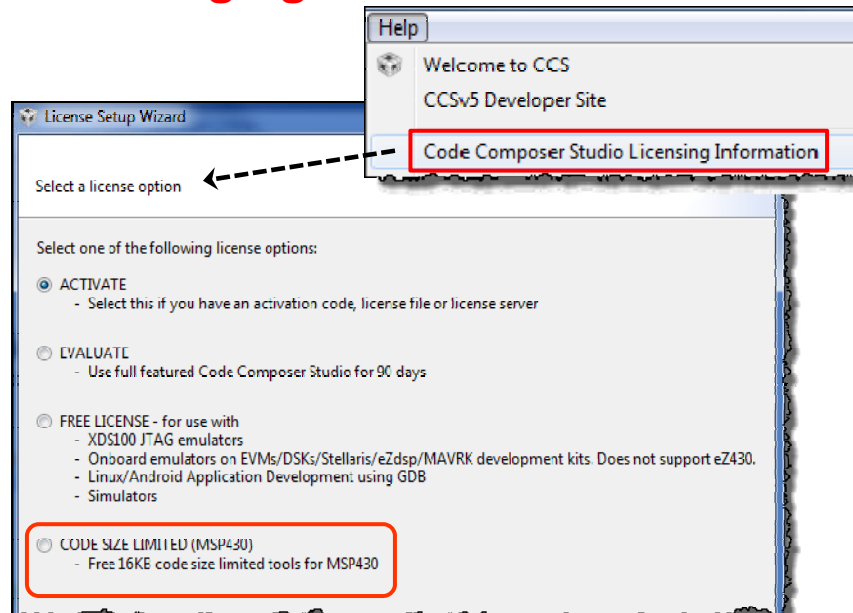


Item	Description	Price	Annual
Platinum Eval Tools	Full tools with 90 day limit (all EMU)	FREE	
Platinum Bundle	XDS100; Simulators; many TI dev'l boards (such as Tiva-C Launchpad)	FREE	
Platinum Node Lock	Full tools tied to a machine	\$445 (1)	\$99
Platinum Floating	Full tools shared across machines	\$795	\$159
MSP430 Code-Limited	MSP430 (16KB code limit)	FREE	

(1) Download version; \$495 when disc is shipped to you

Managing the CCS license...

Changing CCS User License



Writing MSP430 C Code

Build Config & Options

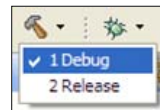
Compiler Build Options

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

	Options	Description
	-mv7M4	Generate Cortex M4 code
	-vmbsp	Generate MSP430 code
Debug	-g	Enables src-level symbolic debugging
	-ss	Interlist C statements into assembly listing
Optimize (release)	-o3	Invoke optimizer (-o0, -o1, -o2/-o, -o3)
	-mf	Speed/code size tradeoff
	-k	Keep asm files, but don't interlist

- ◆ To make things easier, CCS creates two BUILD CONFIGURATIONS:

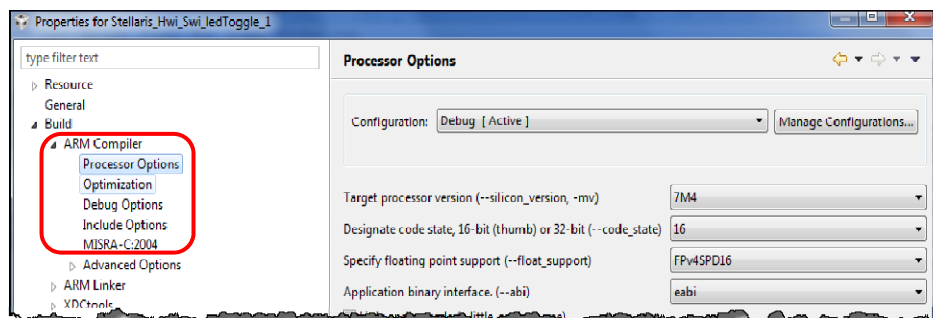
- *Debug* (-g, no opt) – great for LOGICAL debug
- *Release* (no -g, opt) – good for PERFORMANCE/Size
- Users can create their own custom build configs



How do you CHANGE compiler build options or configurations?

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 Opt*):



Data Types

MSP430 C Data Types (ELF format)

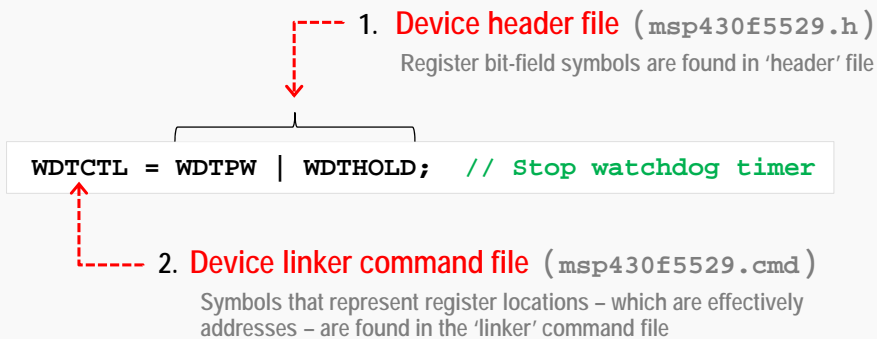
Type	Bits	Representation
char	8	(aligned to 8-bit boundary)
short	16	Binary, 2's complement
int	16	Binary, 2's complement
long	32	Binary, 2's complement
long long	64	Binary, 2's complement
float	32	IEEE 32-bit
double	64	IEEE 64-bit
long double	64	IEEE 64-bit

- ◆ 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)

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.



Device Specific Files (.h/.cmd)

- ◆ New CCS projects automatically contain two files based upon the “Target CPU” selection:
 1. **Device header file** (`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** (`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

MSP430 Compiler Intrinsic Functions

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:

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

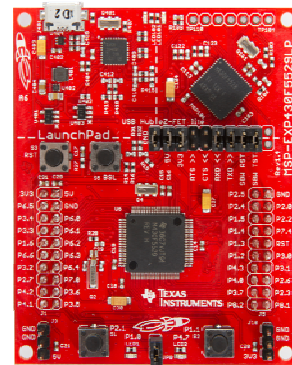
Lab 2 – CCSv5 Projects

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.

Since none of the Value Line MSP430 devices have more than 16K of flash memory, the free, 16K license of Code Composer Studio can be considered fully functional. If you want to work with larger MSP430 (or other) devices, you'll need to purchase a license.

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 CCSv5 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	2-15
<i>Lab 2 – CCSv5 Projects.....</i>	<i>2-17</i>
Lab 2a – Creating a New CCS Project	2-19
Intro to Workshop Files	2-19
Start Code Composer Studio and Open a Workspace	2-20
00430 ... Licensed to Develop	2-21
“CCS Edit” Perspective	2-22
Create a New Project	2-23
Build The Code (ignore advice).....	2-26
Debug The Code	2-27
Fix Your Project.....	2-31
Build, Load, Connect and Run ... with the Easy Button	2-32
Lab 2b – My First Blinky.....	2-33
Create and Examine Project	2-33
Build, Load, Run.....	2-34
Restart, Single-Step, Run To Line	2-35
Lab 2c – Putting the OOB back into your device	2-37
(Optional) Lab 2d – MSP430Flasher	2-38
Programming the UE OOB demo using MSP430Flasher.....	2-38
Programming Blinky with MSP430Flasher.....	2-41
Cleanup.....	2-42

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    (coming 1st Quarter 2014)
```

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.x should have already been installed during the workshop installation procedure.

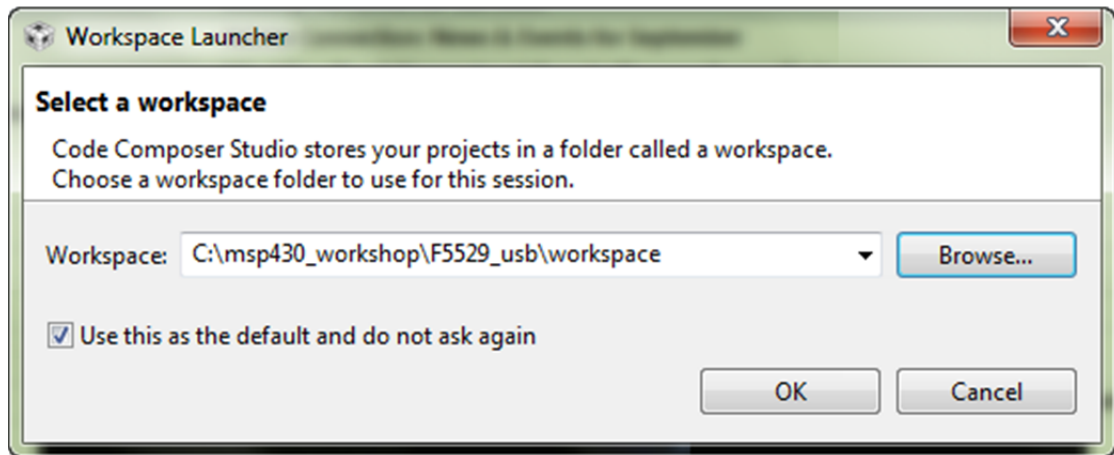
2. Start Code Composer Studio (CCS).

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 everytime 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 v5

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

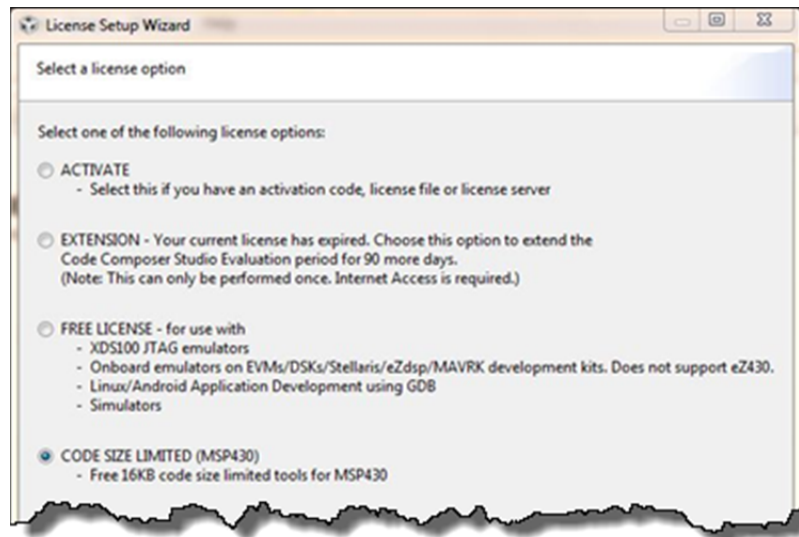
00430 ... Licensed to Develop

5. Set CCSv5 license ... if required.

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:

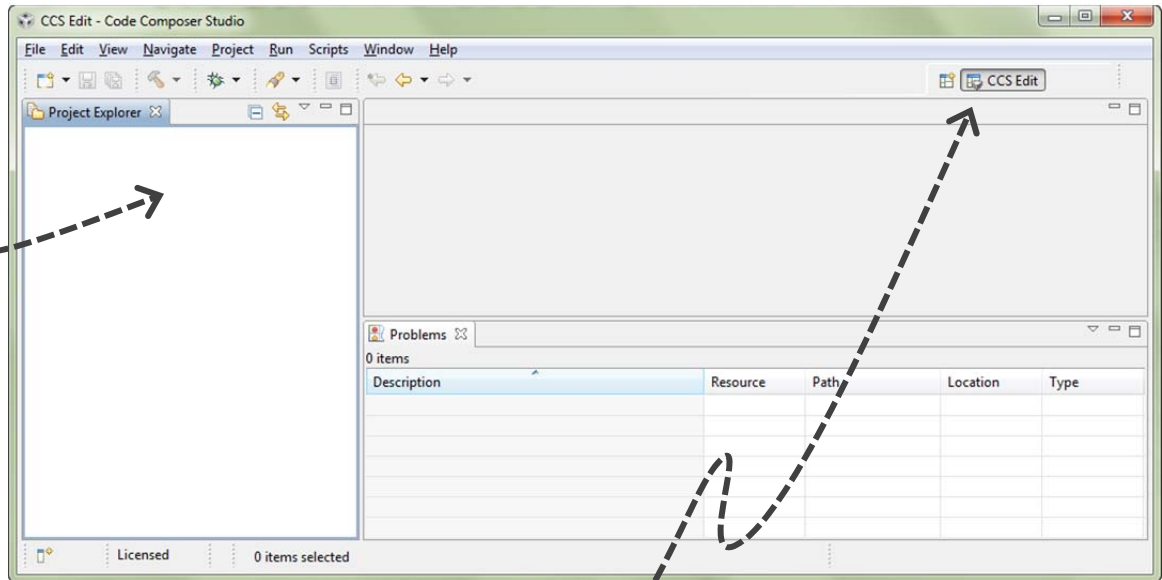
CODE SIZE LIMITED (MSP430)

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.



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.

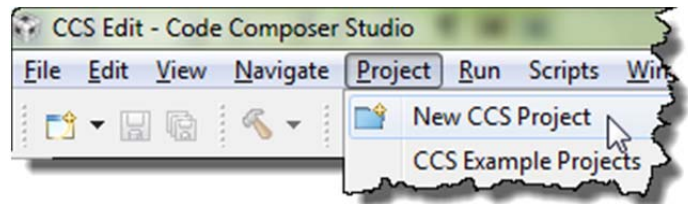
Hint: Most of us find the `Window → Reset Perspective...` handy for those times when we’ve messed our windows up a bit too much.

Create a New Project

7. Select New CCSP 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:

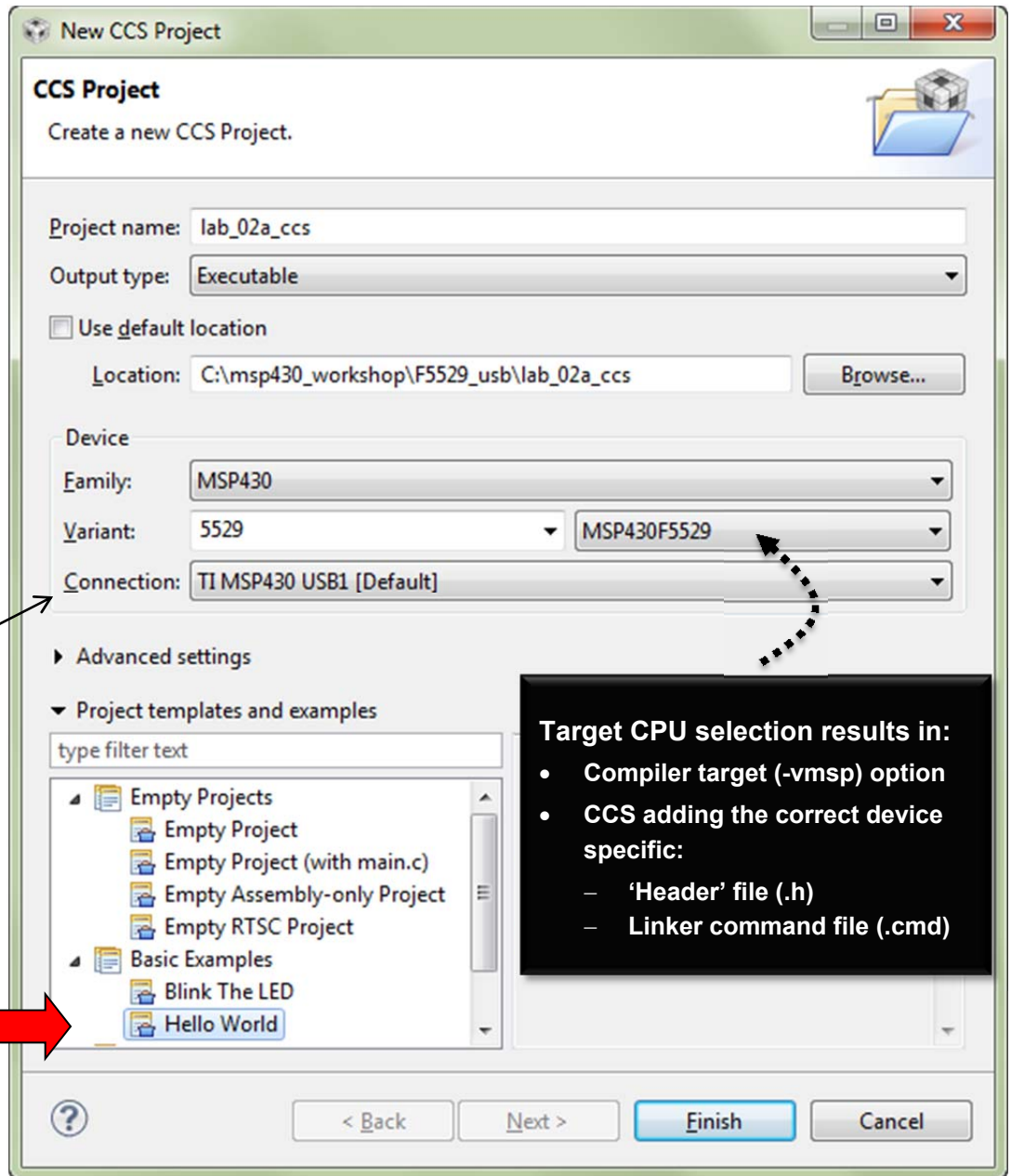
File → New → CCS Project



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) 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 .ccsxml file for you)
- h) Select template: Hello World



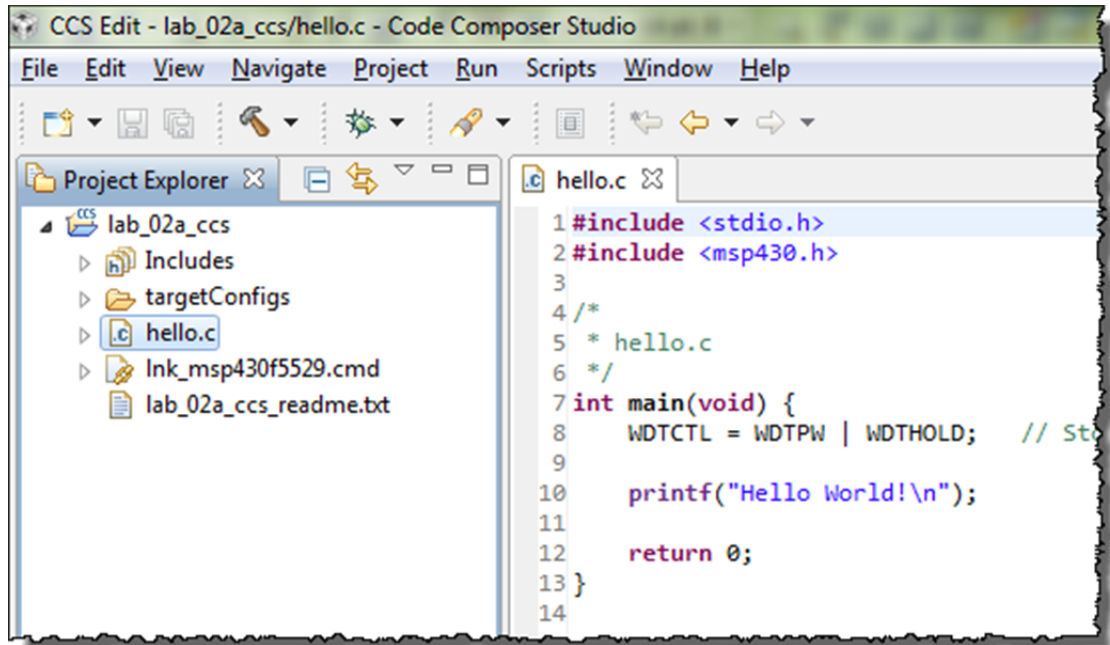
Target CPU selection results in:

- Compiler target (-vmsp) option
- CCS adding the correct device specific:
 - 'Header' file (.h)
 - Linker command file (.cmd)

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)

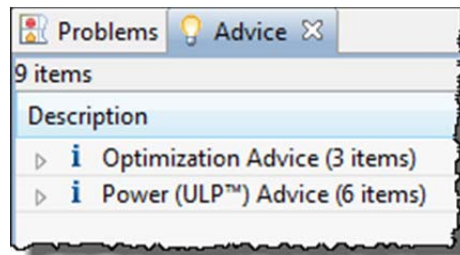
12. Build your project using “the hammer” and check for errors.

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.

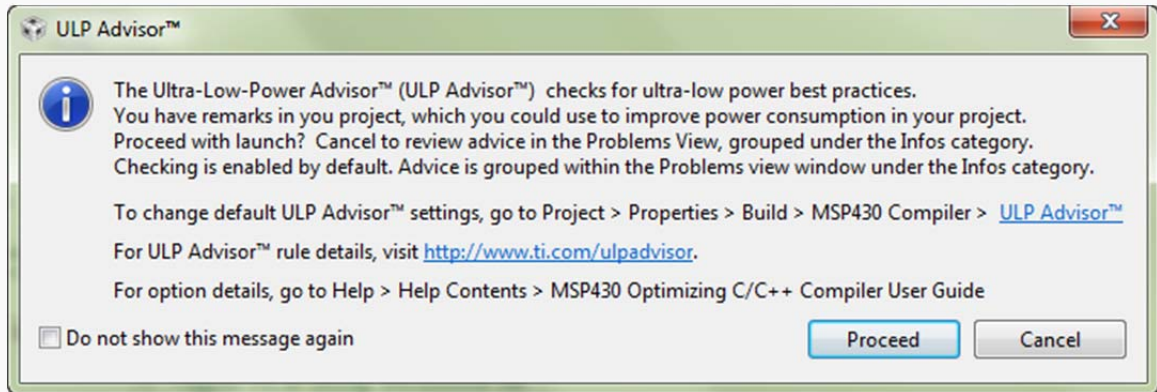


At this point, we’re just going to ignore their advice. It’s better to get code running first. Later, we return and investigate some of these items further.

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 nanoamp.

Debug The Code

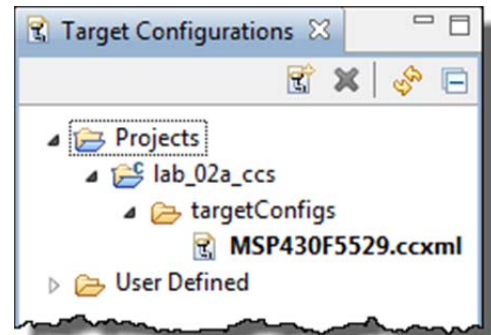
Starting up the debugger is a 3-step process. You could even call it five steps, if you include building and running the code. (In a few minutes, we'll show you a quick shortcut.)

13. Launch a debug session.

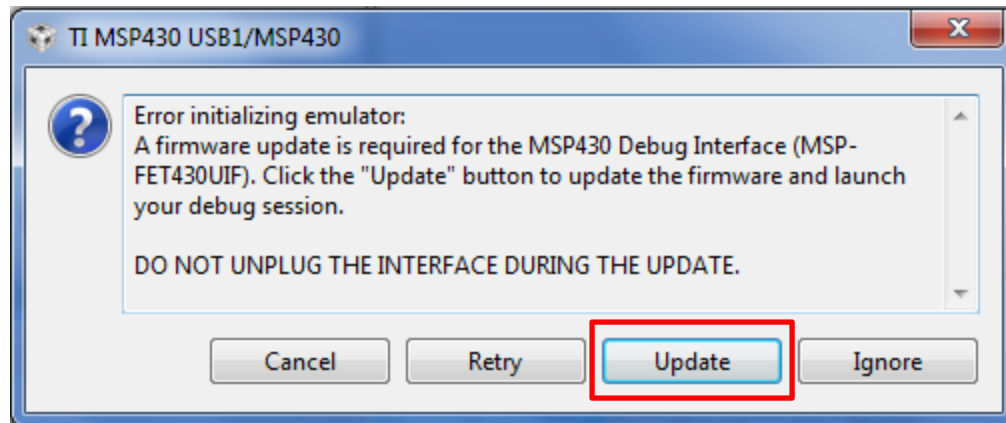
This starts the *CCS Debugger* and then switches to the *Debug* perspective.

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

Rt-click .ccxml file → Launch Selected Configuration



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

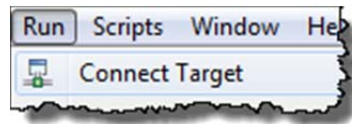


This occurs when CCS finds that the FET firmware – that is, the firmware in your Launchpad's debugger – is out-of-date. We recommend that you choose to update the firmware. Once complete, CCS should finish launching the debugger.

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

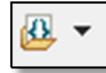
If you run into this, check for each of these possibilities. In the case of the Windows USB driver try:

- 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.

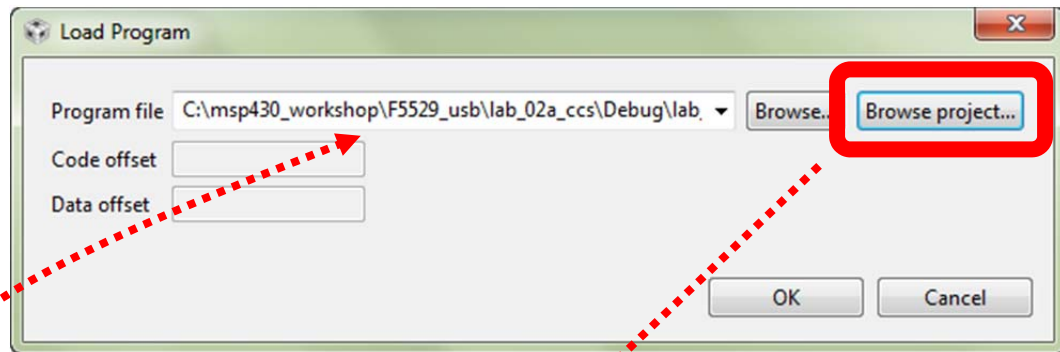
15. Load the code.

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

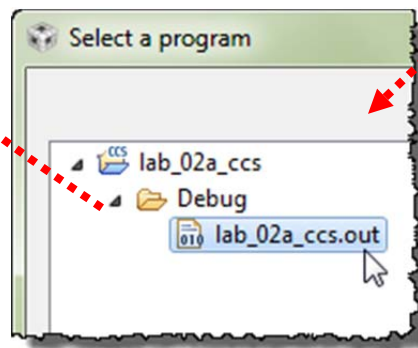
Run → Load → Load Program – or – use the download button:



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



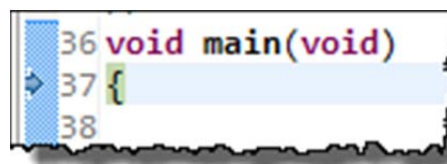
... and navigate to the executable (.out) file in your project:

**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):

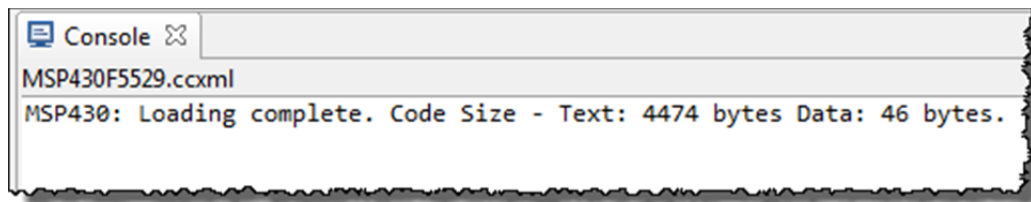


Warning: Pause is different than Terminate !!!

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.

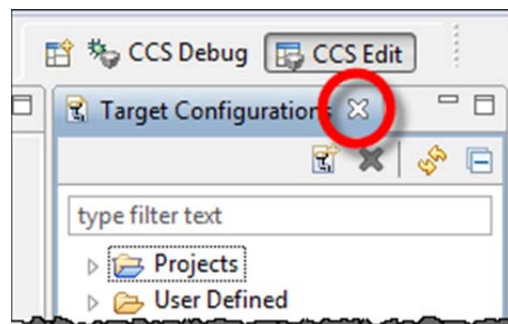
OK, this time we really want to terminate our debug session.

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.



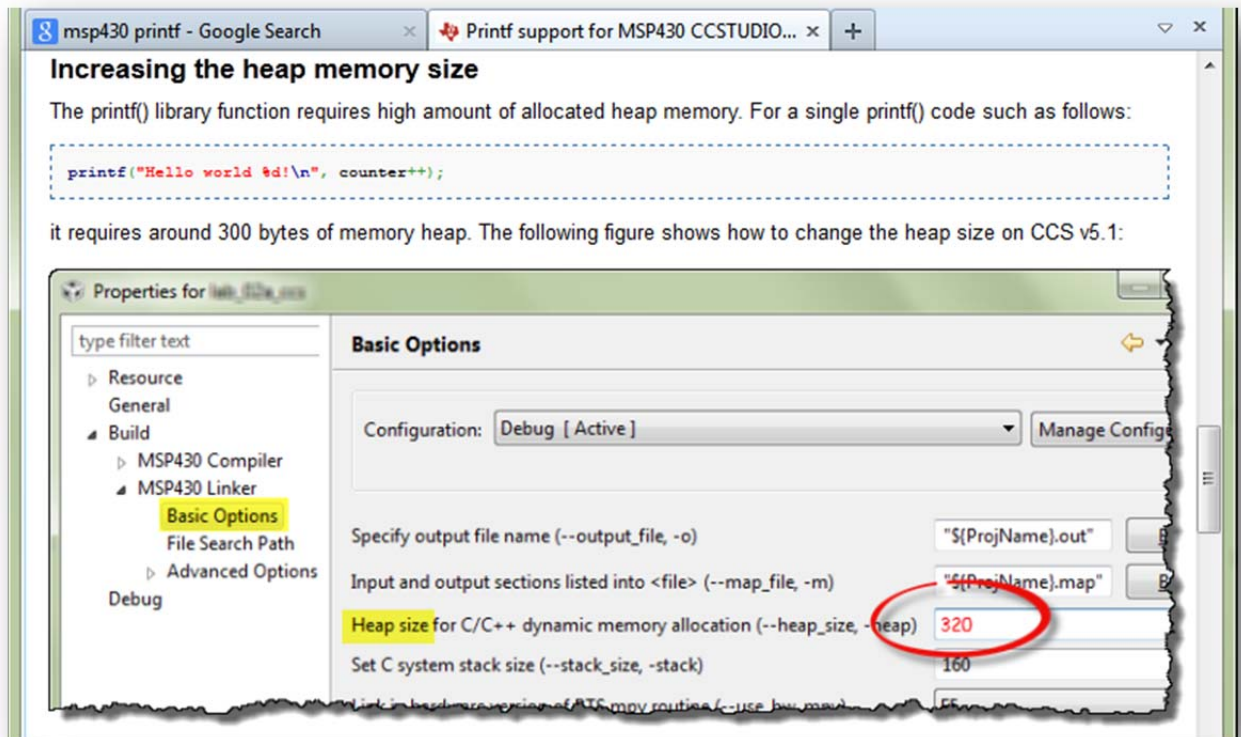
Fix Your Project

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 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, when/if that needs to be done.

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

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

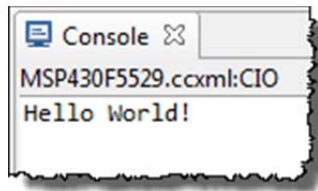
Here's the “easy button” (i.e. one button) method for debugging your code. First, make sure you terminated your previous debug session and you are in the Edit perspective.

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.

Closing a project is both handy and prevents errors.

Rt-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.

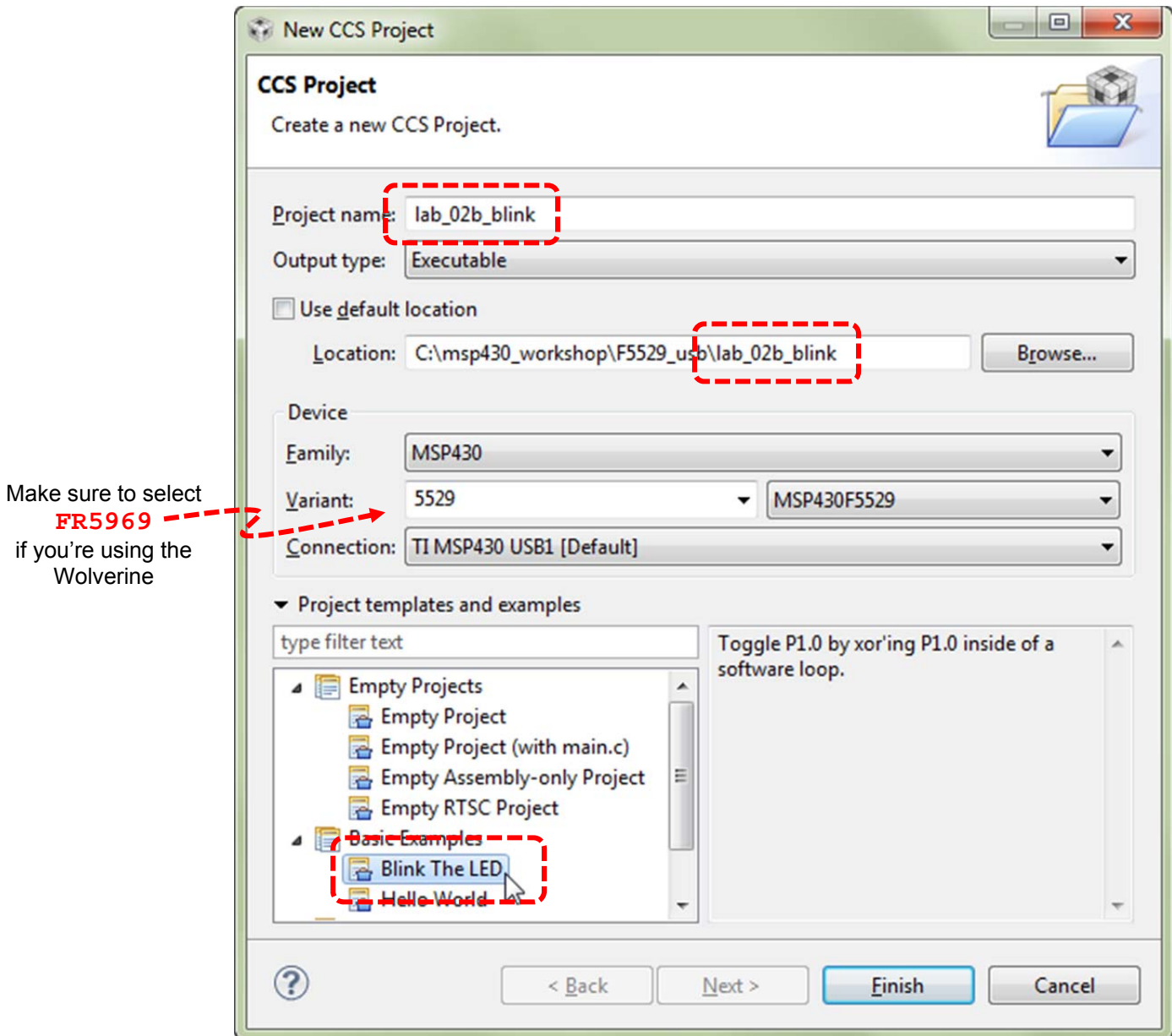
Lab 2b – My First Blinky

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 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 | WDTHOLD;    // Stop watchdog timer

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

    for(;;) {
        volatile unsigned int i; // volatile to prevent op-timization

        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.*)

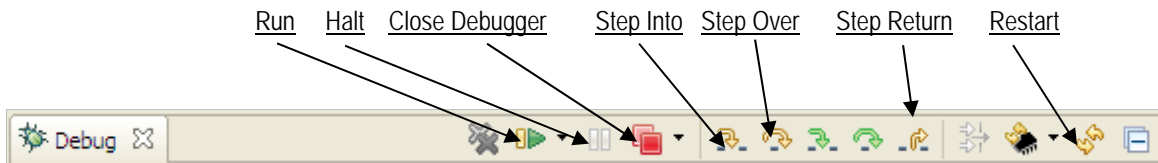
```

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 function 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.

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.

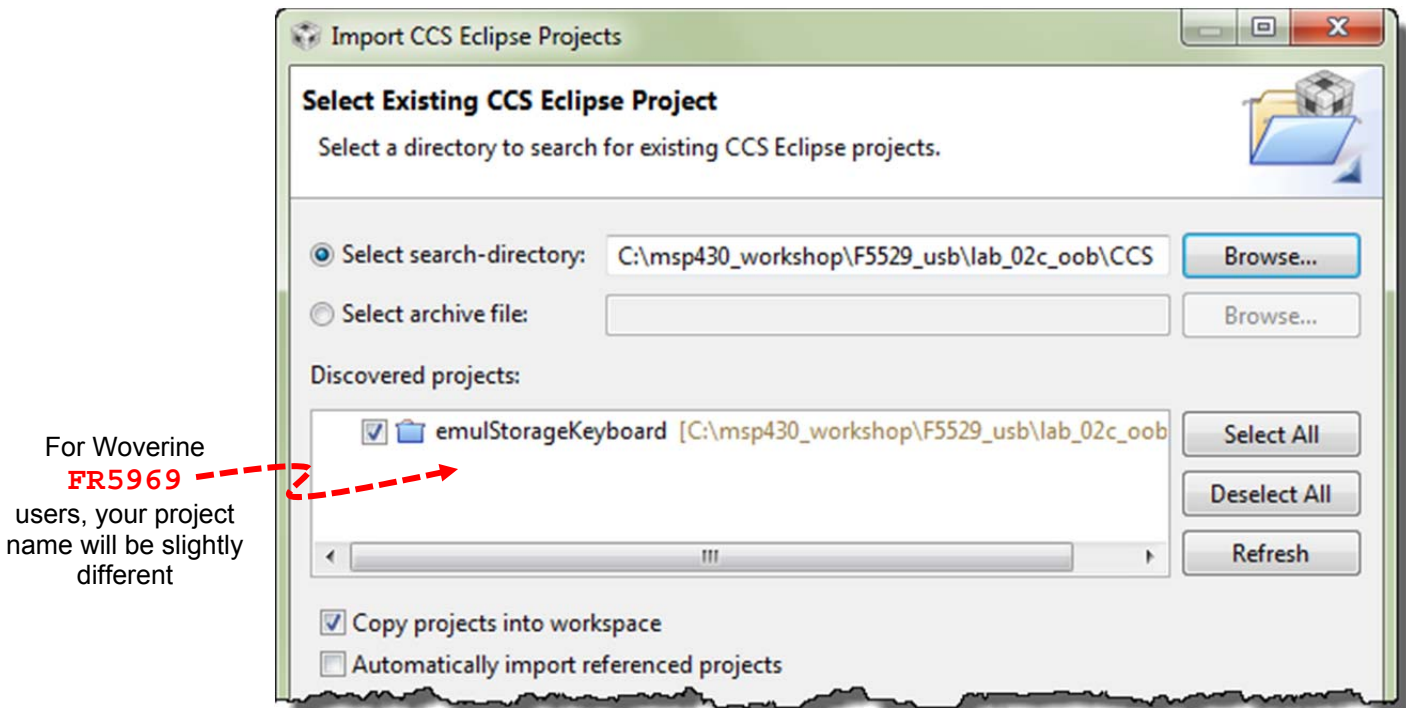
1. Import OOB demo project.

As part of the workshop files, we already placed a copy of the project used to build the original Launchpad demo into the `lab_02c_oob` folder.

Project → Import Existing CCS Eclipse Project

`C:\msp430_workshop\F5529_usb\lab_02c_oob\CCS\`

Browse to the `lab_02c_oob` folder for your target and import the project.



Note, this project was obtained from the Launchpad's webpage: ti.com/msp-exp430f5529lp

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

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.

3. 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.)

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

This runs the original demo that was just re-programmed into Flash. (We unplugged from Windows to get Windows to recognize the memory-stick (MSC) feature of the demo program. (You can refer back to `lab_01_oob` 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\<<target>\lab_02d_flasher\ue.bat
```

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

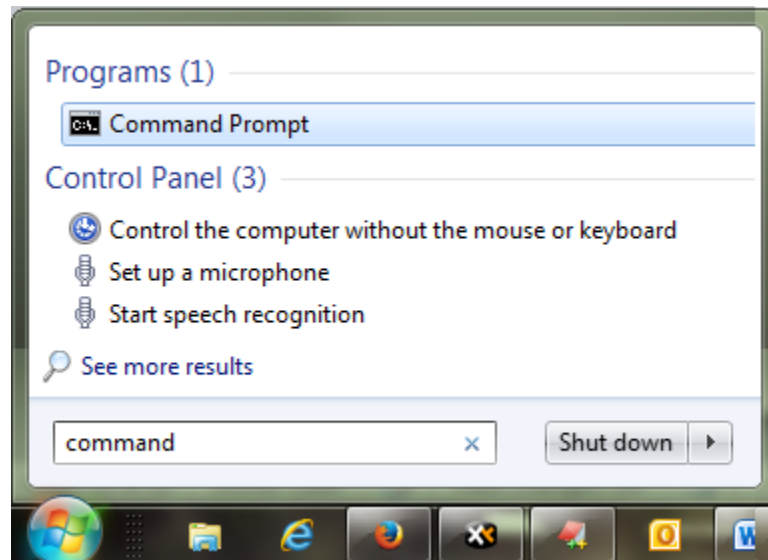
```
CLS
C:\ti\msp430\MSP430Flasher_1.2.2\MSP430Flasher.exe -n MSP430F5529
-w "C:\msp430_workshop\F5529_usb\workspace\emulStorageKeyboard\Debug\emulStorageKeyboard.txt" -v
pause
```

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

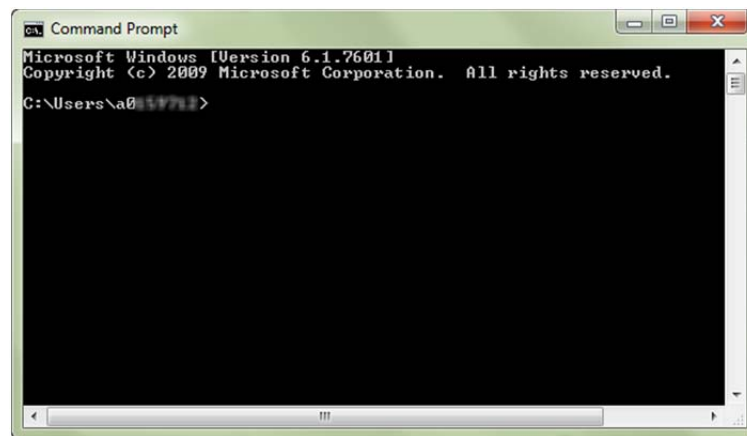
We used the default locations for MSP430Flasher and our lab exercises. You will have to change them if you installed these items to other locations on your hard drive.

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\
```

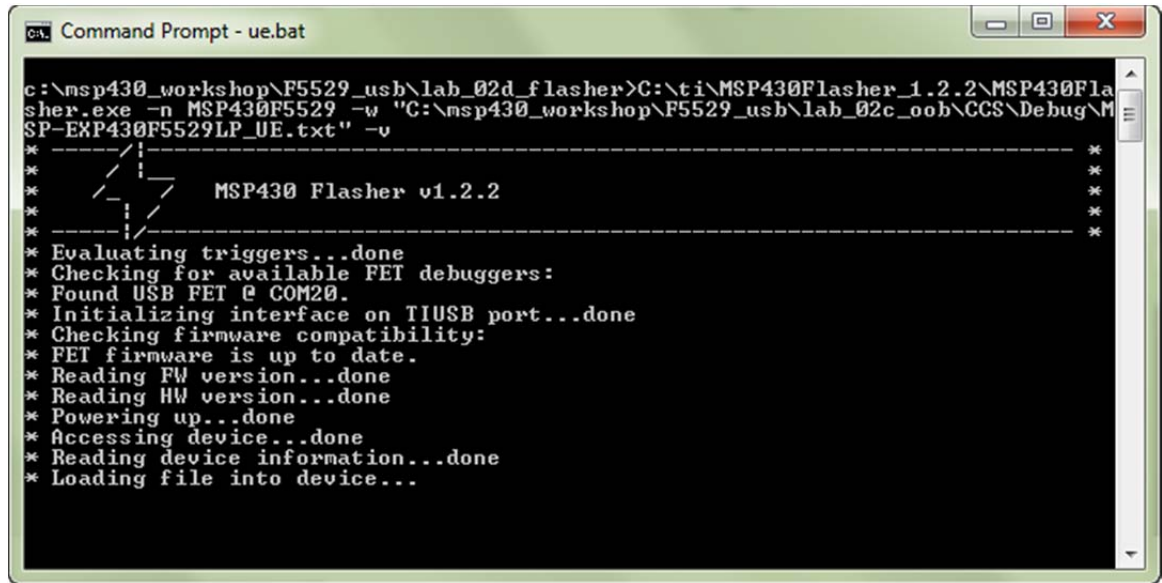
Once there, you should be able to list the directories contents using the *dir* command.

```
dir
```

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

ue.bat ↵

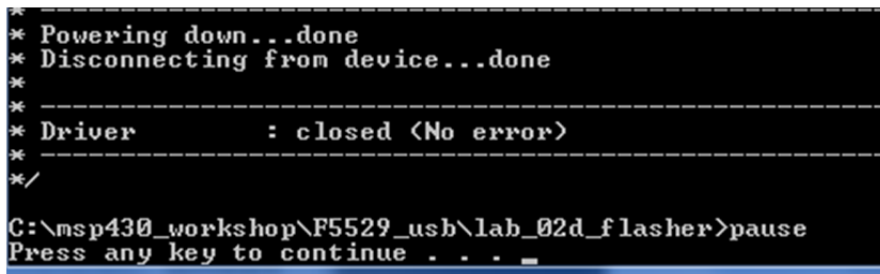
You should see it running ... here's a screen capture we caught mid-programming:



6. When complete, hit enter to finish the batch program.

We ended our batch program with a *pause* since – depending upon how you invoked it – the command window could close automatically. Pause forces the window to stay open, so that you can see the feedback shown above.

Below is what the command window should look like right before you hit any key, which should end the batch program.



7. Once again, verify the Launchpad program works.

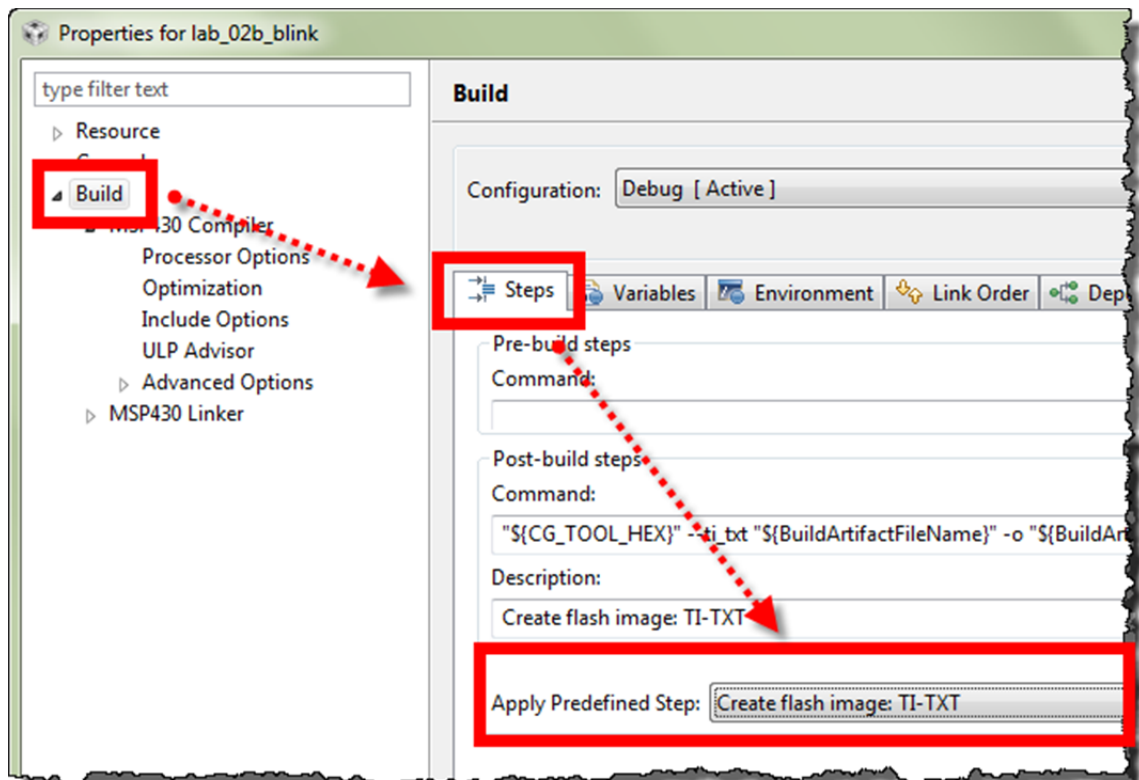
Programming Blinky with MSP430Flasher

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.

8. Open your `lab_02b_blink` project.
9. Open the project properties for you project.

With the project selected, hit *Alt-Enter*.

10. Change the required build setting, as shown below.



This is documented at:

[http://processors.wiki.ti.com/index.php/Generating and Loading MSP430 Binary Files](http://processors.wiki.ti.com/index.php/Generating_and>Loading_MSP430_Binary_Files)

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.

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

```
C:\msp430_workshop\

```

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

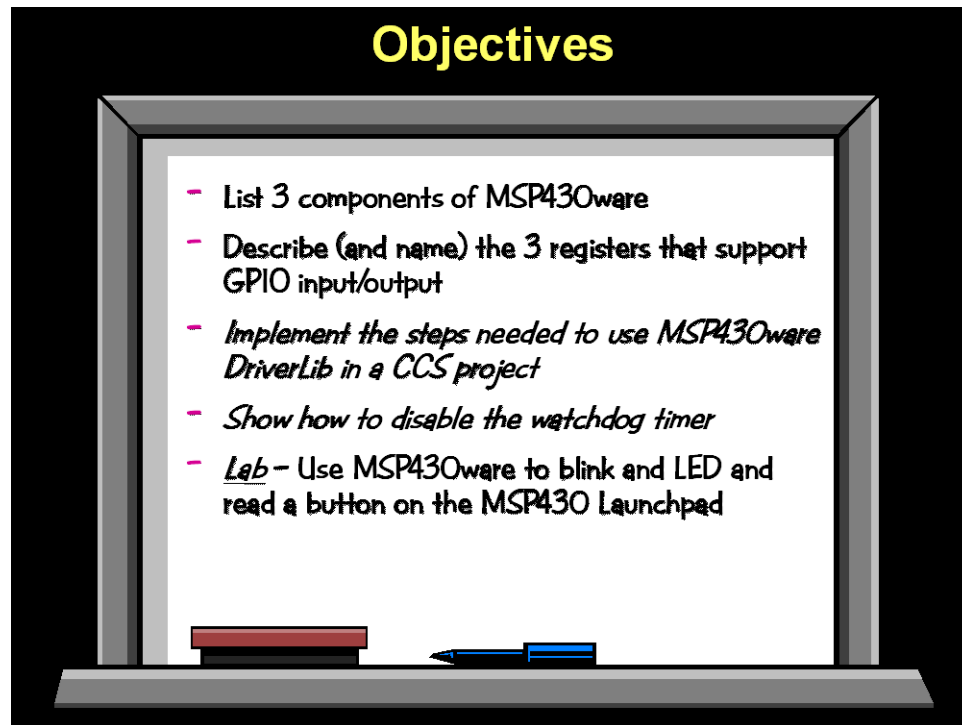
In the previous lab exercise blinked an LED on the MSP430 Launchpad, but we didn't write the code – we were able to import a generic 'blink' template that ships with CCSv5.

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 to the many way 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.

Finally, now that we are introducing a library we can link into our project, we explore the concept of "portable projects". By creating and using IDE path variables, it becomes easy to migrate projects from one computer to another. Another benefit is that it becomes much easier to upgrade to new library versions, as they become available.

Learning Objectives



Chapter Topics

Using GPIO with MSP430ware	3-1
<i>MSP430ware (DriverLib)</i>	3-3
<i>MSP430 GPIO</i>	3-5
GPIO Basics.....	3-5
Flexible Pin Usage (Muxing)	3-8
Summary	3-10
<i>Before We Get Started Coding</i>	3-11
#include Files	3-11
Disable Watchdog Timer.....	3-12
Pin Unlocking (Wolverine).....	3-12
<i>Lab 3</i>	3-13
Lab 3 Worksheet.....	3-15
MSP430ware DriverLib	3-15
GPIO Output	3-15
GPIO Input	3-16
Lab 3a – Blinking an LED.....	3-17
???	3-18
Add MSP430ware DriverLib.....	3-19
Add the Code to <code>main.c</code>	3-21
Debug.....	3-22
Lab 3b – Reading a Push Button	3-24
File Management	3-24
Add Setup Code (to reference push button)	3-26
Modify Loop.....	3-27
Verify Code.....	3-28
Optional Exercises	3-28
<i>Chapter 3 Appendix</i>	3-29

MSP430ware (DriverLib)

- ◆ Libraries
 - ◆ DriverLib*
 - ◆ Graphics
 - ◆ USB Stack
 - ◆ CapTouch
 - ◆ MathLib
 - ◆ IEC60730
- ◆ Examples
 - ◆ All device generations
 - ◆ Development boards
- ◆ Software Tools
 - ◆ Grace
 - ◆ ULP
 - ◆ Optimization Advisor

MSP430ware

Welcome to MSP430ware

Navigate using the tabs!

Navigate using the folder tree!

* Other tools/libraries covered in later workshop chapters

Driver Library vs Traditional C Coding (PWM example)

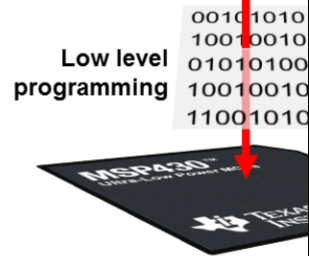
Driver Library

```
GPIO_setAsPeripheralModuleFunctionOutputPin(PARAMETERS);
Timer_generatePWM(PARAMETERS)
```

Traditional C code
with Header Files

```
P2DIR |= 0x04;
TA1CCTL1 = OUTMOD_7;
P2SEL |= 0x04;
TA1CCR1 = 384;
TA1CCR0 = 511;
TA1CTL = TASSEL_1 + MC_1 + TACLK;
```

- ◆ Driver Library offers easy-to-understand functions
- ◆ No more cryptic registers to configure
- ◆ Functional coding of peripherals rather than bitwise programming
- ◆ High-level API makes it easy to port code between most MSP430 devices
- ◆ Minimal overhead



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

Summary

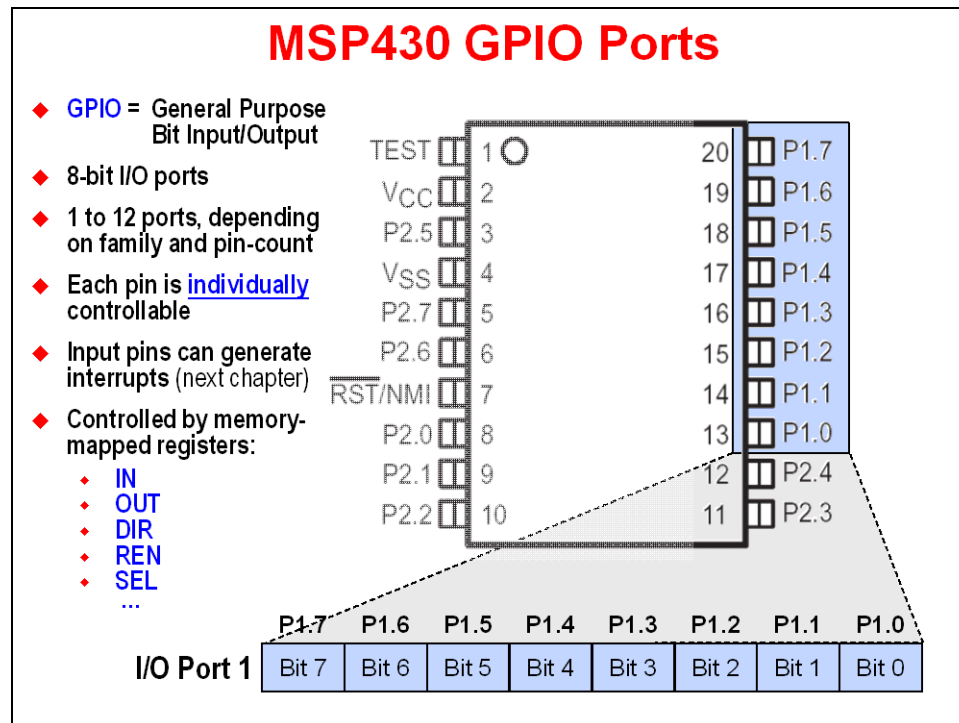
Name 3 ways to program GPIO:

1. Using device specific header & command files (.h/.cmd) Ch2
2. MSP430ware DriverLib (F5xx/6xx and Wolverine devices) Ch3
3. Grace graphical driverlib tool (Value-line and Wolverine devices) *

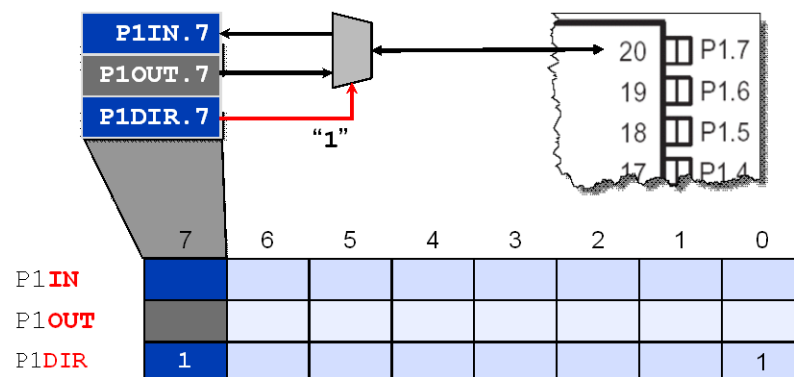
*see G2553 Value-Line Launchpad Workshop (Ch8)

MSP430 GPIO

GPIO Basics



PxDIR (Pin Direction): Input or Output



- ◆ PxDIR.y: 0 = input
1 = output
- ◆ Register example:
P1DIR &= 0x81;
- ◆ MSP430ware example:

```
#include <driverlib.h>
GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 + GPIO_PIN7 );
```

GPIO Output

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

- ◆ PxOUT.y: 0 = low
1 = high
- ◆ Register example:
P1OUT &= 0x80;
- ◆ MSP430ware example:

```
GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN7 );
```

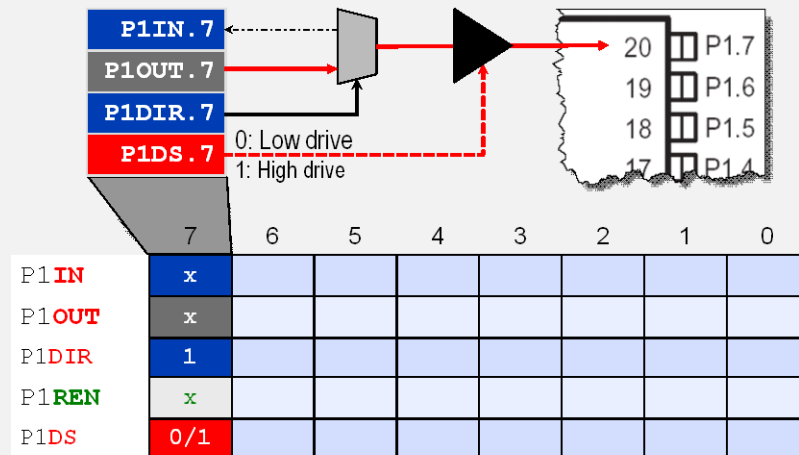
GPIO Input (Resistors)

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

- ◆ Input pins are high-impedance (Hi-Z) state – so they can react to 0 or 1
- ◆ When not driven, Hi-Z inputs may float up/down ... prevent this with pullup/pulldown resistors
- ◆ PxREN enables resistors
PxOUT selects pull-up (1) or -down (0)
- ◆ Lower cost devices may not provide up/down resistors for all ports

```
unsigned short usiButton = 0;
GPIO_setAsInputPinWithPullUpresistor( GPIO_PORT_P1, GPIO_PIN7 );
usiButton = GPIO_getInputPinValue( GPIO_PORT_P1, GPIO_PIN7 );
```

Output Drive Strength (F5xx only)



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

```
GPIO_setDriveStrength( GPIO_PORT_P1, GPIO_PIN7 );
```

Flexible Pin Usage (Muxing)

Controlling GPIO Ports

◆ Most pins on MCU's are multiplexed to provide you with greater flexibility

P1SEL0 & P2SEL1 Example (FR5xx)

PIN NAME (P1.x)	x	FUNCTION	CONTROL BITS/SIGNALS ⁽¹⁾		
			P1DIR.x	P1SEL1.x	P1SEL0.x
P1.0/TA0.1/DMAE0/RTC CLK/A0/C0/REF- /VeREF-	0	P1.0 (I/O)	I: 0; O: 1	0	0
		TA0.CC1A	0	0	1
		TA0.1	1	0	1
		DMAE0	0	1	0
		RTCCLK	1	1	0

Port Map (PM) Module (F5xx only)

◆ Port mapping allows for additional *digital* signals to be mapped to one or several output pins:

- ◆ PM_XXX = port-mappable signal
- ◆ Datasheet specifies which ports can be mapped
- ◆ By default, single configuration per reset (PUC)
- ◆ Port Mapping Reconfigure bit (PMRECNFG) allows for runtime re-configurations

◆ Port mapping configuration is password protected

52	<input type="checkbox"/>	P4.5/PM_UCA1RXD/PM_UCA1SOMI
51	<input type="checkbox"/>	P4.4/PM_UCA1TXD/PM_UCA1SIMO
50	<input type="checkbox"/>	DVCC2
49	<input type="checkbox"/>	DVSS2
48	<input type="checkbox"/>	P4.3/PM_UCB1CLK/PM_UCA1STE
47	<input type="checkbox"/>	P4.2/PM_UCB1SOMI/PM_UCB1SCL
46	<input type="checkbox"/>	P4.1/PM_UCB1SIMO/PM_UCB1SDA
45	<input type="checkbox"/>	P4.0/PM_UCB1STE/PM_UCA1CLK
44	<input type="checkbox"/>	P3.7/TB0OUTH/SVMOUT
43	<input type="checkbox"/>	P3.6/TB0.6
42	<input type="checkbox"/>	P3.5/TB0.5
41	<input type="checkbox"/>	P3.4/UCA0RXD/UCA0SOMI
37	<input type="checkbox"/>	
38	<input type="checkbox"/>	
39	<input type="checkbox"/>	
40	<input type="checkbox"/>	

Summary

GPIO Summary: F5529 vs FR5969 vs G2553

	PA		PB		PC		PD		PJ* (4-bit)						
	P1†	P2	P3	P4	P5	P6	P7	P8 (3-bit)							
PxIN	All Three Devices support Ports 1 and 2		F5529 and FR5969 (only)		F5529 (only)				F55 & FR59						
PxOUT															
PxDIR															
PxREN															
PxDS															
PxSEL															
PxIV										FR5969 (only)					
PxIES															
PxIE															
PxIFG															

F5529 only (80-pin)

FR5969 only (48-pin)

G2553 only (20-pin)

*PJ: 4-bits shared with JTAG pins

†P1: 4-bits shared with JTAG pins ('G2553)

- ◆ Each numbered port has 8 bits, unless noted otherwise
- ◆ At reset, all I/O pins are set to ... input
- ◆ You should initialize all pins (to prevent floating inputs)
- ◆ Analog functions can 'preempt' pin function selection

MSP430ware GPIO Summary

GPIO_getInputPinValue	↙
GPIO_setOutputHighOnPin	
GPIO_setOutputLowOnPin	
GPIO_toggleOutputOnPin	
↓	
GPIO_setAsInputPinWithPullDownresistor	↘
GPIO_setAsInputPinWithPullUpresistor	
↓	
GPIO_setAsInputPin	↘
GPIO_setAsOutputPin	
GPIO_setAsPeripheralModuleFunctionInputPin	
GPIO_setAsPeripheralModuleFunctionOutputPin	
GPIO_setDriveStrength	
↓	
GPIO_interruptEdgeSelect	↘
GPIO_disableInterrupt	
GPIO_enableInterrupt	↘
GPIO_getInterruptStatus	
GPIO_clearInterruptFlag	

PA		PB					
P1†	P2	P3	P4				
All Three Devices support Ports 1 and 2		F5529 and FR5969 (only)					
				FR5969 (only)			

PxIN

PxOUT

PxREN

PxDIR

PxSEL

PxDS

PxIV

PxIES

PxIE

PxIFG

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)

#Include Files

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](#)

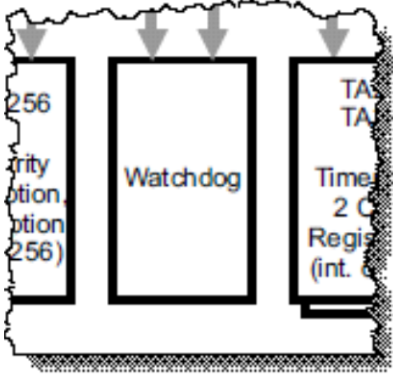
TIMER_A	AES	GPIO
TIMER_B	CRC	PM
TIMER_D	DMA	SPI
WDT_A	MPY32	...

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

Disable Watchdog Timer

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



```

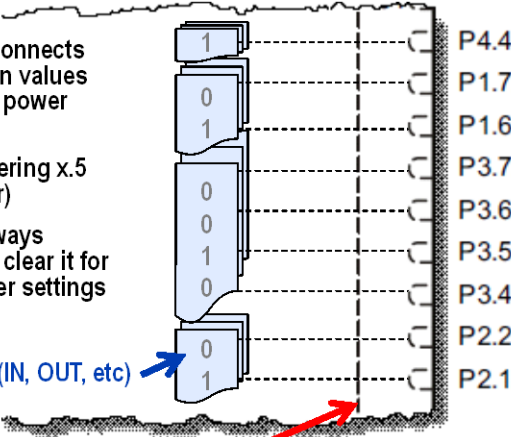
#include <driverlib.h>

WDT_A_hold(WDT_A_BASE); //Stop watchdog timer
    
```

Pin Unlocking (Wolverine)

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 x.5 mode (see Low Power Chapter)
- ◆ **FR58/59 Wolverine** devices always power-on this way – 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( PMM_BASE ); // unlock pins after setting all gpio registers
    
```

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.

While this is just simple LED blinking code, we implement with the MSP430ware DriverLib library. This gives us a good example for learning to use, as well as link in, a library. This library will become even more important as we explore other peripherals in later lab exercises.

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 CCSv5. 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 learn about in a later chapter, would be a much 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: Only lighting LED while "polling" the button is very inefficient!

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

Lab 3 Worksheet

MSP430ware DriverLib

1. Where is your MSP430ware folder located?

-
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"?

```
_____ ;
```

```
_____ ;
```

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)
    _____ ;
    _delay_cycles( ONE_SECOND );
    // Set pin to "0"
    _____ ;
    _delay_cycles( ONE_SECOND );
}
```

GPIO Input

6. 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_60_01_11\driverlib\doc\MSP430F5xx_6xx\

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

8. Assuming you need a pull-up resistor for a GPIO input, write line of code required to setup pin P2.1 for input:

9. Complete the following code to read pin P2.1:

```
volatile unsigned short usiButton1 = 0;
while(1) {
    // Read push-button pin (SW1)

    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 );
    }
}
```

10. 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.

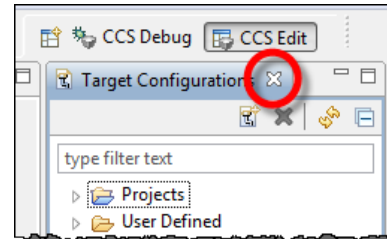
Lab 3a – Blinking an LED

1. Close any open project and file.

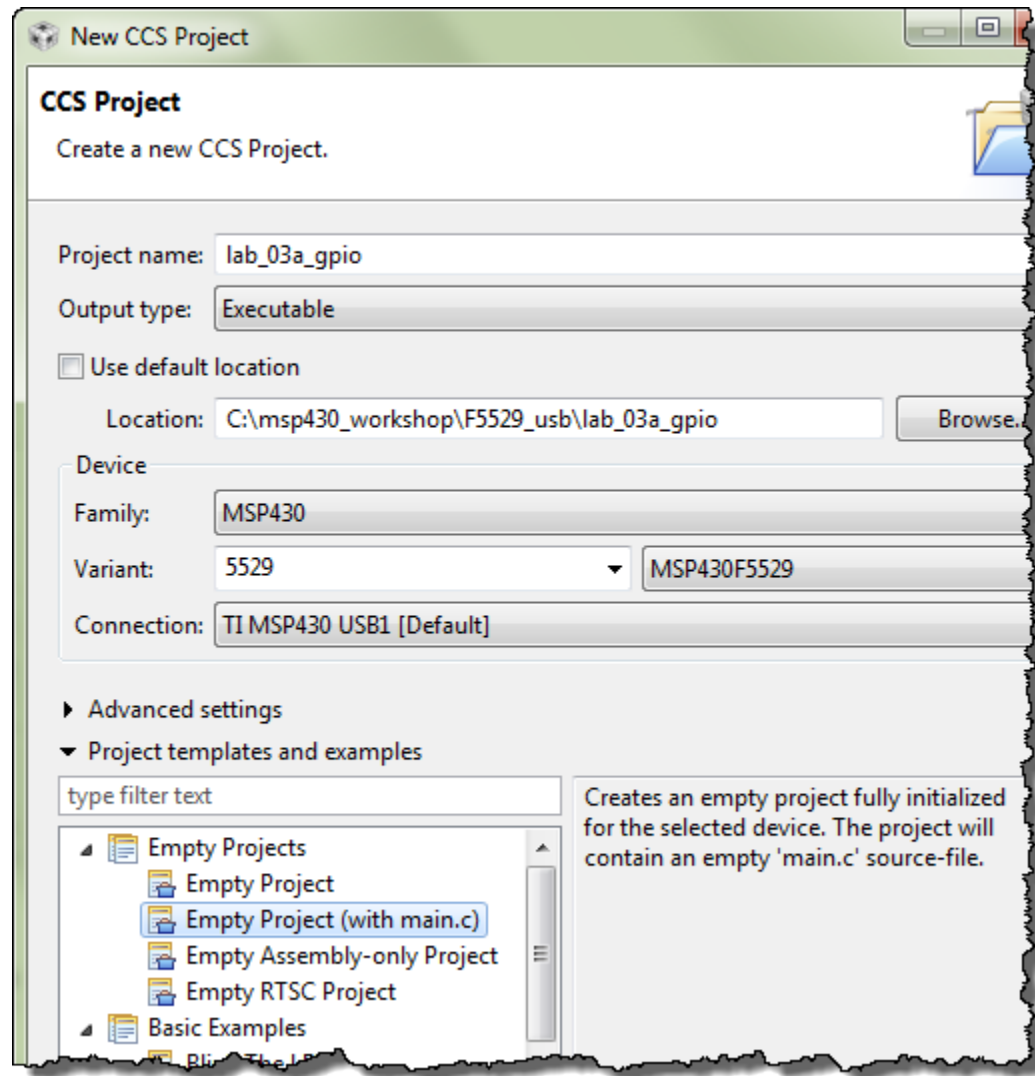
This helps to prevent us from accidentally working on the wrong file, which is easy to do when we have multiple lab exercises that use "main.c". If a previous project is open:

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.



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

You can replace this “register-based” code with the driverlib function, although this is not required. Either way works fine. *If you want to use driverlib, please reference your Worksheet answer #3 (on page 3-15).*

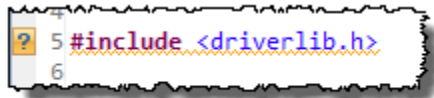
5. Add required header files.

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?



```
5 #include <driverlib.h>
```

The image shows a snippet of code from a text editor. Line 5 contains the text `#include <driverlib.h>`. A yellow question mark icon is positioned to the left of the code on this line. Line 6 is empty. The code is highlighted with a light blue background.

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.

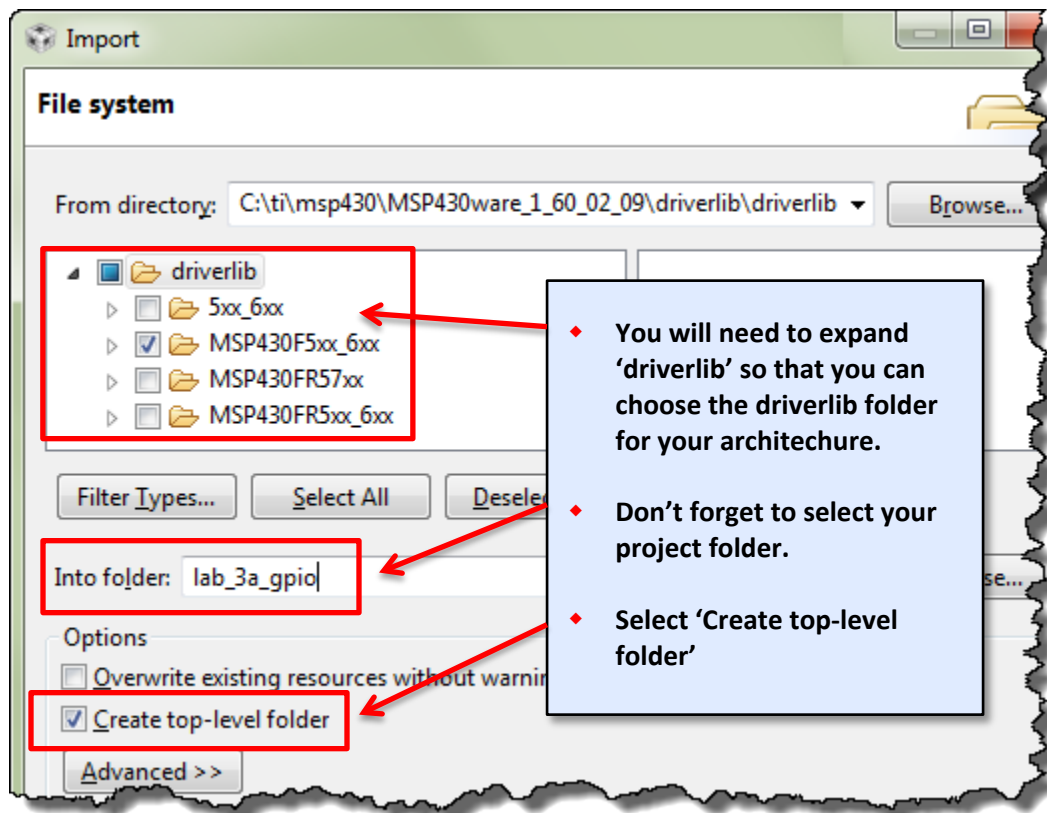
Adding the DriverLib library is a two-step process:

- 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 – your path may be different than what is shown below. (See Worksheet question #1.)



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 dir to #include search path:

Right-click project → Properties

Then select:

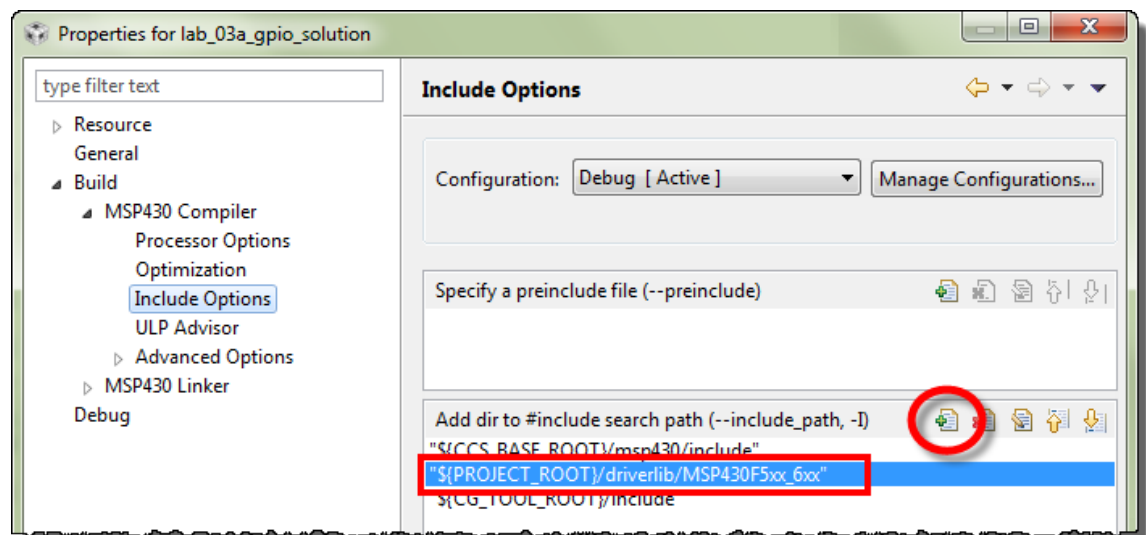
Build → MSP430 Compiler → Include Options

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 our edits, thusfar, are all correct.

Add the Code to `main.c`

10. Setup P1.0 as output pin.

Reference Worksheet question #4 (page 3-15).

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



```
*main.c X
1
2 #include <driverlib.h>
3
4 /*
5  * main.c
6  */
7 int main(void) {
8     WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
9
10    return 0;
11 }
12
```

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

Reference Worksheet question #5 (page 3-15). Begin the `while()` loop after the code you wrote in the previous step. Also, don't forget to add the `#define` for "ONE_SECOND".



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.

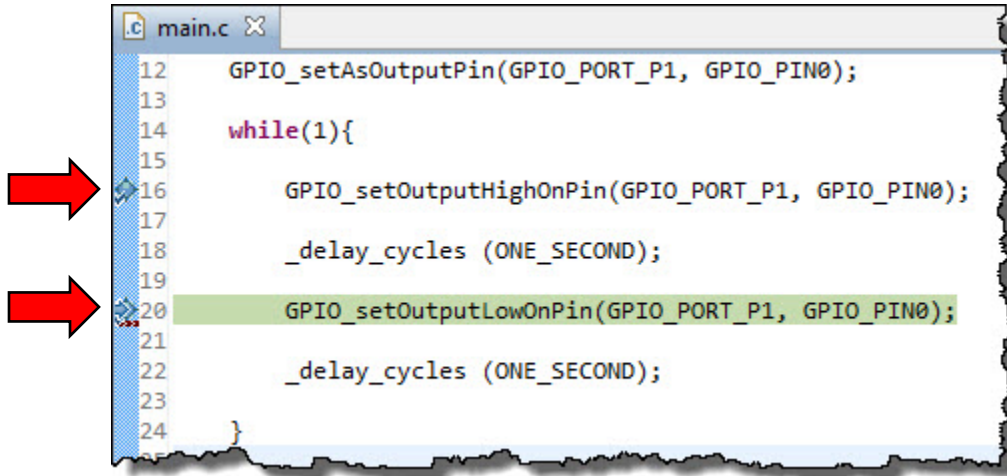
You should see the P1OUT register change as you step over the appropriate function.

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 X
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: Oh, and we ended up finding the problem with our code. A cut and paste error left us with two lines of code in our loop that both turned off the LED. Oops!

Lab 3b – Reading a Push Button

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

Help → Welcome to CCS

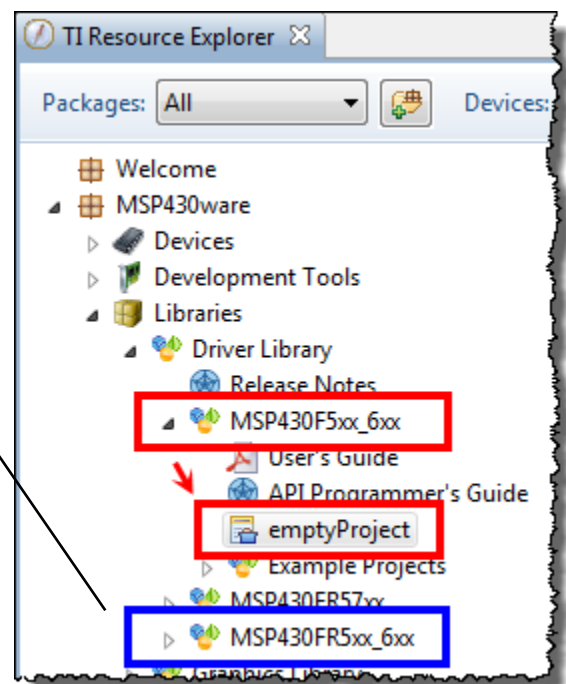
Hint: If you don't see a listing of resource in the window, click the *Home* button.



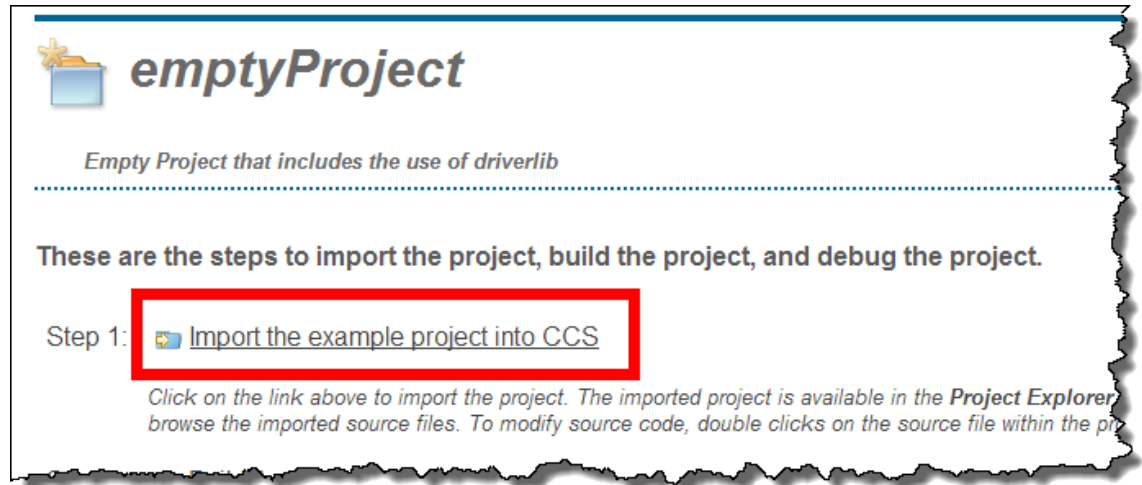
b) Locate the *emptyProject* example.

Look for it as shown here:

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_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 source project 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 CCSv5 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 just 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; it's value depends upon the push button'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 'Button2' later on. (We could have also used the names 'SW1' and 'SW2' as they're labeled on the Launchpad, but we liked 'Button' better.)

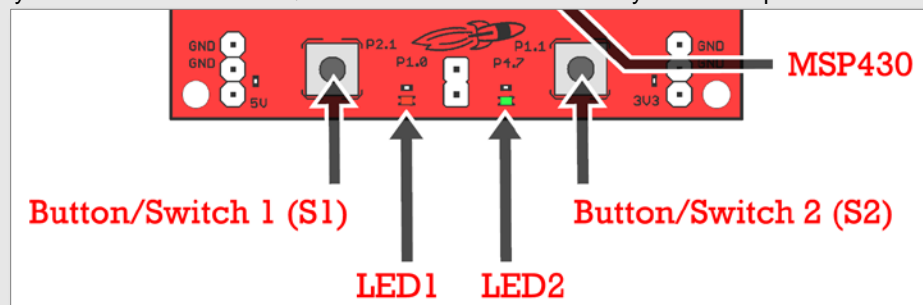
=0 ... well, that's just good style. You should always initialize your variables. Many embedded processor compilers do not automatically initialize variables for you.

8. In `main()`, add code to setup push button (SW1) as an input with pullup resistor.

This setup code should go before the `while{}` loop.

And don't forget, this code was the answer to Worksheet question #8 (page 3-16). Reminder – SW1 is connected to Port 2, 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 SW1 – and the pin it is connected to. It may easier to reference the Quick Start sheet that came with your Launchpad.



Modify Loop

9. Modify the while loop to light LED when SW1 push button is pressed.

Comment out (or delete) LED blinking code and replace it with the code we created in the Worksheet question #9 (page 3-16).

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

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

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

//***** Defines *****
#define ONE_SECOND 800000
#define HALF_SECOND 400000

//***** 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, P2.1 as input with pullup resistor
    GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );
    GPIO_setAsInputPinWithPullUpresistor( GPIO_PORT_P2, GPIO_PIN1 );

    while(1) {
        // Read P2.1 pin connected to push button 1
        usiButton1 = GPIO_getInputPinValue ( GPIO_PORT_P2,
                                             GPIO_PIN1 );

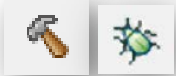
        if ( usiButton1 == GPIO_INPUT_PIN_LOW ) {
            // If button is down, turn on LED
            GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        }
        else {
            // If button is up, turn off LED
            GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        }
    }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

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 CCSv5 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 glory 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 pin as an interrupt ... which we'll do in Chapter 5.

Optional Exercises

- Try this lab without pullup (or pulldown) 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

Lab3 – Worksheet

1. Where is your MSP430ware folder located?

Most likely: `C:\ti\msp430\MSP430ware_1_60_01_11\`

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 ) ;
```

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"?

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

Lab3 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 ) ;  
}
```

Lab3 Worksheet

6. 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_60_01_11\driverlib\doc\MSP430F5xx_6xx\

GPIO_setAsInputPin()

GPIO_setAsInputPinWithPullDownresistor()

GPIO_setAsInputPinWithPullUpresistor()

7. 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.

8. Assuming you need a pull-up resistor for a GPIO input, write line of code required to setup pin P1.0 for input:

GPIO_setAsInputPinWithPullUpresistor (GPIO_PORT_P1, GPIO_PIN0) ;

Lab3 Worksheet

9. Complete the following code to read pin P2.1:

```
volatile unsigned short usiButton1 = 0;
while(1) {
    // Read push-button pin (SW1)
    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_P1, GPIO_PIN0 );
    }
    else {
        // Otherwise, if button is up, turn off LED
        GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
    }
}
```

10. 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

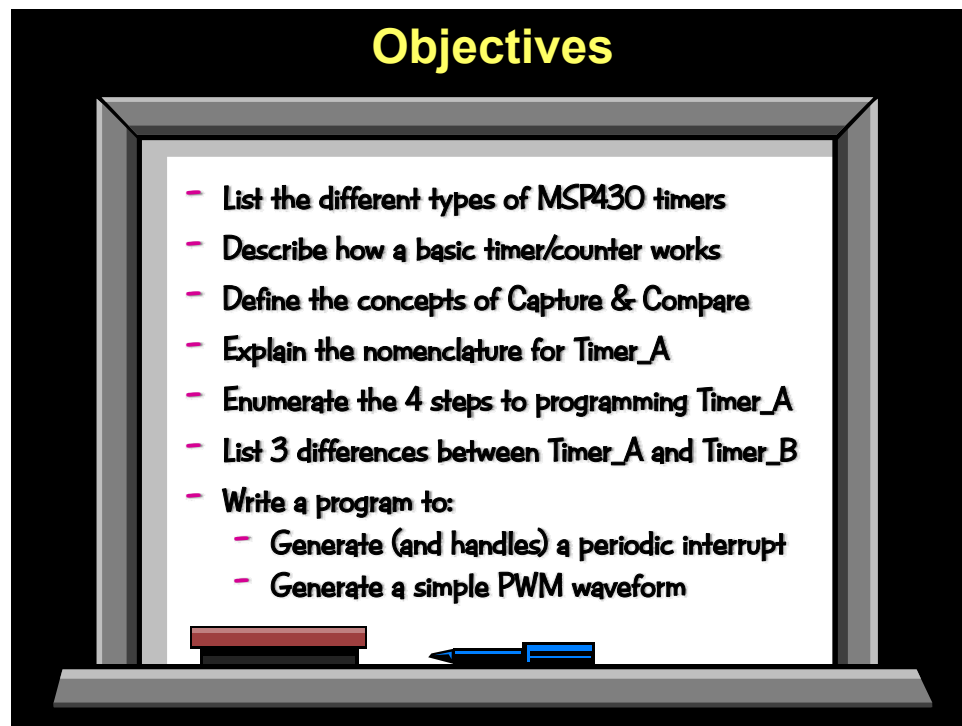
Introduction

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 use through 3 internal clock signals, which power the CPU along with 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>	<i>4-3</i>
BOR → POR → PUC → Active (AM)	4-3
<i>Clocking.....</i>	<i>4-5</i>
What Do You Need?	4-5
MCLK, SMCLK, ACLK	4-6
Clock Sources	4-6
Clock Details (by Device Family)	4-8
<i>DCO Setup and Calibration</i>	<i>4-14</i>
How the DCO Works	4-14
Factory Calibration (G2xx, FR5xx)	4-17
Runtime Calibration.....	4-18
VLO 'Calibration'	4-19
<i>Other Initialization (WDT, PMM)</i>	<i>4-20</i>
Watchdog	4-20
PMM with LDO, SVM, SVS, and BOR.....	4-21
Initialization Summary (template).....	4-24
<i>Lab Exercise</i>	<i>4-25</i>

Operating Modes (Reset → Active)

BOR → POR → PUC → Active (AM)

Brownout Reset (BOR)

At power-up, the brownout circuitry holds device in reset until V_{cc} is above hysteresis point

Startup from BOR:

- ◆ RST/NMI pin is configured as reset
- ◆ I/O pins are configured as inputs
- ◆ Clocks are configured
- ◆ Peripherals and CPU registers are initialized (see user guide)
- ◆ Status register (SR) is reset
- ◆ Watchdog timer powers up active in watchdog mode
- ◆ Program counter (PC) is loaded with reset vector location (0xFFFE)
If reset vector is blank (0xFFFFh), the device enters LPM4

```

    graph TD
      BF(Brownout fault) --> BOR(BOR)
      BOR --> AM(Active Mode)
    
```

BOR → POR → PUC → Active (AM)

Three Levels of Reset

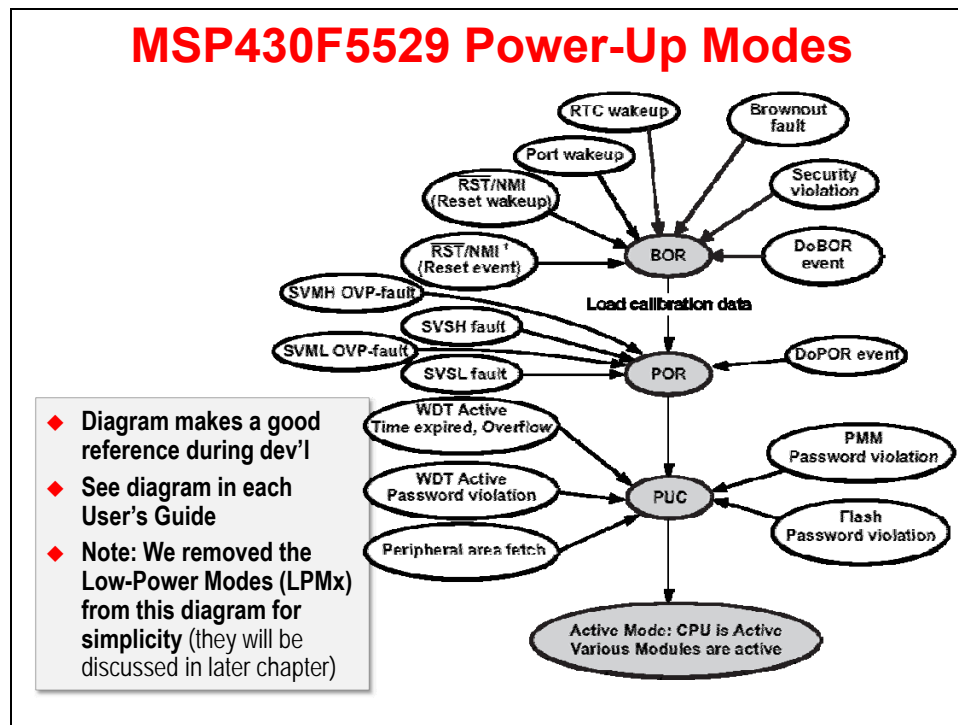
- ◆ BOR is most comprehensive, followed by:
 - ◆ POR = Power-On Reset
 - ◆ PUC = Power-Up Clear
- ◆ Different events trigger different resets; e.g.
 - ◆ SVS (power supervisor) triggers POR
 - ◆ WDT (watchdog) triggers PUC
- ◆ Each level touches different bits in CPU and peripheral registers → **User Guide** notation:

Register Bit Accessibility and Initial Condition

Key	Bit Accessibility
rw	Read/write
r	Read only
w	Write only
nu	Clear on hardware reset
-0,-1	Condition after PUC
-(0),-(1)	Condition after POR
-{0},-{1}	Condition after BOR
-{0},-{1}	Condition after Brownout

```

    graph TD
      BF(Brownout fault) --> BOR(BOR)
      RSTNMI(RST/NMI (Reset event)) --> BOR
      BOR --> POR(POR)
      SVS(SVS fault) --> POR
      POR --> PUC(PUC)
      WDT(WDT Active Time expired, Overflow) --> PUC
      PUC --> AM(Active Mode: CPU is Active Various Modules are active)
    
```



Clocking

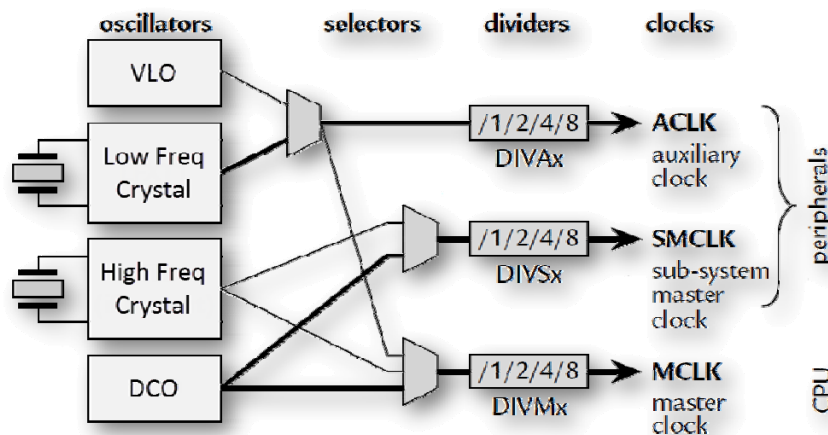
What Do You Need?

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 – Lot's of Options



- ◆ Variety of **osc sources** – on-chip (cheap, reliable) and off-chip (accurate)
- ◆ Rich **selection** of oscillator sources routed to internal clocks
- ◆ Many clock **dividers** enhance the available clock frequencies
- ◆ All MSP430 devices provide at least 3 **internal clocks** – provides flexibility in tuning system's power vs performance

MCLK, SMCLK, ACLK

MSP430 Clock Options

Name	Description	Used-by	Typical Speed
<input type="checkbox"/> MCLK	Master Clock	CPU	Fast
<input type="checkbox"/> SMCLK	Sub-Master Clock	Peripherals	Fast
<input type="checkbox"/> ACLK	Auxiliary Clock	Peripherals	Slow

Clocks – Fast or Slow

- ◆ All MSP430 devices provide at least 3 clocks
- ◆ Tune system peripherals by choice of clock:
 - ◆ Fast = Performance
 - ◆ Slow = Low-power
- ◆ Fast/slow clocks also provide wider timing



Clock Sources

Typical Clock Sources

	Frequency	
VLO	~10 KHz	
REFO	32768 Hz	
XT1	• LF: < 50 KHz • HF: 4-Max MHz	
XT2	4-40 MHz	
DCO	100 KHz to CPU Max	
MODOSC	• 5 MHz • 5 MHz / 128	



*Note: This is a general description, please refer to datasheet/UsersGuide for complete details regarding your device

Typical Clock Sources

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

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

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

Clock Details (by Device Family)

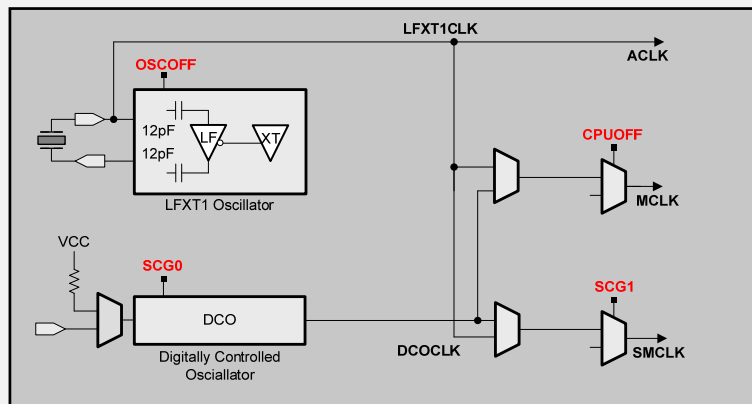
MSP430 Clock Modules

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

F1xx Basic Clock System (BCS)

Reserved	V	SCG1	SCG0	OSC OFF	CPU OFF	GIE	N	Z	C
----------	---	------	------	---------	---------	-----	---	---	---

R2/SR



F2xx/G2xx Basic Clock System (BCS+)

◆ **Very Low Power/Low Frequency Oscillator (VLO)***

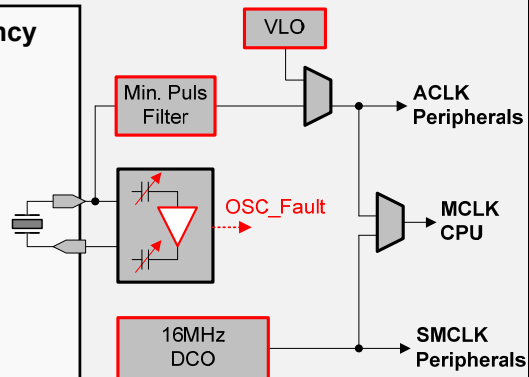
- ◆ 4 – 20kHz (typical 12kHz)
- ◆ 500nA standby
- ◆ 0.5%/°C and 4%/Volt drift
- ◆ Not in '21x1 devices

◆ **Crystal oscillator (LFXT1)**

- ◆ Programmable capacitors
- ◆ Failsafe OSC_Fault
- ◆ Minimum pulse filter

◆ **Digitally Controlled Oscillator (DCO)**

- ◆ 0-to-16MHz
- ◆ ± 3% tolerance
- ◆ Factory calibration in Flash



On PUC, MCLK and SMCLK are sourced from DCOCLK at ~1.1 MHz. ACLK is sourced from LFXT1CLK in LF mode with an internal load capacitance of 6pF. If LFXT1 fails, ACLK defaults to VLO.

* Not on all devices. Check the datasheet.

F5xx: Unified Clock System (UCS)

◆ UCS is available on F5xx/F6xx devices

◆ Six independent clock sources

- ◆ Low Frequency
 - ◆ LF XT1 32768 Hz crystal
 - ◆ VLO 10 kHz
 - ◆ REFO 32 kHz
- ◆ High Frequency
 - ◆ HF XT1 4 – 32 MHz crystal
 - ◆ XT2 4 – 32 MHz crystal
 - ◆ DCO FLL calibration

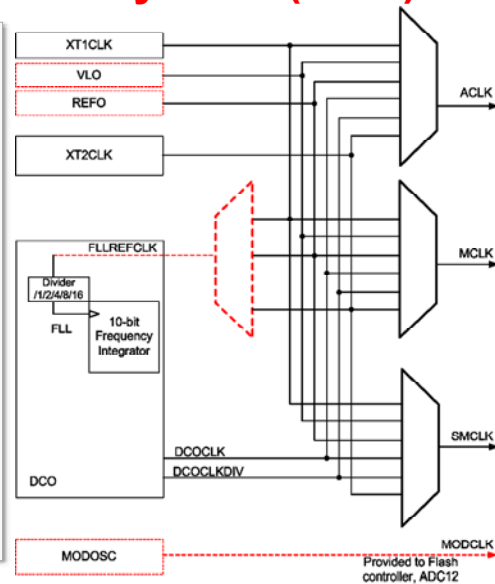
◆ FLL references (divisible, too)

- ◆ LFXT1 / XT1
- ◆ REFO
- ◆ XT2

◆ **Orthogonal:** Any source to any clock

◆ MODOSC provided for Flash & ADC12

◆ Clocks on demand



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

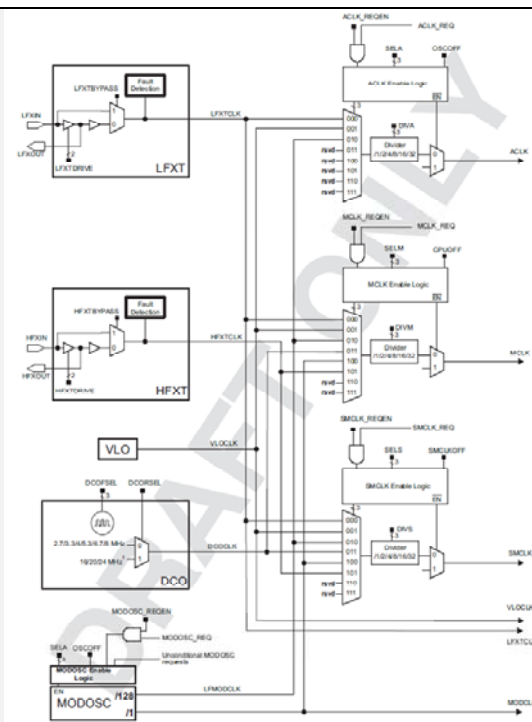
Main Features:

- ◆ 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

Wolverine's Clock System (CS)

Clock System (CS)

- ◆ CS found on Wolverine (FR58/59xx)
- ◆ Five independent clock sources
 - ◆ Low Freq
 - ◆ LFXT1 (32768 Hz crystal)
 - ◆ VLO (10 kHz)
 - ◆ LFMODCLK (MODCLK/128)
 - ◆ High Freq
 - ◆ XT1 (4 – 24 MHz crystal)
 - ◆ XT2 (4 – 24 MHz crystal)
 - ◆ DCO (Specific CAL range)
 - ◆ MODCLK (Internal 5MHz)
- ◆ Notes:
 - ◆ MODOSC provided to ADC12, MODCLK and LFMODCLK
 - ◆ LF and HF ranges for XT1
- ◆ Defaults:
 - ◆ DCO = 1MHz
 - ◆ ACLK = Only LF sources
- ◆ Failsafes:
 - ◆ XT1LF: LFMODCLK (~42kHz)
 - ◆ XT1HF or XT2: MODCLK (5MHz)



Using MSP430ware to configure clocking

DriverLib – Selecting Clock Sources

```
#include <driverlib.h>

void myClkInit(void) {
    //Set ACLK = REFO
    UCS_clockSignalInit (
        UCS_BASE,
        UCS_ACLK,           // Configure ACLK
        UCS_REFOCLK_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)

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 );

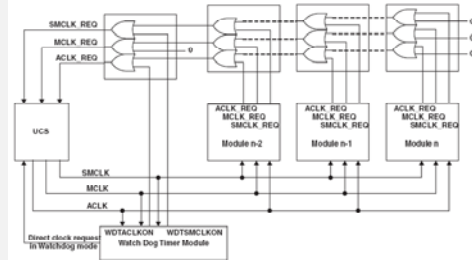
//Start the XT1 oscillator, wait until it's running
UCS_LFXT1Start( UCS_BASE, UCS_XT1_DRIVE0, UCS_XCAP_3 );

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 it's return value

Clock Requests (don't turn off clocks, if needed)

- ◆ Modules place clock requests to the system clocks
- ◆ LPM3 entry can be prevented if a module requires SMCLK to operate properly!
- ◆ Must be very conscious of the clocks required in the system.



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.

Additional Clock Features				
Clock Feature		G2553 (BCS+)	F5529 (UCS)	F5969 (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 (1MHz)	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 (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)		WDT+ only	Yes	Yes

DCO Setup and Calibration

Calibrating DCO

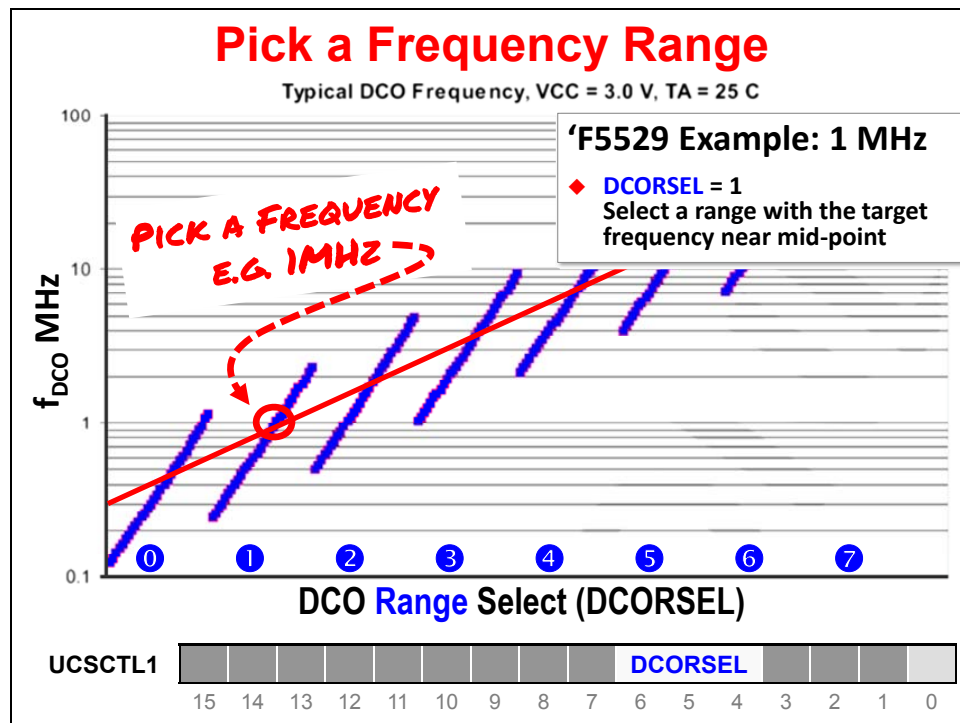
Additional Clock Features

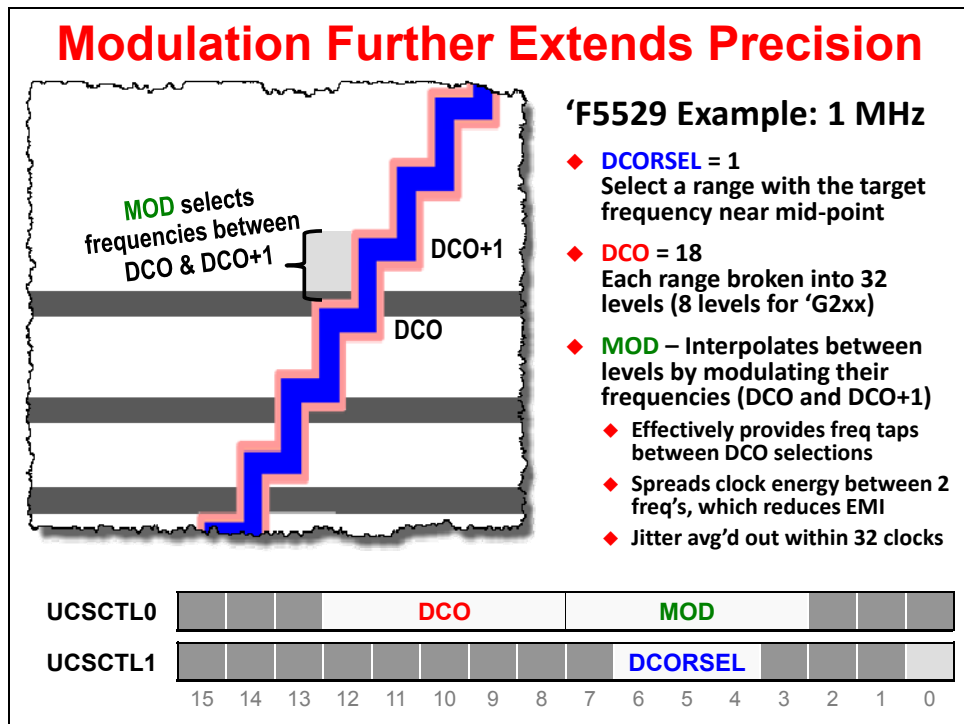
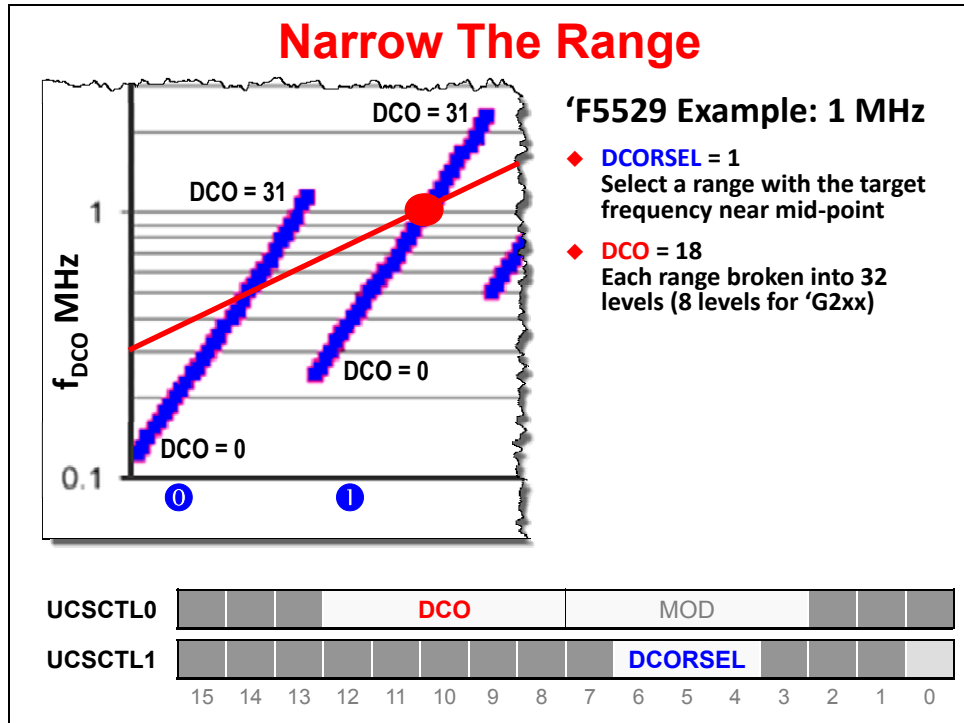
Clock Feature	G2553 (BCS+)	F5529 (UCS)	F5969 (CS)
Available Clocks	MCLK VLO, LFXT1, XT1CLK	MCLK VLO, LFXT1, XT1CLK	FXT, LFMODCLK, MODCLK, DCOCLK VLO, LFXT, LFMODCLK
Clock Defaults (at PUC reset)	MCLK ACLK	DCO (1.1MHz) XT1CLK	DCO (1MHz) LFXT
External Clk Failsafe	ACLK = VLO S/MCLK = DCO	LFXT1 = REFO HFXT1/XT2 = 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

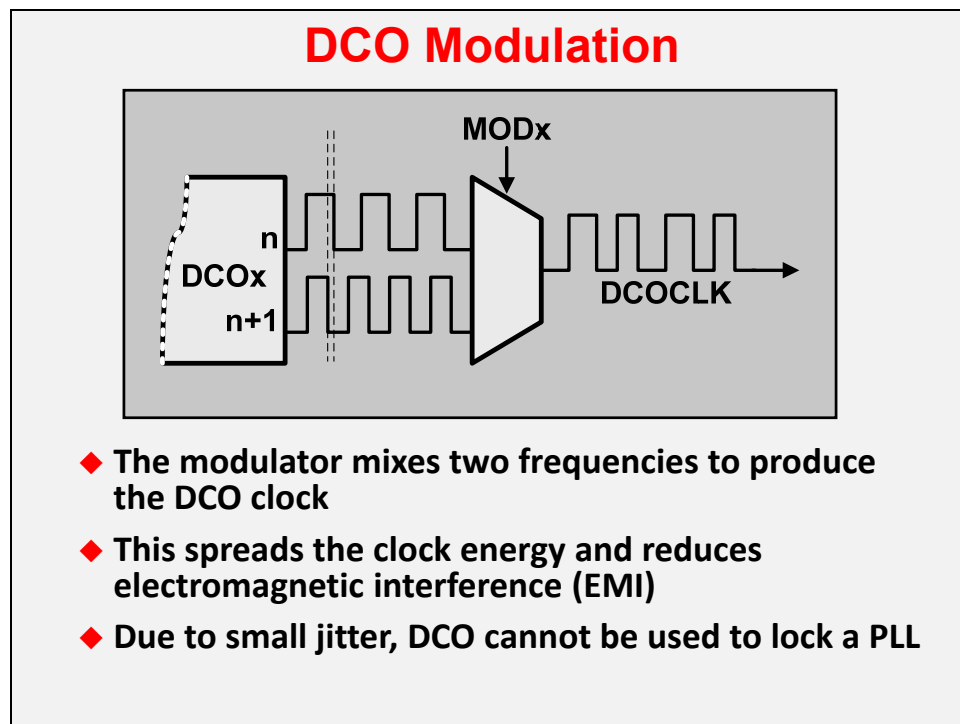
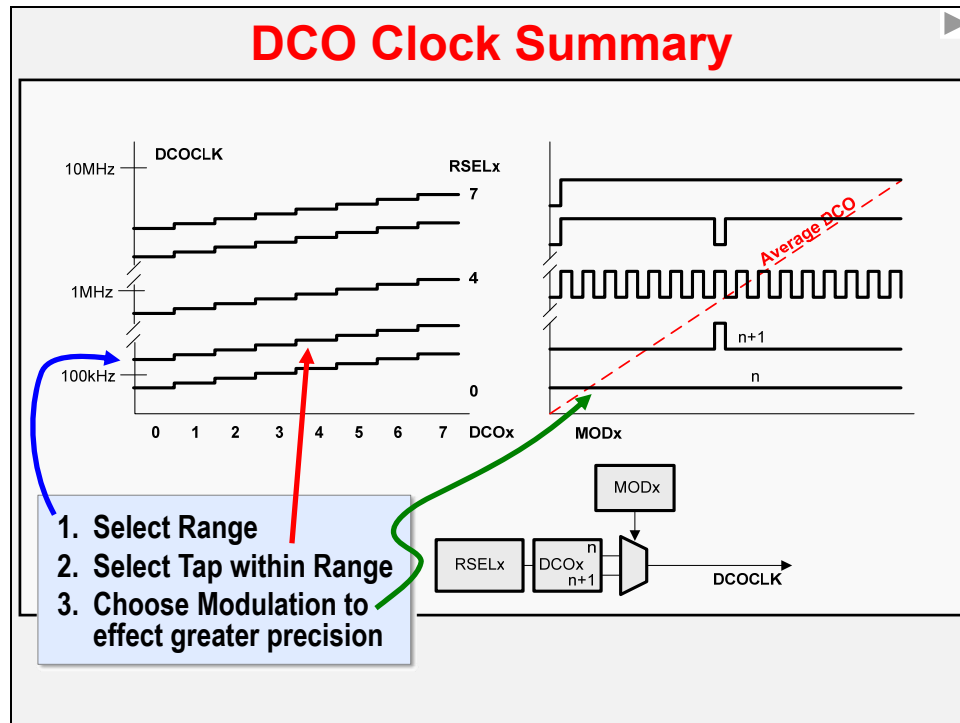
EARLIER, WE DESCRIBED HOW THE DCO IS CALIBRATED

Before we look at the details of calibration, let's start with "How does the DCO work?"

How the DCO Works







Factory Calibration (G2xx, FR5xx)

FR5xx DCO – Calibrated Frequencies

- ◆ Clock System (CS) module found on FR5xx devices
- ◆ DCO (CS module) provides multiple pre-defined & calibrated frequencies
- ◆ Factory Trimmed Accuracy:
±2% from 0-50C
±3.5% from -40 to 85C
- ◆ FR5xx CS module requires psw to write clock reg's
- ◆ *If DCOCLK = 20 or 24MHz it must be divided down for MCLK

DCORSEL	DCOFSEL	DCO (MHz)
0 or 1	000	1
0	001	2.667
0	010	3.333
0	011	4
0/1	100/001	5.33
0/1	101/010	6.67
Ex: 0/1	110/011	8
1	100	16
1	101	20*
1	110	24*



```
// Set DCO to 8MHz
```

```
CS_setDCOFreq(CS_BASE, CS_DCORSEL_1, CS_DCOFSEL_3);
```

'G2xxx DCO – Calibration Constants

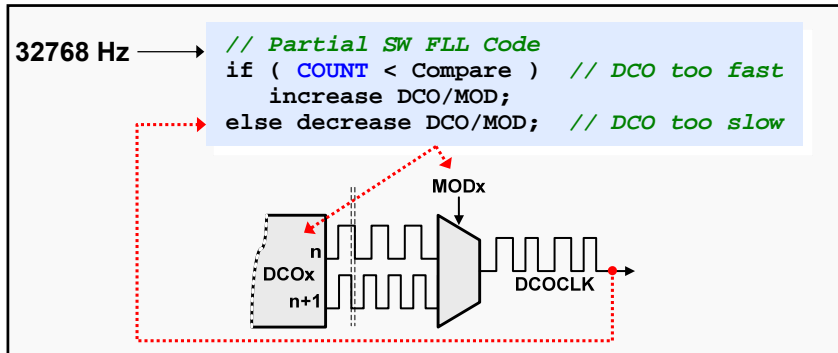
DCO Calibration Data (provided from factory in flash info memory segment A)			
DCO Frequency	Calibration Register	Size	Address
1 MHz	CALBC1_1MHz	byte	010FFh
	CALDCO_1MHz	byte	010FEh
8 MHz	CALBC1_8MHz	byte	010FDh
	CALDCO_8MHz	byte	010FCh
12 MHz	CALBC1_12MHz	byte	010FBh
	CALDCO_12MHz	byte	010FAh
16 MHz	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
```

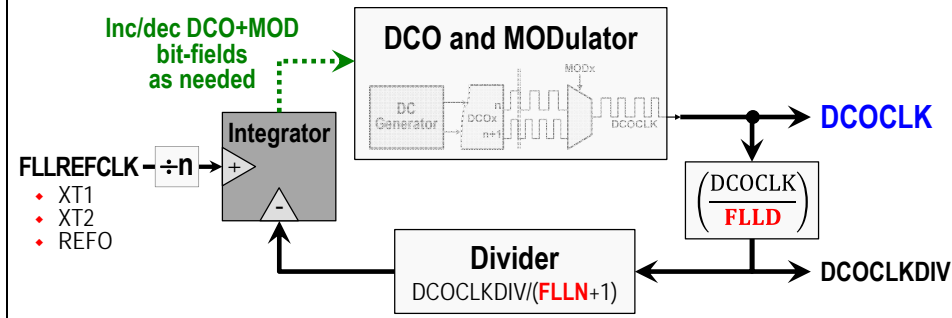
Runtime Calibration

Dynamic Calibration of DCO in Software



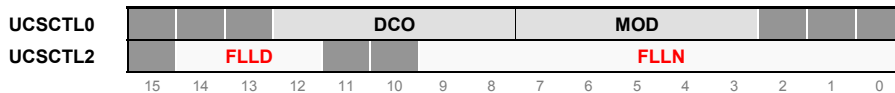
- ◆ **Minimize frequency drift** due to changes in voltage or temperature
 - DCO clock precision is achieved by periodic adjustment in loop
 - Modify settings (DCO, MOD) in loop based upon comparison of DCO to another known/stable freq, such as 32kHz crystal (or 50/60Hz AC power)
- ◆ **Frequency Locked 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

‘F5xx Hardware FLL



$$DCOCLK = (FLLREFCLK/n) * FLLD * (FLLN + 1)$$

where: n = FLLREFDIV



Setting 'F5529 DCO with MSP430ware

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

#define MCLK_FREQ_KHZ      8000
#define FLLREF_KHZ        32
#define MCLK_FLLREF_RATIO MCLK_FREQ_KHZ/FLLREF_KHZ // Ratio=250

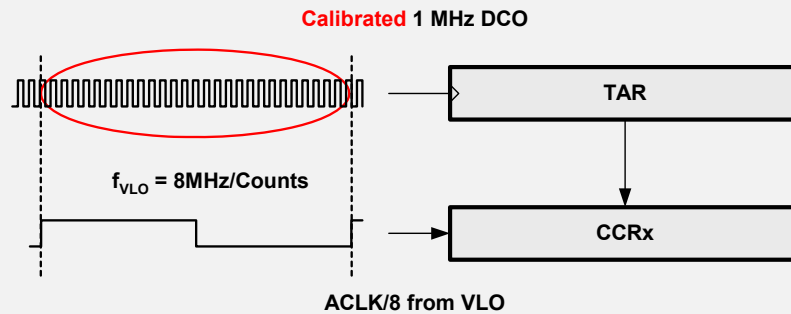
void myInitDCO (void) {
    // Set DCO FLLREF to 32KHz = REF0
    UCS_clockSignalInit ( UCS_BASE,
                          UCS_FLLREF,           // Setup FLLREFCLK
                          UCS_REFOCLK_SELECT,    // FLLREFCLK=REFO
                          UCS_CLOCK_DIVIDER_1    // FLLREFDIV=1
                        );

    // Setup DCO and FLL to provided freq (sets FLLD, FLLN, etc.)
    // once clk settled, use as source for MCLK & SMCLK
    UCS_initFLLSettle( UCS_BASE,
                       MCLK_FREQ_KHZ,
                       MCLK_FLLREF_RATIO);
}
```



VLO 'Calibration'

Run Time 'Calibration' of VLO

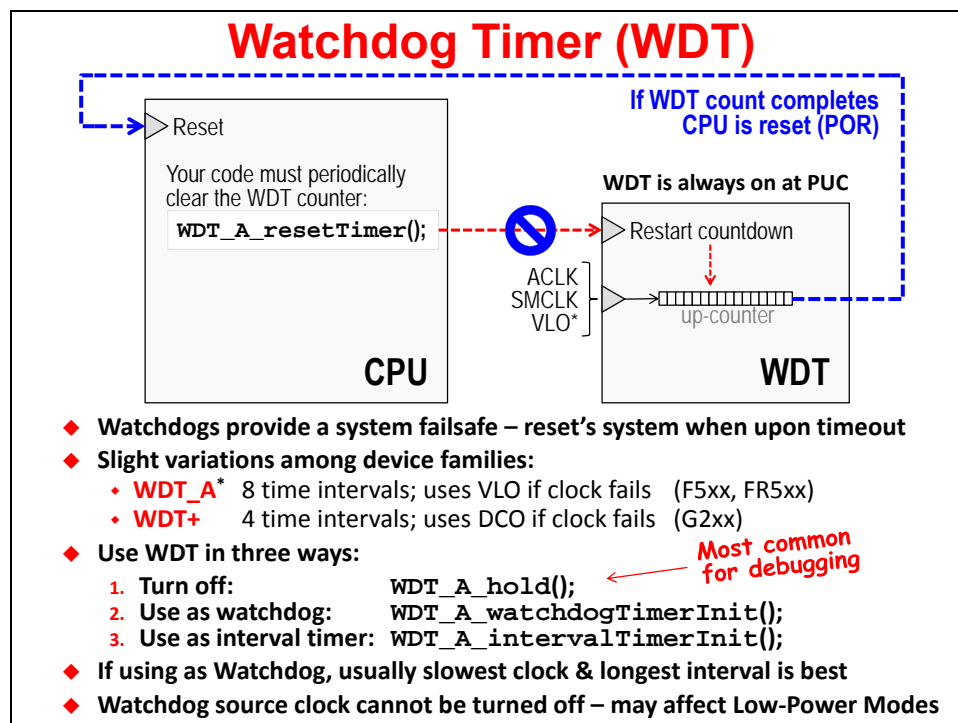


- ◆ Calibrate the VLO during runtime
- ◆ Example:
 - ◆ Timer_A clocked at calibrated 1MHz (from DCO)
 - ◆ Capture with rising edge of ACLK/8 from VLO
 - ◆ $f_{VLO} = 8\text{MHz}/\text{Counts}$
- ◆ Code library on the web (search for "SLAA340")

Other Initialization (WDT, PMM)

Software Initialization			
Initialization Step	Required Action?	Who is Responsible	Where Discussed
1. Initialize the stack pointer (SP)	Yes	Yes / Compiler	N/A
2. Initialize <u>watchdog timer</u> (usually OFF when debugging)	Yes	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

Watchdog

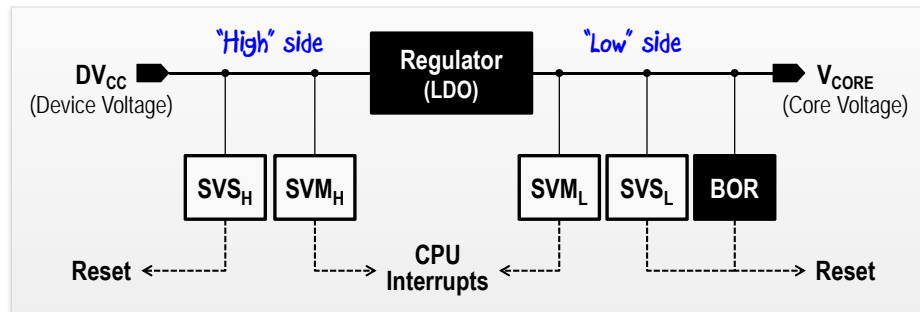


PMM with LDO, SVM, SVS, and BOR

Power Management Module (PMM)

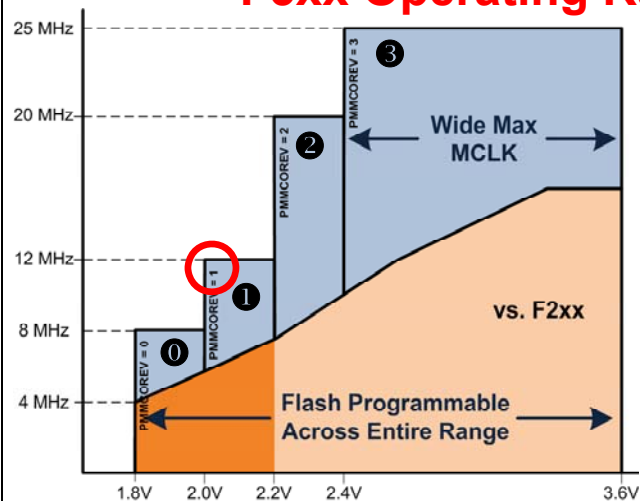
The on-chip PMM manages all functions related to the power supply and its supervision for the device. Its primary functions are:

1. Generate a **supply voltage for the core logic (LDO)**
2. Provide several mechanisms for the **supervision and monitoring (SVS/SVM)**



SVM	Supply Voltage Monitor	Warn if voltage is getting low	Optional
SVS	Supply Voltage Supervisor	Reset if voltage is too low	Optional
BOR	Brown-Out Reset	Reset if core voltage too low	Always On

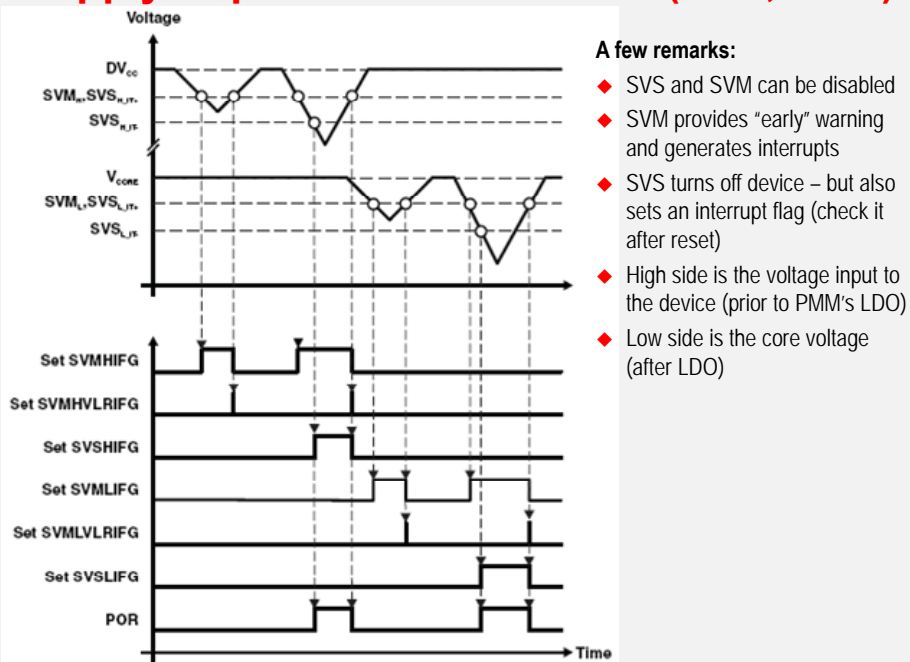
'F5xx Operating Range



- ◆ 25MHz peak performance
- ◆ More performance across V_{CC} range vs 'F/G2xx:
 - ◆ Flash ISP @ min. V_{CC}
 - ◆ 8MHz @ min. V_{CC}
 - ◆ Up to 25MHz @ 2.4V-3.6V
- ◆ Programmable V_{CORE} maximizes power efficiency; power vs performance
- ◆ V_{CORE} register bits: $PMMCTL0.PMMCOREV$
- ◆ When using SVS, changing V_{CORE} is a 4 step process, but it's easy with Driverlib: `PMM_setVCore();`

```
#include <driverlib.h>
//Set VCore = 1 for 12MHz clock
PMM_setVCore( PMM_BASE, PMM_CORE_LEVEL_1 );
```

Supply Supervisor and Monitor (SVS, SVM)

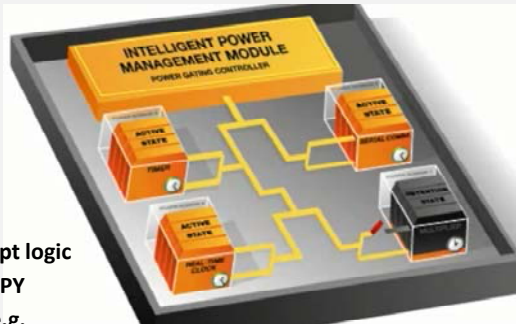


Power Management Summary

	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	No	Yes	Yes

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

Voltage Supervision & Monitoring

SVS / SVM disabled

- SVS / SVM disabled
- Zero-power BOR protection is ALWAYS ON
- 5 us wakeup from LPM2,3,4
- +0 uA active & LPM2,3,4 current consumption

High-side Full Performance Mode

- High-side Full Performance Mode
- Low-side SVS / SVM disabled
- +4uA active current consumption
- +0uA LPM2,3,4 current consumption
- Automatic high-side protection when CPU is active

5 us wakeup from LPMx

Maximum Robustness

- Fast Performance Mode
- 5 us wakeup from LPM2,3,4
- +8 uA active & LPMx current consumption

Power on Default Mode

- ◆ Normal Performance Mode
- ◆ +800 nA active current consumption
- ◆ 0 nA LPM2,3,4 current consumption

150 us wakeup from LPMx

High-side Fast Performance Mode

- High-side Fast Performance Mode
- Low-side SVS / SVM disabled
- 5 us wakeup from LPM2,3,4
- +4 uA active & LPM2,3,4 current consumption
- Automatic high-side protection when CPU is active

Current

Initialization Summary (template)

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) {
        ...
    }
}
```

Lab 4 - Abstract

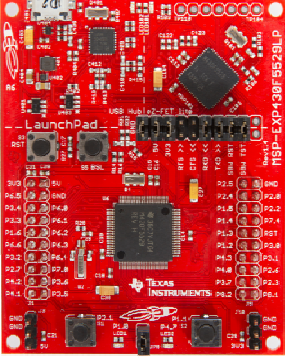
Lab 4 explores a variety of initialization tasks; the largest one being to setup the clocks for the MSP430.

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 Labs:

- ◆ **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	4-24
<i>Lab 4 - Abstract</i>	4-25
<i>Lab 4 Worksheet</i>	4-27
<i>Lab 4a – Program the MSP430 Clocks</i>	4-31
File Management	4-31
Do Clock Code	4-31
Initialization Code - Three more simple changes.....	4-34
Debugging the Clocks	4-35
Extra Credit (i.e. Optional Step) – Change the Rate of Blinking.....	4-37
<i>(Optional) Lab 4b – Exploring the Watchdog Timer</i>	4-38
First, a couple of Questions	4-38
Play with last lab exercise	4-38
File Management	4-39
Edit the Source File.....	4-39
Keep it Running.....	4-41
Extra Credit – Try DriverLib’s Watchdog Example (#3).....	4-41
<i>(Optional) Lab 4c – Using Crystal Oscillators</i>	4-42
<i>Chapter 04 Appendix</i>	4-44

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_60_02_09\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_60_02_09\driverlib\driverlib\MSP430F5xx_6xx\ucs.h

- ◆ Finally, we recommend you also reference the driverlib UCS example #4:

\msp430\MSP430ware_1_60_02_09\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

4. ('F5529 Launchpad users only)

Write the driverlib function that sets the core voltage needed to run MCLK at 8MHz.

_____ (PMM_BASE, _____);

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 chapter".

```
_____ ( UCS_BASE,
          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;
```

*Refer to clocking section
of DriverLib User's Guide*

```
myACLK = _____ ( ______BASE );
mySMCLK = _____ ( ______BASE );
myMCLK = _____ ( ______BASE );
```

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
_____ ( _____ _BASE,
          _____ _ACLK,           // Clock to setup
          _____, // Source clock
          _____ _CLOCK_DIVIDER_1
        );

```

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_BASE,
          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
_____ ( UCS_BASE,
          MCLK_DESIRED_FREQUENCY_IN_KHZ,
          _____ );

```

Hint: There's a discussion slide very similar to this question

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_BASE,
               _____, // Set Frequency range (DCOR)
               _____ // Set Frequency (DCOF)
);

// Set MCLK to use DCO clock source
_____ ( CS_BASE,
        _____,
        _____,
        UCS_CLOCK_DIVIDER_1 );
```

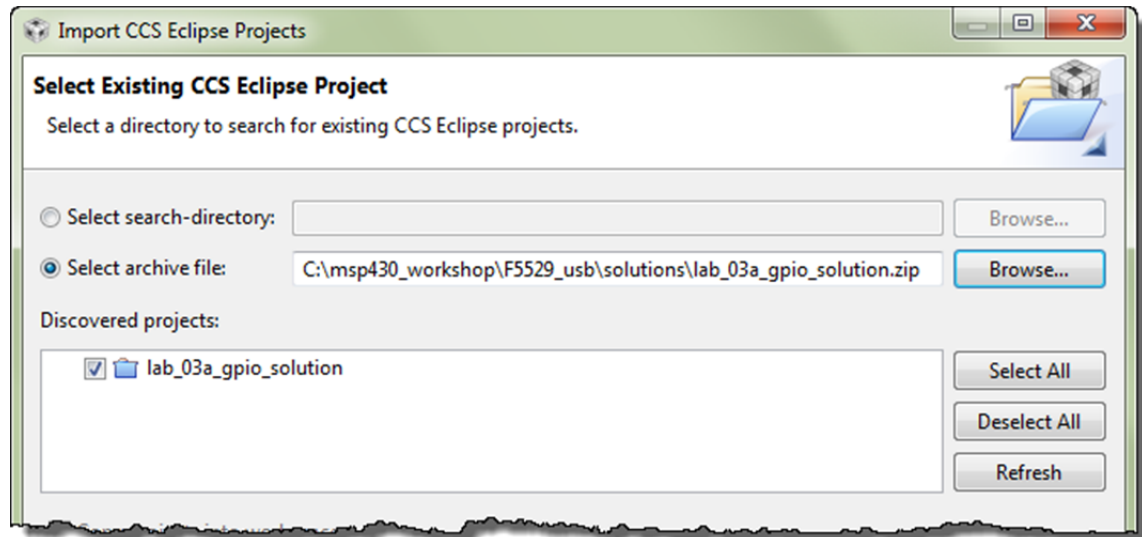
Check your answers against ours ... see the Chapter 4 Appendix

Lab 4a – Program the MSP430 Clocks

File Management

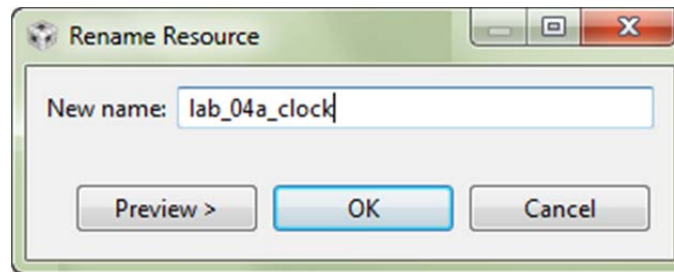
1. Import previous lab_03a_gpio solution.

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...

5. 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 #11/12

```

/***** 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( UCS_BASE );
    mySMCLK = UCS_getSMCLK( UCS_BASE );
    myMCLK = UCS_getMCLK( UCS_BASE );

    // Setup ACLK to use REFO as its oscillator source
    UCS_clockSignalInit( UCS_BASE,
        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_BASE,
        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( UCS_BASE,
        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( UCS_BASE );
    mySMCLK = UCS_getSMCLK( UCS_BASE );
    myMCLK = UCS_getMCLK( UCS_BASE );
}

```



6. Try building to see if there are any errors.

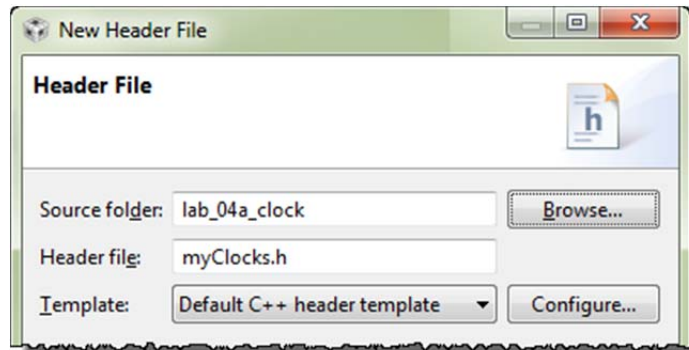
Hopefully you don't have any typographic or syntax errors, but you should see this error:

```
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.

7. Create a new source file called `myclocks.h`.

File → New → Header File

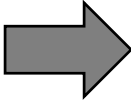


Then click 'Finish'.

8. 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_ */
```

9. 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.



10. Try building again. Keep fixing errors until they're all gone.

Initialization Code - Three more simple changes

11. 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.

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

```
// -----
// 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, _____ );
}
```




12. Build the code and fix any errors. When no errors exist, launch the debugger.



Debugging the Clocks

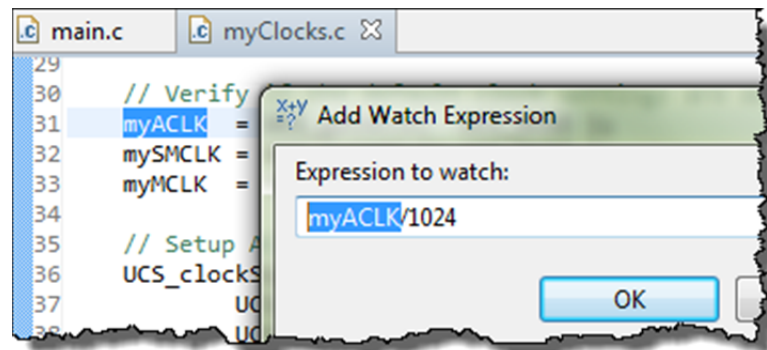
Before running the code, let's set some breakpoints and watch expressions.

13. Open `myClocks.c` in the debugger.

14. Add a watch expression for `myACLK` (in KHz).

Select `myACLK` in your code → Rt-click → Add Watch Expression...

Enter `'myACLK/1024'` into the dialog and hit OK. Upon hitting "OK", the *Expressions* window should open up, if it's not already open.



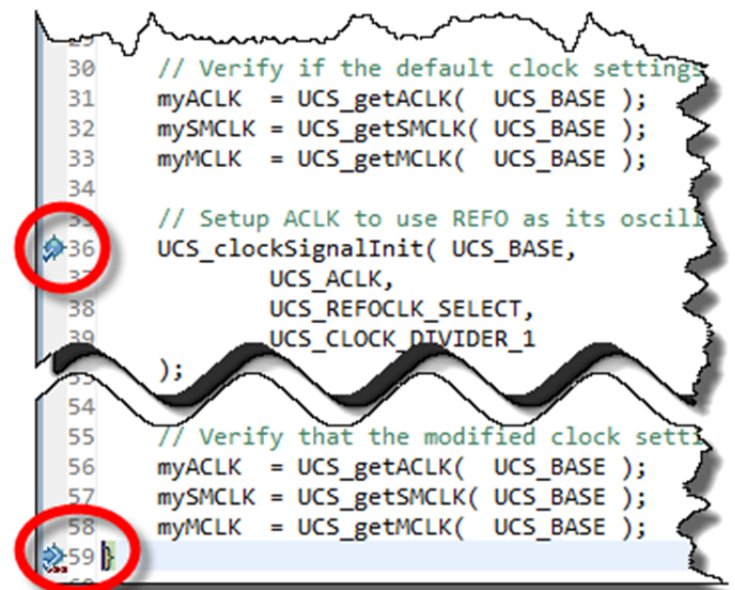
In a minute, this should give us a value of 32, if ACLK is running at 32KHz.

15. Go ahead and create similar watch expressions for `SMCLK` and `MCLK`.

```
mySMCLK/1024
myMCLK/1024
```

16. Finally, let's add two breakpoints to `myClocks.c`.

These breakpoints will let us view the expressions before ... and after our clock initialization code runs.



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.



17. Run the code to the first breakpoint and write down the Express values:

myACLK/1024: _____

mySMCLK/1024: _____

myMCLK/1024: _____

Are these the values that you expected? _____

(Look back at Worksheet question #9, if you need a reminder.)



18. Run to the next breakpoint – at the end of the initClocks() function.

Check on the values again:

myACLK/1024: _____

mySMCLK/1024: _____

myMCLK/1024: _____

Are these the values we were asked to implement? _____

(Look back at Worksheet questions 10-12.)



19. Let the program run from the breakpoint and watch the blinking LED.

Extra Credit (i.e. Optional Step) – Change the Rate of Blinking



20. Halt the processor and terminate the debugger session.

21. Add a function call to `initClocks()` to force MCLK to use the REFO oscillator.

We suggest that you copy/paste the function that sets up ACLK... then change the ACLK parameter to MCLK.

Our code 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 );

```



22. Build your code and launch the debugger.



23. 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.

24. 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...

(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, call it 'pet' or 'feed' 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 keep resetting 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 compilers #include search path setting.

If you need a reminder on how to do this, look back at **Lab3a** under the heading:

“Add MSP430ware Driverlib”



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:

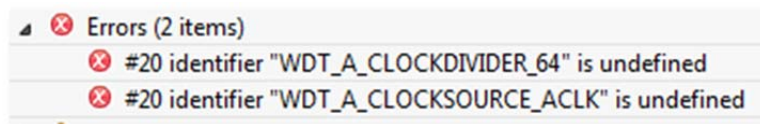
```
WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
```

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 4-38.)



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 this error:



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 | WDTHOLD; // 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, of: 1, 9, 589 (respectively) ... did you wait all the way to 589 before giving up?

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

1. Import `lab_04a_clock_solution`.

If you don't remember how to do this, refer back to lab step 1 (on page 4-31).

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.

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.

```
//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
);
```

By default – on some MSP430 devices, such as the F5529 – these pins default to GPIO mode. Thus, we have to connect them by reprogramming the GPIO.

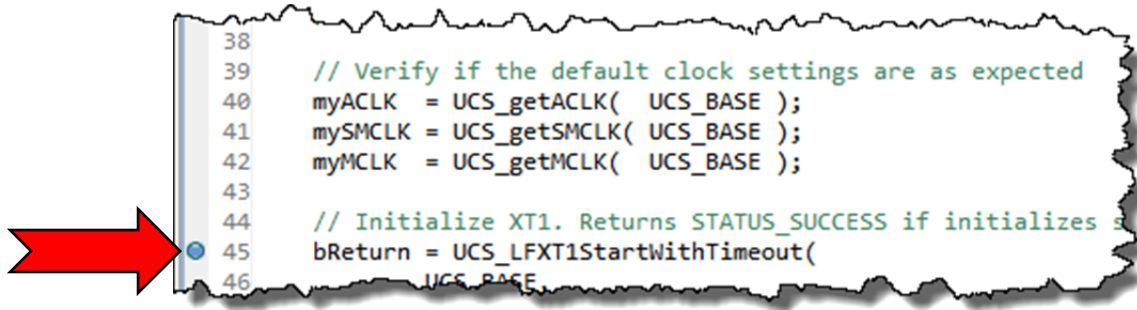
Note: In our solution, we connected all four pins using the `GPIO_setAsPeripheralModuleFunctionInputPin()`. We found this done two different ways in two different examples. One example was similar to ours, the other set the IN pins with the 'InputPin' function, while the setting the OUT pins using the `GPIO_setAsPeripheralModuleFunctionOutputPins()` function. We found that either of these solutions works. We chose the solution with less typing.

9. Build and launch the debugger.

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( UCS_BASE );
41 mySMCLK = UCS_getSMCLK( UCS_BASE );
42 myMCLK = UCS_getMCLK( UCS_BASE );
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:

Expression	Default Settings	First setup	Second setup
<code>myACLK/1024</code>			
<code>mySMCLK/1024</code>			
<code>myMCLK/1024</code>			

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_60_01_11\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_60_01_11\driverlib\driverlib\MSP430F5xx_6xx\uc\ucs.h
- ◆ Finally, we recommend you also reference the driverlib UCS example #4:
msp430\MSP430ware_1_60_01_11\driverlib\examples\MSP430F5xx_6xx\uc\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_BASE, 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
```

7. What function specifies these crystal frequencies to the DriverLib?
Hint: Look in the MSP430ware Driverlib User's Guide – "UCS chapter".

```
UCS_setExternalClockSource ( UCS_BASE,
(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 ( UCS_BASE );
mySMCLK = UCS_getSMCLK ( UCS_BASE );
myMCLK = UCS_getMCLK ( UCS_BASE );
```

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?

ACLK: 32 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'

```
// Setup ACLK
UCS_clockSignalInit ( UCS_BASE,
                     UCS_ACLK,           // Clock to setup
                     UCS_REFOCLK_SELECT // Source clock
                     UCS_CLOCK_DIVIDER_1
                     );
```

or CS_VLOCLK_SELECT

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_BASE,
                     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 ( UCS_BASE,
                  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_BASE,
               CS_DCORSEL_1, // Set Frequency range (DCOR)
               CS_DCOFSEL_3 // Set Frequency (DCOF)
);

// Set MCLK to use DCO clock source
CS_clockSignalInit( CS_BASE,
                   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 );
```

Notes:

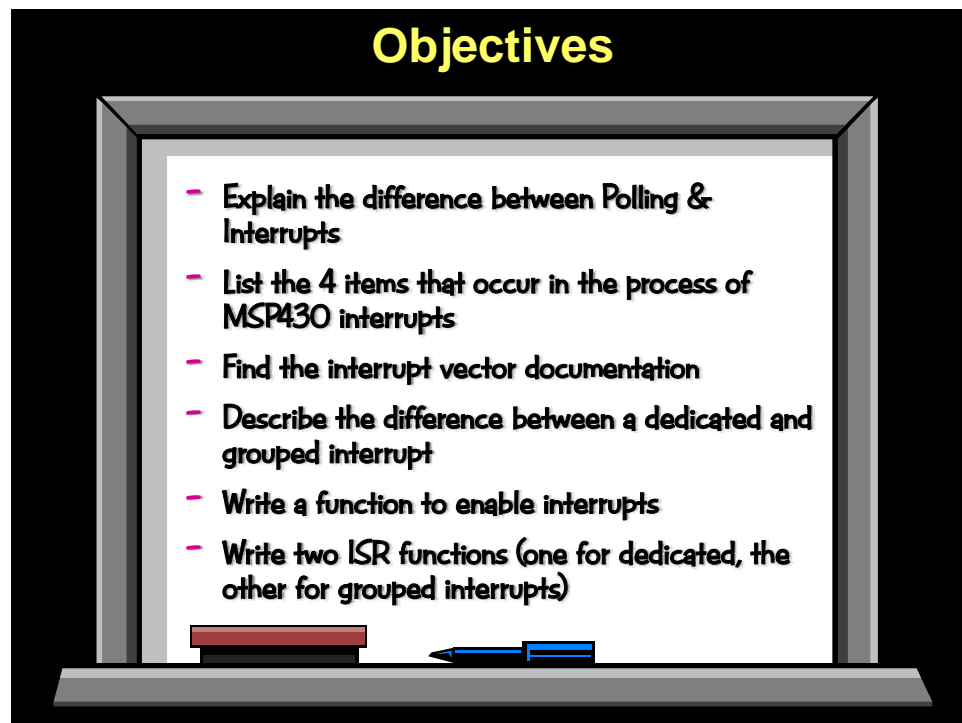
Introduction

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 the 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>
<i>How Interrupts Work</i>	<i>5-5</i>
1. Interrupt Must Occur	5-6
2. Interrupt is Flagged (and must be Enabled)	5-7
3. CPU's Hardware Response	5-8
4. Your Software ISR	5-9
<i>Interrupt Vectors & Priorities</i>	<i>5-10</i>
<i>Coding Interrupts.....</i>	<i>5-12</i>
Dedicated ISR (Interrupt Service Routine).....	5-12
Grouped ISR (Interrupt Service Routine).....	5-13
Enabling Interrupts	5-14
<i>Misc Topics</i>	<i>5-15</i>
<i>Interrupts and TI-RTOS Scheduling.....</i>	<i>5-17</i>
<i>Lab Exercise</i>	<i>5-21</i>

Interrupts, The Big Picture

Waiting for an Event: Family vacation



Polling

Interrupts

Are we there yet?
 Are we there yet?
 Are we there yet?
 Are we there yet?
 Are we there yet?
 Are we there yet?
 Are we there yet?

Wake me up when we get there...

Waiting for an Event: Button Push



Polling

Interrupts

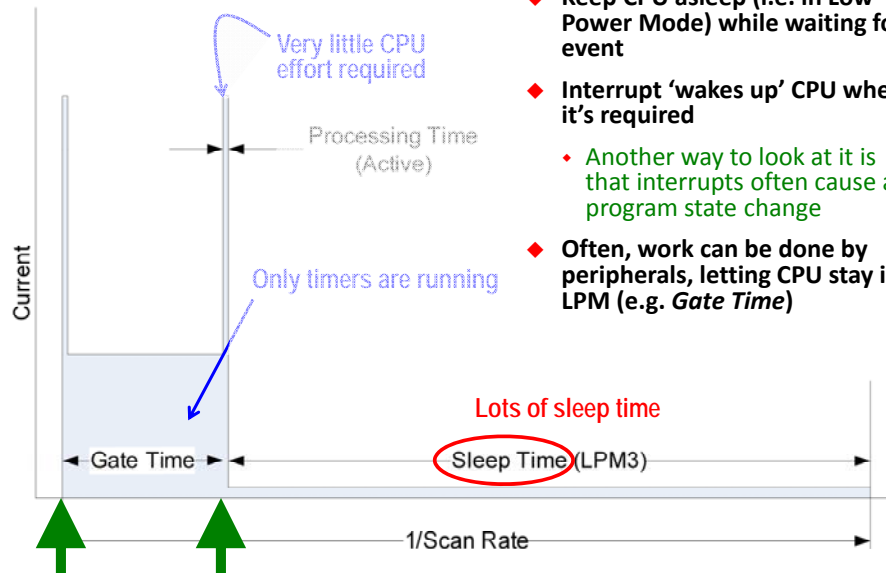
```
while(1) {
    // Polling GPIO button
    while (GPIO_getInputPinValue()==1)
        GPIO_toggleOutputOnPin();
}
```

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

100% CPU Load

> 0.1% CPU Load

Interrupts Help Support Ultra Low Power



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

Foreground / Background Scheduling

```

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

    while(1){
        background
        or LPMx
    }
}

ISR1
get data
process

ISR2
set a flag
    
```

System Initialization

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

Background

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

Foreground

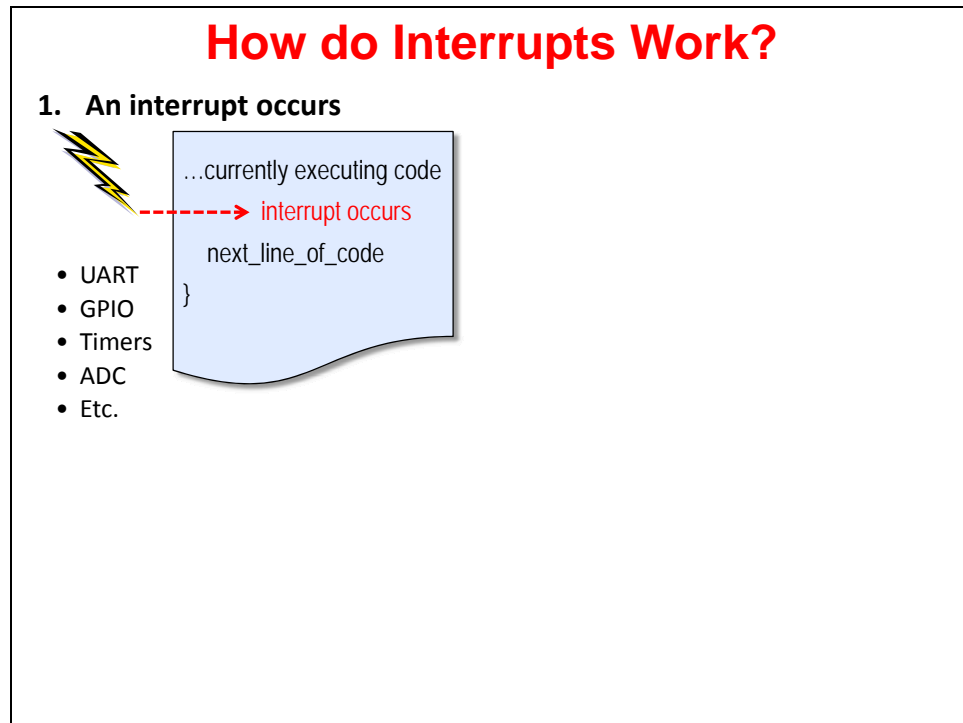
- ◆ **Interrupt Service Routine (ISR)** runs in response to enabled hardware interrupt
- ◆ These events may change modes in Background – such as waking the CPU out of low-power mode
- ◆ ISR's, by default, are not interruptible
- ◆ Some processing may be done in ISR, but it's usually best to keep them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

How Interrupts Work

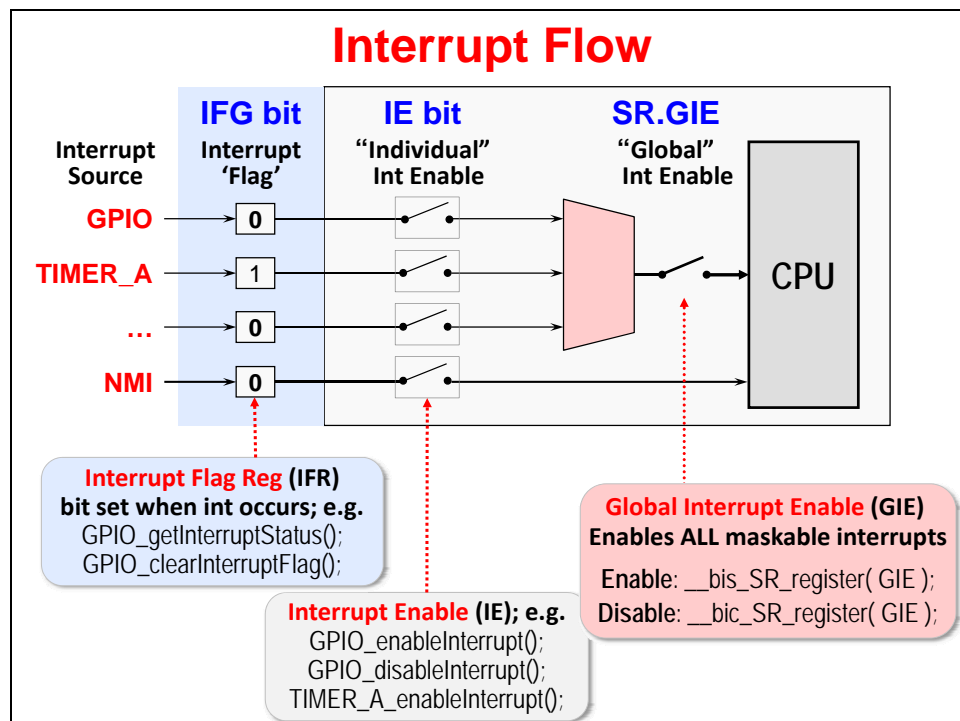
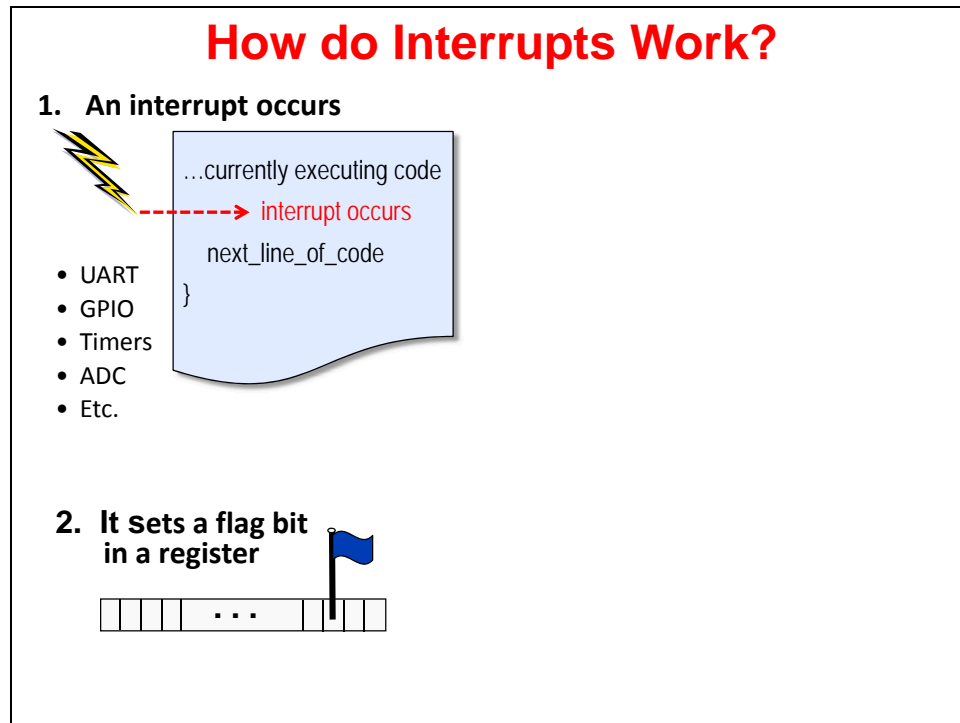
How do Interrupts Work?

Slide left intentionally blank...

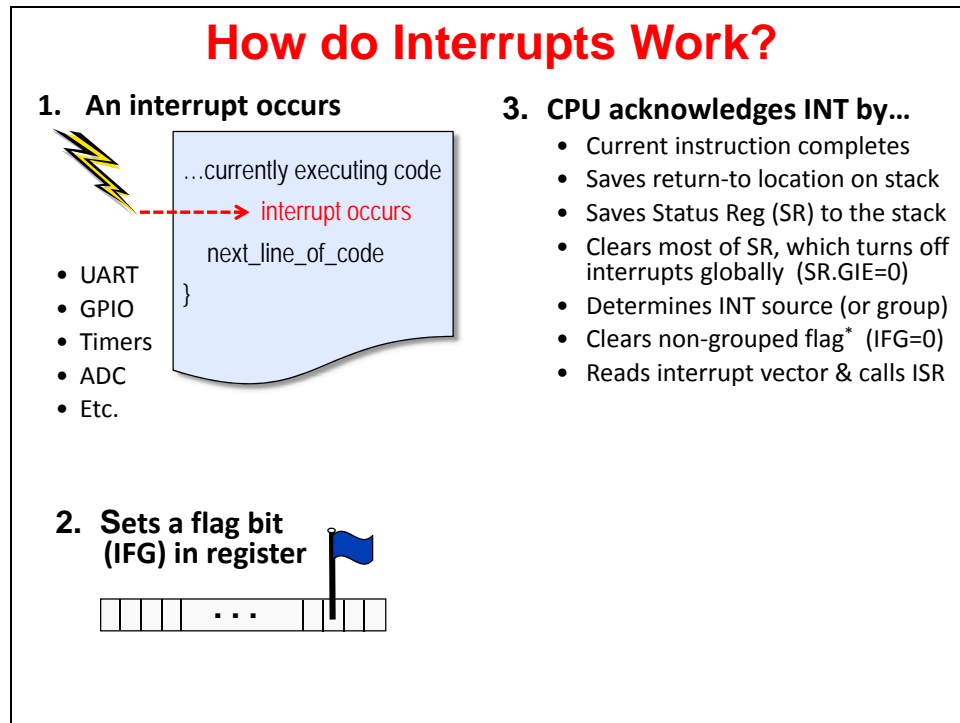
1. Interrupt Must Occur



2. Interrupt is Flagged (and must be Enabled)



3. CPU's Hardware Response



4. Your Software ISR

4. Interrupt Service Routine (ISR)

...currently executing code

next_line_of_code

}

Vector Table

&myISR

Using **Interrupt Keyword**

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

```
#pragma vector=WDT_VECTOR
interrupt myISR(void){
    • Save context of system
    • (optional) Re-enable interrupts
    • *If group INT, read assoc IV Reg (determines source & clears IFG)
    • Run your interrupt's code
    • Restore context of system
    • Continue where it left off (RETI)
}
```

How do Interrupts Work?

1. **An interrupt occurs**
 - UART
 - GPIO
 - Timers
 - A/D Converter
 - Etc.
2. **Sets a flag bit (IFG) in register**
3. **CPU acknowledges INT by...**
 - Current instruction completes
 - Saves return-to location on stack
 - Saves Status Reg (SR) to the stack
 - Clears most of SR, which turns off interrupts globally (SR.GIE=0)
 - Determines INT source (or group)
 - Clears non-grouped flag* (IFG=0)
 - Reads interrupt vector & calls ISR
4. **ISR (Interrupt Service Routine)**
 - Save context of system
 - (optional) Re-enable interrupts
 - *If group INT, read assoc IV Reg (determines source & clears IFG)
 - **Run your interrupt's code**
 - Restore context of system
 - Continue where it left off (RETI)

Interrupt Vectors & Priorities

Interrupt Priorities (F5529)

INT Source	Priority
System Reset	<div style="display: flex; align-items: center; justify-content: center;"> <div style="width: 100%; height: 100%; background: linear-gradient(to bottom, #ccc, #eee);"></div> <div style="margin-left: 5px;"> <p>high</p> <p>low</p> </div> </div>
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	

- ◆ **There are 23 interrupts** (partially shown here)
- ◆ **Most of these represent ‘groups’ of interrupt source flags**
 - ◆ 145 IFG’s map into these 23 interrupts
- ◆ **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

Interrupt Vector (IV) Registers

- ◆ **IV = Interrupt Vector register**
- ◆ **Most MSP430 interrupts can be caused by more than one source; for example:**
 - ◆ Each 8-bi GPIO port one 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**

Interrupt Vectors & Priorities (F5529)

INT Source	IV Register	Vector Address	Loc'n	Priority
System Reset	SYSRSTIV	RESET_VECTOR	63	<div style="display: flex; align-items: center;"> <div style="width: 20px; height: 20px; background-color: #ccc; margin-right: 5px;"></div> <div style="writing-mode: vertical-rl; transform: rotate(180deg);">high</div> </div> <div style="width: 20px; height: 20px; background-color: #ccc; margin-right: 5px;"></div> <div style="writing-mode: vertical-rl; transform: rotate(180deg);">low</div>
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 Converter	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

Memory Map

'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, VLRLIFG, VLRHIFG, VMAIFG, JMBNIFG, JMBOUTIFG (SYSSNIV) ⁽¹⁾	(Non)maskable	0FFFCh	62
User NMI NMI Oscillator Fault Flash Memory Access Violation	NMIIFG, QFIFG, ACCVIFG, RIISIFG (SYSUNIV) ⁽¹⁾⁽²⁾	(Non)maskable	0FFFAh	61
Comparator D	COMP_D interrupt flags (60)	Maskable	0FFE0h	60
DMA	DMA0IFG, DMA1IFG, DMA2IFG (DMAIV) ⁽¹⁾⁽²⁾	Maskable	0FFE4h	50
TA1	TA1CCR0 CCIFG0 ⁽²⁾	Maskable	0FFE2h	49
TA1	TA1CCR1 CCIFG1 to TA1CCR2 CCIFG2, TA1IFG (TA1IV) ⁽¹⁾⁽²⁾	Maskable	0FFE0h	48
I/O Port P1	P1IFG.0 to P1IFG.7 (P1IV) ⁽¹⁾⁽²⁾	Maskable	0FFDEh	47
USCI_A1 Receive or Transmit	UCA1RXIFG, UCA1TXIFG (UCA1IV) ⁽¹⁾⁽²⁾	Maskable	0FFDCh	46
USCI_B1 Receive or Transmit	UCB1RXIFG, UCB1TXIFG (UCB1IV) ⁽¹⁾⁽²⁾	Maskable	0FFDAh	45
TA2	TA2CCR0 CCIFG0 ⁽²⁾	Maskable	0FFD8h	44
TA2	TA2CCR1 CCIFG1 to TA2CCR2 CCIFG2, TA2IFG (TA2IV) ⁽¹⁾⁽²⁾	Maskable	0FFD6h	43
I/O Port P2	P2IFG.0 to P2IFG.7 (P2IV) ⁽¹⁾⁽²⁾	Maskable	0FFD4h	42
RTC_A	RTCRDYIFG, RTCTEVIIFG, RTCAIFG, RTOPSIFG, RT1PSIFG (RTCIV) ⁽¹⁾⁽²⁾	Maskable	0FFD2h	41
			0FFD0h	40

Coding Interrupts

Dedicated ISR (Interrupt Service Routine)

Interrupt Vectors & Priorities (F5529)

INT Source	IV Register	Vector Address	Loc'n	Priority
System Reset	SYSRSTIV	RESET_VECTOR	63	high ↓ low
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

Memory Map

The memory map shows the following components from top to bottom: Flash (128K), INT Vectors (80), RAM (8K), USB RAM (2K), Info Memory (512), Boot Loader (2K), and Peripherals (4K). The INT Vectors are located at address 0xFFFF.

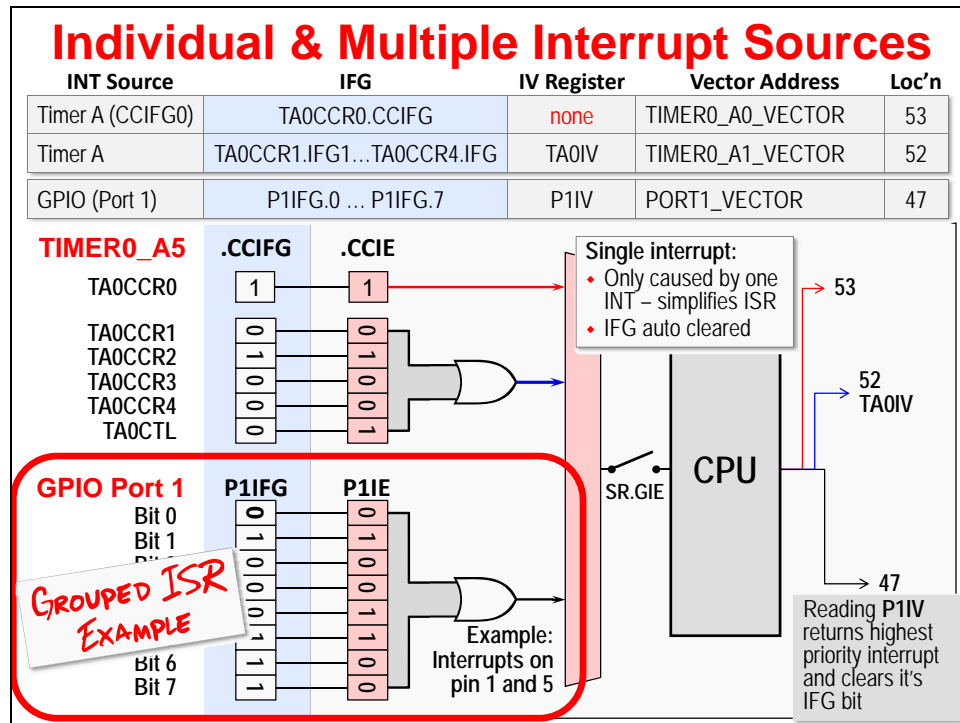
Interrupt Service Routine (Dedicated INT)

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( ... );
}
```

Grouped ISR (Interrupt Service Routine)



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();
    }
}
            
```

Enabling Interrupts

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 );
}
```

Misc Topics

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();
}
```

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)
- ◆ 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, these nested functions could modify unknown registers that were not saved by the interrupt handler does not use.
- ◆ Interrupts can be handled directly with C/C++ functions by using the interrupt pragma or the interrupt keyword
 - ... Conversely, the TI-RTOS kernel easily manages Hwi context
- ◆ 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...

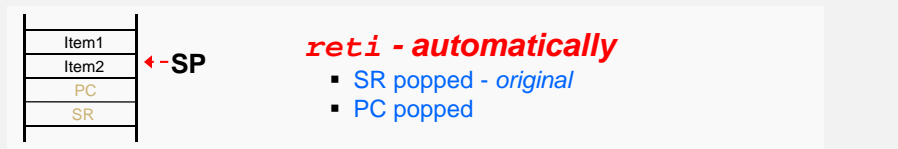
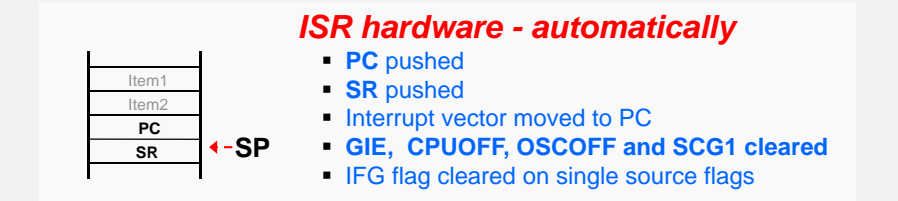
GPIO Interrupt Register Summary

	PA		PB		PC		PD	PJ*
	P1†	P2	P3	P4	P5	P6	P7	P8 (3-bit)
PxIN	All Three Devices support Ports 1 and 2		F5529 and FR5969 (only)		F5529 (only)			F55 & FR59
PxOUT								
PxDIR								
PxREN								
PxD5								
PxSEL								
PxIV				FR5969 (only)				
PxIES								
PxIE								
PxIFG								

NOTE: Only 2 (or 4) 8-bit I/O Ports support interrupts

- ◆ P1IV: Interrupt Vector Highest Priority Interrupt enabled on Port 1
- ◆ P1IES: Interrupt Polarity Select (0 = triggered on high/low edge? (0 = low-to-high))
- ◆ P1IE: Interrupt Enable register for Port 1
- ◆ P1IFG: Interrupt Flag register for Port 1

Interrupt Processing



Interrupts and TI-RTOS Scheduling

What is a Thread?

```

main() {
  init code
  while(1) {
    nonRT Fxn
  }
}
            
```

Background thread

```

UART ISR
get byte
process
output
            
```

Foreground threads

```

Timer ISR
Scan keyboard
            
```

- ◆ 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**

Foreground / Background Scheduling

```

main() {
  init code
  while(1) {
    nonRT Fxn
  }
}
            
```

```

H/W ISR
get data
process
printf()
            
```

```

main() {
  init
  BIOS_start()
}
            
```

Idle
nonRT
+ instrumentation

Hwi
get data
process
LOG_info1()

RTOS Scheduler

- ◆ Idle events run in sequence when no Hwi's are posted
- ◆ Hwi is ISR with automatic vector table generation + context save/restore
- ◆ Hwi performs "process" – typical use is to perform HRT need, then post "follow-up activity"

Notes:

TI-RTOS Thread Types – More Design Options

Priority ↑

- Hwi**
Hardware Interrupts
 - ◆ Hardware event triggers Hwi to run
 - ◆ BIOS handles context save/restore, nesting
 - ◆ Hwi triggers follow-up processing
 - ◆ Priorities set in silicon
- Swi**
Software Interrupts
 - ◆ Software posts Swi to run
 - ◆ Performs Hwi 'follow-up' activity (process data)
 - ◆ Up to 32 priority levels (16 on C28x)
 - ◆ Often favored by traditional h/w interrupt users
- Task**
Tasks
 - ◆ Usually enabled to run by posting a 'semaphore' (a task signaling mechanism) (similar to Posix)
 - ◆ Designed to run concurrently – pauses when waiting for data (semaphore)
 - ◆ Favored by folks experienced in high-level OS's
- Idle**
Background
 - ◆ Runs as an infinite while(1) loop
 - ◆ Users can assign multiple functions to Idle
 - ◆ Single priority level

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:

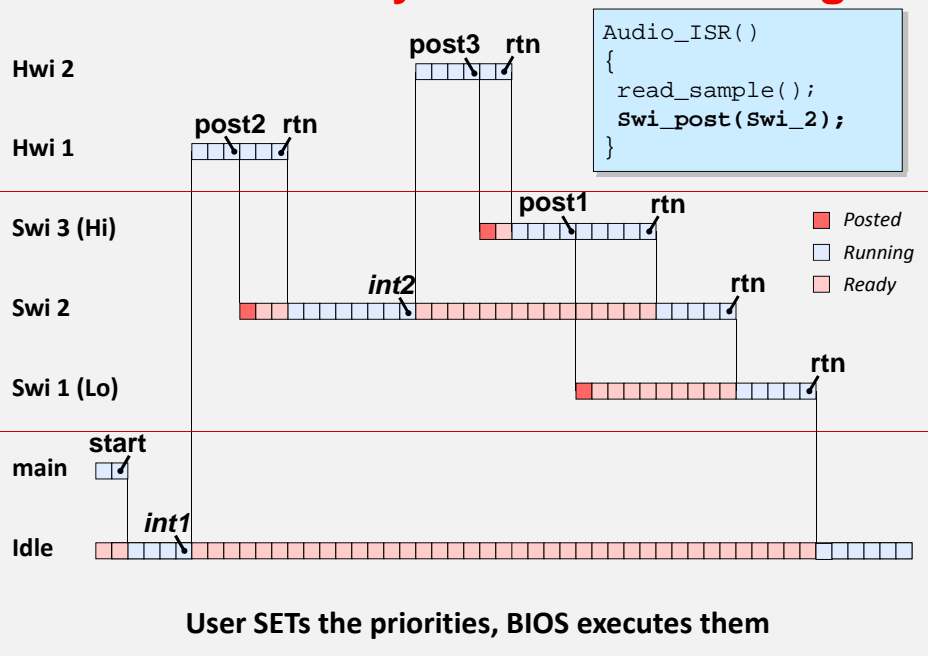
- ◆ **Memory Mgmt** (stack, heap, cache)
- ◆ **Real-time Analysis** (logs, graphs, loads)
- ◆ **Scheduling** (various thread types)
- ◆ **Synchronization** (e.g. semaphores, events)

Attend the 2-day TI RTOS kernel workshop for more information on all of these services

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

BIOS – Priority Based Scheduling



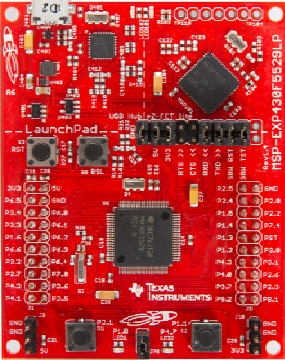
Lab 5 – Interrupts

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



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.

Chapter Topics

Interrupts	5-20
<i>Lab 5 – Interrupts</i>	<i>5-21</i>
Lab 5 Worksheet	5-23
General Interrupt Questions	5-23
Interrupt Flow	5-23
Interrupt Priorities & Vectors	5-24
ISR’s for Group Interrupts	5-25
Lab 5a – Push Your Button	5-26
File Management	5-26
Configure/Enable GPIO Interrupt ... Then Verify it Works.....	5-29
Add a Simple Interrupt Service Routine (ISR)	5-32
Sidebar – Vector Errors.....	5-32
Upgrade Your Interrupt Service Routine (ISR)	5-34
(Optional) Lab 5b – Can You Make a Watchdog Blink?	5-35
Import and Explore the WDT_A Interval Timer Example	5-35
Run the code	5-37
Change the LED blink rate	5-37
Appendix	5-38

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 interrupt 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

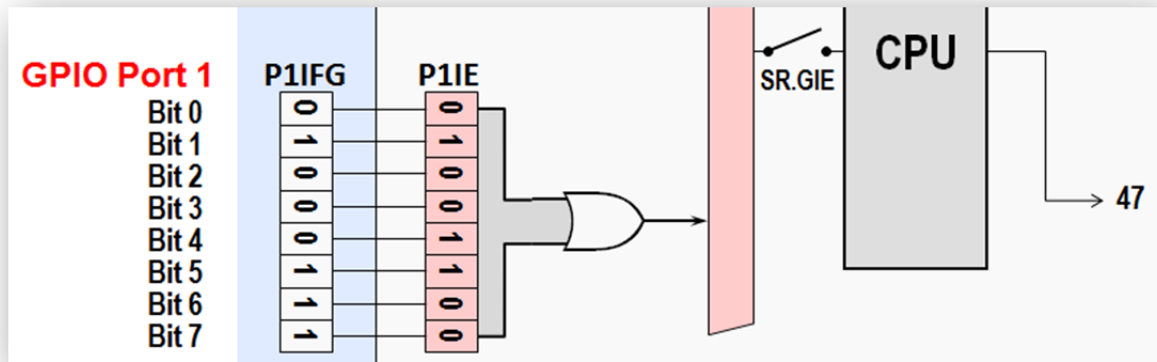
```
__interrupt void pushbutton_ISR (void)
{
    // Toggle the LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```

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) 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: //break;         // Pin 1

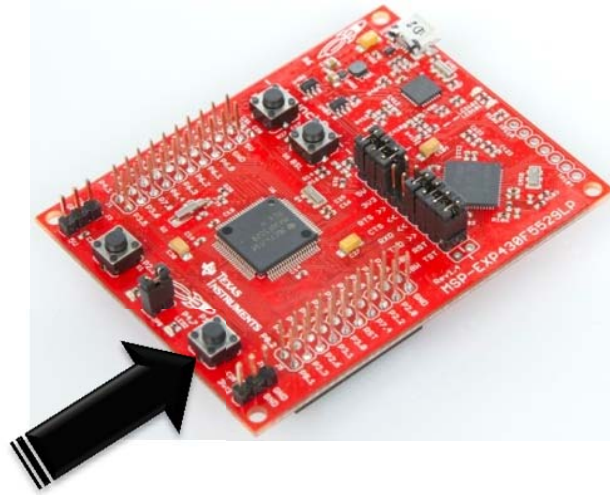
        break;
        case 0x06: break;           // Pin 2
        case 0x08: break;           // Pin 3
        case 0x0A: break;           // Pin 4
        case 0x0C: break;           // Pin 5
        case 0x0E: break;           // Pin 6
        case 0x10: break;           // Pin 7
        default:  __never_executed();
    }
}
```

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

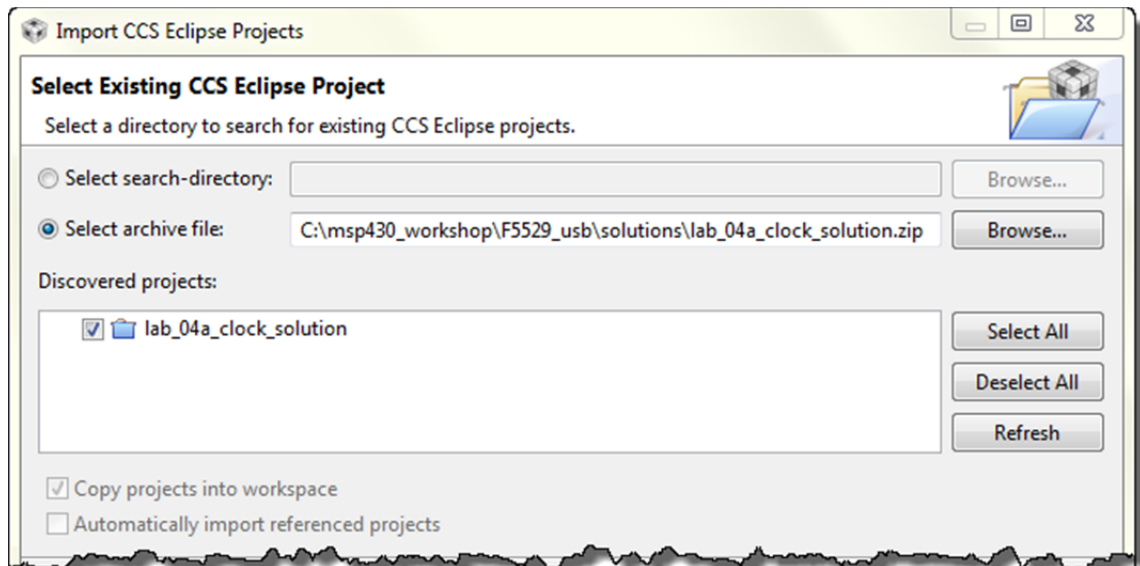


File Management

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

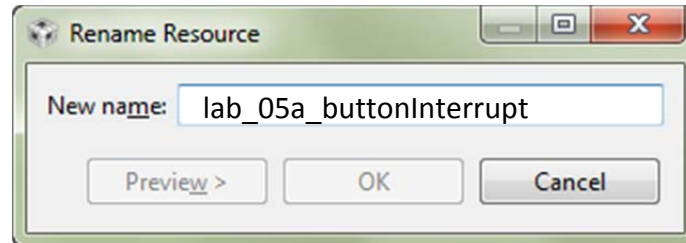


3. Rename the imported project to: lab_05a_buttonInterrupt

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 **C**

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 5-23).

Note that the pin numbers are the same, but the switch names differ for these Launchpads:

- For the F5529 Launchpad, we’re using pushbutton SW2 (P1.1)
- For the FR5969 Launchpad, we’re using pushbutton SW3 (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 5-24).



9. Build your code.

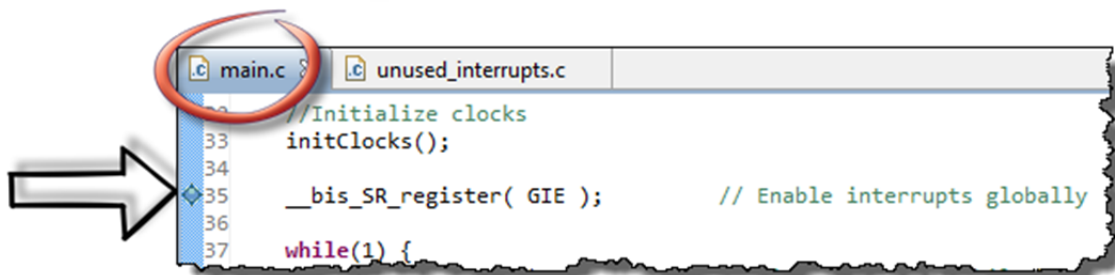
Fix any typos or errors.

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.

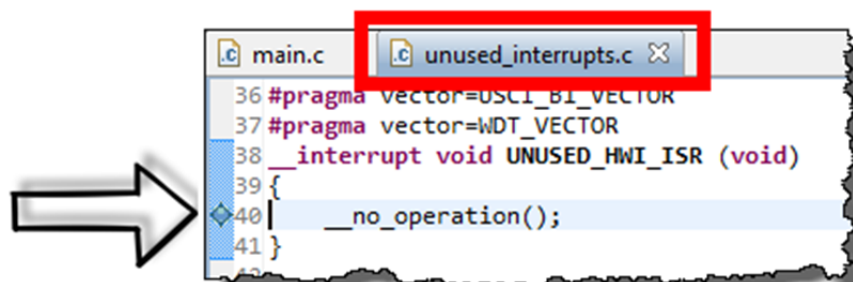
11. Set a breakpoint on the “enable GIE” line of code in `main.c`.



```

main.c | unused_interrupts.c
32 //Initialize clocks
33 initClocks();
34
35 __bis_SR_register( GIE );           // Enable interrupts globally
36
37 while(1) {
  
```

12. Next, set a breakpoint inside the ISR in the `unused_interrupts.c` file.



```

main.c | unused_interrupts.c X
36 #pragma vector=USCI_B1_VECTOR
37 #pragma vector=WDT_VECTOR
38 __interrupt void UNUSED_HWI_ISR (void)
39 {
40 |   __no_operation();
41 }
  
```

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 [I]
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
CPUOFF	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 α 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 o

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 5-24).



21. Build your program to test for any errors.

You should have gotten the error ...

```
../driverlib/MSP430F5xx_6xx/adc10_a.obj" ../driverlib/MSP430F5xx_6xx/dak_batt.o
"./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.

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=SYSCONTROL_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 RED LED on/off
66     GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
67 }

```



27. Run your code ... nce 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 13 from our worksheet (page 5-25). 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 5-25).

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.

Select the 'ugly' code and press **F** - **I**

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.

There are two different ways we can import the example project:

- Use the Project→Import Existing CCS Eclipse Project (as we've done before)
- Utilize the TI Resource Explorer (as we did to import our 'Empty Project' in Lab3)

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

Help → Welcome to CCS

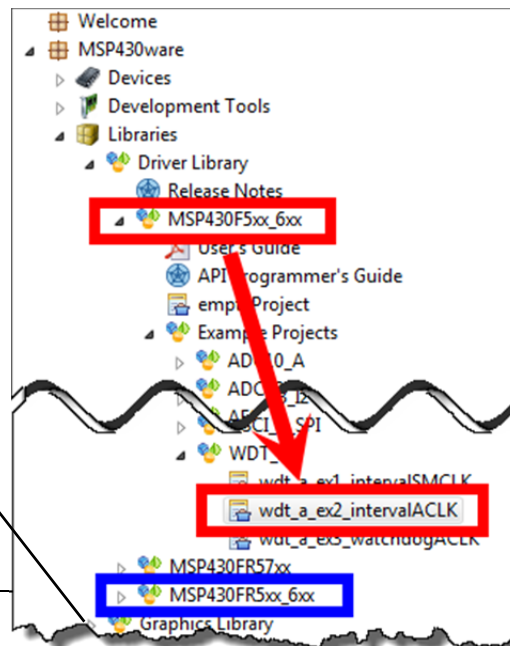
Hint: If you don't see a listing of resource in the window, click the *Home* button.



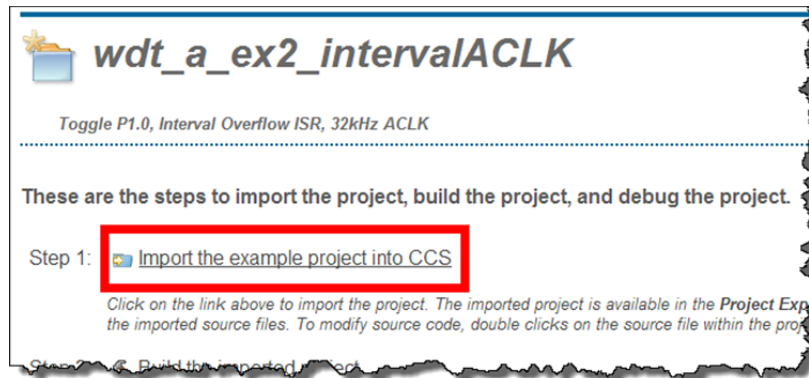
b) Locate the `wdt_a_ex2_intervalACLK` example.

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 the 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.)

Hopefully this ISR is straight-forward. It uses the #pragma to set up the vector; and, it manages the context using the __interrupt keyword. 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.

```

32 void main(void)
33 {
34     //Initialize WDT module in timer interval mode,
35     //with ACLK as source at an interval of 250 ms.
36     WDT_A_intervalTimerInit(WDT_A_BASE,
37                             WDT_A_CLOCKSOURCE_ACLK,
38                             WDT_A_CLOCKDIVIDER_8192);
39
40     WDT_A_start(WDT_A_BASE);
41
42     //Enable Watchdog Interrupt
43     SFR_clearInterrupt(SFR_BASE,
44                       WDTIFG);
45     SFR_enableInterrupt(SFR_BASE,
46                       WDTIE);
47
48     //Set P1.0 to output direction
49     GPIO_setAsOutputPin(
50         GPIO_PORT_P1,
51         GPIO_PIN0
52     );
53
54     //Enter LPM3, enable interrupts
55     __bis_SR_register(LPM3_bits + GIE);
56     //For debugger
57     __no_operation();
58 }
59
60 //Watchdog Timer interrupt service routine
61 #pragma vector = WDT_VECTOR
62 __interrupt 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 ...

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 2 second intervals.

(Hint: choose clock divide by 32K.)



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

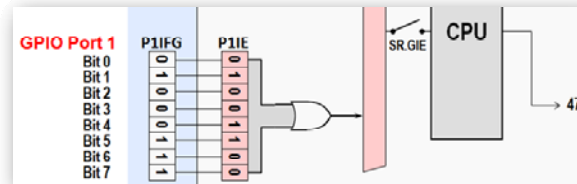
Lab 05 Worksheet (5)

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

Often times, Timers are the heartbeat (and lifeblood) of an embedded system.

Whether you need a periodic wake-up call, a one-time delay, or need a means of verifying that the system is running without a failure, Timers are the solution.

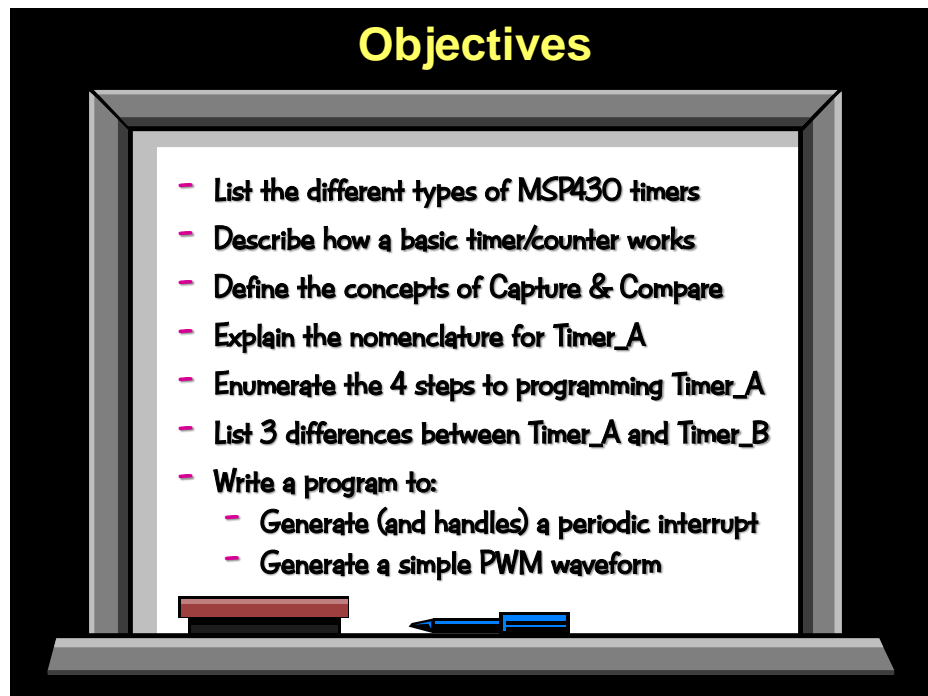
This chapter begins with a brief summary of the Timers found on the MSP430F5529 device. Most of the chapter, though, is spent digging into the details of the MSP430's TIMER_A module. Not only does it provide the rudimentary counting/timing features, but provides sophisticated capture and compare features that allow a variety of complex waveforms – or interrupts – to be generated. In fact, this timer can even generate PWM signals.

Along the way, we examine the MSP430ware DriverLib code required to setup and utilize TIMER_A.

The chapter nears conclusion with a brief summary of the difference between TIMER_A and TIMER_B. The single sentence summary of TIMER_B would be ... if you know how to use TIMER_A, then you can use TIMER_B.

Finally, we borrow a little advice from the author of MSP430 Microcontroller Basics¹. His summary of which MSP430 timer to use, and when, is spot on.

Learning Objectives



¹ MSP430 Microcontroller Basics by John H. Davies, (ISBN-10 0750682760) [Link](#)

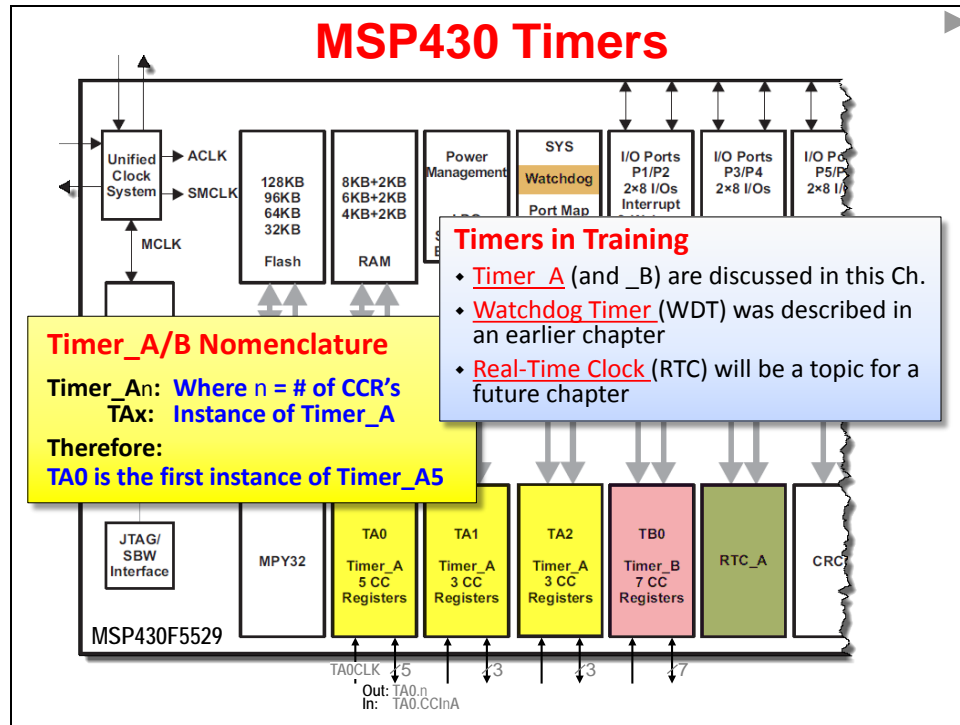
Chapter Topics

Timers	6-1
<i>Prerequisites and Tools</i>	<i>6-2</i>
<i>Overview of MSP430 Timers</i>	<i>6-3</i>
<i>Timer Basics: How timers work.....</i>	<i>6-4</i>
Counter.....	6-4
Capture.....	6-4
Compare.....	6-5
<i>Timer Details: Configuring TIMER_A.....</i>	<i>6-7</i>
1. Counter: TIMER_A_configure...()	6-7
2a. Capture: TIMER_A_initCapture()	6-12
2b. Compare: TIMER_A_initCompare()	6-13
3. Clear Interrupt Flags and TIMER_A_startTimer()	6-18
4. Interrupt Code (Vector & ISR).....	6-19
<i>TIMER_A API Summary</i>	<i>6-20</i>
<i>Differences between Timer's A and B.....</i>	<i>6-21</i>
<i>Lab Exercise</i>	<i>6-23</i>

Prerequisites and Tools

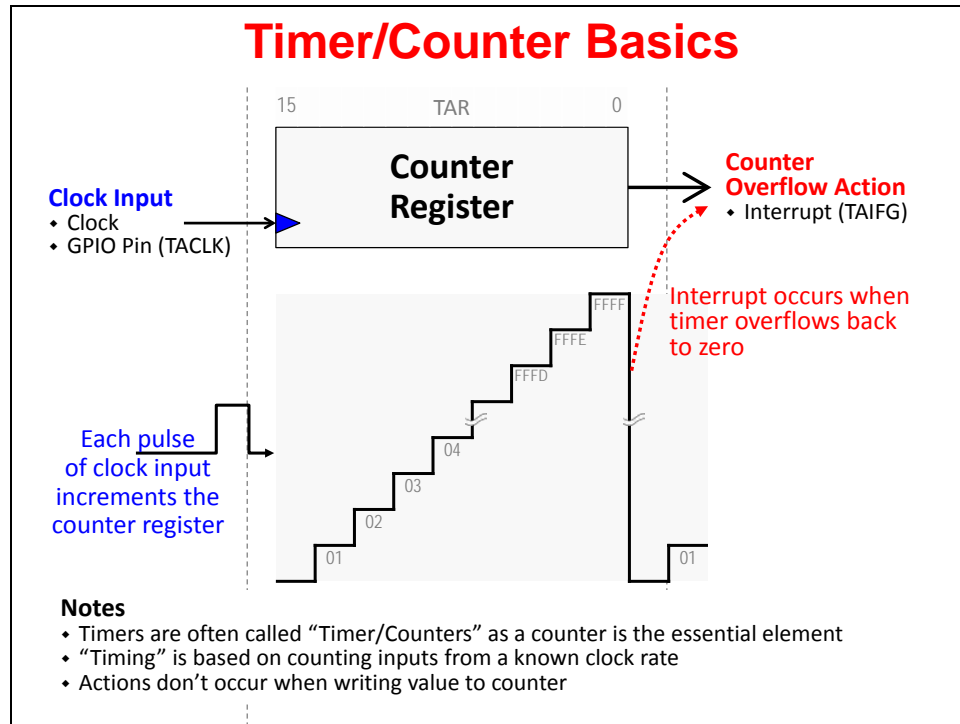
Prerequisites & Tools	
◆ Skills	Chapter
◆ Creating a CCS Project for MSP430 Launchpad(s)	(Ch 2)
◆ Basic knowledge of:	
◆ C language	
◆ Using a C libraries and header files (MSP430ware DriverLib)	(Ch 3)
◆ Setting up MSP430 clocks	(Ch 4)
◆ Using interrupts (setup and ISR's)	(Ch 5)
◆ Hardware	
◆ Windows (XP, 7, 8) PC with available USB port	
◆ MSP430F5529 Launchpad (with included USB micro cable)	
◆ One (1) jumper wire (female to female)	
◆ Software	
◆ CCSv5.5	
◆ MSP430ware (v1.60.02.09)	

Overview of MSP430 Timers

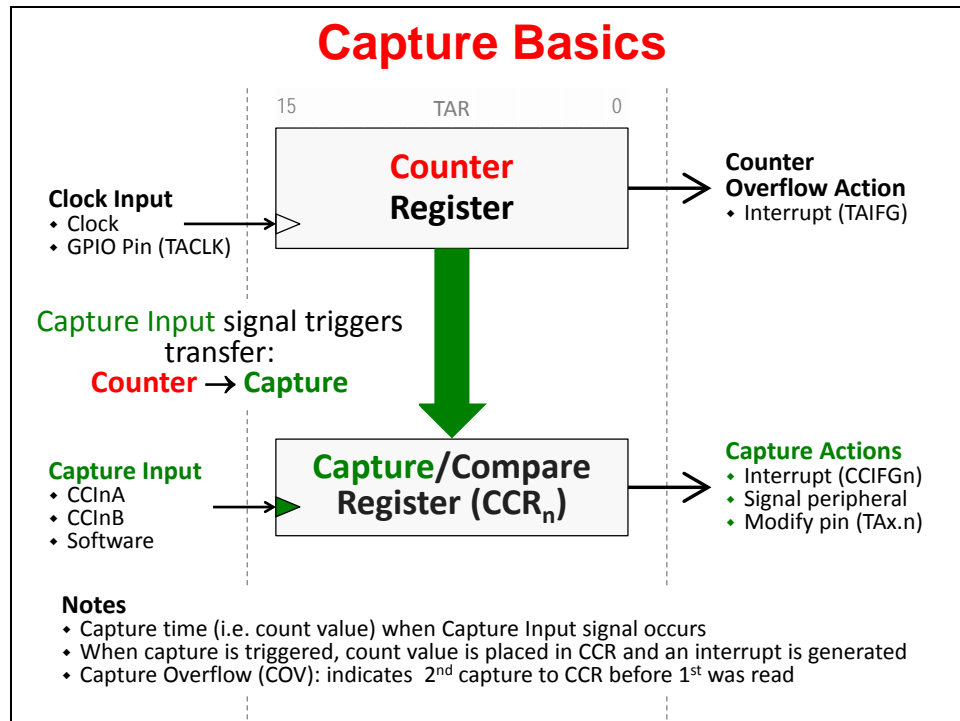


Timer Basics: How timers work

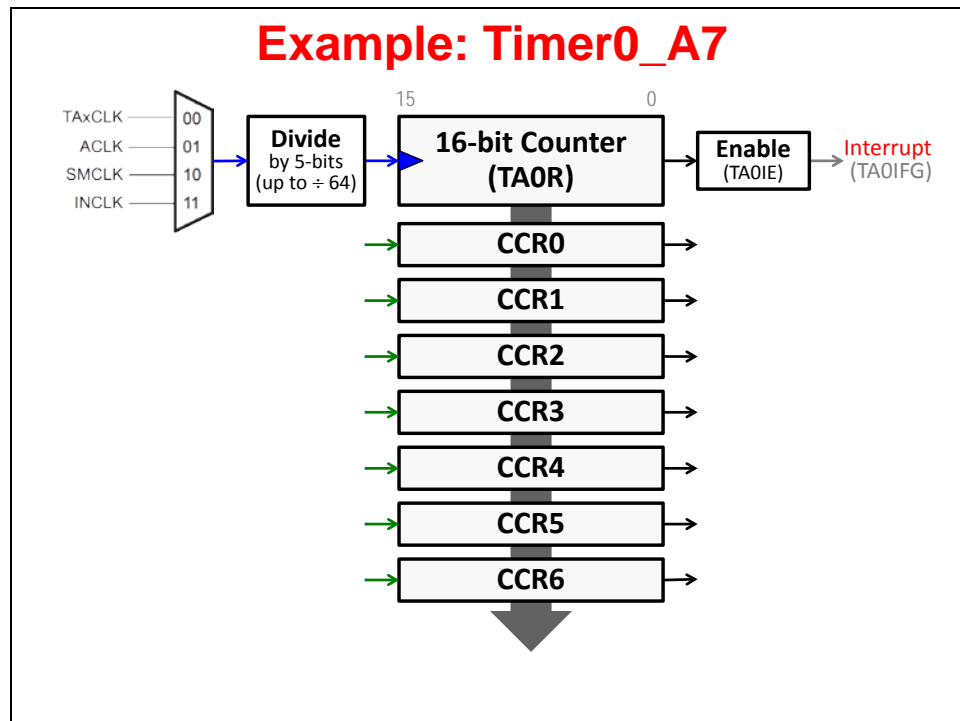
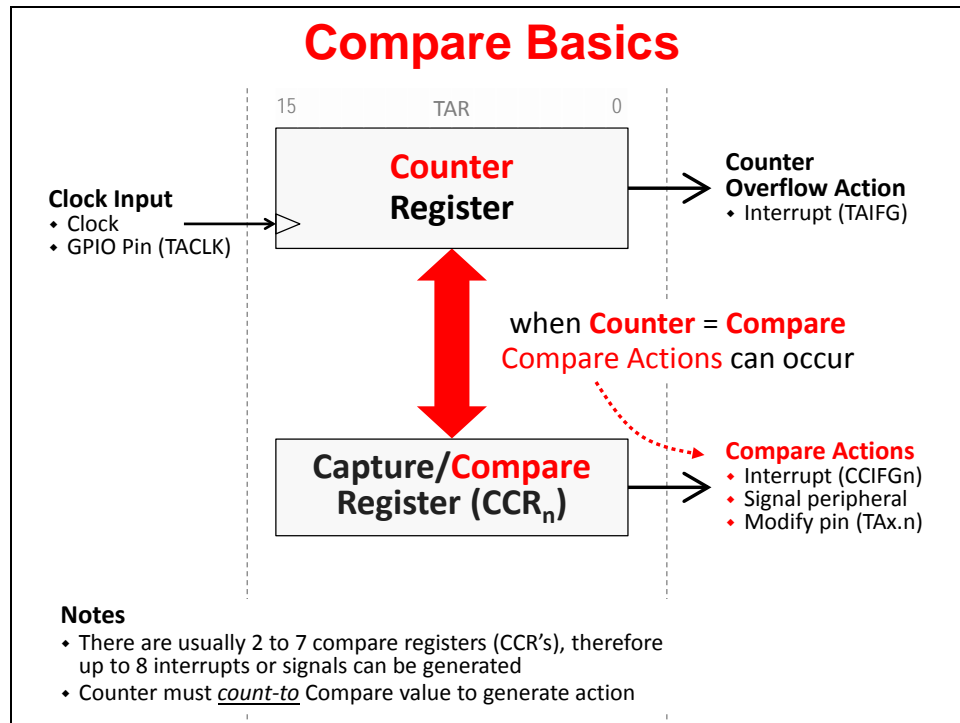
Counter



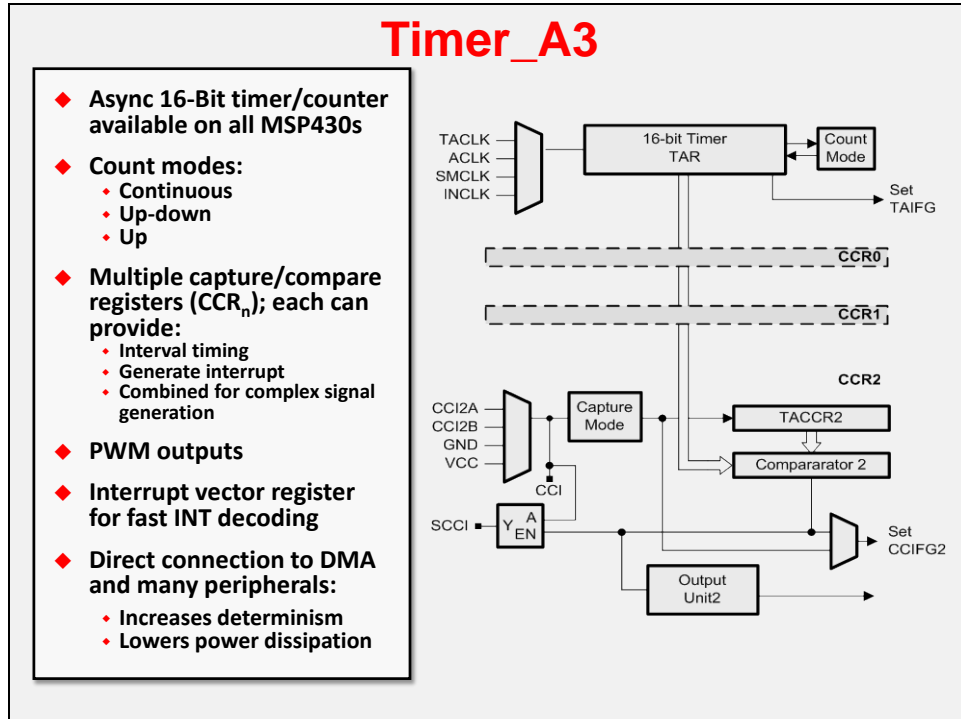
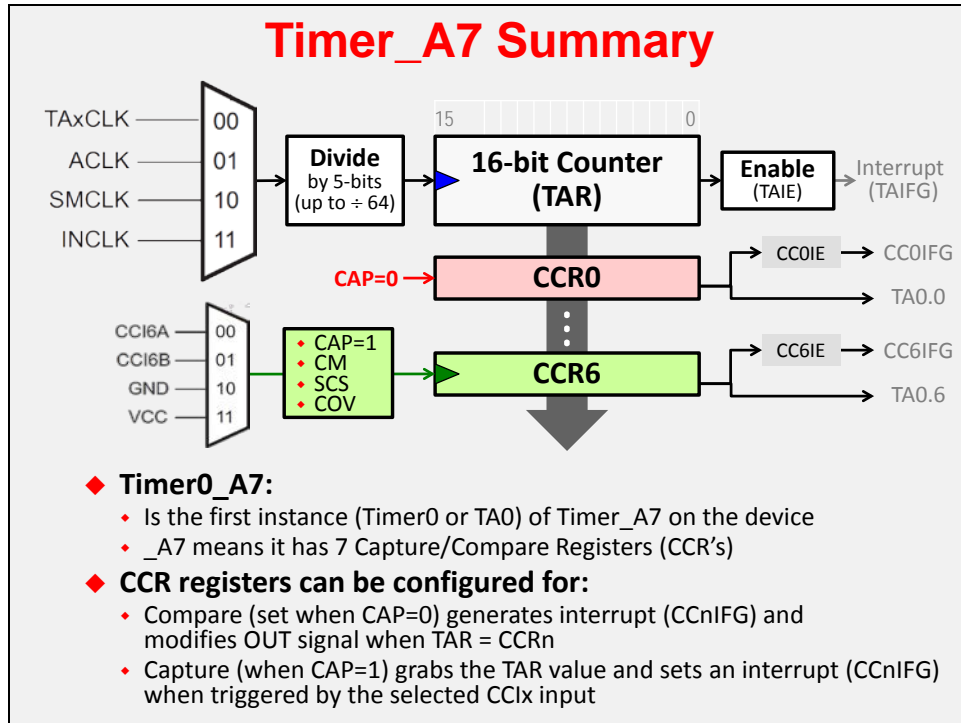
Capture



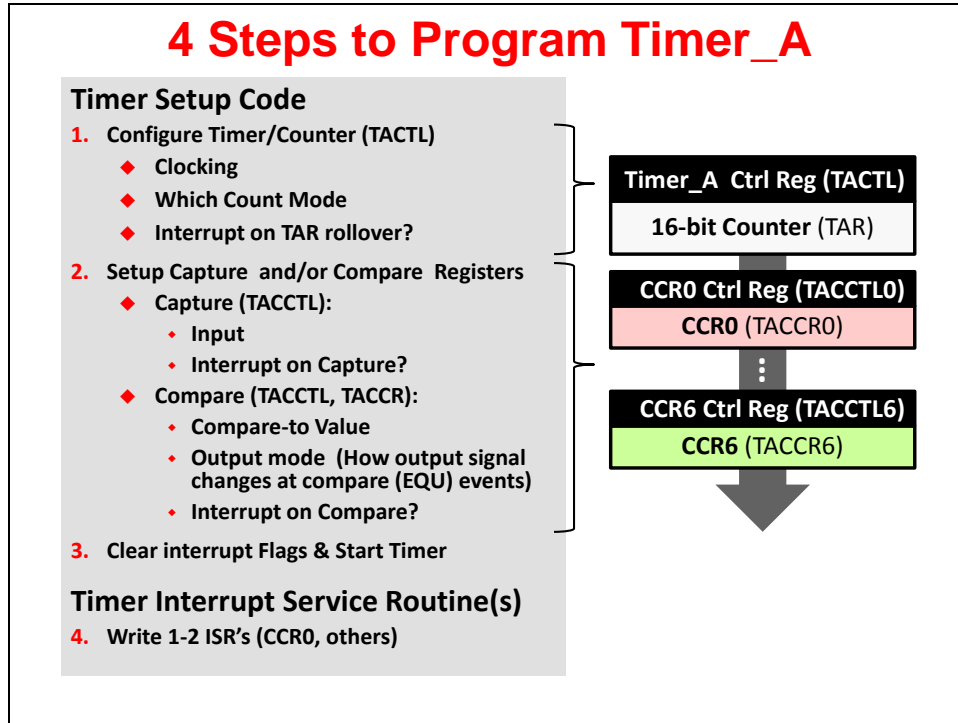
Compare



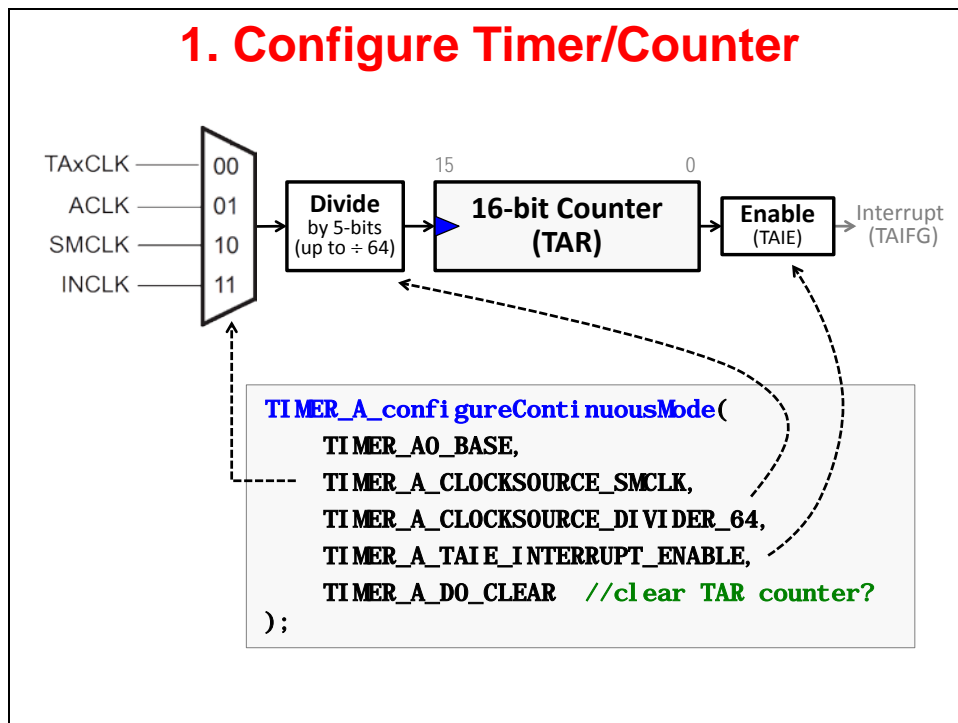
Summary



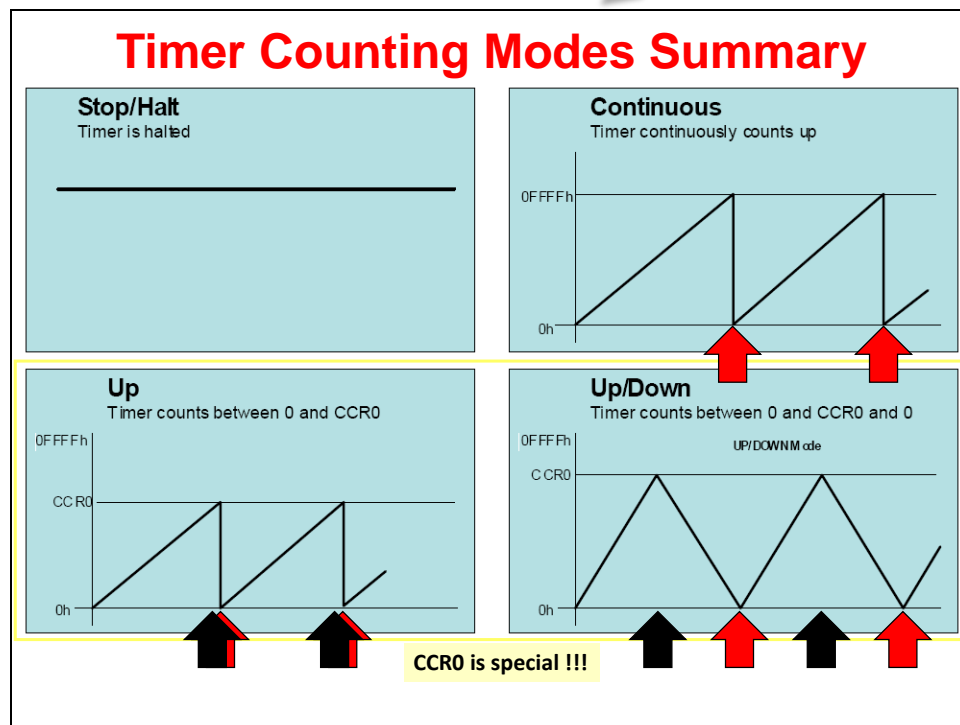
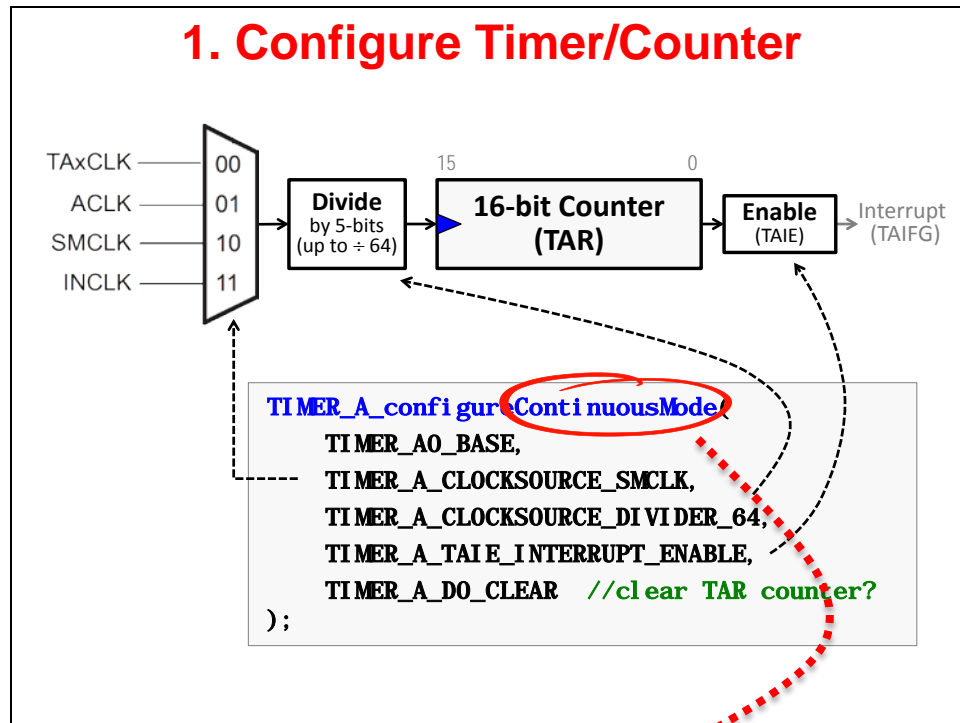
Timer Details: Configuring TIMER_A



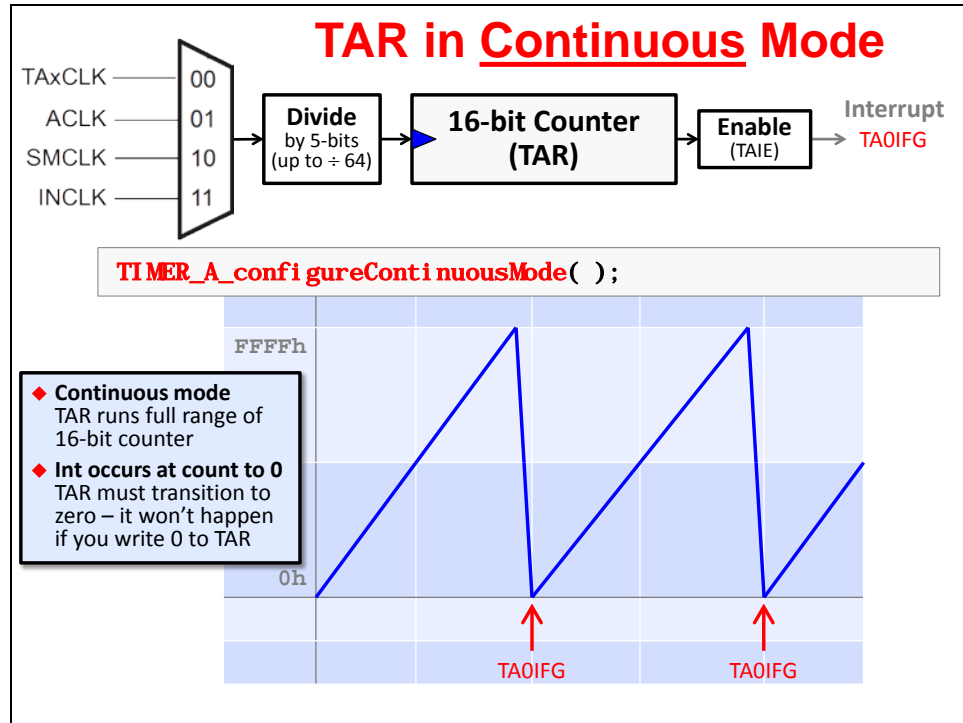
1. Counter: TIMER_A_configure...()



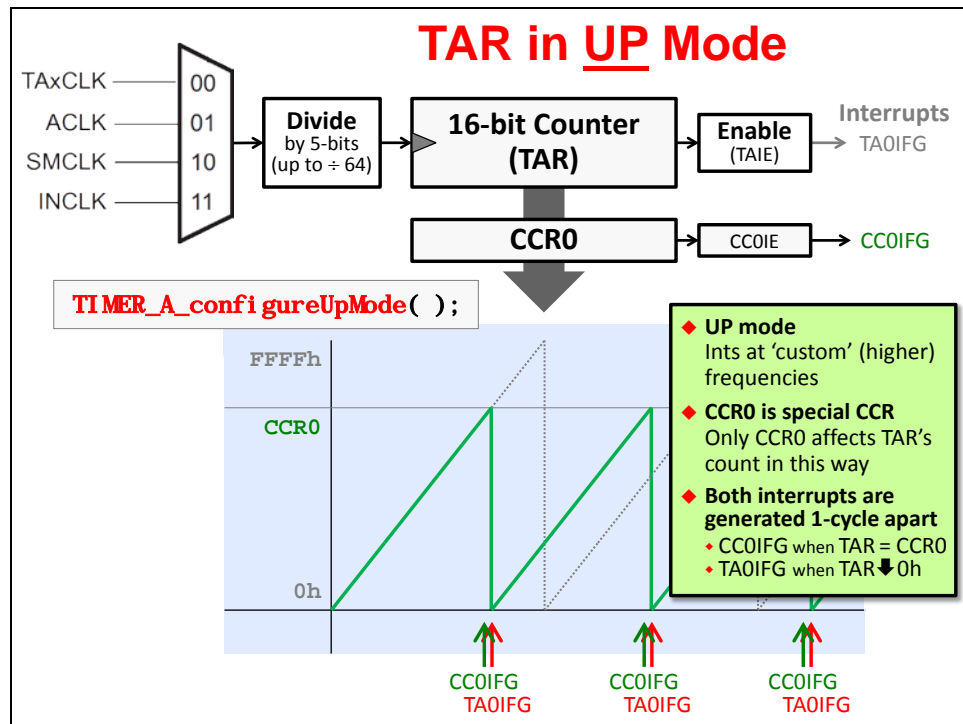
Timer Counting Modes



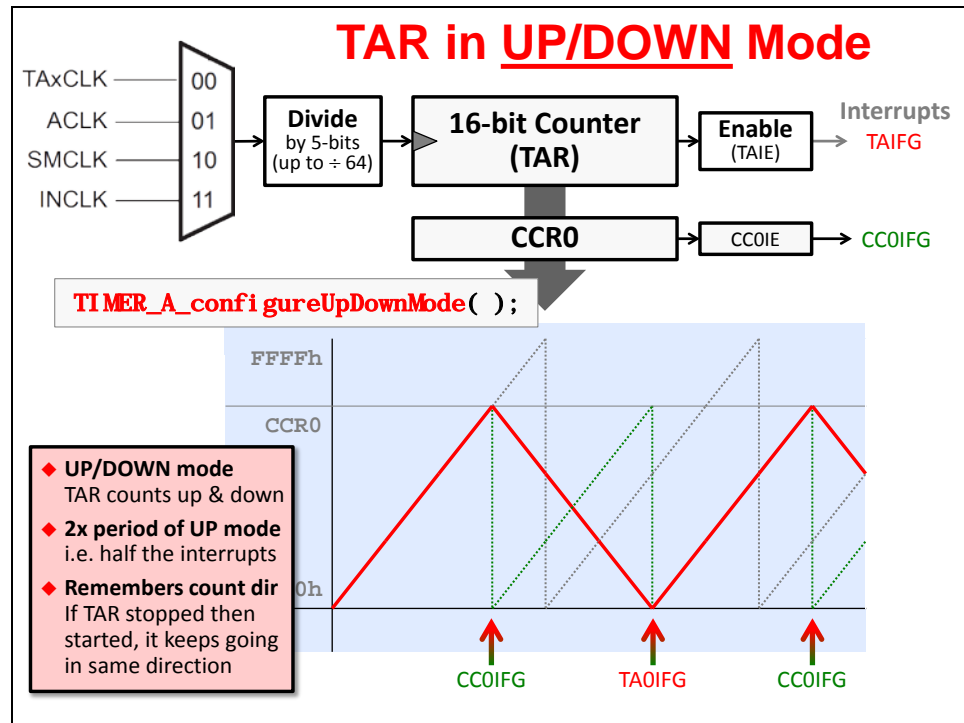
Continuous Mode



Up Mode



Up/Down Mode



Summary of Timer Setup Code – Part 1

Timer Code Example (Part 1)

```
#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
    initTimers();

    __bis_SR_register( GIE );
    while(1) {
        ...
    }
}
```

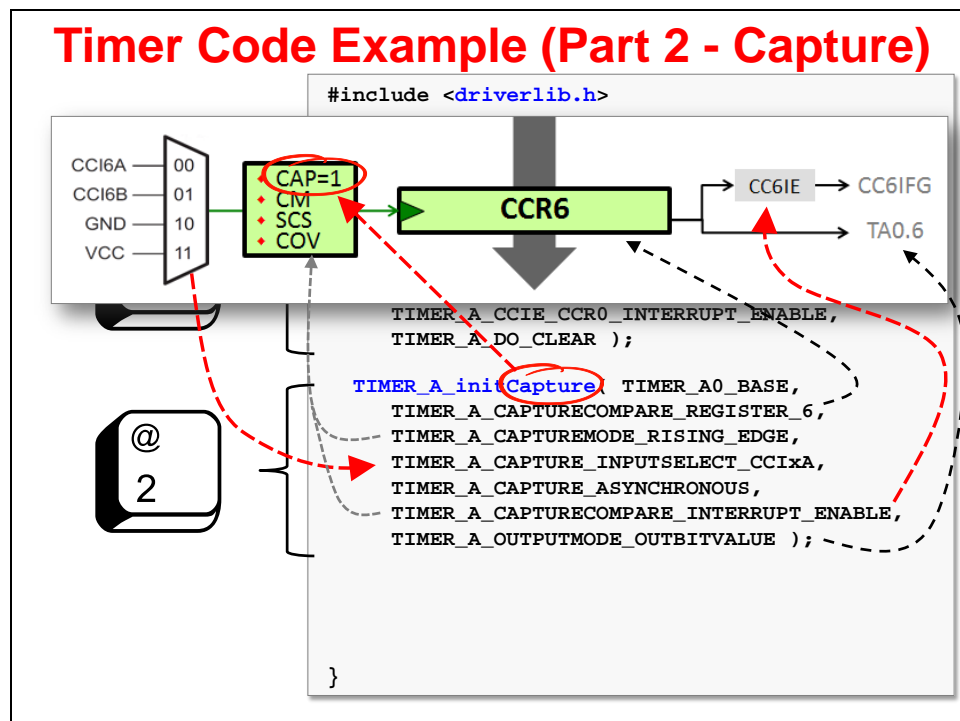
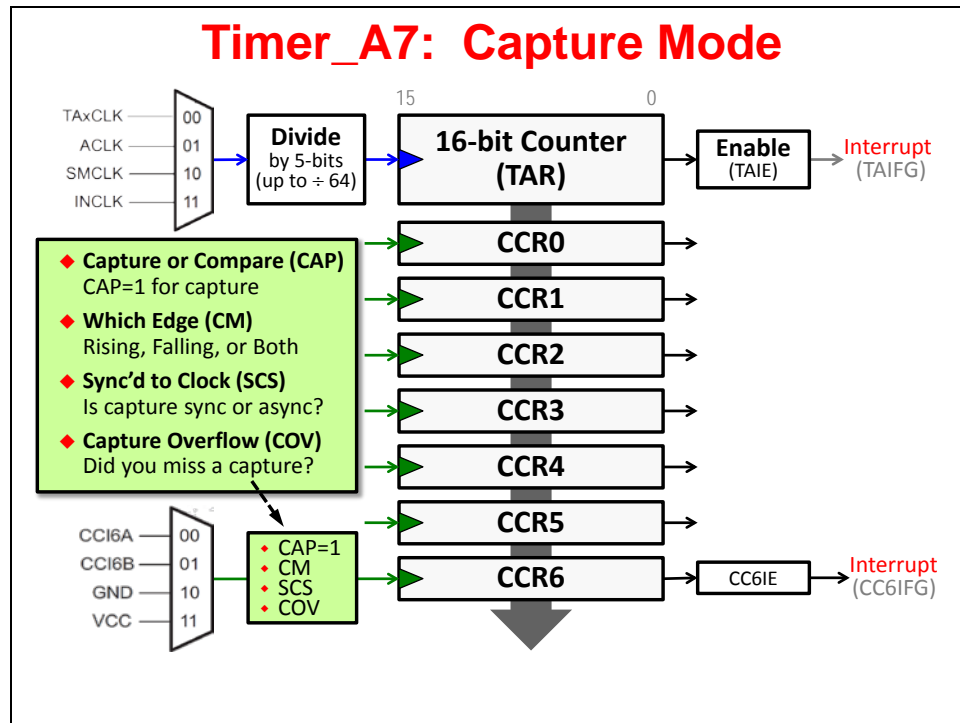
Timer Code Example (Part 1)



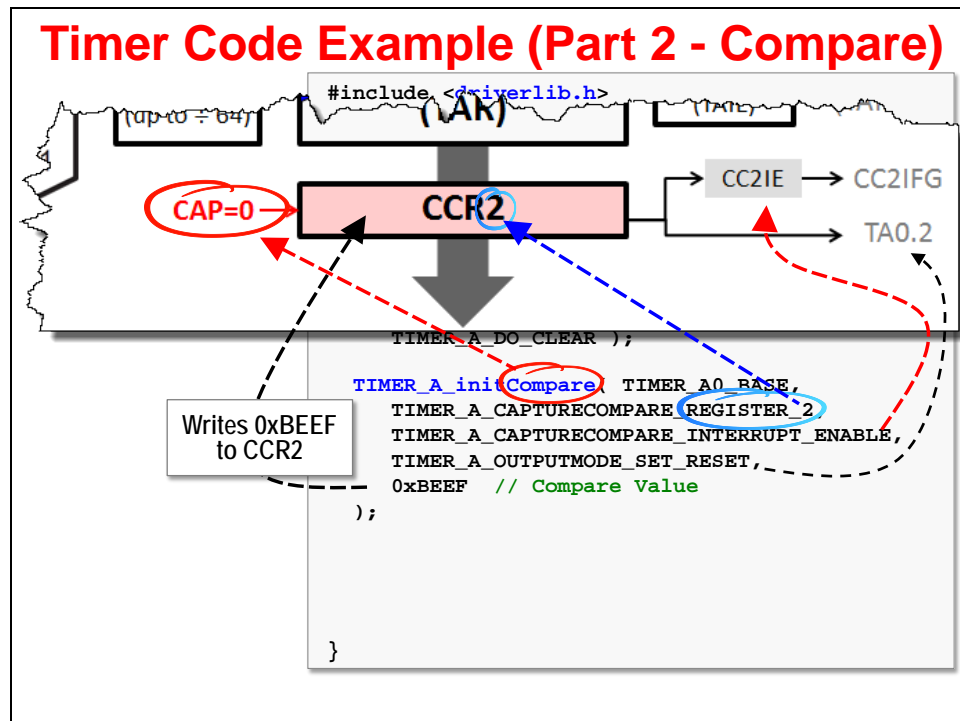
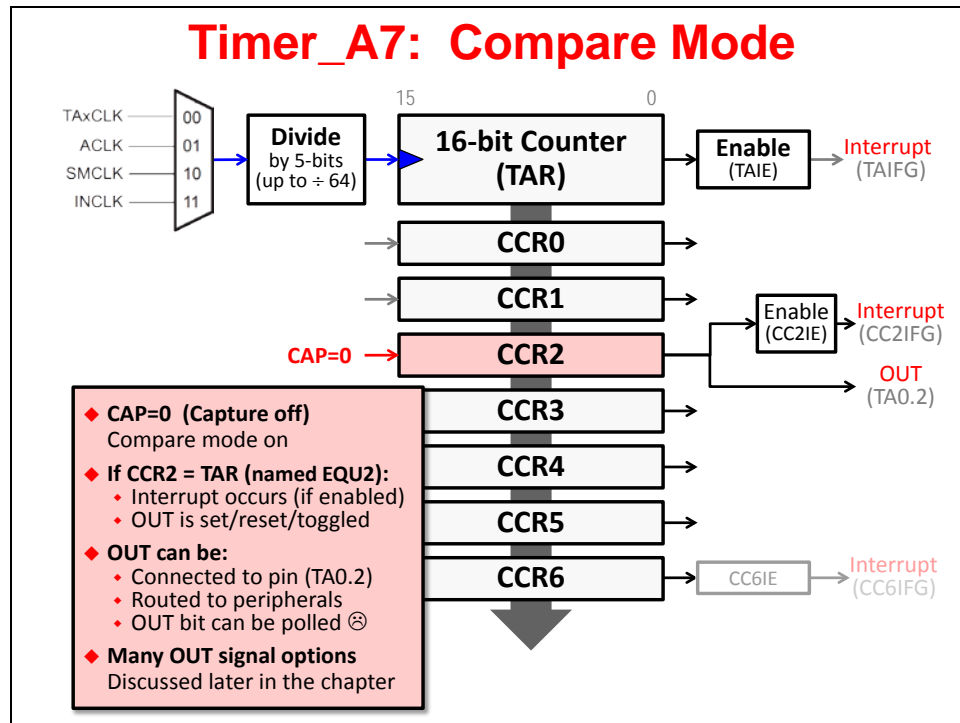
```
#include <driverlib.h>

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()



2b. Compare: TIMER_A_initCompare()



Summary of Timer Setup Code – Part 2

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_CCR0_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
    );
}
        
```

Output Modes

Timer CCR (Compare) Output Mode 01

- ◆ As discussed earlier, each CCR has it's own signal (e.g. TA0.1)
 - ◆ Input for capture (CCI)
 - ◆ Output for compare (OUT)

- ◆ For capture, the value in register bit CCRn.OUT is routed to TA0.n
- ◆ Value of OUT is affected by Output Mode (CCRn.OUTMOD) as described over the next few slides
- ◆ Note: If OUTMOD=0, then OUT (and hence the signal) is under software control

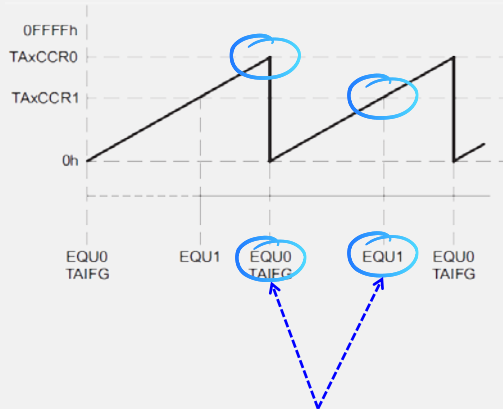


Note: Interrupts don't vary with OUTMOD, only the OUTPUT signal changes

Output Mode 1

- ◆ OUTMOD = 01 is called "Set"
- ◆ This means that OUT (e.g.TA0.1) is set on EQU1
- ◆ That is, whenever TAR=CCR1

EQU: When TAR = CCR



- ◆ Nomenclature used in MSP430 User's Guide
- ◆ EQU0 and EQU1 are names for when CCR0 and CCR1 compare events occur (e.g. CCR1 = TAR)
- ◆ Similar EQU_n events exist for each CCR register
- ◆ TAIFG is the generic timer interrupt whenever the count (in TAR) goes to zero

Timer CCR (Compare) Output Mode 02

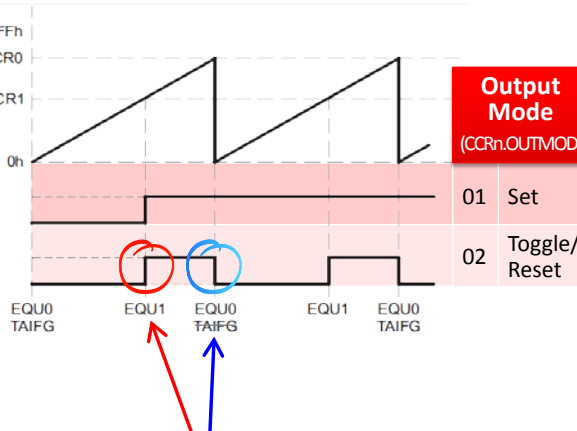
- ◆ **OUT is actually affected by two events:**

- ◆ EQU_n : when TAR=CCR_n
- ◆ EQU0 : when TAR=CCRO

- ◆ In other words, the two events are CCR_nIFG and CCR0IFG, respectively

- ◆ Output Mode 02 is called:

Toggle	/	Reset
Toggles OUT on EQU _n		Resets OUT on CCR0



- ◆ As stated earlier, CCR0 is special. It affects all other CCR compare outputs in this same way.

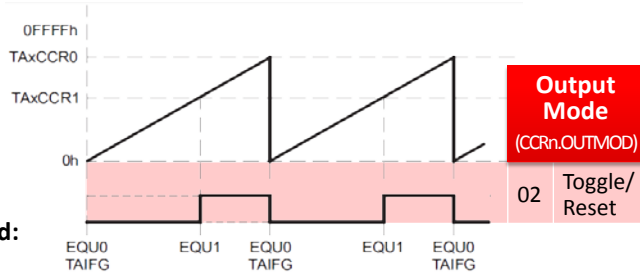
- ◆ **Note:** In this example, EQU0 and TAIFG happen at the same time; but TAIFG does not affect OUT.

Output Mode 2

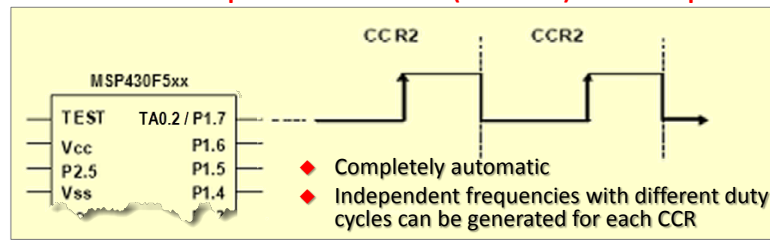
- ◆ OUTMOD = 02 is called "Toggle/Reset"
- ◆ This means that OUT (e.g. TA0.1) is **Toggled** upon EQU1
- ◆ And **Reset** on EQU0 (i.e. CCR0 match)

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

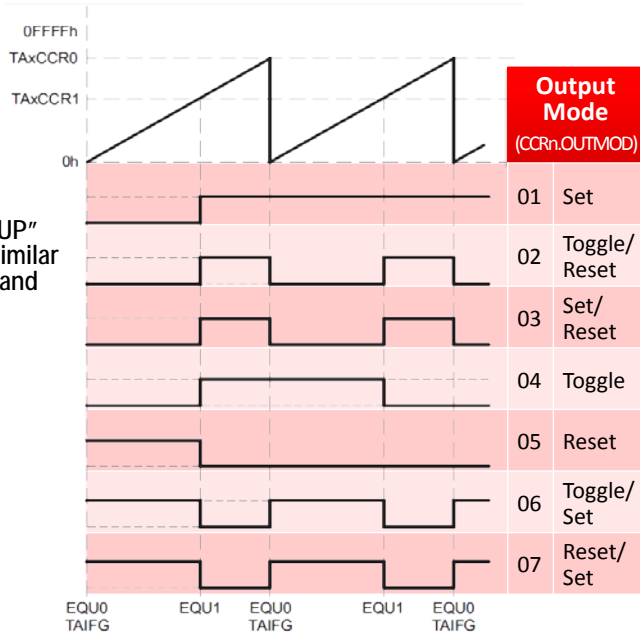


Here's an example of routine TA0.2 (i.e. OUT2) to a GPIO pin:

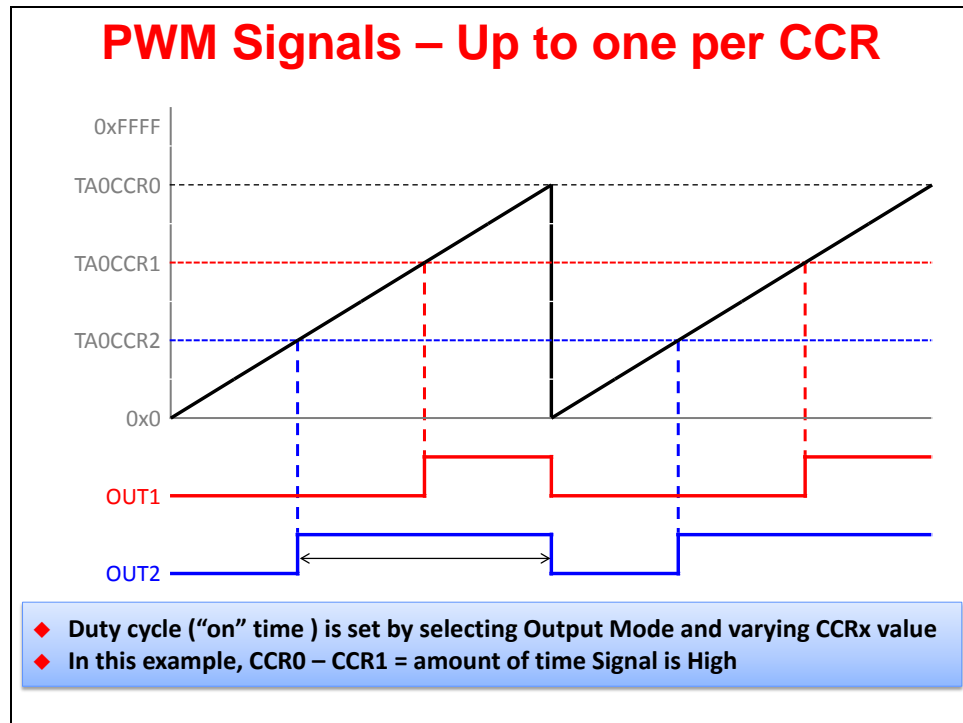


Capture "Output Modes" Summary

- ◆ Use different OUTMOD settings to create various signal patterns
- ◆ Output modes 2, 3, 6, and 7 are not useful for output unit 0 because EQU_n = EQU₀
- ◆ This summary is for the "UP" mode. User's Guide has similar diagrams for Continuous and Up/Down counter modes



PWM Signals



3. Clear Interrupt Flags and TIMER_A_startTimer()

Timer Code Ex. (Part 3 – Clear IFG's/Start)

!
1

@
2

3

```
#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_CCR0_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
}
```


4. Interrupt Code (Vector & ISR)

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

- ◆ 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 associated highest priority pending interrupt and clears its IFG bit

Timer Code Example (Part 4 – ISR's)

CCR0
ISR

\$
4

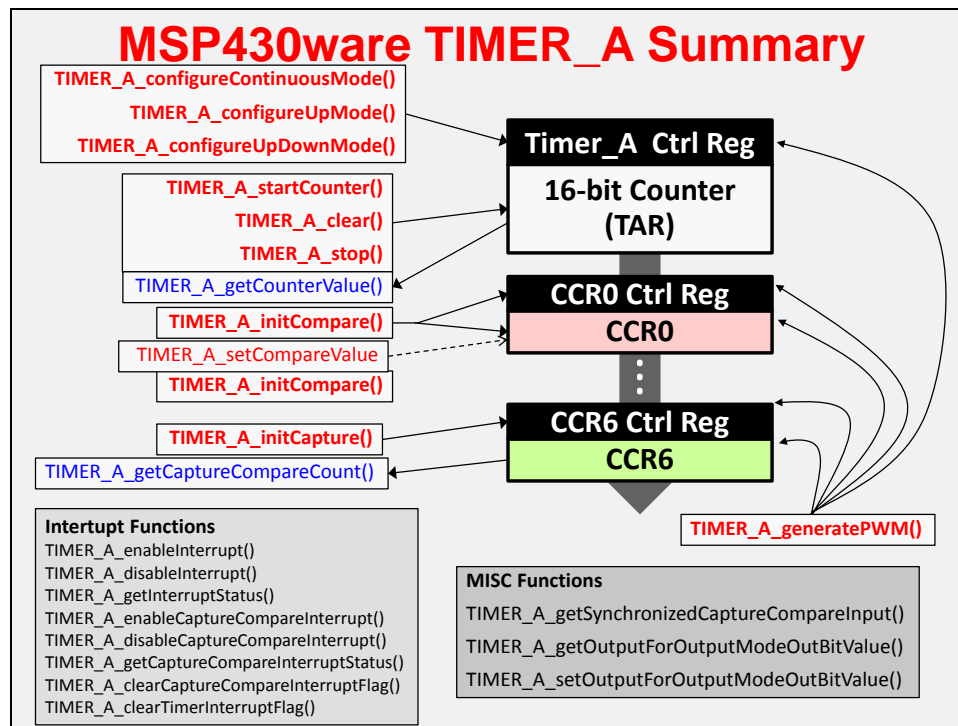
CCR2
and TAO
ISR's

```

#pragma vector=TIMER0_A0_VECTOR
__interrupt void myISR_TA0_CCR0(void) {
    GPIO_toggleOutputOnPin( ... );
}

#pragma vector=TIMER0_A1_VECTOR
__interrupt void myISR_TA0_Other(void) {
    switch(__even_in_range( TA0IV, 10 )) {
        case 0x00: break;           // None
        case 0x02: break;           // CCR1 IFG
        case 0x04: break;           // CCR2 IFG
        case 0x06: break;           // CCR3 IFG
        case 0x08: break;           // CCR4 IFG
        case 0x0A: break;           // CCR5 IFG
        case 0x0C: break;           // CCR6 IFG
        case 0x0E: break;           // TAOIFG
        default: _never_executed();
    }
}
                
```

TIMER_A API Summary



Differences between Timer's A and B

Similarities

Timer_A vs Timer_B

- ◆ **Timer_B's default functionality is identical to Timer_A**
- ◆ **Names are (almost) the same: TAR → TBR, TAOCTL → TBOCTL, 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 Timer_B devices 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 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

Which Timer Should I Use?

Pulse-width modulation: Use Timer_B, if available, otherwise Timer_A. Connect the load directly to an output of the timer so that it can be driven directly by hardware.

Less regular outputs: Connect directly to an output of Timer_A or B.

- ◆ Use the Up mode if the intervals between changes are always the same, as in many forms of communication.
- ◆ Continuous mode is easier if the intervals vary.

Inputs to be sampled at regular intervals: Connect directly to an input of Timer_A and use the Sampling mode (the Compare mode with the SCCI bit). This applies mainly to communications.

Inputs to be timed: Connect slow inputs directly to a Capture input of Timer_A or B. Fast signals should be connected to one of the timer clock inputs, such as TACLK or INCLK.

Periodic software interrupts:

- ◆ Try the **watchdog timer** if it's not used as watchdog, though, WDT has a limited set of interval periods. For longer intervals use ACLK, shorter use SMCLK.
- ◆ If WDT isn't suitable try **Timer_A or B**, which can produce almost any interval desired. Though, this may interfere with the use of their more advanced features.

Less regular software interrupts: Use Timer_A or B, preferably in the Continuous mode.

* Adapted from *MSP430 Microcontroller Basics* by John Davies (Newnes) (ISBN 978-0750682763)

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

Timers	6-21
<i>Lab 6 – Using Timer_A</i>	<i>6-23</i>
<i>Lab 6a – Simple Timer Interrupt</i>	<i>6-25</i>
Lab 6a Worksheet	6-25
File Management	6-29
Setup the Timer.....	6-30
Debug/Run	6-31
<i>(Extra Credit) Lab 6b – Timer using Up Mode</i>	<i>6-32</i>
Lab 6b Worksheet	6-32
File Management	6-35
Change the Timer Setup Code	6-35
Debug/Run	6-36
Archive and Close the Project.....	6-37
<i>(Extra Credit) Lab 6c – Timer using Up Mode</i>	<i>6-39</i>
Lab 6c Worksheet	6-39
File Management	6-42
Change the GPIO Setup	6-42
Change the Timer Setup Code	6-43
Debug/Run	6-44
<i>Chapter 6 Appendix</i>	<i>6-47</i>

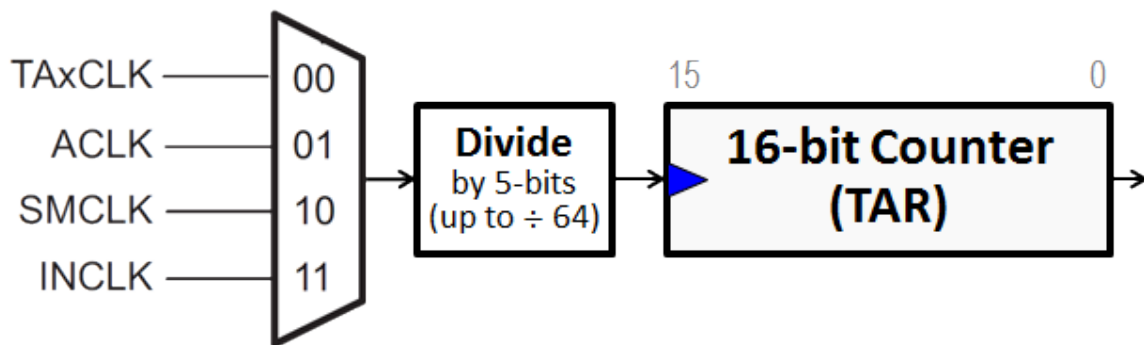
Lab 6a – Simple Timer Interrupt

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 `TimerA0` which generates 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 clock input and divider value will get our timer near 2 seconds?**

Clock input: _____

Divide value: _____

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
ACLK	32 KHz
SMCLK	8 MHz
MCLK	8 MHz

What clock did you choose in the previous step?

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. Write the `TIMER_A_configureContinuousMode()` function.

Where to get help for writing this function? We highly recommend the MSP430ware DriverLib users guide. (See docs folder inside MPS430ware's **driverlib** folder.) Another suggestion would be to examine the header file: (`timer_a.h`).

```
TIMER_A_configureContinuousMode(
    TIMER_A0_BASE, // Setup Timer A0
    _____, // Timer clock source
    _____, // Timer clock divider
    _____, // Enable interrupt on TAR counter rollover
    TIMER_A_DO_CLEAR // Clear TAR & previous divider state
);
```


5. 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 used in the previous question
- Clear the appropriate interrupt flag (IFG)
- Start the timer

```
// Clear the timer flag and start the timer

_____ ( TIMER_A0_BASE ); // Clear TA0IFG

_____ ( TIMER_A0_BASE, // Start timer
TIMER_A_CONTINUOUS_MODE ); // in Continuous mode
```

6. Change the following interrupt code to toggle the Green LED when TimerA0 rolls-over.

Here's the interrupt code that exists from our previous lab 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: On your Launchpad, what Port/Pin number does the Green LED use? _____

Please verify your answers before moving onto the lab exercise.

Notes:

File Management

1. **Verify that all projects (and files) in your workspace are closed.**

If some are open, we recommend closing them.

2. **Import previous solution for lab_05a_buttonInterrupt.**

Import the solution from the archive:

```
C:\msp430_workshop\<target>\solution\lab_05a_buttonInterrupt_solution.zip
```

3. **Rename the project to: lab_06a_timer**

4. **Delete old, unnecessary code.**

We won't be using the *pushbutton*, so you can delete that code from the `initGPIO()` function. Make sure, though, that the the LED setup code remains.

5. **Make sure both LED's are configured.**

Verify that both LED GPIO outputs are properly configured – adding the code, if necessary. That means, on the F5229 Launchpad, setting up both P1.0 and P4.7.

Also, we recommend initializing LEDs by turning them off.

6. **Build the project to verify no errors were introduced.**

Setup the Timer

In this part of Lab 6, we will be setting up TimerA0 in Continuous Mode.

7. Add a new `initTimers()` function to `main.c`.

This requires three steps:

a) In `main()`, add a call to `initTimers()`.

We recommend that you add the call below the `initClocks()` function call.

b) Add a prototype for the `initTimers()` function at the top of `main.c`.

c) Create an empty `initTimers()` function.

We recommend placing this below the `initGPIO()` function in `main.c`, though, you could really put it anywhere inside the file. We'll add the code to this function in the next step.

```
void initTimers(void) {  
    }  
}
```

d) Build the code to make sure there are no syntax errors.

8. Add the code needed to setup the timer to the `initTimers()` function.

You may remember that the Timer setup code consisted of three parts. In this exercise, though, we're only use parts ❶ and ❸, since the Continuous mode does not require us to setup a capture/compare register.

a) Configure the timer with a call to `TIMER_A_configureContinuousMode()`.

Please refer to the Lab Worksheet for assistance. (Step 4, Page 6-26).

b) Add code to clear the interrupt flag, enable the interrupt, and start the timer.

Please refer to the Lab Worksheet for assistance. (Step 5, Page 6-27).

c) Build the code to make sure there are not syntax errors.

9. Modify the ISR to handle the TimerA0 interrupt, rather than GPIO port interrupt.

Please refer to the Lab Worksheet for assistance. (Step 6, Page 6-27).

10. 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.)

11. Build the code. Verify there are no syntax errors.

Debug/Run

12. Launch the debugger.

13. Set a breakpoint inside the ISR.

We found it worked well to set a breakpoint on the 'switch' statement.

14. Run your code.

If all worked well, when the counter rolled over to zero, an interrupt should have occurred ... which should have resulted in the processor halting at a breakpoint inside the ISR.

15. If the breakpoint occurred, skip to the next step ...

If you did not reach the breakpoint inside your ISR, here's a few thing to look for:

- Is the interrupt flag bit (IFG) set?
- Is the interrupt enable bit (IE) set?
- Are interrupts enabled globally?

16. 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 TA0IV should clear the flag, so the likelihood of this problem occurring is small, but for one reason or another, the problem occurs more often than you might expect.

17. 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 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.

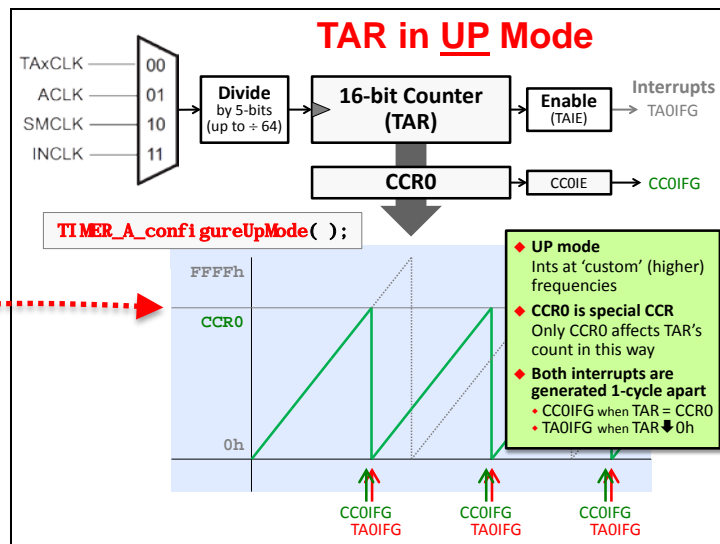
The goal of this part of the lab is to:

```
// TimerA0 in Up mode using ACLK
// Toggle Green LED every 1/2 second using TAOIFG
// Toggle Red LED every 1/2 second using CCR0IFG
```

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 1/2 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_A0_BASE,                // Setup Timer A0  
    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_A0_BASE );           // Clear TA0IFG

_____

TIMER_A_startCounter( TIMER_A0_BASE,                       // Start timer
                     TIMER_A_CONTINUOUS_MODE );           // in _____ mode
```

4. Add a new ISR to toggle the Red LED when the CCR0 interrupt fires.

On your Launchpad, what Port/Pin number does the Red LED use? _____

Here we've given you a bit of code to get you started:

```
#pragma vector= _____
__interrupt void ccr0_ISR (void)
{
    // Toggle the Red LED 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.**
 - a) In Project Explorer, right-click on the lab_06a_timer 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_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. **Make sure both LED's are configured.**

We only used one in the last lab, make sure that both still are setup in the code, as we'll be using both of them in this exercise.
5. **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.

6. **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. (Step 2, Page 6-33).
7. **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. (Step 3, Page 6-34).
8. **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. (Step 4, Page 6-34).
9. **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.)
10. **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.

11. Launch the debugger and set a breakpoint inside the both ISR's.

12. 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? _____

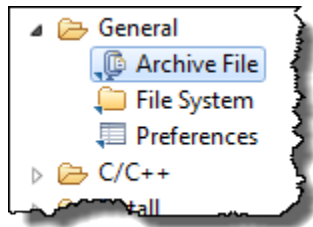
13. Remove the breakpoints and let the code run. Do both LED's toggle?

Archive and Close the Project

Thus far in this workshop, we have imported many projects from archives ... but we haven't asked you to make an archive. It's not hard, as you'll find out.

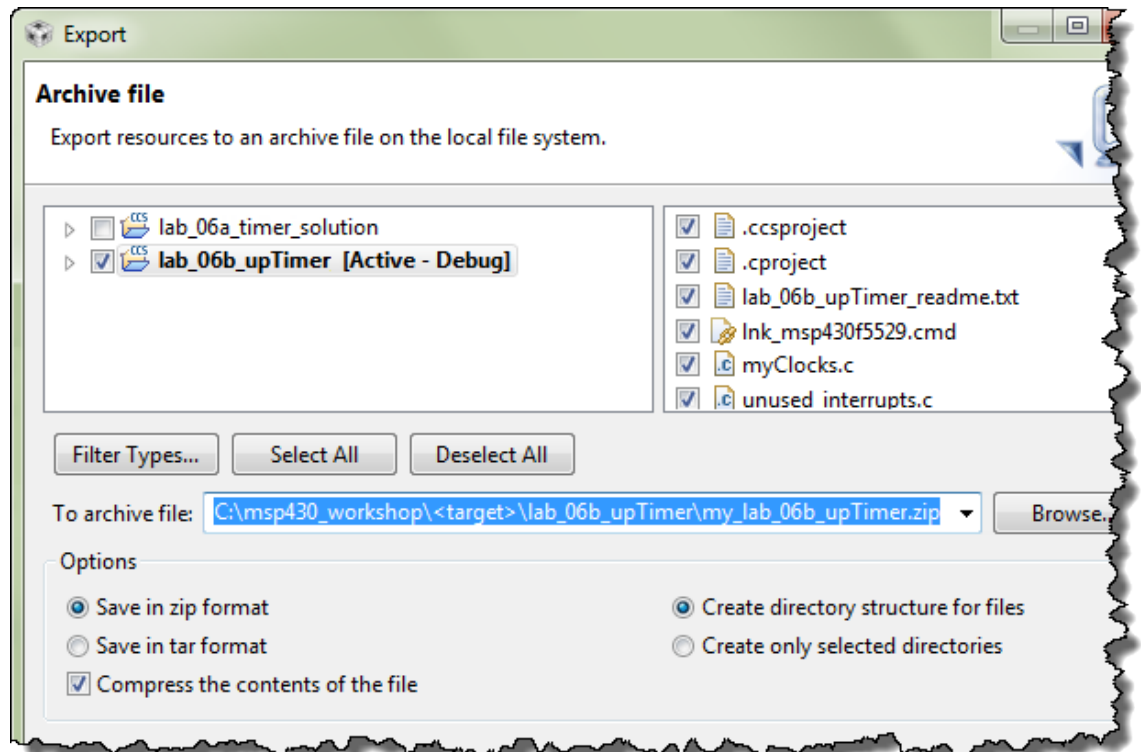
14. Export your project to the lab's file folder.

- 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



Notes:

(Extra Credit) Lab 6c – Timer using Up Mode

This lab is a minor adaptation of the code from the previous exercise. The main difference is that we'll connect the output of Timer_A0 CCR2 (TA0.2) directly to a GPIO pin. (*Remember, CCR0 is used for resetting TAR back to 0; we are still using Up mode in this lab. But, we CCR0 already in use, we chose 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 - here's an excerpt from the lab solution:

```
// When running this lab exercise, you will need to pull the JP8 jumper and
// use a jumper wire to connect signal from ____ (on boosterpack pinouts) to
// JP8.2 (bottom pin) of LED1 jumper ... this lets the TA0.2 signal drive the
// RED LED directly (without having to use interrupts)
```

Lab 6c Worksheet

1. Figure out which boosterpack pin to drive with the timer's output (i.e. TA0.2).

We want to choose a boosterpack pin, as this will make it easy for us to jumper the signal over to the Red LED. Which boosterpack pin can support the TA0.2 output?

There are really two parts to this question:

- a) What GPIO output is TA0.2 combined with?

Hint: *There are a couple places in the datasheet to find this information. We recommend opening your device's datasheet and searching for "TA0.2".*

GPIO pin: _____

- b) Next, what boosterpack pin is this GPIO connected to?

This is easy to read directly from the Launchpad. Scan the silkscreened labels next to the boosterpack pins. (If you're getting a little older, you may need a magnifying glass to answer this question...or zoom in while viewing the Launchpad's photo in this document.)

Boosterpack pin: _____

2. Write the function to set this Pin/Port to be used as a timer pin (as opposed to an output pin).

GPIO_setAs_____ (_____ , _____);

3. Which Port/Pin drives the Red LED?

Port _____ Pin _____

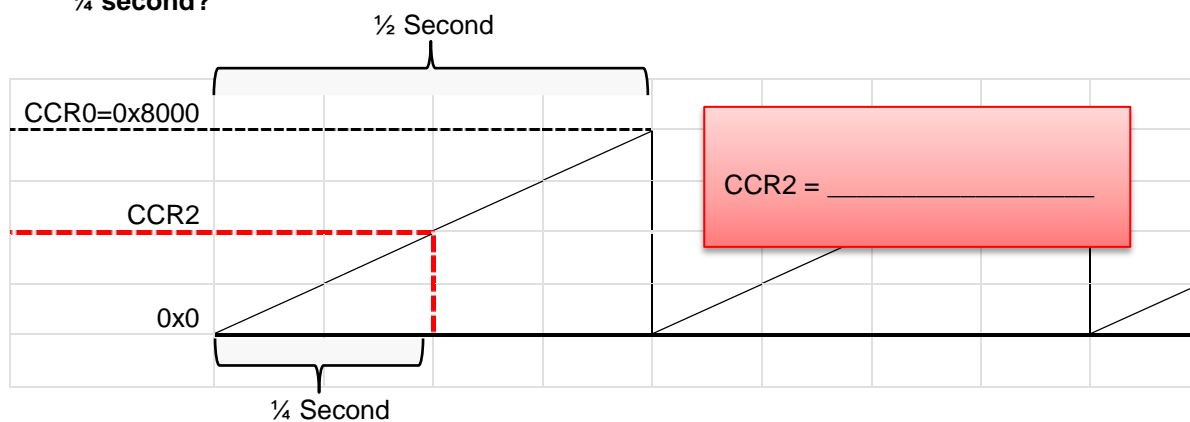
4. Modify the `TIMER_A_configureUpMode()` function?

Here is the code we wrote for the previous part of the 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_A0_BASE, // Setup Timer A0  
    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  
);
```

5. What 'compare' value does CCR2 need to equal in order to toggle the output signal at 1/4 second?



6. 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_A0_BASE, // Setup Timer A0
    _____, // 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 ¼ second
);
```

7. Compare your previous code to that below.

What did we change? _____

```
#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: // 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 be skipped, and thus, the breakpoint never reached?

8. 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 peripheral (i.e. timer) functionality.**

Please refer to the Lab6c Worksheet for assistance. (Step 2, Page 6-39).

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 4, Page 6-40).

7. **Add a call to the TIMER_A function that configures CCR2.**

Please refer to the Lab Worksheet for assistance. (Step 6, Page 6-41).

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, not interrupt is required where we need to 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... so a 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 7, Page 6-41).

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 is set.)

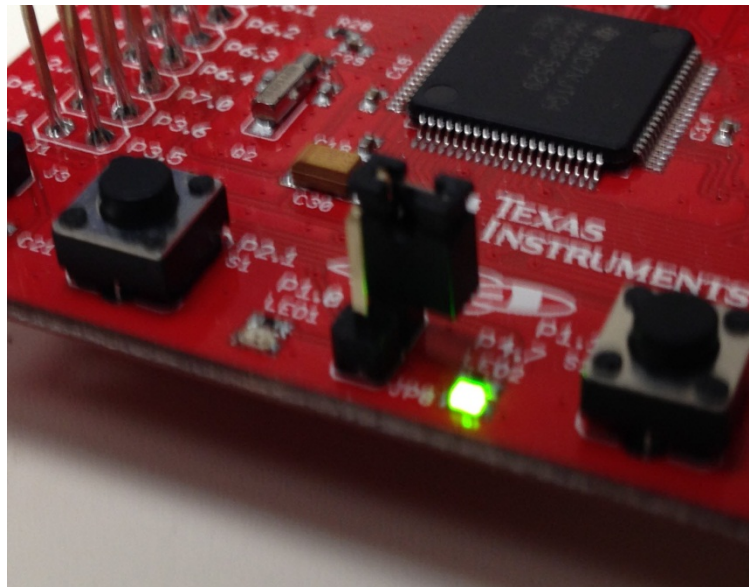
```
97 //*****
98 // Interrupt Service Routines
99 //*****
100 #pragma vector=TIMER0_A0_VECTOR
101 __interrupt void ccr0_ISR (void)
102 {
103     // 4. Timer ISR and vector
104
105     // Toggle the Red LED on/off
106     GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
107 }
108
109 #pragma vector=TIMER0_A1_VECTOR
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: break;           // CCR2 IFG
118         case 6: break;           // CCR3 IFG
119         case 8: break;           // CCR4 IFG
120         case 10: break;          // CCR5 IFG
121         case 12: break;          // CCR6 IFG
122         case 14:                 // TAR overflow
123             // Toggle the Green LED on/off
124             GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN7 );
125             break;
126         default: _never_executed();
127     }
128 }
```

12. Remove the breakpoints and let the code run. Do both LED's toggle?

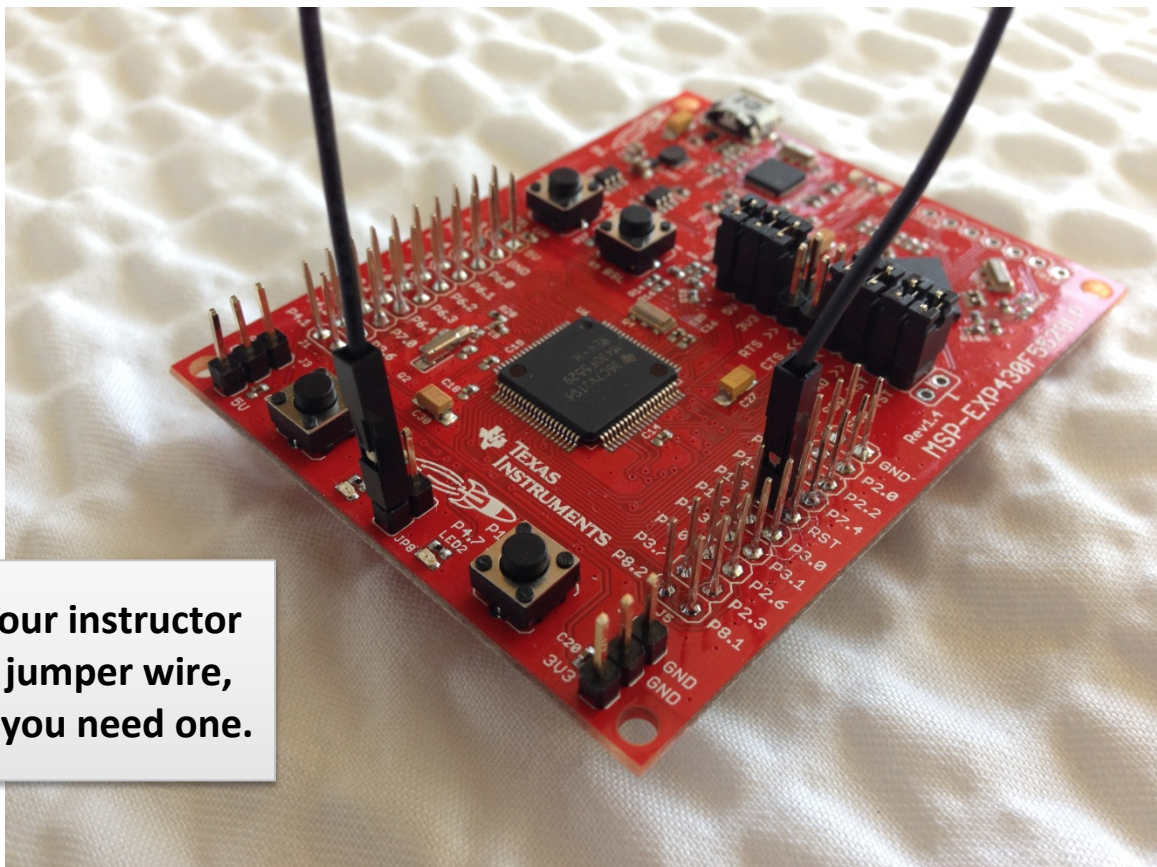
Why doesn't the **Red** LED toggle? _____

13. Add the jumper wire to your board to connect the timer output to the LED.

- a) Remove the jumper (JP8) that connects the Red LED to P1.0.
(We recommend reconnecting it to the top pin of the jumper so that you don't lose it.)



- b) On the 'F5529 Launchpad, connect P1.3 (fifth pin down, right-side of board, inside row of pins) to the bottom of the LED jumper (JP8) using the jumper wire.



**Ask your instructor
for a jumper wire,
when you need one.**

14. Run your code.

Hopefully both LED's are not blinking. The Red LED should toggle first, then the Green LED.

Do they both blink at the same rate? _____

Why is that? _____

15. Terminate the debugger and go back to your `main.c` file.

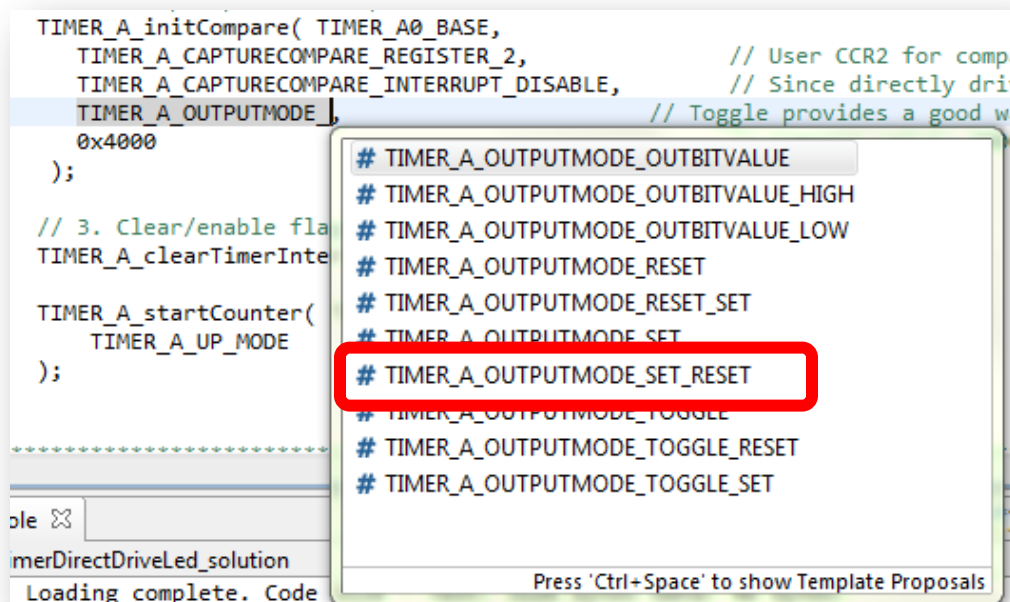
16. Modify one parameter of the function that configures CCR2, changing it to use the mode:

```
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
```



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.

Chapter 6 Appendix

Lab 6a Worksheet

Goal: Write a function setting up TimerA0 which generates an interrupt every two seconds.

1. How many clock cycles does it take for the 16-bit TimerA0 to ‘rollover’? (Hint: 16-bit timer)

$$2^{16} = 64K$$

2. If our goal is to generate a two second interrupt rate, what clock input and divider value will get our timer near 2 seconds?

Clock input: ACLK - which is configured for 32KHz / sec

Divide value: Clock input divide of 1 should work (as we'll see in next)

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
ACLK	32 KHz
SMCLK	8 MHz
MCLK	8 MHz

What clock did you choose in the previous step?

Timer Clock Source: ACLK

Clock Frequency = 32KHz / sec cycles/second

Timer Frequency = $\frac{32\text{KHz / sec}}{\text{clock frequency}} \div \frac{1}{\text{timer clock divider}} = \frac{32\text{KHz / sec}}$

Timer Output = $\frac{32\text{KHz / sec}}{\text{timer frequency}} \div \frac{64\text{K}}{\text{counts for timer to rollover}} = \frac{1}{2} \text{ sec}$

Lab 6a Worksheet

4. Write the `TIMER_A_configureContinuousMode()` function.

Where to get help for 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`).

```
TIMER_A_configureContinuousMode(
    TIMER_A0_BASE,                // Setup Timer A0
    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
);
```

Lab 6a Worksheet

5. 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 used in the previous question
- Clear the appropriate interrupt flag (IFG)
- Start the timer

```
// Clear the timer flag and start the timer
TIMER_A_clearTimerInterruptFlag ( TIMER_A0_BASE ); // Clear TAOIFG
TIMER_A_startCounter ( TIMER_A0_BASE, // Start timer
TIMER_A_CONTINUOUS_MODE ); // in Continuous mode
```

Lab 6a Worksheet

6. Change the following interrupt code to toggle the Green LED when TimerA0 rolls-over.

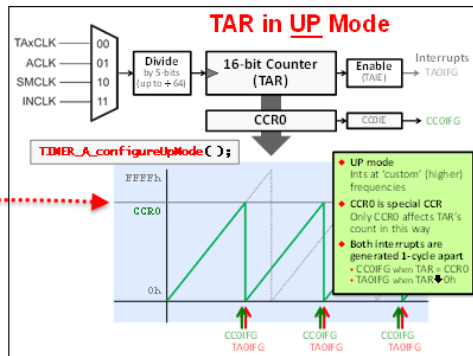
Here's the interrupt code that exists from our previous lab exercise, change it as needed:

```
__interrupt void pushbutton_ISR (void)
#pragma vector=PORT1_VECTOR TIMER0_A1_VECTOR
void timer0_ISR TAOIV 14
{
    switch(__even_in_range( Div , 16 )) {
        case 0: break; // No interrupt
        case 2: break;
        case 4:
            GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
            break;
        case 6: break;
        case 8: break;
        case 10: break;
        case 12: break;
        case 14: GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN7 );
            break;
case 16: break;
        default: _never_executed();
    }
}
```

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?

$$\text{Timer Frequency} = \frac{32 \text{ KHz}}{\text{clock frequency}} \div \frac{1}{\text{timer clock divider}} = 32 \text{ KHz}$$

$$\text{Timer Output} = \frac{32 \text{ KHz}}{\text{timer frequency}} \div \frac{0x8000}{\text{timer period (i.e. CCR0 value)}} = \frac{1}{2} \text{ SECOND}$$

Lab 6b Worksheet

2. Complete the `TIMER_A_configureUpMode()` function?

This function will replace the `TIMER_A_configureContinuousMode()` call we made in our previous lab exercise.

```
TIMER_A_configureUpMode(
    TIMER_A0_BASE,                // Setup Timer A0
    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. 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_A0_BASE );           // Clear TA0IFG
TIMER_A_clearCaptureCompareInterruptFlag( TIMER_A0_BASE,
  TIMER_A_CAPTURECOMPARE_REGISTER_0 );
_____

TIMER_A_startCounter( TIMER_A0_BASE,                       // Start timer
  TIMER_A_CONTINUOUS_MODE );                               // in UP mode
UP
```

Lab 6b Worksheet

4. Add a new ISR to toggle the Red LED when the CCR0 interrupt fires.

On your Launchpad, what Port/Pin number does the Red LED use? P1.0

Here we've given you a bit of code to get you started:

```
#pragma vector= TIMER0_A0_VECTOR
__interrupt void ccr0_ISR (void)
{
    // Toggle the Red LED on/off

    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
}
```

Lab 6c Worksheet

- Figure out which boosterpack pin to drive with the timer's

a) What GPIO output is TA0.2 combined with?


Hint: There are a couple places in the datasheet to find recommend opening your device's datasheet and

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 type="checkbox"/>	24
P1.4/TA0.3	<input type="checkbox"/>	25
P1.5/TA0.4	<input type="checkbox"/>	26

GPIO pin: P1.3

b) Next, what boosterpack pin is this GPIO connected to?

Boosterpack pin: see photo


- Write the function to set this Pin/Port to be used as a timer output.

```
GPIO_setAsPeripheralModuleFunctionOutputPin ( GPIO_PORT_P1, GPIO_PIN3 );
```
- Which Port/Pin drives the Red LED?

Port 1 Pin 0

Lab 6c Worksheet

- Modify the `TIMER_A_configureUpMode()` function?

Here is the code we wrote for the previous part of the 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.

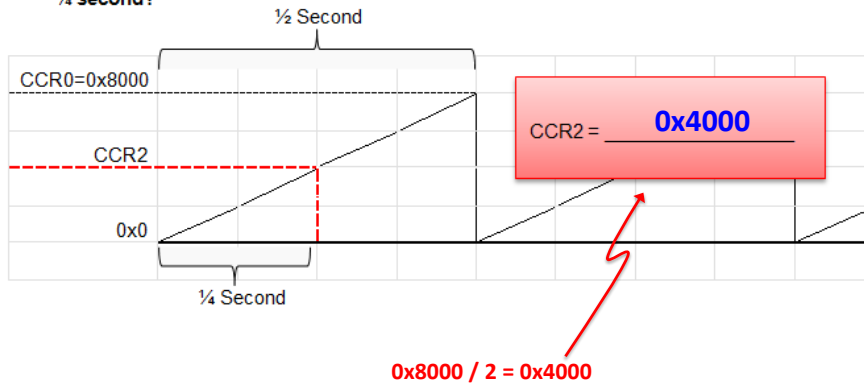
Mark of the code below to disable the interrupt. (We'll bet you can make this change without even looking at the API documentation. Intuitive code is one of the benefits of using DriverLib!)

```
TIMER_A_configureUpMode(
    TIMER_A0_BASE,                // Setup Timer A0
    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_DISABLE, // Enable CCR0 compare interrupt
    TIMER_A_DO_CLEAR              // Clear TAR & previous divider state
);
```

We changed 'ENABLE' to 'DISABLE'

Lab 6c Worksheet

5. What 'compare' value does CCR2 need to equal in order to toggle the output signal at ¼ second?



Lab 6c Worksheet

6. Add a new function call to setup Capture and Compare Register 2 (CCR2). This should be added to `initTimers()`.

```
TIMER_A_initCompare _____ (
    TIMER_A0_BASE, // Setup Timer A0
    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 ¼ second
);
```

Lab 6c Worksheet

7. 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
                GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN7 );
                break;
        default:  _never_executed();
    }
}
```

Lab 6c Worksheet

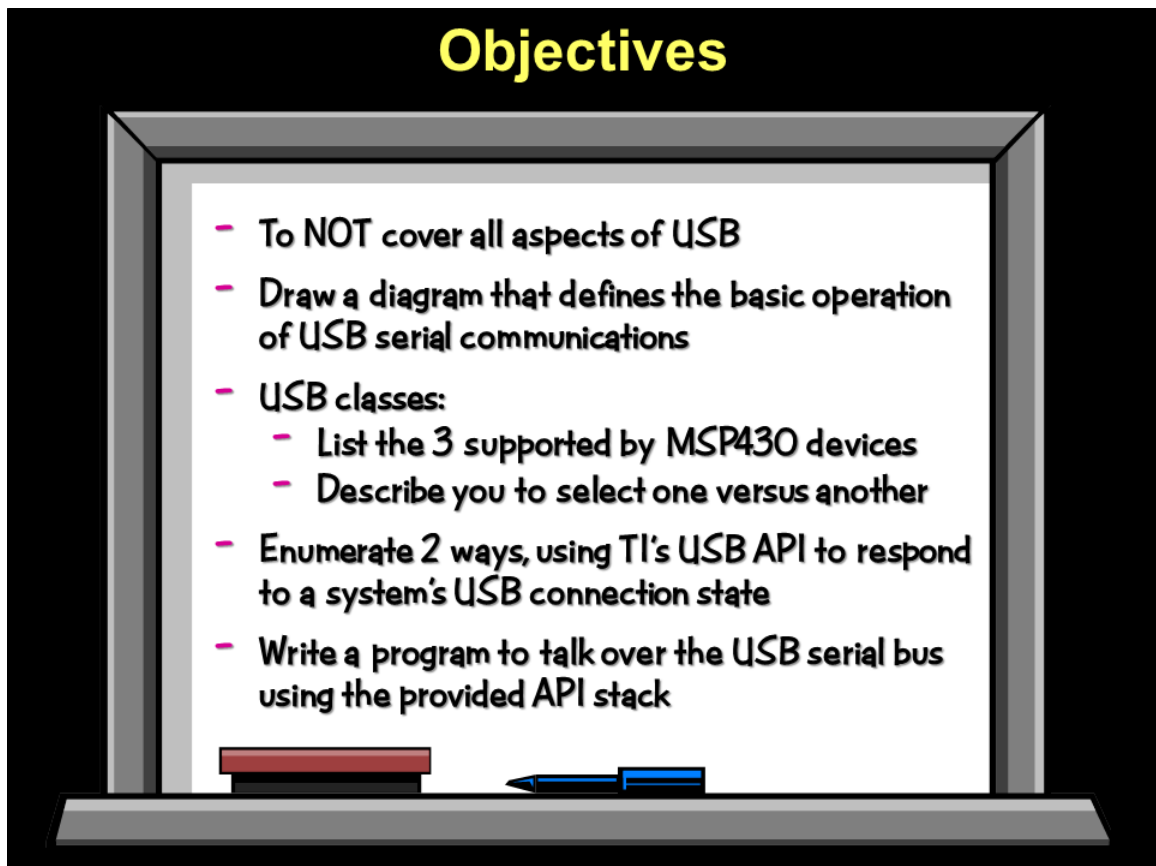
8. Why is better to toggle the LED directly from the timer, as opposed to using an interrupt (as we've done in the previous lab exercises)?

- ◆ **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!

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.

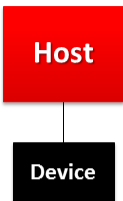
Learning Objectives



Chapter Topics

USB Devices	7-1
<i>Introduction</i>	<i>7-1</i>
<i>What is USB?</i>	<i>7-3</i>
<i>MSP430's USB Support.....</i>	<i>7-5</i>
<i>How USB Works</i>	<i>7-11</i>
<i>Descriptions and Classes.....</i>	<i>7-14</i>
<i>Quick Overview of MSP430's USB Stack.....</i>	<i>7-19</i>
<i>ABC's of USB.....</i>	<i>7-21</i>
A. Plan Your System	7-21
B. Connect & Enumerate	7-22
C. Managing my App & Transferring Data	7-24
<i>Final Thoughts</i>	<i>7-27</i>
<i>Lab Exercise</i>	<i>7-29</i>

What is USB?

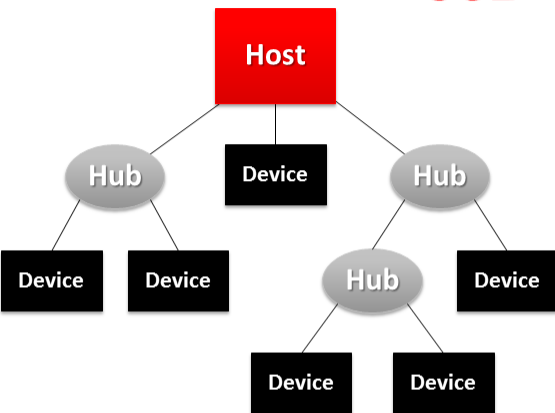


```

graph TD
    Host[Host] --- Device[Device]
            
```

USB

- ◆ USB is a serial connection, hence the name Universal Serial Bus
- ◆ It's a master/slave bus architecture where Host initiates all transfers
- ◆ Consists of a single Host along with 1 or more devices



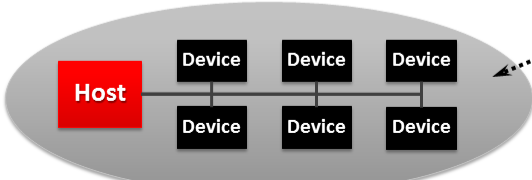
```

graph TD
    Host[Host] --- H1((Hub))
    Host --- D1[Device]
    Host --- H2((Hub))
    H1 --- D2[Device]
    H1 --- D3[Device]
    H2 --- H3((Hub))
    H2 --- D4[Device]
    H3 --- D5[Device]
    H3 --- D6[Device]
            
```

USB

- ◆ USB is a serial connection, hence the name Universal Serial Bus
- ◆ It's a master/slave bus architecture where Host initiates all transfers
- ◆ Consists of a single Host along with 1 or more devices
- ◆ Up to 127 devices can be connected to host; directly or in a star arrangement (using hubs)
- ◆ USB devices identified by unique Vendor and Product ID (VID/PID)
- ◆ Host keeps track of topology such that it knows what devices are gone when a hub is disconnected

◆ From a "device" perspective, it looks like a single, addressable bus



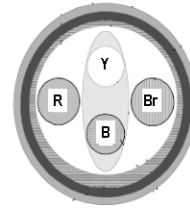
```

graph LR
    Host[Host] --- D1[Device]
    Host --- D2[Device]
    Host --- D3[Device]
    Host --- D4[Device]
    Host --- D5[Device]
    Host --- D6[Device]
            
```

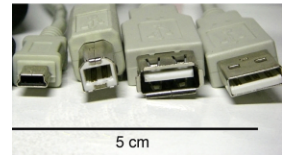
USB ... Physical Layer

◆ 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

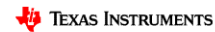
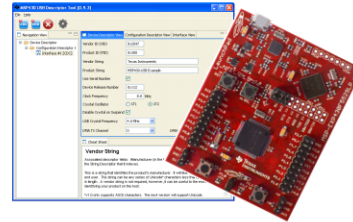


F6xx

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

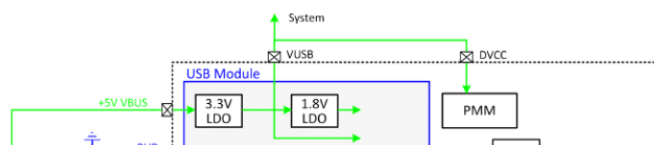
MSP430 USB Module

2.3 MSP430 USB Module

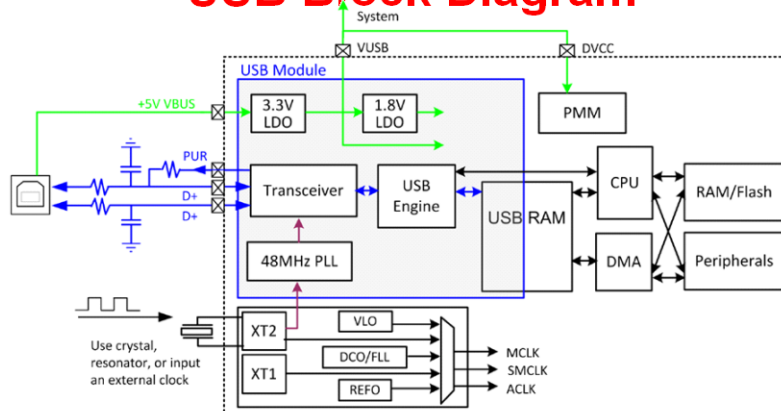
Features of the MSP430 USB module are as follows:

- **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.

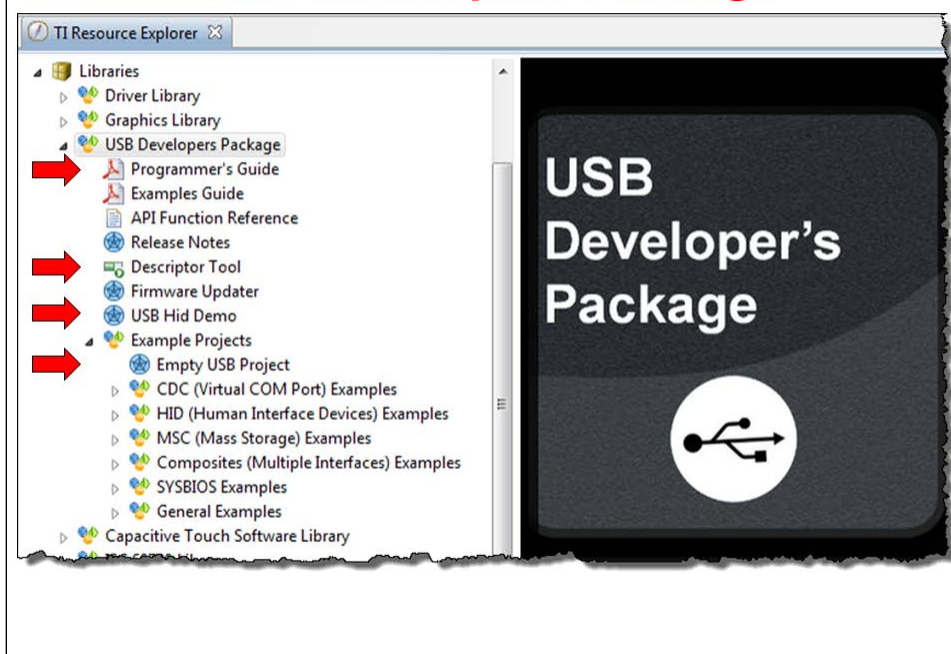
TEXAS INSTRUMENTS

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**

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 PID'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

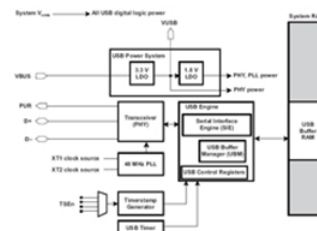
MSP430 + USB

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

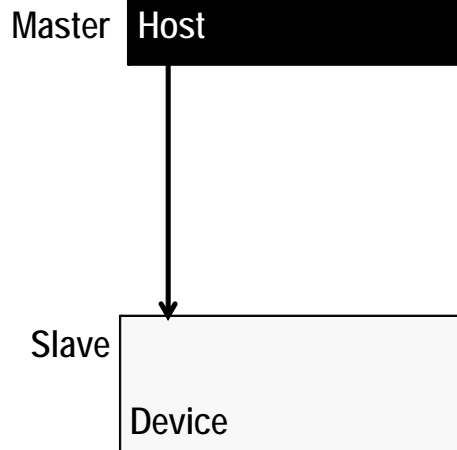
- > TI E2E Community
- > Contact Technical Support
- > MSP430 Discussion Group
- > Third-Party Network

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>

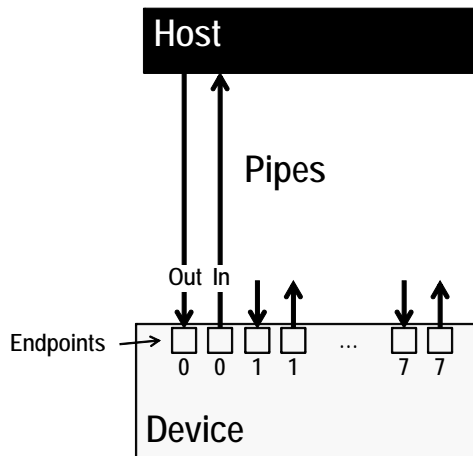
How USB Works

Logical Connection Between Host & Device

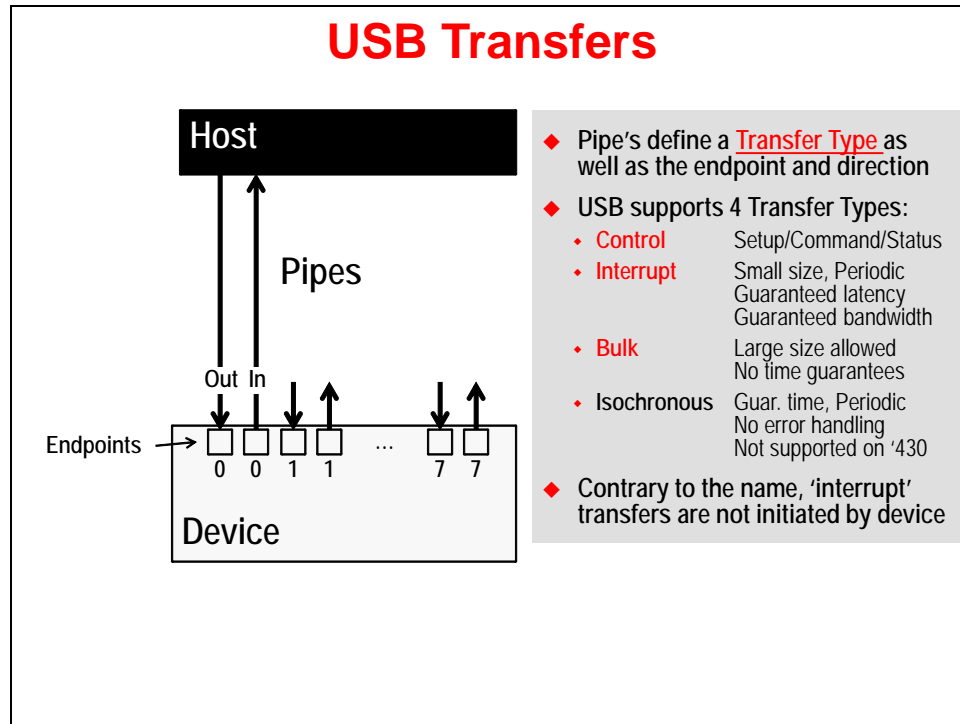


- ◆ Communication takes place between the host and device
- ◆ Host controls ALL communication
- ◆ Device is addressable (assigned by host)

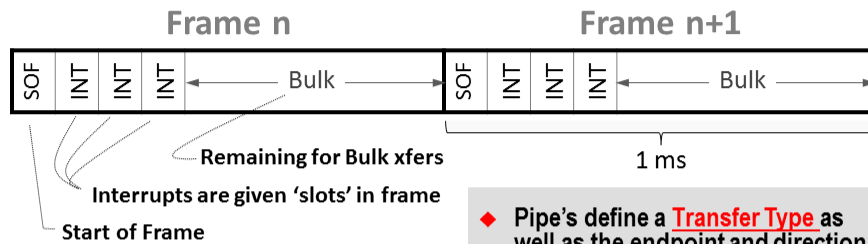
MSP430 Supports 8 Ins/Outs



- ◆ Host/Device communication occurs thru a Pipe
- ◆ The host sets up pipe connections to one or more device "endpoints"
- ◆ An endpoint is essentially a buffer in the device
- ◆ Pipes/Endpoints are unidirectional
- ◆ In/Out is from the Host perspective
- ◆ Endpoints are enumerated (from 0)
- ◆ Endpoint 0 is special case – all USB devices must have EP0, which is used for setup and control
- ◆ MSP430 Endpoints:
 - ◆ 16 endpoints (8 in, 8 out)
 - ◆ Each has 64 byte buffer



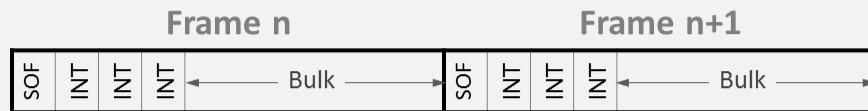
How is Bandwidth Guaranteed?



- ◆ 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

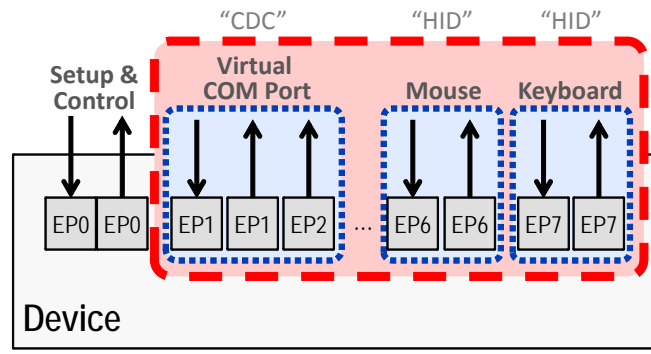
Device Descriptors

Device (example shown here is 'composite' device with multiple I/F's)

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

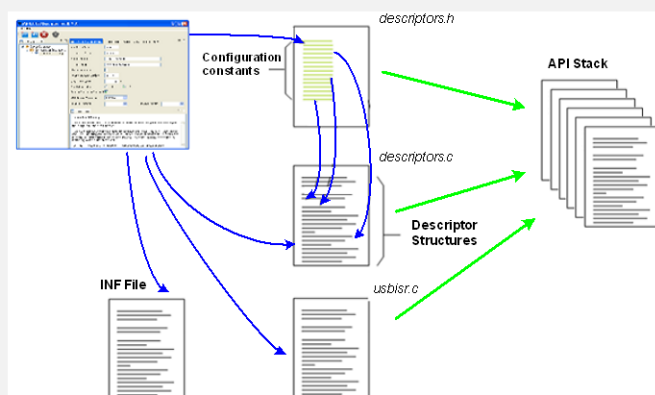


MSP430 USB Descriptor Tool

◆ 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)

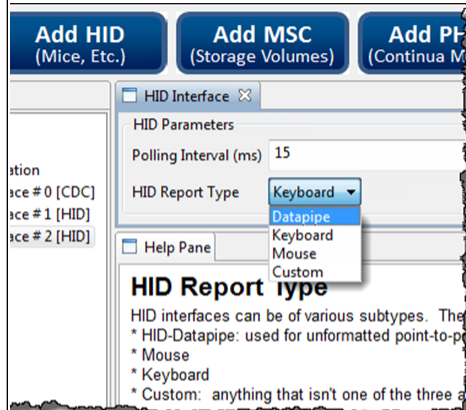


Communications Data Class (CDC)



- ◆ 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)

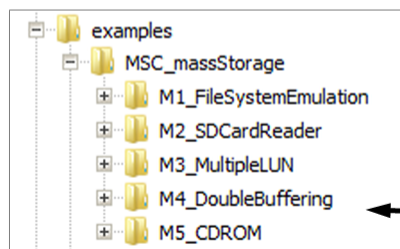
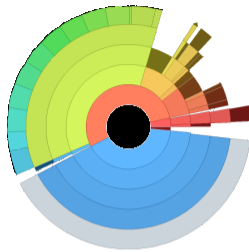


Datapipe mode allows the benefits of HID without some of its downsides

- ◆ **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)

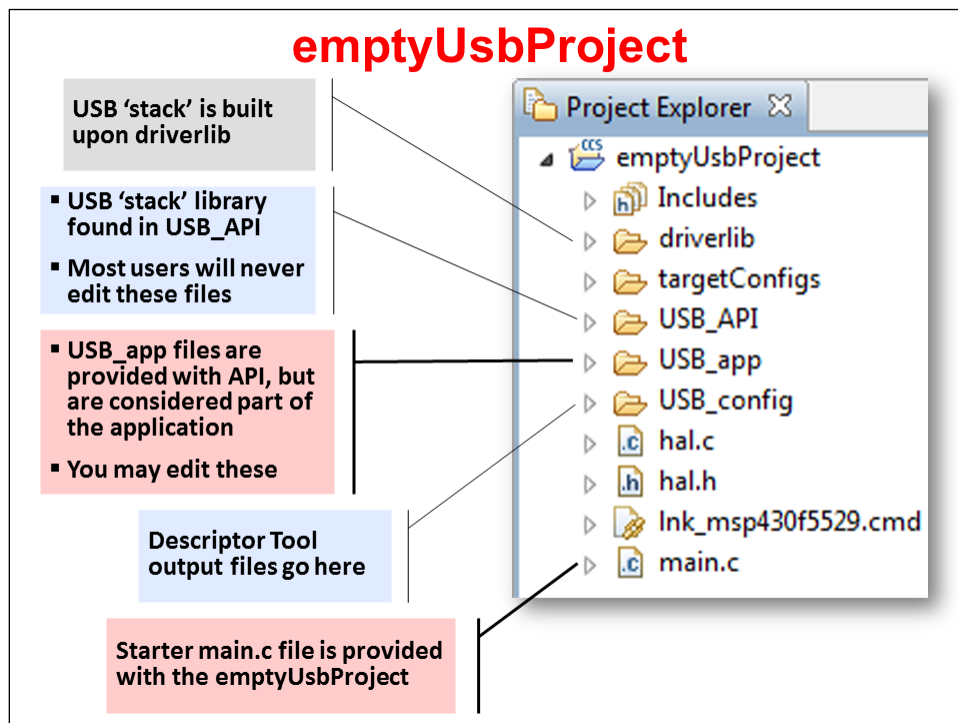
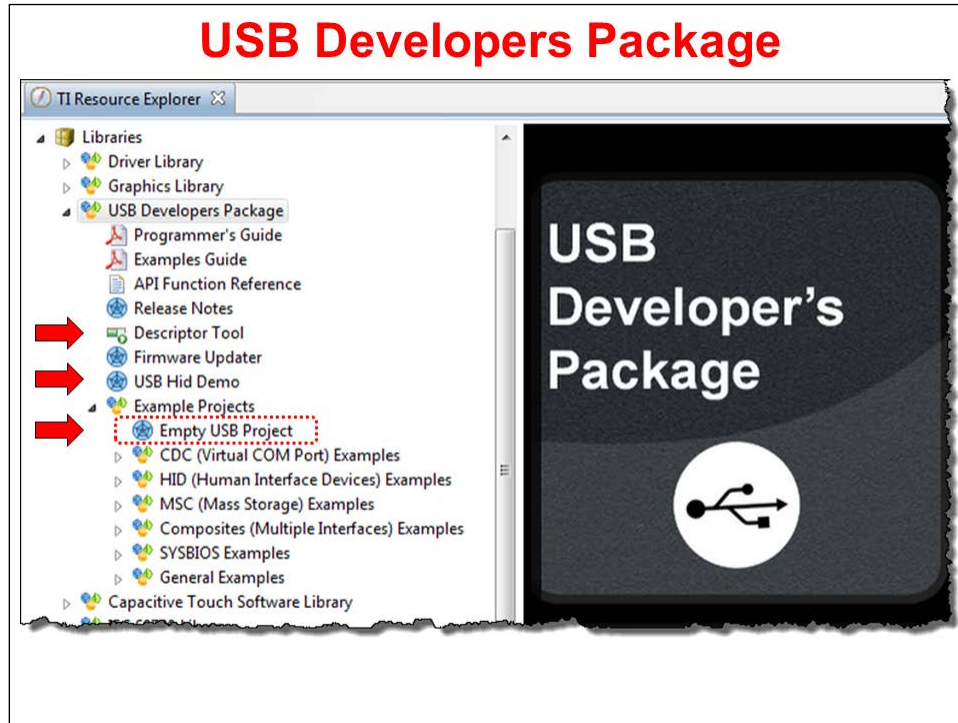


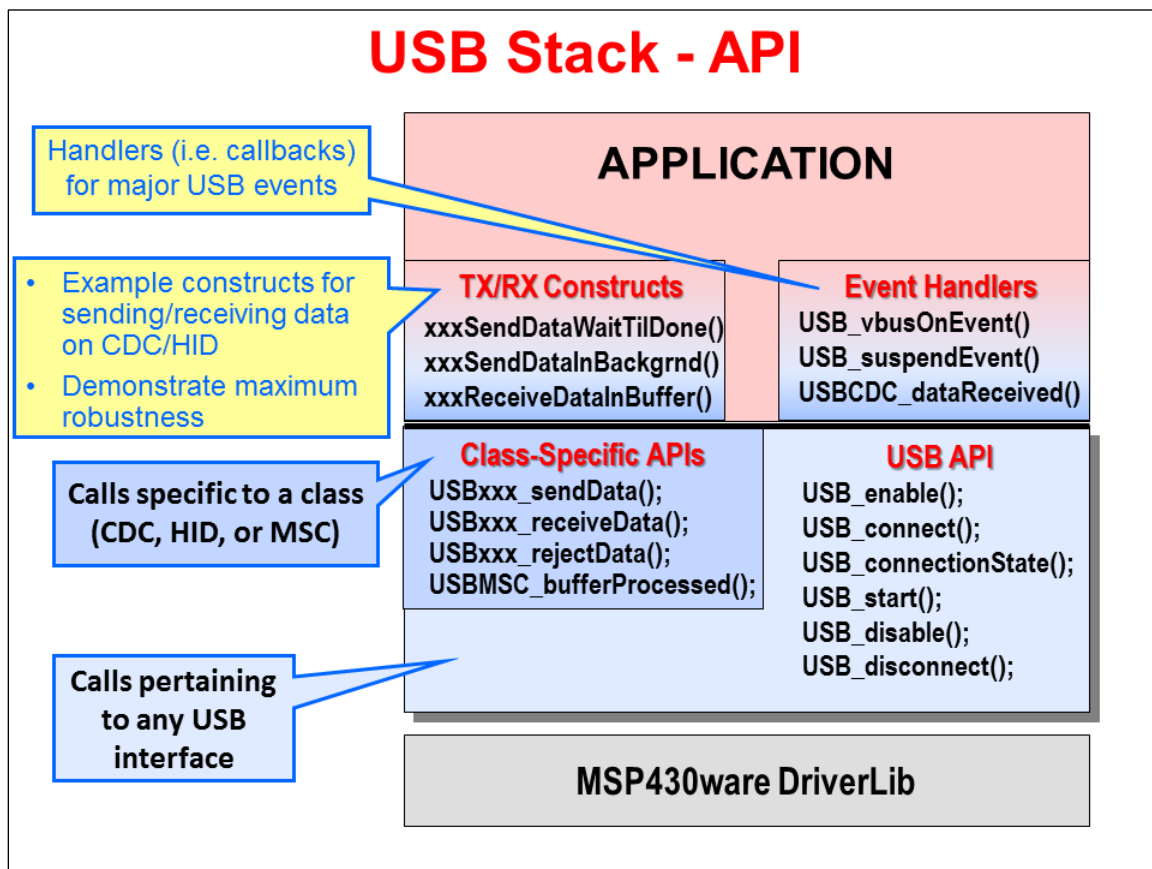
- ◆ Allows easy creation of a USB storage device
- ◆ No RTOS required
 - ◆ But can easily be ported to one
 - ◆ TI-RTOS (coming soon for MSP430) will provide a port with examples
- ◆ USB Developers Package includes a port of the open-source FAT file system (FatFS)
 - ◆ FatFS is provided as an example
 - ◆ 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





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

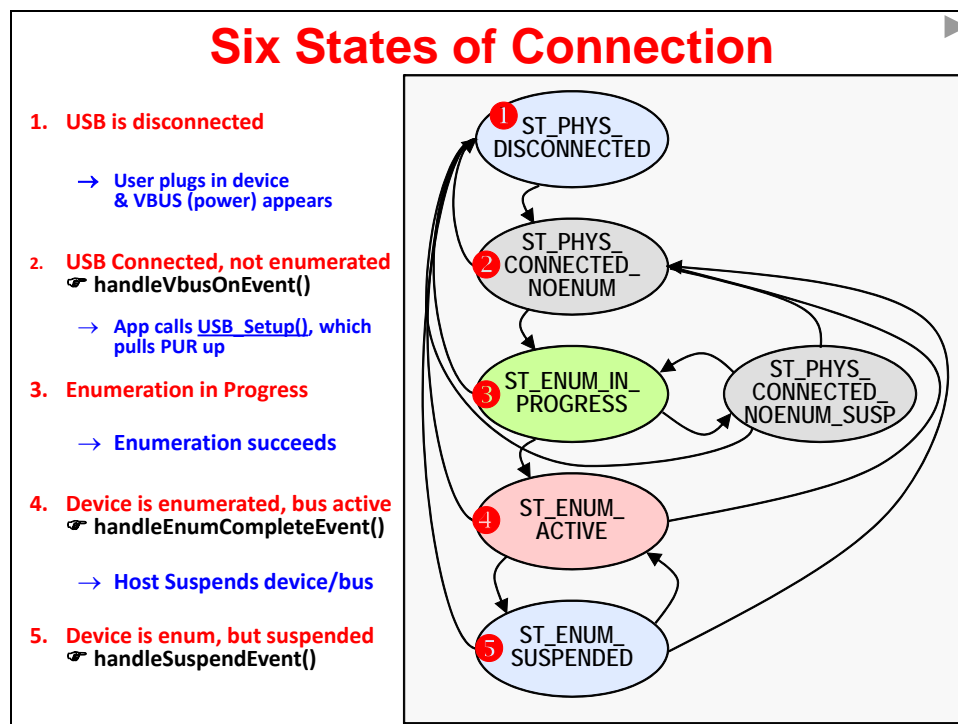
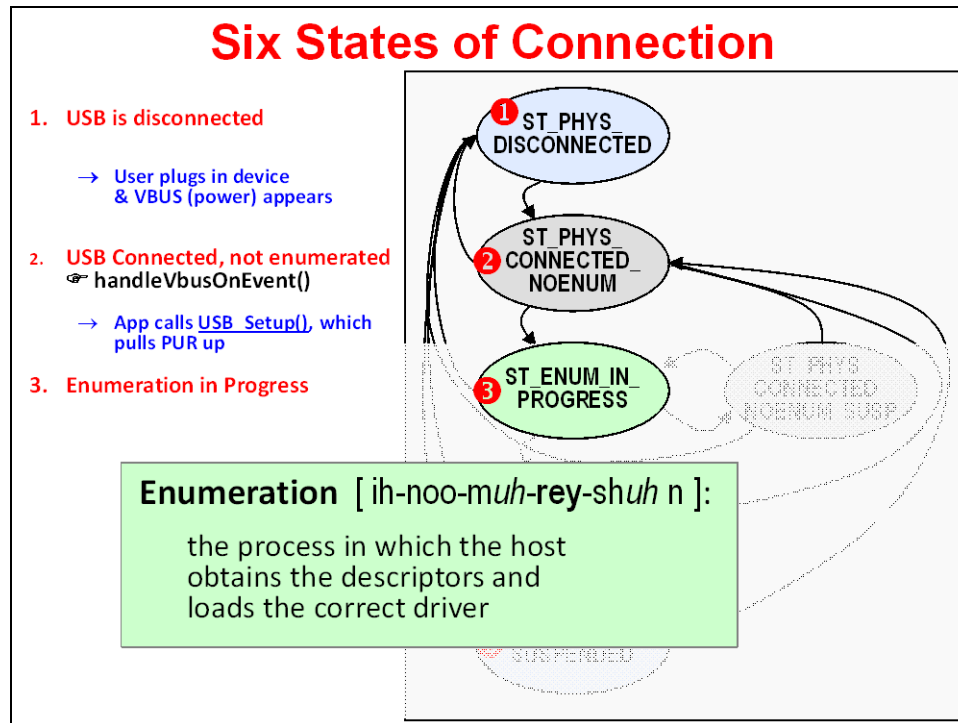
Clipped from, "USB Complete: The Developer's Guide" by Jan Axelson (ISBN 1931448086) ↗

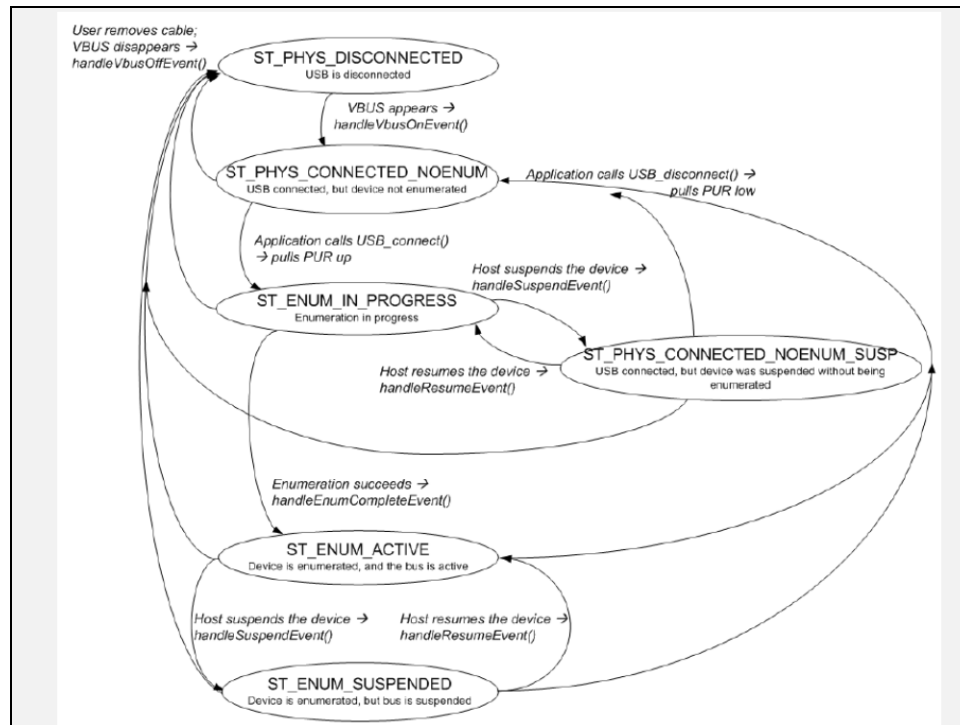
A. Plan Your System

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

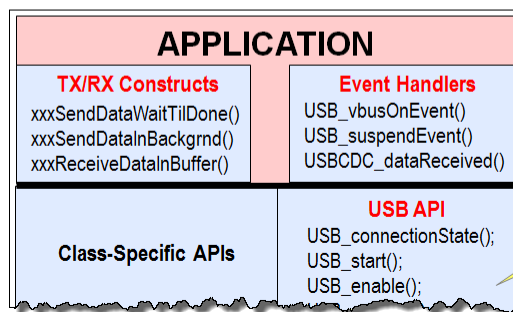
B. Connect & Enumerate





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

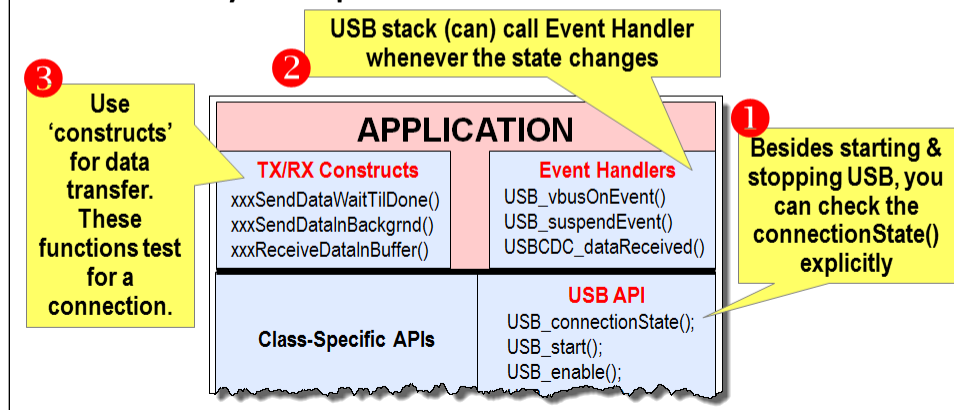
Respond to Connection State (as needed)

- ◆ 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

```

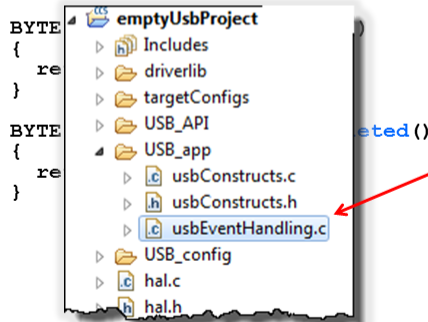
BYTE USB_handleVbusOnEvent() {
    if (USB_enable() == kUSB_succeed) //Connect when VBUS appears
    {
        USB_start();
    }
    return FALSE;
}

```

```

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

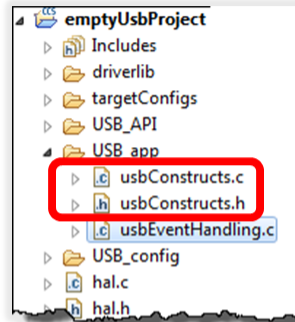
3 Construct Functions

```

// 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
{

```

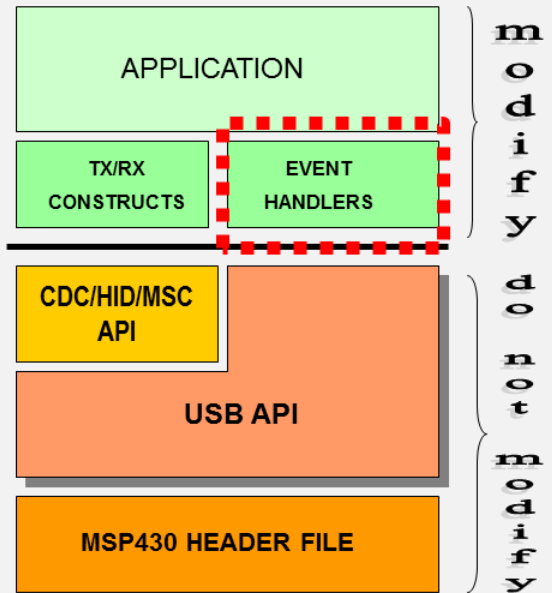


- ◆ Function begins USB send operation and returns immediately, while send occurs in background (i.e. asynchronous function)
- ◆ Retries will be attempted if the previous send hasn't completed
- ◆ If the bus isn't present, it does nothing and simply returns
- ◆ Constructs are defined in usbConstructs.c/h
- ◆ They are example code – you can use and/or modify them



Under the Hood: API Stack Diagram

- **Constructs**
 - Recommended constructs for sending/receiving data
- **Events**
 - API-driven “interrupts” for major USB events
- **Device class-specific API**
 - CDC, HID, or MSC
- **USB protocol level API**
 - Calls common to any USB device
- **Device header file**
 - Standard definitions



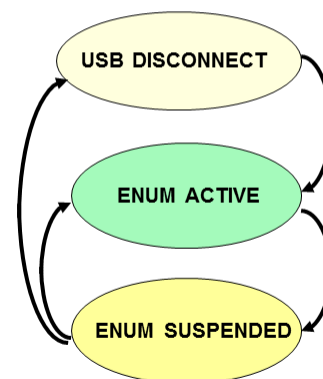
Final Thoughts

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 7 – Using USB Devices

Lab 7 – USB Devices

- ◆ **Lab 7a – 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 7b – CDC LED On/Off Toggle**
 - Similar to Lab7a, but using CDC class to transfer the data
 - Host-side uses CCS serial Terminal (or Putty)
- ◆ **Lab 7c – 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 7d – 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 CDC)
 - Read data on host with serial terminal



Lab Topics

USB Devices	7-27
<i>Lab 7 – Using USB Devices.....</i>	<i>7-29</i>
<i>Lab 7a – LED On/Off HID Example</i>	<i>7-31</i>
<i>Lab 7b – LED On/Off CDC Example.....</i>	<i>7-34</i>
Play with the demo.....	7-37
<i>Lab 7c – CDC ‘Simple Send’ Example</i>	<i>7-39</i>
<i>Lab 7d – Creating a CDC Push Button App.....</i>	<i>7-41</i>
Import Empty USB Project Steps.....	7-41
Use the Descriptor Tool	7-42
Add ‘Custom’ Code to Project.....	7-44

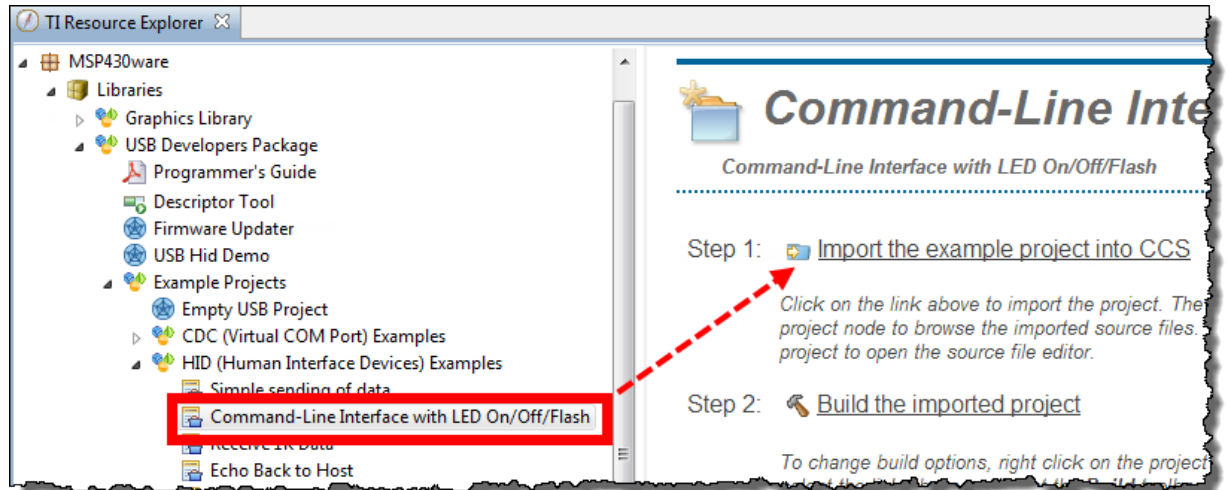
Lab 7a – LED On/Off HID Example

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*



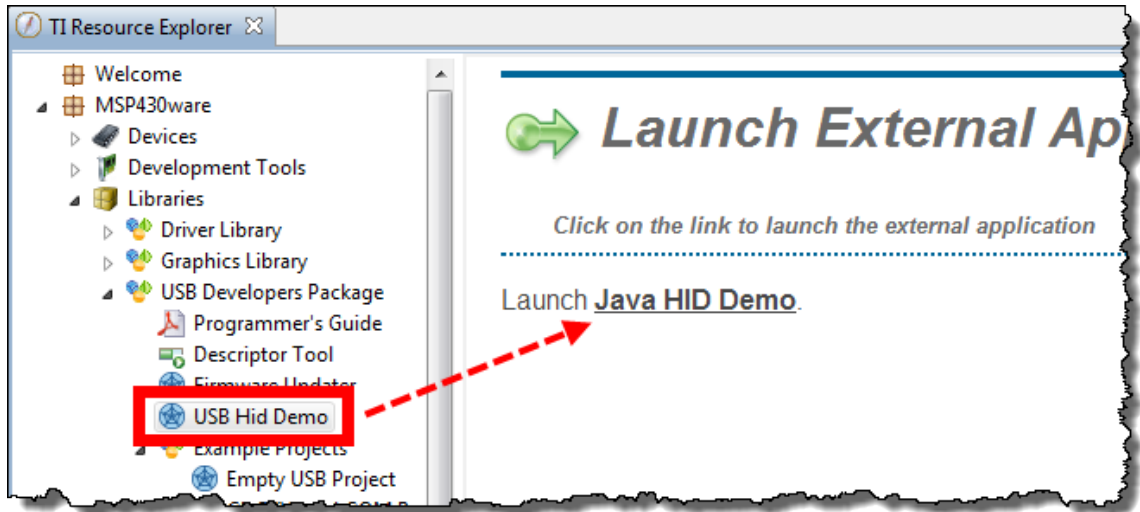
2. Build the project.

3. Launch the debugger and wait for the program to load to flash; then start the program running.

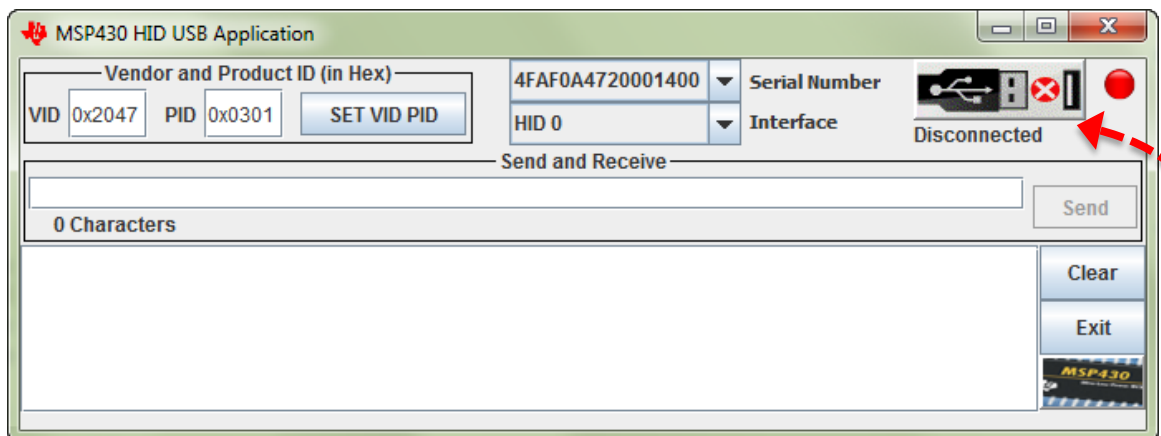
At this point, the MSP430 should start running the USB application. You may see Windows enumerate the USB device (in this case, your Launchpad); this usually appears as a popup message from the system tray saying that a USB device (“USB input device”) was enumerated.

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.

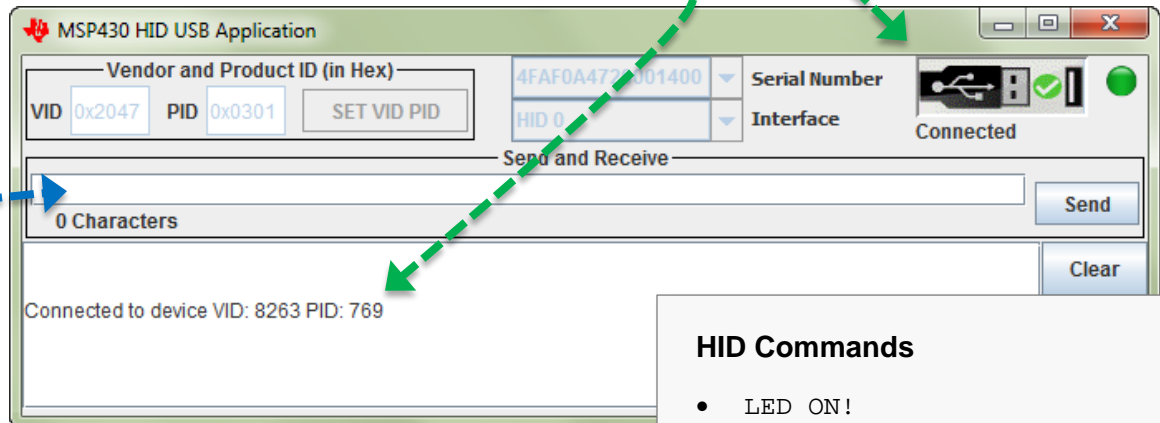
5. Switch back to the USB HID Demo application.

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.

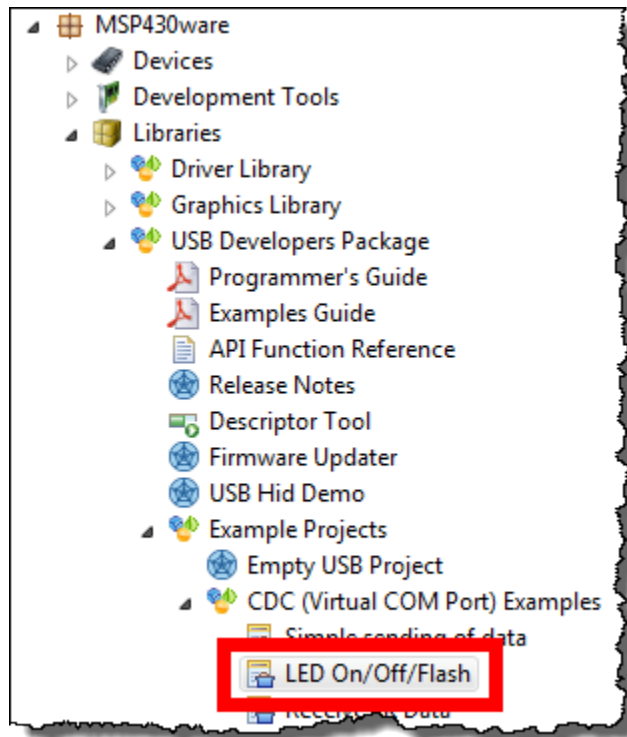
Along with the LED changing, you will see the command repeated back to the log.

Lab 7b – 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.

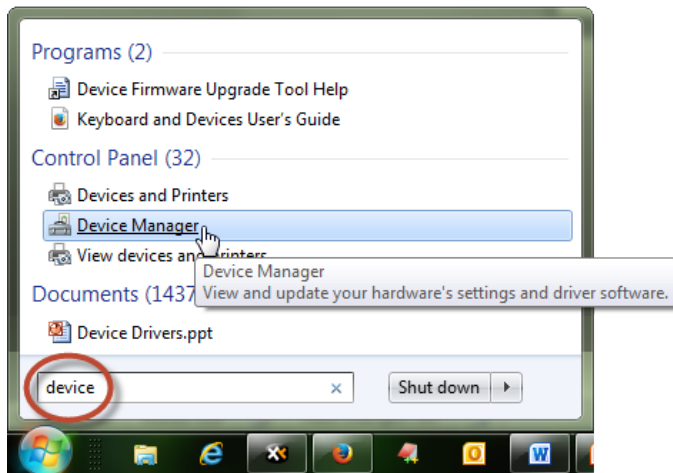


The first time you run the program, Windows may not be able to enumerate the USB CDC driver. You might see an error such as this pop up.

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:

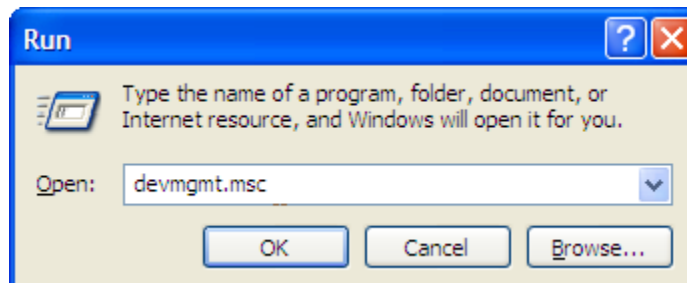


In most versions of Windows, such as Windows XP, you can also run the following program from a command line to start the Device Manager:

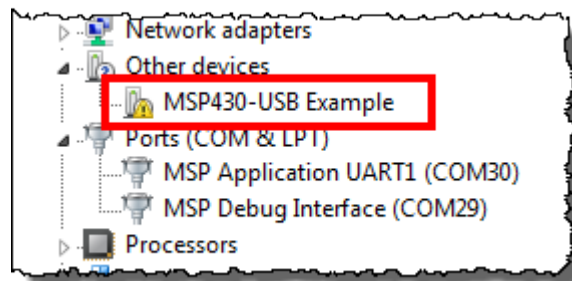
```
devmgmt.msc
```

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.

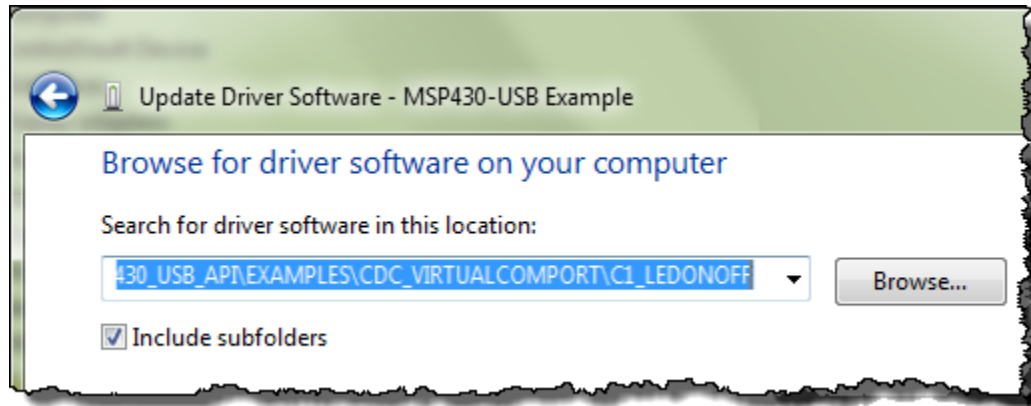
For Windows 7, the steps include:

Right-click on the driver → Update Driver Software...

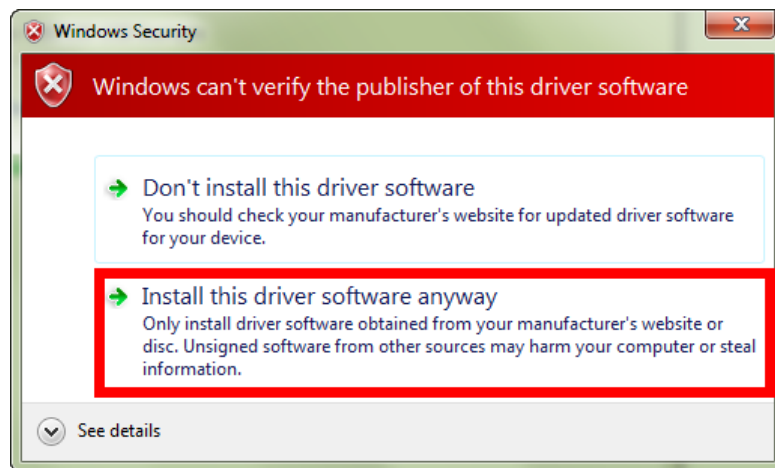
Click Browse my computer for driver software

Select the following (or wherever you installed the USB Developers Package)

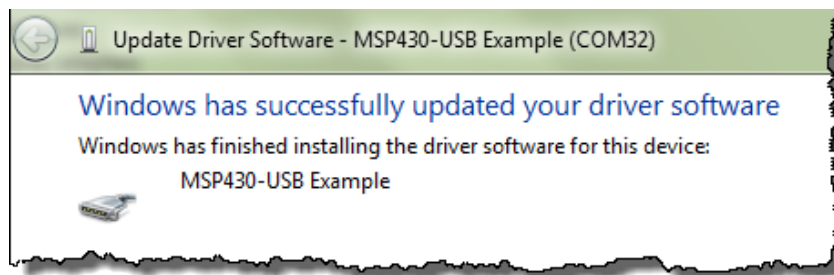
```
C:\TI\MSP430\MSP430USBDEVELOPERSPACKAGE_4_00_02\MSP430_USB_SOFTWARE\MSP430_USB_API\EXAMPLES\CDC_VIRTUALCOMPORT\C1_LEDONOFF
```



During the installation, the following dialog may appear. If so, choose to *Install* the driver.



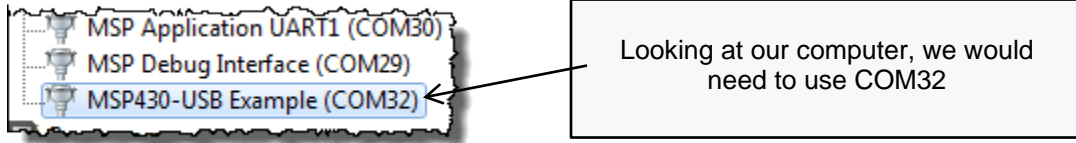
When complete you should see:



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

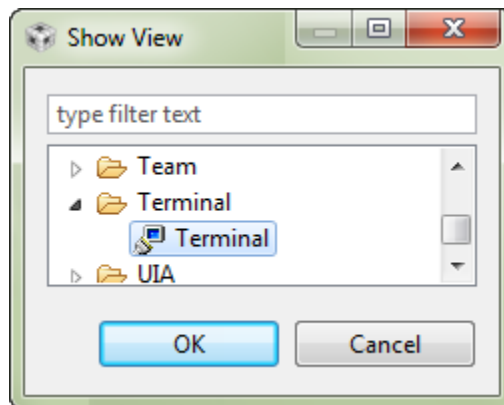
Before we can communicate with the device, though, we also need to open a serial terminal.

16. Open your favorite serial terminal and connect to the MSP430.

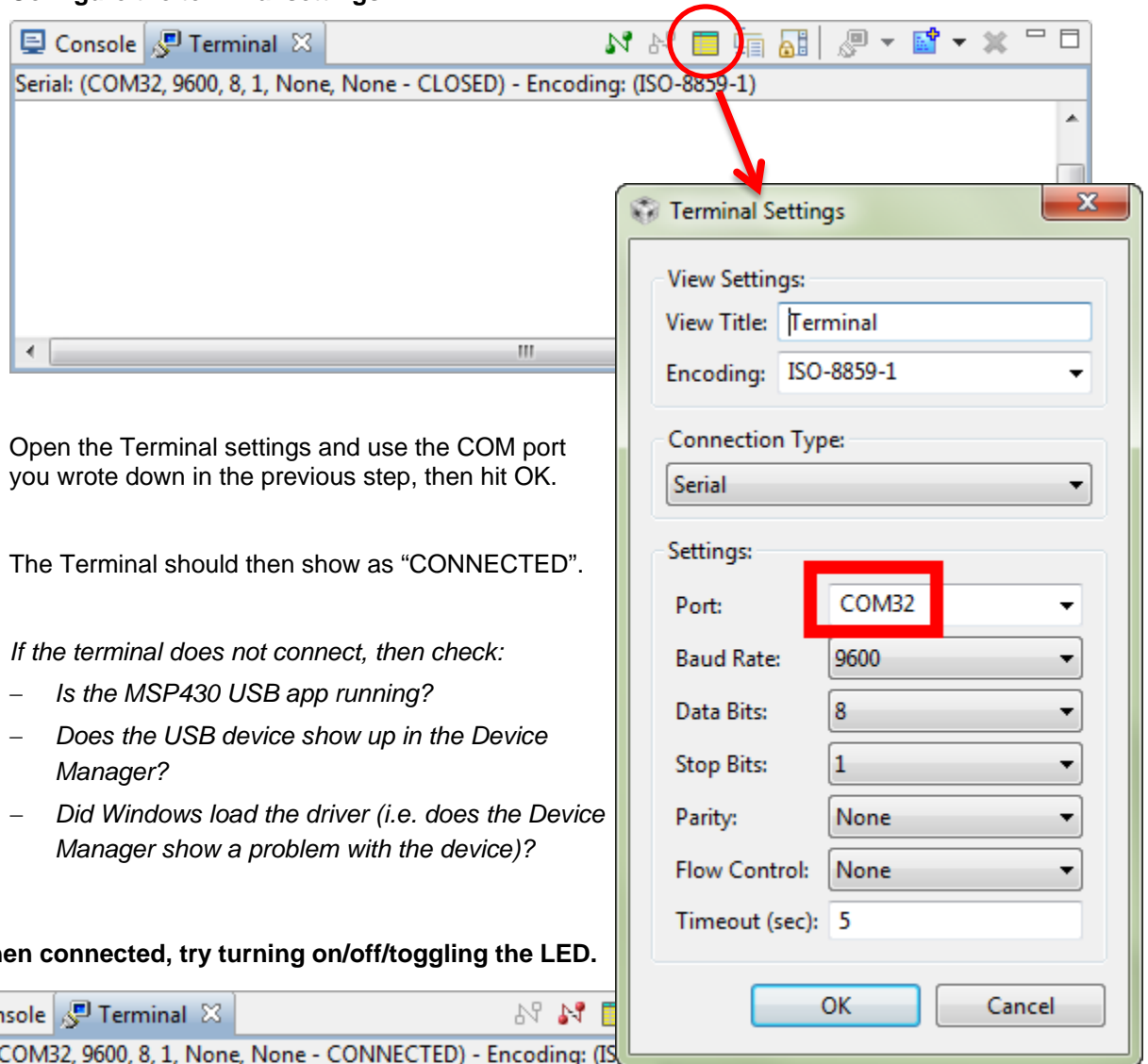
Putty and Tera Term are common favorites, but we'll provide directions for using the Terminal built into CCS.

a) Open the Terminal window.

Window → Show View → Other...



b) Configure the terminal settings:



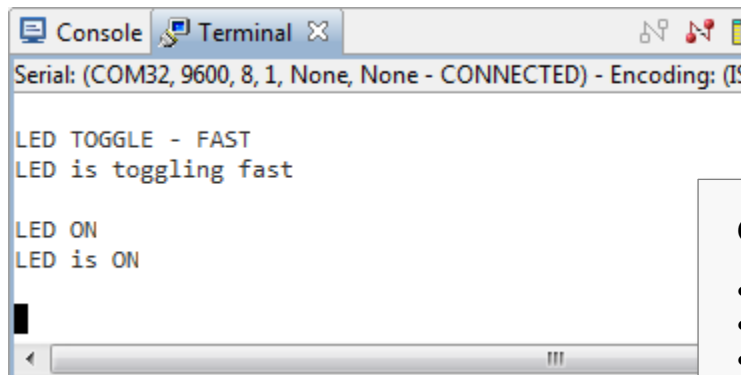
Open the Terminal settings and use the COM port you wrote down in the previous step, then hit OK.

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.

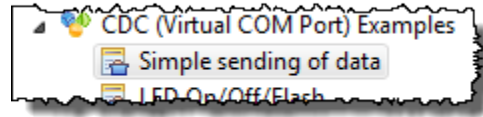
18. When done experimenting...

- Stop the terminal (hit red disconnect button).
- Terminate the debugger.
- Close the project.

Lab 7c – 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.

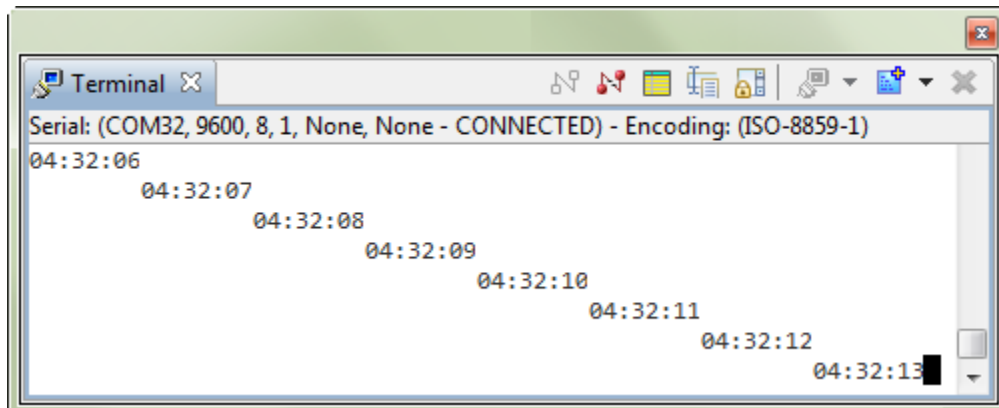
21. Start the program.

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 7d – 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 a virtual serial (CDC) comm port in Windows. Thus, we'll use a CDC 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.

CDC Simple Send Example – we'll borrow a bit of code from the CDC 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 displays the TI Resource Explorer interface. On the left, a tree view shows the project structure under 'Example Projects', with 'Empty USB Project' highlighted by a red rectangular box. The main content area on the right features a header for 'Empty USB project' with a sub-header 'Creates an empty USB project to start development'. Below this, a bolded instruction reads: 'These are the steps to import the project, use the descriptor tool, build the project'. Three steps are listed:

- 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 imported source files. 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

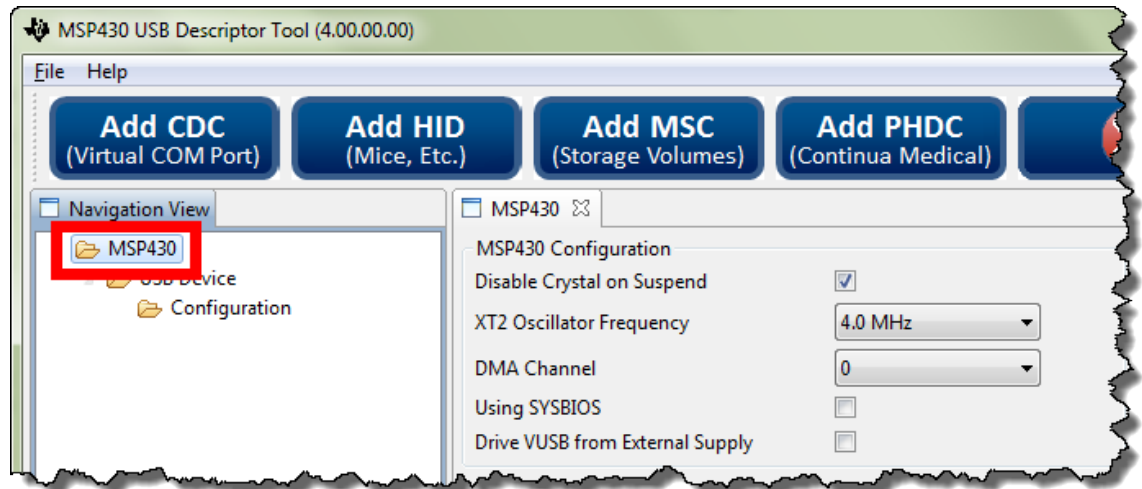
2. Launch the Descriptor Tool.

-  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.

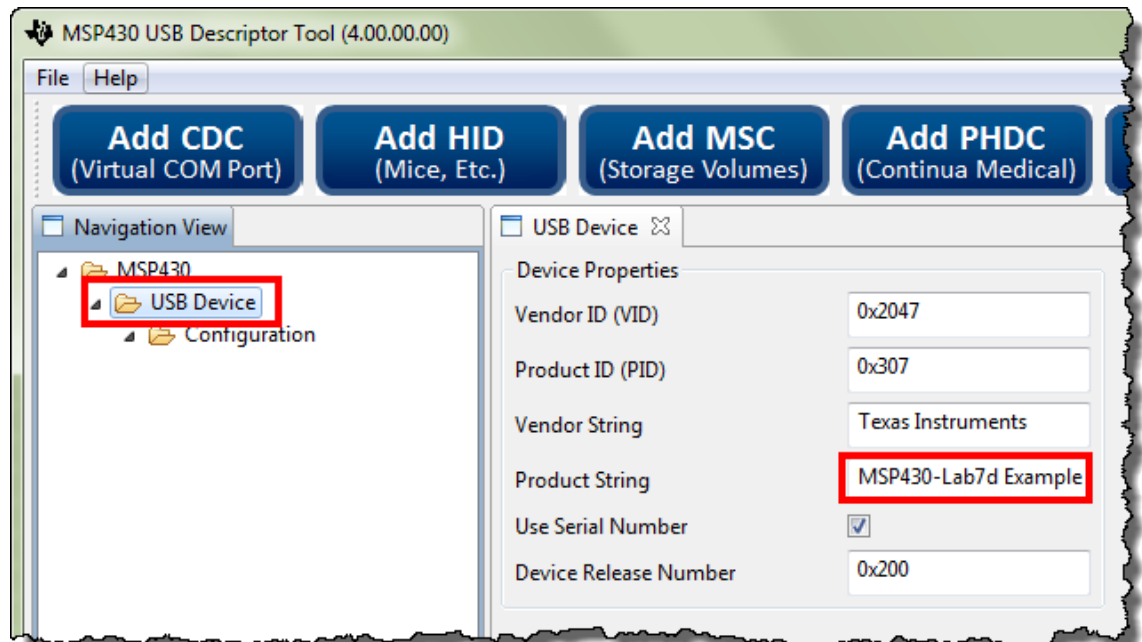
We will take a quick look at the organization levels in the tool. In most cases, we will use the tools defaults.

a) MSP430 level ... use the defaults.



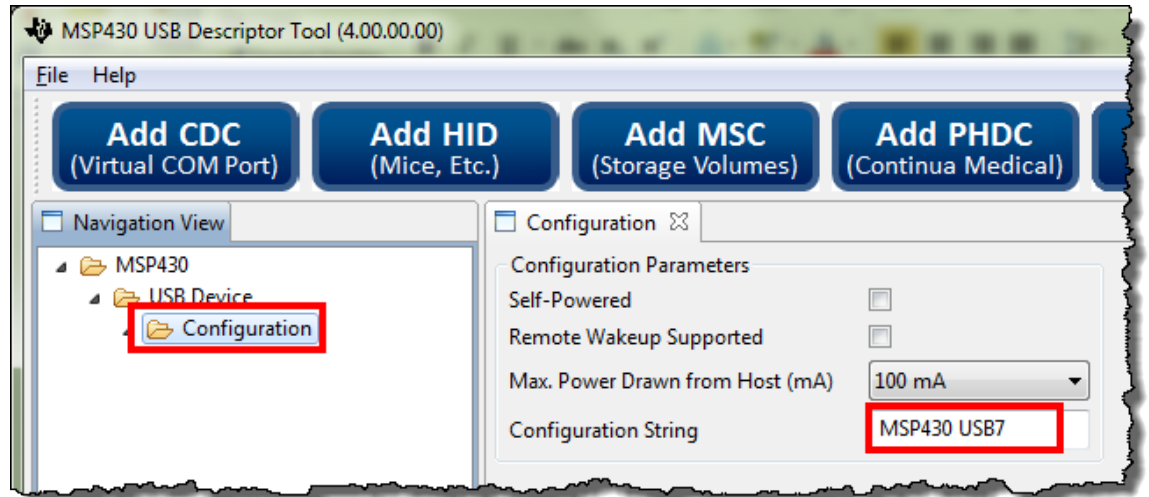
b) USB Device

We suggest changing the Product String – easier to see that it is different than other examples. Also, we recommend changing the PID (we picked '307' arbitrarily). For a real design, you would usually need to purchase the VID/PID (or obtain a free PID from TI).



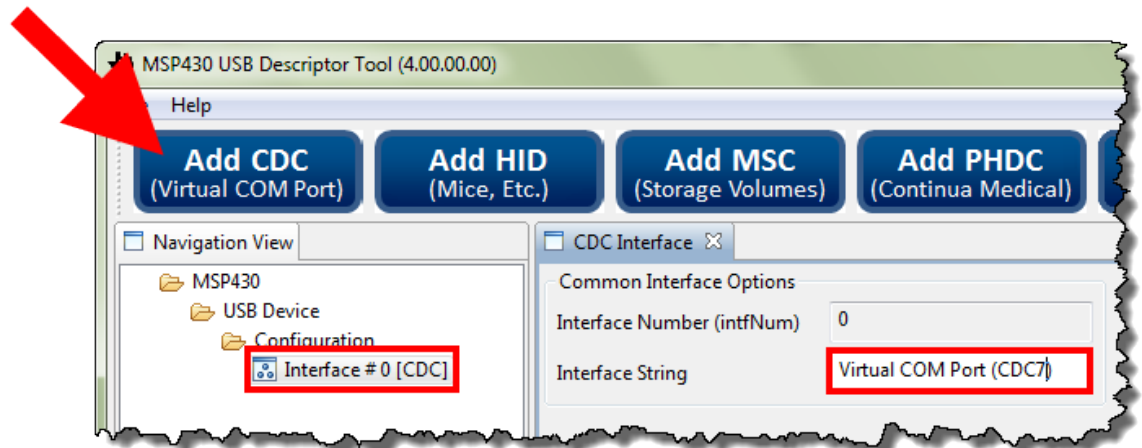
c) Configuration

Once again, we chose to vary the string so that it would be a little bit less generic.



d) Add CDC (Virtual COM Port)

Once again, we chose to vary the string so that it would be a little bit less generic.



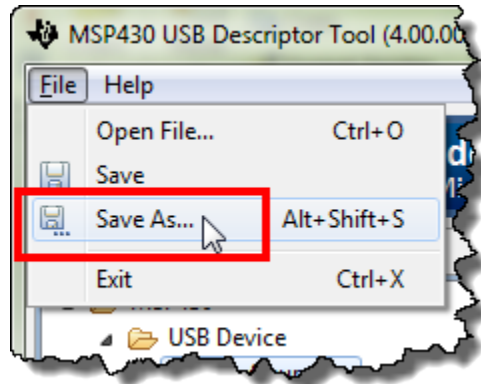
e) Click the button to generate the descriptor files.

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.)



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\<target>\workspace\emptyUsbProject\USB_config\
```

g) You can close the Descriptor Tool.

4. Rename the project to lab_07d_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\<target>\lab_07d_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.

```
char pbStr[5] = ""; // Stores the string to send
volatile unsigned short usiButton1 = 0; // Stores the button state
```

9. Add additional setup code.

```
GPIO_setAsOutputPin( GPIO_PORT_P4, GPIO_PIN7 );
GPIO_setAsInputPinWithPullUpresistor( GPIO_PORT_P2, GPIO_PIN1 );
initTimers();
```

10. Add code to ST_ENUM_ACTIVE state.

```
// 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
    // 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((BYTE*)pbStr, 5, 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
    }
}
```

11. Add #include "USB_app/usbConstructs.h".

We need to use this header file since it supports the `cdcSendDataInBackground()` function we are using to send data via USB.

12. Build and launch debugger. Then run the program.

13. Make sure the Windows driver installs.

As discussed during the presentation, the Windows needs help finding the .INF file for CDC driver interfaces.

Open Windows Device Manager

When the USB driver shows up, if it's the first time you have connected the 'device', you will need to help Windows load the driver by pointing Windows towards the .inf file created by the Descriptor Tool. (It should have been placed in the USB_config folder inside your project.)

Which Com port did Windows associate with it? _____

14. Verify your program works

Once the the driver is loaded and working properly, open your Terminal, making sure to use the proper com port.

At this point:

- The Red LED should be blinking on/off.
- The Green LED should light when the button is pushed ...
- ... and the state of the button should be written to the serial Terminal.

Note: Ocassionally, you may run into a Windows driver error at this point. It's evidenced by:

Red LED is flashing and Green LED lights when button is pushed – which means the application seems to be working fine ... but, no status shows up in Windows serial terminal.

You may also notice that if you try starting the terminal that you get the error msg: "COM xx is already in use".

This is most likely due to Windows getting confused with our use (and reuse) of USB devices – especially if we have used the sameUSB VID/PID for multiple interfaces. In other words, Windows thinks it has the right driver loaded ... but it may end up having an old match.

This can be solved by deleting the driver; removing the USB (for the Launchpad); waiting for Windows to register the device is missing; then reinserting the device and helping Windows load the driver again. (*While that usually works, every once-in-a-while we've needed to reboot our computer, as well.*)

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*
 - ◆ *MSP430F5529 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)	8-1
<i>What is Arduino</i>	8-3
<i>Energia</i>	8-4
<i>Programming Energia (and Arduino)</i>	8-7
Programming with ‘Wiring’.....	8-7
Wiring Language/Library Reference.....	8-8
How Does ‘Wiring’ Compare?.....	8-9
Hardware pinout.....	8-10
<i>Energia IDE</i>	8-12
Examples, Lots of Examples.....	8-13
<i>Energia/Arduino References</i>	8-14
<i>Lab 8</i>	8-15
Installing Energia.....	8-16
Installing the LaunchPad drivers.....	8-16
Installing Energia.....	8-16
Starting and Configuring Energia.....	8-17
Lab 8a – Blink.....	8-20
Your First Sketch.....	8-20
Modifying Blink.....	8-23
Lab 8b – Pushing Your button.....	8-24
Examine the code.....	8-24
Reverse button/LED action.....	8-25
Lab 8c – Serial Communication (and Debugging).....	8-26
What if the Serial Monitor is blank? (‘G2553 Launchpad Configuration’).....	8-27
Blink with Serial Communication.....	8-28
Another Pushbutton/Serial Example.....	8-28
Lab 8d – Using Interrupts.....	8-29
Adding an Interrupt.....	8-29
Lab 8e – Using TIMER_A.....	8-31
<i>Appendix – Looking ‘Under the Hood’</i>	8-32
Where, oh where, is Main.....	8-32
Two ways to change the MSP430 clock source.....	8-34
Sidebar – initClocks().....	8-35
Sidebar Cont’d - Where is <i>F_CPU</i> defined?.....	8-36
<i>Lab Debrief</i>	8-37

What is Arduino

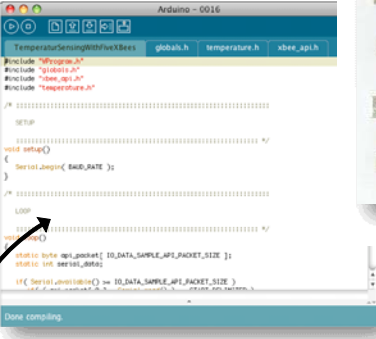
Physical Computing ... Hardware Hacking ... a couple of the names given to Arduino.

- *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.

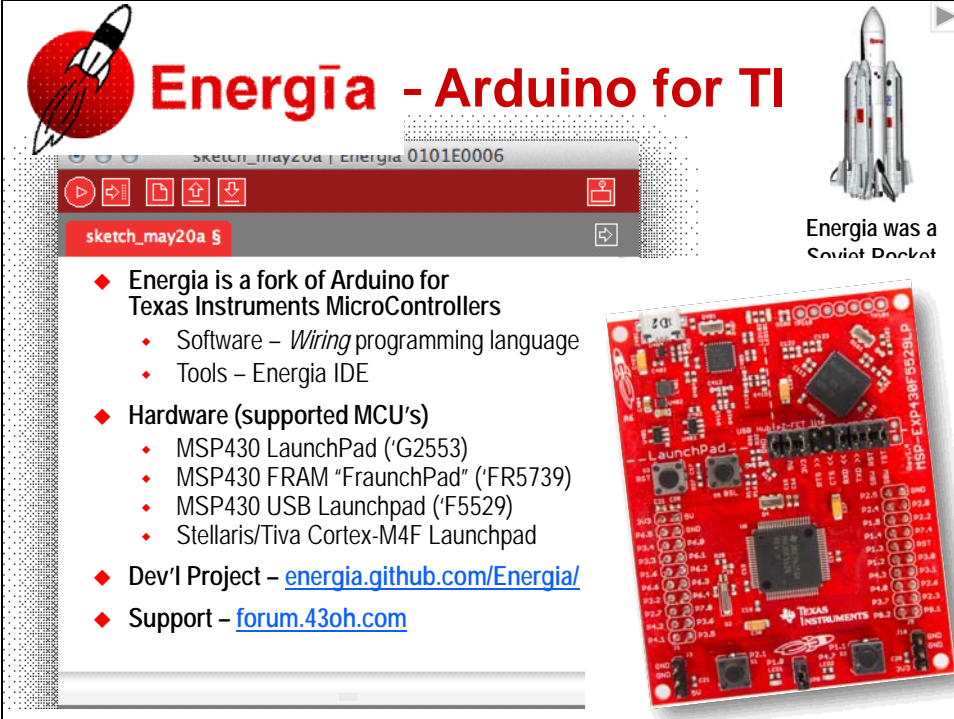
¹ <http://www.wired.com/gadgetlab/2008/04/just-what-is-an/>

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

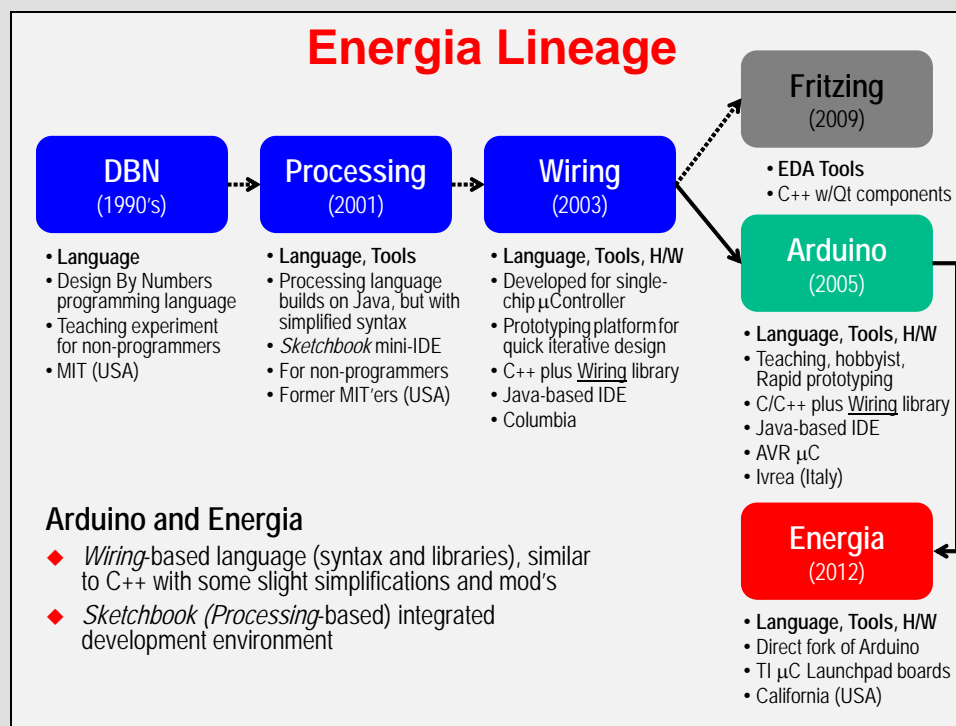
Energia was a Soviet Rocket

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.

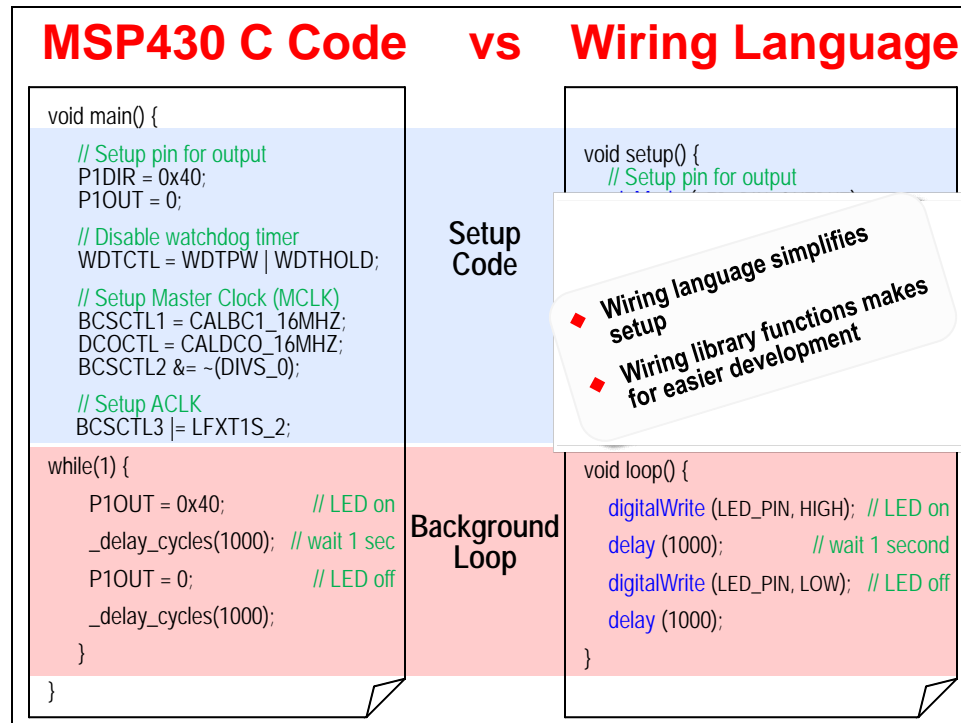
The screenshot shows the 'Wiring Library Reference' page. At the top, there is a navigation bar with links: Home, Download, Getting Started, Reference, Getting Help, FAQ, Projects Using Energia, and Contact Us. The main heading is 'Language Reference'. Below this, a note states: 'Energia programs can 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?



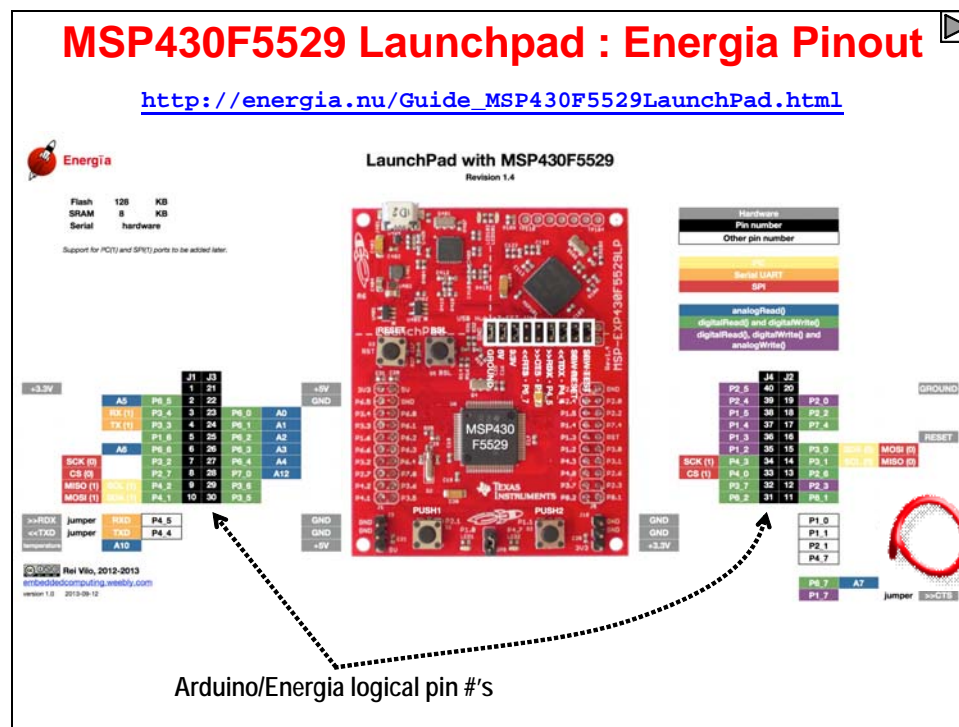
This comparison helps to demonstrate the simplicity of programming with Energia. As stated before, this can make for very effective rapid prototyping.

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.

Hardware pinout

Arduino programming refers to Arduino “pins” throughout the language and examples. In the original implementation, these refer directly to the original hardware platform.

When adapting the Arduino library/language over to other processors, such as the TI microcontrollers, these pins must be mapped to the available hardware. The following screen capture from the Energia wiki shows the mapping for the MSP430 (v1.5 ‘G2553) Launchpad development board. There are similar diagrams for the other supported TI boards; please find these at wiki page: <https://github.com/energia/Energia/wiki/Hardware>.



Color Coded Pin Mapping

The wiki authors have color coded the pins to try and make things easier. The **Black** numbers represent the *Arduino Pin Numbers*. Thus, you can write to the pins using the pin numbers:

```
pinMode(2, OUTPUT);  
digitalWrite(2, HIGH);
```

The **Grey** values show the hardware elements that are being mapped, such as the LED's or PushButton. You can use these alternative names: RED_LED; GREEN_LED; PUSH2; and TEMPESENSOR. Thus, to turn on the red LED, you could use:

```
pinMode(RED_LED, OUTPUT);  
digitalWrite(RED_LED, HIGH);
```

Pins can also be address by there alternative names, such as P1_0. These correlate to the GPIO port (P1) and pin (0) names (P1.0) as defined by the MSP430. (In fact, the Launchpads conveniently show which I/O pins are mapped to the Boosterpack header connectors.) Using these symbols, we can write to pins using the following:

```
pinMode(P1_0, OUTPUT);  
digitalWrite(P1_0, HIGH);
```

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](https://github.com), 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.

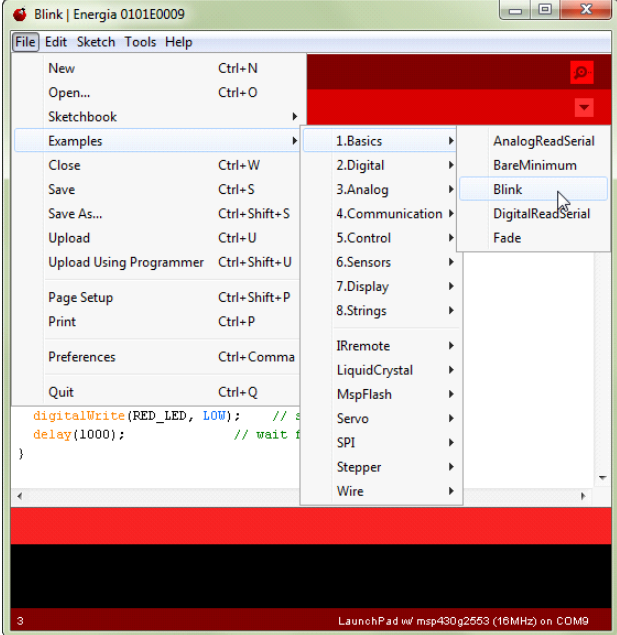
Compiling and **downloading** the program is as simple as clicking the *Download* button.

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 the LED on (HIGH = the LED lights up)
delay(1000); // wait for a 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 8

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)

Installing Energia

If you already installed Energia as part of the workshop prework, then you can skip this step and continue to [Lab 8a – 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

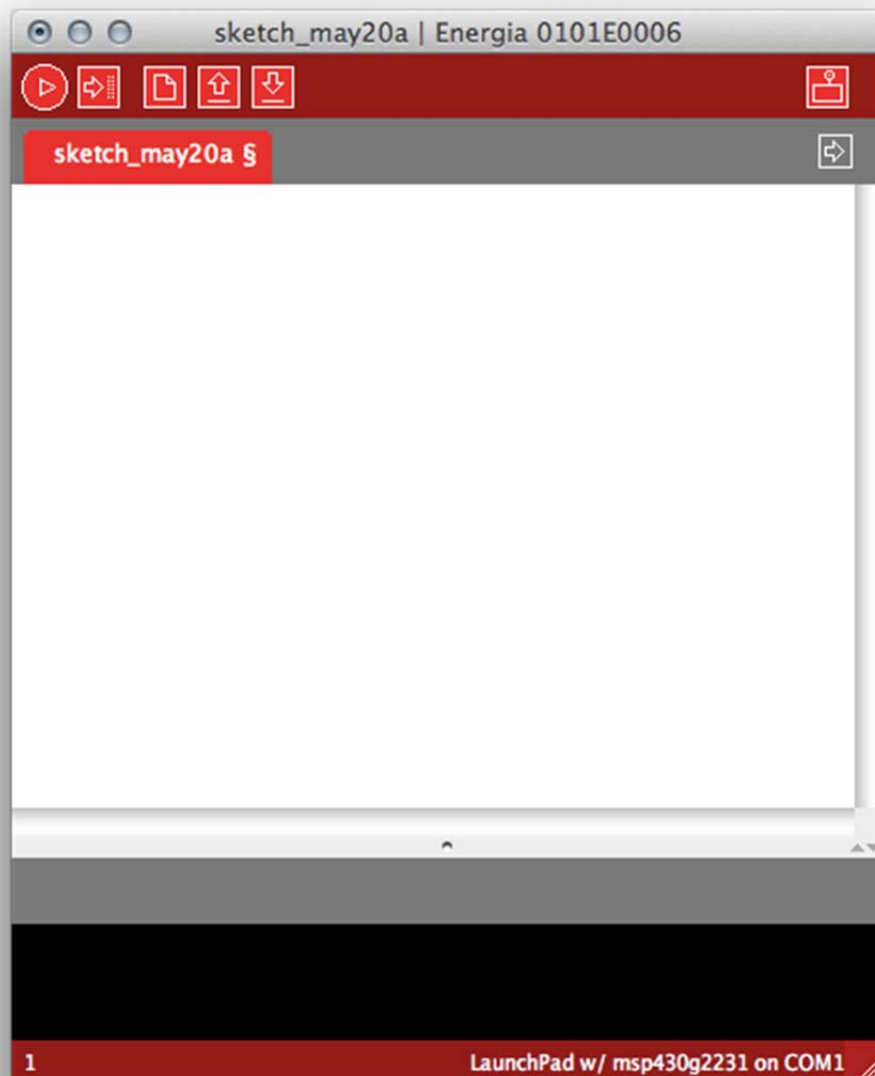
Double click and extract the energia-0101EXXX-windows.zip file to a desired location.

(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.

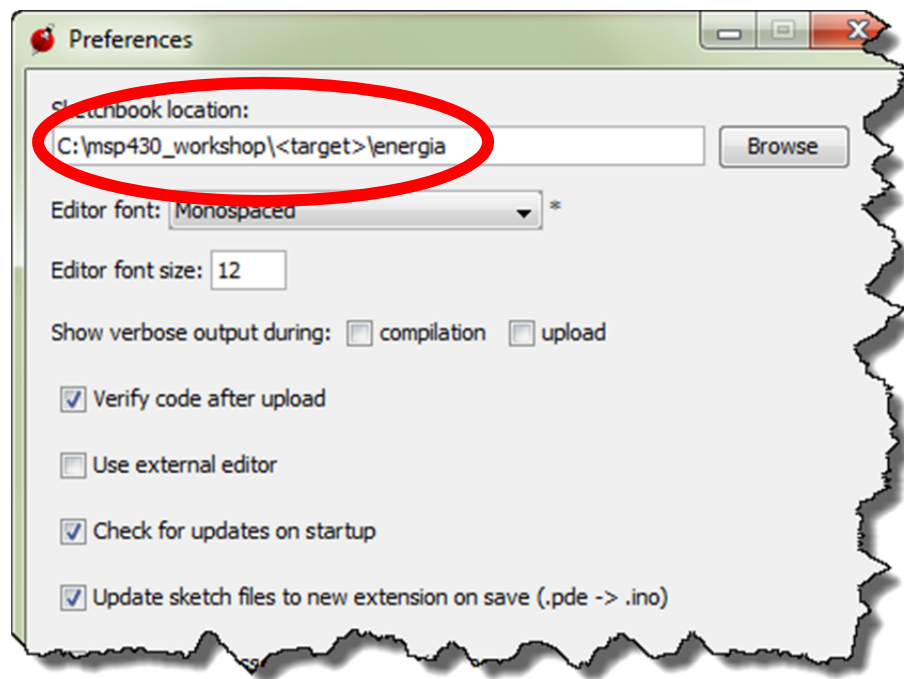
The easiest way to set this locations is via Energia's preferences dialog:

File → Preferences

Then set the *Sketchbook location* to:

C:\msp430_workshop*<target>*\energia

Which opens:



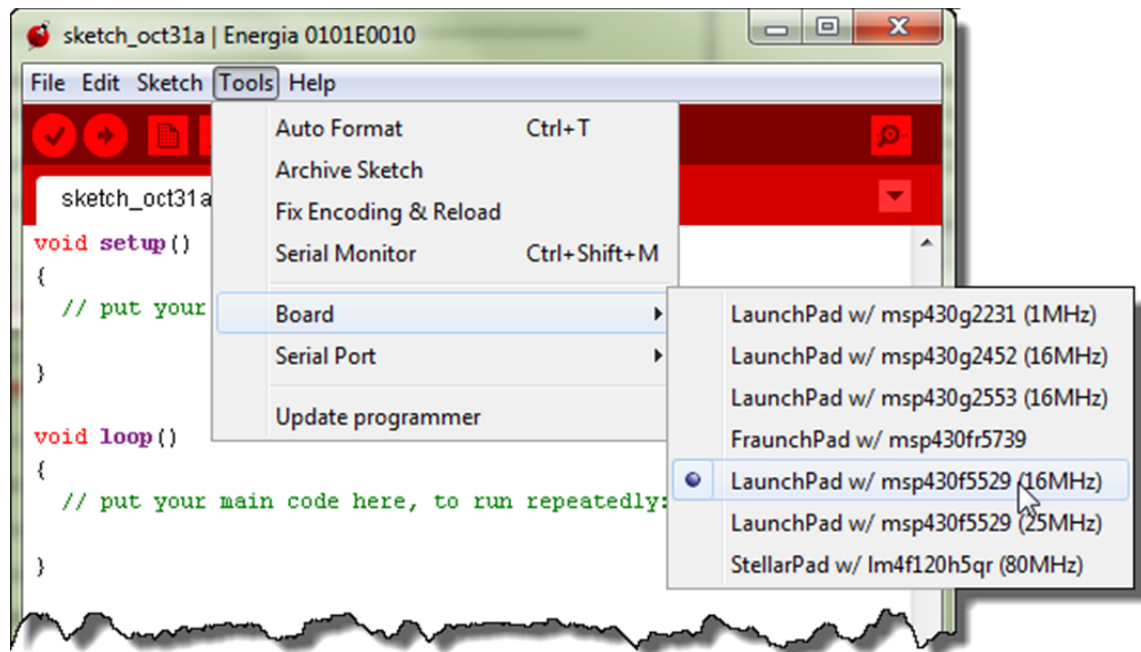
5. Selecting the Serial Port

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 8a – 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.

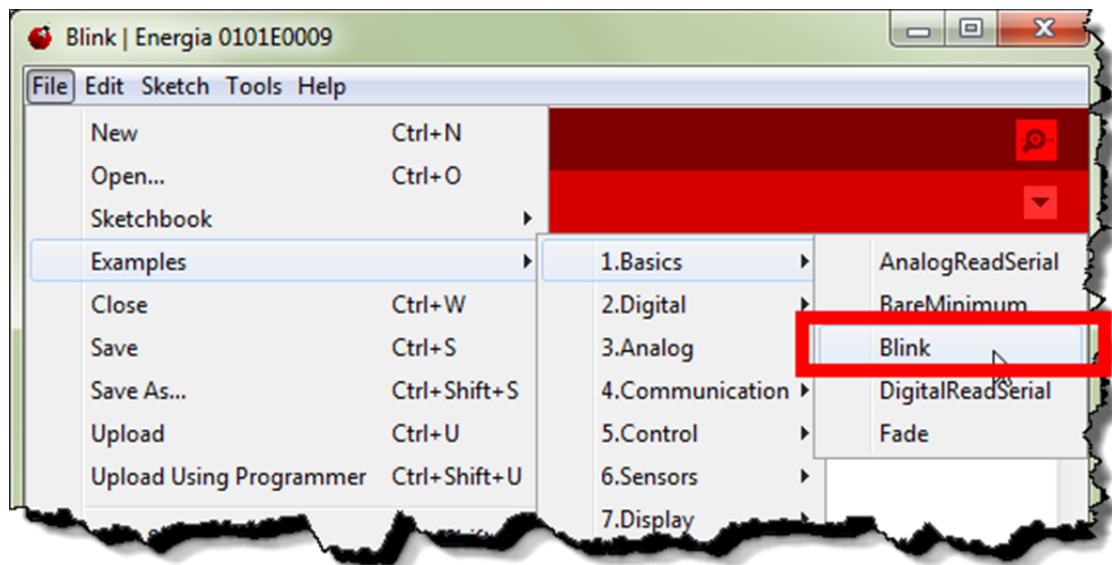
As we pointed out during the *Energia* chapter discussion, the *Wiring* language simplifies the code quite a bit.

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 Lab8d 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 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 8b – 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.

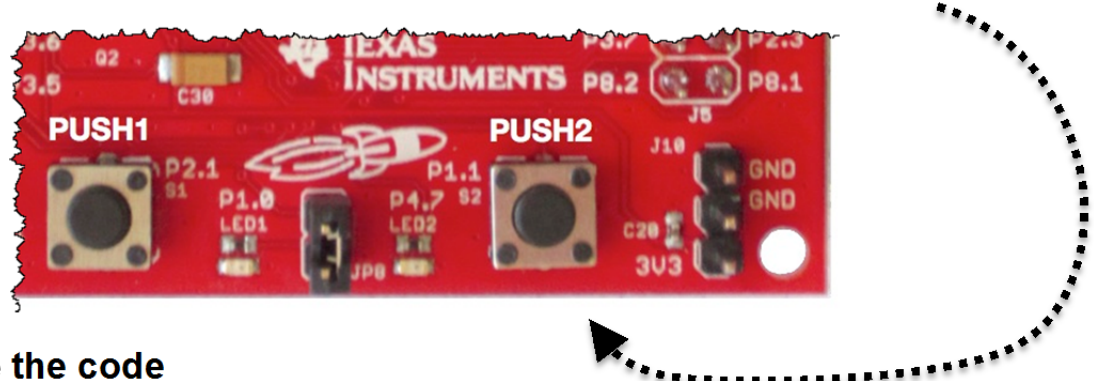
File → Examples → 2.Digital → Button

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 8c – 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.

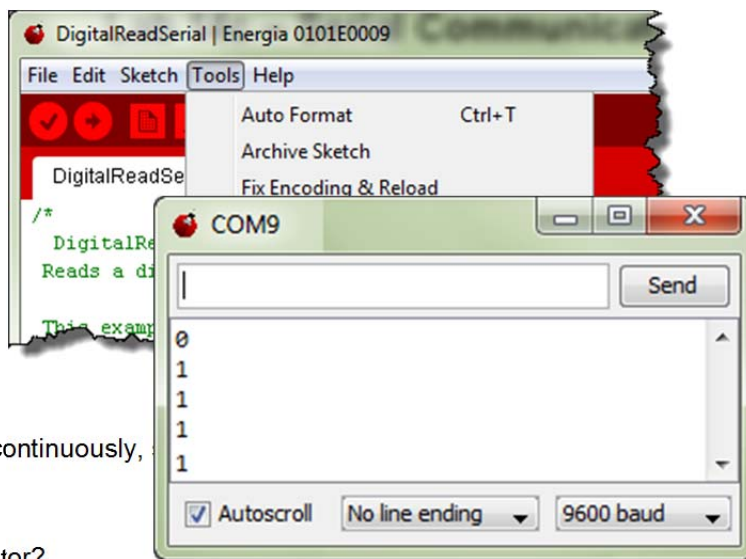
Energia includes a simple serial terminal program. It makes it easy to view (and send) serial streams via your computer.

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.

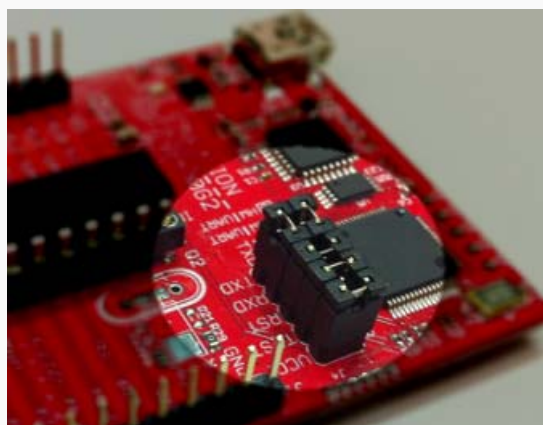
Also, notice that the value is updated continuously, writes it to port in the **loop()** function.

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 8C, 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 8d – 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 8e – 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 8d. 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)
{
  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
    BALBC1_16MHZ;
    DCOCTL = CALDCO_16MHZ;
  #elif (F_CPU >= 1000000L)
    BCSTL1 = CALBC1_1MHZ;
    DCOCTL = CALDCO_1MHZ;
  #endif

  BCSTL2 &= ~(DIVS_0);
  BCSTL3 |= LFXTS_2;

  CSTL2 &= ~SELM_7;
  CSTL2 |= SELM_DCOCLK;
  CSTL3 &= ~(DIVM_3|DIVS_3);

  #if F_CPU >= 16000000L
    CSTL1 = DCORSEL;
  #elif F_CPU >= 1000000L
    CSTL1 = DCOFSEL0|DCOFSEL1;
    CSTL3 |= 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=2048
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

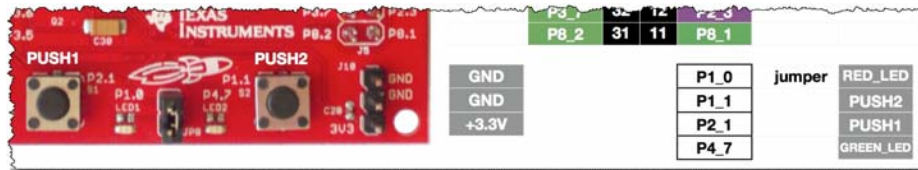
#####
```

Lab Debrief

Q&A: Lab8A (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: Lab8A (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

Q&A: Lab8B (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 µC provide these resistors as part of their hardware design).

Q&A: Lab8B (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);  
}
```

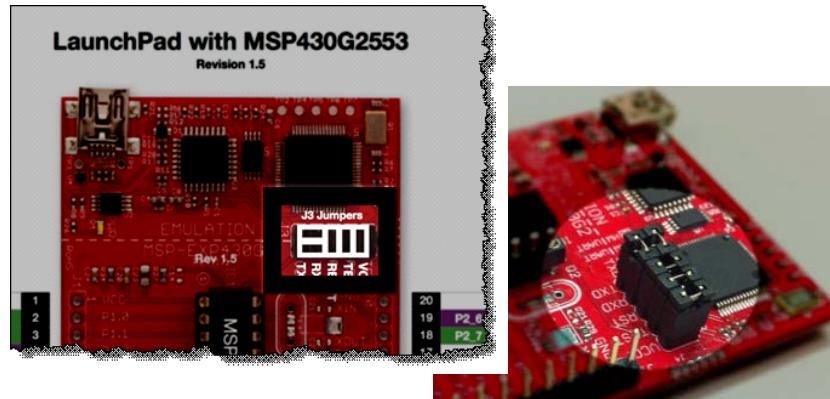
LOW (with a red slash through it)
HIGH (with a red slash through it)

Q&A: Lab8C (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: Lab8C (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: Lab8C (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.

Q&A: Lab8D

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.