MSP430 Design Workshop

STUDENT GUIDE

TEXAS INSTRUMENTS

MSP430 Design Workshop
Revision 4.01
February 2015

Important Notice

Revision History

October 2013 – Revision 3.0 (based on MSP-EXP430F5529 USB Launchpad)

November 2013 – Revision 3.01

January 2014 – Revision 3.02

February 2014 – Revision 3.10 (MSP-EXP430F5529 & MSP-EXP430FR5969 Launchpad's)

July 2014 – Revision 3.21

Jan 2015 – Revision 4.00 (based on F5529, FR5969 and FR4133 Launchpad's)

Feb 2015 - Revision 4.01

Mailing Address

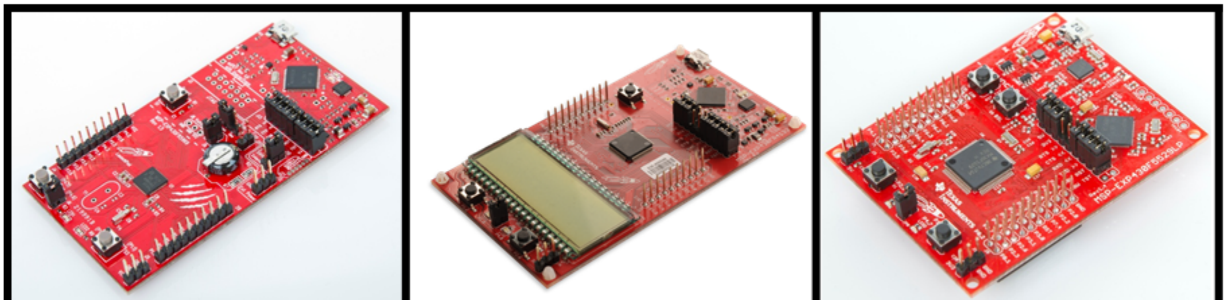
Texas Instruments
Training Technical Organization
6500 Chase Oaks Blvd – Bldg 2
M/S 8437
Plano, Texas 75023

Introduction to MSP430

Introduction

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

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



MSP-EXP430FR5969	MSP-EXP430FR4133	MSP-EXP430F5529

Chapter Topics

Introduction to MSP430 1-1

Administrative Topics 1-3

Workshop Agenda 1-4

TI Products 1-6
TI's Entire Portfolio 1-6
Wireless Products 1-7

TI's Embedded Processors 1-8

MSP430 Family 1-10

MSP430 CPU 1-14

MSP430 Memory 1-18
Memory Map 1-18
FRAM 1-21

MSP430 Peripherals 1-24
GPIO 1-24
Timers 1-25
Clocking and Power Management 1-26
Analog 1-27
Communications (Serial ports, USB, Radio) 1-29
Hardware Accelerators 1-30
Summary 1-31

ULP 1-32
Profile Your Activities 1-33

Community / Resources 1-37
References 1-39

Launchpad's 1-40
MSP-EXP430F5529LP Launchpad 1-40
MSP-EXP430FR5969 Launchpad 1-41
MSP-EXP430FR4133 Launchpad 1-41

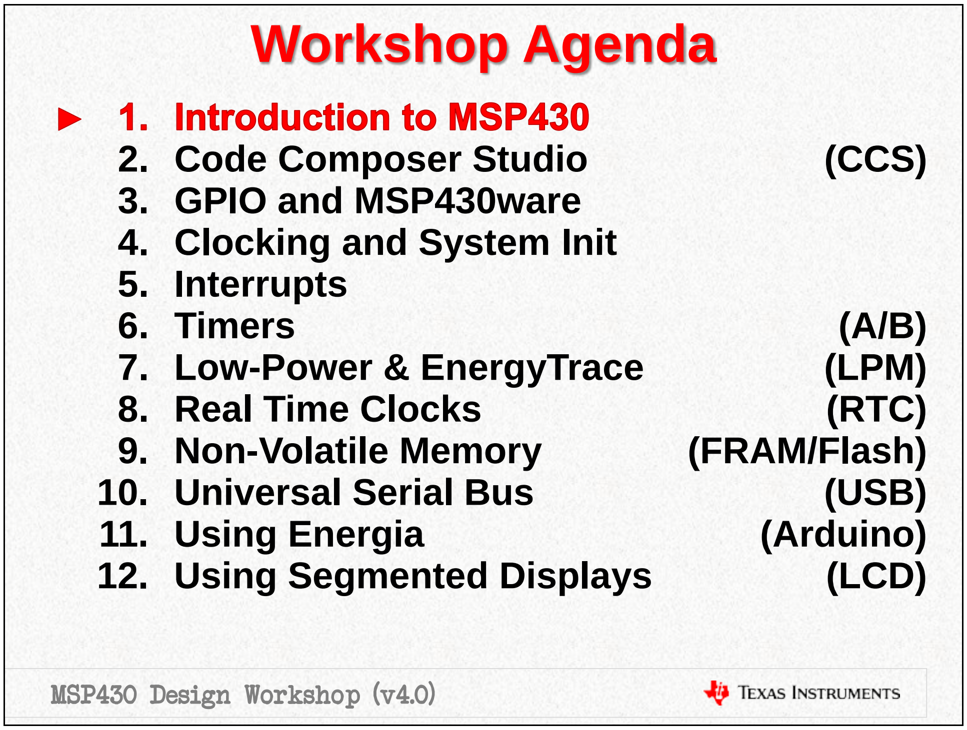
Lab 1 – Out-of-Box User Experience Lab 1-43

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.

Workshop Agenda

Here's the outline of chapters in this workshop.



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: *Low Power Optimization* – shows the basic steps for lowering power usage. Following the ULP (ultra-low power) Advisor, we can find ways to minimize power in our code. Energy Trace is a new tool for measuring power and, on the 'FR58/59xx devices, examining the states of peripherals and clocks.

Chapter 8: *Real-Time Clocks* provides a very low-power timer to keep track of calendar, time and alarms.

Chapter 9: *Nov-Volatile Memory* – provides persistant storage, even when power is removed from the device. Most MSP430 devices contain either Flash or FRAM non-volatile memory.

Chapter 10: *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 11: *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.

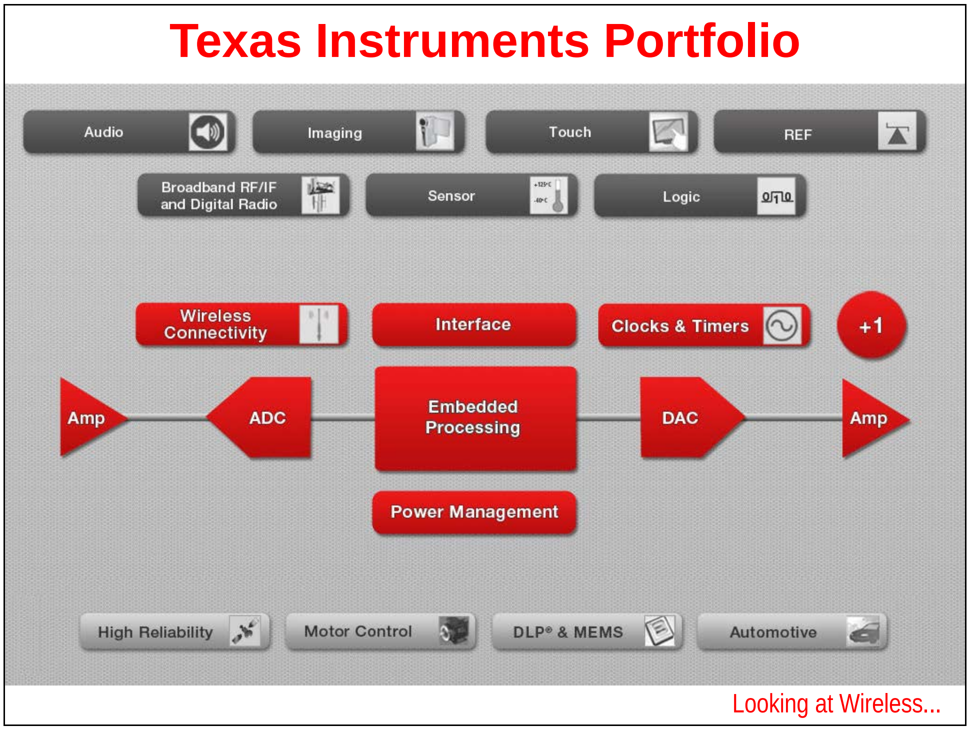
Chapter 12: Segmented LCD's (Liquid Crystal Displays) provide a convenient, low-power way of communicating with your system end-users. The 'FR4133 provides the lowest power LCD controller in the market. This chapter introduces you to LCD's in general, then to the specifics of using TI's LCD_E controller found on the 'FR4133 and its 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 CC3100 CC3200

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	Keystone
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 ▪ 250nA (RTC) ▪ 770nA (LCD) • 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: 128K	512K Flash	1MB 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	120 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 ⅓ 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.

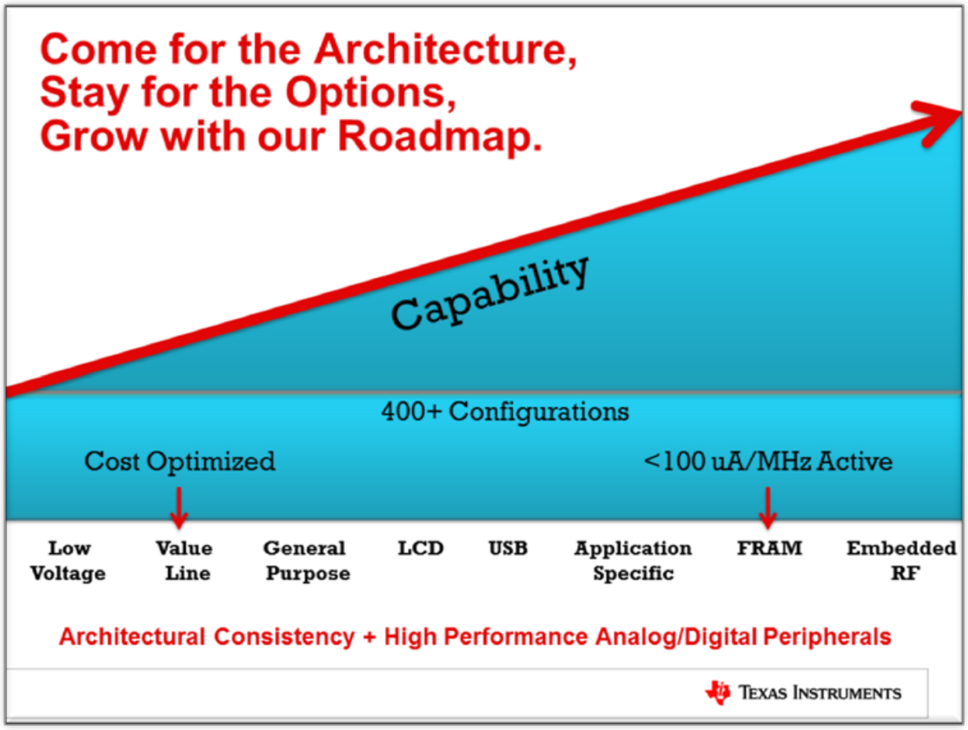
Driving to the *lowest-power dissipation* is one of those areas. In the end, the venerable **MSP430** is not to be outdone on the low end. As the MSP430 teams says, Ultra Low-Power (ULP) is "in our DNA". You know you're doing something right when the 10-year shelf-life of the battery ends up self-dissipating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

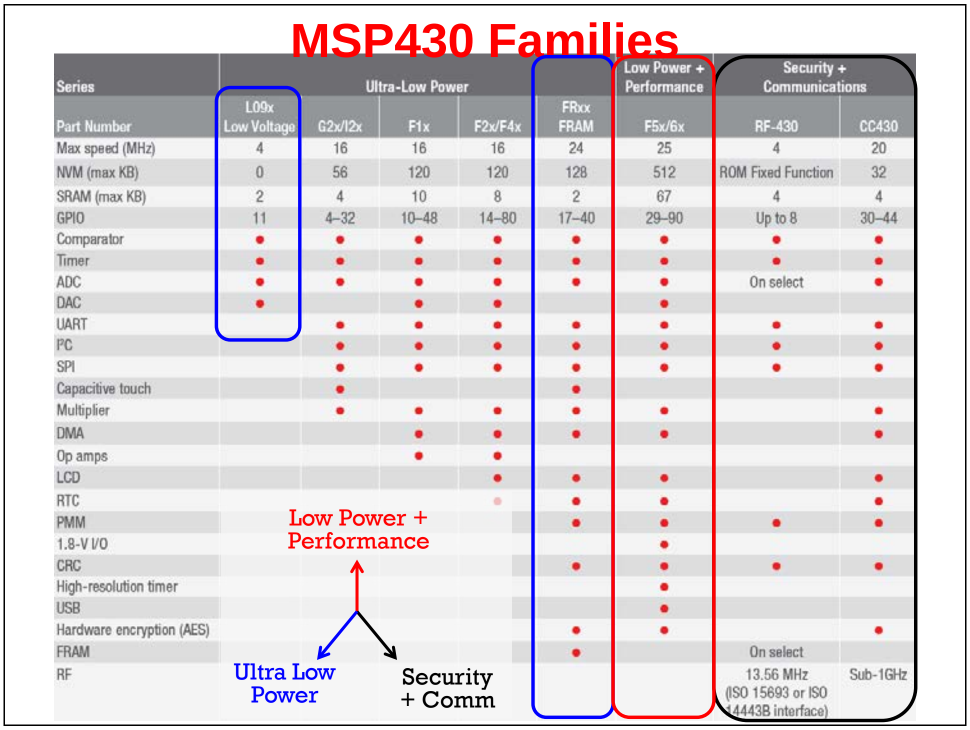
As stated, low-power is 'in our DNA'. Though, it's not all the MSP430 is known for.



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

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

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



Series	Ultra-Low Power					Low Power + Performance	Security + Communications	
Part Number	L09x Low Voltage	G2x/I2x	F1x	F2x/F4x	FRxx FRAM	F5x/6x	RF-430	CC430
Max speed (MHz)	4	16	16	16	24	25	4	20
NVM (max KB)	0	56	120	120	128	512	ROM Fixed Function	32
SRAM (max KB)	2	4	10	8	2	67	4	4
GPIO	11	4–32	10–48	14–80	17–40	29–90	Up to 8	30–44
Comparator	●	●	●	●	●	●	●	●
Timer	●	●	●	●	●	●	●	●
ADC	●	●	●	●	●	●	On select	●
DAC	●		●	●		●		
UART		●	●	●	●	●	●	●
I²C		●	●	●	●	●	●	●
SPI		●	●	●	●	●	●	●
Capacitive touch		●			●			
Multiplier		●	●	●	●	●		●
DMA			●	●	●	●		●
Op amps			●	●				
LCD				●	●	●		●
RTC				●	●	●		●
PMM					●	●	●	●
1.8-V I/O						●		
CRC					●	●	●	●
High-resolution timer						●		
USB						●		
Hardware encryption (AES)					●	●		●
FRAM					●		On select	
RF							13.56 MHz (ISO 15693 or ISO 14443B interface)	Sub-1GHz

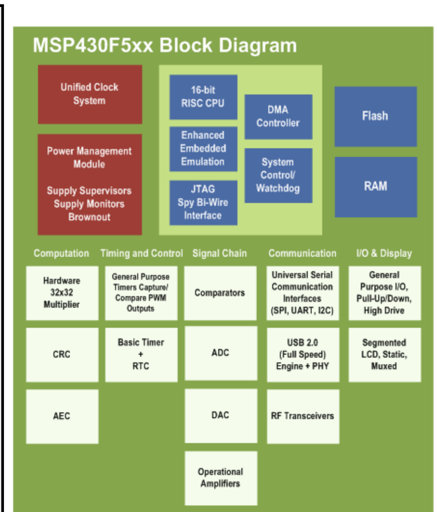
Here's a quick overview of the device we'll be using in this workshop. The MSP430F5529 is part of the F5xx series of devices and is found on the new 'F5529 USB Launchpad.

F5xx Key Features

Ultra-Low Power
- 160 µA/MIPS
- 2.5 µA standby mode
- Integrated LDO, BOR, WDT+, RTC
- 12 MHz @ 1.8V
- Wake up from standby in <5 µs

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

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



Looking at the 'FR59xx...

MSP430FR58xx/59xx

MSP430FR58/59xx
Ultra Low Power
16-bit MCU

16MHz

Memory
- FRAM (32/48 / 64 KB)
- RAM (1 or 2 KB)
- MPU

Debug
- Real Time JTAG
- Embedded emulation
- Bootstrap Loader

Accelerators
- 32x32 Multiplier
- DMA (3 Ch)
- CRC16
- AES256 Encryption (FR59xx)

Serial Interfaces
- 3 Serial Interfaces (eUSCI)
- 2 UART + IrDA or SPI
- 2 I2C or SPI

Analog
- 12-bit SAR ADC (up to 16 ch)
  - Differential inputs
  - Window comparators
- Comparator (Comp_E)
- Vref (REF_A)

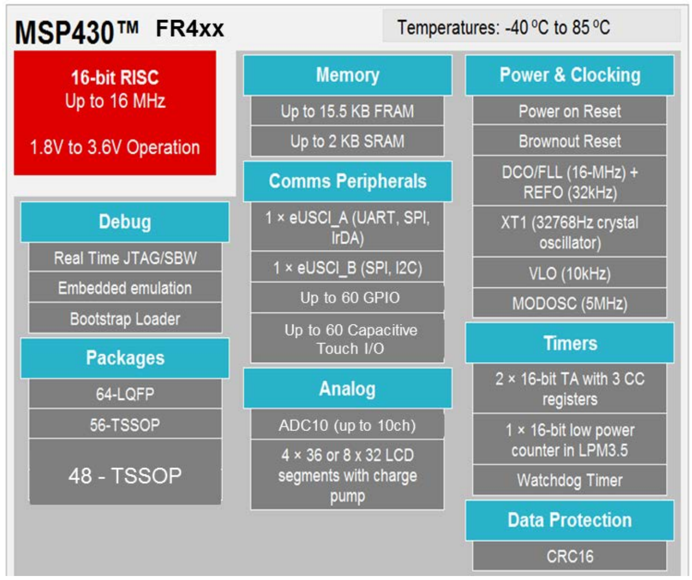
Timers
- Watch Dog Timer (WDT_A)
- Real Time Clock (RTC_B)
- Two 16-bit w/3 CCR (TA0, TA1)
- Two 16-bit w/2 CCR (TA2, TA3)
- One 16-bit w/7 CCR (TB0)

Connectivity
- Up to 40 GPIO (Interrupt/Wake)
- Cap touch IO

Power & Clocking
- Brownout Reset
- Supply Voltage Supervisor (SVS)
- Low Power Vreg (1.5V LDO)
- External Oscillators: LFXT, HFXT
- Internal Oscillators: VLO, DCO (±2%)

MSP430FR4133

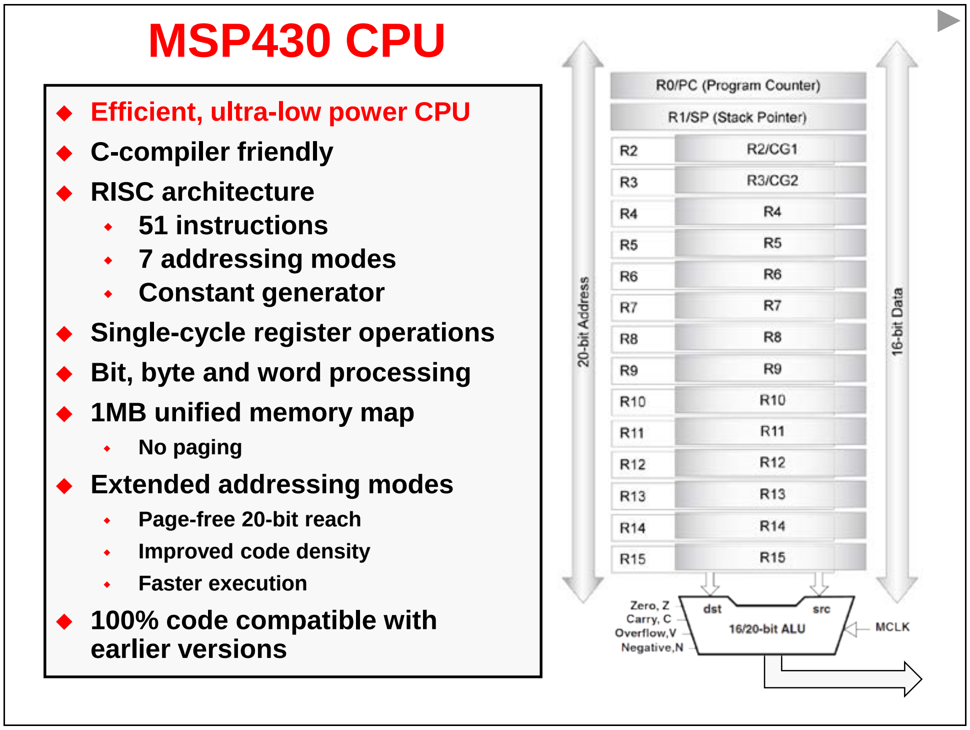
- Integrated LCD
  - Configurable Pin-out
  - On-chip Charge Pump
  - Operates in LPM3.5 ultra low-power mode
  - Integrated LCD driver offers size and system cost advantages
- Up to 60 GPIO pins
- Non-volatile FRAM Technology
  - Flexible use for code or data allocation
  - Ultra-low-power write
  - No external EEPROM needed with write endurance of 10^{15}



These are three of TI's line-up of MSP430 devices – each featuring highly integrated set of peripherals. We will be exploring quite a bit more about them as we go through this workshop.

MSP430 CPU

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



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

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

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

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

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

Bytes, Words And CPU Registers

```
16-bit addition                                     Code/Cycles
5405            add.w   R4,R5                         ; 1/1
529202000202    add.w   &0200,&0202                   ; 3/6

8-bit addition
5445            add.b   R4,R5                         ; 1/1
52D202000202    add.b   &0200,&0202                   ; 3/6
```

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

Seven addressing modes ...

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.

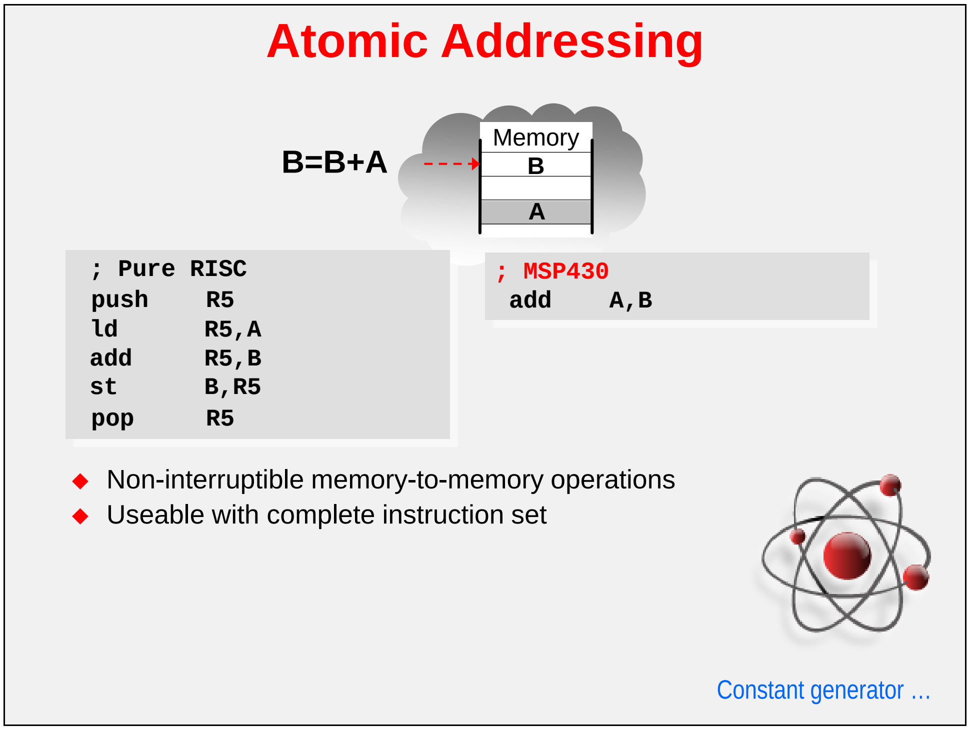
Seven Addressing Modes

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

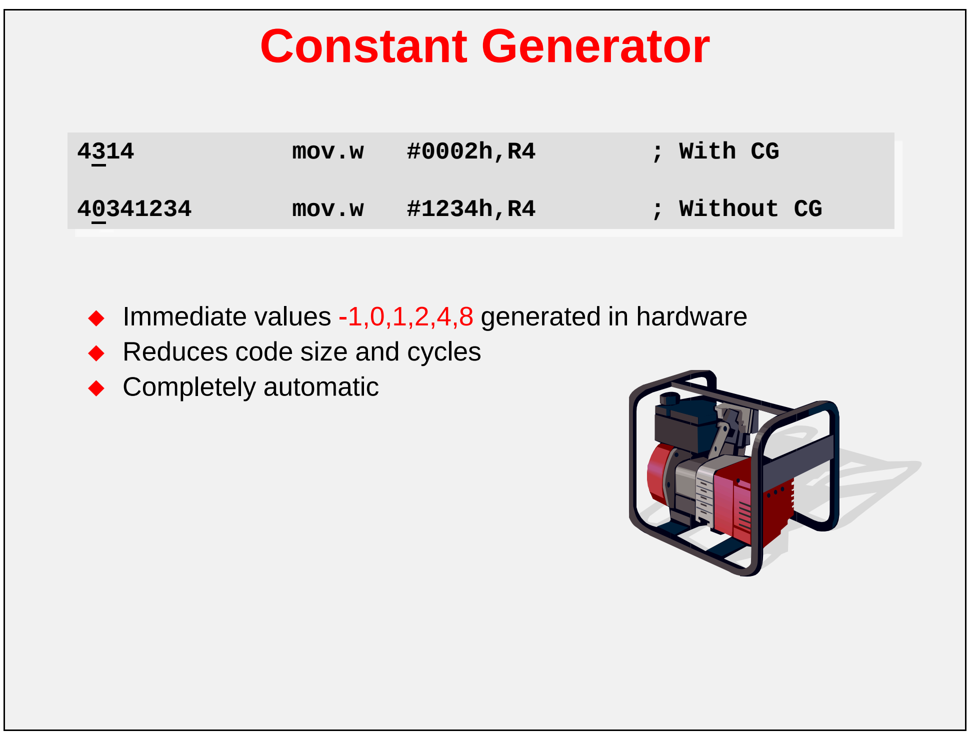
Atomic

Atomic addressing ...

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.



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.



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

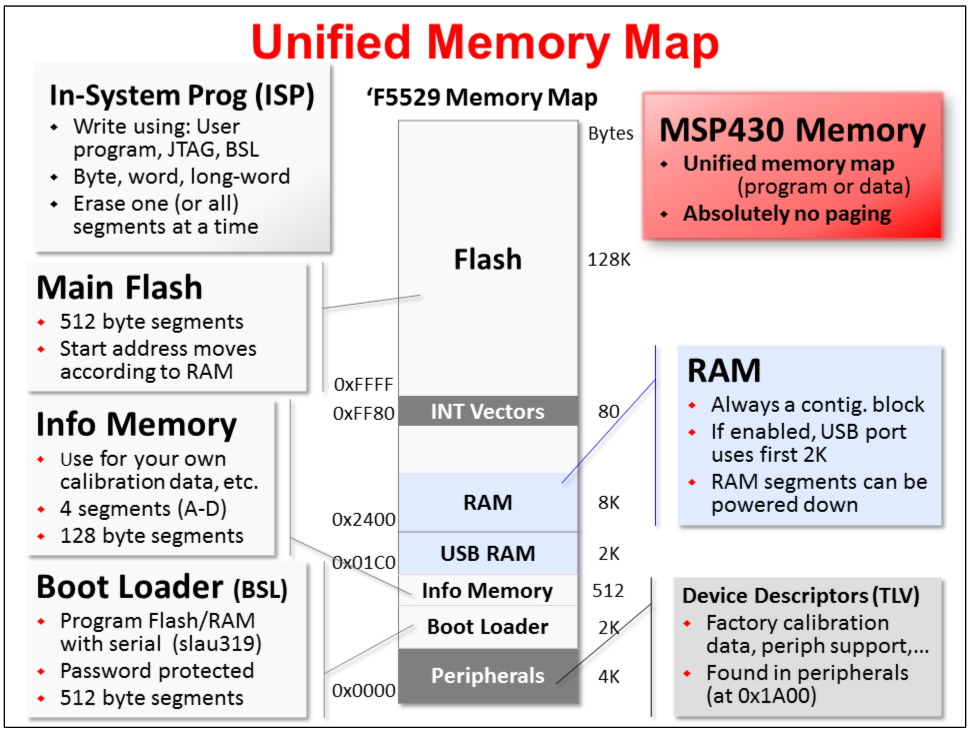
MSP430 Memory

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 though 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 1-14), is implemented using 20-bit addressing; therefore, the MSP430 can directly address the full 1M memory map without resorting to paging schemes. (If you have ever had to deal with paging, we expect you might be cheering at this point.)



Flash

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

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

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

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

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

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

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

RAM

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

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

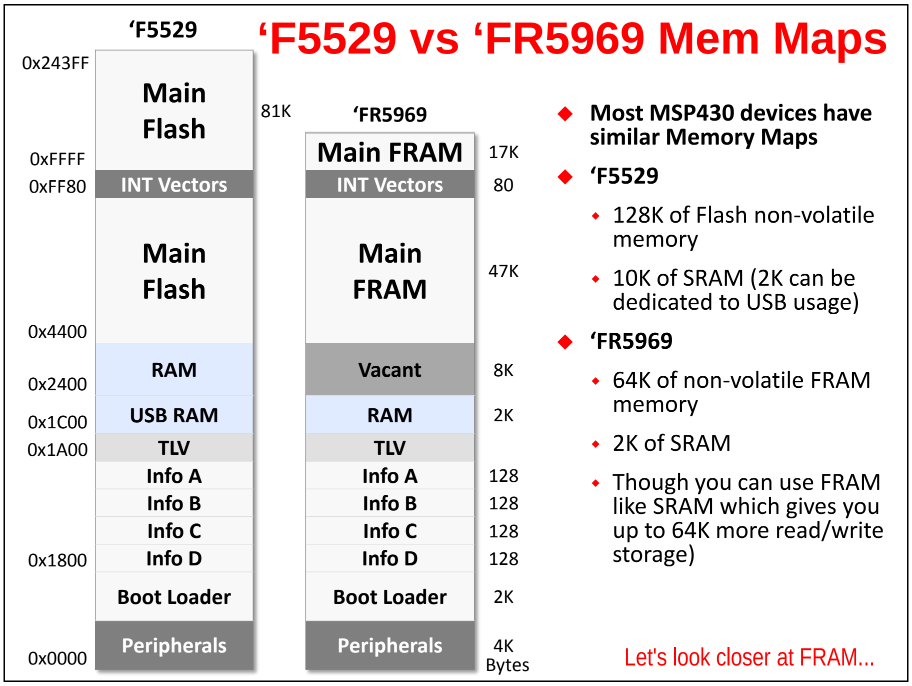
The ‘F5529 has one aspect that is common among MSP430 devices which include the USB peripheral. These devices have an extra 2KB of RAM; this RAM is dedicated to the USB peripheral when it is in use, but available to your programs when the USB port is not being used. Please refer to the USB Developers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

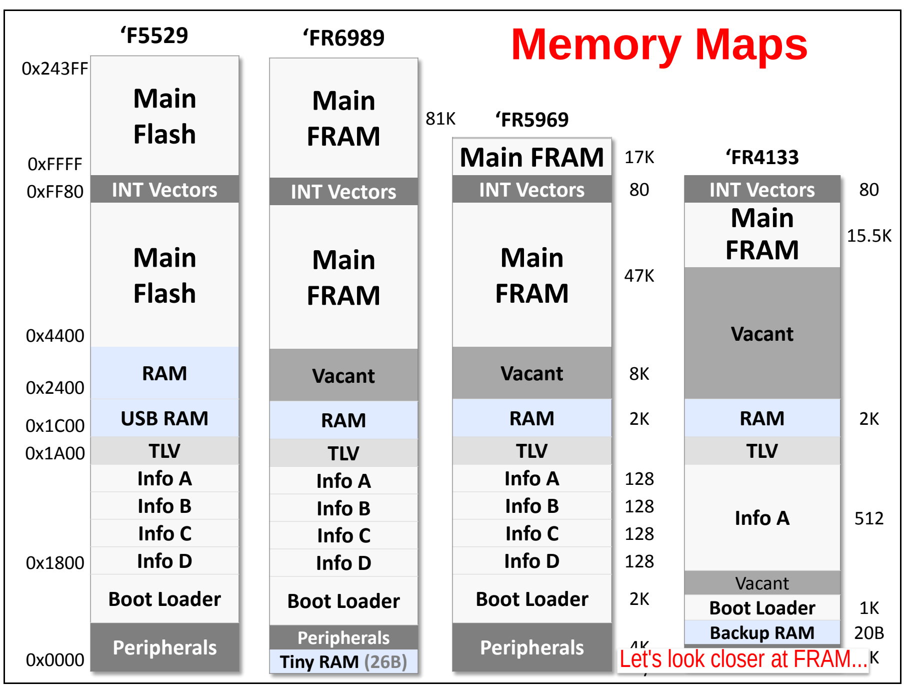
TLV

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

Comparing Memory Maps

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

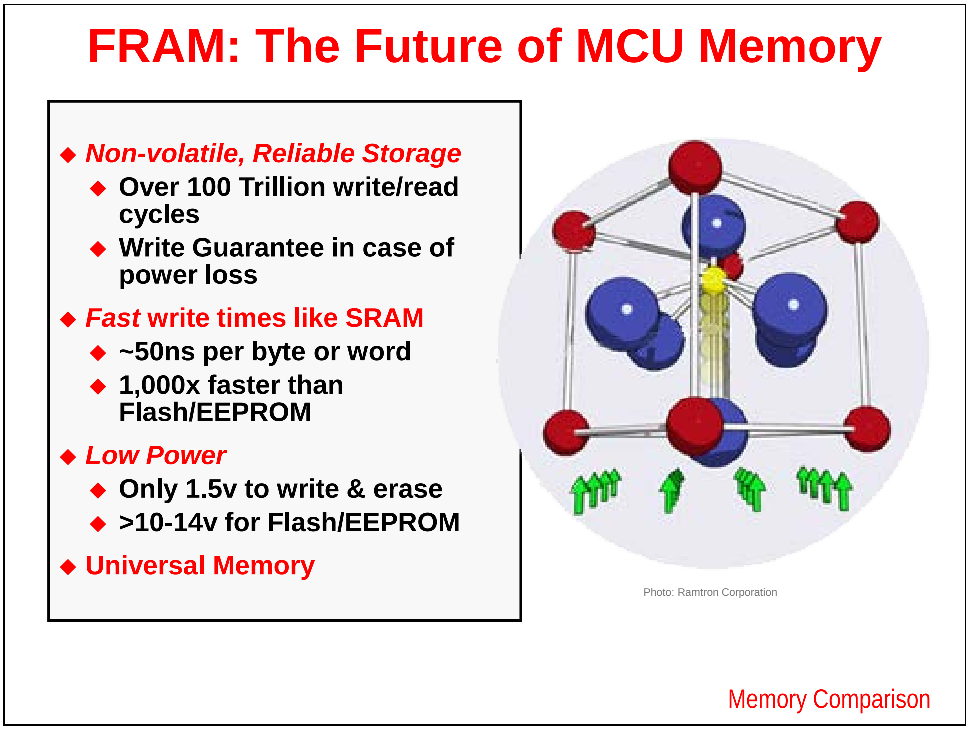




The devices shown here have one other major differentiating factor, the 'F5529 uses *Flash* technology while the 'FR5969 uses *FRAM* technology to store its non-volatile information. We briefly compare these two technologies in the next section, though you may want to refer to the *Non-Volatile Memory (Flash & FRAM)* chapter for more details.

FRAM

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



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.

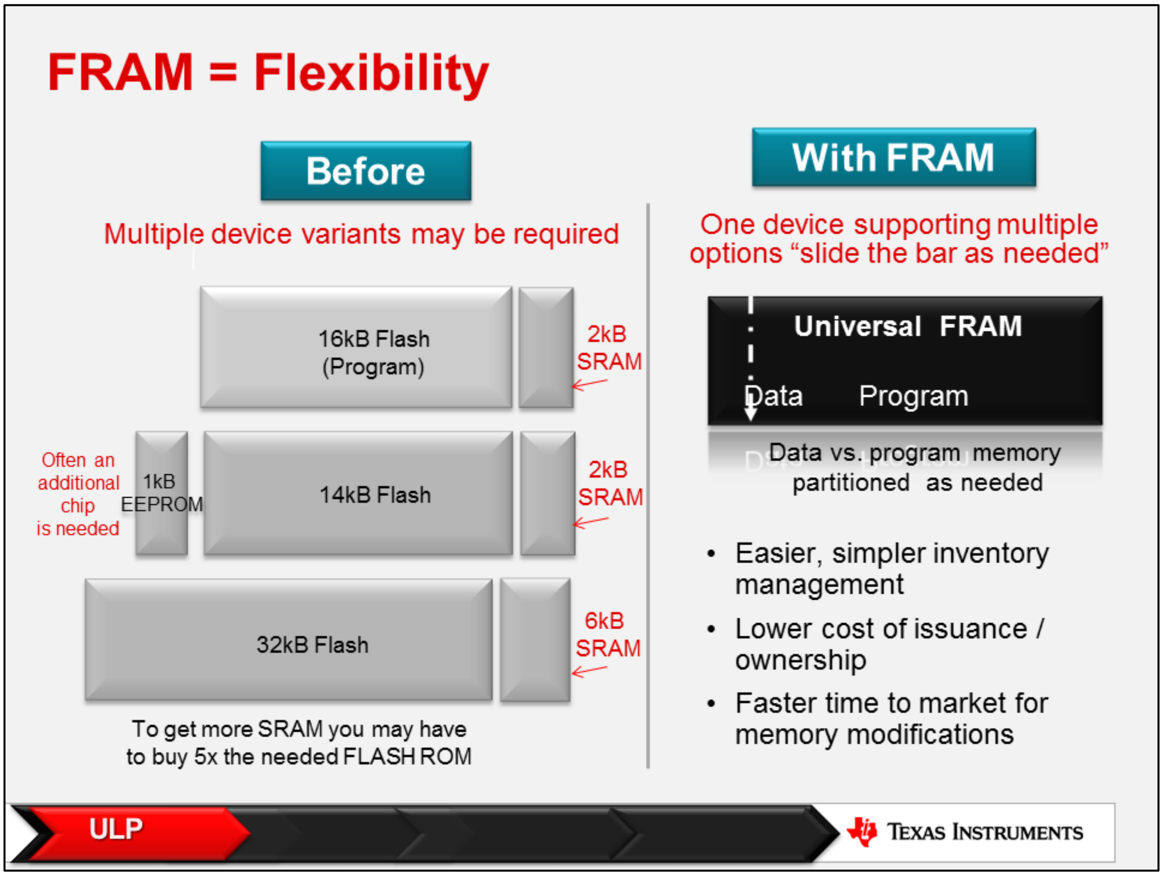
The downside, of course, is that your program could be just as easily overwritten in the same fashion. (We shouldn’t have code that writes to program addresses – but accidents occur.) To this end, the FRAM based devices provide a memory protection unit (MPU) that lets you create 1 to 3 segments of FRAM. 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 *(µA/MHz)*	110	<60	50mA+	230
Write endurance	10^{15}	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.



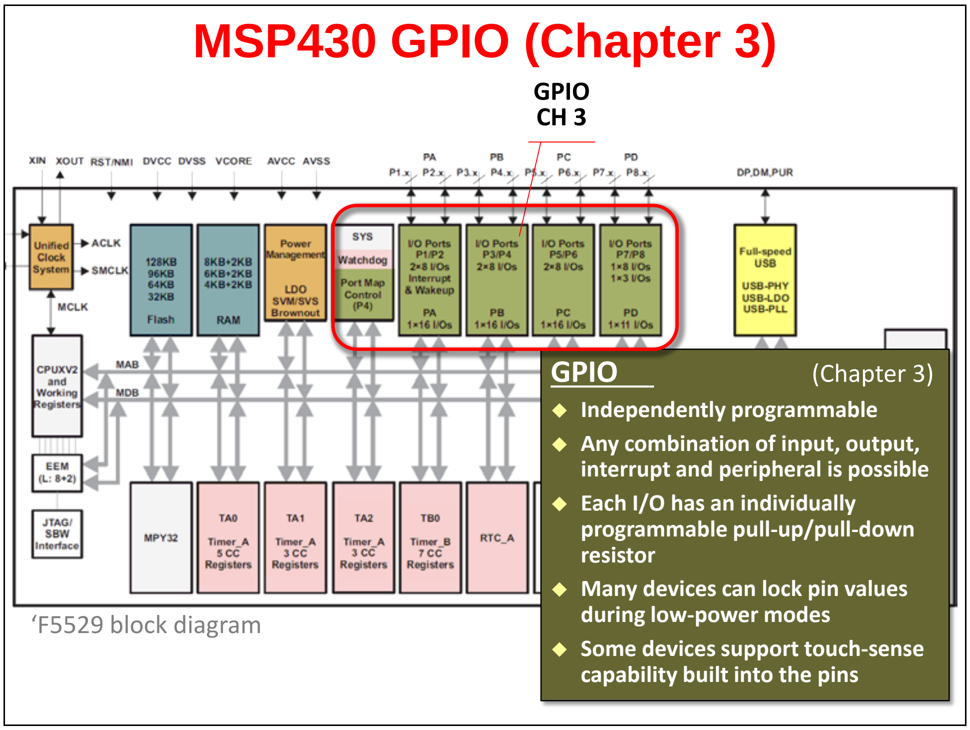
MSP430 Peripherals

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.

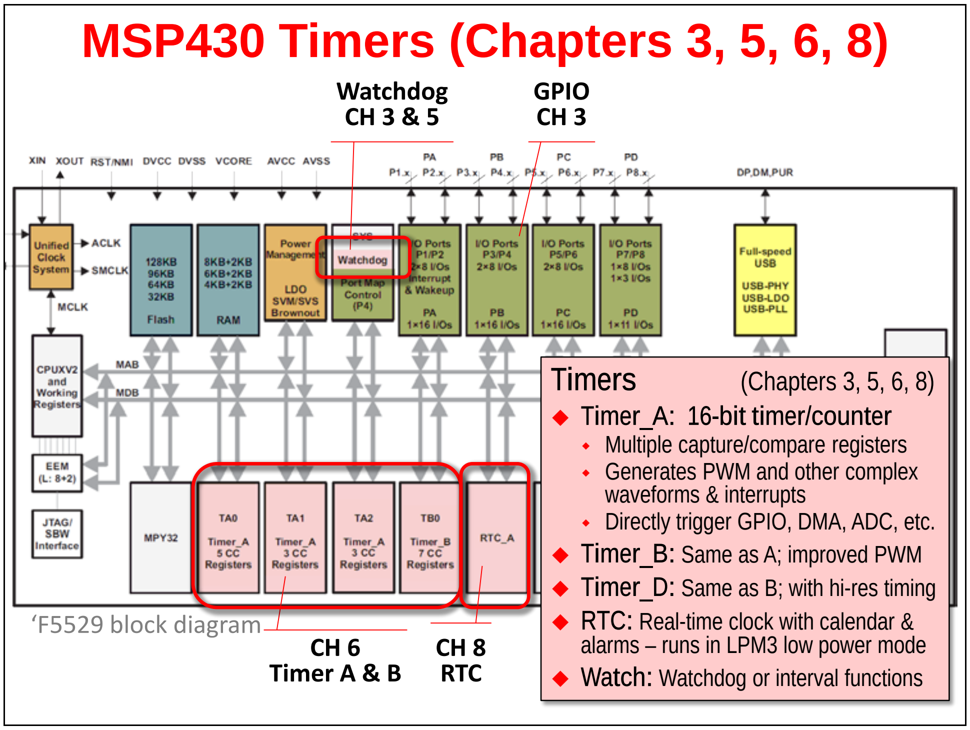
Like most current day microcontrollers, the pins on our devices are heavily multiplexed. That is, you often have one of several choices of signals that can be output to a given pin. The MSP430 makes each signal independently programmable, which affords maximum flexibility.



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

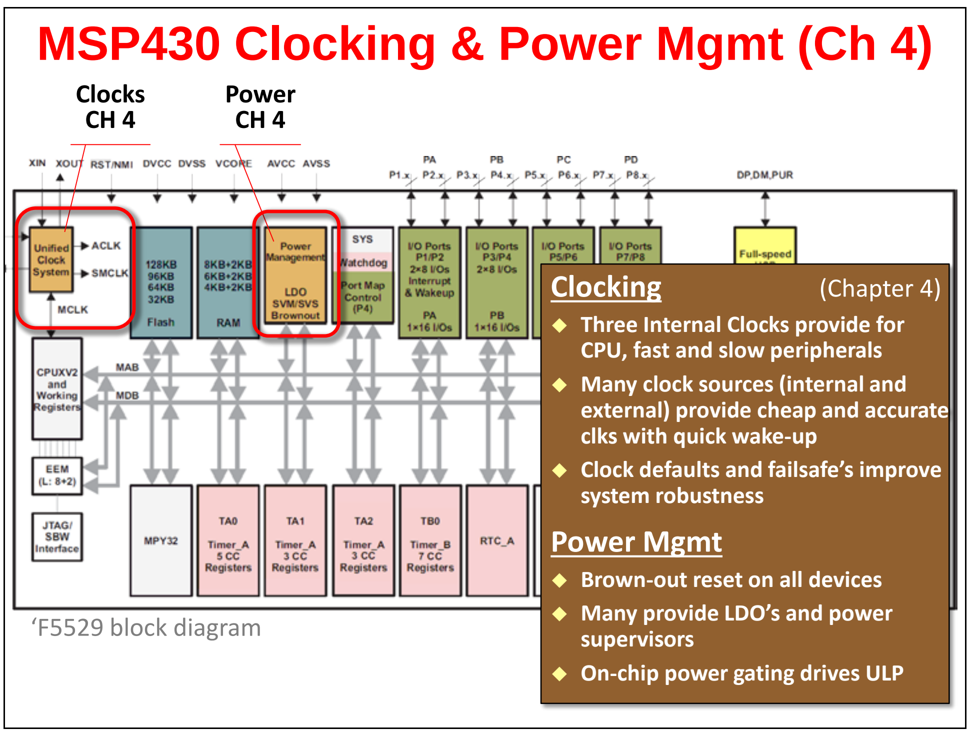
MSP430 Clocks (Chapter 4)

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.

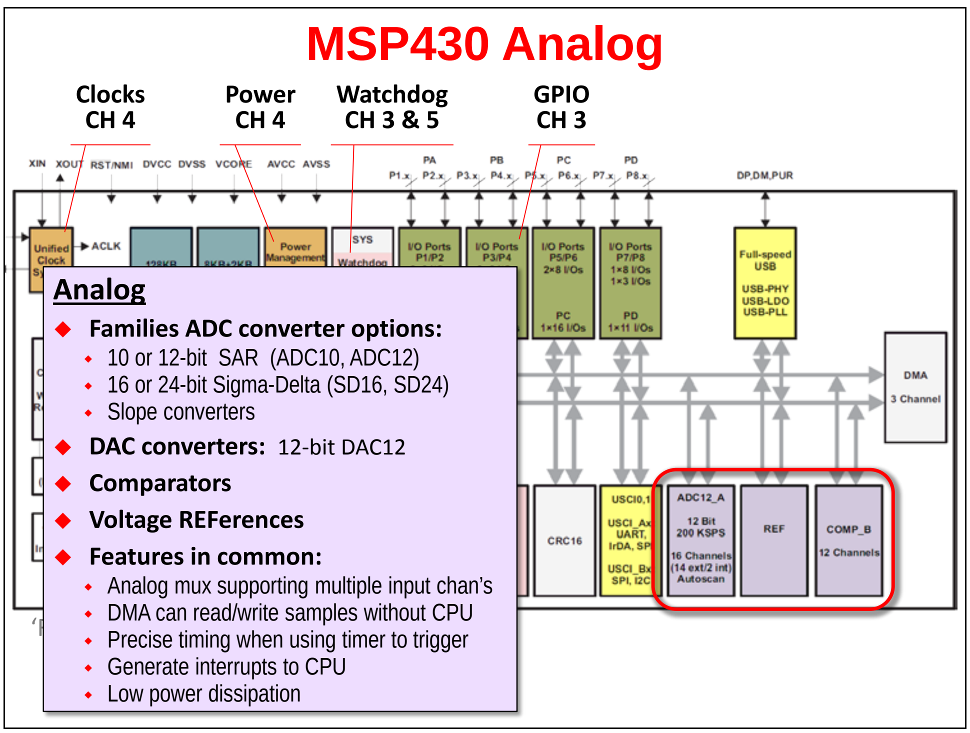
Analog

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.

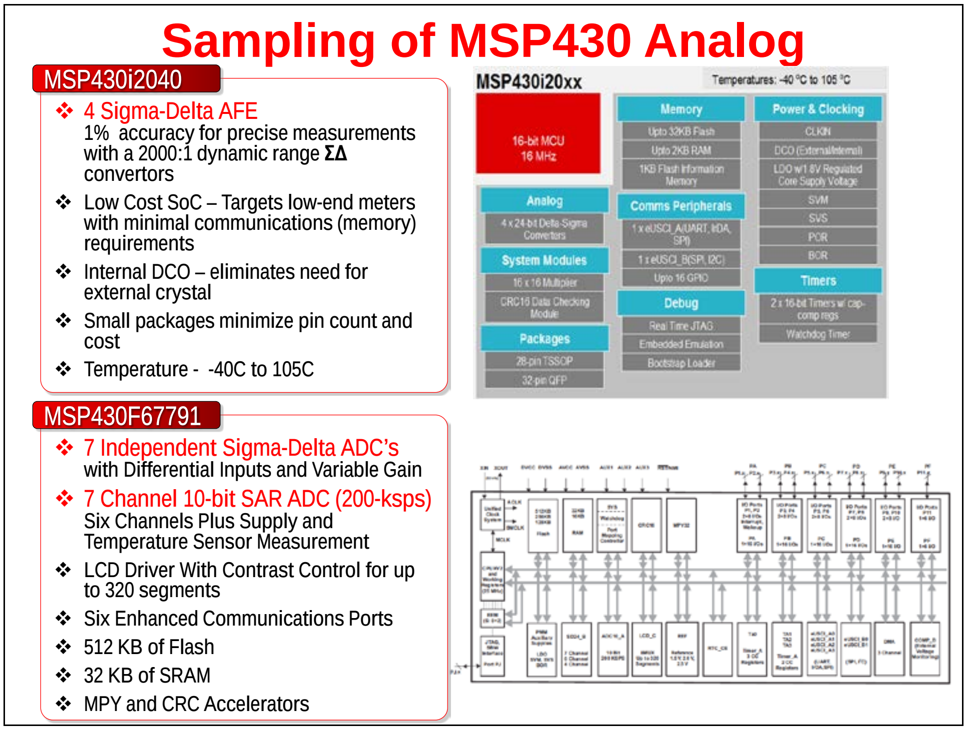
MSP430 analog peripherals cover a wide range of needs. At one end, you'll find most every device contains one or more analog comparators. These signal the processor when an analog input crosses a boundary. (Comparators are often used to build a "poor mans" analog to digital converter.)

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.



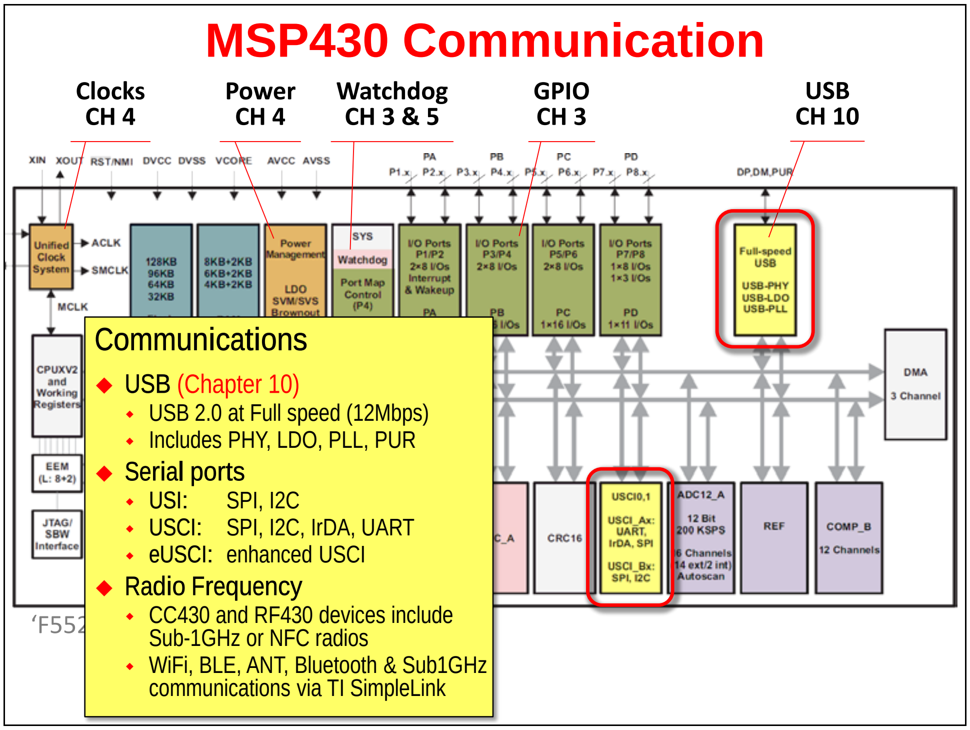
The following slide shows a couple of devices which really show off the MSP430 analog capabilities. The MSP430i2040 provides 4 sigma-delta convertors into a low-cost SOC. The MSP430F67791 packs seven (7) sigma-delta convertors, along with an additional 10-bit SAR analog to digital convertor.



We've seen folks choose these parts just to get access to their highly integrated analog capabilities. The MSP430 CPU being a big bonus! It's like buying a stand-alone convertor and getting the CPU for free!

Communications (Serial ports, USB, Radio)

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



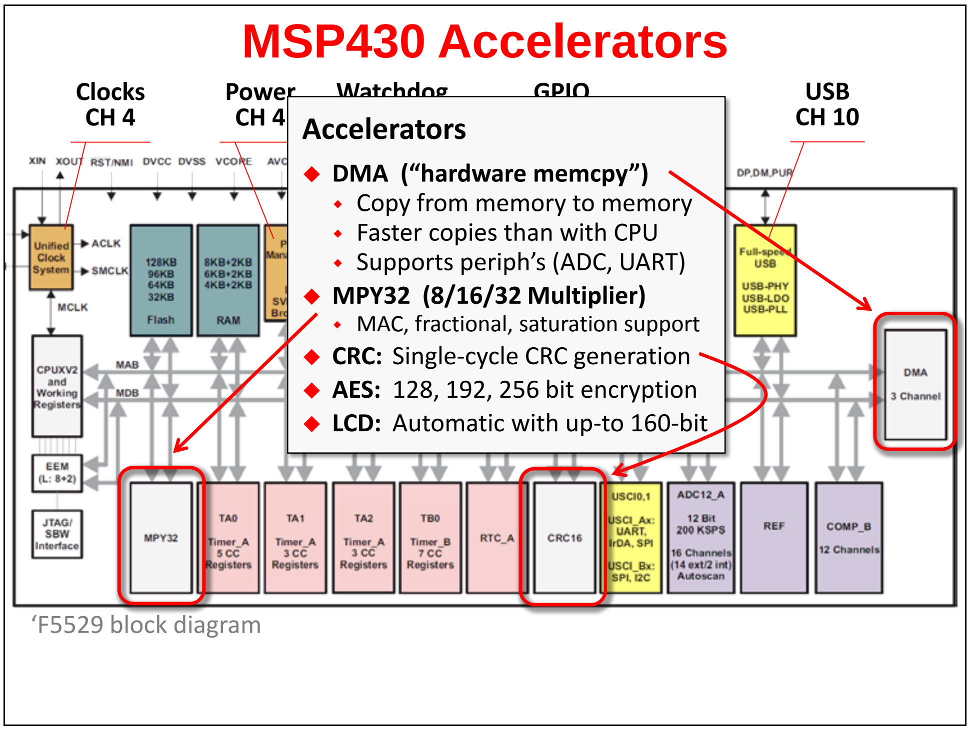
The additional of radios to some MSP430 devices makes them quite unique in the industry. Beyond that, TI has created wireless chips and modules that can be used from any MSP430 device. It's really telling when the cheapest Value Line MSP430 device can actually talk Wi-Fi using TI's CC3000 module. A similar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

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

Hardware Accelerators

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

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

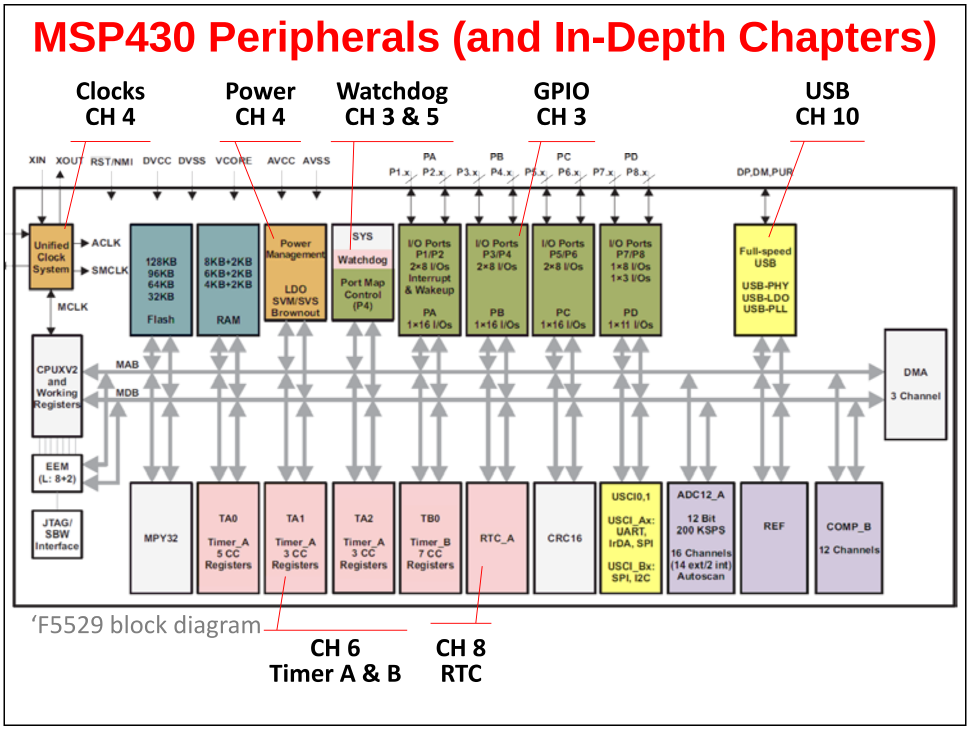


Another example is the multiplier. We can benefit from it without any programming effort, since the compiler automatically uses this hardware, when it's available.

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

Summary

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



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

MSP430 Peripheral Overview

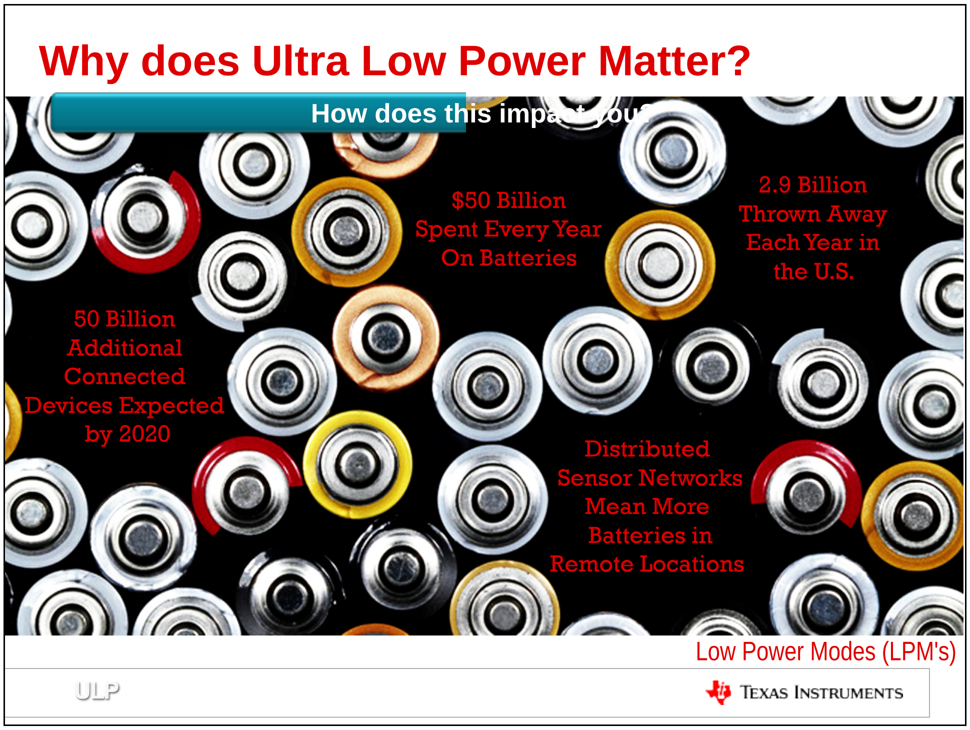
1xx	2xx	4xx	5xx
Basic Clock System	Basic Clock System +	FLL, FLL+	Unified Clock System
Core voltage same as supply voltage (1.8-3.6V)	Core voltage same as supply voltage (1.8-3.6V)	Core voltage same as supply voltage (1.8-3.6V)	Programmable core voltage with integrated PMM (1.8-3.6V)
16-bit CPU	16-bit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

ULP

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

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

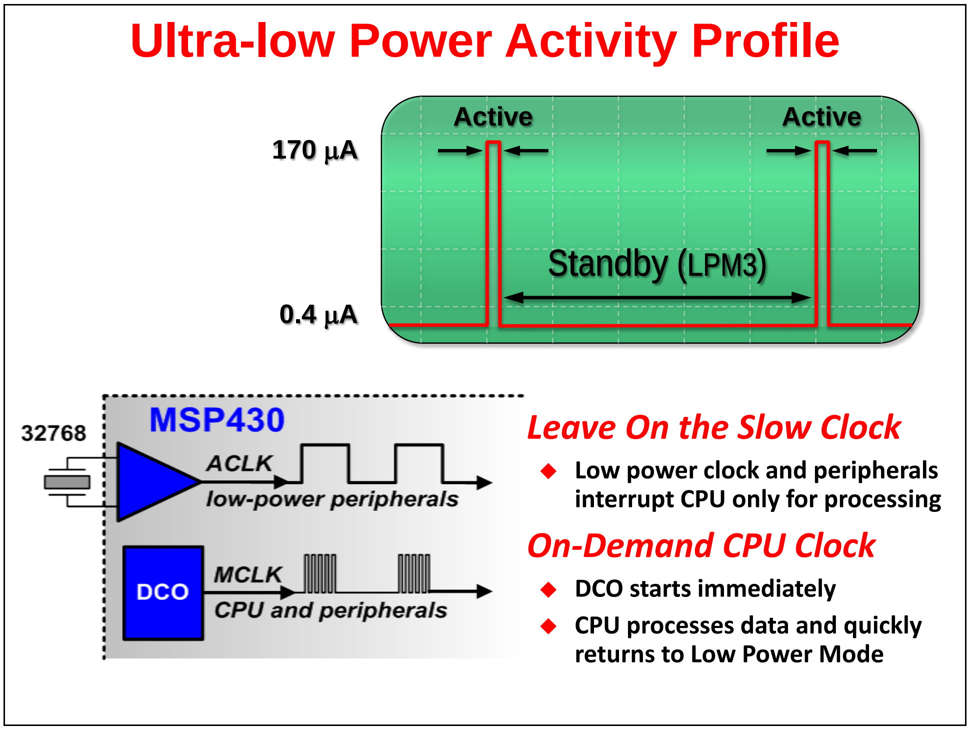
But even when the application does not demand low power, we think it still matters. The trend in electronics over the past few years has been, "Why consume power if you don't have to?" In fact, the MSP430 has found many new applications in the last couple of years where end-users are demanding the reduction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



Profile Your Activities

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

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



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

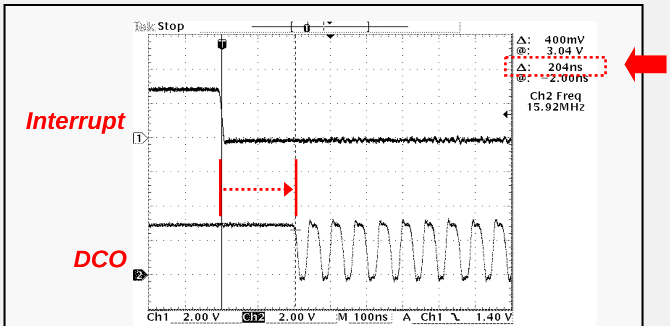
Low-Power Modes

Operating Mode	CPU (MCLK)	SMCLK	ACLK	RAM Retention	BOR	Self Wakeup	Interrupt Sources
Active	☒	☒	☒	☒	☒		Timers, ADC, DMA, WDT, I/0, External Interrupt, COMP, Serial, RTC, other
LPM0		☒	☒	☒	☒	☒	
LPM1		☒	☒	☒	☒	☒	
LPM2			☒	☒	☒	☒	
LPM3			☒	☒	☒	☒	
LPM3.5					☒	☒	External Interrupt, RTC
LPM4				☒	☒		External Interrupt
LPM4.5					☒		External Interrupt

LPM is great, but waking up...

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



Immediate-stable clock start for quick reaction to events

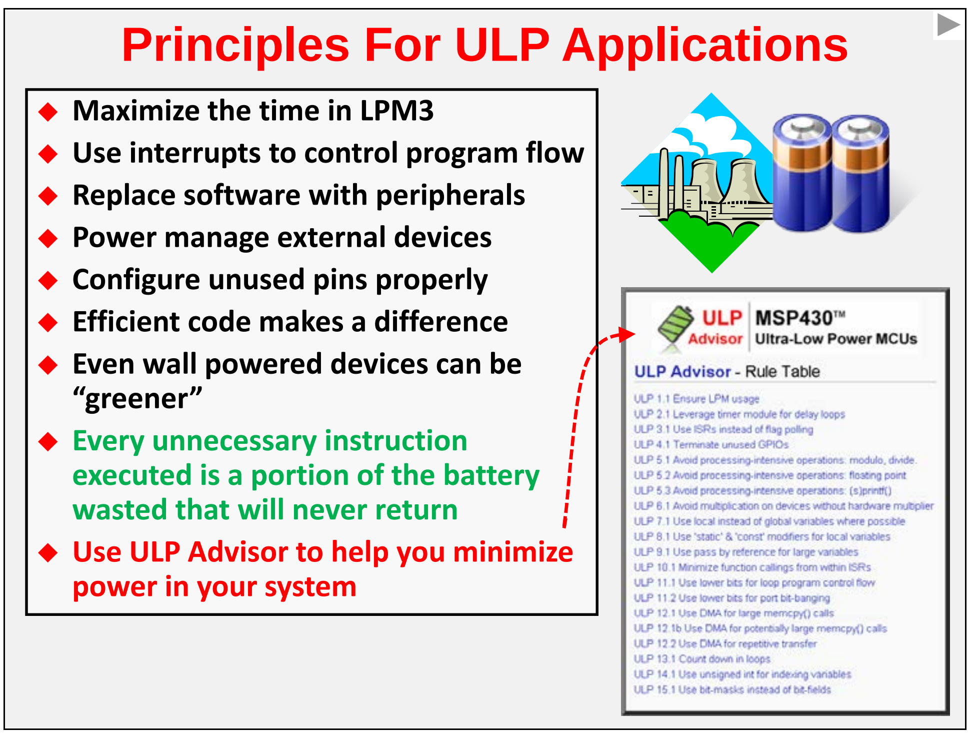
This slide shows some of the quantitative data for different LPM's across a few different devices. Please, keep in mind that you should always design your system by referencing the datasheet, but this slide does give us a good comparison between the various MSP430 generations.

MSP430™ Series Comparison

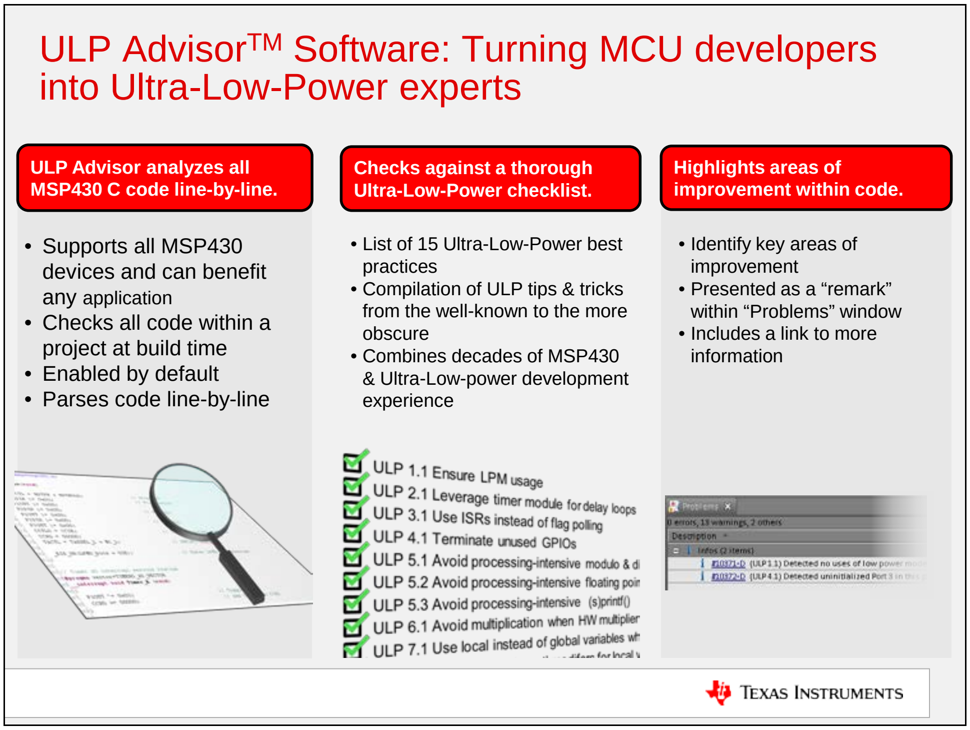
Mode		G2xx	F5xx	FR57xx	FR58xx FR59xx
Performance (max)		16 MHz	25 MHz	24 MHz (FRAM at 8MHz)	16 MHz (FRAM at 8MHz)
Flex Unified Memory		No	No	FRAM (16K)	FRAM (64K)
Active	AM	230 µA (1MHz)	180 µA/MHz	100 µA/MHz	<100 µA/MHz
Standby RTC	LPM3 LPM3.5	0.7 µA	1.9 µA 2.1 µA	6.3 µA 1.5 µA	0.7 µA 0.4 µA
Off	LPM4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

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.



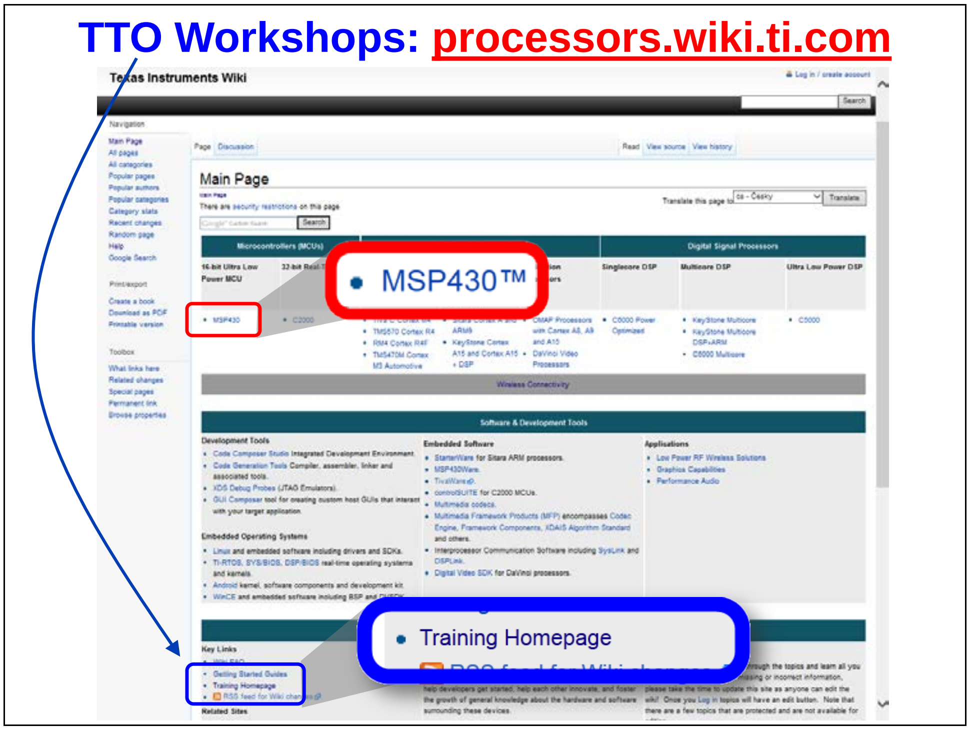
Many of these guidelines have been distilled into a static code analysis tool that is part of the TI (and IAR) compiler. This tool can help us learn what techniques to apply - or for the more experienced, help us not overlook something we already know.



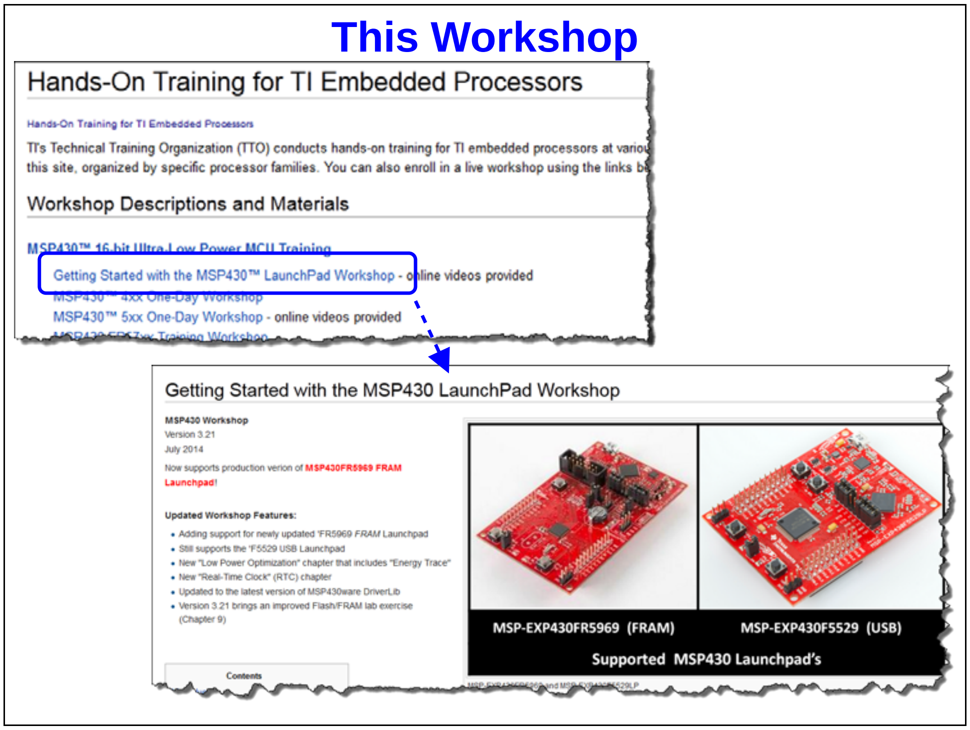
Community / Resources

Wiki

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

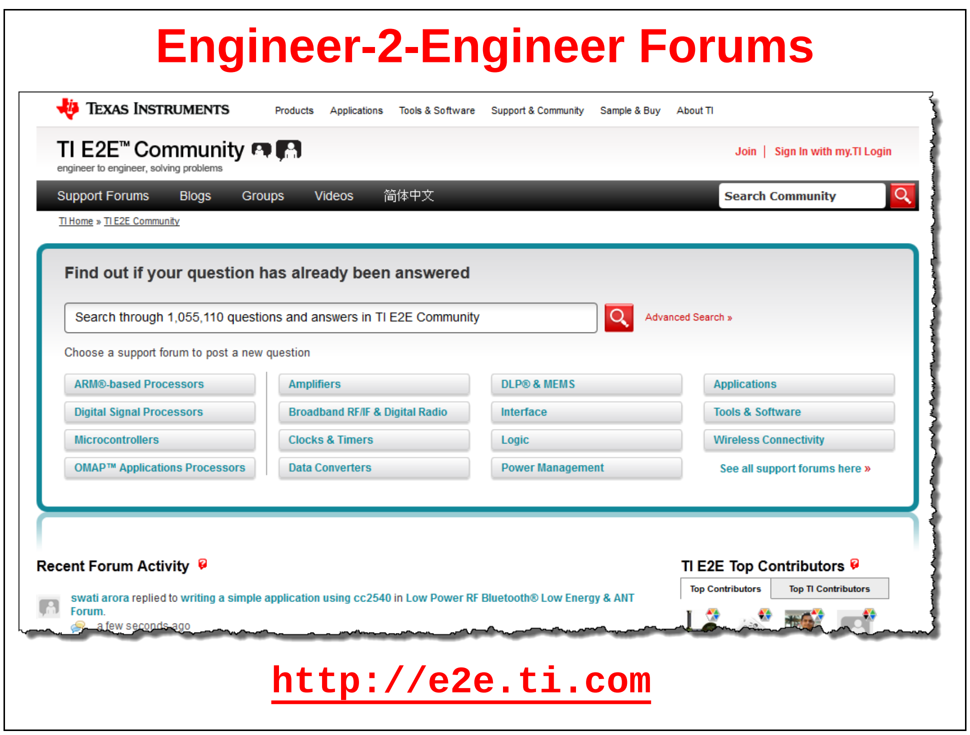


From the TTO wiki page you'll find a link to this workshop. You most likely already found this page when following our download/installation instructions to get ready for the workshop. You may also want to return here often to access updates to these workshop materials.



Forums

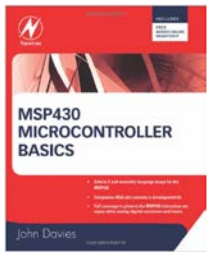
There are a wide ranging set of user-to-user forums. Check them out, when you have a ???



References

There are many great references for learning more about the MSP430. Here's two of them that are favored by a number of us in TI's field applications.

Further Reading…



MSP430 Microcontroller Basics **by John H. Davies,**

(ISBN-10 0750682760) [**Link↗**]

Microcontroller Programming and Interfacing: Texas Instruments MSP430 (Synthesis Lectures on Digital Circuits and Systems)

by Steven Barrett and Daniel Pack ,

(ISBN-10 0750682760) [**Link↗**]

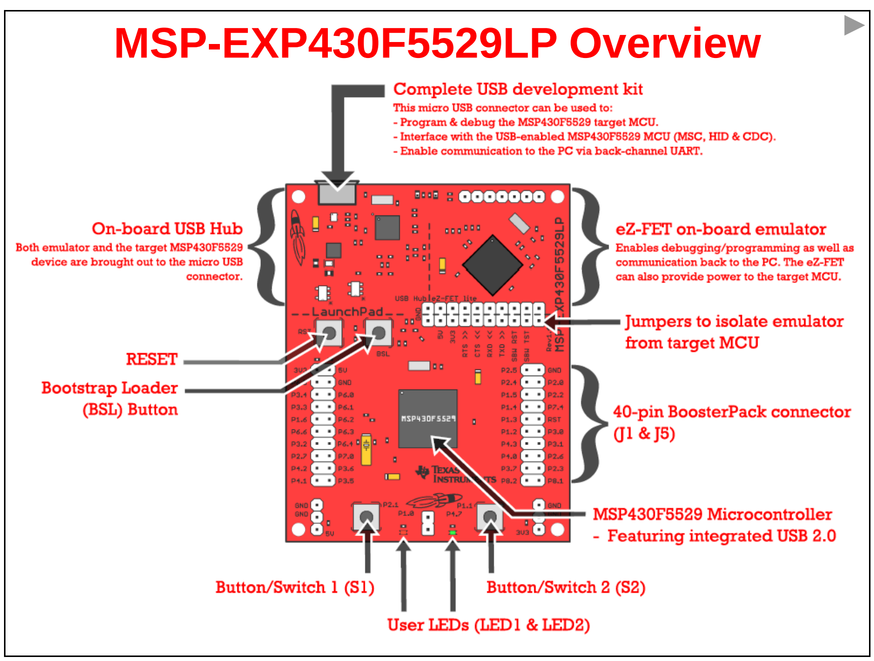
Launchpad's

MSP-EXP430F5529LP Launchpad

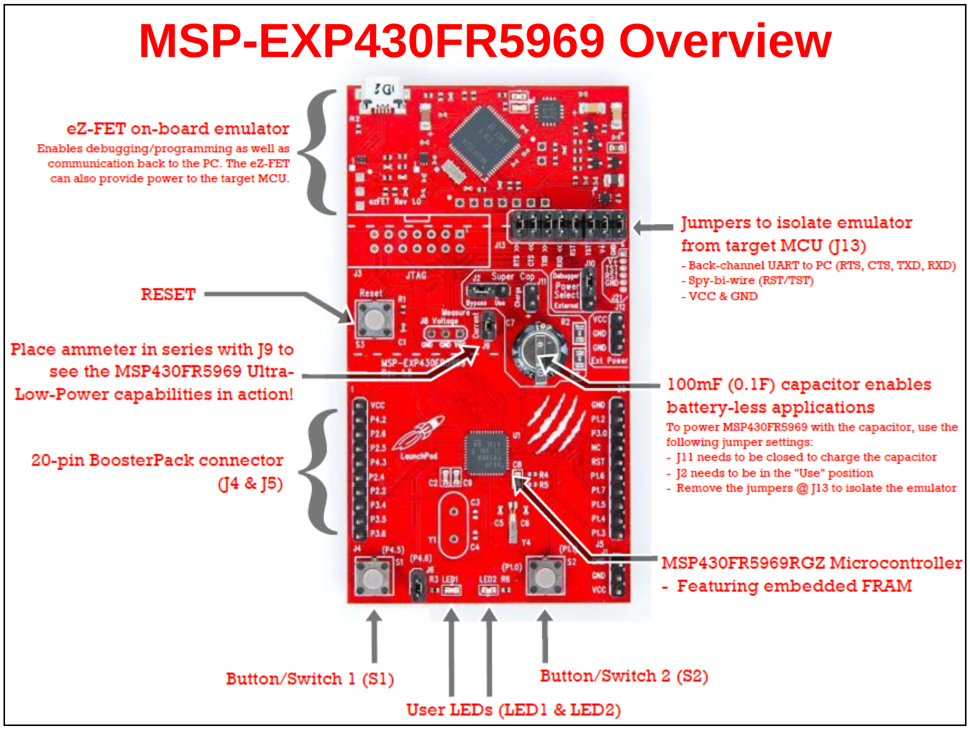
The MSP430F5529 Launchpad is a powerful, low-cost evaluation (and development) tool.

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

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



MSP-EXP430FR5969 Launchpad



MSP-EXP430FR4133 Launchpad

Notes:

Lab: Introduction to the MSP430

Introduction

The first lab exercise in this workshop introduces you to the Launchpad you have selected to work with – running its pre-loaded demonstration program (also called the Out-of-the-Box demo).

Future lab exercises will over-write the original program, but in Lab 2c and 2d we will show you how to restore the original Out-of-the-Box demo, should you want to do so.

Lab 1 Topics

Lab: Introduction to the MSP430 1-43

Lab 1a – MSP-EXP430F5529LP User Experience 1-44
Examine the LaunchPad Kit Contents 1-44

Lab 1b – MSP-EXP430FR5969 LaunchPad OOB 1-46
First Steps – Out-of-Box Experience 1-46
(‘FR5969) Extra Credit 1-51

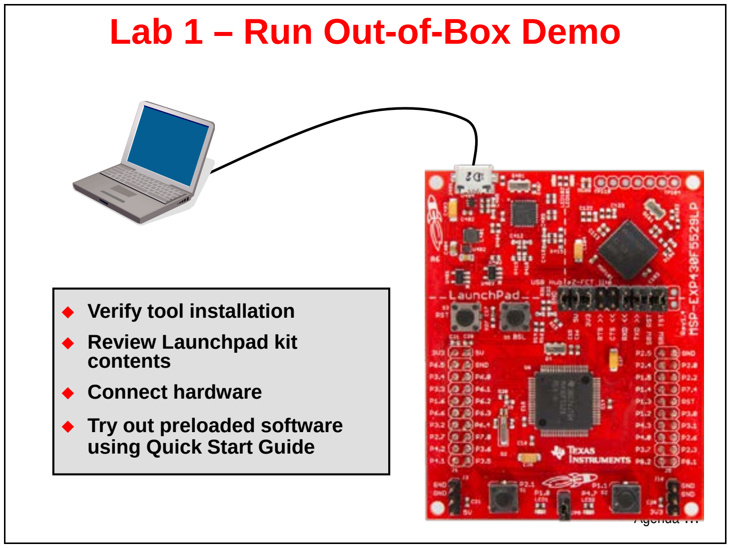
Lab 1c – MSP-EXP430FR4133 LaunchPad OOB 1-52

Lab 1a – MSP-EXP430F5529LP User Experience

'FR5969 FRAM Launchpad users should jump to Lab 1b on page 1-46.
'FR4133 FRAM Launchpad users should jump to Lab 1c on page 1-52.

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

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



Examine the LaunchPad Kit Contents

1. **Open up your MSP430F5529 LaunchPad box. You should find the following:**
   – The MSP-EXP430F5529LP LaunchPad Board
   – USB cable (A-male to micro-B-male)
   – "Meet the MSP430F5529 Launchpad Evaluation Kit" card

2. **Initial Board Set-Up**

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

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

3. **Run the User Experience Application**

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

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

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

4. View the contents of the emulated hard drive

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

– **Button1.txt** – the contents of this file are "typed out" to the PC, using the emulated keyboard when you press button S1

– **Button2.txt** – the contents of this file are "typed out" to the PC, using the emulated keyboard when you press button S2

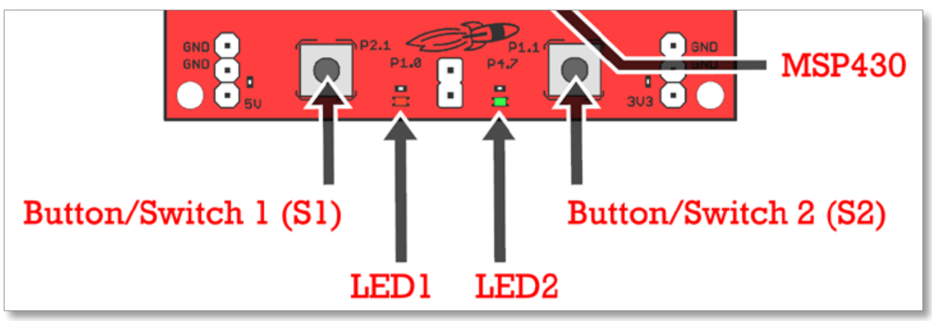
– **MSP430 USB LaunchPad.url** – when you double-click, your browser launches the MSP- EXP430F5529LP home page

– **README.txt** – a text file that describes this example

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

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

```
Open Notepad. In the start menu, type “Run”, then type “Notepad”

To send the strings to Notepad, press S1.
```



What do you see? ____

Now press S2. What happens now? ________________________________

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

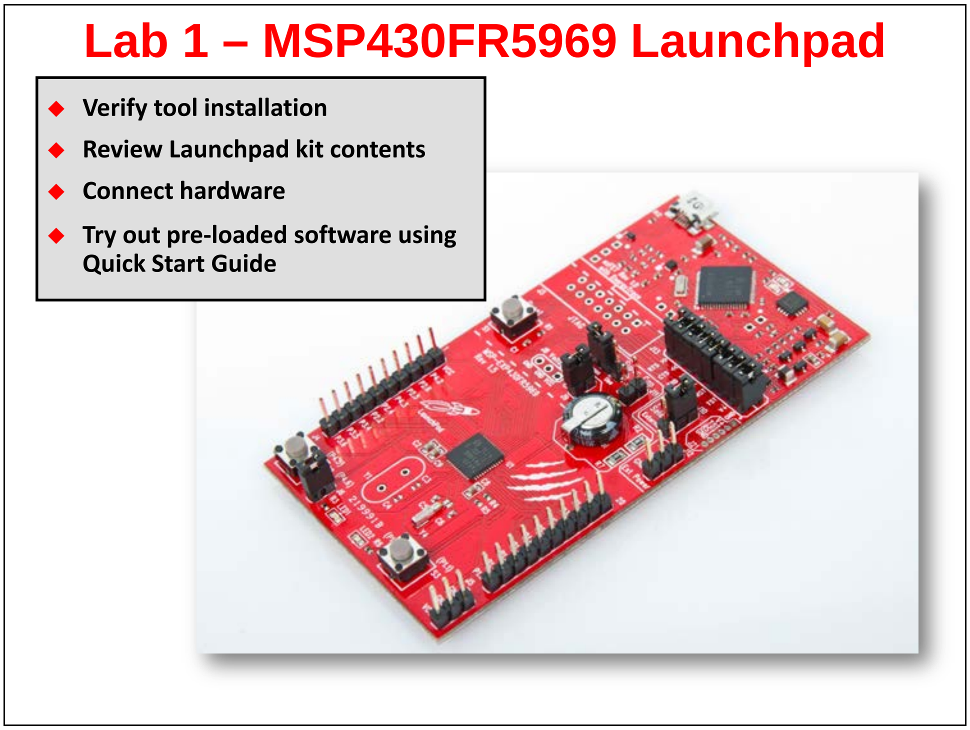
– **Button1.txt:** "Hello world"

– **Button2.txt:** an ASCII-art picture of the LaunchPad rocket

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

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

Lab 1b – MSP-EXP430FR5969 LaunchPad OOB



First Steps – Out-of-Box Experience

These steps were taken from Section 1.4 and 3.0 of the *MSP-EXP430FR5969 LaunchPad™ User's Guide* (slau535a.pdf).

An easy way to get familiar with the EVM is by using its pre-programmed out-of-box demo code, which demonstrates some key features of the MSP-EXP430FR5969 LaunchPad.

The out-of-box demo showcases MSP430FR5969's ultra-low power FRAM by utilizing the device's internal temperature sensor while running only off of the on-board Super Capacitor.

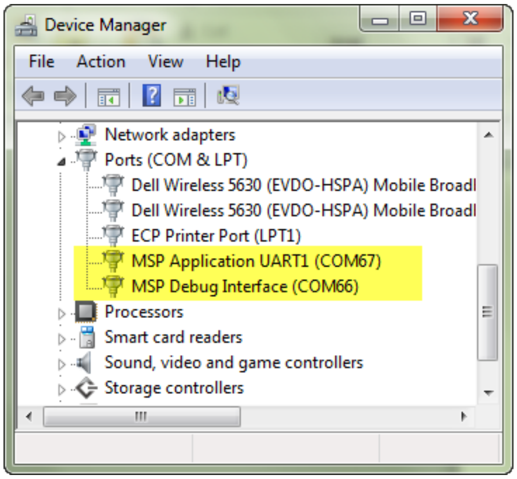
1. **First step is to connect the LaunchPad to your computer using the included Micro-USB cable.**

   The RED and GREEN LEDs near the bottom of the LaunchPad toggle a few times to indicate the preprogrammed out-of-box demo is running.

   After the LEDs toggle, the MSP430FR5969 CPU enters low-power mode 3 and waits for commands to come from the PC GUI via the backchannel UART. (A backchannel UART is the name given the UART to USB connection where the UART signals on the MSP430 are turned into a USB CDC class protocol by the MSP430 emulator.)

   The Out-of-Box GUI is required to connect to the serial port that the LaunchPad's UART communication uses. But, to use the GUI we need to know which COM port our Launchpad was assigned to by Windows.

2. Open Windows Device Manager and find the two COM ports assigned to the MSP430 Launchpad.



Write down the two ports listed on your computer.

MSP Application UART1: ________________

MSP Debug Interface: ___________________

3. **Start the out-of-box demo GUI.**

Using the out-of-box demo GUI, the user can place the LaunchPad into two different modes.

- **Live Temperature Mode**

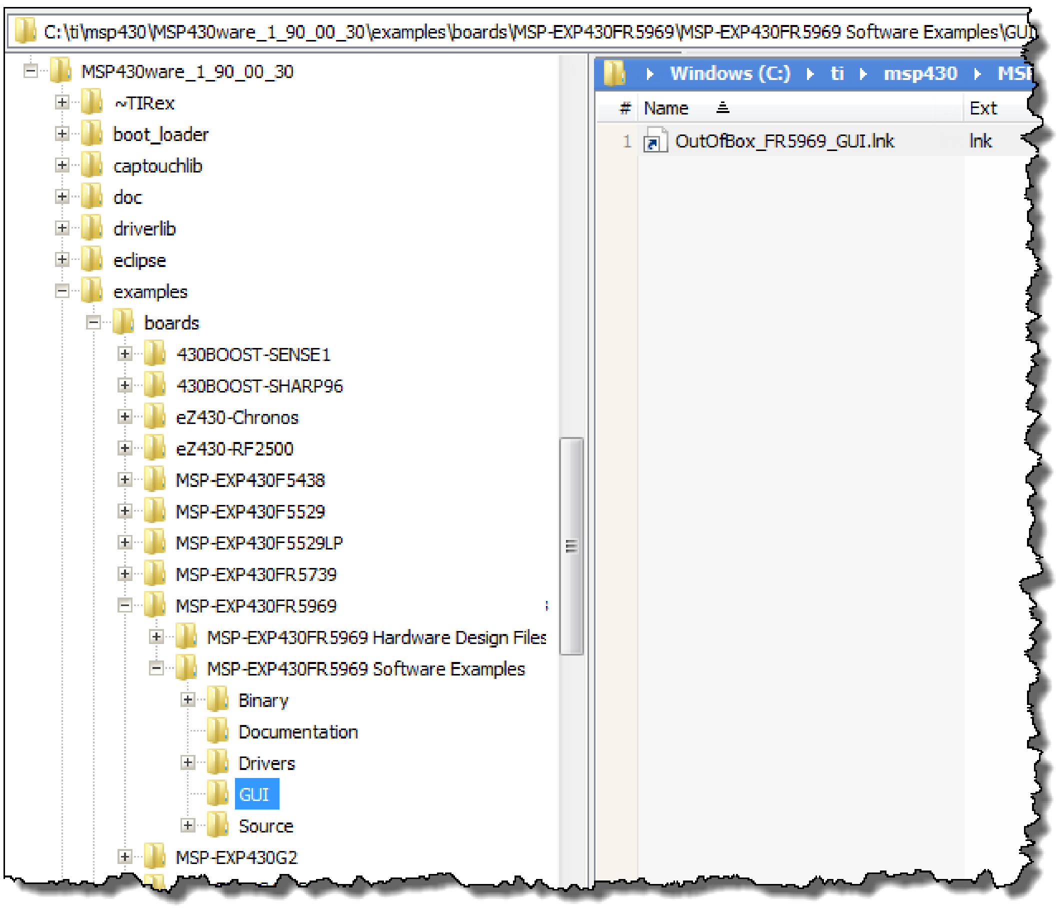
  This mode provides live temperature data streaming to the PC GUI. The user is able to influence the temperature of the device and see the changes on the GUI.

- **FRAM Logging Mode**

  This mode shows the FRAM data logging capabilities of the MSP430FR5969. After starting this mode, the LaunchPad will wake up every five seconds from sleep mode (indicated by LED blink) to log both temperature and input voltage values. After reconnecting to the GUI, these values can be uploaded and graphed in the GUI.

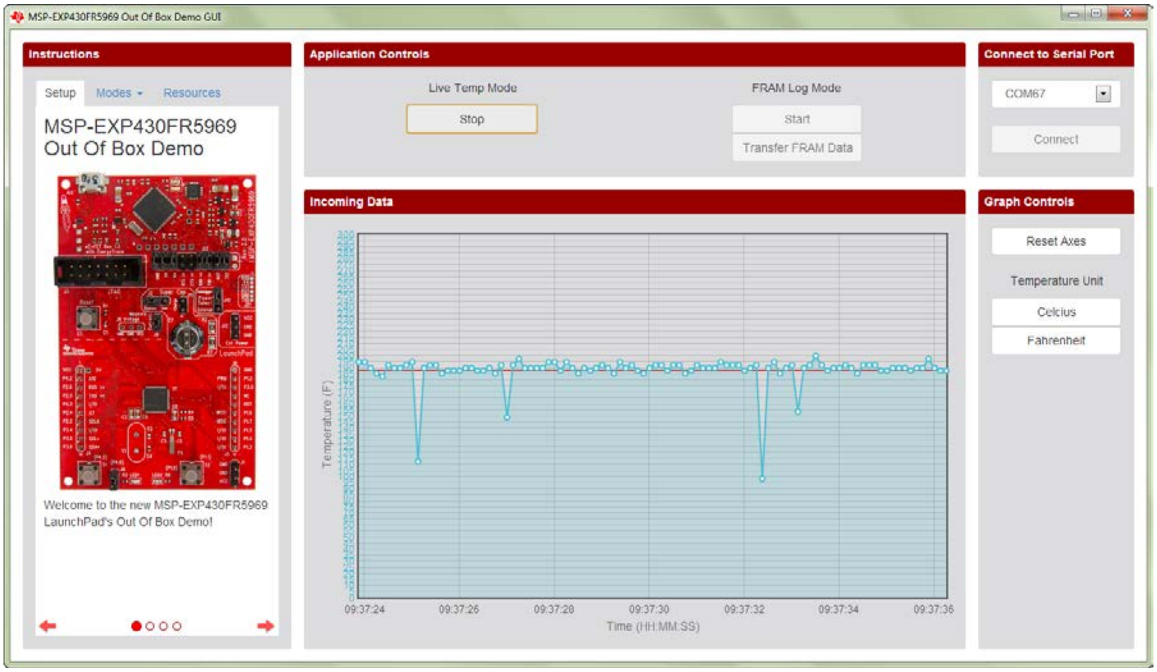
The easiest way to **start the GUI** is to double-click the link found in the MSP430ware library folder.

```
C:\ti\msp430\MSP430ware_1_97_00_47\examples\boards\MSP-EXP430FR5969\MSP-EXP430FR5969 Software Examples\GUI\OutOfBox_FR5969_GUI.lnk
```



The Out-of-Box example and GUI are included in the latest version of MSP430ware (as we mentioned earlier) as well as the MSP-EXP430FR5969 Software Examples download package (SLAC645).

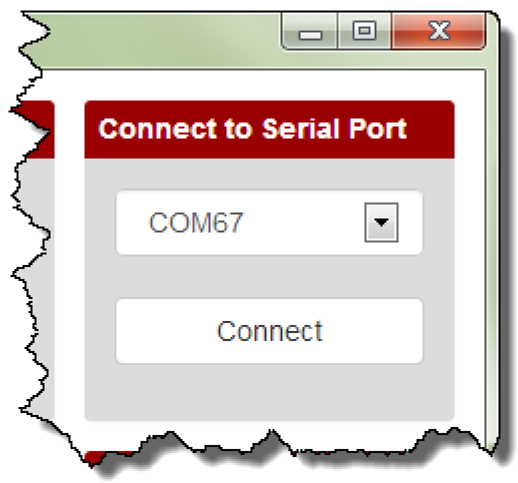
Here's a snapshot of the GUI.



4. Connect the GUI to your Launchpad.

To get it to display data, we first need to connect with it.

Select the "MSP Application UART1" communications port from the list and click the *Connect* button.



5. **Once connected, to enter the live temperature mode, click the "Start" button below "Live Temp Mode" in the GUI's *Application Controls* panel.**

   At this point, you should see the graph of temperature data populating the *Incoming Data* panel.

   What is 'FR5969 Doing?

   It sets up its 12-bit ADC for sampling and converting the signals from its internal temperature sensor. A hardware timer is also configured to trigger the ADC conversion every 0.125 seconds before the device enters low-power mode 3 to conserve power. As soon as the ADC sample and conversion is complete, the raw ADC data is sent the through the UART backchannel to the PC GUI.

   As the raw ADC data is received by the PC GUI, Celsius and Fahrenheit units are calculated first. The PC GUI keeps a buffer of the most recent 100 temperature measurements, which are graphed against the PC's current time on the Incoming Data panel.

   A red horizontal line is drawn across the data plot to indicate the moving average of the incoming data.

6. **To exit *Live Temp* mode, click the "Stop" button under "Live Temp Mode". You must exit this mode before starting the FRAM Log Mode.**

7. **To enter the *FRAM Log Mode*, click the "Start" button under "FRAM Log Mode" in the GUI's *Application Controls* panel.**

   When the MSP430FR5969 receives the UART command from the GUI, it starts the entry sequence by initializing the Real-Time Clock to trigger an interrupt every 5 seconds. The red LED blinks three times to indicate successful entry into FRAM Log Mode.

   Unlike in the Live Temperature Mode, the MSP430FR5969 enters low-power mode 3.5 to further decrease power consumption and wakes up every 5 seconds to perform data logging. Because the UART communication module does not retain power in LPM3.5, the GUI automatically disconnects from the LaunchPad after entry into FRAM Log Mode.

   Each time the device wakes up, the green LED lights up to indicate its state to the user. The 12-bit ADC is set up to sample and convert the signals from its internal temperature sensor and battery monitor (Super Cap voltage).

   A section of the device's FRAM is allocated to store the raw ADC output data (address 0x9000 to 0xEFFF). This allows the demo to store up to 6144 temperature and voltage data points (5 seconds/sample is approximately 8.5 hours of data).

8. **To exit the *FRAM Log Mode*, press the S2 (right) push button on the LaunchPad.**

   The red LED turns on briefly to indicate successful exit.

   The LaunchPad returns to the Power up and Idle state and you can reconnect the LaunchPad with the GUI to transfer the logged data from FRAM to the PC.

9. **Make sure the Launchpad is connected to the GUI and click the "Transfer FRAM Data" button in the GUI to begin transfer.**

   A progress bar shows progress until the transfer completes, and the temperature and voltage data are plotted in the Incoming Data panel.

(‘FR5969) Extra Credit

Open up the *MSP-EXP430FR5969 LaunchPad™ User’s Guide* (slau535a.pdf) to section “2.4.5 Super Cap”. Try using the FRAM Log Mode while powered from the Super Cap.

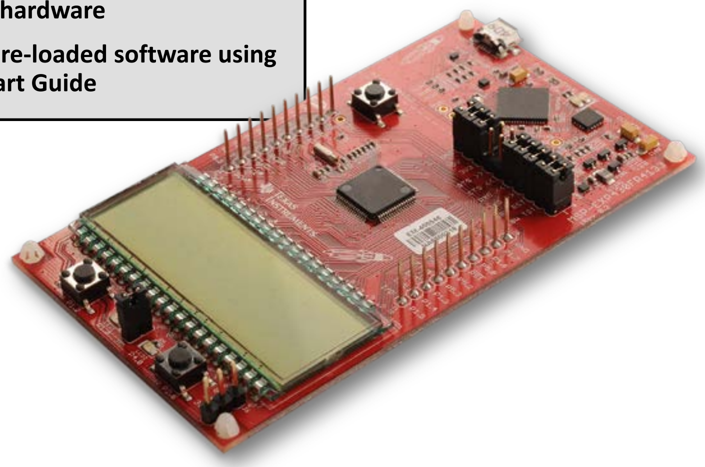
> *The FRAM Log Mode also provides the option to log temperature data while powered either through the USB cable or only by the on-board Super Cap. The PC GUI contains step-by-step instructions in its side panel for configuring the jumpers on the LaunchPad to power the device with the Super Cap.*

Hint: We suggest that you look carefully at the initial jumper locations so that you can easily return the jumpers to their original locations after playing with the Super Cap.

Lab 1c – MSP-EXP430FR4133 LaunchPad OOB

Lab 1c – MSP430FR4133 Launchpad

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



Out-of-box Demo

1. Connecting to the computer
Connect the LaunchPad using the included USB cable to a computer. A green power LED should illuminate. For proper operation, drivers are needed. It is recommended to get drivers by installing an IDE such as TI's CCS or IAR EW430. Drivers are also available at ti.com/MSPdrivers.

2. Running the Out-of-box Demo
When connected to your computer, the LaunchPad will power up and display a greeting message on the LCD. Press and hold the S1 and S2 buttons simultaneously to select a new mode.

Stopwatch Mode
This mode provides a simple stopwatch application. It supports split time, where the display freezes while the stopwatch continues running in the background.

Timer Stopped:
S1 - Start time
S2 - Reset time
Timer Running:
S1 - Stop time
S2 - Split time (lap time)

Temperature Mode
This mode provides a simple thermometer application. Using the on-chip temperature sensor, the temperature is displayed on the LCD.

S1 - Pause current temperature
S2 - Toggle temperature between °F/C

These steps were taken from the MSP-EXP430FR4133 LaunchPad™ Quick Start Guide (slau594.pdf)

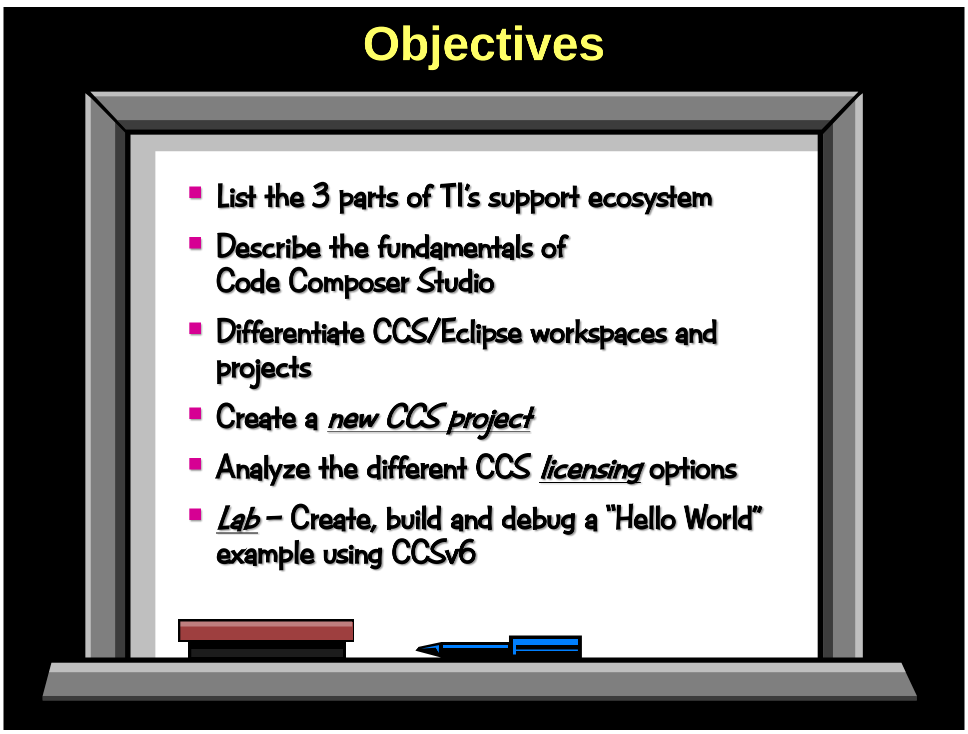
Programming in C with CCS

Introduction

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

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

Learning Objectives



Chapter Topics

Programming in C with CCS 2-1

TI Support Ecosystem 2-3
Run-Time Software 2-4
Low-level C Header Files 2-4
MSP430ware (DriverLib) 2-4
Energia 2-5
TI-RTOS 2-5
Development Tools 2-6
Integrated Development Environments (IDE) 2-6
Other MSP430 Tools 2-7

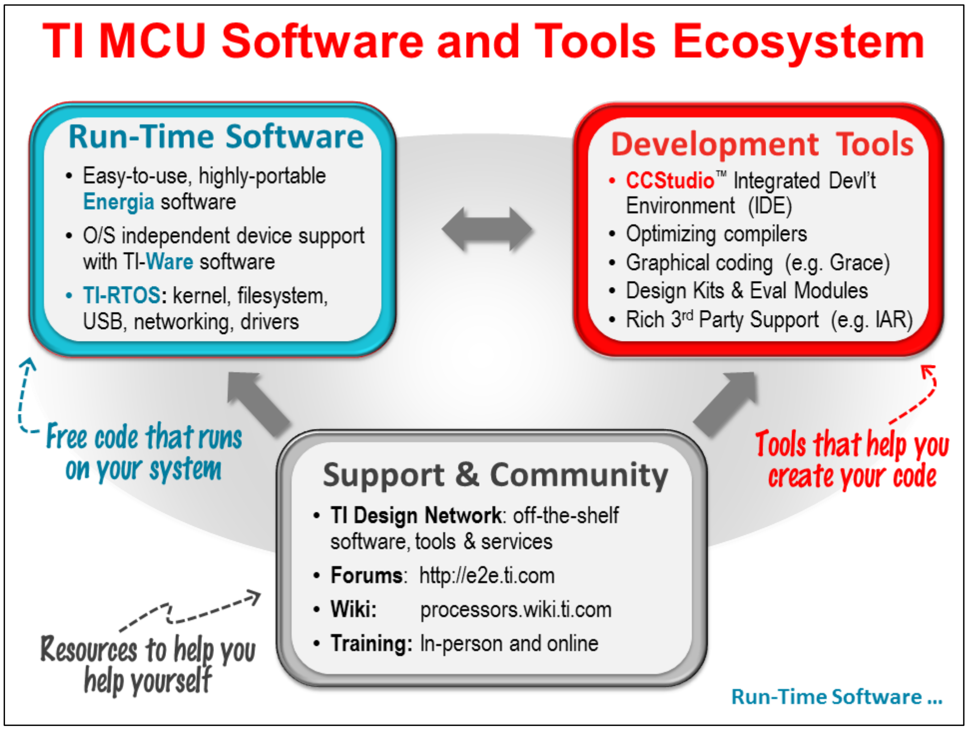
Examining Code Composer Studio 2-8
Functional Overview 2-8
Editing 2-8
Debugging 2-10
Target Config & Emulation 2-10
Emulation Hardware 2-11
Perspectives 2-12
Workspaces & Projects 2-13
Some Final Notes about CCS/Eclipse 2-14
Portable Projects 2-15
Creating a Project 2-16
Adding Files to a project 2-17
Licensing/Pricing 2-18
Changing a CCS User Licence 2-19

Writing MSP430 C Code 2-20
Build Config & Options 2-20
Debug Options 2-21
Optimize Options (aka “Release” Options) 2-21
Build Configurations 2-22
Data Types 2-23
Device Specific Files (.h and .cmd) 2-24
MSP430 Compiler Intrinsic Functions 2-26

Lab 2 – CCStudio Projects 2-27

TI Support Ecosystem

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



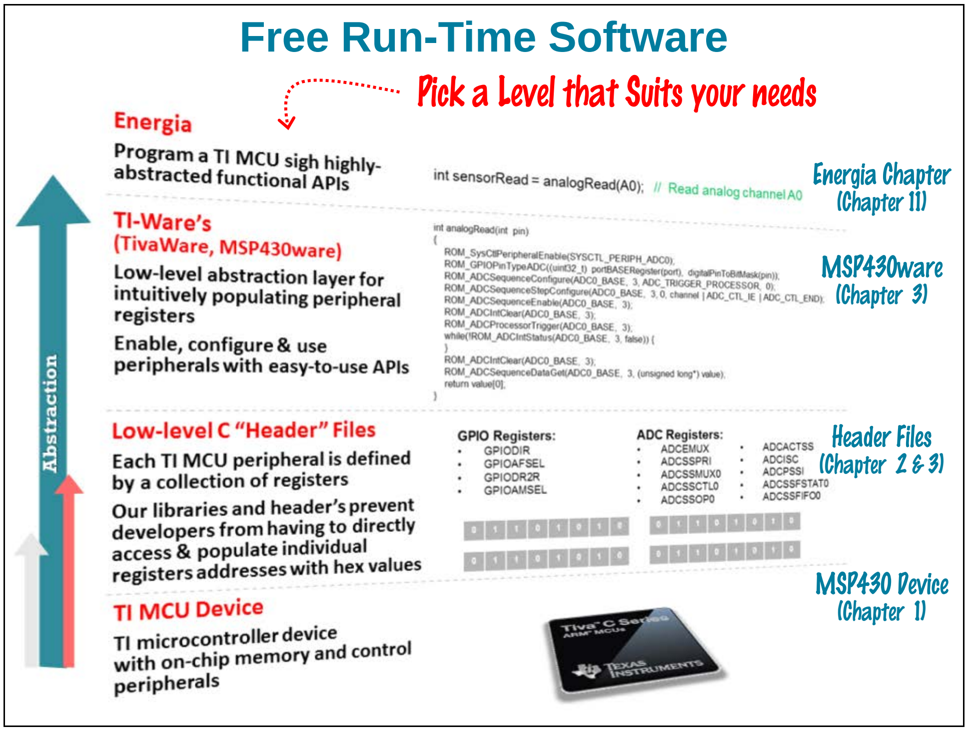
We'll take a brief look at all three parts of the Ecosystem:

- Run-Time Software
- Development Tools

Support and Community was examined back in Chapter 1.

Run-Time Software

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



Low-level C Header Files

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

MSP430ware (DriverLib)

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

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

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

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

Energia

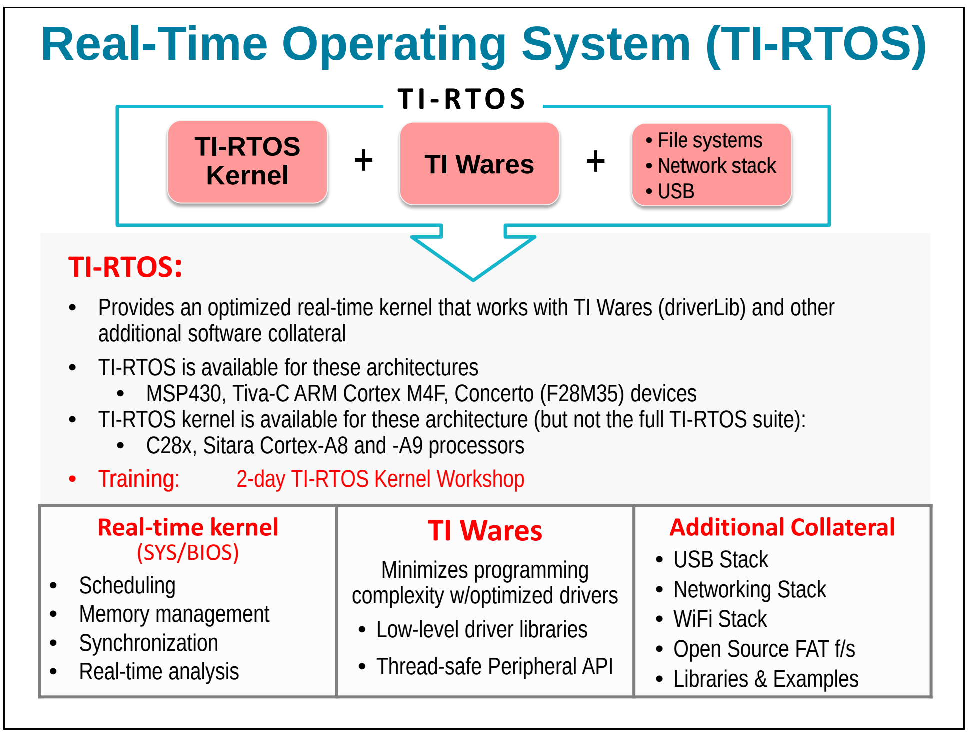
Energia is a community-based port of the ever-popular **Arduino**. This software makes it easy for users to grab code already available in the Arduino community and put it to good use on TI's MSP430 Launchpads. In other words, it puts the word "rapid" in rapid-prototyping.

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

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

TI-RTOS

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



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

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

Development Tools

Integrated Development Environments (IDE)

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

Development Tools for MSP430

	IAR SYSTEMS	Code Composer Studio	Energia	Open Source
Evaluation License	❑ 32KB code-size or 30-day limit ❑ Upgradeable	❑ Full function ❑ JTAG limited after 90-days	N/A	N/A
Compiler	IAR C/C++	TI C/C++ or GCC	GCC*	GCC
Debugger and IDE	❑ C-SPY ❑ Embedded Workbench	❑ 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 ❑ serial.printf() ❑ LED or scope	MSP-FET430UIF $99

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

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

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

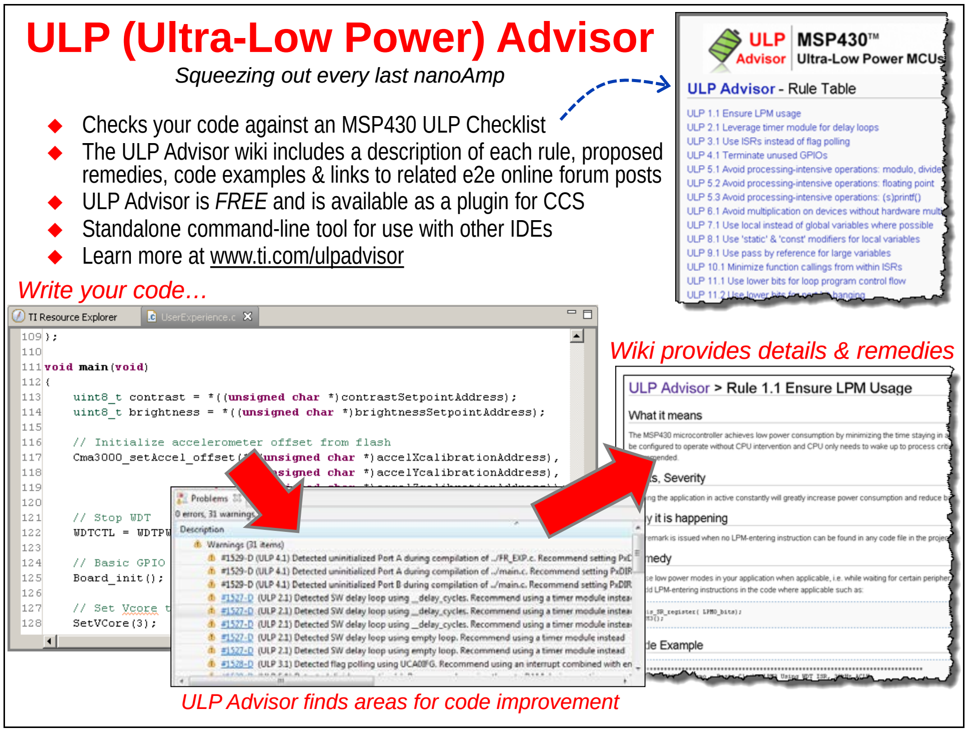
At the other end of the spectrum, we know that some of our customers cannot even afford the low-cost price-point of CCS. For hobbyists and folks needing to rapid-prototype systems, the Energia open-source port of Arduino is a great option.

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

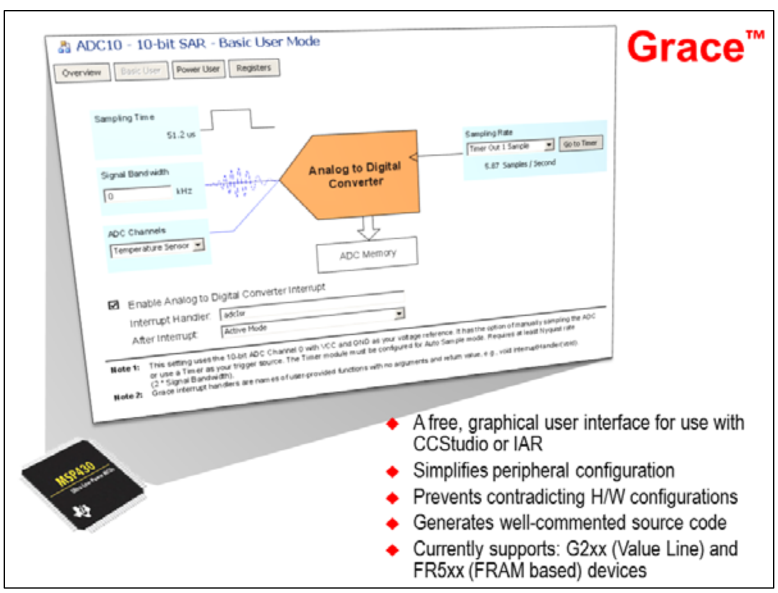
It doesn't matter which tool suite you choose, in any case, you'll still have all the other MSP430 ecosystem components at your disposal. For example, MSP430ware DriverLib works in all of these environments.

Other MSP430 Tools

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



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

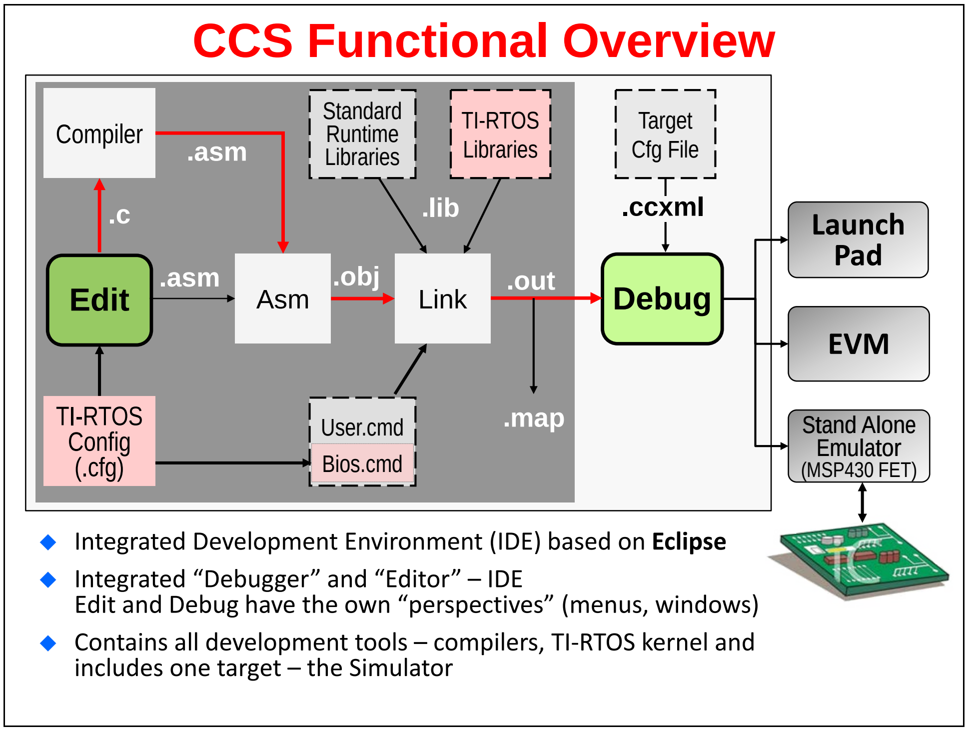


Examining Code Composer Studio

Functional Overview

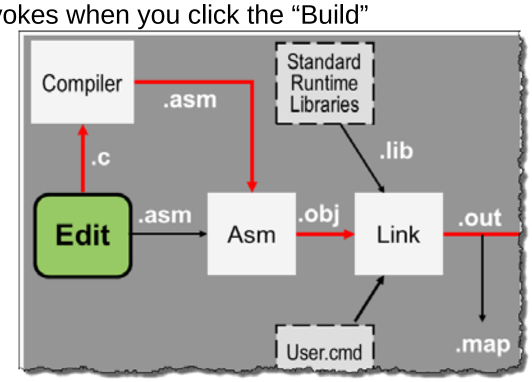
As described earlier, Code Composer Studio is TI's Eclipse based Integrated Development Environment (IDE). You might also think of IDE as meaning, "Integrated Debugger and Editor", since that's really what it provides. CCS is made up of a suite of tools that help you:

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



Editing

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



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

.c — Your C (or C++) source code files

.asm — Assembly files are created by the compiler. By default, they're considered temporary and deleted; though, you can tell CCS to retain them.

.obj — Relocatable object files. Again thought of as temporary and deleted when build is complete.

.lib — Any object library you want to reference in your code.
By default, TI's compiler ships with a run-time support library (RTS) that provides standard C functions. See the compiler user's guide for more information. (slau132.pdf)

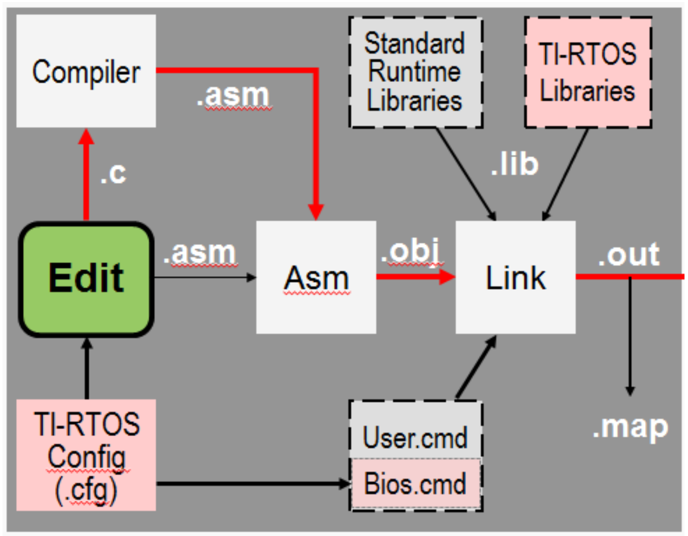
.cmd Linker command files tells the linker how to allocate memory and stitch your code and libraries together. TI provides a default linker command file specific to each MSP430 device; it is automatically added to your project when you create a new project. You can edit it, if needed, though most users get by without ever touching it.

.out The executable output file. This is the file that is loaded into Flash or FRAM on your MSP430 MCU whenever you click the “Debug” button on your CCS toolbar.

.map The map file is a report created by the linker describing where all your code and data sections were linked to in memory.

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

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



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

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

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

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

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

Debugging

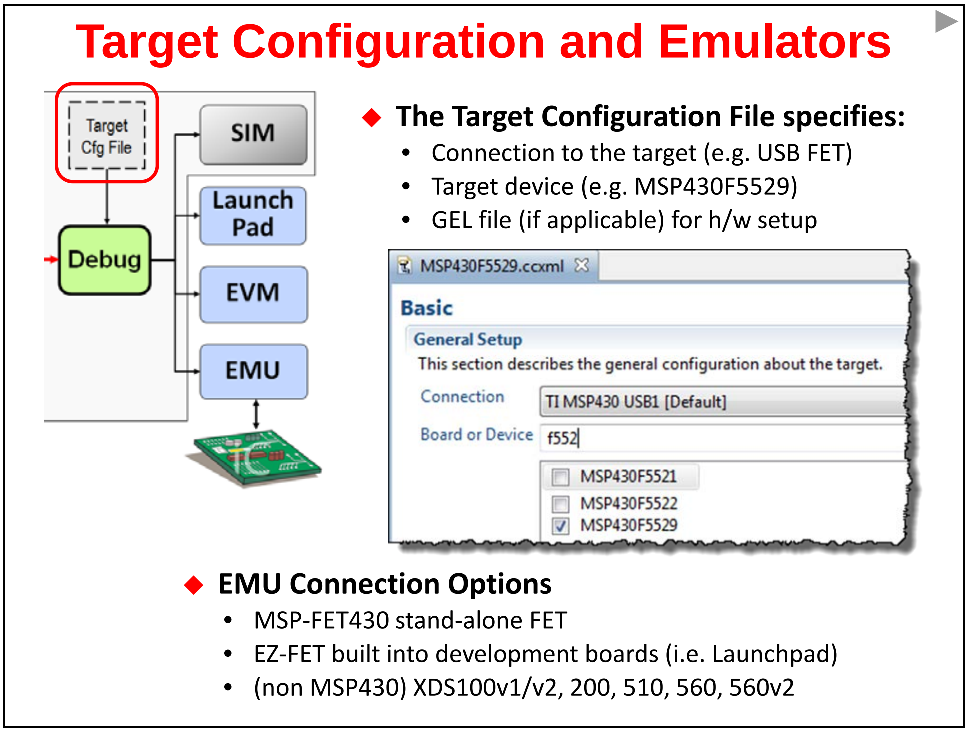
Once again, the “debug” side of the Code Composer Studio lets you download your executable output (.out) file onto your target processor (i.e. MSP430 device on your Launchpad) and then run your code using various debugging tools: breakpoint, single-step, view memory and registers, etc.

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

Target Config & Emulation

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

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



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

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

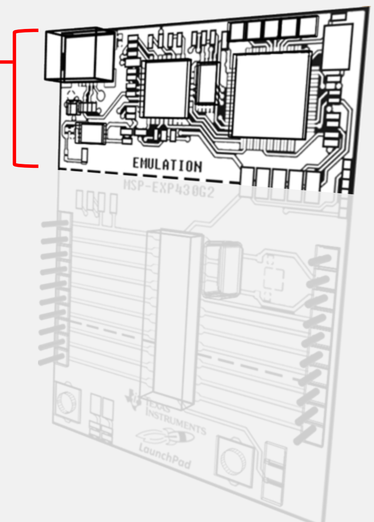
Note: If you ever get an error that indicates CCS doesn’t know how to connect to the target, you probably didn’t specify the “connection” when creating your project. You can easily fix this by editing the project’s properties.

Emulation Hardware

MSP430 JTAG Emulators

Integrated Flash Emulation Tool

- Eliminates need for external tool
- Integrated USB-powered emulator
  - Mini USB cable
- Program & debug any MSP430 Value Line MCU through the Spy Bi-Wire (2-wire JTAG) protocol
- Use LaunchPad as a programmer ANY Spy Bi-Wire enabled MSP430 (not officially supported by TI)



Flash Emulation Tool (MSP-FET)

One tool to rule them all – Direct replacement to MSP-FET430UIF

Features:

- USB debugging interface to connect any MSP430 MCU to a PC for real-time, in-system programming and debugging
- Enables EnergyTrace™ technology for energy measurement and debugging on all MSP430 devices
- Up to 4x faster than its predecessor (MSP-FET430UIF)
- Includes Backchannel UART for bi-directional communication between the MSP430 and a PC

Order now @ www.ti.com/tool/msp_fet

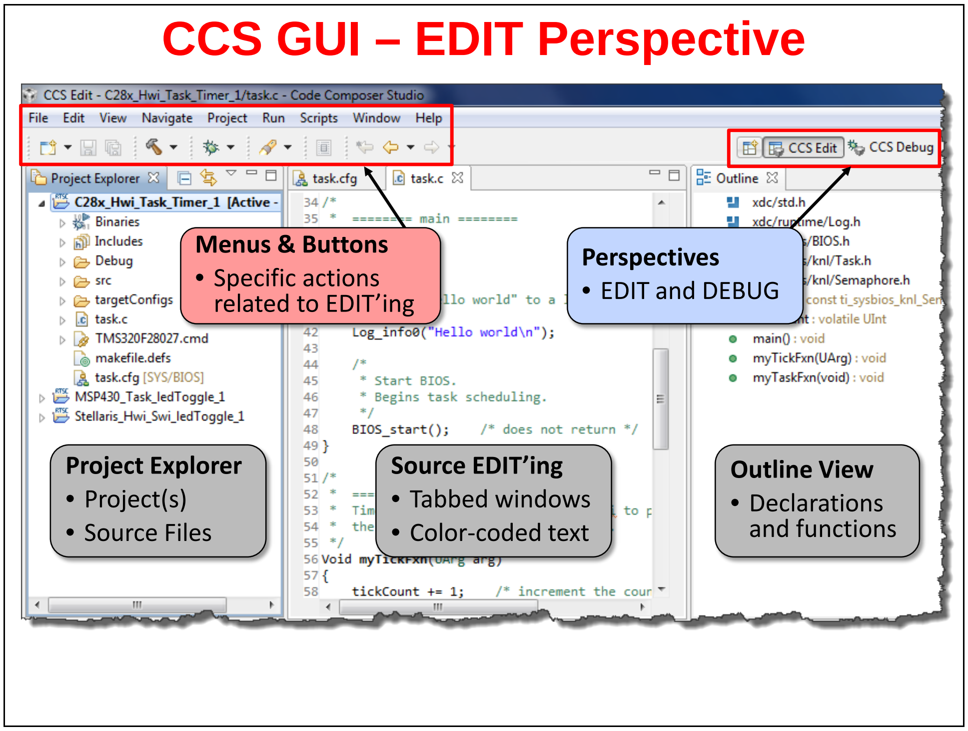
Technical Specifications:

- Software configurable supply voltage between 1.8 V and 3.6 V at 100 mA
- Supports JTAG Security Fuse blow to protect code
- Supports all MSP430 boards with JTAG header
- Supports both JTAG and Spy-Bi-Wire (2-wire JTAG) debug protocols

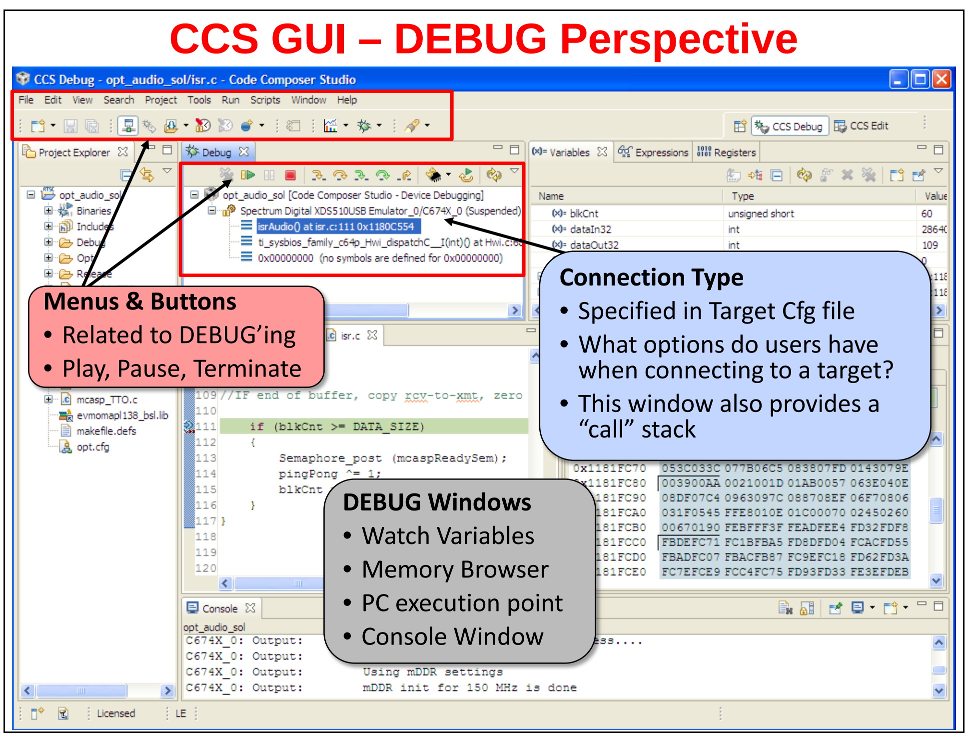
Texas Instruments

Perspectives

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



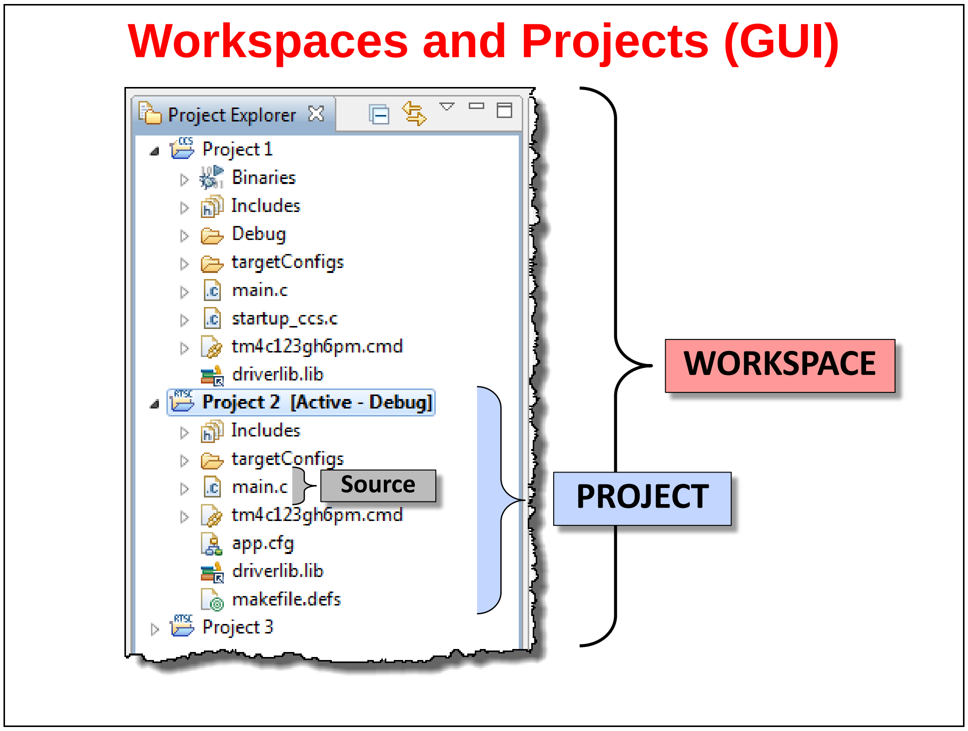
Eclipse even varies the toolbars and menus between perspectives.



Workspaces & Projects

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

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



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

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

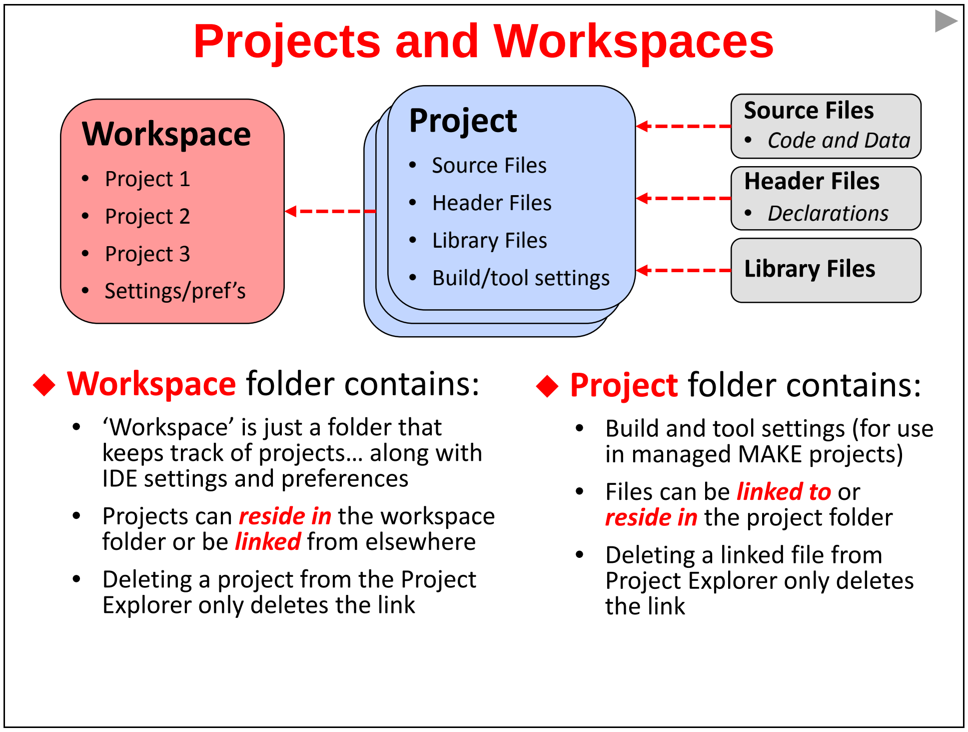
In our workshop, we have chosen to create one *workspace* which will hold all of our lab files. This makes it easy to switch back and forth between exercises, if you should want to do so.

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

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

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

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



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

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

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

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

Some Final Notes about CCS/Eclipse

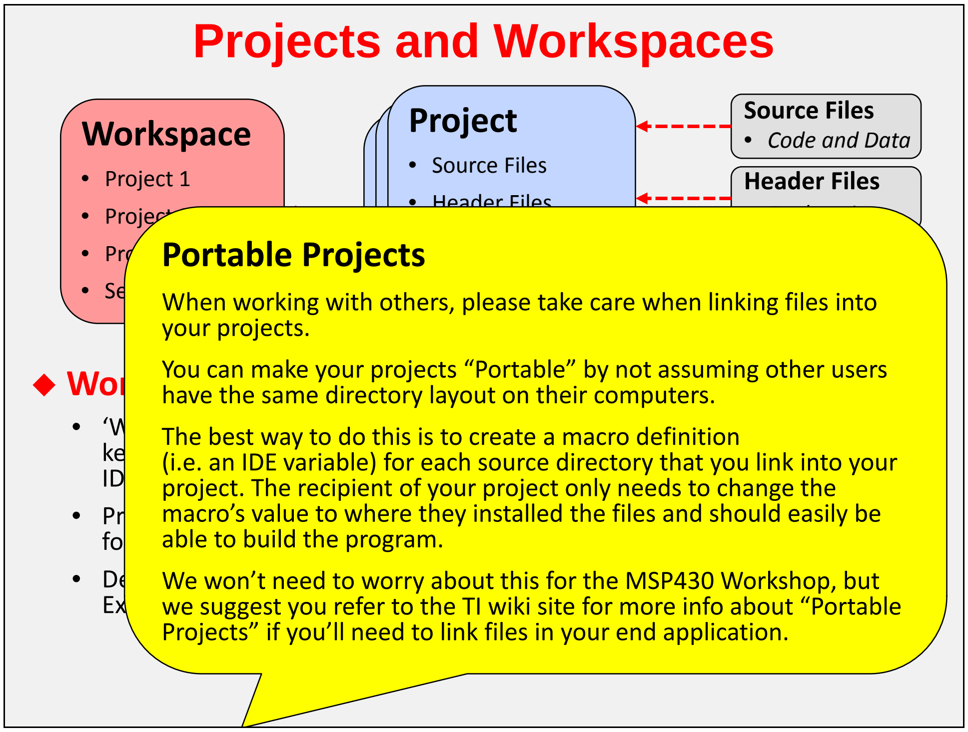
- If you create a new source file in CCS/Eclipse, it will automatically be stored in the project folder.
- If you copy a source file (e.g. C file) into the project folder using the O/S filesystem, it will automatically show up in the project. That is, if you copy a C file into the project folder using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

Portable Projects

While this will not be an issue when working with the MSP430 – at least in this workshop – you should be aware that build issues can arise when sources (files and/or libraries) are ***linked*** into a project. It isn't normally an issue on the system where the project is created, but rather, build problems can show up when sharing the project with other team members.

If your teammates do not have *exactly* the same file directory hierarchy as the person who created the project, the tools may not be able to find all of the sources – and thus, the build will fail.

This is not a TI specific problem; hence, the Eclipse IDE provides a solution.



As described here, the solution involves creating a "pointer" to each directory which contains linked source or library files. Officially, these "pointer" is called a "macro"; although it might be better described by the term "IDE variable".

Whatever you call this feature, a teammate who wants to build the project just needs to verify that the "pointer" macro contains the same directory path as the original user. If not, by updating any macro that differs in their system, the new user can easily build the project.

This is one of those problems that you might not realize is important… until you run into it.

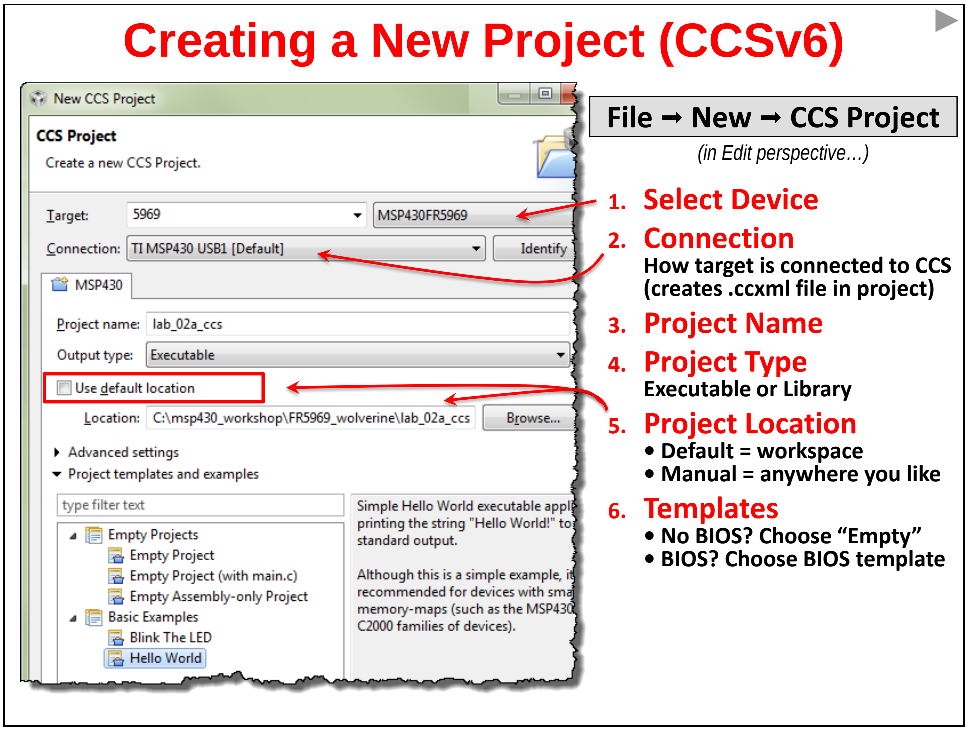
Note: In the case of the MSP430 applications team, they recommend importing the entire MSP430 Driver Library into your project. This not only eliminates the problem of linked libraries, but it also means that the library will be built with the same compiler options as the rest of your project.

Creating a Project

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

```
File → New → CCS Project
```

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



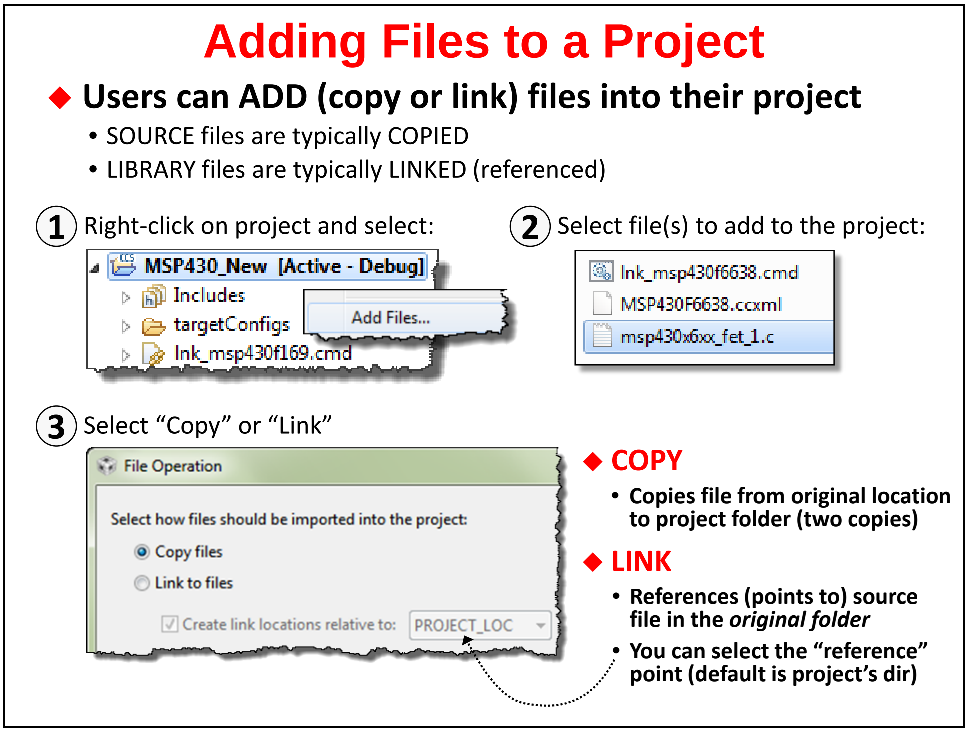
When creating a new project you need to define:

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

Adding Files to a project

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

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



Portable Projects

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

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

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

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

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

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

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

Licensing/Pricing

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

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

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

CCStudio Licensing and Pricing

Licensing

- Wide variety of options (node locked, floating, time based)
- All versions (full, DSK, free tools) use same image
- Annual subscription - $99 *($159 for floating)*
- Updates available online

	Item	Each
 SINGLE USER	**TMDSCCS-ALLN01D** Code Composer Studio IDE - Node Locked Single User (N01D) Download Only / NO DVDs	$445.00

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

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

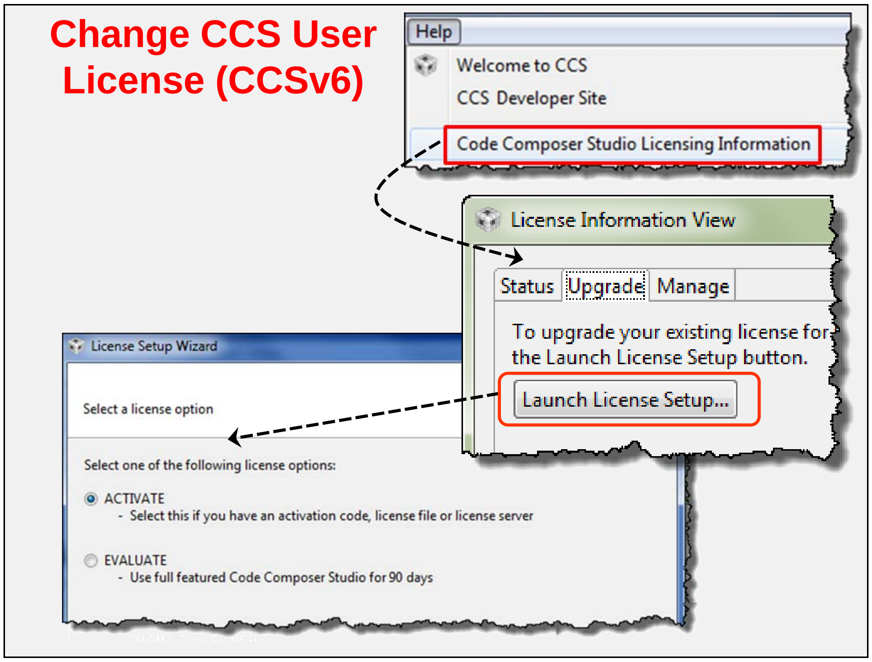
For those cases where you need to use a more sophisticated (i.e. faster) JTAG development connection, TI provides a 90-day free evaluation license. After that, you need to purchase the tool. Thankfully, when you encounter one of these cases, CCS for only costs $445.

Changing a CCS User Licence

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

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

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



Writing MSP430 C Code

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

Build Config & Options

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

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

Compiler Build Options

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

	Options	Description
Debug	-ss	Interlist C statements into assembly listing
Optimize (Release)	-o3	Invoke optimizer (-o0, -o1, **-o2**/-o, -o3, -o4)
	-mf	Speed/code size tradeoff (-mf0 thru -mf5)
	-k	Keep asm files, but don't interlist

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

How do you CHANGE compiler build options or configurations?

***Debug* Options**

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

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

Optimize Options (aka "Release" Options)

We highlight 3 optimization options:

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

Build Configurations

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

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

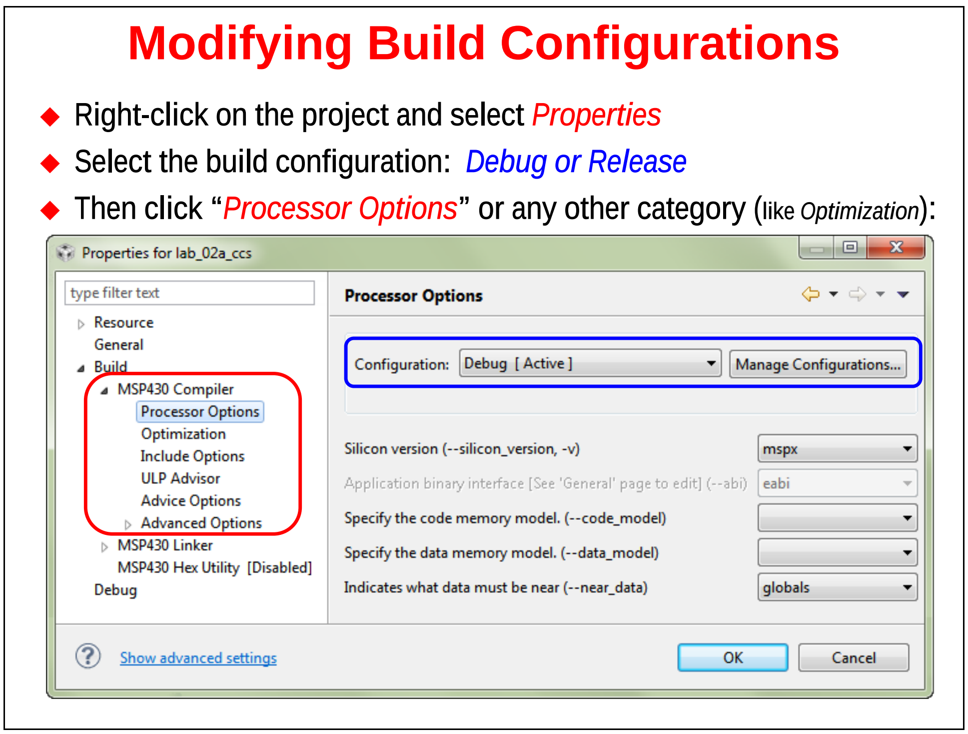
Rather than forcing you to continuously tweak options by hand, you can use *Build Configurations*. Think of these as 'groups' of options.

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

- Debug
- Release

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

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



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

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

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

Data Types

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

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

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

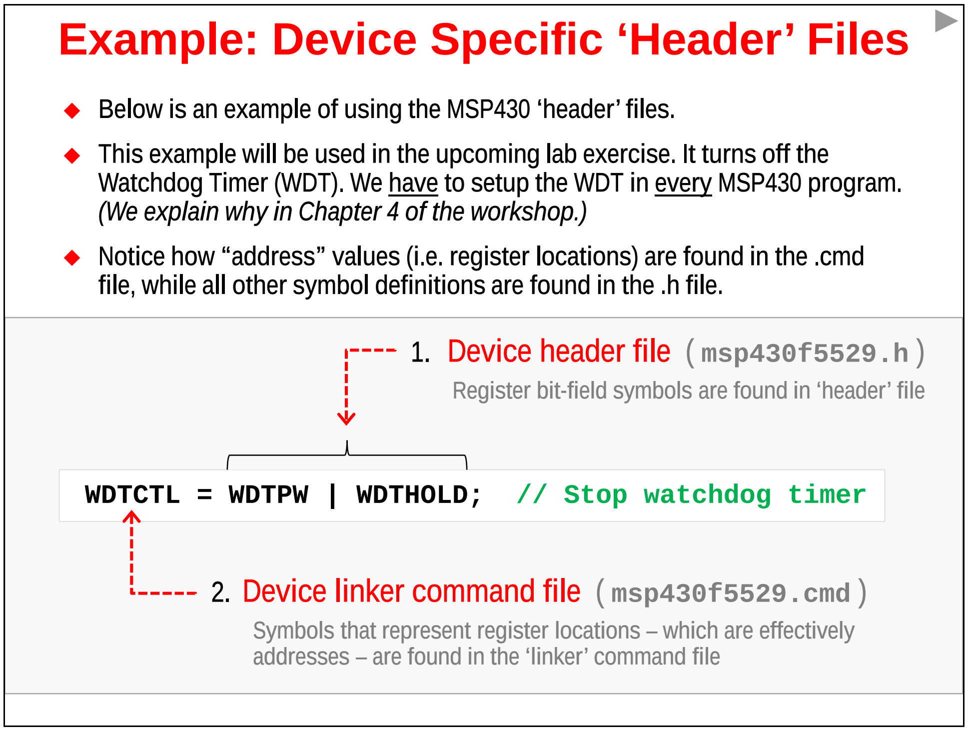
MSP430 C Data Types (ELF format)

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

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

Device Specific Files (.h and .cmd)

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



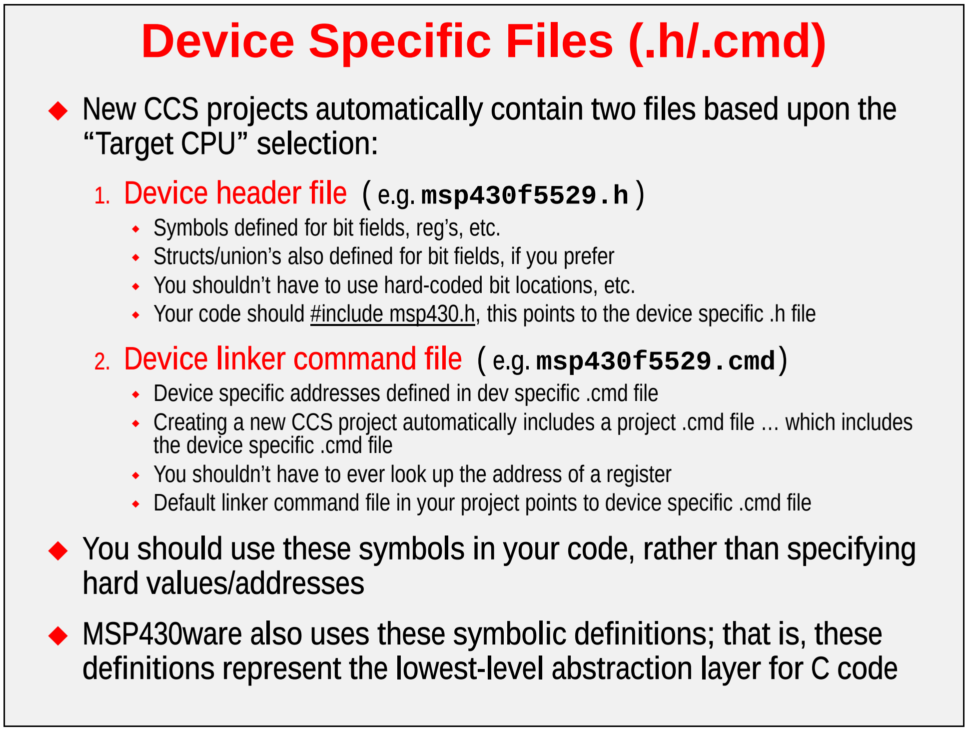
As described in the above diagram, these two files provide symbolic definitions for all registers and bitfields found in each CPU and its peripherals.

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

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

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

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



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

MSP430 Compiler Intrinsic Functions

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

Intrinsics for MSP430 C Compiler

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

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

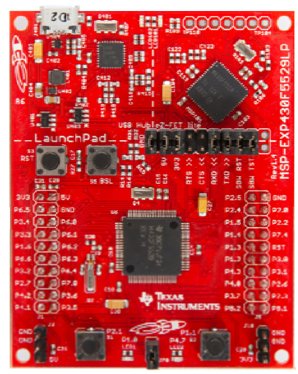
We've circled some of the intrinsic functions we'll use in this class; from setting and/or clearing bits in the Status Register (SR) to putting the processor into low-power modes.

Lab 2 – CCStudio Projects

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

Lab 2 – Creating CCS Projects

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



Time: 45 minutes

Lab Outline

Programming C with CCS 2-25

Lab 2 – CCStudio Projects 2-27

Lab 2a – Creating a New CCS Project 2-29

Intro to Workshop Files 2-29

Start Code Composer Studio and Open a Workspace 2-30

"CCS Edit" Perspective 2-31

Create a New Project 2-32

Build The Code (ignore advice) 2-35

Verify Energy Trace is 'Off' 2-36

Debug The Code 2-36

Fix The Example Project 2-39

Build, Load, Connect and Run … using the Easy Button 2-40

Lab 2b – My First Blinky 2-41

Create and Examine Project 2-41

Build, Load, Run 2-42

Restart, Single-Step, Run To Line 2-43

(Optional) Lab 2c – Restoring the OOB 2-45

(Optional) Lab 2d – MSP430Flasher 2-47

Programming the OOB demo using MSP430Flasher 2-47

Programming Blinky with MSP430Flasher 2-51

Cleanup 2-52

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 CCS. If you already have experience with CCS (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_FRAM:

   ```
   C:\msp430_workshop\F5529_USB
   C:\msp430_workshop\FR4133_FRAM
   C:\msp430_workshop\FR5969_FRAM
   ```

   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.
- The readme file provides the solution code that you can copy/paste, if necessary. That said, you won't need to do that in this lab exercise.

Start Code Composer Studio and Open a Workspace

Note: CCSv6 should already be installed; if not please refer to the workshop installation guide.

2. Start Code Composer Studio (CCS).

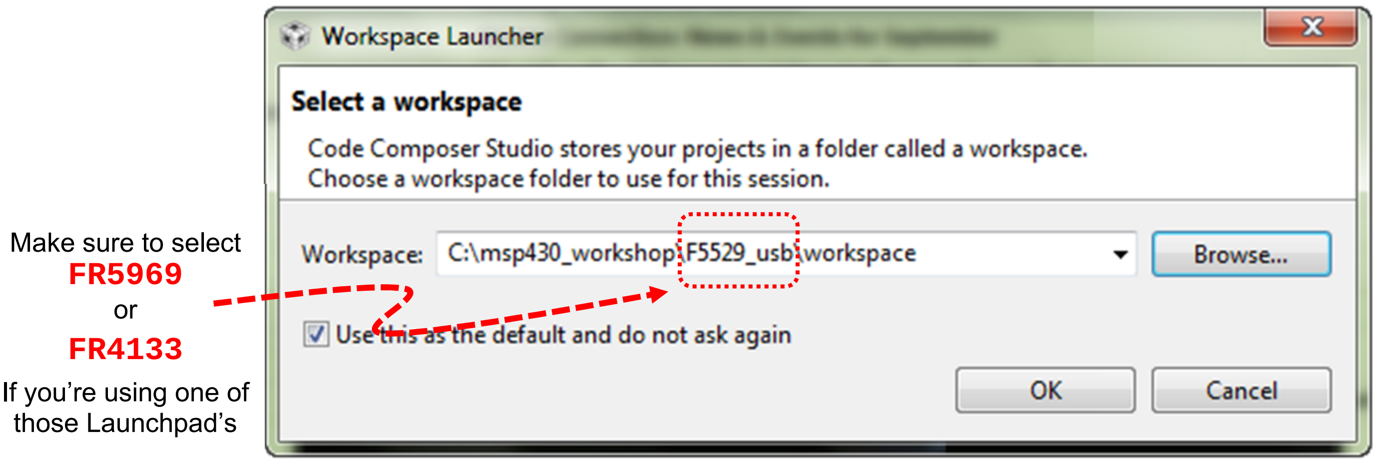
Double-click CCS's icon on the desktop or select it from the Windows Start menu.

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 either one of: (to match your target)

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 and views. The workspace is saved automatically when CCS is closed.

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

4. Click OK to close the Select a workspace dialog.

5. After quickly examining the "Getting Started" window, you can close it, too.

When CCS opens to a new workspace, the *Getting Started* window is automatically opened and you're greeted with a variety of options. We want to mention two items:

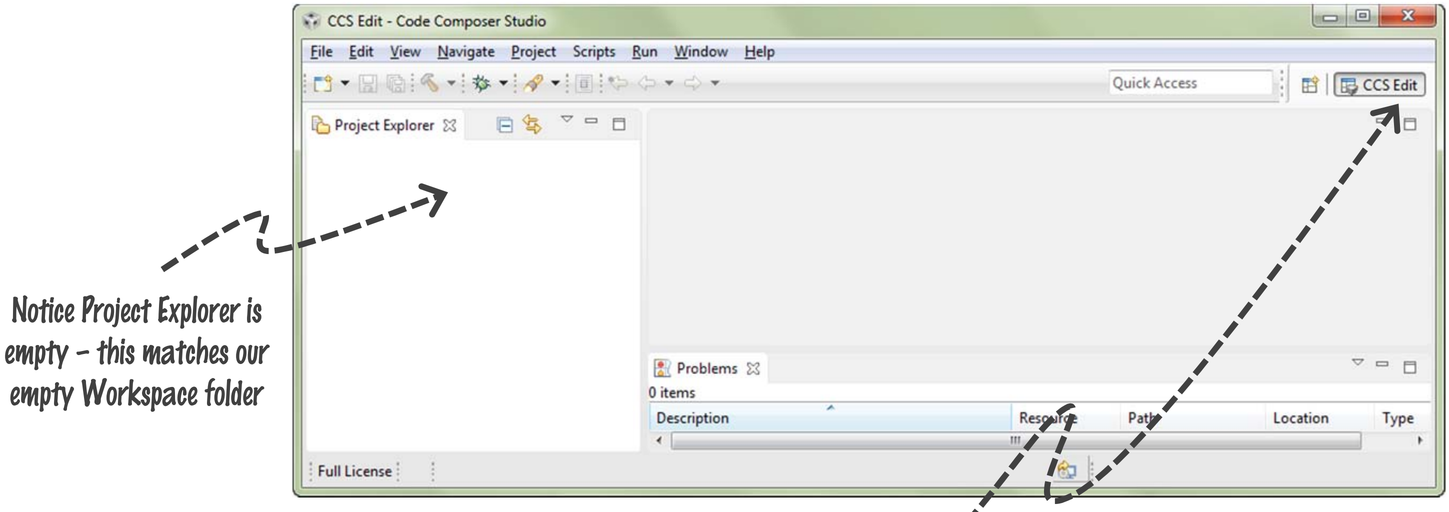
- **App Center** – you can download additional TI tools and content here. For example, this is one way to install MSP430ware or TI-RTOS.
- **Simple Mode** – We suggest that you <u>do not</u> put CCS into Simple Mode when following our lab instructions, as we've chosen to use the full-featured interface.

Later on, you may want to come back and check out the remaining links and videos.

"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.

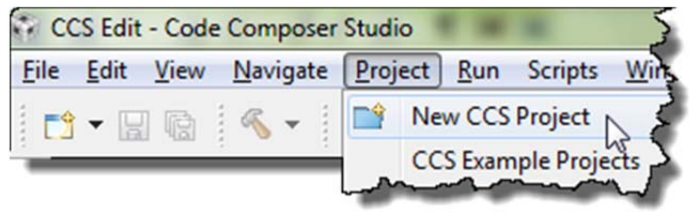
Hint: The `Window → Reset Perspective…` is handy for those times when you've changed the windows and want to get back to the original view.

Create a New Project

7. Select New CCS Project from the menu.

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

`File → New → CCS Project`



8. Make project choices as shown here:

Note: Your dialog may look slightly different than this one. This is how it looked for CCSv6.0 (build 190).

a) Type "**5529**", "**5969**" or "**4133**" into *variant* to quickly select **Target CPU**

b) Use *Default* debugger connection *(this creates the .ccsxml file for you)*

c) Name: `lab_02a_ccs`

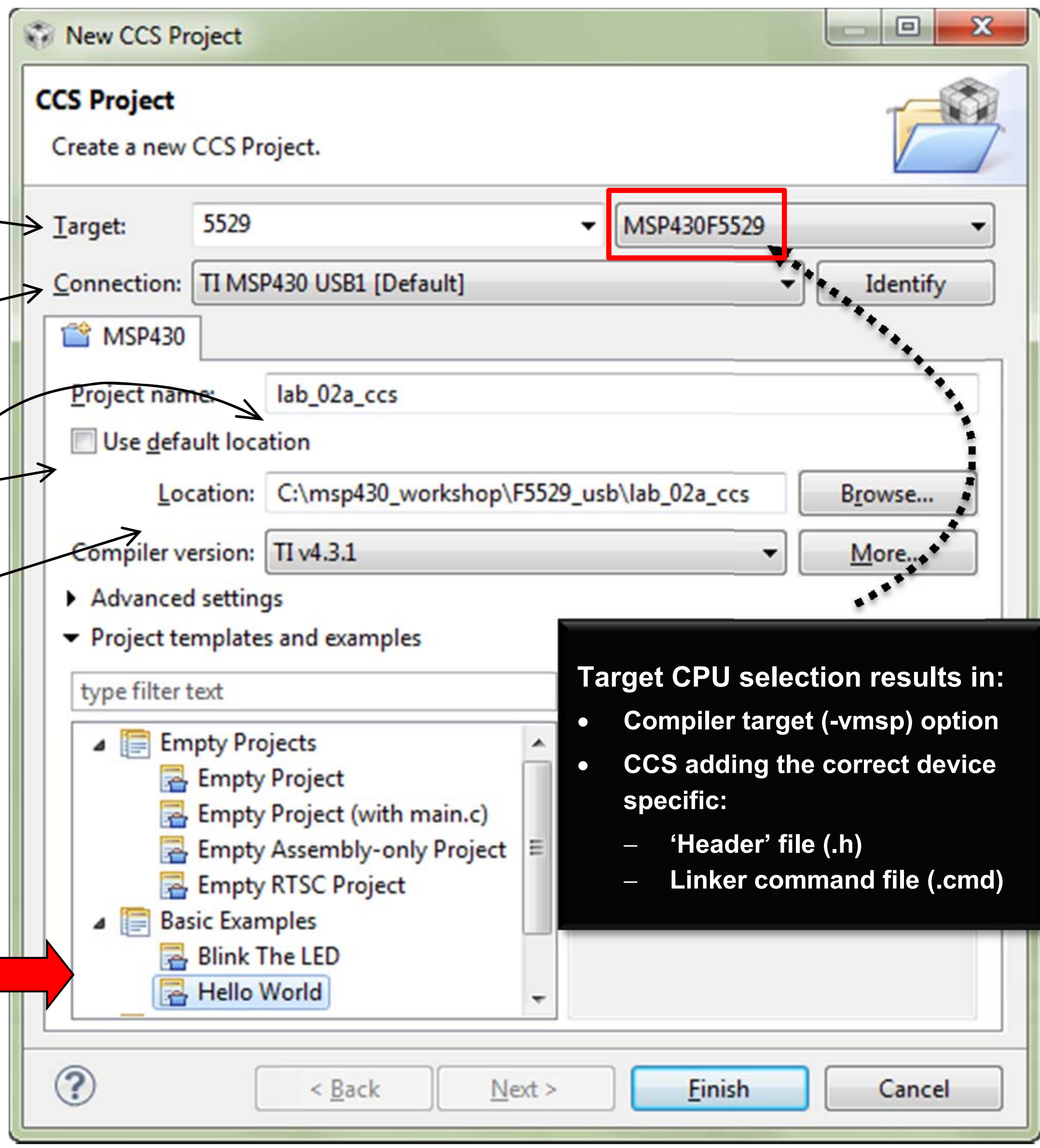
d) <u>Don't</u> use default location

e) Choose your *target's* `lab_02a_ccs` folder

f) Select template:

`Hello World`

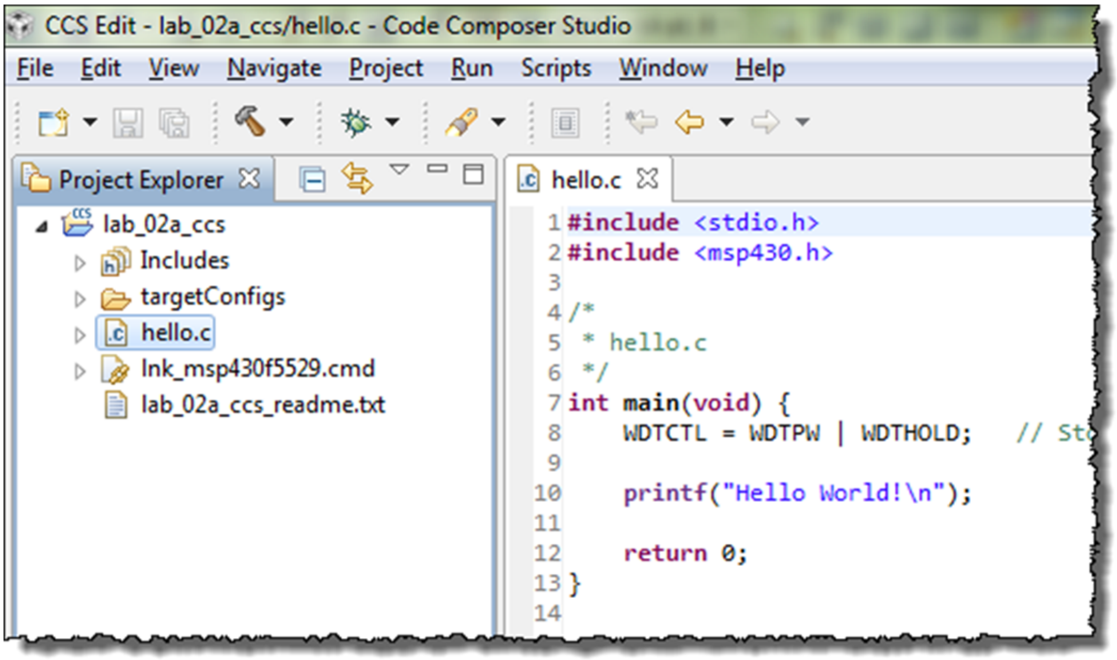
g) Click 'Finish' when done.



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').

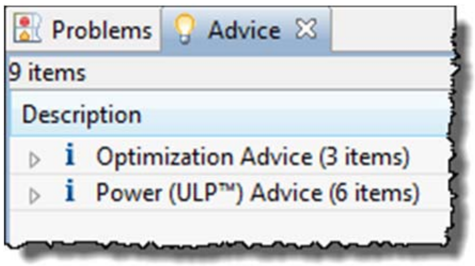
Build The Code (ignore advice)

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 and Power (ULP™) Advice.

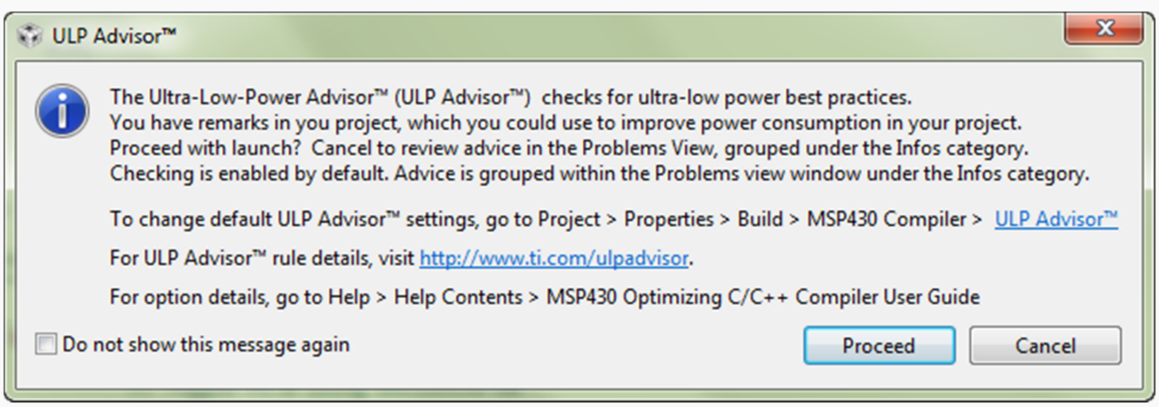


At this point, we're just going to ignore their advice. It's better to get code running first. Later, we return and investigate some of these items further.

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.

<u>Sidenote: ULP Advisor</u>

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



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

`☑ Do not show this message again`

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

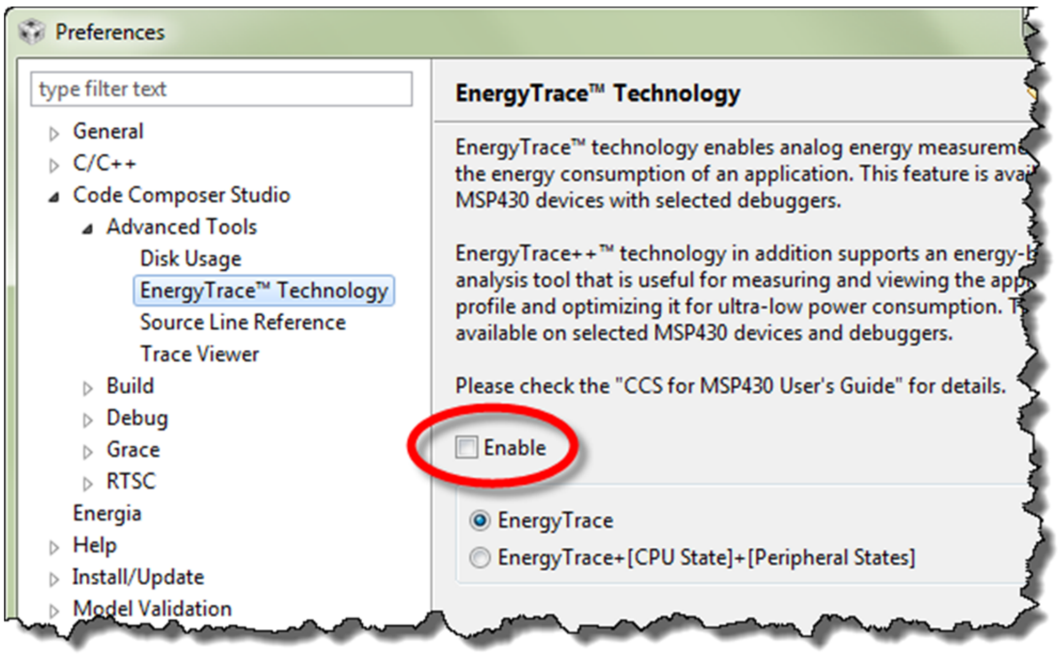
Verify Energy Trace is 'Off'

We really like the new EnergyTrace features in CCS. It provides an incredible amount of information – but, we really don't need all of that info when we're just trying to get an LED to blink. Some versions of CCS turn this new feature 'on' by default. We suggest turning it off – for now. We'll re-enable it during the Low Power Optimzation chapter.

13. Disable EnergyTrace (or verify it's disabled).

`Window → Preferences`

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



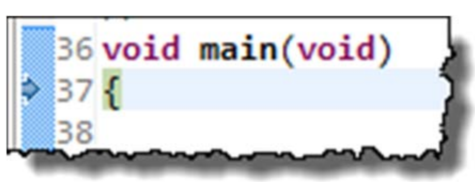
Debug The Code

14. Debug your program.

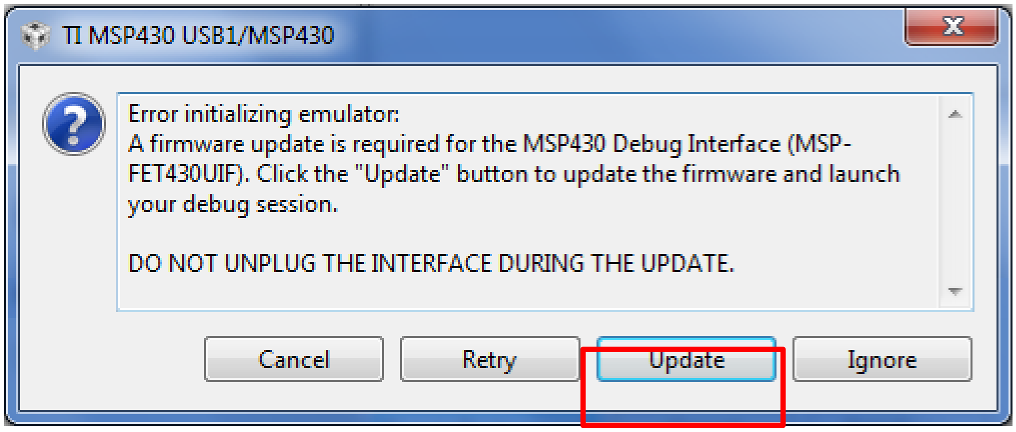
Clicking the Debug button will: Build the program (if needed); Launch the debugger; Connect to Target; and Load your program

`Click the BUG toolbar button:` 

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



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.

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. 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.

16. Pause the code.

To stop your program running, ► click the **Suspend** button to pause): 

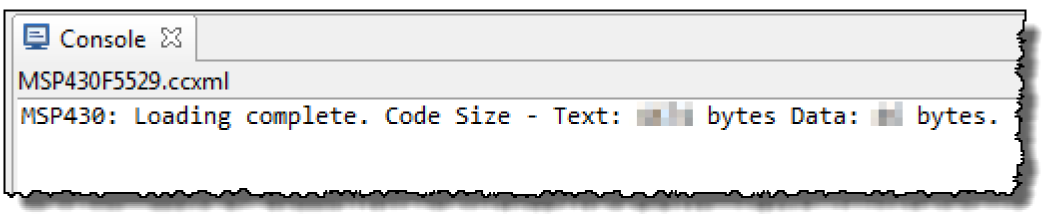
Warning: Suspend is different than **Terminate** !!!

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.

---

17. Did printf work?

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



Nope, it didn't show up for us. ☹

18. Let's Terminate the debug session and go fix "their" project.

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.

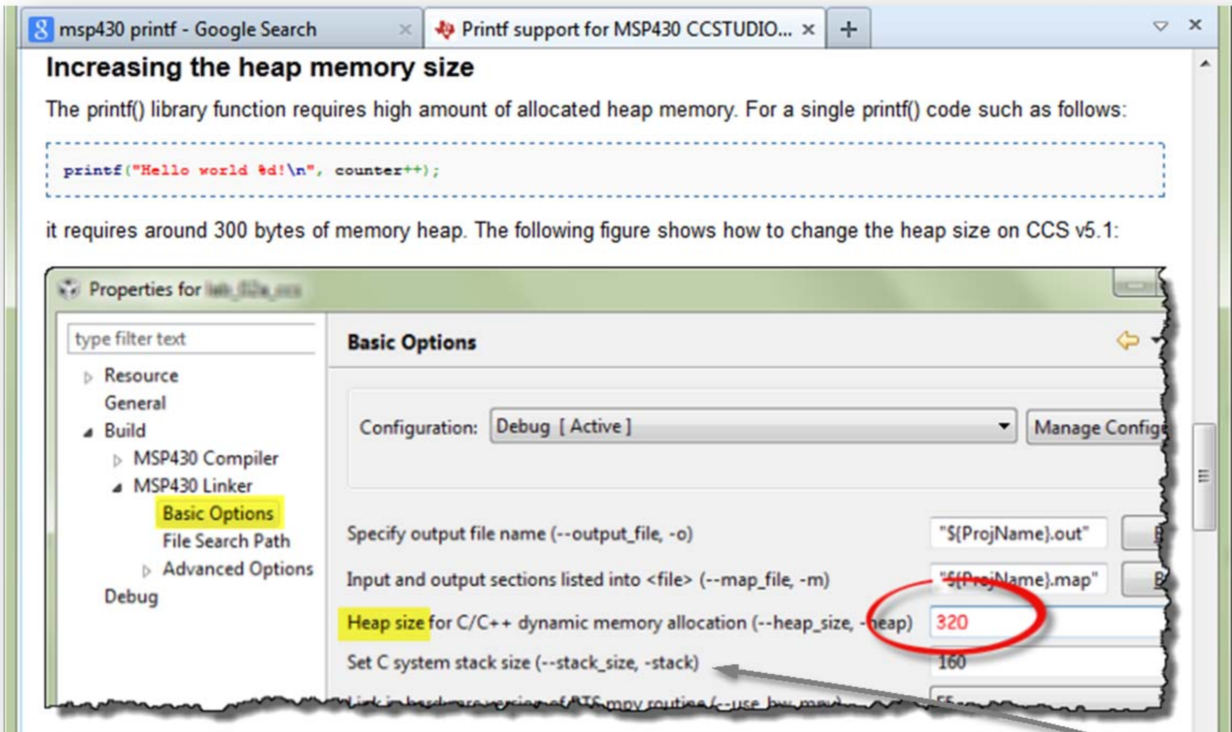
Fix The Example Project

19. What is wrong?   Increase the heap size.

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

```
Right-click project → Properties → MSP430 Linker → Basic Options
```

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



You can find a description of this problem by searching the internet for: "msp430 printf"

From that, you should find a MSP430 wiki page that describes how to get printf() to work:

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

(In fact, this is how we figured out how to solve the problem.)

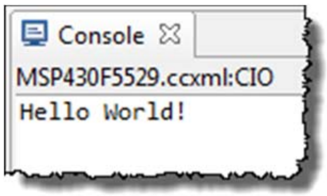
Hint: As a side note, if you look just below the entry for setting the Heap size, you will see the setting for Stack size. This is where you would change the stack size of you system, if you ever need to do that.

Build, Load, Connect and Run … using the Easy Button

20. Rebuild and Reload your program.

First, make sure you terminated your previous debug session and you are in the Edit perspective.

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



FR4133

Note: The 'FR4133 may stop half-way through the printf() routine – if this happens, just click the Run/Resume button again and it should continue.

You can avoid this unintended breakpoint by setting the FRAM waitstates to 0. The default waitstates value on the 'FR4133 is 1, which covers running the processor up to its full speed. If you stay at or below 8MHz, then they can be set to 0.

Eliminating this pause isn't really necessary for this lab, though we'll need to employ this trick for `lab_4b_wdt`. By Lab 4, we'll have learned how to change waitstates using Driver Library; for now, adding this line of code somewhere before the call to printf() will solve the problem:

```
FRCTL0 = FRCTLPW | NWAITS_0;
```

---

22. Terminate and Close the `lab_02a_ccs` project.

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

```
Right-click project → Close Project
```

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

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

Lab 2b – My First Blinky

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

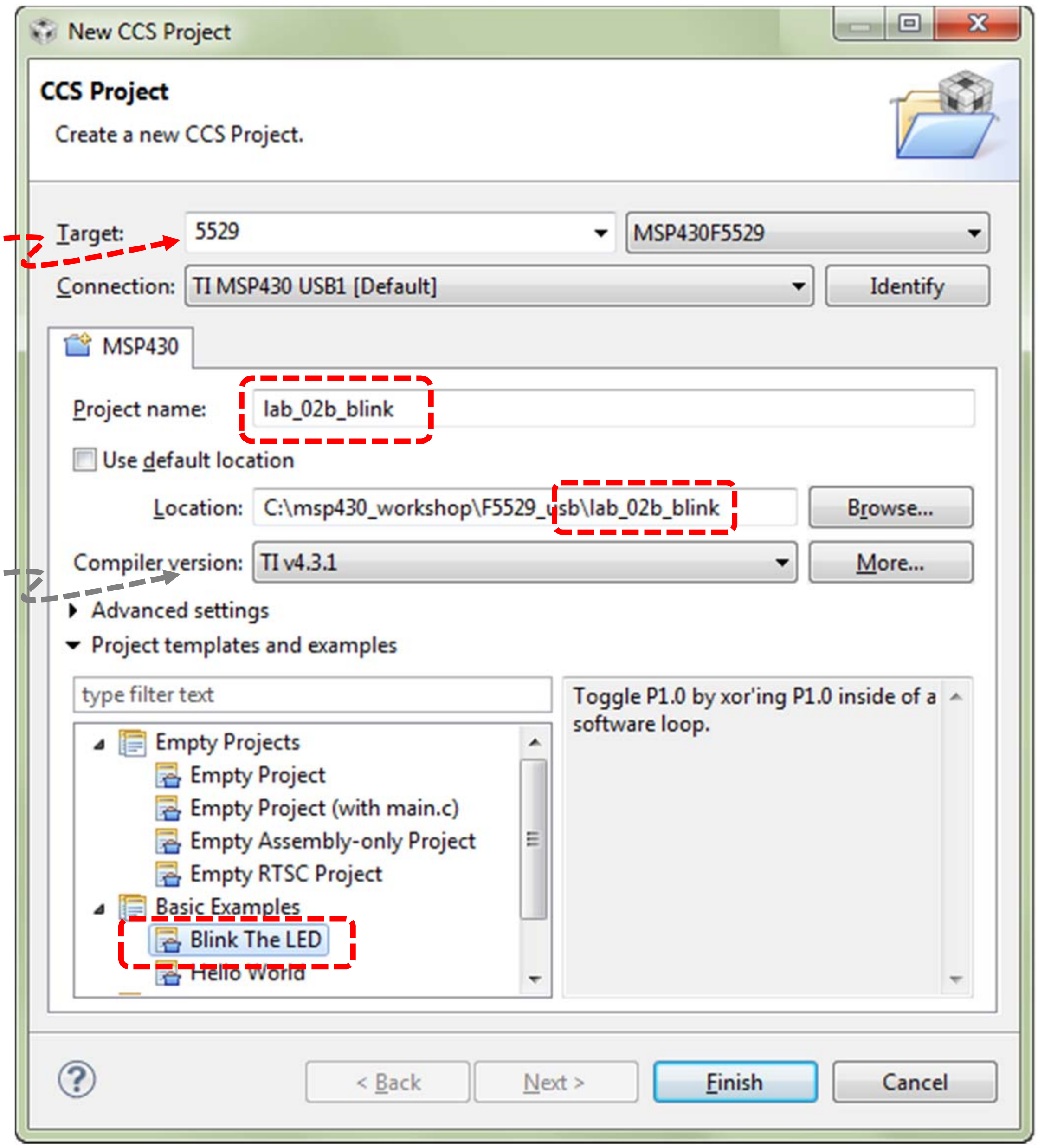
In the lab exercise, we want to teach you a few additional debugging basics – and need some code to work with. To that end, we're going to use the Blink template found in CCS. This is generic, low-level MSP430 code, but it should suite our purposes for now.

Create and Examine Project

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

Make sure to select **5969** or **4133** if you're using one of them

Choose the default compiler version



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) {
    WDTCTL = WDTPW | WDTHOLD;     // Stop watchdog timer

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

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

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

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

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

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

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

Build, Load, Run

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

   If you don't remember how, please refer back to *lab_02a_ccs*.

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

   `Click the Resume button on the toolbar (or press F8)`

   You should see the LED toggling on/off.

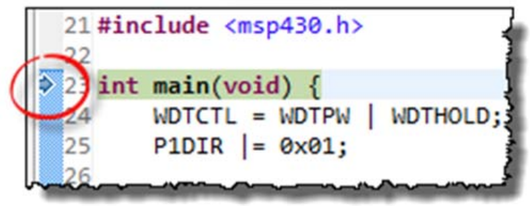
5. **Halt the debugger by clicking the "Suspend" button … don't terminate!**

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: 



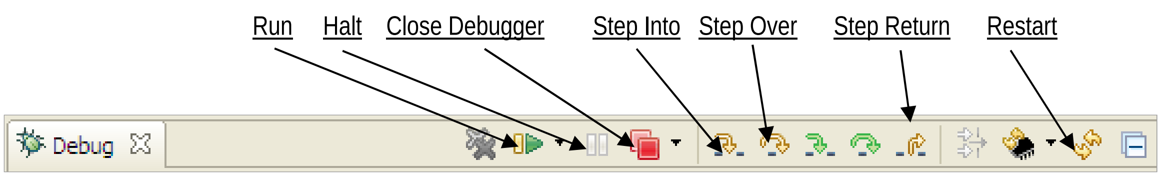
Notice how the arrow, which represents the Program Counter (PC) ends up at main() after your restart your program. This is where your code will start executing next.

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.)*

7. **Single-step your program.**

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



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

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

8. **Single-step 10,000 times**

`Try stepping over-and-over again until the light toggles again…`

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

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

Click on the line of code that toggles the LED.

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

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

`Single-step once more to toggle the LED`

10. Set a breakpoint.

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

`Double-click the blue bar on the line of code`

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

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

11. Run to breakpoint.

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

`Press F8` (multiple times)

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

12. Terminate your debug session.

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

13. Close the project.

If any edit windows are still open after closing the project, we recommend closing them, too.

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

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

(Optional) Lab 2c – Restoring the OOB

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

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

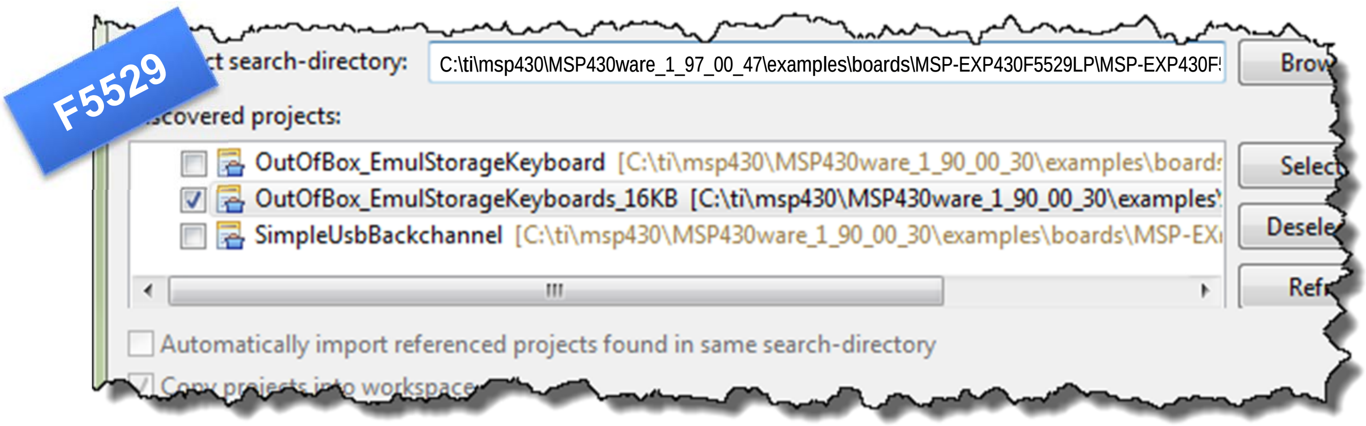
1. **Import OOB demo project.**

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

   `Project → Import CCS Projects...`

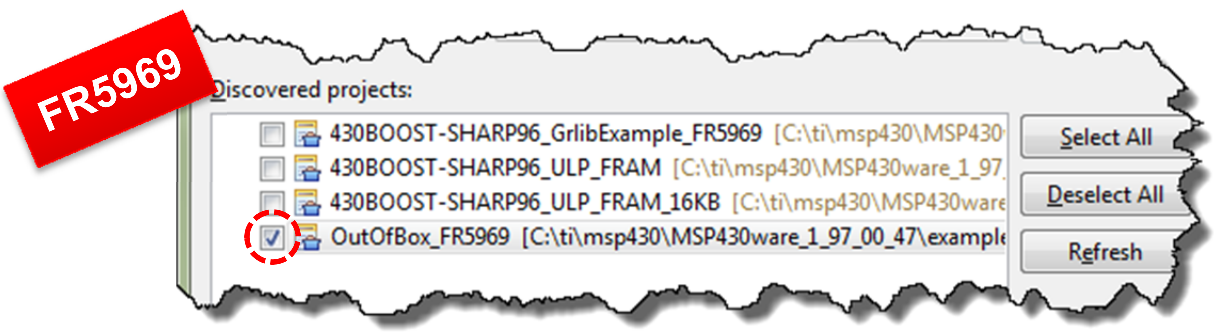
For **‘F5529** users, import the project **`OutOfBox_EmulStorageKeyboards_16KB`** from the following:

`C:\ti\msp430\MSP430ware_1_97_00_47\examples\boards\MSP-EXP430F5529LP\MSP-EXP430F5529LP Software Examples`



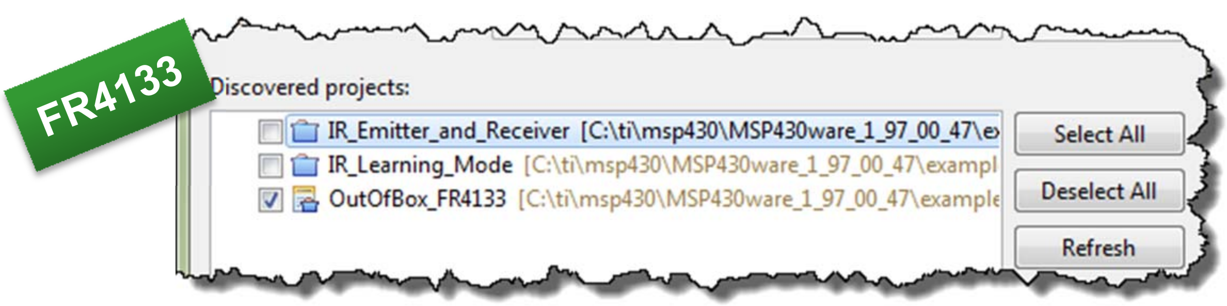
For **‘FR5969** users, import the project **`OutOfBox_FR5969`** from:

`C:\ti\msp430\MSP430ware_1_97_00_47\examples\boards\MSP-EXP430FR5969\MSP-EXP430FR5969 Software Examples`



For **‘FR4133** users, import the project **`OutOfBox_FR4133`** from:

`C:\ti\msp430\MSP430ware_1_97_00_47\examples\boards\MSP-EXP430FR4133\MSP-EXP430FR5969 Software Examples`



In all cases, if you have a choice, check *"Copy projects into workspace"* and then hit the *Finish* button.

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

3. **Click the Debug button to launch the debugger, and load the program to flash.**

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

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

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

   The original demo, which was just re-programmed into Flash/FRAM, should now be running. *(You can refer back to `Lab1` if you have questions on how to use the demo.)*

(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 during development. The batch files we're provide should'nt hurt your Launchpad – but you should treat this tool with caution.

Programming the 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 optional 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\oob.bat
```

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

```
CLS

C:\ti\MSP430Flasher_1.3.3\MSP430Flasher.exe -n MSP430F5529 -w
"C:\msp430_workshop\F5529_usb\workspace\OutOfBox_EmulStorageKeyboards\Debug\OutOfBox_EmulStorageKeyboards.txt" -v
```

```
CLS

C:\ti\MSP430Flasher_1.3.3\MSP430Flasher.exe -n MSP430FR5969 -w
C:\msp430_workshop\FR5969_fram\workspace\OutOfBox_FR5969\Debug\OutOfBox_FR5969.txt" -v
```

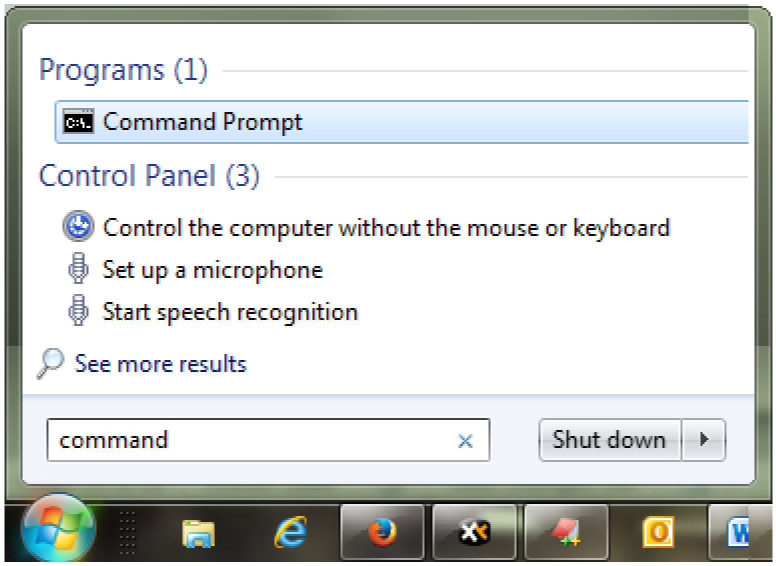
```
CLS

C:\ti\MSP430Flasher_1.3.3\MSP430Flasher.exe -n MSP430FR4133 -w
C:\msp430_workshop\FR4133_fram\workspace\OutOfBox_FR4133\Debug\OutOfBox_FR4133.txt" -v
```

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

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

3. **Open up a DOS command window.**

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



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

4. **Navigate to your `lab_02d_flasher` folder.**

   The DOS command for changing directories is: “*cd*”

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

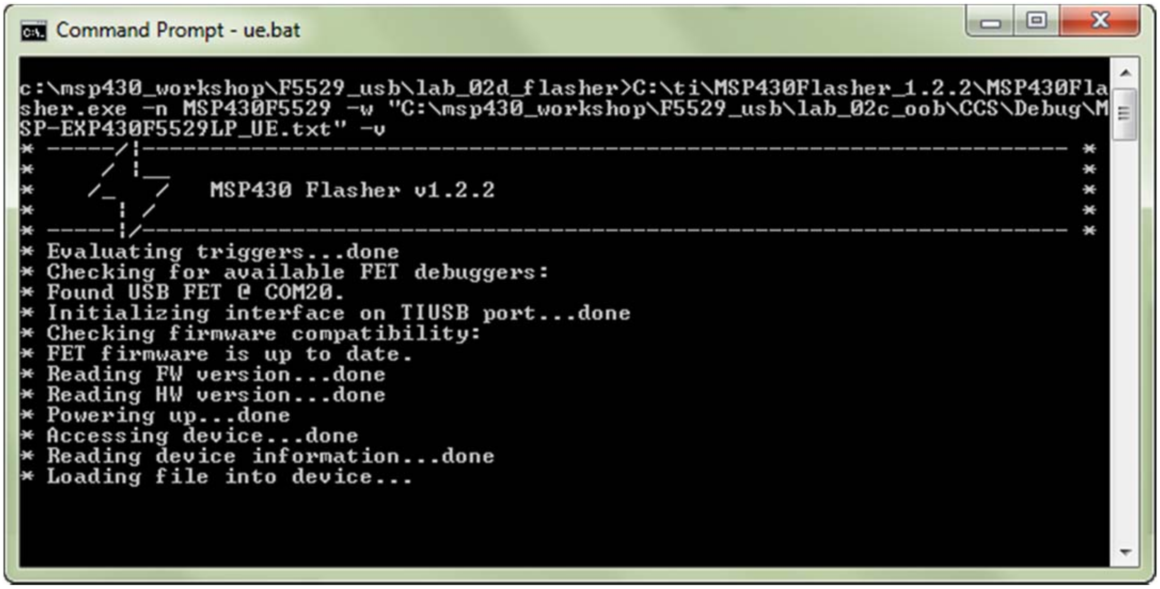
   Once there, you should be able to list the directories contents using the *dir* command.

```
dir
```

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

   `oob.bat ↵`

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



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

6. **Once again, verify the Launchpad program works.**

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

Notes:

Programming Blinky with MSP430Flasher

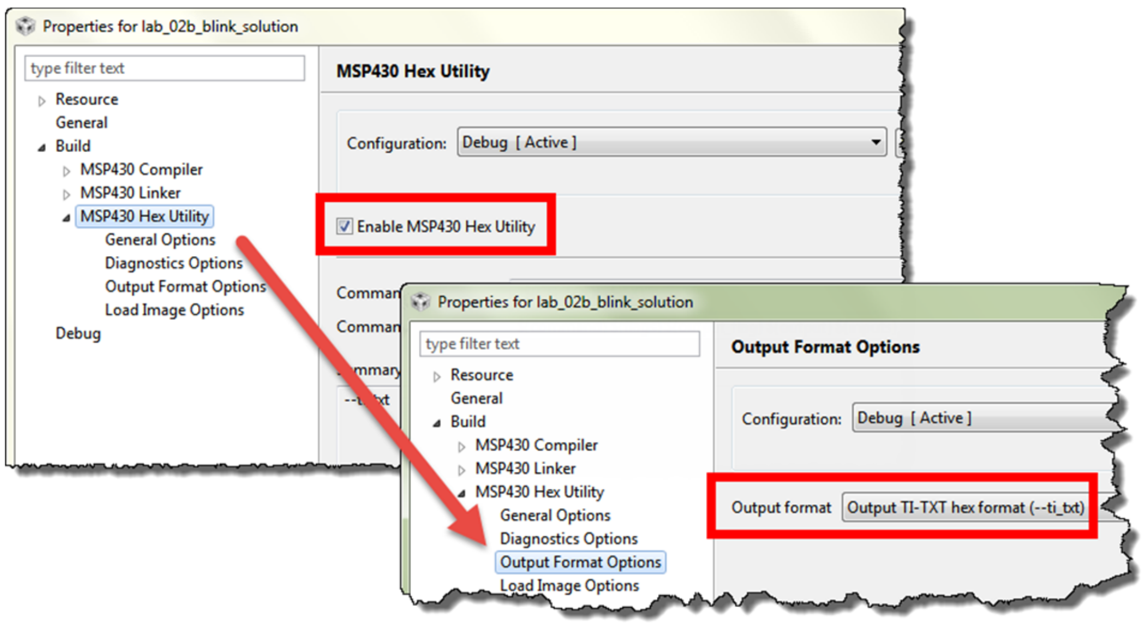
We can use this same utility to burn other programs to our target. Before we can do that, though, we need to create the binary file of our program. The UE app already did this as part of their build process, but we need to make a quick modification to our project to have it build the correct binary format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

7. Open your `lab_02b_blink` project.

8. Open the project properties for you project.

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

9. (CCSv6) Change the following settings in your project, as shown below:



Hint: This procedure is documented at: http://processors.wiki.ti.com/index.php/Generating_and_Loading_MSP430_Binary_Files.

10. 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`

11. Verify that `lab_02b_blink.hex (or lab_02b_blink.txt)` was created in the `/Debug` directory.

```
lab_02b_blink.txt
```

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

```
C:\msp430_workshop\<target>\lab_02d_flasher\blink.bat
```

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

13. Run `blink.bat` from the DOS command window.

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

Cleanup

14. Close your `lab_02b_blink` project.

15. You can also close the DOS command window, if it's still open.

Using GPIO with MSP430ware

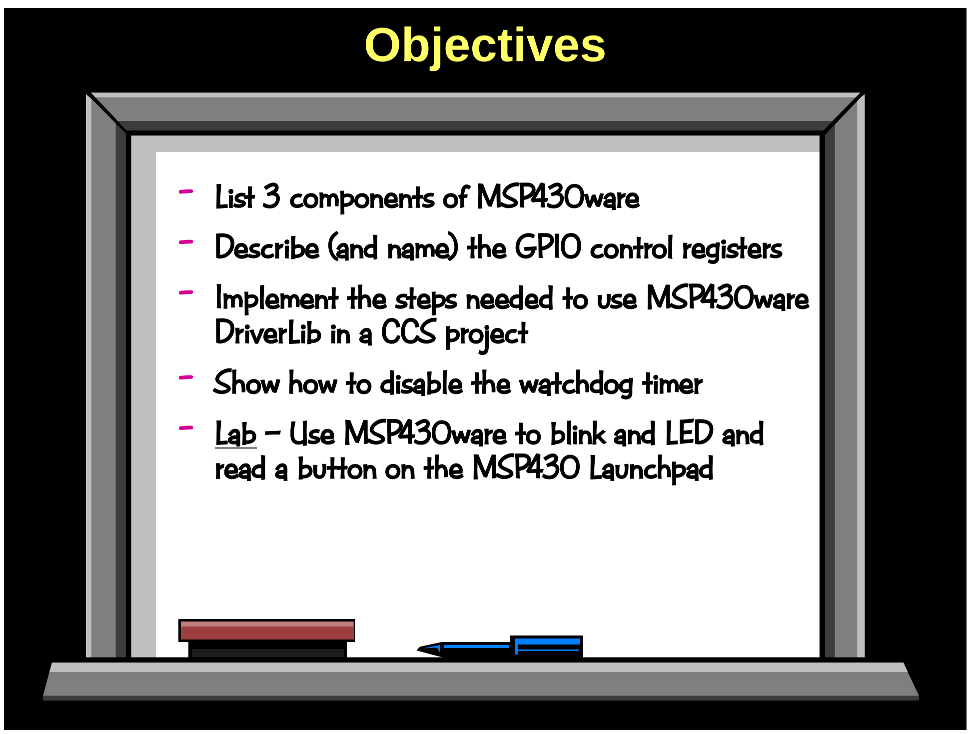
Introduction

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

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

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

Learning Objectives



Chapter Topics

Using GPIO with MSP430ware 3-1

MSP430ware (DriverLib) 3-3
Installing MSP430ware 3-3
DriverLib 3-4
DriverLib Modules 3-5
Programming Methods – Summary 3-5

MSP430 GPIO 3-6
GPIO Basics 3-6
Input or Output 3-7
GPIO Output 3-8
GPIO Input 3-9
Drive Strength 3-10
Flexible Pin Useage (Muxing) 3-11
Pin Selection 3-12
Devices with Multiple Pin Selection Registers 3-13
Port Mapping 3-14
Summary 3-15

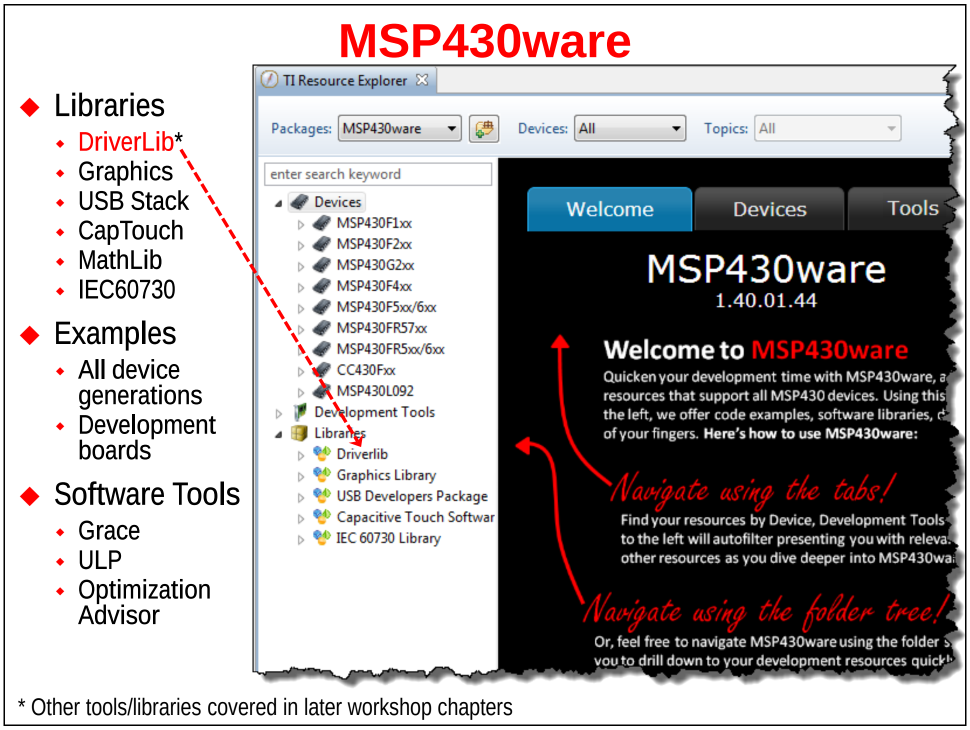
Before We Get Started Coding 3-17
1. #Include Files 3-17
2. Disable Watchdog Timer 3-18
3. Pin Unlocking (Wolverine only) 3-19

Lab 3 3-21
Lab 3a Worksheet 3-23
MSP430ware DriverLib 3-23
GPIO Output 3-23
Lab 3a – Blinking an LED 3-25
Add MSP430ware DriverLib 3-27
Add the Code to `main.c` 3-30
Debug 3-31
Lab 3b – Reading a Push Button 3-33
GPIO Input Worksheet 3-33
File Management 3-35
Add Setup Code (to reference push button) 3-37
Modify Loop 3-38
Verify Code 3-39
Optional Exercises 3-39

Chapter 3 Appendix 3-41

MSP430ware (DriverLib)

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



Installing MSP430ware

When you install MSP430ware as part of CCSv6 – or from the stand-alone MSP430ware installer downloaded directly from the TI website – it is, by default, installed to,

```
C:\ti\msp430\MSP430ware_1_97_00_47\
```

When MSP430ware is updated, they increase the revisions numbers – for example, from 1_60_02_09 to 1_80_01_03. *Note that it's possible that our lab exercises may show a slightly older version of the MSP430ware libraries.*

To update MSP430ware, you by using the auto-update feature of CCS. Alternatively, you can download the stand-alone installer from the MSP430ware webpage.

DriverLib

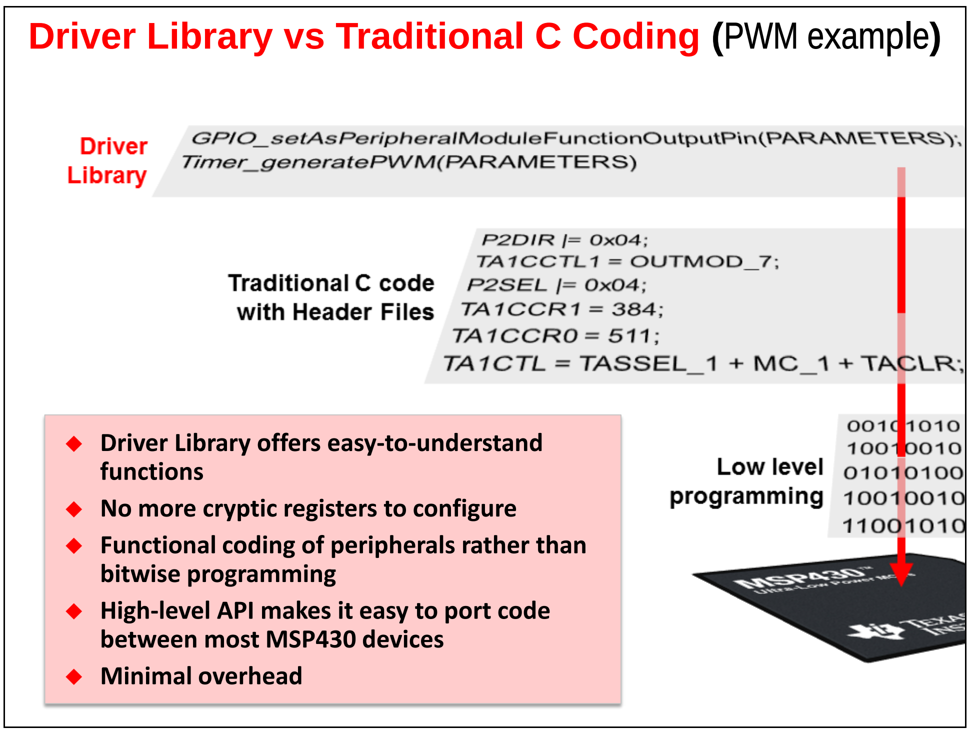
The MSP430ware library used most often in this workshop will be the Driver Library – often called DriverLib.

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

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

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

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



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

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

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

DriverLib Modules

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

MSP430ware DriverLib Modules

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

Used in this chapter (GPIO, WDT)

- 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

Programming Methods – Summary

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

Summary

Name 4 ways to program GPIO:

1. **Using device specific header & command files** (.h/.cmd) — Ch2
2. **MSP430ware DriverLib** (F5xx and FR59xx devices) — Ch3
3. **Energia** — Ch11
4. **Grace graphical driverlib tool** (Value-line and FR58/59xx devices) *

**see Chapter 8 in the "G2553 Value-Line Launchpad Workshop"*

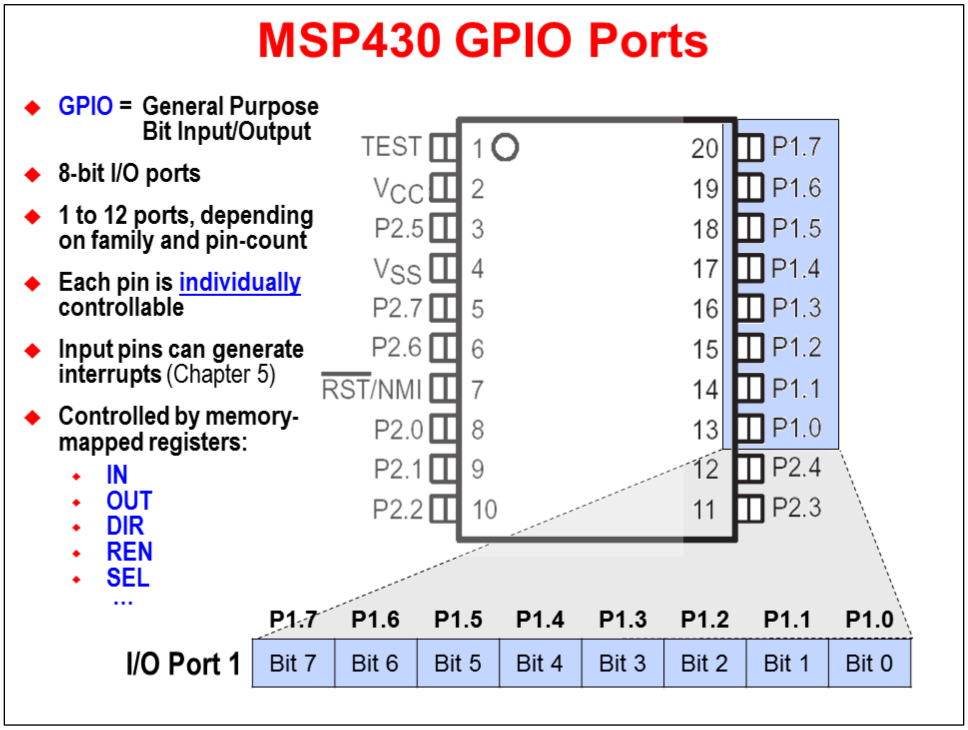
* http://processors.wiki.ti.com/index.php/Getting_Started_with_the_MSP430G2553_Value-Line_LaunchPad_Workshop

MSP430 GPIO

GPIO Basics

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

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



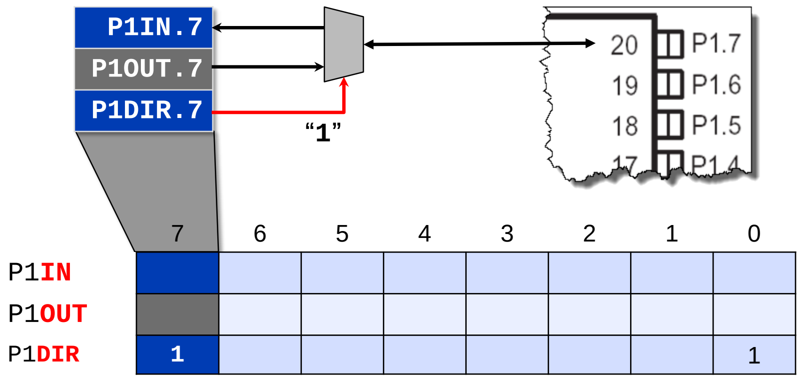
Why did we say pin/bit number? Each I/O pin is individually controllable via CPU registers. For example, if we want to read the input value on P1.0, we can look in bit 0 of the register called P1IN. There are a number of registers used to view and control the I/O pins. In this chapter we'll examine most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

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

Input or Output

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

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 );
```

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

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

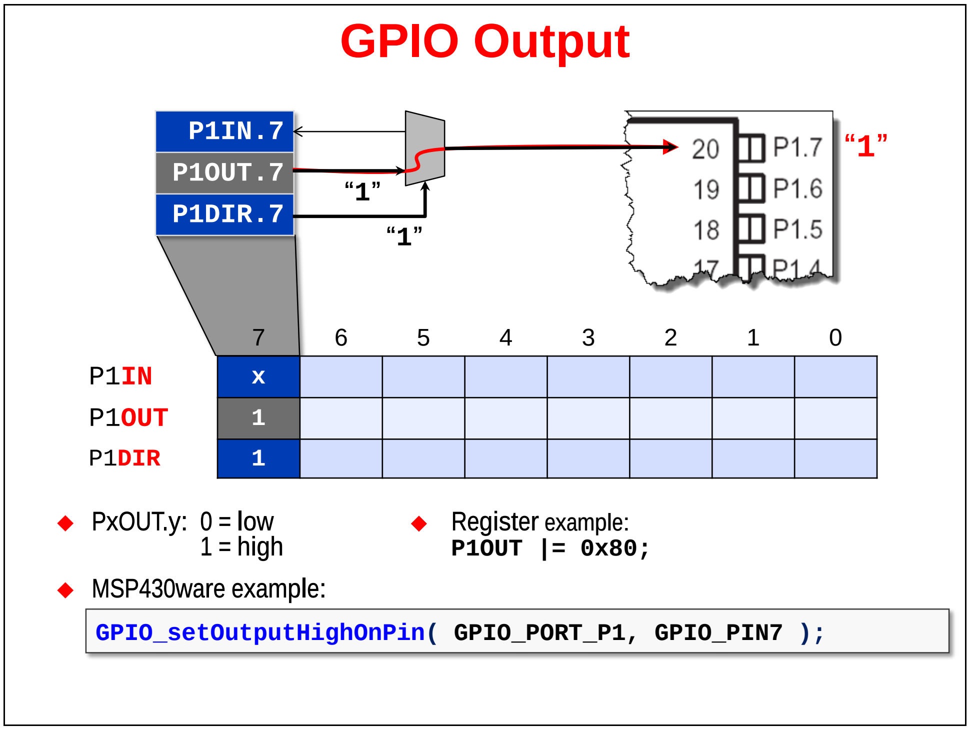
Both methods will end up setting the same registers to the same bit values, though nowadays most teams prefer the more intuitive coding of the DriverLib example. Why? Because you don't really even have to know the register details to understand that pins 0 and 7 are set up as outputs.

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

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

GPIO Output

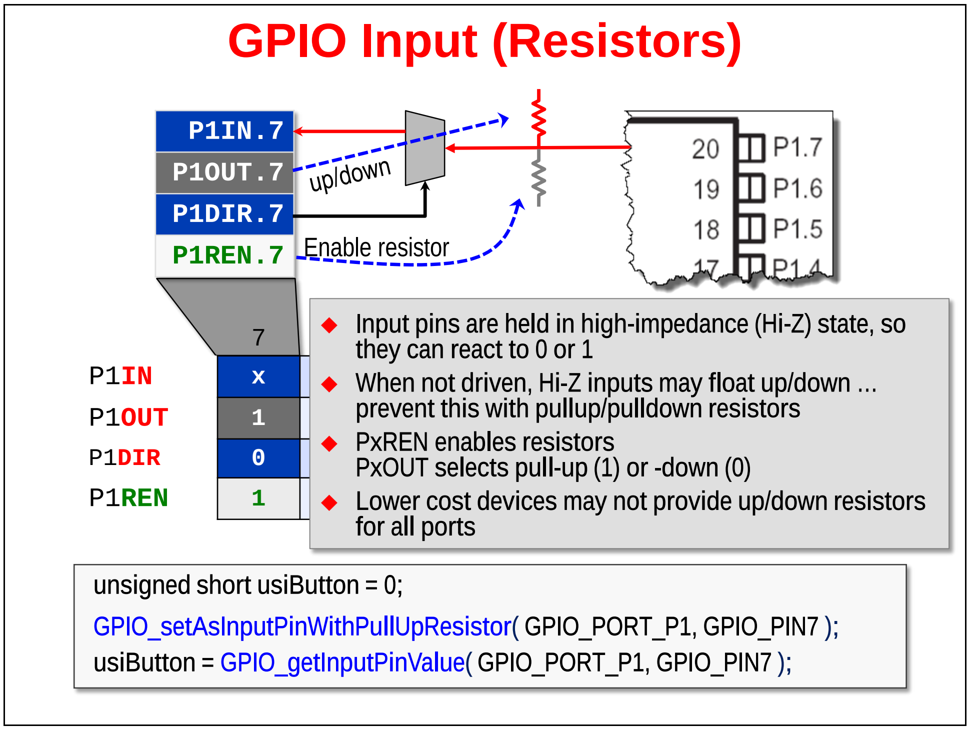
Once you've configured a pin as an output with the PxDIR register, you can set the pins value using the PxOUT register. For P1.7, this would be the P1OUT register.



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

GPIO Input

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



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

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

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

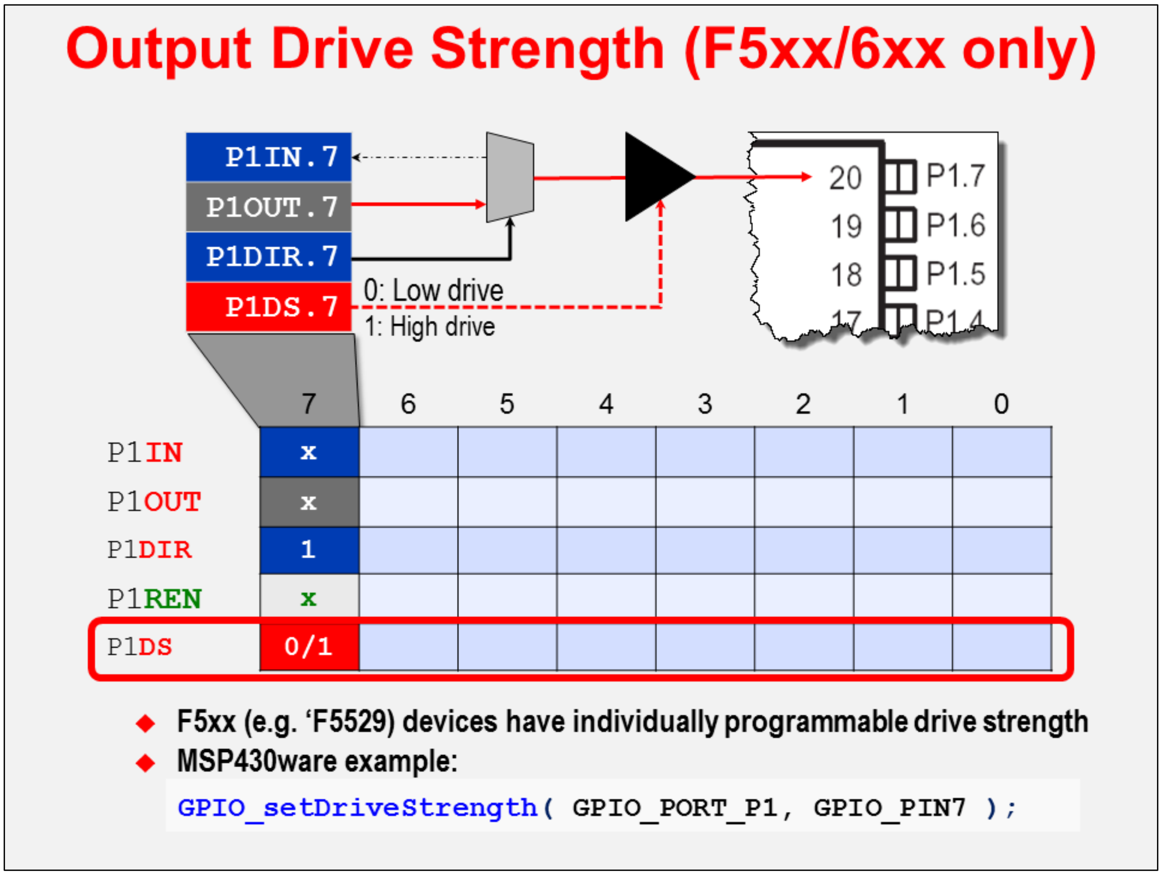
Of course, the easiest way to configure the pull-up or pull-down resistor is to use one of the two GPIO DriverLib functions:

```
GPIO_setAsInputPinWithPullUpResistor()
GPIO_setAsInputPinWithPullDownResistor()
```

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

Drive Strength

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



Flexible Pin Useage (Muxing)

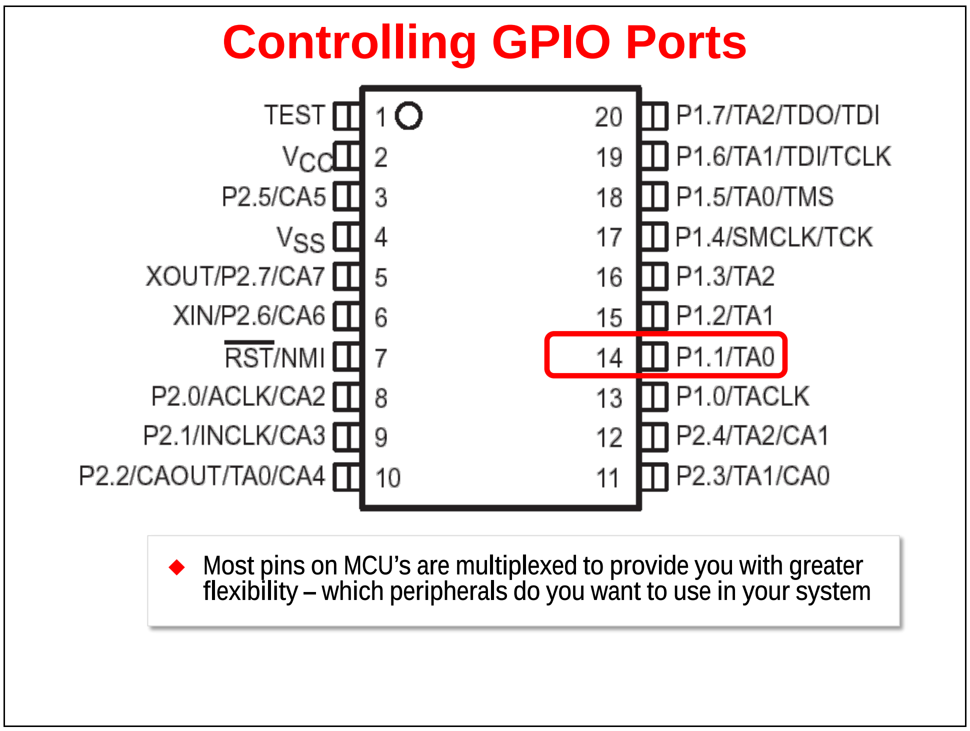
Microcontroller designers have to deal with two conflicting user needs:

More Capability vs Lower Cost

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

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

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



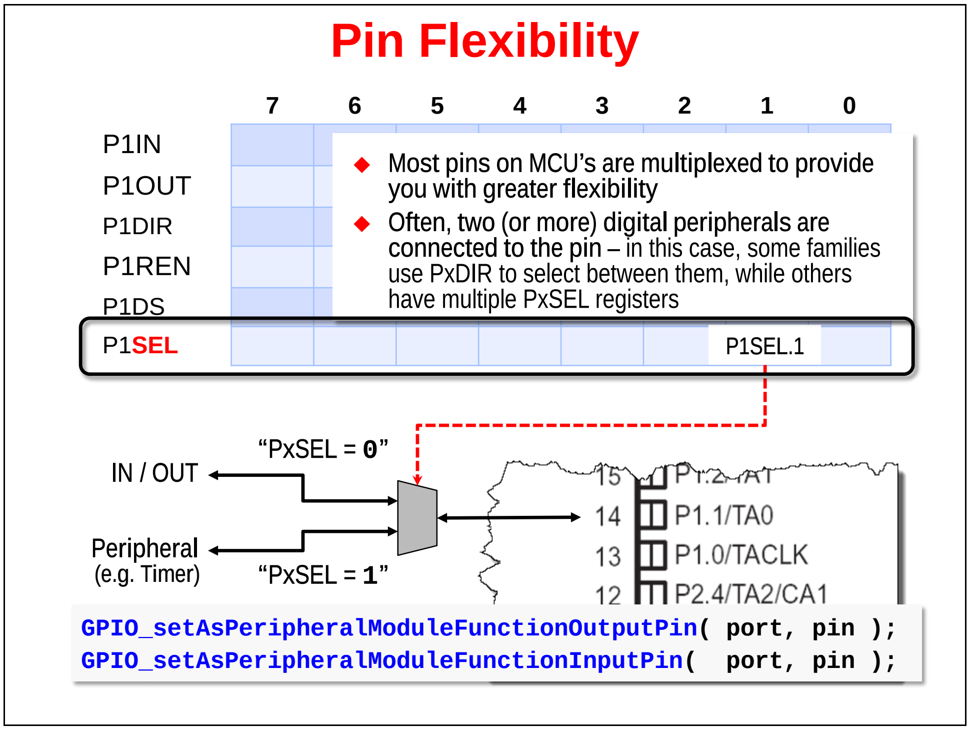
While these pin configurations can be changed at runtime, most users do not find this very useful. The primary reason for this flexibility is so you can choose which features are needed for your specific system.

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

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

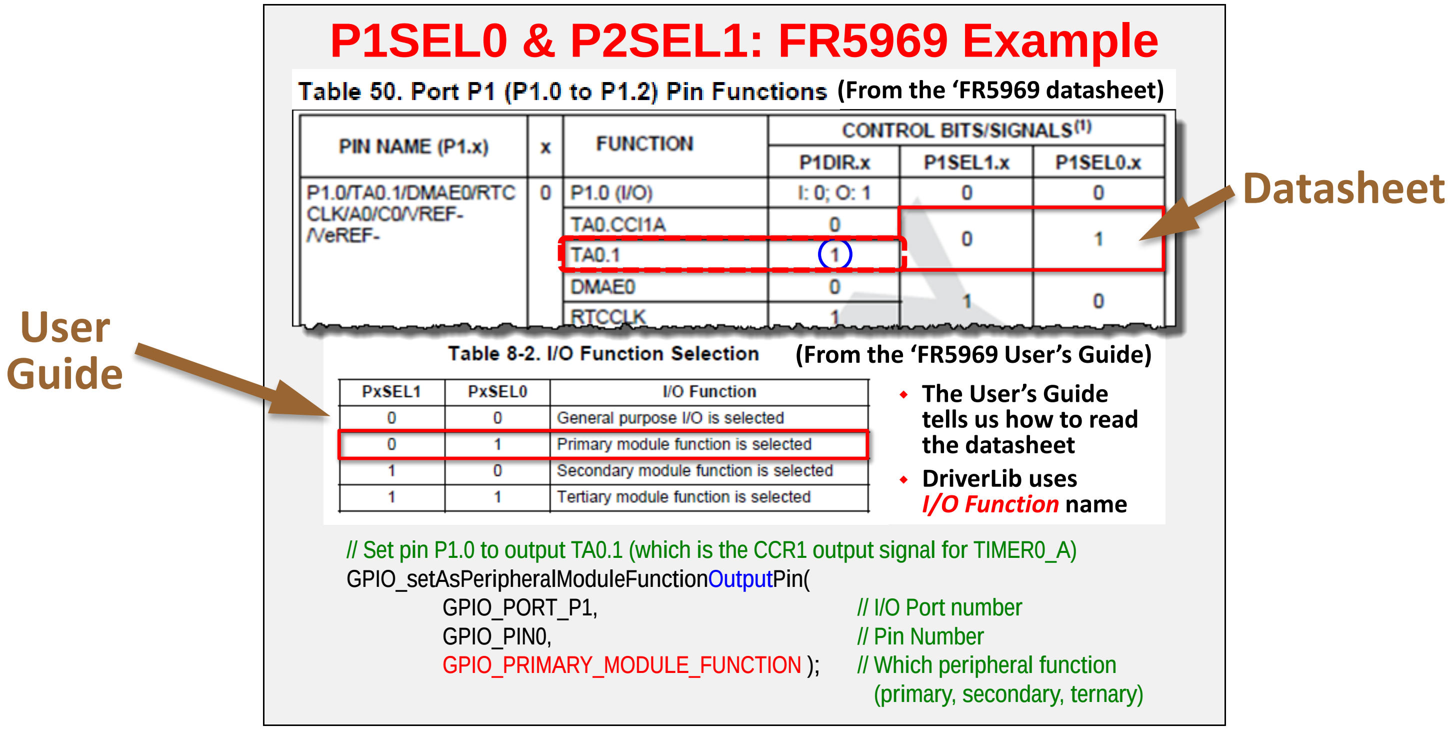
Pin Selection

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



Devices with Multiple Pin Selection Registers

Some MSP430 devices actually have two *pin select* registers, as this is needed to support the greater number of pin mux options.



PIN NAME (P1.x)	x	FUNCTION	CONTROL BITS/SIGNALS(1)		
			P1DIR.x	P1SEL1.x	P1SEL0.x
P1.0/TA0.1/DMAE0/RTCCLK/A0/C0/VREF-/VeREF-	0	P1.0 (I/O)	I: 0; O: 1	0	0
		TA0.CCI1A	0	0	1
		TA0.1	1		
		DMAE0	0	1	0
		RTCCLK	1		

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

```
// Set pin P1.0 to output TA0.1 (which is the CCR1 output signal for TIMER0_A)
GPIO_setAsPeripheralModuleFunctionOutputPin(
        GPIO_PORT_P1,                        // I/O Port number
        GPIO_PIN0,                           // Pin Number
        GPIO_PRIMARY_MODULE_FUNCTION );      // Which peripheral function
                                             //   (primary, secondary, ternary)
```

In the device *User's Guide*, they generically name the different peripheral I/O selections (first, second, and third) with the names:

- Primary
- Secondary
- Tertiary

Because the specific peripheral selections can vary from device-to-device, the detailed options are not described in the *User's Guide*, but rather in each device's *datasheet*. Unfortunately, though, the datasheets do not use the actual Primary/Secondary/Tertiary terminology. That said, you can match the PSEL bit values to figure this out. For example, on the 'FR5969 (in above diagram):

If P1DIR = 1, then TA0.1 is the ***Primary*** selection since P1SEL1.0:P1SEL2.0 = 01

Another way to say this is that because the *datasheet* shows that the TA0.1 PSEL values are "01", we know from the *User's Guide* that this corrolates to the *Primary* function.

The DriverLib functions let you set both "Select" registers with one call. This is done by adding a third argument in which to specify which I/O function you want to enable:

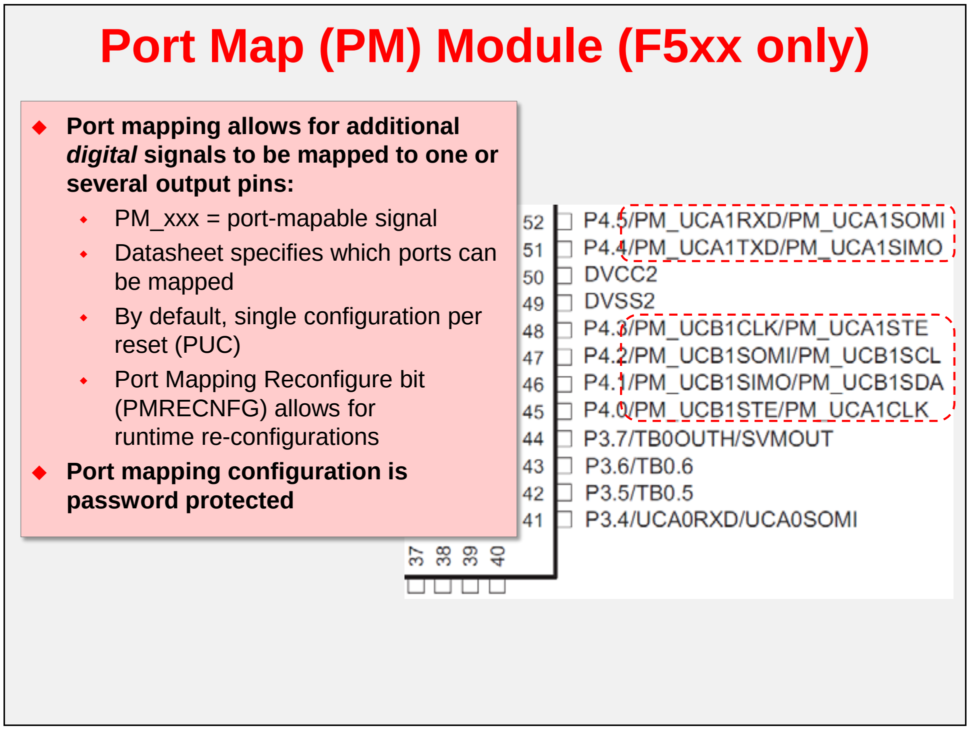
- GPIO_PRIMARY_MODULE_FUNCTION
- GPIO_SECONDARY_MODULE_FUNCTION
- GPIO_TERNARY_MODULE_FUNCTION

You can see an example of this function in the above graphic.

As we've seen, you can figure out which enumeration to use by comparing the selections from both the datasheet and user's guide. *(In fact, 'FR5969 users will do this for the Timer chapter's lab exercise.)*

Port Mapping

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



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

Summary

The following graphic summarizes the GPIO features (and nomenclature) across three MSP430 devices. These three devices provide a good cross-section of MSP430 sub-families:

- The F5529 is an example of the ‘F5xx/6xx series.
- ‘FR5969 is one of the new Wolverine FRAM devices.
- ‘G2553 is the Value-Line processor found on the current Value-Line Launchpad.

GPIO Summary: F5529 vs FR5969 vs G2553

	PA		PB		PC		PD		PJ*	Reset Value (PUC)
	P1†	P2	P3	P4	P5	P6	P7	P8	(4-bit)	
PxIN	All Four Devices support Ports 1 and 2		F5529 FR4133 FR5969 (only)		F5529 (P8 x3-bits) FR4133 (P8 x12-bits)				F55 & FR59	undef
PxOUT										unchg
PxDIR										0x00
PxREN										0x00
PxDS										0x00
PxSEL										0x00
PxIV			FR5969 (only)							0x00
PxIES										unchg
PxIE										0x00
PxIFG										0x00

F5529/FR4133 only (80-pin)
FR5969 only (48-pin)
G2553 only (20-pin)

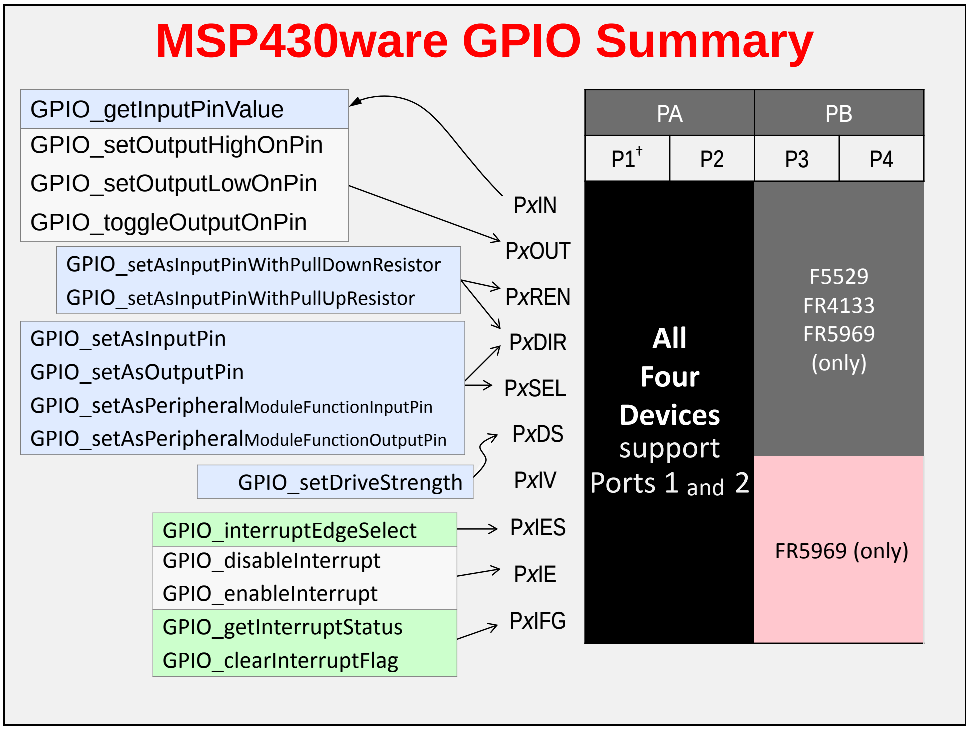
* PJ: 4-bits shared with JTAG pins
† P1: 4-bits shared with JTAG pins (‘G2553)

- 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

What can we derive from the table above?

- The various GPIO memory-mapped registers are shown here listed down the first column. Most of these registers were described in the preceding discussion.
- All three devices (and most all MSP430 devices) contain two 8-bit I/O ports (P1, P2) which provide the GPIO functionality – including interrupt inputs. We demonstrated this above by using the ‘black’ fill under ports P1 and P2; notice it covers every register’s row.
- Alternatively, you can program ports 1 and 2 simultaneously by writing to port “PA”. This means by writing to PAOUT, you can concurrently configure the outputs of all 16-pins.
- The ‘G2553 Value-Line device only includes P1 and P2. (There just aren’t enough pins on this device to support more I/O ports.)
- The new ‘FR5969 Wolverine devices added interrupt support for PB (i.e. ports P3 & P4).
- Only the ‘F5529, of our three example devices, has enough pins to support ports P5 – P8. Note, though, that the P8 port only contains 3-bits.
- Port PJ is unique. For these devices, it’s only 4-bits wide. These signals represent the 4 JTAG pins; although, any of these four pins can also be reconfigured for GPIO.

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



Before We Get Started Coding

Getting Your Program Started

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

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

1. #Include Files

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

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

Include Files

TIMER_A	AES	GPIO
TIMER_B	CRC	PM
TIMER_D	DMA	[illegible]
WDT_A	MPY32	[illegible]

- **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`**

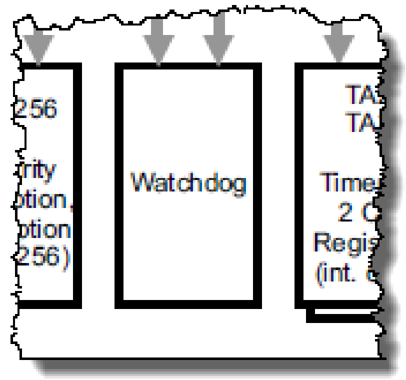
```
#include <driverlib.h>
GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN7);
```

2. Disable Watchdog Timer

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

Disable WatchDog Timer

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



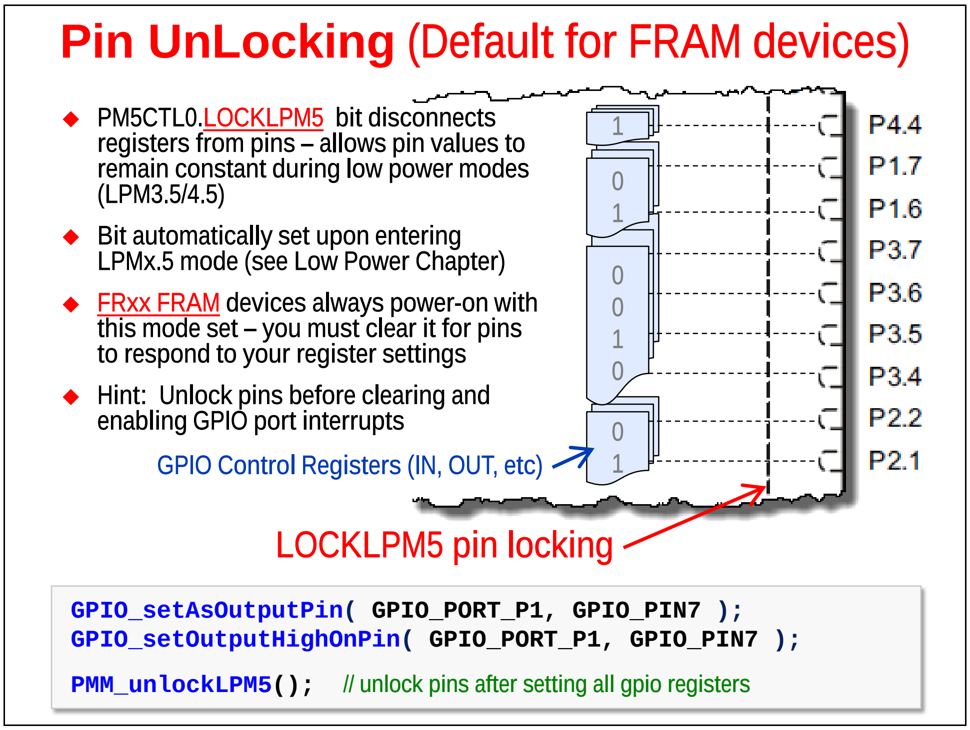

```
#include <driverlib.h>

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

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

3. Pin Unlocking (Wolverine only)

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

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



Many devices (prior to the FRAM), such as the 'F5529, provide the pin-locking feature – although, it's not enabled by default. The new 'FRxx (FRAM) devices, though, have this feature enabled by default … therefore, the pins are always locked at power-up.

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

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

Notes

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.

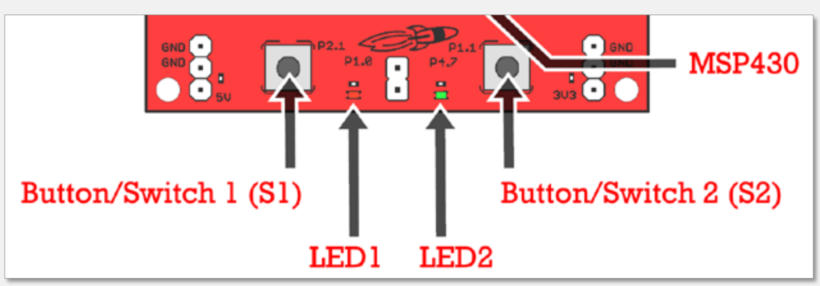
Lab 3 – Blink with MSP430ware

- **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 blinking LED
  - Use CCS 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 Chapter 5)

Here's a helpful Port/Pin summary for the Launchpad's LEDs and Buttons.

Launchpad Pins for LEDs/Switches

Launchpad →	F5529	FR4133	FR5969	LED Color
LED1	P1.0	P1.0	P4.6	Red LED (with Jumper)
LED2	P4.7	P4.0	P1.0	Green LED
Button 1	P2.1	P1.2	P4.5	
Button 2	P1.1	P2.6	P1.1	



Lab3 Abstract

Lab 3a – GPIO

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

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

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

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

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

Lab 3b - Button

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

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

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

Note: "Polling" the button is very inefficient!

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

Hint: The MSP430 DriverLib Users Guide is a good resource to help you answer the questions on the next page. It can be found in the MSP430ware "doc" folder:

\MSP430ware_1_97_00_47*driverlib**driverlib**doc**MSP430F5xx_6xx*\

\MSP430ware_1_97_00_47*driverlib**driverlib**doc**MSP430FR2xx_4xx*\

\MSP430ware_1_97_00_47*driverlib**driverlib**doc**MSP430FR5xx_6xx*\

Lab 3a Worksheet

MSP430ware DriverLib

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

   ____

2. To use the MSP430ware GPIO and Watchdog API, which header file needs to be included in your source file? *(Hint: We discussed this during the presentation in the "Before We Get Started" section.)*

   `#include < ______________________ >`

3. Which DriverLib function stops the Watchdog timer?
   (Hint: Look in DriverLib User's Guide or the "Before We Get Started" section of this chapter.)

   ____ ;

GPIO Output

4. Which I/O pin on Port 1 is connected to an LED (on your Launchpad)?

   ____

   What two GPIO DriverLib functions are required to initialize this GPIO pin (from previous question) as an output and set its value to "1"?
   (Hint: Look at the chapter slides titled: "PxDIR (Pin Direction)" and "GPIO Output".)

   ____;

   ____;

FR4133

FR5969

For FRAM devices, what additional function is needed to make the I/O work (i.e. to connect the GPIO registers to the pin)?

____;

5. Using the _delay_cycles() intrinsic function (from the last chapter), write the code to blink an LED with a 1 second delay setting the pin (P1.0) high, and then low?
   (Hint: What two GPIO functions set an I/O Pin high and 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 );
}
```

Double-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:

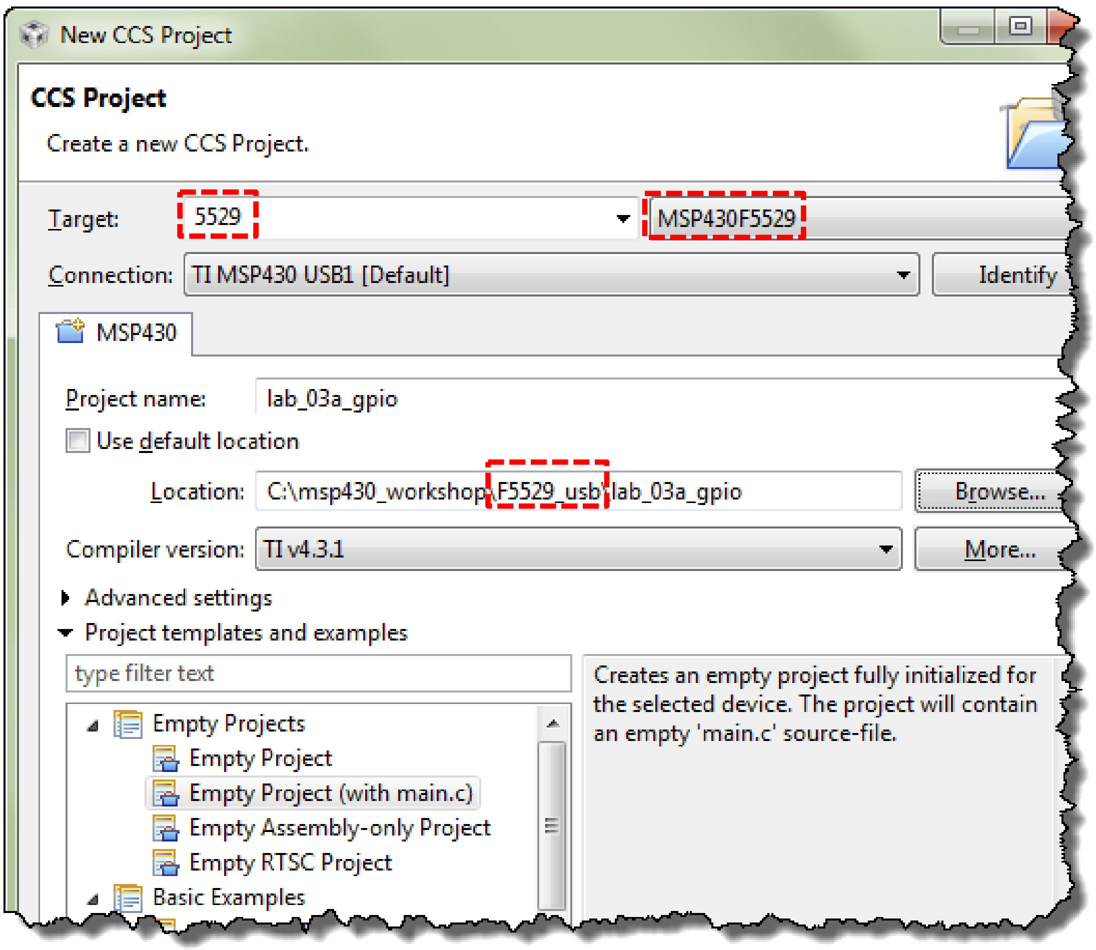
   `Right-click on the project and select “Close Project”`

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

2. **Create a new project.**

   Name the new project: `lab_03a_gpio`

   Fill in the new project dialog as shown below, then click Finish.



   If you have questions about creating CCS projects, you can refer back to Lab 2b.

Note: If you’re working with the ‘FR5969 or ‘FR4133, please replace the F5529 references shown above with those required for your Launchpad.

Also, your compiler version may be more recent than the one shown in the screen capture.

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

Although this is not required, you can replace this “register-based” code with the DriverLib function. Either way works fine. *If you want to use DriverLib, please reference your Worksheet answer #3 (on page 3-23).*

4. 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 `#include <msp430.h>`. You can replace this with the DriverLib #include. It’s OK to have both of them, but the DriverLib header file references `msp430.h` for you.

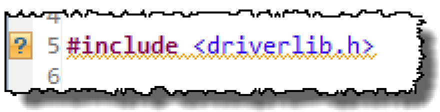
5. Build your program.

Even though we haven’t added any code yet, try building the program.

???

6. Why the build error?

Depending upon which version of CCS you have, you might have seen a question mark (?) in front of the #include before you built the program.



When building your program, you should have received a build error. What caused this error?

____

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.

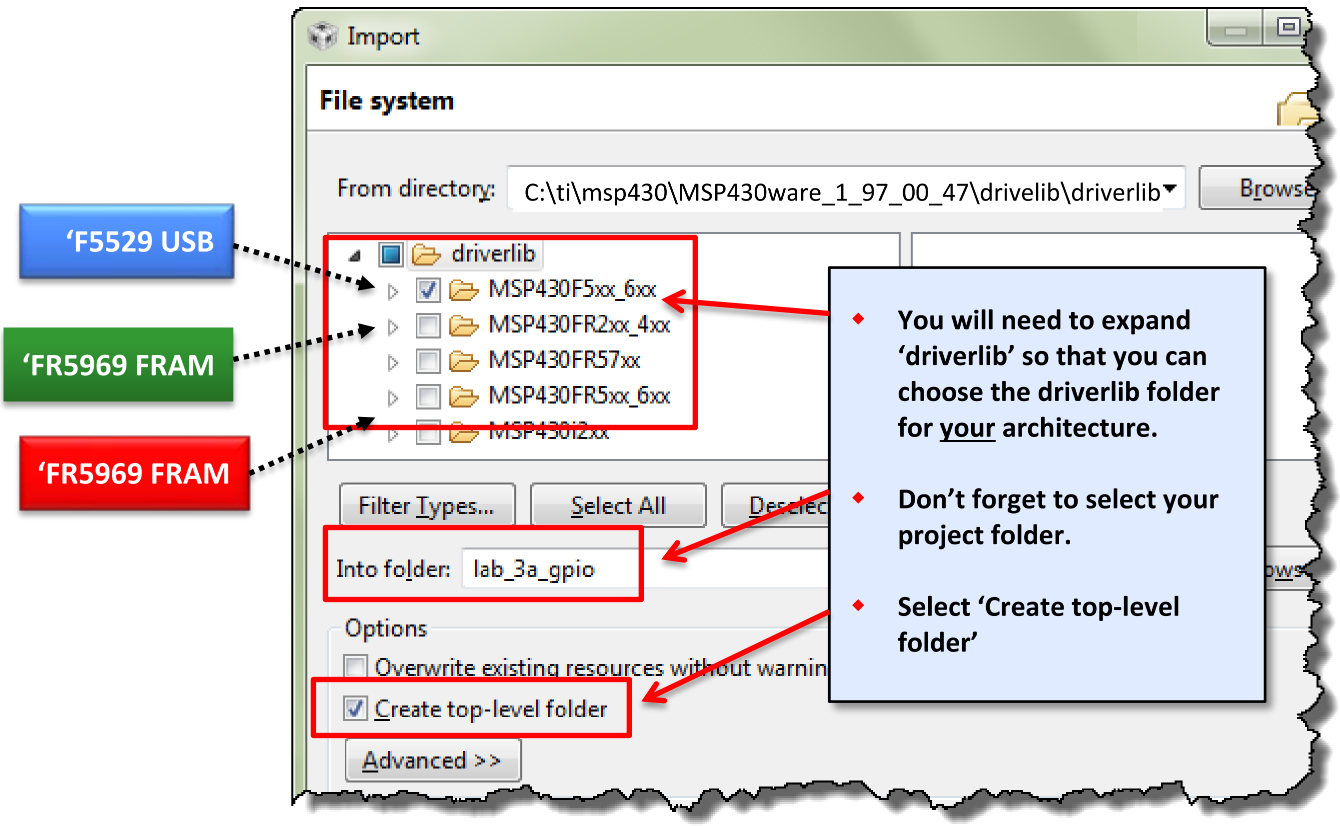
Adding the DriverLib library is a two-step process:

- 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 → Import... → General → File System`

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



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

`▷ driverlib/MSP430F5xx_6xx`

or one of these, depending on which Launchpad you're using:

`driverlib/MSP430FR5xx_6xx`

`driverlib/MSP430FR2xx_4xx`

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

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

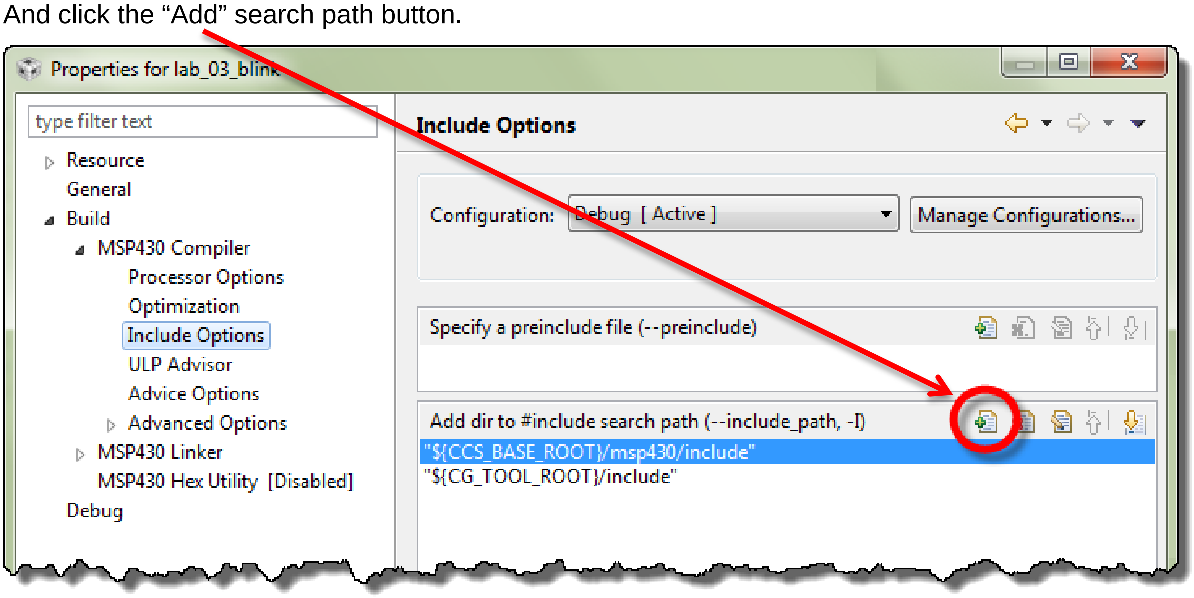
Open the Include Options and add the directory to #include search path:

`Right-click project → Properties`

Then select:

`Build → MSP430 Compiler → Include Options`

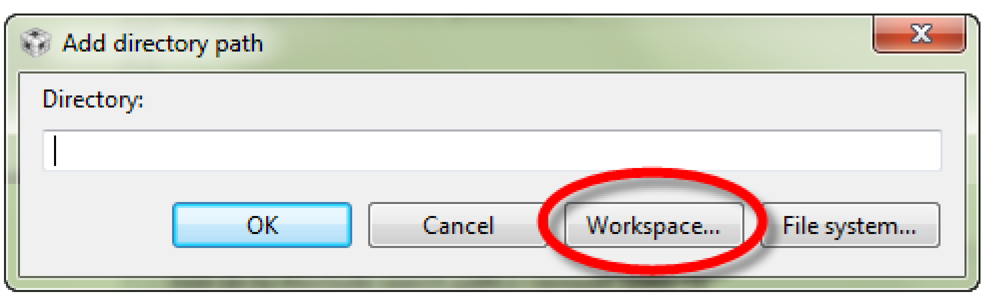
And click the “Add” search path button.



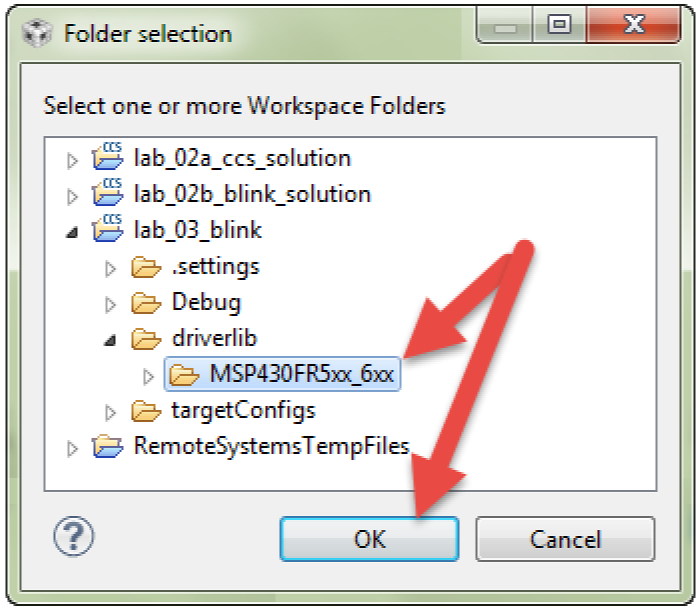
When the “Add directory path” dialog appears, you can add the path manually:

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

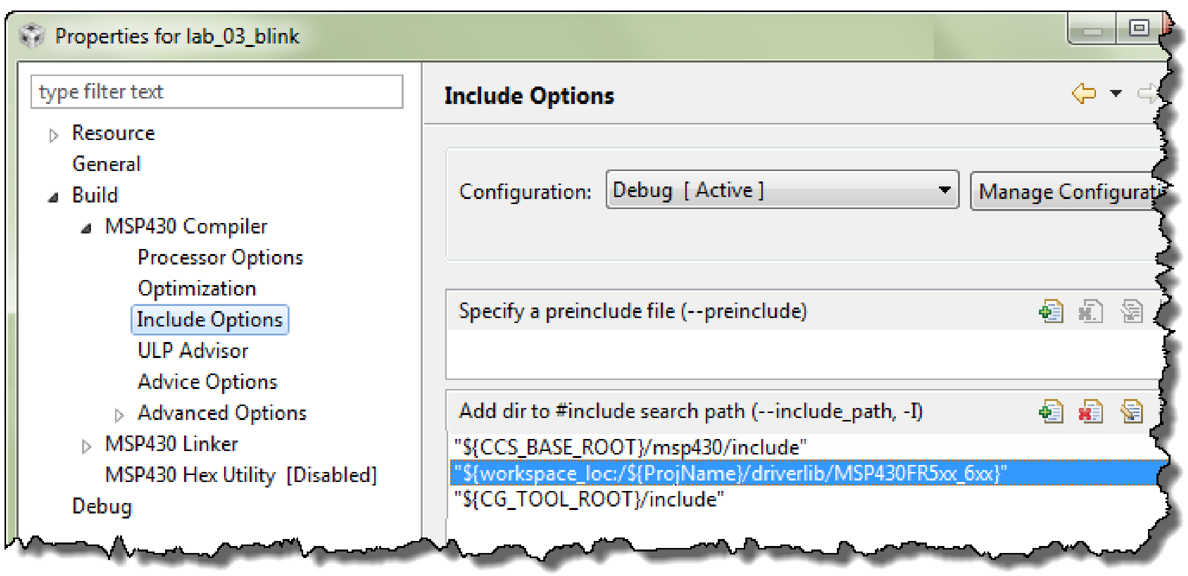
or minimize typing errors by selecting it from the **Workspace** (as shown below).



Select the driverlib folder and click OK.



Clicking OK once more returns us to the project's properties. Notice that the driverlib directory – found inside the workspace & project directory – has now been added to the project #include search path.



After inspecting the new search path, you can close the project properties dialog.

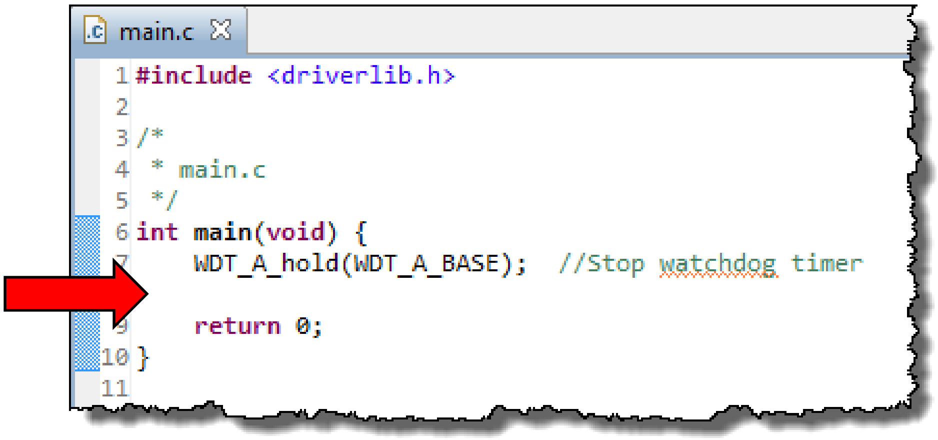
9. Click the build toolbar button to verify that your edits, thus far, are correct.

Add the Code to `main.c`

10. Set up P1.0 as output pin.

Reference Worksheet question #4 (page 3-23).

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



Hint: If you're using the 'FR5969 or 'FR4133 Launchpad, don't forget to add the line of code which unlocks the pins. *(Reference Worksheet question 4b (page 3-23).*

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

Reference Worksheet question #0 (page 3-24). Begin the while{} loop after the code you wrote in the previous step (to set up the output pin).

Also, don't forget to add the #define for "ONE_SECOND" towards the top of the file.

12. Build your program with the Hammer icon.

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

13. Load and Run your program.

Click the *Debug* button to start the debugger and download your program. Then click the *Resume* button to run the code.

Does your LED flash? ____

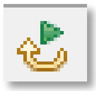
If it doesn't, let's hope following debug steps help you to track down your error.

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.

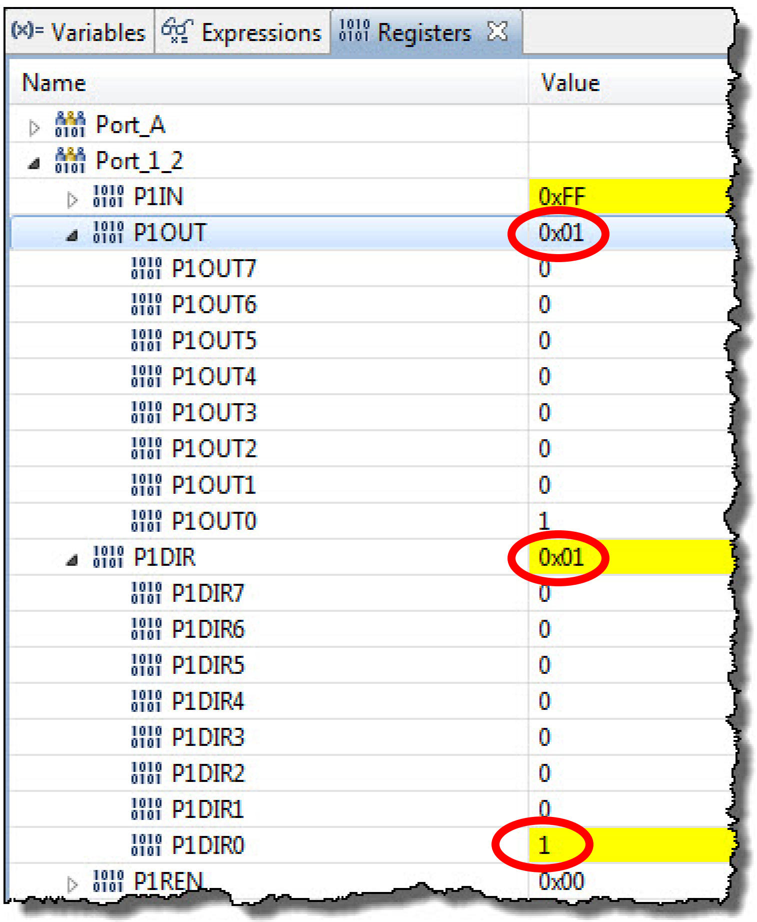
16. Open the Registers window and view P1DIR and P1OUT. Then single-step past the GPIO DriverLib functions.

```
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:



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

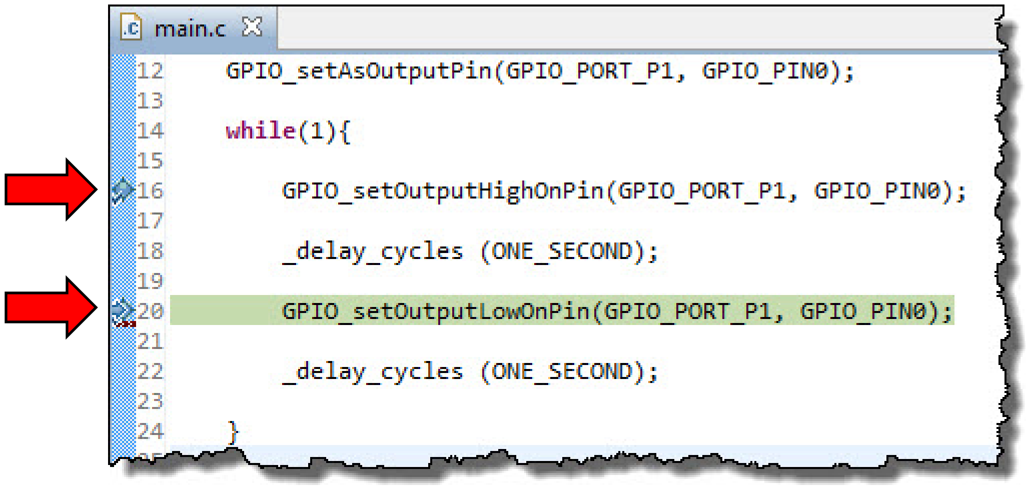
You should see the P1OUT register change as you step over the appropriate function.

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:



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

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

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

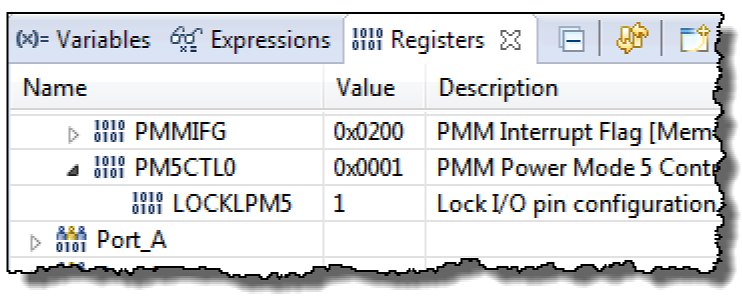
---

19. If you're using an FRAM Launchpad, you may want to exaine the PM5CTL0 register.

FR4133

FR5969

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



a) If running, suspend your program.

`Alt-F8`

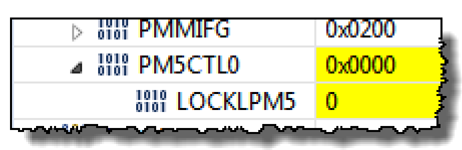
b) Open the register window and display the LOCKLPM5 bit.

c) Perform a *Hard Reset*.

`Run` → `Reset` → `Hard Reset`

d) Then, restart the program.

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



the

Lab 3b – Reading a Push Button

GPIO Input Worksheet

1. What three different DriverLib functions can set up a GPIO pin for input?

   Hint: One place to look would be the MSP430 DriverLib Users Guide found in the MSP430ware folder: \MSP430ware_1_97_00_47*\driverlib\driverlib\doc\MSP430F5xx_6xx*
   \MSP430ware_1_97_00_47*\driverlib\driverlib\doc\MSP430FR2xx_4xx*
   \MSP430ware_1_97_00_47*\driverlib\driverlib\doc\MSP430FR5xx_6xx*

   ____

   ____

   ____

2. What can happen to an input pin that isn't tied high or low? *(Hint: See "GPIO Input" topic on pg 3-9.)*

   ____

   ____

3. Which I/O pin on Port 1 is connected to a Switch (on your Launchpad)?

   ____

   Assuming you need a pull-up resistor for a GPIO input, write the line of code required to setup this pin as an input: *(Hint: See "GPIO Input" topic on pg 3-9.)*

   ______________________________;

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

```
volatile unsigned short usiButton1 = 0;
while(1) {
   // Read the pin for push-button 2
   usiButton1 = ___________________________________________________;
   if ( usiButton1 == GPIO_INPUT_PIN_LOW ) {
      // If button is down, turn on LED
      GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
   }
   else {
      // Otherwise, if button is up, turn off LED
      GPIO_setOutputLowOnPin(  GPIO_PORT_P1, GPIO_PIN0 );
}  }
```

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

Check your answers against ours … see the Chapter 3 Appendix

File Management

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

<u>Use the driverlib project template</u>

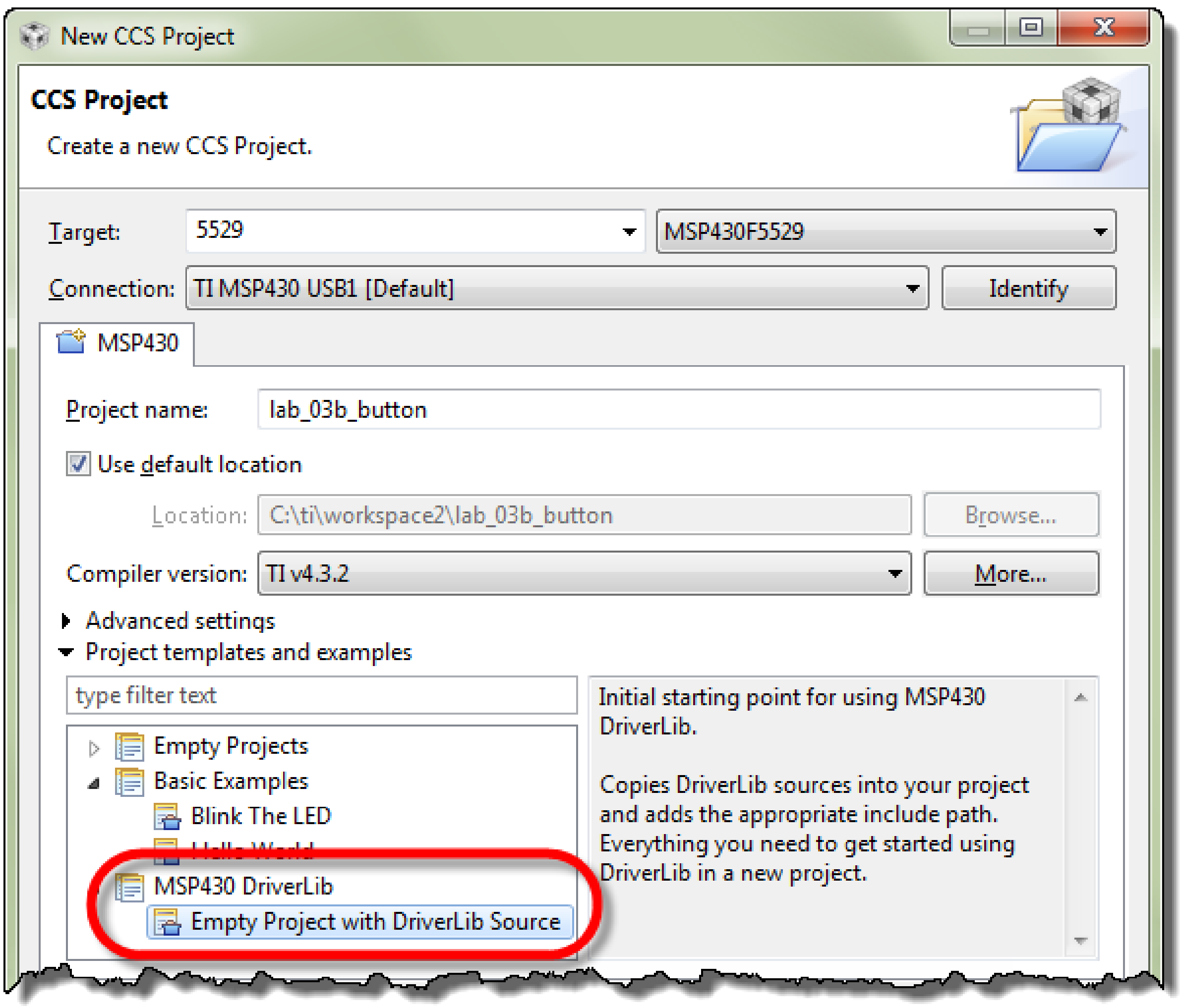
1. Terminate the debugger (if it's still running).

2. Create a new driverlib project.

There are a couple different ways to import the example projects, but we've picked the easiest method, using the DriverLib project template.

Create a new project – as you have done previously – but in this case you should select the template, as shown below:

```
Empty Project with DriverLib Source
```



3. 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.

Much thanks to the MSP430ware team for making this so easy!

Copy our code from the previous project

4. Delete the ‘empty’ `main.c` from the new project.

5. Copy/Paste `main.c` from lab_03a_gpio **to** lab_03b_button**.**

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

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

6. Close the previous lab: `lab_03a_gpio.`

As we’ve learned, this should close the .c source files associated with those projects, 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”.
```

7. Make sure the new project is active and then build the new lab, just to make sure everything was copied correctly.

Add Setup Code (to reference push button)

8. **Open `main.c` for editing.**

9. **Before the main() function, add the global variable: `usiButton1`**

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

Let's explain some of our choices:

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

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

Does this variable change outside the scope of C? ________________________________

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

unsigned short … You tell us, why did we pick that? ______________________________

____

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

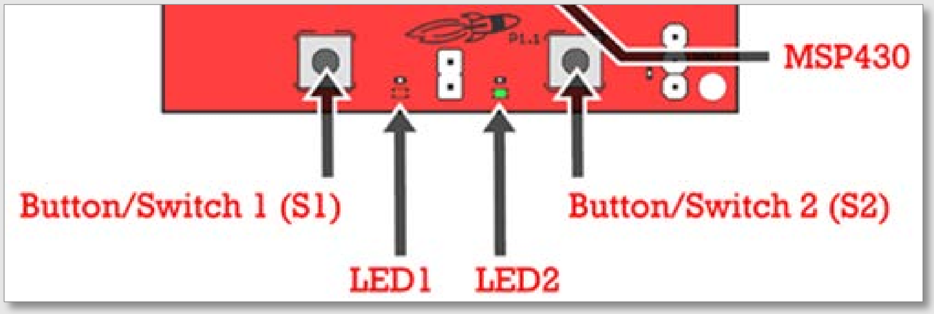
=0 … well, that's just good style. You should always initialize your variables. Many embedded processor compilers do not automatically initialize variables for you.

10. **In *main()*, add code to set push button as an input with pull-up resistor.**

This setup code should go before the while{} loop. *(And for the FRAM devices, we recommend placing this code before the unlock LPM5 function.)*

And don't forget, this code was the answer to Worksheet question #3 (page 3-33).

Hint: We should have recommended bringing a magnifying glass to read the silk screen on the Launchpad board. It can be difficult to read the button (and LED) labels. It may easier to reference the Quick Start sheet that came with your Launchpad.



Modify Loop

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

Comment out (or delete) LED blinking code and replace it with the code we created in the Worksheet question #0 (page 3-34).

At this point, your `main.c` file should look similar to following code. The 'FR4133 code uses a different pin number (P1.2).

```
// ----------------------------------------------------------------
// main.c  (for lab_03b_button project) ('FR5969 Launchpad)
// ----------------------------------------------------------------

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

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

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

    // Set pin P1.0 to output direction and initialize low
    GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );
    GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );

    // Set switch 2 (S2) as input button (connected to P1.1)
    GPIO_setAsInputPinWithPullUpResistor( GPIO_PORT_P1, GPIO_PIN1 );

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


    while(1) {
        // Read P1.1 pin connected to push button 2
        usiButton1 = GPIO_getInputPinValue ( GPIO_PORT_P1, GPIO_PIN1 );

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

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

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

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

Verify Code

12. Build & Load program.

13. 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.

14. 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.

15. Run the program.

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

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

Optional Exercises

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

  Without the resistor, is the pushbutton's value always consistent? (yes / no) ________________
- Try using the other LED on the board …
- … or the other pushbutton.

Notes

Chapter 3 Appendix

Lab3a – Worksheet

1. **Where is your MSP430ware folder located?**

   Most likely: `C:\ti\msp430\MSP430ware_1_97_00_47\`

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

   #include < driverlib.h >

3. What DriverLib function stops the Watchdog timer**?**

   `WDT_A_hold( WDT_A_BASE )` ;

4a. **Which I/O pin on Port 1 is connected to an LED (on your Launchpad)?**

   F5529/FR5969/FR4133: P1.0

4b. **What two GPIO DriverLib functions are required to initialize this GPIO pin (from previous question) as an output and set its value to "1"?**

   `GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 )` ;

   `GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 )`;

4c. For the FRAM devices, what additional function is needed to make it work (i.e. to connect the I/O to the pin)**?**

   `PMM_unlockLPM5();` ;

Lab3a – Worksheet

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

```
#define ONE_SECOND 800000

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

Lab3b – Worksheet

1. What 3 function options can be used to set a pin for GPIO input?
   Hint, one place to look would be the MSP430 Driverlib Users Guide found here:
   `\MSP430ware_1_97_00_47\driverlib\doc\<target>\`

   `GPIO_setAsInputPin()`

   `GPIO_setAsInputPinWithPullDownResistor()`

   `GPIO_setAsInputPinWithPullUpResistor()`

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

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

3a. Which I/O pin on Port 1 is connected to a Switch (on your Launchpad)?

   `F5529/FR5969: P1.1` `FR4133: P1.2`

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

   `GPIO_setAsInputPinWithPullUpResistor ( GPIO_PORT_P1, GPIO_PIN1);`

   or `GPIO_setAsInputPinWithPullUpResistor ( GPIO_PORT_P1, GPIO_PIN2);`

Lab3b – Worksheet

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

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

For 'FR4133 use GPIO_PIN2

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

   "Polling"

MSP430 Clocks & Initialization

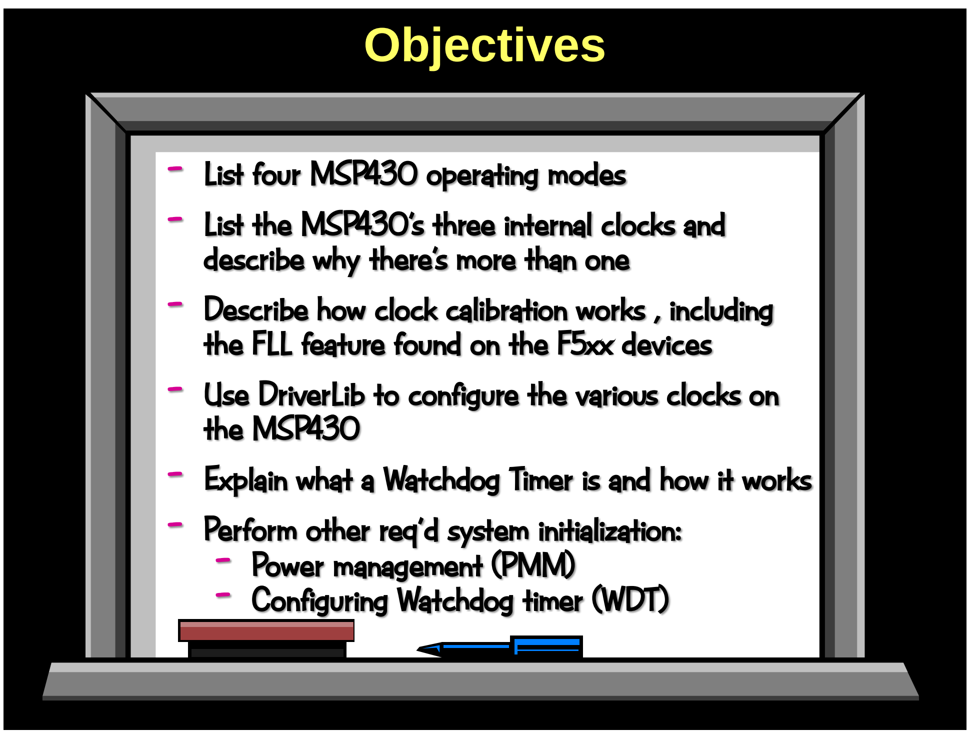
Introduction

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

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

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

Learning Objectives



Chapter Topics

MSP430 Clocks & Initialization 4-1

Operating Modes (Reset → Active) 4-3
BOR 4-3
BOR → POR → PUC → Active (AM) 4-4

Clocking 4-6
What Clocks Do You Need? 4-6
MCLK, SMCLK, ACLK 4-8
Oscillators (Clock Sources) 4-9
Clock Details (by Device Family) 4-11
Using MSP430ware to Configure Clocking 4-16
Additional Clock Features 4-18

DCO Setup and Calibration 4-21
How the DCO Works 4-22
Factory Callibration (FR5xx, G2xx) 4-26
Runtime Calibration (F4xx, F5xx, F6xx) 4-28
FR2xx/4xx DCO Calibration 4-31
VLO 'Calibration' 4-32

Other Initialization (WDT, PMM) 4-33
Watchdog Timer 4-34
PMM with LDO, SVM, SVS, and BOR 4-35
Operating Voltages 4-37
Summary 4-38
Initialization Summary (template) 4-40

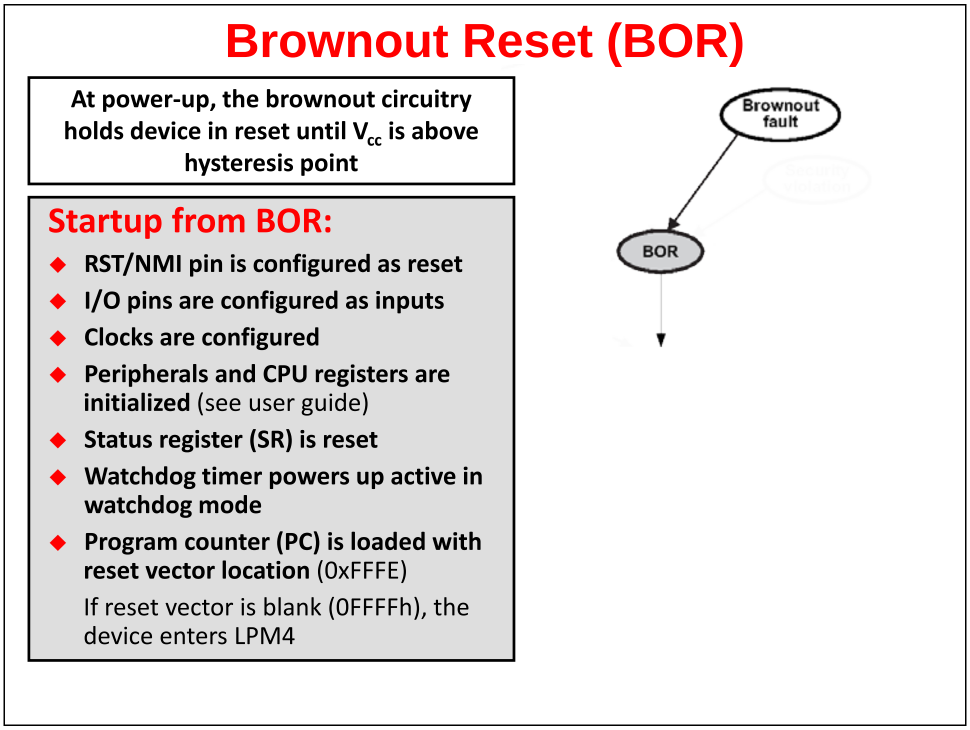
Lab Exercise 4-41

Operating Modes (Reset → Active)

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

BOR

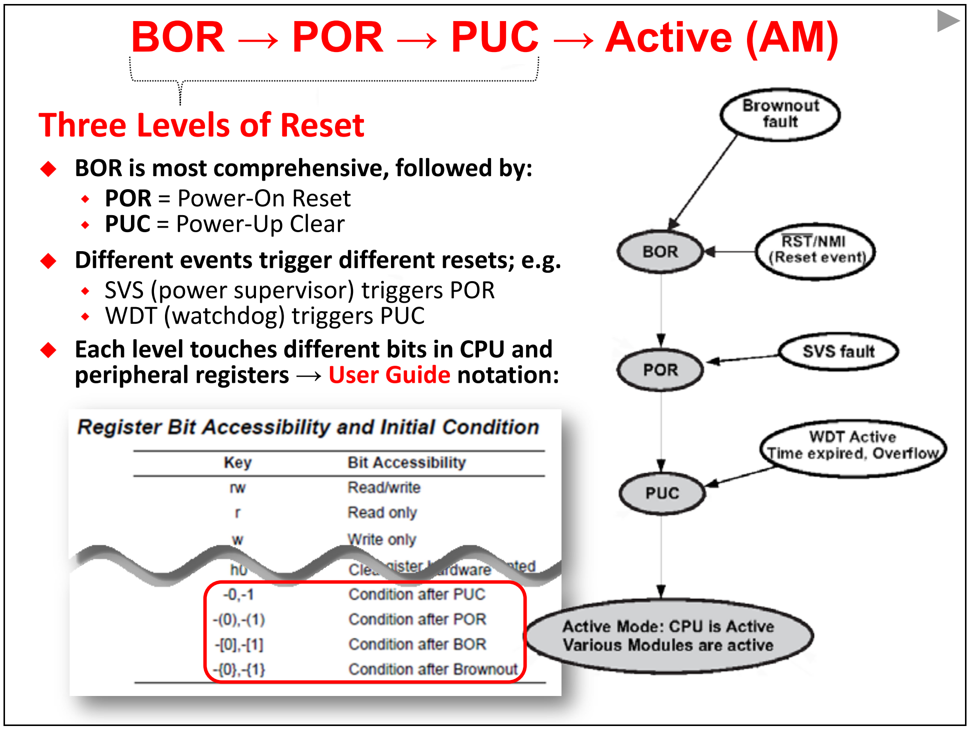
The MSP430 starts out in the Brown-Out Reset (BOR) mode. A Brownout Fault (i.e. not enough power) is the most common event that brings the CPU to this state.



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

BOR → POR → PUC → Active (AM)

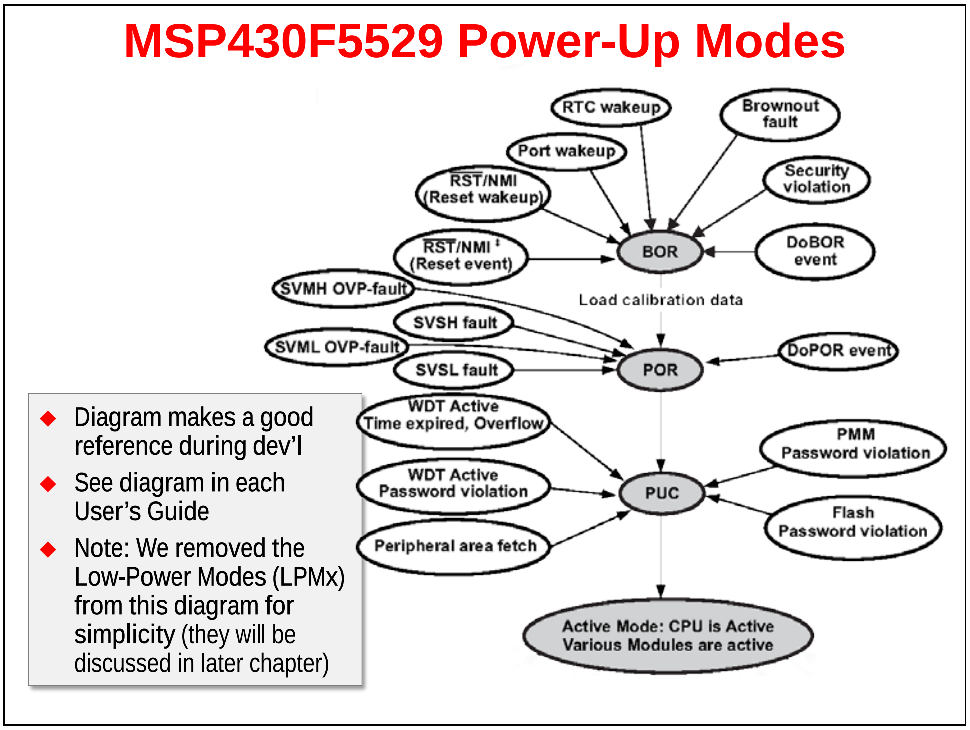
As shown below, BOR is the first of three reset states.



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

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

Here's the full diagram showing the Reset and Active modes for the 'F5529[1]. This shows all the various events that direct the MSP430 CPU into its different *Reset* states. You can find a similar diagram for each series of MSP430 processors.

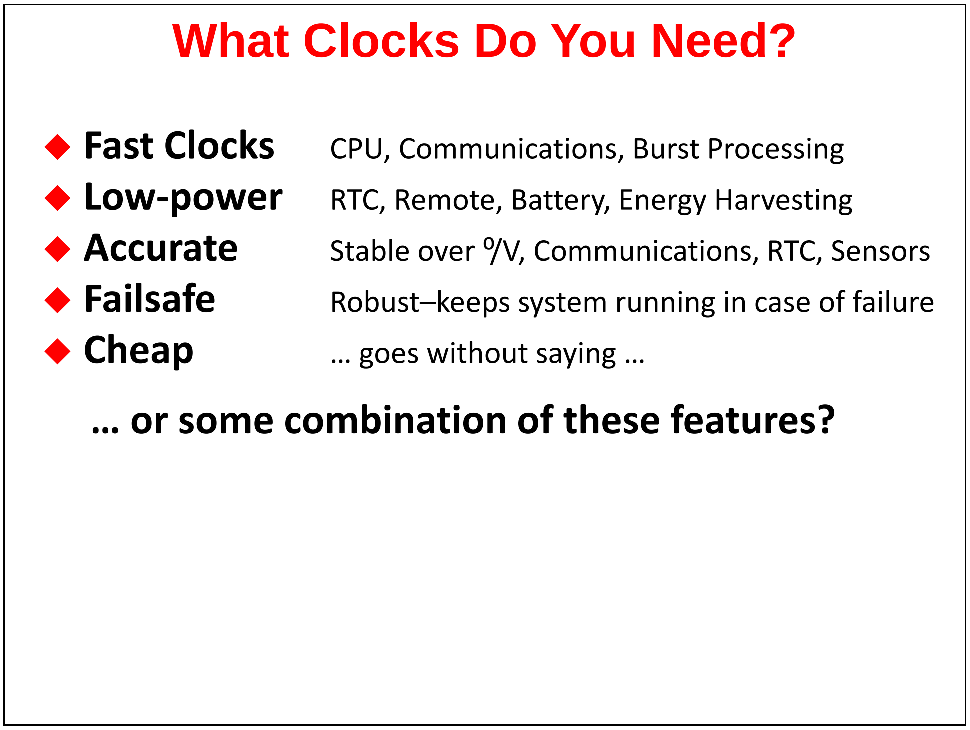


[1] *MSP430x5xx and MSP430x6xx Family User's Guide*, slau208m.pdf, (Texas Instruments, 2013) pg. 63

Clocking

What Clocks Do You Need?

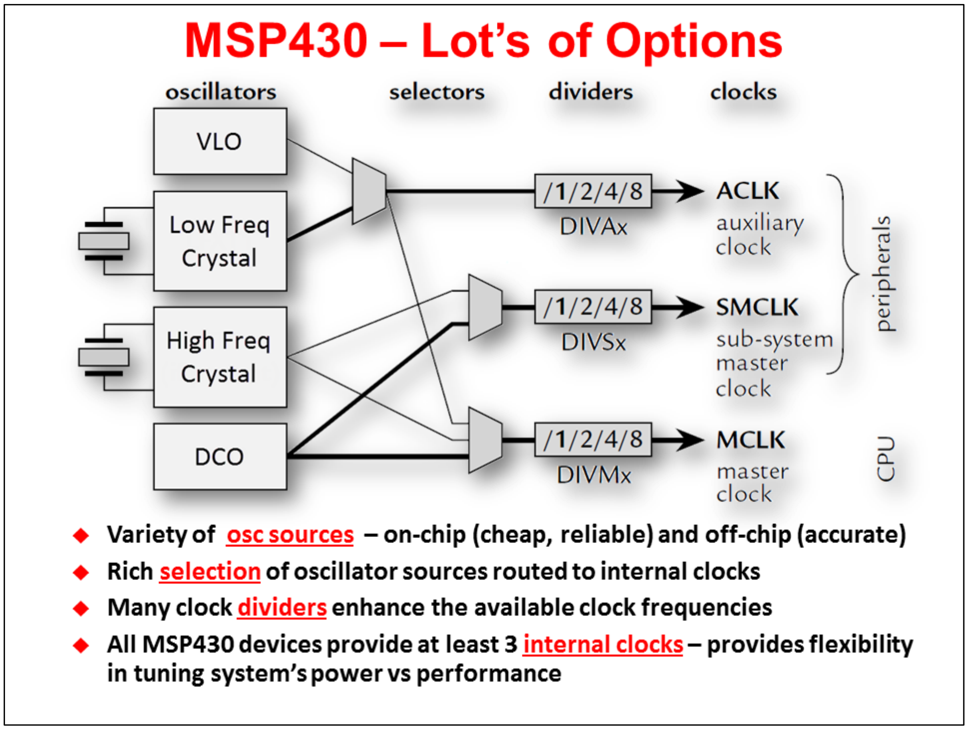
MSP430 provides a wide range of clocking options. Before choosing and configuring the clocks, though, you need to determine which clock features are most important for your system: Fast, low-power, accurate, etc. At times, choosing these various options may force you to make tradeoffs; hence, it's important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



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

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

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



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

MCLK, SMCLK, ACLK

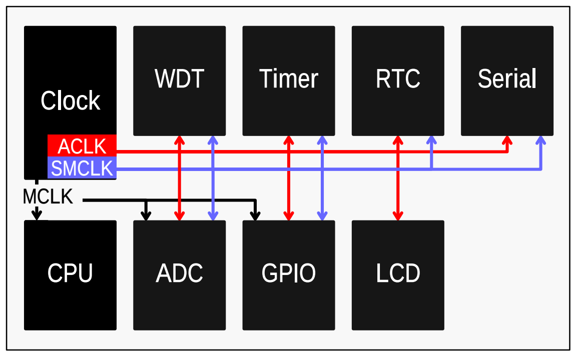
As described in the following graphic, MCLK drives the clock rate of the CPU. It typically runs at a "fast" speed – from 1 MHz up to 16 or 25 MHz (depending upon the upper limit of the given device). MCLK can run slower than this, but it's more common to see the CPU run in the MHz range in order to get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

MSP430 Clock Options

Name	Description	Used-by	Typical Speed
❑ **MCLK**	Master Clock	CPU	Fast
❑ **SMCLK**	Sub-Master Clock	Peripherals	Fast
❑ **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**



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

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

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

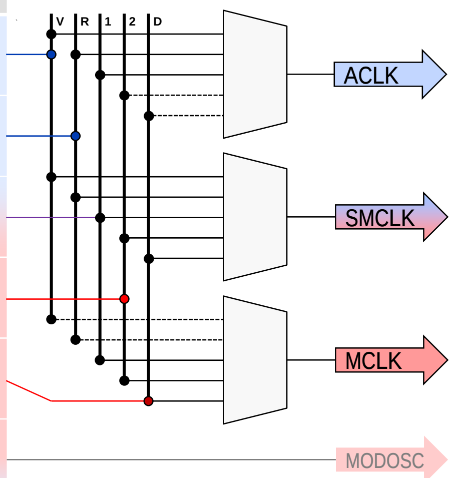
The preceding graphic shows how one might use these various clocks on the MSP430. Please refer to the datasheet, though, since these vary slightly by device. For example, some devices allow all three clocks (MCLK, SMCLK, ACLK) to drive all of the peripherals while others only allow SMCLK and ACLK.

Oscillators (Clock Sources)

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

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*

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

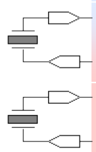
Typical Clock Sources

	Frequency	'G2553 Value-line	'F5529 USB	'FR4133 FRAM	'FR5969 FRAM
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 (MODCLK)	• 5 MHz • 5 MHz / 128	☑	☑	☑	☑

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

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

Clock Source Details ('F5529)



	Frequency	Precision	Current / Startup	Comments
VLO	~10 KHz	Very Low (±40%)	60nA	Use as Ultra Low Power tick
REFO	32768 Hz	Med/High (3.5%)	3µA 25µS	Trimmed to 3.5%
XT1	• 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	• 5 MHz • 5 MHz / 128	Med	N/A	Used by FLASH or ADC

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

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

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

- Not all devices provide both clock sources; for example, we saw on the previous page that the 'G2553 only has XT1 (in fact, it's actually called LFXT1 on that device).
- Why would you need two external clocks? For those cases when you need very precise low and high frequency clocks. For example, you might use XT1 to drive a real-time clock (RTC) while the 'F5529 uses XT2 to source a high-speed, high-precision clock to the USB peripheral.
- It should also be noted that you can connect a digital oscillator signal directly to these inputs; that is, you don't have to use a crystal if you've already got the necessary frequency on your board.
- Bottom line, the XT inputs provide the highest possible precision, but are a little less robust since crystals can often be one of the most delicate components in a system.

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

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

Clock Details (by Device Family)

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

MSP430 Clock Modules

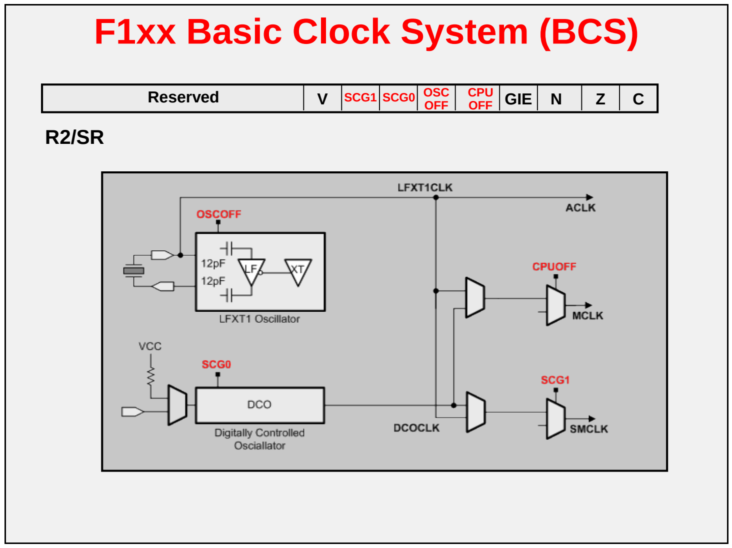
Module	Clock Module Name	MSP430 Device Family
BCS	Basic Clock System	F1xx / F2xx
BCS+	Basic Clock System +	F2xx / G2xx
FLL+	Frequency Locked Loop +	F4xx
UCS	Unified Clock System	F5xx / F6xx
CS	Clock System	FR5xx / FR6xx
CS	Clock System (slightly different than FR5xx/6xx version)	FR2xx / FR4xx
CCS	Compact Clock System	L092

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

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

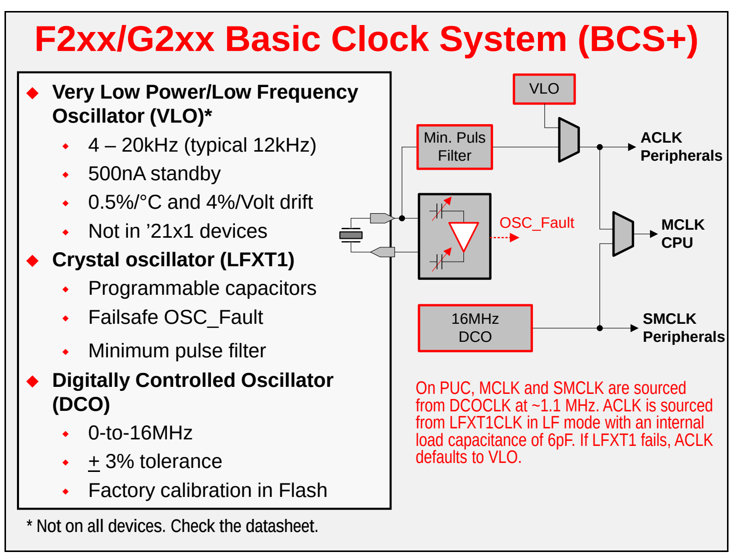
F1xx Clocking (BCS)

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



F2xx/G2xx Clocks (BCS+)

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

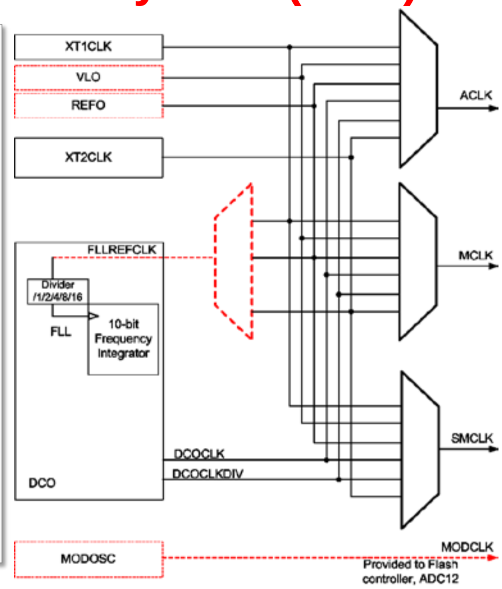


F5xx/F6xx Clocks (UCS)

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

F5xx: Unified Clock System (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/F6xx: Unified Clock System

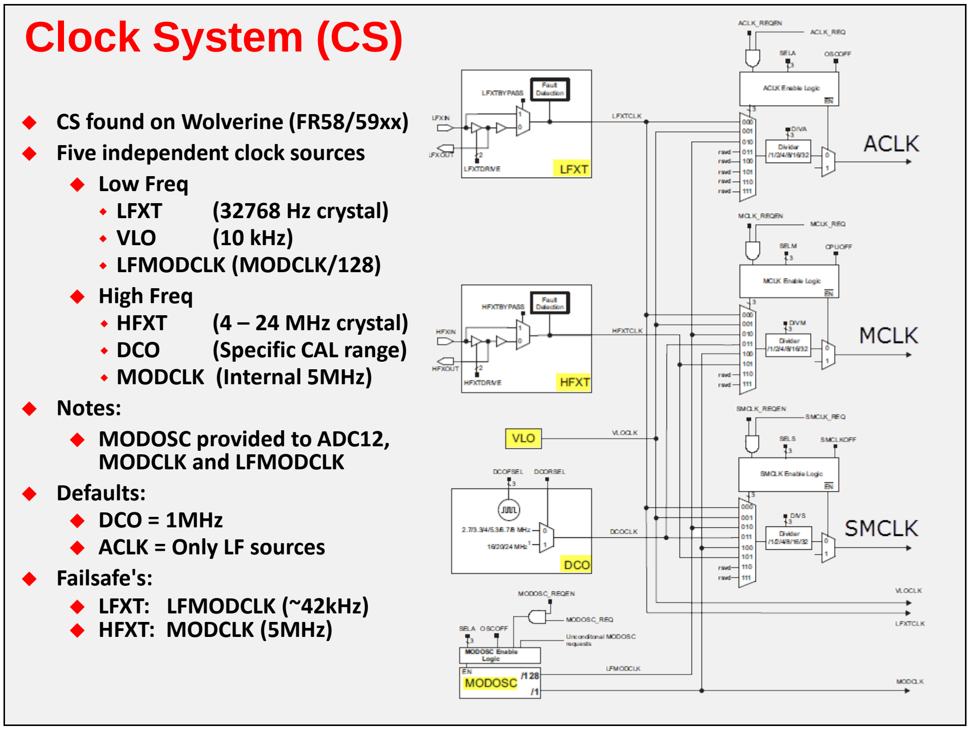
- Orthogonal clock system
  - Any source can drive any clock signal
- 2 Integrated clock sources:
  - REFO: 32kHz, trimmed osc.
  - VLO: 12kHz, ultra-low power
- DCO & FLL provide high frequency accurate timing
- MODOSC provides bullet proof timing for Flash
- Crystal pins muxed with I/O function

Main Features:

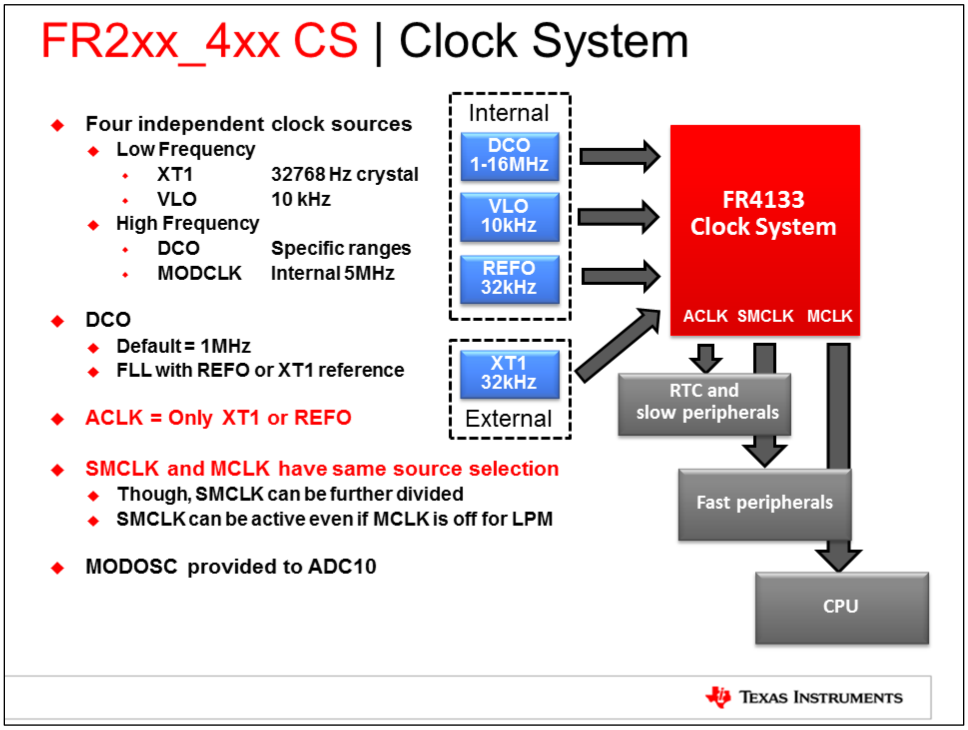
- Any OSC can drive any system clock (MCLK,ACLK,SMCLK)
- Clock divider up to 32 for each system clock
- Control the CLK in Low Power Modes (stopped or running) and react to module CLK requests
- OSC enable logic according requests
- Supporting the FLL as sub-module and providing the control registers
- MODOSC as Clock source for Flash and ADC

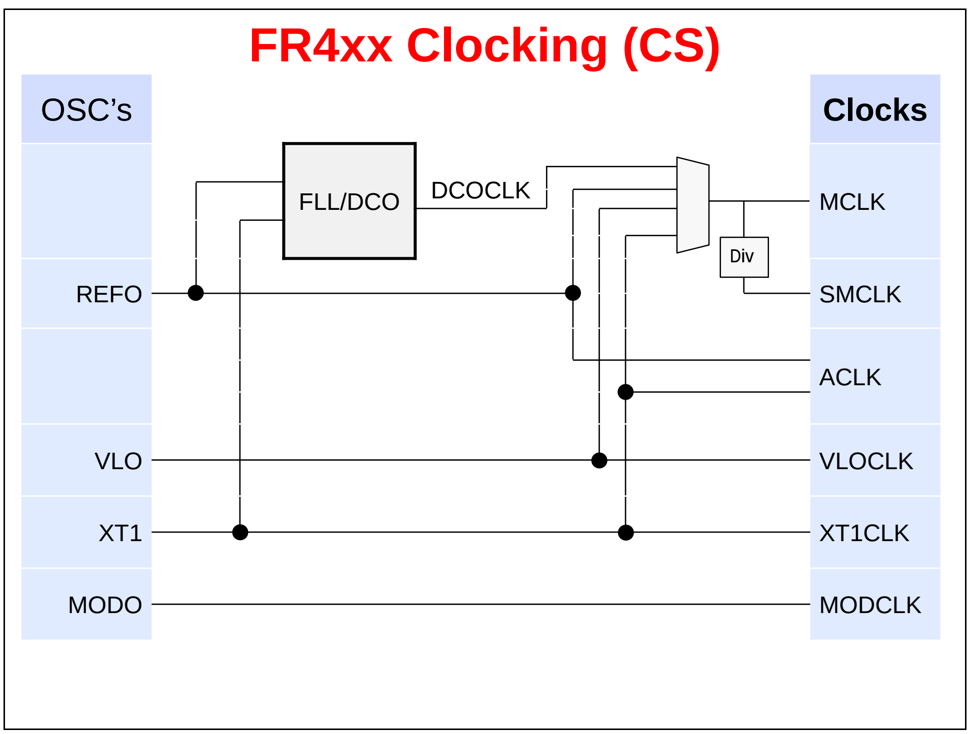
FR58xx/FR59xx - Clock System (CS)

The Clock System (CS) used in the new ‘FR5xx devices provides almost as much flexibility as the UCS peripheral, although – as we’ll see later – it’s easier to configure.



FR2xx/FR4xx - Clock System (CS)





Using MSP430ware to Configure Clocking

As we have done with our other peripherals (e.g. GPIO), we can use MSP430ware's DriverLib to configure the clocking options. For example, in the following diagram the UCS_clockSignalInit() function can be used to configure ACLK to use the REFO clock source.

DriverLib – Selecting Clock Sources

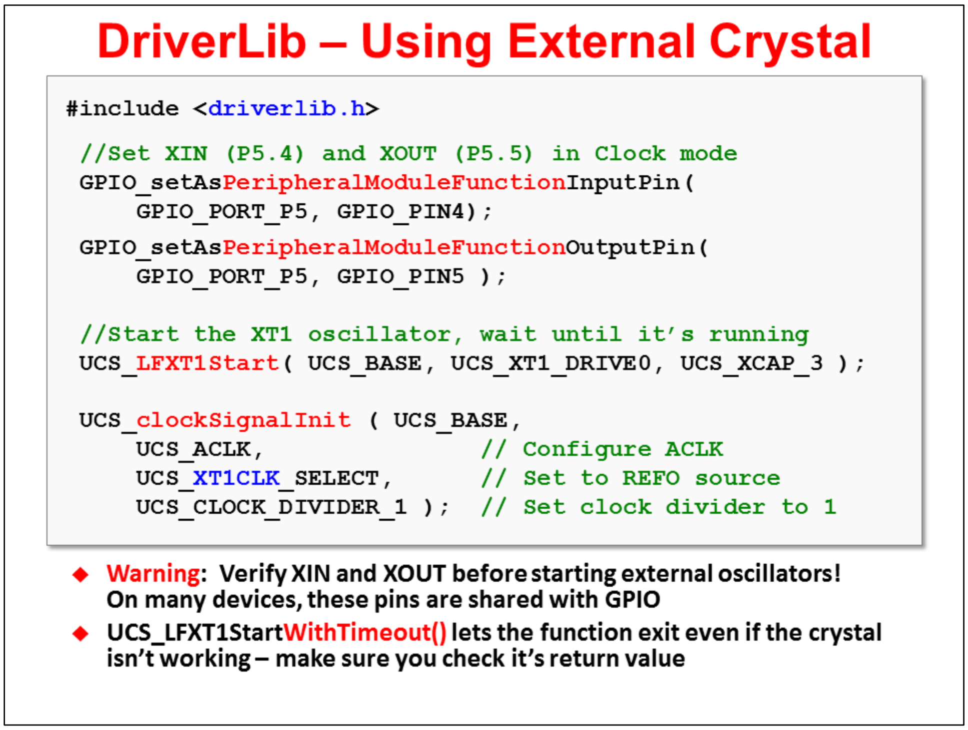
```
#include <driverlib.h>

void myClkInit(void) {

  //Set ACLK = REFO
  UCS_clockSignalInit (
      UCS_BASE,
      UCS_ACLK,                // Configure ACLK
      UCS_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)**

An earlier clock diagram demonstrated the many places where the clock input frequencies can be divided-down; once again, this provides you with a greater possible clock range. In this code example, we just chose to set the clock divider to 1. Conveniently, the DriverLib API provides an enumeration for each possible field value, including all the various clock divider options. *(DriverLib, with these enumerations, makes the code very easy to read.)*

Using an external clock crystal is a bit more involved than using an internal oscillator source. Before you can configure the clock using the same UCS_clockSignal_init() function, you must:

- Setup the XIN/XOUT as clock pins. (On many devices, these pins default to their GPIO modes.)
- The crystal oscillators must be started up before they can be used to source a clock. The clock API provides two start functions: one will not exit until the oscillator has started, while the other one can timeout and return even if the crystal hasn't started running correctly. (If you use the latter, make sure you evaluate its return value.)



As pointed out in the slide, there are two functions that can be used to start each of the crystal oscillator sources: one will continue until the crystal has started (and will run forever); while the other provides a timeout option.

The crystal startup functions provide two arguments for selecting the crystal drive strength and on-chip load capacitance.

- For Low Frequency (LF) crystals, the drive strength option allows you to tune the power needed to drive the crystal; also, you can select an on-chip capacitor that meets your crystals requirements. (Additional external capacitors can be added if necessary.)
- For HF crystals, different crystal or resonator ranges are supported by choosing the proper drive settings. In this case, you will need to use external capacitors.

If you choose to use the XT1 (and/or XT2) inputs with an external clock signal on XIN (XT2IN), you need to set them for bypass mode. Conveniently, DriverLib provides clock (UCS or CS) functions for putting the interfaces into bypass mode.

The optional lab exercise for this chapter provides a crystal oscillator example for you to explore.

Additional Clock Features

There are a number of additional clock features that are summarized for our three example devices in the following table

Additional Clock Features

Clock Feature		'G2553 (BCS+)	'F5529 (UCS)	'FR5969 (CS)
Available Clock Sources	MCLK, SMCLK	VLO, LFXT1, XT2, DCO	VLO, REFO, XT1, XT2, DCOCLK, DCOCLKDIV	VLO, LFXT, LFMODCLK, HFXT, MODCLK, DCOCLK
	ACLK	VLO, LFXT1	VLO, REFO, XT1, XT2, DCOCLK, DCOCLKDIV	VLO, LFXT, LFMODCLK
Clock Defaults (at PUC Reset)	MCLK, SMCLK	DCO (1.1MHz)	DCOCLKDIV (1.048 MHz)	DCO (1MHz)
	ACLK	LFXT1	XT1CLK (32KHz)	LFXT
External Clk Failsafe		ACLK = VLO S/MCLK = DCO	LF XT1 = REFO HF XT1/XT2 = DCO	LFXT= LFMODCLK (38KHz) HFXT=MODCLK (4.8MHz)
DCO Calibration		Factory Constant	FLL (Run-time)	Factory Trimmed
Password Needed (To change clock settings)		No	No	Yes
Clock Request (Periph can force clk on)		WDT+ only	Yes	Yes

There's quite a bit of information on this table. We'll summarize the features row-by-row.

Available Clock Sources: The various clock oscillator sources were described earlier in this chapter. This table shows which clock sources can be used for MCLK, SMCLK, and ACLK. You might notice that, as we described earlier, the UCS peripheral (found on the 'F5529) allows any source to be used with any of the three clocks.

Clock Defaults: What happens if you do not configure the clock peripheral? As you might expect, at (PUC) reset the three internal clocks default to a specific clock source and rate. These are shown in the table.

External Clock Failsafe: What happens if the external crystal breaks or falls off your board? The MSP430 clocks will default to an internal clock. While this may not be the rate/precision you were expecting to run at, it's better than having the system fail outright. There are clock fault events that indicate if the external clock is not working correctly. (Note: it is expected that the clock will be in a 'fault' state while the crystal is initializing.)

DCO Calibration: As we mentioned earlier – and will discuss in more detail later – different generations of the MSP430 use different methods for calibrating the DCO. The first generation forced you to do this in software; later generations use hardware or pre-calibrated constants.

Password: The latest generation of the MSP430 devices requires a password to modify the clock configuration. The purpose of this is to prevent a software error from accidentally changing the settings.

Clock Request: Some devices, such as the ‘F5529, have a “clock request” signal running from their peripherals to the UCS module – these signals *request* that their clock source must remain on. In other words, when this feature is enabled, it prevents you from accidentally turning off a clock that is in use by a peripheral.

For example:

> Let’s say that you wanted to put the CPU to sleep using Low-Power Mode 3 (LPM3) and wait in that mode until the UART received a byte and created an interrupt.
>
> A problem would occur, though, if your UART was being clocked by SMCLK since LPM3 turns off SMCLK. In other words, what happens if the peripheral you were using to wake the processor up just happened to be using that clock, you would never wake up.

The *Clock Request* feature allows a peripheral, such as the UART, to prevent its source clock from being turned off. The CPU will still go into LPM3 mode, but in this case SMCLK would remain on.

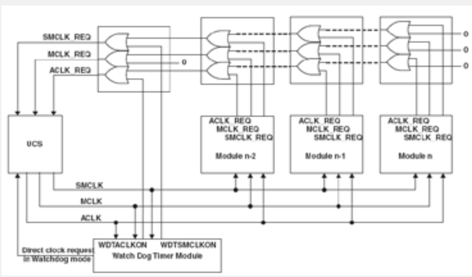
The caveat of *Clock Request* is that it affects power dissipation. By preventing a clock from turning off, your processor will consume more power.

On the ‘G2553, only the clock being used by the Watchdog (WDT+) cannot be turned off, even if the power mode (LPM) normally turns off that specific clock.

Our other two example devices (‘F5529, ‘FR5969) use a bit more advanced scheme. That is, additional peripherals can ‘request’ a clock to remain on, even if a specific LPM normally disables that clock.

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.**



Note: While this feature is a handy failsafe, it can also prevent your system from reaching its lowest power state.

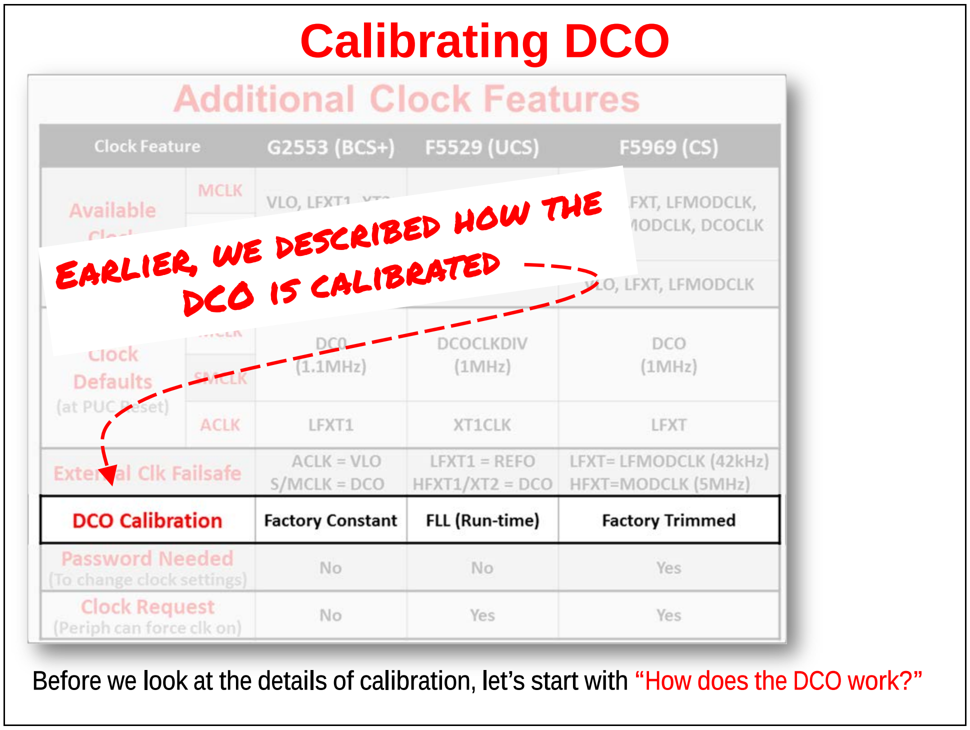
Additional Clock Notes/Warnings

Here's an assortment of notes and warnings about the clocks.

Other Clock Notes/Warnings

- Devices with shared IO's for GPIO and XIN/XOUT:
  - Configure the XIN/XOUT ports correct, if you forget this the Fault will be still available.
  - If using a loop or interrupt for clearing the fault flag you will loop forever
- After clearing the fault flag in the Clock system successfully you need to clear the OFIFG flag inside the SFR as well.
  - If you don't do this you run always with the failsafe clock. Two stage Fault logic is new for 5xx series
- If LFXT is disabled when entering into a low-power mode:
  - It is not fully enabled and stable upon exit from the low-power mode, because its enable time is much longer than the wakeup time.
  - If the application needs to keep LFXT enabled during a low-power mode, the LFXTOFF bit can be cleared prior to entering the low-power mode which causes LFXT to remain enabled.
  - Similarly, the HFXTOFF bit can be cleared prior to entering the low-power mode. This causes HFXT to remain enabled.

DCO Setup and Calibration

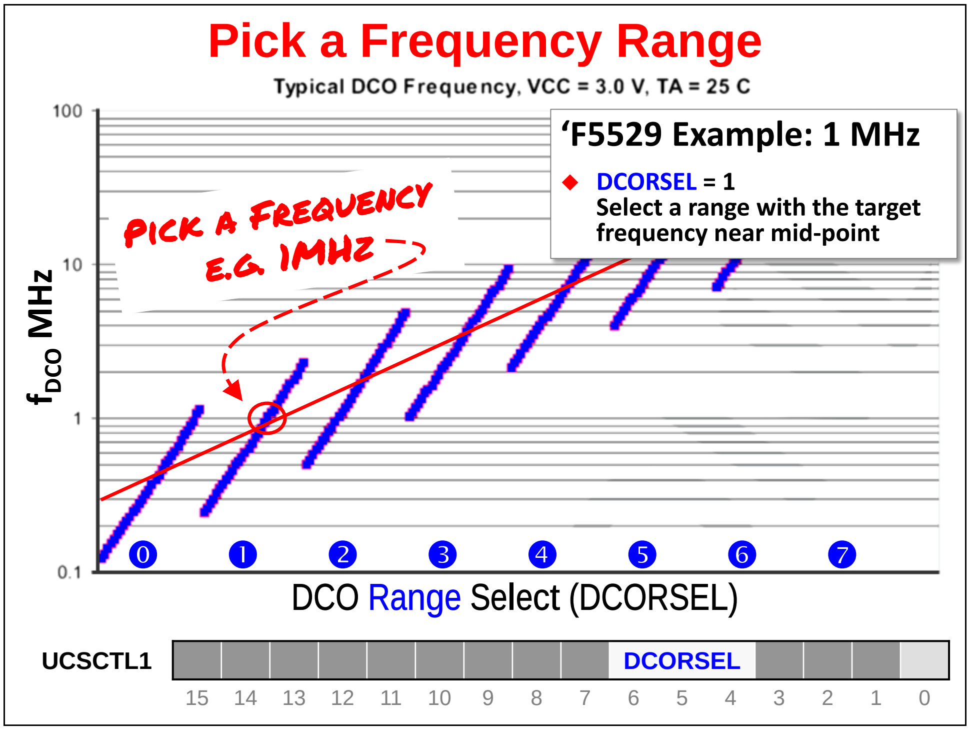


As you can see from our earlier table, the DCO (digitally controlled oscillator) can be calibrated in a variety of different ways, depending upon which generation MSP430 processor you're using. Before discussing these various calibration options, let's first look at how the DCO works.

How the DCO Works

The DCO is configured using three register fields. On most devices they're named: DCORSEL, DCO, and MOD. In the process of discovering how the DCO works, we'll see how each of these fields affects the DCO's output.

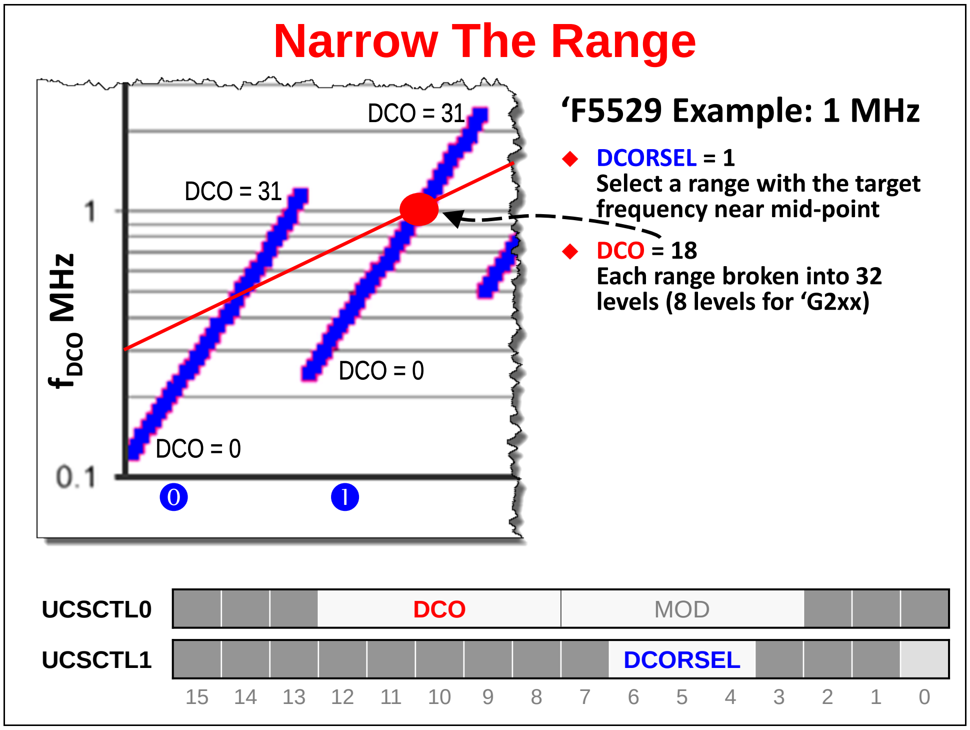
The DCO can operate in a number of different frequency ranges. On the 'F5529, you can select from one of eight different frequency ranges. You might notice that these ranges overlap each other quite a bit. The goal would be to pick a range where your desired frequency sits near the middle. (This is not required, but provides the greatest flexibility - as we'll see in a minute.)



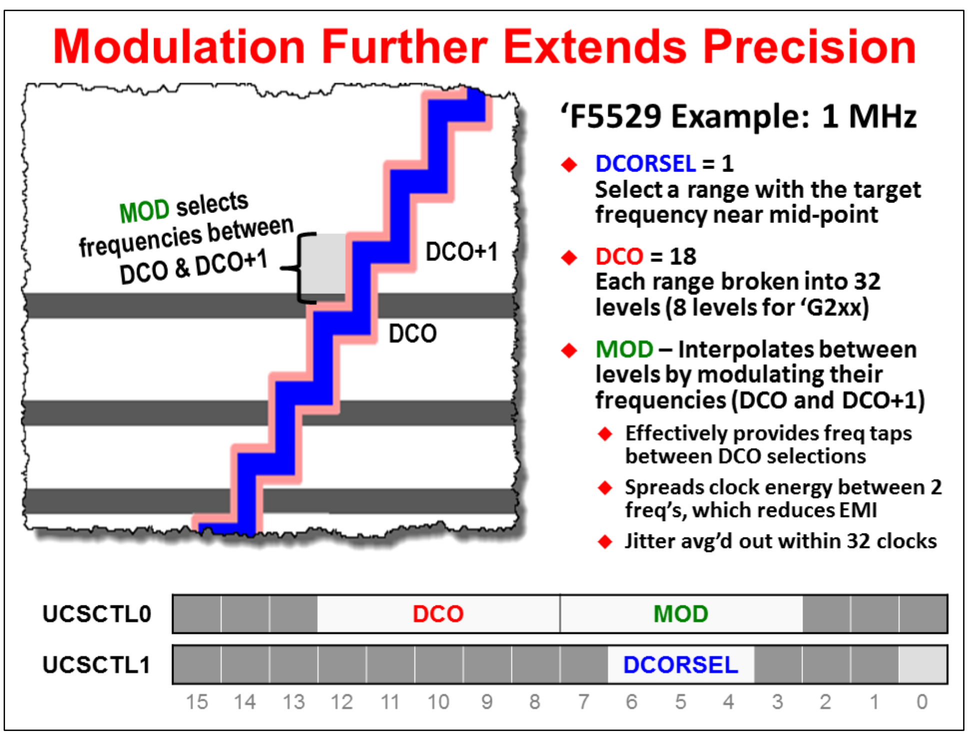
In the diagram above, if we wanted to run at 1 MHz, range "one" happens to be a good choice. Any of the first three would work, but range "1" puts our desired frequency close to the middle of the range.

Notice that the DCORSEL (DCO Range SELect) register field provides a means of selecting which DCO range you want to use.

While the DCORSEL allows you to select a range of frequencies, it's the DCO field that allows us to indicate which frequency we desire within that range. On the 'F5529 the DCO field is 5-bits long, which means we're provided 32 different frequency levels in our chosen range.

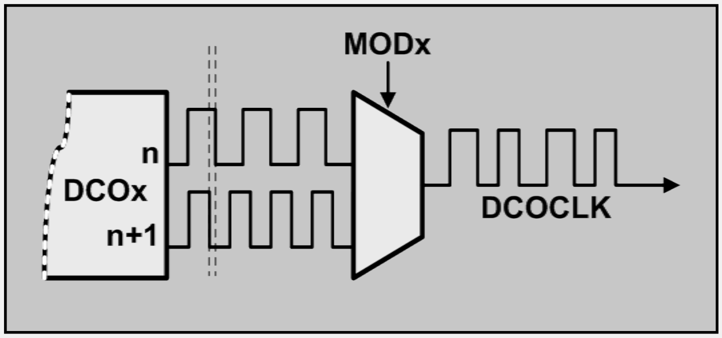


What happens when the frequency you're interested in falls between two levels specified by the DCO field? In other words, what happens if the granularity of the DCO field is not enough to specify our frequency of interest? (I.E. our frequency falls between a value of DCO and DCO+1.)



This is where the final field, called MOD, comes into play. MOD lets you tell the MSP430 clock to modulate between two frequency levels: DCO and DCO+1. By mixing these two frequencies you can obtain a very close approximation to your chosen clock frequency.

DCO Modulation

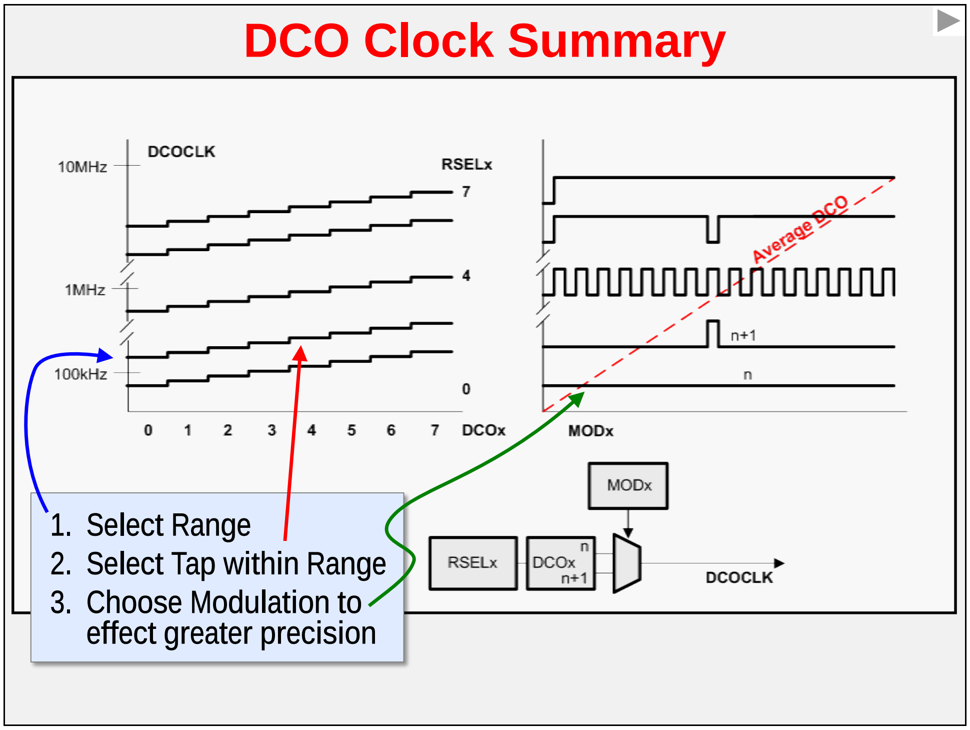


- The modulator mixes two frequencies to produce the DCO clock
- This spreads the clock energy and reduces electromagnetic interference (EMI)
- Due to small jitter, DCO cannot be used to lock a PLL

Naturally, you will probably configure DCO and MOD (and DCORSEL) during system initialization (probably early in your main() function). If the temperature or input voltage varies over time, though, you will likely want to tweak (i.e. tune) DCO and MOD to compensate for your systems changing environment. On older MSP430 devices, these tweaks had to be done in software; on later devices, hardware was added to automate this task for you. We'll look at these tuning options in the next section of the chapter.

DCO Summary

Here's a summary of the DCO features we just discussed – the graphic is just drawn a little differently. In essence, you must: (1) pick a range; (2) select a level within the range; and (3) pick a modulation scheme that allows you to interpolate between adjacent ranges, as needed.



Factory Callibration (FR5xx, G2xx)

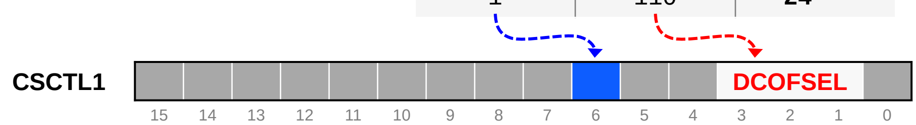
The Value-Line (‘G2xx) and FRAM (‘FR5xx) devices use static, pre-calibrated settings, chosen during device testing, to allow your DCO to meet the frequencies and tolerances specified in the device datasheet.

‘FR5xx Devices

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_DCORSEL_1, CS_DCOFSEL_3);
```

Configuration of the ‘FR5xx devices is the easiest of all the MSP430 devices. Looking at the table in the datasheet (which has been replicated above), you just need to choose the value of the DCORSEL and DCOFSEL fields to match the frequency you want to run at. The silicon is trimmed at the factory so that the device meets the accuracy specified in the datasheet.

'G2xx Devices

The 'G2xxx Value-Line devices take a slightly different approach. Rather than trimming the silicon, as is done with the 'FR5xx devices, the factory stores calibration values into each device's Flash memory (INFOA section) during device test.

'G2xxx DCO – Calibration Constants

DCO Calibration Data (provided from factory in flash info memory segment A)			
DCO Frequency	**Calibration Register**	**Size**	**Address**
1 MHz	CALBC1_1MHz	byte	010FFh
	CALDCO_1MHz	byte	010FEh
8 MHz	CALBC1_8MHz	byte	010FDh
	CALDCO_8MHz	byte	010FCh
12 MHz	CALBC1_12MHz	byte	010FBh
	CALDCO_12MHz	byte	010FAh
16 MHz	CALBC1_16MHz	byte	010F9h
	CALDCO_16MHz	byte	010F8h

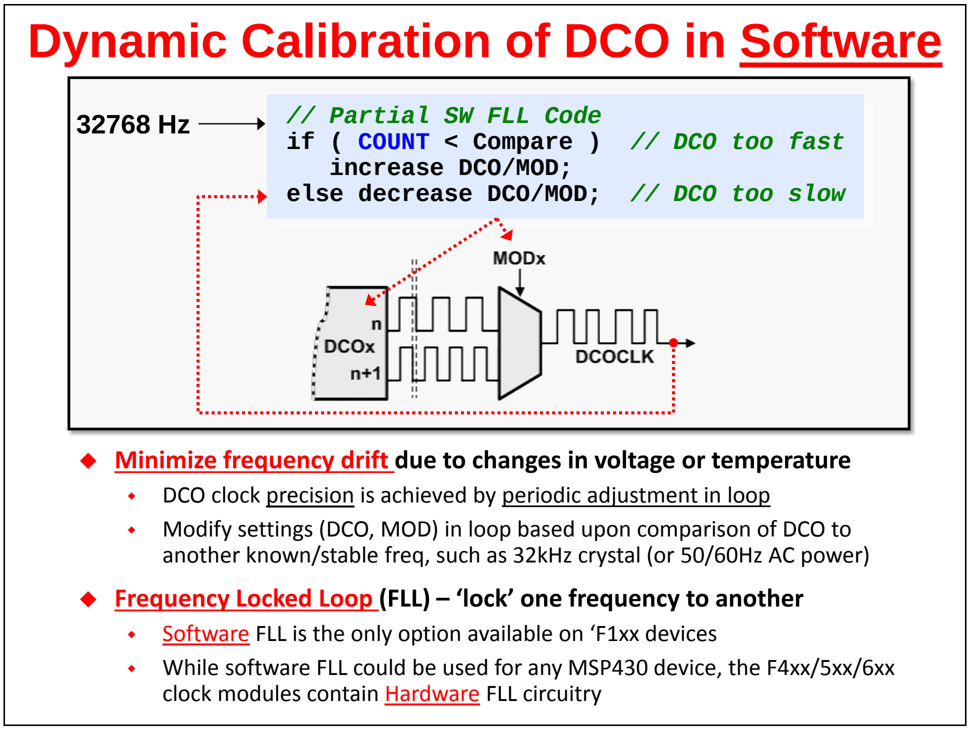
- Most G2xx devices provide pre-calibrated clock settings – applying these sets the Range, DCO, and MCO values
- Clock (and ADC) calibration values are calculated at the factory and stored into Flash memory (INFOA)
- G2xx1 provide 1MHz calibration; G2xx2/3 provides all 4 frequencies

```
 // Setting the DCO to 1MHz
if (CALBC1_1MHZ ==0xFF || CALDCO_1MHZ == 0xFF)
   while(1);              // Erased calibration data? Trap!
BCSCTL1 = CALBC1_1MHZ;    // Set range
DCOCTL  = CALDCO_1MHZ;    // Set DCO step + modulation
```

Basically, the device tester measures the silicon to determine what value of DCO and MOD is required to run the DCO at a set of pre-determined frequencies. These calibration values are stored into INFOA memory by the tester. You can then copy the appropriate calibration constant from Flash into your DCO control register to run the clock at a specified frequency.

Runtime Calibration (F4xx, F5xx, F6xx)

The MSP430F5xx series (along with the ‘F4xx and ‘F6xx) of processors can perform dynamic calibration of the Digitally Controlled Oscillator (DCO) using the Frequency-Locked Loop (FLL) hardware built into the Unified Clock System (UCS).

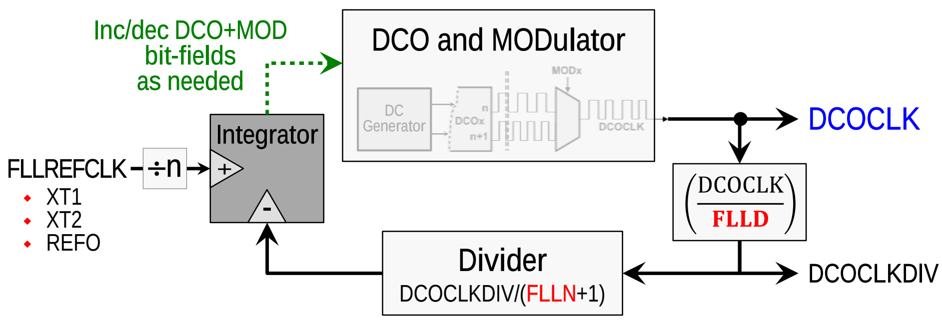


In earlier MSP430 processors, this needed to be handled in software. Using the FLL, the Modulation (MOD) parameter (i.e. field of the DCO control register) is adjusted up or down based upon the count of DCO cycles versus an accurate reference clock (most commonly, a 32KHz watch crystal).

At the top center of the following diagram, you'll see the DCO circuitry. The output of the DCO is labeled DCOLCLK. To provide more flexibility, this signal is divided by a bit-field value called FLLD to make up a second clock frequency called DCOCLKDIV; not only can this clock be used to source MCLK, SMCLK or ACLK, but it is also part of the clock's feedback stabilization.

DCOCKLDIV is divided again by the bit-field FLLN which is then fed into an integrator. Once you have selected a reference input clock to the integrator, the FLL will tweak the MOD bits as needed to make sure the number of DCO clock outputs correlate to the FLL reference clock. Thus even with varying voltage and temperature, as long as the FLL reference remains stable, so will the DCO clock.

'F5xx Hardware FLL



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

where: n = FLLREFDIV

Register	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UCSCTL0				DCO					MOD							
UCSCTL2		FLLD					FLLN									

As long as you know the desired value of DCOCLK and the FLL Reference Clock, it's a simple matter of choosing values for the 3 divider/multiplier fields (n, FLLD, FLLN) to solve the equation.

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

The UCS API found in the MSP430ware DriverLib makes setting up the FLL and DCO easy.

As seen below, you must first configure the FLL reference clock using the UCS_clockSignalInit() function. (In this example, we used REFO as the FLL reference clock.)

Setting 'F5529 DCO with MSP430ware

```
#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_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( MCLK_FREQ_KHZ,
                     MCLK_FLLREF_RATIO);
}
```

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UCSCTL0				DCO					MOD							
UCSCTL1										DCORSEL						
UCSCTL2		FLLD					FLLN									

With the FLL reference clock set, the UCS_initFLLSettle() function configures the FLL and DCO using the two clock frequencies you've chosen (DCOCLK and FLLREFCLK). Additionally, this function adds time needed for the FLL feedback loop to 'settle'. Alternatively, you could use the UCS_initFLL() function if you didn't want the function to add the clock settling time.

Note: The UCS initFLL functions configure both MCLK and SMCLK. A common mistake is to configure SMCLK before calling the FLL init function.

For example, when creating our optional lab exercise, we configured SMCLK to use the XT2 high-frequency crystal before configuring the FLL. We didn't find our mistake until we realized that SMCLK was running at the same speed as MCLK.

One last note, the initFLL functions will set MCLK and SMCLK to DCOCLK if the frequency is greater than 16MHz, otherwise it will use the divided down DCOCLKDIV.

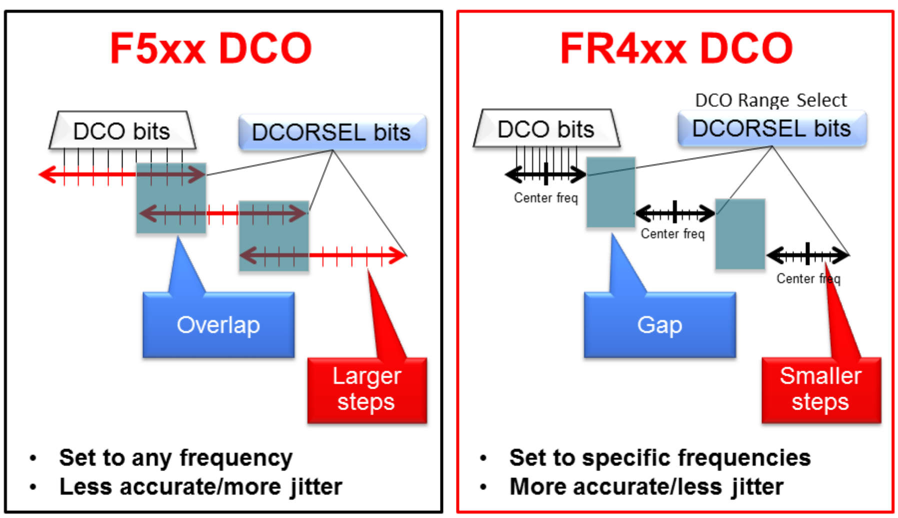
FR2xx/4xx DCO Calibration

FR2xx_4xx Clock System (CS)

- DCO setup is hybrid of FR5xx DCO and F5xx DCO + FLL
- Specific frequency ranges
  - Ranges centered on 1, 2, 4, 8, 12, 16MHz
  - Selected with DCORSEL bits
- Uses FLL with reference frequency to tune within frequency range
- 512 DCO steps within these smaller ranges = smaller steps
  - Allows very accurate DCO + FLL even with just REFO – no crystal (+/-2% over temperature)
  - Even more accurate with crystal (+/-0.5% over temperature)
  - Much less jitter because steps are smaller
- FLL allows compensation for temperature drift

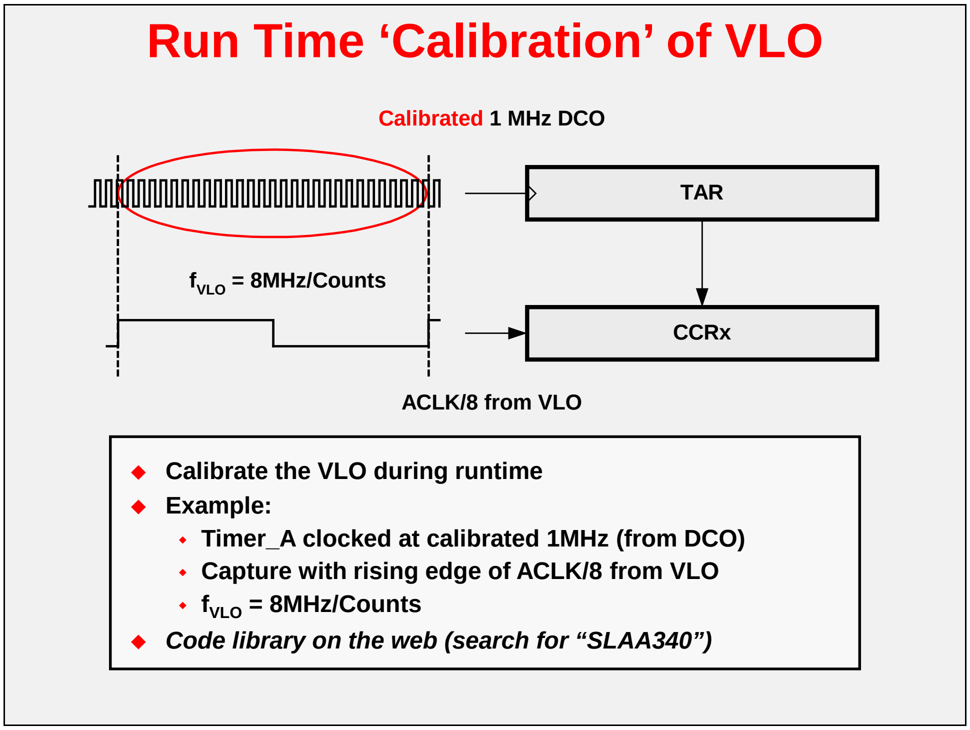
FR2xx_4xx Clock System (CS)

DCO + FLL Comparison



VLO 'Calibration'

The app note and library mentioned on the slide below can be used to calibrate the VLO clock at runtime. While originally not known for its high accuracy, the VLO can be ‘calibrated’ using another clock. The example shown here uses the DCO and TIMER_A to calibrate the VLO.



Other Initialization (WDT, PMM)

When starting up a system, there are a number of elements that must be initialized. Here's a generic summary detailing these items.

Software Initialization

	Initialization Step	Required Action?	Who is Responsible	Where Discussed
1.	Initialize the stack pointer (SP)	Yes	Compiler	N/A
2.	Initialize watchdog timer (usually OFF when debugging)	Yes	User	Chapter 4
3.	Setup Power Manager & Supervisors	No	User	Chapter 4
4.	Configure GPIO pins	No	User	Chapter 3
5.	Reconfigure clocks (if desired)	No	User	Chapter 4 (earlier)
6.	Configure peripheral modules	No	User	Later chapters

The **Stack Pointer** must be initialized but the compiler does this for us, which is why we don't directly discuss this in this workshop.

As discussed many times already in this workshop, since the **Watchdog Timer** defaults to "ON", it must be configured. During development and debugging we usually turn it off. The next section discusses the Watchdog in further detail.

Some of the more feature-rich series of the MSP430 devices contain an on-chip **LDO** along with **Power Manager** and **Supervisor** circuitry. If these features exist on your chosen device, you will probably want to configure them. This is discussed later in this chapter.

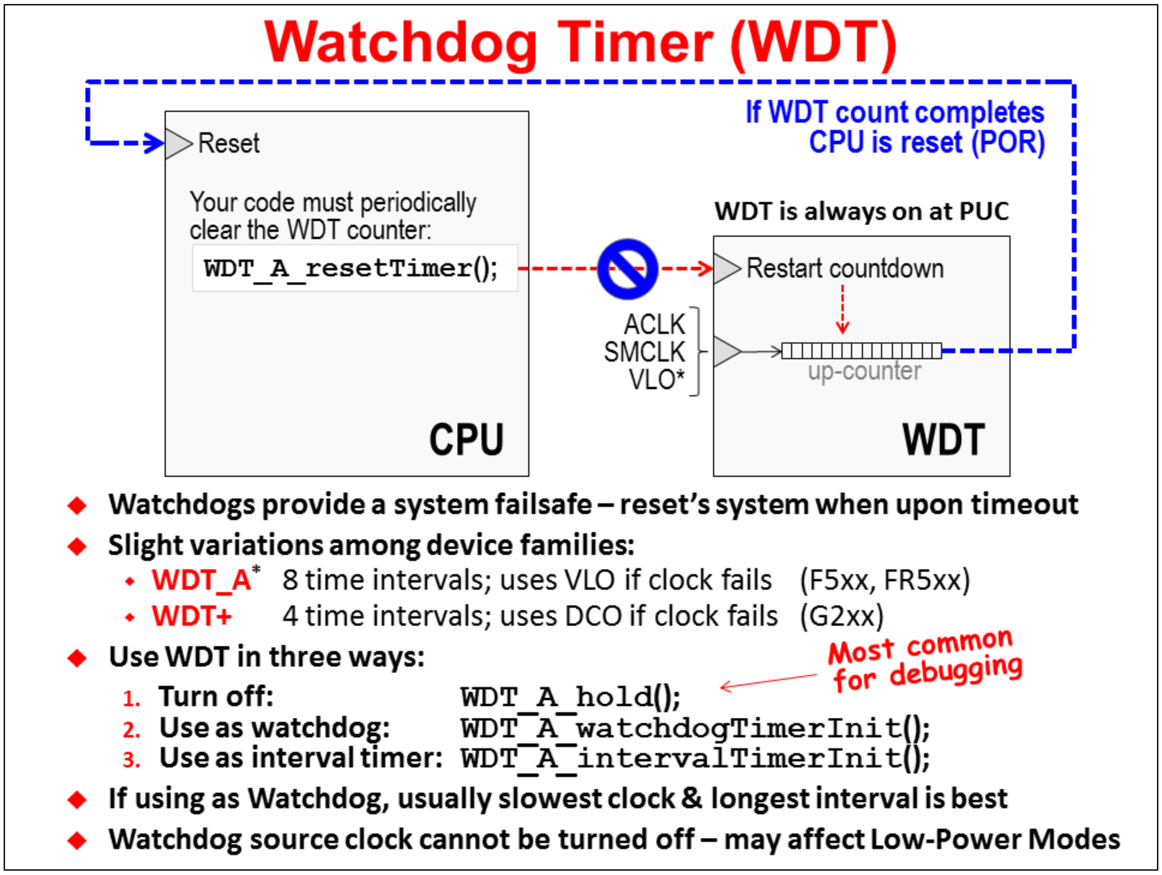
In the last chapter we discussed and used **GPIO pins** (general purpose bit I/O). It highly recommended that you configure all GPIO pins on your device. Obviously, those being used need to be configured, but you should also configure those pins not in use so as to minimize power dissipation.

Earlier in this chapter we discussed the many, varied **clock options** for the MSP430 devices. Unless the default clock options are exactly what you need for your system, these need to be configured.

Finally, you will need to setup and configure the remaining **peripherals** that will be used in your application. We won't try to list them all here – and they vary based upon the selected device – but this is usually handled in main() before starting your while{} loop.

Watchdog Timer

Watchdog Timers provide a system failsafe; if their counter ever rolls over (back to zero) they reset the processor. To prevent your system from continuously resetting itself, you should clear the counter at regular intervals. The goal here is to prevent your system from locking-up due to some unexpected fault.



As mentioned frequently in this class, the MSP430 watchdog timer is "on" by default. You should always disable the watchdog or configure it as needed.

The preceding slide describes three ways to utilize this peripheral:

1. Turn it off – which is useful while developing or debugging your application. You can use the MSP430ware DriverLib "hold" function to accomplish this.
2. Use the Watchdog for its intended function. Again, the provided DriverLib function can be used to perform this initialization.
3. Finally, if you do not need a watchdog for your system, you could re-purpose the peripheral as a generic interval timer. Used this way, for example, you might setup the timer to create periodic interrupts.

Note: As discussed earlier in this chapter, the clock being actively used by the Watchdog timer cannot be disabled. Keep this in mind as you plan out your system and calculate its power requirements.

PMM with LDO, SVM, SVS, and BOR

The power management module (**PMM**) integrates a number of power supply features that may help you minimize external power supply hardware – and cost.

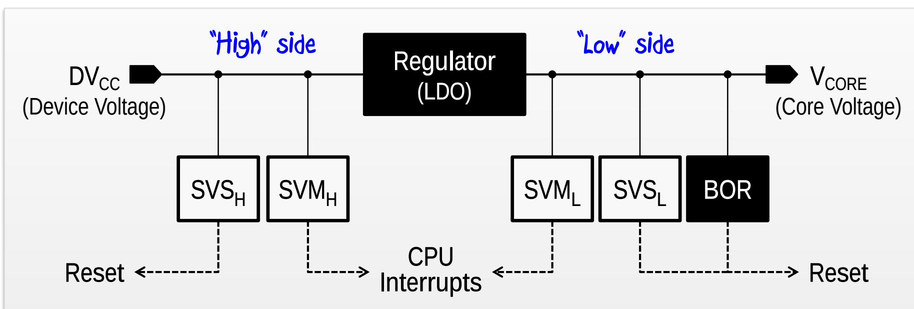
From the diagram below, you can see that we've drawn the **LDO** (low dropout voltage regulator) right in the center of the diagram. This is to drive home the idea that it's a central feature of the PMM. The LDO will provide a regulated, stable voltage to the CPU core from the device voltage applied to the DVcc pins. The device user's guide defines the following nomenclature (as shown below):

- **High Side:** unregulated voltage
- **Low Side:** regulated voltage

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 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

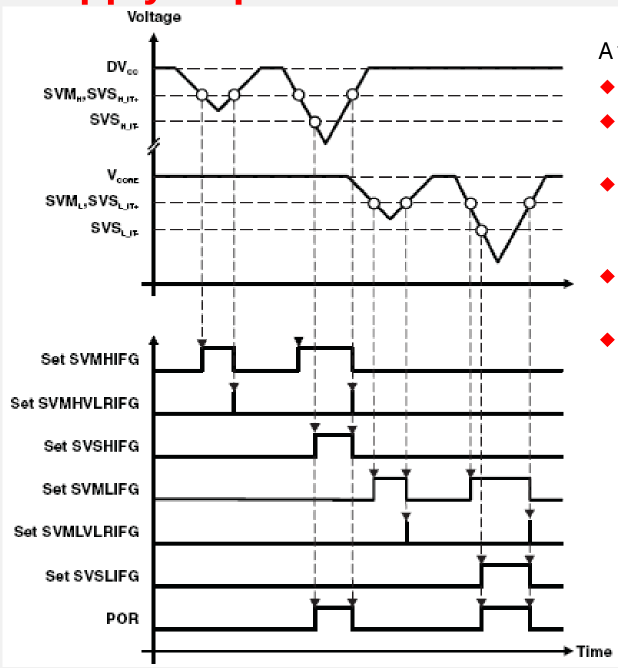
The **SVM** (supply voltage monitor) circuitry is intended to warn you (via interrupts) when the high- or low-side voltages are getting close to their lower limits. You might use this to correct the power supply or prepare for a power error/shutdown. (You can choose not to use this feature if you want to save the small amount of power it consumes.)

The **SVS** (supply voltage supervisor) is another step further in supervision (vs SVM). The SVS actually forces a reset if the high- or low-side voltages fall too low. This helps to prevent possible errors from running the CPU out-of-spec. (You can choose not to use this feature if you want to save the small amount of power it consumes.)

The **BOR** (brown-out reset) circuitry is found on every MSP430 device. You might remember us talking about this hardware at the beginning of the chapter. In a sense, it is redundant to the SVS_L circuitry, although it is always on – and consumes very little power.

The following diagram may help you visualize how the Supply Supervisors work:

Supply Supervisor and Monitor (SVS, SVM)



A few remarks:

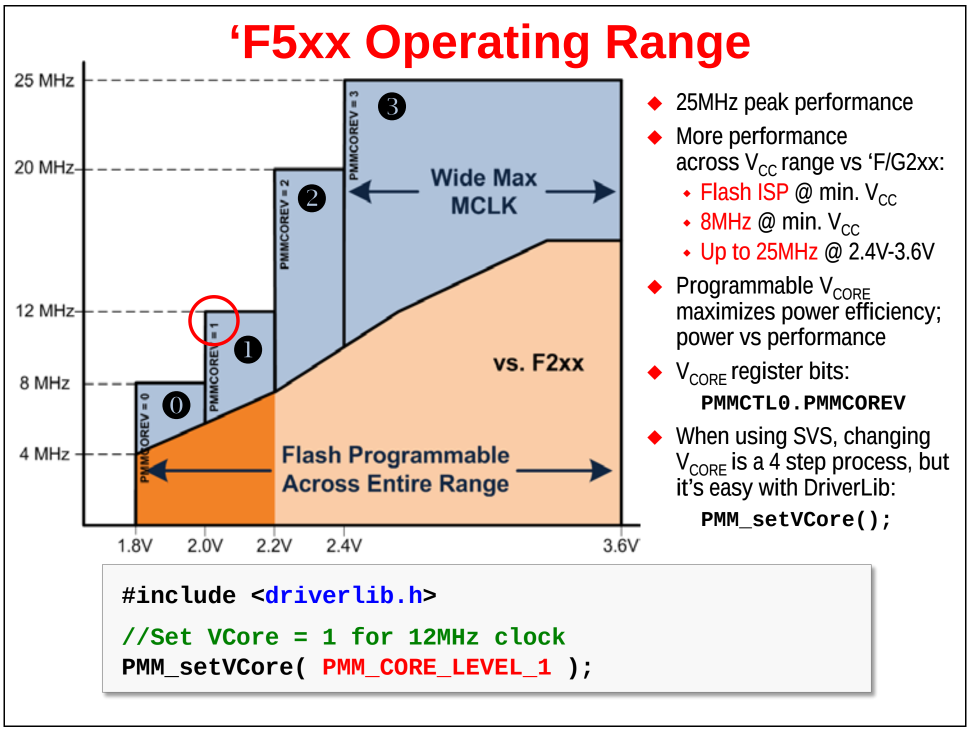
- SVS and SVM can be disabled
- SVM provides "early" warning and generates interrupts
- SVS turns off device – but also sets an interrupt flag (check it after reset)
- High side is the voltage input to the device (prior to PMM's LDO)
- Low side is the core voltage (after LDO)

Operating Voltages

For many of the MSP430 devices, their capabilities can vary based upon the input voltage supply. For example, most of the devices do not support in-system Flash programming when running below 2.2V. Another example is that many devices require higher voltages to run at their faster speeds.

Two examples of this are shown below:

- The ‘F2xx and ‘G2xx devices require 2.2V in order to perform in-system flash programming. Also, their frequency is proportional to the input voltaage
- The F5529 can operate at any one of four voltage ranges. You would need to choose the input voltage range appropriate for the speed you want to run. For example, if you want to run at 10MHz you could run at power mode 1, but 25MHz requires power mode 3. On the other hand, the ‘F5529 can program its flash memory across the entire input voltage range.



The advantage to running with lower power voltage settings is that you, well, save power. The tradeoff is that you give up capability when you run at the lower settings. Then again, you could always change the Vcore setting on-the-fly, as needed by your application at any given time.

One big advantage of the new FRAM devices (e.g. ‘FR5969) is that they can write to their FRAM and at full speed, even when running at their lowest input voltage. This really helps to minimize power while providing you with maximum convenience.

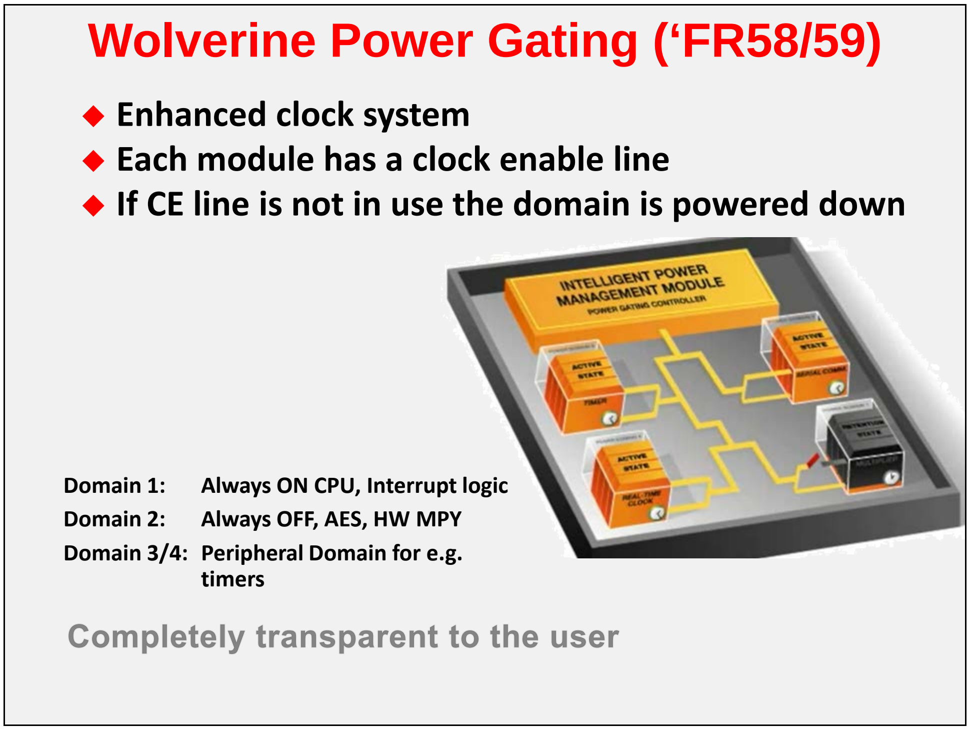
Summary

We have summarized three MSP430 devices in the table below. They demonstrate some of the differences between the various series of MSP430: Value-Line, F5xx, and FR5xx FRAM.

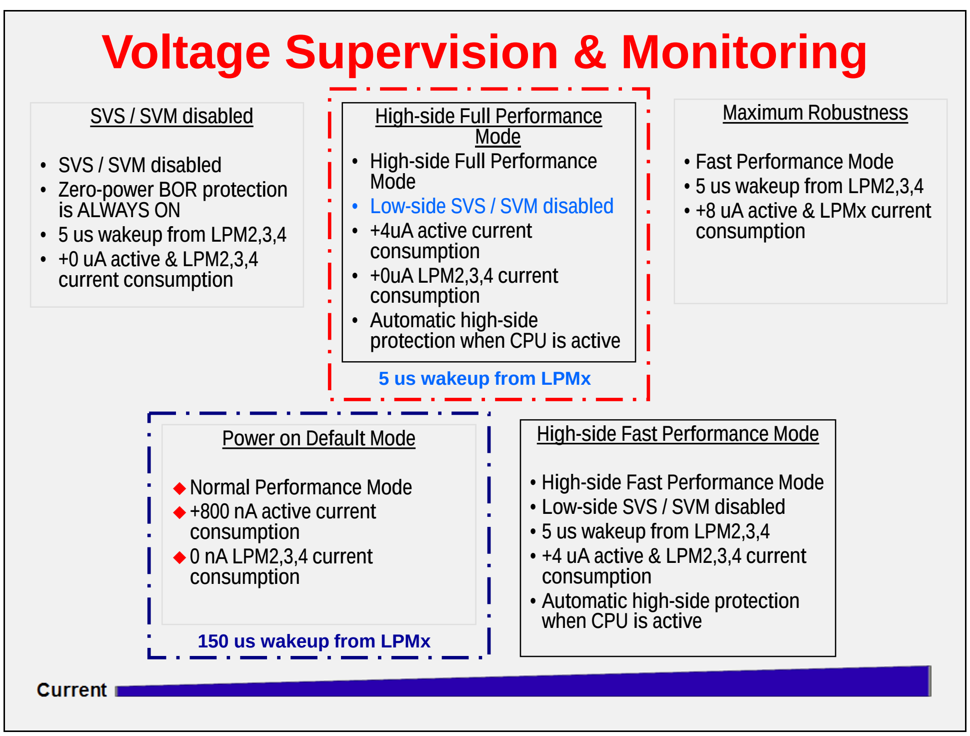
Power Management Summary

	G2553	F5529	FR5969
Input Voltage (DV_{CC})	1.8 - 3.6 Volts	1.8 - 3.6 Volts	1.8 - 3.6 Volts
Internal Regulators (LDO)	None	3 LDO's (LP, HP, USB)	4 LDO's (LP, HP, RTC, FRAM)
# of V_{CORE} Levels (Configuration)	N/A	4 Power Levels (Manual)	Intelligent Power (Automatic)
Speed affected by Input Voltage	Yes 1.8V: up to 6MHz 3.3V: up to 16MHz	Yes 1.8V: up to 8MHz 2.4V: up to 25MHz	No All speeds available over entire range
Flash/FRAM Voltage (In-System Programming)	2.2 V and above	Full Range	Full Range
Brown-Out Reset (BOR)	Yes	Yes	Yes
Power Supervisor (SVS)	F2xx (but not G2xx)	Yes	Yes
Power Monitor (SVM)	No	Yes	Yes
I/O protection (LOCKLPM5)	No	Yes	Yes

The following two slides provide backup information. The first shows the advanced power-gating found in the FRAM devices...



This slide shows a bit more information regarding the voltage supervision/monitoring.



Initialization Summary (template)

To some of you the following template may seem obvious, but we thought it might be handy to provide a template, of sorts, for a main() function in an MSP430 program.

Summary: Initializing MSP430

```
#include <driverlib.h>

void main(void) {
  // Setup/Hold Watchdog Timer (WDT+ or WDT_A)
  initWatchdog();

  // Configure GPIO ports/pins
  initGPIO();

  // Setup Clocking: ACLK, SMCLK, MCLK (BCS+, UCS, or CS)
  initClocks();

  //--------------------------------------------------------
  // Then, configure any other required peripherals and GPIO
  ...

  while(1) {
  ...
  }
}
```

Notice that there are function calls provided for many of the initialization steps discussed in this chapter. Of course, it's up to you to provide the necessary code for each of these functions. The following lab exercises will provide some examples of these functions – which we'll continue to build upon in future chapters.

Lab 4 - Abstract

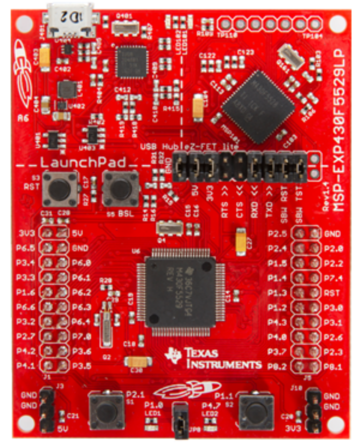
Lab 4 explores a variety of initialization tasks; the largest one being to setup the clocks for the MSP430.

Lab 4 – Clocks & Init

- **Initialize the Lab with a Worksheet:**
  - Clock setup
  - DCO setup
  - Watchdog configuration
- **Lab 4a – Program MSP430 Clocks**
  - Program MCLK, SMCLK, and ACLK
  - Evaluate using 'get' clock rate functions

Extra Credit:

- **Lab 4b – Exploring the Watchdog Timer**
  - What happens if the WDT times-out?
- **Lab 4c – Utilizing Crystals**
  - Configure SMCLK using the external high-speed crystal
  - Configure ACLK using the off-chip external 'watch' crystal



Time:
Worksheet – 15 mins
Lab 4a – 30 mins

This lab also starts off with a worksheet where we will answer a number of questions (and write a little code) that will be used in the upcoming lab procedure.

Lab 4a – Program MSP430 Clocks

We explore the default clock rates for each of MSP430's three internal clocks; then, set them up with a set of specified clock rates.

(Extra) Lab 4b – Blink LED with Different Clocks

If you have time, this lab provides an opportunity to explore the Watchdog Timer.

(Extra) Lab 4C – Utilizing Crystals as Clock Sources

Once again, if you have time, this lab gives us a chance to configure our system to use the external crystal oscillators found on the Launchpad.

Lab Topics

MSP430 Clocks & Initialization 4-40

Lab 4 - Abstract 4-41

Lab 4 Worksheet 4-43
Hints: 4-43
Reset and Operating Modes & Watchdog Timers 4-43
Power Management 4-43
Clocking 4-43

Lab 4a – Program the MSP430 Clocks 4-47
File Management 4-47
Add the Clock Code 4-47
Initialization Code - Three more simple changes 4-52
Debugging the Clocks 4-53
Extra Credit (i.e. Optional Step) – Change the Rate of Blinking 4-56

(Optional) Lab 4b – Exploring the Watchdog Timer 4-57
What happens if WDT is allowed to Run 4-57
A couple of Questions about Watchdogs 4-57
File Management 4-58
Edit the Source File 4-59
Keep it Running 4-61
Extra Credit – Try DriverLib's Watchdog Example (#3) 4-62

(Optional) Lab 4c – Using Crystal Oscillators 4-63
File Management 4-63
Modify GPIO 4-64
Debug 4-65

Chapter 04 Appendix 4-66

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_97_00_47\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_97_00_47\driverlib\driverlib\MSP430F5xx_6xx\ucs.h`

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

`\msp430\MSP430ware_1_97_00_47\driverlib\examples\MSP430F5xx_6xx\ucs\ucs_ex4_XTSourcesDCOInternal.c`

Reset and Operating Modes & Watchdog Timers

1. Name all 3 types of resets:

   ____

2. If the Watchdog (WDT) times out, which reset does it invoke?

   ____

3. Write the DriverLib function that stops (halts) the watchdog timer:

   `________________________( WDT_A_BASE );`

Power Management

F5529

4. **(‘F5529 Launchpad users only)** Write the DriverLib function that sets the core voltage needed to run MCLK at 8MHz.

   `________________________( ________________________ );`

Clocking

5. Why does MSP430 provide 3 different types of internal clocks?

   ____

   ____

   ____

   Name them:

   ____________   ____________   ____________

6. What is the speed of the crystal oscillators on your board?
   (Hint: look in the Hardware section of the Launchpad Users Guide.)

F5529 **FR5969**

‘F5529 and ‘FR5969:

```
#define  LF_CRYSTAL_FREQUENCY_IN_HZ ______________
#define  HF_CRYSTAL_FREQUENCY_IN_HZ ______________
```

FR4133

‘FR4133:

```
#define  XT1_CRYSTAL_FREQUENCY_IN_HZ _____________
```

7. What function specifies these crystal frequencies to the DriverLib?
   (Hint: Look in the MSP430ware DriverLib User’s Guide – “UCS or CS chapter”.)

F5529 **FR5969**

```
______________________( LF_CRYSTAL_FREQUENCY_IN_HZ ,
                        HF_CRYSTAL_FREQUENCY_IN_HZ );
```

FR4133

```
______________________( XT1_CRYSTAL_FREQUENCY_IN_HZ );
```

8. At what frequencies are the clocks running? There’s an API for that…
   Write the code that returns your current clock frequencies:

```
uint32_t myACLK  = 0;
uint32_t mySMCLK = 0;
uint32_t myMCLK  = 0;

myACLK  = ______________________();

mySMCLK = ______________________();

myMCLK  = ______________________();
```

Refer to clocking section of DriverLib User’s Guide

9. We didn’t set up 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? *(You can compare this to your results when running the code.)*

ACLK: ____________ SMCLK: ____________ MCLK: __________

10. Set up ACLK:
    - Use REFO for the F5529 device
    - Use VLO for the FR5969/FR4133 device

```
// Setup ACLK
_____________________________(
    _______ _ACLK,                    // Clock to setup
    ________________________________, // Source clock
    ______ _CLOCK_DIVIDER_1 );
```

11. **(F5529 User's)** Write the code to setup MCLK. It should be running at 8MHz using the DCO+FLL as its oscillator source.

F5529

```
#define MCLK_DESIRED_FREQUENCY_IN_KHZ   _____________________________
#define MCLK_FLLREF_RATIO  ______________________ /(UCS_REFOCLK_FREQUENCY/1024 )

// Set the FLL's clock reference clock to REFO
_____________________________(
    UCS_FLLREF,               // Clock you're configuring
    ________________________, // Clock Source
    UCS_CLOCK_DIVIDER_1 );

// Config the FLL's freq, let it settle, and set MCLK & SMCLK to use DCO+FLL as clk source
_____________________________(
    MCLK_DESIRED_FREQUENCY_IN_KHZ,
    _________________________________);
```

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

FR4133

(FR4133 User's) Write the code to setup MCLK. It should be running at 8MHz using the DCO+FLL as its oscillator source. *(Hint: Look at the chapter discussion slides – it's very similar to 'F5529.)*

```
#define MCLK_DESIRED_FREQUENCY_IN_KHZ   ______________________

#define MCLK_FLLREF_RATIO  _________________ /(UCS_REFOCLK_FREQUENCY/1024 )

// Set the FLL's clock reference clock to REFO
______________________(
    CS_FLLREF,                // Clock you're configuring
    _________________,        // Clock Source
    CS_CLOCK_DIVIDER_1 );

// Config the FLL's freq, let it settle, and set MCLK & SMCLK to use DCO+FLL as clk source
______________________(
    MCLK_DESIRED_FREQUENCY_IN_KHZ,
    __________________________);
```

FR5969

(FR5969 Users) Write the code to setup MCLK. It should be running at 8MHz using the DCO as its oscillator source. *(Hint: Look at the chapter discussion slides.)*

```
// Set DCO to 8MHz
CS_setDCOFreq(
        ____________________,  // Set Frequency range (DCOR)
        ____________________   // Set Frequency (DCOF)
);

// Set MCLK to use DCO clock source
______________________(
    __________________________________________,
    __________________________________________,
    UCS_CLOCK_DIVIDER_1 );
```

☞ **Please verify your answers before moving onto the lab exercise.**

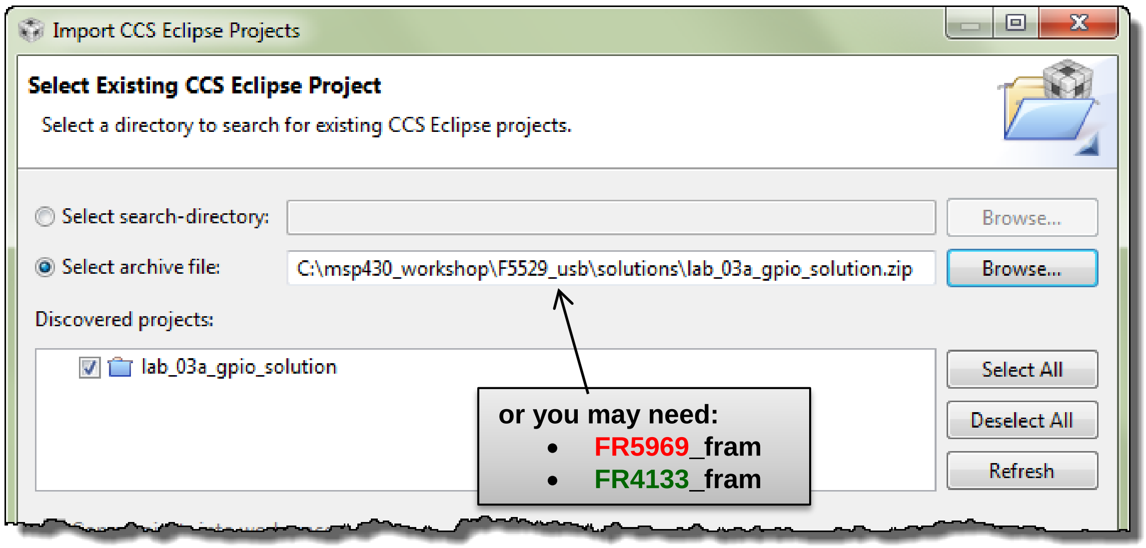
(Find them in the Chapter 4 Appendix)

Lab 4a – Program the MSP430 Clocks

File Management

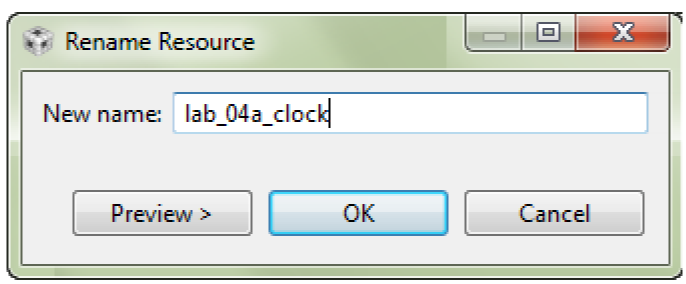
1. Import previous lab_03a_gpio solution.

`Project → Import CCS Projects…`



2. Rename the project to `lab_04a_clock` and click OK.

`Right-Click on Project → Rename`



3. Make sure the project is active, then Build it, to be sure the import was error-free.

Add the Clock Code

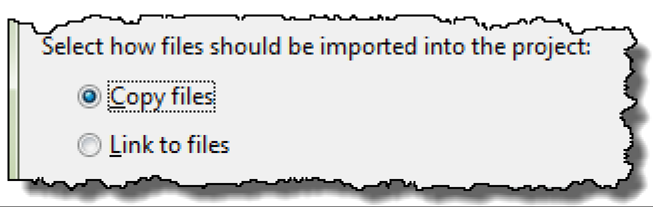
4. Add `myClocks.c` into the project (from the `lab_04a_clock` folder).

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`

Then select:

`Copy files`



F5529

5. **('F5529) 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 #4

Worksheet Question #7

Worksheet Question #8

Worksheet Question #10

Worksheet Question #11

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

//***** Defines ***********************************************************
#define LF_CRYSTAL_FREQUENCY_IN_HZ      _______
#define HF_CRYSTAL_FREQUENCY_IN_HZ      _______

#define MCLK_DESIRED_FREQUENCY_IN_KHZ _______
#define MCLK_FLLREF_RATIO             ______/(UCS_REFOCLK_FREQUENCY/1024)

//***** Global Variables **************************************************
uint32_t myACLK  = 0;
uint32_t mySMCLK = 0;
uint32_t myMCLK  = 0;

//***** Functions *********************************************************
void initClocks(void) {
     // Set core voltage level to handle 8MHz clock rate
     PMM_setVCore( _________________________ );

    // Initialize the XT1 and XT2 crystal frequencies being used
    //  so driverlib knows how fast they are
    _______________________(
            _______________________,
            _______________________ );

    // Verify if the default clock settings are as expected
    myACLK  = UCS_getACLK();
    mySMCLK = UCS_getSMCLK();
    myMCLK  = UCS_getMCLK();

    // Setup ACLK to use REFO as its oscillator source
    UCS_clockSignalInit(
            UCS_ACLK,                       // Clock you're configuring
            ___________________,            // Clock source
            UCS_CLOCK_DIVIDER_1 );          // Divide down clock source

    // Set the FLL's clock reference clock source
    UCS_clockSignalInit(
            UCS_FLLREF,                     // Clock you're configuring
            ___________________,            // Clock source
            UCS_CLOCK_DIVIDER_1             // Divide down clock source
    );

    // Configure the FLL's frequency and set MCLK & SMCLK to use the FLL
    UCS_initFLLSettle(
            MCLK_DESIRED_FREQUENCY_IN_KHZ,  // MCLK frequency
            ___________________             // Ratio between MCLK and
                                            //  FLL's ref clock source
    );

    // Verify that the modified clock settings are as expected
    myACLK  = UCS_getACLK();
    mySMCLK = UCS_getSMCLK();
    myMCLK  = UCS_getMCLK();
}
```

FR4133

('FR4133) Update `myclocks.c` – adding answers from the worksheet

Fill in the __blanks__ with code you wrote on the worksheet.

Worksheet Question #6

Worksheet Question #11

Worksheet Question #7

Worksheet Question #8

Worksheet Question #10

Worksheet Question #11

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

//***** Defines ***********************************************************
#define XT1_CRYSTAL_FREQUENCY_IN_HZ    _______

#define MCLK_DESIRED_FREQUENCY_IN_KHZ _______
#define MCLK_FLLREF_RATIO  _________________/(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  = CS_getACLK();
    mySMCLK = CS_getSMCLK();
    myMCLK  = CS_getMCLK();

    // Setup ACLK to use REFO as its oscillator source
    CS_clockSignalInit(
            CS_ACLK,                        // Clock you're configuring
            ___________________,            // Clock source
            CS_CLOCK_DIVIDER_1 );           // Divide down clock source

    // Set the FLL's clock reference clock source
    CS_clockSignalInit(
            CS_FLLREF,                     // Clock you're configuring
            ___________________,            // Clock source
            CS_CLOCK_DIVIDER_1             // Divide down clock source
    );

    // Configure the FLL's frequency and set MCLK & SMCLK to use the FLL
    CS_initFLLSettle(
            MCLK_DESIRED_FREQUENCY_IN_KHZ,  // MCLK frequency
            _____________________________   // Ratio between MCLK and
                                            //  FLL's ref clock source
    );

    // Verify that the modified clock settings are as expected
    myACLK  = CS_getACLK();
    mySMCLK = CS_getSMCLK();
    myMCLK  = CS_getMCLK();
}
```

FR5969

('FR5969) Update `myclocks.c` – adding answers from the worksheet

Fill in the __blanks__ with code you wrote on the worksheet.

Worksheet Question #6

Worksheet Question #7

Worksheet Question #8

Worksheet Question #10

Worksheet Question #11

Worksheet Question #11

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

//***** Defines ***********************************************************
#define LF_CRYSTAL_FREQUENCY_IN_HZ    _______
#define HF_CRYSTAL_FREQUENCY_IN_HZ    0

//***** Global Variables **************************************************
uint32_t myACLK  = 0;
uint32_t mySMCLK = 0;
uint32_t myMCLK  = 0;

//***** Functions *********************************************************
void initClocks(void) {

    // Initialize the LFXT and HFXT crystal frequencies being used
    //  so driverlib knows how fast they are
    _______________________(
               _______________________,
               _______________________

    // Verify if the default clock settings are as expected
    myACLK  = CS_getACLK();
    mySMCLK = CS_getSMCLK();
    myMCLK  = CS_getMCLK();

    // Setup ACLK to use VLO as its oscillator source
    CS_clockSignalInit(
            CS_ACLK,                             // Clock you're configuring
            ___________________,                 // Clock source
            CS_CLOCK_DIVIDER_1                   // Divide down clock source
    );

    // Set DCO to 8MHz
    CS_setDCOFreq(
            ___________________,                 // Set Frequency range (DCOR)
            CS_DCOFSEL_3                         // Set Frequency (DCOF)
    );

    // Set SMCLK to use the DCO clock
    CS_clockSignalInit(
            CS_SMCLK,                            // Clock you're configuring
            ___________________,                 // Clock source
            CS_CLOCK_DIVIDER_1 );                // Divide down clock source

    // Set MCLK to use the DCO clock
    CS_clockSignalInit(
            CS_MCLK,                             // Clock you're configuring
            ___________________,                 // Clock source
            CS_CLOCK_DIVIDER_1 );                // Divide down clock source

    // Verify that the modified clock settings are as expected
    myACLK  = CS_getACLK();
    mySMCLK = CS_getSMCLK();
    myMCLK  = CS_getMCLK();
}
```

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

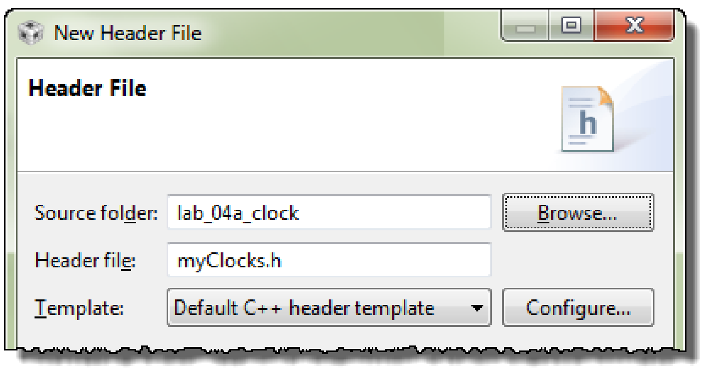
   Hopefully you don't have any typographic or syntax errors, but you should see this error:

   ```
   fatal error #1965: cannot open source file "myClocks.h"
   ```

   Since we placed the init clock function into a separate file, we should use a header file to provide an external interface for that 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.

   `#include "myClocks.h"`

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

Initialization Code - Three more simple changes

11. Reorganize main.c to group initialization code into functions.

We've outlined the 3 areas you will need to adapt to create a little better code organization.

a) Add a prototype for a new function `initGPIO()`.

b) Call `initGPIO()` and `initClocks()` from the main.

c) Create the `initGPIO()` function. Notice that the code for this function already exists; we're just moving it from `main()` to its own function `initGPIO()`.

a) Since the setup code is now organized into functions, prototypes need to be included for them

b) This follows the init code 'template' discussed in class

c) Create GPIO initializatio n function

```
// ----------------------------------------------------------------------
// main.c  (for lab_04a_clock project)
// ----------------------------------------------------------------------

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

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

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

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

    //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 pin P1.0 to output direction and initialize low
    GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );
    GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```

FR5969

FR4133

12. (FRAM devices only) Unlock the pins.

Don't forget to add the *PMM_unlockLPM5()* function to *initGPIO()*, if you haven't already done so.

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

Debugging the Clocks

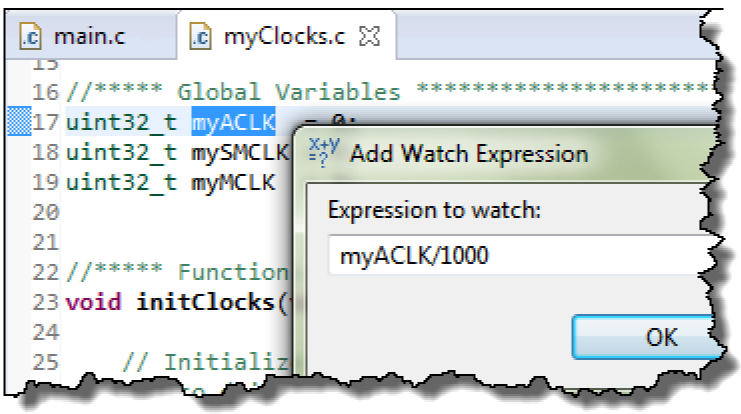
Before running the code, let's set some breakpoints and watch expressions.

14. Open `myClocks.c`.

15. Add a watch expression for *myACLK* (in KHz).

```
Select myACLK in your code → Right-click → Add Watch Expression…
```

Enter `'myACLK/1000'` into the dialog and hit OK. Upon hitting "OK", the *Expressions* window should open up, if it's not already open.



When we run the code, this should give us a value of 32, if ACLK is running at 32KHz.

16. Go ahead and create similar watch expressions for *mySMCLK* and *myMCLK*.

```
mySMCLK/1000
myMCLK/1000
```

17. Export expressions.

CCS lets you export and import expressions. Let's save them so that we can quickly import them later.

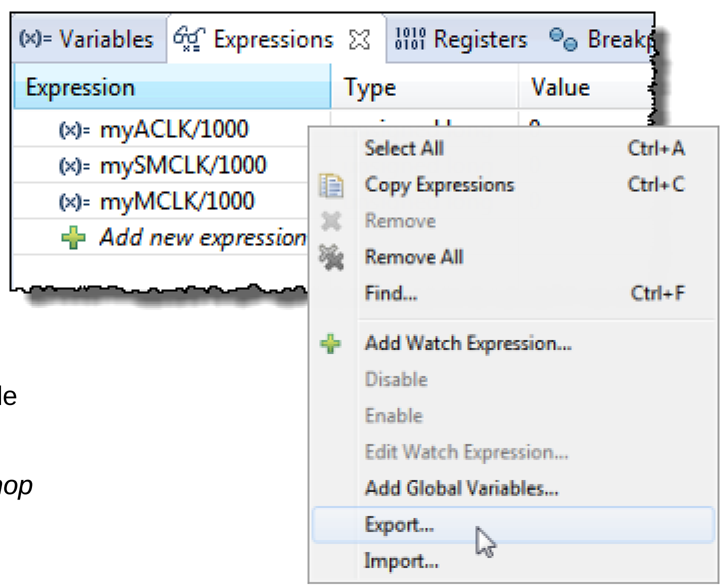
a) Right-click on *Expressions* window

b) Select *Export…*

c) And choose a name & location for the file

– We called it: *myExpressions.txt*

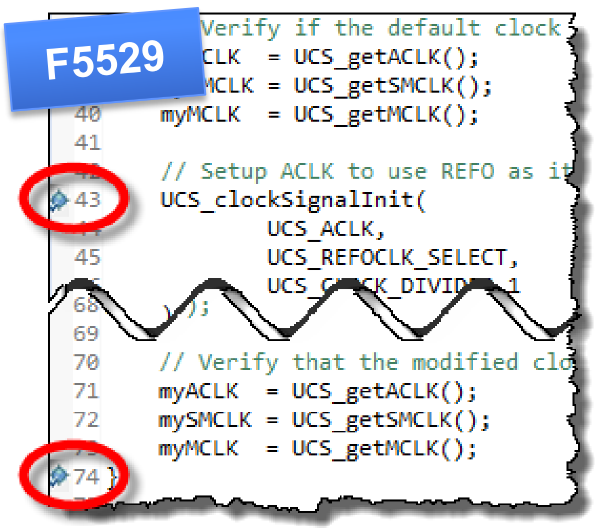
– and placed it at: *C:\msp430_workshop*

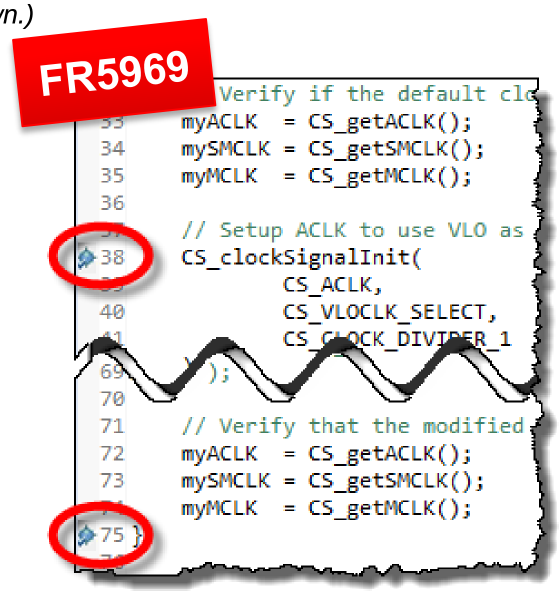


Note: Before you run the code to the first breakpoint, you may see an error in the Expressions window similar to "Error: identifier not found". This happens when the variable in the expression is out-of-scope. For example, this can happen if you defined the variable as a local, but you were currently executing code in another function. Then again, it will also happen if you delete a variable that you had previously added to the Expression watch window.

18. 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: We've shown the F5529 and FR5969 code – we hope you FR4133 users can deduce the correct location based on your own.)*





Note: Some versions of the 'FR5969 debugger for CCSv6 gives an error whenever you 'load a program', 'reset' or 'restart' the processor while multiple breakpoints are set. If you find this happens to you, you can either:

- Clear all breakpoints before performing one of these actions
- Only set one breakpoint … as an alternative, we like to place the cursor where we want to stop and then use ***Control-R*** to "run to the cursor".

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

myACLK/1000: ____

mySMCLK/1000: ____

myMCLK/1000: ____

Are these the values that you expected? ______________________________

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

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

Check on the values again:

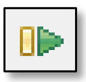
myACLK/1000: ____

mySMCLK/1000: ____

myMCLK/1000: ____

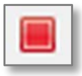
Are these the values we were asked to implement? ______________________

(Look back at Worksheet questions 0-0.)



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

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



22. Halt the processor and terminate the debugger session.

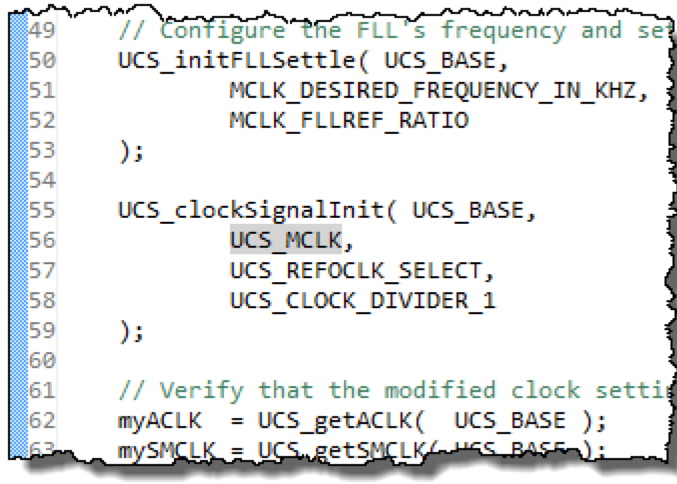
23. Add a function call to initClocks() to force MCLK to use a different oscillator.

- ‘F5529 and ‘FR4133 users, try using REFO.
- ‘FR5969 users, try using VLO since you don’t have the REFO oscillator.

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

The ‘F5529 example is to the right:

As this code demonstrates, it sets up MCLK *(via the UCS_initFLLSettle() function)* then changes it again right away … but that’s OK. No harm done.

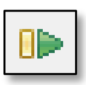


```
    // Configure the FLL's frequency and se
    UCS_initFLLSettle( UCS_BASE,
            MCLK_DESIRED_FREQUENCY_IN_KHZ,
            MCLK_FLLREF_RATIO
    );

    UCS_clockSignalInit( UCS_BASE,
            UCS_MCLK,
            UCS_REFOCLK_SELECT,
            UCS_CLOCK_DIVIDER_1
    );

    // Verify that the modified clock setti
    myACLK  = UCS_getACLK(  UCS_BASE );
    mySMCLK = UCS_getSMCLK( UCS_BASE );
```

F

FYI: DriverLib version 1.70 removed the “_BASE” argument from many of the DriverLib functions.

24. Build your code and launch the debugger.



25. Run the code, stopping at both breakpoints.

Did the value for MCLK change? ______________________________

It should be much slower now that it’s running from REFO or VLO.

26. After the second breakpoint, watch the blinking light.

When the code leaves the initClocks() function and starts executing the while{} loop, it should take a very looooooong time to run the _delay_cycles() functions; our “ONE_SECOND” time was based upon a very fast clock, not one this slow.

To wait for 1 seccond, we set the __delay_cycles() to wait for 8 million cycles (when running at 8MHz). Now that we’re running with a slower clock, how long will it take?

REFOCLK: 8,000,000 cycles / 32,768 cycles/sec = ____244____ sec

VLOCLK: 8,000,000 cycles / 10,000 cycles/sec = ____800____ sec

If you’re patient enough, you should see the light blink…

(You have to be VERY, VERY patient to see the LED blink for VLO clock.)

(Optional) Lab 4b – Exploring the Watchdog Timer

What happens if WDT is allowed to Run

Before we create a new lab exercise, let's quickly test our old one with regards to the Watchdog.

1. **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.)*

2. **Terminate the Debugger.**

3. **Edit the source file by commenting out the Watchdog hold function.**

```
//    WDT_A_hold( WDT_A_BASE );
```

4. **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. WDT_A must be resetting the processor before we even get to the while{} loop.)

5. **Close the `lab_04a_clock` project.**

A couple of Questions about Watchdogs

6. **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
```

7. **Write the code to reset the Watchdog Timer.**

   Often this is called 'kicking the dog' or 'feeding the dog'.

   The purpose of the watchdog is reset the processor if your code doesn't reset it before its timer count runs out. What driverlib function can you used to reset the timer?

   *(Hint: look in the **Driver Library Users Guide** or the **`wdt_a.h`** file inside the **driverlib** folder.)*

   ____

File Management

8. Import the "Hello World" solution for `lab_02a_ccs`.

`Project → Import CCS Projects…`

Import the archived solution file:

`C:\msp430_workshop\`<*target*>`\solutions\lab_02a_ccs_solution.zip`

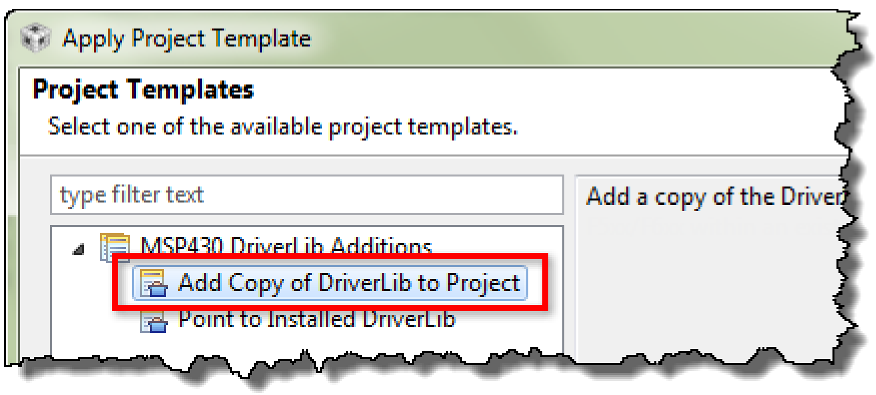
9. Rename the project to: `lab_04b_wdt`

10. Build the project, just to verify it still works correctly.

11. Import DriverLib into your project and add the appropriate path to the compiler's #include search path setting.

You could repeat the steps we completed to add DriverLib in **Lab3a** under the heading: *"Add MSP430ware Driverlib".* But it's easier to use the DriverLib project template that the MSP430ware team has provided.

`Right-Click on Project → Source → Apply Project Template…`



`Select "Add Copy of DriverLib to Project" and click OK`

This adds the appropriate DriverLib library to your project and adds the correct directory search path to the compiler's build options.

12. Build the project to verify that we haven't introduced any errors.

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.

15. Replace the watchdog hold code with the two WDT_A functions you wrote earlier.

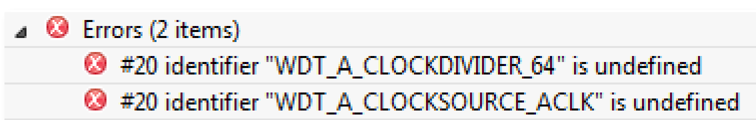
Remember that we didn't actually write this code. It 'holds' the watchdog by using register-based syntax. So, this is the line you want to replace:

```
    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 #6 on page 4-57.)*

16. Build the code to test that it's error-free (syntax wise).

Did you get an error? Unless you are a really experienced programmer and changed one other item, you should have received an error similar to this:



Where are these values defined? ______________________________

17. Include driverlib.h in your hello.c file.

Yep, when we added the driverlib code, we needed to add the driverlib header file, too. Actually, you can replace the #include of 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: ______________________________

Here are the results we obtained (at the time of writing), but they can vary with new compiler releases:

- *'F5529: 1, 10, 589 (respectively) … did you wait all the way to 589 before giving up?*
- *'FR5969: 0, 2, 141*

If you're really curious about what is happening under-the-hood, try examining the Watchdog control register. You can see it sets a different value for each of the divisor arguments. For example, on the 'FR5969, the arguments releate to these values:

- ÷ Default: 4 (i.e. ÷32K)
- ÷ 64: 7
- ÷ 512: 6
- ÷ 32K: 4

Keep it Running

20. Add the function call that will keep the CPU running without a watchdog reset.

Add the line of code to the while{} loop – our answer to question # in this lab – that will reset the watchdog and keep the program running.

```
        WDT_A_resetTimer( WDT_A_BASE );
```

> **Hint:** You may want to change the clock divisor back to WDT_A_CLOCKDIVER_64 to make it easier to see the change. Then, if the count goes past “1” you’ll know the watchdog is being serviced.

21. Build and run the program to observe the watchdog resetting the MSP430.

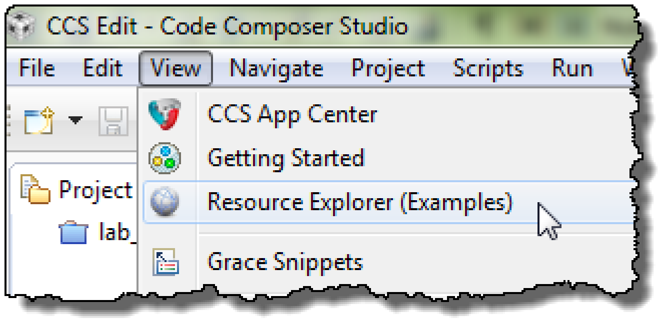
How many times will it run now? ____

22. When done playing with the program, terminate your debug session close the project.

Extra Credit – Try DriverLib's Watchdog Example (#3)

The driverlib library contains an example for 'watching' the watchdog timer. Give it a test to watch every time the watchdog rolls-over.

23. Import the `wdt_a_ex3_watchdogACLK` **project using the CCS Resource Explorer.**



If you cannot remember how to import a project using the 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 pin.

If you look in the User Guide for your MSP430 device, you can see that while the PDIR (pin direction) register is reset after a Power-Up Clear (PUC), the POUT value is left alone. This is the trick used to make the pin toggle after every watchdog reset.

Note, PUC was described during this chapter, while the GPIO pins were discussed in Chapter 3.

25. Build and run the program to observe the watchdog resetting the MSP430.

26. When you're done, close the project.

(Optional) Lab 4c – Using Crystal Oscillators

File Management

1. **Import `lab_04a_clock_solution`.**

   If you don't remember how to do this, refer back to lab step 1 (on page 4-47).

2. **Rename the project to `lab_04c_crystals`.**

3. **Make sure the project builds correctly.**

4. **Delete three files from the project:**
   - `myClocks.c`
   - `myClocks.h`
   - 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`
   - `myClocks.h`
   - `lab_04c_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 and .h files.**

   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 that the file imported correctly.**

Modify GPIO

8. **Add the following code to the initGpio() function in `main.c`.**

   Rather than having you build and run the project only to find out it doesn't work (like what happened to the course author), we'll give you a hint: connect the clock pins to the crystals.

   As you can see, the two different devices are pinned-out differently. Pick the code to match your processor.

F5529

```
// Connect pins to crystal in/out pins
GPIO_setAsPeripheralModuleFunctionInputPin(
        GPIO_PORT_P5,
        GPIO_PIN5 +                         // XOUT on P5.5
        GPIO_PIN4 +                         // XIN on P5.4
        GPIO_PIN3 +                         // XT2OUT on P5.3
        GPIO_PIN2                           // XT2IN on P5.2
);
```

or

FR5969

```
// Connect pins to crystal in/out pins
//  Note, PJ.6 and PJ.7 not needed as HF crystal is not present
GPIO_setAsPeripheralModuleFunctionInputPin(
        GPIO_PORT_PJ,
        GPIO_PIN4 +                         // LFXIN  on PJ.4
        GPIO_PIN5,                          // LFXOUT on PJ.5
        // GPIO_PIN6 +                      // HFXTIN on PJ.6
        // GPIO_PIN7                        // HFXOUT on PJ.7
        GPIO_PRIMARY_MODULE_FUNCTION
);
```

or

FR4133

```
// Set XT1 (low freq crystal pins) to crystal input (rather than GPIO):
GPIO_setAsPeripheralModuleFunctionInputPin(
        GPIO_PORT_P4,
        GPIO_PIN1 +                         // XIN  on P4.1
        GPIO_PIN2 ,                         // XOUT on P4.2
        GPIO_PRIMARY_MODULE_FUNCTION
);
```

By default – most MSP430 devices, these pins default to GPIO mode. Thus, we have to connect them to the crystals by reprogramming the GPIO.

One difference between the two processors – besides the port number being used – is that we had to specify "`GPIO_PRIMARY_MODULE_FUNCTION`" for the 'FR5969. This device allows multiple Peripheral I/O pin options. (Refer back to Chapter 3 for more details on this topic.)

Note: Above, we connect all four pins to their clock functions using the `GPIO_setAsPeripheralModuleFunctionInputPin()`.

Normally, connecting IN/OUT pins to Peripheral Functions requires two functions. For example , you would set the IN pins with the 'InputPin' function, while the setting the OUT pins using the `GPIO_setAsPeripheralModuleFunctionOutputPins()` function.

Connecting crytal pins works with either solution… so we *chose the one with less typing.*

---

9. Build and launch the debugger.

Debug

10. Set three breakpoints in the `myClocksWithCrystals.c` file.

Set a breakpoint <u>after</u> each instance of the code where we read the clock settings.

For example:

```

    // Verify if the default clock settings are as expected
    myACLK  = UCS_getACLK(           );
    mySMCLK = UCS_getSMCLK(          );
    myMCLK  = UCS_getMCLK(           );

    // Initialize XT1. Returns STATUS_SUCCESS if initializes s
    bReturn = UCS_LFXT1StartWithTimeout(
            UCS_BASE,
```

11. Run the code (click 'Resume') three times and record the clock settings:

Because of the way the FLL clock is handled on the **'F5529** and **'FR4133**, we have three places to record the clock values. With the **'FR5969**, you only need the first two columns.

Expression	Default Settings	First Clock Get	Second Clock Get
myACLK/1000			
mySMCLK/1000			
myMCLK/1000			

On the **'F5529** and **'FR4133**, why didn't SMCLK get set correctly on the first setup?
For example, on the 'F5529 we set SMCLK to use XT2CLK, but it didn't' seem to take:

____

____

Hint: Read the comments in the code itself (myClocksWithCrystals.c). It explains what caused this.

12. When done experimenting with this code, <u>terminate</u> the debugger and <u>close</u> the project.

Chapter 04 Appendix

Chapter 4 Worksheet (1)

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_97_00_47\driverlib\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_97_00_47\driverlib\driverlib\MSP430F5xx_6xx\ucs.h`

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

\msp430\MSP430ware_1_97_00_47\driverlib\examples\MSP430F5xx_6xx\ucs\ucs_ex4_XTSourcesDCOInternal.c

Reset and Operating Modes & Watchdog Timers

1. Name all 3 types of resets:

   BOR, POR, PUC

2. If the Watchdog (WDT) times out, which reset does it invoke?

   PUC

3. Write the DriverLib function that stops (halts) the watchdog timer:

   WDT_A_hold (WDT_A_BASE);

Chapter 4 Worksheet (2)

Power Management

4. ('F5529 Launchpad users only)

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

   initPowerMgmt (**PMM_CORE_LEVEL_1**);

Clocking

5. Why does MSP430 provide 3 different types of internal clocks?

   To meet the varying demands of performance, accuracy, and power.

   One clock runs the CPU, while the other two provide fast and slow/low-power clocking to the peripherals

   Name them:

   MCLK **SMCLK** **ACLK**

Chapter 4 Worksheet (3)

6. What is the speed of the crystal oscillators on your board?
(Hint: look in the Hardware section of the Launchpad Users Guide.)

'F5529:

```
#define  LF_CRYSTAL_FREQUENCY_IN_HZ  32768
#define  HF_CRYSTAL_FREQUENCY_IN_HZ  4000000
```

'FR5969:

```
#define  LF_CRYSTAL_FREQUENCY_IN_HZ  32768
#define  HF_CRYSTAL_FREQUENCY_IN_HZ  0
```

(for FR5969: We chose "0" for High Frequency crystal , since the Launchpad doesn't ship with one)

'FR4133:

```
#define  XT1_CRYSTAL_FREQUENCY_IN_HZ  32768
```

Chapter 4 Worksheet (4)

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

'F5529:

```
UCS_setExternalClockSource(
    LF_CRYSTAL_FREQUENCY_IN_HZ,
    HF_CRYSTAL_FREQUENCY_IN_HZ);
```

'FR5969:

```
CS_setExternalClockSource(
    LF_CRYSTAL_FREQUENCY_IN_HZ,
    HF_CRYSTAL_FREQUENCY_IN_HZ);
```

'FR4133:

```
CS_setExternalClockSource(XT1_CRYSTAL_FREQUENCY_IN_HZ);
```

Chapter 4 Worksheet (5)

8. At what frequencies are the clocks running? There's an API for that…
   Write the code that returns your current clock frequencies:

```
uint32_t myACLK  = 0;
uint32_t mySMCLK = 0;
uint32_t myMCLK  = 0;

myACLK  = UCS_getACLK();
mySMCLK = UCS_getSMCLK();
myMCLK  = UCS_getMCLK();
```

F5529 Prefix = 'UCS'
FR5969 Prefix = 'CS'
FR4133 Prefix = 'CS'

9. We didn't set up 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	SMCLK	MCLK
'F5529/FR4133	32 KHz	1.048 MHz	1.048 MHz
'FR5969	39 KHz	1 MHz	1 MHz

Chapter 4 Worksheet (6)

10. Set up ACLK:
    - Use REFO for the F5529 device
    - Use VLO for the FR5969/4133 devices

F5529 Prefix = 'UCS'
FR5969 Prefix = 'CS'
FR4133 Prefix = 'CS'

F5529

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

FR5969 / FR4133

```
// Setup ACLK
UCS_clockSignalInit(
    CS_ACLK,                 // Clock to setup
    CS_VLOCLK_SELECT,        // Source clock
    CS_CLOCK_DIVIDER_1
);
```

Chapter 4 Worksheet (7)

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    ____8000____________________

#define MCLK_FLLREF_RATIO __MCLK_DESIRED_FREQUENCY_IN_KHZ__/(UCS_REFOCLK_FREQUENCY/1024 )

// Set the FLL's clock reference clock to REFO
_UCS_clockSignalInit________(
    UCS_FLLREF,                // Clock you're configuring
    _UCS_REFOCLK_SELECT___,    // Clock Source
    UCS_CLOCK_DIVIDER_1 );

// Config the FLL's freq, let it settle, and set MCLK & SMCLK to use DCO+FLL as clk source
_UCS_initFLLSettle__________(
    MCLK_DESIRED_FREQUENCY_IN_KHZ,
    _MCLK_FLLREF_RATIO_____________);
```

Chapter 4 Worksheet (9)

11. (FR5969 Users only) Write the code to setup MCLK. It should be running at 8MHz using the DCO as its oscillator source.

```
// Set DCO to 8MHz
CS_setDCOFreq(
        _CS_DCORSEL_1___________,  // Set Frequency range (DCOR)
        _CS_DCOFSEL_3___________   // Set Frequency (DCOF)
);

// Set MCLK to use DCO clock source
_CS_clockSignalInit_____________(
    _CS_MCLK______________________________________,
    _CS_DCOCLK_SELECT_____________________________,
    UCS_CLOCK_DIVIDER_1 );
```

Chapter 4b Worksheet

6. 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 );
```

7. Write the code to 'kick the dog'?

```
_____WDT_A_resetTimer_____( WDT_A_BASE );
```

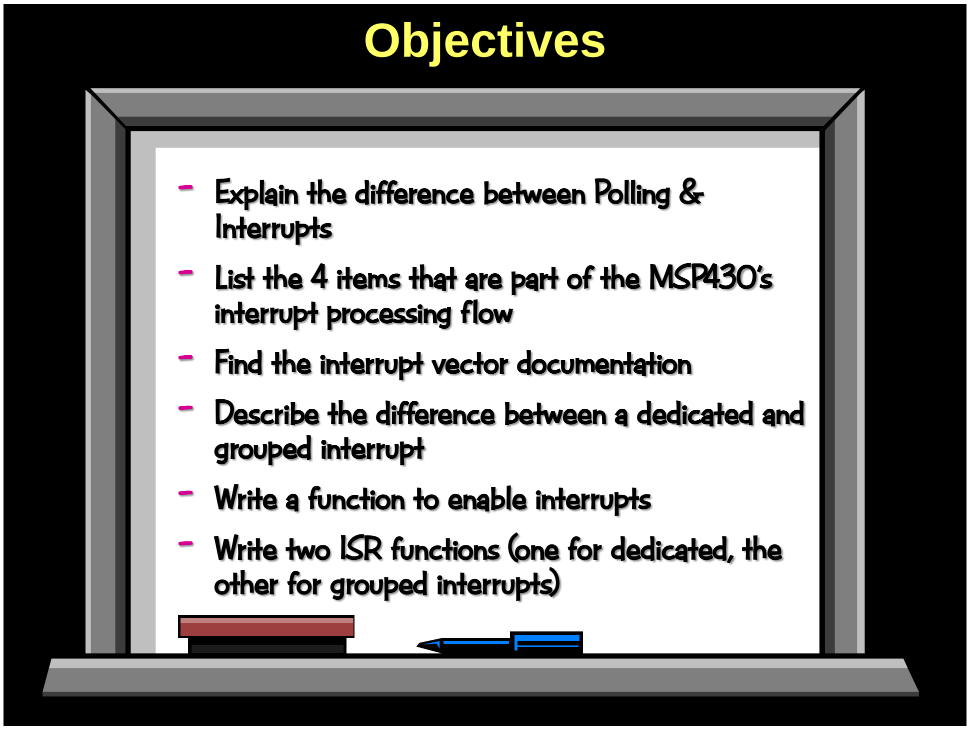
Introduction

What is an embedded system without interrupts?

If you just needed to solve a math problem you would most likely sit down and use a desktop computer. Embedded systems, on the other hand, take inputs from real-world events and then act upon them. These real-world events usually translate into ‘interrupts’ – asynchronous signals provided to the microcontroller: timers, serial ports, pushbuttons … and so on.

This chapter discusses how interrupts work; how they are implemented on the MSP430 MCU, and what code we need to write in order to harness their functionality. The lab exercises provided are relatively simple (using a pushbutton to generate an interrupt), but the skills we learn here will apply to all the remaining chapters of this workshop.

Learning Objectives



Chapter Topics

Interrupts **5-1**

Interrupts, The Big Picture *5-3*
Polling vs Interrupts 5-3
Processor States and Interrupts 5-5
Threads: Foreground and Background 5-6

How Interrupts Work *5-7*
1. Interrupt Must Occur 5-9
2. Interrupt is Flagged (and must be Enabled) 5-10
3. CPU's Hardware Response 5-12
4. Your Software ISR 5-14

Interrupts: Priorities & Vectors *5-17*
Interrupts and Priorities 5-17
Interrupt Vector (IV) Registers 5-18
Interrupt Vector Table 5-19

Coding Interrupts *5-22*
Dedicated ISR (Interrupt Service Routine) 5-22
Grouped ISR (Interrupt Service Routine) 5-24
Enabling Interrupts 5-26

Miscellaneous Topics *5-28*
Handling Unused Interrupts 5-28
Interrupt Service Routines – Coding Suggestions 5-29
GPIO Interrupt Summary 5-30
Interrupt Processing Flow 5-30

Interrupts and TI-RTOS Scheduling *5-31*
Threads – Foreground and Background 5-31
TI-RTOS Thread Types 5-33
Summary: TI-RTOS Kernel 5-36

Lab Exercise *5-37*

Interrupts, The Big Picture

While many of you are already familiar with interrupts, they are so fundamental to embedded systems that we wanted to briefly describe what they are all about.

From Wikipedia:

> *A **hardware interrupt** is an electronic alerting signal sent to the processor from an external device, either a part of the [device, such as an internal peripheral] or an external peripheral.*

In other words, the interrupt is a signal which notifies the CPU that an event has occurred. If the interrupt is configured, the CPU will respond to it immediately – as described later in this chapter.

Polling vs Interrupts

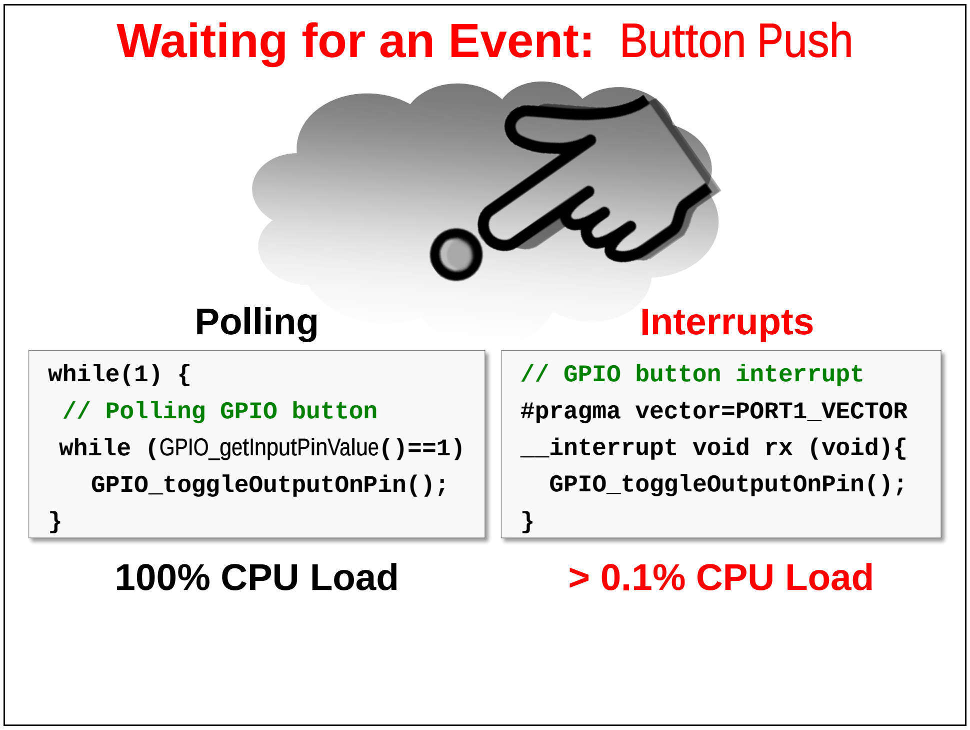
In reality, though, there are two methods that events can be recognized by the processor. One is called "Polling"; the other is what we just defined, "Interrupts".

We start with a non-engineering analogy for these two methods. If you've ever taken a long family vacation, you've probably dealt with the "Are we there yet" question. In fact, kids often ask it over-and-over again. Eventually … the answer will be, "Yes, we're there". The alternative method is when my spouse says, "Wake me up when we get there".

Both methods signal that we have arrived at our destination. In most cases, though, the use of Interrupts tends to be <u>much more efficient</u>. For example, in the case of the MSP430, we often want to sleep the processor while waiting for an event. When the event happens and signals us with an interrupt, we can wake up, handle the event and then return to sleep waiting for the next event.

A real-world event might be our system responding to a push-button. Once again, the event could be handled using either **Polling** or **Interrupts**.

It is common to see "simple" example code utilize **Polling**. As you can see from the left-side example below, this can simply consist of a while{} loop that keeps repeating until a button-push is detected. The big downfall here, though, is that the processor is constantly running– asking the question, "Has the button been pushed, yet?"



The example on the right shows an **Interrupt** based solution. Since this code is not constantly running, as in the previous example's while{} loop, the CPU load is very low.

Why do simple examples often ignore the use of interrupts? Because they are "simple". Interrupts, on the other hand, require an extra three items to get them running. We show two of them in the right-hand example above.

- The #pragma sets up the interrupt vector. The MSP430 has a handy pragma which makes it easy to configure this item. (Note: we'll cover the details of all these items later in this chapter.)
- The __interrupt keyword tells the compiler to code this function as an interrupt service routine (ISR). Interrupt functions require a context save and restore of any resources used within them.

While not shown above, we thought we'd mention the third item needed to get interrupts to work. For a CPU to respond to an interrupt, you also need to enable the interrupt. (Oh, and you may also have to setup the interrupt source; for example, we would have to configure our GPIO pin to be used as an interrupt input.)

So, in this chapter we leave the simple and inefficient examples behind and move to the real-world – where real-world embedded systems thrive on interrupts.

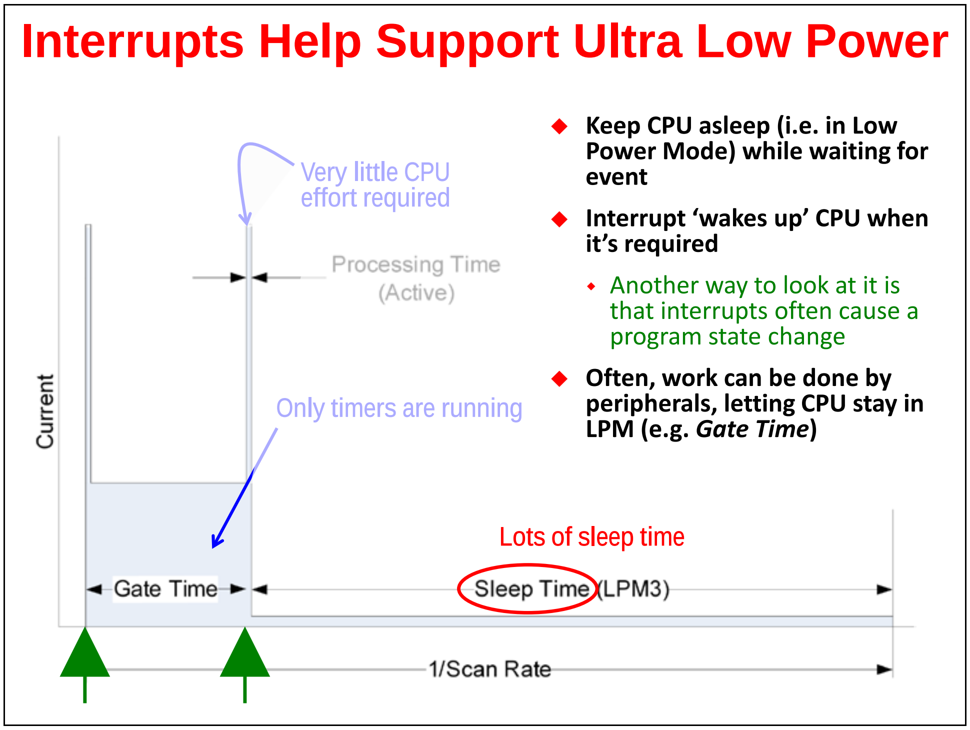
Processor States and Interrupts

In a previous chapter we covered many of the MSP430's processor states. To summarize, the MSP430 CPU can reside in: Reset, Active, or one of many Low-Power Modes (LPM). In many cases, interrupts cause the CPU to change states. For example, when sitting in Low Power Mode, an interrupt can "wake-up" the processor and return it to its active mode.

To help demonstrate this point, we stole the following slide from a discussion about Capacitive Touch. While most of this slide's content is not important for our current topic, we thought the *current vs time graph* was interesting. It tries to visually demonstrate the changing states of the device by charting power usage over time.

Notice the four states shown in this diagram:

- Notice how the current usage goes up at the beginning event – this is when the CPU is woken up so it can start a couple of peripherals (timers) needed to read the CapTouch button.
- The CPU can then go back to sleep while the sensor is being 'read' by the timers.
- When the read is complete (defined by something called "Gate" time, the CPU gets interrupted and wakes up again in order to calculate the CapTouch button's value from the sensor data.
- Finally the CPU (and CapTouch hardware) can go back to sleep and wait for another system wake-up event.

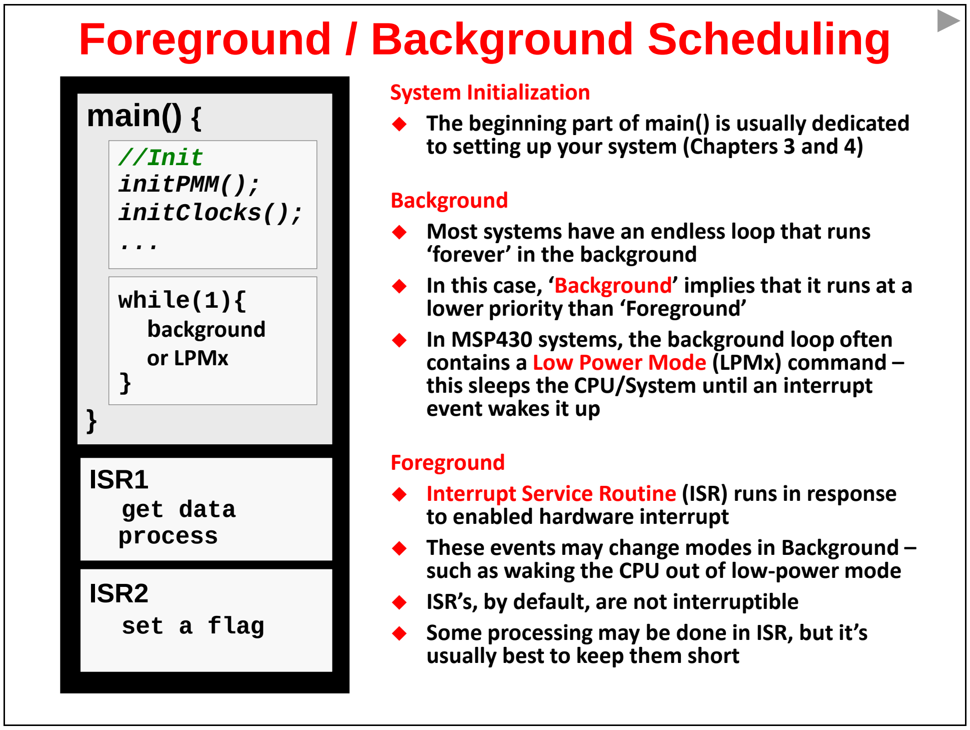


Threads: Foreground and Background

We conclude our Interrupts introduction by defining a few common terms used in interrupt-driven systems: **Thread**, **Foreground** and **Background**.

If you look at the “code” below, you will see that there are three individual – and independent – code segments below: main, ISR1, and ISR2.

We use the word *independent* because, if you were to examine the code in such a system, there are no calls between these three routines. Each one begins and ends execution without calling the others. It is common to call these separate segments of code: “Threads”.



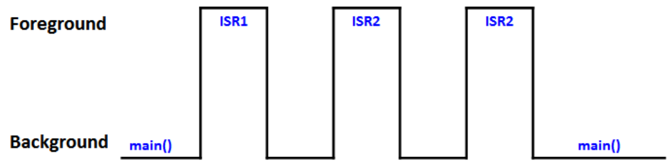
As we’ve seen in the workshop already, it is our main() thread that begins running once the processor has been started. The compiler’s initialization routine calls main() when its work is done. (In fact, this is why all C programs start with a main() function. Every compiler works the same way, in this regard.)

With the main() thread started, since it is coded with a while(1) loop, it will keep running forever. That is, unless a hardware interrupt occurs.

When an enabled interrupt is received by the CPU, it preempts the main() thread and runs the associated ISR routine – for example, ISR1. In other words, the CPU stops running main() temporarily and runs ISR1; when ISR1 completes execution, the CPU goes back to running main().

Here's where the terms **Foreground** and **Background** come into play. We call main() the **Background** thread since it is our "default" thread; that is, the program is designed such that we start running main() and go back to it whenever we're done with our other threads, such as ISR1.

Whenever an interrupt causes another thread to run, we call that a **Foreground** thread. The foreground threads preempt the Background thread, returning to the Background once completed.



The words "**Foreground**" and "**Background**" aren't terribly important. They just try to provide a bit of context that can be visualized in this common way.

It should be noted that it's important to keep your interrupt service routines short and quick. This, again, is common practice for embedded systems.

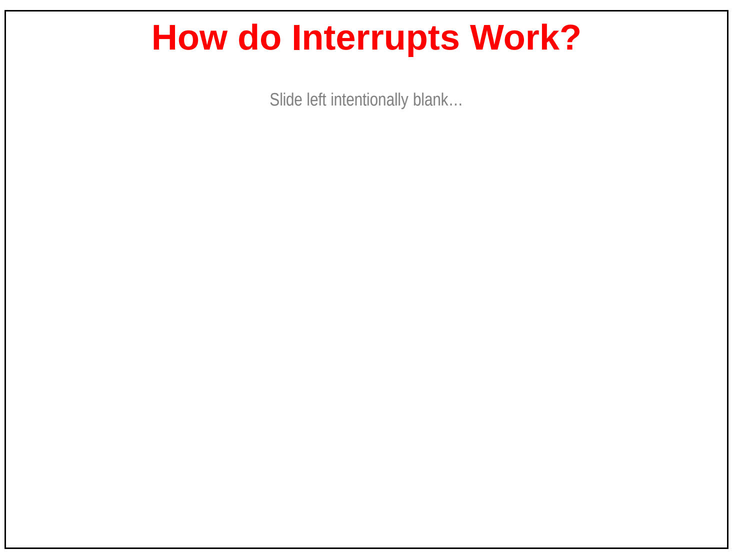
Note: We realize that our earlier definition of "Thread" was a little weak. What we said was true, but not complete. The author's favorite definition for "Thread" is as follows:

"A function or set of functions that operate independently of other code – running within their own context."

The key addtion here is that a thread runs within its own context. When switching from one thread to another, the context (register values and other resources) must be saved and restored.

How Interrupts Work

Now that we have a rough understanding of what interrupts are used for, let's discuss what mechanics are needed to make them work. Hint, there are 4 steps to getting interrupts to work…

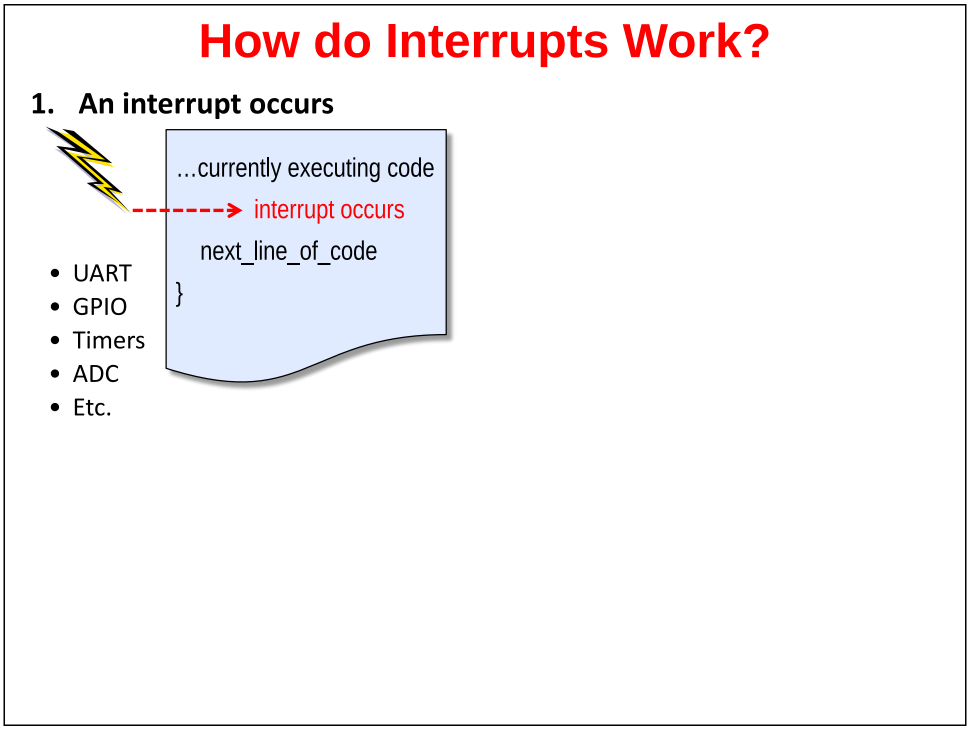


If you've been reading this chapter, you might notice that we've already covered these four items. Over the next few pages we enumerate these steps again, filling-in additional details.

Notes

1. Interrupt Must Occur

For the processor to respond to an interrupt, it must have occurred. There are many possible sources of interrupts. Later in this chapter we will delve into the MSP430 datasheet which lists all of the interrupt sources.



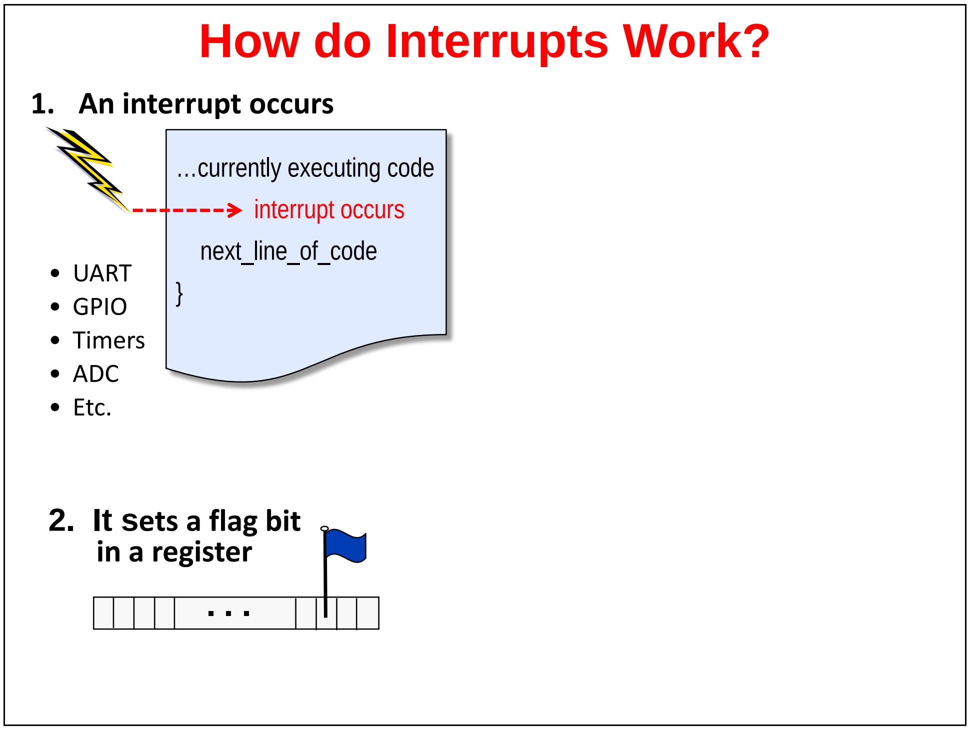
Suffice it to say that most peripherals can generate interrupts to provide status and information to the CPU. Most often, the interrupt indicates that data is available (e.g. serial port) and/or an event has occurred that needs processing (e.g. timer). In some cases, though, an interrupt may be used to indicate an error or exception in a peripheral that the CPU needs to handle.

Interrupts can also be generated from GPIO pins. This is how an external peripheral, or some other controller, can signal the MSP430 CPU. Most MSP430 devices allow the pins from the first two I/O ports (P1 and P2) to be individually configured for interrupt inputs. On the larger devices, there may be additional ports that can be configured for this, as well.

Finally, your software can often generate interrupts. The logic for some interrupts on the processor allow you to manually set a flag bit, thus ‘emulating’ a hardware interrupt. Not all interrupts provide this feature, but when available, it can be a handy way to test your interrupt service routine.

2. Interrupt is Flagged (and must be Enabled)

When an interrupt signal is received, an interrupt flag (IFG) bit is latched. You can think of this as the processor's "copy" of the signal. As some interrupt sources are only on for a short duration, it is important that the CPU registers the interrupt signal internally.



MSP430 devices are designed with "distributed" interrupt management. That is, most IFG bits are found inside each peripheral's control registers; this is different from most processors which have a common, dedicated set of interrupt registers.

The distributed nature of the interrupts provides a number of benefits in terms of device flexibility and future feature expansion; further, it fits nicely with the low-power nature of the MSP430.

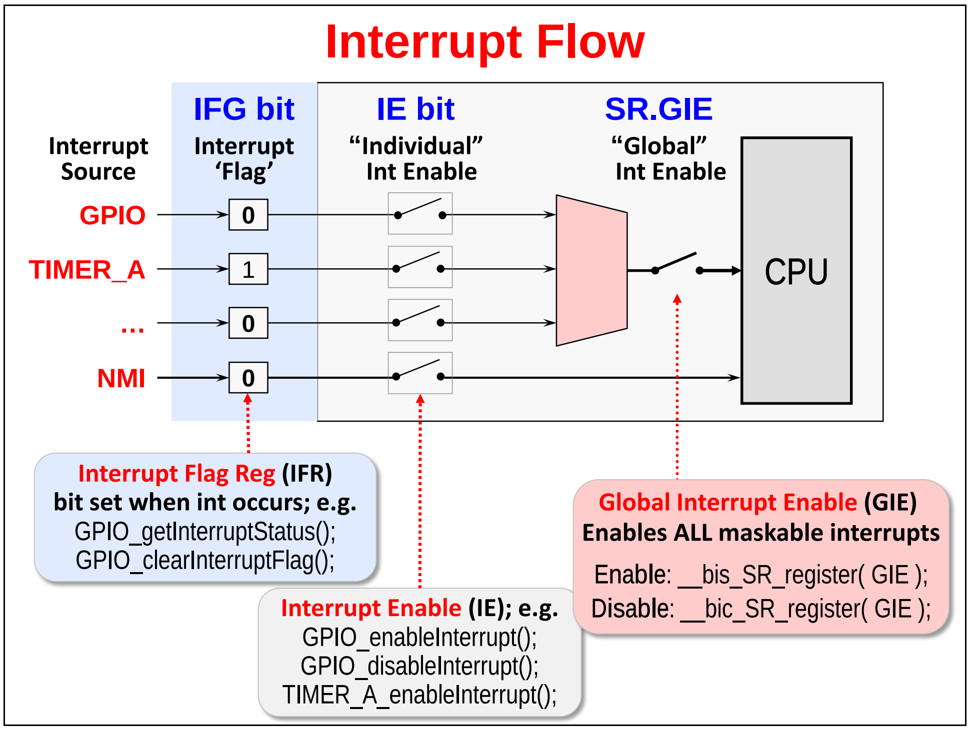
The only 'negative' of distributed interrupts might be that it's different — it's just that many of us older engineers are used to seeing all the interrupts grouped together. Bottom line, though, is that working with interrupts (enabling interrupts, clearing flags, responding to them) is the same whether the hardware is laid out centrally or in a distributed fashion.

Interrupt Flow

How does the interrupt signal reach the CPU?

We've just talked about the interrupt flag (IFG) bit – let's start there. As described on the previous page, when the interrupt source signal is received, the associated IFG bit is set. In fact, DriverLib contains functions to read the status of most IFG bits. (Handy in those few cases where you need to poll an interrupt source.)

When the IFG is set, the MSP430 *device* now sees that the signal has occurred, but the signal hasn't made its way to the *CPU*, yet. For that to happen, the interrupt must be **enabled**.



Interrupt enable bits (IE) exist to protect the CPU … and thus, your program. Even with so many peripherals and interrupt sources, it's likely that your program will only care about a few of them. The enable bits provide your program with 'switches' that let you ignore all those sources you don't need.

By default, all interrupt bits are disabled (except the Watchdog Timer). It is your program's responsibility to enable those interrupt sources that are needed. To that end, once again, DriverLib provides a set of functions that make it easy for you to set the necessary IE bits.

Finally, there's a "master" switch that turns all interrupts off. This lets you turn off interrupts without having to modify all of the individual IE bits. The MSP430 calls this the global interrupt enable (GIE). It is found in the MSP430 Status Register (SR).

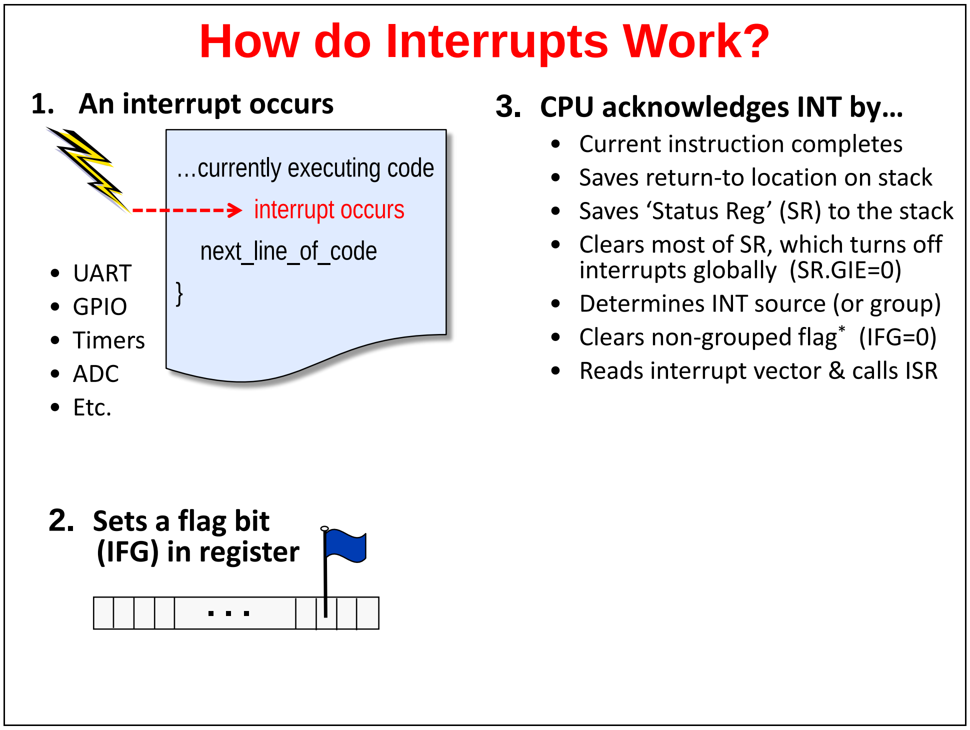
Why would you need a GIE bit? Sometimes your program may need to complete some code *atomically*; that is, your program may need to complete a section of code without the fear that an interrupt could preempt it. For example, if your program shares a global variable between two threads – say between main() and an ISR – it may be important to prevent interrupts while the main code reads and modifies that variable.

Note: There are a few non-maskable interrupts (NMI). These sources bypass the GIE bit. These interrupts are often considered critical events – i.e. 'fatal' events – that could be used to provide a *warm reset* of the CPU.

3. CPU's Hardware Response

At this point, let's assume you have an interrupt that has: occurred; been flagged; and since it was enabled, its signal has reached the CPU. What would the CPU do in response to the interrupt?

Earlier in the chapter we stated: "The interrupt preempts the current thread and starts running the interrupt service routine (ISR)." While this is true, there are actually a number of items performed by the hardware to make this happen – as shown below:



We hope the first 3 items are self-explanatory; the current instruction is completed while the Program Counter (PC) and Status Register (SR) are written to the system stack. (You might remember, the stack was setup for the MSP430 by the compiler's initialization routine. Please refer to the compiler user's guide for more information.)

After saving the context of SR, the interrupt hardware in the CPU clears most of the SR bits. Most significantly, it clears GIE. That means that by default, whenever you enter an ISR function, all maskable interrupts have been turned off. (We'll address the topic of 'nesting' interrupts in the next section.)

The final 3 items basically tell us that the processor figures out which interrupt occurred and calls the associated interrupt service routine; it also clears the interrupt flag bit (if it's a dedicated interrupt). The processor knows which ISR to run because each interrupt (IFG) is associated with an ISR function via a look-up table – called the Interrupt Vector Table.

Interrupt Vector Table – How is it different than other MCU's?

The MSP430 Vector Table is similar and dissimilar to other microcontrollers:

- The MSP430, like most microcontrollers, uses an Interrupt Vector Table. This is an area of memory that specifies a vector (i.e. ISR address) for each interrupt source.
- Some processors provide a unique ISR (and thus, vector) for every interrupt source. Other processors provide only 1 interrupt vector and make the user program figure which interrupt occurred. To maximize flexibility and minimize cost and power, the MSP430 falls in between these two extremes. There are some interrupts which have their own, dedicated interrupt vector – while other interrupts are logically grouped together.
- Where the MSP430 differs from many other processors is that it includes an Interrupt Vector (IV) register for each grouped interrupt; reading this register returns the highest-priority, enabled interrupt for that group of interrupt sources. As we'll see later in this chapter, all you need to do is read this register to quickly determine which specific interrupt to handle.

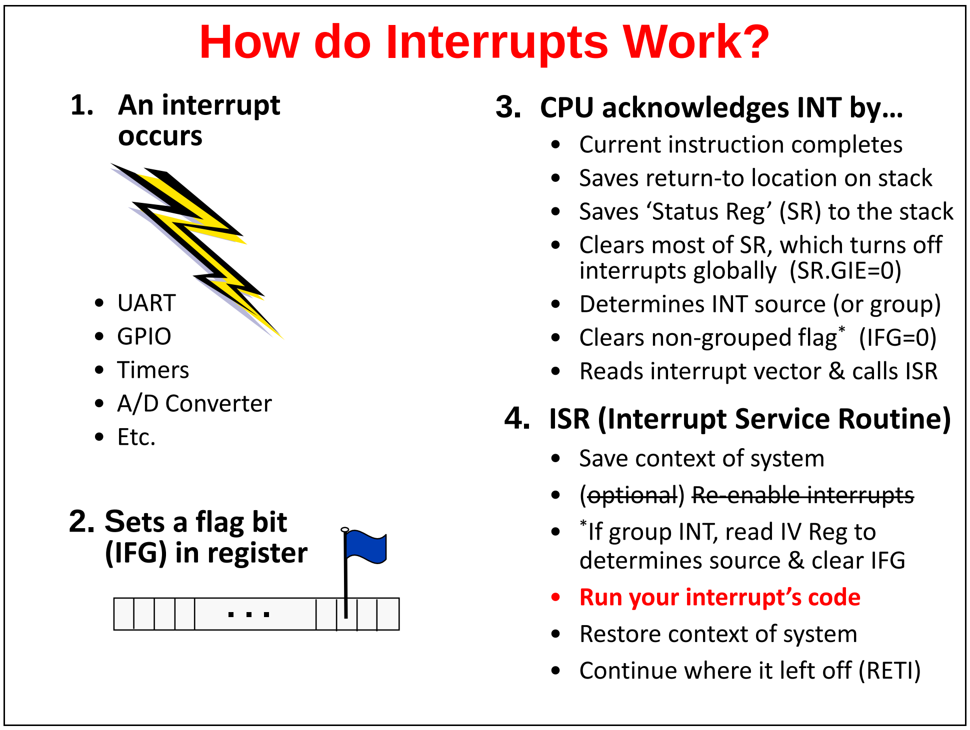
Note: We'll describe Interrupt Vector Table in more detail later in the chapter.

4. Your Software ISR

An *interrupt service routine* (ISR), also called an *interrupt handler*, is the code you write that will be run when a hardware interrupt occurs. Your ISR code must perform whatever task you want to execute in response to the interrupt, but without adversely affecting the threads (i.e. code) already running in the system.

Before we examine the details of the ISR; once again, how did we get to this point?

> Looking at the diagram below, we can see that (1) the interrupt must have occurred; (2) the processor flags the incoming interrupt; (3) if enabled, the interrupt flag signal is routed to the CPU where it saves the Status Register and Return-to address and then branches to the ISR's address found in the appropriate location in the vector table. (4) Finally, your ISR is executed.



The crux of the ISR is doing what needs to be done in response to the interrupt; the 4th bullet (listed in red) reads:

- **Run your interrupt's code**

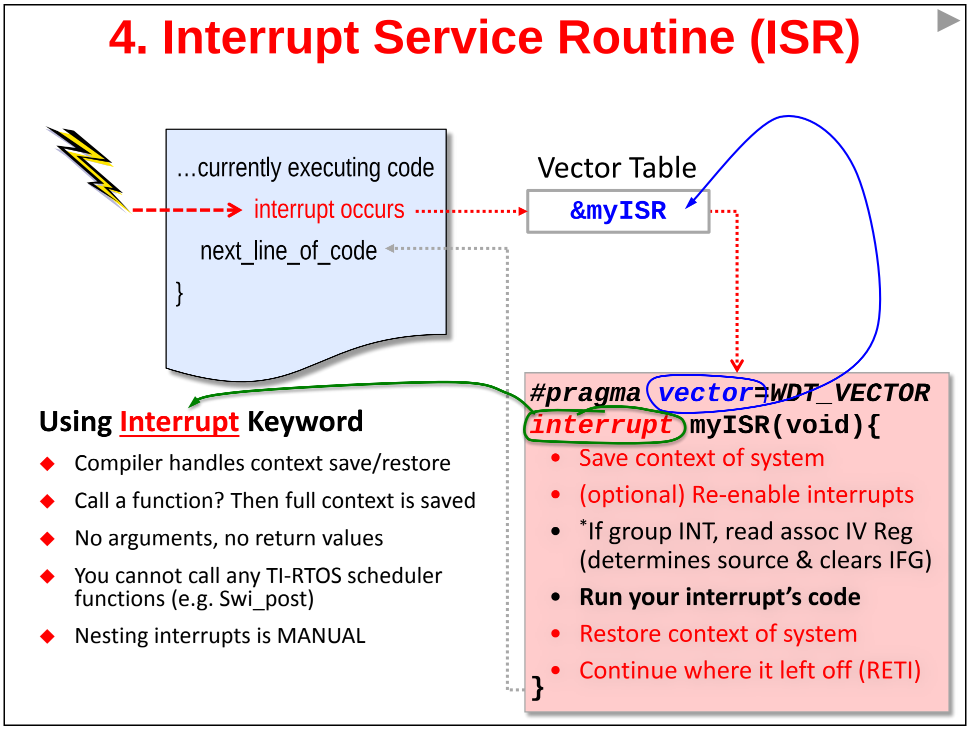
This is meant to describe the code you write to handle the interrupt. For example, if it's a UART interrupt, your code might read an incoming byte of data and write it to memory.

We'll discuss the 2nd (optional) bullet on the next page.

The 3rd bullet indicates that if this is a "grouped" interrupt, you have to add code to figure out which interrupt, in the group, needs to be handled. This is usually done by reading the group's IV register. (This bullet was in red because it is code you need to write.)

The other bullets listed under "4. ISR" are related to saving and restoring the context of the system. This is required so that the condition mentioned earlier can be met: *"without adversely affecting the code threads already running in the system."*

We show the interrupt flow in a slightly different fashion in the following diagram. As you can see, when an enabled interrupt occurs, the processor will look up the ISR's branch-to address from a specific address in memory (called the *interrupt vector*). For the MSP430, this address is defined using the *vector* pragma.



The context of the system – for example, the CPU registers used by the ISR – must be saved before running your code and restored afterwards. Thankfully, the compiler handles this for you when the function is declared as an *interrupt*. (As part of the "context restore", the compiler will return to running the previous thread of code by using the RETI instruction).

Please note the bullets under "Using the Interrupt Keyword" from the preceding diagram.

Using this keyword, the compiler handles all of the context save/restore for you and knows how to return to your previous code – even restoring the original value for the Status Register (SR).

Hint: If you call a function within your ISR, the compiler will have to save/restore every CPU register, not just the ones that it uses to implement your C code. This is because it doesn't know what resources the function call may end up using.

Since the interrupt occurs asynchronously to the background thread, you cannot pass arguments to and receive return values from the ISR. You must communicate between threads using global variables (or other appropriate data objects).

TI's real-time operating system (TI-RTOS) provides a rich set of scheduling functions that are often used within interrupt service routines. Be aware, though, that some of these functions can only be used with RTOS "managed" interrupts. In fact, it's actually easier to let TI-RTOS manage your interrupts; it automatically handles plugging the interrupt vector as well as context save/restore. (All you have to do is write a standard C function.) But, the details of TI-RTOS are outside the scope of this workshop. While we provide a brief discussion of TI-RTOS at the end of this chapter, please refer to the *Introduction to TI-RTOS Kernel* workshop for more details.

Nesting Interrupts (not recommended)

Finally, while the MSP430 allows nesting of interrupts, it is not recommended.

- *Nesting* interrupts means one interrupt can interrupt another interrupt.
- You must manually configure nesting. That is, before running your interrupt handling code you must:
  - Disable any interrupts that you do not want to occur during your ISR. In other words, you must first save, then disable, any IE bit that correlates to an interrupt that you do not want to interrupt your ISR.
  - Then, turn on interrupts globally by setting GIE = 1.
  - At this point you can run your code that responds to the original interrupt. It may end up being interrupted by any source that you left enabled.
  - When you've completed your original interrupt code, you need to disable interrupts before returning from the function. That is, set GIE = 0. (This is the state GIE was in when entering your ISR code.
  - You can now restore the IE bits that you saved before enabling GIE.
  - At this point, you can return from the ISR and let the compiler's code handle the remaining context save and return branch back to the original thread.
- In general, it's considered better programming practice to keep interrupt service routines very short – i.e. lean-and-mean. Taking this further, with low-power and efficiency in mind, the MSP430 team recommends you follow the no-nesting general principle.

Hint: We encourage you to avoid nesting, if at all possible. Not only is it difficult, and error prone, it often complicates your programs ability to reach low-power modes.

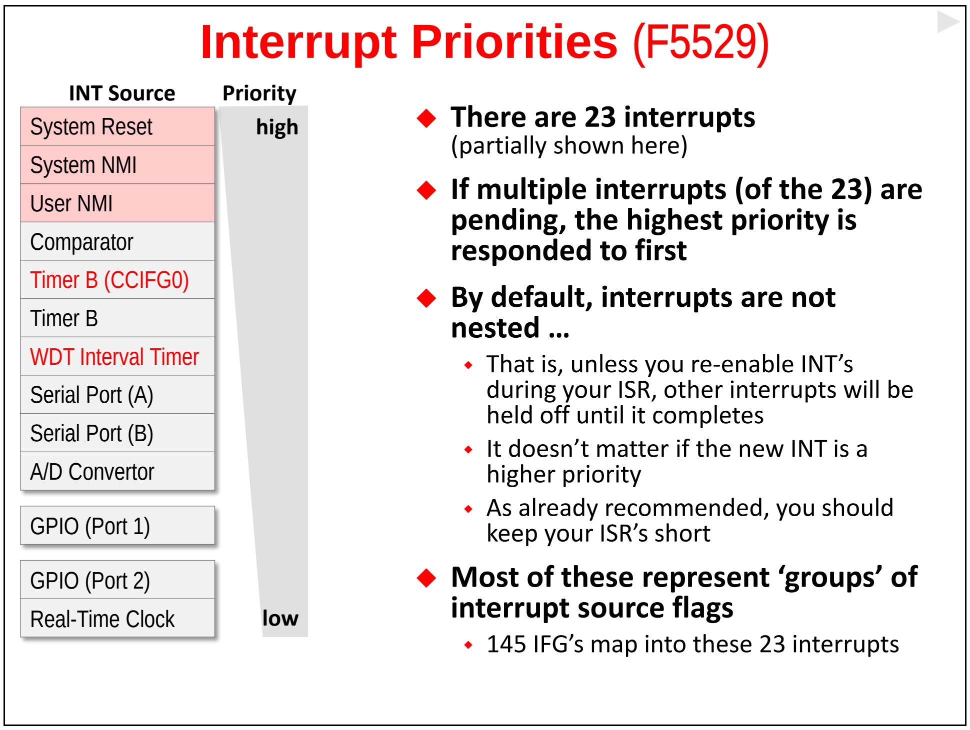
Interrupts: Priorities & Vectors

Interrupts and Priorities

Each MSP430 device datasheet defines the *pending* priority for each of its hardware interrupts. In the case of the MSP430F5529, there are 23 interrupts shown listed below in decreasing priority.

In the previous paragraph we used the phrase "pending priority" deliberately. As you might remember from the last topic in this chapter, interrupts on the MSP430 do not nest within each other by default. This is because the global interrupt (GIE) bit is disabled when the CPU acknowledges and processes an interrupt. Therefore, if an interrupt occurs while an ISR is being executed, it will have to wait for the current ISR to finish before it can be handled … even if the new interrupt is of higher priority.

On the other hand, if two interrupts occur at the same time – that is, if there are two interrupts currently pending – then the highest priority interrupt is acknowledged and handled first.



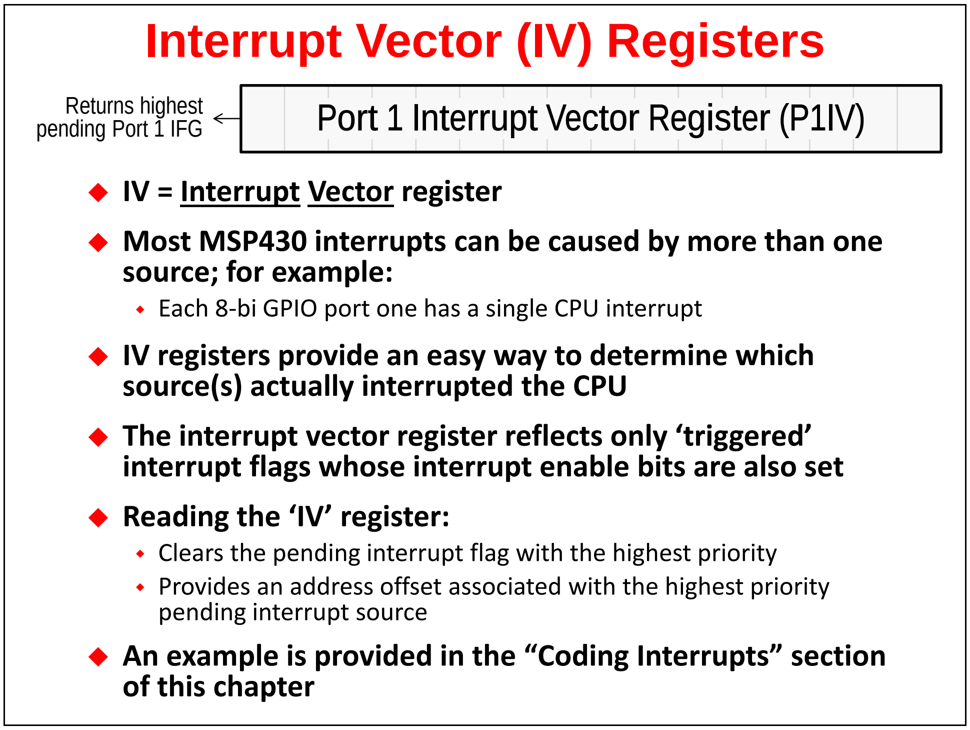
Most of the 23 interrupts on the 'F5529 represent 'groups' of interrupts. There are actually 145 interrupt sources – each with their own interrupt flag (IFG) – that map into these 23 interrupts.

For example, the "Timer B (CCIFG0)" interrupt represents a single interrupt signal. When the CPU acknowledges it, it will clear its single IFG flag.

On the other hand, the next interrupt in line, the "Timer B" interrupt, represents all the rest of the interrupts that can be initiated by Timer0_B. When any one of the interrupts in this group occurs, the ISR will need to determine which specific interrupt source occurred and clear its flag (along with executing whatever code you want to associate with it).

Interrupt Vector (IV) Registers

As has been mentioned a couple of times in this chapter, to make responding to *grouped* interrupts easier to handle, the MSP430 team created the concept of *Interrupt Vector (IV) Registers*. Reading an IV register will return the *highest-priority, pending interrupt* in that group; it will also clear that interrupts associated flag (IFG) bit.



For grouped interrupts, most users read the IV register at the beginning of the ISR and use the return value to pick the appropriate code to run. This is usually implemented with a Switch/Case statement. (We will explore an example of this code later in the chapter.)

Interrupt Vector Table

We can expand the previous interrupt source & priority listing to include a few more items. First of all, we added a column that provides the IV register associated with each interrupt. (Note, the two names shown in red text represent the IFG bits for dedicated/individual interrupts.)

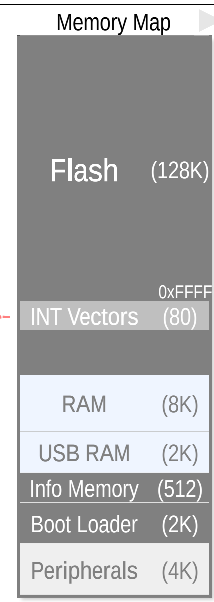
Additionally, the first 3 rows (highlighted with red background fill) indicate that these interrupt groups are non-maskable; therefore, they bypass the GIE bit.

Interrupt Vectors & Priorities (F5529)

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

Legend:

Non-maskable	Group'd IFG bits
Maskable	Dedicated IFG bits



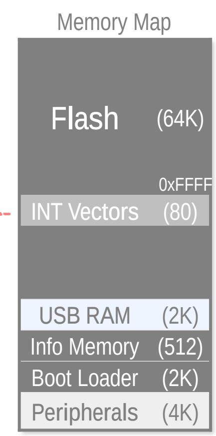
The final column in the above diagram hints at the location of each interrupts address vector in the memory map. For example, when using the WDT as an interval timer, you would put the address of your appropriate ISR into location "57". As we saw in a previous topic, this can easily be done using the *vector* pragma.

The MSP430 devices reserve the range 0xFFFF to 0xFF80 for the interrupt vectors. This means that for the 'F5529, the address for the System Reset interrupt service routine will sit at addresses 0xFFFE – 0xFFFF. (A 16-bit address requires two 8-bit memory locations.) The remaining interrupt vectors step down in memory from this point. The map to the right of the table shows where the interrupt vectors appear within the full MSP430 memory map.

Here's a quick look at the same table showing the MSP430FR5969 interrupt vectors and priorities. The list is very similar to the 'F5529; the main differences stem from the fact that the two devices have a slightly different mix of peripherals.

Interrupt Vectors & Priorities ('FR5969)

INT Source	IV Register	Vector Address	Loc'n	Priority
System Reset	SYSRSTIV	RESET_VECTOR		high
System NMI	SYSSNIV	SYSNMI_VECTOR	54	
User NMI	SYSUNIV	UNMI_VECTOR	53	
Comparator_E	CEIV	COMP_B_VECTOR	52	
Timer B0 (CCIFG0)	CCIFG0	TIMER0_B0_VECTOR	51	
Timer B0	TB0IV	TIMER0_B1_VECTOR	50	
WDT Interval Timer	WDTIFG	WDT_VECTOR	49	
Serial Port (A0)	UCA0IV	USCI_A0_VECTOR	48	
Serial Port (B0)	UCB0IV	USCI_B0_VECTOR	47	
ADC12_B	ADC12IV	TIMER0_B0_VECTOR	46	
GPIO (Port 1)	P1IV	PORT1_VECTOR	39	
Real-Time Clock	RTCIV	RTC_VECTOR	31	
AES256 Accelerator	AESRDYIFG	AES256_VECTOR	30	low



Legend:		
	Non-maskable	Group'd IFG bits
	Maskable	Dedicated IFG bits

The preceding interrupt tables were re-drawn to make them easier to view when projected during a workshop. The following slide was captured from ‘F5529 datasheet. This is what you will see if you examine the MSP430 documentation.

‘F5529 Vector Table (From Datasheet)

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
System Reset Power-Up External Reset Watchdog Timeout, Password Violation Flash Memory Password Violation	WDTIFG, KEYV (SYSRSTIV)(1)(2)	Reset	0FFFEh	63, highest
System NMI PMM Vacant Memory Access JTAG Mailbox	SVMLIFG, SVMHIFG, DLYLIFG, DLYHIFG, VLRLIFG, VLRHIFG, VMAIFG, JMBNIFG, JMBOUTIFG (SYSSNIV)(1)	(Non)maskable	0FFFCh	62
User NMI NMI Oscillator Fault Flash Memory Access Violation	NMIIFG, OFIFG, ACCVIFG, BUSIFG (SYSUNIV)(1)(2)	(Non)maskable	0FFFAh	61
Comp_B	...tor B interrupt flags ...	Maskable	[illegible]	60
DMA	DMA0IFG, DMA1IFG, DMA2IFG (DMAIV)(1)(3)	Maskable	0FFE4h	50
TA1	TA1CCR0 CCIFG0(3)	Maskable	0FFE2h	49
TA1	TA1CCR1 CCIFG1 to TA1CCR2 CCIFG2, TA1IFG (TA1IV)(1)(3)	Maskable	0FFE0h	48
I/O Port P1	P1IFG.0 to P1IFG.7 (P1IV)(1)(3)	Maskable	0FFDEh	47
USCI_A1 Receive or Transmit	UCA1RXIFG, UCA1TXIFG (UCA1IV)(1)(3)	Maskable	0FFDCh	46
USCI_B1 Receive or Transmit	UCB1RXIFG, UCB1TXIFG (UCB1IV)(1)(3)	Maskable	0FFDAh	45
TA2	TA2CCR0 CCIFG0(3)	Maskable	0FFD8h	44
TA2	TA2CCR1 CCIFG1 to TA2CCR2 CCIFG2, TA2IFG (TA2IV)(1)(3)	Maskable	0FFD6h	43
I/O Port P2	P2IFG.0 to P2IFG.7 (P2IV)(1)(3)	Maskable	0FFD4h	42
RTC_A	RTCRDYIFG, RTCTEVIFG, RTCAIFG, RT0PSIFG, RT1PSIFG (RTCIV)(1)(3)	Maskable	0FFD2h	41
			0FFD0h	40

Each device’s datasheet provides a similar *vector table* listing. If you are using the ‘G2553 or ‘FR5969 devices, for example, you will find a similar table in each of their respective datasheets.

Coding Interrupts

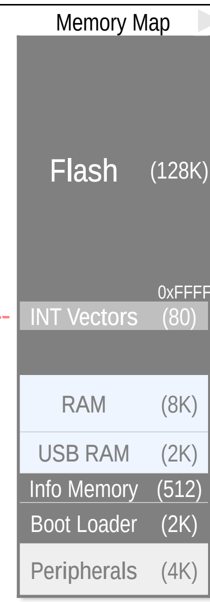
As previously discussed, the code within your interrupt service routine will vary slightly based on whether it handles a dedicated, single interrupt or if it handles a grouped interrupt. We will cover both cases; starting with the easier, dedicated case.

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
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	low

Legend: Non-maskable / Maskable; Group'd IFG bits / Dedicated IFG bits

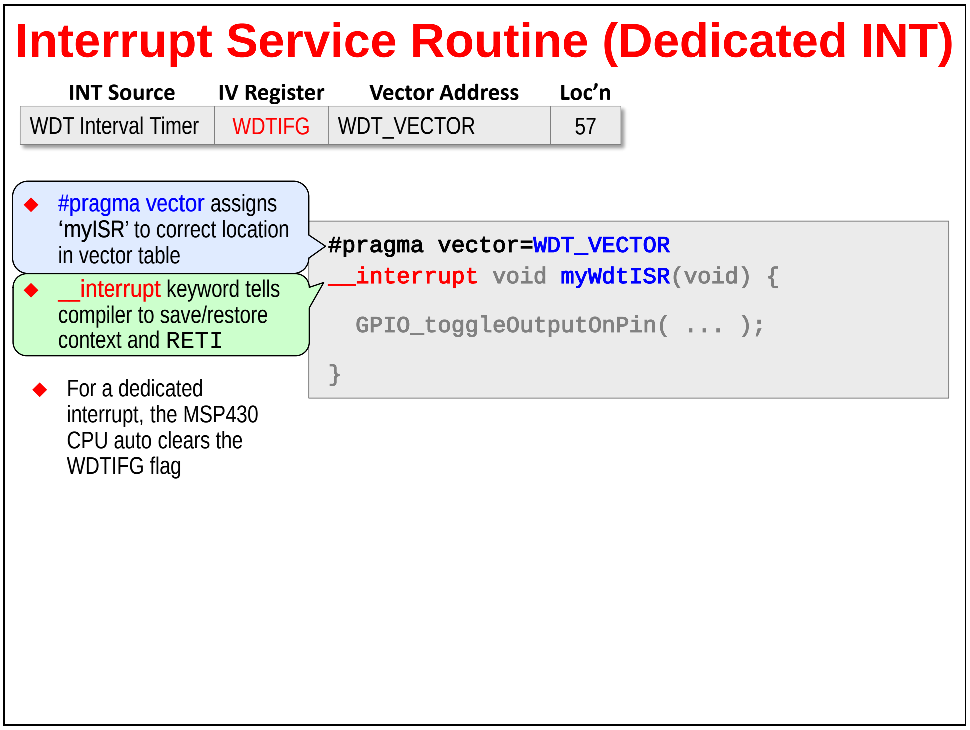


The watchdog interrupt flag vector (WDTIFG) is a dedicated interrupt; therefore, your ISR code only needs to respond to the single interrupt condition. Additionally, because it is a dedicated interrupt, the CPU hardware automatically clears the WDTIFG bit when responding to the interrupt and branching to your ISR.

When writing an ISR for dedicated interrupts, you code must address three items:

1. Put the ISR address into the vector table (using the vector #pragma)
2. Save/Restore the CPU context (using the __interrupt keyword)
3. Write your interrupt handler code (in other words, "Do what needs doing")

We will use the following code example to demonstrate these three items.



Plug the Vector Table (#pragma vector)

In our example, the following line of code:

```
#pragma vector=WDT_VECTOR
```

tells the compiler to associate the function (on the following line) with the WDT_VECTOR. Looking in the MSP430F5529 device-specific linker command file, you should find this vector name ("WDT_VECTOR") associated with vector #57. This matches with the datasheet documentation we looked at earlier in the chapter.

Save/Restore CPU context (__interrupt keyword)

The *__interrupt* keyword tells the compiler that this function is an interrupt service routine and thus it needs to save (and then restore) the context of the processor (i.e. CPU registers) before (and after) executing the function's code.

Don't forget, functions using the *__interrupt* keyword cannot accept arguments or return values.

Hint: Empirical analysis shows that "__interrupt" and "interrupt" are both accepted by the compiler.

Your Interrupt Code

In this example, the output of a GPIO pin is toggled every time the watchdog timer interrupt event occurs. Not all ISR's will be this short, but we hope this gives you a good starting example to work from.

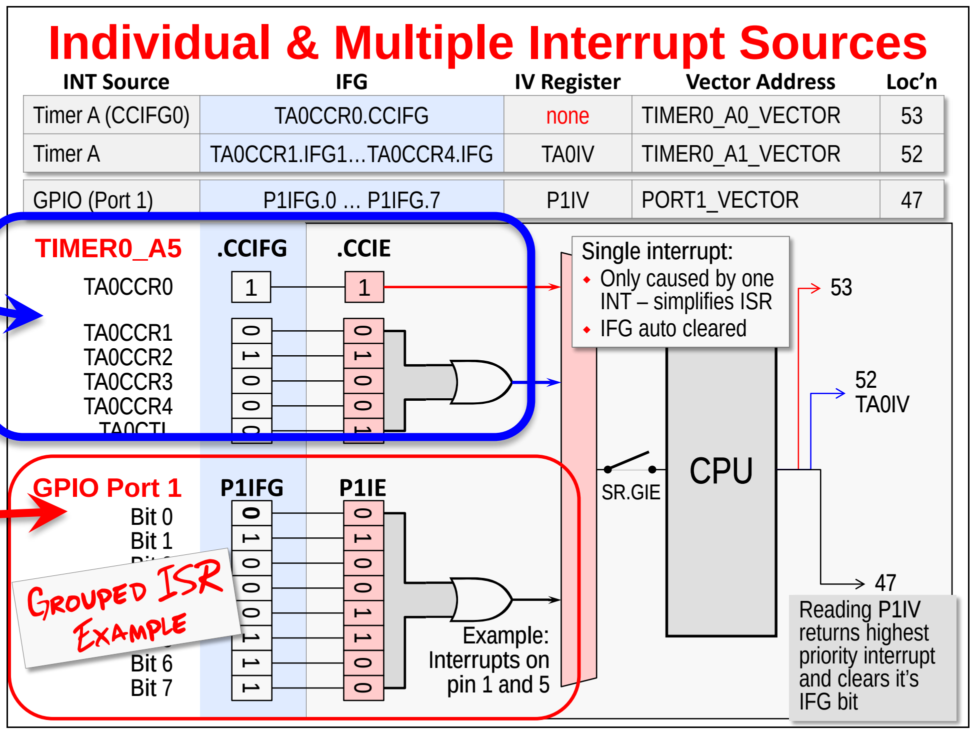
Grouped ISR (Interrupt Service Routine)

Logical Diagram for Grouped Interrupts

Before examining the code for a grouped ISR, let's first examine the grouped interrupt using a logical diagram.

As we briefly mentioned earlier in the chapter (and will discuss in full detail in a later chapter), the Timer_A and Timer_B peripherals are provided with two interrupts. For example, when looking at Timer0_A5, there is a dedicated interrupt for TA0CCR0 (which stands for Timer0_A Capture/Compare Register 0). Notice below how this is routed directly to the GIE input mux.

The remaining five Timer0_A5 interrupts are logically AND'd together; this combination provides a second interrupt signal from Timer0_A5 to the GIE input mux.



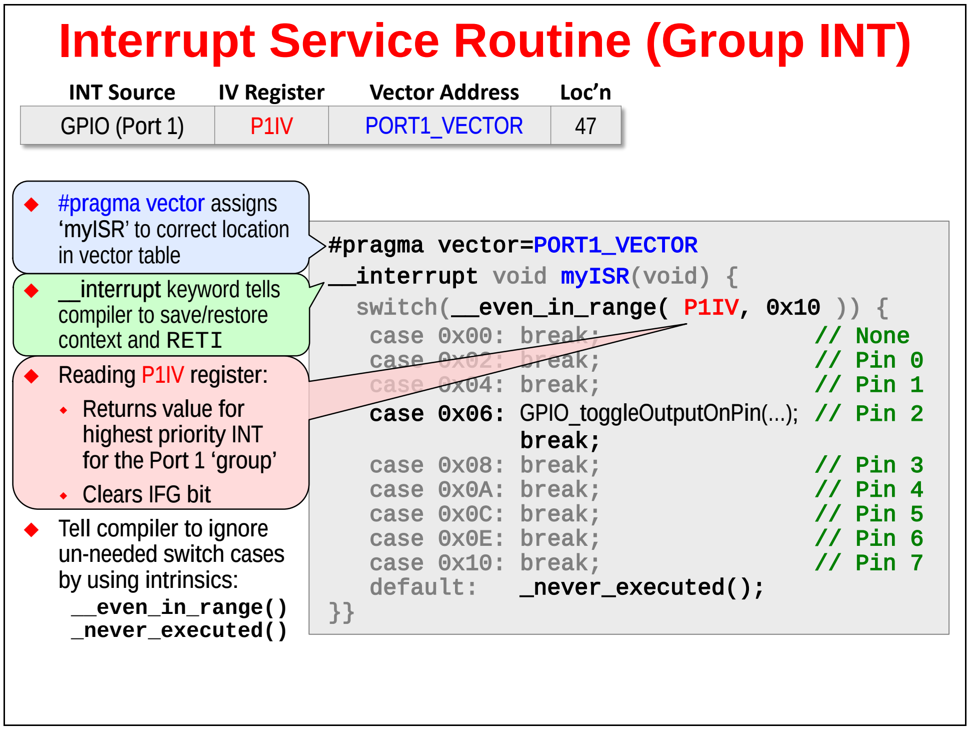
This diagram also shows that all of the input pins for GPIO port 1 (P1) share a single, grouped interrupt. This means your GPIO ISR must always verify which pin actually caused an interrupt whenever the ISR is executed.

The interrupt logic within the CPU recognizes each of these interrupt sources, therefore:

- If the first interrupt (TA0CCR0) occurs, it will cause the code at vector address 53 (TIMER_A0_VECTOR) to be executed.
- Similarly, the remaining Timer0 interrupts are associated with vector 52.
- Finally, the GPIO port (P1) was assigned (by the chip designer) to vector 47.

ISR Example for Grouped Interrupts

The code for a grouped ISR begins similar to any MPS430 interrupt service routine; you should use the #pragma vector and __interrupt keyword syntax.



For grouped interrupts, though, we also need to determine which specific source caused the CPU to be interrupted. As we've described, the Interrupt Vector (IV) register is an easy way to determine the highest-priority, pending interrupt source. In the case of GPIO port 1, we would read the P1IV register.

It's common to see the IV register read within the context of a switch statement. In the above case, if the P1IV register returns "6", it means that pin 2 was our highest-priority, enabled interrupt on Port 1; therefore, its case statement is executed. (Note, the return values for each IV register are detailed in the *F5xx device Users Guide* and the *F5xx DriverLib User's Guide*. You will find similar documentation for all MSP430 devices..)

If our program was using Pin 2 on Port 1, you should see the code for case 0x06 executed if the GPIO interrupt occurs.

By the way, there are two items in the above code example which help the compiler to produce better, more optimized, code. While these intrinsic functions are not specific to interrupt processing, they are useful in creating optimized ISR's.

- The *__even_in_range()* intrinsic function provides the compiler a bounded range to evaluate. In other words, this function tells the compiler to only worry about even results that are lower or equal to 10.
- Likewise the *_never_executed()* intrinsic tells the compiler that, in this case, "default" will never occur.

Enabling Interrupts

Earlier in the chapter we learned that for the CPU to recognize an interrupt two enable bits must be set:

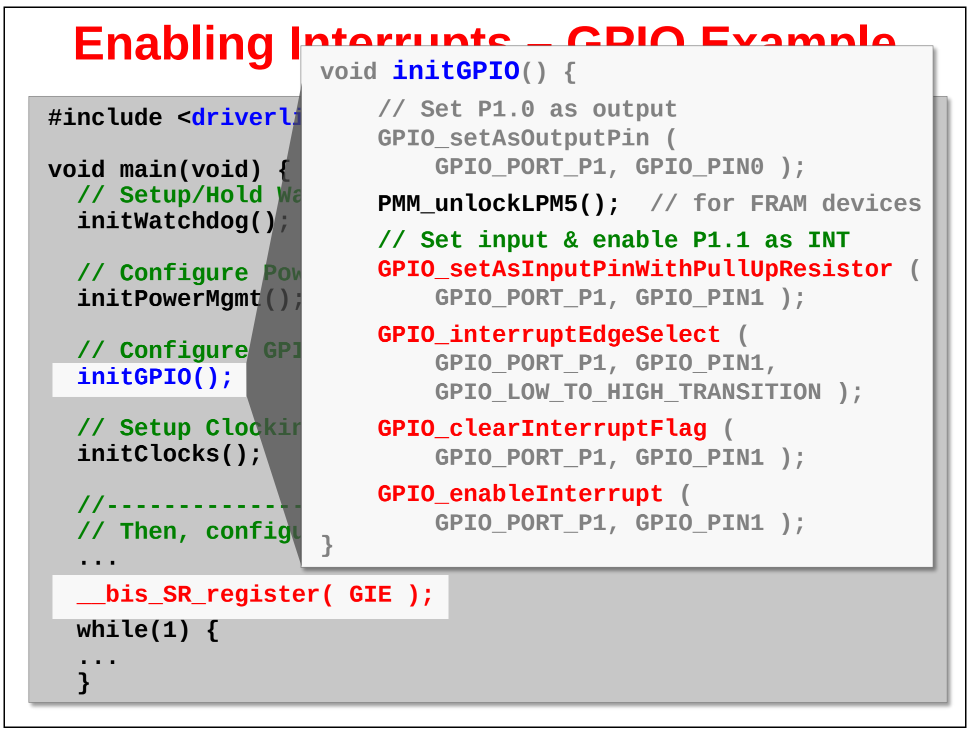
- **Individual Enable** – one IE bit for each interrupt source
- **Global Interrupt Enable** – GIE is a common "master" enable bit for all interrupts (except those defined as non-maskable)

In the example below we show the code required to setup a GPIO pin as an interrupt. We chose to enable the interrupt, as well as configuring the other GPIO pins, in a function called initGPIO(); implementing your code in this way is not required, but it's how we decided to organize our code.

The key DriverLib function which enables the external interrupt is:

```
GPIO_enableInterrupt()
```

You will find that most of the MSP430ware DriverLib interrupt enable functions take a similar form: `<module>_enableInterrupt()`.



Within initGPIO() we highlighted three other related functions in **Red**:

- **GPIO_setAsInputPinWithPullUpResistor()** is required to configure the pin as an input. On the Launchpad, the hardware requires a pull-up resistor to complete the circuit properly. Effectively, this function configures our interrupt "source".
- **GPIO_interruptEdgeSelect()** should be used to configure what edge transition (low-to-high or high-to-low) will trigger an interrupt. This configures bits in the port's IES register – which are left uninitialized after reset.

- **GPIO_clearInterruptFlag()** clears the IFG bit associated with our pin (e.g. P1.1). This is not required but is commonly used right before a call to "enable" an interrupt. You would clear the IFG before setting IE when you want to ignore any prior interrupt event; in other words, clear the flag first if you only care about interrupts that will occur now – or in the future.

Finally, once you have enabled each individual interrupt, the global interrupt needs to be enabled. This can be done in a variety of ways. The two most common methods utilize compiler intrinsic functions:

- **__bis_SR_register(GIE)** instructs the compiler to set the GIE bit in the Status Register
  - bis = bit set
  - SR = Status Register
  - GIE = which bit to set in the SR
- **__enable_interrupts(void)** tells the compiler to enable interrupts. The compiler uses the EINT assembly instruction which pokes 1 into the GIE bit.

Sidebar – Where in your code should you enable GIE?

The short answer, "Whenever you need to turn on interrupts".

A better answer, as seen in our code example, is "right before the while{} loop".

Conceptually, the main() function for most embedded systems consists of two parts:

- Setup
- Loop

That is, the first part of the main() function is where we tend to setup our I/O, peripherals, and other system hardware. In our example, we setup the watchdog timer, power management, GPIO, and finally the system clocks.

The second part of main() usually involves an infinite loop – in our example, we coded this with an endless while{} loop. An infinite loop is found in almost all embedded systems since we want to run forever after the power is turned on.

The most common place to enable interrupts globally (i.e. setting GIE) is right between these two parts of main(). Looking at the previous code example, this is right where we placed our function that sets GIE.

As a product example, think of the A/C power adaptor you use to charge your computer; most of these, today, utilize an inexpensive microcontroller to manage them. (In fact, the MSP430 is very popular for this type of application.) When you plug in your power adapter, we're guessing that you would like it to run as long as it's plugged in. In fact, this is what happens; once plugged in, the first part of main() sets up the required hardware and then enters an endless loop which controls the adaptor. What makes the MSP430 such a good fit for this application is: (1) it's inexpensive; and (2) when a load is not present and nothing needs to be charged, it can turn off the external charging components and put itself to sleep – until a load is inserted and wakes the processor back up.

Miscellaneous Topics

Handling Unused Interrupts

While you are not required to provide interrupt vectors – or ISR's – for every CPU interrupt, it's considered good programming practice to do so. To this end, the MSP430 compiler issues a warning whenever there are "unhandled" interrupts.

The following code is an example that you can include in all your projects. Then, as you implement an interrupt and write an ISR for it, just comment the associated #pragma line from this file.

Handling Unused Interrupts

- The MSP430 compiler issues warning whenever all interrupts are not handled (i.e. when you don't have a vector specified for each interrupt)
- Here's a simple example of how this might be handled:

```
// Example for UNUSED_HWI_ISR()

#pragma vector=ADC12_VECTOR
#pragma vector=COMP_B_VECTOR
#pragma vector=DMA_VECTOR
#pragma vector=PORT1_VECTOR
 ...
#pragma vector=TIMER1_A1_VECTOR
#pragma vector=TIMER2_A0_VECTOR
#pragma vector=TIMER2_A1_VECTOR
#pragma vector=UNMI_VECTOR
#pragma vector=USB_UBM_VECTOR
#pragma vector=WDT_VECTOR
__interrupt void UNUSED_HWI_ISR (void)
{
  __no_operation();
}
```

Note: The TI code generation tools distinguish between "warnings" and "errors". Both represent issues found during compilation and build, but whereas a *warning* is issued and code building continues … when an *error* is encountered, an *error* statement is issued and the tools stop before creating a final executable.

Interrupt Service Routines – Coding Suggestions

Listed below are a number of required and/or good coding practices to keep in mind when writing hardware interrupt service routines. Many of these have been discussed elsewhere in this chapter.

Hardware ISR's – Coding Practices

- An interrupt routine must be declared with no arguments and must return void
  - Global variables are often used to "pass" information to or from an ISR
- Do not call interrupt handling functions directly (Rather, write to IFG bit)
- Interrupts can be handled directly with C/C++ functions using the *interrupt* keyword or pragma
  ... Conversely, the TI-RTOS kernel easily manages *Hwi* context
- Calling functions in an ISR
  - If a C/C++ interrupt routine doesn't call other functions, usually, only those registers that the interrupt handler uses are saved and restored.
  - However, if a C/C++ interrupt routine does call other functions, the routine saves all the save-on-call registers if any other functions are called
  - Why? The compiler doesn't know what registers could be used by a nested function. It's safer for the compiler to go ahead and save them all.
- Re-enable interrupts? (Nesting ISR's)
  - DON'T – it's not recommended – better that ISR's are "lean & mean"
  - If you do, change IE masking before re-enabling interrupts
  - Disable interrupts before restoring context and returning (RETI re-enables int's)
- Beware – Only You Can Prevent Reentrancy...

We wrote the last bullet, regarding reentrancy, in a humorous fashion. That said, it speaks to an important point. If you decide to enable interrupt nesting, you need to be careful that you either prevent reentrancy - or that your code is capable of reentrancy.

Wikipedia defines reentrancy as:

> In *computing*, a *computer program* or *subroutine* is called **reentrant** if it can be interrupted in the middle of its execution and then safely called again ("re-entered") before its previous invocations complete execution.

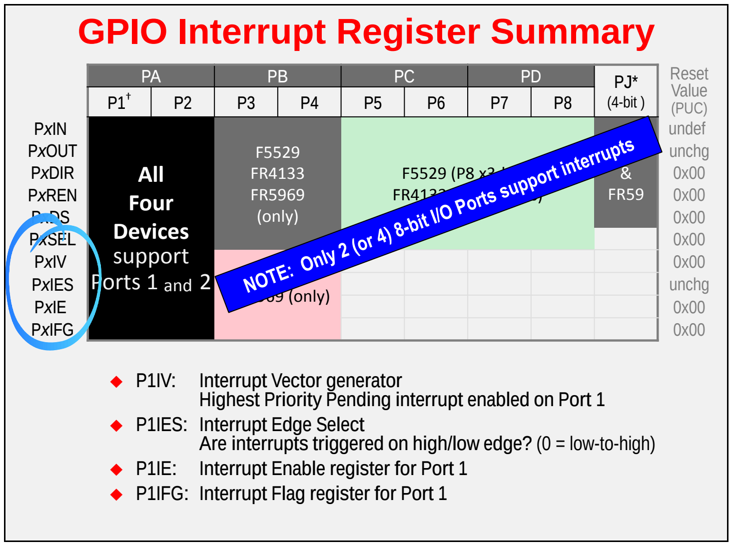
This type of program/system error can be very difficult to debug (i.e. find and fix). This is especially true if you call functions within your interrupt service routines. For example, the C language's malloc() function is not reentrant. If you were to call this function from an ISR and it was interrupted, and then it is called again by another ISR, your system would most likely fail – and fail in a way that might be very difficult to detect.

So, we stated this humorously, but it is very true. We recommend that:

- You shouldn't nest interrupts
- If you do, verify the code in your ISR is reentrant
- Never call malloc() – or similar functions - from inside an ISR

GPIO Interrupt Summary

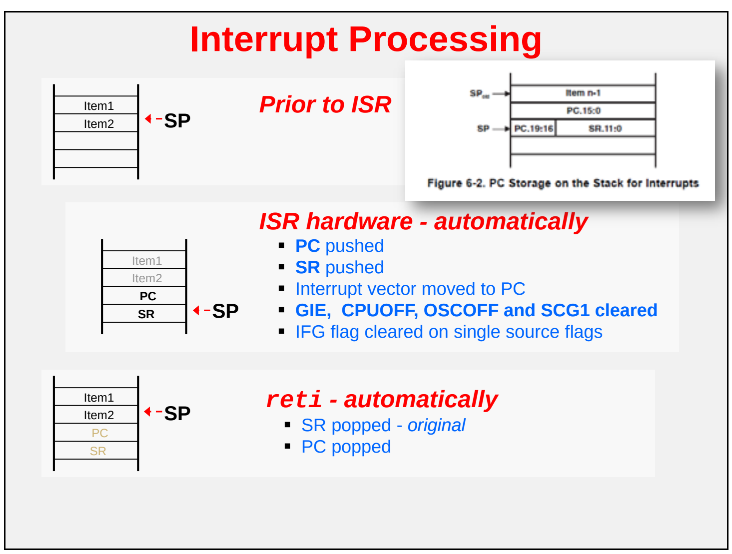
The diagram used to summarize the GPIO control registers in a previous chapter is a good way to visualize the GPIO interrupt capabilities of our devices. From the diagram below we can see that most MSP430 processors allow ports P1 and P2 to be used as external interrupt sources; we see this from the fact that these ports actually have the required port interrupt registers.



There are other devices in the MSP430 family that support interrupts on more than 2 ports, but of the three example processors we use throughout this course, only the FR5969 (FRAM) devices support interrupt inputs on additional ports (P3 and P4).

Interrupt Processing Flow

The following information was previously covered in this chapter, but since the slide is a good summary of the interrupt processing flow, we have included it anyway.



Interrupts and TI-RTOS Scheduling

When embedded systems start to become more complex – that is, when you need to juggle more than a handful of events – using a Real-Time Operating System (RTOS) can greatly increase your system's reliability … while decreasing your time-to-market, frustration and costs.

The Texas Instruments RTOS (TI-RTOS) – also known as SYS/BIOS – provides many functions that you can use within your program; for example, the TI-RTOS kernel includes: *Scheduling*, *Instrumentation*, and *Memory Management*. You can choose which parts of TI-RTOS are needed and discard the rest (to saves memory).

Think of TI-RTOS as a library and toolset to help you build and maintain robust systems. If you're doing just "one" thing, it's probably overkill. As you end up implementing more and more functionality in your system, though, the tools and code will save you time and headaches.

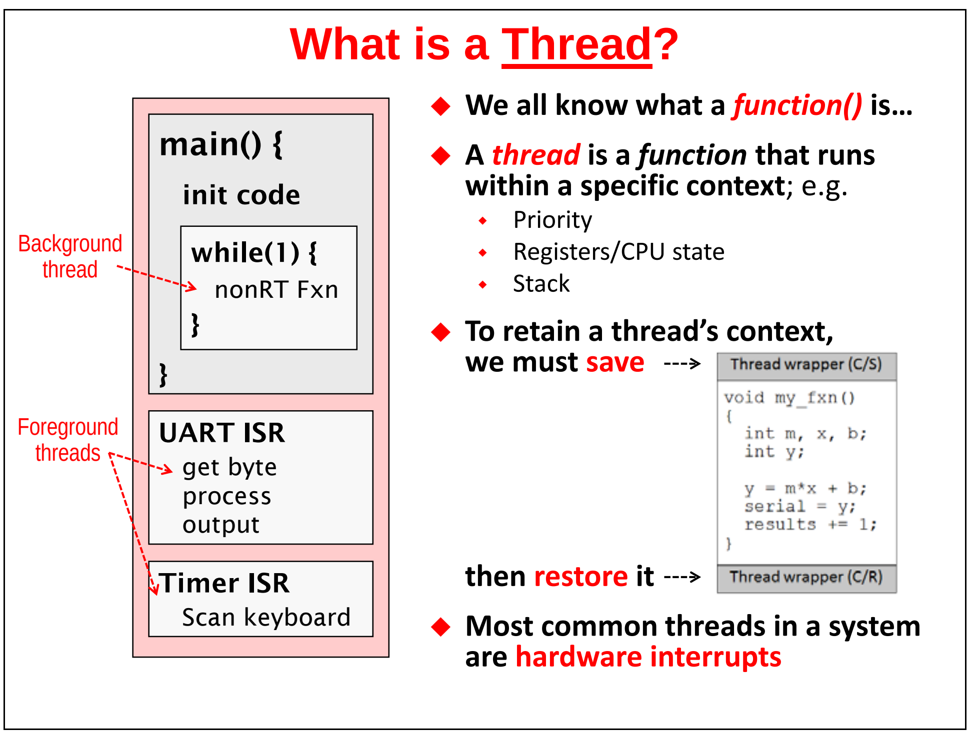
The only part of TI-RTOS discussed in this chapter is "Scheduling". We talk about this because it is very much related to the topics covered throughout this chapter – interrupts and threads. In many cases, if you're using an RTOS, it will manage much of the interrupt processing for you; it will also provide additional options for handling interrupts – such as post-processing of interrupts.

As a final note, we will only touch on the topics of scheduling and RTOS's. TI provides a 2-day workshop where you can learn all the details of the TI-RTOS kernel. You can view a video version of the TI-RTOS course or take one live. Please check out the following wiki page for more information:

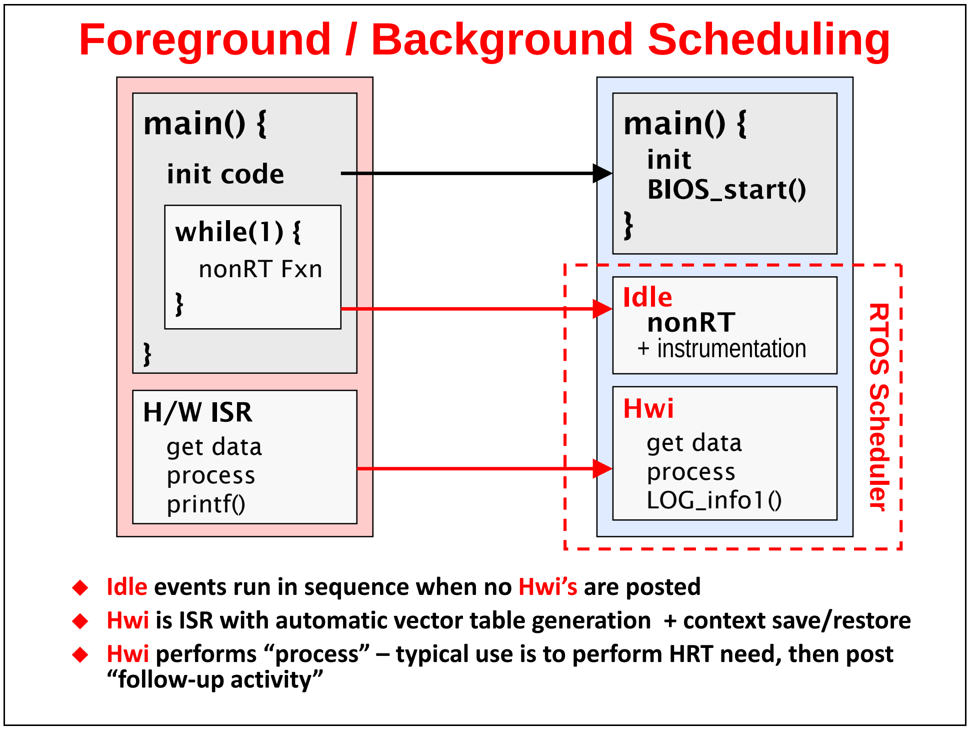
http://processors.wiki.ti.com/index.php/Introduction_to_the_TI-RTOS_Kernel_Workshop

Threads – Foreground and Background

Our quick introduction to TI-RTOS begins with a summary of threads. While we discussed these concepts earlier in the chapter, they are very important to how a RTOS scheduler works.



We also discussed the idea of foreground and background threads as part of the interrupts chapter. In the case shown below (on the left), the endless loop in main() will run forever and be pre-empted by higher-priority hardware interrupts.



TI-RTOS utilizes these same concepts … only the names and threads change a little bit.

Rather than main() containing both the *setup* and *loop* code as described earlier, TI-RTOS creates an ***Idle*** thread that operates in place of the while{} loop found previously in main(). In other words, rather than adding your functions to a while{} loop, TI-RTOS has you add them to ***Idle***. (TI-RTOS includes a GUI configuration tool that makes this very easy to do.)

Since interrupts are part of the MSP430's hardware, they essentially work the same way when using TI-RTOS. What changes when using RTOS are:

- TI-RTOS calls them ***Hwi*** threads … for Hardware Interrupts
- Much of the coding effort is handled automatically for you by TI-RTOS (very nice)

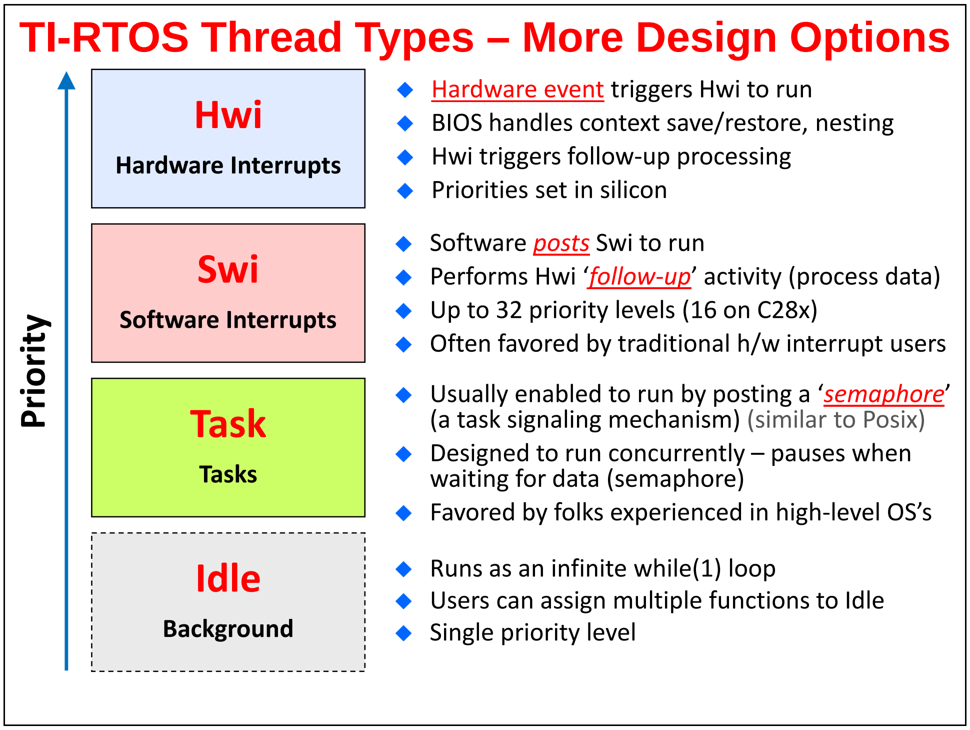
Don't worry, though, you're not locked into anything. You can mix-and-match how you handle interrupts. Let TI-RTOS manage some of your interrupts while handling others in your own code, just as we described in this chapter.

Hint: When using TI-RTOS, the author prefers to let it manage all of the interrupts because it's easier that way. Only

Only in a rare case – like to save a few CPU cycles – would there be a need to managed an interrupt outside of TI-RTOS. Thusfar, the only reason I've actually done this is to provde that it works.

TI-RTOS Thread Types

We already described two types of threads: ***Hwi*** and ***Idle***. Using these two threads is very similar to what we described throughout this chapter.



TI-RTOS provides two additional thread types: ***Software Interrupts*** (***Swi***) and ***Tasks*** (***Task***). As you can see above, these thread types fall between ***Hwi*** and ***Idle*** in terms of priority.

Each of these threads can be used to extend your system's processing organization.

What do we mean by this?

You might remember that we HIGHLY recommended that you DO NOT nest interrupts. But what happens if you want to run an algorithm based on some interrupt event? For example, you want to run a filter whenever you receive a value from an A/D converter or from the serial port.

Without an RTOS, you would need to organize your main while{} loop to handle all of these interrupt, follow-up tasks. This is not a problem for one or two events; but for lots of events, this can become very complicated – especially when they all run at different rates. This way of scheduling your processing is called a *SuperLoop*.

With an RTOS, we can post follow-up activity to a Swi or Task. A ***Swi*** acts just like a software triggered interrupt service routine. ***Tasks***, on the other hand, run all the time (*have you heard the term multi-tasking before?*) and utilize ***Semaphores*** to signal when to run or when to *block* (i.e. pause).

In other words, ***Swi's*** and ***Task's*** provide two different ways to schedule follow-up processing code. They let us keep our hardware interrupts (Hwi's) very short and simple – for example, all we need to do is read our ADC and then post an associated ***Swi*** to run.

If all of this sounds complicated, it really isn't. While outside the scope of this course, the TI-RTOS course will have you up-and-running in no time. Once you experience the effective organization provided by an RTOS, you may never build another system without one.

TI-RTOS Details

The following slide provides some "characteristics" of the TI-RTOS kernel. The bottom-line here is that it is a priority-based scheduler. The highest priority thread gets to run, period. (Remember, hardware interrupts are always the highest priority.)

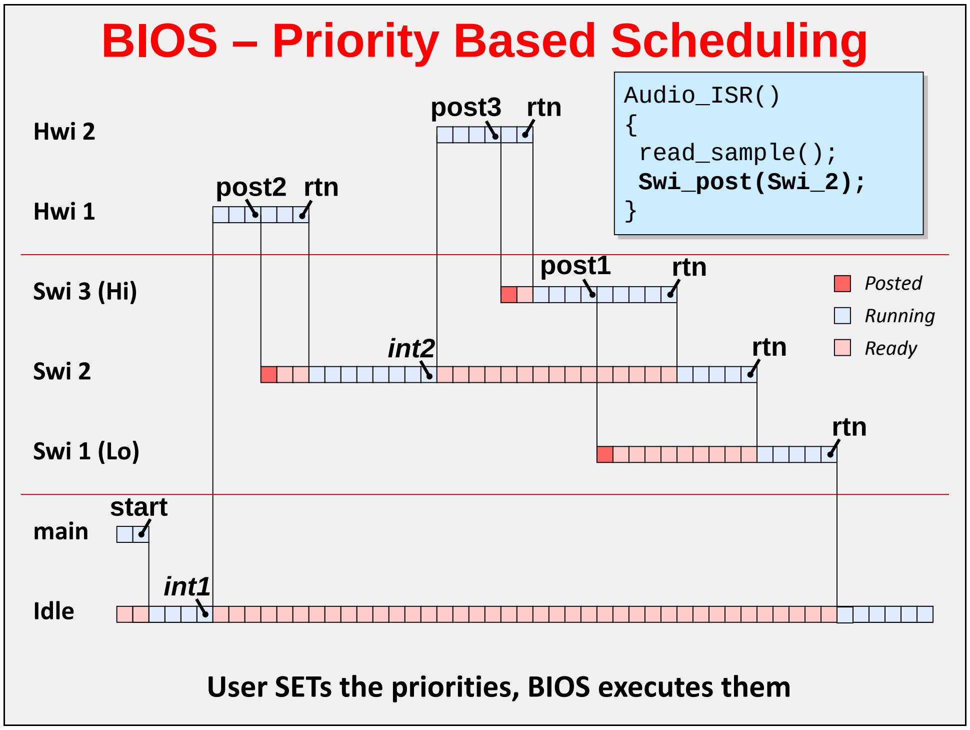
TI-RTOS Kernel – Characteristics

- RTOS means "Real-time O/S" – so the intent of this O/S is to provide common services to the user WITHOUT disturbing the real-time nature of the system
- The TI-RTOS Kernel (SYS/BIOS) is a PRE-EMPTIVE scheduler. This means the highest priority thread ALWAYS RUNS FIRST. Time-slicing is not inherently supported.
- The kernel is EVENT-DRIVEN. Any kernel-configured interrupts or user calls to APIs such as `Swi_post()` will invoke the scheduler. The kernel is NOT time-sliced although threads can be triggered on a time bases if so desired.
- The kernel is OBJECT BASED. All APIs (methods) operate on self-contained objects. Therefore when you change ONE object, all other objects are unaffected.
- Being object-based allows most RTOS kernel calls to be DETERMINISTIC. The scheduler works by updating event queues such that all context switches take the same number of cycles.
- Real-time Analysis APIs (such as Logs) are small and fast – the intent is to LEAVE them in the program – even for production code – yes, they are really that small

While you can construct a time-slicing system using TI-RTOS, this is not commonly done. While time-slicing can be a very effective technique in host operating systems (like Windows or Linux), it is not a common method for scheduling threads in an embedded system.

Hwi – Swi – Idle Scheduling

Here's a simple, visual example of what real-time scheduling might look like in an RTOS based system.



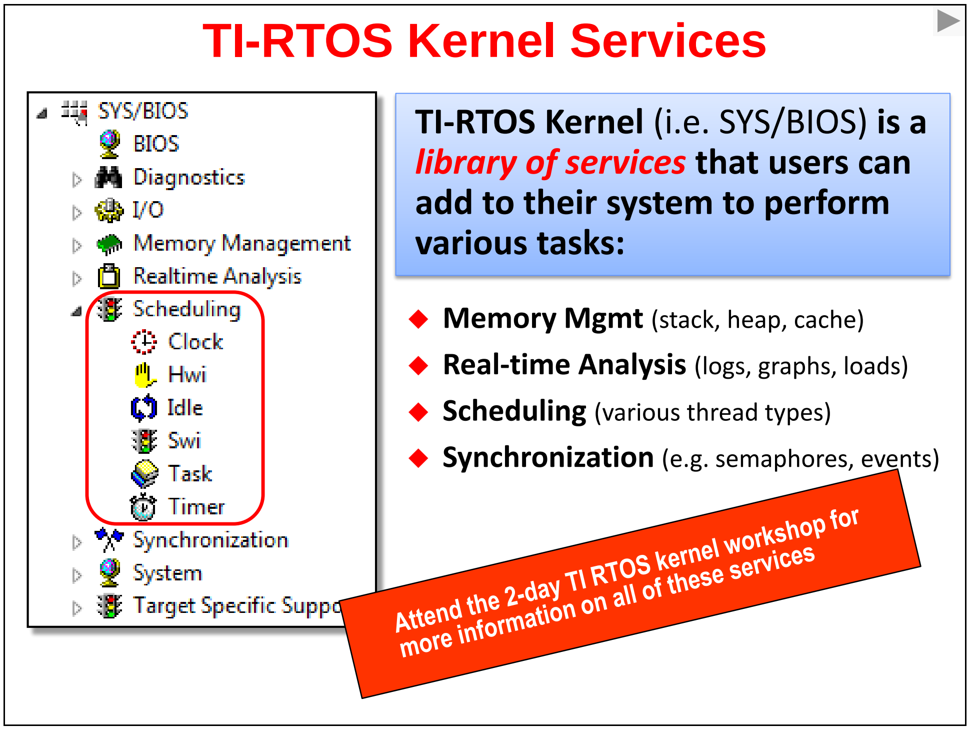
Notice how the system enters Idle from main(). Idle is always ready to run (just as our old while{} loop was always ready to run).

When a hardware interrupt (Hwi) occurs, we leave Idle and execute the Hwi thread's code. Since it appears the Hwi posted a Swi, that's where the TI-RTOS scheduler goes to once the Hwi finishes.

We won't go through the remaining details in this course, though we suspect that you can all follow the diagram. For this slide, and a lot more information, please refer to the TI-RTOS Kernel Workshop.

Summary: TI-RTOS Kernel

The following slide summarizes much of the functionality found in the TI-RTOS kernel. In this chapter we've only touched on the scheduling features.



The TI-RTOS product includes the kernel, shown above, along with a number of additional drivers and stacks. Oh, and the kernel comes with complete source code – nothing is hidden from you.

For many, though, one of the compelling features of TI-RTOS is that it's FREE*.

Remember, we make our money selling you devices. Our code and tools are there to help you get your programs put together – and your systems to market – more quickly.

* That is, it's free for use on all Texas Instruments processors.

Lab 5 – Interrupts

This lab introduces you to programming MSP430 interrupts. Using interrupts is generally one of the core skills required when buiding 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

Lab 5a covers all the essential details of interrupts:

- Setup the interrupt vector
- Enable interrupts
- Create an ISR

When complete, you should be able to push the SW1 button and toggle the Red LED on/off.

Lab 5b is listed as optional since, while these skills are valuable, you should know enough at the end of Lab 5a to move on and complete the other labs in the workshop.

Lab Topics

Interrupts **5-36**

Lab 5 – Interrupts *5-37*
Lab 5 Worksheet 5-39
General Interrupt Questions 5-39
Interrupt Flow 5-40
Setting up GPIO Port Interrupts 5-40
Interrupt Priorities & Vectors 5-41
ISR's for Group Interrupts 5-42
Lab 5a – Push Your Button 5-44
File Management 5-44
Configure/Enable GPIO Interrupt … Then Verify it Works 5-47
Add a Simple Interrupt Service Routine (ISR) 5-50
Sidebar – Vector Error 5-50
Upgrade Your Interrupt Service Routine (ISR) 5-52
(Optional) Lab 5b – Can You Make a Watchdog Blink? 5-53
Import and Explore the WDT_A Interval Timer Example 5-53
Run the code 5-55
Change the LED blink rate 5-55

Appendix *5-56*

Lab 5 Worksheet

General Interrupt Questions

Hint: You can look in the Chapter 5 discussion for the answers to these 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? That is, 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.)*
   Hint: Look at the chapter discussion, datasheet or User's Guide for this answer.

   TIMER_A ______________________________

   ____

   ____

   ____

6. What signifies that an interrupt has occurred?
   Hint: Look at the "Interrupt Flow" part of this chapter discussion.

   A __________ bit is set

   What's the acronym for these types of 'bits" __________

Setting up GPIO Port Interrupts

Next, let's review the code required to setup one of the Launchpad buttons for GPIO input.
(Hint: Look in the Chapter 5 "Enabling Interrupts" discussion for help on the next two questions.)

7. Write the code to enable a GPIO interrupt for the listed Port.Pin?

```
// GPIO pin to use:   F5529 = P1.1,   FR4133 = P1.2,  FR5969 = P1.1

_____________________________________________ // setup pin as input

_____________________________________________ // set edge select

_____________________________________________ // 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 sidebar discussion where we showed how to do this.)

   ____

Interrupt Priorities & Vectors

9. Check the interrupt that has higher priority. (*Hint: Look at the chapter discussion or device datasheet for the answer.)*

- ❑ GPIO Port 2
- ❑ WDT Interval Timer

10. Where do you find the name of an “interrupt vector” (e.g. PORT1_VECTOR)?

Hint: Which header file defines symbols for each device?

____

11. Write the code to set the interrupt vector? *(To help, we’ve provided a simple ISR to go with the line of code we’re asking you to complete. Finish the #pragma statement...)*

```
// Put’s the ISR function’s address into the Port 1 vector location

#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?

____

____

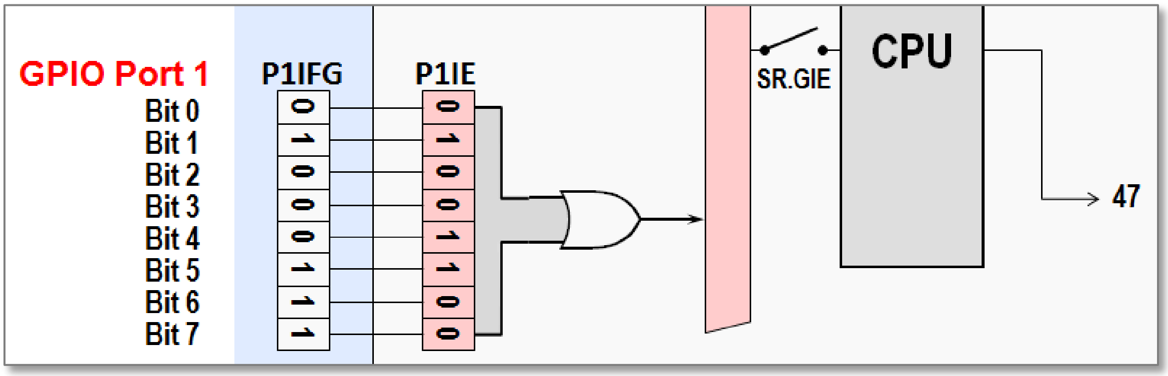
12. How do you pass a value into (or out from) and interrupt service routine (ISR)?

Hint: Look at the chapter topic “Interrupt Service Routines – Coding Suggestions”.

____

ISR's for Group Interrupts

As we learned earlier, most MSP430 interrupts are grouped. For example, the GPIO port interrupts are all grouped together. *(Hint: To answer these last two questions, look at the discussion titled "Grouped ISR" in this chapter's discussion.)*



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 appropriate blank line to respond to the Port 1 pin used for the pushbutton on your Launchpad. (F5529/FR5969 = P1.1; FR4133 = P1.2)
    - Add the code needed to toggle the LED (on `P1.0`) in response to the button interrupt.

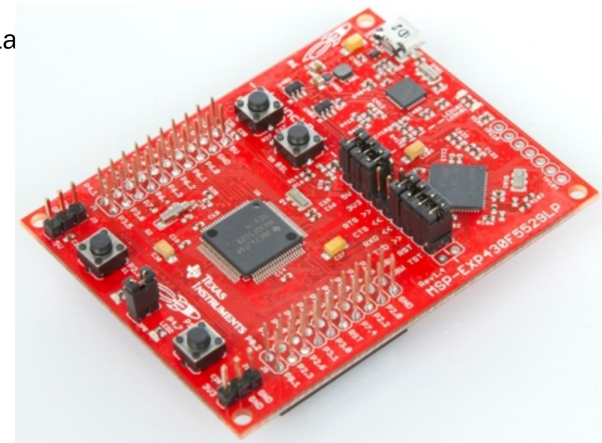
```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {
   switch(__even_in_range( ____________________, 0x10 )){
        case 0x00: break;          // None
        case 0x02: break;          // Pin 0

             ____________________________________________
             break;
        case 0x04:                 // Pin 1

             ____________________________________________
             break;
        case 0x06:                 // Pin 2

             ____________________________________________
             break;
        case 0x08:                 // Pin 3

             ____________________________________________
             break;
        case 0x0A:                 // Pin 4

             ____________________________________________
             break;
        case 0x0C:                 // Pin 5

             ____________________________________________
             break;
        case 0x0E:                 // Pin 6

             ____________________________________________
             break;
        case 0x10:                 // 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 La
code and add the following.

- Setup the interrupt vector
- Enable interrupts
- Create an ISR



Launchpad	Pin	Button
F5529	P1.1	S2
FR5969	P1.1	S2
FR4133	P2.2	S1

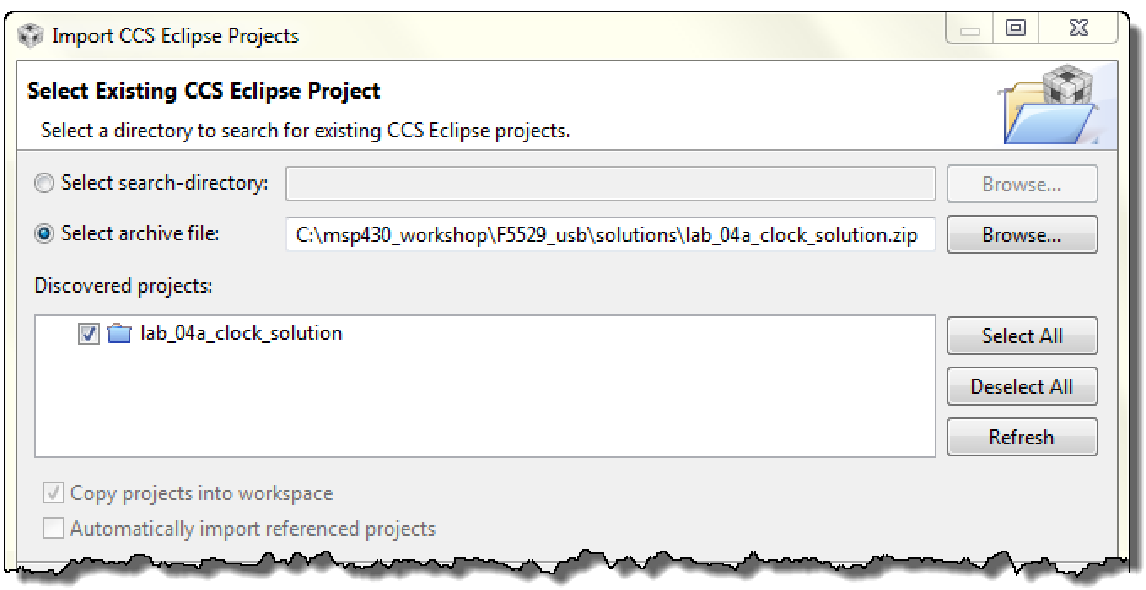
File Management

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 CCS Projects…
```

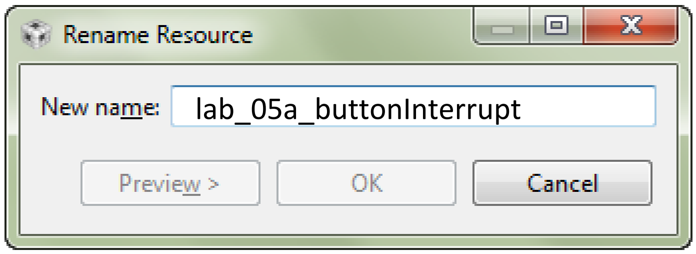


3. **Rename the imported project to: `lab_05a_buttonInterrupt`**

   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 F2
   ```

   When the following dialog pops up, fill in the new project name:

4. **Verify the project is active, then check that it 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.

   When complete, **terminate** the debugger.

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
   ```

   ```
   "Copy" the file into your project
   ```

   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 … comment out the old code from while{} loop.

Open `main.c` and comment out the code in the while{} loop. This is the old code that flashes the LED using the inefficient __delay_cycles() function.

The easiest way to do this is to:

`Select all the code in the while{} loop`

Ctrl - / (This toggles the line comments on/off)

Once commented, the loop should look similar to that below:

```
    while(1) {
//        // Turn on LED
//        GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
//
//        // Wait about a second
//        _delay_cycles( HALF_SECOND );
//
//        // Turn off LED
//        GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
//
//        // Wait another second
//        _delay_cycles( HALF_SECOND );
    }
}
```

After commenting out the while code, just double-check for errors by clicking the build button. (Fix any error that pops up.)

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 # 7 (see page 5-40).

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

– For the 'F5529 Launchpad, we're using pushbutton S2 (P1.1)

– For the 'FR5969 Launchpad, we're using pushbutton S2 (P1.1)

– For the 'FR4311 Launchpad, we're using pushbutton S1 (P1.2)

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 # 8 (see page 5-40).

9. Build your code.

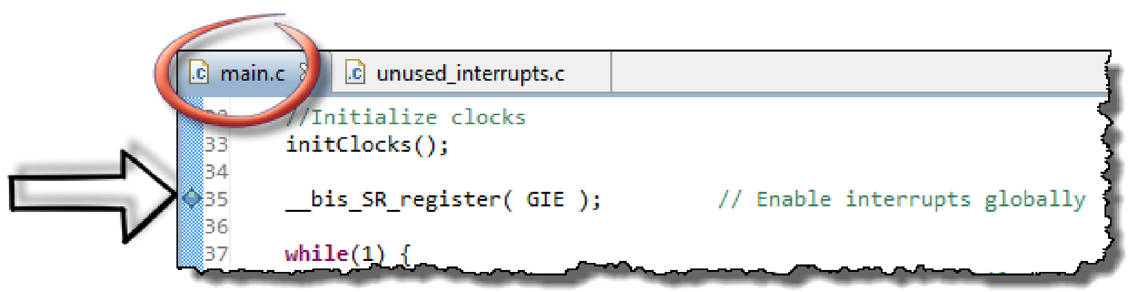
Fix any typos or errors.

Start the Debugger and Set Breakpoints

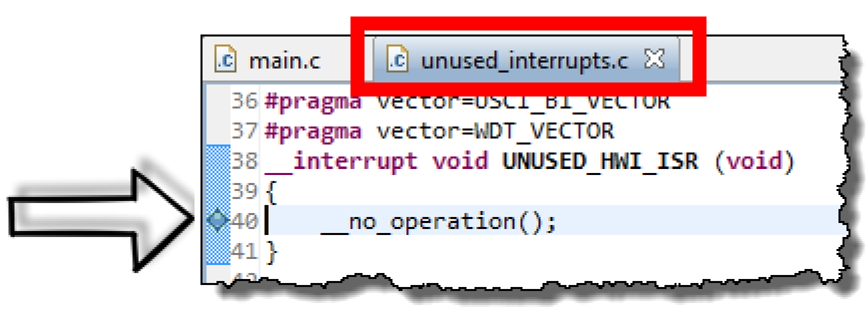
Once the debugger opens, we'll setup two breakpoints. This allows us to verify the interrupts were enabled, as well as trapping the interrupt when it occurs.

10. Launch the debugger.

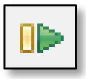
11. Set a breakpoint on the "enable GIE" line of code in `main.c`.



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



Run Code to Verify Interrupts are Enabled



13. Click Resume … 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:

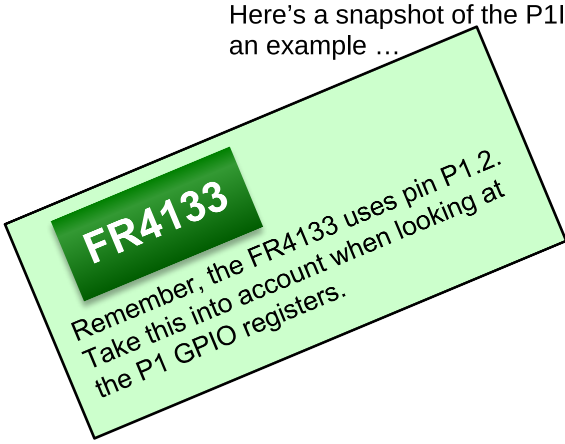
`View → Registers`

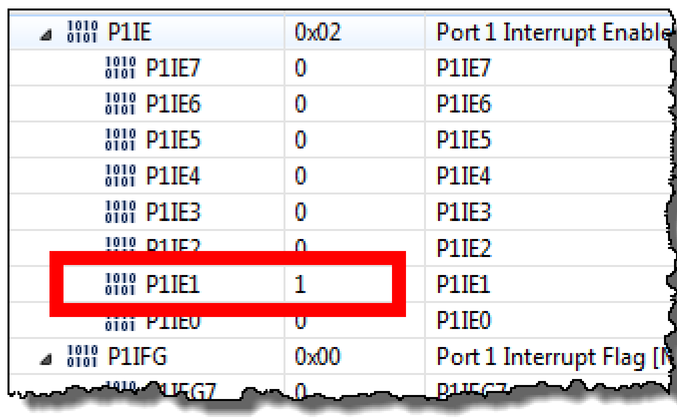
15. Verify Port1 bits: DIR, OUT, REN, IE, IFG.

The first breakpoint halts the processor right before setting the GIE bit. Before turning on the interrupts, let's 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)
- `P1IES.1 = 0` (configured to generate an interrupt on a low-to-high transition)
- `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 …

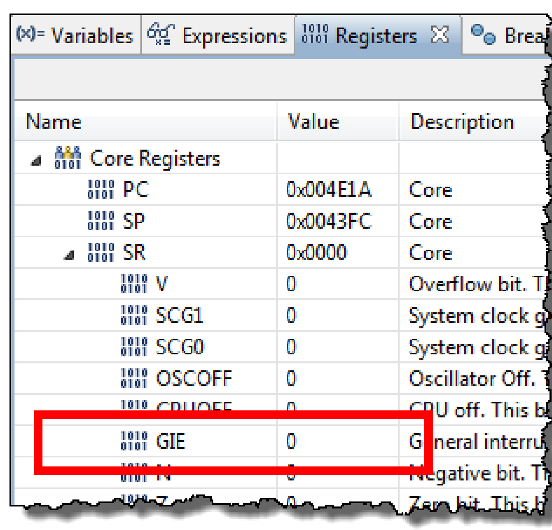




16. Next, let's look at the *Status Register* (SR).

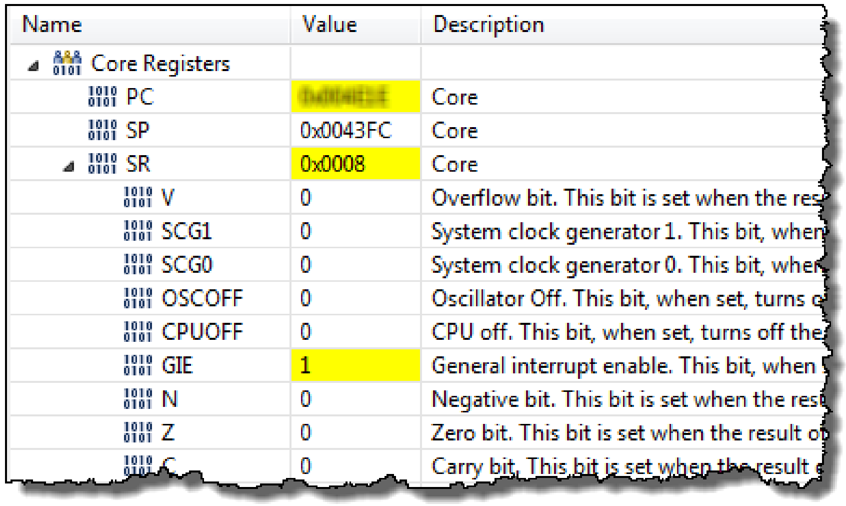
You can find it under the *Core Registers* at the top of the *Registers* window.

You should notice that the GIE bit equals 0, since we haven't executed the line of code enabling interrupts globally, yet.



17. Single-step the processor (i.e. Step-Over) and watch GIE change.

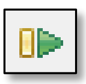
Click the toolbar button or tap the F6 key. Either way, the *Registers* window should update:



Name	Value	Description
Core Registers		
PC	[illegible]	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, whe
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 resu
Z	0	Zero bit. This bit is set when the result o
C	0	Carry bit. This bit is set when the result

Testing your Interrupt

With everything set up properly, let's try out our code.



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 program counter is sitting in the while{} loop, as expected.

19. Press the appropriate pushbutton 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)

Thus far we have used the HWI_UNUSED_ISR. We will now add an ISR specifically for our push-button's GPIO interrupt.

20. Add the Port 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 5-41).

21. Build the program to test for any errors.

You should have gotten the error …

```
"./driverlib/MSP430F5xx_6xx/adc10_a.obj" "./unused_interrupts.obj" "./myClocks.obj" "./main.obj" "../lnk_msp430f5529
error #10056: symbol "__TI_int47" redefined: first defined in "./unused_interrupts.obj"; redefined in "./main.obj"
error #10010: errors encountered during linking; "lab_05a_buttonInterrupt.out" not built
<Linking>
gmake: *** [lab_05a_buttonInterrupt.out] Error 1
gmake: Target `all' not remade because of errors.

>> Compilation failure
```

This error tells us that the linker cannot fit the PORT1_VECTOR into memory because the interrupt vector is defined twice. (INT47 on the 'F5529 and 'FR4133; INT39 on the 'FR5969)

We just created one of these vectors, where is the other one coming from?

____

Sidebar – Vector Error

First, how did we recognize this error?

1. It says, *"errors encountered during linking"*. This tells us the complilation 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 offending files were `.obj`, but these were directly created from their `.c` source 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`.

```
#pragma vector=COMP_B_VECTOR
#pragma vector=DMA_VECTOR
//#pragma vector=PORT1_VECTOR
#pragma vector=PORT2_VECTOR
#pragma vector=RTC_VECTOR
```

23. Try building it again

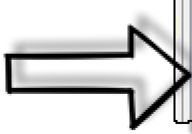
It should work this time… *our fingers are crossed for you.*

24. Launch the debugger.

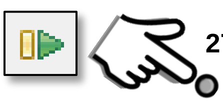
25. Remove all breakpoints.

```
View → Breakpoints
Click the Remove All button
```

26. Set a breakpoint inside your new ISR.

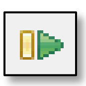


```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void)
{
    // Toggle the LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```



27. Run your code … once the code is running, push the button to generate an interrupt.

The processor should stop at your ISR (location shown above). Breakpoints like this can make it easier to see that we reached the interrupt. (A good debugging trick.)



28. *Resuming* once again, at this point inside the ISR should toggle-on the LED.

If it works, call out "Hooray!"

29. Push the button again.

Hmmm… did you get another interrupt? We didn't appear to.

We didn't see the light toggle-off – and we didn't stop at the breakpoint inside the ISR.

Some of you may have already known this was going to happen. If you're still unsure, go back to Step #0 from our worksheet (page 5-43). We discussed it there.

Upgrade Your Interrupt Service Routine (ISR)

If you hadn't already guessed what the problem was, we can deduce that 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; and, it clears the correct IFG bit for us.

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 5-43).

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. *(Note: this is the code for the 'F5529 and 'FR5969. Remember that the 'FR4133 uses a different pin on Port 1.)*

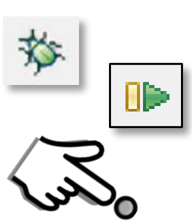
```
//*********************************************************************
// Interrupt Service Routines
//*********************************************************************
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {
        switch(__even_in_range( ????, 0x10 )) {
           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 way to correct this using (<ctrl>-`I`).

`Select the 'ugly' code and press` Ctrl - 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/Resume. 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 then you'll need to click *Resume*).

You're welcome to explore 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 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 grouped interrupts that shared one vector.)

Import and Explore the WDT_A Interval Timer Example

1. Import the *wdt_a_ex2_intervalACLK* project from the MSP430 DriverLib examples.

We're going to "cheat" and use the example provided with MSP430ware to get the WDT_A timer up and running.

As we discussed in Chapter 3, there are two ways we can import an example project:

– Use the Project→Import CCS Projects (as we've done before)

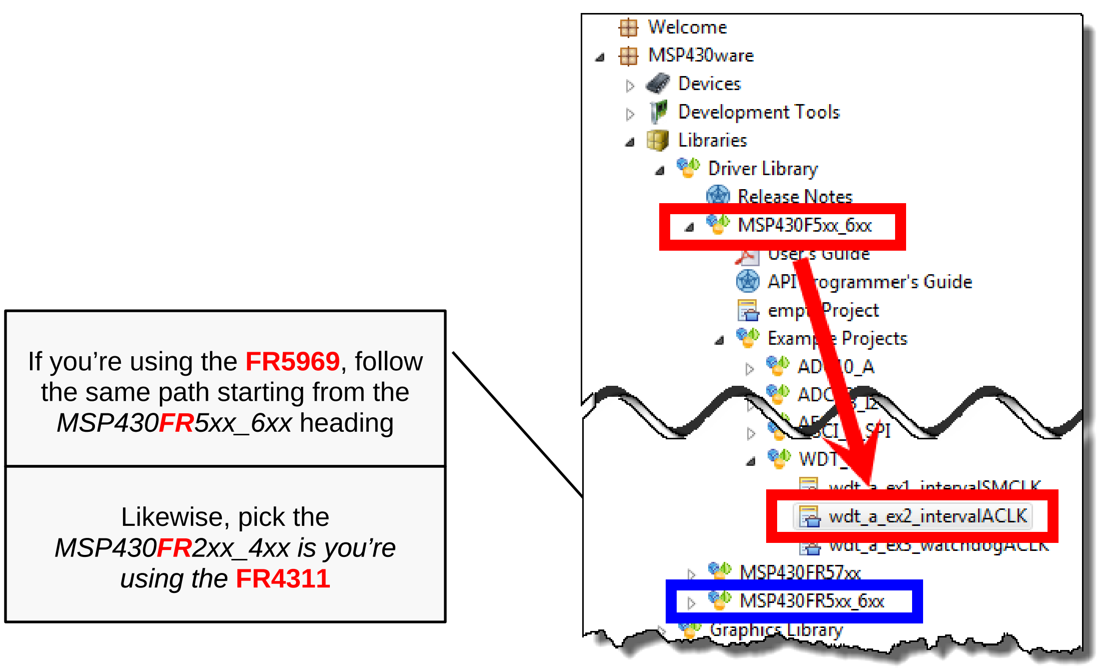
– Utilize the TI Resource Explorer (which is what we'll do again)

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

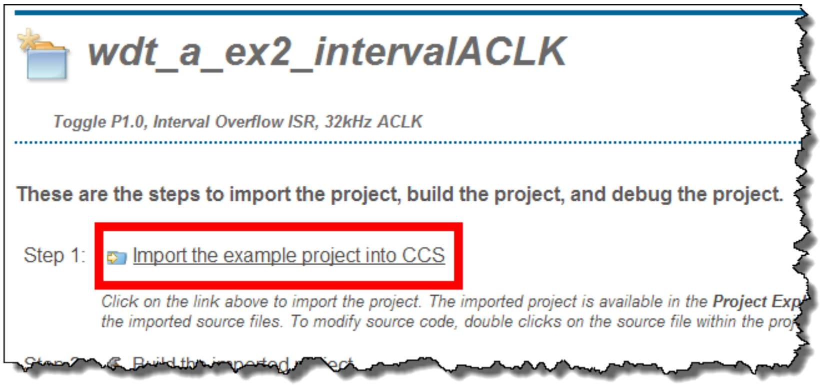
```
View → Resource Explorer (Examples)
```

b) Locate the *wdt_a_ex2_intervalACLK* example for your processor.

Look for it as shown here under: Example Projects → WDT_A



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:**

```
main(void)

    //Initialize WDT module in timer interval mode,
    //with ACLK as source at an interval of 250 ms.
    WDT_A_intervalTimerInit(WDT_A_BASE,
                            WDT_A_CLOCKSOURCE_ACLK,
                            WDT_A_CLOCKDIVIDER_8192);

    WDT_A_start(WDT_A_BASE);

    //Enable Watchdog Interupt
    SFR_clearInterrupt(SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT);
    SFR_enableInterrupt(SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT);

    //Set P1.0 to output direction
    GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );

    //Enter LPM3, enable interrupts
    __bis_SR_register(LPM3_bits + GIE);
    //For debugger
    __no_operation();

...chdog Timer interrupt service routine
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector = WDT_VECTOR
__interrupt
#elif defined(__GNUC__)
__attribute__((interrupt(WDT_VECTOR)))
#endif
void WDT_A_ISR(void)
{
    //Toggle P1.0
    GPIO_toggleOutputOnPin(
        GPIO_PORT_P1,
        GPIO_PIN0);
}
```

Notice the DriverLib function that sets up the WDT_A for interval timing.

You can choose which clock to use; we selected ACLK. By the way, what speed is ACLK running at? (This example uses ACLK at the default rate.)

As described, dividing ACLK/8192 gives us an interval of ¼ second.

The WDT_A is a system (SYS) interrupt, so it's IFG and IE bits are in the Special Functions Register. It's always good practice to clear a flag before enabling the interrupt. (Remember, CPU won't be interrupted until we set GIE.)

Along with enabling interrupts globally (GIE=1), this example puts the CPU into low power mode (LPM3).

When the interrupt occurs, the CPU wake up and handles it, then goes back into LPM3. (Low Power modes will be discussed further in a future chapter.)

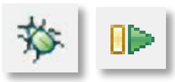
These GPIO functions should be familiar by now …

They got a little bit fancy with the interrupt vector syntax. This code has been designed to work with 3 different compilers:

TI, IAR, and GNU C compiler.

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.

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 about 1 second intervals.

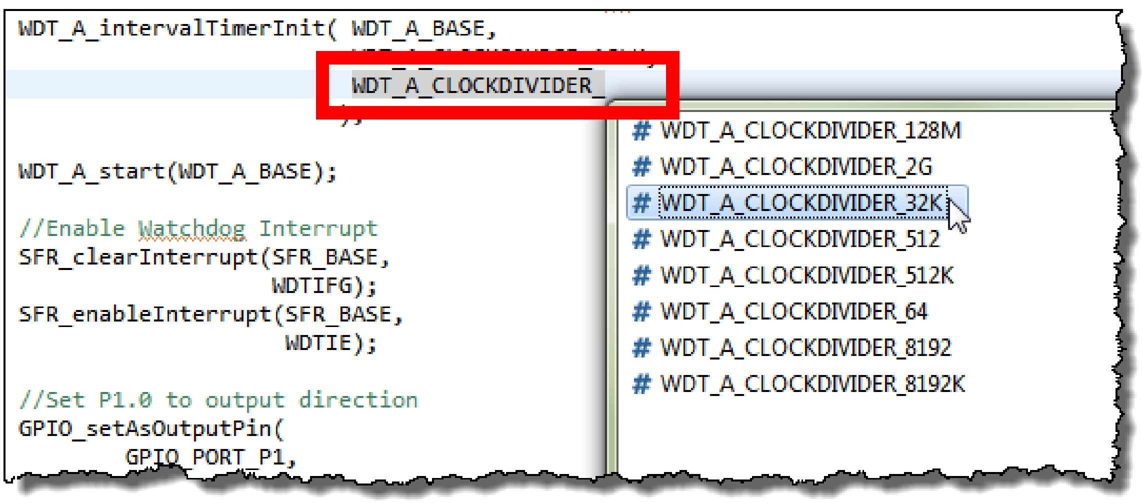
Tip: If you want help with selecting and typing function arguments, you can you the autocomplete feature of CCS. Just type part of the test, such as:

```
WDT_A_CLOCKDIVER_
```

and then hit:

```
Control-TAB
```

and a popup box appears providing you with choices – select the one you want. In this case, we suggest you divide by 32K.



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's blink rate.

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**

Lab 05 Worksheet (3)

7. Write the code to enable a GPIO interrupt for the listed Port.Pin?
   GPIO pin to use: F5529 = P1.1, FR4133 = P1.2, FR5969 = P1.1

F5529 and FR5969:

```
GPIO_setAsInputPinWithPullUpResistor ( GPIO_PORT_1, GPIO_PIN1 );                    // set up pin as input
GPIO_interruptEdgeSelect ( GPIO_PORT_P1, GPIO_PIN1, GPIO_LOW_TO_HIGH_TRANSITION);   // set edge select
GPIO_clearInterruptFlag ( GPIO_PORT_P1, GPIO_PIN1 );                                // clear individual INT
GPIO_enableInterrupt ( GPIO_PORT_P1, GPIO_PIN1 );                                   // enable individual INT
```

FR4133:

```
GPIO_setAsInputPinWithPullUpResistor ( GPIO_PORT_1, GPIO_PIN2 );                    // set up pin as input
GPIO_interruptEdgeSelect ( GPIO_PORT_P1, GPIO_PIN2, GPIO_LOW_TO_HIGH_TRANSITION);   // set edge select
GPIO_clearInterruptFlag ( GPIO_PORT_P1, GPIO_PIN2 );                                // clear individual INT
GPIO_enableInterrupt ( GPIO_PORT_P1, GPIO_PIN2 );                                   // enable individual INT
```

Lab 05 Worksheet (4)

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().

Lab 05 Worksheet (5)

Interrupt Priorities & Vectors

9. Check the interrupt that has higher priority:

	F5529	FR4133	FR5969
☐ GPIO Port 2	int42	int36	int36
☑ WDT Interval Timer	int56	int49	int41

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 set up interrupt nesting (though this isn't recommended)

Lab 05 Worksheet (6)

10. Where do you find the name of an "interrupt vector"?

It's defined in the device specific header file.

For example: msp430f5529.h, msp430fr5969.h, or msp430fr4133.h

Sidebar – Interrupt Vector Symbols

We needed all of these vector names to create an 'unused vectors' source file that's provided you for in this lab exercise:

`unused_interrupts.c`

To get all of these symbols, we followed these steps:

1. Copy every line from the header file with the string "_VECTOR".
2. Delete the duplicate lines (each vector symbol shows up twice in the file)
3. Replace "#define " with "#pragma vector=" (and remove the text after the vector name)
4. Delete the "RESET_VECTOR" symbol as this vector is handled by the compiler's initialization routine

Lab 05 Worksheet (7)

11. How do you write the code to set the interrupt vector?

```
// 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 grouped interrupts. It's better to read the P1IV register so you can handle multiple pin interrupts using switch/case statement

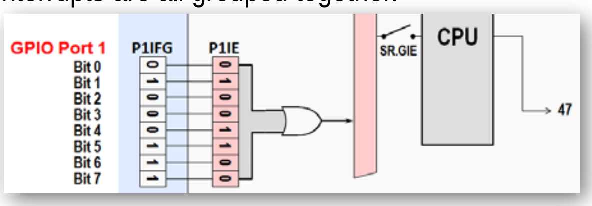
Lab 05 Worksheet (8)

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 (9)

14. Creating ISR's for grouped interrupts is as easy as following a 'template'. Toggle P1.0 when button is pressed. F5529/FR5969 uses P1.1;

```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {
   switch(__even_in_range( ___P1IV___, 0x10 )) {
// F5529 and FR5969 use P1.1 for button:
      case 0x02: break;          // Pin 0
      case 0x04:                 // Pin 1
            GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
                 break;
      case 0x06: break;          // Pin 2
// FR4311 uses P1.2 for button:
      case 0x02: break;          // Pin 0
      case 0x04: break;          // Pin 1
      case 0x06:                 // Pin 2
            GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
                 break;
```

Timers

Introduction

Timers are often thought of as the heartbeat of an embedded system.

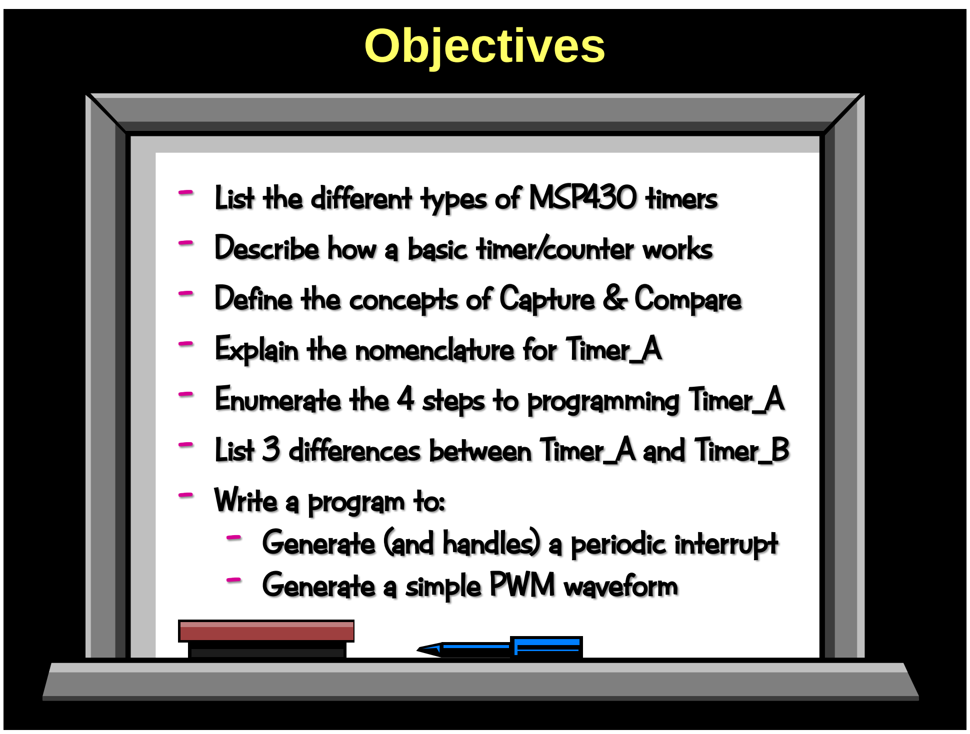
Whether you need a periodic wake-up call, a one-time delay, or need a means to verify that the system is running without failure, Timers are the solution.

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

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

As the chapter nears conclusion, there's a brief summary of the differences between TIMER_A and TIMER_B. Bottom line, if you know how to use TIMER_A, then you can use TIMER_B; but, there are a couple of extra features that TIMER_B provides.

Learning Objectives



Chapter Topics

Timers **6-1**

Prerequisites and Tools *6-2*

Overview of MSP430 Timers *6-3*
TIMER_A/B Nomenclature 6-4
Timer Summary 6-5

Timer Basics: How Timers Work *6-6*
Counter 6-6
Frequency, Time-Period, Resolution 6-7
Capture 6-8
Compare 6-9

Timer Details: Configuring TIMER_A *6-12*
1. Counter: TIMER_A_configure…() 6-13
Timer Counting Modes 6-14
Summary of Timer Setup Code – Part 1 6-18
2a. Capture: TIMER_A_initCapture() 6-19
2b. Compare: TIMER_A_initCompare() 6-21
Summary of Timer Setup Code – Part 2 6-23
Output Modes 6-24
PWM anyone? 6-29
3. Clear Interrupt Flags and TIMER_A_startTimer() 6-30
4. Interrupt Code (Vector & ISR) 6-31

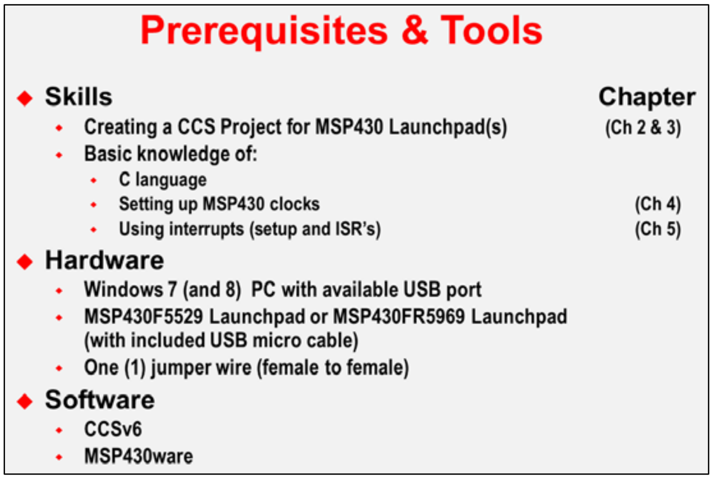
TIMER_A DriverLib Summary *6-32*

Differences between Timer's A and B *6-33*

Lab Exercise *6-35*

Prerequisites and Tools

To get full entitlement from this chapter, we expect that you are already familiar with MSP430ware's DriverLib as well as MSP430 clocking and interrupts. The "extra" piece of hardware required for this chapter is a single jumper wire.



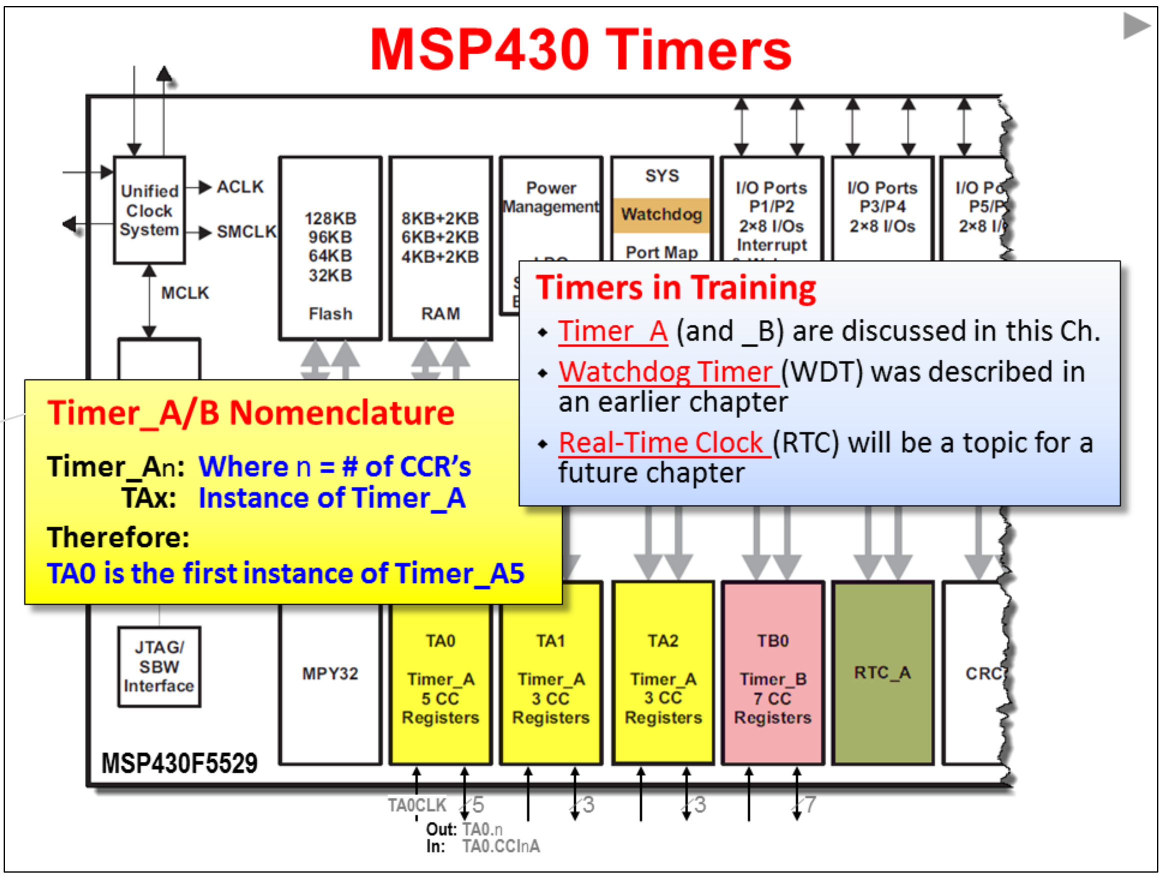
Overview of MSP430 Timers

The MSP430F5529 timers are highlighted in the following block diagram.

- Yellow marks the three instances of the TIMER_A module.
- Pink was used for TIMER_B.
- Dark brown highlights the real-time clock (RTC_A).
- Light brown differentiates the Watchdog timer inside the SYS block

The “Timers in Training” callout box describes where the various timers are discussed in this workshop. Timers A and B are covered in this chapter. We have already covered the Watchdog timer in a previous chapter.

The RTC module will be discussed in a future chapter. A brief description of the RTC tells us that it’s a very low-power clock; has built-in calendar functions; and often includes “alarms” that can interrupt the CPU. It is frequently used for keeping a time-base while the CPU is in low-power mode.



Nomenclature is discussed on the next page

TIMER_A/B Nomenclature

The nomenclature of the TIMER_A and _B peripherals is a little unusual. First of all, you may have already noticed that the MSP430 team often adds one of two suffixes to their peripheral names to indicate when features have been added (or modified).

- Some peripherals, such as the Watchdog Timer go from “WDT” to “WDT+”. That is, they add a “+” to indicate the peripheral has been updated (usually with additional features).
- Other peripherals are enumerated with letters. For example, three sophisticated MSP430 timers have been introduced: TIMER_A, TIMER_B, and TIMER_D. *(What happened to _C? Even I don’t know that. <ed>)*

The use of a suffix is the generic naming convention found on the MSP430. With the timers, though, there are a couple more naming variations to be discussed.

As we will cover in great detail during this chapter, these timers contain one or more Capture and Compare Registers (CCR); these are useful for creating sophisticated timings, interrupts and waveforms. The more CCR registers a timer contains, the more independent waveforms that can be generated. To this end, the documentation often includes the number of CCR registers when listing the name of the timer. For example, if TIMER_A on a given device has 5 CCR registers, they often name it:

```
Timer_A5
```

But wait, that’s not all. What happens when a device, such as the ‘F5529 has more than one instance of TIMER_A? Each of these instances needs to be enumerated as well. This is done by appending the instance number after the word “Timer”, as in Timer0.

To summarize, here’s the long (and short) names for each of the ‘F5529 TIMER_A modules:

Instance	Long Name	Short Name
0	Timer0_A5	TA0
1	Timer1_A3	TA1
2	Timer2_A3	TA2

Timer Summary

The ‘F5529 contains most of the different types of timers found across the MSP430 family; in fact, the only type of timer not present on this device is the high-resolution TIMER_D.

The following summary provides a snapshot of what timers are found on various MSP430 devices. You’ll find our ‘F5529 and ‘FR5969 devices in the last two columns of the table.

A one-line summary of each type of timer is listed below the table.

MSP430 Timers

	L092	G2553	FR4133	F5172	F5529	FR5969
Timer_A	2 x A3	2 x A3	2 x A3	1 x A3	1 x A5 2 x A3	2 x A3 2 x A2*
Timer_B					1 x B7	1 x B7
Timer_D				2 x D3		
Real-Time Clock			RTC Counter		RTC_A	RTC_B
Watchdog	WDT_A	WDT+	WDT_A	WDT_A	WDT_A	WDT_A

Timer_A: ‘A3’ means it has 3 Capture/Compare Registers (used to generate signals & ints)
Timer_B: Same as A, but improves PWM
Timer_D: Same as B, adding hi-res timing
WDT+: Watchdog or Interval Modes; PSW Protected; Can stop; Select Clk; Clk fail-safe
WDT_A: Same as WDT+, but with 8 timer intervals rather than 4
BT1/RTC: Basic timer has 2x8-bit counters (can use as 1x16-bits) with calendar functions
RTC_A: 32-bit counter with a calendar, flexible programmable alarm, and calibration
RTC_B: Same as RTC_A, but adds switchable battery backup in case main-power fails

Timer Basics: How Timers Work

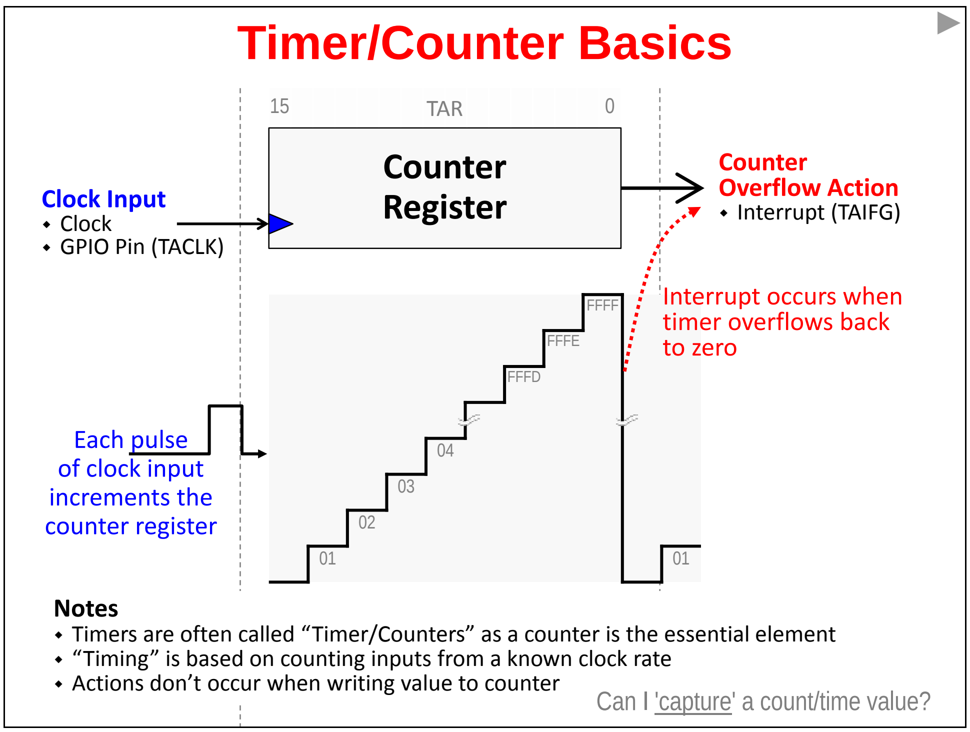
Before we discuss the details of TIMER_A, let's begin with a quick overview describing how timers work. Specifically, we will start by describing how a timer is constructed using a **Counter**. Next, we'll investigate the **Capture** and **Compare** capabilities found in many timers.

Counter

A **counter** is the fundamental hardware element found inside a timer.

The other essential element is a **clock** input. The counter is incremented each time a clock pulse is applied to its clock input. Therefore, a 16-bit timer will count from zero (0x0000) up to 64K (0xFFFF).

When the counter reaches it reaches its maximum value, it overflows – that is, it returns to zero and starts counting upward again. Most timer peripherals can generate an interrupt when this overflow event occurs; on TIMER_A, the interrupt flag bit for this event is called TAIFG (TIMER_A Interrupt Flag).



The clock input signal for TIMER_A (named TACLK) can be one of the internal MSP430 clocks or a signal coming from a GPIO pin.

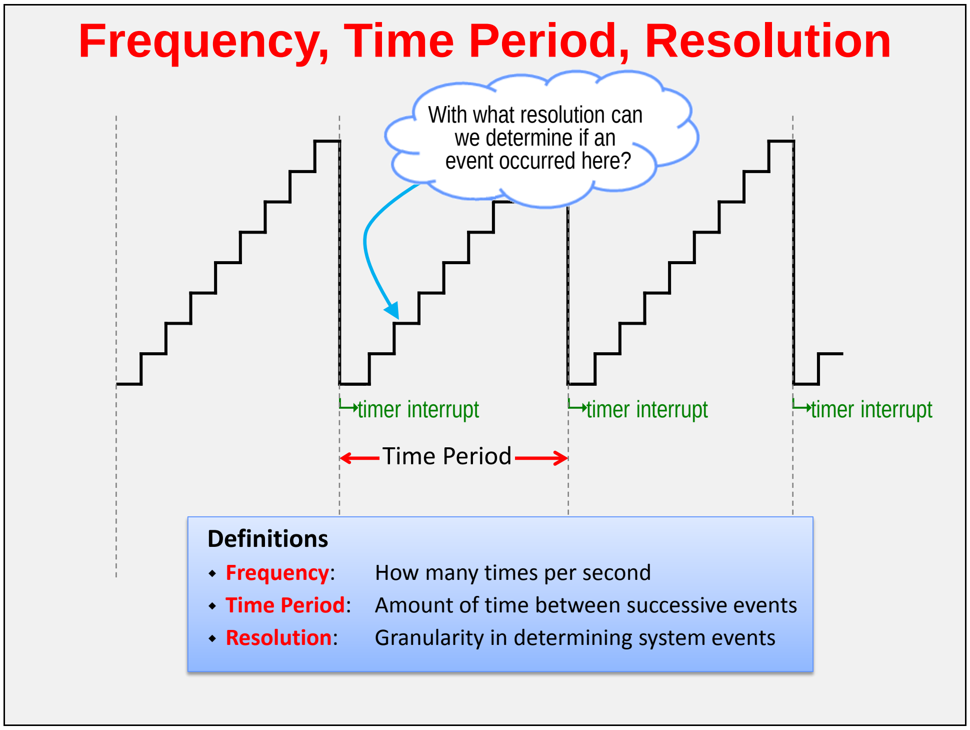
Many engineers call these peripherals "Timer/Counters" as they provide both sets of functionality. They can generate interrupts or waveforms at a specific time-base – or could be used to count external events occurring in your system.

One final note about the MSP430 timers: they do not generate interrupts (or other actions) when you write to the counter register. For example, writing "0" to the counter won't generate the TAIFG interrupt.

Frequency, Time-Period, Resolution

The Timer's ability to create a consistent, periodic interrupt is quite valuable to system designers. *Frequency* and *Time Period* are two terms that are often used to describe the rate of interrupts.

- How many times per second that a timer creates an interrupt defines its ***Frequency***.
- Conversely, the amount of time in-between interrupt events is defined as the ***Time Period***.



If a timer only consisted of a single counter, its *resolution* would be limited to the size of the counter.

If some event were to happen in a system – say, a user pushed a button – we could only ascertain if that event occurred within a time period. In other words, we can only determine if it happened between two interrupts.

Looking at the above diagram, we can see that there is "more data" available – that is, if we were to read the actual counter value when the event occurred. Actually, we can do this by setting up a GPIO interrupt; then, having the ISR read the value from the counter register. In this case the resolution would be better, but it is still limited by:

- It takes more hardware (an extra GPIO pin is needed)
- The CPU has to execute code – this consumes power and processing cycles
- The resolution is less deterministic because it's based upon the latency of the interrupt response; in other words, how fast can the CPU get to reading the counter … and how consistent can this be each time it occurs

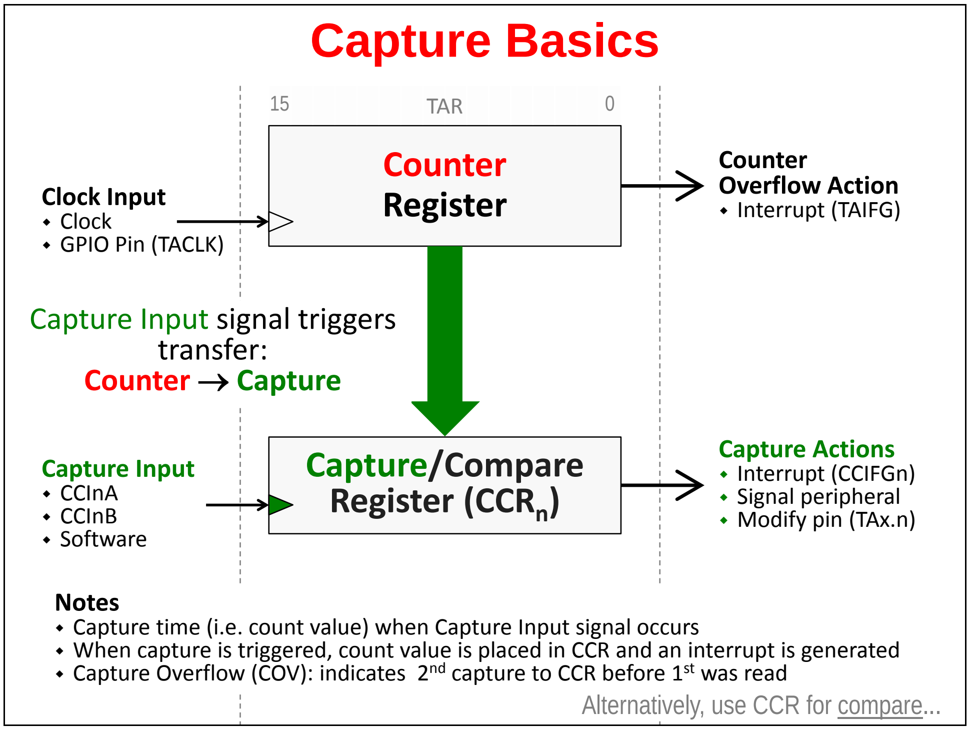
There is a better way to implement this in your system … turn the page and let's examine the timer's ***Capture*** feature.

Capture

The ***Capture*** feature does just that. When a capture input signal occurs, a snapshot of the Counter Register is *captured*; that is, it is copied into a capture register (CCR for Capture and Compare Register). This is ideal since it solves the problems discussed on the previous page; we get the timer counter value captured with no latency and very, very little power used (the CPU isn't even needed, so it can even remain in low-power mode).

The diagram below builds upon our earlier description of the timer. The top part of the diagram is the same; you should see the Counter Register flanked by the Clock Input to the left and TAIFG action to the right.

The bottom portion of the slide is new. In this case, when a Capture Input signal occurs, the value from the Counter Register is copied to a capture register (i.e. CCR).



A few notes about the *capture* feature:

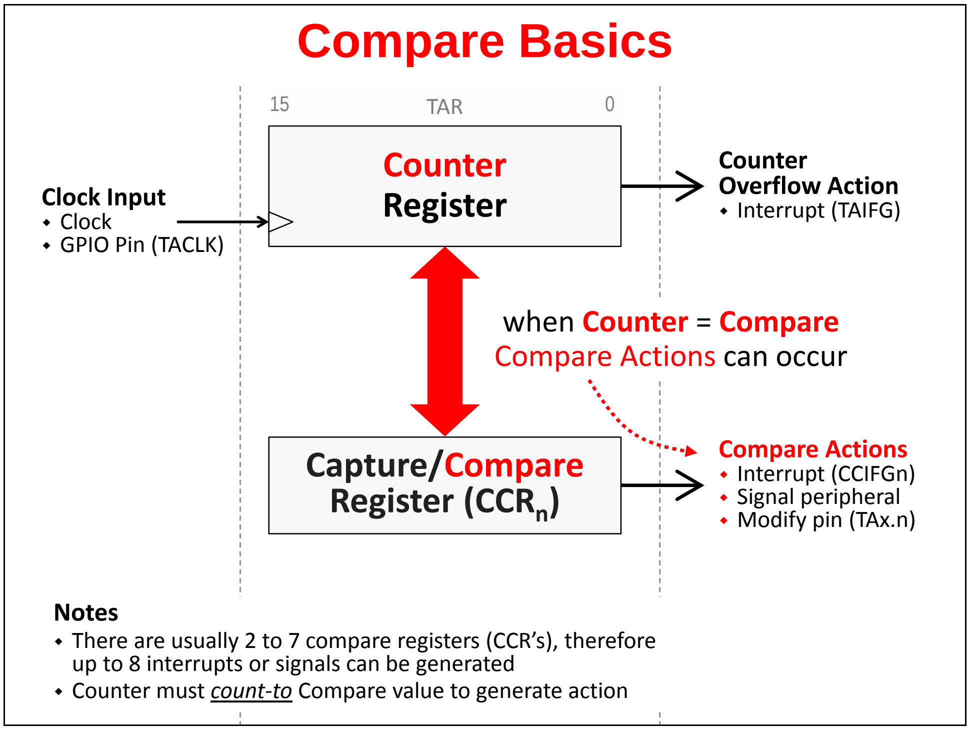
- As we discussed earlier, the MSP430 timers (TIMER_A, TIMER_B, and TIMER_D) have multiple CCR registers; check your datasheet to determine how many are available per timer peripheral. Each CCR, though, has its own capture input signal.
- The Capture Input signal can be connected to a couple of different signals (CCInA, CCInB) or triggered in software
- The Capture Input hardware signals (CCInA, CCInB) are connected differently for each CCR register and device. You need to reference the datasheet to verify what options are available on your specific device.
- When a capture occurs, the CCR can trigger further actions. This "action" signal can generate an interrupt to the CPU, trigger another peripheral, and/or modify the value of a pin.

As we just discussed, the Capture feature provides a deterministic method of capturing the count value when triggered. While handy, there is another important requirement for timers…

Compare

A key feature for timers is the ability to create a consistent, periodic interrupts.

As we know, TIMER_A can do this, but the timer's frequency (i.e. time period) is limited to dividing the input clock by 2^{16}. So, while the timer may be *consistent*, but not very flexible. Thankfully, the ***Compare*** feature of TIMER_A (TIMER_B & TIMER_D) solves this problem.



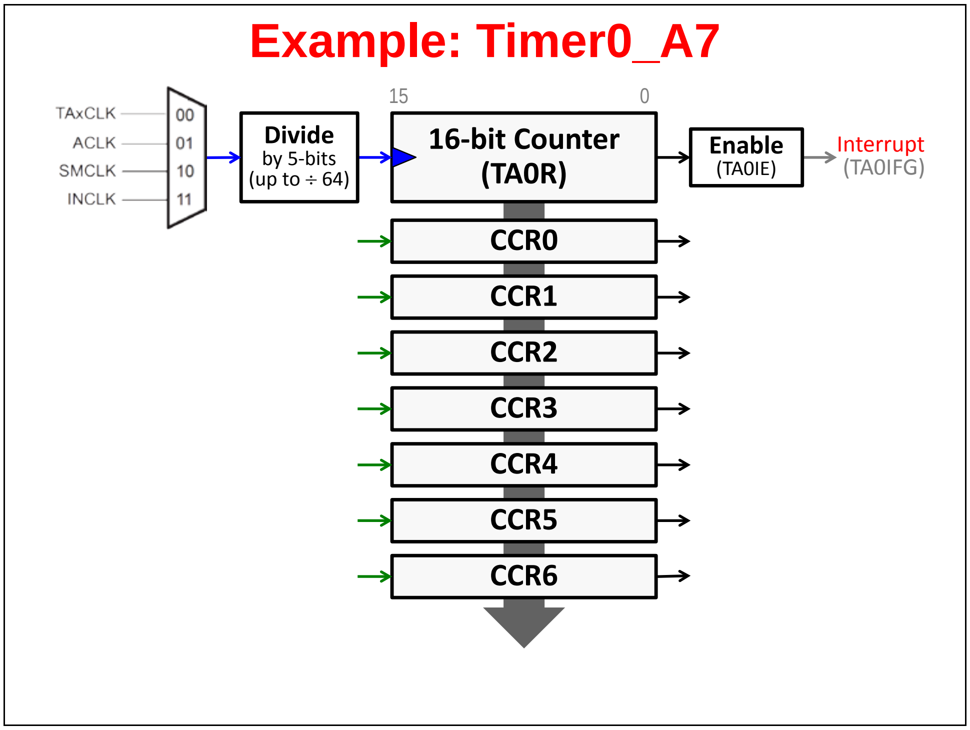
Once again, the top portion of this diagram remains the same (Clock Input + Counter Register).

The bottom portion of the diagram differs from the previous diagrams. In this case, rather than using the CCR register for capture, it's used as a *compare* register. In this mode, whenever a match between the Counter and Compare occurs, a compare action is triggered. The compare actions include generating an interrupt, signaling another peripheral (e.g. triggering an ADC conversion), or changing the state of an external pin.

The "modify pin" action is a very powerful capability. Using the timer's *compare* feature, we can create sophisticated PWM waveforms. (Don't worry, there's more about this later in the chapter.)

Timer Summary – showing multiple CCR's

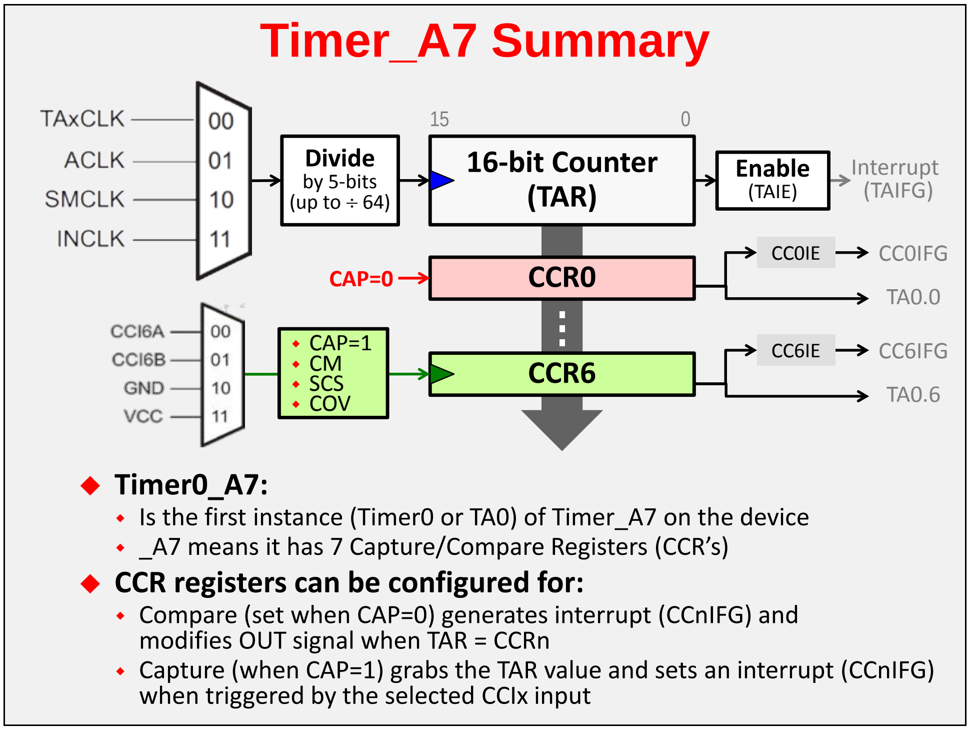
The following example of a Timer0_A7 provides us a way to summarize the timer's hardware.



Remember:

- Timer0 means it's the first instance of Timer_A on the device.
- _A7 means that it's a Timer_A device and has 7 capture/compare registers (CCR's)
- The clock input, in this example, can be driven by a TACLK signal/pin, ACLK, SMCLK or another internal clock called INCLK.
- The clock input can be further divided down by a 5-bit scalar.
- The TA0IE interrupt enable can be used to allow (or prevent) an interrupt (TA0IFG) from reaching the CPU whenever the counter (TA0R) rolls over.

This next diagram allows us to look more closely at the Capture and Compare functions.



Every CCR register has its own control register. Notice above, that the "CAP" bit configures whether the CCR will be used in capture (CAP=1) or compare mode (CAP=0).

You can also see that each CCR has an interrupt flag, enable, and output signal associated with it. The output signal can be routed to a pin or a number of other internal peripherals.

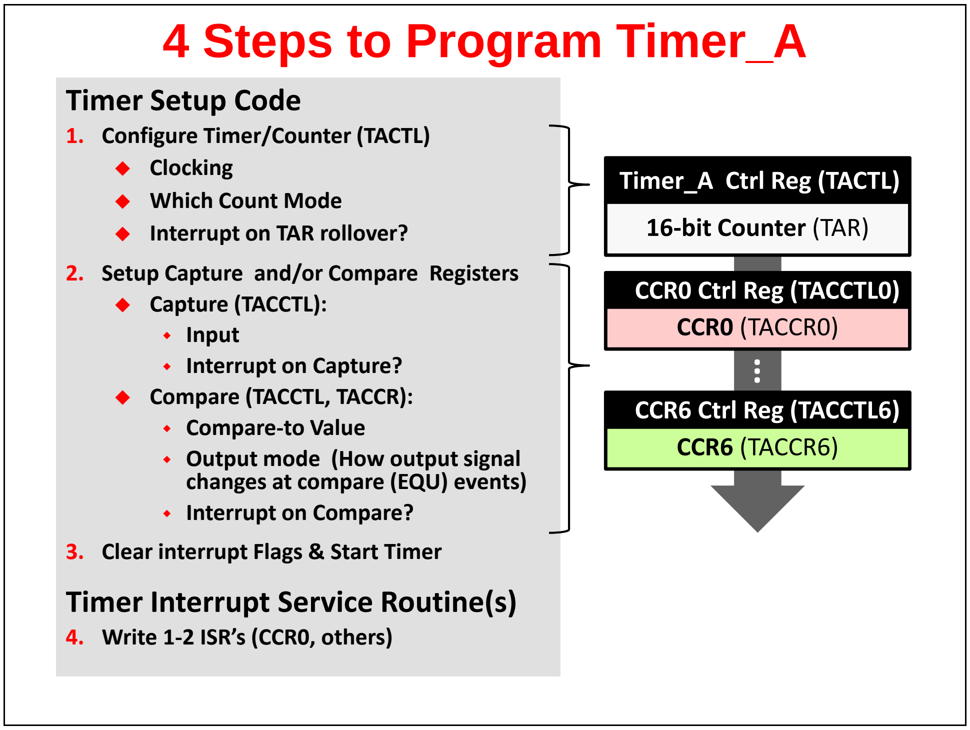
As we go through the rest of this chapter, we'll examine further details of the CCR registers as well as the various "actions" that the timer generates.

In the next section, we'll begin examining how to configure the timer using the MSP430ware DriverLib API.

Timer Details: Configuring TIMER_A

There are four steps required to get Timer_A working in your system:

1. Configure the *main Timer/Counter* by programming the TACTL control register.
2. Setup each CCR that is needed for your application. We will examine this step from both the *Capture* and *Compare* perspective.
3. Next, you need to start the timer. (We also listed clearing the timer IFG bits, which is normally done right before starting the timer.)



4. Finally, if your timer is generating interrupts, you need to have an associated ISR for each one. *(While interrupts were covered in the last chapter, we briefly summarize this again in context of the Timer_A.)*

We will intermix how to write code for the timer with further examination of the timer's features.

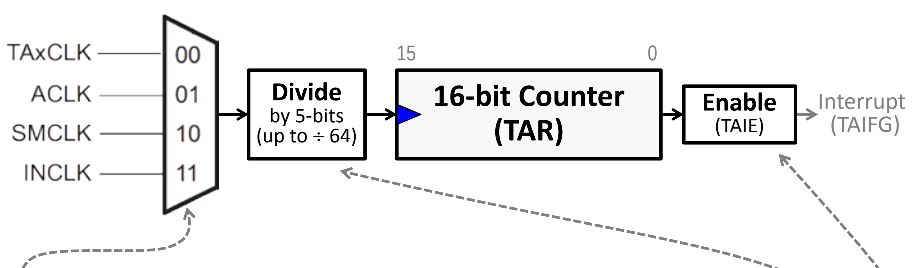
1. Counter: TIMER_A_configure…()

The first step to using TIMER_A is to program the main timer/counter control register. The MSP430ware Driver Library provides 3 different functions for setting up the main part of the timer:

```
TIMER_A_configureContinuousMode()
TIMER_A_configureUpMode()
TIMER_A_configureUpDownMode()
```

We will address the different modes on the next page. For now, let's choose 'continuous' mode and see how we can configure the timer using the DriverLib function.

1. Configure Timer/Counter




```
Timer_A_initContinuousModeParam  initParam = { 0 };
    initParam.clockSource =                   TIMER_A_CLOCKSOURCE_ACLK;
    initParam.clockSourceDivider =            TIMER_A_CLOCKSOURCE_DIVIDER_1;
    initParam.timerInterruptEnable_TAIE =     TIMER_A_TAIE_INTERRUPT_ENABLE;
    initParam.timerClear =                    TIMER_A_DO_CLEAR;
    initParam.startTimer =                    false;
Timer_A_initContinuousMode( TIMER_A0_BASE, &initParam );
```

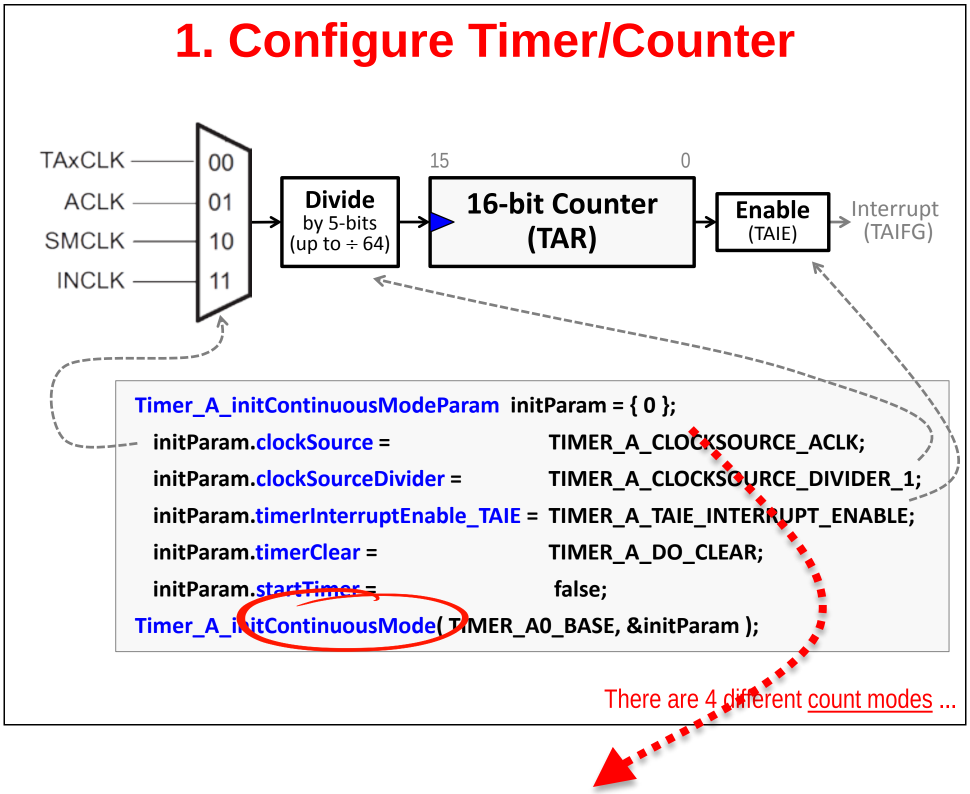
From the diagram, we can see that 3 different hardware choices need to be made for our timer configuration; the arrows demonstrate how the function parameters relate to these choices. Let's look at each parameter one-by-one:

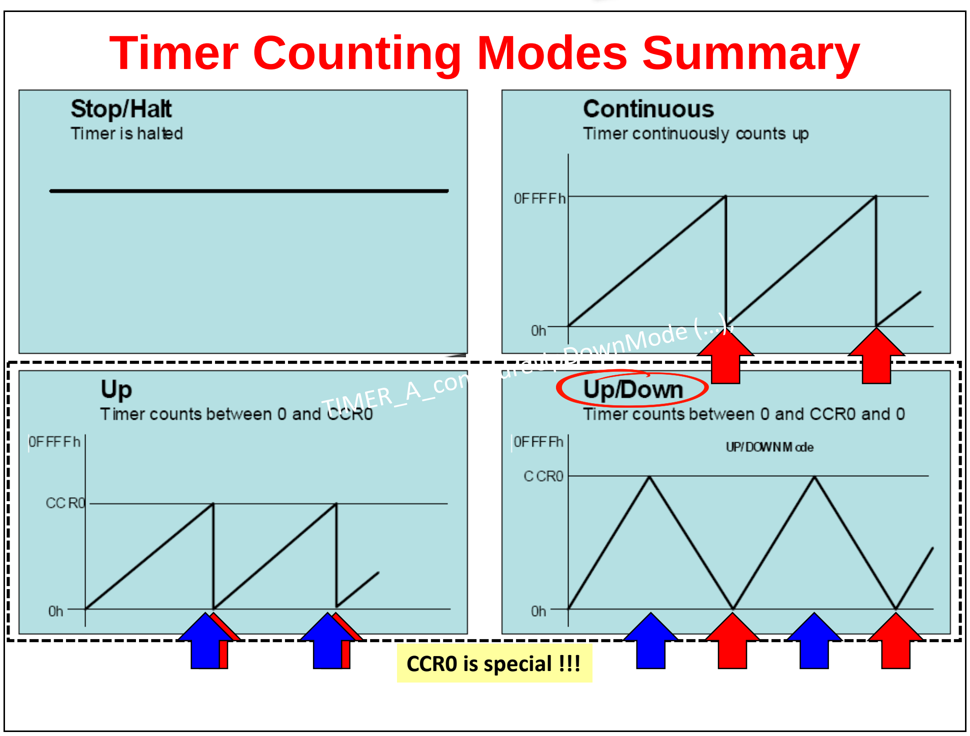
- The first parameter chooses which Timer_A instance you want to program. In our example, we have chosen to program TA0 (i.e. Timer0_A). Conveniently, the DriverLib documentation provides enumerations for all the supported choices. *(This is the same for all DriverLib parameters, so we won't keep repeating this statement. But, this is very handy to know!)*
- The 2nd parameter lets you choose which clock source you want to use. We chose SMCLK.
- The next parameter picks one of the provided clock pre-scale values. The h/w lets you choose from one of 20 different values; we picked ÷ 64.
- Parameter four lets us choose whether to interrupt the processor when the counter (TA0R) rolls over to zero. This parameter ends up setting the TA0IE bit.
- Finally, do you want to have the timer counter register (TA0R) reset when the other parameters are configured?

> **Remember…**
>
> **TAR**: Timer_A count Register
>
> **TA0R**: Name for count register when referring to instance "0" (i.e. Timer0_A)

Timer Counting Modes

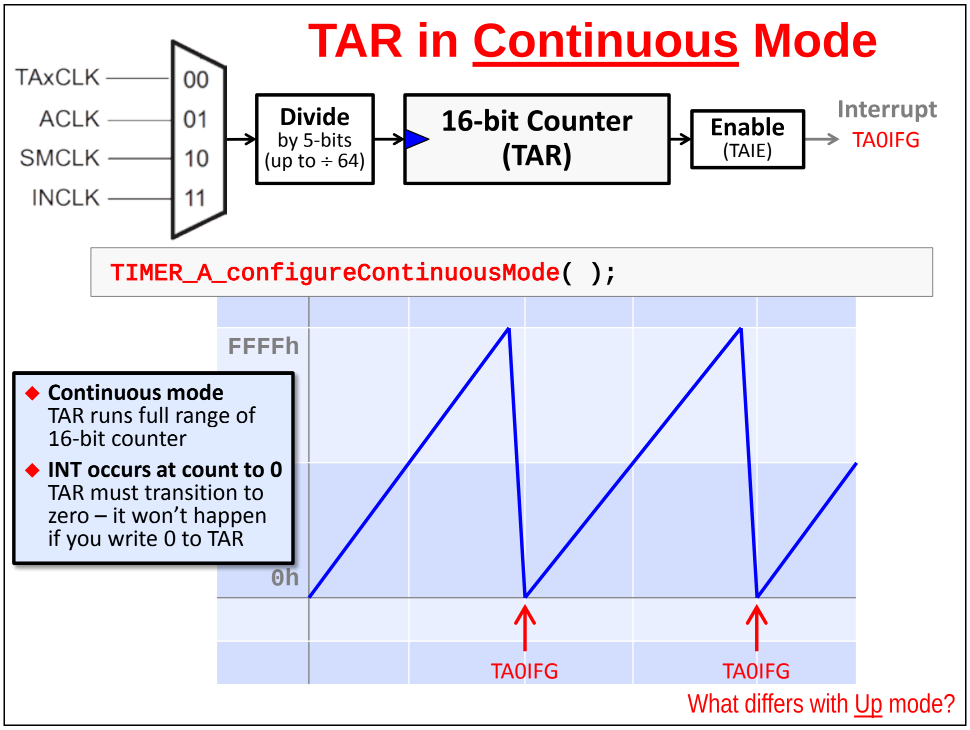
There are three different ways that the timer counter (TAR) can be incremented. These correlate to the three configuration functions listed on the previous page. This page provides a single-slide summary of the different modes – but we'll examine each one over the following three pages.





Continuous Mode

Thus far we have described the timer's counter operating in the ***Continuous*** mode; in fact, this was the configuration example we just discussed.



The different counting modes describe how the timer counter register (TAR) is incremented or decremented. For example, in *Continuous* mode, the timer counts from 0x0 up to 0xFFF and then rolls back to 0x0, where it begins counting up again. (This is shown in the diagram above.)

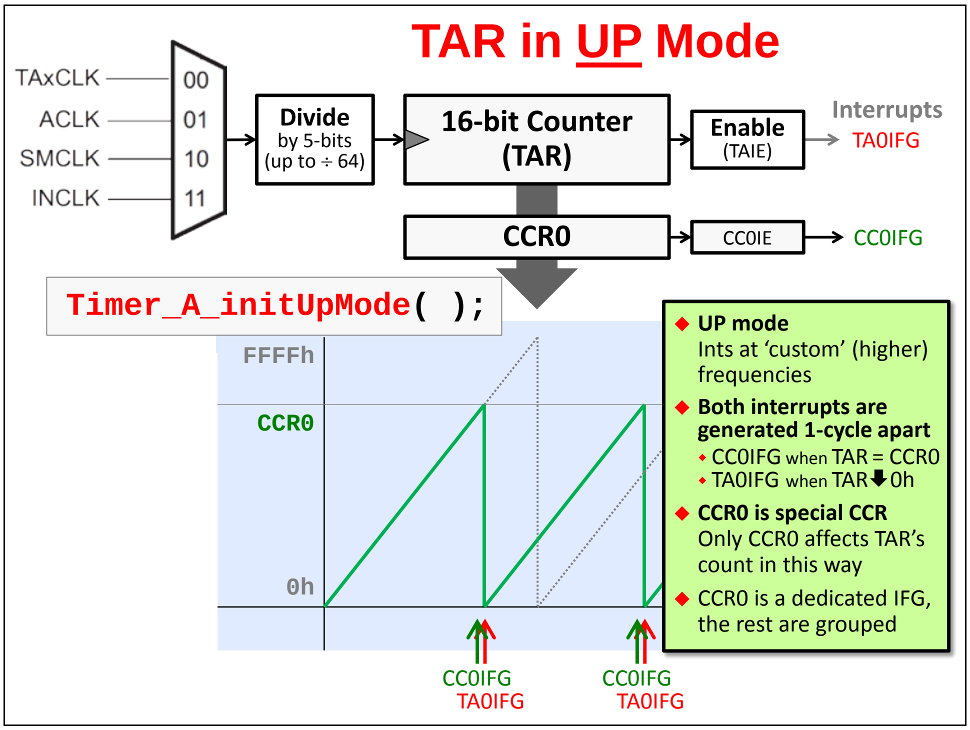
As you can see, every time the counter rolls back to zero, the TAIFG bit gets set; which, if enabled, interrupts the processor every 2^{16} input clocks. (Since our previous example was for Timer0_A, the diagram shows TA0IFG getting set.)

Up Mode

The ***Up*** counting mode differs from the *Continuous* mode by resetting back to zero whenever the counter matches CCR0 (Capture and Compare Register 0).

You can see the different waveforms compared on the slide below. The green waveform counts ***Up*** to the value found in CCR0, and then resets back to zero.

On the other hand, the grey dotted waveform shows how, when in *Continuous* mode, the counter goes past CCR0 and all the way to 0xFFFF.



In *Up* mode, since we are using the CCR0 register, the timer can actually generate two interrupts:

- CC0IFG (for Timer0_A, this bit is actually called TA0CC0IFG)
- TAIFG (for Timer0_A, this bit is called TA0IFG)

You're not seeing a color misprint; the two interrupts do not happen at the exact same time, but rather 1 cycle apart. The CC0IFG occurs when there is a compare match, while the TA0IFG interrupt occurs once the counter goes back to zero.

If you compare these two *Up* mode interrupts to the one generated in the *Continuous* mode, you'll see they occur at a more rapid frequency. This is a big advantage of the *Up* mode; your frequency is not limited to 2^{16} counts, but rather can be anywhere within the 16-bit counter's range. (The downside is that you also have to configure the CCR0 registers.)

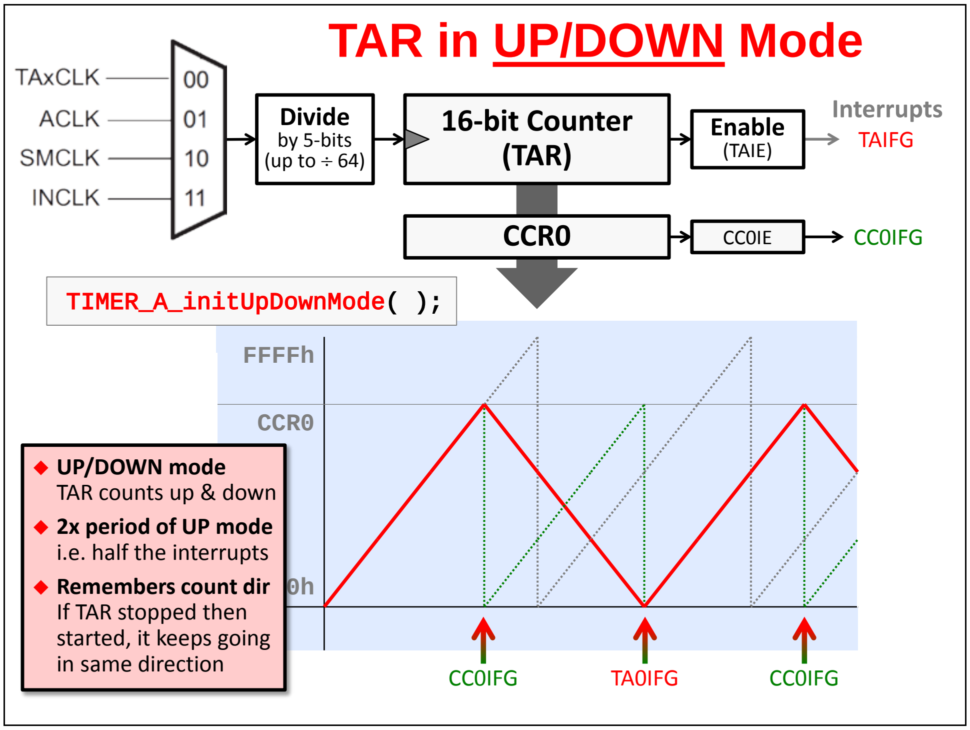
Note: The CCR0 (Capture and Control Register 0) is special. That is, it is special in comparison to the other CCR registers. It is only CCR0 that can be used to define the upper limit of the counter in Up (or UpDown) count mode.

The other special feature of CCR0 is that it provides a dedicated interrupt (CC0IFG). In other words, there is an Interrupt Vector location dedicated to CC0IFG. All the other Timer_A interrupts share a common vector location (i.e. they make up a grouped interrupt).

Up/Down Mode

The ***UpDown*** count mode is similar to *Up* in that the counter only reaches the value in CCR0 before changing. In this case, though, it actually changes direction and starts counting down rather than resetting immediately back to zero.

Not only does this double the time period (i.e. half the timer's frequency), but it also spreads out the two interrupts. Notice how CC0IFG occurs at the peak of the waveform, while TAIFG occurs at the base of the waveform.



In our diagram we show all three counter mode waveforms. The ***UpDown*** mode is shown in red; ***Up*** is shown in green; and the ***Continuous*** mode is shown in grey.

Which Count Mode Should I Use?

When using TIMER_A (or TIMER_B), you have a choice as to which counter mode to use. Here are some things to keep in mind.

- Using ***Continuous*** mode doesn't "tie up" your CCR0 register. It also means you don't have program the CCR0 register.
- ***Up*** mode allows you better control the timer's frequency – that is, you can now control the time period for when the counter resets back to zero.
- On the other hand, the ***UpDown*** mode not only lets you control the frequency better, but it also allows for lower frequencies – since it effectively halves the frequency of the ***Up*** mode.
- Two more considerations of ***UpDown*** mode are:
  - The two interrupts are spaced at ½ the time period from each other.
  - When using multiple CCR registers, you can get two compare interrupts per cycle. (We'll see more on this later.)

Summary of Timer Setup Code – Part 1

Let's summarize Part 1 of the timer setup code – which configures the timer's count options. First of all, as you can see below, we chose to place our timer setup code into its own function. Obviously, this is not a requirement, but it's how we wanted to organize our code examples.

Timer Code Example (Part 1)

```
#include <driverlib.h>

void main(void) {
  // Setup/Hold Watchdog Timer (WDT+ or WDT_A)
  initWatchdog();

  // Configure GPIO ports/pins
  initGPIO();

  // Setup Clocking: ACLK, SMCLK, MCLK (BCS+, UCS, or CS)
  initClocks();

  //--------------------------------------------------------------
  // Then, configure any other required peripherals and GPIO
  initTimers();

  __bis_SR_register( GIE );

  while(1) {
  ...
  }
```

Our earlier example for the Timer/Counter setup code demonstrated using the *Continuous* mode. The following example shows using the *Up* mode. Here's a quick comparison between the two functions – notice that the *Up* mode requires two additional parameters.

Parameter	ContinuousMode Function	UpMode Function
Which Timer?	TIMER_A0_BASE	
Clock Source	TIMER_A_CLOCKSOURCE_SMCLK	
Clock Pre-scaler	TIMER_A_CLOCKSOURCE_DIVDER_xx	
Timer Period	Not applicable	Used to set the CCR0 value
Enable the TAIE interrupt?	TIMER_A_TAIE_INTERRUPT_xxxxxx	
Enable the CCR0 interrupt?	Not applicable	Used to set TA0CC0IFG
Clear the counter (TAR) ?	TIMER_A_DO_CLEAR	

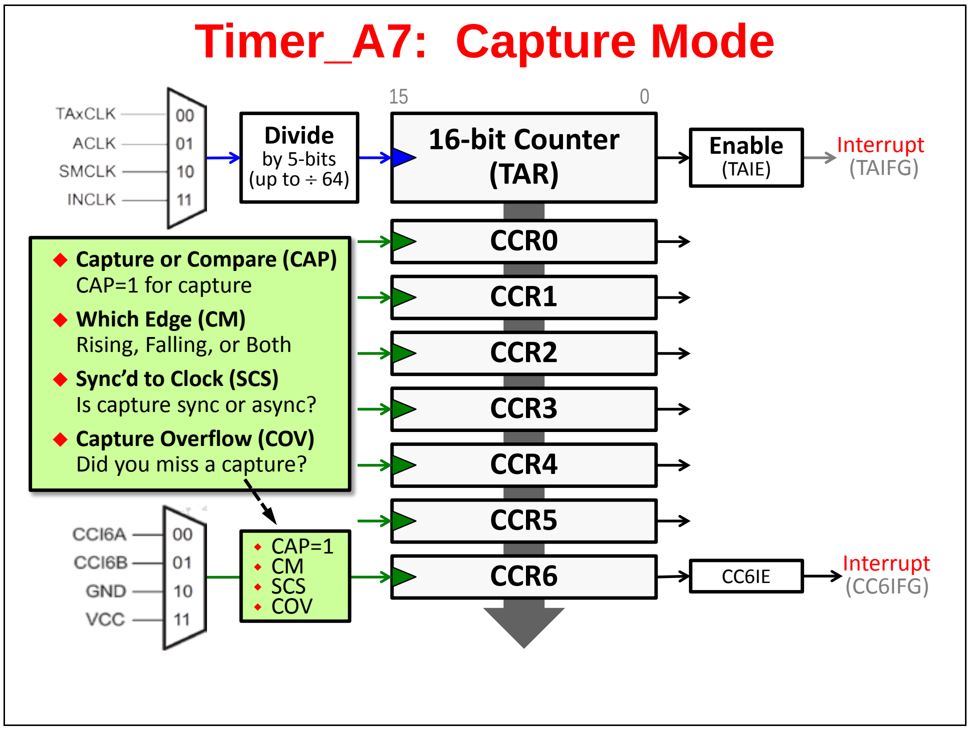
Timer Code Example (Part 1)


```
#include <driverlib.h>

void initTimerA0(void) {
// Setup TimerA0 in Up mode
Timer_A_initContinuousModeParam  initParam = { 0 };
  initParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_
  initParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_
  initParam.timerClear = TIMER_A_DO_CLEAR;
  initParam.startTimer = false;
Timer_A_initContinuousMode( TIMER_A0_BASE, &initParam );
```

2a. Capture: TIMER_A_initCapture()

Before we try writing the code to setup a CCR register for *Capture*, let's first examine the timer's hardware options.

- Most importantly, when wanting to use the *Capture* features, you need to set CAP = 1.
- The CM bit indicates which clock edge to use for the capture input signal.
- Do you want the capture input signal sync'd with the clock input? If so, that's what SCS is for.
- While you don't configure COV, this bit indicates if a capture overflow occurred. In other words, did a 2nd capture occur before you read the captured value from the CCR register?
- Finally, you can select what hardware signal you want to have "trigger" the capture.



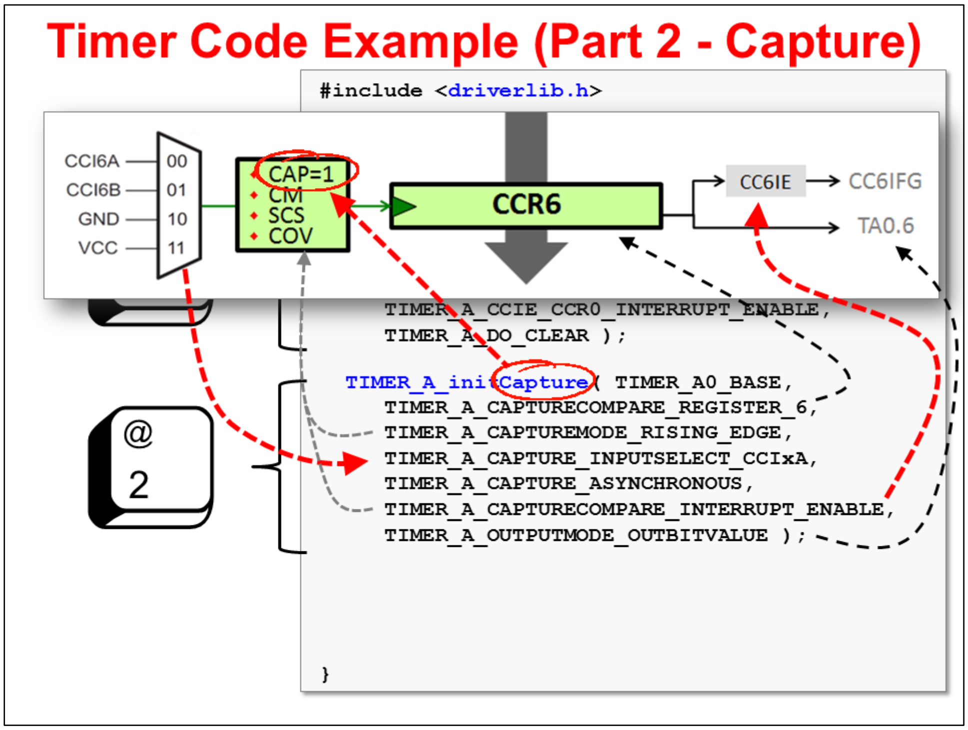
Hint: Each CCR can be configured independently. The flip side to this is that you must configure each one that you want to use; this might involve calling the 'capture' and/or 'compare' configuration functions multiple times.

Use one for capture and the rest for compare. Or, use all for capture. You get to decide how they are used.

Warning: If you are using *Up* or *UpDown* count modes, you should not configure CCR0. Just remember that the `TIMER_A_configureUpMode()` and `TIMER_A_configureUpDownMode()` configuration functions handle this for you.

Capture Code Example

With the Capture mode details in mind, let's examine the code.



To configure a CCR register for Capture mode, use the TIMER_A_initCapture() function. Thankfully, when using DriverLib the code is pretty easy to read (and maintain). Hopefully between the diagram and the following table, you can make sense of the parameters.

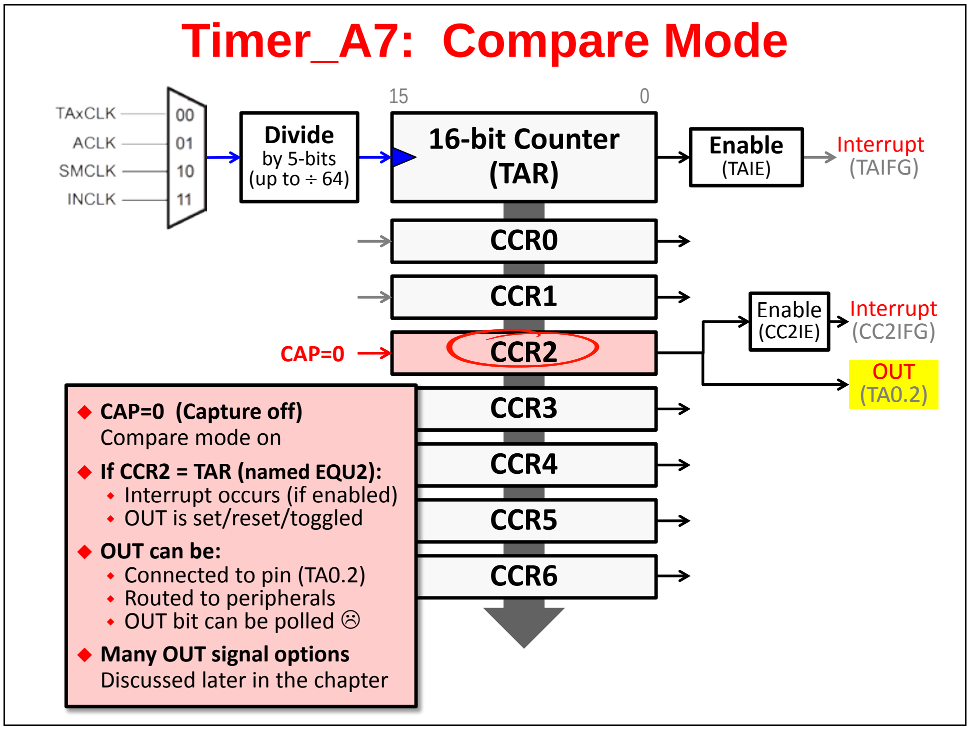
Example's Parameter Value	What is Parameter For?	Value
TIMER_A0_BASE	Which timer are you using?	TA0
TIMER_A_CAPTURECOMPARE_REGISTER_6	Which CCR is being configured?	CCR6
TIMER_A_CAPTUREMODE_RISING_EDGE	Which edge of the capture signal are you using?	Rising
TIMER_A_CAPTURE_INPUTSELECT_CCIxA	The signal used to trigger the capture	CCIn6A
TIMER_A_CAPTURE_ASYNCHRONOUS	Sync the signal to the input clock?	No, don't sync
TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE	Enable the CCR interrupt?	CC6IE = 1
TIMER_A_OUTPUTMODE_OUTBITVALUE	How should the output signal be handled?	OUTMOD=0x0

We've briefly talked about every feature (i.e. function parameter) found in this function except *OutputMode*. The "OUTBITVALUE" (for CCR6) indicates that the value of CCR6's IFG bit should be output to CCR6's Output signal. The output signal can be used by other peripherals or routed to the TA0.6 pin.

Note: With regards to OutputMode, this is just the tip-of-the-iceberg. There are actually 8 possible output mode settings. We will take you through them later in the chapter.

2b. Compare: TIMER_A_initCompare()

The other use of CCR is for *comparisons* to the main timer/counter (TAR).

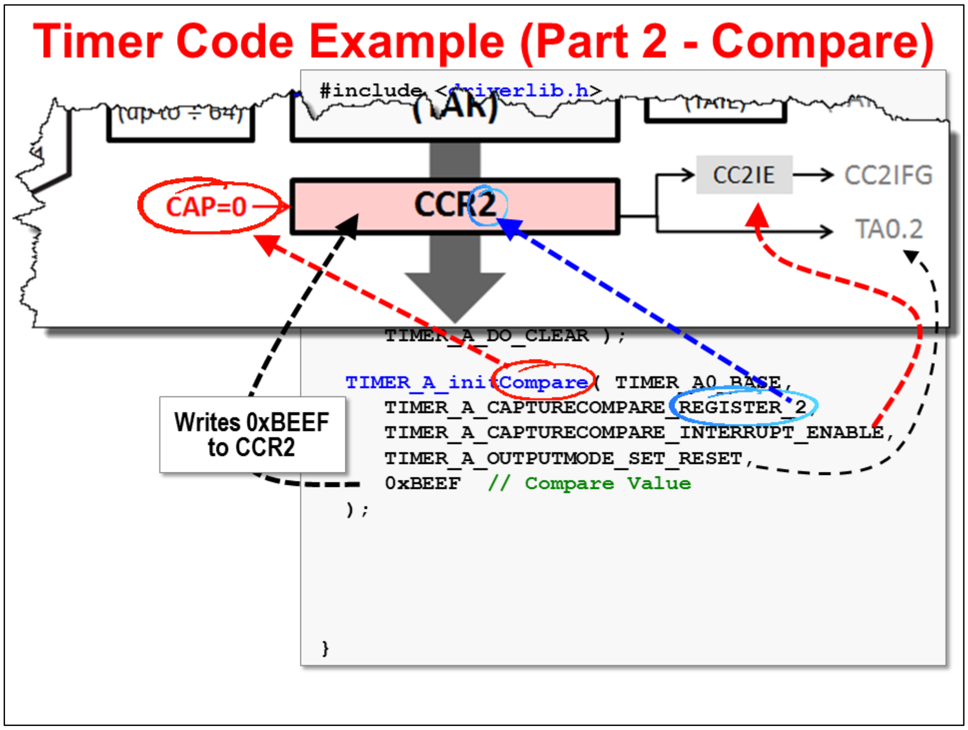


Once again, before we walk through the function that initializes CCR for Compare, let's examine its options:

- Set CAP=0 for the CCR to be configured for Compare Mode. (Opposite from earlier.)
- You must set the CCR2 register to a 16-bit value. When TAR = CCR2:
  - An internal signal called EQU2 is set.
  - If enabled, EQU2 drives the interrupt flag high (CC2IFG).
  - Similar to the Capture mode, the CCR's output signal is modified by EQU2. Again, this signal is available to other internal peripherals and/or routed to a pin (in this case, TA0.2).
  - Again, similar to the Capture mode, there are a variety of possible output modes for the OUT2 signal (which will be discussed shortly).

Compare Code Example

Let's look at the code required to setup CCR2 for use in a Compare operation.



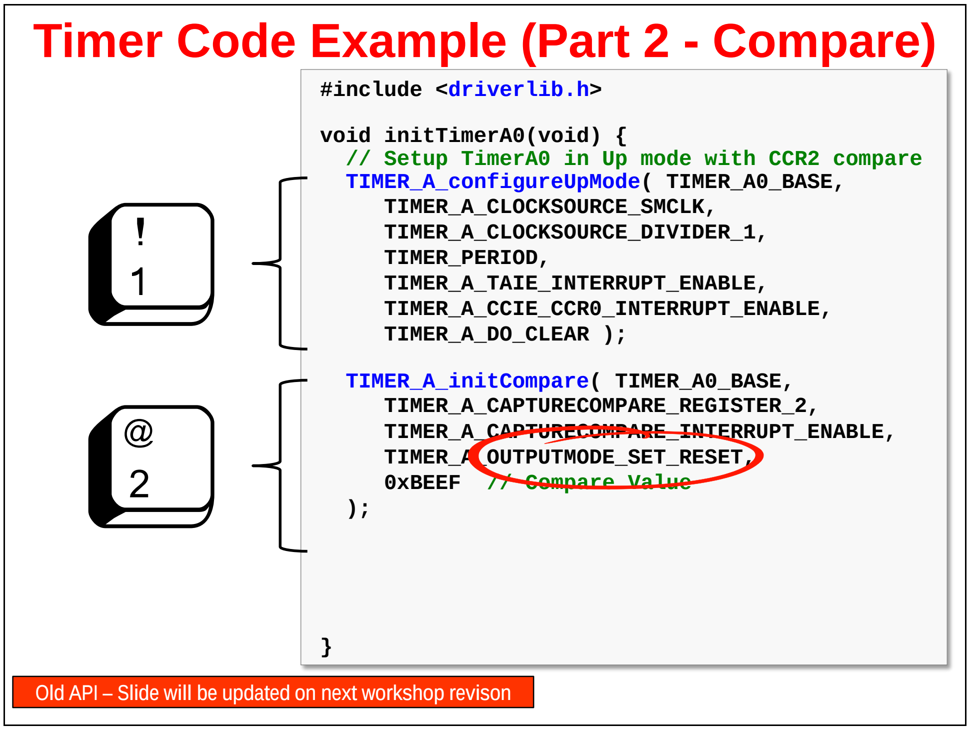
One thing you might notice about the TIMER_A_initCompare() function is that it requires fewer parameters than the complementary initCompare function.

Example's Parameter Value	What is Parameter For?	Value
TIMER_A0_BASE	Which timer are you using?	TA0
TIMER_A_CAPTURECOMPARE_REGISTER_2	Which CCR is being configured?	CCR2
TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE	Enable the CCR interrupt?	CC2IE = 1
TIMER_A_OUTPUTMODE_SET_RESET	How should the output signal be handled?	OUTMOD=0x3
0xBEEF	What 'compare' value will be written to CCR2?	CCR2 = 0xBEEF

The OutputMode setting will be configured using the "Set/Reset" mode (which correlates to the value 0x3). Once again, with so many different output mode choices, we'll defer the full explanation of this until the next topic.

Summary of Timer Setup Code – Part 2

Here's a summary of the timer setup code we have looked at thus far.



Part 1 of our code configures the timer/counter; i.e. the main element of Timer_A.

Part 2 configures the various Capture/Compare registers (CCR). Due to limited space on the slide we have only included the initCompare function for CCR2. In a real application, you might use all of the CCR registers – in which case, our `initTimerA0()` function would become a lot longer.

Before we move onto Part 3 of our timer configuration code, let's spend a few pages explaining the 8 different output mode options available when configuring Capture/Compare Registers.

Output Modes

As you may have already seen, each CCR register has its own associated pin. For CCR1 on Timer0 this pin would be named “TA0.1”. Depending upon which mode you put the CCR into; this pin can be used as an input (for Capture) or an output (for either Capture or Compare).

When the pin is used as an output, its value is determined by the OUT bit-field in its control register. The exact details for this are TA0.1 = TA0CCTL1.OUT. (Sometimes you’ll just see this OUT bit abbreviated as OUT1.)

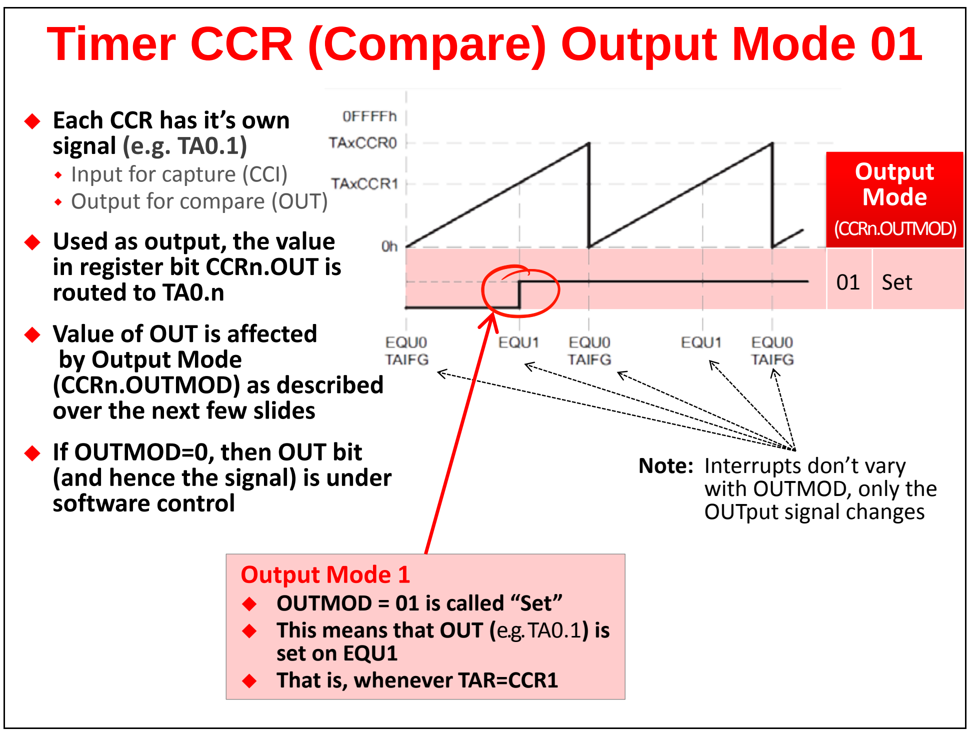
Besides routing the CCR OUT signal to a pin, it can also be used by other MSP430 peripherals. For example, on some devices the A/D converter could be triggered by the timer directly.

So, what is the value of OUT for any given CCR register?

The value of OUT is determined by the OutputMode, as we discussed earlier. (Each CCR control register has its own OUTMOD bit-field). This setting tells the OUT bit how to react as each compare or capture occurs. As previously stated, there are 8 different OutputMode choices.

For example, setting OUTMOD = 0 mean it’s not changed by the timer’s hardware. That is, it’s under software control. You can set OUT to whatever you like by writing to it in the CCRx control register.

What happens to OUT when OUTMOD = 1 (“Set” mode)?



As we can see from the diagram above, when the timer/counter (TAR) counts up to the value in CCR1 (i.e. TAR = CCR1), then a valid comparison is true.

The enumeration for OUTMOD = 1 is called *“Set”*; whenever TAR=CCR1, then OUT will be “Set” (i.e. OUT = 1). In fact, OUT will remain = 1 until the CCR is reconfigured.

Why use “Set” mode? You might find this mode useful in creating a one-shot type of signal.

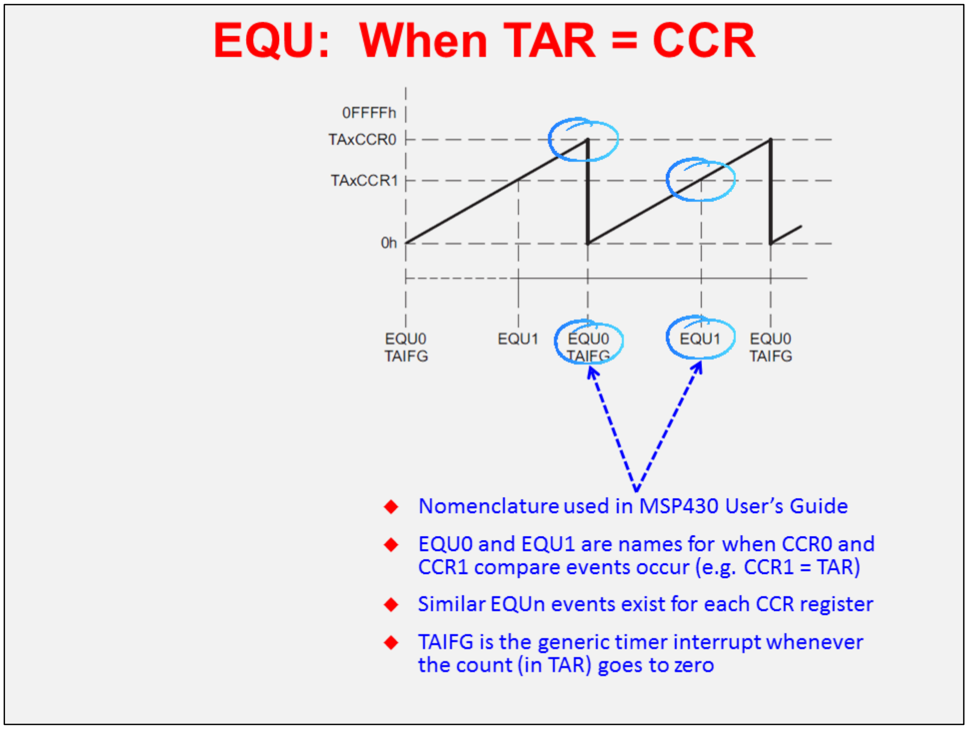
EQU

Before we examine OutputMode 2, let's consider the nomenclature used in the MSP430 User's Guide.

Apparently, there is an EQU (equate) signal inside the timer for each CCR. For example, the equate signal for CCR1 would be called EQU1. While these EQU values cannot be read directly from any of the timer control registers, the documentation makes use of them to describe when a comparison becomes true.

Therefore, when the timer counter (TAR) becomes equal to a compare register (CCR), the associated EQU signal becomes true.

This can be seen in the following diagram captured from the TIMER_A documentation. Notice how EQU0 becomes true when TAR=CCR0; likewise, EQU1 becomes true when TAR=CCR1.



OUTMOD = 2 (“Toggle/Reset” mode)

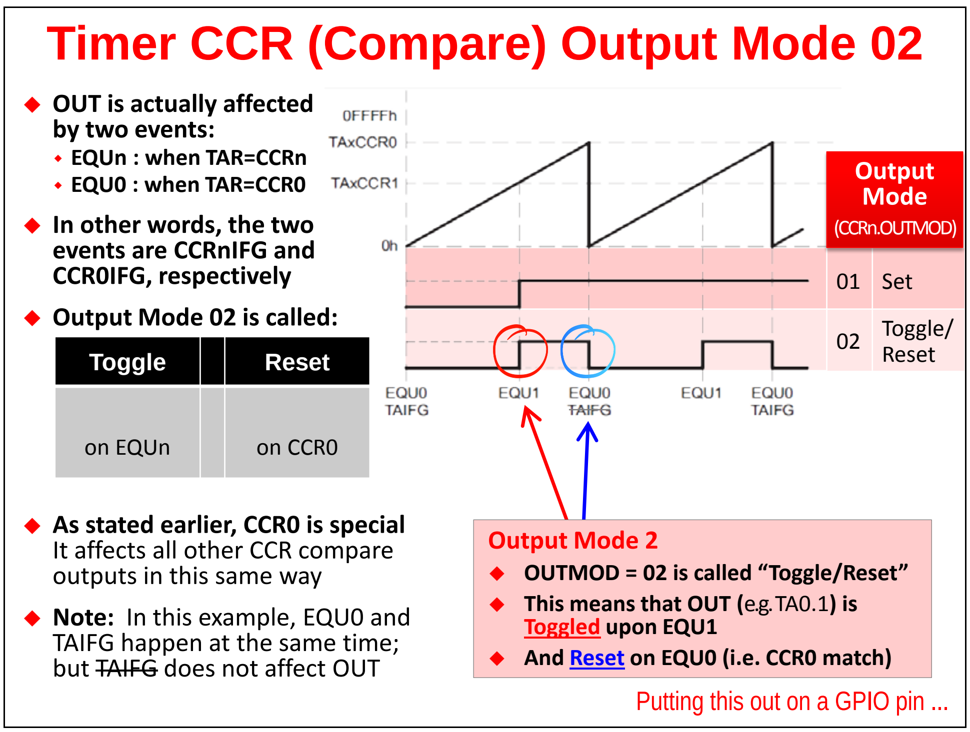
OutputMode 2 is a bit more interesting than the previous output modes. Notice how this mode is called “Toggle/Reset”. Each of these names corresponds to a different event.

- Toggle - This means that OUT_n should be toggled whenever $TAR=CCR_n$
- Reset - This implies that OUT=0 (i.e. reset) whenever TAR=CCR0

In other words, when the OutputModes are defined by two names, the first one dictates the value of OUTn whenever the $TAR=CCR_n$ (i.e. whenever EQU_n becomes true). The second name describes what happens to OUT_n whenever TAR=CCR0.

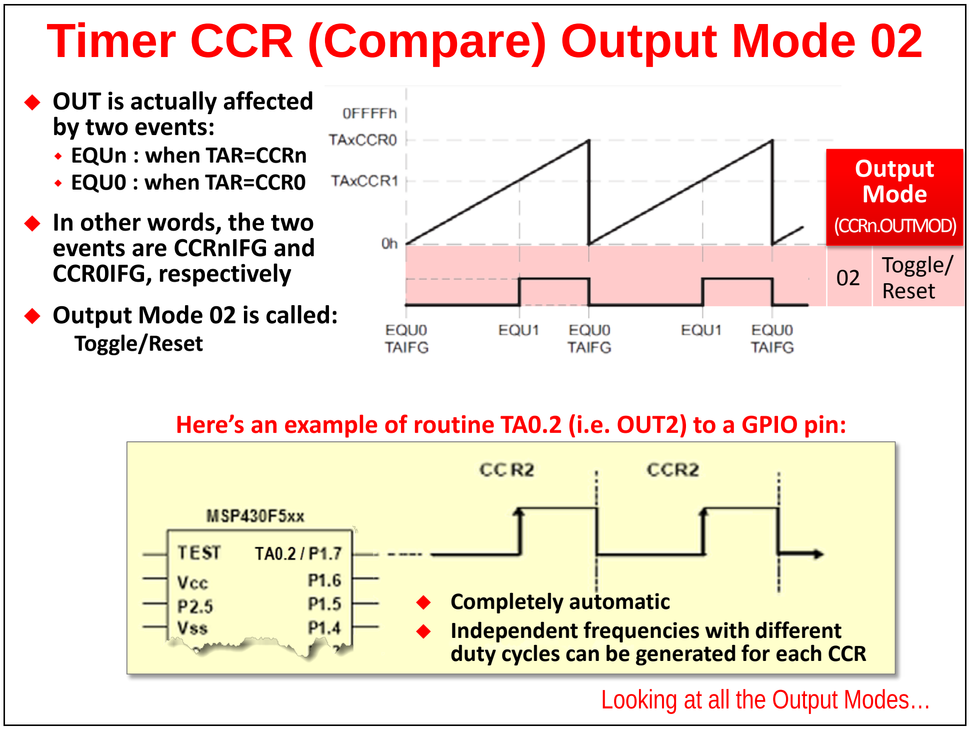
Note: Remember what we said earlier, CCR0 is often used in a special way. This is another example of how CCR0 behaves differently than the rest of the CCR’s.

Looking at the diagram below, we can see that in OutputMode 2, the OUT1 signal appears to be a pulse whose duty cycle (i.e. width) is proportional to the difference between CCR0 and CCR1.



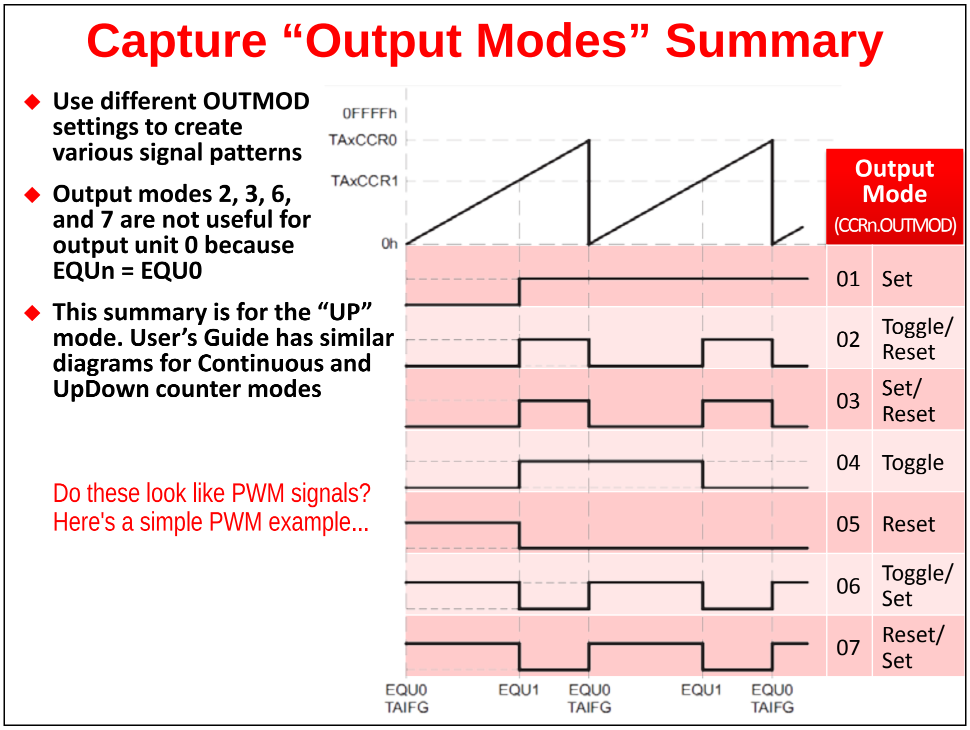
By showing both OUTMOD=1 and OUTMOD=2 in the same diagram, you can see how the value of OUT_n can be very different depending upon the OutputMode selected.

Routing the OUT signal to a pin, as shown here, lets us drive external hardware directly from the output of the timer. *(In fact, we'll use this feature to let the timer directly drive an LED during one of the upcoming lab exercises.)*



Summary of Output Modes

While we have only studied a couple of the output modes, we hope you will be able to decipher the remaining modes based on their names. Here is a comparison of all the different OUTput waveforms based upon the value of OUTMOD.



Point of Clarification – Only use modes 1, 4, and 5 for CCR0

The second bullet, in the diagram above, states that four of the Output Modes (2, 3, 6, and 7) are not useful when you are working with CCR0.

Why are they not useful?

All four of these OutputModes include two actions:

- One action when: CCRn=TAR
- A second action when: CCR0=TAR

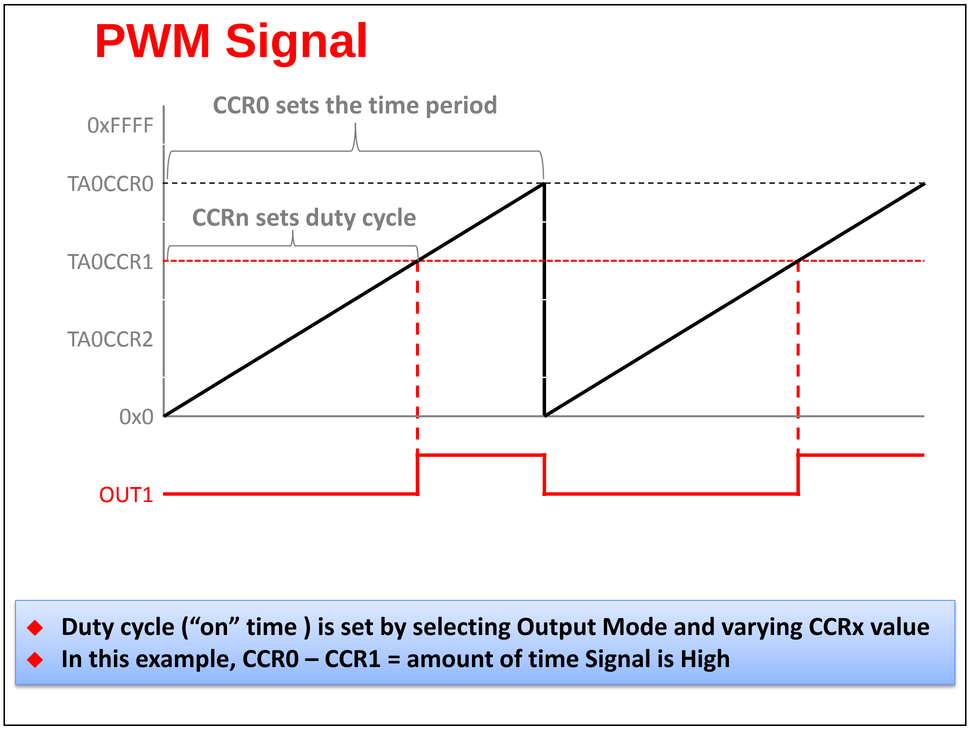
In this case, though, CCRn = CCR0. That means these modes could be trying to change OUT0 in two different ways at the same time.

Bottom Line: When using CCR0, only set OUTMOD to 0, 1, 4, or 5.

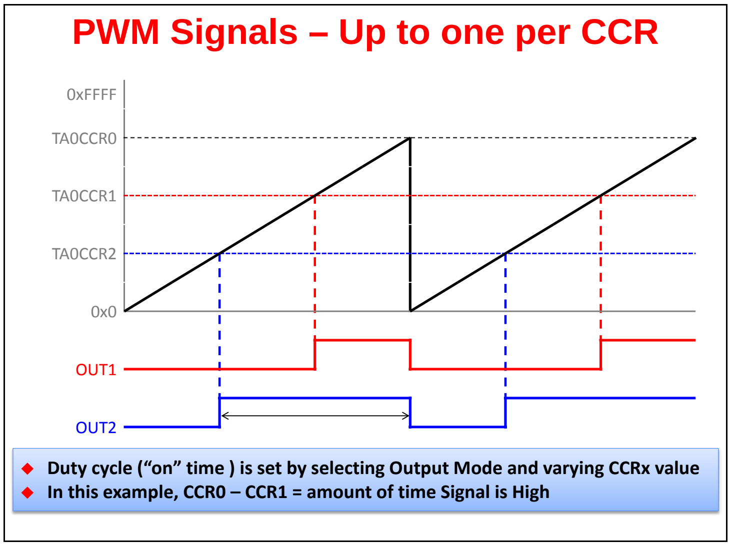
PWM anyone?

PWM, or pulse-width modulation, is commonly used to control the amount of energy going into a system. For example, by making the pulse widths longer, more energy is supplied to the system.

Looking again at the previous example where OUTMOD = 2, we can see that by changing the difference between the values of CCR0 and CCRn we can set the width of OUTn.



In the case of the MSP430, any timer can generate a PWM waveform by configuring the CCR registers appropriately. In fact, if you are using a Timer_A5, you could output 4 or 5 different PWM waveforms.



3. Clear Interrupt Flags and TIMER_A_startTimer()

Part 3 of our timer configuration code is for clearing the interrupt flags and starting the timer.

As described earlier in the workshop, you are not required to clear interrupt flags before enabling an interrupt, but once again, this is common practice. In Part 3 of the example below, we first clear the Timer flag (TA0IFG) using the function call provided by DriverLib. Then, we clear all the CCR interrupts using a single function; notice that the "+" operator tells the function that we want to clear both of these IFG bits.

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

```
#include <driverlib.h>

void initTimerA0(void) {
  // Setup TimerA0 in Up mode with CCR2 Compare
  TIMER_A_configureUpMode( TIMER_A0_BASE,
     TIMER_A_CLOCKSOURCE_SMCLK,
     TIMER_A_CLOCKSOURCE_DIVIDER_1,
     TIMER_PERIOD,
     TIMER_A_TAIE_INTERRUPT_ENABLE,
     TIMER_A_CCIE_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
```

Old API – Slide will be updated on next workshop revison

We conclude the code for Part 3 by starting the timer. The start function only has two parameters:

- It's probably obvious that you need to specify which timer that needs to be started.
- The other parameter specifies, once again, the count mode for the timer's counter.

Warning!

Did we get your attention? The timer "start" function ended up being one of the biggest problems during the development of this workshop.

As dumb as it sounds, we missed the fact that you need to set the counter mode (e.g. "UP") in this function. When we cut/pasted this function from another example, we never thought to change this parameter.

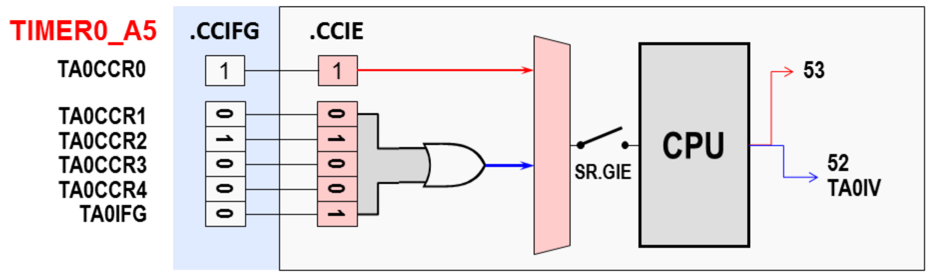
Why, because we thought it had already been specified by using the TIMER_A_configureUpMode() function. Well, we found out the hard way that you need to do both. Use the correct function AND specify the correct count mode in the *start* function.

4. Interrupt Code (Vector & ISR)

The last part of our timer code is actually a review since interrupts were covered, in detail, in a previous workshop chapter.

Timer0_A5 Interrupts Review

INT Source	IFG	IV Register	Vector Address	Loc'n
Timer A (CCIFG0)	TA0CCR0.CCIFG	none	TIMER0_A0_VECTOR	53
Timer A	TA0CCR1.IFG1…TA0CCR4.IFG	TA0IV	TIMER0_A1_VECTOR	52



- In the interrupts chapter, we learned that most MPS430 interrupts are grouped together and share an interrupt vector, although a few have their own dedicated vector
- Timers A and B have two vectors: one for CCR0 and the other shared
- When the CPU responds to TIMER0_A0_VECTOR, the CCR0IFG is auto cleared
- In the TIMER0_A1_VECTOR ISR, reading **TA0IV** register returns the associated highest priority pending interrupt and clears it's IFG bit

Remember, TIMER_A has two interrupt vectors: one dedicated to CCR0; another shared by TAIFG and all the other CCR's. Below, we provide a simple example of handling both.

Timer Code Example (Part 4 – ISR's)

CCR0 ISR

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

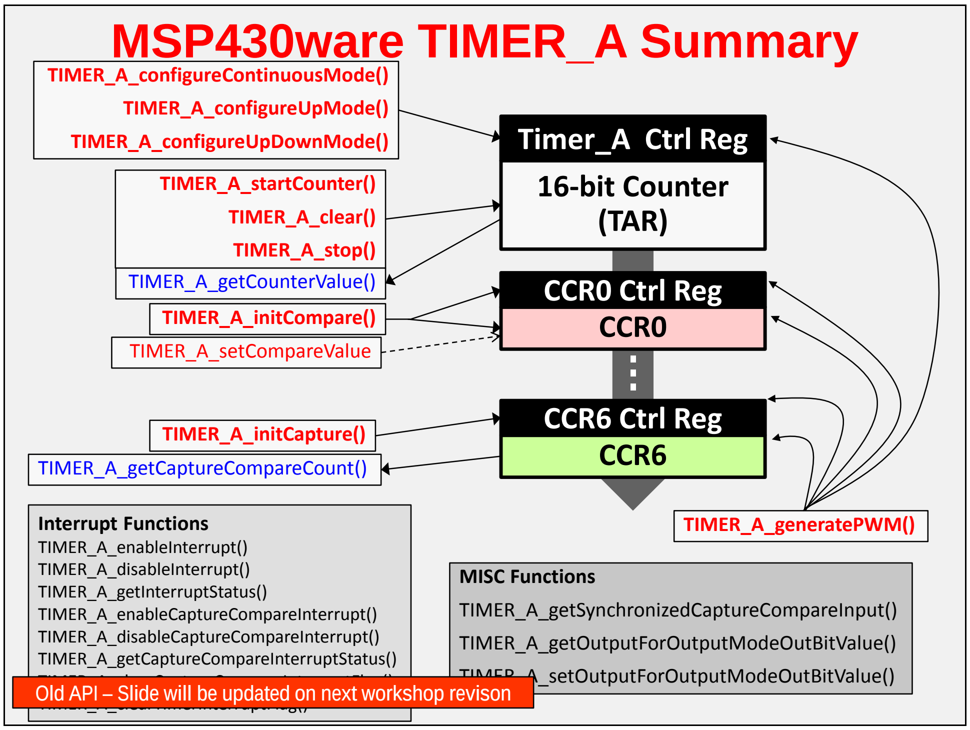
ISR for CCR2 and TA0IFG

```
#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:                     // CCR2 IFG
        GPIO_toggleOutputOnPin(…);
              break;
   case 0x06: break;              // CCR3 IFG
   case 0x08: break;              // CCR4 IFG
   case 0x0A: break;              // CCR5 IFG
   case 0x0C: break;              // CCR6 IFG
   case 0x0E:                     // TA0IFG
        GPIO_toggleOutputOnPin(…);
        break;
   default:   _never_executed();
  }
}
```

TIMER_A DriverLib Summary

This diagram attempts to summarize the functions found in the TIMER_A module of the MSP430ware Driver Library.

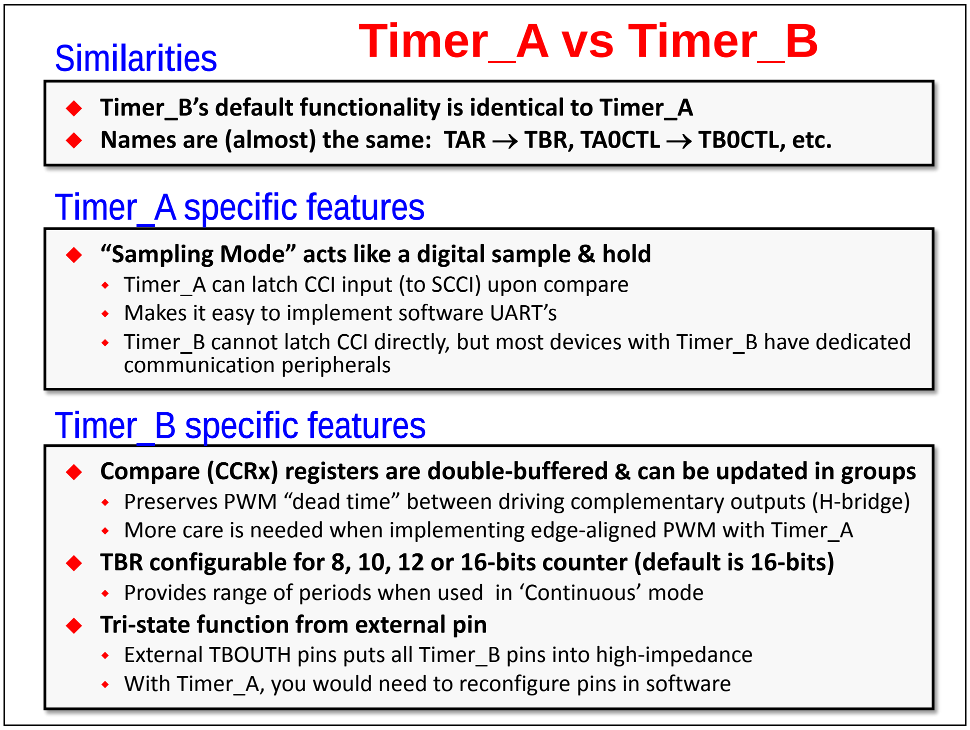
Many of the functions have arrows pointed to/from the three main parts of the timer peripheral: TAR (the main timer/counter); CCR (used for Compare); and CCR (used for Capture). The arrows indicate whether the function reads or writes the associated registers.



The bottom of the slide contains two boxes: one summarizes the Interrupt related functions while the other contains three functions that read/write the input and output bit values.

Differences between Timer's A and B

The Timer_A and Timer_B peripherals are very similar. The following slide highlights the few differences between them.



Hint: For a more complete understanding of these differences, we highly recommend that you refer to *MSP430 Microcontroller Basics*. John Davies does a great job of describing the differences between these timers. Furthermore, his discussion of generating PWM waveforms using these timers is extremely good. If you've never heard of the differences between *edge-aligned* and *centered* PWM waveforms, check out his MSP430 book.

MSP430 Microcontroller Basics by John H. Davies, (ISBN-10 0750682760) **Link↗**

Notes

Lab 6 – Using Timer_A

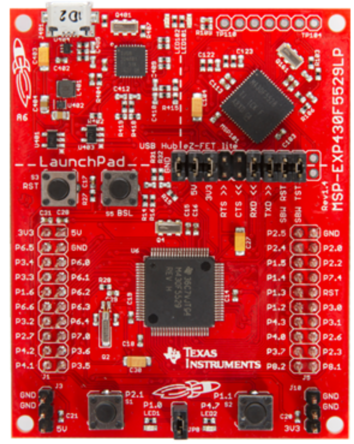
Lab 6 – Using Timer_A

- **Time for the lab prep Worksheet:**
  - What time is it?
  - Capture vs Compare
  - 4 steps to timer programming
  - Simple PWM generation
- **Lab 6a – Simple Timer Interrupt**
  - Create a CCR0 interrupt with the timer counting in Continuous Mode
  - ISR toggles LED
- **Optional Exercises**

  Lab 6b – Timer using Up Mode
  - Similar to Lab6a, but using Up mode

  Lab 6c – Timer with Directly Driven LED
  - Similar to Lab6b, but with the timer directly driving the LED

  Lab 6d – Simple PWM Signal
  - Alter the brightness of the LED by changing the PWM duty cycle



Time:

Worksheet – 15 mins

Labs – 30 mins

Note: The solutions exist for all of these exercises, but the instructions for Lab 6d are not yet included. These will appear in a future version of the course.

Lab Topics

Timers 6-33

Lab 6 – Using Timer_A 6-35

Lab 6a – Simple Timer Interrupt 6-37
Lab 6a Worksheet 6-37
Lab 6a Procedure 6-42
Edit *myTimers.c* 6-43
Debug/Run 6-44

(Extra Credit) Lab 6b – Timer using Up Mode 6-45
Lab 6b Worksheet 6-45
File Management 6-48
Change the Timer Setup Code 6-49
Debug/Run 6-49
Archive the Project 6-50
Timer_B (Optional) 6-51

(Extra Credit) Lab 6c – Drive GPIO Directly From Timer 6-52
Lab 6c Abstract 6-52
Lab 6c Worksheet 6-53
File Management 6-57
Change the GPIO Setup 6-57
Change the Timer Setup Code 6-58
Debug/Run 6-59
(Optional) Lab 6c – Portable HAL 6-63

(Optional) Lab 6d – Simple PWM (Pulse Width Modulation) 6-64

Chapter 6 Appendix 6-65

Lab 6a – Simple Timer Interrupt

Similarly to `lab_05a_buttonInterrupt`, we want to toggle an LED based upon an interrupt. In this case, though, we'll use TIMER_A to generate an interrupt; during the interrupt service routine, we'll toggle the GPIO value that drives an LED on our Launchpad board.

As we write the ISR code, you should see that TIMER_A has two interrupts:

– One is dedicated to CCR0 (capture and compare register 0).
– The second handles all the other timer interrupts

This first TIMER_A lab will use the main timer/counter rollover interrupt (called TA0IFG). As with our previous interrupt lab (with GPIO ports), this ISR should read the TimerA0 IV register (TA0IV) and decipher the correct response using a switch/case statement.

Lab 6a Worksheet

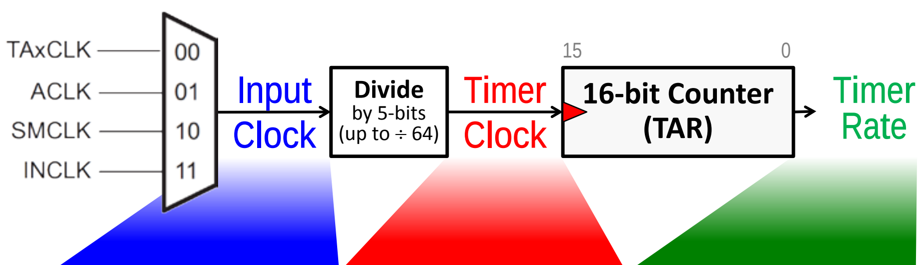
Goal: Write a function setting up Timer_A to generate an interrupt every two seconds.

1. **How many clock cycles does it take for a 16-bit counter to 'rollover'?** *(Hint: 16-bits)*

____

2. **Our goal is to generate a two second interrupt rate based on the timer clock input diagramed above.**

Using `myClocks.c` provided for this lab, we created a table of example clock & timer rates:



Input Clock		Timer Clock		Timer Rate	
Source	Frequency	Divider	Resolution	Frequency	Period
SMCLK	8 MHz	1	⅛ µs	122 Hz	8 ms
SMCLK	8 MHz	8	1 µs	15 Hz	66 ms
ACLK	32 KHz	2	62 µs	½ Hz	4 s
ACLK	32 KHz	8	240 µs	$\frac{1}{16}$ Hz	16 s

Pick a source clock for the timer. *(Hint: At 2 seconds, a slow clock might work best.)*

Clock input (circle one): **ACLK** **SMCLK**

3. **Calculate the Timer settings for the clocks & divider values needed to create a timer interrupt every 2 seconds. (That is, how can we get a timer period rate of 2 seconds.)**

   Which clock did you choose in the previous step? Write its frequency below and then calculate the *timer period rate*.

   Input Clock: ACLK running at the frequency = ______________

   Timer Clock = ______________ ÷ ______________ = ______________
   input clock frequency — *timer clock divider* — *timer clock freq*

   Timer Rate = ______________ X 65536 = ______________
   timer clock period (i.e. 1 / timer clock freq) — *counts for timer to rollover* — *timer rate period*

   I.E. 64K

4. **Which Timer do you need to use for this lab exercise?**

   In a later lab exercise we will output the timer directly to a BoosterPack pin. Unfortunately, the two Launchpad's map different timers to their BoosterPack pinouts. (This is due to the 'FR5969 having few pins and only using the 20-pin BoosterPack layout; versus the 40-pin XL layout for the 'F5529.)

   Here are the recommended timers:

Launchpad	Timer	Short Name	Timer's Enum
'F5529	Timer0_A3	TA0	TIMER_A0_BASE
'FR4133	Timer0_A3	TA0	TIMER_A0_BASE
'FR5969	Timer1_A5	TA1	TIMER_A1_BASE

   Write down the timer enumeration you need to use: **TIMER_ _________ _BASE**

5. **Write the *TIMER_A_initContinuousMode()* function.**

   The first part of our timer code is to setup the Timer control registers (TAR, TACTL). Fill in the code to specify the clock and dividers we just figured out. Also, enable the TAIE interrupt and clear the counter – but don't start the timer, yet.

```
Timer_A_initContinuousModeParam initContParam = { 0 };

  initContParam.clockSource = _____________________________________________;

  initContParam.clockSourceDivider =  ______________________________________;

  initContParam.timerInterruptEnable_TAIE = _________________________________;
  initContParam.timerClear = TIMER_A_DO_CLEAR;
  initContParam.startTimer = false;
Timer_A_initContinuousMode(TIMER_ _____ _BASE, &initContParam );
```

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

6. **Skip this step … it's not required.**

   We outlined 4 steps to configure Timer_A. The second step is where you would set up the Capture and Compare features. Since this exercise doesn't need to use those features, you can skip this step.

7. **Complete the code to for the 3rd part of the "Timer Setup Code".**

   The third part of the timer setup code includes:

   – ~~Enable the interrupt (IE)~~ *… we don't have to do this, since it's done by the TIMER_A_configureContinuousMode() function (from question 5 on page* 6-*39).*

   – Clear the appropriate interrupt flag (IFG)

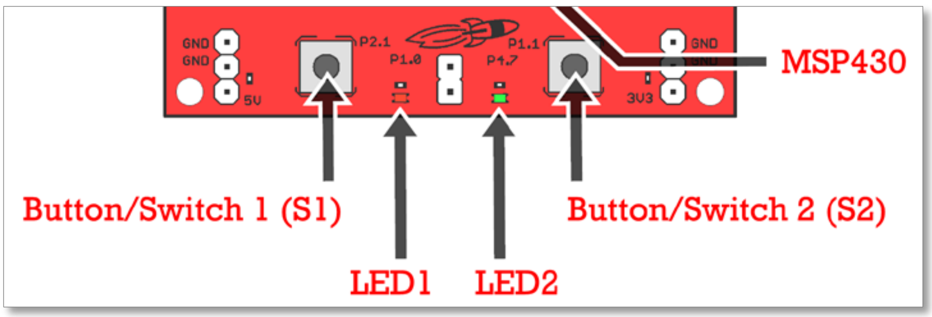
   – Start the timer

```
// Clear the timer interrupt flag

_______________________________________ ( TIMER_____BASE );    // Clear TA0IFG


// Start the timer

_______________________________________ (                      // Function to start timer

        TIMER_____BASE,                                         // Which timer?

        _______________________________                         // Run in Continuous mode
```

8. Change the following interrupt code to toggle LED2 when Timer_A rolls-over.

Hint:

	'F5529 LP	'FR5969 LP	Color
LED1 (Jumper)	P1.0	P4.6	**Red**
LED2	P4.7	P1.0	**Green**
Button 1	P2.1	P4.5	
Button 2	P1.1	P1.1	



a) Fill in the details for your Launchpad.

Port/Pin number for LED2: `Port _____, Pin _____`

Timer Interrupt Vector: `#pragma vector = ________________ _VECTOR`

Timer Interrupt Vector register: ________________

(Hint: We previously used `P1IV` *for GPIO Port 1)*

b) Here is the interrupt code that exists from a previous exercise, change it as needed.

Mark up the following code – crossing out what is old or not needed and writing in the modifications needed for our timer interrupt.

```
#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();
     }
}
```

Please verify your answers before moving onto the lab exercise.

Lab 6a Procedure

File Management

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

   If some are open, we recommend closing them.

2. **Import the `lab_06a_timer` project.**

   We have already created the initial lab project for you to import.

   `C:\msp430_workshop\<target>\lab_06a_timer`

   It doesn't matter whether you copy this project into your workspace or not. If you "copy" it into your workspace, the original files will remain untouched. If do not copy, but rather "link" to the project, you will only have one set of files and any changes you make inside of CCS will be reflected in the *C:\msp430_workshop\<target>\lab_06a_timer* directory.

3. **Briefly examine the project files**

   This project uses code we have written earlier in the workshop, though we have partitioned some of this code into separate files:

   - `myGpio.c`
     - The LED pins are configured as outputs and set to Low.
     - For the 'FR5969, the LFXT pins are set as clock inputs; and, the pins are unlocked.
   - `myClocks.c`
     - For 'F5529 users, this is the same code you wrote in the *Clocks* chapter.
     - For 'FR5969 users, we used the file from Lab4c so that ACLK uses the 32KHz crystal rather than VLO. Also, MCLK and SMCLK are set to 8MHz.

Edit *myTimers.c*

4. Edit the `myTimers.c` source file.

We want to setup the timer to generate an interrupt two seconds. The TAIFG interrupt service routine will then toggle LED2 on/off.

Worksheet Question #5 (page 6-39)

Worksheet Question #7

Worksheet Question #8

```
void initTimers(void)
{
    // 1. Setup Timer (TAR, TACTL)in Continuous mode using ACLK
    TIMER_A_ _______________________(
        TIMER_A__BASE,                          // Which timer
        TIMER_A_ ________________,              // Which clock
        TIMER_A_ _____________________,         // Clock divider
        TIMER_A_ _____________________,         // Enable INT on rollover?
        TIMER_A_DO_CLEAR                        // Clear timer counter
    );

    // 2. Setup Capture & Compare features
    //    This example does not use these features

    // 3. Clear/enable flags and start timer
    TIMER_A_ _______________________( TIMER_A1_BASE );  // Clear Timer Flag

    TIMER_A_startCounter(
        TIMER__BASE,
        TIMER_A_ _______________                // Which timer mode
    );
}

//****** Interrupt Service Routine *****************************************
#pragma vector=TIMER1_A1_VECTOR
__interrupt void timer1_ISR (void)
{
    // 4. Timer ISR and vector
    switch( __even_in_range( _____, 14 )) {     // Read timer IV register
     case  0: break;                // None
     case  2: break;                // CCR1 IFG
     case  4: break;                // CCR2 IFG
     case  6: break;                // CCR3 IFG
     case  8: break;                // CCR4 IFG
     case 10: break;                // CCR5 IFG
     case 12: break;                // CCR6 IFG
     case 14:                       // TAR overflow
              // Toggle LED2 (Green) on/off
              GPIO_toggleOutputOnPin( _____________, _______________ );
              break;
     default: _never_executed();
    }
}
```

5. **Modify the *Unused Interrupts* source file.**

   Since our timer code uses an interrupt, we need to comment out its associated vector from the `unused_interrupts.c` file.

6. **Build your code and repair any errors.**

Debug/Run

7. **Launch the debugger.**

   Notice that you may still see the clock variables in the Expressions pane. This is convenient, if you want to double-check the MSP430 clock rates.

8. **Set a breakpoint inside the ISR.**

   We found it worked well to set a breakpoint on the 'switch' statement (*in the* `myTimer.c` *file*).

9. **Run your code.**

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

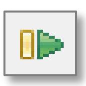
   If the breakpoint occurred, skip to the next step …

   If you did not reach the breakpoint inside your ISR, here are a few things to look for:

   – *Is the interrupt flag bit (IFG) set?*

   – *Is the interrupt enable bit (IE) set?*

   – *Are interrupts enabled globally?*



10. **If the breakpoint occurred, then resume running again.**

    You should always verify that your interrupts work by taking more than 'one' of them. *A common cause of problems occurs when the IFG bit is not cleared. This means you take one interrupt, but never get a second one.*

    In our current example, reading the TA1IV (or TA0IV for 'F5529 users) should clear the flag, so the likelihood of this problem occurring is small, but sometimes the problem still occurs due to a logical error while coding the interrupt routine.

11. **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.

12. **Once you've got the LED toggling, you can terminate your debug session.**

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

In this timer lab we switch our code from counting in the "Continuous" mode to the "Up" mode. This gives us more flexibility on the frequency of generating interrupts and output signals.

From the discussion you might remember that TIMER_A has two interrupts:

- One is dedicated to CCR0 (capture and compare register 0).
- The second handles all the other timer interrupts

In our previous lab exercise, we created an ISR for the grouped (non-dedicated) timer interrupt service routine (ISR). This lab adds an ISR for the dedicated (CCR0 based) interrupt.

Each of our two ISR's will toggle a different colored LED.

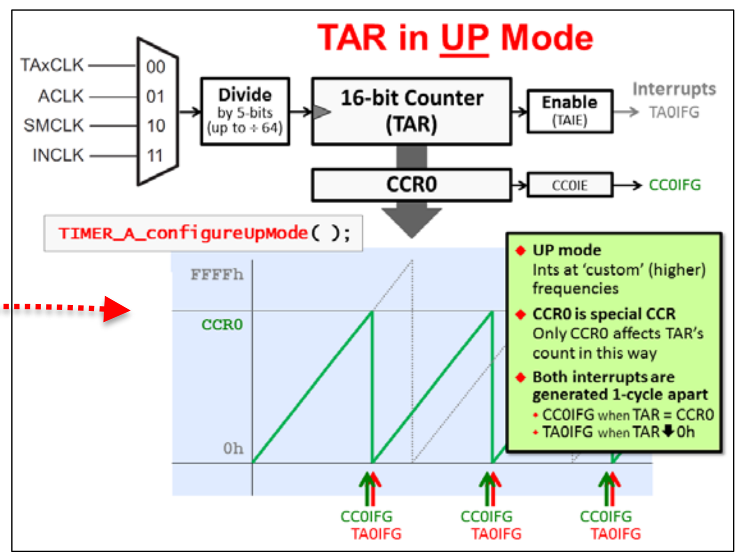
The goal of this part of the lab is to:

```
//    Timer_A in Up mode using ACLK
//    Toggle LED1 on/off every second using CCR0IFG
//    Toggle LED2 on/off every second using TA0IFG (or TA1IFG for 'FR5969)
```

Lab 6b Worksheet

1. Calculate the timer period (for CCR0) to create a 1 second 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?

Timer Clock	=	32 KHz	÷	1	=	32 KHz
		input clock frequency		*timer clock divider*		*timer clock freq*
Timer Rate	=	1/32768	X		=	1 SECOND
		timer clock period (i.e. 1 / timer clock freq)		*timer counter period (i.e. CCR0 value)*		*timer rate period*

2. **Complete the *TIMER_A_initUpMode()* 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_initUpModeParam initUpParam = { 0 };
  initUpParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initUpParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;

  initUpParam.timerPeriod =______________________________; // (calculated in previous question)
  initUpParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initUpParam.captureCompareInterruptEnable_CCR0_CCIE =

        ________________________________________________; // Enable CCR0 compare interrupt
  initUpParam.timerClear = TIMER_A_DO_CLEAR;
  initUpParam.startTimer = false;

Timer_A_initUpMode(TIMER____BASE, &initUpParam ););
```

3. **Modifying our previous code, we need to clear both interrupts and start the timer.**

   We copied the following code from the previous exercise. It needs to be modified to meet the 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 'startCounter' function. Is there anything that needs to change when switching from Continuous to Up mode?*

```
// Clear the timer flag and start the timer
Timer_A_clearTimerInterruptFlag( TIMER____BASE );           // Clear TA0IFG

______________________________________________ (            // Clear CCR0IFG

      TIMER______BASE,

      ______________________________________________ );

Timer_A_startCounter( TIMER____BASE,                         // Start timer in

     TIMER_A_______MODE );                                   // ____ mode
```

4. Add a second ISR to toggle the LED1 whenever the CCR0 interrupt fires.

On your Launchpad, what Port/Pin number does the LED1 use? ______________

Hints:

- ❑ *What port/pin does your LED1 use? Look back at question 8 (page 6-40).*
- ❑ *Look at the unused_interrupts.c file for a list of interrupt vector symbol names.*

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

```
#pragma vector= _________________________________
__interrupt void ccr0_ISR (void)
{
    // Toggle the LED1 on/off

   ______________________________________________________________________

}
```

Please verify your answers before moving onto the lab exercise.

File Management

5. Copy/Paste lab_06a_timer to lab_06b_upTimer.

a) In CCS Project Explorer, *right-click* on the lab_06a_timer project and select “Copy”.

b) Then, click in an open area of Project Explorer pane and select “Paste”.
This will create a new copy of your project inside the Workspace directory.

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.

---

6. Close the previous project: lab_06a_timer

7. Delete the old, readme file and import the new one.

You can import the new readme text file from this folder:

C:\msp430_workshop\<*target*>\lab_06b_upTimer

8. Make sure the project is selected (i.e. active) and build it to verify no errors were introduced during the copy.

Change the Timer Setup Code

In this part of Lab 6, we will be setting up TimerA in Up Mode.

9. **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. (Question2, Page 6-46).

10. **Modify the rest of the timer set up code, where we clear the interrupt flags, enable the individual interrupts and start the timer.**

    Please refer to the Lab Worksheet for assistance. (Question 3, Page 6-46).

11. **Add the new ISR we wrote in the Lab Worksheet to handle the CCR0 interrupt.**

    When this step is complete, you should have two ISR's in your `main.c` file.

    Please refer to the Lab Worksheet for assistance. (Question 4, Page 6-47).

12. **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 *Interrupts* chapter lab exercise.)

13. **Build the code to verify that there are no syntax (or any other kind of) errors; fix any errors, as needed.**

Debug/Run

Follow the same basic steps as found in the previous lab for debugging.

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

15. **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? ____

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

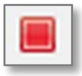
    An easy way to quickly remove all of the breakpoints is to open the Breakpoints View window:

    ```
    View → Breakpoints
    ```

    Then click the Remove all Breakpoints toolbar button.

Archive the Project

Thus far in this workshop, we have imported projects from archives … but we haven't asked you to create an archive, yet. It's not hard, as you'll see.

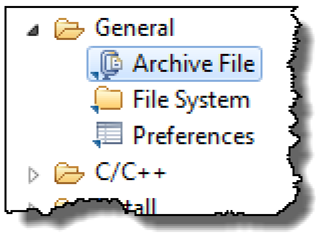


17. Terminate the debugger, if it's still open.

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

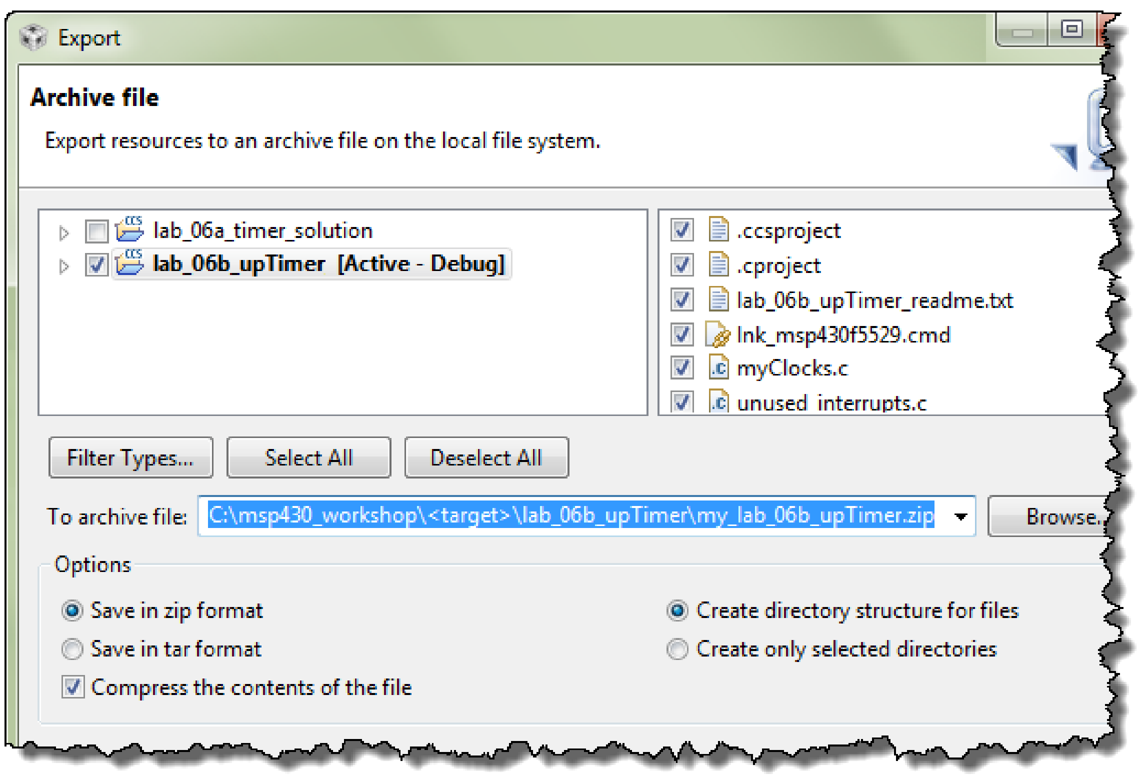
– `Right-click the project and select 'Export'`

– `Select 'Archive File' for export, then click Next`



– Fill out the dialog as shown below, choosing: the 'upTimer' lab; "Save in zip format", "Compress the contents of the file"; and the following destination:

`C:\msp430_workshop\<target>\lab_06b_upTimer\my_lab_06b_upTimer.zip`



Timer_B (Optional)

Note: Since the ‘FR4133 does not include the Timer_B peripheral, you can skip this exercise if you’re using the MSP-EXP430FR4133 Launchpad.

Do you remember during the discussion that we said Timer_A and Timer_B were very similar? In fact, the timer code we have written can be used to operate Timer_B … with 4 simple changes:

- **It’s a different API … but not really.**

  Rather than using the TIMER_A module from DriverLib, you will need to use TIMER_B; unless you’re using one of the few unique features of TIMER_B, the rest of the API is the same. In other words, you can carefully search and replace TIMER_A for TIMER_B.

- **Specify a different timer.**

  Since you’re using a different timer, you need to specify a different timer ‘base’. For either the ‘F5529 or ‘FR5969 you should use TIMER_B0_BASE to specify the timer instance you want to use.

- **You need to use the TIMER_B interrupt vector.**

  This changes the #pragma line where we specify the interrupt vector.

- **You need to use the TIMER_B interrupt vector register.**

  You need to read the TB0IV register to ascertain which TIMER_B flag interrupted the CPU.

All of these are simple changes. Try implementing TIMER_B on your own.

Note: While we don’t provide step-by-step directions, we did create a solution file for this challenge.

(Extra Credit) Lab 6c – Drive GPIO Directly From Timer

Lab 6c Abstract

This lab is a minor adaptation of the TIMER_A code in the previous exercise. The main difference is that we'll connect the output of Timer_A CCR2 (TA0.2 or TA1.2) directly to a GPIO pin.

We are still using Up mode, which means that CCR0 is used to reset TAR back to 0. We needed to choose another signal to connect to the external pin… we arbitrarily 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. As we alluded to earlier in the chapter, in the case of Timer_A, the Launchpad's route different timer pins to the BoosterPack pin-outs.

Here's an excerpt from the 'F5529 lab solution:

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

And a similar statement from the 'FR5969 lab solution:

```
// When running this lab exercise, you will need to pull the J6 jumper and
// use a jumper wire to connect signal from pin ____ (on boosterpack pinouts) to
// J6.2 (bottom pin) of the LED1 jumper ... this lets the TA1.2 signal drive
// LED1 directly (without having to use interrupts)
```

And for those of you using the 'FR4133:

```
// When running this lab exercise, you will need to pull the JP1 jumper and
// use a jumper wire to connect signal from pin _____ (on boosterpack pins) to
// JP1.2 (bottom pin) of LED1 jumper ... this lets the TA1.2 signal drive
// LED1 directly (without having to use interrupts)
```

(Note: Later in the lab instructions, we'll show a picture of connecting the jumper wire.)

Lab 6c Worksheet

1. **Figure out which BoosterPack pin will be driven by the timer's output.**

   To accomplish our goal of driving the LED from a timer, we need to choose which Timer CCR register to output to a pin on the device. In the lab abstract (on the previous page) we stated that for this lab writeup, we arbitrarily chose to use CCR2.

   Based on the choice of CCR2, we know that the timer's output signal will be: `TAn.2`.

   We've summarized this information in the following table:

Device	Timer	CCRx	Signal	GPIO Port/Pin	Is Pin on Boosterpack?
'F5529	**TimerA0**	**CCR2**	**TA0.2**		
'FR4133	**TimerA1**	**CCR2**	**TA1.2**		
'FR5969	**TimerA1**	**CCR2**	**TA1.2**		

   Your job is to fill in the remaining two columns for the device that you are using.

   a) Looking at the datasheet, which GPIO port/pin is combined with TA0.2 (or TA1.2)? For example, here we see that P1.1 is combined with TA0.0:

   Look for the correct pin in your device's datasheet and enter it in the table above.

   > **Hint:** *There are a couple places in the datasheet to find this information. We recommend searching your device's datasheet for "TA0.2" or "TA1.2".*

   b) Next, is that signal output to the BoosterPack?

   This information can be found directly from the Launchpad. Look for the silkscreened labels next to each BoosterPack pin. When you find it, write YES/NO in the column above.

   (If you're getting a little older, you may need a magnifying glass to answer this question…or you may need to zoom in on the Launchpad's photo.)

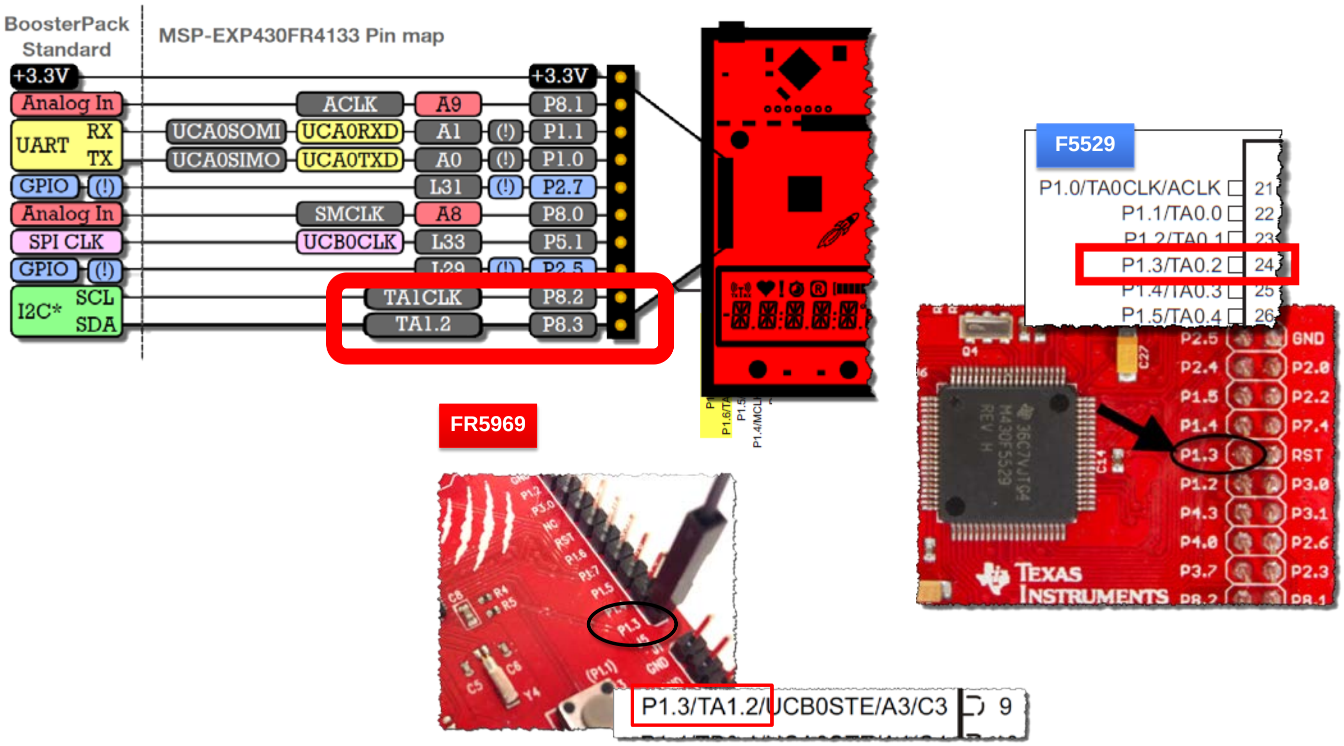
Sidebar – Choosing a Timer For This Exercise

Our choice of TimerA0 (for 'F5529) and TimerA1 (for 'FR5969 & 'FR4133) was not arbitrary. Even further, our choice of CCR2 was not entirely arbitrary.

Bottom line, we wanted to choose a Timer pin that was connected to the BoosterPack pinout since it would make it easy for us to jumper that signal over to LED1.

The problem was that neither board connected the same TimerA outputs to its Boosterpack pinout. In looking carefully at the datasheets for both devices, as well as the Boosterpack pinouts for each Launchpad, we found a timer that we could use. The only issue is that one device mapped TA0.2 to a pin, while the other mapped out TA1.2.

Did you find the correct pins on your Launchpad's BoosterPack?



2. Complete the following function to "select" P1.3 as a timer function (as opposed to GPIO)**.**

Hint: We discussed the port select function in the GPIO chapter. You can also find the details of this function in the Driver Library User's Guide.

F5529

'F5529 Users, here's the function you need to complete:

```
GPIO_setAs______________________________________(
    _______________________,
    _______________________ );
```

FR5969

FR4133

'FR5969 or 'FR4133 Users, your function requires one more argument:

```
GPIO_setA s_____________________________________(
    _______________________,
    _______________________,
    ________________________________________________ );
```

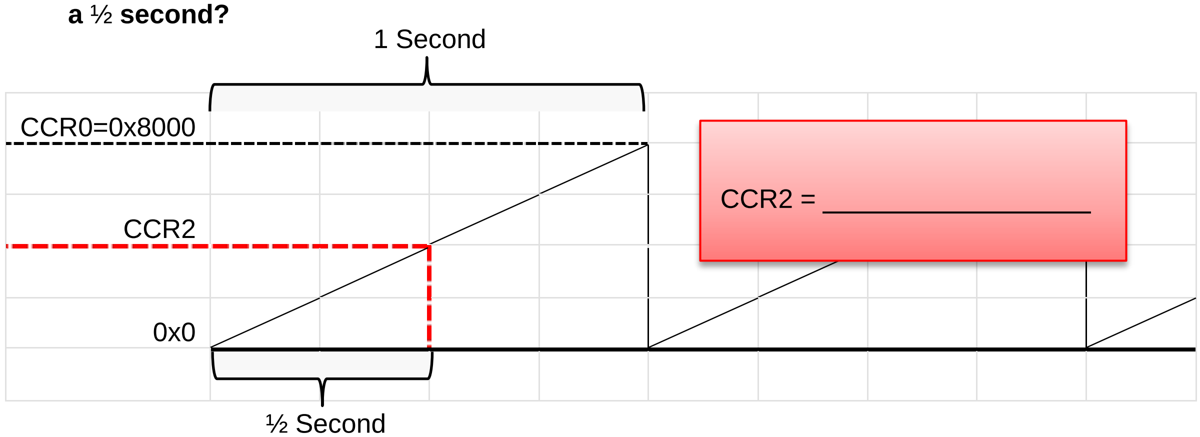
3. **Modify the TIMER_A_configureUpMode() function?**

   Here is the code we wrote for the previous exercise. We only need to make one change to it. Since we will drive the signal directly from the timer, we don't need to generate the CCR0 interrupt anymore.

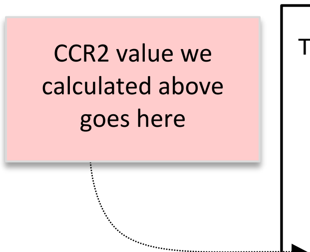
   Mark up the code below to disable the interrupt. *(We'll bet you can make this change without even looking at the API documentation. Intuitive code is one of the benefits of using DriverLib!)*

```
Timer_A_initUpModeParam initUpParam = { 0 };
  initUpParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initUpParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;
  initUpParam.timerPeriod = 0xFFFF / 2;
  initUpParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initUpParam.captureCompareInterruptEnable_CCR0_CCIE = TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE;
  initUpParam.timerClear = TIMER_A_DO_CLEAR;
  initUpParam.startTimer = false;
Timer_A_initUpMode( TIMER____BASE, &initUpParam );
```

4. **What 'compare' value does CCR2 need to equal in order to toggle the output signal at a ½ second?**



5. **Add a new function call to set up Capture and Compare Register 2 (CCR2). This should be added to initTimers().**




```
Timer_A_init___________________  initCcr2Param = { 0 };
  initCcr2Param.compareRegister = _____________________________________________;
  initCcr2Param.compareInterruptEnable = TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE;
  initCcr2Param.compareOutputMode = TIMER_A_OUTPUTMODE_TOGGLE_RESET;
  initCcr2Param.compareValue = _______________________________________;
Timer_A_init___________________( TIMER____BASE, &initCcr2Param );
```

6. **Compare your ISR code from myTimers.c in the previous lab to the code below. What is different in the code shown here?**

   What did we change? ______________________________

   Note, this is the 'F5529 code example. The 'FR5969 uses a slightly different interrupt vector symbol and interrupt vector register.

```
#pragma vector=TIMER0_A1_VECTOR
__interrupt void timer0_ISR(void)
{
     switch(__even_in_range( TA0IV, 14  )) {
            case 0:  break;                    // No interrupt
            case 2:  break;                    // CCR1 IFG
            case 4:                            // CCR2 IFG
                     _no_operation();
                     break;
            case 6:  break;                    // CCR3 IFG
            case 8:  break;                    // CCR4 IFG
            case 10: break;                    // CCR5 IFG
            case 12: break;                    // CCR6 IFG
            case 14: break;                    // TAR overflow
                     GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN7 );
                     break;
            default:   _never_executed();
     }
}
```

   During debug, we will ask you to set a breakpoint on 'case 4'.

   Why should case 4 not occur in our program, and thus, the breakpoint never reached?

   ____

   ____

7. Why is it better to toggle the LED directly from the timer, as opposed to using an interrupt (as we've done in 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 parts A and B of this lab, we will make the changes discussed in the lab worksheet.

5. **Modify the initGPIO function, defining the appropriate pin to be configured for the timer peripheral function.**

   Please refer to the Lab6c Worksheet for assistance. (Question 2, Page 6-54).

Change the Timer Setup Code

6. **Modify the timer configuration function; we are still using 'Up' mode, but we can eliminate one of the interrupts.**

   Please refer to the Lab Worksheet for assistance. (Step 3, Page 6-55).

7. **Add the TIMER_A function to your code that configures CCR2.**

   Please refer to the Lab Worksheet for assistance. (Step 5, Page 6-55).

8. **Delete or comment out the call to clear the CCR0IFG flag.**

   We won't need this because the timer will drive the LED directly – that is, no interrupt is required where we need to manually toggle the GPIO with a function call.

```
TIMER_A_clearCaptureCompareInterruptFlag( TIMER_A0_BASE,
        TIMER_A_CAPTURECOMPARE_REGISTER_0                    //Clear CCR0IFG
);
```

   Then again, it doesn't hurt anything if you leave it in the code… if so, an unused bit gets cleared.

9. **Make the minor modification to the timer isr function as shown in the worksheet.**

   Please refer to the Lab Worksheet for assistance. (Step 6, Page 6-56).

   'FR5969 users – we only showed the 'F5529 code in the worksheet. Please be careful that you do not change the interrupt vector or IV register values in your code. That's not what we're asking you to do in this step.

10. **Build the code verifying there are no syntax errors; fix any as needed.**

Debug/Run

11. Launch the debugger and set three breakpoints inside the two ISR's.

- When we run the code, the first breakpoint will indicate if we received the CCR0 interrupt. If we wrote the code properly, we should NOT stop here.
- We should NOT stop at the second breakpoint either. CCR2 was set up to change the Output Signal, not generate an interrupt.
- We should stop at the 3rd breakpoint. We left the timer configured to break whenever TAR rolled-over to zero. (That is, whenever TA0IFG or TA1IFG gets set.)

```
//*********************************************************************
// Interrupt Service Routines
//*********************************************************************
#pragma vector=TIMER0_A0_VECTOR
__interrupt void ccr0_ISR (void)
{
    // 4. Timer ISR and vector

    // Toggle the Red LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}

#pragma vector=TIMER0_A1_VECTOR
__interrupt void timer0_ISR (void)
{
    // 4. Timer ISR and vector

    switch(__even_in_range( TA0IV, 14 )) {
     case  0: break;                 // None
     case  2: break;                 // CCR1 IFG
     case  4:                        // CCR2 IFG
              _no_operation();       // gives us something to set a
              break;
     case  6: break;                 // CCR3 IFG
     case  8: break;                 // CCR4 IFG
     case 10: break;                 // CCR5 IFG
     case 12: break;                 // CCR6 IFG
     case 14:                        // TAR overflow
              // Toggle the Green LED on/off
              GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN7 );
              break;
     default: _never_executed();
```

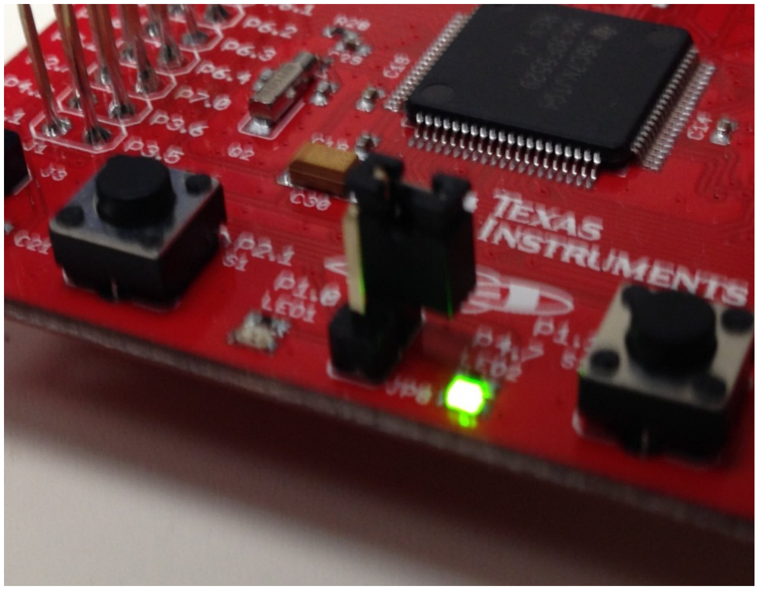
Note: As of this writing, due to an emulator bug with the 'FR5969 – as we discussed in an earlier lab exercise – terminating, restarting, or resetting the 'FR5969 with two or more breakpoints set may cause an error. If this occurs, load a different program, then reload the current one again.

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

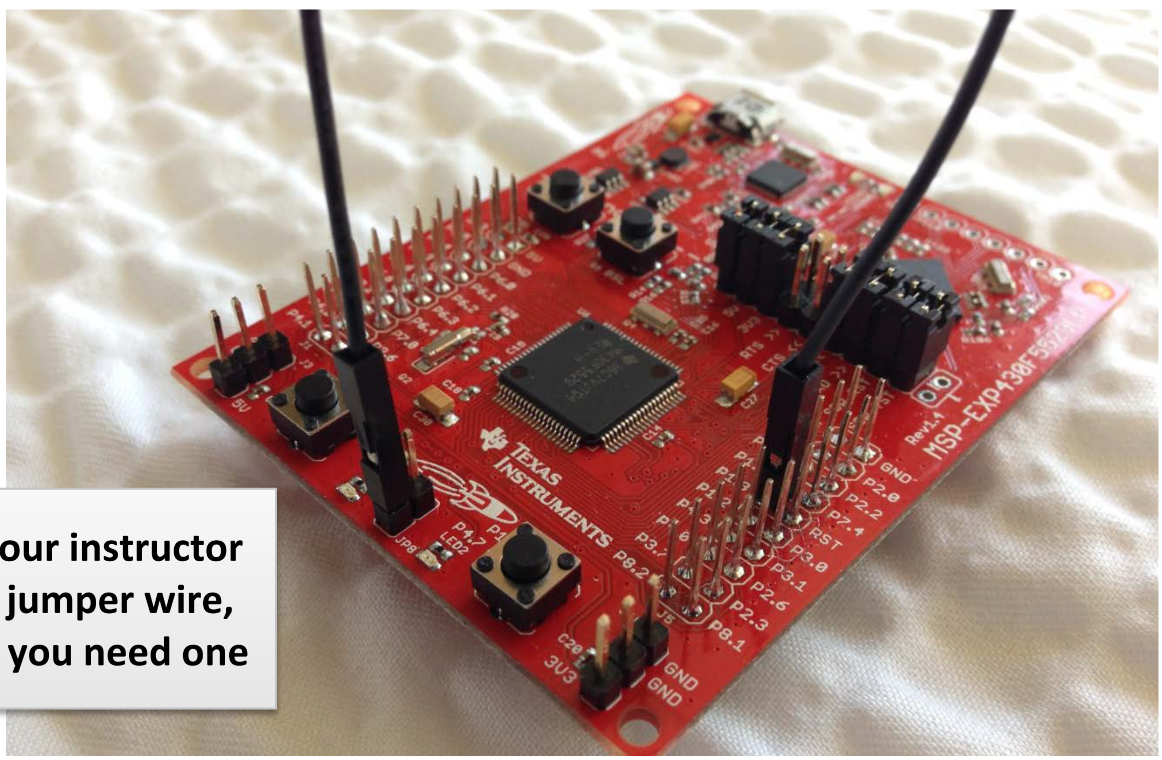
Why doesn't the LED1 toggle? ____

13. Add the jumper wire to your board to connect the timer output pin to LED1.

a) Remove the jumper (JP8 or J6) that connects the LED1 to P1.0 (or P4.6).
(We recommend reconnecting it to the top pin of the jumper so that you don't lose it.)

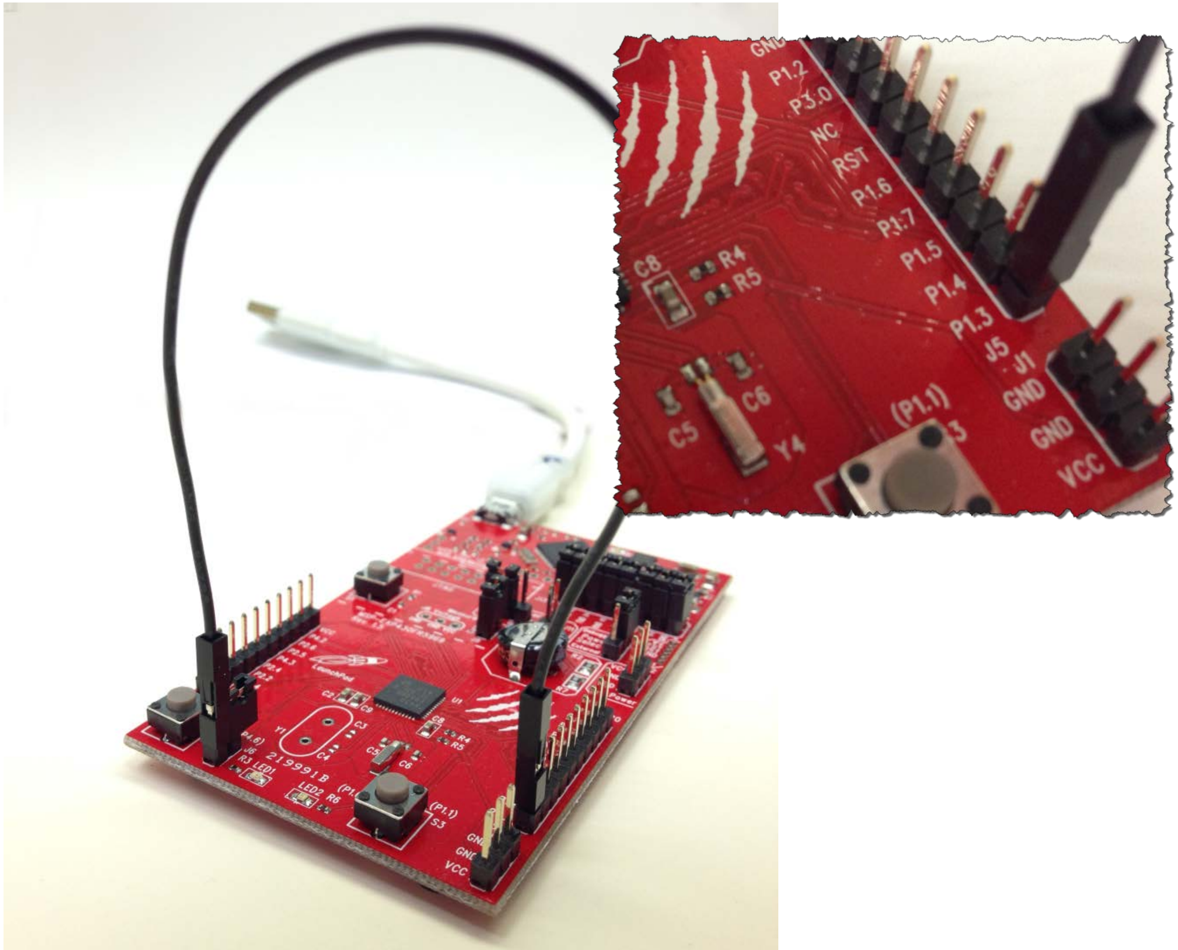


b) On the 'F5529 Launchpad, connect P1.3 (fifth pin down, right-side of board, inside row of pins) to the bottom of the LED1 jumper (JP8) using the jumper wire.
(See the next page for the 'FR5969 Launchpad.)



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

c) On the 'FR5969 (not shown), connect P1.3 (in the lower, right-hand corner of the BoosterPack pins to the LED1 jumper (J6).



d) We didn't include a picture showing the 'FR4133 pin P8.3 being connected to LED1. It's fairly easy to find, though as it's in the lower-left corner of the Boosterpack pins.

14. Run your code.

Hopefully both LED's are now blinking. LED1 should toggle first, then the LED2.

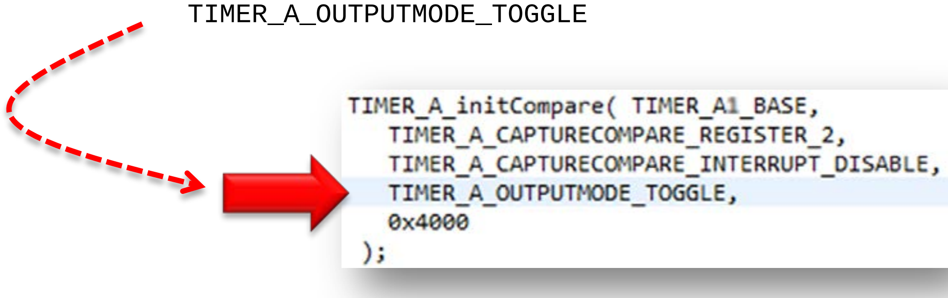
Do they both blink at the same rate? ____

Why or why not? ____

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

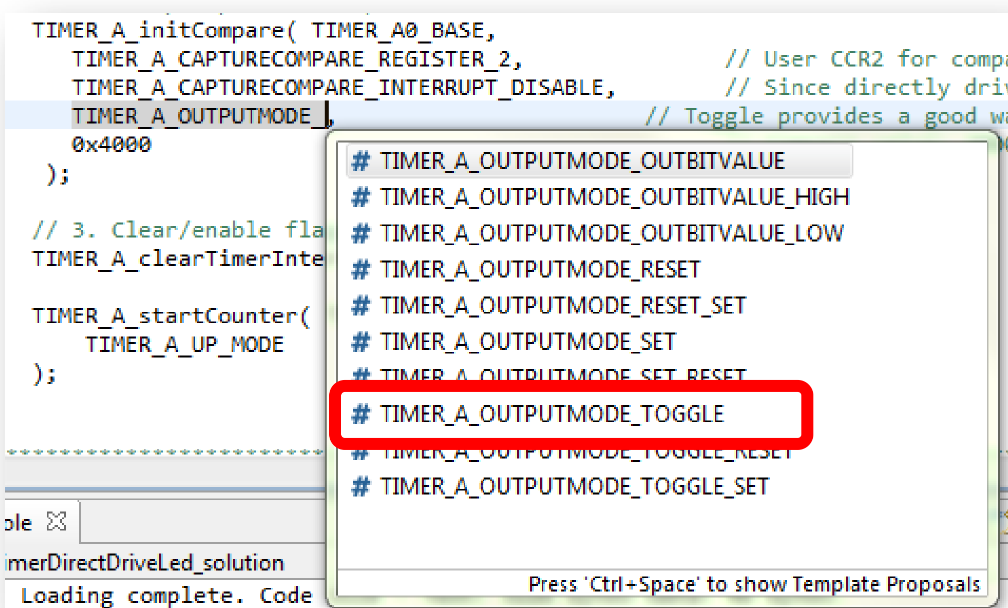
16. Modify one parameter of the function that configures CCR2, changing it to use the mode:

```
TIMER_A_OUTPUTMODE_TOGGLE
```



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? ______________________________

If a compare match (TAR = CCR2) causes the output to be SET (i.e. LED goes ON), what causes the output to be RESET (LED going OFF)?

____

How would this differ if you used the "TIMER_A_OUTPUTMODE_**SET_RESET**" 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, terminate and close the project.

(Optional) Lab 6c – Portable HAL

Can you create a single timer source file that would work on multiple platforms?

For the most part, "Yes". This is often done by creating a HAL (hardware abstraction layer). We've created a rudimentary HAL version of Lab 6c. You can find this in the solution file:

```
lab_06c_timerHal_solution.zip
```

While the timer file is shared between the two HAL solutions, we didn't get too fancy with this. There are a couple of things we didn't handle; for example, we didn't do anything with *unused_interrupts.c* and so it hade to be edited manually when porting between processors.

Play with it as you wish…

(Optional) Lab 6d – Simple PWM (Pulse Width Modulation)

While we don't have a complete write-up for our Simple PWM lab exercise, we created a solution that shows off the TIMER_A_simplePWM() DriverLib function.

The `lab_06d_simplePWM` project uses this DriverLib function to create a single PWM waveform. As with Lab 6c, the output is routed to LED1 using a jumper wire. By default, it creates a 50% duty cycle … which means it blinks the light on/off (50% on, 50% off) similar (but slightly faster) than our previous lab exercise.

One big change, though, is that we added two arguments to the initTimers() function. These values are the "Period" and "Duty Cycle" values that are passed to the simplePWM function. We also rewrote the main while{} loop so that it calls initTimers() every second.

The purpose of these changes was to allow you to have an easy way to experiment with different Period & Duty Cycle values without having to re-build and re-start the program over-and-over again. The values for period and duty-cycle were created as global variables – again, this makes it easier to change them while debugging the project.

The easiest way to experiement with this program once you've started it running is to:

- Halt (i.e. Suspend) the program
- View the two values in the Expressions watch window
- Change the values, as desired
- Continue running the program – in a second, literally, the values should take effect

By the way, if you change the period to something smaller, you won't be able to see the LED going on/off anymore – it will just appear to stay on. At this point, changing the duty cycle will cause the LED to appear bright (or dim).

As the name implies, this is a simple example, using a Driver Library function to quickly get PWM running.

Both Timer_A and Timer_B peripherals can create multiple/complex PWM (pulse-width modulation) waveforms. At some point, we may add additional PWM examples to the workshop, but if you want to learn more right now, we highly recommend that you review the excellent discussion in John Davies book: ***MSP430 Microcontroller Basics*** **by John H. Davies, (ISBN-10 0750682760) Link↗**

Chapter 6 Appendix

Lab6a Answers

Lab 6a Worksheet (1-2)

Goal: Write a function setting up Timer_A to generate an interrupt every two seconds.

1. **How many clock cycles does it take for a 16-bit counter to 'rollover'?** *(Hint: 16-bits)*

   2^{16} = 64K

2. **Our goal is to generate a two second interrupt rate based on the timer clock input diagramed above.**

   Pick a source clock for the timer. *(Hint: At 2 seconds, a slow clock might work best.)*

   Clock input (circle one): ACLK (circled) SMCLK

   In Lab 4c we configured ACLK for 32KHz

Lab 6a Worksheet (3)

3. **Calculate the Timer settings for the clocks & divider values needed to create a timer interrupt every 2 seconds. (That is, how can we get a timer period rate of 2 seconds.)**

   Which clock did you choose in the previous step? Write its frequency below and then calculate the *timer period rate*.

Input Clock: ACLK running at the frequency = 32 KHz

Timer Clock	=	32KHz *(input clock frequency)*	÷	1 *(timer clock divider)*	=	32K / sec *(timer clock freq)*
Timer Rate	=	1 sec / 32K cycles *(timer clock period (i.e. 1 / timer clock freq))*	X	65536 *(counts for timer to rollover)*	=	2 sec *(timer rate period)*

I.E. 64K

Lab 6a Worksheet (4-5)

4. Which Timer do you need to use for this lab exercise?

Launchpad	Timer	Short Name	Timer's Enum
'F5529	Timer0_A3	TA0	TIMER_A0_BASE
'FR4133	Timer0_A3	TA0	TIMER_A0_BASE
'FR5969	Timer1_A5	TA1	TIMER_A1_BASE

Pick the one req'd for your board: AO or A1

Write down the timer enumeration you need to use: **TIMER_ ________ _BASE**

5. Write the *TIMER_A_initContinuousMode()* function.

```
Timer_A_initContinuousModeParam initContParam = { 0 };
  initContParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initContParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;
  initContParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initContParam.timerClear = TIMER_A_DO_CLEAR;
  initContParam.startTimer = false;
Timer_A_initContinuousMode(TIMER_ AO/A1 _BASE, &initContParam );
```

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

Lab 6a Worksheet (7)

7. Complete the code to for the 3rd part of the "Timer Setup Code".

The third part of the timer setup code includes:

- ~~Enable the interrupt (IE)~~ … *we don't have to do this, since it's done by the TIMER_A_configureContinuousMode() function (from step5 on page 6-39).*
- Clear the appropriate interrupt flag (IFG)
- Start the timer

```
// Clear the timer interrupt flag
Timer_A_clearTimerInterruptFlag ( TIMER_ _BASE );   // Clear TA0IFG
```

```
// Start the timer
Timer_A_startCounter (                 // Function to start timer
    TIMER_ _BASE,                      // Which timer?   (AO or A1)
    TIMER_A_CONTINUOUS_MODE            // Run in Continuous mode
```

Lab 6a Worksheet (8a)

'F5529 Solution

8. Change the following interrupt code to toggle LED2 when Timer_A rolls-over.

a) Fill in the details for your Launchpad.

Port/Pin number for LED2: `Port` **4**, `Pin` **7**

Interrupt Vector: `#pragma vector =` **TIMER0_A1** `_VECTOR`

Interrupt Vector register: **TA0IV**

(for example, we used `P1IV` for GPIO Port 1)

'FR5969 Solution

8. Change the following interrupt code to toggle LED2 when Timer_A rolls-over.

a) Fill in the details for your Launchpad.

Port/Pin number for LED2: `Port` **1**, `Pin` **0**

Interrupt Vector: `#pragma vector =` **TIMER1_A1** `_VECTOR`

Interrupt Vector register: **TA1IV**

(for example, we used `P1IV` for GPIO Port 1)

Lab 6a Worksheet (8b)

b) Here is the interrupt code that exists from a previous exercise, change it as needed:

```
#pragma vector=~~PORT1_VECTOR~~  TIMER1_A1_VECTOR
__interrupt void ~~pushbutton_ISR~~ timer_ISR (void)
{
    switch( __even_in_range( ~~P1IV~~ TA1IV , ~~16~~ 14 ) ) {
        case 0: break;                          // No
        case 2: break;                          // Pi
        case 4:                                 // Pi
            ~~GPIO_toggleOutputOnPin( GPIO_P~~
            break;
        case 6: break;                          // Pin 2
        case 8: break;                          // Pin 3
        case 10: break;
        case 12: break;
        case 14: GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
            break;                              // Pin 6
        ~~case 16: break;~~                     // Pin 7
        default:   _never_executed();
    }
}
```

TA0IV (for 'F5529 and 'FR4133)
TA1IV (for 'FR5969)

or for the 'F5529:
GPIO_toggleOutputOnPin(GPIO_PORT_P4, GPIO_PIN7);

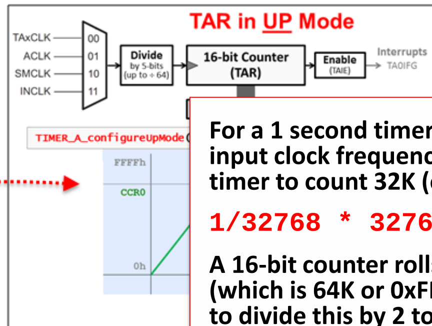
- 'FR5969 Answers are shown
- For 'FR4133, use:
  - TIMER1_A1_VECTOR
  - TA1IV
  - P4.0
- For 'F5529, use:
  - TIMER0_A1_VECTOR
  - TA0IV
  - P4.7

Lab6b Answers

Lab 6b Worksheet (1)

1. **Calculate the timer period (for CCR0) to create a 1 second interrupt rate.**

   Here's a quick review from our discussion.



For a 1 second timer rate and a 32KHz input clock frequency, we need the timer to count 32K (or 32768) times:

`1/32768 * 32768 = 1 sec`

A 16-bit counter rolls over at 2^{16} counts (which is 64K or 0xFFFF). We just need to divide this by 2 to get 32K:

`Period = 0xFFFF/2 = 0x8000`

Timer_A's counter (TAR) will count up u... then reset back to zero. What value do v...

Timer Clock	=	32 KHz	÷	1	=	32 KHz
		input clock frequency		*timer clock divider*		*timer clock freq*
Timer Rate	=	1/32768	x	0x8000	=	1 SECOND
		timer clock period (i.e. 1 / timer clock freq)		*timer counter period (i.e. CCR0 value)*		*timer rate period*

Lab 6b Worksheet (2)

2. **Complete the *TIMER_A_initUpMode()* 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_initUpModeParam initUpParam = { 0 };
  initUpParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initUpParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;

  initUpParam.timerPeriod = 0xFFFF / 2 ; // (calculated in previous question)
  initUpParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initUpParam.captureCompareInterruptEnable_CCR0_CCIE =
      TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE ; // Enable CCR0 compare interrupt
  initUpParam.timerClear = TIMER_A_DO_CLEAR;
  initUpParam.startTimer = false;                    A0 or A1
Timer_A_initUpMode(TIMER___BASE, &initUpParam );
```

Lab 6b Worksheet (3)

3. Modifying our previous code, we need to clear both interrupts and start the timer.

We copied the following code from the previous exercise. It needs to be modified to meet the 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 'startCounter' function. Is there anything that needs to change when switching from Continuous to Up mode?*

AO or A1

```
// Clear the timer flag and start the timer
Timer_A_clearTimerInterruptFlag( TIMER___BASE );          // Clear TA0IFG
Timer_A_clearCaptureCompareInterruptFlag (                // Clear CCR0IFG
      TIMER_____BASE,
      TIMER_A_CAPTURECOMPARE_REGISTER_0 );
Timer_A_startCounter( TIMER___BASE,                       // Start timer in
     TIMER_A_UP_MODE );                                   // UP mode
```

Lab 6b Worksheet (4)

4. Add a second ISR to toggle the LED1 whenever the CCR0 interrupt fires.

On your Launchpad, what Port/Pin number does the LED1 use? **P4.6 (for 'FR5969)** / **P1.0 (for 'F5529 & FR4133)**

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

```
#pragma vector= TIMER1_A0_VECTOR   (or TIMER1_A0_VECTOR for 'F5529)
__interrupt void ccr0_ISR (void)
{
    // Toggle the LED1 on/off

    GPIO_toggleOutputOnPin( GPIO_PORT_P____, GPIO_PIN____ );

}
```

Reflects the value from above

Lab 6b : Lab Debrief

Debug/Run

Follow the same basic steps as found in the previous lab for debugging.

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

11. Run your code.

If all worked well, when the counter rolled over to zero, an interrupt should occur. Actually, two interrupts should occur. Once you reach the first breakpoint, resume running your code and you should reach the other ISR.

Which ISR was reached first? **LED1 then LED2**

Why? **Because the CCR0 interrupt occurs before the TAIFG interrupt**

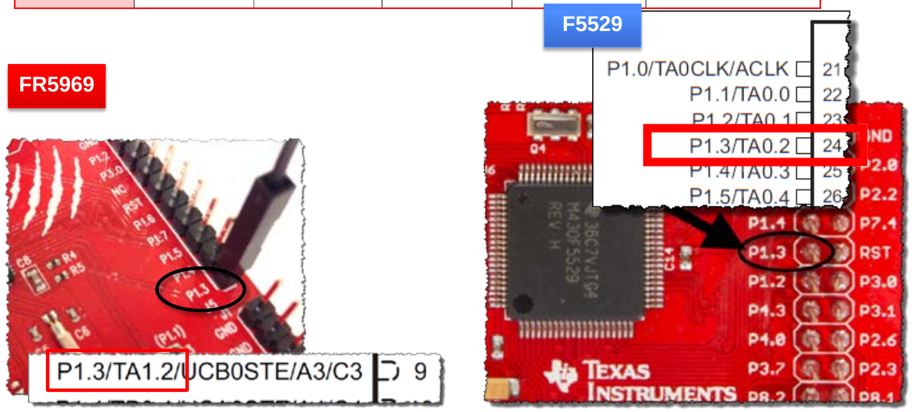
This is shown on the slide entitled "TAR in UP Mode". Since they occur at nearly the same instant in time, you have to set breakpoints in order to see that LED1 happens before LED2.

Lab6c Answers

Lab 6c Worksheet (1)

1. Figure out which BoosterPack pin will be driven by the timer's output.

Device	Timer	CCRx	Signal	GPIO Port/Pin	Is Pin on Boosterpack?
'F5529	TimerA0	CCR2	TA0.2	P1.3	Yes
'FR4133	TimerA1	CCR2	TA1.2	P8.3	Yes
'FR5969	TimerA1	CCR2	TA1.2	P1.3	Yes



Lab 6c Worksheet (2)

2. Complete the following function to "select" P1.3 as a timer function

F5529

'F5529 Users, here's the function you need to complete:

GPIO_setAs PeripheralModuleFunctionOutputPin (
GPIO_PORT_P1 ,
GPIO_PIN3);

FR5969

'FR5969 Users, your function requires one more argument:

GPIO_setAs PeripheralModuleFunctionOutputPin (
GPIO_PORT_P1 ,
GPIO_PIN3 ,
GPIO_PRIMARY_MODULE_FUNCTION);

FR4133

'FR4133 Users, your function requires one more argument:

GPIO_setAs PeripheralModuleFunctionOutputPin (
GPIO_PORT_P1 ,
GPIO_PIN3 ,
GPIO_PRIMARY_MODULE_FUNCTION);

Lab 6c Worksheet (3)

3. **Modify the TIMER_A_configureUpMode() function?**

   Here is the code we wrote for the previous lab exercise. We only need to make one change to the code. Since we will drive the signal directly from the timer, we don't need to generate the CCR0 interrupt anymore.

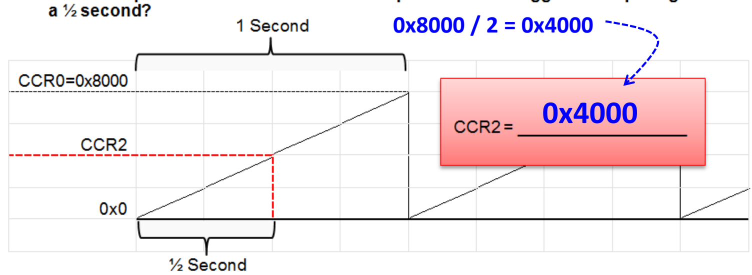
   Mark up the code below to disable the interrupt. *(We'll bet you can make this change without even looking at the API documentation. Intuitive code is one of the benefits of using DriverLib!)*

```
Timer_A_initUpModeParam initUpParam = { 0 };
  initUpParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initUpParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;
  initUpParam.timerPeriod = 0xFFFF / 2;
  initUpParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initUpParam.captureCompareInterruptEnable_CCR0_CCIE = TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE;
  initUpParam.timerClear = TIMER_A_DO_CLEAR;
  initUpParam.startTimer = false;
Timer_A_initUpMode( TIMER___BASE, &initUpParam );
```

We changed 'ENABLE' to 'DISABLE'

Lab 6c Worksheet (4-5)

4. **What 'compare' value does CCR2 need to equal in order to toggle the output signal at a ½ second?**



5. **Add a new function call to set up Capture and Compare Register 2 (CCR2). This should be added to initTimers().**

CCR2 value we calculated above goes here

```
Timer_A_initCompareModeParam initCcr2Param = { 0 };
  initCcr2Param.compareRegister = TIMER_A_CAPTURECOMPARE_REGISTER_2;
  initCcr2Param.compareInterruptEnable = TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE;
  initCcr2Param.compareOutputMode = TIMER_A_OUTPUTMODE_TOGGLE_RESET;
  initCcr2Param.compareValue = 0x4000;
Timer_A_initCompareMode( TIMER___BASE, &initCcr2Param );
```

Lab 6c Worksheet (6)

6. **Compare your previous code to that below.**

What did we change? Added _no_operation() – something to breakpoint on

```
#pragma vector=TIMER0_A1_VECTOR
__interrupt void timer0_ISR(void)
{
    switch(__even_in_range( TA0IV, 14 )) {
        case 0:  break;            // No interrupt
        case 2:  break;            // CCR1 IFG
        case 4:                    // 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

        def
    }
}
```

During debug, we will ask you to set a breakpoint on 'case 4'.

Why should case 4 not occur, and thus, the breakpoint never reached?

`TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE,`

We disabled the INT because we're driving the signal directly to the pin

Lab 6c Worksheet (7)

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

- **Lower Power:**
  When the Timer drives the pin; no need to wake up the CPU. (Either that, or it leaves the CPU free for other processing.)

- **Less Latency:**
  When the CPU toggles the pin, there is a slight delay that occurs since the CPU must be interrupted, then go run the ISR.

- **More Deterministic:**
  The delay caused by generating/responding to the interrupt may vary slightly. This could be due to another interrupt being processed (or a higher priority interrupt occurring simultaneously). Directly driving the output removes the variance and makes it easy to "determine" the time that the output will change!

Lab 6c Debrief

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

Why doesn't the LED1 toggle? We removed the interrupt that caused us to run the GPIO toggle function and replaced it with code to let the timer directly drive the LED … but we haven't hooked up the LED, yet.

14. Run your code.

Hopefully both LED's are now blinking. LED1 should toggle first, then the LED2.

Do they both blink at the same rate? No

Why or why not? LED2 is based on the timer counting up to the value in CCR0 (0x8000); while LED1 toggles when the counter reaches CCR2 (set to 0x4000) and is reset whenever the counter reaches CCR0.

Lab 6c Debrief

17. Build and run your code with the new OutputMode setting.

Do they both blink at the same rate? Yes (although offset by ½ second)

If a compare match (TAR = CCR2) causes the output to be SET (i.e. LED goes ON), what causes the output to be RESET (LED going OFF)?

The next time TAR equals CCR2

How would this differ if you used the "TIMER_A_OUTPUTMODE_**SET_RESET**" 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)?

In this case, the "RESET" occurs when TAR = CCR0

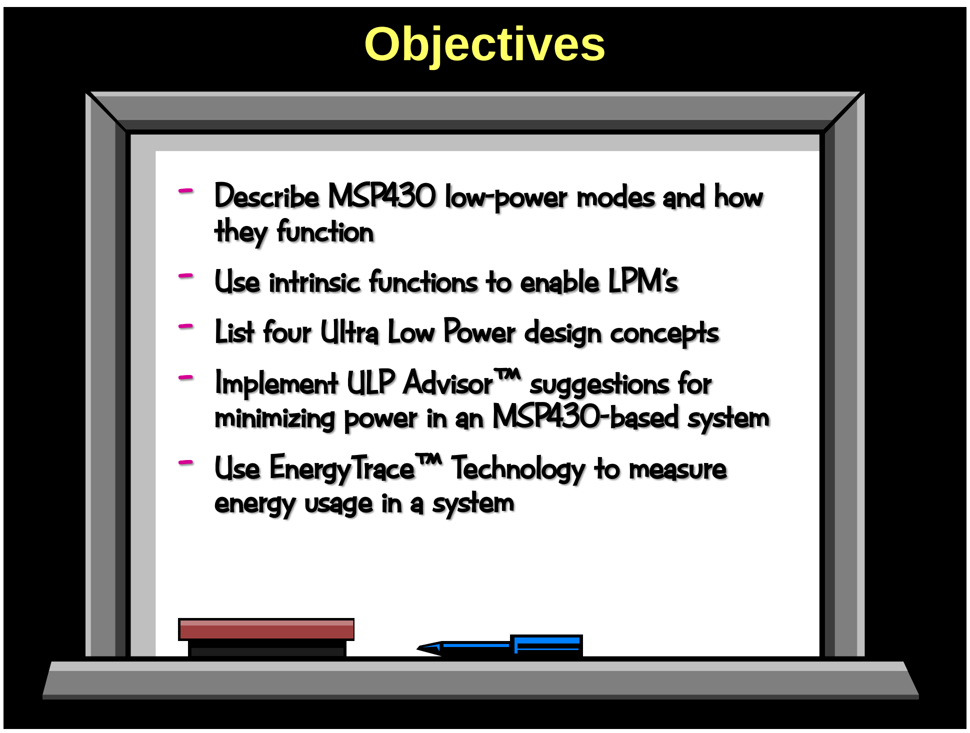
Low Power Optimization

Introduction

Ultra-low power is in our DNA.

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

Learning Objectives



Chapter Topics

Low Power Optimization 7-1

Low Power Modes (LPM) 7-3
Using Low Power Modes 7-5

Low Power Concepts 7-7
Use Interrupts and Low-Power Modes 7-7
Replace Software with Peripherals 7-8
Configure Unused Pins 7-8
Efficient Code Makes a Difference 7-9

Follow the Rules (ULP Advisor™) 7-10
About ULP Advisor™ 7-10
The List … of ULP Rules 7-12
How Do You Enable ULP Advisor™? 7-13

EnergyTrace™ 7-14
How does EnergyTrace Work? 7-16

Lab 7 – Low Power Optimization 7-17

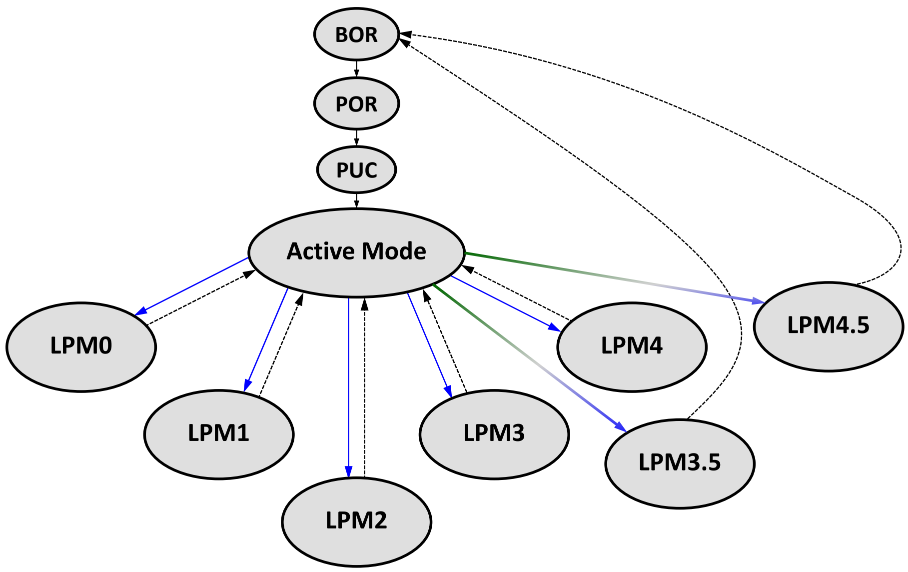
Prerequisites and Tools

Prerequisites & Tools

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

Low Power Modes (LPM)

Low Power Modes



Low-Power Modes

Operating Mode	CPU (MCLK)	SMCLK	ACLK	RAM Retention	BOR	Self Wakeup	Interrupt Sources
Active	☒	☒	☒	☒	☒		Timers, ADC, DMA, WDT, I/0, External Interrupt, COMP, Serial, RTC, other…
LPM0		☒	☒	☒	☒	☒	
LPM1		☒	☒	☒	☒	☒	
LPM2			☒	☒	☒	☒	
LPM3			☒	☒	☒	☒	
LPM3.5					☒	☒	External Interrupt, RTC
LPM4				☒	☒		External Interrupt
LPM4.5					☒		External Interrupt

Low-Power Modes (Bit Settings)

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

* SCG = System Clock Generator

MSP430™ Series Comparison

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

Using Low Power Modes

Entering Low Power Modes

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

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

Automatically Re-entering LPM (after ISR)

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

  _low_power_mode_3();
  //while(1);
}
```

LPM3

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

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

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

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

Leaving LPM (after ISR)

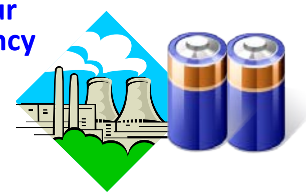
```
main()
{
  initGpio();
  initClocks();
  initTimers();
  while(1){
    _low_power_mode_3();        LPM3
    filter();
  }
}

#pragma vector = TIMER1_A0
__interrupt ISR()
{
  getSample();
  _low_power_mode_off_on_exit();
}  // Return from interrupt (RETI)
```

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

Low Power Concepts

Principles For ULP Applications

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

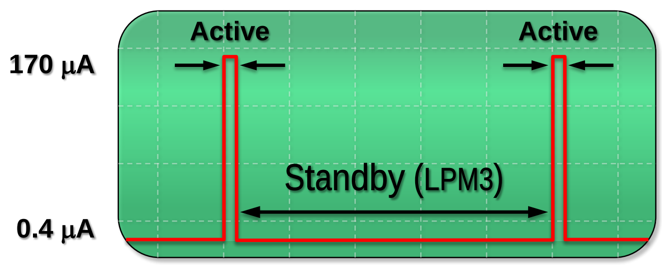


- Use interrupts to control program flow
- Maximize the time in LPM3
- Replace software with peripherals
- Configure unused pins properly
- Power manage external devices
- Efficient code makes a difference

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

Use Interrupts and Low-Power Modes

Use Interrupts & Maximize LPM3



Leave On the Slow Clock

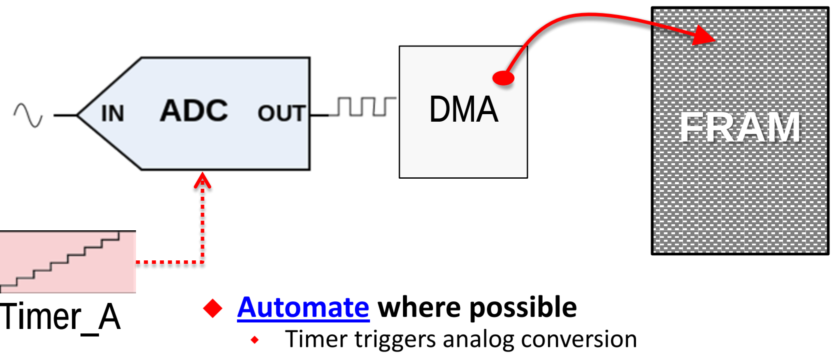
- Low power clock and peripherals interrupt CPU only for processing

On-Demand CPU Clock

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

Replace Software with Peripherals

Replace Software With Peripherals

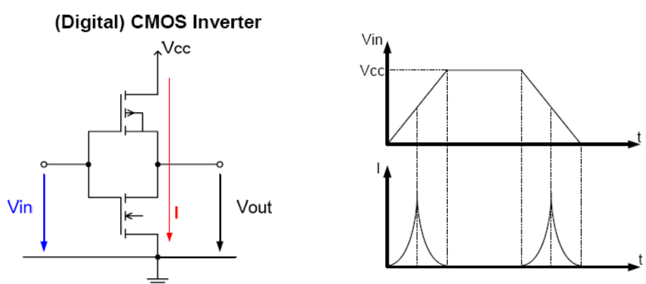


- **Automate where possible**
  - Timer triggers analog conversion
  - ADC triggers DMA to move result to memory
- **Saves power since CPU and high-speed clock can be turned off**
- **Higher precision and less latency for analog sampling since timer directly triggers conversion**
- **Faster results since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

Configure Unused Pins

Configure Unused Pins

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



Efficient Code Makes a Difference

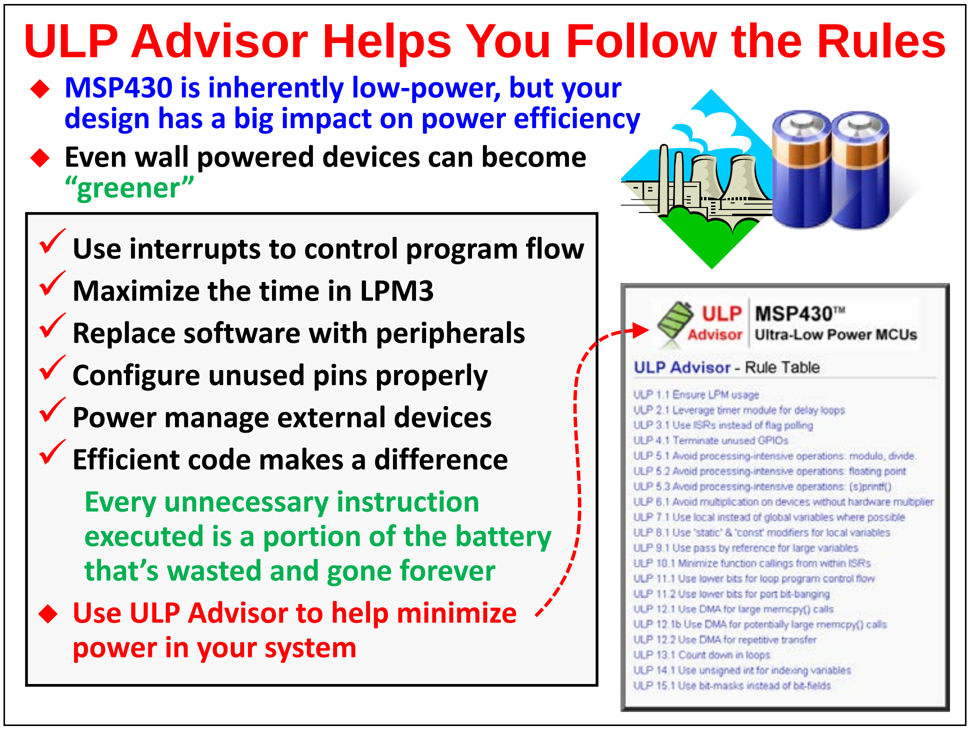
ULP "Sweet Spot"

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

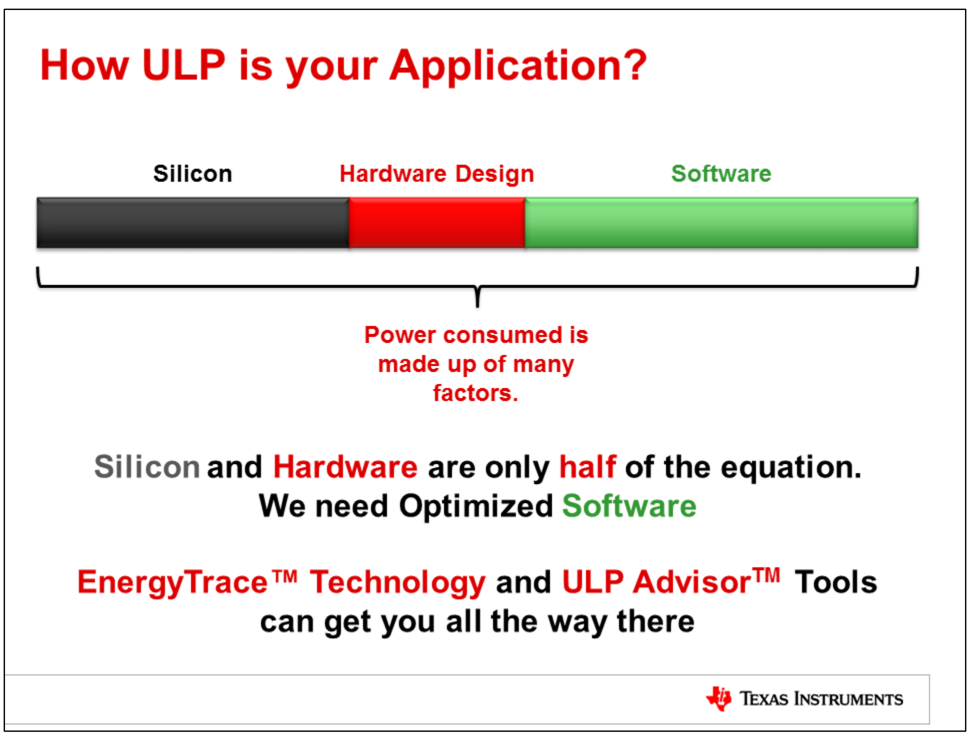
Optimize Performance

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

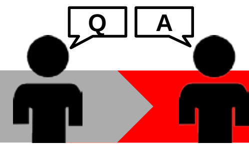
Follow the Rules (ULP Advisor™)



About ULP Advisor™



ULP Advisor™ benefits all experience levels



Beginning ULP developers	Experienced ULP developers
• Teaching tool for new MSP430 users • Practical introduction to ULP techniques • Immediate coding feedback • Wiki provides quick solution and detailed background info • Learn more from the community & E2E	• Not everybody remembers all the rules all the time • New rules might come in • Saves time vs. manually going through a large project or library to check for ULP • Helpful when developers inherit code from other sources • ULP Advisor should always be used regardless of the application or target device. • Contribute to wiki & E2E

Texas Instruments

MSP430 | Ultra-Low Power is in our DNA

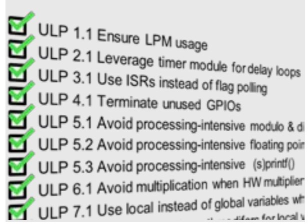
ULP Advisor^{TM} Software: Turning MCU developers into Ultra-Low-Power experts

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

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

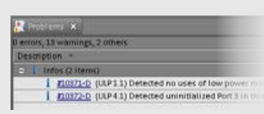
Checks against a thorough Ultra-Low-Power checklist.

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



Highlights areas of improvement within code.

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



Texas Instruments

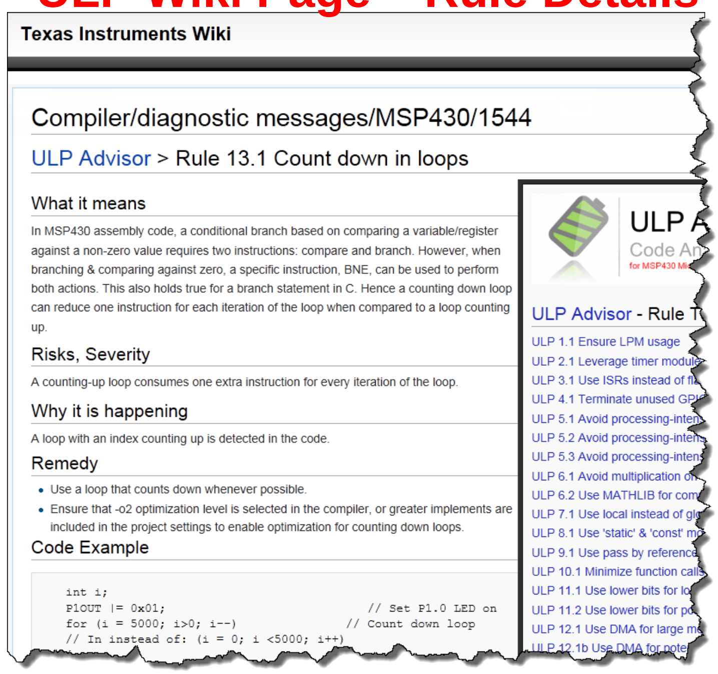
The List … of ULP Rules

ULP Advisor™
Code Analysis Tool
for MSP430 Microcontrollers

ULP Advisor Rules

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

ULP Wiki Page – Rule Details


Texas Instruments Wiki

Compiler/diagnostic messages/MSP430/1544

ULP Advisor > Rule 13.1 Count down in loops

What it means

In MSP430 assembly code, a conditional branch based on comparing a variable/register against a non-zero value requires two instructions: compare and branch. However, when branching & comparing against zero, a specific instruction, BNE, can be used to perform both actions. This also holds true for a branch statement in C. Hence a counting down loop can reduce one instruction for each iteration of the loop when compared to a loop counting up.

Risks, Severity

A counting-up loop consumes one extra instruction for every iteration of the loop.

Why it is happening

A loop with an index counting up is detected in the code.

Remedy

- Use a loop that counts down whenever possible.
- Ensure that -o2 optimization level is selected in the compiler, or greater implements are included in the project settings to enable optimization for counting down loops.

Code Example

```
int i;
P1OUT |= 0x01;                          // Set P1.0 LED on
for (i = 5000; i>0; i--)               // Count down loop
// In instead of: (i = 0; i <5000; i++)
```

ULP A
Code An
for MSP430 Mi

ULP Advisor - Rule T

ULP 1.1 Ensure LPM usage
ULP 2.1 Leverage timer module
ULP 3.1 Use ISRs instead of fl
ULP 4.1 Terminate unused GPI
ULP 5.1 Avoid processing-inten
ULP 5.2 Avoid processing-inten
ULP 5.3 Avoid processing-inten
ULP 6.1 Avoid multiplication o
ULP 6.2 Use MATHLIB for com
ULP 7.1 Use local instead of gl
ULP 8.1 Use 'static' & 'const' m
ULP 9.1 Use pass by reference
ULP 10.1 Minimize function calls
ULP 11.1 Use lower bits for lo
ULP 11.2 Use lower bits for po
ULP 12.1 Use DMA for large m
ULP 12.1b Use DMA for pote

How Do You Enable ULP Advisor™?

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

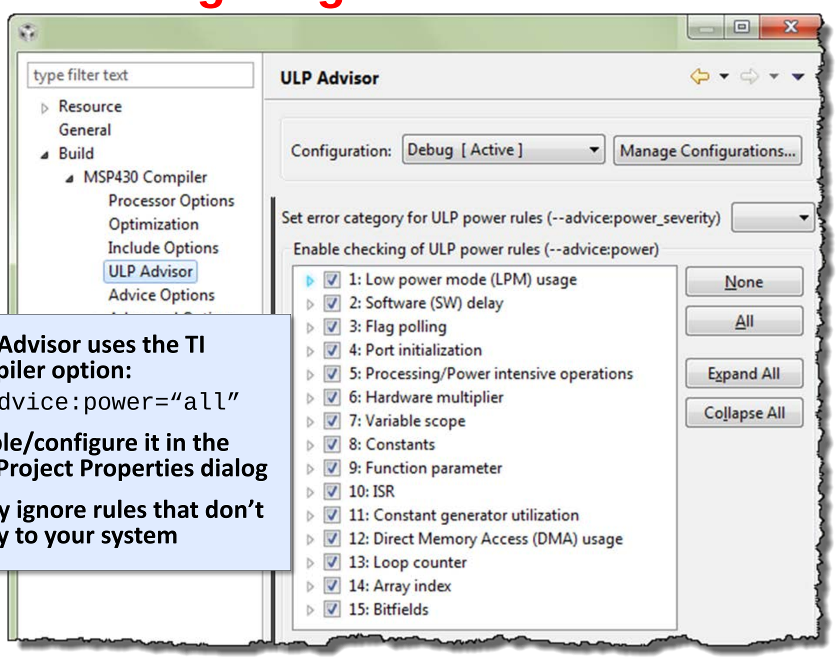
- Integrated into popular MSP430 IDEs for seamless operation
- ULP Advisor is automatically enabled & checks all code at build time
- Joins differentiated MSP430 software tools integrated into CCS, including MSP430Ware & Grace

Integrated into TI's Code Composer Studio

Integrated into IAR Embedded Workbench

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

Configuring ULP Advisor



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

EnergyTrace™

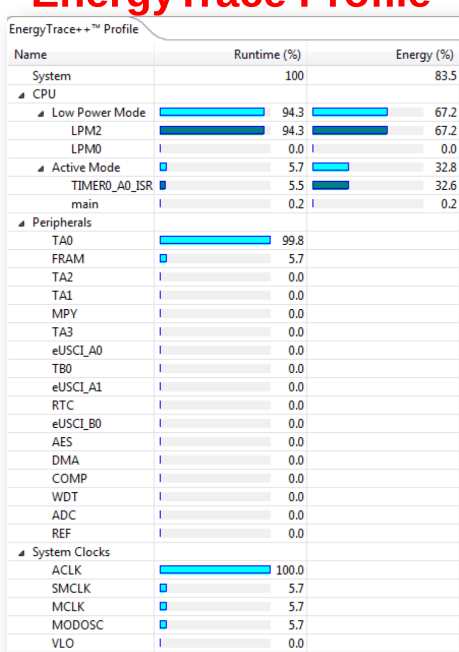
Energy Aware Debugging

MSP-EXP430FR5969 Launchpad with on-board MSP-FET

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

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

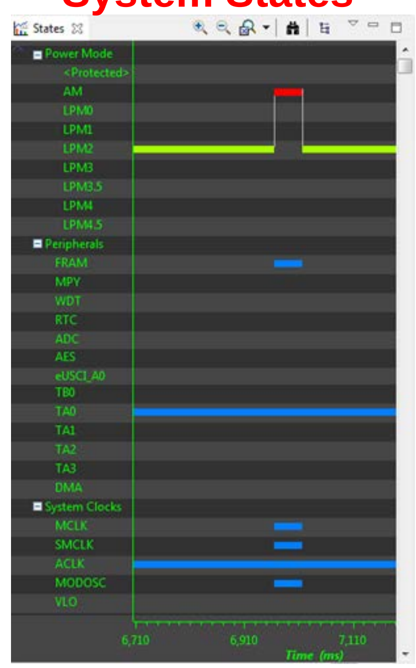
EnergyTrace Profile



EnergyTrace++™ Profile

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

System States



Power & Energy Graphs

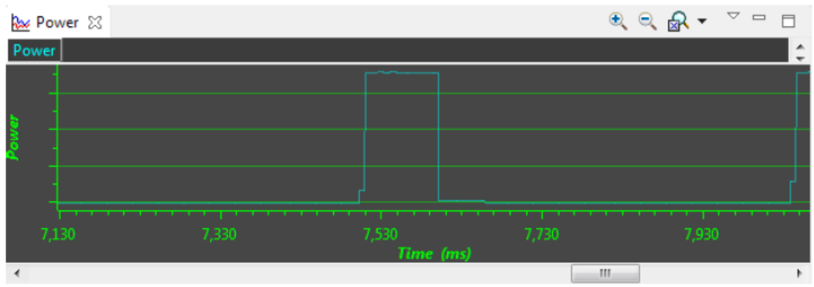


Figure 3-9. Power Window

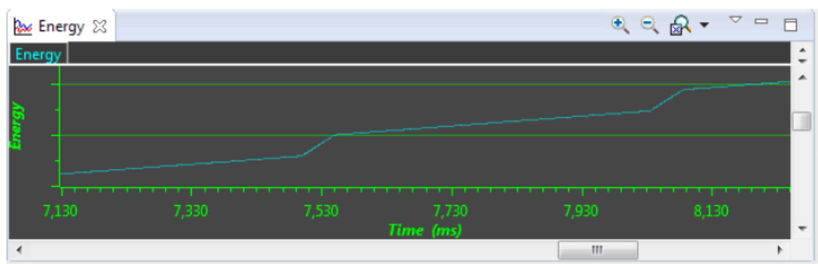
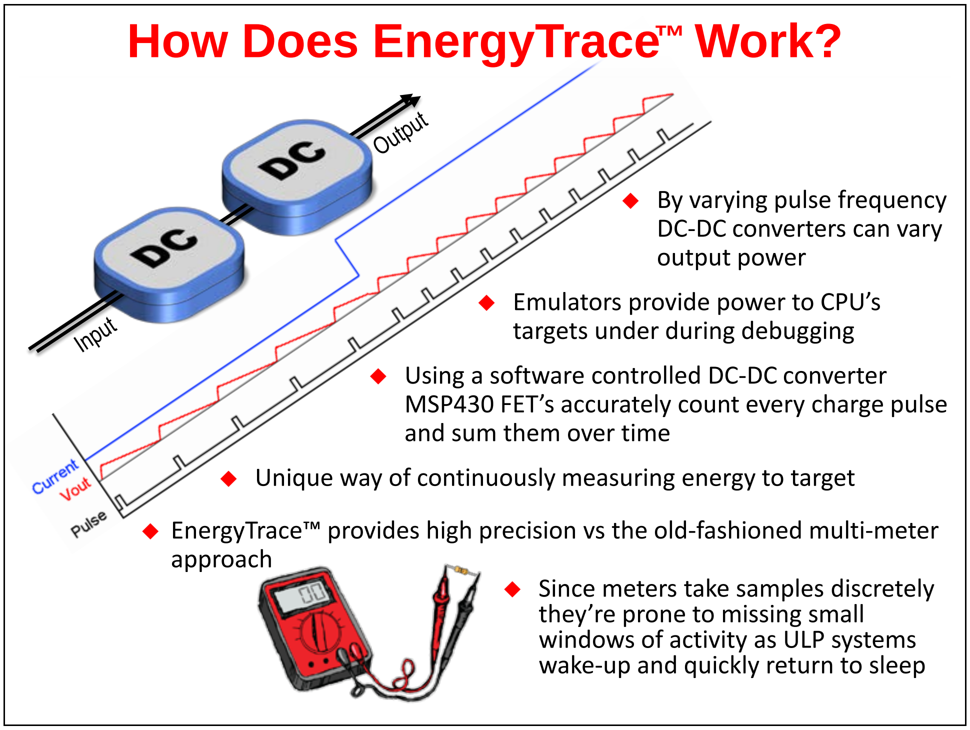


Figure 3-10. Energy Window

EnergyTrace Profile Comparison

How does EnergyTrace Work?



Lab 7 – Low Power Optimization

Abstract

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

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

Lab 7 – Optimizing for Low-Power

A. Getting Started with EnergyTrace™

Explore tools by comparing Lab4a & Lab4c

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

B. Using ULP Advisor, Interrupts and LPM3

Improve power using Lab4c & Lab6b

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

C. Does Initializing GPIO Ports Make a Difference?

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

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

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

Chapter Topics

Low Power Optimization 7-15

Lab 7 – Low Power Optimization 7-17
Abstract 7-17
Notice - Measuring Energy in Lab 7 7-19
How to Measure Energy 7-19
Lab Exercise Energy Measurement Recommendations 7-20

Lab 7a – Getting Started with Low-Power Optimization 7-21
Prelab Worksheet 7-21
Configure CCS and Project for EnergyTrace 7-22
Build Project and Run with EnergyTrace 7-24
EnergyTrace with Free Run 7-28
Compare EnergyTrace Profiles 7-28
Create Energy Profile for lab_04c_crystals 7-29
What have we learned in Lab7a? 7-30
(Optional) Viewing 'FR5969 EnergyTrace++ States 7-31

Lab 7b – Reducing Power with ULP Advisor, LPM's and Interrupts 7-32
Get Suggestions from ULP Advisor 7-32
Replace __delay_cycles() 7-35
Using Low-Power Mode (LPM3) 7-39
(Optional) Viewing 'FR5969 EnergyTrace++ States 7-40
(Optional) Directly Driving the LED from Timer_A 7-41

Lab 7c – Configuring Ports for Lowest Power 7-42
Import and Modify Program 7-42
Capture Baseline Reference 7-43
Add GPIO Port Initialization Code 7-43
Improve on GPIO Port Initialization 7-45

Chapter 7 Appendix 7-46
Connecting MSP-FET to 'F5529 USB Launchpad 7-46
Lab 7 Debrief and Solutions 7-48

Notice - Measuring Energy in Lab 7

How to Measure Energy

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

1. The 'FR5969 FRAM Launchpad supports the full EnergyTrace++ feature set – which includes energy measurement as well as tracing the CPU modes and peripheral states.
2. The 'FR4133 FRAM Launchpad supports EnergyTrace – that is, it allows energy measurements but does not include the ++ features of tracing modes & states.
3. The new MSP-FET (Flash Emulation Tool) – supports measurement of energy with the EnergyTrace feature for all MSP430 devices.
4. If you do not have either tool which supports TI's EnergyTrace, you will need to measure it the old fashioned way – using a multi-meter to determine the current being drawn by the MSP430 CPU. We refer you to Section 2.3 of the *MSP-EXP430F5529 Launchpad User's Guide* (slau533b.pdf) for a detailed procedure on how this can be done.

Measuring Energy in Lab 7

MSP-EXP430FR5969 Launchpad

MSP-FET

- Now Available (as of June 2014)

- **Four ways to measure Energy**
  1. **MSP-EXP430FR5969 Launchpad** supports full EnergyTrace++
  2. **MSP-EXP430FR4133 Launchpad** supports EnergyTrace
  3. **MSP-FET** supports EnergyTrace energy measurement
  4. Old fashioned **Multi-Meter** crudely measures CPU's current draw
- **Lab steps written assuming EnergyTrace hardware is available**
  - Refer to Chapter Appendix for "how to" connect MSP-FET to the **'F5529 USB Launchpad**
  - If using multi-meter, substitute current measurement procedure whenever lab steps ask you to read from energy data from the EnergyTrace window

Lab Exercise Energy Measurement Recommendations

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

FR5969

‘FR5969 FRAM Launchpad

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

F5529

‘F5529 USB Launchpad

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

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

Bottom Line

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

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

Using a Multi-Meter

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

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

Lab 7a – Getting Started with Low-Power Optimization

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

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

Prelab Worksheet

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

 ____

 ____

 Which devices support EnergyTrace++? ____

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

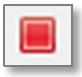
 ____

3. How can you calculate energy without EnergyTrace? ______________________________

 ____

 What is the downside to this method? ____

Configure CCS and Project for EnergyTrace

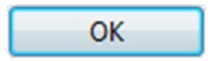


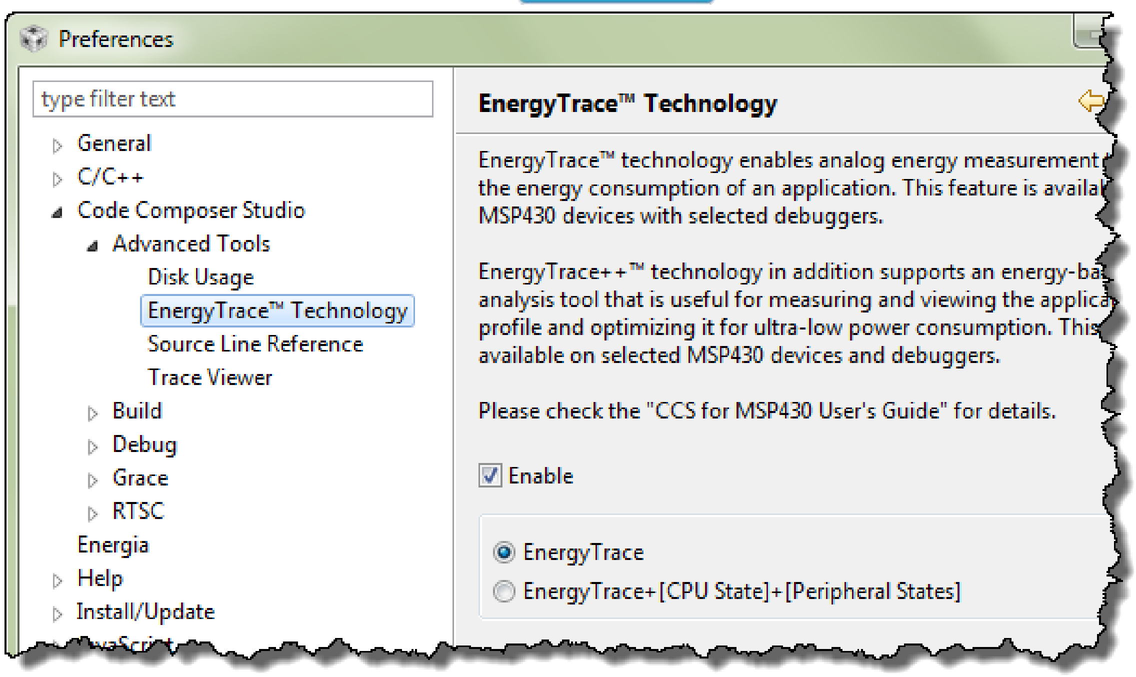
1. Terminate the debugger if it's still open and close all projects and files that may be open in your CCS workspace.

2. Enable EnergyTrace profiling.

`Window → Preferences`

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

☑ Enable ⦿ EnergyTrace 



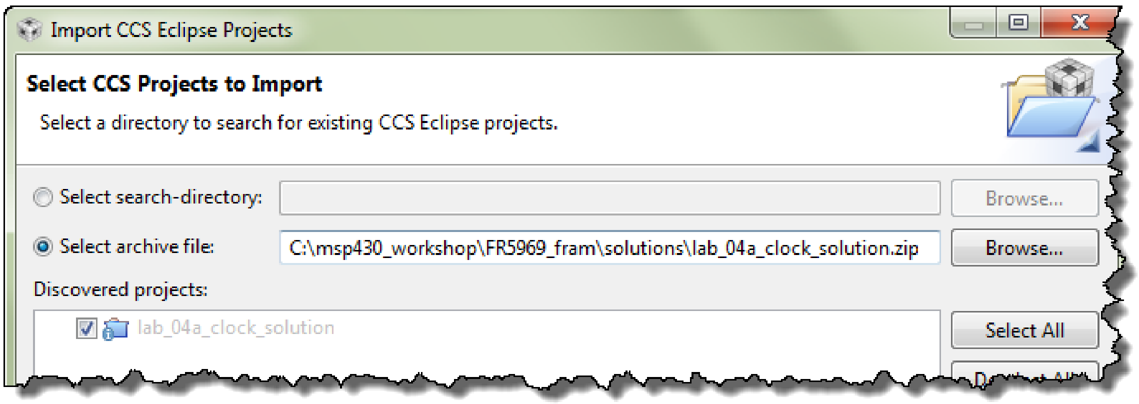
Note: 'FR5969 users, we'll look at the +States mode later on in the lab exercise.

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

   `Project → Import CCS Projects`

   Then select the project based upon the board you're using - then click OK.

```
C:\msp430_workshop\F5529_usb\solutions\lab_04a_clock_solution.zip
C:\msp430_workshop\FR4133_fram\solutions\lab_04a_clock_solution.zip
C:\msp430_workshop\FR5969_fram\solutions\lab_04a_clock_solution.zip
```

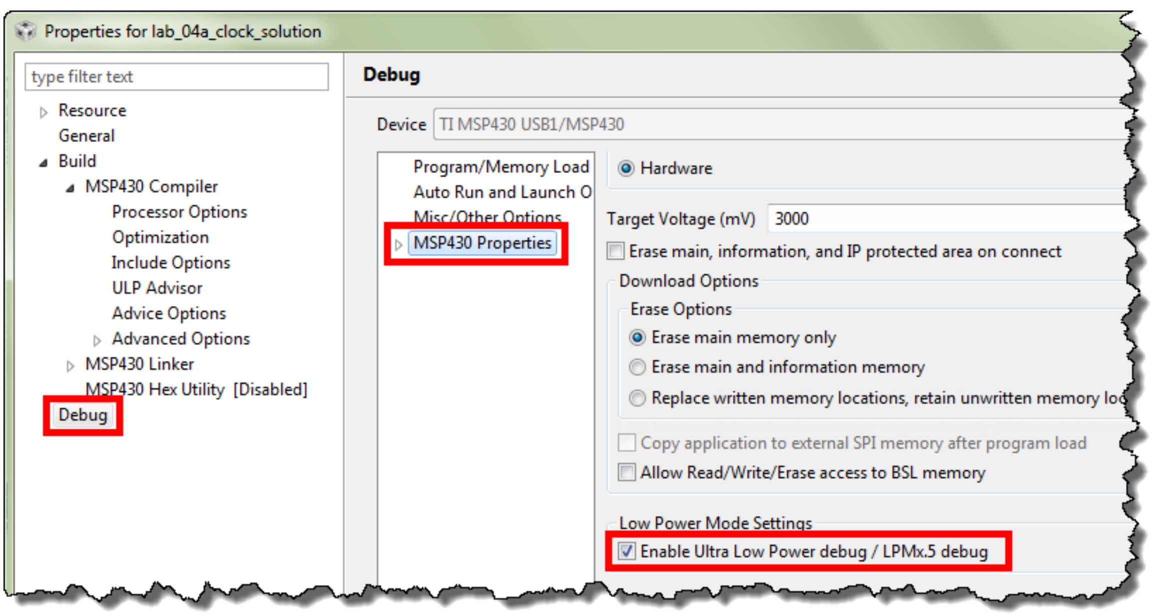


4. **('FR5969 only) Verify debugger is enabled for low-power (LPMx.5) modes.**

FR5969

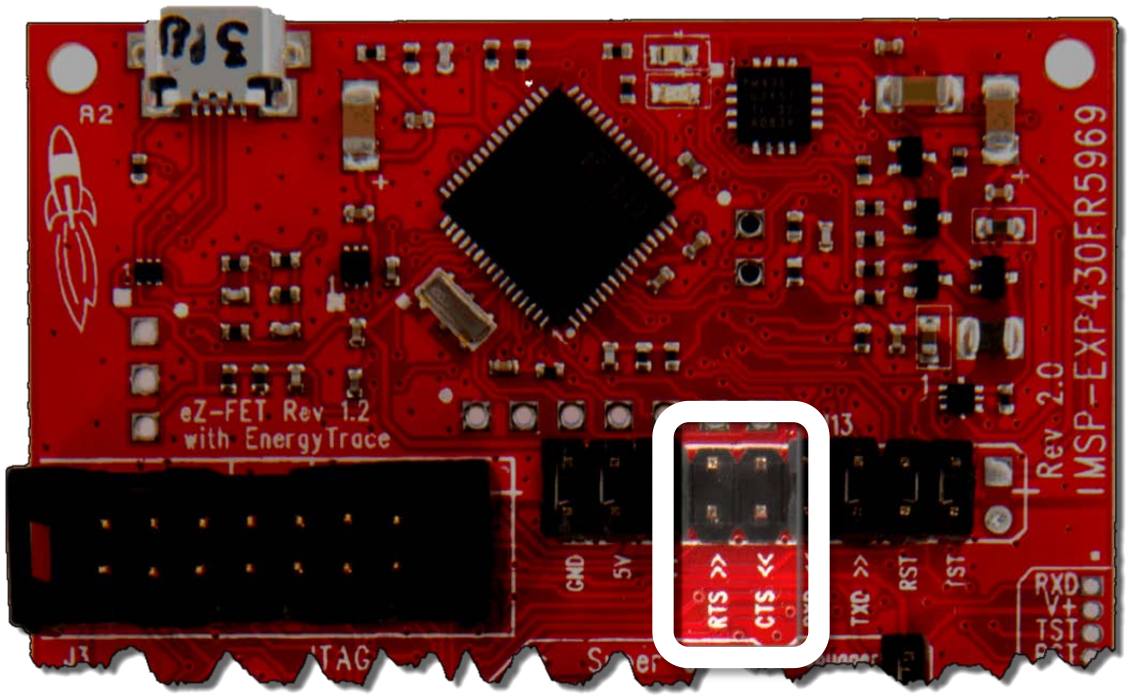
   `Right-Click on project → Properties → Debug → MSP430 Properties`

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



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

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



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

Build Project and Run with EnergyTrace

6. **Build the project.**

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

7. **Start the debugger.**

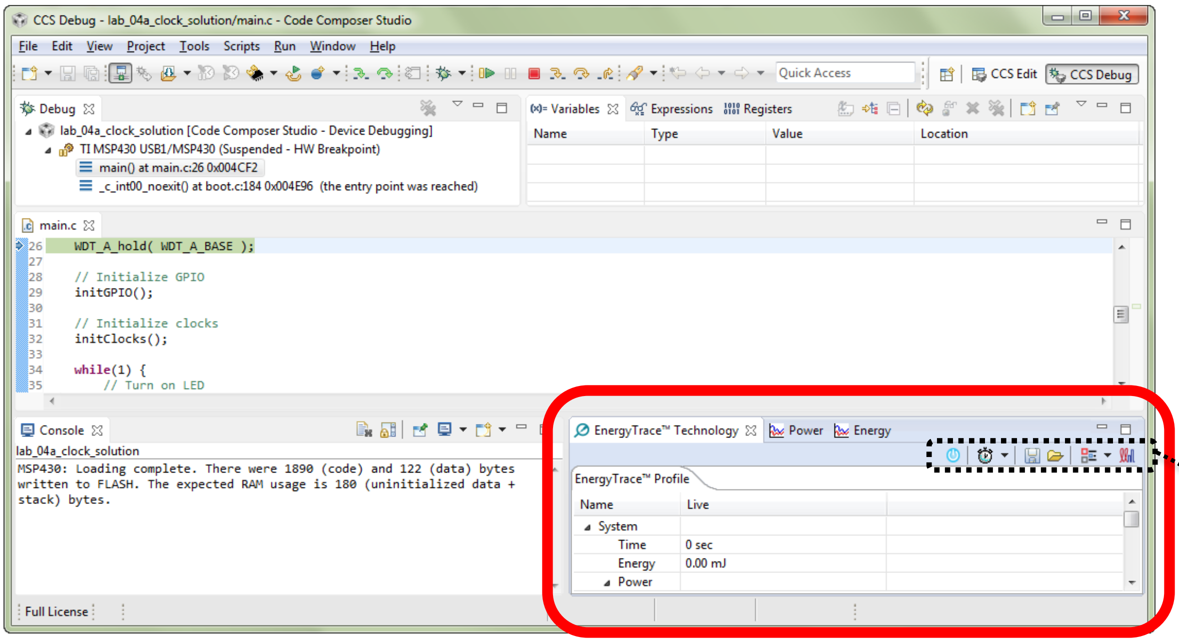
8. **Briefly examine the EnergyTrace window.**

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

   If the EnergyTrace window did not open:

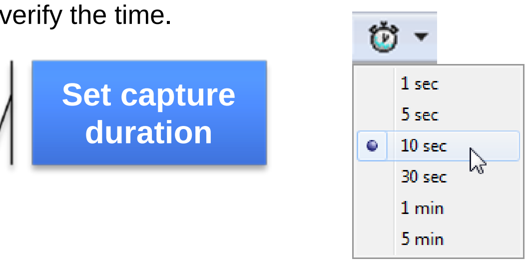
   – Double-check EnergyTrace is enabled.

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

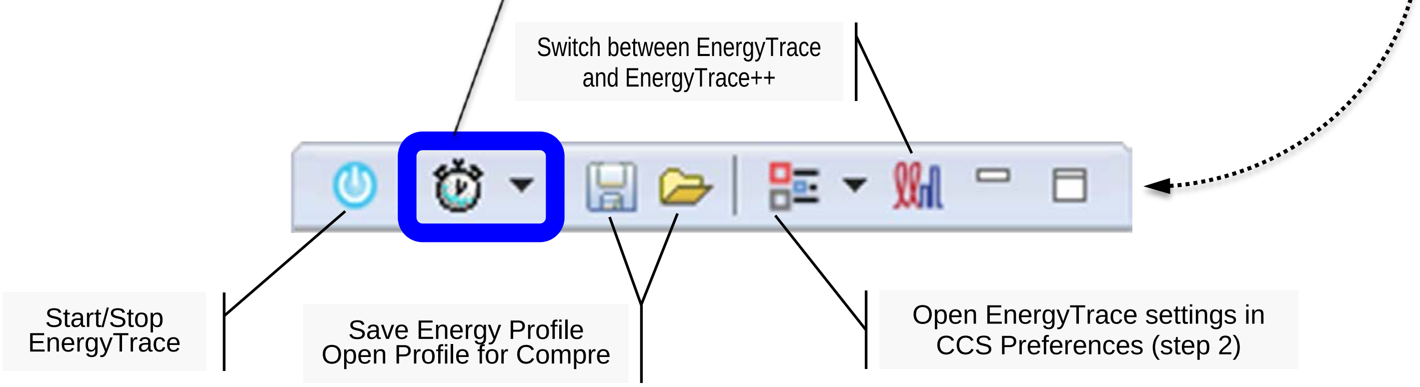


9. **Set the EnergyTrace capture duration to 10 seconds.**

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



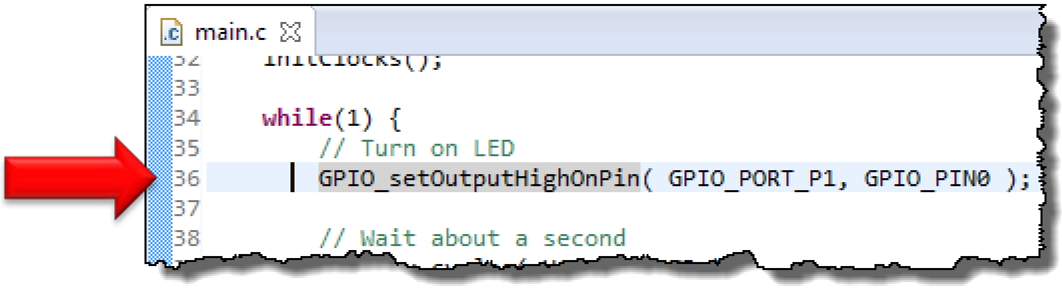
   While we're looking at the toolbar, please note some of its other buttons.



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

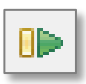
In most systems, we care more about "continuous" power usage rather than "initialization" power usage. Because of this, we want to run past our initialization code before we start collecting energy data.

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



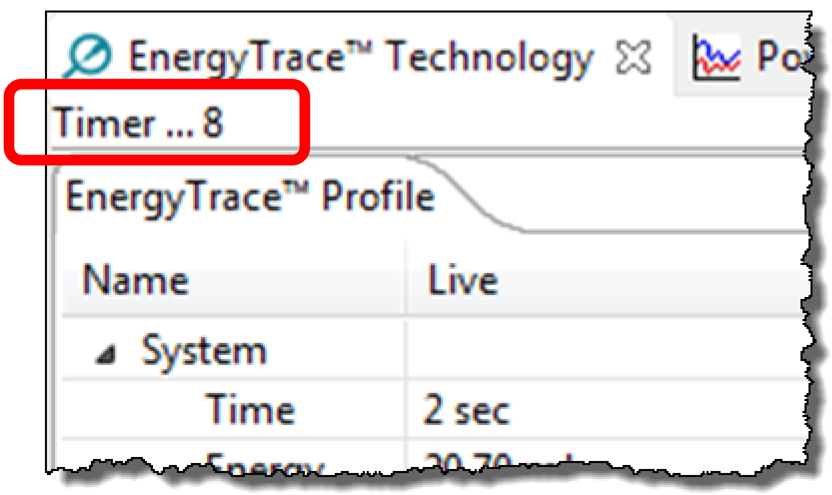
11. Run to the cursor

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



12. Click Resume and watch the duration count down.

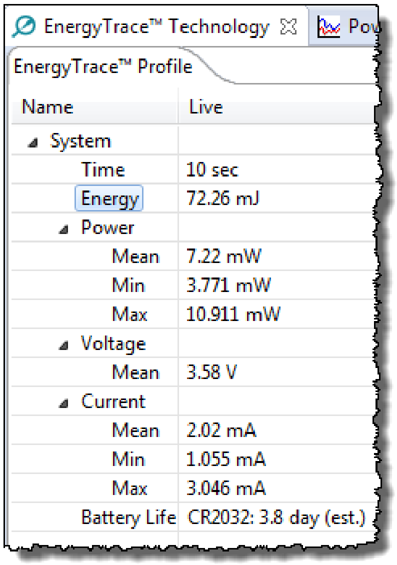
When we begin running the code it will execute the while{} loop and capture the energy data for 10 seconds.



13. Suspend your program after count reaches zero.

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

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



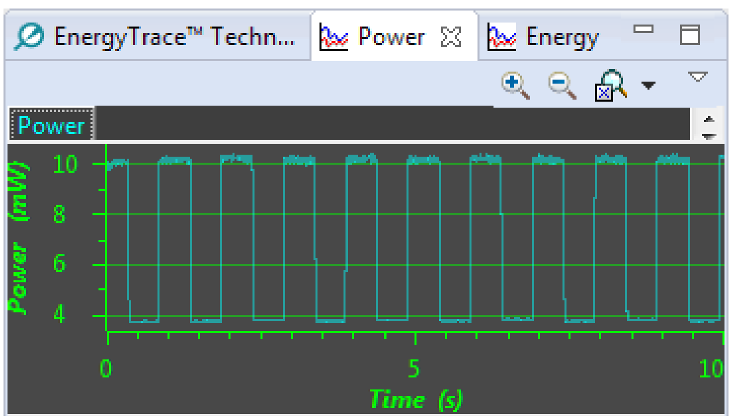
We see that our processor consumed 72.26mJ in the 10 second capture period.

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

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

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

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



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

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

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

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

EnergyTrace with Free Run

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

17. Make sure your program is suspended.

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

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

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

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

`Run → Run Free`

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

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

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

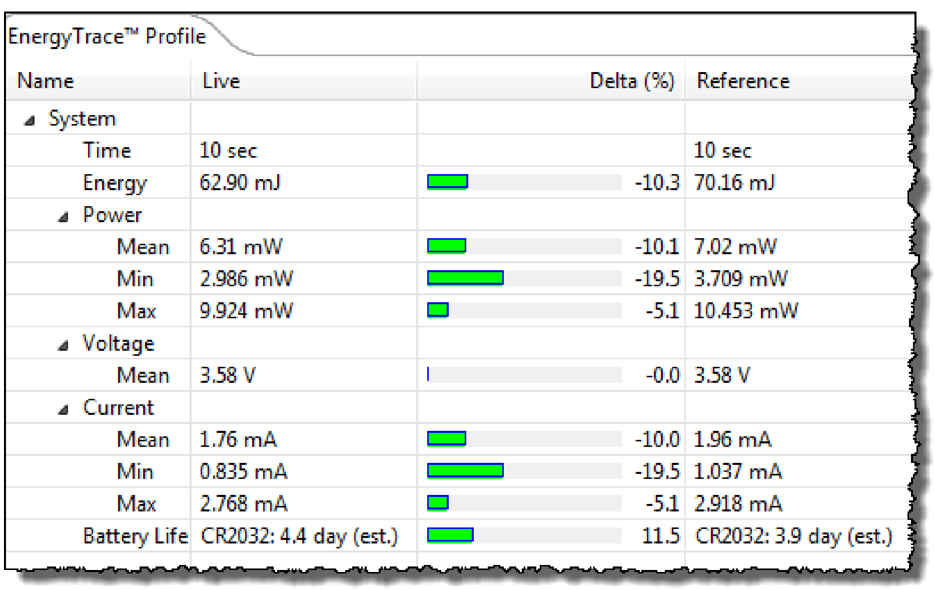
This isn't required, but it allows us to reference this information in a later comparison.

Compare EnergyTrace Profiles

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

Choose your first EnergyTrace profile: `Lab4a.profxml`

23. View the EnergyTrace profile comparison that opens.



EnergyTrace™ Profile

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

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

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

25. Terminate the debug session.

26. Close the lab_04a_clock_solution project.

Create Energy Profile for lab_04c_crystals

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

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

28. Build the project and start the debugger.

Control R

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

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

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

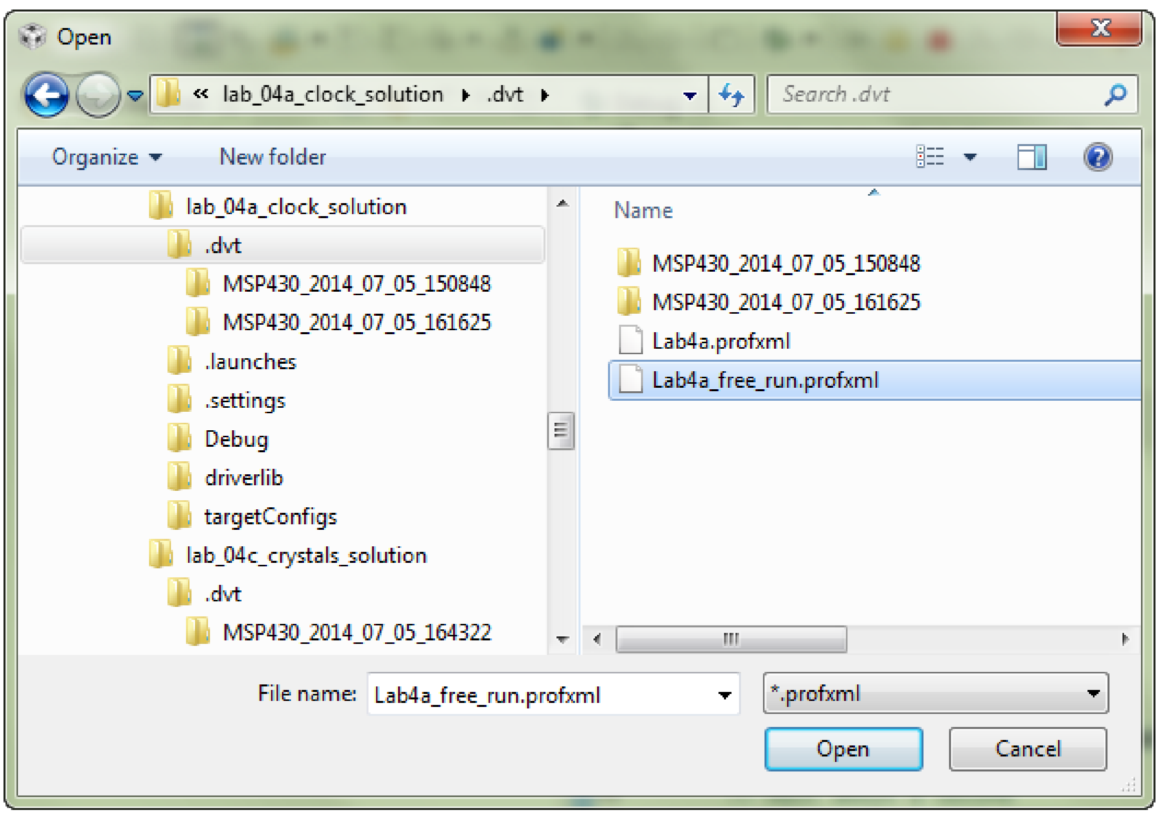
This time, rather than hitting the Resume button, we want to run our target FREE of any emulation.

`Run → Run Free`

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

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

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



34. How do the two profiles compare?

Add your values to the chart below.

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

Project Energy Profile	Time	Energy
Lab4a_free_run	10 sec	
Lab4c_free_run	10 sec	

Which version consumed less energy? ______________________________________

Why? ____

____

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

35. Terminate the debug session.

What have we learned in Lab7a?

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

FR5969

(Optional) Viewing 'FR5969 EnergyTrace++ States

Remember that the 'FR58/59xx and 'FR68/69xx devices support additional tracing of their internal CPU and peripheral states. Let's examine this great new capability.

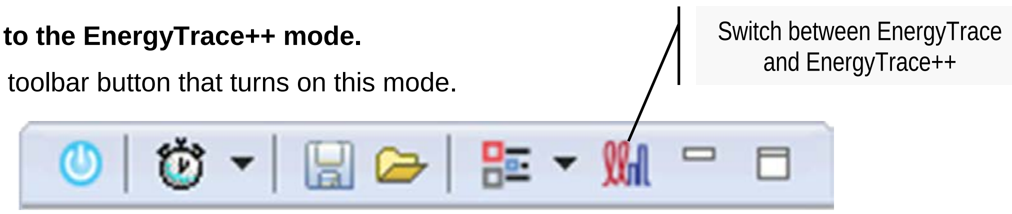
36. Open `lab_4c_crystal_solution` for debugging.

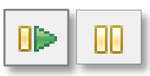
37. Verify that EnergyTrace is enabled.

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

38. Change to the EnergyTrace++ mode.

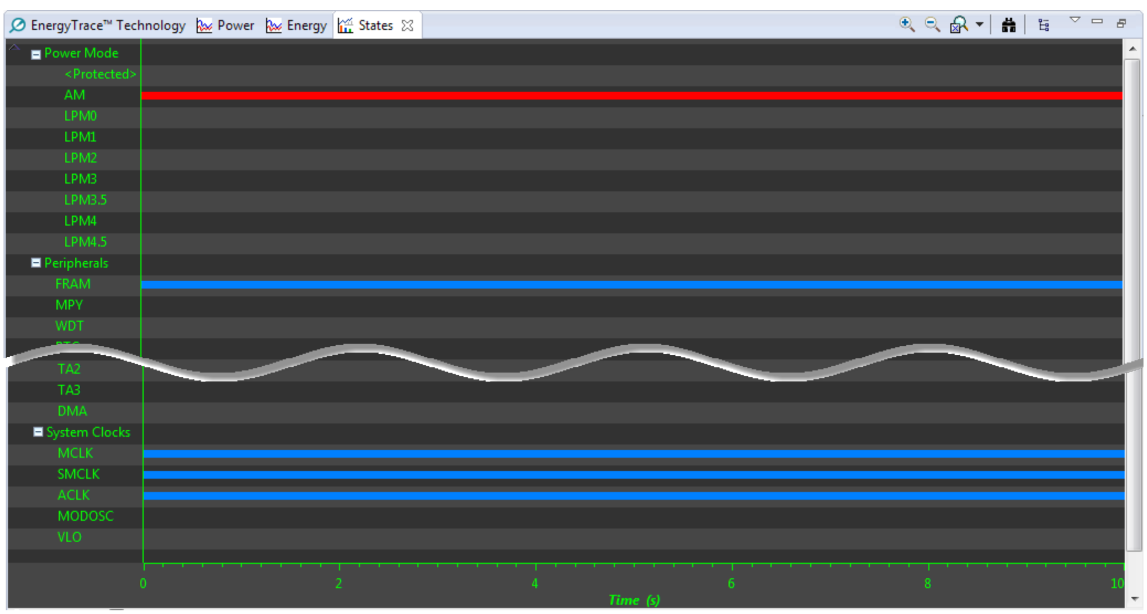
Click the toolbar button that turns on this mode.





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

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



Notice the following:

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

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

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

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

Get Suggestions from ULP Advisor

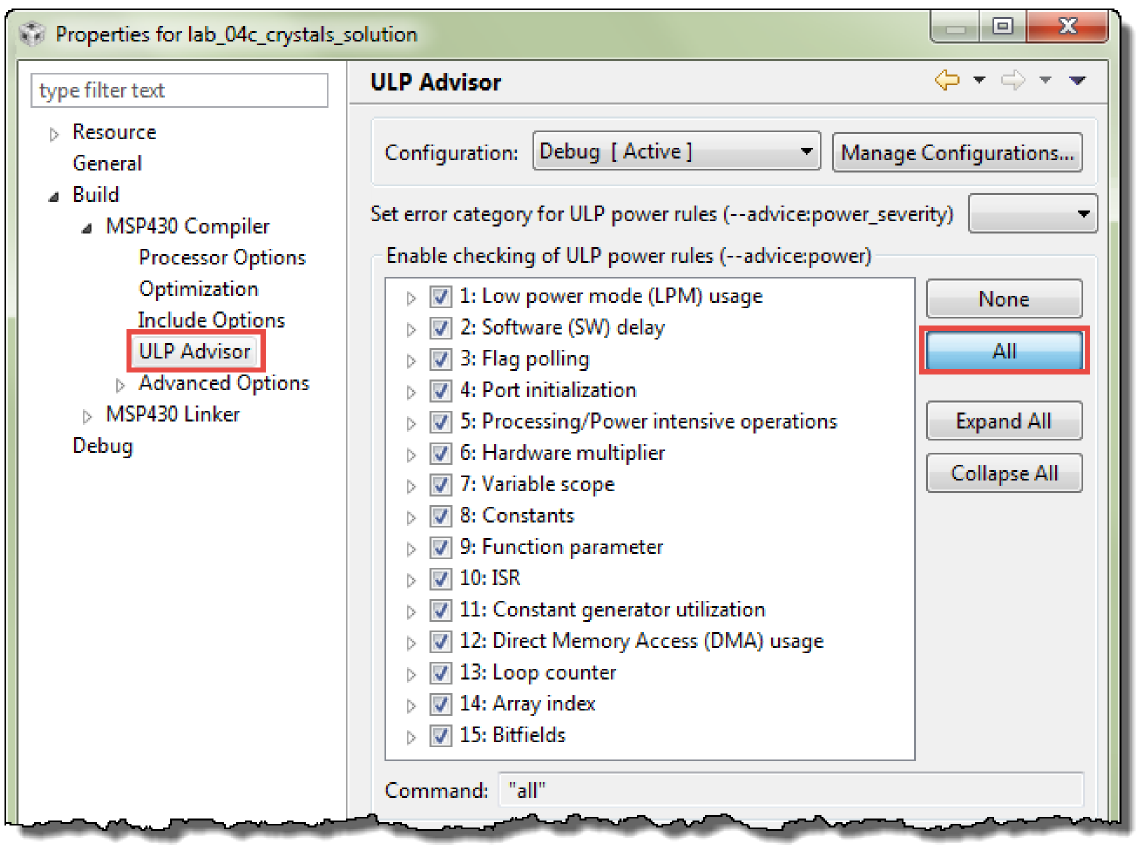
1. **Just to verify, all projects should be closed except `lab_4c_crystals_solution`; that is, the project we were just working with.**

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

Select the project `lab_4c_crystals_solution`

`Press the key combination` Alt — Enter

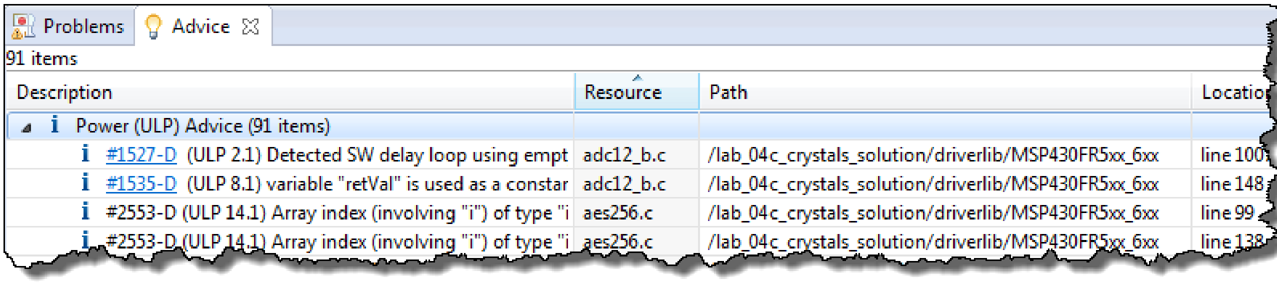
And select *All* the rules, as shown below:



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

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

```
View → Advice
```

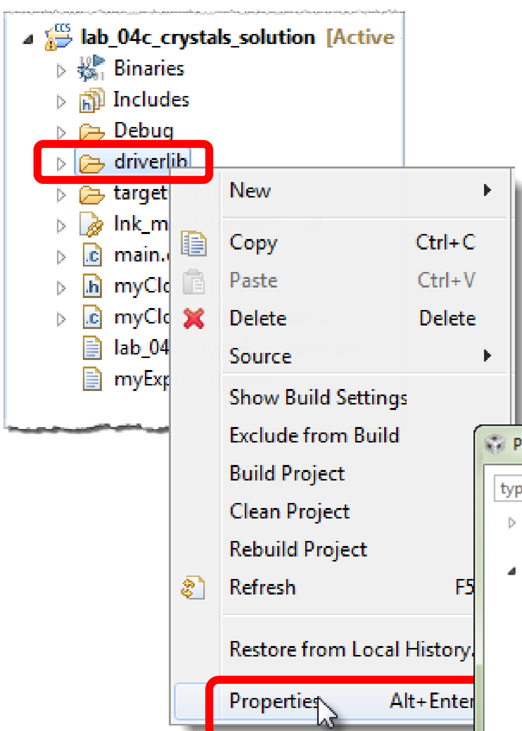


You results may vary based upon which processor you are using, but running with ULP Advisor, we received 91 items of advice. You may notice that most of the items relate to DriverLib code … further, most of them are related to peripheral source code that we're not even using in our program. (Thus, the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

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

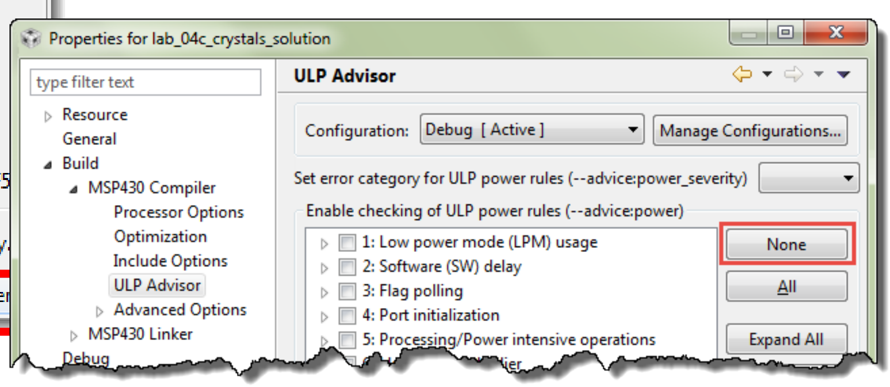
4. **Modify the project options to focus ULP Advisor on our source code.**

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



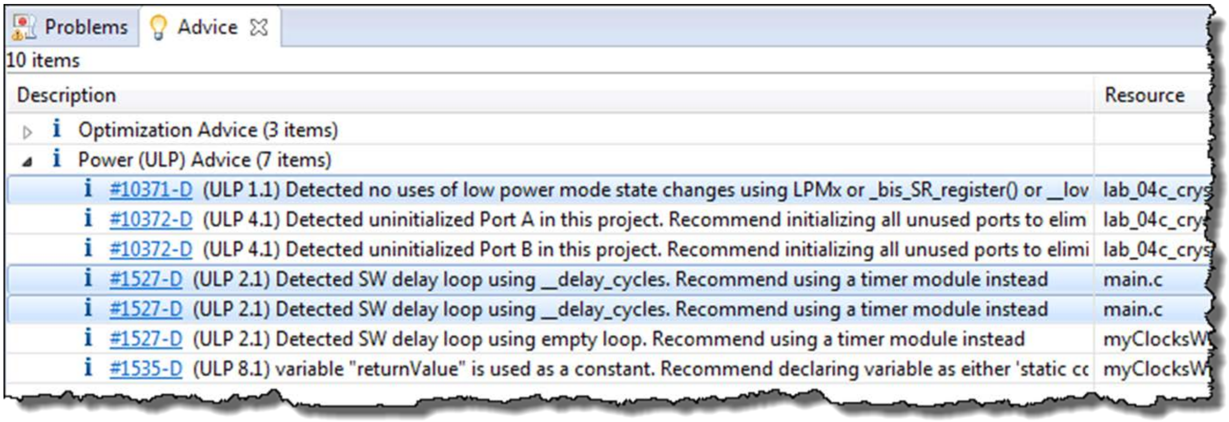

```
Right-click on the 'driverlib' folder
Select Properties
Click None
Click OK
```

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



5. **Build the project again.**

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

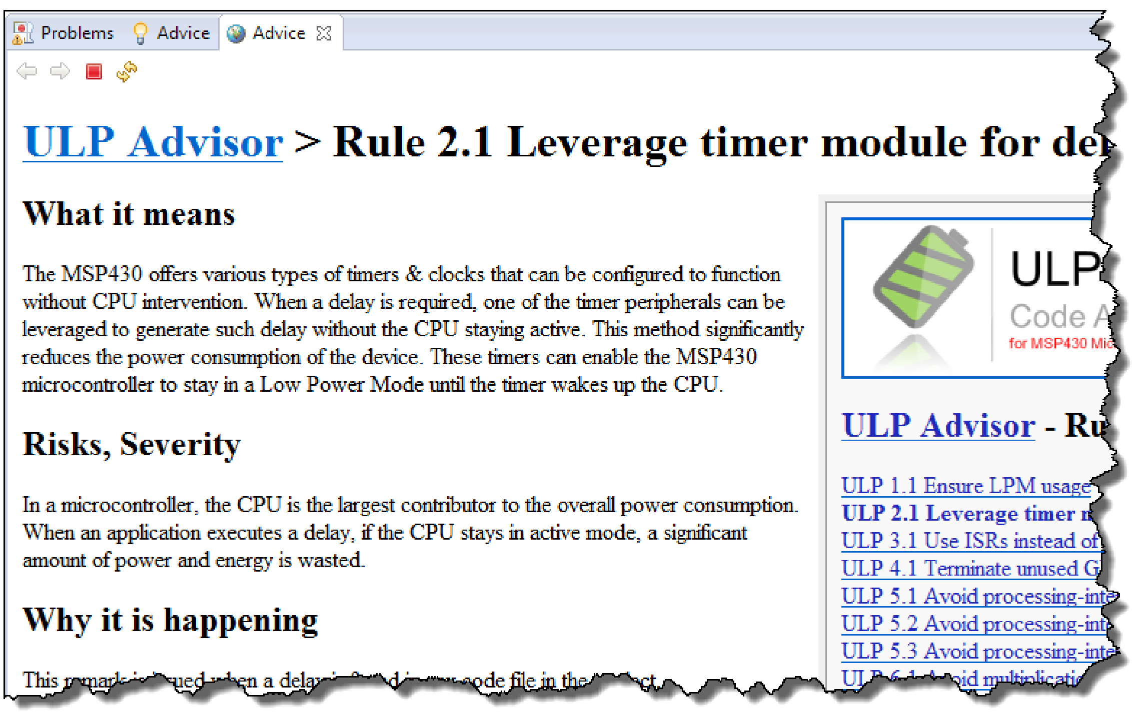


   In Lab7b, we're plan to improve upon the items highlighted above; i.e. rules ULP 1.1 and 2.1.

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

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

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



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

Replace __delay_cycles()

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

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

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

Lab Exercise	Timer Module Used
lab_05b_wdtBlink	
lab_06a_timer	
lab_06b_upTimer	
lab_06c_timerDirectDriveLed	
lab_06d_simplePWM	'F5529: Timer0_A 'FR4133: Timer0_A 'FR5969: Timer1_A

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

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

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

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

   We chose this exercise because:

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

10. Rename the project to lab_07b_lpm_timer.

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

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

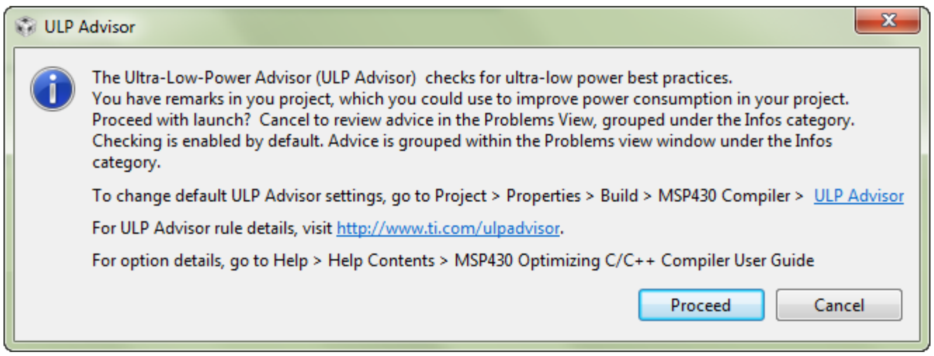
(Hint: If you need a reminder, look at Steps 2-4 (page 7-32) for how this was done.)

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

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

13. Start the debugger and load the program.

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



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

FR5969

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

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

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

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

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

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

```
Run → Run to Line
```

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

```
Run → Free Run
```

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

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

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

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

Project Energy Profile	Time	Energy
Lab4c_free_run	10 sec	
Lab7b_original	10 sec	

Which project uses more power? ______________________________

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

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

____

____

20. Terminate your debugging session.

21. Comment out the toggling of LED1.

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

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

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

myTimers.c

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

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

Note

Shown here is the 'FR5969 code.

If using the 'F5529 or 'FR4133 you'll be using Timer0 and LED1 uses a different Port/Pin on each.

22. Build your project and fix any syntax errors.

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

Control R

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

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

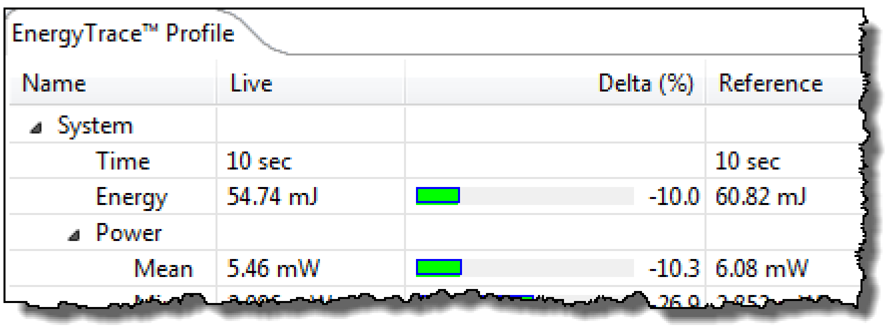
Once again, compare this to the Lab4c energy profile.

Project Energy Profile	Time	Energy
Lab4c_free_run	10 sec	
Lab7b_one_led	10 sec	

Which project uses more power? ______________________________

Here's the comparison we found for the 'FR5969 at the time of writing this exercise. As you can see below, using the timer (versus the CPU running __delay_cycles) saved us 10% of our energy. You should see similar improvements for the 'FR4133, as well.

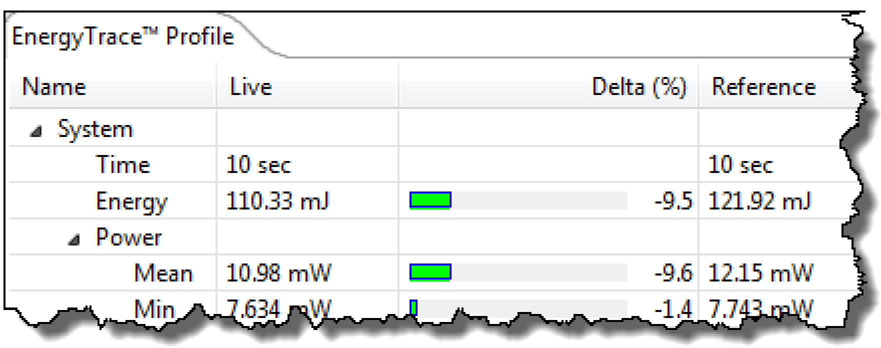
FR5969



EnergyTrace™ Profile

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

F5529



EnergyTrace™ Profile

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

Using Low-Power Mode (LPM3)

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

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

26. Modify `lab_07b_lpm_timer` to use LPM3.

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

```
main.c
    __bis_SR_register( GIE );

    while(1) {
        //__no_operation();
        __low_power_mode_3();
    }
```

As we learned during the Chapter 7 discussion:

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

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

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

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

27. Build your code and fix any syntax errors.

28. Start the debugger.

Control R

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

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

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

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

Project Energy Profile	Time	Energy
Lab7b_one_led	10 sec	
Lab7b_lpm	10 sec	

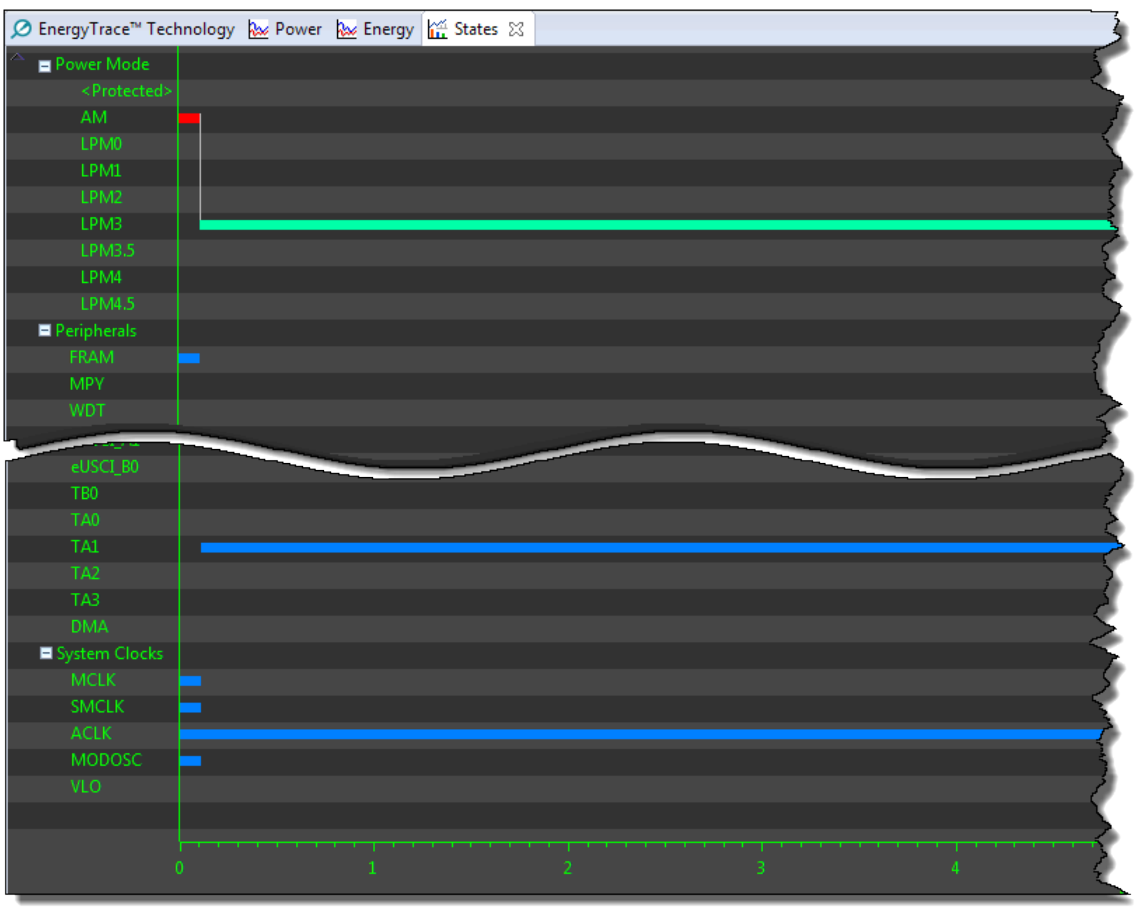
Which profile uses less power? ____

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

FR5969

(Optional) Viewing 'FR5969 EnergyTrace++ States

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



(Optional) Directly Driving the LED from Timer_A

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

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

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

Rough lab exercise procedural

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

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

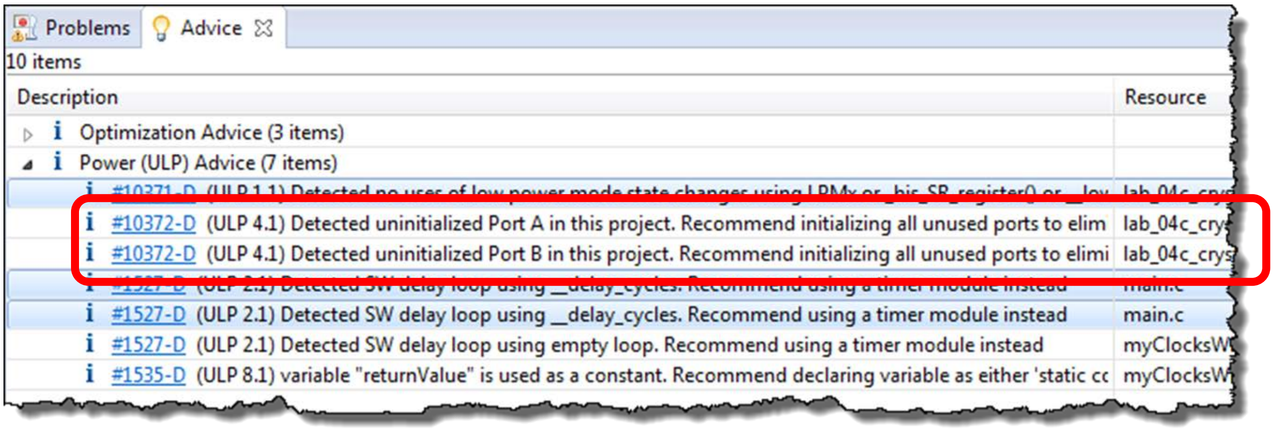
  When we did this, we found that (using the 'FR5969 Launchpad) the directly driven LED project took quite a bit more energy … these results shocked us.

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

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

Lab 7c – Configuring Ports for Lowest Power

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



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

Import and Modify Program

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

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

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

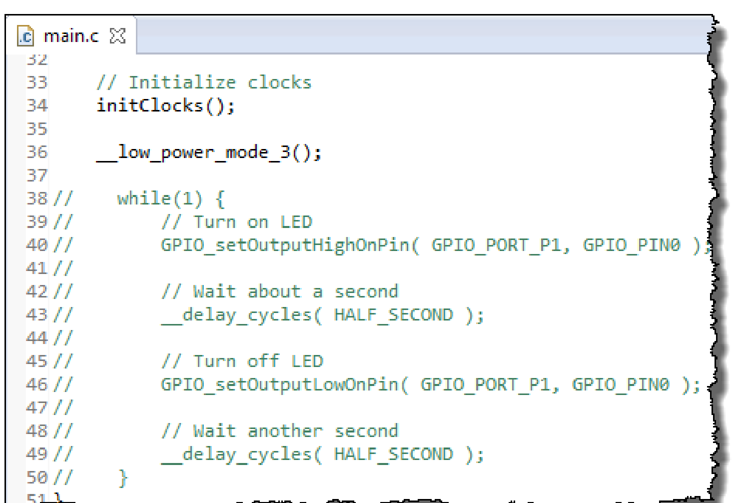
   a) In CCS Project Explorer, `right-click and copy lab_04c_crystals_solution`

   b) Then `right-click and paste it`

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

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

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



Capture Baseline Reference

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

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

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

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

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

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

Add GPIO Port Initialization Code

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

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

9. Add three new files to your project.

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

Navigate to the appropriate directory for you processor:

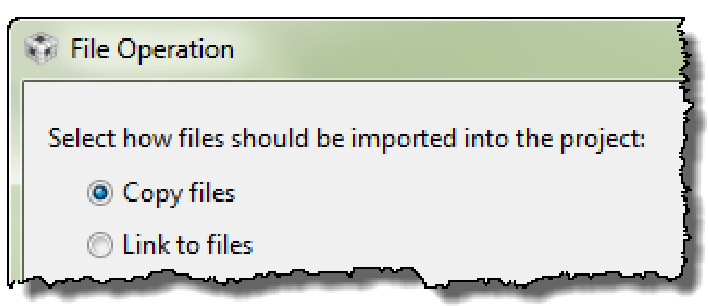
```
C:\msp430_workshop\<target\lab_07c_ports
```

Select the following three files and click *Open*.

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

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

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



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

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

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

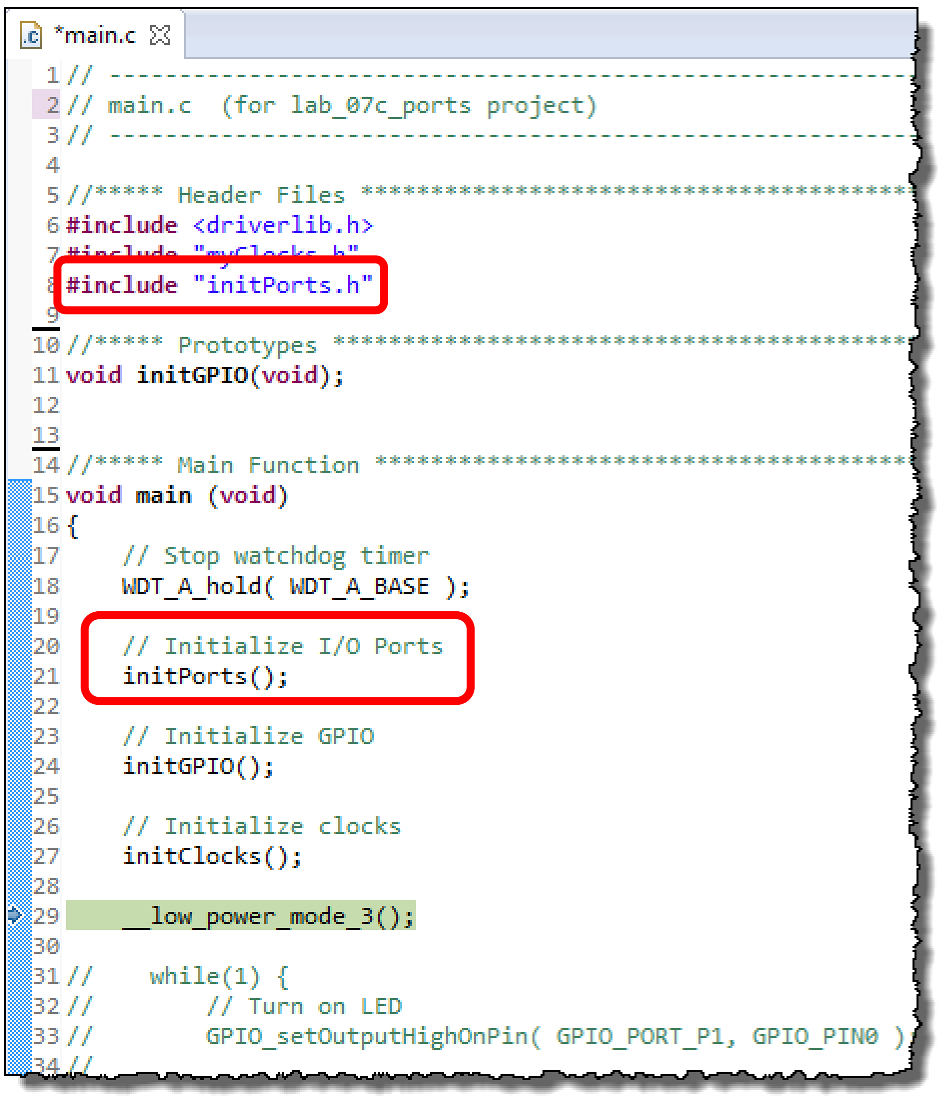
While we've added the files to the project, we haven't add the call to the initPorts() function, yet. Immediately after the Watchdog hold function, add the new function to your program.

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

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

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

Your main() function should now look like this:



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

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

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

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

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

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

____

Improve on GPIO Port Initialization

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

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

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

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

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

____

17. Exclude from build...

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

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

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

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

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

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

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

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

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

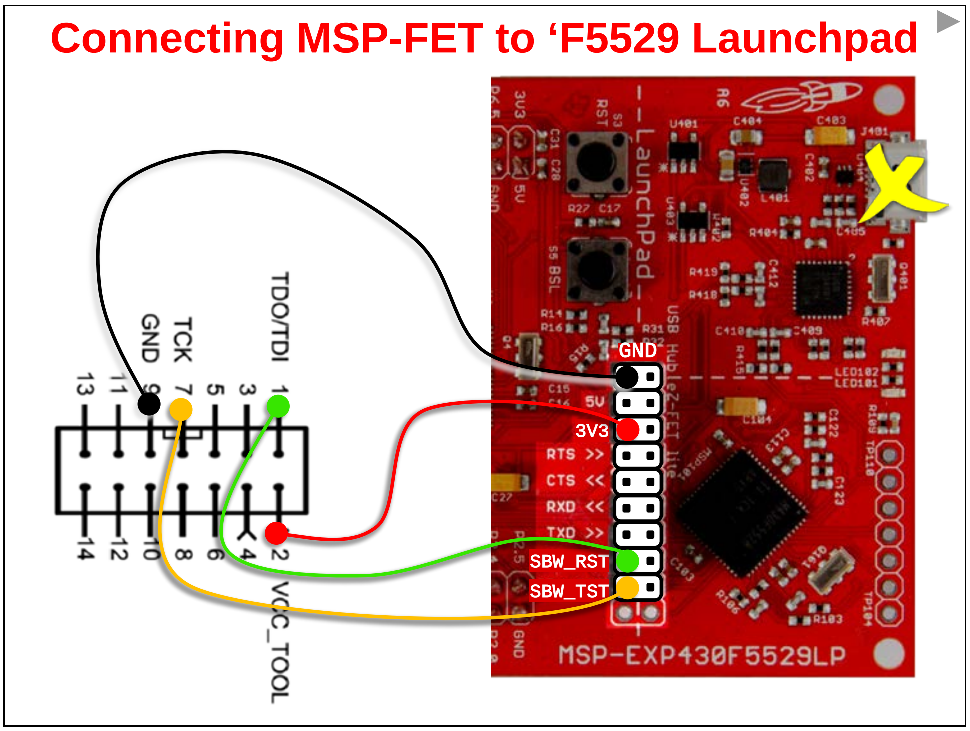
____

Chapter 7 Appendix

Connecting MSP-FET to 'F5529 USB Launchpad

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

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



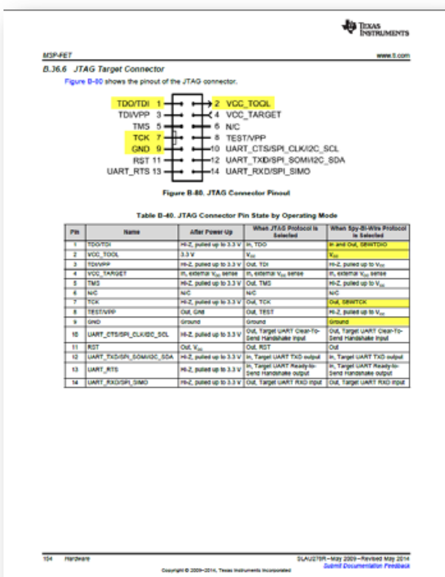
MSP-FET to 'F5529 Launchpad Summary of Pin Connections

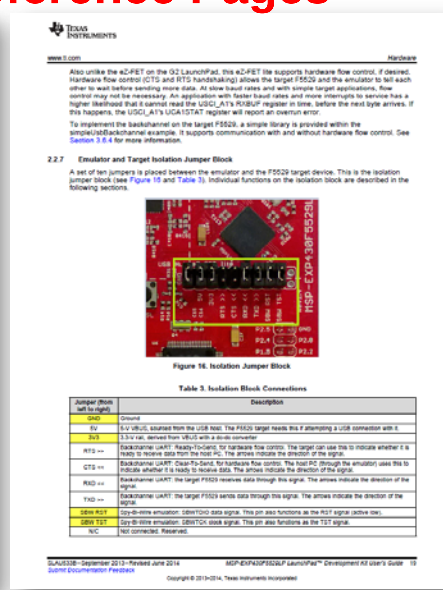
MSP-FET Signal	MSP-FET Pin		'F5529 Launchpad (Isolation Jumper Block) Signal	'F5529 Launchpad Pin
GND	9	→	GND	JP3
VCC_TOOL	2	→	3V3	JP2
TDO/TDI	1	→	SBW_RST	JP4.2
TCK	7	→	SBW_TST	JP4.1

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

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

User Guide Reference Pages





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

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

Lab 7 Debrief and Solutions

Lab 7a - Worksheet

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

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

   Which devices support Energy Trace++? **MSP430FR5xx/FR69xx devices**

2. What hardware options are available that supports Energy Trace? **'FR5969 Launchpad, 'FR4133 Launchpad, and any MSP430 connected to the MSP-FET**

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

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

Lab 7a – Debrief ('FR5969)

34. How do the two profiles compare?

Add your values to the chart below.

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

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

Which version consumed less energy? **Lab4c**

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

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

Lab 7a – Debrief (‘FR4133)

34. How do the two profiles compare?

Add your values to the chart below.

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

Project Energy Profile	Time	Energy
Lab4a_free_run	10 sec	74.39 mJ
Lab4c_free_run	10 sec	75.22 mJ

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

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

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

Lab 7a – Debrief (‘F5529)

34. How do the two profiles compare?

Add your values to the chart below.

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

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

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

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

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

Lab 7b

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

Lab Exercise	Timer Module Used
lab_05b_wdtBlink	Watchdog (Interval Timer mode)
lab_06a_timer	'F5529: Timer0_A 'FR4133: Timer0_A 'FR5969: Timer1_A
lab_06b_upTimer	'F5529: Timer0_A 'FR4133: Timer0_A 'FR5969: Timer1_A
lab_06c_timerDirectDriveLed	'F5529: Timer0_A 'FR4133: Timer0_A 'FR5969: Timer1_A
lab_06d_simplePWM	'F5529: Timer0_A 'FR4133: Timer0_A 'FR5969: Timer1_A

Lab 7b

'F5529 values are shown here

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

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

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

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

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

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

Lab 7b

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

'F5529 values are shown here

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

Which profile uses less power? **Lab7b_lpm is much better**

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

Lab 7c ('FR5969)

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

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

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

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

Lab 7c ('FR4133)

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

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

Project Energy Profile	Capture Duration Time	Energy (mJ)	Battery Life (Days)
Lab7c_noinit	10 sec	0.69	401.1
Lab7c _initPortsAsOutputs	10 sec	5.20	52.9
Lab7c_initPortsAsInputs	10 sec	0.05	5589.4

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

Lab 7c ('F5529)

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

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

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

Steps 13/19 asked if initializing the GPIO made much of a different to energy usage… a little bit. On the 'F5529, though, no noticeable difference if GPIO was set as outputs or inputs (unlike the 'FR4133 or 'FR5969).

Real-Time Clock (RTC)

Introduction

The Real-Time Clock (RTC) peripheral is a sophisticated timer that keeps track of Calendar, Month, and Time information. It operates in Binary or BCD modes; whichever is most useful for your application.

The RTC affords you the ability to set Calendar/Time based Alarms (i.e. Interrupts).

This peripheral is extremely power sensitive and operates in many low-power modes. In fact, on the MSP430FR5969, it even operates in LPM3.5 mode.

Learning Objectives

- Describe the architecture of the Real-Time Clock module.
- Learn to set alarms/interrupts for the RTC.

Chapter Topics

Real-Time Clock (RTC) 8-1

What is a Real-Time Clock? 8-3

How Does the RTC Work? 8-4

RTC Block Diagram 8-4

RTC Interrupts 8-5

Programming the RTC 8-6

Additional Considerations 8-7

Summary 8-8

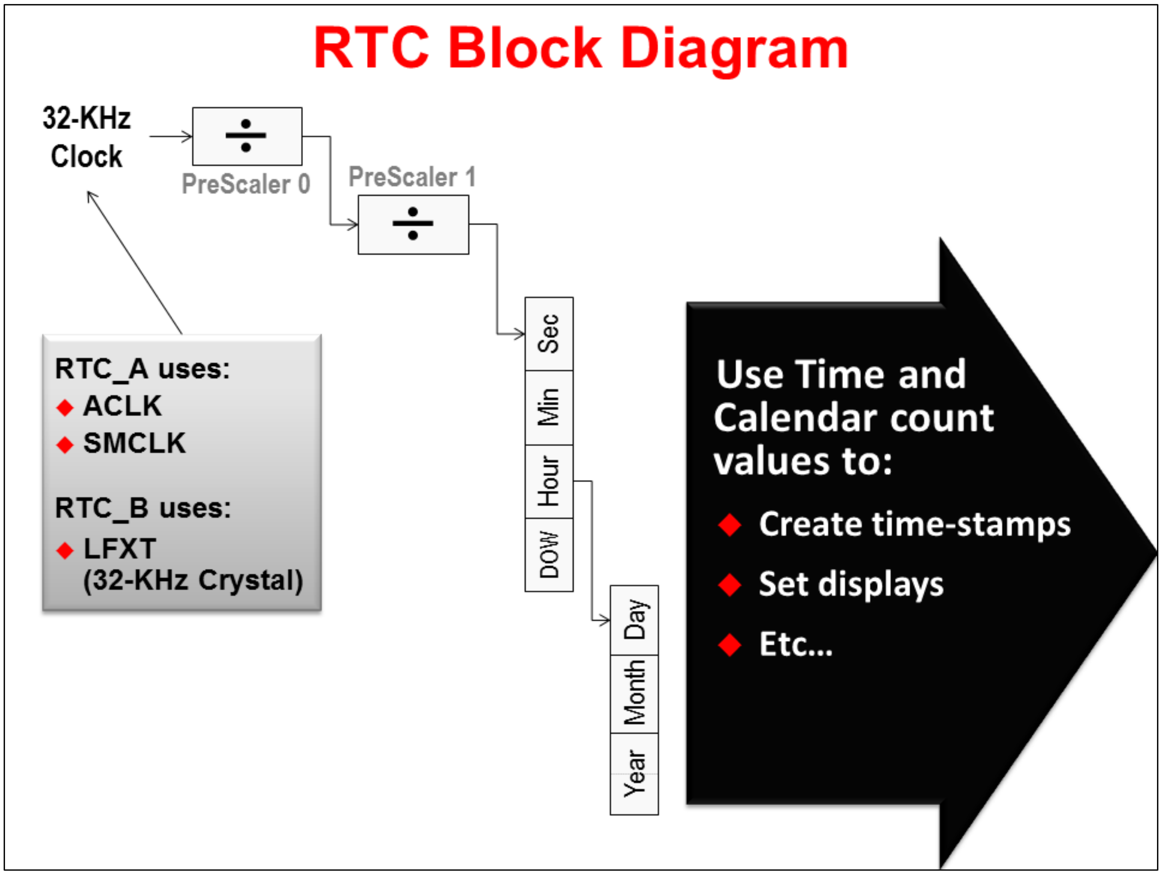
What is a Real-Time Clock?

What is a Real-Time Clock (RTC)?

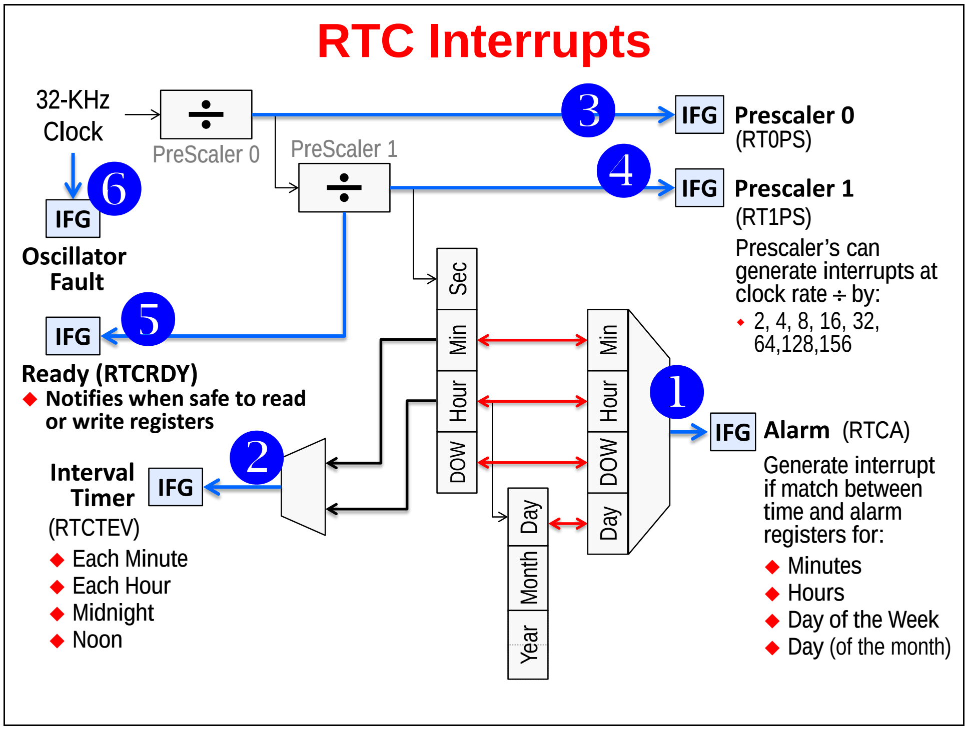
It's just that...
an alarm Clock
with Calendar functions

How Does the RTC Work?

RTC Block Diagram

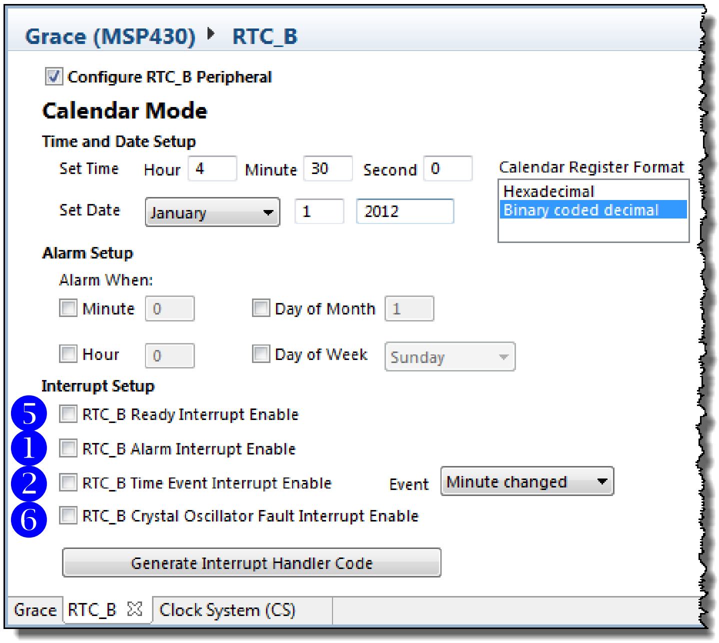


RTC Interrupts



Programming the RTC

Setting RTC using GRACE



Grace supports:

- Devices
  - F2xx, G2xx
  - FR5xx
- RTC use cases
  - Calendar mode only
- BCD or Hex modes
- Creates interrupt handler template:
  - Alarm
  - Events
  - Ready
  - Osc Fault

DriverLib Example Code

```
// Initialize Calendar Mode of RTC
RTC_B_calendarInit ( RTC_B_BASE, currentTime, RTC_B_FORMAT_BINARY);

// Setup Calendar Alarm for 5:00pm on the 5th day of the week.
// Note: Does not specify day of the week.
RTC_B_setCalendarAlarm ( RTC_B_BASE,0x00,0x17, RTC_B_ALARMCONDITION_OFF,0x05);

// Specify an interrupt  to assert every minute
RTC_B_setCalendarEvent ( RTC_B_BASE, RTC_B_CALENDAREVENT_MINUTECHANGE);

// Clear interrupt bits before starting RTC
RTC_B_clearInterrupt ( RTC_B_BASE, RTC_B_CLOCK_READ_READY_INTERRUPT + RTC_B

// Enable interrupt for RTC Ready Status, that let's us know RTC registers are ready to read.
// Also, enable interrupts for the Calendar alarm & event.
RTC_B_enableInterrupt ( RTC_B_BASE, RTC_B_CLOCK_READ_READY_INTERRUPT + RTC_

// Start RTC Clock
RTC_B_startClock ( RTC_B_BASE);

// Enter LPM3 mode with interrupts enabled
__low_power_mode_3 ();
__no_operation();
```

Additional Considerations

Additional Features

- **Using RTC in LPM3.5 Mode**
  - **All RTC's work in LPM3 mode**
  - **Since RTC_B and RTC_C can directly access the LF crystal, they can operate in the "3.5" low-power mode**
  - **LPM3.5 provides the lowest possible power dissipation with RTC wake-up capability**
- **Easy Conversion Between BCD and Hex**
  - **RTC_B/RTC_C provide BCD conversions in hardware**
  - **Driver Library function provides easy access to this hardware feature**
- **Counter Mode**
  - **RTC_A can be used as a 32-bit counter (rather than Calendar mode)**
  - **Counter mode generates overflow interrupts at 8-, 16-, 24- and 32-bits**

Exercise Caution

- **Clear bit-fields before setting counters and alarms**
  - Prior to setting an alarms, clear all alarm registers, including the alarm enable (AE) bits
  - To prevent potential erroneous alarms when setting time values, clear the interrupt enable (IE) bits, as well as the AE bits
  - Writes to count registers takes effect immediately. Note that the RTC clock is stopped during the write and both pre-scale registers are reset. This could result in losing up to 1 second during a write.
- **Invalid time and alarm settings are not validated or handled via hardware (measure twice, program once)**
- **Reading Registers**
  - Care should be taken when reading (or writing) RTC time/calendar/prescale registers so that your actions do not occur during counter transitions
  - These options can help to prevent erroneous results:
    1. Let the RTC Ready (RTCRDY) interrupt you just after an update – you'll have ~1 sec before the next update
    2. Check the RTCRDY bit before reading or writing the registers
    3. Read the registers multiple times and take the majority vote
    4. Hold the RTC before reading or writing any registers

Summary

RTC Comparison

	Feature	RTC_A	RTC_B	RCT_C
	Highlights	32-bit Counter Mode	LPM3.5, Calendar Mode Only	Protection Plus Improved Calibration & Compensation
Modes	**Calendar Mode with Programmable Alarms**	Yes	Yes	Yes
Modes	**Counter Mode**	Yes	No	Device-dependent
	Input Clocks	ALCK, SMCLK	32-kHz crystal	32-kHz crystal
	LPM3.5 Support	No	Yes	Yes
Compensation & Calibration	**Offset Calibration Register**	Yes	Yes	Yes
Compensation & Calibration	**Temperature Compensation Register**	No	No	Yes
Compensation & Calibration	**Temperature Compensation**	With software, manipulating offset calibration value		With software using separate temperature compensation register
Compensation & Calibration	**Calibration and Compensation Period**	64 min	60 min	1 min
Features	**BCD to Binary Conversion**	Integrated for Calendar Mode	Integrated for Calendar Mode plus separate conversion registers	
Features	**Event/Tamper Detect With Time Stamp**	No	No	Device-dependent
Features	**Password Protected Calendar Registers**	No	No	Yes

Non-Volatile Memory: Flash & FRAM

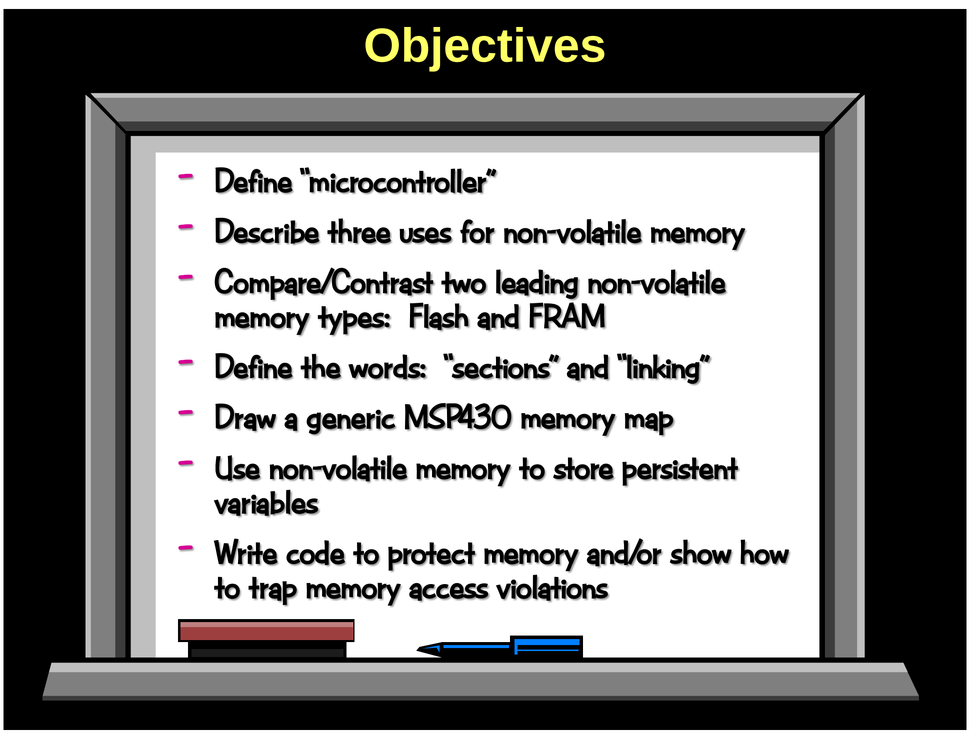
Introduction

What makes a microcontroller a microcontroller? That's part of this chapter's discussion. The inclusion of memory – especially non-volatile memory – makes a microprocessor into a microcontroller.

Non-volatile memory (NVM for short) is an important part of a microcontroller's memory system; this type of memory stays initialized (i.e. keeps its data) even when power is removed from the device. Storing program code is the most obvious use of NVM, though many applications store data tables and calibration data in NVM, as well.

Flash technology is the most common type of NVM used in today's microcontrollers. In the last couple of years, though, Texas Instruments has introduced the use of FRAM technology into their MSP430 microcontroller family. With near infinite write cycles and extremely low power dissipation, it is a great fit for many end applications.

Learning Objectives



Chapter Topics

Non-Volatile Memory: Flash & FRAM 9-1

What is a Microcontroller? 9-3

Non-Volatile Memory: Flash & FRAM 9-4
Flash Memory 9-5
FRAM Memory 9-6
Comparing FRAM and Flash 9-7
FRAM Benefits and Applications 9-8

Memory Maps & Linking 9-10
Memory Maps 9-10
How is NVM Used? 9-11
Comparing Device Memory Maps 9-13
Sections 9-14
Linking 9-16
Linker Command File 9-16
Custom Sections 9-18

Using Flash 9-20
Using DriverLib to Write to Flash 9-22

Using FRAM (and the MPU) 9-23
FRAM Controller 9-23
Unified Memory 9-24
What Could Happen to FRAM? 9-25
Memory Protection Unit (MPU) 9-26
Using the Memory Protection Unit (MPU) 9-27
MPU Graphical User Interface 9-30
FRAM Code Example 9-32
Configuring the MPU using DriverLib 9-33
Putting Variables into FRAM 9-35
Setting FRAM Waitstates 9-37

Memory Protection on the 'FR2xx/4xx 9-39

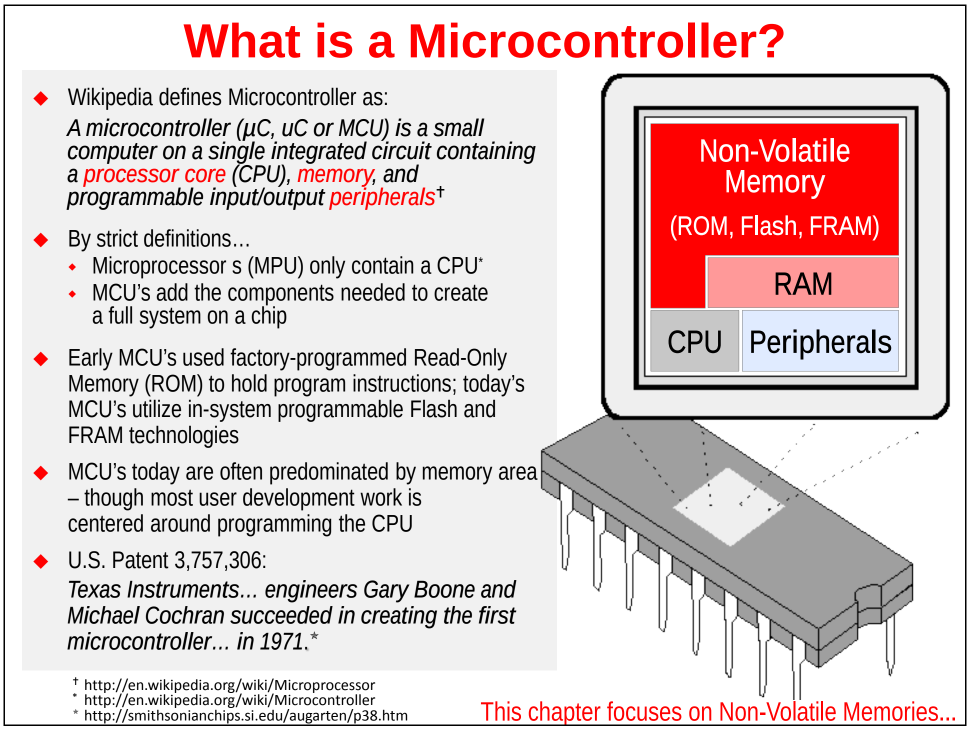
System Init Functions 9-40

Lab 9 Exercises 9-41
Lab 9a – Using Non-Volatile Variables 9-42
lab_09a_info_fram (or lab_09a_info_flash) 9-42
(FRAM Devices Only) lab_09a_persistent 9-49
('F5529 Only) (Optional) lab_09a_low_wear_flash 9-52
('FR5969 Only) Lab 9b – Protecting Memory 9-53
lab_09b_mpu_gui 9-53
(Optional) lab_09b_mpu_with_driverlib 9-56

Chapter 9 Appendix 9-59

What is a Microcontroller?

Texas Instruments was awarded the patent for the microcontroller (which we'll nickname MCU) when Gary Boone and Michael Cochran accomplished building a processor that contained memory and peripherals. The inclusion of these two items causes a microprocessor to be called a microcontroller.

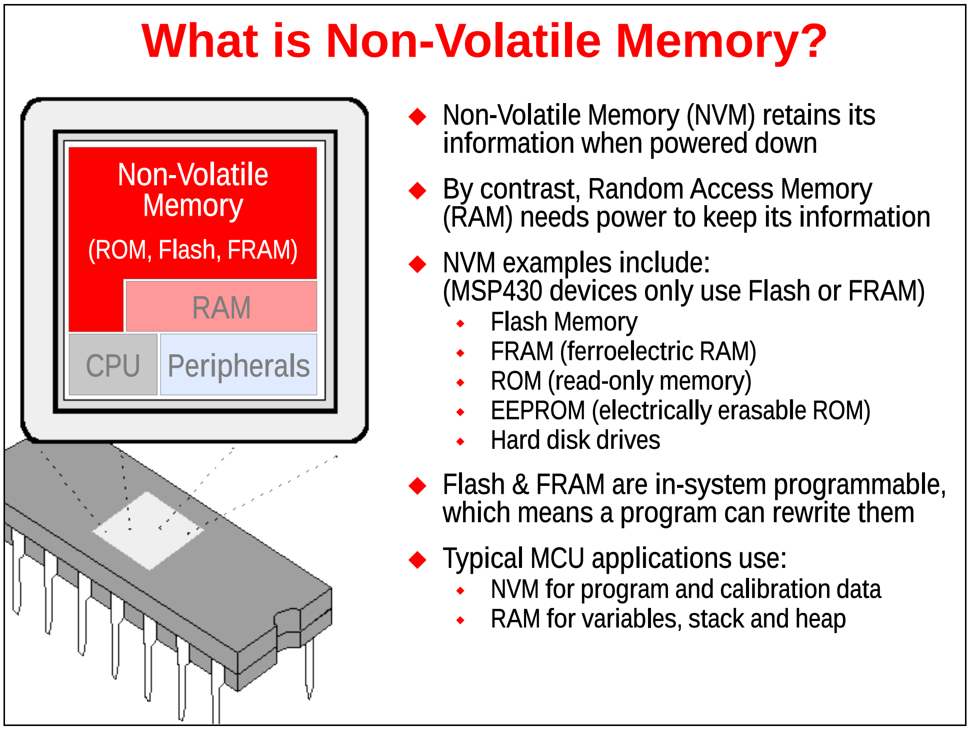


The earliest microcontrollers used ROM (read-only memory) which was programmed into the device as part of the processor's manufacturing. High volume was required to make this worth the cost.

Non-Volatile Memory: Flash & FRAM

Non-Volatile Memory (NVM) retains its information, even when power is removed. This is different than RAM (e.g. SRAM, DRAM) memory which loses its information when powered down.

NVM is important for storing your microcontroller's code. It doesn't do much good to write the code into a microcontroller if it disappears whenever the processor is turned off. Microprocessors solve this problem by using external non-volatile memory, which has to be loaded up each time the processor starts up. This is unattractive in many applications since it raises the cost and greatly increases start-up time.



Users really needed a way to program (and erase) their processor memories themselves. This need has driven a number of enhancements in NVM since the early days of ROM's.

MCU's adopted Erasable/Programmable Read-Only Memory (EPROM). These devices had a little window over the silicon that allowed the user to erase the program with a UV light. The code could be programmed electrically with a special stand-alone programmer. Due to a demand for low-cost, EPROM chips ended up being packaged in plastic without a window; these were commonly known as OTP's – for one-time programmables.

Nowadays, Flash memory technology is used by most microprocessors. This allows processors to be programmed – and erased – electronically. Companies can purchase "empty" devices and program them on their own; erasing them and re-programming, as needed.

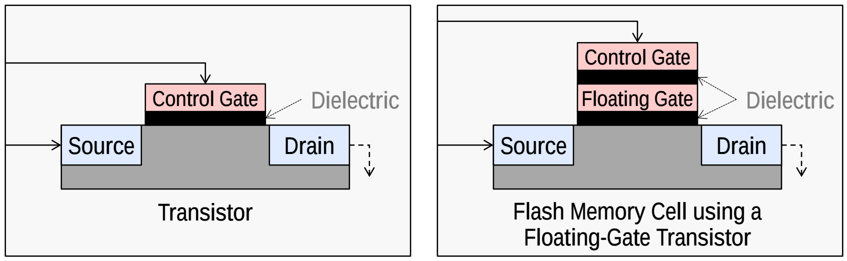
While Flash was a major step forward in NVM technology, it has a few limitations, such as power-hungry writes and limited endurance (i.e. the number of times you can erase and re-write the memory).

FRAM technology, which has been available for a decade in stand-alone devices, is now available from Texas Instruments in their MSP430 line-up. With low-power in its DNA, FRAM technology is a natural fit for many MSP430 applications.

Flash Memory

Flash memory made it cheap and convenient to create microcontrollers that were electrically erasable and programmable.

How Flash Memory Works



- In this simplified example of a Flash Memory cell, the addition of a floating gate makes it "sticky"
- Dielectric provides insulation allowing the floating gate to remain charged (or not) for a very, very long time
- Overcoming the dielectric to erase/charge the floating gate requires a high voltage (~14 Volts); Flash-based processors contain charge pumps to reach these high voltages
- You must erase a Flash cell before it can be programmed; most Flash memory implementations require an entire block to be erased at one time

References:
- *MSP430 Flash Memory Characteristics* by Peter Forstner (SLAA334A.pdf)
- EE 216: Principles and Models of Semiconductor Devices by Chintan Hossain at Stanford University http://www.youtube.com/watch?v=s7JLXs5es7I

How do flash devices work? In a nutshell, they use the concept of a floating gate transistor.

Usually a transistor is "off" or "on" depending upon the value applied at its control gate. Apply power to the gate and it causes electrons to flow from the source to the drain; take the power away from the gate and the electricity stops.

Flash memories use floating gates that are "sticky"; that is, they can "remember" their value. By submerging the floating gate in a sea of dielectric, its charge value takes a very long time (hundreds of years+) to leak away.

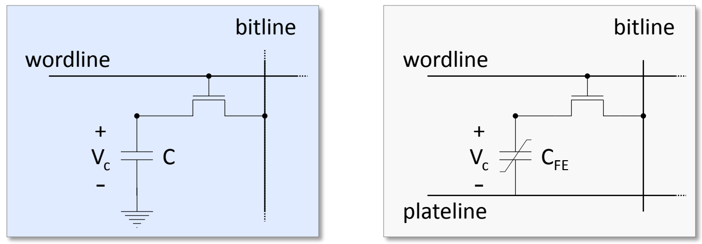
But, if it takes a long time to lose their value, how do you program a new value into them? You must use a very high voltage – somewhere around 14 Volts – to program a new value into them. Since most MCU's run off of 5 Volts (or less), single-chip MCU manufacturer's embed charge pumps into them to generate the voltages required.

Even with the need for this extra high-voltage circuitry, flash memories have served the industry quite well. Many of the MSP430 devices, such as the MSP430F5529 utilize flash non-volatile memory.

FRAM Memory

As we stated earlier, while FRAM technology has been used for stand-alone memory chips, it's relatively new to microcontrollers. Its high endurance and low-power operation make it idea for many applications.

How FRAM Memory Works



DRAM cell **FRAM cell**

- Ferroelectric RAM (FRAM) is similar to Dynamic RAM (DRAM) – except that FRAM uses ferroelectric capacitors – as opposed to traditional (dielectric) capacitors
- Applying a field to the ferroelectric capacitor flips its state; the amount of energy required indicates the previous value
- Similar to DRAMs, reads are destructive; although FRAM implementations immediately write back the original value
- FRAM (aka Fe-RAM) does not contain element "Fe" (Iron) – rather the name is based on the ferroelectric hysteresis loop waveform, which is key to its operation
- Reads and writes only require about 1.5V – thus, no charge pump required

References:
- *FRAM for Dummies* - http://www.edn.com/design/systems-design/4394387/FRAM-MCUs-For-Dummies--Part-1
- http://www.radio-electronics.com/info/data/semicond/memory/fram-ferroelectric-random-access-memory-technology-operation-theory.php

FRAM – Ferroelectric Random Access Memory – is much like other types of RAM memory. You can read and write this memory just as you might an SRAM found in most processors. This said, its closest cousin might be the DRAM (Dynamic RAM) cell.

DRAM's use capacitance to hold information. As most electronics savvy folks know, applying a field across a capacitor causes it to store a charge. The presence (or not) of this charge can be sensed, which is how we read the DRAM cell. While DRAM is useful as a read/write memory, it must remain powered-on and refreshed in order to retain their contents; therefore, they cannot be used for non-volatile memory. (Instead, they might be thought of as the best example of 'volatile' memory.)

FRAM's utilize the same basic concept as DRAM's but utilize ferroelectric capacitance (C_{fe}) to retain their information. The ferroelectric crystal contains a dipole whose atom can be moved into an up or down state based upon the application of a field. The atoms position can then be sensed, allowing us to read its value. Thankfully, the processes of setting the dipole's state can be done with as little as 1.5 Volts … making FRAM a very low-power technology.

Like a DRAM, the read is a destructive process, though FRAM memory implementations include hardware to immediately write-back the value without any intervention needed from the user. Unlike DRAM, though, the C_{fe} doesn't lose its value if the power is removed. This makes it ideal for use as a non-volatile memory.

One of the most commonly asked questions is whether FRAM's contain the element Fe (Lead). The answer is "No". (Sorry, you can't hang FRAM chips on your refrigerator like magnets.) Rather, the name comes from the ferroelectric hysteresis cycle  that maps its value.

Comparing FRAM and Flash

The table below compares FRAM and Flash memories – as well as SRAM and EEPROM (which is another popular NVM technology).

Comparison of Non-Volatile Memory

	FRAM	SRAM	Flash	EEPROM
Non-Volatile Retains data without power	**Yes**	No	**Yes**	Yes
Avg Active Power (µA/MHz)	**100**	< 60	230	50,000+
Write Power for 12KB/s	**9 µA**	N/A	2200 µA	N/A
Write Speeds (13KB)	**10 ms**	< 10 ms	1 sec	2 secs
Write Endurance	**10^{15}**	Unlimited	10^{5}	10^{5}
Bit-wise Programmable	**Yes**	Yes	No	No
Data Erase Required	**No**	No	Segment	Page
Unified: Code and Data	**Yes**	No	No	No
Read Speeds	8 MHz	**up to 25MHz** (on some devices)		N/A

FRAM, like SRAM, lets you read and write memory without any special code or procedure.

Alternatively, Flash and EEPROM require a multi-step process to update their contents. Even worse, these technologies require that you erase an entire block before you can write a single byte into it. These two items preclude their use for volatile memory operations – such as variables, stack, heap, etc.

SRAM can store code or data; in fact, we can even execute code from SRAM. Unfortunately, it loses its contents when power is removed. Sure, it doesn't need much power, but it's just not well suited for non-volatile applications. (*Note: To use SRAM for executing code, you must first copy the code into the SRAM memory before executing it.*)

FRAM, on the other hand, can be used for both volatile and non-volatile applications. It's often called a "unified" memory since it can be used to store both code and data. Throw in its low-power nature as well as its nearly unlimited write endurance and you've got an exceptional memory technology. (It seems every year the FRAM write endurance specs get bumped up another notch; last year it was 10^{14}, this year 10^{15} – it takes a lot of time to run these endurance tests.)

Today, the FRAM technology limits us with its read frequency. It significantly out speeds Flash for write operations, but it falls behind in reads. Obviously, this means it is not well suited for high-end multi-GHz application processors; but, it fits nicely into low-power applications, which makes it ideal for the MSP430 family.

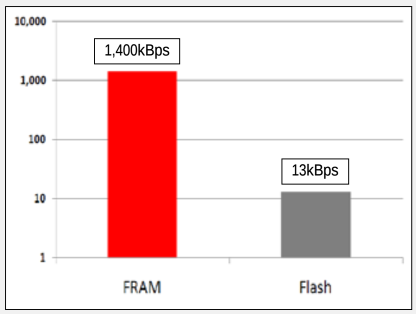
FRAM Benefits and Applications

The next two pages show five slides from the FRAM marketing presentations. They do a good job demonstrating the advantages of FRAM. We offer them for your perusal. Though we won't address them individually, these slides confirm the information found in the previous comparison table.

FRAM = Ultra-Fast Writes

- Case Example: MSP430FR5739 vs. MSP430F2274
- Both devices use System clock = 8MHz
- Maximum Speed FRAM = 1.4MBps [100x faster]
- Maximum Speed Flash = 13kBps

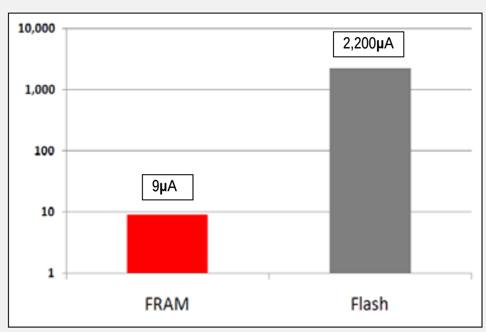
Max. Throughput:



FRAM = Low Active Write Duty Cycle

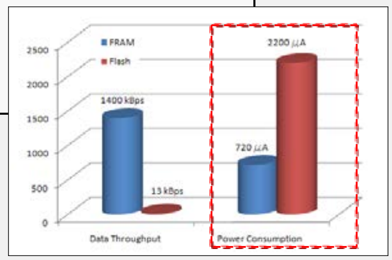
- Use Case Example: MSP430FR5739 vs. MSP430F2274
- Both devices write to NV memory @ 13kBps
- FRAM remains in standby for 99% of the time
- Power savings: >200x of flash

Consumption @ 13kBps:



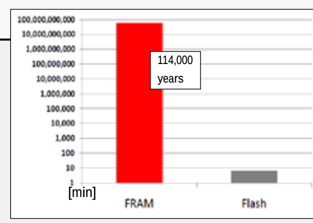
FRAM = Ultra-Low Power

- Use Case Example: MSP430FR5739 vs. MSP430F2274
- Average power FRAM = 720µA @ 1400kBps
- Average power Flash = 2200µA @ 13kBps
- 100 times faster using half the power
- Enables more unique energy sources
- FRAM = Non-blocking writes
  - CPU is not held
  - Interrupts allowed



FRAM = High Endurance

- Use Case Example: MSP430FR5739 vs. MSP430F2274
- FRAM Endurance >= ~~100 Trillion [10^15]~~ UNLIMITED
- Flash Endurance < 100,000 [10^5]
- Comparison: write to a 512 byte memory block @ a speed of 12kBps
  - Flash = 6 minutes
  - FRAM = 100+ years



FRAM Benefits --- Example App's

Non-Volatile (black)
- Retains data without power

Fast Write / Update (red)
- RAM like performance.
- Up to ~ 50ns/byte access times today (> 1000x faster than Flash/EEPROM)

Low Power (blue)
- FRAM only needs 1.5V for writes versus Flash/EEPROM >10-14V
- No charge pump needed for FRAM!

High Write Endurance (green)
- 100 Trillion read/write cycles

Superior Data Reliability (yellow)
- 'Write Guarantee' in case of power loss

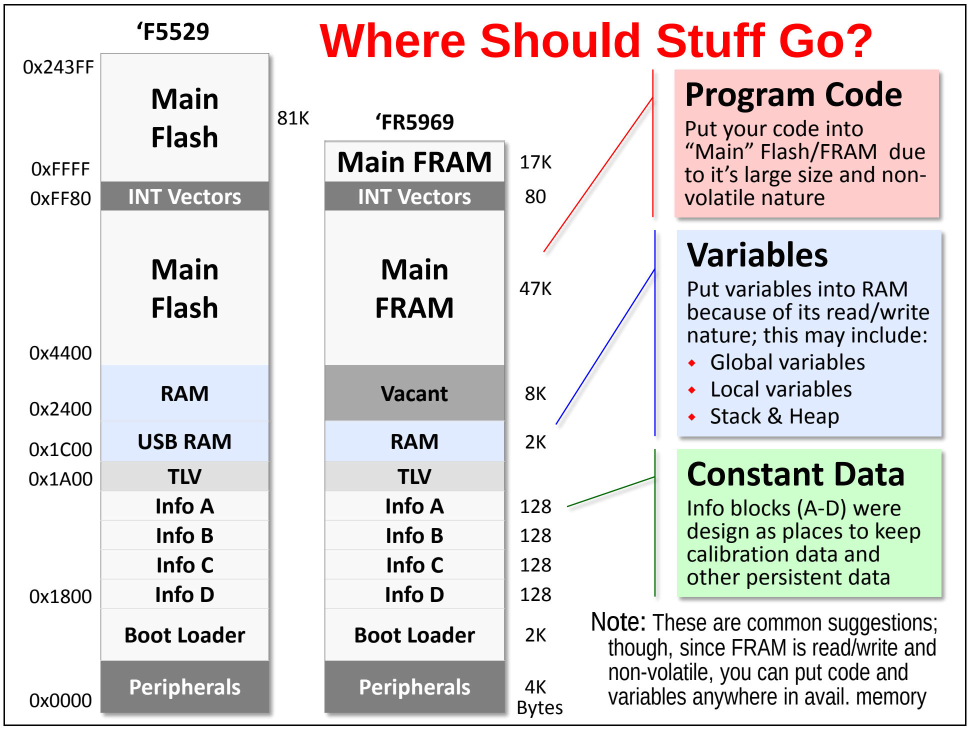
Example App's:

- Data logging & remote sensor applications (black, blue, green, yellow)
- Digital Rights Management (DRM) (black, green, yellow)
- Low Power Applications (e.g. Mobile & Consumer products) (blue, yellow)
- Energy Harvesting (especially wireless) (red, blue, yellow)
- Battery-Backed SRAM Replacement (black, red, blue, green, yellow)

Memory Maps & Linking

Memory Maps

As you might already know, memory-maps provide a tabular description for how memory addresses are used. In our microcontrollers, they indicate how the chip designers have allocated the memory addresses to Non-volatile memory (Flash or FRAM), volatile memory (RAM) and a variety of other uses, such as peripheral control registers, boot-loaders, and such.



Unlike the "old" days, we don't worry about the specific addresses used by each item anymore. The need for this has been deprecated by the use of symbolic, high-level languages. For example, rather than remembering the specific hex address used for a serial port register, we can use the convenient symbol name defined for us in the libraries TI provides. Using DriverLib throughout this workshop has shown us just how powerful – and easy – this can be.

Even though we might not be required to look up (and memorize) specific addresses nowadays, the memory map is still enormously important. It shows us *how much* and of *what type* of memory we have available in our system.

In fact, it's this awareness of memory, and how to use it, that largely differentiates an Embedded Processor programmer from a standard application programmer. For example, when first writing programs in school, we usually didn't care how much – or what types – of memory was available. In other words, memory was (for me at least) a vaguely unlimited resource. (To infinity and beyond…)

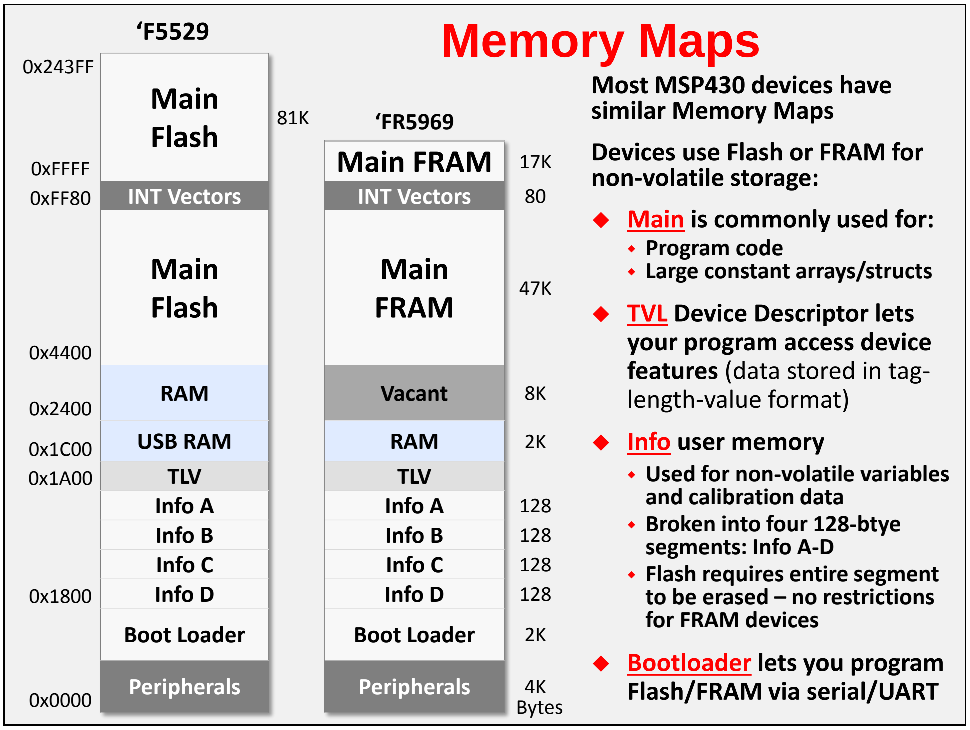
In real-world embedded systems, though, memory is an expensive, and limited, critical resource. If you pick a device that has more than enough memory, your boss will probably accuse you of overspending. Also, as we've learned throughout this chapter, not all memory is equal – you don't want to put your variables into Flash … or your program code into RAM. (At least not at power-up.)

Bottom Line: We must think about what types of memory we have; how much we have of each type; and how we should allocate our use of each.

How is NVM Used?

The previous slide roughly outlined where we should store various types of information.

The following slide provides a brief outline of how non-volatile memory (Flash/FRAM) is used in two example MSP430 devices. As you can see, in both devices, the NVM is broken into three areas: Main, Info, and Bootloader.



Thus far, this part of the chapter has discussed the memory-map. This provides us with a picture of what memory is available for our application. At this point, we can state that:

"We want our *program* to be placed into *'Main'* memory".

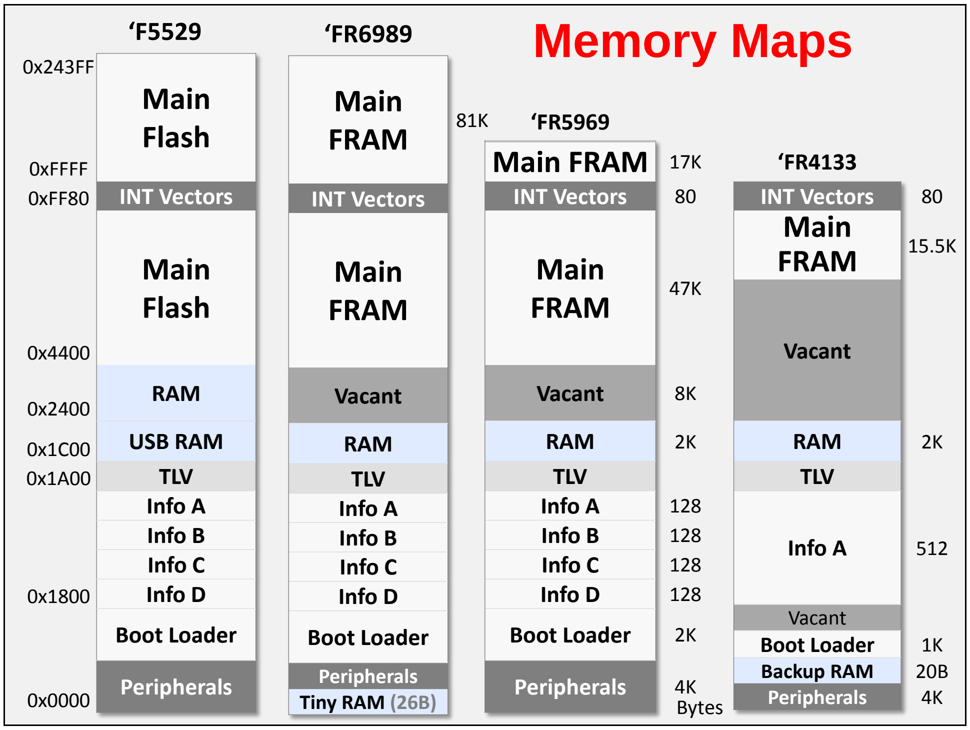
The next two topics help us understand *how* we get the right information to the right place.

- ***Sections*** describes how our program is broken-up (by the build tools) into different pieces.
- ***Linking*** shows us how to make those pieces (i.e. sections) end up in the parts of memory where we want them to go.

Notes

Comparing Device Memory Maps

Here's a quick comparison between the F5529, FR4133 and FR5969 memory maps.



The F5529 has the most RAM, but the FR6989 (and FR5989) now provide as much non-volatile memory using FRAM. The FR4133 has the least amount of FRAM, but this allows it to be used in lower-cost applications.

Sections

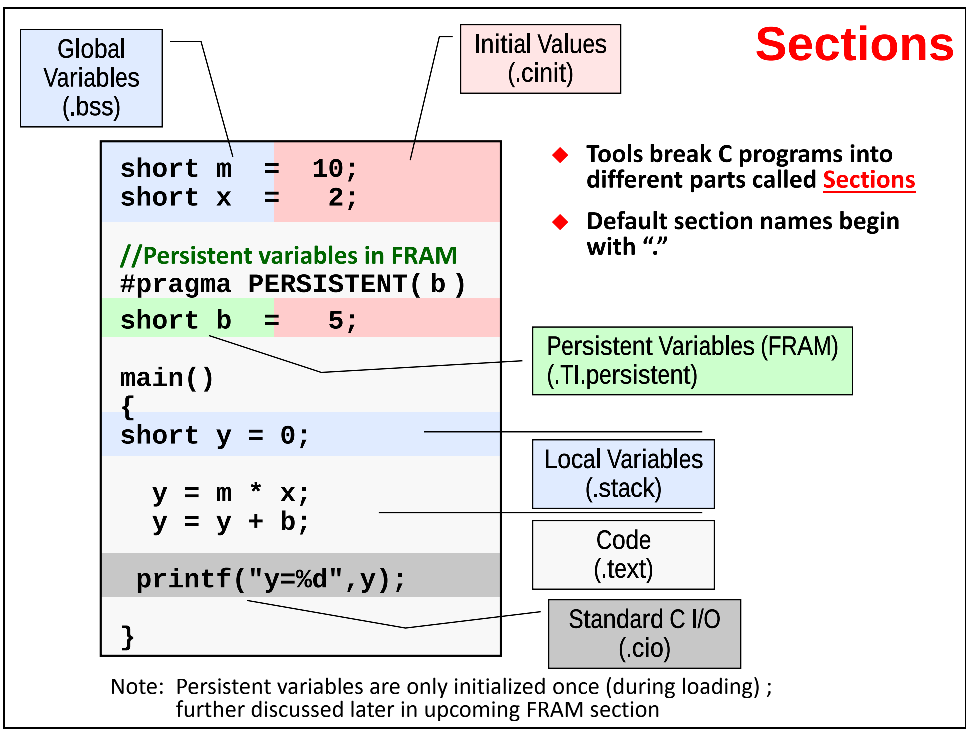
From a high-level we've already learned that there are different types of memory – for example, non-volatile (ROM-like) and volatile (RAM-like) memories.

In a similar fashion, the compiler breaks our program down into different ***Sections***. Function-by-function, file-by-file, the code generation tools will group together similar information into different sections.

Let's take the first two *Sections* shown at the top of the following slide:

- Global Variables
- Initial Values

All C programmers should recognize these two items – maybe not their names, but at least their functionality. This is one of the first things we're taught when starting to learn the C language. But, let's think about them from an embedded system point-of-view. What type of memory does each need to go into?



You may have realized that both of these Sections need to be placed into different types of memory. While ***Global Variables*** need to go into a RAM-like memory (so that we can read/write their values), the ***Initial Values*** need to be stored in a ROM-like (non-volatile) memory so that they'll always exist (even after a power-cycle).

The compiler team has assigned common, pre-defined names for these two *Sections*:

- **.bss** = Global Variables
- **.cinit** = Initial Values

By the way, the compiler's initialization routine copies the initial values into their respective global variables – as well as setting up the stack and heap – before calling main(). (If you're interested, you can find the compiler's initialization source code (rts.src) in the Run-Time Support library.)

In our simple program example we demonstrated five different Sections: Global Variables (.bss), Initial Values (.cinit), Code (.text), Stack (.stack), and Standard I/O data buffers (.cio). These represent about half of the various types of Sections the compiler may create.

Here's a table showing most of the compiler's Section types. Notice that the top 5 are intended for non-volatile memory, whereas the bottom ones should be placed in volatile – also known as uninitialized – memory.

Common Sections Created by TI Compiler

Section Name	Description	Memory Type
.text	Code	Non-Volatile
.data	Global and static non-const variables that are explicitly initialized	Non-Volatile
.cinit	Initial values for global/static vars	Non-Volatile
.TI.persistent	Initialized var's declared with PERSISTENT pragma	FRAM
.TI.noinit	Non-initialized var's declared with NOINIT pragma	Uninitialized
.bss	Global and static variables	Uninitialized
.stack	Stack (local variables)	Uninitialized
.sysmem	Memory for malloc fcns (heap)	Uninitialized
.cio	Buffers for stdio functions	Uninitialized

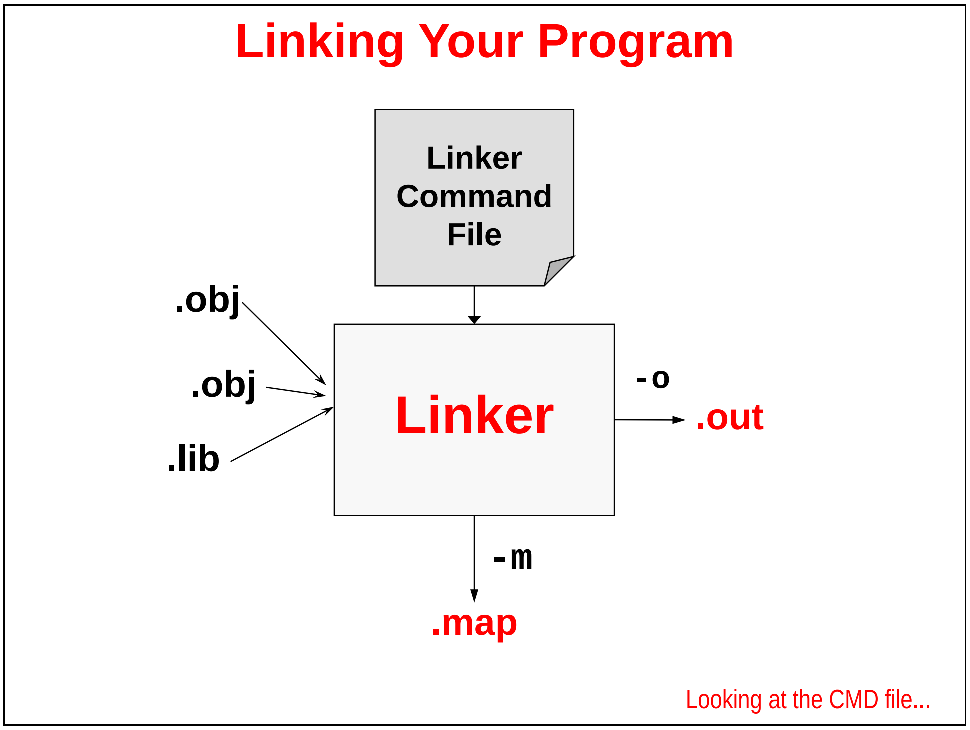
For more details, see *MSP430 Optimizing C/C++ Compiler User's Guide* (pg 69 - slau132i.pdf)

Please note, though, that not all of your programs will necessarily contain all of these Sections. For example, if you do not use Standard I/O in your programs, the compiler won't create a .cio section, as it's not needed.

For a complete list of Sections, please refer to the *MSP430 Compiler User's Guide*.

Linking

Linking is the process of bringing together all of your programs object files and assigning addresses to everything that requires memory.



The inputs to the Linker include the object files created from each of your program source files – whether you wrote the code in C, assembly, or any other language. The object files also include any binary object libraries that you've specified in your code.

Note: By default, the compiler always includes the Run-Time Support (RTS) library since it provides the compiler's initialization routine, along with a variety of other common support functions – such as standard I/O, math, trig, etc.

From these object files the Linker will create an executable binary file (.out). It also creates a Map (.map) file that provides you with a report describing what Sections it found, where it put those Sections, and where every global variable was allocated in memory.

Linker Command File

The other "optional" input to the Linker is the *Linker Command File* (.cmd). We say "optional" because, in reality, it is not optional. Sure, the linker has default settings that will allow it to build a binary file without any user direction – but these defaults rarely work for real-world systems. Realistically, you must use a linker command file.

We show a simple example of a linker command file on the next page…

Every linker command file is composed of three parts:

1. Input files and linker options: This is not shown below since it is rarely used when the code-generation tools are called from an IDE (like CCS).
2. MEMORY: This part of the .cmd file tells the Linker what memory it can allocate.
3. SECTIONS: This part lets us tell the compiler how – and where – we want each of our Sections to be allocated.

Simple Linker Command File

```
MEMORY
{  RAM:     origin = 0x2400, length = 0x2000
   INFOA:   origin = 0x1980, length = 0x0080
   INFOB:   origin = 0x1900, length = 0x0080
   INFOC:   origin = 0x1880, length = 0x0080
   INFOD:   origin = 0x1800, length = 0x0080
   FLASH:   origin = 0x4400, length = 0xBB80
   FLASH2:  origin = 0x10000,length = 0x14400
}
SECTIONS
{
   .bss         : {} > RAM
   .data        : {} > RAM
   .sysmem      : {} > RAM
   .stack       : {} > RAM

   .text        : {}>> FLASH2 | FLASH
   .text:_isr   : {} > FLASH
   .cinit       : {} > FLASH  | FLASH2
   .const       : {} > FLASH  | FLASH2
   .cio         : {} > RAM
}
```

Operators:

{ file.obj } specifies files to include in output section

>> indicates output section can be split (if necessary)

| used as 'or' symbol; allows list of memory segments as targets for output section

As you can see, each line in MEMORY{} defines a memory segment's location and size. It is common to find each of the different areas of our memory-map described here. The MEMORY specifications can be broken up or combined as needed for your system, though this isn't very common.

In the SECTIONS{} portion of the .cmd file we see each of our Sections being directed into the appropriate memory segment. In many systems, it's really as simple as shown above. Of course, there are more complicated systems that require a "finer" control of memory placement. To this end, the Linker is incredibly flexible.

Unfortunately, digging into all the Linker's details is outside the scope of this workshop. We'll see an advanced example later in this chapter, but we refer you to the *MSP430 Assembly Language Tools User's Guide* (slau131j.pdf) for all the gory details.

Hint: The MSP430 team has created a default linker command file for each specific MSP430 device. This is very handy!

In fact, you may never have to create (or even modify) a linker command file.

Even if you have to do so, their default file provides you a great starting point. This is surely better than the days where everyone had to create their own from scratch.

Custom Sections

One last topic that spans Sections and Linking – you can create custom sections within your C code. This gives you the advantage of being able to place any specific code or data item into any location in memory.

Create Custom Sections

- Create custom code section using a pragma:

```
#pragma CODE_SECTION(dotp, "critical");
int dotp(a, x)
```

... or create a sub-section:

```
#pragma CODE_SECTION(ctrl, ".text:_ctrl");
int ctrl(z)
```

- There's a data section pragma, as well:

```
#pragma DATA_SECTION (x, "InfoC_Vars");
#pragma DATA_SECTION (y, "InfoC_Vars");
int x[32];
short y;
```

** Also, look for the SET_CODE_SECTION and SET_DATA_SECTION pragmas in the compiler user's guide*

The #pragma statements shown above let you create CODE or DATA sections. For code sections you need to specify the function and the name of the "section". You are allowed to put as many functions into one section as you would like.

Similarly, you can put as many variables into a data section as you want. We've provided an example of this above.

Finally, the Linker allows the concept of sub-sections. This allows you to specify a custom section for a function (or data variable) – but have it be a part of a larger section, too. Sub-sections give you a choice for how they will be linked. If you call-out a subsection in the SECTIONS{} statement of your linker command file, you can specify exactly where and how you want it to be placed into memory. On the other hand, if you don't specify it in your linker command file, it will be combined with the 'parent' section and placed accordingly. In the example shown above, the `_ctrl` sub-section would be allocated with the rest of `.text` you specifically listed it in your linker command file.

In the example linker command file below, we didn't specify the `_ctrl` sub-section, so it will end up being allocated with the rest of `.text`. Alternatively, notice that another sub-section (`.text:_isr`) was specifically called out and will be linked independently from the rest of .text.

CMD File with Custom Sections

```
MEMORY
{  RAM:     origin = 0x2400,  length = 0x
   INFOA:   origin = 0x1980,  length = 0x
   INFOB:   origin = 0x1900,  length = 0x
   INFOC:   origin = 0x1880,  length = 0x
   INFOD:   origin = 0x1800,  length = 0x
   FLASH:   origin = 0x4400,  length = 0x
   FLASH2:  origin = 0x10000, length = 0x
}
SECTIONS
{   critcal     : {} > 0x2400
   .bss         : {} > RAM
   .data        : {} > RAM
   .sysmem      : {} > RAM
   .stack       : {} > RAM
    InfoC_Vars  : {} > INFOC type=NOINIT

   .text        : {}>> FLASH2 | FLASH
   .text:_isr   : {} > FLASH
   .cinit       : {} > FLASH | FLASH2
   .const       : {} > FLASH | FLASH2
   .cio         : {} > RAM  }
```

- Custom sections allow you to place code:
  - At specific locations
  - In a specific order
- NOINIT *type* tells system init code to ignore initialization for that output section
- Sub-sections allow you to specify the sub-sect's location
  … or, if not specified, it's linked along with the parent section (ie ".text")
- This is a contrived example to show the mechanism; we'll see 'real' examples later in the chapter

---

Note: Let us caution you, though, that you should use this judiciously. We recommend that you use Custom Sections (and/or customize the linker command file) only when "something" has to go in a very specific location. In fact, though, we will show you an example of this later in this chapter.

Sidebar – Using the "wrong" type of memory

As stated earlier, even though this goes against common style, you can place:

- "Code into RAM"
- "Variables into Flash"

While this is not a problem for the linker (because it only assigns addresses), it is tricky from a hardware point-of-view. Making these options work correctly requires extra code.

For example, before you run code from internal RAM, must first copy it from its non-volatile memory location into the RAM. This could be done with either the CPU or the DMA.

Updating variables stored in Flash requires a series of steps – as does any programming of Flash memory. We provide an example of this in the upcoming lab exercises.

Using Flash

The Flash Memory Controller provides access to the Flash non-volatile memory. Read accesses occur normally, just as you might read from a RAM memory. Writes, on the other hand, require a correct procedure to be followed. Writing directly to Flash causes an interrupt (if enabled) … and doesn't modify the Flash memory.

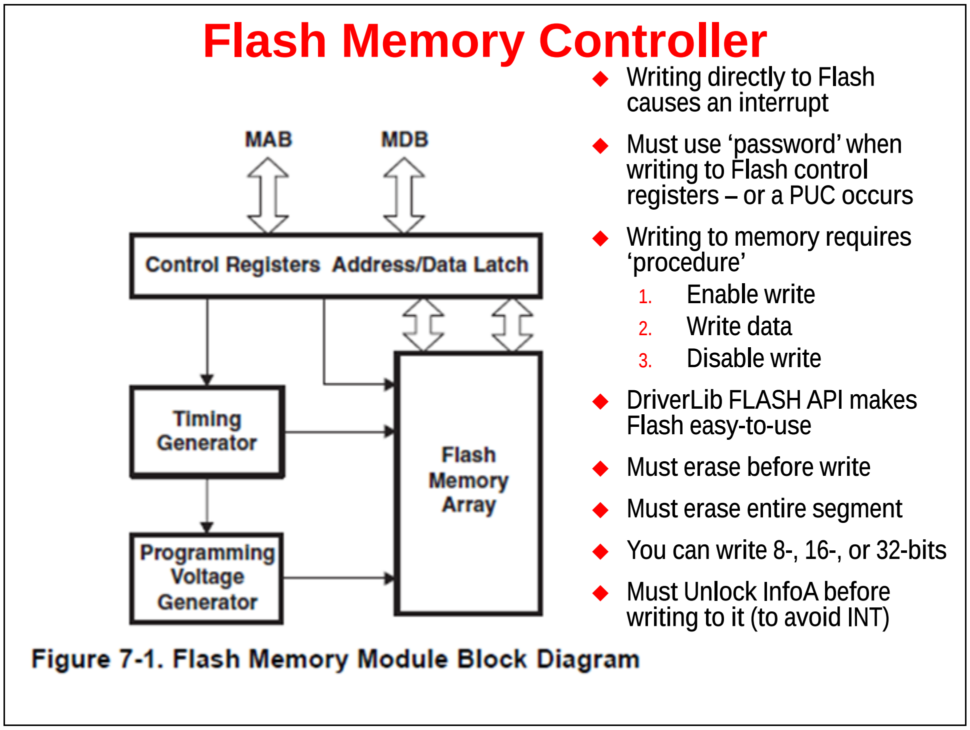


Figure 7-1. Flash Memory Module Block Diagram

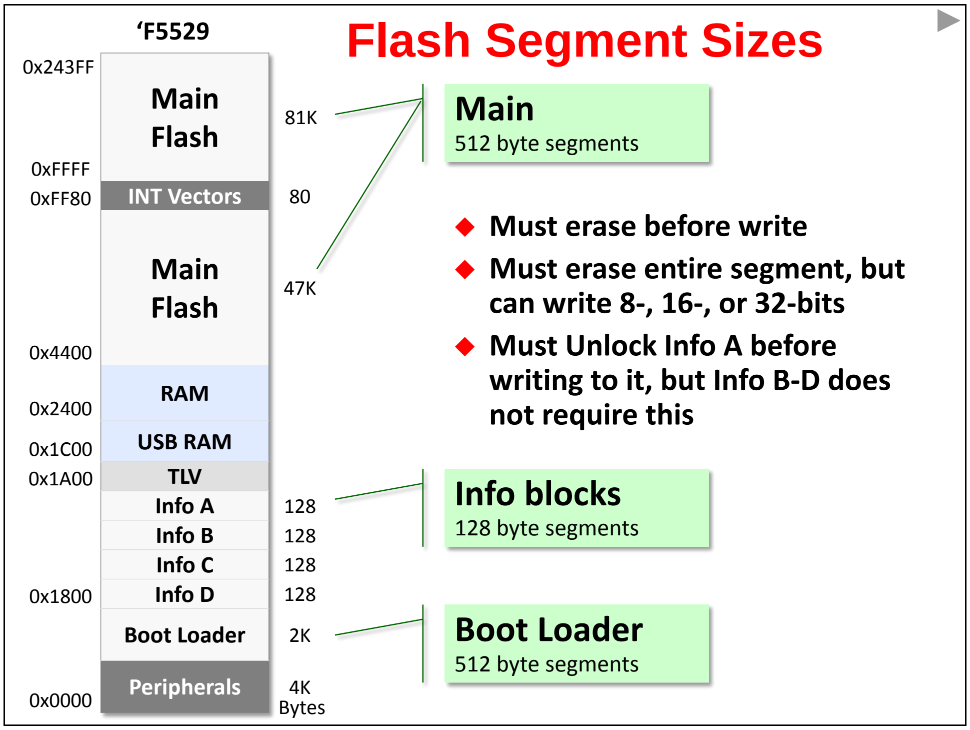
The Flash write procedure includes:

- Disable the Watchdog Timer, if it is running.
- Clear the Flash LOCK bit (using the appropriate Flash Control Register password)
- Enable Flash write mode by setting WRT=1 (again, using the correct password)
- Writing to the memory as needed – checking the BUSY bit to make sure each write is complete before starting another write.
- Disable write mode and re-LOCK the Flash (yet again, using the correct register password).

Note: Due to the complexity of these write operations, we recommend that you utilize the FLASH DriverLib API, which will be discussed shortly.

Before you can write to Flash memory, it must first be erased; in fact, the entire segment must first be erased. Writes, though, can be done on a byte-by-byte basis.

On the ‘F5529 you might remember that we have three areas of Flash memory: Main, Info, and Boot. The diagram below shows these along with their segment sizes.



The Info blocks are popular locations to store calibration data because of their smaller 128 byte segment size. This means less memory must be erased when needing to (re)write data. It also minimizes interference with the “Main” Flash, which is often used for program code.

Using DriverLib to Write to Flash

Notice the functions found in DriverLib's FLASH module – these let you erase, write, fill and check the status of the MSP430's Flash memory.

FLASH API

Flash erase operations are managed by:

- FlashCtl_segmentErase()
- FlashCtl_eraseCheck()
- FlashCtl_bankErase()

Flash writes are managed by:

- FlashCtl_write8()
- FlashCtl_write16()
- FlashCtl_write32()
- FlashCtl_memoryFill32()

Status is given by:

- FlashCtl_status()
- FlashCtl_eraseCheck()

Segment InfoA memory lock/unlock:

- FlashCtl_lockInfoA()
- FlashCtl_unlockInfoA()

- Writing to memory requires 'procedure'
  1. Enable write
  2. Write data
  3. Disable write
- Writing directly to Flash causes an interrupt
- Must use 'password' when writing to Flash control registers – or a PUC occurs
- DriverLib FLASH API makes Flash easy-to-use
- Must erase before write
- Must erase entire segment
- You can write 8-, 16-, or 32-bits
- Must Unlock InfoA before writing to it (to avoid INT)

The following code example uses DriverLib to perform a block erase on Info A; then write an array of data to it. Remember, Info A has an extra "lock" feature that you need to unlock beforehand, then should re-lock afterwards (this is not required for the other Info segments).

Code Example: Writing to "Info A"

```
#pragma DATA_SECTION (calibration_data_char, ".infoA")
uint8_t  calibration_data_char[16] = { 0x00,0x01,0x02,...};
uint16_t status;

// Unlock Info Segment A
FlashCtl_unlockInfoA();

do { // Erase INFOA
     FlashCtl_segmentErase( (uint8_t*)INFOA_START );
     status = FlashCtl_eraseCheck((uint8_t*)INFOA_START,128);
} while ( status == STATUS_FAIL );

// Write calibration data to INFOA
FlashCtl_write8( calibration_data_char,
              (uint8_t*)INFOA_START, 16 );

// Lock Info Segment A
FlashCtl_lockInfoA();
```

Using FRAM (and the MPU)

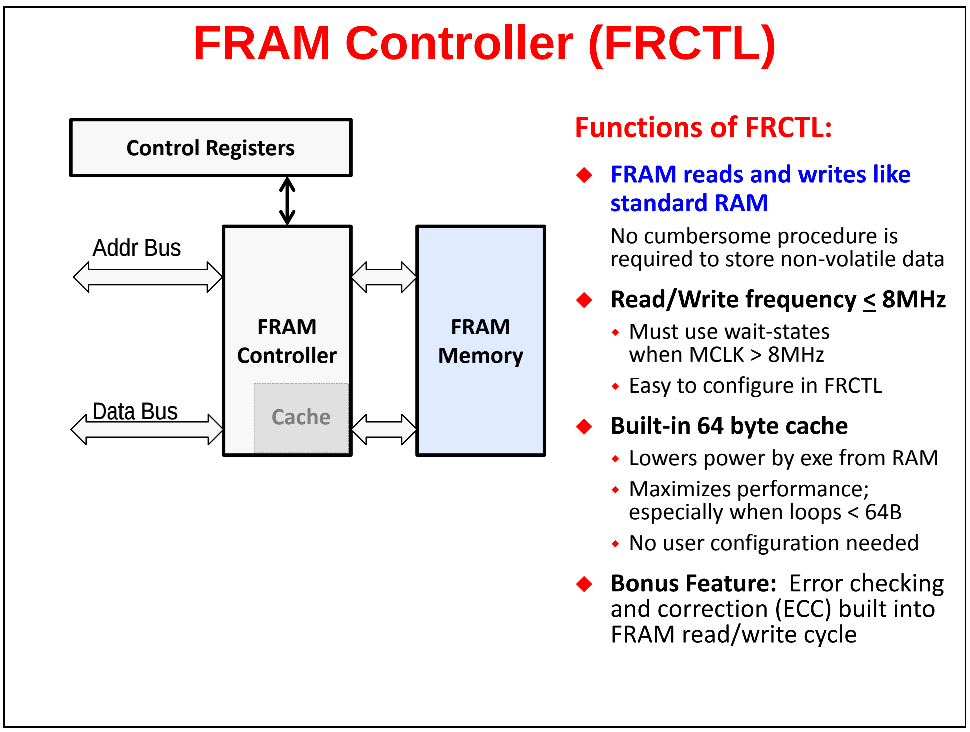
Similar to Flash, there is a controller which handles the reading and writing of FRAM. Other than the fact that both controllers require that you use a password to modify their registers, though, there is very little else that they have in common.

FRAM Controller

Unlike Flash, FRAM allows users to easily read and write to them. This leaves the FRAM Controller with only two things to do:

- Managing read/write access when the CPU is running > 8 MHz; including the use of cache to minimize sequential/repetitive program accesses.
- Implementing error correction and control (ECC) memory checking.

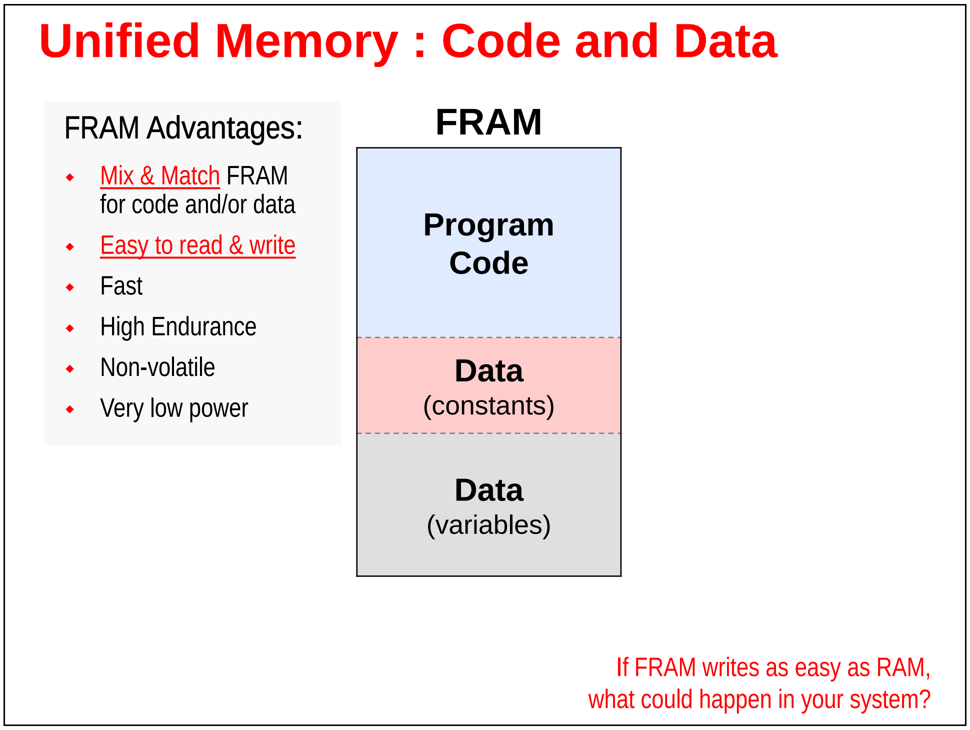
Other than needing to set the waitstate value (if the CPU is running > 8 MHz), both of these run transparent to the user.



If you care about the ECC feature, you will need to enable the associated interrupt bits so that you'll be warned in the case of a memory error/warning event.

Unified Memory

FRAM supports *unified* memory – which means that you can store both Code and Data in FRAM.



It's often common to see the FRAM contain program code, constant data (i.e. read-only data), as well as read/write (random access) data.

Can you think of what might go wrong, though, when using FRAM in this way?

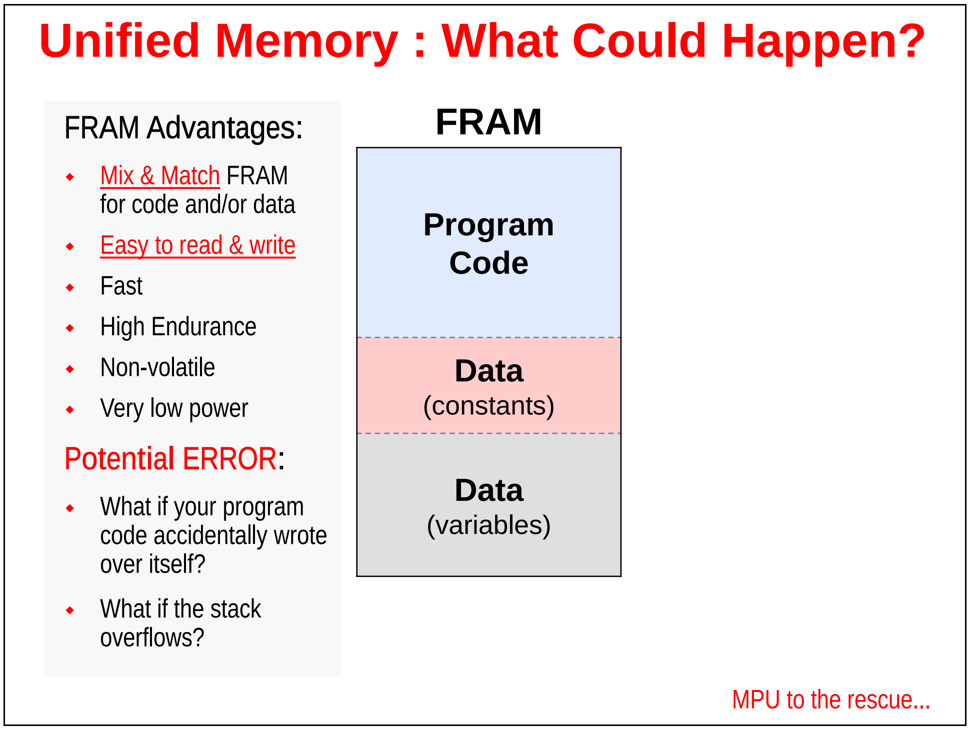
____

____

Actually, it's not a problem with the multi-use of FRAM; it's more a problem with how easy it is to write to FRAM…

What Could Happen to FRAM?

The problem, as we said, is that FRAM is so easy to write to – unlike Flash. While generally this is a good thing, what happens if your program goes rogue? For example, what happens if an error causes your program stack to overrun its "boundary"?

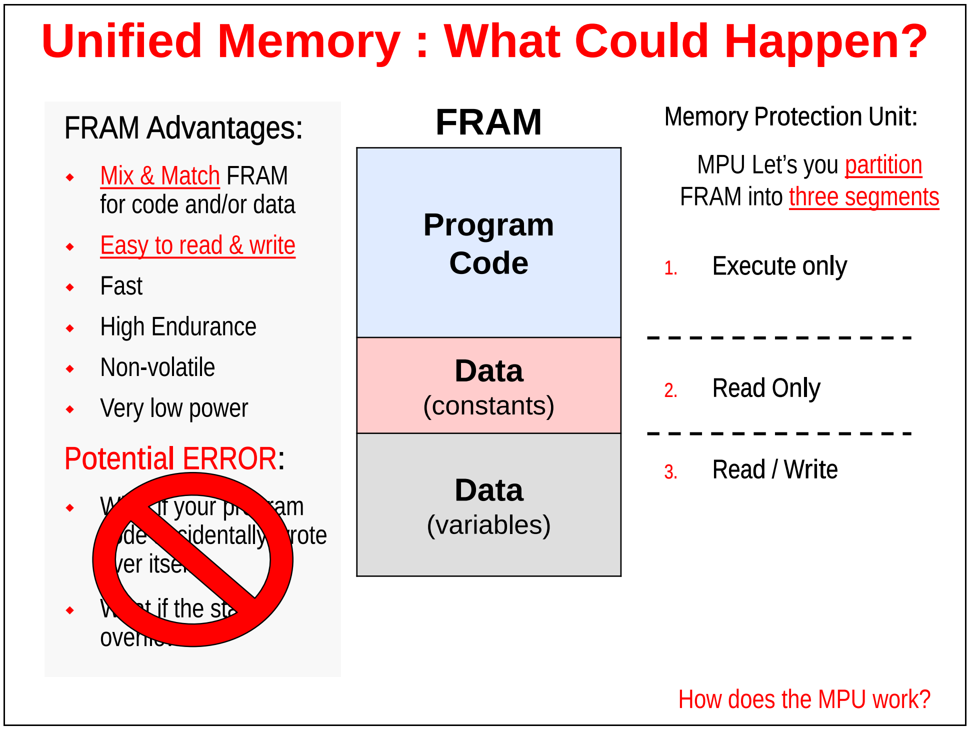


When using Flash, this problem would usually cause a system reset (PUC) since you cannot write directly to it without using the proper procedure. With FRAM, though, there isn't a technological restriction to these types of programmatic errors.

The solution chosen by TI was to include a Memory Protection Unit (MPU) in these devices.

Memory Protection Unit (MPU)

The MPU allows you to divide your FRAM into 2 or 3 segments and then individually apply access permissions to each of these segments. As shown below, our FRAM was broken into 3 segments with: one segment (our code) set to only allow code Execution; another segment only allows Read access; while the last allows read or write accesses.

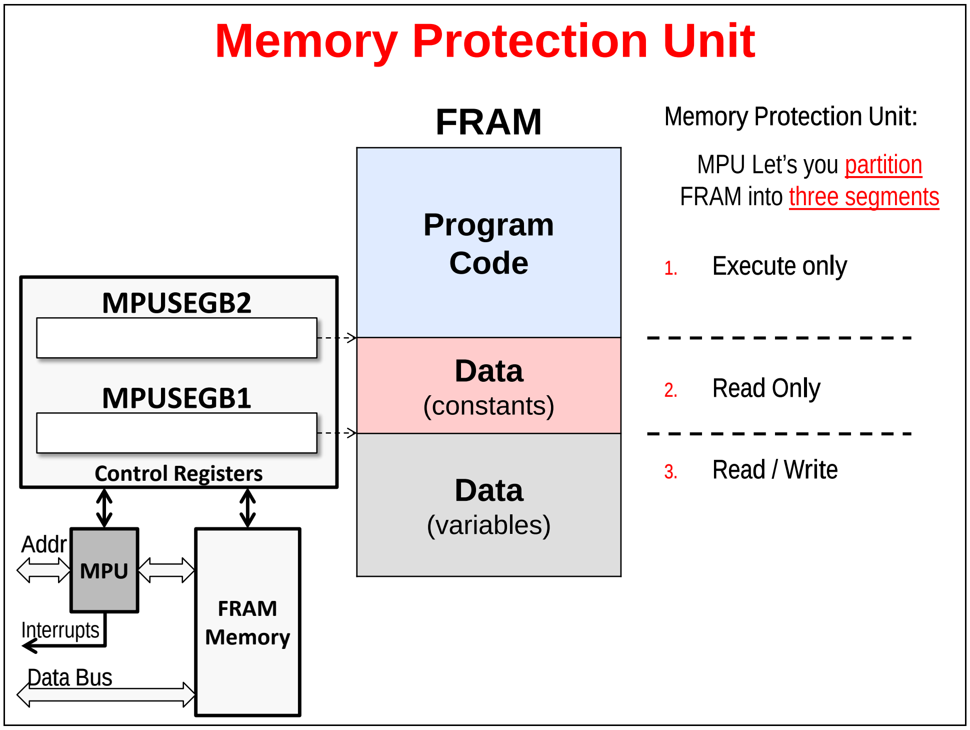


With the MPU configured and enabled in this manner, a write access to the "code" segment generates an exception. This exception either causes a reset (PUC) or a non-maskable interrupt (NMI) depending upon how you've configured the MPU.

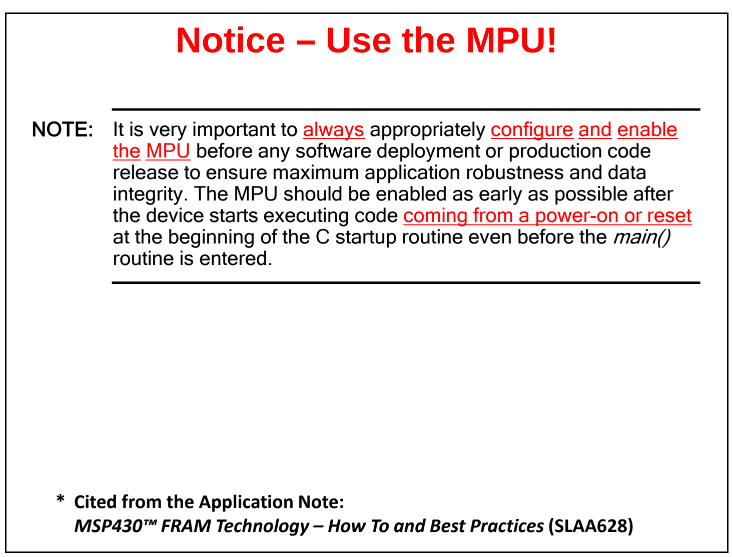
In this way, you're protected from potential errors due to errant writes to FRAM.

Using the Memory Protection Unit (MPU)

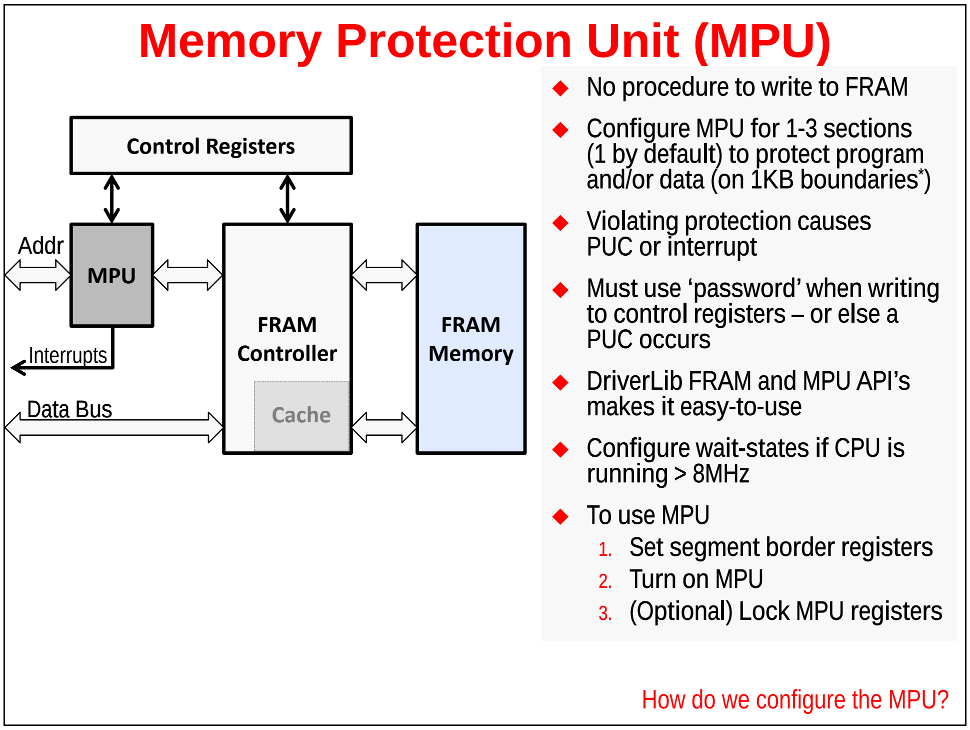
Looking at the MPU more closely, we see that two registers define the boundaries for the three segments. Writing addresses to these registers defines each segment's location and size. An upcoming example will show how we can use linker symbols to set these boundaries appropriately.



After a PUC reset, the MPU registers are set to their default state. This causes the FRAM to be configured as a single segment with all access permissions enabled (Read, Write and Execute).



As we might expect, looking at the following diagram we see that the MPU watches addresses flowing into the FRAM controller. This allows it to intercept non-approved memory accesses to FRAM.



Using the MPU requires:

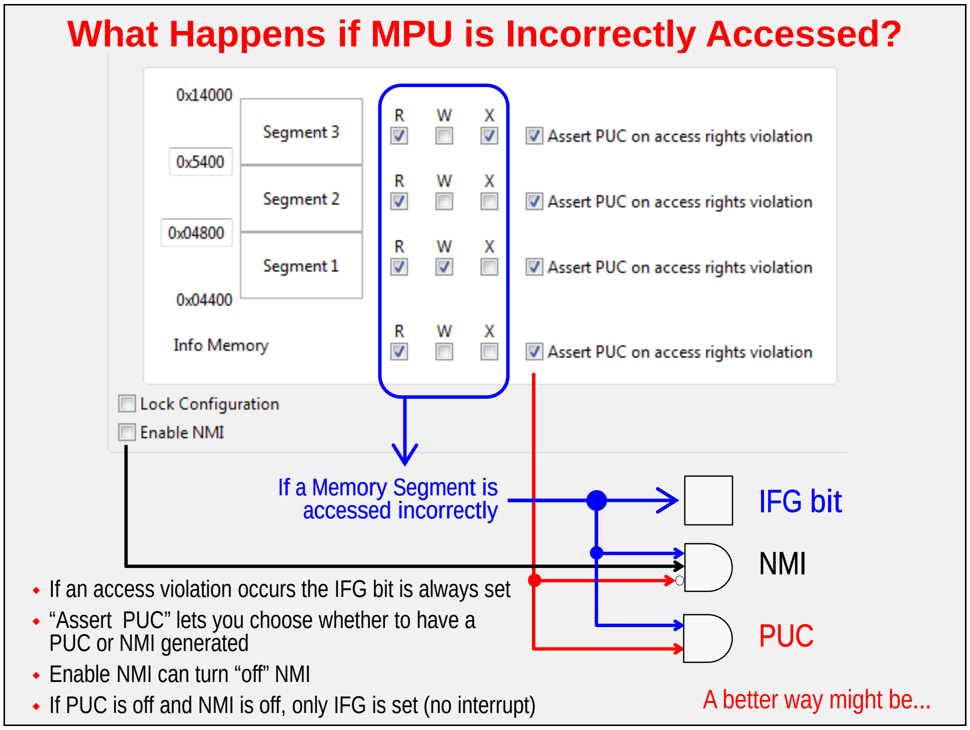
- Writing a password to the MPU registers
- Setting the address segment boundary registers
- Setting the Read/Write/Execute permissions for each segment
- Configuring the violation response – should a PUC or NMI be generated whenever a segment is incorrectly accessed
- Turn on the MPU
- Finally, you may wish to Lock the MPU to prevent any changes (until the next BOR reset)

While the procedure here might appear as long as the Flash writing procedure, remember that you only need to do this once … not every time you want to write to FRAM.

A couple of additional notes about the MPU:

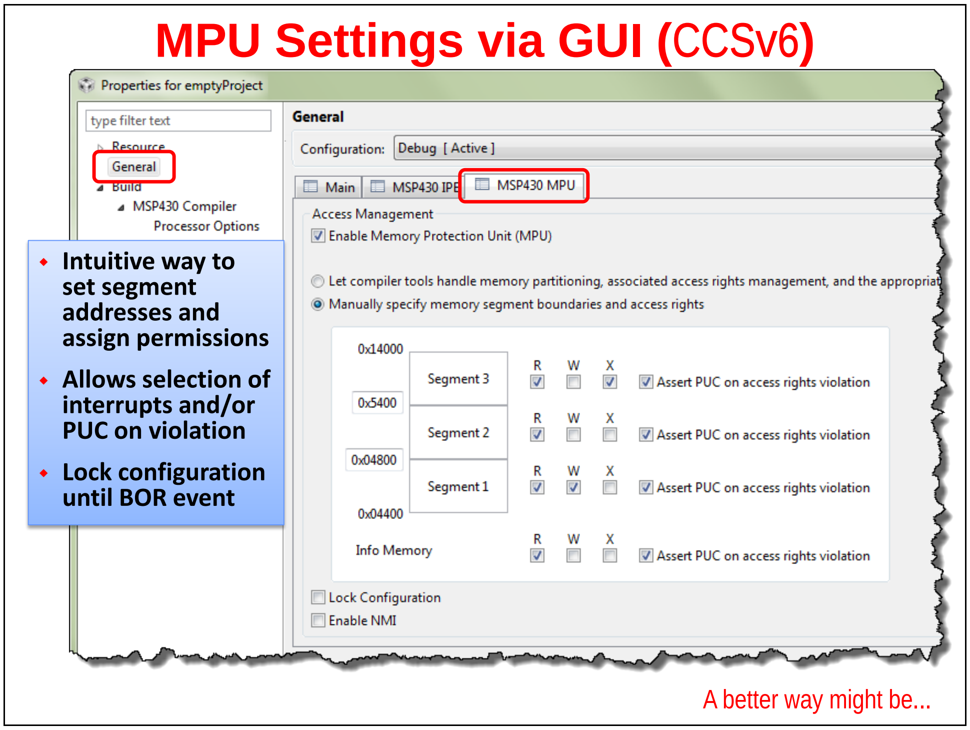
- Each segment can be configured individually for access permissions.
- You can also set access permissions for the Info blocks (as a whole).
- You can continue to change the MPU settings even after the MPU is enabled … that is, unless you lock the MPU registers, in which case a reset is required before you can access the MPU registers again.

What are the Interrupt Options?

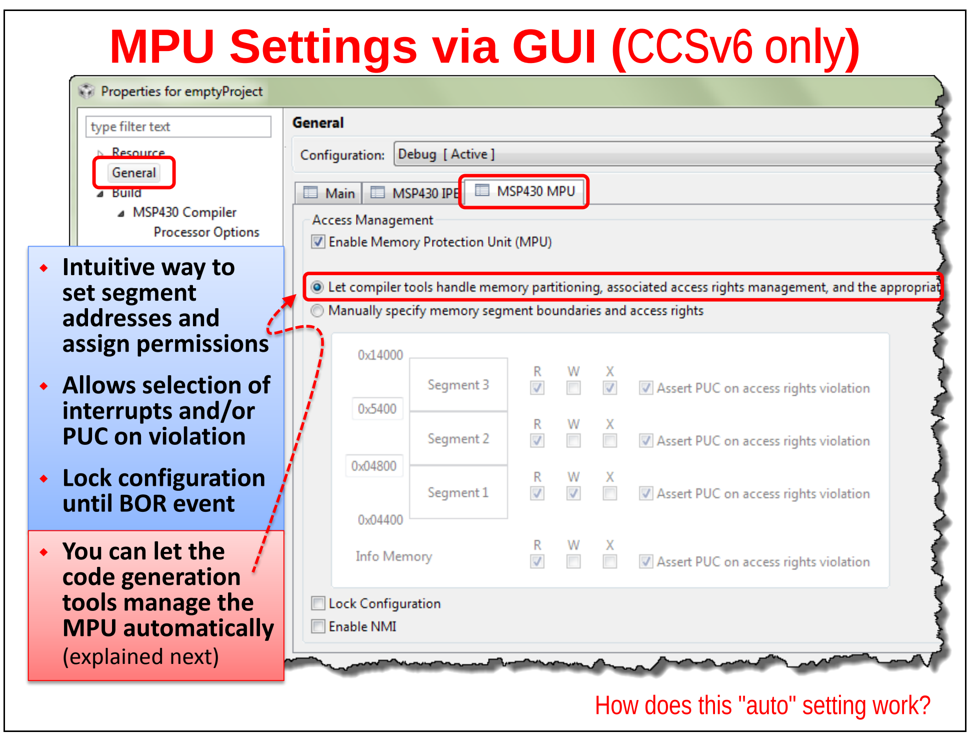


MPU Graphical User Interface

Starting with CCSv6, the MSP430 team has created a convenient GUI to simplify the process of setting up the MPU. The following screen capture shows the simple text/check boxes required to set it up.



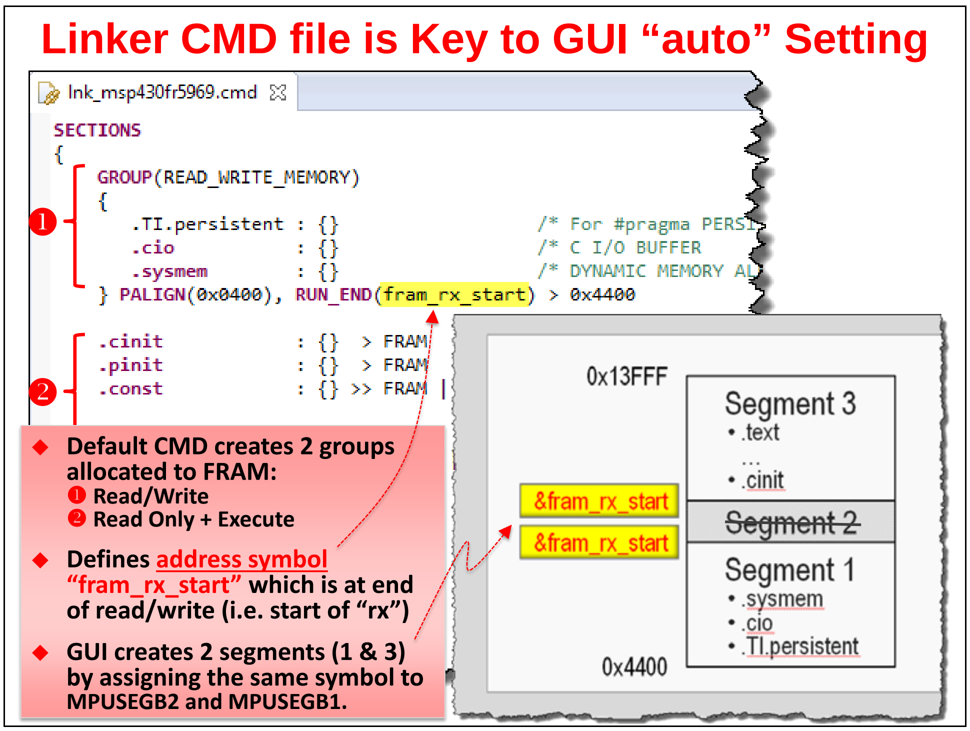
In fact, you can even elect to the let the GUI handle setting the boundary registers for you.



The key to automating the GUI is found in the Linker Command File (CMD). The following *fancy* linker syntax – found in the default linker CMD file – groups the non-volatile read/write sections created by the linker.

The GUI tool creates 2 MPU segments:

- Segment 1 contains the Read/Write input sections that require non-volatile storage … or, in the case of .cio, are large enough that they are often stored in "Main" FRAM space
- Segment 2 is not created as the starting address of Read/Execute non-volatile memory is assigned to both MPU Segment Border registers.
- Segment 3 contains the input sections for Read-Only and Execute:
  - Read Only sections such as the initial values for variables
  - Finally, the Executable sections (.i.e. .text) which contain the code



Along with creating these output sections (and linking them into FRAM), the linker syntax above also creates a symbol which defines the *end* of the Read/Write group and the *start* of each the Read/Execute output sections.:

- `fram_rx_start`

The MPU GUI uses this symbol, but you can also access it from your code by using the proper external declaration in your C file. An example of this is coming up later in this chapter.

FRAM Code Example

The FRAM and MPU DriverLib functions can be used to configure FRAM access. We'll examine a few of these functions in our code example on the next page.

DriverLib FRAM & MPU API's

FRAM writes can be managed by:

- FRAMCtl_write8()
- FRAMCtl_write16()
- FRAMCtl_write32()
- FRAMCtl_memoryFill32()

FRAM interrupts are handled by:

- FRAMCtl_enableInterrupt()
- FRAMCtl_getInterruptStatus()
- FRAMCtl_disableInterrupt()

Status is given by:

- FRAMCtl_configureWaitStateControl()
- FRAMCtl_delayPowerUpFromLPM()

The MPU initialization function is

- MPU_start()

The MPU memory segmentation and access right

- MPU_initTwoSegments()
- MPU_initThreeSegments()
- MPU_initInfoSegment()

The MPU interrupt handler functions

- MPU_enablePUCOnViolation()
- MPU_disablePUCOnViolation()
- MPU_getInterruptStatus()
- MPU_clearInterruptFlag()
- MPU_clearAllInterruptFlags()
- MPU_enableNMIevent()

The MPU lock function is

- MPU_lockMPU()

Note: Setting the MPU with DriverLib is an alternative to using the GUI tool. If you're not using CCSv6, yet, then this is absolutely your best option. But this is also a good solution for those of you who prefer to use code versus using a GUI.

Configuring the MPU using DriverLib

The following example uses the symbols created by the linker to configure the boundaries between the MPU's three segments. The *MPU_initTwoSegments()* function makes it easy to configure the segment boundaries and set the access permissions.

After configuring the segments, we tell the MPU we don't want to generate a PUC when a violation in Segment 1 occurs – instead, we'll get an NMI if a violation occurs in this segment.

Finally, we start the MPU running.

Configuring MPU in Software

For CCSv5.5 users or if you want to setup the MPU using C code:

```
extern const uint16_t fram_rx_start;

void initMPU(void)
{
    // Configure MPU as two Segments
    MPU_initTwoSegments( MPU_BASE,
      (uint16_t) &fram_rx_start >> 4,  // Bound between 1 & 3
      MPU_READ | MPU_WRITE | MPU_EXEC, // Seg 1: all access
      MPU_READ | MPU_EXEC );           // Seg 3: read & exe

    // Disable PUC on segment access violation for segment 1
    MPU_disablePUCOnViolation( MPU_BASE,
        MPU_FIRST_SEG );

    // Enable PUC on segment access violation for segment 3
    MPU_enablePUCOnViolation( MPU_BASE,
        MPU_THIRD_SEG );

    // Start MPU protection
    MPU_start( MPU_BASE );
```

Here's a second example that configures the MPU for three segments.

Configuring MPU For 3 Segments

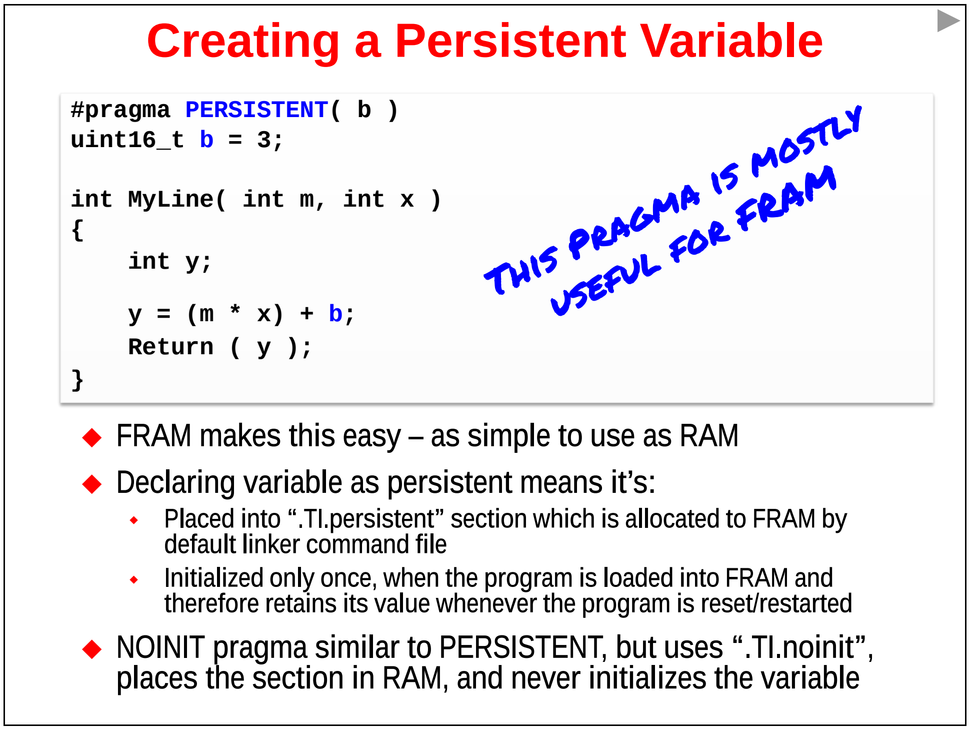
For CCSv5.5 users or if you want to setup the MPU using C code:

```
extern const uint16_t myFram_ro_start;
extern const uint16_t myFram_rx_start;

void initMPU(void)
{
    // Initialize struct for three segments configuration
    MPU_initThreeSegmentsParam myMPU;
    myMPU.seg1boundary = &fram_ro_start;            // Boundary between 1 & 2
    myMPU.seg1boundary = & fram_rx_start;           // Boundary between 2 & 3
    myMPU.seg1accmask = MPU_READ|MPU_WRITE|MPU_EXEC; //Seg 1: all access
    myMPU.seg2accmask = MPU_READ;                   // Seg 2: read only
    myMPU.seg3accmask = MPU_READ|MPU_EXEC;          // Seg 3: read & exe

    // Configure MPU Segments
    MPU_initThreeSegments( MPU_BASE,&myMPU);

    // Disable PUC on segment access violation for segment 2
    MPU_disablePUCOnViolation( MPU_BASE, MPU_SECOND_SEG );

    // Start MPU protection
    MPU_start( MPU_BASE );
```

Putting Variables into FRAM

A unique advantage to placing variables in FRAM – besides the extra storage space it provides – is that it allows variables to be non-volatile. That is, their value is retained – even upon power loss.

An easy way to direct a variable to FRAM is to make it "persist"; that is, we can use a compiler pragma to indicate that the variable's value should persist even when power is removed from the device.



Placing Variables into FRAM's INFO Memory

Info memory is simple to use on FRAM-based devices. You just need is to indicate that your variable should be placed into the Info B section using the *Custom Section* feature we discussed earlier.

Code Example: Putting Var into "Info B"

```
#pragma DATA_SECTION ( b, ".infoB" )
uint16_t b = 0;

int MyLine( int m, int x )
{
    int y;

    y = (m * x) + b;
    Return ( y );
}
```

- Place variable into INFO section using #pragma
- Default linker command file already assigns .infoB to a memory segment called INFOB: `infoB : {} > INFOB`
- By default, all EABI sections are initialized at boot; tell linker you don't want INFOB to be initialized by setting 'type':
  `infoB : {} > INFOB type=NOINIT`
- All Info blocks are defined in the CMD file in a similar fashion

This example lets us use FRAM for read/write variables, just like SRAM.

Note: If you want your INFO variable to persist – even after the processor is reset – you need to declare the output section as **NOINIT** in the linker command file. We will see an example of this in the upcoming lab exercise.

Put Any Section into FRAM

In fact, you can allocate any section to FRAM, it just takes a little editing of the using the linker command file (.cmd).

Setting FRAM Waitstates

Setting the FRAM's waitstates involves a simple call to one DriverLib function. Look in the datasheet to find the number of waitstate values you should use for your system.

Setting FRAM Wait-States

Recommended Operating Conditions

			MIN	NOM	MAX	UNIT
V_{CC}	Supply voltage range applied at all DVCC and AVCC pins.(1) (2) (3)		1.8(4)		3.6	V
C_{DVCC}	Recommended capacitor value at DVCC(5)			1		μF
f_{SYSTEM}	Processor frequency (maximum MCLK frequency)(6)	No FRAM wait states (NWAITSx=0)	0		8(7)	MHz
		With FRAM wait states (NWAITSx=1)(8)	0		16(9)	MHz
f_{ACLK}	Maximum ACLK frequency				50	kHz
f_{SMCLK}	Maximum SMCLK frequency				16(9)	MHz

```
// If you run the CPU > 8 MHz, you need to set wait-states
FRAMCtl_configureWaitStateControl( FRAM_ACCESS_TIME_CYCLES_1 );
```

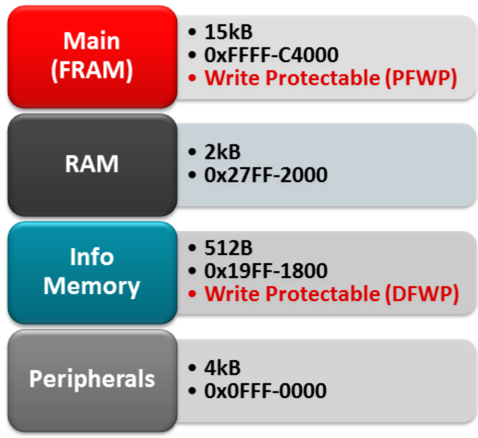
Hint: Place this in your initClocks() function – near your MCLK setup code

Notes

Memory Protection on the 'FR2xx/4xx

‘FR2xx/4xx FRAM Controller

- Like previous FRAM devices, wait-states are required when running > 8MHz
  - By default, NWAITS is set to “1” wait-state (if <= to 8MHz, you can change to “0”)
- Unlike previous FRAM devices, these parts do not have a full Memory Protection Unit (MPU) – rather, they have a two, simple protection flags
- Protection bits in SYSCFG0 register:
  - Program FRAM Write Protection for “Main” FRAM (PFWP)
  - Data FRAM Write Protection for “Info” FRAM (DFWP)
- Protection enabled by default
- Before writing to FRAM, code must clear corresponding bit
- Remember to re-enable protection after the write
- For convenience, we recommend using DriverLib!



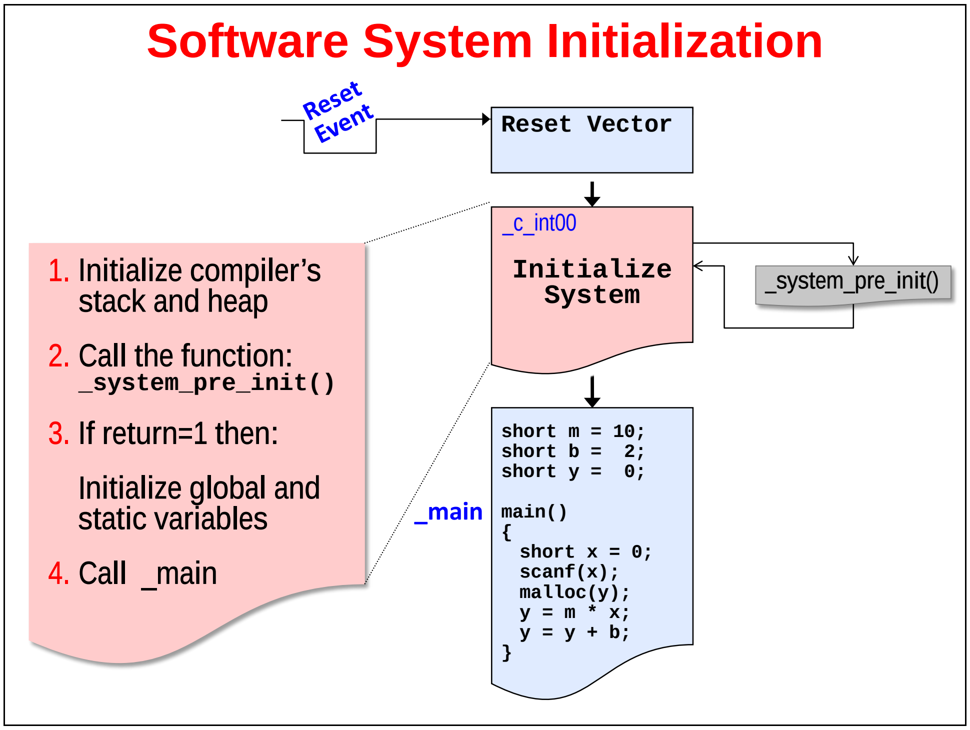
FRAMCtl_write16() Example

FRAM writes can be managed by:

- FRAMCtl_write8()
- FRAMCtl_write16()
- FRAMCtl_write32()
- FRAMCtl_memoryFill32()

```
#pragma PERSISTENT( count )       // Direct count into FRAM
uint16_t count = 0;
uint16_t temp = 0;

temp = 5;

// Write the value of temp back to the 'count'
FRAMCtl_write16(
                 &temp,            // 'from' address of data
                 &count,           // 'to' address of data
                 1                 // How many elements to
);
```

System Init Functions



For more information on "reset events", please refer to Chapter 4.

Example _system_pre_init()

```
int _system_pre_init(void)
{
    // Stop watchdog timer
    WDT_A_hold( WDT_A_BASE );

    // Configure and start MPU
    initMPU( );

    // Returning "1" tells compiler to complete variable
    // initialization; alternatively, "0" says to skip it
    return(1);
}
```

- **Perform "early" system initializations by writing _system_pre_init() function:**
  - It's called by compiler's boot routine (rts430_eabi.lib)
  - Overload compiler's function by writing your own
  - Compiler's default pre-init function is found in the Run-Time Support library – it's empty except for return(1);
- **Returning 1 tells the compiler to initialize global and static variables, while 0 tells it to skip this this step**

Lab 9 Exercises

Lab Exercises

- **Lab A – Count Power Cycles with Non-Volatile Variable**
  - **Create a non-volatile variable – use it to count the # of power-cycles**
    - **Blink LED the # of times there's been a power cycle**
    - **printf() to console the # of power cycles**
  - **Use custom sections and linker command file to create the NVM variable**
  - **(Flash only) Use API to write to NVM**
  - **Use memory map and memory browser to ascertain where variables were allocated by the linker**
- **(FRAM) Alternate Lab A – Use PERSISTENT pragma**
- **(F5529) Alternate Lab A – Low Wear Flash writes**
  - **Explore the provided, albeit simple, low-wear flash write example**
- **(FR5969) Lab B – MPU Configuration**
  - **Configure MPU to use 2 segments**
  - **Write to 'read/execute-only' segment of FRAM to cause a memory violation interrupt**

** Note: We don't have a (FR4133) Lab B… but the LCD chapter contains an extension of Lab9a.*

Lab 9a – Using Non-Volatile Variables

lab_09a_info_fram (or lab_09a_info_flash)

This lab uses non-volatile memory to store a data value so that it will be available after a power-cycle.

The value will be stored in Info memory, which a non-volatile memory (NVM) segment set aside for data information. The 'F5529 uses flash technology to store non-volatile information, while the 'FR5969 & 'FR4133 use FRAM.

The code will keep track of how many power-cycles (BOR's) have occurred. After power up and initializing the GPIO, the code looks for a count value in NVM, it then increments the count value and:

- Writes the updated value back to Flash or FRAM
- Prints out the # of power-cycle counts with printf()
- Blinks the LED count # of times

To minimize typing, we created the project for you. The "hello.c" file in this project is an amalgam of labs:

– `lab_03a_gpio` for the gpio setup

– `lab_04b_wdt` for the printf functionality

To this we've added:

– Logic to manage the "count" value

– For the ‘F5529, we wrote a function which writes to Flash Info B – since it needs to be erased before being written to. (The FRAM devices don’t need this step!)

– You will need to fill in a few answers from your Lab 9a worksheet

There is no MPU "protection" setup for the 'FR5969 FRAM in this exercise. That is shown in `lab_09b_mpu_gui` or `lab_09b_mpu_with_driverlib`. *(Note that the F5529 and FR4133 devices don't have an MPU.)*

Worksheet

1. Examine the linker command file (.cmd) and find the name of the memory area that represents the Info memory. (You only need to complete the table for your processor.)

Processor	Memory	Section Name	Address
F5529	**INFOB**		
FR5969	**INFOB**		
FR4133	**INFOA**		

Finish this line of code:

```
#pragma __________________ (count, "____________")
static uint16_t count;
```

2. Again, looking at the linker command file, what address symbol is created by the linker to represent the starting address of executable code?

____

F5529

3. **('F5529 only)** What functions are needed to erase and write to Flash?

(Note: We're interested in writing 16-bit integers to Flash.)

```
        //Erase INFOB
        do {

            ___________________________( (uint8_t*)INFOB_START );
            status = FlashCtl_eraseCheck(
                        (uint8_t*)INFOB_START,
                         NUMBER_OF_BYTES );
        } while (status == STATUS_FAIL);

        //Flash Write

        _________________________________________(

                       (uint16_t*) value,
                       (uint16_t*) flashLocation,
                       1
        );
```

File Management

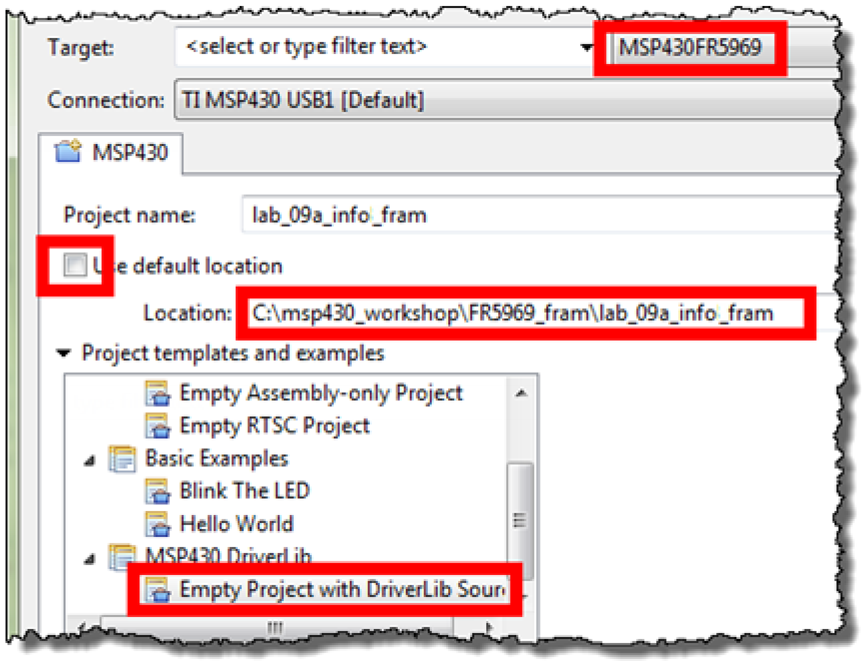
1. **Close any open projects or files.**
2. **Create a new project in the appropriate lab folder.**

   Use the *"Empty Project with DriverLib Source"* project template.

   Make sure you create your project in the **correct folder**:

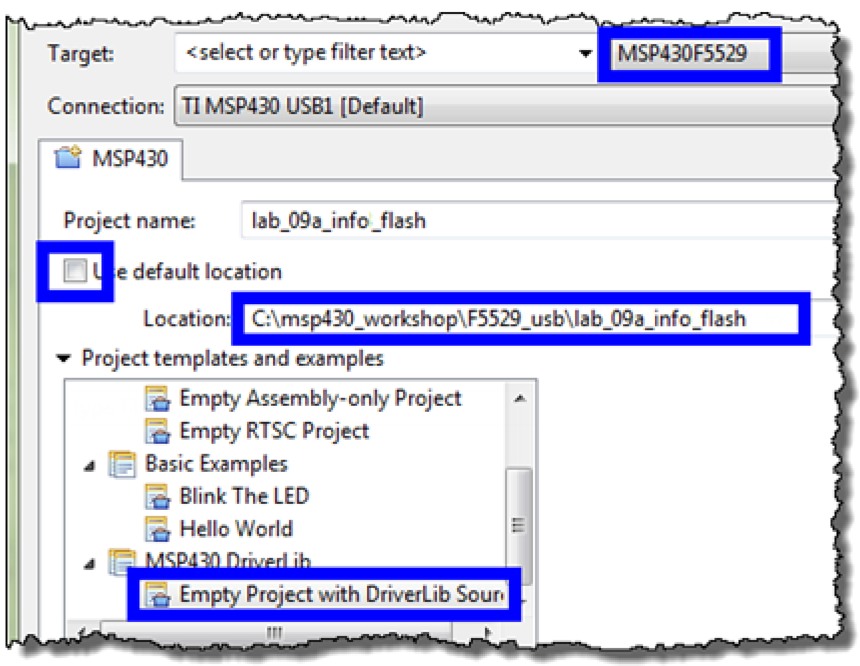
FR5969

`C:\msp430_workshop\F5529_usb\lab_09a_info_flash`



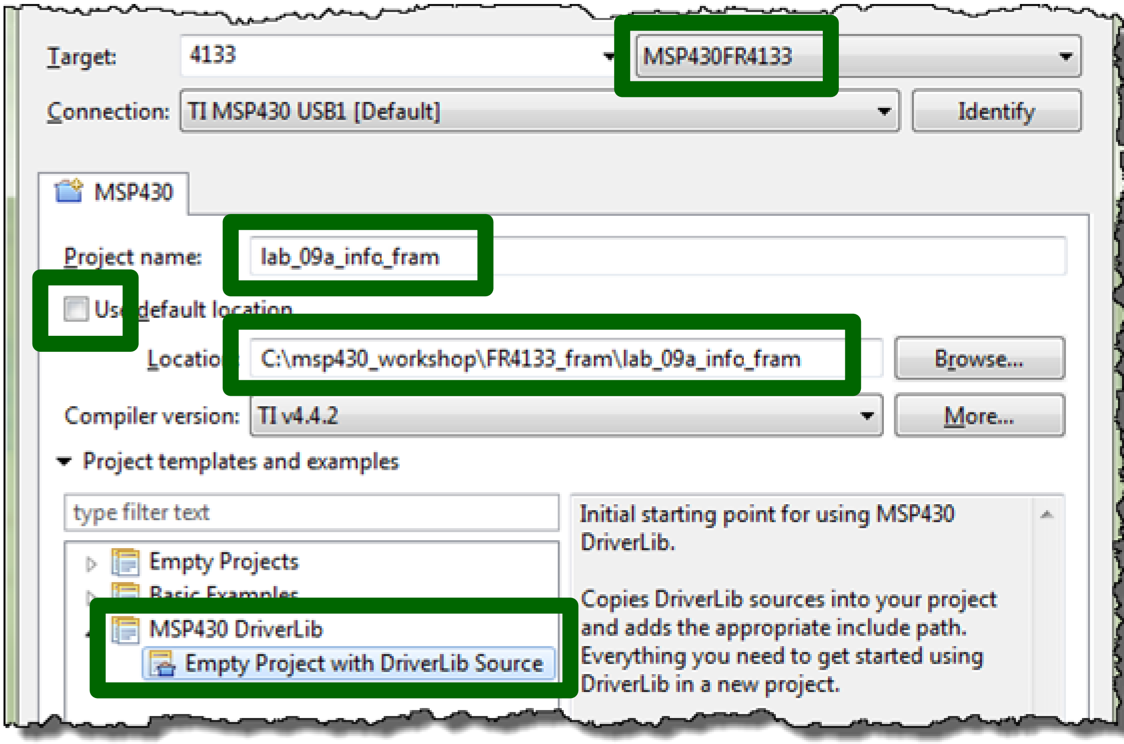
or `C:\msp430_workshop\fr5969_fram\lab_09a_info_fram`

F5529



FR4133

or `C:\msp430_workshop\fr4133_fram\lab_09a_info_fram`

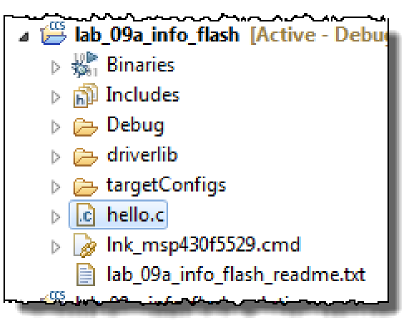


3. **Delete `main.c` from the project.**

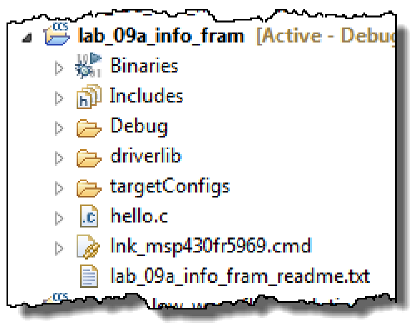
   This isn't needed since we've provided the file `hello.c` file which contains *main()*.

4. **Verify that your project contains the file `hello.c`.**

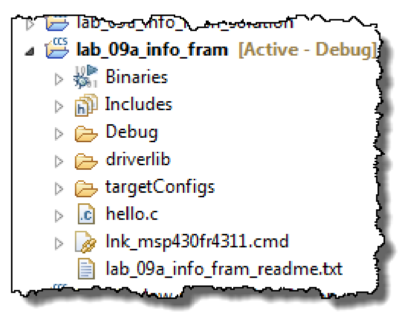
   It should look like:



or



or



If this file is missing, then you probably created the project in the wrong directory. You can either add this file to your project (from the directory shown in Step 1) or delete the project and start over again.

Edit Code

5. Fill in the blanks in the `hello.c` file.

Use your answers from the worksheet questions (page 9-43).

6. Increase the heap size to 320.

This was a change we performed back in Lab 2 in order to get C Standard I/O to work. Here's a quick reminder:

```
Right-click on Project → Properties…
Build → MSP430 Linker → Basic Options → Heap Size
```

FR5969

7. ('FR5969 only) Modify the .infoB setting in linker command file.

Since FRAM reads/writes like SRAM, the compiler auto-initializes it each time our C program starts … just like any other global variable. Of course, that's not what we want in this instance – we want to use the non-volatile nature of FRAM to maintain the value of 'count' when the power is off. To make this happen, we can tell the tools to "not initialize" the variables. This can be done by editing one line in the linker command file to add the NOINIT type.

```
.infoB     : {} > INFOB   type=NOINIT
```

We could have limited the scope of our NOINIT modification, but it's an easier edit to set this type for the entire .infoB section.

Note: This step isn't needed on the 'FR4133 device – even though it's also FRAM based.

While the 'FR5969 has a more advanced MPU, it's not turned on by default. Conversely, the 'FR4133 has a simple memory protection mechanism, but it is enabled by default.

Build and Evaluate

8. **Build program the program.**

   Fix any syntax errors and rebuild until your program compiles successfully.

9. **Open the .map file (from your project's Debug folder) and answer the questions below.**

   The .map file is a report created by the linker which records where memory was allocated. (We used **INFOA** for the **FR4133** and **INFOB** for **FR5969** and **F5529**).

	'F5529	'FR5969	'FR4133
Which INFO Section was used?	**INFOB**	**INFOB**	**INFOA**
Address of INFOA or INFOB			
Where was this INFOA/INFOB address specified to the tools?			
Address of .infoA or .infoB			
Compiler's Boot Routine: **_c_int00** (.text:isr)			
Main Code (.text)			
Length of code* (.text)			
Address of *count*			
fram_rx_start			

*Note that turning on the optimizer may allow the compiler to build a smaller program. Also, you would not want to use printf() in a production level program as this leads to very inefficient programs.

10. **Why does the code (.text) section start so far away from the beginning of Main Flash or FRAM?** *(Hint: Look at the section allocations in the .cmd file.)*

____

____

Run the Program to Watch the Non-Volatile Variable

11. Launch the debugger.

12. Open the Memory Browser window.

```
View → Memory Browser
```

Try looking at some of the locations used in our code:

```
0x1900 (or 0x1800)
&fram_rx_start (for 'FR5969 devices)
&count
```

From the Memory Browser, what is the address of: &count ____________________________

13. To watch their values, add variables to the Expressions Window for:

```
count
c (for 'F5529 devices)
i (you can also see 'i' in the local Variables window)
```

Hint:	You may want to change the number format for "c" to "hex": `Right-click expression → Number Format → Hex`

14. Single-Step through the code to watch it work.

The Memory Browser is interesting because you can see the variable in Flash (or FRAM).

Hint:	You can also modify the value in Flash by changing it in the Memory Browser. This is convenient if you want to reset the value back to 0. This same hint works for FRAM too, but it's not as surprising that we can change FRAM so easily in the debugger

15. Restart the program.

If you let the program run without a breakpoint, you may need to S*uspend* it before *Restart.*

16. Step through the code again ... hopefully it retained its count value.

You should see the printf() statement output the latest *count* value, as well as the LED blink one more time than during the previous run.

17. Terminate the debugger and unplug the board – then plug it back in.

Do you see the LED blinking? Again, it should be 1 more time than previously.

18. Reset the Launchpad with the reset button ... does the LED blink 1-more-time each time its reset or power-cycled?

Just clicking the reset button on your board (without unplugging/plugging it) should be enough to restart the program and increment *count*.

FR5969

FR4133

(FRAM Devices Only) lab_09a_persistent

As discussed in this chapter, the MSP430 compiler has a pragma to define *persistent* variables. This method of creating persistent variables is easier to use than the method shown in lab_09a_info.

Worksheet

(Hint: Please refer to the Chapter 9 discussion in the Workshop PDF for help with these questions.)

1. **Write the line of code that tells the compiler to make the variable "count" into a persistent, non-volatile variable.**

   In the previous part of this exercise, creating a non-volatile variable took two steps:

   - ⃠ Specify the variable should go into a specific section using #pragma DATA_SECTION
   - ⃠ Edit the linker command file to declare the output data section as "type=NOINIT"

   What new pragma replaces these two steps?

```
#pragma ___________________________________ ( count )
uint16_t count = 0;
```

2. **When using this pragma, what section name does the compiler place the variable into?**

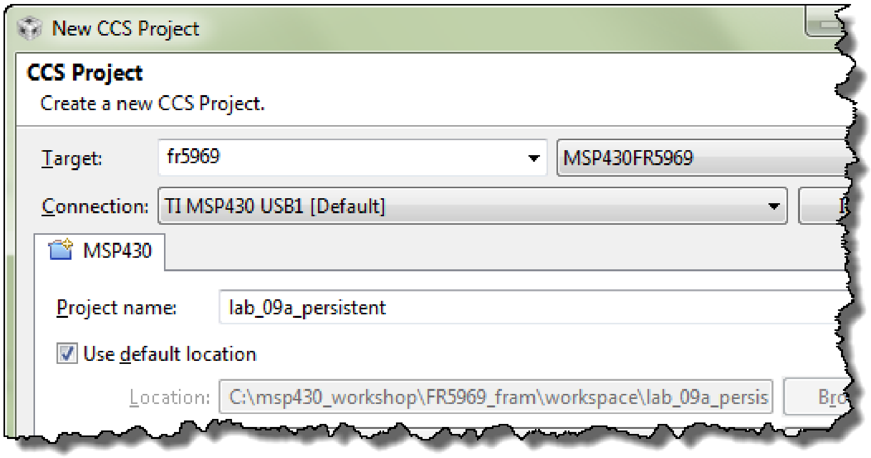
   ____

3. **What action causes a Persistent variable to be initialized?**

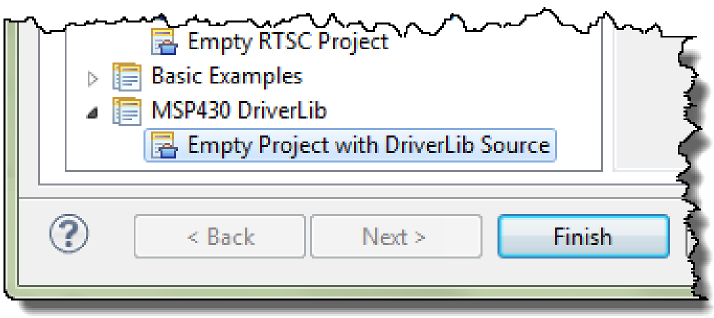
   ____

File Management

4. **Create a new CCS DriverLib project named `lab_09a_persistent`.**



Make sure you choose the DriverLib template in the dialog, then click Finish.



5. **Copy/Paste the file `hello.c` from the previous lab exercise.**

   In Project Explorer, copy `hello.c` from `lab_09a_info_fram` and paste it into `lab_09a_persistent`.

6. **You can now close the `lab_09a_info_fram` project.**

7. **Delete `main.c` from the new project.**

   We don't need to keep the generic/default `main.c` file since `hello.c` (which we just copied into our project) contains the *main()* function.

8. **Increase the *heap* size to "320" so that STDIO will work.**

9. **Build your project and fix any errors.**

   Before we started editing the code, let's make sure we didn't introduce any errors when creating our new project. *(In fact, this is how we realized that we needed to tell you to delete the default `main.c` file.)*

Edit Code

10. Edit `hello.c` to use the new pragma rather than the old one.

Comment out the old pragma that specified the infoB (or infoA) data section and enter the new pragma which declares our variable as *persistent* (referring back to your answer in step 1 on pg 9-49).

Your code should now look something like this:

```

//***** Global Variables ***********************
//#pragma DATA_SECTION (count, ".infoB")
#pragma PERSISTENT ( count )
uint16_t count = 0;

```

Build and Run

11. Build the project and fix any errors you encounter.

12. Look up the following details in the `lab_09a_persistent.map` file.

Hint: (1) *Look for the .map file in the project's Debug folder.*
(2) *Double-click linker command file to open in the CCS editor*
(3) *Use Control-F to open search dialog – then search for "count" and ".TI.persistent"*

What address is count located at? ______________________________

Is this address located in the .TI.persistent output section? ________________

Referring to the memory-map shown in the chapter, what part of the memory map is .TI.persistent located at? (Circle the correct answer)

`INFOA    INFOB    INFOC    INFOD    MAIN`

13. Click the *Debug* toolbar button to enter the debugger and load the program to your FRAM Launchpad.

14. Verify that your code works as expected.

Similar to the previous lab exercise (`lab_09a_info` steps 15-18 pg. 9-48), verify that your count variable persists – and is incremented – after each reset and/or power cycle.

Initialzing a Persistent Variable

15. Terminate the debugger, if CCS is currently in Debug mode.

16. Power-cycle the Launchpad and count the number of LED blinks. (By unplugging, and then re-plugging in your board.)

We're asking you this so that we can get a baseline number for our next step. Remember, each time we power-cycle the board, *count* should be incremented and the LED should blink that number of times.

# of LED blinks after power-cycle: ____

17. Make sure your Launchpad is plugged in and then click the *Debug* toolbar button.

18. After the debugger is launched and the program is loaded into FRAM by CCS… what is the current value of *count*?

Look in the Expressions Window (or the Memory Window) to get the value for *count*.

count = ____________________

Explain how *count* was changed to its new value? _________________________________

____

____

19. Terminate the debugger and close the project

F5529

('F5529 Only) (Optional) lab_09a_low_wear_flash

'F5529 only -- FRAM parts rarely need to worry about wear issues due to their high endurance.

This example modifies `lab_09a_info_flash` by using the entire infoB segment. In the original exercise, we wrote count to the first location in Info B. On the next power-cycle we erased the entire Info B segment and only wrote one location; we did this again-and-again on every power-cycle.

This solution provides a simple method of minimizing FLASH wear. Rather than erasing the entire flash on each power-cycle, we now use consecutive locations in flash. We keep doing this until we reach the end of InfoB; only when we reach the end of InfoB do we erase the entire segment and start over again.

While there are probably better algorithms to handle these types of flash wear issues, this is a simple example solution to the problem.

```
Import and explore the lab_09a_low_wear_flash solution
```

FR5969

('FR5969 Only) Lab 9b – Protecting Memory

As explored in Chapter 9, it's important to protect your executable program and read-only data stored in FRAM using the Memory Protection Unit (MPU). The *FRAM – Usage and Best Practices* application note puts it this way:

> **NOTE:** It is very important to always appropriately configure and enable the MPU before any software deployment or production code release to ensure maximum application robustness and data integrity. The MPU should be enabled as early as possible after the device starts executing code coming from a power-on or reset at the beginning of the C startup routine even before the *main()* routine is entered.

The following lab exercise takes you through a couple of different ways you can set up the MPU:

- Using the MPU Graphical User Interface (GUI) found in CCSv6
- Using DriverLib code in MPU initialization function called from *main()*
- Using DriverLib code in MPU initialization function called from *_system_pre_init()*

You'll find the GUI method to be quick and easy – thus we recommend that all FRAM users complete this exercise. While the 2nd and 3rd examples are not difficult, evaluating their code takes a little bit more time and effort, therefore we've marked them as "optional".

lab_09b_mpu_gui

Using the CCSv6 GUI to automatically configure the MSP430 MPU.

File Management

1. **Import the `lab_09a_persistent_solution.zip` project file.**

   You can skip this step if you completed this project and want to use it, otherwise, import the previous lab's project solution.

2. **Rename the project you just imported to: `lab_09b_mpu_gui`**

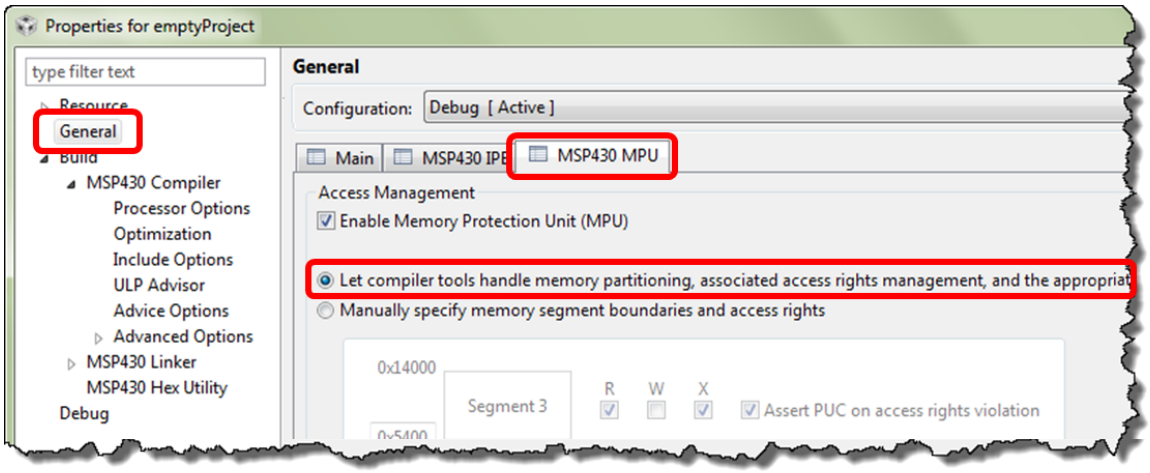
3. **Verify all other projects are closed.**

4. **Build the project to verify the project imported correctly.**

Enable MPU

5. Open the `lab_09b_mpu_gui` project properties and setup the MPU GUI.

`Right-click on the project → Properties`



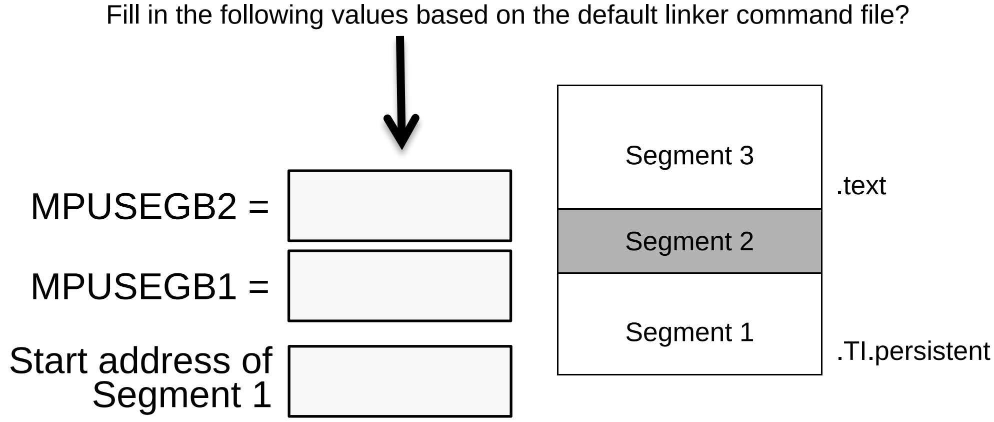
Click OK once you have configured the MPU as shown.

6. Build the project.

7. Open the linker command file (.cmd) and determine the expected MPU settings.

The GUI – along with the linker command file – configures the MPU as two segments. In this case, it sets both segment border registers to the same value.

Fill in the following values based on the default linker command file?



`Hint:` The MPU segment registers should be set to the address shifted right by 4.
For example: `fram_rx_start >> 4`

8. Open the `lab_09b_mpu_gui.map` file to determine the starting address of Segment 1.

What is the starting address of `.TI.persistent`? ______________________________

How does this compare with your expectation? ______________________________

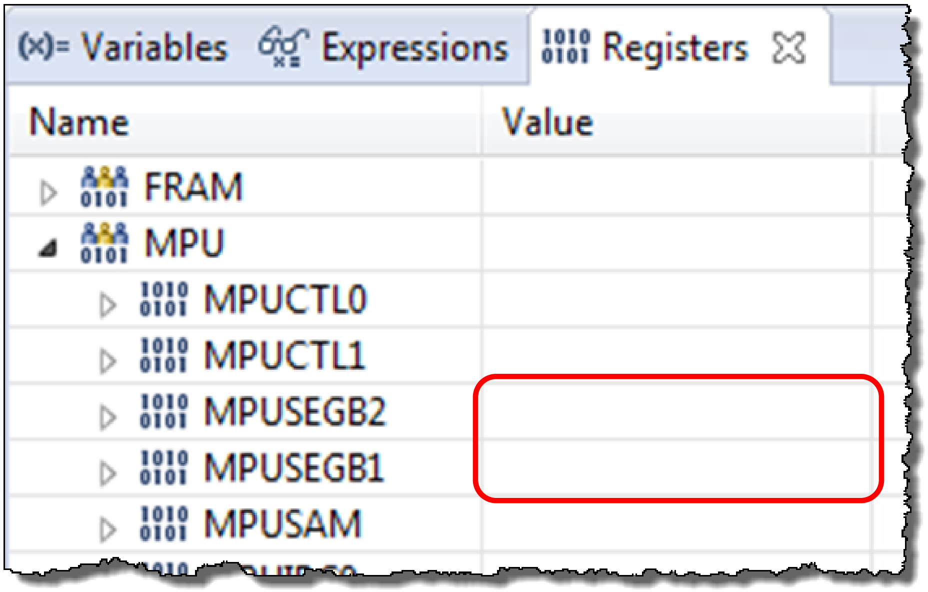
Debug and Verify

9. **Launch the debugger. Let the program load and run to *main()*.**

10. **Compare your expectations versus the actual MPU register settings.**

    The MPU settings, as configured by the GUI, are written to the registers during as part of the compiler's initialization; therefore, the MPU settings are already set by the time the program counter reaches *main()*.

    Copy down the settings for the MPU Segment Border registers:



    How do they compare to your expectations? ____________________________________

11. **Once you're done exploring the automatic GUI settings, you can Terminate the debugger and close the project.**

FR5969

(Optional) lab_09b_mpu_with_driverlib

This lab explores the use of the Memory Protection Unit (MPU). We program the MPU using DriverLib and then set about violating the assigned protections by trying to write into protected memory segments. We set up these violations to create NMI (non-maskable interrupt) events.

Project comments

- Builds on lab_09a_info_fram (that flashes the LED the number of times the program has been reset or power-cycled)
- Uses *_system_pre_init()* function to configure WDT and MPU before reaching *main()*
- Initializes the MPU:
  - Using 2 segments (with border address defined by the linker command file)
  - Setting up violation on write to Segment 3 (where code is located)
  - System NMI is generated on violation (as opposed to PUC)
  - MPU is started, but not locked
- A "violation" function in the program tests the MPU's configuration by writing to the various segments – trying to create violations; the results are reported back via *printf()*
- An example of the FR5969 reset handlers are provided; including a function that tests for why the program was last reset
- A simple example for creating SYSTEM event flags is provided. This can be used to flag reset/interrupt events so that your main program can respond to them (if needed). These flags were allocated with PERSISTENT storage.

Files in the project:

- **`hello.c`** : Carried over from the previous lab, but quite a bit has been added to it.
- **`myMPU.c`** : Provides the function that initializes the MPU; as well as the function which causes memory violations
- **`system_isr_routines.c`** : Includes the interrupt handlers for Reset, System NMI, and User NMI events. Additionally, it contains our *_system_pre_init()* function call.

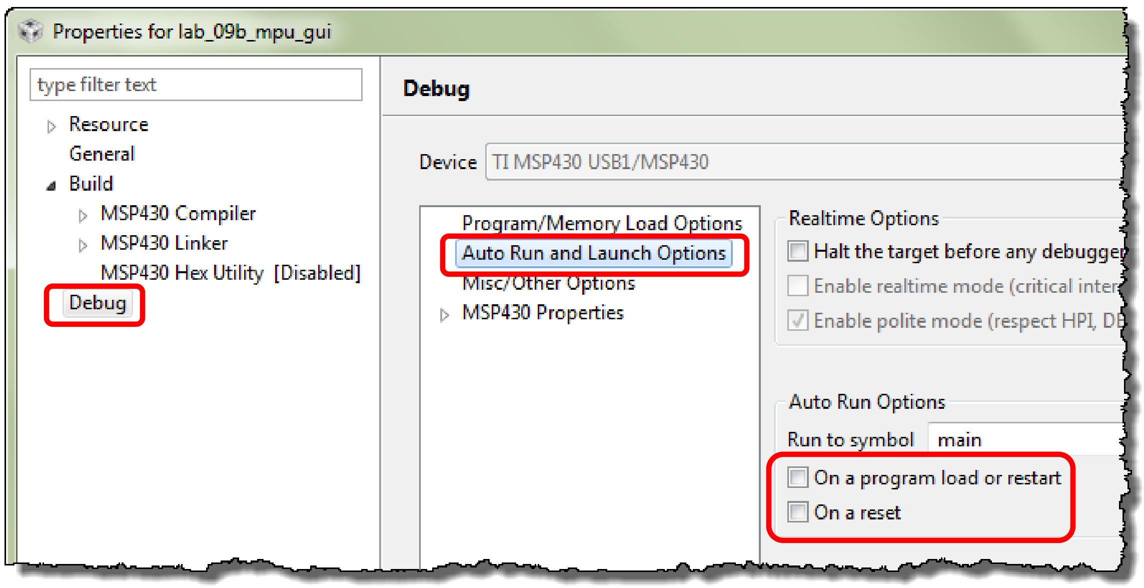
Reference

The `system_isr_routines.c` file provides a good template for handling MSP430 System Reset Events. For more information about this, check out the wiki page:

http://processors.wiki.ti.com/index.php/Handling_MSP430_System_Reset_Events

Basic Lab steps

- ❑ Import the `lab_09b_mpu_with_driverlib` project
- ❑ Build the project
- ❑ Run the program and examine the *printf()* output to the *Console* window
- ❑ Suspend the program and put a breakpoint at the start of *_system_pre_init()*
- ❑ Import "watch_expressions.txt" from the lab folder into the Expressions window
- ❑ Reset the CPU and single-step through the *initMPU()* to see how these functions work – watch how the MPU registers get modified
- ❑ Set breakpoints on the different cases in the NMI interrupt handler that are related to the 4 different FRAM segments. Why don't we get *Info* and *Segment 1* interrupts?
- ❑ Try changing the 'enablePUC' and 'enableNMI' options, each time rebuilding the program to see how this affects the results of the memory segment violation tests
- ❑ Before launching the debugger, turn off the "auto run" feature:



Notes

Chapter 9 Appendix

Worksheet (Q1, Q2)

1. Examine the linker command file (.cmd) and find the name of the memory area that represents the Info memory.

Processor	Memory	Section Name	Address
F5529	INFOB	.infoB	0x1900
FR5969	INFOB	.infoB	0x1900
FR4311	INFOA	.infoA	0x1800

Finish this line of code:

```
#pragma DATA_SECTION (count, "__________")
static uint16_t count;
```

2. Again, looking at the linker command file, what address symbol is created by the linker to represent the starting address of executable code?

fram_rx_start

Worksheet (Q3) – 'F5529 Only

3. **('F5529 only)** What functions are needed to erase and write to Flash?

(Note: We're interested in writing 16-bit integers to Flash.)

```
//Erase INFOB
do {
    FlashCtl_segmentErase( (uint8_t*)INFOB_START );
    status = FLASH_eraseCheck(
                (uint8_t*)INFOB_START,
                NUMBER_OF_BYTES );
} while (status == STATUS_FAIL);

//Flash Write
FlashCtl_write16 (
                (uint16_t*) value,
                (uint16_t*) flashLocation,
                1
);
```

lab_09a_info (Q9)

9. **Open the .map file (from your project's Debug folder) and answer the questions below.**
   The .map file is a report created by the linker which records where memory was allocated.

	'F5529	'FR5969	'FR4133
Which INFO Section was used?	INFOB	INFOB	INFOA
Address of INFOA or INFOB	0x001900	0x001900	0x001800
Where was this INFOA/INFOB address specified to the tools?	Linker Command File		
Address of .infoA or .infoB	0x001900	0x001900	0x001800
Compiler's Boot Routine: _c_int00 (.text:isr)	0x004400	0x004800	0x00D61C
Main Code (.text)	0x010000	0x010000	0x00C5D0
Length of code* (.text) (Your values may vary…)	0x0012FC	0x001258	0x0011F8
Address of count	0x001900	0x001900	0x001800
fram_rx_start	N/A	0x004800	N/A

lab_09a_info (Q10)

10. **Why does the code (.text) section start so far away from the beginning of Main Flash or FRAM?** *(Hint: Look at the section allocations in the .cmd file.)*

Because that's how they were specified in the default linker command file (.cmd).

Here's some snippets from the 'FR5969 linker command file.

```
FRAM                     : origin = 0x4400, length = 0xBB80
FRAM2                    : origin = 0x10000,length = 0x4000
```

```
.text:_isr        : {}  > FRAM          /* CODE ISRs */
.text             : {} >> FRAM2 | FRAM  /* CODE      */
```

You'll find similar results for "FLASH" in the 'F5529 linker command file.

lab_09a_persistent (FR5969 Only)

1. Write the line of code that tells the compiler to make the variable "count" into a persistent, non-volatile variable.

In the previous part of this exercise, creating a non-volatile variable took two steps:

- Specify the variable should go into a specific section using #pragma DATA_SECTION
- Edit the linker command file to declare the output data section as "type=NOINIT"

What new pragma replaces these two steps?

```
#pragma PERSISTENT ( count )
uint16_t count = 0;
```

2. When using this pragma, what section name does the compiler place the variable into?

.TI.persistent

3. What action causes a Persistent variable to be initialized?

Loading the program into FRAM using CCS

lab_09a_persistent (FR5969 Only)

12. Look up the following details in the `lab_09a_persistent.map` file.

Hint: *(1) Look for the .map file in the project's Debug folder.*
(2) Double-click linker command file to open in the CCS editor
(3) Use Control-F to open search dialog – then search for "count" and ".TI.persistent"

What address is count located at? **0x4400**

Is this address located in the .TI.persistent output section? **Yes**

Referring to the memory-map shown in the chapter, what part of the memory map is .TI.persistent located at? Circle the correct answer:

INFOA INFOB INFOC INFOD MAIN (circled)

18. After the debugger is launched and the program is loaded into FRAM by CCS… what is the current value of *count*?

Look in the Expressions Window (or the Memory Window) to get the value for *count*.

count = **0**

Explain how *count* was changed to its new value? **Clicking Debug toolbar button causes CCS to load the program… which initializes *Persistent* variables**

lab_09b_mpu_gui (FR5969 Only)

7. Open the linker command file (.cmd) and determine the expected MPU settings.

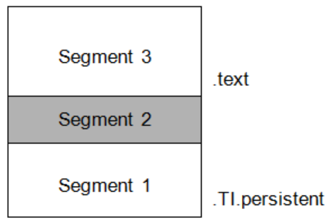
The GUI – along with the linker command file – configures the MPU as two segments. In this case, it sets both segment border registers to the same value.

Fill in the following values based on the default linker command file?

MPUSEGB2 = **0x480**

MPUSEGB1 = **0x480**

Start address of Segment 1 **0x4400**



8. Open the `lab_09b_mpu_gui.map` file to determine the starting address of Segment 1.

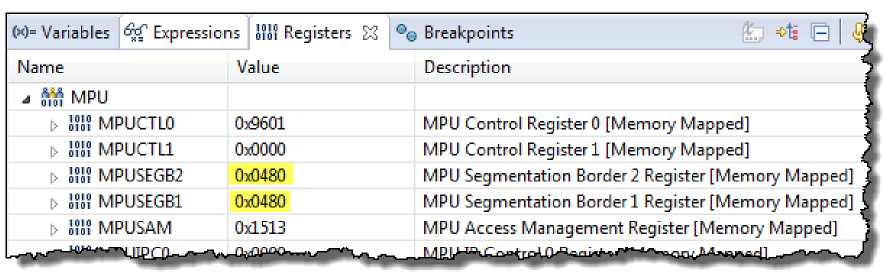
What is the starting address of `.TI.persistent`? **0x4400**

How does this compare with your expectation? **Matches our expectation; we expected Segment 1 to contain the read/write data – while Segment 3 would contain the read/execute content**

10. Compare your expectations versus the actual MPU register settings.

The MPU settings, as configured by the GUI, are written to the registers during as part of the compiler's initialization; therefore, the MPU settings are already set by the time the program counter reaches *main()*.

Copy down the settings for the MPU Segment Border registers:



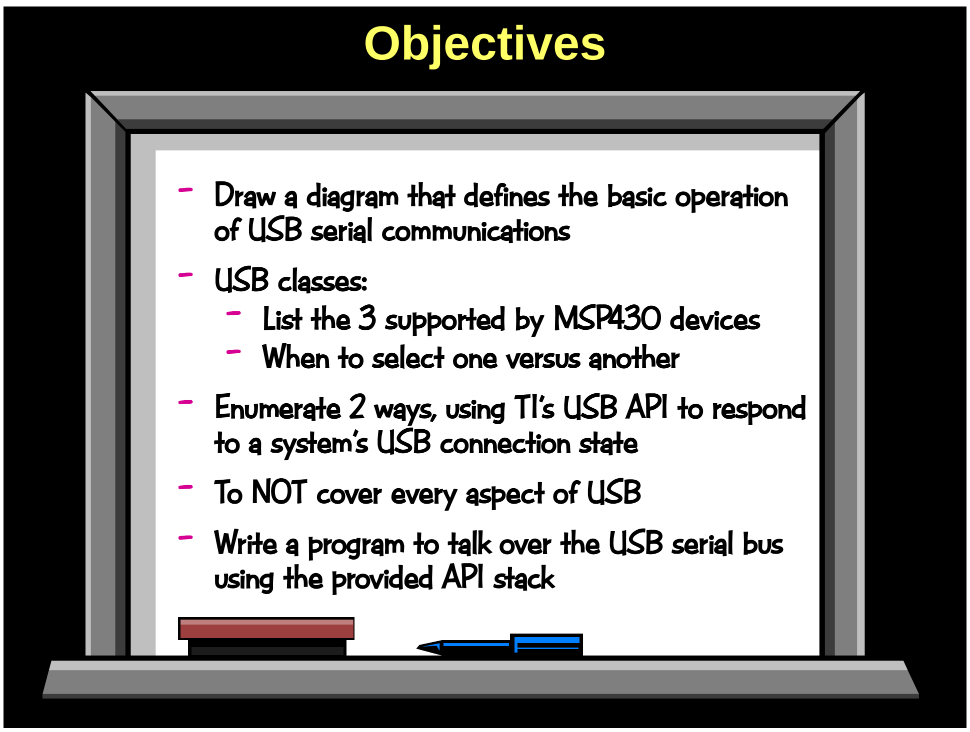
Name	Value	Description
MPU		
MPUCTL0	0x9601	MPU Control Register 0 [Memory Mapped]
MPUCTL1	0x0000	MPU Control Register 1 [Memory Mapped]
MPUSEGB2	0x0480	MPU Segmentation Border 2 Register [Memory Mapped]
MPUSEGB1	0x0480	MPU Segmentation Border 1 Register [Memory Mapped]
MPUSAM	0x1513	MPU Access Management Register [Memory Mapped]

How do they compare to your expectations? **Matches expectations**

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 10-1

Introduction 10-1

What is USB? 10-3

MSP430's USB Support 10-7
USB Fees 10-13

How USB Works 10-15
Pipes and Endpoints 10-16
USB Transfer Types 10-19
The USB Frame 10-20

Descriptions and Classes 10-22

Quick Overview of MSP430's USB Stack 10-28

ABC's of USB 10-31
A. Plan Your System 10-31
B. Connect & Enumerate 10-32
C. Managing my App & Transferring Data 10-34

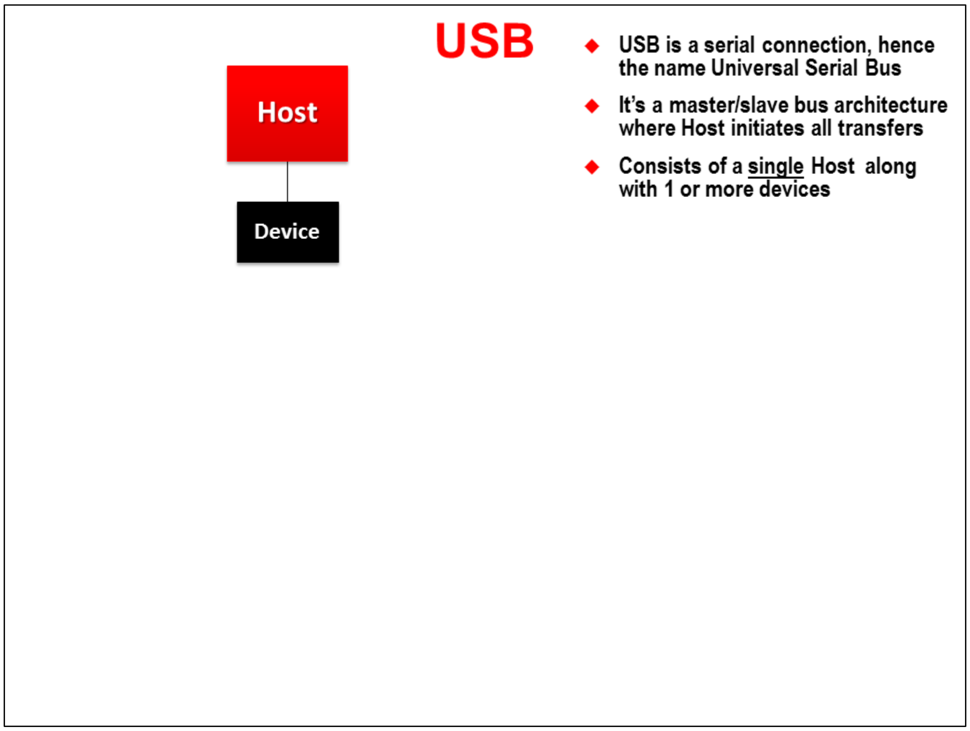
Final Thoughts 10-37

Lab Exercise 10-39

What is USB?

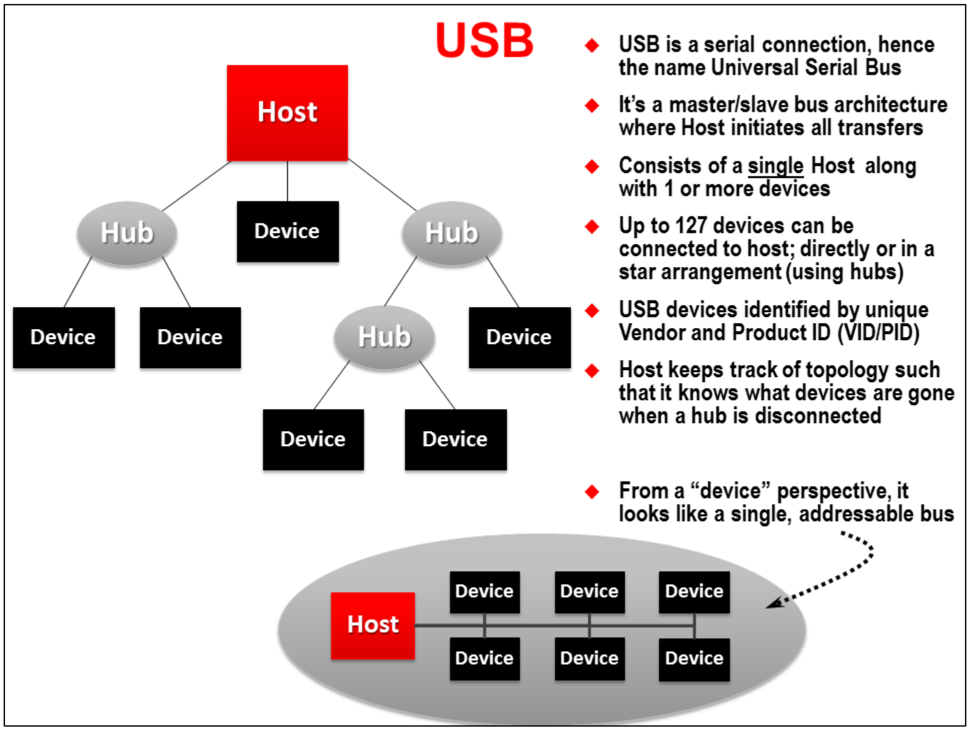
Universal Serial Bus (USB) is just that, a universal serial connection between a “Host” and “Device”. It has taken the place of many synchronous and asynchronous serial bus connections in systems such as personal computers.

In the case of USB, the *host* manages all transfers, whether moving data to or from the *host* – often this is called a master/slave architecture, where the *host* is the bus master. At a minimum, there needs to be one *host* and one *device*.



But… USB supports many more than just a single *device*, the standard can actually support up to 127 different *devices*. Commonly, systems with multiple devices use hubs as interconnection points between the *host* and *devices* – which results in a star arrangement.

Each type of device is distinguished using Vendor and Product ID (VID/PID). The combination of VID and PID allows a host to identify the type of device that is connected and manage the point-to-point communications with it – in most cases, this requires the host to load the appropriate drivers required to talk with that specific type of device. (We'll discuss this is greater detail later in the chapter.)



The Universal Serial Bus protocol has gone through a few versions over time. Back in 1995 USB revision 1.1 was released. This version provided separate host and device connectors along with supporting two different speeds: *Low* speed moved data at speeds up to 1.5Mpbs (megabits-per-second); while *Full* speed provided data rates up to 12Mbps.

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	• 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	• 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

In 2000, USB 2.0 was released as an upgrade to USB 1.1. Along with *Low* and *Full* speeds, a much faster *High* 480Mbps rate was added. Other major additions to the standard included a power supply of 500 mA provided from the USB cable, as well as capability for advanced devices to switch between Host and Device modes – called On-The-Go (OTG) mode. The OTG feature is handy in some applications where a product might have to be a Device or a Host depending upon what it is connected to.

The MSP430's USB port supports the USB 2.0 standard, but only operating at the *Full* rate. (Seeing that the fastest MSP430 devices only run up to 25MHz, it's not hard to wonder why they cannot support the 480Mbps rate.) Additionally, since the MSP430 doesn't provide Host support, it therefore does not provide the OTG Host/Device switching feature.

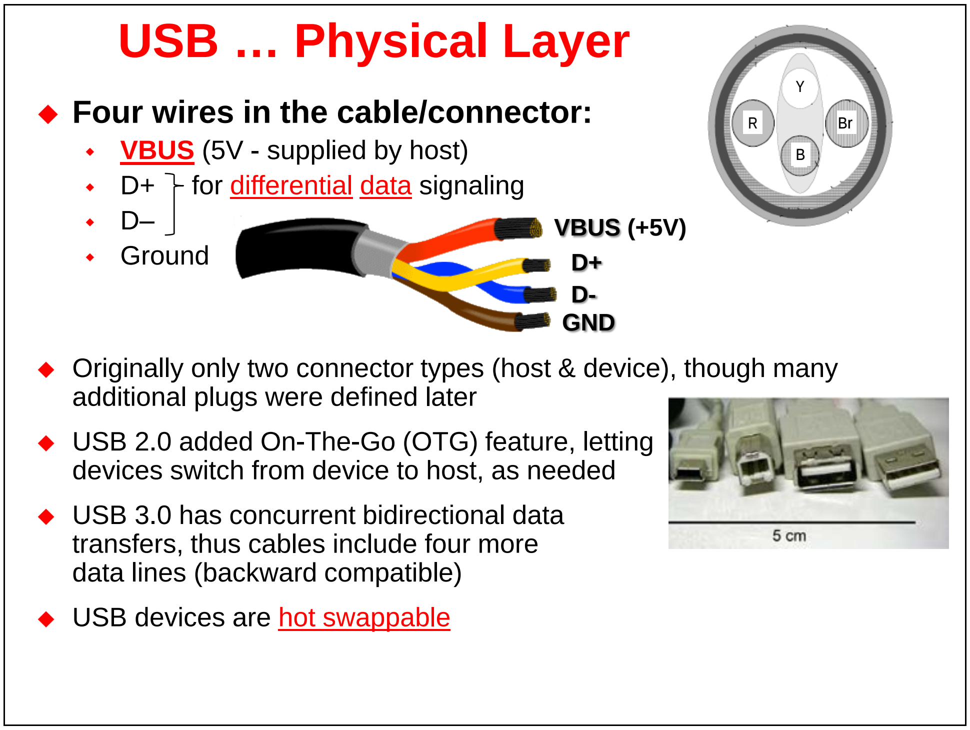
Hint: If your product needs Host and or OTG support, you may want to check out TI's Tiva-C line of ARM Cortex M4F processors.

Just a few years ago, in 2008, USB added the 3.0 revision. While once again backward compatible to USB 1.1 and USB 2.0, the new revision added an additional *Super* 4.8Gpbs rate. It also included full-duplex operation, a higher power sourcing availability of 900 mA as well as other power-management features. While this is quite advantageous for many types of end-applications – such as hard disk drives, high-end imaging systems (i.e. scanners), and such – it's overkill for many other systems, where low power and cost are primary directives.

Bus standards, such as USB, contain a variety of layers. While these physical and data specifications are important, exploring them in great detail is outside the scope of this chapter.

On the following slide, we'll introduce a couple basic features of the *physical* layer – that is, of the USB cable. Later on in the chapter, we will discuss some of the details regarding data and software layers.

Bottom Line: We have tried to approach USB, in this chapter, from a pragmatic perspective. That is, rather than examining the details of the specification, we want to figure out how to use TI's devices and tooling in order to just get something running.



As shown above, the USB cable provides four different signals:

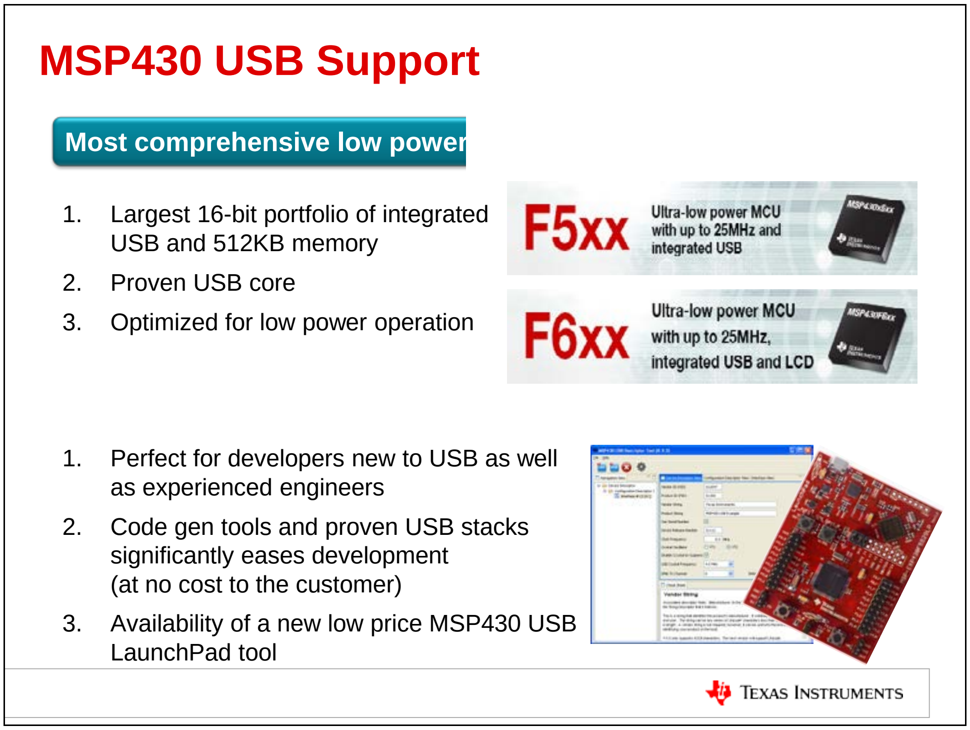
- One signal pair provides power and ground. The power signal, called VBUS, is a +5V supply. Not only does this pair provide USB 2.0 devices with up to 500 mA of power, but bringing this signal high (+5V) is how the Host begins communicating to a Device. (We'll see more about this later in the chapter.)
- The other pair of signals, D+ and D-, provides a differential data signal from the Host to the Device. As most hardware engineers know, using differential signaling provides more robust data transmissions.

USB 3.0 cables provide more additional signals to support its higher performance; although, that's not something we need to deal with in this chapter.

Finally, the USB standard supports "hot swappable" devices. This means they can be connected and disconnected as needed. The USB protocol provides a means for handling this feature. To this same end, your USB application should remain flexible. By this, we mean that your application needs to be written so that it can handle an asynchronous change in the USB connection state. This might include the Host putting your Device into *Suspend* mode (where you receive a reduced power supply) … or the end-user disconnecting your *Device* from the *Host* by "yanking the cable".

MSP430's USB Support

As we stated on the first page, the MSP430 proves to be an excellent solution for building USB Devices. Many devices in the F5xx and F6xx MSP families contain the USB peripheral. Coupling this proven USB hardware port with the low-power nature of the MSP430 makes possible some interesting USB applications.



Besides the low-power advantages of the MSP430, though, the software tools and USB stack make the MSP solution really stand-out.

The USB standard is a very capable, and therefore involved, protocol. The TI tools, along with the MSP430 USB stack (i.e. USB library), make it possible for novices and experienced users to take advantage of this capability.

Combining these software tools with the MSP430 USB Launchpad makes an excellent low-cost development environment.

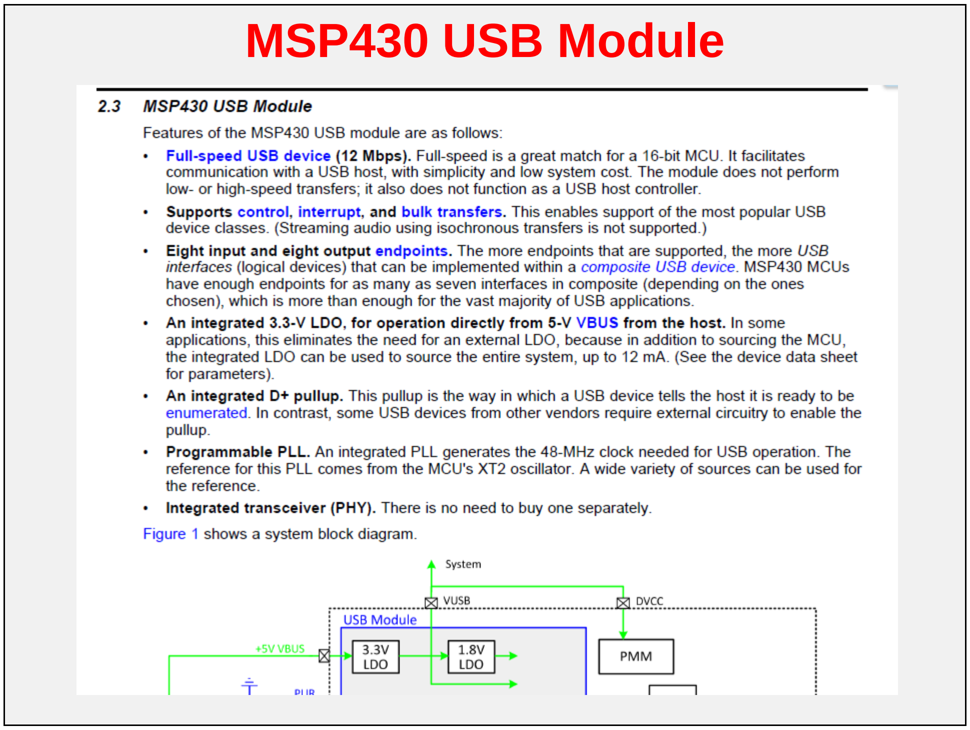
This table summarizes some of the MSP430 devices that provide USB functionality. As you can see, there are a variety of processors with different memory and peripheral options.

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**

The following slide, taken from the 'F5529 User's Guide, lists many of the MSP430 USB module's features. While we've already spoken about the *Full*-speed capability, unless you're already quite familiar with the USB standard, most of the other features listed probably won't make much sense… yet.



MSP430 USB Module

2.3 MSP430 USB Module

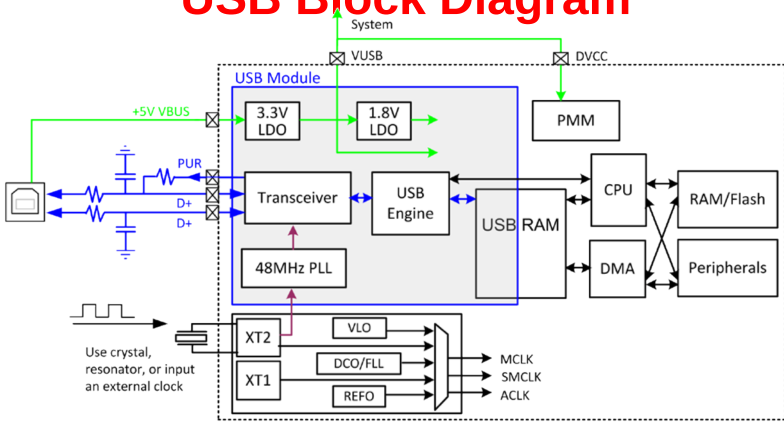
Features of the MSP430 USB module are as follows:

- **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.

We'll address many of the data/system oriented features throughout the rest of the presentation. You might note here, though, the hardware specific features. For example, "including the PHY" (physical interface) means there's one less thing for you to put on your board. Also, the USB port has its own dedicated block of SRAM (though the system can use it when the USB port is disabled).

USB Block Diagram



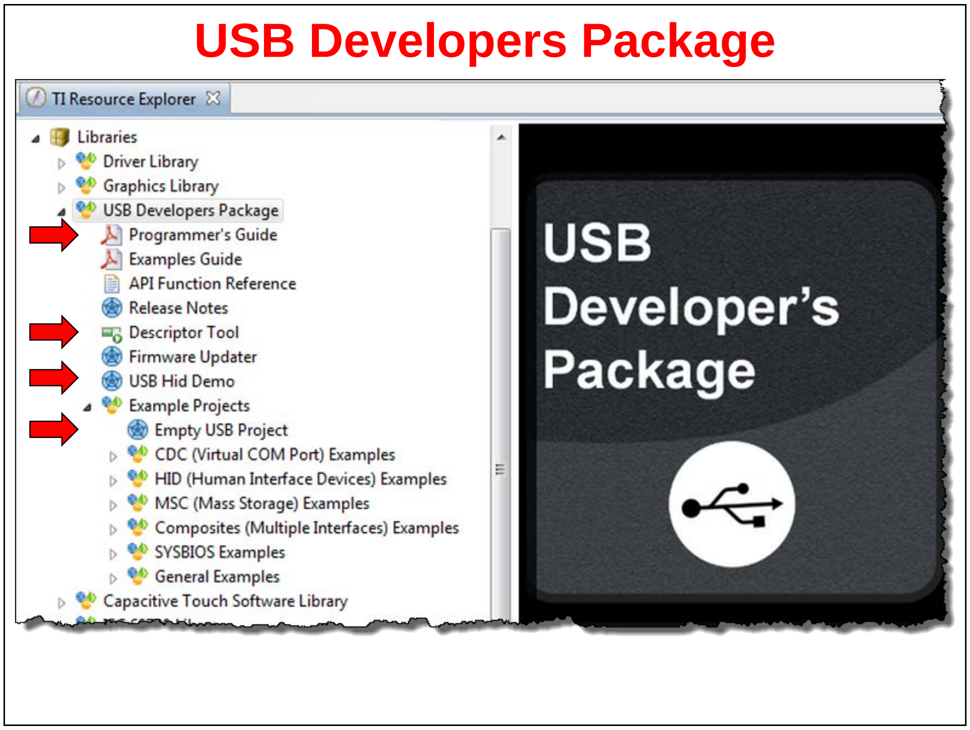
MSP430 USB Peripheral Supports

- 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)
- USB 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)

Also, notice the LDO voltage regulators. These let the port (and even the MSP430 device itself) operate from the +5V supply coming from an attached USB cable. Finally, the built-in PLL handles the required USB clock requirements, utilizing one of the MSP430 external clock inputs.

We bragged about the MSP430 development support. Here's a peek at it. Looking at the items pointed out by the red arrows:

- We begin with the excellent USB Programmers Guide
- The Descriptor Tool is truly unique. It makes easy work of the tedious (and error prone) job of creating USB interface descriptors.
- The USB HID Demo is a Host-side tool that lets you interact with custom devices implementing the Human Interface Device class. It's like a serial terminal program for HID devices.
- Finally, the rich set of examples provided by TI not only provides a way to "play" with USB, they also make excellent starting points for your own applications.



Sidebar – MSP430 USB API Features

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

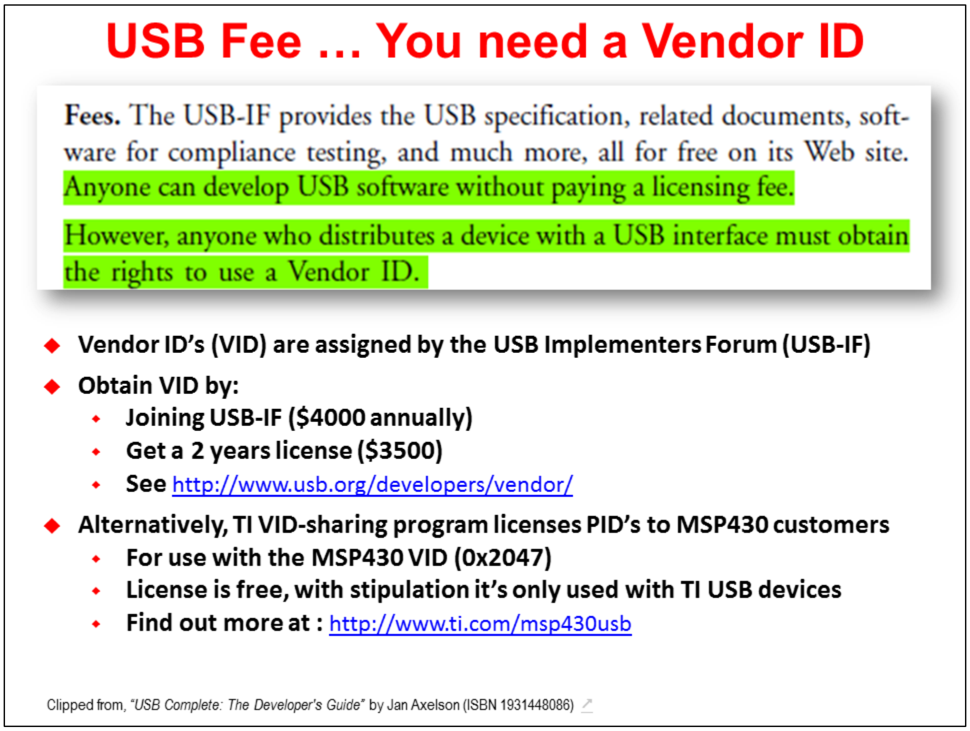
MSP430 USB API Features, cont.

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 Fees

As we described earlier, your USB product needs a Vendor and Product ID (VID & PID) in order to meet the requirements of the standard. The USB Implementers Forum (USB-IF) charges a fee to license a Vendor ID.

As an alternative to purchasing your own VID, silicon vendors such as Texas Instruments, will provide you the ability to use their VID when using the MSP430 USB-based devices. Please refer to TI's website for more information on obtaining a VID/PID.

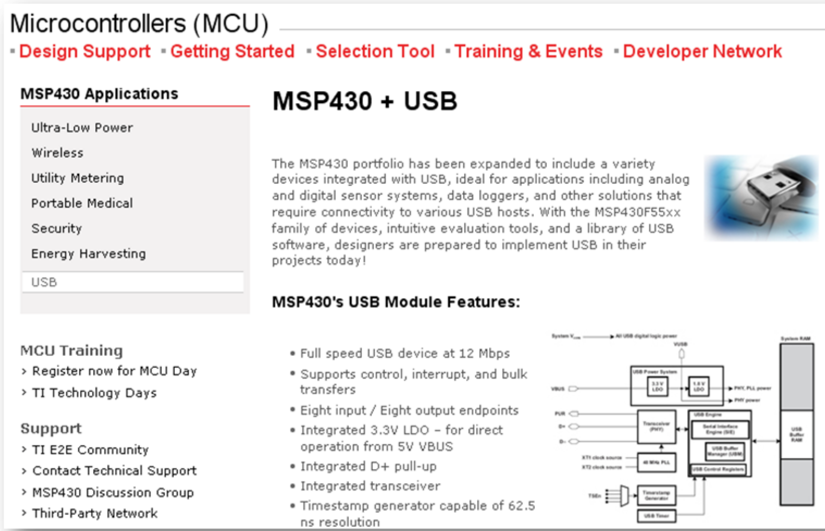


Additional USB Resources

Along with TI's MSP430 USB page, we've provided some USB references that we found useful.

http://www.ti.com/msp430usb

Come here to get up to date for all things related to MSP430 USB!



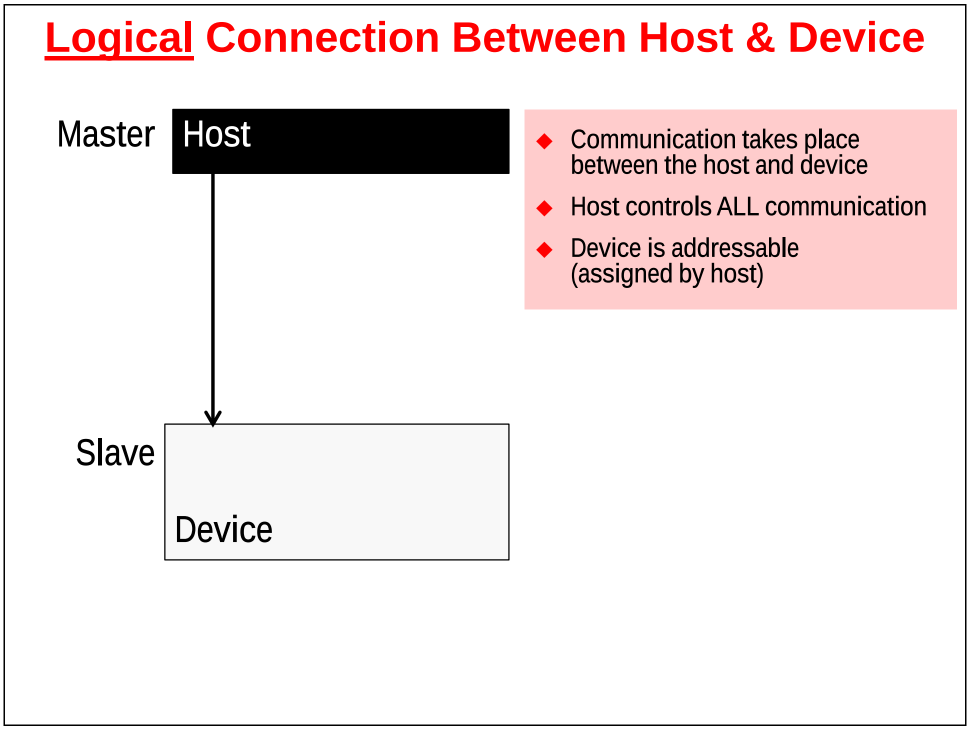
Suggested Reading

- ***"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

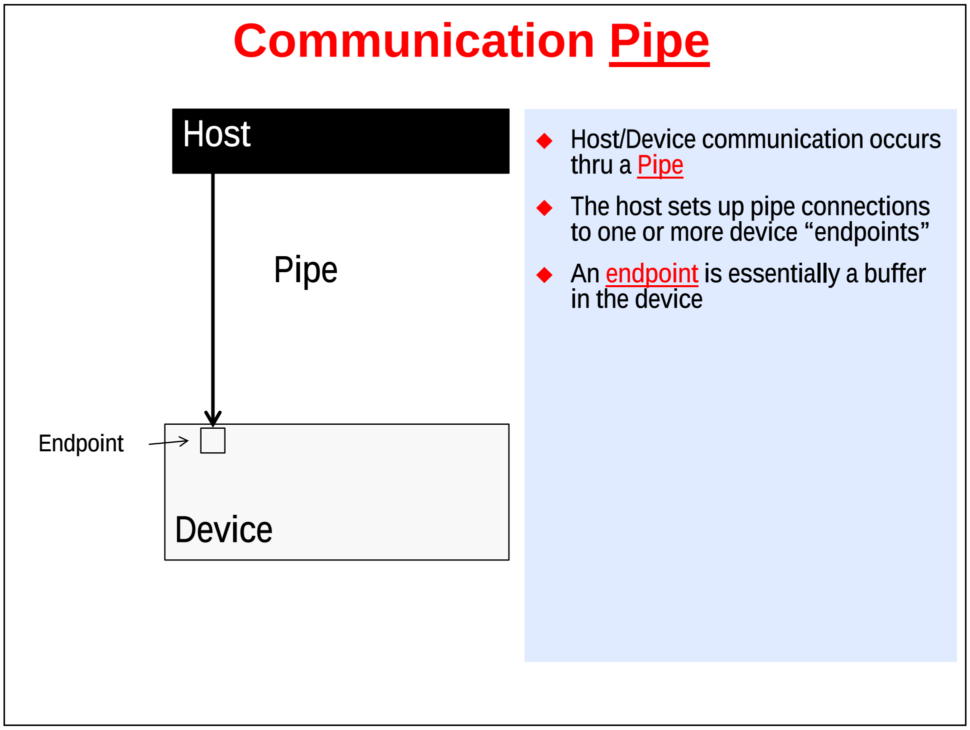
As we stated at the beginning of the chapter, USB is a serial, Master/Slave communication protocol. That is, the Host acts as the Master; communication to and from the *Host* is directed by the *Host*. The *Device* only responds to requests from the *Host*.

The USB standard allows many *Devices* to be connected to a single *Host*. The *Host* assigns an address to each *Device* as it is connected (i.e. enumerated) to the *Host*. This is really a minor detail, though, since – as a *Device* – we don't need or use this information.

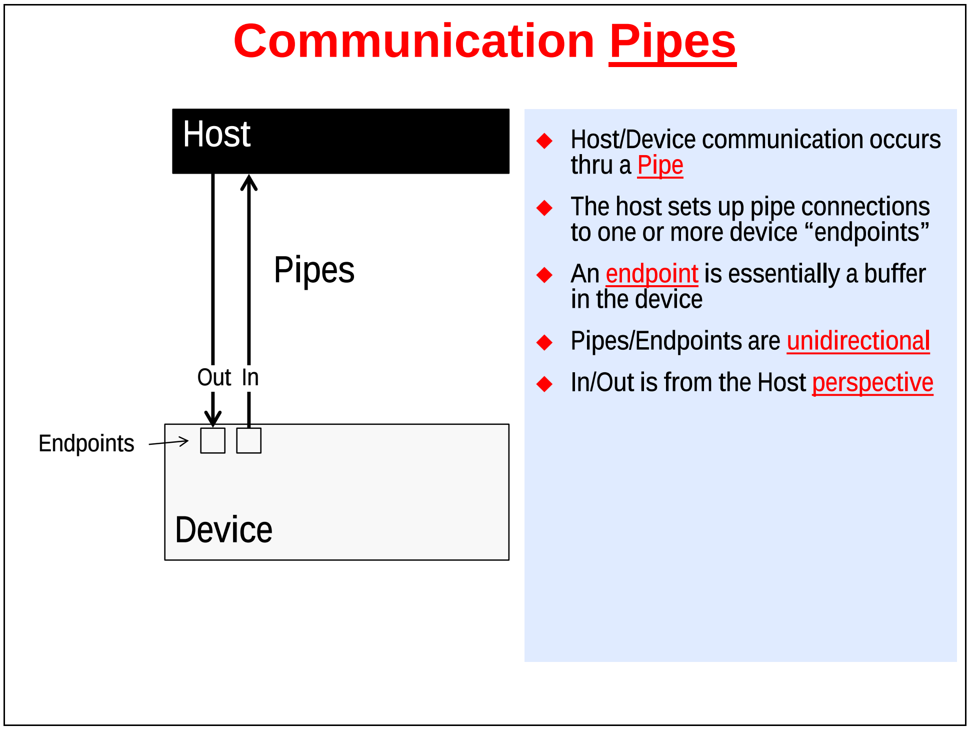


Pipes and Endpoints

To be more specific, a *Host* communicates with a *Device* through a ***Pipe***; that is the name given to this communication pathway. The *Pipe* makes a connection to a *Device* ***Endpoint***; which is essentially just a buffer in the *Device*. (As we'll see in a minute, the MSP430 has dedicated *Endpoints* in its USB port hardware.)

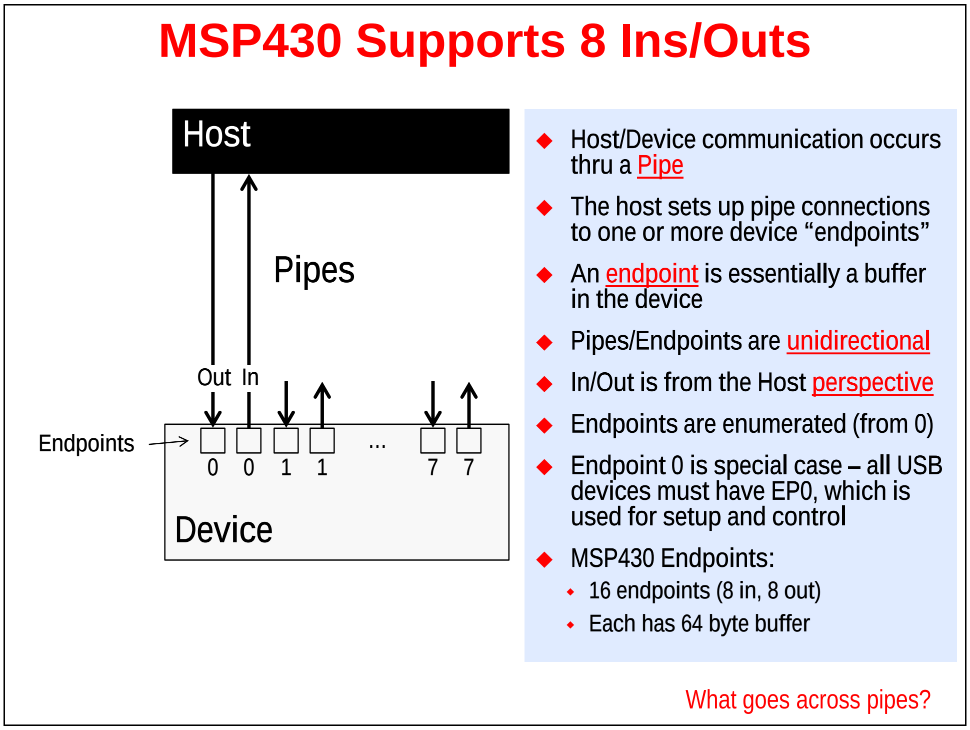


Pipes specify unidirectional data movement. If you want to move data in both directions, two *Pipes* must be created – which requires 2 Endpoints. Also, seeing as *Pipes* (and USB, in general) are *Host* centric, the directions *In* and *Out* are from the *Host's* perspective.



While the USB standard only requires a *Device* to have one Input and one output *Endpoint*, the MSP430 USB port provides 16 Endpoints: 8 Input and 8 Output. Additionally, the MSP430 Endpoints each contain a 64-byte buffer – the largest specified in the USB specification. All-in-all, this hardware provides the MSP430 with a lot of flexibility in the types of communications it supports.

As shown below, the set of Input and Output Endpoints are numbered 0 – 7.



We often see the *Endpoints* referred to as EP0, EP1, … EP7.

The In/Out *Endpoints* do not have to be used in bidirectional pairs – sometimes you may find that your *Device* needs 2 Inputs and 1 Output.

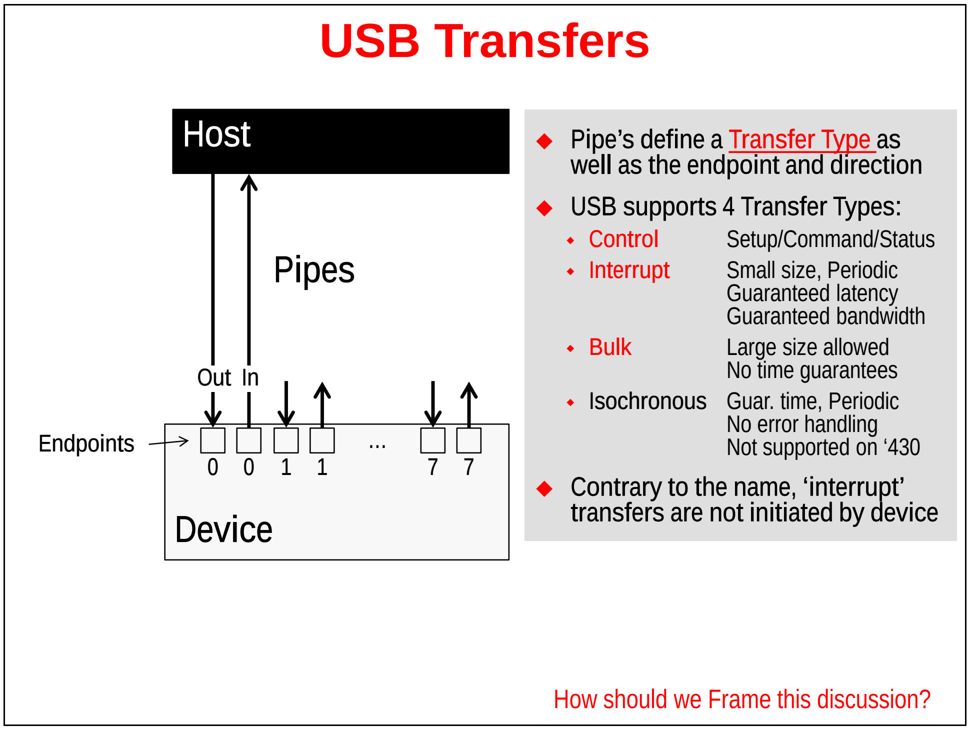
By the way, do you remember when we said that the USB spec requires a *Device* to have at least 1 *Endpoint*?

That happens to be *Endpoint* 0 (EP0). EP0 is a special case; the *Host* uses EP0 (both directions) to setup and control USB operations. Without the *Host* being able to rely on a known *Endpoint* 0 always being available, it wouldn't know how to start talking to new *Devices* as they're physically connected.

So, we've established the concept of a communication Pipe … what gets transferred across it?

USB Transfer Types

Along with specifying an *Endpoint* and direction, a *Pipe* also specifies the "*Type*" of communication transfer. The USB specification supports four *Transfer Types*, as defined in this diagram.



If all we cared about was passing data across the Pipe, we wouldn't need to further define the *Transfer Type* of a Pipe. The fact is, sometimes we care about "*when*" data will arrive, just as much as the data itself.

Each of the *Transfer Types*, listed above, briefly describe their temporal nature. Notice how "*Interrupt*" types provide a guaranteed latency and bandwidth, although the tradeoff is a smaller data payload. Conversely, "*Bulk"* transfers allow large sizes, but give up the time-oriented guarantees.

Hint: "Interrupt" transfer types do not have anything to do with microprocessor "interrupts". It is just the word used in the USB specification to describe these types of transfers.

Similarly, "Interrupt" transfer types are initiated by the Host, just as all USB transfers are initiated and controlled by the Host. (We'll see more about this on the next page.)

Note: The MSP430 USB stack (i.e. USB library) only supports *Control*, *Interrupt*, and *Bulk* transfer types. Currently, the MSP430 does not support Isochronous types, which are more typically used in audio or video types of applications.

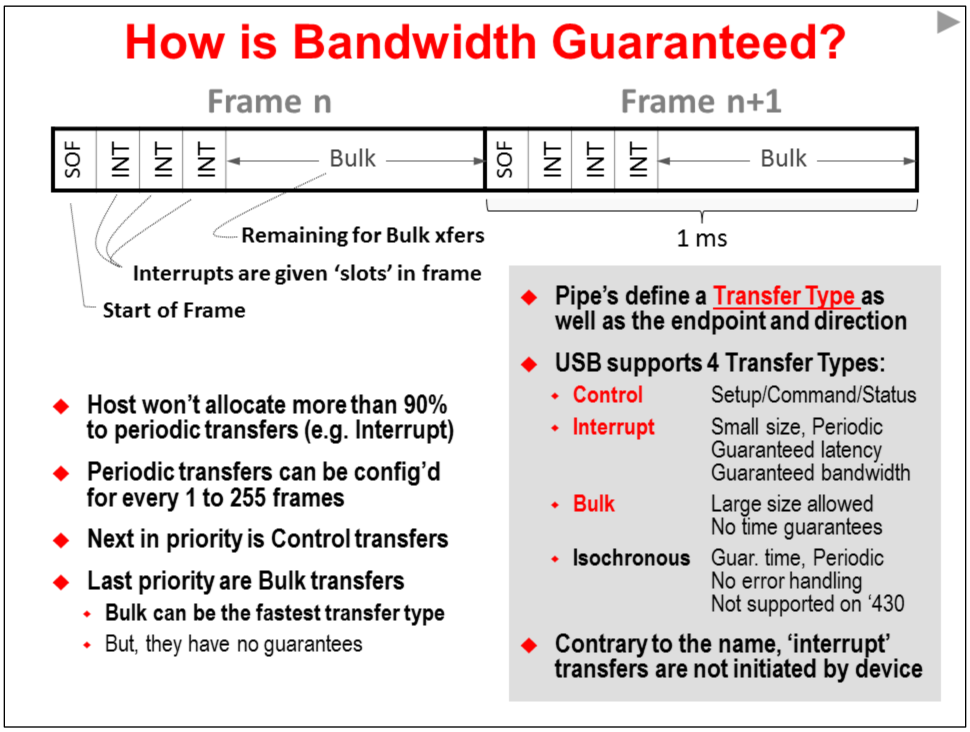
The USB Frame

If we're talking about time-oriented concepts, such as latency and bandwidth, how are these defined?

USB describes communications occurring within a 1 ms Frame. Each Frame begins with a Start-of-Frame (SOF). After that comes 'interrupt' transfer types, then 'control' types, and finally 'bulk' transfer types.

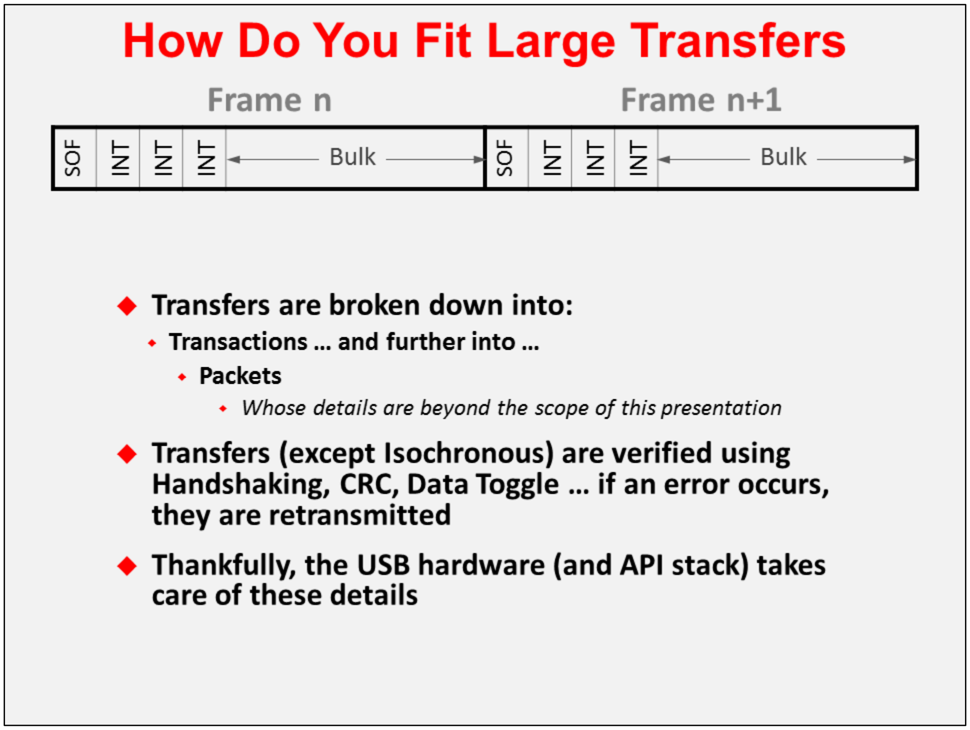
In this way, interrupt transfers are guaranteed to occur. Conversely, if you have so many interrupt transfers that the frame is near fully utilized, then bulk transfers might occur very slowly. Then again, if you don't have many interrupt or control transfers, bulk transfers will get most of the frame and complete more quickly.

Providing further flexibility, periodic transfer types (e.g. interrupt transfers) can be configured to occur in every frame – or as infrequently as once every 255 frames. This lets you specify the amount bandwidth and latency needed for a given periodic transfer – as well as potentially free up bandwidth for bulk transfer types.



Sidebar – Packets

Realistically, large transfers must be broken down into smaller chunks. USB defines these smaller chunks as 'packets'.

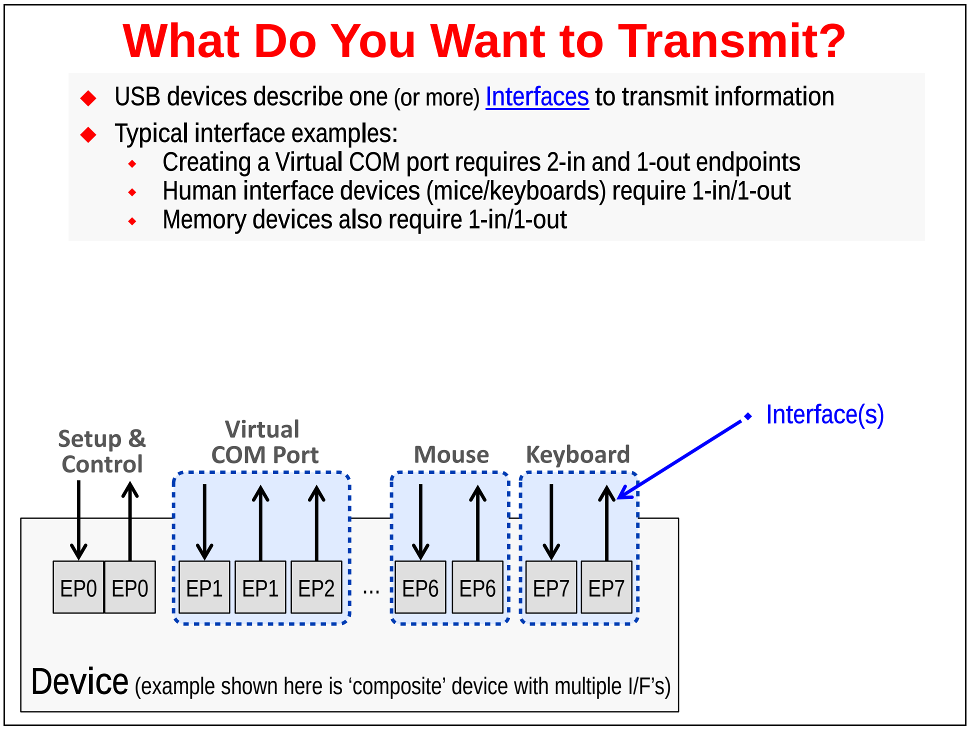


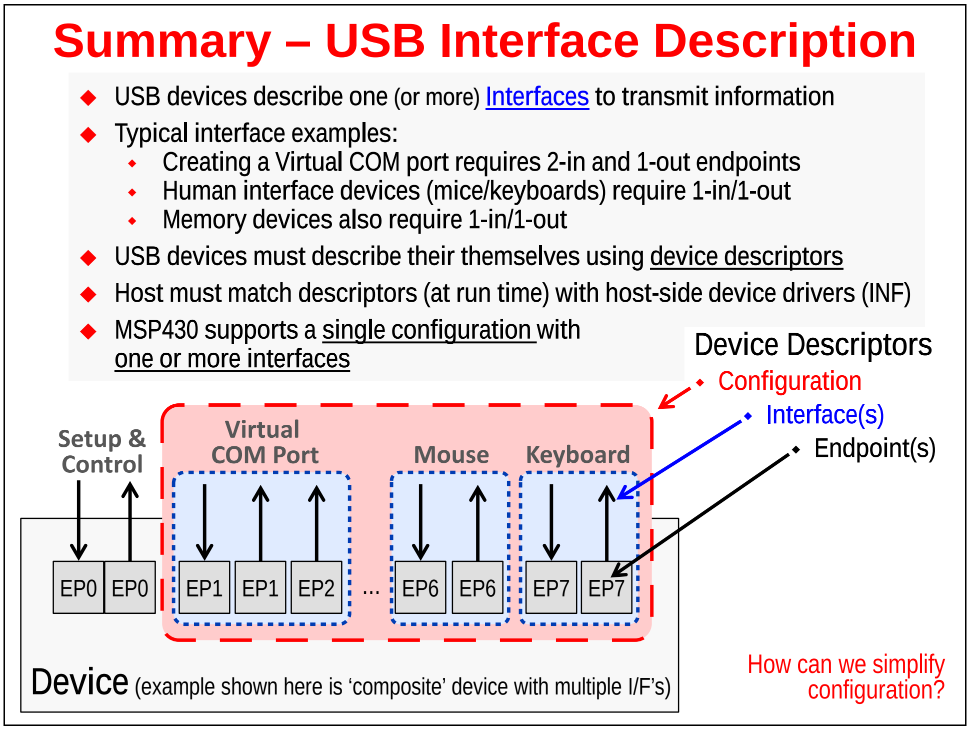
We've chosen not to dig into the details of packets – or a number of other details like handshaking, error detection, and so on. This decision was based on two factors: one, there just isn't enough time to go through every detail of the USB specification in this chapter; and two, the USB peripheral's hardware – and the TI USB stack – manage these details for us. In other words, we don't have to know them in order to get our USB application built and working.

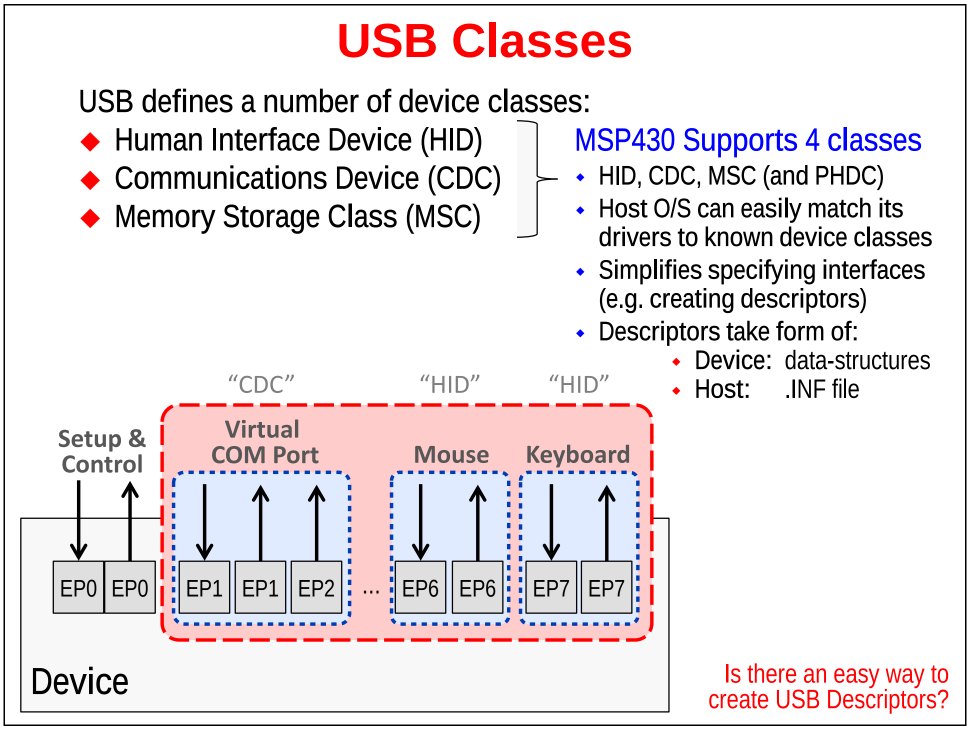
Descriptions and Classes

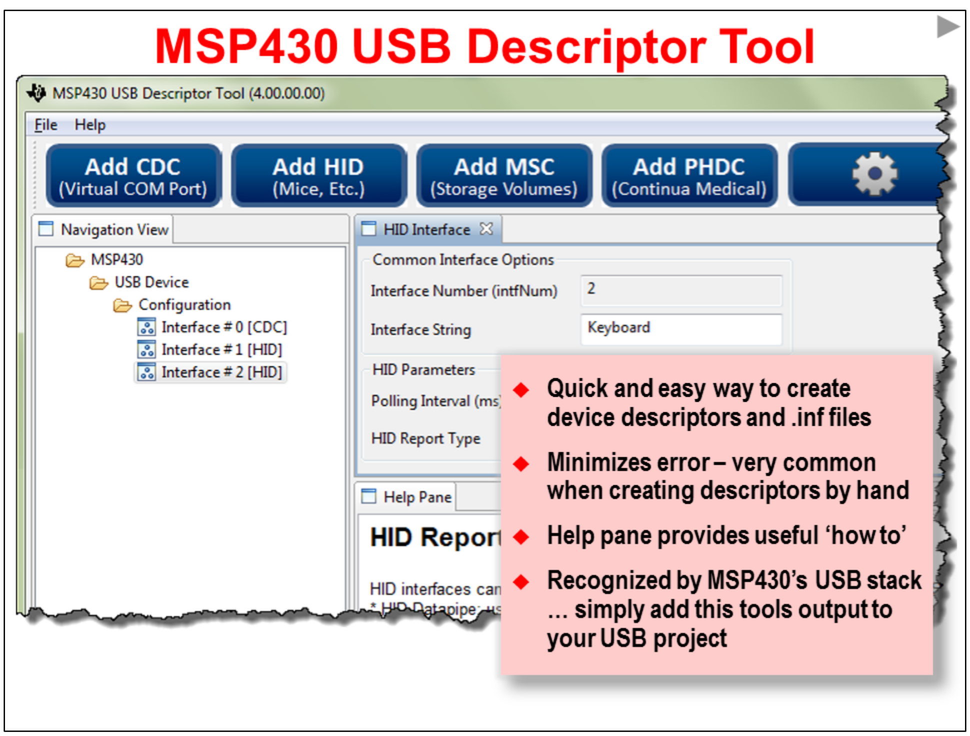
As we say on the following slide, “What do you want to Transmit?”

Are you looking to send data across the USB bus similar to a standard serial port? Maybe you’re building a human interface device and want to send mouse or keyboard data.



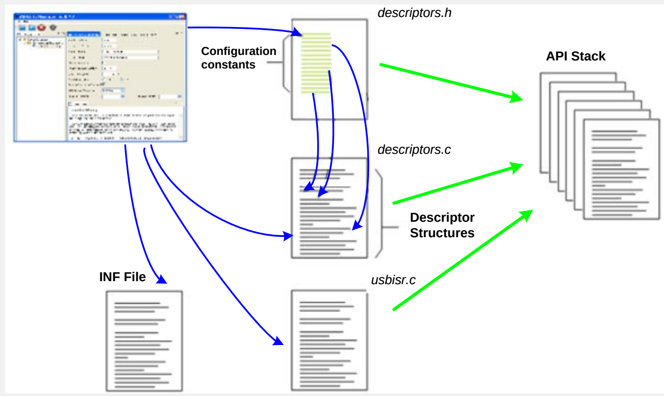

Summary – USB Interface Description
USB devices describe one (or more) Interfaces to transmit information
Typical interface examples:
Creating a Virtual COM port requires 2-in and 1-out endpoints
Human interface devices (mice/keyboards) require 1-in/1-out
Memory devices also require 1-in/1-out
USB devices must describe their themselves using device descriptors
Host must match descriptors (at run time) with host-side device drivers (INF)
MSP430 supports a single configuration with one or more interfaces
Device Descriptors
Configuration
Interface(s)
Endpoint(s)
Setup & Control
Virtual COM Port
Mouse
Keyboard
EP0
EP0
EP1
EP1
EP2
...
EP6
EP6
EP7
EP7
Device (example shown here is 'composite' device with multiple I/F's)
How can we simplify configuration?


USB Classes
USB defines a number of device classes:
Human Interface Device (HID)
Communications Device (CDC)
Memory Storage Class (MSC)
MSP430 Supports 4 classes
HID, CDC, MSC (and PHDC)
Host O/S can easily match its drivers to known device classes
Simplifies specifying interfaces (e.g. creating descriptors)
Descriptors take form of:
Device: data-structures
Host: .INF file
"CDC"
"HID"
"HID"
Setup & Control
Virtual COM Port
Mouse
Keyboard
EP0
EP0
EP1
EP1
EP2
...
EP6
EP6
EP7
EP7
Device
Is there an easy way to create USB Descriptors?


MSP430 USB Descriptor Tool
MSP430 USB Descriptor Tool (4.00.00.00)
File Help
Add CDC (Virtual COM Port)
Add HID (Mice, Etc.)
Add MSC (Storage Volumes)
Add PHDC (Continua Medical)
Navigation View
MSP430
USB Device
Configuration
Interface # 0 [CDC]
Interface # 1 [HID]
Interface # 2 [HID]
HID Interface
Common Interface Options
Interface Number (intfNum)
2
Interface String
Keyboard
HID Parameters
HID Report Type
Help Pane
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

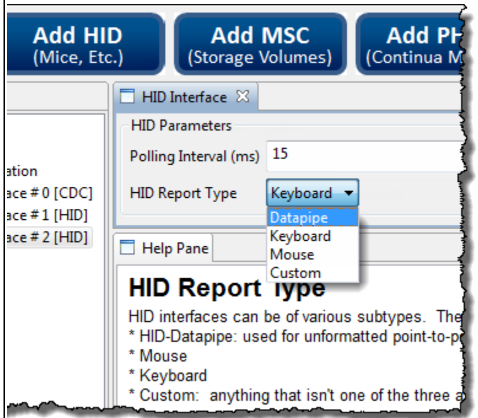
- The Tool is tightly integrated with the API
- Generates three source files that configure the rest of the stack
- Also generates the INF file (for CDC on Windows)



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)

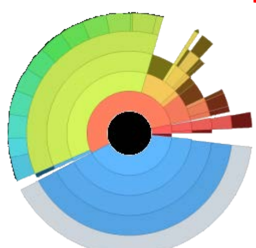


Datapipe mode allows the benefits of HID without some of its downsides

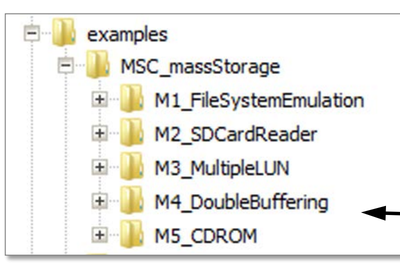
- Silent loading on the host
- Avoids USB's complex HID report structures
- Enables a unique value tradeoff

- 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)



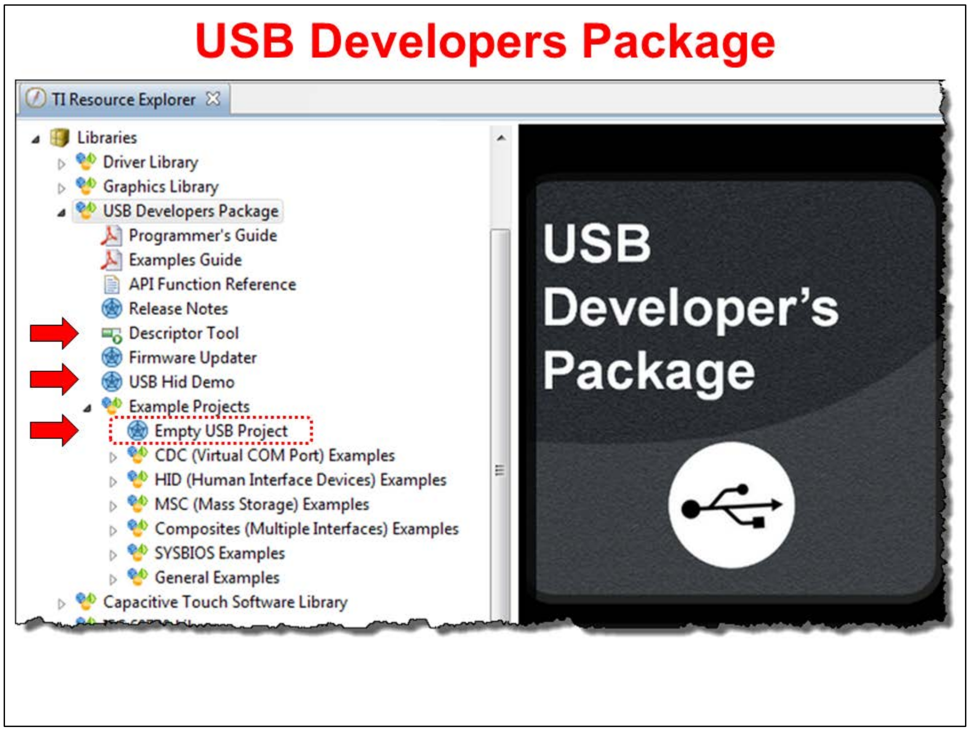
- Allows easy creation of a USB storage device
- No RTOS required
  - But can easily be ported to one
  - TI-RTOS (coming soon for MSP430) will provide a port with examples
- USB Developers Package includes a port of the open-source FAT file system (FatFS)
  - FatFS is provided as an example
  - USB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

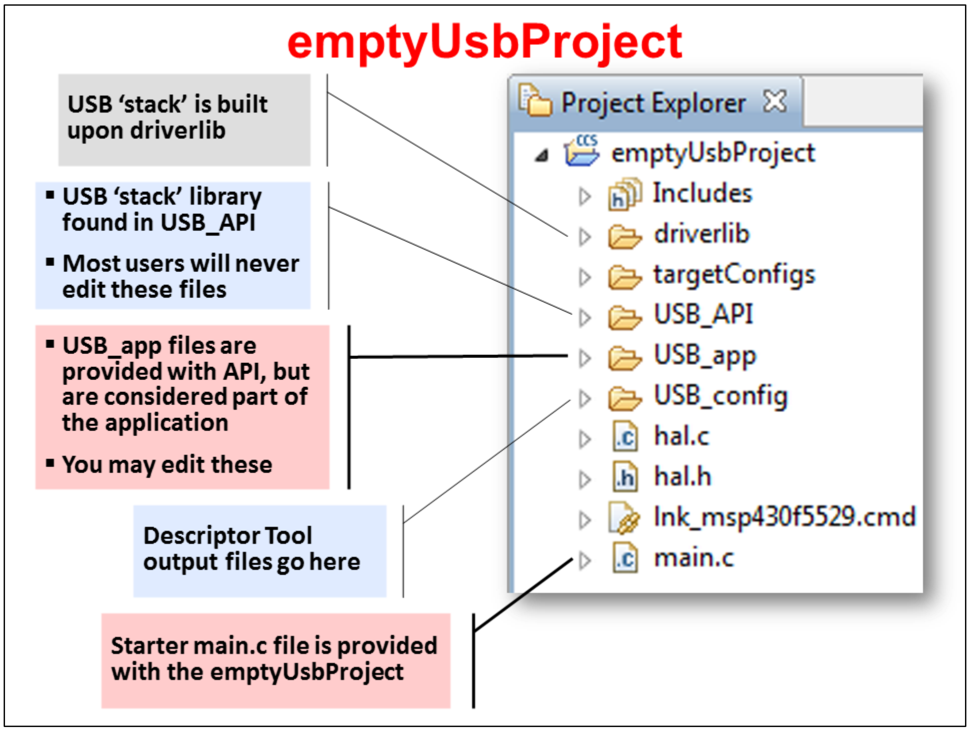


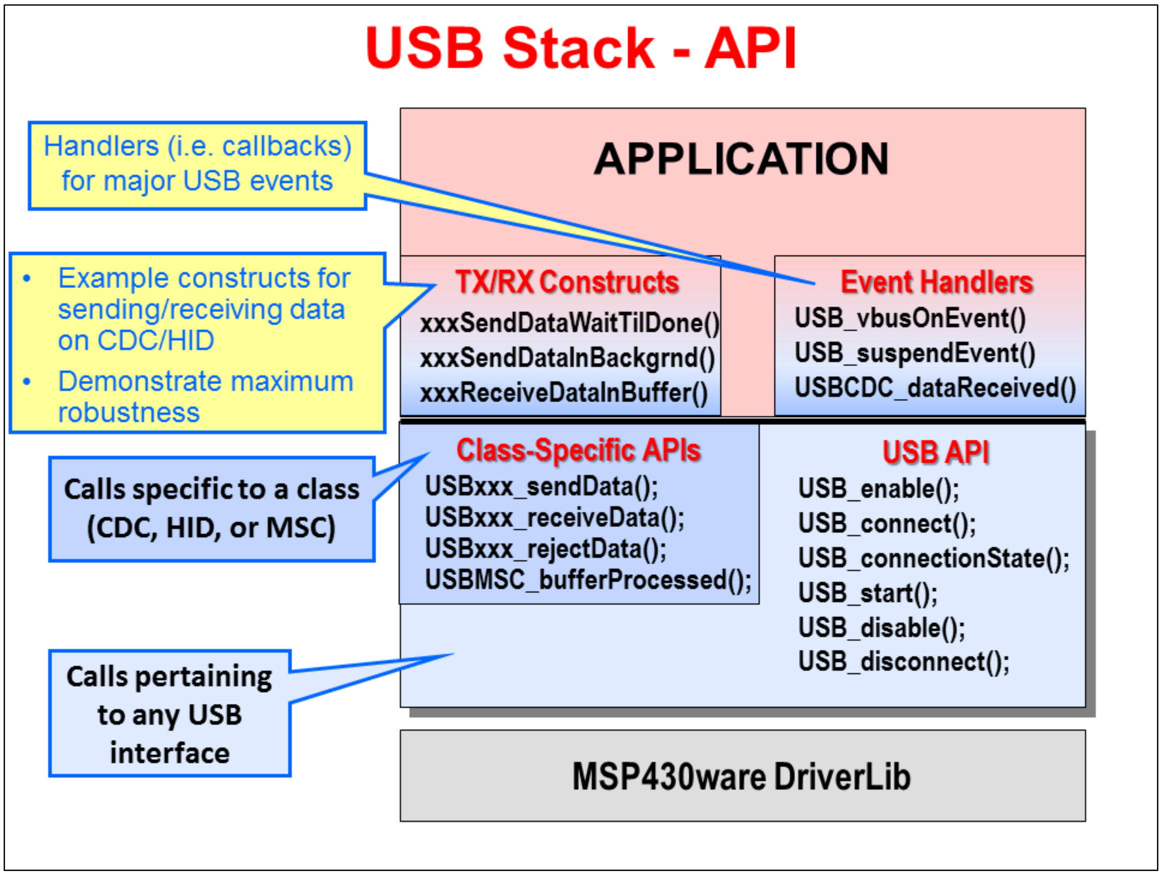
MSC will be covered in more detail in an new chapter under development

	CDC	HID	MSC
Host Interface	COM Port	HID device	Storage Volume
Host Loading	User Intervention (user loads .inf file)	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	▪ Familiar to user ▪ Bulk transport ▪ Common host apps	▪ Silent loading ▪ Interrupt xfers ▪ Mouse/Keybd	▪ Familiar to user ▪ Allows storage of data using filesys

Quick Overview of MSP430's USB Stack






USB Stack - API
APPLICATION
Handlers (i.e. callbacks) for major USB events
Example constructs for sending/receiving data on CDC/HID
Demonstrate maximum robustness
TX/RX Constructs
xxxSendDataWaitTilDone()
xxxSendDataInBackgrnd()
xxxReceiveDataInBuffer()
Event Handlers
USB_vbusOnEvent()
USB_suspendEvent()
USBCDC_dataReceived()
Calls specific to a class (CDC, HID, or MSC)
Class-Specific APIs
USBxxx_sendData();
USBxxx_receiveData();
USBxxx_rejectData();
USBMSC_bufferProcessed();
USB API
USB_enable();
USB_connect();
USB_connectionState();
USB_start();
USB_disable();
USB_disconnect();
Calls pertaining to any USB interface
MSP430ware DriverLib

Notes:

ABC's of USB

ABC's of USB Implementation

Transfer Basics

You can divide USB communications into two categories: communications used in enumerating the device and communications used by the application 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

Six States of Connection

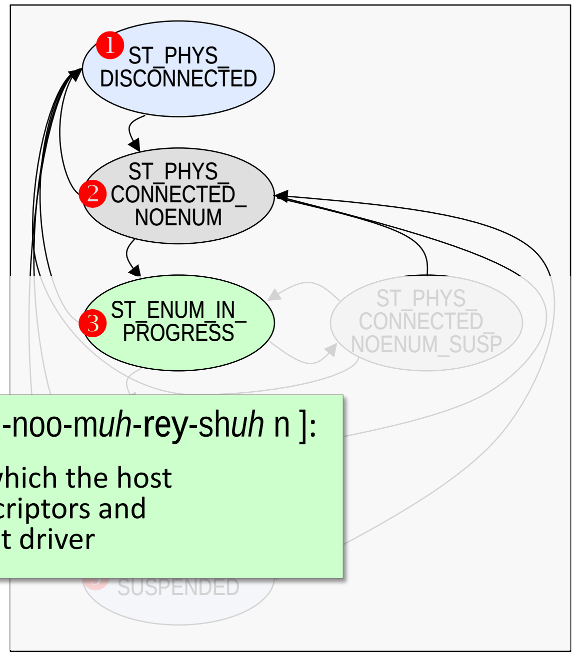
1. **USB is disconnected**

   → **User plugs in device & VBUS (power) appears**

2. **USB Connected, not enumerated**
   ☞ **handleVbusOnEvent()**

   → **App calls USB_Setup(), which pulls PUR up**

3. **Enumeration in Progress**



Enumeration [ih-noo-*muh*-**rey**-*shuh* n]:

the process in which the host obtains the descriptors and loads the correct driver

Six States of Connection

1. **USB is disconnected**

   → **User plugs in device & VBUS (power) appears**

2. **USB Connected, not enumerated**
   ☞ **handleVbusOnEvent()**

   → **App calls USB_Setup(), which pulls PUR up**

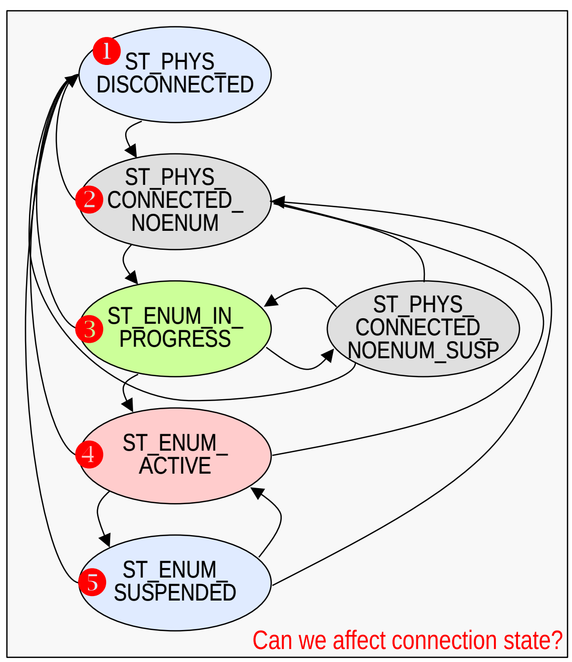
3. **Enumeration in Progress**

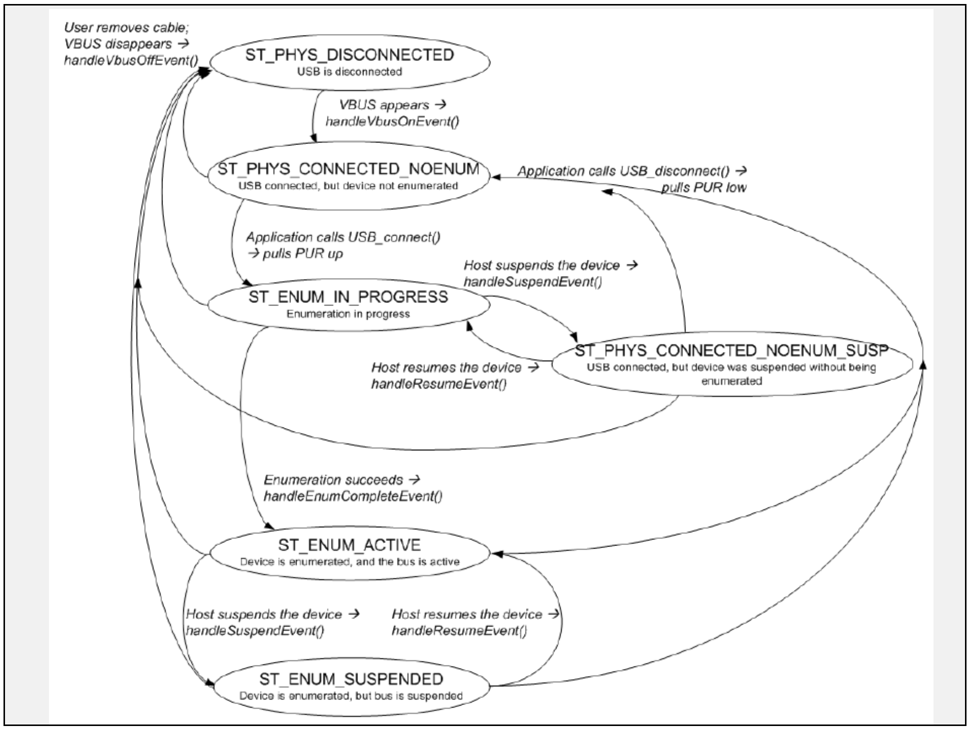
   → **Enumeration succeeds**

4. **Device is enumerated, bus active**
   ☞ **handleEnumCompleteEvent()**

   → **Host Suspends device/bus**

5. **Device is enum, but suspended**
   ☞ **handleSuspendEvent()**

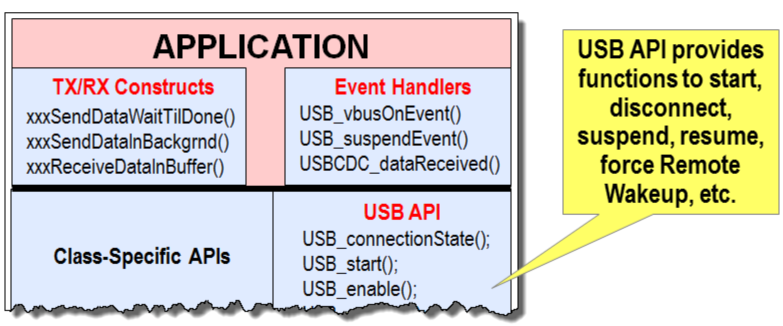




How Can I Modify Connection State?

- 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:

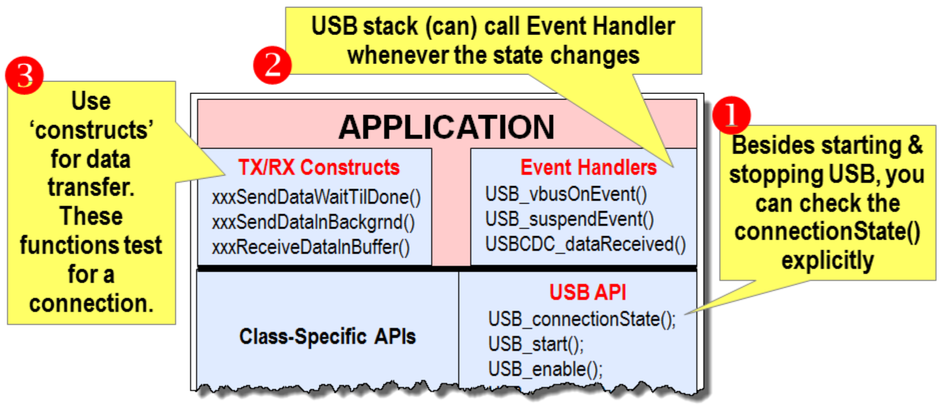
  `USB_setup()` - To start the USB stack running
- Additionally, you can elect to disconnect from the USB bus



C. Managing my App & Transferring Data

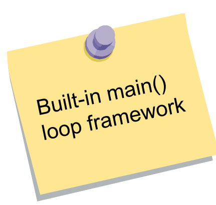
Respond to Connection State (as needed)

- **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:**



❶ 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:;
     }
  }
```

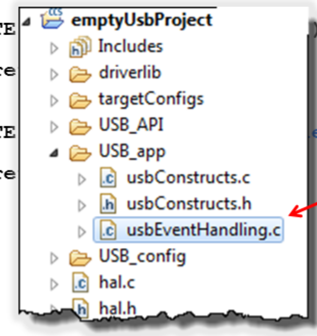


These three states are where the application spends most of its time (ST_USB_DISCONNECTED, ST_ENUM_ACTIVE, ST_ENUM_SUSPENDED)

- **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**

❷ Respond to 'Stack' Events

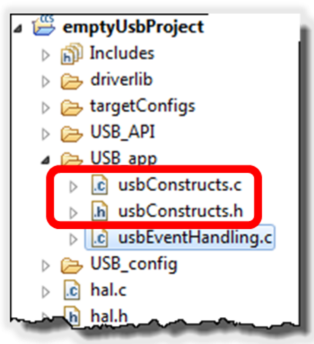
```
BYTE USB_handleVbusOnEvent()   {
  if (USB_enable() == kUSB_succeed)   //Connect when VBUS appears
  {
     USB_start();
  }
  return FALSE;
}

BYTE USB_handleSuspendEvent()
{
  return TRUE;
}

BYTE
{
  re
}

BYTE                              eted()
{
  re
}
```



- 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

❸ Construct Functions

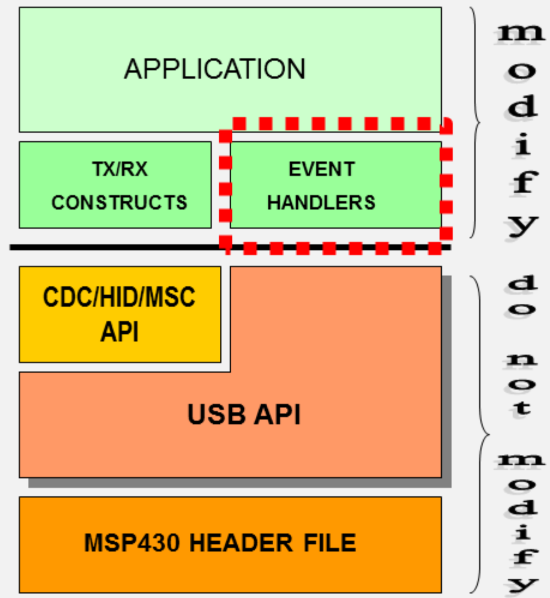
```
// From Example: C0_SimpleSend
   convertTimeBinToASCII(timeStr);

    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



TEXAS INSTRUMENTS

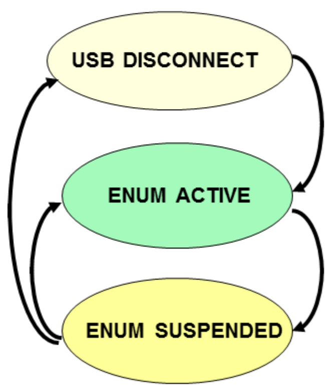
Final Thoughts

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**

Texas Instruments

Lab 10 – Using USB Devices

Lab 10 – USB Devices

- **Lab 10a – 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 10b – CDC LED On/Off Toggle**
  - Similar to Lab10a, but using CDC class to transfer the data
  - Host-side uses CCS serial Terminal (or Putty)
- **Lab 10c – 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 10d – Send Pushbutton State to Host**
  - Starts by importing the Empty USB Example
  - You add code to read the state of the pushbutton and send it to the host (via HID)
  - Read data on host with serial terminal

Lab Topics

USB Devices **10-37**

Lab 10 – Using USB Devices *10-39*

Lab 10a – LED On/Off HID Example *10-41*

Lab 10b – LED On/Off CDC Example *10-44*

Play with the demo 10-47

Lab 10c – CDC 'Simple Send' Example *10-49*

Lab 10d – Creating a CDC Push Button App *10-51*

Import Empty USB Project Steps 10-51

Use the Descriptor Tool 10-52

Add 'Custom' Code to Project 10-55

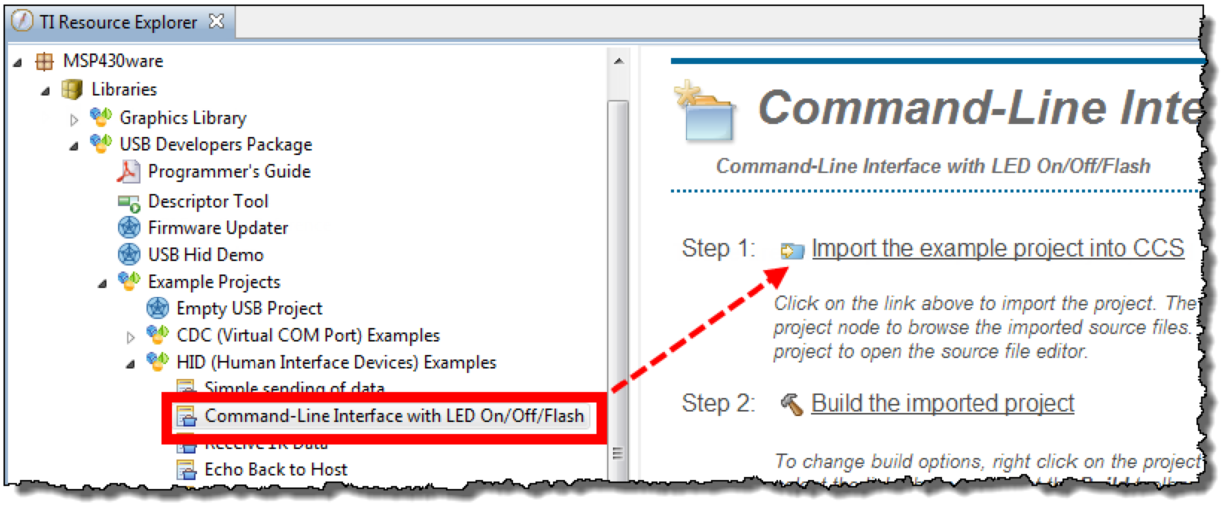
Lab 10a – LED On/Off HID Example

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`*



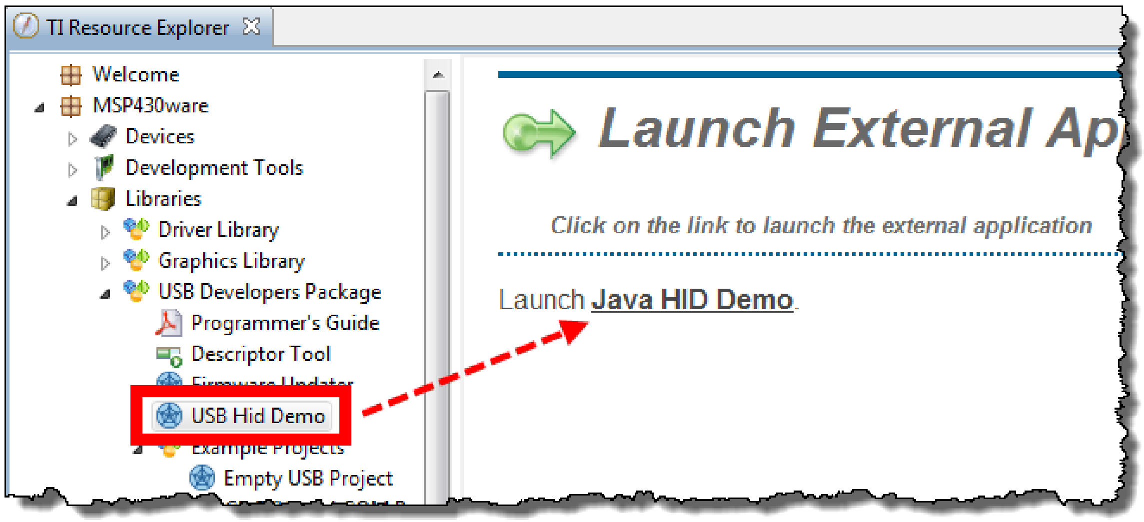
2. Build the project.

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

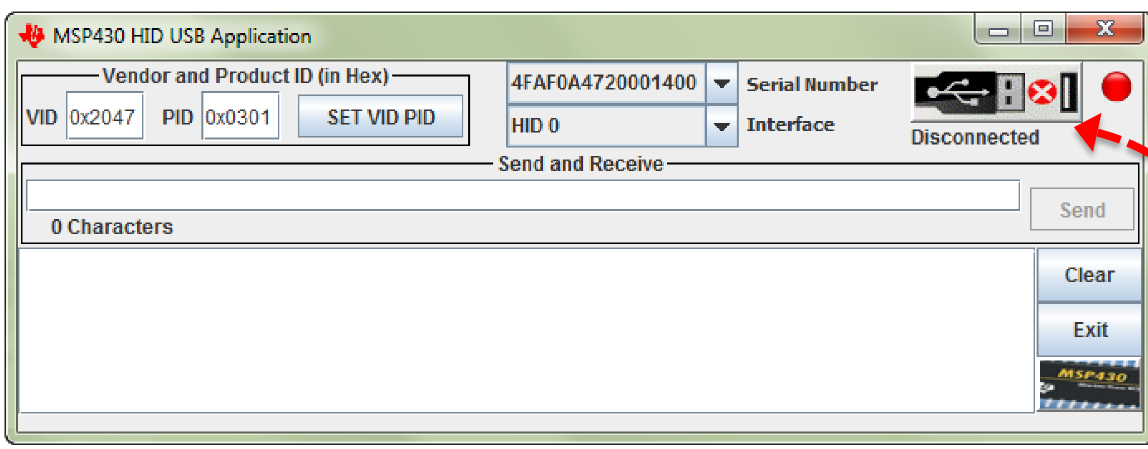
At this point, the MSP430 should start running the USB application. You may see Windows enumerate the USB device (in this case, your Launchpad); this usually appears as a popup message from the system tray saying that a USB device (“USB input device”) was enumerated.

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.

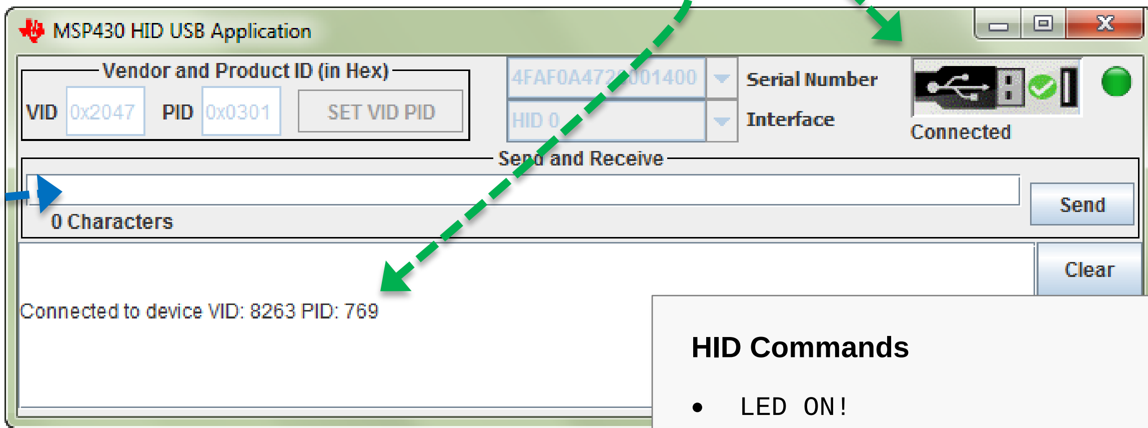
5. **Switch back to the USB HID Demo application.**

   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**

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.

8. **In the HID USB application, disconnect from the USB device; then close the application.**

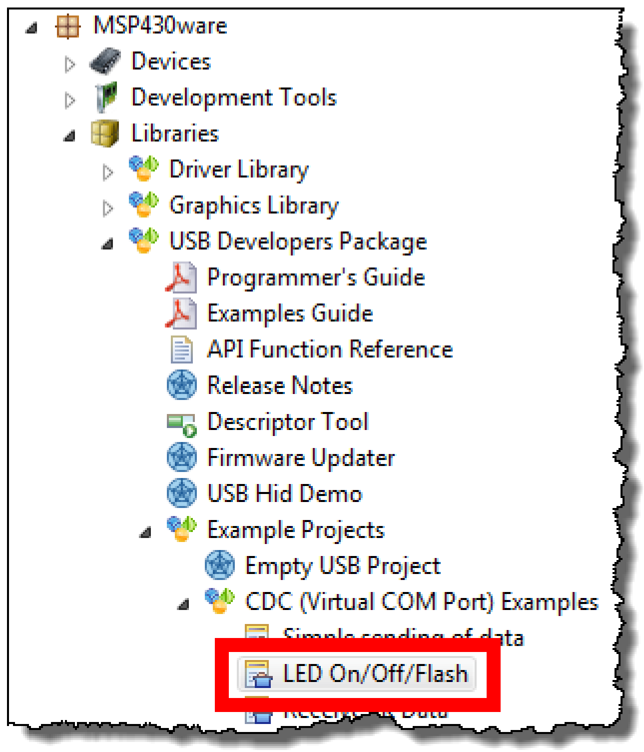
9. **Switch back to CCS and *Terminate* the debugger and close the project.**

Lab 10b – 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.

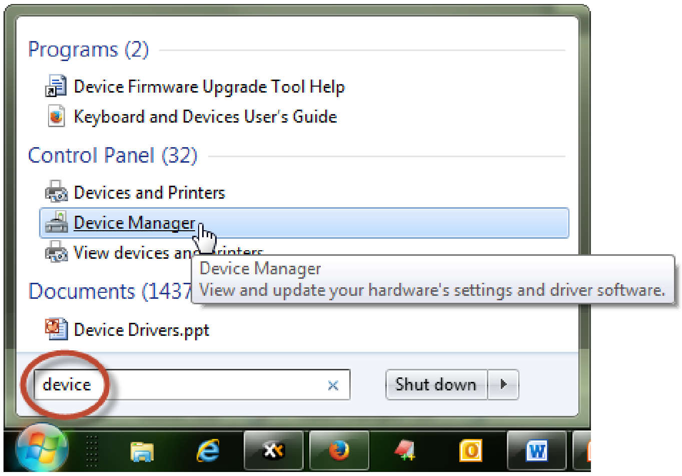
12. Run the program.

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 is to start the device manager is to type “Device” into the Start menu:

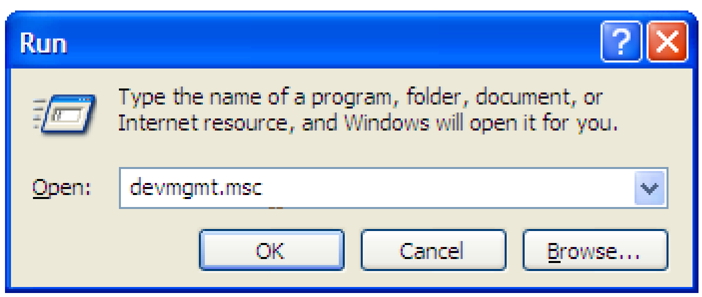


In most versions of Windows, such as Windows XP, you can also run the following program from a command line to start the Device Manager:

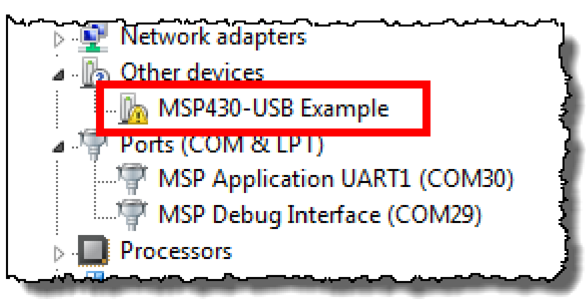
```
devmgmt.msc
```

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.

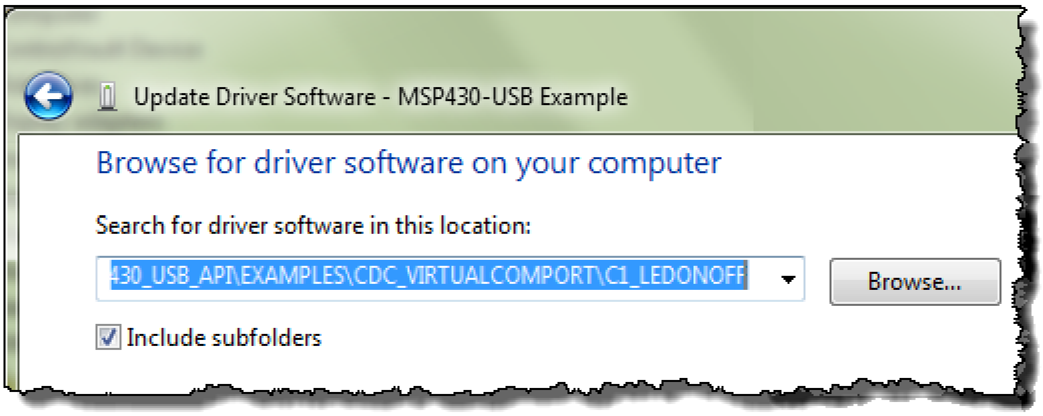
For Windows 7, the steps include:

`Right-click on the driver → Update Driver Software…`

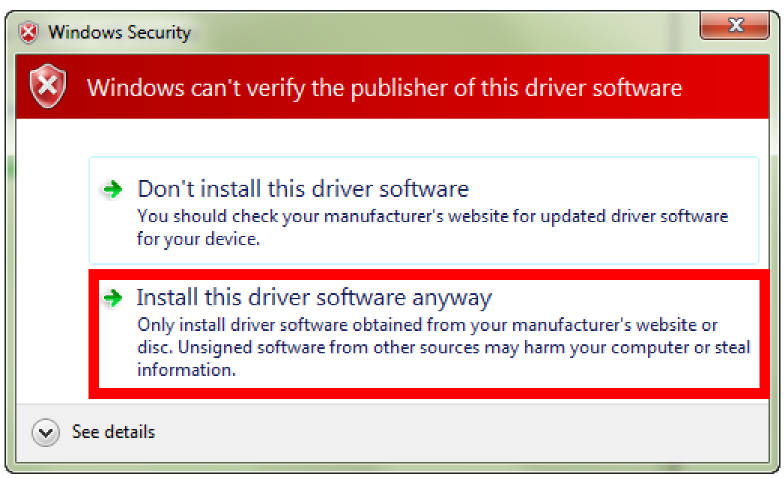
`Click Browse my computer for driver software`

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

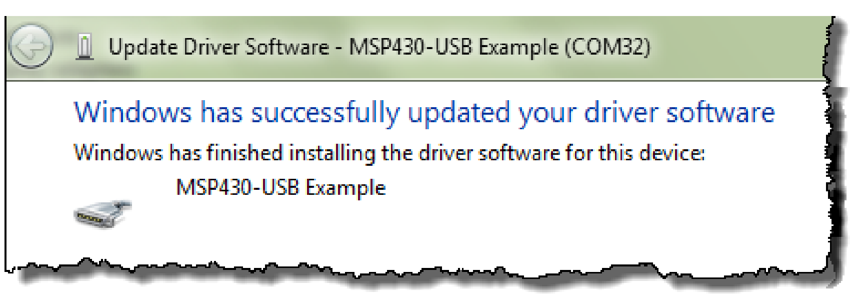
```
C:\TI\MSP430\MSP430USBDEVELOPERSPACKAGE_4_00_02\MSP430_USB_SOFTWA
RE\MSP430_USB_API\EXAMPLES\CDC_VIRTUALCOMPORT\C1_LEDONOFF
```



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



When complete you should see:

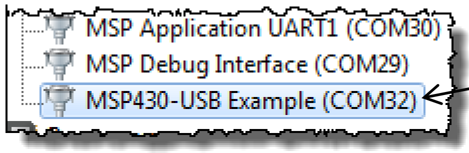


Note: The steps to install the USB CDC driver are also documented in the:

`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:



Looking at our computer, we would need to use COM32

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

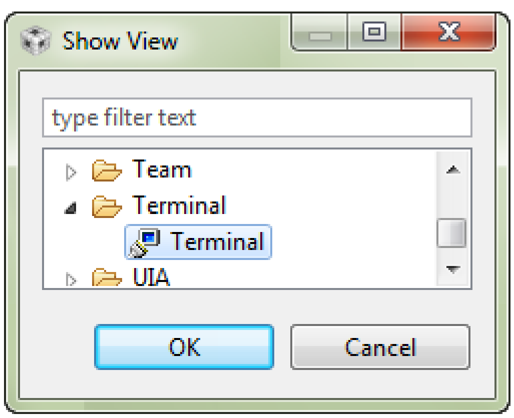
Before we can communicate with the device, though, we also need to open a serial terminal.

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

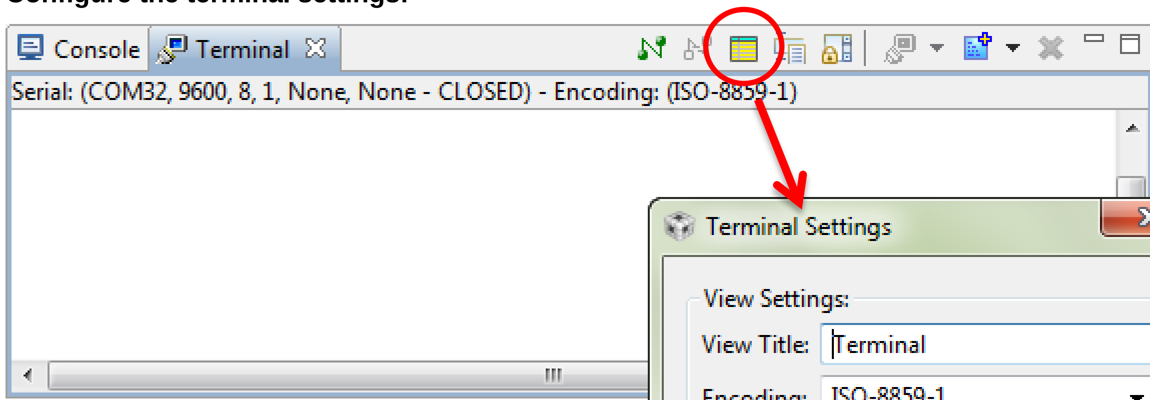
Putty and Tera Term are common favorites, but we'll provide directions for using the Terminal built into CCS.

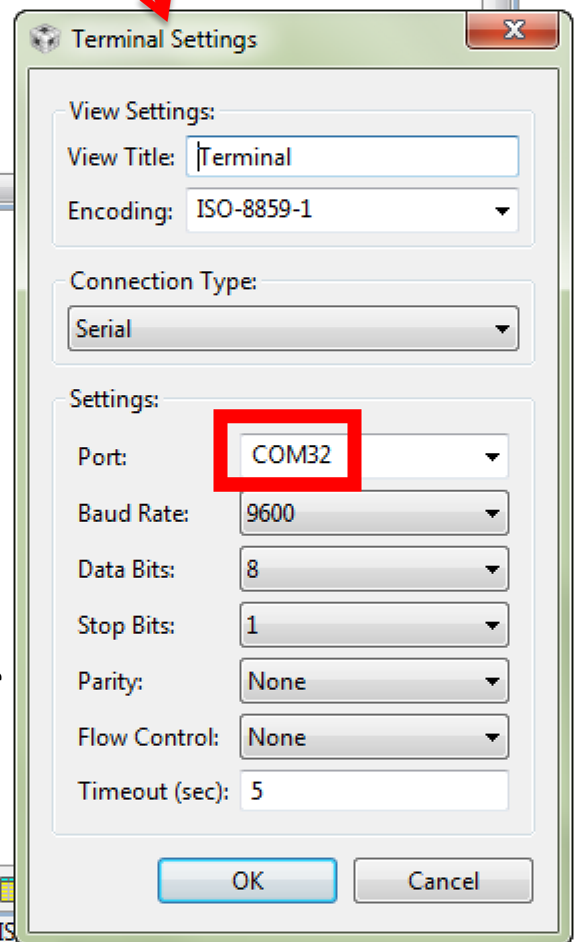
a) Open the Terminal window.

`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”.

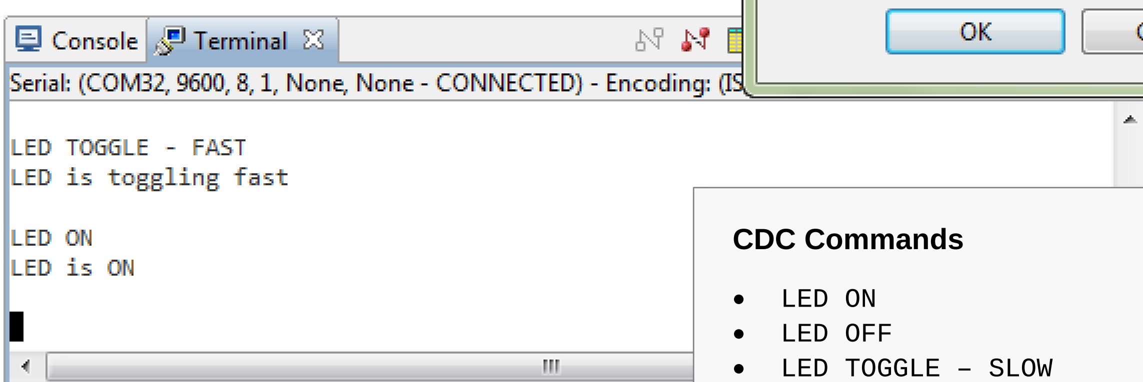
If the terminal does not connect, then check:

– *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.

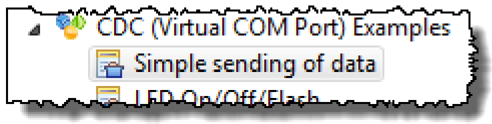
18. **When done experimenting…**

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

Lab 10c – 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.

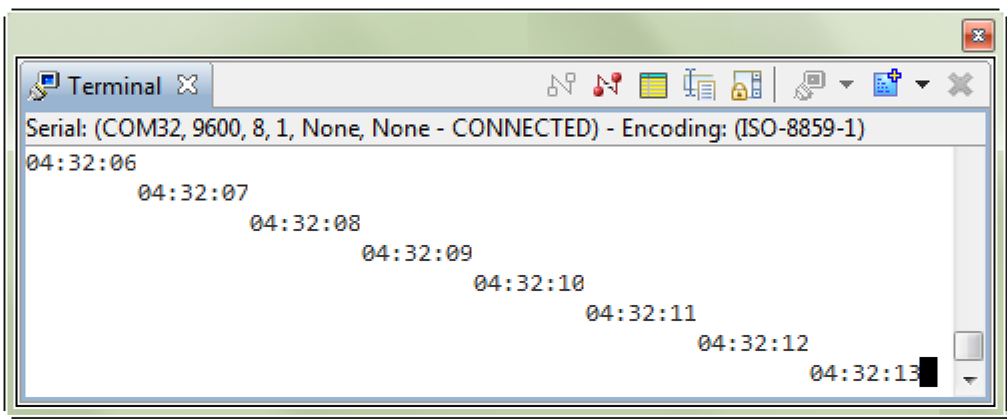
21. Start the program.

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 an 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 10d – Creating a CDC Push Button App

We have experimented with three example USB applications. It's finally time to build one from "scratch". Well, not really from scratch, since we can start with the "Empty USB Example".

The goal of our application is to send the state of the Launchpad button to the PC via USB – using the HID Datapipe interface. Thus, we'll use a HID class driver. This application will borrow from a number of programs we've already written:

GPIO – We will read the push button and light the LED when it is pushed. Also, we'll send "DOWN" when it's down and "UP" when it's up.

Timer – We'll use a timer to generate an interrupt every second. In the Timer ISR we'll set a flag. When the flag is TRUE, we'll read the button and send the proper string to the host.

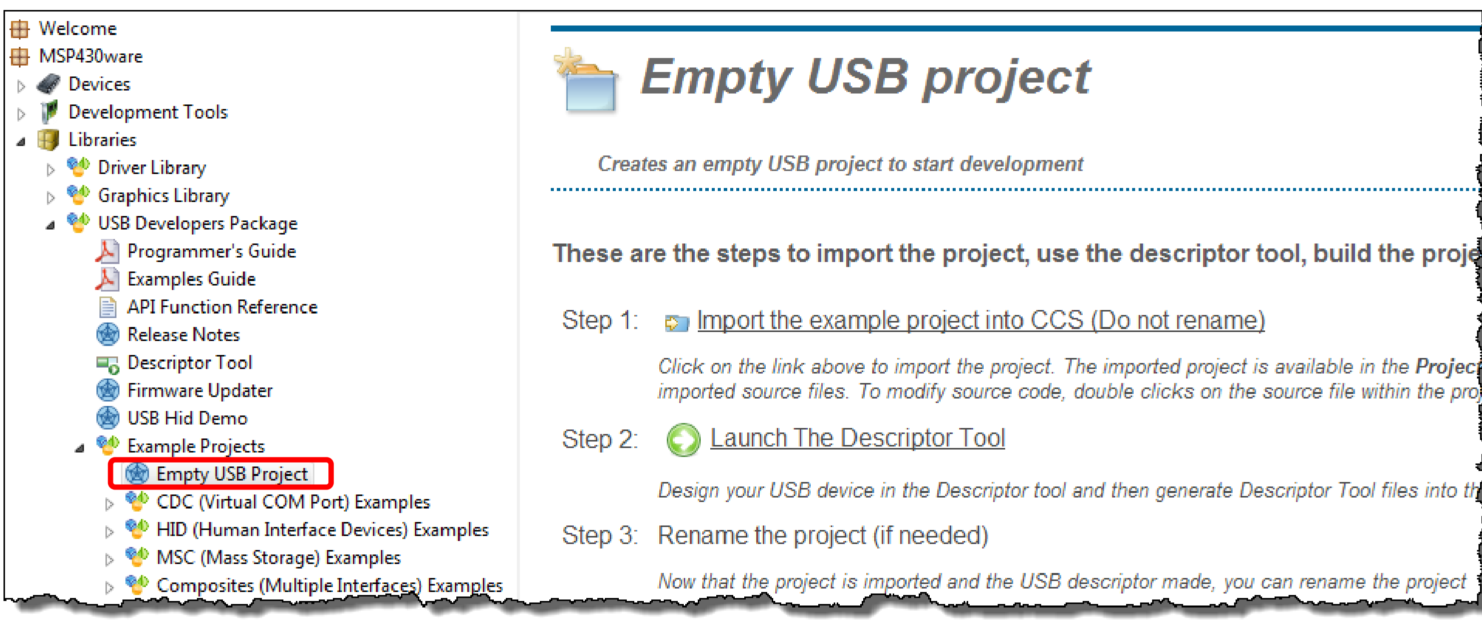
HID Simple Send Example – we'll borrow a bit of code from the HID example we just ran to 'package' up our string and send it via USB to the host.

Finally, we're going to start by following the first 3 steps provided in TI Resource Explorer for the **Empty USB Example**.

Import Empty USB Project Steps

1. **Import the Empty USB Project.**

   As it states in the Resource Explorer, DO NOT RENAME the project (yet).



Use the Descriptor Tool

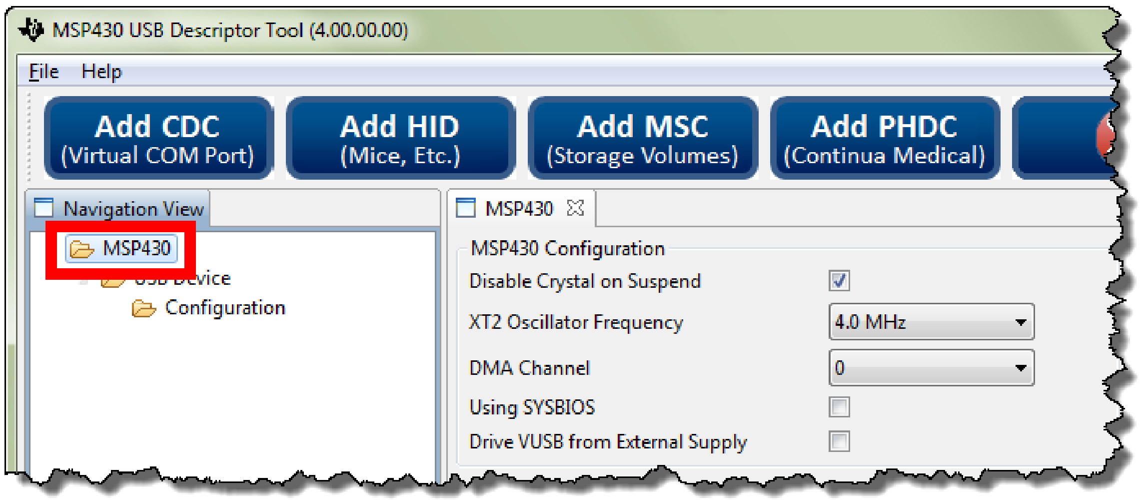
2. Launch the Descriptor Tool.

Just as the Resource Explorer directs us, launch the Descriptor Tool. The easiest way to do this is to click the link as shown above.

3. Generate descriptor files using the Descriptor Tool.

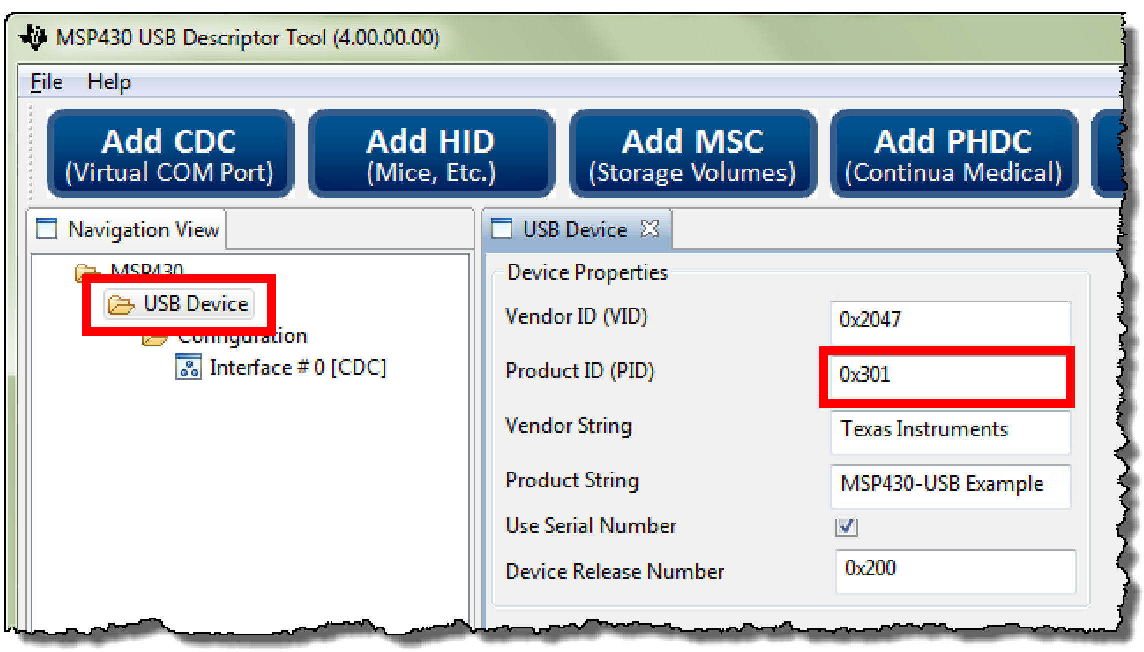
We will take a quick look at the organization levels in the tool. In most cases, we will use the tools defaults.

a) MSP430 level … use the defaults.



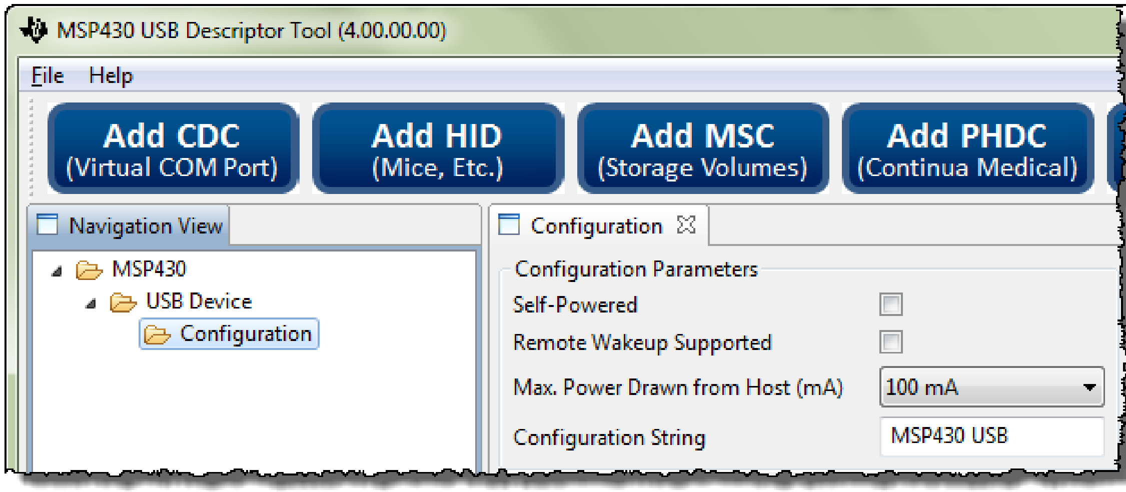
b) USB Device … MSP430-Button Example

We suggest changing the Product String – so it'll be easier to see that it is different than previous examples. Also, we suggest changing the PID (we picked '301' arbitrarily). For a real design, you might end up purchasing the VID/PID (or obtain a free PID from TI).



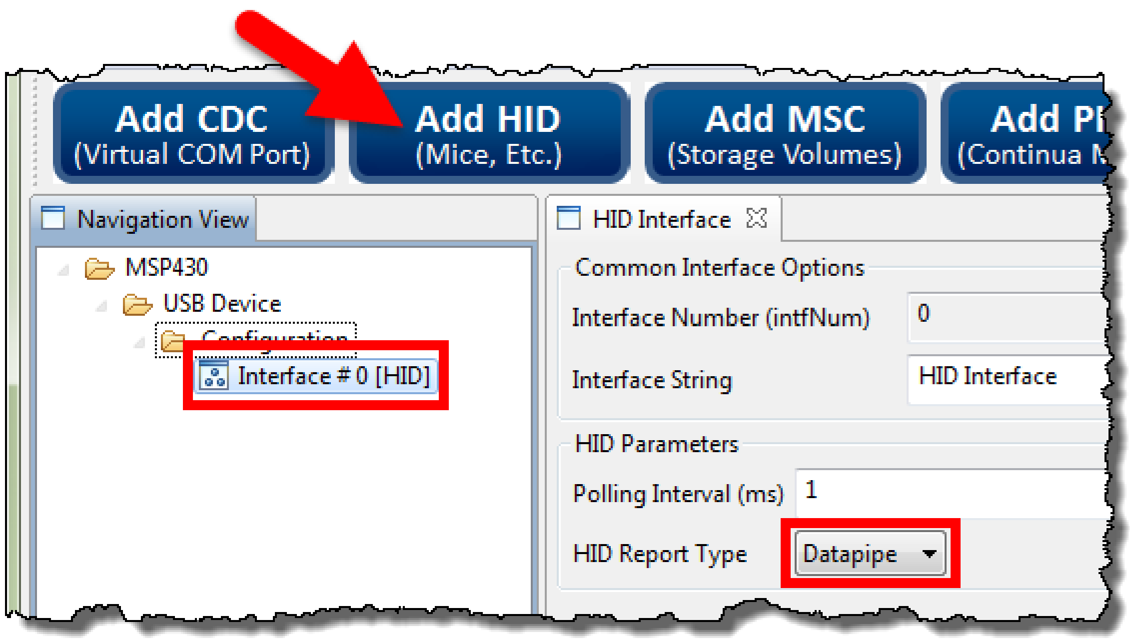
c) Configuration

Nothing to do on the configuration screen.



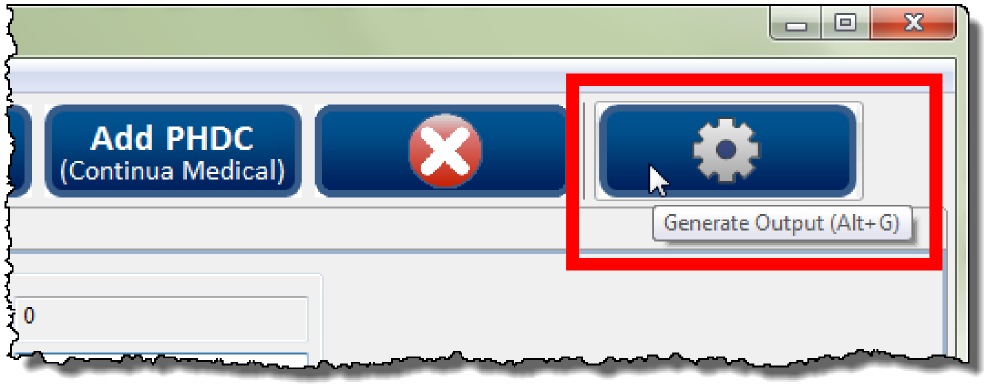
d) Add HID Interface

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



e) **Click the button to generate the descriptor files.**

Notice they get written to your empty project. (This is the reason we were asked not to change the name until after we had used the Descriptor Tool.)

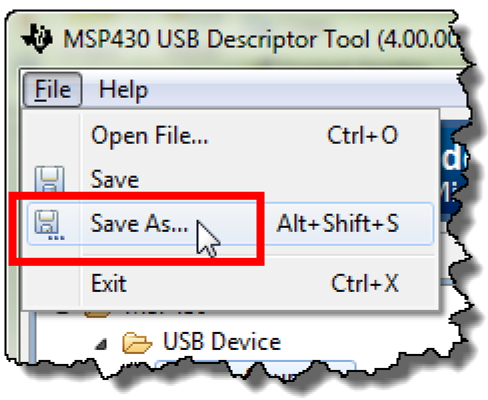


The files should be saved to our "empty" project … but if you're asked where to save them, choose the USB_config folder:

`C:\msp430_workshop\F5529_usb\workspace\emptyUsbProject\USB_config\`

f) **Save the Descriptor Tool settings.**

While not required, this is handy if you want to open the tool and view the settings at some later point in time. Notice that 'Save' puts the resulting .dat file into the same folder as our descriptor files.



Save to your emptyProject USB_config folder. This is a pretty good place for it, since this is where all of the descriptor files it generates are placed. For example:

`C:\msp430_workshop\F5529_usb\workspace\emptyUsbProject\USB_config\`

g) **You can close the Descriptor Tool.**

4. **Rename the project to lab_10d_usb.**

As you can see, the reason they didn't want us to rename the project before now was that the descriptor tool generates files to the empty project.

5. **Build, just to make sure we're starting off with a 'clean' project.**

Add 'Custom' Code to Project

6. **Copy `myTimer.c` and `myTimer.h (and the readme file)` to the project folder.**

   We've already written the timer routine for you. (Look back to our Timer chapter if you want to know the details of how this code was developed.)

   `Right-click the project → Add Files…`

   Choose the three files from the location:

   `C:\msp430_workshop\F5529_usb\lab_10d_usb\`

7. **Open `main.c` and add a #include for the `myTimer.h`.**

   We suggest doing this somewhere below #include "`driverlib.h`".

8. **Add global variables.**

   These are used to capture (and send) the button up/down state.

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

9. **Add additional setup code.**

   We need to initialize an LED and pushbutton. We also need to call the initTimers() function that was just added to our project in a previous step.

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

10. **Modify the low-power state of the program.**

    Search down toward the end of main() until you find the intrinsic that sets the program into low-power mode. Rather than using LPM3, we want to switch this to LPM0.

```
//  _bis_SR_register(LPM3_bits + GIE);
    __bis_SR_register(LPM0_bits + GIE);
```

Notes:

11. Add code to ST_ENUM_ACTIVE state.

The active state is where we want to put our communication code. (It only makes sense to that we send data to the host when we're actively connected.

When connected, we will read the pin, set the Launchpad's LED and then construct a string to send to the host. Finally, we send the data to the host in the background; that is, we won't wait for a response – although we do set a timeout in our code below.

Note that it's the timer that wakes us up every second to check the state – and if the USB is in the connected state, to run through the routine below.

```
// If USB is present, sent the button state to host. Flag set every sec
if (bSend)
{
    bSend = FALSE;

    usiButton1 = GPIO_getInputPinValue ( GPIO_PORT_P2, GPIO_PIN1 );

    if ( usiButton1 == GPIO_INPUT_PIN_LOW ) {
        // If button is down, turn on LED
        GPIO_setOutputHighOnPin( GPIO_PORT_P4, GPIO_PIN7 );
        pbStr[0] = 'D';
        pbStr[1] = 'O';
        pbStr[2] = 'W';
        pbStr[3] = 'N';
        pbStr[4] = '\n';
    }
    else {
        // If button is up, turn off LED
        GPIO_setOutputLowOnPin( GPIO_PORT_P4, GPIO_PIN7 );
        pbStr[0] = 'U';
        pbStr[1] = 'P';
        pbStr[2] = ' ';
        pbStr[3] = ' ';
        pbStr[4] = '\n';
    }

    // This function begins the USB send operation, and immediately
    // returns, while the sending happens in the background.
    // Send pbStr, 5 bytes, to intf #0 (which is enumerated as a
    // HID port).  1000 retries.  (Retries will be attempted if the
    // previous send hasn't completed yet).  If the bus isn't present,
    // it simply returns and does nothing.
    if (cdcSendDataInBackground((BYTE*)pbStr, 5, HID0_INTFNUM, 1000))
    {
      _NOP();  // If it fails, it'll end up here.  Could happen if
               // the cable was detached after the connectionState()
    }           // check, or if somehow the retries failed
}
```

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

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

13. Build the program and launch debugger.

14. Start your program and open the USB HID demo tool.

You can either run the program from within the debugger – or – terminate the debugger and unplug and then plug the Launchpad back in. In either case, your USB program should be running.

We need to use the HID tool to view the communications coming from the Launchpad. As we mentioned earlier, it acts as a “terminal” for our HID Datapipe datastream.

If you cannot remember how to open it, please refer back to Step 4 on page 10-42.

Hint: You might have to set the PID depending upon the value you selected while using the Descriptor tool.

15. Verify your program works

Once the the driver is loaded and working properly, open your Terminal, making sure to use the proper comm port. *(As a reminder, all of these steps we discussed earlier in this chapter.)*

At this point:

- The Red LED should be blinking on/off.
- The Green LED should light when Button1 is pushed …
- … and the state of the button should be written to the HID Terminal.

Remember that the code only tests the button once per second. So, you will need to hold (or release) it for more than a second for it to take effect.

Using Energia (Arduino)

Introduction

This chapter of the MSP430 workshop explores Energia, the Arduino port for the Texas Instruments Launchpad kits.

After a quick definition and history of Arduino and Energia, we provide a quick introduction to Wiring – the language/library used by Arduino & Energia.

Most of the learning comes from using the Launchpad board along with the Energia IDE to light LED's, read switches and communicate with your PC via the serial connection.

Learning Objectives, Requirements, Prereq's

Prerequisites & Objectives

- Prerequisites
  - Basic knowledge of C language
  - Basic understanding of using a C library and header files
  - This chapter doesn't explain clock, interrupt, and GPIO features in detail; this is left to the other chapters in the MSP430 Design Workshop
- Requirements - Tools and Software
  - Hardware
    - *Windows 7 (or 8) PC with available USB port*
    - *MSP430F5529 or MSP430FR5969 Launchpad*
  - Software
    - *Energia Download*
    - *Launchpad drivers*
    - *(Optional) MSP430ware / Driverlib*

Already installed, if you have installed CCSv5 or 6

- Objectives
  - Define 'Arduino' and describe what is 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

Chapter Topics

Using Energia (Arduino)..11-1

What is Arduino...11-3

Energia...11-4

Programming Energia (and Arduino) 11-7
Programming with 'Wiring' 11-7
Wiring Language/Library Reference 11-8
How Does 'Wiring' Compare?..11-9
Hardware pinout...11-10

Energia IDE..11-12
Examples, Lots of Examples..11-13

Debugging Energia (Arduio) with CCSv6...11-13

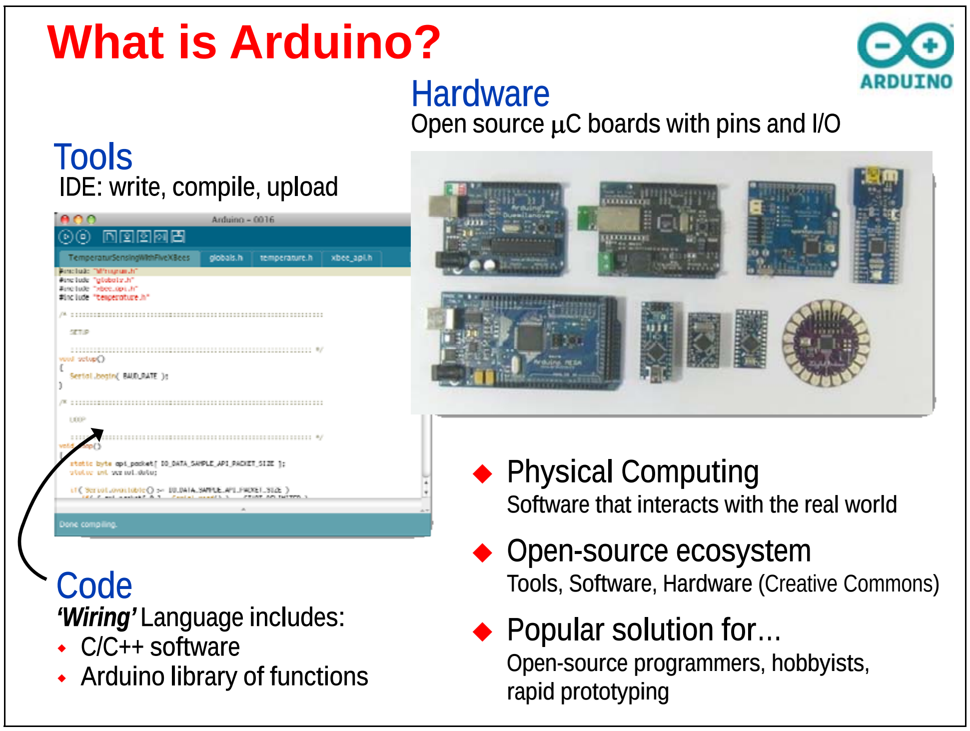
Energia/Arduino References...11-14

Lab 11 11-15

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]



- *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, Arudino is basically an ecosystem for easy, hardware-oriented, real-world programming. It combines the Tools, Software and Hardware for talking to the world.

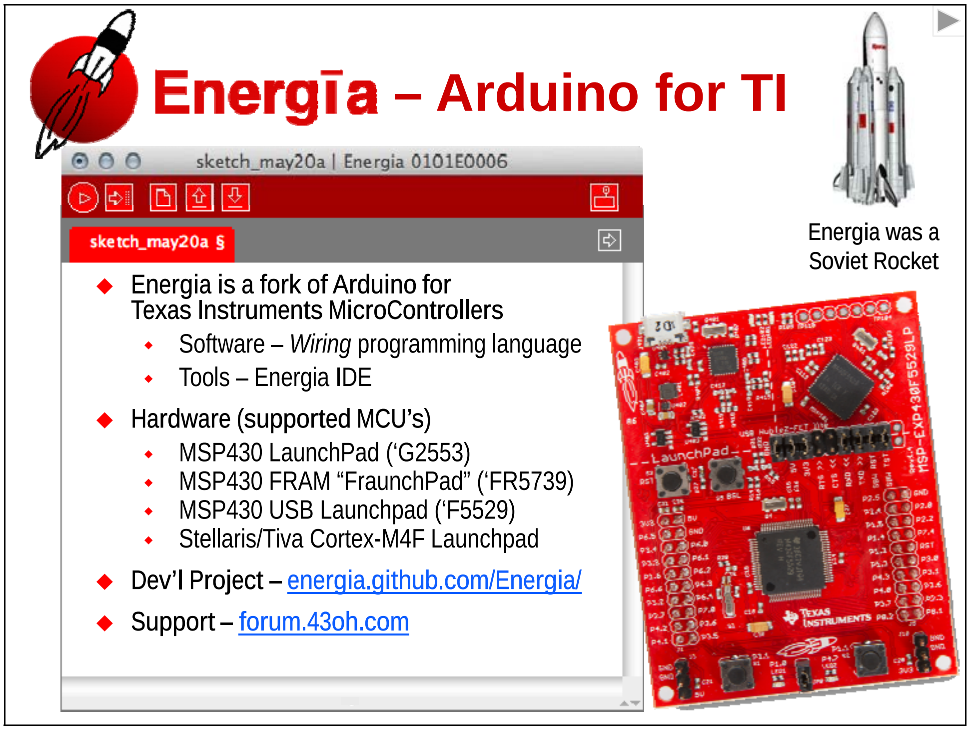
---

[1] http://www.wired.com/gadgetlab/2008/04/just-what-is-an/
[2] http://en.wikipedia.org/wiki/Wiring_%28development_platform%29

Energia

/enerˈgia/ ; 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]

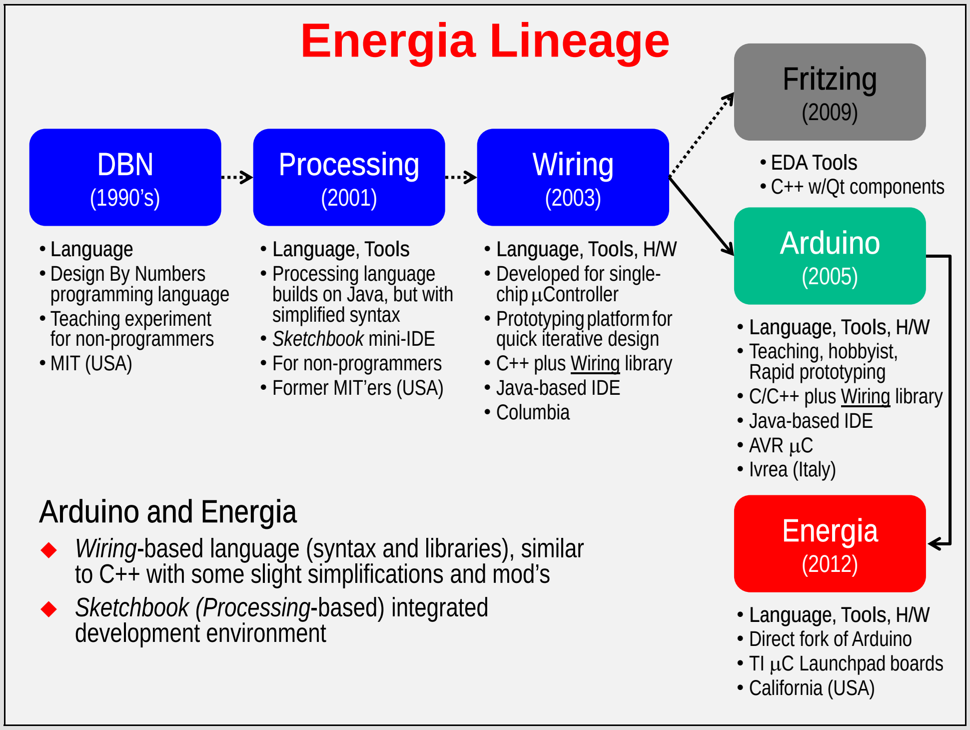


Energia is a rapid electronics prototyping platform for the Texas Instruments msp430 LaunchPad. Energia is based on Wiring and Arduino and uses the Processing IDE. It is a fork of the Arduino ecosystem, but centered around the popular TI microntrollers: MSP430 and ARM Cortex-M4F.

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)
6 http://en.wikipedia.org/wiki/Wiring_%28development_platform%29

Energia Lineage (cont'd)

Arduino[7] - 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.

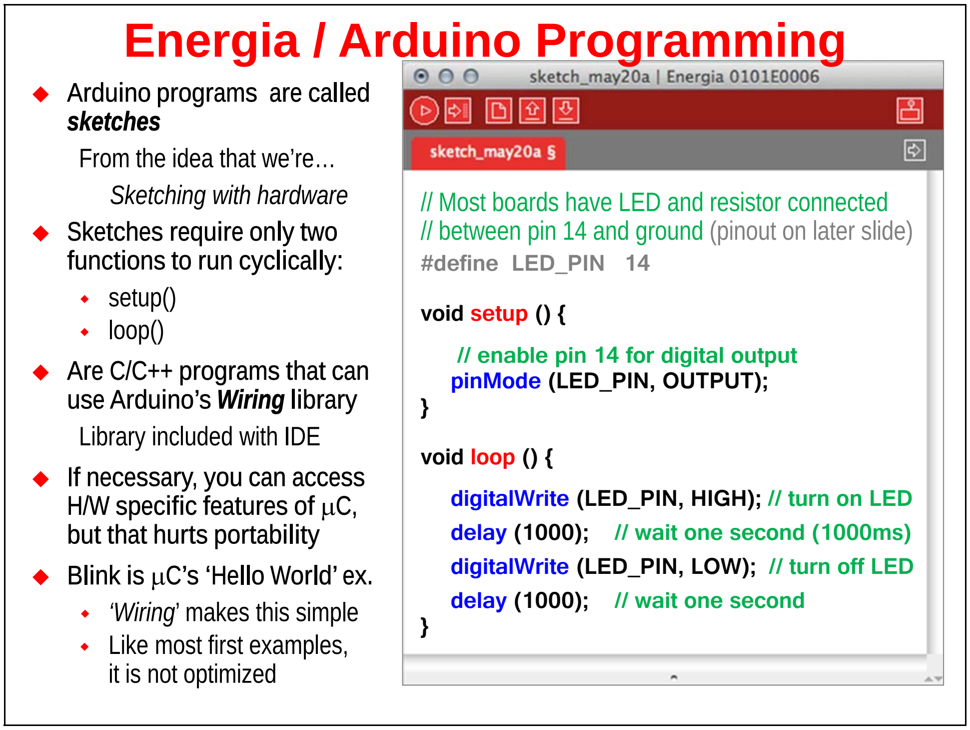
Fritzing (2009)[8] - An open-source initiative to support designers, artists, researchers and hobbyists to take the step from physical prototyping to actual product.

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

Programming Energia (and Arduino)

Programming with 'Wiring'



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:

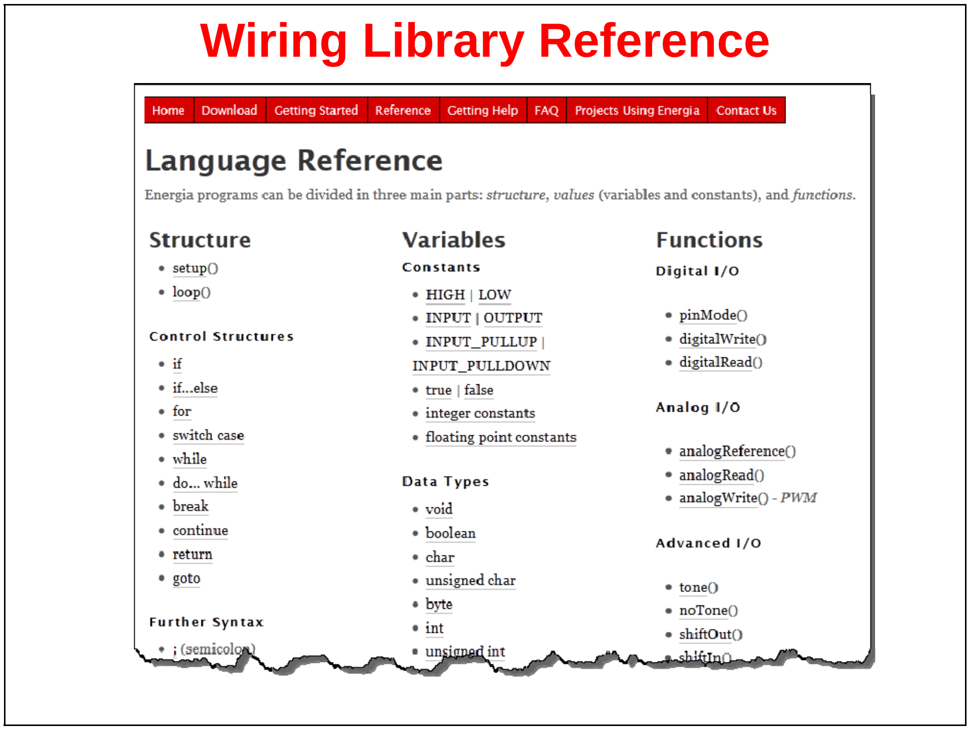
- A sketch only requires two functions:
  - **setup()** – a function run once at the start of a program which can be used to define initial environment settings
  - **loop()** – a function called repeatedly until the board is powered off
- Reading and Writing pins (i.e. General Purpose Input Output – 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?

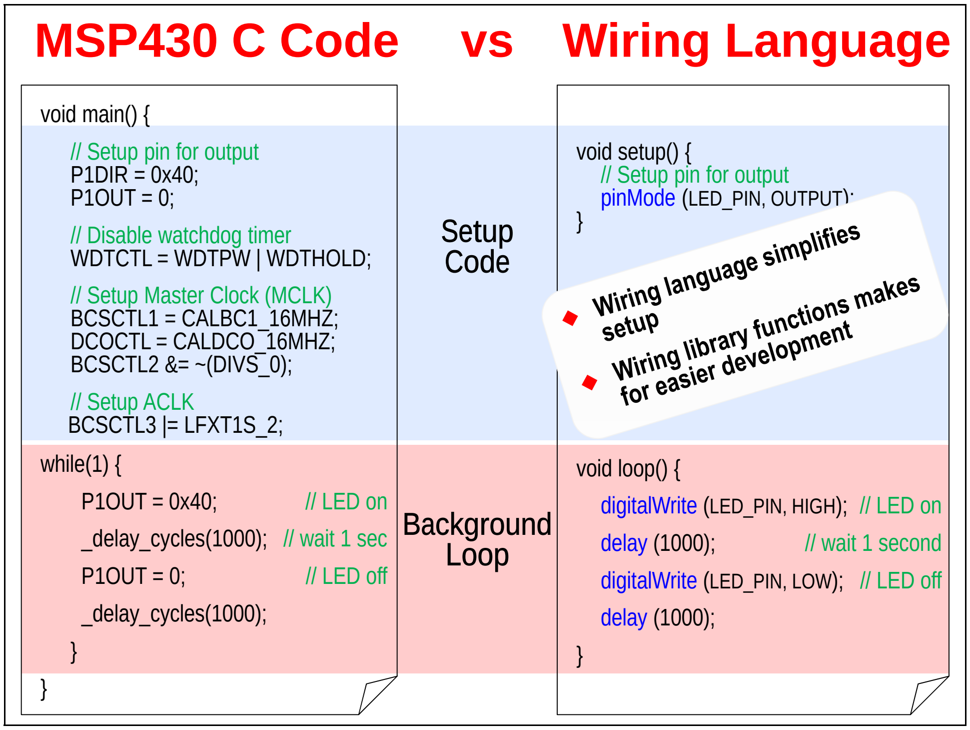
Arduino provides a language reference on their website. This defines the operators, controls, and functions needed for programming in Arduino (and Energia).[9] You will also find a similar HTML reference available in the Energia installation zip file.



9 http://arduino.cc/en/Reference/HomePage

How Does 'Wiring' Compare?

How does the *'Wiring'* language compare to standard C code?



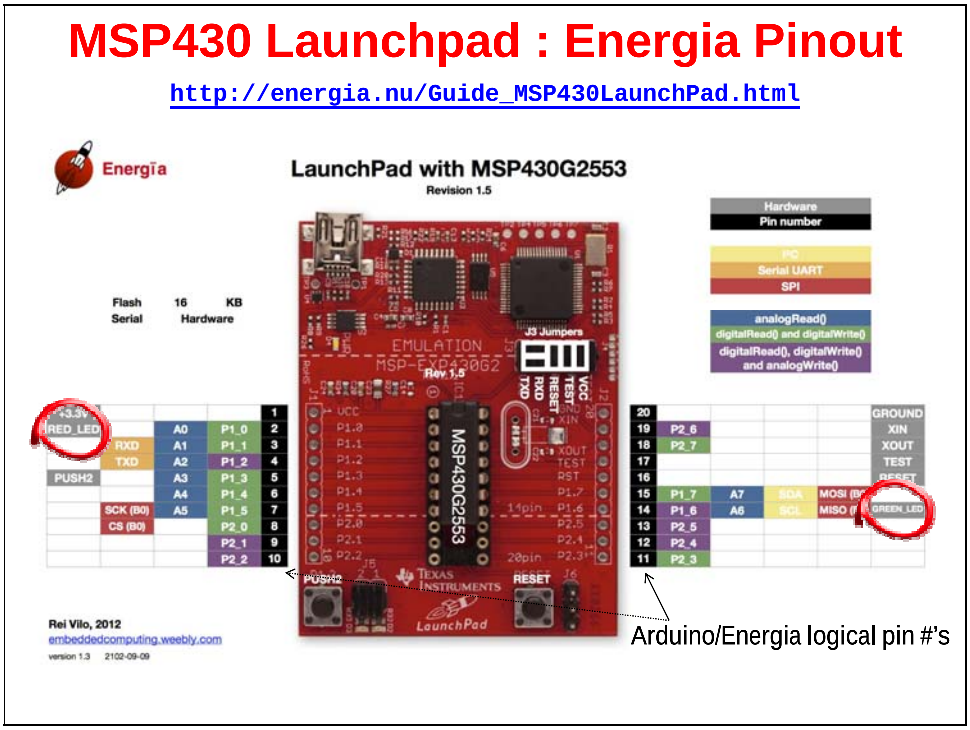
This comparison helps to demonstrate the simplicity of programming with Energia. As stated before, this can make for very effective rapid prototyping.

Later, during one of the lab exercises, we will examine some of the underpinings 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 TEMPSENSOR. 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, or in the files installed with Energia.

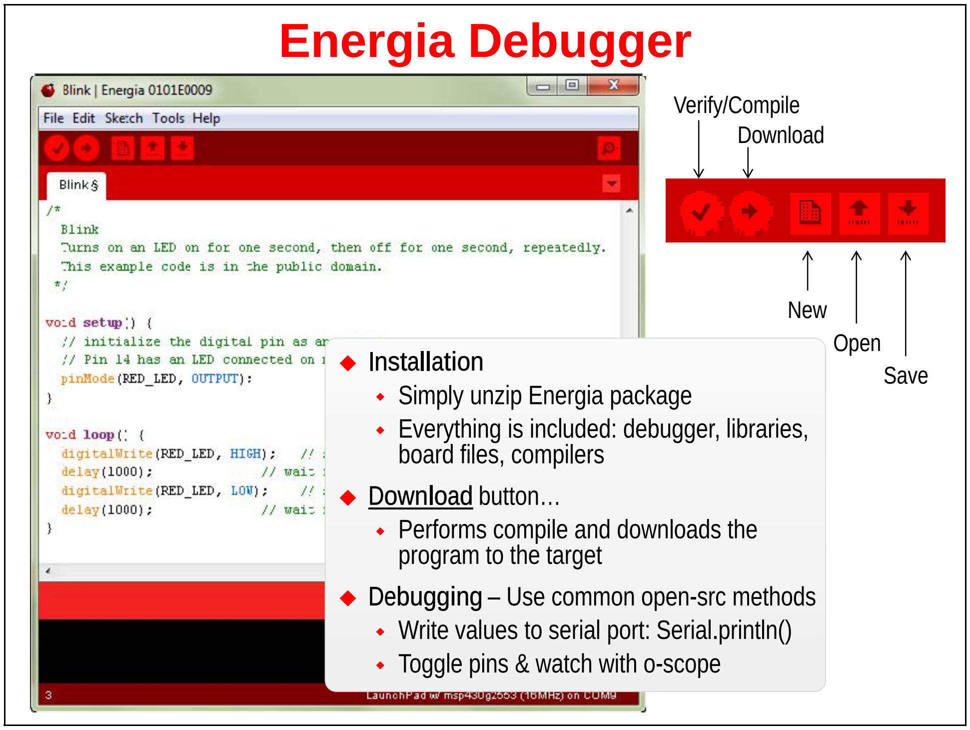
Sidebar

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 (Wndows, Mac, Linux).



Installation of the tools couldn't be much simplier – 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.)

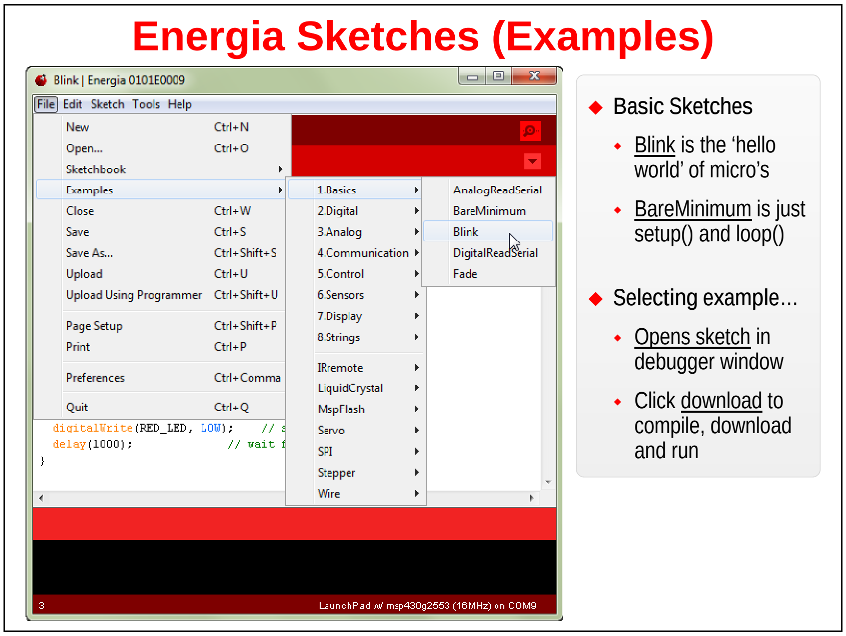
Editing code is straightforward. Syntax highlighting, as well as brace matching help to minimize errors.

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

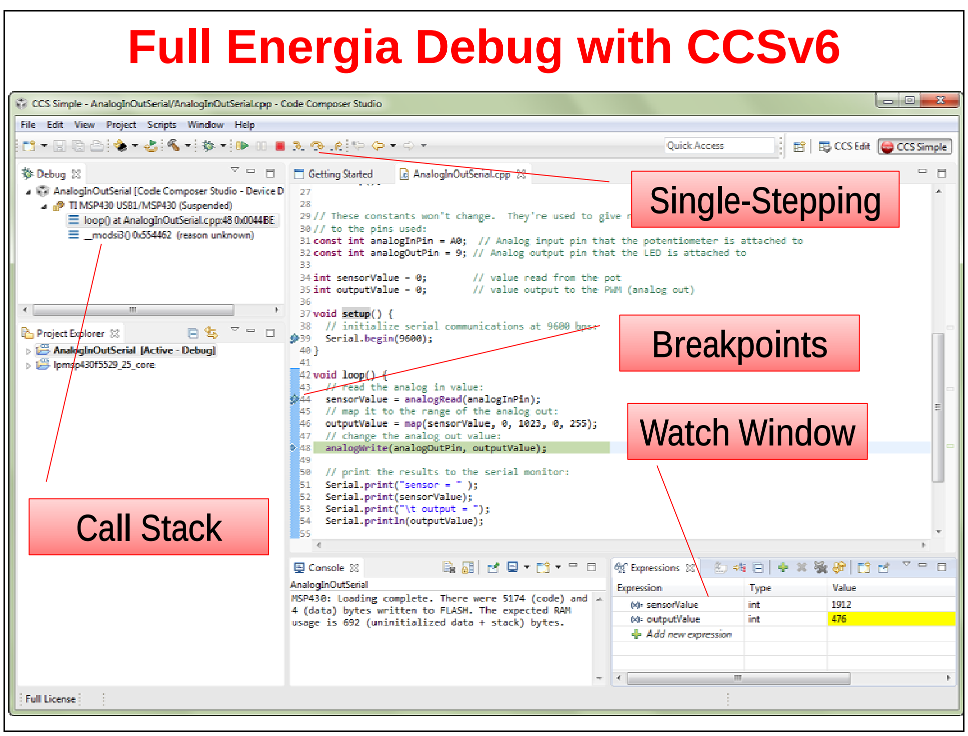
Debugging code is handled in the common, open-source fashion: printf() style. Although, rather than using printf(), you can use the Serial print functions to keep track of what is going on with your programs. Similarly, we often use LED's to help indicate status of program execution. And, if you have 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.

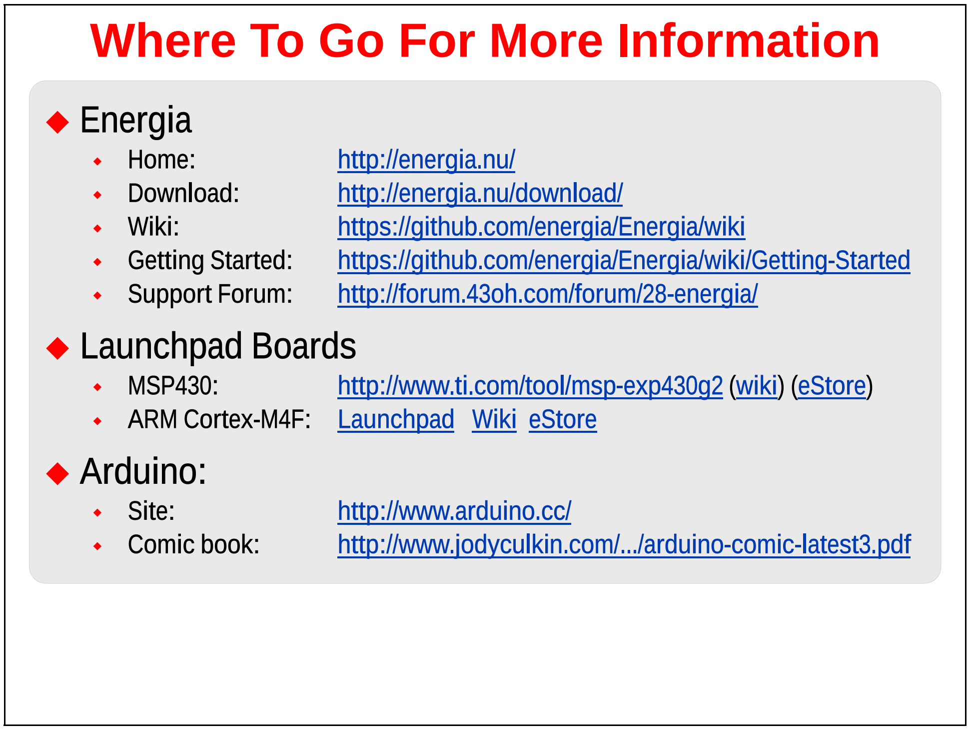


Debugging Energia (Arduio) with CCSv6



Energia/Arduino References

There are many more Arduino references that could possibly be listed here, but this should help get you started.



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 11

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. Push-Button Interrupt
E. Timer Interrupt (Uses Non-Energia Code)

Lab Topics

Using Energia (Arduino)........ 11-14

Lab 11 11-15
- Installing Energia........ 11-17
  - Installing the LaunchPad drivers........ 11-17
  - Installing Energia........ 11-17
  - Starting and Configuring Energia........ 11-18
- Lab 11a – Blink 11-21
  - Your First Sketch........ 11-21
  - Modifying Blink........ 11-24
- Lab 11b – Pushing Your button 11-25
  - Examine the code 11-25
  - Reverse button/LED action 11-26
- Lab 11c – Serial Communication (and Debugging) 11-27
  - What if the Serial Monitor is blank? ('G2553 Launchpad Configuration)........ 11-28
  - Blink with Serial Communication........ 11-29
  - Another Pushbutton/Serial Example 11-29
- Lab 11d – Using Interrupts........ 11-30
  - Adding an Interrupt........ 11-30
- Lab 11e – Using TIMER_A 11-32

Appendix – Looking 'Under the Hood' 11-33
  - Where, oh where, is Main 11-33
  - Two ways to change the MSP430 clock source 11-35
  - Sidebar – initClocks()........ 11-36
  - Sidebar Cont'd - Where is *F_CPU* defined? 11-37

Lab Debrief........ 11-38
- Lab 11a 11-38
- Lab 11b 11-39
- Lab 11c........ 11-40
- Lab 11d 11-42

Installing Energia

If you already installed Energia as part of the workshop prework, then you can skip this step and continue to Lab 11a – 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 6.x with MSP430 suport 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

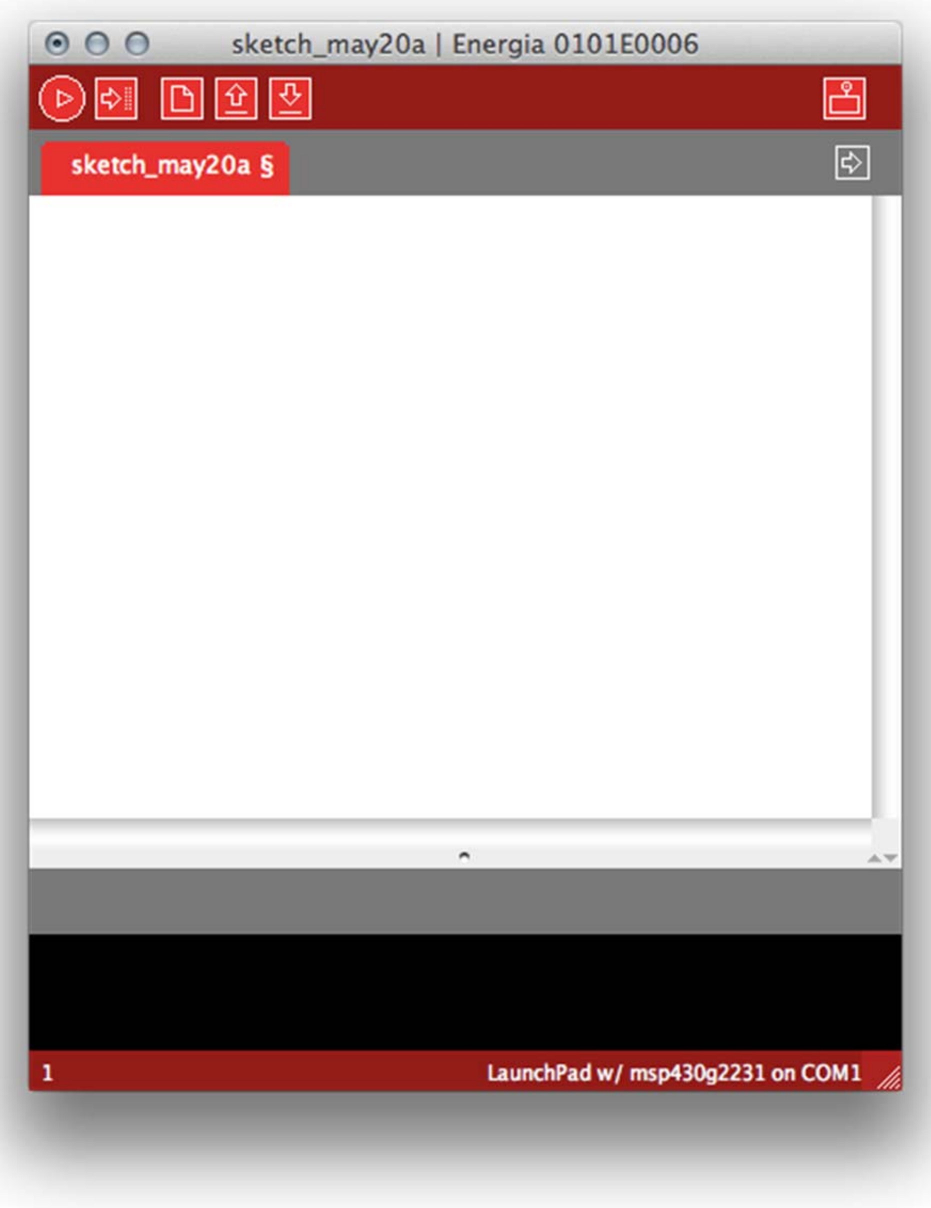
Double click and extract the energia-0101EXXXX-windows.zip file to a desired location.

(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.

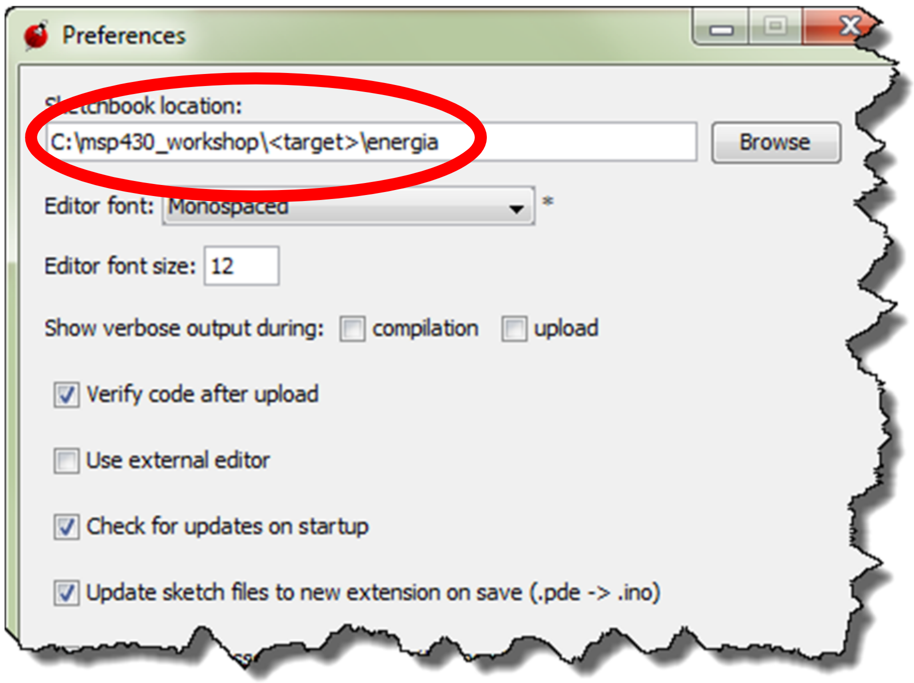
The easiest way to set this locations is via Energia's preferences dialog:

```
File → Preferences
```

Then set the *Sketchbook location* to:

```
C:\msp430_workshop\<target>\energia
```

Which opens:



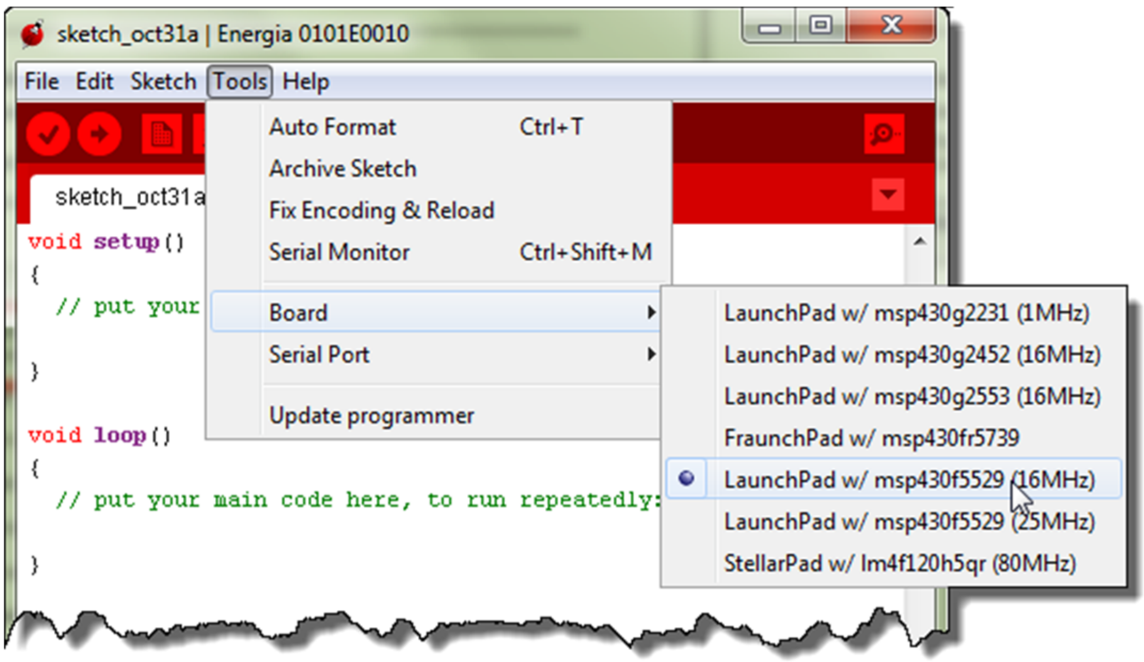
5. Selecting the Serial Port

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 11a – 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.

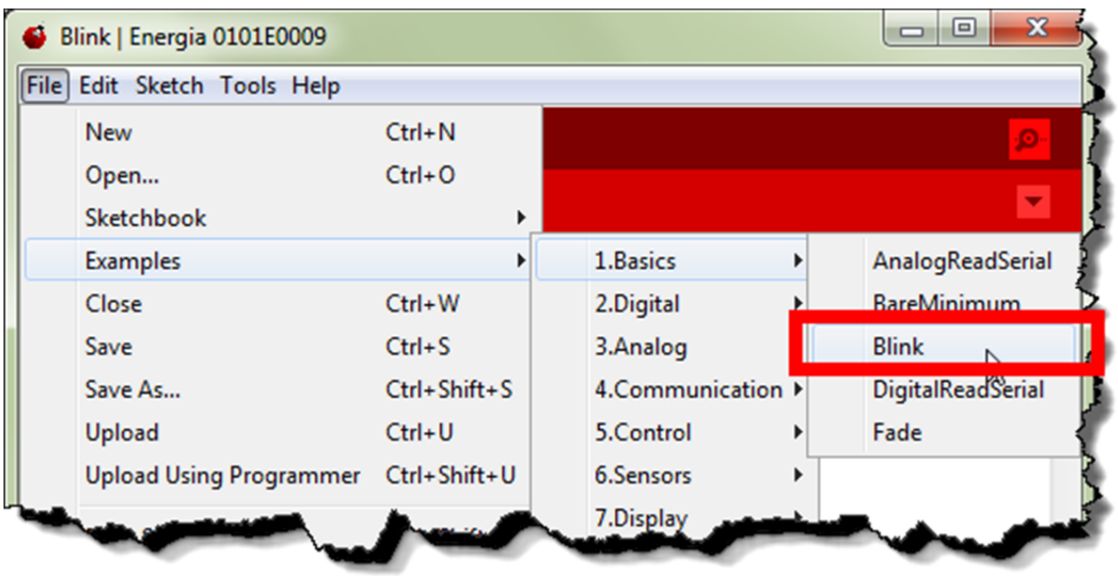
As we pointed out during the *Energia* chapter discussion, the *Wiring* language simplifies the code quite a bit.

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 Lab11d we examine some of the specific differences between this sketch and C code.)

At this point, due to their similarity to standard C language code, we will assume that you recognize most of the elements of this code. By that, we mean you should recognize and understand the following items:

- **#define** – to declare symbols
- **Functions** – what a function is, including: void, () and {}
- **Comments** – declared here using // characters

What we do want to comment on is the names of the two functions defined here:

- **setup():** happens one time when program starts to run
- **loop():** repeats over and over again

This is the basic structure of an Energia/Arduino sketch. Every sketch should have – at the very least – these two functions. Of course, if you don't need to setup anything, for example, you can leave it empty.

```
/*
  Blink
  Turns on an LED on for one second, then off for one second,
  repeatedly. This example code is in the public domain.
 */

void setup () {
  // initialize the digital pin as an output.
  // Pin 14 has an LED connected on most Arduino boards:
        pinMode (RED_LED, OUTPUT);
}

void loop () {
        digitalWrite (RED_LED, HIGH);     // turn on LED
        delay (1000);                     // wait one second (1000ms)
        digitalWrite (RED_LED, LOW);      // turn off LED
        delay (1000);                     // wait one second
}
```

3. Compile and upload your program to the board.

To compile and upload the Sketch to the LaunchPad click the  button.

Do you see the LED blinking? What color LED is blinking? ___________________________

What pin is this LED connected to? ____

(Be aware, in the current release of Energia, this could be a trick question.)

Hint:	We recommend you check out the Hardware Pin Mapping to answer this last question. There's a copy of it in the presentation. Of course, the original is on the Energia wiki.

Modifying Blink

4. Copy sketch to new file before modification.

We recommend saving the original Blink sketch to a new file before modifying the code.

```
File → Save As…
```

Save it to:

```
C:\msp430_workshop\<target>\energia\Blink_Green
```

Hint: This will actually save the file to:

```
C:\msp430_workshop\<target>\energia\Blink_Green\Blink_Green.ino
```

Energia requires the sketch file (.ino) to their to be in a folder named for the project.

5. How can you change which color LED blinks?

Examine the H/W pin mapping for your board to determine what needs to change.

Please describe it here: ____

____

6. Make the other LED blink.

Change the code, to make the other LED blink.

When you've changed the code, click the ***Upload*** button to: compile the sketch; upload the program to the processor's Flash memory; and, run the program sketch.

Did it work? ____

(We hope so. Please ask for help if you cannot get it to work.)

Lab 11b – 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.

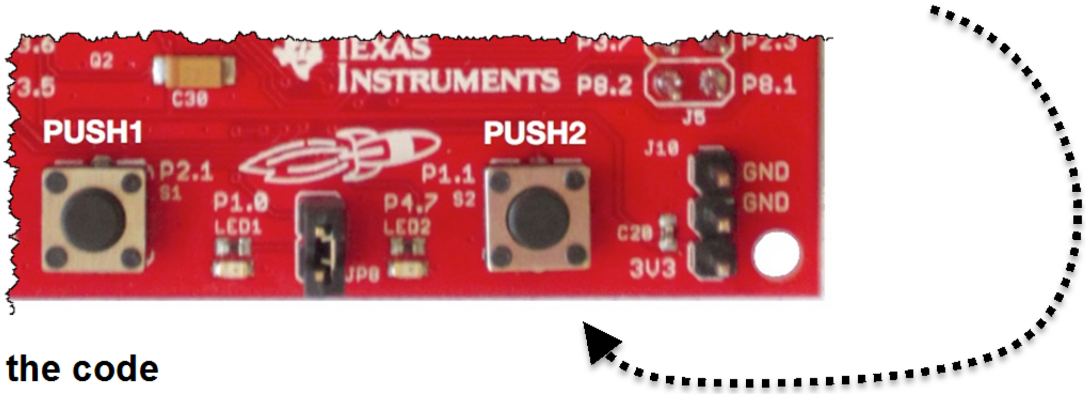
   `File → Examples → 2.Digial → Button`

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 11c – 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.**

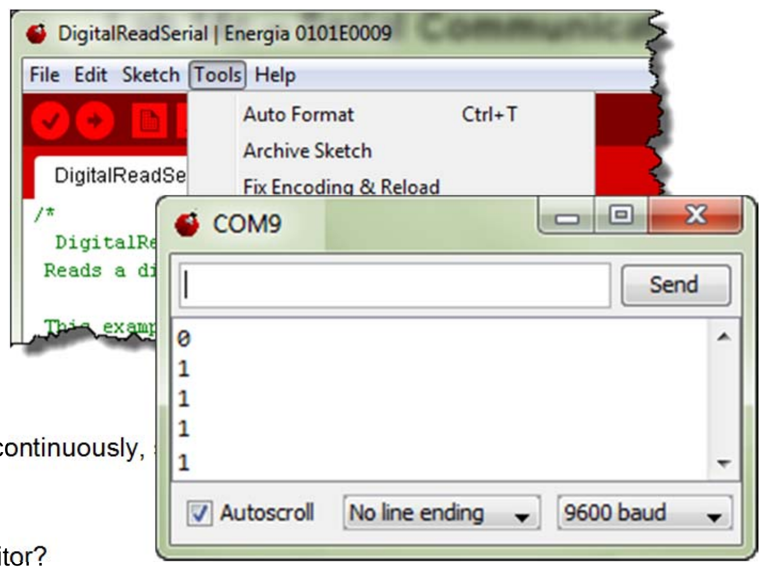
   With the code downloaded and (automatically) running on the Launchpad, go ahead and `push the button`.

   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:

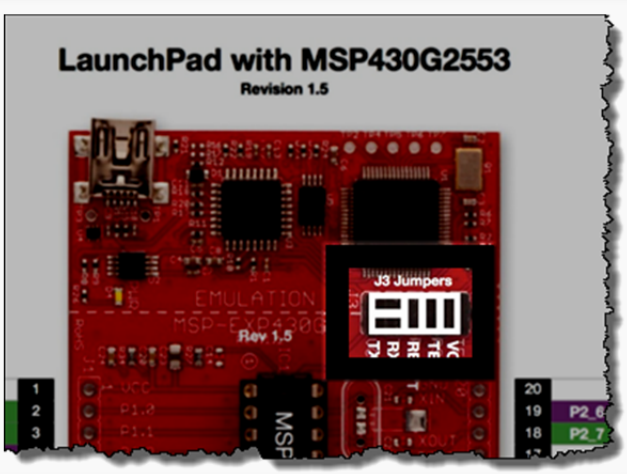
   You should see either a "1" or "0" depending upon whether the putton is up or down.

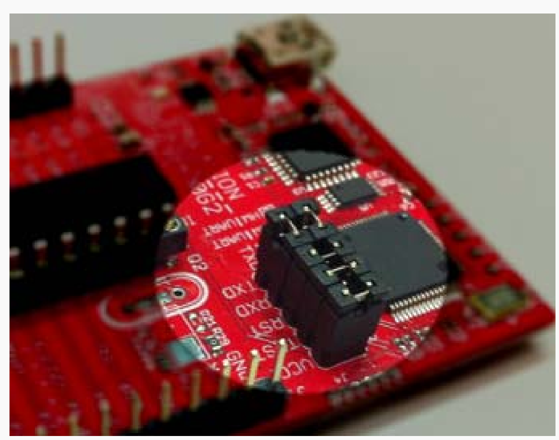
   Also, notice that the value is updated continuously, writes it to port in the **loop()** function.

   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.Digial → 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 11C, let's look at one more example.

11. Open the *StateChangeDetection* sketch.

Load the *sketch* from the *Examples* menu.

```
File → Examples → 2.Digial → 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 11d – 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. ____

One 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 11e – 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 essense, though, the same three steps are required:

a) The interrupt source must be setup. In our example, this means seting 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 11d. 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 – Inititialization 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)

____

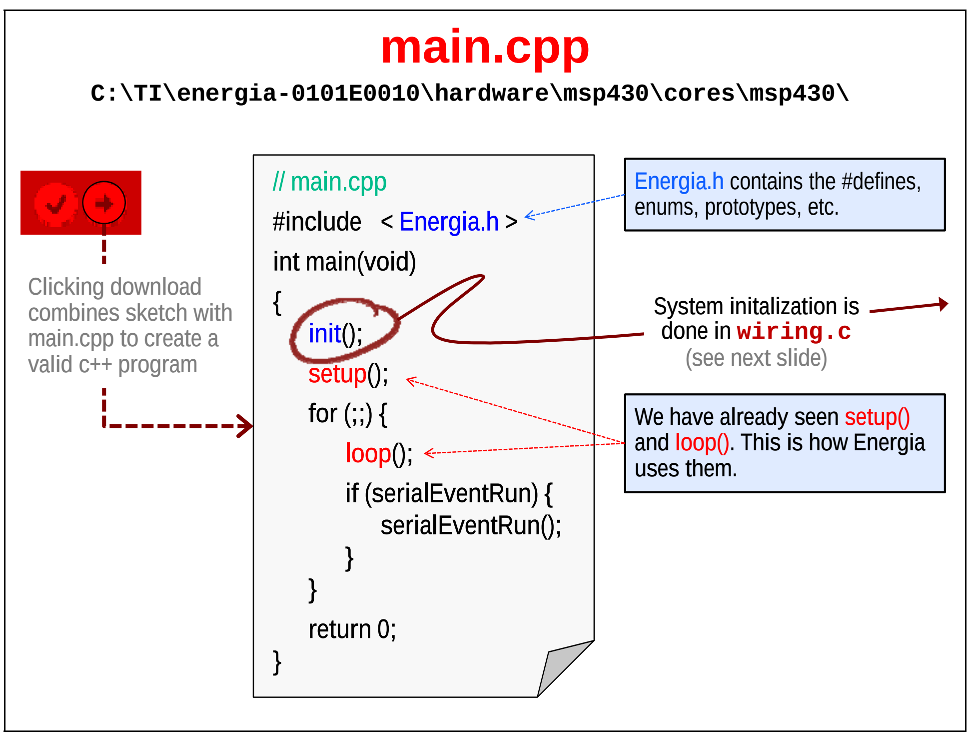
If Energia/Arduino is built around the C language, where is the *main()* function? Once we answer this question, then we will see how the system clock is initialized.

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:



Where do you think the MSP430 clocks are initialized? ________________________

Follow the trail. Open `wiring.c` to find how *init()* is implemented.

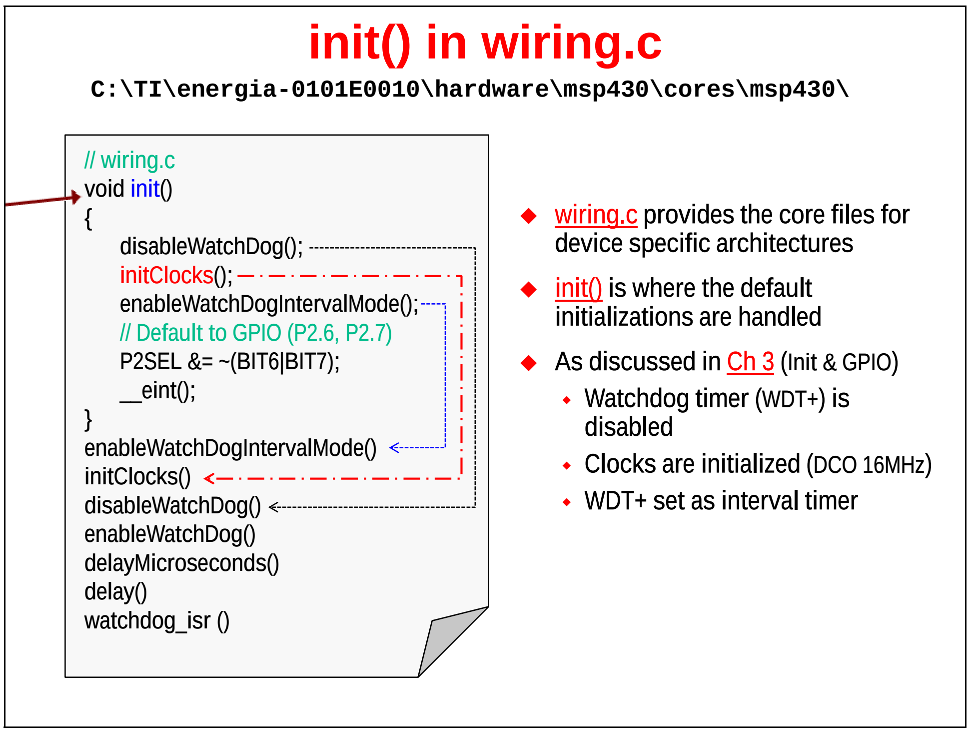
`C:\TI\energia-0101E0010\hardware\msp430\cores\msp430\wiring.c`

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 activies:

- 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



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

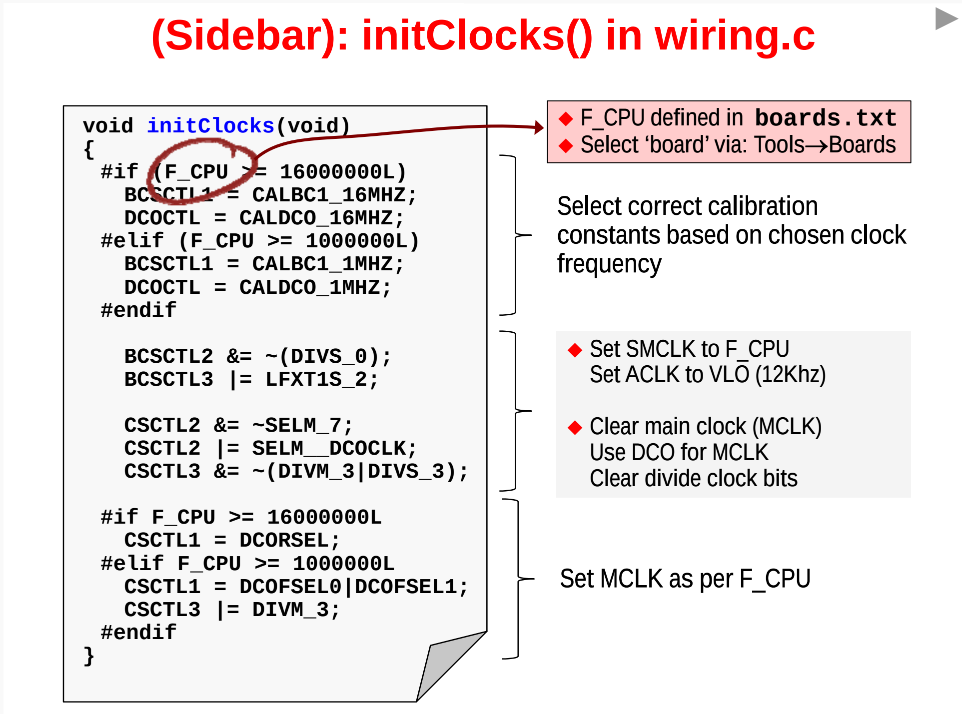
Disdvantages

- 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.



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 specifiy 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)

##############################################################
lpmsp430g2553.name=LaunchPad w/ msp430g2553 (16MHz)
lpmsp430g2553.upload.protocol=rf2500
lpmsp430g2553.upload.maximum_size=16384
lpmsp430g2553.build.mcu=msp430g2553
lpmsp430g2553.build.f_cpu=16000000L
lpmsp430g2553.build.core=msp430
lpmsp430g2553.build.variant=launchpad

##############################################################
lpmsp430fr5739.name=FraunchPad w/ msp430fr5739
lpmsp430fr5739.upload.protocol=rf2500
lpmsp430fr5739.upload.maximum_size=15872
lpmsp430fr5739.build.mcu=msp430fr5739
lpmsp430fr5739.build.f_cpu=16000000L
lpmsp430fr5739.build.core=msp430
lpmsp430fr5739.build.variant=fraunchpad

##############################################################
```

Lab Debrief

Lab 11a

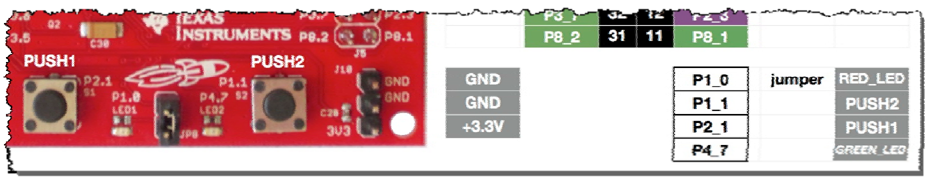
Q&A: Lab11A (1)

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: Lab11A (2)

5. How can you change which color LED blinks?

   Examine the H/W pin mapping for your board to determine what needs to change.

   Please describe it here: ___Change from P1_0 to P4_7, for the green LED to blink___
   (Easier yet, just use the pre-defined symbol: GREEN_LED)

6. Make the other LED blink.

   Did it work? ___Yes___

Lab 11b

Q&A: Lab11B (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: Lab11B (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.

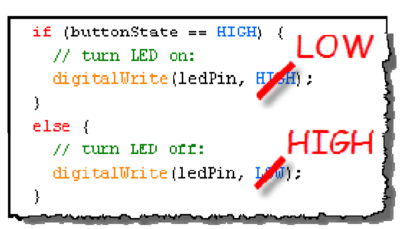
   What would be a more efficient way to handle responding to a pushbutton? (And why would this be important to many of us MSP430 users?)

   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)



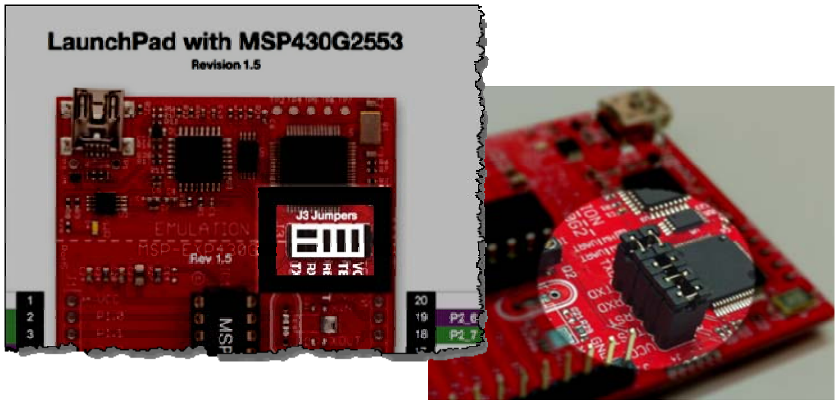
Lab 11c

Q&A: Lab11C (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

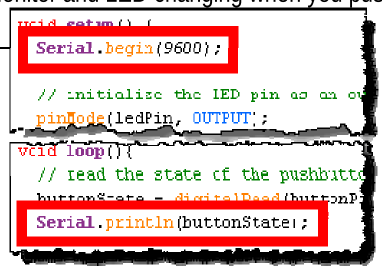


Q&A: Lab11C (2)

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)



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: Lab11C (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 sketchs – thusfar – have used polling to determine the state of the button. It would be more efficient to let the processor sleep; then be woken up by an interrupt generated when the pushbutton is depressed.

Lab 11d

Q&A: Lab11D

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.

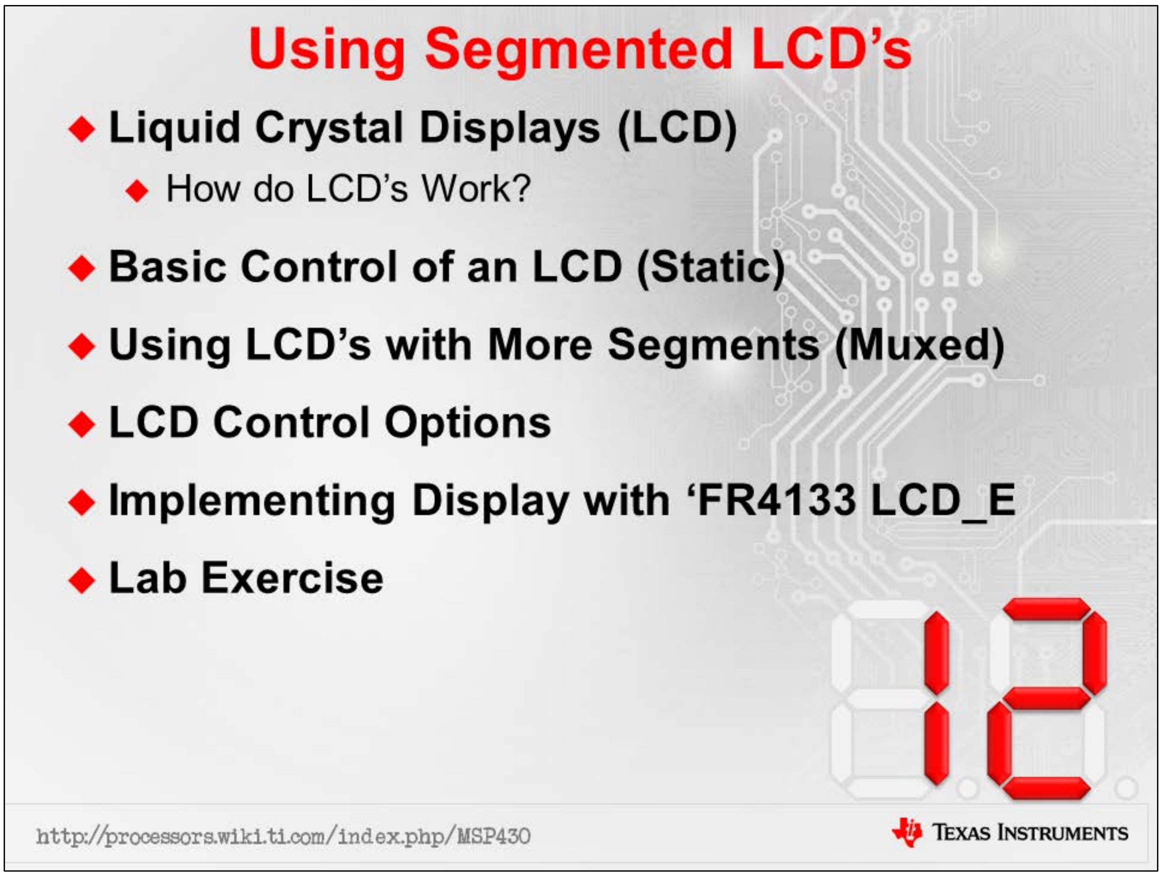
Using Segmented Displays (LCD)

Introduction

This chapter introduces the segmented liquid crystal display (LCD). We begin with a quick introduction to LCD's and how they work. Second, we look at how they can be controlled and used within an embedded system.

Finally we learn how to implement designs with the LCD_E controller found on the MSP430FR4133 microcontroller. The 'FR4133 Launchpad – with its built-in LCD display – makes a great platform platform for LCD experimentation.

Learning Objectives



Chapter Topics

Using Segmented Displays (LCD) 12-1

For More Information on LCD's 12-2

Liquid Crystal Displays (LCD) 12-3
How do LCD's Work? 12-5

Basic Control of an LCD (Static) 12-11

Using LCD's with More Segments (Muxed) 12-16
Static vs Muxed 12-16
Muxed Control Signals 12-19

LCD Control Options 12-21
Bit Banging a Display 12-21
Displays with Built-in Drivers 12-23
MSP430 LCD Peripherals 12-24

Implementing Display with 'FR4133 LCD_E 12-26
Choose Display and Pin Layout 12-26
LCD Init Code 12-29
Controlling Segments 12-49
Dual Memories & Blinking 12-55

Lab Exercise 12-57

For More Information on LCD's

For More Information

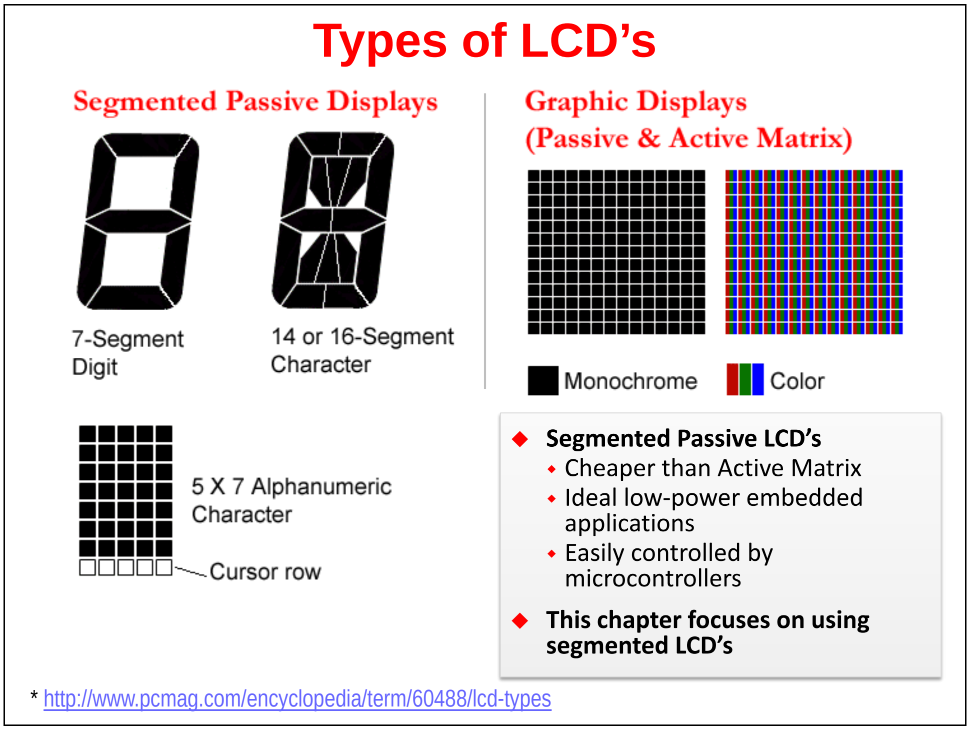
Here are a couple of resources you can refer to for more information concerning the use of Segemented LCD's:

- *Designing With MSP430™ MCUs and Segment LCDs* application note: www.ti.com/lit/pdf/slaa654
- *MSP430 Microcontroller Basics* by John H. Davies, (ISBN-10 0750682760) Link↗

Liquid Crystal Displays (LCD)

There are many types of Liquid Crystal Displays available today – from the simple 7-Segment single-digit displays, all the way up to the displays used for computer and television screens.

MSP430-based applications tend to favor low-cost, low-power segmented displays – typically with less than 100-400 segments (or dots).



Large graphic displays like a computer or portable DVD player, on the other hand, tend to be used in systems with high-level operating systems and much faster processors. As you might imagine, this chapter focuses on the simpler lower-cost, segmented displays.

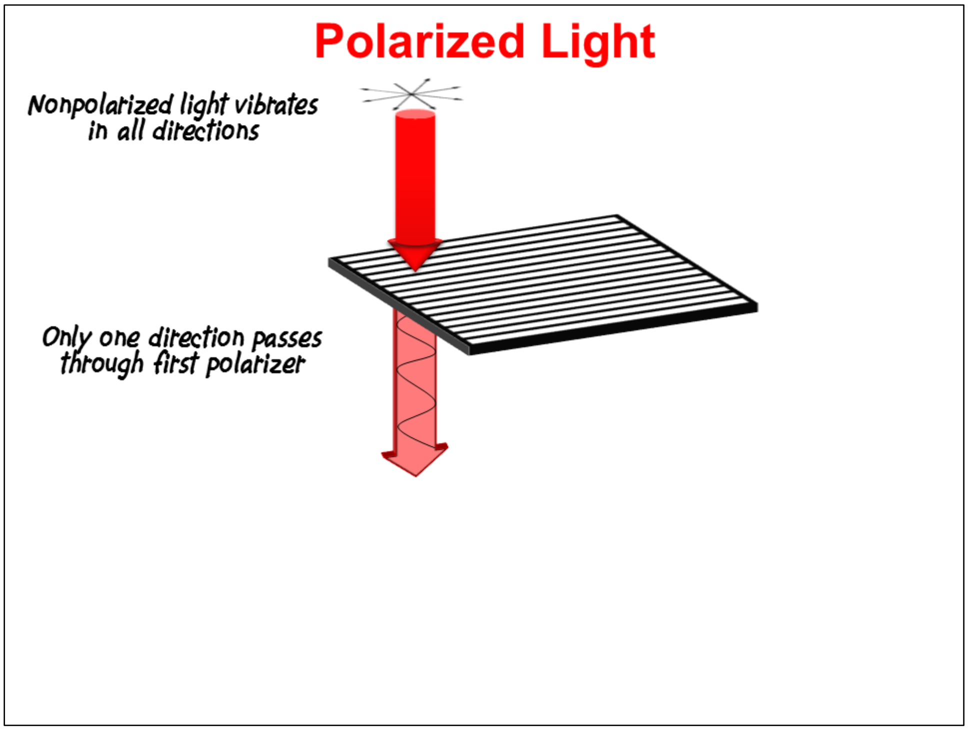
A large of number of end applications can benefit from MSP430s with integrated LCD controllers. These include remote controls, blood glucose meters, and any LCD application where battery power matters.

How do LCD's Work?

Two physical properties are the fundamental elements of LCD displays:

- Polarizers
- Liquid Crystals.

Let's begin with polarizers.



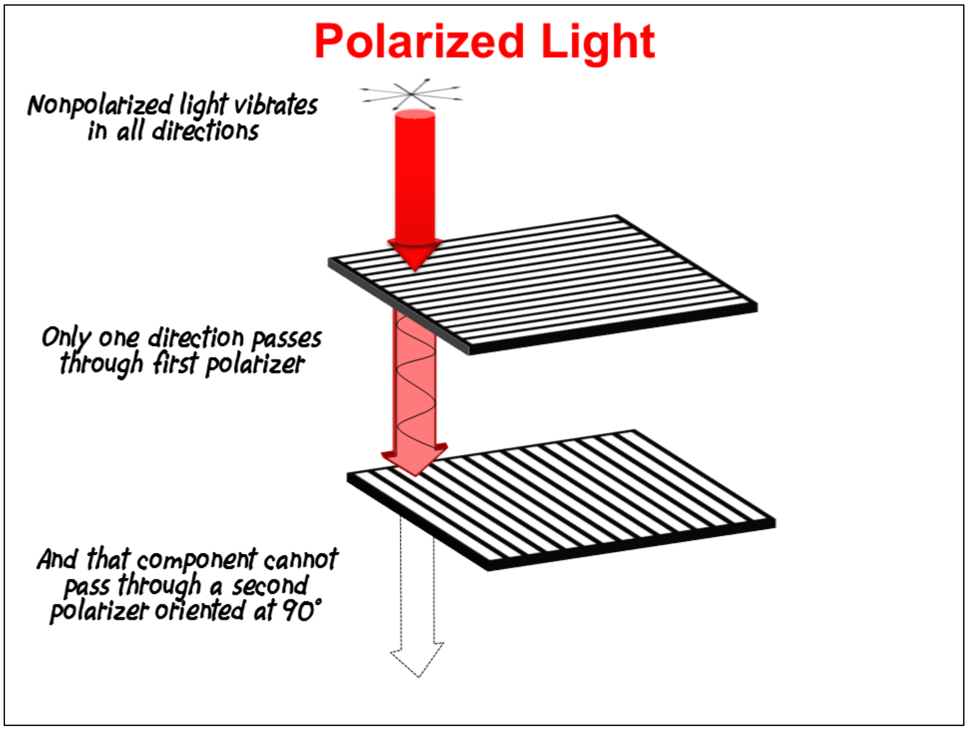
As you might remember from high-school physics, light consists of both particles and waves. For our purposes, we're interested in the "wave" concept. Shown at the top of the diagram light waves are vibrating in all directions. But after passing through a polarizer, light only vibrates in a single direction.

In other words, you might say polarizers filter all light waves except those vibrating in a single direction.

What happens if we add a second polarizer – one that's oriented differently?

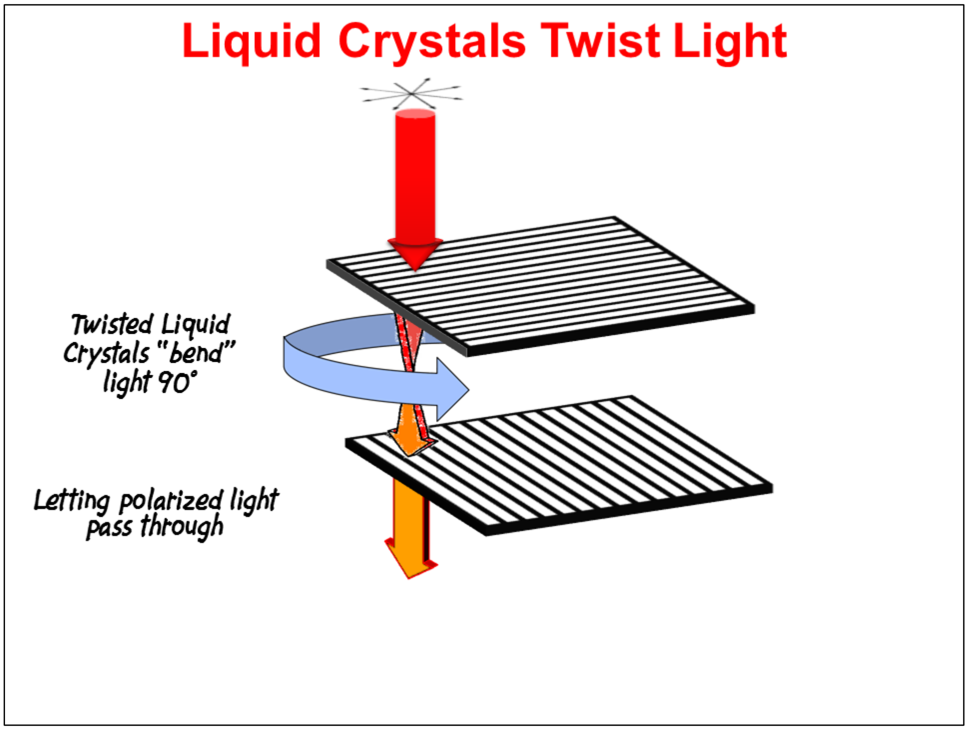
A second polarizer – rotated 90 degrees from the original – ends up blocking the light from making it through the pair of polarizers.

This is because, after passing through the first polarizer, there aren't any light waves that still vibrate in the same direction as the second polarizer.



Effectively, using only two polarizers, all of the light would be filtered out. By itself, this isn't the effect we want; but it serves our purposes when we sandwich liquid crystals between the two polarizers.

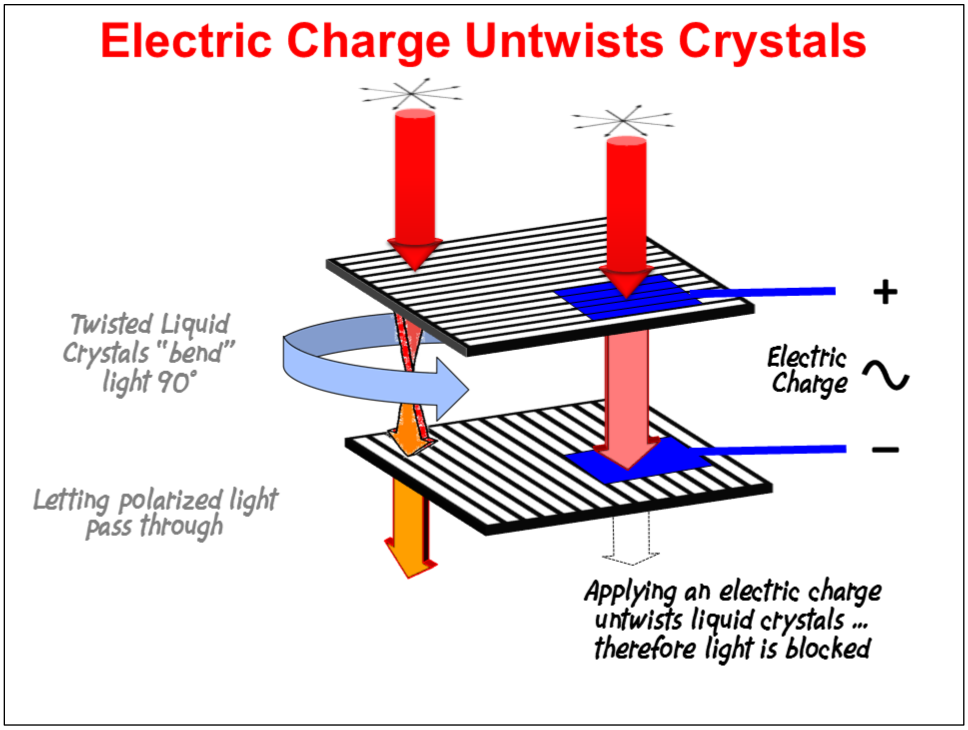
One of the unique benefits of liquid crystals is that they bend light 90 degrees. Putting these crystals between the two polarizers allows light polarized by the top one to still be able to pass through the bottom one.



If you add a reflector (i.e. mirror) to the bottom of the display, the light will bounce back and – due once again to rotation by the liquid crystals – will pass all the way through.

It's important for light to make it all the way back to the LCD user's eye, but we're still missing one key component… how to turn parts of the display on and off.

Here we see another key feature of liquid crystals… when you apply a charge to them, they untwist.



Untwisting the crystals means we're back to the point where light cannot pass through the two polarizers. Thus, by applying transparent electrodes to portions (or segments) of an LCD display, we can use an electric charge to turn that segment "on" or "off".

That is, where charge is applied, light gets blocked, and won't reflect back to the user – who sees the *dark* segment as "on".

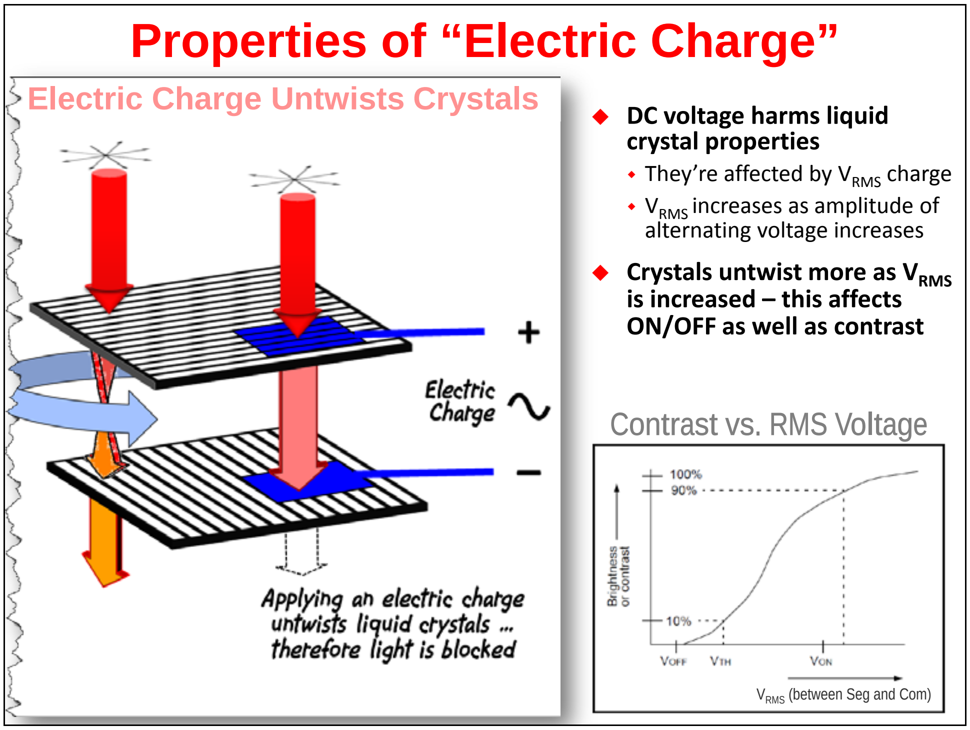
And where no charge is applied, the light gets reflected back to the user. Thus, by simply applying an electric charge, you can control the appearance of an LCD display.

One problem, though, is that liquid crystals deteriorate when direct current (i.e. DC) charge is applied to them.

It's unfortunate that Direct Current (or DC) causes harm to liquid crystals. Why, because this is the same type of charge output by a battery… as well as the general purpose I/O pins on a microcontroller.

Since DC cannot be used, that means we need to drive LCD's using Alternating Current (i.e. AC). The most common place we see Alternating Current is the electrical charge coming out of our wall sockets at home. While AC power from the electric company is too much power for driving an LCD, we can learn something from how it works. Here's the question… "If household current alternates back-and-forth – spending as much time positively charged as negative – doesn't that add up to *Zero*?"

We know that alternating currents transmit power, but a standard DC calculation doesn't seem to work out. The key, favored by engineering, is to describe AC power using a Root Mean Squared (known as RMS) calculation. *The RMS value of a periodic current is equal to the direct current (DC) that delivers the same average power to a resistor as the periodic current.*



Looking back at our LCD, we'll toggle our processors pins back-and-forth to create the required Alternating Current. Similar to household power, it's the RMS power generated by our pins that "untwists" the liquid crystals. A larger RMS charge (above the threshold) turns a segment "on" while a smaller charge lets it remain "off".

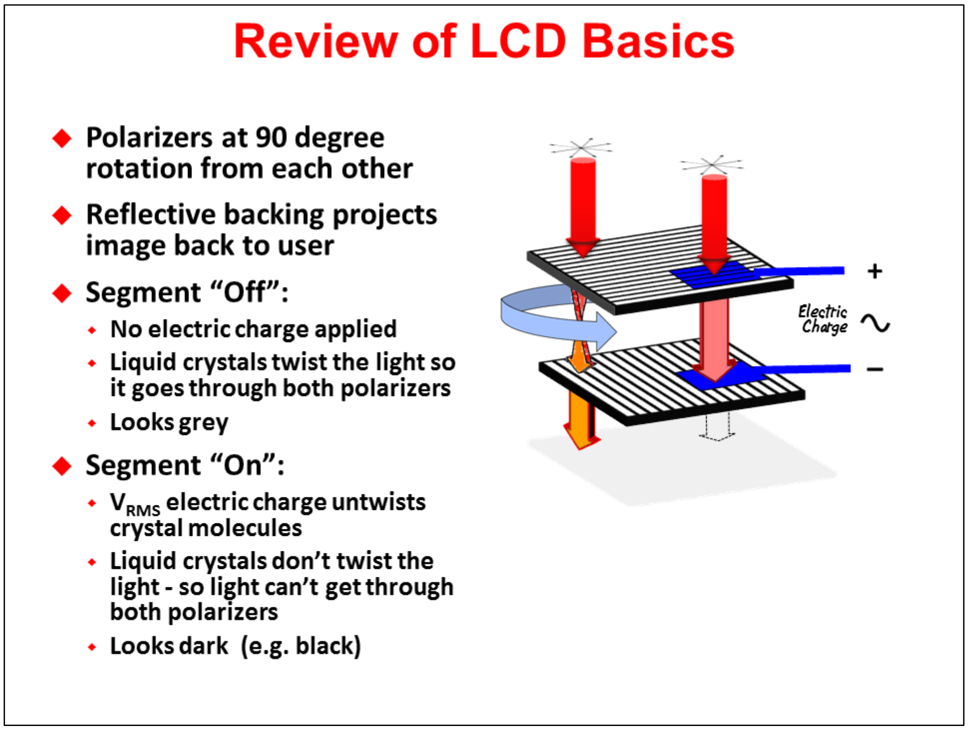
Notice the chart in the bottom corner of the slide. An LCD really isn't just "on" or "off". Rather, since these crystals react in an analog sort of fashion, using different voltages can make them appear darker or lighter. Using this idea often gives us a way to control the contrast of the display.

To quickly review how Liquid Crystal Displays work: Two polarizer planes are situated at 90˚ angles from each other. Without the liquid crystals, no light would be reflected back from the mirror-like backing.

Displays are broken up into different segments. These are defined by translucent electrodes that apply a charge across liquid crystals.

If no charge is applied, then the crystals twist the light allowing it though both polarizers. Hence, the segment is “off”. Conversely, when a charge is applied, the crystals don’t bend the light and light is not permitted to reflect back to the user, hence the segment is “on”.

Finally… and what a pain it is… direct current damages the liquid crystals; therefore, an RMS voltage (that is, alternating current) must be used for to control the segments.

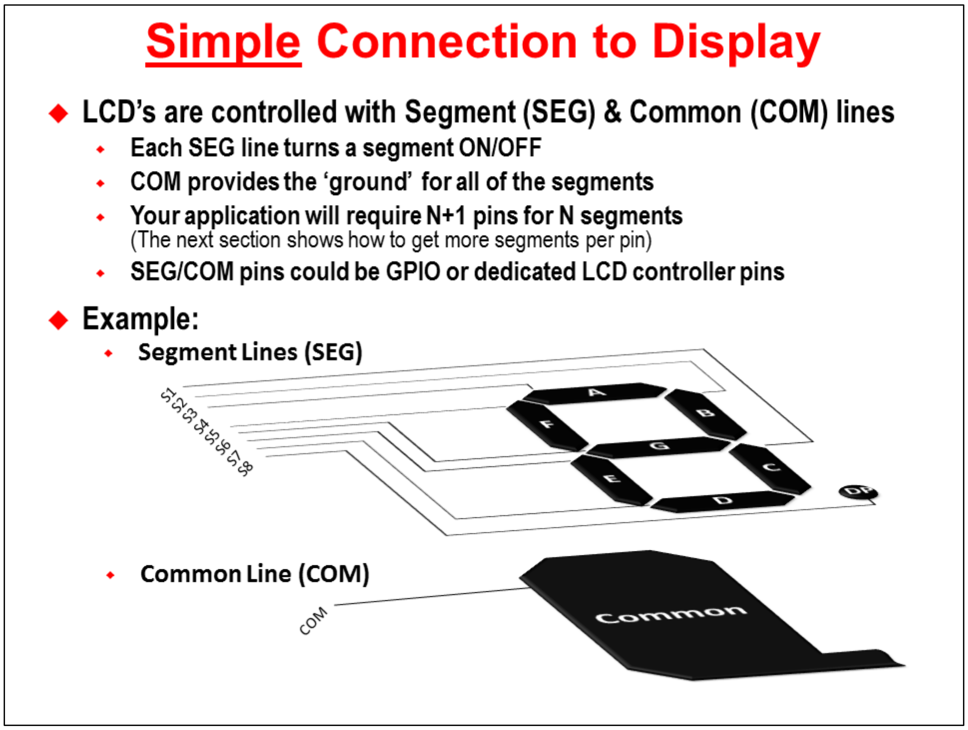


Basic Control of an LCD (Static)

With that background on how LCD's work, let's turn to what control signals can be used to operate the display.

Some simple terminology – liquid crystal displays are controlled by ***Segment*** and ***Common*** lines… called "Seg" and "Com", for short.

In this example, we can see 8 segment lines (S1 thru S8) being routed to each of the segment electrodes. 8 SEG's are needed, because it's a "7-segment" display with a "decimal point".



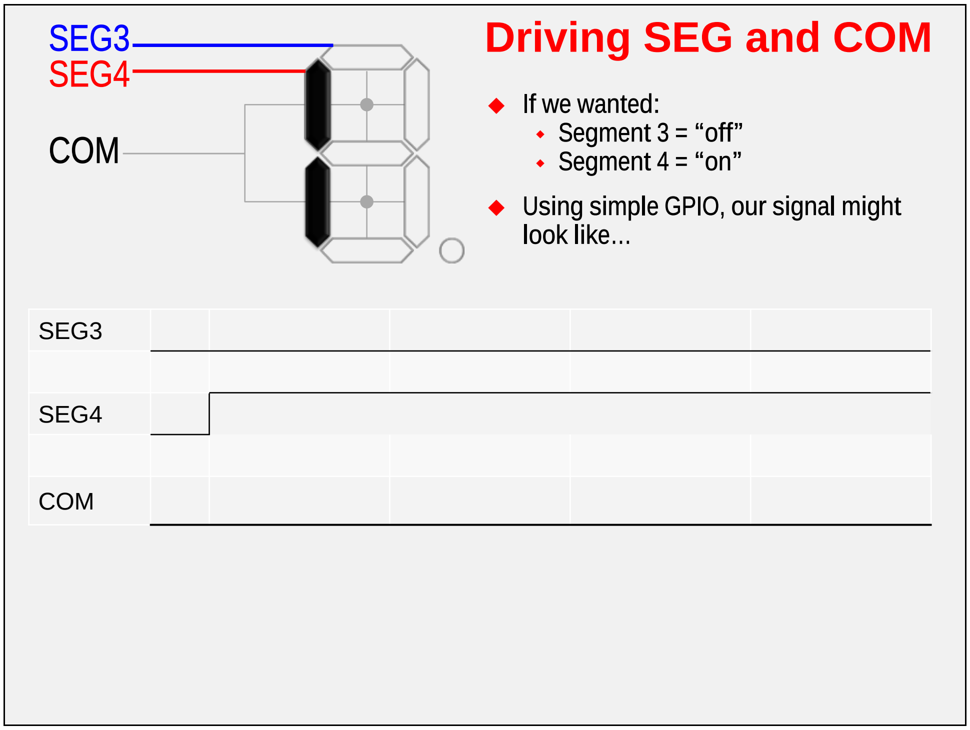
The *Common* is shown here as 1 big electrode – this is meant to emphasize that all the segments utilize the same COMMON line, hence the name. In real displays, the common electrodes tend to be shaped just like the segments, but just wired all together.

Therefore, we'll need 9 pins to drive 8 segments – one for each segment + the common line.

In an ideal world, you would be able to drive all of these N+1 signals from simple GPIO pins, where N represents the number of segments in the display.

Over the next few pages, we'll explore how we do this in the "real" world.

Once again, in an "ideal" world, we could just apply the "on" and "off" values of "1" and "0" to each segment of our display – always keeping the COM low (like a ground).

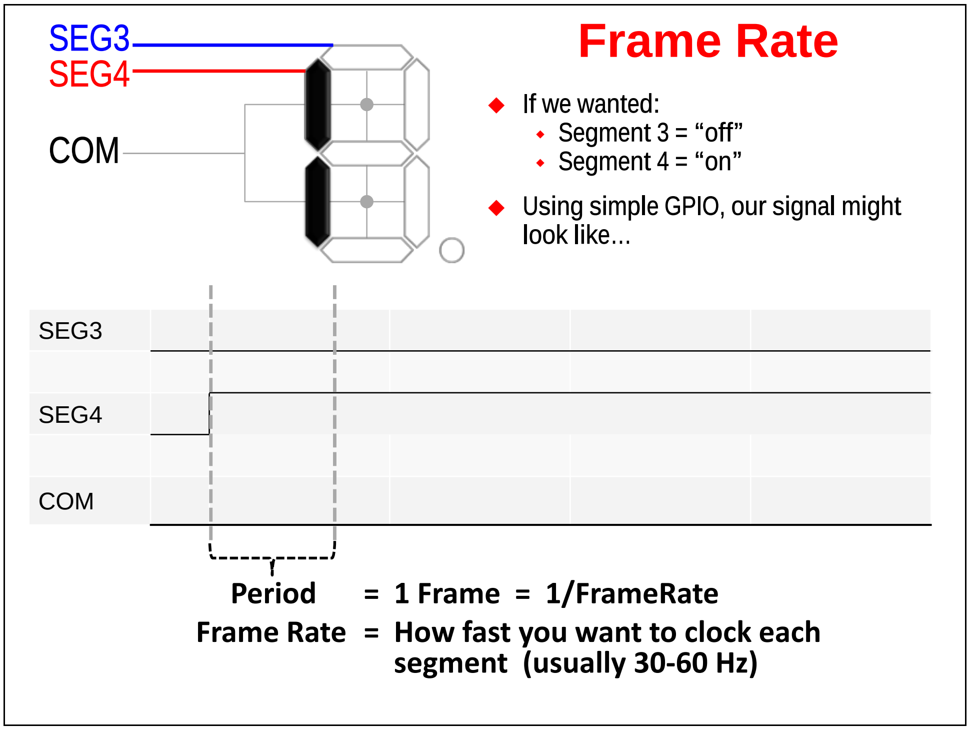


If this were the case, the only timing we would have to worry about is:

"How often do we need to update the display?"

This leads to the term: “Frame-rate” – which is the frequency with which we update the display. This term comes from video, where each frame of video is represented by a picture – the rate of successive pictures (or frames) comes down to how fast do you need to show the pictures in order to make the video appear smooth and pleasing to the eye.

The same concept applies here, even though we’re not talking ‘pictures’, but rather how frequently we need to manipulate all of the segments.

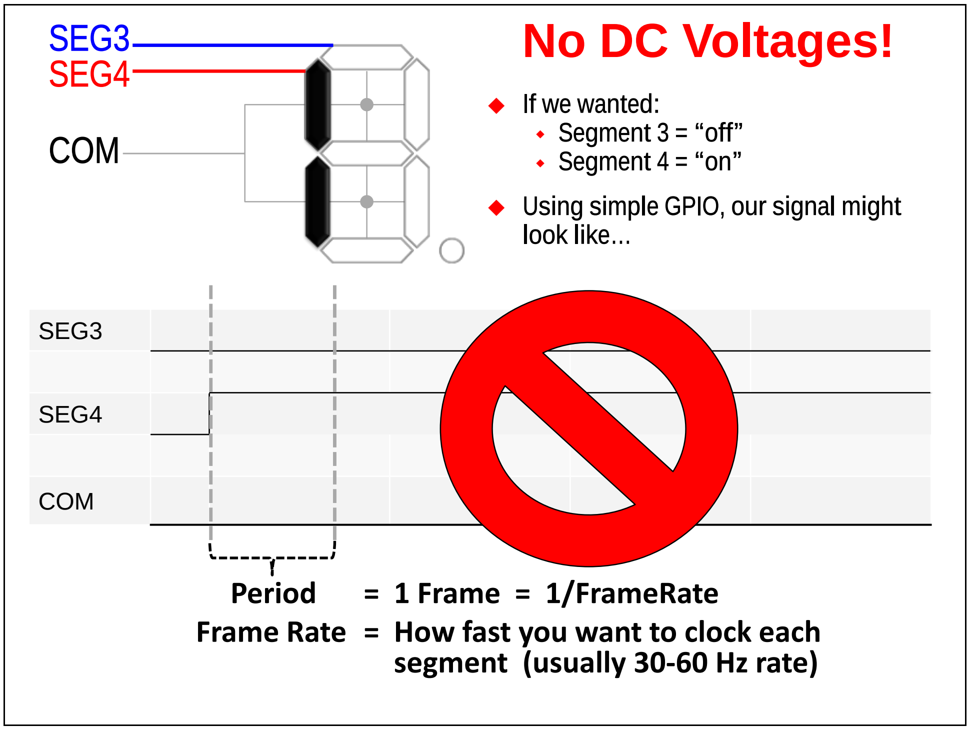


In any case, since both LCD’s and video are watched with the human eye, it’s common to find both frame-rates around 30-60 Hz.

One more term defined in this image is “Period” – this is 1 over the frame-rate frequency. The time period becomes significant since it specifies how much time we have to manipulate all of our segments.

By the way, have you noticed what is wrong with the control signals shown here? What rule (from the first part of this chapter) does this diagram violate?

If you yelled out, "YOU CANNOT USE DC VOLTAGES", then you were correct.

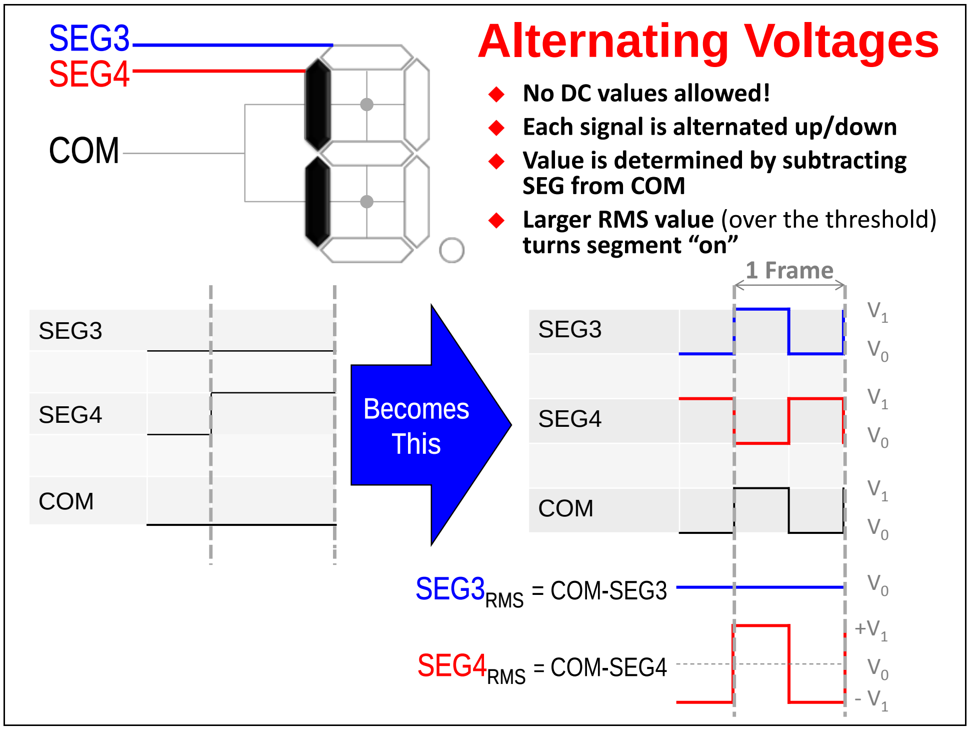


As we stated, in an ideal world we'd simply set segments on/off using DC signals from our GPIO pins… but liquid crystals deteriorate with DC voltages.

This means our signals have to become a bit more elaborate.

So, instead of using simple DC voltages, we have to convert them to AC signals – toggling our pins up and down to communicate "on" or "off". (In fact, this AC rule even applies to COM signal. Notice that even it is alternated up then down in each frame.)

With this in mind, to keep segment "off" we need to apply zero charge. Notice how SEG3 accomplishes this by having its signal follow COM; when combing the two signals together, Segment 3 is off because COM-SEG3 equates to zero average charge.



And, this means we can keep Segment 4 on by setting SEG4 to be the opposite of COM. The combination of these two signals provides enough average power to the segment to untwist the liquid crystals.

This solves one of our real-world problems. Next, we'll look at how to solve the other one.

Using LCD's with More Segments (Muxed)

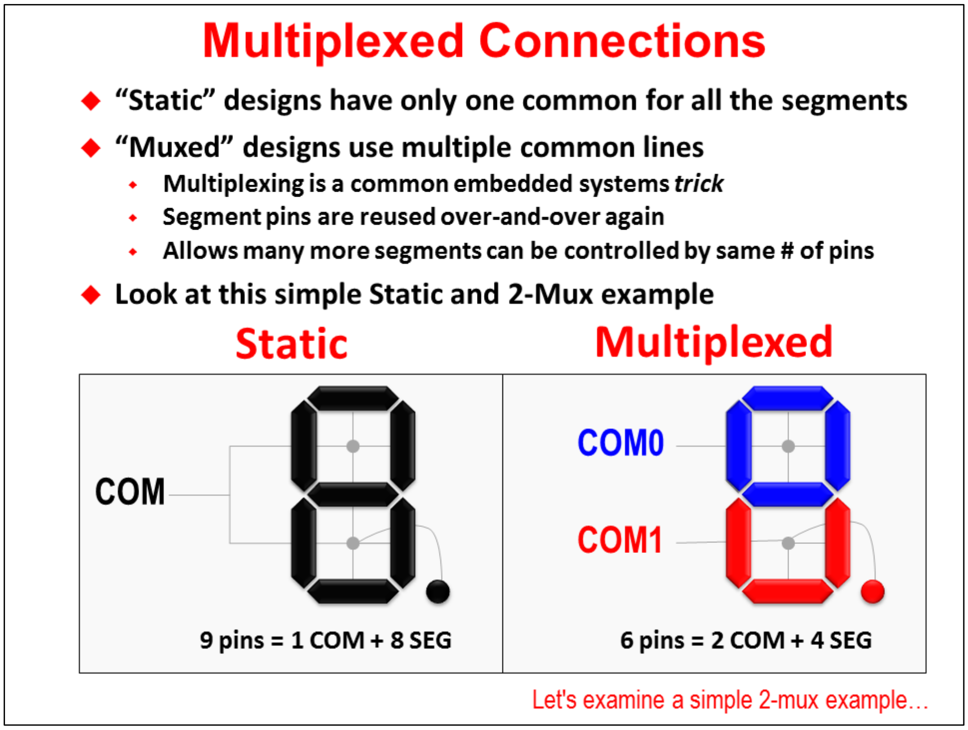
Using AC voltages solves one problem, but there's another problem that often occurs when using displays… What happens if your microcontroller doesn't have enough pins to drive every segment in the display?

Static vs Muxed

One big limitation of real-world processors is pin-count. From a cost perspective, you can only put so many pins on any microcontroller.

"Static" displays, such as that shown on the left (of the following graphic), require you to have N+1 pins… where N is the number of segments in the display. For example, an 8-segment display requires 9 pins. That's not too bad, but what happens if you want a 256-segment display? Does your micro have 257 pins available – only for use by the display?

Thankfully, we can apply a common engineering solution to this problem. ***Multiplexing*** provides a solution where each of the pins can be used for multiple purposes. (In fact, we see this concept applied in many different ways, all throughout the MSP430.)



"Multiplexed" displays – also called "Muxed" displays – utilize this solution so that you can drive many segments with a smaller number of pins. In the example to the right (of the proceeding diagram), you'll see that you can save 1/3 of the needed pins by parsing up the display segments so that the top portion of the "8" character are driven first, and then the bottom ones are driven second. In both cases, each portion utilizes the same SEG pins, but are differentiated by using different COM pins.

Again, this simple example only saves 3 pins. But, for a 256-segment display, a muxed display might only need 40-pins, versus the 257 we described earlier for static displays. Now that a BIG difference.

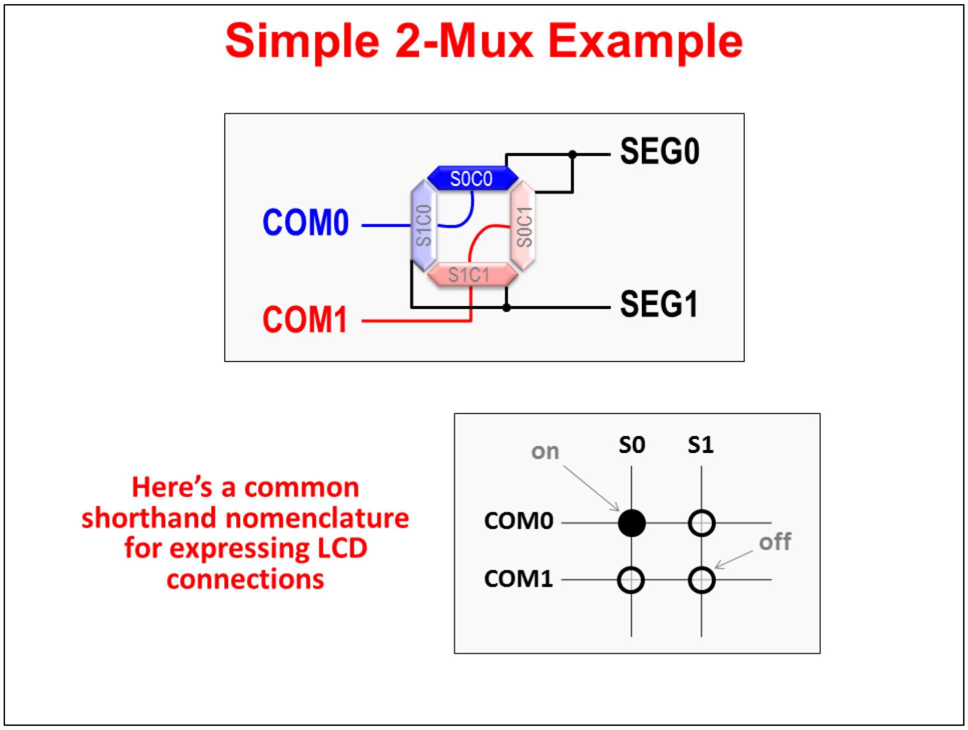
Since *Muxing* is important, let's take another look at a simple 4-segment example.

As discussed, each segment needs a SEG line, as well as a COM line. In this diagram, notice how each of our 4 segments are connected: the top and right-side segments are controlled by SEG0, while the other two are controlled by SEG1.

The only way this works is by time-division multiplexing. That is, during the first half of the time period: SEG0 and SEG1 are driven together with COM0.

During the second half of the time period: SEG0 and SEG1 are driven along with COM1.

We tried to highlight this by labeling each segment below, such as *S0C0* to represent SEG0 and COM0 control signals.

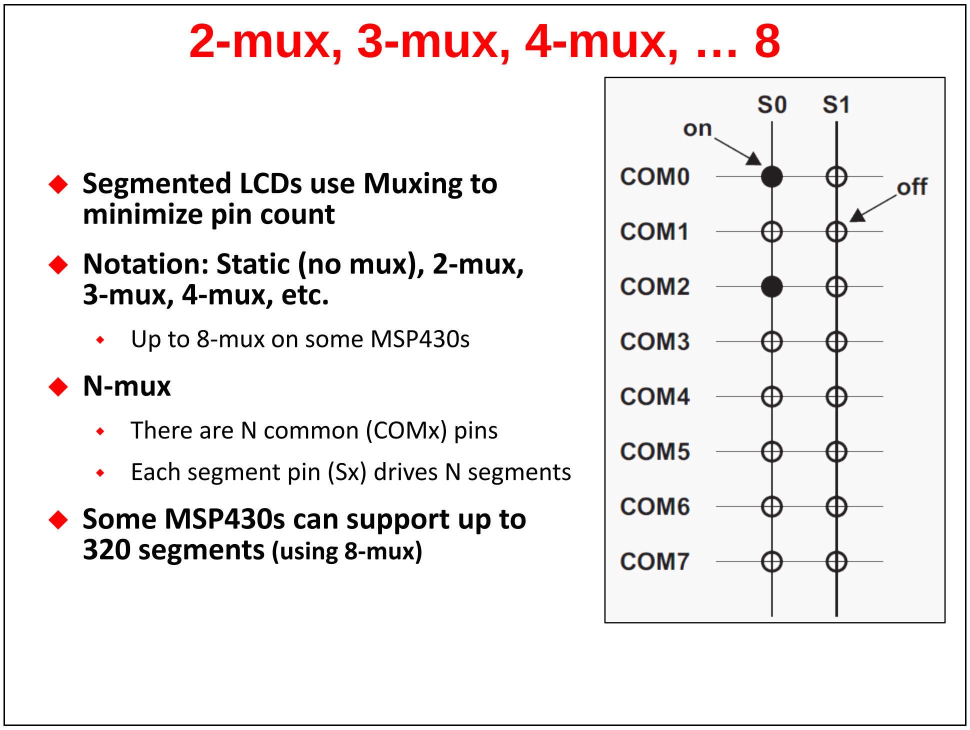


Given the example shown here, only S0C0 is supposed to be on. (We tried to show it as darker, with the others being more translucent.)

A common shorthand notation is given towards the bottom of the slide.The possible connections are shown in a matrix, with the "on" connection represented by a solid circle.

Scaling from a simple multiplexed by 2 example, known as 2-mux, the MSP430 can also support 3-mux, 4-mux … all the way up to 8-mux displays.

In fact, this is an example of 8-mux – which means there are 8 common lines. (Alternatively, you could state that each segment line can support 8 segments.)



To summarize, multiplexed displays allow us to drive many segments, while minimizing the number of pins required.

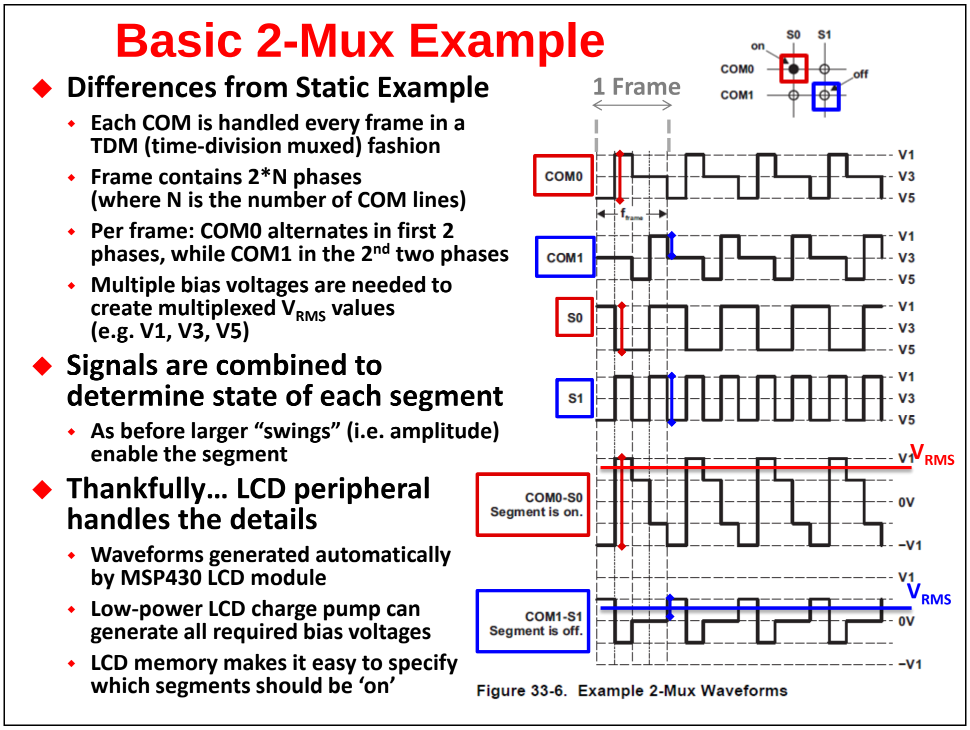
This works well, but… if you're thinking, wouldn't applying alternating RMS voltages to 8-mux frames get a little complicated… You're correct!

Muxed Control Signals

Combining AC voltages with muxed control signals can make for complicated waveforms.

Don't worry though… before we even get started talking about this slide, let's get one thing clear right up front… **the LCD controller peripheral does all of this for us!**

Thus, this page is just background information. With that in mind let's break this diagram down.



For a given frame, there must be 2 phases for every COM line. Therefore, a 2-mux display would need 4 phases (2 phases * 2-mux). In fact, this fits with our previous 2-mux example; remember that we only needed 2 phases for our "static" (i.e. 1-mux) display.

Extrapolating from the number of phases, your clock would need to be 4x the frame-rate. If you wanted to update your display at 60 frames per second, you would need to set the LCD control clock to 240 Hz (or greater)

That takes care of clock timing, but what about controlling the "on" and "off" values?

This is also a little more complicated. Rather than using just two voltages to signify "on" and "off", it's easier to create the required RMS average charge values by using multiple voltages – called "bias" voltages. You can see these to the right of the diagram, represented by: V1, V3, and V5.

While this makes the diagram more complicated, we don't really have to worry about it since the LCD controller handles all of the signals, combining them to achieve the proper "on" and "off" values for each multiplexed segment.

Note: This example shows 3 bias voltages – which is sometime called 1/3 bias since they differ from each other by thirds. Some displays require fewer (1/2 bias), while others require more bias voltages – though 1/3 bias is probably most common. In any case, the display's documentation should indicate what bias voltages it requires.

To reiterate… if you're using an **LCD CONTROLLER, IT HANDLES ALL OF THE COMPLEX SIGNALS, TIMING, and VOLTAGES**.

This means… *you* only need to define a basic set of parameters to the LCD controller.

For example, selecting a specific display for your application will determine the number of muxes (i.e. N-mux), the number of bias voltages and the number of pins that are required.

LCD Controller

- **As stated previously, LCD Controller peripheral handles all the complex timing, signal and voltages**
- **You only need to select the controllers modes and sources:**
  - **The choice of display defines:**
    N-mux, # of bias voltages and # of pins required
  - **LCD controller's clock source**
  - **Voltage reference source (for Bias voltage generation)**
    May be external (via resistors) or from one of the internal voltage references
  - **Which segments are "on" and "off"**
  - **Other device specific features, such as 'blinking'**

What other parameters do you need to specify?

- Since the controller handles all the fancy timing and signal generation, you only need to choose which *clock source* it should use.
- Similarly, you need to define the *reference voltage* and the controller will auto-generate all the other required voltages. (Of course, you also have the option to create the voltages externally using a resistor network – in this mode, the voltages become inputs to the controller.)
- Finally, you'll choose – and may vary during runtime – which segments of the display should be on, off, and/or blinking.

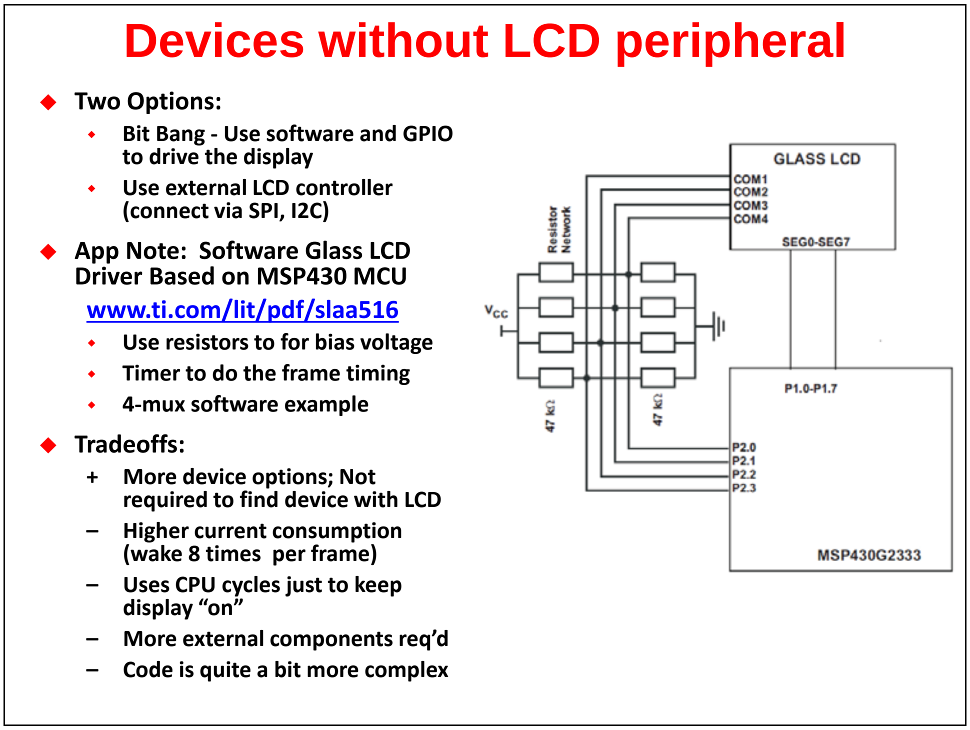
In a few pages we'll examine the code required to select each of these options.

LCD Control Options

We'll take a very brief look at three different ways you can control an LCD – before focusing our efforts on the 3rd method, which uses the dedicated on-chip LCD controller peripheral.

Bit Banging a Display

As listed here, "bit banging" is one option. You could generate the waveforms required by the LCD by using GPIO, as well as timers and/or serial ports.

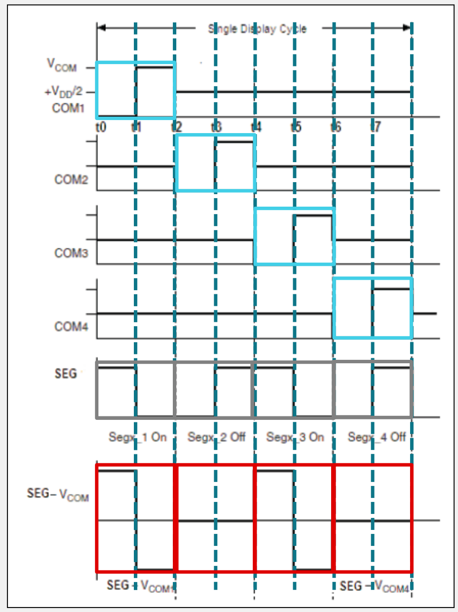


In fact, there's an app note (highlighted in the slide) which you can refer to if you choose to go this route.

SLAA516 App Note

- Example code for 4-mux included with app note
- Frame divided into 8 time slots –
  - 4 (1 for each COM)
  - each divided into two parts because no DC on LCD (must toggle)
- Timer used to generate the 8 slots
- Must wake on each slot and software set all COM & SEG lines
- SEG same as COM = off
- SEG opposite COM = on

Time Slots



Texas Instruments

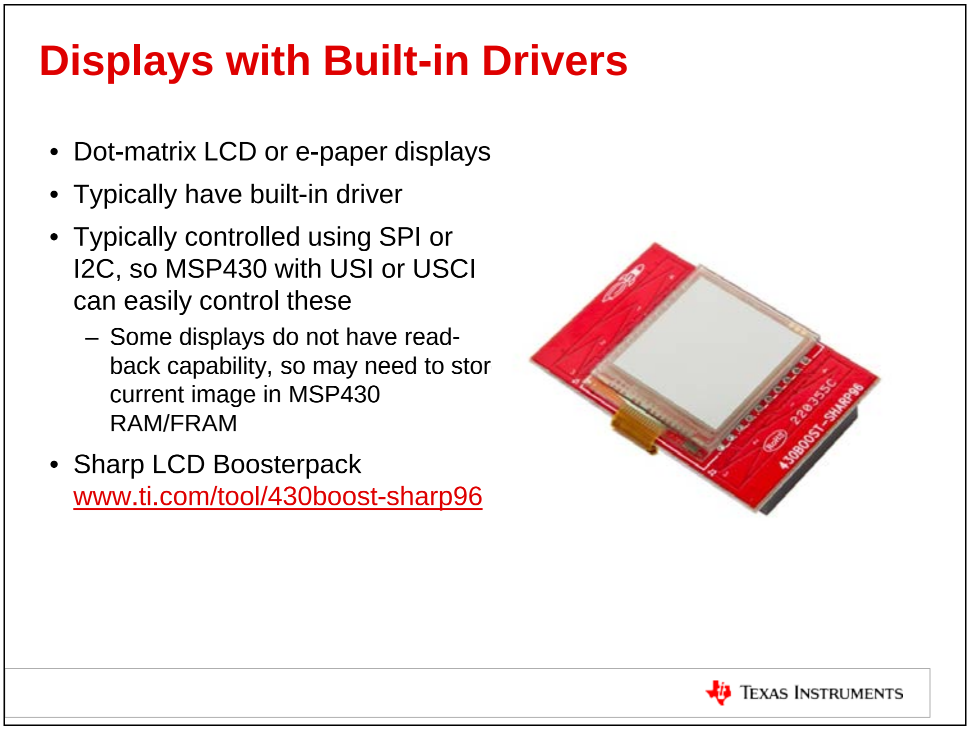
Another option is to use a use an external LCD controller – or maybe even an FPGA to generate the waveforms.

In the end, we can only think of one advantage with the bit-banging technique: you're not limited to only those devices with an LCD controller. Other than that, this isn't a very good option; it's more difficult and requires more power.

Bottom line – most folks choose one of the next two options.

Displays with Built-in Drivers

The second most popular solution is to pick a “smart display”… that is, one with a built-in controller.



As shown here, the popular Sharp LCD Boosterpack is a good example of a smart display. In this case, you would use one of the MSP430 serial ports to communicate with the display.

These are handy to use, as you only need to send commands and the display takes care of all the messy work. The only downside is that these displays may be slightly more expensive.

MSP430 LCD Peripherals

Finally, dedicated LCD controllers are the most efficient way to handle displays. Thankfully, many MSP devices have these on-chip peripherals.

MSP430 LCD portfolio

Parameter	LCD	LCD_A	LCD_B	LCD_C	LDC_E
Number of Segments	128/4-MUX	160/4-MUX	160/4-MUX	320/8-MUX	448/8-MUX
Number of LCD Pins			up to 4x46	up to 4 x 50 or 8 x 46	up to 4 x 60 or 8 x 56
Segment functionality against port pin selection	Minimum is group of 16	Selection done in groups of 4 segments	Individual selection can be done	Individual selection can be done	Individual selection can be done
COM/SEG Pin Assignments	COM Fixed	COM Fixed	COM Fixed	COM Fixed	Any LCD pin
LCD Clock selection	ACLK	ACLK	ACLK, VLO	ACLK, VLO	XT1, ACLK, VLO
Interrupt capabilities	NO	NO	YES (4 sources)	YES (4 sources)	YES (3 sources)
Individual segment blinking capabilities	NO	NO	YES	YES	YES
Prog. blinking frequency	N/A	N/A	YES	YES	YES
Dual memory display	NO	NO	YES	YES	YES
Charge Pump voltages	N/A	3 x V_{REF}	Programmable (15 Levels)		
Works in LPM3.5	No	No	No	No	Yes

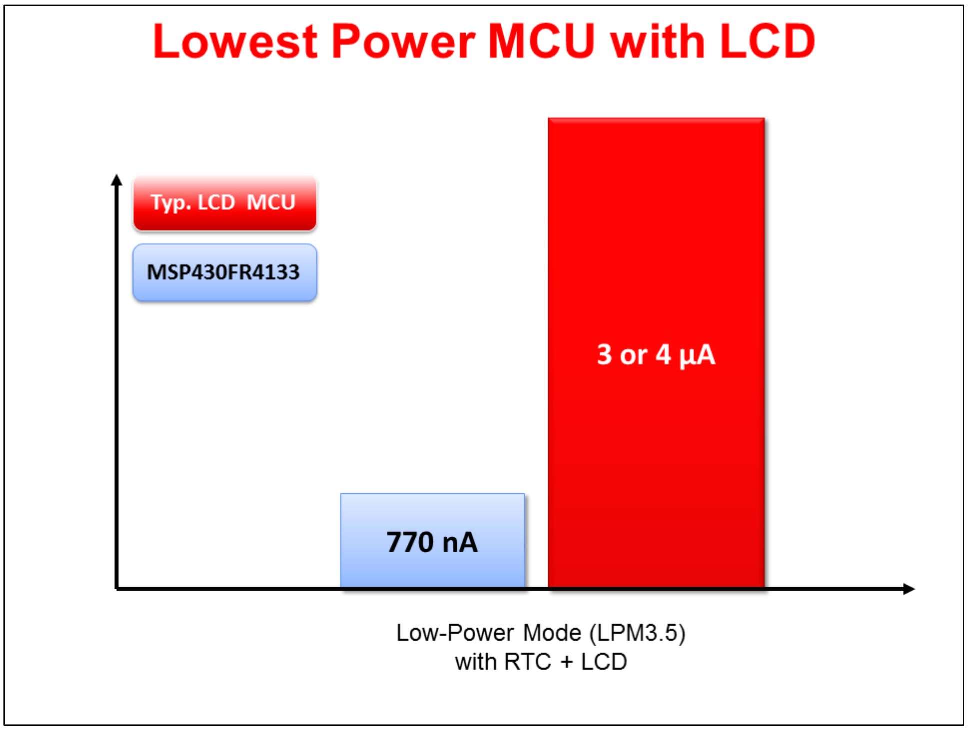
* For the full table, see the application note: *Designing With MSP430™ MCUs and Segment LCDs* (slaa654.pdf)

Low-cost, low-power, display applications have favored the MSP430 for many years. In that time, TI has continually tweaked the LCD peripheral; employing continuous improvements over the years.

The latest FRAM-based ‘FR4133 processor utilizes the newest controller, the “LCD_E” model.

Most of the features follow its recent predecessors, but for one big new advantage…

The new LCD controller can operate in the LPM3.5 … our “extreme” low-power mode.



This allows the ‘FR4133 to drive LCD displays (as well as a real-time clock) while dissipating less than 800 nano amps… AMAZING!

Implementing Display with ‘FR4133 LCD_E

In the final part of the discussion, we examine the steps required to implement a display using the FR4133’s LCD Controller.

There are four basic elements to an LCD-based design. Let’s begin with “Choosing a Display”.

Choose Display and Pin Layout

It’s difficult for us to prescribe the correct display for your application’s needs. The “glass” (that is, “display”) you choose will be based upon the needs of your system.

Maybe you’ll only need a single, 7-segment display… or, you might need to drive up to the full 256 segments the ‘FR4133 controller can handle.

The display shown here was captured from the FR4133’s Launchpad User’s Guide. Looking at its (albeit negligible specifications), we can see that it requires 3 volts; 4 COM lines and 3 bias voltages.

Select a Segmented LCD

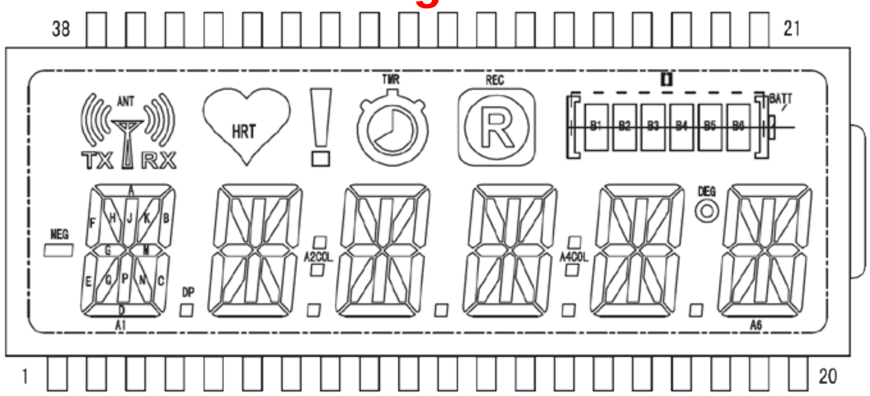


Figure 8. LCD Segment Layout

- **This display is found on the MSP-EXP430FR4133 Launchpad**
- **LCD datasheet defines:**
  - Driving Voltage = 3.0V
  - Duty cycle = ¼ (which implies 4-mux)
  - BIAS Voltages = 1/3
- **Of display’s 38 pins: 4 COM, 27 SEG (and 7 unused)**

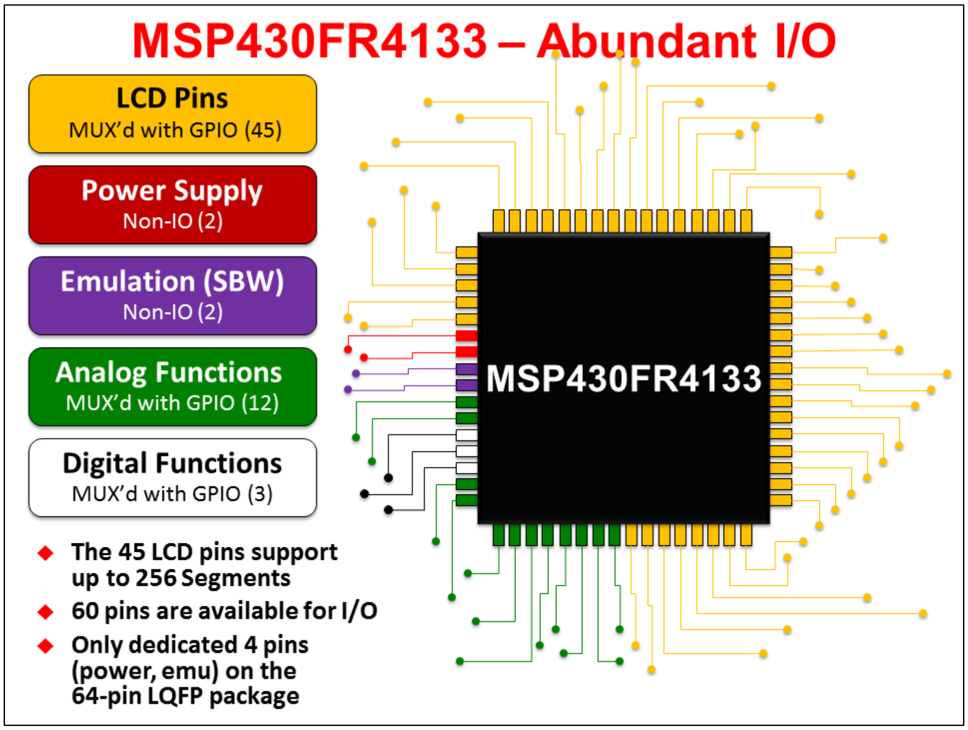
While the display has 38 pins, looking through the Launchpad’s documentation shows that only 31 of these are needed to drive all the segments.

Does the MSP430FR4133 have enough pins for this display?

Yes, this device has a plethora of I/O.

The LCD controller has been assigned 45 of the devices 64 pins. If we assign the 31 pins required by the Launchpad's display, that means we've got 14 left over that can be utilized for general purpose I/O.

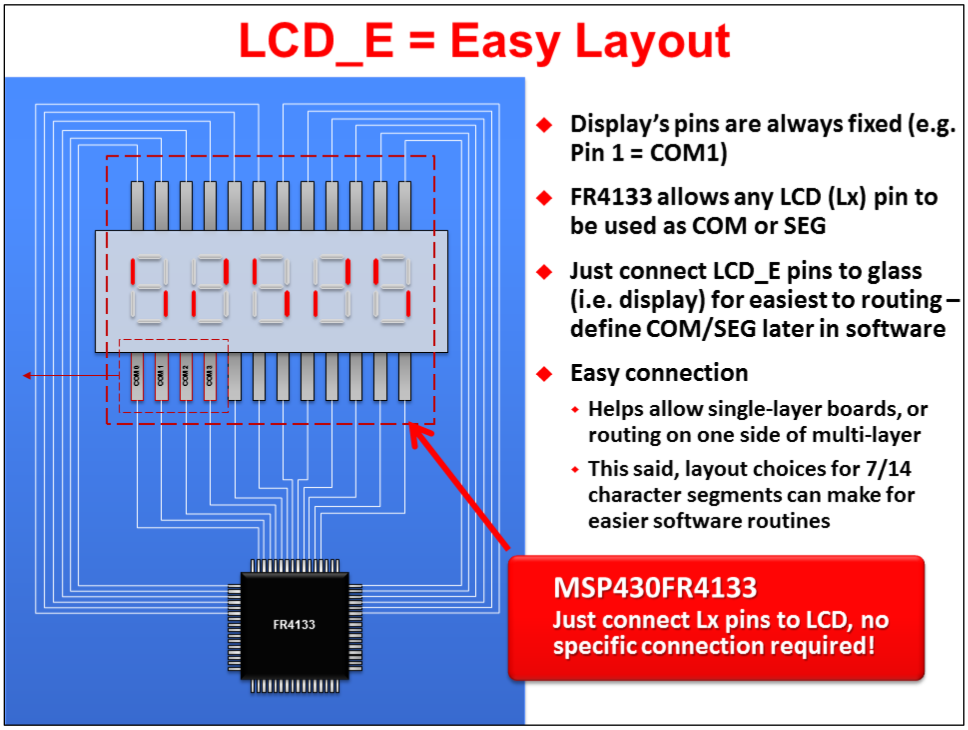
Of the remaining 19 pins found on this package, only 4 of them are dedicated and cannot be used for I/O. Two are used for the chip's power supply, the other two are dedicated for in-circuit emulation.



This is a very flexible device. In fact, turning to the next page, we'll see that the LCD pins themselves are not even "fixed".

The FR4133 does not provide hard definitions for any of its LCD pins. You can route them to the display’s pinout in whatever way makes your board layout the easiest.

Then, in software you can specify which LCD pin should be used for each of the COM lines.



LCD controllers don’t get any more flexible than this! Hardware designers REJOICE!

LCD Init Code

The next step in our LCD implementation is to initialize the LCD_E controller.

In this step we'll configure the pin assignments; select the N-mux mode; and specify the clock and voltage references needed for our application. This slide summarizes the init procedure.

The LCD's init function includes:

1. **Turn off the LCD controller**
2. **Set Lx pins to be used by LCD (versus GPIO)**
3. **Choose Mux and Timing modes**
   - **N-mux mode**
   - **Timing (clock source and rate)**
4. **Specify voltage reference and sources**
   - **Voltage reference source (V_{REF})** – should they be internal or external
   - **If internal, set the V_{REF} voltage level** – remember, this is programmable (15-levels)
   - **Source of bias voltages** – external resistor network or internal charge-pump
5. **Turn the LCD controller "on"**

We begin with turning off the LCD controller and assigning the pins.

Turn the LCD_E Off

The LCD_E controller should be turned off before changing many of its modes – so let’s start off our initialization routine by doing just that.

LCD Init – Turn off Controller

- **Using the LCD datasheet’s specs:**
  - Driving Voltage = 3.0V
  - Duty cycle = ¼ (which implies 4-mux)
  - BIAS Voltages = 1/3
- **Of display’s 38 pins: 4 COM, 27 SEG (and 7 unused)**

```
void initLCD(void) {
 // Turn off LCD
 LCD_E_off(LCD_E_BASE);
```

Allocate Pins

Next…looking at the LCD’s specifications from our earlier step, we need to assign 31 of the LCD (i.e. Lx) pins to the LCD controller.

Thankfully, driverlib has a function that lets us assign a whole range of LCD pins to the LCD controller.

LCD Init – Allocate Pins

- **Using the LCD datasheet’s specs:**
  - Driving Voltage = 3.0V
  - Duty cycle = ¼ (which implies 4-mux)
  - BIAS Voltages = 1/3
- **Of display’s 38 pins: 4 COM, 27 SEG (and 7 unused)**

```
void initLCD(void) {
 // Turn off LCD
 LCD_E_off(LCD_E_BASE);

 // Select range(s) of FR4133 LCD pins (Lx) to connect to LCD
 // Note: this means they won’t be available for GPIO
 LCD_E_setPinAsLCDFunctionEx( LCD_E_BASE,
     LCD_E_SEGMENT_LINE_0,     // assign range from pin L0
     LCD_E_SEGMENT_LINE_26     // through pin L26
 );
 LCD_E_setPinAsLCDFunctionEx( LCD_E_BASE,
     LCD_E_SEGMENT_LINE_36,    // assign range from pin L36
     LCD_E_SEGMENT_LINE_39     // through pin L39
 );
```

Since this display, and the layout routed by the Launchpad’s hardware designer, connected the MSP430 using two discontinuous ranges of GPIO pins, we were required to call the LCD_E_setPinsAsLCDFunctionEx() function twice.

Finally, since this controller allows any LCD pin to act as a COM pin, we need to specify which four pins (because we're using a 4-mux display) should be assigned as COM lines. Once again, a dedicated driverlib function makes this easy.

LCD Init – Assign COMs

- **Using the LCD datasheet's specs:**
  - Driving Voltage = 3.0V
  - Duty cycle = ¼ (which implies 4-mux)
  - BIAS Voltages = 1/3
- **Of display's 38 pins: 4 COM, 27 SEG (and 7 unused)**

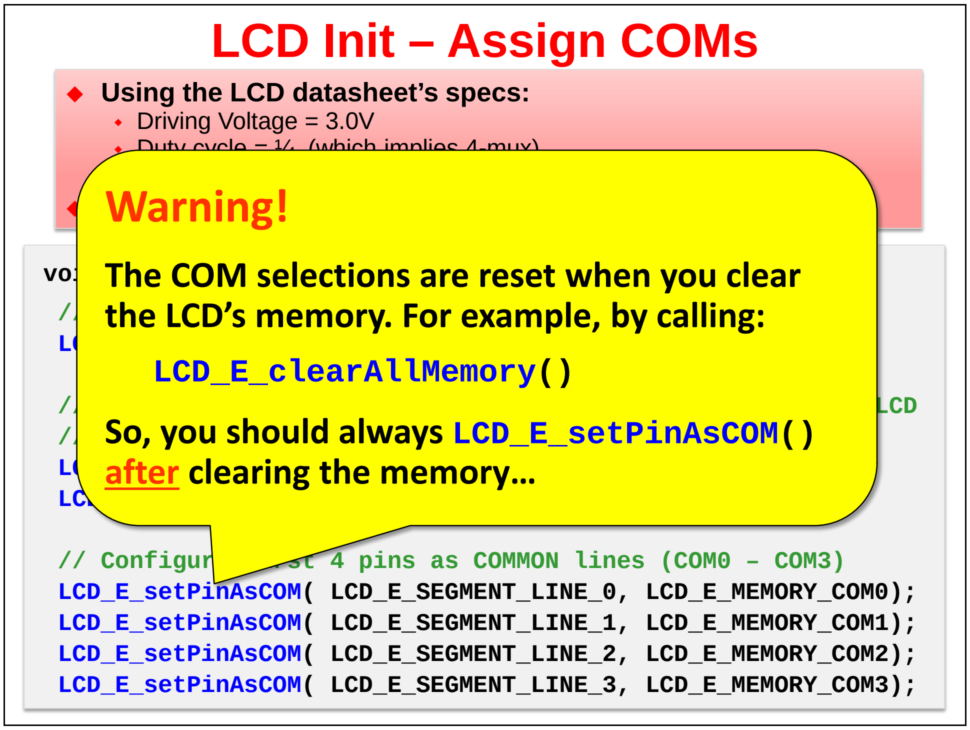
```
void initLCD(void) {
 // Turn off LCD
 LCD_E_off(LCD_E_BASE);

 // Select range(s) of FR4133 LCD pins (Lx) to connect to LCD
 // Note: this means they won't be available for GPIO
 LCD_E_setPinAsLCDFunctionEx( … );
 LCD_E_setPinAsLCDFunctionEx( … );

 // Configure first 4 pins as COMMON lines (COM0 – COM3)
 LCD_E_setPinAsCOM( LCD_E_SEGMENT_LINE_0, LCD_E_MEMORY_COM0);
 LCD_E_setPinAsCOM( LCD_E_SEGMENT_LINE_1, LCD_E_MEMORY_COM1);
 LCD_E_setPinAsCOM( LCD_E_SEGMENT_LINE_2, LCD_E_MEMORY_COM2);
 LCD_E_setPinAsCOM( LCD_E_SEGMENT_LINE_3, LCD_E_MEMORY_COM3);
```

Actually, the hardware layout made this very easy, we only need to assign the first four LCD pins as our COMmon lines.

Warning!

Here's something we stumbled into by accident as we wrote the workshop lab exercises.

It appears the COM pin selections are stored in the LCD's memory (which we'll talk about in a couple of minutes). Suffice it to say, if you clear the LCD's memory, you also erase the COM pin selections.



Therefore, if you clear the memory in your init routine, you should do it BEFORE you assign the COM pins. Do this and save yourself a couple of hours worth of debugging…

Choose Mux and Timing Modes

The 3rd step asks us to select the mux and timing modes…

The LCD's init function includes:

1. ✓ **Turn off the LCD controller**
2. ✓ **Set Lx pins to be used by LCD (versus GPIO)**
3. **Choose Mux and Timing modes**
   - **N-mux mode**
   - **Timing (clock source and rate)**
4. **Specify voltage reference and sources**
   - **Voltage reference source (V_{REF})** – should they be internal or external
   - **If internal, set the V_{REF} voltage level** – remember, this is programmable (15-levels)
   - **Source of bias voltages** – external resistor network or internal charge-pump
5. **Turn the LCD controller "on"**

As with many of the peripherals we've configured throughout this workshop, we begin by creating an *initParams* variable and setting it equal to its default parameters.

For our Launchpad's display, we're only going to change two of the parameters: *clockDivider* and *muxrate*.

LCD Init – Configure Modes

- **Using the LCD datasheet's specs:**
  - Driving Voltage = 3.0V
  - Duty cycle = ¼ (which implies 4-mux)
  - BIAS Voltages = 1/3
- **Of display's 38 pins: 4 COM, 27 SEG (and 7 unused)**

```
// Initialze LCD Clock and Mux mode
LCD_E_initParam initParams = LCD_E_INIT_PARAM;
initParams.clockDivider = CLOCKDIV_3;  // Set frame rate
```

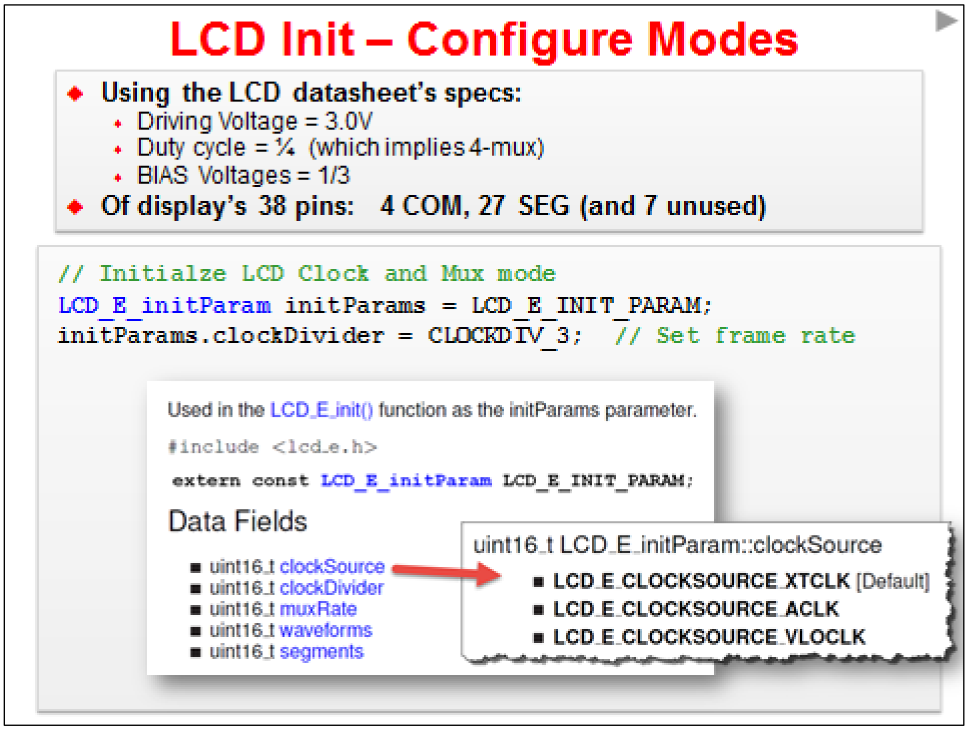
clockDivider

As shown on the previous slide, we want to divide the input clock by 3, to get our frame-rate close to 60 Hz.

Wait a second, what clock source are we using? In other words, what's getting divided by 3?

To figure this out, let's look at the DriverLib User's Guide. (Although, you could also look in the "lcd_e.h" header file.)

The DriverLib User's Guide (and header file) shows us that unless we specify otherwise, the LCD_E peripheral will be configured to use the external clock source (XT1).



That means our clock source is calculated from 32KHz and is pre-divided by 3.

Note: Calculating the frame-rate from these values, though, requires crunching two simple equations. We'll leave the details of this until the upcoming lab exercise.

Notes:

muxrate

Since, according to the User’s Guide, the DriverLib defaults to Static (that is, 1-mux mode), we need to set the controller to run in 4-mux mode to match our display’s specifications.

With those two changes to *initParams*, you can go ahead and use it to initialize the LCD_E controller with LCD_E_init().

LCD Init – Configure Modes

- **Using the LCD datasheet’s specs:**
  - Driving Voltage = 3.0V
  - Duty cycle = ¼ (which implies 4-mux)
  - BIAS Voltages = 1/3
- **Of display’s 38 pins: 4 COM, 27 SEG (and 7 unused)**

```
// Initialze LCD Clock and Mux mode
LCD_E_initParam initParams = LCD_E_INIT_PARAM;
initParams.clockDivider = CLOCKDIV_3;  // Set frame rate
initParams.muxRate      = LCD_E_4_MUX; // Select mux
LCD_E_init(LCD_E_BASE, &initParams);
```

Specify Voltage Reference/Source

The last thing we need to do before turning the display "on" is to configure the bias voltages that will be used to drive the LCD. The 3 sets of options (3 bullets under #4 in the diagram) listed here are summarized in the Device User's Guide. The User's Guide defines 6 modes, which describe the various permutations…

The LCD's init function includes:

1. ✓ **Turn off the LCD controller**
2. ✓ **Set Lx pins to be used by LCD (versus GPIO)**
3. ✓ **Choose Mux and Timing modes**
   - **N-mux mode**
   - **Timing (clock source and rate)**
4. **Specify voltage reference and sources**
   - **Source of bias voltages** – external resistor network or internal charge-pump
   - **Voltage reference source (V_{REF})** – should they be internal or external
   - **If internal, set the V_{REF} voltage level** – remember, this is programmable (15-levels)
5. **Turn the LCD controller "on"**

As you can see, the 6 modes are broken up into 2 groups. One uses external resistors to create the Bias voltages; the other set uses the LCD controller’s charge pump.

Summary: LCD_E Voltage Modes

Mode 0: Charge Pump Disabled
(Requires external resistor ladder across pins: R33 → R23 → R13 → GND)

Mode	Charge Pump	Voltage Bias ‘Source’	Contrast Control
0a	Disabled		
0b	Disabled		

Mode 1, 2, 3, 4: Charge Pump Enabled
(Requires external capacitors between pins: R33 → GND, R23 → GND, R13 → GND)

Mode	Charge Pump	Voltage Bias ‘Source’	Contrast Control
1	Enabled		
2	Enabled		
3	Enabled		
4	Enabled		

Let’s take a brief look at each mode – highlighting its key characteristics. We’ll then summarize by “filling-in” this table.

Mode 0a

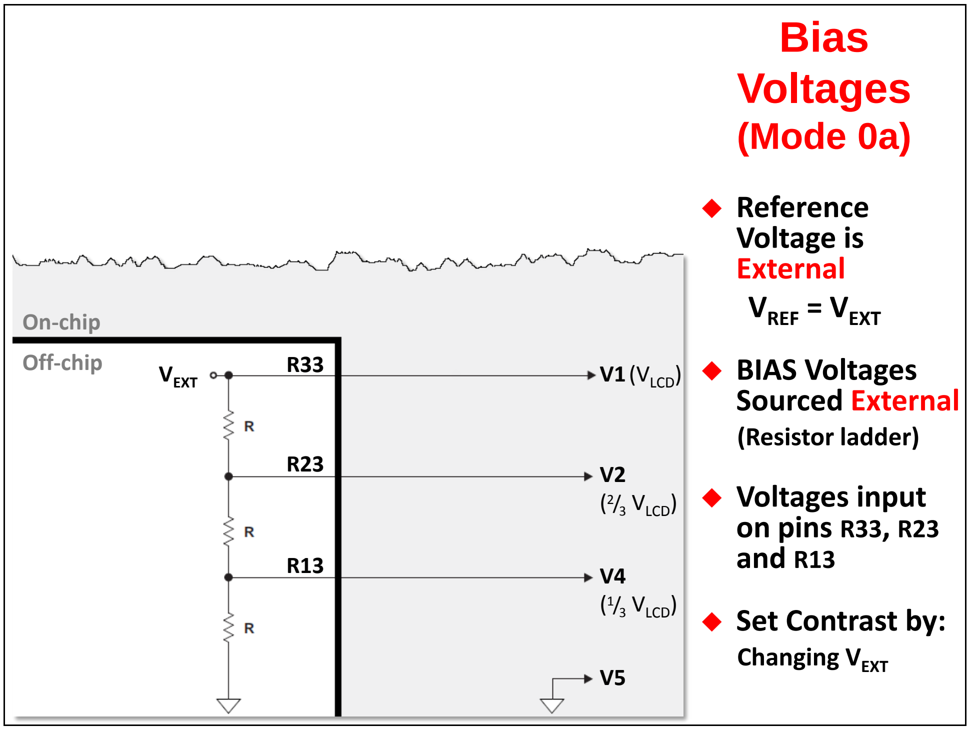
Note: As a point of reference, we've highlighted the parts of the diagram that are found "on-the-device" using a gray background. The "off-chip" portions are found in the lower-left corner in white. Therefore, the lines represented by R33, R23, and R13 represent pins on the MSP430FR4133 device.

OK, with that stated, what are the unique points about Mode 0a…

First of all, you'll notice that the 3 bias voltages are created using an external ladder of resistors. The top one (highest voltage) is call V_{LCD}, while the 2nd one represents 2/3 the value of V_{LCD} and finally the V4 voltage only has 1/3 of the original V_{LCD} value.

The other salient point is that the Voltage Reference source is "external". In other words, V_{LCD} equals the voltage applied to pin R33.

Likewise, the voltages created by the resistor network become the input values for V2 and V4.

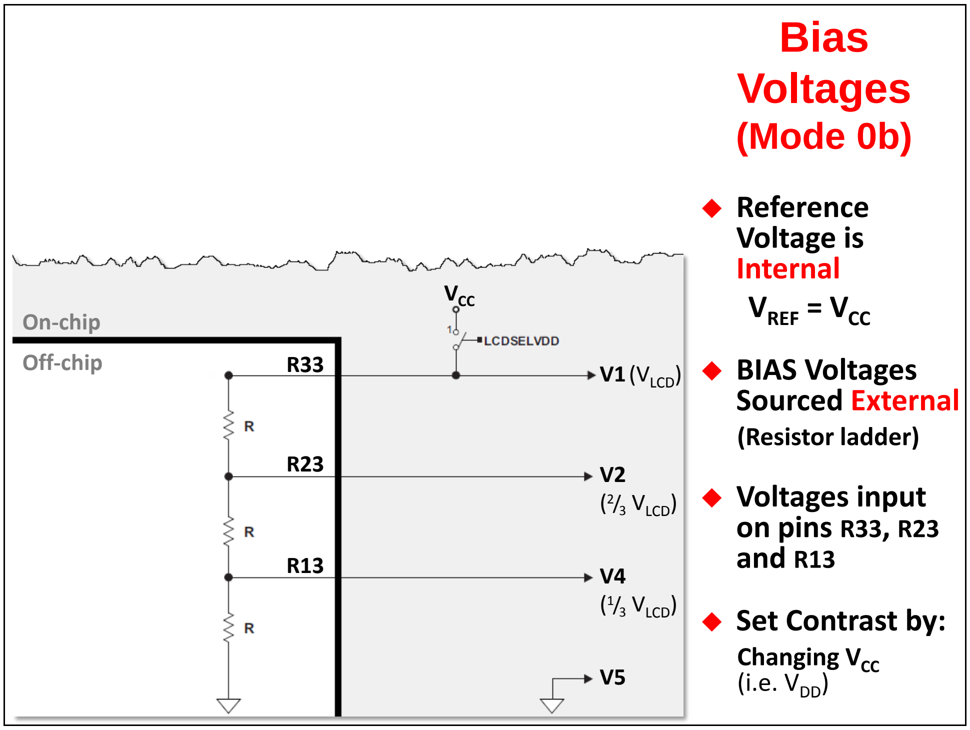


In Mode 0a, you need to change the value on R33 (i.e. V_{EXT}) in order to adjust the contrast of the display.

Mode 0b

Mode 0b is very similar to 0a, except that V_{LCD} is referenced internally – from V_{CC} – rather than externally.

The only other difference this creates is that to adjust the contrast of the display, you need to vary V_{CC} rather than $V_{EXT.}$



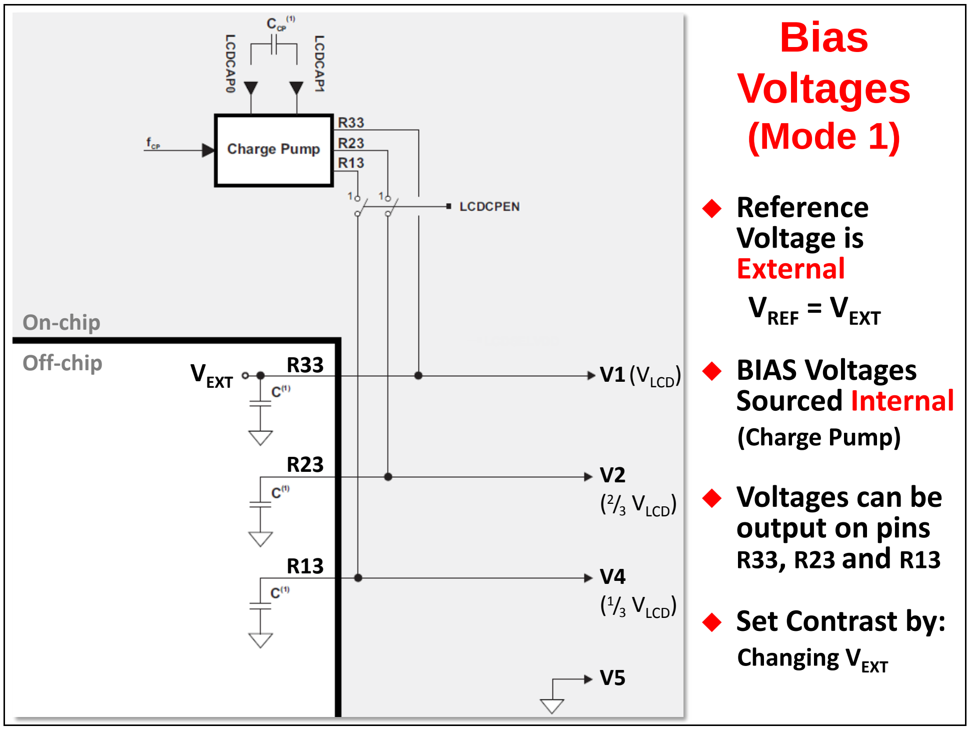
Modes 1 thru 4

The rest of the modes (Modes 1 thru 4) differ in that we use the internal charge pump to generate the bias voltages.

The big difference between Modes 1 and 2 comes down to the source of our voltage reference. Similar to Modes 0a and 0b, one is external and the other is internal.

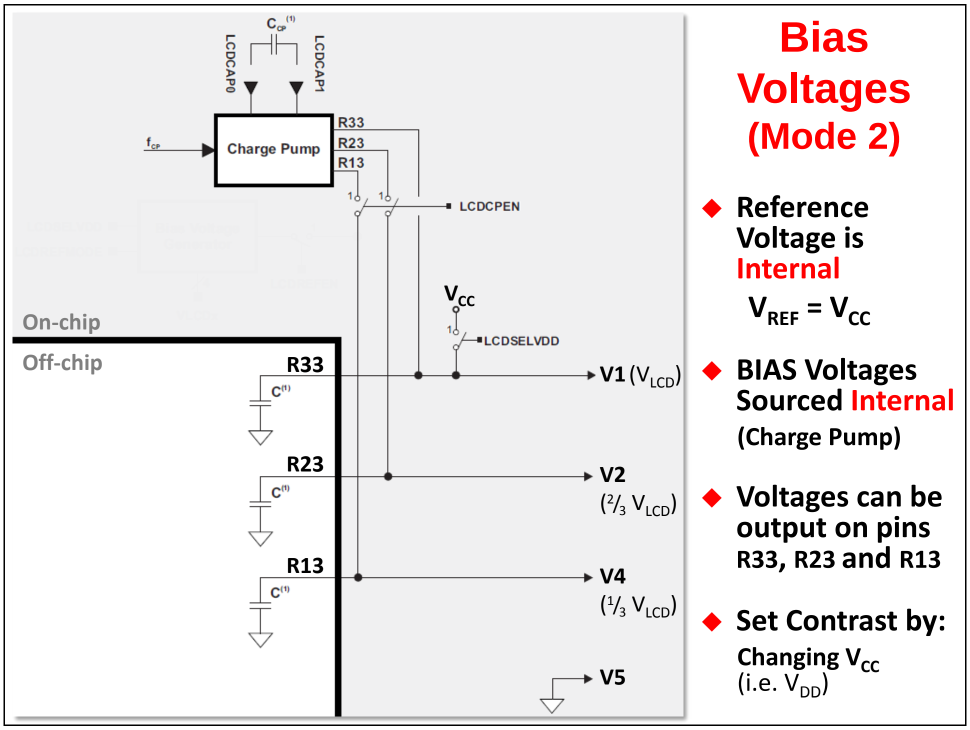
Mode 1

As we can see, Mode 1 uses an external voltage V_{EXT} applied to pin R33.



Mode 2

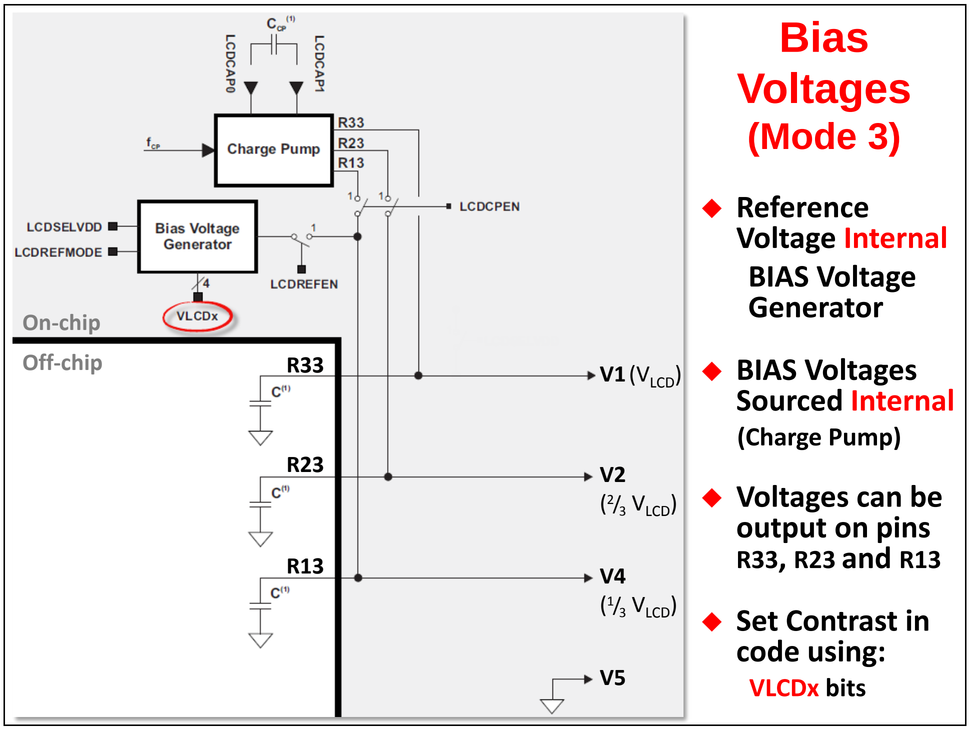
While Mode 2 uses the internal V_{CC} voltage as reference.



Mode 3

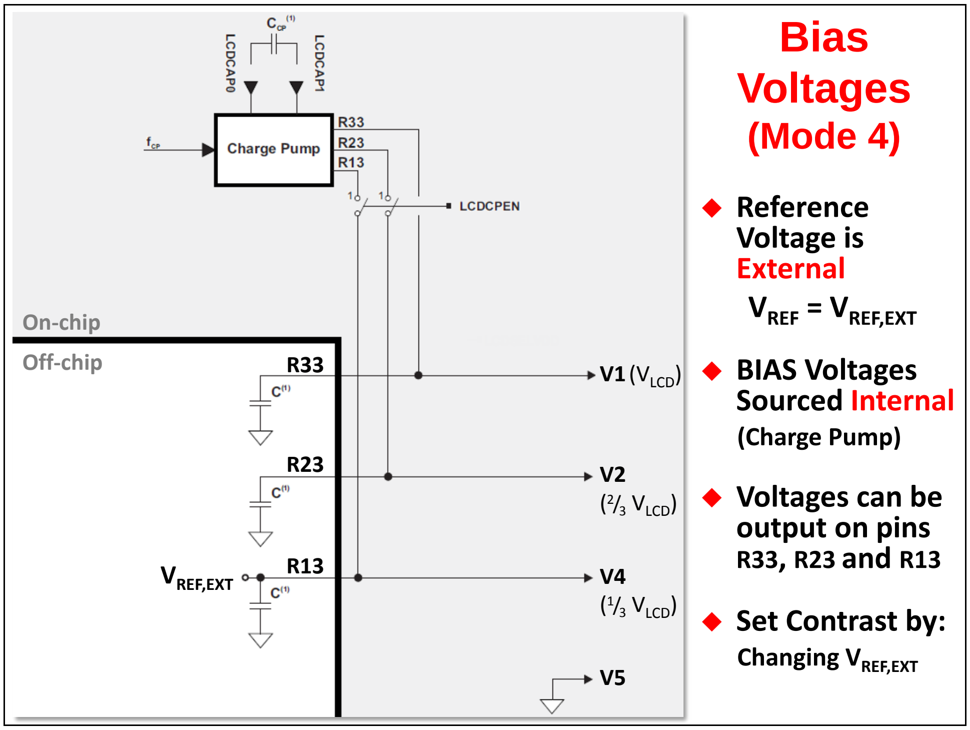
Mode 3 is different from all the other modes in that it uses a separate internal Voltage Reference – called the "Bias Voltage Generator". The benefit here is that the Bias Voltage Generator can be configured to output any one of 16 different voltages. (From 2.6V up to 3.5V.)

The beauty of this is that we can now control the contrast via software. That is, by tweaking the VLCD bits in the LCD Voltage control register, we can easy vary the display's contrast.



Mode 4

Finally, Mode 4 is similar to Mode 2 in that the voltage reference is once again external. In this case, though, the reference is applied to the lowest voltages (at pin R13) rather than the highest voltage (at pin R33).



Summarize Voltage Modes

To summarize the various modes, we’ve completed the earlier table.

Summary: LCD_E Voltage Modes

Mode 0: Charge Pump Disabled

(Requires external resistor ladder across pins: R33 → R23 → R13 → GND)

Mode	Charge Pump	Voltage Bias ‘Source’	Contrast Control
0a	Disabled	R33 sourced with V_{EXT} (LCDSELVDD = 0)	Changing V_{EXT}
0b	Disabled	R33 sourced with V_{CC} (LCDSELVDD = 1)	Changing V_{DD}/V_{CC} (1.8 to 3.6V)

Mode 1, 2, 3, 4: Charge Pump Enabled

(Requires external capacitors between pins: R33 → GND, R23 → GND, R13 → GND)

Mode	Charge Pump	Voltage Bias ‘Source’	Contrast Control
1	Enabled	R33 sourced with V_{EXT} (LCDSELVDD = 0)	Changing V_{EXT}
2	Enabled	R33 sourced with V_{CC} (LCDSELVDD = 1)	Changing V_{DD}/V_{CC} (1.8 to 3.6V)
3	Enabled	Internal V_{REF} from Bias Voltage Generator R13 sourced with internal ($V_{REF,INT}$) (LCDSELVDD = 0 prevents V_{cc} from driving R33) (V_{EXT} is not connected)	Software programmable by changing VLCD bits *(One reason that out-of-box demo used Mode 3)*
4	Enabled	R13 sourced with external $V_{REF,EXT}$ (LCDSELVDD = 0 prevents V_{cc} from driving R33)	Changing V_{REF} (from 0.8 to 1.2V)

Subtly pointed out here, the out-of-box demo that comes with the ‘FR4133 Launchpad uses *Mode 3*. They chose this because the voltage references – and contrast control – were all handled internally, under software control. In fact, for these reasons, this is probably the most popular mode amongst users.

Voltage Init Code

Applying what we just learned to our code example, we set the VLCDSource voltage as needed for “Mode 3”.

Then, we set the VLCD bits (that is, the VLCD Voltage) to 2.96 volts. (The apps team came up with this value empirically, by trying various voltages and observing what looked best.)

Finally, we turn on the internal Charge Pump and set its frequency.

LCD Init – Configure Modes

- **Using the LCD datasheet’s specs:**
  - Driving Voltage = 3.0V
  - Duty cycle = ¼ (which implies 4-mux)
  - BIAS Voltages = 1/3
- **Of display’s 38 pins: 4 COM, 27 SEG (and 7 unused)**

```
// Initialze LCD Clock and Mux mode
LCD_E_initParam initParams = LCD_E_INIT_PARAM;
initParams.clockDivider = CLOCKDIV_3;  // Set frame rate
initParams.muxRate      = LCD_E_4_MUX; // Select mux
LCD_E_init(LCD_E_BASE, &initParams);

// Configure Voltage Sources for the LCD Controller (Mode 3)
LCD_E_setVLCDSource(INTERNAL_REF_VOLTAGE, EXTERNAL_SUPPLY_V…
LCD_E_setVLCDVoltage(LCD_E_REFERENCE_VOLTAGE_2_96V);
LCD_E_enableChargePump(LCD_E_BASE);
LCD_E_setChargePumpFreq(LCD_E_BASE, LCD_E_CHARGEPUMP_FREQ_16);
```

Turn On the LCD Controller

That takes us to the final command in our initialization routine… Turning "on" the LCD controller…

The LCD's init function includes:

- **Turn off the LCD controller**
- **Set Lx pins to be used by LCD (versus GPIO)**
- **Choose Mux and Timing modes**
  - **N-mux mode**
  - **Timing (clock source and rate)**
- **Specify voltage reference and sources**
  - **Voltage reference source (V_{REF})** – should they be internal or external
  - **If internal, set the V_{REF} voltage level** – remember, this is programmable (15-levels)
  - **Source of bias voltages** – external resistor network or internal charge-pump

5. **Turn the LCD controller "on"**

One of the great advantages to using DriverLib… it's not hard to figure out what "LCD_E_on" means!

LCD Init – Configure Modes

- **Using the LCD datasheet's specs:**
  - Driving Voltage = 3.0V
  - Duty cycle = ¼ (which implies 4-mux)
  - BIAS Voltages = 1/3
- **Of display's 38 pins: 4 COM, 27 SEG (and 7 unused)**

```
// Initialze LCD Clock and Mux mode
LCD_E_initParam initParams = LCD_E_INIT_PARAM;
initParams.clockDivider = CLOCKDIV_3;  // Set frame rate
initParams.muxRate      = LCD_E_4_MUX; // Select mux
LCD_E_init(LCD_E_BASE, &initParams);

// Configure Voltage Sources for the LCD Controller
LCD_E_setVLCDSource(INTERNAL_REF_VOLTAGE, EXTERNAL_SUPPLY_V…
LCD_E_setVLCDVoltage(LCD_E_REFERENCE_VOLTAGE_2_96V);
LCD_E_enableChargePump(LCD_E_BASE);
LCD_E_setChargePumpFreq(LCD_E_BASE, LCD_E_CHARGEPUMP_FREQ_16);

// Turn LCD on
 LCD_E_on(LCD_E_BASE);
```

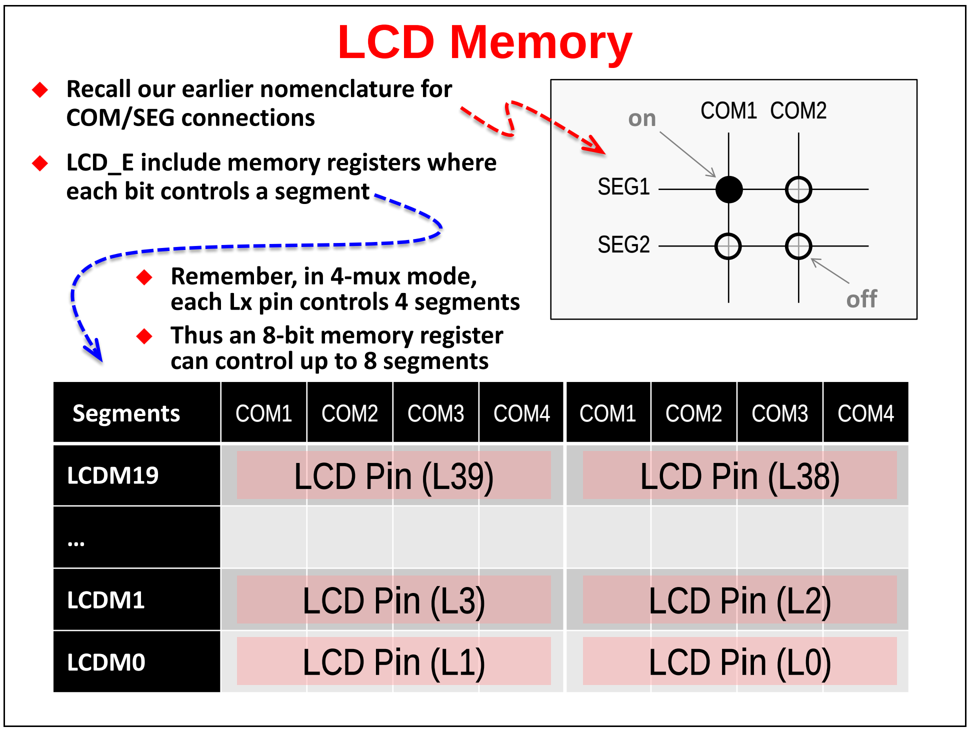
Controlling Segments

OK, now that we've got the LCD controller initialized and turned on… what do you want to display? In other words, which segments do you want turned "on"?

Remember the shorthand we showed earlier in the chapter? Let's see how this translates to actually driving the LCD display…

You might remember that our LCD controller has its own memory - enough memory to specify the on/off value for each segment line.

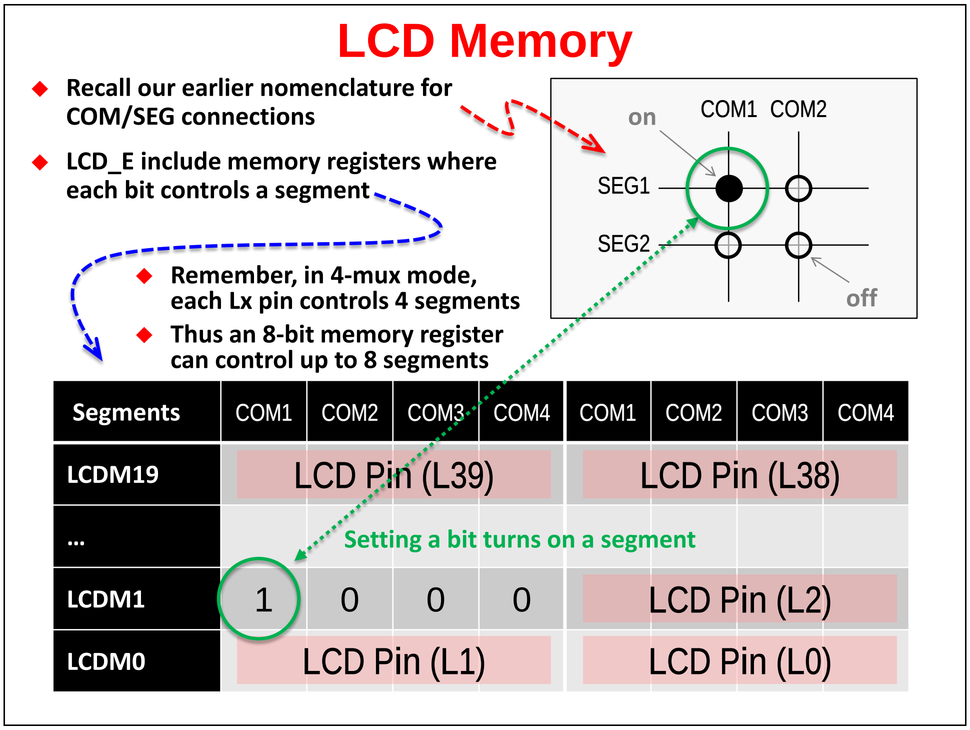
Since we're using a 4-mux display, each segment line is associated with 4 COM pins. Therefore, each byte of LCD memory lets us control two LCD segment pins. (We' tried to highlight this fact in the following diagram.)



From this, we can tell that 20 locations in LCD memory can handle the 40 LCD pins (4-mux mode).

How do we "set" a bit… for example, how would we set SEG1 at COM1 to "on"?

All you need to do is set the associated bit in the LCD memory. It really can't get much easier than this.



DriverLibrary makes this easy to do, just call the LCD_E_setMemory() function:

```
LCD_E_setMemory( LCD_E_BASE, 1, 0x80 );
```

In fact, DriverLib makes it even easier because we don't even have to look up or calculate the hex address locations for the LCD memory; instead, you can use LCDM1 – or just "1" – to indicate the appropriate LCD memory location.

Turning to the 'FR4133 Launchpad, how did the hardware designer connect the LCD pins to the display?

Looking in the Launchpad's User Guide, we find this table. The designer did a great job of summarizing all of the details we need to drive the display.

FR4133 Launchpad LCD Connections

- **Launchpad User's Guide nicely describes pin connections:**
  - **e.g. FR4133 pin (L1), Port Pin (P7.1), LCD's pin (23)**
  - **Along with which segment each LCDMEM bit enables**

LCDMEM	Port Pin	FR4133 Pin	LCD Pin	COM3	COM2	COM1	COM0	Port Pin	FR4133 Pin	LCD Pin	COM3	COM2	COM1	COM0
LCDM19	P5.7	L39	38	A6H	A6J	A6K	A6P	P5.6	L38	37	A6Q	TX	A6N	RX
LCDM18	P5.5	L37	36	A6A	A6B	A6C	A6D	P5.4	L36	35	A6E	A6F	A6G	A6M
LCDM17	P5.3	L35						P5.2	L34					
LCDM16	P5.1	L33						P5.0	L32					
LCDM15	P2.7	L31						P2.6	L30					
LCDM14	P2.5	L29						P2.4	L28					
LCDM13	P2.3	L27						P2.2	L26	34	B5	B3	B1	[]
LCDM12	P2.1	L25	33	B6	B4	B2	BATT	P2.0	L24	32	TMR	HRT	REC	!
LCDM11	P6.7	L23	16	A4H	A4J	A4K	A4P	P6.6	L22	15	A4Q	A4COL	A4N	A4DP
LCDM10	P6.5	L21	14	A4A	A4B	A4C	A4D	P6.4	L20	13	A4R	A4F	A4G	A4M
LCDM9	P6.3	L19	12	A3H	A3J	A3K	A3P	P6.2	L18	11	A3Q	ANT	A3N	A3DP
LCDM8	P6.1	L17	10	A3A	A3B	A3C	A3D	P6.0	L16	9	A3R	A3F	A3G	A3M
LCDM7	P3.7	L15	8	A2H	A2J	A2K	A2P	P3.6	L14	7	A2Q	A2COL	A2N	A2DP
LCDM6	P3.5	L13	6	A2A	A2B	A2C	A2D	P3.4	L12	5	A2E	A2F	A2G	A2M
LCDM5	P3.3	L11	4	A1H	A1J	A1K	A1P	P3.2	L10	3	A1Q	NEG	A1N	A1DP
LCDM4	P3.1	L9	2	A1A	A1B	A1C	A1D	P3.0	L8	1	A1E	A1F	A1G	A1M
LCDM3	P7.7	L7	20	A5H	A5J	A5K	A5P	P7.6	L6	19	A5Q	DEG	A5N	A5DP
LCDM2	P7.5	L5	18	A5A	A5B	A5C	A5D	P7.4	L4	17	A5E	A5F	A5G	A5M
LCDM1	P7.3	L3	21	COM3	-	-	-	P7.2	L2	22	-	COM2	-	-
LCDM0	P7.1	L1	23	-	-	COM1	-	P7.0	L0	24	-	-	-	COM0

Table 4. LCD to MSP430 Connections

It shows us the LCD memory locations, the LCD pin – as well as its associated GPIO Port and Pin numbers. It even shows the pin on the display that it's connected to.

Finally, each of the segments are written into the table so that we can see what bits must be set in order to turn a specific segment on.

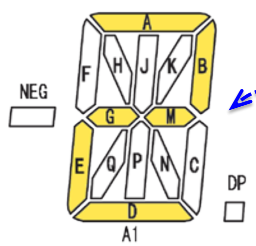
Let's focus on the first character in our display.

Here we've highlighted the two memory locations that represent the first character… which is called "A1".

Choosing Layout for Easier Software

LCDM6	P3.5	L13	6	A2A	A2B	A2C	A2D	P3.4	L12	5	A2E	A2F	A2G	A2M
LCDM5	P3.3	L11	4	A1H	A1J	A1K	A1P	P3.2	L10	3	A1Q	NEG	A1N	A1DP
LCDM4	P3.1	L9	2	A1A	A1B	A1C	A1D	P3.0	L8	1	A1E	A1F	A1G	A1M
LCDM3	P7.7	L7	20	A5H	A5J	A5K	A5P	P7.6	L6	19	A5Q	DEG	A5N	A5DP
LCDM2	P7.5	L5	18	A5A	A5B	A5C	A5D	P7.4	L4	17	A5E	A5F	A5G	A5M
LCDM1	P7.3	L3	21	COM3	-	-	-	P7.2	L2	22	-	COM2	-	-
LCDM0	P7.1	L1	23	-	-	COM1	-	P7.0	L0	24	-	-	-	COM0

- **Notice how we can set an entire character (e.g. "2") by writing one or two consecutive LCD Memory locations (LCD4 and LCD5)**
- **Hence the comment: *"Choosing a good pin layout can ease software"***
- **Make programming easier by pre-defining segment values for digits; you can also define the alphabet in a similar fashion**




```
const char digit[10] =
{
    0xFC,   /* "0" */
    0x60,   /* "1" */
    0xDB,   /* "2" */
    0xF3,   /* "3" */
    0x67,   /* "4" */
    0xB7,   /* "5" */
    ...
```

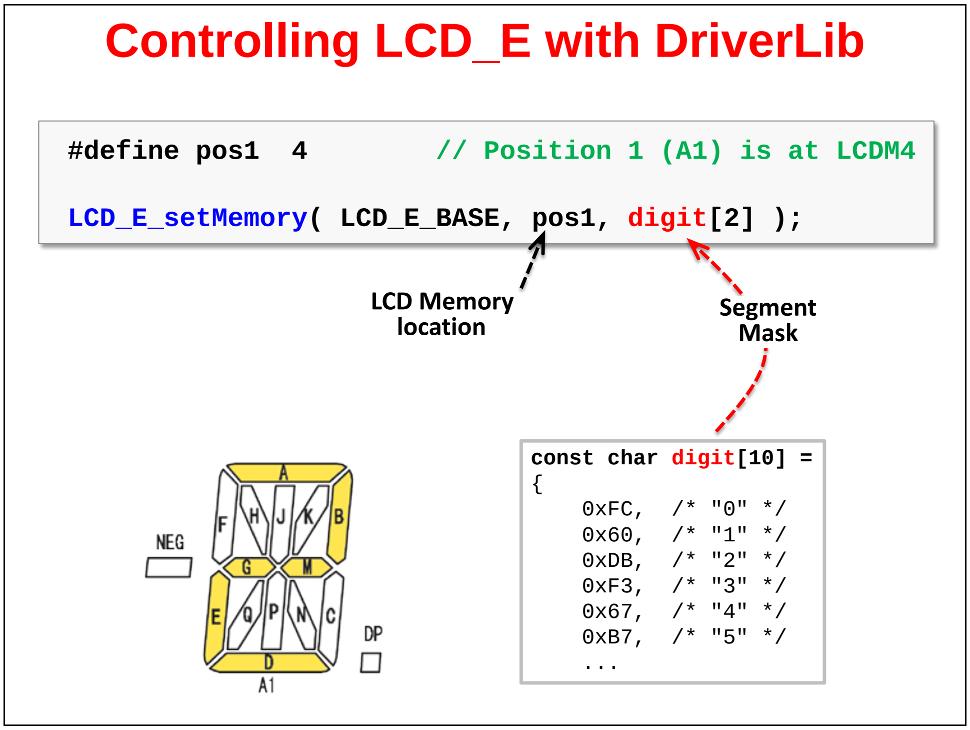
See how each of the segments that make up A1 are assigned their own SEG/COM bits?

It appears that if we want to display the number "2" on "A1", we'll need to enable segments "A1A", "A1B" as well as A1… M, G, E and D.

We could poke these bits individually, but it's more common to do this the easy way…

Creating masks for each digit and character is an easier – and less error prone – way to display an alphanumeric character.

Going back to our DriverLib “setMemory” example, this might look like:



Here, the setMemory example is going to apply the “digit[2]” mask value to “position 1”.

Where each of the six alphanumeric elements of the display are called “position” 1, 2, 3, 4, 5, and 6.

BTW, why is position 1 = 4? Look back to the launchpad’s “*LCD to MSP430 Connections*” table to figure out why…

DriverLib allows us to do more than just "set" a memory location.

We can also "clear" the memory location, as well as "toggle" its bit values.

Set/Clear Segments with DriverLib

```
#define pos1  4        // Position 1 (A1) is at LCDM4

LCD_E_setMemory( LCD_E_BASE, pos1, digit[2] );
```

Turn segments on/off:

- LCD_E_setMemory() Overwrites LCDMx memory with provided value
- LCD_E_clearMemory() Clears the specified bits of LCDMx register
- LCD_E_toggleMemory() Toggles all 8-bits in bits in LCDMx register
- LCD_E_updateMemory() Sets the specified bits of LCDMx (LCDMx |= mask)

Finally, the "update" function can be a handy alternative to "set". Whereas "set" clears all the bits in the memory location before setting specified mask value; "update" just OR's the mask into the memory location. In other words, "update" leaves alone any bits that were already set in that memory.

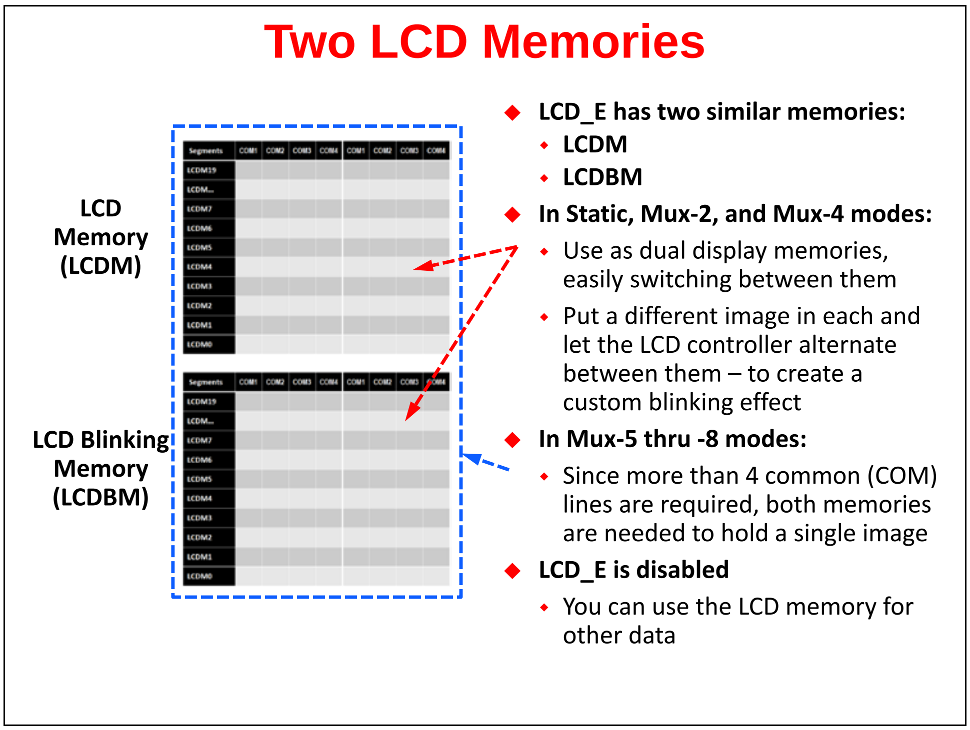
Dual Memories & Blinking

We’re almost done with the chapter discussion. We’ve just got a couple of additional features to go over…

Just a couple of minutes ago, we discussed the fact that our LCD peripheral has its own memory.

The ‘FR4133 device actually has 40 memory locations.

Mux-4 (or lower) only requires 20 memory locations, because we can control 2 pins per location. This means that we can make two memory blocks out of the 40 locations. As shown here, the first one is called the “LCD Memory” (abbreviated LCDM); while the second is called “LCD Blinking Memory” (or LCDBM). At any given time, the display can be showing either one of them.



This works out great because you can put a different ‘image’ in each memory block and then switch between them – which lets you create custom blinking patterns. In fact, the controller can automatically switch between the two memories on its own – while the rest of the microcontroller stays asleep. VERY POWERFUL… well, POWERFUL in an ULTRA LOW-POWER way.

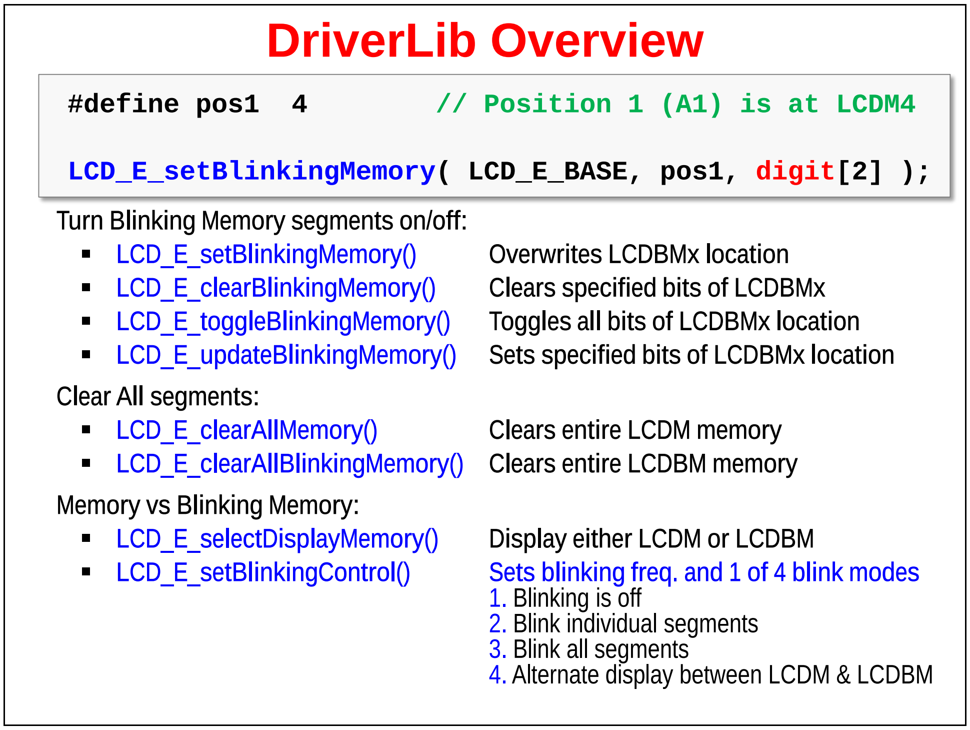
On the other hand, if your display is big enough to require Mux-5 or above, all of the memory is required to configure a single display. In other words, you only get the one “LCD Memory” block.

This doesn’t mean you can’t do blinking on big displays… there are other built-in blinking modes, but you’ll lose the ability to blink between two custom patterns without the CPU having to wake up and perform the switch.

DriverLib LCD_E Summary

Here we summarize the DriverLib functions for manipulating the memory and segments.

To our earlier four functions, we've added two functions to clear the memory… as well as two others which deal with blinking and memory block selection.



The lower-right hand corner of the slide shows the four options for the "setBlinkingControl" function. Of course, you can turn blinking "off". But, you can also blink individual segments, all segments, or – as we just described on the previous slide – alternate between the two memory blocks.

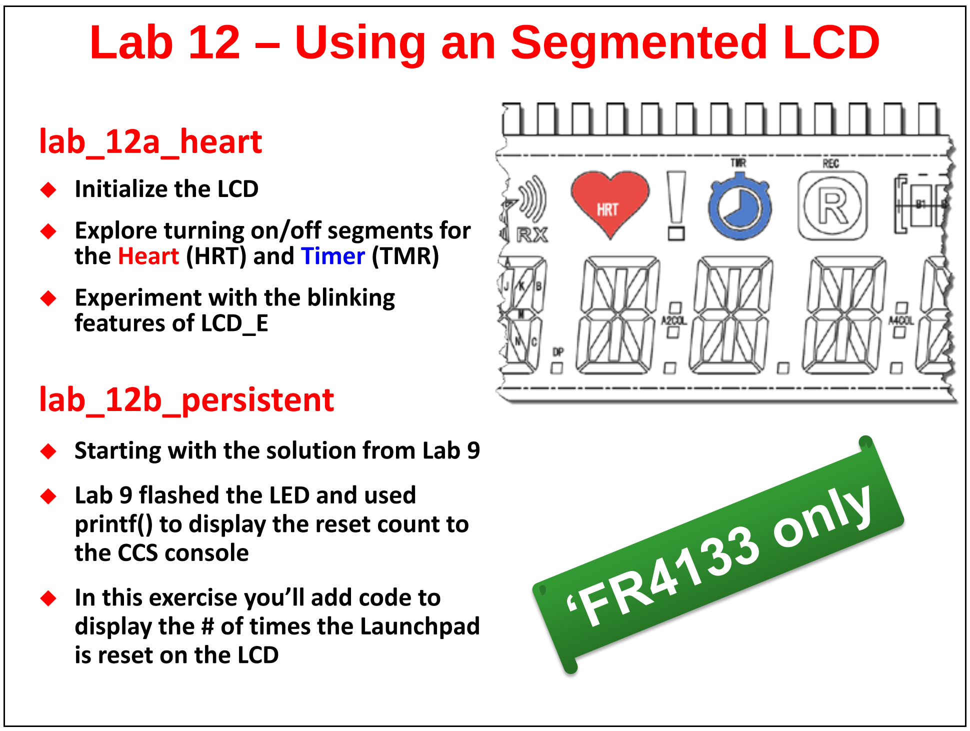
Using Segmented Displays (LCD)

There are two parts to this exercise – A and B.

lab_12a_heart

In the first part, we'll explore turning on and off the "heart" and "timer" segments. We'll also play with blinking these segments – even alternating between them.

After filling out a few questions in the lab worksheet, we'll import and edit the lab project. In the debugging phase of this lab, we'll ask you to set a breakpoint, run to that point, and then single-step through your code. This seems to be the easiest way to watch how the display responds to the various "set", "clear", "update" and blinking functions.



lab_12b_persistent

Finally, if you have time, you can try the next part of the lab. Here we'll go back and add some code to the Persistent FRAM exercise.

You might recall that we used FRAM to "persist" a variable. That is, we were able to retain the value even after resetting the device. In fact, we used this value to track how many times the board was reset; then flashed the LED that many times.

Now that we know how to use your board's display, why don't we go ahead and show the count value on the LCD?

Have fun with the lab!

Lab 12a – A Launchpad with Heart

Using an LCD requires a few of steps:

- **Planning** – figuring out what LCD you need for your application; verifying the LCD controller can operate that display (often called 'glass'); and, implementing the hardware design. For this exercise, we assume these steps have been completed and that you have a board – such as the 'FR4133 Launchpad – that is ready for software.
- **Initialization** – like most other peripherals, we have to choose the proper modes of operation for our application and ready the device. Here are the basic initialization steps:
  - Turn off LCD_E
  - Set Lx I/O pins needed by controller
  - Setup the input clocking and frame rate (and enable the segment pins)
  - Configure voltage requirements – including enabling the built-in Charge Pump, if used.
  - Set COM pins (not required for all LCD controllers, but necessary for the 'FR4133 since any LCD pin can be used as a COM line.
  - Finally, turn on the LCD_E controller
- **Runtime** – display the segements need for your application; changing them as necessary.

Initialization Worksheet

1. **From the MSP-EXP430FR4133 Launchpad User's Guide, what 'FR4133 LCD pins (Lx) need to be configured for use by the display. (Hint, look on page 15 of slau595.pdf.)**

   ____

   ____

2. **Complete the DriverLib function which sets these Lx pins as LCD pins.**

   (Hint: Look in the DriverLib User's Guide for the proper syntax.)

```
LCD_E_setPinAsLCDFunctionEx( LCD_E_BASE,

        ________________________________________  //starting pin

        ________________________________________  //ending pin
)
LCD_E_setPinAsLCDFunctionEx( LCD_E_BASE,

        ________________________________________  //starting pin

        ________________________________________  //ending pin
)
```

3. How fast will the frame rate (F_{frame}) be given this initialization code?

This code is used to initialize the LCD controller.

```
LCD_E_initParam initParams = LCD_E_INIT_PARAM;
  initParams.clockDivider  = LCD_E_CLOCKDIVIDER_3;
  initParams.muxRate       = LCD_E_4_MUX;
  initParams.segments      = LCD_E_SEGMENTS_ENABLED;
LCD_E_init(LCD_E_BASE, &initParams);
```

Here's a brief line-by-line description of the code:

a) Creates an inititialization variable (initParam) and sets it to a set of default values. (The default values are specified in the DriverLib User's Guide.)

   The remaining 3 lines of code alter these elements from their defaults. Other structure elements, such as *initParams.clockSource* is, left to its default = XT1CLK.

b) The clock divider alters the F_{lcd}, which in turn affects F_{frame}.

c) Static displays are the default, but the Launchpad use a 4-mux display.

d) By default all segments are left disabled. We want to leave them enabled.

e) The LCD_E_init() call applies the parameters to the LCD controller.

F_{LCD} = ______________________________________

F_{FRAME} = ______________________________________

Hints:

- The 'FR4133 User's Guide provides two formulas to help you calculate the frame rate.
  - As we discussed, the LCD frequency should be:

$$f_{LCD} = 2 \times mux \times f_{FRAME}$$

  - The LCD frequency can also be calculated with this expression:

$$f_{LCD} = \frac{f_{SOURCE}}{(LCDDIVx + 1) \times MUXDIVDER}$$

  - The code snippet in this lab step provides us the f_{source} and LCDDIVx values.
  - The trickiest part is figuring out the value of MUXDIVIDER. It isn't the "obvious" value, which would be "4". Rather, the value is specified in a table within the FR4133 User's Guide – look for it in the LCD_E section entitled *"LCD Timing Generation"*.

4. **Write two lines of code to clear all the LCD Memory.**

____

____

5. **Which bits are set by these 4 lines of code?**

```
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_0, LCD_E_MEMORY_COM0);
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_1, LCD_E_MEMORY_COM1);
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_2, LCD_E_MEMORY_COM2);
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_3, LCD_E_MEMORY_COM3);
```

These functions tell LCD_E which (Lx) pins to use for the common (COM) signals. Where is this information stored? (That is, what gets altered by this code?)

____

____

Runtime Worksheet

6. **Which address/bit controls each of the following segments? Fill out the table.**

Just to get you started, we added the Antenna symbol to the table.

Symbol	Memory Location	Bit Location	Hex Value
Antenna (ANT)	LCDM9	2	0x04
Heart (HRT)			
Timer (TMR)			

7. **Write the line of code that sets (i.e. turns on) the "Heart" segment.**

```
LCD_E________________________________( LCD_E_BASE,
              ____________________________,   // Location
              ____________________________    // Mask (hex)
);
```

8. **What happens if we set (turn on) the HRT symbol, then set the TMR symbol?**

   Will they both be enabled, or will the second one replace the first one?

   ____

   ____

9. **What function lets us clear one symbol without affecting another controlled by the same memory location (LDCMx)?**

   ____

   Complete the function to clear the Timer (TMR) symbol.

```
LCD_E____________________________( LCD_E_BASE,
           LCD_E_MEMORY_BLINKINGMEMORY_12,  // Location
           0x8 );                           // Mask (hex)
```

10. **What's the greatest advantage to the automatic blinking features of the 'FR4133?**

    ____

    ____

    ____

11. **Finish the following line of code so that it enables the LCD controller's blinking feature – switching between both banks of memory.**

```
LCD_E____________________________( LCD_E_BASE,
           LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_64,

           ______________________________________________ );
```

Lab File Management

12. Verify CCS is open and close any projects that are open in the workspace.

13. Import the `lab_12a_heart` project.

```
Project → Import CCS Projects…
C:\msp430_workshop\fr4133_fram\lab_12a_heart
Copy the project into your workspace
```

Note: For your reference, to created this project by copying/pasting lab_06a_timer and renaming it. We then deleted: main.c, timer.h, timer.c

Finally, we added the files: myLcd.h, myLcd.c, (and a new file called) main.c

Examine and Tweak LCD Files

We want to quickly introduce you to each of the three new files found in this project. Note, some will require a little bit of editing.

myLcd.h (No edits required)

Defines or declares a number of items that can be used in your programs. The three main categories are:

- Definitions for character positions – one for each character on the display (from Left→Right)
- Global variables that define values for numerical digits and the alphabet. With them, you can easily print a “3” or a “”B” to the LCD. They are defined in `mylcd.c`.
- Finally, the header includes prototypes for three functions defined in `mylcd.c`.

myLcd.h

```


//***** Defines *********************************************
/* LCD Symbol Bits */
#define LCD_HEART                    (0x0004)
#define LCD_TIMER                    (0x0008)

/* LCD memory locations for each character position on the d
#define pos1 4   /* Digit A1 - L4  */
#define pos2 6   /* Digit A2 - L6  */
#define pos3 8   /* Digit A3 - L8  */
#define pos4 10  /* Digit A4 - L10 */
#define pos5 2   /* Digit A5 - L2  */
#define pos6 18  /* Digit A6 - L18 */

/* Define word access definitions to LCD memories */
//#define LCDMEMW ((int*)LCDMEM)


//***** Global Variables ************************************
extern const char digit[10][2];           // Segment values
extern const char alphabetBig[26][2];     // Segment values


//***** Prototypes ******************************************
void LCD_init(void);
void LCD_showChar( char, int );
void LCD_displayNumber( unsigned long );

```

:

myLcd.c (Some edits required)

In a nutshell, here's the things you'll find in this file. *(By the way, thanks to the MSP applications team as we borrowed quite a bit of code from their out-of-box demo application.*

- It begins with the initialization of the 'digit' and 'alphabet' arrays. Once again, this makes it easy to use symbols without needing sprinkle hex values all thoughout your code.
- LCD_init() function
  - Turn off LCD_E
  - Set Lx I/O pins needed by controller
  - Setup the input clocking and frame rate (and enable the segment pins)
  - Configure voltage requirements – including enabling the built-in Charge Pump, if used.
  - Set COM pins (not required for all LCD controllers, but necessary for the 'FR4133 since any LCD pin can be used as a COM line.
  - Finally, turn on the LCD_E controller
- LCD_showChar() function
  - This function displays a character given a character/digit and position within the display
- LCD_displayNumber()
  - This function takes a numerical value (using the long data type) and displays it on the LCD.
  - If the value isn't a number, the function displays "ERROR".
  - Also, the value is displayed in a right-justified fashion.
  - We developed this function for use in the `lab_12b_persistence` exercise.

Now, on to the edits for this file… we've left a few items for you to fill-in, based upon the earlier worksheet questions.

14. Fill in the details for the two functions which assign Lx pins to the LCD Controller.

Refer back question #2 (on page 12-58).

15. Write in the two functions needed to clear the LCD memory.

Refer back question #4 (on page 12-60).

16. What happens if you set the COM pins and cleared the memory in reverse order?

If you're not quite sure, Question #5 (on page 12-60) should help. That is, thinking about where the COM bits are stored .

____

____

main.c (Some edits required)

Main Edits

Only edits here are to fill in the details for three missing functions.

17. Fill in the function that sets the heart to display.

Refer back question #7 (on page 12-60).

18. How do we clear some LCD memory location? Fix that line of the file.

Refer back question #9 (on page 12-61).

19. Finally, complete the function which turns on blinking by switching back-and-forth between memories.

Refer back question #11 (on page 12-61).

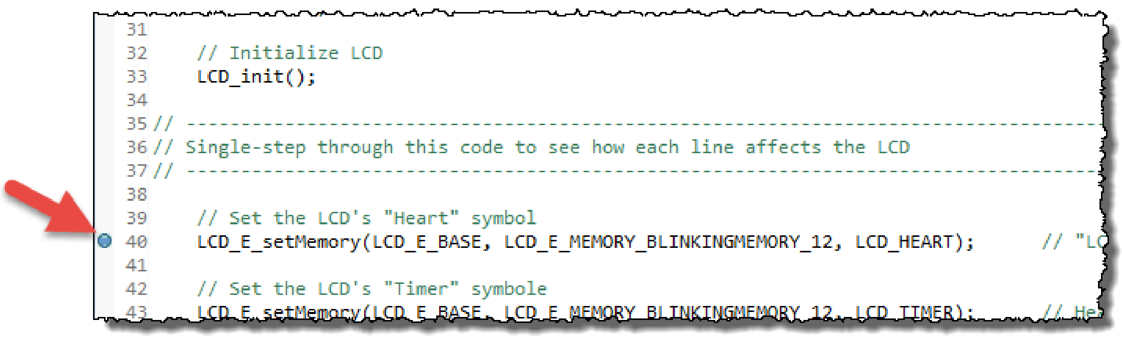
20. Build your code and fix any typos and errors.

Step and Observe

21. Launch the debugger to load your code into the 'FR4133.

22. Set a breakpoint on the first line of code where we begin manipulating the display.

Up until this point, the code is the same as it was in Lab6a – the only difference being that we've initialized the LCD rather than a Timer.



23. Stepping over the first LCD_E_setMemory() function, you should see the "Heart" appear on the LCD display.

Did the heart appear?____

24. The next stepover should display the Timer symbol.

Did the Timer appear? ____

Is the Heart still there? ____

Refer back to Question #8 (on page 12-61). Was your prediction correct?

25. The next three Step-overs demonstrate the 'update', 'clear' and 'toggle' memory functions.

Verify they worked as expected.

26. The next step takes us back to the 'setMemory' function.

```
// Set both the "Heart" and "Timer" symbols
LCD_E_setMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_12, LCD_HEART | LCD_TIMER);
```

What's different about this function this time? ________________________________

Made you Blink

27. The next three function calls explore many of the blinking features.

```
// Let's explore the Blinking features

    // Blinks enaboled segments on the display (in this case, the Heart and Timer)
    // Notice how the blinking continues, even when the processor is halted at a breakpoint (or during single-stepping)
    // This is because the LCD controller is automatically handling the blinking -- no timers or interrupts are required!
    LCD_E_setBlinkingControl(LCD_E_BASE, LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_64, LCD_E_BLINK_MODE_INDIVIDUAL_SEGMENTS);

    // Turn off the Timer symbol
    LCD_E_clearMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_12, LCD_TIMER);            // Erases just the Timer symbol

    // Turns off the blinking feature
    LCD_E_setBlinkingControl(LCD_E_BASE, LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_64, LCD_E_BLINK_MODE_DISABLED);
```

First we enable all the individual segments to flash. Does that mean every segment flashes? Or just the enabled segments?

____

28. The final set of "blinking" function calls:

a) First has us populating – and using – the Blinking memory (LCDBM). This shows us how to switch back and forth (manually) between displaying either memory.

b) Next, we can start to see how to use these two memories to make custom (more complicated) blinking patterns. With the "Heart" enabled in one memory… and the "Timer" in the other…

Did the two icons alternate flashing? ________________________________

Why would this solution be superior to using a timer ISR to tell you when to go switch what is being displayed?

____

D.O.N.E.

29. Well, actually the last step in our program just spells DONE (not D.O.N.E.).

30. When you're all done playing and tweaking the code, please go ahead and close the project.

(Optional) Lab 12b – Displaying Persistent Data

We thought it would be fun to take `lab_09a_persistent` and write the count value to the LCD display, rather than just to the CCS console.

Go ahead and run this lab. You can probably tell right away that it's a mashup of `lab_09a_persistent` and `lab_12a_heart`.

Explore the code, build it and test it out.

If you're looking for more of a challenge, you could remove the printf() and/or LED feedback options. Alternatively, you could program the buttons (using the code from Lab 5) to reset the count value or increment it further.

Notes:

Appendix

Initialization Worksheet

1. **From the MSP-EXP430FR4133 Launchpad User's Guide, what 'FR4133 LCD pins (Lx) need to be configured for use by the display. (Hint, look on page 15 of slau595.pdf.)**

   pins L0~L26 as well as

   pins L36~L39

2. **Complete the DriverLib function which sets these Lx pins as LCD pins.**

   (Hint: Look in the DriverLib User's Guide for the proper syntax.)

```
LCD_E_setPinAsLCDFunctionEx( LCD_E_BASE,
        LCD_E_SEGMENT_LINE_0          //starting pin
        LCD_E_SEGMENT_LINE_26         //ending pin
)
LCD_E_setPinAsLCDFunctionEx( LCD_E_BASE,
        LCD_E_SEGMENT_LINE_36         //starting pin
        LCD_E_SEGMENT_LINE_39         //ending pin
)
```

3. **How fast will the frame rate (F_{frame}) be given this initialization code?**

   This code is used to initialize the LCD controller.

```
LCD_E_initParam initParams = LCD_E_INIT_PARAM;
  initParams.clockDivider   = LCD_E_CLOCKDIVIDER_3;
  initParams.muxRate        = LCD_E_4_MUX;
  initParams.segments       = LCD_E_SEGMENTS_ENABLED;
LCD_E_init(LCD_E_BASE, &initParams);
```

F_{LCD} = **32768 / (3 + 1) * 16 = 512**

F_{FRAME} = **512 / 8 = 64 Hz**

Hints:

- The 'FR4133 User's Guide provides two formulas to help you calculate the frame rate.
  - As we discussed, the LCD frequency should be:

$$f_{LCD} = 2 \times mux \times f_{FRAME}$$

  - The LCD frequency can also be calculated with this expression:

$$f_{LCD} = \frac{f_{SOURCE}}{(LCDDIVx + 1) \times MUXDIVDER}$$

4. **Write two lines of code to clear all the LCD Memory.**

```
LCD_E_clearAllMemory(LCD_E_BASE);
LCD_E_clearAllBlinkingMemory(LCD_E_BASE);
```

5. **Which bits are set by these 4 lines of code?**

```
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_0, LCD_E_MEMORY_COM0);
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_1, LCD_E_MEMORY_COM1);
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_2, LCD_E_MEMORY_COM2);
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_3, LCD_E_MEMORY_COM3);
```

These functions tell LCD_E which (Lx) pins to use for the common (COM) signals. Where is this information stored? (That is, what gets altered by this code?)

The COM pin assignments is stored in the LCD memory

Warning – if you clear the LCD, you erase these assignments

Runtime Worksheet

6. **Which address/bit controls each of the following segments? Fill out the table.**

Just to get you started, we added the Antenna symbol to the table.

Symbol	Memory Location	Bit Location	Hex Value
Antenna (ANT)	LCDM9	2	0x04
Heart (HRT)	LCDM12	2	0x04
Timer (TMR)	LCDM12	3	0x08

7. **Write the line of code that sets (i.e. turns on) the "Heart" segment.**

```
LCD_E_setMemory( LCD_E_BASE,
    LCD_E_MEMORY_BLINKINGMEMORY_12,   // Location
    0x08                              // Mask (hex)
);
```

8. **What happens if we set (turn on) the HRT symbol, then set the TMR symbol?**

   Will they both be enabled, or will the second one replace the first one?

   Only one will be "on", as the "setMemory" function overwrites the memory location

9. **What function lets us clear one symbol without affecting another controlled by the same memory location (LDCMx)?**

   LCD_E_clearMemory

   Complete the function to clear the Timer (TMR) symbol.

```
LCD_E_clearMemory( LCD_E_BASE,
          LCD_E_MEMORY_BLINKINGMEMORY_12,   // Location
          0x8 );                            // Mask (hex)
```

10. **What's the greatest advantage to the automatic blinking features of the 'FR4133?**

    While it's easy to use, the greatest advantage is Ultra Low-Power. You get these advantages, even when the device is sleeping in LPM3.5 mode!

11. **Finish the following line of code so that it enables the LCD controller's blinking feature – switching between both banks of memory.**

```
LCD_E_setBlinkingControl( LCD_E_BASE,
          LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_64,
     LCD_E_BLINK_MODE_SWITCHING_BETWEEN_DISPLAY_CONTENTS );
```

16. **What happens if you set the COM pins and cleared the memory in reverse order?**

    If you're not quite sure, Question #5 (on page 12-28) should help. That is, thinking about where the COM bits are stored .

    Warning – if you clear the LCD memory, this will erase the COM pin assignments

23. Stepping over the first LCD_E_setMemory() function, you should see the "Heart" appear on the LCD display.

Did the heart appear? Yes

24. The next stepover should display the Timer symbol.

Did the Timer appear? Yes

Is the Heart still there? No

What's different about this function this time? Sets both segments at the same time

Made you Blink

27. The next three function calls explore many of the blinking features.

```
// Let's explore the Blinking features

    // Blinks enaboled segments on the display (in this case, the Heart and Timer)
    // Notice how the blinking continues, even when the processor is halted at a breakpoint (or during single-stepping)
    // This is because the LCD controller is automatically handling the blinking -- no timers or interrupts are required!
    LCD_E_setBlinkingControl(LCD_E_BASE, LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_64, LCD_E_BLINK_MODE_INDIVIDUAL_SEGMENTS);

    // Turn off the Timer symbol
    LCD_E_clearMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_12, LCD_TIMER);            // Erases just the Timer symbol

    // Turns off the blinking feature
    LCD_E_setBlinkingControl(LCD_E_BASE, LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_64, LCD_E_BLINK_MODE_DISABLED);
```

First we enable all the individual segments to flash. Does that mean every segment flashes? Or just the enabled segments?

Just the enabled ones

28. The final set of "blinking" function calls:

a) First has us populating – and using – the Blinking memory (LCDBM). This shows us how to switch back and forth (manually) between displaying either memory.

b) Next, we can start to see how to use these two memories to make custom (more complicated) blinking patterns. With the "Heart" enabled in one memory… and the "Timer" in the other…

Did the two icons alternate flashing? Yes

Why would this solution be superior to using a timer ISR to tell you when to go switch what is being displayed?

Lower CPU overhead; more precise timing; MUCH, MUCH lower power