

MSP430 Clocks & Initialization

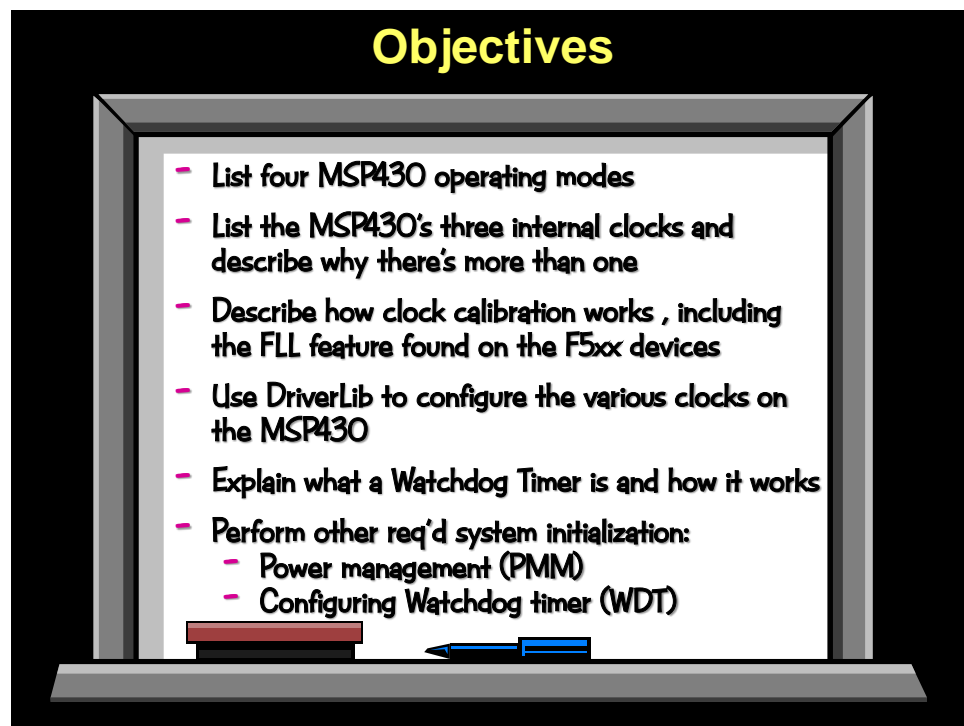
Introduction

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

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

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

Learning Objectives



Chapter Topics

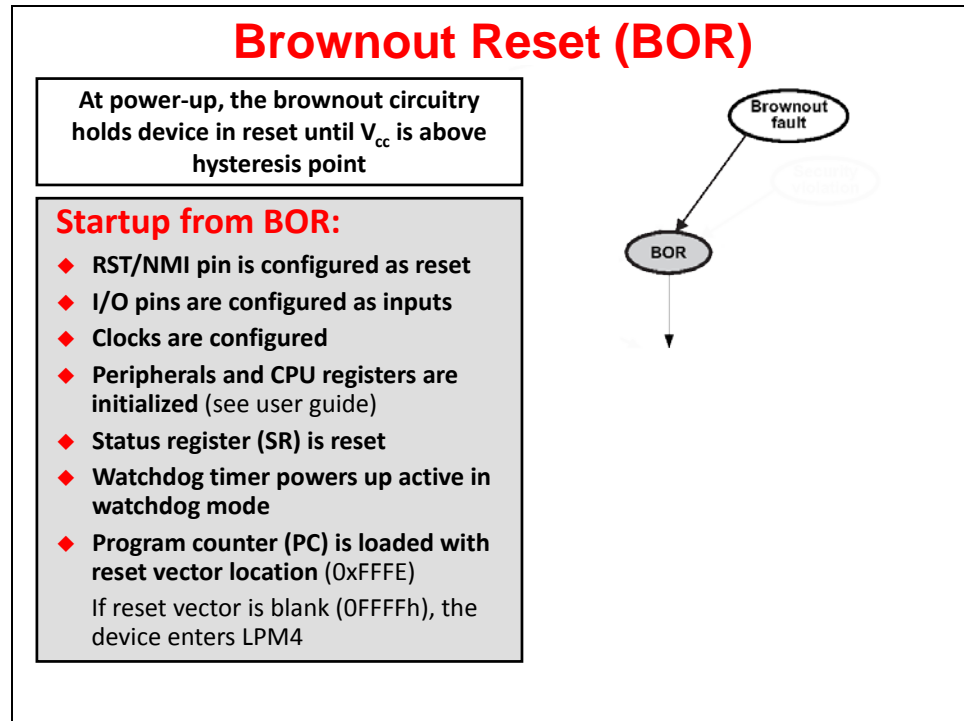
MSP430 Clocks & Initialization	4-1
<i>Operating Modes (Reset → Active)</i>	<i>4-3</i>
BOR.....	4-3
BOR → POR → PUC → Active (AM)	4-4
<i>Clocking.....</i>	<i>4-6</i>
What Clocks Do You Need?	4-6
MCLK, SMCLK, ACLK	4-8
Oscillators (Clock Sources).....	4-9
Clock Details (by Device Family)	4-11
Using MSP430ware to Configure Clocking	4-16
Additional Clock Features	4-18
<i>DCO Setup and Calibration</i>	<i>4-21</i>
How the DCO Works.....	4-22
Factory Calibration (FR5xx, G2xx).....	4-26
Runtime Calibration (F4xx, F5xx, F6xx).....	4-28
FR2xx/4xx DCO Calibration	4-31
VLO 'Calibration'	4-32
<i>Other Initialization (WDT, PMM)</i>	<i>4-33</i>
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
<i>Lab Exercise</i>	<i>4-41</i>

Operating Modes (Reset → Active)

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

BOR

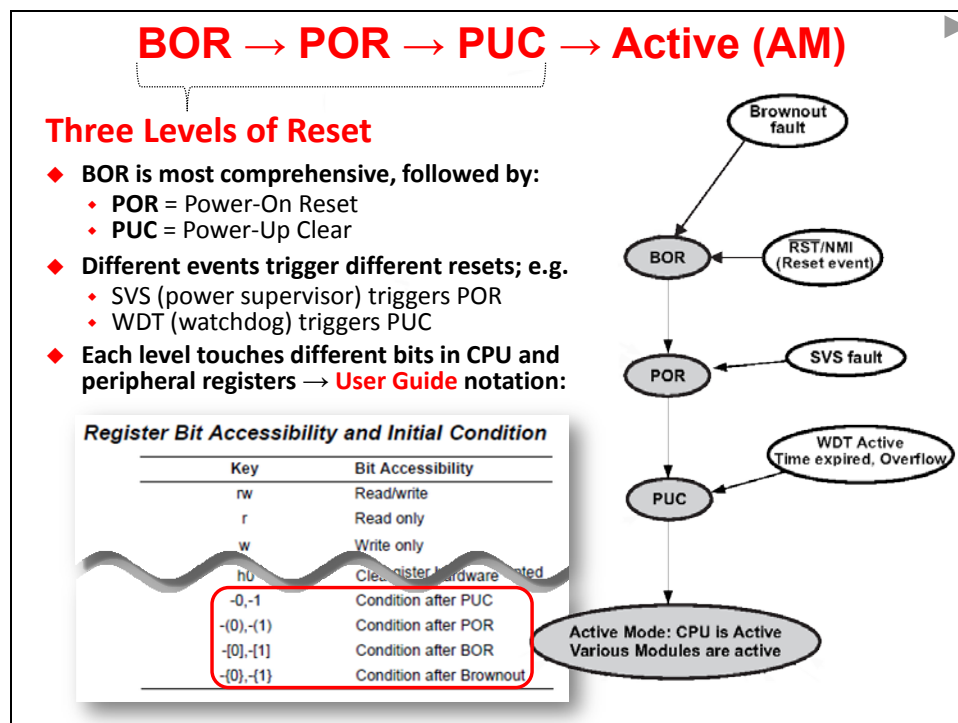
The MSP430 starts out in the Brown-Out Reset (BOR) mode. A Brownout Fault (i.e. not enough power) is the most common event that brings the CPU to this state.



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

BOR → POR → PUC → Active (AM)

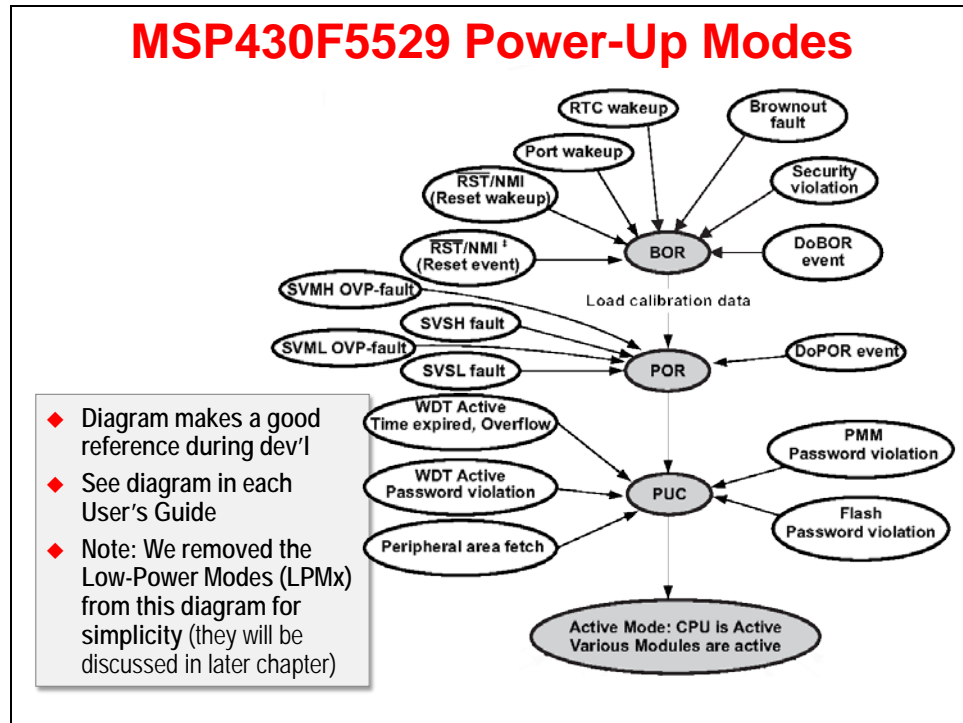
As shown below, BOR is the first of three reset states.



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

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

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



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

Clocking

What Clocks Do You Need?

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

What Clocks Do You Need?

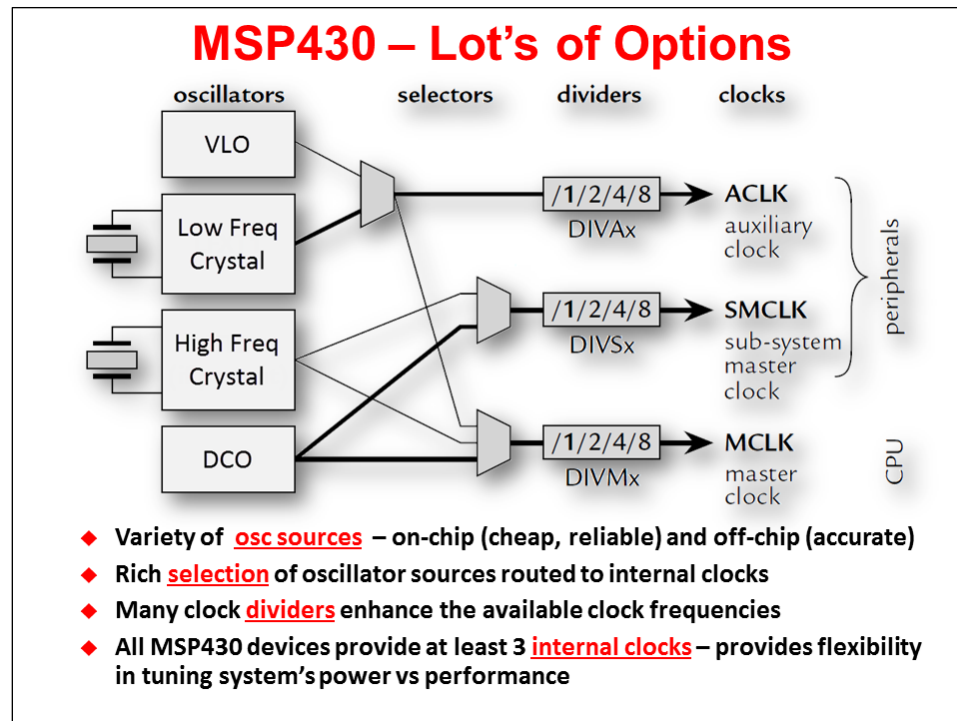
- ◆ **Fast Clocks** CPU, Communications, Burst Processing
- ◆ **Low-power** RTC, Remote, Battery, Energy Harvesting
- ◆ **Accurate** Stable over %V, Communications, RTC, Sensors
- ◆ **Failsafe** Robust—keeps system running in case of failure
- ◆ **Cheap** ... goes without saying ...

... or some combination of these features?

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

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

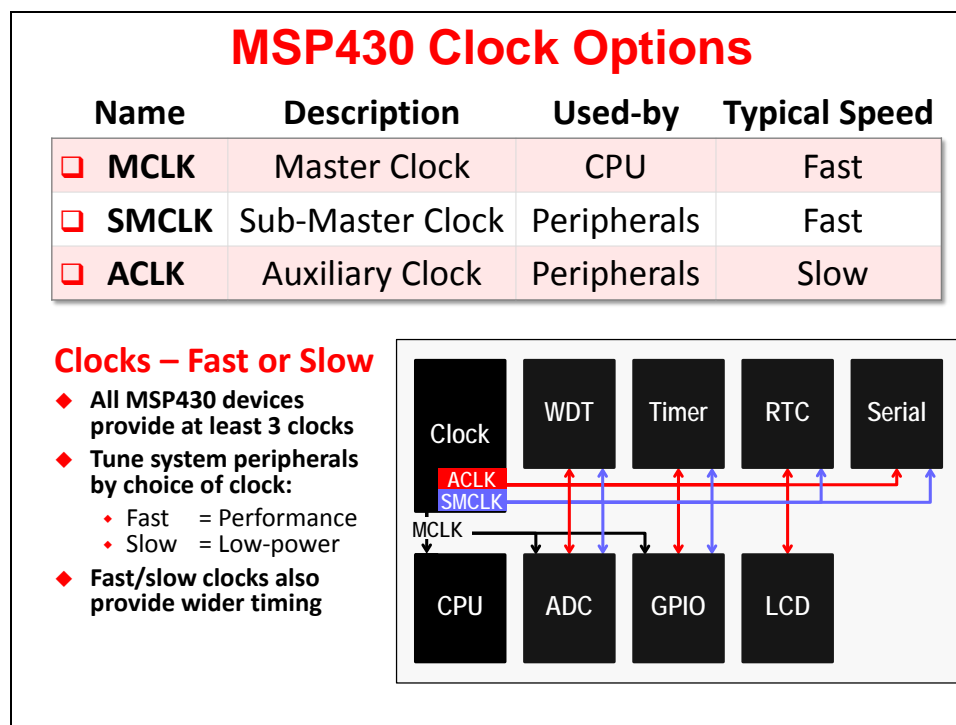
On the left-hand side, there are internal and external oscillators which provide both high and slow speed clock sources.



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

MCLK, SMCLK, ACLK

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



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

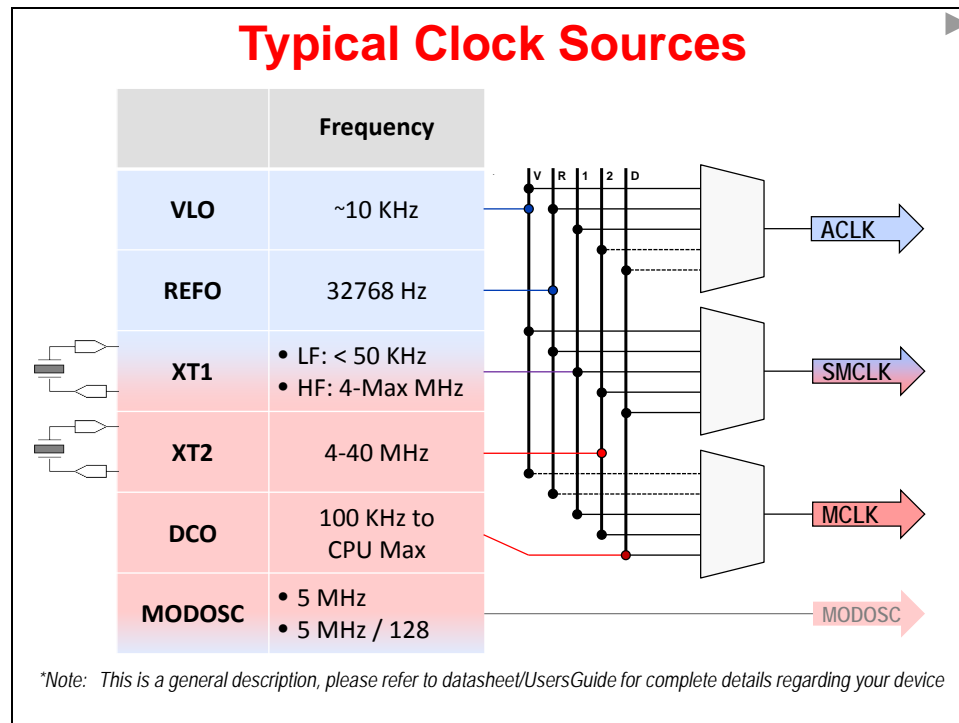
Some peripherals (such as serial ports) often require a fast clock to meet the communication data rate requirements while other peripherals (e.g. timers) may not always need to run as fast. The ability to provide a low-speed clock can provide two advantages:

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

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

Oscillators (Clock Sources)

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



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

Typical Clock Sources

	Frequency	'G2553 Value-line	'F5529 USB	'FR4133 FRAM	'FR5969 FRAM
VLO	~10 KHz	☑	☑	☑	☑
REFO	32768 Hz		☑	☑	
XT1	<ul style="list-style-type: none"> LF: < 50 KHz HF: 4-Max MHz 	☑	☑	☑	☑
XT2	4-40 MHz		☑		☑
DCO	100 KHz to CPU Max	☑	☑	☑	☑
MODOSC (MODCLK)	<ul style="list-style-type: none"> 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 ($\pm 40\%$)	60nA	Use as Ultra Low Power tick
REFO	32768 Hz	Med/High (3.5%)	3 μ A 25 μ S	Trimmed to 3.5%
XT1	<ul style="list-style-type: none"> LF: < 50 KHz HF: 4-Max MHz 	High	75nA 500-1k mS	Crystal or Ext Clock
XT2	4-40 MHz	High	260 μ A (12MHz) 400 μ S	Crystal or Ext Clock
DCO	100 KHz to CPU Max	Low/Med	60 μ A 200nS	Calibrate with Constant/FLL
MODOSC	<ul style="list-style-type: none"> 5 MHz 5 MHz / 128 	Med	N/A	Used by FLASH or ADC

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

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

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

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

DCO: the Digitally Controlled Oscillator (DCO) provides a fast, low-loss, 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 MODule 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.

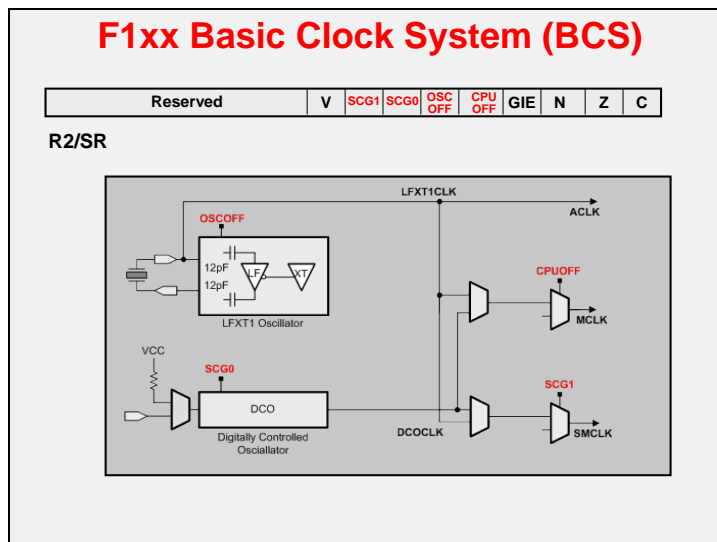
MSP430 Clock Modules		
Module	Clock Module Name	MSP430 Device Family
BCS	Basic Clock System	F1xx / F2xx
BCS+	Basic Clock System +	F2xx / G2xx
FLL+	Frequency Locked Loop +	F4xx
UCS	Unified Clock System	F5xx / F6xx
CS	Clock System	FR5xx / FR6xx
CS	Clock System (slightly different than FR5xx/6xx version)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

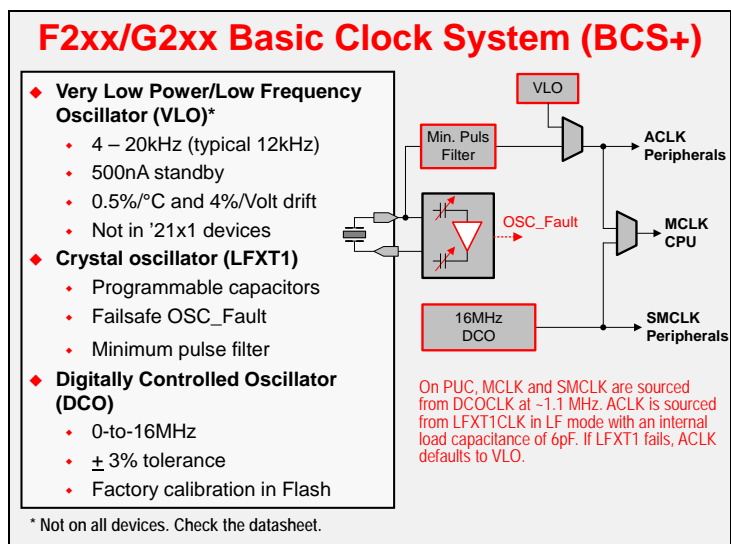
F1xx Clocking (BCS)

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.)



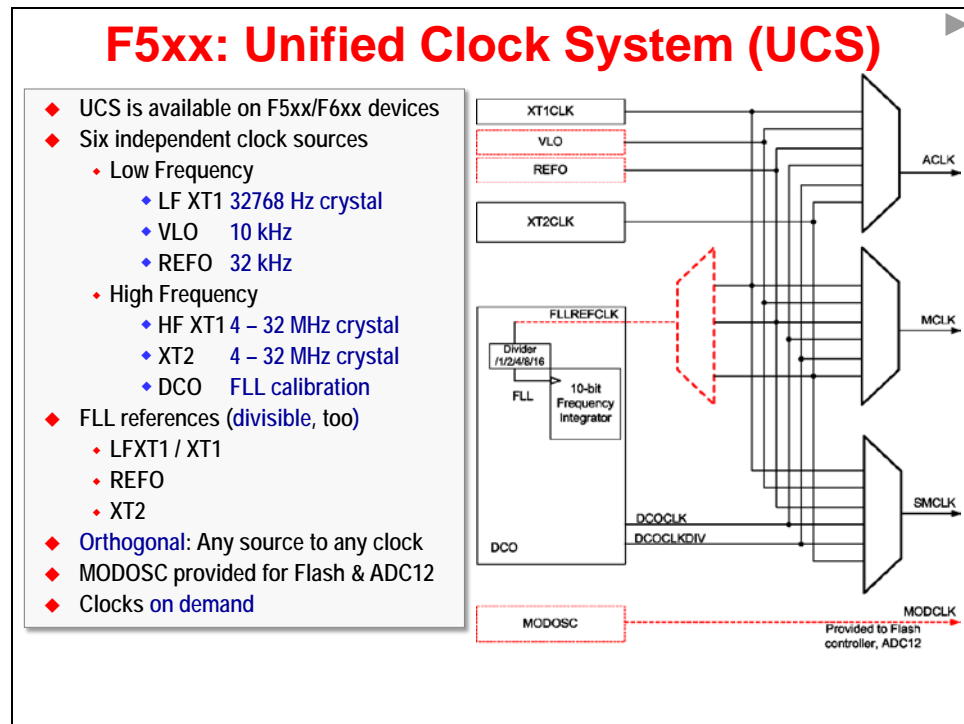
F2xx/G2xx Clocks (BCS+)

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/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:

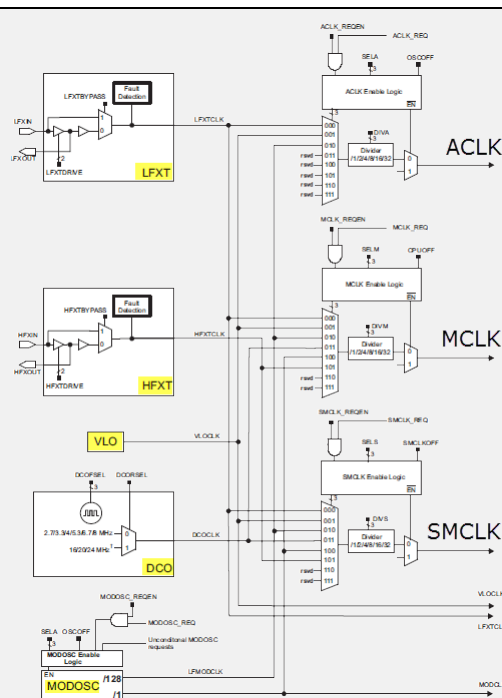
- ◆ Any OSC can drive any system clock (MCLK, ACLK, SMCLK)
- ◆ Clock divider up to 32 for each system clock
- ◆ Control the CLK in Low Power Modes (stopped or running) and react to module CLK requests
- ◆ OSC enable logic according requests
- ◆ Supporting the FLL as sub-module and providing the control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

FR58xx/FR59xx - Clock System (CS)

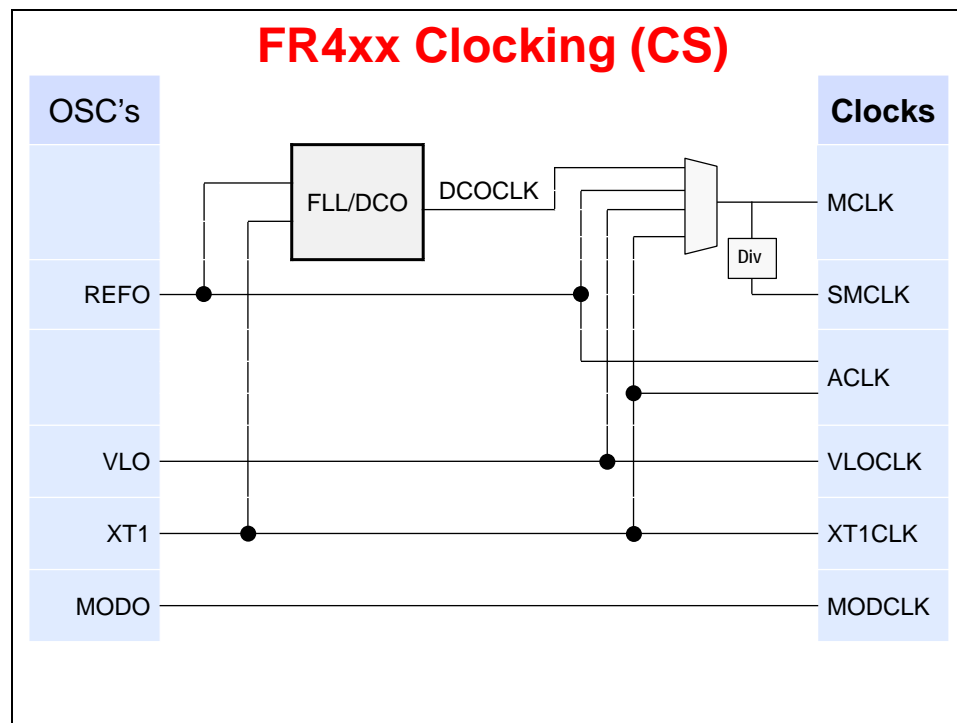
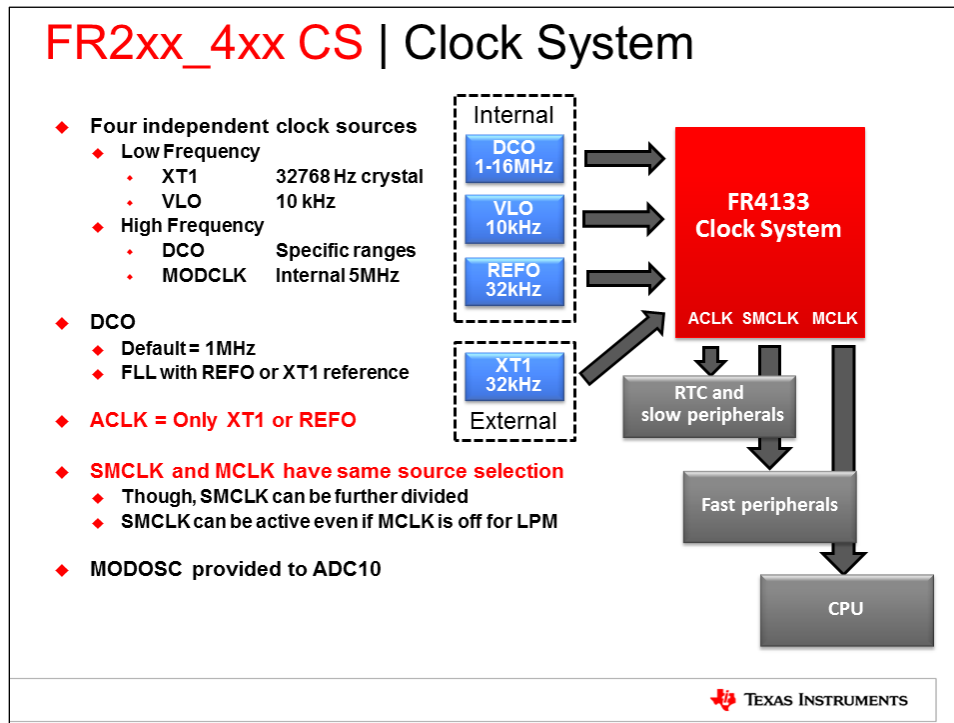
The Clock System (CS) used in the new 'FR5xx devices provides almost as much flexibility as the UCS peripheral, although – as we'll see later – it's easier to configure.

Clock System (CS)

- ◆ CS found on Wolverine (FR58/59xx)
- ◆ Five independent clock sources
 - ◆ Low Freq
 - ◆ LFXT (32768 Hz crystal)
 - ◆ VLO (10 kHz)
 - ◆ LFMODCLK (MODCLK/128)
 - ◆ High Freq
 - ◆ HFXT (4 – 24 MHz crystal)
 - ◆ DCO (Specific CAL range)
 - ◆ MODCLK (Internal 5MHz)
- ◆ Notes:
 - ◆ MODOSC provided to ADC12, MODCLK and LFMODCLK
- ◆ Defaults:
 - ◆ DCO = 1MHz
 - ◆ ACLK = Only LF sources
- ◆ Failsafe's:
 - ◆ LFXT: LFMODCLK (~42kHz)
 - ◆ HFXT: MODCLK (5MHz)



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.)

DriverLib – Using External Crystal

```
#include <driverlib.h>

//Set XIN (P5.4) and XOUT (P5.5) in Clock mode
GPIO_setAsPeripheralModuleFunctionInputPin(
    GPIO_PORT_P5, GPIO_PIN4);
GPIO_setAsPeripheralModuleFunctionOutputPin(
    GPIO_PORT_P5, GPIO_PIN5 );

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

UCS_clockSignalInit ( UCS_BASE,
    UCS_ACLK,           // Configure ACLK
    UCS_XT1CLK_SELECT,  // Set to REFO source
    UCS_CLOCK_DIVIDER_1 ); // Set clock divider to 1
```

- ◆ **Warning:** Verify XIN and XOUT before starting external oscillators! On many devices, these pins are shared with GPIO
- ◆ `UCS_LFXT1StartWithTimeout()` lets the function exit even if the crystal isn't working – make sure you check its return value

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

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

- For Low Frequency (LF) crystals, the drive strength option allows you to tune 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	VLO, LFXT1, XT2, DCO	VLO, REFO, XT1, XT2, DCOCLK, DCOCLKDIV	VLO, LFXT, LFMODCLK, HFXT, MODCLK, DCOCLK
	SMCLK			
	ACLK	VLO, LFXT1		VLO, LFXT, LFMODCLK
Clock Defaults (at PUC Reset)	MCLK	DCO (1.1MHz)	DCOCLKDIV (1.048 MHz)	DCO (1MHz)
	SMCLK			
	ACLK	LFXT1	XT1CLK (32KHz)	LFXT
External Clk Failsafe		ACLK = VLO S/MCLK = DCO	LF XT1 = REFO HF XT1/XT2 = DCO	LFXT= LFMODCLK (38KHz) HFXT=MODCLK (4.8MHz)
DCO Calibration		Factory Constant	FLL (Run-time)	Factory Trimmed
Password Needed (To change clock settings)		No	No	Yes
Clock Request (Periph can force clk on)		WDT+ only	Yes	Yes

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

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

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

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

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

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

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

For example:

Let's say that you wanted to put the CPU to sleep using Low-Power Mode 3 (LPM3) and wait in that mode until the UART received a byte and created an interrupt.

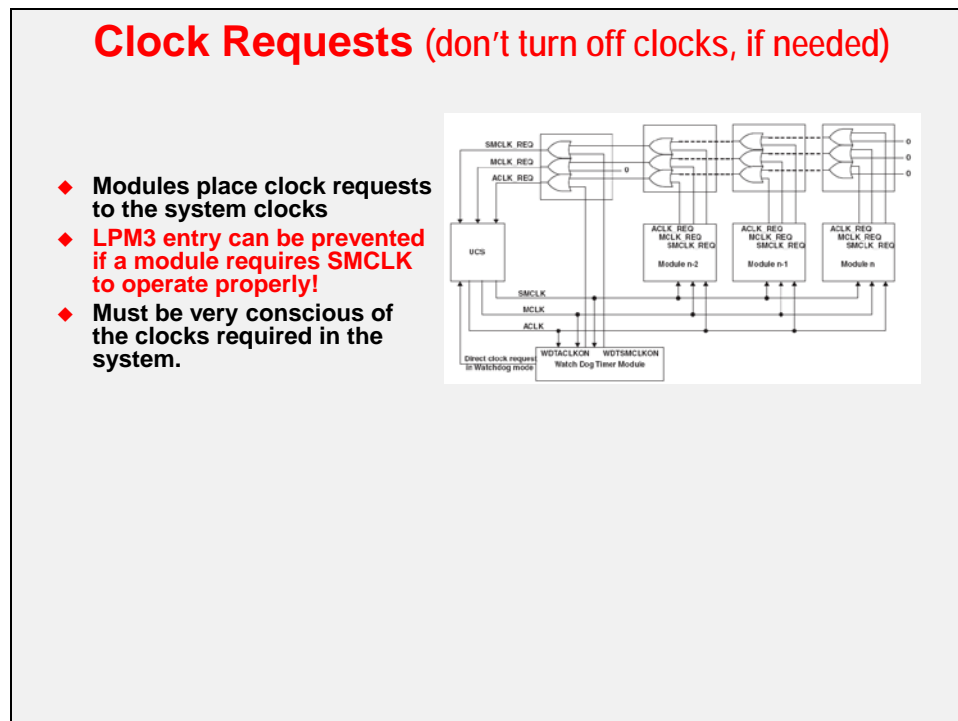
A problem would occur, though, if your UART was being clocked by SMCLK since LPM3 turns off SMCLK. In other words, what happens if the peripheral you were using to wake the processor up just happened to be using that clock, you would never wake up.

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

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

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

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



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

Additional Clock Notes/Warnings

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

Other Clock Notes/Warnings

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

DCO Setup and Calibration

Calibrating DCO

Additional Clock Features

Clock Feature		G2553 (BCS+)	F5529 (UCS)	F5969 (CS)
Available Clocks	MCLK	VLO, LFXT1, VLO	VLO, LFXT1, VLO	FXT, LFMODCLK, MODCLK, DCOCLK
Clock Defaults (at PUC reset)	MCLK	DCO (1.1MHz)	DCOCLKDIV (1MHz)	DCO (1MHz)
	ACLK	LFXT1	XT1CLK	LFXT
External Clk Failsafe		ACLK = VLO S/MCLK = DCO	LFXT1 = REFO HFXT1/XT2 = DCO	LFXT = LFMODCLK (42kHz) HFXT = MODCLK (5MHz)
DCO Calibration		Factory Constant	FLL (Run-time)	Factory Trimmed
Password Needed (To change clock settings)		No	No	Yes
Clock Request (Periph can force clk on)		No	Yes	Yes

EARLIER, WE DESCRIBED HOW THE DCO IS CALIBRATED

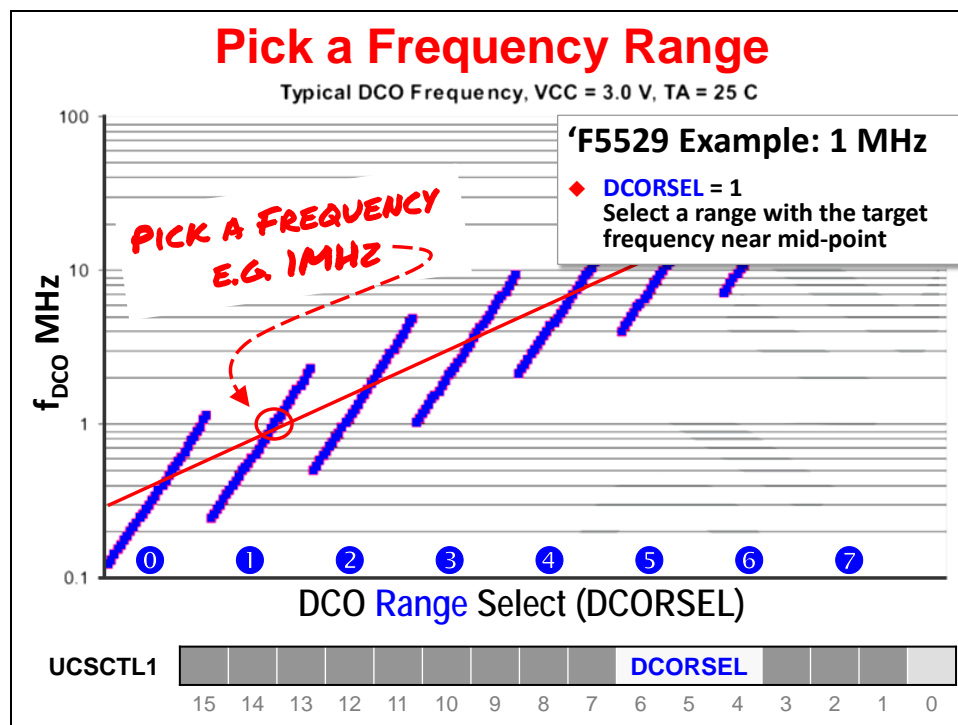
Before we look at the details of calibration, let's start with "How does the DCO work?"

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

How the DCO Works

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

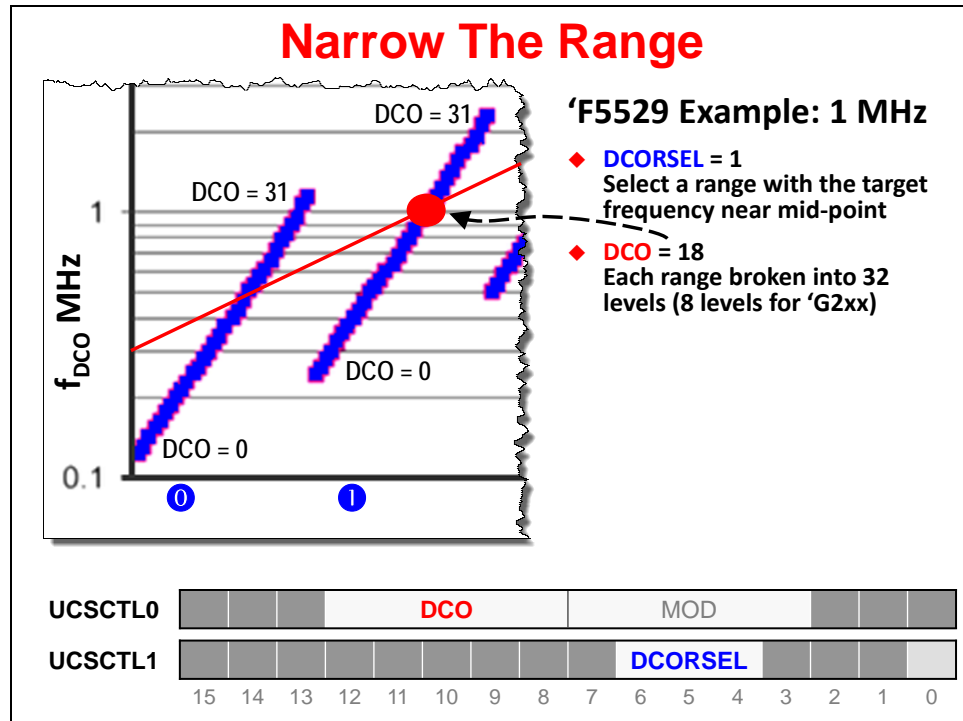
The DCO can operate in a number of different frequency ranges. On the 'F5529, you can select from one of eight different frequency ranges. You might notice that these ranges overlap each other quite a bit. The goal would be to pick a range where your desired frequency sits near the middle. (This is not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



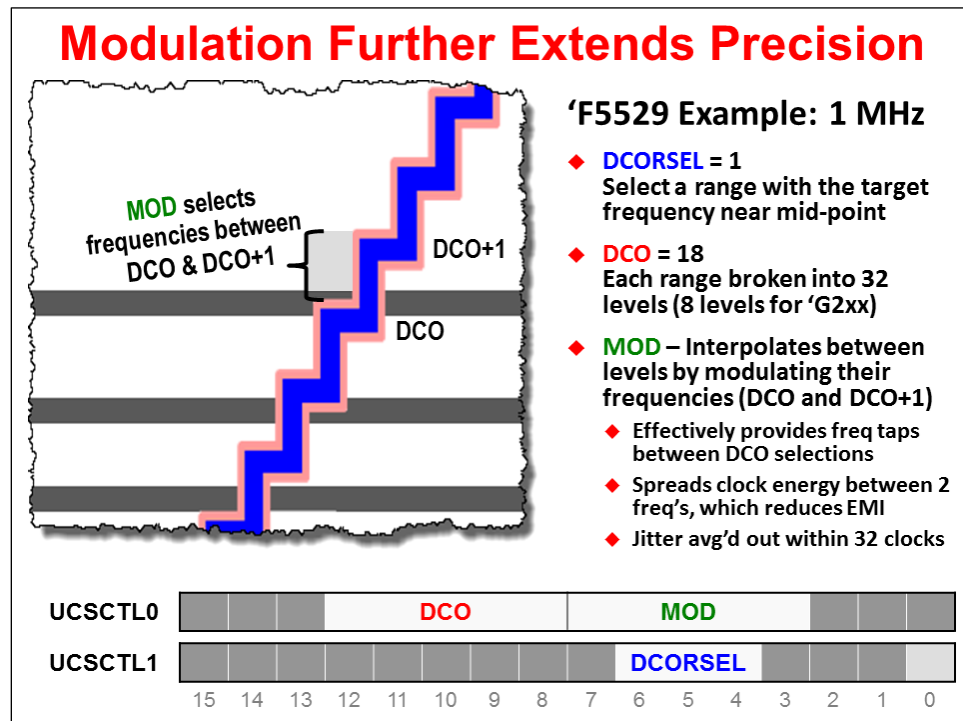
In the diagram above, if we wanted to run at 1 MHz, range "one" happens to be a good choice. Any of the first three would work, but range "1" puts our desired frequency close to the middle of the range.

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

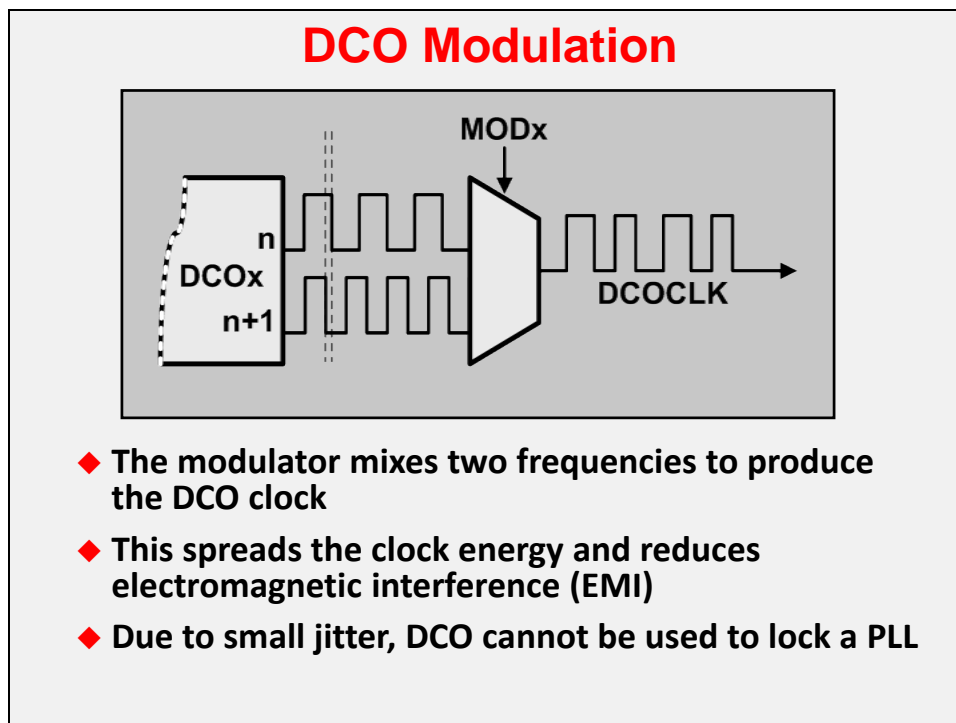
While the DCORSEL allows you to select a range of frequencies, it's the DCO field that allows us to indicate which frequency we desire within that range. On the 'F5529 the DCO field is 5-bits long, which means we're provided 32 different frequency levels in our chosen range.



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



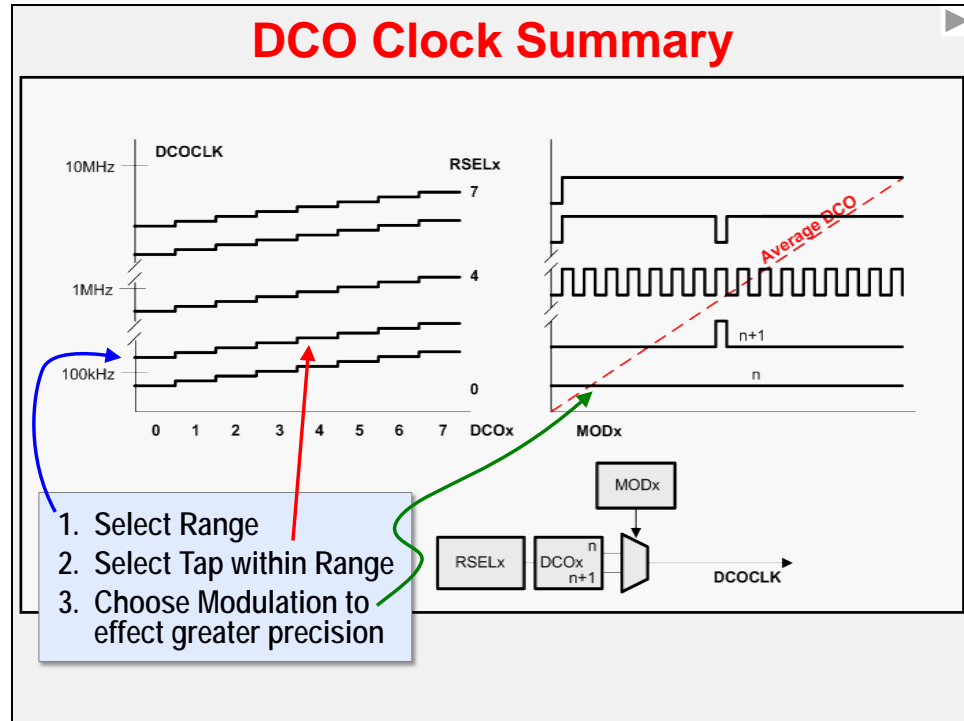
This is where the final field, called MOD, comes into play. MOD lets you tell the MSP430 clock to modulate between two frequency levels: DCO and DCO+1. By mixing these two frequencies you can obtain a very close approximation to your chosen clock frequency.



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

DCO Summary

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



Factory Calibration (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

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*

◆ Clock System (CS) module found on FR5xx devices

◆ DCO (CS module) provides multiple pre-defined & calibrated frequencies

◆ Factory Trimmed Accuracy:
 $\pm 2\%$ from 0-50C
 $\pm 3.5\%$ 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

CSCTL1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
										DCORSEL						

```
// Set DCO to 8MHz
CS_setDCOFreq( CS_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

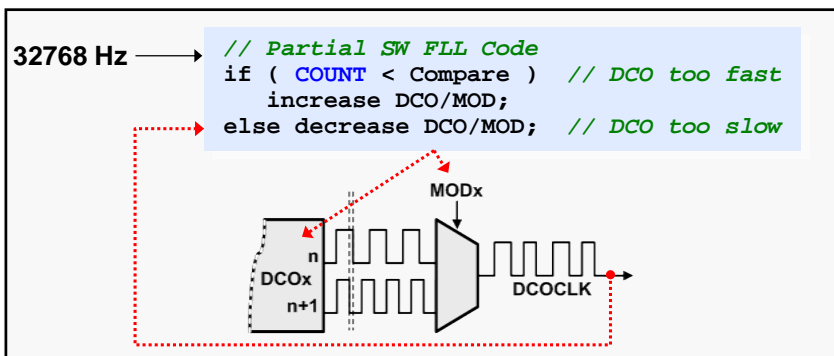
```
// Setting the DCO to 1MHz
if (CALBC1_1MHZ == 0xFF || CALDCO_1MHZ == 0xFF)
    while(1); // Erased calibration data? Trap!
BCSCTL1 = CALBC1_1MHZ; // Set range
DCOCTL = CALDCO_1MHZ; // Set DCO step + modulation
```

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).

Dynamic Calibration of DCO in Software

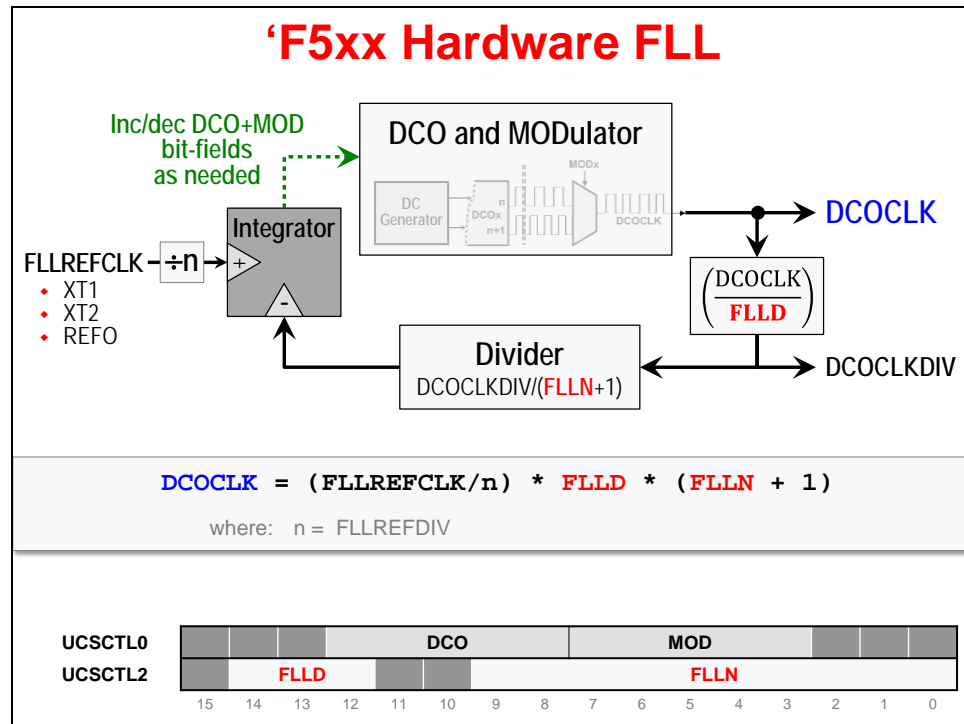


- ◆ **Minimize frequency drift** due to changes in voltage or temperature
 - DCO clock precision is achieved by periodic adjustment in loop
 - Modify settings (DCO, MOD) in loop based upon comparison of DCO to another known/stable freq, such as 32kHz crystal (or 50/60Hz AC power)
- ◆ **Frequency Locked Loop (FLL) – ‘lock’ one frequency to another**
 - Software FLL is the only option available on 'F1xx devices
 - While software FLL could be used for any MSP430 device, the F4xx/5xx/6xx clock modules contain Hardware FLL circuitry

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

At the top center of the following diagram, you'll see the DCO circuitry. The output of the DCO is labeled DCOCLK. 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.

DCOCLKDIV 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.



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.

$$\text{DCOCLK} = (\text{FLLREFCLK}/n) * \text{FLLD} * (\text{FLLN} + 1)$$

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.)

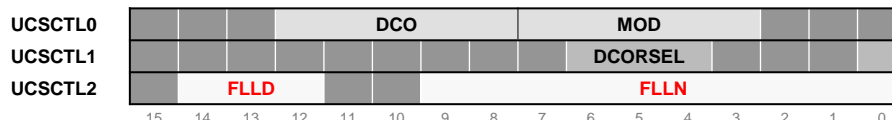
Setting 'F5529 DCO with MSP430ware

```
#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 = REFO
    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);
}
```



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.

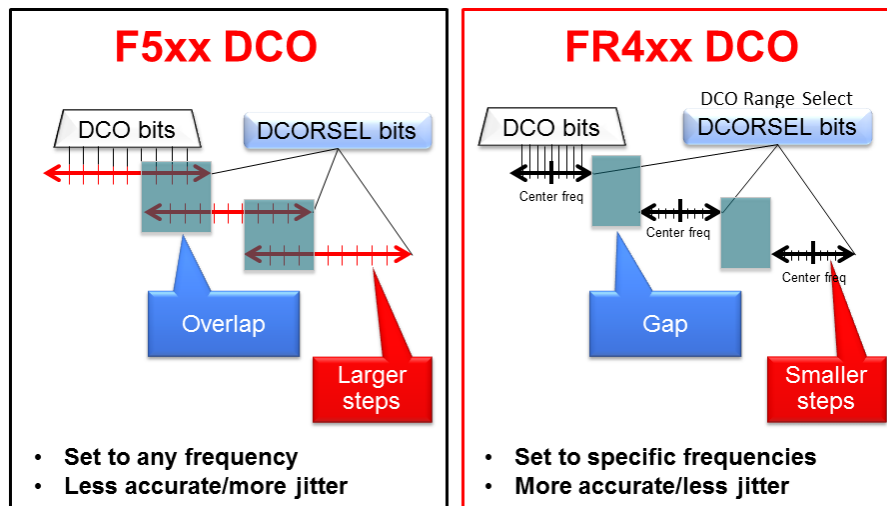
One last note, the initFLL functions will set MCLK and SMCLK to DCOCLK if the frequency is greater than 16MHz, otherwise it will use the divided down DCOCLKDIV.

FR2xx/4xx DCO Calibration

FR2xx_4xx Clock System (CS)

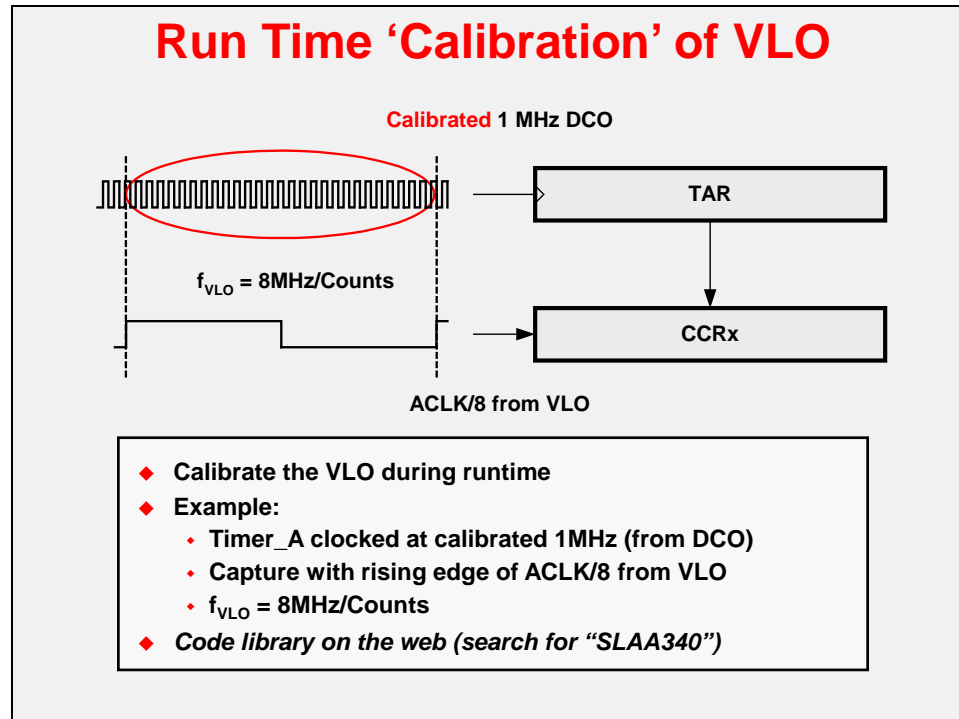
- ◆ DCO setup is hybrid of **FR5xx DCO** and **F5xx DCO + FLL**
- ◆ Specific frequency ranges
 - Ranges centered on 1, 2, 4, 8, 12, 16MHz
 - Selected with DCORSEL bits
- ◆ Uses FLL with reference frequency to tune within frequency range
- ◆ 512 DCO steps within these smaller ranges = smaller steps
 - 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 <u>watchdog timer</u> (usually OFF when debugging)	Yes	User	Chapter 4
3.	Setup Power Manager & Supervisors	No	User	Chapter 4
4.	Configure GPIO pins	No	User	Chapter 3
5.	Reconfigure clocks (if desired)	No	User	Chapter 4 (earlier)
6.	Configure peripheral modules	No	User	Later chapters

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

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

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

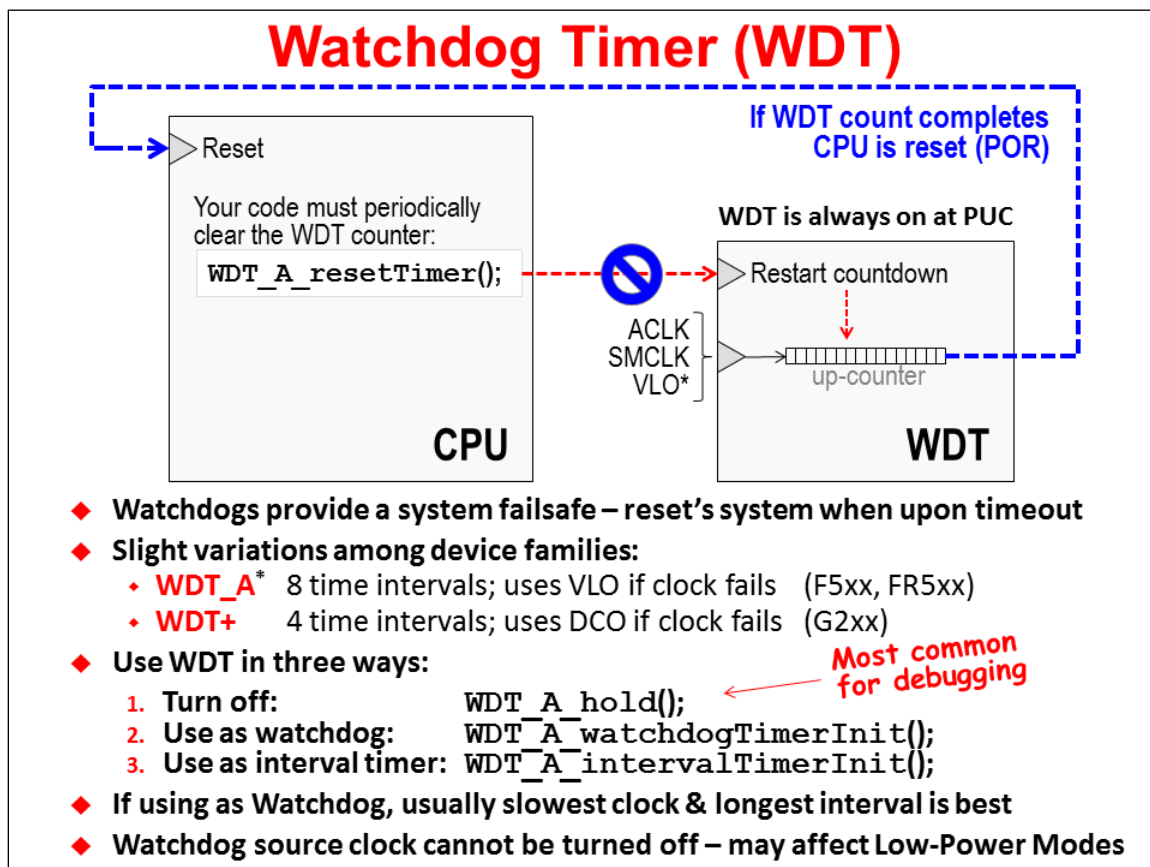
In the last chapter we discussed and used **GPIO pins** (general purpose bit I/O). It is highly recommended that you configure all GPIO pins on your device. Obviously, those being used need to be configured, but you should also configure those pins not in use so as to minimize power dissipation.

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.

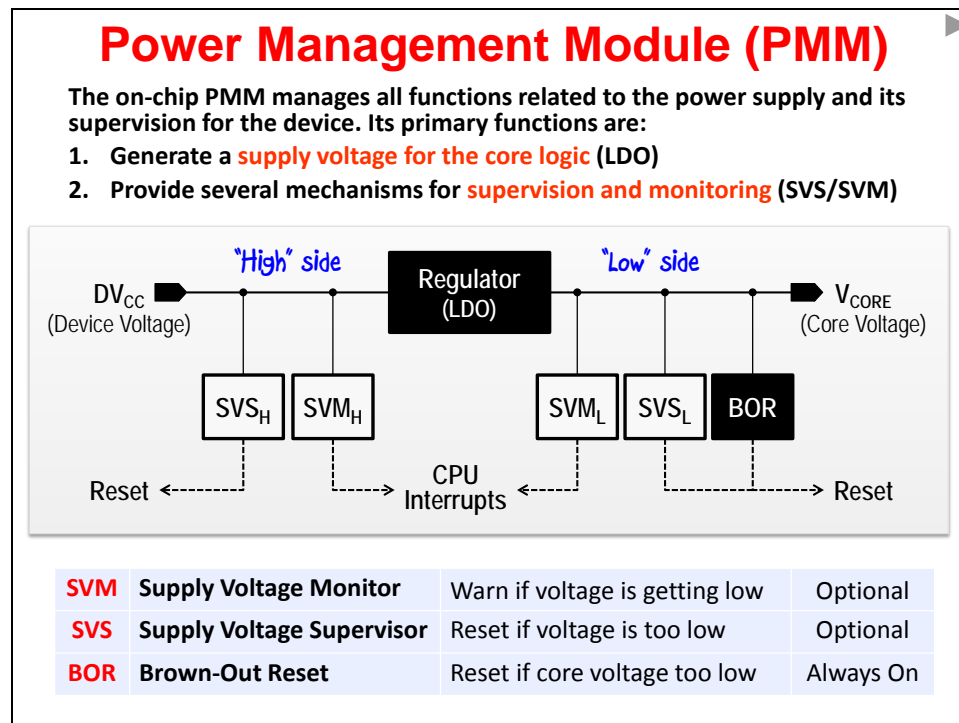
Note: As discussed earlier in this chapter, the clock being actively used by the Watchdog timer cannot be disabled. Keep this in mind as you plan out your system and calculate its power requirements.

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

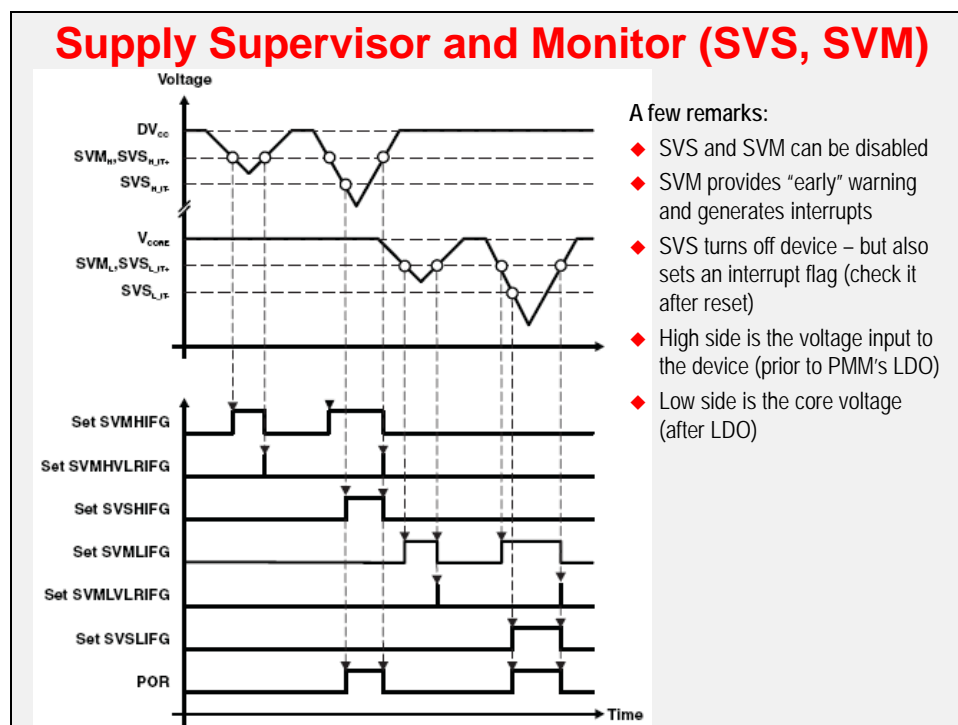


The **SVM** (supply voltage monitor) circuitry is intended to warn you (via interrupts) when the high- or low-side voltages are getting close to their lower limits. You might use this to correct the power supply or prepare for a power error/shutdown. (You can choose not to use this feature if you want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

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

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

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

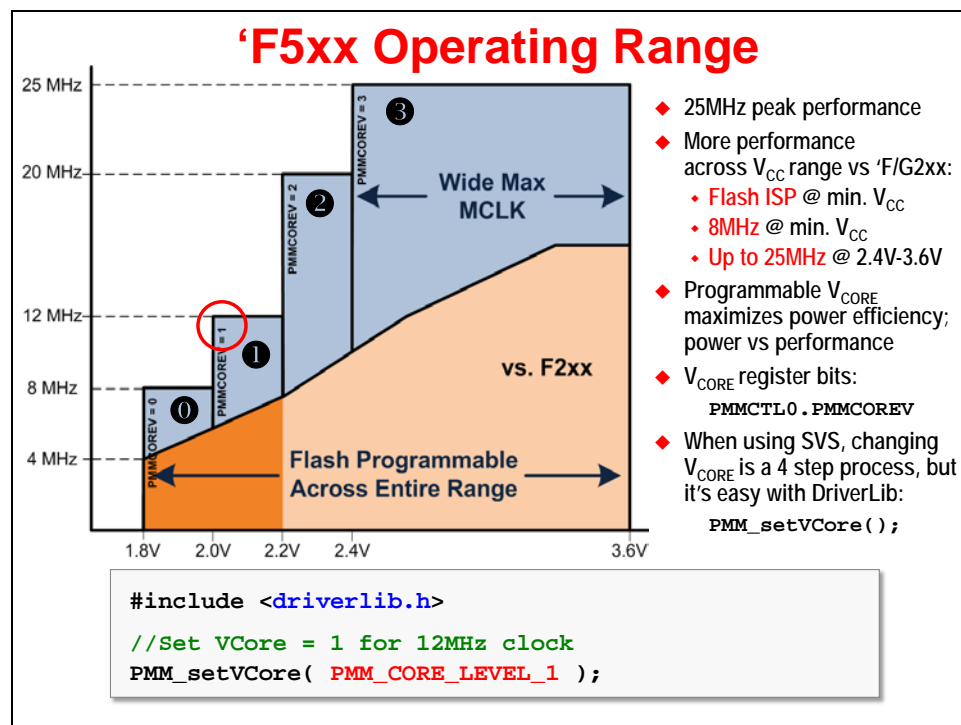


Operating Voltages

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

Two examples of this are shown below:

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



The advantage to running with lower power voltage settings is that you, well, save power. The tradeoff is that you give up capability when you run at the lower settings. Then again, you could always change the Vcore setting on-the-fly, as needed by your application at any given time.

One big advantage of the new 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

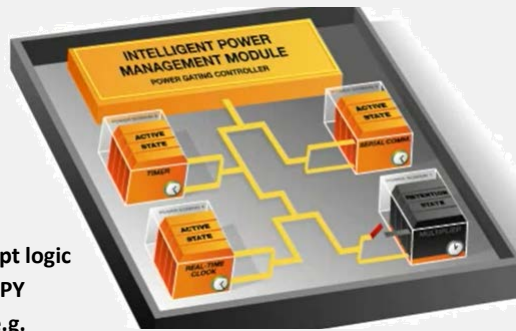
We have summarized three MSP430 devices in the table below. They demonstrate some of the differences between the various series of MSP430: Value-Line, F5xx, and FR5xx FRAM.

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 Regulator s (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...

Wolverine Power Gating ('FR58/59)

- ◆ Enhanced clock system
- ◆ Each module has a clock enable line
- ◆ If CE line is not in use the domain is powered down



Domain 1: Always ON CPU, Interrupt logic

Domain 2: Always OFF, AES, HW MPY

Domain 3/4: Peripheral Domain for e.g. timers

Completely transparent to the user

This slide shows a bit more information regarding the voltage supervision/monitoring.

Voltage Supervision & Monitoring

SVS / SVM disabled

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

High-side Full Performance Mode

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

5 us wakeup from LPMx

Maximum Robustness

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

Power on Default Mode

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

150 us wakeup from LPMx

High-side Fast Performance Mode

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

Current

Initialization Summary (template)

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
<i>Lab 4 - Abstract.....</i>	<i>4-41</i>
<i>Lab 4 Worksheet.....</i>	<i>4-43</i>
Hints:	4-43
Reset and Operating Modes & Watchdog Timers	4-43
Power Management	4-43
Clocking.....	4-43
<i>Lab 4a – Program the MSP430 Clocks.....</i>	<i>4-47</i>
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
<i>(Optional) Lab 4b – Exploring the Watchdog Timer.....</i>	<i>4-57</i>
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
<i>(Optional) Lab 4c – Using Crystal Oscillators</i>	<i>4-63</i>
File Management	4-63
Modify GPIO.....	4-64
Debug.....	4-65
<i>Chapter 04 Appendix</i>	<i>4-66</i>

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

- Name all 3 types of resets:

- If the Watchdog (WDT) times out, which reset does it invoke?

- Write the DriverLib function that stops (halts) the watchdog timer:

_____ (WDT_A_BASE);

Power Management

F5529

- (**F5529 Launchpad users only**) Write the DriverLib function that sets the core voltage needed to run MCLK at 8MHz.

_____ (_____);

Clocking

- 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 and 'FR5969:

```
#define LF_CRYSTAL_FREQUENCY_IN_HZ _____
```

```
#define HF_CRYSTAL_FREQUENCY_IN_HZ _____
```

'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".)

```
_____ ( LF_CRYSTAL_FREQUENCY_IN_HZ ,  
          HF_CRYSTAL_FREQUENCY_IN_HZ );
```

```
_____ ( 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/FR4311 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 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.)

FR4133

```
#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 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.)

FR5969

```
// 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.

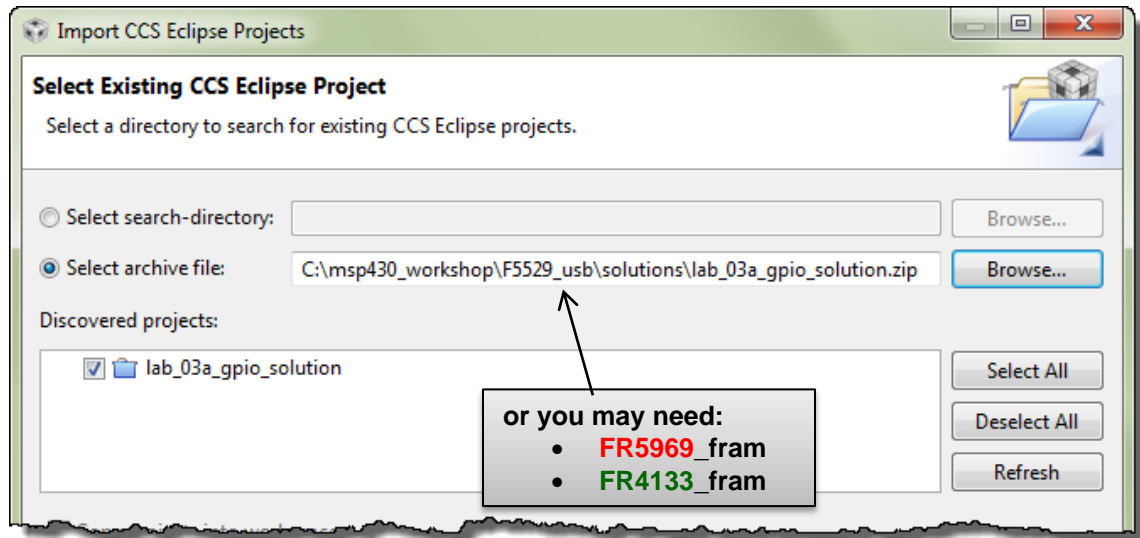
(Find them in the Chapter 4 Appendix)

Lab 4a – Program the MSP430 Clocks

File Management

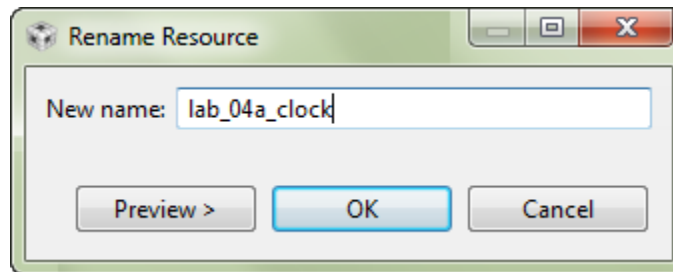
1. Import previous lab_03a_gpio solution.

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).

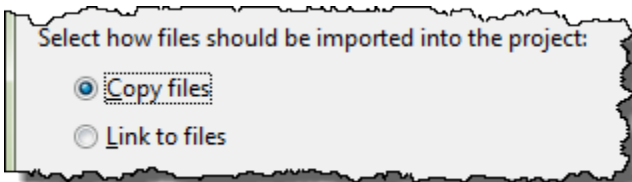
Since there can be quite a few lines of code (if you setup all the clocks), we decided to place the clock initialization into its own file.

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 #6Worksheet
Question #0Worksheet
Question #4Worksheet
Question #7Worksheet
Question #8Worksheet
Question #10Worksheet
Question #0

```

/***** 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 #6Worksheet
Question #0Worksheet
Question #7Worksheet
Question #8Worksheet
Question #10Worksheet
Question #0

```

/***** 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 _____/(REFCLK_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
        CS_CLOCK_DIVIDER_1 ); // Clock source
                                // Divide down clock source

    // Set the FLL's clock reference clock source
    CS_clockSignalInit(
        CS_FLLREF, _____, // Clock you're configuring
        CS_CLOCK_DIVIDER_1, // Clock source
        _____ ); // 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 #6Worksheet
Question #7Worksheet
Question #8Worksheet
Question #10Worksheet
Question #11Worksheet
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.

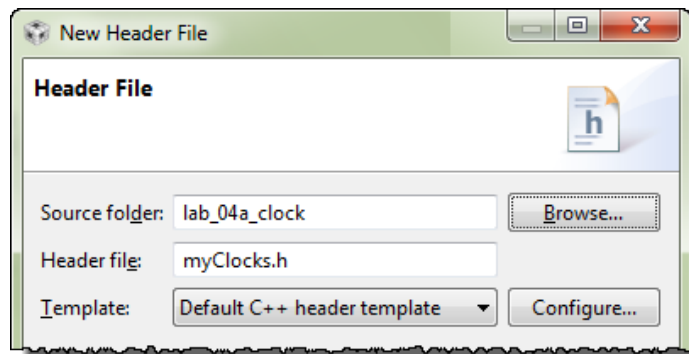
Hopefully you don't have any typographic or syntax errors, but you should see this error:

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

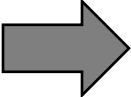


Then click 'Finish'.

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.

- Add a prototype for a new function `initGPIO()`.
- Call `initGPIO()` and `initClocks()` from the main.
- 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 initialization 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**12. (FRAM devices only) Unlock the pins.****FR4133**

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.**

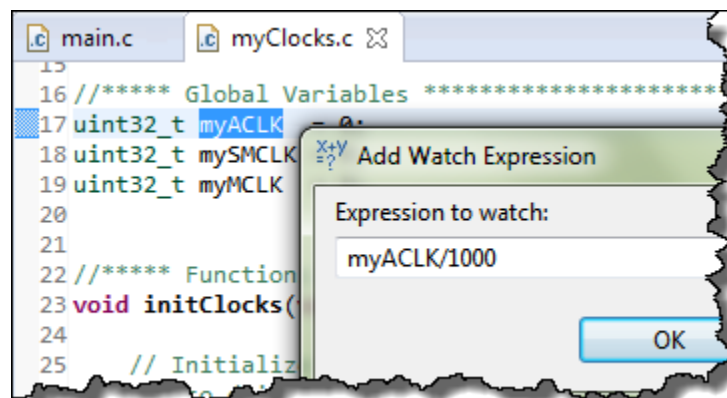
Debugging the Clocks

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.

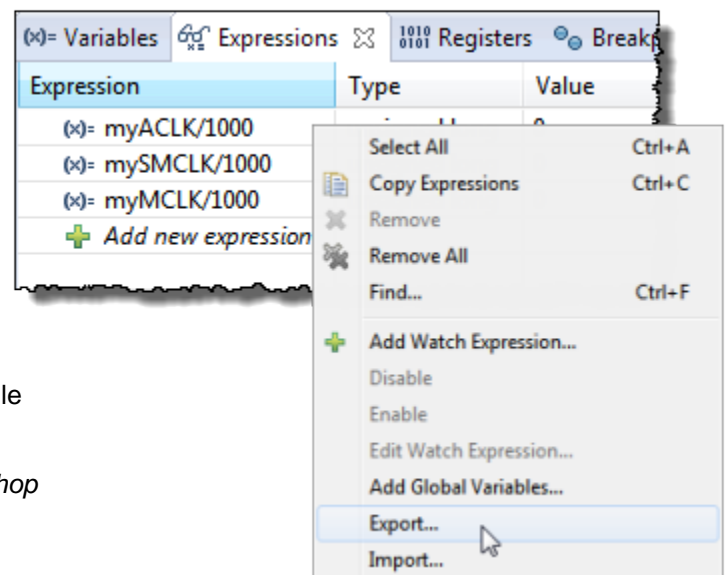
16. Go ahead and create similar watch expressions for `mySMCLK` and `myMCLK`.

```
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.

- Right-click on *Expressions* window
- Select *Export...*
- 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.)

F5529

```

Verify if the default clock
33 myACLK = UCS_getACLK();
34 myMCLK = UCS_getMCLK();
40 myMCLK = UCS_getMCLK();
41
42 // Setup ACLK to use REFO as its source
43 UCS_clockSignalInit(
44     UCS_ACLK,
45     UCS_REFOCLK_SELECT,
46     UCS_CLOCK_DIVIDER_1
47 );
48
49 // Verify that the modified clock is correct
50 myACLK = UCS_getACLK();
51 myMCLK = UCS_getMCLK();
52
53 }
  
```

FR5969

```

Verify if the default clock
33 myACLK = CS_getACLK();
34 mySMCLK = CS_getSMCLK();
35 myMCLK = CS_getMCLK();
36
37 // Setup ACLK to use VLO as its source
38 CS_clockSignalInit(
39     CS_ACLK,
40     CS_VLOCLK_SELECT,
41     CS_CLOCK_DIVIDER_1
42 );
43
44 // Verify that the modified clock is correct
45 myACLK = CS_getACLK();
46 mySMCLK = CS_getSMCLK();
47 myMCLK = CS_getMCLK();
48
49 }
  
```

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.

```

49 // Configure the FLL's frequency and set
50 UCS_initFLLSettle( UCS_BASE,
51                   MCLK_DESIRED_FREQUENCY_IN_KHZ,
52                   MCLK_FLLREF_RATIO
53 );
54
55 UCS_clockSignalInit( UCS_BASE,
56                     UCS_MCLK,
57                     UCS_REFOCLK_SELECT,
58                     UCS_CLOCK_DIVIDER_1
59 );
60
61 // Verify that the modified clock setti
62 myACLK = UCS_getACLK( UCS_BASE );
63 mySMCLK = UCS_getSMCLK( UCS_BASE );

```

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 loooooong 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 second, 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 use 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

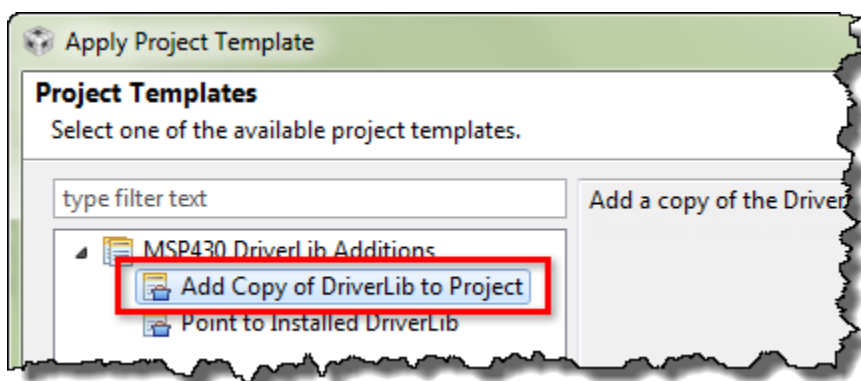
9. Rename the project to: lab_04b_wdt

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:

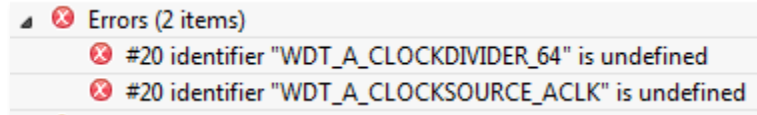
```
WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
```

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 relate to these values:

- ÷ Default: 4 (i.e. ÷32K)
- ÷ 64: 7
- ÷ 512: 6
- ÷ 32K: 4

Keep it Running

20. Add the function call that will keep the CPU running without a watchdog reset.

Add the line of code to the while{} loop – our answer to question # in this lab – that will reset the watchdog and keep the program running.

```
WDT_A_resetTimer( WDT_A_BASE );
```

Hint: You may want to change the clock divisor back to WDT_A_CLOCKDIVER_64 to make it easier to see the change. Then, if the count goes past “1” you’ll know the watchdog is being serviced.

21. Build and run the program to observe the watchdog resetting the MSP430.

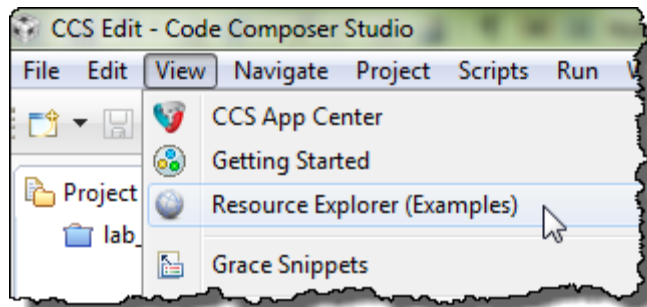
How many times will it run now? _____

22. When done playing with the program, terminate your debug session close the project.

Extra Credit – Try DriverLib's Watchdog Example (#3)

The driverlib library contains an example for 'watching' the watchdog timer. Give it a test to watch every time the watchdog rolls-over.

23. Import the `wdt_a_ex3_watchdogACLK` project using the 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 +         // HFXIN 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 crystal pins works with either solution... so we chose the one with less typing.

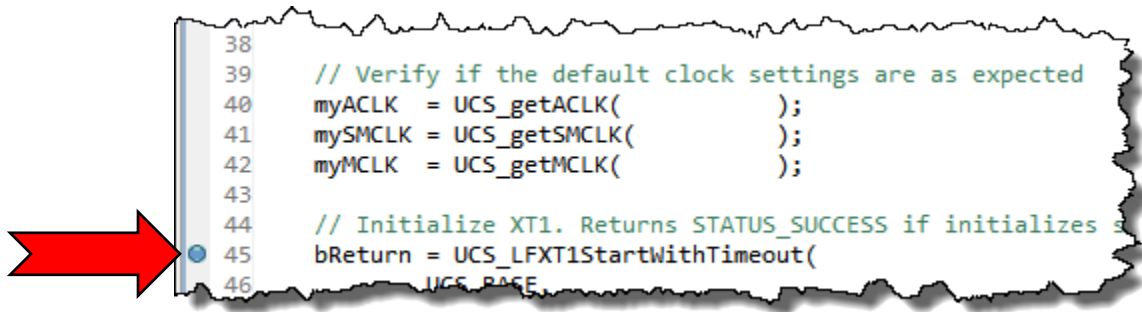
9. Build and launch the debugger.

Debug

10. Set three breakpoints in the `myClocksWithCrystals.c` file.

Set a breakpoint after each instance of the code where we read the clock settings.

For example:



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, terminate the debugger and close the project.

Chapter 04 Appendix

Hints:

Chapter 4 Worksheet (1)

- ◆ The MSP430 DriverLib Users Guide will be useful in helping to answer these workshop questions. Find it in your MSP430ware DriverLib doc folder:
e.g. \MSP430ware_1_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\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?

'F5529 ACLK: 32 KHz SMCLK: 1.048 MHz MCLK: 1.048 MHz
'FR5969 ACLK: 39 KHz SMCLK: 1 MHz MCLK: 1 MHz

Chapter 4 Worksheet (6)

10. Set up ACLK:

- Use REFO for the F5529 device
- Use VLO for the FR5969/4311 devices

F5529 Prefix = 'UCS'

FR5969 Prefix = 'CS'

FR4311 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 );
```