Timers

Introduction

Timers are often thought of as the heartbeat of an embedded system.

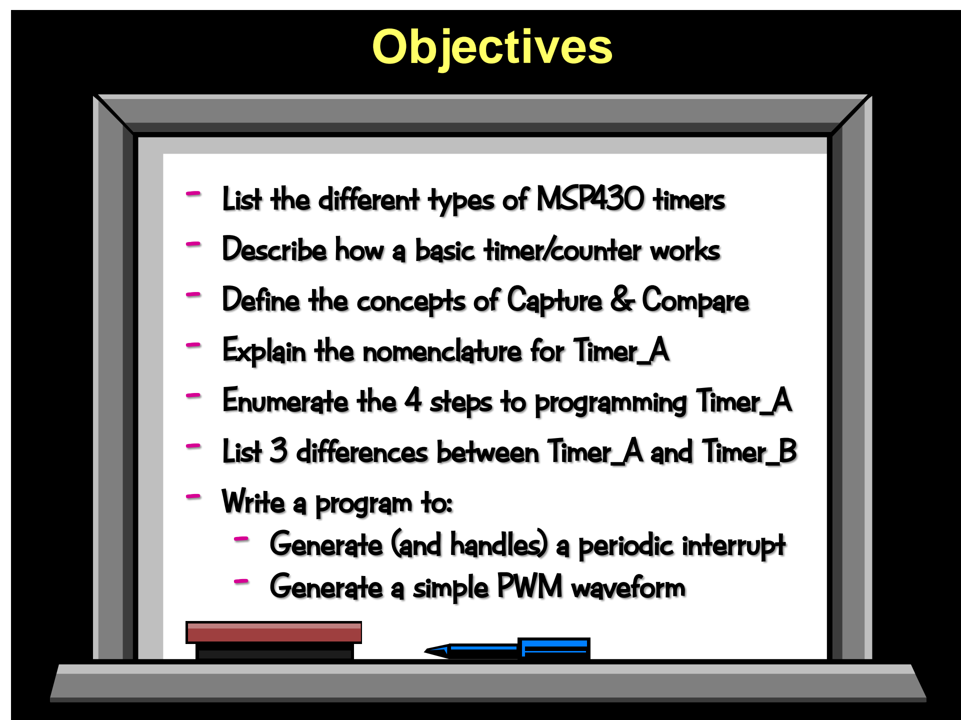
Whether you need a periodic wake-up call, a one-time delay, or need a means to verify that the system is running without failure, Timers are the solution.

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

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

As the chapter nears conclusion, there's a brief summary of the differences between TIMER_A and TIMER_B. Bottom line, if you know how to use TIMER_A, then you can use TIMER_B; but, there are a couple of extra features that TIMER_B provides.

Learning Objectives



Chapter Topics

Timers **6-1**

Prerequisites and Tools *6-2*

Overview of MSP430 Timers *6-3*
TIMER_A/B Nomenclature 6-4
Timer Summary 6-5

Timer Basics: How Timers Work *6-6*
Counter 6-6
Frequency, Time-Period, Resolution 6-7
Capture 6-8
Compare 6-9

Timer Details: Configuring TIMER_A *6-12*
1. Counter: TIMER_A_configure…() 6-13
Timer Counting Modes 6-14
Summary of Timer Setup Code – Part 1 6-18
2a. Capture: TIMER_A_initCapture() 6-19
2b. Compare: TIMER_A_initCompare() 6-21
Summary of Timer Setup Code – Part 2 6-23
Output Modes 6-24
PWM anyone? 6-29
3. Clear Interrupt Flags and TIMER_A_startTimer() 6-30
4. Interrupt Code (Vector & ISR) 6-31

TIMER_A DriverLib Summary *6-32*

Differences between Timer's A and B *6-33*

Lab Exercise *6-35*

Prerequisites and Tools

To get full entitlement from this chapter, we expect that you are already familiar with MSP430ware's DriverLib as well as MSP430 clocking and interrupts. The "extra" piece of hardware required for this chapter is a single jumper wire.

Prerequisites & Tools

- **Skills** — **Chapter**
  - Creating a CCS Project for MSP430 Launchpad(s) — (Ch 2 & 3)
  - Basic knowledge of:
    - C language
    - Setting up MSP430 clocks — (Ch 4)
    - Using interrupts (setup and ISR's) — (Ch 5)
- **Hardware**
  - Windows 7 (and 8) PC with available USB port
  - MSP430F5529 Launchpad or MSP430FR5969 Launchpad (with included USB micro cable)
  - One (1) jumper wire (female to female)
- **Software**
  - CCSv6
  - MSP430ware

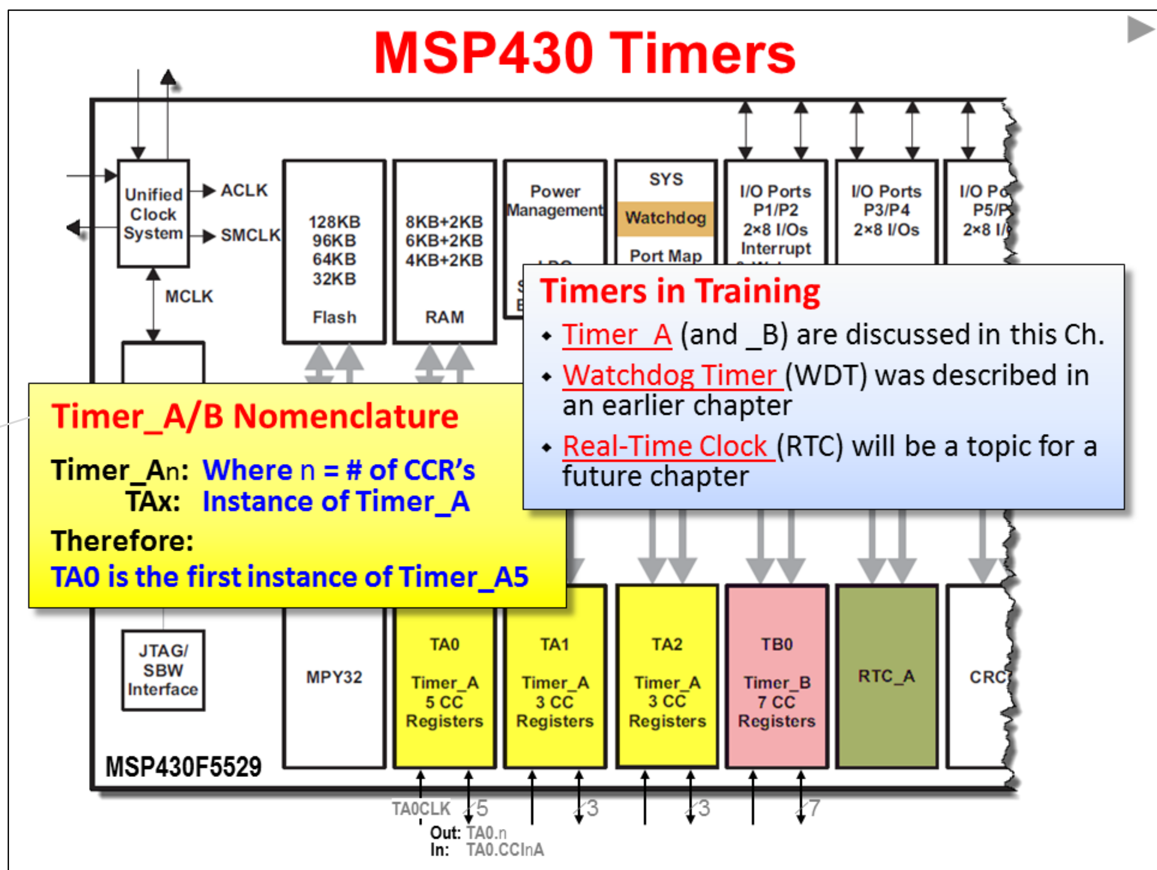
Overview of MSP430 Timers

The MSP430F5529 timers are highlighted in the following block diagram.

- Yellow marks the three instances of the TIMER_A module.
- Pink was used for TIMER_B.
- Dark brown highlights the real-time clock (RTC_A).
- Light brown differentiates the Watchdog timer inside the SYS block

The “Timers in Training” callout box describes where the various timers are discussed in this workshop. Timers A and B are covered in this chapter. We have already covered the Watchdog timer in a previous chapter.

The RTC module will be discussed in a future chapter. A brief description of the RTC tells us that it’s a very low-power clock; has built-in calendar functions; and often includes “alarms” that can interrupt the CPU. It is frequently used for keeping a time-base while the CPU is in low-power mode.



Nomenclature is discussed on the next page

TIMER_A/B Nomenclature

The nomenclature of the TIMER_A and _B peripherals is a little unusual. First of all, you may have already noticed that the MSP430 team often adds one of two suffixes to their peripheral names to indicate when features have been added (or modified).

- Some peripherals, such as the Watchdog Timer go from “WDT” to “WDT+”. That is, they add a “+” to indicate the peripheral has been updated (usually with additional features).
- Other peripherals are enumerated with letters. For example, three sophisticated MSP430 timers have been introduced: TIMER_A, TIMER_B, and TIMER_D. *(What happened to _C? Even I don’t know that. <ed>)*

The use of a suffix is the generic naming convention found on the MSP430. With the timers, though, there are a couple more naming variations to be discussed.

As we will cover in great detail during this chapter, these timers contain one or more Capture and Compare Registers (CCR); these are useful for creating sophisticated timings, interrupts and waveforms. The more CCR registers a timer contains, the more independent waveforms that can be generated. To this end, the documentation often includes the number of CCR registers when listing the name of the timer. For example, if TIMER_A on a given device has 5 CCR registers, they often name it:

```
Timer_A5
```

But wait, that’s not all. What happens when a device, such as the ‘F5529 has more than one instance of TIMER_A? Each of these instances needs to be enumerated as well. This is done by appending the instance number after the word “Timer”, as in Timer0.

To summarize, here’s the long (and short) names for each of the ‘F5529 TIMER_A modules:

Instance	Long Name	Short Name
0	Timer0_A5	TA0
1	Timer1_A3	TA1
2	Timer2_A3	TA2

Timer Summary

The ‘F5529 contains most of the different types of timers found across the MSP430 family; in fact, the only type of timer not present on this device is the high-resolution TIMER_D.

The following summary provides a snapshot of what timers are found on various MSP430 devices. You’ll find our ‘F5529 and ‘FR5969 devices in the last two columns of the table.

A one-line summary of each type of timer is listed below the table.

MSP430 Timers

	L092	G2553	FR4133	F5172	F5529	FR5969
Timer_A	2 x A3	2 x A3	2 x A3	1 x A3	1 x A5 2 x A3	2 x A3 2 x A2*
Timer_B					1 x B7	1 x B7
Timer_D				2 x D3		
Real-Time Clock			RTC Counter		RTC_A	RTC_B
Watchdog	WDT_A	WDT+	WDT_A	WDT_A	WDT_A	WDT_A

Timer_A: ‘A3’ means it has 3 Capture/Compare Registers (used to generate signals & ints)
Timer_B: Same as A, but improves PWM
Timer_D: Same as B, adding hi-res timing
WDT+: Watchdog or Interval Modes; PSW Protected; Can stop; Select Clk; Clk fail-safe
WDT_A: Same as WDT+, but with 8 timer intervals rather than 4
BT1/RTC: Basic timer has 2x8-bit counters (can use as 1x16-bits) with calendar functions
RTC_A: 32-bit counter with a calendar, flexible programmable alarm, and calibration
RTC_B: Same as RTC_A, but adds switchable battery backup in case main-power fails

Timer Basics: How Timers Work

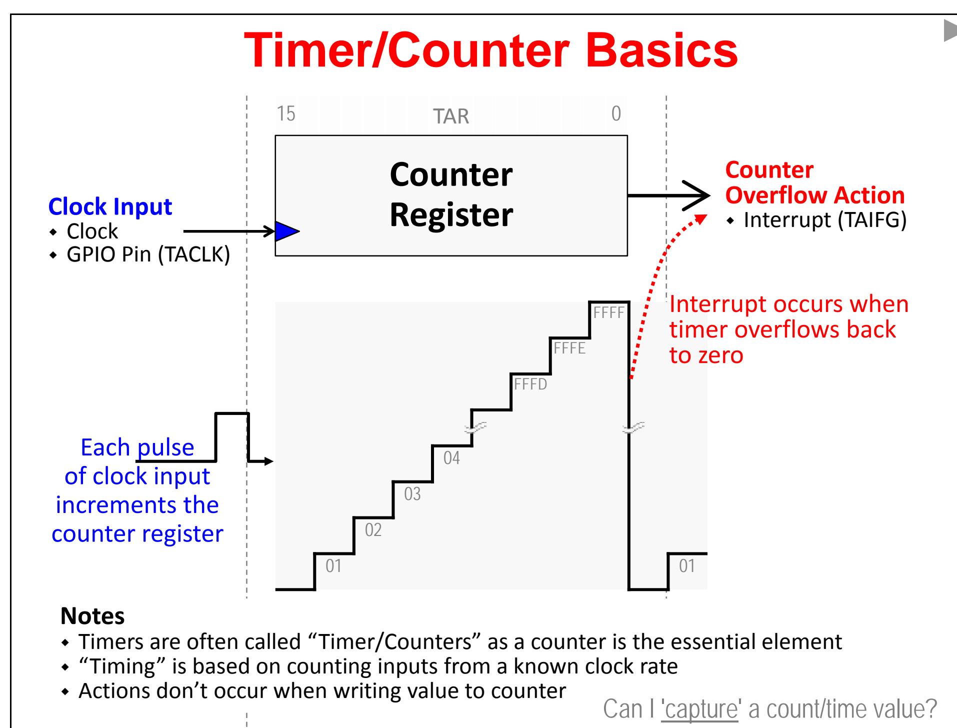
Before we discuss the details of TIMER_A, let's begin with a quick overview describing how timers work. Specifically, we will start by describing how a timer is constructed using a **Counter**. Next, we'll investigate the **Capture** and **Compare** capabilities found in many timers.

Counter

A **counter** is the fundamental hardware element found inside a timer.

The other essential element is a **clock** input. The counter is incremented each time a clock pulse is applied to its clock input. Therefore, a 16-bit timer will count from zero (0x0000) up to 64K (0xFFFF).

When the counter reaches it reaches its maximum value, it overflows – that is, it returns to zero and starts counting upward again. Most timer peripherals can generate an interrupt when this overflow event occurs; on TIMER_A, the interrupt flag bit for this event is called TAIFG (TIMER_A Interrupt Flag).



The clock input signal for TIMER_A (named TACLK) can be one of the internal MSP430 clocks or a signal coming from a GPIO pin.

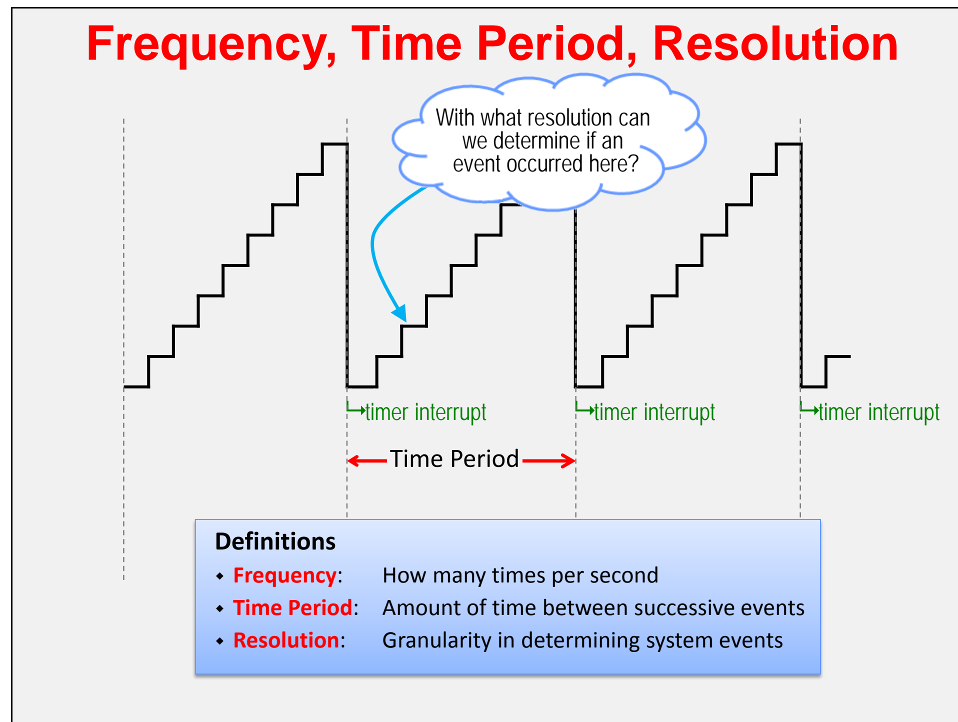
Many engineers call these peripherals "Timer/Counters" as they provide both sets of functionality. They can generate interrupts or waveforms at a specific time-base – or could be used to count external events occurring in your system.

One final note about the MSP430 timers: they do not generate interrupts (or other actions) when you write to the counter register. For example, writing "0" to the counter won't generate the TAIFG interrupt.

Frequency, Time-Period, Resolution

The Timer's ability to create a consistent, periodic interrupt is quite valuable to system designers. *Frequency* and *Time Period* are two terms that are often used to describe the rate of interrupts.

- How many times per second that a timer creates an interrupt defines its ***Frequency***.
- Conversely, the amount of time in-between interrupt events is defined as the ***Time Period***.



If a timer only consisted of a single counter, its *resolution* would be limited to the size of the counter.

If some event were to happen in a system – say, a user pushed a button – we could only ascertain if that event occurred within a time period. In other words, we can only determine if it happened between two interrupts.

Looking at the above diagram, we can see that there is "more data" available – that is, if we were to read the actual counter value when the event occurred. Actually, we can do this by setting up a GPIO interrupt; then, having the ISR read the value from the counter register. In this case the resolution would be better, but it is still limited by:

- It takes more hardware (an extra GPIO pin is needed)
- The CPU has to execute code – this consumes power and processing cycles
- The resolution is less deterministic because it's based upon the latency of the interrupt response; in other words, how fast can the CPU get to reading the counter … and how consistent can this be each time it occurs

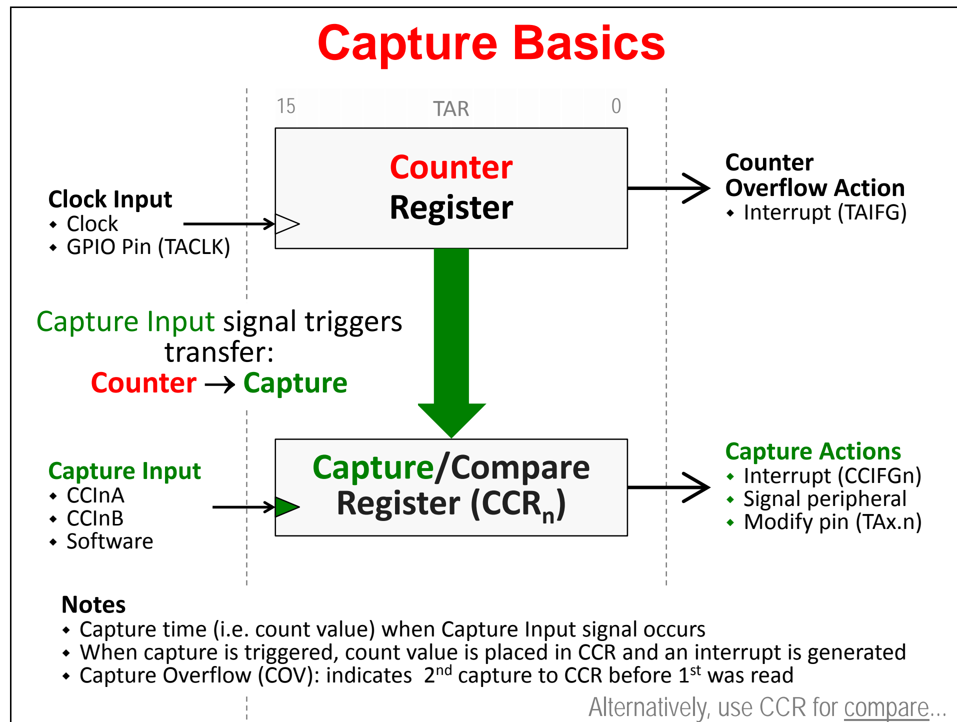
There is a better way to implement this in your system … turn the page and let's examine the timer's ***Capture*** feature.

Capture

The ***Capture*** feature does just that. When a capture input signal occurs, a snapshot of the Counter Register is *captured*; that is, it is copied into a capture register (CCR for Capture and Compare Register). This is ideal since it solves the problems discussed on the previous page; we get the timer counter value captured with no latency and very, very little power used (the CPU isn't even needed, so it can even remain in low-power mode).

The diagram below builds upon our earlier description of the timer. The top part of the diagram is the same; you should see the Counter Register flanked by the Clock Input to the left and TAIFG action to the right.

The bottom portion of the slide is new. In this case, when a Capture Input signal occurs, the value from the Counter Register is copied to a capture register (i.e. CCR).



A few notes about the *capture* feature:

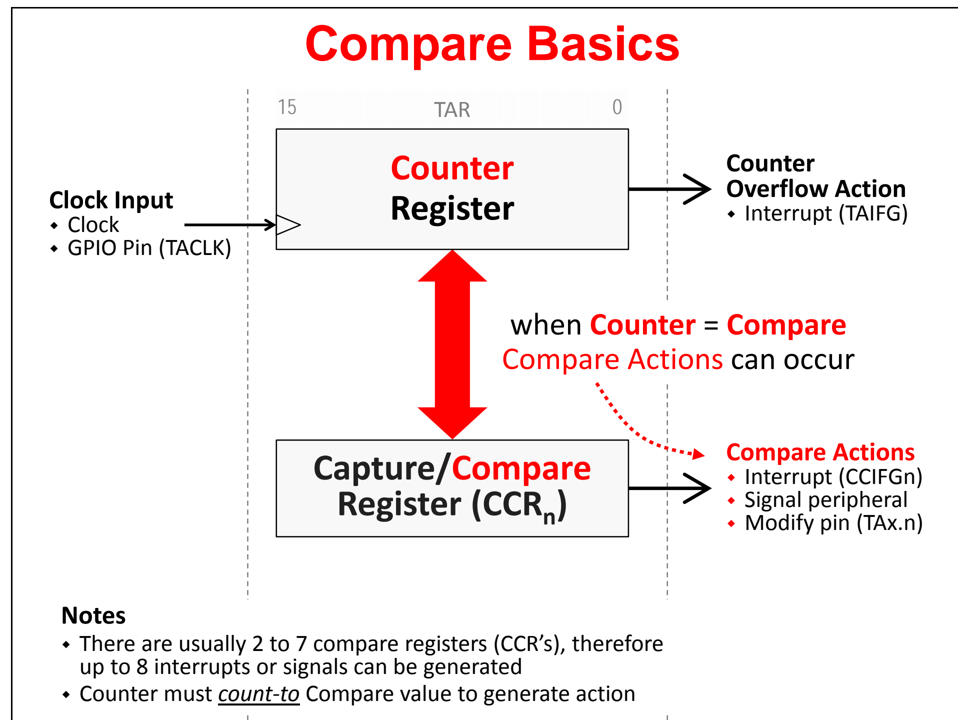
- As we discussed earlier, the MSP430 timers (TIMER_A, TIMER_B, and TIMER_D) have multiple CCR registers; check your datasheet to determine how many are available per timer peripheral. Each CCR, though, has its own capture input signal.
- The Capture Input signal can be connected to a couple of different signals (CCInA, CCInB) or triggered in software
- The Capture Input hardware signals (CCInA, CCInB) are connected differently for each CCR register and device. You need to reference the datasheet to verify what options are available on your specific device.
- When a capture occurs, the CCR can trigger further actions. This "action" signal can generate an interrupt to the CPU, trigger another peripheral, and/or modify the value of a pin.

As we just discussed, the Capture feature provides a deterministic method of capturing the count value when triggered. While handy, there is another important requirement for timers…

Compare

A key feature for timers is the ability to create a consistent, periodic interrupts.

As we know, TIMER_A can do this, but the timer's frequency (i.e. time period) is limited to dividing the input clock by 2^{16}. So, while the timer may be *consistent*, but not very flexible. Thankfully, the ***Compare*** feature of TIMER_A (TIMER_B & TIMER_D) solves this problem.



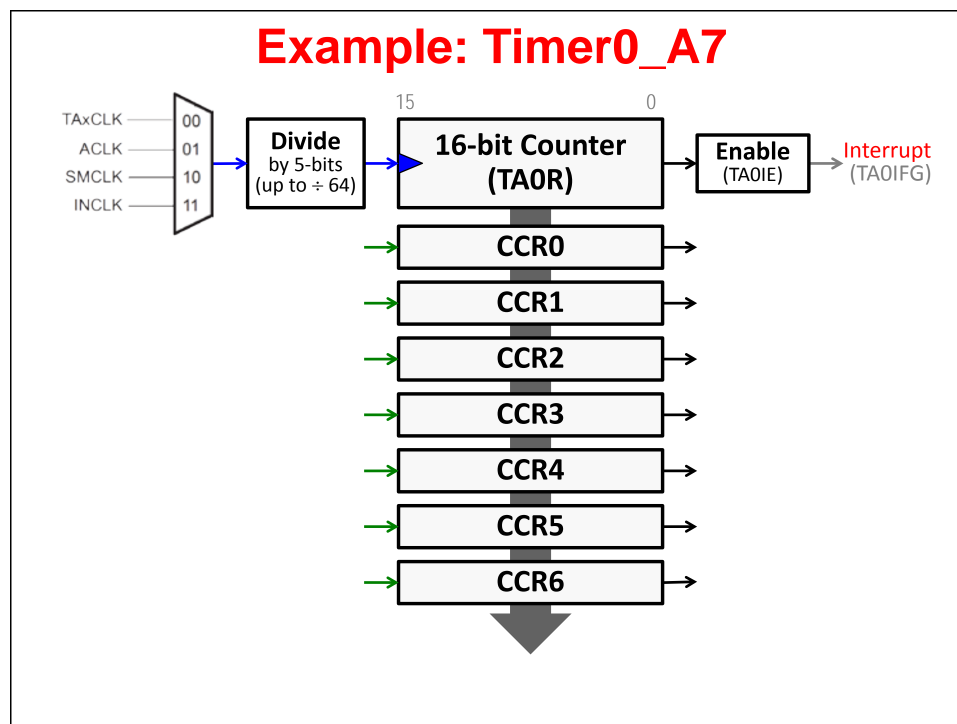
Once again, the top portion of this diagram remains the same (Clock Input + Counter Register).

The bottom portion of the diagram differs from the previous diagrams. In this case, rather than using the CCR register for capture, it's used as a *compare* register. In this mode, whenever a match between the Counter and Compare occurs, a compare action is triggered. The compare actions include generating an interrupt, signaling another peripheral (e.g. triggering an ADC conversion), or changing the state of an external pin.

The "modify pin" action is a very powerful capability. Using the timer's *compare* feature, we can create sophisticated PWM waveforms. (Don't worry, there's more about this later in the chapter.)

Timer Summary – showing multiple CCR's

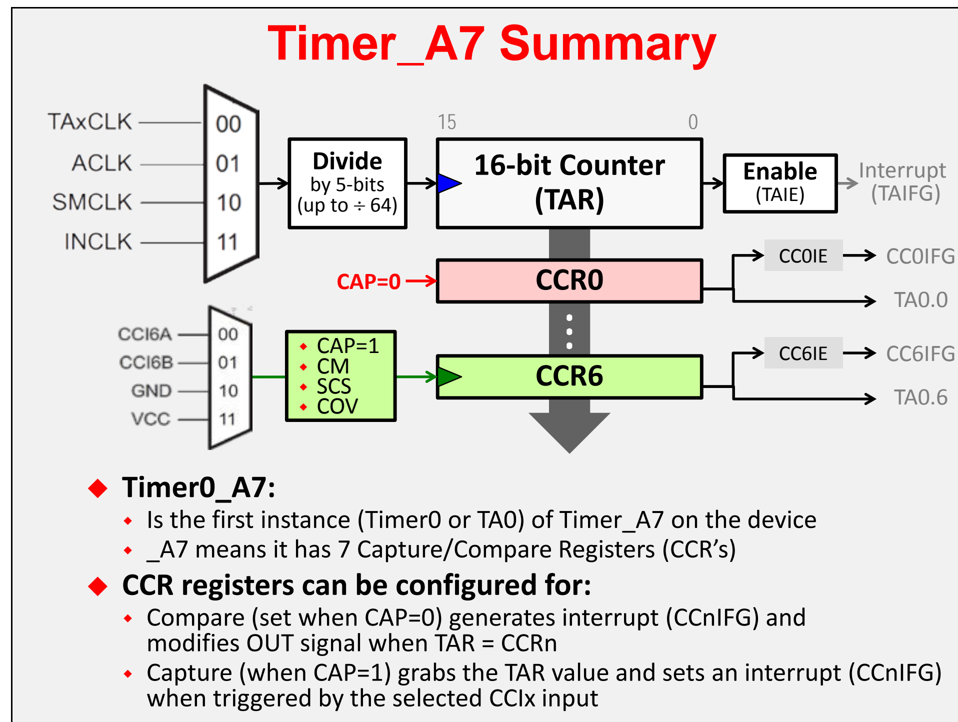
The following example of a Timer0_A7 provides us a way to summarize the timer's hardware.



Remember:

- Timer0 means it's the first instance of Timer_A on the device.
- _A7 means that it's a Timer_A device and has 7 capture/compare registers (CCR's)
- The clock input, in this example, can be driven by a TACLK signal/pin, ACLK, SMCLK or another internal clock called INCLK.
- The clock input can be further divided down by a 5-bit scalar.
- The TA0IE interrupt enable can be used to allow (or prevent) an interrupt (TA0IFG) from reaching the CPU whenever the counter (TA0R) rolls over.

This next diagram allows us to look more closely at the Capture and Compare functions.



Every CCR register has its own control register. Notice above, that the "CAP" bit configures whether the CCR will be used in capture (CAP=1) or compare mode (CAP=0).

You can also see that each CCR has an interrupt flag, enable, and output signal associated with it. The output signal can be routed to a pin or a number of other internal peripherals.

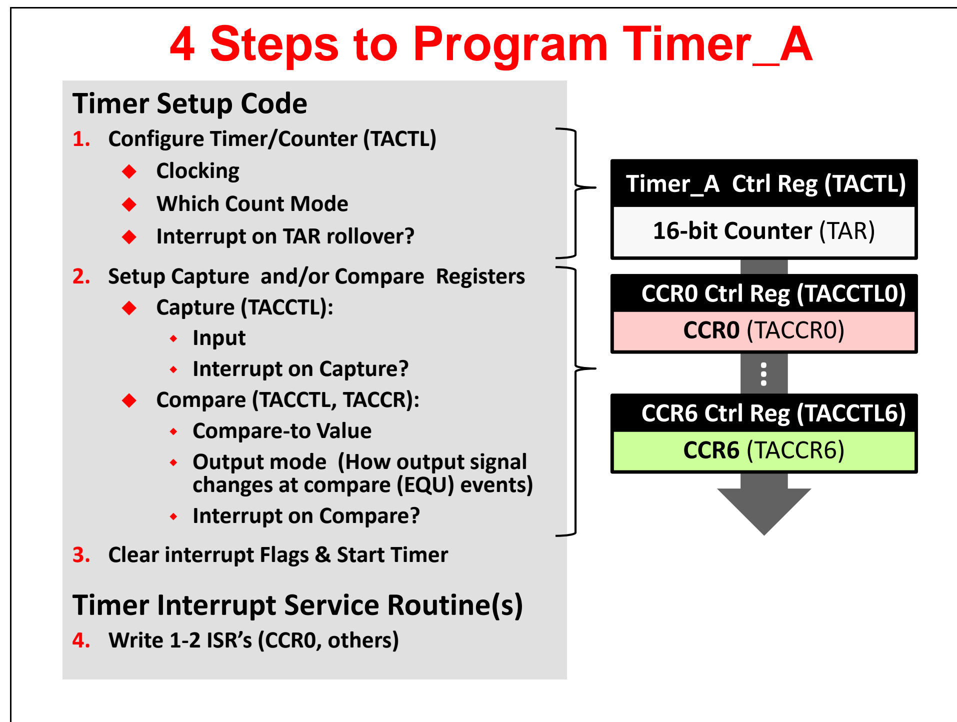
As we go through the rest of this chapter, we'll examine further details of the CCR registers as well as the various "actions" that the timer generates.

In the next section, we'll begin examining how to configure the timer using the MSP430ware DriverLib API.

Timer Details: Configuring TIMER_A

There are four steps required to get Timer_A working in your system:

1. Configure the *main Timer/Counter* by programming the TACTL control register.
2. Setup each CCR that is needed for your application. We will examine this step from both the *Capture* and *Compare* perspective.
3. Next, you need to start the timer. (We also listed clearing the timer IFG bits, which is normally done right before starting the timer.)



4. Finally, if your timer is generating interrupts, you need to have an associated ISR for each one. *(While interrupts were covered in the last chapter, we briefly summarize this again in context of the Timer_A.)*

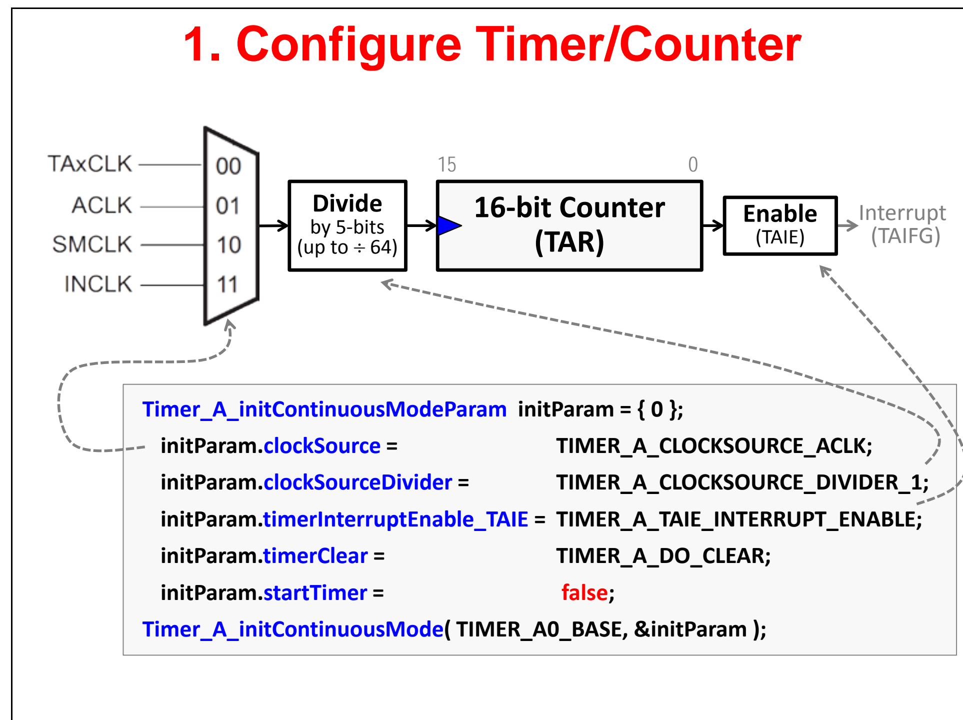
We will intermix how to write code for the timer with further examination of the timer's features.

1. Counter: TIMER_A_configure…()

The first step to using TIMER_A is to program the main timer/counter control register. The MSP430ware Driver Library provides 3 different functions for setting up the main part of the timer:

```
TIMER_A_configureContinuousMode()
TIMER_A_configureUpMode()
TIMER_A_configureUpDownMode()
```

We will address the different modes on the next page. For now, let's choose 'continuous' mode and see how we can configure the timer using the DriverLib function.



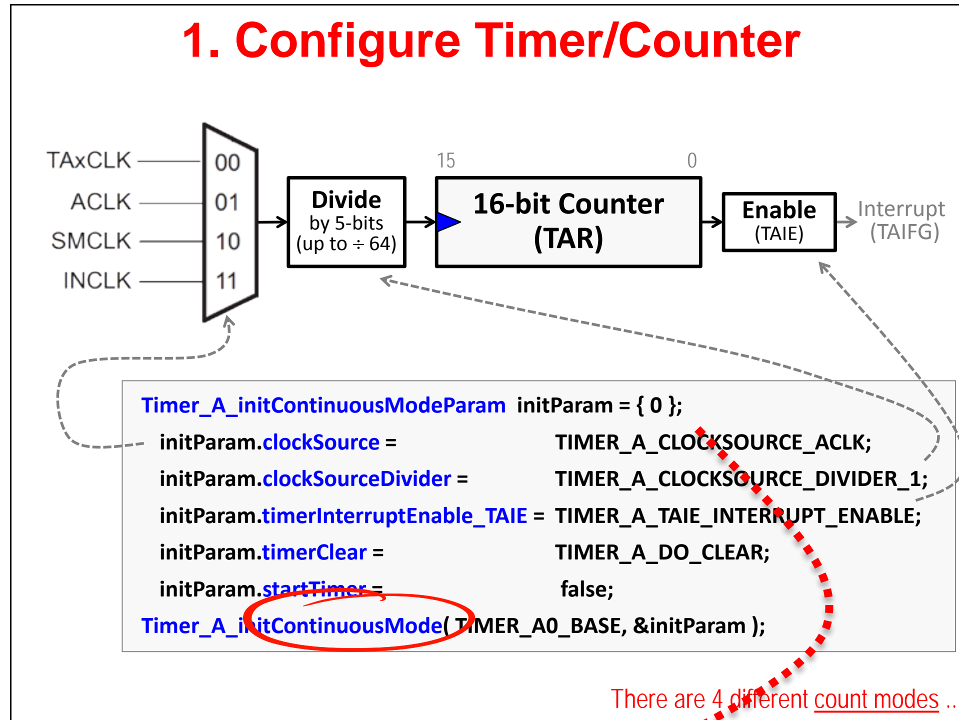
From the diagram, we can see that 3 different hardware choices need to be made for our timer configuration; the arrows demonstrate how the function parameters relate to these choices. Let's look at each parameter one-by-one:

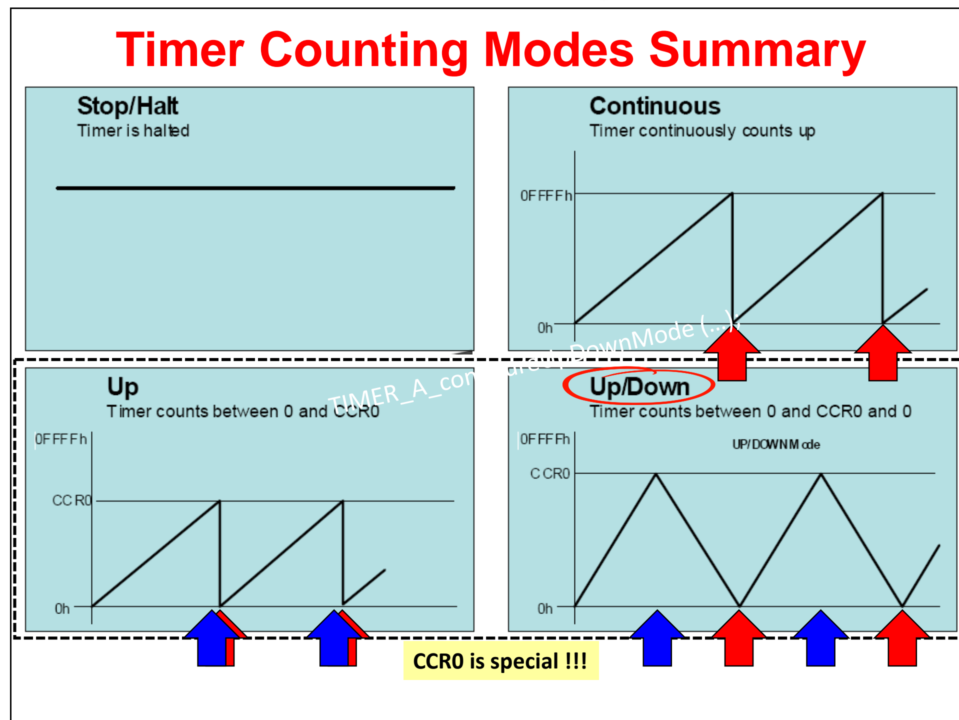
- The first parameter chooses which Timer_A instance you want to program. In our example, we have chosen to program TA0 (i.e. Timer0_A). Conveniently, the DriverLib documentation provides enumerations for all the supported choices. *(This is the same for all DriverLib parameters, so we won't keep repeating this statement. But, this is very handy to know!)*
- The 2nd parameter lets you choose which clock source you want to use. We chose SMCLK.
- The next parameter picks one of the provided clock pre-scale values. The h/w lets you choose from one of 20 different values; we picked ÷ 64.
- Parameter four lets us choose whether to interrupt the processor when the counter (TA0R) rolls over to zero. This parameter ends up setting the TA0IE bit.
- Finally, do you want to have the timer counter register (TA0R) reset when the other parameters are configured?

> **Remember…**
>
> **TAR**: Timer_A count Register
>
> **TA0R**: Name for count register when referring to instance "0" (i.e. Timer0_A)

Timer Counting Modes

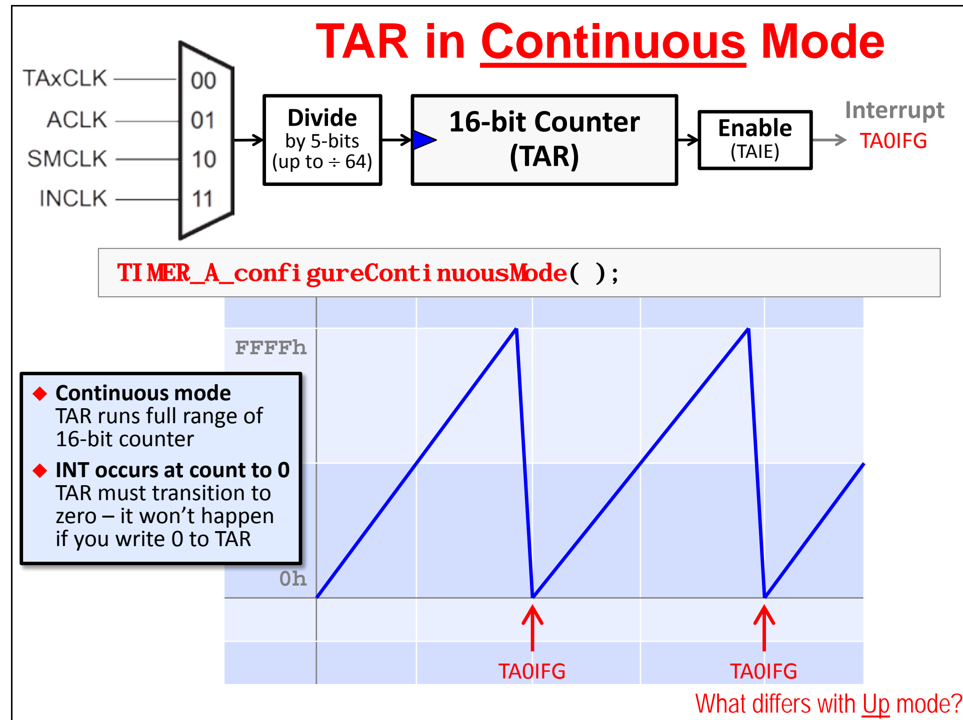
There are three different ways that the timer counter (TAR) can be incremented. These correlate to the three configuration functions listed on the previous page. This page provides a single-slide summary of the different modes – but we'll examine each one over the following three pages.





Continuous Mode

Thus far we have described the timer's counter operating in the ***Continuous*** mode; in fact, this was the configuration example we just discussed.



The different counting modes describe how the timer counter register (TAR) is incremented or decremented. For example, in *Continuous* mode, the timer counts from 0x0 up to 0xFFF and then rolls back to 0x0, where it begins counting up again. (This is shown in the diagram above.)

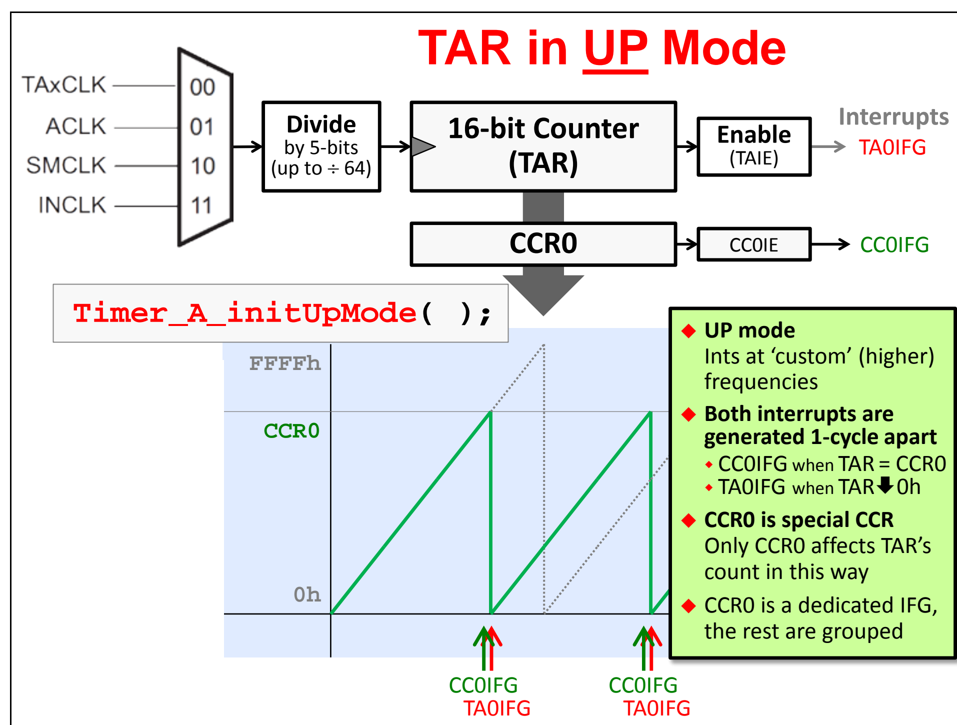
As you can see, every time the counter rolls back to zero, the TAIFG bit gets set; which, if enabled, interrupts the processor every 2^{16} input clocks. (Since our previous example was for Timer0_A, the diagram shows TA0IFG getting set.)

Up Mode

The ***Up*** counting mode differs from the *Continuous* mode by resetting back to zero whenever the counter matches CCR0 (Capture and Compare Register 0).

You can see the different waveforms compared on the slide below. The green waveform counts ***Up*** to the value found in CCR0, and then resets back to zero.

On the other hand, the grey dotted waveform shows how, when in *Continuous* mode, the counter goes past CCR0 and all the way to 0xFFFF.



In *Up* mode, since we are using the CCR0 register, the timer can actually generate two interrupts:

- CC0IFG (for Timer0_A, this bit is actually called TA0CC0IFG)
- TAIFG (for Timer0_A, this bit is called TA0IFG)

You're not seeing a color misprint; the two interrupts do not happen at the exact same time, but rather 1 cycle apart. The CC0IFG occurs when there is a compare match, while the TA0IFG interrupt occurs once the counter goes back to zero.

If you compare these two *Up* mode interrupts to the one generated in the *Continuous* mode, you'll see they occur at a more rapid frequency. This is a big advantage of the *Up* mode; your frequency is not limited to 2^{16} counts, but rather can be anywhere within the 16-bit counter's range. (The downside is that you also have to configure the CCR0 registers.)

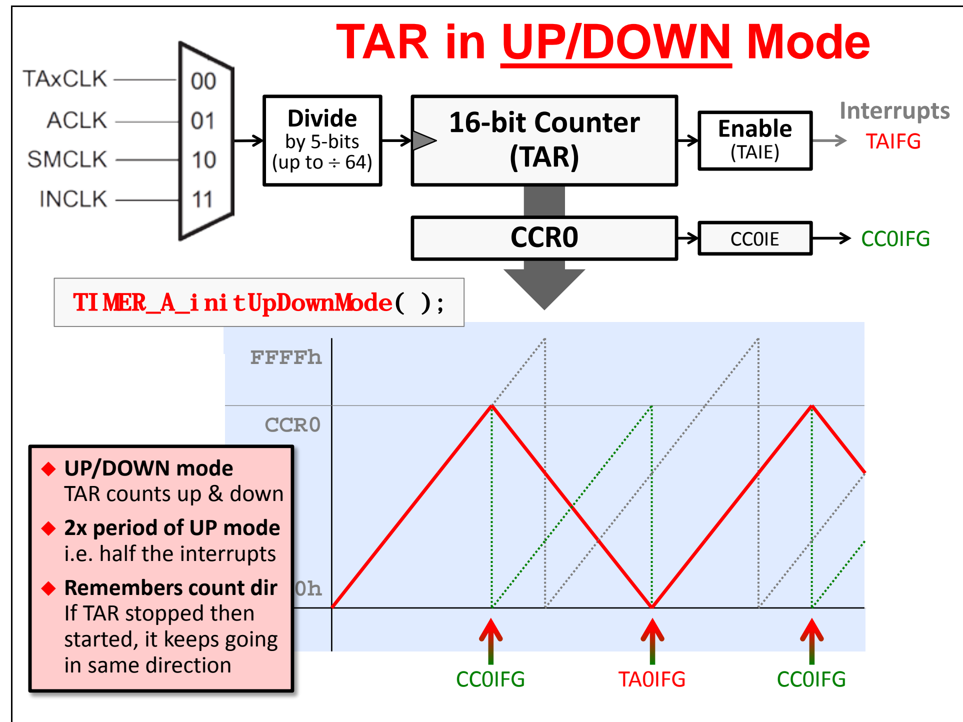
Note: The CCR0 (Capture and Control Register 0) is special. That is, it is special in comparison to the other CCR registers. It is only CCR0 that can be used to define the upper limit of the counter in Up (or UpDown) count mode.

The other special feature of CCR0 is that it provides a dedicated interrupt (CC0IFG). In other words, there is an Interrupt Vector location dedicated to CC0IFG. All the other Timer_A interrupts share a common vector location (i.e. they make up a grouped interrupt).

Up/Down Mode

The ***UpDown*** count mode is similar to *Up* in that the counter only reaches the value in CCR0 before changing. In this case, though, it actually changes direction and starts counting down rather than resetting immediately back to zero.

Not only does this double the time period (i.e. half the timer's frequency), but it also spreads out the two interrupts. Notice how CC0IFG occurs at the peak of the waveform, while TAIFG occurs at the base of the waveform.



In our diagram we show all three counter mode waveforms. The ***UpDown*** mode is shown in red; ***Up*** is shown in green; and the ***Continuous*** mode is shown in grey.

Which Count Mode Should I Use?

When using TIMER_A (or TIMER_B), you have a choice as to which counter mode to use. Here are some things to keep in mind.

- Using ***Continuous*** mode doesn't "tie up" your CCR0 register. It also means you don't have program the CCR0 register.
- ***Up*** mode allows you better control the timer's frequency – that is, you can now control the time period for when the counter resets back to zero.
- On the other hand, the ***UpDown*** mode not only lets you control the frequency better, but it also allows for lower frequencies – since it effectively halves the frequency of the ***Up*** mode.
- Two more considerations of ***UpDown*** mode are:
  - The two interrupts are spaced at ½ the time period from each other.
  - When using multiple CCR registers, you can get two compare interrupts per cycle. (We'll see more on this later.)

Summary of Timer Setup Code – Part 1

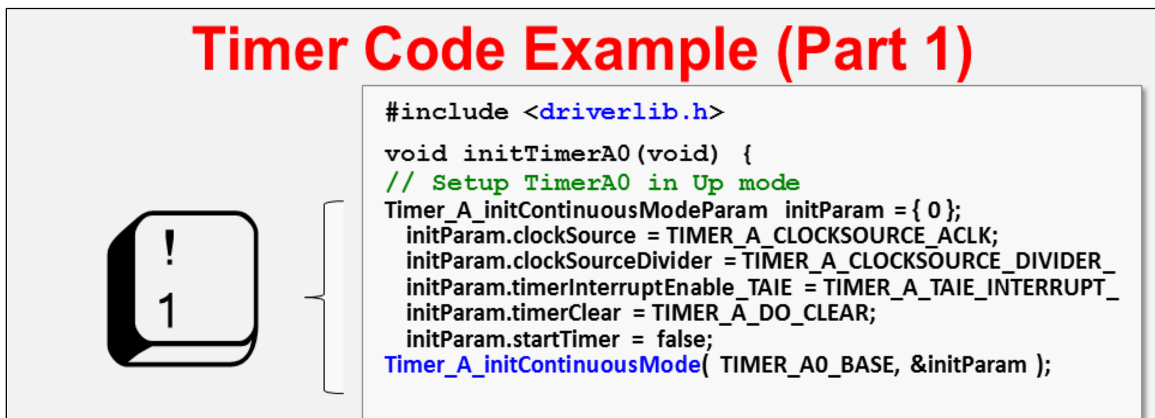
Let's summarize Part 1 of the timer setup code – which configures the timer's count options. First of all, as you can see below, we chose to place our timer setup code into its own function. Obviously, this is not a requirement, but it's how we wanted to organize our code examples.

Timer Code Example (Part 1)

```
#include <driverlib.h>

void main(void) {
  // Setup/Hold Watchdog Timer (WDT+ or WDT_A)
  initWatchdog();

  // Configure GPIO ports/pins
  initGPIO();

  // Setup Clocking: ACLK, SMCLK, MCLK (BCS+, UCS, or CS)
  initClocks();

  //----------------------------------------------------------
  // Then, configure any other required peripherals and GPIO
  initTimers();

  __bis_SR_register( GIE );

  while(1) {
  ...
  }
```

Our earlier example for the Timer/Counter setup code demonstrated using the *Continuous* mode. The following example shows using the *Up* mode. Here's a quick comparison between the two functions – notice that the *Up* mode requires two additional parameters.

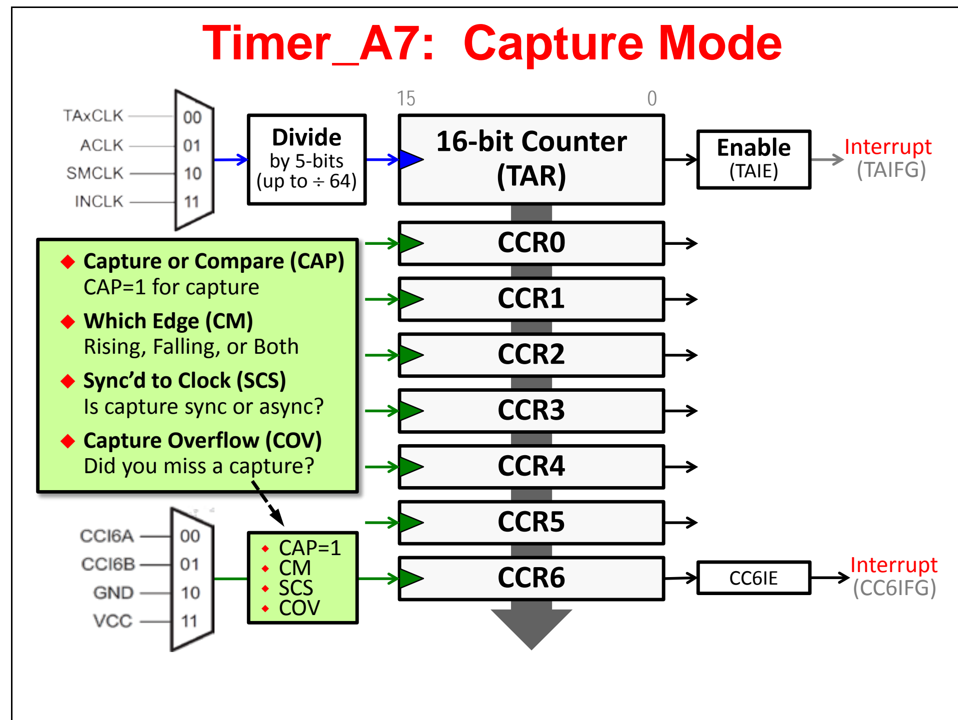
<table>
<tr><th>Parameter</th><th>ContinuousMode Function</th><th>UpMode Function</th></tr>
<tr><td>Which Timer?</td><td colspan="2">TIMER_A0_BASE</td></tr>
<tr><td>Clock Source</td><td colspan="2">TIMER_A_CLOCKSOURCE_SMCLK</td></tr>
<tr><td>Clock Pre-scaler</td><td colspan="2">TIMER_A_CLOCKSOURCE_DIVDER_xx</td></tr>
<tr><td>Timer Period</td><td>Not applicable</td><td>Used to set the CCR0 value</td></tr>
<tr><td>Enable the TAIE interrupt?</td><td colspan="2">TIMER_A_TAIE_INTERRUPT_xxxxxx</td></tr>
<tr><td>Enable the CCR0 interrupt?</td><td>Not applicable</td><td>Used to set TA0CCOIFG</td></tr>
<tr><td>Clear the counter (TAR) ?</td><td colspan="2">TIMER_A_DO_CLEAR</td></tr>
</table>



2a. Capture: TIMER_A_initCapture()

Before we try writing the code to setup a CCR register for *Capture*, let's first examine the timer's hardware options.

- Most importantly, when wanting to use the *Capture* features, you need to set CAP = 1.
- The CM bit indicates which clock edge to use for the capture input signal.
- Do you want the capture input signal sync'd with the clock input? If so, that's what SCS is for.
- While you don't configure COV, this bit indicates if a capture overflow occurred. In other words, did a 2nd capture occur before you read the captured value from the CCR register?
- Finally, you can select what hardware signal you want to have "trigger" the capture.



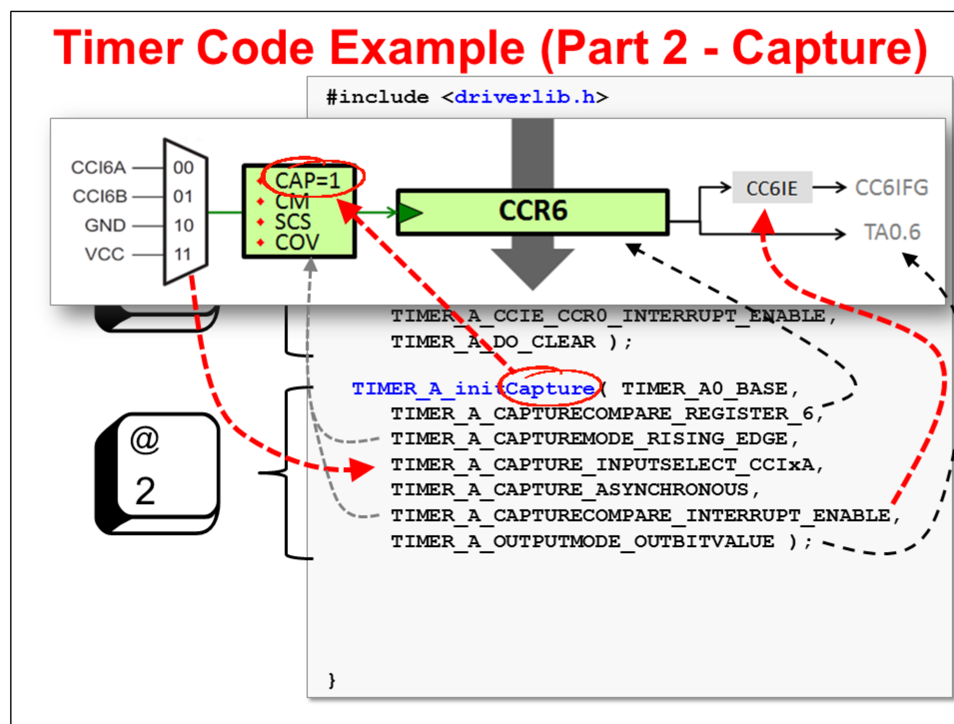
Hint: Each CCR can be configured independently. The flip side to this is that you must configure each one that you want to use; this might involve calling the 'capture' and/or 'compare' configuration functions multiple times.

Use one for capture and the rest for compare. Or, use all for capture. You get to decide how they are used.

Warning: If you are using *Up* or *UpDown* count modes, you should not configure CCR0. Just remember that the `TIMER_A_configureUpMode()` and `TIMER_A_configureUpDownMode()` configuration functions handle this for you.

Capture Code Example

With the Capture mode details in mind, let's examine the code.



To configure a CCR register for Capture mode, use the TIMER_A_initCapture() function. Thankfully, when using DriverLib the code is pretty easy to read (and maintain). Hopefully between the diagram and the following table, you can make sense of the parameters.

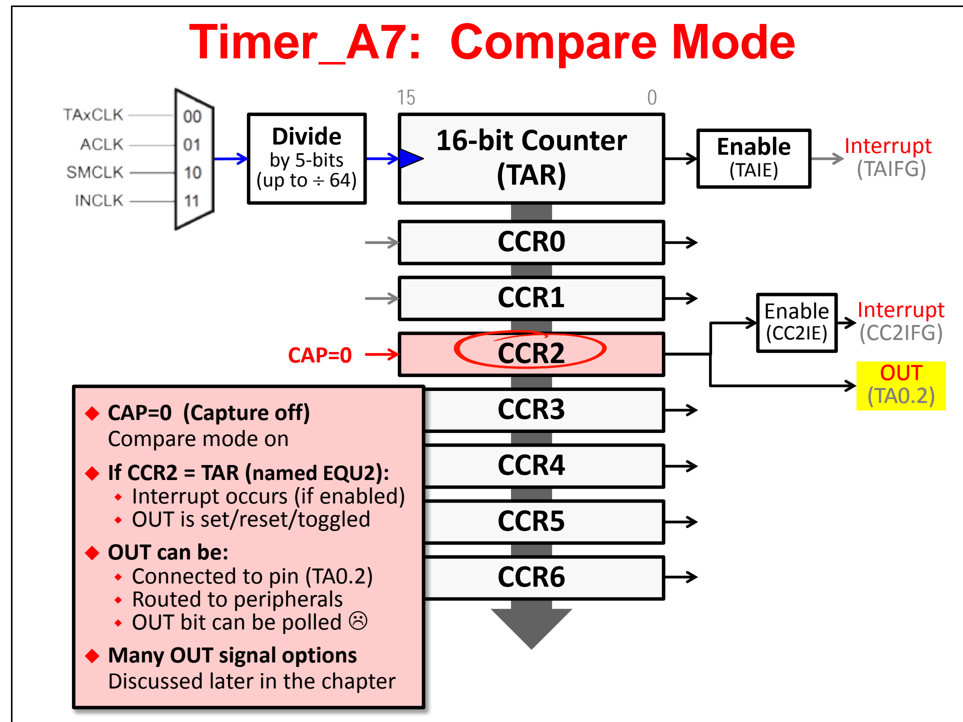
Example's Parameter Value	What is Parameter For?	Value
TIMER_A0_BASE	Which timer are you using?	TA0
TIMER_A_CAPTURECOMPARE_REGISTER_6	Which CCR is being configured?	CCR6
TIMER_A_CAPTUREMODE_RISING_EDGE	Which edge of the capture signal are you using?	Rising
TIMER_A_CAPTURE_INPUTSELECT_CCIxA	The signal used to trigger the capture	CCIn6A
TIMER_A_CAPTURE_ASYNCHRONOUS	Sync the signal to the input clock?	No, don't sync
TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE	Enable the CCR interrupt?	CC6IE = 1
TIMER_A_OUTPUTMODE_OUTBITVALUE	How should the output signal be handled?	OUTMOD=0x0

We've briefly talked about every feature (i.e. function parameter) found in this function except *OutputMode*. The "OUTBITVALUE" (for CCR6) indicates that the value of CCR6's IFG bit should be output to CCR6's Output signal. The output signal can be used by other peripherals or routed to the TA0.6 pin.

Note: With regards to OutputMode, this is just the tip-of-the-iceberg. There are actually 8 possible output mode settings. We will take you through them later in the chapter.

2b. Compare: TIMER_A_initCompare()

The other use of CCR is for *comparisons* to the main timer/counter (TAR).

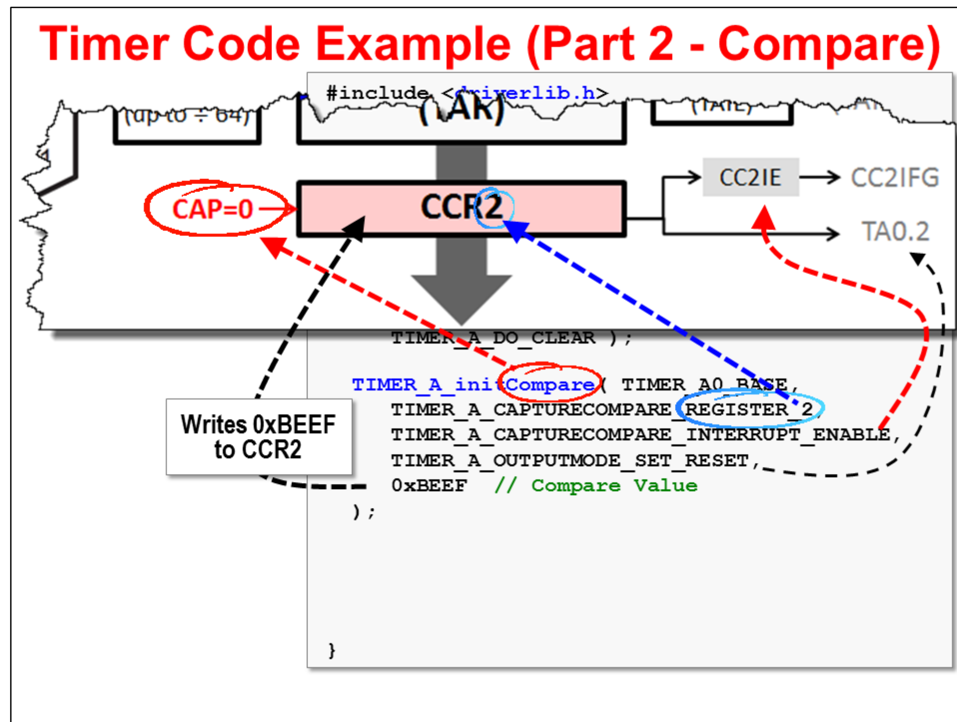


Once again, before we walk through the function that initializes CCR for Compare, let's examine its options:

- Set CAP=0 for the CCR to be configured for Compare Mode. (Opposite from earlier.)
- You must set the CCR2 register to a 16-bit value. When TAR = CCR2:
  - An internal signal called EQU2 is set.
  - If enabled, EQU2 drives the interrupt flag high (CC2IFG).
  - Similar to the Capture mode, the CCR's output signal is modified by EQU2. Again, this signal is available to other internal peripherals and/or routed to a pin (in this case, TA0.2).
  - Again, similar to the Capture mode, there are a variety of possible output modes for the OUT2 signal (which will be discussed shortly).

Compare Code Example

Let's look at the code required to setup CCR2 for use in a Compare operation.



One thing you might notice about the TIMER_A_initCompare() function is that it requires fewer parameters than the complementary initCompare function.

Example's Parameter Value	What is Parameter For?	Value
TIMER_A0_BASE	Which timer are you using?	TA0
TIMER_A_CAPTURECOMPARE_REGISTER_2	Which CCR is being configured?	CCR2
TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE	Enable the CCR interrupt?	CC2IE = 1
TIMER_A_OUTPUTMODE_SET_RESET	How should the output signal be handled?	OUTMOD=0x3
0xBEEF	What 'compare' value will be written to CCR2?	CCR2 = 0xBEEF

The OutputMode setting will be configured using the "Set/Reset" mode (which correlates to the value 0x3). Once again, with so many different output mode choices, we'll defer the full explanation of this until the next topic.

Summary of Timer Setup Code – Part 2

Here’s a summary of the timer setup code we have looked at thus far.

Timer Code Example (Part 2 - Compare)

! 1

@ 2

```
#include <driverlib.h>

void initTimerA0(void) {
  // Setup TimerA0 in Up mode with CCR2 compare
  TIMER_A_configureUpMode( TIMER_A0_BASE,
     TIMER_A_CLOCKSOURCE_SMCLK,
     TIMER_A_CLOCKSOURCE_DIVIDER_1,
     TIMER_PERIOD,
     TIMER_A_TAIE_INTERRUPT_ENABLE,
     TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE,
     TIMER_A_DO_CLEAR );

  TIMER_A_initCompare( TIMER_A0_BASE,
     TIMER_A_CAPTURECOMPARE_REGISTER_2,
     TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE,
     TIMER_A_OUTPUTMODE_SET_RESET,
     0xBEEF   // Compare Value
  );



}
```

Old API – Slide will be updated on next workshop revison

Part 1 of our code configures the timer/counter; i.e. the main element of Timer_A.

Part 2 configures the various Capture/Compare registers (CCR). Due to limited space on the slide we have only included the initCompare function for CCR2. In a real application, you might use all of the CCR registers – in which case, our `initTimerA0()` function would become a lot longer.

Before we move onto Part 3 of our timer configuration code, let’s spend a few pages explaining the 8 different output mode options available when configuring Capture/Compare Registers.

Output Modes

As you may have already seen, each CCR register has its own associated pin. For CCR1 on Timer0 this pin would be named "TA0.1". Depending upon which mode you put the CCR into; this pin can be used as an input (for Capture) or an output (for either Capture or Compare).

When the pin is used as an output, its value is determined by the OUT bit-field in its control register. The exact details for this are TA0.1 = TA0CCTL1.OUT. (Sometimes you'll just see this OUT bit abbreviated as OUT1.)

Besides routing the CCR OUT signal to a pin, it can also be used by other MSP430 peripherals. For example, on some devices the A/D converter could be triggered by the timer directly.

So, what is the value of OUT for any given CCR register?

The value of OUT is determined by the OutputMode, as we discussed earlier. (Each CCR control register has its own OUTMOD bit-field). This setting tells the OUT bit how to react as each compare or capture occurs. As previously stated, there are 8 different OutputMode choices.

For example, setting OUTMOD = 0 mean it's not changed by the timer's hardware. That is, it's under software control. You can set OUT to whatever you like by writing to it in the CCRx control register.

What happens to OUT when OUTMOD = 1 ("Set" mode)?

Timer CCR (Compare) Output Mode 01

- **Each CCR has it's own signal (e.g. TA0.1)**
  - Input for capture (CCI)
  - Output for compare (OUT)
- **Used as output, the value in register bit CCRn.OUT is routed to TA0.n**
- **Value of OUT is affected by Output Mode (CCRn.OUTMOD) as described over the next few slides**
- **If OUTMOD=0, then OUT bit (and hence the signal) is under software control**

0FFFFh
TAxCCR0
TAxCCR1
0h
EQU0 TAIFG
EQU1
EQU0 TAIFG
EQU1
EQU0 TAIFG

Output Mode (CCRn.OUTMOD)

01	Set

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

Output Mode 1

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

As we can see from the diagram above, when the timer/counter (TAR) counts up to the value in CCR1 (i.e. TAR = CCR1), then a valid comparison is true.

The enumeration for OUTMOD = 1 is called *"Set"*; whenever TAR=CCR1, then OUT will be "Set" (i.e. OUT = 1). In fact, OUT will remain = 1 until the CCR is reconfigured.

Why use "Set" mode? You might find this mode useful in creating a one-shot type of signal.

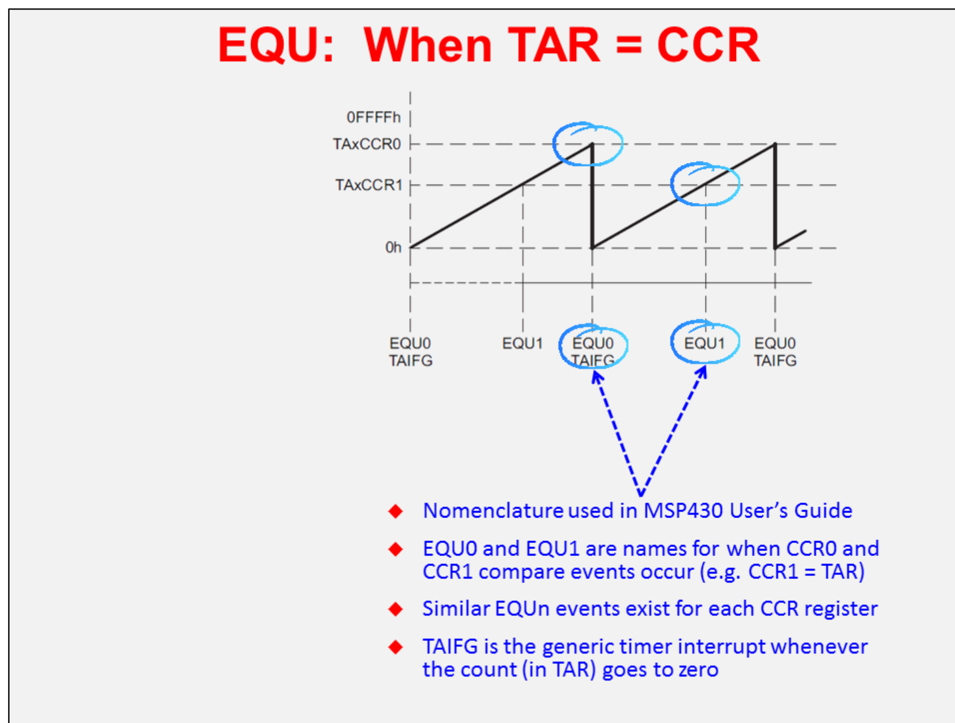
EQU

Before we examine OutputMode 2, let's consider the nomenclature used in the MSP430 User's Guide.

Apparently, there is an EQU (equate) signal inside the timer for each CCR. For example, the equate signal for CCR1 would be called EQU1. While these EQU values cannot be read directly from any of the timer control registers, the documentation makes use of them to describe when a comparison becomes true.

Therefore, when the timer counter (TAR) becomes equal to a compare register (CCR), the associated EQU signal becomes true.

This can be seen in the following diagram captured from the TIMER_A documentation. Notice how EQU0 becomes true when TAR=CCR0; likewise, EQU1 becomes true when TAR=CCR1.



OUTMOD = 2 ("Toggle/Reset" mode)

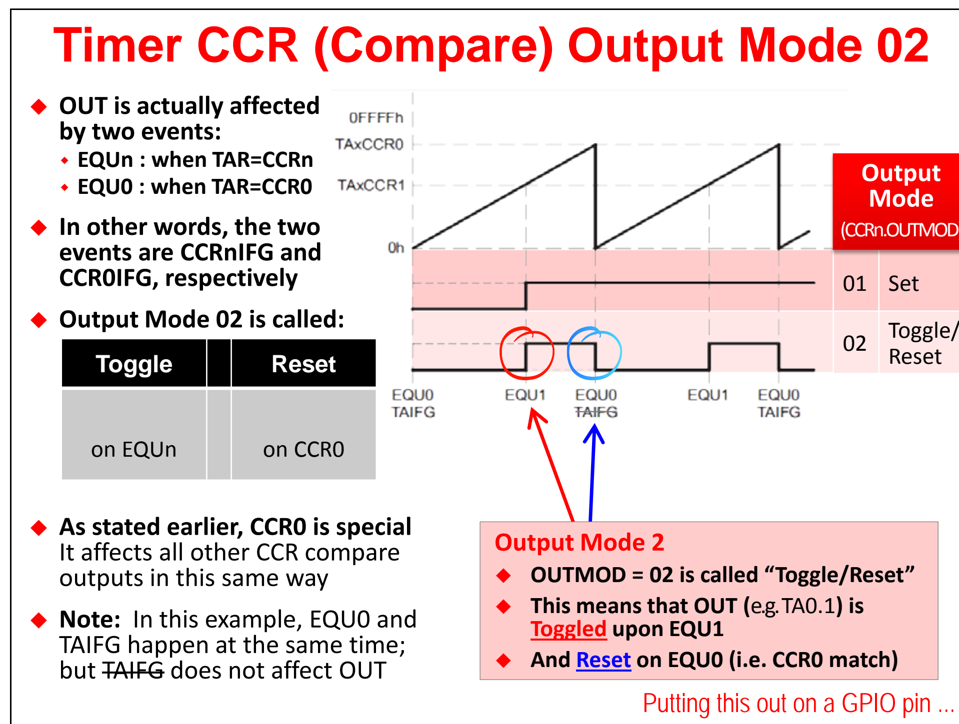
OutputMode 2 is a bit more interesting than the previous output modes. Notice how this mode is called "Toggle/Reset". Each of these names corresponds to a different event.

- Toggle - This means that OUT_n should be toggled whenever $TAR=CCR_n$
- Reset - This implies that OUT=0 (i.e. reset) whenever TAR=CCR0

In other words, when the OutputModes are defined by two names, the first one dictates the value of OUTn whenever the $TAR=CCR_n$ (i.e. whenever EQU_n becomes true). The second name describes what happens to OUT_n whenever TAR=CCR0.

Note: Remember what we said earlier, CCR0 is often used in a special way. This is another example of how CCR0 behaves differently than the rest of the CCR's.

Looking at the diagram below, we can see that in OutputMode 2, the OUT1 signal appears to be a pulse whose duty cycle (i.e. width) is proportional to the difference between CCR0 and CCR1.

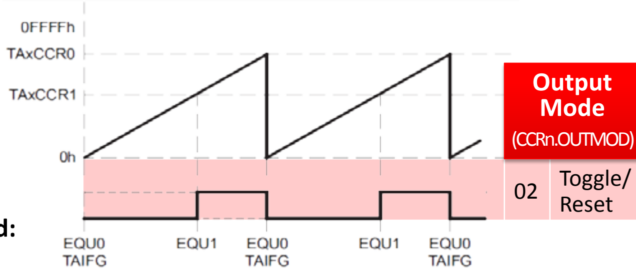


By showing both OUTMOD=1 and OUTMOD=2 in the same diagram, you can see how the value of OUT_n can be very different depending upon the OutputMode selected.

Routing the OUT signal to a pin, as shown here, lets us drive external hardware directly from the output of the timer. *(In fact, we'll use this feature to let the timer directly drive an LED during one of the upcoming lab exercises.)*

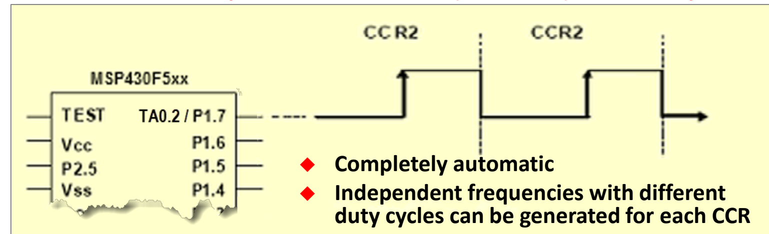
Timer CCR (Compare) Output Mode 02

- **OUT is actually affected by two events:**
  - **EQUn : when TAR=CCRn**
  - **EQU0 : when TAR=CCR0**
- **In other words, the two events are CCRnIFG and CCR0IFG, respectively**
- **Output Mode 02 is called:** **Toggle/Reset**



Output Mode (CCRn.OUTMOD)	
02	Toggle/ Reset

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

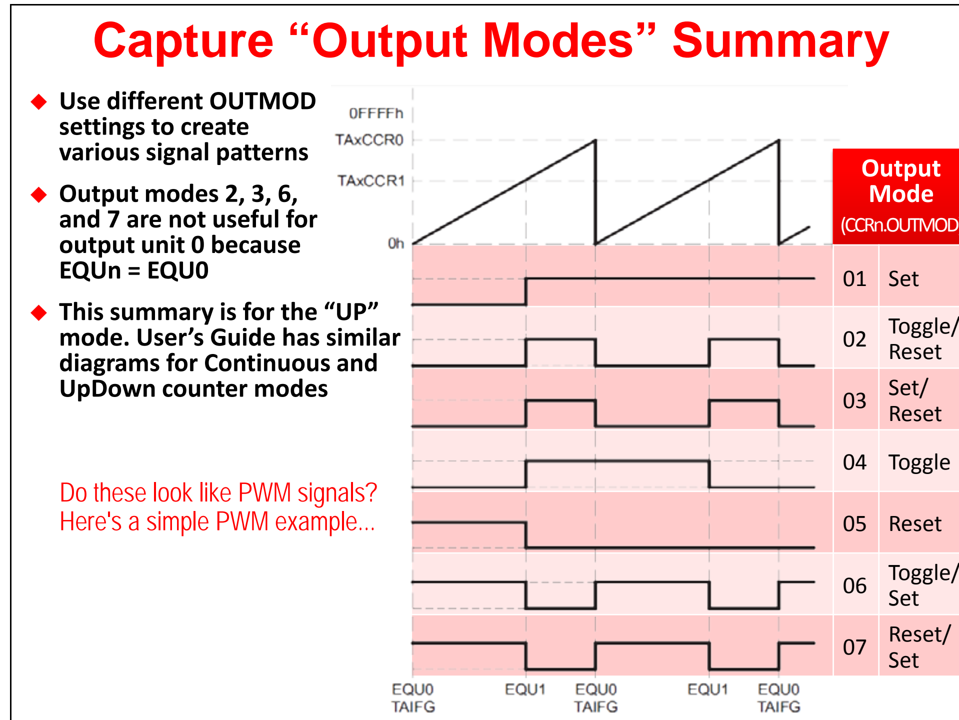


- **Completely automatic**
- **Independent frequencies with different duty cycles can be generated for each CCR**

Looking at all the Output Modes…

Summary of Output Modes

While we have only studied a couple of the output modes, we hope you will be able to decipher the remaining modes based on their names. Here is a comparison of all the different OUTput waveforms based upon the value of OUTMOD.



Point of Clarification – Only use modes 1, 4, and 5 for CCR0

The second bullet, in the diagram above, states that four of the Output Modes (2, 3, 6, and 7) are not useful when you are working with CCR0.

Why are they not useful?

All four of these OutputModes include two actions:

- One action when: CCRn=TAR
- A second action when: CCR0=TAR

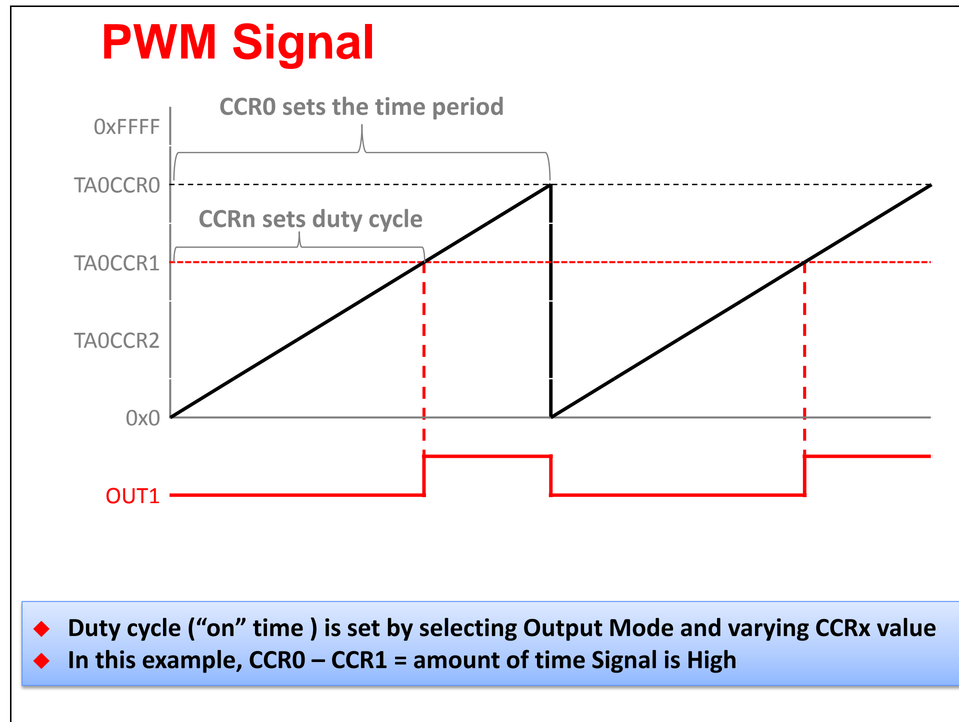
In this case, though, CCRn = CCR0. That means these modes could be trying to change OUT0 in two different ways at the same time.

Bottom Line: When using CCR0, only set OUTMOD to 0, 1, 4, or 5.

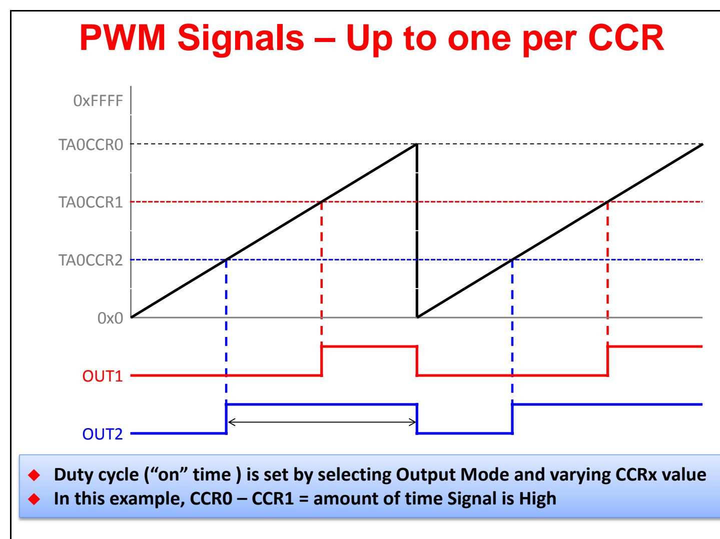
PWM anyone?

PWM, or pulse-width modulation, is commonly used to control the amount of energy going into a system. For example, by making the pulse widths longer, more energy is supplied to the system.

Looking again at the previous example where OUTMOD = 2, we can see that by changing the difference between the values of CCR0 and CCRn we can set the width of OUTn.



In the case of the MSP430, any timer can generate a PWM waveform by configuring the CCR registers appropriately. In fact, if you are using a Timer_A5, you could output 4 or 5 different PWM waveforms.



3. Clear Interrupt Flags and TIMER_A_startTimer()

Part 3 of our timer configuration code is for clearing the interrupt flags and starting the timer.

As described earlier in the workshop, you are not required to clear interrupt flags before enabling an interrupt, but once again, this is common practice. In Part 3 of the example below, we first clear the Timer flag (TA0IFG) using the function call provided by DriverLib. Then, we clear all the CCR interrupts using a single function; notice that the "+" operator tells the function that we want to clear both of these IFG bits.

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

```
#include <driverlib.h>

void initTimerA0(void) {
  // Setup TimerA0 in Up mode with CCR2 Compare
  TIMER_A_configureUpMode( TIMER_A0_BASE,
     TIMER_A_CLOCKSOURCE_SMCLK,
     TIMER_A_CLOCKSOURCE_DIVIDER_1,
     TIMER_PERIOD,
     TIMER_A_TAIE_INTERRUPT_ENABLE,
     TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE,
     TIMER_A_DO_CLEAR );

  TIMER_A_initCompare( TIMER_A0_BASE,
     TIMER_A_CAPTURECOMPARE_REGISTER_2,
     TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE,
     TIMER_A_OUTPUTMODE_SET_RESET,
     0xBEEF );    // Compare Value

  TIMER_A_clearTimerInterruptFlag(
     TIMER_A0_BASE );
  TIMER_A_clearCaptureCompareInterruptFlag(
     TIMER_A0_BASE,
     TIMER_A_CAPTURECOMPARE_REGISTER_0 +
     TIMER_A_CAPTURECOMPARE_REGISTER_2 );
  TIMER_A_startCounter( TIMER_A0_BASE,
     TIMER_A_UP_MODE );   //Make sure this
}                         //  matches config fxn
```

! 1

@ 2

3

Old API – Slide will be updated on next workshop revison

We conclude the code for Part 3 by starting the timer. The start function only has two parameters:

- It's probably obvious that you need to specify which timer that needs to be started.
- The other parameter specifies, once again, the count mode for the timer's counter.

Warning!

Did we get your attention? The timer "start" function ended up being one of the biggest problems during the development of this workshop.

As dumb as it sounds, we missed the fact that you need to set the counter mode (e.g. "UP") in this function. When we cut/pasted this function from another example, we never thought to change this parameter.

Why, because we thought it had already been specified by using the TIMER_A_configureUpMode() function. Well, we found out the hard way that you need to do both. Use the correct function AND specify the correct count mode in the *start* function.

4. Interrupt Code (Vector & ISR)

The last part of our timer code is actually a review since interrupts were covered, in detail, in a previous workshop chapter.

Timer0_A5 Interrupts Review

INT Source	IFG	IV Register	Vector Address	Loc'n
Timer A (CCIFG0)	TA0CCR0.CCIFG	none	TIMER0_A0_VECTOR	53
Timer A	TA0CCR1.IFG1…TA0CCR4.IFG	TA0IV	TIMER0_A1_VECTOR	52

TIMER0_A5
.CCIFG
.CCIE
TA0CCR0 1 1
TA0CCR1 0 0
TA0CCR2 1 1
TA0CCR3 0 0
TA0CCR4 0 0
TA0IFG 0 1
SR.GIE
CPU
53
52
TA0IV

- In the interrupts chapter, we learned that most MPS430 interrupts are grouped together and share an interrupt vector, although a few have their own dedicated vector
- Timers A and B have two vectors: one for CCR0 and the other shared
- When the CPU responds to TIMER0_A0_VECTOR, the CCR0IFG is auto cleared
- In the TIMER0_A1_VECTOR ISR, reading **TA0IV** register returns the associated highest priority pending interrupt and clears it's IFG bit

Remember, TIMER_A has two interrupt vectors: one dedicated to CCR0; another shared by TAIFG and all the other CCR's. Below, we provide a simple example of handling both.

Timer Code Example (Part 4 – ISR's)

CCR0 ISR

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

ISR for CCR2 and TA0IFG

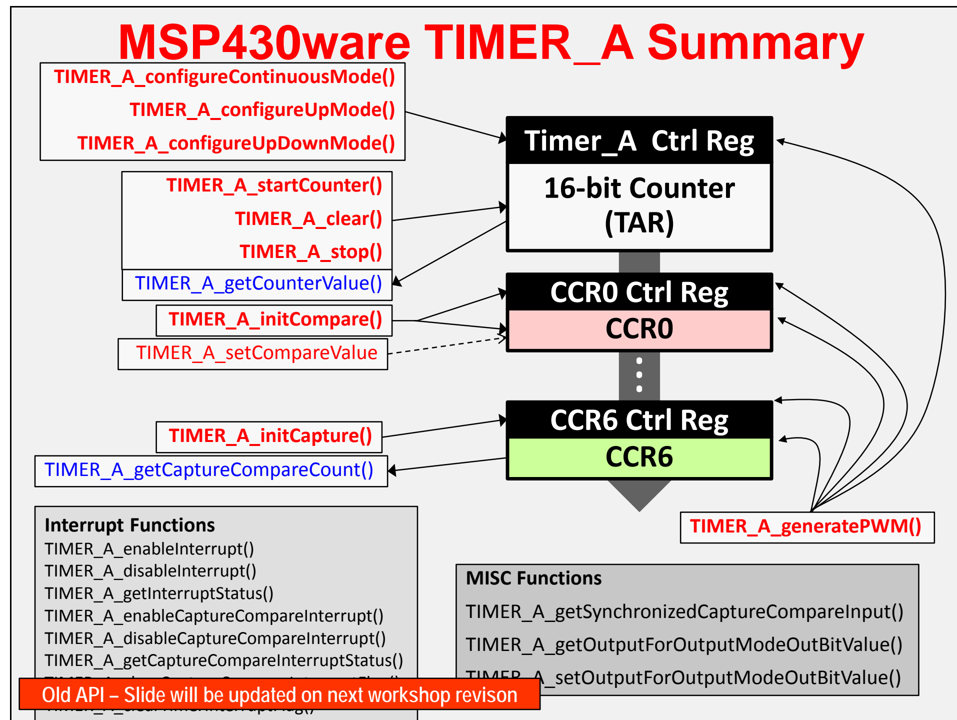
```
#pragma vector=TIMER0_A1_VECTOR
__interrupt void myISR_TA0_Other(void) {
  switch(__even_in_range( TA0IV, 10 )) {
   case 0x00: break;              // None
   case 0x02: break;              // CCR1 IFG
   case 0x04:                     // CCR2 IFG
        GPIO_toggleOutputOnPin(…);
              break;
   case 0x06: break;              // CCR3 IFG
   case 0x08: break;              // CCR4 IFG
   case 0x0A: break;              // CCR5 IFG
   case 0x0C: break;              // CCR6 IFG
   case 0x0E:                     // TA0IFG
        GPIO_toggleOutputOnPin(…);
        break;
   default:   _never_executed();
  }
}
```

$
4

TIMER_A DriverLib Summary

This diagram attempts to summarize the functions found in the TIMER_A module of the MSP430ware Driver Library.

Many of the functions have arrows pointed to/from the three main parts of the timer peripheral: TAR (the main timer/counter); CCR (used for Compare); and CCR (used for Capture). The arrows indicate whether the function reads or writes the associated registers.



The bottom of the slide contains two boxes: one summarizes the Interrupt related functions while the other contains three functions that read/write the input and output bit values.

Differences between Timer's A and B

The Timer_A and Timer_B peripherals are very similar. The following slide highlights the few differences between them.

Timer_A vs Timer_B

Similarities

- **Timer_B's default functionality is identical to Timer_A**
- **Names are (almost) the same: TAR → TBR, TA0CTL → TB0CTL, etc.**

Timer_A specific features

- **"Sampling Mode" acts like a digital sample & hold**
  - Timer_A can latch CCI input (to SCCI) upon compare
  - Makes it easy to implement software UART's
  - Timer_B cannot latch CCI directly, but most devices with Timer_B have dedicated communication peripherals

Timer_B specific features

- **Compare (CCRx) registers are double-buffered & can be updated in groups**
  - Preserves PWM "dead time" between driving complementary outputs (H-bridge)
  - More care is needed when implementing edge-aligned PWM with Timer_A
- **TBR configurable for 8, 10, 12 or 16-bits counter (default is 16-bits)**
  - Provides range of periods when used in 'Continuous' mode
- **Tri-state function from external pin**
  - External TBOUTH pins puts all Timer_B pins into high-impedance
  - With Timer_A, you would need to reconfigure pins in software

Hint: For a more complete understanding of these differences, we highly recommend that you refer to *MSP430 Microcontroller Basics*. John Davies does a great job of describing the differences between these timers. Furthermore, his discussion of generating PWM waveforms using these timers is extremely good. If you've never heard of the differences between *edge-aligned* and *centered* PWM waveforms, check out his MSP430 book.

MSP430 Microcontroller Basics by John H. Davies, (ISBN-10 0750682760) **Link↗**

Notes

Lab 6 – Using Timer_A

Lab 6 – Using Timer_A

- **Time for the lab prep Worksheet:**
  - What time is it?
  - Capture vs Compare
  - 4 steps to timer programming
  - Simple PWM generation
- **Lab 6a – Simple Timer Interrupt**
  - Create a CCR0 interrupt with the timer counting in Continuous Mode
  - ISR toggles LED
- **Optional Exercises**

  Lab 6b – Timer using Up Mode
  - Similar to Lab6a, but using Up mode

  Lab 6c – Timer with Directly Driven LED
  - Similar to Lab6b, but with the timer directly driving the LED

  Lab 6d – Simple PWM Signal
  - Alter the brightness of the LED by changing the PWM duty cycle

Time:

Worksheet – 15 mins

Labs – 30 mins

Note: The solutions exist for all of these exercises, but the instructions for Lab 6d are not yet included. These will appear in a future version of the course.

Lab Topics

Timers **6-33**

Lab 6 – Using Timer_A *6-35*

Lab 6a – Simple Timer Interrupt *6-37*
Lab 6a Worksheet 6-37
Lab 6a Procedure 6-42
Edit *myTimers.c* 6-43
Debug/Run 6-44

(Extra Credit) Lab 6b – Timer using Up Mode *6-45*
Lab 6b Worksheet 6-45
File Management 6-48
Change the Timer Setup Code 6-49
Debug/Run 6-49
Archive the Project 6-50
Timer_B (Optional) 6-51

(Extra Credit) Lab 6c – Drive GPIO Directly From Timer *6-52*
Lab 6c Abstract 6-52
Lab 6c Worksheet 6-53
File Management 6-57
Change the GPIO Setup 6-57
Change the Timer Setup Code 6-58
Debug/Run 6-59
(Optional) Lab 6c – Portable HAL 6-63

(Optional) Lab 6d – Simple PWM (Pulse Width Modulation) *6-64*

Chapter 6 Appendix *6-65*

Lab 6a – Simple Timer Interrupt

Similarly to `lab_05a_buttonInterrupt`, we want to toggle an LED based upon an interrupt. In this case, though, we'll use TIMER_A to generate an interrupt; during the interrupt service routine, we'll toggle the GPIO value that drives an LED on our Launchpad board.

As we write the ISR code, you should see that TIMER_A has two interrupts:

- One is dedicated to CCR0 (capture and compare register 0).
- The second handles all the other timer interrupts

This first TIMER_A lab will use the main timer/counter rollover interrupt (called TA0IFG). As with our previous interrupt lab (with GPIO ports), this ISR should read the TimerA0 IV register (TA0IV) and decipher the correct response using a switch/case statement.

Lab 6a Worksheet

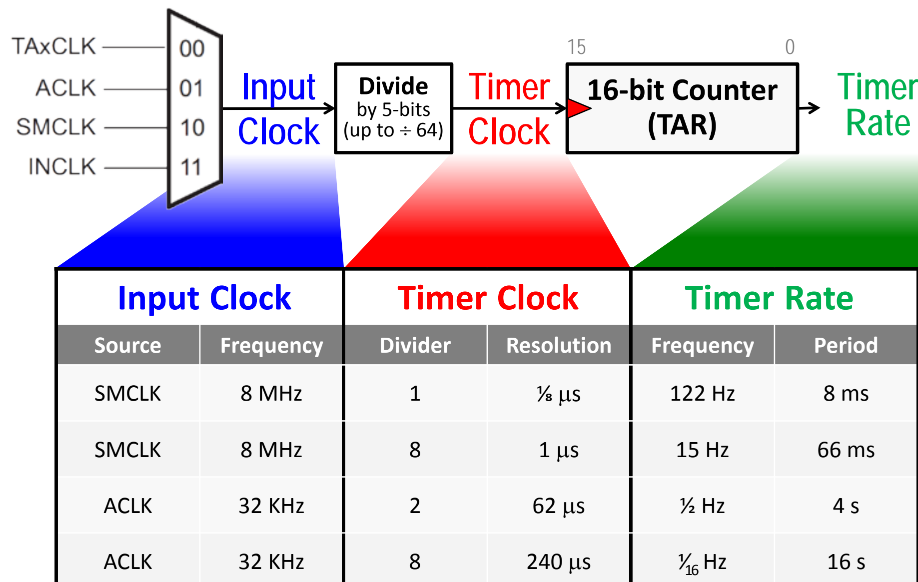
Goal: Write a function setting up Timer_A to generate an interrupt every two seconds.

1. **How many clock cycles does it take for a 16-bit counter to 'rollover'?** *(Hint: 16-bits)*

____

2. **Our goal is to generate a two second interrupt rate based on the timer clock input diagramed above.**

   Using `myClocks.c` provided for this lab, we created a table of example clock & timer rates:



Input Clock		Timer Clock		Timer Rate	
Source	Frequency	Divider	Resolution	Frequency	Period
SMCLK	8 MHz	1	⅛ µs	122 Hz	8 ms
SMCLK	8 MHz	8	1 µs	15 Hz	66 ms
ACLK	32 KHz	2	62 µs	½ Hz	4 s
ACLK	32 KHz	8	240 µs	1/16 Hz	16 s

Pick a source clock for the timer. *(Hint: At 2 seconds, a slow clock might work best.)*

Clock input (circle one): **ACLK** **SMCLK**

3. Calculate the Timer settings for the clocks & divider values needed to create a timer interrupt every 2 seconds. (That is, how can we get a timer period rate of 2 seconds.)

Which clock did you choose in the previous step? Write its frequency below and then calculate the *timer period rate*.

Input Clock: ACLK ________ running at the frequency = ________

Timer Clock = ________ ÷ ________ = ________

input clock frequency — *timer clock divider* — *timer clock freq*

Timer Rate = ________ X 65536 = ________

timer clock period (i.e. 1 / timer clock freq) — *counts for timer to rollover* — *timer rate period*

I.E. 64K

4. Which Timer do you need to use for this lab exercise?

In a later lab exercise we will output the timer directly to a BoosterPack pin. Unfortunately, the two Launchpad's map different timers to their BoosterPack pinouts. (This is due to the 'FR5969 having few pins and only using the 20-pin BoosterPack layout; versus the 40-pin XL layout for the 'F5529.)

Here are the recommended timers:

Launchpad	Timer	Short Name	Timer's Enum
'F5529	Timer0_A3	TA0	TIMER_A0_BASE
'FR4133	Timer0_A3	TA0	TIMER_A0_BASE
'FR5969	Timer1_A5	TA1	TIMER_A1_BASE

Write down the timer enumeration you need to use: **TIMER_ ________ _BASE**

5. **Write the *TIMER_A_initContinuousMode()* function.**

   The first part of our timer code is to setup the Timer control registers (TAR, TACTL). Fill in the code to specify the clock and dividers we just figured out. Also, enable the TAIE interrupt and clear the counter – but don't start the timer, yet.

```
Timer_A_initContinuousModeParam initContParam = { 0 };

  initContParam.clockSource = ___________________________________________________;

  initContParam.clockSourceDivider =  ____________________________________________;

  initContParam.timerInterruptEnable_TAIE = ______________________________________;

  initContParam.timerClear = TIMER_A_DO_CLEAR;

  initContParam.startTimer = false;

Timer_A_initContinuousMode(TIMER_ _____ _BASE, &initContParam );
```

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

6. **Skip this step … it's not required.**

   We outlined 4 steps to configure Timer_A. The second step is where you would set up the Capture and Compare features. Since this exercise doesn't need to use those features, you can skip this step.

7. **Complete the code to for the 3rd part of the "Timer Setup Code".**

   The third part of the timer setup code includes:

   – ~~Enable the interrupt (IE)~~ *… we don't have to do this, since it's done by the TIMER_A_configureContinuousMode() function (from question 5 on page* 6-*39).*

   – Clear the appropriate interrupt flag (IFG)

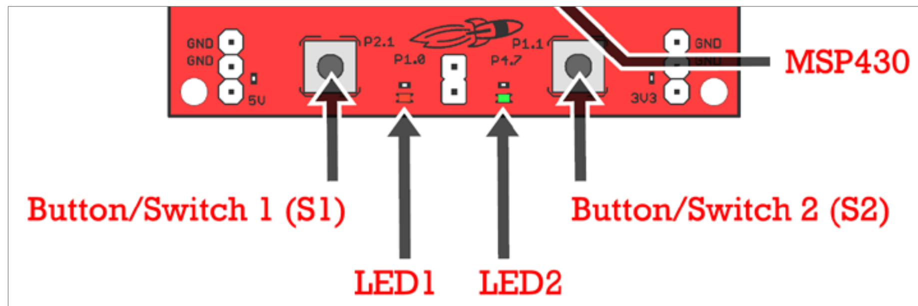
   – Start the timer

```
// Clear the timer interrupt flag

_______________________________________ ( TIMER_____BASE );    // Clear TA0IFG



// Start the timer

_______________________________________ (                       // Function to start timer

        TIMER_____BASE,                                         // Which timer?

        ______________________________                          // Run in Continuous mode
```

8. **Change the following interrupt code to toggle LED2 when Timer_A rolls-over.**

Hint:

	‘F5529 LP	‘FR5969 LP	Color
LED1 (Jumper)	P1.0	P4.6	**Red**
LED2	P4.7	P1.0	**Green**
Button 1	P2.1	P4.5	
Button 2	P1.1	P1.1	



a) **Fill in the details for your Launchpad.**

Port/Pin number for LED2: `Port _____, Pin _____`

Timer Interrupt Vector: `#pragma vector = _______________ _VECTOR`

Timer Interrupt Vector register: _______________

(Hint: We previously used `P1IV` *for GPIO Port 1)*

b) Here is the interrupt code that exists from a previous exercise, change it as needed.

Mark up the following code – crossing out what is old or not needed and writing in the modifications needed for our timer interrupt.

```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void)
{
    switch( __even_in_range(    P1IV    ,    16    )) {
            case 0: break;                      // No interrupt
            case 2: break;                      // Pin 0
            case 4:                             // Pin 1
                    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
                    break;
            case 6: break;                      // Pin 2
            case 8: break;                      // Pin 3
            case 10: break;                     // Pin 4
            case 12: break;                     // Pin 5
            case 14:
                    break;                      // Pin 6
            case 16: break;                     // Pin 7
            default:   _never_executed();
    }
}
```

Please verify your answers before moving onto the lab exercise.

Lab 6a Procedure

File Management

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

   If some are open, we recommend closing them.

2. **Import the `lab_06a_timer` project.**

   We have already created the initial lab project for you to import.

   ```
   C:\msp430_workshop\<target>\lab_06a_timer
   ```

   It doesn't matter whether you copy this project into your workspace or not. If you "copy" it into your workspace, the original files will remain untouched. If do not copy, but rather "link" to the project, you will only have one set of files and any changes you make inside of CCS will be reflected in the *C:\msp430_workshop\<target>\lab_06a_timer* directory.

3. **Briefly examine the project files**

   This project uses code we have written earlier in the workshop, though we have partitioned some of this code into separate files:

   - `myGpio.c`
     - The LED pins are configured as outputs and set to Low.
     - For the 'FR5969, the LFXT pins are set as clock inputs; and, the pins are unlocked.
   - `myClocks.c`
     - For 'F5529 users, this is the same code you wrote in the *Clocks* chapter.
     - For 'FR5969 users, we used the file from Lab4c so that ACLK uses the 32KHz crystal rather than VLO. Also, MCLK and SMCLK are set to 8MHz.

Edit *myTimers.c*

4. Edit the `myTimers.c` source file.

We want to setup the timer to generate an interrupt two seconds. The TAIFG interrupt service routine will then toggle LED2 on/off.

Worksheet Question #5 (page 6-39)

Worksheet Question #7

Worksheet Question #8

```
void initTimers(void)
{
    // 1. Setup Timer (TAR, TACTL)in Continuous mode using ACLK
    TIMER_A_ _______________________(
        TIMER_A__BASE,                          // Which timer
        TIMER_A_ ________________,              // Which clock
        TIMER_A_ _____________________,         // Clock divider
        TIMER_A_ _____________________,         // Enable INT on rollover?
        TIMER_A_DO_CLEAR                        // Clear timer counter
    );

    // 2. Setup Capture & Compare features
    //    This example does not use these features

    // 3. Clear/enable flags and start timer
    TIMER_A_ _______________________( TIMER_A1_BASE );  // Clear Timer Flag

    TIMER_A_startCounter(
        TIMER__BASE,
        TIMER_A_ _______________                // Which timer mode
    );
}

//****** Interrupt Service Routine ****************************************
#pragma vector=TIMER1_A1_VECTOR
__interrupt void timer1_ISR (void)
{
    // 4. Timer ISR and vector
    switch( __even_in_range( _____, 14 )) {     // Read timer IV register
     case  0: break;                  // None
     case  2: break;                  // CCR1 IFG
     case  4: break;                  // CCR2 IFG
     case  6: break;                  // CCR3 IFG
     case  8: break;                  // CCR4 IFG
     case 10: break;                  // CCR5 IFG
     case 12: break;                  // CCR6 IFG
     case 14:                         // TAR overflow
              // Toggle LED2 (Green) on/off
              GPIO_toggleOutputOnPin( _____________, _______________ );
              break;
     default: _never_executed();
    }
}
```

5. **Modify the *Unused Interrupts* source file.**

   Since our timer code uses an interrupt, we need to comment out its associated vector from the `unused_interrupts.c` file.

6. **Build your code and repair any errors.**

Debug/Run

7. **Launch the debugger.**

   Notice that you may still see the clock variables in the Expressions pane. This is convenient, if you want to double-check the MSP430 clock rates.

8. **Set a breakpoint inside the ISR.**

   We found it worked well to set a breakpoint on the 'switch' statement (*in the* `myTimer.c` *file*).

9. **Run your code.**

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

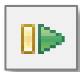
   If the breakpoint occurred, skip to the next step …

   If you did not reach the breakpoint inside your ISR, here are a few things to look for:

   – *Is the interrupt flag bit (IFG) set?*

   – *Is the interrupt enable bit (IE) set?*

   – *Are interrupts enabled globally?*



10. **If the breakpoint occurred, then resume running again.**

    You should always verify that your interrupts work by taking more than 'one' of them. *A common cause of problems occurs when the IFG bit is not cleared. This means you take one interrupt, but never get a second one.*

    In our current example, reading the TA1IV (or TA0IV for 'F5529 users) should clear the flag, so the likelihood of this problem occurring is small, but sometimes the problem still occurs due to a logical error while coding the interrupt routine.

11. **Did the LED toggle?**

    If you are executing the ISR (i.e. hitting the breakpoint) and the LED is not toggling, try single-stepping from the point where the breakpoint occurs. Make sure your program is executing the GPIO instruction.

    A common error, in this case, is accidentally putting the "do something" code (in our case, the GPIO toggle function) into the wrong 'case' statement.

12. **Once you've got the LED toggling, you can terminate your debug session.**

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

In this timer lab we switch our code from counting in the "Continuous" mode to the "Up" mode. This gives us more flexibility on the frequency of generating interrupts and output signals.

From the discussion you might remember that TIMER_A has two interrupts:

- One is dedicated to CCR0 (capture and compare register 0).
- The second handles all the other timer interrupts

In our previous lab exercise, we created an ISR for the grouped (non-dedicated) timer interrupt service routine (ISR). This lab adds an ISR for the dedicated (CCR0 based) interrupt.

Each of our two ISR's will toggle a different colored LED.

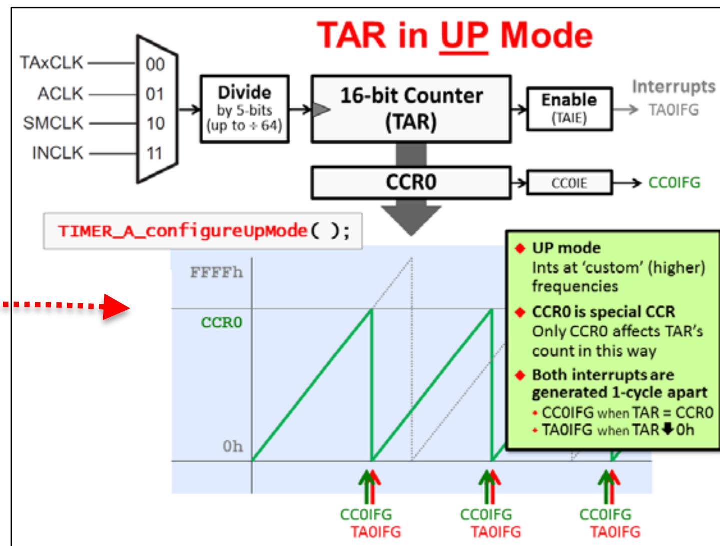
The goal of this part of the lab is to:

```
//    Timer_A in Up mode using ACLK
//    Toggle LED1 on/off every second using CCR0IFG
//    Toggle LED2 on/off every second using TA0IFG (or TA1IFG for 'FR5969)
```

Lab 6b Worksheet

1. Calculate the timer period (for CCR0) to create a 1 second interrupt rate.

Here's a quick review from our discussion.



Timer_A's counter (TAR) will count up until it reaches the value in the CCR0 capture register, then reset back to zero. What value do we need to set CCR0 to get a ½ second interval?

Timer Clock	=	32 KHz	÷	1	=	32 KHz
		input clock frequency		*timer clock divider*		*timer clock freq*
Timer Rate	=	1/32768	X		=	1 SECOND
		timer clock period (i.e. 1 / timer clock freq)		*timer counter period (i.e. CCR0 value)*		*timer rate period*

2. **Complete the *TIMER_A_initUpMode()* function?**

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

> **Hint:** Where to get help for writing this function? Once again, we recommend the MSP430ware DriverLib users guide ("docs" folder inside MPS430ware's DriverLib).
>
> Another suggestion would be to examine the `timer_a.h` header file.

```
Timer_A_initUpModeParam initUpParam = { 0 };
  initUpParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initUpParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;

  initUpParam.timerPeriod =______________________________; // (calculated in previous question)
  initUpParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initUpParam.captureCompareInterruptEnable_CCR0_CCIE =

      ________________________________________________; // Enable CCR0 compare interrupt
  initUpParam.timerClear = TIMER_A_DO_CLEAR;
  initUpParam.startTimer = false;

Timer_A_initUpMode(TIMER____BASE, &initUpParam );););
```

3. **Modifying our previous code, we need to clear both interrupts and start the timer.**

   We copied the following code from the previous exercise. It needs to be modified to meet the new objectives for this lab.

   Here are some hints:

   – Add an extra line of code to clear the CCR0 flag *(we left a blank space below for this)*

   – *Don't make the mistake we made … look very carefully at the 'startCounter' function. Is there anything that needs to change when switching from Continuous to Up mode?*

```
// Clear the timer flag and start the timer
Timer_A_clearTimerInterruptFlag( TIMER____BASE );            // Clear TA0IFG

____________________________________________ (                // Clear CCR0IFG

      TIMER______BASE,

      ______________________________________________ );

Timer_A_startCounter( TIMER____BASE,                          // Start timer in

     TIMER_A_______MODE );                                    // ____ mode
```

4. **Add a second ISR to toggle the LED1 whenever the CCR0 interrupt fires.**

On your Launchpad, what Port/Pin number does the LED1 use? ______________

Hints:

- ❑ *What port/pin does your LED1 use? Look back at question 8 (page 6-40).*
- ❑ *Look at the unused_interrupts.c file for a list of interrupt vector symbol names.*

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

```
#pragma vector= _________________________________
__interrupt void ccr0_ISR (void)
{
    // Toggle the LED1 on/off

   _____________________________________________________________________

}
```

Please verify your answers before moving onto the lab exercise.

File Management

5. Copy/Paste `lab_06a_timer` to `lab_06b_upTimer`.

a) In CCS Project Explorer, *right-click* on the `lab_06a_timer` project and select “Copy”.

b) Then, click in an open area of Project Explorer pane and select “Paste”.
This will create a new copy of your project inside the Workspace directory.

c) Finally, rename the copied project to `lab_06b_upTimer`.

Note: If you didn’t complete `lab_06a_timer` – or you just want a clean starting solution – you can import the `lab_06a_timer` archived solution.

---

6. Close the previous project: `lab_06a_timer`

7. Delete the old, readme file and import the new one.

You can import the new readme text file from this folder:

```
C:\msp430_workshop\<target>\lab_06b_upTimer
```

8. Make sure the project is selected (i.e. active) and build it to verify no errors were introduced during the copy.

Change the Timer Setup Code

In this part of Lab 6, we will be setting up TimerA in Up Mode.

9. Modify the timer configuration function, configuring it for 'Up' mode.

You should have a completed copy of this code in the Lab 6b Worksheet.

Please refer to the Lab Worksheet for assistance. (Question2, Page 6-46).

10. Modify the rest of the timer set up code, where we clear the interrupt flags, enable the individual interrupts and start the timer.

Please refer to the Lab Worksheet for assistance. (Question 3, Page 6-46).

11. Add the new ISR we wrote in the Lab Worksheet to handle the CCR0 interrupt.

When this step is complete, you should have two ISR's in your `main.c` file.

Please refer to the Lab Worksheet for assistance. (Question 4, Page 6-47).

12. Don't forget to modify the "unused" vectors (`unused_interrupts.c`).

Failing to do this will generate a build error. (Most of us saw this error back during the *Interrupts* chapter lab exercise.)

13. Build the code to verify that there are no syntax (or any other kind of) errors; fix any errors, as needed.

Debug/Run

Follow the same basic steps as found in the previous lab for debugging.

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

15. Run your code.

If all worked well, when the counter rolled over to zero, an interrupt should occur. Actually, two interrupts should occur. Once you reach the first breakpoint, resume running your code and you should reach the other ISR.

Which ISR was reached first? ____

Why? ____

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

An easy way to quickly remove all of the breakpoints is to open the Breakpoints View window:

```
View → Breakpoints
```

Then click the Remove all Breakpoints toolbar button.

Archive the Project

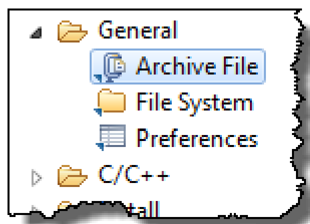
Thus far in this workshop, we have imported projects from archives … but we haven't asked you to create an archive, yet. It's not hard, as you'll see.

17. Terminate the debugger, if it's still open.

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

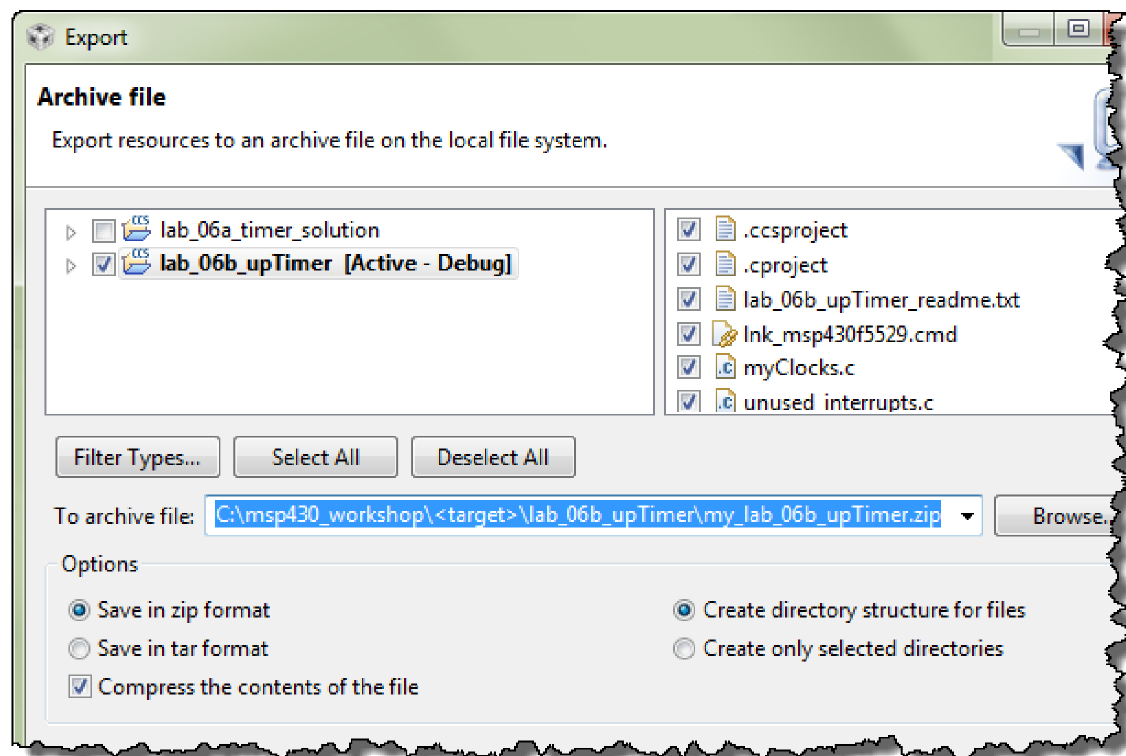
– Right-click the project and select 'Export'

– Select 'Archive File' for export, then click Next



– Fill out the dialog as shown below, choosing: the 'upTimer' lab; "Save in zip format", "Compress the contents of the file"; and the following destination:

```
C:\msp430_workshop\<target>\lab_06b_upTimer\my_lab_06b_upTimer.zip
```



Timer_B (Optional)

Note: Since the 'FR4133 does not include the Timer_B peripheral, you can skip this exercise if you're using the MSP-EXP430FR4133 Launchpad.

Do you remember during the discussion that we said Timer_A and Timer_B were very similar? In fact, the timer code we have written can be used to operate Timer_B … with 4 simple changes:

- **It's a different API … but not really.**

  Rather than using the TIMER_A module from DriverLib, you will need to use TIMER_B; unless you're using one of the few unique features of TIMER_B, the rest of the API is the same. In other words, you can carefully search and replace TIMER_A for TIMER_B.

- **Specify a different timer.**

  Since you're using a different timer, you need to specify a different timer 'base'. For either the 'F5529 or 'FR5969 you should use TIMER_B0_BASE to specify the timer instance you want to use.

- **You need to use the TIMER_B interrupt vector.**

  This changes the #pragma line where we specify the interrupt vector.

- **You need to use the TIMER_B interrupt vector register.**

  You need to read the TB0IV register to ascertain which TIMER_B flag interrupted the CPU.

All of these are simple changes. Try implementing TIMER_B on your own.

Note: While we don't provide step-by-step directions, we did create a solution file for this challenge.

(Extra Credit) Lab 6c – Drive GPIO Directly From Timer

Lab 6c Abstract

This lab is a minor adaptation of the TIMER_A code in the previous exercise. The main difference is that we'll connect the output of Timer_A CCR2 (TA0.2 or TA1.2) directly to a GPIO pin.

We are still using Up mode, which means that CCR0 is used to reset TAR back to 0. We needed to choose another signal to connect to the external pin… we arbitrarily chose to use CCR2 to generate our output signal for this exercise.

In our case, we want to drive an LED directly from the timer's output signal…

…unfortunately, the Launchpad does not have an LED connected directly to a timer output pin, therefore we'll need to use a jumper in order to make the proper connection. As we alluded to earlier in the chapter, in the case of Timer_A, the Launchpad's route different timer pins to the BoosterPack pin-outs.

Here's an excerpt from the 'F5529 lab solution:

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

And a similar statement from the 'FR5969 lab solution:

```
// When running this lab exercise, you will need to pull the J6 jumper and
// use a jumper wire to connect signal from pin ____ (on boosterpack pinouts) to
// J6.2 (bottom pin) of the LED1 jumper ... this lets the TA1.2 signal drive
// LED1 directly (without having to use interrupts)
```

And for those of you using the 'FR4133:

```
// When running this lab exercise, you will need to pull the JP1 jumper and
// use a jumper wire to connect signal from pin _____ (on boosterpack pins) to
// JP1.2 (bottom pin) of LED1 jumper ... this lets the TA1.2 signal drive
// LED1 directly (without having to use interrupts)
```

(Note: Later in the lab instructions, we'll show a picture of connecting the jumper wire.)

Lab 6c Worksheet

1. **Figure out which BoosterPack pin will be driven by the timer's output.**

   To accomplish our goal of driving the LED from a timer, we need to choose which Timer CCR register to output to a pin on the device. In the lab abstract (on the previous page) we stated that for this lab writeup, we arbitrarily chose to use CCR2.

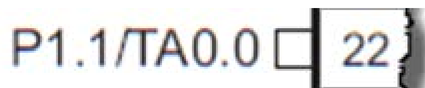
   Based on the choice of CCR2, we know that the timer's output signal will be: `TAn.2`.

   We've summarized this information in the following table:

Device	Timer	CCRx	Signal	GPIO Port/Pin	Is Pin on Boosterpack?
'F5529	TimerA0	CCR2	TA0.2		
'FR4133	TimerA1	CCR2	TA1.2		
'FR5969	TimerA1	CCR2	TA1.2		

   Your job is to fill in the remaining two columns for the device that you are using.

   a) Looking at the datasheet, which GPIO port/pin is combined with TA0.2 (or TA1.2)? For example, here we see that P1.1 is combined with TA0.0:



   Look for the correct pin in your device's datasheet and enter it in the table above.

   Hint: *There are a couple places in the datasheet to find this information. We recommend searching your device's datasheet for "TA0.2" or "TA1.2".*

   b) Next, is that signal output to the BoosterPack?

   This information can be found directly from the Launchpad. Look for the silkscreened labels next to each BoosterPack pin. When you find it, write YES/NO in the column above.

   (If you're getting a little older, you may need a magnifying glass to answer this question…or you may need to zoom in on the Launchpad's photo.)

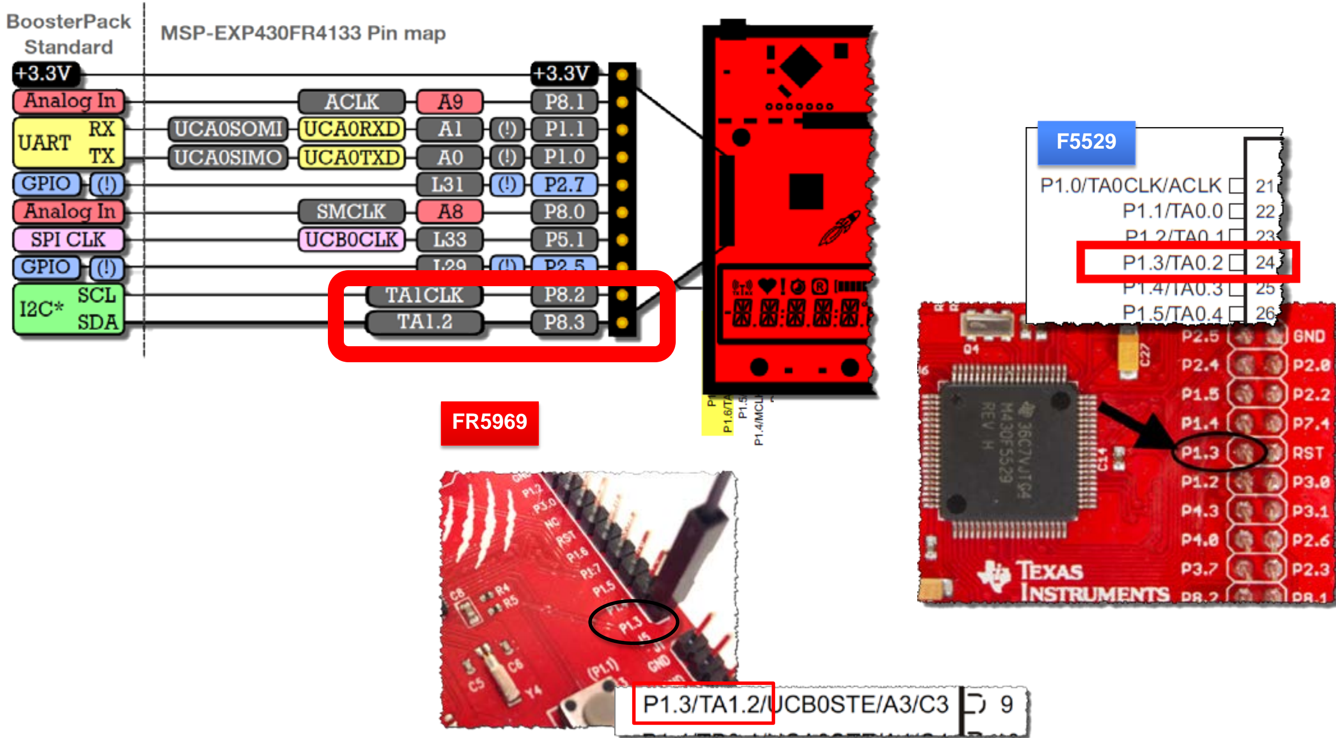
Sidebar – Choosing a Timer For This Exercise

Our choice of TimerA0 (for 'F5529) and TimerA1 (for 'FR5969 & 'FR4133) was not arbitrary. Even further, our choice of CCR2 was not entirely arbitrary.

Bottom line, we wanted to choose a Timer pin that was connected to the BoosterPack pinout since it would make it easy for us to jumper that signal over to LED1.

The problem was that neither board connected the same TimerA outputs to its Boosterpack pinout. In looking carefully at the datasheets for both devices, as well as the Boosterpack pinouts for each Launchpad, we found a timer that we could use. The only issue is that one device mapped TA0.2 to a pin, while the other mapped out TA1.2.

Did you find the correct pins on your Launchpad's BoosterPack?



2. Complete the following function to "select" P1.3 as a timer function (as opposed to GPIO)**.**

Hint: We discussed the port select function in the GPIO chapter. You can also find the details of this function in the Driver Library User's Guide.

F5529

'F5529 Users, here's the function you need to complete:

```
GPIO_setAs________________________________________(

    ________________________,

    ________________________  );
```

FR5969

FR4133

'FR5969 or 'FR4133 Users, your function requires one more argument:

```
GPIO_setA s________________________________________(

    ________________________,

    ________________________,

    __________________________________________________  );
```

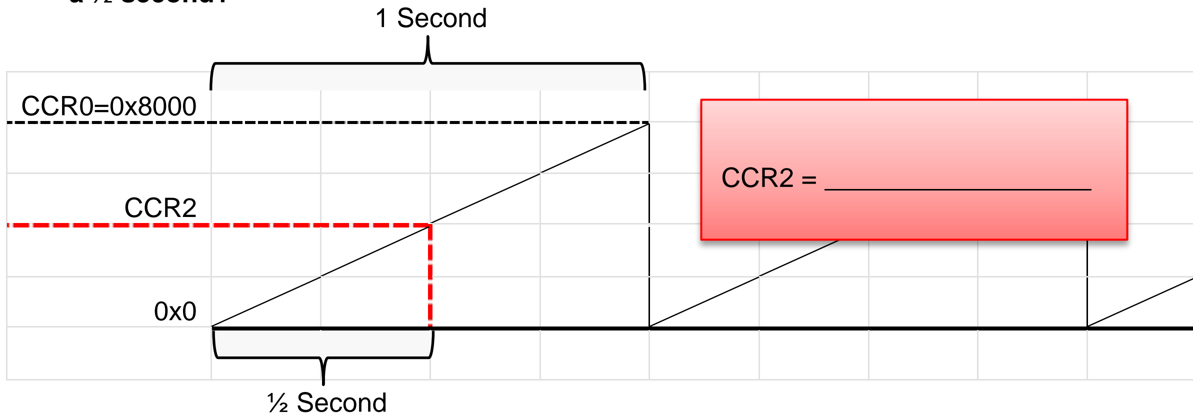
3. Modify the TIMER_A_configureUpMode() function?

Here is the code we wrote for the previous exercise. We only need to make one change to it. Since we will drive the signal directly from the timer, we don't need to generate the CCR0 interrupt anymore.

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

```
Timer_A_initUpModeParam initUpParam = { 0 };
  initUpParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initUpParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;
  initUpParam.timerPeriod = 0xFFFF / 2;
  initUpParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initUpParam.captureCompareInterruptEnable_CCR0_CCIE = TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE;
  initUpParam.timerClear = TIMER_A_DO_CLEAR;
  initUpParam.startTimer = false;
Timer_A_initUpMode( TIMER____BASE, &initUpParam );
```

4. What 'compare' value does CCR2 need to equal in order to toggle the output signal at a ½ second?



5. Add a new function call to set up Capture and Compare Register 2 (CCR2). This should be added to initTimers().

CCR2 value we calculated above goes here

```
Timer_A_init____________________  initCcr2Param = { 0 };

  initCcr2Param.compareRegister = ______________________________________________;
  initCcr2Param.compareInterruptEnable = TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE;
  initCcr2Param.compareOutputMode = TIMER_A_OUTPUTMODE_TOGGLE_RESET;

► initCcr2Param.compareValue = ________________________________________;

Timer_A_init____________________( TIMER____BASE, &initCcr2Param );
```

6. **Compare your ISR code from myTimers.c in the previous lab to the code below. What is different in the code shown here?**

   What did we change? ____

   Note, this is the 'F5529 code example. The 'FR5969 uses a slightly different interrupt vector symbol and interrupt vector register.

```
#pragma vector=TIMER0_A1_VECTOR
__interrupt void timer0_ISR(void)
{
     switch(__even_in_range( TA0IV, 14  )) {
            case 0:  break;                   // No interrupt
            case 2:  break;                   // CCR1 IFG
            case 4:                           // CCR2 IFG
                     _no_operation();
                     break;
            case 6:  break;                   // CCR3 IFG
            case 8:  break;                   // CCR4 IFG
            case 10: break;                   // CCR5 IFG
            case 12: break;                   // CCR6 IFG
            case 14: break;                   // TAR overflow
                     GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN7 );
                     break;
            default:   _never_executed();
     }
}
```

   During debug, we will ask you to set a breakpoint on 'case 4'.

   Why should case 4 not occur in our program, and thus, the breakpoint never reached?

   ____

   ____

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

   ____

   ____

   ____

   ____

File Management

1. **Copy/Paste the `lab_06b_upTimer` to `lab_06c_timerDirectDriveLed`.**

   a) In Project Explorer, right-click on the `lab_06b_upTimer` project and select “Copy”.

   b) Then, click in an open area of Project Explorer and select paste.

   c) Finally, rename the copied project to `lab_06c_timerDirectDriveLed`.

Note: If you didn’t complete `lab_06b_upTimer` – or you just want a clean starting solution – you can import the archived solution for it.

---

2. **Close the previous project: `lab_06b_upTimer`**

3. **Delete old, readme file.**

   Delete the old readme file and import the new one from:

   `C:\msp430_workshop\<target>\lab_06c_timerDirectDriveLed`

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

Change the GPIO Setup

Similar to the parts A and B of this lab, we will make the changes discussed in the lab worksheet.

5. **Modify the initGPIO function, defining the appropriate pin to be configured for the timer peripheral function.**

   Please refer to the Lab6c Worksheet for assistance. (Question 2, Page 6-54).

Change the Timer Setup Code

6. Modify the timer configuration function; we are still using 'Up' mode, but we can eliminate one of the interrupts.

Please refer to the Lab Worksheet for assistance. (Step 3, Page 6-55).

7. Add the TIMER_A function to your code that configures CCR2.

Please refer to the Lab Worksheet for assistance. (Step 5, Page 6-55).

8. Delete or comment out the call to clear the CCR0IFG flag.

We won't need this because the timer will drive the LED directly – that is, no interrupt is required where we need to manually toggle the GPIO with a function call.

```
TIMER_A_clearCaptureCompareInterruptFlag( TIMER_A0_BASE,
       TIMER_A_CAPTURECOMPARE_REGISTER_0                         //Clear CCR0IFG
);
```

Then again, it doesn't hurt anything if you leave it in the code… if so, an unused bit gets cleared.

9. Make the minor modification to the timer isr function as shown in the worksheet.

Please refer to the Lab Worksheet for assistance. (Step 6, Page 6-56).

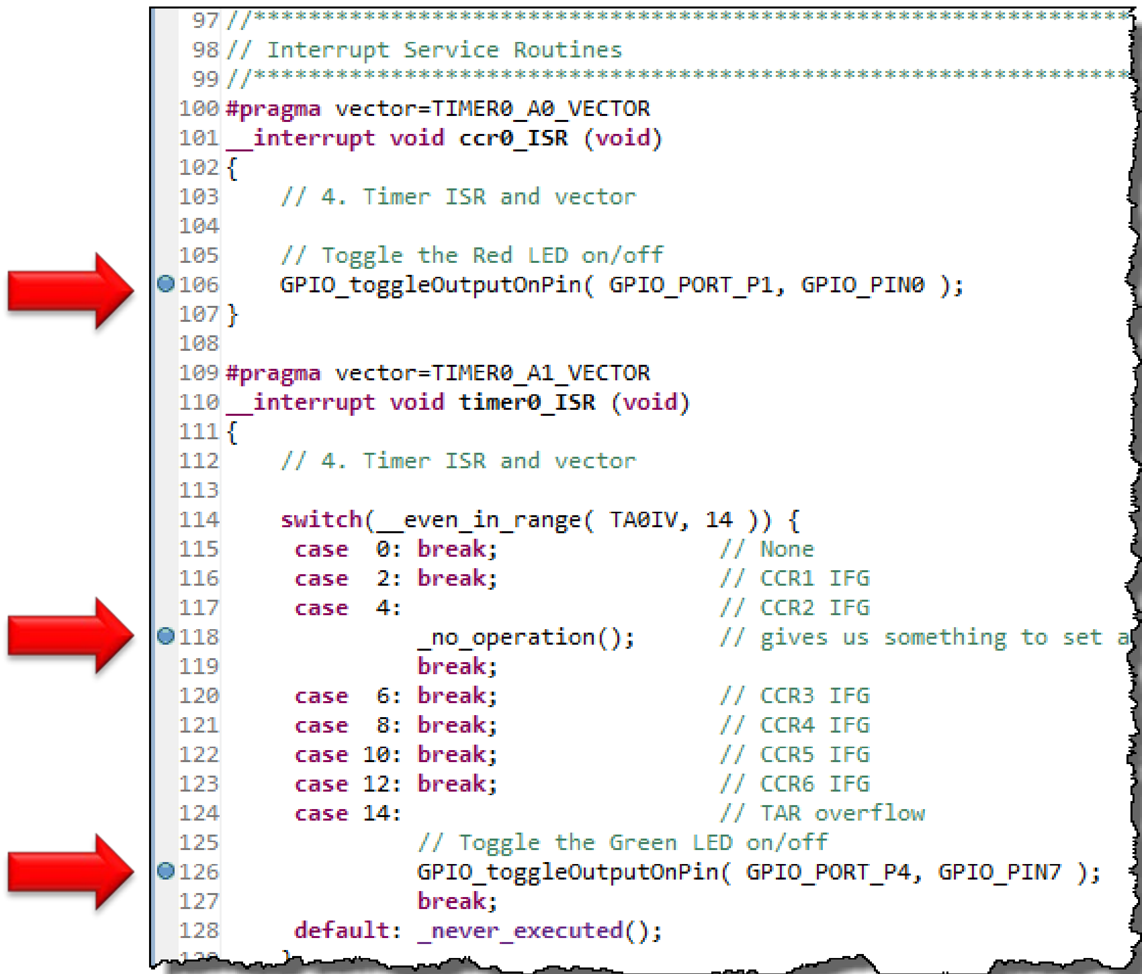
'FR5969 users – we only showed the 'F5529 code in the worksheet. Please be careful that you do not change the interrupt vector or IV register values in your code. That's not what we're asking you to do in this step.

10. Build the code verifying there are no syntax errors; fix any as needed.

Debug/Run

11. Launch the debugger and set three breakpoints inside the two ISR's.

- When we run the code, the first breakpoint will indicate if we received the CCR0 interrupt. If we wrote the code properly, we should NOT stop here.
- We should NOT stop at the second breakpoint either. CCR2 was set up to change the Output Signal, not generate an interrupt.
- We should stop at the 3rd breakpoint. We left the timer configured to break whenever TAR rolled-over to zero. (That is, whenever TA0IFG or TA1IFG gets set.)



```
 97 //*****************************************************************
 98 // Interrupt Service Routines
 99 //*****************************************************************
100 #pragma vector=TIMER0_A0_VECTOR
101 __interrupt void ccr0_ISR (void)
102 {
103     // 4. Timer ISR and vector
104
105     // Toggle the Red LED on/off
106     GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
107 }
108
109 #pragma vector=TIMER0_A1_VECTOR
110 __interrupt void timer0_ISR (void)
111 {
112     // 4. Timer ISR and vector
113
114     switch(__even_in_range( TA0IV, 14 )) {
115      case  0: break;                 // None
116      case  2: break;                 // CCR1 IFG
117      case  4:                        // CCR2 IFG
118               _no_operation();       // gives us something to set a
119               break;
120      case  6: break;                 // CCR3 IFG
121      case  8: break;                 // CCR4 IFG
122      case 10: break;                 // CCR5 IFG
123      case 12: break;                 // CCR6 IFG
124      case 14:                        // TAR overflow
125               // Toggle the Green LED on/off
126               GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN7 );
127               break;
128      default: _never_executed();
```

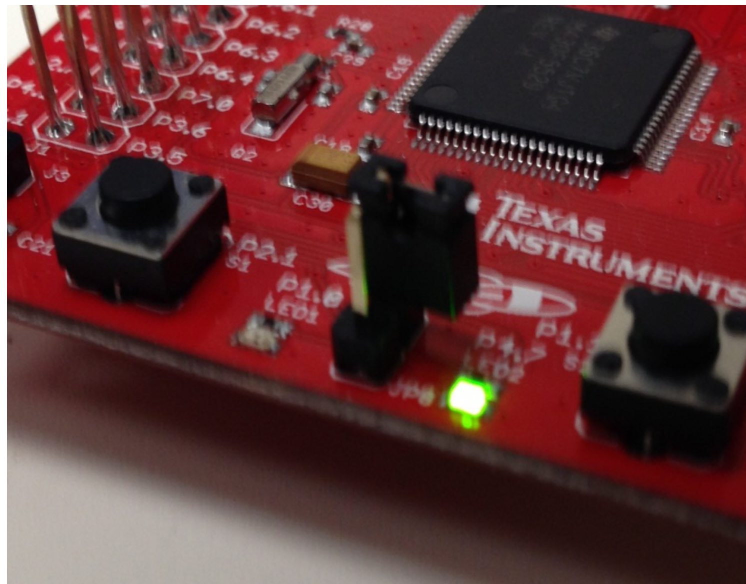
Note: As of this writing, due to an emulator bug with the 'FR5969 – as we discussed in an earlier lab exercise – terminating, restarting, or resetting the 'FR5969 with two or more breakpoints set may cause an error. If this occurs, load a different program, then reload the current one again.

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

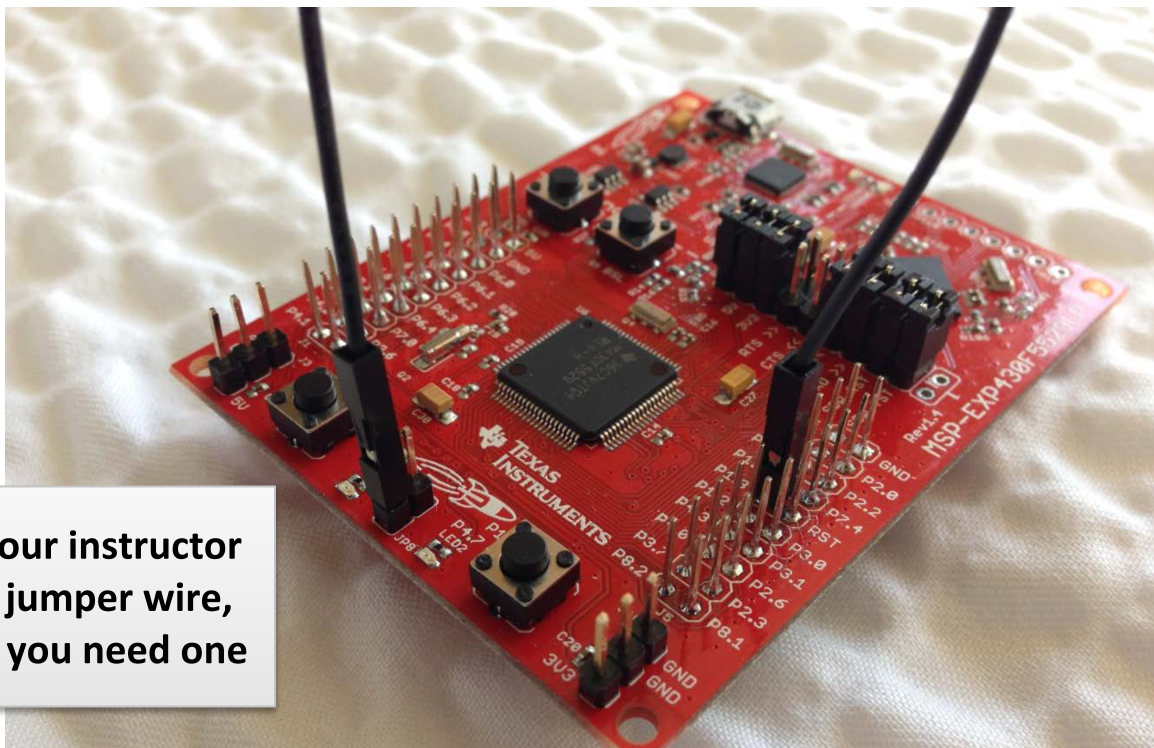
Why doesn't the LED1 toggle? ______________________________

13. **Add the jumper wire to your board to connect the timer output pin to LED1.**

    a) Remove the jumper (JP8 or J6) that connects the LED1 to P1.0 (or P4.6).
    (We recommend reconnecting it to the top pin of the jumper so that you don't lose it.)

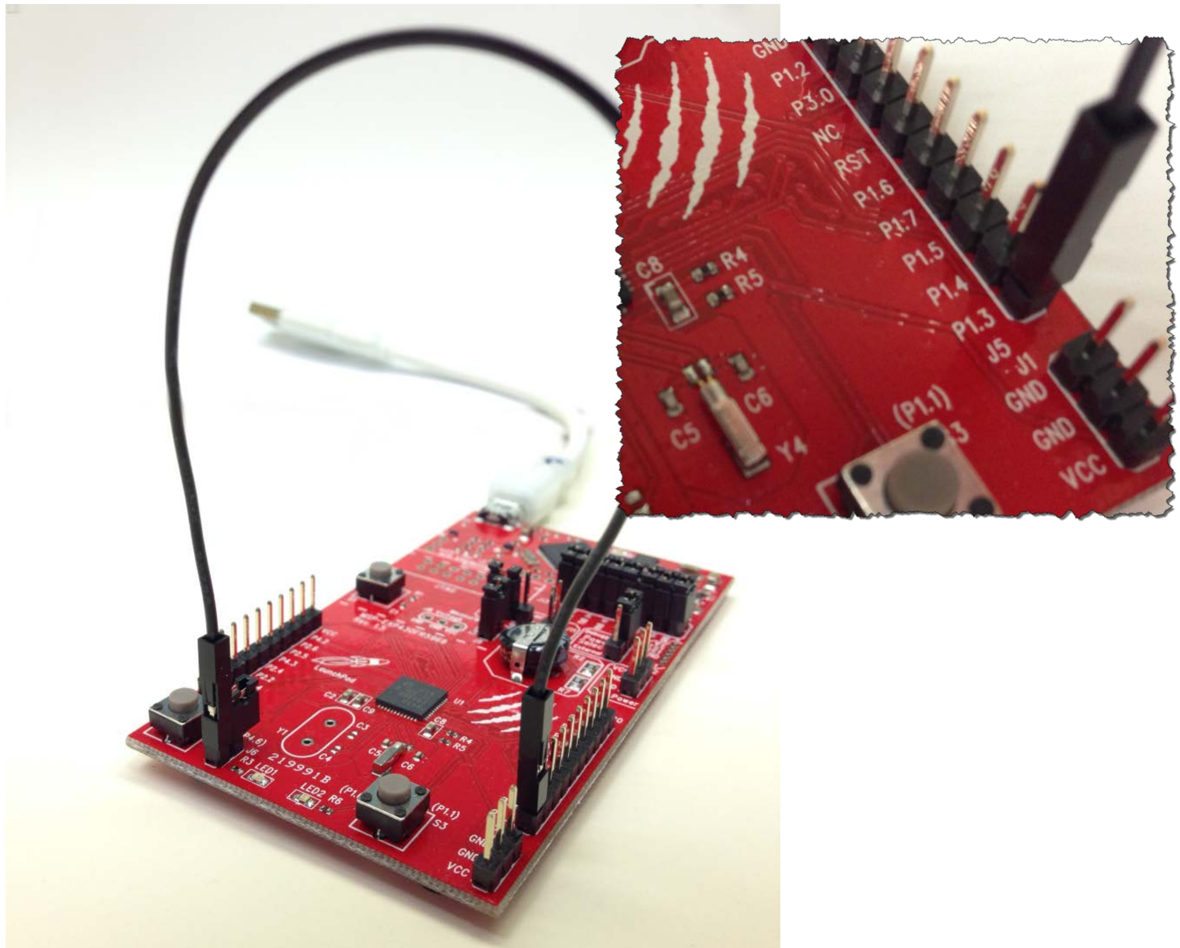


    b) On the 'F5529 Launchpad, connect P1.3 (fifth pin down, right-side of board, inside row of pins) to the bottom of the LED1 jumper (JP8) using the jumper wire.
    (See the next page for the 'FR5969 Launchpad.)



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

c) On the ‘FR5969 (not shown), connect P1.3 (in the lower, right-hand corner of the BoosterPack pins to the LED1 jumper (J6).



d) We didn’t include a picture showing the ‘FR4133 pin P8.3 being connected to LED1. It’s fairly easy to find, though as it’s in the lower-left corner of the Boosterpack pins.

14. Run your code.

Hopefully both LED’s are now blinking. LED1 should toggle first, then the LED2.

Do they both blink at the same rate? ____

Why or why not? ____

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

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

`TIMER_A_OUTPUTMODE_TOGGLE`

```
TIMER_A_initCompare( TIMER_A1_BASE,
   TIMER_A_CAPTURECOMPARE_REGISTER_2,
   TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE,
   TIMER_A_OUTPUTMODE_TOGGLE,
   0x4000
 );
```

Hint, if you haven't already tried this trick, delete the last part of the parameter and hit Ctrl_Space:

`TIMER_A_OUTPUTMODE_` then hit `Control-Space`

```
TIMER_A_initCompare( TIMER_A0_BASE,
   TIMER_A_CAPTURECOMPARE_REGISTER_2,           // User CCR2 for compa
   TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE,    // Since directly driv
   TIMER_A_OUTPUTMODE_,                      // Toggle provides a good wa
   0x4000
 );

// 3. Clear/enable fla
TIMER_A_clearTimerInte

TIMER_A_startCounter(
    TIMER_A_UP_MODE
);
```

```
# TIMER_A_OUTPUTMODE_OUTBITVALUE
# TIMER_A_OUTPUTMODE_OUTBITVALUE_HIGH
# TIMER_A_OUTPUTMODE_OUTBITVALUE_LOW
# TIMER_A_OUTPUTMODE_RESET
# TIMER_A_OUTPUTMODE_RESET_SET
# TIMER_A_OUTPUTMODE_SET
# TIMER_A_OUTPUTMODE_SET_RESET
# TIMER_A_OUTPUTMODE_TOGGLE
# TIMER_A_OUTPUTMODE_TOGGLE_RESET
# TIMER_A_OUTPUTMODE_TOGGLE_SET
Press 'Ctrl+Space' to show Template Proposals
```

Eclipse will provide the possible variations. Double-click on one (or select one and hit return) to enter it into your code.

17. Build and run your code with the new Output Mode setting.

Do they both blink at the same rate? ______________________________

If a compare match (TAR = CCR2) causes the output to be SET (i.e. LED goes ON), what causes the output to be RESET (LED going OFF)?

____

How would this differ if you used the "TIMER_A_OUTPUTMODE_**SET_RESET**" mode …

If a compare match (TAR = CCR2) causes the output to be SET (i.e. LED goes ON), what causes the RESET (LED going OFF)?

____

____

You may want to experiment with a few other output mode settings. It can be fun to see them in action.

18. When done experimenting, terminate and close the project.

(Optional) Lab 6c – Portable HAL

Can you create a single timer source file that would work on multiple platforms?

For the most part, "Yes". This is often done by creating a HAL (hardware abstraction layer). We've created a rudimentary HAL version of Lab 6c. You can find this in the solution file:

```
lab_06c_timerHal_solution.zip
```

While the timer file is shared between the two HAL solutions, we didn't get too fancy with this. There are a couple of things we didn't handle; for example, we didn't do anything with *unused_interrupts.c* and so it hade to be edited manually when porting between processors.

Play with it as you wish…

(Optional) Lab 6d – Simple PWM (Pulse Width Modulation)

While we don't have a complete write-up for our Simple PWM lab exercise, we created a solution that shows off the TIMER_A_simplePWM() DriverLib function.

The `lab_06d_simplePWM` project uses this DriverLib function to create a single PWM waveform. As with Lab 6c, the output is routed to LED1 using a jumper wire. By default, it creates a 50% duty cycle … which means it blinks the light on/off (50% on, 50% off) similar (but slightly faster) than our previous lab exercise.

One big change, though, is that we added two arguments to the initTimers() function. These values are the "Period" and "Duty Cycle" values that are passed to the simplePWM function. We also rewrote the main while{} loop so that it calls initTimers() every second.

The purpose of these changes was to allow you to have an easy way to experiment with different Period & Duty Cycle values without having to re-build and re-start the program over-and-over again. The values for period and duty-cycle were created as global variables – again, this makes it easier to change them while debugging the project.

The easiest way to experiement with this program once you've started it running is to:

- Halt (i.e. Suspend) the program
- View the two values in the Expressions watch window
- Change the values, as desired
- Continue running the program – in a second, literally, the values should take effect

By the way, if you change the period to something smaller, you won't be able to see the LED going on/off anymore – it will just appear to stay on. At this point, changing the duty cycle will cause the LED to appear bright (or dim).

As the name implies, this is a simple example, using a Driver Library function to quickly get PWM running.

Both Timer_A and Timer_B peripherals can create multiple/complex PWM (pulse-width modulation) waveforms. At some point, we may add additional PWM examples to the workshop, but if you want to learn more right now, we highly recommend that you review the excellent discussion in John Davies book: ***MSP430 Microcontroller Basics*** **by John H. Davies, (ISBN-10 0750682760) Link↗**

Chapter 6 Appendix

Lab6a Answers

Lab 6a Worksheet (1-2)

Goal: Write a function setting up Timer_A to generate an interrupt every two seconds.

1. **How many clock cycles does it take for a 16-bit counter to 'rollover'?** *(Hint: 16-bits)*

   2^{16} = 64K

2. **Our goal is to generate a two second interrupt rate based on the timer clock input diagramed above.**

   Pick a source clock for the timer. *(Hint: At 2 seconds, a slow clock might work best.)*

   Clock input (circle one): (ACLK) SMCLK

   In Lab 4c we configured ACLK for 32KHz

Lab 6a Worksheet (3)

3. **Calculate the Timer settings for the clocks & divider values needed to create a timer interrupt every 2 seconds. (That is, how can we get a timer period rate of 2 seconds.)**

   Which clock did you choose in the previous step? Write its frequency below and then calculate the *timer period rate*.

   Input Clock: ACLK running at the frequency = 32 KHz

   Timer Clock = 32KHz (*input clock frequency*) ÷ 1 (*timer clock divider*) = 32K / sec (*timer clock freq*)

   Timer Rate = 1 sec / 32K cycles (*timer clock period (i.e. 1 / timer clock freq)*) X 65536 (*counts for timer to rollover*) = 2 sec (*timer rate period*)

   I.E. 64K

Lab 6a Worksheet (4-5)

4. Which Timer do you need to use for this lab exercise?

Launchpad	Timer	Short Name	Timer's Enum
'F5529	Timer0_A3	TA0	TIMER_A0_BASE
'FR4133	Timer0_A3	TA0	TIMER_A0_BASE
'FR5969	Timer1_A5	TA1	TIMER_A1_BASE

Pick the one req'd for your board: A0 or A1

Write down the timer enumeration you need to use: **TIMER_ ________ _BASE**

5. Write the *TIMER_A_initContinuousMode()* function.

```
Timer_A_initContinuousModeParam initContParam = { 0 };
  initContParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initContParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;
  initContParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initContParam.timerClear = TIMER_A_DO_CLEAR;
  initContParam.startTimer = false;
Timer_A_initContinuousMode(TIMER_ A0/A1 _BASE, &initContParam );
```

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

Lab 6a Worksheet (7)

7. Complete the code to for the 3rd part of the "Timer Setup Code".

The third part of the timer setup code includes:

- ~~Enable the interrupt (IE)~~ … *we don't have to do this, since it's done by the TIMER_A_configureContinuousMode() function (from step5 on page 6-39).*
- Clear the appropriate interrupt flag (IFG)
- Start the timer

```
// Clear the timer interrupt flag
Timer_A_clearTimerInterruptFlag ( TIMER_ _BASE );    // Clear TA0IFG
```

AO or A1

```
// Start the timer
Timer_A_startCounter (                               // Function to start timer
    TIMER_ _BASE,                                    // Which timer?
    TIMER_A_CONTINUOUS_MODE                          // Run in Continuous mode
```

Lab 6a Worksheet (8a)

'F5529 Solution

8. Change the following interrupt code to toggle LED2 when Timer_A rolls-over.

a) Fill in the details for your Launchpad.

Port/Pin number for LED2: `Port` **4**, `Pin` **7**

Interrupt Vector: `#pragma vector =` **TIMER0_A1** `_VECTOR`

Interrupt Vector register: **TA0IV**

(for example, we used `P1IV` for GPIO Port 1)

'FR5969 Solution

8. Change the following interrupt code to toggle LED2 when Timer_A rolls-over.

a) Fill in the details for your Launchpad.

Port/Pin number for LED2: `Port` **1**, `Pin` **0**

Interrupt Vector: `#pragma vector =` **TIMER1_A1** `_VECTOR`

Interrupt Vector register: **TA1IV**

(for example, we used `P1IV` for GPIO Port 1)

Lab 6a Worksheet (8b)

b) Here is the interrupt code that exists from a previous exercise, change it as needed:

```
#pragma vector=PORT1_VECTOR          → TIMER1_A1_VECTOR
__interrupt void pushbutton_ISR (void)   → timer_ISR
{
    switch( __even_in_range( P1IV , 16 ))   → TA1IV , 14
        case 0: break;              // No
        case 2: break;              // Pin
        case 4:                     // Pin
            GPIO_toggleOutputOnPin( GPIO_P...    (struck out)
            break;
        case 6: break;              // Pin 2
        case 8: break;              // Pin 3
        case 10: break;
        case 12: break;
        case 14: GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
            break;                  // Pin 6
        case 16: break;             // Pin 7   (struck out)
        default:  _never_executed();
    }
}
```

TA0IV (for 'F5529 and 'FR4133)
TA1IV (for 'FR5969)

or for the 'F5529:
GPIO_toggleOutputOnPin(GPIO_PORT_P4, GPIO_PIN7);

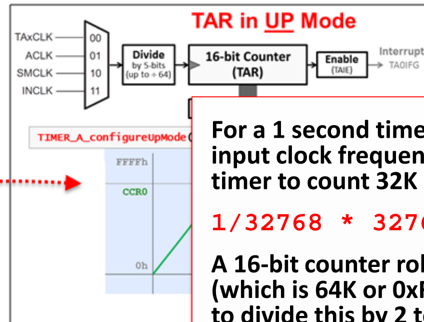
- 'FR5969 Answers are shown
- For 'FR4133, use:
  - TIMER1_A1_VECTOR
  - TA1IV
  - P4.0
- For 'F5529, use:
  - TIMER0_A1_VECTOR
  - TA0IV
  - P4.7

Lab6b Answers

Lab 6b Worksheet (1)

1. **Calculate the timer period (for CCR0) to create a 1 second interrupt rate.**

Here's a quick review from our discussion.



For a 1 second timer rate and a 32KHz input clock frequency, we need the timer to count 32K (or 32768) times:

`1/32768 * 32768 = 1 sec`

A 16-bit counter rolls over at 2^{16} counts (which is 64K or 0xFFFF). We just need to divide this by 2 to get 32K:

`Period = 0xFFFF/2 = 0x8000`

Timer_A's counter (TAR) will count up u... then reset back to zero. What value do ...

Timer Clock	=	32 KHz *input clock frequency*	÷	1 *timer clock divider*	=	32 KHz *timer clock freq*
Timer Rate	=	1/32768 *timer clock period (i.e. 1 / timer clock freq)*	x	0x8000 *timer counter period (i.e. CCR0 value)*	=	1 SECOND *timer rate period*

Lab 6b Worksheet (2)

2. **Complete the *TIMER_A_initUpMode()* function?**

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

Hint: Where to get help for writing this function? Once again, we recommend the MSP430ware DriverLib users guide ("docs" folder inside MPS430ware's DriverLib).

Another suggestion would be to examine the `timer_a.h` header file.

```
Timer_A_initUpModeParam initUpParam = { 0 };
  initUpParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initUpParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;

  initUpParam.timerPeriod = 0xFFFF / 2;  // (calculated in previous question)
  initUpParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initUpParam.captureCompareInterruptEnable_CCR0_CCIE =
     TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE;  // Enable CCR0 compare interrupt
  initUpParam.timerClear = TIMER_A_DO_CLEAR;
  initUpParam.startTimer = false;
Timer_A_initUpMode(TIMER___BASE, &initUpParam );
```

A0 or A1

Lab 6b Worksheet (3)

3. Modifying our previous code, we need to clear both interrupts and start the timer.

We copied the following code from the previous exercise. It needs to be modified to meet the new objectives for this lab.

Here are some hints:

- Add an extra line of code to clear the CCR0 flag *(we left a blank space below for this)*
- *Don't make the mistake we made … look very carefully at the 'startCounter' function. Is there anything that needs to change when switching from Continuous to Up mode?*

```
// Clear the timer flag and start the timer
Timer_A_clearTimerInterruptFlag( TIMER___BASE );           // Clear TA0IFG
Timer_A_clearCaptureCompareInterruptFlag (                  // Clear CCR0IFG
    TIMER___BASE,
    TIMER_A_CAPTURECOMPARE_REGISTER_0 );
Timer_A_startCounter( TIMER___BASE,                        // Start timer in
    TIMER_A_UP_MODE );                                      // UP mode
```

AO or A1

Lab 6b Worksheet (4)

4. Add a second ISR to toggle the LED1 whenever the CCR0 interrupt fires.

On your Launchpad, what Port/Pin number does the LED1 use? P4.6 (for 'FR5969) P1.0 (for 'F5529 & FR4133)

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

```
#pragma vector= TIMER1_A0_VECTOR   (or TIMER1_A0_VECTOR for 'F5529)
__interrupt void ccr0_ISR (void)
{
    // Toggle the LED1 on/off

    GPIO_toggleOutputOnPin( GPIO_PORT_P____, GPIO_PIN____ );

}
```

Reflects the value from above

Lab 6b : Lab Debrief

Debug/Run

Follow the same basic steps as found in the previous lab for debugging.

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

11. Run your code.

If all worked well, when the counter rolled over to zero, an interrupt should occur. Actually, two interrupts should occur. Once you reach the first breakpoint, resume running your code and you should reach the other ISR.

Which ISR was reached first? **LED1 then LED2**

Why? **Because the CCR0 interrupt occurs before the TAIFG interrupt**

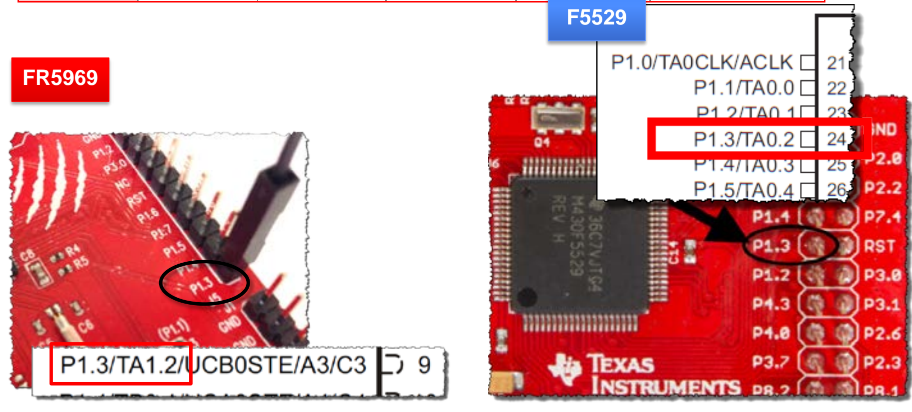
This is shown on the slide entitled "TAR in UP Mode". Since they occur at nearly the same instant in time, you have to set breakpoints in order to see that LED1 happens before LED2.

Lab6c Answers

Lab 6c Worksheet (1)

1. Figure out which BoosterPack pin will be driven by the timer's output.

Device	Timer	CCRx	Signal	GPIO Port/Pin	Is Pin on Boosterpack?
'F5529	TimerA0	CCR2	TA0.2	P1.3	Yes
'FR4133	TimerA1	CCR2	TA1.2	P8.3	Yes
'FR5969	TimerA1	CCR2	TA1.2	P1.3	Yes



Lab 6c Worksheet (2)

2. Complete the following function to "select" P1.3 as a timer function

F5529

'F5529 Users, here's the function you need to complete:

```
GPIO_setAs PeripheralModuleFunctionOutputPin (
    GPIO_PORT_P1 ,
    GPIO_PIN3 );
```

FR5969

'FR5969 Users, your function requires one more argument:

```
GPIO_setA s PeripheralModuleFunctionOutputPin (
    GPIO_PORT_P1 ,
    GPIO_PIN3 ,
    GPIO_PRIMARY_MODULE_FUNCTION );
```

FR4133

'FR4133 Users, your function requires one more argument:

```
GPIO_setA s PeripheralModuleFunctionOutputPin (
    GPIO_PORT_P1 ,
    GPIO_PIN3 ,
    GPIO_PRIMARY_MODULE_FUNCTION );
```

Lab 6c Worksheet (3)

3. Modify the TIMER_A_configureUpMode() function?

Here is the code we wrote for the previous lab exercise. We only need to make one change to the code. Since we will drive the signal directly from the timer, we don't need to generate the CCR0 interrupt anymore.

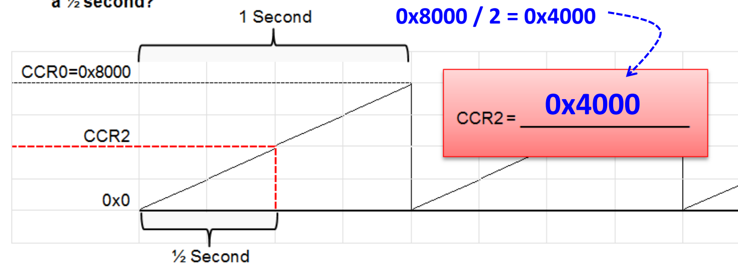
Mark up the code below to disable the interrupt. *(We'll bet you can make this change without even looking at the API documentation. Intuitive code is one of the benefits of using DriverLib!)*

```
Timer_A_initUpModeParam initUpParam = { 0 };
  initUpParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initUpParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;
  initUpParam.timerPeriod = 0xFFFF / 2;
  initUpParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initUpParam.captureCompareInterruptEnable_CCR0_CCIE = TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE;
  initUpParam.timerClear = TIMER_A_DO_CLEAR;
  initUpParam.startTimer = false;
Timer_A_initUpMode( TIMER___BASE, &initUpParam );
```

We changed 'ENABLE' to 'DISABLE'

Lab 6c Worksheet (4-5)

4. What 'compare' value does CCR2 need to equal in order to toggle the output signal at a ½ second?



5. Add a new function call to set up Capture and Compare Register 2 (CCR2). This should be added to initTimers().

CCR2 value we calculated above goes here

```
Timer_A_initCompareModeParam initCcr2Param = { 0 };
  initCcr2Param.compareRegister = TIMER_A_CAPTURECOMPARE_REGISTER_2;
  initCcr2Param.compareInterruptEnable = TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE;
  initCcr2Param.compareOutputMode = TIMER_A_OUTPUTMODE_TOGGLE_RESET;
  initCcr2Param.compareValue = 0x4000;
Timer_A_initCompareMode( TIMER___BASE, &initCcr2Param );
```

Lab 6c Worksheet (6)

6. Compare your previous code to that below.

What did we change? Added _no_operation() – something to breakpoint on

```
#pragma vector=TIMER0_A1_VECTOR
__interrupt void timer0_ISR(void)
{
    switch(__even_in_range( TA0IV, 14  )) {
        case 0:  break;                  // No interrupt
        case 2:  break;                  // CCR1 IFG
        case 4:                          // CCR2 IFG
                 _no_operation();
                 break;
        case 6:  break;                  // CCR3 IFG
        case 8:  break;                  // CCR4 IFG
        case 10: break;                  // CCR5 IFG
        case 12: break;                  // CCR6 IFG
        case 14: break;                  // TAR overflow
        def...
    }
}
```

During debug, we will ask you to set a breakpoint on 'case 4'.

Why should case 4 not occur, and thus, the breakpoint never reached?

`TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE,`

We disabled the INT because we're driving the signal directly to the pin

Lab 6c Worksheet (7)

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

- **Lower Power:**
  When the Timer drives the pin; no need to wake up the CPU. (Either that, or it leaves the CPU free for other processing.)

- **Less Latency:**
  When the CPU toggles the pin, there is a slight delay that occurs since the CPU must be interrupted, then go run the ISR.

- **More Deterministic:**
  The delay caused by generating/responding to the interrupt may vary slightly. This could be due to another interrupt being processed (or a higher priority interrupt occurring simultaneously). Directly driving the output removes the variance and makes it easy to "determine" the time that the output will change!

Lab 6c Debrief

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

Why doesn't the LED1 toggle? **We removed the interrupt that caused us to run the GPIO toggle function and replaced it with code to let the timer directly drive the LED … but we haven't hooked up the LED, yet.**

14. Run your code.

Hopefully both LED's are now blinking. LED1 should toggle first, then the LED2.

Do they both blink at the same rate? **No**

Why or why not? **LED2 is based on the timer counting up to the value in CCR0 (0x8000); while LED1 toggles when the counter reaches CCR2 (set to 0x4000) and is reset whenever the counter reaches CCR0.**

Lab 6c Debrief

17. Build and run your code with the new OutputMode setting.

Do they both blink at the same rate? **Yes (although offset by ½ second)**

If a compare match (TAR = CCR2) causes the output to be SET (i.e. LED goes ON), what causes the output to be RESET (LED going OFF)?

The next time TAR equals CCR2

How would this differ if you used the "TIMER_A_OUTPUTMODE_**SET_RESET**" mode …

If a compare match (TAR = CCR2) causes the output to be SET (i.e. LED goes ON), what causes the RESET (LED going OFF)?

In this case, the "RESET" occurs when TAR = CCR0