

F2833x PWM, Capture and QEP

Introduction

Today's electronic systems are described using terms such as "direct digital control", "digital power supply", "digital power converters" and so on. A core feature of all these applications is the ability to generate different series of digital pulse patterns to control power electronic switches based on the results of sophisticated numerical calculations. The F283xx family provides such hardware units; several pulse width modulation (PWM) output signals, along with time measurements units ("Capture Units").

In Chapter 6 we have already implemented a time base unit, using the CPU core timers 0 to 2. Although these units are also hardware based time units, they are only able to 'signal' the end of a pre-defined period. On such an event, an interrupt service routine could be requested to start and perform desired activities by a software sequence. While this scenario is sufficient for most time-based software activities, it is not suitable for hardware related actions, such as switching the control line of an output stage from passive to active. In this case we need much more precise and automatic response to the actuator control lines, based on different events on the timeline. This is where PWM - lines come into the play.

The main applications of PWM are:

- Digital Motor Control (DMC)
- Control of switching pulses for Digital Power Supply (DPS) systems
- Analogue Voltage Generators

Later we will discuss these main application areas in more detail. The F2833x is equipped with different and independent numbers of PWM channels; a F28335, for example, has 6 PWMs.

The F2833x is also able to perform time measurements using hardware signals. With the help of independent edge detector state machines, called 'Capture Units' we can measure the time difference between edges to determine the speed of a rotating shaft in revolutions per minute or the active duty cycle of a feedback signal.

A third hardware part of the Control System is called a 'Quadrature Encoder Pulse' -unit (QEP). This is a unit that is used to derive the speed and direction information of a rotating shaft directly from hardware signals from incremental encoders or resolvers.

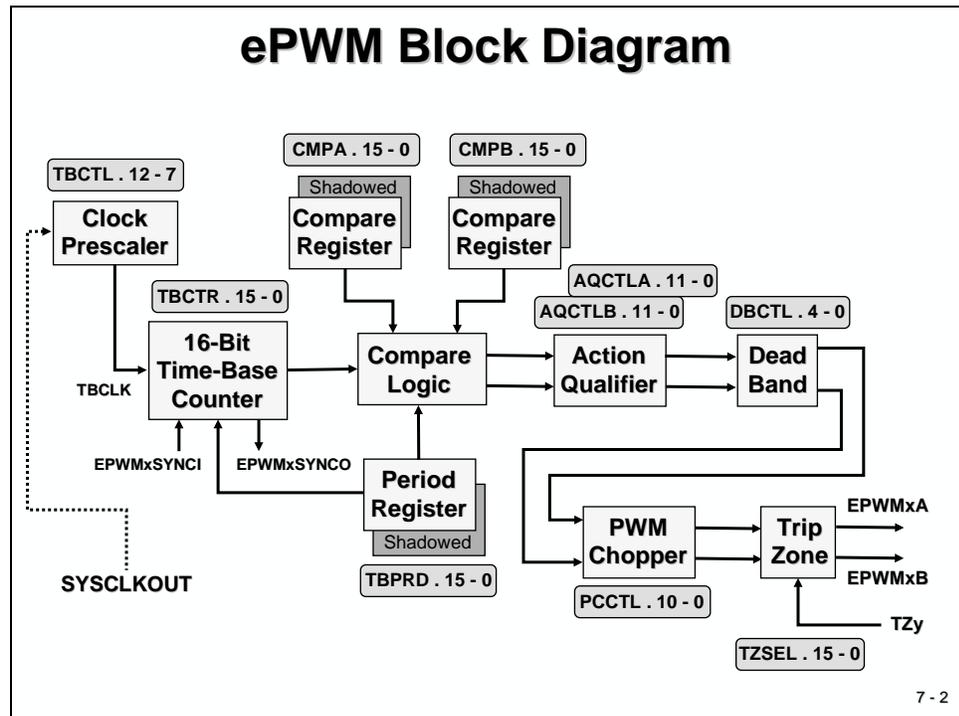
Our lab series Lab 7-1 to Lab 7-9 will include the most important operating modes of a PWM signal. A typical requirement in control loop calculations is the operation using complex numbers, which are translated according to Euler's law into sine and cosine components. Instead of calculating a new sine-value each time we need one, we can access a look-up table, which is already available inside the F283xx! This is exactly what we will do in Lab 7- 9 ("Generate a pulse width modulated sine wave signal") to implement a practical example.

Module Topics

F2833x PWM, Capture and QEP	7-1
<i>Introduction.....</i>	7-1
<i>Module Topics.....</i>	7-2
<i>ePWM Block Diagram</i>	7-3
<i>ePWM Time Base Unit.....</i>	7-4
<i>ePWM Phase Synchronisation</i>	7-5
<i>Timer Operating Modes</i>	7-6
<i>Time Base Registers</i>	7-7
<i>Lab 7_1: Generate an ePWM signal.....</i>	7-11
<i>Lab 7_2: Generate a 3 - phase signal system.....</i>	7-16
<i>Purpose of Pulse Width Modulation</i>	7-19
<i>ePWM Compare Unit.....</i>	7-21
<i>ePWM Action Qualifier Unit.....</i>	7-24
<i>Lab 7_3: A 1 kHz with variable pulse width</i>	7-30
<i>Lab 7_4: a pair of complementary 1 kHz-Signals.....</i>	7-32
<i>Lab 7_5: Independent Modulation on ePWM1A / 1B.....</i>	7-34
<i>ePWM Dead Band Module.....</i>	7-38
<i>Lab 7_6: Dead Band Unit on ePWM1A / 1B</i>	7-43
<i>ePWM Chopper Module.....</i>	7-46
<i>Lab 7_7: Chopped Signals at ePWM1A / 1B</i>	7-50
<i>ePWM Over Current Protection.....</i>	7-52
<i>Lab 7_8: Trip Zone protection with TZ6.....</i>	7-56
<i>ePWM Interrupt Sources.....</i>	7-61
<i>Lab7_9: ePWM Sine Wave Modulation</i>	7-65
<i>eCAP Capture Module</i>	7-71
<i>Capture Units Registers</i>	7-74
<i>Lab7_10: ePWM1A 1 kHz captured by eCAP1.....</i>	7-79
<i>Enhanced QEP module</i>	7-82
<i>Infrared Remote Control.....</i>	7-84
<i>Lab7_11: eCAP4 to receive a RC5 IR-signal.....</i>	7-87

ePWM Block Diagram

Each enhanced Pulse Width Modulation (ePWM) unit is controlled by its own logic block, as shown in Slide 7_2 below. This logic is able both to automatically generate signals on different time events and also to request various interrupt services from the F2833x PIE interrupt system, to support its operational modes.



A unique feature of an ePWM - module is its ability to start the Analogue to Digital Converter (ADC) without software interaction, directly from an internal hardware event. A common microcontroller would have to request an interrupt service to do the same - the F2833x does this automatically. We will use this feature in the next module!

Note: There are two basic operating modes of the ePWM system: (1) standard ePWM 16-bit mode and (2) 24-bit High Resolution PWM mode (HRPWM). For now we will discuss the 16-bit mode.

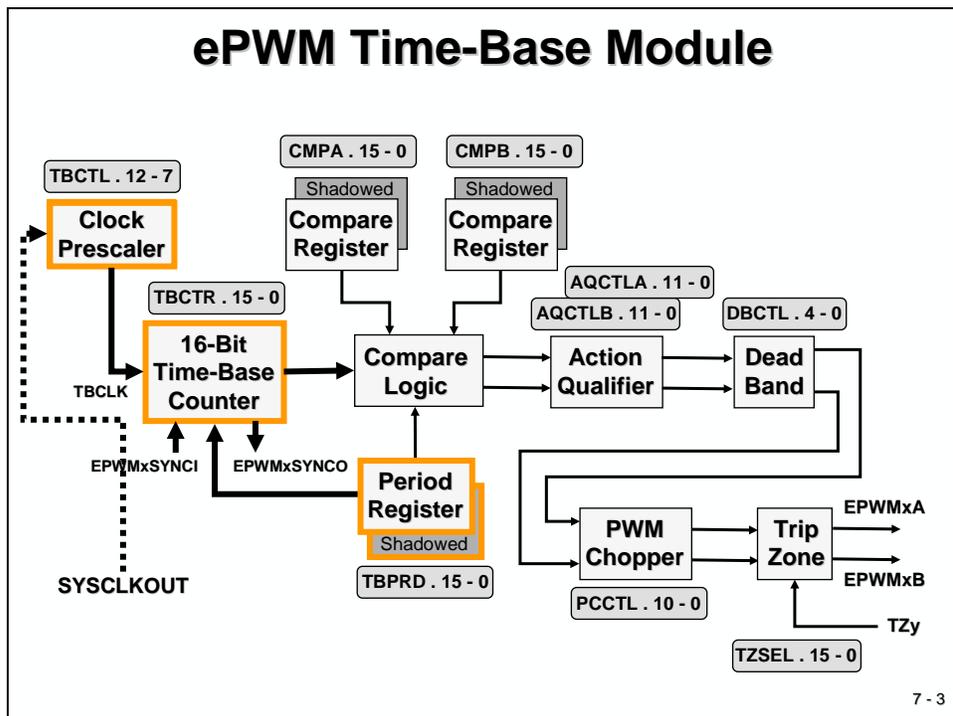
The purpose of an ePWM unit is to generate a single ended signal or a pair of output signals, called EPWMxA and EPWMxB, which are related to each other. The lower case letter x is a placeholder for the number of the ePWM unit, e.g. 1...6.

Note: to generate a physical output signal on the F2833x we have to set the multiplex registers for the I/O ports accordingly - please refer to Chapter 5!

As you can see from Slide 7-2, to generate a physical output signal we will have to setup a few units: time base, compare logic, action qualifier, dead band unit, chopper and trip zone. On first glance this looks cumbersome. However, it does allow us to setup a range of different operating modes, all of which can be used in modern digital control. So, let us make use of it!

ePWM Time Base Unit

The central block of an ePWM unit is a 16-bit timer (register "TBCTR"), with signal SYSCLKOUT as its time-base. In Chapter 5 we initialized the core to run at 100 MHz or 150 MHz, depending on the external clock of the F2833x. This frequency sets the time-base for all ePWM units.



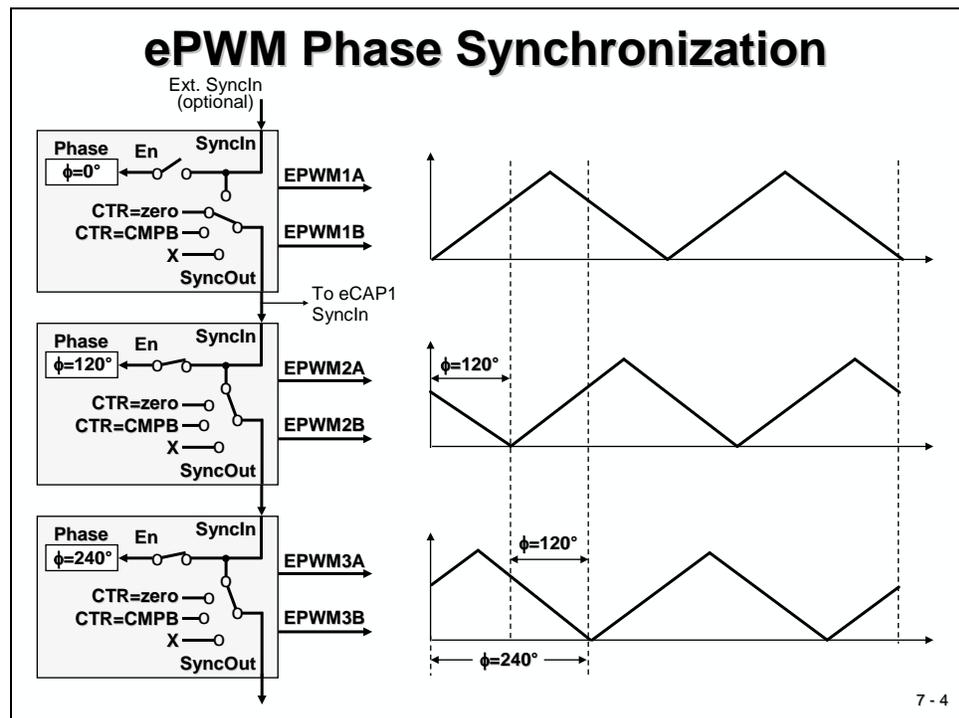
A clock prescaler (register TBCTL, bits 12 to 7) can be used to reduce the input counting frequency by a selectable factor between 1 and 1792.

Register TBPRD defines the length of a period of an output signal, in multiples of the time-period of the input signal.

Another unique feature of the F2833x is its “shadow” functionality of operating registers, in the case of ePWM units available for compare register A, B and period register. For some applications it is necessary to modify the values inside a compare or period register, every period. The advantage of the background registers is that we can prepare the values for the next period in the current one. Without a background function we would have to wait for the end of the current period, and then trigger a high prioritized interrupt. Sometimes this form of scheduling will miss its deadline...

ePWM Phase Synchronisation

Two hardware signals "SYNCI" (synch in) and "SYNCO" (synch out) can be used to synchronize ePWM units to each other. For example, we could define one ePWM unit as a "master" to generate an output signal "SYNCO" each time the counter equals period. Two more ePWM units could be initialized to recognize this signal as "SYNCI" and start immediately counting, each time they receive this signal. In such way we have established a synchronous set of 3 ePWM channels. But we can do even better. By using another register called "TBPHS" we can introduce a phase shift between master, slave 1 and slave 2, an absolute necessity for three-phase control systems.



Slide 7-4 shows such an example, where register TBCNT of ePWM2 and ePWM3 are preloaded with a start value that corresponds to 120° and 240° respectively. In this example ePWM1 has been initialized as master to generate SYNCO each time the counter register equals zero. With the enabled phase input feature for ePWM2 and ePWM3 the two channels operate as slave 1 and slave 2 and will load their counter registers TBCNT with numbers stored in the corresponding phase registers TBPHS.

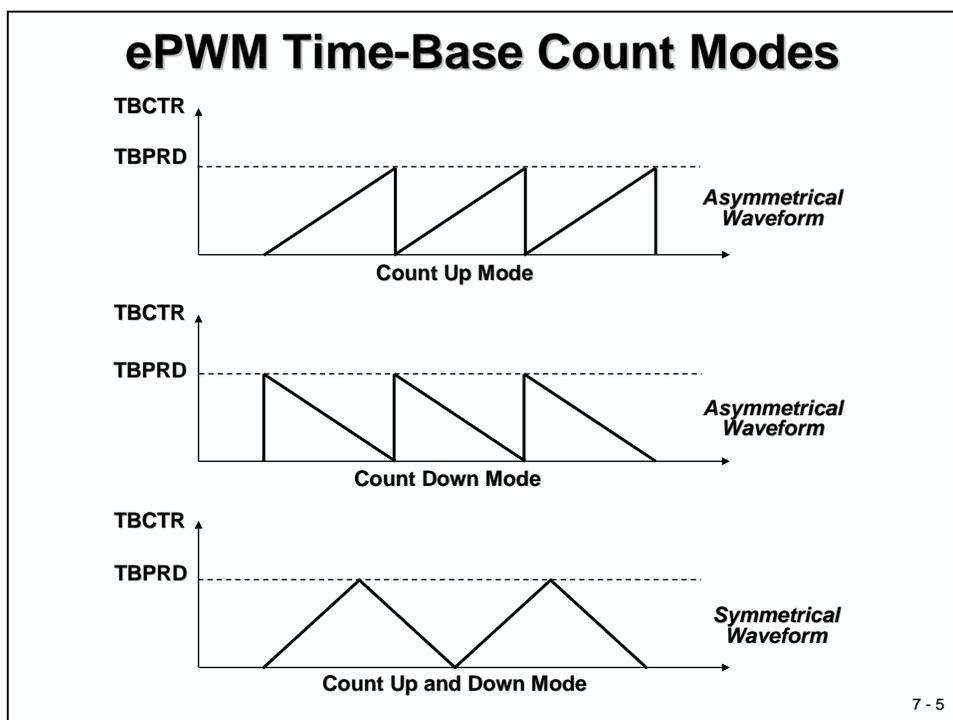
Example:

- ePWM1 counts from 0 to 6000. TBPRD = 6000
- ePWM2 register TBPHS = 2000
- ePWM3 register TBPHS = 4000

Timer Operating Modes

Each ePWM module is able to operate in one of 3 different counting modes, selected by bits 1 and 0 of register TBCTL:

- count up mode
- count down mode
- count up and down mode



Which of the three modes is used is mostly determined by the application. The first two operating modes are called "Asymmetrical" because in of the shape of the counting pattern from 0 to TBPRD (count up) or from TBPRD to 0 (count down). Also, in a three phase system, one could define three different timing events between 0 and TBPRD to switch a phase output signal to "ON" and to use the match between TBCNT and TBPRD to switch "OFF" all three phases simultaneously, thus generating an asymmetrical shape of the switch signals.

In "Symmetrical" waveform mode, the register TBCNT starts from zero to count up until it equals TBPRD. Then TBCNT turns direction to count down back to zero to finish a counting period.

Time Base Registers

To initialize the time base for one of the ePWM units it is necessary to initialize a first group of registers, shown in slide 7-6:

ePWM Time-Base Module Registers		
Name	Description	Structure
TBCTL	Time-Base Control	EPwm_xRegs.TBCTL.all =
TBSTS	Time-Base Status	EPwm_xRegs.TBSTS.all =
TBPHS	Time-Base Phase	EPwm_xRegs.TBPHS =
TBCTR	Time-Base Counter	EPwm_xRegs.TBCTR =
TBPRD	Time-Base Period	EPwm_xRegs.TBPRD =

7 - 6

To access these registers using the C programming language, we can take advantage of the source code file "DSP2833x_GlobalVariableDefs.c", which defines all memory mapped hardware registers as global variables. All variables are based on structure and union data types, also already defined by Texas Instruments and included with a master header file "DSP2833x_headers.h".

For the purpose of ePWMs this file defines 6 structures "EPwm1Regs" to "EPwm6Regs", which include all registers that belong to one of these hardware units.

Time related registers such as the period register can be accessed directly, e.g. to define a period of 6000 count pulses we can use:

```
EPwm1Regs.TBPRD = 6000;
```

For control registers, such as TBCTL, the structure members have been defined as unions. This technique allows us to access the register en bloc (union member "all") or just individual bit groups (union member "bit"). For example, a line to write the full register TBCTL would look like this:

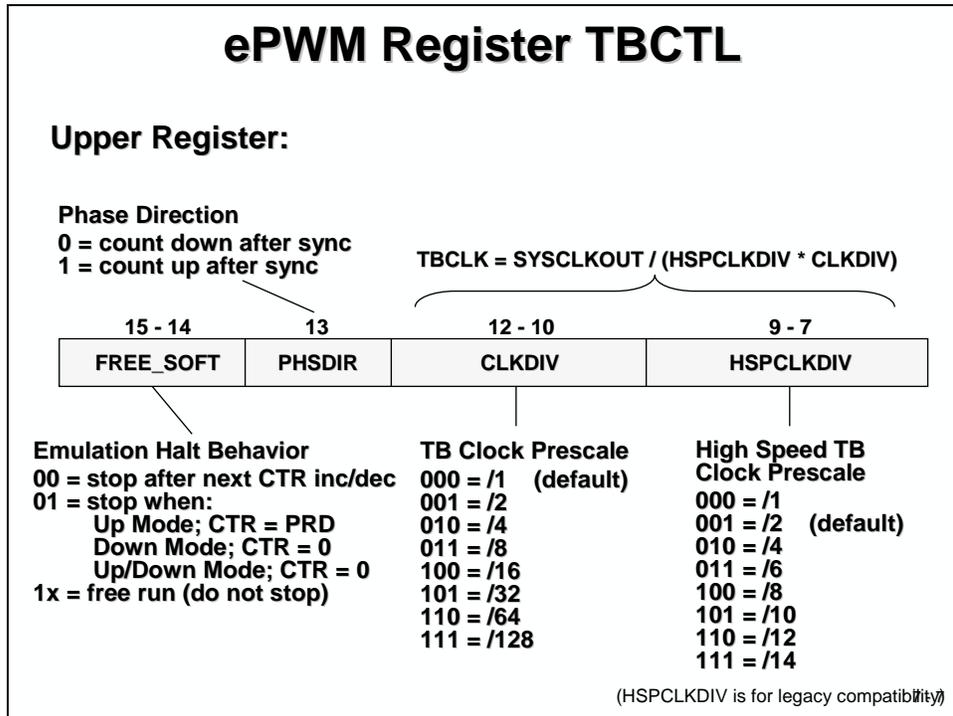
```
EPwm1Regs.TBCTL.all = 0x1234;
```

A bit field access to fields "CLKDIV" only would look like:

```
EPwm1Regs.TBCTL.bit.CLKDIV = 7;
```

Time Base Control Register TBCTL

The master control register for an ePWM unit is register TBCTL.



FREE_SOFT:

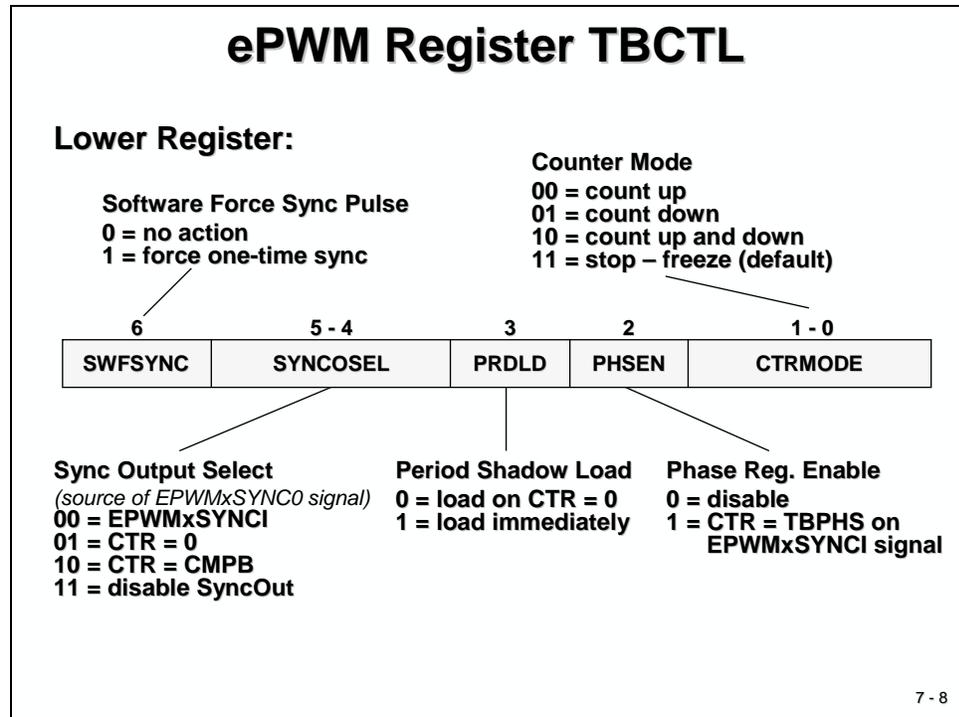
- controls the interaction between the DSC and the JTAG - Emulator.
- if the execution sequence of the code hits a breakpoint, we can specify what should happen with to this ePWM unit.

PHSDIR:

- specifies if this ePWM unit starts counting up or down after a SYNCIN pulse has been seen.
- In case of a single ePWM setup with a disabled sync in feature, this bit is a "don't care"

CLKDIV and HSPCLKDIV:

- Prescaler Bit fields to reduce the input frequency "SYSCLKOUT"
- For a 100MHz-System each pulse translates into 10 ns, for a 150MHz - System into 6.667 ns.

**SWFSYNC:**

- An instruction that sets this bit will immediately produce a "SYNCO" pulse from this ePWM unit

SYNCOSEL:

- Selection of the source for the SYNCO signal.
- If no channel synchronization is used, switch off this feature

PRDL:

- Enables (0) or disables (1) the shadow register function of TBPRD. If disabled, all write instructions to TBPRD will directly change the period register. If enabled, a write instruction will store a new value in shadow. With the next event CTR = 0 the shadow value will be loaded into TBPRD automatically.

PHSEN:

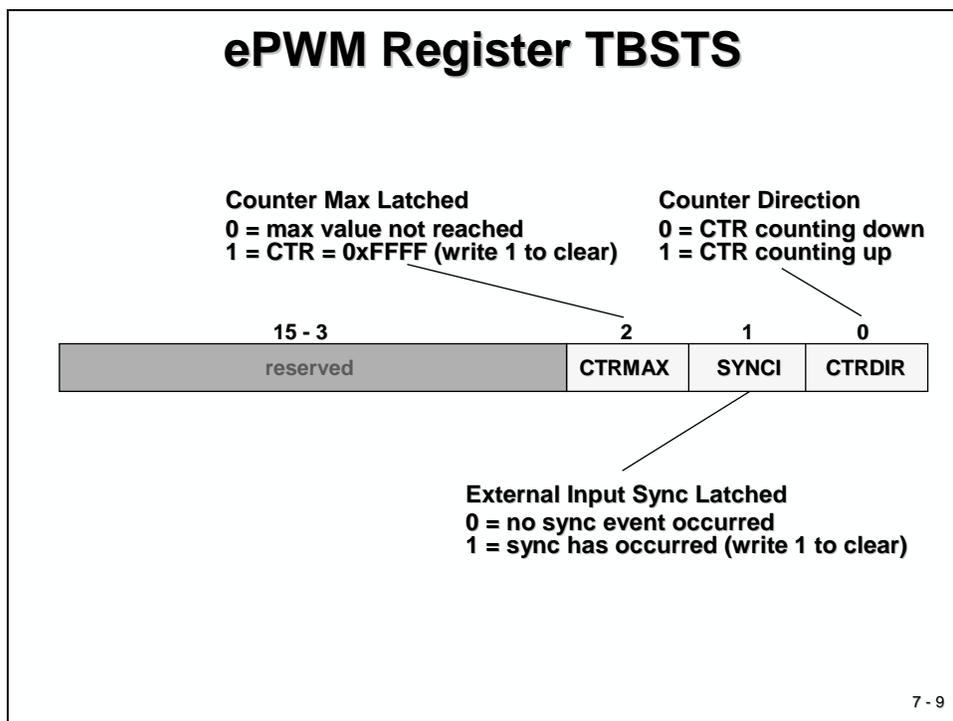
- Enables (1) the preload of register TBCTR from TBPHS by a "SYNCIN" trigger

CTRMODE:

- Defines the operating mode of this ePWM unit

Time Base Status Register TBSTS

This register flags the current status of the ePWM unit



CTRDIR:

- Indicates, if ePWM counts up (1) or down(0)

SYNCI:

- If an SYNCI event has been seen by this ePWM unit, this bit is 1, if not, it is 0.
- Note: To clear this bit, one must write a 1 into it!

CTRMAX:

- If for some reason the 16-bit counter register TBCTR overflows, bit "CTRMAX" will be set to 1. Under normal circumstances this should not happen, so we can treat this bit as a security alert signal.
- Note: To clear this bit, one must write a 1 into it!

Lab 7_1: Generate an ePWM signal

Although we have not discussed all the remaining modules inside the ePWM units, let us start an exercise to generate a single ended ePWM output signal. We will resume the discussion of additional modules in an ePWM unit later. The following procedure will guide you through the task of the exercise and will give you all necessary information.

Lab 7_1: Generate a 1 KHz Signal at ePWM1A

Objective:

- Generate a 1 KHz square wave signal at ePWM1A with a duty cycle of 50 %
- Measure it with an oscilloscope or
- Connect the signal to an external buzzer or loudspeaker

- Registers involved:
 - TBPRD: define signal frequency
 - TBCTL: setup operating mode and time prescale
 - AQCTLA: define signal shape for ePWM1A

$$TBPRD = \frac{1}{2} * \frac{T_{PWM}}{T_{SYSCLKOUT} * CLKDIV * HSPCLKDIV}$$

7 - 10

Objective

The objective of this lab is to generate a square wave signal of 1 kHz at line ePWM1A. With the help of an oscilloscope connected to header J6-1 of the Peripheral Explorer Board, we can monitor the signal. A small external circuit featuring a buzzer would allow us to make the signal audible. A possible schematic is given at the end of this exercise.

Procedure

Create a new Project File

1. Using Code Composer Studio, create a new CCS project, called **Lab7.pjt** in C:\DSP2833x_V4\Labs (or in another path that is accessible by you; ask your teacher or a technician for an appropriate location!).
2. Open the file Lab6.c from C:\DSP2833x_V4\Labs\Lab6 and save it as Lab7_1.c in C:\DSP2833x_V4\Labs\Lab7.

3. Define the size of the C system stack. In the project window, right click at project “Lab6” and select “Properties”. In category “C/C++ Build”, “C2000 Linker”, “Basic Options” set the C stack size to 0x400.

As we did in previous labs, let us add some of the files, provided by Texas Instruments, to the project:

4. In the C/C++ perspective, right click at project “Lab7” and select “**Link Files to Project**”. Go to folder “C:\tidcs\c28\dsp2833x\v131\DSP2833x_headers\source” and link:
 - **DSP2833x_GlobalVariableDefs.c**
5. Repeat the “**Link Files to Project**” step. From C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\source add:
 - **DSP2833x_CodeStartBranch.asm**
 - **DSP2833x_SysCtrl.c**
 - **DSP2833x_ADC_cal.asm**
 - **DSP2833x_usDelay.asm**
 - **DSP2833x_CpuTimers.c**
 - **DSP2833x_PieCtrl.c**
 - **DSP2833x_PieVect.c**
 - **DSP2833x_DefaultIsr.c**
6. From C:\tidcs\c28\dsp2833x\v131\DSP2833x_headers\cmd link to project “Lab7”:
 - **DSP2833x-Headers_nonBIOS.cmd**

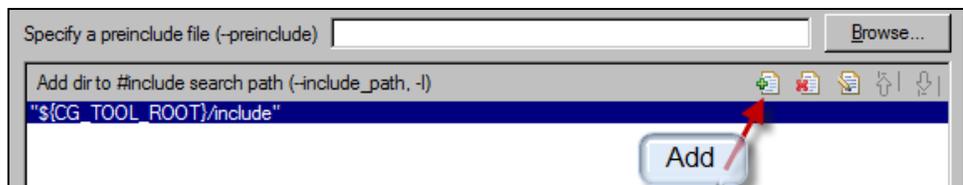
Project Build Options

7. We also have to extent the search path of the C-Compiler for include files. Right click at project “Lab7” and select “Properties”. Select “C/C++ Build”, “C2000 Compiler”, “Include Options”. In the box: “Add dir to #include search path”, add the following lines:

C:\tidcs\C28\dsp2833x\v131\DSP2833x_headers\include

C:\tidcs\c28\DSP2833x\v131\DSP2833x_common\include

Note: Use the “Add” Icon to add the new paths:



Close the Property Window by Clicking <OK>.

Build, Load and Test

8. Click the “Rebuild Active Project ” button or perform:

Project → Rebuild All (Alt +B)

and watch the tools run in the build window. If you get errors or warnings debug as necessary.

9. Load the output file in the debugger session:

Target → Debug Active Project

and switch into the “Debug” perspective.

10. Verify that in the debug perspective the window of the source code “Lab7_1.c” is high-lighted and that the blue arrow for the current Program Counter position is placed under the line “void main(void)”.

11. Perform a real time run.

Target → Run

If the code does not work as it did in Lab6, do not continue with the next steps! Go back and try to find out which step of the procedure you missed.

Modify Source Code

12. In CCS, switch to the “C/C++” perspective. In function "Gpio_select()", set multiplex register line GPIO0 to enable ePWM1A as output signal.
13. In “main()”, just after the call to the function "Gpio_select()", call a new function "Setup_ePWM1A()". Also, add a new function prototype at the beginning of “Lab7_1.c”:

```
void Setup_ePWM1A(void);
```

14. At the end of Lab7_1.c, add the definition of the new function "Setup_ePWM1A()". We will use this function to initialize ePWM1 to generate a 1 kHz square wave signal. We have to initialize the following registers:

- **EPwm1Regs.TBCTL**
- **EPwm1Regs.TBPRD**
- **EPwm1Regs.AQCTLA**

To setup the registers we can use either the "all"-member of the register union or the individual bit field member "bit". An instruction to "all" would require us to calculate a hexadecimal number for all 16 bits. By using the "bit" - structure we can leave the task to calculate the correct logical and/or -instruction to set or clear individual bit fields with the C-compiler. As

an example, an instruction to setup the operating mode to "up/down"-mode would look like this:

- **EPwm1Regs.TBCTL.bit.CTRMODE = 2;**

Furthermore, we have to calculate the value for register TBPRD. If we use the "up/down" - counting operating mode for ePWM1A, the formula is:

$$TBPRD = \frac{1}{2} * \frac{f_{SYSCLKOUT}}{f_{PWM} * CLKDIV * HSPCLKDIV}$$

The factor 1/2 must be used in "up/down operating mode. Remember that TBPRD is a 16-bit register, therefore the maximum number for TBPRD is $(2^{16}-1)$ or 65535.

Now, recall the objective is to generate a PWM signal of 1 kHz with the F28335ControlCard running at 150 MHz. Your task is to calculate appropriate numbers for CLKDIV, HSPCLKDIV and TBPRD.

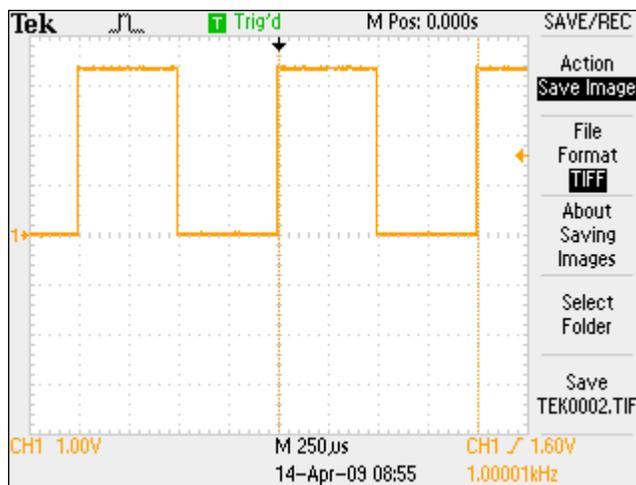
In function "Setup_ePWM1A()" initialize:

```

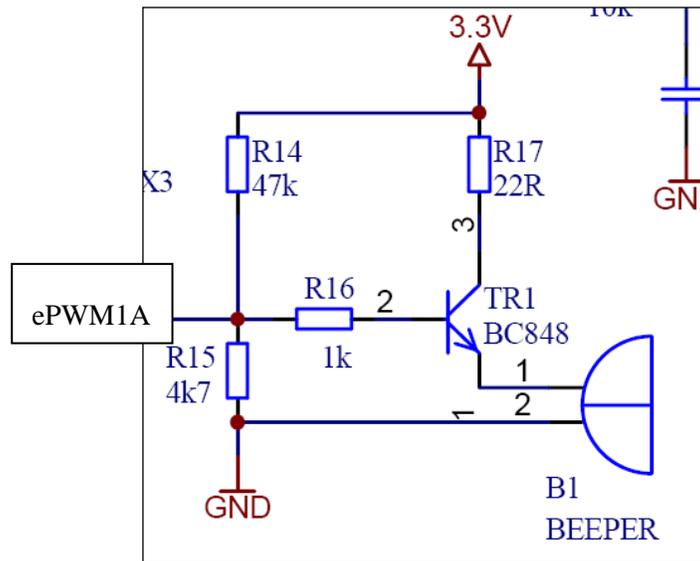
EPwm1Regs.TBCTL.bit.CLKDIV    =    ?
EPwm1Regs.TBCTL.bit.HSPCLKDIV =    ?
EPwm1Regs.TBCTL.bit.CTRMODE   =    2; // up-down mode
EPwm1Regs.TBPRD =              ?
EPwm1Regs.AQCTLA.all = 0x0006; // zero = set; period = clear
    
```

Re-Build, Load and Test

15. Now rebuild, load and test the new project. The program should still show the binary counter from Lab6 at LEDs LD1...LD4. The new addition is a 1 kHz - signal at output ePWM1A (header J6-1 at the Peripheral Explorer Board).
16. Use a scope to inspect this signal. It should look like:



17. Optional exercise: experiment with different frequencies by changing the value for register TBPRD!
18. Optional Hardware: Make your frequency audible! By adding the following circuitry to your Peripheral Explorer Board, we can do it!



Device B1 (“Beeper”) can be a Digisound F/SMD8585JSLF (Mouser Part # 847 - FSMD8585JS) or a Digisound F/PCW04A.

END of LAB 7_1

Lab 7_2: Generate a 3 - phase signal system

Now let us experiment with a 3-phase system with a phase shift of 120° and 240° between the signals. We will use ePWM1A, ePWM2A and ePWM3A for this exercise. Signal ePWM1A will be the master phase and ePWM2A and 3A will trail at 120° and 240° respectively.

Lab 7_2: Generate a 3 phase system

Objective:

- Generate three 1 KHz square wave signals at ePWM1A, 2A and 3A with duty cycles of 50 % and a phase shift of 120° and 240° between the signals
- Measure all three signals with an oscilloscope
- Registers involved:
 - TBPRD: define signal frequency
 - TBCTL: setup operating mode and time prescale
 - AQCTLA: define signal shape for ePWM1A
 - TBPHS: definition of the phase shift for 2A and 3A

$$TBPRD = \frac{1}{2} * \frac{T_{PWM}}{T_{SYSCLKOUT} * CLKDIV * HSPCLKDIV}$$

7 - 11

Objective

The objective of this lab is to generate a set of 3 square wave signals of 1 kHz each at lines ePWM1A, ePWM2A and ePWM3A. With the help of a 4 channel oscilloscope connected to header J6-1, 2 and 3 of the Peripheral Explorer Board, we can visualize the signal.

Procedure

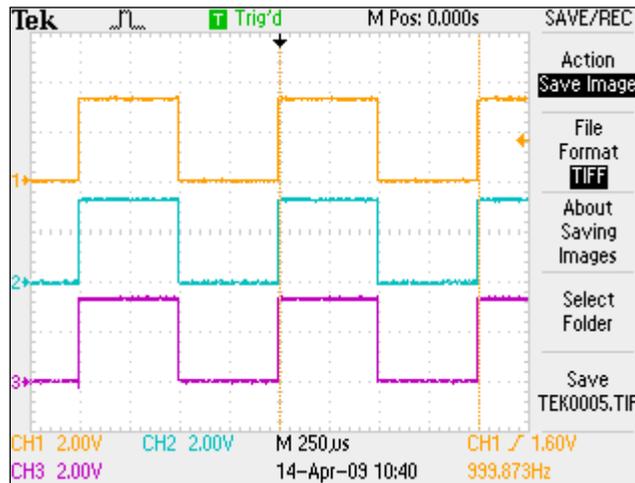
Open Project File

1. If not still open from Lab7_1, re-open project **Lab7.pjt** in “C/C++” - perspective.
2. Open file “Lab7_1.c” and save it as “Lab7_2.c”
3. Exclude file “Lab7_1.c” from build. Use a right mouse click at file “Lab7_1.c”, and enable “Exclude File(s) from Build”.

Modify Source Code

4. In file “Lab7_2.c” change the function name “Setup_ePWM1A”. Since we will also initialize ePWM2A and ePWM3A with this function, the function name is now somewhat misleading. Change the name into “Setup_ePWM”, including the function prototype and the calling line in the “main()” - loop.
5. In local function “Gpio_select()”, add instructions to initialize the pin functions of GPIO2 and GPIO4 to ePWM2A and ePWM3A respectively.
6. In function “Setup_ePWM()”, repeat the initialization for ePWM1A with the same instructions for ePWM2A and ePWM3A. Apply identical values as for ePWM1A to the following registers:
 - EPwm2Regs.TBCTL
 - EPwm2Regs.TBPRD
 - EPwm2Regs.AQCTLA
 - EPwm3Regs.TBCTL
 - EPwm3Regs.TBPRD
 - EPwm3Regs.AQCTLA

If you now recompile, load and test your new code, you should get 3 identical 1 kHz - signals with zero phase-shift between the 3 ePWM lines:



7. Now let us add the phase shift commands between ePWM1A, ePWM2A and ePWM3A. To do so, we will have to program the phase registers of ePWM2A and ePWM3A. Also, we must define ePWM1A as the master phase to generate a SYNCOUT pulse each time its counter register TBCNT equals zero. For ePWM2, we must enable a SYNCIN - pulse and also define SYNCIN as SYNCOUT to drive it into ePWM3 unit. Recall that the period register TBPRD of ePWM1A has been initialized with a value that corresponds to a time period of 1 millisecond. Now for ePWM2 and ePWM3 we need a phase shift of $1/3^{\text{rd}}$ and $2/3^{\text{rd}}$ of that value preloaded in register TBPHS.

Summary: In function “Setup_ePWM()” add the following instructions:

EPwm1Regs.TBCTL:

- Sync Out Select: generate a signal if CTR = 0

EPwm2Regs.TBCTL:

- Set phase enable
- Sync Out Select: SYNCIN = SYNCOUT

EPwm2Regs.TBPHS:

- Load it with 1/3rd of TBPRD
- Since TBPHS is a union type, a valid access is made like this:

EPwm2Regs.TBPHS.half.TBPHS = ????? ;

Epwm3Regs.TBCTL:

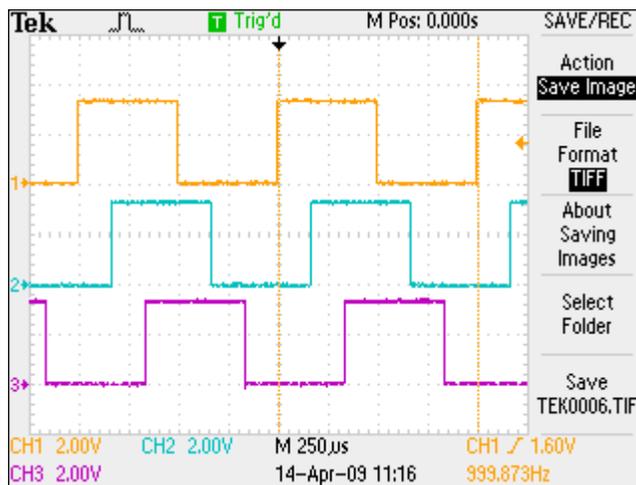
- Set phase enable

EPwm3Regs.TBPHS:

- Load TBPHS with 2/3rd of TBPRD

Build, Load and Test

8. Now build, load and test the modified project. Using an oscilloscope you should see 3 time shifted signals on ePWM1A, ePWM2A and ePWM3A:



END OF LAB 7_2

Purpose of Pulse Width Modulation

In Lab7_1 and Lab7_2 we created square wave signals with a pulse duty cycle of 50% low and 50% high. We are also able to produce a sequence of time-shifted signals on a group of output signals. But so far, we are still not able to change or “to modulate” the width of the pulses - even though this hardware unit is called “Pulse Width Modulation”. This modulation is based on another set of control registers of a unit called “Compare Module”.

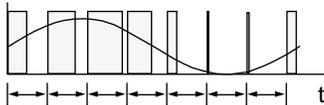
Before we discuss the compare module, let us look into the technical background and purpose of PWM.

What is Pulse Width Modulation?

- ◆ **PWM is a scheme to represent a signal as a sequence of pulses**
 - ◆ **fixed carrier frequency**
 - ◆ **fixed pulse amplitude**
 - ◆ **pulse width proportional to instantaneous signal amplitude**
 - ◆ **PWM energy \approx original signal energy**



Original Signal



PWM representation

7 - 12

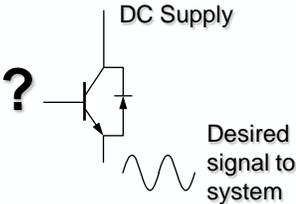
PWM is nothing more than a digital output signal with binary amplitude, 0 or 1. In technical terms, the voltage at this output pin is either 0V or 3.3V. However, we can setup a point within a period, at which we switch the output from 0 to 3.3V and vice versa. By changing this set-point between 0 and 100% of the period, we can adjust the duty cycle of the output signal.

With a PWM signal we can represent any analogue output signal as a series of digital pulses! All we need to do with this pulse series is to integrate it (with a simple low pass filter) to imitate the desired signal. This way we can build a sine wave shaped output signal. The more pulses we use for one period of the desired signal, the more precisely we can imitate it. We speak very often of two different frequencies, the PWM-frequency (or sometimes “carrier frequency”) and the desired signal frequency.

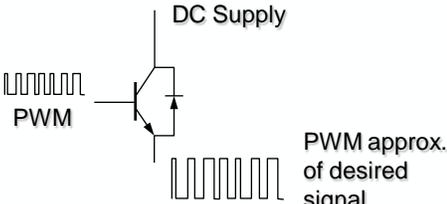
A lot of practical applications have an internal integrator. For example the windings of an electrical motor are perfectly suited to behave as a low-pass filter.

Why use PWM with Power Switching Devices?

- ◆ **Desired output currents or voltages are known**
- ◆ **Power switching devices are transistors**
 - Difficult to control in proportional region
 - Easy to control in saturated region
- ◆ **PWM is a digital signal → easy for DSP to output**



Unknown Base Signal



Base Signal known with PWM

7 - 13

One of the most used applications of PWM is (A) Digital Motor Control (DMC) and (B) Digital Power Supply (DPS) - sometimes also called “Switched Power Supply”.

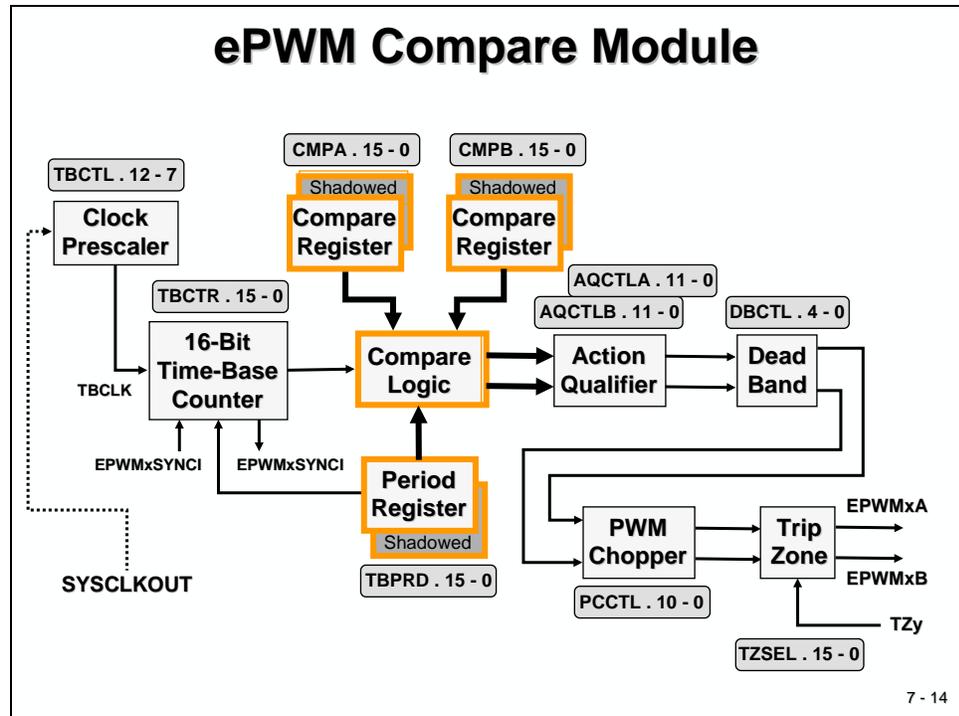
Why is that? Answer: The overall goal is to control electrical drives by inducing harmonic voltages and currents into the windings of the motor. This is done to avoid electromagnetic distortions of the environment and to achieve a high power factor. To induce a sine wave shaped signal into the windings of a motor we would have to use an amplifier to achieve high currents. The simplest amplifier is a standard NPN or PNP transistor that proportionally amplifies the base current into the collector current. The problem is, for high currents we cannot force the transistor into its linear region; this would generate a lot of thermal losses and likely to exceed its maximum power dissipation.

The solution is to use this transistor in its static switch states only (On: $I_{ce} = I_{cesat}$, Off: $I_{ce} = 0$). In these states, a transistor has its minimum power dissipation. AND: by adapting the switch pattern of a PWM (recall: amplitude is 1 or 0 only) we can induce a sine wave shaped current!

Environmentally friendly power supply units use switching technologies to increase the efficiency factor of traditional power supply units. Instead of converting a lot of primary energy just in pure thermal energy, these techniques, known as “Buck”- or “Boost” - converters, allow customers to build reduce the package of their goods and more important to help save our environment.

ePWM Compare Unit

The module to control the active phase of a pulse pattern and the position of the switching points in a PWM is called the “Compare Unit”, highlighted in the next slide:



Its functionality is based on a pair of registers, called “Compare Register A and B” (CMPA and CMPB). Note that there is no relationship between the letters A and B in these registers and the naming of the two output signals in the lower right corner, EPWMxA and EPWMxB. This naming convention is a little bit misleading, it would have been better to use different names such as CMP1 and CMP2, but the decision was made by Texas Instruments.

Depending on the pre-selected operating mode of the ePWM unit, it is possible to define 2 or 4 events within a period of the PWM - frequency, by choosing the appropriate values in CMPA and/or CMPB.

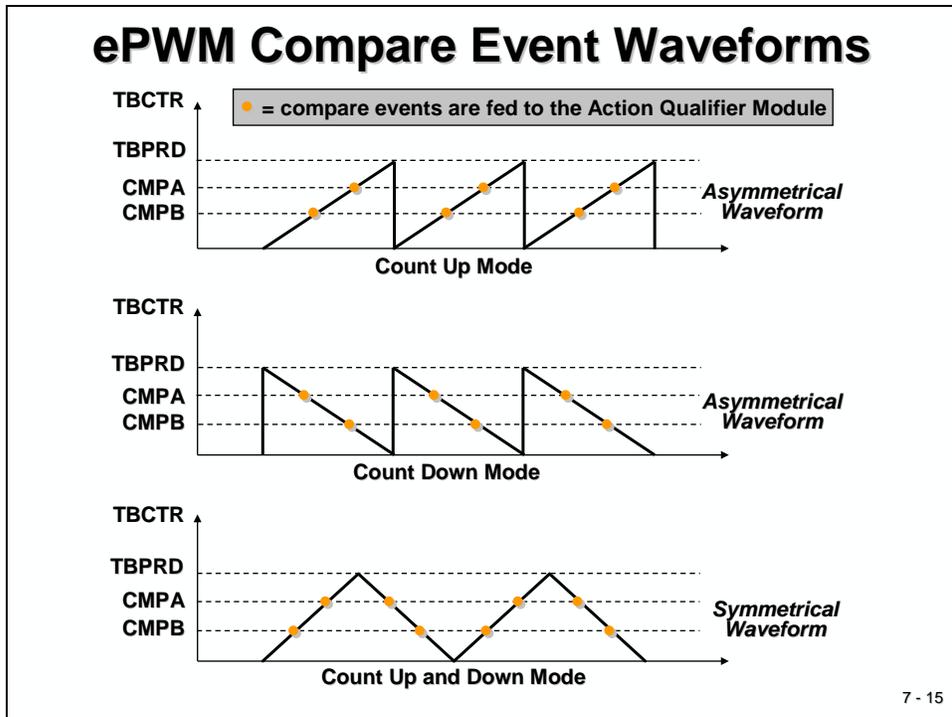
Have you kept in mind these operating modes? If not, please review Slide 7-5. Here is a summary:

- count up mode
- count down mode
- count up and down mode

In Lab7_1 and Lab7_2 we used the up/down mode to generate the 1 kHz signal. We have used two events to change the voltage level on the output line:

- counter register is zero (TBCNT = 0)
- counter register is equal to period register (TBCNT = TBPRD)

Now we can use 2 or 4 more events:



Instead of using 0 or TBPRD we now can use up to 4 more points per period to trigger an action. What action? Well, the type of action will be defined in another module, coming next. For now let us summarize the Compare Unit registers:

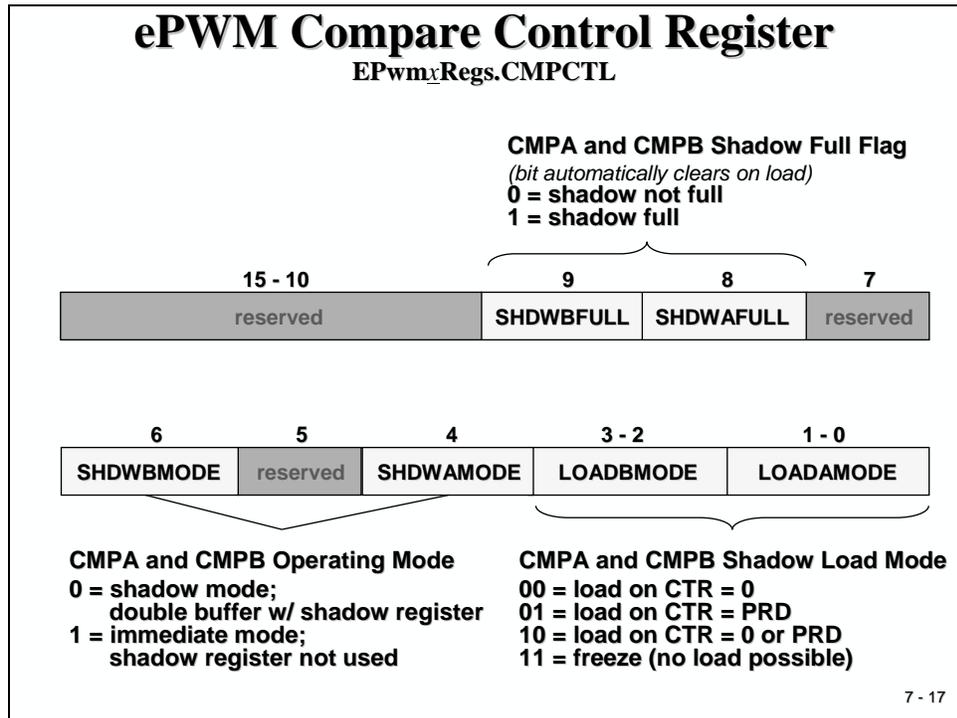
ePWM Compare Module Registers

Name	Description	Structure
CMPCTL	Compare Control	EPwm x Regs.CMPCTL.all =
CMPA	Compare A	EPwm x Regs.CMPA =
CMPB	Compare B	EPwm x Regs.CMPB =

7 - 16

While CMPA and CMPB are just number registers to specify the point of action relatively to the counter register, CMPCTL controls the operation of the shadow registers behind CMPA and CMPB. Do you recall the purpose of “Shadow” registers? Shadows or Background

registers can be used to prepare a new value for the next coming period while the current period is still running and may still rely on the value in the foreground.



LOAD_xMODE:

- define the hardware event, which will copy a value from background into the active foreground register

SHDW_xMODE:

- enable (0) or disable (1) the background update mode. If disabled, all write instructions will immediately change the value in register CMPA or CMPB

SHDW_xFULL:

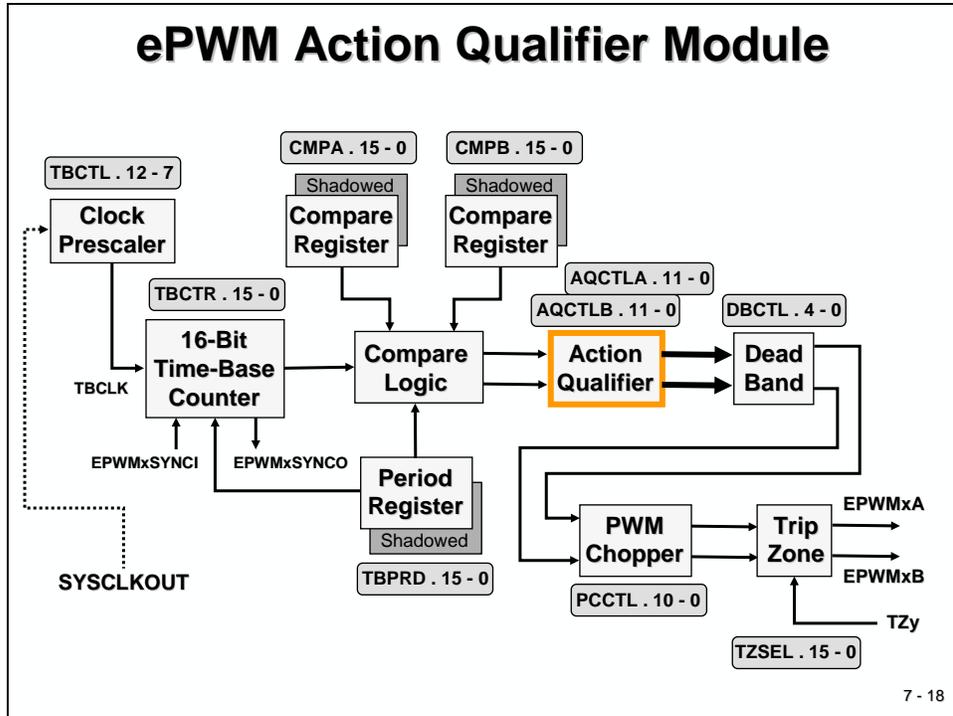
- read only status field. If shadow is full (1) and the hardware copies the value into foreground, the bit is cleared automatically

For most applications it is highly recommended to use this shadow feature, since it eases the urgency of accesses to the CMP registers, when we change these values on a cycle-by-cycle base, sometimes called “on the fly”.

After a hardware reset, or by default, shadow mode is enabled and LOAD_xMODE is set to “load on CTR=0”; If we don’t initialize CMPCTL at all, the default mode will be active.

ePWM Action Qualifier Unit

Now let us inspect another unit, which we need to generate a series of pulses at EPWMxA and EPWMxB - the Action Qualifier Module.



We can initialize this unit by a set of two control registers, AQCTLA for output line A and AQCTLB for line B. For each of the 6 events on a timescale (Zero-match; CMPA-up, CMPB - up, Period, CMPA - down and CMPB - down) we can specify a certain action at the corresponding signal line:

- set line to high (rising edge)
- clear line to low (falling edge)
- toggle the line (low to high OR high to low)
- do nothing (ignore this event)

Furthermore we can also force the corresponding line to a certain level by executing a software instruction in one of two software force registers. In most cases, the latter option is not used, because it cannot be synchronized with other hardware activities of the PWM unit. Sometimes however, especially for emergency routines, it is welcome to have such a force option.

The next slide summarizes the available options for the Action Qualifier Unit. The icons used in this slide will also be used in the following slides to highlight some popular control patterns for PWM systems.

ePWM Action Qualifier Actions

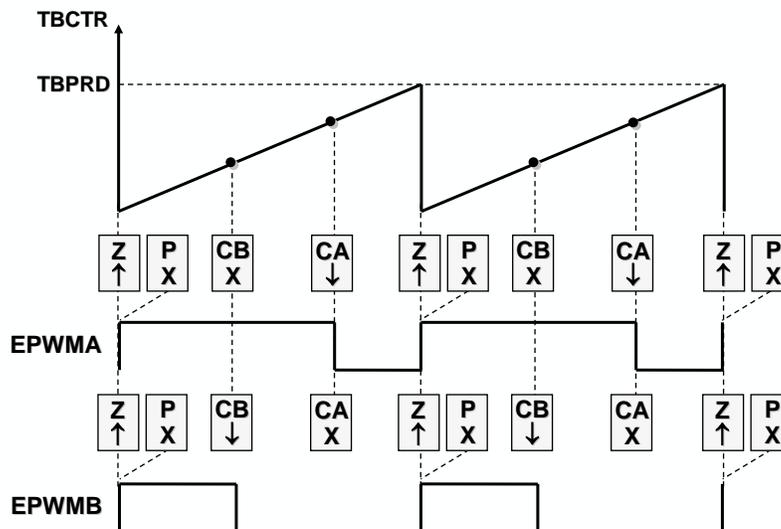
S/W Force	Time-Base Counter equals:				EPWM Output Actions
	Zero	CMPA	CMPB	TBPRD	
SW X	Z X	CA X	CB X	P X	Do Nothing
SW ↓	Z ↓	CA ↓	CB ↓	P ↓	Clear Low
SW ↑	Z ↑	CA ↑	CB ↑	P ↑	Set High
SW T	Z T	CA T	CB T	P T	Toggle

7 - 19

Independent Duty Cycle on line A and B

The first example uses the lines A and B in count-up mode. The duty cycles are independently controlled by CMPA for line A and CMPB for line B.

Independent Modulation on EPWMA / B

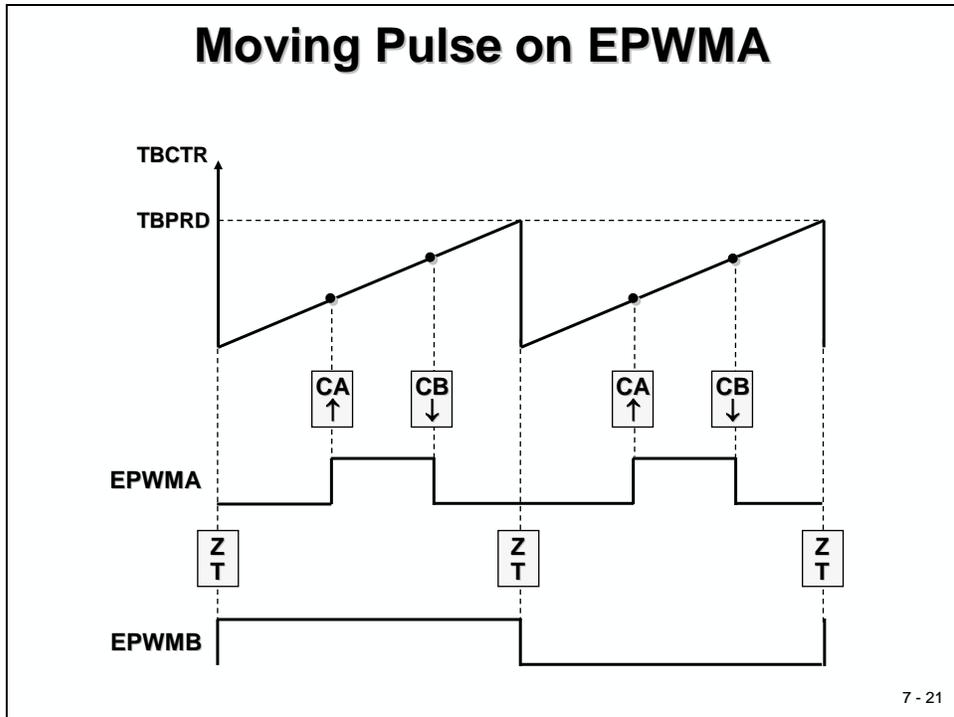


7 - 20

Moving Pulse on EPWMA

This example uses EPWMB just to indicate half of the period of the PWM - frequency. CMPA and CMPB are both used to control

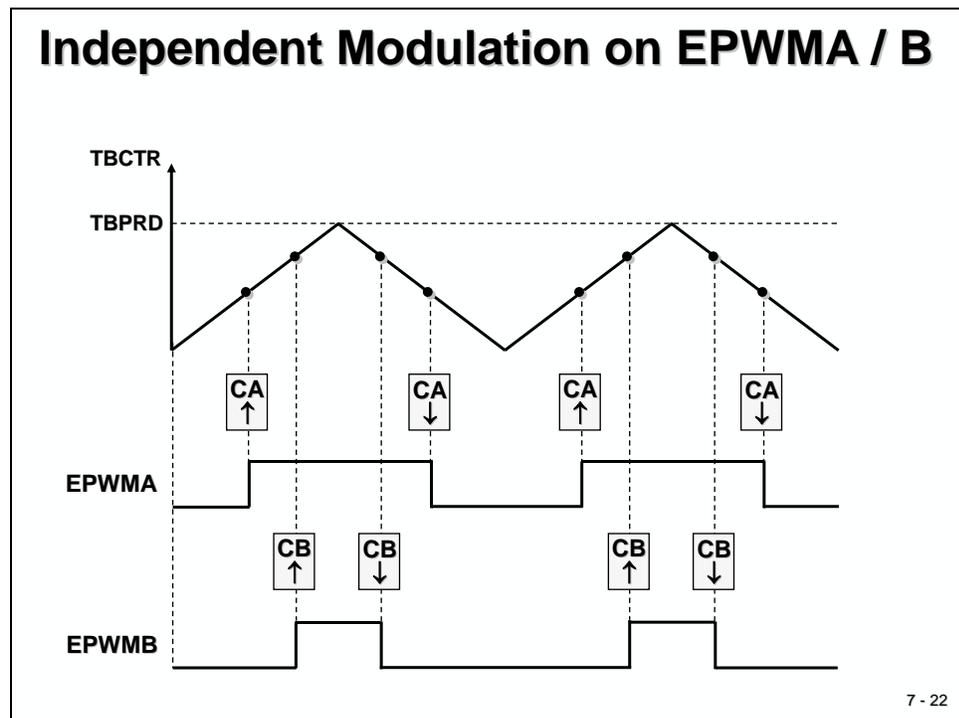
- (1) the position and
- (2) the size of the pulse on line EPWMA



Independent modulation of two pulses

Here both lines EPWMA and EPWMB carry a control signal. EPWMA is solely controlled by CMPA and is always centered on the period match event. By reducing the difference between CMPA and TBPRD we can reduce the size of the pulse, by extending the difference the pulse will grow towards 100%.

Register CMPB is used to control the pulse size of EPWMB independently of EPWMA. In this example output pulse EPWMB is also center aligned on the period match event.



There are many more application examples and operating modes than those, which we discussed in the previous slides, especially when you recall typical 3-phase systems with their well known complementary switching patterns.

Let us postpone these industrial applications for now and focus on what we have learned so far. To perform an exercise with the basic pulse sequences shown above, we will have to include the Action Qualifier Unit (AQU) into our exercises.

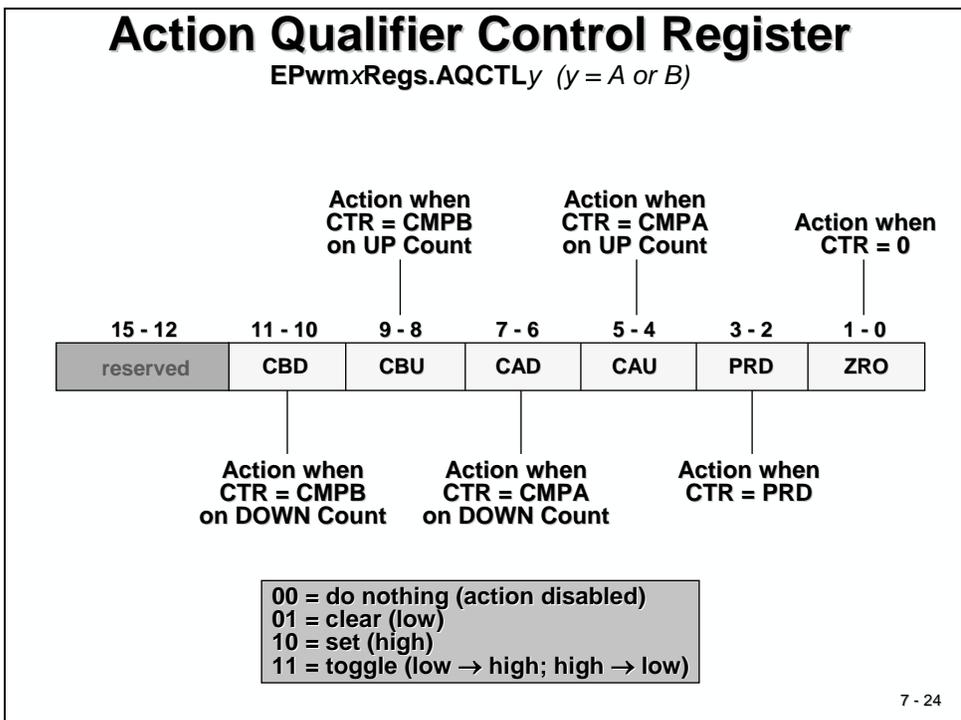
We have not discussed the layout of the control registers for the AQU. The group of registers is shown on the next slide.

Action Qualifier Registers

ePWM Action Qualifier Module Registers		
Name	Description	Structure
AQCTLA	AQ Control Output A	EPwm _x Regs.AQCTLA.all =
AQCTLB	AQ Control Output B	EPwm _x Regs.AQCTLB.all =
AQSFRC	AQ S/W Force	EPwm _x Regs.AQSFRC.all =
AQCSFRC	AQ Cont. S/W Force	EPwm _x Regs.AQCSFRC.all =

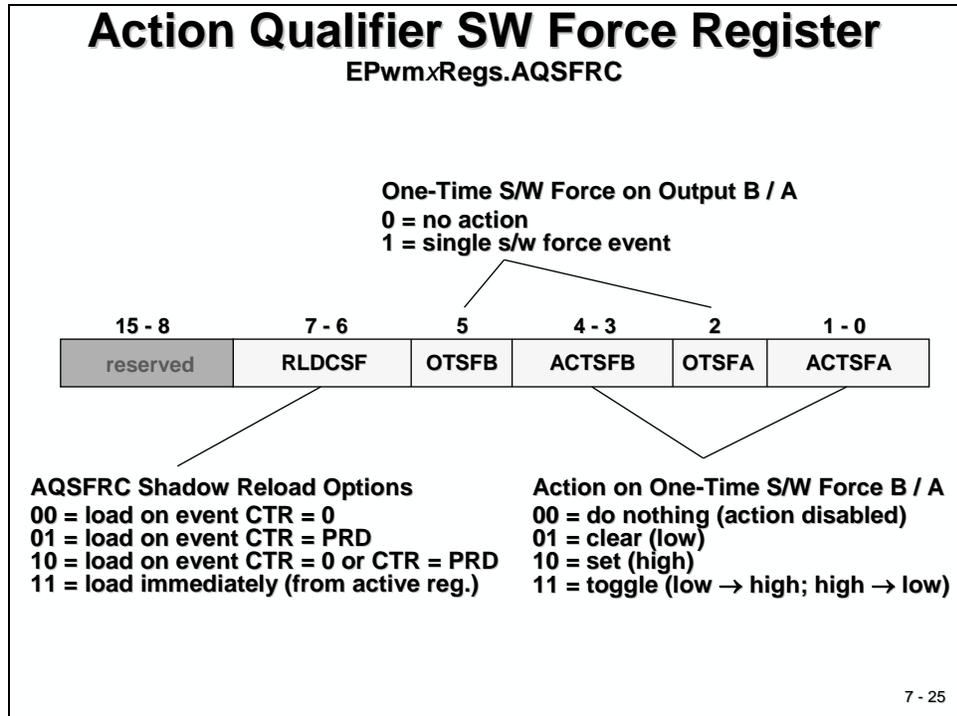
7 - 23

Action Control Register A and B

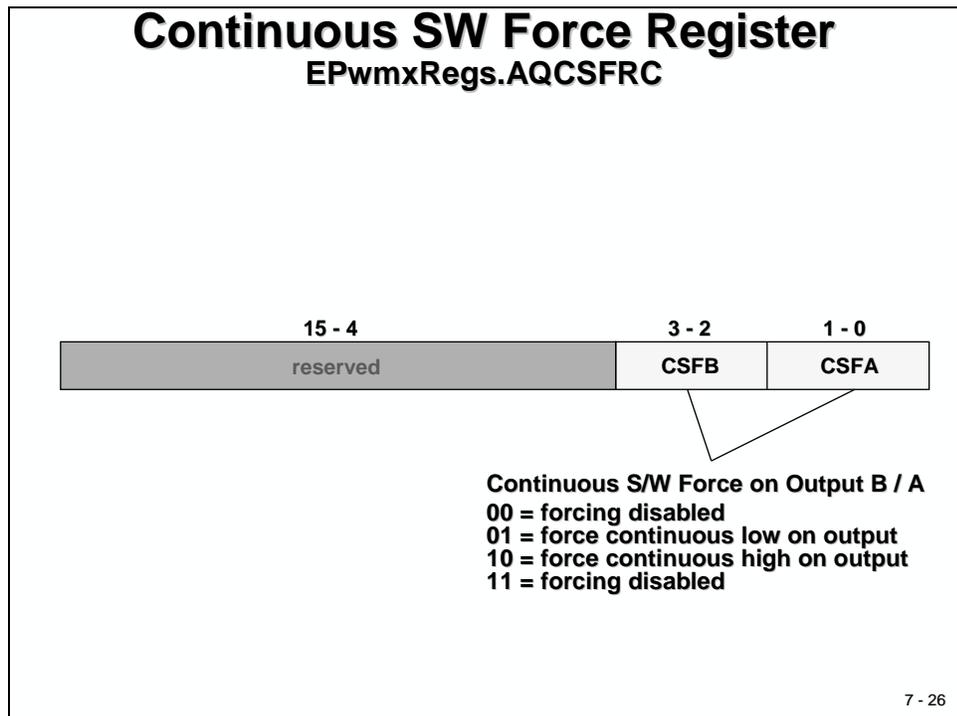


Software Forcing Registers

This register allows forcing an output line into a defined state. “One-Time” stands for the duration of the current period of the PWM - frequency.



“Continuous Force” will hold the line permanently in the selected state.



Lab 7_3: A 1 kHz with variable pulse width

Now let us experiment with a variable pulse width signal. The starting point is again Lab7_1. We will now use CpuTimer0 as a time-base to change the pulse width of the 1 kHz signal once every 100 milliseconds between 0 and 100 %.

Lab 7_3: 1 KHz Signal with variable pulse width at ePWM1A

Objective:

- Generate a 1 KHz square wave signal at ePWM1A with a variable duty cycle between 0 and 100%
- Measure the pulse with an oscilloscope
- Registers involved:
 - TBPRD: define signal frequency
 - TBCTL: setup operating mode and time prescale
 - CMPA: setup the pulse width for ePWM1A
 - AQCTLA: define signal shape for ePWM1A

$$TBPRD = \frac{1}{2} * \frac{T_{PWM}}{T_{SYSCLKOUT} * CLKDIV * HSPCLKDIV}$$

7 - 27

Objective

The objective of this lab is to generate a square wave signal of 1 kHz at line ePWM1A. With the help of an oscilloscope connected to header J6-1 of the Peripheral Explorer Board, we can monitor the signal. Using CPU - Timer 0, we will change CMPA between 0 and TBPRD to generate a pulse width between 100 and 0%.

Procedure

Open Project File

1. In the “C/C++” perspective of CCS open or re-open project **Lab7.pjt**.
2. Open file “Lab7_1.c” and save it as “Lab7_3.c”
3. Exclude file “Lab7_2.c” from build. Use a right mouse click at file “Lab7_2.c”, and enable “Exclude File(s) from Build”.

4. In file “Lab7_3.c”, edit the function “Setup_ePWM1A()”. We will again use count up/down mode, so we can keep the existing setup for bit field TBCTL.CTRMODE. However, now we would like to set ePWM1A to 1 on “CMPA - up match” and to clear ePWM1A on event “CMPA - down match”. Change the setup for register AQCTLA accordingly!
5. In the function “Setup_ePWM1A()” add a line to initialize CMPA to 0, which will define a pulse width of 100%:

EPwm1Regs.CMPA.half.CMPA = 0;

6. In “main()”, change the function call “ConfigCpuTimer()” to define a period of 100 microseconds for timer 0:

ConfigCpuTimer(&CpuTimer0, 150, 100);

7. CpuTimer0 is still active from Lab exercise Lab6. It has been initialized to request an interrupt service once every 100 microseconds. Now we can use its interrupt service routine “cpu_timer0_isr()” to increment the value in register CMPA until it reaches the value in TBPRD - thus we will change the pulse width gradually from 100% to 0%. If you like, you can add a second sequence to increase the pulse width of ePWM1A again back to 100%.

Note: All registers of ePWM1 are read- and writable. To compare the current value of CMPA against TBPRD you can use:

if (EPwm1Regs.CMPA.half.CMPA < EPwm1Regs.TBPRD) ...

Build, Load and Test

8. Now build, load and test the modified project. A screenshot of signal ePWM1A could look like this:



Result: The pulse width of your signal should change gradually between 100% and 0 %.

END of LAB 7_3

Lab 7_4: a pair of complementary 1 kHz-Signals

Most power electronic systems require pairs of PWM pulse series to control two power switches in such a way, that if one switch is on (conducting), the other switch is off (open-circuit). In the following exercise you will modify Lab7_3 to generate such a pair of output pulses at ePWM1A and ePWM1B. Again we will use CpuTimer0 as a time-base to change the pulse width of the 1 kHz signal every 100 milliseconds between 0 and 100 %.

Lab 7_4: a pair of complementary 1 KHz signals at ePWM1A and ePWM1B

Objective:

- Generate a 1 KHz square wave signal at ePWM1A with a variable duty cycle between 0 and 100%
- Generate a complementary signal at ePWM1B
- Measure the pulses with an oscilloscope

- Registers involved:
 - TBPRD: define signal frequency
 - TBCTL: setup operating mode and time prescale
 - CMPA: setup the pulse width for ePWM1A / 1B
 - AQCTLB: define signal shape for ePWM1B
 - AQCTLA: define signal shape for ePWM1A

7 - 28

Objective

The objective of this lab is to generate a square wave signal of 1 kHz at line ePWM1A and a second signal at ePWM1B with opposite voltage levels. With the help of an oscilloscope connected to header J6-1 of the Peripheral Explorer Board, we can monitor the signal. Based on CPU - Timer 0, we will change CMPA between 0 and TBPRD to generate a pulse width between 100 and 0%.

Procedure

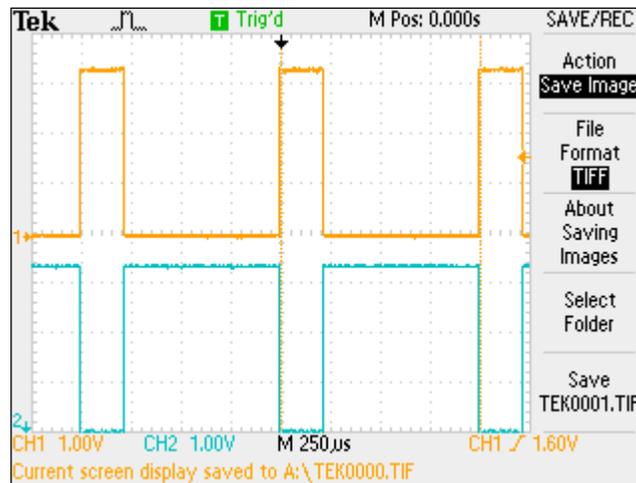
Open Project File

1. If not still open from Lab7_3, re-open project **Lab7.pjt** in the “C/C++” perspective of Code Composer Studio.
2. Open file “Lab7_3.c” and save it as “Lab7_4.c”
3. Exclude file “Lab7_3.c” from build. Use a right mouse click at file “Lab7_3.c”, and enable “Exclude File(s) from Build”.

4. In file “Lab7_4.c” edit function “Gpio_select()”. In the multiplex block enable line GPIO1 to drive ePWM1B.
5. Rename function “Setup_ePWM1A()” to “Setup_ePWM1()”, because we will now initialize both line A and B with this function. Also, rename the function prototype at the beginning of “Lab7_4.c” and the function call in “main()”.
6. In “Setup_ePWM1()”, add a line to initialize register EPwm1Regs.AQCTLB. Recall that we initialized EPwm1Regs.AQCTLA to set ePWM1A on CMPA - up and to clear ePWM1A on CMPA - down match. For register EPwm1Regs.AQCTLB we will have to modify that setup to generate a complementary signal at ePWM1B.

Build, Load and Test

7. Now build, load and test the modified project. A oscilloscope screenshot of signal ePWM1A and ePWM1B should look like the following graph:

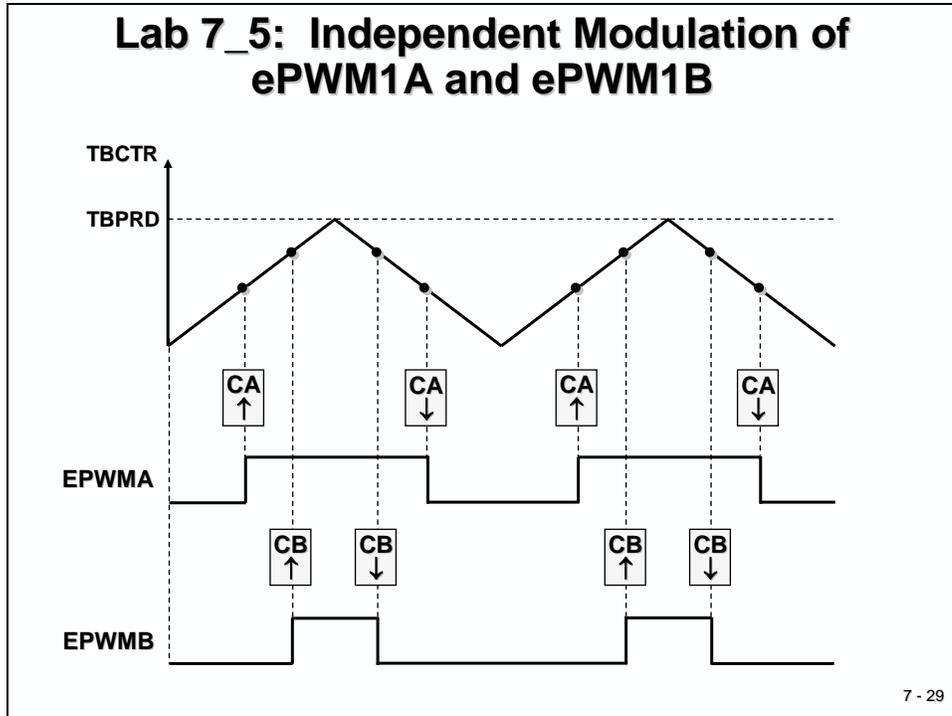


Result: The pulse width of your pair of signals should change gradually between 100% and 0 %.

END of LAB 7_4

Lab 7_5: Independent Modulation on ePWM1A / 1B

Before we continue to discuss other modules of the ePWM - units we will perform an exercise to produce the exact pulse pattern, as shown in Slide 7-29:



Objective

The objective of this lab is to generate a square wave signal of 1 kHz at line ePWM1A and a second signal at ePWM1B with independent modulation of the pulse widths. Signal ePWM1A will be controlled by register CMPA and ePWM1B by register CMPB. This time we will also use a real-time operating mode to change the values of CMPA and CMPB in a variable watch window while the program is running.

Procedure

Open Project File

1. If not still open from Lab7_3, re-open project **Lab7.pjt** in the “C/C++” perspective of Code Composer Studio.
2. Open file “Lab7_4.c” and save it as “Lab7_5.c”
3. Exclude file “Lab7_4.c” from build. Use a right mouse click at file “Lab7_4.c”, and enable “Exclude File(s) from Build”.
4. In the function “Setup_ePWM1()”, change the line to initialize register EPwm1Regs.AQCTLB. The new setup for AQCTLB should be to set ePWM1B on CMPB - up and to clear ePWM1B on CMPB - down match.

- After the line to initialize register TBPRD, add two lines to set register CMPA and CMPB to initially generate a pulse width of 50%.

EPwm1Regs.CMPA.half.CMPA = EPwm1Regs.TBPRD / 2;

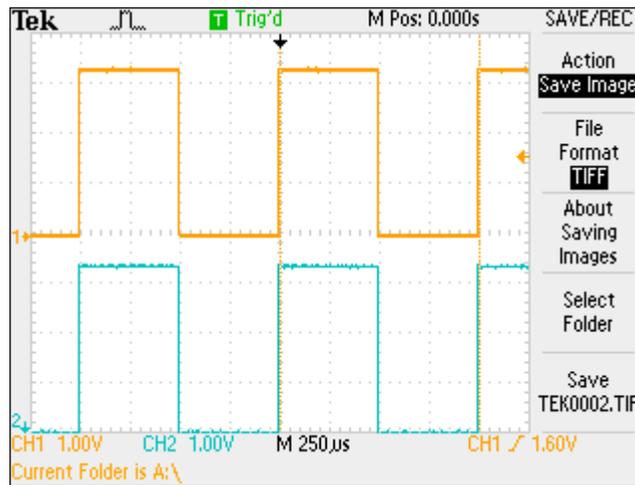
EPwm1Regs.CMPB = EPwm1Regs.TBPRD / 2;

Note the difference between the structure data types of the two registers. This difference is caused by a second operating mode, called “High Resolution PWM” (HRPWM), which is available only for the signal line(s) ePWMxA. To support this mode, TI has enhanced the structure type for register CMPA.

- In the function “cpu_timer0_isr()”, remove all instructions to change the pulse width by register CMPA. We will use a fixed pulse width for this exercise, initially 50% for both ePWM1A and ePWM1B.

Build, Load and Test

- Now build, load and test the modified project. A oscilloscope screenshot of signal ePWM1A and ePWM1B should look like the following graph:



- Stop the code execution:

Target → Halt, followed by

Target → Reset → Reset CPU

- Now open a Watch Window:

View → Watch

In window “Watch 1” add the two variables:

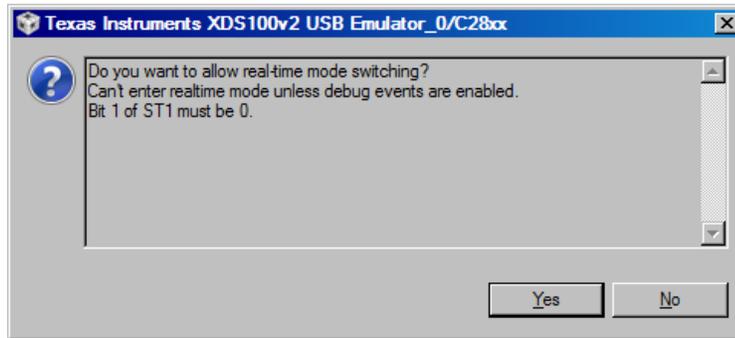
EPwm1Regs.CMPA.half.CMPA and

EPwm1Regs.CMPB

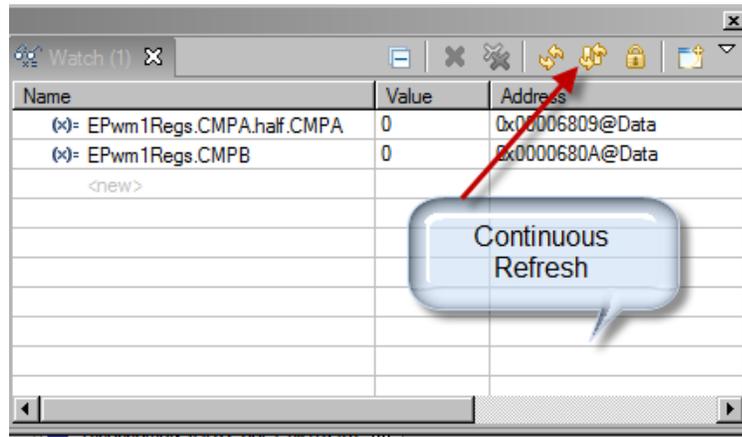
10. Enable Real Time Debug Mode:

Target → Advanced → Enable Silicon Realtime Mode

A warning might pop up on your screen to inform you, that you will enter a real time data exchange debug mode now. Answer this window with “Yes”:



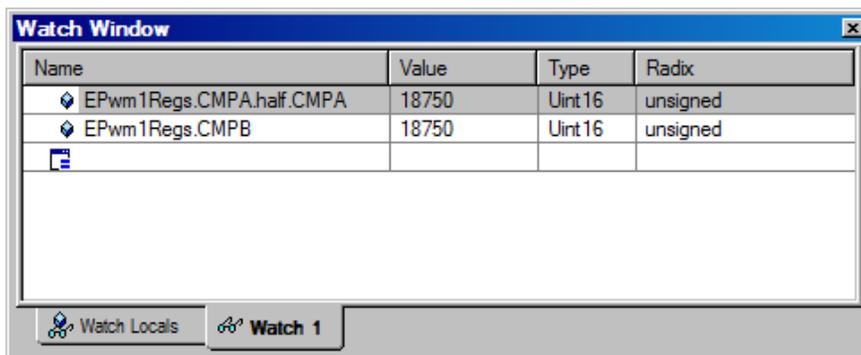
In the Watch window, enable the icon “Continuous Refresh”:



11. Restart your Test, this time with a new sequence:

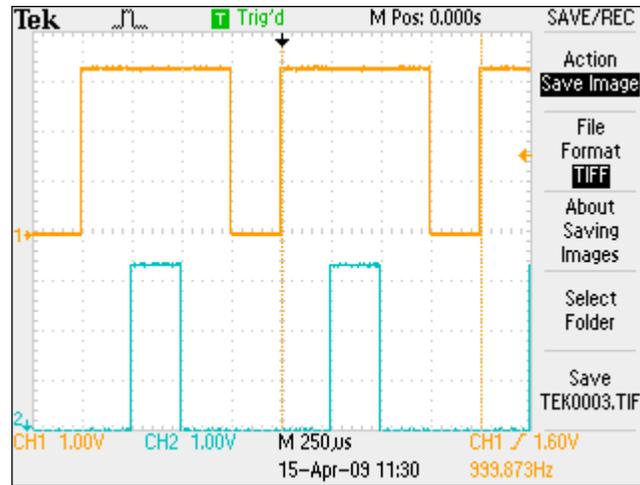
Scripts → Realtime Emulation Control → Run_Realtime_with_Restart

Your Watch window should display the current values for CMPA and CMPB:



Now, while the code is still running, change the values in CMPA and CMPB to 9375 and 28125 respectively.

The result should look like this:



Try other combinations of CMPA and CMPB and verify the changes with your scope!

12. If you are done with this exercise, it is important to fully halt the DSC. Since we are currently running in real time mode, we have to apply a different command sequence:

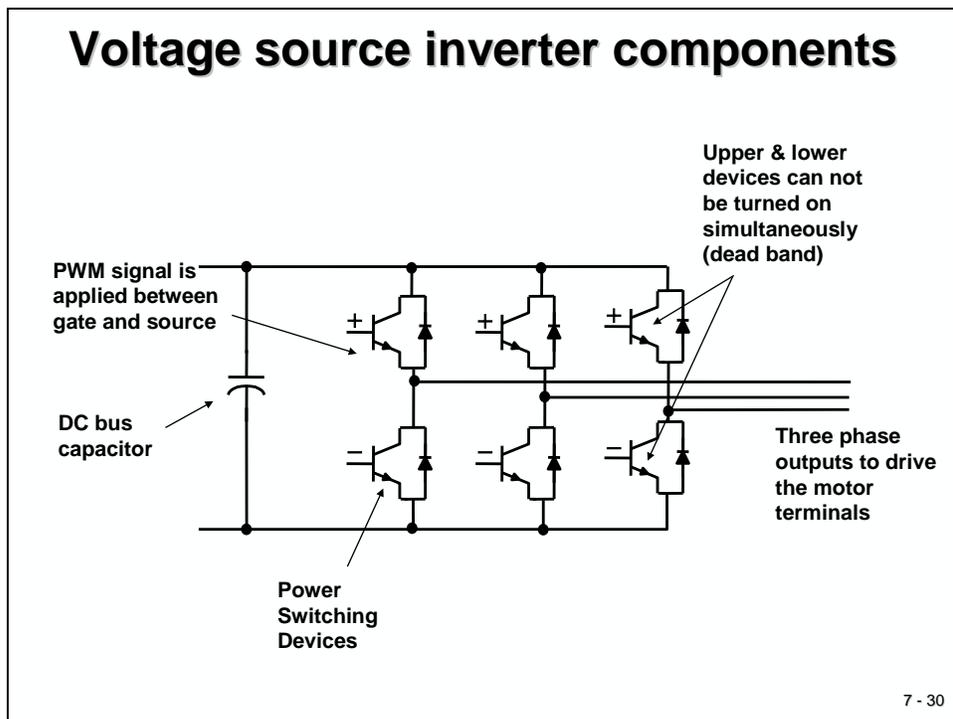
Scripts → Realtime Emulation Control → Full_Halt_with_Reset

END of LAB 7_5

ePWM Dead Band Module

Motivation for Dead - Band

In switched mode power electronics, a typical configuration to drive a 3-phase system is shown in the next slide (Slide 7-30). A typical system consists of a 3-phase current or voltage injection circuit, in which a pair of power switches per phase is controlled by a sequence of PWM - pulses. A phase current flows either from a DC bus voltage through a top switch into the winding of a motor or via a bottom switch from the motor winding back to ground. Of course, we have to prevent both switches from conducting at the same time.



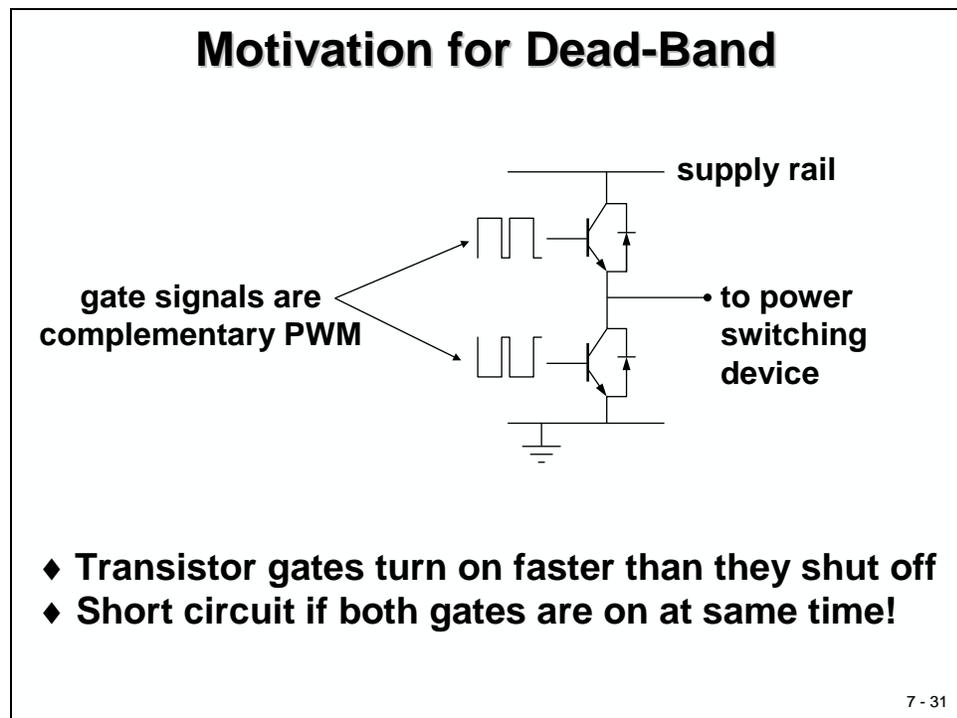
A minor problem arises from the fact that power switches usually turn on faster than they turn off. If we would apply an identical but complementary pulse pattern to the top and bottom switch of a phase, we would end up in a short period in time with a shoot-through situation.

Dead-band control provides a convenient means of combating current “shoot-through” problems in a power converter. “Shoot-through” occurs when both the upper and lower transistors in the same phase of a power converter are on simultaneously. This condition shorts the power supply and results in a large current draw. Shoot-through problems occur because transistors (especially FET’s) turn on faster than they turn off and also because high-side and low-side power converter transistors are typically switched in a complimentary fashion. Although the duration of the shoot-through current path is finite during PWM cycling, (i.e. the transistor will eventually turn off), even brief periods of a short circuit condition can produce excessive heating and stress the power converter and power supply.

Two basic approaches exist for controlling shoot-through: modify the transistors, or modify the PWM gate signals controlling the transistors. In the first case, the switch-on time of the transistor gate must be increased so that it (slightly) exceeds the switch-off time.

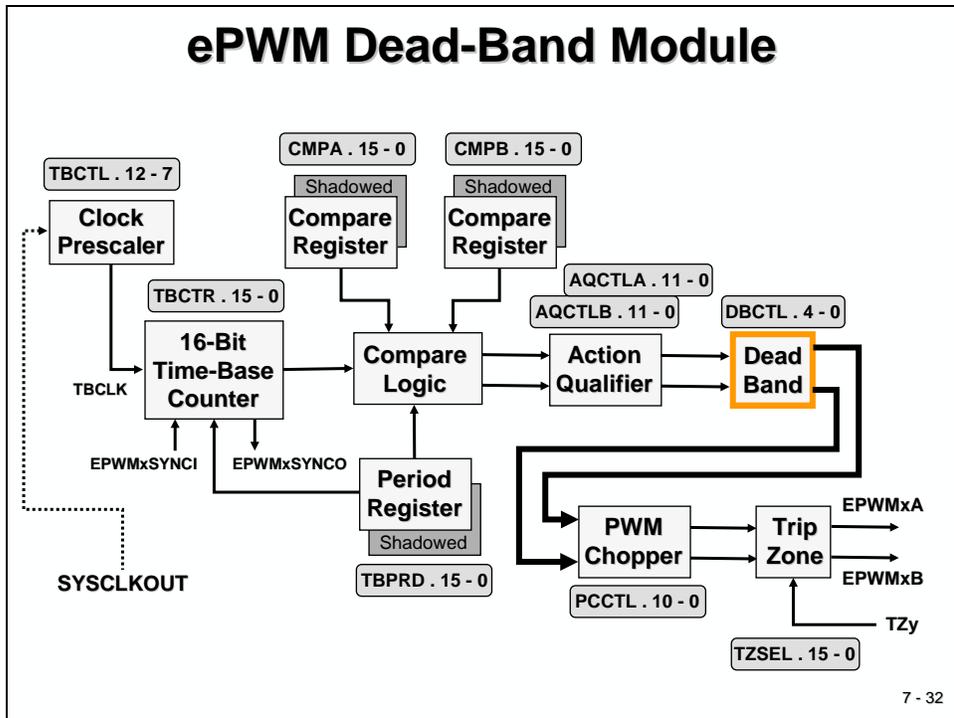
The hard way to accomplish this is by adding a cluster of passive components such as resistors and diodes in series with the transistor gate to act as low-pass filter to implement the delay.

The second approach to shoot-through control separates transitions on complimentary PWM signals with a fixed period of time. This is called dead-band. While it is possible to perform software implementation of dead-band, the F2833x offers on-chip hardware for this purpose that requires no additional CPU overhead. Compared to the passive approach, dead-band offers more precise control of gate timing requirements.

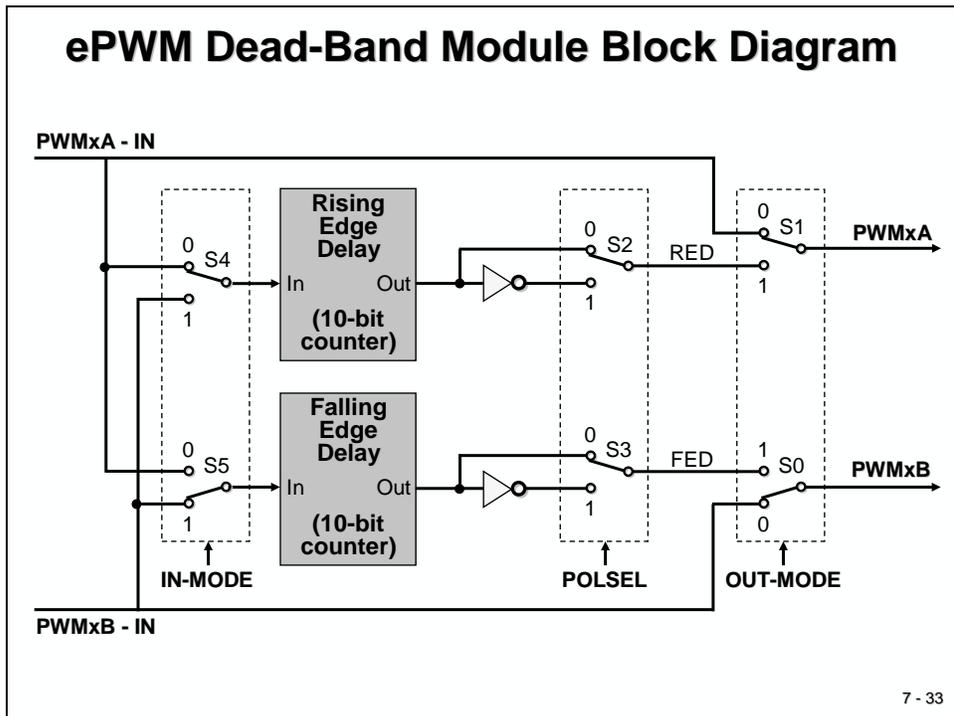


Hardware Dead Band Unit

All ePWM modules of the F2833x feature a hardware dead band unit.



The block diagram shows the different options available for this module:



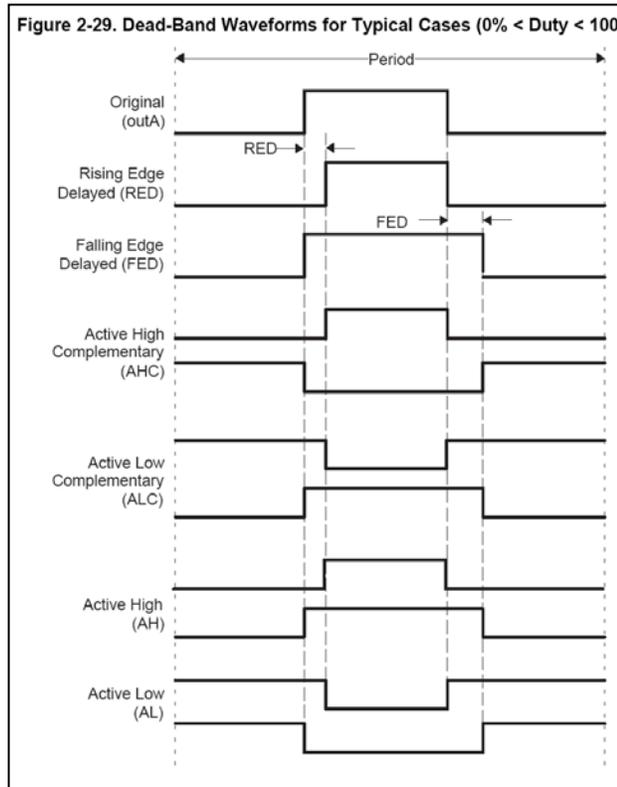
The setup of the dead-band unit is based on six switches, S0 to S5.

Although all combinations are supported, not all modes would be used in practice. The more classical modes assume that S4=0 and S5=0 [IN_MODE] is configured such that “EPWMxA-IN” is the source for both the falling-edge and rising-edge delay. Enhanced or non-traditional modes can be achieved by changing the input signal source.

Mode	Mode Description ⁽¹⁾	DBCTL[POLSEL]		DBCTL[OUT_MODE]	
		S3	S2	S1	S0
1	EPWMxA and EPWMxB Passed Through (No Delay)	X	X	0	0
2	Active High Complementary (AHC)	1	0	1	1
3	Active Low Complementary (ALC)	0	1	1	1
4	Active High (AH)	0	0	1	1
5	Active Low (AL)	1	1	1	1
6	EPWMxA Out = EPWMxA In (No Delay) EPWMxB Out = EPWMxA In with Falling Edge Delay	0 or 1	0 or 1	0	1
7	EPWMxA Out = EPWMxA In with Rising Edge Delay EPWMxB Out = EPWMxB In with No Delay	0 or 1	0 or 1	1	0

⁽¹⁾ These are classical dead-band modes and assume that DBCTL[IN_MODE] = 0,0. That is, EPWMxA in is the source for both the falling-edge and rising-edge delays. Enhanced, non-traditional modes can be achieved by changing the IN_MODE configuration.

The corresponding pulse sequences are:



Operating mode “Active High Complementary” (AHC) is the desired one for a pair of power switches in one phase of a 3-phase motor control system.

Dead Band Unit Registers

ePWM Dead-Band Module Registers

Name	Description	Structure
DBCTL	Dead-Band Control	EPwmXRegs.DBCTL.all =
DBRED	10-bit Rising Edge Delay	EPwmXRegs.DBRED =
DBFED	10-bit Falling Edge Delay	EPwmXRegs.DBFED =

Rising Edge Delay = $T_{TBCLK} \times DBRED$

Falling Edge Delay = $T_{TBCLK} \times DBFED$

7 - 34

The Dead Band Control Register combines the bit fields for switches S0 to S5:

ePWM Dead Band Control Register

Polarity Select

00 = active high
 01 = active low complementary (RED)
 10 = active high complementary (FED)
 11 = active low

15 - 6 reserved	5 - 4 IN_MODE	3 - 2 POLSEL	1 - 0 OUT_MODE
--------------------	------------------	-----------------	-------------------

In-Mode Control

00 = PWMxA is source for RED and FED
 01 = PWMxA is source for FED
 PWMxB is source for RED
 10 = PWMxA is source for RED
 PWMxB is source for FED
 11 = PWMxB is source for RED and FED

Out-Mode Control

00 = disabled (DBM bypass)
 01 = PWMxA = no delay
 PWMxB = FED
 10 = PWMxA = RED
 PWMxB = no delay
 11 = RED & FED (DBM fully enabled)

7 - 35

Lab 7_6: Dead Band Unit on ePWM1A / 1B

Objective

The objective of this lab is to introduce a delay time for rising edges in a pair of complementary PWM signals at ePWM1A and ePWM1B. The desired operating mode is “Active High Complementary” (AHC) and the two output signals are generated from input signal ePWM1A - in from the action qualifier unit.

Lab 7_6: Dead Band Unit for ePWM1A and ePWM1B

Objective:

- Add a delay time for rising edges on a pair of complementary signals ePWM1A and ePWM1B
- Active High Complementary (AHC) Mode
- Input signal to Dead-Band Unit is ePWM1A
- Dead Band Unit will generate ePWM1A and ePWM1B
- Use Lab7_4 as starting point

- New Registers involved:
 - DBRED: Dead Band Unit Rising Edge Delay
 - DBFED: Dead Band Unit Falling Edge Delay
 - DBCTL: Dead Band Unit Control Register

7 - 36

Procedure

Open Project File

1. If not still open from Lab7_5, re-open project **Lab7.pjt** in the “C/C++” perspective of Code Composer Studio.
2. Open file “Lab7_4.c” and save it as “Lab7_6.c”
3. Exclude file “Lab7_5.c” from build. Use a right mouse click at file “Lab7_5.c”, and enable “Exclude File(s) from Build”.
4. In file “Lab7_6.c” edit the function “cpu_timer0_isr()”. Remove all instructions to change the pulse width by register CMPA. We will use a fixed pulse width of 50% for this exercise, both for ePWM1A and ePWM1B.
5. In the function “Setup_ePWM1()”, initialize the pulse width to 50% of TBPRD:

EPwm1Regs.CMPA.half.CMPA = EPwm1Regs.TBPRD / 2;

6. Next, in the function “Setup_ePWM1()”, remove the instruction to initialize register AQCTLB. When using the dead band unit, both output pulse sequences ePWM1A and ePWM1B are normally derived from a single input signal, usually from internal signal ePWM1A of the action qualifier module.
7. In the function “Setup_ePWM1()”, add lines to initialize the dead band unit. Delay times are calculated in multiples of TBCLK, which we calculated at the beginning of Lab7_1 directly from SYSCLKOUT with CLKDIV set to 1 and HSPCLKDIV set to 2. In case of the F28335ControlCard running at 150MHz, TBCLK equals to 13.33334 ns. In our example we will setup a delay time of 10 microseconds, just as an example.

EPwm1Regs.DBRED = 750;

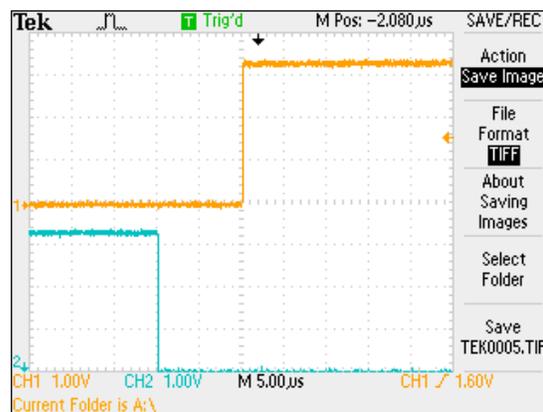
EPwm1Regs.DBFED = 750;

To initialize register DBCTL, we have to take into account switches S0 to S5 in Slide 7-33:

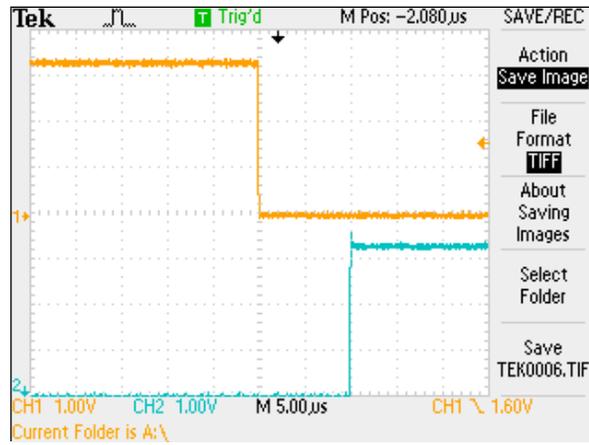
- Set S4 and S5 to 0: this way we will solely use input signal ePWM1A from unit AQCTL to generate the two output signals ePWM1A and ePWM1B.
- Set S2 = 0 and S3=1 to invert the polarity of signal ePWM1B against input ePWM1A.
- Set S0 = 1 and S1 = 1 to include a time delay for both switching points.

Build, Load and Test

8. Now build, load and test the modified project. A oscilloscope screenshot of signal ePWM1A and ePWM1B should look like this, when you trigger at the rising edge of channel 1 (ePWM1A):



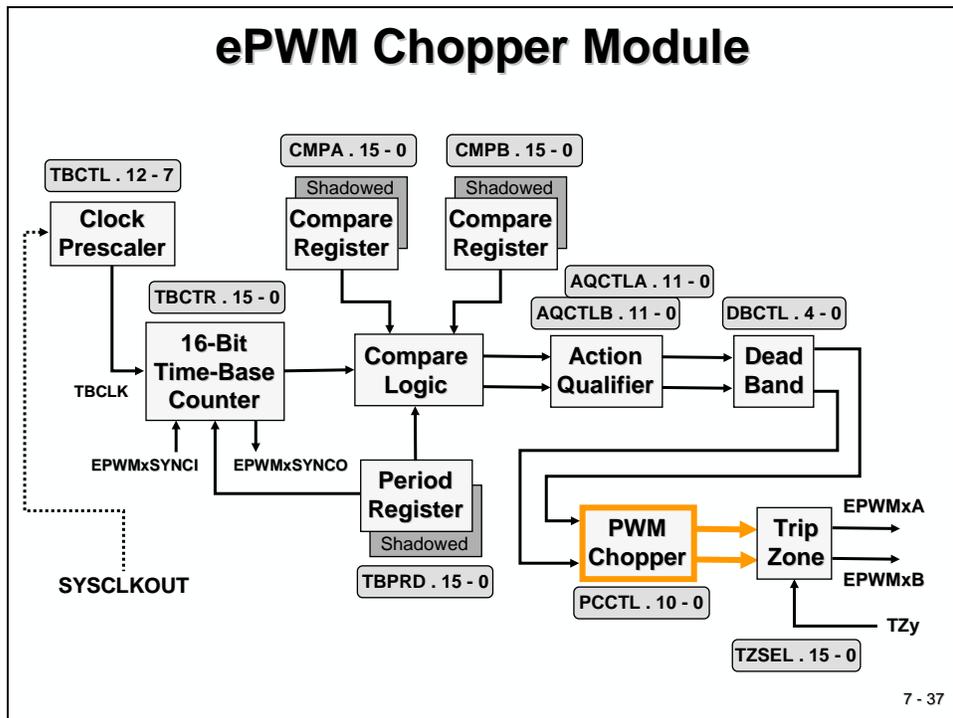
If you trigger at the falling edge of channel 1 (ePWM1A, yellow), again you should see a delayed rising edge, now at signal ePWM1B (blue):



END of LAB 7_6

ePWM Chopper Module

The PWM-chopper sub module allows a high-frequency carrier signal to modulate the PWM waveform generated by the action-qualifier and dead-band sub modules. This capability is important if you need pulse transformer-based gate drivers to control the power switching elements.



The key functions of the PWM-chopper sub module are:

- Programmable chopping (carrier) frequency
- Programmable pulse width of first pulse
- Programmable duty cycle of second and subsequent pulses
- Can be fully bypassed if not required

Purpose of Chopping

Purpose of the PWM Chopper Module

- ◆ **Allows a high frequency carrier signal to modulate the PWM waveform generated by the Action Qualifier and Dead-Band modules**
- ◆ **Used with pulse transformer-based gate drivers to control power switching elements**

7 - 38

The carrier clock of the ePWM Chopper Module is derived from SYSCLKOUT. The frequency and duty cycle of the chopper unit are controlled via the CHPFREQ and CHPDUTY bits in the PCCTL register.

The one-shot block is a feature that provides a high-energy first pulse to ensure hard and fast power switch turn on, while the subsequent pulses sustain pulses, ensuring the power switch remains on. The one-shot width is programmed via the OSHTWTH bits.

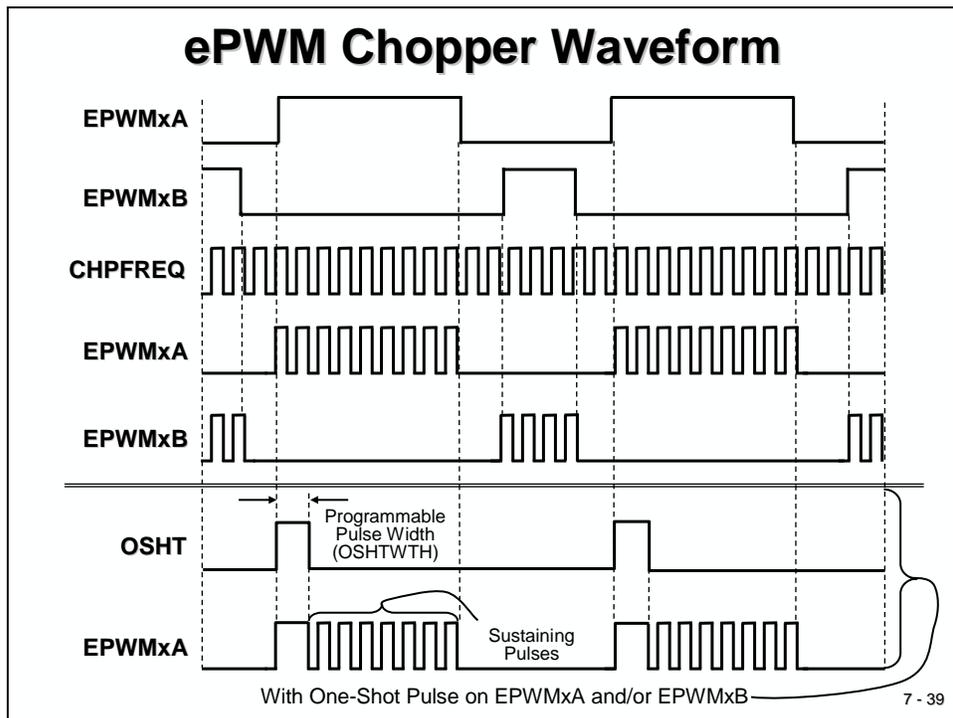
The PWM-chopper sub module can be fully disabled (bypassed) via the CHPEN bit.

Waveform Diagram of Chopped Signals

The top half of the following slide (Slide 7-39) shows the simplified waveforms of the chopping module action.

The bottom part of this slide shows a diagram of the special "one shot" mode, in which the duration of the first pulse can be programmed independently of all sustaining pulses of the chopper sequence.

Note: The duty-cycle control mode of the chopper module is not shown in the slide. This additional mode allows the setup of a different pulse width other than 50%.



The width of the first pulse can be programmed to any of 16 possible pulse width values. The width or period of the first pulse is given by:

$$T_{1stPULSE} = T_{SYSCLKOUT} * 8 * OSHTWTH$$

Where $T_{SYSCLKOUT}$ is the period of the system clock (SYSCLKOUT) and OSHTWTH is set by four control bits to a value between 1 and 16.

Chopper Mode Control Registers

ePWM Chopper Module Registers

Name	Description	Structure
PCCTL	PWM-Chopper Control	EPwm _x Regs.PCCTL.all =

7 - 40

ePWM Chopper Control Register

EPwm_xRegs.PCCTL

Chopper Clk Duty Cycle

000 = 1/8 (12.5%)
 001 = 2/8 (25.0%)
 010 = 3/8 (37.5%)
 011 = 4/8 (50.0%)
 100 = 5/8 (62.5%)
 101 = 6/8 (75.0%)
 110 = 7/8 (87.5%)
 111 = reserved

Chopper Clk Freq.

000 = SYSCLKOUT/8 + 1
 001 = SYSCLKOUT/8 + 2
 010 = SYSCLKOUT/8 + 3
 011 = SYSCLKOUT/8 + 4
 100 = SYSCLKOUT/8 + 5
 101 = SYSCLKOUT/8 + 6
 110 = SYSCLKOUT/8 + 7
 111 = SYSCLKOUT/8 + 8

Chopper Enable
 0 = disable (bypass)
 1 = enable

15 - 11	10 - 8	7 - 5	4 - 1	0
reserved	CHPDUTY	CHPFREQ	OSHTWTH	CHPEN

One-Shot Pulse Width

0000 = 8 / SYSCLKOUT	1000 = 72 / SYSCLKOUT
0001 = 16 / SYSCLKOUT	1001 = 80 / SYSCLKOUT
0010 = 24 / SYSCLKOUT	1010 = 88 / SYSCLKOUT
0011 = 32 / SYSCLKOUT	1011 = 96 / SYSCLKOUT
0100 = 40 / SYSCLKOUT	1100 = 104 / SYSCLKOUT
0101 = 48 / SYSCLKOUT	1101 = 112 / SYSCLKOUT
0110 = 56 / SYSCLKOUT	1110 = 120 / SYSCLKOUT
0111 = 64 / SYSCLKOUT	1111 = 128 / SYSCLKOUT

7 - 41

Lab 7_7: Chopped Signals at ePWM1A / 1B

Objective

We will add a chopper frequency modulation to the software developed in Chapter 7. In Lab7_5 we controlled the pulse width of ePWM1A by register CMPA independently of ePWM1B, which was controlled by CMPB. The objective now is to chop the active phase of the pulses at ePWM1A and ePWM1B with a higher frequency.

Lab 7_7: Chopper Mode Signals add ePWM1A and ePWM1B

Objective:

- The pair of complementary signals ePWM1A and ePWM1B will be modulated by a chopper frequency of 2.344 MHz
- Chopper Mode Duty Cycle = 50%
- One shot pulse width = 800 ns
- Use Lab7_5 as starting point



7 - 42

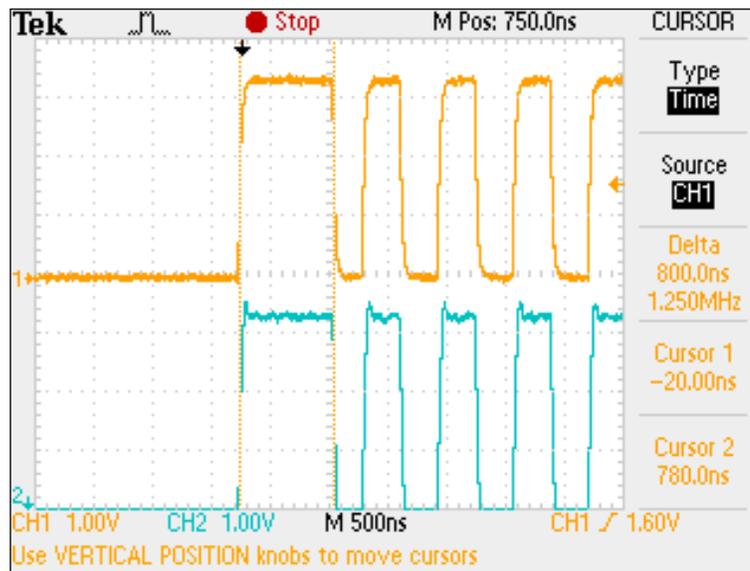
Procedure

Open Project File

1. In project "Lab7" open file "Lab7_5.c" and save it as "Lab7_7.c"
2. Exclude "Lab7_6.c" from build.
3. In the function "Setup_ePWM1()", initialize the chopper module. Remember that SYSCLKOUT has been set to 150 MHz (assuming an external clock of 30 MHz at the F28335ControlCard). In register "EPwm1Regs.PCCTL":
 - Set chopper frequency to 2.34375 MHz (SYSCLKOUT / 64).
 - Set chopper duty cycle to 50%
 - Set one shot pulse to 800 ns
 - Enable the chopper unit.

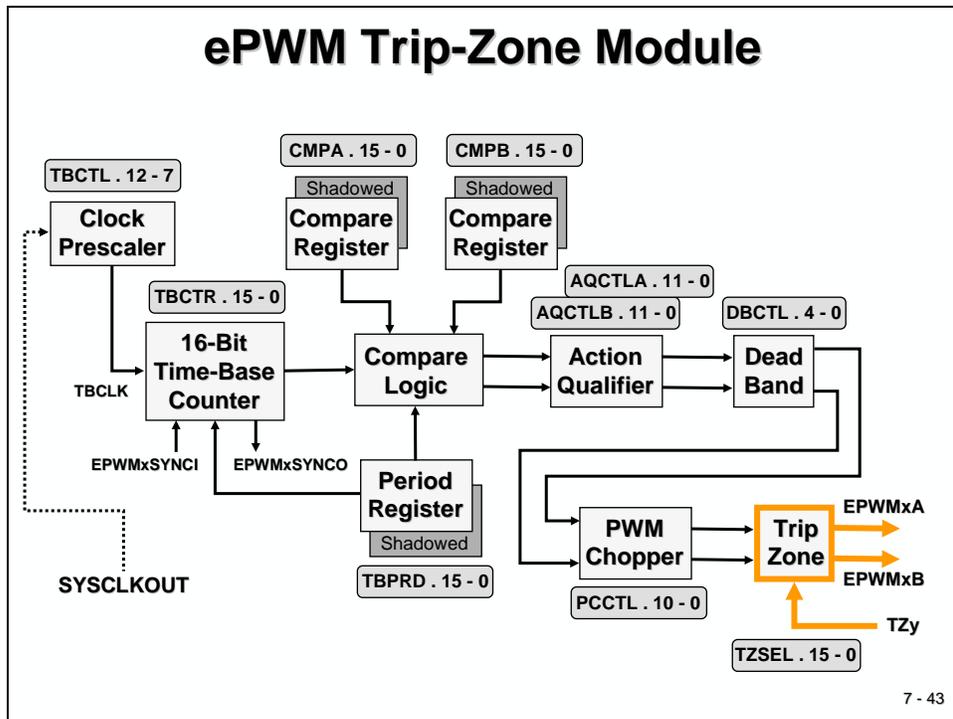
Build, Load and Test

- Build, load and test the modified project. A oscilloscope screenshot of signal ePWM1A and ePWM1B should look like the following graph, when you trigger at the rising edge of channel 1 (ePWM1A):



ePWM Over Current Protection

Each ePWM module is connected to six Trip - Zone signals (TZ1 to TZ6) that are sourced from the GPIO MUX. These signals indicate external fault or trip conditions, and the ePWM outputs can be programmed to respond accordingly when faults occur.

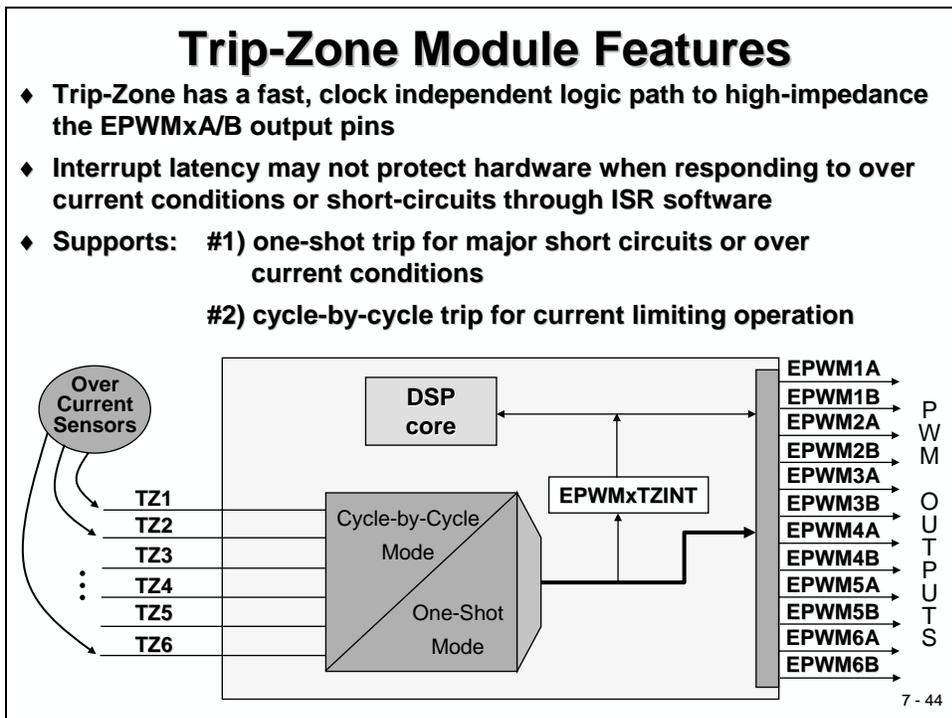


Purpose of the Trip-Zone Submodule

Trip Zone signals are usually generated by over-current sensors, which set a signal if a threshold is passed. The key functions of the Trip-Zone sub module are:

- Trip inputs TZ1 to TZ6 can be flexibly mapped to any ePWM module.
- Upon a fault condition, outputs EPWMxA and EPWMxB can be forced to one of the following:
 - High
 - Low
 - High-impedance
 - No action taken
- One-shot trip (OSHT) mode to support major short circuits or over-current conditions.
- Support for cycle-by-cycle tripping (CBC) for current limiting operation.
- Each trip-zone input pin can be allocated to either one-shot or cycle-by-cycle operation.
- Interrupt generation is possible on any trip-zone pin.
- Software-forced tripping is also supported.

- The trip-zone sub module can be fully bypassed if it is not required.



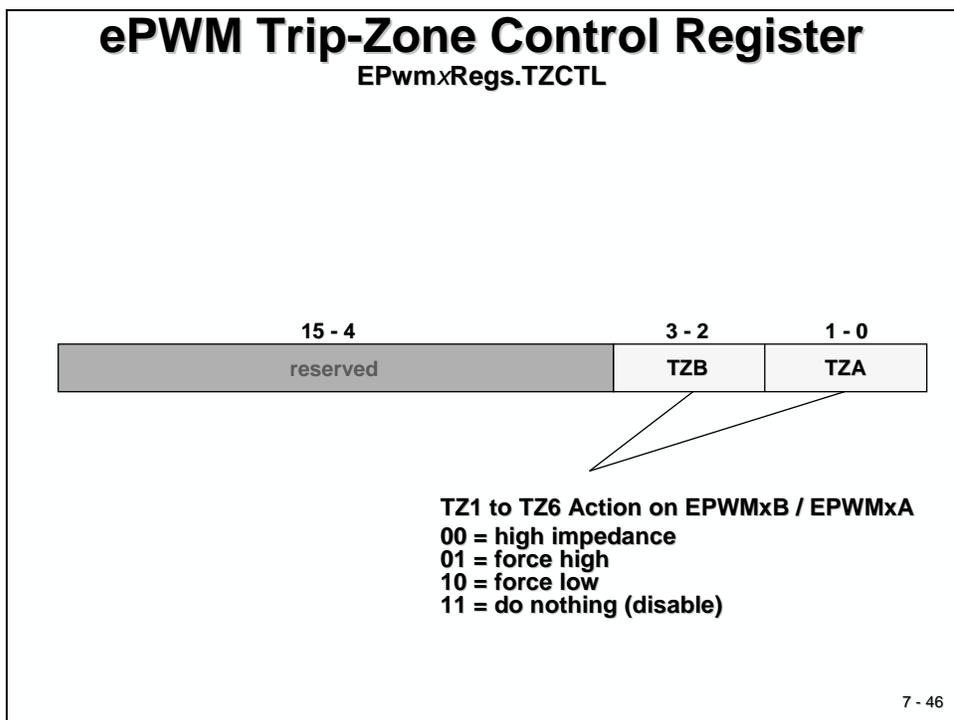
ePWM Trip - Zone Registers

ePWM Trip-Zone Module Registers

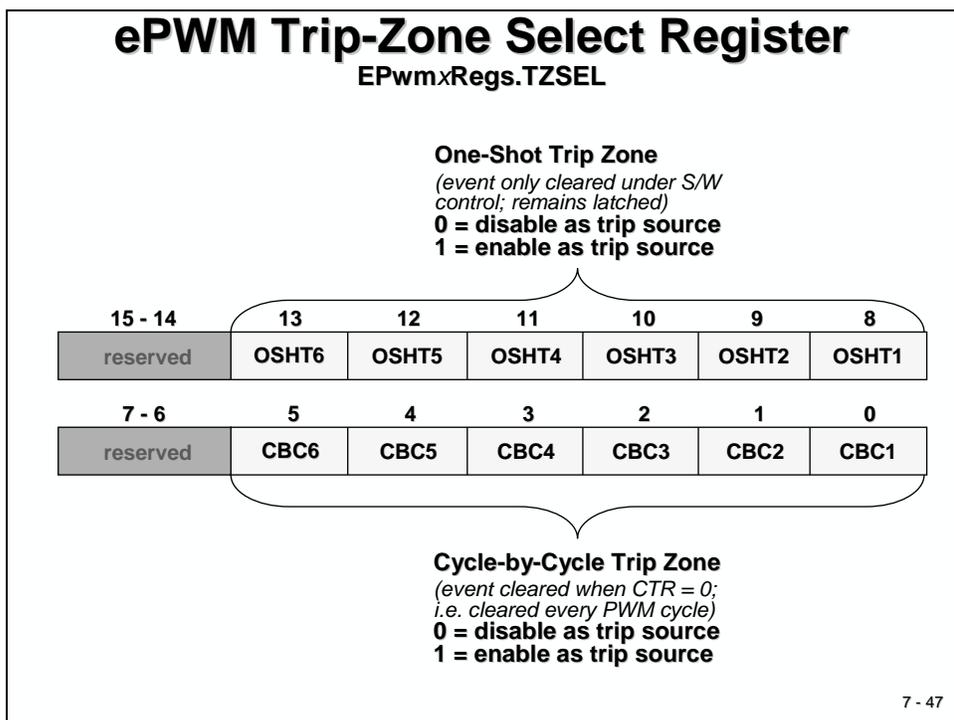
Name	Description	Structure
TZCTL	Trip-Zone Control	EPwmxRegs.TZCTL.all =
TZSEL	Trip-Zone Select	EPwmxRegs.TZSEL.all =
TZEINT	Enable Interrupt	EPwmxRegs.TZEINT.all =
TZFLG	Trip-Zone Flag	EPwmxRegs.TZFLG.all =
TZCLR	Trip-Zone Clear	EPwmxRegs.TZCLR.all =
TZFRC	Trip-Zone Force	EPwmxRegs.TZFRC.all =

7 - 45

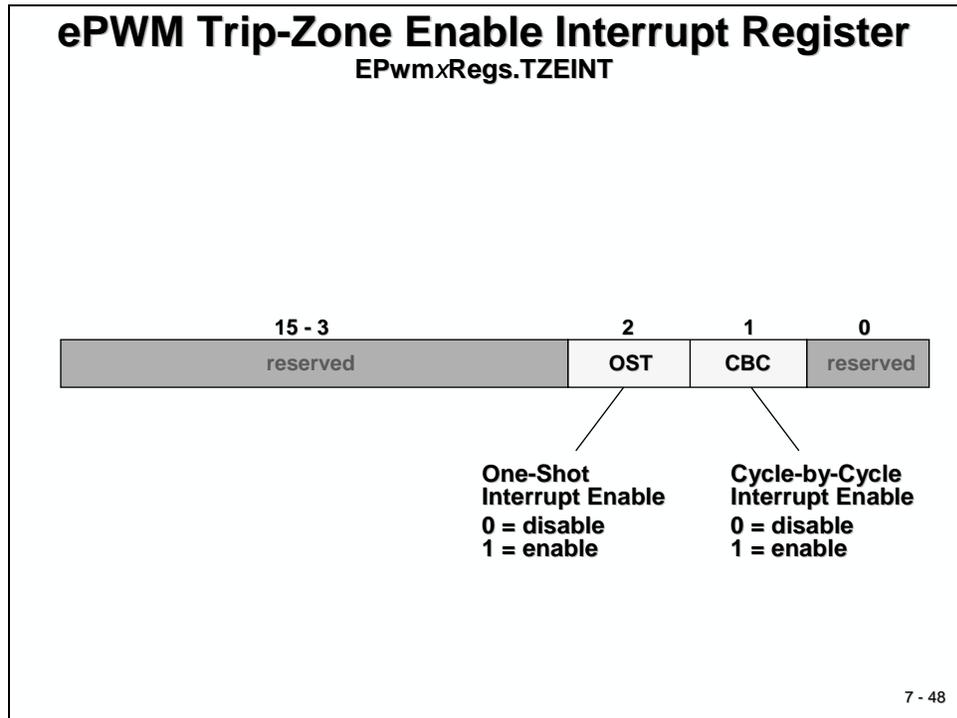
Note: Trip Zone Registers are protected! When you initialize these registers, you must EALLOW the access, before you can change the values. After you are done, close the protection again with an EDIS instruction!



Register TZCTL is used to define the state of line ePWMxA and ePWMxB in case of an over current signal.



With register TZSEL, we can specify which input signal TZx should be used as a cycle-by-cycle or as a permanent (one shot) switch off signal.



Register TZEINT can be used to request an interrupt service request in case of an over current situation in a closed loop control system. We can use either a cycle - by-cycle or a one-shot over current interrupt request, depending on the selection in register TZSEL.

What should be done in such an interrupt event? Well, this depends on the application and on the seriousness of the fault.

Lab 7_8: Trip Zone protection with TZ6

Objective

Again we will start with file "lab7_5.c". Trip Zone signal "/TZ6" is multiplexed with input signal GPIO17, which on the Peripheral Explorer Board is connected to push button PB1. So a very simple setup is to use this button to "simulate" an over current signal. When we push this button, we can produce an active signal TZ6. The objective is to force both ePWM1A and ePWM1B permanently to low in case of this button is pushed.

Lab 7_8: Over Current Protection with Trip Zone Signals TZx

Objective:

- Trip Zone Signal TZ6 is connected to GPIO17, push – button PB1 at Peripheral Explorer Board
- Active Signal PB1 will force ePWM1A and ePWM1B to low
- Use Lab7_5 as starting point

- New registers in this lab:
 - TZCTL: Trip Zone Control
 - TZSEL: Trip Zone Select
 - TZEINT: Trip Zone Enable Interrupt
 - TZCLR: Trip Zone Clear Interrupt Flags

7 - 49

Procedure

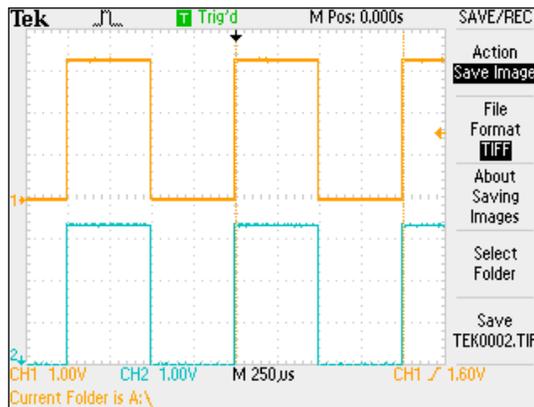
Open Project File

1. In project "Lab7", open file "Lab7_5.c" and save it as "Lab7_8.c"
2. Exclude the file "Lab7_7.c" from build.
3. In the function "Gpio_select()", set multiplex register GPAMUX2 to use /TZ6 for GPIO17.
4. The in the function "Setup_ePWM1()", initialize the trip zone registers.

- In the register "EPwm1Regs.TZCTL", set TZA and TZB to force ePWM1A and ePWM1B to zero in case of an active TZ6.
- In the register "EPwm1Regs.TZSEL", select TZ6 as source for a one shot over current signal. In the event of an active TZ6 (when we push button PB1), both lines ePWM1A and ePWM1B will be switched off permanently.
- Remember that both registers are EALLOW - protected, so please do not forget to open / close the access to these registers.

Build, Load and Test

5. Build, load and test the modified project. A oscilloscope screenshot of signal ePWM1A and ePWM1B should show the desired pattern at ePWM1A an ePWM1B:



6. Now push button PB1. Both ePWM1A and ePWM1B should be switched off (0V) permanently.

One Shot Mode

7. Now let us modify the code in such a way, that an active button PB1 (our trip zone TZ6) will request a cycle-by-cycle switch off of the two signals ePWM1A and ePWM1B.
 - In the function "Setup_ePWM1()", change register "EPwm1Regs.TZSEL" so that TZ6 will now be the source for a cycle-by-cycle over current signal, and no longer for a one-shot procedure.

Re-Build, Load and Test

8. Build, load and test the modified project. Please do not forget to reset the DSC before you perform a new test. This is always a good practice, since the chip will always start from a known state! Here once more is the required sequence:
 - Debug → Reset CPU
 - Debug → Restart
 - Debug → Go Main

- **Debug → Run**

The scope should again show the pulse sequences at ePWM1A and ePWM1B.

When you push PB1, the signals should fade out to ground and keep this ground voltage, as long as you keep your finger on PB1 to hold it down. But, when you release PB1, the pulse pattern at ePWM1A and ePWM1B should reappear again. That's why we this time initialized the F2833x to resume the PWM operation on a cycle-by-cycle basis!

Add an Interrupt Service

Although we do not have a real power stage system and just the Peripheral Explorer Board, it still allows us also to perform an exercise with an interrupt service in the event of an over current situation.

9. At the beginning of "Lab7_8.c", add a prototype for an interrupt service routine:

```
interrupt void ePWM1_TZ_isr(void);
```

10. In "main()", look for the line, in which we change the entry in PieVectTable for TINT0. After this line, add a new line to replace the entry for EPWM1_TZINT:

```
PieVectTable.EPWM1_TZINT = &ePWM1_TZ_isr;
```

11. Interrupt EPWM1_TZINT is wired to PIE - interrupt line INT2 bit 1. We have to enable this line. In "main()", search for the line, where we enabled PIEIER1.bit.INTx7. Add a new line to also enable interrupt 2.1:

```
PieCtrlRegs.PIEIER2.bit.INTx1 = 1;
```

12. Change the line "IER |= 1;" so that the two lines INT1 and INT2 are enabled:

```
IER |= 3;
```

13. In the function "Setup_ePWM1()", add a line to enable cycle-by-cycle interrupts in register EPwm1Regs.TZEINT. Include this new instruction in the EALLOW - EDIS block!

14. At the end of "Lab7_8.c", add the definition for function "ePWM1_TZ_isr()". In this function include the following actions:

- Clear the two flags "CBC" and "INT" in register "EPwm1Regs.TZCLR" to re-enable TZ6 for the next interrupt service:

```
EPwm1Regs.TZCLR.bit.CBC = 1;
```

```
EPwm1Regs.TZCLR.bit.INT = 1;
```

Recall that this register is EALLOW - protected!

- Now, because we "simulate" our over current signal TZ6 using a push button, the duration of the "over-current" signal depends on how fast we can take our finger off the button. So what happens, if we push it too long? Answer: TZ6 will trigger a next interrupt immediately after we return from interrupt function "ePWM1_TZ_isr()".

Remember that we have three different software activities in Lab7_8:

- “main()” - loop, where we execute the watchdog service #1;
- interrupt service "cpu_timer0_isr()", where we execute the watchdog service #2;
- new interrupt service "ePWM1_TZ_isr()".

Because the interrupt service "cpu_timer0_isr()" has a higher priority than "ePWM1_TZ_isr()", it will interleave with our finger triggered series of interrupt requests. The problem is, that the “main()”-loop, and consequently our watchdog service #1, will be locked out - as long as we keep pushing button PB1.

Solution: push quickly! Or, if you like to push slowly, include the watchdog service #1 into the new interrupt service function "ePWM1_TZ_isr()":

SysCtrlRegs.WDKEY = 0x55;

Remember that this register is also EALLOW - protected!

- To indicate, that we are executing code from the new interrupt service routine "ePWM1_TZ_isr", add a line to toggle LED GPIO9:

GpioDataRegs.GPATOGGLE.bit.GPIO9 = 1;

- To acknowledge that we are done with the interrupt service, in PIE group 2, add:

PieCtrlRegs.PIEACK.all = 2;

15. In the while(1) - loop of “main()”, remove the code for the binary counter at GPIO9, GPIO11, GPIO34 and GPIO49. Because we will use GPIO9 as an indicator for the new interrupt service function "ePWM1_TZ_isr()", we cannot use that old block of code any more. Optionally, you can add a toggle instruction for GPIO11 to the second interrupt service function "cpu_timer0_isr()".
16. In “main()”, change the line to setup CPU - Timer 0 back to a period of 100 milliseconds:

ConfigCpuTimer(&CpuTimer0,100,100000);

Re-Build, Load and Test

17. Build, load and test the modified project. Please do not forget to reset the device before you perform a new test. This is always a good practice, since the chip will always start from a known state! Here's ones more the sequence:
 - **Debug → Reset CPU**
 - **Debug → Restart**
 - **Debug → Go Main**
 - **Debug → Run**

The scope should again show the pulse sequences at ePWM1A and ePWM1B.

When you push PB1 the signals should fade out to ground and keep this ground voltage, as long as you keep your finger on PB1 to hold it down. When you release PB1, the pulse pattern at ePWM1A and ePWM1B should reappear again.

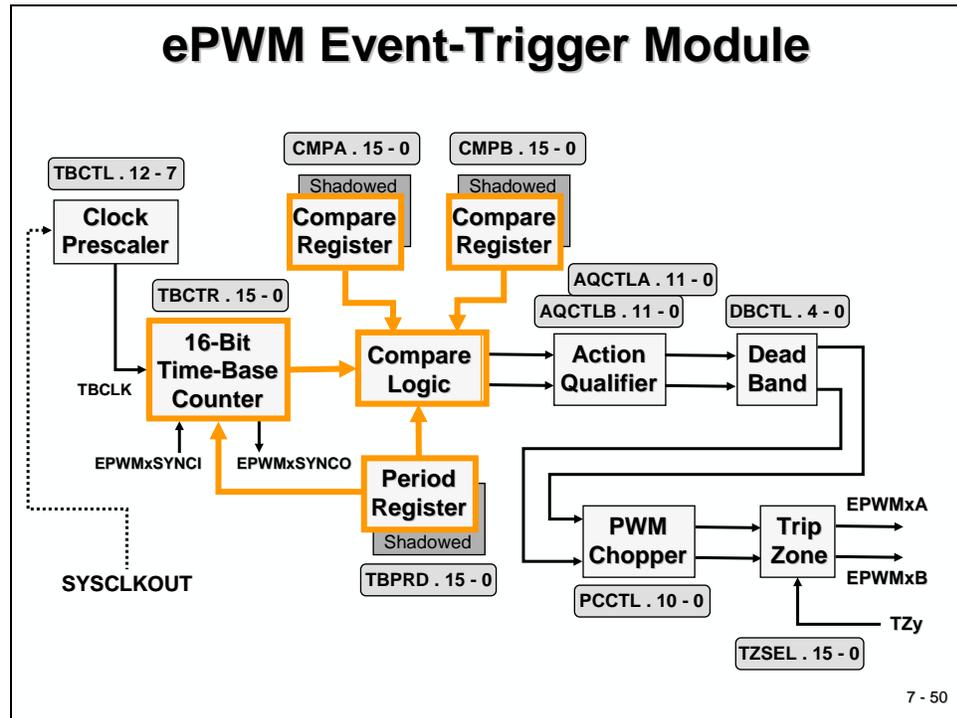
LED LD2 (GPIO11) should toggle with a period of 100 milliseconds.

Each time you push PB1, LED LD1 (GPIO9) should toggle, as an indication of the execution of the over current interrupt service routine "ePWM1_TZ_isr()". Please note that button PB1 is a switch with bouncing contacts, so it might request more than one interrupt, when you press it down.

Of course, in a real-world application, an over-current signal will never be generated by a push button; we would use a real sensor unit to measure the current in a power stage! Nevertheless, this exercise with the limited features of the Peripheral Explorer Board includes all the software features of such a real-world application.

END of Lab7_8

ePWM Interrupt Sources



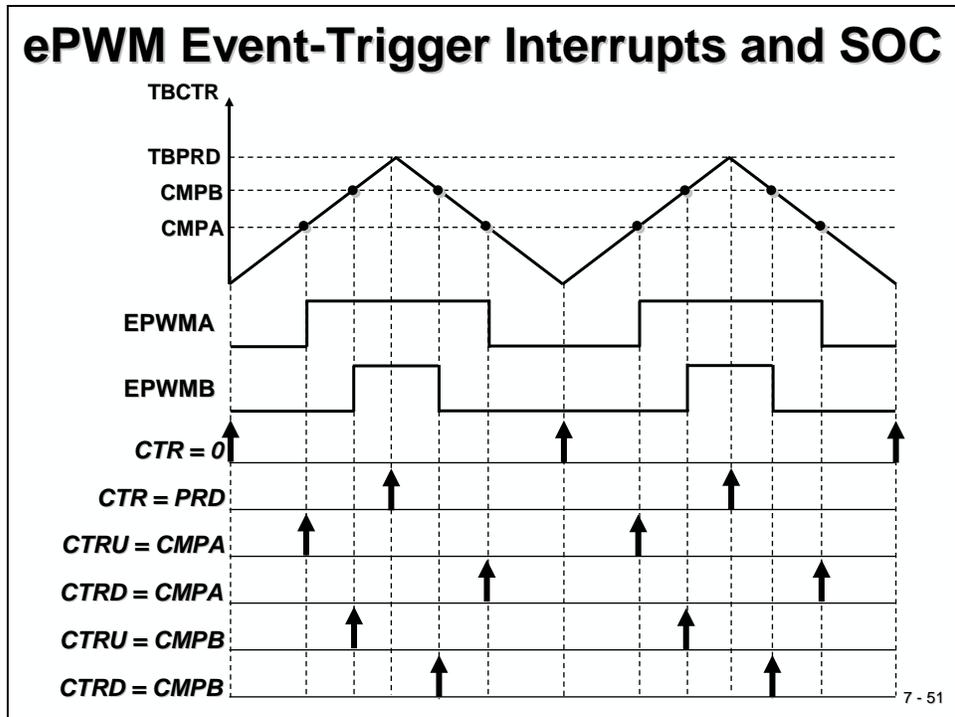
We still have left one module of the ePWM unit: the **event trigger sub module**. It monitors various event conditions, such as

- Counter value TBCTR = zero
- Counter value TBCTR = TBPRD
- Counter value TBCTR = CMPA
- Counter value TBCTR = CMPB

and can be configured to prescale these events before issuing an Interrupt request or an ADC start of conversion. The event-trigger prescaling logic can issue Interrupt requests and ADC start of conversion at:

- Every event
- Every second event
- Every third event

The next slide is an example for symmetrical PWM operation mode and shows available point of actions for interrupt service requests or to start an analogue to digital conversion:



The Event-Trigger- Sub module is initialized by a set of registers:

- ETSEL - This register selects which of the possible events will trigger an interrupt or start an ADC conversion
- ETPS - This register programs the event prescaling options mentioned above.
- ETFLG - Register with flag bits to indicate the status of the selected and prescaled events.
- ETCLR - These bits allow you to clear the flag bits in the ETFLG register via software.
- ETFRC - These bits allow software forcing of an event. Useful for debugging or s/w intervention.

We will use one of the interrupts of the event trigger module in the next lab exercise Lab7_9 to request a change of the pulse width on a cycle by cycle base (or "on the fly") to generate a sine wave modulated signal at ePWM1A.

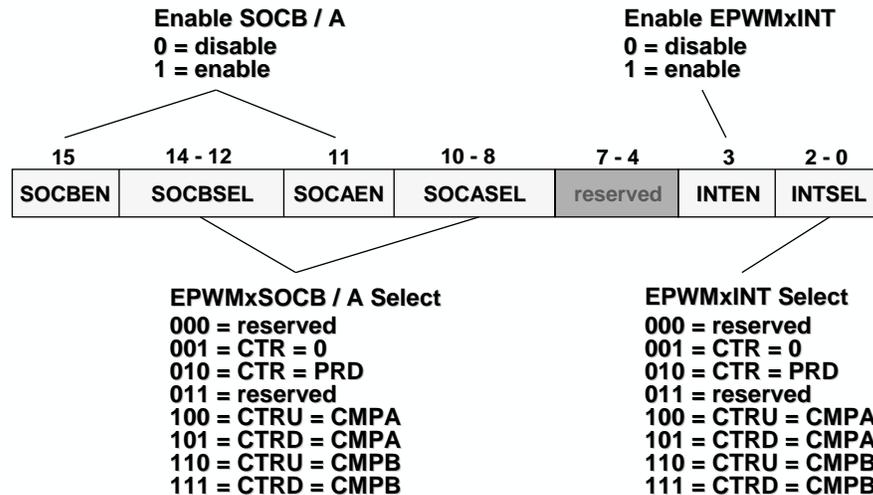
ePWM Event-Trigger Module Registers

Name	Description	Structure
ETSEL	Event-Trigger Selection	EPwm _x Regs.ETSEL.all =
ETPS	Event-Trigger Pre-Scale	EPwm _x Regs.ETPS.all =
ETFLG	Event-Trigger Flag	EPwm _x Regs.ETFLG.all =
ETCLR	Event-Trigger Clear	EPwm _x Regs.ETCLR.all =
ETFRC	Event-Trigger Force	EPwm _x Regs.ETFRC.all =

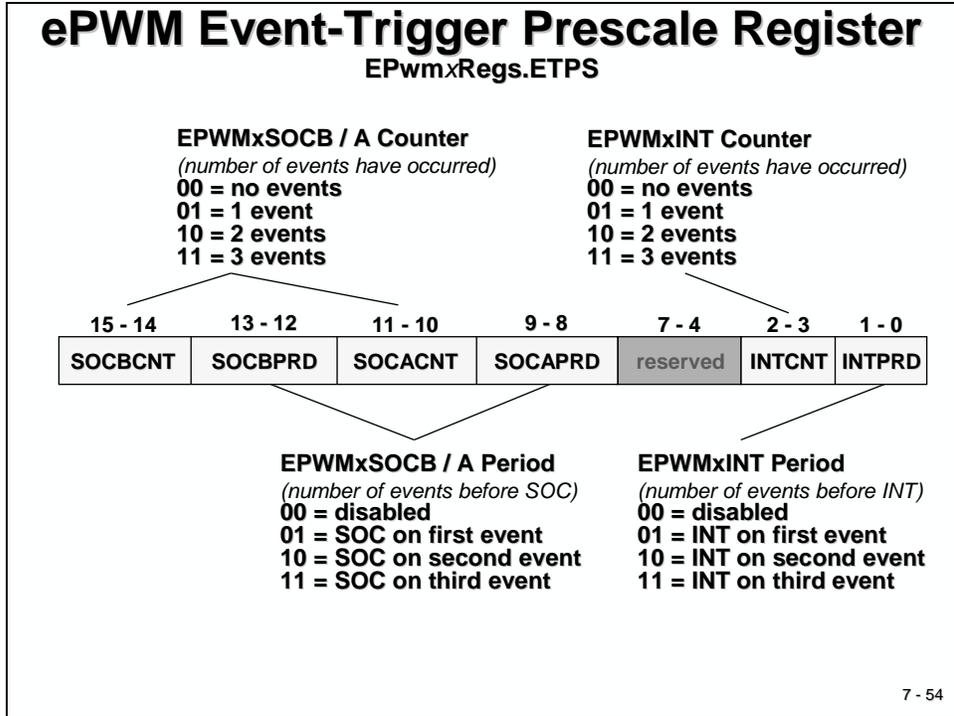
7 - 52

ePWM Event-Trigger Selection Register

EPwm_xRegs.ETSEL



7 - 53



Lab7_9: ePWM Sine Wave Modulation

Objective

The F28335ControlCARD is used in combination with the Peripheral Explorer Board to output a sine wave signal at ePWM1A. Channel ePWM1A is set up in standard 16-bit resolution. The generated signal is connected to a second order low pass filter with a cut-off frequency of 105 kHz. The filter output signal can be monitored at header J11-1 (“DAC-1”) of the Peripheral Explorer Board.

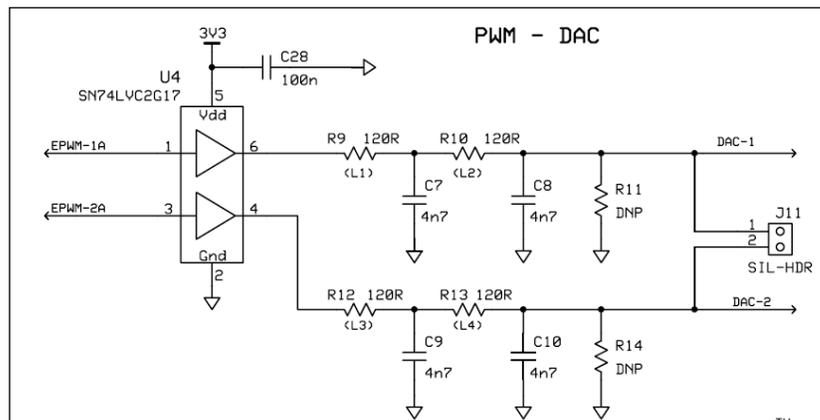
Lab 7_9: Sine Wave PWM signal at ePWM1A

Objective:

- Generate a sine wave modulated pulse sequence at ePWM1A
- ePWM1A carrier frequency is 500 KHz
- Sine wave frequency is 976 Hz

7 - 55

Channel ePWM1A is set up for a 500 kHz PWM frequency, ePWM1 compare down event triggers an interrupt service routine (ISR), according to the frequency the trigger appears every 2000 ns.



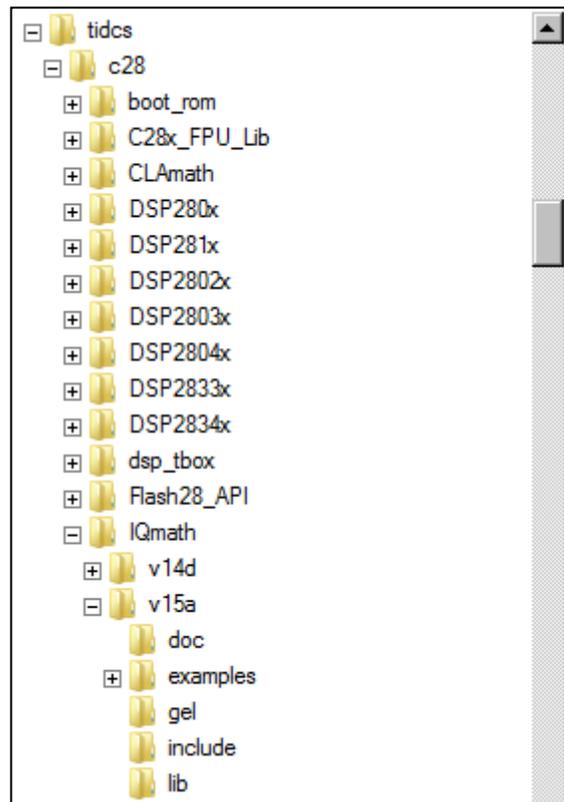
The ISR with a code execution time of 630ns takes advantage of the Boot-ROM sine wave lookup-table to calculate the next compare value for the next ePWM1A period. The lookup-table consists of 512 values in I2Q30-format and is located at address 0x3FE000. Every ISR call is used to read the next entry of this table, thus a full period of the resulting sine wave takes $512 * 2000 \text{ ns} = 1024 \mu\text{s}$. The synthesized sine wave signal has a frequency of $1/1024\mu\text{s} = 976 \text{ Hz}$. Due to the type of look-up values in I2Q30-format, functions of a library called "IQmath" are used to calculate the next value for the duty cycle.

Although we have not discussed the background of fixed-point binary mathematics and especially of Texas Instruments IQMath yet, we will use this library in a 'black box' method. We will resume the discussion of this mathematical approach in a later chapter of this teaching course.

Procedure

Install IQMath

If not already installed on your PC, you will have to install the IQMath library now. The standard installation path is "C:\tidcs\c28\IQmath":



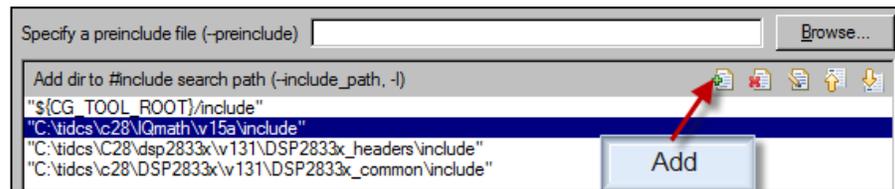
If this library isn't available on your PC, you will have to install it first. If you are in a classroom and you do not have administrator installation rights, ask your teacher for assistance. You can find the installation file under number "sprc087.zip" in the utility part of this CD-ROM or at the Texas Instruments Website (www.ti.com).

Open Project File

1. In project "Lab7" open the file "Lab7_8.c" and save it as "Lab7_9.c"
2. Exclude the file "Lab7_8.c" from build.
3. Change the Build options.

We have to extend the preprocessors include search path. In the "C/C++" perspective, in the project window right click at project "Lab7" and open "Properties". In the "C/C++ Build" category, open "Include Options:" and add a new entry:

C:\tidcs\c28\IQmath\v15a\include



Close the "C/C++ Build" options menu with <OK>

4. Link the IQmath library to your project. Right click at project "Lab7" and select function "Link Files to Project. Link:

C:\tidcs\c28\IQmath\v15a\lib\IQmath_fpu32.lib

5. At the beginning of "Lab7_9.c" include the header file for IQmath:

```
#include "IQmathLib.h"
```

Also at the beginning of "Lab7_9.c", add a new global variable "sine_table[512]" of data type "_iq30" to "Lab7_9.c":

```
#pragma DATA_SECTION(sine_table, "IQmathTables");  
_iq30 sine_table[512];
```

The pragma statement is a directive for the compiler to generate a new data section for "sine_table". The linker command file "28335_RAM_lnk.cmd", which is already part of our project, will connect the section "IQmathTables" to physical address 0x3FE000, which is where our lookup table is stored in ROM.

6. In "Lab7_9.c" remove everything that is related to CpuTimer0, including external function prototypes, the call to functions "InitCpuTimers()", "ConfigCpuTimer()" and Interrupt Service Routine "cpu_timer0_isr()", including its prototype and definition. In the while(1) loop of main, also remove all instruction related to variable "CpuTimer0.InterruptCount".

Also remove everything that is related to variable "counter". We do not need this variable any more.

7. Also at the beginning of "Lab7_9.c", replace the function prototype of ISR "ePWM1_TZ_isr()" by a new interrupt service function prototype:

```
interrupt void ePWM1A_compare_isr(void);
```

8. In "main()", remove the entry instruction to write into "PieVectTable.EPWM1_TZINT" and add a new instruction:

PieVectTable.EPWM1_INT = &ePWM1A_compare_isr;

PWM1 interrupts are connected to PIE group 3, bit 1. Therefore change the line to enable PIE interrupts into:

PieCtrlRegs.PIEIER3.bit.INTx1 = 1;

Change register IER to allow interrupts at line 3:

IER |= 4;

9. In the while(1) - loop of main keep just the instruction to service the watchdog instruction #1 (value 0x55) to register WDKEY. Remember that the register WDKEY is EALLOW protected!
10. Next, in the function "Gpio_select()", just keep ePWM1A as PWM output signal. Remove the instructions to enable lines ePWM1B and TZ6.
11. In the function "Setup_ePWM1()", change the period of ePWM1 to 500 kHz. In up/down mode the value for TBPRD is calculated by:

$$TBPRD = \frac{1}{2} * \frac{f_{SYSCLKOUT}}{f_{PWM} * CLKDIV * HSPCLKDIV}$$

with CLKDIV and HSPCLKDIV both set to "divide by 1" and $f_{SYSCLKOUT} = 150\text{MHz}$, TBPRD should be initialized to 150.

12. Then in the function "Setup_ePWM1()", remove the initialization lines for registers CMPB and AQCTLB, since we will not generate a signal at ePWM1B.
13. At the end of the function "Setup_ePWM1()", remove the code to initialize the trip zone unit, including all instructions for registers TZCTL, TZSEL and TZEINT.
14. At the end of the function "Setup_ePWM1()", add code to initialize the Event Trigger module. In the register "ETSEL", enable bit "INTEN" and set the bit field "INTSEL" to select an interrupt request, if CTRD = CMPA (counter down matches CMPA). In the register "ETPS", set bit field "INTPRD" to request an interrupt on first event.
15. At the end of "Lab7_9.c" add the definition of function "ePWM1A_compare_isr()":

```
interrupt void ePWM1A_compare_isr(void)
{
```

First define a static variable "index" and initialize it to zero. This variable will be used as an index into lookup-table "sine_table[512]":

static unsigned int index = 0;

Next we have to service the second half of the watchdog - key sequence to register WDKEY (value 0xAA). Remember that this register is EALLOW protected!

Now we have to calculate a new value for register CMPA. Here is the line:

```

EPwm1Regs.CMPA.half.CMPA =
EPwm1Regs.TBPRD - IQsat(
  _IQ30mpy((sine_table[index]+_IQ30(0.9999))/2, EPwm1Regs.TBPRD),
EPwm1Regs.TBPRD,0);

```

Confusing, isn't it?

Here is an attempt to explain it, should you be interested in the details:

- Recall, the difference between TBPRD and CMPA defines the pulse width of the PWM signal. The bigger the difference, the bigger the pulse. It means that we have to subtract a percentage value from TBPRD to define the next pulse width and store this percent value in CMPA.
- To find that next value to be subtracted from TBPRD we have to access the sine table. Variable "index" points to this table, which consists of 512 entries for a unit circle of 360 degrees. The value taken from this table is in I2Q30-Format and between 0 for sin(0), 1 for sin(90°), 0 for sin(180°), -1 for sin(270°) and again 0 for sin(360°).
- So, we read a number between +1 and -1, which corresponds to the current amplitude of the sine. However, we cannot use a negative number for the calculation of a result between 0 and 100% of TBPRD. What we do is we add an offset of +1 in the form of an IQ-number (`_IQ30(0.9999)`) to obtain numbers between 0 and +2. Next we divide the result by 2 to scale it into a range between 0 and 1 (or 0% and 100%).
- Now we multiply this relative number (0 to 1) by TBPRD with a call of function `_IQ30mpy()`. If TBPRD has been set to 100, the result will be a number between 0 and 100.
- The function `_IQsat()` is a saturation function that will limit the first parameter (our result) between maximum (parameter 2, TBPRD) and minimum (parameter 3, zero). To call this function is just a precaution to avoid any calculation overflows, which could result in catastrophic output signals, where a large positive number suddenly becomes a large negative number.

After this calculation, still inside `"ePWM1A_compare_isr()"`, we have to increment variable "index" and to reset it, if we are at the end of the sine_table:

```

index +=1;
if( index > 511) index = 0;

```

Finally, we have to clear the interrupt flags of the event trigger module and the PIE-unit:

```

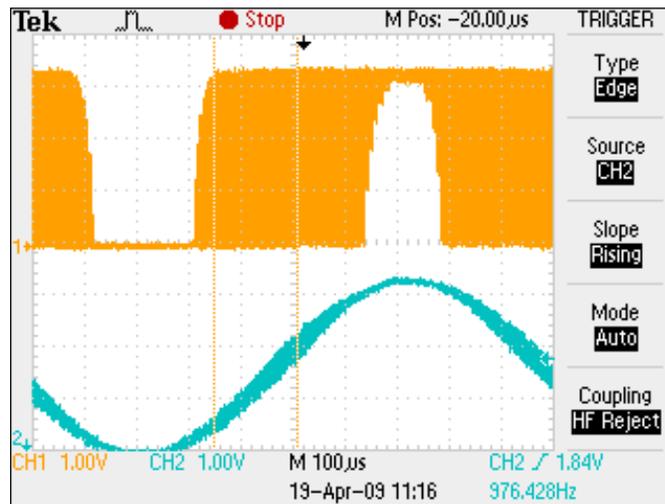
EPwm1Regs.ETCLR.bit.INT = 1;
PieCtrlRegs.PIEACK.all = 4;

```

Close function `"ePWM1A_compare_isr()"` with a closing curly brace (`}`).

Build, Load and Test

- Build, load and test the modified project. Please do not forget to reset the DSC before you perform a new test. This is always a good practice, since the chip will always start from a known state! Here's the sequence:
 - Trace → Reset → Reset CPU
 - Trace → Restart
 - Trace → Run
- A scope should show the 500 kHz-pulse sequence at ePWM1A (Peripheral Explorer Board Jumper J6-1) and a sine wave signal of 976 Hz at DAC1 (Peripheral Explorer Board Jumper J11-1).

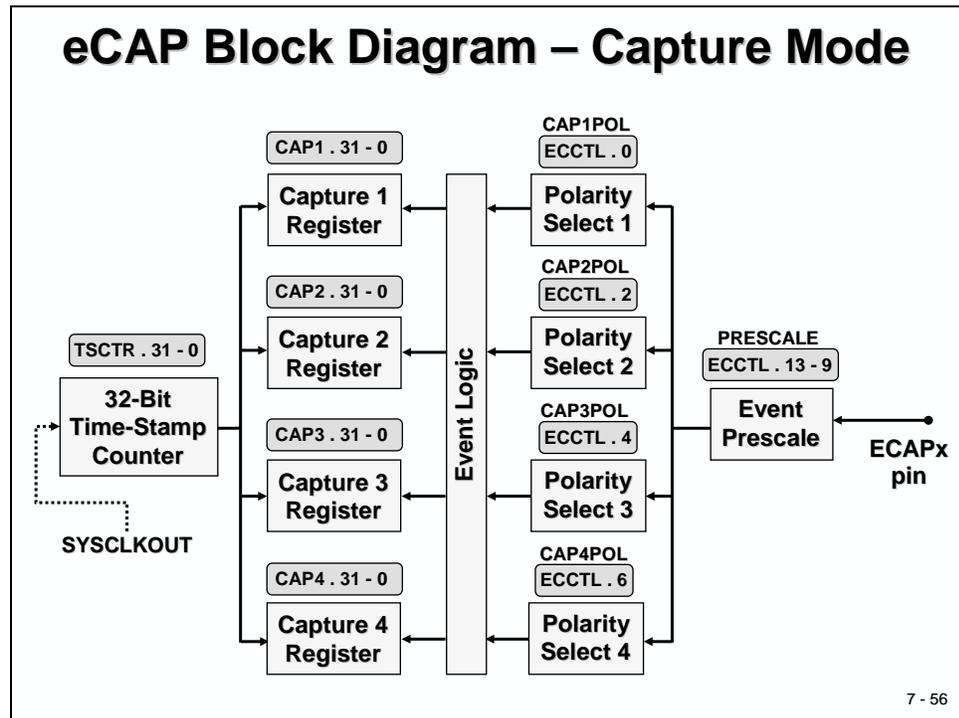


End of Lab7_9

eCAP Capture Module

The enhanced Capture (eCAP) module provides measurement units, which are useful for accurate time stamps of external events, such as rising or falling edges of digital signals.

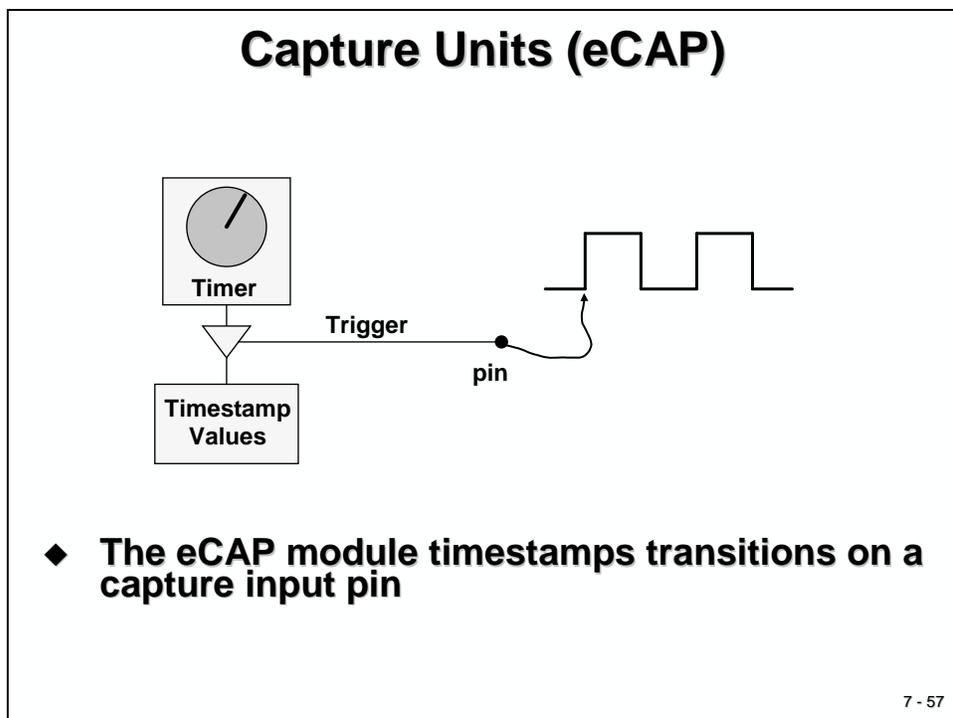
Capture Operating Mode



The capture units allow time-based logging of external logic level signal transitions on the capture input pins.

Devices in the F2833x family have four independent capture units; one of them is shown in Slide 7-56 above. Each capture unit is associated with a capture input pin. An event prescaler can be initialized to reduce the input frequency. Four polarity select bit fields define rising or falling edges as the trigger events for capture events 1 to 4. The measurement time-base is derived from the frequency SYSCLKOUT, in the case of the F28335ControlCard, this is 100 MHz. This signal will increment a 32-bit Time-Stamp Counter. In the event of a capture trigger signal the current value of this counter is captured and stored in the corresponding capture register.

Multiple identical eCAP modules can be contained in a 2833x system as shown in Slide 7-56. The number of modules is device-dependent and is based on target application needs.



Typical uses for the Capture Units are:

- Period and duty cycle measurements of pulse train signals
- Low speed measurement of a rotating machinery (e.g., toothed sprockets sensed via Hall sensors). A potential advantage for low speed estimation is given when we use “time capture” (32-bit resolution) instead of position pulse counting, which has a poor resolution at slow operating speeds.
- Elapsed time measurements between position sensor pulses.
- Decoding current or voltage amplitude derived from duty cycle encoded current/voltage sensors

Additionally, if the capture operation is not used in an application, an ePWM channel can be used as another single ended ePWM - output channel, with 32-bit resolution for frequency and duty cycle register setup. Since this operation mode is not the primary purpose of this unit, it is called "Auxiliary PWM" mode.

Some Uses for the Capture Units

- ◆ Measure the time width of a pulse
- ◆ Low speed velocity estimation from incr. encoder:

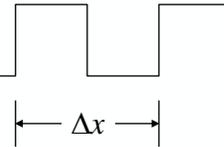
Problem: At low speeds, calculation of speed based on a measured position change at fixed time intervals produces large estimate errors

$$v_k \approx \frac{x_k - x_{k-1}}{\Delta t}$$

Alternative: Estimate the speed using a measured time interval at fixed position intervals

$$v_k \approx \frac{\Delta x}{t_k - t_{k-1}}$$

Signal from one quadrature encoder channel



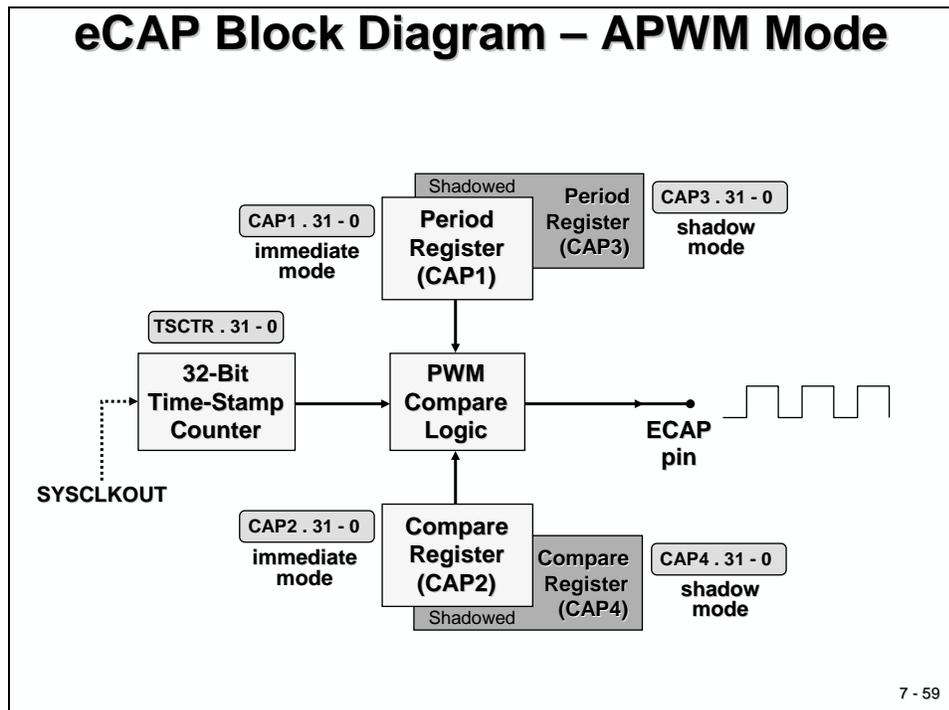
- ◆ Auxiliary PWM generation

7 - 58

Auxilliary PWM Operating Mode

As a second operating mode of a capture unit, auxiliary PWM mode can be used. In this case a single ended output PWM signal can be generated. Register CAP1 features as period register and register CAP2 as compare register.

eCAP Block Diagram – APWM Mode



7 - 59

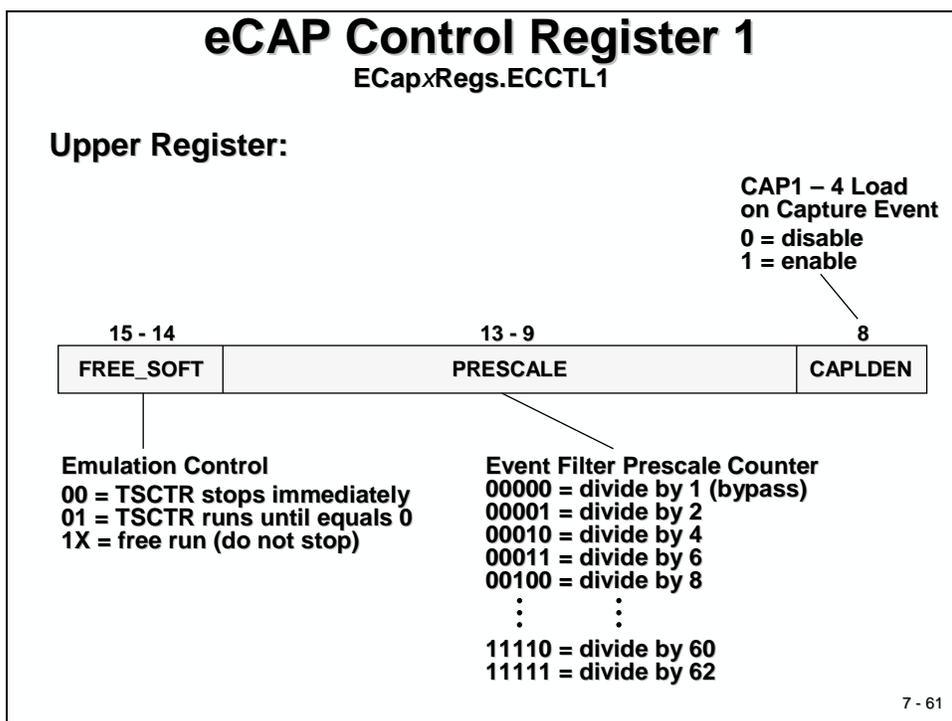
Capture Units Registers

Each of the four capture units is controlled by a set of individual registers.

eCAP Module Registers		
Name	Description	Structure
ECCTL1	Capture Control 1	ECapxRegs.ECCTL1.all =
ECCTL2	Capture Control 2	ECapxRegs.ECCTL2.all =
TSCTR	Time-Stamp Counter	ECapxRegs.TSCTR =
CTRPHS	Counter Phase Offset	ECapxRegs.CTRPHS =
CAP1	Capture 1	ECapxRegs.CAP1 =
CAP2	Capture 2	ECapxRegs.CAP2 =
CAP3	Capture 3	ECapxRegs.CAP3 =
CAP4	Capture 4	ECapxRegs.CAP4 =
ECEINT	Enable Interrupt	ECapxRegs.ECEINT.all =
ECFLG	Interrupt Flag	ECapxRegs.ECFLG.all =
ECCLR	Interrupt Clear	ECapxRegs.ECCLR.all =
ECFRC	Interrupt Force	ECapxRegs.ECFRC.all =

7 - 60

eCAP Control Register 1



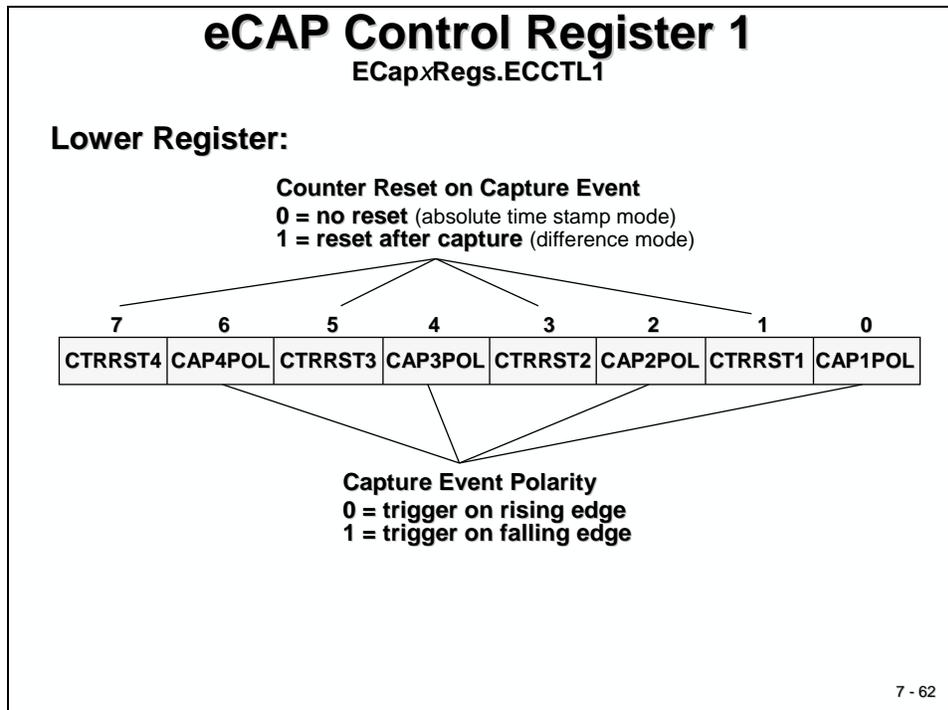
ECCTL1 [15-14] specify the interaction between the DSC and the JTAG emulation unit. If a running code hits a breakpoint, these two bits define how the capture unit behaves in this situation.

The prescaler counter in ECCTL1 [13-9] is used as an input filter of the capture signal. If set to "00001", every other edge is used to trigger the capture unit.

ECCTL [8] is the global enable switch for the particular capture unit.

ECCTL1 [6, 4, 2, 0] define rising (0) or falling (1) edge as trigger signal for capture event 1 to 4

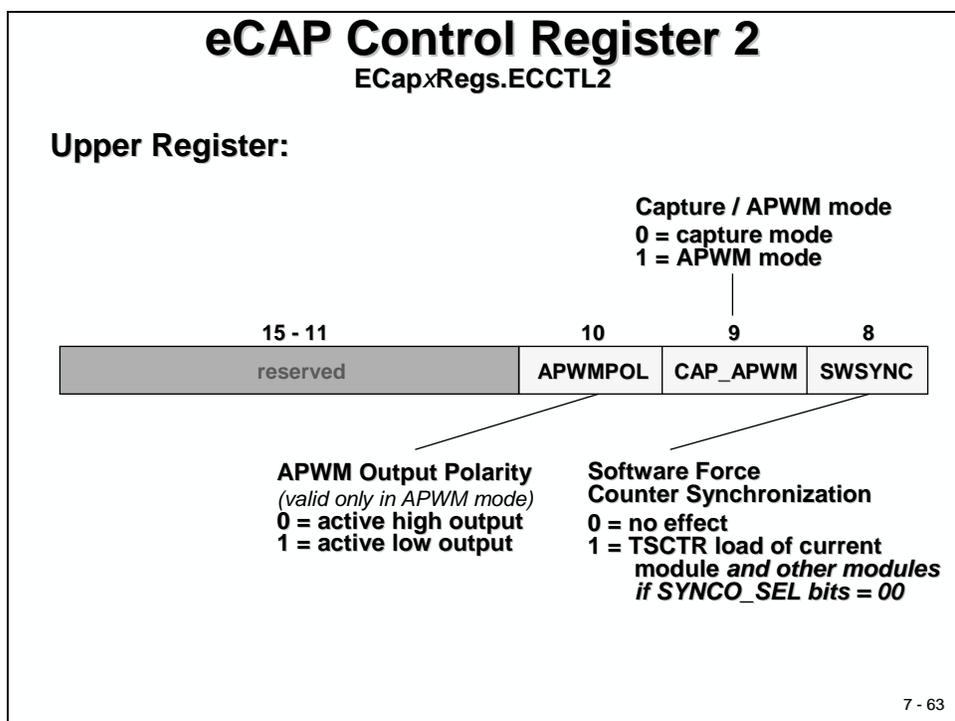
ECCTL1 [7, 5, 3, 1] specify absolute (0) or relative (1) time stamp mode. Absolute mode will never clear the timestamp - counter, after the capture unit has been started. Relative mode will clear the timestamp - counter simultaneously with the trigger event. For example, if bits 0 and 1 are initialized to 1, the first falling edge after enabling the capture unit will zero the timestamp-counter.



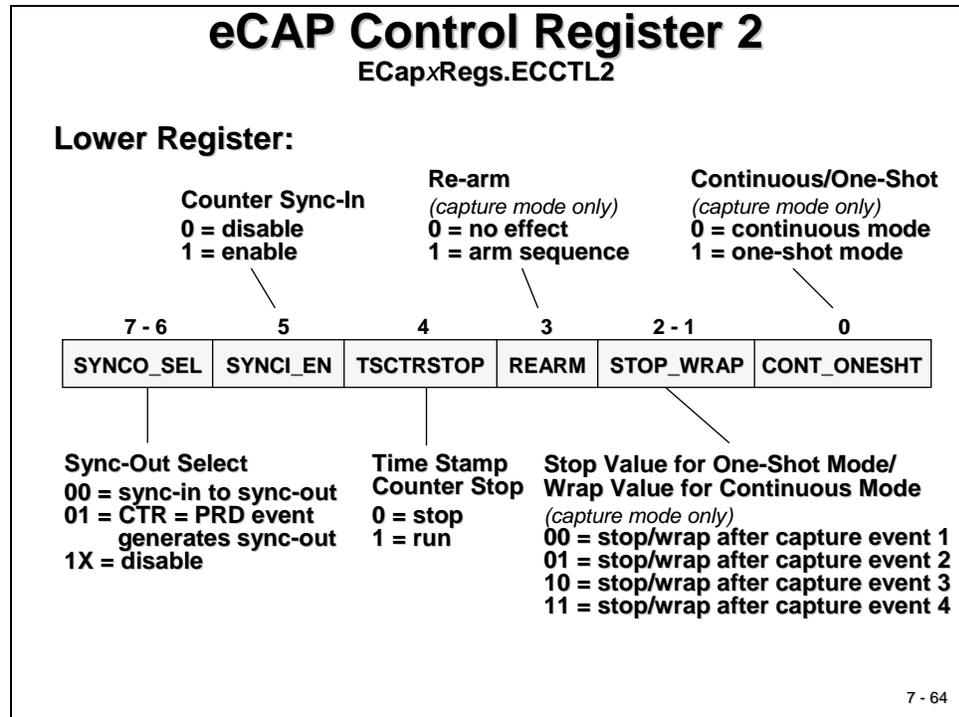
eCAP Control Register 2

ECCTL2 [10] defines the shape of an ePWM - output signal in auxiliary PWM operation mode to be active high (0) or active low (1). In capture operating mode, this bit is don't care.

ECTTL2 [9] selects either capture operating mode (0) or auxiliary PWM mode (1).



ECTTL2 [8] can be used in APWM-Mode to synchronize different capture units with each other. In case of an active sync input signal, register TSCTR is loaded with a start value.



ECTTL2 [7-6] are used to specify the source of the sync output signal (to achieve synchronized APWM channels). The code 00 will directly drive a sync input signal to the sync output. Code 01 will sent a sync output signal to other capture channels, if TBCTR = TBPRD.

ECTTL2 [5] allows the APWM sync input feature.

ECTTL2 [4] is the master switch to enable the capture counter unit.

ECTTL2 [3-0]: The continuous/one-shot block controls the start/stop and reset (zero) functions of a Modulo 4 event counter via a mono-shot type of action that can be triggered by the stop-value comparator and re-armed via software control.

One shot mode:

Once armed, the eCAP module waits for 1 to 4 (defined by the stop-value) capture events before freezing both the Modulo 4 event counter and the contents of registers CAP1 to 4 (i.e. time-stamps). Re-arming prepares the eCAP module for another capture sequence. Also re-arming clears the Modulo 4 counter to zero and permits loading of CAP1-4 registers again, providing that the CAPLDEN bit is set.

Continuous Mode:

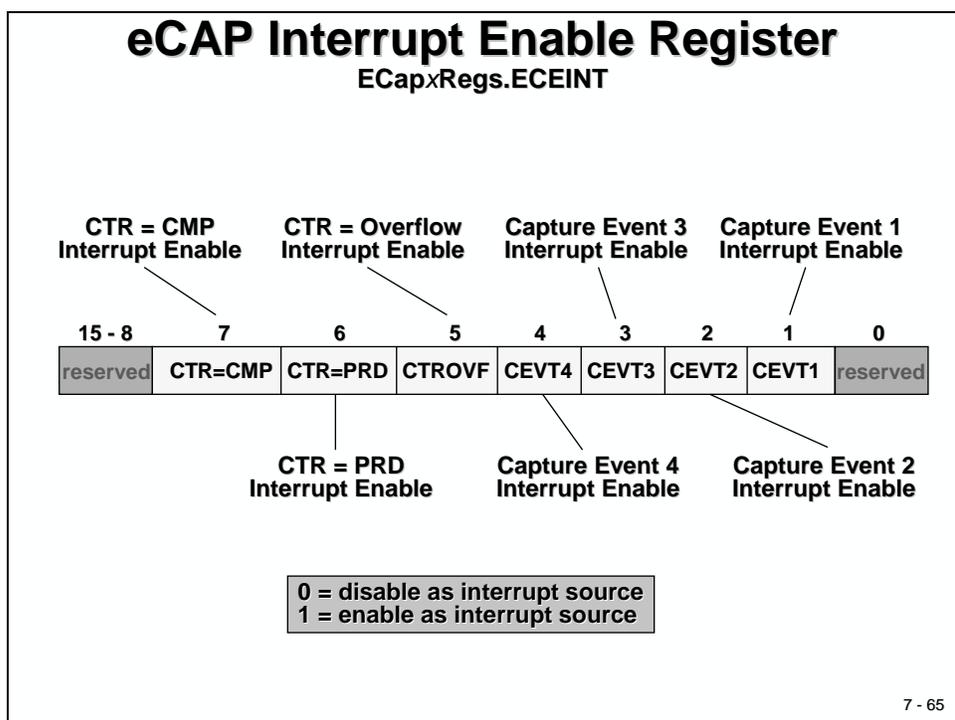
In continuous mode, the Modulo 4 event counter continues to run (0->1->2->3->0), the one-shot action is ignored, and capture values continue to be written to capture result registers CAP1 - x in a circular buffer sequence. The wrap around value will limit number x to the pre-selected result register.

eCAP Interrupt Enable Register

Interrupts can be requested based on internal events of the capture or APWM module.

ECEINT [4-1] will enable an interrupt request with capture event 1 to 4.

ECEINT [5] can be used to request an ISR in case of an overflow of the 32-bit register TBCTR. It is important to note such an overflow when results will be subtracted in a later calculation.



ECEINT [7, 6] are interrupt enable bits used in APWM mode. If TBCTR matches either register CMP or PRD, a corresponding interrupt service routine can be requested.

Lab7_10: ePWM1A 1 kHz captured by eCAP1

Objective

The F28335ControlCARD is used in combination with the Peripheral Explorer Board to output a 1 kHz square wave signal with a duty cycle of 50% at ePWM1A. We will use unit eCAP1 to measure period and duty cycle of this signal.

Note: for this exercise you will have to connect header J6-1 (ePWM1A) to header J10-1 (eCAP1) on the Peripheral Explorer Board.

Procedure

Open Project File

1. In project "Lab7" open the file "Lab7_1.c" and save it as "Lab7_10.c"
2. Exclude the file "Lab7_9.c" from build.

Edit Source File

3. In the function "Gpio_select()", switch the eCAP1 to pin GPIO24. On the Peripheral Explorer Board we can access eCAP1 via header J10-1, which is wired to pin GPIO24. Adjust register GPAMUX2 accordingly.
4. At the beginning of "Lab7_10.c", add a function prototype for a new local function "Setup_eCAP1()":

```
void Setup_eCAP1(void);
```

We will also need a new interrupt service routine for eCAP1. Add a new prototype:

```
interrupt void eCAP1_isr(void);
```

5. At the end of "Lab7_10.c" add the definition of the new function "Setup_eCAP1()". The objective is to initialize eCAP1 to capture 3 edges of signal ePWM1A:
 - 1st capture: rising edge
 - 2nd capture: falling edge
 - 3rd capture: rising edge

For register ECCTL2:

- use continuous mode
- set wrap counter to "wrap after 4 captures"
- do not re-arm
- enable counter
- disable the sync features
- select capture mode

For register ECCTL1:

- stop TSCTR immediately on Emulation Suspend

- prescaler : divide by 1
- enable capture load results
- edge select: CAP1 - falling ; CAP2 - rising; CAP3 - falling; CAP4 - rising
- reset TSCTR on CAP4 - event

For register ECEINT:

- enable event CAP3 interrupt request

6. In the function "main()", add a line to call function "Setup_eCAP1". The best position is directly after the function call to "Setup_ePWM1A()".
7. Next, in the function "main()", add a line to enable eCAP1 interrupt. Recall that eCAP1 is connected to bit 0 in PIE group 4. Also, change the code line to enable core interrupts in register IER. For the new exercise we have to enable INT1 (CPU Timer 0) and INT4 (eCAP1).
8. In the function "main()", search for the line in which we changed the PieVectTable entry for the CPU Timer 0 interrupt service (TINT0) and add a new line to load a new interrupt service routine address into PieVectTable for eCAP1:

```
PieVectTable.ECAP1_INT = & eCAP1_isr;
```

9. At the beginning of "Lab7_10.c", add two global variables:

```
Uint32 PWM_Period;
```

```
Uint32 PWM_Duty;
```

We will use the two variables to calculate the difference between CAP2 and CAP1 (duty) and CAP3 and CAP1 (period).

10. At the end of "Lab7_10.c", add the definition of the interrupt service function "eCAP1_isr()". Add the following commands to this function:
 - Clear flag "INT" in register ECCLR.
 - Clear flag "CEVT3" in register ECCLR. This will re-enable the CAP3 interrupt.
 - Calculate the differences:

```
PWM_Duty = (int32) ECap1Regs.CAP2 - (int32) ECap1Regs.CAP1;
```

```
PWM_Period = (int32) ECap1Regs.CAP3 - (int32) ECap1Regs.CAP1;
```

- Acknowledge the PIE - group interrupt 4:

```
PieCtrlRegs.PIEACK.all = 8;
```

Build, Load and Test

11. Build the modified project.

```
Project → Rebuild Active Project
```

12. Use a wire and connect header J6-1 (ePWM1A) to header J10-1 (eCAP1).

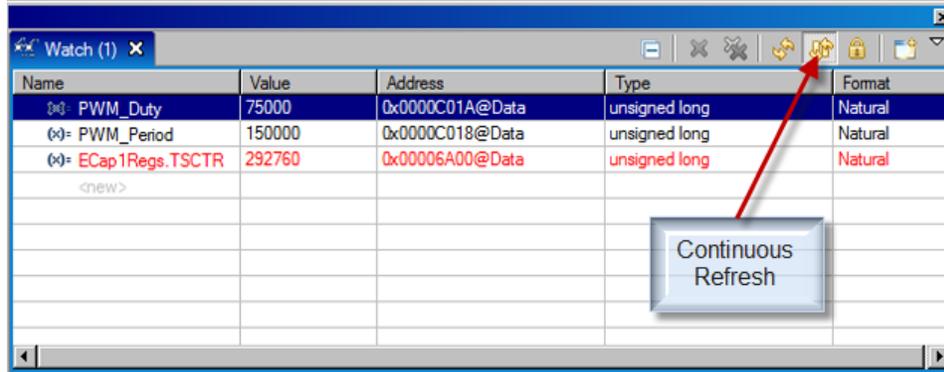
13. Load the modified code:

```
Target → Debug Active Project
```

14. Test the code:

Scripts → Realtime Emulation Control → Run_Realtime_with_Restart

15. Open the Watch Window and add the variables "PWM_Duty", "PWM_Period" and "ECap1Regs.TSCTR" to it. Also click right mouse in the Watch Window and enable "Continuous Refresh".



What do the values in "PWM_Duty" and "PWM_Period" mean? Remember that ePWM1A is a signal of 1 kHz with a period of 1 millisecond and a pulse width of 0.5 milliseconds. Our measurement unit has a resolution of $1/150\text{MHz} = 6.667\text{ ns}$. Therefore the value of 150,000 for "PWM_Period" translates into $150,000 * 6.667\text{ ns} = 1\text{ millisecond}$.

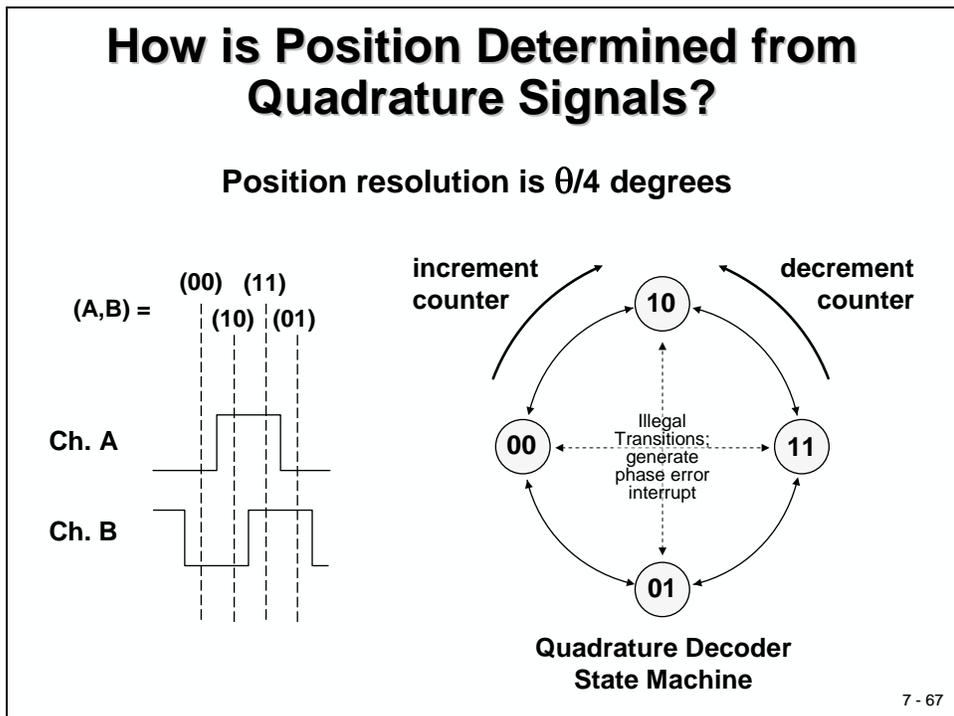
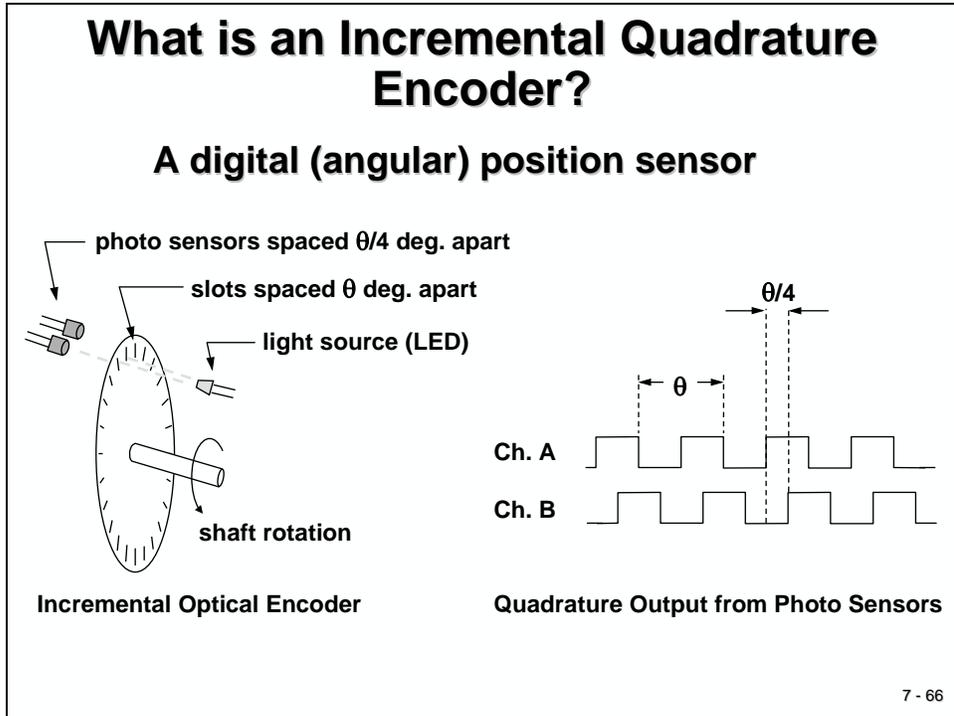
16. Finally halt the DSC:

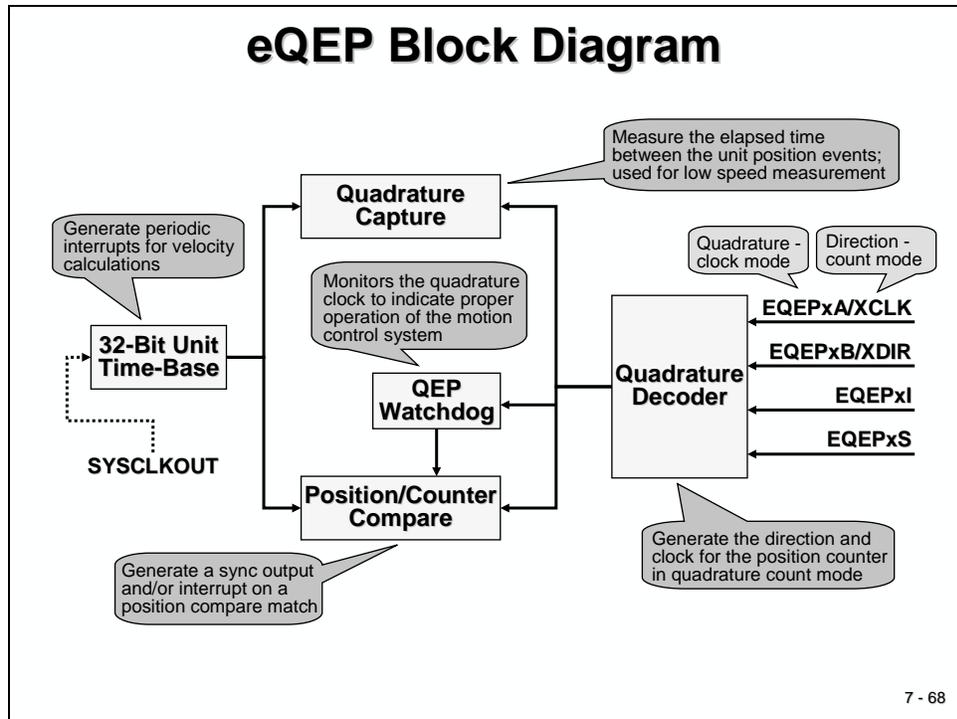
Scripts → Realtime Emulation Control → Full_Halt_with_Reset

END of LAB 7_10

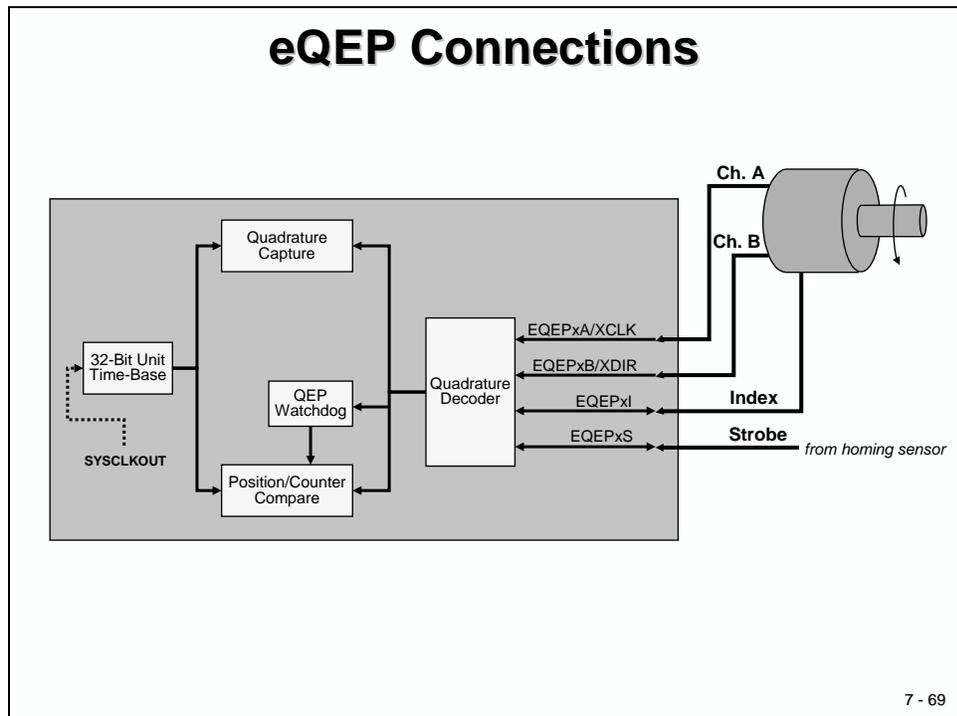
Enhanced QEP module

The F28335 device contains two enhanced Quadrature Encoder Positioning (eQEP) modules. These modules are usually used as hardware support units for incremental encoder devices.





The QEP is used (a) to estimate the speed and direction of a rotation or (b) to perform a positioning movement.



Infrared Remote Control

An interesting example for the capture unit is an infrared (IR) remote receiver. IR-signals are widely used for all kinds of handheld remote control devices, such as TV, radio tuners and amplifiers, DVD players, satellite receivers and many others. On the Peripheral Explorer Board an IR - sensor (TSOP32238 - <http://www.vishay.com>) is connected to capture unit ECAP4 (GPIO27). This unit will be used to measure the pulse widths of each pulse in a series sent to the IR-receiver.

IR - Protocols

Although IR-remote is widely used in consumer electronics, there are different and incompatible protocols. In a typical living room, you will usually find a collection of different remote control units:

Typical IR protocols are:

- RC5 code:
 - designed by Philips and also used by Loewe, Bang & Olufsen, Bose, Grundig, Marantz, Hauppauge, in model making and other areas
 - 14 - Bit code to address up to 32 devices with 64 instructions each
- SIRCS/ CNTRL - S Code:
 - designed by Sony
 - up to 21 data bits
- DENON code:
 - 16 bit transmission
- MOTOROLA - Code:
 - Similar to RC5
 - 11 bit transmission
 -

For our exercise we will focus on RC5 code.

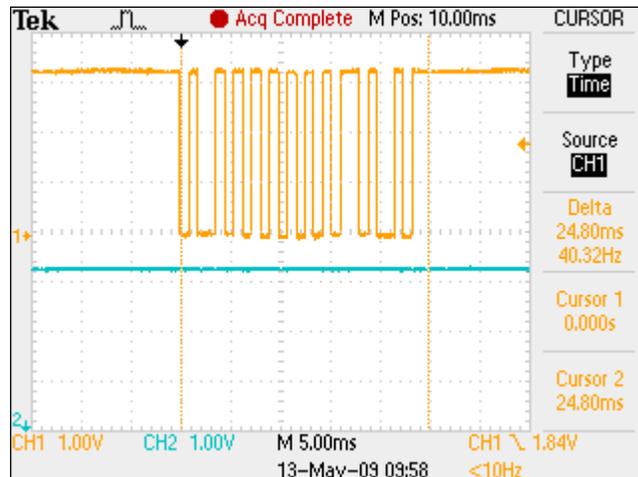
RC5 protocol

A RC5 protocol consists of 14 bits per transmission:

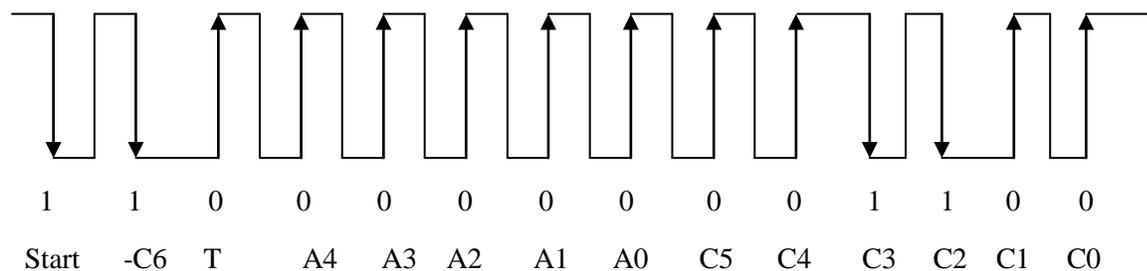
- 2 Start Bits (always '1'). Used to synchronize the transmission and to adjust the amplification of the receiver.
- 1 Toggle Bit (alternate '1' or '0'). Level is changed each time a button is pressed. Used to distinguish between a long duration (permanent pressing of a button) and a repetitive use of a button.
- 5 address bits. Allow the control of up to 32 devices by the same control unit.

00	01	02	03
TV1	TV2	Videotext	Video VD
04	05	06	07
Video LV1	VCR1	VCR2	experimental
08	09	10	11
Sat-Receiver	Camera	Sat-Receiver 2	

12 Video-CD	13 Camcorder	14	15
16 Audio- Amplifier 1	17 Receiver / Tuner	18 Audio Tape Recorder	19 Audio-Amplifier 2 / experimental
20 CD-Player	21 record player	22	23 DAT-Tape, MD- Recorder
24	25	26 CDR	27
28	29 lighting	30 lighting 2	31 Telephone
00 "0"	01 "1"	02 "2"	03 "3"
04 "4"	05 "5"	06 "6"	07 "7"
08 "8"	09 "9"	10	11
12 Standby	13 Mute	14 Default Setup	15
16 Volume +	17 Volume -	18 Brightness +	19 Brightness -
20 Color +	21 Color -	22 Bass +	23 Bass -
24 Highs +	25 Highs -	26 Balance right	27 Balance left
28	29	30	31
32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47
48 Pause	49	50 <<	51
52 >>	53 Play	54 Stop	55 Record
56	57	58	59
60	61	62	63 System select



The figure above shows the pattern for the "POWER" - Button of the PHILIPS universal remote control, as supplied with the Peripheral Explorer Board. RC5 is a bi-phase code with duration of 1778 μ s for a single bit. The following figure will explain the details:



The diagram above translates into address = 0 (TV) and command = 12 (ON/OFF/STANDBY). We will use this command in Lab7_11 to toggle LED LD2 of the Peripheral Explorer Board each time the POWER button of the remote control is pushed.

The space between the signal edges is either 889 μ s or 1778 μ s.

The RC5 idle separator between transmissions sequences is defined as 113ms.

We will use eCAP4 unit to capture four consecutive edges, in the sequence "falling - rising-falling - rising" and repeat this four edges capture until the end of the pulse series. After the capture of a full command, Lab7_11 must then decode the code and in case of address = 0 and code = 12 toggle led LD2.

Lab7_11: eCAP4 to receive a RC5 IR-signal

Objective

The F28335ControlCARD is used in combination with the Peripheral Explorer Board to receive a RC5 sequence (Phillips-Specification) from an IR remote control unit.

Procedure

Open Project File

1. In the project "Lab7", open the file "Lab7_1.c" and save it as "Lab7_11.c"
2. Exclude the file "Lab7_10.c" from build.
3. Add the provided source code file "Lab7_11_IR.c" to your project. This file is located in directory \Labs\Lab7.

Edit Source File

4. In file "Lab7_11.c" search the for function "Gpio_select()" and select eCAP4 function for pin GPIO27, which is connected to the IR-receiver TSOP32238.
5. At the beginning of "Lab7_11.c", add two new function prototypes for external functions:

```
extern void Calculate_IR_code(void);  
extern interrupt void eCAP4_isr(void);
```

6. Also add a new function prototype for local function:

```
void Setup_eCAP4(void);
```

7. Add the following global variables:

```
Uint16 result[100];           // distances between edges  
Uint16 signal_IR_ready=0;    // decode switch  
Uint16 IR_address;          // IR device address  
Uint16 IR_command;         // IR command  
Uint16 IR_Toggle;          // status of IR - Toggle bit
```

8. In "main()", after the basic initialization of the PIE vector table, add a line to load the address of our local interrupt function "eCAP4_isr" into the PIE vector table:

```
PieVectTable.ECAP4_INT = &eCAP4_isr;
```

Remember that this memory location is EALLOW protected!

9. In "main()", after the initialization of CPU Timer 0, add a for-loop to clear all elements of array "result[100]".

10. Next, add a line to enable the PIE - interrupt line for eCAP4:

```
PieCtrlRegs.PIEIER4.bit.INTx4 = 1;
```

11. Modify the line to initialize register IER accordingly! Recall that eCAP4 is controlled by line INT4!
12. In the endless while(1) loop of "main()", after the wait construction to wait for 100 milliseconds, add the following code:

```
if (signal_IR_ready == 1)
{
    Calculate_IR_code();
    if(IR_command == 12) GpioDataRegs.GPATOGGLE.bit.GPIO11 = 1;
    for (i=0;i<100;i++) result[i] = 0;
    signal_IR_ready = 0;
}
```
13. At the beginning of "main()", add a local integer variable "i".
14. In "main()", after the function call to "Gpio_select()", add a function call to a new function "Setup_eCAP4(). We will define this function shortly.
15. At the end of "Lab7_11.c", add the definition of function "Setup_eCAP4()". Take into account:
 - In register ECCTL1:
 - Set Polarity for CAP1 to 4 to: falling - rising - falling - rising
 - Select difference mode or "delta" mode for all 4 capture events
 - Enable loading of CAP registers
 - Do not use the prescale feature
 - For register ECCTL2 initialize:
 - Enable Capture mode
 - Disable all synchronization signals
 - For TSCTRSTOP select free running mode
 - Select continuous mode
 - Set Stop_Wrap to wrap after capture event 4
 - For register ECEINT:
 - Enable Interrupt after the 4th event.

Build, Load and Test

16. Build and Load the modified project.
 - **Project** → **Rebuild Active Project**
 - **Target** → **Debug Active Project**
17. In the "Debug" perspective, test the code:
 - **Scripts** → **Realtime Emulation Control** → **Run_Realtime_with_Restart**

Now Use an IR-Remote control Unit with RC5 - code (Philips, Loewe) and press the "ON/OFF" - key in front of the IR-Receiver at the Peripheral Explorer Board.

Each time you press the "POWER" - button of the remote control, LED LD2 (GPIO11) at the Peripheral Explorer Board should toggle.