C28x Event Manager

Introduction

Now it is time to discuss one of the most powerful hardware modules of the C28x, called the ‘Event Manager (EV)’. An EV is a unit that is able to deal with different types of time-based procedures. The core of this manager is a little bit similar to the DSPs core timer units Timer 0, 1 and 2. Although the Event Manager Timer units are also called "Timer 1, 2, 3 and 4", these timers are totally independent of the three core timers. So please do not mix them up! From now, when we speak of a timer unit, we have to clarify if it is a Core timer or an Event Manager timer!

The Event Manager Timer unit is a 16-bit counter/timer unit, whereas a Core Timer is a 32-bit register. The most important difference between the Event Manager and the Core timers is its input/output system. An EV is able to produce hardware signals directly from an internal time event. Thus this unit is most often used to generate time based digital hardware signals. This signal is a digital pulse with binary amplitude (0, 1). With the help of the EV-logic we can modify the frequency and/or the pulse width of these output signals. When we apply an internal control scheme to modify the shape of the signals during run time, we call this ‘Pulse Width Modulation’ (PWM).

PWM is used for two main purposes:

- Digital Motor Control (DMC)
- Analogue Voltage Generator

We will discuss these two main areas a little bit later. The C28x is able to generate up to 16 PWM output signals.

The Event Manager is also able to perform time measurements based on hardware signals. With the help of 6 edge detectors, called ‘Capture Unit’s’ we can measure the time difference between two hardware signals to determine the speed of a rotating shaft in rotations per minute.

The third part of the Event Manager is called ‘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.

The C28x is equipped with two Event Managers, called EVA and EVB. These are two identical hardware units; two 16-bit timers within each of these EVs generate the time base for all internal operations. In case of EVA the timers are called ‘General Purpose Timer’ T1 and T2, in case of EVB they are called T3 and T4.

This module includes also two lab-exercises ‘Lab5’ and ‘Lab5A’ based on the eZdsp and the Zwickau Adapter board. To perform Lab5A you will need a simple analogue oscilloscope.

Module Topics

Event Manager Block Diagram

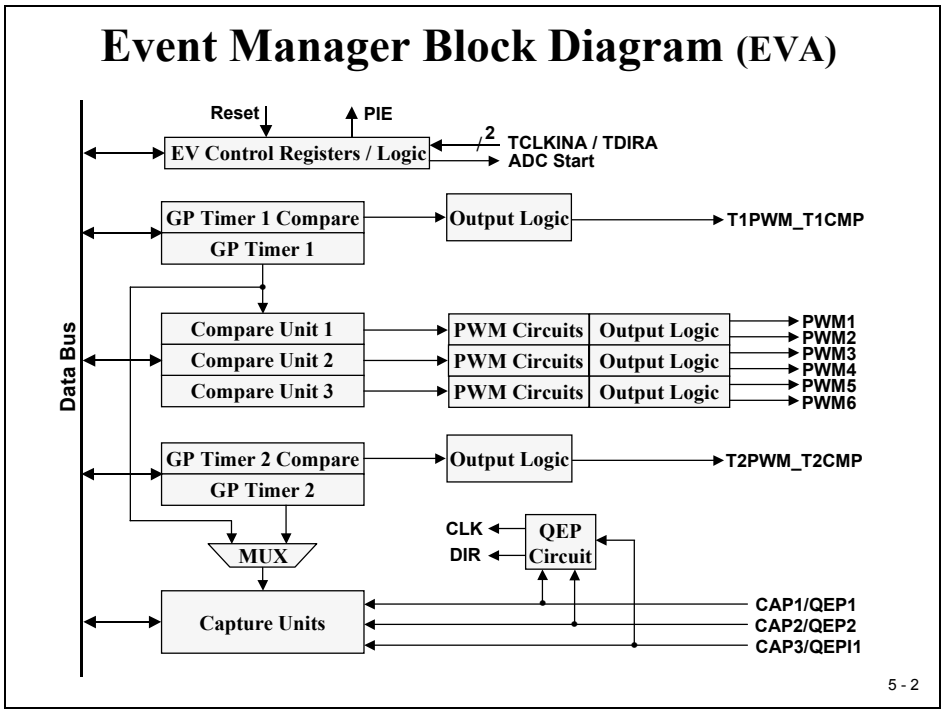
Each Event Manager is controlled by its own logic block. This logic is able to request various interrupt services from the C28x PIE unit to support its operational modes. Two external input signals 'TCLKINA' and 'TDIRA' are optional control signals and are used in some specific operational modes. A unique feature of the Event Manager is it ability to start the Analogue to Digital Converter (ADC) from an internal event. A large number of common microprocessors would have to request an interrupt service to do the same – the C28x does this automatically. We will use this feature in the next module!

The GP Timers 1 and 2 are two 16-bit timers with their own optional output signals T1PWM/T1CMP and T2PWM/T2CMP. We can also use the two timers for internal purposes only. Recall: to use any of the C28x units we have to set the multiplex registers for the I/O ports accordingly!

Compare Unit 1 to 3 are used to generate up to 6 PWM signals using GP Timer 1's time base. A large number of technical applications require exactly 6 control signals, e.g. three phase electrical motors or three phase electrical power converters.

Three independent capture units CAP1, 2, and 3 are used for speed and time estimation. An incoming pulse on one of the CAP lines will take a 'time stamp' from either GP Timer 1 or 2. This time stamp is proportional to the time between this event and the previous one.

The QEP-unit redefines the 3 input lines CAP1, 2, and 3 to be used as sensed edge pulses (QEP1, 2) and a zero degree index pulse (QEPI1) for an incremental encoder.

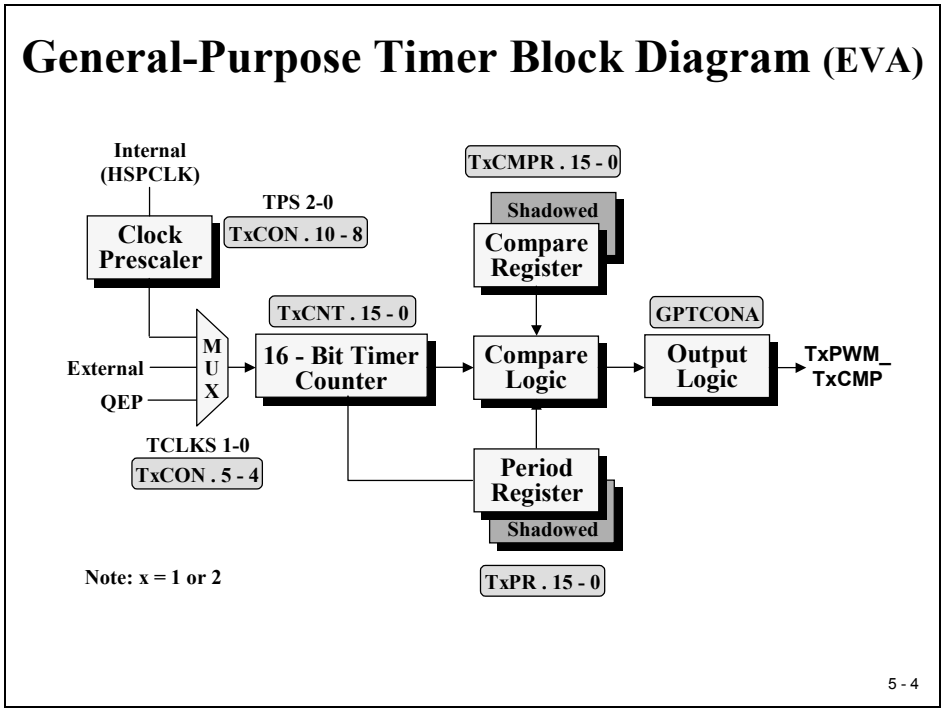


General Purpose Timer

The central logic of a General Purpose Timer is its Compare Block. This unit continually compares the value of a 16-bit counter (TxCNT) against two other registers: Compare (TxCMPR) and Period (TxPR). If there is a match between counter and compare, a signal is sent to the output logic to switch on the external output signal (TxPWM). If counter matches period, the signal is switched off. This is the basic operation in "asymmetric" mode. The second basic operating mode - "symmetric" mode – will be explained a little bit later. Register GPTCONA controls the shape of the physical output signal.

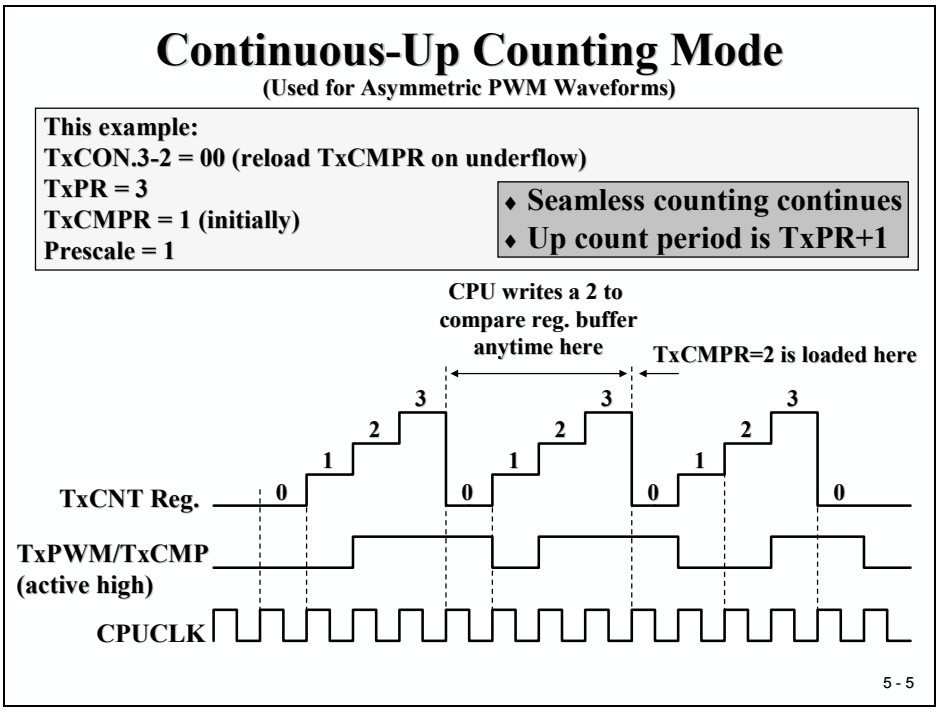
The timer's clock source is selectable to be an external signal (TCLKIN), the QEP-unit or the internal clock. TxCON-bits5 and 4 control the multiplexer. In case of internal clock selection the clock is derived from the high-speed clock prescaler (HSPCLK). When you calculate the desired period you will have to take into account the setup of register HISPCP! To adjust the period of a General Purpose Timer one can use an additional prescaler (TPS, TxCON2-0), which gives a scaling factor between 1 and 128.

The direction of counting depends on the selected operation mode.

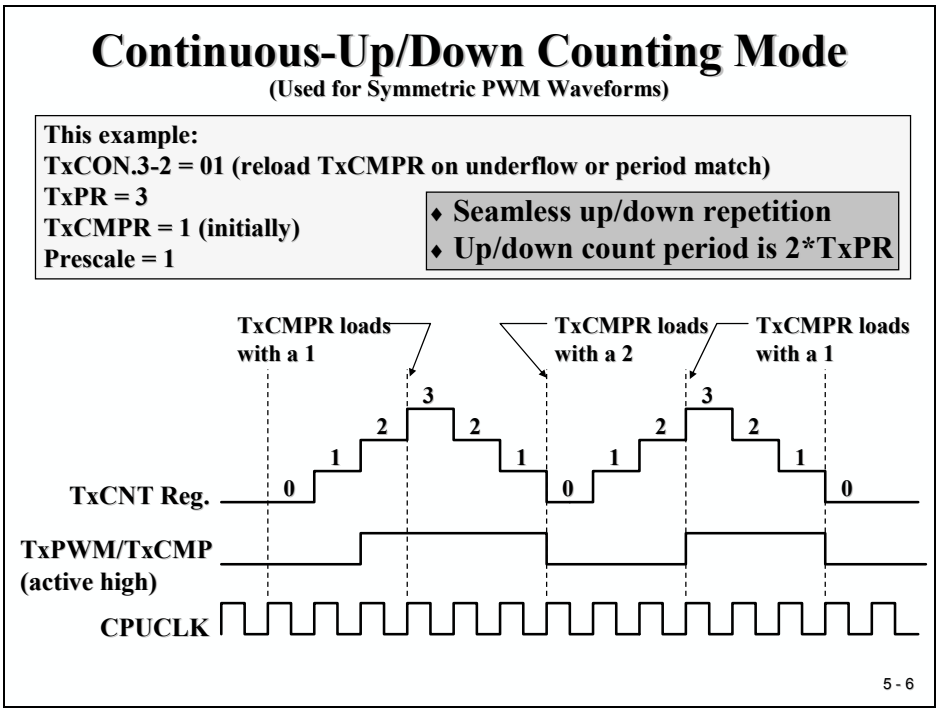


Another unique feature of the C28x is its "shadow" functionality of operating registers, in the case of GP Timers 1 and 2 available for compare register and period register. For some applications it is necessary to modify the values inside a compare or period register every period. The advantage of the background registers is that we can prepare the values for the next period in the previous one. Without a background function we would have to wait for the end of the current period, and then trigger a high priorized interrupt. Sometimes this principle will miss its deadline…

Timer Operating Modes

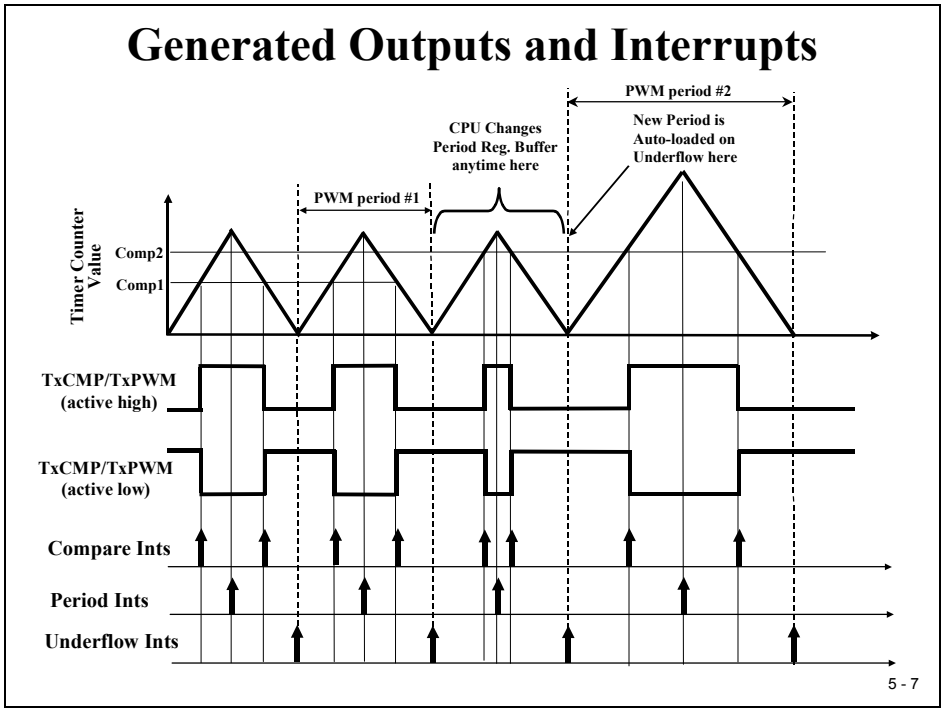


The slides give two examples of the two most used operating modes. Note: There are two more modes – see data sheet.



Interrupt Sources

Each of the two timers of Event Manager A (EVA) is able to generate four types of interrupt requests: timer underflow (counter equals zero), timer compare (counter equals compare register), timer period (counter equals period register) and timer overflow (counter equals 0xFFFF – not shown on slide). The slide also shows two options for the physical shape of the output signal (TxPWM) -"active high" and "active low". The two options not shown are "forced low" and "forced high". All four options are controlled by register GPTCONA.

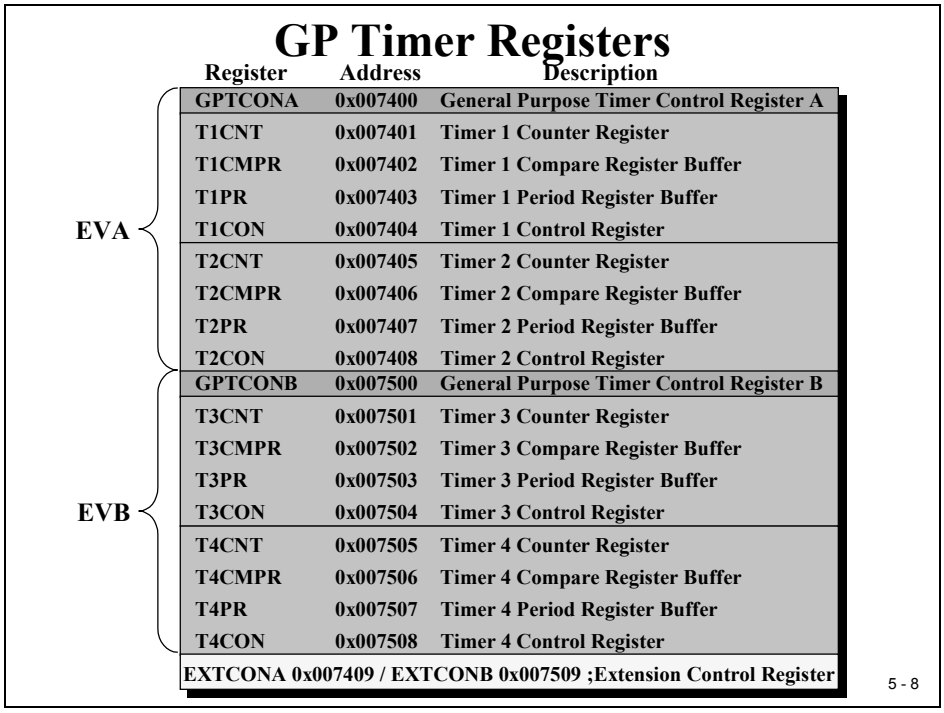


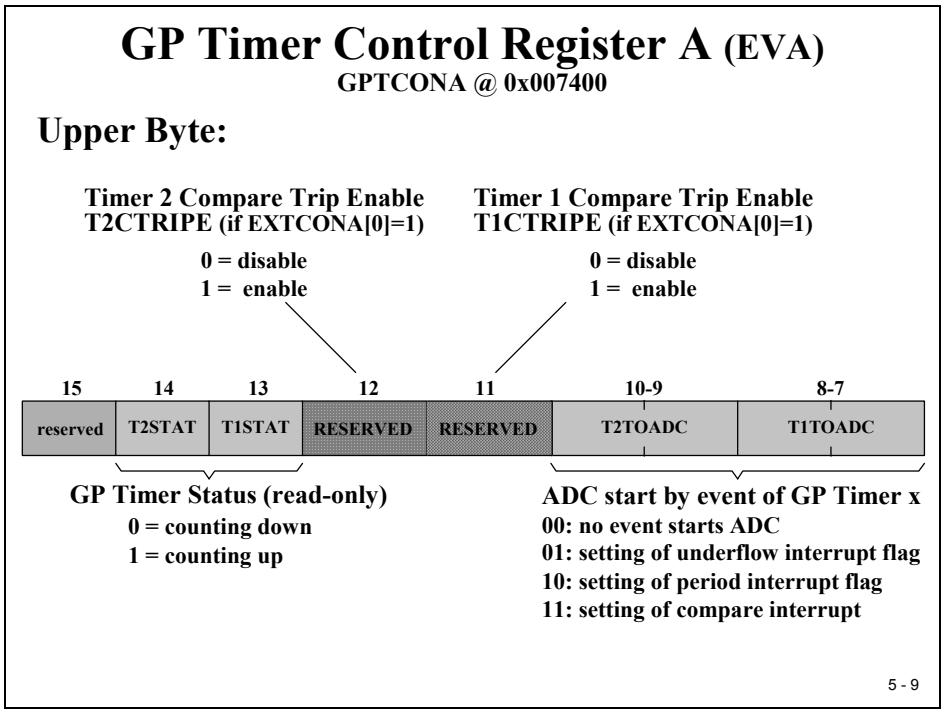
The example on the slides assumes "Counting up/down Mode" and that the timer starts with value "Comp1" loaded into TxCMPR and "period #1" in TXPR. At some point in period 2 our code changes the value in TXCMPR from "Comp1" to "Comp2". Thanks to the compare register background (or "shadow") function this value is taken into foreground with the next reload condition. This leads to the new shape of the output signal for period 3. Somewhere in period 3 our code modifies register TxPR – preparing the shape of period 4.The answer is the PIE (Peripheral Interrupt Expansion)-unit.

Please note that we have two points for Compare Interrupts within each period. Question: How do we distinguish between them? When we go through all timer control registers on the next pages please remember this question. There must be a way to specify whether we are in the first or second half of a period.

GP Timer Registers

To set up an Event Manager Timer we have to configure five registers per timer. If we'd like to use one or more timer interrupt sources, then we have to set up a few more registers. The next slides are going through the registers step by step, at the end a lab exercise is waiting for you!

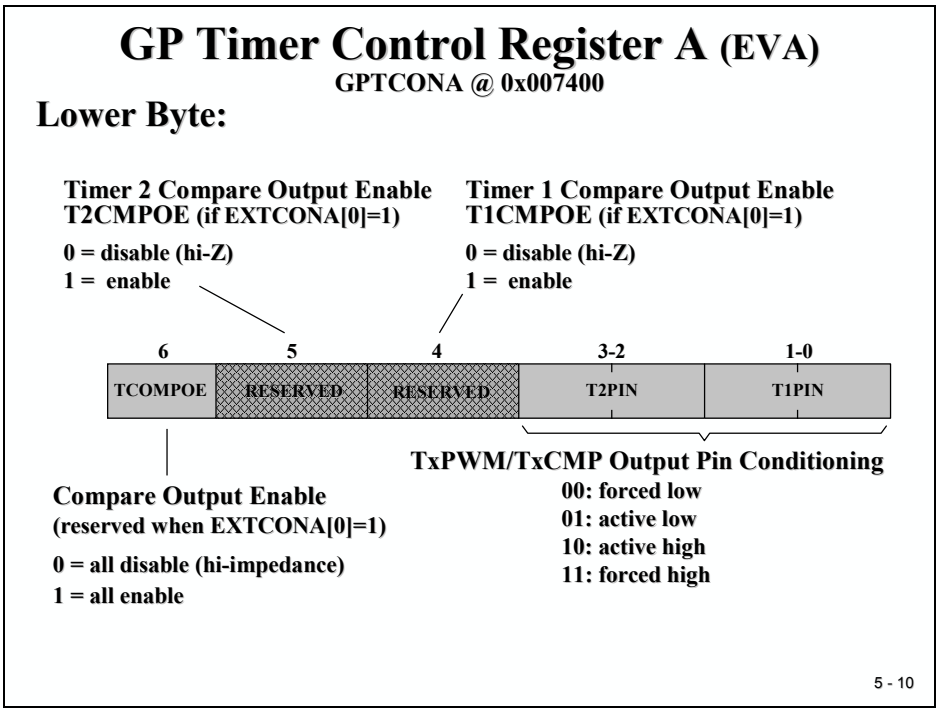




GP Timer Control Register GPTCONA

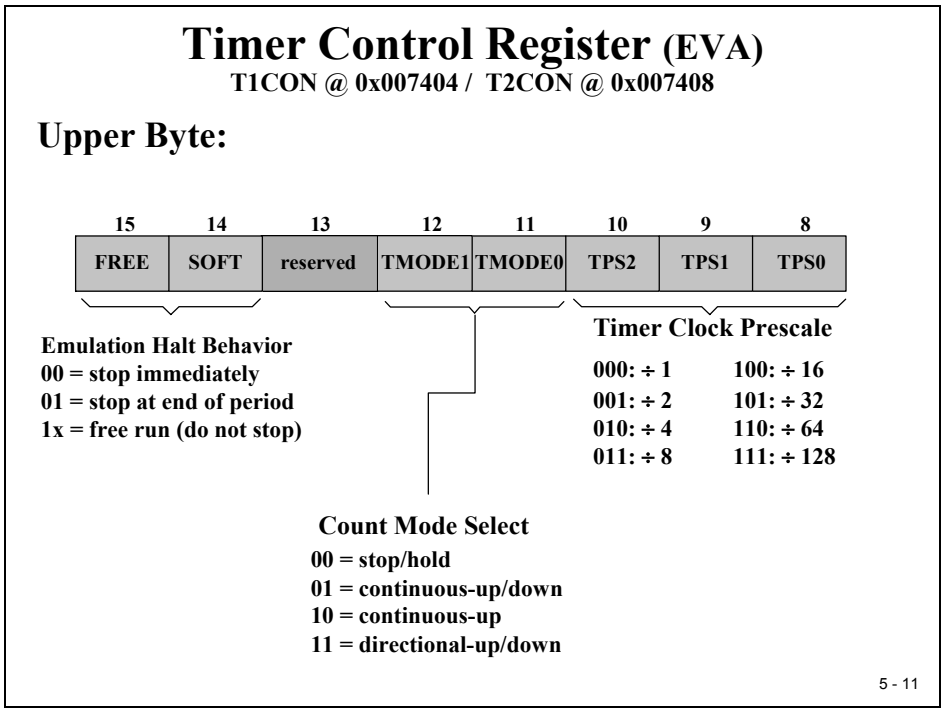
Bits 14 and 13 are status bits used to report if the timer is counting up or down. Bits 10 to 7 are used to perform the automatic start of the ADC from the specified timer event. Bits 3 to 0 define the shape of the output signal. Bit 6 is used to enable the two physical output signals for timer 1 and timer 2 simultaneously.

Note: There is an enhanced operating mode available. This extended mode is switched on by setting bit 0 in register EXTCONA to 1. In this case the definition of some of the register bits will change. Bit 6 is no longer used; instead bits 5 and 4 are used to enable/disable the output signals separately for timer 1 and timer2. Bits 12 and 11 are now used to enable a new power electronic safety feature called "Timer Compare Trip". We will not go into these extended operating modes during this tutorial, so just treat all these new control bits as "reserved". Reserved means for a write operation into registers you can set the bit position as a "don't care".



Timer Control Register TxCON

Next, we have to set up the individual timer control registers. The layout is shown on the next two slides. Bits 15 and 14 are responsible for the interaction between the timer unit and a command executed by the JTAG-Emulator unit. We will find this control bits pair for all other peripheral units of the C28x. It is very important to be able to set up a definite behaviour when, for example, the execution of our code hits a breakpoint. In case of real hardware connected to the outputs it could be very dangerous to stop the outputs in a random fashion. For the timer unit we can specify to stop its operation immediately, at the end of a period, or not at all. Of course this depends on the hardware project, for our lab exercises its good practice to stop immediately.



Bits 12-11 select the operation mode. We've discussed the two most important modes before; the two remaining modes are the "Directional Up/Down" mode and the "Stop/Hold" mode. The first mode uses an external input (TDIRA) to specify the counting direction; the latter just halts the timer in its current status, no re-initialization needed to resume afterwards.

Bits 10 to 8 are the input clock prescaler to define another clock division factor. Recall that the counting frequency is derived from:

- The external oscillator (30MHz)
- The internal PLL-status (PLLCR: multiply by 10/2 = 150 MHz)
- The High speed clock prescaler (HSPCP = divide by 2: 75 MHz) and
- The Timer Clock Prescale factor (1 to 128)

This gives us the option to specify the desired period for a timer. For example, to setup a timer period of 100 milliseconds, we can use this calculation:

Timer input pulse = (1/ext_clock_freq) * 1/PLL * HISPCP * Timer TPS

1,7067 µs = (1/ 30 MHz) * 1/5 * 2 * 128

100 ms / 1,7067 µs = 58593.

➔ Load TxPR with 58593 to set the timer period to 100 milliseconds.

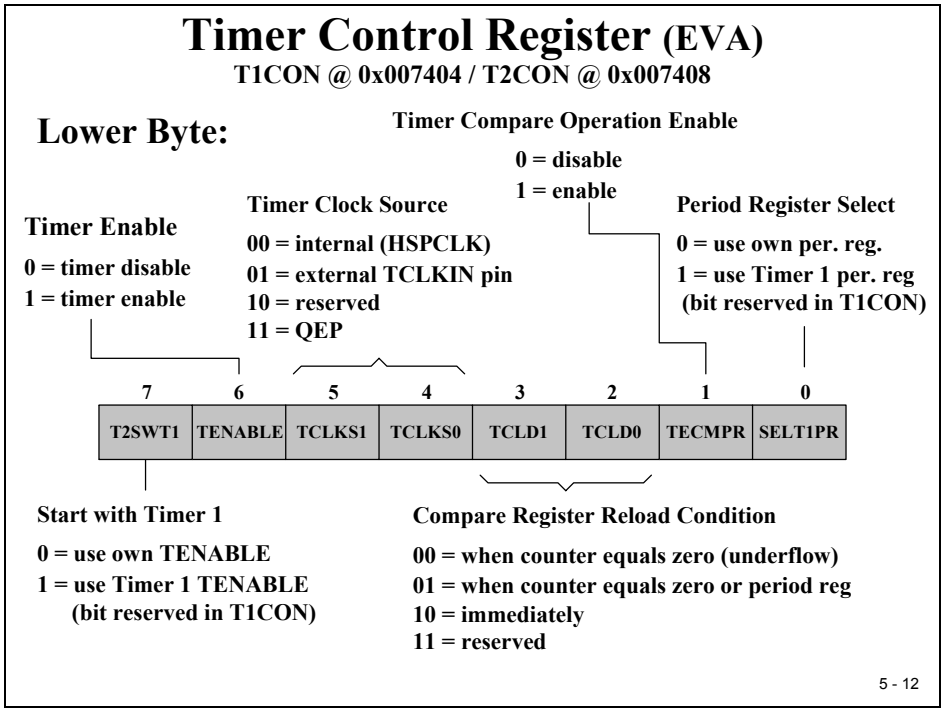
Bit 6 enables the timer operation. At the end of an initialization procedure we will have to set bit 6 to 1 to start the timer.

Bits 5 and 4 select the timer clock source; Bits 3 and 2 define the point of time to reload the value out of the background register into the foreground compare register.

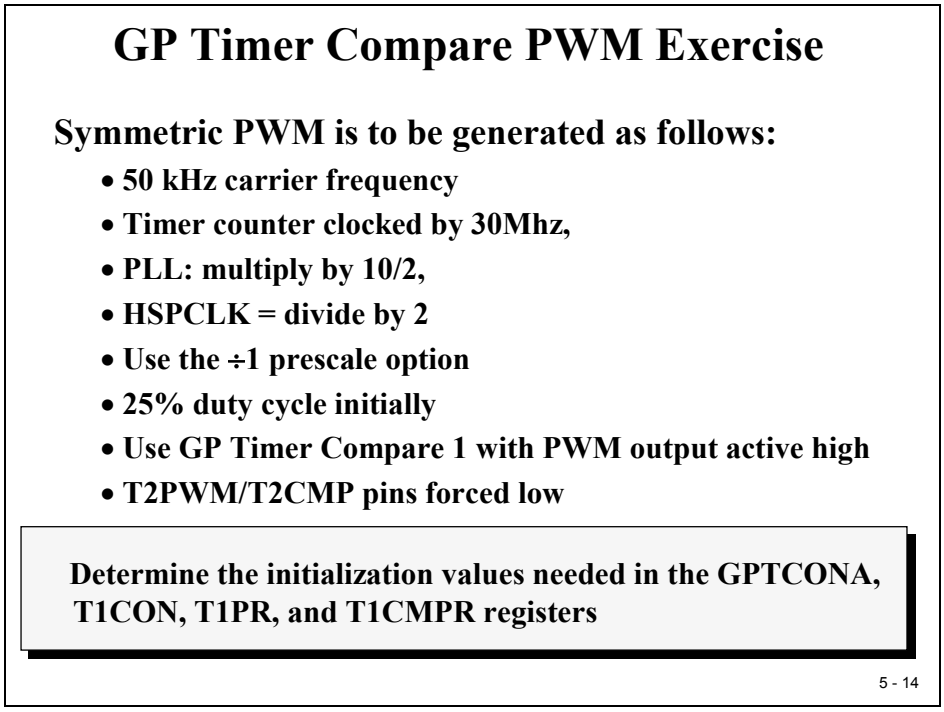
Bit 1 is used to enable the compare operation. Sometimes you'll need an internal period generator only, for these applications you can switch off the compare operation that is the generation of switch patterns for the output lines.

Bits 7 and 0 are timer 2 specific bit fields. They are "don't cares" for register T1CON. With the help of bit 7 we can force a start of timer 1 and 2 simultaneously. In this case, both timers start if bit 6 (TENABLE) of T1CON is set.

Bit 0 forces timer 2 to use period register of timer 1 as base to generate a synchronized period for timer 2 and timer 1.



Now let's make a calculation. Assume that your task is to setup a PWM signal with a period of 50 kHz and a pulse width of 25%:



Solution :

- PLLCR =
- HSPCLK =
- GPTCONA =
- T1CON =
- T1PR =
- T1CMPR =

GP Timer Interrupts

To enable one of the interrupt sources of Event Manager A we have to set a bit inside EVAIMRA, B or C.

EVAIMRA Register

@ 0x742C

15	14	13	12	11	10	9	8
-	-	-	-	-	T1OFINT	T1UFINT	T1CINT

7	6	5	4	3	2	1	0
T1PINT	-	-	-	CMP3INT	CMP2INT	CMP1INT	PDPINT

Interrupt Mask Bits	Bit	Event
0 = disable interrupt	10:	Timer 1 Overflow
1 = enable interrupt	9:	Timer 1 Underflow
	8:	Timer 1 Compare match
	7:	Timer 1 Period match
	3:	Compare Unit 3, Compare match
	2:	Compare Unit 2, Compare match
	1:	Compare Unit 1, Compare match
	0:	Power Drive Protect input, EVA

5 - 16

EVAIMRB Register

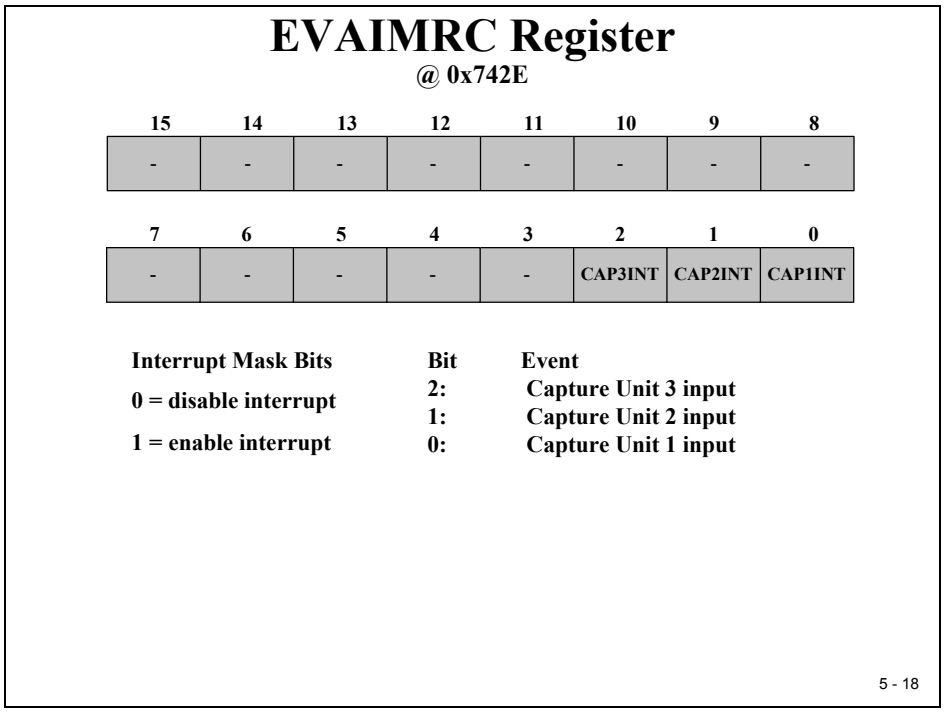
@ 0x742D

15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-

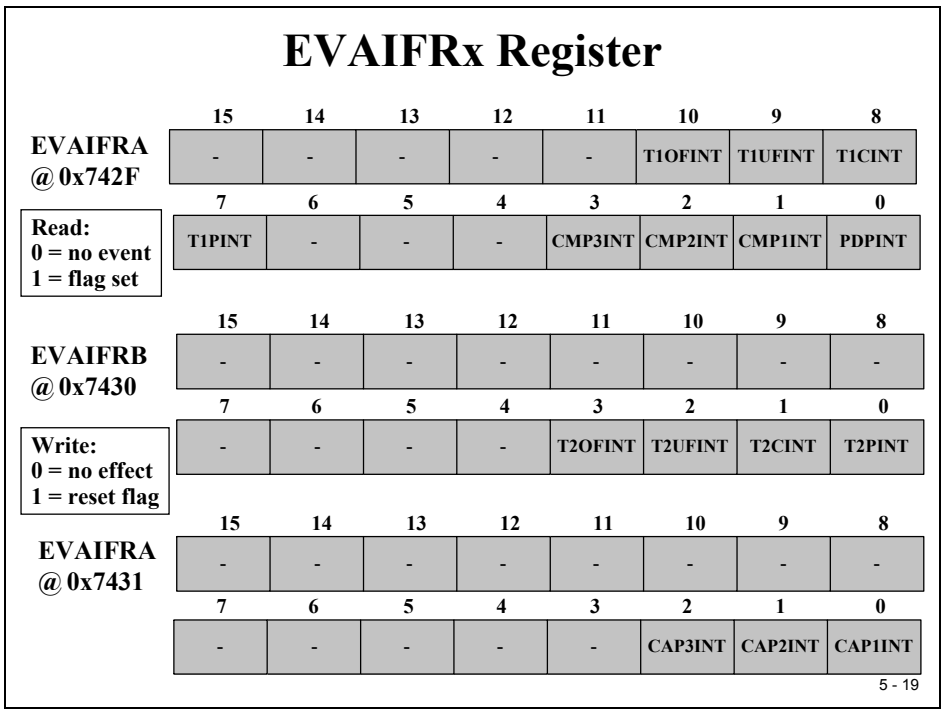
7	6	5	4	3	2	1	0
-	-	-	-	T2OFINT	T2UFINT	T2CINT	T2PINT

Interrupt Mask Bits	Bit	Event
0 = disable interrupt	3:	Timer 2 Overflow
1 = enable interrupt	2:	Timer 2 Underflow
	1:	Timer 2 Compare match
	0:	Timer 2 Period match

5 - 17



An interrupt event will be marked by the DSP in Register EVAIFRA, B and C.



Lab 5: Let's play a tune!

Objective

The Zwickau Adapter board has a small loudspeaker connected to output T1PWM (a small on-board amplifier is also necessary) - close Jumper JP3 of the adapter board. The task for this lab exercise is to play 8 basic notes of an octave. Optionally, you can improve this lab to play a real tune. To keep it simple we will generate all notes as simple square waves. Of course, for a real musician this would be an offence because a real note is a pure sine wave and harmonics. To generate a sine wave one would have to adjust the pulse width of the PWM signal to the instantaneous voltage of the sine. This scheme is a basic principle to generate sine waves with the help of a PWM output, but as I said, let's keep it simple. If you've additional time in your laboratory and you'd like to hear pure notes, then try it later.

Lab 5: Let's play a tune !

Aim:

- Exercise with Event Manager A General Purpose Timer 1
- Use Lab 4 as a starting point. In Lab 4 we initialised Core Timer 0 to request an interrupt every 50 ms. We can use this ISR to load the next note to T1PWM.
- Timer1 output 'T1PWM' is connected to a loudspeaker

Basic Tune Frequencies:

c^1	: 264 Hz
d	: 297 Hz
e	: 330 Hz
f	: 352 Hz
g	: 396 Hz
a	: 440 Hz
h	: 495 Hz
c^2	: 528 Hz

5 - 20

The result of Lab4 is a good starting point for Lab5. Recall that we initialized the core timer 0 to request an interrupt service every 50 milliseconds. Now we can use this interrupt service routine to load the next note into the period and compare register of T1. A time of 50 milliseconds is a little bit too fast, but we have a 50ms variable "CpuTimer0.InterruptCount". If we wait until the value of this variable is 10 we know that an interval of 500ms is over. After this period we can play the next note starting with c^1 and go to c^2 in an endless loop. Or, try to play the notes alternately as an ascending and descending sequence (or: recall a nursery rhyme).

New Registers involved in Lab 5:

• General Purpose Timer Control A	:	GPTCONA
• Timer 1 Control Register	:	T1CON
• Timer 1 Period Register	:	T1PR
• Timer 1 Compare Register	:	T1CMPR
• Timer 1 Counter Register	:	T1CNT
• EV- Manager A Interrupt Flag A	:	EVAIFRA
• EV- Manager A Interrupt Flag B	:	EVAIFRB
• EV-Manager A Interrupt Flag C	:	EVAIFRC
• EV- Manager A Interrupt Mask A	:	EVAIMRA
• EV- Manager A Interrupt Mask B	:	EVAIMRB
• EV- Manager A Interrupt Mask C	:	EVAIMRC
• Interrupt Flag Register	:	IFR
• Interrupt Enable Register	:	IER

5 - 21

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab5.pjt** in E:\C281x\Labs.

2. Open the file Lab4.c from E:\C281x\Labs\Lab4 and save it as Lab5.c in E:\C281x\Labs\Lab5.

3. Add the source code file to your project:
   - **Lab5.c**

4. From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source* add:
   - **DSP281x_GlobalVariableDefs.c**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\cmd* add:
   - **F2812_EzDSP_RAM_lnk.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\cmd* add:
   - **F2812_Headers_nonBIOS.cmd**

From *C:\ti\c2000\cgtoolslib* add:

- **rts2800_ml.lib**

From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add to project:

- **DSP281x_CpuTimers.c**
- **DSP281x_PieCtrl.c**
- **DSP281x_PieVect.c**
- **DSP281x_DefaultIsr.c**

Project Build Options

5. Setup the search path to include the peripheral register header files. Click:

   Project → Build Options

   Select the Compiler tab. In the preprocessor Category, find the Include Search Path (-i) box and enter:

   C:\tidcs\C28\dsp281x\v100\DSP281x_headers\include;..\include

6. Setup the stack size: Inside Build Options select the Linker tab and enter in the Stack Size (-stack) box:

   400

   Close the Build Options Menu by Clicking <**OK**>.

Build and Load

7. Click the "Rebuild All" button or perform:

   Project → Build

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

   So far we just generated a new project 'Lab5' with the old code from Lab4. If you run the code now you should see the 'Knight Rider' of Lab4. Now we can start to modify our code in 'Lab5.c'.

Modify Source Code

8. Open Lab5.c to edit: double click on "Lab5.c" inside the project window. First we have to cancel the parts of the code that we do not need any longer. The definition of array "LED [8]" in function "main" is of no use for this lab – cancel it.

9. Next we have to change the GPIO multiplex status; we need T1PWM as signal at the pin. Go into function "Gpio_select()" and modify the multiplex register setup.

10. Go into your local function "InitSystem" and enable the clock for Event Manager A.

11. Inside "main", just before the line:

    CpuTimer0Regs.TCR.bit.TSS = 0;

    we have to initialize the Event Manager Timer 1 to produce a PWM signal. This involves the registers "GPTCONA", "T1CON", "T1CMPR" and "T1PR".

    For register "GPTCONA" it is recommended to use the bit-member of this predefined union to set bit "TCMPOE" to 1 and bit field "T1PIN" to "active low".

    For register "T1CON" set

    - The "TMODE"-field to "counting up mode";
    - Field "TPS" to "divide by 128";
    - Bit "TENABLE" to **"disable timer"** (we will enable it later)
    - Field "TCLKS" to "internal clock"
    - Field "TCLD" to "reload on underflow" and
    - Bit "TECMPR" to "enable compare operation"

12. Last question is: how do we initialize "T1PR"? Well, obviously we need 8 different values for our 8 basic notes. So let's define a new integer array "frequency [8]" as a local variable in main!

13. How do we initialize array "frequency [8]"?

    We can initialize the array together with the definition inside main:

    int frequency [8] = {?,?,?,?,?,?,?,?};

    A basic octave is a fixed series of 8 frequencies. AND: there is a relationship between the basic note c1 (264 Hz) and the next notes:

264 Hz (c1)	396Hz (g) = 3/2 * c1
297Hz (d) = 9/8 * c1	440Hz (a) = 5/3 * c1
330Hz (e) = 5/4 * c1	495Hz (b) = 15/8 * c1
352Hz (f) = 4/3 * c1	528 Hz (c2) = 2 * c1

What is the relationship between these frequencies and T1PWM? Answer: We have to setup T1PR to generate a PWM period according to this list. The equation is:

T1_PWM_Freq = 150MHz / (HISPCP * TPS * T1PR)

For c1 = 264 Hz we get: T1PR = 150MHz / (2 * 128 * 264 Hz) = 2219.

For d = 297 Hz we use: T1PR = 150MHz / (2 * 128 * 297 Hz) = 1973.

Calculate the 8 initial numbers and complete the initial part for array "frequency"!

14. Next step: Modify the endless while(1) loop of main! Recall:

    i. The Core Timer T0 requests an interrupt every 50 milliseconds.

    ii. The Watchdog Timer is alive! It will trigger a reset after 33ns*512*256*WDPS. If WDPS was initialized to 64 this reads as 280ms.

    iii. Timer T0 Interrupt Service Routine increment variable "CpuTimer0.InterruptCount" every 50 milliseconds

    We have to reset the watchdog every 200 milliseconds and we should play the next note after 500 milliseconds. Two tasks within this while(1) loop. Later we will learn that this type of multi tasking is much better solved with the help of "DSP/BIOS" – Texas Instruments Real Time Operating System. For now we have to do it by our self.

    How can we find out, if a period of 200ms is over?

    We just have to test if "CpuTimer0.InterruptCount" is a multiple of 4. In language C this could be done by modulo division with 4 → reminder is zero:

    if ((CpuTimer0.InterruptCount%4)==0)

    If it is TRUE then we have to perform the second half of the watchdog re-trigger sequence:

    EALLOW;

    SysCtrlRegs.WDKEY = 0xAA;

    EDIS;

    In a similar technique we can wait for 10 times 50 ms = 500 ms before we apply the next note into T1PR and T1CMPR. In Lab4 we did a reset of variable "CpuTimer0.InterruptCount" every time a period was over. Doing so, we limited the values for this variable between 0 and 3, which was fine for this single task exercise. When we have to take care of more activities with different periods, it is not a good recommendation to reset this variable. A better approach is to build a time interval out of two read operations of "CpuTimer0.InterruptCount". With the first access we gather the actual time and store this value in a local unsigned long variable

"time_stamp". With a second access to "CpuTimer0.InterruptCount" we can read the new time information and the difference between this value and the value of "time_stamp" is the elapsed time in multiples of 50ms.

A wait instruction for 500ms could now look like this:

if ((CpuTimer0.InterruptCount – time_stamp) > 10)

If TRUE, then:

- Load "time_stamp" with "CpuTimer0.InterruptCount"
- Load the next note into EvaRegs.T1PR
- Load EvaRegs.T1CMPR = EvaRegs.T1PR/2
- Enable T1PWM, set EvaRegs.T1CON.bit.TENABLE = 1
- Implement and handle a status counter (variable i) to loop through array "frequency[8]

Build and Load

15. Click the "Rebuild All" button or perform:

    Project → Build

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

16. Load the output file down to the DSP Click:

    File → Load Program and choose the desired output file.

Test

17. Reset the DSP by clicking on:

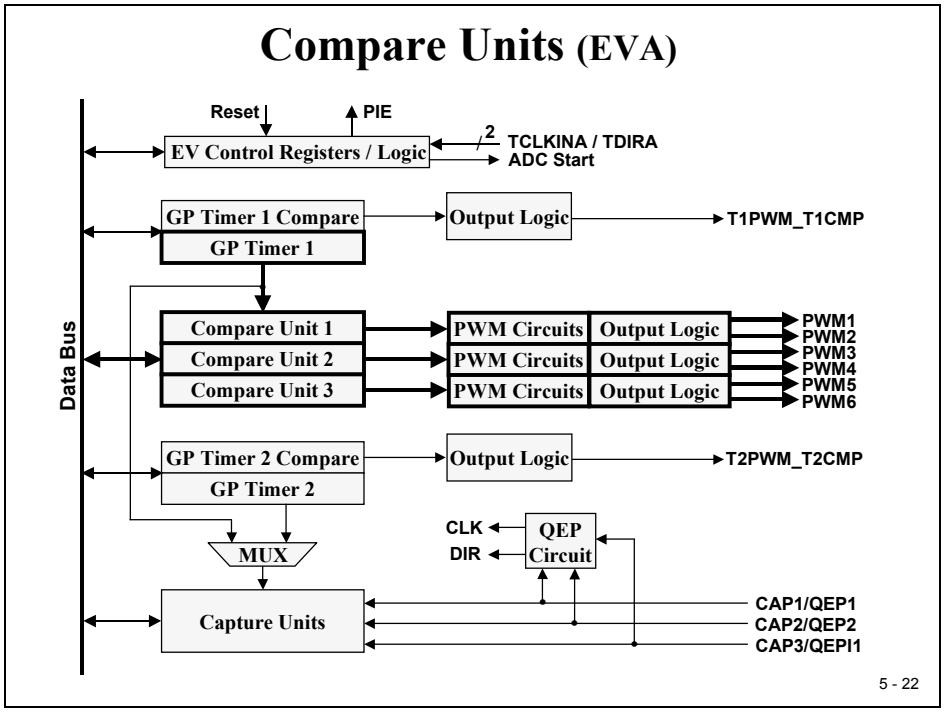
    Debug → Reset CPU followed by
    Debug → Restart

18. Run the program until the first line of your C-code by clicking:

    Debug → Go main.

19. Debug your code as you've done in previous labs.

Event Manager Compare Units



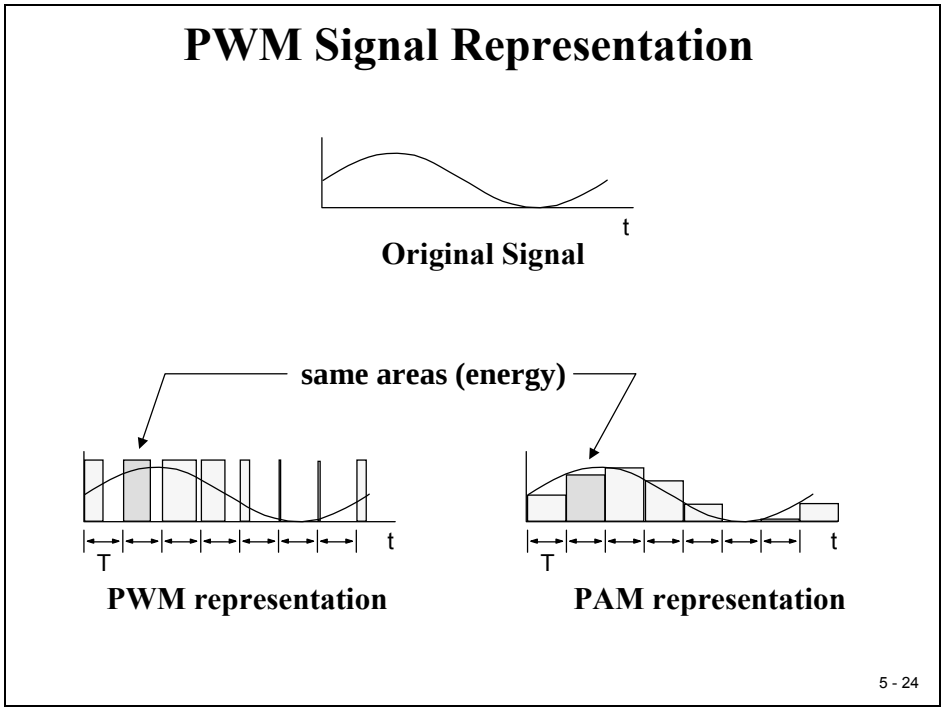
A compare unit is a peripheral that is designed to generate pulse width modulated (PWM) output signals. What is a PWM signal and what is it used for?

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 ≈ original signal energy**
- **Differs from PAM (Pulse Amplitude Modulation)**
  - **fixed width, variable amplitude**

5 - 23

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.



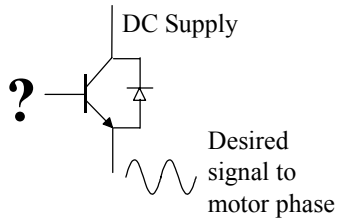
A lot of practical applications have an internal integrator, for example the windings of an electrical motor are perfectly suited to behave as a low-pass filter.

One of the most used applications of PWM is digital motor control. Why is that? Answer: The overall goal is to control electrical drives by imprinting 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. Problem is, for high currents we can't force the transistor into its linear area; this would generate a lot of thermal losses and for sure exceed its maximal power dissipation.

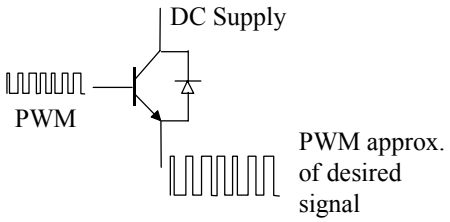
The solution is to use this transistor in its static switch states only (On: $I_{ce} = I_{cesat}$, Off: $I_{ce} = 0$). In this states a transistor has its smallest power dissipation. AND: by adapting the switch pattern of a PWM (recall: amplitude is 1 or 0 only) we can induce a sine wave shaped current!

Why Use PWM in Digital Motor Control?

- **Desired motor phase 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 Gate Signal

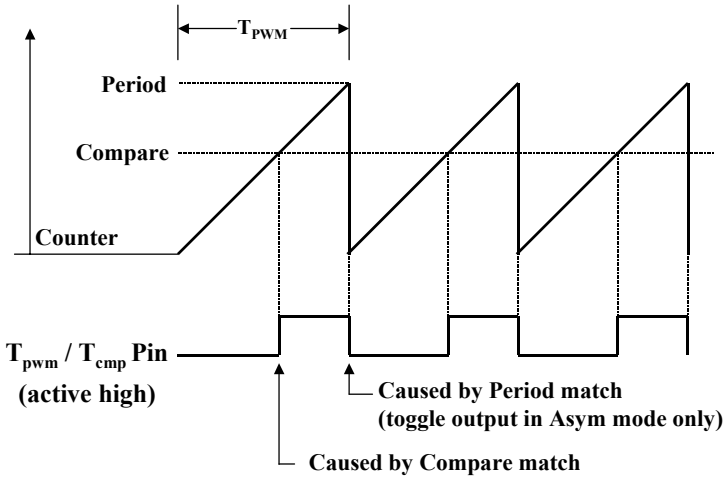


Gate Signal Known with PWM

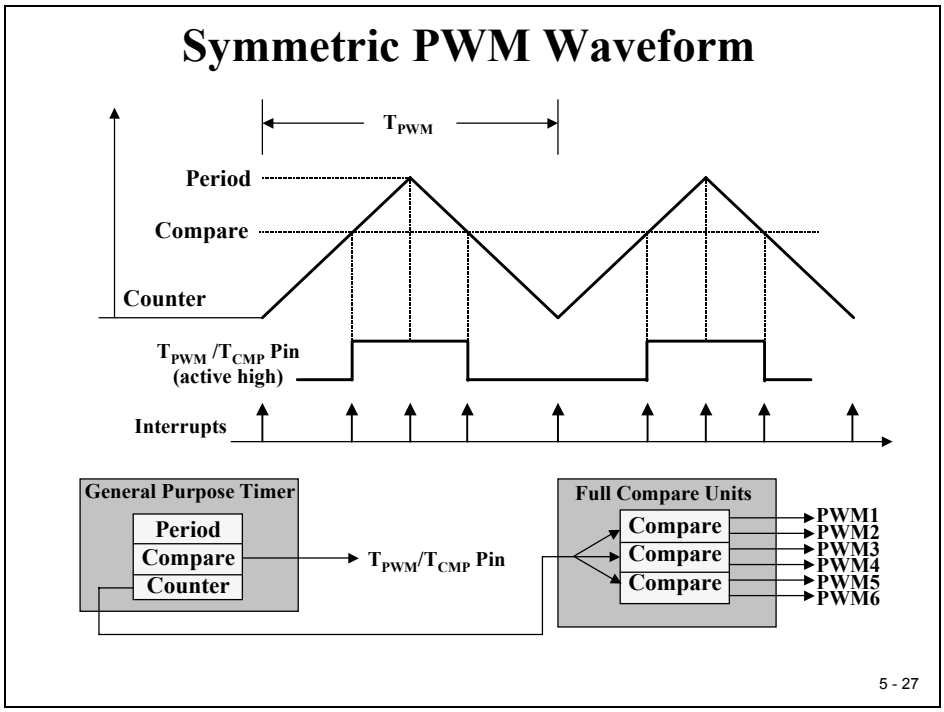
5 - 25

We have two different options to generate a PWM-signal, asymmetric and symmetric PWM.

Asymmetric PWM Waveform



5 - 26



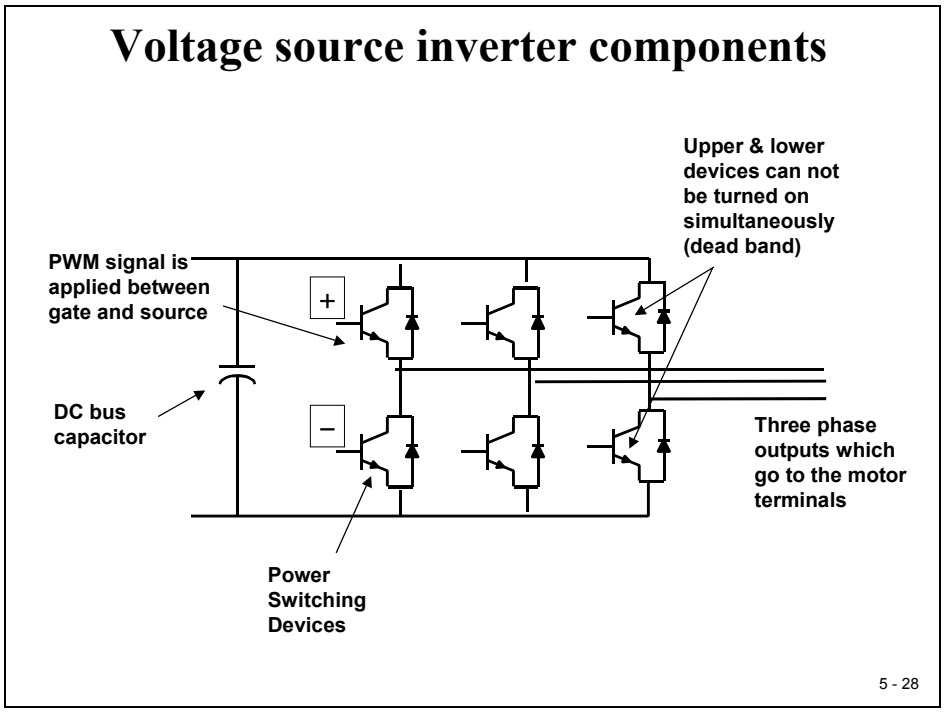
NOTE: The value in T1PR defines the length of a period “TPWM” in asymmetric operating mode. For symmetric mode the value of TxPR defines only half of the length of a period “TPWM”.

The Compare Unit consists of 6 output signals “PWM1” to “PWM6”. The time base is derived from Event Manager Timer1, e.g. register “T1PR” together with the setup for T1 (Register “T1CON”) defines the length of a PWM-period for all six output signals. Register “T1CNT” is used as the common counter register.

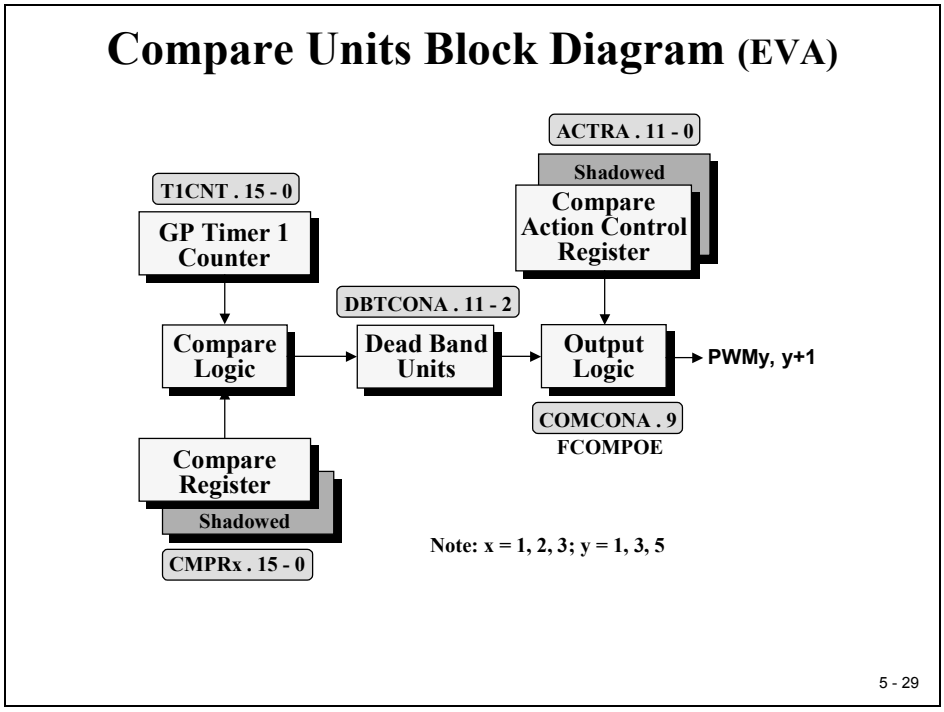
With 3 new registers “CMPR1”,”CMPR2”and “CMPR3” we can specify 3 different switch pattern based on T1PR. Obviously this leads to a 3-phase control pattern for 3 phase electrical motors.

Each Compare Unit is able to drive a pair of two output signals. With the help of its own output logic we usually define the two lines to be opposite or 180-degree out of phase to each other - a typical pattern for digital motor control.

The next slide shows a typical layout for a three-phase power-switching application.



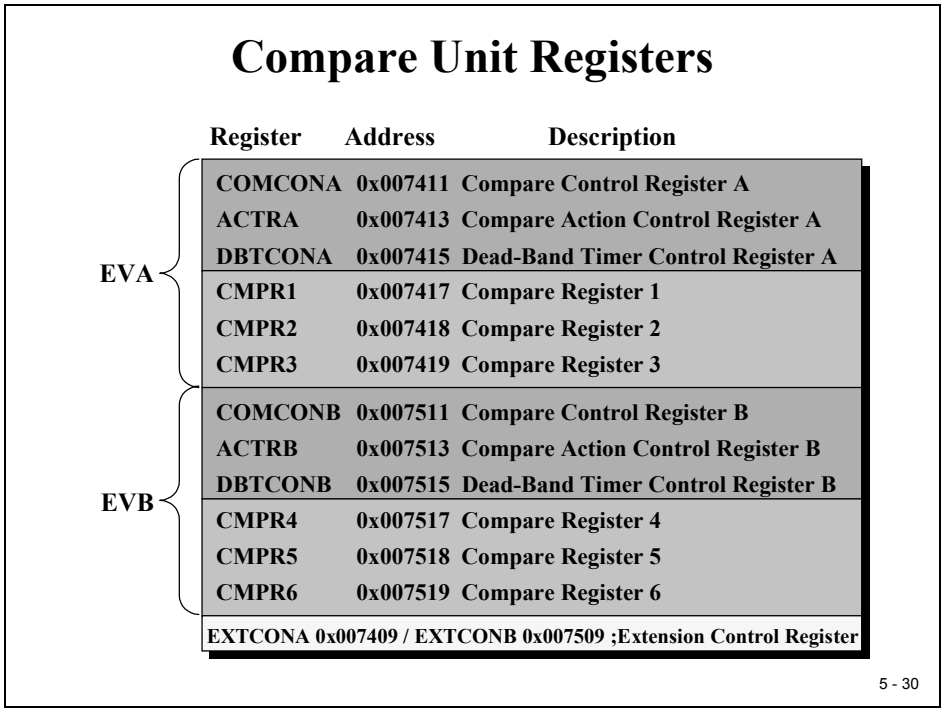
Compare Units Block Diagram



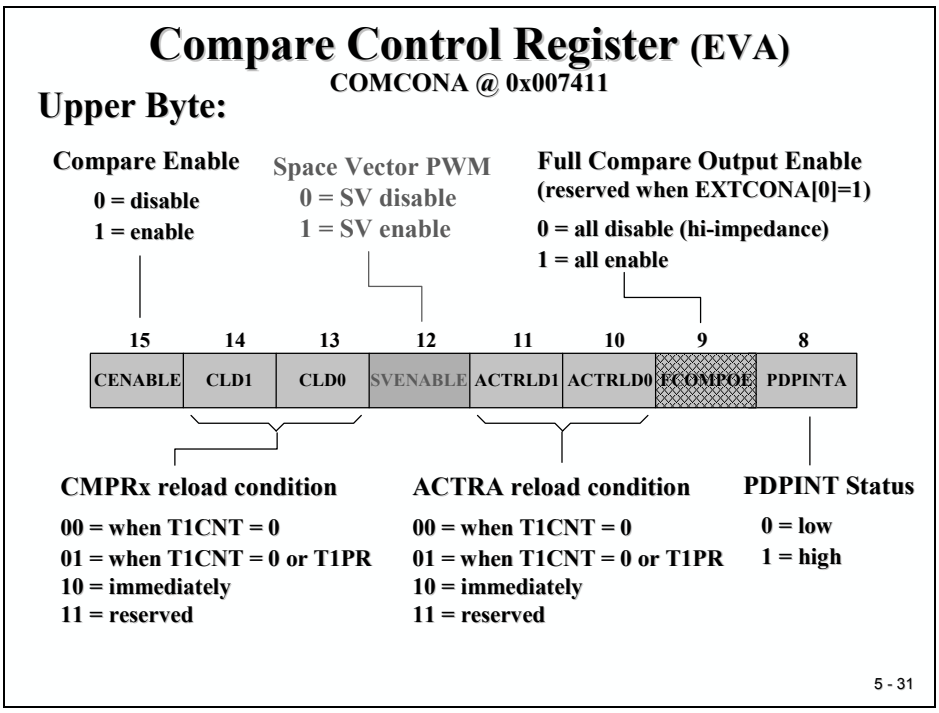
The central block of the Compare Unit is a compare logic that compares the value of Event Manager Timer 1 counter register “T1CNT” against Compare Register “CMPRx”. If there is a first match, a rising edge signal goes into the next block called “Dead Band Unit”. With the second match between “T1CNT” and “CMPRx” in symmetric PWM mode a falling edge signal is generated. We will discuss this “Dead Band Unit” a little bit later. We do have three Compare Units available.

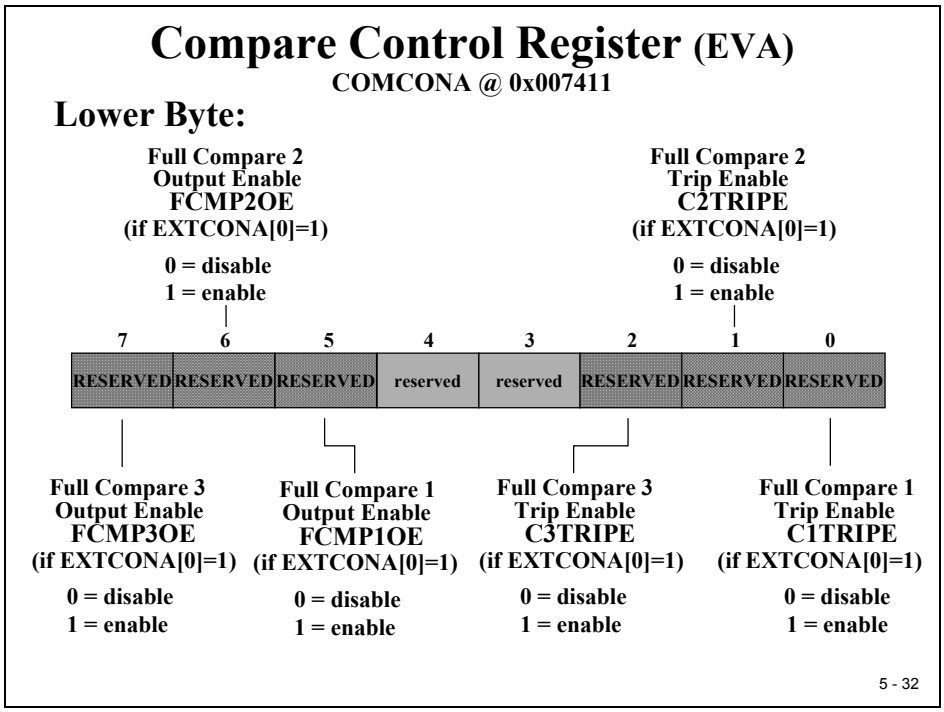
The output logic is controlled by means of a register, called “Action Control Register – ACTRA” and register “COMCONA”. With the help of this register set we can adjust the shape of the physical PWM output signal to our needs. We can specify four types for all 6 output lines:

- Active High:
    - First CMPRx match switches PWM output from 0 to 1. After second CMPRx match the signal is set back to 0.
- Active Low:
    - First CMPRx match switches PWM output from 1 to 0. After second CMPRx match the signal is setback to 1.
- Forced High:
    - PWM output always at 0.
- Forced Low:
    - PWM output always at 1.



The next two slides explain the set up for the individual bit fields of COMCONA. Most of the bits are reserved in basic operation mode (EXTCONA [0] = 0).





COMCONA [15] is the enable bit for the three phase compare units.

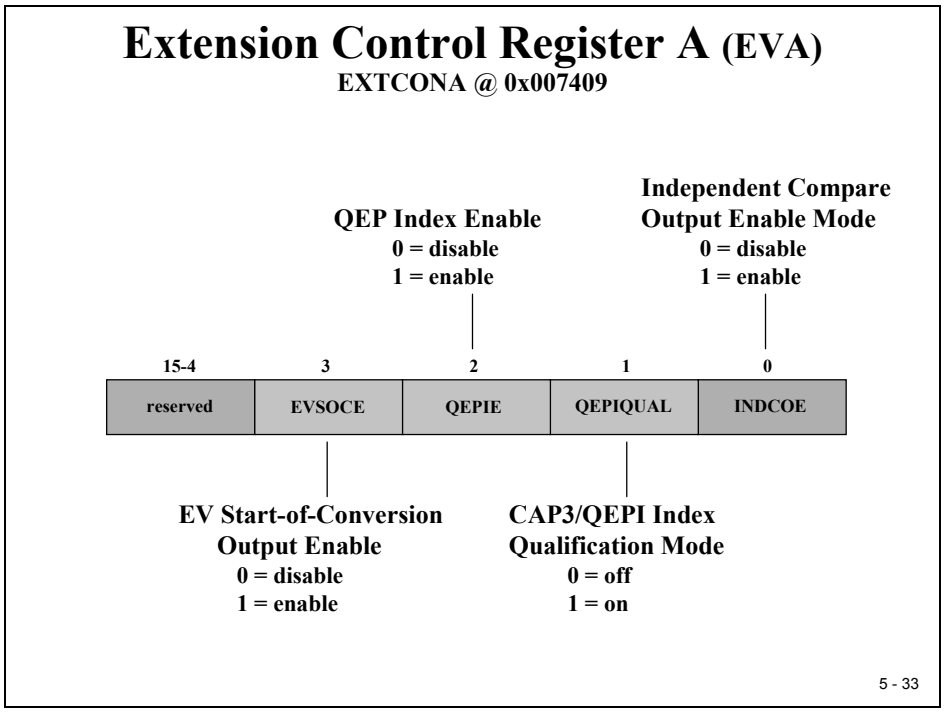
With COMCONA [14:13] and COMCONA [11:10] we specify the point in time when the compare registers and action control registers are reloaded (shadow register content into foreground). As we have seen with the timers we can prepare the next period in the current running period.

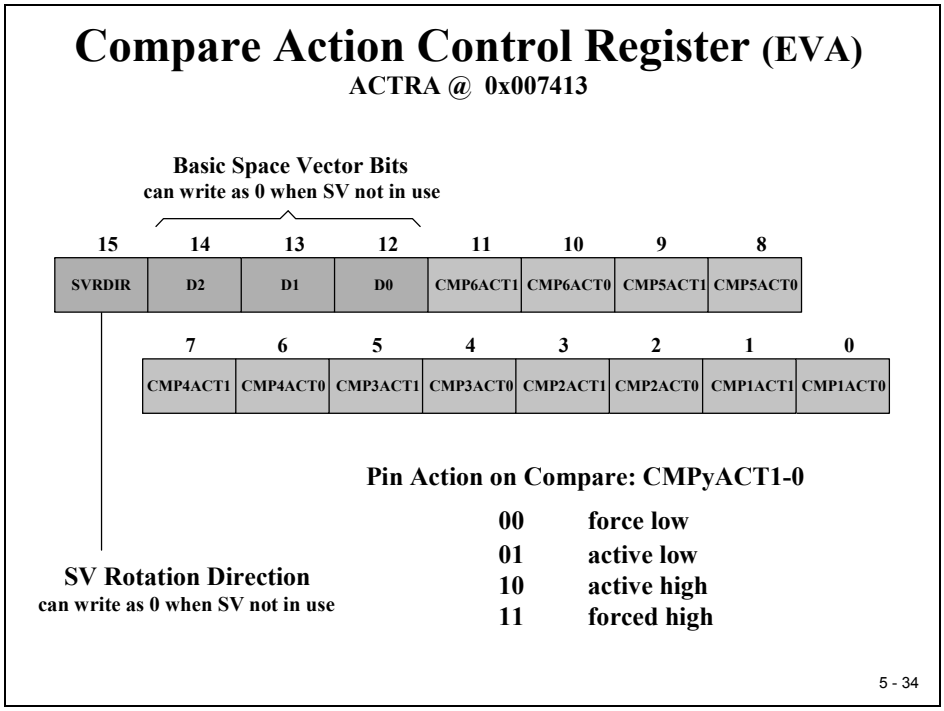
COMCONA [8] shows the status of the power drive protection flag. If it is a 1, the DSP has seen an interrupt request from its over-current input PDPINT.

With EXTCONA [0] =1 all three pairs of compare output lines can be enabled independently of each other.

If EXTCONA [0] =0 then all six lines are enabled with COMCONA [9] =1. If EXTCONA [0] =1 we can use three more individual over current inputs signals. To use these over current signals we can enable or disable this feature using bits COMCONA [2:0].

COMCONA [12] = 1 enables a special switch pattern for digital motor control, called “Space Vector Modulation” (SVM). This feature is built-in hardware support for one specific theoretical control algorithm. For details see literature or your lectures about power electronics.





ACTRA [11:0] define the shapes of the six PWM output signals as discussed before.

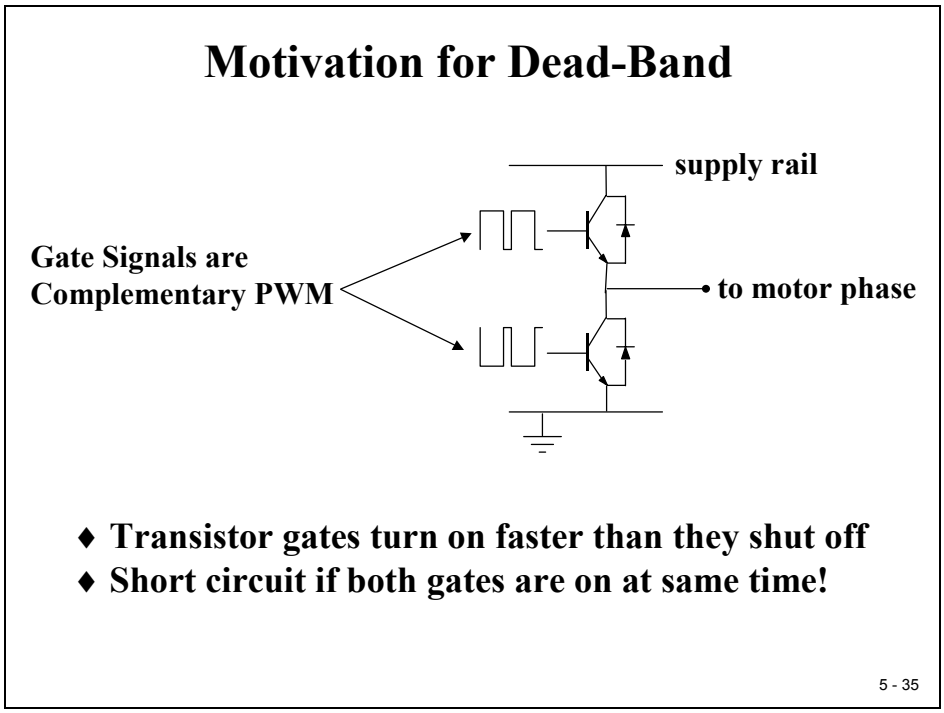
ACTRA [15:12] are used to support Space Vector Modulation. Bit 15 defines the rotation direction of the resulting electromagnetic vector as clockwise or anti clockwise.

ACTRA [14:12] declare the Basic Space Vector for the next PWM periods. A Basic Space Vector is a 60-degree section of the unit circle. This gives 6 vectors per rotation plus two virtual vectors with no current imprint.

If SVM is not used, one can initialize bits 15:12 to zero.

Hardware Dead Band Unit

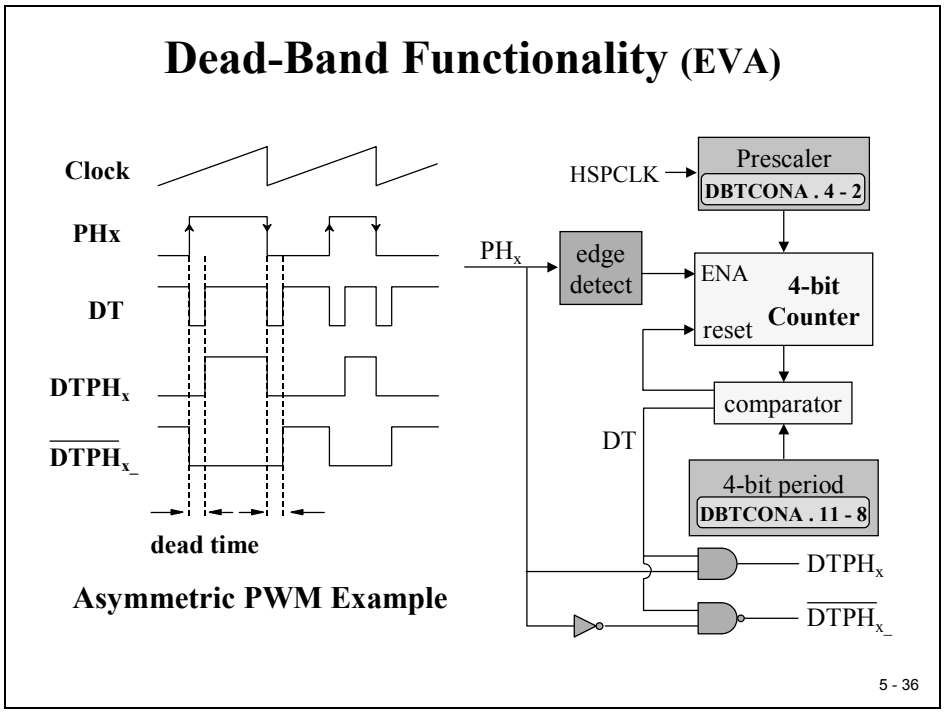
Dead-band control provides a convenient means of combating current “shoot-through” problems in a power converter. “Shoot-through” occurs when both the upper and lower transistors in the same phase of a power converter are on simultaneously. This condition shorts the power supply and results in a large 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.



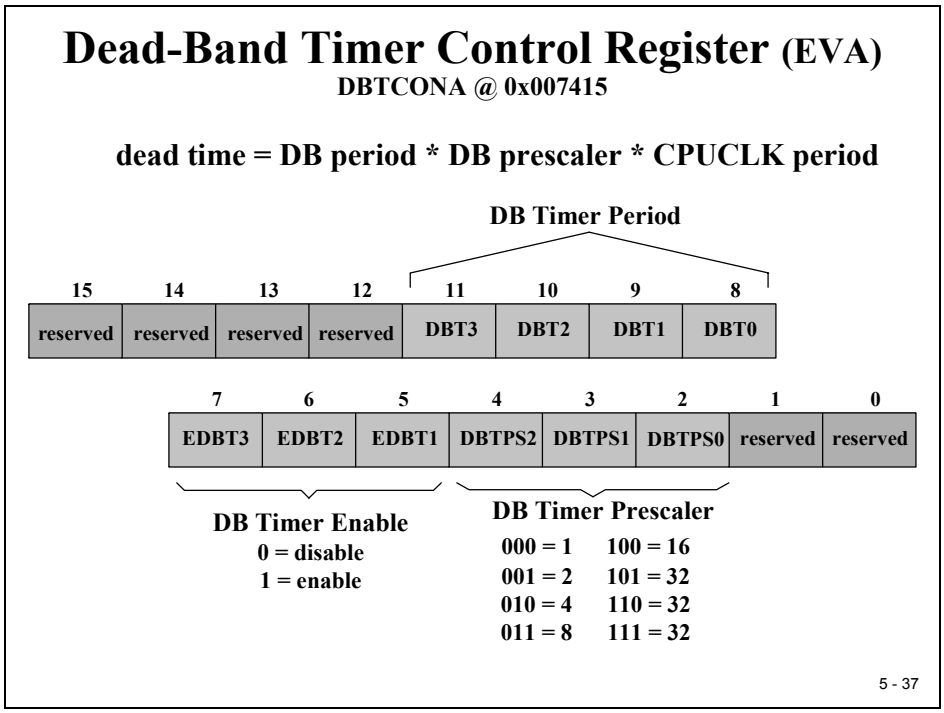
Two basic approaches exist for controlling shoot-through: modify the transistors, or modify the PWM gate signals controlling the transistors. In the first case, the switch-on time of the transistor gate must be increased so that it (slightly) exceeds the switch-off time.

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.

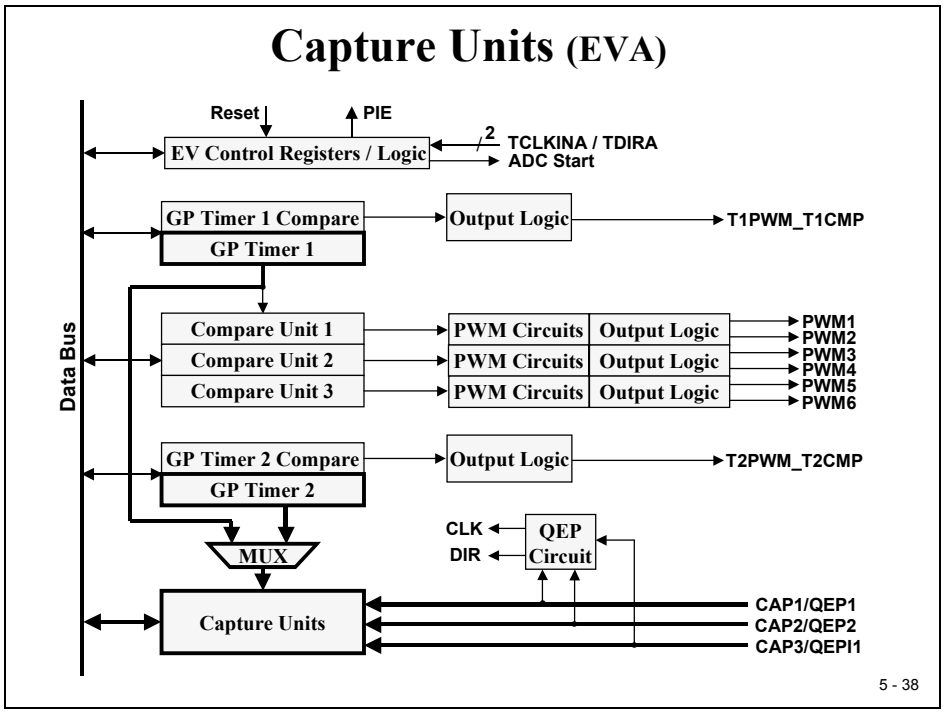
The second approach to shoot-through control separates transitions on complimentary PWM signals with a fixed period of time. This is called dead-band. While it is possible to perform software implementation of dead-band, the C28x offers on-chip hardware for this purpose that requires no additional CPU overhead. Compared to the passive approach, dead-band offers more precise control of gate timing requirements.



Each compare unit has a dead-band timer, but shares the clock prescaler unit and the dead-band period with the other compare units. Dead-band can be individually enabled for each unit.



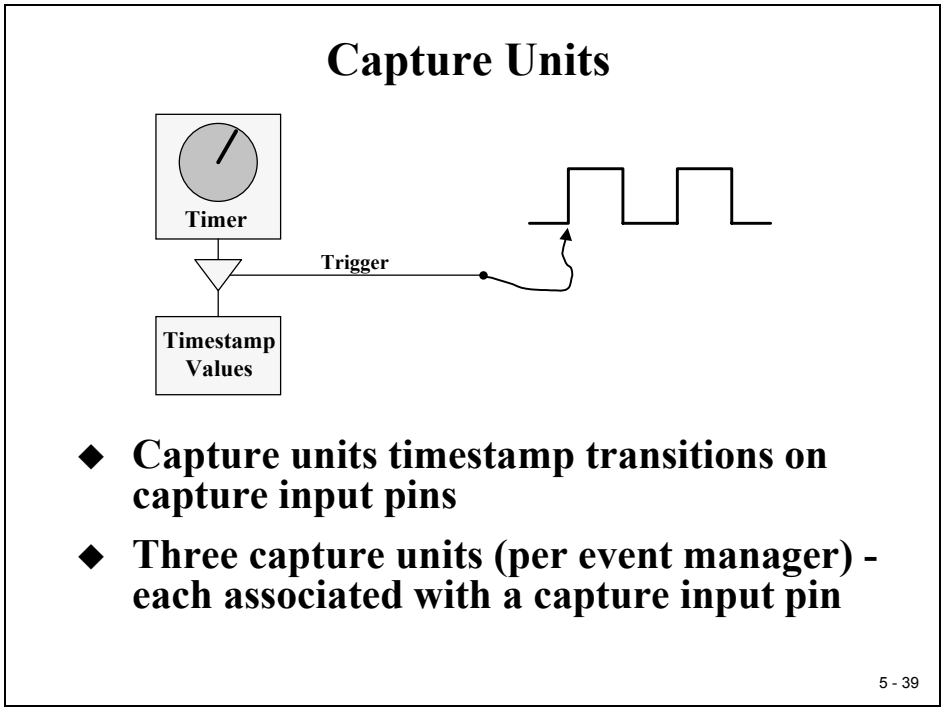
Capture Units



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

Event Manager A has three capture units, and each is associated with a capture input pin. The time base is selectable to be either GP timer 1 or 2. The timer value is captured and stored in the corresponding 2-level-deep FIFO stack when a specified transition is detected on a capture input pin.

Capture Unit 3 can be configured to trigger an A/D conversion that is synchronized with an external signal of a capture event.



Three potential uses for the Capture Units are:

- Measurement of the width of a pulse or a digital signal
- Automatic start of the AD – Converter by a Capture Event from CAP3
- Low speed estimation of a rotating shaft. A potential advantage for low speed estimation is given when we use “time capture” (16Bit resolution) instead of position pulse counting (poor resolution in slow mode).

Some Uses for the Capture Units

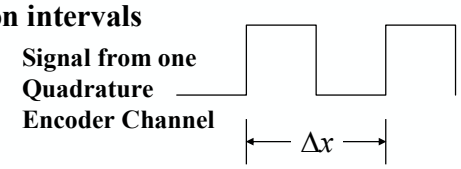
- Synchronized ADC start with capture event
- Measure the time width of a pulse
- Low speed velocity estimation from incr. encoder:

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

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

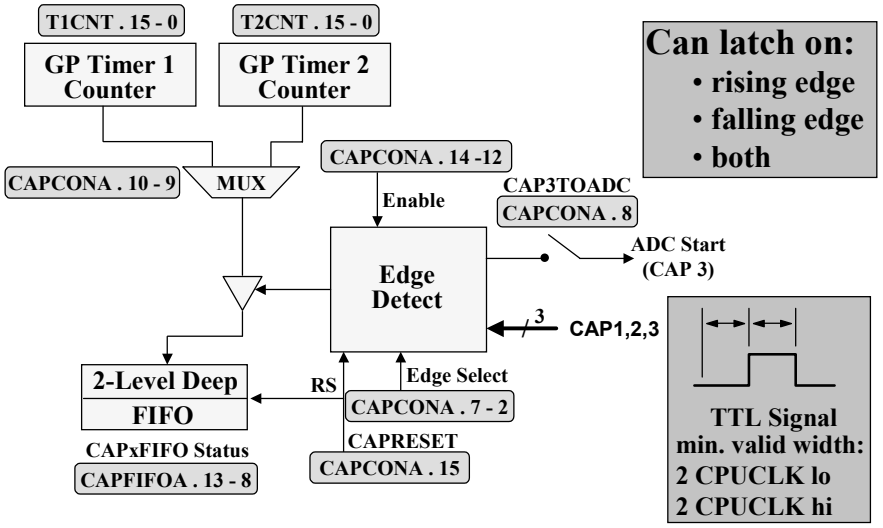


5 - 40

Capture Units Block Diagram

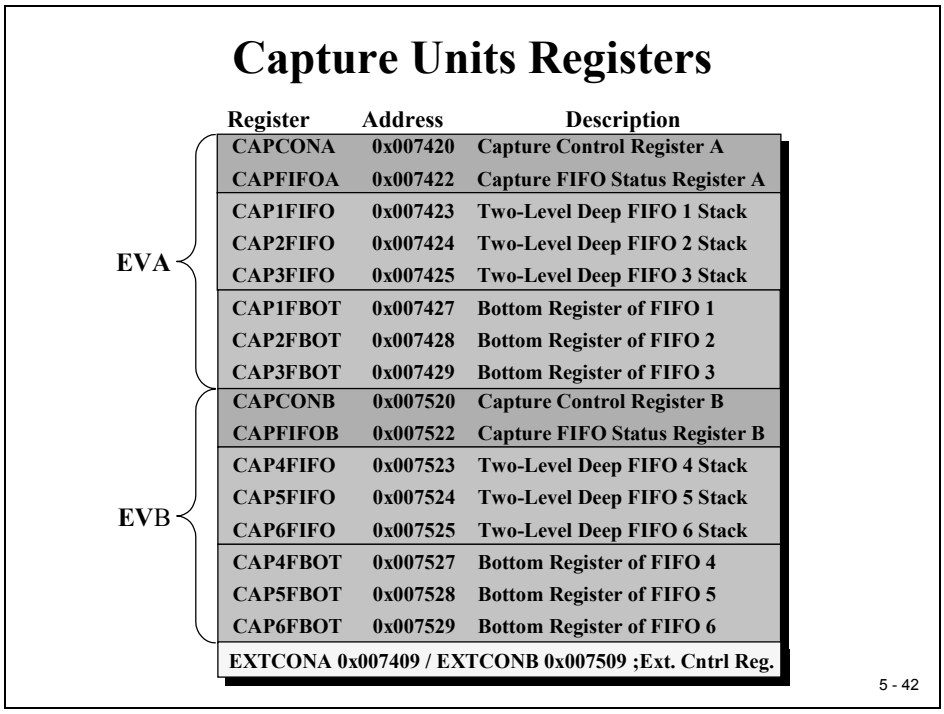
The edge detector stores the current value of the time base counter into a FIFO-buffer.

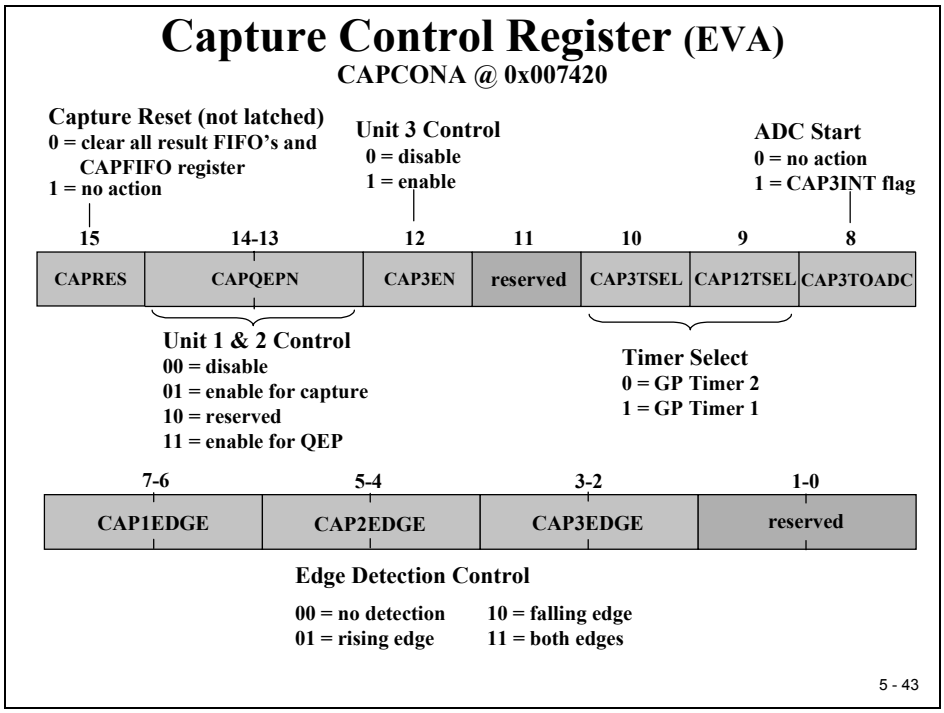
Capture Units Block Diagram (EVA)



5 - 41

Capture Units Registers





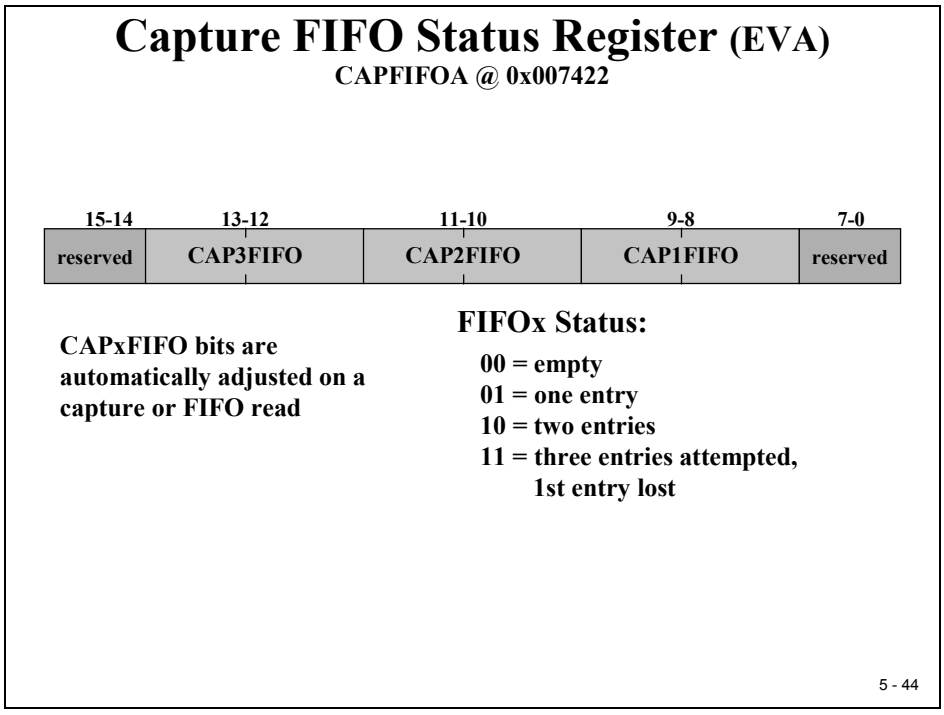
CAPCONA [15] is a reset bit for the Capture state machine and the status of the FIFO. It should be used in a single instruction to reset the Capture units during initialization. Note: to execute reset you will have to apply a zero!

With CAPCONA [14-12] the Capture Units are enabled. Please note that CAP1 and CAP2 are enabled jointly, whereas CAP3 has its own enable bit.

CAPCONA [10-9] are used to select the clock base for the capture units. Again, for CAP1 and CAP2 we have to select the same GP timer.

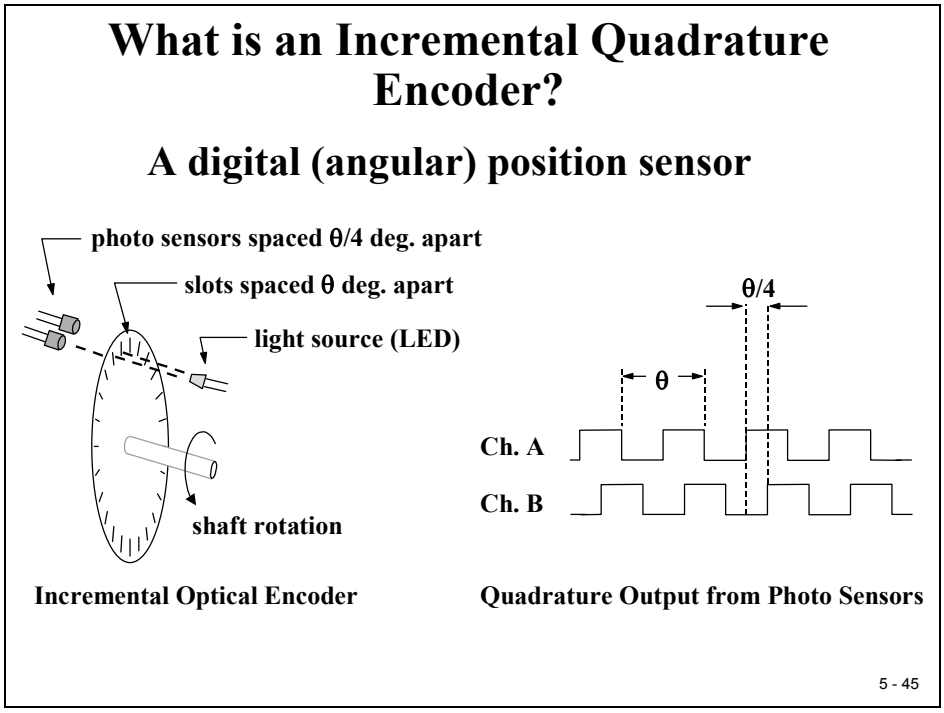
CAPCONA [8] allows CAP3 to start an AD conversion. Of course, before we use this option, we have to initialize the ADC. This will be explained in the next chapter.

CAPCONA [7-2] specify if the capture units are triggered with a rising or falling edge or with both edges.

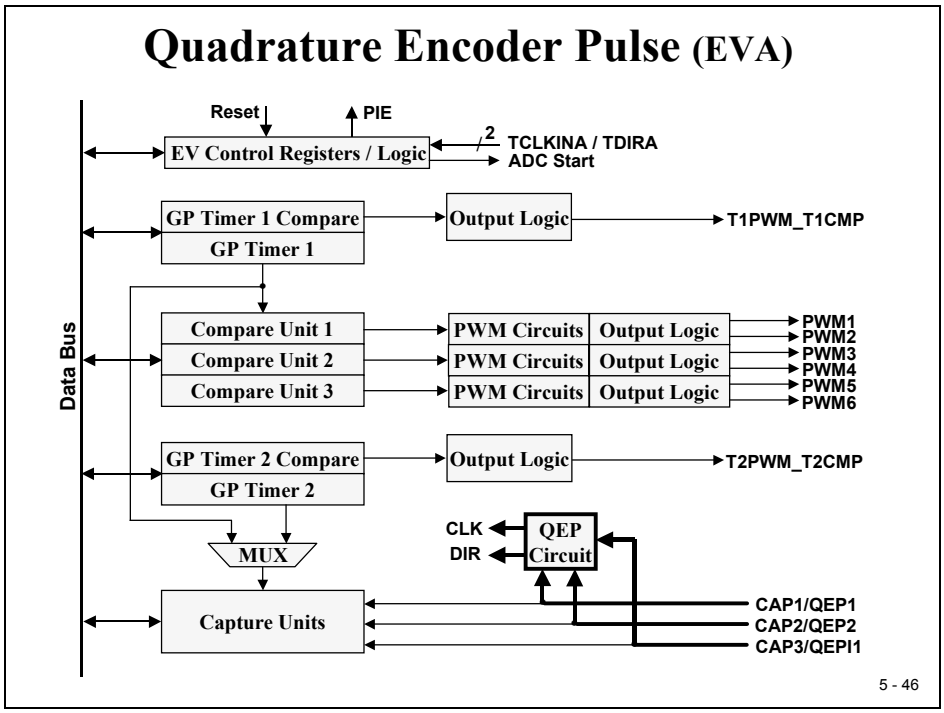


Register CAPFIFOA reflects the filling status of the three result register FIFO's. In case of an overflow the oldest entry will be lost. This principle ensures that a capture unit stores the two latest measurement results. If our program performs a read access to one of the FIFO result registers the status value in the corresponding CAPFIFOA bit field is decremented.

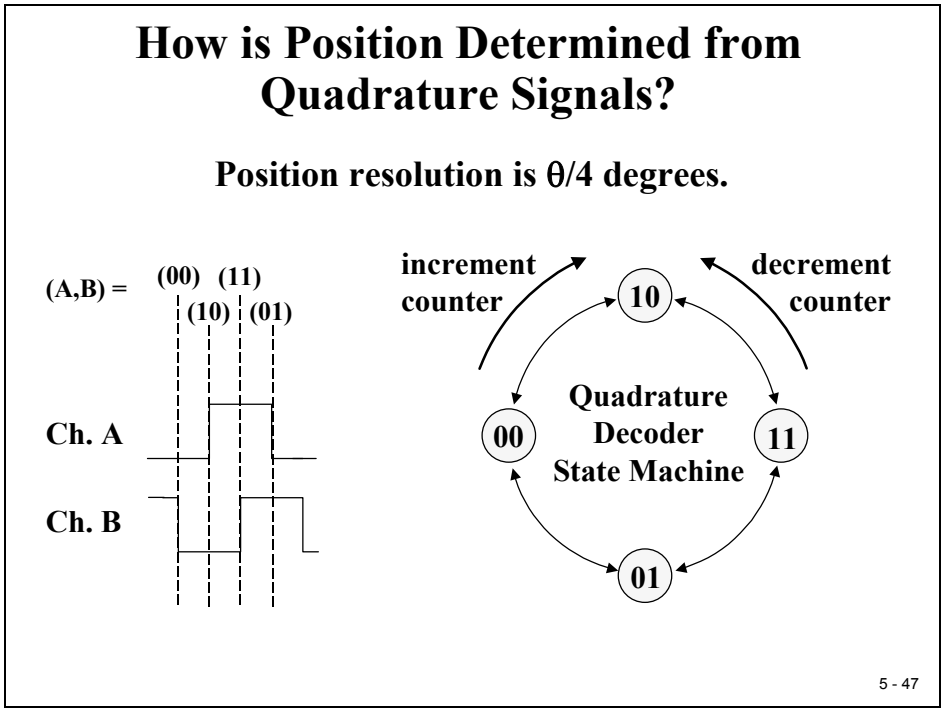
Quadrature Encoder Pulse Unit (QEP)

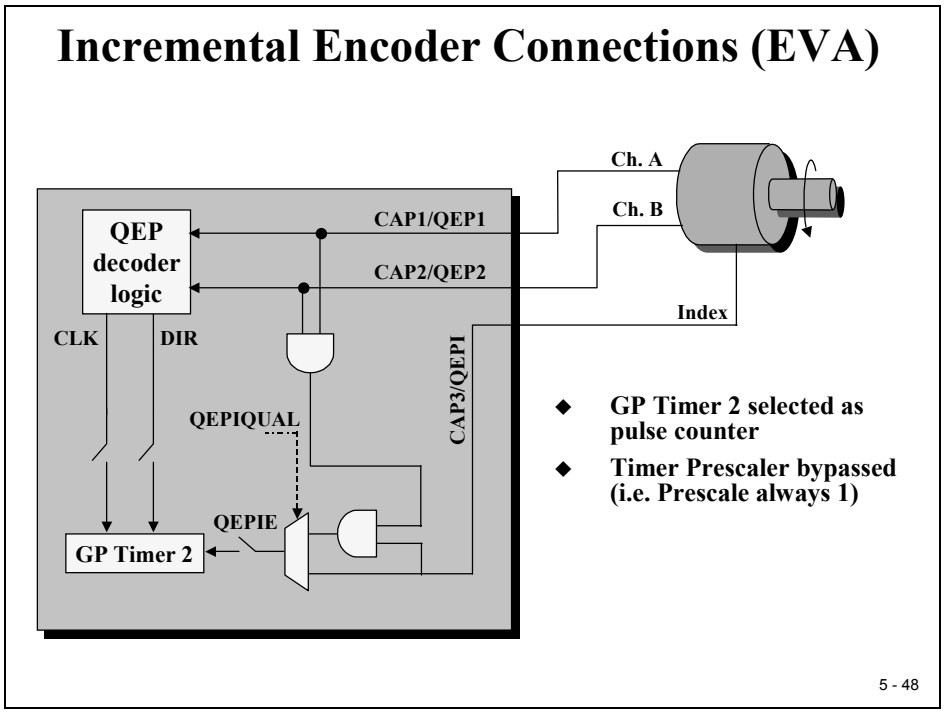


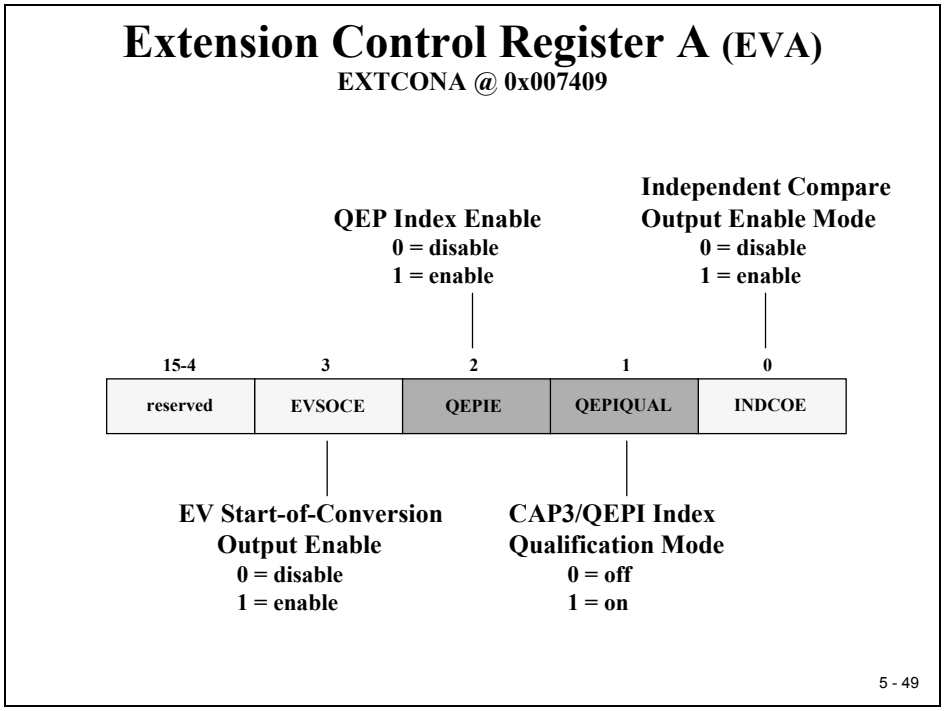
A QEP unit is normally used to derive direction and speed information from an incremental encoder circuit mounted on a rotating shaft. As shown on the previous slide two sensor signals are used to generate two digital pulse streams “Channel A” and “Channel B”.



The time relationship between A and B lead to a state machine with four states. Depending on the sequence of states and the speed of alternation, the QEP unit timer is decremented or incremented. By reading and comparing this timer counter information at fixed intervals, we can obtain speed and/or position information.







The third capture input pin “QEPI1”can be used as an absolute position information signal for a zero degree crankshaft position. This signal is then used to reset the QEP timer to its initial state. To enable the “QEPI1” index function we have set EXTCONA [2] to 1. Then we have two more options, selected with EXTCONA [1]:

- Use index pulse “QEPI1” independent from the state of QEP1 and QEP2
- Use index pulse “QEPI1” as a valid trigger pulse only if this event is qualified by the state of QEP1 = 1 AND QEP2 = 1.

Lab 5A: Generate a PWM sine wave

Objective

So far we generated square wave signals to drive a loudspeaker. To ask a musician to listen to this type of music would be impudent. So let's try to improve the shape of our output signals. A musical note is a pure - or harmonic – sine wave signal of a fixed frequency. The objective of this lab exercise is to generate a harmonic sine wave signal out of a series of pulse width modulated digital pulses (PWM).

Remark: The first generation of cell phones used the square wave technology to generate ringing sounds. Compare this with today's latest cell phones!

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab5A.pjt** in E:\C281x\Labs.
2. Open the file Lab5.c from E:\C281x\Labs\Lab5 and save it as Lab5A.c in E:\C281x\Labs\Lab5A.
3. Add the source code file to your project:
   - **Lab5A.c**
4. From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source* add:
   - **DSP281x_GlobalVariableDefs.c**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\cmd* add:
   - **F2812_EzDSP_RAM_lnk.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\cmd* add:
   - **F2812_Headers_nonBIOS.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add to project:
   - **DSP281x_PieCtrl.c**
   - **DSP281x_PieVect.c**
   - **DSP281x_DefaultIsr.c**

   From *C:\ti\c2000\cgtoolslib* add:

- **rts2800_ml.lib**

Project Build Options

5. Setup the search path to include the peripheral register header files. Click:

   Project → Build Options

   Select the Compiler tab. In the preprocessor Category, find the Include Search Path (-i) box and enter:

   C:\tidcs\C28\dsp281x\v100\DSP281x_headers\include;
   ..\include; C:\tidcs\C28\IQmath\cIQmath\include

6. Setup the stack size: Inside Build Options select the Linker tab and enter in the Stack Size (-stack) box:

   400

   Close the Build Options Menu by Clicking <**OK**>.

Modify Source Code

7. Open Lab5A.c to edit: double click on "Lab5A.c" inside the project window. First we have to cancel the parts of the code that we do not need any longer. We will not use the CPU core timer 0 in this exercise; therefore we do not need the prototype of interrupt service routine "cpu_timer0_isr()". Instead, we need a new ISR for GP Timer1-Compare-Interrupt. Add a new prototype interrupt function: "interrupt void T1_Compare_isr(void)".

8. We do not need the variables "i","time_stamp" and frequency[8]" from Lab5 - delete their definition lines at the beginning of function "main".

9. Next, modify the re-map lines for the PIE entry. Instead of "PieVectTable.TINT0 = & cpu_timer0_isr" we need to re-map:

   PieVectTable.T1CINT = &T1_Compare_isr;

10. Delete the next two function calls: "InitCpuTimers();" and "ConfigCpuTimer(&CpuTimer0, 150, 50000);" and add an instruction to enable the EVA – GP Timer1 – Compare interrupt. Recall Module 4 "Interrupt System" and verify that the EVA – GP Timer1 – Compare interrupt is connected to PIE Group2 Interrupt 5. Which Register do we have to initialize? Answer:

    PieCtrlRegs.PIEIER2.bit.INTx5 = 1;

    Also modify the set up for register IER into:

    IER = 2;

11. Next we have to initialize the Event Manager Timer 1 to produce a PWM signal. This involves the registers “GPTCONA”, “T1CON”, “T1CMPR” and “T1PR”.

    For register “GPTCONA” it is recommended to use the bit-member of this predefined union to set bit “TCMPOE” to 1 and bit field “T1PIN” to “active low”.

    For register “T1CON” set

    - The “TMODE”-field to “counting up mode”;
    - Field “TPS” to “divide by 1”;
    - Bit “TENABLE” to **“disable timer”;**
    - Field “TCLKS” to “internal clock”
    - Field “TCLD” to “reload on underflow”
    - Bit “TECMPR” to “enable compare operation”

12. Remove the 3 lines before the while(1)-loop in main:

    - “CpuTimer0Regs.TCR.bit.TSS = 0;”
    - “i = 0;”
    - “time_stamp = 0;”

    and add 4 new lines to initialise T1PR, T1CMPR, to enable GP Timer1 Compare interrupt and to start GP Timer 1:

    ```
    EvaRegs.T1PR = 1500;

    EvaRegs.T1CMPR = EvaRegs.T1PR/2;

    EvaRegs.EVAIMRA.bit.T1CINT = 1;

    EvaRegs.T1CON.bit.TENABLE = 1;
    ```

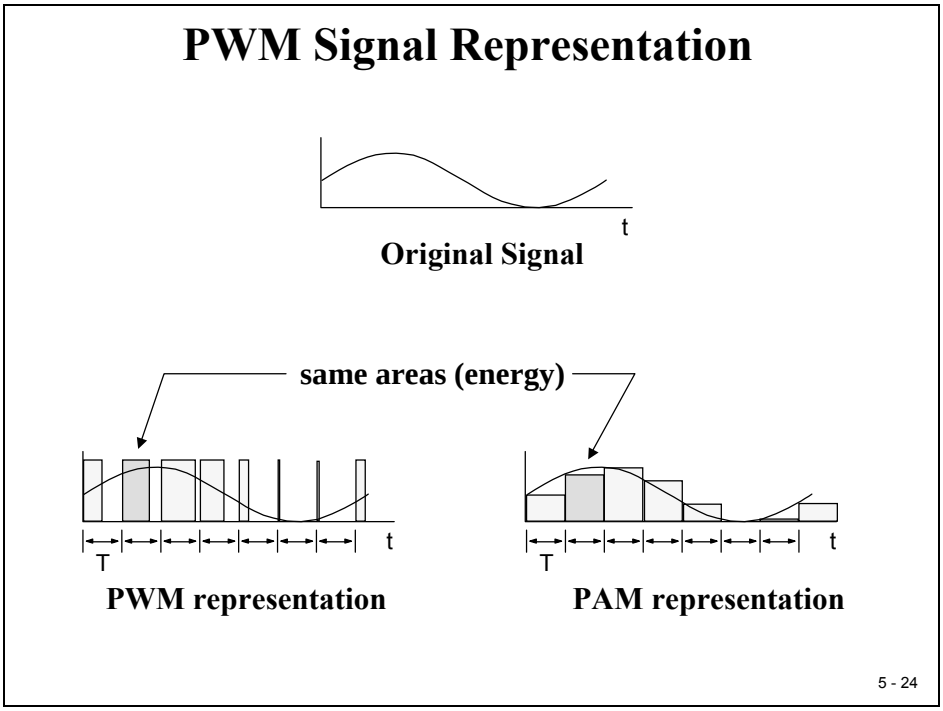
    What is this number 1500 for? Well, it defines the length of a PWM period:

    $$f_{PWM} = \frac{f_{CPU}}{T1PR \cdot TPS_{T1} \cdot HISCP}$$

    with TPS_{T1}=1, HISCP = 2, f_{CPU} = 150MHz and a desired f_{PWM} = 50kHz we derive: T1PR = 1500!

    T1CMPR is preloaded with half of T1PR. Why’s that? Well, in general T1CMPR defines the width of the PWM-pulse. Our start-up value obviously defines a pulse width of 50%.

Recall slide 5-24: “PWM-Representation”



A duty cycle of 50% represents a sine angle of 0 degrees! And, it makes sense to initialize the PWM unit for this angle. From the bottom left of the slide we can derive:

Degree	Sin	Duty – Cycle
0°	0	50%
90°	1	100%
180°	0	50%
270°	-1	0 %
360°	0	50%

13. Modify the endless while(1) loop of main! We will perform all activities using GP Timer 1 Compare Interrupt Service. Therefore we can delete almost all lines of this main background loop, we only have to keep the watchdog service:

```
while(1)
{
        EALLOW;
        SysCtrlRegs.WDKEY = 0xAA;
        EDIS;
}
```

14. Rename the interrupt service routine "cpu_timer0_isr" into "T1_Compare_isr". Remove the line "CpuTimer0.InterruptCount++;" and replace the last line of this routine by:

```
PieCtrlRegs.PIEACK.all = PIEACK_GROUP2;
```

Before this line add another one to acknowledge the GP Timer 1 Compare Interrupt Service is done. Remember how? The Event Manager has 3 interrupt flag registers "EVAIFRA","EVAIFRB" and "EVAIFRC". We have to clear the T1CINT bit (done by setting of the bit):

```
EvaRegs.EVAIFRA.bit.T1CINT = 1;
```

Build and Load

15. Click the "Rebuild All" button or perform:

Project → Build

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

16. Load the output file down to the DSP Click:

File → Load Program and choose the desired output file.

Test

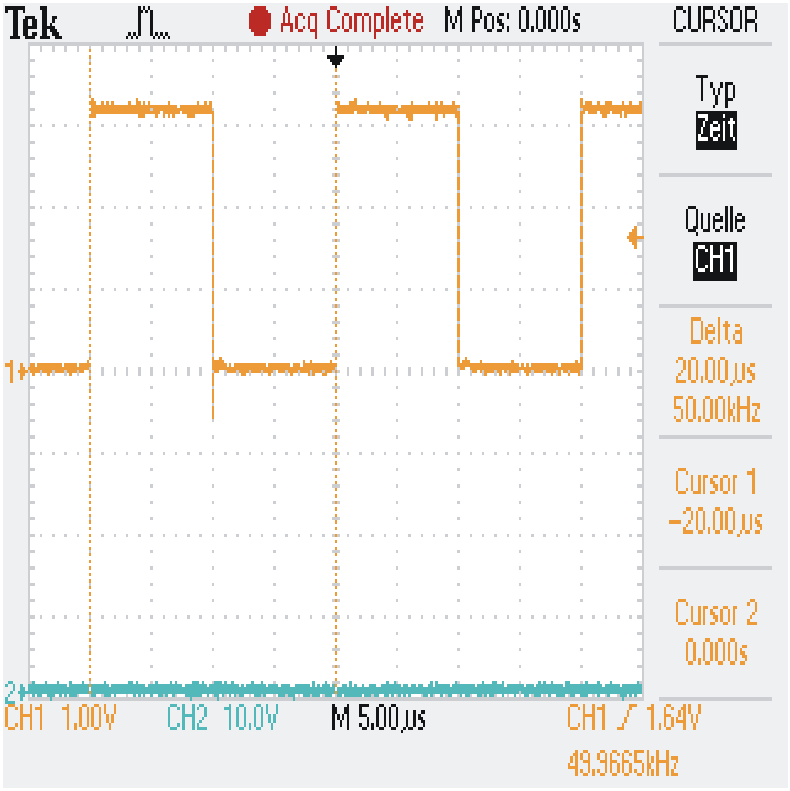
17. Reset the DSP by clicking on:

Debug → Reset CPU followed by
Debug → Restart and
Debug → Go main.

18. When you now run the code the DSP should generate a 50 kHz PWM signal with a duty cycle of 50% on T1PWM. If you have an oscilloscope you can use jumper JP7 (in front of the loudspeaker) of the Zwickau Adapter board to measure the signal.

    If your laboratory can't provide a scope, you can set a breakpoint into the interrupt service routine of T1 Compare at line "PieCtrlRegs.PIEACK.all = PIEACK_GROUP2; Verify that your breakpoint is hit periodically, that register T1PR holds 1500 and register T1CMPR is initialized with 750. Use the watch window to do so.

    Do not continue with the next steps until this point is reached successfully! Instead go back and try to find out, what went wrong during the modification of your source code.



Generate the sine wave

So far we generated a pure square wave PWM signal of 50 KHz. Our goal is to produce a sine wave signal which is build up from a series of this 50 KHz carrier period cycles. We have to modify the pulse width according to the current sine angle. Obviously we have to increment the angle from 0° to 360° in a couple of steps.

Question is: how do we calculate the sine-values?

- **Use the sin(x) function**

  This function is part of the C compilers "math.lib". All we would need to do is to add the header file "math.h" to our project. Problem: sin(x) is a floating-point function; our DSP is a fixed point processor. That means the compiler has to generate quite a lot of assembler instructions to calculate the sine values. This will cost us a quite a lot of CPU time, just to calculate the same series of sine values over and over.

  ➔ Feasible, but not recommended

- **Use a lookup table with pre calculated sine values**

  We do not need a floating-point precision; our goal is to adjust the 16-bit register "T1CMPR". It is much quicker to prepare an array with precalculated sine values. Instead of calculating the next value during runtime we access a table with pre-defined results of sine calculations. This principle is called "Lookup Table Access" and is widely used in embedded control. Almost all control units for automotive electronics are using one or more of these lookup tables, not only for trigonometric functions but also for control parameters.

  ➔ Highly recommended

Next Question:

How do we generate a lookup table? Well, use a calculator, note all results and type them into an array! How many values? Well, the more values we have the better we can approximate the analogue sine wave shape! Recall, we need sine values from 0° to 360°. Sounds like a lot of boring work, doesn't it?

Answer: Texas Instruments has already done the work for you. The C28x – DSP comes with a "BOOT-ROM" (see memory map – module 1). A part of this memory area is a sine wave table!

From Address 0x3F F000 to 0x3F F3FF we find 512 values for sin(x). The numbers are stored as 32 Bit – numbers in "Q30"-notation. With 512 entries we have an angle step of 0.703° (360°/512) for a unit circle.

But what is "Q30"?

"Q30" or "I2Q30" is a fractional fixed-point representation of 32-Bit numbers. We will discuss the advantage of fractional numbers for embedded control in detail in Part 2 of this DSP course. So far, let's try to understand the basics:

The data format "Q30" separates a 32-bit number into an integer part and a fractional part. The integer part is the usual sequence of positive powers of 2; the fractional part is the sequence of negative powers of 2. For a "Q30" number we get the following binary representation:

Bit 31	Bit 30	Bit 29	Bit 28	Bit 27	Bit 26	Bit 25
$(-1)*2^{1}$	$1*2^{0}$	$1*2^{-1}$	$1*2^{-2}$	$1*2^{-3}$	$1*2^{-4}$	$1*2^{-5}$

The decimal range of a "Q30" number is –2...+1.9999:

Most negative number:	0x8000 0000	-2
Decimal minus 1:	0xC000 0000	-1
Smallest negative number	0xFFFF FFFF	-9.31322e-10
Zero	0x0000 0000	0
Smallest positive number	0x0000 0001	+9.31322e-10
Decimal plus 1:	0x4000 0000	+1
Most positive number:	0x7FFF FFFF:	+1.999999999

IQ-Math Library

Texas Instruments has built a whole library of fixed-point math operations based on this "Q"-format. This library called "IQ-Math" is widely used in closed control applications like digital motor control, FAST FOURIER TRANSFORM (FFT) or digital filters (FIR, IIR). The library is free, no royalties and can be downloaded from TI's web. The appendix of this CD contains the current version of IQ-Math. We will discuss and use this library in a specific module in Part 2 of this DSP course.

Boot ROM Table

The following table shows the contents of the sine wave area of the Boot ROM:

Address	Low word	High Word	Angle in °	Sine value in IQ30
0x3F F000	0x0000	0x0000	0	0.0
0x3F F002	0x0E90	0x00C9	0.703	0.01227153838
0x3F F004	0x155F	0x0192	1.406	0.02454122808
0x3F F006	0x0CAF	0x025B	2.109	0.03680722322
...				
0x3F F100	0x0000	0x4000	90	1.0
....				
0x3F F200	0x0000	0x0000	180	0.0
....				
0x3F F300	0x0000	0xC000	270	-1.0
....				
0x3F F3FE	0xF170	0xFF36	359,3	-0.01227153838

Resume Lab Exercise 5A

Let's resume the procedure for Lab5A:

19. How do we get access to this boot ROM sine wave table?

    We have to add some basic support from the "IQ-Math" library to our project. At the top of your code, just after the line "#include "DSP281x_Device.h" add the next lines:

```
#include "IQmathLib.h"

#pragma DATA_SECTION(sine_table, "IQmathTables");

_iq30  sine_table[512];
```

The #pragma statement declares a specific data memory area, called “IQmathTables”. This area will be linked in the next procedure step to the address range of the Boot ROM sine table. The global variable “sine_table[512]” is an array of this new data type “IQ30”.

20. Add an additional Linker Command file to your project. From E:\C281x\Labs\Lab5A add:

    Lab5A.cmd

    Open and inspect this file. You will see that we add just one entry for the physical memory location (“ROM“) in data page 1 and that we connect the memory area “IQmathTables” to address “ROM”. The attribute “NOLOAD” assures that the debugger will not try to download this area into the DSP when we load the program – because it is already there, it is ROM – read only memory.

21. Modify “T1_Compare_isr( )”

    This interrupt service routine is a good point to modify the pulse width of the PWM signal. Recall, we do have now a global array “sine_table[512]” that holds all the sine values we need for our calculation.

    Now we have to do a little bit of math’s.

    What is the relationship between this sine value and the value of T1CMPR?

    Answer:

    (1) We know that the difference between T1PR and T1CMPR defines the pulse width of the current PWM period. So the goal is to calculate a new value for T1CMPR.

    (2) Next, we have to take into account that the sine table delivers signed values between +1 and –1. Therefore we have to add an offset of +1.0 to this value.

    (3) This shifted sine value has to be multiplied with T1PR/2.

    Summary:

$$T1CMPR = T1PR - \left(\left(\sin e\_table[index] + 1.0 \right) * \frac{T1PR}{2} \right)$$

    To code this into the “IQ-Math” form we use:

    T1CMPR = T1PR - _IQ30mpy(sine_table[index] + _IQ30(0.9999),T1PR/2)

    “_IQ30mpy(a,b)” is an intrinsic function call to do a multiplication in IQ30-Format. The value from sine_table is already in IQ30-format, whereas the constant 1.0 has to be translated into it by function call “_IQ30(0.9999)”.

To avoid fixed-point overflows we can embed the calculation into a saturation function "_IQsat(x,max,min)" to limit the result between T1PR and 0. This leads to our final instruction:

```
EvaRegs.T1CMPR =
    EvaRegs.T1PR – _IQsat(
            _IQmpy( sine_table[index] + _IQ30(0.9999), EvaRegs.T1PR/2),
                                                    EvaRegs.T1PR,0) ;
```

Add this line just after the "EDIS" instruction that follows the service of the Watchdog timer.

22. Setup the sine wave frequency.

Recall that the Boot ROM sine table consists of 512 entries for a unit circle. The frequency of the sine wave is given by:

$$f_{SIN} = \frac{f_{PWM}}{Number \text{ of PWM - periods per } 360°}$$

For example, if we use all 512 entries of the Boot ROM table we get:

$$f_{SIN} = \frac{50KHz}{512} = 97{,}6Hz$$

If we use only one lookup table entry out of 4, we end up with:

$$f_{SIN} = \frac{50KHz}{128} = 390{,}6Hz$$

Let's use this last set up. It means we have to increment the index by 4 to make the next access to the lookup table. Add the next line after the update line for T1CMPR:

```
index += 4;
```

Do not forget to:

(1) Reset variable index to 0 if it is increased above 511.

(2) Declare the integer variable index to be static.

Build and Load

23. Click the "Rebuild All" button or perform:

Project → Build

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

24. Load and test the final version of the output file as you've done before. With the help of the oscilloscope we should see now a change of the pulse width of the PWM signal 'on the fly'.

25. Optional: Low pass filter:

    The low pass filter capacity of the tiny loudspeaker is not strong enough to integrate the pulse sequence to a sine wave shaped signal. We can improve this by adding a simple low-pass filter between the two connectors of jumper JP7-2 (DSP-T1PWM) and JP7-1 (Loudspeaker). Build a passive low pass filter of first order with a frequency of 25 KHz:

$$f_{Filter} = \frac{1}{R \cdot C} = 25KHz$$

Optional Exercise

How about other frequencies?

In the previous exercise we generated a modulated sine wave of 390 Hz. We used the 512-point look-up table and stepped through it using an increment of 4.

How do we generate other frequencies?

Answer: when we change the step size for variable "index" we can generate more (or less PWM-periods per 360°. More means we slow down the sine frequency, less means we increase the sine wave frequency. The following table shows the different sine wave frequencies for a PWM carrier frequency of 50 KHz and a lookup table of 512 points per 360°:

$$f_{SIN} = \frac{f_{PWM}}{Number \text{ of PWM - periods per } 360°}$$

Incremental step of "index"	Number of PWM periods per 360°	Sine wave frequency In Hz
1	512	97.6
2	256	195.3
3	171	293
4	128	390
5	102	488
10	51	976
15	34	1,460
20	26	1,950
50	10	4,880

We can't increase the step size much above 50 because this gives us only 10 points per 360° to synthesize the sine wave.

What about frequencies that we do not match with any of these incremental steps? Recall our Lab Exercise 5 with the range of 8 notes; it started with a note of 264Hz.

How do we generate a sine wave of 264 Hz?

The answer is: We have to modify the PWM frequency itself. So far we did all experiments with a fixed PWM signal of 50 KHz. Let's fix now the number of points taken out of the look up table to 128 (that is an index increment by 4). To get a sine wave of 264Hz we calculate:

$$f_{SIN} = \frac{f_{PWM}}{128} = 264Hz$$

$$f_{PWM} = 264Hz * 128 = 33{,}792KHz$$

To setup a PWM signal of 33,792 KHz we have to re-calculate T1PR:

$$f_{PWM} = \frac{f_{CPU}}{T1PR \cdot TPS_{T1} \cdot HISCP}$$

$$T1PR = \frac{150MHz}{33.792KHz \cdot 1 \cdot 2} = 2{,}219.46$$

T1PR has to be loaded with an integer value, so we have to round the result to 2219.

Test:
$$f_{PWM} = \frac{f_{CPU}}{T1PR \cdot TPS_{T1} \cdot HISCP} = \frac{150MHz}{2219 * 1 * 2} = 33.799KHz$$

$$f_{SIN} = \frac{f_{PWM}}{128} = \frac{33.799KHz}{128} = 264.05Hz$$

That's a reasonable result; the intended frequency of 264Hz is missed by an error of 0.02%.

26. Try to setup your code to generate a sine wave of 264Hz!

27. If you have additional time in your laboratory try to improve Lab5 to generate all 8 notes with sine wave modulated PWM's!

End of Lab 5A

This page was intentionally left blank.