F28027 USB - Stick ePWM

Introduction

This chapter is an add-on to module 7. It describes the laboratory procedures for ePWM-exercises, based on the F28027 Piccolo-USB-Stick (Texas Instruments part no: TMDX28027USB). For the description of the ePWM-unit and control registers, please refer to the documentation of chapter 7.

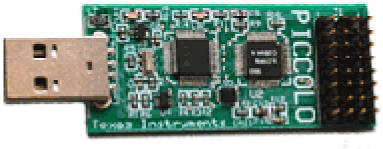
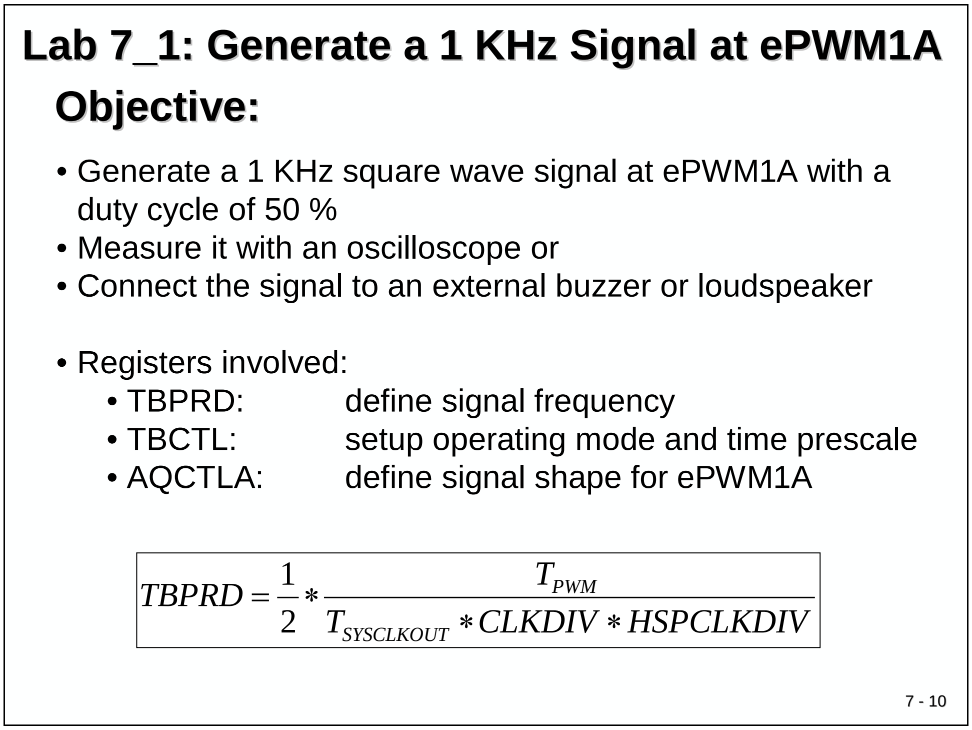


Table of contents

Lab 7_1: Generate an ePWM signal



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 J1-17 of the Piccolo USB-Stick device we can monitor the signal. A small external circuit incorporating 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 project, called **Lab7A.pjt** in C:\DSP2802x\Labs (or in another path that is accessible by you; ask your teacher or a technician for an appropriate location!).

2. Create a new source code file “Lab7_1.c” and add it to your new project:

   - **Lab7_1.c**

3. From C:\tidcs\c28\dsp2802x\v110\DSP2802x_headers\source add:

   - **DSP2802x_GlobalVariableDefs.c**

4. From *C:\tidcs\c28\dsp2802x\v110\DSP2802x_common\source* add:

- **DSP2802x_CodeStartBranch.asm**

5. From *C:\tidcs\c28\dsp2802x\v110\DSP2802x_common\cmd* add:

- **28027_RAM_lnk.cmd**

6. From *C:\tidcs\c28\dsp2802x\v110\DSP2802x_headers\cmd* add:

- **DSP2802x_Headers_nonBIOS.cmd**

7. From *C:\CCStudio_v3.3\c2000\cgtools\lib* add:

- **rts2800_ml.lib**

8. From *C:\tidcs\c28\dsp2802x\v110\DSP2802x_common\source* add:

- **DSP2802x_SysCtrl.c**
- **DSP2802x_usDelay.asm**

Project Build Options

9. We also have to setup the search path of the C-Compiler for include files. Click:

Project → Build Options

Select the Compiler tab. In the "Preprocessor" category, find the Include Search Path (-i) box and enter the following two lines in this box:

C:\tidcs\C28\dsp2802x\v110\DSP2802x_headers\include;
C: tidcs\C28\dsp2802x\v110\DSP2802x_common\include

10. 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**>.

Edit Source Code

11. Now we can start to edit file "Lab7_1.c". First include the declaration of data types used for global variables:

```
#include "DSP2802x_Device.h"
```

12. Next, declare the prototypes for an external function, provided by Texas Instruments:

```
extern void InitSysCtrl(void);
```

13. Now continue with declarations for our own local functions:

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

14. Now we can start to write the main function. The first activity in main is to call a function, provided by Texas Instruments, to initialize the core of the DSC:

```
InitSysCtrl();
```

If you would like to keep the watchdog unit running, which is always a good practice, re-enable this unit:

```
EALLOW;
SysCtrlRegs.WDCR = 0x00AF;
EDIS;
```

Next, call your two local functions to initialize ePWM1 and the multiplex registers for GPIO:

```
Gpio_select();
Setup_ePWM1A();
```

Finally, enter an endless while(1) - loop. The only activity in this loop is to permanently service the watchdog unit.

```
EALLOW;
SysCtrlRegs.WDKEY = 0x55;
SysCtrlRegs.WDKEY = 0x55;
EDIS;
```

15. After “main()”, add the definition of function “Gpio_Select()”. In this function set all multiplex registers to GPIO functions. For line GPIO0, set the multiplex bit to allow ePWM1A as output signal.

16. Finally, add the definition for function “Setup_ePWM1A()” at the end of your code. We have to set registers TBCTL, TBPRD and AQCTLA. The frequency of the output signal is given by:

$$\boxed{TBPRD = \frac{f_{CPU}}{2 * f_{PWM} * CLKDIV * HSPCLKDIV}}$$

For f_{PWM} = 1 kHz and f_{CPU} = 60MHz we could use CLKDIV = divide by 1 and HSPCLKDIV = divide by 1 to get TBPRD = 30000.

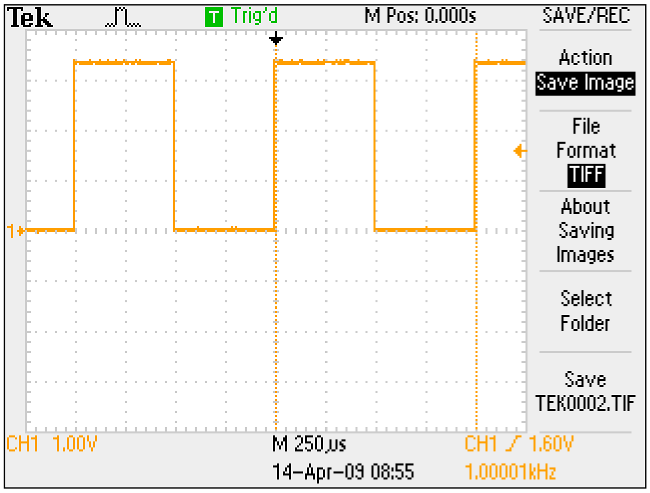
Register TBCTL covers the bit fields for CLKDIV, HSPCLKDIV and the operation mode (CTRMODE), which should be set to “up-down-mode”

In register AQCTLA select ZRO = set and PRD = clear.

Re - Build, Load and Test

17. Now rebuild, load and test the new project. The program should still show the binary counter from Lab6 at LED LD1 and LD2. The new addition is a 1 kHz - signal at output ePWM1A (header J1-17 on the Piccolo-USB-Stick).

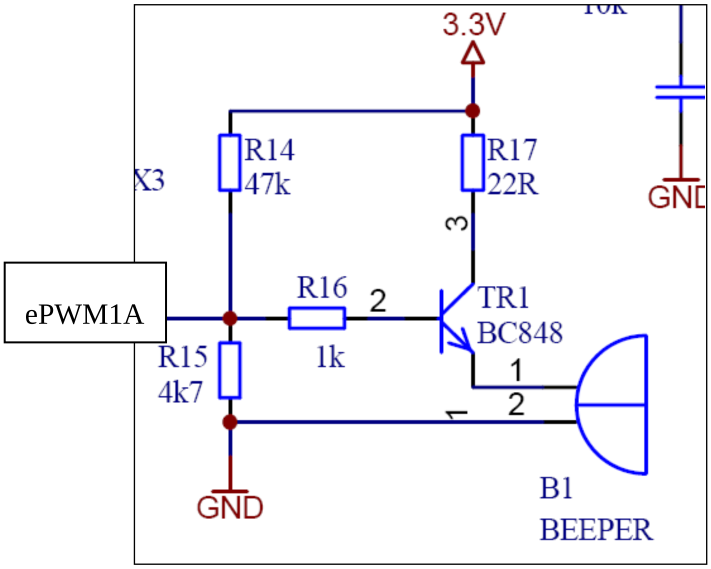
18. Use a scope to inspect this signal. It should look like:



M Pos: 0.000s
CH1 1.00V
M 250µs
CH1 ⁄ 1.60V
14-Apr-09 08:55
1.00001kHz

19. Optional exercise: experiment with different frequencies by changing the value for register TBPRD!

20. Optional Hardware: Make your frequency audible! By adding the following circuitry to your Piccolo-USB-Stick, we can do it!

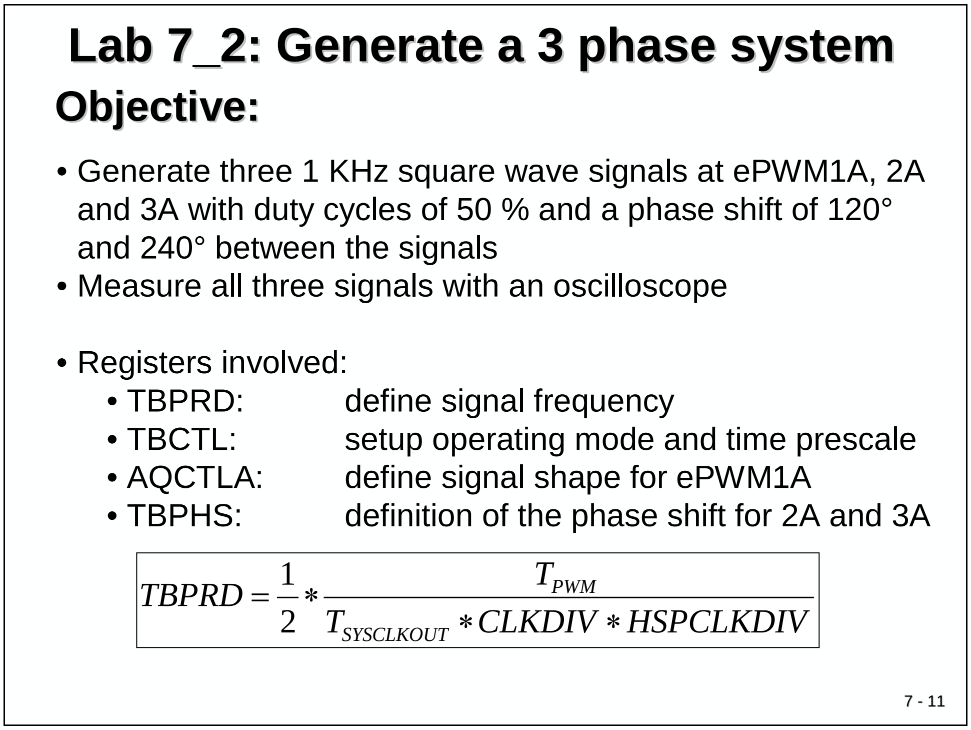


Device B1 ("Beeper") can be a Digisound F/SMD8585JSLF (Mouser Part # 847 - FSMD8585JS) or a Digisound F/PCW04A.

END of LAB 7_1

Lab 7_2: Generate a 3 - phase signal system

Now let us experiment with a 3-phase system and phase shifts 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 ePWM3A will trail at 120° and 240°.



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, 2A and 3A. With the help of a 4-channel oscilloscope connected to header J1-17, J1-27 and J1-21 of the Piccolo - USB - stick we can monitor the signal.

Procedure

Open Project File

1. If not still open from Lab7_1, re-open project **Lab7A.pjt.**

2. Open file “Lab7_1.c” and save it as “Lab7_2.c”

3. Remove file “Lab7_1.c” from project and add “Lab7_2.c” to it. Note: optionally you can also keep “Lab7_1.c” but exclude it from build. Use a right mouse click on file “Lab7_1.c”, select “File Specific Options”; in category “General” enable “Exclude from Build”.

Modify Source Code

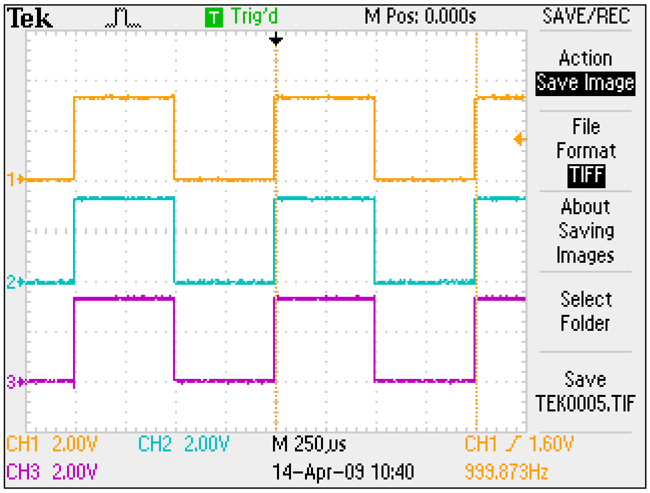
4. In the file “Lab7_2.c”, change the function name “Setup_ePWM1A()”. Since we will also initialize ePWM2A and ePWM3A with this function, the function name is now somewhat misleading. Change the name into “Setup_ePWM()”, including the function prototype and the calling line in the “main()” loop.

5. Next, in the local function “Gpio_select()”, add instructions to initialize the pin functions of GPIO2 and GPIO4 to ePWM2A and ePWM3A.

6. In the function “Setup_ePWM()”, repeat the initialization for ePWM1A with the same instructions for ePWM2A and ePWM3A. Apply identical values now 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 obtain 3 identical 1 kHz - signals with zero phase-shift between the 3 ePWM lines:



7. Now let us add the phase shifts between ePWM1A, ePWM2A and ePWM3A. To do so, we will have to program the phase registers of ePWM2A and ePWM3A. We must also define ePWM1A as the master phase to generate a SYNCOUT pulse each time its counter register TBCNT equals zero. For ePWM2, we must allow 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 ePWM2A and ePWM3A we need a phase shift of 1/3rd and 2/3rd 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:

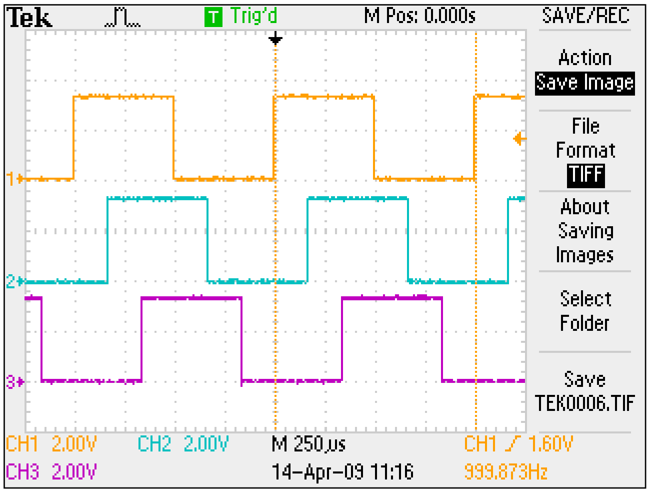
- Set phase enable

EPwm3Regs.TBPHS:

- Load it 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:

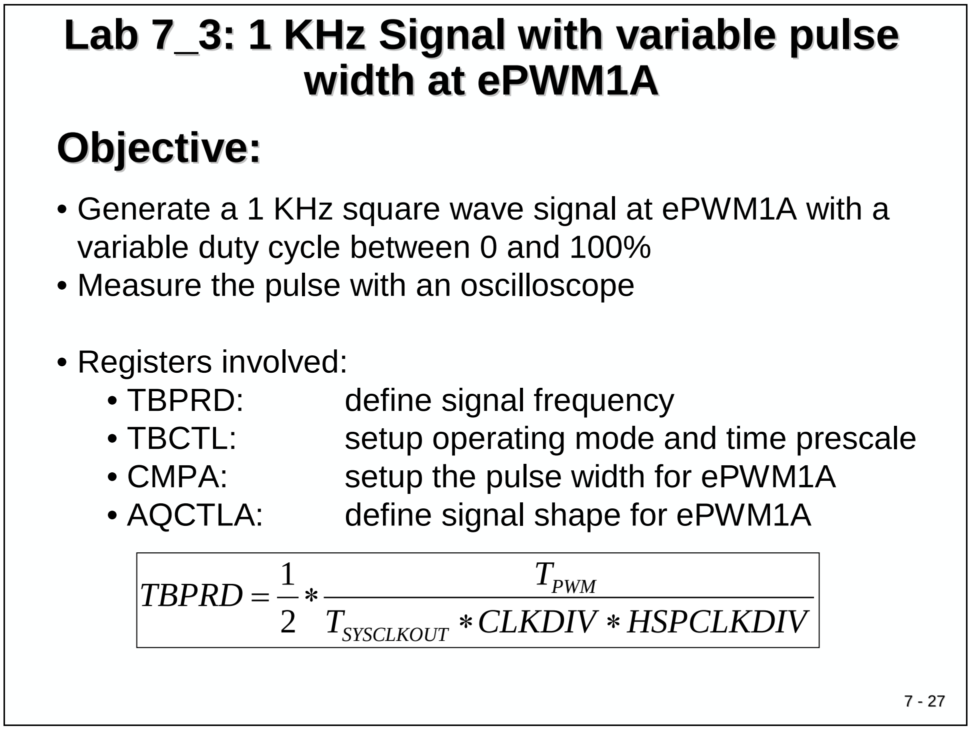


CH1 2.00V CH2 2.00V M 250µs CH1 ╱ 1.60V
CH3 2.00V 14-Apr-09 11:16 999.873Hz

END OF LAB 7_2

Lab 7_3: 1 kHz signal 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 the time-base, which is used to change the pulse width of the 1 kHz signal 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 J1-17 of the Piccolo - USB - Stick we can monitor the signal. Using CPU - Timer 0, we will change CMPA to generate a pulse width between 100 and 0%.

Procedure

Open Project File

1. In CCS, if not still open from Lab7_2, re-open project **Lab7A.pjt.**

2. Open file “Lab7_1.c” and save it as “Lab7_3.c”

3. Remove file “Lab7_2.c” from project and add “Lab7_3.c” to it. Note: optionally you can also keep “Lab7_2.c” and exclude it from build. Use a right mouse click on file “Lab7_2.c”, select “File Specific Options”; in category “General” enable “Exclude from Build”.

4. First we have to initialize the interrupt system. To do so we have to include the following source code files into our project:

   From *C:\tidcs\c28\dsp2802x\v110\DSP2802x_common\source* add:

   - **DSP2802x_PieCtrl.c**
   - **DSP2802x_PieVect.c**
   - **DSP2802x_DefaultIsr.c**
   - **DSP2802x_CpuTimers.c**

   Next, add external function prototype declarations at the beginning of “Lab7_3.c”:

```
extern void InitPieCtrl(void);
extern void InitPieVectTable(void);
extern void InitCpuTimers(void);
extern void ConfigCpuTimer(struct CPUTIMER_VARS *, float, float);
```

   In main, after the function call of “Setup_ePWM1A()” add new lines to initialize the interrupt system and CPU Timer 0:

```
InitPieCtrl();
InitPieVextTable();
EALLOW;
PieVextTable.TINT0 = &cpu_timer0_isr;
EDIS;
InitCpuTimers();
ConfigCpuTimer(&CpuTimer0,60,100);
PieCtrlRegs.PIEIER1.bit.INTx7=1;
IER |= 1;
EINT;
CpuTimer0Regs.TCR.bit.TSS = 0;
```

5. In file “Lab7_3.c” edit 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!
- Next, add a line to initialize CMPA to 0, which will define a pulse width of 100%:

```
EPwm1Regs.CMPA.half.CMPA = 0;
```

6. CpuTimer0 will request an interrupt every 100 microseconds. In step 4 we connected this interrupt to a function “cpu_timer0_isr()”. Now we have to provide this function.

    - First, define a function prototype at the beginning of “Lab7_3.c”:

      interrupt void cpu_timer0_isr(void);

7. At the end of “Lab7_3.c” add function “cpu_timer0_isr()”. Take into account:

    - Increment the value in register CMPA with each interrupt execution until it equals 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%.

    - At the end of this function acknowledge that interrupt service:

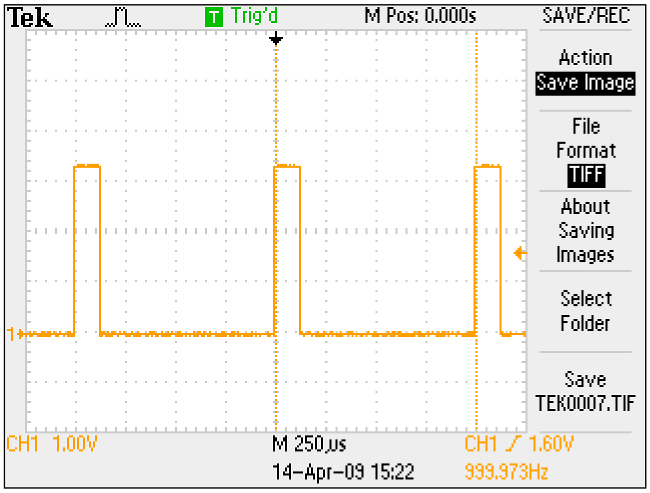
      PieCtrlRegs.PIEACK.all = 1;

    Note: All registers of ePWM1 are readable 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:

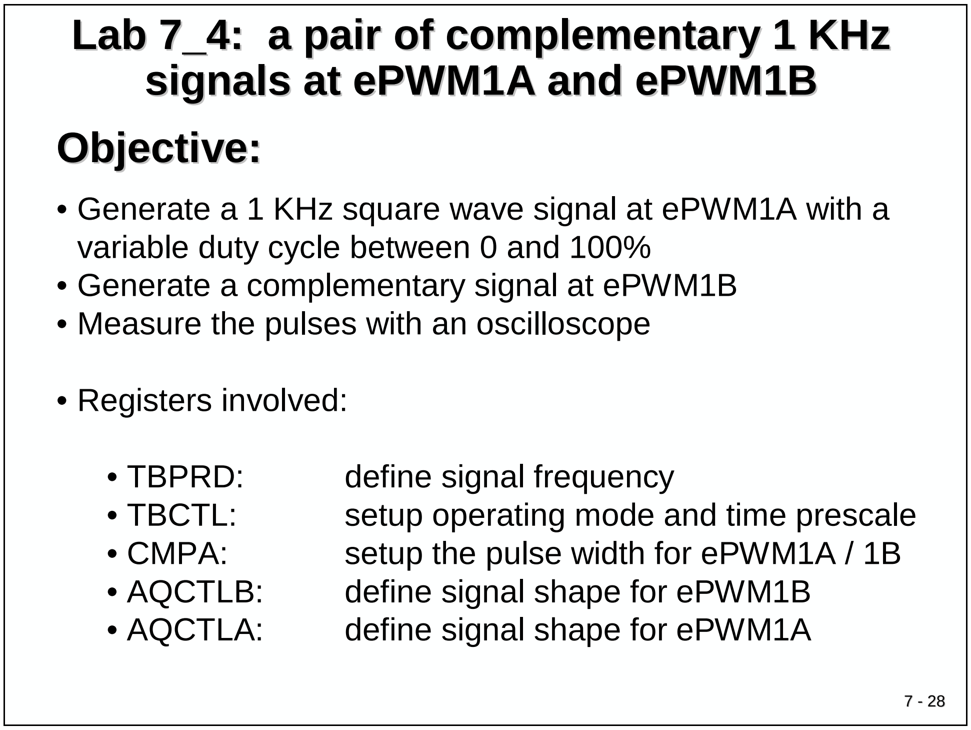


Result: The pulse width of your signal should change gradually between 100% and 0%.

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, the other switch is off. In the following exercise you will modify Lab7_3 to generate a pair of output pulses at ePWM1A and ePWM1B. Again CpuTimer0 will be used as the 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 on the ePWM1A line and a second signal at ePWM1B with opposite voltage levels. With the help of an oscilloscope connected to header J1-17 (ePWM1A) and J1-21 (ePWM1B) of the Piccolo-USB-Stick, we can monitor the signals. 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. If not still open from Lab7_3, re-open project **Lab7A.pjt** in Code Composer Studio.

2. Open file “Lab7_3.c” and save it as “Lab7_4.c”

3. Remove file “Lab7_3.c” from project and add “Lab7_4.c” to it. Note: optionally you can also keep “Lab7_3.c” and exclude it from build. Use a right mouse click on file

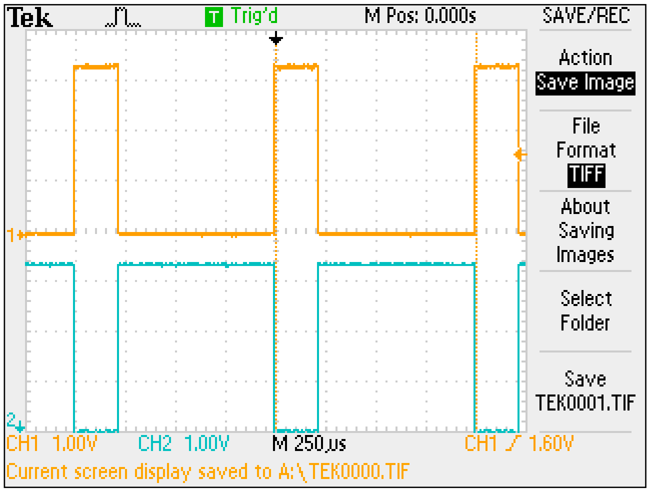
"Lab7_3.c", select "File Specific Options"; in category "General" enable "Exclude from Build".

4. In file "Lab7_4.c" edit function "Gpio_select()". In the multiplex block enable line GPIO1 to drive ePWM1B.

5. In "Setup_ePWM1A()" 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 this setup to generate a complementary signal at ePWM1B.

Build, Load and Test

6. Now build, load and test the modified project. An oscilloscope screenshot of signal ePWM1A (J1-17) and ePWM1B (J1-21) should look like the following picture:



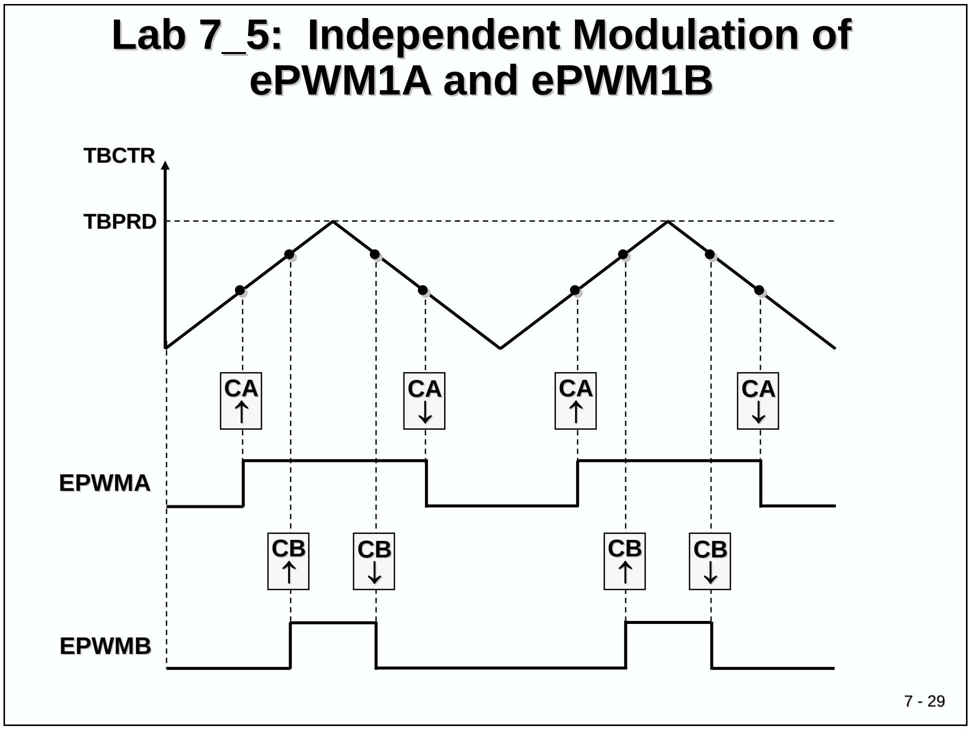
Current screen display saved to A:\TEK0000.TIF

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 pulse pattern, shown in Slide 7-29:



Lab 7_5: Independent Modulation of ePWM1A and ePWM1B

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 pulses width. 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_4, re-open project **Lab7A.pjt** in Code Composer Studio.

2. Open file “Lab7_4.c” and save it as “Lab7_5.c”

3. Remove file “Lab7_4.c” from project and add “Lab7_5.c” to it. Note: optionally you can also keep “Lab7_4.c” and exclude it from build. Use a right mouse click on file “Lab7_4.c”, select “File Specific Options”; in category “General” enable “Exclude from Build”.

4. In function “Setup_ePWM1A()” 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.

5. 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;

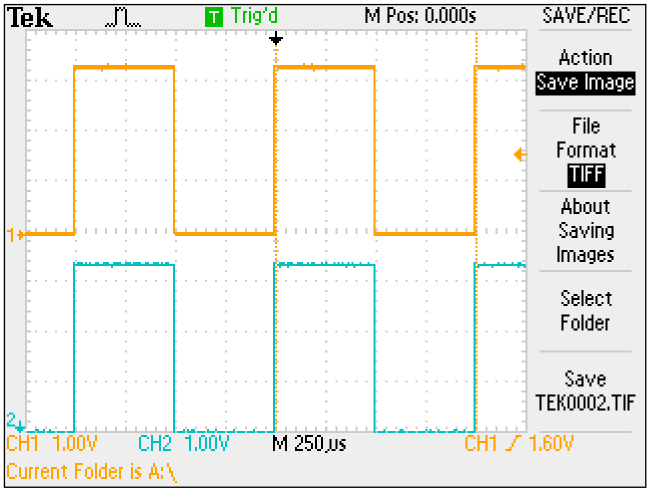
   EPwm1Regs.CMPB = EPwm1Regs.TBPRD / 2;

   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.

6. In 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

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



8. Stop the code execution:

   Debug ➔ Halt, followed by

   Debug ➔ Reset CPU

9. Now open a Watch Window:

View ➔ Watch Window

In window “Watch 1” add the two variables:

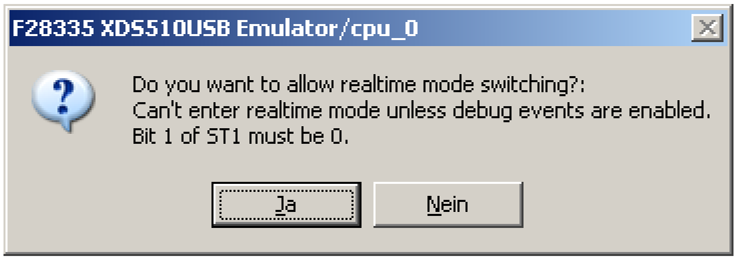
EPwm1Regs.CMPA.half.CMPA and

EPwm1Regs.CMPB

10. Enable Real Time Debug Mode:

Debug ➔ Real-time 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”:



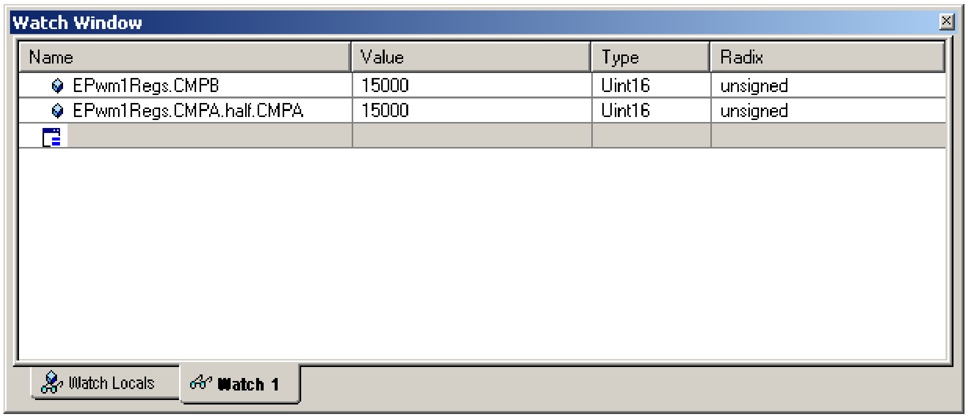
Do you want to allow realtime mode switching?:
Can't enter realtime mode unless debug events are enabled.
Bit 1 of ST1 must be 0.

Right mouse click into the Watch window and select “Continuous Refresh”

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

GEL ➔ Realtime Emulation Control ➔ Run_Realtime_with_Restart

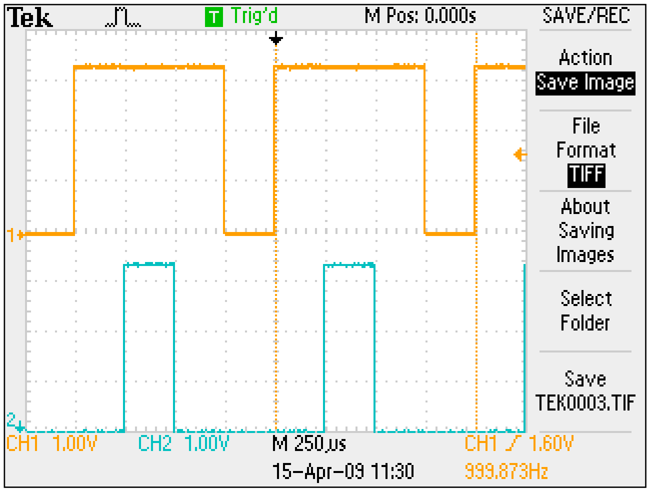
Your Watch window should display the current values for CMPA and CMPB, e.g.:



Name	Value	Type	Radix
EPwm1Regs.CMPB	15000	Uint16	unsigned
EPwm1Regs.CMPA.half.CMPA	15000	Uint16	unsigned

Now, while the code is still running, change the values in CMPA and CMPB to 7500 and 22500 respectively.

The result should look like:



Try other combinations for 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:

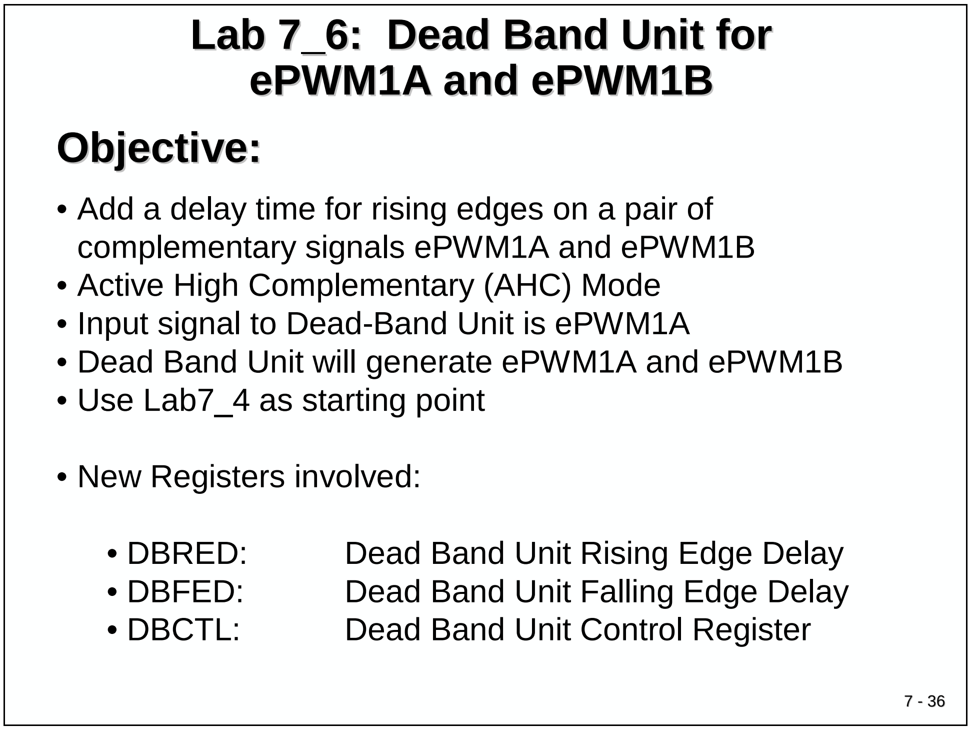
GEL ➔ Realtime Emulation Control ➔ Full_Halt_with_Reset

END of LAB 7_5

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 **Lab7A.pjt** in Code Composer Studio.

2. Open file "Lab7_4.c" and save it as "Lab7_6.c"

3. Add "Lab7_6.c" to the project and exclude "Lab7_5.c" from build. Use a right mouse click on file "Lab7_5.c", select "File Specific Options"; in category "General" enable "Exclude from Build".

4. In 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_ePWM1A()", initialize the pulse width to 50% of TBPRD:

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

6. Next, in the function "Setup_ePWM1A()", remove the instruction to initialize register AQCTLB. When using the dead band unit both output pulse sequences ePWM1A and ePWM1B are usually derived from a single input signal, usually from internal signal ePWM1A of the action qualifier module.

7. Finally, in the function "Setup_ePWM1A()", 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 and HSPCLKDIV set to 1. In case of the F28027USB stick at 60 MHz TBCLK equals to 16.666 ns. In our example we will setup a delay time of 10 microseconds, just as an example.

   EPwm1Regs.DBRED = 600;

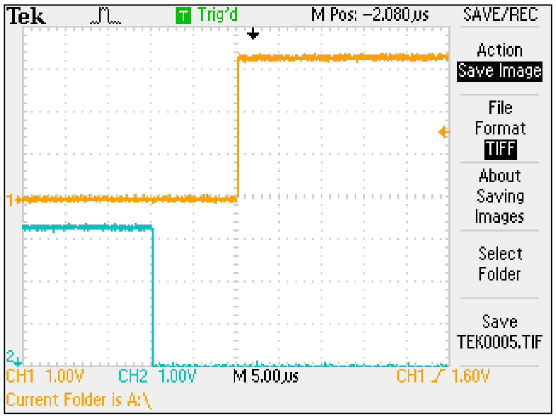
   EPwm1Regs.DBFED = 600;

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

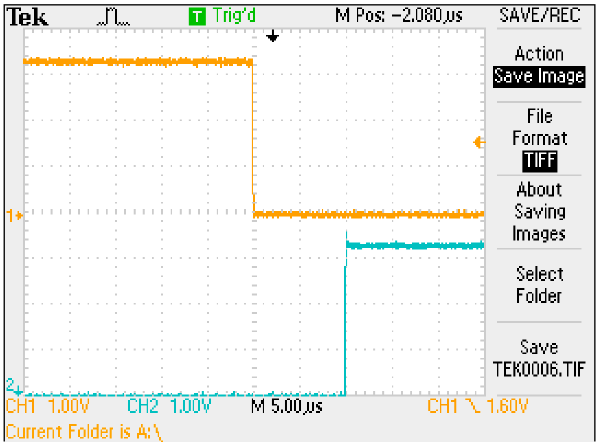
   - Set S4 and S5 to 0: this way we will solely use input signal ePWM1A from unit AQCTL to generate the two output signals ePWM1A and ePWM1B.
   - Set S2 = 0 and S3=1 to invert the polarity of signal ePWM1B against input ePWM1A.
   - Set S0 = 1 and S1 = 1 to include a time delay for both switching points

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), you should see again a delayed rising edge, now on signal ePWM1B (blue):



M Pos: -2.080µs

CH1 1.00V CH2 1.00V M 5.00µs CH1 ↘ 1.60V

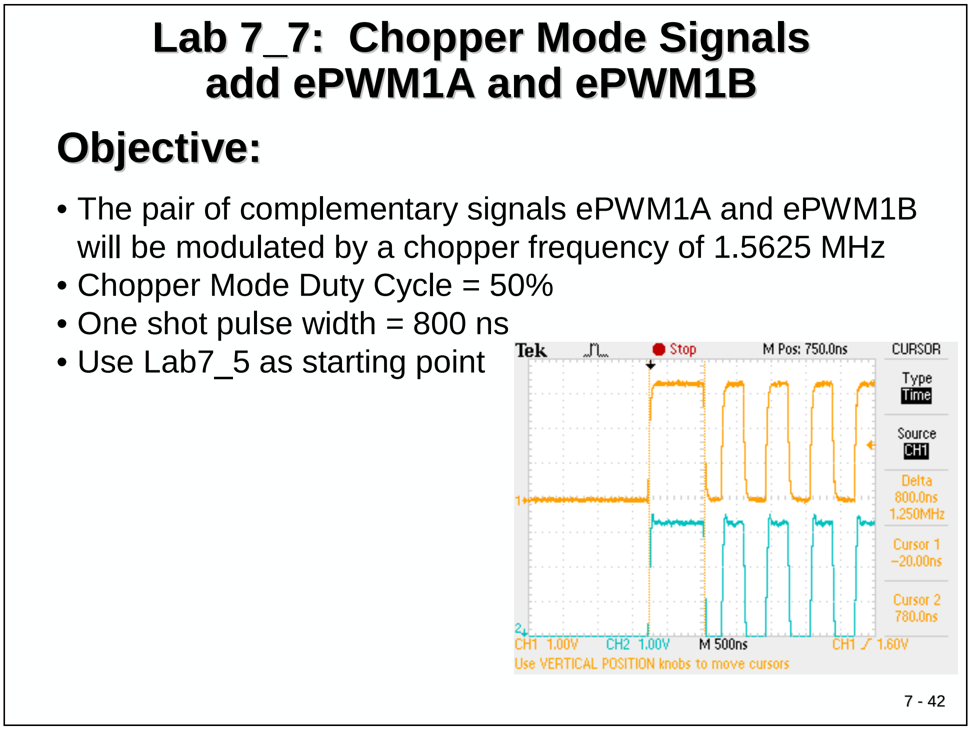
Current Folder is A:\

END of LAB 7_6

Lab 7_7: Chopped Signals at ePWM1A / 1B

Objective

We will add a chopper frequency modulation to a solution of one of the previous labs, that is -Lab7_5. 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 1.5625 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 “Lab7A” open file “Lab7_5.c” and save it as “Lab7_7.c”

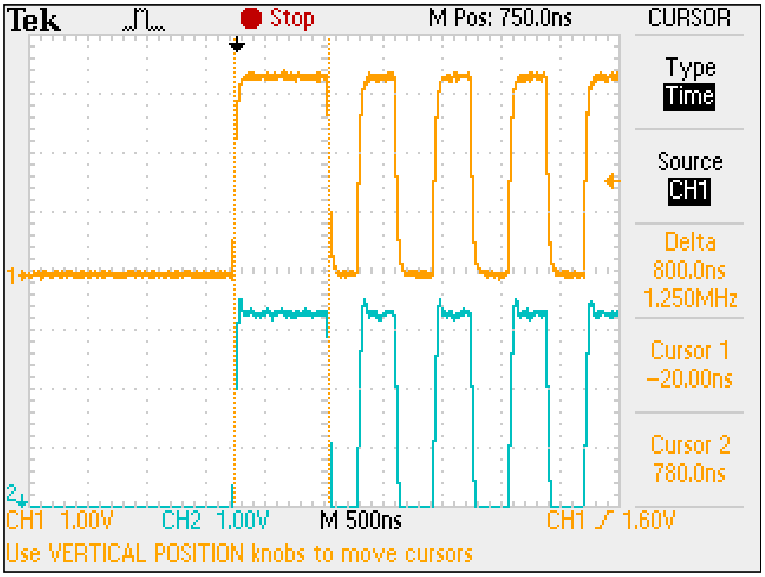
2. Add “Lab7_7.c” to the project and exclude “Lab7_6.c” from build. Use a right mouse click on file “Lab7_6.c”, select “File Specific Options”; in category “General” enable “Exclude from Build”.

3. In function “Setup_ePWM1A()”, initialize the chopper module. Recall that SYSCLKOUT has been set to 60 MHz. For the chopper unit the input clock is 60MHz / 8 = 7.5MHz, e.g. a period of 133.33 ns. In register **“EPwm1Regs.PCCTL”**:

- Set chopper frequency to 1.5 MHz, e.g. a period of 666 ns
- Set chopper duty cycle to 50%
- Set one shot pulse to 800 ns
- Enable the chopper unit.

Build, Load and Test

4. 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):

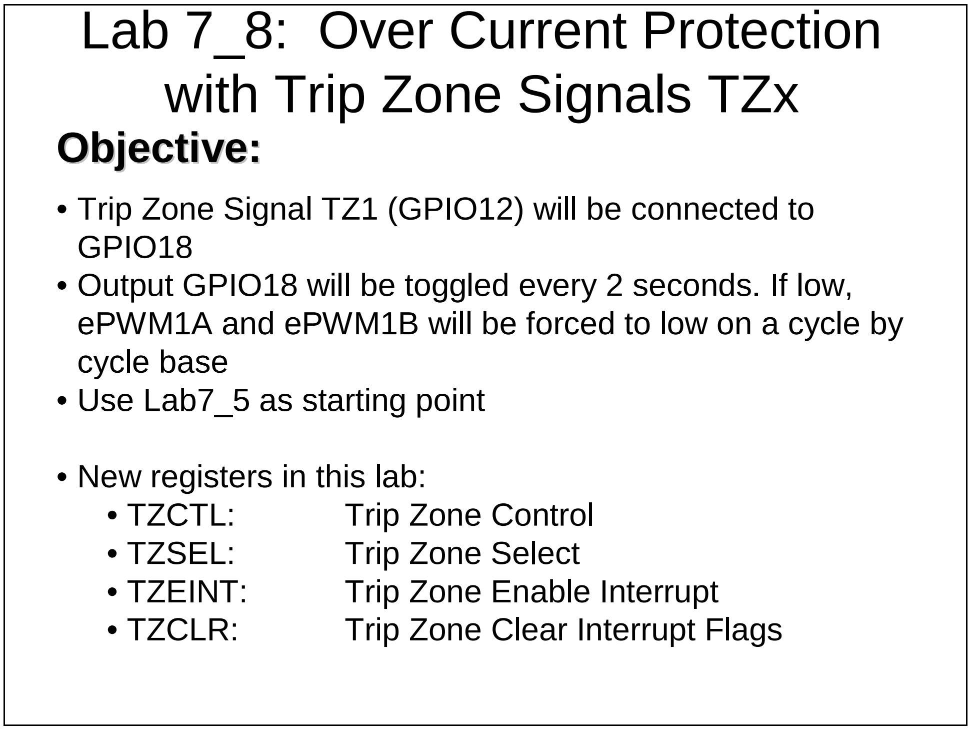


END of LAB 7_7

Lab 7_8: Trip Zone protection with TZ6

Objective

Unfortunately the Piccolo - USB Stick does not have a push-button. However, we can use a standard GPIO - pin (GPIO18) as the source for an over-current signal. We can toggle this pin periodically every 2 seconds by software. We will use Trip Zone signal /TZ1 which is multiplexed with input signal GPIO12. The objective is to force both ePWM1A and ePWM1B "cycle by cycle" to low in case of a low active /TZ1.



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

Objective:

- Trip Zone Signal TZ1 (GPIO12) will be connected to GPIO18
- Output GPIO18 will be toggled every 2 seconds. If low, ePWM1A and ePWM1B will be forced to low on a cycle by cycle base
- 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

Procedure

Open Project File

1. In project "Lab7A" open file "Lab7_5.c" and save it as "Lab7_8.c"

2. Add "Lab7_8.c" to the project and exclude "Lab7_7.c" from build. Use a right mouse click on file "Lab7_7.c", select "File Specific Options"; in category "General" enable "Exclude from Build".

3. In the function "Gpio_select()", set multiplex register GPAMUX2 to use /TZ1 for GPIO12. In Register GPADIR set GPIO18 to output and in GPADAT set bit GPIO18 to 1.

4. In the function "Setup_ePWM1A()", initialize the trip zone registers.

- In register “EPwm1Regs.TZCTL” set TZA and TZB to force ePWM1A and ePWM1B to zero in case of an active TZ1.
- In register “EPwm1Regs.TZSEL” selects TZ6 as source for a one shot over current signal. In the event of an active TZ6 (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.

5. Next, in the Interrupt Service Routine “cpu_timer0_isr()”, add a line to increment the interrupt counter:

```
CpuTimer0.InterruptCount++;
```

If this variable has reached value 10, toggle pin GPIO18 and clear CpuTimer0.InterruptCount.

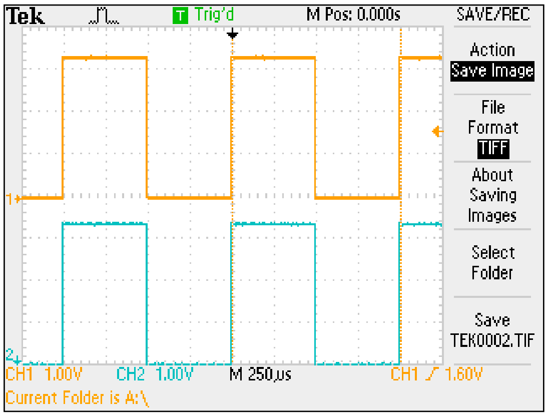
6. In function “main()”, change parameter 3 of the function call “ConfigCpuTimer()” to 200 milliseconds:

```
ConfigCpuTimer(&CpuTimer0,60,200000);
```

Build, Load and Test

7. Use a wire and connect J1-8 (GPIO12) to J1-29 (GPIO18).

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



9. After a runtime of two seconds, the signals should disappear and both lines should be permanently zero. That is the result of the over current signal (GPIO18), which was generated after that time.

One Shot Mode

10. Now let us modify the code in such a way, that a low active GPIO18 will request a cycle-by-cycle switch off of the two signals ePWM1A and ePWM1B.

- In the function “Setup_ePWM1A()”, 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

11. 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 sequence:

    - **Debug ➔ Reset CPU**
    - **Debug ➔ Restart**
    - **Debug ➔ Go Main**
    - **Debug ➔ Run**

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

    Now every 2 seconds the signals ePWM1A and ePWM1B should fade out to ground and remain at this ground voltage. Then 2 seconds later, 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 Piccolo-USB-Stick, it still allows us also to perform an exercise, which uses an interrupt service - routine in the event of an over-current situation.

12. At the beginning of “Lab7_8.c”, add a prototype for an interrupt service routine:

    interrupt void ePWM1_TZ_isr(void);

13. 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;

14. 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;

15. Change the line “IER |= 1;” so that the two lines INT1 and INT2 are enabled:

    IER |= 3;

16. In the function “Setup_ePWM1A()”, add a line to enable cycle-by-cycle interrupts in register EPwm1Regs.TZEINT. Include this new instruction in the EALLOW - EDIS block!

17. At the end of “Lab7_8.c”, add the definition for the 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 TZ1 for the next interrupt service:

      EPwm1Regs.TZCLR.bit.CBC = 1;

      EPwm1Regs.TZCLR.bit.INT = 1;

    Remember that this register is EALLOW - protected!

    - Now, because we “simulate” our over current signal TZ1 by a 2 seconds toggle signal, the duration of the “over-current” signal is exactly 2 seconds long. It means that TZ1 will trigger a next interrupt immediately after we return from interrupt function “ePWM1_TZ_isr()”.

    Recall that we have three different software activities in Lab7_8:

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

    Because interrupt service “cpu_timer0_isr()” has 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!

    Solution: 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 GPIO34:

      GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;

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

      PieCtrlRegs.PIEACK.all = 2;

Re-Build, Load and Test

18. 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 oscilloscope should again show the pulse sequences at ePWM1A and ePWM1B.

    As before, every 2 seconds the signals should fade out to ground and reappear after that pause.

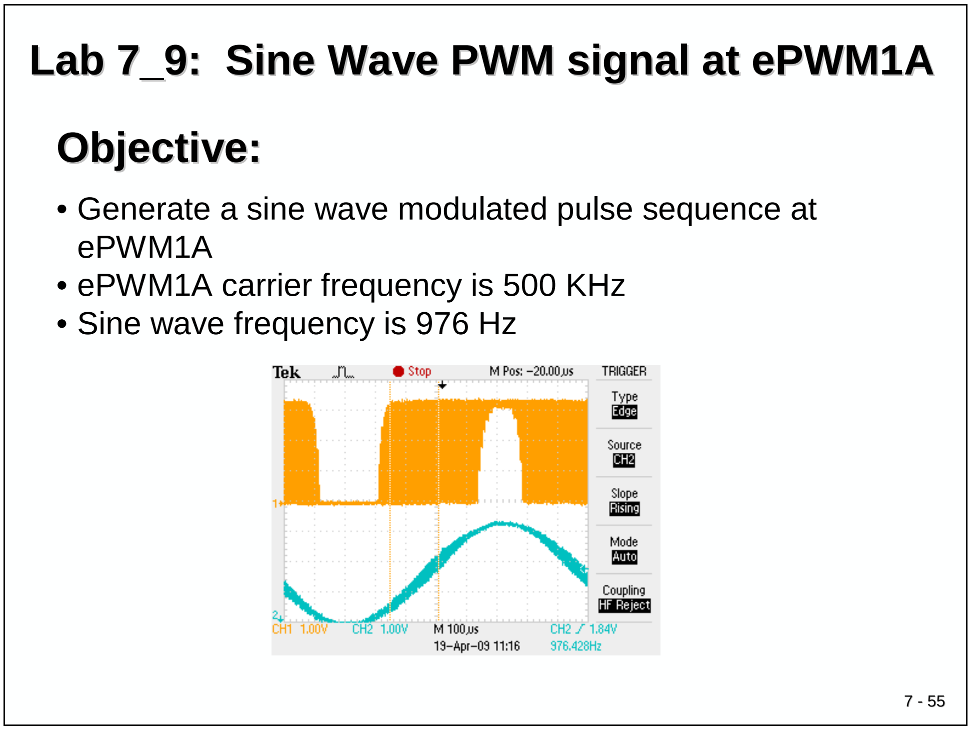
LED LD2 (GPIO34) should be off for 2 seconds, then half on for 2 seconds (that is where we permanently get TZ1- interrupts and toggle the LED frequently, and then 2 seconds ON.

END of Lab7_8

Lab7_9: ePWM Sine Wave Modulation

Objective

The F28027 Piccolo USB Stick is used to generate a sine wave signal at ePWM1A. Channel ePWM1A is set up in standard 16-bit resolution. The generated signal is connected to a first order passive low pass filter R12 / C15. The filter output signal can be monitored at header J1-31 ("HR-DAC") of the stick.



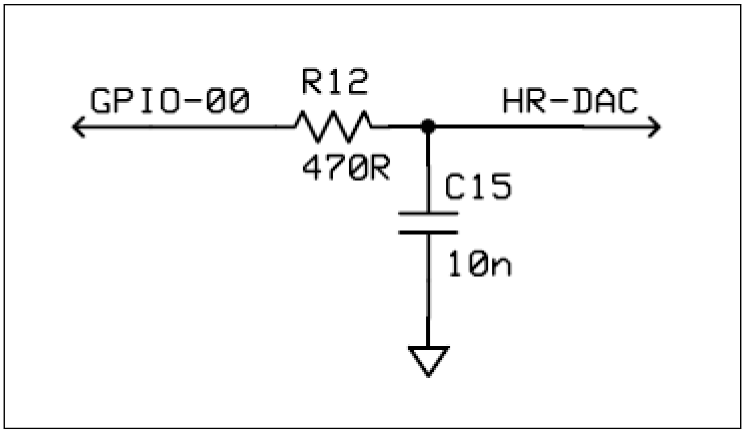
Lab 7_9: Sine Wave PWM signal at ePWM1A

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 2.000 ns.



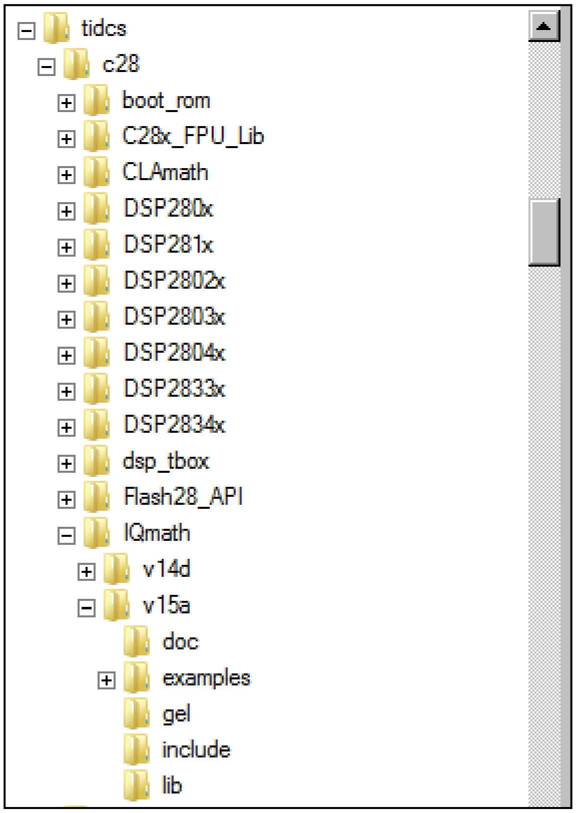
The ISR with a code execution time of 630ns takes advantage of the Boot-ROM sine wave lookup-table to calculate the next compare value for the next ePWM1A period. The lookup-table consists of 512 values in I2Q30-format and is located at address 0x3FE000. Every ISR call is used to read the next entry of this table, thus a full period of the resulting sine wave takes 512 * 2000 ns = 1024 µs. The synthesized sine wave signal has a frequency of 1/1024µs = 976 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 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 “Lab7A” open file “Lab7_8.c” and save it as “Lab7_9.c”

2. Add “Lab7_9.c” to the project and exclude “Lab7_8.c” from build. Use a right mouse click on file “Lab7_8.c”, select “File Specific Options”; in category “General” enable “Exclude from Build”.

3. Change Build options.

   We have to extend the preprocessors “#include”- search path. Open Project Build Options and go into the preprocessor category. The “Include Search Path” already has two entries. Do not delete these entries! Instead, add at the end of this line a semicolon and a third entry:

   ; C:\tidcs\c28\IQmath\v15\include

   Close the Build options menu with <OK>

4. Add IQmath library to your project. Add:

   C:\tidcs\c28\IQmath\v15\lib\IQmath_fpu32.lib

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

   #include “IQmathLib.h”

   Next and 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 “28027_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.

   In “Lab7_9.c” remove everything that is related to CpuTimer0, including external function prototypes, the call of functions “InitCpuTimers()”, “ConfigCpuTimer()” and Interrupt Service Routine “cpu_timer0_isr()”, including its prototype and definition.

6. 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);

7. 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;
```

8. In the while(1) - loop of “main()” keep just the instruction to service the watchdog instruction #1 (value 0x55) to register WDKEY. Recall that register WDKEY is EALLOW protected!

9. Next, in the function “Gpio_select()”, just keep ePWM1A as the PWM output signal. Remove the instructions to enable lines ePWM1B and TZ1.

10. In the function “Setup_ePWM1A()”, change the period of ePWM1 to 500 kHz. In up/down mode the value for TBPRD is calculated by:

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

with CLKDIV and HSPCLKDIV both set to “divide by 1” and $T_{SYSCLKOUT}$ = 16.666 ns, TBPRD should be initialized to 60.

11. Then in the function “Setup_ePWM1A()”, remove the initialization lines for registers CMPB an AQCTLB, since we will not generate a signal on ePWM1B.

12. At the end of function “Setup_ePWM1A()”, remove the code to initialize the trip zone unit, including all instructions for registers TZCTL, TZSEL and TZEINT.

13. At the end of function “Setup_ePWM1A” add code to initialize the Event Trigger module. In register “ETSEL” enable bit “INTEN” and set bit field “INTSEL” to select an interrupt request, if CTRD = CMPA (counter down matches CMPA). In register “ETPS” set bit field “INTPRD” to request an interrupt on first event.

14. 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 percentage value in CMPA.
- To find that next value to be subtracted from TBPRD we have to access the sine table. Variable "index" points into this table, which consists of 512 entries for a unit circle of 360 degree. 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 front 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 signal suddenly becomes a large negative signal.

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:

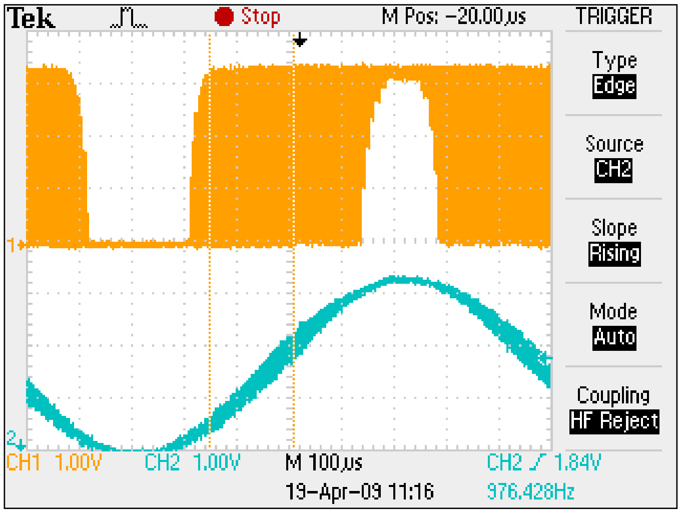
```
EPwm1Regs.ETCLR.bit.INT = 1;
PieCtrlRegs.PIEACK.all = 4;
```

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

Build, Load and Test

15. 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:
    - **Debug ➔ Reset CPU**
    - **Debug ➔ Restart**
    - **Debug ➔ Go Main**
    - **Debug ➔ Run**

16. An oscilloscope should show the 500 kHz - pulse sequence at ePWM1A (Header J1-17) and a sine wave signal of 976 Hz at HR-DAC (Header J1-31).



M Pos: −20.00µs

TRIGGER

Type Edge

Source CH2

Slope Rising

Mode Auto

Coupling HF Reject

CH1 1.00V CH2 1.00V M 100µs CH2 ╱ 1.84V

19−Apr−09 11:16 976.428Hz

END of LAB 7_9

Lab7_10: ePWM1A 1 kHz signal captured by eCAP1

Objective

The F28027 Piccolo-USB-Stick is used to generate a 1 kHz square wave signal with a duty cycle of 50% on ePWM1A. We will use the eCAP1 unit to measure period and duty cycle of this signal.

Note: for this exercise you will need to connect header J1-17 (ePWM1A) to header J1-15 (eCAP1) of the Piccolo-USB-Stick.

Procedure

Open Project File

1. In project “Lab7A”, open the file “Lab7_3.c” and save it as “Lab7_10.c”

2. Add “Lab7_10.c” to the project and exclude “Lab7_9.c” from build. Use a right mouse click on file “Lab7_9.c”, select “File Specific Options”; in category “General” enable “Exclude from Build”.

Edit Source File

3. In the function “Gpio_select()”, select eCAP1 function for pin GPIO5. On the Piccolo-USB-Stick we can access eCAP1 via header J1-15, which is wired to pin GPIO5. Adjust register GPAMUX1 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 function "main()" add a line to call the function "Setup_eCAP1()". The best position is directly after the function call "Setup_ePWM1A()".

7. Next, in 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 this new exercise we have to enable INT1 (CPU Timer 0) and INT4 (eCAP1).

8. Also, in 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;
```

11. In interrupt service routine "cpu_timer0_isr()", remove everything but the watchdog service instruction and the PIE acknowledge line.

12. Finally, in the function "Setup_ePWM1A()", initialize register EPwmRegs.CMPA for a duty cycle of 50%, e.g. with the value of TBPRD/2.

Build, Load and Test

13. Build the modified project.

    ➔ **Project** ➔ **Rebuild All**

14. Use a wire to connect header J1-17(ePWM1A) to header J1-15 (eCAP1).

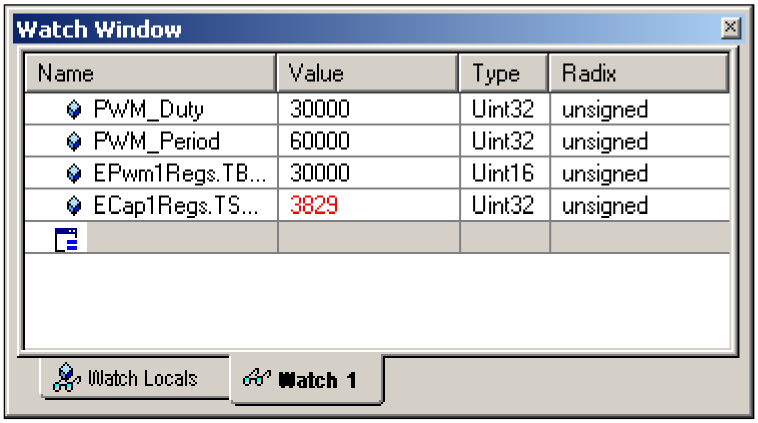
15. Load the modified code:

    ➔ **File** ➔ **Load Program**

16. Test the code:

    ➔ **GEL** ➔ **Realtime Emulation Control** ➔ **Run_Realtime_with_Restart**

17. Open the Watch Window and add the variables "PWM_Duty", "PWM_Period" and "ECap1Regs.TSCTR" to it. With a left mouse click in the "Radix" column one can change the format to "unsigned" for all 3 variables. Now click right mouse in the Watch Window and enable "Continuous Refresh".



Name	Value	Type	Radix
PWM_Duty	30000	Uint32	unsigned
PWM_Period	60000	Uint32	unsigned
EPwm1Regs.TB...	30000	Uint16	unsigned
ECap1Regs.TS...	3829	Uint32	unsigned

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 ms and a pulse width of 0.5 milliseconds. Our measurement unit has a resolution of 1/60MHz = 16.667 ns. Therefore the value of 60.000 for "PWM_Period" translates into 60.000 * 16.667ns = 1 ms.

18. Finally halt the DSC:

    ➔ **GEL** ➔ **Realtime Emulation Control** ➔ **Full_Halt_with_Reset**

END of LAB 7_10