PWM Control Using the TMS320F2837xD Dual-Core DSP

Juan Camilo Garcıa Perez. *Department of Electronic Engineering, Santo Tomas University,* Bogota, Colombia (e-mail: juancgarciap@usantotomas.edu.co)

Abstract: This paper presents a detailed study of *Pulse Width Modulation (PWM)* control using the TMS320F2837xD Dual-Core DSP. The paper explores the various features and capabilities of the DSP's PWM peripheral, and demonstrates how it can be utilized for precise control of power electronics systems. The paper also presents several real-world applications of PWM control using the DSP: (1) generating a 3-phase signal system using the three different PWM modules, (2) generating a 1 kHz signal with variable pulse width, (3) generating a pair of complementary 1 kHz signals at EPWM1A and EPWM2A, (4) independently modulating ePWM1A and ePWM1B, and (5) setting up a dead band unit for ePWM1A and ePWM1B rising edge. The purpose of this paper is to demonstrate the versatility of the TMS320F2837xD DSP in generating different PWM configurations for various applications.

Introduction

The TMS320F2837xD Dual-Core DSP is a powerful and versatile device that is widely used in power electronics applications. One of the key features of the device is its built-in PWM peripheral, which enables precise control of power electronics systems. This paper presents an in-depth study of PWM control using the TMS320F2837xD DSP, and explores the various features and capabilities of the device's PWM peripheral, demonstrating the versatility and performance of the device.

In this paper, we explore five different methods for configuring the PWM signals on the TMS320F2837xD. The first method involves generating a 3-phase signal system using the three different PWM channels available on the DSP. This is a common configuration used in motor control applications.

The second method involves generating a 1 kHz signal with variable pulse width. This can be used in applications where the PWM duty cycle needs to be adjusted dynamically, such as in power electronics.

The third method involves generating a pair of complementary 1 kHz signals at EPWM1A and EPWM2A. This configuration is useful for applications that require two PWM signals that are out of phase with each other.

The fourth method involves independent modulation of ePWM1A and ePWM1B. This can be used to generate two separate PWM signals with different duty cycles and frequencies, which can be useful in applications that require complex control algorithms.

Finally, the fifth method involves using the dead band unit for ePWM1A and ePWM1B rising edges. This configuration is useful for applications that require precise timing control, as it allows for a delay between the rising edges of the two PWM signals.

Each of these methods will be described in detail, including the required configuration settings and any special considerations that need to be taken into account. By the end of this paper, readers will have a comprehensive understanding of how to configure the PWM signals on the TMS320F2837xD for a variety of real-time applications.

Objectives

Main Objective

To provide a comprehensive study of PWM control using the TMS320F2837xD Dual-Core DSP, and to demonstrate how the device's built-in PWM peripheral can be utilized for precise control of power electronics systems.

Specific Objectives

1. To explore the various features and capabilities of the TMS320F2837xD Dual-Core DSP's PWM peripheral.
2. To demonstrate the use of the DSP's PWM peripheral for precise control of power electronics systems.
3. To present several real-world applications of PWM control using the TMS320F2837xD DSP, highlighting the versatility and performance of the device.

Procedure

To develop this paper, it was required to configure the signal frequency, in the following way of setting up an ePWM module a $f = 1kHz$ square wave signal is generated. Knowing the frequency of the signal the PWM peripheral needs to set up its signal frequency: *Time base period (TBPRD)* as its shown in eq(1).

$$TBPRD = \frac{f_{DSP}}{(1/2) * f_{PWM} * D_1 * D_2}$$

(1)

Eq(1) shows us the divider value can be modify from 1 to 2, 1 if it's used the up or down ePWM counter mode and 2 if it's used the up and down ePWM counter mode, also shows two dividers, which can be modified, $CLKDIV = D_1 = [1, 2, 4, 8, 16, 32, 64, 128]$ and $HSPCLKDIV = D_2 = [1, 2, 4, 6, 8, 10, 12, 14]$, the $f_{DSP} = 10MHz$ wich is maximum

value that can be used in this peripheral and $f_{PWM} = 1kHz$ which is the desired square wave signal frequency. It's important to know that the maximum TBPRD value is 65535 if that value is greter than or equal to it you must modify the dividers values, for this configuration eq(1) is set:

$$TBPRD = \frac{10MHz}{2 * 1kHz * 1 * 1} = 50000$$

This value is going to be necessary in all the next PWM configurations.

1. Generating a 3-phase signal system using the three different PWM modules.

This method involves using three different PWM signals, where the duty cycles are modulated in such a way as to produce a 3-phase output signal. This can be used in various applications such as motor control, where a 3-phase AC voltage is required to drive the motor. In this implementation of a function called *InitEPWM1()*, which initializes three ePWM modules, namely EPWM1_BASE, EPWM2_BASE, and EPWM3_BASE. These modules are typically used to generate precise digital signals with varying duty cycles.

The function begins by setting the time base phase and period of the ePWM modules. In this case, the time base period is set to 50 kHz using the EPWM_setTimeBasePeriod() function.

The clock prescaler is then set to EPWM_CLOCK_DIVIDER_1 and the high speed clock divider is set to EPWM_HSCLOCK_DIVIDER_1.

The time base counter mode is set to EPWM_COUNTER_MODE_UP_DOWN, which causes the counter to count up and down repeatedly.

The phase shift is enabled using the EPWM_enablePhaseShiftLoad() function, and the phase shift value is set using the EPWM_setPhaseShift() function. The phase shift value for EPWM1_BASE is set to 0, while the phase shift value for EPWM2_BASE and EPWM3_BASE is set to 33.3e3, because:

$$\frac{2 * TBPRD}{3} = 33.3e3$$

The action qualifier (AQ) configuration for the ePWM modules is then set up using the EPWM_setActionQualifierAction() function. The AQ configuration determines the behavior of the ePWM output signals. In this case, EPWM1_BASE is configured to output a high signal when the time base counter is zero and a low signal when the time base counter equals the period. EPWM2_BASE is configured to output a low signal when the time base counter is zero and a high signal when the time base counter equals the period. EPWM3_BASE is configured to output a high signal when the time base counter is zero and a low signal when the time base counter equals the period.

Finally, the count mode after sync is set to EPWM_COUNT_MODE_DOWN_AFTER_SYNC for the three PWM. This causes the time base counter to count down after a software or hardware sync event.

According to this, was used a 2 chanel scope in order to verify the wave's behavior, as its shown in fig. 1.

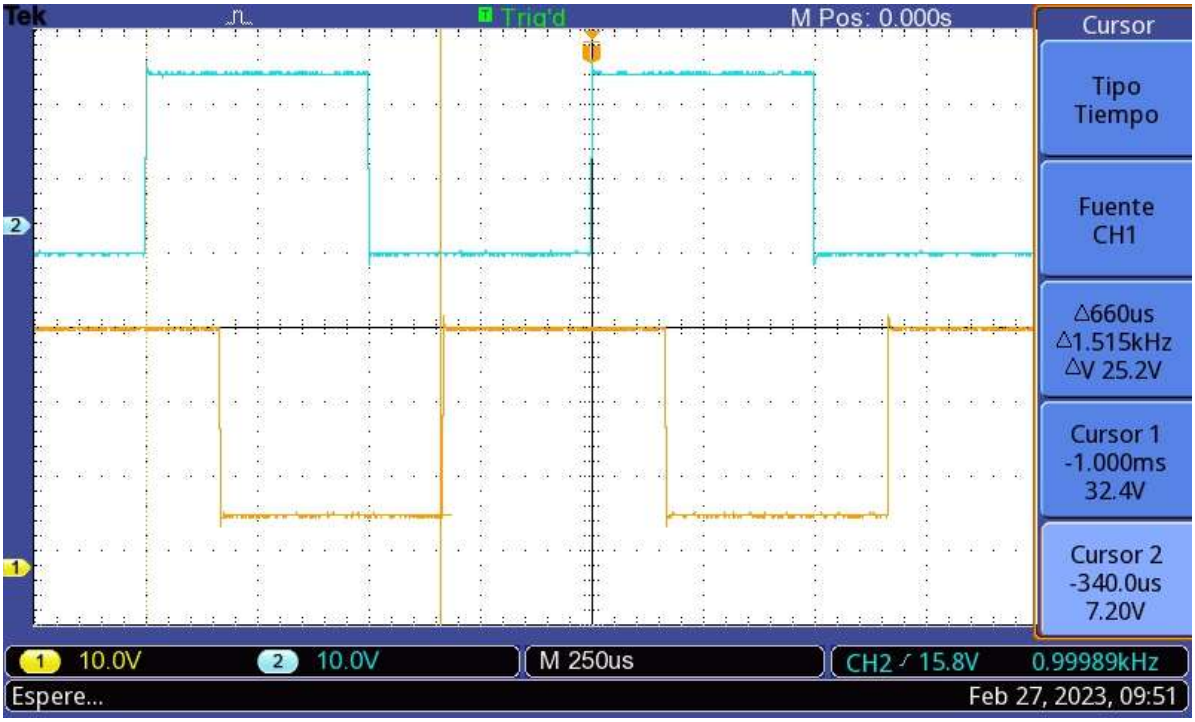

Fig. 1.2 – Phase Signal System.

In this case, we are comparing the EPWM1 with the EPWM3, we already knew that cutting the signal off in 120° when the main signal was at $\frac{2}{3}$ of *TBPRD*, was the key to understand how to configure the EPWM3 so using the qualifiers in order to count when the signal takes that value and it was going down, we were able to acieve the offset, and using the scope we set the triggers in order to comper the cycles of each PWM and it's verified the lag of $660\mu s$ as it was planned.

2. Generating a 1 kHz signal with variable pulse width.

In this implementation of a function called *InitEPWM3()*, it's configured the ePWM1 module to generate a pulse-width modulated signal with a fixed frequency of 1 kHz, but a variable pulse width that can be adjusted by modifying the value of the VALCOMP register.

The first part of the code sets up the ePWM module, similarly to the previous examples we have discussed. The time base counter is initialized to 0, and the time base period is set to 50,000. The clock prescaler is set to divide the clock by 1, and the counter mode is set to up-down. A phase shift of 0 is applied, and phase shift load is enabled.

Next, the VALCOMP register is set to 35,000. This sets the initial pulse width of the PWM signal.

The code then enters an infinite loop. Inside the loop, the code checks if the VALCOMP register has been changed from its previous value. If it has, the OLDVALCOMP variable is updated to the new value of VALCOMP, and the PWM signal is reconfigured to have a pulse width equal to VALCOMP.

To modify the pulse width, the code uses the EPWM_setCounterCompareValue function to set the compare value of the counter to the current value of VALCOMP. This sets the point in the PWM cycle at which the output will transition from high to low.

The code then uses the EPWM_setActionQualifierAction function to configure the output behavior of the ePWM module. In this case, the A output is set to go high at timebase zero and low at the compare A event.

Because the loop is infinite, the code will continuously check for changes to the VALCOMP register and update the PWM signal accordingly. This allows for real-time adjustment of the pulse width of the PWM signal.

According to this, was used a 2 chanel scope in order to verify the wave's behavior, as its shown in fig. 2, 3, and 4.

Fig. 2. Pulse Width when $VALCOMP = 35000$.

Fig. 3. *Pulse Width when* $VALCOMP = 15000$.

Fig. 4. *Pulse Width when* $VALCOMP = 50000$.

The pulse width was set equal than TBPRD but according our configuration of the qualifiers that is why it only reaches the 50% of the duty.

3. Generating a pair of complementary 1 kHz signals at EPWM1A and EPWM2A.

This method involves two complementary 1 kHz signals at EPWM1A and EPWM2A. The EPWM1A signal has a high pulse width when the timebase counter is at zero and a low pulse width when the counter is at the compare value. The EPWM2A signal has a high pulse width when the timebase counter is at the compare value and a low pulse width when the counter is at zero.

The EPWM1A signal is configured using the EPWM1_BASE registers. It is set to count from 0 to 50000, with a prescaler of 1 and a high-speed clock divider of 1. The counter counts up and down, and the phase shift is set to 0.

The EPWM2A signal is configured using the EPWM2_BASE registers. It is set to count from 0 to 50000, with a prescaler of 1 and a high-speed clock divider of 1. The counter counts up and down, and the phase shift is set to 0.

Both signals have a counter compare value of 10000, and their action qualifier actions are set up to generate complementary signals. Specifically, when the EPWM1A signal is at 0, the output is high, and when it is at the counter compare value, the output is low. For EPWM2A, when the signal is at the counter compare value, the output is high, and when it is at 0, the output is low.

Finally, the count mode after sync for EPWM2A is set to count down after a sync event. According to this, was used a 2 chanel scope in order to verify the wave's behavior, as its shown in fig. 5.

Fig. 5. *Complemetary Signals.*

4. Independently modulating ePWM1A and ePWM1B.

This method involves two independent ePWM channels, ePWM1A and ePWM1B. Both channels are set to have a time base period of 50000 and use a clock prescaler of 1. They both operate in up-down counting mode and have phase shift enabled with a phase shift value of 0.

For ePWM1A, the counter compare value is set to 15000, and two action qualifier actions are set. When the timebase counter is equal to the up compare value (CMPA), the output is set high, and when the timebase counter is equal to the down compare value (CMPA), the output is set low.

For ePWM1B, the counter compare value is set to 35000, and two action qualifier actions are set. When the timebase counter is equal to the up compare value (CMPB), the output is set high, and when the timebase counter is equal to the down compare value (CMPB), the output is set low.

In summary, this code initializes two independent ePWM channels with different counter compare values and action qualifier actions. The outputs of the channels will change state at specific timebase counter values based on the compare values and action qualifiers that were set.

According to this, was used a 2 chanel scope in order to verify the wave's behavior, as its shown in fig. 6.

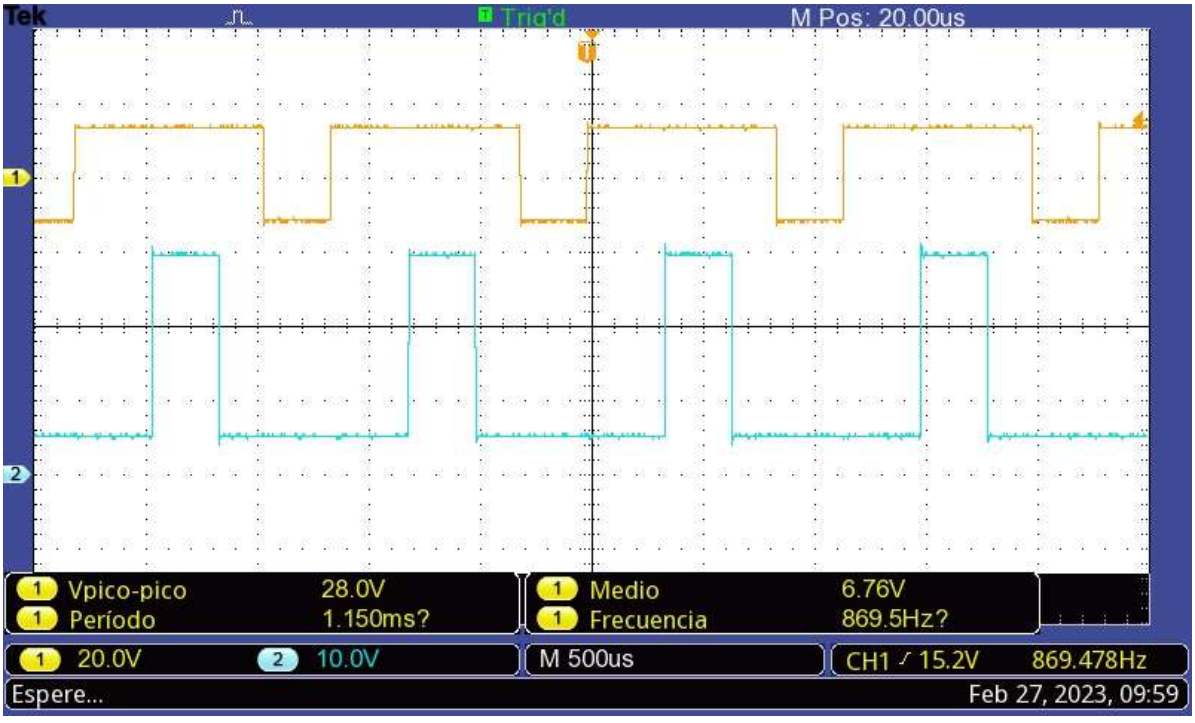

Fig. 6. *Independently Modulation.*

5. Setting up a dead band unit for ePWM1A and ePWM1B rising edge

This method involves adding a dead-band unit to the PWM signals. The ePWMs are configured to have a period of 50000 and are set up for up-down counting mode. The dead band delay is enabled for rising edge and set to 750 counts, which adds a delay to the rising edge of the PWM signal.

For ePWM1A, the rising edge delay input is set to EPWM1A, and the compare value is set to 25000. The action qualifiers are set up to output a high signal on the up-count and a low signal on the down-count.

For ePWM1B, the rising edge delay input is set to EPWM1A, and the compare value is set to 25000. The action qualifiers are set up to output a low signal on the up-count and a high signal on the down-count.

Dead band is a time delay inserted between the rising and falling edge of a PWM signal to avoid the occurrence of shoot-through currents in the output stage of an H-bridge. Shoot-through is a condition when both the upper and lower switches of an H-bridge are on at the same time, resulting in a short circuit between the power supply rails. This can lead to damage to the power stage, and therefore it is essential to prevent it from occurring. The dead band delay ensures that the lower switch turns off before the upper switch turns on and vice versa, preventing shoot-through currents from flowing.

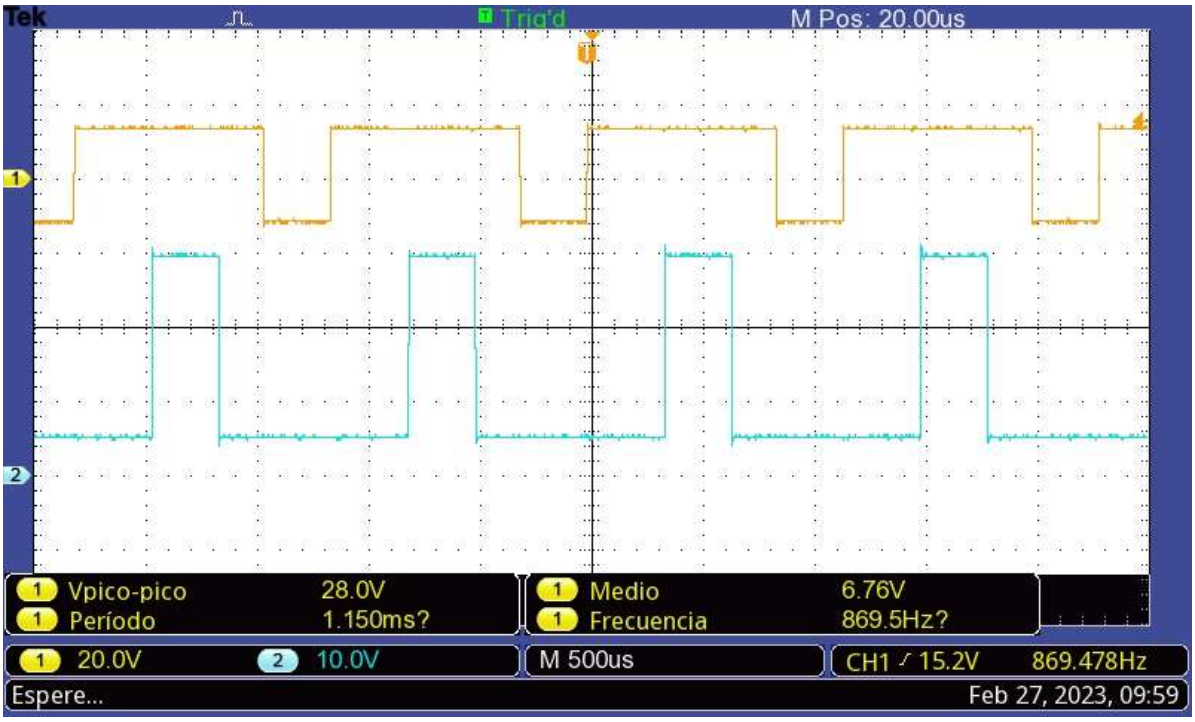

Fig. 6. *Dead Band* (*Rising Edge*).

Conclusion

The ePWM module is a powerful tool for controlling pulse width modulation signals in embedded systems. It allows you to precisely control the frequency and duty cycle of your

signal, which is useful in many applications. The specific ePWM configurations you've provided each have a different purpose. For example, some are designed to create specific pulse shapes or output waveforms, while others are intended for use in dead time management or other types of control applications.

Each ePWM configuration requires careful parameter selection and setup to achieve the desired output behavior. In some cases, this may involve configuring multiple registers and modules to work together in a specific way.

Many of the ePWM configurations you've provided involve setting up different types of action qualifiers, which determine the output behavior of the PWM signal in different situations. For example, action qualifiers can be used to control what happens when the PWM counter reaches its upper or lower limit, or when a compare match occurs.

Some of the ePWM configurations you've provided also involve setting up dead band delays, which are used to prevent shoot-through current in power electronics applications.

Dead bands introduce a delay between turning off one switch and turning on another, which helps to prevent current from flowing through both switches simultaneously and causing a short circuit.

Finally, the ePWM configurations involve configuring the ePWM module using the TI C2000 family of microcontrollers. However, the basic concepts and techniques used to configure ePWM signals are likely to be similar across different platforms and microcontroller families.

References

[1] TMS320F2837xD Dual-Core Microcontrollers Technical Reference Manual.

[2] F2837xD Firmware Development Package. USER’S GUIDE.

[3] DSP2839D Plans. Texas Instruments.

```c
#include "driverlib.h"
#include "device.h"

// Function prototypes
void Config_GPIO(void);
void InitEPWM1();
void InitEPWM3();
void InitEPWM4();
void InitEPWM5();
void InitEPWM6R();
void config_timer0(void);
__interrupt void Timer0ISR(void);

uint16_t LAB, OLDLAB, VALCOMP, OLDVALCOMP;

// Main function
void main(void)
{
    // Initialization
    Device_init();
    Device_initGPIO();
    Interrupt_initModule();
    Interrupt_initVectorTable();
    config_timer0();
    Config_GPIO();


    // Enable clock sync
    SysCtl_disablePeripheral(SYSCTL_PERIPH_CLK_TBCLKSYNC);
    SysCtl_enablePeripheral(SYSCTL_PERIPH_CLK_TBCLKSYNC);

    // Register and enable interrupt
    Interrupt_register(INT_EPWM1, &Timer0ISR);
    Interrupt_enable(INT_EPWM1);
    EINT;
    ERTM;

    // Initial values for LAB and OLDLAB
    LAB = 4;
    OLDLAB = 0;
    while(1)
    {
        // Check if LAB value has changed
        if(LAB != OLDLAB)
        {
            OLDLAB = LAB;
            switch (LAB)
            {
                case 1:
                    InitEPWM1();
                    break;
                case 2:
                    InitEPWM1();
                    break;
                case 3:
                    InitEPWM3();
                    break;
```

```
58                 case 4:
59                     InitEPWM4();
60                     break;
61                 case 5:
62                     InitEPWM5();
63                 default:
64                     InitEPWM6R();
65                     break;
66             }
67         }
68     }
69 }
70
71 __interrupt void Timer0ISR(void)
72 {
73     EPWM_setCounterCompareValue(EPWM2_BASE, EPWM_COUNTER_COMPARE_A, 10000);
74     Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP1);
75 }
76
77 void Config_GPIO(void)
78 {
79     GPIO_setPinConfig(GPIO_0_EPWM1A);
80     GPIO_setPinConfig(GPIO_1_EPWM1B);
81     GPIO_setPinConfig(GPIO_2_EPWM2A);
82     GPIO_setPinConfig(GPIO_4_EPWM3A);
83     GPIO_setPadConfig(0, GPIO_PIN_TYPE_PULLUP);
84     GPIO_setPadConfig(1, GPIO_PIN_TYPE_PULLUP);
85     GPIO_setPadConfig(2, GPIO_PIN_TYPE_PULLUP);
86     GPIO_setPadConfig(4, GPIO_PIN_TYPE_PULLUP);
87 }
88
89 void config_timer0(void)
90 {
91     //
92     //  To define a period of 100 microseconds
93     //  for timer 0: ConfigCpuTimer(&CpuTimer0, 150, 100);
94     //  ConfigCpuTimer(Timer, Freq, Period)
95     //
96     CPUTimer_setPeriod(CPUTIMER0_BASE, 100);
97     CPUTimer_setPreScaler(CPUTIMER0_BASE, 150);
98     CPUTimer_reloadTimerCounter(CPUTIMER0_BASE);
99     CPUTimer_enableInterrupt(CPUTIMER0_BASE);
100    CPUTimer_startTimer(CPUTIMER0_BASE);
101 }
102
103
104 // Generate a 3 - phase signal system
105 void InitEPWM1()
106 {
107
108    // Initializing the time based registers
109    // Time Base Phase              0
110    // Time Base Period             50kHz
111
112    // Setting up EPWM01
113
114    EPWM_setTimeBaseCounter(EPWM1_BASE, 0U);
```

```
115     // TBPRD = 50000
116     EPWM_setTimeBasePeriod(EPWM1_BASE, 50000);
117     EPWM_setClockPrescaler(EPWM1_BASE, EPWM_CLOCK_DIVIDER_1,
118                                        EPWM_HSCLOCK_DIVIDER_1);
119     EPWM_setTimeBaseCounterMode(EPWM1_BASE, EPWM_COUNTER_MODE_UP_DOWN);
120     EPWM_enablePhaseShiftLoad(EPWM1_BASE);
121     // (2*TBPRD)/3
122     EPWM_setPhaseShift(EPWM1_BASE, 0U);
123
124     //  Define signal shape for ePWM
125         EPWM_setActionQualifierAction(EPWM1_BASE,
126                                       EPWM_AQ_OUTPUT_A,
127                                       // Set output pins to High
128                                       EPWM_AQ_OUTPUT_HIGH,
129                                       // Time base counter up equals ZERO
130                                       EPWM_AQ_OUTPUT_ON_TIMEBASE_ZERO);
131
132     EPWM_setActionQualifierAction(EPWM1_BASE,
133                                       EPWM_AQ_OUTPUT_A,
134                                       EPWM_AQ_OUTPUT_LOW,
135                                       // Time base counter up equals PERIOD
136                                       EPWM_AQ_OUTPUT_ON_TIMEBASE_PERIOD);
137
138     // Setting up EPWM02
139
140     EPWM_setTimeBaseCounter(EPWM2_BASE, 0U);
141     EPWM_setTimeBasePeriod(EPWM2_BASE, 50000U);
142     EPWM_setClockPrescaler(EPWM2_BASE,
143                                EPWM_CLOCK_DIVIDER_1,
144                                EPWM_HSCLOCK_DIVIDER_1);
145     EPWM_setTimeBaseCounterMode(EPWM2_BASE, EPWM_COUNTER_MODE_UP_DOWN);
146     EPWM_enablePhaseShiftLoad(EPWM2_BASE);
147     // 50000*(2/3) //33.3e3
148     EPWM_setPhaseShift(EPWM2_BASE, 33.3e3);
149     EPWM_setActionQualifierAction(EPWM2_BASE,
150                                       EPWM_AQ_OUTPUT_A,
151                                       EPWM_AQ_OUTPUT_LOW,
152                                       EPWM_AQ_OUTPUT_ON_TIMEBASE_PERIOD);
153
154     EPWM_setActionQualifierAction(EPWM2_BASE,
155                                       EPWM_AQ_OUTPUT_A,
156                                       EPWM_AQ_OUTPUT_HIGH,
157                                       EPWM_AQ_OUTPUT_ON_TIMEBASE_ZERO);
158
159     EPWM_setCountModeAfterSync(EPWM2_BASE, EPWM_COUNT_MODE_DOWN_AFTER_SYNC);
160
161     // Setting up EPWM03
162
163     EPWM_setTimeBaseCounter(EPWM3_BASE, 0U);
164     EPWM_setTimeBasePeriod(EPWM3_BASE, 50000U);
165     EPWM_setClockPrescaler(EPWM3_BASE,
166                                EPWM_CLOCK_DIVIDER_1,
167                                EPWM_HSCLOCK_DIVIDER_1);
168     EPWM_setTimeBaseCounterMode(EPWM3_BASE, EPWM_COUNTER_MODE_UP_DOWN);
169
170
171     EPWM_enablePhaseShiftLoad(EPWM3_BASE);
```

```
172     EPWM_setPhaseShift(EPWM3_BASE, 33.3e3);
173
174     EPWM_setActionQualifierAction(EPWM3_BASE,
175                                   EPWM_AQ_OUTPUT_A,
176                                   EPWM_AQ_OUTPUT_HIGH,
177                                   EPWM_AQ_OUTPUT_ON_TIMEBASE_ZERO);
178
179     EPWM_setActionQualifierAction(EPWM3_BASE,
180                                     EPWM_AQ_OUTPUT_A,
181                                     EPWM_AQ_OUTPUT_LOW,
182                                     EPWM_AQ_OUTPUT_ON_TIMEBASE_PERIOD);
183
184 }
185
186 // A 1 kHz with variable pulse width
187 void InitEPWM3()
188 {
189         EPWM_setTimeBaseCounter(EPWM1_BASE, 0U);
190         EPWM_setTimeBasePeriod(EPWM1_BASE, 50000);
191         EPWM_setClockPrescaler(EPWM1_BASE, EPWM_CLOCK_DIVIDER_1,
192                                            EPWM_HSCLOCK_DIVIDER_1);
193         EPWM_setTimeBaseCounterMode(EPWM1_BASE, EPWM_COUNTER_MODE_UP_DOWN);
194         EPWM_enablePhaseShiftLoad(EPWM1_BASE);
195         EPWM_setPhaseShift(EPWM1_BASE, 0U);
196
197         // Modify the VALCOMP value in order to vary the pulse width
198         VALCOMP = 35000;
199         OLDVALCOMP = 0;
200         while(1)
201         {
202             if(VALCOMP != OLDVALCOMP)
203                     {
204                         OLDVALCOMP = VALCOMP;
205                         EPWM_setCounterCompareValue(EPWM1_BASE, EPWM_COUNTER_COMPARE_A,
    VALCOMP);
206
207                         EPWM_setActionQualifierAction(EPWM1_BASE,
208                                                       EPWM_AQ_OUTPUT_A,
209                                                       EPWM_AQ_OUTPUT_HIGH,
210
    EPWM_AQ_OUTPUT_ON_TIMEBASE_ZERO);
211
212                         EPWM_setActionQualifierAction(EPWM1_BASE,
213                                                       EPWM_AQ_OUTPUT_A,
214                                                       EPWM_AQ_OUTPUT_LOW,
215
    EPWM_AQ_OUTPUT_ON_TIMEBASE_UP_CMPA);
216                     }
217         }
218 }
219
220 // A pair of complementary 1 kHz-Signals at EPWM1A and EPWM2A
221 void InitEPWM4()
222 {
223         //Setting up the EPWM01
224         EPWM_setTimeBaseCounter(EPWM1_BASE, 0U);
225         EPWM_setTimeBasePeriod(EPWM1_BASE, 50000);
```

```
226         EPWM_setClockPrescaler(EPWM1_BASE, EPWM_CLOCK_DIVIDER_1,
227                                            EPWM_HSCLOCK_DIVIDER_1);
228         EPWM_setTimeBaseCounterMode(EPWM1_BASE, EPWM_COUNTER_MODE_UP_DOWN);
229         EPWM_enablePhaseShiftLoad(EPWM1_BASE);
230         EPWM_setPhaseShift(EPWM1_BASE, 0U);
231
232         EPWM_setCounterCompareValue(EPWM1_BASE, EPWM_COUNTER_COMPARE_A, 10000);
233
234         EPWM_setActionQualifierAction(EPWM1_BASE,
235                                                EPWM_AQ_OUTPUT_A,
236                                                EPWM_AQ_OUTPUT_HIGH,
237                                                EPWM_AQ_OUTPUT_ON_TIMEBASE_ZERO);
238
239         EPWM_setActionQualifierAction(EPWM1_BASE,
240                                                EPWM_AQ_OUTPUT_A,
241                                                EPWM_AQ_OUTPUT_LOW,
242                                                EPWM_AQ_OUTPUT_ON_TIMEBASE_UP_CMPA);
243
244         //Setting up the EPWM02
245         EPWM_setTimeBaseCounter(EPWM2_BASE, 0U);
246         EPWM_setTimeBasePeriod(EPWM2_BASE, 50000U);
247         EPWM_setClockPrescaler(EPWM2_BASE,
248                                EPWM_CLOCK_DIVIDER_1,
249                                EPWM_HSCLOCK_DIVIDER_1);
250         EPWM_setTimeBaseCounterMode(EPWM2_BASE, EPWM_COUNTER_MODE_UP_DOWN);
251
252         EPWM_enablePhaseShiftLoad(EPWM2_BASE);
253         EPWM_setPhaseShift(EPWM2_BASE, 0U);
254
255         EPWM_setCounterCompareValue(EPWM2_BASE, EPWM_COUNTER_COMPARE_A, 10000);
256
257         EPWM_setActionQualifierAction(EPWM2_BASE,
258                                                EPWM_AQ_OUTPUT_A,
259                                                EPWM_AQ_OUTPUT_HIGH,
260                                                EPWM_AQ_OUTPUT_ON_TIMEBASE_UP_CMPA);
261
262         EPWM_setActionQualifierAction(EPWM2_BASE,
263                                               EPWM_AQ_OUTPUT_A,
264                                               EPWM_AQ_OUTPUT_LOW,
265                                               EPWM_AQ_OUTPUT_ON_TIMEBASE_ZERO);
266
267         EPWM_setCountModeAfterSync(EPWM2_BASE, EPWM_COUNT_MODE_DOWN_AFTER_SYNC);
268 }
269
270 // Independent Modulation of ePWM1A and ePWM1B
271 void InitEPWM5()
272 {
273         //Setting up the ePWM1A
274         EPWM_setTimeBaseCounter(EPWM1_BASE, 0U);
275         EPWM_setTimeBasePeriod(EPWM1_BASE, 50000);
276         EPWM_setClockPrescaler(EPWM1_BASE, EPWM_CLOCK_DIVIDER_1,
277                                            EPWM_HSCLOCK_DIVIDER_1);
278         EPWM_setTimeBaseCounterMode(EPWM1_BASE, EPWM_COUNTER_MODE_UP_DOWN);
279         EPWM_enablePhaseShiftLoad(EPWM1_BASE);
280         EPWM_setPhaseShift(EPWM1_BASE, 0U);
281
282         EPWM_setCounterCompareValue(EPWM1_BASE, EPWM_COUNTER_COMPARE_A, 15000);
```

```
283
284         EPWM_setActionQualifierAction(EPWM1_BASE,
285                                       EPWM_AQ_OUTPUT_A,
286                                       EPWM_AQ_OUTPUT_HIGH,
287                                       EPWM_AQ_OUTPUT_ON_TIMEBASE_UP_CMPA);
288
289         EPWM_setActionQualifierAction(EPWM1_BASE,
290                                       EPWM_AQ_OUTPUT_A,
291                                       EPWM_AQ_OUTPUT_LOW,
292                                       EPWM_AQ_OUTPUT_ON_TIMEBASE_DOWN_CMPA);
293
294         //Setting up the ePWM1B
295         EPWM_setTimeBaseCounter(EPWM1_BASE, 0U);
296         EPWM_setTimeBasePeriod(EPWM1_BASE, 50000U);
297         EPWM_setClockPrescaler(EPWM1_BASE,
298                                EPWM_CLOCK_DIVIDER_1,
299                                EPWM_HSCLOCK_DIVIDER_1);
300         EPWM_setTimeBaseCounterMode(EPWM1_BASE, EPWM_COUNTER_MODE_UP_DOWN);
301
302         EPWM_enablePhaseShiftLoad(EPWM1_BASE);
303         EPWM_setPhaseShift(EPWM1_BASE, 0U);
304
305         EPWM_setCounterCompareValue(EPWM1_BASE, EPWM_COUNTER_COMPARE_B, 35000);
306
307         EPWM_setActionQualifierAction(EPWM1_BASE,
308                                       EPWM_AQ_OUTPUT_B,
309                                       EPWM_AQ_OUTPUT_HIGH,
310                                       EPWM_AQ_OUTPUT_ON_TIMEBASE_UP_CMPB);
311         EPWM_setActionQualifierAction(EPWM1_BASE,
312                                       EPWM_AQ_OUTPUT_B,
313                                       EPWM_AQ_OUTPUT_LOW,
314                                       EPWM_AQ_OUTPUT_ON_TIMEBASE_DOWN_CMPB);
315 }
316
317 // Dead Band Unit for  ePWM1A and ePWM1B - Rising edge
318 void InitEPWM6R()
319 {
320         //Setting up the ePWM1A
321         EPWM_setTimeBaseCounter(EPWM1_BASE, 0U);
322         EPWM_setTimeBasePeriod(EPWM1_BASE, 50000);
323         EPWM_setClockPrescaler(EPWM1_BASE, EPWM_CLOCK_DIVIDER_1,
324                                            EPWM_HSCLOCK_DIVIDER_1);
325         EPWM_setTimeBaseCounterMode(EPWM1_BASE, EPWM_COUNTER_MODE_UP_DOWN);
326         EPWM_enablePhaseShiftLoad(EPWM1_BASE);
327         EPWM_setPhaseShift(EPWM1_BASE, 0U);
328
329         EPWM_setDeadBandDelayMode(EPWM1_BASE, EPWM_DB_RED, true);
330         EPWM_setRisingEdgeDeadBandDelayInput(EPWM1_BASE, EPWM_DB_INPUT_EPWMA);
331         EPWM_setRisingEdgeDelayCount(EPWM1_BASE, 750);
332
333         EPWM_setCounterCompareValue(EPWM1_BASE, EPWM_COUNTER_COMPARE_A, 25000);
334
335         EPWM_setActionQualifierAction(EPWM1_BASE,
336                                       EPWM_AQ_OUTPUT_A,
337                                       EPWM_AQ_OUTPUT_HIGH,
338                                       EPWM_AQ_OUTPUT_ON_TIMEBASE_UP_CMPA);
339
```

```
340         EPWM_setActionQualifierAction(EPWM1_BASE,
341                                       EPWM_AQ_OUTPUT_A,
342                                       EPWM_AQ_OUTPUT_LOW,
343                                       EPWM_AQ_OUTPUT_ON_TIMEBASE_DOWN_CMPA);
344
345         //Setting up the ePWM1B
346         EPWM_setTimeBaseCounter(EPWM1_BASE, 0U);
347         EPWM_setTimeBasePeriod(EPWM1_BASE, 50000U);
348         EPWM_setClockPrescaler(EPWM1_BASE,
349                                EPWM_CLOCK_DIVIDER_1,
350                                EPWM_HSCLOCK_DIVIDER_1);
351         EPWM_setTimeBaseCounterMode(EPWM1_BASE, EPWM_COUNTER_MODE_UP_DOWN);
352
353         EPWM_enablePhaseShiftLoad(EPWM1_BASE);
354         EPWM_setPhaseShift(EPWM1_BASE, 0U);
355
356
357         EPWM_setCounterCompareValue(EPWM1_BASE, EPWM_COUNTER_COMPARE_B, 25000);
358
359         EPWM_setActionQualifierAction(EPWM1_BASE,
360                                       EPWM_AQ_OUTPUT_B,
361                                       EPWM_AQ_OUTPUT_LOW,
362                                       EPWM_AQ_OUTPUT_ON_TIMEBASE_UP_CMPB);
363
364         EPWM_setActionQualifierAction(EPWM1_BASE,
365                                       EPWM_AQ_OUTPUT_B,
366                                       EPWM_AQ_OUTPUT_HIGH,
367                                       EPWM_AQ_OUTPUT_ON_TIMEBASE_DOWN_CMPB);
368 }
369
```