SVPWM SIMPLE IMPLEMENTATION GUIDE

Maciej Radosław Brzycki

There are many available publications describing the SVPWM algorithm. There are also many example implementations on GitHub. However, you may notice that there is a lack of supporting material showing how to combine these two worlds - theory and implementation, and this document is the answer to that. It shows the way starting from the theoretical side of SVPWM, successively implementations in C, and ending with the configuration of PWM on specific pins along with deadtimes.

This manual provides step-by-step advice on how to implement a simple SVPWM algorithm. The operation of the algorithm is explained, and the behavior of the time and voltage vectors with example parameters is shown - this is intended to make debugging easier for the person implementing the algorithm. The implementation (along with the physical PWM signal and deadtimes) on two popular microcontrollers - the F28379D from the **C2000** series from **Texas Instruments** and the H745ZI-Q with **cortex M7/M4** core from **STM** - is presented.

CONTENT

SVPWM CONTROL – BASICS

SVPWM works by representing the three-phase voltage of an AC motor as a single rotating vector in a two-dimensional plane, called the space vector. Instead of directly generating sinusoidal waveforms for the motor phases, SVPWM calculates which combinations of the inverter's switching states can best approximate the desired space vector at any given time. The method divides one cycle into six sectors and selects two active vectors (producing voltage) and one zero vector (no voltage) within each sector to create the desired output. By adjusting the timing (duty cycle) of these vectors, SVPWM achieves precise control over the motor's voltage and current, resulting in smoother operation, reduced harmonics and less power losses. SVPWM can produce up to 15% more voltage compared to traditional sinusoidal methods from the same DC link voltage [3].
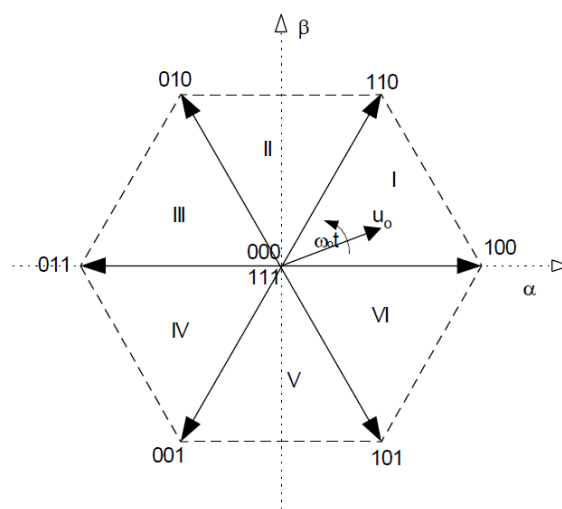

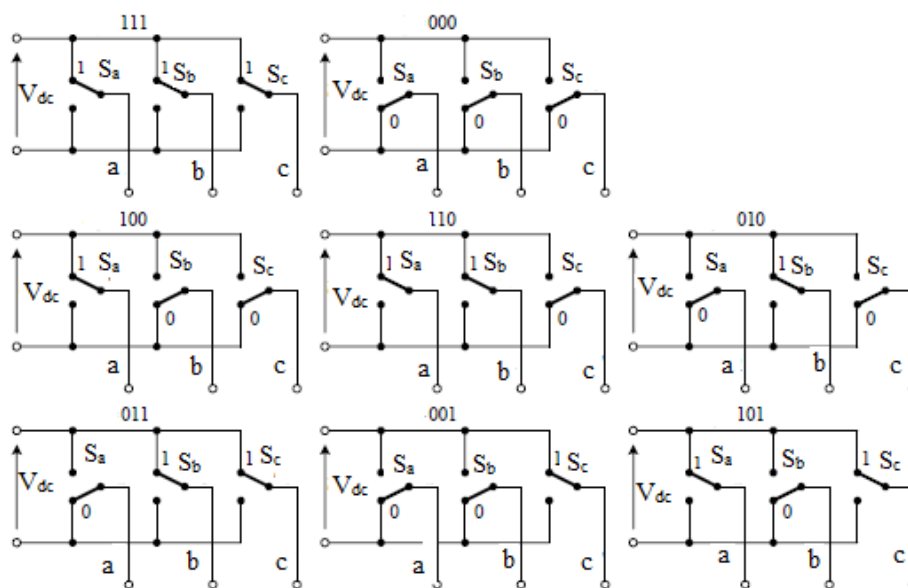Figure 1 Position sectors of the output voltage vector of the inverter [1]

Figure 2 Eight possible switching states of Voltage Source Inverter [2]

Table 1 Sectors and the corresponding values of the components of the output voltage vector of the inverter

Upper Transistor states (T1, T2, T3)	100	110	010	011	001	101	000	111
Down Transistor states (T4, T5, T6)	011	001	101	100	110	010	111	000
V_α vector voltage	$\frac{\sqrt{2}}{\sqrt{3}}U_{dc}$	$\frac{1}{\sqrt{6}}U_{dc}$	$\frac{-1}{\sqrt{6}}U_{dc}$	$\frac{-\sqrt{2}}{\sqrt{3}}U_{dc}$	$\frac{-1}{\sqrt{6}}U_{dc}$	$\frac{1}{\sqrt{6}}U_{dc}$	0	0
V_β vector voltage	0	$\frac{1}{\sqrt{2}}U_{dc}$	$\frac{1}{\sqrt{2}}U_{dc}$	0	$\frac{1}{\sqrt{2}}U_{dc}$	$\frac{-1}{\sqrt{2}}U_{dc}$	0	0

SVPWM C IMPLEMENTATION

ALFA BETA AND DQ TRANSFORMATION

To prepare the SVPWM to work with the most common type of drive control (FOC), this algorithm takes input data in the form of d, q components and electric phase angle of the machine θ.

Since the target SVPWM algorithm works in alfa and beta axes, it is necessary to make transition from the d, q to alfa beta system (1).

The transformation dq to alfa beta in many papers is expressed in matrix form [4, 5, 6]:

$$\begin{bmatrix} u_\alpha \\ u_\beta \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} u_d \\ u_q \end{bmatrix} \tag{1}$$

and it corresponds to the two equations:

$$u_\alpha = u_d\cos(\theta) - u_d\sin(\theta) \tag{2}$$
$$u_\beta = u_d\cos(\theta) - u_q\cos(\theta) \tag{3}$$

C implementation:

```
//DQ to alfa beta transformation
u_alfa = Ud * cosf(theta) - Uq * sinf(theta);
u_beta = Ud * sinf(theta) + Uq * cosf(theta);
```

If q component is not equal to zero, there is a phase shift induced between electric angle of machine θ and phase angle φ of SVPWM control (between alfa and beta component). It is crucial to recalculate φ angle.

$$\varphi = \arctan\left(\frac{u_\beta}{u_\alpha}\right)$$

```
phi = atan2f(u_beta, u_alfa) + PI;
```

ROTATING VECTOR POSITION

The algorithm calculates the current position of the voltage vector in the alfa beta axis. The floor function assigns it to one of six sectors. Depending on the current sector, the algorithm determines the two voltage vectors forming this sector (V1 and V2) and assigns the corresponding transistor states (Figure 1). V1 is the vector associated with the values corresponding to the current sector, and V2 with the values corresponding to sector + 1.

C implementation – determination of current sector:

```c
//Calculate two main vectors V1 and V2 and their alfa beta component parts
sektor = floorf(phi * one_by_pi_by_3);
switch (sektor)
{
case 0:
    u_alfa_1 = sqrt2_by_sqrt3 * U_dc;
    u_beta_1 = 0.0;
    u_alfa_2 = one_by_sqrt6 * U_dc;
    u_beta_2 = one_by_sqrt2 * U_dc;
    break;
case 1:
    u_alfa_1 = one_by_sqrt6 * U_dc;
    u_beta_1 = one_by_sqrt2 * U_dc;
    u_alfa_2 = -one_by_sqrt6 * U_dc;
    u_beta_2 = one_by_sqrt2 * U_dc;
    break;
    //etc...
```

C implementation - determination of transistor states associated with the vector:

```c
const short int OUT[6][3] = { {1,0,0},{1,1,0},{0,1,0},{0,1,1},{0,0,1},{1,0,1} }; //space vectors
```

```c
//First sector part (V1)
T1[1] = OUT[sektor][0];
T1[2] = OUT[sektor][1];
T1[3] = OUT[sektor][2];
```

```c
//Second sector part (V2)
if (sektor == 5)
{
    T2[1] = OUT[0][0];
    T2[2] = OUT[0][1];
    T2[3] = OUT[0][2];
}
else
{
    T2[1] = OUT[sektor + 1][0];
    T2[2] = OUT[sektor + 1][1];
    T2[3] = OUT[sektor + 1][2];

}
```

CALCULATING TIME VECTORS

The vector U_0 continuously spins at a given frequency and the algorithm in real time calculates the changing lengths of time vectors. The amplitude of the U_0 vector is equal to the geometric sum of the given amplitudes of the d and q components. If the q component is equal to 0, the amplitude of the U_0 vector is equal to the d component.

The vectors $U_{\alpha 1}$ and $U_{\beta 1}$ define the components of the first resultant voltage vector $\alpha\beta$ (V_1). The vectors $U_{\alpha 2}$ and $U_{\beta 2}$ contain the component values of voltages for the next vector (V_2). The SVPWM modulation operates in such a way that, using two adjacent active vectors switched on for specific durations (t_1, t_2,) it is possible to generate a voltage vector located between these vectors (U_0).
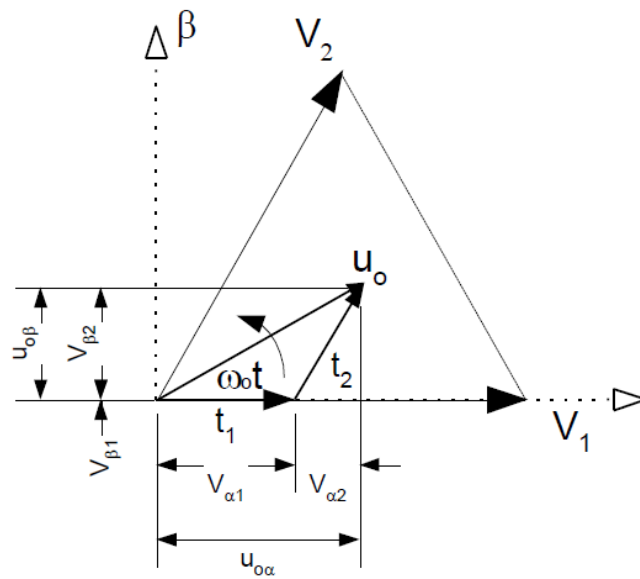

Figure 3 Determination of the component vectors of time and voltage

Formulas for the relative values of time vectors - calculated relative to one switching period. To get the actual values of these vectors, multiply them by the inverse of the switching frequency (T_{imp}) of the algorithm (for example, for 10kHz it would be 0.0001).

$$t_1 = \frac{u_{0\alpha} \cdot V_{\beta 2} - u_{0\beta} \cdot V_{\alpha 2}}{V_{\alpha 1} \cdot V_{\beta 2} - V_{\beta 1} \cdot V_{\alpha 2}} \tag{4}$$

$$t_2 = \frac{-u_{0\alpha} \cdot V_{\beta 1} - u_{0\beta} \cdot V_{\alpha 1}}{V_{\alpha 1} \cdot V_{\beta 2} - V_{\beta 1} \cdot V_{\alpha 2}} \tag{5}$$

$$t_0 = 1 - t_1 - t_2 \tag{6}$$

C implementation – time vector calculation:

```
//Calculating time vectors
time1_vector = (u_alfa * u_beta_2 - u_beta * u_alfa_2) / (u_alfa_1 * u_beta_2 - u_beta_1 * u_alfa_2);
time2_vector = ((-u_alfa * u_beta_1 + u_beta * u_alfa_1) / (u_alfa_1 * u_beta_2 - u_beta_1 * u_alfa_2));
time0_vector = (1 - time1_vector - time2_vector);
```

Adding the component vectors (*PWM_Vector*) to form the resultant vector (*duty_cycles*):

```
PWM_vector_1[0] = T1[1] * time1_vector;
PWM_vector_1[1] = T1[2] * time1_vector;
PWM_vector_1[2] = T1[3] * time1_vector;
```
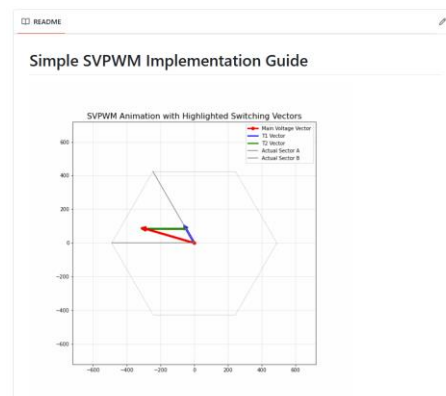
```
PWM_vector_2[0] = T2[1] * time2_vector;
PWM_vector_2[1] = T2[2] * time2_vector;
PWM_vector_2[2] = T2[3] * time2_vector;
```

```
//Sum of time vectors
duty_cycles.d1d4 = PWM_vector_1[0] + PWM_vector_2[0] + PWM_vector_0[0];
duty_cycles.d2d5 = PWM_vector_1[1] + PWM_vector_2[1] + PWM_vector_0[1];
duty_cycles.d3d6 = PWM_vector_1[2] + PWM_vector_2[2] + PWM_vector_0[2];
```

Whole c-code can be found in *SVPWM.c* and *SVPWM.h* files in GitHub repository.

https://github.com/MRadekTCZ/SVPWM-simple

SIMULATION TEST

Code written in C can be directly verified by simulation, for example, in plecs or matlab. The c-script SVPWMf is a main function called in microcontroller with 10kHz frequency. Symmetrical PWM block simulates work of PWM peripheral in μC.
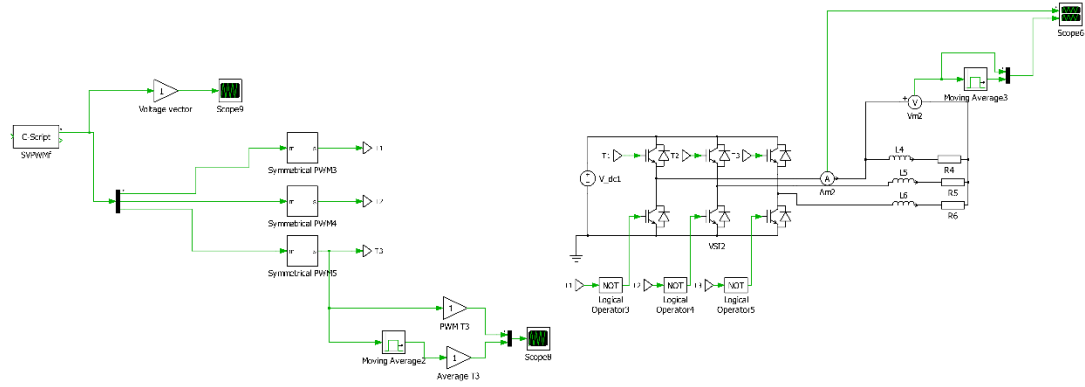
Figure 4 Example SVPWM simulation schematic in Plecs

OUTPUT VOLTAGE AND CURRENT

Phase voltage (blue) has two levels (2/3 U_{dc} and 1/3 U_{dc}). It changes with PWM switching frequency. Moving average of this voltage should create sinusoidal signal with amplitude equal to 0.866 of set U_d voltage. If load contains some inductance, the current will be filtered by it, so the load current will also be similar to sinusoid without additional averaging.
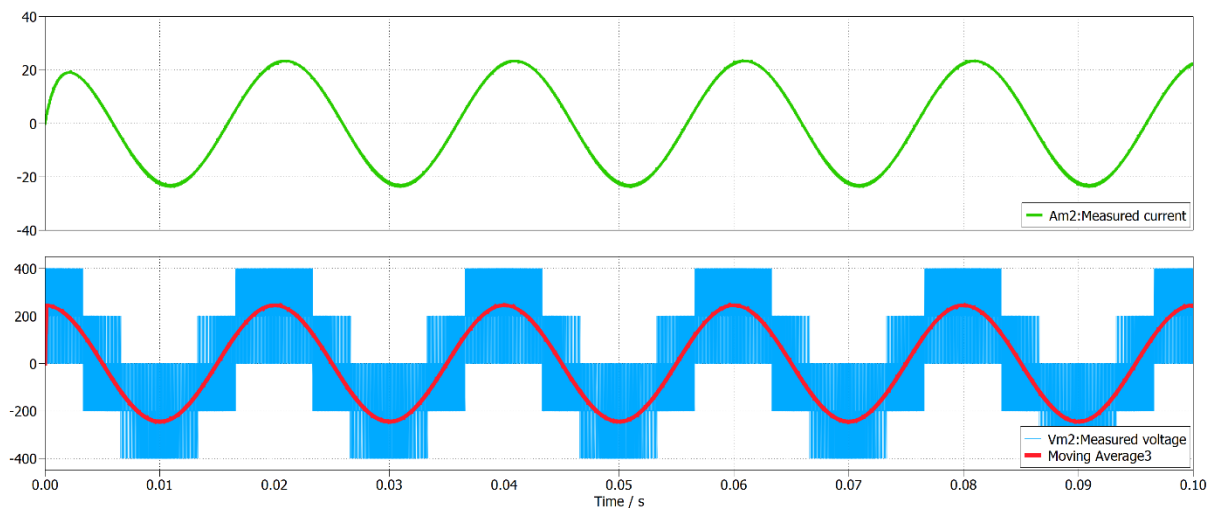
Figure 5 Current and Voltage measured on Load (L = 0.01H, R = 10Ω)

TIME VECTORS

Calculated time vectors are relative regarding one switching period. Their value can be interpreted as duty cycle for particular PWM period. Sum of the three time vectors t0, t1, t2 at any time should be equal to 1. The value of every time vector must be greater than zero and less than one. These requirements may not be met if the modulation index is too high (for a too high set amplitude).
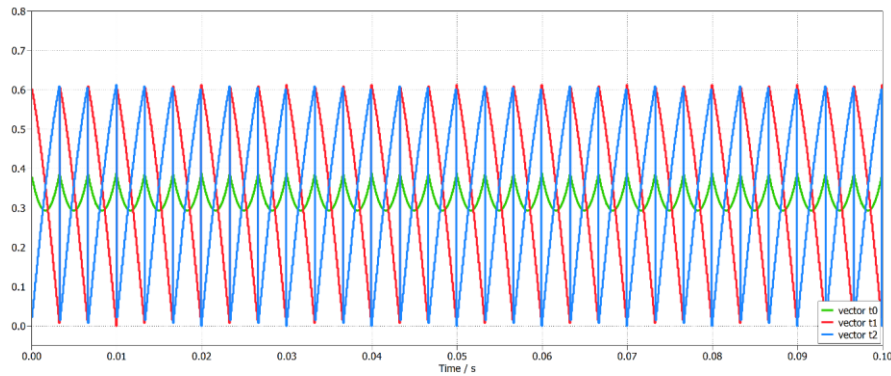
Figure 6 Time vectors

TRANSISTOR GATE SIGNAL

During a single cycle of the sine wave, a single transistor takes part in the switching in 2/3 part of the cycle. Averaged gate signal switching should look as one of the SVPWM voltage vectors (red).

Figure 7 Gate signal (green) with averaged gate signal (red)

Figure 8 Three averaged gate signals

Microcontroller implementation

For any development environment, the first implementation step will be to import the SVPWM library (located in the GitHub repository). The following code should be placed in an interrupt of a certain frequency. The whole algorithm is fit inside one function with 4 input arguments: U_d and U_q component voltages, actual angle of rotating vector and dc link voltage.
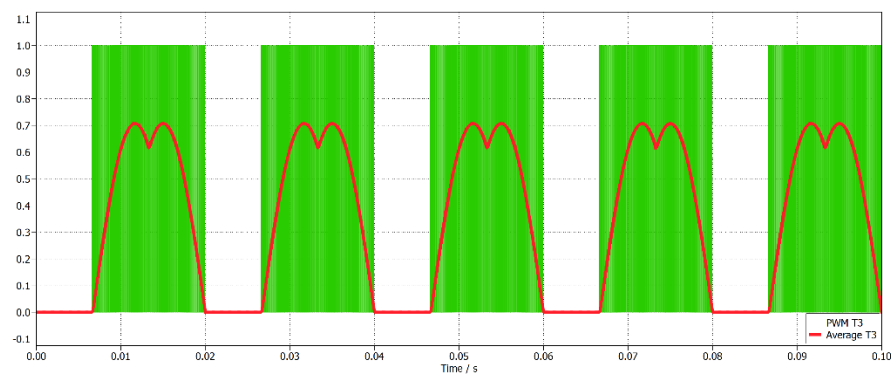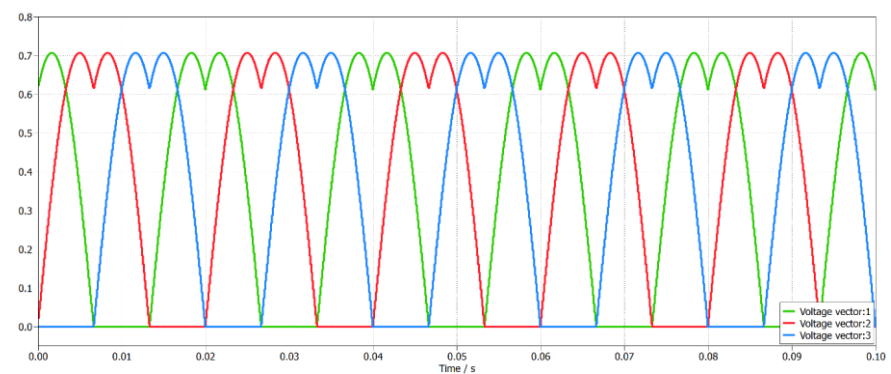
```
angle = angle + angle_dt;
if (angle >= 2 * PI)angle = angle - 2 * PI;
if (angle < 0) angle = angle + 2 * PI; // shift 2pi for negative values
svpwm = svPWM(Ud, Uq, angle, U_dc);
```

Texas Instruments C2000

The program was developed in Code Composer studio. Before testing the code remember to change floating point mode optimization to *relaxed* for floating point operations support – this will significantly boost execution speed of this algorithm.
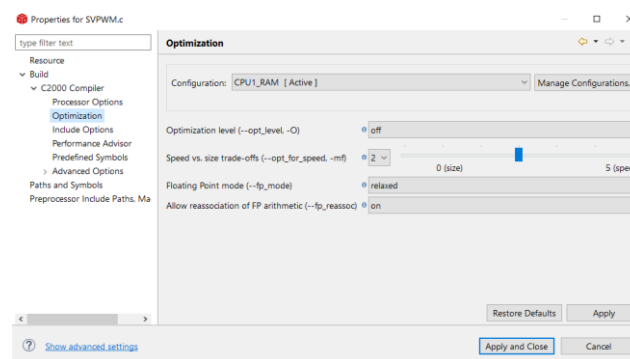

Figure 9 Changing floating point mode to relaxed

Timer configuration

To configure the 10kHz timer there is a need to call the Init and Config functions. In the example given in GitHub, both these functions are in *TI_TIMER.c* file.

```
8 #include "TI_TIMER.h"
9 void initCPUTimers(uint16_t *cpuTimer0IntCount)
10 {
11     //
12     // Initialize timer period to maximum
13     //
14     CPUTimer_setPeriod(CPUTIMER0_BASE, 0xFFFFFFFF);
15     //
16     // Initialize pre-scale counter to divide by 1 (SYSCLKOUT)
17     //
18     CPUTimer_setPreScaler(CPUTIMER0_BASE, 0);
19     //
20     // Make sure timer is stopped
21     //
22     CPUTimer_stopTimer(CPUTIMER0_BASE);
23
24     //
25     // Reload all counter register with period value
26     //
27     CPUTimer_reloadTimerCounter(CPUTIMER0_BASE);
28     //
29     // Reset interrupt counter
30     //
31     *cpuTimer0IntCount = 0;
32
33 }
```

```
34 void configCPUTimer(uint32_t cpuTimer, float freq, float period, uint16_t *cpuTimer0IntCount)
35 {
36     uint32_t temp;
37
38     //
39     // Initialize timer period:
40     //
41     temp = (uint32_t)(freq / 1000000 * period);
42     CPUTimer_setPeriod(cpuTimer, temp);
43
44     //
45     // Set pre-scale counter to divide by 1 (SYSCLKOUT):
46     //
47     CPUTimer_setPreScaler(cpuTimer, 0);
48
49     //
50     // Initializes timer control register. The timer is stopped, reloaded,
51     // free run disabled, and interrupt enabled.
52     // Additionally, the free and soft bits are set
53     //
54     CPUTimer_stopTimer(cpuTimer);
55     CPUTimer_reloadTimerCounter(cpuTimer);
56     CPUTimer_setEmulationMode(cpuTimer,
57                 CPUTIMER_EMULATIONMODE_STOPAFTERNEXTDECREMENT);
58     CPUTimer_enableInterrupt(cpuTimer);
59
60     //
61     // Resets interrupt counters for the three cpuTimers
62     //
63     if (cpuTimer == CPUTIMER0_BASE)
64     {
65         //GPIO_togglePin(DEVICE_GPIO_CFG_LED1);
66         *cpuTimer0IntCount = 0;
67     }
68
69 }
```

In main.c file, before the infite loop, but after the Board_init() function, a number of functions must be called:

```
Interrupt_register(INT_TIMER0, &cpuTimer0ISR);
initCPUTimers(&cpuTimer0IntCount);
configCPUTimer(CPUTIMER0_BASE, DEVICE_SYSCLK_FREQ, 100, &cpuTimer0IntCount); // 10000Hz
CPUTimer_enableInterrupt(CPUTIMER0_BASE);
Interrupt_enable(INT_TIMER0);
CPUTimer_startTimer(CPUTIMER0_BASE);
```

100 in timer argument means, that main timer frequency (1Mhz) is divided by 100, so the timer frequency is 10kHz.

Timer interrupt call in Texas instruments microcontroller:

```
277 //
278 // cpuTimer0ISR - Counter for CpuTimer0
279 //
280 __interrupt void
281 cpuTimer0ISR(void)
282 {
283     angle=angle + angle_dt;
284     if(angle>=2*PI)angle = angle - 2*PI;
285     if(angle<0) angle=angle+2*PI; // shift 2pi for negative values
286     svpwm = svPWM(Ud, Uq, angle, U_dc);
287     cpuTimer0IntCount++;
288     Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP1);
289 }
```

PWM CONFIGURATION

At the very beginning it is necessary to activate the pins associated with ePWM in the syscfg file.

EPWM (3 of 12 Added)		⊕ ADD ☰ REMOVE ALL
✓ myEPWM1		📋 🗑
✓ myEPWM2		📋 🗑
✓ myEPWM3		📋 🗑
Name	myEPWM1	
Use Case	ALL	▼
PinMux Peripheral and Pin Configuration		∧
EPWM Peripheral	EPWM1	▼ 🔒
EPWMA	GPIO0/160	▼ 🔒
EPWMB	GPIO1/161	▼ 🔒

Functions to configure the PWM signals are in TI_PWM.c file. Three ePWM signals are initialized. Every of these signals has two channels A and B which are negated. The declaration of every ePWM is the same with only difference in ePWM base address.

```
10 //
11 void initEPWM1(epwmInformation *epwmInfo)
12 {
13     //
14     // Set-up TBCLK
15     //
16     EPWM_setTimeBasePeriod(myEPWM1_BASE, EPWM1_TIMER_TBPRD);
17     EPWM_setPhaseShift(myEPWM1_BASE, 0U);
18     EPWM_setTimeBaseCounter(myEPWM1_BASE, EPWM1_TIMER_TBPRD);
19
20     //
21     // Set counter mode to DOWN
22     //
23     EPWM_setTimeBaseCounterMode(myEPWM1_BASE, EPWM_COUNTER_MODE_DOWN);
24
25     // Additional configuration for down-counting
26     EPWM_disablePhaseShiftLoad(myEPWM1_BASE);
27     EPWM_setClockPrescaler(myEPWM1_BASE,
28                            EPWM_CLOCK_DIVIDER_1,
29                            EPWM_HSCLOCK_DIVIDER_1);
30     //
31     // Set up shadowing for compare values
32     //
33     EPWM_setCounterCompareShadowLoadMode(myEPWM1_BASE,
34                            EPWM_COUNTER_COMPARE_A,
35                            EPWM_COMP_LOAD_ON_CNTR_ZERO);
36     EPWM_setCounterCompareShadowLoadMode(myEPWM1_BASE,
37                            EPWM_COUNTER_COMPARE_B,
38                            EPWM_COMP_LOAD_ON_CNTR_ZERO);
39     //
40     // Set actions for down-counting (invert the action behavior)
41     //
```

```
42     EPWM_setActionQualifierAction(myEPWM1_BASE,
43                            EPWM_AQ_OUTPUT_A,
44                            EPWM_AQ_OUTPUT_HIGH,
45                            EPWM_AQ_OUTPUT_ON_TIMEBASE_DOWN_CMPA);
46     EPWM_setActionQualifierAction(myEPWM1_BASE,
47                            EPWM_AQ_OUTPUT_A,
48                            EPWM_AQ_OUTPUT_LOW,
49                            EPWM_AQ_OUTPUT_ON_TIMEBASE_ZERO);
50     EPWM_setActionQualifierAction(myEPWM1_BASE,
51                            EPWM_AQ_OUTPUT_B,
52                            EPWM_AQ_OUTPUT_HIGH,
53                            EPWM_AQ_OUTPUT_ON_TIMEBASE_DOWN_CMPB);
54     EPWM_setActionQualifierAction(myEPWM1_BASE,
55                            EPWM_AQ_OUTPUT_B,
56                            EPWM_AQ_OUTPUT_LOW,
57                            EPWM_AQ_OUTPUT_ON_TIMEBASE_ZERO);
58     //
59     // Interrupt on time base counter zero event, generate INT on 3rd event
60     //
61     EPWM_setInterruptSource(myEPWM1_BASE, EPWM_INT_TBCTR_ZERO);
62     EPWM_enableInterrupt(myEPWM1_BASE);
63     EPWM_setInterruptEventCount(myEPWM1_BASE, 3U);
64     //
65     // Initialize ePWM information structure
66     //
67     epwmInfo->epwmCompADirection = EPWM_CMP_UP;  // Not used anymore
68     epwmInfo->epwmCompBDirection = EPWM_CMP_UP;  // Not used anymore
69     epwmInfo->epwmTimerIntCount = 0U;
70     epwmInfo->epwmModule = myEPWM1_BASE;
71     epwmInfo->epwmMaxCompA = EPWM1_MAX_CMPA;
72     epwmInfo->epwmMinCompA = EPWM1_MIN_CMPA;
73     epwmInfo->epwmMaxCompB = EPWM1_MAX_CMPB;
74     epwmInfo->epwmMinCompB = EPWM1_MIN_CMPB;
75 }
```

Pwm signal initialization in main.c:

Interrupt_register(INT_EPWM1, &epwm1ISR);
Interrupt_register(INT_EPWM2, &epwm2ISR);
Interrupt_register(INT_EPWM3, &epwm3ISR);
initEPWM1(&epwm1Info);
initEPWM2(&epwm2Info);
initEPWM3(&epwm3Info);
Interrupt_enable(INT_EPWM1);
Interrupt_enable(INT_EPWM2);
Interrupt_enable(INT_EPWM3);

```
185 __interrupt void epwm1ISR(void)
186 {
187     //
188     // Update the CMPA and CMPB values
189     //
190     updateCompare(&epwm1Info, svpwm.d1d4);
191
192     //
193     // Clear INT flag for this timer
194     //
195     EPWM_clearEventTriggerInterruptFlag(myEPWM1_BASE);
196
197     //
198     // Acknowledge interrupt group
199     //
200     Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP3);
201 }
```

With every PWM interrupt *updateCompare* function is called. One of this function arguments is duty cycle of the vector calculated by SVPWM algorithm. This is how PWM counting time is changed every cycle.

```
247 void updateCompare(epwmInformation *epwmInfo, float duty)
248 {
249
250     uint16_t newDutyCycleA = 0;
251
252     newDutyCycleA = duty*EPWM1_TIMER_TBPRD;
253
254
255     // Update Compare A and Compare B values
256     EPWM_setCounterCompareValue(epwmInfo->epwmModule, EPWM_COUNTER_COMPARE_A, newDutyCycleA);
257     //EPWM_setCounterCompareValue(epwmInfo->epwmModule, EPWM_COUNTER_COMPARE_B, newDutyCycleB);
258
259     //
260     // Increment timer interrupt count
261     epwmInfo->epwmTimerIntCount++;
262
263     // Reset the interrupt flag if needed
264     EPWM_clearEventTriggerInterruptFlag(epwmInfo->epwmModule);
265
266     // Acknowledge the interrupt group
267     Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP3);
268 }
```

Before using PWM it is necessary to define proper counting periods. In *TI_PWM.h* file there are defined values:

#define EPWM1_TIMER_TBPRD 10000U
#define EPWM1_MAX_CMPA   9999U
#define EPWM1_MIN_CMPA   0U
#define EPWM1_MAX_CMPB   9999U
#define EPWM1_MIN_CMPB   0U

EPWM1_TIMER_TBPRD is a value to which PWM counter counts in one period. Clock frequency of TMS-F28379D microcontroller is 200MHz. The highest possible PWM counting frequency is equal to main clock frequency divided by 2 (100Mhz). If *EPWM1_TIMER_TBPRD* is equal to 10000, then PWM timer will count to 10000 clock cycles in one period, so PWM switching frequency will be 10000Hz.

DEADTIME CONFIGURATION

In *TI_PWM.c* file there are multiple deadtime activation functions (eg. *setupEPWMOutputSwap, setupEPWMActiveHighComplementary, setupEPWMActiveLow*). For SVPWM the most suitable deadtime function is *setupEPWMActiveHighComplementary*.

```
321 void setupEPWMActiveHighComplementary(uint32_t base)
322 {
323     //
324     // Use EPWMA as the input for both RED and FED
325     //
326     EPWM_setRisingEdgeDeadBandDelayInput(base, EPWM_DB_INPUT_EPWMA);
327     EPWM_setFallingEdgeDeadBandDelayInput(base, EPWM_DB_INPUT_EPWMA);
328
329     //
330     // Set the RED and FED values
331     //
332     EPWM_setFallingEdgeDelayCount(base, 10);
333     EPWM_setRisingEdgeDelayCount(base, 10);
334
335     //
336     // Invert only the Falling Edge delayed output (AHC)
337     //
338     EPWM_setDeadBandDelayPolarity(base, EPWM_DB_RED, EPWM_DB_POLARITY_ACTIVE_HIGH);
339     EPWM_setDeadBandDelayPolarity(base, EPWM_DB_FED, EPWM_DB_POLARITY_ACTIVE_LOW);
340
341     //
342     // Use the delayed signals instead of the original signals
343     //
344     EPWM_setDeadBandDelayMode(base, EPWM_DB_RED, true);
345     EPWM_setDeadBandDelayMode(base, EPWM_DB_FED, true);
346
347     //
348     // DO NOT Switch Output A with Output B
349     //
350     EPWM_setDeadBandOutputSwapMode(base, EPWM_DB_OUTPUT_A, false);
351     EPWM_setDeadBandOutputSwapMode(base, EPWM_DB_OUTPUT_B, false);
352
353 }
```

The value of deadtime prescaler for both falling and rising edge is equal to **10**. It means that the chosen deadtime period is equal to 10 PWM clock cycles (100ns).

Activating deadtimes in *main.c* file:

setupEPWMActiveHighComplementary(myEPWM1_BASE);
setupEPWMActiveHighComplementary(myEPWM2_BASE);
setupEPWMActiveHighComplementary(myEPWM3_BASE);

STM32

Program and configuration were developed in STM cube IDE.

TIMER CONFIGURATION

In this program, two timers are used - one for the interrupts associated with the main algorithm with a frequency of 10 kHz - TIM13 and the other for PWM generation (also with a switching frequency of 10kHz - TIM1). Before configuring the timers, it is necessary to find the clock frequency associated with the particular timer base. There are two main timer frequencies - APB1 and APB2 timer clocks. Check the documentation or configuration files with which frequency the currently configured timer is associated. For the STM32H745ZI microcontroller, Timer 13 is associated with the APB1 frequency (figure 1. in STN32H745 datasheet [7]).
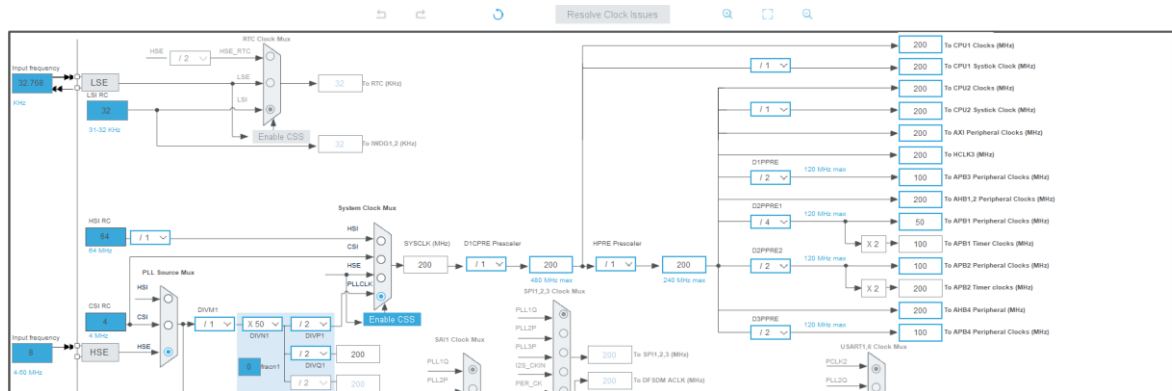
Since the frequency of APB1 is 100 Mhz, after setting the prescaler to 99 (counting starts from 0 rather than 1) and the counting period to 99, the resulting frequency is 100000000/(100*100) is 10kHz.
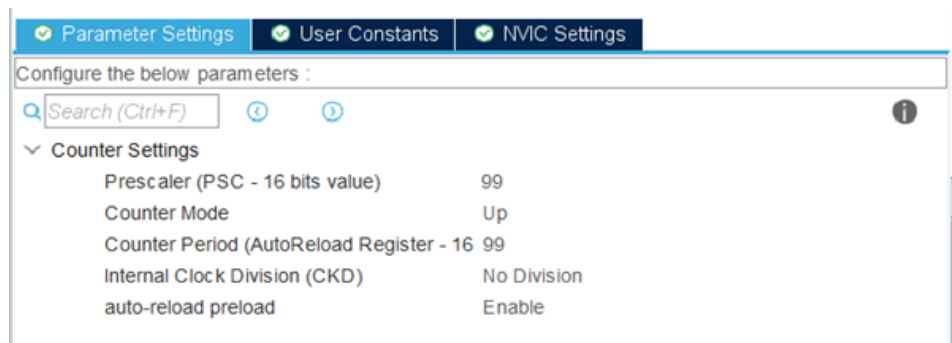
Be sure to set auto-reload preload to enable and allow interrupts in the NVIC panel

NVIC1 Interrupt Table	Enabled	Preemption Priority	Sub Priority
TIM8 update interrupt and TIM13 global interrupt	☑	0	0

In the code, timer is activated with the command:

```
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start_IT(&htim13);
```

PWM CONFIGURATION

To initialize SVPWM control, 3 PWM channels will be necessary - each with two complementary outputs. The PWM counts pulses at APB1 frequency, and the counting period is 10000 pulses - the PWM switching frequency is 10 kHz. The count mode should be set as PWM mode 2 (it will matter later whether the deadtime is implemented as two enabled outputs or two disabled outputs). The polarity of CH and CHN should be set for both as HIGH (CHN initially is negated, so negating it here would give a double negation effect). For the same reason, the Idle state of both channels should be set as reset.

✔ User Constants	✔ NVIC Settings	✔ DMA Settings	✔ GPIO Settings
	✔ Parameter Settings		

Configure the below parameters :

Q Search (Ctrl+F)   ◁   ▷   ⓘ

∨ Counter Settings
Prescaler (PSC - 16 bits value)	0
Counter Mode	Up
Counter Period (AutoReload Register - 16 bit..9999	
Internal Clock Division (CKD)	No Division
Repetition Counter (RCR - 16 bits value)	0
auto-reload preload	Enable

Runtime contexts:
Cortex-M7	Cortex-M4	PowerDomain
☑	☐	D2

Slave Mode	Disable	∨
Trigger Source	Disable	∨
Clock Source	Internal Clock	∨
Channel1	PWM Generation CH1 CH1N	∨
Channel2	PWM Generation CH2 CH2N	∨
Channel3	PWM Generation CH3 CH3N	∨

✔ NVIC Settings	✔ DMA Settings	✔ GPIO Settings
✔ Parameter Settings	✔ User Constants	

NVIC1 Interrupt Table	Enabled	Preemption Priority	Sub Priority
TIM1 break interrupt	☑	0	0
TIM1 update interrupt	☑	0	0
TIM1 trigger and commutation interrupts	☑	0	0
TIM1 capture compare interrupt	☑	0	0

∨ PWM Generation Channel 1 and 1N
Mode	PWM mode 2
Pulse (16 bits value)	0
Output compare preload	Enable
Fast Mode	Disable
CH Polarity	High
CHN Polarity	High
CH Idle State	Reset
CHN Idle State	Reset

Be sure to initialize the complementary channels in the code in addition to the basic initialization of the PWM channels:

```
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2);
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_3);
HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_1);
HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_2);
HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_3);
// Enable the complementary output
__HAL_TIM_MOE_ENABLE(&htim1);
```

DEADTIME CONFIGURATION

Deadtime configured to last 20 clock cycles of APB1 (200ns)

∨ Break And Dead Time management - Output ...
Automatic Output State   Enable
Off State Selection for Run Mode (OSSR) Disable
Off State Selection for Idle Mode (OSSI)   Disable
Lock Configuration   Off
Dead Time   19
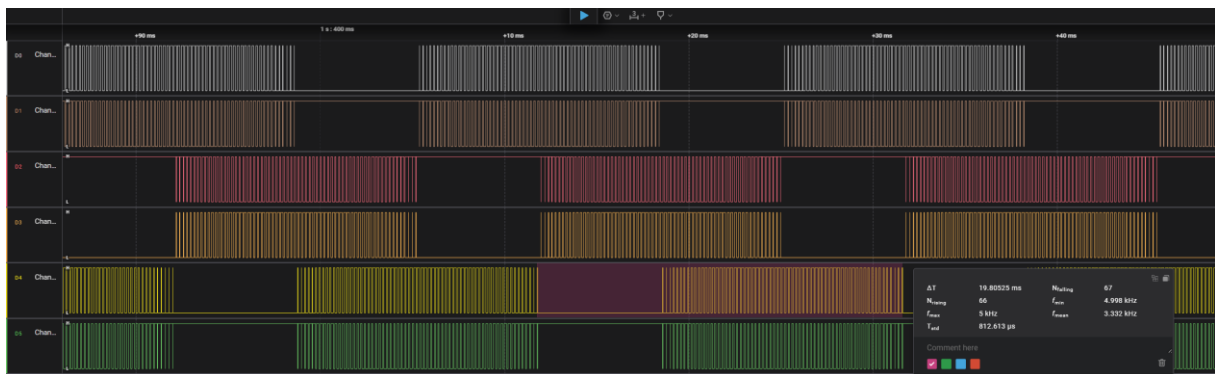
RESULT VERIFICATION

Figure 10 Transistor gate signals overview - generated by microcontroller
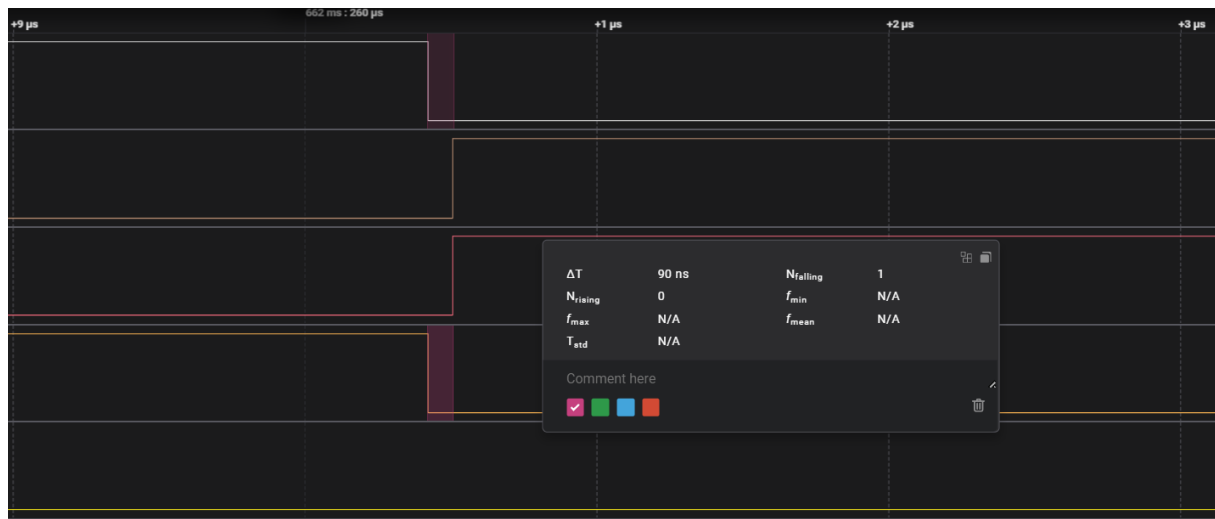

Figure 11 Gate signal overview - zoom in

Figure 12 Deadtime on rising edge and falling edge

REFERENCES

[1] A. Lewicki *Space Vector PWM. Effect of dead time on drive operation* - lab manual.

[2] Pradeep, Jayarama & Devanathan, Rajagopalan. (2015). *Adoption of Park's Transformation for Inverter Fed Drive. International Journal of Power Electronics and Drive Systems*. 5. 366-373. 10.11591/ijpeds.v5i3.6849.

[3] N. Prakash, J. Jacob and V. Reshmi, "Comparison of DVR performance with Sinusoidal and Space Vector PWM techniques," *2014 Annual International Conference on Emerging Research Areas: Magnetics, Machines and Drives (AICERA/iCMMD)*, Kottayam, India, 2014, pp. 1-6, doi: 10.1109/AICERA.2014.6908196.

[4] Z. Wen and S. He, "Analysis of three-phase magnitude-phase detection method based on double dq transformation," *Proceedings of The 7th International Power Electronics and Motion Control Conference*, Harbin, China, 2012, pp. 340-344, doi: 10.1109/IPEMC.2012.6258869.

[5] M. Gonzalez, V. Cardenas and F. Pazos, "DQ transformation development for single-phase systems to compensate harmonic distortion and reactive power," *9th IEEE International Power Electronics Congress, 2004. CIEP 2004*, Celaya, Mexico, 2004, pp. 177-182, doi: 10.1109/CIEP.2004.1437575.

[6] V. M. Goswami and K. Vakharia, "High Performance Induction Machine Drive Using Rotor Field Oriented Control," *2019 International Conference on Intelligent Sustainable Systems (ICISS)*, Palladam, India, 2019, pp. 559-564, doi: 10.1109/ISS1.2019.8907959.

[7] STM32H745ZxI datasheet - https://www.st.com/resource/en/datasheet/stm32h745zg.pdf