Digital Control in Power Electronics

Regular Sampling Technique on TI f28379D

DSP Configuration

Ruzica Cvetanovic

August 19, 2020

1 Introduction

This document is intended as an explanation for the configuration of TI F28379 peripherals relevant for a typical power electronics application. Implementation discussed in this document assumes regular sampling technique, whereas it is configurable for both single and double update rate with and without oversampling based feedback acquisition (one switching period averaging). The code has been developed using two single pole systems - series RC connections - driven independently by PWM outputs. Therefore, some slight modifications exist in the configuration of ePWM outputs compared to a typical half bridge configuration, as it will be explained in the following section.

2 EPWM configuration

We differentiate between two configurations of PWM channels. The one intended for switching (slower counter) and the other aimed at triggering ADC (faster counter). The former one will be explained first. Source code for the configuration of ePWM is provided below.

Depending on the application, several ePWMs might be configured for switching. In this specific, only ePWM1 is used for switching PWM. Counter period PWM_TBPRD is determined according to the desired switching frequency (FSW). In this specific case $FSW = 10kHz$. Note that counter period is adjusted so that it is multiple of a desired number of samples to be measured on switching period (NOS). Counter mode is set to up-down in order to make the realization of the double update rate possible. Time base counter clock ratio to ePWM clock (which is half of the system clock) is set to 1. Shadowing of the compare registers is enabled and CMPA/CMPB load event is defined by the update rate. In case of a single update rate CMPA/CMPB load occurs on TBCTR=0, while in case of a double update rate loading of the compare registers is allowed on either zero or period counter value. Actions are set using Action qualifier register. PWM output A/B is cleared during the up count when CMPA/B=TBCTR, and set during the down count on the same event. In the specific application PWMA is used for one and PWMB for the other RC load, whereas PWMA and PWMB are regulated independently. In a typical power electronics application, one of the PWM outputs should be inversion of the other. Dead time can be configured either using dedicated hardware (DB), or via adjusting the desired duty cycle in software.

The second of the aforementioned ePWM configurations is faster PWM used for ADC triggering to implement oversampling based feedback acquisition. EPWM5 is used for this purpose. Counter period ADC_TBPRD is determined according to the desired number of samples to be measured on switching period (NOS) and period of the switching counter (PWM_TBPRD). Counter mode is set to up-down. Time base counter clock ratio to ePWM clock is set to 1. Start of conversion signal is set to trigger ADC whenever TBCTR=0.

It is important to emphasize that in order to ensure intended behaviour, it is essential that two PWMs are synchronized. Since phasing of each PWM is disabled by default, by stopping ePWM clock and syncing it back on with CPU after configuration of all PWMs, it is ensured that all PWMs are synchronized.

```
#define UR 2                                        // Update rate (double or single)
#define OVERSAMPLING 1                              // Logic variable to differentiate between
     case with and without oversampling
#define NOS 16                                      // Number of samples to be measured on PWM
     period (if oversampling==1 NOS is oversampling factor)
#define NOS_UR (NOS/UR)                             // Ratio between NOS and UR
#define LOG2_NOS_UR (log2(NOS_UR))                  // Used for averaging if oversampling==1
#define FTB 100e6                                   // FTB=EPWMCLK=SYSCLKOUT/2 (time base
    clock ratio to EPWM clock = 1 assumed)
#define FPWM 10e3                                   // Switching frequency
// Counter period for ePWM used for switching, up-down mode assumed; closest to (Uint16)(FTB
    /(2*FPWM)-1) so that PWM_TBPRD%16=0
#define PWM_TBPRD 4992
#define TPWM (2*PWM_TBPRD/FTB)                      // Switching period
#define ADC_TBPRD  (PWM_TBPRD/NOS)                  // Counter period for ePWM used for ADC
    triggering, up-down mode assumed
#define TS (TPWM/UR)                                // Regulation period
#define LOAD_CMPA (UR==1 ? 0 : 2)                   // Load CMPA on TBCTR=0 if UR==1,
    otherwise on both TBCTR=0 and TBCTR=TBPRD
#define LOAD_CMPB (UR==1 ? 0 : 2)                   // Load CMPB on TBCTR=0 if UR==1,
    otherwise on both TBCTR=0 and TBCTR=TBPRD
void Configure_ePWM(void)
{

    // EPWM1 for switching

    EPwm1Regs.TBCTL.bit.CLKDIV =  0;            // CLKDIV=1 TBCLK=EPWMCLK/(HSPCLKDIV*CLKDIV)
    EPwm1Regs.TBCTL.bit.HSPCLKDIV = 0;          // HSPCLKDIV=1
    EPwm1Regs.TBCTL.bit.CTRMODE = 2;            // Up-down mode
    EPwm1Regs.TBCTR = 0x0000;                   // Clear counter
    EPwm1Regs.TBCTL.bit.PHSEN = 0;              // Phasing disabled

    EPwm1Regs.TBPRD = PWM_TBPRD;                 // Counter period

    EPwm1Regs.CMPCTL.bit.SHDWAMODE = 0;                 // Shadow mode active for CMPA
    EPwm1Regs.CMPCTL.bit.SHDWBMODE = 0;                 // Shadow mode active for CMPB
    EPwm1Regs.CMPCTL.bit.LOADAMODE = LOAD_CMPA;         // Load on TBCTR=0 if UR==1, otherwise
     on either TBCTR=0 or TBCTR=TBPRD
    EPwm1Regs.CMPCTL.bit.LOADBMODE = LOAD_CMPB;         // Load on TBCTR=0 if UR==1, otherwise
     on either TBCTR=0 or TBCTR=TBPRD

    EPwm1Regs.CMPA.bit.CMPA = 0;                // Value of the CMPA at the beginning
    EPwm1Regs.AQCTLA.bit.CAU = 1;               // Set EPWMA low on TBCTR=CMPA during up count
    EPwm1Regs.AQCTLA.bit.CAD = 2;               // Set EPWMA high on TBCTR=CMPA during down
    count

    EPwm1Regs.CMPB.bit.CMPB = 0;                // Value of the CMPB at the beginning
    EPwm1Regs.AQCTLB.bit.CBU = 1;               // Set EPWMB low on TBCTR=CMPB during up count
    EPwm1Regs.AQCTLB.bit.CBD = 2;               // Set EPWMB high on TBCTR=CMPB during down
    count

    // EPWM5 for ADC triggering

    EPwm5Regs.TBCTL.bit.CLKDIV =  0;            // CLKDIV=1 TBCLK=EPWMCLK/(HSPCLKDIV*CLKDIV)
    EPwm5Regs.TBCTL.bit.HSPCLKDIV = 0;          // HSPCLKDIV=1
    EPwm5Regs.TBCTL.bit.CTRMODE = 2;            // Up-down mode
    EPwm5Regs.TBCTR = 0x0000;                   // Clear counter
    EPwm1Regs.TBCTL.bit.PHSEN = 0;              // Phasing disabled

    EPwm5Regs.TBPRD = ADC_TBPRD;                // Counter period
    EPwm5Regs.ETSEL.bit.SOCASEL = 1;            // ADCSOCA on TBCTR=0
    EPwm5Regs.ETPS.bit.SOCAPRD = 1;             // Generate SOCA on 1st event
    EPwm5Regs.ETSEL.bit.SOCAEN = 1;             // Enable SOCA generation

    // set PWM output to notify SOCA (JUST FOR DEBUGGING)
    EPwm5Regs.AQCTLA.bit.ZRO = 2;               // Set PWM A high on TBCTR=0
    EPwm5Regs.AQCTLA.bit.PRD = 1;               // Set PWM A low on TBCTR=TBPRD


}
```

3 ADC configuration

In order to measure two variables (capacitor voltage of both RC loads), ADCA and ADCB are used to measure variables simultaneously and are configured in a similar way. Source code for the configuration of ADC is provided below. ADC clock is set be be 4 times slower than the system clock. Twelve bit resolution and a single mode conversion is used. Pulse position is set to late and acquisition window is set to 31 system clock periods. Start of conversion is triggered by ePWM5 SOC signal. ADCA converts channel 0, whereas ADCB converts channel 2. ADCA's end of conversion sets ADCAINT1 flag. This signal is used to trigger DMA.

```
void Configure_ADC(void)
{
    EALLOW;

    // configure ADCA

    AdcaRegs.ADCCTL2.bit.PRESCALE = 6;          // Set ADCCLK divider; ADCCLK=SYSCLK/4
    AdcSetMode(ADC_ADCA, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE); // Set the resolution
    and signal mode for a given ADC
    AdcaRegs.ADCCTL1.bit.INTPULSEPOS = 1;       // Set pulse positions to late
    AdcaRegs.ADCCTL1.bit.ADCPWDNZ = 1;          // Power up the ADC
    DELAY_US(1000);                             // Delay for 1ms to allow ADC time to power up

    AdcaRegs.ADCSOC0CTL.bit.CHSEL = 0;          // SOC0 will convert pin A0
    AdcaRegs.ADCSOC0CTL.bit.ACQPS = 30;         // Acquisition window is ACQPS+1 SYSCLK cycles
    AdcaRegs.ADCSOC0CTL.bit.TRIGSEL = 13;       // Trigger on ePWM5 SOC0

    AdcaRegs.ADCINTSEL1N2.bit.INT1SEL = 0;      // EOC A0 will set ADCINT1 flag
    AdcaRegs.ADCINTSEL1N2.bit.INT1E = 1;        // Enable ADCINT1
    AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;      // Make sure INT1 flag is cleared


    // configure ADCB

    AdcbRegs.ADCCTL2.bit.PRESCALE = 6;          // Set ADCCLK divider; ADCCLK=SYSCLK/4
    AdcSetMode(ADC_ADCB, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE); // Set the resolution
    and signal mode for a given ADC
    AdcbRegs.ADCCTL1.bit.INTPULSEPOS = 1;       // Set pulse positions to late
    AdcbRegs.ADCCTL1.bit.ADCPWDNZ = 1;          // Power up the ADC
    DELAY_US(1000);                             // Delay for 1ms to allow ADC time to power up

    AdcbRegs.ADCSOC0CTL.bit.CHSEL = 2;          // SOC0 will convert pin B2 (B2 is available
    on launch pad)
    AdcbRegs.ADCSOC0CTL.bit.ACQPS = 30;         // Acquisition window is ACQPS+1 SYSCLK cycles
    AdcbRegs.ADCSOC0CTL.bit.TRIGSEL = 13;       // Trigger on ePWM5 SOC0

    // JUST FOR DEBUGGING
    AdcbRegs.ADCINTSEL1N2.bit.INT1SEL = 2;      // EOC B2 will set ADCINT1 flag
    AdcbRegs.ADCINTSEL1N2.bit.INT1E = 0;        // Enable ADCINT1
    AdcbRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;      // Make sure INT1 flag is cleared

    EDIS;
}
```

4 DMA configuration

Only one DMA channel (DMACH1) is used to collect and transfer data from both ADCA and ADCB. Source code for the configuration of DMA is provided below. Trigger and synchronization source of DMACH1 is set to ADCAINT1. Peripheral interrupt source is set to ADCAINT1. One shot mode is disabled, meaning that one burst per SW interrupt occurs. Continuous mode is enabled so that DMA does not stop after each transfer. Data size to be transferred is 16 bit and ePIE interrupt is generated at the beginning of each transfer. Configuration of burst and transfer registers will be explained further on. For a better understanding see Fig. 1.

Burst registers are configured in a following manner. In each burst two 16 bit words are transferred from ADC result registers to DMA buffer. Source burst address is incremented by 32 because address of ADCB result is located 32 16 bit memory slots below ADCA result. Destination burst address increment is set to NOS/UR, because first NOS/UR 16 bit words in DMA buffer are used for ADCA results, and remaining NOS/UR are used for ADCB results.

Regarding configuration of transfer registers, transfer size is set so that within one transfer, NOS/UR bursts are completed. Source transfer address increment is set to -32 because when the burst is completed, the source address points to the address of ADCB result, and we want that the next burst starts from ADCA result. Destination transfer step is set to $-(NOS/UR-1)$ (to go back to the part of the DMA buffer which is relevant for ADCA results). Once the transfer is completed (all NOS/UR bursts are done), DMA interrupt flag is set.

Wrapping of both destination and source registers is disabled by setting wrap size to be larger than transfer size. Both source addresses and starting source address point at ADCA result 0. Both destination address and starting destination address point at the DMA buffer 1. Note that the destination address should be toggled in DMA interrupt service routine between DMA buffers 1 and 2 (ping-pong algorithm), in order to prevent a potential data loss due to simultaneously taking data and writing new one in the DMA buffer.

```
void Configure_DMA(void)
{
    EALLOW;

    // Configure CH1

    // Set up MODE Register
    DmaClaSrcSelRegs.DMACHSRCSEL1.bit.CH1 = 1;       // Select the Trigger and Sync Source of
    DMACH1 to ADCAINT1
    DmaRegs.CH1.MODE.bit.PERINTSEL = 1;              // ADCAINT1 as peripheral interrupt source
    DmaRegs.CH1.MODE.bit.PERINTE = 1;                // Peripheral interrupt enabled
    DmaRegs.CH1.MODE.bit.ONESHOT = 0;                // 1 burst per SW interrupt
    DmaRegs.CH1.MODE.bit.CONTINUOUS = 1;             // Do not stop after each transfer
    DmaRegs.CH1.MODE.bit.DATASIZE = 0;               // 16-bit data size transfers
    DmaRegs.CH1.MODE.bit.CHINTMODE = 0;              // Generate ePIE interrupt at the
    beginning of transfer
    DmaRegs.CH1.MODE.bit.CHINTE = 1;                 // Channel Interrupt to CPU enabled

    // Set up BURST registers
    DmaRegs.CH1.BURST_SIZE.all = 1;                   // Number of 16-bit words per burst (N-1)
    DmaRegs.CH1.SRC_BURST_STEP = 32;                  // Increment source burst address by 32
    (16-bit) (ADCB result address = ADCA result address + 32)
    DmaRegs.CH1.DST_BURST_STEP = NOS_UR;              // Increment destination burst address by
     NOS_UR (16-bit)

    // Set up TRANSFER registers
    DmaRegs.CH1.TRANSFER_SIZE = NOS_UR-1;             // Number of bursts per transfer (N-1)
    DmaRegs.CH1.SRC_TRANSFER_STEP = -32;              // Increment source transfer address by
    -32 (16-bit)
    DmaRegs.CH1.DST_TRANSFER_STEP = -(NOS_UR-1);      // Increment destination transfer
    address by -(NOS_UR-1) (16-bit)

    // Set up WRAP registers
    DmaRegs.CH1.SRC_WRAP_SIZE = 17;                   // Wrap after 17 burst (wrapping
    disabled because wrap_size > transfer_size)
    DmaRegs.CH1.DST_WRAP_SIZE = 17;                   // Wrap after 17 burst (wrapping
    disabled because wrap_size > transfer_size)
    DmaRegs.CH1.SRC_WRAP_STEP = 0;
    DmaRegs.CH1.DST_WRAP_STEP = 0;

    // Set up SOURCE address
    DmaRegs.CH1.SRC_BEG_ADDR_SHADOW = (Uint32)&AdcaResultRegs.ADCRESULT0;    // Point to
    beginning of source buffer
    DmaRegs.CH1.SRC_ADDR_SHADOW =     (Uint32)&AdcaResultRegs.ADCRESULT0;    // Source address

    // Set up DESTINATION address
    DmaRegs.CH1.DST_BEG_ADDR_SHADOW = (Uint32)&DMAbuffer1[0];                // Point to
    beginning of destination buffer
    DmaRegs.CH1.DST_ADDR_SHADOW =     (Uint32)&DMAbuffer1[0];                // Toggle
    destination address in dmach1_isr

    // Clear any spurious flags & errors
    DmaRegs.CH1.CONTROL.bit.PERINTCLR = 1;            // Clear any spurious interrupt flags
    DmaRegs.CH1.CONTROL.bit.ERRCLR = 1;               // Clear any spurious sync error

    EDIS;
}
```

Logic variable $OVERSAMPLING$ is used to differentiate between cases with and without oversampling. In case of oversampling based feedback acquisition, averaging of the collected data is done in DMA interrupt service routine. If a single update rate is chosen (by setting UR to 1), DMA interrupt occurs once per switching period and it is enough to do averaging of the data stored in DMA buffer for each of the measured variables (regulation period averaging). If a double update rate is chosen (by setting UR to 2), DMA interrupt occurs twice per switching period, and apart from the afore mentioned averaging, we need to perform additional one to achieve oversampling rate of NOS on the switching period. In case without oversampling, the last measurement stored in DMA buffer is used as a measurement result. For a better understanding see below part of the DMA interrupt service routine source code relevant for the implementation of the ping-pong algorithm and averaging.

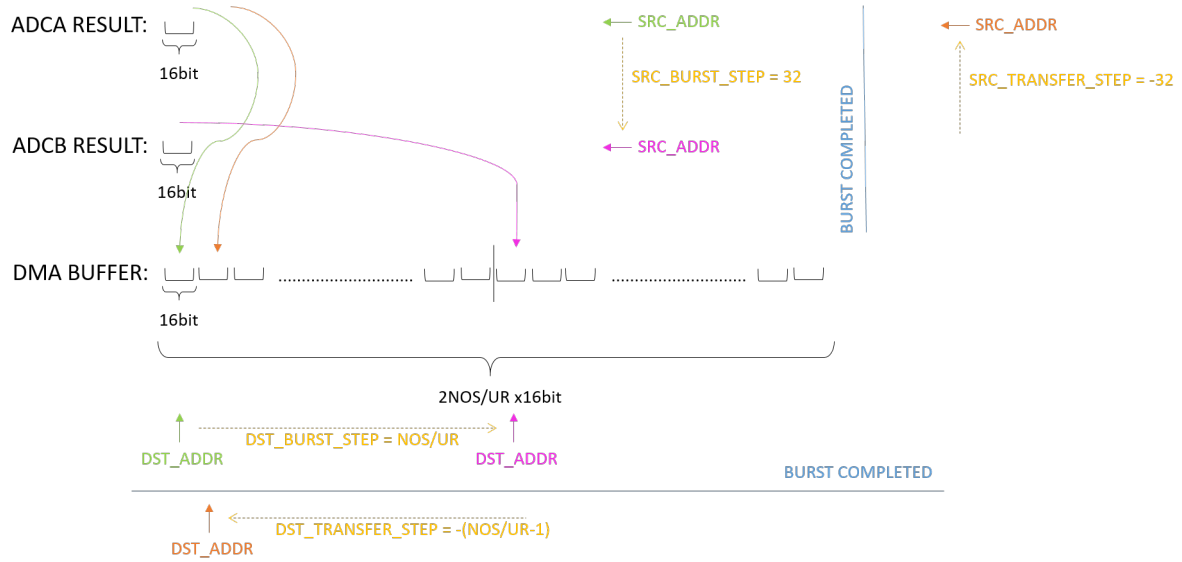

Figure 1: Configuration of DMA burst and transfer registers.

```
__interrupt void dmach1_isr(void)
{
    GpioDataRegs.GPCSET.bit.GPIO66 = 1;              // notify dmach1_isr start

    #if (OVERSAMPLING)
        int i_for = 0;
    #endif

    static int dma_sgn = 1;      // logic variable to indicate state of the ping pong algorithm

    Measurement_a = 0;
    Measurement_b = 0;
    dma_count++;

    // change dma_sgn to indicate state of the ping pong algorithm
    if (dma_sgn==1)
    {
        EALLOW;
        DmaRegs.CH1.DST_ADDR_SHADOW = (Uint32)&DMAbuffer2[0];   //PING DST ADRESA ZA SLEDECI
    PUT - ti trenutno pises u DMAbuffer1, a ja treba da citam iz proslog!!!
        EDIS;
        dma_sgn = -1;

        #if (OVERSAMPLING)
            // Collect data to be averaged
            for (i_for=0;i_for<NOS_UR;i_for++)
            {
                Measurement_a+=DMAbuffer2[i_for];
                Measurement_b+=DMAbuffer2[i_for+NOS_UR];
            }
        #else
            // Take last measurement
            Measurement_a = DMAbuffer2[NOS_UR-1];
            Measurement_b = DMAbuffer2[NOS_UR-1+NOS_UR];
        #endif
    }
```

```
    else if (dma_sgn==-1)
    {
        EALLOW;
        DmaRegs.CH1.DST_ADDR_SHADOW =  (Uint32)&DMAbuffer1[0];   //PONG
        EDIS;
        dma_sgn=1;

        #if (OVERSAMPLING)
            // Collect data to be averaged
            for (i_for=0;i_for<NOS_UR;i_for++)
            {
                Measurement_a+=DMAbuffer1[i_for];
                Measurement_b+=DMAbuffer1[i_for+NOS_UR];
            }
        #else
            // Take last measurement
            Measurement_a = DMAbuffer1[NOS_UR-1];
            Measurement_b = DMAbuffer1[NOS_UR-1+NOS_UR];
        #endif
    }

    #if (OVERSAMPLING)
        // Averaging on regulation period
        AvgMeas_a[0] = Measurement_a>>((int)LOG2_NOS_UR);
        AvgMeas_b[0] = Measurement_b>>((int)LOG2_NOS_UR);
        #if (UR==2)
            // Additional averaging if UR==2 to achieve given averaging on switching period
            Vmeas_a = (float32)((AvgMeas_a[0] + AvgMeas_a[1])>>1);
            Vmeas_b = (float32)((AvgMeas_b[0] + AvgMeas_b[1])>>1);
            // Save averaged measurements for next additional averaging
            AvgMeas_a[1] = AvgMeas_a[0];
            AvgMeas_b[1] = AvgMeas_b[0];
        #else
            Vmeas_a = (float32)(AvgMeas_a[0]);
            Vmeas_b = (float32)(AvgMeas_b[0]);
        #endif
    #else
        // Take last measurement stored in DMA buffer
        Vmeas_a = (float32)Measurement_a;
        Vmeas_b = (float32)Measurement_b;
    #endif

    // ADC scaling 3.0 --> 4095 (zero offset assumed)
    Vmeas_a = Vmeas_a*0.0007326f;
    Vmeas_b = Vmeas_b*0.0007326f;

    // Regulation...

    PieCtrlRegs.PIEACK.all = PIEACK_GROUP7;            // Clear acknowledge register

}
```