Self-Project

Digital Controller Implementation in Code Composer Studio (CCS) for Closed-Loop Control of a Buck Converter

Made by,
Vigneshwaran.R,
Power Engineering,
M.Tech, 2nd year,
224102114.

Objective:

To implement a digital PI controller in Digital Signal Processor (DSP) for a fixed output buck converter, using Code Composer Studio. Verify the working of the controller by connecting the controller to an existing buck converter and monitoring the output through Digital Oscilloscope.

Buck converter:

Buck converter consists of a switch, diode, inductor, capacitor and load resistor. Input voltage is higher than the desired output voltage. This converter is used to step down the input dc voltage.

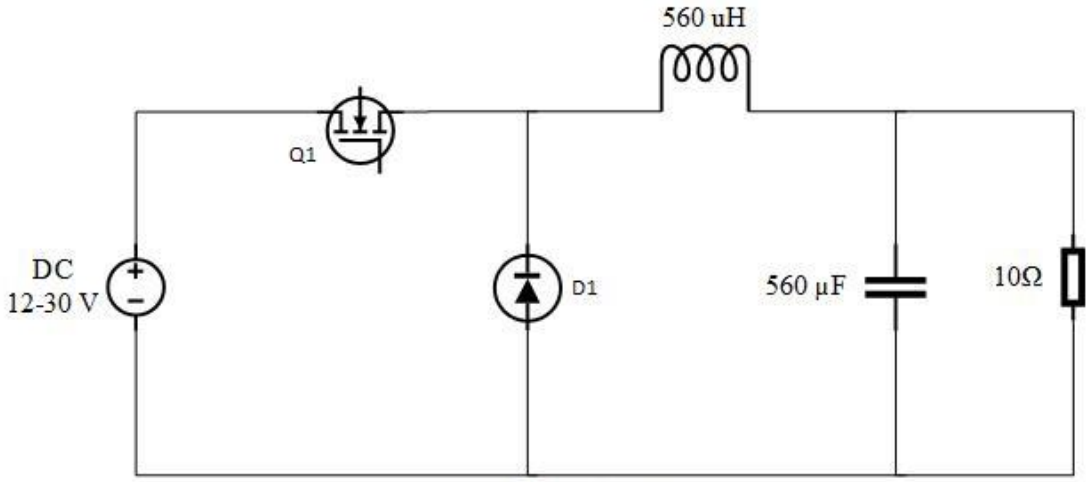


Figure 1- Buck converter Circuit diagram

a). When switch Q1 is on connecting input to the inductor, energy stored in its magnetic field. Diode is reverse biased and preventing current flow through it.

b). When switch Q1 is off, disconnecting input voltage from the inductor. The inductor releasing its energy, driving current through diode and supplying energy to the load.

For this project, existing buck converter in the lab is used.

Digital Control Design:

By trial-and-error method, K_p and K_i values of PI controller is determined in the MATLAB-Simulink. K_p and K_i values with less overshoot and settling time of less than one second is chosen.

For,

$$K_p = 0.05326 \text{ and } K_i = 0.5326$$

The transfer function of the comparator comes out to be,

$$G_C(s) = \frac{0.05326(s + 10)}{s}$$

Equ-1

This is a continuous time transfer function, to implement in **Digital Signal Processor (DSP),** we need a discrete time transfer function.

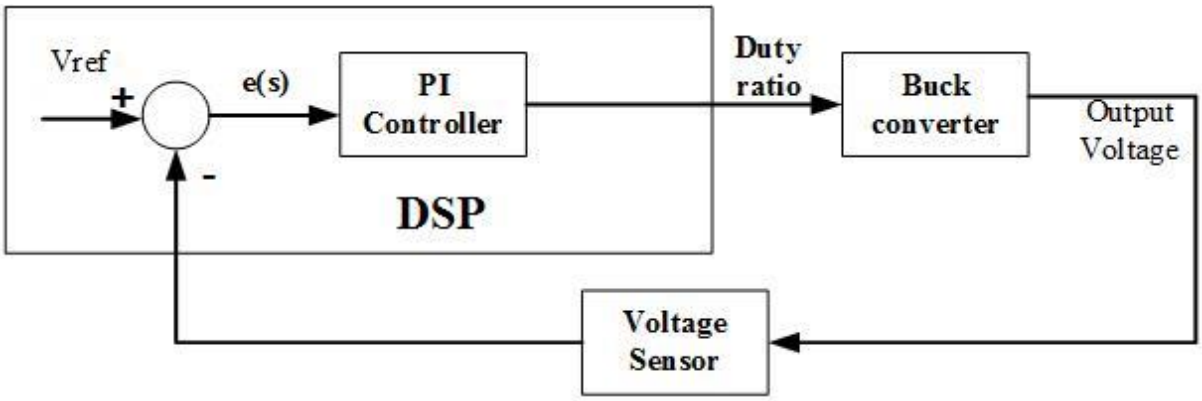


Figure 2- Block Diagram

Discretizing Transfer Function by Trapezoidal method:

To discretize, continuous time transfer function **Trapezoidal method** is used. The aim is to obtain an equivalent discrete-time transfer function in the z-domain. The trapezoidal method approximates the integral over a small-time interval using the average of the function values at the interval's endpoint.

We need to map the continuous s-plane to discrete z-plane, which can be done by substituting

$$\frac{1}{s} = \frac{T}{2} * \frac{(Z+1)}{(Z-1)}$$

Equ – 2

Where, T = Sampling period.

Controller transfer function can also be written as,

$$G_c(s) = K_p + \frac{K_i}{s}$$

Equ – 3

From the block diagram, it can be observed that,

$$\text{Duty ratio (D)} = G_c(s) * e(s)$$

Where,

$$e(s) = V_{ref} - V_o$$

Equ – 4

Therefore,

$$D = \left(K_p + K_i * \left(\frac{T}{2}\left(\frac{(Z+1)}{(Z-1)}\right)\right)\right) * e(s)$$

Equ – 5

By solving, equ 5, we get

$$D - DZ^{-1} = \left(K_p + K_i * T\right) + \left(-K_p + K_i * T\right)Z^{-1}$$

Equ – 6

And by taking, Z^{-1} Transform, we get

$$D[n] = D[n-1] + (K_p + K_i * T) * e[n] + \left(-K_p + K_i * T\right) * e[n-1]$$

Equ – 7

Where,

D[n] = Duty ratio calculated of present.

D[n – 1] = Duty ratio calculated for previous sample.

e[n] = Error between V_{ref} and V_{out} in present sample.

e[n-1] = Error between V_{ref} and V_{out} in Previous sample.

T = Sampling time (Here, it is same as switching frequency).

This logic is implemented in the DSP code, in the Interrupt Service Routine (ISR) created by the ePWM while it is reaching end of the time period in the counter of ePWM. So, sampling and switching frequency is same. Here, snippet of code for reference.

```
interrupt void ePWM2A_compare_isr(void)
// ISR is triggered by ePWM2 PRD event
{
    //static unsigned int index=0;
    // Service watchdog every interrupt
    Voltage_VR1 = AdcMirror.ADCRESULT0; // store results global
    EALLOW;
    SysCtrlRegs.WDKEY = 0xAA;       // Service watchdog #2
    EDIS;

    Vol= (Voltage_VR1*3/4095)*10;
    V[1]=V[0];
    V[0]=Vol;
    D[1]=D[0];

    D[0]=D[1]+((kp+(ki*0.00025))*(VolRef-V[0]))+((-kp+(ki*0.00025))*(VolRef-V[1]));

    if (D[0]>0.9)
    { D[0]=0.9;}

    else if (D[0]<.1)
    {
        D[0]=.1;
    }
    var=EPwm2Regs.TBPRD*(1-D[0]);
    EPwm2Regs.CMPA.half.CMPA=EPwm2Regs.TBPRD*(1-D[0]);
    EPwm2Regs.ETCLR.bit.INT = 1;        // Clear ePWM1 Interrupt flag
    // Acknowledge this interrupt to receive more interrupts from group 3
    PieCtrlRegs.PIEACK.all = 4;
}
```

Figure 3 – PI Controller implementation in code

Code Composer Studio (CCS) and DSP:

CCS is an Integrated Development Environment (IDE) for TI's microcontroller and processors. The controller used for this project is Texas Instrument's Delfino – f28335, which is suitable for control of power electronic converters. It has 8 separate PWM's and 16 input pins dedicated for ADC inputs.

ePWM generation:

By programming the ePWM registers of DSP, PWM can be generated as per out requirement.

By setting the value of TBPRD, Frequency of the PWM can be obtained.

By setting the value of CMPA or CMPB, duty ratio can be decided.

Epwm2Regs.CMPA.half.CMPA = Epwm2Regs.TBPRD*(1-D[0])

CMPA is assigned with the value based on duty ratio calculated in that particular sample period. It is programmed in such a way that output of PWM is set to '1' when counter reaches value of CMPA and reset to '0' when it reaches TBPRD.

Analog to Digital Conversion (ADC):

ADC Start of Conversion (SOC) can be triggered by the event in the ePWM. Here, when ePWM counter reaches TBPRD value, ADC Start of Conversion (SOC) is triggered. Such that the CMPA value is updated only once during single switching cycle.

Voltage_VR1 = AdcMirror.ADCRESULT0

By this way value measured in ADC is stored in a global variable, Voltage_VR1.

Voltage Sensor:

Voltage given to the input of the ADC pin should not exceed 3 V. But buck output voltage exceeds this value. So, the voltage of the buck is attenuated 10:1 ratio before connecting to the DSP. Which is achieved by differential amplifier and buffer circuit using op-amp.

After sensing through the voltage sensor, the value is scaled to the actual value and compared with the reference voltage to produce the error signal, which is used to compute the duty ratio.

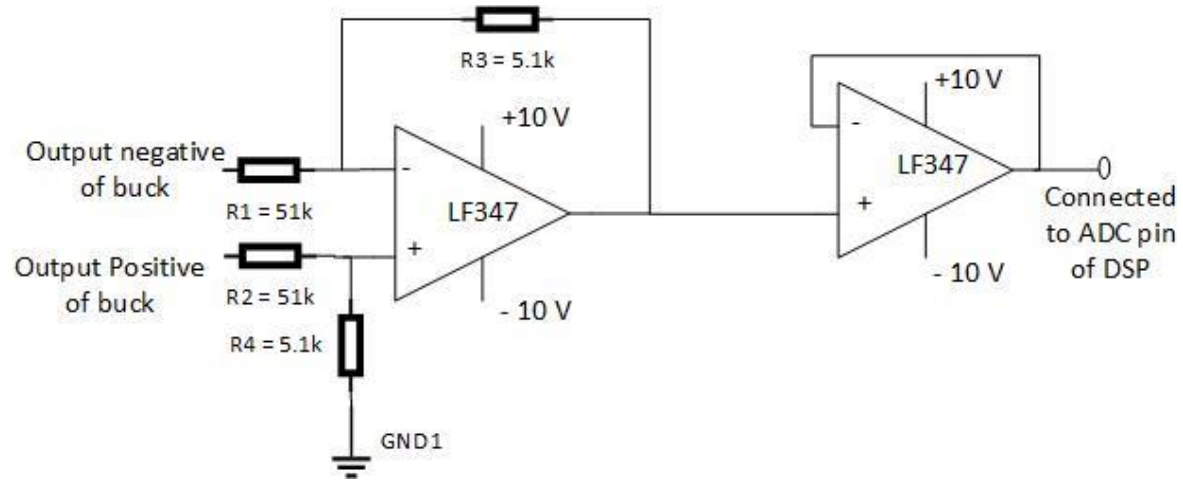


Figure 4 – Voltage Sensor

Experimental Setup:

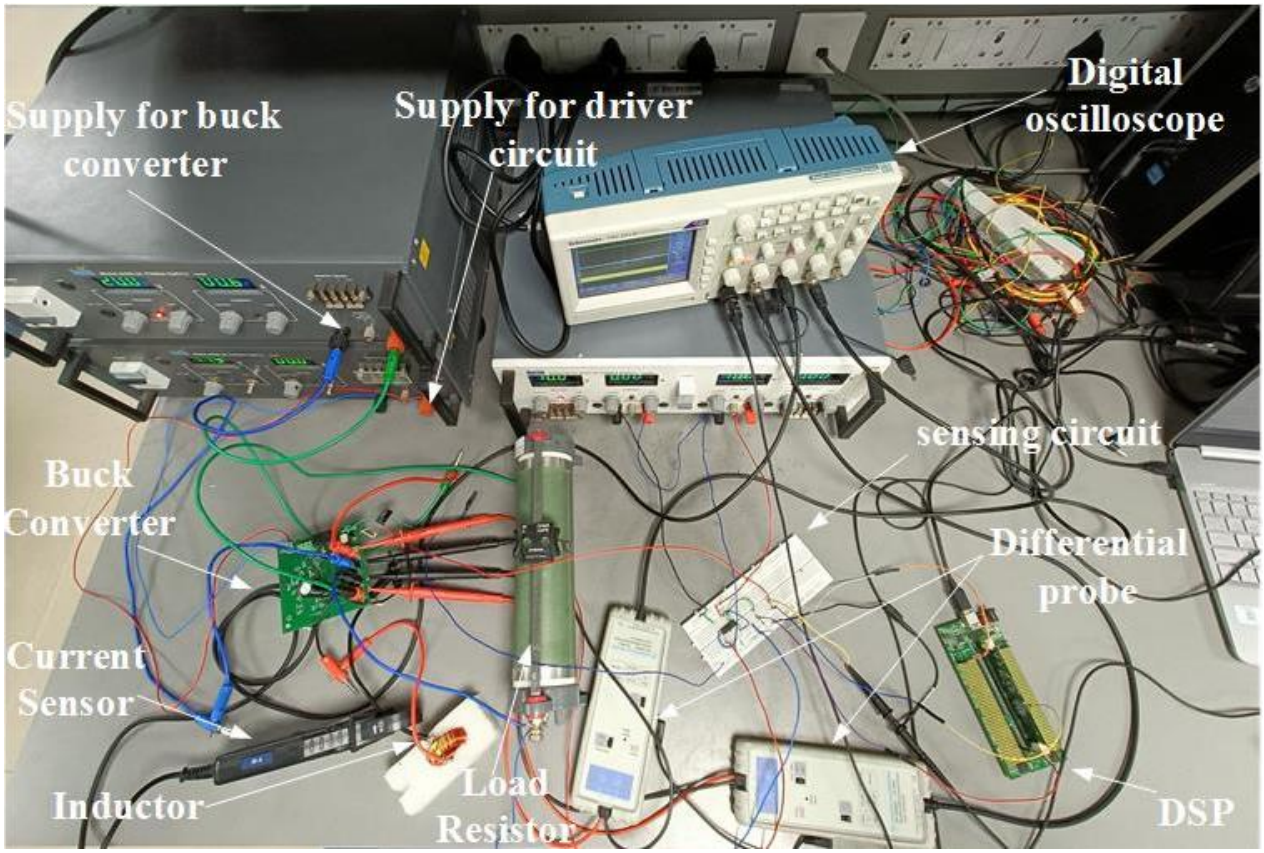


Figure 5 - Experimental Setup

Output and input voltage of the buck converter is connected to oscilloscope through differential probe to the oscilloscope. Current sensor is used to measure current through the inductor. Dual supply is given to the voltage sensor circuit. And separate power supply is given to the gate driver of the buck converter.

After connecting the circuit, input voltage of the buck converter is varied from 12 V to 30 V and output voltage of the buck converter is observed through oscilloscope.

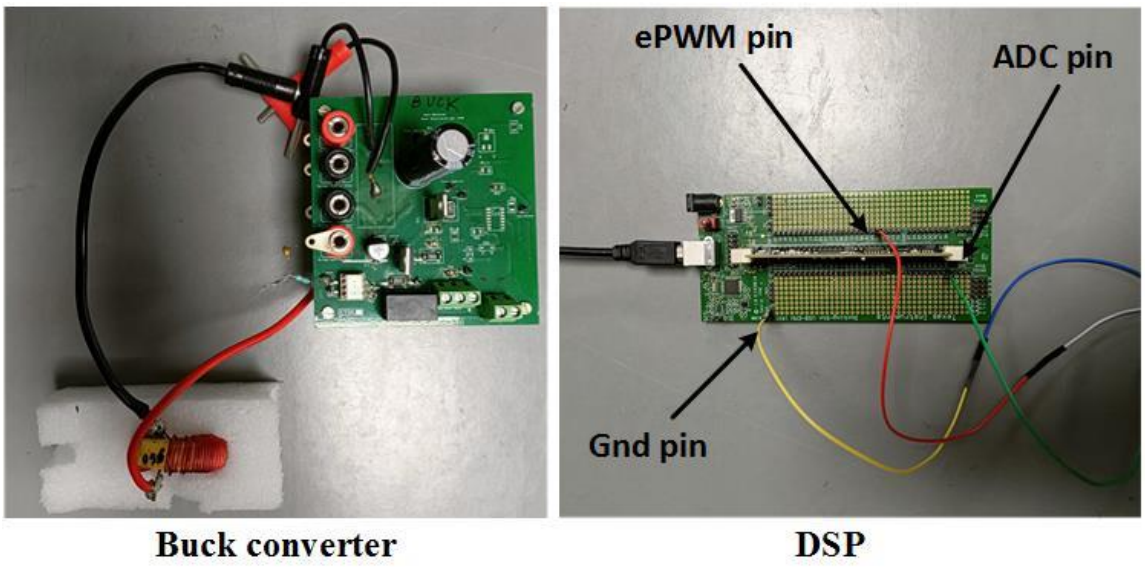
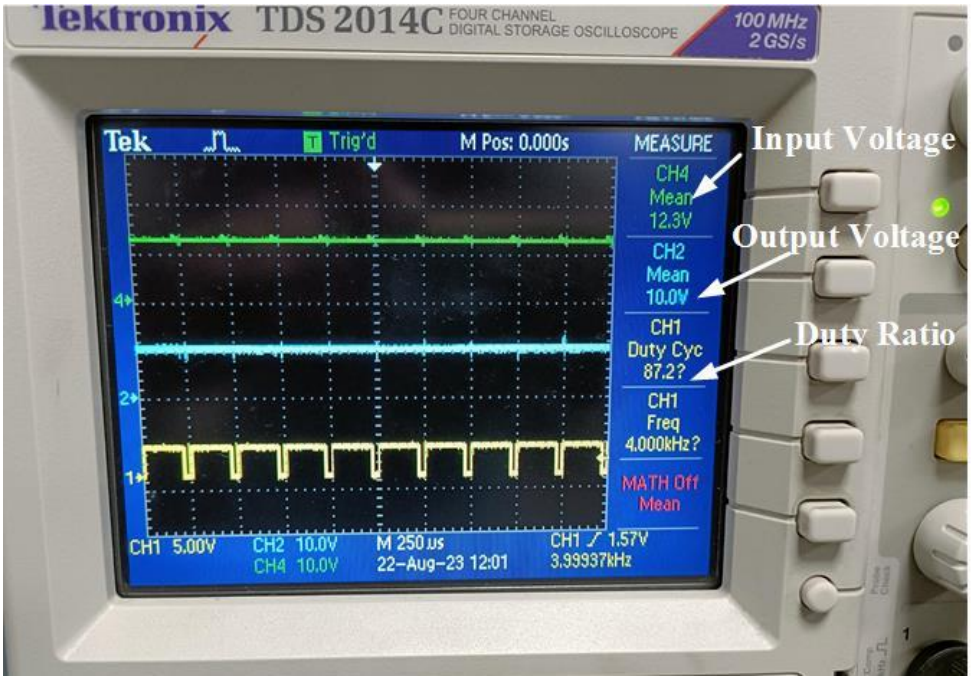


Figure 6 - Buck converter and DSP

Experimental Results:

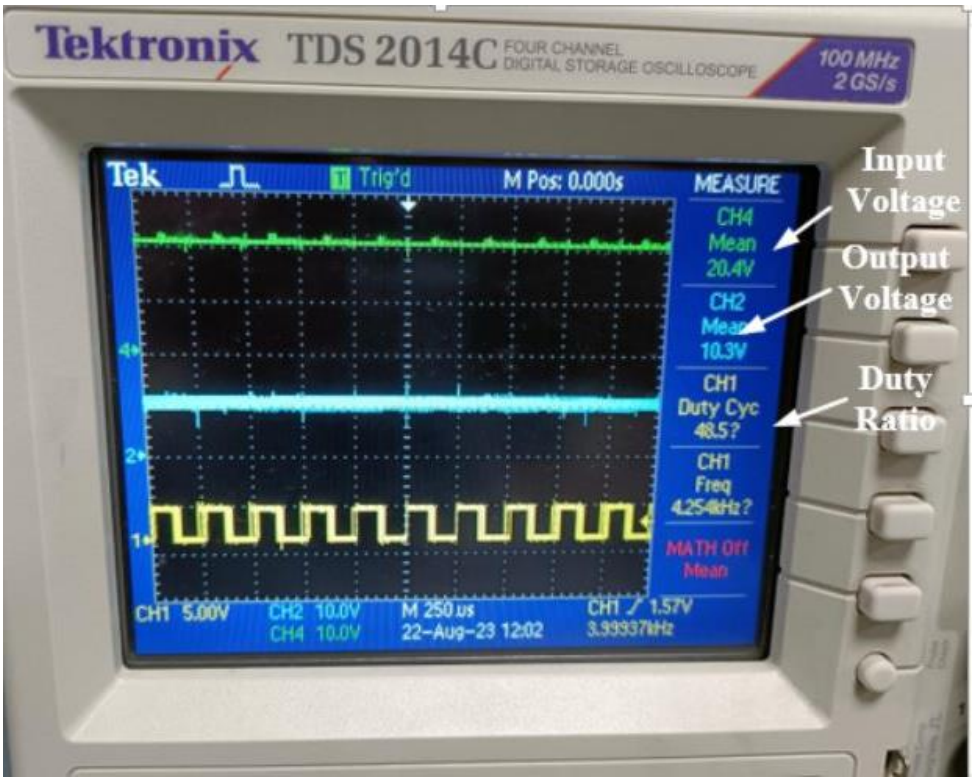
Here, reference voltage is given as 10 V to the DSP, and input voltage of the Buck Converter is varied form 12 V to 30 V. And output voltage of the buck converter is monitored through oscilloscope.

1. **For input voltage of Buck Converter = 12 V**.
   Output voltage is maintained at 10 V by the PI controller of the DSP, with a **Duty cycle of 87.2%.**



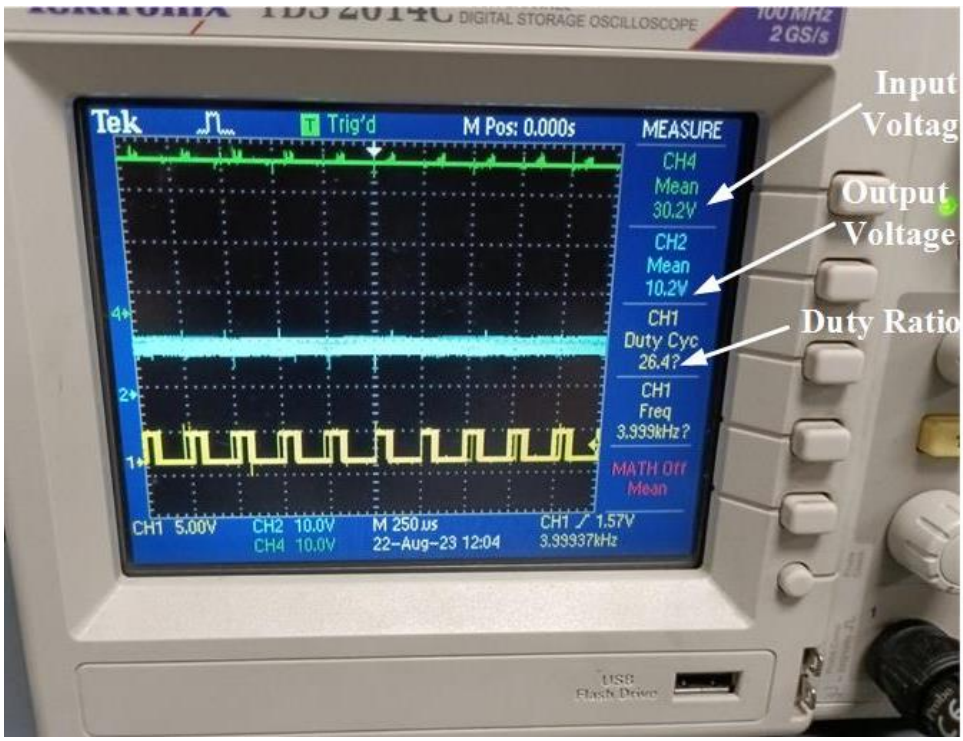
2. **For input voltage of Buck converter = 20 V.**

Output voltage is maintained at 10 V by the PI controller of the DSP, with a **Duty cycle of 87.2%.**



3. **For input voltage of Buck converter = 30 V.**

Output voltage is maintained at 10 V by the PI controller of the DSP, with a **Duty cycle of 26.4%.**



Conclusion:

This digital PI controller implemented through DSP – Delfino f28335 is successfully able to control the output voltage of buck converter for variable input voltage of the buck converter. When the input voltage is decreased the duty ratio is automatically increased by the PI-Controller and output voltage is maintained constant. Similarly, when output voltage is increased the duty ratio of the buck converter is automatically decreased by the PI-Controller. Therefore, output voltage of the buck converter is regulated.

Appendix: C-Code for DSP:

// **Start of Source code**

```
#include "DSP2833x_Device.h"

// external function prototypes
extern void InitAdc(void);
extern void InitSysCtrl(void);
extern void InitPieCtrl(void);
extern void InitPieVectTable(void);
extern void InitCpuTimers(void);
extern void ConfigCpuTimer(struct CPUTIMER_VARS *, float, float);

// Prototype statements for functions found within this file.
void Gpio_select(void);
interrupt void cpu_timer0_isr(void);
interrupt void ePWM2A_compare_isr(void);

// Global Variables
int VolRef=10;
float Voltage_VR1;
int counter=0;
float Vol;
float V[2]={0,0};
float D[2]={0,0};
float kp=0.05326,ki=0.5326;
float var;
//###########################################################################
//                main code
//###########################################################################
void main(void)
{
```

```
InitSysCtrl();  // Basic Core Init from DSP2833x_SysCtrl.c

EALLOW;
SysCtrlRegs.WDCR= 0x00AF;   // Re-enable the watchdog
EDIS;         // 0x00AF  to NOT disable the Watchdog, Prescaler = 64

DINT;             // Disable all interrupts

Gpio_select();      // To initialize epwm and ADC pins

InitPieCtrl();      // basic setup of PIE table; from DSP2833x_PieCtrl.c

InitPieVectTable(); // default ISR's in PIE

InitAdc();         // Basic ADC setup, incl. calibration

AdcRegs.ADCTRL1.all = 0;
AdcRegs.ADCTRL1.bit.ACQ_PS = 7;     // 7 = 8 x ADCCLK
AdcRegs.ADCTRL1.bit.SEQ_CASC =1;    // 1=cascaded sequencer
AdcRegs.ADCTRL1.bit.CPS = 0;       // divide by 1
AdcRegs.ADCTRL1.bit.CONT_RUN = 0;   // single run mode

AdcRegs.ADCTRL2.all = 0;
AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = 1;   // 1=enable SEQ1 interrupt
AdcRegs.ADCTRL2.bit.EPWM_SOCA_SEQ1 =1;  // 1=SEQ1 start from ePWM_SOCA trigger
AdcRegs.ADCTRL2.bit.INT_MOD_SEQ1 = 0;   // 0= interrupt after every end of sequence

AdcRegs.ADCTRL3.bit.ADCCLKPS = 3;   // ADC clock: FCLK = HSPCLK / 2 * ADCCLKPS
                        // HSPCLK = 75MHz (see DSP2833x_SysCtrl.c)
                        // FCLK = 12.5 MHz
```

```
AdcRegs.ADCMAXCONV.all = 0x0001;    // 2 conversions from Sequencer 1

AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0; // Setup ADCINA0 as 1st SEQ1 conv.
AdcRegs.ADCCHSELSEQ1.bit.CONV01 = 1; // Setup ADCINA1 as 2nd SEQ1 conv.

EPwm2Regs.TBCTL.all = 0xC030;   // Configure timer control register
EPwm1Regs.TBCTL.bit.HSPCLKDIV = 1;  // HSPCLKDIV = 4
/*
 bit 15-14     11:     FREE/SOFT, 11 = ignore emulation suspend
 bit 13        0:      PHSDIR, 0 = count down after sync event
 bit 12-10     000:    CLKDIV, 000 => TBCLK = HSPCLK/1
 bit 9-7       000:    HSPCLKDIV, 000 => HSPCLK = SYSCLKOUT/1
 bit 6         0:      SWFSYNC, 0 = no software sync produced
 bit 5-4       11:     SYNCOSEL, 11 = sync-out disabled
 bit 3         0:      PRDLD, 0 = reload PRD on counter=0
 bit 2         0:      PHSEN, 0 = phase control disabled
 bit 1-0       00:     CTRMODE, 00 = count up mode
*/

EPwm2Regs.AQCTLA.all = 0x0024;      // set ePWM2A on CMPA up
                            // clear ePWM2A on TBPRD
EPwm2Regs.TBPRD = 37500; // TPPRD +1  =  TPWM / (HSPCLKDIV * CLKDIV * TSYSCLK)
                  //          =  20  s / 6.667 ns

EPwm2Regs.CMPA.half.CMPA = EPwm2Regs.TBPRD / 2;
EPwm2Regs.ETPS.all = 0x0100;           // Configure ADC start by ePWM2

/*
 bit 15-14     00:     EPWMxSOCB, read-only
 bit 13-12     00:     SOCBPRD, don't care
 bit 11-10     00:     EPWMxSOCA, read-only
```

```
 bit 9-8      01:    SOCAPRD, 01 = generate SOCA on first event
 bit 7-4      0000:  reserved
 bit 3-2      00:    INTCNT, don't care
 bit 1-0      00:    INTPRD, don't care
*/

EPwm2Regs.ETSEL.all = 0x0A0A;          // Enable SOCA to ADC and interrupt enable for
                                       // for epwm interrupt on prd match
// EPwm2Regs.ETSEL.bit.INTEN = 1;      // interrupt enable for ePWM2
// EPwm2Regs.ETSEL.bit.INTSEL = 4;     // interrupt on CMPA UP match
 EPwm2Regs.ETPS.bit.INTPRD = 1;      // interrupt on first event
 /*
 bit 15       0:     SOCBEN, 0 = disable SOCB
 bit 14-12    000:   SOCBSEL, don't care
 bit 11       1:     SOCAEN, 1 = enable SOCA
 bit 10-8     010:   SOCASEL, 010 = SOCA on PRD event
 bit 7-4      0000:  reserved
 bit 3        0:     INTEN, 0 = disable interrupt
 bit 2-0      000:   INTSEL, don't care
*/

EALLOW;
PieVectTable.TINT0 = &cpu_timer0_isr;
PieVectTable.EPWM2_INT = &ePWM2A_compare_isr;
EDIS;

InitCpuTimers();   // basic setup CPU Timer0, 1 and 2

ConfigCpuTimer(&CpuTimer0,150,100000);

PieCtrlRegs.PIEIER1.bit.INTx7 = 1;     // CPU Timer 0
```

```
   //PieCtrlRegs.PIEIER1.bit.INTx6 = 1;     // ADC
   IER |=1;
   // Enable EPWM2A INT in the PIE: Group 3 interrupt 1
      PieCtrlRegs.PIEIER3.bit.INTx2 = 1;
      IER |=5;          // enable INT4 for ePWM2
   EINT;
   ERTM;
   CpuTimer0Regs.TCR.bit.TSS = 0;  // start timer0
   while(1)
      {
         while(CpuTimer0.InterruptCount == 0)
         {
            EALLOW;
            SysCtrlRegs.WDKEY = 0x55;   // service WD #1
            SysCtrlRegs.WDKEY = 0xAA;   // service WD #2
            EDIS;
         }
         CpuTimer0.InterruptCount = 0;
      }
   }
void Gpio_select(void)
{
   EALLOW;
   GpioCtrlRegs.GPAMUX1.all = 0;      // GPIO15 ... GPIO0 = General Puropse I/O
   GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 1; // ePWM2A active
   GpioCtrlRegs.GPAMUX2.all = 0;      // GPIO31 ... GPIO16 = General Purpose I/O
   GpioCtrlRegs.GPBMUX1.all = 0;      // GPIO47 ... GPIO32 = General Purpose I/O
   GpioCtrlRegs.GPBMUX2.all = 0;      // GPIO63 ... GPIO48 = General Purpose I/O
   GpioCtrlRegs.GPCMUX1.all = 0;      // GPIO79 ... GPIO64 = General Purpose I/O
   GpioCtrlRegs.GPCMUX2.all = 0;      // GPIO87 ... GPIO80 = General Purpose I/O
```

```
   GpioCtrlRegs.GPADIR.all = 0;
   GpioCtrlRegs.GPADIR.bit.GPIO9 = 1;  // peripheral explorer: LED LD1 at GPIO9
   GpioCtrlRegs.GPADIR.bit.GPIO11 = 1; // peripheral explorer: LED LD2 at GPIO11

   GpioCtrlRegs.GPBDIR.all = 0;        // GPIO63-32 as inputs
   GpioCtrlRegs.GPBDIR.bit.GPIO34 = 1; // peripheral explorer: LED LD3 at GPIO34
   GpioCtrlRegs.GPBDIR.bit.GPIO49 = 1; // peripheral explorer: LED LD4 at GPIO49

   GpioCtrlRegs.GPCDIR.all = 0;        // GPIO87-64 as inputs
   EDIS;
}
interrupt void cpu_timer0_isr(void)
{
   CpuTimer0.InterruptCount++;
   EALLOW;
   SysCtrlRegs.WDKEY = 0xAA;   // service WD #2
   EDIS;
   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}
interrupt void ePWM2A_compare_isr(void)
// ISR is triggered by ePWM2 PRD event
{
   //static unsigned int index=0;
   // Service watchdog every interrupt
   Voltage_VR1 = AdcMirror.ADCRESULT0; // store results global
   EALLOW;
   SysCtrlRegs.WDKEY = 0xAA;       // Service watchdog #2
   EDIS;
   Vol= (Voltage_VR1*3/4095)*10;
   V[1]=V[0];
   V[0]=Vol;
```

```
   D[1]=D[0];
   D[0]=D[1]+((kp+(ki*0.00025))*(VolRef-V[0]))+((-kp+(ki*0.00025))*(VolRef-V[1]));
   if (D[0]>0.9)
   { D[0]=0.9;}
   else if (D[0]<.1)
   {
      D[0]=.1;
   }
   var=EPwm2Regs.TBPRD*(1-D[0]);
   EPwm2Regs.CMPA.half.CMPA=EPwm2Regs.TBPRD*(1-D[0]);
         EPwm2Regs.ETCLR.bit.INT = 1;       // Clear ePWM1 Interrupt flag
   // Acknowledge this interrupt to receive more interrupts from group 3
   PieCtrlRegs.PIEACK.all = 4;
}
// End of SourceCode.
```