Application Report
SPRABT3A–July 2013–Revised July 2017

# Software Phase Locked Loop Design Using C2000™ Microcontrollers for Single Phase Grid Connected Inverter

*Manish Bhardwaj*

**ABSTRACT**

Grid connected applications require an accurate estimate of the grid angle to feed power synchronously to the grid. This is achieved using a software phase locked loop (PLL). This application report discusses different challenges in the design of software phase locked loops and presents a methodology to design phase locked loops using C2000 controllers for single phase grid connection applications.

**Contents**



**List of Figures**



## Trademarks



## 1 Introduction

The phase angle of the utility is a critical piece of information for the operation of power devices feeding power into the grid like PV inverters. A phase locked loop is a closed loop system in which an internal oscillator is controlled to keep the time and phase of an external periodical signal using a feedback loop. The PLL is simply a servo system that controls the phase of its output signal such that the phase error between the output phase and the reference phase is minimum. The quality of the lock directly affects the performance of the control loop in grid tied applications. As line notching, voltage unbalance, line dips, phase loss and frequency variations are common conditions faced by equipment interfacing with electric utility, the PLL needs to be able to reject these sources of error and maintain a clean phase lock to the grid voltage.

A functional diagram of a PLL is shown in Figure 1, which consists of a phase detect (PD), a loop filter (LPF), and a voltage controlled oscillator (VCO).

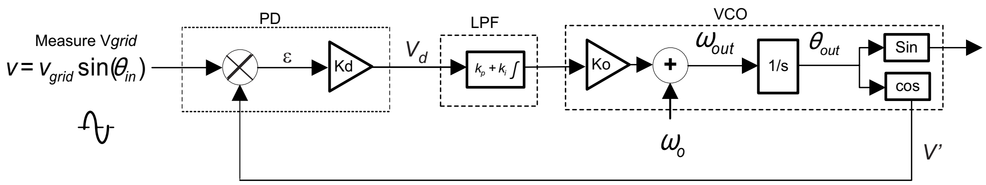



**Figure 1. Phase Locked Loop Basic Structure**

The measured grid voltage can be written in terms the grid frequency ($w_{grid}$) as follows:

$$v = v_{grid}\sin(\theta_{in}) = v_{grid}\sin(w_{grid}t + \theta_{grid}) \tag{1}$$

Now, assuming VCO is generating sine waves close to the grid sinusoid, VCO output can be written as,

$$v' = \cos(\theta_{out}) = \cos(w_{PLL}t + \theta_{PLL}) \tag{2}$$

The purpose of the phase detect block is to compare the input sine with the locked sine from the VCO and to generate an error signal proportional to the angle error. For this, the phase detect block multiplies the VCO output and the measured input value to get:

$$v_d = \frac{K_d v_{grid}}{2}[\sin((w_{grid} - w_{PLL})t + (\theta_{grid} - \theta_{PLL})) + \sin((w_{grid} + w_{PLL})t + (\theta_{grid} + \theta_{PLL}))] \tag{3}$$

From Equation 3, it is clear that the output of PD block has information of the locking error. However, the locking error information available from the PD is not linear, and has a component which is varying at twice the grid frequency. To use this locking error information to lock the PLL angle, twice the grid frequency component must be removed.

For now, ignoring the twice of grid frequency component, the lock error is given as:

$$\overline{v_d} = \frac{K_d v_{grid}}{2}\sin((w_{grid} - w_{PLL})t + (\theta_{grid} - \theta_{PLL})) \tag{4}$$

For steady state operation, the $w_{grid}$ - $w_{PLL}$ term can be ingored, for small values of theta $sin(\theta) \sim \theta$. Hence, a linearized error is given as:

$$err = \frac{v_{grid}(\theta_{grid} - \theta_{PLL})}{2} \tag{5}$$

This error is the input to loop filter, which is nothing but a PI controller, that is used to reduce the locking error at steady state to zero. The small signal analysis is done using network theory, where the feedback loop is broken to get the open loop transfer equation and then the closed-loop transfer function:

Closed Loop TF = Open Loop TF / (1+ OpenLoopTF)

Thus, for the linearized feedback the PLL transfer function can be written as follows:

Closed loop Phase TF:

$$H_o(s) = \frac{\theta_{out}(s)}{\theta_{in}(s)} = \frac{LF(s)}{s + LF(s)} = \frac{v_{grid}(k_p s + \frac{k_p}{T_i})}{s^2 + v_{grid}k_p s + v_{grid}\frac{k_p}{T_i}}$$

Closed loop error transfer function:

$$E_o(s) = \frac{V_d(s)}{\theta_{in}(s)} = 1 - H_o(s) = \frac{s}{s + LF(s)} = \frac{s^2}{s^2 + k_p s + \frac{k_p}{T_i}}$$

Comparing the closed loop phase transfer function to a generic second order system transfer function, which is given as:

$$H(s) = \frac{2\zeta\omega_n s + \omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \quad (6)$$

The natural frequency and the damping ration of the linearized PLL are given as:

$$\omega_n = \sqrt{\frac{v_{grid} K_p}{T_i}} \quad (7)$$

(8)

$$\zeta = \sqrt{\frac{v_{grid} T_i K_p}{4}} \quad (9)$$

Note in the PLL, the PI serves dual purpose:

- To filter out high frequency that is at twice the frequency of the carrier and grid
- Control response of the PLL to step changes in the grid conditions, for example, phase leaps, magnitude swells, and so forth.

As the loop filter has low-pass filter characteristic, it can be used to filter out the high frequency component that was ignored earlier. If the carrier frequency/ frequency of the signal being locked is high, the low-pass characteristics of the PI are good enough to cancel the twice of carrier frequency component. However, for grid connected applications as the grid frequency is very low (50Hz-60Hz), the roll off provided by the PI is not satisfactory enough and introduces a high frequency element into the loop filter output, which affects the performance of the PLL.

From the discussion above, it is clear that the LPF characteristic of the PI controller cannot be used to eliminate the twice to grid frequency component from the phase detect output in case of grid connected applications. Hence, alternative methods must be used that linearize the PD block. In this application report, two PLL methods that linearize the PD output, are illustrated:

- One uses a notch filter to filter out twice the grid frequency component from the PD output
- The other uses an orthogonal signal generation method to use stationary reference frame PLL technique in single phase PLL

## 2 PLL With Notch Filter

A notch filter can be used at the output of the phase detect block, which attenuates twice the grid frequency component very well. An adaptive notch filter can also be used to selectively notch the exact frequency in case there are variations in the grid frequency. Section 2.1 illustrates the selection procedure of the PI coefficients, their digital implementation and mapping. The design of the adaptive notch filter is illustrated and a method to calculate the coefficients automatically, and on line is illustrated with the embedded code implementation.

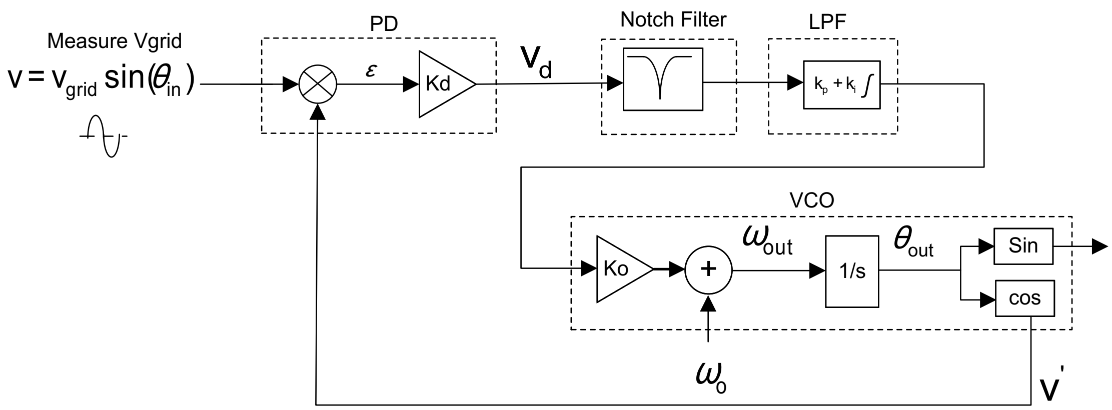



**Figure 2. Single Phase PLL With Notch Filter**

As discussed in Section 1, with the addition of the notch filter, the PI tuning can be done solely based on dynamic response of the PLL. Section 2.1 illustrates digital implementation of the PI controller and the selection of the coefficients for the PI controller to be used.

## 2.1 Discrete Implementation of PI Controller

The loop filter or the PI is implemented as a digital controller with Equation 10:

$$ylf[n] = ylf[n-1]*A1 + ynotch[n]*B0 + ynotch[n-1]*B1 \quad (10)$$

Using z transform, Equation 10 can be re-written as:

$$\frac{ylf(z)}{ynotch(z)} = \frac{B0 + B1*z^{-1}}{1-z^{-1}} \quad (11)$$

It is well known the PI controller in laplace transform is given by:

$$\frac{ylf(s)}{ynotch(s)} = K_p + \frac{K_i}{s} \quad (12)$$

Using bi-linear transformation, replace $s = \frac{2}{T}\left(\frac{z-1}{z+1}\right)$, where T = Sampling Time.

$$\frac{ylf(z)}{ynotch(z)} = \frac{\left(\frac{2*K_p + K_i*T}{2}\right) - \left(\frac{2*K_p - K_i*T}{2}\right)z^{-1}}{1-z^{-1}} \quad (13)$$

Equation 11 and Equation 13 can be compared to map the proportional and integral gain of the PI controller into the digital domain. The next challenge is selecting an appropriate value for the proportional and integral gain.

The step response to a general second order equation:

$$H(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \quad (14)$$

is given as:

$$y(t) = 1 - ce^{-\sigma t}\sin(\omega_d t + \varphi) \quad (15)$$

ignoring the LHP zero from Equation 15. The settling time is given as the time it takes for the response to settle between an error band, say this error is $\partial$, then:

$$1-\partial = 1 - ce^{-\sigma t_s} \Rightarrow \partial = ce^{-\sigma t_s} \Rightarrow t_s = \frac{1}{\sigma}*\ln\left(\frac{c}{\sigma}\right) \quad (16)$$

$$\text{Where, } \sigma = \varsigma\omega_n \text{ and } c = \frac{\omega_n}{\omega_d} \text{ and } \omega_d = \sqrt{1-\varsigma^2}\,\omega_n \quad (17)$$

Using settling time as 30 ms, and the error band as 5% and damping ratio to be 0.7, the natural frequency is obtained to be 158.6859. Back substituting $K_p$ =222.1603 and $K_i$ = 25181.22.

Back substituting these values into the digital loop filter coefficients:

$$B0 = \left(\frac{2*K_p + K_i*T}{2}\right) \text{ and } B1 = -\left(\frac{2*K_p - K_i*T}{2}\right) \quad (18)$$

For 50 Khz run rate of the PLL, B0 = 223.4194 and B1 = -220.901.

## 2.2 Adaptive Notch Filter Design

The notch filter used in the PLL shown in Figure 2 needs to attenuate twice the grid frequency component. Grid frequency, though stable, can have some variation, and with increasing renewable content larger variation are possible. Therefore, to precisely notch twice the grid frequency, an adaptive notch filter is used. A typical notch filter equation is 's' domain as shown in Equation 19:

$$H_{nf}(s) = \frac{s^2 + 2\zeta_2\omega_n s + \omega_n^2}{s^2 + 2\zeta_1\omega_n s + \omega_n^2} \text{ where } \zeta_2 \ll \zeta_1 \text{ for notch action to occur} \quad (19)$$

Discretizing Equation 19 using zero order hold, $s = \frac{(z-1)}{T}$, the equation is reduced to:

$$H_{nf}(z) = \frac{z^2 + (2\zeta_2\omega_n T - 2)z + (-2\zeta_2\omega_n T + \omega_n^2 T^2 + 1)}{z^2 + (2\zeta_1\omega_n T - 2)z + (-2\zeta_1\omega_n T + \omega_n^2 T^2 + 1)} = \frac{B_0 + B_1 z^{-1} + B_2 z^{-2}}{A_0 + A_1 z^{-1} + A_2 z^{-2}} \quad (20)$$

Equation 20 maps well into a digital two-pose two-zero structure and the coefficients for the notch filter can be adaptively changed as the grid frequency varies by calling a routine in the background that estimates the coefficients based on measure grid frequency.

For example, taking $\zeta_2$ = 0.00001 and $\zeta_1$ = 0.1 ($\zeta_2 << \zeta_1$), the response of the notch is as shown in Figure 3 for 50Hz and 60Hz grid, where the coefficients are updated based on grid frequency.

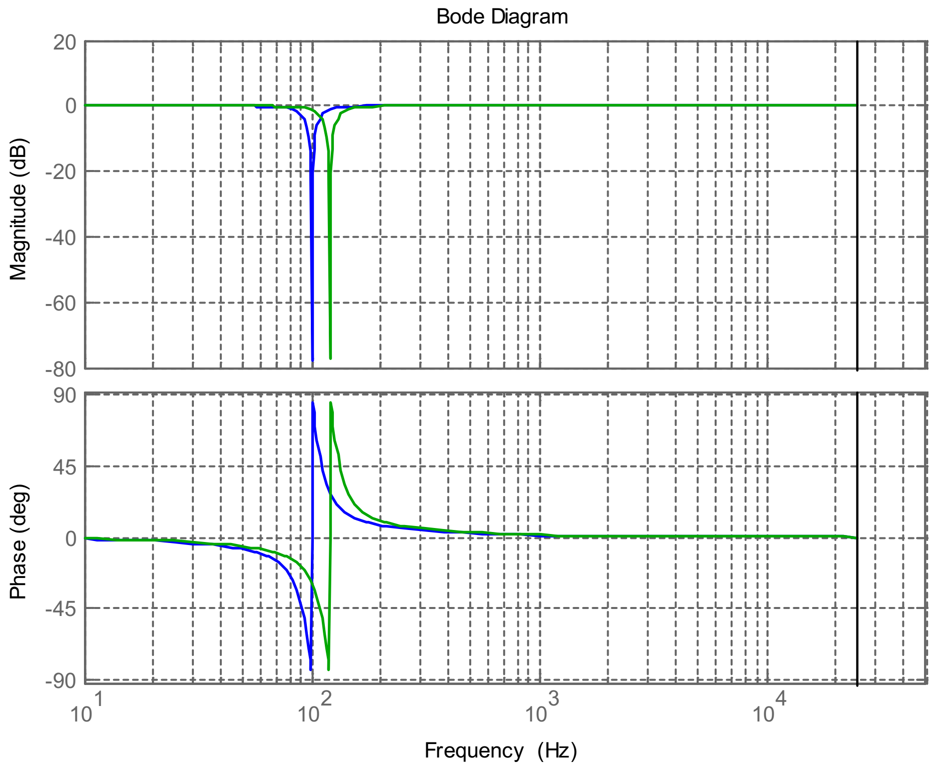



**Figure 3. Bode Diagram**

## 2.3 Sine and Cosine Generation

The PLL uses sin and cos calculation, these calculations can consume large number of cycles in a typical microcontroller. To avoid this issue, the sine and cosine value is generated in this module by applying the principle of integration.

$$y(t + \Delta t) = y(t) + \frac{dy(t)}{dt} * \Delta t \quad (21)$$

For sine and cosine signal, this reduces to:

$$\sin(t + \Delta t) = \sin(t) + \frac{d\sin(t)}{dt} * \Delta t = \sin(t) + \cos(t) * \Delta t$$

$$\cos(t + \Delta t) = \cos(t) + \frac{d\cos(t)}{dt} * \Delta t = \cos(t) - \sin(t) * \Delta t \quad (22)$$

## 2.4 Simulating the Phase Locked Loop for Varying Conditions

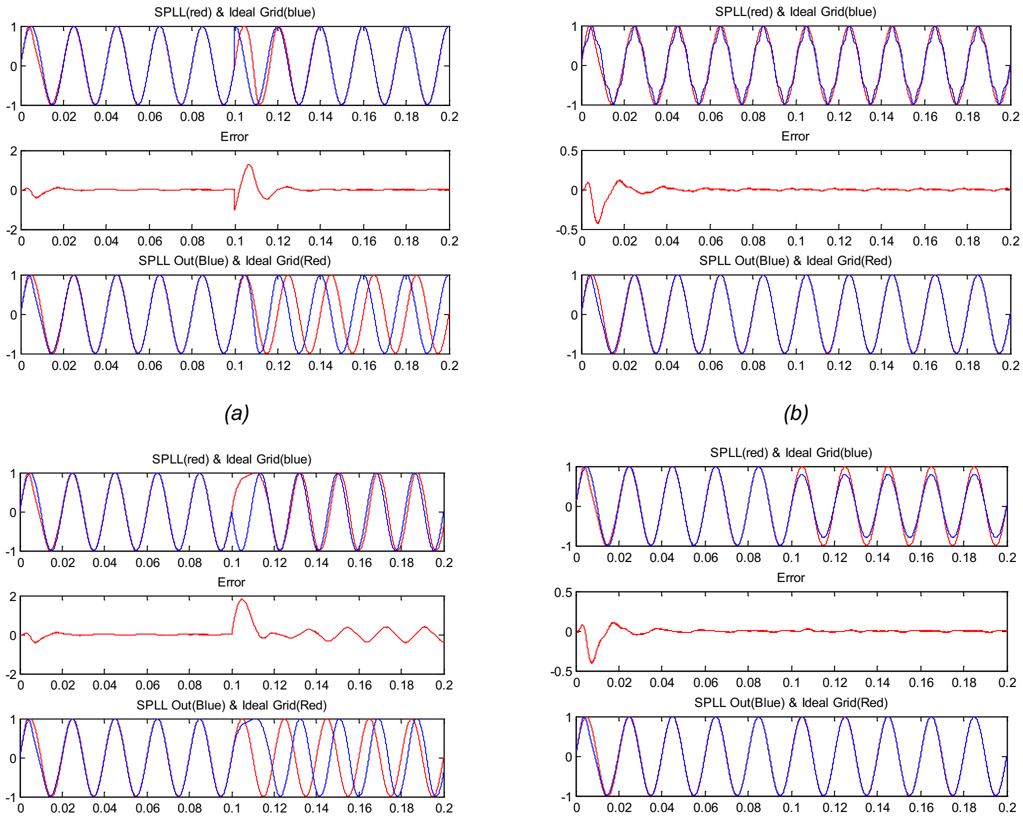
Before coding the SPLL structure it is essential to simulate the behavior of the PLL for different conditions on the grid. Fixed-point processors are used, for their for lower cost in many grid tied converters. IQ Math is a convenient way to look at fixed point numbers with a decimal point. C2000 IQ math library provides built-in functions that can simplify handling of the decimal point by the programmer. However, coding in fixed point can have additional issues of dynamic range and precision; therefore, it is better to simulate the behavior of fixed-point processors in a simulation environment. Hence, MATLAB® is used to simulate and identify the Q point at which the algorithm needs to run. Below, is the MATLAB script using the fixed-point toolbox that tests the PLL algorithm with varying grid condition.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%TI C2000
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Select numeric type, let's choose Q21
T=numerictype('WordLength',32,'FractionLength',21);

%Specify math attributes to the fimath object
F=fimath('RoundMode','floor','OverflowMode','wrap');
F.ProductMode='SpecifyPrecision';
F.ProductWordLength=32;
F.ProductFractionLength=21;
F.SumMode='SpecifyPrecision';
F.SumWordLength=32;
F.SumFractionLength=21;

%specify fipref object, to display warning in cases of overflow and
%underflow
P=fipref;
P.LoggingMode='on';
P.NumericTypeDisplay='none';
P.FimathDisplay='none';

%PLL Modelling starts from here
Fs=50000;               %Sampling frequency = 50Khz
GridFreq=50;            %Nominal Grid Frequency in Hz
Tfinal=0.2;             %Time the simulation is run for = 0.5 seconds

Ts=1/Fs;                %Sampling Time = 1/Fs
t=0:Ts:Tfinal;          %Simulation Time vector
wn=2*pi*GridFreq;       %Nominal Grid Frequency in radians

%generate input signal and create a fi object of it

%input wave with a phase jump at the mid point of simulation

% CASE 1 : Phase Jump at the Mid Point
L=length(t);
for n=1:floor(L)
    u(n)=sin(2*pi*GridFreq*Ts*n);
end
for n=1:floor(L)
    u1(n)=sin(2*pi*GridFreq*Ts*n);
end
for n=floor(L/2):L
    u(n)=sin(2*pi*GridFreq*Ts*n+pi/2);
end


%CASE 2 : Harmonics
%  L=length(t);
%  for n=1:floor(L)
%      u(n)=0.9*sin(2*pi*GridFreq*Ts*n)+0.1*sin(2*pi*5*GridFreq*Ts*n);
%  end
%  for n=1:floor(L)
%     u1(n)=sin(2*pi*GridFreq*Ts*n);
```

```
%  end

%CASE 3 : Frequency Shift
% L=length(t);
% for n=1:floor(L)
%     u(n)=sin(2*pi*GridFreq*Ts*n);
% end
% for n=1:floor(L)
%    u1(n)=sin(2*pi*GridFreq*Ts*n);
% end
% for n=floor(L/2):L
%     u(n)=sin(2*pi*GridFreq*1.1*Ts*n);
% end


%CASE 4: Amplitude Variations
% L=length(t);
% for n=1:floor(L)
%     u(n)=sin(2*pi*GridFreq*Ts*n);
% end
% for n=1:floor(L)
%    u1(n)=sin(2*pi*GridFreq*Ts*n);
% end
% for n=floor(L/2):L
%     u(n)=0.8*sin(2*pi*GridFreq*Ts*n);
% end;

u=fi(u,T,F);
u1=fi(u1,T,F);

%declare arrays used by the PLL process
Upd=fi([0,0,0],T,F);
ynotch=fi([0,0,0],T,F);
ynotch_buff=fi([0,0,0],T,F);
ylf=fi([0,0],T,F);

SinGen=fi([0,0],T,F);
Plot_Var=fi([0,0],T,F);
Mysin=fi([0,0],T,F);
Mycos=fi([fi(1.0,T,F),fi(1.0,T,F)],T,F);

theta=fi([0,0],T,F);
werror=fi([0,0],T,F);

%notch filter design
c1=0.1;
c2=0.00001;
X=2*c2*wn*2*Ts;
Y=2*c1*wn*2*Ts;
Z=wn*2*wn*2*Ts*Ts;

B_notch=[1 (X-2) (-X+Z+1)];
A_notch=[1 (Y-2) (-Y+Z+1)];

B_notch=fi(B_notch,T,F);
A_notch=fi(A_notch,T,F);

% simulate the PLL process
for n=2:Tfinal/Ts    % No of iteration of the PLL process in the simulation time

    % Phase Detect
    Upd(1)= u(n)*Mycos(2);

    %Notch Filter
    ynotch(1)=-A_notch(2)*ynotch(2)-
A_notch(3)*ynotch(3)+B_notch(1)*Upd(1)+B_notch(2)*Upd(2)+B_notch(3)*Upd(3);
```

```
    %update the Upd array for future sample
    Upd(3)=Upd(2);
    Upd(2)=Upd(1);


    % PI Loop Filter
    %ts=30ms, damping ration = 0.7
    % we get natural frequency = 110, Kp=166.6 and Ki=27755.55
    % B0=166.877556 & B1=-166.322444
    ylf(1)= fi(1.0,T,F)*ylf(2)+fi(166.877556,T,F)*ynotch(1)+fi(-166.322444,T,F)*ynotch(2);
    %update Ynotch for future use
    ynotch(3)=ynotch(2);
    ynotch(2)=ynotch(1);
    ynotch_buff(n+1)=ynotch(1);

    ylf(1)=min([ylf(1) fi(200.0,T,F)]);

    ylf(2)=ylf(1);

    wo=fi(wn,T,F)+ylf(1);

    werror(n+1)=(wo-wn)*fi(0.00318309886,T,F);

    %integration process
    Mysin(1)=Mysin(2)+wo*fi(Ts,T,F)*(Mycos(2));
    Mycos(1)=Mycos(2)-wo*fi(Ts,T,F)*(Mysin(2));
    %limit the oscillator integrators
    Mysin(1)=max([Mysin(1) fi(-1.0,T,F)]);
    Mysin(1)=min([Mysin(1) fi(1.0,T,F)]);
    Mycos(1)=max([Mycos(1) fi(-1.0,T,F)]);
    Mycos(1)=min([Mycos(1) fi(1.0,T,F)]);

    Mysin(2)=Mysin(1);
    Mycos(2)=Mycos(1);

    %update the output phase
    theta(1)=theta(2)+wo*Ts;
    %output phase reset condition
    if(Mysin(1)>0 && Mysin(2) <=0)
        theta(1)=-fi(pi,T,F);
    end

    SinGen(n+1)=Mycos(1);

    Plot_Var(n+1)=Mysin(1);

end

% CASE 1 : Phase Jump at the Mid Point
error=Plot_Var-u;

%CASE 2 : Harmonics
%error=Plot_Var-u1;

%CASE 3: Frequency Variations
%error=Plot_Var-u;

%CASE 4: Amplitude Variations
%error=Plot_Var-u1;

figure;
subplot(3,1,1),plot(t,Plot_Var,'r',t,u,'b'),title('SPLL(red) & Ideal Grid(blue)');
subplot(3,1,2),plot(t,error,'r'),title('Error');
subplot(3,1,3),plot(t,u1,'r',t,Plot_Var,'b'),title('SPLL Out(Blue) & Ideal Grid(Red)');
```

Figure 4 shows the results of the varying grid condition on the PLL simulation.




*(a)* *(b)*

*(c)* *(d)*

**Figure 4. PLL Response to Varying Grid Conditions**

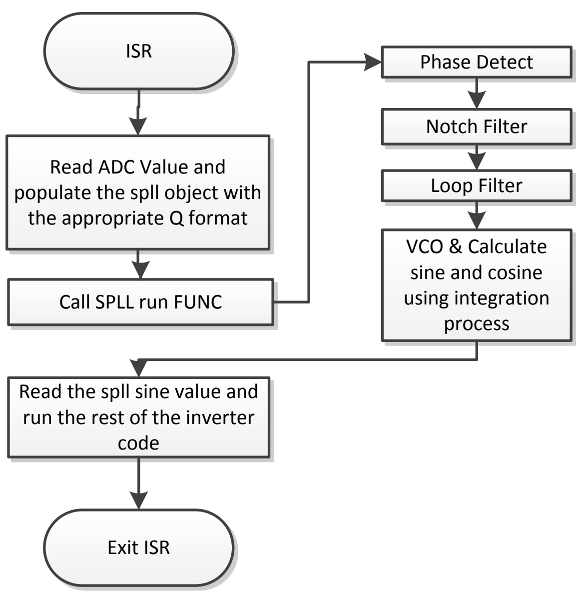
## 2.5 Implementing PLL on C2000 Controller Using IQ Math

Typical inverter code uses IQ24. From Section 2.4, it is known that the Q point chosen for the PLL is IQ21; thus, the code for the PLL can be written as follows:

```
#define _SPLL_1ph_H_

#define SPLL_Q _IQ21
#define SPLL_Qmpy _IQ21mpy

typedef struct{
       int32  B2_notch;
       int32  B1_notch;
       int32  B0_notch;
       int32  A2_notch;
       int32  1_notch;
}SPLL_NOTCH_COEFF;

typedef struct{
       int32  B1_lf;
       int32  B0_lf;
       int32  A1_lf;
}SPLL_LPF_COEFF;
typedef struct{
       int32  AC_input;
       int32  theta[2];
       int32  cos[2];
       int32  sin[2];
       int32  wo;
       int32  wn;

       SPLL_NOTCH_COEFF notch_coeff;
       SPLL_LPF_COEFF   lpf_coeff;

       int32   Upd[3];
       int32   ynotch[3];
       int32   ylf[2];
       int32   delta_t;
}SPLL_1ph;

void SPLL_1ph_init(int Grid_freq, long DELTA_T, SPLL_1ph *spll, SPLL_LPF_COEFF lpf_coeff);
void SPLL_1ph_notch_coeff_update(float delta_T, float wn,float c2, float c1, SPLL_1ph *spll_obj);
inline void SPLL_1ph_run_FUNC(SPLL_1ph *spll1);

void SPLL_1ph_init(int Grid_freq, long DELTA_T, SPLL_1ph *spll_obj, SPLL_LPF_COEFF lpf_coeff)
{
       spll_obj->Upd[0]=SPLL_Q(0.0);
       spll_obj->Upd[1]=SPLL_Q(0.0);
       spll_obj->Upd[2]=SPLL_Q(0.0);

       spll_obj->ynotch[0]=SPLL_Q(0.0);
       spll_obj->ynotch[1]=SPLL_Q(0.0);
       spll_obj->ynotch[2]=SPLL_Q(0.0);

       spll_obj->ylf[0]=SPLL_Q(0.0);
       spll_obj->ylf[1]=SPLL_Q(0.0);

       spll_obj->sin[0]=SPLL_Q(0.0);
       spll_obj->sin[1]=SPLL_Q(0.0);

       spll_obj->cos[0]=SPLL_Q(0.999);
       spll_obj->cos[1]=SPLL_Q(0.999);

       spll_obj->theta[0]=SPLL_Q(0.0);
       spll_obj->theta[1]=SPLL_Q(0.0);

       spll_obj->wn=SPLL_Q(2*3.14*Grid_freq);
```

```
        //coefficients for the loop filter
        spll_obj->lpf_coeff.B1_lf=lpf_coeff.B1_lf;
        spll_obj->lpf_coeff.B0_lf=lpf_coeff.B0_lf;
        spll_obj->lpf_coeff.A1_lf=lpf_coeff.A1_lf;

        spll_obj->delta_t=DELTA_T;

}


void SPLL_1ph_notch_coeff_update(float delta_T, float wn,float c2, float c1, SPLL_1ph *spll_obj)
{
        // Note c2<<c1 for the notch to work
        float x,y,z;
        x=(float)(2.0*c2*wn*delta_T);
        y=(float)(2.0*c1*wn*delta_T);
        z=(float)(wn*delta_T*wn*delta_T);

        spll_obj->notch_coeff.A1_notch=SPLL_Q(y-2);
        spll_obj->notch_coeff.A2_notch=SPLL_Q(z-y+1);
        spll_obj->notch_coeff.B0_notch=SPLL_Q(1.0);
        spll_obj->notch_coeff.B1_notch=SPLL_Q(x-2);
        spll_obj->notch_coeff.B2_notch=SPLL_Q(z-x+1);
}

inline void SPLL_1ph_run_FUNC(SPLL_1ph *spll_obj)
{

        //-------------------//
        // Phase Detect      //
        //-------------------//

        spll_obj->Upd[0]=SPLL_Qmpy(spll_obj->AC_input,spll_obj->cos[1]);

        //-------------------//
        //Notch filter structure//
        //-------------------//

        spll_obj->ynotch[0]=-SPLL_Qmpy(spll_obj->notch_coeff.A1_notch,spll_obj->ynotch[1])-
SPLL_Qmpy(spll_obj->notch_coeff.A2_notch,spll_obj->ynotch[2])+SPLL_Qmpy(spll_obj-
>notch_coeff.B0_notch,spll_obj->Upd[0])+SPLL_Qmpy(spll_obj->notch_coeff.B1_notch,spll_obj-
>Upd[1])+SPLL_Qmpy(spll_obj->notch_coeff.B2_notch,spll_obj->Upd[2]);

        // update the Upd array for future
        spll_obj->Upd[2]=spll_obj->Upd[1];
        spll_obj->Upd[1]=spll_obj->Upd[0];

        //---------------------------//
        // PI loop filter             //
        //---------------------------//

        spll_obj->ylf[0]=-SPLL_Qmpy(spll_obj->lpf_coeff.A1_lf,spll_obj->ylf[1])+SPLL_Qmpy(spll_obj-
>lpf_coeff.B0_lf,spll_obj->ynotch[0])+SPLL_Qmpy(spll_obj->lpf_coeff.B1_lf,spll_obj->ynotch[1]);

        //update array for future use
        spll_obj->ynotch[2]=spll_obj->ynotch[1];
        spll_obj->ynotch[1]=spll_obj->ynotch[0];

        spll_obj->ylf[1]=spll_obj->ylf[0];

        //------------------//
        // VCO              //
        //------------------//

        spll_obj->wo=spll_obj->wn+spll_obj->ylf[0];
```

```
        //integration process to compute sine and cosine
        spll_obj->sin[0]=spll_obj->sin[1]+SPLL_Qmpy((SPLL_Qmpy(spll_obj->delta_t,spll_obj-
>wo)),spll_obj->cos[1]);
        spll_obj->cos[0]=spll_obj->cos[1]-SPLL_Qmpy((SPLL_Qmpy(spll_obj->delta_t,spll_obj-
>wo)),spll_obj->sin[1]);

        if(spll_obj->sin[0]>SPLL_Q(0.99))
               spll_obj->sin[0]=SPLL_Q(0.99);
        else if(spll_obj->sin[0]<SPLL_Q(-0.99))
                   spll_obj->sin[0]=SPLL_Q(-0.99);

        if(spll_obj->cos[0]>SPLL_Q(0.99))
               spll_obj->cos[0]=SPLL_Q(0.99);
        else if(spll_obj->cos[0]<SPLL_Q(-0.99))
               spll_obj->cos[0]=SPLL_Q(-0.99);

        //compute theta value

        spll_obj->theta[0]=spll_obj->theta[1]+SPLL_Qmpy(SPLL_Qmpy(spll_obj-
>wo,SPLL_Q(0.159154943)),spll_obj->delta_t);

        if(spll_obj->sin[0]>SPLL_Q(0.0) && spll_obj->sin[1] <=SPLL_Q (0.0) )
        {
               spll_obj->theta[0]=SPLL_Q(0.0);

        spll_obj->theta[1]=spll_obj->theta[0];

        spll_obj->sin[1]=spll_obj->sin[0];
        spll_obj->cos[1]=spll_obj->cos[0];


}
#endif
```



To use this block in an end application, you need to include the header file and declare objects for the SPLL structure, and loop filter coefficients.

```
#include "SPLL_1ph.h"
// ------------- Software PLL for Grid Tie Applications ----------
SPLL_1ph spll1;
SPLL_LPF_COEFF spll_lpf_coef1;
```

You know from the analysis before the loop filter coefficients, write these into the SPLL lpf coeff structure.

```
#define B0_LPF SPLL_Q(166.877556)
#define B1_LPF SPLL_Q(-166.322444)
#define A1_LPF SPLL_Q(-1.0)

spll_lpf_coef1.B0_lf=B0_LPF;
spll_lpf_coef1.B1_lf=B1_LPF;
spll_lpf_coef1.A1_lf=A1_LPF;
```

Call the SPLL_1ph_init routine with the frequency of the ISR the SPLL will be executed in as parameter and the grid frequency and then call the notch filter update coefficient update routine.

```
SPLL_1ph_init(GRID_FREQ,_IQ21((float)(1.0/ISR_FREQUENCY)) &spll1,spll_lpf_coef1);
c1=0.1;
c2=0.00001;
SPLL_1ph_notch_coeff_update(((float)(1.0/ISR_FREQUENCY)),(float)(2*PI*GRID_FREQ*2),(float)c2,(floa
t)c1, &spll1);
```

In the ISR, read the sinusoidal input voltage from the ADC and feed it into the SPLL block, and write to invsine value with the sinus of the current grid angle. This can then be used in control operations.

```
inv_meas_vol_inst =((long)((long)VAC_FB<<12))-offset_165)<<1;
spll1.AC_input=(long)InvSine>>3; // Q24 to Q21
SPLL_1ph_run_FUNC(&spll1);
InvSine=spll2.sin<<3; // shift from Q21 to Q24
```

## 2.6 Results of Notch SPLL

Results of the implemented SPLL on the C2000 microcontroller F28035x are shown in Figure 5 at steady state and with a phase jump with scope captures.

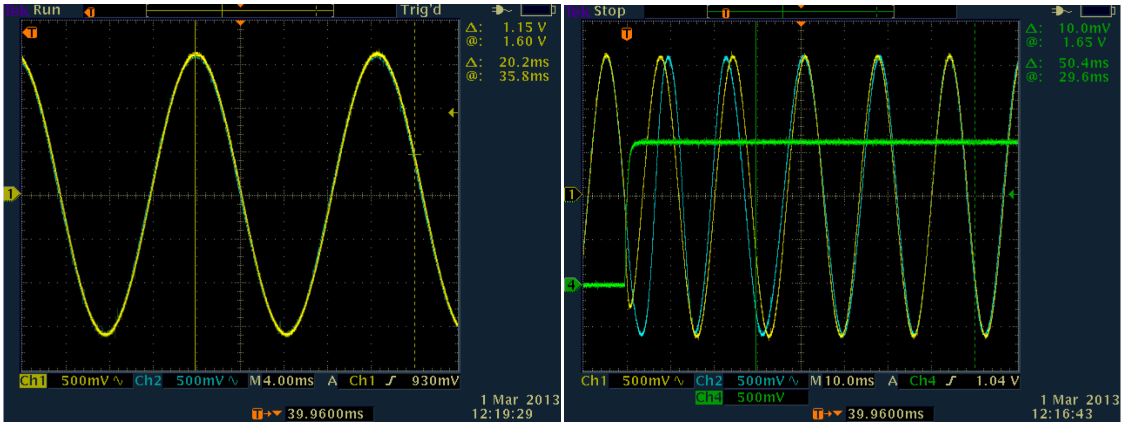



**Figure 5. Implemented SPLL at Steady State With Phase Jump and Scope Captures**

## 3 Orthogonal Signal Generator PLL

As discussed earlier, single phase grid software PLL design is tricky because of twice the grid frequency component present in the phase detect output. Notch filter was previously used to selectively eliminate this component and satisfactory results were achieved. Another alternative, to linearize the PD output, is to use an orthogonal signal generator scheme and then use park transformation. Synchronous reference frame PLL can then be used for single phase application. A functional diagram of such a PLL is shown in Figure 6, which consists of a PD consisting of orthogonal signal generator and park transform, LPF and VCO.

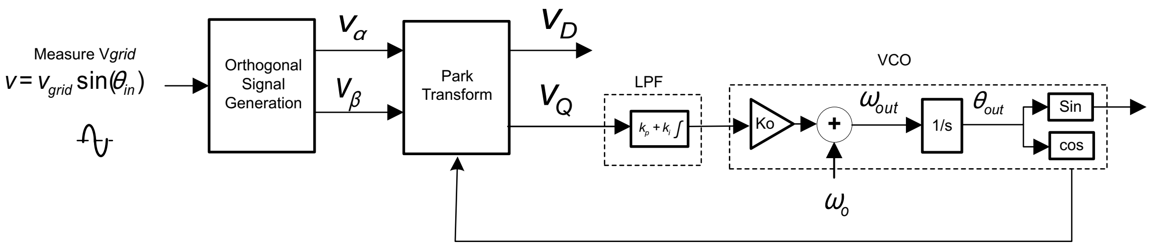



**Figure 6. OSG Based Single Phase PLL**

The orthogonal component from the input voltage signal can be generated in variety of different ways like transport delay, Hilbert transform, and so forth. A widely discussed method is to use a second order integrator as proposed in 'A New Single Phase PLL Structure Based on Second Order Generalized Integrator', Mihai Ciobotaru, et al, PESC'06. This method has advantage as it can be used to selectively tune the orthogonal signal generator to reject other frequencies except the grid frequency.

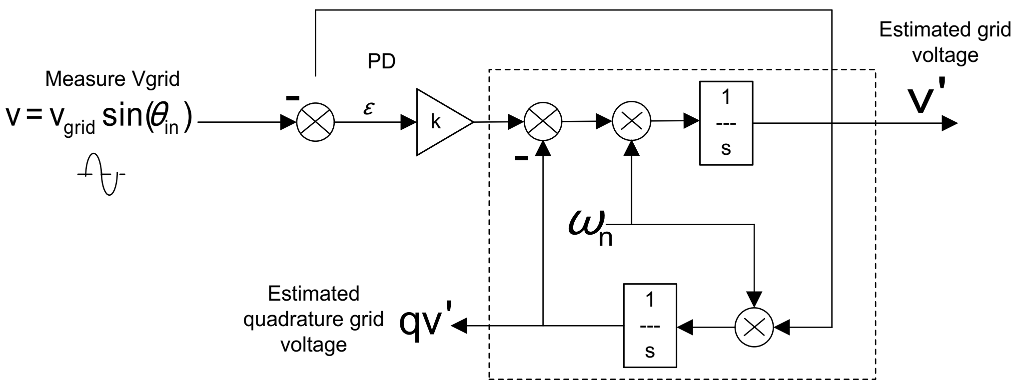



**Figure 7. Second Order Generalized Integrator for Orthogonal Signal Generation**

The second order generalized integrator closed-loop transfer function can be written as:

$$H_d(s) = \frac{v'}{v}(s) = \frac{k\omega_n s}{s^2 + k\omega_n s + \omega_n^2} \quad \text{and } H_q(s) = \frac{qv'}{v}(s) = \frac{k\omega_n^2}{s^2 + k\omega_n s + \omega_n^2} \tag{23}$$

As discussed in Section 2.6, the grid frequency can change, therefore, this orthogonal signal generator must be able to tune its coefficients in case of grid frequency change. To achieve this, trapezoidal approximation is used to get the discrete transfer function as follows:

$$H_d(z) = \frac{k\omega_n \frac{2}{T_s}\frac{z-1}{z+1}}{\left(\frac{2}{T_s}\frac{z-1}{z+1}\right)^2 + k\omega_n \frac{2}{T_s}\frac{z-1}{z+1} + \omega_n^2} = \frac{(2k\omega_n T_s)(z^2-1)}{4(z-1)^2 + (2k\omega_n T_s)(z^2-1) + (\omega_n T_s)^2 (z+1)^2} \tag{24}$$

Now, using $x = 2k\omega_n T_s$ and $y = (\omega_n T_s)^2$:

$$H_d(z) = \frac{\left(\frac{x}{x+y+4}\right) + \left(\frac{-x}{x+y+4}\right)z^{-2}}{1 - \left(\frac{2(4-y)}{x+y+4}\right)z^{-1} - \left(\frac{x-y-4}{x+y+4}\right)z^{-2}} = \frac{b_0 + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}} \tag{25}$$

Similarly,

$$H_q(z) = \frac{\left(\frac{k.y}{x+y+4}\right) + 2\left(\frac{k.y}{x+y+4}\right)z^{-1} + \left(\frac{k.y}{x+y+4}\right)z^{-2}}{1 - \left(\frac{2(4-y)}{x+y+4}\right)z^{-1} - \left(\frac{x-y-4}{x+y+4}\right)z^{-2}} = \frac{qb_0 + qb_1 z^{-1} + qb_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}} \quad (26)$$

Once the orthogonal signal has been generated, park transform is used to detect the Q and D components on the rotating reference frame. This is then fed to the loop filter that controls the VCO of the PLL. The tuning of the loop filter is similar to what is described for the notch filter in Section 2.1.

Additionally, the coefficients of the orthogonal signal generator can be tuned for varying grid frequency and sampling time (ISR frequency). The only variable is k, which determines the selectiveness of the frequency for the second order integrator. The second order generalized integrator presented can also be modified to extract the harmonic frequency component in a grid monitoring application, if needed. A lower k value must be selected for this purpose; however, lower k has an effect slowing the response.

Figure 8 illustrates the extraction of the fifth harmonic using the SOGI. The implementation of this follows from the SOGI implementation, which is discussed next; however, details are left out.

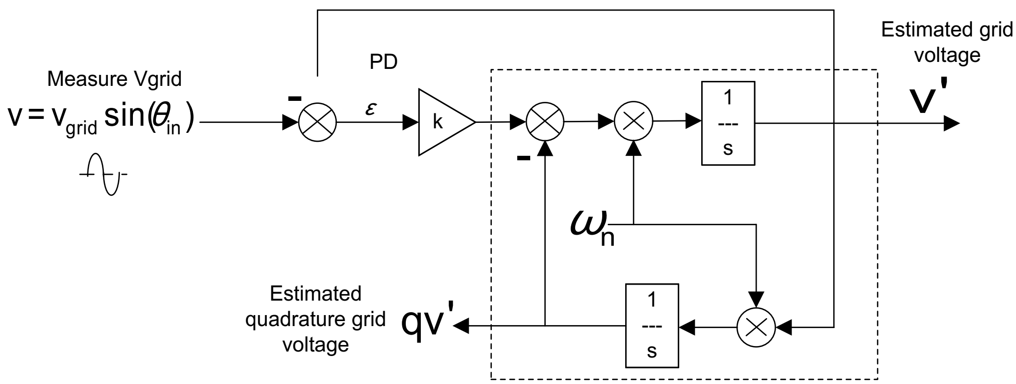



**Figure 8. Extraction of the Fifth Harmonic Using the SOGI**

Additionally, the RMS voltage of the grid can also be estimated using Equation 27:

$$V_{RMS} = \frac{1}{\sqrt{2}}\sqrt{v'^2 + qv'^2} \quad (27)$$

## 3.1 Simulating the Phase Locked Loop for Varying Conditions

Before coding the SPLL structure, it is essential to simulate the behavior of the PLL for different conditions on the grid. Fixed-point processors are used for lower cost in many grid tied converters. IQ Math is a convenient way to look at fixed point numbers with a decimal point. C2000 IQ math library provides built-in functions that can simplify handling of the decimal point by the programmer. First, MATLAB is used to simulate and identify the Q point at which the algorithm needs to run. Below is the MATLAB script using the fixed-point toolbox that tests the PLL algorithm with varying grid condition.

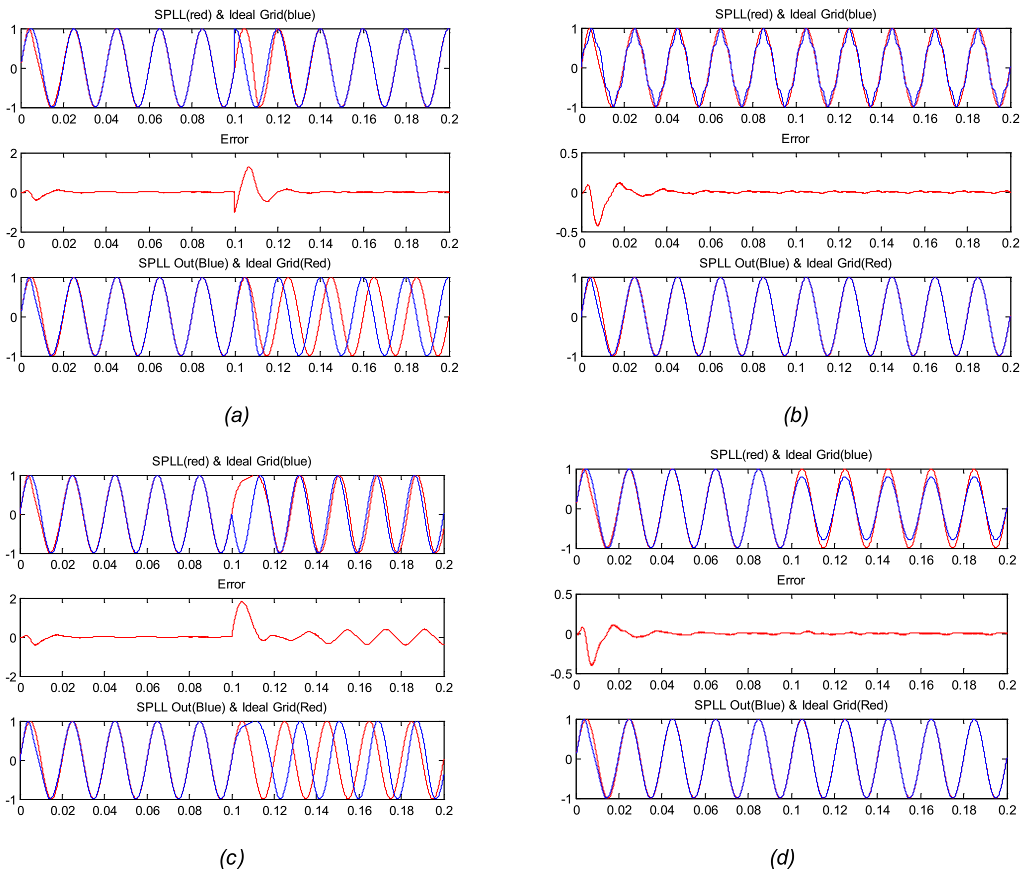
```
clear all;
close all;
clc;
% define the math type being used on the controller using objects from the
% fixed-point tool box in matlab

%Select numeric type, let's choose Q23
T=numerictype('WordLength',32,'FractionLength',23);

%Specify math attributes to the fimath object
F=fimath('RoundMode','floor','OverflowMode','wrap');
F.ProductMode='SpecifyPrecision';
F.ProductWordLength=32;
F.ProductFractionLength=23;
F.SumMode='SpecifyPrecision';
```

```
F.SumWordLength=32;
F.SumFractionLength=23;

%specify fipref object, to display warning in cases of overflow and
%underflow
P=fipref;
P.LoggingMode='on';
P.NumericTypeDisplay='none';
P.FimathDisplay='none';

%PLL Modelling starts from here
Fs=50000;               %Sampling frequency = 50Khz
GridFreq=50;            %Nominal Grid Frequency in Hz
Tfinal=0.2;             %Time the simulation is run for = 0.5 seconds

Ts=1/Fs;                %Sampling Time = 1/Fs
t=0:Ts:Tfinal;          %Simulation Time vector
wn=2*pi*GridFreq;       %Nominal Grid Frequency in radians

%declare arrays used by the PLL process
err=fi([0,0,0,0,0],T,F);
ylf=fi([0,0,0,0,0],T,F);
Mysin=fi([0,0,0,0,0],T,F);
Mycos=fi([1,1,1,1,1],T,F);
theta=fi([0,0,0,0,0],T,F);
dc_err=fi([0,0,0,0,0],T,F);
wo=fi(0,T,F);

% used for plotting
Plot_Var=fi([0,0,0,0],T,F);
Plot_theta=fi([0,0,0,0],T,F);
Plot_osgu=fi([0,0,0,0],T,F);
Plot_osgqu=fi([0,0,0,0],T,F);
Plot_D=fi([0,0,0,0],T,F);
Plot_Q=fi([0,0,0,0],T,F);
Plot_dc_err=fi([0,0,0,0,0],T,F);

%orthogonal signal generator
%using trapezoidal approximation
osg_k=0.5;
osg_x=2*osg_k*wn*Ts;
osg_y=(wn*wn*Ts*Ts);
osg_b0=osg_x/(osg_x+osg_y+4);
osg_b2=-1*osg_b0;
osg_a1=(2*(4-osg_y))/(osg_x+osg_y+4);
osg_a2=(osg_x-osg_y-4)/(osg_x+osg_y+4);

osg_qb0=(osg_k*osg_y)/(osg_x+osg_y+4);
osg_qb1=2*osg_qb0;
osg_qb2=osg_qb0;


osg_k=fi(osg_k,T,F);
osg_x=fi(osg_x,T,F);
osg_y=fi(osg_y,T,F);
osg_b0=fi(osg_b0,T,F);
osg_b2=fi(osg_b2,T,F);
osg_a1=fi(osg_a1,T,F);
osg_a2=fi(osg_a2,T,F);
osg_qb0=fi(osg_qb0,T,F);
osg_qb1=fi(osg_qb1,T,F);
osg_qb2=fi(osg_qb2,T,F);

osg_u=fi([0,0,0,0,0,0],T,F);
osg_qu=fi([0,0,0,0,0,0],T,F);
```

```
u_Q=fi([0,0,0],T,F);
u_D=fi([0,0,0],T,F);

%generate input signal

% CASE 1 : Phase Jump at the Mid Point
L=length(t);
for n=1:floor(L)
    u(n)=sin(2*pi*GridFreq*Ts*n);
end
for n=1:floor(L)
    u1(n)=sin(2*pi*GridFreq*Ts*n);
end

for n=floor(L/2):L
       u(n)=sin(2*pi*GridFreq*Ts*n+pi/2);
end

u=fi(u,T,F);

% simulate the PLL process
for n=3:Tfinal/Ts    % No of iteration of the PLL process in the simulation time

    %Orthogonal Signal Generator
    osg_u(1)=(osg_b0*(u(n)-u(n-2)))+osg_a1*osg_u(2)+osg_a2*osg_u(3);
    osg_u(3)=osg_u(2);
    osg_u(2)=osg_u(1);

    osg_qu(1)=(osg_qb0*u(n)+osg_qb1*u(n-1)+osg_qb2*u(n-2))+osg_a1*osg_qu(2)+osg_a2*osg_qu(3);
    osg_qu(3)=osg_qu(2);
    osg_qu(2)=osg_qu(1);

    %park trasnform from alpha beta to d-q axis
    u_Q(1)=Mycos(2)*osg_u(1)+Mysin(2)*osg_qu(1);
    u_D(1)=-Mysin(2)*osg_u(1)+Mycos(2)*osg_qu(1);

    %Loop Filter
    ylf(1)=fi(1,T,F)*ylf(2)+fi(166.877556,T,F)*u_Q(1)+fi(-166.322444,T,F)*u_Q(2);

    u_Q(2)=u_Q(1);
    u_D(2)=u_D(1);

    %Limit LF according to its Q? size pipeline
    ylf(1)=max([ylf(1) fi(-128,T,F)]);
    ylf(1)=min([ylf(1) fi(128,T,F)]);

    ylf(2)=ylf(1);

    %update output frequency
    wo=GridFreq+ylf(1);

    %update the output phase
    theta(1)=theta(2)+wo*fi(Ts,T,F);

    if(theta(1)>fi(1.0,T,F))
        theta(1)=fi(0,T,F);
    end

    theta(2)=theta(1);
    Mysin(1)=sin(theta(1)*fi(2*pi,T,F));
    Mycos(1)=cos(theta(1)*fi(2*pi,T,F));
    Mysin(2)=Mysin(1);
    Mycos(2)=Mycos(1);

    Plot_theta(n+1)=theta(1);
    Plot_osgu(n+1)=osg_u(1);
```

```
    Plot_osgqu(n+1)=osg_qu(1);
    Plot_Var(n+1)=Mysin(1);
    Plot_D(n+1)=u_D(1);
    Plot_Q(n+1)=u_Q(1);

end

% CASE 1 : Phase Jump at the Mid Point
error=Plot_Var-u;

%CASE 2 : Harmonics
%error=Plot_Var-u1;

%CASE 3: Frequency Variations
%error=Plot_Var-u;

%CASE 4: Amplitude Variations
%error=Plot_Var-u1;

subplot(3,1,1),plot(t,Plot_Var,'r',t,u,'b'),title('SPLL(red) & Ideal Grid(blue)');
subplot(3,1,2),plot(t,error,'r'),title('Error');
subplot(3,1,3),plot(t,u1,'r',t,Plot_Var,'b'),title('SPLL Out(Blue) & Ideal Grid(Red)');
```

Figure 9 shows the results of the varying condition on the PLL.




A Phase Jump of 90°

B Phase Jump of 90°

C Frequency drift at mid point highlights the need for adaptive notch filter

D Amplitude change (Voltage Sags and Dips)

**Figure 9. PLL Response to Varying Grid Conditions**

### 3.2 Implementing PLL on C2000 Controller Using IQ Math

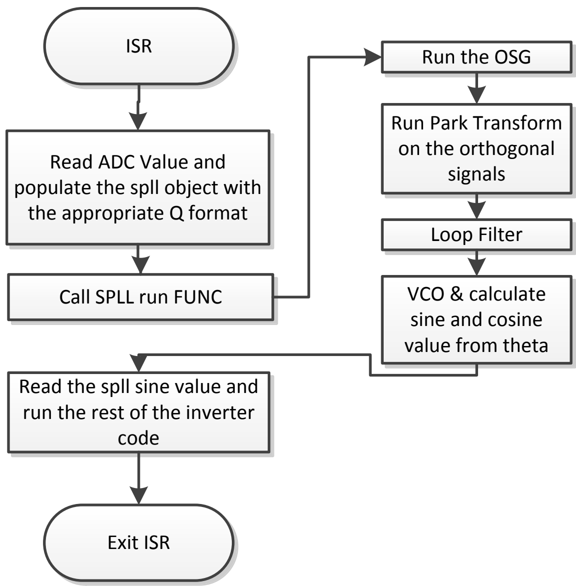
As the regular inverter typically uses IQ24, and as shown in Section 3.1, the Q point chosen for the PLL is IQ23. The code for the PLL can be written as follows:

```
#ifndef SPLL_1ph_SOGI_H_
#define SPLL_1ph_SOGI_H_

#define SPLL_SOGI_Q _IQ23
#define SPLL_SOGI_Qmpy _IQ23mpy
#define SPLL_SOGI_SINE _IQ23sin
#define SPLL_SOGI_COS  _IQ23cos

//*********** Structure Definition ********//
typedef struct{
               int32    osg_k;
               int32    osg_x;
               int32    osg_y;
               int32    osg_b0;
               int32    osg_b2;
               int32    osg_a1;
               int32    osg_a2;
               int32    osg_qb0;
               int32    osg_qb1;
               int32    osg_qb2;
}SPLL_SOGI_OSG_COEFF;

typedef struct{
               int32    B1_lf;
               int32    B0_lf;
               int32    A1_lf;
}SPLL_SOGI_LPF_COEFF;

typedef struct{
               int32      u[3];  // Ac Input
               int32   osg_u[3];
               int32   osg_qu[3];
               int32   u_Q[2];
               int32   u_D[2];
               int32   ylf[2];
               int32   fo; // output frequency of PLL
               int32   fn; //nominal frequency
               int32      theta[2];
               int32      cos;
               int32      sin;
               int32   delta_T;
               SPLL_SOGI_OSG_COEFF osg_coeff;
               SPLL_SOGI_LPF_COEFF lpf_coeff;
}SPLL_1ph_SOGI;

//*********** Function Declarations *******//
inline void SPLL_1ph_SOGI_init(int Grid_freq, long DELTA_T, volatile SPLL_1ph_SOGI *spll,
volatile SPLL_SOGI_LPF_COEFF lpf_coeff);
inline void SPLL_1ph_SOGI_coeff_update(float delta_T, float wn, volatile SPLL_1ph_SOGI *spll);
inline void SPLL_1ph_SOGI_run_FUNC(SPLL_1ph_SOGI *spll1);

//*********** Structure Init Function ****//
inline void SPLL_1ph_SOGI_init(int Grid_freq, long DELTA_T, volatile SPLL_1ph_SOGI *spll_obj,
volatile SPLL_SOGI_LPF_COEFF lpf_coeff)
{
               spll_obj->u[0]=SPLL_SOGI_Q(0.0);
               spll_obj->u[1]=SPLL_SOGI_Q(0.0);
               spll_obj->u[2]=SPLL_SOGI_Q(0.0);

               spll_obj->osg_u[0]=SPLL_SOGI_Q(0.0);
               spll_obj->osg_u[1]=SPLL_SOGI_Q(0.0);
               spll_obj->osg_u[2]=SPLL_SOGI_Q(0.0);
```

```
                spll_obj->osg_qu[0]=SPLL_SOGI_Q(0.0);
                spll_obj->osg_qu[1]=SPLL_SOGI_Q(0.0);
                spll_obj->osg_qu[2]=SPLL_SOGI_Q(0.0);

                spll_obj->u_Q[0]=SPLL_SOGI_Q(0.0);
                spll_obj->u_Q[1]=SPLL_SOGI_Q(0.0);

                spll_obj->u_D[0]=SPLL_SOGI_Q(0.0);
                spll_obj->u_D[1]=SPLL_SOGI_Q(0.0);

                spll_obj->ylf[0]=SPLL_SOGI_Q(0.0);
                spll_obj->ylf[1]=SPLL_SOGI_Q(0.0);

                spll_obj->fo=SPLL_SOGI_Q(0.0);
                spll_obj->fn=SPLL_SOGI_Q(Grid_freq);

                spll_obj->theta[0]=SPLL_SOGI_Q(0.0);
                spll_obj->theta[1]=SPLL_SOGI_Q(0.0);

                spll_obj->sin=SPLL_SOGI_Q(0.0);
                spll_obj->cos=SPLL_SOGI_Q(0.0);

    //coefficients for the loop filter
                spll_obj->lpf_coeff.B1_lf=lpf_coeff.B1_lf;
                spll_obj->lpf_coeff.B0_lf=lpf_coeff.B0_lf;
                spll_obj->lpf_coeff.A1_lf=lpf_coeff.A1_lf;

                spll_obj->delta_T=DELTA_T;

}

//*********** Structure Coeff Update *****//
inline void SPLL_1ph_SOGI_coeff_update(float delta_T, float wn, volatile SPLL_1ph_SOGI *spll)
{
                float osgx,osgy,temp;
                   spll->osg_coeff.osg_k=SPLL_SOGI_Q(0.5);
                   osgx=(float)(2.0*0.5*wn*delta_T);
                   spll->osg_coeff.osg_x=SPLL_SOGI_Q(osgx);
                   osgy=(float)(wn*delta_T*wn*delta_T);
                   spll->osg_coeff.osg_y=SPLL_SOGI_Q(osgy);
                   temp=(float)1.0/(osgx+osgy+4.0);
                   spll->osg_coeff.osg_b0=SPLL_SOGI_Q((float)osgx*temp);
                   spll->osg_coeff.osg_b2=SPLL_SOGI_Qmpy(SPLL_SOGI_Q(-1.0),spll->osg_coeff.osg_b0);
                   spll->osg_coeff.osg_a1=SPLL_SOGI_Q((float)(2.0*(4.0-osgy))*temp);
                   spll->osg_coeff.osg_a2=SPLL_SOGI_Q((float)(osgx-osgy-4)*temp);
                   spll->osg_coeff.osg_qb0=SPLL_SOGI_Q((float)(0.5*osgy)*temp);
                   spll->osg_coeff.osg_qb1=SPLL_SOGI_Qmpy(spll-
>osg_coeff.osg_qb0,SPLL_SOGI_Q(2.0));
                   spll->osg_coeff.osg_qb2=spll->osg_coeff.osg_qb0;
}

//*********** Function Definition ********//
inline void SPLL_1ph_SOGI_run_FUNC(SPLL_1ph_SOGI *spll_obj)
{
                // Update the spll_obj->u[0] with the grid value before calling this routine

                   //-------------------------------//
                // Orthogonal Signal Generator      //
                //-------------------------------//
                spll_obj->osg_u[0]=SPLL_SOGI_Qmpy(spll_obj->osg_coeff.osg_b0,(spll_obj->u[0]-
spll_obj->u[2]))+SPLL_SOGI_Qmpy(spll_obj->osg_coeff.osg_a1,spll_obj-
>osg_u[1])+SPLL_SOGI_Qmpy(spll_obj->osg_coeff.osg_a2,spll_obj->osg_u[2]);

                    spll_obj->osg_u[2]=spll_obj->osg_u[1];
                    spll_obj->osg_u[1]=spll_obj->osg_u[0];
```

```
                    spll_obj->osg_qu[0]=SPLL_SOGI_Qmpy(spll_obj->osg_coeff.osg_qb0,spll_obj-
>u[0])+SPLL_SOGI_Qmpy(spll_obj->osg_coeff.osg_qb1,spll_obj->u[1])+SPLL_SOGI_Qmpy(spll_obj-
>osg_coeff.osg_qb2,spll_obj->u[2])+SPLL_SOGI_Qmpy(spll_obj-
                >osg_coeff.osg_a1,spll_obj->osg_qu[1])+SPLL_SOGI_Qmpy(spll_obj-
>osg_coeff.osg_a2,spll_obj->osg_qu[2]);

                   spll_obj->osg_qu[2]=spll_obj->osg_qu[1];
                spll_obj->osg_qu[1]=spll_obj->osg_qu[0];

                spll_obj->u[2]=spll_obj->u[1];
                spll_obj->u[1]=spll_obj->u[0];

                //----------------------------------------------------------//
                // Park Transform from alpha beta to d-q axis              //
                //----------------------------------------------------------//
                spll_obj->u_Q[0]=SPLL_SOGI_Qmpy(spll_obj->cos,spll_obj-
>osg_u[0])+SPLL_SOGI_Qmpy(spll_obj->sin,spll_obj->osg_qu[0]);
                spll_obj->u_D[0]=SPLL_SOGI_Qmpy(spll_obj->cos,spll_obj->osg_qu[0])-
SPLL_SOGI_Qmpy(spll_obj->sin,spll_obj->osg_u[0]);

                //----------------------------------//
                // Loop Filter                      //
                    /----------------------------------//
                spll_obj->ylf[0]=spll_obj->ylf[1]+SPLL_SOGI_Qmpy(spll_obj-
>lpf_coeff.B0_lf,spll_obj->u_Q[0])+SPLL_SOGI_Qmpy(spll_obj->lpf_coeff.B1_lf,spll_obj->u_Q[1]);
                //spll_obj->ylf[0]=(spll_obj->ylf[0]>SPLL_SOGI_Q(20.0))?SPLL_SOGI_Q(20.0):spll_obj-
>ylf[0];
                //spll_obj->ylf[0]=(spll_obj->ylf[0] < SPLL_SOGI_Q(-20.0))?SPLL_SOGI_Q(-
20.0):spll_obj->ylf[0];
                spll_obj->ylf[1]=spll_obj->ylf[0];

                spll_obj->u_Q[1]=spll_obj->u_Q[0];
                //spll_obj->u_D[1]=spll_obj->u_D[0];

                //----------------------------------//
                // VCO                              //
                //----------------------------------//
                spll_obj->fo=spll_obj->fn+spll_obj->ylf[0];

                spll_obj->theta[0]=spll_obj->theta[1]+SPLL_SOGI_Qmpy(SPLL_SOGI_Qmpy(spll_obj-
>fo,spll_obj->delta_T),SPLL_SOGI_Q(2*3.1415926));
                if(spll_obj->theta[0]>SPLL_SOGI_Q(2*3.1415926))
                               spll_obj->theta[0]=spll_obj->theta[0]-SPLL_SOGI_Q(2*3.1415926);

                spll_obj->theta[1]=spll_obj->theta[0];

                spll_obj->sin=SPLL_SOGI_SINE(spll_obj->theta[0]);
                spll_obj->cos=SPLL_SOGI_COS(spll_obj->theta[0]);
}

//*********** Macro Definition ***********//
#define SPLL_1ph_SOGI_run_MACRO(v) \
                v.osg_u[0]=SPLL_SOGI_Qmpy(v.osg_coeff.osg_b0,(v.u[0]-
v.u[2]))+SPLL_SOGI_Qmpy(v.osg_coeff.osg_a1,v.osg_u[1])+SPLL_SOGI_Qmpy(v.osg_coeff.osg_a2,v.osg_u[2
]);\
                v.osg_u[2]=v.osg_u[1];\
                v.osg_u[1]=v.osg_u[10];\

v.osg_qu[0]=SPLL_SOGI_Qmpy(v.osg_coeff.osg_qb0,v.u[0])+SPLL_SOGI_Qmpyu(v.osg_coeff.osg_qb1,v.u[1])
+SPLL_SOGI_Qmpy(v.osg_coeff.osg_qb2,v.u[2])+SPLL_SOGI_Qmpy(v.osg_coeff.osg_a1,v.osg_qu[1])+
                SPLL_SOGI_Qmpy(v.osg_coeff.osg_a2.v.osg_qu[2]);
\
                v.osg_qu[2]=v.osg_qu[1];           \
                v.osg_qu[1]=v.osg_qu[0];           \
                v.u[2]=v.u[1];    \
```

```
                v.u[1]=v.u[0];    \
                v.u_Q[0]=SPLL_SOGI_Qmpy(v.cos,v.osg_u[0])+SPLL_SOGI_Qmpy(v.sin,v.osg_qu[0]);  \
                v.u_D[0]=SPLL_SOGI_Qmpy(v.cos,v.osg_qu[0])-SPLL_SOGI_Qmpy(v.sin,v.osg_u[0]);    \

v.ylf[0]=v.ylf[1]+SPLL_SOGI_Qmpy(v.lpf_coeff.B0_lf,v.u_Q[0])+SPLL_SOGI_Qmpy(v.lpf_coeff.B1_lf,v.u_
Q[1]);                \
                v.ylf[1]=v.ylf[0];               \
                v.u_Q[1]=v.u_Q[0];        \
                v.fo=v.fn+v.ylf[0]; \

v.theta[0]=v.theta[1]+SPLL_SOGI_Qmpy(SPLL_SOGI_Qmpy(v.fov.v.delta_T),SPLL_Q(2*3.1415926)); \
                if(v.theta[0]>SPLL_SOGI_Q(2*3.1415926)) \
                                v.theta[0]=SPLL_SOGI_Q(0.0); \
                v.theta[1]=v.theta[0]; \
                v.sin=SPLL_SOGI_SINE(v.theta[0]); \
                v.cos=SPLL_SOGI_COS(v.theta[0]);

#endif/* SPLL_1ph_SOGI_H_ */
```




To use this block in an end application, you must include the header file and declare objects for the SPLL structure and loop filter coefficients.

```
#include "SPLL_1ph_SOGI.h"
SPLL_1ph_SOGI spll1;
SPLL_SOGI_LPF_COEFF spll_lpf_coef1;
```

From the analysis in Section 2.1, the loop filter coefficients are known, write these into the spll lpf coeff structure.

```
#define B0_LPF SPLL_SOGI_Q(166.877556)
#define B1_LPF SPLL_SOGI_Q(-166.322444)
#define A1_LPF SPLL_SOGI_Q(-1.0)
spll_lpf_coef1.B0_lf=B0_LPF;
spll_lpf_coef1.B1_lf=B1_LPF;
spll_lpf_coef1.A1_lf=A1_LPF;
```

Call the SPLL_1ph_SOGI_init routine with the frequency of the ISR. The SPLL will be executed in as the parameter and the grid frequency and then call the notch filter update coefficient update routine.

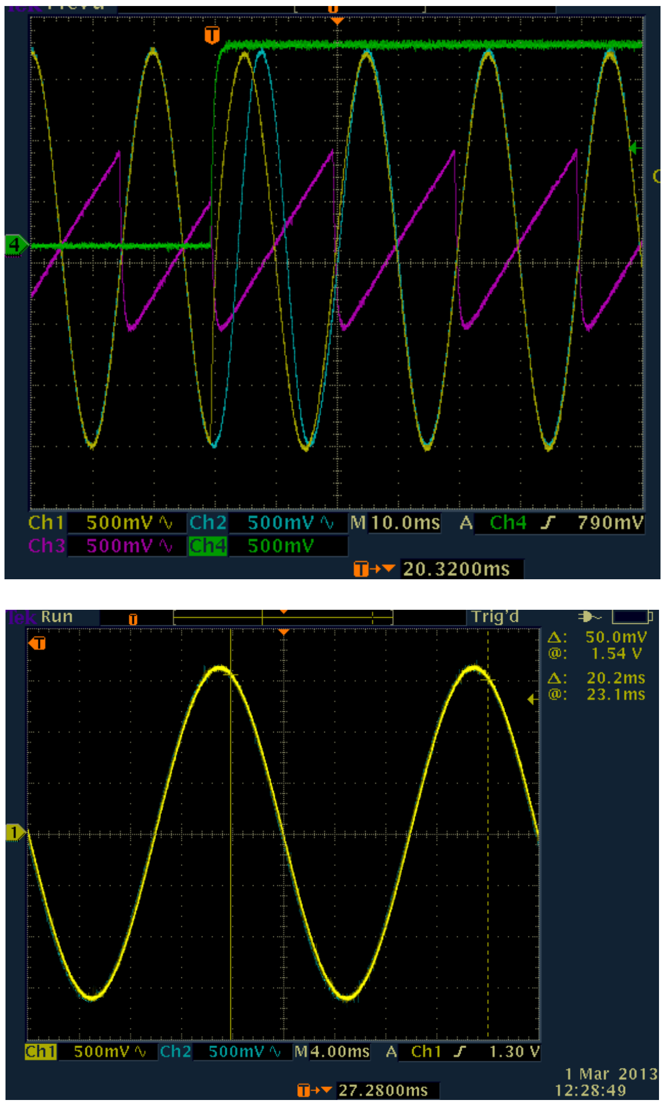
```
SPLL_1ph_SOGI_init(GRID_FREQ,_IQ23((float)(1.0/ISR_FREQUENCY)),&spll2,spll_lpf_coef2);
SPLL_1ph_SOGI_coeff_update(((float)(1.0/ISR_FREQUENCY)),(float)(2*PI*GRID_FREQ),&spll2);
```

In the ISR, read the sinusoidal input voltage from the ADC and feed it into the SPLL block, and write to invsine value with the sinus of the current grid angle. This can then be used in control operations.

```
inv_meas_vol_inst =((long)((long)VAC_FB<<12))-offset_165)<<1;
spll1.AC_input=(long)InvSine>>1; // Q24 to Q23
SPLL_1ph_SOGI_run_FUNC(&spll1);
InvSine=spll2.sin<<1; // shift from Q23 to Q24
```

## 3.3 Result of SOGI PLL

Very fast transient response is possible as shown in Figure 10.




A Ch1: is the grid sine value

B ch2: is the PLL lock

C ch3: is the grid theta

D ch4: is the phase jump

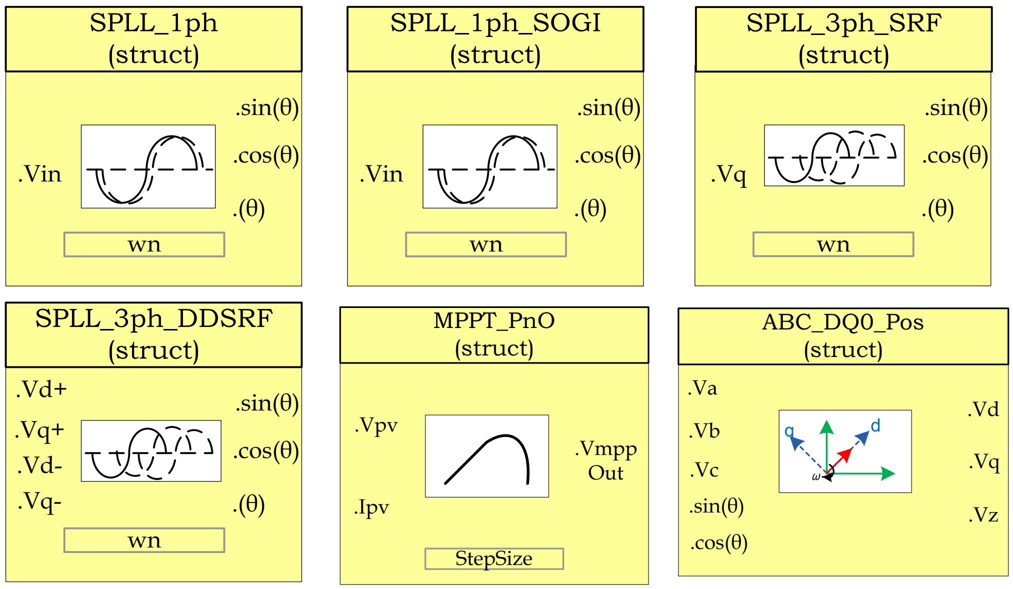
**Figure 10. Transient Response**

## 4 Solar Library and ControlSuite™

C2000 provides software for development of control application using C2000 devices. The software is available for download from http://www.ti.com/controlSUITE.

Various library, like the IQ math Library and Solar Library, simplify the design and development of algorithms on the C2000 device. IQ math library provides a convenient way to easily translate the fixed-point toolbox code into controller code.

The solar library provides the software blocks discussed in this application report, and many more.




## 5 References


- Francisco D. Freijedo et al, “Robust Phase Locked Loops Optimized for DSP implementation in Power Quality Applications”, IECON 2008, 3052-3057
- *TMS320F28030, TMS320F28031, TMS320F28032, TMS320F28033, TMS320F28034, TMS320F28035 Piccolo Microcontrollers Data Manual* (SPRS584)
- C2000 controlSUITE
- Marco Liserre, Pedro Rodriguez Remus Teodrescu, Grid Converters for Phototvoltaic and Wind Power Systems: John Wiley & Sons Ltd., 2011.
- P. Rodriguez et al, "Double Synchronous Reference Frame PLL for Power Converters Control," vol. 22, no. 2, 2007.

# Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

| Changes from Original (July 2013) to A Revision | Page |
| --- | --- |
| • Changes were made in Section 2.1 | 4 |

## IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES