C28x Solar Library

v1.0

Jan-12

Module User's Guide

C28x Foundation Software

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Trademarks

Acronyms

C28x: Refers to devices with the C28x CPU core.

IQmath: Fixed-point mathematical functions in C.

Q-math: Fixed point numeric format defining the binary resolution in bits.

Float: IEEE single precision floating point number

Contents

Chapter 1. Introduction

1.1. Introduction

Texas Instruments Solar library is designed to enable flexible and efficient coding of systems designed to use/process solar power using the C28x processor.

Solar applications need different software algorithms like maximum power tracking, phase lock loop for grid synchronization, power monitoring etc. Several different algorithms have been proposed in literature for the various tasks in a solar system. The Solar library provides a framework structure with known algorithms for the user to implement Solar System quickly. The source code for all the blocks is provided and hence the user can modify / enhance the modules for use in their applications with C2000 family of devices microcontrollers.

Chapter 2. Installing the Solar Library

2.1. Solar Library Package Contents

The TI Solar library consists of the following components:

- Header files consisting of the macro structure definition and macro code
- Documentation

2.2. How to Install the Solar Library

The Solar Library is distributed through the controlSUITE installer. The user must select the Solar Library Checkbox to install the library in the controlSUITE directory. By default, the installation places the library components in the following directory structure:

<base> install directory is C:\ti\controlSUITE\libs\app_libs\solar\vX.X

The following sub-directory structure is used:

<base>\float	Contains floating point implementation of the solar library blocks for floating point devices
<base>\IQ	Contains fixed point implementation of the solar library blocks for fixed point devices

Chapter 3. Module Summary

3.1. Solar Library Function Summary

The Solar Library consists of modules than enable the user to implement digital control of solar based systems. The following table lists the modules existing in the solar library and a summary of cycle counts.

Module Name	**Module Type**	**Description**	**Cycles**	**Multiple Instance Support**
mppt_pno	MPPT	Perturb and Observe MPPT Algorithm Module	~100	Yes
mppt_incc	MPPT	Incremental Conductance MPPT Algorithm Module	~300	Yes
spll_1ph	PLL	Software PLL for single phase grid connected application	~187	Yes
pid_grando	CNTL	PID module	~86	Yes
SineAnalyzer_diff	MATH	Calculate average and rms of a sinusoidal signal	~40 (data buffering) ~348 (calculations)	Yes

Chapter 4. Solar Lib Modules

4.1. Maximum Power Point Tracking (MPPT)

A simplistic model of a PV cell is given by Figure 1

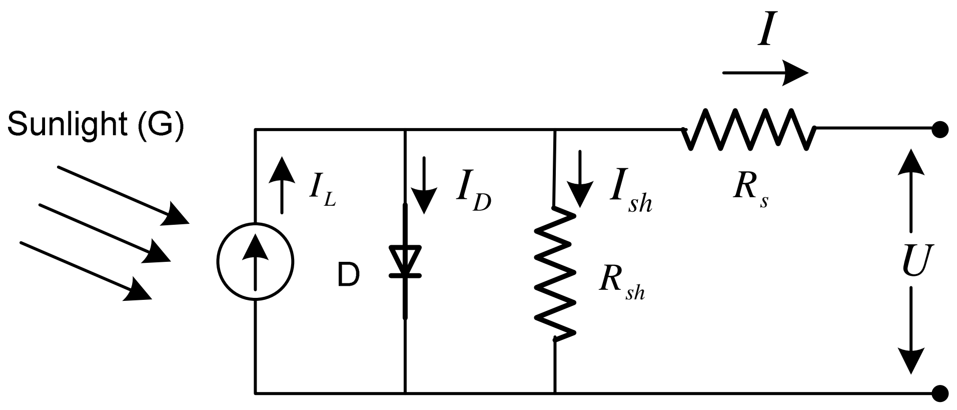


Figure 1 PV Cell Model

From which the equation for the current from the PV cell is given by :

$$I = I_L - I_o\left(e^{\frac{q(V+IR_s)}{nkT}} - 1\right)$$

Thus the V-I Curves for the solar cell is as shown Figure 2:

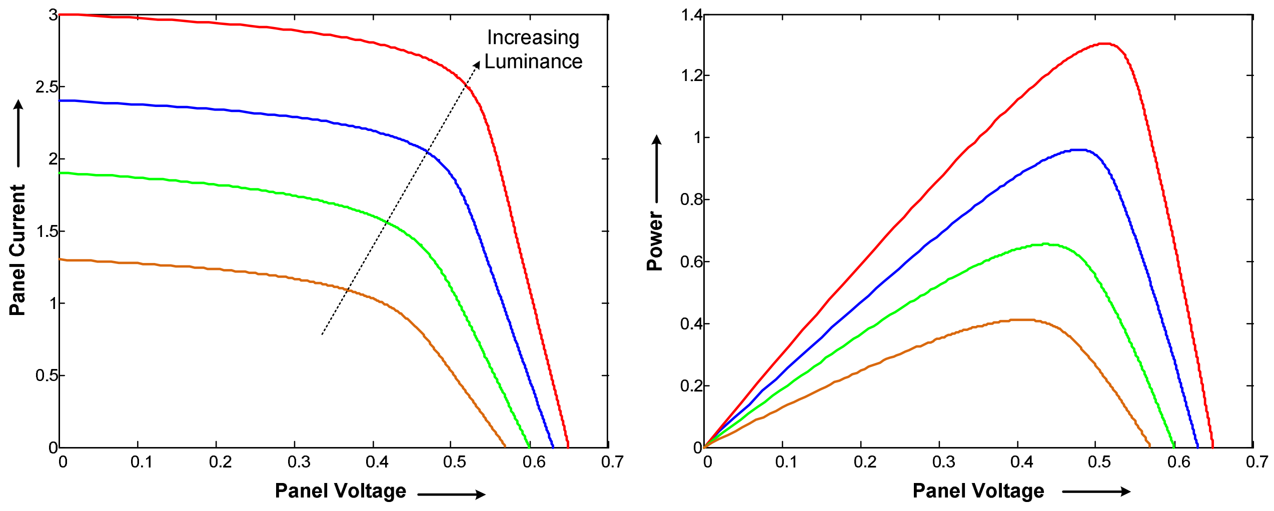
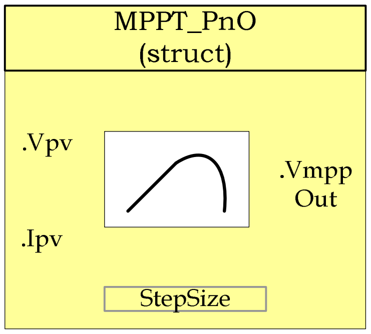


Figure 2 Solar Cell Characteristics

It is clear from the above V vs I curve that PV does not have a linear voltage and current relationship. Thus (P vs V) curve clearly shows a presence of a maximum. To get the most energy/utilization out of the PV system installation it must be operated at the maximum power point of this curve. The maximum power point however is not fixed due to the non linear nature of the PV –cell and changes with temperature, light intensity etc and varies from panel to panel. Thus different techniques are used to locate this maximum power point of the panel like Perturb and Observe, incremental conductance. The C2000 Solar library consists of blocks that can be used to track the MPP using well known MPP algorithms.

MPPT_PNO — *Perturb and Observe MPPT Algorithm Module*

Description: This software module implements the classical perturb and observe (P&O) algorithm for maximum power point tracking purposes.



Macro File: `mppt_pno.h`

Technical: Tracking for Maximum power point is an essential part of PV system implementation. Several MPP tracking methods have been implemented and documented for PV systems. This software module implements a very widely used MPP tracking method called “Perturb and Observe” algorithm. MPPT is achieved by regulating the Panel Voltage at the desired reference value. This reference is commanded by the MPPT P&O algorithm. The P&O algorithm keeps on incrementing and decrementing the panel voltage to observe power drawn change. First a perturbation to the panel reference is applied in one direction and power observed, if the power increases same direction is chosen for the next perturbation whereas if power decreases the perturbation direction is reversed. For example when operating on the left of the MPP (i.e. VpvRef < Vpv_mpp) increasing the VpvRef increases the power. Whereas when on the right of the MPP(VpvRef>Vpv_mpp) increasing the VpvRef decreases the power drawn from the panel. In Perturb and Observe (P&O) method the VpvRef is perturbed periodically until MPP is reached. The system then oscillates about the MPP. The oscillation can be minimized by reducing the perturbation step size. However, a smaller perturbation size slows down the MPPT in case of changing lighting conditions. Figure 3 illustrates the complete flowchart for the P&O MPPT algorithm

This module expects the following inputs:

1) Panel Voltage (Vpv): This is the sensed panel voltage signal sampled by ADC and ADC result converted to Q24 (or Global_Q) format.
2) Panel Current (Ipv): This is the sensed panel current signal sampled by ADC and ADC result converted to Q24 (or Global_Q) format.

3) Step Size (Stepsize): Size of the step used for changing the MPP voltage reference output, direction of change is determined by the slope calculation done in the MPPT algorithm.

Upon Macro call – Panel power (P(k)=V(k)*I(k)) is calculated, and is compared with the panel power obtained on the previous macro call. The direction of change in power determines the action on the voltage output reference generated. If current panel power is greater than previous power voltage reference is moved in the same direction, as earlier. If not, the voltage reference is moved in the reverse direction.

This module generates the following Outputs:

1) Voltage reference for MPP (VmppOut): Voltage reference for MPP tracking obtained by incremental conductance algorithm. Output in Q24 (Global_Q) format.

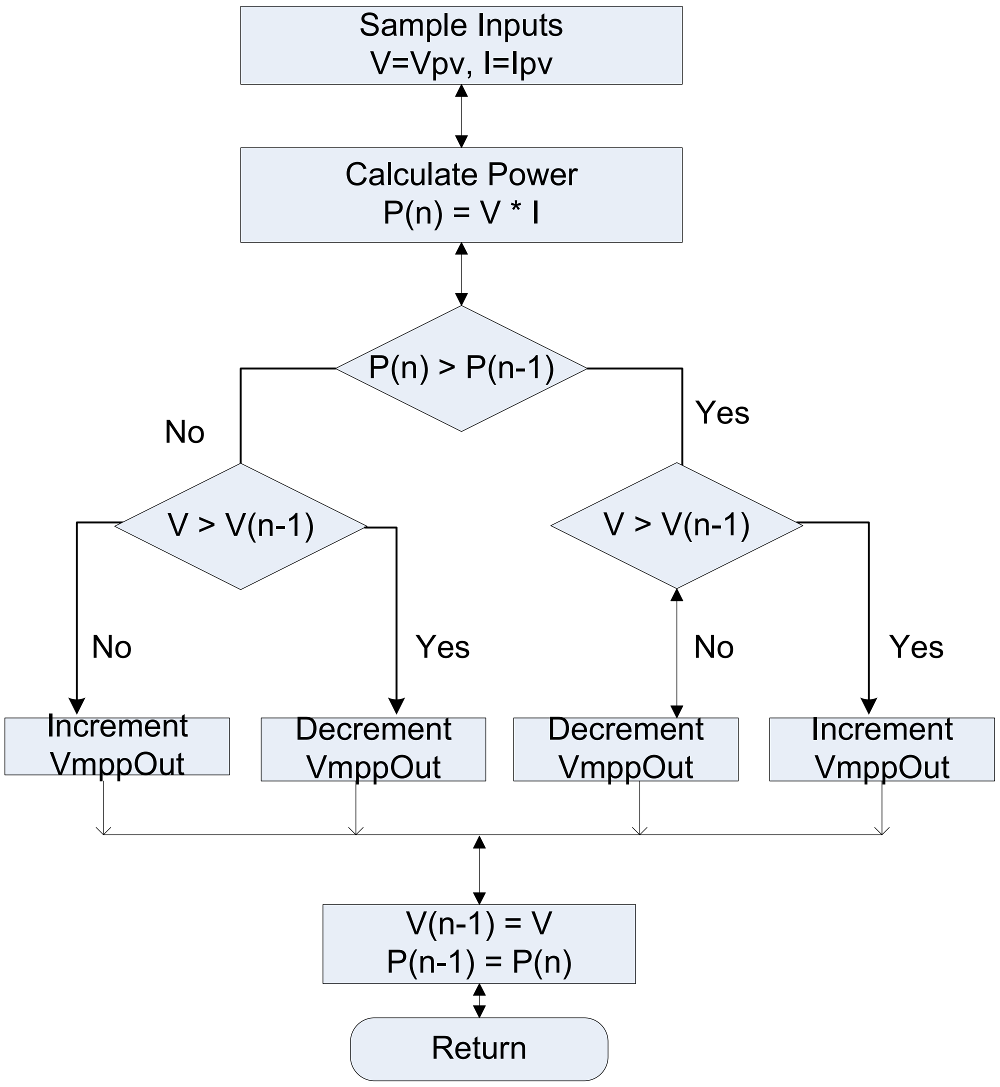


Figure 3 Perturb & Observe Algorithm Flowchart for MPPT

Object Definition:

```
typedef struct {          _iq  Ipv;
                          _iq  Vpv;
                          _iq  DeltaPmin;
                          _iq  MaxVolt;
                          _iq  MinVolt;
                          _iq  Stepsize;
                          _iq  VmppOut;
                          // internal variables
                          _iq  DeltaP;
                          _iq  PanelPower;
                          _iq  PanelPower_Prev;
                          Uint16 mppt_enable;
                          Uint16 mppt_first;
                          } mppt_pno;
```

Special Constants and Data types

mppt_pno
The module definition is created as a data type. This makes it convenient to instance an interface to the mppt_pno module. To create multiple instances of the module simply declare variables of type mppt_pno.

mppt_pno_DEFAULTS
Structure symbolic constant to initialize mppt_pno module. This provides the initial values to the terminal variables as well as method pointers.

Module interface Definition:

Module Element Name	Type	Description	Acceptable Range
Vpv	Input	Panel Voltage input	Q24 [0,1)
Ipv	Input	Panel Current input	Q24 [0,1)
StepSize	Input	Step size input used for changing reference MPP voltage output generated	Q24 [0,1)
DeltaPmin	Input	Threshold limit of power change for which perturbation takes place.	Q24
MaxVolt	Input	Upper Limit on the voltage reference value generated by MPPT algorithm – max value of VmppOut	Q24
MinVolt	Input	Lower Limit on the voltage reference value generated by MPPT algorithm – Min value of VmppOut	Q24
VmppOut	Output	MPPT output voltage reference generated	Q24

DeltaP	Internal	Change in Power	Q24
PanelPower	Internal	Latest Panel power calculated from Vpv and Ipv	Q24
PanelPower_prev	Internal	Previous value of Panel Power	Q24
mppt_enable	Internal	Flag to enable mppt computation – enabled by default	Uint16
mppt_first	Internal	Flag to indicate mppt macro is called for the first time. Used for setting initial values for vref.	Uint16

Usage: This section explains how to use this module.

Step 1 Add library header file in the file `{ProjectName}-Includes.h`

```
#include "mppt_pno.h"
```

Step 2 Creation of macro structure in C file `{ProjectName}-Main.c`

```
mppt_pno  mppt_pno1 = mppt_pno_DEFAULTS;
```

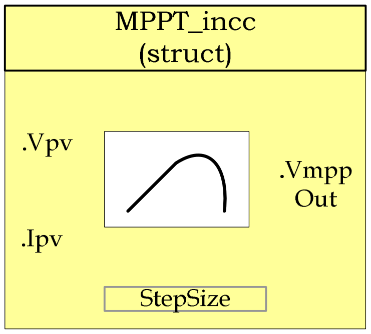
Step 4 Input initialization in C file `{ProjectName}-Main.c`

```
      //mppt pno
      mppt_pno1.DeltaPmin = _IQ(0.00001);
      mppt_pno1.MaxVolt = _IQ(0.9);
      mppt_pno1.MinVolt = _IQ(0.0);
      mppt_pno1.Stepsize = _IQ(0.005);
```

Step 5 Using the Macro in MPPT Task – MPPT is run at a slower rate generally, the MPPT macro is called after inputting the panel current and the panel voltage scaled values into the MPPT structure**.**

```
      // Write normalized panel current and voltage values
      // to the MPPT macro
      mppt_pno1.Ipv = IpvRead; \\ Normalized Panel Current
      mppt_pno1.Vpv = VpvRead; \\ Normalized Panel Voltage
      // Invoking the MPPT computation macro
      mppt_pno_MACRO (mppt_pno1);
      // Output of the MPPT macro can be written to the reference of
      // the voltage regulator
      Vpvref_mpptOut = mppt_pno1.VmppOut;
```

Description: This software module implemented the incremental conductance algorithm used for maximum power point tracking purposes.



Macro File: `mppt_incc.h`

Technical: Tracking for Maximum power point is an essential part of PV system implementation. Several MPP tracking methods have been implemented and documented in PV systems. This software module implements a very widely used MPP tracking method called "Incremental Conductance" algorithm. The incremental conductance (INCC) method is based on the fact that the slope of the PV array power curve is zero at the MPP, positive on the left of the MPP, and negative on the right.

$$\Delta I / \Delta V = -I / V \text{ , At MPP}$$

$$\Delta I / \Delta V < -I / V \text{ , Right of MPP}$$

$$\Delta I / \Delta V > -I / V \text{ , Left of MPP}$$

The MPP can thus be tracked by comparing the instantaneous conductance (I/V) to the incremental conductance ($\Delta I / \Delta V$) as shown in the flowchart in below. Vref is the reference voltage at which the PV array is forced to operate. At the MPP, Vref equals to V_{MPP} of the panel. Once the MPP is reached, the operation of the PV array is maintained at this point unless a change in ΔI is noted, indicating a change in atmospheric conditions and hence the new MPP. Figure 4 illustrates the flowchart for the incremental conductance method. The algorithm decrements or increments Vref to track the new MPP.

This module expects the following basic inputs:

1) Panel Voltage (Vpv): This is the sensed panel voltage signal sampled by ADC and ADC result converted to Q24 (or Global_Q) format.
2) Panel Current (Ipv): This is the sensed panel current signal sampled by ADC and ADC result converted to Q24 (or Global_Q) format.
3) Step Size (Stepsize): Size of the step used for changing the MPP voltage reference output, direction of change is determined by the slope calculation done in the MPPT algorithm.

The increment size determines how fast the MPP is tracked. Fast tracking can be achieved with bigger increments but the system might not operate exactly at the MPP and oscillate about it instead; so there is a tradeoff.

Upon Macro call – change in the Panel voltage and current inputs is calculated, conductance and incremental conductance are determined for the given operating conditions. As per the flowchart below – voltage reference for MPP tracing is generated based on the conductance and incremental conductance values calculated.

This module generates the following Outputs:

1) Voltage reference for MPP (VmppOut): Voltage reference for MPP tracking obtained by incremental conductance algorithm. Output in Q24 (Global_Q) format.

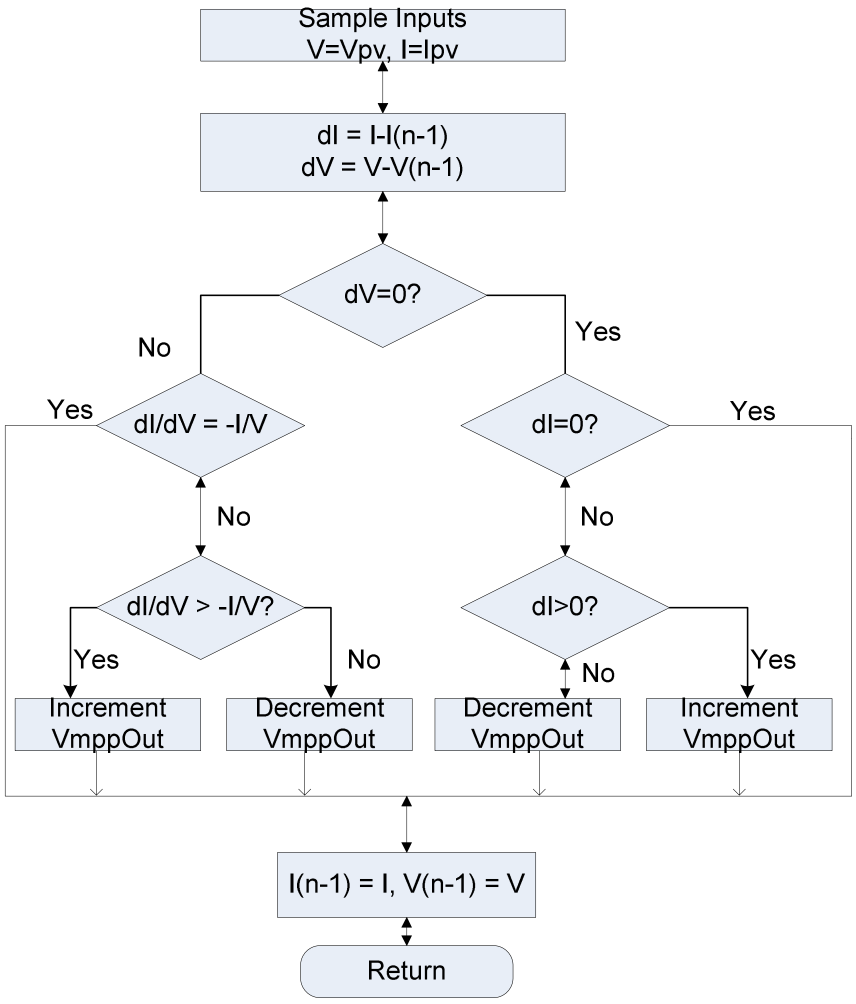


Figure 4 Incremental Conductance Method Flowchart

Object Definition:

```
typedef struct {        _iq  Ipv;
                        _iq  Vpv;
                        _iq  IpvH;
                        _iq  IpvL;
                        _iq  VpvH;
                        _iq  VpvL;
                        _iq  MaxVolt;
                        _iq  MinVolt;
                        _iq  Stepsize;
                        _iq  VmppOut;
                        // internal variables
                        _iq  Cond;
                        _iq  IncCond;
                        _iq  DeltaV;
                        _iq  DeltaI;
                        _iq  VpvOld;
                        _iq  IpvOld;
                        Uint16 mppt_enable;
                        Uint16 mppt_first;
                        } mppt_incc;
```

Special Constants and Data types

mppt_incc
The module definition is created as a data type. This makes it convenient to instance an interface to the mppt_incc module. To create multiple instances of the module simply declare variables of type mppt_incc.

mppt_incc_DEFAULTS
Structure symbolic constant to initialize mppt_incc module. This provides the initial values to the terminal variables as well as method pointers.

Module interface Definition:

Module Element Name	Type	Description	Acceptable Range
Vpv	Input	Panel Voltage input	Q24
Ipv	Input	Panel Current input	Q24
StepSize	Input	Step size input used for changing reference MPP voltage output generated	Q24
VpvH	Input	Threshold limit for change in voltage in +ve direction	Q24
VpvL	Input	Threshold limit for change in voltage in -ve direction	Q24
IpvH	Input	Threshold limit for change in Current in +ve direction	Q24

IpvL	Input	Threshold limit for change in Current in -ve direction	Q24
MaxVolt	Input	Upper Limit on the voltage reference value generated by MPPT algorithm – max value of VmppOut	Q24
MinVolt	Input	Lower Limit on the voltage reference value generated by MPPT algorithm – Min value of VmppOut	Q24
VmppOut	Output	MPPT output voltage reference generated	Q24
Cond	Internal	Conductance value calculated	Q24
IncCond	Internal	Incremental Conductance value calculated	Q24
DeltaV	Internal	Change in Voltage	Q24
DeltaI	Internal	Change in Current	Q24
VpvOld	Internal	Previous value of Vpv	Q24
IpvOld	Internal	Previous value of Ipv	Q24
mppt_enable	Internal	Flag to enable mppt computation – enabled by default	Uint16
mppt_first	Internal	Flag to indicate mppt macro is called for the first time. Used for setting initial values for vref.	Uint16

Usage: This section explains how to use this module.

Step 1 Add library header file in the file {ProjectName}-Includes.h

```
#include "mppt_incc.h"
```

Step 2 Creation of macro structure in C file {ProjectName}-Main.c

```
mppt_incc  mppt_incc1 = mppt_incc_DEFAULTS;
```

Step 4 Input initialization in C file {ProjectName}-Main.c

```
         // mppt incc
        mppt_incc1.IpvH = _IQ(0.0001);
        mppt_incc1.IpvL = _IQ(-0.0001);
        mppt_incc1.VpvH = _IQ(0.0001);
        mppt_incc1.VpvL = _IQ(-0.0001);
        mppt_incc1.MaxVolt = _IQ(0.9);
```

```
mppt_incc1.MinVolt = _IQ(0.0);
mppt_incc1.Stepsize = _IQ(0.005);
mppt_incc1.mppt_first=1;
mppt_incc1.mppt_enable=0;
```

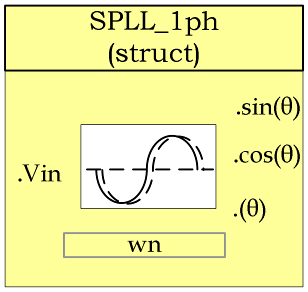
`Step 5` **Using the Macro in MPPT Task –** MPPT is run at a slower rate generally, the MPPT macro is called after inputting the panel current and the panel voltage scaled values into the MPPT structure**.**

```
// Write normalized panel current and voltage values

// to the MPPT macro

mppt_incc1.Ipv = IpvRead; \\ Normalized Panel Current

mppt_incc1.Vpv = VpvRead; \\ Normalized Panel Voltage

// Invoking the MPPT computation macro

mppt_incc_MACRO (mppt_incc1);

// Output of the MPPT macro can be written to the reference of

// the voltage regulator

Vpvref_mpptOut = mppt_incc1.VmppOut;
```

4.2. Phase Locked Loop Modules

SPLL_1ph ***Software Phase Lock Loop for Single Phase Grid Tied Systems***

Description: This software module implemented a software phase lock loop to calculate the instantaneous phase of a single phase grid. It also computed the sine and cosine values of the grid that are used in the closed loop control.



Macro File: `SPLL_1ph.h`

Technical: The phase angle of the utility is a critical piece of information for 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/phase of an external periodical signal using a feedback loop. The PLL is simply a servo system which 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 effects the performance of the control loop of 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 the Figure 5, which consists of a phase detect(PD), a loop filter(LPF) and a voltage controlled oscillator(VCO)

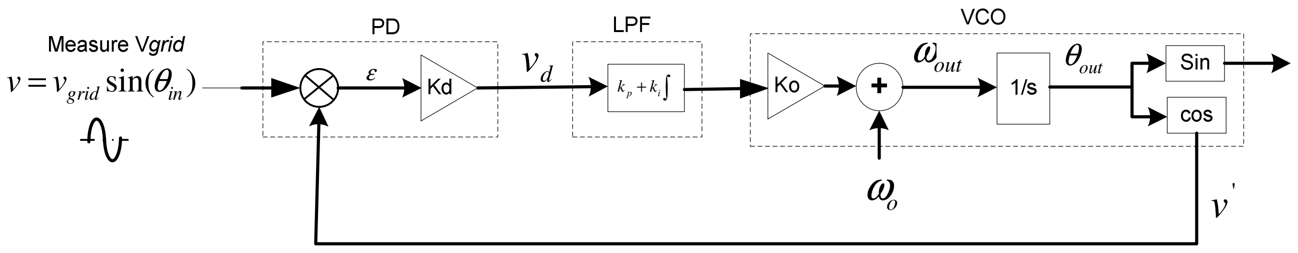


Figure 5 Phase Lock Loop Basic Structure

A sinusoidal measured value of the gird is given by,

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

Now let the VCO output be,

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

Phase Detect block multiplies the VCO output and the measured input value to get,

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

The output of PD block has information of the phase difference. However it has a high frequency component as well.

Thus the second block the loop filter, which is nothing but a PI controller is used which to low pass filter the high frequency components. Thus the output of the PI is

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

For steady state operation, ignore the $w_{grid} - w_{PLL}$ term, and $\sin(\theta) = \theta$ the linearized error is given as,

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

Small signal analysis is done using the network theory, where the feedback loop is broken to get the open loop transfer equation and then the closed loop transfer function is given by

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

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

The closed loop phase transfer function represents a low pass filter characteristics, which helps in attenuating the higher order harmonics. From the error transfer function it is clear that there are two poles at the origin which means that it is able to track even a constant slope ramp in the input phase angle without any steady state error.

Comparing the closed loop phase transfer function to the generic second order system transfer function

$$H\ (s) = \frac{2\xi\omega_n s + \omega_n^2}{s^2 + 2\xi\omega_n s + \omega_n^2}$$

Now comparing this with the closed loop phase transfer function, we can get the natural frequency and the damping ration of the linearalized PLL.

$$\omega_n = \sqrt{\frac{v_{grid} K_p}{T_i}}$$

$$\xi = \sqrt{\frac{v_{grid} T_i K_p}{4}}$$

Note in the PLL the PI serves dual purpose

1. To filter out high frequency which is at twice the frequency of the carrier/grid
2. Control response of the PLL to step changes in the grid conditions i.e. phase leaps, magnitude swells etc.

Now if the carrier is high enough in frequency, the low pass characteristics of the PI are good enough and one does not have to worry about low frequency passing characteristics of the LPF and only tune for the dynamic response of the PI. However as the grid frequency is very low (50Hz-60Hz) the roll off provided by the PI is not satisfactory enough and introduces high frequency element to the loop filter output which affects the performance of the PLL.

Therefore a notch filter is used at the output of the Phase Detect block which attenuates the twice the grid frequency component very well.

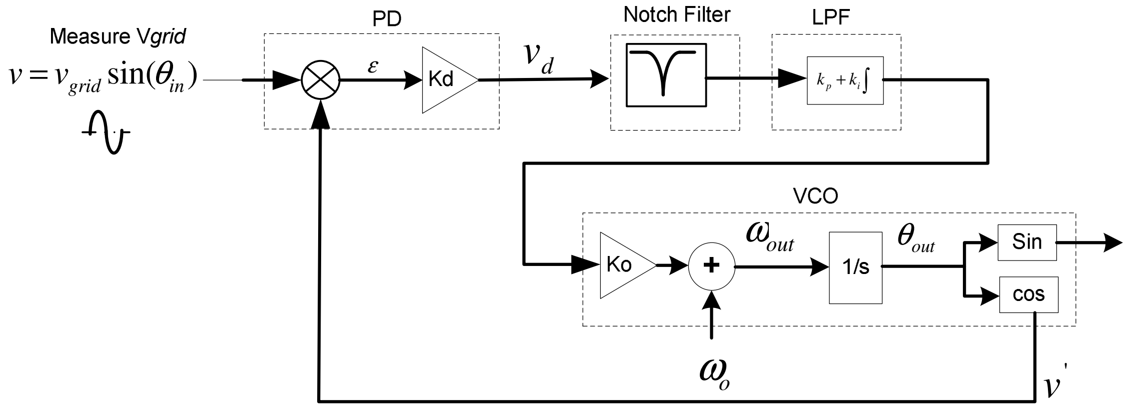


Figure 6 Single Phase PLL with Notch Filter

In this case the PI tuning can be done solely based on dynamic response of the PLL and not worry about the LPF characteristics.

The software module provides the structure for a software based PLL to be used in a single phase grid tied application using the method described in Figure 6.

The coefficients for lock to both 60Hz and 50Hz single phase grid are provided in the module.

Object Definition:

```
#define SPLL_Q _IQ21
#define SPLL_Qmpy _IQ21mpy

typedef struct{
      int32 B2_notch;
      int32 B1_notch;
      int32 B0_notch;
      int32 A2_notch;
      int32 A1_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;
```

The Q value of the PLL block can be specified independent of the global Q , that's why the module uses a SPLL_Q declaration for the IQ math operation instead of Q.

Special Constants and Data types

SPLL_1ph The module definition is created as a data type. This makes it convenient to instance an interface to the SPLL_1ph module. To create multiple instances of the module simply declare variables of type SPLL_1ph.

Module interface Definition:

Module Element Name	Type	Description	Acceptable Range
AC_input	Input	1ph AC Signal measured and normalized	Float(-1,1)
wn	Input	Grid Frequency in radians/sec	Float
theta[2]	Output	grid phase angle	Float
cos[2]	Output	Cos(grid phase angle)	Float
sin[2]	Output	Sin(grid phase angle)	Float
wo	Internal	Instantaneous Grid Frequency in radians/sec	Float
notch_coeff	Internal	Notch Filter Coefficients	Float
lpf_coeff	Internal	Loop Filer Coefficients	Float
Upd[3]	Internal	Internal Data Buffer for phase detect output	Float
ynotch[3]	Internal	Internal Data Buffer for the notch output	Float
ylf	Internal	Internal Data Buffer for Loop Filter output	Float
delta_t	Internal	1/Frequency of calling the PLL routine	Float

Usage:

This section explains how to use this module.

`Step 1` **Add library header file** `in the file {ProjectName}-Includes.h`

```
#include "SPLL_1ph.h"
```

`Step 2` **Creation of macro structure in C** file `{ProjectName}-Main.c`

```
// ------------- Software PLL for Grid Tie Applications ----------
SPLL_1ph spll1;
```

`Step 4` **Initialization in C** file `{ProjectName}-Main.c` , where the inputs to the initialization function are the grid frequency (50/60Hz), the inverter ISR period value and the address of the pll object. The routine initializes all the internal data buffers and variables, and sets the coefficients of the notch filter according to the grid frequency.

```
SPLL_1ph_init(60,_IQ21(0.00005),&spll1);
```

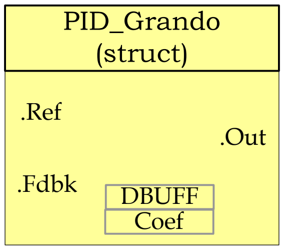
`Step 5` Using the SPLL macro in the Inverter ISR–

```
// SPLL call
SPLL_1ph_run(&spll1);
```

4.3. Controller Modules

PID Grando **PID regulator**

Description: This software module implemented the incremental conductance algorithm used for maximum power point tracking purposes.



Macro File: pid_grando.h

Technical: The PID_grando module implements a basic summing junction and PID control law with the following features:

- Programmable output saturation
- Independent reference weighting on proportional path
- Independent reference weighting on derivative path
- Anti-windup integrator reset
- Programmable derivative filter

All intput, output and internal data is in I8Q24 fixed-point format. A block diagram of the internal controller structure is shown in Figure 7.

The code is supplied as a C macro in a single header file named "`PID_grando.h`". The controller variables are grouped into three short C structures as follows.

1. **Terminals**

Ref	// Input: reference set-point
Fdb	// Input: feedback
Out	// Output: controller output
c1	// Internal: derivative filter coefficient
c2	// Internal: derivative filter coefficient

2. **Parameters**

Kr	// Parameter: proportional reference
Kp	// Parameter: proportional loop gain
Ki	// Parameter: integral gain
Kd	// Parameter: derivative gain
Km	// Parameter: derivative reference weighting
Umax	// Parameter: upper saturation limit
Umin	// Parameter: lower saturation limit

3. **Data**

up	// Data: proportional term
ui	// Data: integral term
ud	// Data: derivative term
v1	// Data: pre-saturated controller output
i1	// Data: integrator storage: ui(k-1)
d1	// Data: differentiator storage: ud(k-1)
d2	// Data: differentiator storage: d2(k-1)
w1	// Data: saturation record: [u(k-1) - v(k-1)]

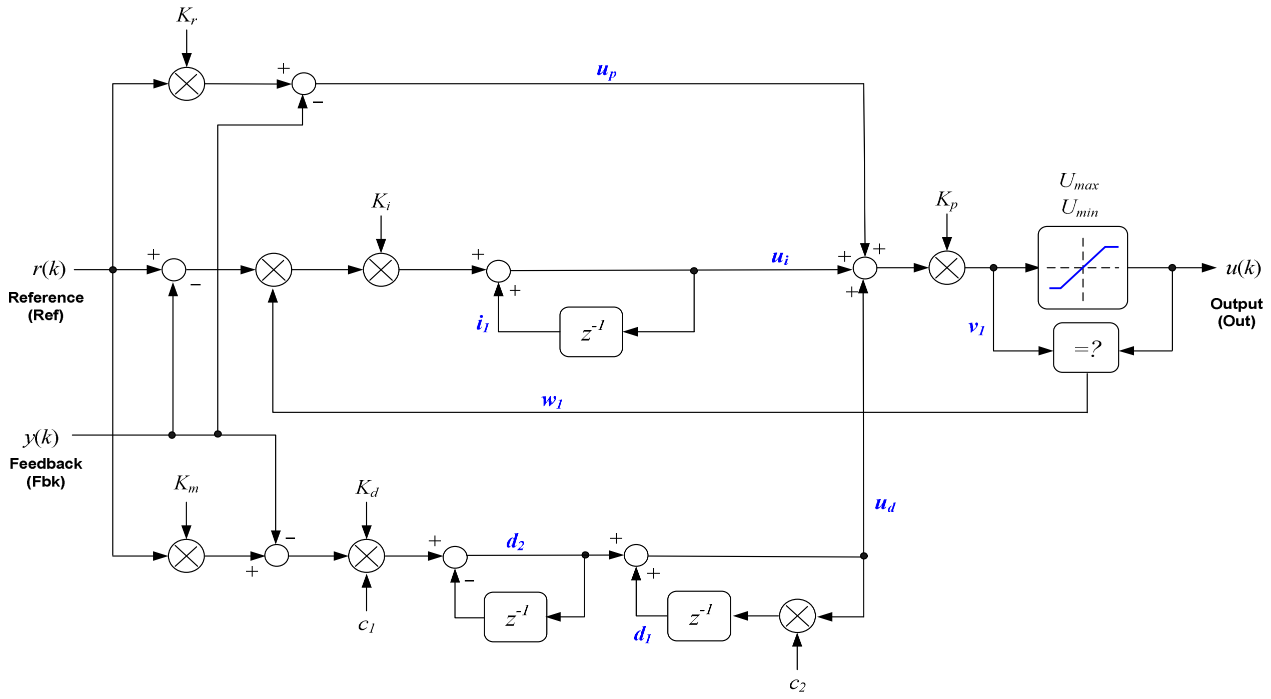


Figure 7 PID_Grando Controller structure

Technical description

a) *Proportional path*

The proportional term is taken as the difference between the reference and feedback terms. A feature of this controller is that sensitivity to the reference input can be weighted differently to the feedback path. This provides an extra degree of freedom when tuning the controller response to a dynamic input. The proportional law is:

$$u_p(k) = K_r r(k) - y(k) \quad \text{.......} \quad (1)$$

Note that "proportional" gain is applied to the sum of all three terms and will be described in section d).

b) *Integral path*

The integral path consists of a discrete integrator which is pre-multiplied by a term derived from the output module. The term w1 is either zero or one, and provides a means to disable the integrator path when output saturation occurs. This prevents the integral term from “winding up” and improves the response time on recovery from saturation. The integrator law used is based on a backwards approximation.

$$u_i(k) = u_i(k-1) + K_i\left[r(k) - y(k)\right] \quad (2)$$

c) <u>*Derivative path*</u>

The derivative term is a backwards approximation of the difference between the current and previous inputs. The input is the difference between the reference and feedback terms, and like the proportional term, the reference path can be weighted independently to provide an additional variable for tuning.

A first order digital filter is applied to the derivative term to reduce nose amplification at high frequencies. Filter cutoff frequency is determined by two coefficients (c1 & c2). The derivative law is shown below.

$$e(k) = K_m r(k) - y(k) \quad (3)$$

$$u_d(k) = K_d\left[c_2 u_i(k-1) + c_1 e(k) - c_1 e(k-1)\right] \quad (4)$$

Filter coefficients are based on the cut-off frequency (a) in Hz and sample period (T) in seconds as follows:

$$c_1 = a \quad (5)$$

$$c_2 = 1 - c_1 T \quad (6)$$

d) <u>*Output path*</u>

The output path contains a multiplying term (Kp) which acts on the sum of the three controller parts. The result is then saturated according to user programmable upper and lower limits to give the output term.

The pre-and post-saturated terms are compared to determine whether saturation has occurred, and if so, a zero or one term is produced which is used to disable the integral path (see above). The output path law is defined as follows.

$$v_1(k) = K_p\left[u_p(k) + u_i(k) + u_d(k)\right] \quad (7)$$

$$u(k) = \begin{cases} U_{\max} : v_1(k) > U_{\max} \\ U_{\min} : v_1(k) < U_{\min} \\ v_1(k) : U_{\min} < v_1(k) < U_{\max} \end{cases} \qquad (8)$$

$$w_1(k) = \begin{cases} 0 : v_1(k) \neq u(k) \\ 1 : v_1(k) = u(k) \end{cases} \qquad (9)$$

Tuning the controller

Default values for the controller coefficients are defined in the macro header file which apply unity gain through the proportional path, and disable both integral and derivative paths. A suggested general technique for tuning the controller is now described.

Steps 1-4 are based on tuning a transient produced either by a step load change or a set-point step change.

Step 1. Ensure integral and derivative gains are set to zero. Ensure also the reference weighting coefficients (K_r & K_m) are set to one.

Step 2. Gradually adjust proportional gain variable (K_p) while observing the step response to achieve optimum rise time and overshoot compromise.

Step 3. If necessary, gradually increase integral gain (K_i) to optimize the return of the steady state output to nominal. The controller will be very sensitive to this term and may become unstable so be sure to start with a very small number. Integral gain will result in an increase in overshoot and oscillation, so it may be necessary to slightly decrease the K_p term again to find the best balance. Note that if the integral gain is used then set to zero, a small residual term may persist in u_i.

Step 4. If the transient response exhibits excessive oscillation, this can sometimes be reduced by applying a small amount of derivative gain. To do this, first ensure the coefficients c_1 & c_2 are set to one and zero respectively. Next, slowly add a small amount of derivative gain (K_d).

Steps 5 & 6 only apply in the case of tuning a transient set-point. In the regulator case, or where the set-point is fixed and tuning is conducted against changing load conditions, they are not useful.

Step 5. Overshoot and oscillation following a set-point transient can sometimes be improved by lowering the reference weighting in the proportional path. To do this, gradually reduce the K_r term from its nominal unity value to optimize the transient. Note that this will change the loop sensitivity to the input reference, so the steady state condition will change unless integral gain is used.

Step 6. If derivative gain has been applied, transient response can often be improved by changing the reference weighting, in the same way as step 6 except that in the derivative case steady state should not be affected. Slowly reduce the K_m variable from it's nominal unity value to optimize overshoot and oscillation. Note that in many cases optimal performance is achieved with a reference weight of zero in the derivative path, meaning that the differential term acts on purely the output, with no contribution from the input reference.

The derivative path introduces a term which has a frequency dependent gain. At higher frequencies, this can cause noise amplification in the loop which may degrade servo performance. If this is the case, it is possible to filter the derivative term using a first order digital filter in the derivative path. Steps 7 & 8 describe the derivative filter.

Step 7. Select a filter roll-off frequency in radians/second. Use this in conjunction with the system sample period (T) to calculate the filter coefficients c_1 & c_2 (see equations 5 & 6).

Step 8. Note that the c_1 coefficient will change the derivative path gain, so adjust the value of K_d to compensate for the filter gain. Repeat steps 5 & 6 to optimize derivative path gain.

Object Definition:

```
typedef struct {
      _iq  Ref;   // Input: reference set-point
      _iq  Fbk;   // Input: feedback
      _iq  Out; // Output: controller output
      _iq  c1;  // Internal: derivative filter coefficient 1
      _iq  c2;  // Internal: derivative filter coefficient 2
      } PID_GRANDO_TERMINALS;

// note: c1 & c2 placed here to keep structure size under 8 words

typedef struct {
      _iq  Kr;    // Parameter: reference set-point weighting
      _iq  Kp;    // Parameter: proportional loop gain
      _iq  Ki;    // Parameter: integral gain
      _iq  Kd;    // Parameter: derivative gain
      _iq  Km;  // Parameter: derivative weighting
      _iq  Umax;// Parameter: upper saturation limit
      _iq  Umin;// Parameter: lower saturation limit
      } PID_GRANDO_PARAMETERS;

typedef struct {
      _iq  up;    // Data: proportional term
      _iq  ui;    // Data: integral term
      _iq  ud;    // Data: derivative term
      _iq  v1;    // Data: pre-saturated controller output
      _iq  i1;    // Data: integrator storage: ui(k-1)
      _iq  d1;    // Data: differentiator storage: ud(k-1)
      _iq  d2;    // Data: differentiator storage: d2(k-1)
      _iq  w1;    // Data: saturation record: [u(k-1) - v(k-1)]
      } PID_GRANDO_DATA;


typedef struct {  PID_GRANDO_TERMINALS    term;
                  PID_GRANDO_PARAMETERS   param;
                  PID_GRANDO_DATA         data;
            } PID_GRANDO_CONTROLLER;
```

Special Constants and Data types

PID_GRANDO_CONTROLLER

The module definition is created as a data type. This makes it convenient to instance an interface to the pid_grando module. To create multiple instances of the module simply declare variables of type pid_grando_controller.

PID_TERM_DEFAULTS, PID_PARAM_DEFAULTS, PID_DATA_DEFAULTS
Default values for initializing the PID structure

Usage: This section explains how to use this module.

Step 1 Add library header file in the file {ProjectName}-Includes.h

```
#include "pid_grandoincc.h"
```

Step 2 Creation of macro structure in C file {ProjectName}-Main.c

```
PID_GRANDO_CONTROLLER   pidGRANDO_Iinv = {PID_TERM_DEFAULTS,
                        PID_PARAM_DEFAULTS, PID_DATA_DEFAULTS};
```

Step 4 Input initialization in C file {ProjectName}-Main.c

```
      pidGRANDO_Iinv.param.Kp=0.8;
      pidGRANDO_Iinv.param.Ki=(0.15);
      pidGRANDO_Iinv.param.Kd=(0.0);
      pidGRANDO_Iinv.param.Kr=(1.0);
      pidGRANDO_Iinv.param.Umax=(1.0);
      pidGRANDO_Iinv.param.Umin=(-1.0);
```

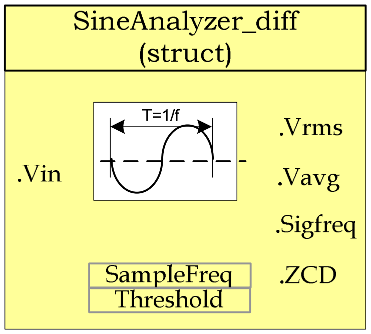
Step 5 Using the Macro in ISR Task

```
      // Using PID Grando Module

      pidGRANDO_Iinv.term.Fbk=inv_meas_cur_inst;

      pidGRANDO_Iinv.term.Ref=inv_ref_cur_inst;

      PID_GR_MACRO(pidGRANDO_Iinv);
```

4.4. Math Modules

SineAnalyzer_diff ***Computes rms and avg value of a sinusoidal signal***

Description: This software module analyzes the input sine wave and calculates several parameters like RMS, Average and Frequency.



Macro File: `SineAnalyzer_diff.h`

Technical: This module accumulates the sampled sine wave inputs, checks for threshold crossing point and calculates the RMS, Average values of the input sine wave. This module can also calculate the Frequency of the sine wave and indicate zero (or threshold) crossing point.

This module expects the following inputs:

2) Sine wave in Q15 format (Vin): This is the signal sampled by ADC and ADC result converted to Q15 format. This module expects a sine wave i.e. from -1 to 1.

3) Threshold Value (Threshold): Threshold value is used for detecting the cross over of the input signal across the threshold value set, in Q15 format. By default threshold is set to Zero.

4) Sampling Frequency (SampleFreq): This input should be set to the Frequency at which the input sine wave is sampled, in Q15 format and the sine analyzer block is called.

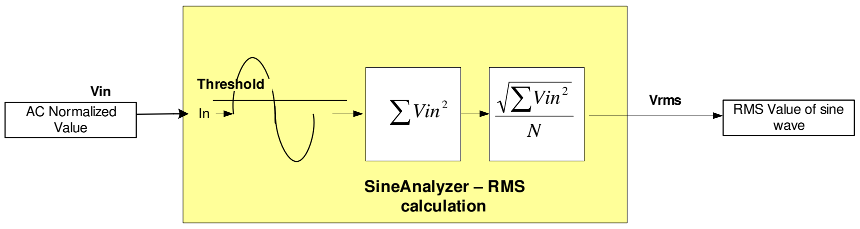
Upon Macro call – Input sine wave (Vin) is checked to see if the signal crossed over the threshold value. Once the cross over event happens, successive Vin samples are accumulated until occurrence of another threshold cross over point. Accumulated values are used for calculation of Average, RMS values of input signal. Module keeps track of number of samples between two threshold cross over points and this together with the signal sampling frequency (SampleFreq input) is used to calculate the frequency of the input sine wave.

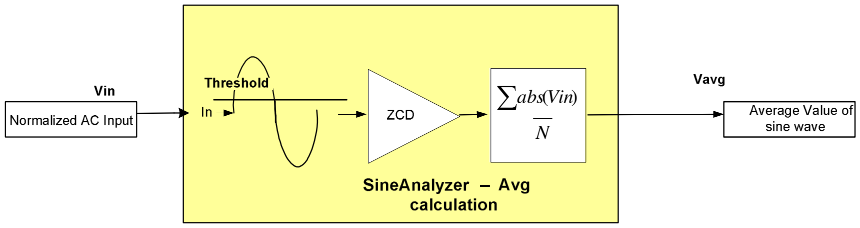
This module generates the following Outputs:

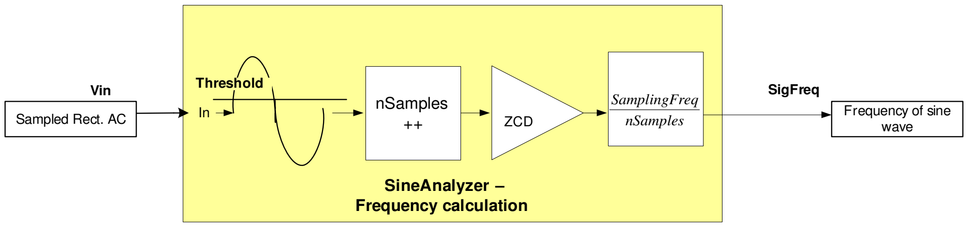
4) RMS value of sine wave (Vrms): Output reflects the RMS value of the sine wave input signal in Q15 format. RMS value is calculated and updated at every threshold crossover point.

5) Average value of sine wave (Vrms): Output reflects the Average value of the sine wave input signal in Q15 format. Average value is calculated and updated at every threshold crossover point.

6) Signal Frequency (SigFreq): Output reflects the Frequency of the sine wave input signal in Q15 format. Frequency is calculated and updated at every threshold crossover point.




Vin
Normalized AC Input
Threshold
In
ZCD
$\frac{\sum abs(Vin)}{N}$
Vavg
Average Value of sine wave
SineAnalyzer – Avg calculation



Object Definition:

```
typedef struct {
      _iq  Vin;// Input: Sine Signal
      _iq  SampleFreq; // Input: Signal Sampling Freq
      _iq  Threshold;// Input: Voltage level corresponding to zero i/p
      _iq  Vrms;   // Output: RMS Value
      _iq  Vavg;   // Output: Average Value
      _iq  SigFreq;// Output: Signal Freq
      Uint16  ZCD; // Output: Zero Cross detected
      // internal variables
      _iq15 Vacc_avg ;
      _iq15 Vacc_rms ;
      _iq15 curr_sample_norm; // normalized value of current sample
      Uint16 prev_sign ;
      Uint16 curr_sign ;
      Uint32 nsamples ; // samples in half cycle input waveform
      _iq15 inv_nsamples;
      _iq15 inv_sqrt_nsamples;
      } SineAnalyzer_diff;
```

Special Constants and Data types

SineAnalyzer_diff

The module definition is created as a data type. This makes it convenient to instance an interface to the Sine Analyzer module. To create multiple instances of the module simply declare variables of type SineAnalyzer.

SineAnalyzer _DEFAULTS

Structure symbolic constant to initialize SineAnalyzer module. This provides the initial values to the terminal variables as well as method pointers.

Module interface Definition:

Net name	**Type**	**Description**	**Acceptable Range**
Vin	Input	Sampled Sine Wave input	Q15
Threshold	Input	Threshold to be used for cross over detection	Q15
SampleFreq	Input	Frequency at which the Vin (input sine wave) is sampled, in Hz	Q15
Vrms	Output	RMS value of the sine wave input (Vin) updated at cross over point	Q15
Vavg	Output	Average value of the sine wave input (Vin) updated at cross over point	Q15
SigFreq	Output	Frequency of the sine wave input (Vin) updated at cross over point	Q15
ZCD	Output	When 'I' - indicates that Cross over happened and stays high till the next call of the macro.	Q15
Vacc_avg	Internal	Used for accumulation of samples for Average value calculation	Q15
Vacc_rms	Internal	Used for accumulation of squared samples for RMS value calculation	Q15
Nsamples	Internal	Number of samples between two crossover points	Int32
inv_nsamples	Internal	Inverse of nsamples	Q15
inv_sqrt_nsamples	Internal	Inverse square root of nsamples	Q15
Prev_sign, Curr_sign	Internal	Used for calculation of cross over detection	Int16

Usage: This section explains how to use this module.

Step 1 Add library header file in the file {ProjectName}-Includes.h

```
#include "SineAnalyzer_diff.h"
```

Step 2 Creation of macro structure in C file {ProjectName}-Main.c

```
// ------------- Sine Analyzer Block to measure RMS, frequency and ZCD
SineAnalyzer_diff sine_mainsV = SineAnalyzer_diff_DEFAULTS;
```

Step 4 Input initialization in C file {ProjectName}-Main.c

```
//sine analyzer initialization
sine_mainsV.Vin=0;
sine_mainsV.SampleFreq=_IQ15(20000.0);
sine_mainsV.Threshold=_IQ15(0.0);
```

Step 5 Calling the Macro in ISR

```
SineAnalyzer_diff_MACRO (sine_mainsV);
VrmsReal = _IQ15mpy (KvInv, sine_mainsV.Vrms);
```

Chapter 5. Revision History

Version	Date	Notes
V1.0	Jan, 31 2011	First Release of Solar Library