CLA C Solar Library

v1.1

Oct-12

Module User's Guide

C28x Foundation Software

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible or liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that products or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products.
www.ti.com/sc/docs/stdterms.htm

Mailing Address:
Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Trademarks

TMS320, C2000, Piccolo are the trademarks of Texas Instruments Incorporated.
All other trademarks mentioned herein are property of their respective companies

Acronyms

C28x: Refers to devices with the C28x CPU core.

CLA: Refers to the Control Law Accelerator this is present as a co-processor on some of the C2000 family of devices. Please refer to the device specific datasheet to find out whether CLA is present on a device.

Float: IEEE single precision floating point number

ClaToCpu_Volatile: This data type is primarily used for variables that are accessed across the CLA and the CPU, variables that reside in CLA-CPU message RAMs or CLA writable space.

Note: Use ClaToCpu_Volatile for variables that are CLA writable but are being monitored by the CPU. This will help the CLA C compiler optimize the code better.

Contents

Chapter 1. Introduction 5

1.1. Introduction 5

Chapter 2. Installing the CLA Solar Library 6

2.1. Solar Library Package Contents 6

2.2. How to Install the CLA Solar Library 6

Chapter 3. Module Summary 7

3.1. CLA Solar Library Function Summary 7

Chapter 4. Solar Lib Modules 8

4.1. Maximum Power Point Tracking (MPPT) 8

MPPT_PNO_CLA_C 9

MPPT_INCC_CLA_C 14

4.2. Phase Locked Loop Modules 19

SPLL_1ph_CLA_C 19

4.3. Controller Modules 25

PID_Grando_CLA_C 25

4.4. Math Modules 30

SineAnalyzer_Diff_CLA_C 30

Chapter 5. Revision History 35

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 a 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.

This document covers the Solar Library for Control Law Accelerator (CLA). The CLA supports a restricted syntax C compiler, thus the CLA C library takes these restrictions into account and provides ready blocks that can be used to construct a control system.

Chapter 2. Installing the CLA Solar Library

2.1. Solar Library Package Contents

The TI CLA 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 CLA Solar Library

The CLA 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
<base>\CLA_C	Contains implementation of solar library blocks on Control Law Accelerator using the CLA C Compiler
<base>\doc	Contains documentation for the library i.e. this file

Chapter 3. Module Summary

3.1. CLA Solar Library Function Summary

The CLA 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	Cycles with Code Optimization	Multiple Instance Support
MPPT_PNO_CLA_C	MPPT	Perturb and Observe MPPT Algorithm Module	~84	~61	Yes
MPPT_INCC_CLA_C	MPPT	Incremental Conductance MPPT Algorithm Module	~126	~129	Yes
SPLL_1ph_CLA_C	PLL	Software PLL for single phase grid connected application	~154	~108	Yes
PID_Grando_CLA_C	CNTL	PID module	~65	~62	Yes
SineAnalyzer_Diff_CLA_C	MATH	Calculate average and RMS of a sinusoidal signal	~122	~118	Yes

Note: The following compilation directives were used when compiling the CLA task file when profiling library modules with code optimization:

- Compiler optimization level 4 (-O 4)
- Optimize for space (-ms)

Chapter 4. Solar Lib Modules

4.1. Maximum Power Point Tracking (MPPT)

A simplistic model of a photo-voltaic (PV) cell is given by Figure 1

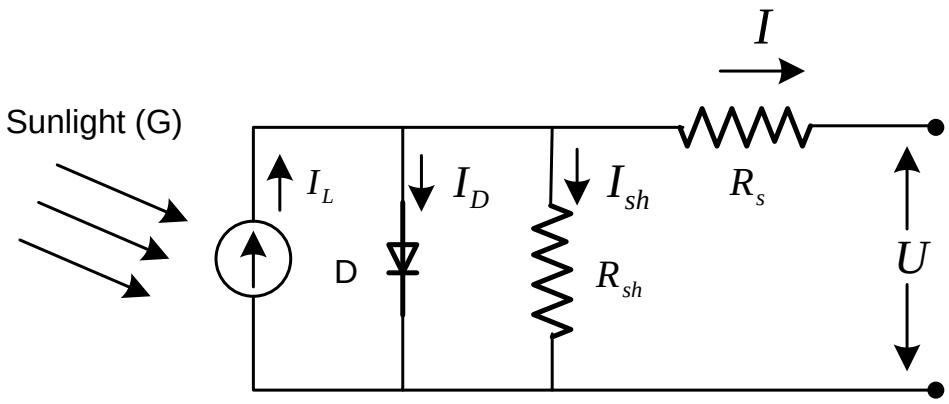


Figure 1 PV Cell Model

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

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

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

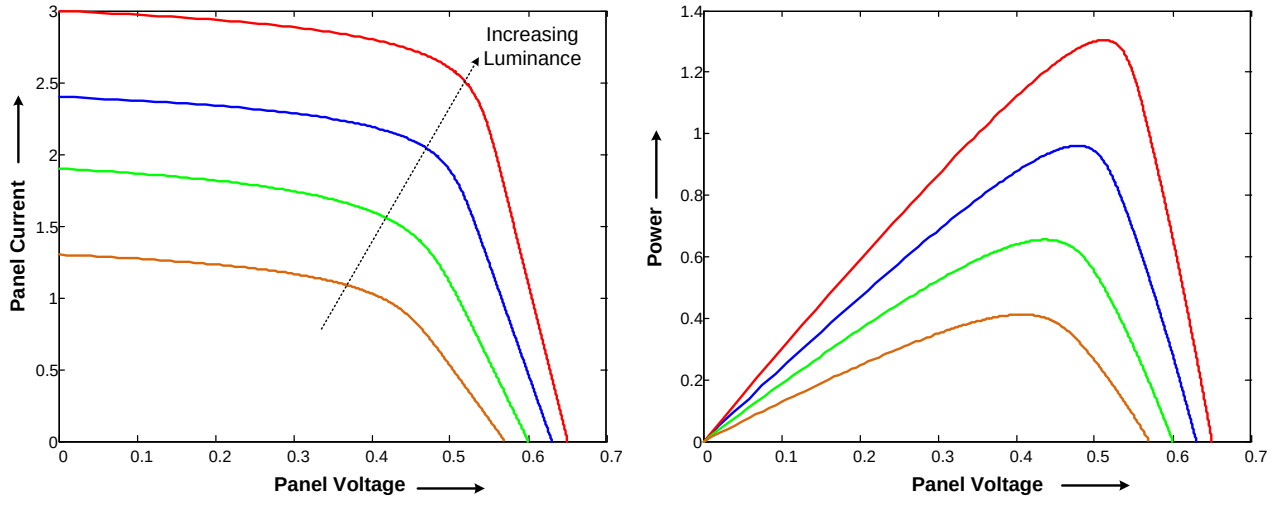


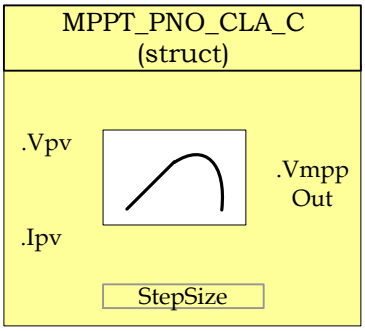
Figure 2 Solar Cell Characteristics

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 (MPP) of this curve. The maximum power point however is not fixed due to the non-linear

nature of the PV cell and changes due to temperature, light intensity, etc. 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 CLA Solar library consists of blocks that can be used to track the MPP using well known MPP algorithms.

MPPT_PNO_CLA_C	***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_CLA_C.h`

Technical: Tracking for MPP 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 normalized float format.
2) Panel Current (Ipv): This is the sensed panel current signal sampled by ADC and ADC result converted to normalized float 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 is in float format.

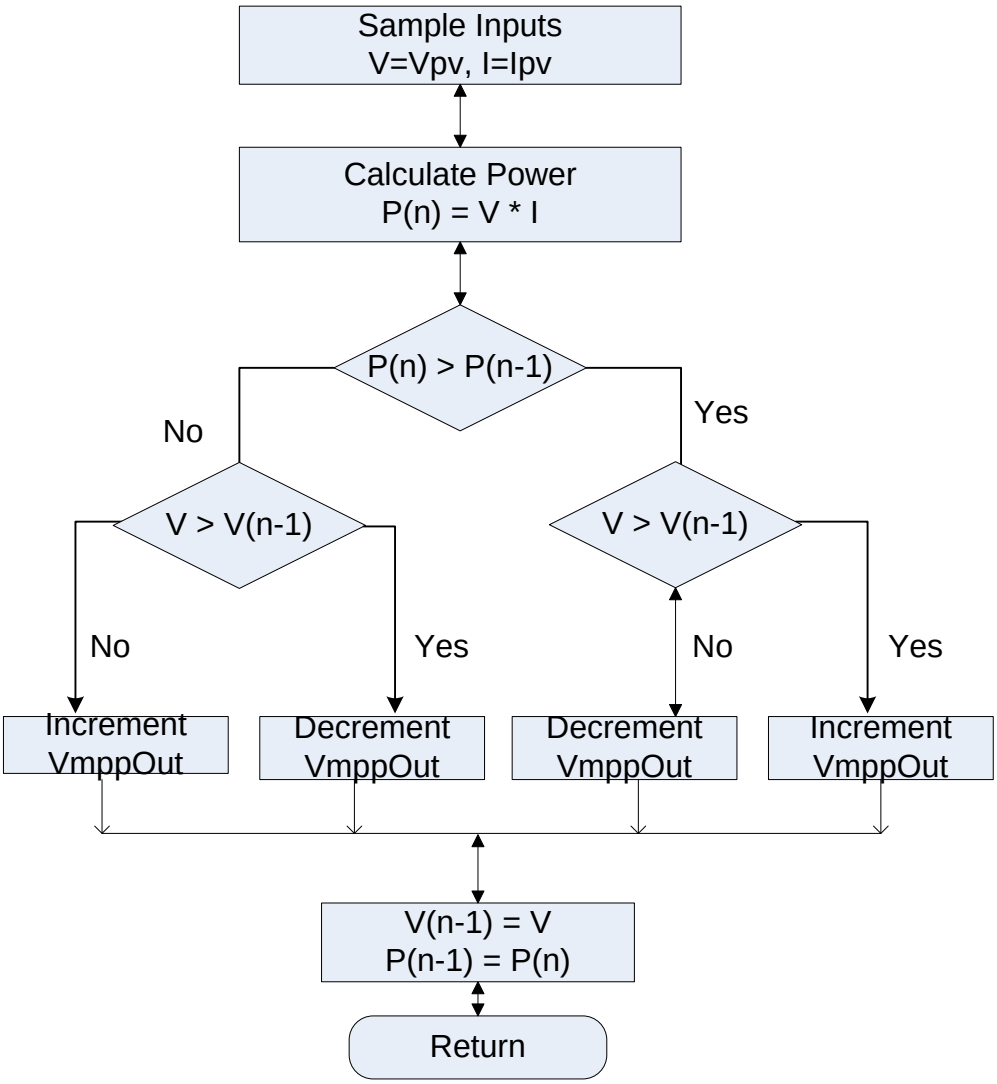


Figure 3 Perturb & Observe Algorithm Flowchart for MPPT

Object Definition:

```
typedef struct {
      float32 Ipv;
      float32 Vpv;
      float32 DeltaPmin;
      float32 MaxVolt;
      float32 MinVolt;
      float32 Stepsize;
      float32 VmppOut;

      // Internal variables
      float32 DeltaP;
      float32 PanelPower;
      float32 PanelPower_Prev;
      float32 MPPT_Enable;
      float32 MPPT_First;
} MPPT_PNO_CLA_C;
```

Special Constants and Data types

MPPT_PNO_CLA_C

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

Module interface definition:

Net name	Type	Description	Acceptable Range
Vpv	Input	Panel Voltage input	float32 [0,1)
Ipv	Input	Panel Current input	float32 [0,1)
Stepsize	Input	Step size input used for changing reference MPP voltage output generated	float32 [0,1)
DeltaPmin	Input	Threshold limit of power change for which perturbation takes place.	float32 [0,1)
MaxVolt	Input	Upper Limit on the voltage reference value generated by MPPT algorithm – max value of VmppOut	float32 [0,1)
MinVolt	Input	Lower Limit on the voltage reference value generated by MPPT algorithm – Min value of VmppOut	float32 [0,1)
VmppOut	Output	MPPT output voltage reference generated	float32 [0,1)
DeltaP	Internal	Change in Power	float32 (-1,1)
PanelPower	Internal	Latest Panel power calculated from Vpv and Ipv	float32 [0,1)
PanelPower_Prev	Internal	Previous value of Panel Power	float32 [0,1)
MPPT_Enable	Internal	Flag to enable MPPT computation – enabled by default	float32

MPPT_First	Internal	Flag to indicate MPPT macro is called for the first time. Used for setting initial values for vref.	float32

Usage: This section explains how to use this module.

Step 1 Add library header file to `{ProjectName}-CLA_Shared.h`

```
#include "MPPT_PNO_CLA_C.h"
```

Step 2 Creation of MPPT_PNO_CLA_C structure in `{ProjectName}-CLA_Tasks.cla` – Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(mppt_pno1, "Cla1ToCpuMsgRAM");
ClaToCpu_Volatile MPPT_PNO_CLA_C mppt_pno1;
```

If object needs to be accessed by the CPU, add the variable declaration to the {`ProjectName}-CLA_Shared.h` file:

```
extern ClaToCpu_Volatile MPPT_PNO_CLA_C mppt_pno1;
```

Note: Use ClaToCpu_Volatile for variables that are CLA writable but are being monitored by the CPU. This will help the CLA C compiler optimize the code better.

Step 3 Initialization in `{ProjectName}-Main.c` – Initialization can be achieved by MPPT_PNO_CLA_C_INIT macro which will set default values to the structure members:

```
MPPT_PNO_CLA_C_INIT(mppt_pno1);
```

Or values may be set individually as follows:

```
//mppt_pno1 macro initialization
mppt_pno1.DeltaPmin = 0.00001;
mppt_pno1.MaxVolt = 0.9;
mppt_pno1.MinVolt = 0.0;
mppt_pno1.Stepsize = 0.005;
mppt_pno1.MPPT_First = 1;
mppt_pno1.MPPT_Enable = 1;
```

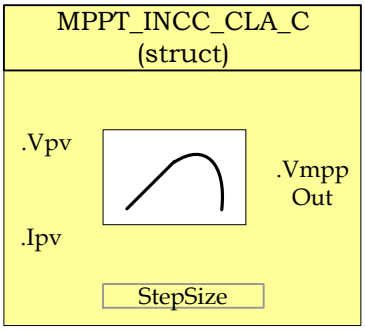
Step 4 Configure CLA memory in `{ProjectName}-Main.c` – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU:

```
// configure the RAM as CLA program memory
Cla1Regs.MMEMCFG.bit.PROGE = 1;
// configure RAM L1, F28035 as CLA Data memory 0
Cla1Regs.MMEMCFG.bit.RAM0E = 1;
// configure RAM L2, F28035 as CLA data memory 1
Cla1Regs.MMEMCFG.bit.RAM1E = 1;
```

Step 5 Using the Macro in CLA 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**.**

```
interrupt void Cla1Task1(void) {
        …
        // Write normalized panel current and voltage
        // values to MPPT object.
        mppt_pno1.Ipv = IpvRead;
        mppt_pno1.Vpv = VpvRead;

        // Invoke the MPPT macro
        MPPT_PNO_CLA_C(mppt_pno1);

        // Output of the MPPT macro can be written to
        // the reverence 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_CLA_C.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 proceeds to decrement or increment 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 normalized float format.
2) Panel Current (Ipv): This is the sensed panel current signal sampled by ADC and ADC result converted to normalized float 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 normalized float format.

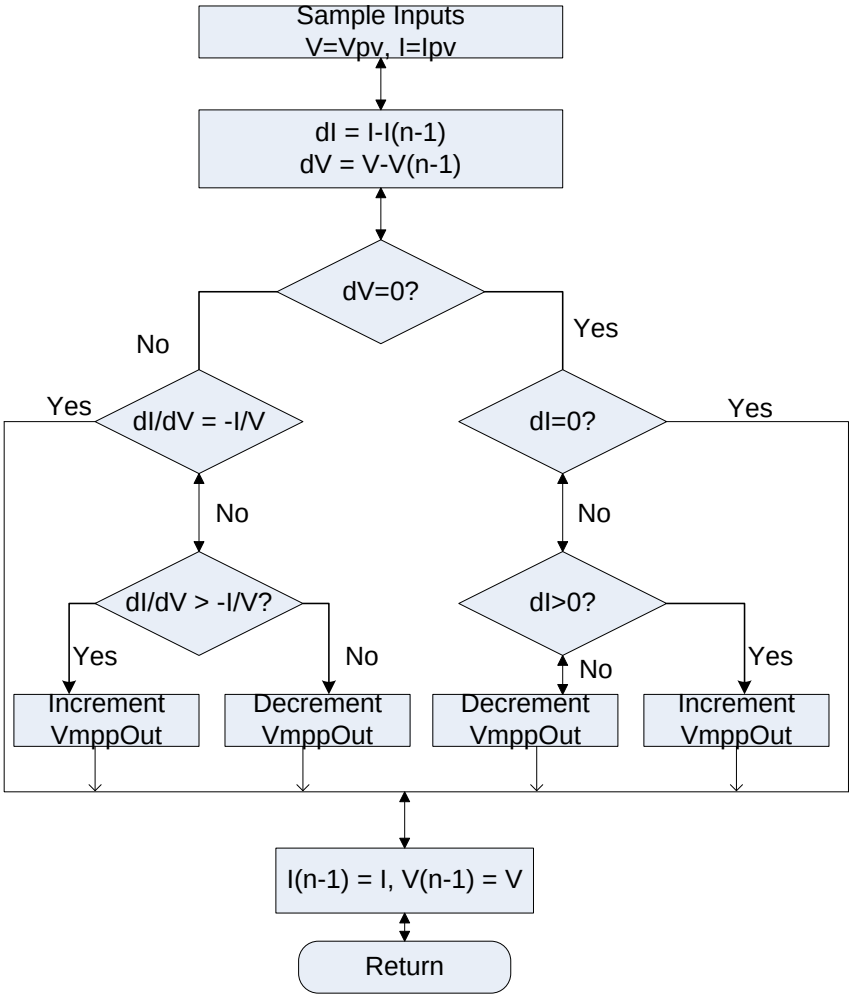


Figure 4 Incremental Conductance Method Flowchart

Object Definition:

```
typedef struct {
      float32 Ipv;
      float32 Vpv;
      float32 IpvH;
      float32 IpvL;
      float32 VpvH;
      float32 VpvL;
      float32 MaxVolt;
      float32 MinVolt;
      float32 Stepsize;
      float32 VmppOut;

      // Internal variables
      float32 Cond;
      float32 IncCond;
      float32 DeltaV;
      float32 DeltaI;
      float32 VpvOld;
      float32 IpvOld;
      float32 MPPT_Enable;
      float32 MPPT_First;
} MPPT_INCC_CLA_C;
```

Special Constants and Data types

MPPT_INCC_CLA_C

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

Module interface definition:

Net name	Type	Description	Acceptable Range
Vpv	Input	Panel Voltage input	float32 [0,1)
Ipv	Input	Panel Current input	float32 [0,1)
Stepsize	Input	Step size input used for changing reference MPP voltage output generated	float32 [0,1)
VpvH	Input	Threshold limit for change in voltage in +ve direction	float32 [0,1)
VpvL	Input	Threshold limit for change in voltage in -ve direction	float32 [0,1)
IpvH	Input	Threshold limit for change in Current in +ve direction	float32 [0,1)
IpvL	Input	Threshold limit for change in Current in -ve direction	float32 [0,1)
MaxVolt	Input	Upper Limit on the voltage reference value generated by MPPT algorithm – max value of VmppOut	float32 [0,1)

MinVolt	Input	Lower Limit on the voltage reference value generated by MPPT algorithm – Min value of VmppOut	float32 [0,1)
VmppOut	Output	MPPT output voltage reference generated	float32 [0,1)
Cond	Internal	Conductance value calculated	float32
IncCond	Internal	Incremental Conductance value calculated	float32
DeltaV	Internal	Change in Voltage	float32 [0,1)
DeltaI	Internal	Change in Current	float32 [0,1)
VpvOld	Internal	Previous value of Vpv	float32 [0,1)
IpvOld	Internal	Previous value of Ipv	float32 [0,1)
MPPT_Enable	Internal	Flag to enable MPPT computation – enabled by default	float32
MPPT_First	Internal	Flag to indicate MPPT macro is called for the first time. Used for setting initial values for vref.	float32

Usage: This section explains how to use this module.

Step 1 Add library header file to the file {ProjectName}-CLA_Shared.h

```
#include "MPPT_INCC_CLA_C.h"
```

Step 2 Creation of MPPT_INCC_CLA_C structure in {ProjectName}-CLA_Tasks.cla – Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(mppt_incc1, "Cla1ToCpuMsgRAM");
ClaToCpu_Volatile MPPT_INCC_CLA_C mppt_incc1;
```

If object needs to be accessed by the CPU, add the variable declaration to the {ProjectName}-CLA_Shared.h file:

```
extern ClaToCpu_Volatile MPPT_INCC_CLA_C mppt_INCC1;
```

Step 3 Initialization in {ProjectName}-Main.c - Initialization can be achieved by MPPT_INCC_CLA_C_INIT macro which will set default values to the structure members:

```
MPPT_INCC_CLA_C_INIT(mppt_incc1);
```

Or values may be set individually as follows:

```
//mppt incc macro initializations
mppt_incc1.IpvH = 0.0001;
mppt_incc1.IpvL = -0.0001;
mppt_incc1.VpvH = 0.0001;
mppt_incc1.VpvL = -0.0001;
```

```
mppt_incc1.MaxVolt = 0.9;
mppt_incc1.MinVolt = 0.0;
mppt_incc1.Stepsize = 0.005;
mppt_incc1.MPPT_First = 1;
mppt_incc1.MPPT_Enable = 1;
```

Step 4 Configure CLA memory in `{ProjectName}-Main.c` – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU:

```
// configure the RAM as CLA program memory
Cla1Regs.MMEMCFG.bit.PROGE = 1;
// configure RAM L1, F28035 as CLA Data memory 0
Cla1Regs.MMEMCFG.bit.RAM0E = 1;
// configure RAM L2, F28035 as CLA data memory 1
Cla1Regs.MMEMCFG.bit.RAM1E = 1;
```

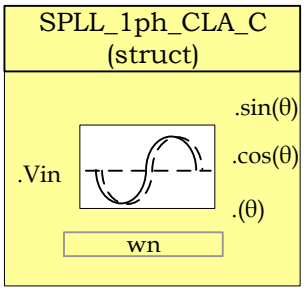
Step 5 Using the Macro in CLA Task – MPPT is run at a slower rate generally, the MPPT macro is called after inputting the panel current and the panel voltage scaled values into the MPPT structure**.**

```
interrupt void Cla1Task1(void) {
        …
        // Write normalized panel current and voltage
        // values to MPPT object.
        mppt_incc1.Ipv = IpvRead;
        mppt_incc1.Vpv = VpvRead;

        // Invoke the MPPT macro
        MPPT_INCC_CLA_C(mppt_pno1);

        // Output of the MPPT macro can be written to
        // the reverence of the voltage regulator
        Vpvref_mpptOut = mppt_INCC1.VmppOut;
        …
}
```

4.2. Phase Locked Loop Modules

SPLL_1ph_CLA_C	*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_CLA_C.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 minimal. The quality of the lock directly affects 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).

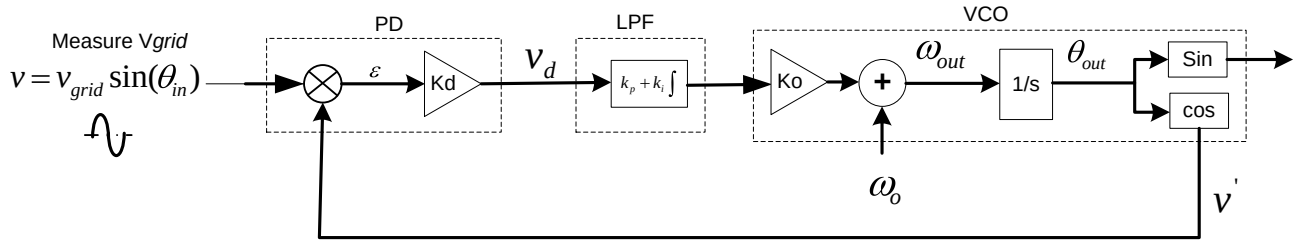


Figure 5 Phase Lock Loop Basic Structure

A sinusoidal measured value of the grid 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+ Open Loop TF)

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 linearized 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.

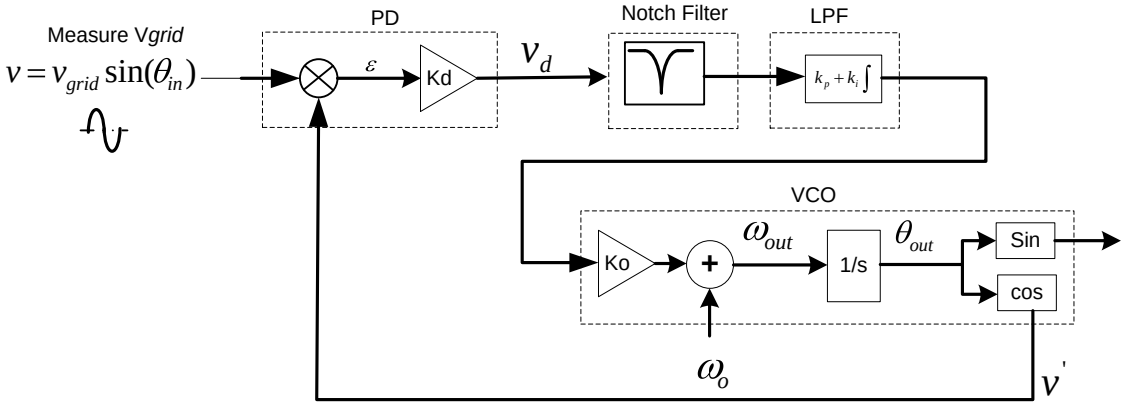


Figure 6 Single Phase PLL with Notch Filter

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:

```
typedef struct {
      float32 B2_notch;
      float32 B1_notch;
      float32 B0_notch;
      float32 A2_notch;
      float32 A1_notch;
} SPLL_NOTCH_COEFF;

typedef struct {
      float32 B1_lf;
      float32 B0_lf;
      float32 A1_lf;
} SPLL_LPF_COEFF;

typedef struct {
      // Inputs
      float32 AC_input;   // 1ph AC Signal measured
      float32 wn;         // Grid Frequency in cycles/s

      // Outputs
      float32 theta0;     // Grid phase angle
      float32 theta1;
      float32 cos0;       // COS (grid phase angle)
      float32 cos1;
      float32 sin0;             // SIN (grid phase angle)
      float32 sin1;

      // Internal
      float32 wo;               // Instantaneous Grid Frequency in cycles/s
      SPLL_NOTCH_COEFF notch_coeff;    // Notch Filter Coefficients
      SPLL_LPF_COEFF lpf_coeff;        // Loop Filer Coefficients
      float32 Upd0;       // Internal Data Buffer for phase detect output
      float32 Upd1;
      float32 Upd2;
      float32 ynotch0;    // Internal Data Buffer for the notch output
      float32 ynotch1;
      float32 ynotch2;
      float32 ylf0;       // Internal Data Buffer for Loop Filter output
      float32 ylf1;
      float32 delta_t;    // 1/Frequency of calling the PLL routine
} SPLL_1ph_CLA_C;
```

Special Constants and Data types

SPLL_1ph_CLA_C The module definition is created as a data type. This makes it convenient to instance an interface to the SPLL_1ph_CLA_C module. To create multiple instances of the module simply declare variables of type SPLL_1ph_CLA_C.

Module interface definition:

Net name	Type	Description	Acceptable Range
AC_input	Input	1ph AC Signal measured and normalized	float32 (-1,1)
wn	Input	Grid Frequency in radians/sec	float32
theta	Output	grid phase angle	float32 (-2*pi, 2*pi)
cos	Output	Cos(grid phase angle)	float32 (-1,1)
sin	Output	Sin(grid phase angle)	float32 (-1,1)
wo	Internal	Instantaneous Grid Frequency in radians/sec	float32
notch_coeff	Internal	Notch Filter Coefficients	float32
lpf_coeff	Internal	Loop Filer Coefficients	float32
Upd	Internal	Internal Data Buffer for phase detect output	float32
ynotch	Internal	Internal Data Buffer for the notch output	float32
ylf	Internal	Internal Data Buffer for Loop Filter output	float32
delta_t	Internal	1/Frequency of calling the PLL routine	float32

Usage: This section explains how to use this module.

Step 1 Add library header file to the file {ProjectName}-CLA_Shared.h

```
#include "SPLL_1ph_CLA_C.h"
```

Step 2 Creation of SPLL_1ph_CLA_C structure in {ProjectName}-CLA_Tasks.cla – Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(spll1, "Cla1ToCpuMsgRAM");
ClaToCpu_Volatile SPLL_1ph_CLA_C spll1;
```

If object needs to be accessed by the CPU, add the variable declaration to the {ProjectName}-CLA_Shared.h file:

```
extern ClaToCpu_Volatile SPLL_1ph_CLA_C spll1;
```

Step 3 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_CLA_C_INIT(spll1, 50, (0.00005));
```

Step 4 Configure CLA memory in `{ProjectName}-Main.c` – Assign memory to CLA. Message RAM and data RAM must be configured by the C28x:

```
// configure the RAM as CLA program memory
Cla1Regs.MMEMCFG.bit.PROGE = 1;
// configure RAM L1, F28035 as CLA Data memory 0
Cla1Regs.MMEMCFG.bit.RAM0E = 1;
// configure RAM L2, F28035 as CLA data memory 1
Cla1Regs.MMEMCFG.bit.RAM1E = 1;
```

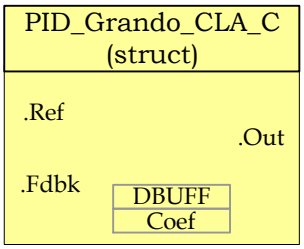
Step 5 Using the SPLL macro in the Inverter ISR– MPPT is run at a slower rate generally, the MPPT macro is called after inputting the panel current and the panel voltage scaled values into the MPPT structure**.**

```
interrupt void Cla1Task1(void) {
      …
      // SPLL call
      spll1.AC_input = Vac_in;
      SPLL_1ph_CLA_C(spll1);
      InvSine = (spll1.sin0);

      …
}
```

4.3. Controller Modules

PID_Grando_CLA_C	PID regulator

Description: This software implements a basic summing junction PID control algorithm.



Macro File: PID_Grando_CLA_C.h

Technical: The PID_Grando_CLA_C 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 input, output and internal data is in normalized floating point value. 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_CLA_C.h. The controller variables are grouped into three short C structures as follows.

Terminals:

Inputs:
- Ref - Reference set-point
- Fdk - Feedback

Outputs:
- Out - Controller output

Internal:
- c1 – First derivative filter coefficient
- c2 – Second derivative filter coefficient

Parameters:

- Kr - Proportional reference
- Kp - Proportional loop gain
- Ki - Integral gain
- Kd - Derivative gain
- Km - Derivative reference weighting
- Umax - Upper saturation limit
- Umin - Lower saturation limit

Data

up - Proportional term
ui - Integral term
ud - Derivative term
v1 - Pre-saturated controller output
i1 - Integrator storage: ui(k-1)
d1 - Differentiator storage: ud(k-1)
d2 - Differentiator storage: d2(k-1)
w1 - Saturation record: [u(k-1) - v(k-1)]

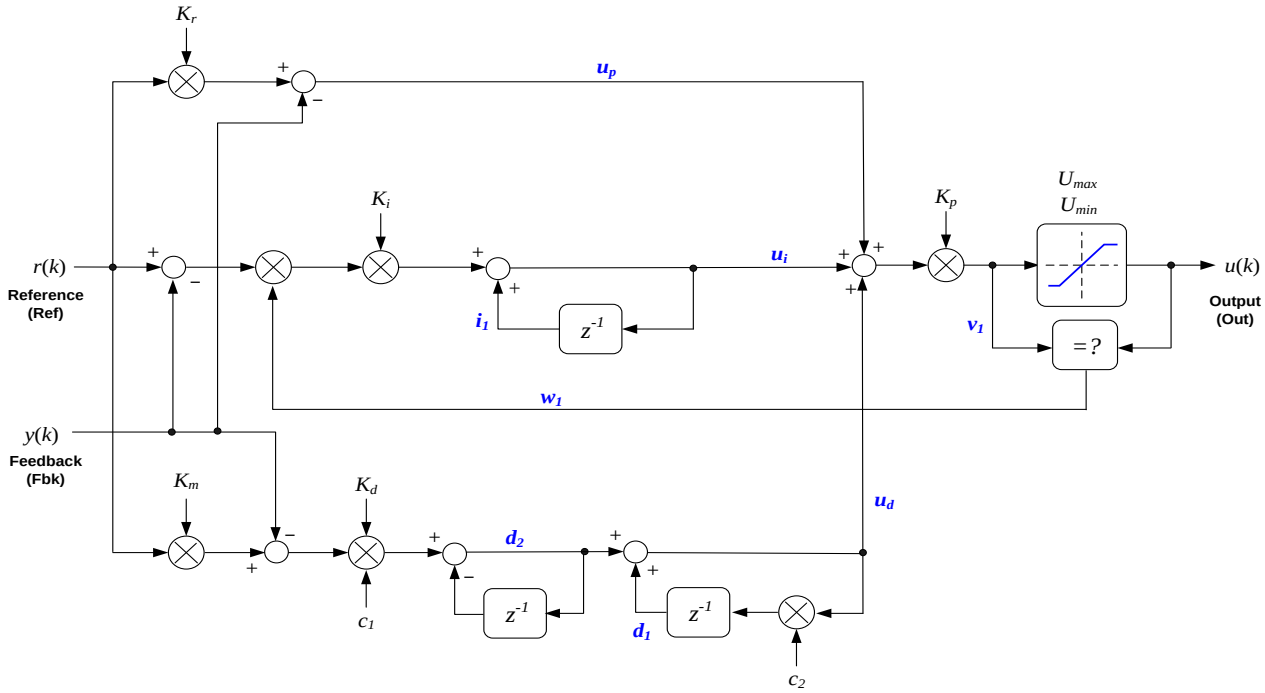


Figure 7 PID Grando Internal

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:

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.

c) *Derivative path*

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.

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

d) *Output path*

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.

Object Definition:

```
typedef struct {
      // Inputs
      float32 Ref;        // Reference set-point
      float32 Fbk;        // Feedback

      // Output
      float32 Out;        // Controller output

      // Internal
      float32 c1;         // Derivative filter coefficient 1
      float32 c2;         // Derivative filter coefficient 2
} PID_GRANDO_TERMINALS;

typedef struct {
      float32 Kr;         // Reference set-point weighting
      float32 Kp;         // Proportional loop gain
      float32 Ki;         // Integral gain
```

```
        float32 Kd;         // Derivative gain
        float32 Km;         // Derivative weighting
        float32 Umax;       // Upper saturation limit
        float32 Umin;       // Lower saturation limit
} PID_GRANDO_PARAMETERS;

typedef struct {
        float32 up;         // Proportional term
        float32 ui;         // Integral term
        float32 ud;         // Derivative term
        float32 v1;         // Pre-saturated controller output
        float32 i1;         // Integrator storage: ui(k-1)
        float32 d1;         // Differentiator storage: ud(k-1)
        float32 d2;         // Differentiator storage: d2(k-1)
        float32 w1;         // 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_CNTLR_CLA_C;
```

Special Constants and Data types

PID_GRANDO_CNTLR_CLA_C

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

Usage: This section explains how to use this module.

Step 1 Add library header file to the file {ProjectName}-CLA_Shared.h

```
#include "PID_Grando_CLA_C.h"
```

Step 2 Creation of PID_GRANDO_CNTLR_CLA_C structure in C file {ProjectName}-CLA_Tasks.cla – Declare the variable and specify and appropriate location in CLA memory.

```
#pragma DATA_SECTION(pidGRANDO_Iinv, "Cla1ToCpuMsgRAM");
ClaToCpu_Volatile PID_GRANDO_CNTLR_CLA_C pidGRANDO_Iinv;
```

If object needs to be accessed by the CPU, add the variable declaration to the {ProjectName}-CLA_Shared.h file:

```
extern ClaToCpu_Volatile PID_GRANDO_CNTLR_CLA_C pidGRANDO_Iinv;
```

Step 3 Initialization in {ProjectName}-Main.c – Initialization can be achieved by PID_GRANDO_CNTLR_CLA_C_INIT macro which will set default values to the structure members:

```
PID_GRAND__CNTLR_CLA_C_INIT(pidGRANDO_Iinv);
```

Or values may be set individually as follows:

```
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 4 Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU:

```
// configure the RAM as CLA program memory
Cla1Regs.MMEMCFG.bit.PROGE = 1;
// configure RAM L1, F28035 as CLA Data memory 0
Cla1Regs.MMEMCFG.bit.RAM0E = 1;
// configure RAM L2, F28035 as CLA data memory 1
Cla1Regs.MMEMCFG.bit.RAM1E = 1;
```

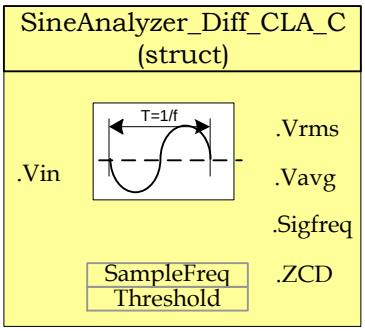
Step 5 Using the Macro in CLA Task

```
interrupt void Cla1Task1(void) {
      …
      // Using PID Grando Module
      pidGRANDO_Iinv.term.Ref = inv_meas_cur_inst;
      pidGRANDO_Iinv.term.Fbk = inv_ref_cur_inst;
      PID_GR_MACRO(pidGRANDO_Iinv);
      …
}
```

4.4. Math Modules

SineAnalyzer_Diff_CLA_C	*Computes RMS and average value of a sinusoidal*

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



Macro File: `SineAnalyzer_Diff_CLA_C.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 (Vin): This is the signal sampled by ADC and ADC result converted to normalized float format with offset applied for the swing fo the signal to be from -1 to 1 for full scale ADC reading.

3) Threshold Value (Threshold): Threshold value is used for detecting the crossover of the input signal across the threshold value set, in float 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, and the sine analyzer block is called.

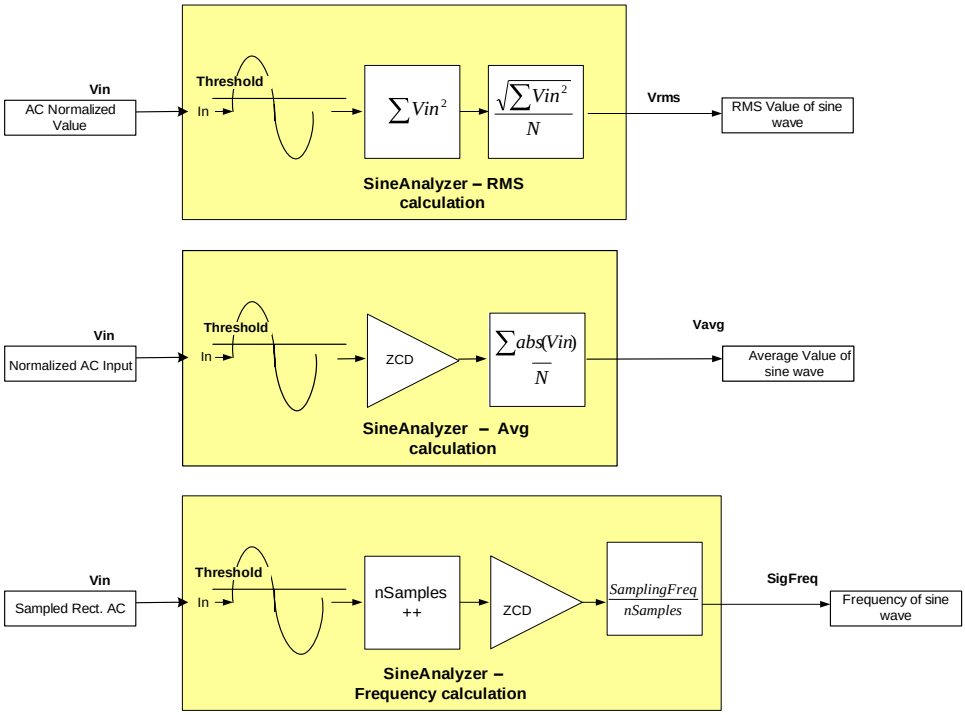
Upon Macro call – Input sine wave (Vin) is checked to see if the signal crossed over the threshold value. Once the cross over event happens, successive Vin samples are accumulated until occurrence of another threshold cross over point. Accumulated values are used for calculation of Average, RMS values of input signal. Module keeps track of number of samples between two threshold crossover 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. 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. 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. Frequency is calculated and updated at every threshold crossover point.



Object Definition:

```
typedef struct {
      // Inputs
      float32 Vin;              // Sine signal
      float32 SampleFreq;       // Signal sampling frequency
      float32 Threshold;        // Voltage level corresponding to zero i/p

      // Outputs
      float32 Vrms;             // RMS value
      float32 Vavg;             // Average value
      float32 SigFreq;          // Signal frequency
      float32 ZCD;              // Zero Cross detected

      // Internal variables
      float32 Vacc_avg;
      float32 Vacc_rms;
      float32 curr_sample_norm; // Normalized value of current sample
      float32 prev_sign;
      float32 curr_sign;
      Uint32 nsamples;          // Samples in half cycle input waveform
      float32 inv_nsamples;
      float32 inv_sqrt_nsamples;
} SineAnalyzer_Diff_CLA_C;
```

Special Constants and Data types

SineAnalyzer_Diff_CLA_C

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.

Module interface definition:

Net name	Type	Description	Acceptable Range
Vin	Input	Sampled Sine Wave input	float32 (-1,1)
Threshold	Input	Threshold to be used for cross over detection	float32 (-1,1)
SampleFreq	Input	Frequency at which the Vin (input sine wave) is sampled, in Hz	float32
Vrms	Output	RMS value of the sine wave input (Vin) updated at cross over point	float32
Vavg	Output	Average value of the sine wave input (Vin) updated at cross over point	float32
SigFreq	Output	Frequency of the sine wave input (Vin) updated at cross over point	float32
ZCD	Output	When 'I' - indicates that Cross over happened and stays high till the next call of the macro.	float32
Vacc_avg	Internal	Used for accumulation of samples for Average value calculation	float32
Vacc_rms	Internal	Used for accumulation of squared samples for RMS value calculation	float32
nsamples	Internal	Number of samples between two crossover points	Uint32
inv_nsamples	Internal	Inverse of nsamples	float32
inv_sqrt_nsamples	Internal	Inverse square root of nsamples	float32
prev_sign, curr_sign	Internal	Used for calculation of cross over detection	float32

Usage: This section explains how to use this module.

Step 1 Add library header file to the file {ProjectName}-CLA_Shared.h.

```
#include "SineAnalyzer_Diff_CLA_C.h"
```

The CLAmath library is included by this module and needs to be manually added to the project. Please use version 400 or higher for this module to work appropriately.

The library can be found at:
controlSUITE\libs\math\CLAmath\v400\lib\CLAmath.lib

Step 2 Creation of SineAnalyzer_Diff_CLA_C structure in {ProjectName}-CLA_Tasks.cla – Declare the variable and specify an appropriate location in CLA memory.

```
// Sine Analyzer Block to measure RMS, frequency and ZCD
#pragma DATA_SECTION(sine_mainsV, "Cla1ToCpuMsgRAM");
ClaToCpu_Volatile SineAnalyzer_Diff_CLA_C sine_mainsV;
```

If object needs to be accessed by the C28x, add the variable declaration to the {ProjectName}-CLA_Shared.h file:

```
extern ClaToCpu_Volatile SineAnalyzer_Diff_CLA_C sine_mainsV;
```

Step 3 Initialization in {ProjectName}-Main.c – Initialization can be achieved by SineAnalyzer_Diff_CLA_C_INIT macro which will set default values to the structure members:

```
SineAnalyzer_Diff_CLA_C_INIT(sine_mainsV);
```

Or values may be set individually as follows:

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

Step 4 Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the C28x:

```
// configure the RAM as CLA program memory
Cla1Regs.MMEMCFG.bit.PROGE = 1;
// configure RAM L1, F28035 as CLA Data memory 0
Cla1Regs.MMEMCFG.bit.RAM0E = 1;
// configure RAM L2, F28035 as CLA data memory 1
Cla1Regs.MMEMCFG.bit.RAM1E = 1;
```

Step 5 Using the Macro in Inverter Task

```
// Connect inputs, compute RMS, Avg, Freq & ZCD
interrupt void Cla1Task1(void) {
      …
      sine_mainsV.Vin = Vac_in;
      SineAnalyzer_Diff_CLA_C(sine_mainsV);
      VrmsReal = (KvInv* sine_mainsV.Vrms);
      …
}
```

Chapter 5. Revision History

Version	Date	Notes
V1.0	Jan 31, 2011	First Release of Solar Library
V1.1	Oct 29, 2012	Added CLA C version of Solar Library