C28x Solar Library

v1.2

Jan-14

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.

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 these tasks. The Solar library provides a framework structure, with known algorithms, for the user to implement algorithms needed for Sola Power Conversion Systems 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 software algorithm module
- 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
<base>\CLA	Contains CLA implementation of the solar library blocks

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.

				SOLAR LIB MODULE CYCLE-COUNT PROFILING RESULTS						
				IQ*			Float*			CLA*
Module		Module Type	Description	Asm	Function	Macro	Asm	Function	Macro	Macro
1	ABC_DQ0_POS	Transform	ABC to DQ0 Transform for positive sequence		114	95		67	49	50
2	ABC_DQ0_NEG	Transform	ABC to DQ0 Transform for negative sequence		114	94		66	51	48
3	DQ0_ABC	Transform	DQ0 to ABC Transform		93	77		68	45	42
4	SPLL_1ph	PLL	SPLL 1ph based on notch filter		218	188		161	136	155
5	SPLL_1ph_SOGI	PLL	SPLL 1ph based on second order generalized integrator		316	279		210	175	176
6	SPLL_3ph_SRF	PLL	SPLL 3ph stationary reference frame based		84	67		62	42	45
7	SPLL_3ph_DDSRF	PLL	SPLL 3ph decoupled double synchronous reference frame based		416	351		294	245	197
8	CLARKE	Transform	Clarke Transform		67	55		44	29	30
9	PARK	Transform	Park Transform		70	54		52	31	32
10	iPARK	Transform	Inverse Park Transform		67	55		52	29	30
11	iCLARKE	Transform	Inverse Clark Transform		58	50		41	27	23
12	MPPT_PNO	MPPT	Perturb and Observe MPPT Algorithm Module		82	60		85	78	82
13	MPPT_INCC	MPPT	Incremental Conductance MPPT Algorithm		228	198		87	78	132

			Module							
14	MPPT_INCC_I	MPPT	Incremental Conductance MPPT Algorithm Module		237	207		94	75	140
15	CNTL_2P2Z	CNTL	Control Law Two Pole Two Zero	44	80	60	49	60	42	45
16	CNTL_3P3Z	CNTL	Control Law Three Pole Three Zero	52	122	82	59	79	56	59
17	CNTL_PI	CNTL	Control Law PI		45	25		49	31	32
18	MATH_EMAVG	MATH	Moving Average/ Low Pass Filter Block		22	14		20	12	16
19	DLOG_1CH	UTIL	Data Logger Single Channel		65	19		63	49	93
20	DLOG_4CH	UTIL	Data Logger Four Channel		105	86		102	79	134
21	PID_GRANDO	CNTL	PID Module		115	79		80	54	65
22	SINEANALYZER_ DIFF (MAX)	MATH	Sine Analyzer		382	354		141	118	104
23	SINEANALYZER_ wPWR_DIFF (MAX)	MATH	Sine Analyzer with Power Measurement		251	212		208	179	138
24	NOTCH_FLTR	MATH	Notch Filter	50	63	51		40	33	
All results representative of non-volatile functions/macros. Library compiled with 6.2.3										

Note* the cycle counts reported should not be used as comparison with other devices as the cycle numbers can significantly change according to the profiling method and code structure. The numbers should be used as a guideline for CPU utilization when using the solar library and are provided for this purpose only.

3.2. Per Unit Format

The Solar Library supports IQ, Float and CLA based maths. Per unit values are typically used for variables in the solar library, per unit value is found by dividing the current reading by the maximum. For example if the voltage sense max is 20V and the instantaneous reading is 5V the per unit value is 5/20=0.25.

For IQ based blocks _IQ24 format is used and all the values are scaled from 0-1 in IQ24 format. For CLA and Float 32 but single precision floating point representation is used.

Chapter 4. Solar Lib Modules

4.1. Maximum Power Point Tracking (MPPT)

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

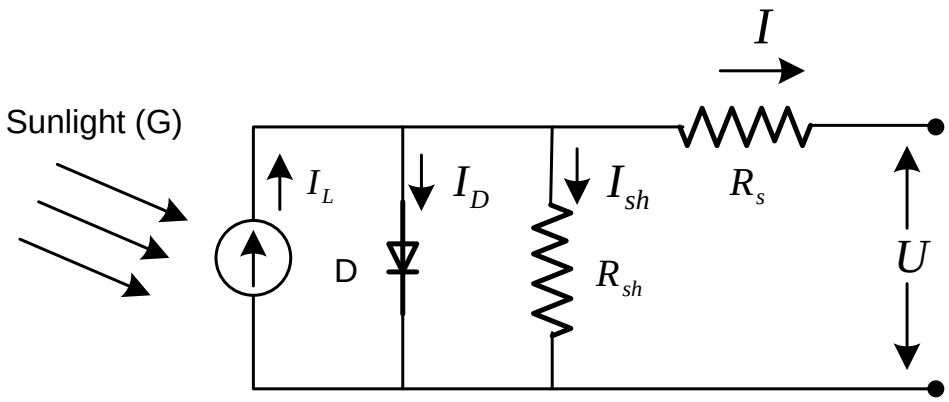


Figure 1 PV Cell Model

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

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

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

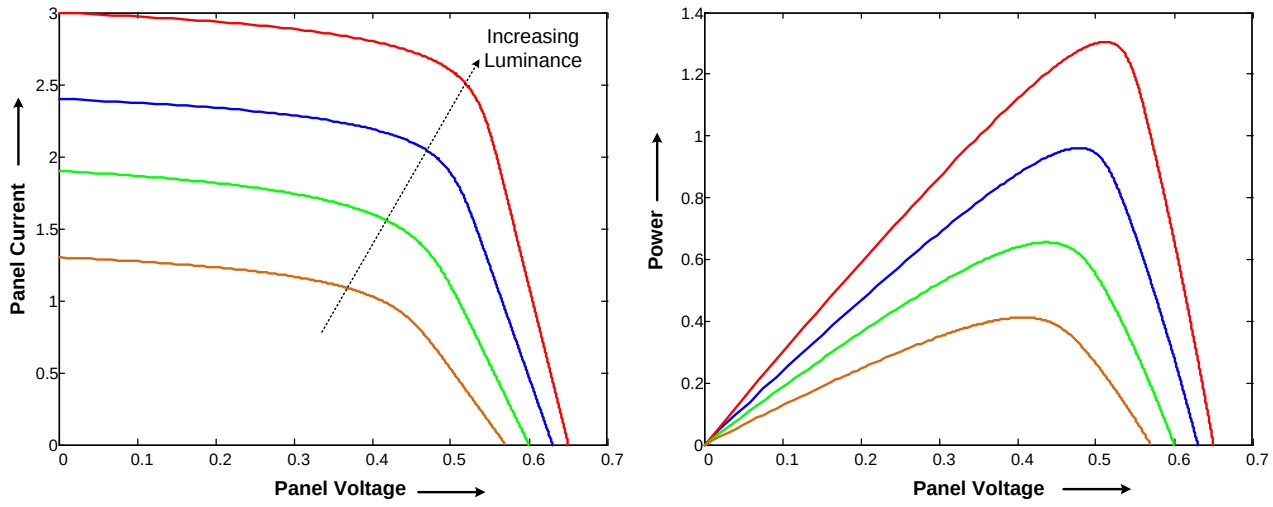


Figure 2 Solar Cell Characteristics

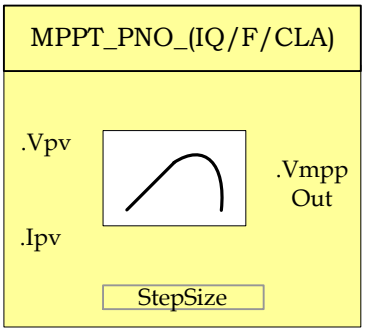
It is clear from the above V vs I curve that PV does not have a linear voltage and current relationship. Thus (P vs V) curve clearly shows a presence of a maximum. To get the most energy/utilization out of the PV system installation it must be operated at the maximum power point of this curve. The maximum 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.



Module File: <*base_folder*>\MPPT_PNO_(IQ/F/CLA).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 per unit format.
2) Panel Current (Ipv): This is the sensed panel current signal sampled by ADC and ADC result converted to per unit 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 per unit format.

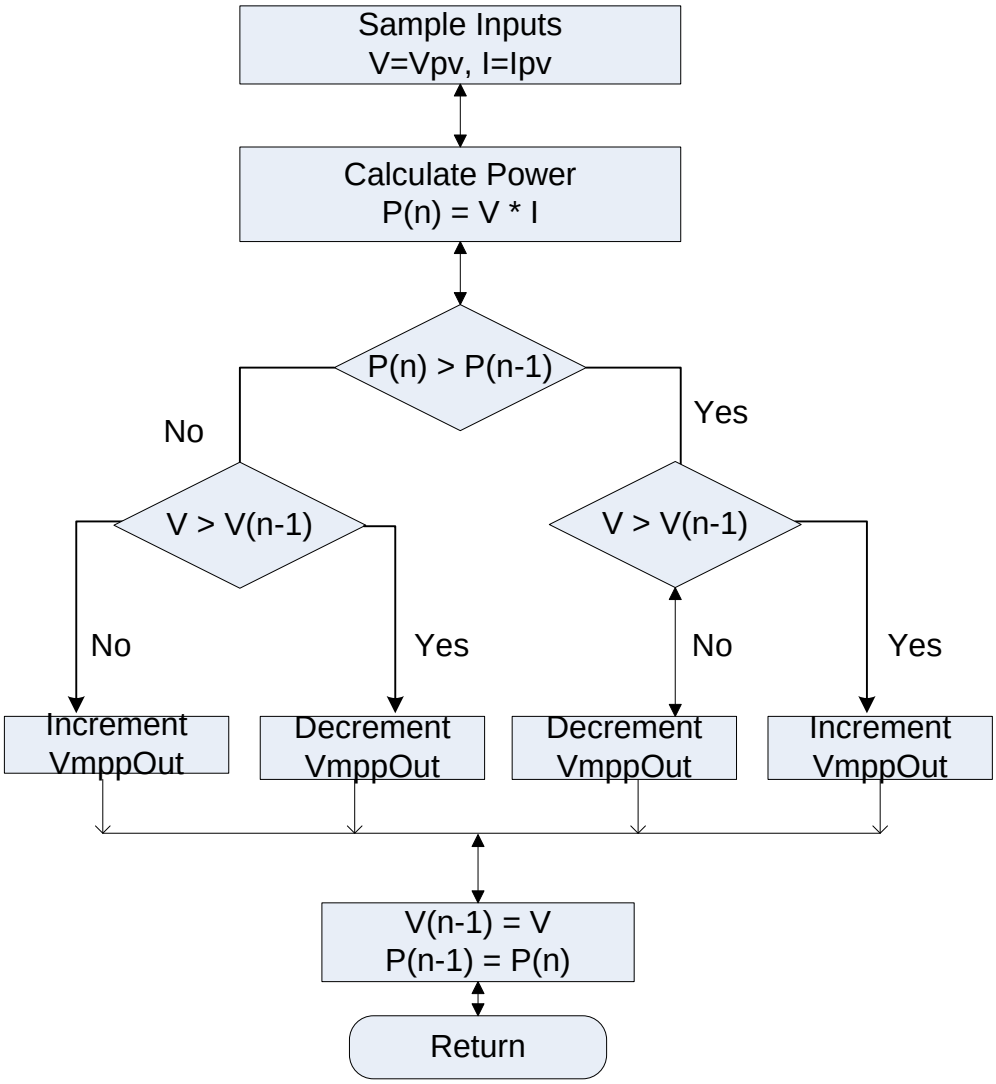


Figure 3 Perturb & Observe Algorithm Flowchart for MPPT

Object Definition:

- **Fixed Point (IQ)**

```
//************ Structure Definition ********//
typedef struct {
      int32  Ipv;
      int32  Vpv;
      int32  DeltaPmin;
      int32  MaxVolt;
      int32  MinVolt;
      int32  Stepsize;
      int32  VmppOut;
      int32  DeltaP;
      int32  PanelPower;
      int32  PanelPower_Prev;
      Uint16 mppt_enable;
      Uint16 mppt_first;
} MPPT_PNO_IQ;
```

- **Floating Point (F)**

```
//************ Structure Definition ********//
typedef struct {
      float32  Ipv;
      float32  Vpv;
      float32  DeltaPmin;
      float32  MaxVolt;
      float32  MinVolt;
      float32  Stepsize;
      float32  VmppOut;
      float32  DeltaP;
      float32  PanelPower;
      float32  PanelPower_Prev;
      Uint16 mppt_enable;
      Uint16 mppt_first;
} MPPT_PNO_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//************ Structure Definition ********//
typedef struct {
      float32  Ipv;
      float32  Vpv;
      float32  DeltaPmin;
      float32  MaxVolt;
      float32  MinVolt;
      float32  Stepsize;
      float32  VmppOut;
      float32  DeltaP;
      float32  PanelPower;
      float32  PanelPower_Prev;
      Uint16 mppt_enable;
      Uint16 mppt_first;
} MPPT_PNO_CLA;
```

Module interface Definition:

Module Element Name	Type	Description	Acceptable Range
Vpv	Input	Panel Voltage input	Q24 [0,1) / Float32[0,1)
Ipv	Input	Panel Current input	Q24 [0,1) / Float32[0,1)
StepSize	Input	Step size input used for changing reference MPP voltage output generated	Q24 [0,1) / Float32[0,1)
DeltaPmin	Input	Threshold limit of power change for which perturbation takes place.	Q24 [0,1) / Float32[0,1)
MaxVolt	Input	Upper Limit on the voltage reference value generated by MPPT algorithm – max value of VmppOut	Q24 [0,1) / Float32[0,1)
MinVolt	Input	Lower Limit on the voltage reference value generated by MPPT algorithm – Min value of VmppOut	Q24 [0,1) / Float32[0,1)
VmppOut	Output	MPPT output voltage reference generated	Q24 [0,1) / Float32[0,1)
DeltaP	Internal	Change in Power	
PanelPower	Internal	Latest Panel power calculated from Vpv and Ipv	
PanelPower_prev	Internal	Previous value of Panel Power	
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:

- **Fixed Point (IQ)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
MPPT_PNO_IQ mppt_pno1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
MPPT_PNO_IQ_init(&mppt_pno1);
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 4 – **Using the module**

```
// 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_IQ_FUNC(&mppt_pno1);
Alternatively the macro routine can be called as below:
      MPPT_PNO_IQ_MACRO(mppt_pno1);

// Output of the MPPT macro can be written to the reference
// of the voltage regulator
      Vpvref_mpptOut = mppt_pno1.VmppOut;
```

- **Floating Point (F)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
MPPT_PNO_F mppt_pno1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

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

Step 4 – Using the module

```
// 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_F_FUNC(&mppt_pno1);
Alternatively the macro routine can be called as below:
      MPPT_PNO_F_MACRO(mppt_pno1);
// Output of the MPPT macro can be written to the reference
// of the voltage regulator
      Vpvref_mpptOut = mppt_pno1.VmppOut;
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(mppt_pno1, "Cla1ToCpuMsgRAM");
ClaToCpu_Volatile MPPT_PNO_CLA 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 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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
        ….
        //mppt_pno1 macro initialization
        MPPT_PNO_CLA_init(mppt_pno1);
        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;
        ….
}
```

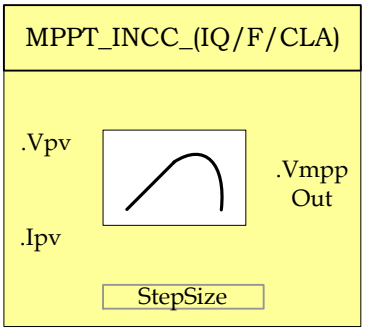
The task is forced from `{ProjectName}-Main.c` by calling:

```
Cla1ForceTask8andWait();
```

Step 5 – Using the module 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_MACRO(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.



Module File: <*base_folder*>\MPPT_INCC_(IQ/F/CLA).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 per unit format.
2) Panel Current (Ipv): This is the sensed panel current signal sampled by ADC and ADC result converted to per unit 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 per unit format.

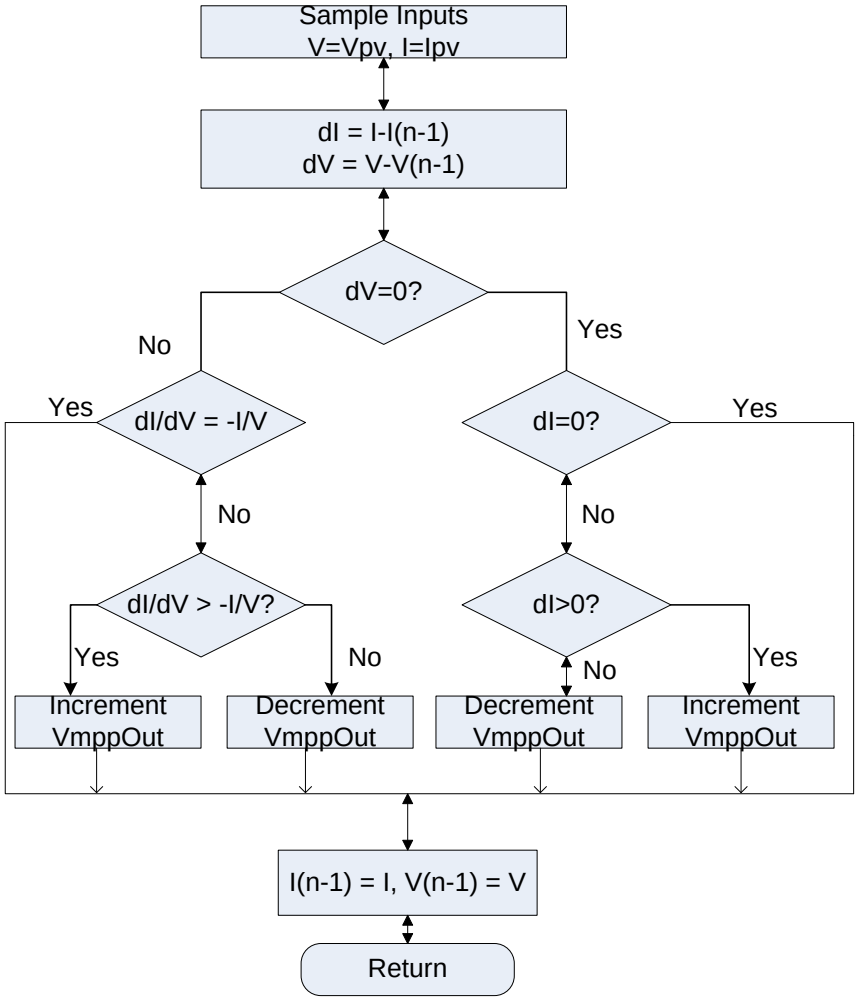


Figure 4 Incremental Conductance Method Flowchart

Object Definition:

- **Fixed Point (IQ)**

```
//************ Structure Definition ********//
typedef struct {
      int32  Ipv;
      int32  Vpv;
      int32  IpvH;
      int32  IpvL;
      int32  VpvH;
      int32  VpvL;
      int32  MaxVolt;
      int32  MinVolt;
      int32  Stepsize;
      int32  VmppOut;
      // internal variables
      int32  Cond;
      int32  IncCond;
      int32  DeltaV;
      int32  DeltaI;
      int32  VpvOld;
      int32  IpvOld;
      int32  StepFirst;
      Uint16 mppt_enable;
      Uint16 mppt_first;
} MPPT_INCC_IQ;
```

- **Floating Point (F)**

```
//************ Structure Definition ********//
typedef struct {
      float32  Ipv;
      float32  Vpv;
      float32  IpvH;
      float32  IpvL;
      float32  VpvH;
      float32  VpvL;
      float32  MaxVolt;
      float32  MinVolt;
      float32  Stepsize;
      float32  VmppOut;
      float32  Cond;
      float32  IncCond;
      float32  DeltaV;
      float32  DeltaI;
      float32  VpvOld;
      float32  IpvOld;
      Uint16 mppt_enable;
      Uint16 mppt_first;
} MPPT_INCC_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//************ Structure Definition ********//
typedef struct {
      float32  Ipv;
      float32  Vpv;
      float32  IpvH;
      float32  IpvL;
      float32  VpvH;
      float32  VpvL;
      float32  MaxVolt;
      float32  MinVolt;
      float32  Stepsize;
      float32  VmppOut;
      float32  Cond;
      float32  IncCond;
      float32  DeltaV;
      float32  DeltaI;
      float32  VpvOld;
      float32  IpvOld;
      Uint16 mppt_enable;
      Uint16 mppt_first;
} MPPT_INCC_CLA;
```

Module interface Definition:

Module Element Name	Type	Description	Acceptable Range
Vpv	Input	Panel Voltage input	Q24 [0,1) / Float32[0,1)
Ipv	Input	Panel Current input	Q24 [0,1) / Float32[0,1)
StepSize	Input	Step size input used for changing reference MPP voltage output generated	Q24 [0,1) / Float32[0,1)
VpvH	Input	Threshold limit for change in voltage in +ve direction	Q24 [0,1) / Float32[0,1)
VpvL	Input	Threshold limit for change in voltage in -ve direction	Q24 [0,1) / Float32[0,1)
IpvH	Input	Threshold limit for change in Current in +ve direction	Q24 [0,1) / Float32[0,1)
IpvL	Input	Threshold limit for change in Current in -ve direction	Q24 [0,1) / Float32[0,1)
MaxVolt	Input	Upper Limit on the voltage reference value generated by MPPT algorithm – max value of VmppOut	Q24 [0,1) / Float32[0,1)
MinVolt	Input	Lower Limit on the voltage reference value generated by MPPT algorithm – Min value of VmppOut	Q24 [0,1) / Float32[0,1)
VmppOut	Output	MPPT output voltage reference generated	Q24 [0,1) / Float32[0,1)
Cond	Internal	Conductance value calculated	Q24 [0,1) / Float32[0,1)

IncCond	Internal	Incremental Conductance value calculated	Q24 [0,1) / Float32[0,1)
DeltaV	Internal	Change in Voltage	Q24 [0,1) / Float32[0,1)
DeltaI	Internal	Change in Current	Q24 [0,1) / Float32[0,1)
VpvOld	Internal	Previous value of Vpv	Q24 [0,1) / Float32[0,1)
IpvOld	Internal	Previous value of Ipv	Q24 [0,1) / Float32[0,1)
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:

- **Fixed Point (IQ)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
MPPT_INCC_IQ mppt_incc1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
//mppt INCC
MPPT_INCC_IQ_init(&mppt_incc1);
mppt_incc1.DeltaPmin = _IQ(0.00001);
mppt_incc1.MaxVolt = _IQ(0.9);
mppt_incc1.MinVolt = _IQ(0.0);
mppt_incc1.Stepsize = _IQ(0.005);
```

Step 4 – **Using the module**

```
// 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_IQ_FUNC (&mppt_incc1);
```

Alternatively the macro routine can be called as below:

```
      MPPT_INCC_IQ_MACRO(mppt_incc1);
// Output of the MPPT macro can be written to the reference of
// the voltage regulator
   Vpvref_mpptOut = mppt_incc1.VmppOut;
```

- **Floating Point (F)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
MPPT_INCC_F mppt_incc1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
//mppt INCC
MPPT_INCC_IQ_init(&mppt_incc1);
mppt_incc1.DeltaPmin = 0.00001;
mppt_incc1.MaxVolt = 0.9;
mppt_incc1.MinVolt = 0.0;
mppt_incc1.Stepsize = 0.005;
```

Step 4 – Using the module

```
// 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_F_FUNC (&mppt_incc1);
```

Alternatively the macro routine can be called as below:

```
      MPPT_INCC_F_MACRO(mppt_incc1);
// Output of the MPPT macro can be written to the reference of
// the voltage regulator
   Vpvref_mpptOut = mppt_incc1.VmppOut;
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

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

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
….
    //mppt_incc1 macro initialization
    MPPT_INCC_CLA_init(mppt_incc1);
    mppt_incc1.DeltaPmin = 0.00001;
    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;
….
```

```
}
```

The task is forced from `{ProjectName}-Main.c` by calling:

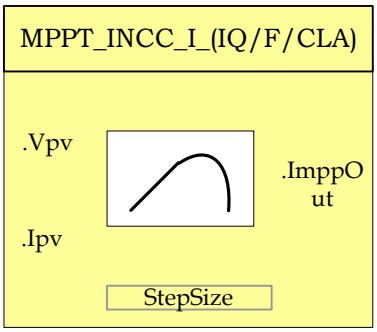
```
Cla1ForceTask8andWait();
```

Step 5 Using the module 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_MACRO(mppt_incc1);
    // Output of the MPPT macro can be written to
    // the reverence of the voltage regulator
    Vpvref_mpptOut = mppt_incc1.VmppOut;
    …
}
```

MPPT_INCC_I	*Incremental Conductance Method for MPPT*

Description: This software module implemented the incremental conductance algorithm used for maximum power point tracking purpose based on current.



Module File: <*base_folder*>\MPPT_INCC_I_(IQ/F/CLA).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. Iref is the reference current that is forced to be drawn from the PV array. At the MPP, Iref equals to I_{MPP} of the panel. Once the MPP is reached, the operation of the PV array is maintained at this point unless a change is noted. Figure 5 illustrates the flowchart for the incremental conductance method. The algorithm decrements or increments Iref 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 per unit value.
2) Panel Current (Ipv): This is the sensed panel current signal sampled by ADC and ADC result converted to per unit value.

3) Step Size (Stepsize): Size of the step used for changing the MPP 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 – current reference for MPP tracing is generated based on the conductance and incremental conductance values calculated.

This module generates the following Outputs:

1) Current reference for MPP (ImppOut): Current reference for MPP tracking obtained by incremental conductance algorithm. Output in per unit format.

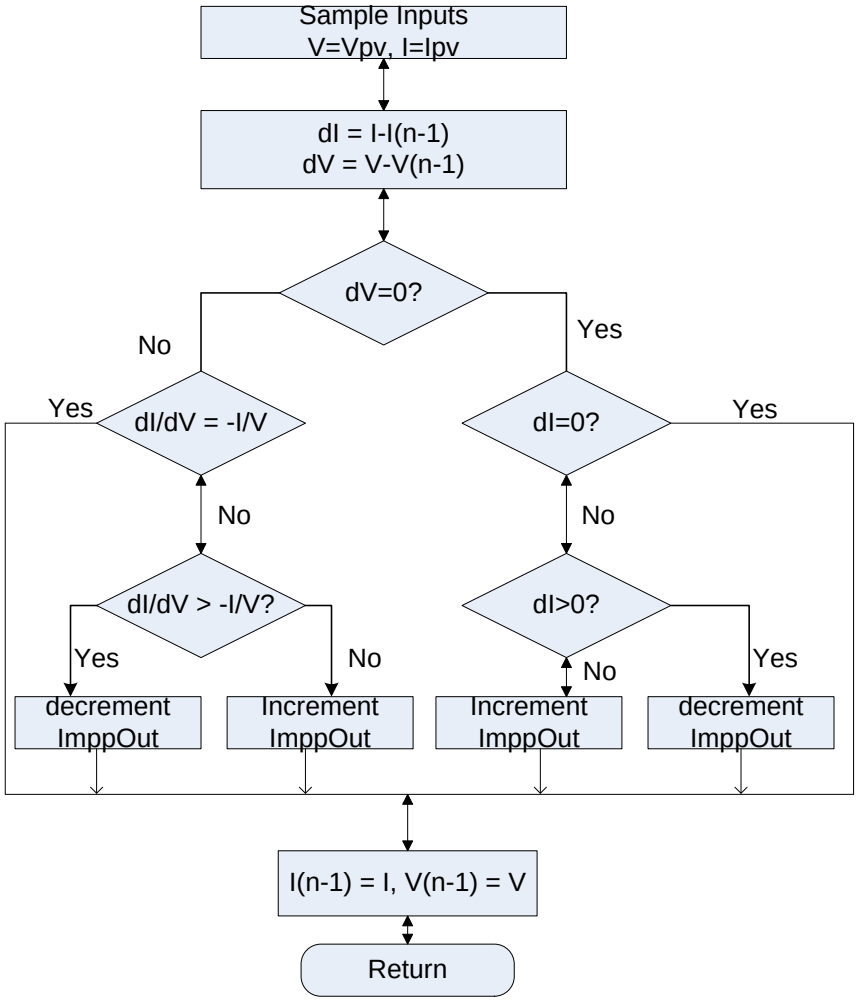


Figure 5 Incremental Conductance Method Current Based Flowchart

Object Definition:

- **Fixed Point (IQ)**

```
//************ Structure Definition ********//
typedef struct{
      int32  Ipv;
      int32  Vpv;
      int32  IpvH;
      int32  IpvL;
      int32  VpvH;
      int32  VpvL;
      int32  MaxI;
      int32  MinI;
      int32  Stepsize;
      int32  ImppOut;
      int32  Cond;
      int32  IncCond;
      int32  DeltaV;
      int32  DeltaI;
      int32  VpvOld;
      int32  IpvOld;
      int32  StepFirst;
      Uint16 mppt_enable;
      Uint16 mppt_first;
} MPPT_INCC_I_IQ;
```

- **Floating Point (F)**

```
//************ Structure Definition ********//
typedef struct {
      float32  Ipv;
      float32  Vpv;
      float32  IpvH;
      float32  IpvL;
      float32  VpvH;
      float32  VpvL;
      float32  MaxI;
      float32  MinI;
      float32  Stepsize;
      float32  ImppOut;
      // internal variables
      float32  Cond;
      float32  IncCond;
      float32  DeltaV;
      float32  DeltaI;
      float32  VpvOld;
      float32  IpvOld;
      float32  StepFirst;
      Uint16 mppt_enable;
      Uint16 mppt_first;
} MPPT_INCC_I_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//************ Structure Definition ********//
typedef struct {
```

```
        float32  Ipv;
        float32  Vpv;
        float32  IpvH;
        float32  IpvL;
        float32  VpvH;
        float32  VpvL;
        float32  MaxI;
        float32  MinI;
        float32  Stepsize;
        float32  ImppOut;
        // internal variables
        float32  Cond;
        float32  IncCond;
        float32  DeltaV;
        float32  DeltaI;
        float32  VpvOld;
        float32  IpvOld;
        float32  StepFirst;
        Uint16 mppt_enable;
        Uint16 mppt_first;
} MPPT_INCC_I_CLA;
```

Module interface Definition:

Module Element Name	Type	Description	Acceptable Range
Vpv	Input	Panel Voltage input	Q24 [0,1) / Float32[0,1)
Ipv	Input	Panel Current input	Q24 [0,1) / Float32[0,1)
StepSize	Input	Step size input used for changing reference MPP voltage output generated	Q24 [0,1) / Float32[0,1)
VpvH	Input	Threshold limit for change in voltage in +ve direction	Q24 [0,1) / Float32[0,1)
VpvL	Input	Threshold limit for change in voltage in -ve direction	Q24 [0,1) / Float32[0,1)
IpvH	Input	Threshold limit for change in Current in +ve direction	Q24 [0,1) / Float32[0,1)
IpvL	Input	Threshold limit for change in Current in -ve direction	Q24 [0,1) / Float32[0,1)
MaxI	Input	Upper Limit on the current reference value generated by MPPT algorithm – max value of ImppOut	Q24 [0,1) / Float32[0,1)
MinI	Input	Lower Limit on the current reference value generated by MPPT algorithm – Min value of ImppOut	Q24 [0,1) / Float32[0,1)
ImppOut	Output	MPPT output current reference generated	Q24 [0,1) / Float32[0,1)
Cond	Internal	Conductance value calculated	Q24 [0,1) / Float32[0,1)

IncCond	Internal	Incremental Conductance value calculated	Q24 [0,1) / Float32[0,1)
DeltaV	Internal	Change in Voltage	Q24 [0,1) / Float32[0,1)
DeltaI	Internal	Change in Current	Q24 [0,1) / Float32[0,1)
VpvOld	Internal	Previous value of Vpv	Q24 [0,1) / Float32[0,1)
IpvOld	Internal	Previous value of Ipv	Q24 [0,1) / Float32[0,1)
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:

- **Fixed Point (IQ)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_I_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
MPPT_INCC_I_IQ mppt_incc_I1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
//mppt INCC_I
MPPT_INCC_I_IQ_init(&mppt_incc1);
mppt_incc_I1.MaxI = _IQ(0.9);
mppt_incc_I1.MinI = _IQ(0.0);
mppt_incc_I1.Stepsize = _IQ(0.005);
```

Step 4 – Using the module

```
// Write normalized panel current and voltage values
// to the MPPT macro
      mppt_incc_I1.Ipv = IpvRead; \\ Normalized Panel Current
      mppt_incc_I1.Vpv = VpvRead; \\ Normalized Panel Voltage
// Invoking the MPPT computation macro
      MPPT_INCC_I_IQ_FUNC(&mppt_incc_I1);
```

Alternatively the macro routine can be called as below:

```
      MPPT_INCC_I_IQ_MACRO(mppt_incc_I1);
// Output of the MPPT macro can be written to the reference of
```

```
// the voltage regulator
Ipvref_mpptOut = mppt_incc_I1.ImppOut;
```

- **Floating Point (F)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
MPPT_INCC_I_F mppt_incc_I1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
//mppt INCC_I
MPPT_INCC_I_IQ_init(&mppt_incc_I1);
mppt_incc_I1.MaxI = 0.9;
mppt_incc_I1.MinI = 0.0;
mppt_incc_I1.Stepsize = 0.005;
```

Step 4 – Using the module

```
// Write normalized panel current and voltage values
// to the MPPT macro
    mppt_incc_I1.Ipv = IpvRead; // Normalized Panel Current
    mppt_incc_I1.Vpv = VpvRead; // Normalized Panel Voltage
// Invoking the MPPT computation macro
    MPPT_INCC_I_F_FUNC (&mppt_incc_I1);
```

Alternatively the macro routine can be called as below:

```
    MPPT_INCC_I_F_MACRO(mppt_incc_I1);
// Output of the MPPT macro can be written to the reference of
// the voltage regulator
    Ipvref_mpptOut = mppt_incc_I1.ImppOut;
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(mppt_incc_I1, "Cla1ToCpuMsgRAM");
ClaToCpu_Volatile MPPT_INCC_I_CLA mppt_incc_I1;
```

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_I_CLA mppt_incc_I1;
```

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
    ….
    //mppt_incc_I1 macro initialization
    MPPT_INCC_I_CLA_init(mppt_incc_I1);
    mppt_incc_I1.MaxI = 0.9;
    mppt_incc_I1.MinI = 0.0;
    mppt_incc_I1.Stepsize = 0.005;
    mppt_incc_I1.MPPT_First = 1;
    mppt_incc_I1.MPPT_Enable = 1;
    ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

```
Cla1ForceTask8andWait();
```

Step 5 Using the module 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**.**

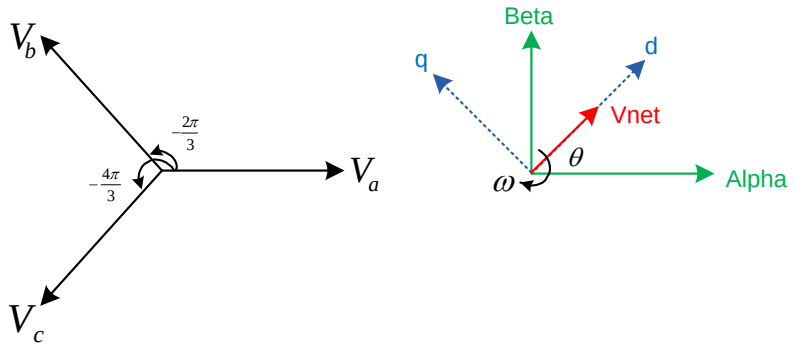
```
interrupt void Cla1Task1(void) {
    …
    // Write normalized panel current and voltage
```

```
        // values to MPPT object.
        mppt_incc_I1.Ipv = IpvRead;
        mppt_incc_I1.Vpv = VpvRead;
        // Invoke the MPPT macro
        MPPT_INCC_I_CLA_MACRO(mppt_incc_I1);
        // Output of the MPPT macro can be written to
        // the reverence of the voltage regulator
        Ipvref_mpptOut = mppt_incc_I1.ImppOut;
        …
}
```

4.2 Transforms

A three phase time varying system can be reduced to a dc system, with a rotating reference frame with the help of transforms. This is a convenient control method used in three phase power control. Assuming the below equation for the three phase quantities the sequence of the voltages is $V_a \rightarrow V_c \rightarrow V_b$, and the frequency is ω.

$$\begin{bmatrix} V_a \\ V_b \\ V_c \end{bmatrix} = V \begin{bmatrix} \cos(\omega t) \\ \cos(\omega t - 2\pi/3) \\ \cos(wt - 4\pi/3) \end{bmatrix}$$



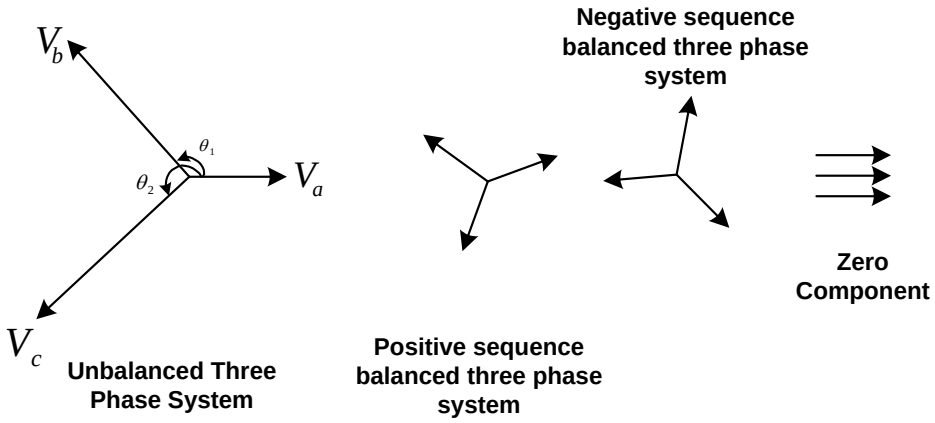
These three phase quantities care first reduced to an orthogonal component system (alpha, beta also called stationary reference frame), this is called the clark transform, by taking the projections of the three phase quantities on the orthogonal axis.

$$\begin{bmatrix} V_\alpha \\ V_\beta \\ V_o \end{bmatrix} = \sqrt{\frac{2}{3}} \begin{bmatrix} 1 & \cos(2\pi/3) & \cos(4\pi/3) \\ 0 & \sin(2\pi/3) & \sin(4\pi/3) \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} X \begin{bmatrix} V_a \\ V_b \\ V_c \end{bmatrix}$$

Now the net voltage vector can be assumed to be making an angle θ with the orthogonal reference frame and rotating at a frequency of ω. Thus the system can be reduced to DC by taking projection of the orthogonal components on the rotating reference frame.

$$\begin{bmatrix} v_d \\ v_q \\ v_o \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} X \begin{bmatrix} v_\alpha \\ v_\beta \\ v_o \end{bmatrix}$$

The grid is subject to varying conditions which result in imbalances in the phase voltages. From the theory of symmetrical components we know that any unbalanced three phase system can be reduced to two symmetrical systems and zero component. The behavior of unbalanced voltages on park and clark transform is analyzed in the section below.



Now an unbalanced three phase system can be written as summation of balanced three phase systems one rotating with the sequence of the three phase quantities called the positive sequence and one rotating in the opposite sequence called the negative sequence.

$$v = V^{+1}\begin{bmatrix}\cos(wt)\\ \cos(wt-2\pi/3)\\ \cos(wt-4\pi/3)\end{bmatrix} + V^{-1}\begin{bmatrix}\cos(wt)\\ \cos(wt-4\pi/3)\\ \cos(wt-2\pi/3)\end{bmatrix} + V^{0}\begin{bmatrix}1\\ 1\\ 1\end{bmatrix}$$

Now resolving the components in the orthogonal axis,

$$v_{\alpha\beta+} = T_{abc->\alpha\beta+} * v = V^{+1}\begin{bmatrix}\cos(wt)\\ \sin(wt)\end{bmatrix} + V^{-1}\begin{bmatrix}\cos(-wt)\\ \sin(-wt)\end{bmatrix}$$

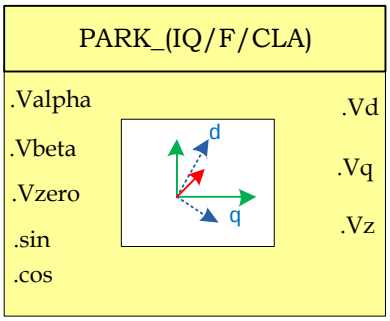
And taking the projections on the rotating reference frame we observe that any negative sequence component appears with twice the frequency on the positive sequence rotating frame axis and vice versa.

$$v_{dq+} = T_{abc->dq0+} = V^{+1}\begin{bmatrix}1\\ 0\end{bmatrix} + V^{-1}\begin{bmatrix}\cos(-2wt)\\ \sin(-2wt)\end{bmatrix}$$

$$v_{dq-} = T_{abc->dq0-} = V^{+1}\begin{bmatrix}\cos(-2wt)\\ \sin(-2wt)\end{bmatrix} + V^{-1}\begin{bmatrix}1\\ 0\end{bmatrix}$$

This can cause errors in the control loop and estimation of the grid angle and needs to taken into account while designing a converter. The solar library provides full transforms for three phase quantities.

Description: This software module implements the transform from the stationary reference frame to rotating reference frame



Module File: <*base_folder*>\PARK_(IQ/F/CLA).h

Technical: The block converts the stationary reference frame quantities to rotating reference frame quantities

$$T_{abc->dq0} = \begin{bmatrix} v_d \\ v_q \\ v_o \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} X \begin{bmatrix} v_\alpha \\ v_\beta \\ v_o \end{bmatrix}$$

Object Definition:

- **Fixed Point (IQ)**

```
//*********** Structure Definition ********//
typedef struct{
      int32 alpha;
      int32 beta;
      int32 zero;
      int32 sin;
      int32 cos;
      int32 d;
      int32 q;
      int32 z;
}PARK_IQ;
```

- **Floating Point (F)**

```
//*********** Structure Definition ********//
typedef struct{
      float32 alpha;
      float32 beta;
      float32 zero;
      float32 sin;
      float32 cos;
      float32 d;
      float32 q;
      float32 z;
}PARK_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//************ Structure Definition ********//
typedef struct{
      float32 alpha;
      float32 beta;
      float32 zero;
      float32 sin;
      float32 cos;
      float32 d;
      float32 q;
      float32 z;
}PARK_CLA;
```

Module Interface Definition:

Module Element Name	Type	Description	Acceptable Range
alpha,beta	Input	ABC component on the stationary reference frame Alpha, Beta	IQ(-1,1) Float(-1,1)
sin,cos	Input	Sin and cos of the grid angle	IQ Float(-1,1)
d,q,z	Output	Rotating Reference Frame Quantities	IQ Float(-1,1)

Usage:

- **Fixed Point (IQ)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
PARK_IQ park1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
PARK_IQ_init(&park1);
```

Step 4 – **Using the module**

```
park1.alpha = Valpha;
park1.beta = Vbeta;
park1.zero = Vzero;
park1.sin = sin(theta);
park1.cos = cos(theta);
PARK_IQ_FUNC(&park1);
```

Alternatively the macro routine can be called as bellow:

```
PARK_IQ_MACRO(park1);
```

- **Floating Point (F)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
PARK_F park1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
PARK_F_init(&park1);
```

Step 4 – Using the module

```
park1.alpha = Valpha;
park1.beta = Vbeta;
park1.zero = Vzero;
park1.sin = sin(theta);
park1.cos = cos(theta);
PARK_F_FUNC(&park1);
```

Alternatively the macro routine can be called as bellow:

```
PARK_F_MACRO(park1);
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(park1,"CLADataRAM")
ClaToCpu_Volatile PARK_CLA park1;
```

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

```
extern ClaToCpu_Volatile PARK_CLA park1;
```

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
      ….
      PARK_CLA_init(park1);
      ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

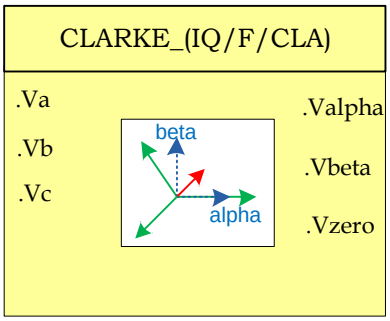
```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
      ….
      // Write per unit values of voltage/current
      park1.alpha = Valpha;
      park1.beta = Vbeta;
      park1.zero = Vzero;
      park1.sin = sin(theta);
      park1.cos = cos(theta);
      // Invoke the run time macro
      PARK_CLA_MACRO(park1);
      ….
}
```

CLARKE — *Clarke Transform*

Description: This software module implements the transform from three phase quantities ABC to stationary reference frame.



Module File: <*base_folder*>\CLARKE_(IQ/F/CLA).h

Technical: The block converts the three phase quantities into orthogonal axis components alpha and beta.

$$V_{\alpha\beta} = T_{abc->\alpha\beta} = \frac{2}{3}\begin{bmatrix} 1 & \cos(2\pi/3) & \cos(4\pi/3) \\ 0 & \sin(2\pi/3) & \sin(4\pi/3) \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix} x \begin{bmatrix} V_a \\ V_b \\ V_c \end{bmatrix} = \frac{2}{3}\begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix} x \begin{bmatrix} V_a \\ V_b \\ V_c \end{bmatrix}$$

Object Definition:

- **Fixed Point (IQ)**

```
//*********** Structure Definition ********//
typedef struct{
      int32 a;
      int32 b;
      int32 c;
      int32 alpha;
      int32 beta;
      int32 zero;
}CLARKE_IQ;
```

- **Floating Point (F)**

```
//*********** Structure Definition ********//
typedef struct{
      float32 a;
      float32 b;
      float32 c;
      float32 alpha;
      float32 beta;
      float32 zero;
}CLARKE_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//*********** Structure Definition ********//
typedef struct{
      float32 a;
      float32 b;
      float32 c;
      float32 alpha;
      float32 beta;
      float32 zero;
}CLARKE_CLA;
```

Module interface Definition:

Module Element Name	Type	Description	Acceptable Range
a,b,c	Input	3ph AC Signal measured and normalized	IQ(-1,1), Float(-1,1)
alpha, beta, zero	Output	ABC component on the stationary reference frame Alpha, Beta	IQ(-1,1), Float(-1,1)

Usage:

- **Fixed Point (IQ)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
CLARKE_IQ clarke1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
CLARKE_IQ_init(&clarke1);
```

Step 4 – **Using the module –** The voltage/current values are read and converted to per unit format and provided to the CLARKE structure.

```
// Write per unit current / voltage values
clarke1.a= Vmeas_a;
clarke1.b= Vmeas_b;
clarke1.c= Vmeas_c;
CLARKE_IQ_FUNC(&clarke1);
```

Alternatively the macro routine can be called as below:

```
CLARKE_IQ_MACRO(clarke1);
```

- **Floating Point (F)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
CLARKE_F clarke1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
CLARKE_IQ_init(&clarke1);
```

Step 4 – **Using the module –** The voltage/current values are read and converted to per unit format and provided to the CLARKE structure.

```
// Write per unit current / voltage values
clarke1.a= Vmeas_a;
clarke1.b= Vmeas_b;
clarke1.c= Vmeas_c;
CLARKE_F_FUNC(&clarke1);
```

Alternatively the macro routine can be called as below:

```
CLARKE_F_MACRO(clarke1);
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(clarke1, "Cla1ToCpuMsgRAM");
ClaToCpu_Volatile CLARKE_CLA clarke1;
```

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

```
extern ClaToCpu_Volatile CLARKE_CLA clarke1;
```

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
      ….
      CLARKE_CLA_init(clarke1);
      ….
}
```

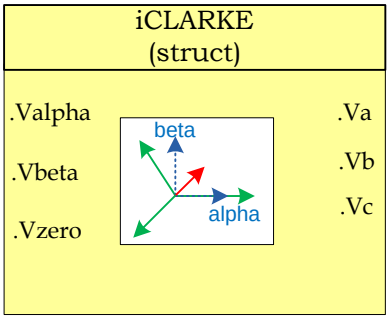
The task is forced from {ProjectName}-Main.c by calling:

```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
      ….
      // Write per unit values of voltage/current
      clarke1.a= Vmeas_a;
      clarke1.b= Vmeas_b;
      clarke1.c= Vmeas_c;
      // Invoke the run time macro
      CLARKE_CLA_MACRO(clarke1);
      ….
}
```

Description: This software module implements the transform from stationary reference frame quantities back into ABC frame



Module File: <*base_folder*>\iCLARKE_(IQ/F/CLA).h

Technical: The block converts the stationary reference frame quantities to three phase quantities.

$$V_{abc} = T^{-1}_{abc->\alpha\beta} * V_{\alpha\beta} = \begin{bmatrix} 1 & 0 & \frac{1}{2} \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} & \frac{1}{2} \\ -\frac{1}{2} & -\frac{\sqrt{3}}{2} & \frac{1}{2} \end{bmatrix} x \begin{bmatrix} V_{alpha} \\ V_{beta} \\ V_{zero} \end{bmatrix}$$

Object Definition:

- **Fixed Point (IQ)**

```
//*********** Structure Definition ********//
typedef struct{
      int32 a;
      int32 b;
      int32 c;
      int32 alpha;
      int32 beta;
      int32 zero;
}iCLARKE_IQ;
```

- **Floating Point (F)**

```
//*********** Structure Definition ********//
typedef struct{
      float32 a;
      float32 b;
      float32 c;
      float32 alpha;
      float32 beta;
      float32 zero;
}iCLARKE_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//************ Structure Definition ********//
typedef struct{
      float32 a;
      float32 b;
      float32 c;
      float32 alpha;
      float32 beta;
      float32 zero;
}iCLARKE_CLA;
```

Module interface Definition:

Module Element Name	Type	Description	Acceptable Range
a,b,c	Output	3ph AC Signal measured and normalized	IQ(-1,1) Float(-1,1)
alpha, beta, zero	Input	ABC component on the stationary reference frame Alpha, Beta	IQ Float(-1,1)

Usage:

- **Fixed Point (IQ)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
iCLARKE_IQ iclarke1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
iCLARKE_IQ_init(&iclarke1);
```

Step 4 – **Using the module**

```
iclarke1.alpha = Valpha;
iclarke1.beta = Vbeta;
iclarke1.zero = Vzero;
iCLARKE_IQ_FUNC(&iclarke1);
```

Alternatively the macro routine can be called as bellow:

```
iCLARKE_IQ_MACRO(iclarke1);
```

- **Floating Point (F)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
iCLARKE_F iclarke1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
iCLARKE_F_init(&iclarke1);
```

Step 4 – **Using the module**

```
iclarke1.alpha = Valpha;
iclarke1.beta = Vbeta;
iclarke1.zero = Vzero;
iCLARKE_F_FUNC(&iclarke1);
```

Alternatively the macro routine can be called as bellow:

```
iCLARKE_F_MACRO(iclarke1);
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(iclarke1,"CLADataRAM")
ClaToCpu_Volatile iCLARKE_CLA iclarke1;
```

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

```
extern ClaToCpu_Volatile iCLARKE_CLA iclarke1;
```

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-Main.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
      ….
      iCLARKE_CLA_init(iclarke1);
      ….
}
```

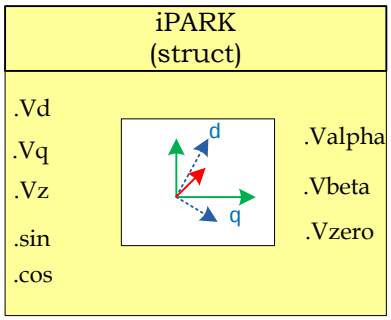
The task is forced from {ProjectName}-Main.c by calling:

```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
      ….
      iclarke1.alpha = Valpha;
      iclarke1.beta = Vbeta;
      iclarke1.zero = Vzero;
      // Invoke the run time macro
      iCLARKE_CLA_MACRO(iclarke1);
      ….
}
```

Description: This software module implements the transform from the rotating reference frame to the stationary reference frame



Module File: <*base_folder*>\iPARK_(IQ/F/CLA).h

Technical: The block converts the rotating reference frame quantities to stationary reference frame quantities

$$V_{\alpha\beta} = T^{-1}{}_{\alpha\beta->dq0} * \begin{bmatrix} v_d \\ v_q \\ v_o \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} X \begin{bmatrix} v_d \\ v_q \\ v_z \end{bmatrix}$$

Object Definition:

- **Fixed Point (IQ)**

```
//*********** Structure Definition ********//
typedef struct{
       int32 alpha;
       int32 beta;
       int32 zero;
       int32 sin;
       int32 cos;
       int32 d;
       int32 q;
       int32 z;
}iPARK_IQ;
```

- **Floating Point (F)**

```
//*********** Structure Definition ********//
typedef struct{
       float32 alpha;
       float32 beta;
       float32 zero;
       float32 sin;
       float32 cos;
       float32 d;
       float32 q;
       float32 z;
}iPARK_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//*********** Structure Definition ********//
typedef struct{
      float32 alpha;
      float32 beta;
      float32 zero;
      float32 sin;
      float32 cos;
      float32 d;
      float32 q;
      float32 z;
}iPARK_CLA;
```

Module interface Definition:

Module Element Name	Type	Description	Acceptable Range
alpha,beta	Output	ABC component on the stationary reference frame Alpha, Beta	IQ(-1,1) Float(-1,1)
sin,cos	Input	Sin and cos of the grid angle	IQ Float(-1,1)
d,q,z	Input	Rotating Reference Frame Quantities	IQ Float(-1,1)

Usage:

- **Fixed Point (IQ)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
iPARK_IQ ipark1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
iPARK_IQ_init(&ipark1);
```

Step 4 – **Using the module**

```
ipark1.alpha = Valpha;
ipark1.beta = Vbeta;
ipark1.zero = Vzero;
ipark1.sin = sin(theta);
ipark1.cos = cos(theta;
```

```
iPARK_IQ_FUNC(&ipark1);
```

Alternatively the macro routine can be called as bellow:

```
iPARK_IQ_MACRO(ipark1);
```

- **Floating Point (F)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
iPARK_F park1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
iPARK_F_init(&ipark1);
```

Step 4 – Using the module

```
ipark1.alpha = Valpha;
ipark1.beta = Vbeta;
ipark1.zero = Vzero;
ipark1.sin = sin(theta);
ipark1.cos = cos(theta);
iPARK_F_FUNC(&ipark1);
```

Alternatively the macro routine can be called as bellow:

```
iPARK_F_MACRO(ipark1);
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(ipark1,"CLADataRAM")
ClaToCpu_Volatile iPARK_CLA park1;
```

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

```
extern ClaToCpu_Volatile iPARK_CLA ipark1;
```

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
      ….
      iPARK_CLA_init(ipark1);
      ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

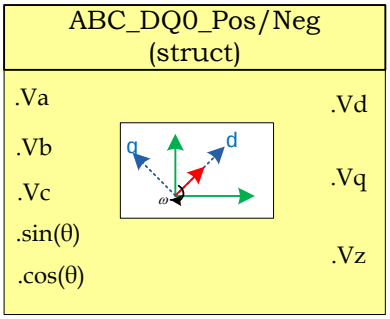
```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
      ….
      // Write per unit values of voltage/current
      ipark1.alpha = Valpha;
      ipark1.beta = Vbeta;
      ipark1.zero = Vzero;
      ipark1.sin = sin(theta);
      ipark1.cos = cos(theta);
      // Invoke the run time macro
      iPARK_CLA_MACRO(ipark1);
      ….
```

```
}
```

Description: This software module implements the transform from ABC to DQ0 axis.



Module File: <*base_folder*>\ ABC_ DQ0_ (POS/NEG)_(IQ/F/CLA).h

Technical: The block converts the three phase quantities into DC quantities in rotating reference frame for positive and negative sequence.

$$v_{dq+} = T_{abc->dq0+} V_{abc} = T_{abc->\alpha\beta} * T_{\alpha\beta->dq} * V_{abc}$$

Where:

$$T_{abc->\alpha\beta} = \frac{2}{3}\begin{bmatrix} 1 & \cos(2\pi/3) & \cos(4\pi/3) \\ 0 & \sin(2\pi/3) & \sin(4\pi/3) \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

and

$$T_{abc->dq0+} = \begin{bmatrix} v_d \\ v_q \\ v_o \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} X \begin{bmatrix} v_\alpha \\ v_\beta \\ v_o \end{bmatrix}$$

and

$$T_{abc->dq0-} = \begin{bmatrix} v_d \\ v_q \\ v_o \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} X \begin{bmatrix} v_\alpha \\ v_\beta \\ v_o \end{bmatrix}$$

Object Definition:

- **Fixed Point (IQ)**

```
//*********** Structure Definition ********//
typedef struct{
	int32 a;
	int32 b;
	int32 c;
	int32 alpha;
	int32 beta;
	int32 sin;
	int32 cos;
	int32 d;
	int32 q;
	int32 z;
}ABC_DQ0_POS_IQ;

//*********** Structure Definition ********//
typedef struct{
	int32 a;
	int32 b;
	int32 c;
	int32 alpha;
	int32 beta;
	int32 sin;
	int32 cos;
	int32 d;
	int32 q;
	int32 z;
}ABC_DQ0_NEG_IQ;
```

- **Floating Point (F)**

```
//*********** Structure Definition ********//
typedef struct{
	float32 a;
	float32 b;
	float32 c;
	float32 alpha;
	float32 beta;
	float32 sin;
	float32 cos;
	float32 d;
	float32 q;
	float32 z;
}ABC_DQ0_POS_F;

//*********** Structure Definition ********//
typedef struct{
	float32 a;
	float32 b;
	float32 c;
	float32 alpha;
	float32 beta;
```

```
        float32 sin;
        float32 cos;
        float32 d;
        float32 q;
        float32 z;
}ABC_DQ0_NEG_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//*********** Structure Definition ********//
typedef struct{
        float32 a;
        float32 b;
        float32 c;
        float32 alpha;
        float32 beta;
        float32 sin;
        float32 cos;
        float32 d;
        float32 q;
        float32 z;

}ABC_DQ0_POS_CLA;


//*********** Structure Definition ********//
typedef struct{
        float32 a;
        float32 b;
        float32 c;
        float32 alpha;
        float32 beta;
        float32 sin;
        float32 cos;
        float32 d;
        float32 q;
        float32 z;

}ABC_DQ0_NEG_CLA;
```

Module interface Definition:

Module Element Name	Type	Description	Acceptable Range
a,b,c	Input	3ph AC Signal measured and normalized	IQ(-1,1) Float32(-1,1)
Sin,Cos	Input	Sin and cos of the grid angle	IQ Float32
d,q,z	Output	Positive and Negative DQ component	IQ Float32

Usage:

- **Fixed Point (IQ)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
ABC_DQ0_POS_IQ abc_dq0_pos1;
ABC_DQ0_NEG_IQ abc_dq0_neg1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
ABC_DQ0_POS_IQ_init(&abc_dq0_pos1);
ABC_DQ0_NEG_IQ_init(&abc_dq0_neg1);
```

Step 4 – **Using the module**

```
// ABC_DQ0_POS_IQ
    abc_dq0_pos1.a = GridMeas1;
    abc_dq0_pos1.b = GridMeas2;
    abc_dq0_pos1.c = GridMeas3;
    abc_dq0_pos1.sin = sin(theta);
    abc_dq0_pos1.cos = cos(theta);
    ABC_DQ0_POS_IQ_FUNC(&abc_dq0_pos1);
```

Alternatively the macro routine can be called as bellow:

```
    ABC_DQ0_POS_IQ_MACRO(abc_dq0_pos1);
```

```
// ABC_DQ0_NEG_IQ
    abc_dq0_neg1.a = GridMeas1;
    abc_dq0_neg1.b = GridMeas2;
    abc_dq0_neg1.c = GridMeas3;
    abc_dq0_neg1.sin = sin(theta);
    abc_dq0_neg1.cos = cos(theta);
    ABC_DQ0_NEG_IQ_FUNC(&abc_dq0_neg1);
```

Alternatively the macro routine can be called as bellow:

```
    ABC_DQ0_NEG_IQ_MACRO(abc_dq0_neg1);
```

- **Floating Point (F)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
ABC_DQ0_POS_F abc_dq0_pos1;
ABC_DQ0_NEG_F abc_dq0_neg1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
ABC_DQ0_POS_F_init(&abc_dq0_pos1);
ABC_DQ0_NEG_F_init(&abc_dq0_neg1);
```

Step 4 – Using the module

```
// ABC_DQ0_POS_F
abc_dq0_pos1.a = GridMeas1;
abc_dq0_pos1.b = GridMeas2;
abc_dq0_pos1.c = GridMeas3;
abc_dq0_pos1.sin = sin(theta);
abc_dq0_pos1.cos = cos(theta);
ABC_DQ0_POS_F_FUNC(&abc_dq0_pos1);
```

Alternatively the macro routine can be called as bellow:

```
ABC_DQ0_POS_F_MACRO(abc_dq0_pos1);
```

```
// ABC_DQ0_NEG_F
abc_dq0_neg1.a = GridMeas1;
abc_dq0_neg1.b = GridMeas2;
abc_dq0_neg1.c = GridMeas3;
abc_dq0_neg1.sin = sin(theta);
abc_dq0_neg1.cos = cos(theta);
ABC_DQ0_NEG_F_FUNC(&abc_dq0_neg1);
```

Alternatively the macro routine can be called as bellow:

```
ABC_DQ0_NEG_F_MACRO(abc_dq0_neg1);
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(abc_dq0_pos1, "CLADataRAM")
ClaToCpu_Volatile ABC_DQ0_POS_CLA abc_dq0_pos1;
#pragma DATA_SECTION(abc_dq0_neg1, "CLADataRAM")
ClaToCpu_Volatile ABC_DQ0_NEG_CLA abc_dq0_neg1;
```

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

```
extern ClaToCpu_Volatile ABC_DQ0_POS_CLA abc_dq0_pos1;
extern ClaToCpu_Volatile ABC_DQ0_NEG_CLA abc_dq0_neg1;
```

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
      ….
      ABC_DQ0_POS_CLA_init(&abc_dq0_pos1);
      ABC_DQ0_NEG_CLA_init(&abc_dq0_neg1);
      ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

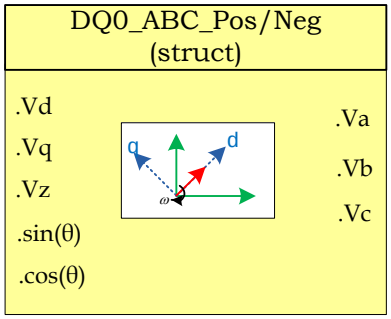
```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
        ….
        abc_dq0_neg1.a = (cos_temp);
        abc_dq0_neg1.c = (cos_temp2);
        abc_dq0_neg1.b = (cos_temp3);
        abc_dq0_neg1.sin = sin_ramp;
        abc_dq0_neg1.cos = cos_ramp;
        ABC_DQ0_NEG_CLA_MACRO(abc_dq0_neg1);

        dq0_abc1.d = abc_dq0_pos1.d;
        dq0_abc1.q = abc_dq0_pos1.q;
        dq0_abc1.z = abc_dq0_pos1.z;
        dq0_abc1.sin = sin_ramp;
        dq0_abc1.cos = cos_ramp;
        DQ0_ABC_CLA_MACRO(dq0_abc1);
        ….
}
```

DQ0_ABC_Pos/Neg — *DQ0 to ABC Transform for positive and negative sequence*

Description: This software module implements the transform from DQ0 to ABC for positive and negative rotating vectors.



Module File: <*base_folder*>\DQ0_ABC _(IQ/F/CLA).h

Technical: The block converts the stationary frame quantities into three phase quantities

$$V_{abc} = T_{dq0+->abc}\, v_{dqo+}$$

Also

$$V_{abc} = T_{dq0-->abc}\, v_{dqo-}$$

Object Definition:

- **Fixed Point (IQ)**

```
//*********** Structure Definition ********//
typedef struct{
      int32 a;
      int32 b;
      int32 c;
      int32 alpha;
      int32 beta;
      int32 sin;
      int32 cos;
      int32 d;
      int32 q;
      int32 z;
}DQ0_ABC_IQ;
```

- **Floating Point (F)**

```
//*********** Structure Definition ********//
typedef struct{
```

```
        float32 a;
        float32 b;
        float32 c;
        float32 alpha;
        float32 beta;
        float32 sin;
        float32 cos;
        float32 d;
        float32 q;
        float32 z;
}DQ0_ABC_F;
```

- **Control Law Accelerated Floating point (CLA)**

```
//*********** Structure Definition ********//
typedef struct{
        float32 a;
        float32 b;
        float32 c;
        float32 alpha;
        float32 beta;
        float32 sin;
        float32 cos;
        float32 d;
        float32 q;
        float32 z;
}DQ0_ABC_CLA;
```

Module interface Definition:

Module Element Name	Type	Description	Acceptable Range
a,b,c	Output	3ph AC Signal measured and normalized	IQ(-1,1) Float 32(-1,1)
Sin,Cos	Input	Sin and cos of the grid angle	IQ Float32(-1,1)
d,q,,z	Input	Positive and Negative DQ component	IQ Float32(-1,1)

Usage:

- **Fixed Point (IQ)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
DQ0_ABC_IQ dq0_abc1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
DQ0_ABC_IQ_init(&dq0_abc1);
```

Step 4 – **Using the module**

```
dq0_abc1.d =Vd;
dq0_abc1.q =Vq;
dq0_abc1.z =Vz;
dq01_abc.sin =sin(theta);
dq01_abc.cos =cos(theta);

DQ0_ABC_FUNC(&dq0_abc1);
```

Alternatively the macro routine can be called as bellow:

```
DQ0_ABC_MACRO(dq0_abc1);
```

- **Floating Point (F)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
DQ0_ABC_F dq0_abc1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
DQ0_ABC_F_init(&dq0_abc1);
```

Step 4 – Using the module

```
dq0_abc1.d =Vd;
dq0_abc1.q =Vq;
dq0_abc1.z =Vz;
dq01_abc.sin =sin(theta);
dq01_abc.cos =cos(theta);

DQ0_ABC_FUNC(&dq0_abc1);
```

Alternatively the macro routine can be called as bellow:

```
DQ0_ABC_MACRO(dq0_abc1);
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(dq0_abc1,"CLADataRAM")
ClaToCpu_Volatile DQ0_ABC_CLA dq0_abc1;
```

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

```
extern ClaToCpu_Volatile DQ0_ABC_CLA dq0_abc1;
```

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
      ….
      DQ0_ABC_CLA_init(dq0_abc1);
      ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

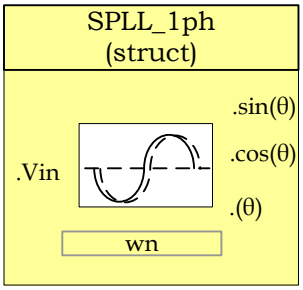
```
interrupt void Cla1Task1(void) {
        ….
        // Write per unit values of voltage/current
        dq0_abc1.d =Vd;
        dq0_abc1.q =Vq;
        dq0_abc1.z =Vz;
        dq01_abc.sin =sin(theta);
        dq01_abc.cos =cos(theta);
        // Invoke the run time macro
        DQ0_ABC_MACRO(dq0_abc1);
        ….
}
```

4.3 Phase Locked Loop Modules

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. PLL for single phase and three phase grid connection are discussed in the below section:

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.



Module File: <*base_folder*>\SPLL_1ph _(IQ/F/CLA).h

Technical: A functional diagram of a PLL is shown in the Figure 6, which consists of a phase detect(PD), a loop filter(LPF) and a voltage controlled oscillator(VCO)

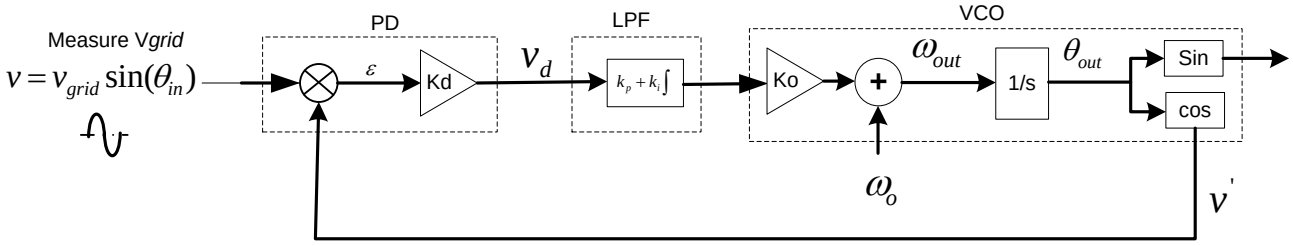


Figure 6 Phase Lock Loop Basic Structure

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

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

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

$$\omega_n = \sqrt{\frac{v_{grid} K_p}{T_i}} \qquad \zeta = \sqrt{\frac{v_{grid} T_i K_p}{4}}$$

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. An adaptive notch filter is used to selectively notch the exact frequency in case there are variations in the grid frequency.

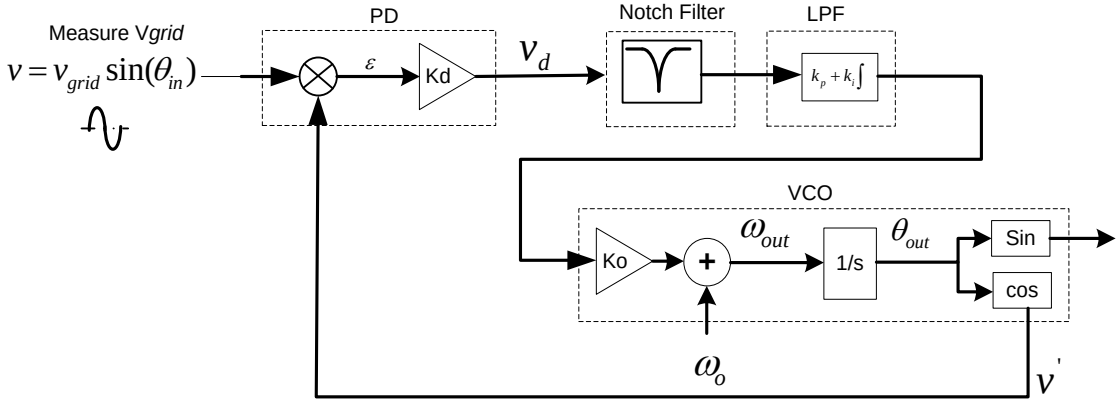


Figure 7 Single Phase PLL with Notch Filter

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 Loop Filter or PI is implemented as a digital controller with the following equation

$$ylf[n] = ylf[n-1] * A1 + ynotch[n] * B0 + ynotch[n-1] * B1$$

Using Z transform this equation can be written as

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

Now the laplace transform of the loop filter from the analog domain is

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

Now using Bi-linear transformation $s = \frac{2}{T}(\frac{z-1}{z+1})$ =>

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

where T= Sampling Time

Now the question is how to select an appropriate value for the proportional and integral gain. Now the step response to a general second order equation i.e.

$H(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$ is given as: $y(t) = 1 - ce^{-\sigma t}\sin(\omega_d t + \phi)$

Ignoring the LHP zero from the above equation. Now settling time is given as the time it takes for the response to to settle between an error band, let's say this error is ∂, then

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

Where $\sigma = \varsigma\omega_n$ and $c = \frac{\omega_n}{\omega_d}$ and $\omega_d = \sqrt{1 - \varsigma^2}\,\omega_n$

Now using settling time to be 30ms and the error band to be 5% and damping ratio to be 0.7 we can obtain the natural frequency to be 119.014 and then back substituting we get $K_p = 166.6$ and $K_i = 27755.55$

Back substituting these values into the digital loop filter coefficients we get:

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

Now for 50Khz run rate for the PLL, B0= 166.877556 and B1= -166.322444

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 7.

Design of the notch filter is achieved using discretizing the notch filter equation from s domain:

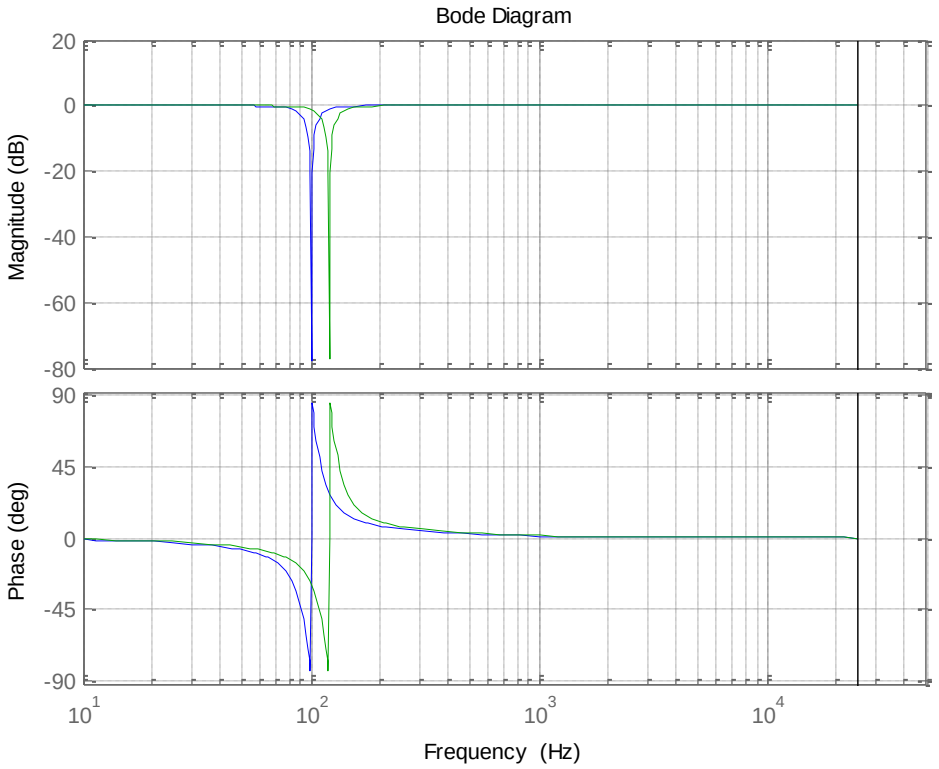
$$H_{nf}(s) = \frac{s^2 + 2\zeta_2 \omega_n s + \omega_n^2}{s^2 + 2\zeta_1 \omega_n s + \omega_n^2} \text{ where } \zeta_2 << \zeta_1$$

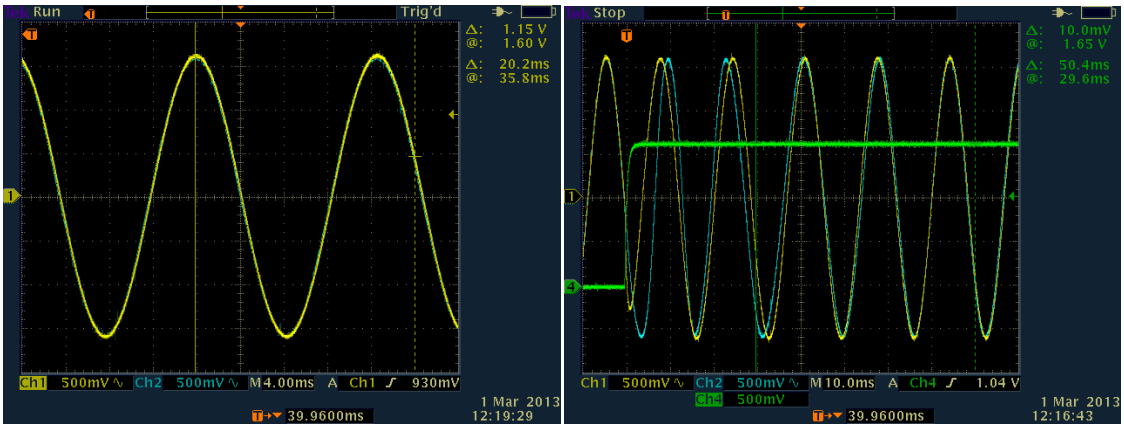
Using zero order hold i.e. $s = \frac{(z-1)}{T}$ we get

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

Hence the coefficients for the notch filter can be adaptively changed as the grid frequency varies. The coefficients are chosen such that attenuation at twice the grid frequency is steep without affecting other frequencies.

Taking $\zeta_2 << \zeta_1$, and ζ_2=0.00001 and ζ_1=0.1, the response of the notch is as shown below for 50 and 60Hz grid where the coefficients are calculated based on the grid frequency estimate.





Object Definition:

- **Fixed Point (IQ)**

```
//************ Structure Definition ********//
typedef struct{
      int32 B2_notch;
      int32 B1_notch;
      int32 B0_notch;
      int32 A2_notch;
      int32 A1_notch;
}SPLL_1ph_IQ_NOTCH_COEFF;
```

```
typedef struct{
      int32 B1_lf;
      int32 B0_lf;
      int32 A1_lf;
}SPLL_1ph_IQ_LPF_COEFF;

typedef struct{
      int32 AC_input;
      int32 theta[2];
      int32 cos[2];
      int32 sin[2];
      int32 wo;
      int32 wn;

      SPLL_1ph_IQ_NOTCH_COEFF notch_coeff;
      SPLL_1ph_IQ_LPF_COEFF   lpf_coeff;

      int32 Upd[3];
      int32 ynotch[3];
      int32 ylf[2];
      int32 delta_t;
}SPLL_1ph_IQ;
```

- **Floating Point (F)**

```
//************ Structure Definition ********//
typedef struct{
      float32 B2_notch;
      float32 B1_notch;
      float32 B0_ntoch;
      float32 A2_notch;
      float32 A1_notch;
}SPLL_1ph_F_NOTCH_COEFF;

typedef struct{
      float32 B1_lf;
      float32 B0_lf;
      float32 A1_lf;
}SPLL_1ph_F_LPF_COEFF;

typedef struct{
      float32 AC_input;
      float32 theta[2];
      float32 cos[2];
      float32 sin[2];
      float32 wo;
      float32 wn;

      SPLL_1ph_F_NOTCH_COEFF notch_coeff;
      SPLL_1ph_F_LPF_COEFF lpf_coeff;

      float32 Upd[3];
      float32 ynotch[3];
      float32 ylf[2];
      float32 delta_t;
```

```
}SPLL_1ph_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//*********** Structure Definition ********//
typedef struct{
      float32 B2_notch;
      float32 B1_notch;
      float32 B0_notch;
      float32 A2_notch;
      float32 A1_notch;
}SPLL_1ph_CLA_NOTCH_COEFF;

typedef struct{
      float32 B1_lf;
      float32 B0_lf;
      float32 A1_lf;
}SPLL_1ph_CLA_LPF_COEFF;

typedef struct{
      float32 AC_input;
      float32 theta[2];
      float32 cos[2];
      float32 sin[2];
      float32 wo;
      float32 wn;

      SPLL_1ph_CLA_NOTCH_COEFF notch_coeff;
      SPLL_1ph_CLA_LPF_COEFF lpf_coeff;

      float32 Upd[3];
      float32 ynotch[3];
      float32 ylf[2];
      float32 delta_t;
}SPLL_1ph_CLA;
```

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	IQ21(-1,1) Float32(-1,1)
Wn	Input	Grid Frequency in radians/sec	IQ21 Float32

theta[2]	Output	grid phase angle	IQ21 Float32
cos[2]	Output	Cos(grid phase angle)	IQ21 Float32
sin[2]	Output	Sin(grid phase angle)	IQ21 Float32
Wo	Internal	Instantaneous Grid Frequency in radians/sec	IQ21 Float32
notch_coeff	Internal	Notch Filter Coefficients	IQ21 Float32
lpf_coeff	Internal	Loop Filer Coefficients	IQ21 Float32
Upd[3]	Internal	Internal Data Buffer for phase detect output	IQ21 Float32
ynotch[3]	Internal	Internal Data Buffer for the notch output	IQ21 Float32
ylf	Internal	Internal Data Buffer for Loop Filter output	IQ21 Float32
delta_t	Internal	1/Frequency of calling the PLL routine	IQ21 Float32

Usage:

- **Fixed Point (IQ)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
SPLL_1ph_IQ spll1;
SPLL_1ph_IQ_NOTCH_COEFF spll_notch_coef1;
```

Step 3 – **Initialize module** in {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. Ensure that the ISR is running at a minimum frequency of 20KHz. 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_IQ_init(GRID_FREQ,_IQ21((float)(1.0/ISR_FREQUENCY)
),&spll1);
SPLL_1ph_IQ_notch_coeff_update(((float)(1.0/ISR_FREQUENCY))
,(float)(2*PI*GRID_FREQ*2),(float)0.00001,(float)0.1,&spll1
);
```

Step 4 – Using the module

```
spll1.AC_input =(long)(GridMeas>>3);
```

```
// GridMeas is in IQ24 and is converted to IQ21 for SPLL
// SPLL call
SPLL_1ph_IQ_FUNC(&spll1);
```

Alternatively the macro routine can be called as bellow:

```
SPLL_1ph_IQ_MACRO(spll1);
```

- **Floating Point (F)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
SPLL_1ph_F spll1;
SPLL_1ph_F_NOTCH_COEFF spll_notch_coef1;
```

Step 3 – **Initialize module** in {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. Ensure that the ISR is running at a minimum frequency of 20KHz. 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_F_init(GRID_FREQ,((float)(1.0/ISR_FREQUENCY)),
&spll1);
SPLL_1ph_F_notch_coeff_update(((float)(1.0/ISR_FREQUENCY)),
(float)(2*PI*GRID_FREQ*2),(float)0.00001,(float)0.1,
&spll1);
```

Step 4 – **Using the module**

```
spll1.AC_input = (Float32)GridMeas;
// SPLL call
SPLL_1ph_F_FUNC(&spll1);
```

Alternatively the macro routine can be called as bellow:

```
SPLL_1ph_F_MACRO(spll1);
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

```
#pragma DATA_SECTION(spll1,"CLADataRAM")
ClaToCpu_Volatile SPLL_1ph_CLA 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 spll1;
```

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.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. Ensure that the ISR is running at a minimum frequency of 20KHz. The routine initializes all the internal data buffers and variables, and sets the coefficients of the notch filter according to the grid frequency. – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
    ….
    SPLL_1ph_CLA_init(GRID_FREQUENCY,
    ((float32)(1.0/ISR_FREQUENCY)), spll1);
    SPLL_1ph_CLA_notch_coeff_update(((float32)(1.0/ISR_FR
    EQUENCY)), (float32)(2*PI*GRID_FREQUENCY*2),
    (float32)0.00001, (float32)0.1, spll1);
    ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

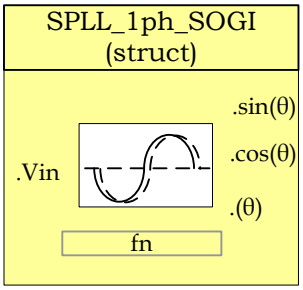
```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
```

```
        ….
        spll1.AC_input = (Float32)GridMeas;
        // Invoke the run time macro
        SPLL_1ph_CLA_MACRO(spll1);
        ….
}
```

Description: This software module implements a software phase lock loop based on orthogonal signal generation using second order generalized integrators. This block can be further used in P and Q style control for single phase grid connected equipment.



Module File: <*base_folder*>\SPLL_1ph _SOGI_(IQ/F/CLA).h

Technical: As discussed in the SPLL_1ph single phase grid software PLL design is tricky because of twice the grid frequency component present in the phase detect output. Notch filter is used in the SPLL_1ph implementation for this and achieves satisfactory results. Another alternative is to use an orthogonal signal generator scheme and then use park transformation. This software block uses a second order integrator to generate the orthogonal signal from the sensed single phase grid voltage (As proposed in 'A New Single Phase PLL Structure Based on Second Order Generalized Integrator', Mihai Ciobotaru, PESC'06).

A functional diagram of a PLL is shown in the figure below, which consists of a phase detect (PD) consisting of park transform, a loop filter(LPF) and a voltage controlled oscillator(VCO)

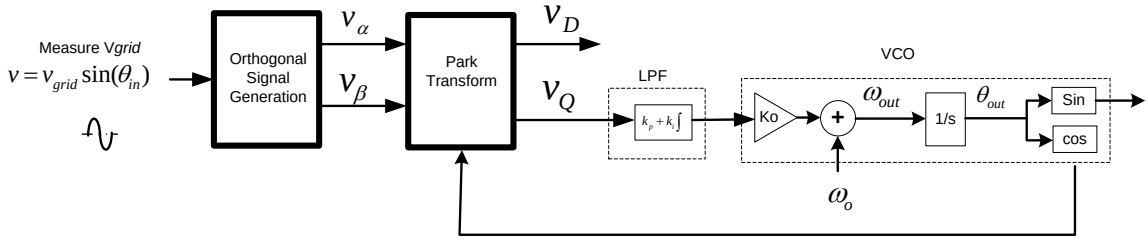


Figure 8 Phase Lock Loop Basic Structure

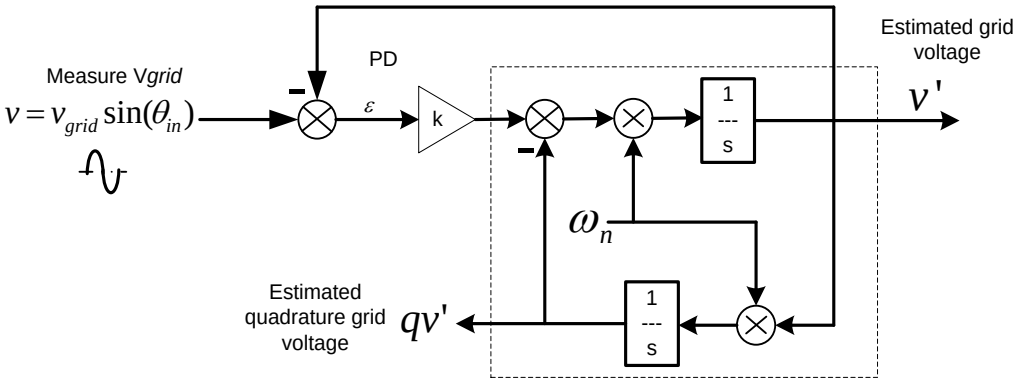


Figure 9 Second Order Generalized integrator for Orthogonal Signal Generation

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

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

For discrete implementation using trapezoidal approximation:

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

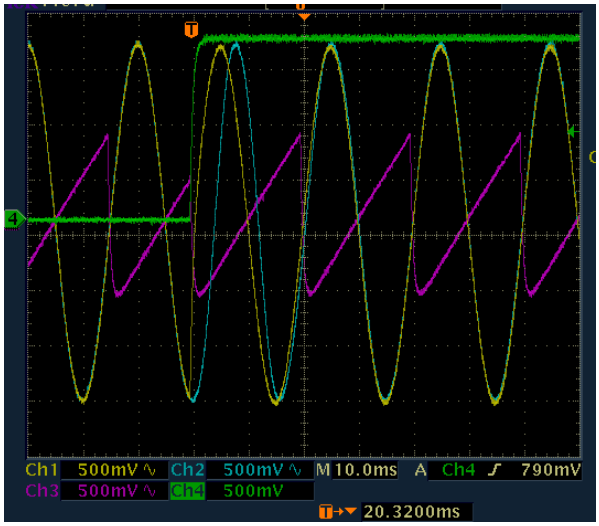
Now using $x = 2k\omega_n T_s$ and $y = (\omega_n T_s)^2$:

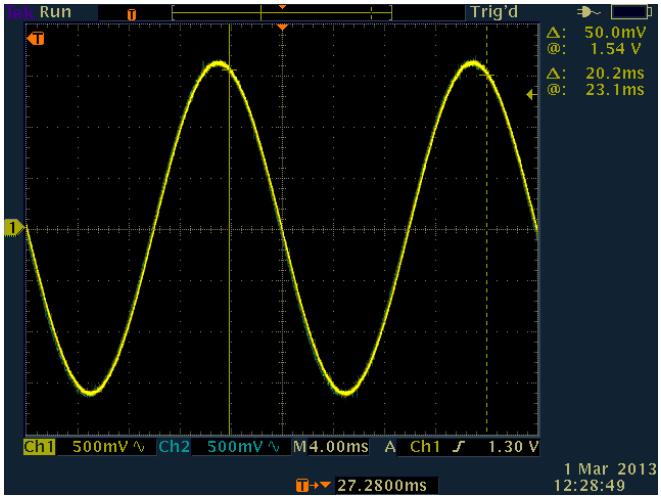
$$H_d(z) = \frac{\left(\dfrac{x}{x+y+4}\right) + \left(\dfrac{-x}{x+y+4}\right)z^{-2}}{1 - \left(\dfrac{2(4-y)}{x+y+4}\right)z^{-1} - \left(\dfrac{x-y-4}{x+y+4}\right)z^{-2}} = \frac{b_0 + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}}$$

Similarly

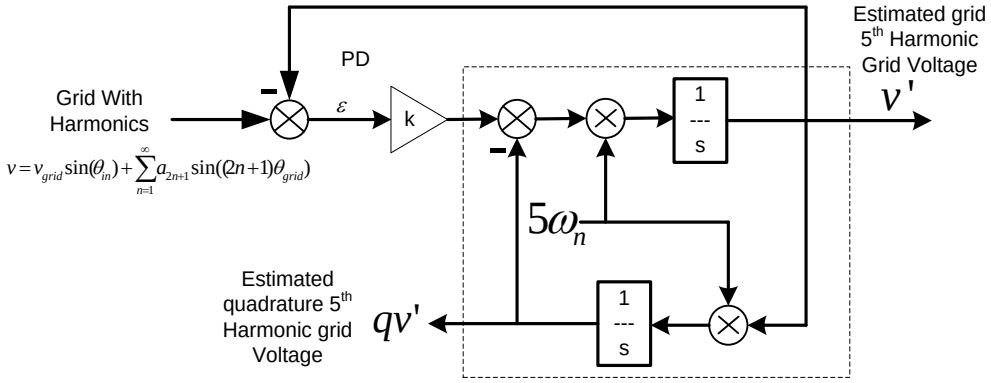
$$H_q(z) = \frac{\left(\dfrac{k.y}{x+y+4}\right) + 2\left(\dfrac{k.y}{x+y+4}\right)z^{-1} + \left(\dfrac{k.y}{x+y+4}\right)z^{-2}}{1 - \left(\dfrac{2(4-y)}{x+y+4}\right)z^{-1} - \left(\dfrac{x-y-4}{x+y+4}\right)z^{-2}} = \frac{qb_0 + qb_1 z^{-1} + qb_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}}$$

Once the orthogonal signal is generated park transform is used to detect the Q and D components on the rotating reference frame. This is then fed to the loop filter which controls the VCO of the PLL. The tuning of the loop filter is similar to what is described in the SPLL_1ph description. Very fast transient response is possible as shown in the figure below where Ch1: is the grid sine value ch2: is the PLL lock ch3: is the grid theta and ch4: is the phase jump.





The coefficients of the orthogonal signal generator can be tuned for varying grid frequency and sampling time (ISR frequency). The only variable is k, which determines the selectiveness of the frequency for the second order integrator. The second order generalized integrator presented can be also modified to extract the harmonic frequency component if needed in a grid monitoring application. A lower k value must be selected for this purpose, however lower k has an effect slowing the response. The figure below shows the extraction of the 5^{th} harmonic using the SOGI. The implementation of this is left for the user as it directly follows from the SOGI implementation shown in SPLL_1ph_SOGI.



Additionally the RMS voltage of the grid can also be estimated using the below equation:

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

Object Definition:

```
#define SPLL_SOGI_Q _IQ23
#define SPLL_SOGI_Qmpy _IQ23mpy
```

- **Fixed Point (IQ)**

```
//*********** Structure Definition ********//
typedef struct{
      int32 osg_k;
      int32 osg_x;
      int32 osg_y;
      int32 osg_b0;
      int32 osg_b2;
      int32 osg_a1;
      int32 osg_a2;
      int32 osg_qb0;
      int32 osg_qb1;
      int32 osg_qb2;
}SPLL_1ph_SOGI_IQ_OSG_COEFF;

typedef struct{
      int32 B1_lf;
      int32 B0_lf;
      int32 A1_lf;
}SPLL_1ph_SOGI_IQ_LPF_COEFF;

typedef struct{
      int32 u[3];  // Ac Input
      int32 osg_u[3];
      int32 osg_qu[3];
      int32 u_Q[2];
      int32 u_D[2];
      int32 ylf[2];
```

```
        int32 fo; // output frequency of PLL
        int32 fn; //nominal frequency
        int32 theta[2];
        int32 cos;
        int32 sin;
        int32 delta_T;
        SPLL_1ph_SOGI_IQ_OSG_COEFF osg_coeff;
        SPLL_1ph_SOGI_IQ_LPF_COEFF lpf_coeff;
}SPLL_1ph_SOGI_IQ;
```

- **Floating Point (F)**

```
//*********** Structure Definition ********//
typedef struct{
        float32 osg_k;
        float32 osg_x;
        float32 osg_y;
        float32 osg_b0;
        float32 osg_b2;
        float32 osg_a1;
        float32 osg_a2;
        float32 osg_qb0;
        float32 osg_qb1;
        float32 osg_qb2;
}SPLL_1ph_SOGI_F_OSG_COEFF;

typedef struct{
        float32 B1_lf;
        float32 B0_lf;
        float32 A1_lf;
}SPLL_1ph_SOGI_F_LPF_COEFF;

typedef struct{
        float32 u[3];  // Ac Input
        float32 osg_u[3];
        float32 osg_qu[3];
        float32 u_Q[2];
        float32 u_D[2];
        float32 ylf[2];
        float32 fo; // output frequency of PLL
        float32 fn; //nominal frequency
        float32 theta[2];
        float32 cos;
        float32 sin;
        float32 delta_T;
        SPLL_1ph_SOGI_F_OSG_COEFF osg_coeff;
        SPLL_1ph_SOGI_F_LPF_COEFF lpf_coeff;
}SPLL_1ph_SOGI_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//*********** Structure Definition ********//
typedef struct{
        float32 osg_k;
        float32 osg_x;
        float32 osg_y;
```

```
        float32 osg_b0;
        float32 osg_b2;
        float32 osg_a1;
        float32 osg_a2;
        float32 osg_qb0;
        float32 osg_qb1;
        float32 osg_qb2;
}SPLL_1ph_SOGI_CLA_OSG_COEFF;

typedef struct{
        float32 B1_lf;
        float32 B0_lf;
        float32 A1_lf;
}SPLL_1ph_SOGI_CLA_LPF_COEFF;

typedef struct{
        float32 u[3];  // Ac Input
        float32 osg_u[3];
        float32 osg_qu[3];
        float32 u_Q[2];
        float32 u_D[2];
        float32 ylf[2];
        float32 fo; // output frequency of PLL
        float32 fn; //nominal frequency
        float32 theta[2];
        float32 cos;
        float32 sin;
        float32 delta_T;
        SPLL_1ph_SOGI_CLA_OSG_COEFF osg_coeff;
        SPLL_1ph_SOGI_CLA_LPF_COEFF lpf_coeff;
}SPLL_1ph_SOGI_CLA;
```

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_SOGI 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
u[3]	Input	1ph AC Signal measured and normalized	IQ23(-1,1) Float32(-1,1)
osg_u[3]	Internal	Estimated Grid Voltage	IQ23(-1,1) Float32(-1,1)
osg_qu[3]	Internal	Estimated Orthogonal Grid Voltage	IQ23(-1,1) Float32(-1,1)
u_Q[2]	Output	Q axis component of the estimated grid	IQ23(-1,1)

			Float32(-1,1)
u_D[2]	Output	D axis component of the estimated grid	IQ23(-1,1) Float32(-1,1)
Fn	Input	Grid Frequency Nominal	IQ23 Float32
Fo	Output	Instantaneous Grid Frequency	IQ23 Float32
theta[2]	Output	grid phase angle	IQ23 Float32
Cos	Output	Cos(grid phase angle)	IQ23(-1,1) Float32(-1,1)
Sin	Output	Sin(grid phase angle)	IQ23(-1,1) Float32(-1,1)
lpf_coeff	Internal	Loop filter coefficients	IQ23 Float32
osg_coeff	Internal	OSG coefficients	IQ23 Float32
delta_t	Internal	1/Frequency of calling the PLL routine	IQ23 Float32

Usage:

- **Fixed Point (IQ)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
SPLL_1ph_SOGI_IQ spll1;
```

Step 3 – **Initialize module** in {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. Ensure that the ISR is running at a minimum frequency of 20KHz. 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_SOGI_IQ_init(GRID_FREQ,_IQ23((float)(1.0/ISR_FREQU
ENCY)),&spll1);
SPLL_1ph_SOGI_IQ_coeff_update(((float)(1.0/ISR_FREQUENCY)),
(float)(2*PI*GRID_FREQ),&spll1);
```

Step 4 – **Using the module**

```
spll1.u[0]=(long)(GridMeas>>1);
// GridMeas is in IQ24 and is converted to IQ23 for SPLL
// SPLL call
SPLL_1ph_SOGI_IQ_FUNC(&spll1);
```

Alternatively the macro routine can be called as bellow:

```
SPLL_1ph_SOGI_IQ_MACRO(spll1);
```

- **Floating Point (F)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
SPLL_1ph_SOGI_F spll1;
```

Step 3 – **Initialize module** in {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. Ensure that the ISR is running at a minimum frequency of 20KHz. 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_SOGI_F_init(GRID_FREQ,((float)(1.0/ISR_FREQUENCY))
,&spll2);
SPLL_1ph_SOGI_F_coeff_update(((float)(1.0/ISR_FREQUENCY)),
(float)(2*PI*GRID_FREQ),&spll2);
```

Step 4 – Using the module

```
spll1.u[0]=(Float32)(GridMeas);
// SPLL call
SPLL_1ph_SOGI_F_FUNC(&spll2);
```

Alternatively the macro routine can be called as bellow:

```
SPLL_1ph_SOGI_F_MACRO(spll2);
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(spll1,"CLADataRAM")
ClaToCpu_Volatile SPLL_1ph_SOGI_CLA 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_SOGI_CLA spll1;
```

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.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. Ensure that the ISR is running at a minimum frequency of 20KHz. The routine initializes all the internal data buffers and variables, and sets the coefficients of the notch filter according to the grid frequency. – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
    ….
    SPLL_1ph_SOGI_CLA_init(GRID_FREQ,((float32)(1.0/ISR_F
    REQUENCY)),spll1);
    SPLL_1ph_SOGI_CLA_coeff_update(((float32)(1.0/ISR_FRE
    QUENCY)),(float32)(2*PI*GRID_FREQ),spll1);
    ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

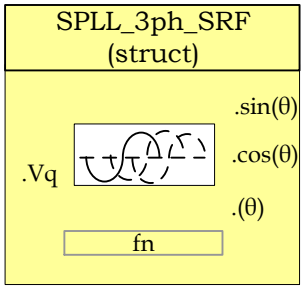
```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
    ….
    Spll1.u[0] = (Float32)(GridMeas);;
    // SPLL call
```

```
        SPLL_1ph_SOGI_CLA_MACRO(spll1);
        ….
}
```

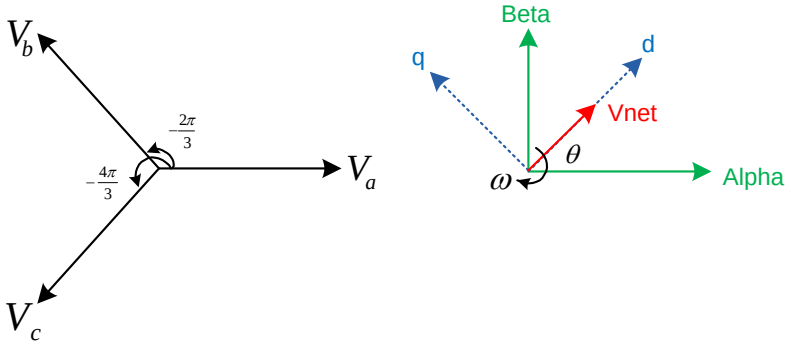
Description: This software module implements a software phase lock loop based on synchronous reference frame for grid connection to three phase grid.



Module File: <*base_folder*>\SPLL_3ph _SRF_(IQ/F/CLA).h

Technical: It is common to transform three phase time varying system to a dc system, in a rotating reference frame with the help of transforms. Assuming the below equation for the three phase quantities the sequence of the voltages is $V_a \rightarrow V_c \rightarrow V_b$, and the frequency is ω.

$$\begin{bmatrix} V_a \\ V_b \\ V_c \end{bmatrix} = V \begin{bmatrix} \cos(\omega t) \\ \cos(\omega t - 2\pi/3) \\ \cos(wt - 4\pi/3) \end{bmatrix}$$



The three phase quantities are first reduced to an orthogonal component system (alpha, beta also called stationary reference frame), by taking the projections of the three phase quantities on the orthogonal axis. This is called the clark transform:

$$V_{\alpha\beta 0} = T_{abc \rightarrow \alpha\beta 0} V_{abc}$$

$$\begin{bmatrix} V_\alpha \\ V_\beta \\ V_o \end{bmatrix} = \sqrt{\frac{2}{3}} \begin{bmatrix} 1 & \cos(2\pi/3) & \cos(4\pi/3) \\ 0 & \sin(2\pi/3) & \sin(4\pi/3) \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} X \begin{bmatrix} V_a \\ V_b \\ V_c \end{bmatrix} = \sqrt{\frac{2}{3}} \begin{bmatrix} \cos(\omega t) \\ \sin(\omega t) \\ 0 \end{bmatrix}$$

Now the net voltage vector can be assumed to be making an angle θ with the orthogonal reference frame and rotating at a frequency of ω. Thus the system can be reduced to DC by taking projection of the orthogonal components on the rotating reference frame:

$$V_{dq0} = T_{\alpha\beta 0 \to dq0} V_{\alpha\beta 0}$$

$$\begin{bmatrix} v_d \\ v_q \\ v_o \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} X \begin{bmatrix} v_\alpha \\ v_\beta \\ v_o \end{bmatrix}$$

$$\begin{bmatrix} v_d \\ v_q \\ v_o \end{bmatrix} = \sqrt{\frac{2}{3}} \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} X \begin{bmatrix} \cos(\omega t) \\ \sin(\omega t) \\ 0 \end{bmatrix} = \sqrt{\frac{2}{3}} \begin{bmatrix} \cos(\theta) * \cos(\omega t) + \sin(\theta)\sin(\omega t) \\ -\sin(\theta) * \cos(\omega t) + \cos(\theta)\sin(\omega t) \\ 0 \end{bmatrix}$$

Using trigonometric identities:

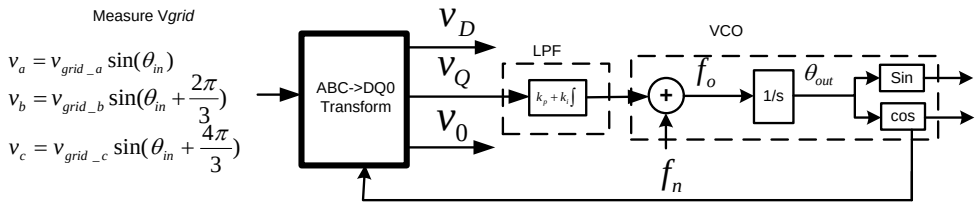
$$\begin{bmatrix} v_d \\ v_q \\ v_o \end{bmatrix} == \sqrt{\frac{2}{3}} \begin{bmatrix} \cos(\omega t - \theta) \\ \sin(\omega t - \theta) \\ 0 \end{bmatrix}$$

For the PLL to be almost locked, i.e. θ is equal to ωt the quadrature component can be linearized as follows:

$$v_q \approx (\omega t - \theta)$$

As seen in the above analysis the q component is zero for a balanced three phase system. This property is exploited in the phase locked loop for three phase grid connected application. The q component is used as the error signal for the PLL to track to the appropriate phase. First the three phase quantities are transformed into the rotating reference frame and the q component is used as the phase detect. A low pass filter/PI is then used to eliminate steady state error and the output fed to a VCO which generated the angle and sine values.

A functional diagram of a PLL is shown in the figure below, which consists of a phase detect (PD) consisting of park transform, a loop filter(LPF) and a voltage controlled oscillator(VCO):



Phase Lock Loop Basic Structure

The ABC-> DQ0 transform is kept separate from the PLL structure.

Using equations from the SPLL_1ph description above using settling time to be 30ms and the error band to be 5% and damping ratio to be 0.7 we can obtain the natural frequency to be 119.014 and then back substituting we get $K_p = 166.6$ and $K_i = 27755.55$

Back substituting these values into the digital loop filter coefficients we get:

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

Now for 50Khz run rate for the PLL, B0= 166.877556 and B1= -166.322444

Object Definition:

```
#define SPLL_SRF_Q _IQ21
#define SPLL_SRF_Qmpy _IQ21mpy
```

- **Fixed Point (IQ)**

```
//*********** Structure Definition ********//
typedef struct{
      int32 B1_lf;
      int32 B0_lf;
      int32 A1_lf;
}SPLL_3ph_SRF_IQ_LPF_COEFF;

typedef struct{
      int32 v_q[2];
      int32 ylf[2];
      int32 fo; // output frequency of PLL
      int32 fn; //nominal frequency
      int32 theta[2];
      int32 delta_T;
```

```
        SPLL_3ph_SRF_IQ_LPF_COEFF lpf_coeff;
}SPLL_3ph_SRF_IQ;
```

- **Floating Point (F)**

```
//************ Structure Definition ********//
typedef struct{
        float32     B1_lf;
        float32     B0_lf;
        float32     A1_lf;
}SPLL_3ph_SRF_F_LPF_COEFF;

typedef struct{
        float32 v_q[2];
        float32 ylf[2];
        float32 fo; // output frequency of PLL
        float32 fn; //nominal frequency
        float32 theta[2];
        float32 delta_T;
        SPLL_3ph_SRF_F_LPF_COEFF lpf_coeff;
}SPLL_3ph_SRF_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//************ Structure Definition ********//
typedef struct{
        float32     B1_lf;
        float32     B0_lf;
        float32     A1_lf;
}SPLL_3ph_SRF_CLA_LPF_COEFF;

typedef struct{
        float32 v_q[2];
        float32 ylf[2];
        float32 fo; // output frequency of PLL
        float32 fn; //nominal frequency
        float32 theta[2];
        float32 delta_T;
        SPLL_3ph_SRF_CLA_LPF_COEFF lpf_coeff;
}SPLL_3ph_SRF_CLA;
```

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_3ph_SRF The module definition is created as a data type. This makes it convenient to instance an interface to the SPLL_3ph_SRF module. To create multiple instances of the module simply declare variables of type SPLL_1ph.

Module interface Definition:

Module Element	Type	Description	Acceptable

Name			Range
v_q[]	Input	Rotating reference Frame Q-axis value	IQ21(-1,1) Float32(-1,1)
fn	Input	Nominal Grid Frequency in Hz	IQ21 Float32
theta[2]	Output	grid phase angle	IQ21 Float32
fo	Output	Instantaneous Grid Frequency in Hz	IQ21 Float32
lpf_coeff	Internal	Loop Filer Coefficients	IQ21 Float32
ylf[2]	Internal	Internal Data Buffer for Loop Filter output	IQ21 Float32
delta_t	Internal	1/Frequency of calling the PLL routine	IQ21 Float32

Usage:

- **Fixed Point (IQ)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
SPLL_3ph_SRF_IQ spll1;
```

Step 3 – **Initialize module** in {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. Ensure that the ISR is running at a minimum frequency of 20KHz.

```
SPLL_3ph_SRF_IQ_init(GRID_FREQ,_IQ21((float)(1.0/ISR_FREQUE
NCY)),&spll1);
```

Step 4 – **Using the module**

```
abc_dq0_pos1.a = _IQmpy(GridMeas1,_IQ(0.5));
abc_dq0_pos1.b = _IQmpy(GridMeas2,_IQ(0.5));
abc_dq0_pos1.c = _IQmpy(GridMeas3,_IQ(0.5));
abc_dq0_pos1.sin=_IQsin((spll1.theta[1])<<3);// Q24 to Q21
abc_dq0_pos1.cos=_IQcos((spll1.theta[1])<<3);// Q24 to Q21
ABC_DQ0_POS_IQ_MACRO(abc_dq0_pos1);

// Q24 to Q21
spll1.v_q[0] = (int32)(_IQtoIQ21(abc_dq0_pos1.q));
```

```
// SPLL call
SPLL_3ph_SRF_IQ_FUNC(&spll1);
```

Alternatively the macro routine can be called as bellow:

```
SPLL_3ph_SRF_IQ_MACRO(spll1);
```

- **Floating Point (F)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
SPLL_3ph_SRF_F spll1;
```

Step 3 – **Initialize module** in {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. Ensure that the ISR is running at a minimum frequency of 20KHz.

```
SPLL_3ph_SRF_F_init(GRID_FREQ,((float)(1.0/ISR_FREQUENCY)),
&spll1);
```

Step 4 – Using the module

```
abc_dq0_pos1.a = (GridMeas1);
abc_dq0_pos1.b = (GridMeas2);
abc_dq0_pos1.c = (GridMeas3);
abc_dq0_pos1.sin = (float)sin((spll1.theta[1]));
abc_dq0_pos1.cos = (float)cos((spll1.theta[1]));
ABC_DQ0_POS_F_MACRO(abc_dq0_pos1);

spll1.v_q[0] = (abc_dq0_pos1.q);
// SPLL call
SPLL_3ph_SRF_F_FUNC(&spll1);
```

Alternatively the macro routine can be called as bellow:

```
SPLL_3ph_SRF_F_MACRO(spll1);
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(spll1, "CLADataRAM")
ClaToCpu_Volatile SPLL_3ph_SRF_CLA 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_3ph_SRF_CLA spll1;
```

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.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. Ensure that the ISR is running at a minimum frequency of 20KHz. – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
      ….
      SPLL_3ph_SRF_CLA_init(GRID_FREQUENCY,((float32)(1.0/I
      SR_FREQUENCY)),spll1);
      ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

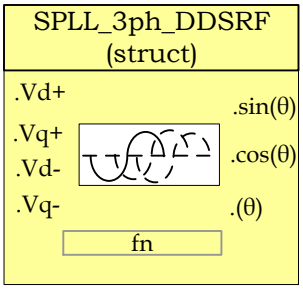
```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
```

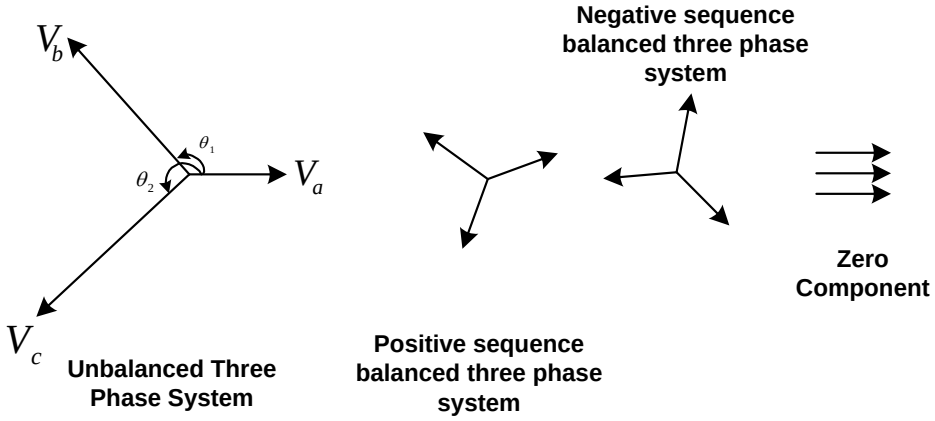
```
        ….
        abc_dq0_pos1.a = (GridMeas1);
        abc_dq0_pos1.b = (GridMeas2);
        abc_dq0_pos1.c = (GridMeas3);
        abc_dq0_pos1.sin = CLAsin(spll1.theta[1]);
        abc_dq0_pos1.cos = CLAcos(spll1.theta[1]);
        ABC_DQ0_POS_CLA_MACRO(abc_dq0_pos1);

        spll1.v_q[0] = (abc_dq0_pos1.q);
        // SPLL call
        SPLL_3ph_SRF_CLA_MACRO(spll1);
        ….
}
```

Description: This software module implements a software phase lock loop based on decoupled double synchronous reference frame for grid connection to three phase grid.



Module File: SPLL_3ph_DDSRF.h

Technical: The grid is subject to varying conditions which result in imbalances in the phase voltages. From the theory of symmetrical components we know that any unbalanced three phase system can be reduced to two symmetrical systems and zero component. The behavior of unbalanced voltages on park and clark transform is analyzed in the section below.



Now an unbalanced three phase system can be written as summation of balanced three phase systems; one rotating with the sequence of the three phase quantities called the positive sequence and one rotating in the opposite sequence called the negative sequence.

$$v = V^{+1}\begin{bmatrix}\cos(wt)\\ \cos(wt-2\pi/3)\\ \cos(wt-4\pi/3)\end{bmatrix} + V^{-1}\begin{bmatrix}\cos(wt)\\ \cos(wt-4\pi/3)\\ \cos(wt-2\pi/3)\end{bmatrix} + V^{0}\begin{bmatrix}1\\1\\1\end{bmatrix}$$

Now resolving the components in the orthogonal axis,

$$v_{\alpha\beta} = T_{abc->\alpha\beta} * v = \frac{2}{3}\begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix} * v = V^{+1}\begin{bmatrix} \cos(wt) \\ \sin(wt) \end{bmatrix} + V^{-1}\begin{bmatrix} \cos(-wt) \\ \sin(-wt) \end{bmatrix}$$

And taking the projections on the rotating reference frame we observe that any negative sequence component appears with twice the frequency on the positive sequence rotating frame axis and vice versa.

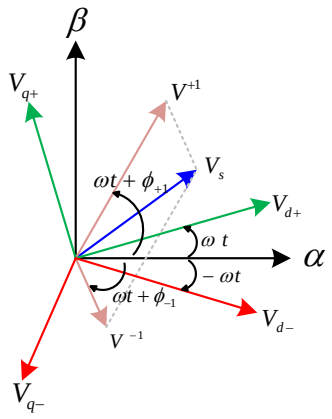
$$v_{dq+} = T_{abc->dq0+} * v_{\alpha\beta} = \begin{bmatrix} \cos(\omega t) & \sin(\omega t) \\ -\sin(\omega t) & \cos(\omega t) \end{bmatrix} = V^{+1}\begin{bmatrix} 1 \\ 0 \end{bmatrix} + V^{-1}\begin{bmatrix} \cos(-2wt) \\ \sin(-2wt) \end{bmatrix}$$

$$v_{dq-} = T_{abc->dq0-} * v_{\alpha\beta} = \begin{bmatrix} \cos(\omega t) & -\sin(\omega t) \\ \sin(\omega t) & \cos(\omega t) \end{bmatrix} = V^{+1}\begin{bmatrix} \cos(-2wt) \\ \sin(-2wt) \end{bmatrix} + V^{-1}\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

This can cause errors in the control loop and estimation of the grid angle and needs to taken into account while designing a phase locked loop for three phase grid connected application.

Hence assuming the instance just before the PLL is locked to the positive vector, the grid voltages can be written as :

$$v = V^{+1}\begin{bmatrix} \cos(wt + \phi_1) \\ \cos(wt - 2\pi/3 + \phi_1) \\ \cos(wt - 4\pi/3 + \phi_1) \end{bmatrix} + V^{-1}\begin{bmatrix} \cos(wt + \phi_{-1}) \\ \cos(wt - 4\pi/3 + \phi_{-1}) \\ \cos(wt - 2\pi/3 + \phi_{-1}) \end{bmatrix} + V^{0}\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$



Taking the clark transform and ignoring the zero component and the zero sequence

$$v_{\alpha\beta} = V^{+1}\begin{bmatrix}\cos(\omega t+\phi_{+1})\\ \sin(\omega t+\phi_{+1})\end{bmatrix}+V^{-1}\begin{bmatrix}\cos(-\omega t+\phi_{-1})\\ \sin(-\omega t+\phi_{-1})\end{bmatrix}$$

Now taking the park transform using the angle locked by the PLL for the positive sequence:

$$v_{dq+} = \left(V^{+1}\begin{bmatrix}\cos(\omega t+\phi_{+1})\\ \sin(\omega t+\phi_{+1})\end{bmatrix}+V^{-1}\begin{bmatrix}\cos(-\omega t+\phi_{-1})\\ \sin(-\omega t+\phi_{-1})\end{bmatrix}\right)*\begin{bmatrix}\cos(\omega t) & \sin(\omega t)\\ -\sin(\omega t) & \cos(\omega t)\end{bmatrix}$$

$$\Rightarrow v_{dq+} = \left(V^{+1}\begin{bmatrix}\cos(\phi_{+1})\\ \sin(\phi_{+1})\end{bmatrix}+V^{-1}\begin{bmatrix}\cos(-\omega t+\phi_{-1})\\ \sin(-\omega t+\phi_{-1})\end{bmatrix}*\begin{bmatrix}\cos(\omega t) & \sin(\omega t)\\ -\sin(\omega t) & \cos(\omega t)\end{bmatrix}\right)$$

$$\Rightarrow v_{dq+} = \left(V^{+1}\begin{bmatrix}\cos(\phi_{+1})\\ \sin(\phi_{+1})\end{bmatrix}+V^{-1}\begin{bmatrix}\cos(-\omega t+\phi_{-1})\cos(\omega t)+\sin(-\omega t+\phi_{-1})\sin(\omega t)\\ -\cos(-\omega t+\phi_{-1})\sin(\omega t)+\sin(-\omega t+\phi_{-1})\cos(\omega t)\end{bmatrix}\right)$$

$$\Rightarrow v_{dq+} = \left(V^{+1}\begin{bmatrix}\cos(\phi_{+1})\\ \sin(\phi_{+1})\end{bmatrix}+V^{-1}\begin{bmatrix}\cos(-2\omega t+\phi_{-1})\\ \sin(-2\omega t+\phi_{-1})\end{bmatrix}\right)$$

$$\Rightarrow v_{dq+} = \left(V^{+1}\begin{bmatrix}\cos(\phi_{+1})\\ \sin(\phi_{+1})\end{bmatrix}+V^{-1}\begin{bmatrix}\cos(\phi_{-1})\cos(2\omega t)+\sin(\phi_{-1})\sin(2\omega t)\\ \sin(\phi_{-1})\cos(2\omega t)-\cos(\phi_{-1})\sin(2\omega t)\end{bmatrix}\right)$$

$$\Rightarrow v_{dq+} = \left(V^{+1}\begin{bmatrix}\cos(\phi_{+1})\\ \sin(\phi_{+1})\end{bmatrix}+V^{-1}\cos(\phi_{-1})\begin{bmatrix}\cos(2\omega t)\\ -\sin(2\omega t)\end{bmatrix}+V^{-1}\sin(\phi_{-1})\begin{bmatrix}\sin(2\omega t)\\ \cos(2\omega t)\end{bmatrix}\right)$$

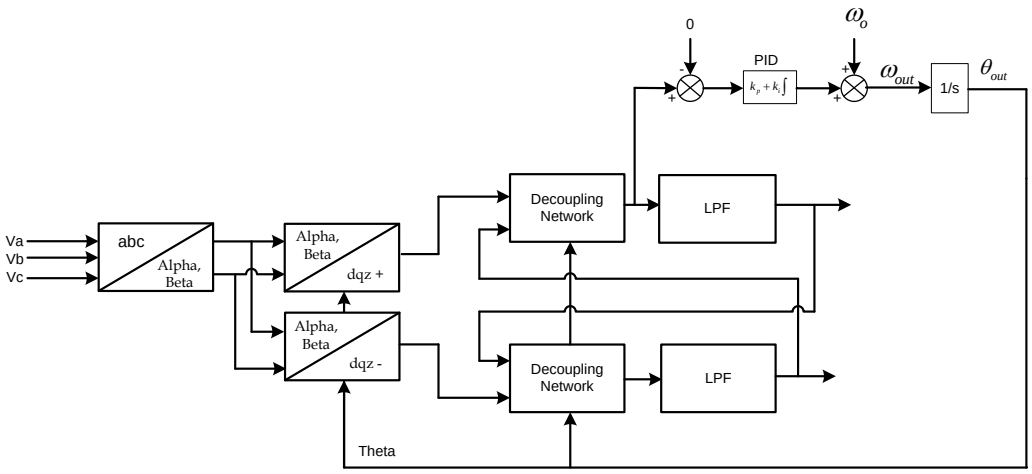
$$\Rightarrow v_{dq+} = \left(V^{+1}\begin{bmatrix}\cos(\phi_{+1})\\ \sin(\phi_{+1})\end{bmatrix}+\bar{v}_{d-}\begin{bmatrix}\cos(2\omega t)\\ -\sin(2\omega t)\end{bmatrix}+\bar{v}_{q-}\begin{bmatrix}\sin(2\omega t)\\ \cos(2\omega t)\end{bmatrix}\right)$$

Hence to get the decoupled value:

$$v_{d+\_decoupled} = V^{+1}\cos(\phi_{+1}) = v_{d+} - \bar{v}_{d-}\cos(2\omega t) - \bar{v}_{q-}\sin(2\omega t)$$

$$v_{q+\_decoupled} = V^{+1}\sin(\phi_{+1}) = v_{q+} + \bar{v}_{d-}\sin(2\omega t) - \bar{v}_{q-}\cos(2\omega t)$$

The PLL structure is then shown as:



Where the low pass filter transfer function is implemented as follows:

A typical LPF transfer function is given by

$$LPF(s) = \frac{\omega_f}{(s+\omega_f)}$$

In the analog domain, now using bilinear transformation,

$$LPF(z) = \frac{\omega_f}{(\frac{2}{T}\frac{(z-1)}{(z+1)} + \omega_f)} = \frac{\frac{\omega_f T}{(2+T\omega_f)}(z+1)}{(z+\frac{(\omega_f T-2)}{(\omega_f T+2)})} = \frac{k_1(z+1)}{(z+k_2)}$$

Where T is the sampling period the digital low pass filter is run at. Using T=1/(10Khz)=0.0001 and from the discussion in [1] it is known that $\frac{\omega_f}{\omega} < \frac{1}{\sqrt{2}}$ for stable response of the PLL, hence choosing $\omega_f = 30$ we get $k_1 = 0.00933678, k_2 = -0.9813264$

Object Definition:

```
#define SPLL_DDSRF_Q _IQ22
#define SPLL_DDSRF_Qmpy _IQ22mpy
```

- **Fixed Point (IQ)**

```
//*********** Structure Definition ********//
typedef struct{
      int32 B1_lf;
      int32 B0_lf;
      int32 A1_lf;
}SPLL_3ph_DDSRF_IQ_LPF_COEFF;

typedef struct{
      int32 d_p;
      int32 d_n;
      int32 q_p;
      int32 q_n;

      int32 d_p_decoupl;
      int32 d_n_decoupl;
      int32 q_p_decoupl;
      int32 q_n_decoupl;

      int32 cos_2theta;
      int32 sin_2theta;

      int32 y[2];
      int32 x[2];
      int32 w[2];
      int32 z[2];
      int32 k1;
      int32 k2;
      int32 d_p_decoupl_lpf;
      int32 d_n_decoupl_lpf;
      int32 q_p_decoupl_lpf;
      int32 q_n_decoupl_lpf;

      int32 v_q[2];
      int32 theta[2];
      int32 ylf[2];
      int32 fo;
      int32 fn;
      int32 delta_T;
      SPLL_3ph_DDSRF_IQ_LPF_COEFF lpf_coeff;
}SPLL_3ph_DDSRF_IQ;
```

- **Floating Point (F)**

```
//*********** Structure Definition ********//
typedef struct{
      float32     B1_lf;
      float32     B0_lf;
      float32     A1_lf;
}SPLL_3ph_DDSRF_F_LPF_COEFF;
```

```
typedef struct{
      float32 d_p;
      float32 d_n;
      float32 q_p;
      float32 q_n;

      float32 d_p_decoupl;
      float32 d_n_decoupl;
      float32 q_p_decoupl;
      float32 q_n_decoupl;

      float32 cos_2theta;
      float32 sin_2theta;

      float32 y[2];
      float32 x[2];
      float32 w[2];
      float32 z[2];
      float32 k1;
      float32 k2;
      float32 d_p_decoupl_lpf;
      float32 d_n_decoupl_lpf;
      float32 q_p_decoupl_lpf;
      float32 q_n_decoupl_lpf;

      float32 v_q[2];
      float32 theta[2];
      float32 ylf[2];
      float32 fo;
      float32 fn;
      float32 delta_T;
      SPLL_3ph_DDSRF_F_LPF_COEFF lpf_coeff;
}SPLL_3ph_DDSRF_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//*********** Structure Definition ********//
typedef struct{
      float32     B1_lf;
      float32     B0_lf;
      float32     A1_lf;
}SPLL_3ph_DDSRF_CLA_LPF_COEFF;

typedef struct{
      float32 d_p;
      float32 d_n;
      float32 q_p;
      float32 q_n;

      float32 d_p_decoupl;
      float32 d_n_decoupl;
      float32 q_p_decoupl;
      float32 q_n_decoupl;

      float32 cos_2theta;
```

```
        float32 sin_2theta;

        float32 y[2];
        float32 x[2];
        float32 w[2];
        float32 z[2];
        float32 k1;
        float32 k2;
        float32 d_p_decoupl_lpf;
        float32 d_n_decoupl_lpf;
        float32 q_p_decoupl_lpf;
        float32 q_n_decoupl_lpf;

        float32 v_q[2];
        float32 theta[2];
        float32 ylf[2];
        float32 fo;
        float32 fn;
        float32 delta_T;
        SPLL_3ph_DDSRF_CLA_LPF_COEFF lpf_coeff;
}SPLL_3ph_DDSRF_CLA;
```

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

Special Constants and Data types:

SPLL_3ph_DDSRF The module definition is created as a data type. This makes it convenient to instance an interface to the SPLL_3ph_DDSRF module. To create multiple instances of the module simply declare variables of type SPLL_3ph_DDSRF.

Module interface Definition:

Module Element Name	Type	Description	Acceptable Range
q_p	Input	Positive Rotating reference Frame Q-axis value	IQ22(-1,1) Float32(-1,1)
d_p	Input	Positive Rotating reference Frame D-axis value	IQ22(-1,1) Float32(-1,1)
q_n	Input	Negative Rotating reference Frame Q-axis value	IQ22(-1,1) Float32(-1,1)
d_n	Input	Negative Rotating reference Frame D-axis value	IQ22(-1,1) Float32(-1,1)
fn	Input	Nominal Grid Frequency in Hz	IQ22 Float32
cos_2theta, sin_2theta	Input	Cos and sin of twice the grid frequency angle	IQ22 Float32

theta[2]	Output	grid phase angle	IQ22 Float32
fo	Output	Instantaneous Grid Frequency in Hz	IQ22 Float32
lpf_coeff	Internal	Loop Filer Coefficients	IQ22 Float32
ylf[2]	Internal	Internal Data Buffer for Loop Filter output	IQ22 Float32
delta_t	Internal	1/Frequency of calling the PLL routine	IQ22 Float32
d_p_decoupl	Internal	Decoupled positive sequence D-axis component	IQ22 Float32
q_p_decoupl	Internal	Decoupled positive sequence Q-axis component	IQ22 Float32
d_n_decoupl	Internal	Decoupled negative sequence D-axis component	IQ22 Float32
q_n_decoupl	Internal	Decoupled negative sequence Q-axis component	IQ22 Float32
d_p_decoupl_lpf	Internal	Decoupled positive sequence D-axis component filtered	IQ22 Float32
q_p_decoupl_lpf	Internal	Decoupled positive sequence Q-axis component filtered	IQ22 Float32
d_n_decoupl_lpf	Internal	Decoupled negative sequence D-axis component filtered	IQ22 Float32
q_n_decoupl_lpf	Internal	Decoupled negative sequence Q-axis component filtered	IQ22 Float32
y[2],x[2],w[2],z[2]	Internal	Used to store history for filtering the decoupled D and Q axis components	IQ22 Float32
k1,k2	Internal	Lpf coefficients	IQ22 Float32

Usage:

- **Fixed Point (IQ)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
SPLL_3ph_DDSRF_IQ spll1;
```

Step 3 – Initialize module in {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. Ensure that the ISR is running at a minimum frequency of 20KHz.

```
SPLL_3ph_DDSRF_IQ_init(GRID_FREQ,_IQ22((float)(1.0/ISR_FREQ
UENCY)), SPLL_DDSRF_Q(0.00933678), SPLL_DDSRF_Q(-0.9813264)
,&spll1);
```

Step 4 – Using the module

```
spll1.d_p =abc_dq0_pos1.d>>2;       // Convert Q24 to Q22
spll1.q_p =abc_dq0_pos1.q>>2;       // Convert Q24 to Q22
spll1.d_n =abc_dq0_neg1.d>>2;       // Convert Q24 to Q22
spll1.q_n =abc_dq0_neg1.q>>2;       // Convert Q24 to Q22
spll1.cos_2theta=_IQ22cos(_IQ22mpy(_IQ22(2),
spll1.theta[0]));
spll1.sin_2theta=_IQ22sin(_IQ22mpy(_IQ22(2),
spll1.theta[0]));
// SPLL call
SPLL_3ph_DDSRF_IQ_FUNC(&spll1);
```

Alternatively the macro routine can be called as bellow:

```
SPLL_3ph_DDSRF_IQ_MACRO(spll1);
```

- **Floating Point (F)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
SPLL_3ph_DDSRF_F spll1;
```

Step 3 – Initialize module in {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. Ensure that the ISR is running at a minimum frequency of 20KHz.

```
SPLL_3ph_DDSRF_F_init(GRID_FREQ,((float)(1.0/ISR_FREQUENCY)
), (float)(0.00933678), (float)(-0.9813264),&spll1);
```

Step 4 – Using the module

```
spll1.d_p = abc_dq0_pos1.d;
spll1.q_p = abc_dq0_pos1.q;
spll1.d_n = abc_dq0_neg1.d;
spll1.q_n = abc_dq0_neg1.q;
spll1.cos_2theta=(float)cos((2)*spll1.theta[0]);
spll1.sin_2theta=(float)sin((2)*spll1.theta[0]);
```

```
// SPLL call
SPLL_3ph_DDSRF_F_FUNC(&spll1);
```

Alternatively the macro routine can be called as bellow:

```
SPLL_3ph_DDSRF_F_MACRO(spll1);
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(spll1, "CLADataRAM")
ClaToCpu_Volatile SPLL_3ph_DDSRF_CLA 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_3ph_DDSRF_CLA spll1;
```

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – Initialize module in {ProjectName}-CLA_Tasks.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. Ensure that the ISR is running at a minimum frequency of 20KHz. – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
    ….
```

```
        SPLL_3ph_DDSRF_CLA_init(GRID_FREQUENCY,
        ((float32)(1.0/ISR_FREQUENCY)),
        ((float32)(0.00933678)), ((float32)(-0.9813264)),
        spll1);

        ….

}
```

The task is forced from {ProjectName}-Main.c by calling:

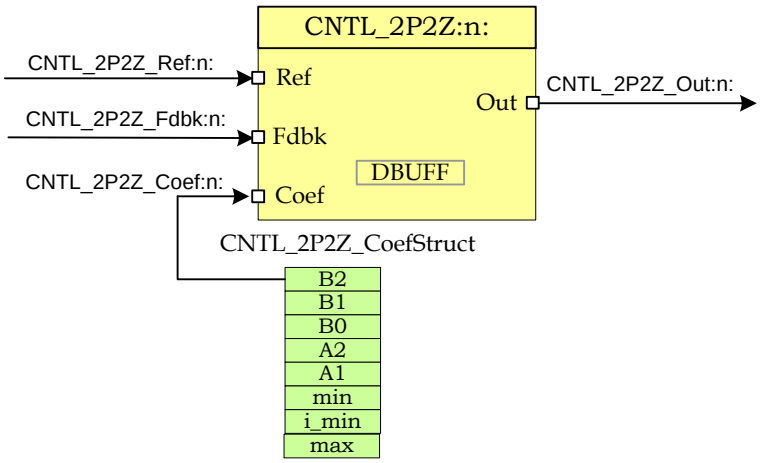
```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {

        ….

        spll1.d_p = abc_dq0_pos1.d;

        spll1.q_p = abc_dq0_pos1.q;

        spll1.d_n = abc_dq0_neg1.d;

        spll1.q_n = abc_dq0_neg1.q;

        spll1.cos_2theta=(float32)CLAcos((2.0)*spll1.theta[1]
        );

        spll1.sin_2theta=(float32)CLAsin((2.0)*spll1.theta[1]
        );

        // SPLL call

        SPLL_3ph_DDSRF_CLA_MACRO(spll1);

        ….

}
```

4.4 Controller Modules

Description: This assembly macro implements a second order control law using a 2-pole, 2-zero construction. The code implementation is a second order IIR filter with programmable output saturation.



Macro File: <*base_folder*>\CNTL_2P2Z_(IQ/F/CLA).h

Technical: The 2-pole 2-zero control block implements a second order control law using an IIR filter structure with programmable output saturation. This type of controller requires two delay lines: one for input data and one for output data, each consisting of two elements.

The discrete transfer function for the basic 2P2Z control law is…

$$\frac{U(z)}{E(z)} = \frac{b_2 z^{-2} + b_1 z^{-1} + b_0}{1 - a_1 z^{-1} - a_2 z^{-2}}$$

This may be expressed in difference equation form as:

$$u(n) = a_1 u(n-1) + a_2 u(n-2) + b_0 e(n) + b_1 e(n-1) + b_2 e(n-2)$$

Where…

$u(n)$ = present controller output (after saturation)

$u(n\text{-}1)$ = controller output on previous cycle

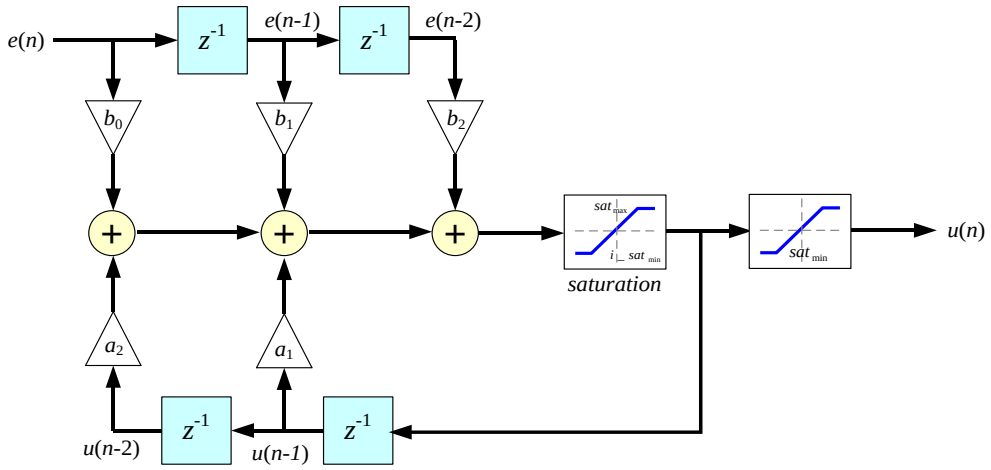
$u(n\text{-}2)$ = controller output two cycles previously

$e(n)$ = present controller input

$e(n\text{-}1)$ = controller input on previous cycle

$e(n\text{-}2)$ = controller input two cycles previously

The 2P2Z control law may be represented graphically as shown below:



Input and output data are located in internal RAM with address designated by CNTL_2P2Z_DBUFF as shown below. Note that to preserve maximum resolution the module saves the values inside CNTL_2P2Z_DBUFF in _IQ30 format.

CNTL_2P2Z_DBUFF

0	$u(n\text{-}1)$
2	$u(n\text{-}2)$
4	$e(n)$
6	$e(n\text{-}1)$
8	$e(n\text{-}2)$

Controller coefficients and saturation settings are located in memory as follows:

CNTL_2P2Z_CoefStruct

$_IQ26(b_2)$
$_IQ26(b_1)$
$_IQ26(b_0)$
$_IQ26(a_2)$
$_IQ26(a_1)$
$_IQ24(sat_{max})$
$_IQ24(i\_sat_{min})$
$_IQ24(sat_{min})$

Where sat_{max} and sat_{min} are the upper and lower control effort bounds respectively. i_sat_{min} is the value used for saturating the lower bound of the control effort when storing the history of the output. This allows the value of the history have negative values which can help avoid oscillations on the output in case of no load. The user can specify it's own value however it is recommended to use _IQ24(-0.9). Also, note that to preserve maximum resolution the coefficients are saved in Q26 format and the saturation limits are stored in Q24 format to match the output format.

Controller coefficients must be initialized before the controller is used. A structure CNTL_2P2Z_COEFFS is used to ensure that the coefficients are stored exactly as shown in the table as the CNTL_2P2Z accesses them relative to a base address pointer.

Object Definition:

- **Fixed Point (IQ)**

```
typedef struct {
      // Coefficients
      int32 Coeff_B2;
      int32 Coeff_B1;
      int32 Coeff_B0;
      int32 Coeff_A2;
      int32 Coeff_A1;

      // Output saturation limits
      int32 Max;
      int32 IMin;
      int32 Min;
} CNTL_2P2Z_IQ_COEFFS;

typedef struct {
      int32 Out1;
      int32 Out2;
      // Internal values
      int32 Errn;
```

```
        int32 Errn1;
        int32 Errn2;
        // Inputs
        int32 Ref;
        int32 Fdbk;
        // Output values
        int32 Out;
} CNTL_2P2Z_IQ_VARS;
```

- **Floating Point (F)**

```
typedef struct {
        // Coefficients
        float32 Coeff_B2;
        float32 Coeff_B1;
        float32 Coeff_B0;
        float32 Coeff_A2;
        float32 Coeff_A1;

        // Output saturation limits
        float32 Max;
        float32 IMin;
        float32 Min;
} CNTL_2P2Z_F_COEFFS;

typedef struct {
        float32 Out1;
        float32 Out2;
        // Internal values
        float32 Errn;
        float32 Errn1;
        float32 Errn2;
        // Inputs
        float32 Ref;
        float32 Fdbk;
        // Output values
        float32 Out;
} CNTL_2P2Z_F_VARS;
```

- **Control Law Accelerated Floating Point (CLA)**

```
typedef struct {
        // Coefficients
        float32 Coeff_B2;
        float32 Coeff_B1;
        float32 Coeff_B0;
        float32 Coeff_A2;
        float32 Coeff_A1;

        // Output saturation limits
        float32 Max;
        float32 IMin;
        float32 Min;
} CNTL_2P2Z_CLA_COEFFS;

typedef struct {
```

```
        // Inputs
        float32 Ref;
        float32 Fdbk;

        // Internal values
        float32 Errn;
        float32 Errn1;
        float32 Errn2;

        // Output values
        float32 Out;
        float32 Out1;
        float32 Out2;
        float32 OutPresat;
} CNTL_2P2Z_CLA_VARS;
```

Module interface Definition:

Usage:

- **Fixed Point (IQ)**

Step 1 – Add library header file to {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
CNTL_2P2Z_IQ_COEFFS cntl_2p2z_coeffs1;
CNTL_2P2Z_IQ_VARS cntl_2p2z_vars1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
CNTL_2P2Z_IQ_COEFFS_init(&cntl_2p2z_coeffs1);
cntl_2p2z_coeffs1.Coeff_A1 = _IQ(0.1);
cntl_2p2z_coeffs1.Coeff_A2 = _IQ(0.2);
cntl_2p2z_coeffs1.Coeff_B0 = _IQ(0.1);
cntl_2p2z_coeffs1.Coeff_B1 = _IQ(0.2);
cntl_2p2z_coeffs1.Coeff_B2 = _IQ(0.3);
CNTL_2P2Z_IQ_VARS_init(&cntl_2p2z_vars1);
```

Step 4 – **Using the module**

```
cntl_2p2z_vars1.Ref = _IQ(1.0);
cntl_2p2z_vars1.Fdbk = _IQ(0.1);
CNTL_2P2Z_IQ_ASM(&cntl_2p2z_coeffs1,&cntl_2p2z_vars1);
```

Alternatively the macro routine can be called as bellow:

```
CNTL_2P2Z_IQ_MACRO(cntl_2p2z_coeffs1,cntl_2p2z_vars1);
```

- **Floating Point (F)**

Step 1 – Add library header file to {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h "
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
CNTL_2P2Z_F_COEFFS cntl_2p2z_coeffs1;
CNTL_2P2Z_F_VARS cntl_2p2z_vars1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
CNTL_2P2Z_F_COEFFS_init(&cntl_2p2z_coeffs1);
cntl_2p2z_coeffs1.Coeff_A1 = (0.1);
cntl_2p2z_coeffs1.Coeff_A2 = (0.2);
cntl_2p2z_coeffs1.Coeff_B0 = (0.1);
cntl_2p2z_coeffs1.Coeff_B1 = (0.2);
cntl_2p2z_coeffs1.Coeff_B2 = (0.3);
CNTL_2P2Z_F_VARS_init(&cntl_2p2z_vars1);
```

Step 4 – Using the module

```
cntl_2p2z_vars1.Ref = (1.0);
cntl_2p2z_vars1.Fdbk = (0.1);
CNTL_2P2Z_F_ASM(&cntl_2p2z_coeffs1,&cntl_2p2z_vars1);
```

Alternatively the macro routine can be called as bellow:

```
CNTL_2P2Z_F_MACRO(cntl_2p2z_coeffs1,cntl_2p2z_vars1);
```

- **Control Law Accelerated Floating Point (CLA)**

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

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(cntl_2p2z_coeffs1, "CLADataRAM")
ClaToCpu_Volatile CNTL_2P2Z_CLA_COEFFS cntl_2p2z_coeffs1;
#pragma DATA_SECTION(cntl_2p2z_vars1, "CLADataRAM")
ClaToCpu_Volatile CNTL_2P2Z_CLA_VARS cntl_2p2z_vars1;
```

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

```
extern ClaToCpu_Volatile CNTL_2P2Z_CLA_COEFFS
   cntl_2p2z_coeffs1;
extern ClaToCpu_Volatile CNTL_2P2Z_CLA_VARS
   cntl_2p2z_vars1;
```

Step 3 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
      ….
      CNTL_2P2Z_CLA_COEFFS_init(cntl_2p2z_coeffs1);
      cntl_2p2z_coeffs1.Coeff_A1 = (0.1);
      cntl_2p2z_coeffs1.Coeff_A2 = (0.2);
      cntl_2p2z_coeffs1.Coeff_B0 = (0.1);
      cntl_2p2z_coeffs1.Coeff_B1 = (0.2);
      cntl_2p2z_coeffs1.Coeff_B2 = (0.3);
      CNTL_2P2Z_CLA_VARS_init(cntl_2p2z_vars1);
      ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

```
Cla1ForceTask8andWait();
```

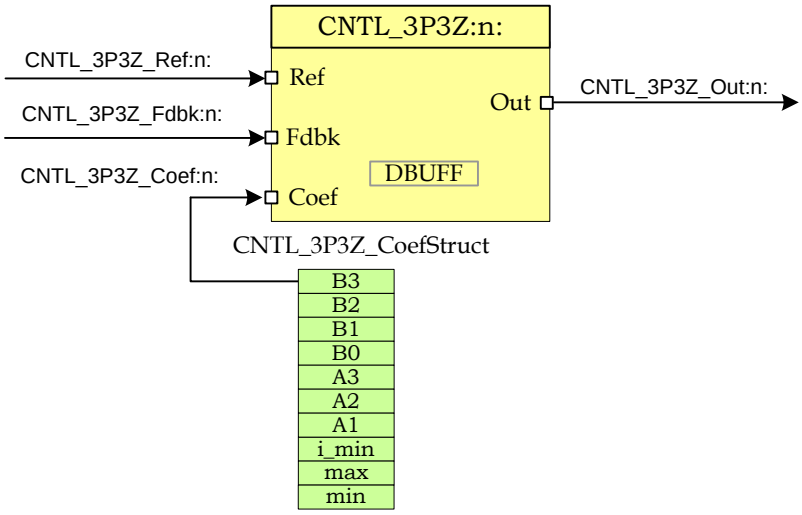
Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
```

```
        ….
        cntl_2p2z_vars1.Ref = (1.0);
        cntl_2p2z_vars1.Fdbk = (0.1);
        CNTL_2P2Z_CLA_MACRO(cntl_2p2z_coeffs1,cntl_2p2z_vars1
           );
        ….
}
```

CNTL_3P3Z — Three Pole Three Zero Controller

Description: This assembly macro implements a third order control law using a 3-pole, 3-zero construction. The code implementation is a third order IIR filter with programmable output saturation.



Macro File: *<base_folder>*\

Technical: The 3-pole 3-zero control block implements a third order control law using an IIR filter structure with programmable output saturation. This type of controller requires three delay lines: one for input data and one for output data, each consisting of three elements.

The discrete transfer function for the basic 3P3Z control law is…

$$\frac{U(z)}{E(z)} = \frac{b_3 z^{-3} + b_2 z^{-2} + b_1 z^{-1} + b_0}{1 - a_3 z^{-3} - a_2 z^{-2} - a_1 z^{-1}}$$

This may be expressed in difference equation form as:

$$u(n) = a_1 u(n-1) + a_2 u(n-2) + a_3 u(n-3) + b_0 e(n) + b_1 e(n-1) + b_2 e(n-2) + b_3 e(n-3)$$

Where…

$u(n)$ = present controller output (after saturation)

$u(n\text{-}1)$ = controller output on previous cycle

u(*n*-2) = controller output two cycles previously

u(*n*-3) = controller output three cycles previously

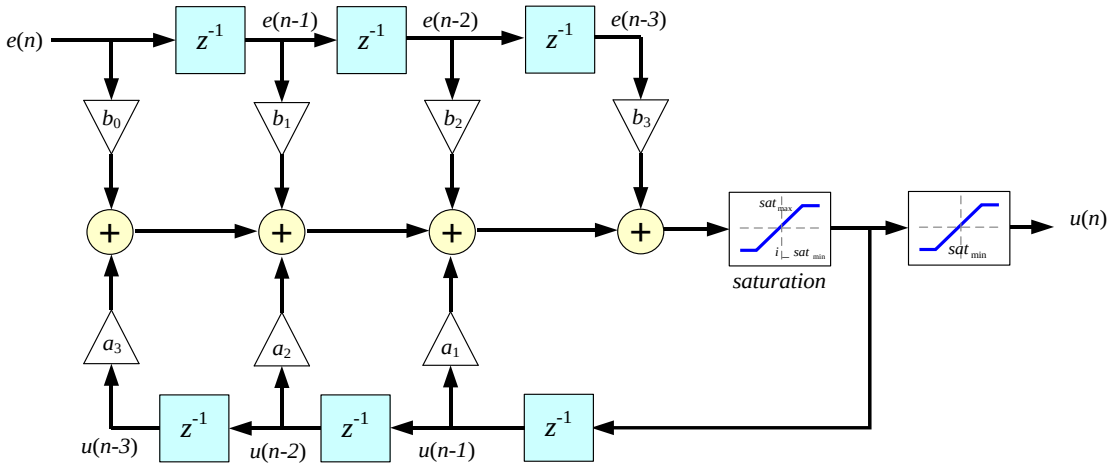
e(*n*) = present controller input

e(*n*-1) = controller input on previous cycle

e(*n*-2) = controller input two cycles previously

e(*n*-3) = controller input three cycles previously

The 3P3Z control law may also be represented graphically as shown below:



Input and output data are located in internal RAM with address designated by CNTL_3P3Z_DBUFF as shown below. Note that to preserve maximum resolution the module saves the values inside CNTL_3P3Z_DBUFF in _IQ30 format.

CNTL_3P3Z_DBUFF

0	*u*(*n*-1)
2	*u*(*n*-2)
4	u(*n*-3)
6	*e*(n)
8	*e*(*n*-1)
10	*e*(*n*-2)
12	*e*(*n*-3)

Controller coefficients and saturation settings are located in memory as shown:

CNTL_3P3Z_CoefStruct

_IQ26(b_3)
_IQ26(b_2)
_IQ26(b_1)
_IQ26(b_0)
_IQ26(a_3)
_IQ26(a_2)
_IQ26(a_1)
_IQ24(sat_{max})
_IQ24(i_sat_{min})
_IQ24(sat_{min})

Where sat_{max} and sat_{min} are the upper and lower control effort bounds respectively. i_sat_{min} is the value used for saturating the lower bound of the control effort when storing the history of the output. This allows the value of the history have negative values which can help avoid oscillations on the output in case of no load. The user can specify it's own value however it is recommended to use _IQ24(-0.9). Note that to preserve maximum resolution the coefficients are saved in Q26 format and the saturation limits are stored in Q24 format to match the output format.

Controller coefficients must be initialized before the controller is used. A structure CNTL_3P3Z_COEFFS is used to ensure that the coefficients are stored exactly as shown in the table as the CNTL_3P3Z accesses them relative to a base address pointer.

Object Definition:

- **Fixed Point (IQ)**

```
//*********** Structure Definition ********//
typedef struct {
      // Coefficients
      int32 Coeff_B3;
      int32 Coeff_B2;
      int32 Coeff_B1;
      int32 Coeff_B0;
      int32 Coeff_A3;
      int32 Coeff_A2;
      int32 Coeff_A1;

      // Output saturation limits
      int32 Max;
      int32 IMin;
      int32 Min;
```

```
} CNTL_3P3Z_IQ_COEFFS;

typedef struct {
      int32 Out1;
      int32 Out2;
      int32 Out3;
      // Internal values
      int32 Errn;
      int32 Errn1;
      int32 Errn2;
      int32 Errn3;
      // Inputs
      int32 Ref;
      int32 Fdbk;
      // Output values
      int32 Out;
} CNTL_3P3Z_IQ_VARS;
```

- **Floating Point (F)**

```
//*********** Structure Definition ********//
typedef struct {
      // Coefficients
      float32 Coeff_B3;
      float32 Coeff_B2;
      float32 Coeff_B1;
      float32 Coeff_B0;
      float32 Coeff_A3;
      float32 Coeff_A2;
      float32 Coeff_A1;

      // Output saturation limits
      float32 Max;
      float32 IMin;
      float32 Min;
} CNTL_3P3Z_F_COEFFS;

typedef struct {
      float32 Out1;
      float32 Out2;
      float32 Out3;
      // Internal values
      float32 Errn;
      float32 Errn1;
      float32 Errn2;
      float32 Errn3;
      // Inputs
      float32 Ref;
      float32 Fdbk;
      // Output values
      float32 Out;
} CNTL_3P3Z_F_VARS;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//*********** Structure Definition ********//
```

```
typedef struct {
      // Coefficients
      float32 Coeff_B3;
      float32 Coeff_B2;
      float32 Coeff_B1;
      float32 Coeff_B0;
      float32 Coeff_A3;
      float32 Coeff_A2;
      float32 Coeff_A1;

      // Output saturation limits
      float32 Max;
      float32 IMin;
      float32 Min;
} CNTL_3P3Z_CLA_COEFFS;

typedef struct {
      // Inputs
      float32 Ref;
      float32 Fdbk;

      // Internal values
      float32 Errn;
      float32 Errn1;
      float32 Errn2;
      float32 Errn3;

      // Output values
      float32 Out;
      float32 Out1;
      float32 Out2;
      float32 Out3;
      float32 OutPresat;
} CNTL_3P3Z_CLA_VARS;
```

Module interface Definition:

Usage:

- **Fixed Point (IQ)**

Step 1 – Add library header file to {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
CNTL_3P3Z_IQ_COEFFS cntl_3p3z_coeffs1;
CNTL_3P3Z_IQ_VARS cntl_3p3z_vars1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
CNTL_3P3Z_IQ_COEFFS_init(&cntl_3p3z_coeffs1);
cntl_3p3z_coeffs1.Coeff_A1 = _IQ(0.01);
cntl_3p3z_coeffs1.Coeff_A2 = _IQ(0.05);
cntl_3p3z_coeffs1.Coeff_A3 = _IQ(0.1);
cntl_3p3z_coeffs1.Coeff_B0 = _IQ(0.01);
cntl_3p3z_coeffs1.Coeff_B1 = _IQ(0.05);
cntl_3p3z_coeffs1.Coeff_B2 = _IQ(0.1);
cntl_3p3z_coeffs1.Coeff_B3 = _IQ(0.15);
CNTL_3P3Z_IQ_VARS_init(&cntl_3p3z_vars1);
```

Step 4 – **Using the module**

```
cntl_3p3z_vars1.Ref = _IQ(1.0);
cntl_3p3z_vars1.Fdbk = _IQ(0.1);
CNTL_3P3Z_IQ_ASM(&cntl_3p3z_coeffs1,&cntl_3p3z_vars1);
```

Alternatively the macro routine can be called as bellow:

```
CNTL_3P3Z_IQ_MACRO(cntl_3p3z_coeffs1,cntl_3p3z_vars1);
```

- **Floating Point (F)**

Step 1 – Add library header file to {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h "
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
CNTL_3P3Z_F_COEFFS cntl_3p3z_coeffs1;
CNTL_3P3Z_F_VARS cntl_3p3z_vars1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
CNTL_3P3Z_F_COEFFS_init(&cntl_3p3z_coeffs1);
cntl_3p3z_coeffs1.Coeff_A1 = (0.01);
cntl_3p3z_coeffs1.Coeff_A2 = (0.05);
cntl_3p3z_coeffs1.Coeff_A3 = (0.1);
cntl_3p3z_coeffs1.Coeff_B0 = (0.01);
cntl_3p3z_coeffs1.Coeff_B1 = (0.05);
cntl_3p3z_coeffs1.Coeff_B2 = (0.1);
cntl_3p3z_coeffs1.Coeff_B3 = (0.15);
```

```
CNTL_3P3Z_F_VARS_init(&cntl_3p3z_vars1);
```

Step 4 – Using the module

```
cntl_3p3z_vars1.Ref = (1.0);
cntl_3p3z_vars1.Fdbk = (0.1);
CNTL_3P3Z_F_ASM(&cntl_3p3z_coeffs1,&cntl_3p3z_vars1);
```

Alternatively the macro routine can be called as bellow:

```
CNTL_3P3Z_F_MACRO(cntl_3p3z_coeffs1,cntl_3p3z_vars1);
```

- **Control Law Accelerated Floating Point (CLA)**

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

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(cntl_3p3z_coeffs1, "CLADataRAM")
ClaToCpu_Volatile CNTL_3P3Z_CLA_COEFFS cntl_3p3z_coeffs1;
#pragma DATA_SECTION(cntl_3p3z_vars1, "CLADataRAM")
ClaToCpu_Volatile CNTL_3P3Z_CLA_VARS cntl_3p3z_vars1;
```

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

```
extern ClaToCpu_Volatile CNTL_3P3Z_CLA_COEFFS
   cntl_3p3z_coeffs1;
extern ClaToCpu_Volatile CNTL_3P3Z_CLA_VARS
   cntl_3p3z_vars1;
```

Step 3 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
        ….
        CNTL_3P3Z_CLA_COEFFS_init(cntl_3p3z_coeffs1);
        cntl_3p3z_coeffs1.Coeff_A1 = (0.01);
        cntl_3p3z_coeffs1.Coeff_A2 = (0.05);
        cntl_3p3z_coeffs1.Coeff_A3 = (0.1);
        cntl_3p3z_coeffs1.Coeff_B0 = (0.01);
        cntl_3p3z_coeffs1.Coeff_B1 = (0.05);
        cntl_3p3z_coeffs1.Coeff_B2 = (0.1);
        cntl_3p3z_coeffs1.Coeff_B3 = (0.15);
        CNTL_3P3Z_CLA_VARS_init(cntl_3p3z_vars1);
        ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

```
 Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
        ….
        cntl_3p3z_vars1.Ref = (1.0);
        cntl_3p3z_vars1.Fdbk = (0.1);
        CNTL_3P3Z_CLA_MACRO(cntl_3p3z_coeffs1,cntl_3p3z_vars1
);
        ….
}
```

CNTL_PI	PI Controller

Description:

Macro File: *<base_folder>*

Technical:

Object Definition:

- **Fixed Point (IQ)**

```
//*********** Structure Definition ********//
typedef struct {
       int32 Ref;          // Input: reference set-point
       int32 Fbk;          // Input: feedback
       int32 Out;          // Output: controller output
       int32 Kp;           // Parameter: proportional loop gain
       int32 Ki;           // Parameter: integral gain
       int32 Umax;         // Parameter: upper saturation limit
       int32 Umin;         // Parameter: lower saturation limit
       int32 up;           // Data: proportional term
       int32 ui;           // Data: integral term
       int32 v1;           // Data: pre-saturated controller output
       int32 i1;           // Data: integrator storage: ui(k-1)
       int32 w1;           // Data: saturation record: [u(k-1) - v(k-1)]
} CNTL_PI_IQ;
```

- **Floating Point (F)**

```
//*********** Structure Definition ********//
typedef struct {
       float32  Ref;       // Input: reference set-point
       float32  Fbk;       // Input: feedback
       float32  Out;       // Output: controller output
       float32  Kp;        // Parameter: proportional loop gain
       float32  Ki;        // Parameter: integral gain
       float32  Umax;      // Parameter: upper saturation limit
       float32  Umin;      // Parameter: lower saturation limit
       float32  up;        // Data: proportional term
       float32  ui;        // Data: integral term
       float32  v1;        // Data: pre-saturated controller output
       float32  i1;        // Data: integrator storage: ui(k-1)
       float32  w1;        // Data: saturation record: [u(k-1) - v(k-1)]
       } CNTL_PI_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//*********** Structure Definition ********//
typedef struct {
```

```
        float32  Ref;       // Input: reference set-point
        float32  Fbk;       // Input: feedback
        float32  Out;       // Output: controller output
        float32  Kp;        // Parameter: proportional loop gain
        float32  Ki;        // Parameter: integral gain
        float32  Umax;      // Parameter: upper saturation limit
        float32  Umin;      // Parameter: lower saturation limit
        float32  up;        // Data: proportional term
        float32  ui;        // Data: integral term
        float32  v1;        // Data: pre-saturated controller output
        float32  i1;        // Data: integrator storage: ui(k-1)
        float32  w1;        // Data: saturation record: [u(k-1) - v(k-1)]
} CNTL_PI_CLA;
```

Module interface Definition:

Usage:

- **Fixed Point (IQ)**

Step 1 – Add library header file to {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
CNTL_PI_IQ cntl_pi1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
CNTL_PI_IQ_init(&cntl_pi1);
cntl_pi1.Ki = _IQ(0.1);
cntl_pi1.Kp = _IQ(0.2);
```

Step 4 – **Using the module**

```
cntl_pi1.Ref = _IQ(1.0);
cntl_pi1.Fbk = _IQ(0.1);
CNTL_PI_IQ_FUNC(&cntl_pi1);
```

Alternatively the macro routine can be called as bellow:

```
MATH_EMAVG_IQ_MACRO(math_emavg1);
```

- **Floating Point (F)**

Step 1 – Add library header file to {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h "
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
CNTL_PI_F cntl_pi1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
CNTL_PI_F_init(&cntl_pi1);
cntl_pi1.Ki = (0.1);
cntl_pi1.Kp = (0.2);
```

Step 4 – Using the module

```
cntl_pi1.Ref = (1.0);
cntl_pi1.Fbk = (0.1);
CNTL_PI_F_FUNC(&cntl_pi1);
```

Alternatively the macro routine can be called as bellow:

```
MATH_EMAVG_F_MACRO(math_emavg1);
```

- **Control Law Accelerated Floating Point (CLA)**

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

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(cntl_pi1, "CLADataRAM")
ClaToCpu_Volatile CNTL_PI_CLA cntl_pi1;
```

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

```
extern ClaToCpu_Volatile CNTL_PI_CLA cntl_pi1;
```

Step 3 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {

      ….

      CNTL_PI_CLA_init(cntl_pi1);
      cntl_pi1.Ki = (0.1);
      cntl_pi1.Kp = (0.2);

      ….

}
```

The task is forced from {ProjectName}-Main.c by calling:

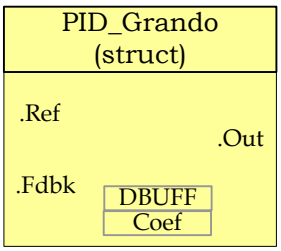
```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {

      ….

      cntl_pi1.Ref = (1.0);
      cntl_pi1.Fbk = (0.1);
      CNTL_PI_CLA_MACRO(cntl_pi1);

      ….

}
```

PID_GRANDO	PID regulator

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



Module File: <*base_folder*>\PID_GRANDO_(IQ/F/CLA).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

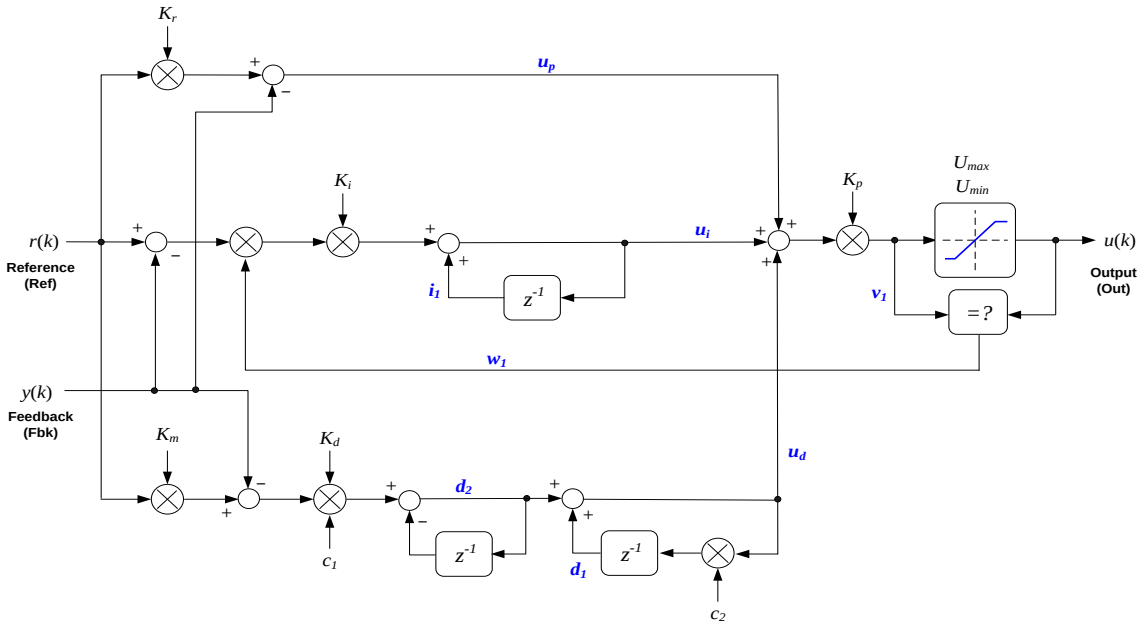


Figure 10 PID_Grando Controller structure

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:

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

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.

$$u_i(k) = u_i(k-1) + K_i[r(k) - y(k)] \quad \text{.....(2)}$$

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.

$$e(k) = K_m r(k) - y(k) \quad \text{.....(3)}$$

$$u_d(k) = K_d[c_2 u_i(k-1) + c_1 e(k) - c_1 e(k-1)] \quad \text{.....(4)}$$

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

$$c_1 = a \quad \text{.....(5)}$$

$$c_2 = 1 - c_1 T \quad \text{..... (6)}$$

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.

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

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

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

Steps to 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:

- **Fixed Point (IQ)**

```
//*********** Structure Definitions *******//
typedef struct {
      int32  Ref;       // Input: reference set-point
      int32  Fbk;       // Input: feedback
      int32  Out;       // Output: controller output
      int32  c1;        // Internal: derivative filter coefficient 1
      int32  c2;        // Internal: derivative filter coefficient 2
} PID_GRANDO_IQ_TERMINALS;
// note: c1 & c2 placed here to keep structure size under 8 words

typedef struct {
      int32  Kr;        // Parameter: reference set-point weighting
```

```
      int32  Kp;        // Parameter: proportional loop gain
      int32  Ki;        // Parameter: integral gain
      int32  Kd;        // Parameter: derivative gain
      int32  Km;        // Parameter: derivative weighting
      int32  Umax;      // Parameter: upper saturation limit
      int32  Umin;      // Parameter: lower saturation limit
} PID_GRANDO_IQ_PARAMETERS;

typedef struct {
      int32  up;        // Data: proportional term
      int32  ui;        // Data: integral term
      int32  ud;        // Data: derivative term
      int32  v1;        // Data: pre-saturated controller output
      int32  i1;        // Data: integrator storage: ui(k-1)
      int32  d1;        // Data: differentiator storage: ud(k-1)
      int32  d2;        // Data: differentiator storage: d2(k-1)
      int32  w1;        // Data: saturation record: [u(k-1) - v(k-1)]
} PID_GRANDO_IQ_DATA;

typedef struct {
      PID_GRANDO_IQ_TERMINALS  term;
      PID_GRANDO_IQ_PARAMETERS param;
      PID_GRANDO_IQ_DATA       data;
} PID_GRANDO_IQ_CONTROLLER;
```

- **Floating Point (F)**

```
//*********** Structure Definitions *******//
typedef struct {
      float32  Ref;     // Input: reference set-point
      float32  Fbk;     // Input: feedback
      float32  Out;     // Output: controller output
      float32  c1;      // Internal: derivative filter coefficient 1
      float32  c2;      // Internal: derivative filter coefficient 2
} PID_GRANDO_F_TERMINALS;
// note: c1 & c2 placed here to keep structure size under 8 words

typedef struct {
      float32  Kr;      // Parameter: reference set-point weighting
      float32  Kp;      // Parameter: proportional loop gain
      float32  Ki;      // Parameter: integral gain
      float32  Kd;      // Parameter: derivative gain
      float32  Km;      // Parameter: derivative weighting
      float32  Umax;    // Parameter: upper saturation limit
      float32  Umin;    // Parameter: lower saturation limit
} PID_GRANDO_F_PARAMETERS;

typedef struct {
      float32  up;      // Data: proportional term
      float32  ui;      // Data: integral term
      float32  ud;      // Data: derivative term
      float32  v1;      // Data: pre-saturated controller output
      float32  i1;      // Data: integrator storage: ui(k-1)
      float32  d1;      // Data: differentiator storage: ud(k-1)
      float32  d2;      // Data: differentiator storage: d2(k-1)
      float32  w1;      // Data: saturation record: [u(k-1) - v(k-1)]
```

```
} PID_GRANDO_F_DATA;

typedef struct {
      PID_GRANDO_F_TERMINALS  term;
      PID_GRANDO_F_PARAMETERS param;
      PID_GRANDO_F_DATA       data;
} PID_GRANDO_F_CONTROLLER;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//************ Structure Definitions *******//
typedef struct {
      float32  Ref;     // Input: reference set-point
      float32  Fbk;     // Input: feedback
      float32  Out;     // Output: controller output
      float32  c1;      // Internal: derivative filter coefficient 1
      float32  c2;      // Internal: derivative filter coefficient 2
} PID_GRANDO_CLA_TERMINALS;
// note: c1 & c2 placed here to keep structure size under 8 words

typedef struct {
      float32  Kr;      // Parameter: reference set-point weighting
      float32  Kp;      // Parameter: proportional loop gain
      float32  Ki;      // Parameter: integral gain
      float32  Kd;      // Parameter: derivative gain
      float32  Km;      // Parameter: derivative weighting
      float32  Umax;    // Parameter: upper saturation limit
      float32  Umin;    // Parameter: lower saturation limit
} PID_GRANDO_CLA_PARAMETERS;

typedef struct {
      float32  up;      // Data: proportional term
      float32  ui;      // Data: integral term
      float32  ud;      // Data: derivative term
      float32  v1;      // Data: pre-saturated controller output
      float32  i1;      // Data: integrator storage: ui(k-1)
      float32  d1;      // Data: differentiator storage: ud(k-1)
      float32  d2;      // Data: differentiator storage: d2(k-1)
      float32  w1;      // Data: saturation record: [u(k-1) - v(k-1)]
} PID_GRANDO_CLA_DATA;

typedef struct {
      PID_GRANDO_CLA_TERMINALS  term;
      PID_GRANDO_CLA_PARAMETERS param;
      PID_GRANDO_CLA_DATA       data;
} PID_GRANDO_CLA_CONTROLLER;
```

Module interface Definition:

Usage:

- **Fixed Point (IQ)**

Step 1 – Add library header file to {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
PID_GRANDO_IQ_CONTROLLER pid_grando_controller1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
PID_GRANDO_IQ_init(&pid_grando_controller1);
pid_grando_controller1.param.Kp = _IQ(0.8);
pid_grando_controller1.param.Ki = _IQ(0.15);
pid_grando_controller1.param.Kd = _IQ(0.0);
pid_grando_controller1.param.Kr = _IQ(1.0);
pid_grando_controller1.param.Umax = _IQ(1.0);
pid_grando_controller1.param.Umin = _IQ(-1.0);
```

Step 4 – **Using the module**

```
pid_grando_controller1.term.Ref = _IQ(1.0);
pid_grando_controller1.term.Fbk = _IQ(.5);
pid_grando_controller1.term.c1 = _IQ(0.1);
pid_grando_controller1.term.c2 = _IQ(0.1);
DLOG_4CH_IQ_FUNC(&dlog_4ch1);
```

Alternatively the macro routine can be called as bellow:

```
DLOG_4CH_IQ_MACRO(dlog_4ch1);
```

- **Floating Point (F)**

Step 1 – Add library header file to {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h "
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
PID_GRANDO_F_CONTROLLER pid_grando_controller1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
PID_GRANDO_F_init(&pid_grando_controller1);
pid_grando_controller1.param.Kp = (0.8);
```

```
pid_grando_controller1.param.Ki = (0.15);
pid_grando_controller1.param.Kd = (0.0);
pid_grando_controller1.param.Kr = (1.0);
pid_grando_controller1.param.Umax = (1.0);
pid_grando_controller1.param.Umin = (-1.0);
```

Step 4 – Using the module

```
pid_grando_controller1.term.Ref = (10.0);
pid_grando_controller1.term.Fbk = (1.0);
pid_grando_controller1.term.c1 = (0.01);
pid_grando_controller1.term.c2 = (0.02);
PID_GRANDO_F_FUNC(&pid_grando_controller1);
```

Alternatively the macro routine can be called as bellow:

```
PID_GRANDO_F_MACRO(pid_grando_controller1);
```

- **Control Law Accelerated Floating Point (CLA)**

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

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(pid_grando_controller1, "CLADataRAM")
ClaToCpu_Volatile PID_GRANDO_CLA_CONTROLLER
   pid_grando_controller1;
```

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_CLA_CONTROLLER
   pid_grando_controller1;
```

Step 3 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
        ….
        PID_GRANDO_CLA_init(pid_grando_controller1);
        pid_grando_controller1.param.Kp = (0.8);
        pid_grando_controller1.param.Ki = (0.15);
        pid_grando_controller1.param.Kd = (0.0);
        pid_grando_controller1.param.Kr = (1.0);
        pid_grando_controller1.param.Umax = (1.0);
        pid_grando_controller1.param.Umin = (-1.0);
        ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

```
Cla1ForceTask8andWait();
```

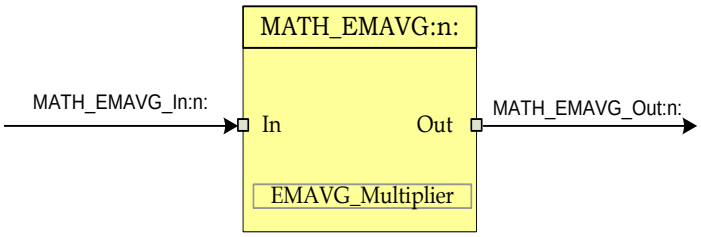
Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
        ….
        pid_grando_controller1.term.Ref = (10.0);
        pid_grando_controller1.term.Fbk = (1.0);
        pid_grando_controller1.term.c1 = (0.01);
        pid_grando_controller1.term.c2 = (0.02);
        PID_GRANDO_CLA_MACRO(pid_grando_controller1);
        ….
}
```

4.5 Math Modules

MATH_EMAVG	Exponential Moving Average

Description: This software module performs exponential moving average



Macro File: <*base_folder*>\MATH_EMAVG_(IQ/F/CLA).h

Technical: This software module performs exponential moving average over data stored in Q24 format, pointed to by MATH_EMAVG_In:n: The result is stored in Q24 format at a 32 bit location pointed to by MATH_EMAVG_Out:n:

The math operation performed can be represented in time domain as follows:

$$EMA(n) = (Input(n) - EMA(n-1)) * Multiplier + EMA(n-1)$$

Where $Input(n)$ is the input data at sample instance 'n',

$EMA(n)$ is the exponential moving average at time instance 'n',

$EMA(n-1)$ is the exponential moving average at time instance 'n-1'.

$Multiplier$ is the weighting factor used in exponential moving average

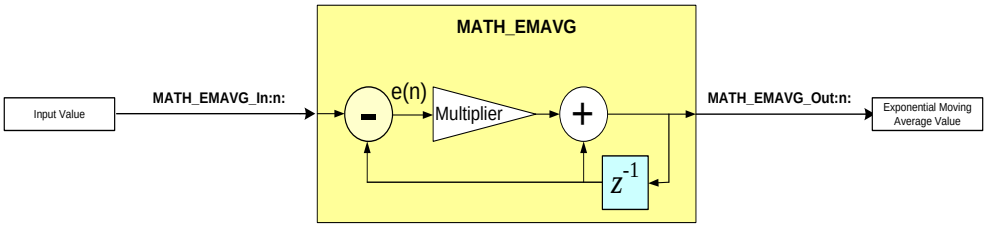
In z-domain the equation can be interpreted as

$$\frac{Output}{Input} = \frac{Multiplier}{1-(1-Multiplier)z^{-1}}$$

This can be seen as a special case for a Low Pass Filter, where pass band gain is equal to $Multiplier$ and filter time constant is $(1-Multiplier)$.

Note $Multiplier$ is always ≤ 1, hence $(1-Multiplier)$ is always a positive value. Also the lower the value of $Multiplier$, the larger is the time constant and more sluggish the response of the filter.

The following diagram illustrates the math function operated on in this block.:



The block is used in the PFC software to get the average value of AC Line. The multiplier value for this can be estimated through two methods as follows:

Time Domain: The PFC stage runs at 100Khz and the input AC signal is 60Hz. As the average of the rectified sine signal is desired the effective frequency of the signal being averaged is 120Hz. This implies that (100Khz/120) = 833 samples in one half sine. For the average to be true representation the average needs to be taken over multiple sine halves (note taking average over integral number of sine halves is not necessary). The multiplier value distributes the error equally over the number of samples for which average is taken. Therefore:

$$Multiplier = \_IQ30(1/SAMPLE\_No) = \_IQ30(1/3332) = \_IQ30(0.0003)$$

For AC line average a value of 4000 samples is chosen, as it averages roughly over 4 sine halves.

Frequency Domain: Alternatively the multiplier value can be estimated from the z-domain representation as well. The signal is sampled at 100Khz and the frequency content is at 60Hz. Only the DC value is desired, therefore assuming a cut-off frequency of 5Hz the value can be estimated as follows:

For a first order approximation, $z = e^{sT} = 1 + sT_s$,

where T is the sampling period and solving the equation:

$$\frac{Out(s)}{Input(s)} = \frac{1 + sT_s}{1 + s\dfrac{T_s}{Mul}}$$

Comparing with the analog domain low pass filter, the following equation can be written :

$$Multiplier = \_IQ30((2*\pi*f_{cutt\_off})/f_{sampling}) = \_IQ30(5*2*3.14/100K) = \_IQ30(0.000314)$$

Object Definition:

- **Fixed Point (IQ)**

```
//*********** Structure Definition ********//
typedef struct {
       int32 In;
       int32 Out;
       int32 Multiplier;
} MATH_EMAVG_IQ;
```

- **Floating Point (F)**

```
//*********** Structure Definition ********//
typedef struct {
       float32 In;
       float32 Out;
       float32 Multiplier;
} MATH_EMAVG_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//*********** Structure Definition ********//
typedef struct {
       float32 In;
       float32 Out;
       float32 Multiplier;
} MATH_EMAVG_CLA;
```

Module interface Definition:

Usage:

- **Fixed Point (IQ)**

Step 1 – Add library header file to {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
MATH_EMAVG_IQ math_emavg1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
MATH_EMAVG_IQ_init(&math_emavg1);
```

Step 4 – **Using the module**

```
math_emavg1.In = _IQ(10);
math_emavg1.Multiplier = _IQ(0.1);
```

```
MATH_EMAVG_IQ_FUNC(&math_emavg1);
```

Alternatively the macro routine can be called as bellow:

```
DLOG_1CH_IQ_MACRO(dlog_1ch1);
```

- **Floating Point (F)**

Step 1 – Add library header file to {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h "
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
MATH_EMAVG_F math_emavg1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
MATH_EMAVG_F_init(&math_emavg1);
```

Step 4 – Using the module

```
math_emavg1.In = (10.0);
math_emavg1.Multiplier = (0.1);
MATH_EMAVG_F_MACRO(math_emavg1);
```

Alternatively the macro routine can be called as bellow:

```
MATH_EMAVG_F_FUNC(&math_emavg1);
```

- **Control Law Accelerated Floating Point (CLA)**

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

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(math_emavg1, "CLADataRAM")
ClaToCpu_Volatile MATH_EMAVG_CLA math_emavg1;
```

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

```
extern ClaToCpu_Volatile_EMAVG_CLA math_emavg1;
```

Step 3 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
      ….
      MATH_EMAVG_CLA_init(math_emavg1);
      ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

```
Cla1ForceTask8andWait();
```

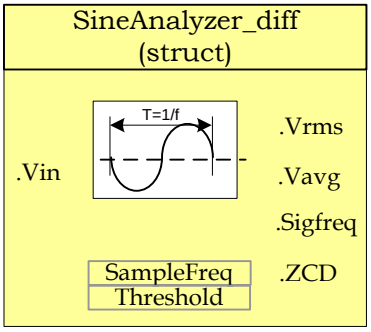
Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
      ….
      math_emavg1.In = (10.0);
      math_emavg1.Multiplier = (0.1);

      MATH_EMAVG_CLA_MACRO(math_emavg1);
      ….
}
```

SINEANALYZER_DIFF

Computes rms, avG, and ema value of sinusoidal signal

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



Module File: <*base_folder*>\SINEANALYZER_DIFF_(IQ/F/CLA).h

Technical: This module accumulates the sampled sine wave inputs, checks for threshold crossing point and calculates the RMS, Average, and EMA 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:

1) 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.
2) 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.
3) 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, and EMA 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.

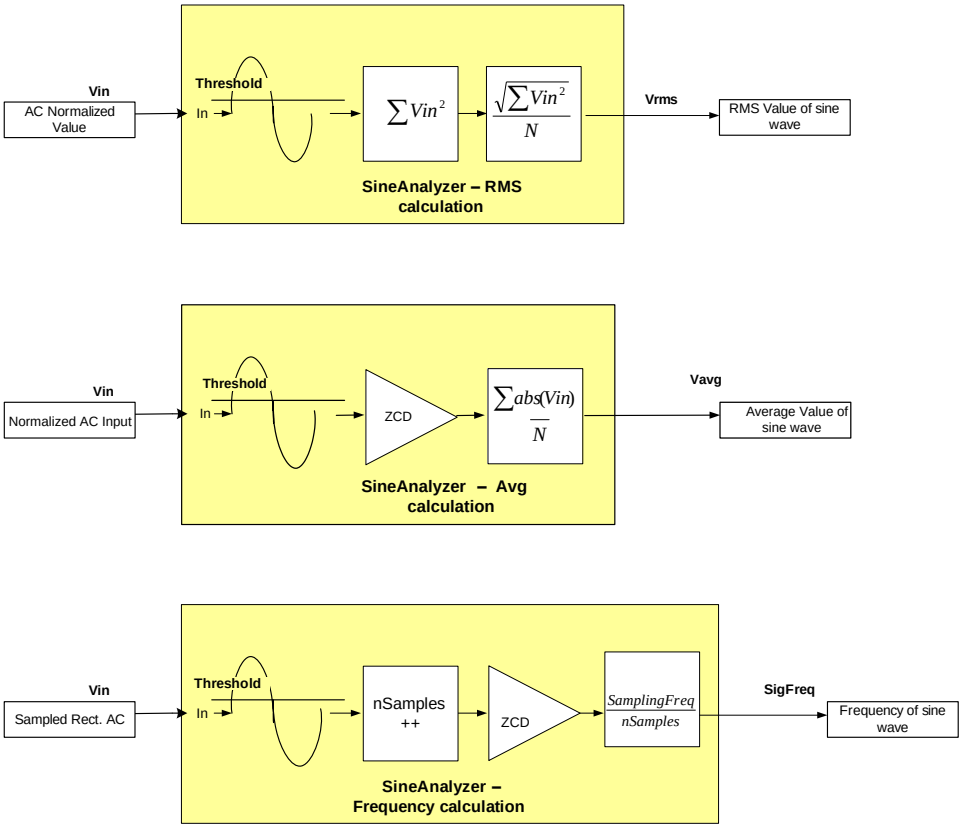
This module generates the following Outputs:

1) 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.

2) 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.

3) Exponential Moving Average value of sine wave (Vema): Output reflects the EMA value of the sine wave input signal in Q15 format. EMA value is calculated and updated at every threshold crossover point.

4) 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.



Object Definition:

- **Fixed Point (IQ)**

```
//************ Structure Definition ********//
typedef struct {
      int32 Vin;              // Input: Sine Signal
      int32 SampleFreq;       // Input: Signal Sampling Freq
      int32 Threshold;  // Input: Voltage level corresponds to zero i/p
      int32 Vrms;             // Output: RMS Value
      int32 Vavg;             // Output: Average Value
      int32 Vema;       // Output: Exponential Moving Average Value
```

```
      int32 SigFreq;          // Output: Signal Freq
      Uint16 ZCD;             // Output: Zero Cross detected
      // internal variables
      int32 Vacc_avg ;
      int32 Vacc_rms ;
      int32 Vacc_ema;
      int32 curr_sample_norm; // normalized value of current sample
      Uint16 prev_sign ;
      Uint16 curr_sign ;
      Uint32 nsamples ;       // samples in half cycle input waveform
      int32 inv_nsamples;
      int32 inv_sqrt_nsamples;
} SINEANALYZER_DIFF_IQ;
```

- **Floating Point (F)**

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

- **Control Law Accelerated Floating Point (CLA)**

```
//************ Structure Definition ********//
typedef struct {
      float32 Vin;            // Input: Sine Signal
      float32 SampleFreq;     // Input: Signal Sampling Freq
      float32 Threshold;// Input: Voltage level corresponds to zero i/p
      float32 Vrms;           // Output: RMS Value
      float32 Vavg;           // Output: Average Value
      float32 Vema;     // Output: Exponential Moving Average Value
      float32 SigFreq;        // Output: Signal Freq
      Uint16 ZCD;             // Output: Zero Cross detected
      // internal variables
      float32 Vacc_avg;
      float32 Vacc_rms;
      float32 Vacc_ema;
      float32 curr_sample_norm;// normalized value of current sample
      Uint16 prev_sign;
```

```
    Uint16 curr_sign;
    Uint32 nsamples;        // samples in half cycle input waveform
    float32 inv_nsamples;
    float32 inv_sqrt_nsamples;
} SINEANALYZER_DIFF_CLA;
```

Module interface Definition:

Net name	Type	Description	Acceptable Range
Vin	Input	Sampled Sine Wave input	Q15 Float32
Threshold	Input	Threshold to be used for cross over detection	Q15 Float32
SampleFreq	Input	Frequency at which the Vin (input sine wave) is sampled, in Hz	Q15 Float32
Vrms	Output	RMS value of the sine wave input (Vin) updated at cross over point	Q15 Float32
Vavg	Output	Average value of the sine wave input (Vin) updated at cross over point	Q15 Float32
Vema	Output	Exponential Moving Average value of the sine wave input (Vin) updated at cross over point	Q15 Float32
SigFreq	Output	Frequency of the sine wave input (Vin) updated at cross over point	Q15 Float32
ZCD	Output	When 'I' - indicates that Cross over happened and stays high till the next call of the macro.	Q15 Float32
Vacc_avg	Internal	Used for accumulation of samples for Average value calculation	Q15 Float32
Vacc_rms	Internal	Used for accumulation of squared samples for RMS value calculation	Q15 Float32
Vacc_ema	Internal	Used for accumulation of samples for exponential moving average value calculation	Q15 Float32
Nsamples	Internal	Number of samples between two crossover points	Int32
inv_nsamples	Internal	Inverse of nsamples	Q15 Float32
inv_sqrt_nsamples	Internal	Inverse square root of nsamples	Q15 Float32
Prev_sign, Curr_sign	Internal	Used for calculation of cross over detection	Int16

Usage:

- **Fixed Point (IQ)**

  Step 1 – Include library in {ProjectName}-Includes.h

  Link Solar Library: (Solar_Lib_IQ.lib) located in:
  controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

  Step 2 – Create and add module structure to {ProjectName}-Main.c

```
SINEANALYZER_DIFF_IQ sineanalyzer_diff1;
```

  Step 3 – **Initialize module** in {ProjectName}-Main.c

```
SINEANALYZER_DIFF_IQ_init(&sineanalyzer_diff1);
sineanalyzer_diff1.SampleFreq = ISR_FREQUENCY;
```

  Step 4 – **Using the module**

```
SINEANALYZER_DIFF_IQ_init(&sineanalyzer_diff1);
sineanalyzer_diff1.SampleFreq = ISR_FREQ;
sineanalyzer_diff1.Threshold = _IQ15(0.2);
sineanalyzer_diff1.nsamplesMin = _IQmpy(_IQdiv((1.0)
      ,(GRID_FREQ+5)),(ISR_FREQ));//153@60Hz,20kHz;
sineanalyzer_diff1.nsamplesMax = _IQmpy(_IQdiv((1.0),(GRID_FREQ-
      5)) ,(ISR_FREQ));//181@60Hz,20kHz;
SINEANALYZER_DIFF_IQ_FUNC(&sineanalyzer_diff1);
```

  Alternatively the macro routine can be called as bellow:

```
SINEANALYZER_DIFF_IQ_MACRO(sineanalyzer_diff1);
```

- **Floating Point (F)**

  Step 1 – Include library in {ProjectName}-Includes.h

  Link Solar Library: (Solar_Lib_F.lib) located in:
  controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

  Step 2 – Create and add module structure to {ProjectName}-Main.c

```
SINEANALYZER_DIFF_F sineanalyzer_diff1;
```

  Step 3 – **Initialize module** in {ProjectName}-Main.c

```
SINEANALYZER_DIFF_F_init(&sineanalyzer_diff1);
sineanalyzer_diff1.SampleFreq = ISR_FREQUENCY;
```

Step 4 – Using the module

```
sineanalyzer_diff1.Vin = (GridMeas*coefficient);
sineanalyzer_diff1.Threshold = (thresholdvalue);


SINEANALYZER_DIFF_F_FUNC(&sineanalyzer_diff1);
```

Alternatively the macro routine can be called as bellow:

```
SINEANALYZER_DIFF_F_MACRO(sineanalyzer_diff1);
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(sineanalyzer_diff1, "CLADataRAM")
ClaToCpu_Volatile SINEANALYZER_DIFF_CLA sineanalyzer_diff1;
```

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

```
extern ClaToCpu_Volatile SINEANALYZER_DIFF_CLA
sineanalyzer_diff1;
```

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
        ….
        SINEANALYZER_DIFF_CLA_init(sineanalyzer_diff1);
        sineanalyzer_diff1.SampleFreq = ISR_FREQUENCY;
        ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

```
Cla1ForceTask8andWait();
```

Step 5 – **Using the macro –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
        ….
        sineanalyzer_diff1.Vin = (sin_ramp*2.0);
        sineanalyzer_diff1.Threshold = (0.1);

        SINEANALYZER_DIFF_CLA_MACRO(sineanalyzer_diff1);
        ….
}
```

Description:

Macro File: <*base_folder*>\SINEANALYZER_DIFF_wPWR_(IQ/F/CLA).h

Technical:

Object Definition:

- **Fixed Point (IQ)**

```
//************ Structure Definition ********//
typedef struct {
      int32  Vin;              // Input: Sine Signal
      int32  SampleFreq;       // Input: Signal Sampling Freq
      int32  Threshold;        // Input: Voltage level corresponding to zero
                               i/p
      int32  Vrms;             // Output: RMS Value
      int32  Vavg;             // Output: Average Value
      int32  Vema;             // Output: Exponential Moving Average Value
      int32  SigFreq;          // Output: Signal Freq
      int32  Iin;              // Input Current Signal
      int32  Irms;             // Output: RMS Value of current
      int32  Prms;             // Output: RMS Value of input power
      Uint16 ZCD;              // Output: Zero Cross detected
      int32  sum_Vacc_avg;     // Internal : running sum for vacc_avg
                               calculation over one sine cycle
      int32  sum_Vacc_rms;     // Internal : running sum for vacc_rms
                               calculation over one sine cycle
      int32  sum_Vacc_ema;     // Internal : running sum for vacc_ema
                               calculation over one sine cycle
      int32  sum_Iacc_rms;     // Internal : running sum for Iacc_rms
                               calculation over one sine cycle
      int32  sum_Pacc_rms;     // Internal : running sum for Pacc_rms
                               calculation over one sine cycle
      int32  curr_vin_norm;    // Internal: Normalized value of the input
                               voltage
      int32  curr_iin_norm;    // Internal: Normalized value of the input
                               current
      Uint16 prev_sign;        // Internal: Flag to detect ZCD
      Uint16 curr_sign;        // Internal: Flag to detect ZCD
      Uint32 nsamples;         // Internal: No of samples in one cycle of
                               the sine wave
      Uint32 nsamplesMin;      // Internal: Lowerbound for no of samples in
                               one sine wave cycle
      Uint32 nsamplesMax;      // Internal: Upperbound for no of samples in
                               one sine wave cycle
      int32  inv_nsamples;     // Internal: 1/( No of samples in one cycle
                               of the sine wave)
```

```
      int32  inv_sqrt_nsamples; // Internal: sqrt(1/( No of samples in one
                                cycle of the sine wave))
      Uint16 slew_power_update; // Internal: used to slew update of the power
                                value
      int32  sum_Pacc_mul;      // Internal: used to sum Pac value over
                                multiple sine cycles (100)
} SINEANALYZER_DIFF_wPWR_IQ;
```

- **Floating Point (F)**

```
//*********** Structure Definition ********//
typedef struct {
      float32 Vin;              // Input: Sine Signal
      float32 SampleFreq;       // Input: Signal Sampling Freq
      float32 Threshold;        // Input: Voltage level corresponding to zero
                                i/p
      float32 Vrms;             // Output: RMS Value
      float32 Vavg;             // Output: Average Value
      float32 Vema;             // Output: Exponential Moving Average Value
      float32 SigFreq;          // Output: Signal Freq
      float32 Iin;              // Input Current Signal
      float32 Irms;             // Output: RMS Value of current
      float32 Prms;             // Output: RMS Value of input power
      Uint16  ZCD;              // Output: Zero Cross detected
      float32 sum_Vacc_avg;     // Internal : running sum for vacc_avg
                                calculation over one sine cycle
      float32 sum_Vacc_rms;     // Internal : running sum for vacc_rms
                                calculation over one sine cycle
      float32 sum_Vacc_ema;     // Internal : running sum fir vacc_ema
                                calculation over one sine cycle
      float32 sum_Iacc_rms;      // Internal : running sum for Iacc_rms
                                calculation over one sine cycle
      float32 sum_Pacc_rms;      // Internal : running sum for Pacc_rms
                                calculation over one sine cycle
      float32 curr_vin_norm;    // Internal: Normalized value of the input
                                voltage
      float32 curr_iin_norm;    // Internal: Normalized value of the input
                                current
      Uint16  prev_sign;        // Internal: Flag to detect ZCD
      Uint16  curr_sign;         // Internal: Flag to detect ZCD
      Uint32  nsamples;         // Internal: No of samples in one cycle of
                                the sine wave
      Uint32  nsamplesMin;      // Internal: Lowerbound for no of samples in
                                one sine wave cycle
      Uint32  nsamplesMax;      // Internal: Upperbound for no of samples in
                                one sine wave cycle
      float32 inv_nsamples;     // Internal: 1/( No of samples in one cycle
                                of the sine wave)
      float32 inv_sqrt_nsamples;// Internal: sqrt(1/( No of samples in one
                                cycle of the sine wave))
      Uint16  slew_power_update; // Internal: used to slew update of the power
                                value
      float32 sum_Pacc_mul;     // Internal: used to sum Pac value over
                                multiple sine cycles (100)
      } SINEANALYZER_DIFF_wPWR_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//*********** Structure Definition ********//
typedef struct {
      float32      Vin;         // Input: Sine Signal
      float32 SampleFreq;       // Input: Signal Sampling Freq
      float32 Threshold;        // Input: Voltage level corresponding to zero
                                i/p
      float32 Vrms;             // Output: RMS Value
      float32 Vavg;             // Output: Average Value
      float32 Vema;             // Output: Exponential Moving Average
      float32 SigFreq;          // Output: Signal Freq
      float32 Iin;              // Input Current Signal
      float32 Irms;             // Output: RMS Value of current
      float32 Prms;             // Output: RMS Value of input power
      Uint16 ZCD;               // Output: Zero Cross detected
      float32 sum_Vacc_avg;     // Internal : running sum for vacc_avg
                                calculation over one sine cycle
      float32 sum_Vacc_rms;     // Internal : running sum for vacc_rms
                                calculation over one sine cycle
      float32 sum_Vacc_ema;     // Internal : running sum for vacc_ema
                                calculation over one sine cycle
      float32 sum_Iacc_rms;     // Internal : running sum for Iacc_rms
                                calculation over one sine cycle
      float32 sum_Pacc_rms;     // Internal : running sum for Pacc_rms
                                calculation over one sine cycle
      float32 curr_vin_norm;    // Internal: Normalized value of the input
                                voltage
      float32 curr_iin_norm;    // Internal: Normalized value of the input
                                current
      int16 prev_sign;          // Internal: Flag to detect ZCD  //changed
                                Uint16 -> int16\
      int16 curr_sign;          // Internal: Flag to detect ZCD //changed
                                Uint16 -> int16
      int32 nsamples;           // Internal: No of samples in one cycle of
                                the sine wave
      int32  nsamplesMin;       // Internal: Lowerbound for no of samples in
                                one sine wave cycle
      int32  nsamplesMax;       // Internal: Upperbound for no of samples in
                                one sine wave cycle
      float32 inv_nsamples;     // Internal: 1/( No of samples in one cycle
                                of the sine wave)
      float32 inv_sqrt_nsamples;// Internal: sqrt(1/( No of samples in one
                                cycle of the sine wave))
      int16   slew_power_update; // Internal: used to slew update of the
                                power value //changed Uint16 -> int16
      float32 sum_Pacc_mul;     // Internal: used to sum Pac value over
                                multiple sine cycles (100)
} SINEANALYZER_DIFF_wPWR_CLA;
```

Module interface Definition:

Usage:

- **Fixed Point (IQ)**

  Step 1 – Add library header file to {ProjectName}-Includes.h

  Link Solar Library: (Solar_Lib_IQ.lib) located in:
  controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

  Step 2 – Create and add module structure to {ProjectName}-Main.c

```
SINEANALYZER_DIFF_wPWR_IQ sineanalyzer_diff_wpwr1;
```

  Step 3 – **Initialize module** in {ProjectName}-Main.c

```
SINEANALYZER_DIFF_wPWR_IQ_init(&sineanalyzer_diff_wpwr1);
sineanalyzer_diff_wpwr1.SampleFreq = ISR_FREQ;
sineanalyzer_diff_wpwr1.Threshold = _IQ15(0.2);
sineanalyzer_diff_wpwr1.nsamplesMin =
      _IQmpy(_IQdiv((1.0),(GRID_FREQ+5))
      ,(ISR_FREQ));//153@60Hz,20kHz;
sineanalyzer_diff_wpwr1.nsamplesMax =
      _IQmpy(_IQdiv((1.0),(GRID_FREQ-5))
      ,(ISR_FREQ));//181@60Hz,20kHz;
```

  Step 4 – **Using the module**

```
sineanalyzer_diff_wpwr1.Vin = _IQtoIQ15(sin_ramp);
sineanalyzer_diff_wpwr1.Iin = _IQtoIQ15(sin_ramp2);
SINEANALYZER_DIFF_IQ_FUNC(&sineanalyzer_diff1);
```

  Alternatively the macro routine can be called as bellow:

```
SINEANALYZER_DIFF_IQ_MACRO(sineanalyzer_diff1);
```

- **Floating Point (F)**

  Step 1 – Add library header file to {ProjectName}-Includes.h

  Link Solar Library: (Solar_Lib_F.lib) located in:
  controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h "
```

  Step 2 – Create and add module structure to {ProjectName}-Main.c

```
SINEANALYZER_DIFF_wPWR_F sineanalyzer_diff_wpwr1;
```

  Step 3 – **Initialize module** in {ProjectName}-Main.c

```
SINEANALYZER_DIFF_wPWR_F_init(&sineanalyzer_diff_wpwr1);
sineanalyzer_diff_wpwr1.SampleFreq = ISR_FREQ;
```

```
sineanalyzer_diff_wpwr1.Threshold = (0.2);
sineanalyzer_diff_wpwr1.nsamplesMin = ((1.0)/ (GRID_FREQ+5)
      )*(ISR_FREQ);//=153@60Hz,20kHz;
sineanalyzer_diff_wpwr1.nsamplesMax = ((1.0)/(GRID_FREQ-5)
       )*(ISR_FREQ);//=181@60Hz,20kHz;
```

Step 4 – Using the module

```
sineanalyzer_diff_wpwr1.Vin = (sin_ramp);
sineanalyzer_diff_wpwr1.Iin = (sin_ramp2);
SINEANALYZER_DIFF_wPWR_F_FUNC(&sineanalyzer_diff_wpwr1);
```

Alternatively the macro routine can be called as bellow:

```
SINEANALYZER_DIFF_wPWR_F_MACRO(sineanalyzer_diff_wpwr1);
```

- **Control Law Accelerated Floating Point (CLA)**

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

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(sineanalyzer_diff_wpwr1, "CLADataRAM")
ClaToCpu_Volatile SINEANALYZER_DIFF_wPWR_CLA
sineanalyzer_diff_wpwr1;
```

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

```
extern ClaToCpu_Volatile SINEANALYZER_DIFF_wPWR_CLA
       sineanalyzer_diff_wpwr1;
```

Step 3 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {

      ….

      SINEANALYZER_DIFF_wPWR_CLA_init(sineanalyzer_diff_wpwr1);
      sineanalyzer_diff_wpwr1.SampleFreq = ISR_FREQ;
      sineanalyzer_diff_wpwr1.Threshold = (0.2);
      sineanalyzer_diff_wpwr1.nsamplesMin = ((1.0)/
            (GRID_FREQ+5))*(ISR_FREQ);//153@60Hz,10kHz;
      sineanalyzer_diff_wpwr1.nsamplesMax = ((1.0)/
            (GRID_FREQ-5))*(ISR_FREQ);//181@60Hz,10kHz;

      ….

}
```

The task is forced from {ProjectName}-Main.c by calling:

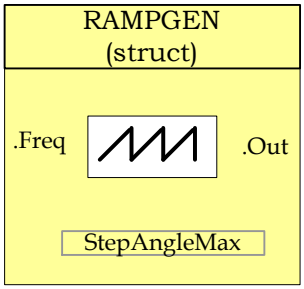
```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {

      ….

      sineanalyzer_diff_wpwr1.Vin = (sin_ramp);
      sineanalyzer_diff_wpwr1.Iin = (sin_ramp2);
      SINEANALYZER_DIFF_wPWR_CLA_MACRO(sineanalyzer_diff_wpwr1);

      ….

}
```

RAMPGEN — *Generates a RAMP signal*

Description: This software module generates a ramp signal of a desired frequency.



Module File: <*base_folder*>\RAMPGEN_(IQ/F/CLA).h

Technical: This module generates a ramp signal of a desired frequency.

Object Definition:

- **Fixed Point (IQ)**

```
//*********** Structure Definition ********//
typedef struct {
      int32  Freq;         // Input: Ramp frequency (pu)
      int32  StepAngleMax; // Parameter: Maximum step angle (pu)
      int32  Angle;        // Variable: Step angle (pu)
      int32  Out;          // Output: Ramp signal (pu)
} RAMPGEN_IQ;
```

- **Floating Point (F)**

```
//*********** Structure Definition ********//
typedef struct {
      float32  Freq;         // Input: Ramp frequency (pu)
      float32  StepAngleMax;// Parameter: Maximum step angle (pu)
      float32  Angle;        // Variable: Step angle (pu)
      float32  Out;          // Output: Ramp signal (pu)
} RAMPGEN_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//*********** Structure Definition ********//
typedef struct {
      float32  Freq;         // Input: Ramp frequency (pu)
      float32  StepAngleMax;// Parameter: Maximum step angle (pu)
      float32  Angle;        // Variable: Step angle (pu)
      float32  Out;          // Output: Ramp signal (pu)
} RAMPGEN_CLA;
```

Module interface Definition:

Net name	Type	Description	Acceptable Range
Freq	Input	Desired RAMP frequency	Q24 Float32
StepAngleMax	Input	Rate of calling the RAMP module /macro	Q24 Float32
Angle	Internal	Internal Angle Value	Q24 Float32
Out	Output	RAMP output value	Q24 Float32

Usage:

- **Fixed Point (IQ)**

  Step 1 – Include library in {ProjectName}-Includes.h

  Link Solar Library: (Solar_Lib_IQ.lib) located in:
  controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

  Step 2 – Create and add module structure to {ProjectName}-Main.c

```
RAMPGEN_IQ Ramp;
```

  Step 3 – **Initialize module** in {ProjectName}-Main.c

```
RAMPGEN_IQ_init(&Ramp);
Ramp.Freq = _IQ24(GRID_FREQ);
Ramp.StepAngleMax = _IQ24(1.0/ISR_FREQUENCY);
```

  Step 4 – **Using the module**

```
RAMPGEN_IQ_MACRO(Ramp);
```

  Alternatively the macro routine can be called as bellow:

```
RAMPGEN_IQ_FUNC(&Ramp);
```

- **Floating Point (F)**

  Step 1 – Include library in {ProjectName}-Includes.h

  Link Solar Library: (Solar_Lib_F.lib) located in:
  controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

  Step 2 – Create and add module structure to {ProjectName}-Main.c

```
RAMPGEN_F Ramp;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
RAMPGEN_F_init(&Ramp);
Ramp.Freq = (float)(GRID_FREQ);
Ramp.StepAngleMax = (float)(1.0/ISR_FREQUENCY);
```

Step 4 – Using the module

```
RAMPGEN_F_MACRO(Ramp);
```

Alternatively the macro routine can be called as bellow:

```
RAMPGEN_F_FUNC(&Ramp);
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(Ramp,"CLADataRAM")
ClaToCpu_Volatile RAMPGEN_CLA Ramp;
```

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

```
extern ClaToCpu_Volatile RAMPGEN_CLA Ramp;
```

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {

        ….
        RAMPGEN_CLA_init(Ramp);
        Ramp.Freq = (float32)(GRID_FREQUENCY);
        Ramp.StepAngleMax = (float32)(1.0/ISR_FREQUENCY);
        ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

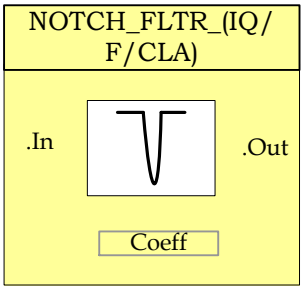
```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {

        ….
        RAMPGEN_CLA_MACRO(Ramp);
        ….
}
```

NOTCH_FLTR — *Notch Filter*

Description: This software module filters out the input signal with a notch filter, thus removing a particular frequency from the input signal.



Module File: <*base_folder*>\NOTCH_FLTR_(IQ/F/CLA).h

Technical: This module filters out the input signal with a notch filter thus getting rid of a particular frequency component from the input signal. Design of the notch filter is achieved using discretizing the notch filter equation from s domain:

$$H_{nf}(s) = \frac{s^2 + 2\zeta_2\omega_n s + \omega_n^2}{s^2 + 2\zeta_1\omega_n s + \omega_n^2}$$ where $\zeta_2 << \zeta_1$

Using zero order hold i.e. $s = \frac{(z-1)}{T}$ we get

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

Hence the coefficients for the notch filter can be changed depending on the frequency that is desired to be removed.

Object Definition:

- **Fixed Point (IQ)**

```
/*********** Structure Definitions ***********/
typedef struct{
    _iq24  B2_notch;
    _iq24  B1_notch;
    _iq24  B0_notch;
    _iq24  A2_notch;
    _iq24  A1_notch;
}NOTCH_COEFF_IQ;

typedef struct{
    _iq24  Out1;
    _iq24  Out2;
    _iq24  In;
    _iq24  In1;
    _iq24  In2;
    _iq24  Out;
}NOTCH_VARS_IQ;
```

- **Floating Point (F)**

```
/*********** Structure Definitions ***********/
typedef struct{
    float32      B2_notch;
    float32      B1_notch;
    float32      B0_notch;
    float32      A2_notch;
    float32      A1_notch;
}NOTCH_COEFF_F;

typedef struct{
    float32      Out1;
    float32      Out2;
```

```
    float32      In;
    float32      In1;
    float32 In2;
    float32      Out;
}NOTCH_VARS_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
/*********** Structure Definitions ***********/
typedef struct{
    float32      B2_notch;
    float32      B1_notch;
    float32      B0_notch;
    float32      A2_notch;
    float32      A1_notch;
}NOTCH_COEFF_CLA;

typedef struct{
    float32      Out1;
    float32      Out2;
    float32      In;
    float32      In1;
    float32      In2;
    float32      Out;
}NOTCH_VARS_CLA;
```

Module interface Definition:

Net name	Type	Description	Acceptable Range
In,In1,In2	Input	Input signal and history	Q24 Float32
Out,Out1,Out2	Output	Output signal and history	Q24 Float32

Usage:

- **Fixed Point (IQ)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
NOTCH_COEFF_IQ notch_TwiceGridFreq;
NOTCH_VARS_IQ Bus_Volt_notch;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
      // NOTCH COEFF init for filtering twice the grid frequency component
from bus voltage
            NOTCH_FLTR_IQ_VARS_init(&Bus_Volt_notch);
            c1 = 0.1;
            c2 = 0.00001;
            NOTCH_FLTR_IQ_COEFF_Update(((float)(1.0/ISR_FREQUENCY)),
            (float)(2*PI*GRID_FREQ*2),(float)c2,(float)c1,
            &notch_TwiceGridFreq);
```

Step 4 – **Using the module**

```
            Bus_Volt_notch.In=vbus_meas_inst;
            NOTCH_FLTR_IQ_ASM(&Bus_Volt_notch,&notch_TwiceGridFreq);
```

- **Floating Point (F)**

Step 1 – Include library in {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
            #include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
            NOTCH_COEFF_F notch_TwiceGridFreq;
            NOTCH_VARS_F Bus_Volt_notch;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
// NOTCH COEFF init for filtering twice the grid frequency component from bus
voltage
            NOTCH_FLTR_F_VARS_init(&Bus_Volt_notch);
            c1 = 0.1;
            c2 = 0.00001;
            NOTCH_FLTR_F_COEFF_Update(((float)(1.0/ISR_FREQUENCY)),
            (float)(2*PI*GRID_FREQ*2),(float)c2,(float)c1,
            &notch_TwiceGridFreq);
```

Step 4 – Using the module

```
            Bus_Volt_notch.In=vbus_meas_inst;
            NOTCH_FLTR_F_run(&Bus_Volt_notch,&notch_TwiceGridFreq);
```

- **Control Law Accelerated Floating Point (CLA)**

Step 1 – Include library in {ProjectName}-CLA_Shared.h

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(Ramp,"CLADataRAM")
ClaToCpu_Volatile RAMPGEN_CLA Ramp;
```

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

```
extern ClaToCpu_Volatile RAMPGEN_CLA Ramp;
```

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 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
        ….
        RAMPGEN_CLA_init(Ramp);
        Ramp.Freq = (float32)(GRID_FREQUENCY);
        Ramp.StepAngleMax = (float32)(1.0/ISR_FREQUENCY);
        ….
}
```

The task is forced from {ProjectName}-Main.c by calling:

```
Cla1ForceTask8andWait();
```

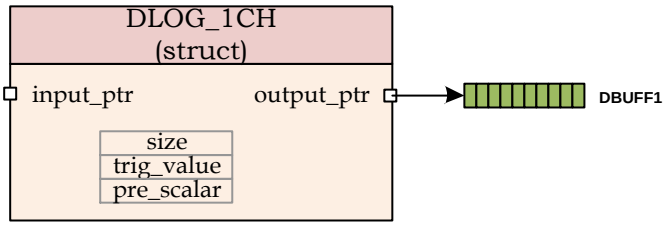
Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {

        ….
        RAMPGEN_CLA_MACRO(Ramp);

        ….

}
```


4.5 Data Logger Modules

DLOG_1CH — *Single Channel Data logger*

Description: This software module performs data logging to emulate an oscilloscope in software to graphically observe a system variable. The data is logged in a buffer that can be viewed as a graph to observe the system variables as waveforms.



Module File: DLOG_1CH.h

Technical: This software module performs data logging over data stored in IQ format in a location pointed to by input_ptr. The input variable value is then scaled to Q15 format and stored in the array pointed to by output_ptr.

The data logger is triggered at the positive edge of the value pointed by the input pointer. The trigger value is programmable by writing the Q24 trig_value to the module variable. The size of the data logger has to be specified and the module can be configured to log data every n number module call, by specifying a pre scalar value

Object Definition:

- **Fixed Point (IQ)**

```
//************ Structure Definition ********//
typedef struct{
      int32 *input_ptr;
      int16 *output_ptr;
      int32 prev_value;
      int32 trig_value;
      int16 status;
      int16 pre_scalar;
      int16 skip_count;
      int16 size;
      int16 count;

}DLOG_1CH_IQ;
```

- **Floating Point (F)**

```
//************ Structure Definition ********//
typedef struct{
      float32 *input_ptr;
```

```
        float32 *output_ptr;
        float32 prev_value;
        float32 trig_value;
        int16 status;
        int16 pre_scalar;
        int16 skip_count;
        int16 size;
        int16 count;
}DLOG_1CH_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//************ Structure Definition ********//
typedef struct{
        float32 *input_ptr;
        float32 *output_ptr;
        float32 prev_value;
        float32 trig_value;
        int16_t status;
        int16_t pre_scalar;
        int16_t skip_count;
        int16_t size;
        int16_t count;
}DLOG_1CH_CLA;
```

Module interface Definition:

Net name	Type	Description	Acceptable Range
input_ptr	Input	Pointer to the value being logged in the data logger	NA
output_ptr	Input	Pointer to the buffer where the logged values are stored	NA
trig_value	Input	Value to begin data logging from	Q15
status	Output	Indicates status of data logger i.e. waiting for trigger, or logging data	NA
pre_scalar	Input	No of samples skipped by the data logger	NA
skip_count	Internal	Variable used to keep track of pre_scalar	NA
size	Input	No of data points being logged, this is the size of the output data buffer	NA
count	Internal	When in data logging phase it keeps count of the no of samples already logged	NA

Usage:

- **Fixed Point (IQ)**

Step 1 – Add library header file to {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
DLOG_1CH_IQ dlog_1ch1;
int16 DBUFF_1CH1[100];
int16 dval1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
DLOG_1CH_IQ_init(&dlog_1ch1);
dlog_1ch1.input_ptr = &dval1; //data value
dlog_1ch1.output_ptr = &DBUFF_1CH1[0];
dlog_1ch1.size = 100;
dlog_1ch1.pre_scalar = 5;
dlog_1ch1.trig_value = desiredTrigVal;
dlog_1ch1.status = 2;
```

Step 4 – **Using the module**

```
dval1=_IQtoIQ15(Ramp1.Out);
DLOG_1CH_IQ_FUNC(&dlog_1ch1);
```

Alternatively the macro routine can be called as bellow:

```
DLOG_1CH_IQ_MACRO(dlog_1ch1);
```

- **Floating Point (F)**

Step 1 – Add library header file to {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
DLOG_1CH_F dlog_1ch1;
float32 DBUFF_1CH1[100];
float32 dval1;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
DLOG_1CH_F_init(&dlog_1ch1);
dlog_1ch1.input_ptr = &dval1; //data value
dlog_1ch1.output_ptr = &DBUFF_1CH1[0];
dlog_1ch1.size = 100;
```

```
dlog_1ch1.pre_scalar = 5;
dlog_1ch1.trig_value = desiredTrigVal;
dlog_1ch1.status = 2;
```

Step 4 – Using the module

```
dval1= (Ramp1.Out);
DLOG_1CH_F_FUNC(&dlog_1ch1);
```

Alternatively the macro routine can be called as bellow:

```
DLOG_1CH_F_MACRO(dlog_1ch1);
```

- **Control Law Accelerated Floating Point (CLA)**

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

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(dlog_1ch1, "CLADataRAM")
ClaToCpu_Volatile DLOG_1CH_CLA dlog_1ch1;
#pragma DATA_SECTION(DBUFF_1CH1, "CLADataRAM")
ClaToCpu_Volatile float32 DBUFF_1CH1[50];
#pragma DATA_SECTION(dval1, "CLADataRAM")
ClaToCpu_Volatile float32 dval1;
```

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

```
extern ClaToCpu_Volatile DLOG_1CH_CLA dlog_1ch1;
extern ClaToCpu_Volatile float32 DBUFF_1CH1[50];
extern ClaToCpu_Volatile float32 dval1;
```

Step 3 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
      ….
      DLOG_1CH_CLA_init(dlog_1ch1);
      dlog_1ch1.input_ptr = &dval1; //data value
      dlog_1ch1.output_ptr = &DBUFF_1CH1[0];
      dlog_1ch1.size = 50;
      dlog_1ch1.pre_scalar = 5;
      dlog_1ch1.trig_value = desiredTrigVal;
      dlog_1ch1.status = 2;
      ….
}
```

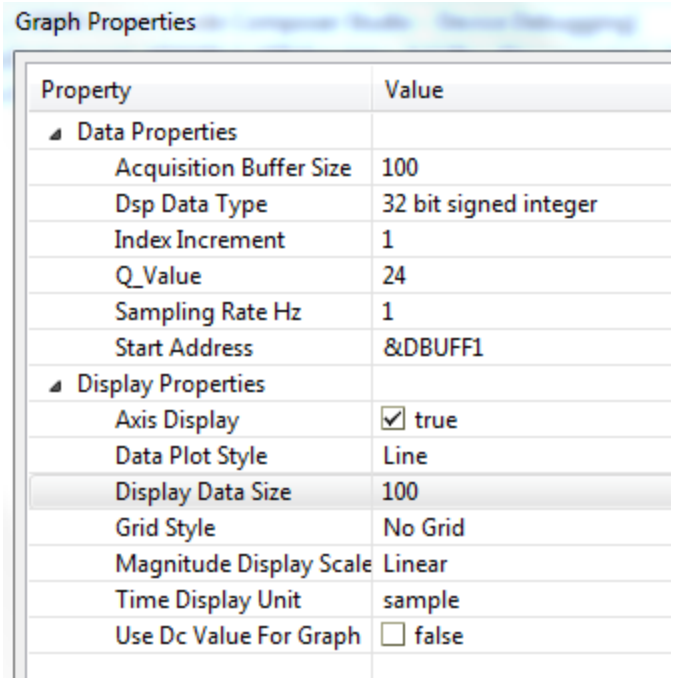
The task is forced from {ProjectName}-Main.c by calling:

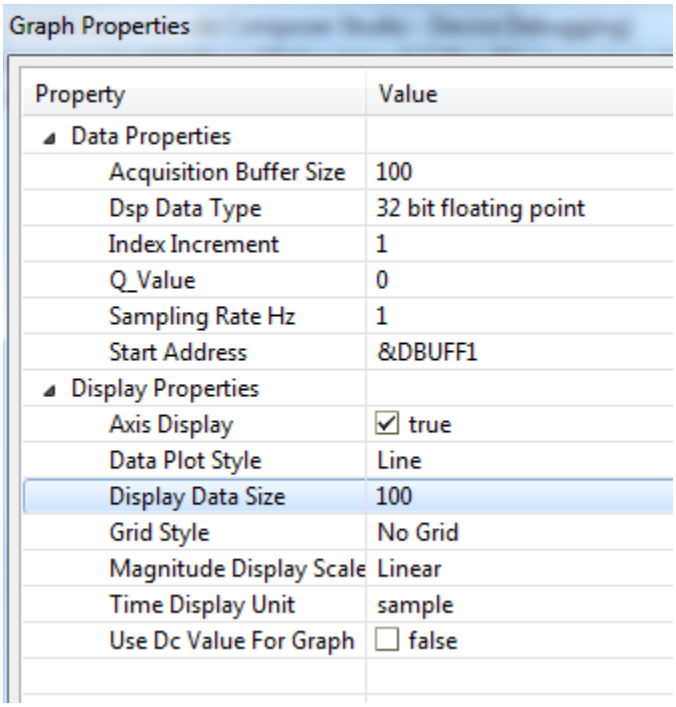
```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
      ….
      dval1 = (Ramp1.Out);
      DLOG_1CH_CLA_MACRO(dlog_1ch1);
      ….
}
```

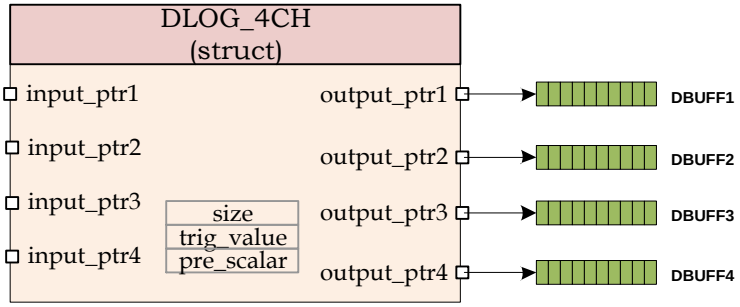
Step 6 – Graphing the buffer value in CCS – In the Debug view go to Tools-> Graph -> SingleTime , in the pop up window populate as follows:





This will now show a Data Logger Graph window in the CCS debug panel.

Description: This software module performs data logging to emulate an oscilloscope in software to graphically observe system variables. The data is logged in the buffers and viewed as graphs in graph windows to observe the system variables as waveforms.



Module File: DLOG_4CH.h

Technical: This software module performs data logging over data stored in IQ format in a location pointed to by input_ptr1-4. The input variable value is then scaled to Q15 format and stored in the array pointed to by output_ptr1-4.

The data logger is triggered at the positive edge of the value pointed by the input pointer1. The trigger value is programmable by writing the Q24 trig_value to the module variable. The size of the data logger has to be specified and the module can be configured to log data every n number module call, by specifying a pre scalar value

Object Definition:

- **Fixed Point (IQ)**

```
//************ Structure Definition ********//
typedef struct{
      int32 *input_ptr1;
      int32 *input_ptr2;
      int32 *input_ptr3;
      int32 *input_ptr4;
      int16 *output_ptr1;
      int16 *output_ptr2;
      int16 *output_ptr3;
      int16 *output_ptr4;
      int16 prev_value;
      int16 trig_value;
      int16 status;
      int16 pre_scalar;
      int16 skip_count;
      int16 size;
      int16 count;
}DLOG_4CH_IQ;
```


- **Floating Point (F)**

```
//*********** Structure Definition ********//
typedef struct{
      float32 *input_ptr1;
      float32 *input_ptr2;
      float32 *input_ptr3;
      float32 *input_ptr4;
      float32 *output_ptr1;
      float32 *output_ptr2;
      float32 *output_ptr3;
      float32 *output_ptr4;
      float32 prev_value;
      float32 trig_value;
      int16 status;
      int16 pre_scalar;
      int16 skip_count;
      int16 size;
      int16 count;
}DLOG_4CH_F;
```

- **Control Law Accelerated Floating Point (CLA)**

```
//*********** Structure Definition ********//
typedef struct{
      float32 *input_ptr1;
      float32 *input_ptr2;
      float32 *input_ptr3;
      float32 *input_ptr4;
      float32 *output_ptr1;
      float32 *output_ptr2;
      float32 *output_ptr3;
      float32 *output_ptr4;
      float32 prev_value;
      float32 trig_value;
      int16_t status;
      int16_t pre_scalar;
      int16_t skip_count;
      int16_t size;
      int16_t count;
}DLOG_4CH_CLA;
```

Module interface Definition:

Net name	Type	Description	Acceptable Range
input_ptr1-4	Input	Pointer to the value being logged in the data logger	NA
output_ptr1-4	Input	Pointer to the buffer where the logged values are stored	NA
trig_value	Input	Value to begin data logging from for input ptr 1	Q15

status	Output	Indicates status of data logger i.e. waiting for trigger, or logging data	NA
pre_scalar	Input	No of samples skipped by the data logger	NA
skip_count	Internal	Variable used to keep track of pre_scalar	NA
size	Input	No of data points being logged, this is the size of the output data buffer	NA
count	Internal	When in data logging phase it keeps count of the no of samples already logged	NA

Usage:

- **Fixed Point (IQ)**

Step 1 – Add library header file to {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_IQ.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\IQ\lib

```
#include "Solar_IQ.h"
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
DLOG_4CH_IQ dlog_4ch1;
int16 DBUFF_4CH1[100];
int16 DBUFF_4CH2[100];
int16 DBUFF_4CH3[100];
int16 DBUFF_4CH4[100];
int16 dval1;
int16 dval2;
int16 dval3;
int16 dval4;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
DLOG_4CH_IQ_init(&dlog_4ch1);
dlog_4ch1.input_ptr1 = &dval1;      //data value
dlog_4ch1.input_ptr2 = &dval2;
dlog_4ch1.input_ptr3 = &dval3;
dlog_4ch1.input_ptr4 = &dval4;
dlog_4ch1.output_ptr1 = &DBUFF_4CH1[0];
dlog_4ch1.output_ptr2 = &DBUFF_4CH2[0];
dlog_4ch1.output_ptr3 = &DBUFF_4CH3[0];
dlog_4ch1.output_ptr4 = &DBUFF_4CH4[0];
dlog_4ch1.size = 100;
```

```
dlog_4ch1.pre_scalar = 5;
dlog_4ch1.trig_value = desiredTrigValue;
dlog_4ch1.status = 2;
```

Step 4 – **Using the module**

```
dval1=_IQtoIQ15(Ramp1.Out);
dval2=_IQtoIQ15(Vin);
dval3=_IQtoIQ15(spll1.Out);
dval4=_IQtoIQ15(ramp_sin);
DLOG_4CH_IQ_FUNC(&dlog_4ch1);
```

Alternatively the macro routine can be called as bellow:

```
DLOG_4CH_IQ_MACRO(dlog_4ch1);
```

- **Floating Point (F)**

Step 1 – Add library header file to {ProjectName}-Includes.h

Link Solar Library: (Solar_Lib_F.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\F\lib

```
#include "Solar_F.h "
```

Step 2 – Create and add module structure to {ProjectName}-Main.c

```
DLOG_4CH_F dlog_4ch1;
float32 DBUFF_4CH1[100];
float32 DBUFF_4CH2[100];
float32 DBUFF_4CH3[100];
float32 DBUFF_4CH4[100];
float32 dval1;
float32 dval2;
float32 dval3;
float32 dval4;
```

Step 3 – **Initialize module** in {ProjectName}-Main.c

```
DLOG_4CH_F_init(&dlog_4ch1);
dlog_4ch1.input_ptr1 = &dval1;
dlog_4ch1.input_ptr2 = &dval2;
dlog_4ch1.input_ptr3 = &dval3;
dlog_4ch1.input_ptr4 = &spll1.theta[0];
dlog_4ch1.output_ptr1 = &DBUFF_4CH1[0];
```

```
dlog_4ch1.output_ptr2 = &DBUFF_4CH2[0];
dlog_4ch1.output_ptr3 = &DBUFF_4CH3[0];
dlog_4ch1.output_ptr4 = &DBUFF_4CH4[0];
dlog_4ch1.size = 100;
dlog_4ch1.pre_scalar = 5;
dlog_4ch1.trig_value = desiredTrigValue;
dlog_4ch1.status = 2;
```

Step 4 – Using the module

```
dval1= (Ramp1.Out);
dval2= (Vin);
dval3= (spll1.Out);
dval4= (ramp_sin);
DLOG_4CH_F_FUNC(&dlog_4ch1);
```

Alternatively the macro routine can be called as bellow:

```
DLOG_4CH_F_MACRO(dlog_4ch1);
```

- **Control Law Accelerated Floating Point (CLA)**

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

Link Solar Library: (Solar_Lib_CLA.lib) located in:
controlSUITE\development\libs\app_libs\solar\v1.2\CLA\lib

```
#include "Solar_CLA.h"
```

Step 2 – Create and add module structure to {ProjectName}-CLA_Tasks.c

Declare the variable and specify an appropriate location in CLA memory.

```
#pragma DATA_SECTION(dlog_4ch1, "CLADataRAM")
ClaToCpu_Volatile DLOG_4CH_CLA dlog_4ch1;
#pragma DATA_SECTION(DBUFF_4CH1, "CLADataRAM")
ClaToCpu_Volatile float32 DBUFF_4CH1[50];
#pragma DATA_SECTION(DBUFF_4CH2, "CLADataRAM")
ClaToCpu_Volatile float32 DBUFF_4CH2[50];
#pragma DATA_SECTION(DBUFF_4CH3, "CLADataRAM")
ClaToCpu_Volatile float32 DBUFF_4CH3[50];
#pragma DATA_SECTION(DBUFF_4CH4, "CLADataRAM")
ClaToCpu_Volatile float32 DBUFF_4CH4[50];
#pragma DATA_SECTION(dval1, "CLADataRAM")
ClaToCpu_Volatile float32 dva1;
```

```
#pragma DATA_SECTION(dval2, "CLADataRAM")
ClaToCpu_Volatile float32 dval2;
#pragma DATA_SECTION(dval3, "CLADataRAM")
ClaToCpu_Volatile float32 dval3;
#pragma DATA_SECTION(dval4, "CLADataRAM")
ClaToCpu_Volatile float32 dval4;
```

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

```
extern ClaToCpu_Volatile DLOG_4CH_CLA dlog_4ch1;
extern ClaToCpu_Volatile float32 DBUFF_4CH1[50];
extern ClaToCpu_Volatile float32 DBUFF_4CH2[50];
extern ClaToCpu_Volatile float32 DBUFF_4CH3[50];
extern ClaToCpu_Volatile float32 DBUFF_4CH4[50];
extern ClaToCpu_Volatile float32 dva1;
extern ClaToCpu_Volatile float32 dval2;
extern ClaToCpu_Volatile float32 dval3;
extern ClaToCpu_Volatile float32 dval4;
```

Step 3 – Configure CLA memory in {ProjectName}-Main.c – Assign memory to CLA. Message RAM and data RAM must be configured by the CPU (this may change from device to device):

```
// 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 4 – **Initialize module** in {ProjectName}-CLA_Tasks.c – Initialization can be achieved by calling the init macro from Task 8 in CLA.

```
interrupt void Cla1Task8(void) {
    ….
    DLOG_4CH_CLA_init(dlog_4ch1);
    dlog_4ch1.input_ptr1 = &dval1;
    dlog_4ch1.input_ptr2 = &dval2;
    dlog_4ch1.input_ptr3 = &dval3;
    dlog_4ch1.input_ptr4 = &dval4;
    dlog_4ch1.output_ptr1 = &DBUFF_4CH1[0];
```

```
        dlog_4ch1.output_ptr2 = &DBUFF_4CH2[0];
        dlog_4ch1.output_ptr3 = &DBUFF_4CH3[0];
        dlog_4ch1.output_ptr4 = &DBUFF_4CH4[0];
        dlog_4ch1.size = 50;
        dlog_4ch1.pre_scalar = 5;
        dlog_4ch1.trig_value = desiredTrigValue;
        dlog_4ch1.status = 2;
        ….
}
```

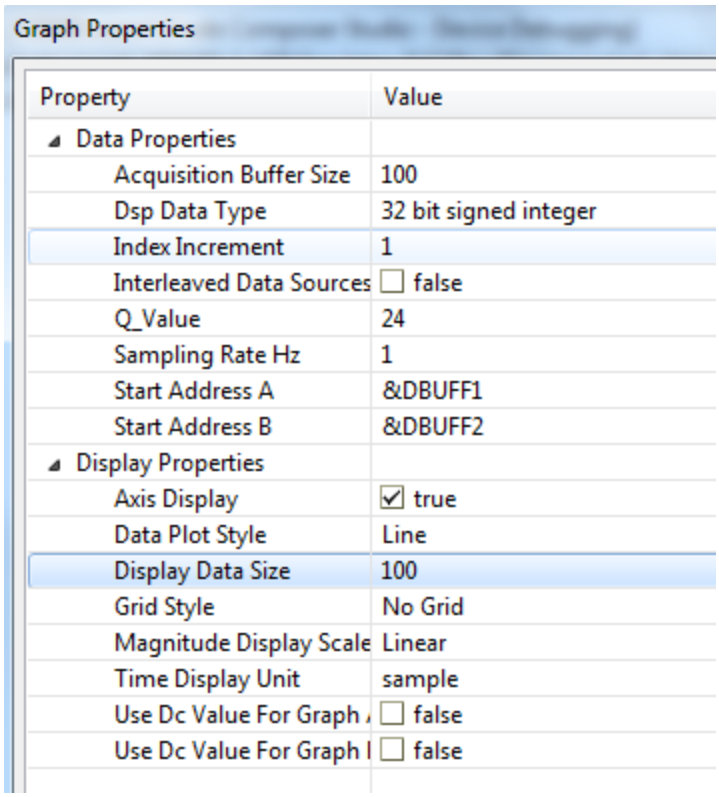
The task is forced from {ProjectName}-Main.c by calling:

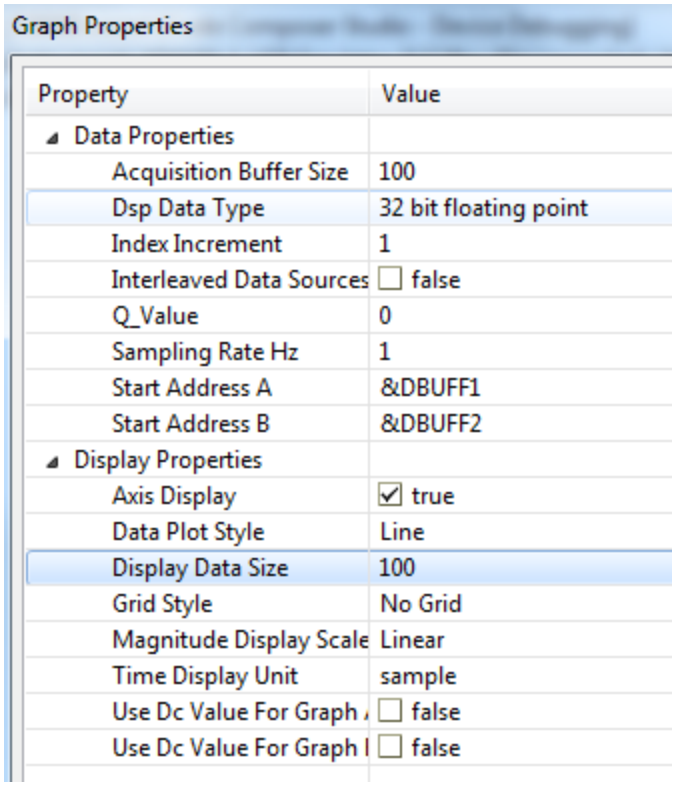
```
Cla1ForceTask8andWait();
```

Step 5 – **Using the module –** The run time macro is called in a task that is periodically triggered like an ISR.

```
interrupt void Cla1Task1(void) {
        ….
        dval1= (Ramp1.Out);
        dval2= (Vin);
        dval3= (spll1.Out);
        dval4= (ramp_sin);
        DLOG_4CH_F_MACRO(&dlog_4ch1);
        ….
}
```

Step 6 – Graphing the buffer value in CCS – In the Debug view go to Tools-> Graph -> DualTime , in the pop up window populate as follows:





This will now show a Data Logger Graph window in the CCS debug panel, two graphs will be shown one with the DBUFF1 value and other with the DBUFF2 value.

Chapter 5. Appendix

5.1. PR Controller Mapped to 2p2z

PR controller is a popular controller type used in design of PV inverters. The following section details how the PR controller coefficients can be mapped into a 2p2z structure available in the solar library.

$G_{PR} = K_p + K_i \frac{s}{s^2 + \omega^2}$ where ω is the resonant pole frequency in radians

Using bi-linear transformation $s = \frac{2}{T}\frac{(z-1)}{(z+1)}$

$$G_{PR}(z) = K_p + \frac{K_i \frac{2(z-1)}{T(z+1)}}{\frac{4(z-1)^2}{T^2(z+1)^2} + \omega^2} = \frac{\left(K_p + \frac{2K_iT}{(4+T^2\omega^2)}\right) + \left(2K_p \frac{(T^2\omega^2-4)}{(T^2\omega^2+4)}\right)z^{-1} + \left(K_p - \frac{2K_iT}{(4+T^2\omega^2)}\right)z^{-2}}{1 + 2\left(\frac{(T^2\omega^2-4)}{(T^2\omega^2+4)}\right)z^{-1} + z^{-2}}$$

Comparing this with the generic structure of 2p2z as implemented in the solar lib

$$\frac{U(z)}{E(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}}$$

We get : $b_0 = \left(K_p + \frac{2K_iT}{(4+T^2\omega^2)}\right)$, $b_1 = \left(2K_p \frac{(T^2\omega^2-4)}{(T^2\omega^2+4)}\right)$, $b_2 = \left(K_p - \frac{2K_iT}{(4+T^2\omega^2)}\right)$,

$a_1 = -2\left(\frac{(T^2\omega^2-4)}{(T^2\omega^2+4)}\right)$, $a_2 = -1$

Chapter 6. Revision History

Version	Date	Notes
V1.0	Jan, 31 2011	First Release of Solar Library
V1.1	June, 5 2012	Value for SPLL theta corrected, document corrected for correct range of SPLL block variables
V1.2	Jan, 2014	Correction to SPLL_1ph block, Added adaptive notch filter to SPLL_1ph Addition of SOGI PLL module Addition of SPLL_3ph_SRF, SPLL_3ph_DDSRF, MPPT_INCC_I, RAMPGEN Adding 2p2z, 3p3z Combining documents for float, CLA and IQ Adding Clarke, Park, iClarke, iPark, ABC_DQ0 transforms Adding SineAnalyzerDiffwPower Modified SineAnalyzerDiff to resilient to ZCD noise Adding Notch_Fltr