CLA Digital Power Library

v3.0

October 2010

Module User's Guide

CLA 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 is the trademark of Texas Instruments Incorporated.
All other trademarks mentioned herein are property of their respective companies

Acronyms

DPlib: Digital Power library. A set of functions and macros used to facilitate the design of digital power control systems.

C28x: Refers to devices with the C28x CPU core.

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

DPlib CLA: Digital Power library, A set of functions and macros used to facilitate the design of digital power control systems using CLA.

IQmath: Fixed-point mathematical functions in C that allow fixed point numbers to be treated in a floating point manner.

Q-math: Fixed point numeric format that defines the number of decimal and integer bits.

Blocks/Macros : Used interchangeably for DPLib Functions that can be connected together to forma system.

Index

Chapter 1. Introduction

1.1. Introduction

Texas Instruments Digital Power Control Law Accelerator (DP CLA) library is designed to enable flexible and efficient coding of digital power supply applications using the CLA co-processor present in some C2000 family of devices. An important consideration for power supply application is their relatively high control loop rate, which imposes certain restrictions on the way the software can be written. In particular, the designer must take care to ensure the real-time portion of the code, normally contained within an Interrupt Service Routine (ISR), must execute in as few cycles as possible. The CLA is an independent, fully-programmable, 32-bit floating-point math processor, with low interrupt latency which allows ADC samples to be read "just-in-time". This significantly reduces the ADC sample to output delay to enable faster system response and higher MHz control loops. This allows CLA to be ideal for many power supply applications. As the CLA is a co-processor to the main C28x core, once the CLA is configured to service time-critical control loops, the main CPU is free to perform other system tasks such as communication and diagnostics.

The DP CLA library provides a software structure that is flexible and adaptable and completely configurable. The DP CLA library is constructed in a modular form similar to the C28x based DP library, with macro functions encapsulated in re-usable code blocks which can be connected together to build any desired software structure. The library enables the designer to experiment with various control loop layouts, and to monitor software variables at various points in the code to confirm correct operation of the system. The major difference between DP CLA and DP C28x Library is the memory allocation for the different components of the modules in the power library. As is the case with any multi processor system, while using the CLA care must be taken when allocating memory. This is explained later in detail.

The strategy of using blocks encourages the use of block diagrams to plan out the software structure of the control loop before the code is written. An example of a simple block diagram showing the connection of three DP CLA library modules to form a closed loop system is shown below.

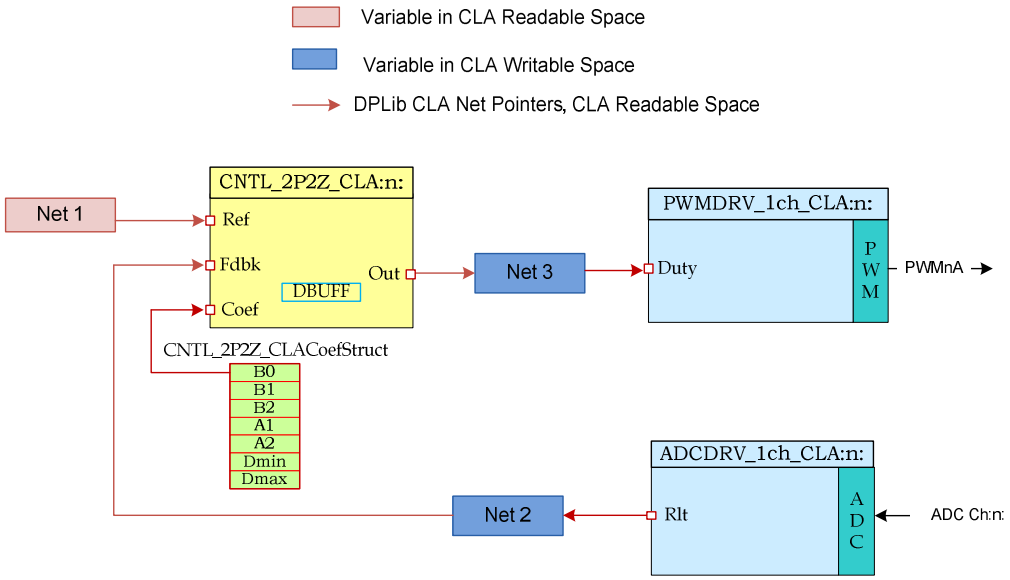


Figure 1 Closed Loop System Using CLA DPlib Blocks

In the example above, three library macro-blocks are connected: an ADC driver, a second order digital controller, and a PWM driver. The labels “Net1”, Net2” and “Net3” correspond to software variables which form the connection points, or “nodes”, in the diagram. The input and output terminals of each block may be connected to nodes by a simple method of C pointer assignment in software. In this way, designs may be rapidly re-configured to experiment with different software configurations.

Library blocks have been color coded: “turquoise” represents blocks that interface to physical device hardware, such as an A/D converter, while “yellow” indicates blocks which are independent of hardware. This distinction is important, since hardware interface blocks must be configured to match only those features present on the device. In particular, care should be taken to avoid creating blocks which access the same hardware (for example two blocks driving the same PWM output may give undesirable results!).

In addition to color coding of the blocks based on functions being performed, the “nets” i.e. the software variables and the net pointers, are also color coded. “Red” is the color for variables residing in the CLA readable space and “Blue” is the color of the variables residing in the CLA writable space. More details on CLA writable and readable spaces can be found in the section “Understanding Memory Allocation for CLA”.

All the blocks require initialization prior to use, and must be configured by connecting their terminals to the desired net nodes. The initialization and configuration process is described in Chapter 3.

As CLA has floating point unit the net variables are stored in single precision floating point format and floating point math is used internally by the blocks for computation. This is in contrast to the C28x Digital Power Library for F2803x and F2802x devices, where a Q24 format is used as the C28x present on these devices is a fixed point core.

Once the blocks have been initialized and connected, they can be configured to be executed in a particular CLA Task. The CLA task can be triggered repeatedly by a PWM or ADC interrupt.

Chapter 2. Installing the DP CLA Library

2.1. DPCLA Library Package Contents

The TI Digital Power CLA library consists of the following components:

- C initialization functions
- Assembly macros files
- An assembly file containing a macro initialization function and a real-time Run functions.
- An example CCS project showing the connection and use of DPCLA library blocks.
- Documentation

2.2. How to Install the Digital Power CLA Library

The DPCLA library is distributed through the controlSUITE installer. The user must select the Digital Power 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\digital_power\<device>

…where <device> is the C28x platform. The following sub-directory structure is used:

<base>\asm	Contains assembly macros and ISR module
<base>\doc	Contains this file
<base>\C	C initialization files
<base>\include>	contains DPLib header file

The installation also installs a template project using the DPLib for the device inside the controlSUITE directory

controlSUITE\development_kits\TemplateProjects_<ver. No.>\
DPLibTemplate-<device_name> _<ver. No.>

These template projects can be quickly modified to start a new project using the DPlib.

Note, when power library is installed, both CLA and C28x Library blocks are installed in the device directory if CLA exists on the device.

2.3. Naming Convention

The initialization and execution code for each DP CLA library macro is contained in a separate assembly include file. An example of the naming convention used comes in two types as shown below:

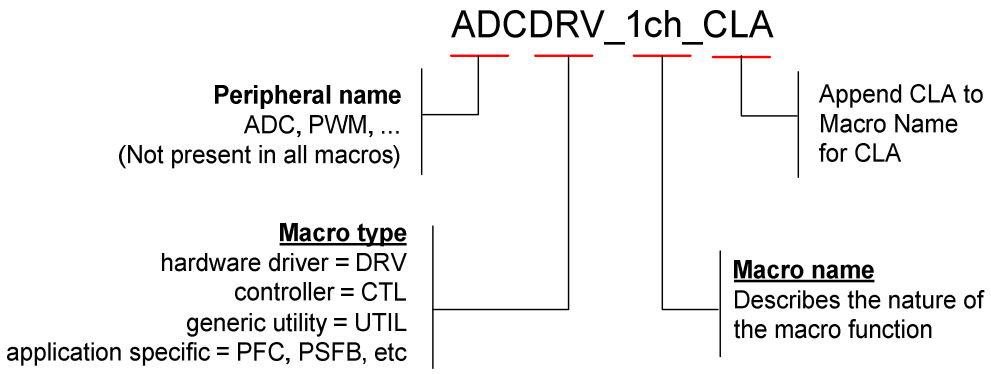


Figure 2 Function Naming Convention

Include files, initialization functions, and execution macros share the same naming. In the above example, these are…

Include file: `ADCDRV_1ch_CLA.asm`

Init function: `ADCDRV_1ch_CLA_INIT n`

Include file: `ADCDRV_1ch_CLA n`

Where `n` refers to the instance number of the macro.

Note: In case of peripheral drivers (i.e. the ADC drivers and the PWM drivers) the instance number also implies the peripheral number the DP library macro drives on the device (an exception to this is the ADCDRV_4ch macro). For example

`ADCDRV_1ch_CLA 0` normalize AdcResult0 of the ADC Peripheral
`PWMDRV_1ch_CLA 3` drive the EPWM3 peripheral present on the device.

Chapter 3. Using Digital Power CLA Library

3.1. Library Description and Overview

Typical CLA user software will consist of a main framework file written in C, a single Task Service Routine (TSR) written in assembly for CLA. The C framework contains code to configure the device hardware and the CLA, connect the library macros and configure peripheral interrupts PWM or ADC to trigger the desired CLA task. Once the CLA task is configured the CLA operates independently of the C28x core, thus freeing up C28x to do other tasks.

Conceptually, the process of setting up and using the DP library can be broken down into three parts.

1 Initialization Macro blocks are initialized from the C environment using a C callable function ("`DPL_CLAInit()`") which is contained in the assembly file `{ProjectName}-DPL- CLA.asm`. This function is prototyped in the library header file "`DPlib.h`" which must be included in the main C file.

2 Configuration C pointers of the macro block terminals are assigned to net nodes to form the desired control structure. Net nodes are 32-bit integer variables declared in the C framework. Note names of these net nodes are not dependent on the macro block.

3 Execution Macro block code is executed in the assembly CLA task (1 through 7, note task 8 is reserved for initialization by the Digital Power Library). This function is defined in the "`{ProjectName}-DPL-CLA.asm`".

An example of this process and the relationship between the main C file and CLA assembly file is shown below.

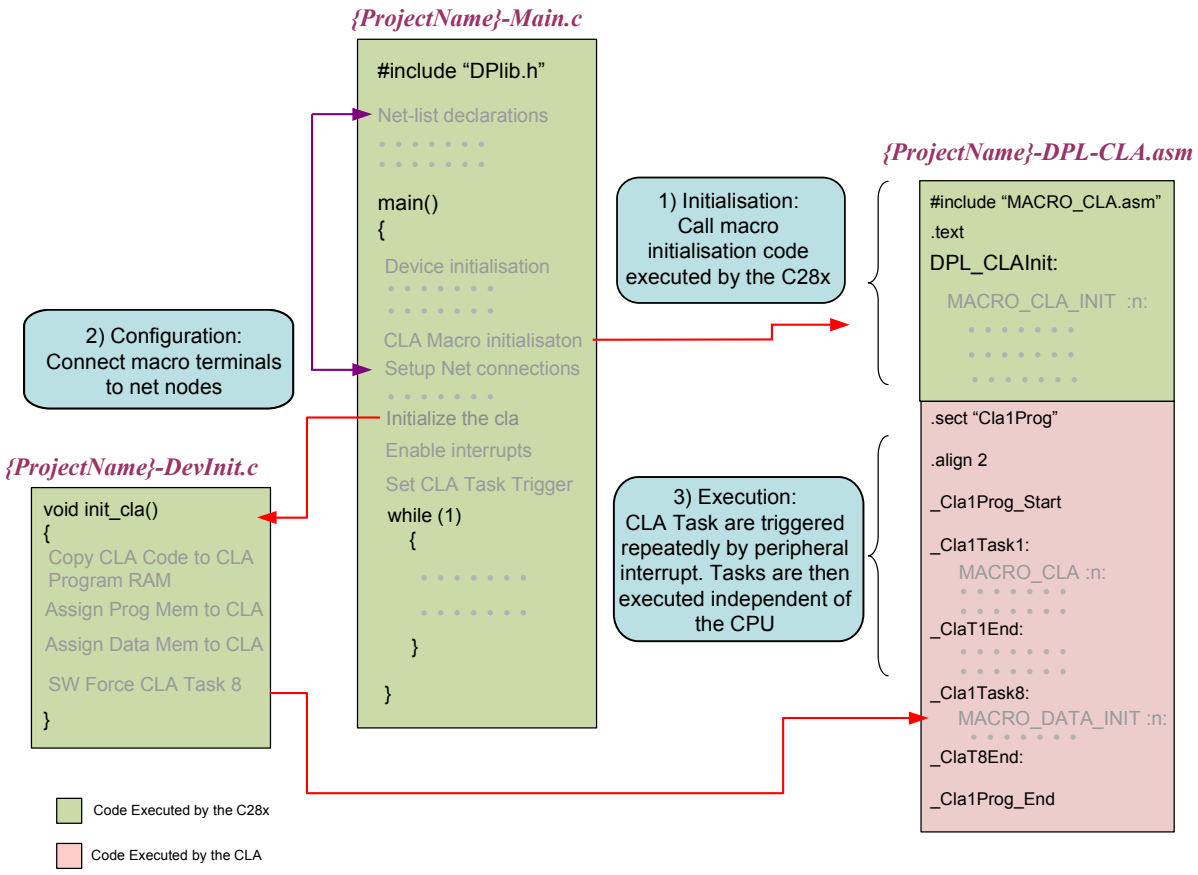


Figure 3 Relation between Main.c and DPL- CLA.asm file

The DP CLA library assembly code has a structure which has been designed to allow the user to freely specify the interconnection between blocks while maintaining a high degree of code efficiency. Before any of the library macros can be called they must be initialized using a short assembly routine located in the relevant `macro.asm` file for each module. The code which calls the macro initialization must reside in the same assembly file as where the library function is called.

The assembly code in "`DPCLA_Init`" is C callable, and its' prototype is in the library header file "`DPlib.h`" described above. To initialize the macros, edit the DPL_CLAInit section of the file "`{ProjectName}-DPL-CLA.asm`" to add initialization calls for each macro required in the application. The order of the calls is not important, providing one call is made for each macro-block required in the application. The respective macro assembly file must be included at the starting of the "`{ProjectName}-DPL_CLA.asm`" file. Note this section of the code is executed by the C28x while configuring the system for the CLA modules.

The internal layout and relationship between the CLA Task asm file and the various macro files is shown diagrammatically below. In this example, three DP library macros are being used. Each library module is contained in an assembly include file (`.asm` extension) which contains both initialization and macro code. The CLA Task file is also divided into two parts: one to initialize the macros which is executed by the C28x, the other is the real-time run time code which is to be run inside a CLA Task.

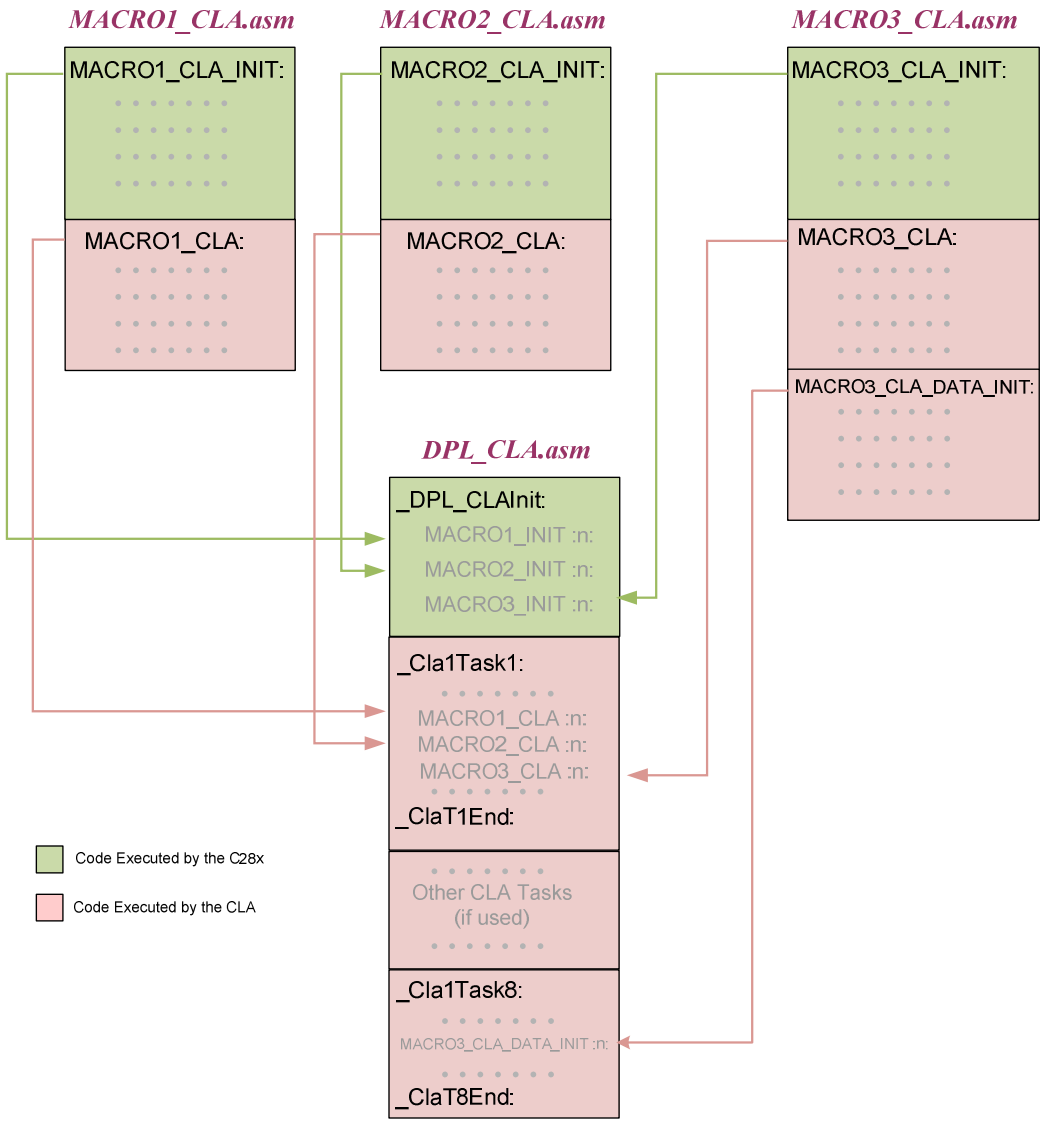


Figure 4 DPlib CLA asm & DPlib macro files

Note as one task runs at a time and is uninterruptable, there is no need for context save, which further reduces the sample to output delay.

3.2. Memory Requirements in CLA based systems

The CLA is a co-processor to the main C28x. As the internal memory available on the devices is fixed, the memory needs to be shared between the C28x and the CLA. As the memory available changes from device to device the following section explains the memory allocation on per device basis, explaining memory requirements of the different elements in the DP CLA library.

3.2.1 Memory Allocation on F28035

The F38032 has the following memory regions that can be used by the CLA:

Program Memory: 4K of RAM memory is available to be used by the CLA as program memory space. The user is responsible for copying the program code into this 4K RAM and assigning the memory to the CLA. By default this memory is assigned to the C28x and if the CLA is not being

used the C28x can use the memory as desired. Copying and allocation of this memory to the CLA is done in the CLA_Init() function which is defined in {ProjectName}-DevInit.c, a template of this file is provided with DP CLA library.

```
/* Copy the CLA program code from its load address to the CLA program
memory. */
MemCopy(&Cla1funcsLoadStart, &Cla1funcsLoadEnd, &Cla1funcsRunStart);

/*Once done, assign the program memory to the CLA.
Make sure there are at least two SYSCLKOUT cycles between assigning the
memory to the CLA and when an interrupt comes in. Call this function
even if Load and Run address is the same for RAM configurations!   */
Cla1Regs.MMEMCFG.bit.PROGE = 1; // Configure as CLA program memory
```

Data Memory: The CLA is a slave to the C28x core, thus when running a control algorithm the C28x may need to perform diagnostic functions / provide instructions for the CLA. The F28035 has three types of memories to serve this purpose:

1. **CpuToCla1MsgRAM:** 80 words of RAM area also called CpuToCla1MsgRAM allows CPU write access and CLA read access, CLA writes are ignored. This memory is configured this way by default and cannot be changed.

2. **Cla1ToCpuMsgRAM:** 80 words of RAM area also called Cla1ToCpuMsgRAM allows CPU read access and CLA write access, CPU writes are ignored. This memory is configured this way by default and cannot be changed.

3. **DataRAM:** Two 2K region of RAM, (RAM L1 and RAM L2) can be individually mapped to the CLA, to be used by the CLA or the C28x as Data RAM. Once assigned to the CLA, the CPU does not have access to these memory regions (aside from debugger accesses). If not mapped to the CLA, CLA reads to data RAM region will return zeros. By default this memory is assigned to the C28x. The mapping of the Data RAM to CLA happens in the CLA_Init() function which is defined in `{ProjectName}-DevInit.c` as shown below. The user can comment out the lines below if the application demands more data RAM than what can be allocated in the message RAMs.

```
// 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;
```

By default all the **net pointers are allocated to the CpuToCla1MsgRAM**, this is done because the C28x configures the system i.e. stores address location for the net variables. This configuration is the one that is shown in the template file and used in the module documentation below.

For **net variables** the user must sketch out the diagram of the control structure desired, this helps to visualize Read and Write requirements to different variables. The requirements can be summarized as following:

1. **CLA Write and CPU Read:** This region can be used to place variables that the CLA computes and CPU wants to monitor. An example of this is AC Line Average being calculated by the MATH_EMAVG_CLA block, the C28x wants to read this location to perform diagnosis on the system. This type of net variable must be kept in Cla1ToCpuMsgRAM.

2. **CLA Read and CPU Write:** This region can be used to place variables that are commands from the C28x to the CLA. An example of this is the C28x specifying the voltage reference for a close loop voltage stage being controlled by the CLA. This type of net variable must be kept in Cpu1ToClaMsgRAM.

3. **CLA Read and Write, CPU Read not necessary:** These memory requirements can exist on variables that are used by the CLA modules to perform calculations internally, and the CPU does not want/need to access these variables. These variables need to be in a CLA writable, hence Cla1ToCpuMsgRAM can be used for these types of variables. If CLA Data RAM is being used in the system, these variables can be allocated in the data RAM.

Example System to Illustrate Memory Assignment

The following example uses a digital power system where it is desired to switch from voltage loop to current loop depending on system conditions to explain memory assignments for DP CLA library components.

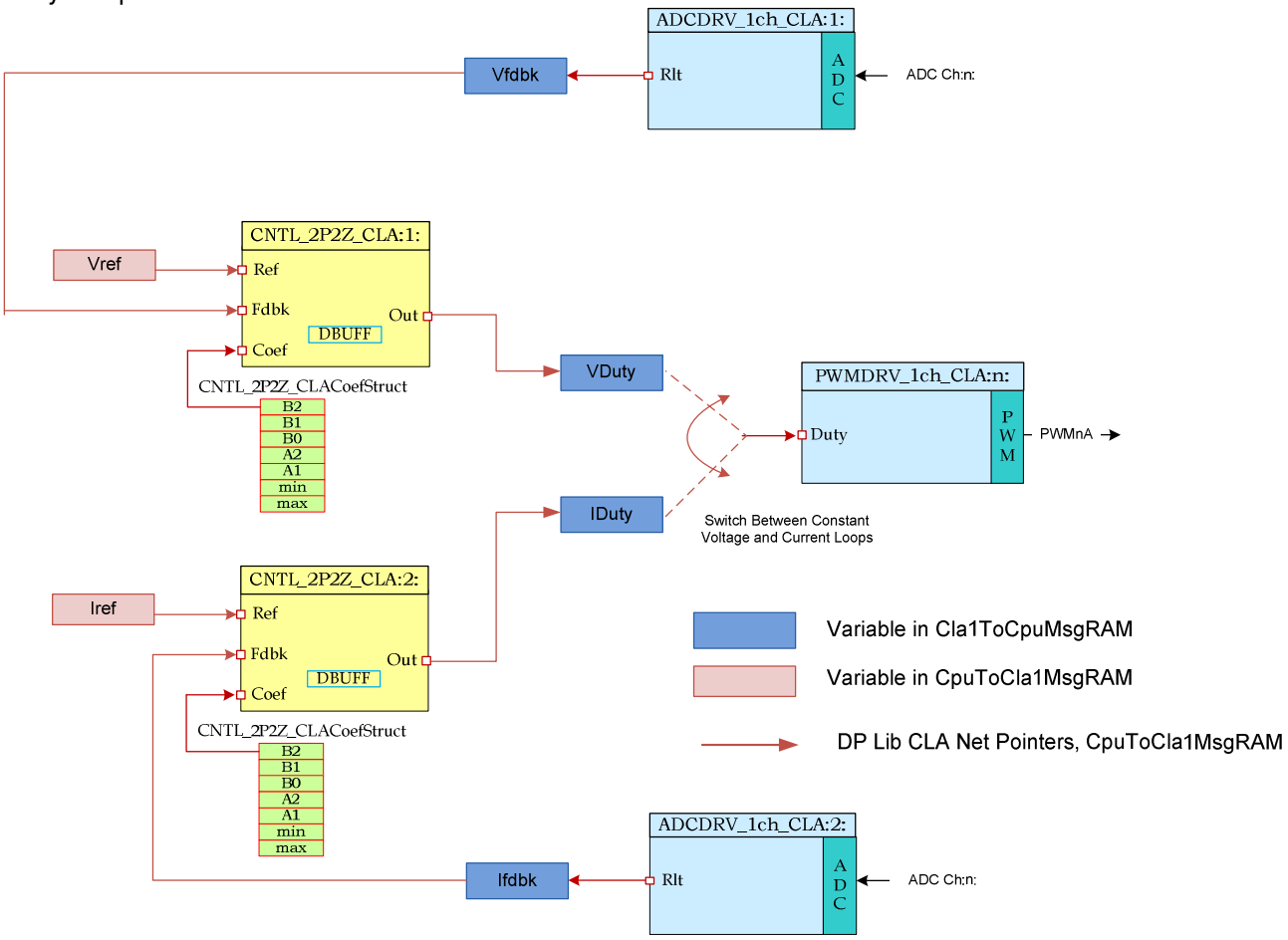


Figure: Example system with default memory configuration

The sketch above shows the following system behavior,

When Closed Loop Voltage Operation:

```
PWMDRV_1ch_CLA_Duty1 = &VDuty;
```

When Closed Loop Current Operation:
`PWMDRV_1ch_CLA_Duty1 = &IDuty`

The variables are color coded to reflect the memory requirement. In the system the CPU provides the reference for the voltage and the current loop and observes the feedback values that are computed, however the CPU may not want to observe the duty that is being outputted by the controller macro (this is just an assumption, to explain some features of the memory requirements).

I) Default Configuration

By default the following memory configuration can be used for the example system

Net Pointers, assignment is done in the `<DeviceName>-<FLASH/RAM>-{ProjectName}.CMD`. By default CpuToCLA1MsgRAM is used. Following is the code snippet to place net terminals for the above example.

Note that only one CMD directive is used for multiple instances of the same module (for ex `ADCDRV_1ch 1` and `ADCDRV_1ch 2`). Also note the `CNTL_2P2Z_DBUFF` which is stored in the `CLA_2P2Z_InternalData` is kept in the `Cla1toCpuMsgRAM` because the CLA uses it to store and read.

```
/*************       DPLIB Sections CLA      ************************/
/* ADCDRV_1ch_CLA section */
ADCDRV_1ch_CLA_Section      : > CPU_CLA_MSGRAM  PAGE = 1

/* CNTL_2P2Z_CLA controller sections */
CNTL_2P2Z_CLA_Section       : >  CPU_CLA_MSGRAM       PAGE = 1
CNTL_2P2Z_CLA_InternalData  : > CLA_CPU_MSGRAM        PAGE = 1
CNTL_2P2Z_CLA_Coef          : > CPU_CLA_MSGRAM  PAGE = 1

/* PWMDRV_1ch_CLA driver section */
PWMDRV_1ch_CLA_Section      : > CPU_CLA_MSGRAM  PAGE = 1
```

Net Variables, assign the net variable depending on the memory constraints to the MsgRAMs. This is done in the {ProjectName}-Main.c file.

```
#pragma DATA_SECTION(Vref,    "CpuToCla1MsgRAM");
#pragma DATA_SECTION(Iref,    "CpuToCla1MsgRAM");

#pragma DATA_SECTION(Vfdbk,   "Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(Ifdbk,   "Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(VDuty,   "Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(IDuty,   "Cla1ToCpuMsgRAM");
```

The C28x provides the Vref and the Iref, hence these are placed in the CpuToCla1MsgRAM. The `ADCDRV_1ch_CLA` and `CNTL_2P2Z_CLA` blocks compute the `Vfdbk, Ifdbk` and `VDuty ,IDuty` and hence are placed in the Cla1ToCpuMsgRAM.

II) Advanced Memory Configuration

For most power systems where the CLA is offloaded with just 1 or 2 control loops the default configuration using only the message RAM will suffice. However as more loops are offloaded to

the CLA it is necessary to use Data RAM for the CLA. The following section illustrates how the Data RAM can be used with the help of the example system while keeping all the configurability.

Note: The following section is only necessary once the system cannot be allocated by the default memory locations. The following code snippets assumes that CLA_Init() has been modified to assign the RAML2 to the CLA.

Net Pointers, The C28x has access to the dataRAM by default, and hence can do assignment to variables present in the dataRAM before CLA_Init() if called. User can recall from the Figure "*Relation between {ProjectName}-Main.c and{ProjectName}-DPL- CLA.asm file"* that CLA_Init() is called after the Net Terminals have been assigned. Hence the Net terminals can be placed in the data RAM.

However if the Net Terminals need to be changed at run time, which is the case in the example system above where while switching from constant voltage loop to constant current loop the PWMDRV_1ch net pointer would need to change from VDuty to IDuty, and hence should be kept in the Message RAM.

The memory assignment for the net pointers is specified in `<DeviceName>-<FLASH/RAM>-{ProjectName}.CMD` The following code snippet places net terminals for the above example system, while offloading some of the data to the dataRAM instead of the Message RAMs.

Note the `CNTL_2P2Z_DBUFF` is placed by the `CNTL_2P2Z_InternalData` and is placed in the dataRAM. The coefficients are kept in the CpuToCla1MsgRAM to allow changing of the coefficients depending on the system condition at run time by the CPU.

Also the `ADCDRV_1ch_CLA` net Terminals and `CNTL_2P2Z_CLA` net terminals are fixed at initialization and need not be changed and hence can be placed in the dataRAM. However as discussed the `PWMDRV_1ch_CLA` net terminal may change at run time, and thus is placed in the CpuToCla1MsgRAM.

```
/*************       DPLIB Sections CLA      ************************/
/* ADCDRV_1ch_CLA section */
ADCDRV_1ch_CLA_Section     : > RAML2            PAGE = 1

/* CNTL_2P2Z_CLA controller sections */
CNTL_2P2Z_CLA_Section      : >  RAML2           PAGE = 1
CNTL_2P2Z_CLA_InternalData      : >  RAML2            PAGE = 1
CNTL_2P2Z_CLA_Coef              : >  CPU_CLA_MSGRAM   PAGE = 1

/* PWMDRV_1ch_CLA driver section */
PWMDRV_1ch_CLA_Section      : > CPU_CLA_MSGRAM  PAGE = 1
```

Net Variables, assign the net variable depending on the memory constraints to the MsgRAMs or dataRAM. This is done in the `{ProjectName}-Main.c` file.

```
#pragma DATA_SECTION(Vref,    "CpuToCla1MsgRAM");
#pragma DATA_SECTION(Iref,    "CpuToCla1MsgRAM");

#pragma DATA_SECTION(Vfdbk,   "Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(Idfbk,   "Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(VDuty,   "RAML2");
#pragma DATA_SECTION(IDuty,   "RAML2");
```

The C28x provides the `Vref` and the `Iref`, hence these variables are placed in the CpuToCla1MsgRAM. The `ADCDRV_1ch_CLA` blocks compute the `Vfdbk` and `Ifdbk` and these values need to be monitored by the C28x. Thus these net variables are placed in the Cla1ToCpuMsgRAM. The `CNTL_2P2Z_CLA` blocks compute `VDuty` and `IDuty`, but are not required to be monitored by the CPU. Hence these are placed in the dataRAM.

The discussion above can be applied to other F2803x devices as well, however please refer to the device specific datasheet for details.

3.3. Steps to use the DP CLA library

The first task before using the DP CLA library should be to sketch out in diagram form the modules and block topology required. The aim should be to produce a diagram similar to that in *Figure 1*. This will indicate which macro-blocks are required and how they interface with one another. Also it should be identified where the different variables need to be placed i.e. whether the CLA needs to access them for read purpose only or if the CLA needs to write to them. Once this is known, the code can be configured as described below.

Note : Before using CLA the –cla_support must be enabled under in the Project Build Properties -> C200 Compiler

Step 1 ***Add the library header file***. The C header file "`DPlib.h`" contains prototypes and variable declarations used by the library. Add the following line at the top of your main C file:

```
#include "DPlib.h"
```

This file is located in the root directory of the power library, i.e.
`controlSUITE\libs\app_libs\digital_power\{device_name_VerNo}\include`
This path needs to be added to the include path in the build options for the project.

Step 2 ***Declare terminal pointers in C***. The "`{ProjectName}-Main.c`" file needs to be edited to add extern declarations to all the macro terminal pointers which will be needed in the application under the "`DPLIB Net Pointers`" section inside this file. In the example below net pointers to an instance of the 2P2Z control block are referenced.

```
// ---------------------------- DPLIB Net Pointers ----------------------
// Declare net pointers that are used to connect the DP Lib Macros  here
// CNTL_2P2Z_CLA #instance 1
extern volatile float   *CNTL_2P2Z_CLA_Ref1;
extern volatile float   *CNTL_2P2Z_CLA_Fdbk1;
extern volatile float   *CNTL_2P2Z_CLA_Out1;
extern volatile float   *CNTL_2P2Z_CLA_Coef1;
```

Step 3 ***Declare signal net nodes/variables in C and provide compiler directive for memory placement***. Edit the "`{ProjectName}-Main.c` " C file to define the net variables which will be needed in the application under the "`DPLIB Variables`" section . In the example below, three arbitrarily named variables are declared as global variables in C. The 2p2z module is configured to accept a Ref value from the CPU, Fdbk comes from another CLA module

```
// ---------------------------- DPLIB Variables ------------------------
// Declare the net variables being used by the DP Lib Macro here
// CPU Provides the reference
#pragma DATA_SECTION(Ref, "CpuToCla1MsgRAM");
// Fdbk comes from another CLA module, Output is written to by CLA
#pragma DATA_SECTION(Fdbk,"Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(Out, "Cla1ToCpuMsgRAM");

volatile float    Ref, Fdbk, Out;
```

Step 4 ***Call the initialization function from C***. Call the initialization function from the C framework using the syntax below.

```
    /* Digital Power (DP) CLA library initialization */
    DPL_CLAInit();          // initialize DP CLA library
```

Step 5 ***Assign macro block terminals to net nodes.*** This step connects macro blocks together via net nodes to form the desired control structure. The process is one of pointer assignment using the net node variables and terminal pointers declared in the previous two steps.

For example, to connect the ADC driver (instance 0) and 2P2Z control block (instance 1) to net node "Net2" as shown in Figure 1, the following assignment would be made:

```
    // feedback node connections
    ADCDRV_1ch_CLA_Rlt0 = &Net2;
    CNTL_2P2Z_CLA_Fdbk1 = &Net2;
```

Note that net pointer assignment can be dynamic: *i.e.* the user code can change the connection between modules at run-time if desired. This allows the user to construct flexible and complex control topologies which adapt intelligently to changing system conditions.

Step 6 ***Call CLA_Init() function to configure the CLA***, the function is provided as a template inside `{ProjectName}-DevInit.c` and can be modified to suit the needs of the application i.e. change memory allocation. The function is used to copy CLA program to RAM and assign the RAM space to be used as CLA Program Space. The function assigns data ram to the CLA if needed (this portion is commented out as default), define the vector addresses for the different CLA Tasks. In the end the function forces a software trigger of task 8 to carry out any initialization functions.

```
void CLA_Init()
{
   // This code assumes the CLA clock is already enabled in
   // the call to DevInit();
   //
   // EALLOW: is needed to write to EALLOW protected registers
   // EDIS: is needed to disable write to EALLOW protected registers
   //
   // The symbols used in this calculation are defined in the CLA
   // assembly code and in the {ProjectName}-CLAShared.h header file

EALLOW;
Cla1Regs.MVECT1 = (Uint16) (&Cla1Task1 -&Cla1Prog_Start)*sizeof(Uint32);
Cla1Regs.MVECT2 = (Uint16) (&Cla1Task2 -&Cla1Prog_Start)*sizeof(Uint32);
Cla1Regs.MVECT7 = (Uint16) (&Cla1Task7 -&Cla1Prog_Start)*sizeof(Uint32);
……..
Cla1Regs.MVECT8 = (Uint16) (&Cla1Task8 -&Cla1Prog_Start)*sizeof(Uint32);


// Map Peripheral interrupt to CLA Task
// Below Task 1 is configured to be triggered by ADC INT1

Cla1Regs.MPISRCSEL1.bit.PERINT1SEL = CLA_INT1_ADCINT1 ;


/* Copy the CLA program code from its load address to the CLA program
memory*/
MemCopy(&Cla1funcsLoadStart, &Cla1funcsLoadEnd, &Cla1funcsRunStart);


/* Once done, assign the program memory to the CLA*/
```

```
Cla1Regs.MMEMCFG.bit.PROGE = 1;

/* Map CLA Data Memory to the processor if required */
// 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;

// Enable the IACK instruction to start a task
// Enable the CLA interrupt 8 and software force it
asm("   RPT #3 || NOP");

Cla1Regs.MCTL.bit.IACKE = 1;
Cla1Regs.MIER.all =  M_INT8;

Cla1ForceTask8();

asm("   RPT #3 || NOP");

EDIS;
}
```

Step 7 ***Add the CLA Task Service Routine Assembly File***. A single assembly file containing the CLA assembly code, that comprises of the macro initialization functions and run time CLA assembly code, must exist in the project. A blank template of this file "ProjectName`-DPL-CLA.asm`" is included with the DP library for this purpose in the template directory. To use this file, rename the file as "`{ProjectName}-DPL-CLA.asm`" and add it to the project.

Step 8 ***Include the required macro header files***. Add assembly file includes to the top of the file "`{ProjectName}-DPL_CLA.asm`" as required. One include (`.asm`) file is required for each block type used in the project. For example, to use the 2P2Z controller block, add this line to the top of the "`{ProjectName}-DPL_CLA.asm`" file:

```
.include "CNTL_2P2Z_CLA.asm"
```

Step 9 ***Initialize required macro blocks***. Edit the function "`DPL_CLAInit`" in the above CLA file to add calls to the initialization code in each macro file. Note this function is executed by the C28x and not the CLA. Each call is invoked with a number to identify the unique instance of that macro block. For example, to create an instance of the 2P2Z control block with the identifier "`1`":

```
CNTL_2P2Z_CLA_INIT      1
```

Step 10 ***Edit the CLA assembly section and execute the macros in the required CLA Task as desired***. In the example below the first instance of a 2P2Z control macro would be executed whenever the CLA Task1 is triggered:

```
_Cla1Task1:
      CNTL_2P2Z_CLA     1
      MSTOP
      MNOP
      MNOP
      MNOP
_ClaT1End:
```

Step 11 ***Add the DP library linker to the command file***. The linker places DP library net terminals and variables in named sections as specified in the linker command file "`{DeviceName-RAM/FLASH-ProjectName}.CMD`". A sample CMD file is provided with the sections specified for the entire DPS library.

This sample linker file specifies where each memory section of DP library modules are placed. The placement for the CNTL_2P2Z_CLA module is shown below.

```
/* CNTL_2P2Z_CLA Controller sections */
CNTL_2P2Z_CLA_Section      : > CLA1_MSGRAMHIGH  PAGE = 1
CNTL_2P2Z_CLA_InternalData      : > CLA1_MSGRAMLOW    PAGE = 1
CNTL_2P2Z_CLA_Coef              : > CLA1_MSGRAMHIGH   PAGE = 1
```

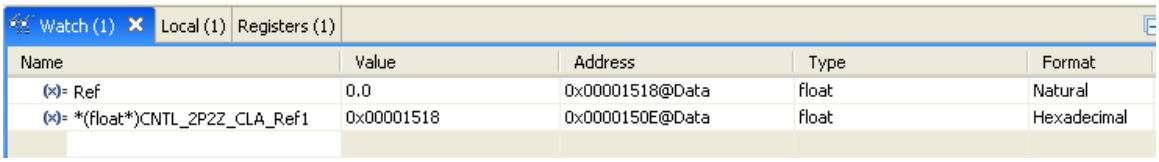
Here CLA1_MSGRAMHIGH is the same as CpuToCla1MsgRAM, and CLA1_MSGRAMLOW is the same as Cla1ToCpuMsgRAM.

Step 12 ***Mapping Peripheral Interrupt to the CLA Task***. Peripheral interrupts need to be mapped to the CLA, this is done once all the configuration of the CLA are complete in the `{ProjectName}-Main.c`. The following code snippet associates the ADCINT1 task to the CLA Task 1.

```
//Set Up CLA Task

// Task 1 has the option to be started by either EPWM1_INT or ADCINT1
// In this case we will allow ADCINT1 to start CLA Task 1
EALLOW;

Cla1Regs.MPISRCSEL1.bit.PERINT1SEL = CLA_INT1_ADCINT1 ;

// Configure the interrupt that would occur each control cycles
Cla1Regs.MIER.all = M_INT1;

asm("   RPT #3 || NOP");

EDIS;
```

3.4. Viewing DP CLA library variables in watch window

If is desired to see the DP CLA library macro variables, i.e. the net pointers can be added to the watch window by adding a qualifier of *(type*). Shown below is the value stored in the net pointer Ref, and the net variable the pointer points to. Note the address of the net variable is stored in the net pointer.



3.5. Read After Write Protection for DPLib Macros

The DPLib Macros write to the net variable at the end of the macro, if these net variables are immediately used in the code by a CLA assembly instruction three MNOP's would need to be placed between the two to prevent Read After Write conflict. For example:

```
ADCDRV_1ch_CLA 1        ; reads the ADCRESULT1 and stores it in Var1
MOV32    MR0,@_Var1
```

This would give rise to RAW as the MR0 would be loaded by the older value of Var1. Hence three MNOP's must be placed as follows

```
ADCDRV_1ch_CLA 1        ; reads the ADCRESULT1 and stores it in Var1
MNOP
MNOP
MNOP
MOV32    MR0,@_Var1
```

However if another DPLib macro is being called then these MNOP's are not required as at the beginning of the DPLib macro the net pointer is loaded in the Auxiliary registers, this takes three cycles and the net variable modified by the previous library block has been stored properly by then. Hence no need for MNOP's.

```
ADCDRV_1ch_CLA    1
CNTL_2P2Z_CLA     1     ; No MNOP's needed
```

Chapter 4. Module Summary

4.1. DP CLA Library Function Summary

The Digital Power CLA Library contains modules that enable the user to implement digital control for different power topologies. The following table lists the modules existing in the power library and a summary of cycle counts and code size.

Note: The memory sizes are given in 16-bit words and cycles are the system clock cycles taken to execute the Macro run file.

Module Name	Type	Description	HW Config File	Cycles	Init Code Size (W)	Run Code Size (W)	Data Size (W)	Multiple Instance Support
CNTL_2P2Z_CLA*	CNTL	Second Order Control Law	NA	31	7	48	18	Yes
CNTL_3P3Z_CLA*	CNTL	Third Order Control Law	NA	38	7	70	18	Yes
ADCDRV_1ch_CLA	HW	Single Channel ADC Driver	Yes	5	5	5	2	Yes
ADCDRV_4ch_CLA	HW	Four Channel ADC Driver	Yes	16	8	40	8	No
PWMDRV_1ch_CLA	HW	Single Channel PWM Driver	Yes	11	5	20	2	Yes
PWMDRV_1chHiRes_CLA	HW	Single Channel PWM Driver with Hi Res capability	Yes	12	5	20	2	Yes
PWMDRV_PFC2PhiL_CLA	HW	PWM driver for Two Phase Interleaved PFC stage	Yes	16	6	28	4	Yes
PWMDRV_PSFB_CLA	HW	PWM driver for Phase Shifted Full Bridge Power stage	Yes	23	7	38	6	Yes
PWMDRV_ComplPairDB_CLA	HW	PWM driver for complimentary pair PWMs	Yes	12	5	15	4	Yes
PWMDRV_DualUpDwnCnt_CLA	HW	PWM driver with independent duty control on ch A and B	Yes	14	6	24	4	Yes
PFC_ICMD_CLA	APPL	Power Factor Correction Current Command Block	NA	13	7	24	10	Yes
PFC_INVSQR_CLA	APPL	Power Factor Correction Inverse Square Block	NA	12	6	22	8	Yes
MATH_EMAVG_CLA	MATH	Exponential moving average	NA	10	6	18	6	Yes

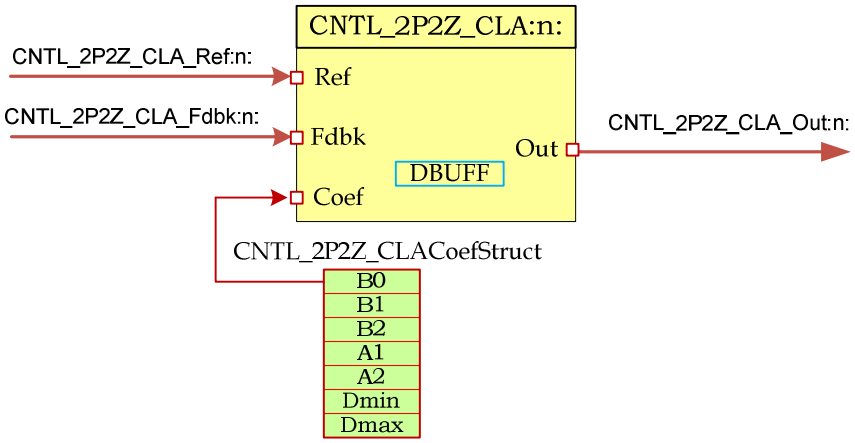
*CNTL_2P2Z_CLA and CNTL_3P3Z_CLA have an additional 14 words long Init code that is executed by the CLA

Chapter 5. DP CLA Module Descriptions

5.1. Controllers

CNTL_2P2Z_CLA	*Two Pole Two Zero Controller using CLA*

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 using the CLA.



Macro File: `CNTL_2P2Z_CLA.asm`

Module Description: 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…

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

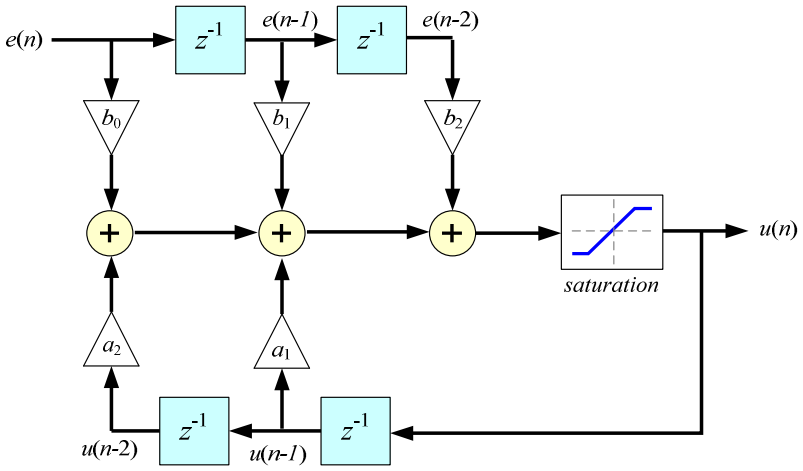
This may be expressed in difference equation form as:

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

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_CLA_DBUFF` as shown below.

CNTL_2P2Z_CLA_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 shown:

CNTL_2P2Z_CLACoefStruct

$float(b_2)$
$float(b_1)$
$float(b_0)$
$float(a_2)$
$float(a_1)$
$float(sat_{max})$
$float(sat_{min})$

Where sat_{max} and sat_{min} are the upper and lower control effort bounds respectively.

Controller coefficients must be initialized before the controller is used. A structure `CNTL_2P2Z_CLACoefStruct` 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. The structure is defined in the library header file DPlib.h to allow easy access to the elements from C.

Usage: This section explains how to use CNTL_2P2Z this module.

Step 1 Add library header file in the file {ProjectName}-Main.c

```
#include "DPLib.h"
```

Step 2 Declare the terminal pointers in C in the file {ProjectName}-Main.c

```
// ---------------------------- DPLIB Net Pointers ---------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
// CONTROL_2P2Z_CLA - instance #1
extern volatile float   *CNTL_2P2Z_CLA_Ref1;    // instance #1
extern volatile float   *CNTL_2P2Z_CLA_Out1;    // instance #1
extern volatile float   *CNTL_2P2Z_CLA_Fdbk1;   // instance #1
extern volatile float   *CNTL_2P2Z_CLA_Coef1;   // instance #1
```

Step 3 Declare signal net nodes/ variables in C in the file {ProjectName}-Main.c, also specify the appropriate memory location variable need using the **#pragma** pre-processor directive. This is determined by whether the CLA reads from the variable or writes to the variable.

For the example below, Ref is placed in the CpuToCla1MsgRAM so that the C28x CPU can specify/change the reference value to the controller. Fdbk is received from another CLA module and Out is written to by the 2P2Z CLA module and hence placed in a CLA writable space which in this case Cla1ToCpuMsgRAM is used.

Note signal net nodes/variable names are not dependent on macro names and/or macro terminal pointer names, these can change from system to system.

```
// ---------------------------- DPLIB Variables ------------------------

// Declare the net variables being used by the DP Lib Macro here

#pragma DATA_SECTION(Out, "Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(Fdbk,"Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(Ref, "CpuToCla1MsgRAM");
volatile float  Out,Fdbk,Ref;

#pragma DATA_SECTION(CNTL_2P2Z_CLACoefStruct1, "CNTL_2P2Z_CLA_Coef");
struct CNTL_2P2Z_CLACoefStruct CNTL_2P2Z_CLACoefStruct1;
```

Step 4 "Call" the DPL_CLAInit() to initialize the macros and **"connect"** the module terminals to the signal nets in "C" in {ProjectName}-Main.c

```
//----------Connect the macros to build a system-------------------
// Digital Power (DP) CLA library initialisation
DPL_CLAInit();

// Connect the CNTL_2P2Z block to the variables
CNTL_2P2Z_CLA_Fdbk1 = &Fdbk;
CNTL_2P2Z_CLA_Out1  = &Out;
CNTL_2P2Z_CLA_Ref1  = &Ref;
CNTL_2P2Z_CLA_Coef1 = &CNTL_2P2Z_CLACoefStruct1.b2;

// Initialize the Controller Coefficients
// Note the CLA Coefficients would reside in the CPUToCLAMsg RAM
CNTL_2P2Z_CLACoefStruct1.b2   =0.05;
CNTL_2P2Z_CLACoefStruct1.b1   =-0.20;
CNTL_2P2Z_CLACoefStruct1.b0   =0.20;
CNTL_2P2Z_CLACoefStruct1.a2   =0.0;
CNTL_2P2Z_CLACoefStruct1.a1   =1.0;
CNTL_2P2Z_CLACoefStruct1.max  =0.70;
CNTL_2P2Z_CLACoefStruct1.min  =0.0;

//Initialize the net Variables residing in the CPU to CLA
Ref=(float)0.0;
```

Step 5 **"Call"** the CLA_Init() function. This function is provided as a template inside `{ProjectName}-Main.c` and can be modified to suit the needs of the application i.e. change memory allocation, add tasks etc. See **Section 3.3** "Steps to use DP CLA Library" for details on how to modify this function to suit your application needs. Note Task 8 is reserved to do any initializations that need to be done to use the macro block when using DP Lib CLA. This task is software forced one time at the end of this function. This task is used to initialize the net variables (discussed in **Step 10**).

```
//----------------------Initialize the CLA-------------------------
CLA_Init();
```

Step 6 **Add** the CLA Task Service Routine and Macro Initialization assembly file "`{ProjectName}-DPL-CLA.asm`" to the project

Step 7 **Include** the Macro's assembly file in the `{ProjectName}-DPL-CLA.asm`

```
;Include files for the Power Library Macro's being used by the system
.include "CNTL_2P2Z_CLA.asm"
```

Step 8 **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-callable DPL_CLAInit() function which is defined in `{ProjectName}-DPL-ISR.asm`

```
;Macro Specific Initialization Functions
CNTL_2P2Z_CLA_INIT 1    ; CNTL_2P2Z Initialization
```

Step 9 **Call the run time macro** in assembly inside the CLA Task which is defined in `{ProjectName}-CLA.asm`. This task is the task which is repeatedly called in a loop for the control algorithm, this task is set in the `CLA_Init()` function.

```
;"Call" the Run macro
_Cla1Task1:
      CNTL_2P2Z_CLA 1    ; Run CNTL_2P2Z_CLA
      MSTOP
      MNOP
      MNOP
_ClaT1End1:
```

Step 10 **Initialize variables** which are in the CLA writable memory location in CLA Task 8,

Run the DBUFF INIT Task for the 2p2Z macro. Note when using the DP CLA Lib Framework Task 8 is forced by the CLA_Init() function.

Note variable names in the following step are not associated with the module and can change depending on Step 2, and or the system in consideration.

```
; run the macro to initialize the internal data of the macro residing in
; the CLA write memory space
CNTL_2P2Z_DBUFF_CLA_INIT 1
;(Initialize variables in CLA writable memory space )
.ref _Fdbk
.ref _Out
MMOVF32 MR0, #0.0L
MF32TOUI32   MR0, MR0
MMOV32 @_Fdbk,MR0
MMOV32 @_Out,MR0
```

Step 11 **Include the memory sections** in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD` for the macro's net terminals and internal data and allocate them to the desired location.
See **section 3.2** "Memory Requirements in CLA based systems for F28035" for more details.

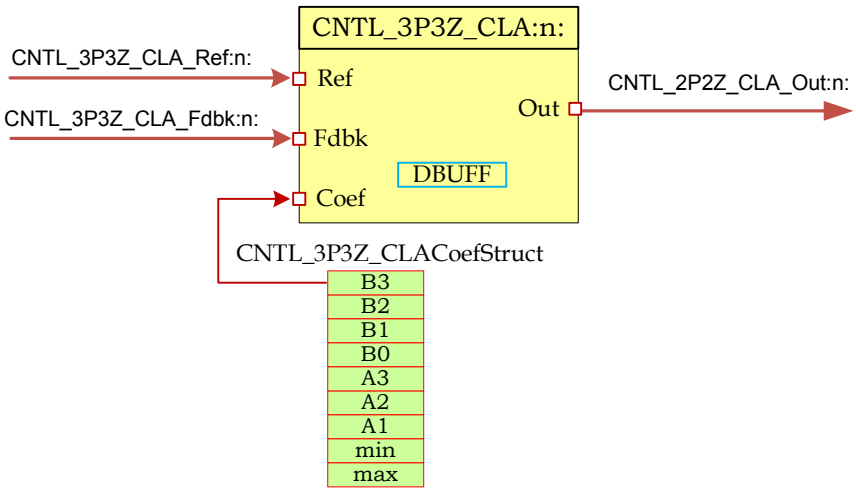
```
/* CNTL_2P2Z_CLA controller sections */
CNTL_2P2Z_CLA_Section      : > CLA1_MSGRAMHIGH  PAGE = 1
CNTL_2P2Z_CLA_InternalData      : > CLA1_MSGRAMLOW    PAGE = 1
CNTL_2P2Z_CLA_Coef              : > CLA1_MSGRAMHIGH   PAGE = 1
```

Module Net Definition:

Net Name (:n: is the instance number)	**Description (memory Restriction)**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
CNTL_2P2Z_CLA_Ref:n:	Input Pointer (CLA Read)	Pointer to 32 bit fixed point input data location storing the Reference value for the controller.	Float: [0, 1)
CNTL_2P2Z_CLA_Fdbk:n:	Input Pointer (CLA Read)	Pointer to 32 bit fixed point input data location storing the Feedback value for the controller.	Float: [0, 1)
CNTL_2P2Z_CLA_Coef:n:	Input Pointer (CLA Read)	Pointer to the location where coefficient structure is stored.	Float
CNTL_2P2Z_CLA_Out:n:	Output Pointer (CLA Read)	Pointer to 32 bit fixed point output location where the reference for the current loop is stored	Float:[0,1)
CNTL_2P2Z_CLA_DBUFF:n:	Internal Data (CLA Write)	Data Variable storing the scaling factor	Float

CNTL_3P3Z_CLA	*Three Pole Three Zero Controller using CLA*

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



Macro File: `CNTL_3P3Z_CLA.asm`

Module Description: The 3-pole 3-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 three elements.

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

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

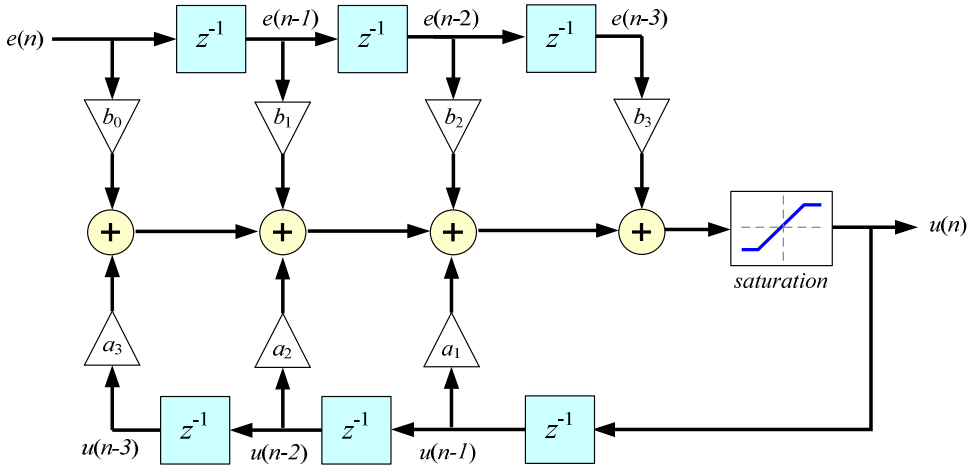
This may be expressed in difference equation form as:

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

Where…

$u(n)$ = present controller output (after saturation)
$u(n\text{-}1)$ = controller output on previous cycle
$u(n\text{-}2)$ = 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 3P3Z control law may be represented graphically as shown below.



Input and output data are located in internal RAM with address designated by `CNTL_3P3Z_CLA_DBUFF` as shown below.

CNTL_3P3Z_DBUFF

0	$u(n\text{-}1)$
2	$u(n\text{-}2)$
4	$u(n\text{-}3)$
6	$e(n)$
8	$e(n\text{-}1)$
10	$e(n\text{-}2)$
12	$e(n\text{-}3)$

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

CNTL_3P3Z_CLACoefStruct

$float(b_3)$
$float(b_2)$
$float(b_1)$
$float(b_0)$
$float(a_3)$
$float(a_2)$
$float(a_1)$
$float(sat_{max})$
$float(sat_{min})$

Where sat_{max} and sat_{min} are the upper and lower control effort bounds respectively.

Controller coefficients must be initialized before the controller is used. A structure `CNTL_3P3Z_CLACoefStruct` 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. The structure is defined in the library header file `DPlib.h` to allow easy access to the elements from C.

Usage: This section explains how to use CNTL_3P3Z this module.

`Step 1` **Add library header file** in the file `{ProjectName}-Main.c`

```
#include "DPLib.h"
```

`Step 2` **Declare the terminal pointers** in C in the file `{ProjectName}-Main.c`

```
// ---------------------------- DPLIB Net Pointers ----------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
// CONTROL_3P3Z_CLA - instance #1
extern volatile float   *CNTL_3P3Z_CLA_Ref1;
extern volatile float   *CNTL_3P3Z_CLA_Out1;
extern volatile float   *CNTL_3P3Z_CLA_Fdbk1;
extern volatile float   *CNTL_3P3Z_CLA_Coef1;
```

`Step 3` **Declare signal net nodes/ variables in C** in the file `{ProjectName}-Main.c`, `also` specify the appropriate memory location variable need using the **#pragma** preprocessor directive. This is determined by whether the CLA reads from the variable or writes to the variable.

For the example below, Ref is placed in the CpuToCla1MsgRAM so that C28x can specify/change the reference value to the controller. Fdbk is received from another CLA module and Out is written to by the 3P3Z CLA module and hence placed in a CLA writable space which in this case Cla1ToCpuMsgRAM is used.

Note signal net nodes/variable names are not dependent on macro names and/or macro terminal pointer names, these can change from system to system.

```
// ---------------------------- DPLIB Variables ------------------------

// Declare the net variables being used by the DP Lib Macro here

#pragma DATA_SECTION(Out, "Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(Fdbk,"Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(Ref, "CpuToCla1MsgRAM");
volatile float  Out,Fdbk,Ref;

#pragma DATA_SECTION(CNTL_3P3Z_CLACoefStruct1, "CNTL_3P3Z_CLA_Coef");
struct CNTL_3P3Z_CLACoefStruct CNTL_3P3Z_CLACoefStruct1;
```

Step 4 **"Call"** the `DPL_CLAInit()` to initialize the macros and **"connect"** the module terminals to the signal nets in "C" in `{ProjectName}-Main.c`

```
//----------Connect the macros to build a system-------------------
// Digital Power (DP) CLA library initialisation
DPL_CLAInit();

// Connect the CNTL_3P3Z block to the variables
CNTL_3P3Z_CLA_Fdbk1 = &Fdbk;
CNTL_3P3Z_CLA_Out1  = &Out;
CNTL_3P3Z_CLA_Ref1  = &Ref;
CNTL_3P3Z_CLA_Coef1 = &CNTL_3P3Z_CLACoefStruct1.b2;

// Initialize the Controller Coefficients
// Note the CLA Coefficients would reside in the CPUToCLAMsg RAM
CNTL_3P3Z_CLACoefStruct1.b2   =0.05;
CNTL_3P3Z_CLACoefStruct1.b1   =-0.20;
CNTL_3P3Z_CLACoefStruct1.b0   =0.20;
CNTL_3P3Z_CLACoefStruct1.a2   =0.0;
CNTL_3P3Z_CLACoefStruct1.a1   =1.0;
CNTL_3P3Z_CLACoefStruct1.max  =0.70;
CNTL_3P3Z_CLACoefStruct1.min  =0.0;

//Initialize the net Variables residing in the CPU to CLA
Ref=(float)0.0;
```

Step 5 **"Call"** the CLA_Init() function, this function is provided as a template inside `{ProjectName}-Main.c` and can be modified to suit the needs of the application i.e. change memory allocation, add tasks etc. See **Section 3.3** "Steps to use DP CLA Library" for details on how to modify this function to suit your application need. Note Task 8 is reserved to do any initializations that need to be done to use the macro block when using DP Lib CLA. This task is software forced one time at the end of this function. This task is used to initialize the net variables (discussed in **Step 10**).

```
//----------------------Initialize the CLA-------------------------
CLA_Init();
```

Step 6 **Add** the CLA Task Service Routine and Macro Initialization assembly file "`{ProjectName}-DPL-CLA.asm`" to the project

Step 7 **Include** the Macro's assembly file in the `{ProjectName}-DPL-CLA.asm`

```
;Include files for the Power Library Macro's being used by the system
.include "CNTL_3P3Z_CLA.asm"
```

Step 8 **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-Callable DPL_CLAInit() function which is defined in `{ProjectName}-DPL-ISR.asm`

```
;Macro Specific Initialization Functions
CNTL_3P3Z_CLA_INIT 1    ; CNTL_3P3Z Initialization
```

Step 9 Call the run time macro in assembly inside the CLA Task which is defined in `{ProjectName}-CLA.asm`. This task is the task which is repeatedly called in a loop for the control algorithm, this task is set in the `CLA_Init()` function.

```
;"Call" the Run macro
_Cla1Task1:
      CNTL_3P3Z_CLA 1   ; Run CNTL_3P3Z_CLA
      MSTOP
      MNOP
      MNOP
_Cla1T1End1:
```

Step 10 Initialize variables which are in the CLA writable memory location in CLA Task 8,

Run the DBUFF INIT Task for the 3P3Z macro. Note when using the DP CLA Lib Framework Task 8 is forced by the CLA_Init() function.

Note variable names in the following step are not associated with the module and can change depending on Step 2, and or the system in consideration.

```
; run the macro to initialize the internal data of the macro residing in
; the CLA write memory space
CNTL_3P3Z_DBUFF_CLA_INIT 1
;(Initialize variables in CLA writable memory space )
.ref _Fdbk
.ref _Out
MMOVF32 MR0, #0.0L
MMOV32 @_Fdbk,MR0
MMOV32 @_Out,MR0
```

Step 11 Include the memory sections in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD` for the macro's net terminals and internal data and allocate them to the desired location.
See **section 3.2** "Memory Requirements in CLA based systems for F28035" for more details.

```
/* CNTL_3P3Z_CLA controller sections */
CNTL_3P3Z_CLA_Section           : > CLA1_MSGRAMHIGH   PAGE = 1
CNTL_3P3Z_CLA_InternalData      : > CLA1_MSGRAMLOW    PAGE = 1
CNTL_3P3Z_CLA_Coef              : > CLA1_MSGRAMHIGH   PAGE = 1
```

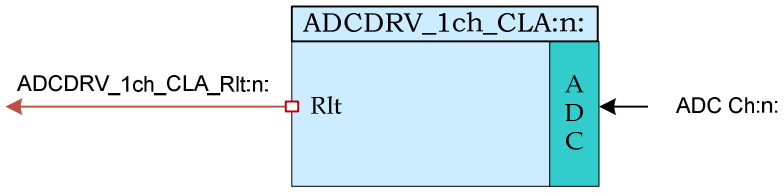
Module Net Definition:

Net Name (:n: is the instance number)	**Description (memory Restriction)**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
CNTL_3P3Z_CLA_Ref:n:	Input Pointer (CLA Read)	Pointer to 32 bit fixed point input data location storing the Reference value for the controller.	Float: [0, 1)
CNTL_3P3Z_CLA_Fdbk:n:	Input Pointer (CLA Read)	Pointer to 32 bit fixed point input data location storing the Feedback value for the controller.	Float: [0, 1)
CNTL_3P3Z_CLA_Coef:n:	Input Pointer (CLA Read)	Pointer to the location where coefficient structure is stored.	See Module Description
CNTL_3P3Z_CLA_Out:n:	Output Pointer (CLA Read)	Pointer to 32 bit fixed point output location where the reference for the current loop is stored	Float:[0,1)
CNTL_3P3Z_CLA_DBUFF:n:	Internal Data (CLA Write)	Data Variable storing the scaling factor	See Module Description

5.2. Peripheral Drivers

ADC_DRV_1ch_CLA	*ADC Driver Single Channel on CLA*

Description: This assembly macro reads a result from the internal ADC module Result Register:n: and delivers it in float format to the output terminal, where :n: is the instance number of the macro. The output is normalized to 0-1.0 such that the minimum input voltage at the ADC pin will generate nominally 0.0 at the driver output, and a maximum full scale input voltage generates +1.0. The result is then stored in the memory location pointed to by the net terminal pointer.



Macro File: `ADCDRV_1ch_CLA.asm`

Peripheral Initialization File: `ADC_SOC_Cnf.c`

Description: The ADCDRV macro reads one pre-defined result register (determined by the instance number of the macro i.e. instance 0 reads AdcResult.ADCRESULT0 and instance 5 reads AdcResult.ADCRESULT5). The module then scales this to normalized float format and writes the result in single precision float format to the output net terminal.

This macro is used in conjunction with the Peripheral configuration file ADC_SOC_Cnf.c The file defines the function

```
void ADC_SOC_CNF(int ChSel[], int Trigsel[], int ACQPS[],
   int IntChSel, int mode)
```

Where

ChSel[] stores which ADC pin is used for conversion when a Start of Conversion is received for the respective channel

TrigSel[] stores what trigger input starts the conversion of the respective channel

ACQPS[] stores the acquisition window size used for the respective channel

IntChSel is the Channel Number that triggers ADCINT 1.

Mode determines what mode the ADC is configured in

Mode =0 Start/Stop mode, configures ADC conversions to be started by the appropriate channel Trigger, an

ADC interrupt is raised whenever conversion is complete for the IntChSel channel. The ADC Interrupt Flag needs to be cleared for the interrupt to be retriggered. This is the mode used for most C28x based projects.

Mode =1 The ADC is configured in continuous conversion mode. This mode is kept to keep compatibility with previous generation ADCs.

Mode =2 CLA Mode, configures ADC conversions to be started by the appropriate channel trigger. an ADC interrupt is raised whenever conversion is complete and the ADC Interrupt Flag is auto cleared. This mode is used for all of the CLA based projects.

Note the function configures the complete ADC module in a single function call. Hence this function is called only once even for multiple ADCDRV modules.

Usage:

`Step 1` **Add library header file** in the file `{ProjectName}-Main.c`

```
#include "DPLib.h"
```

`Step 2` **Declare the terminal pointers** in C in the file `{ProjectName}-Main.c`

```
// ---------------------------- DPLIB Net Pointers --------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//ADCDRV_1ch_CLA - instance #0
extern volatile float *ADCDRV_1ch_CLA_Rlt0;
```

`Step 3` **Declare signal net nodes/ variables in C** in the file {ProjectName}-Main.c, also specify the appropriate memory location variable need using the **#pragma** preprocessor directive. This is determined by whether the CLA reads from the variable or writes to the variable.

Note signal net nodes/variable names are not dependent on macro names and/or macro terminal pointer names, these can change from system to system.

```
// ---------------------------- DPLIB Variables ------------------------
// Declare the net variables being used by the DP Lib Macro here
#pragma DATA_SECTION(AdcResult0,"Cla1ToCpuMsgRAM");
volatile float Out;
```

`Step 4` **Call the Peripheral configuration function** `ADC_SOC_CNF(int ChSel[],int TrigSel[],int ACQPS[], int IntChSel, int mode)` in `{ProjectName}-Main.c`, this function is defined in `ADC_SOC_CNF.c`. This file must be included manually into the project.

```
/* The following code configures ADC channel 0 to convert ADCINB5, and
ADC channel 1 to convert the ADCINA3. The ADC is configured in start
stop mode and channel 0 is configured to raise ADCINT 1. The mode is
set to auto clear this interrupt to enable retrigger of the interrupt.
ADC Channel 0 is configured to use PWM1 SOCA and channel 1 is
configured to use PWM 5 SOCB as trigger. The following code snippet
assumes that the PWM peripherals have been configured appropriately to
generate a SOCA and SOCB */

// Specify ADC Channel - pin Selection for Configuring the ADC
ChSel[0] = 13;    // ADC B5
ChSel[1] = 3;           // ADC A3

// Specify the Conversion Trigger for each channel
TrigSel[0]= ADCTRIG_EPWM1_SOCA;
TrigSel[1]= ADCTRIG_EPWM5_SOCB;

// Call the ADC Configuration Function, in CLA mode
ADC_SOC_CNF(ChSel,TrigSel,ACQPS,2,0);
```

Step 5 **"Call"** the `DPL_CLAInit()` to initialize the macros and **"Connect"** the module terminals to the Signal Nets in "C" in `{ProjectName}-Main.c`

```
//----------Connect the macros to build a system-------------------

// Digital Power (DP) library initialisation
DPL_CLAInit();
// ADCDRV_1ch block connections
ADCDRV_1ch_CLA_Rlt0=&Out;
```

Step 6 **Call the CLA_Init()** function, this function is provided as a template inside the system software Main.c file and can be modified to suit the needs of the application i.e. change memory allocation, add tasks etc. However Task 8 is reserved to do any initializations that need to be done to use the macro block. This task is software forced one time at the end of this function. How this task is used to initialize the net variables is discussed in **Step 10**.

```
//----------------------Initialize the CLA-------------------------

CLA_Init();
```

Step 7 **Add** the CLA Task Service Routine and Macro Initialization assembly file "`{ProjectName}-DPL-CLA.asm`" to the project

Step 8 **Include** the Macro's assembly file in the `{ProjectName}-DPL-CLA.asm`

```
;Include files for the Power Library Macro's being used by the system
.include "ADCDRV_1ch_CLA.asm"
```

Step 9 **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-Callable DPL_CLAInit() function which is defined in `{ProjectName}-DPL-ISR.asm`

```
;Macro Specific Initialization Functions
ADCDRV_1ch_CLA_INIT 0   ; ADCDRV_1ch_CLA Initialization
```

Step 10 **Call the run time macro** in assembly inside the CLA Task which is defined in `{ProjectName}-CLA.asm`. This task is the task which is repeatedly called in a loop for the control algorithm.

```
;"Call" the Run macro
;(Note Result Register 0 i.e. Channel 0 result is used for instance#0)
ADCDRV_1ch_CLA 0  ; Run ADCDRV_1ch
```

Step 11 **Initialize variable** which is in the CLA writable memory location in CLA Task 8, this task is forced by the CLA_Init() function.
Note variable names in the following step are not associated with the module and can change depending on Step 2, and or the system in consideration

```
;(Initialize variables in CLA writable memory space )
.ref _Out
MMOVF32 MR0, #0.0L
MMOV16 @_Out,MR0
```

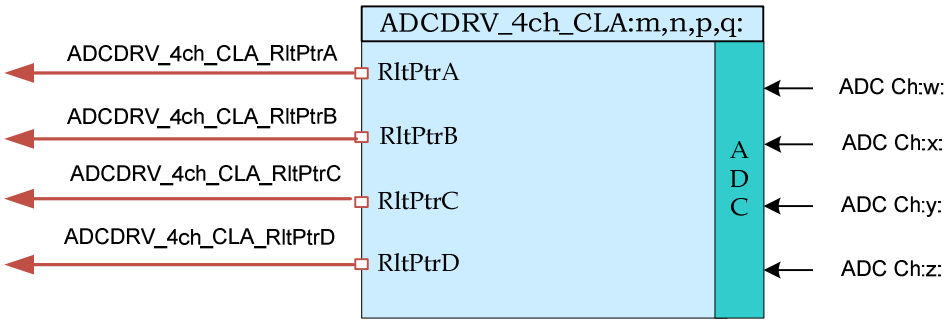
Step 12 **Include the memory sections** the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD` for the macro's net terminals and internal data and allocate them to the desired location.
See **section 3.2** "Memory Requirements in CLA based systems for F28035" for more details.

```
/* ADCDRV_1ch_CLA section */
ADCDRV_1ch_CLA_Section     : > CLA1_MSGRAMHIGH  PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
ADCDRV_1ch_CLA_Rlt:n:	Output Pointer	Pointer to 32 bit fixed point data location storing the result of the module.	Float: [0, 1)

Description: This assembly macro reads four results from the internal ADC module result registers m,n,p,q and delivers them in normalized float format to the output terminals. The output is normalized to 0-1.0 such that the minimum input voltage will generate nominally 0.0 at the driver output, and a maximum full scale input voltage read +1.0. The result are then stored in the memory location pointed to by the net terminal pointers.



Macro File: `ADCDRV_4ch_CLA.asm`

Peripheral Initialization File: `ADC_SOC_Cnf.c`

Description: The ADCDRV macro reads ADC result register which are determined when the module is called at run time by the arguments that are parsed i.e. m,n,p & q. The module then scales this to normalized float format and writes the result in single precision float format to the output net terminals.

This macro is used in conjunction with the peripheral configuration file ADC_SOC_Cnf.c The file defines the function

```
void ADC_SOC_CNF(int ChSel[], int Trigsel[], int ACQPS[],
   int IntChSel, int mode)
```

where

ChSel[] Array that stores which ADC pin is used for conversion when a start of conversion is received for the respective channel

TrigSel[] stores what trigger Input starts the conversion of the respective channel

ACQPS[] stores the acquisition window size used for the respective channel

IntChSel is the channel number that triggers interrupt ADCINT 1. If the ADC interrupt is not being used enter a value of 0x10.

Mode determines what mode the ADC is configured in

Mode =0 Start/Stop mode, configures ADC conversions to be started by the appropriate channel trigger, an ADC interrupt is raised whenever conversion is

complete for the IntChSel channel. The ADC interrupt flag needs to be cleared for the interrupt to be retriggered. This is the mode used for most C28x based projects.

Mode =1 The ADC is configured in continuous conversion mode. This mode maintains compatibility with previous generation ADCs.

Mode =2 CLA Mode, configures ADC conversions to be started by the appropriate channel trigger. an ADC interrupt is raised whenever conversion is complete and the ADC Interrupt Flag is auto cleared. This mode is used for all of the CLA based projects.

Note the function configures the complete ADC module in a single function call. Hence this function is called only once.

This function is responsible for associating the ADC peripheral pins to result registers. The macro run time call is only responsible for reading these registers.

Multiple instantiation of this macro is not supported.

Usage:

`Step 1` **Add library header file** in the file `{ProjectName}-Main.c`

```
#include "DPLib.h"
```

`Step 2` **Declare the terminal pointers** in C in the file `{ProjectName}-Main.c`

```
// ---------------------------- DPLIB Net Pointers --------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//ADCDRV_4ch_CLA
extern volatile float *ADCDRV_4ch_CLA_RltA;
extern volatile float *ADCDRV_4ch_CLA_RltB;
extern volatile float *ADCDRV_4ch_CLA_RltC;
extern volatile float *ADCDRV_4ch_CLA_RltD;
```

`Step 3` **Declare signal net nodes/ variables in C** in the file {ProjectName}-Main.c, also specify the appropriate memory location variable need using the **#pragma** preprocessor directive. This is determined by whether the CLA reads from the variable or writes to the variable.

Note signal net nodes/variable names are not dependent on macro names and/or macro terminal pointer names, these can change from system to system.

```
// --------------------------- DPLIB Variables ------------------------
// Declare the net variables being used by the DP Lib Macro here
#pragma DATA_SECTION(RltA,"Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(RltB,"Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(RltC,"Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(RltD,"Cla1ToCpuMsgRAM");
volatile float RltA,RltB,RltC,RltD;
```

Step 4 **Call the Peripheral configuration function** `ADC_SOC_CNF(int ChSel[],int TrigSel[],int ACQPS[], int IntChSel, int mode)` in `{ProjectName}-Main.c`, this function is defined in `ADC_SOC_CNF.c`. This file must be included manually into the project.

```
/* Configure ADC channel 0 to convert ADCINB5, and ADC channel 1 to
convert the ADCINA3, ADC Channel 2 converts ADCINA7 and ADC channel 5
converts ADCINB2. The ADC is configured in start stop mode and ADC
Interrupt is disabled. ADC Channel 0,2 is configured to use PWM1 SOCA
and channel 1,5 is configured to use PWM 5 SOCB as trigger. The
following code snippet assumes that the PWM peripherals have been
configured appropriately to generate a SOCA and SOCB */

// Specify ADC Channel - pin Selection for Configuring the ADC
ChSel[0] = 13;          // ADC B5
ChSel[1] = 3;           // ADC A3
ChSel[2] = 7;           // ADC A7
ChSel[5] = 10;          // ADC B2

// Specify the Conversion Trigger for each channel
TrigSel[0]= ADCTRIG_EPWM1_SOCA;
TrigSel[1]= ADCTRIG_EPWM5_SOCB;
TrigSel[2]= ADCTRIG_EPWM1_SOCA;
TrigSel[5]= ADCTRIG_EPWM5_SOCB;

// Call the ADC Configuration Function
ADC_SOC_CNF(ChSel,TrigSel,ACQPS,1,0);
```

Step 5 **"Call"** the `DPL_CLAInit()` to initialize the macros and **"connect"** the module terminals to the signal nets in "C" in `{ProjectName}-Main.c`

```
//----------Connect the macros to build a system-------------------
// Digital Power (DP) library initialisation
DPL_CLAInit();
// ADCDRV_4ch block connections
ADCDRV_4ch_CLA_RltA=&RltA;
ADCDRV_4ch_CLA_RltB=&RltB;
ADCDRV_4ch_CLA_RltC=&RltC;
ADCDRV_4ch_CLA_RltD=&RltD;
```

Step 6 **Call the CLA_Init()** function, this function is provided as a template inside the system software Main.c file and can be modified to suit the needs of the application i.e. change memory

allocation, add tasks etc. However Task 8 is reserved to do any initializations that need to be done to use the macro block. This task is software forced one time at the end of this function. How this task is used to initialize the net variables is discussed in **Step 10**.

```
//----------------------Initialize the CLA-------------------------
CLA_Init();
```

Step 7 **Add** the CLA Task Service Routine and Macro Initialization assembly file "{ProjectName}-DPL-CLA.asm" to the project

Step 8 **Include** the Macro's assembly file in the {ProjectName}-DPL-CLA.asm

```
;Include files for the Power Library Macro's being used by the system
.include "ADCDRV_4ch_CLA.asm"
```

Step 9 **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-callable DPL_Init() function which is defined in {ProjectName}-DPL-ISR.asm. Four numbers need to be specified to identify which result registers are read and scaled results written to the respective pointers. The following code snippet illustrates this:

```
AdcResult.ADCRESULT0 -> (Scale) -> *(ADCDRV_4ch_RltA)
AdcResult.ADCRESULT5 -> (Scale) -> *(ADCDRV_4ch_RltB)
AdcResult.ADCRESULT2 -> (Scale) -> *(ADCDRV_4ch_RltC)
AdcResult.ADCSESULT1 -> (Scale) -> *(ADCDRV_4ch_RltD)
```

```
;Macro Specific Initialization Functions
ADCDRV_4ch_CLA_INIT 0,5,2,1    ; ADCDRV_4ch_CLA Initialization
```

Step 10 **Call the run time macro** in assembly inside the CLA Task which is defined in {ProjectName}-CLA.asm. This task is the task which is repeatedly called in a loop for the control algorithm.

```
;"Call" the Run macro
ADCDRV_4ch_CLA 0,5,2,1
```

Step 11 **Initialize variable** which is in the CLA writable memory location in CLA Task 8, this task is forced by the CLA_Init() function.

Note variable names in the following step are not associated with the module and can change depending on Step 2, and or the system in consideration

```
;(Initialize variables in CLA writable memory space )
.ref _RltA
.ref _RltB
.ref _RltC
.ref _RltD
```

```
MMOVF32 MR0, #0.0L
MMOV32 @_RltA,MR0
MMOV32 @_RltB,MR0
MMOV32 @_RltC,MR0
MMOV32 @_RltD,MR0
```

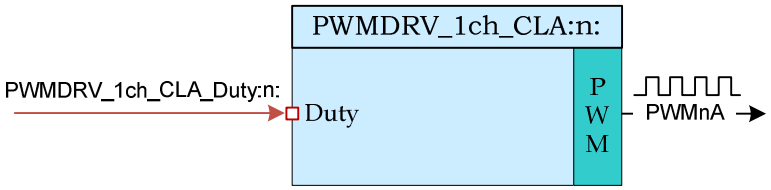
Step 12 Include the memory sections the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD` for the macro's net terminals and internal data and allocate them to the desired location.
See **section 3.2** "Memory Requirements in CLA based systems for F28035" for more details.

```
/* ADCDRV_4ch_CLA section */
ADCDRV_4ch_CLA_Section     : > CLA1_MSGRAMHIGH  PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
ADCDRV_4ch_CLA_RltA	Output Pointer	Pointer to 32 bit fixed point data location storing the result of the module.	Float: [0, 1)
ADCDRV_4ch_CLA_RltB	Output Pointer	Pointer to 32 bit fixed point data location storing the result of the module.	Float: [0, 1)
ADCDRV_4ch_CLA_RltC	Output Pointer	Pointer to 32 bit fixed point data location storing the result of the module.	Float: [0, 1)
ADCDRV_4ch_CLA_RltD	Output Pointer	Pointer to 32 bit fixed point data location storing the result of the module.	Float: [0, 1)

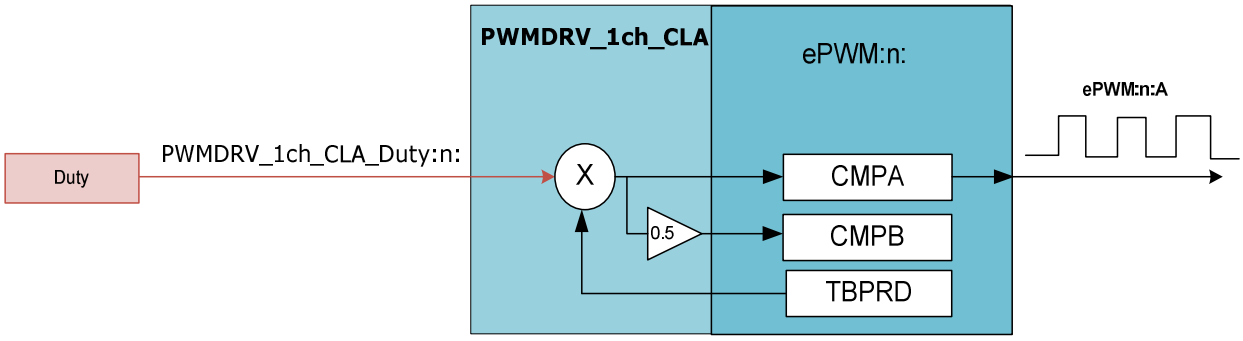
Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a duty on PWM channel A, dependent on the value of the input variable.



Macro File: `PWMDRV_1ch_CLA.asm`

Peripheral Initialization File: `PWMDRV_1ch_Cnf.c`

Description: This assembly macro provides the interface between a DP library net variable and the ePWM module. The macro converts the normalized float input pointed to by the Net Pointer PWMDRV_1ch_CLA_Duty:n: into an unsigned Q0 number scaled by the PWM period value, and stores this value in the EPwmRegs:n:.CMPA. The module also writes half the value of CMPA into CMPB register. This is done to enable ADC Start of conversions to occur close to the mid point of the PWM waveform to avoid switching noise. Which PWM Macro is driven and what period value is used for scaling is determined by the instance number of the macro i.e. :n:.



This macro is used in conjunction with the peripheral configuration file PWMDRV_1ch_Cnf.c. The file defines the function

```
void PWMDRV_1ch_CNF(int16 n, int16 period, int16 mode, int16
   phase)
```

where

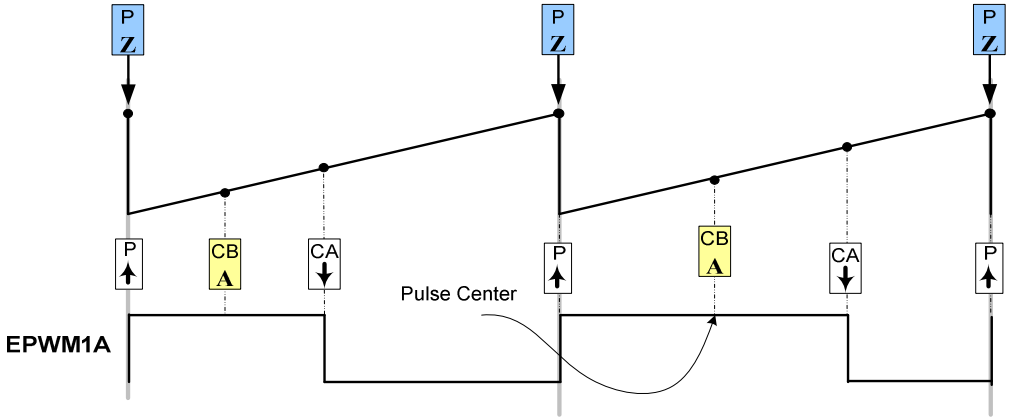
n is the PWM peripheral number which is configured in up-count mode
Period is the maximum count value of the PWM timer

Mode	determines whether the PWM is to be configured as slave or master, when configured as the master the TBSYNC signal is ignored by the PWM module. Mode =1 PWM configured as a master Mode = 0 PWM configured as slave
Phase	Specifies the phase offset that is used when the PWM module is synced, this value only has meaning when the PWM is configured as a slave.

Detailed Description

The following section explains how this module can be used to excite buck power stage. . To configure 100Khz switching frequency with CPU operating at 60Mhz, the Period value needed is (System Clock/Switching Frequency) = 600.

Buck converter driven by PWMDRV_1ch module



PWM generation with the EPWM module.

Usage:

`Step 1` **Add library header file** in the file `{ProjectName}-Main.c`

```
#include “DPLib.h”
```

Step 2 Declare the terminal pointers in C in the file {ProjectName}-Main.c

```
// ---------------------------- DPLIB Net Pointers ---------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PWMDRV_1ch_CLA - instance #1
extern volatile float *PWMDRV_1ch_CLA_Duty1     ;
```

Step 3 Declare signal net nodes/ variables in C in the file {ProjectName}-Main.c, also specify the appropriate memory location variable need using the **#pragma** preprocessor directive. This is determined by whether the CLA reads from the variable or writes to the variable.

Note signal net nodes/variable names are not dependent on macro names and/or macro terminal pointer names, these can change from system to system.

```
// ---------------------------- DPLIB Variables ------------------------
// Declare the net variables being used by the DP Lib Macro here
#pragma DATA_SECTION(Duty, "CpuToCla1MsgRAM");
volatile float Duty;
```

Please note that the memory region where Duty is placed depends on which processor is writing to this value. It can be either CpuToCLA or CLAToCPU, however for convenience we assume an open loop system in which Duty is being set by the C28x CPU.

Step 4 Call the Peripheral configuration function PWMDRV_1ch_CNF(int16 n, int16 period, int16 mode, int16 phase) in {ProjectName}-Main.c, this function is defined in PWMDRV_1ch_Cnf.c. This file must be included manually into the project.

```
// Configure PWM1 for 100Khz, @60Mhz CPU Clock, in master mode
PWMDRV_1ch_CNF(1,600,1,0);
```

Step 5 “Call” the DPL_CLAInit() to initialize the macros and **”connect”** the module terminals to the Signal Nets in “C” in {ProjectName}-Main.c

```
//----------Connect the macros to build a system--------------------
// Digital Power (DP) library initialisation
DPL_CLAInit();
// PWMDRV_1ch block connections
PWMDRV_1ch_CLA_Duty1=&Duty;
// Initialize the net variables
Duty=(float)0.0;
```

Step 6 **Call the CLA_Init()** function, this function is provided as a template inside the system software Main.c file and can be modified to suit the needs of the application i.e. change memory allocation, add tasks etc. However Task 8 is reserved to do any initializations that need to be done to use the macro block. This task is software forced one time at the end of this function. How this task is used to initialize the net variables is discussed in **Step 10**.

```
//----------------------Initialize the CLA-------------------------
CLA_Init();
```

Step 7 **Add** the CLA Task Service Routine and Macro Initialization assembly file "{ProjectName}-DPL-CLA.asm" to the project

Step 8 **Include** the Macro's assembly file in the {ProjectName}-DPL-CLA.asm

```
;Include files for the Power Library Macro's being used by the system
.include "PWMDRV_1ch_CLA.asm"
```

Step 9 **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-Callable DPL_CLAInit() function which is defined in {ProjectName}-DPL-ISR.asm

```
;Macro Specific Initialization Functions
PWMDRV_1ch_CLA_INIT 1   ; PWMDRV_1ch_CLA Initialization
```

Step 10 **Call the run time macro** in assembly inside the CLA Task which is defined in {ProjectName}-CLA.asm. This task is the task which is repeatedly called in a loop for the control algorithm.

```
;"Call" the Run macro
PWMDRV_1ch_CLA 1  ; Run PWMDRV_1ch_CLA
```

Step 11 **Initialize variable** which is in the CLA writable memory location in CLA Task 8, this task is forced by the CLA_Init() function.

Note variable names in the following step are not associated with the module and can change depending on Step 2, and or the system in consideration.

```
;(Initialize variables in CLA writable memory space )
;.ref _Duty
;MMOVF32 MR0, #0.0L
;MMOV16 @_Duty,MR0
```

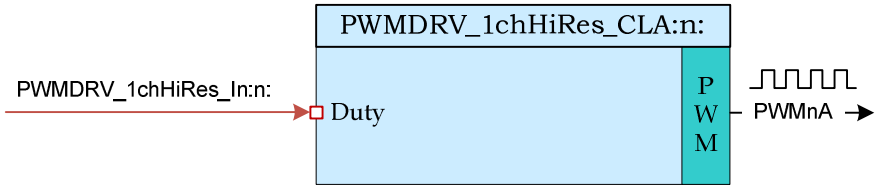
Step 12 Include the memory sections the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD` for the macro's net terminals and internal data and allocate them to the desired location.
See **section 3.2** "Memory Requirements in CLA based systems for F28035" for more details.

```
/* PWMDRV_1ch_CLA driver section */
PWMDRV_1ch_CLA_Section      : > CLA1_MSGRAMHIGH       PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description & Memory Constraints**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
PWMDRV_1ch_CLA_Duty:n:	Input Pointer (CLA Read)	Pointer to 32 bit fixed point input data location storing Duty Value	Float: [0, 1)

Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a high resolution duty on PWM channel A, using the Hi Res feature of the PWM, dependent on the value of the input variable.

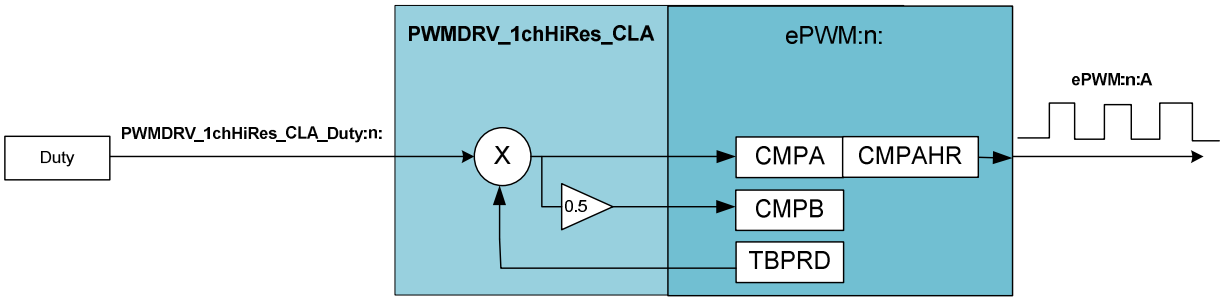


Macro File: `PWMDRV_1chHiRes_CLA.asm`

Peripheral Initialization File: `PWM_1chHiRes_Cnf.c`

Description: With a conventional PWM the resolution achieved is limited by the CPU clock/system clock. C2000 devices have PWM modules with Micro Edge Positioning technology (MEP) which is capable of positioning an edge very finely by subdividing one coarse system clock of a conventional PWM generator. The time step accuracy is of the order of 150ps. See the device specific data sheet for the typical MEP step size on a particular device.

The assembly macro provides the interface between a DP library net variable and the ePWM module on C28x. The macro converts the normalized float input, pointed to by the Net Pointer PWMDRV_1chHiRes_CLA_Duty:n: into an unsigned Q0 number scaled by the PWM period value, and stores this value in the EPwmRegs:n:.CMPA and EPwmRegs:n:.CMPAHR. The module also writes half the value of CMPA into CMPB register. This is done to enable ADC Start of conversions to occur close to the mid point of the PWM waveform to avoid switching noise. Which PWM Macro is driven and what period value is used for scaling is determined by the instance number of the macro i.e. :n:.



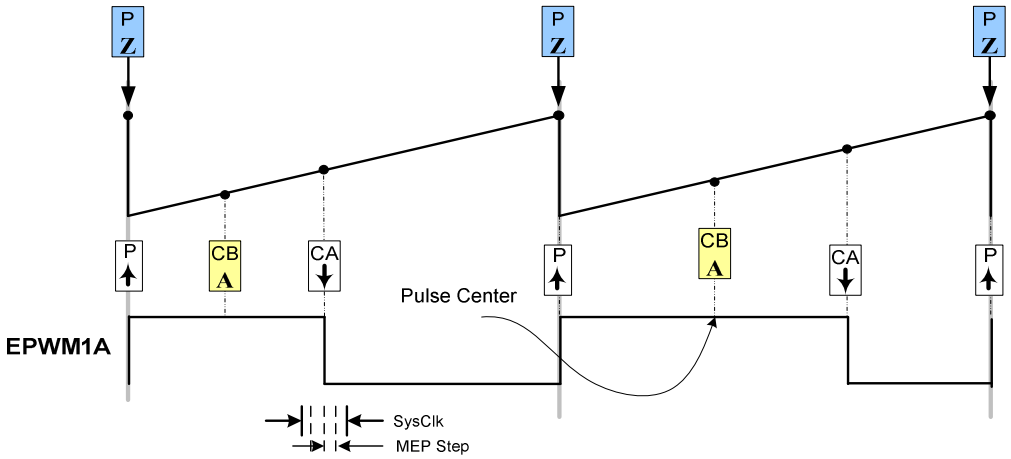
This macro is used in conjunction with the Peripheral configuration file PWM_1chHiRes_Cnf.c. The file defines the function

```
void PWM_1chHiRes_CNF(int16 n, int16 period, int16 mode,
   int16 phase)
```

where

n	is the PWM Peripheral number which is configured in up count mode
Period	is the maximum count value of the PWM timer
Mode	determines whether the PWM is to be configured as slave or master, when configured as the master the TBSYNC signal is ignored by the PWM module. Mode =1 PWM configured as a master Mode = 0 PWM configured as slave
Phase	Specifies the phase offset that is used when the PWM module is synced, this value only has meaning when the PWM is configured as a slave.

The function configures the PWM peripheral in up count mode. In up count mode to configure 100Khz switching frequency for the PWM when CPU is operating at 60Mhz, the Period value needed is (System Clock/Switching Frequency) = 600. The figure below, shows with help of a timing diagram how the PWM is configured to generate the waveform.



PWM generation with the EPWM module.

Note: The MEP varies from device to device and operating condition, for the module to work appropriately a Scale Factor Optimization (SFO) function must be called in a slower background task to auto calibrate the MEP step size. Note the SFO function can only be called by the CPU and not the CLA.

Usage:

Step 1 Add library header file and the Scale Factor Optimizer Library header in the file {ProjectName}-Main.c. Please use V6 or higher of the SFO library for this module to work appropriately. The Library also needs to included in the project manually. The Library can be found at

controlSUITE\device_support\<Device_Name>\<Version>\<Device_Name>_common\lib

```
#include “DPLib.h”
#include "SFO_V6.h"
```

Step 2 Declare the terminal pointers in C in the file {ProjectName}-Main.c and add variable declaration for the variables being used by the SFO Library.

```
// ---------------------------- DPLIB Net Pointers ---------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PWMDRV_1chHiRes_CLA - instance #1
extern volatile float *PWMDRV_1chHiRes_CLA_Duty1;
//=========================================================================
// The following declarations are required in order to use the SFO
// library functions:
//
int MEP_ScaleFactor; // Global variable used by the SFO library
                     // Result can be used for all HRPWM channels
                     // This variable is also copied to HRMSTEP
                     // register by SFO() function.

int status;
```

Step 3 Declare signal net nodes/ variables in C in the file {ProjectName}-Main.c, also specify the appropriate memory location variable need using the **#pragma** preprocessor directive. This is determined by whether the CLA reads from the variable or writes to the variable.

Note these signal nets name may change from system to system, there is no dependency on the signal net names to the module.

```
// ---------------------------- DPLIB Variables ------------------------
// Declare the net variables being used by the DP Lib Macro here
#pragma DATA_SECTION(Duty, "CpuToCla1MsgRAM");
volatile float Duty;
```

Step 4 Call the peripheral configuration function PWM_1chHiRes_CNF(int16 n, int16 period, int16 mode, int16 phase) in {ProjectName}-Main.c, this function is defined in PWM_1chHiRes_Cnf.c. This file must be included manually into the project.

```
// Configure PWM1 for 100Khz, @60Mhz CPU Clock, in master mode

PWM_1chHiRes_CNF(1,600,1,0);
```

Step 5 **"Call"** the `DPL_CLAInit()` to initialize the macros and **"connect"** the module terminals to the signal nets in "C" in `{ProjectName}-Main.c`. **"Call"** the SFO() function to calculate the HRMSTEP, and update the HRMSTEP register if calibration function returns without error. The User may want to call this function in a background task to account for changing operating conditions.

```
//----------Connect the macros to build a system-------------------
// Digital Power (DP) library initialisation
DPL_CLAInit();
// PWMDRV_1ch block connections
PWMDRV_1chHiRes_CLA_Duty1=&Duty;
// Calling SFO() updates the HRMSTEP register with calibrated
MEP_ScaleFactor.
// MEP_ScaleFactor/HRMSTEP must be filled with calibrated value in order
// for the module to work
status = SFO_INCOMPLETE;

while  (status== SFO_INCOMPLETE){  // Call until complete
        status = SFO();
}
if(status!=SFO_ERROR) { // IF SFO() is complete with no errors
      EALLOW;
      EPwm1Regs.HRMSTEP=MEP_ScaleFactor;
      EDIS;
}
if (status == SFO_ERROR) {
            while(1); // SFO function returns 2 if an error occurs
                      // The code would loop here for infinity if it
                      // returns an error
}
// Initialize the net variables
Duty=(float)0.0;
```

Step 6 **Call the CLA_Init()** function, this function is provided as a template inside the system software Main.c file and can be modified to suit the needs of the application i.e. change memory allocation, add tasks etc. However Task 8 is reserved to do any initializations that need to be done to use the macro block. This task is software forced one time at the end of this function. How this task is used to initialize the net variables is discussed in **Step 11**.

```
//----------------------Initialize the CLA-------------------------
CLA_Init();
```

Step 7 **Add** the CLA Task Service Routine and Macro Initialization assembly file "`{ProjectName}-DPL-CLA.asm`" to the project

Step 8 **Include** the macro's assembly file in the `{ProjectName}-DPL-CLA.asm`

```
;Include files for the Power Library Macro's being used by the system
.include "PWMDRV_1chHiRes_CLA.asm"
```

Step 8 **Instantiate the INIT** macro in assembly (this is one-time pass through code) inside the C-callable DPL_CLAInit() function which is defined in {ProjectName}-DPL-CLA.asm

```
;Macro Specific Initialization Functions
PWMDRV_1chHiRes_CLA_INIT 1    ; PWMDRV_1ch Initialization
```

Step 10 **Call the run time macro** in assembly inside the CLA Task which is defined in {ProjectName}-CLA.asm. This task is the task which is repeatedly called in a loop for the control algorithm.

```
;"Call" the Run macro
PWMDRV_1chHiRes_CLA 1   ; Run PWMDRV_1chHiRes_CLA
```

Step 11 **Initialize variable** which is in the CLA writable memory location in CLA Task 8, this task is forced by the CLA_Init() function.

Note variable names in the following step are not associated with the module and can change depending on Step 2, and or the system in consideration.

```
;(Initialize variables in CLA writable memory space )
;.ref _Duty
;MMOVF32 MR0, #0.0L
;MMOV16 @_Duty,MR0
```

Step 12 **Include the memory sections** the {DeviceName}-{RAM/FLASH}-{ProjectName}.CMD for the macro's net terminals and internal data and allocate them to the desired location.
See **section 3.2** "Memory Requirements in CLA based systems for F28035" for more details.

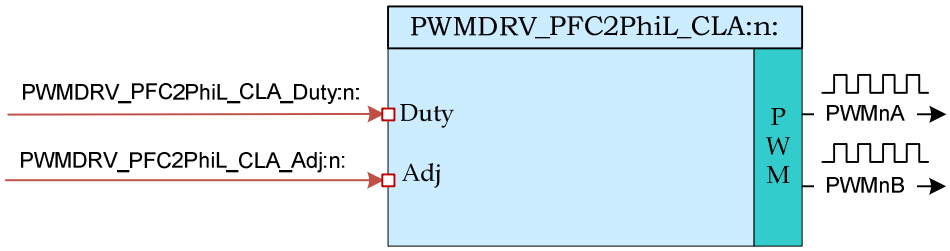
```
/* PWMDRV_1chHiRes_CLA driver section */
PWMDRV_1chHiRes_CLA_Section      : > CLA1_MSGRAMHIGH  PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
PWMDRV_1chHiRes_CLA_Duty:n:	Input Pointer	Pointer to 32 bit fixed point input data location storing Duty Value	Float: [0, 1)

PWMDRV_PFC2PhiL_CLA — *PWM Driver for Two Phase Interleaved PFC Stage using CLA*

Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, controls two PWM generators that cane be used to drive a 2 phase interleaved PFC stage.



Macro File: `PWMDRV_PFC2PhiL_CLA.asm`

Peripheral Initialization File: `PWM_PFC2PhiL_Cnf.c`

Description: This module forms the interface between the control software and the device PWM pins. The macro converts the normalized float input pointed to by the Net Pointer PWMDRV_PFC2PhiL_CLA_Duty:n: into an unsigned Q0 number scaled by the PWM period value, and stores this value in the EPwmRegs:n:.CMPA. Which PWM module is written to is determined by the instance number of the macro i.e. :n:.

The value pointed by the PWMDRV_PFC2PhiL_CLA_Adj:n: stores a normalized float number that is scaled with the PWM period value to add an offset to the duty being driven on PWMnA and PWMnB. The value stored can be positive or negative depending on whether the duty driven on PWMnB needs to greater or smaller relative to PWMnA. In summary:

$$CMPA = Duty * PWMPeriod$$

$$CMPB = (1 - Adj - Duty) * PWMPeriod$$

This macro is used in conjunction with the Peripheral configuration file PWM_PFC2PHIL_CNF.c. The file defines the function

```
void PWM_PFC2PHIL_CNF(int16 n, int16 period)
```

where

n is the PWM Peripheral number which is configured in up down count mode

Period is twice the maximum value of the PWM counter (Note the configuration function takes care of the up down count modulation and scales the TBPRD value accordingly)

Detailed Description

The following section explains how this module can be used to excite a two phase interleaved PFC stage. As up down count mode is used, to configure 100Khz switching frequency, at 60Mhz system clock the period value of (System Clock/Switching Frequency) = 600 must be used for the CNF function. the CNF module divides the value to take into account the up down count mode and stores TBPRD value of 600/2=300.

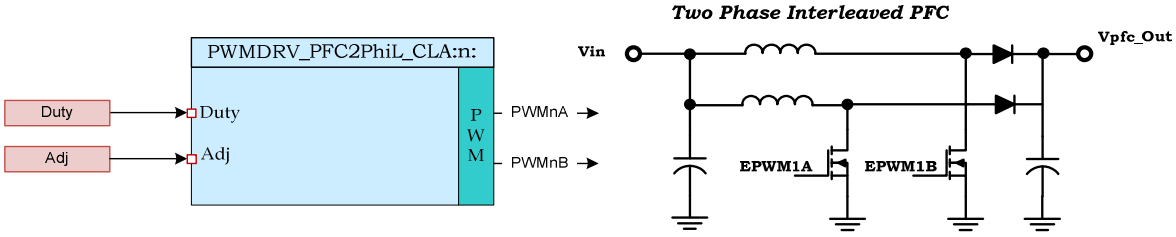


Figure: PFC2PhiL driven by PWMDRV_PFC2PhiL module

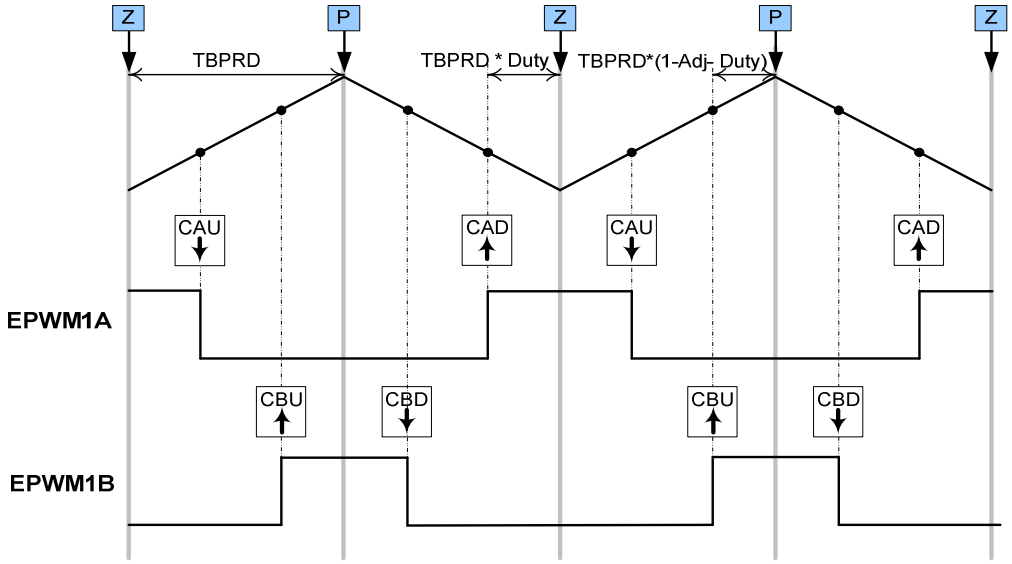


Figure: PWM generation for PFC2PhiL stage with the EPWM module.

Usage:

`Step 1` **Add library header file** in the file `{ProjectName}-Main.c`

```
#include "DPLib.h"
```

Step 2 **Declare the terminal pointers in C** in the file `{ProjectName}-Main.c`

```
// ---------------------------- DPLIB Net Pointers --------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PWMDRV_PFC2PhiL_CLA - instance #1
extern volatile float *PWMDRV_PFC2PhiL_CLA_Duty1;
extern volatile float *PWMDRV_PFC2PhiL_CLA_Adj1;
```

Step 3 **Declare signal net nodes/ variables in C** in the file {ProjectName}-Main.c, also specify the appropriate memory location variable need using the **#pragma** preprocessor directive. This is determined by whether the CLA reads from the variable or writes to the variable.

Note signal net nodes/variable names are not dependent on macro names and/or macro terminal pointer names, these can change from system to system.

```
// ---------------------------- DPLIB Variables ----------------------
// Declare the net variables being used by the DP Lib Macro here
#pragma DATA_SECTION(Duty, "CpuToCla1MsgRAM");
#pragma DATA_SECTION(Adj,  "CpuToCla1MsgRAM");
volatile float Duty,Adj;
```

Please note that the memory region where Duty is placed depends on what processor is writing to this value. It can be either CpuToCLA or CLAToCPU, however for convenience we assume an open loop system in which Duty and Adj is being set by the C28x.

Step 4 **Call the Peripheral configuration function** `PWM_PFC2PHIL_CNF(int16 n, int16 period)` in `{ProjectName}-Main.c`, this function is defined in `PWM_PFC2PhiL_Cnf.c`. This file must be included manually into the project.

```
// Configure PWM1 for switching frequency 100Khz, @60Mhz CPU clock =>
period = (60Mhz/100Khz)/2 =300

PWM_PFC2PHIL_CNF(1,300);
```

Step 5 **"Call"** the `DPL_CLAInit()` to initialize the macros and **"connect"** the module terminals to the signal nets in "C" in `{ProjectName}-Main.c`

```
//----------Connect the macros to build a system-------------------

// Digital Power (DP) library initialisation
DPL_CLAInit();
// PWMDRV_PFC2PhiL_CLA block connections
PWMDRV_PFC2PhiL_CLA_Duty1=&Duty;
PWMDRV_PFC2PhiL_CLA_Adj1 =&Adj;
// Initialize the net variables
Duty=(float)(0.0);
Adj =(float)(0.0);
```

Step 6 **Call the CLA_Init()** function, this function is provided as a template inside the system software Main.c file and can be modified to suit the needs of the application i.e. change memory allocation, add tasks etc. However Task 8 is reserved to do any initializations that need to be done to use the macro block. This task is software forced one time at the end of this function. How this task is used to initialize the net variables is discussed in **Step 10**.

```
//----------------------Initialize the CLA-------------------------
CLA_Init();
```

Step 7 **Add** the CLA Task Service Routine and Macro Initialization assembly file "{ProjectName}-DPL-CLA.asm" to the project

Step 8 **Include** the Macro's assembly file in the {ProjectName}-DPL-CLA.asm

```
;Include files for the Power Library Macro's being used by the system

.include "PWMDRV_PFC2PhiL_CLA.asm"
```

Step 9 **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-Callable DPL_CLAInit() function which is defined in {ProjectName}-DPL-ISR.asm

```
;Macro Specific Initialization Functions
PWMDRV_PFC2PHIL_CLA_INIT 1    ; PWMDRV_PFC2PHIL_CLA Initialization
```

Step 10 **Call the run time macro** in assembly inside the CLA Task which is defined in {ProjectName}-DPL-CLA.asm. This task is the task which is repeatedly called in a loop for the control algorithm.

```
;"Call" the Run macro
PWMDRV_PFC2PhiL_CLA 1   ; Run PWMDRV_PFC2PhiL_CLA
```

Step 11 **Initialize variable** which is in the CLA writable memory location in CLA Task 8, this task is forced by the CLA_Init() function..

Note variable names in the following step are not associated with the module and can change depending on Step 2, and or the system in consideration.

```
;(Initialize variables in CLA writable memory space )
;.ref _Duty
;.ref _Adj
;MMOVF32 MR0, #0.0L
;MMOV16 @_Duty,MR0
;MMOV16 @_Adj,MR0
```

Step 12 Include the memory sections the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD` for the macro's net terminals and internal data and allocate them to the desired location.
See **section 3.2** "Memory Requirements in CLA based systems for F28035" for more details.

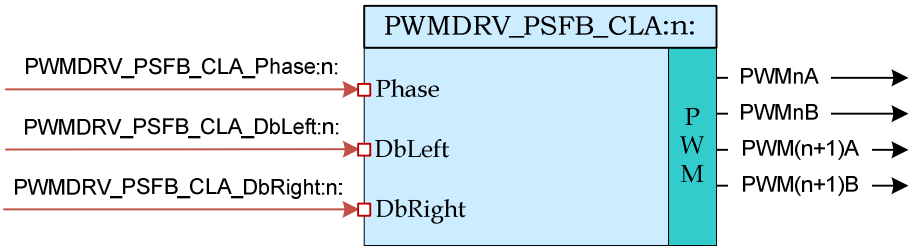
```
/*PWMDRV_PFC2PhiL_CLA sections*/
PWMDRV_PFC2PhiL_CLA_Section   : > CLA1_MSGRAMHIGH     PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description & Memory Restriction**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
PWMDRV_PFC2PhiL_CLA_Duty:n:	Input Pointer (CLA Read)	Pointer to 32 bit fixed point input data location to Duty value	Float(0,1)
PWMDRV_PFC2PhiL_CLA_Adj:n:	Input Pointer (CLA Read)	Pointer to 32 bit fixed point input data location to adjustment value	Float(-1, 1)

PWMDRV_PSFB_CLA	*PWM Driver for Phase Shifted Full Bridge Stage using CLA*

Description: This module controls the PWM generators to control a full bridge by using the phase shifting approach. In addition to phase control, the module offers control over left and right leg dead-band amounts, whereby providing the zero voltage switching capabilities



Macro File: `PWMDRV_PSFB_CLA.asm`

Peripheral Initialization File: `PWM_PSFB_Cnf.c`

Description: This module forms the interface between the control software and the device PWM pins. The macro converts the normalized float input pointed to by the Net Pointer PWMDRV_PSFB_CLA_Phase:n: into an unsigned Q0 number scaled by the PWM period value, and stores this value in the EPwmRegs:n:.TBPHS.

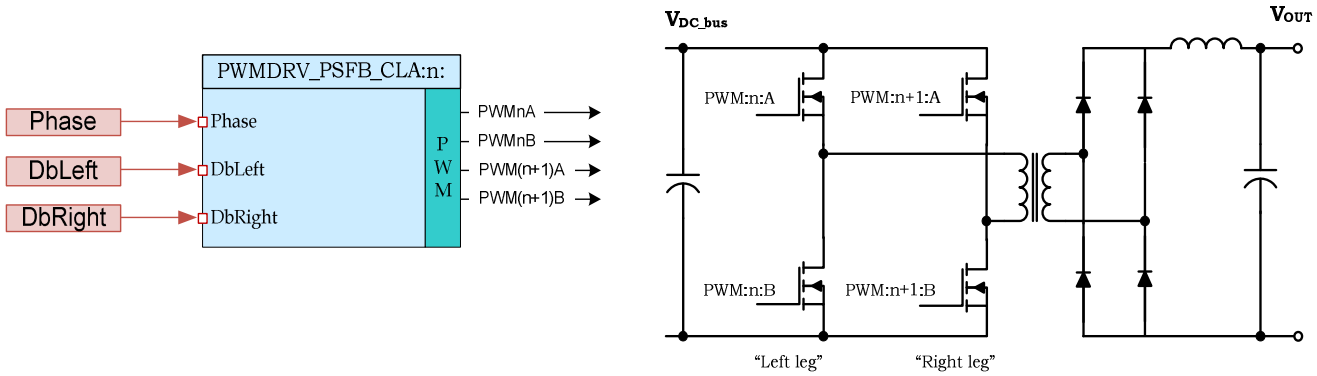
This macro is used in conjunction with the Peripheral configuration file PWM_PSFB_CNF.c. The file defines the function

```
void PWM_PSFB_CNF(int16 n, int16 Period)
```

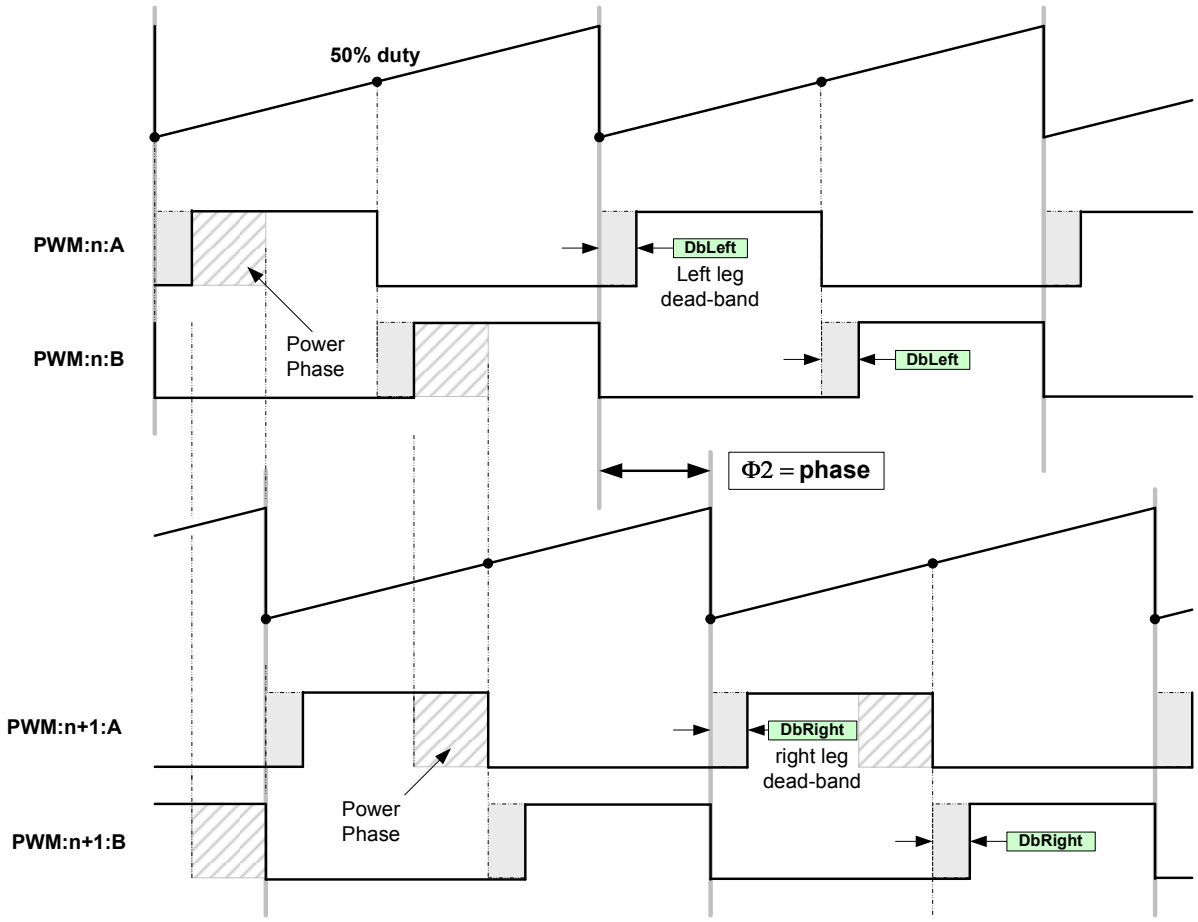
where

n	is the PWM Peripheral number configured for PSFB topology, PWM n+1 is configured to work with synch pulses from PWM n module
Period	is the maximum count value of the PWM timer

Detailed Description The following section explains how module is used to excite PSFB stage. . In up count mode to configure 100Khz switching frequency for the PWM when CPU is operating at 60Mhz, the Period value needed is (System Clock/Switching Frequency) = 600.



Full bridge power converter



Phase shifted PWM generation with the EPWM module.

Usage:

`Step 1` **Add library header file** in the file `{ProjectName}-Main.c`

```
#include “DPLib.h”
```

Step 2 Declare the terminal pointers in C in the file `{ProjectName}-Main.c`

```
// ---------------------------- DPLIB Net Pointers ------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PWMDRV_PSFB - instance #1
extern volatile float *PWMDRV_PSFB_CLA_Phase1;
extern volatile long *PWMDRV_PSFB_CLA_DbLeft1;
extern volatile long *PWMDRV_PSFB_CLA_DbRight1;
```

Step 3 **Declare signal net nodes/ variables in C** in the file `{ProjectName}-Main.c, also` specify the appropriate memory location variable need using the **#pragma** preprocessor directive. This is determined by whether the CLA reads from the variable or writes to the variable.

Note signal net nodes/variable names are not dependent on macro names and/or macro terminal pointer names, these can change from system to system.

```
// ---------------------------- DPLIB Variables ------------------------

// Declare the net variables being used by the DP Lib Macro here

#pragma DATA_SECTION(Phase,   "CpuToCla1MsgRAM");
#pragma DATA_SECTION(DbLeft,  "CpuToCla1MsgRAM");
#pragma DATA_SECTION(DbRight, "CpuToCla1MsgRAM");

volatile float Phase;

volatile long DbLeft,DbRight;
```

Please note that the memory region where Phase, DbLeft and DbRight are placed depends on which processor is writing to these variables. Here for convenience we assume an open loop system in which Duty is being set by the C28x, hence it resides in the CpuToCla1MsgRAM.

Step 4 **Call the Peripheral configuration function** `PWM_PSFB_CNF(int16 n, int16 period)` in `{ProjectName}-Main.c`, this function is defined in `PWM_PSFB_Cnf.c`. This file must be included manually into the project.

```
// Configure PWM1 for 100Khz, @60Mhz CPU Clock, in master mode

PWM_PSFB_CNF(1,600);
```

Step 5 **“Call”** the `DPL_CLAInit()` to initialize the macros and **”Connect”** the module terminals to the Signal Nets in “C” in `{ProjectName}-Main.c`

```
//----------Connect the macros to build a system-------------------

// Digital Power (DP) library initialisation
DPL_CLAInit();
// PWMDRV_PSFB block connections
PWMDRV_PSFB_Phase1    =  &Phase;
```

```
PWMDRV_PSFB_DbLeft1 =   &DbLeft;
PWMDRV_PSFB_DbRight1=   &DbRight;

// Initialize the net variables
Phase=(float)0.0;
DbLeft=0;
DbRight=0;
```

Step 6 Call the CLA_Init() function, this function is provided as a template inside the system software Main.c file and can be modified to suit the needs of the application i.e. change memory allocation, add tasks etc. However Task 8 is reserved to do any initializations that need to be done to use the macro block. This task is software forced one time at the end of this function. How this task is used to initialize the net variables is discussed in **Step 10**.

```
//----------------------Initialize the CLA-------------------------

CLA_Init();
```

Step 7 **Add** the CLA Task Service Routine and Macro Initialization assembly file "{ProjectName}-DPL-CLA.asm" to the project

Step 8 **Include** the Macro's assembly file in the {ProjectName}-DPL-CLA.asm

```
;Include files for the Power Library Macro's being used by the system

.include "PWMDRV_PSFB_CLA.asm"
```

Step 9 **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-Callable DPL_CLAInit() function which is defined in {ProjectName}-DPL-ISR.asm

```
;Macro Specific Initialization Functions
PWMDRV_PSFB_CLA_INIT 1,2      ; PWMDRV_PSFB_CLA Initialization
```

Step 10 **Call the run time macro** in assembly inside the CLA Task which is defined in {ProjectName}-CLA.asm. This task is the task which is repeatedly called in a loop for the control algorithm.

```
;"Call" the Run macro
PWMDRV_PSFB_CLA 1,2     ; Run PWMDRV_PSFB_CLA
```

Step 11 **Initialize variable** which is in the CLA writable memory location in CLA Task 8, this task is forced by the CLA_Init() function..

Note variable names in the following step are not associated with the module and can change depending on Step 2, and or the system in consideration.

```
;(Initialize variables in CLA writable memory space if any)
;.ref _Phase
;MMOVF32 MR0, #0.0L
;MMOV16 @_Phase,MR0
```

Step 12 Include the memory sections the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD` for the macro's net terminals and internal data and allocate them to the desired location.
See **section 3.2** "Memory Requirements in CLA based systems for F28035" for more details.

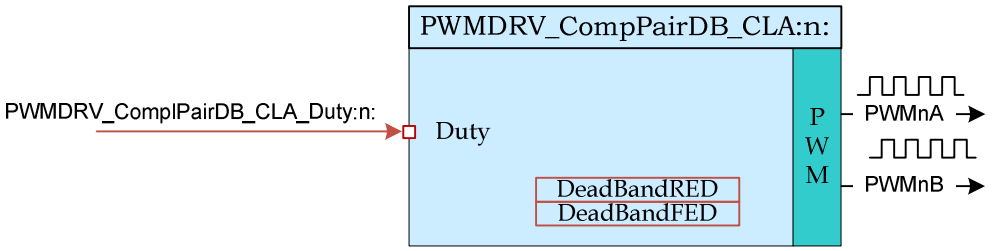
```
/* PWMDRV_PSFB_CLA driver section */
PWMDRV_PSFB_CLA_Section : > CLA1_MSGRAMHIGH     PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
PWMDRV_PSFB_CLA_Phase:n:	Input Pointer	Pointer to 32 bit fixed point input data location to Phase value	Float(0,1)
PWMDRV_PSFB_CLA_DbLeft:n:	Input Pointer	Pointer to 32 bit fixed point input data location to Dead Band Value for Left Leg	Q0
PWMDRV_PSFB_CLA_DbRight:n:	Input Pointer	Pointer to 32 bit fixed point input data location to Dead Band Value for Right Leg	Q0

PWMDRV_ComplPairDB_CLA — *PWM Driver for complimentary chA and chB PWM*

Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a duty on PWM channel A, and a complimentary PWM on channel B with dead band. The module uses the Deadband module inside the EPWM peripheral to generate the complimentary waveforms.



Macro File: `PWMDRV_ComplPairDB_CLA.asm`

Peripheral Initialization File: `PWM_ComplPairDB_Cnf.c`

Description: This assembly macro provides the interface between a DP library net variable and the ePWM module on C28x. The macro converts the normalized float input, pointed to by the Net Pointer PWMDRV_ComplPairDB_CLA_Duty:n: into an unsigned Q0 number scaled by the PWM period value, and stores this value in the EPwmRegs:n:.CMPA. The corresponding configuration file has the deadband module configured to output complimentary waveform on chB with a dead band.

This macro must be used in conjunction with the Peripheral configuration file PWM_ComplPairDB_Cnf.c. The file defines the function

```
void PWM_ComplPairDB_CNF(int16 n, int16 period, int16 DbRed,
int16 DbFed)
```

where

n	is the PWM Peripheral number which is configured in up count mode
Period	is the maximum value of the PWM counter

The function configures the PWM peripheral in up count mode and configured the dead band submodule to output complimentary PWM waveforms. The falling edge delay is implemented by delaying the rising edge of the channel B using the dead band module in the PWM peripheral. The module outputs an active high duty on ChA of the PWM peripheral and a complementary active low duty cycle on ChB.

The configuration function only configures the dead band at initialization time however it may be needed to change the dead band dependent on system condition. This can be done by calling the function

```
void PWM_ComplPairDB_UpdateDB (int16 n, int16 DbRed, int16
   DbFed)
```

where

n	is the PWM Peripheral number
DbRed	is the new rising edge delay
DbFed	is the new falling edge delay

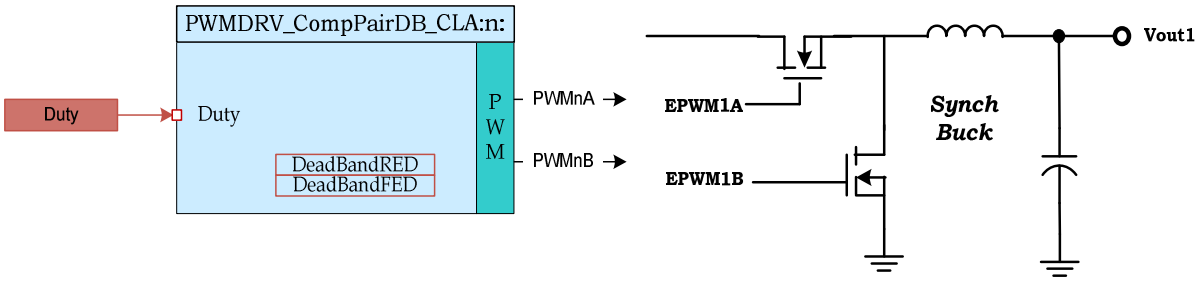
Note this function is only callable from the CPU.

Alternatively an CLA assembly macro is provided to update the deadband, if this needs to be done at a faster rate inside the ISR. The dead band update assembly macro, `PWMDRV_ComplPairDB_CLA_UpdateDB` updates the deadband registers with values stored in the macro variables `PWMDRV_ComplPairDB_CLA_DeadBandRED:n:` and `PWMDRV_ComplPairDB_CLA_DeadBandFED:n:`

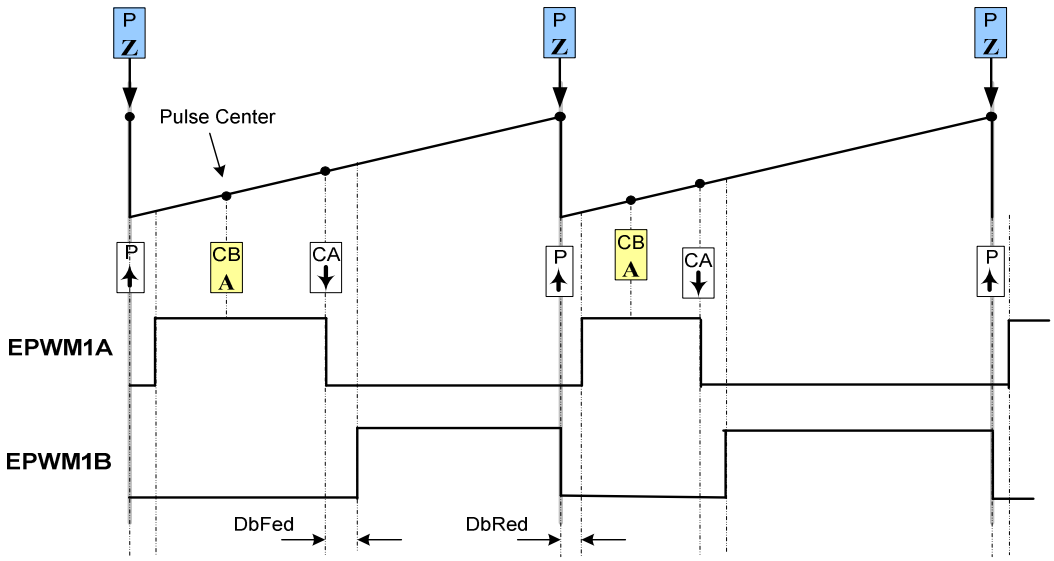
.

Detailed Description

The following section explains how this module can be used to excite a synchronous buck power stage which uses two NFET's. (Please note this module is specific to synchronous buck power stage using NPN transistors only). The function configures the PWM peripheral in up count mode. In up count mode to configure 100Khz switching frequency for the PWM when CPU is operating at 60Mhz, the Period value needed is (System Clock/Switching Frequency) = 600.



Synchronous Buck converter driven by PWMDRV_ComplPairDB module



PWM generation for CompPairDB PWM DRV Maco

Usage:

Step 1 Add library header file in the file {ProjectName}-Main.c

```
#include "DPLib.h"
```

Step 2 Declare the terminal pointers in C in the file {ProjectName}-Main.c

```
// ---------------------------- DPLIB Net Pointers ------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PWMDRV_ComplPairDB_CLA - instance #1
extern volatile float *PWMDRV_ComplPairDB_CLA_Duty1;
extern volatile int16 PWMDRV_ComplPairDB_CLA_DeadBandRED1;
extern volatile int16 PWMDRV_ComplPairDB_CLA_DeadBandFED1;
```

Step 3 Declare signal net nodes/ variables in C in the file {ProjectName}-Main.c, also specify the appropriate memory location variable need using the **#pragma** preprocessor directive. This is determined by whether the CLA reads from the variable or writes to the variable.

Note signal net nodes/variable names are not dependent on macro names and/or macro terminal pointer names, these can change from system to system.

```
// --------------------------- DPLIB Variables ------------------------

// Declare the net variables being used by the DP Lib Macro here

#pragma DATA_SECTION(Duty,    "CpuToCla1MsgRAM");
#pragma DATA_SECTION(DbLeft,  "CpuToCla1MsgRAM");
#pragma DATA_SECTION(DbRight, "CpuToCla1MsgRAM");

volatile float Phase;

volatile int16 DbLeft, DbRight;
```

Please note that the memory region where Duty, DbLeft and DbRight are placed depends on which processor is writing to these variables. Here, for convenience, we assume an open loop system in which the "Duty" variable is being set by the C28x, hence it resides in the CpuToCla1MsgRAM.

Step 4 Call the peripheral configuration function `PWM_ComplPairDB_CNF(int16 n, int16 period)` in `{ProjectName}-Main.c`, this function is defined in `PWM_ComplPairDB_Cnf.c`. This file must be included manually into the project. The following code snippet configures PWM1 in up-count mode and configures the dead band to be 5 and 4 cycles for the rising edge and falling edge respectively.

```
// Configure PWM1 for 100Khz, @60Mhz CPU Clock

PWM_ComplPairDB_CNF(1,600);
PWM_ComplPairDB_UpdateDB(1,5,4);
```

Step 5 "Call" the `DPL_CLAInit()` to initialize the macros and **"connect"** the module terminals to the signal nets in "C" in `{ProjectName}-Main.c`

```
//----------Connect the macros to build a system-------------------

// Digital Power (DP) library initialisation
DPL_CLAInit();
// PWMDRV_PSFB block connections
PWMDRV_ComplPairDB_CLA_Duty1 = &Duty;

// Initialize the net variables
Duty=(float)(0.0);
```

Step 6 "Call" the CLA_Init() function, this function is provided as a template inside the system software Main.c file and can be modified to suit the needs of the application i.e. change memory allocation, add tasks etc. However Task 8 is reserved to do any initializations that need to be done to use the macro block. This task is software forced one time at the end of this function. How this task is used to initialize the net variables is discussed in **Step 10**.

```
//----------------------Initialize the CLA-------------------------

CLA_Init();
```

Step 7 **Add** the CLA Task Service Routine and Macro Initialization assembly file "{ProjectName}-DPL-CLA.asm" to the project

Step 8 **Include** the Macro's assembly file in the {ProjectName}-DPL-CLA.asm

```
;Include files for the Power Library Macro's being used by the system

.include "PWMDRV_ComplPairDB_CLA.asm"
```

Step 9 **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-Callable DPL_CLAInit() function which is defined in {ProjectName}-DPL-ISR.asm

```
;Macro Specific Initialization Functions
PWMDRV_ComplPairDB_CLA_INIT 1
```

Step 10 **Call the run time macro** in assembly inside the CLA Task which is defined in {ProjectName}-CLA.asm. This task is the task which is repeatedly called in a loop for the control algorithm.

```
;"Call" the Run macro
PWMDRV_ComplPairDB_CLA 1       ; Run PWMDRV_PSFB_CLA
```

Step 11 **Initialize variable** which is in the CLA writable memory location in CLA Task 8, this task is forced by the CLA_Init() function..

Note variable names in the following step are not associated with the module and can change depending on Step 2, and or the system in consideration.

```
;(Initialize variables in CLA writable memory space if any)
;.ref _Duty
;MMOVF32 MR0, #0.0L
;MMOV16 @_Duty,MR0
```

Step 12 **Include the memory sections** the {DeviceName}-{RAM/FLASH}-{ProjectName}.CMD for the macro's net terminals and internal data and allocate them to the desired location.
See **section 3.2** "Memory Requirements in CLA based systems for F28035" for more details.

```
/* PWMDRV_ComplPairDB_CLA driver section */
PWMDRV_ComplPairDB_CLA_Section      : > CLA1_MSGRAMHIGH     PAGE = 1
```

Step 13 **Update Dead Band** This can be done by calling the C function in the {ProjectName}-Main.c file. This function is called by the CPU.

```
/*Update dead band delays */
PWMDRV_ComplPairDB_UpdateDB(1,7,4);
```

If the dead band itself is part of the control loop the following CLA assembly macro can be called in one of the CLA Tasks.

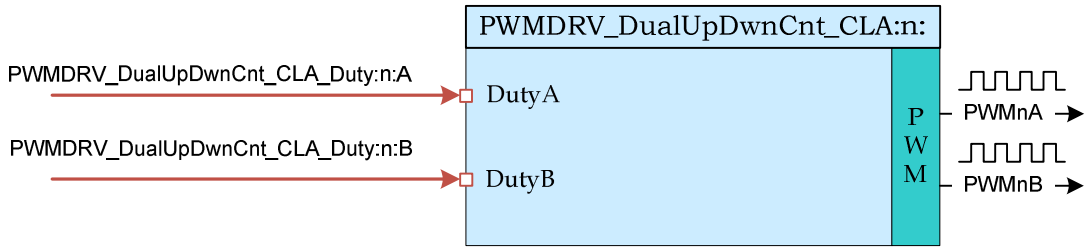
```
;Update dead band delays
PWMDRV_ComplPairDB_CLA_UpdateDB 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
PWMDRV_ComplPairDB_CLA_Duty:n:	Input Pointer	Pointer to 32 bit fixed point input data location storing Duty Value	Float: [0, 1)
PWMDRV_ComplPairDB_CLA_DeadBandRED:n:	Input Variable	Value used by the assembly macro to update the PWM peripheral dead band registers.	Q0
PWMDRV_ComplPairDB_CLA_DeadBandFED:n:	Input Variable	Value used by the assembly macro to update the PWM peripheral dead band registers.	Q0

PWMDRV_DualUpDwnCnt_CLA — *Independent PWM Duty on chA & chB*

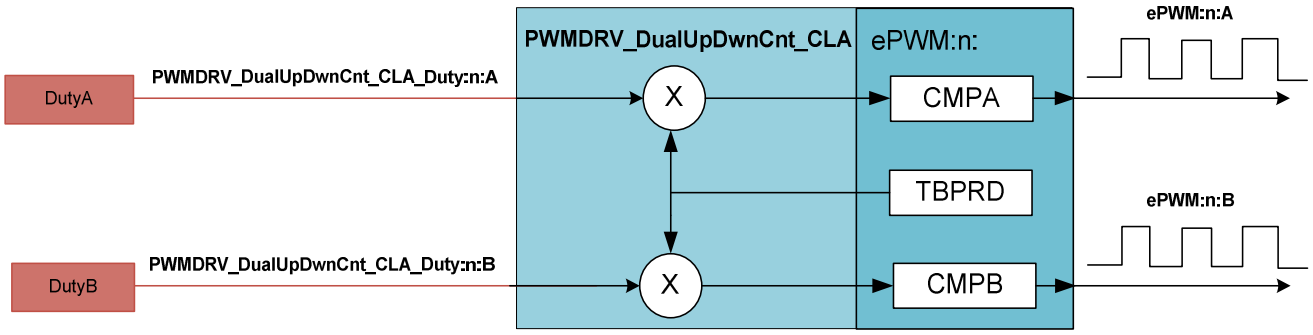
Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a duty on PWM channel A, and PWM channel B dependent on the value of the input variables DutyA and DutyB.



Macro File: `PWMDRV_DualUpDwnCnt_CLA.asm`

Peripheral Initialization File: `PWM_DualUpDwnCnt_Cnf.c`

Description: This assembly macro provides the interface between DP library net variables and the ePWM module on C28x. The macro converts the normalized float input, pointed to by the Net Pointers PWMDRV_DualUPDwnCnt_CLA_Duty:n:A and PWMDRV_DualUPDwnCnt_CLA_Duty:n:B into an unsigned Q0 number scaled by the PWM period value, and stores this value in the EPwmRegs:n:.CMPA and EPwmRegs:n:.CMPB such as to give independently duty cycle control on channel A and B of the PWM module.



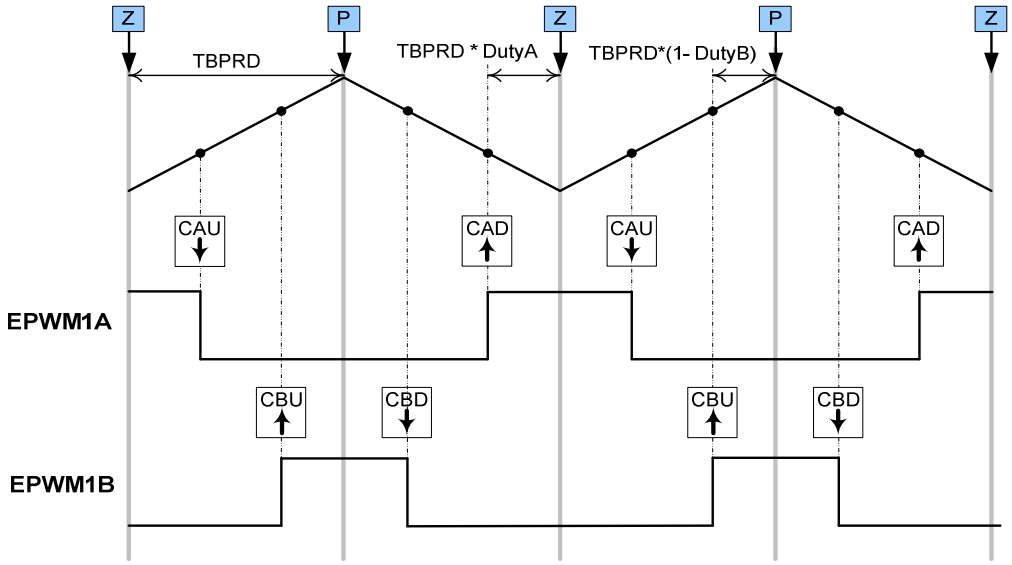
This macro is used in conjunction with the peripheral configuration file PWM_DualUpDwnCnt_Cnf.c. The file defines the function

```
void PWM_DualUpDwnCnt_CNF(int16 n, int16 period)
```

where

n	is the PWM peripheral number which is configured in up-down count mode
Period	is the maximum value of the PWM counter

The function configures the PWM peripheral in up count mode. The figure below, shows with help of a timing diagram how the PWM is configured to generate the waveform. As the module is configured in up-down count mode SOC for the ADC can be triggered at “zero” and/or “period” events to ensure the sample point occurs at the mid-point of the switching cycle. The following section explains how this module can be used to excite a two phase interleaved PFC stage. Up down count mode is used, hence to configure 100Khz switching frequency, at 60Mhz system clock a period value of (System Clock/Switching Frequency) = 600 must be used for the CNF function. The CNF function divides the value to take into account the up down count mode and stored TBPRD value of 600/2=300.



PWM generation with the EPWM module.

Usage:

`Step 1` **Add library header file** in the file `{ProjectName}-Main.c`

```
#include “DPLib.h”
```

`Step 2` **Declare the terminal pointers in C** in the file `{ProjectName}-Main.c`

```
// ---------------------------- DPLIB Net Pointers ------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PWMDRV_DualUpDwnCnt_CLA - instance #1
extern volatile float *PWMDRV_DualUpDwnCnt_CLA_Duty1A;
extern volatile float *PWMDRV_DualUpDwnCnt_CLA_Duty1B;
```

`Step 3` **Declare signal net nodes/ variables in C** in the file {ProjectName}-Main.c, also specify

the appropriate memory location variable need using the **#pragma** preprocessor directive. This is determined by whether the CLA reads from the variable or writes to the variable.

Note signal net nodes/variable names are not dependent on macro names and/or macro terminal pointer names, these can change from system to system.

```
// ---------------------------- DPLIB Variables -----------------------

// Declare the net variables being used by the DP Lib Macro here

#pragma DATA_SECTION(Duty1A,     "CpuToCla1MsgRAM");
#pragma DATA_SECTION(Duty1B,     "CpuToCla1MsgRAM");

volatile float Duty1A,Duty1B;
```

Please note that the memory region where Duty1A,Duty1B are placed depends on which processor is writing to these variables. Here for convenience we assume an open loop system in which Duty is being set by the C28x, hence it resides in the CpuToCla1MsgRAM.

Step 4 Call the peripheral configuration function `PWM_DualUpDwnCnt_CNF(int16 n, int16 period)` in `{ProjectName}-Main.c`, this function is defined in `PWM_DualUpDwnCnt_Cnf.c`. This file must be included manually into the project. The following code snippet configures PWM1 in Up Count mode and configures the dead band to be 5 and 4 cycles for the rising edge and falling edge respectively.

```
// Configure PWM1 for switching frequency 100Khz, @60Mhz CPU clock =>
period = (60Mhz/100Khz)/2 =300

PWM_DualUpDwnCnt_CNF(1,300);
```

Step 5 **"Call"** the `DPL_CLAInit()` to initialize the macros and **"connect"** the module terminals to the signal nets in "C" in `{ProjectName}-Main.c`

```
//----------Connect the macros to build a system-------------------

// Digital Power (DP) library initialisation
DPL_CLAInit();
// PWMDRV_PSFB block connections
PWMDRV_DualUpDwnCnt_CLA_Duty1A = &Duty1A;
PWMDRV_DualUpDwnCnt_CLA_Duty1B = &Duty1B;

// Initialize the net variables
Duty1A=(float)0.0;
Duty1B=(float)0.0;
```

Step 6 **"Call"** the CLA_Init() function, this function is provided as a template inside the system software Main.c file and can be modified to suit the needs of the application i.e. change memory allocation, add tasks etc. However Task 8 is reserved to do any initializations that need to be done to use the macro block. This task is software forced one time at the end of this function. How this task is used to initialize the net variables is discussed in **Step 10**.

```
//----------------------Initialize the CLA-------------------------

CLA_Init();
```

Step 7 **Add** the CLA Task Service Routine and Macro Initialization assembly file "{ProjectName}-DPL-CLA.asm" to the project

Step 8 **Include** the Macro's assembly file in the {ProjectName}-DPL-CLA.asm

```
;Include files for the Power Library Macro's being used by the system

.include "PWMDRV_DualUpDwnCnt_CLA.asm"
```

Step 9 **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-Callable DPL_CLAInit() function which is defined in {ProjectName}-DPL-CLA.asm

```
;Macro Specific Initialization Functions
PWMDRV_DualUpDwnCnt_CLA_INIT 1
```

Step 10 **Call the run time macro** in assembly inside the CLA Task which is defined in {ProjectName}-CLA.asm. This task is the task which is repeatedly called in a loop for the control algorithm.

```
;"Call" the Run macro
PWMDRV_DualUpDwnCnt_CLA 1      ; Run PWMDRV_PSFB_CLA
```

Step 11 **Initialize variable** which is in the CLA writable memory location in CLA Task 8, this task is forced by the CLA_Init() function..

Note variable names in the following step are not associated with the module and can change depending on Step 2, and or the system in consideration.

```
;(Initialize variables in CLA writable memory space if any)
;.ref _Duty1A
;MMOVF32 MR0, #0.0L
;MMOV16 @_Duty1A,MR0
```

Step 12 **Include the memory sections** the {DeviceName}-{RAM/FLASH}-{ProjectName}.CMD for the macro's net terminals and internal data and allocate them to the desired location.
See **section 3.2** "Memory Requirements in CLA based systems for F28035" for more details.

```
/* PWMDRV_DualUpDwnCnt_CLA driver section */
PWMDRV_DualUpDwnCnt_CLA_Section     : > CLA1_MSGRAMHIGH     PAGE = 1
```

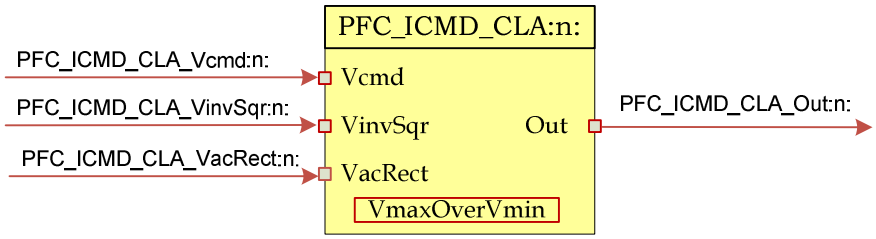
Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
PWMDRV_DualUpDwnCnt_CLA_Duty:n:A	Input Pointer	Pointer to 32 bit fixed point input data location storing DutyA Value	Float: [0, 1)
PWMDRV_DualUpDwnCnt_CLA_Duty:n:B	Input Pointer	Pointer to 32 bit fixed point input data location storing DutyB Value	Float: [0, 1)

5.3. Application Specific

PFC_ICMD_CLA	*Current Command for Power Factor Correction using CLA*

Description: This software module performs a computation of the current command for the power factor correction



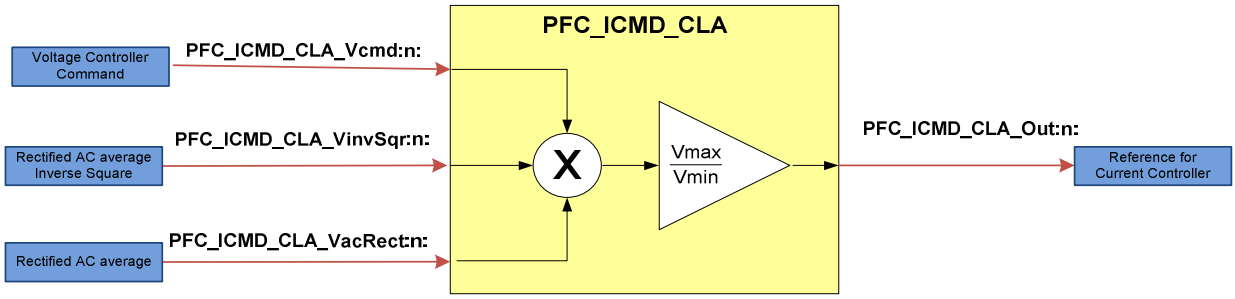
Macro File: `PFC_ICMD_CLA.asm`

Technical: This software module performs a computation of the current command for the power factor correction. The inputs to the module are the inverse-square of the averaged line voltage, the rectified line voltage and the output of the voltage controller. The PFC_ICMD_CLA block then generates an output command profile that is half-sinusoidal, with amplitude dependent on the output of the voltage controller. The output is then connected to the current controller to produce the required inductor current.

The input pointers `PFC_ICMD_CLA_Vcmd:n:`, `PFC_ICMD_CLA_VinvSqr:n:` and `PFC_ICMD_CLA_VacRect:n:` points to a variable represented in normalized float. The module multiplies these values together and then scales them by multiplying with a factor which is stored in the internal data `PFC_ICMD_CLA_VmaxOverVmin:n:` The result in normalized float format is written to the variable pointed by the output pointer `PFC_ICMD_CLA_Out:n:`

A PFC stage is typically designed to work over a range of AC line conditions. `PFC_ICMD_CLA_VminOverVmax:n:` is the ratio of minimum over maximum voltage the PFC stage is designed for represented in float format.

The following diagram illustrates the math function operated on in this block.



Usage: This section explains how to use this module. The example assumes a PFC Stage designed for 230VAC to 90VAC and the voltage feedback is designed for peak 350V. Hence,

```
PFC_ICMD_CLA_VmaxOverVmin:n:  = (float)( 230.0/90.0)= 2.5555
```

Step 1 Add library header file in the file {ProjectName}-Main.c

```
#include "DPLib.h"
```

Step 2 Declare the terminal pointers in C in the file {ProjectName}-Main.c

```
// ---------------------------- DPLIB Net Pointers ---------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PFC_ICMD_CLA - instance #1
extern volatile float *PFC_ICMD_CLA_Vcmd1;
extern volatile float *PFC_ICMD_CLA_VacRect1;
extern volatile float *PFC_ICMD_CLA_VinvSqr1;
extern volatile float *PFC_ICMD_CLA_Out1;
extern volatile float PFC_ICMD_CLA_VmaxOverVmin1;
```

Step 3 Declare signal net nodes/ variables in C in the file {ProjectName}-Main.c, also specify the appropriate memory location variable need using the **#pragma** preprocessor directive. This is determined by whether the CLA reads from the variable or writes to the variable.

Note signal net nodes/variable names are not dependent on macro names and/or macro terminal pointer names, these can change from system to system.

```
// ---------------------------- DPLIB Variables ------------------------

// Declare the net variables being used by the DP Lib Macro here

#pragma DATA_SECTION(Vcmd,    "Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(VacRect, "Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(VinvSqr, "Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(CurrCmd, "Cla1ToCpuMsgRAM");

volatile float Vcmd,VacRect,VinvSqr,CurrCmd;
```

Please note that the memory region where the different variables are placed depends on what processor is writing to this value. For the PFC_ICMD module, as the entire PFC code runs on the CLA, all the input data variables would be in a CLA writable memory space and as the Output is written to by the module this needs to be in a CLA writable space. The above code snippets assume the default allocation for CLA writable memory i.e. Cla1ToCpuMsgRAM.

Step 4 "Call" the DPL_CLAInit() function to initialize the macros and **"connect"** the module terminals to the signal nets in "C" in {ProjectName}-Main.c

```
//----------Connect the macros to build a system-------------------
// Digital Power (DP) library initialisation
DPL_CLAInit();
// PFC_ICMD block connections
PFC_ICMD_CLA_Vcmd1=&Vcmd;
PFC_ICMD_CLA_VacRect1=&VacRect;
PFC_ICMD_CLA_VinvSqr1=&VinvSqr;
PFC_ICMD_CLA_Out1=&CurrCmd;
PFC_ICMD_CLA_VmaxOverVmin1=(float)(2.5555);
```

Step 5 **Call the CLA_Init()** function, this function is provided as a template inside the system software Main.c file and can be modified to suit the needs of the application i.e. change memory allocation, add tasks etc. However Task 8 is reserved to do any initializations that need to be done to use the macro block. This task is software forced one time at the end of this function. How this task is used to initialize the net variables is discussed in **Step 9**.

```
//----------------------Initialize the CLA-------------------------
CLA_Init();
```

Step 6 **Add** the CLA Task Service Routine and Macro Initialization assembly file "{ProjectName}-DPL-CLA.asm" to the project

Step 7 Include the Macro's assembly file in the {ProjectName}-DPL-CLA.asm

```
;Include files for the Power Library Macro's being used by the system

.include "PFC_ICMD_CLA.asm"
```

Step 8 Instantiate the INIT macro in assembly (this is one-time pass through code) inside the C-Callable DPL_CLAInit() function which is defined in {ProjectName}-DPL-CLA.asm

```
;Macro Specific Initialization Functions
PFC_ICMD_CLA_INIT 1     ; PFC_ICMD_CLA Initialization
```

Step 9 Call the run time macro in assembly inside the CLA Task which is defined in {ProjectName}-CLA.asm. This task is the task which is repeatedly called in a loop for the control algorithm.

```
;"Call" the Run macro
PFC_ICMD_CLA 1    ; Run PFC_ICMD_CLA Macro
```

Step 10 **Initialize variable** which is in the CLA writable memory location in CLA Task 8, this task is forced by the CLA_Init() function.

Note variable names in the following step are not associated with the module and can change depending on Step 2, and or the system in consideration.

```
;(Initialize variables in CLA writable memory space )
.ref _Vcmd
.ref _VacRect
.ref _VinvSqr
.ref _CurrCmd
MMOVF32 MR0, #0.0L
MMOV32 @_Vcmd,MR0
MMOV32 @_VacRect,MR0
MMOV32 @_VinvSqr,MR0
MMOV32 @_CurrCmd,MR0
```

Step 11 Include the memory sections the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD` for the macro's net terminals and internal data and allocate them to the desired location.
See **section 3.2** "Memory Requirements in CLA based systems for F28035" for more details.

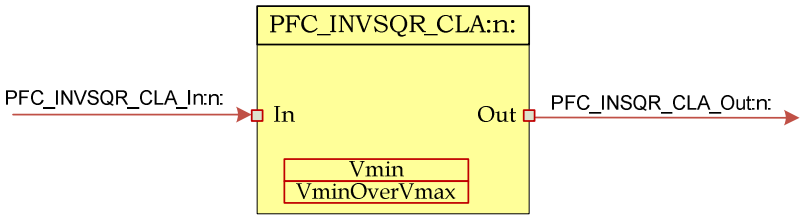
```
/*PFC_ICMD_CLA sections*/
PFC_ICMD_CLA_Section    : > CLA1_MSGRAMHIGH     PAGE = 1
PFC_ICMD_CLA_InternalData     : > CLA1_MSGRAMHIGH     PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description & Memory Restriction**	**Format**	**Acceptable Range**
PFC_ICMD_CLA_Vcmd:n:	Input Pointer (CLA Read)	Pointer to 32 bit fixed point input data location storing the output of the voltage controller	Float: [0, 1)
PFC_ICMD_CLA_VinvSqr:n:	Input Pointer (CLA Read)	Pointer to 32 bit fixed point input data location storing the output of the PFC_INVSQR block	Float: [0, 1)
PFC_ICMD_CLA_VacRect:n:	Input Pointer (CLA Read)	Pointer to 32 bit fixed point input data location storing the output of the MATH_EMAVG block	Float:[0,1)
PFC_ICMD_CLA_Out:n:	Output Pointer (CLA Read)	Pointer to 32 bit fixed point output location where the reference for the current loop is stored	Float:[0,1)
PFC_ICMD_CLA_VmaxOverVmin:n:	Internal Data (CLA Read)	Data Variable storing the scaling factor	Float

PFC_INVSQR_CLA — *Inverse Square Math Block for Power Factor Correction using CLA*

Description: This software module performs a reciprocal function on a scaled unipolar input signal and squares it.

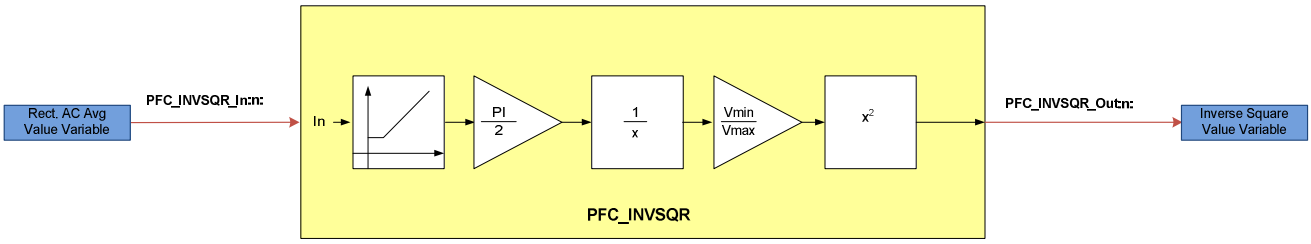


Macro File: `PFC_INVSQR_CLA.asm`

Technical: The input pointer `PFC_INSQR_CLA_In:n:` points to a variable represented in normalized float format. The module scales and inverts this value and writes the result in float to a variable pointed by the output pointer `PFC_INVSQR_CLA_Out:n:` The module uses two internal data variables to specify the range and scaling, which is dependent on the Power Factor Correction stage the module is used for.

A PFC stage is typically designed to work over a range of AC line conditions. `PFC_INVSQR_CLA_VminOverVmax:n:` is the ratio of minimum over maximum voltage the PFC stage is designed for represented in float. The `PFC_INVSQR_CLA_Vmin:n:` is equal or less than the minimum AC Line voltage that the PFC stage is designed represented in a normalized float format. Note that `PFC_INVSQR_CLA_Vmin:n:` depends on what range the voltage feedback in the PFC system is designed for.

The module allows for the fact that the input value is the average of a half-sine (rectified AC), whereas what is desired for power factor correction is the representation of the peak of the sine. In addition the input signal is clamped to a minimum to allow the PFC system to work with very low line voltages without overflows, which can cause undesired effects. The module also saturates the output for a maximum of 1.0. The following diagram illustrates the math function operated on in this block.



Usage: This section explains how to use this module. The example assumes a PFC Stage designed for 230VAC to 90VAC and the voltage feedback is designed for peak 400V. Hence,

```
PFC_INVSQR_CLA_VminOverVmax:n:   =   (float)(90.0/230.0)= 0.3913

PFC_INVSQR_CLA_Vmin:n:           =< (float)(90.0/400.0)= 0.225
```

Step 1 **Add library header file** in the file {ProjectName}-Main.c

```
#include "DPLib.h"
```

Step 2 **Declare the terminal pointers in C** in the file {ProjectName}-Main.c

```
// ---------------------------- DPLIB Net Pointers ---------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PFC_INVSQR - instance #1
extern volatile float *PFC_INVSQR_CLA_In1;
extern volatile float *PFC_INVSQR_CLA_Out1;
extern volatile float PFC_INVSQR_CLA_VminOverVmax1;
extern volatile float PFC_INVSQR_CLA_Vmin1;
```

Step 3 **Declare signal net nodes/ variables in C** in the file {ProjectName}-Main.c, also specify the appropriate memory location variable need using the **#pragma** preprocessor directive. This is determined by whether the CLA reads from the variable or writes to the variable.

Note signal net nodes/variable names are not dependent on macro names and/or macro terminal pointer names, these can change from system to system.

```
// ---------------------------- DPLIB Variables ------------------------

// Declare the net variables being used by the DP Lib Macro here

#pragma DATA_SECTION(In,   "Cla1ToCpuMsgRAM");
#pragma DATA_SECTION(Out,  "Cla1ToCpuMsgRAM");
volatile float In, Out;
```

Please note that the memory region where In and Out are placed depends on what processor is writing to these variables. As the PFC_INVSQR_CLA block is used in the PFC system such that the input is written to by another CLA module, hence In and Out both need to be places in a CLA writable space which can be either the Cla1ToCpuMsgRAM or the CLA data RAM. For convenience example configuration using message RAM is demonstrated in the code snippet.

Step 4 **"Call"** the DPL_CLAInit() to initialize the macros and **"connect"** the module terminals to the signal nets in "C" in {ProjectName}-Main.c

```
//----------Connect the macros to build a system-------------------

// Digital Power (DP) library initialisation
DPL_CLAInit();
// PFC_INVSQR block connections
PFC_INVSQR_CLA_In1=&In;
PFC_INVSQR_CLA_Out1=&Out;
PFC_INVSQR_CLA_VminOverVmax1=(0.3913);
PFC_INVSQR_CLA_Vmin1=(0.225);
```

Step 5 **Call the CLA_Init()** function, this function is provided as a template inside the system software Main.c file and can be modified to suit the needs of the application i.e. change memory allocation, add tasks etc. However Task 8 is reserved to do any initializations that need to be done to use the macro block. This task is software forced one time at the end of this function. How this task is used to initialize the net variables is discussed in **Step 10**.

```
//----------------------Initialize the CLA-------------------------
CLA_Init();
```

Step 6 **Add** the CLA Task Service Routine and Macro Initialization assembly file "{ProjectName}-DPL-CLA.asm" to the project

Step 7 **Include** the Macro's assembly file in the {ProjectName}-DPL-CLA.asm

```
;Include files for the Power Library Macro's being used by the system

.include "PFC_INVSQR_CLA.asm"
```

Step 8 **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-callable DPL_CLAInit() function which is defined in {ProjectName}-DPL-CLA.asm

```
;Macro Specific Initialization Functions
PFC_INVSQR_CLA_INIT 1    ; PFC_INVSQR_CLA Initialization
```

Step 9 **Call the run time macro** in assembly inside the CLA Task which is defined in {ProjectName}-CLA.asm. This task is the task which is repeatedly called in a loop for the control algorithm.

```
;"Call" the Run macro
PFC_INVSQR_CLA 1   ; Run PFC_INVSQR_CLA
```

Step 10 **Initialize variable** which is in the CLA writable memory location in CLA Task 8, this task is forced by the CLA_Init() function.

Note variable names in the following step are not associated with the module and can change depending on Step 2, and or the system in consideration.

```
;(Initialize variables in CLA writable memory space if any)
.ref _In
.ref _Out
MMOVF32 MR0, #0.0L
MMOV16 @_In,MR0
MMOV16 @_Out,MR0
```

Step 11 **Include the memory sections** the {DeviceName}-{RAM/FLASH}-{ProjectName}.CMD for the macro's net terminals and internal data and allocate them to the desired location.
See **section 3.2** "Memory Requirements in CLA based systems for F28035" for more details.

```
/*PFC_INVSQR_CLA sections*/
PFC_INVSQR_CLA_Section  : > CLA1_MSGRAMHIGH     PAGE = 1
PFC_INVSQR_CLA_InternalData   : > CLA1_MSGRAMHIGH     PAGE = 1
```

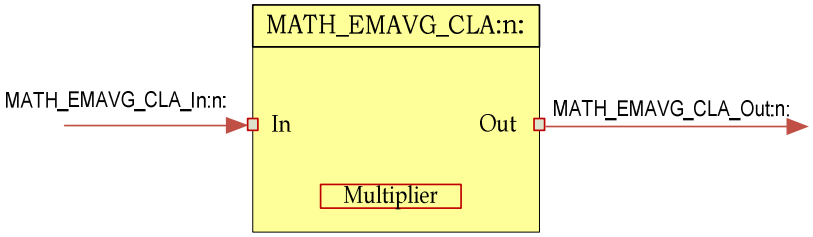
Module Net Definition:

Net name (:n: is the instance number)	**Description & Memory Restrictions**	**Format**	**Acceptable Range**
PFC_INVSQR_CLA_In:n:	Input Pointer (CLA Read)	Pointer to 32 bit fixed point input data location	Float: [0, 1)
PFC_INVSQR_CLA_Out:n:	Output Pointer (CLA Read)	Pointer to 32 bit fixed point data location to write the output	Float: [0, 1)
PFC_INSQR_CLA_VminOverVmax:n:	Internal Data (CLA Read)	Data variable storing scaling information in Q30 format is ratio of the min to max voltage the PFC stage is designed for	Float:[0,1)
PFC_INVSQR_CLA_Vmin:n:	Internal Data (CLA Read)	Data Variable storing information in Q24 format of the ratio of minimum AC line the PFC stage is designed to work for and the max voltage the voltage feedback is designed for	Float:[0,1)

5.4 MATH Blocks

MATH_EMAVG_CLA	*Exponential Moving Average using CLA*

Description: This software module performs exponential moving average



Macro File: `MATH_EMAVG_CLA.asm`

Technical: This software module performs exponential moving average over data stored in float format, pointed to by `MATH_EMAVG_CLA_In:n:` The result is stored in float format at a 32 bit location pointed to by `MATH_EMAVG_CLA_Out:n:`

The math operation performed can be represented in time domain as follows:

$$EMA(n) = (Input(n) - EMA(n-1)) * Multiplier + EMA(n-1)$$

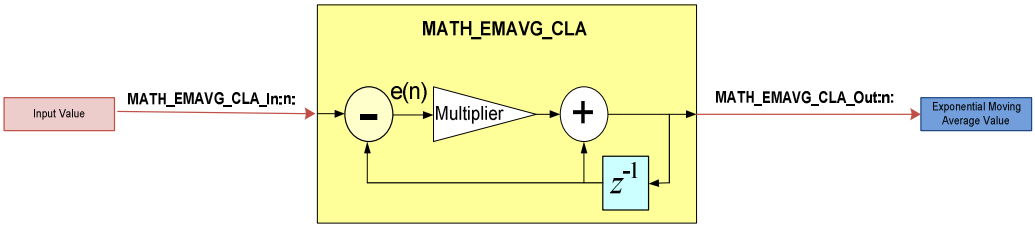
Where $Input(n)$ is the input data at sample instance 'n',
$EMA(n)$ is the exponential moving average at time instance 'n',
$EMA(n-1)$ is the exponential moving average at time 'n-1'.
$Multiplier$ is the weighting factor used in exponential moving average

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 lower the $Multiplier$ value, 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.



Usage: 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

$$Multiplier = 1/SAMPLE\_No = 1/3332 = 0.0003$$

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. 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

$$Multiplier = (2 * \pi * f_{cutt\_off}) / f_{sampling} = (5 * 2 * 3.14)/(100K) = 0.000314$$

Following are the steps to include this module into your system

Step 1 Add library header file in the file `{ProjectName}-Main.c`

```
#include "DPLib.h"
```

Step 2 Declare the terminal pointers in C in the file `{ProjectName}-Main.c`

```
// ---------------------------- DPLIB Net Pointers --------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//MATH_EMAVG_CLA - instance #1
extern volatile float *MATH_EMAVG_CLA_In1;
extern volatile float *MATH_EMAVG_CLA_Out1;
extern volatile float  MATH_EMAVG_CLA_Multiplier1;
```

Step 3 **Declare signal net nodes/ variables in C** in the file `{ProjectName}-Main.c, also` specify the appropriate memory location variable need using the **#pragma** preprocessor directive. This is determined by whether the CLA reads from the variable or writes to the variable.

Note signal net nodes/variable names are not dependent on macro names and/or macro terminal pointer names, these can change from system to system.

```
// ---------------------------- DPLIB Variables ------------------------
// Declare the net variables being used by the DP Lib Macro here
#pragma DATA_SECTION(In, "CpuToCla1MsgRAM");
#pragma DATA_SECTION(Out,"Cla1ToCpuMsgRAM");
volatile float In,Out;
```

Please note that the memory region where In and Out are placed depends on which processor is writing to this value. For Out it would always be a memory region that is CLA writable and whereas In can be in CLA writable or CPU Writable depending on the module is being used in the system.

Step 4 **"Call"** the `DPL_CLAInit()` to initialize the macros and **"connect"** the module terminals to the signal nets in "C" in `{ProjectName}-Main.c`

```
//----------Connect the macros to build a system-------------------
// Digital Power (DP) library initialization
DPL_CLAInit();
// MATH_EMAVG block connections
MATH_EMAVG_CLA_In1=&In;
MATH_EMAVG_CLA_Out1=&Out;
MATH_EMAVG_CLA_Multilpier1=(float) (0.00025);

// Initialize the net variables present in CpuToCLA memory
In=(float)(0.0);
```

Step 5 Call the CLA_Init() function, this function is provided as a template inside the system software Main.c file and can be modified to suit the needs of the application i.e. change memory allocation, add tasks etc. However Task 8 is reserved to do any initializations that need to be done to use the macro block. This task is software forced one time at the end of this function. How this task is used to initialize the net variables is discussed in **Step 9**.

```
//----------------------Initialize the CLA-------------------------
CLA_Init();
```

Step 6 **Add** the CLA Task Service Routine and Macro Initialization assembly file "{ProjectName}-DPL-CLA.asm" to the project

Step 7 **Include** the Macro's assembly file in the {ProjectName}-CLA.asm

```
;Include files for the Power Library Macro's being used by the system

.include "MATH_EMAVG_CLA.asm"
```

Step 8 **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-Callable DPL_CLAInit() function which is defined in {ProjectName}-DPL-CLA.asm

```
;Macro Specific Initialization Functions
MATH_EMAVG_CLA_INIT 1    ; MATH_EMAVG_CLA Initialization
```

Step 9 **Call the run time macro** in assembly inside the CLA Task which is defined in {ProjectName}-CLA.asm. This task is repeatedly called in a loop for the control algorithm.

```
;"Call" the Run macro
MATH_EMAVG_CLA 1   ; Run MATH_EMAVG_CLA
```

Step 10 **Initialize variable** which is in the CLA writable memory location in CLA Task 8, this task is forced by the CLA_Init() function.

Note variable names in the following step are not associated with the module and can change depending on Step 2, and or the system in consideration.

```
;(Initialize variables in CLA writable memory space )
.ref _Out
MMOVF32 MR0, #0.0L
MMOV16 @_Out,MR0
```

Step 11 **Include the memory sections** the {DeviceName}-{RAM/FLASH}-{ProjectName}.CMD for the macro's net terminals and internal data and allocate them to the desired location.
See **section 3.2** "Memory Requirements in CLA based systems for F28035" for more details.

```
/*MATH_EMAVG_CLA Sections */
MATH_EMAVG_CLA_Section        : > CLA1_MSGRAMHIGH     PAGE = 1
MATH_EMAVG_CLA_InternalData   : > CLA1_MSGRAMHIGH     PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description & Memory Restriction**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
MATH_EMAVG_CLA_In:n:	Input Pointer (CLA Read)	Pointer to 32 bit fixed input data location storing the data that needs to averaged	Float: [0, 1)
MATH_EMAVG_CLA_Out:n:	Output Pointer (CLA Read)	Pointer to 32 bit fixed output data location where the computed average is stored	Float: [0, 1)
MATH_EMAVG_CLA_Multiplier:n:	Internal Data (CLA Read)	Data Variable storing the weighing factor for the exponential average.	float

Chapter 6. Revision History

Version	Date	Notes
V2.0	July 6, 2010	Major release of library to support Piccolo platform.
V3.0	October 2010	Major Release as DPLib for CLA is moved to float math from Q24 math, for more efficient operation of the CLA. Fixes to DPlibv2 1. Period value corrected for the PWMDRV_PFC2PhiL, PWMDRV_1ch, PWMDRV_1chHiRes, PWMDRV_DualUpDownCnt, PWMDRV_ComplPairDB, macro 2. Input Name changed from In to Duty for PWMDRV_1ch, PWMDRV_1chHiRes, PWMDRV_ComplPairDB