CLA C Digital Power Library

v3.4

Oct-12

Module User's Guide

CLA Foundation Software

TEXAS INSTRUMENTS

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.

CPU: Refers to the main processing unit present in a device.

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

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 form a system.

ClaToCpu_volatile: Data type primarily used for variables that are accessed across the CLA and the CPU; variables that reside in CLA-CPU message RAMs or CLA writable space.

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

Index

Chapter 1. Introduction

1.1. Introduction

Texas Instruments Control Law Accelerator Digital Power Library (CLA DPlib) is designed to enable flexible and efficient coding of digital power supply applications using the CLA co-processor present in some C2000 devices. An important consideration for power supply applications 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 that 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 frequency control loops. This allows CLA to be ideal for many power supply applications. As the CLA is a co-processor to the main CPU 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.

CLA DPlib provides a software structure that is flexible, adaptable and completely configurable. The 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 CLA DPlib and the C28x DPlib is memory allocation for 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 CLA DPlib modules to form a closed loop system is shown below.

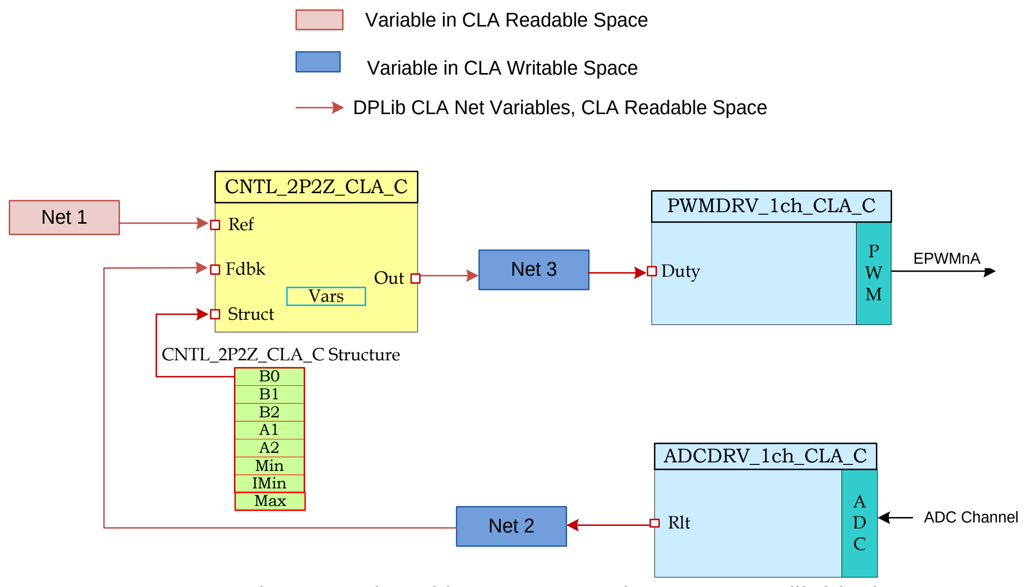


Figure 1: Closed loop system using CLA C DPlib blocks

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 C variables 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, 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 configuration prior to use, the initialization and configuration process is described in Chapter 3.

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

Once the blocks have been initialized and connected, they can 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 C Library

2.1. DPCLA Library Package Contents

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

- C configuration files
- C header macro files
- An example CCS project showing the connection and use of CLA DPlib blocks.
- Documentation

2.2. How to Install the CLA Digital Power Library

The CLA DPlib 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 CPU platform.

The following sub-directory structure is used:

<base>\ C_macros	Contains implementation of the DP Library for the Control Law Accelerator using the CLA C Compiler
<base>\Doc	Contains this file
<base>\CNF	C-language initialization/configuration files
<base>\include	contains DPlib header file

The installation also installs a template project using the CLA DPlib for the device inside the controlSUITE directory:

controlSUITE\development_kits\TemplateProjects\DPLibv3_4Template-F2803x

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 CLA DPlib module is contained in a separate include file. An example of the naming convention used is shown below:

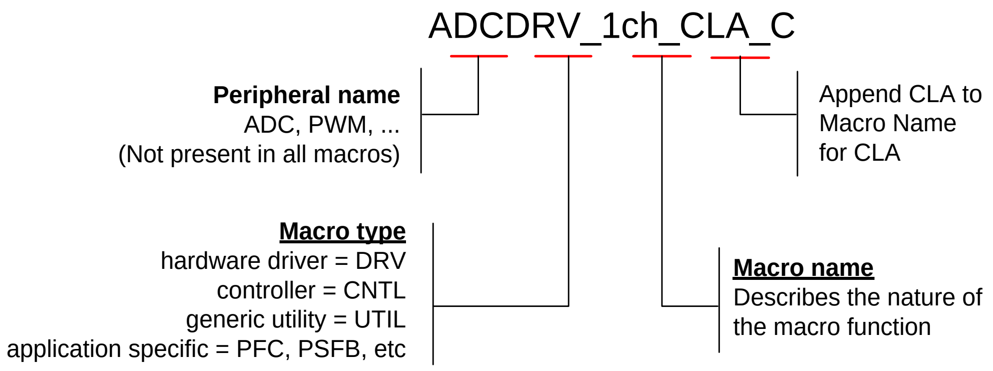


Figure 2: Function naming convention

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

Include file:	`ADCDRV_CLA_C.h`
Macro call:	`ADCDRV_1ch_CLA_C()`

Chapter 3. Using CLA Digital Power Library

3.1. Library Description and Overview

Typical CLA software will consist of a main framework file written in C and a single CLA Task file. The C framework contains code to configure the device hardware, the CLA, peripheral interrupts and CLA-Task triggers from the PWMs or ADCs. Once CLA is configured the CLA operates independently of the CPU core, thus freeing up main CPU to do other tasks.

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

1. **Initialization** – Hardware initialization functions are called from the C environment using a C callable function `DPL_CLAInit()` which should be defined in the main C file. The function is prototyped in the library header file `DPlib.h` that must be included in the main C file.

2. **Configuration** – Some DPlib modules require additional variables, thus memory for the variables must be allocated within CLA accessible memory. Care should be taken to ensure variables are stored in the proper CLA message RAM or data RAM. If variables are to be accessed by the main CPU, variable prototypes should also be included in the `{ProjectName}-CLA_Shared.h` header file.

3. **Execution** – Macro-code is executed in CLA tasks. The eight task functions must be defined in the `{ProjectName}-CLA_Tasks.cla` file.

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

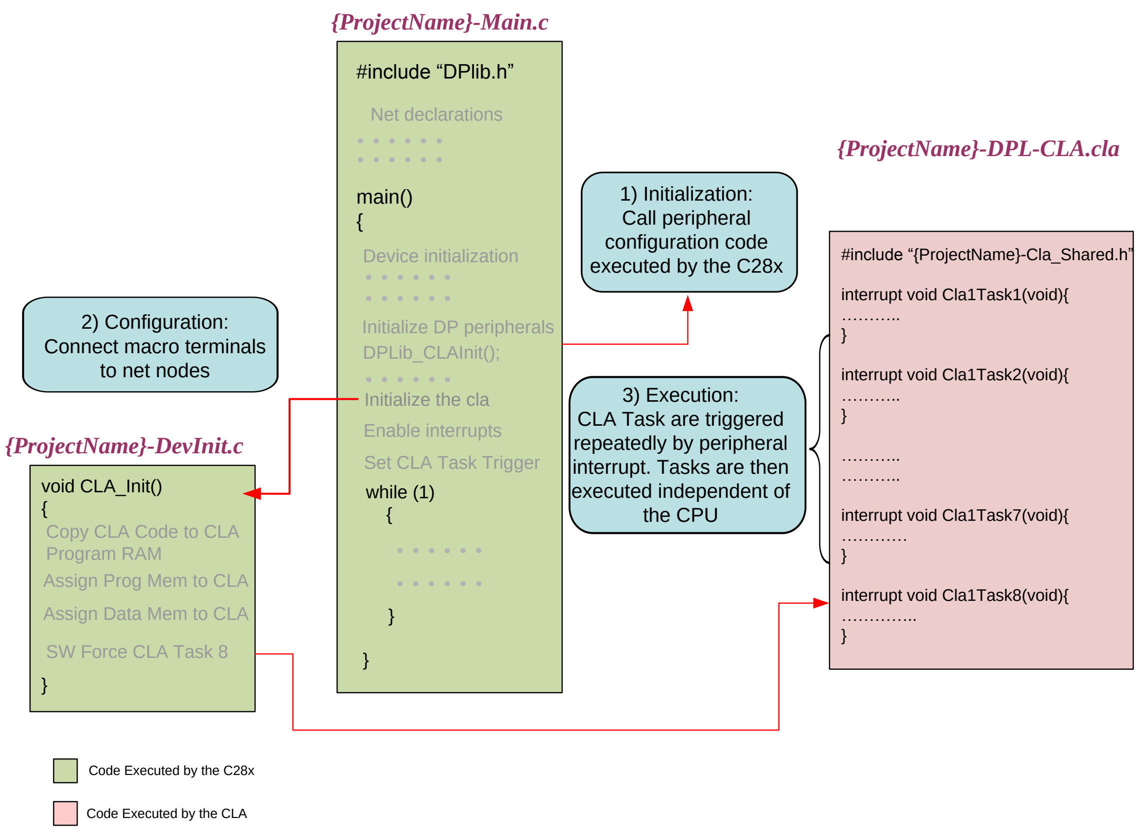

{ProjectName}-Main.c
#include "DPlib.h"
Net declarations
main()
{
Device initialization
Initialize DP peripherals
DPLib_CLAInit();
Initialize the cla
Enable interrupts
Set CLA Task Trigger
while (1)
{
}
}
1) Initialization: Call peripheral configuration code executed by the C28x
2) Configuration: Connect macro terminals to net nodes
3) Execution: CLA Task are triggered repeatedly by peripheral interrupt. Tasks are then executed independent of the CPU
{ProjectName}-DevInit.c
void CLA_Init()
{
Copy CLA Code to CLA Program RAM
Assign Prog Mem to CLA
Assign Data Mem to CLA
SW Force CLA Task 8
}
{ProjectName}-DPL-CLA.cla
#include "{ProjectName}-Cla_Shared.h"
interrupt void Cla1Task1(void){
interrupt void Cla1Task2(void){
interrupt void Cla1Task7(void){
interrupt void Cla1Task8(void){
Code Executed by the C28x
Code Executed by the CLA

Figure 3: Relation between main.c and CLA_Tasks.cla files

The CLA DPlib code structure has been designed to allow the user to freely specify the interconnection between blocks while maintaining a high degree of code efficiency.

To initialize the hardware, edit the `DPL_CLAInit()` function; add function calls to configure the hardware as required for each macro in the application. The order of the calls is not important, provided all functions required for the application are run. All macro header files are included in the `DPlib.h` file; required file includes should be uncommented as necessary. Note: the CPU executes this section of the code.

The internal layout and relationship between the `{ProjectName}-CLA_Tasks.cla` file and the various macro files is shown diagrammatically below. In this example, three CLA DPlib macros are being used. Each library module is contained in a header include file (.h extension) which contains the code.

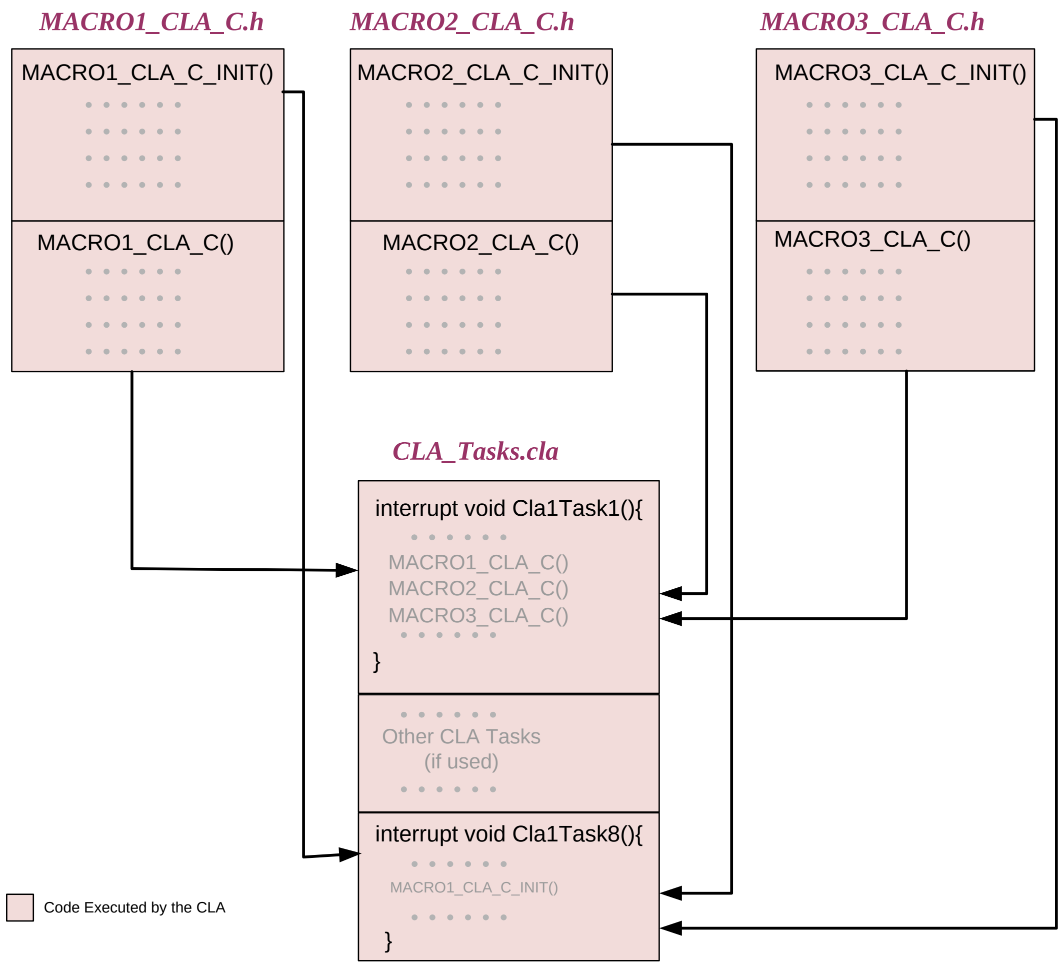


Figure 4: CLA tasks & DPlib macro files

Note: As one task runs at a time and is uninterruptable. There is no need for context save when a task is triggered, 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 CPU. As the internal memory available on the devices is fixed, the memory needs to be shared between the CPU 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 CLA DPlib.

3.2.1 Memory Allocation on F28035

The F38035 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 the 4K RAM and assigning the memory to the CLA. By default this memory is assigned to the CPU and if the CLA is not being used the CPU 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 the CLA DPlib.

```
/* Copy the CLA program code from its load address to the CLA program memory.
*/
memcpy(&Cla1funcsRunStart, &Cla1funcsLoadStart, (Uint32) &Cla1funcsLoadSize);
asm("   RPT #3 || NOP");

/* 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 the RAM as CLA program memory
```

Data Memory: The CLA is a slave to the CPU core, thus when running a control algorithm the CPU may need to perform diagnostic functions & provide values to 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. **Data RAM:** 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 CPU 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 CPU. The mapping of the data RAM to CLA occurs in the `CLA_Init()` function which is defined in `{ProjectName}-DevInit.c` as shown below. The user can uncomment 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;
```

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_C block, the CPU 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 CPU to the CLA. An example of this is the CPU 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 stored in the data RAM also.

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 CLA DPlib components.

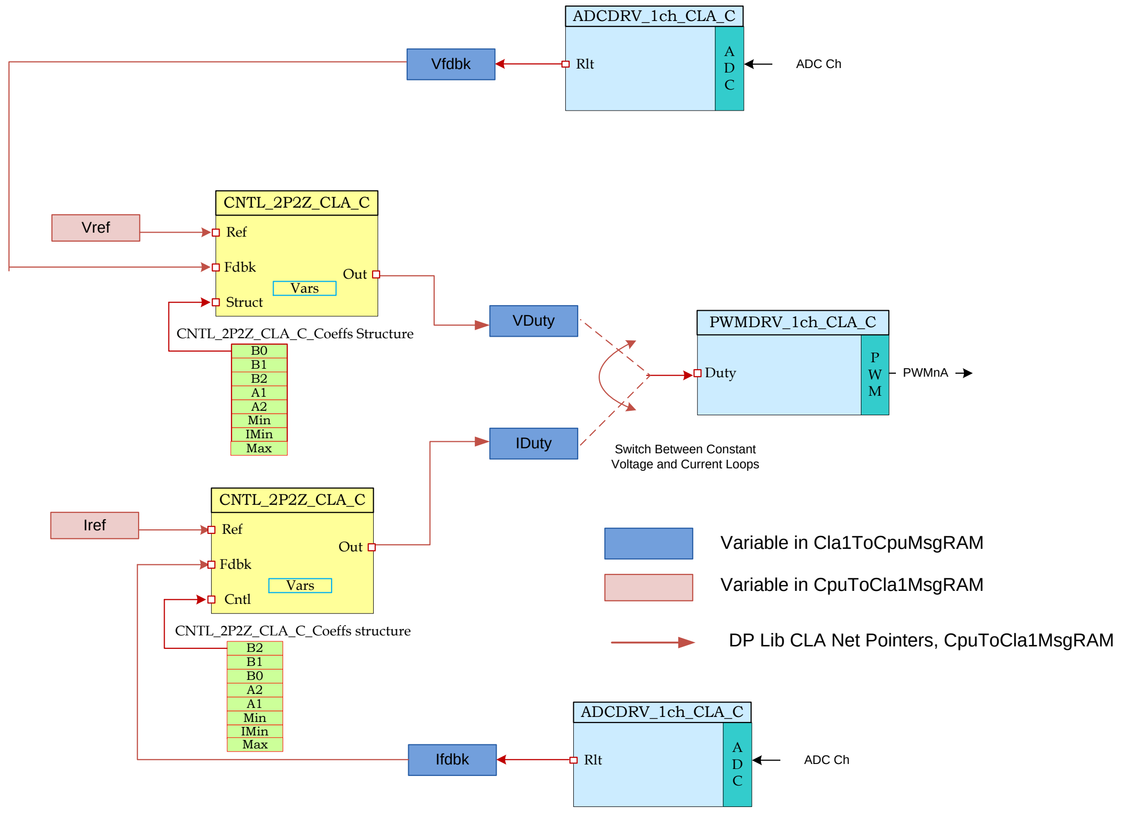


Figure5: Example system with default memory configuration

The variables are color coded to reflect the memory requirement. In the system the CPU provides the references for the voltage and the current loops 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) Configuration
By default the following memory configuration can be used for the example system.

Note: Declarations for variables allocated to message RAMs should have the `ClaToCpu_volatile` data type. Variables within message RAMs can change arbitrarily; using the keyword will ensure the appropriate value is always read. It also allows the compiler to optimize code more efficiently.

Net variables - Assign the net variable depending on the memory constraints to the message RAMs. 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 CPU provides the Vref and the Iref; hence these are placed in the CpuToCla1MsgRAM. The ADCDRV_CLA_C and CNTL_2P2Z_CLA_C blocks compute the Vfdbk, Ifdbk, VDuty and IDuty; hence are placed in the Cla1ToCpuMsgRAM.

Note: A declaration with the **extern** keyword should be added to the `{ProjectName}-CLA_Shared.h` header file. This ensures the CLA and the CPU can access the variables.

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 RAMs for the CLA. The following section illustrates how the data RAMs 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 variables - The CPU has access to the data RAMs by default, and hence can do assignment to variables present in the data RAMs before `CLA_Init()` is called. 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, the variables should be kept in the message RAMs.

The following code snippet places net terminals for the above example system, while offloading some of the data to the data RAM instead of the message RAMs. The coefficients are kept in the CpuToCla1MsgRAM; this allows the CPU to change the coefficients depending on the system condition at run time by the CPU. However as discussed net terminals that may change at run time are placed in the CpuToCla1MsgRAM.

Net variables - Assign the net variable depending on the memory constraints to the message RAMs or data RAM. 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, "Cla1DataRam1")
#pragma DATA_SECTION(IDuty, "Cla1DataRam1")
```

The CPU provides the Vref and the Iref; hence these variables are placed in the CpuToCla1MsgRAM. The ADCDRV_CLA_C blocks compute the Vfdbk and Ifdbk; these values need to be monitored by the CPU. Thus these net variables are placed in the Cla1ToCpuMsgRAM. The CNTL_2P2Z_CLA_C blocks compute VDuty and IDuty; the CPU does not have to monitor these, thus are placed in the data RAM.

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 CLA DPlib 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. It should also identify 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, variable declarations and module includes 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 ***Include the required macro header files***. Un-comment header file includes at the top of the `DPlib.h` file as required. Only one file include is required for each block type used in the project.

Step 3 ***Declare variables in C***. The `{ProjectName}-CLA_Shared.h` file needs to be edited to add extern declarations to all the macro variables, which will be needed in the application under the "DPlib Variables" section inside this file. In the example below net pointers to an instance of the 2P2Z control block are referenced.

```
/* DPlib Variables*/
extern ClaToCpu_volatile CNTL_2P2Z_CLA_C_Coeffs cntl_2p2z_coeffs;
extern ClaToCpu_volatile CNTL_2P2Z_CLA_C_Vars cntl_2p2z_2_vars;
```

Step 4 ***Declare variables in C and provide compiler directive for memory placement***. Edit the `{ProjectName}-CLA_Tasks.cla` file to define the net variables that will be needed in the CLA. Specify the appropriate memory location for variables using the **#pragma** pre-processor directive. The memory location is determined by whether the CLA reads from the variable or writes to the variable.

```
/* DPlib Variables */
#pragma DATA_SECTION(VDuty, "Cla1DataRam1");
#pragma DATA_SECTION(IDuty, "Cla1DataRam1");
```

Step 5 ***Call the initialization function from C***. Call the initialization function from the C framework using the syntax below. Edit the `DPL_CLAInit()` function; it should have all the DPlib C-function calls for initializing hardware peripherals.

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

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. Additionally, it assigns data ram to the CLA if needed (commented by default), defines the vector addresses for the different CLA Tasks and sets task-triggers. This function also triggers Task 8 by software.

```
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

   EALLOW;
   // Assign PIE interrupts
   PieVectTable.CLA1_INT1 = &cla1_task1_isr;
   PieVectTable.CLA1_INT2 = &cla1_task2_isr;
   PieVectTable.CLA1_INT3 = &cla1_task3_isr;
   PieVectTable.CLA1_INT4 = &cla1_task4_isr;
   PieVectTable.CLA1_INT5 = &cla1_task5_isr;
   PieVectTable.CLA1_INT6 = &cla1_task6_isr;
   PieVectTable.CLA1_INT7 = &cla1_task7_isr;
   PieVectTable.CLA1_INT8 = &cla1_task8_isr;

   // Copy CLA program code to CLA program memory & assign it to the CLA
   // Ensure at least two SYSCLKOUT cycles before assigning interrupts
   memcpy(&Cla1funcsRunStart, &Cla1funcsLoadStart,
   (Uint32) &Cla1funcsLoadSize);
   asm("   RPT #3 || NOP");

   // Symbols used in calculation are defined in CLA shared header file
   Cla1Regs.MVECT1 = ((Uint16) Cla1Task1 - (Uint16) &Cla1Prog_Start);
   Cla1Regs.MVECT2 = ((Uint16) Cla1Task2 - (Uint16) &Cla1Prog_Start);
   Cla1Regs.MVECT3 = ((Uint16) Cla1Task3 - (Uint16) &Cla1Prog_Start);
   Cla1Regs.MVECT4 = ((Uint16) Cla1Task4 - (Uint16) &Cla1Prog_Start);
   Cla1Regs.MVECT5 = ((Uint16) Cla1Task5 - (Uint16) &Cla1Prog_Start);
   Cla1Regs.MVECT6 = ((Uint16) Cla1Task6 - (Uint16) &Cla1Prog_Start);
   Cla1Regs.MVECT7 = ((Uint16) Cla1Task7 - (Uint16) &Cla1Prog_Start);
   Cla1Regs.MVECT8 = ((Uint16) Cla1Task8 - (Uint16) &Cla1Prog_Start);

   // Enable interrupt routines.
   Cla1Regs.MIER.all = 0x00FF;
   Cla1Regs.MMEMCFG.bit.PROGE = 1; // Configure the RAM as CLA program memory
   Cla1Regs.MMEMCFG.bit.RAM0E = 1; // Configure RAM L1 as CLA Data RAM 0
   Cla1Regs.MMEMCFG.bit.RAM1E = 1; // Configure RAM L2 as CLA Data RAM 1

   // Enable the IACK instruction to start a task
   __asm("   RPT #3 || NOP");
   Cla1Regs.MCTL.bit.IACKE = 1;
   Cla1Regs.MIER.all = M_INT8;   // Trigger Task 8
   EDIS;
}
```

Step 7 ***Add CLA Task Service Routine file***. File where the CLA runtime code is assigned to one of 8 CLA Task Service Routines. A blank template of this file, `{ProjectName}-CLA_Tasks.cla`, is included with the CLA DPlib in the template project. Note: Task 8 is reserved for initializing macro blocks. Task 8 is triggered by software a

Step 8 ***Edit the CLA Tasks to execute macros as required by the application***. In the example below the first instance of a 2P2Z control macro would be executed whenever the CLA Task1 is triggered:

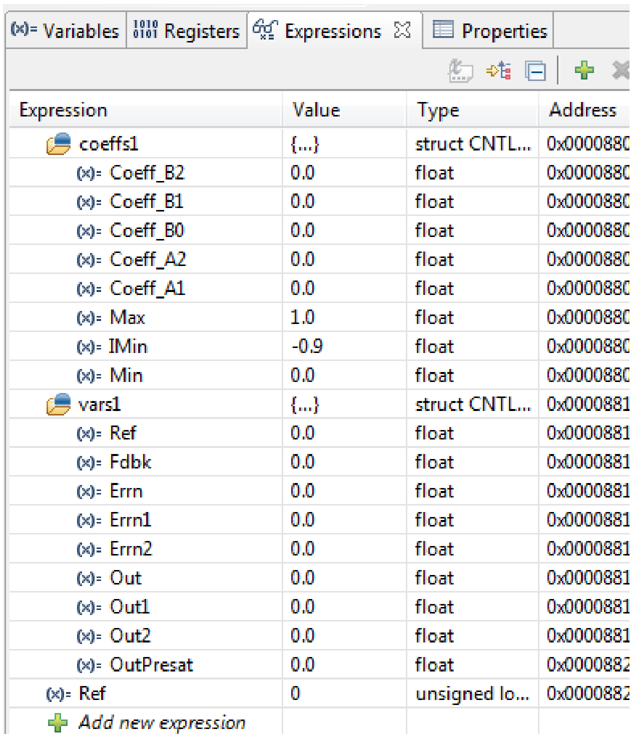
```
interrupt void Cla1Task1(void) {
      ...
      vars1.Ref = Vref;
      vars1.Fdbk = Vfdbk;
      CNTL_2P2Z_2Structs_CLA_C_INIT(coeffs, vars);
      ...
}
```

Step 9 ***Mapping Peripheral Interrupt to the CLA Task***. Peripheral interrupts can to be mapped to the CLA, this is done once all the configuration of the CLA is complete. The assignment code should be 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;
      ...

      // Enable Tasks to be triggered by peripheral interrupts.
      Cla1Regs.MIER.all = M_INT1;
      __asm("   RPT #3 || NOP");
      EDIS;
```

3.4. Viewing CLA DPlib variables in watch window

If desired, the DP CLA library macro variables can be seen in real time by adding them to the Expressions Window within Code Composer Studio. Shown below is the value stored in the net variable Ref and the CNTL_2P2Z_CLA_C structures.



Expression	Value	Type	Address
coeffs1	{...}	struct CNTL...	0x0000880
Coeff_B2	0.0	float	0x0000880
Coeff_B1	0.0	float	0x0000880
Coeff_B0	0.0	float	0x0000880
Coeff_A2	0.0	float	0x0000880
Coeff_A1	0.0	float	0x0000880
Max	1.0	float	0x0000880
IMin	-0.9	float	0x0000880
Min	0.0	float	0x0000880
vars1	{...}	struct CNTL...	0x0000881
Ref	0.0	float	0x0000881
Fdbk	0.0	float	0x0000881
Errn	0.0	float	0x0000881
Errn1	0.0	float	0x0000881
Errn2	0.0	float	0x0000881
Out	0.0	float	0x0000881
Out1	0.0	float	0x0000881
Out2	0.0	float	0x0000881
OutPresat	0.0	float	0x0000882
Ref	0	unsigned lo...	0x0000882
Add new expression			

Chapter 4. Module Summary

4.1. DP CLA C Library Function Summary

The Digital Power CLA C 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, required configuration files, a summary of cycle counts, use of C structures and use of CLA C intrinsic functions.

Module Name	Type	Description	HW Config File	Cycles	Cycles for Optimized Code	CLA Intrinsic	Multiple Instance Support
ADCDRV_CLA_C	HW	ADC driver	Yes	3	3	No	Yes
ADCDRV_4ch_CLA_C*	HW	4 channel ADC driver	Yes	28	28	No	Yes
ADCDRV_8ch_CLA_C*	HW	8 channel ADC driver	Yes	56	56	No	Yes
CNTL_2P2Z_CLA_C*	CNTL	Second order control law	N/A	54	39	No	Yes
CNTL_3P3Z_CLA_C*	CNTL	Third order control law	N/A	60	52	No	Yes
DACDRV_RAMP_CLA_C	HW	DAC driver for slope compensation	Yes	20	20	No	Yes
DLOG_1ch_CLA_C*	UTIL	1 channel data logger module	N/A	83	82	No	Yes
DLOW_4ch_CLA_C*	UTIL	4 channel data logger module	N/A	126	122	No	Yes
MATH_EMAVG_CLA_C*	MATH	Exponential moving average module	N/A	10	7	No	Yes
PFC_BL_ICMD_CLA_C*	APPL	Power factor correction current command block	N/A	47	27	Yes	Yes
PFC_ICMD_CLA_C*	APPL	Power factor correction current command block	N/A	16	11	No	Yes
PFC_InvRmsSqr_CLA_C	APPL	Power Factor Correction Inverse Square Block	N/A	20	11	Yes	Yes
PFC_INVSQR_CLA_C*	APPL	Power factor correction inverse square block	N/A	25	12	Yes	Yes
PWMDRV_1ch_CLA_C	HW	Single channel PWM driver	Yes	14	14	No	Yes
PWMDRV_1ch_UpDwnCnt_CLA_C	HW	Single channel driver with up down modulation	Yes	5	5	No	Yes

PWMDRV_1ch_UpDwnCntCompl_CLA_C	HW	Single channel driver with up down modulation centered around period	Yes	6	6	No	Yes
PWMDRV_1chHiRes_CLA_C	HW	Single channel PWM driver with HiRes capability	Yes	15	15	No	Yes
PWMDRV_1chHiRes_UpDwnCnt_CLA_C	HW	Single channel PWM driver with HiRes capability	Yes	14	14	No	Yes
PWMDRV_2ch_UpCnt_CLA_C	HW	PWM driver with independent duty control on ch A and ch B, using up down count mode	Yes	12	12	No	Yes
PWMDRV_BuckBoost_CLA_C	HW	PWM driver for a four switch Buck Boost stage	Yes	21	21	No	Yes
PWMDRV_BuckBoost_CLA_C_UpdateDB	HW	Update dead band of Buck Boost stage	Yes	6	6	No	Yes
PWMDRV_ComplPairDB_CLA_C	HW	PWM driver for complimentary pair PWM	Yes	14	14	No	Yes
PWMDRV_ComplPairDB_CLA_C_UpdateDB	HW	Update complementary PWM dead band	Yes	6	6	No	Yes
PWMDRV_DualUpDwnCnt_CLA_C	HW	PWM driver with independent duty control on ch A and B	Yes	13	13	No	Yes
PWMDRV_LLC_1ch_UpCntDB_CLA_C	HW	Driver with frequency modulation & rising/falling-edge adjustment via DB registers	Yes	22	22	No	Yes
PWMDRV_LLC_1ch_UpCntDB_CLA_C_UpdateDB	HW	Updates 1ch LLC dead band	Yes	6	6	No	Yes
PWMDRV_LLC_1ch_UpCntDB_Compl_CLA_C	HW	Driver with frequency modulation and rising/falling-edge adjustment via DB registers	Yes	38	38	No	Yes
PWMDRV_LLC_1ch_UpCntDB_Compl_CLA_C_UpdateDB	HW	Dead band adjustment for LLC 1ch UpCntDB Compl	Yes	6	6	No	Yes
PWMDRV_LLC_ComplPairDB_CLA_C	HW	PWM driver for LLC	Yes	23	23	No	Yes
PWMDRV_LLC_ComplPairDB_CLA_C_UpdateDB	HW	Update PWM dead band for LLC ComplPair DB	Yes	6	6	No	Yes

PWMDRV_PFC2PhiL_CLA_C	HW	PWM driver for Two Phase Interleaved PFC stage	Yes	15	15	No	Yes
PWMDRV_PSFB_CLA_C	HW	PWM driver for Phase Shifted Full Bridge Power stage	Yes	24	24	No	Yes
PWMDRV_PSFB_VMC_SR_CLA_C*	HW	PWMDRV for PSFB PCMC with SR	Yes	63	38	No	Yes
SineAnalyzer_CLA_C*	UTIL	Calculates wave RMS, average and frequency.	N/A	76	58	Yes	Yes

Note: All code was profiled running from zero wait-state memory.

* Requires the use of C structure to store internal variables. Each module has additional code to initialize respective structures.

Note: The following compilation directives were used when profiling library modules:

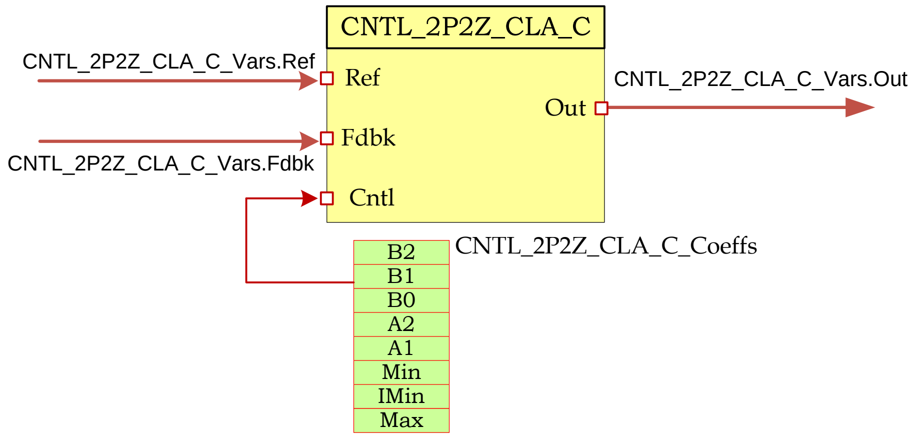
- Compiler optimization level 4 (-O 4)
- Optimize for space (-ms)

Chapter 5. DP CLA C Module Descriptions

5.1. Controllers

CNTL_2P2Z_CLA_C ***Two Pole Two Zero Controller using CLA C***

Description: This 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_C.h`

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…

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

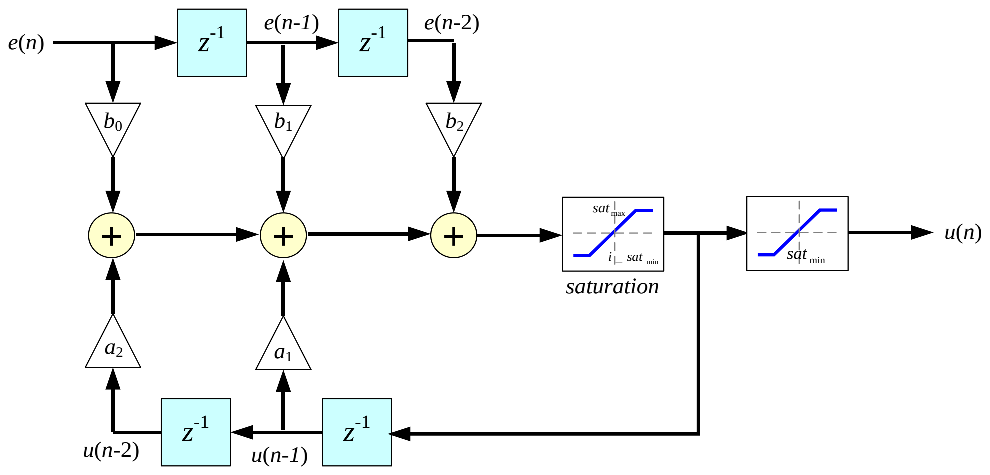
This may be expressed in difference equation form as:

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

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 is stored in memory within a CNTL_2P2Z_CLA_C_Vars structure as follows:

CNTL_2P2Z_CLA_C_Vars

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 stored in memory within a CNTL_2P2Z_CLA_C_Coeffs structure, as follows:

CNTL_2P2Z_CLA_C_Coeffs

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

sat$_{max}$ and *sat*$_{min}$ are the upper and lower control effort bounds. *i_sat*$_{min}$ is the value used for saturating the lower bound of the control effort when storing the history of the output. This allows the value of the history to have negative values which can help avoid output oscillations when no load is present.

Note: Controller coefficients must be initialized before the controller is used.

Usage: This section explains how to use the CNTL_2P2Z_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `CNTL_2P2Z_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare CNTL_2P2Z_CLA_C structures in the `{ProjectName}-CLA_Tasks.cla` file. Specify structure variables and CLA memory locations.

```
/* DPlib variables & memory locations */
#pragma DATA_SECTION(coeffs, "CpuToCla1MsgRAM")
CNTL_2P2Z_CLA_C_Coeffs coeffs;
#pragma DATA_SECTION(vars, "Cla1ToCpuMsgRAM")
CNTL_2P2Z_CLA_C_Vars vars;
```

Step 3 Add shared declaration to `{ProjectName}-CLA_Shared.h` if access is shared between CPU and CLA.

```
/* DPlib variables */
// Declare net variables shared between CPU and CLA here
extern volatile CNTL_2P2Z_CLA_C_Coeffs coeffs;
extern volatile CNTL_2P2Z_CLA_C_Vars vars;
```

Step 4 Add macro initialization to CLA Task 8. Edit the task found in the `{ProjectName}-CLA_Tasks.cla` file to initialize the macro coefficients.

```
interrupt void Cla1Task8(void) {
	...
	CNTL_2P2Z_CLA_C_INIT(coeffs, vars);
	coeffs.Coeff_B2 = 5;
	coeffs.Min = 10;
	coeffs.IMin = 11;
	coeffs.Max = 12;
	...
}
```

Step 5 Connect the CNTL_2P2Z_CLA_C controller within a CLA-Task.

```
interrupt void Cla1Task4(void) {
	...
	vars.Ref = Ref;
	vars.Fdbk = Fbk;
	CNTL_2P2Z_CLA_C(coeffs, vars);
	Out = vars.Out;
	...
}
```

Step 6 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
       ...
       PWM_1ch_CNF(pwm, period, mode, phase);
       ...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Object Definition:

```
/**
 * Two-pole two-zero controller structures.
 */
typedef struct {
       // Coefficients
       float32 Coeff_B2;
       float32 Coeff_B1;
       float32 Coeff_B0;
       float32 Coeff_A2;
       float32 Coeff_A1;

       // Output saturation limits
       float32 Max;
       float32 IMin;
       float32 Min;
} CNTL_2P2Z_CLA_C_Coeffs;

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

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

       // Output values
       float32 Out;
       float32 Out1;
       float32 Out2;
       float32 OutPresat;
} CNTL_2P2Z_CLA_C_Vars;
```

Special Constants and Data types

CNTL_2P2Z_CLA_C_Coeffs

Structure defined to store the controller coefficients and internal limits. To create multiple instances of the module simply declare variables of type CNTL_2P2Z_CLA_C_Coeffs.

CNTL_2P2Z_CLA_C_Vars

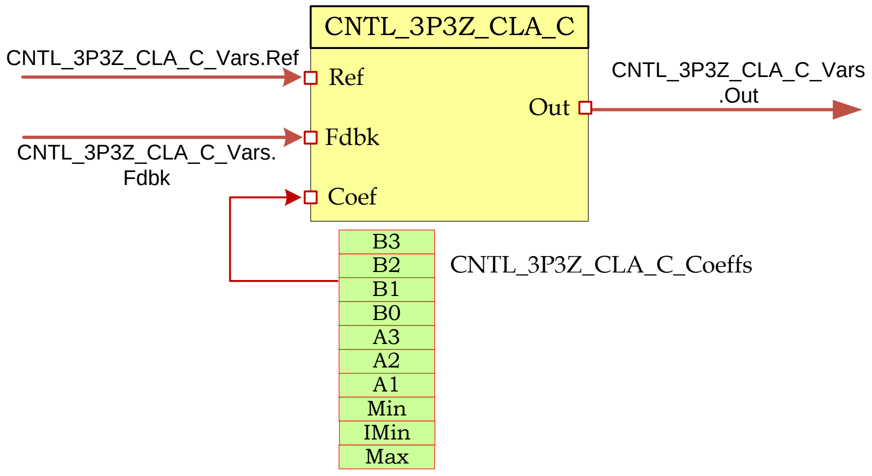
Structure defined to store the internal values and output. To create multiple instances of the module simply declare variables of type CNTL_2P2Z_CLA_C_Vars.

Module Structure Definition:

Parameter Name	Types	Description	Acceptable Range
Ref	Input	Reference value for the controller.	float32 [0, 1)
Fdbk	Input	Feedback value for the controller.	float32 [0, 1)
Coeff_B0, B1, B2, A1, A2	Input	Controller coefficients.	float32
Out	Output	Controller calculated output.	float32 [0,1)
Min, Max, IMin	Input	Internal saturation limits.	float32

CNTL_3P3Z_CLA_C — *Three Pole Three Zero Controller using CLA*

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



Macro File: `CNTL_3P3Z_CLA_C.h`

Module Description: The 3-pole 3-zero control block implements a third order control law using an IIR filter structure with programmable output saturation. This type of controller requires 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…

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

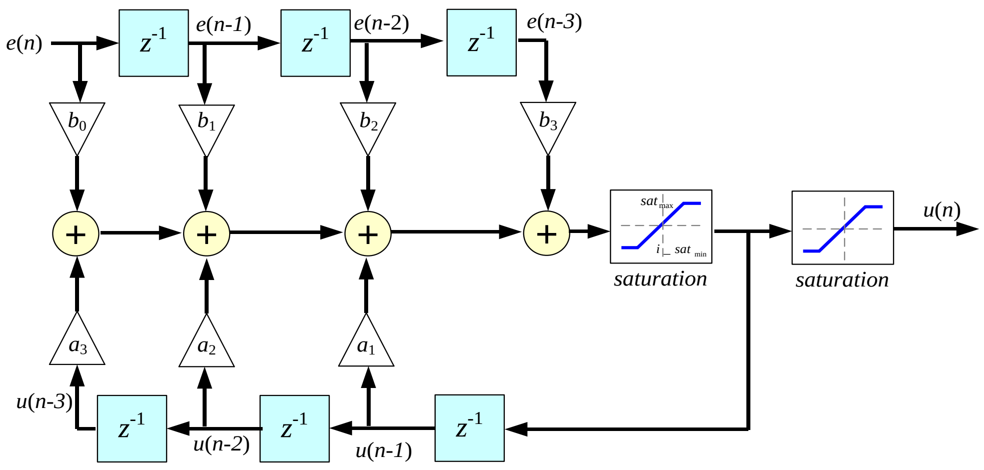
This may be expressed in difference equation form as:

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

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 is stored in memory within a CNTL_3P3Z_CLA_C_Vars structure as shown below.

CNTL_3P3Z_CLA_C_Vars

0	$u(n\text{-}1)$
2	$u(n\text{-}2)$
4	u(n-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 stored in memory within a CNTL3P3Z_CLA_C_Coeffs structure as shown below.

CNTL_3P3Z_CLA_C_Coeffs

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_{IMin})
float(sat_{Min})

sat$_{max}$ and *sat*$_{min}$ are the upper and lower control effort bounds. *i_sat*$_{min}$ is the value used for saturating the lower bound of the control effort when storing the history of the output. This allows the value of the history to have negative values which can help avoid output oscillations when no load is present.

Usage: This section explains how to use the CNTL_3P3Z_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `CNTL_3P3Z_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare CNTL_3P3Z_CLA_C structures in the `{ProjectName}-CLA_Tasks.cla` file. Specify structure variables and CLA memory locations.

```
/* DPlib variables & memory locations */
#pragma DATA_SECTION(coeffs, "CpuToCla1MsgRAM")
CNTL_3P3Z_CLA_C_Coeffs coeffs;
#pragma DATA_SECTION(vars, "Cla1ToCpuMsgRAM")
CNTL_3P3Z_CLA_C_Vars vars;
```

Step 3 Add shared declaration to `{ProjectName}-CLA_Shared.h` file if access is shared between CPU and CLA.

```
/* DPlib variables */
// Declare net variables shared between CPU and CLA here
extern volatile CNTL_3P3Z_CLA_C_Coeffs coeffs;
extern volatile CNTL_3P3Z_CLA_C_Vars vars;
```

Step 4 Initialize macro in CLA Task 8. Edit the task found in the `{ProjectName}-CLA_Tasks.cla` file to initialize the macro coefficients.

```
interrupt void Cla1Task8(void) {
	...
	CNTL_3P3Z_CLA_C_INIT(coeffs, vars);
	coeffs.Coeff_B2 = 5;
	coeffs.Min = 10;
	coeffs.IMin = 11;
	coeffs.Max = 12;
	...
}
```

Step 5 Connect the CNTL_3P3Z_CLA_C controller within a CLA-Task.

```
interrupt void Cla1Task4(void) {
	...
	vars.Ref = Ref;
	vars.Fdbk = Fbk;
	CNTL_3P3Z_CLA_C(coeffs, vars);
	Out = vars.Out;
	...
}
```

Step 4 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
	...
	PWM_1ch_CNF(5, 500, 1, 0);
	...
}
```

Step 5 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** "Steps to use CLA DPlib" for details on how to modify this function to suit your application needs.

Object Definition:

```
/**
 * Three-pole three-zero coefficient structure.
 */
typedef struct {
	// Coefficients
	float32 Coeff_B3;
	float32 Coeff_B2;
	float32 Coeff_B1;
	float32 Coeff_B0;
	float32 Coeff_A3;
	float32 Coeff_A2;
	float32 Coeff_A1;

	// Output saturation limits
	float32 Max;
	float32 IMin;
	float32 Min;
} CNTL_3P3Z_CLA_C_Coeffs;

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

	// Internal values
	float32 Errn;
	float32 Errn1;
	float32 Errn2;
	float32 Errn3;
	// Output values
	float32 Out;
	float32 Out1;
	float32 Out2;
	float32 Out3;
	float32 OutPresat;
} CNTL_3P3Z_CLA_C_Vars;
```

Special Constants and Data types

CNTL_3P3Z_CLA_C_Coeffs

Structure defined to store the controller coefficients and internal limits. To create multiple instances of the module simply declare variables of type CNTL_3P3Z_CLA_C_Coeffs.

CNTL_3P3Z_CLA_C_Vars

Structure defined to store the internal values and output. To create multiple instances of the module simply declare variables of type CNTL_3P3Z_CLA_C_Vars.

Module Structure Definition:

Parameter Name	Types	Description	Acceptable Range
Ref	Input	Reference value for the controller.	float32 [0, 1)
Fdbk	Input	Feedback value for the controller.	float32 [0, 1)
Coeff_B0, B1, B2, B3, A1, A2, A3	Input	Controller coefficients.	float32
Out	Output	Controller calculated output.	float32 [0,1)
Min, Max, IMin	Input	Internal saturation limits.	float32

5.2. Peripheral Configuration

ADC_SOC_Cnf	***ADC Configuration***

Description: This module configures the on chip analog to digital convertor to sample different ADC channels at particular peripheral triggers.

Peripheral Initialization File: `ADC_SOC_Cnf.c`

Description: C2000 devices have on chip analog to digital converter which can be configured to sample various signals in a power supply such as voltage and current in a very flexible manner. The sampling of these signals can be triggered from various peripheral sources and depending on signal characteristics the acquisition sample and hold window of the signal can be configured. The ADC_SOC_CNF function enables the user to configure the ADC as desired. The function is defined as:

```
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(SOC) trigger 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 Non interrupt mode/CLA mode, configures ADC conversions to be started by the appropriate channel trigger. An ADC interrupt is triggered when conversion is complete and the ADC interrupt flag is automatically 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 for the device.

Usage:

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. The ADC is configured in start stop mode and channel 0 is
configured to raise ADCINT 1. ADC Channel 0 is configured to be 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
ADC_SOC_CNF(ChSel,TrigSel,ACQPS,1,0);
```

PWM_PSFB_PCMC_CNF — *PWM Configuration for PCMC controlled PSFB Stage*

Description: This module configures the PWM generators to control a phase shifted full bridge (PSFB) in peak current mode control (PCMC) and also configures synchronous rectifiers (SR), if used.

Peripheral Initialization File: `PWM_PSFB_PCMC_Cnf.c`

Description: This module sets the initial configuration of two PWM peripheral modules to drive the four switches of the full bridge. It also has an option to configure the PWM module driving synchronous rectifier (SR) switches, if used. In this configuration the master module operates in up-down count mode and is used to drive switches in the full bridge leg with passive to active transitions. The next higher module in the PWM chain operates in up-count mode and is used to drive switches in the leg with active to passive transitions. The PWM module that drives SR switches also operates in up-count mode. These two slaved PWM module time-bases are synced at every half period of the master and are also directly synced by the comparator1 output.
This file is used in conjunction with the assembly code in the corresponding project specific ISR file. The `PWM_PSFB_PCMC_Cnf.c` file consists of the PWM configuration function:

```
void PWMDRV_PSFB_PCMC_CNF(int16 n, Uint16 period, int16 SR_Enable, int16 Comp2_Prot)
```

Where:

n	is the master PWM peripheral configured for driving switches in one leg of the full bridge. PWM n+1 is configured to work with synch pulses from PWM n module and drives switches in the other leg. PWM n+3 drives SR switches if SR_Enable is 1.
Period	is the maximum count value of the PWM timer
SR_Enable	This enables drive to SR switches using PWM n+3 module.
Comp2_Prot	Enables catastrophic protection based on on-chip comparator2 and DAC.

Usage:

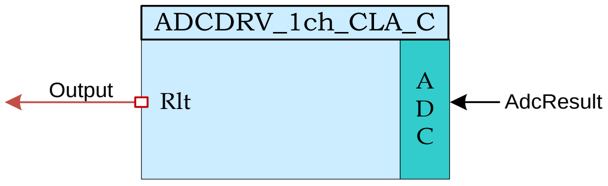
Call the peripheral configuration function PWMDRV_PSFB_PCMC_CNF(int16 n, int16 Period, int16 SR_Enable, int16 Comp2_Prot) in `{ProjectName}-Main.c`, this function is defined in `PWM_PSFB_PCMC_SR_Cnf.c`. This file must be linked manually to the project.

```
// ePWM1 is the master, Period=PWM_PRD, SR_Enable=1, Comp2_Prot=1
PWMDRV_PSFB_PCMC_CNF(1, PWM_PRD, 1, 1);
```

5.3. Peripheral Drivers

ADCDRV_1ch_CLA_C *ADC Driver Single Channel on CLA*

Description: This macro receives values from internal ADC module result registers and returns it in float format. The output is normalized to 0.0 - 1.0 such that the minimum input voltage at the ADC pin will generate 0.0 at the driver output, and a maximum full scale input voltage generates +1.0.



Macro File: `ADCDRV_CLA_C.h`
Peripheral Initialization File: `ADC_SOC_Cnf.c`

Description: The ADCDRV_1ch_CLA_C macro receives one ADC result value (i.e. AdcResult.ADCRESULT0, AdcResult.ADCRESULT5, etc.) and returns a floating point normalized value. This macro is used in conjunction with the `ADC_SOC_Cnf.c` peripheral configuration file.

Usage: This section explains how to use the ADCDRV_1ch_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `ADCDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Connect ADCDRV_1ch_ CLA_C macros within CLA-Tasks.

```
interrupt void Cla1Task4(void) {
        ...
        temp0 = ADCDRV_1ch_CLA_C(AdcResult.ADCRESULT0);
        temp1 = ADCDRV_1ch_CLA_C(AdcResult.ADCRESULT5);
        temp2 = ADCDRV_1ch_CLA_C(AdcResult.ADCRESULT7);
        ...
}
```

Step 3 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
        ...
        ADC_SOC_CNF(ChSel,TrigSel,ACQPS,1,0);
        ...
}
```

Step 4 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

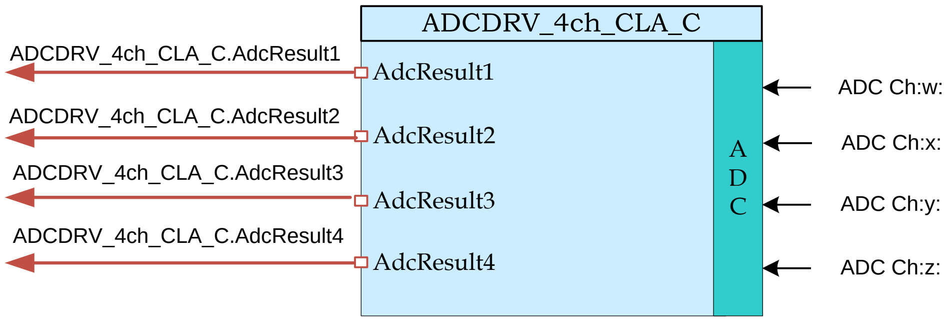
Module Parameters:

```
        ADCDRV_1ch_CLA_C(AdcConvResult);
```

Parameter Name	Types	Description	Acceptable Range
AdcConvResult	Input	ADC conversion result	Uint16

ADCDRV_4ch_CLA_C — *ADC Driver Four Channel using CLA*

Description: This macro reads four results from the internal ADC module result registers and stores them in normalized float format. The output is normalized to 0.0 - 1.0 such that the minimum input voltage will generate 0.0 at the driver output, and a maximum full scale input voltage read 1.0. The results are then stored in the ADCDRV_4ch_CLA_C structure.


ADCDRV_4ch_CLA_C
ADCDRV_4ch_CLA_C.AdcResult1
ADCDRV_4ch_CLA_C.AdcResult2
ADCDRV_4ch_CLA_C.AdcResult3
ADCDRV_4ch_CLA_C.AdcResult4
AdcResult1
AdcResult2
AdcResult3
AdcResult4
ADC
ADC Ch:w:
ADC Ch:x:
ADC Ch:y:
ADC Ch:z:

Macro File: `ADCDRV_CLA_C.h`

Peripheral Initialization File: `ADC_SOC_Cnf.c`

Description: The ADCDRV_4ch_CLA_C macro reads 4 ADC result registers and stores the normalized float numbers in an ADCDRV_4ch_CLA_C structure. Pointers to the ADC result registers must be assigned to variables within the structure. This macro is used in conjunction with the peripheral configuration file `ADC_SOC_Cnf.c`.

Usage: This section explains how to use the ADCDRV_4ch_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `ADCDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare ADCDRV_4ch_CLA_C structures in the `{ProjectName}-CLA_Tasks.cla` file. Specify structure variables and CLA memory locations.

```
/* DPlib variables & memory locations */
#pragma DATA_SECTION(adc4ch_1, "Cla1ToCpuMsgRAM")
ADCDRV_4ch_CLA_C adc4ch_1;
```

Step 3 Add shared declaration to `{ProjectName}-CLA_Shared.h` file if access is shared between CPU and CLA.

```
/* DPlib variables */
// Declare net variables shared between CPU and CLA here
extern volatile ADCDRV_4ch_CLA_C adc4ch_1;
```

Step 4 Add macro initialization to CLA Task 8. Edit the task found in the `{ProjectName}-CLA_Tasks.cla` file to initialize the macro coefficients.

```
interrupt void Cla1Task8(void) {
	...
	ADCDRV_4ch_CLA_C_INIT(adc4ch_1);
	adc4ch_1.AdcResPtr1 = &AdcResult.ADCRESULT3;
	adc4ch_1.AdcResPtr2 = &AdcResult.ADCRESULT2;
	adc4ch_1.AdcResPtr3 = &AdcResult.ADCRESULT1;
	adc4ch_1.AdcResPtr4 = &AdcResult.ADCRESULT6;
	...
}
```

Step 5 Connect the ADCDRV_4ch_CLA_C macro within a CLA-Task.

```
interrupt void Cla1Task4(void) {
	...
	ADCDRV_4ch_CLA_C(adc4ch_1);
	temp0 = adc4ch_1.AdcResult1;
	temp1 = adc4ch_1.AdcResult2;
	temp2 = adc4ch_1.AdcResult3;
	temp3 = adc4ch_1.AdcResult4;
	...
}
```

Step 6 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
	...
	ADC_SOC_CNF(ChSel,TrigSel,ACQPS,1,0);
	...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Object Definition:

```
/**
 * 4 channel ADCDRV structure.
 */
typedef struct{
	volatile Uint16 *AdcResPtr1;
	volatile Uint16 *AdcResPtr2;
	volatile Uint16 *AdcResPtr3;
	volatile Uint16 *AdcResPtr4;

	float32 AdcResult1;
	float32 AdcResult2;
	float32 AdcResult3;
	float32 AdcResult4;
} ADCDRV_4ch_CLA_C;
```

Special Constants and Data types

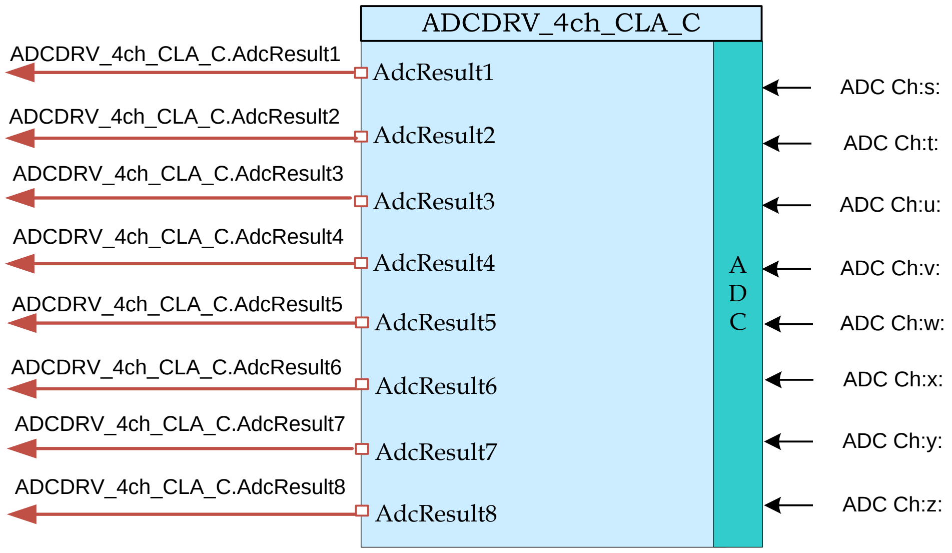
ADCDRV_4ch_CLA_C

Structure defined to store the pointers to ADC result registers and normalized values. To create multiple instances of the module simply declare variables of type ADCDRV_4ch_CLA_C. Care should be taken when creating multiple instances; it is not recommended that a single ADC result register be used in more than one module instance.

Module Structure Definition:

Parameter Name	Types	Description	Acceptable Range
AdcResPtr1,2,3,4	Input	Pointers to ADC result registers	Uint16
AdcResult1,2,3,4	Output	Normalized float ADC output	float32 [0, 1]

Description: This macro reads eight results from the internal ADC module result registers and stores them, in normalized float format, in the output variables. The output is normalized to 0.0 - 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 results are then stored in an ADCDRV_8ch_CLA_C structure in memory.


ADCDRV_4ch_CLA_C
ADCDRV_4ch_CLA_C.AdcResult1
ADCDRV_4ch_CLA_C.AdcResult2
ADCDRV_4ch_CLA_C.AdcResult3
ADCDRV_4ch_CLA_C.AdcResult4
ADCDRV_4ch_CLA_C.AdcResult5
ADCDRV_4ch_CLA_C.AdcResult6
ADCDRV_4ch_CLA_C.AdcResult7
ADCDRV_4ch_CLA_C.AdcResult8
AdcResult1
AdcResult2
AdcResult3
AdcResult4
AdcResult5
AdcResult6
AdcResult7
AdcResult8
ADC
ADC Ch:s:
ADC Ch:t:
ADC Ch:u:
ADC Ch:v:
ADC Ch:w:
ADC Ch:x:
ADC Ch:y:
ADC Ch:z:

Macro File: `ADCDRV_CLA_C.h`

Peripheral Initialization File: `ADC_SOC_Cnf.c`

Description: The ADC module includes a ratio-metric input which enables the user to determine the maximum and minimum input voltages. The ADC converts this input range with 12-bits of resolution. The ADCDRV macro reads the result register then converts these to normalized float format and writes the results in output variables. This macro is used in conjunction with the peripheral configuration file `ADC_SOC_Cnf.c`.

Usage: This section explains how to use the ADCDRV_8ch_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `ADCDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare ADCDRV_8ch_CLA_C structures in the `{ProjectName}-CLA_Tasks.cla` file. Specify structure variables and CLA memory locations.

```
/* DPlib variables & memory locations */
#pragma DATA_SECTION(adc8ch, "Cla1ToCpuMsgRAM")
ADCDRV_8ch_CLA_C adc8ch;
```

Step 3 Add shared declaration to `{ProjectName}-CLA_Shared.h` file if access is shared between CPU and CLA.

```
/* DPlib variables */
// Declare net variables shared between CPU and CLA here
extern volatile ADCDRV_8ch_CLA_C adc8ch;
```

Step 4 Add macro initialization to CLA Task 8. Edit the task found in the `{ProjectName}-CLA_Tasks.cla` file to initialize the macro coefficients.

```
interrupt void Cla1Task8(void) {
	...
	ADCDRV_8ch_CLA_C_INIT(adc8ch);
	adc8ch.AdcResPtr1 = &AdcResult.ADCRESULT1;
	adc8ch.AdcResPtr2 = &AdcResult.ADCRESULT2;
	adc8ch.AdcResPtr3 = &AdcResult.ADCRESULT3;
	adc8ch.AdcResPtr4 = &AdcResult.ADCRESULT4;
	adc8ch.AdcResPtr5 = &AdcResult.ADCRESULT5;
	adc8ch.AdcResPtr6 = &AdcResult.ADCRESULT6;
	adc8ch.AdcResPtr7 = &AdcResult.ADCRESULT7;
	adc8ch.AdcResPtr8 = &AdcResult.ADCRESULT8;
	...
}
```

Step 5 Connect the ADCDRV_8ch_CLA_C macro within a CLA-Task.

```
interrupt void Cla1Task4(void) {
	...
	ADCDRV_8ch_CLA_C(adc8ch);
	temp0 = adc8ch.AdcResult1;
	temp1 = adc8ch.AdcResult2;
	temp2 = adc8ch.AdcResult3;
	temp3 = adc8ch.AdcResult4;
	...
}
```

Step 6 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
	...
	ADC_SOC_CNF(ChSel,TrigSel,ACQPS,1,0);
	...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Object Definition:

```
/**
 * 8 channel ADCDRV structure.
 */
typedef struct{
	volatile Uint16 *AdcResPtr1;
	volatile Uint16 *AdcResPtr2;
	volatile Uint16 *AdcResPtr3;
	volatile Uint16 *AdcResPtr4;
	volatile Uint16 *AdcResPtr5;
	volatile Uint16 *AdcResPtr6;
	volatile Uint16 *AdcResPtr7;
	volatile Uint16 *AdcResPtr8;

	float32 AdcResult1;
	float32 AdcResult2;
	float32 AdcResult3;
	float32 AdcResult4;
	float32 AdcResult5;
	float32 AdcResult6;
	float32 AdcResult7;
	float32 AdcResult8;
} ADCDRV_8ch_CLA_C;
```

Special Constants and Data types

ADCDRV_8ch_CLA_C

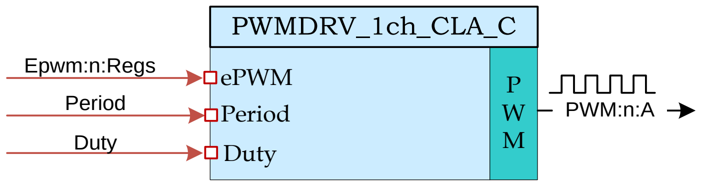
Structure defined to store the pointers to ADC result registers and normalized values. To create multiple instances of the module simply declare variables of type ADCDRV_4ch_CLA_C. Care should be taken when creating multiple instances; it is not recommended that a single ADC result register be used in more than one module instance.

Module Structure Definition:

Parameter Name	**Types**	**Description**	**Acceptable Range**
AdcResPtr1-8	Input	Pointers to ADC result registers	Uint16
AdcResult1-8	Output	Normalized float ADC output	float32 [0, 1]

PWMDRV_1ch_CLA_C — *PWM Driver Single Channel using CLA*

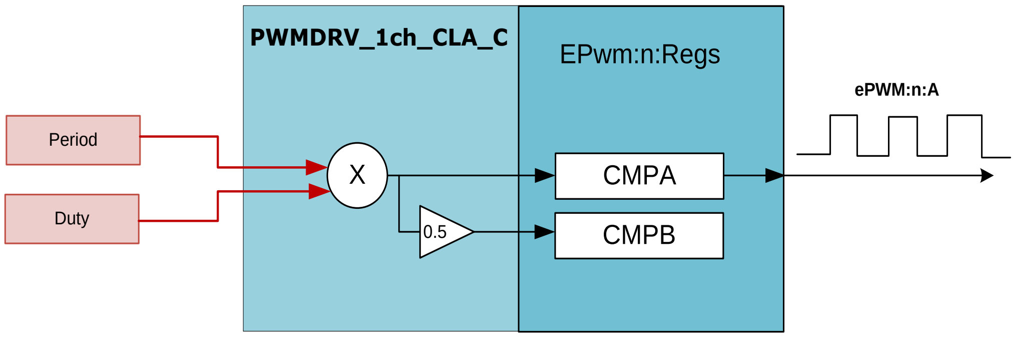
Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a duty on PWM channel A, depending on the duty and period values.



Macro File: `PWMDRV_CLA_C.h`

Peripheral Initialization File: `PWMDRV_1ch_Cnf.c`

Description: This macro provides the interface between DP library variables and the ePWM modules. The macro scales the float Duty value by the Period value and stores this value in the EPwm:n:Regs.CMPA.half.CMPA register. The module also writes half the value of CMPA into the EPwm:n:Regs.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. The PWM driven is determined by the PWM register number 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, Uint16 period, int16 mode, int16 phase)
```

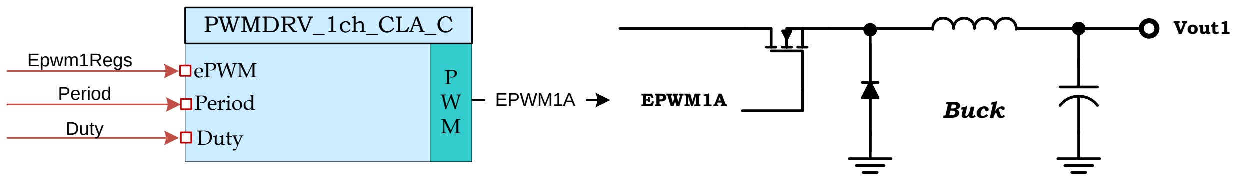
Where:

n	PWM peripheral number; PWM will be configured in up-count mode.
period	the maximum count value of the PWM timer

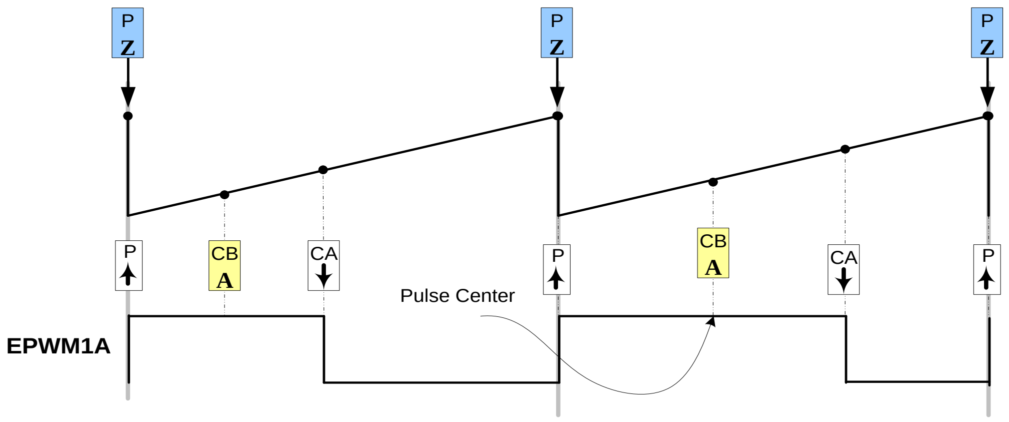
mode	determines whether the PWM is to be configured as slave or master, when configured as the master the TBSYNC signal is ignored by the PWM module.
	Mode = 1 PWM configured as master
	Mode = 0 PWM configured as slave

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 100 kHz switching frequency with CPU operating at 60 MHz, the period value needed is (System Clock/Switching Frequency) = 600.



Buck converter driven by PWMDRV_1ch_CLA_C module



PWM generation with the EPWM module.

Usage: This section explains how to use the PWMDRV_1ch_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Connect the PWMDRV_1ch_CLA_C macro within a CLA-Task.

```
interrupt void Cla1Task4(void) {
        ...
        PWMDRV_1ch_CLA_C(EPwm1Regs, EPwm1Regs.TBPRD, duty);
        ...
}
```

Step 3 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
        ...
        PWM_1ch_CNF(1, 600, 1, 0);
        ...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** "Steps to use CLA DPlib" for details on how to modify this function to suit your application needs.

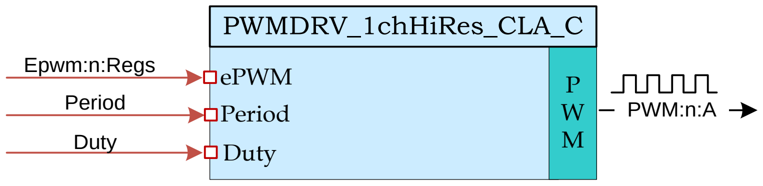
Module Parameters:

```
        PWMDRV_1ch_CLA_C(EPwm:n:Regs, period, duty);
```

Parameter Name	Type	Description	Acceptable Range
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven.	Device dependent
period	Input	Period by which the duty cycle on the PWM module is driven.	Uint16
duty	Input	Desired PWM module duty cycle for the given period	float32 [0, 1)

PWMDRV_1chHiRes_CLA_C — *PWM Driver Single Channel Hi Res*

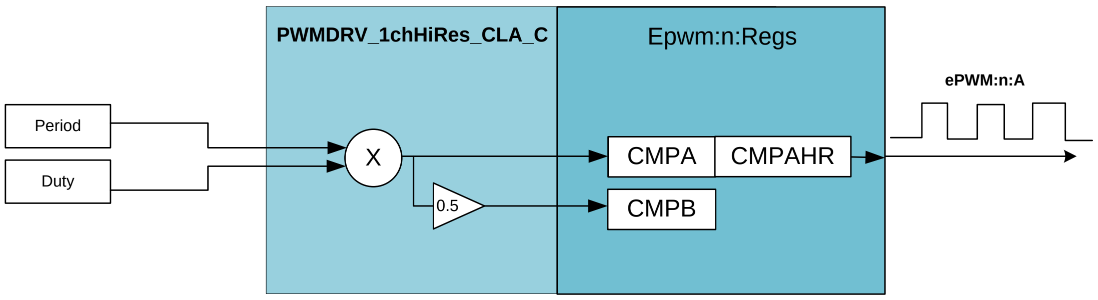
Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a high resolution duty on PWM channel A, using the Hi-Res feature, depending on the value of the duty value.



Macro File: `PWMDRV_CLA_C.h`

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 macro provides the interface between DP library variables and ePWM modules on C28x. The macro scales the float Duty value by the Period value and stores it in the EPwm:n:Regs.CMPA.all register. The module also writes half the value of the CMPA.half.CMPA register into EPwm:n:Regs.CMPB register. This is done to enable ADC start of conversions to occur close to the mid-point of the PWM waveform, avoiding switching noise. Which PWM is driven is determined by the PWM register number 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, Uint16 period, int16 mode, int16
phase)
```

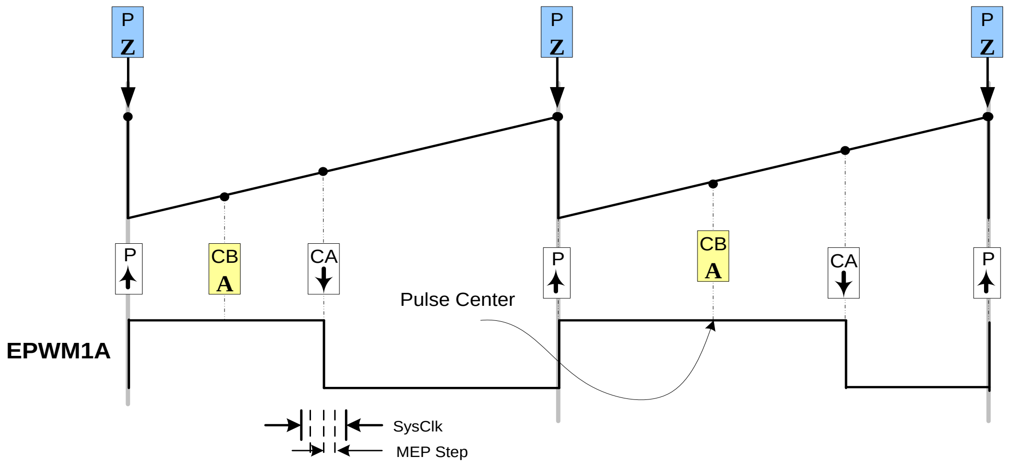
Where:

n	PWM Peripheral number; PWM will be 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 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 a 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 a timing diagram of 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. The SFO function can only be called by the CPU and not the CLA.

Usage: This section explains how to use the PWMDRV_1chHiRes_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_CLA_C.h` include found at the top of the `DPlib.h` file. The SFO library needs to be manually included in the project. Please use V6 or higher of the SFO library for this module to work appropriately.

The library can be found at:
controlSUITE\device_support\<Device_Name>\<Version>\<Device_Name>_common\lib

The header file can be found at:
controlSUITE\device_support\<Device_Name>\<Version>\<Device_Name>_common\include

```
#include "DPlib.h"
#include "SFO_V6.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare SFO global variables in the `{ProjectName}-Main.c` file.

```
// 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 Connect the PWMDRV_1chHiRes_CLA_C macro within a CLA-Task.

```
interrupt void Cla1Task4(void) {
       ...
       PWMDRV_1chHiRes_CLA_C(EPwm1Regs, EPwm1Regs.TBPRD, duty);
       ...
}
```

Step 4 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals. 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.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
       ...
       PWM_1chHiRes_CNF(1, 600, 1, 0);
       ...
       // SFO() updates HRMSTEP register with calibrated MEP_ScaleFactor.
       // MEP_ScaleFactor/HRMSTEP must have calibrated value in order 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); // Loop forever if error returned.
       ...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

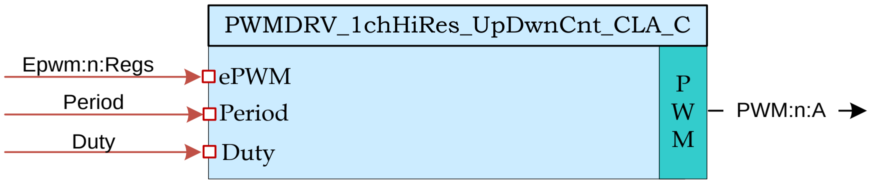
Module Parameters:

```
PWMDRV_1chHiRes_CLA_C(EPwm:n:Regs, period, duty);
```

Parameter Name	**Type**	**Description**	**Acceptable Range**
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven.	Device dependent
period	Input	Period by which the duty cycle on the PWM module is driven.	Uint16
duty	Input	Desired PWM module duty cycle for the given period	float32 [0, 1)

PWMDRV_1chHiRes_UpDwnCnt_CLA_C — *PWM Driver Single Channel Hi Res*

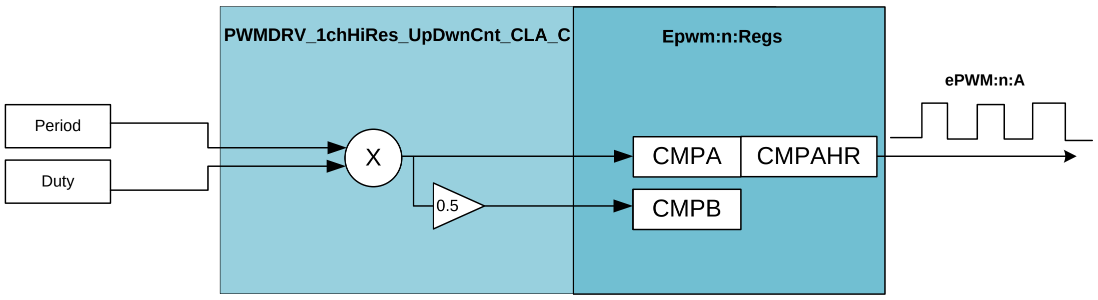
Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a high resolution duty on PWM channel A, using the Hi-Res feature, depending on the value of the duty value.



Macro File: `PWMDRV_CLA_C.h`

Peripheral Initialization File: `PWM_1chHiResUpDwnCnt_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 macro provides the interface between DP library variables and ePWM modules on C28x. The macro scales the float Duty value by the Period value and stores it in the EPwm:n:Regs.CMPA.all register. The module also writes half the value of the CMPA.half.CMPA register into EPwm:n:Regs.CMPB register. This is done to enable ADC start of conversions to occur close to the mid-point of the PWM waveform, avoiding switching noise. Which PWM is driven is determined by the PWM register number i.e. :n:.



This macro is used in conjunction with the peripheral configuration file `PWM_1chHiResUpDwnCnt_Cnf.c`. The file defines the function:

```
void PWM_1chHiRes_UpDwnCnt_CNF(int16 n, Uint16 period, int16
mode, int16 phase)
```

Where:

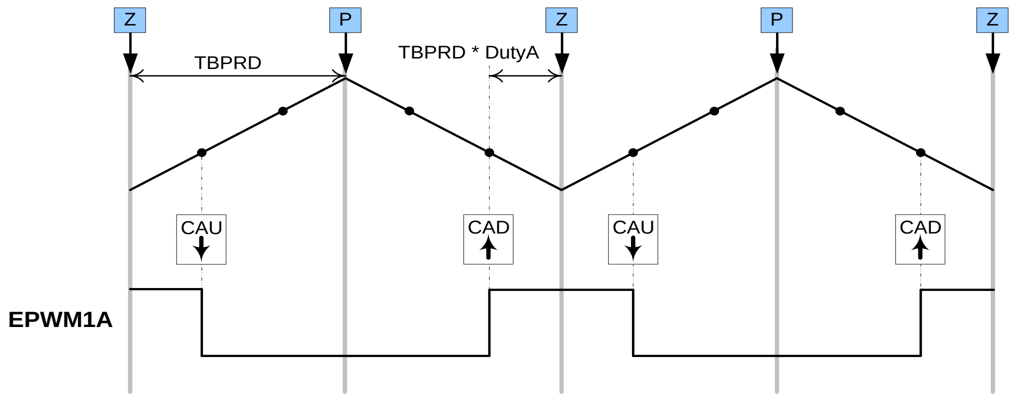
n	PWM Peripheral number; PWM will be configured in up-down count mode

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 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 a 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 a timing diagram of 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. The SFO function can only be called by the CPU and not the CLA.

Usage: This section explains how to use the PWMDRV_1chHiRes_UpDwnCnt_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_CLA_C.h` include found at the top of the `DPlib.h` file. The SFO library needs to be manually included in the project. Please use V6 or higher of the SFO library for this module to work appropriately.

The library can be found at:
controlSUITE\device_support\<Device_Name>\<Version>\<Device_Name>_common\lib

The header file can be found at:
controlSUITE\device_support\<Device_Name>\<Version>\<Device_Name>_common\include

```
#include "DPlib.h"
#include "SFO_V6.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare SFO global variables in the `{ProjectName}-Main.c` file.

```
// 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 Connect the PWMDRV_1chHiRes_UpDwnCnt_CLA_C macro within a CLA-Task.

```
interrupt void Cla1Task4(void) {
        ...
        PWMDRV_1chHiRes_UpDwnCnt_CLA_C(EPwm1Regs, EPwm1Regs.TBPRD, duty);
        ...
}
```

Step 4 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals. 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.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
        ...
        PWM_1chHiRes_CNF(1, 600, 1, 0);
        ...
        // SFO() updates HRMSTEP register with calibrated MEP_ScaleFactor.
        // MEP_ScaleFactor/HRMSTEP must have calibrated value in order 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); // Loop forever if error returned.
      ...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

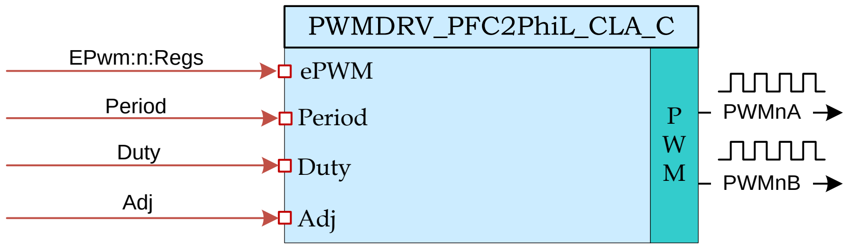
Module Parameters:

```
PWMDRV_1chHiRes_UpDwnCnt_CLA_C(EPwm:n:Regs, period, duty);
```

Parameter Name	Type	Description	Acceptable Range
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven.	Device dependent
period	Input	Period by which the duty cycle on the PWM module is driven.	Uint16
duty	Input	Desired PWM module duty cycle for the given period	float32 [0, 1)

PWMDRV_PFC2PhiL_CLA_C — *PWM Driver for Two Phase Interleaved PFC Stage*

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



Macro File: `PWMDRV_CLA_C.h`

Peripheral Initialization File: `PWM_PFC2PhiL_Cnf.c`

Description: This module forms the interface between the control software and the device PWM pins. The macro scales the float Duty value by the Period value and stores it in the EPwm:n:Regs.CMPA.half.CMPA register. The Adj value is also scaled by the Period value; it adds an offset to the duty being driven on PWM:n:A and PWM:n:B. In summary:

$$CMPA = Duty * Period$$

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

The PWM module updated is determined by the PWM number i.e. :n:

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, Uint16 period)
```

Where:

n	PWM peripheral number, PWM will be 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 a 100kHz switching frequency with 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.

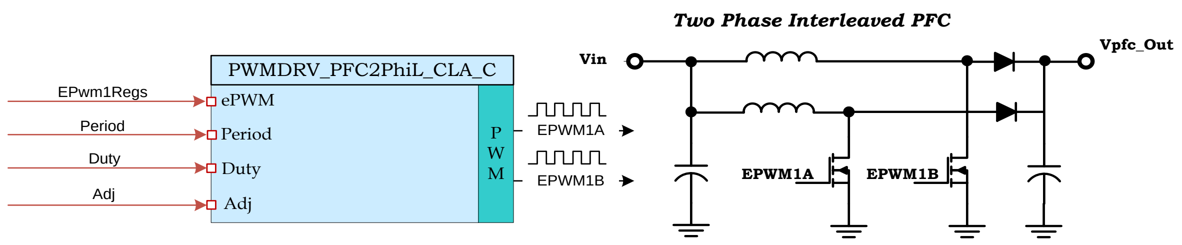


Figure: PFC2PhiL driven by PWMDRV_PFC2PhiL module

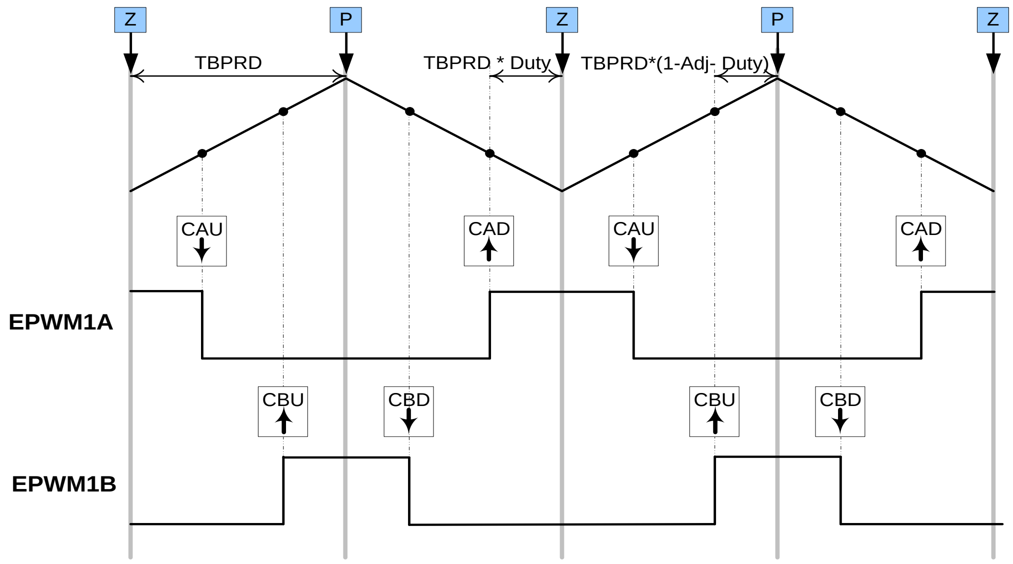


Figure: PWM generation for PFC2PhiL stage with the EPWM module.

Usage: This section explains how to use the PWMDRV_PFC2PhiL_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Connect the PWMDRV_PFC2PhiL_CLA_C controller within a CLA-Task.

```
interrupt void Cla1Task4(void) {
        ...
        PWMDRV_PFC2PhiL_CLA_C(EPwm1Regs, EPwm1Regs.TBPRD, duty, adj);
        ...
}
```

Step 3 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
       ...
       PWM_PFC2PHIL_CNF(1, 600);
       ...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

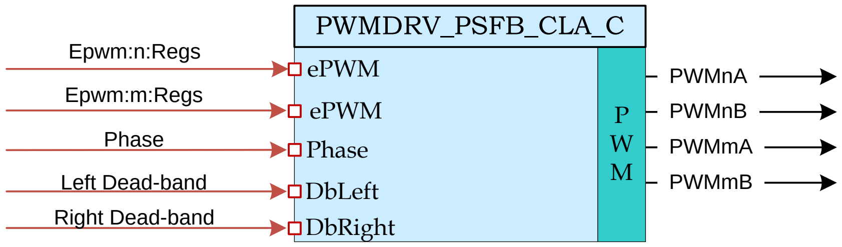
Module Parameters:

```
      PWMDRV_PFC2PhiL_CLA_C(EPwm:n:Regs, period, duty, adj);
```

Parameter Name	Type	Description	Acceptable Range
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven.	Device dependent
period	Input	Period by which the duty cycle on the PWM module is driven.	Uint16
duty	Input	Desired PWM module duty cycle for the given period	float32 [0, 1)
adj	Input	Complementary channel offset.	float32[0, 1)

PWMDRV_PSFB_CLA_C — *PWM Driver for Phase Shifted Full Bridge Stage*

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_CLA_C.h`

Peripheral Initialization File: `PWM_PSFB_Cnf.c`

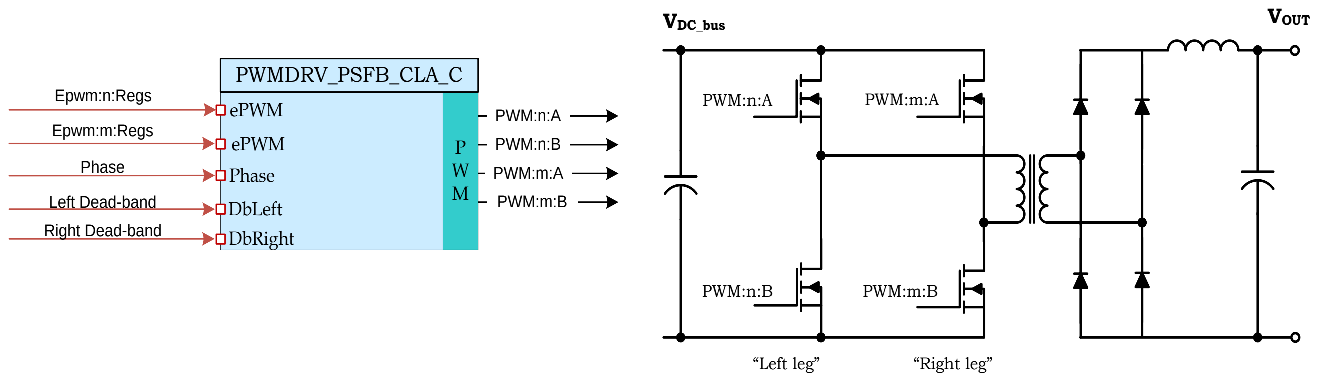
Description: This module forms the interface between the control software and the device PWM pins. The macro scales the Phase value by EPwm:n:Regs period value and stores this value in the EPwm:n:Regs.TBPHS register. This macro is used in conjunction with the Peripheral configuration file `PWM_PSFB_CNF.c`. The file defines the function:

```
void PWM_PSFB_CNF(int16 n, Uint16 period)
```

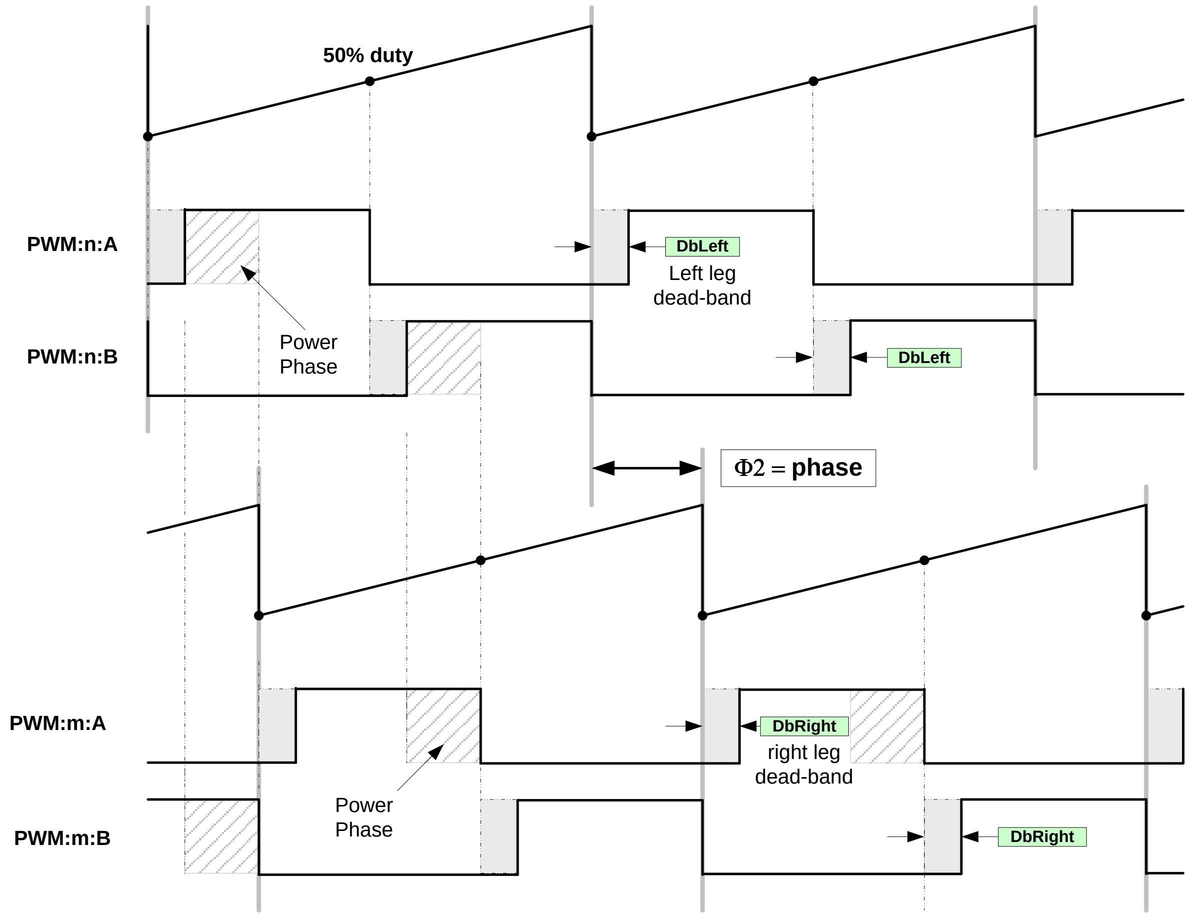
Where:

n	PWM Peripheral number; PWM will be configured for PSFB topology, PWM m (m = 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. Using up-count mode to configure 100kHz switching frequency 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: This section explains how to use the PWMDRV_PSFB_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Connect the PWMDRV_PSFB_CLA_C macro within a CLA-Task.

```
interrupt void Cla1Task4(void) {
        ...
        PWMDRV_PSFB_CLA_C(EPwm1Regs, EPwm2Regs, phase, dbLeft, dbRight);
        ...
}
```

Step 3 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
        ...
        PWM_PSFB_CNF(1, 600);
        ...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

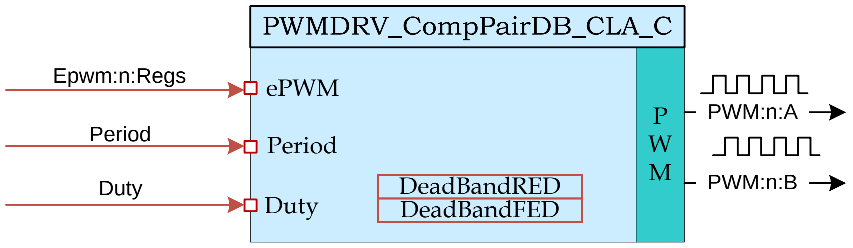
Module Parameters:

```
        PWMDRV_PSFB_CLA_C(EPwm:n:Regs, EPwm:m:Regs, phase, dbLeft, dbRight);
```

Parameter Name	Type	Description	Acceptable Range
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven as left leg of the PSFB topology.	Device dependent
EPwm:m:Regs	Input	PWM module registers. The :m: must be = n + 1. PWM is the right leg of the PSFB topology.	Requirement: m = (n + 1)
phase	Input	Phase shift value.	float32 [0, 1)
dbLeft	Input	Dead-band value for left leg.	Uint16
dbRIght	Input	Dead-band value for right leg.	Uint16

PWMDRV_ComplPairDB_CLA_C — *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 dead-band module inside the EPWM peripheral to generate the complimentary waveforms.



Macro File: `PWMDRV_CLA_C.h`

Peripheral Initialization File: `PWM_ComplPairDB_Cnf.c`

Description: This macro provides the interface between DP library variables and the ePWM modules on C28x. The macro scales the float Duty value by the Period value and stores this value in the EPwm:n:Regs.CMPA.half.CMPA. The corresponding configuration file has the dead-band module configured to output complimentary waveform on channel B 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, Uint16 period, int16 mode,
int16 phase)
```

Where:

n	PWM peripheral number, PWM to be configured in up-count mode
period	is the maximum value of the PWM counter

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 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 and the dead-band sub-module to output complimentary PWM waveforms. The falling edge delay is implemented by delaying the rising edge of channel B using the dead-band module in the PWM peripheral. The module outputs an active high duty on channel A of the PWM peripheral and a complementary active low duty cycle on channel B.

The configuration function only configures the dead-band at initialization time however it may be necessary 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)
```

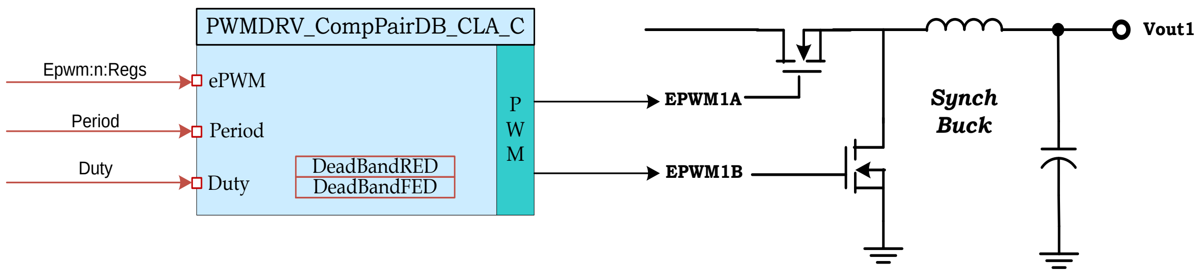
Where:

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

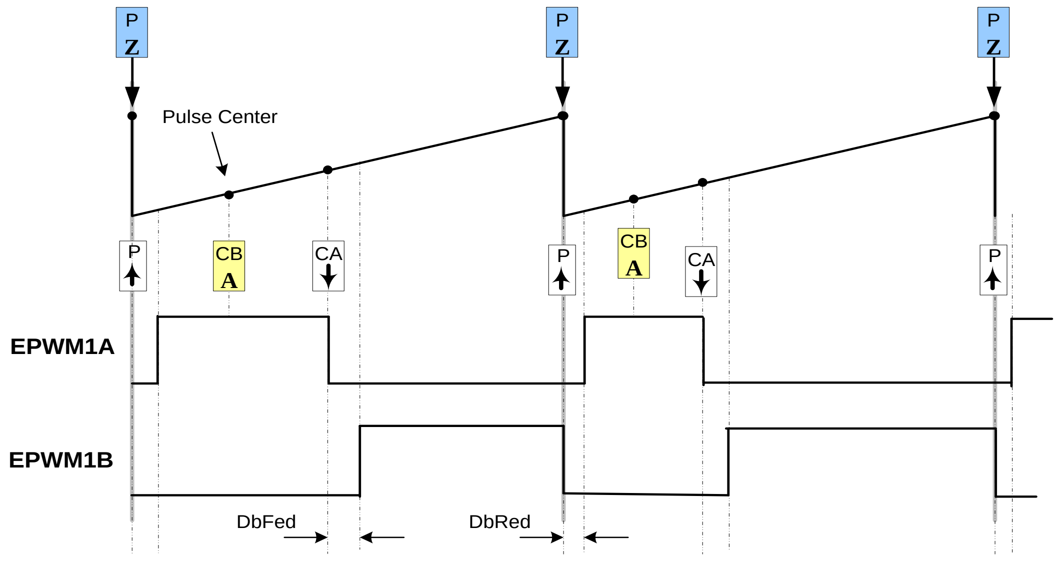
Alternatively a CLA macro is provided to update the dead-band; if this needs to be done at a faster rate inside the ISR. The dead-band update macro is called PWMDRV_ComplPairDB_CLA_C_UpdateDB.

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 PWMDRV Macro

Usage: This section explains how to use the PWMDRV_ComplPairDB_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Connect the PWMDRV_ComplPairDB_CLA_C macro within a CLA-Task. If the dead-band will be updated within the control loop, add calls to the dead-band update macro.

```
interrupt void Cla1Task4(void) {
        ...
        PWMDRV_ComplPairDB_CLA_C(EPwm1Regs, EPwm1Regs.TBPRD, duty);
        ...
        PWMDRV_ComplPairDB_CLA_C_UpdateDB(Epwm1Regs, dbRed, dbFed);
        ...
}
```

Step 3 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
        ...
        PWM_ComplPairDB_CNF(1, 600, 1, 0);
        ...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Module Parameters:

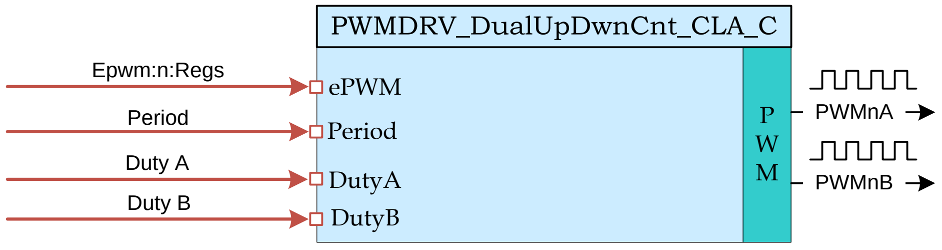
```
PWMDRV_ComplPairDB_CLA_C(EPwm:n:Regs, period, duty);

PWMDRV_ComplPairDB_CLA_C_UpdateDB(Epwm:n:Regs, dbRed, dbFed);
```

Parameter Name	Type	Description	Acceptable Range
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven.	Device dependent
period	Input	Period by which the duty cycle is driven on the PWM module.	Uint16
duty	Input	Desired PWM module duty cycle for the given period.	float32 [0, 1)
dbRed	Input	Dead-band value for rising edge delay.	Uint16
dbFed	Input	Dead-band value for falling edge delay.	Uint16

PWMDRV_DualUpDwnCnt_CLA_C

Independent PWM Duty on chA & chB

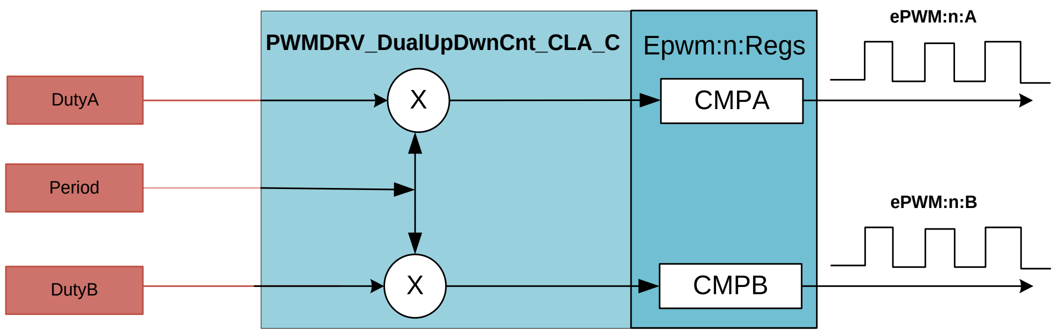
Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drive a duty on PWM channel A, and PWM channel B dependent on the value of the input variables DutyA and DutyB respectively.



Macro File: `PWMDRV_CLA_C.h`

Peripheral Initialization File: `PWM_DualUpDwnCnt_Cnf.c`

Description: This macro provides the interface between DP library variables and the ePWM modules on C28x. The scales the float DutyA and DutyB values by the Period values and stores these values in the EPwmRegs:n:.CMPA.half.CMPA and EPwmRegs:n:.CMPB such as to give independent 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:

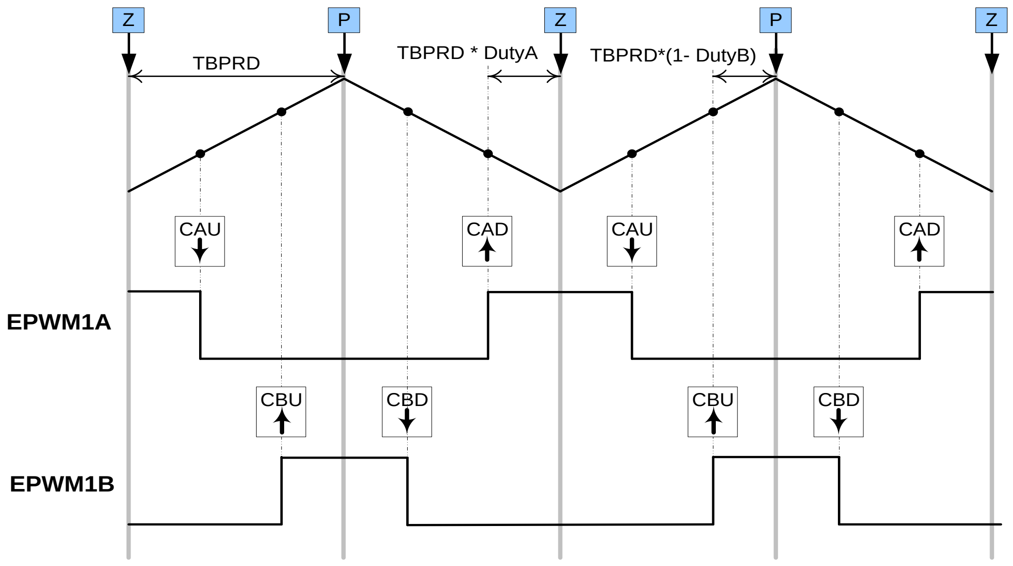
```
void PWM_DualUpDwnCnt_CNF(int16 n, Uint16 period, int16 mode,
    int16 phase)
```

Where:

n	PWM peripheral number, PWM to be configured in up-down count mode
period	is the maximum value of the PWM counter

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-down 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 a 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 create a 100kHz symmetric waveform where the value of CMPA and CMPB change the duty cycle of PWM1A and PWM1B respectively. Since up-down count mode is used, configuring the PWM to run at 100kHz switching frequency with a 60MHz system clock, a period value of (System Clock/Switching Frequency)/2 = 300 must be used in the CNF function.



PWM generation with the EPWM module.

Usage: This section explains how to use the PWMDRV_DualUpDwnCnt_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Connect the PWMDRV_DualUpDwnCnt_CLA_C macro within a CLA-Task.

```
interrupt void Cla1Task4(void) {
        ...
        PWMDRV_DualUpDwnCnt_CLA_C(EPwm1Regs, EPwm1Regs.TBPRD, dutyA, dutyB);
        ...
}
```

Step 3 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
        ...
        PWM_DualUpDwnCnt_CNF(1, 300, 1, 0);
        ...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Module Parameters:

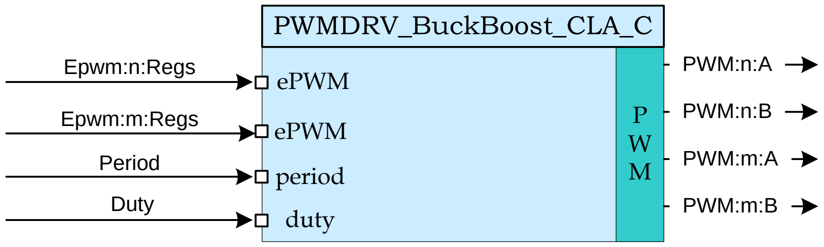
```
PWMDRV_DualUpDwnCnt_CLA_C(EPwm:n:Regs, period, dutyA, dutyB);
```

Parameter Name	Type	Description	Acceptable Range
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven.	Device dependent
period	Input	Period by which the duty cycle is driven on the PWM module.	Uint16
dutyA	Input	Desired PWM module duty cycle for channel A.	float32 [0, 1)
dutyB	Input	Desired PWM module duty cycle for channel B.	float32 [0, 1)

PWMDRV_BuckBoost_CLA_C

Dual PWM Driver, independent Duty on chA & chB
PWM Driver for Buck Boost Stage

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, on two PWM modules. The power stage, described later, is such that a Duty > 0.5 (pu) has a boost effect and a Duty < 0.5 (pu) has a buck effect on output to input voltage relation.



Macro File: `PWMDRV_BuckBoost_CLA_C.h`

Peripheral Initialization File: `PWM_BuckBoost_Cnf.c`

Description: This macro provides the interface between DP library variables and the ePWM modules on the C28x. The macro scales the float Duty value by the float Period value and stores it in the EPwm:n:Regs.CMPA.half.CMPA and EPwm:m:Regs.CMPA.half.CMPA registers. Half of the value is stored in EPwm:n:Regs.CMPB register. The configuration file sets the dead-band module to output a complimentary waveform on channel B. This macro must be used in conjunction with the peripheral configuration file `PWM_BuckBoost_Cnf.c`. The file defines the function:

```
void PWM_BuckBoost_CNF(int16 n, int16 m, Uint16 period)
```

Where:

n	PWM peripheral number; PWM to be configured in up-count mode
m	is (n+1) always; the following PWM module
period	is the maximum value of the PWM counter

The function configures the PWM peripheral in up-count mode and configures the dead-band sub-module to output complimentary PWM waveforms. Periodic sync pulses are also enabled between PWM module :n: and :m:. 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 channel A of the PWM peripheral and a complementary on channel B.

The configuration function however does not set the dead-band values; this is done by calling the C function `PWM_BuckBoost_UpdateDB` function as follows:

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

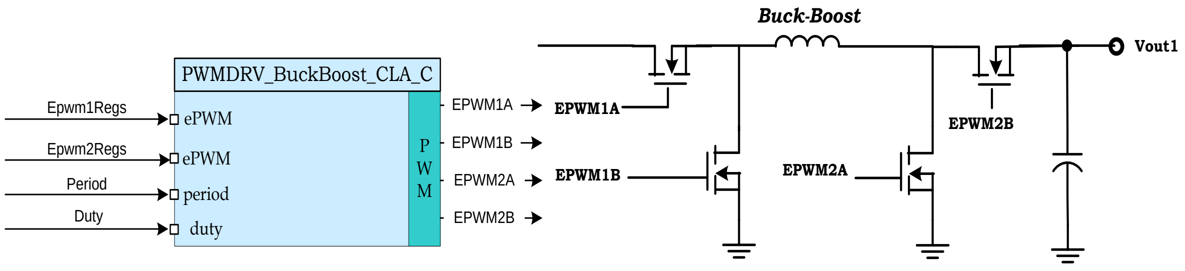
Where:

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

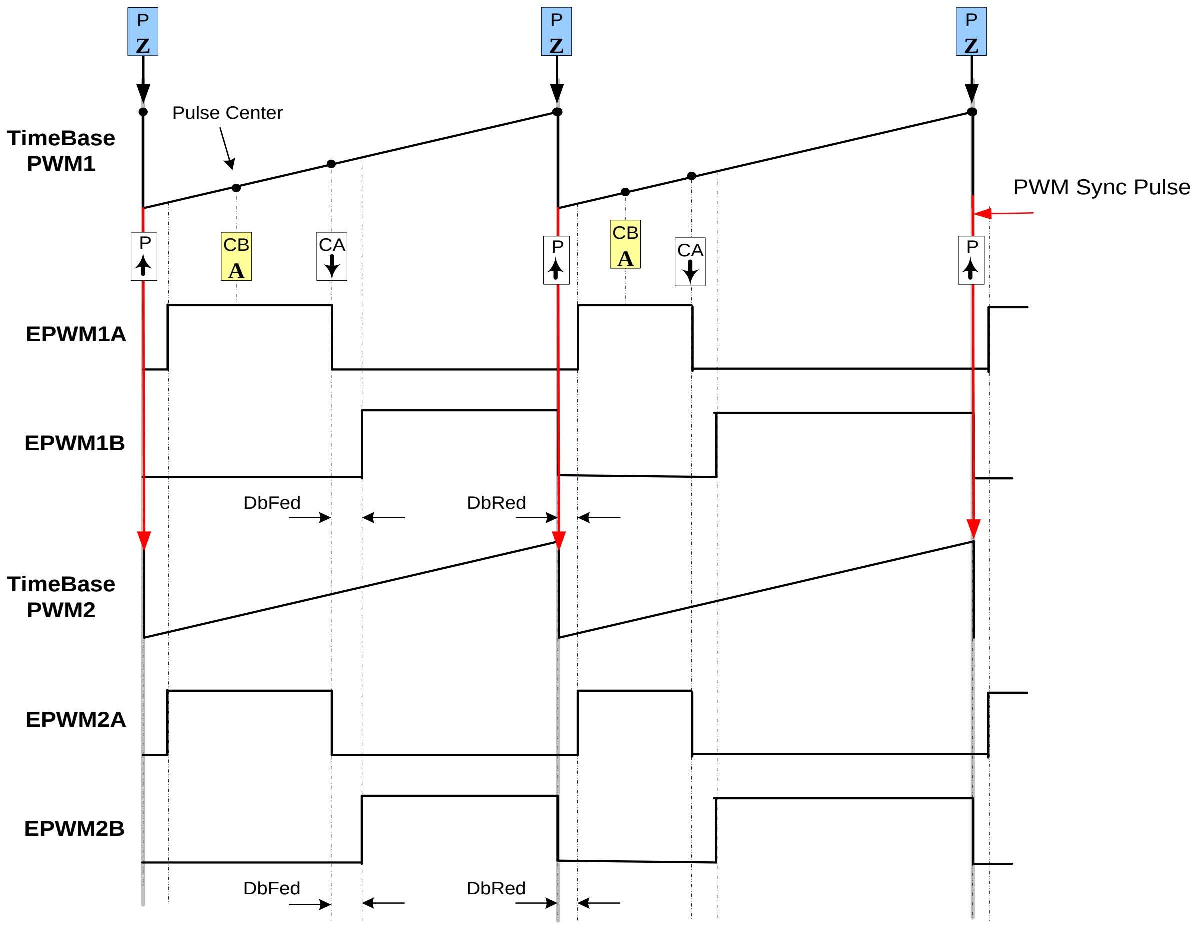
This function would enable fine tuning of the dead-band for this power stage. i.e. PWM 1 dead-band can be adjusted different from PWM 2. If the dead-band must be updated at a faster rate or within the control loop, the PWMDRV_BuckBoost_CLA_C_UpdateDB macro can be used.

Detailed Description

The following section explains how this module can be used to excite a buck boost stage. The module assumes the FET's used are NFET's; an active high is required to turn the FET on. The function configures the PWM peripheral in up-count mode. In up-count mode, to configure a 100kHz PWM switching frequency when CPU is operating at 60Mhz, a period value of (System Clock/Switching Frequency) = 600 is needed by the CNF function. The TBPRD is stored with a value of 600-1, to account for up-count mode; this is taken care of by the CNF function.



Buck Boost Converter driven by PWMDRV_BuckBoost module



PWM generation for Buck-Boost PWM DRV Macro

Few things to note about the power stage: the power stage can be used such that the FETs are switched as buck only and boost only, however these modes would put undue load on the Boot Strap for the high side driver. Using two different PWM's enables individual control of the dead-bands which can enable finer tuning of the power stage. Additionally, phase shifting and other modes are possible as well for the power stage when using two PWM modules.

Usage: This section explains how to use the PWMDRV_BuckBoost_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Connect the PWMDRV_BuckBoost_CLA_C macro within a CLA-Task. If the dead-band will be updated within the control loop, add calls to the dead-band update macro.

```
interrupt void Cla1Task4(void) {
        ...
        PWMDRV_BuckBoost_CLA_C(EPwm1Regs, EPwm2Regs, EPwm1Regs.TBPRD, duty);
        ...
        PWMDRV_BuckBoost_CLA_C(EPwm1Regs, dbRed, dbFed);
        ...
}
```

Step 3 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
        ...
        PWM_BuckBoost_CNF(1, 2, 600);
        PWM_BuckBoost_UpdateDB(1, dbRED, dbFED);
        ...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Module Parameters:

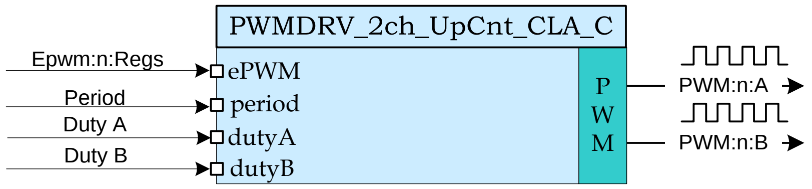
```
        PWMDRV_BuckBoost_CLA_C(EPwm:n:Regs, EPwm:m:Regs, period, duty);

        PWMDRV_BuckBoost_CLA_C(EPwm:n:Regs, dbRed, dbFed);
```

Parameter Name	Type	Description	Acceptable Range
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven as the first switch.	Device dependent
EPwm:m:Regs	Input	PWM module registers. The :m: indicates which PWM is driven as the second switch.	Requirement: m = (n + 1)
period	Input	Period by which the duty cycle is driven on the PWM module.	Uint16
duty	Input	Desired PWM module duty cycle for the given period.	float32 [0, 1)
dbRed	Input	Dead-band value for rising edge delay.	Uint16
dbFed	Input	Dead-band value for falling edge delay.	Uint16

PWMDRV_2ch_UpCnt_CLA_C

Dual PWM Driver, independent Duty on chA & chB
PWM Driver Two Channel

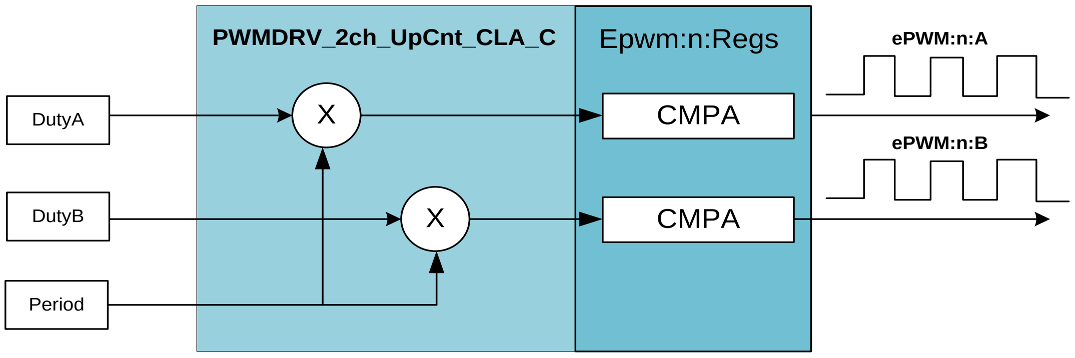
Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives two independent duties on PWM channels A and B, depending on the value of the Duty input variables.



Macro File: `PWMDRV_CLA_C.h`

Peripheral Initialization File: `PWM_2ch_UpCnt_Cnf.c`

Description: This macro provides the interface between DP library variables and the ePWM modules on C28x. The macro scales the float Duty A and Duty B values by the float Period value and stores the results in the EPwm:n:Regs.CMPA.half.CMPA and EPwm:n:Regs.CMPB respectively. The PWM module driven is determined by the PWM number i.e. :n:.



This macro is used in conjunction with the peripheral configuration file `PWM_2ch_UpCnt_Cnf.c`. The file defines the function:

```
void PWM_2ch_UpCnt_CNF(int16 n, Uint16 period, int16 mode, int16 phase)
```

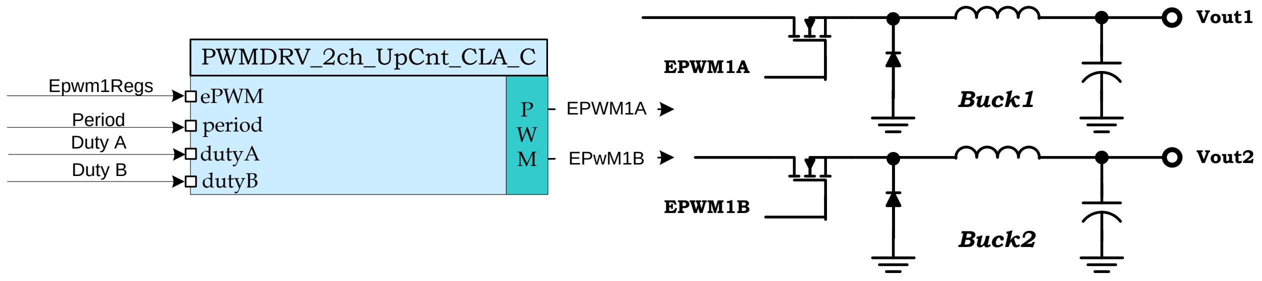
Where:

n	PWM peripheral number; PWM to be 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

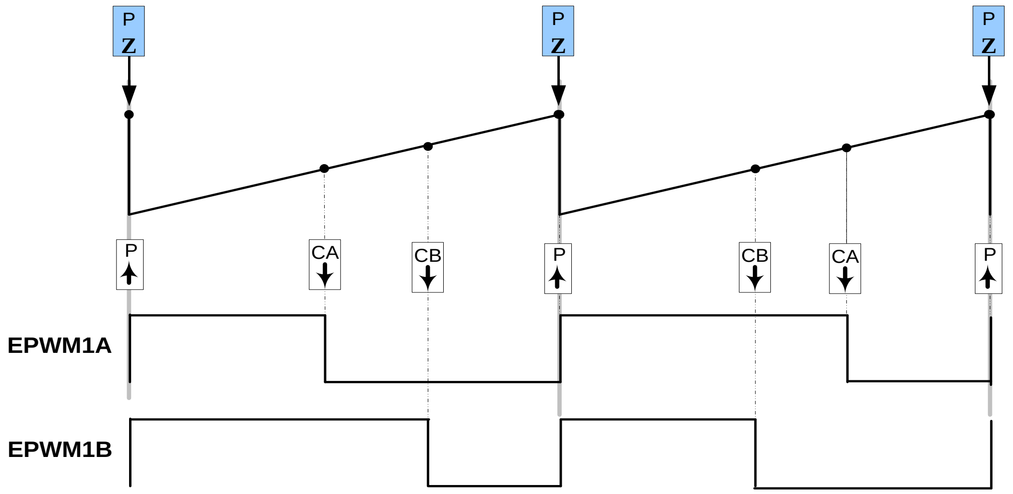
phase specifies the phase offset that is used when the PWM module is synchronized, this value only has meaning when the PWM is configured as a slave.

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 waveforms.

Detailed Description The following section explains how this module can be used to excite two buck power stages. To configure a 100kHz switching frequency with CPU operating at 60Mhz, the period value needed is (System Clock/Switching Frequency) = 600. Note that the period value should be decreased by 1, to take into account the up-count mode; this is done by the CNF function. Hence value of 600 needs to be supplied to the function.



Buck converter driven by PWMDRV_1ch module



PWM generation with the EPWM module.

Usage: This section explains how to use the PWMDRV_2ch_UpCnt_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Connect the PWMDRV_2ch_UpCnt_CLA_C macro within a CLA-Task.

```
interrupt void Cla1Task4(void) {
	...
	PWMDRV_2ch_UpCnt_CLA_C(EPwm1Regs, EPwm1Regs.TBPRD, dutyA, dutyB);
	...
}
```

Step 3 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
	...
	PWM_2ch_UpCnt_CF(1, 600 , 1, 0);
	...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** "Steps to use CLA DPlib" for details on how to modify this function to suit your application needs.

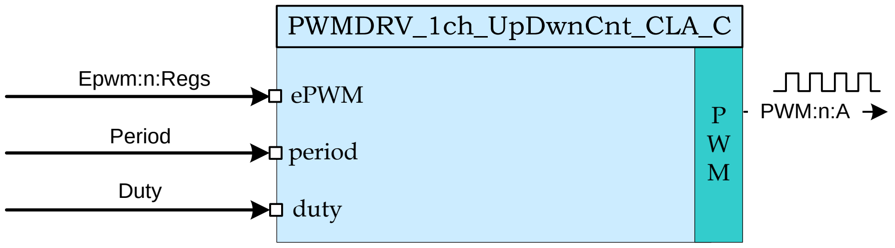
Module Parameters:

```
PWMDRV_2ch_UpCnt_CLA_C(EPwm1Regs, period, dutyA, dutyB);
```

Parameter Name	**Type**	**Description**	**Acceptable Range**
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven.	Device dependent
period	Input	Period by which the duty cycle is driven on the PWM module.	Uint16
dutyA	Input	Desired PWM module duty cycle for channel A.	float32 [0, 1)
dutyB	Input	Desired PWM module duty cycle for channel B.	float32 [0, 1)

PWMDRV_1ch_UpDwnCnt_CLA_C — *PWM Driver, Duty on chA centered at zero*

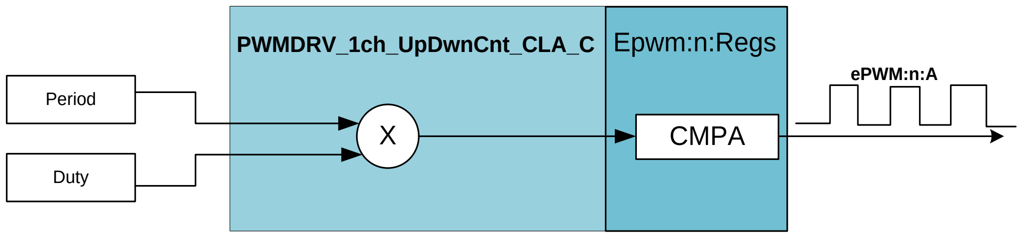
Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a duty on PWM channel A depending on the input value DutyA. The waveform is centered at the zero value of the up-down count time base of the PWM.



Macro File: `PWMDRV_CLA_C.h`

Peripheral Initialization File: `PWM_1ch_UpDwnCnt_Cnf.c`

Description: This macro provides the interface between DP library variables and the ePWM modules on C28x. The macro scales the float Duty value by the float Period value and stores it in the EPwm:n:Regs.CMPA.half.CMPA such as to give duty cycle control on channel A of the PWM module.



This macro is used in conjunction with the peripheral configuration file `PWM_1ch_UpDwnCnt_Cnf.c`. The file defines the function:

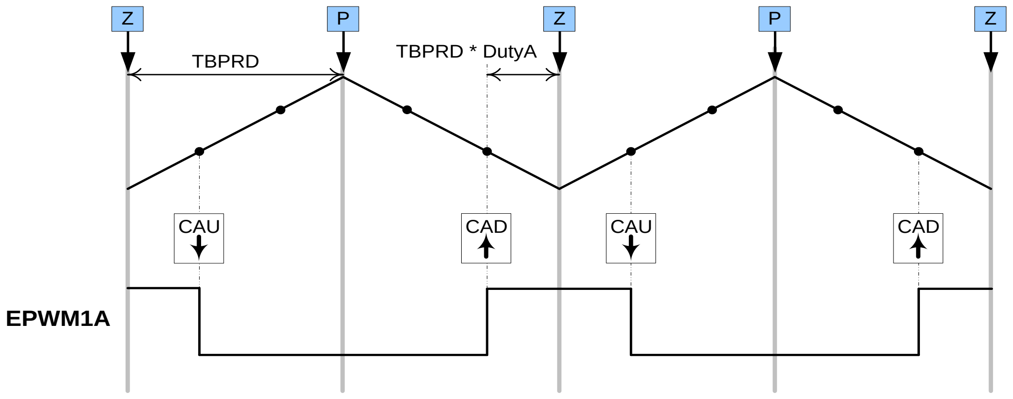
```
void PWM_1ch_UpDwnCnt_CNF(int16 n, Uint16 period, int16 mode,
    int16 phase)
```

Where:

n	PWM peripheral number; PWM to be configured in up-count mode
period	is the maximum value of the PWM counter
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-down 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 boost stage. If up-down count mode is used to configure a 100kHz switching frequency, at 60Mhz system clock, a period value of (System Clock/Switching Frequency) = 600 must be used for the CNF function.



PWM generation with the EPWM module.

Usage: This section explains how to use the PWMDRV_1ch_UpDwnCnt_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Connect the PWMDRV_1ch_UpDwnCnt_CLA_C macro within a CLA-Task.

```
interrupt void Cla1Task4(void) {
       ...
       PWMDRV_1ch_UpDwnCnt_CLA_C(EPwm1Regs, EPwm1Regs.TBPRD, duty);
       ...
}
```

Step 3 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
        ...
        PWM_1ch_UpDwnCnt_CNF(1, 600, 1, 0);
        ...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

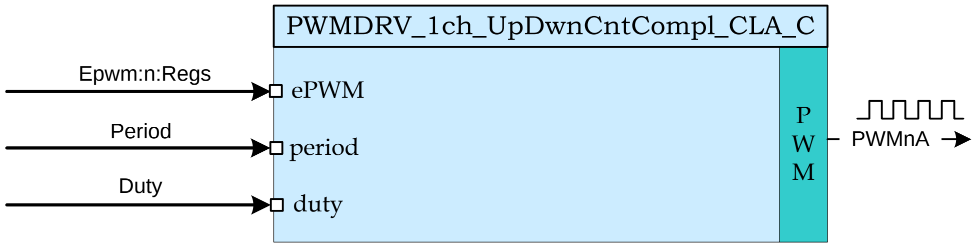
Module Parameters:

```
PWMDRV_1ch_UpDwnCnt_CLA_C(EPwm:n:Regs, period, duty);
```

Parameter Name	**Type**	**Description**	**Acceptable Range**
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven.	Device dependent
period	Input	Period by which the duty cycle is driven on the PWM module.	Uint16
duty	Input	Desired PWM module duty cycle for the given period.	float32 [0, 1)

PWMDRV_1ch_UpDwnCntCompl_CLA_C *PWM Driver, chA Duty centered at period*

Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a duty on PWM channel A depending on the value of the input variable Duty. The waveform is centered at the period value of the up-down count time base of the PWM.



Macro File: `PWMDRV_CLA_C.h`

Peripheral Initialization File: `PWM_1ch_UpDwnCntCompl_Cnf.c`

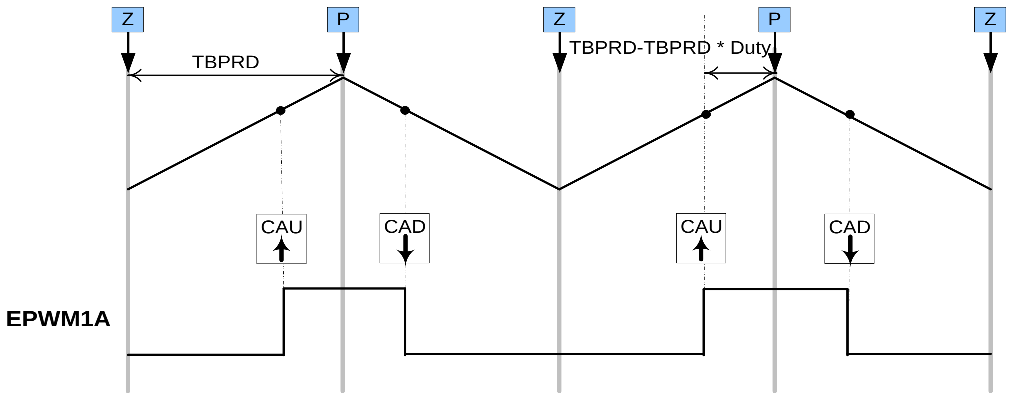
Description: This macro provides the interface between DP library variables and the ePWM modules on C28x. The macro scales the float Duty value by the Period value and stores this value in the EPwm:n:Regs.CMPA.half.CMPA such as to give duty cycle control on channel A of the PWM module. This macro is used in conjunction with the peripheral configuration file `PWM_1ch_UpDwnCntCompl_Cnf.c`. The file defines the function:

```
void  PWM_1ch_UpDwnCntCompl_CNF(int16  n,  Uint16  period,  int16
   mode, int16 phase)
```

Where:

n	PWM peripheral number; PWM to be configured in up-count mode
period	is the maximum value of the PWM counter
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-down 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 boost stage. Up-down count mode is used, hence to configure a 100kHz switching frequency, at 60Mhz system clock a period value of (System Clock/Switching Frequency) = 600 must be used for the CNF function.



PWM generation with the EPWM module.

Usage: This section explains how to use the PWMDRV_1ch_UpDwnCntCompl_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Connect the PWMDRV_1ch_UpDwnCntCompl_CLA_C macro within a CLA-Task.

```
interrupt void Cla1Task4(void) {
        ...
        PWMDRV_1ch_UpDwnCntCompl_CLA_C(EPwm1Regs, EPwm1Regs.TBPRD, duty);
        ...
}
```

Step 3 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
        ...
        PWM_1ch_UpDwnCntCompl_CNF(1, 600 , 1, 0);
        ...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Module Parameters:

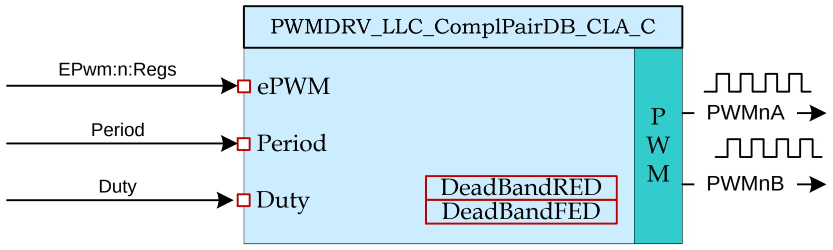
```
PWMDRV_1ch_UpDwnCntCompl_CLA_C(EPwm:n:Regs, period, duty);
```

Parameter Name	Type	Description	Acceptable Range
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven.	Device dependent
period	Input	Period by which the duty cycle is driven on the PWM module.	Uint16
duty	Input	Desired PWM module duty cycle for the given period.	float32 [0, 1)

PWMDRV_LLC_ComplPairDB_CLA_C

PWM Driver for compl. PWM, period modulation

Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a duty on PWM channel A, and the complimentary PWM on channel B with dead-band applied to both channels. The macro uses the dead-band module inside the PWM peripheral to generate the complimentary waveforms. This module also allows for period modulation.



Macro File: `PWMDRV_CLA_C.h`

Peripheral Initialization File: `PWM_ComplPairDB_Cnf.c`

Description: This macro provides the interface between DP library variables and the ePWM modules on C28x. The macro scales the normalized float Period value by 2^{10}. A Period input of 0.0 refers to an infinite frequency whereas a period of 1.0 refers to a frequency of the CPU clock frequency divided by 2^{10}. This value is the new PWM period and is stored in the EPwm:n:Regs.TBPRD register. Any other frequency can be found by the below equation:

$$f_{PWM} = \frac{f_{CPU}}{period * 2^{10}}$$

The Duty value is then scaled by the new period and stored in the Epwm:n:Regs.CMPA.half.CMPA register; it is used to generate the signal source for channel A signal. Half of the CMPA value is stored in the EPwm:n:Regs.CMPB register. The corresponding configuration file has the dead-band module configured to output the complimentary waveform on channel B. Dead-band is applied to both channels. 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, Uint16 period, int16 mode,
int16 phase)
```

Where:

n	is the PWM peripheral number to be configured in up-count mode
period	is the initial period of the PWM peripheral.

$$f_{PWM} = \frac{f_{CPU}}{period}$$

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 and the dead-band sub-module to output complimentary PWM waveforms with dead-band applied. 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 channel A of the PWM peripheral and a complementary active low duty on channel B.

The configuration function only configures the dead-band at initialization time. However, it may be needed to change the dead-band during operation. This can be done by calling the function

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

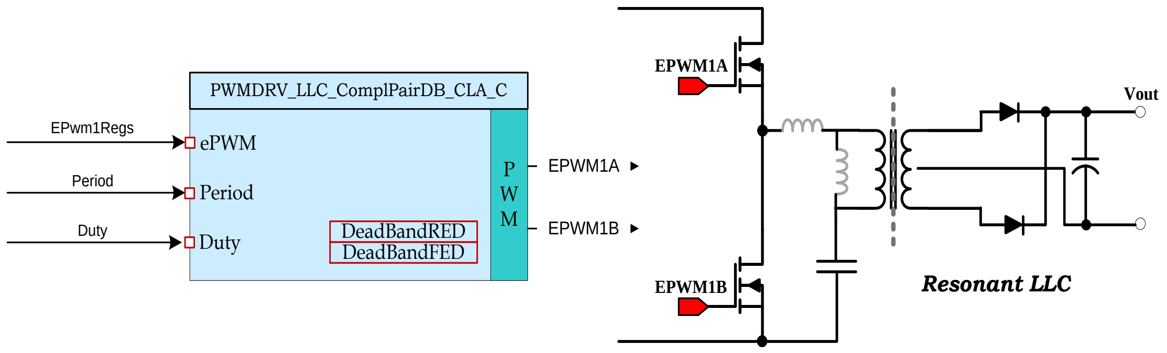
Where:

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

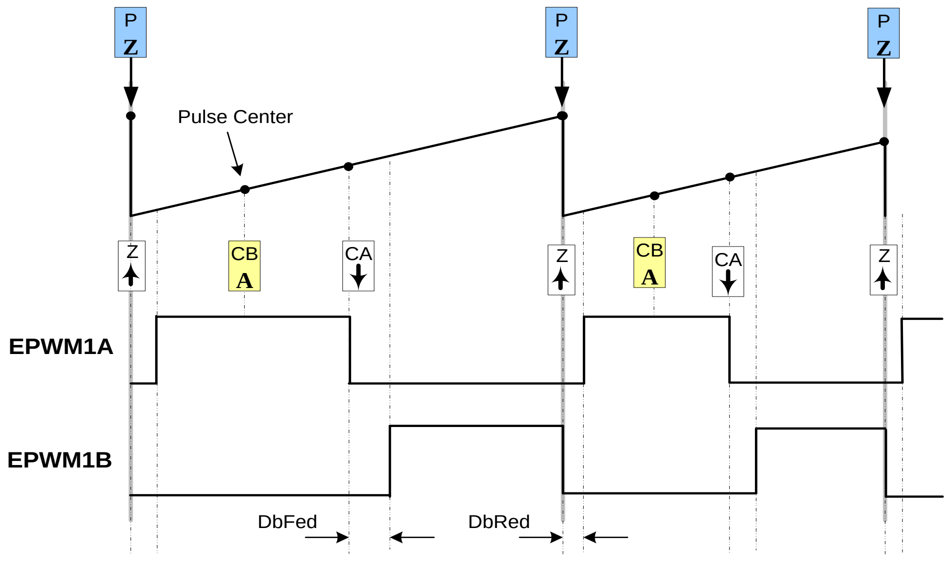
Alternatively, a macro is provided to update the dead-band if the update needs to be done at a faster rate, inside the ISR. The dead-band update macro is named PWMDRV_LLC_ComplPairDB_CLA_C_UpdateDB.

Detailed Description

The following section explains how this module can be used to excite a resonant LLC power stage which uses two NFET's. (Please note this module is specific to Resonant LLC power stage using NPN transistors only). The function configures the PWM peripheral in up-count mode. In order to configure a 100kHz switching frequency for the PWM in up-count mode when CPU is operating at 60MHz, the Period value needed is (System Clock/Switching Frequency) = 600. The TBPRD is stored with a value of 600-1, to take the up-count mode into account by the CNF function itself.



Resonant LLC converter driven by PWMDRV_LLC_ComplPairDB_CLA_C module



PWM generation for PWMDRV_LLC_CompPairDB_CLA Macro

Usage: This section explains how to use the PWMDRV_LLC_ComplPairDB_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Connect the PWMDRV_LLC_ComplPairDB_CLA_C macro within a CLA-Task. If the dead-band will be updated within the control loop, add calls to the dead-band update macro.

```
interrupt void Cla1Task4(void) {
        ...
        PWMDRV_LLC_ComplPairDB_CLA_C(EPwm1Regs, EPwm1Regs.TBPRD, duty);
        ...
        PWMDRV_LLC_ComplPairDB_CLA_C_UpdateDB(Epwm1Regs, dbRed, dbFed);
        ...
}
```

Step 3 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
        ...
        PWM_ComplPairDB_CNF(1, 600, 1, 0);
        ...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Module Parameters:

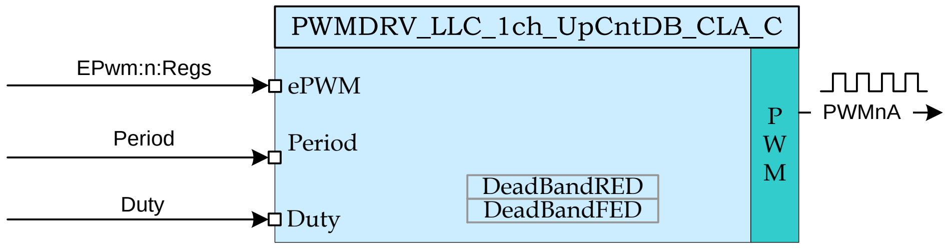
```
PWMDRV_LLC_ComplPairDB_CLA_C(EPwm:n:Regs, period, duty);

PWMDRV_LLC_ComplPairDB_CLA_C_UpdateDB(Epwm:n:Regs, dbRed, dbFed);
```

Parameter Name	**Type**	**Description**	**Acceptable Range**
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven.	Device dependent
period	Input	Period by which the duty cycle is driven on the PWM module.	Uint16
duty	Input	Desired PWM module duty cycle for the given period.	float32 [0, 1)
dbRed	Input	Dead-band value for rising edge delay.	Uint16
dbFed	Input	Dead-band value for falling edge delay.	Uint16

PWMDRV_LLC_1ch_UpCntDB_CLA_C

PWM Driver for chA PWM, edge shift, period mod.

Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a duty on PWM channel A, with edge shifting. The module uses the dead-band module inside the PWM peripheral along with software to generate the rising edge shift (RES) and falling edge shift (FES). This module also allows for period modulation.



Macro File: `PWMDRV_CLA_C.h`

Peripheral Initialization File: `PWM_1ch_UpCntDB_Cnf.c`

Description: This macro provides the interface between DP library variables and the ePWM modules on C28x. The macro loads the Period value to the EPwm:n:Regs.TBPRD register (the PWM period register). The Duty value is scaled by the Period and stored in the EPwm:n:Regs.CMPA.half.CMPA register. Subtracting the EPwm:n:Regs.DBFED register value from the EPwm:n:Regs.CMPA.half.CMPA register, the falling edge is advanced through software. The rising edge is delayed by the EPwm:n:Regs.DBRED register value using the dead-band module. Finally, half of the EPwm:n:Regs.CMPA.half.CMPA register is copied to the EPwm:n:Regs.CMPB register. This macro must be used in conjunction with the peripheral configuration file `PWM_1ch_UpCntDB_Cnf.c`. The file defines the function:

```
void PWM_1ch_UpCntDB_CNF(int16 n, int16 period, int16 mode, int16
phase)
```

Where:

n	PWM Peripheral number, PWM to be configured in up-count mode
period	is the maximum value of the PWM counter
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 and the dead-band sub-module of PWM channel A. The module outputs an active high duty on channel A of the PWM peripheral. The configuration function only configures the dead-band at initialization time. However, it may be needed to change the dead-band during operation. This can be done by calling the following function:

```
void PWM_1ch_UpCntDB_UpdateDB (int16 n, int16 DbRed, int16 DbFed)
```

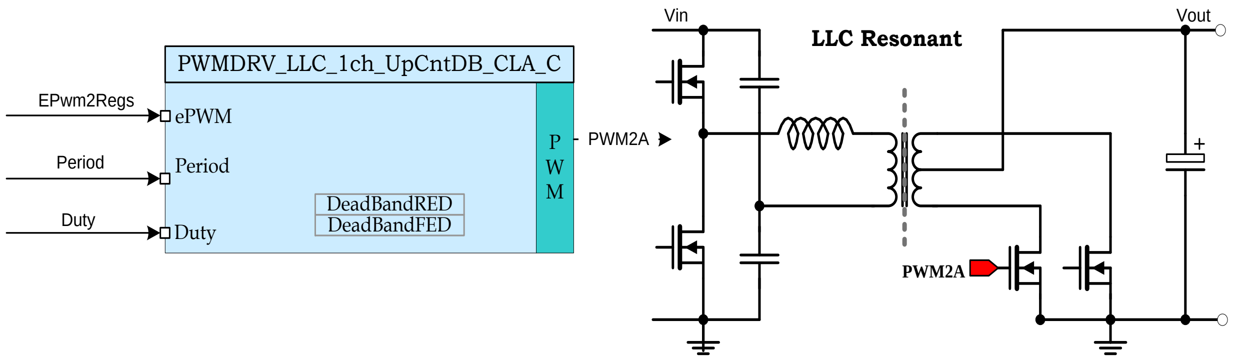
Where:

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

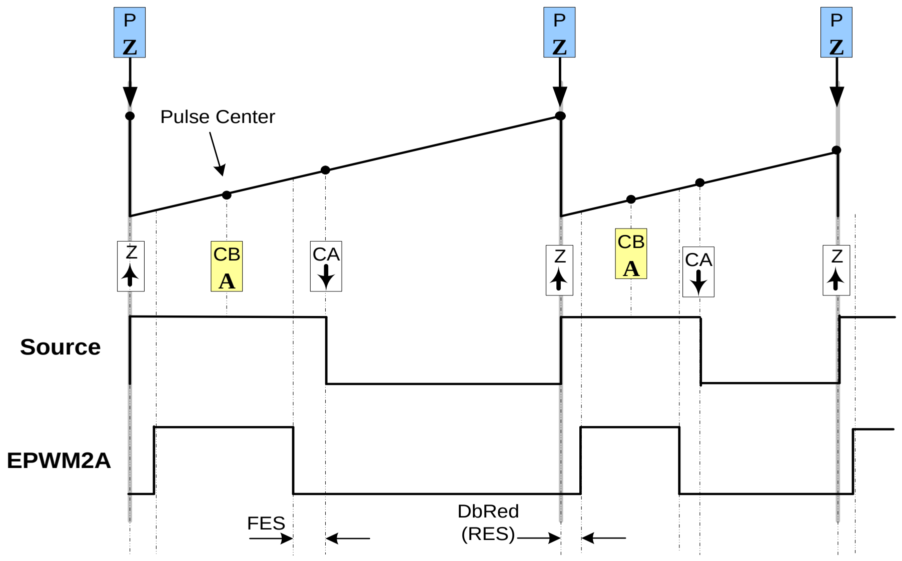
Alternatively, a macro is provided to update the dead-band if the update needs to be done at a faster rate; inside the ISR. The dead-band update macro is called `PWMDRV_LLC_1ch_UpCntDB_CLA_C_UpdateDB`.

Detailed Description

The following section explains how this module can be used to excite one of the Synchronous Rectifier legs of an LLC Resonant power stage (Please note this module is specific to using NPN transistors only). The function configures the PWM peripheral in up-count mode. In order to configure a 100kHz switching frequency when CPU is operating at 60Mhz, the period value needed is (System Clock/Switching Frequency) = 600. The TBPRD is stored with a value of 600-1, to take the up-count mode into account; this is done by the CNF function.



LLC Resonant converter Synchronous Rectifier leg driven by PWMDRV_LLC_1ch_UpCntDB module



PWM generation for PWMDRV_LLC_1ch_UpCntDB Macro

Usage: This section explains how to use the PWMDRV_LLC_1ch_UpCntDB_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Connect the PWMDRV_LLC_1ch_UpCntDB_CLA_C macro within a CLA-Task. If the dead-band will be updated within the control loop, add calls to the dead-band update macro.

```
interrupt void Cla1Task4(void) {
        ...
        PWMDRV_LLC_1ch_UpCntDB_CLA_C(EPwm1Regs, period, duty);
        ...
        PWMDRV_LLC_1ch_UpCntDB_CLA_C_UpdateDB(Epwm1Regs, dbRed, dbFed);
        ...
}
```

Step 3 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
        ...
        PWM_1ch_UpCntDB_CNF(1, 600, 1, 0);
        ...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** "Steps to use CLA DPlib" for details on how to modify this function to suit your application needs.

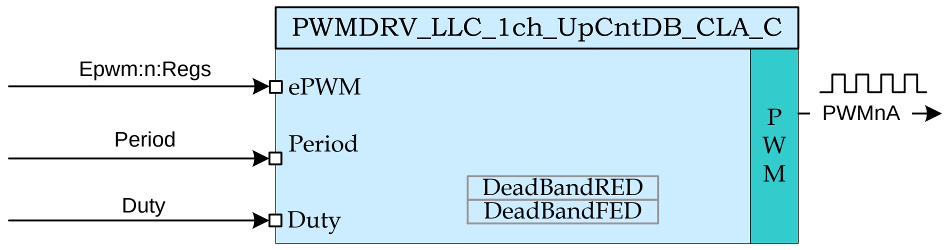
Module Parameters:

```
PWMDRV_LLC_1ch_UpCntDB_CLA_C(EPwm:n:Regs, period, duty);

PWMDRV_LLC_1ch_UpCntDB_CLA_C_UpdateDB(Epwm:n:Regs, dbRed, dbFed);
```

Parameter Name	Type	Description	Acceptable Range
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven.	Device dependent
period	Input	New period for the PWM module.	Uint16
duty	Input	Desired PWM module duty cycle for the given period.	float32 [0, 1)
dbRed	Input	Dead-band value for rising edge delay.	Uint16
dbFed	Input	Dead-band value for falling edge delay.	Uint16

PWMDRV_LLC_1ch_UpCntDB_Compl_CLA_C *Driver, chA PWM, edge shift, period mod.*

Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a duty on PWM channel A, with edge shifting. The module uses the dead-band module inside the PWM peripheral along with software to generate the rising edge shift (RES) and falling edge shift (FES). This module also allows for period modulation. This module generates a signal complementary to the one generated by `PWMDRV_LLC_1ch_UpCntDB_CLA_C.h`.



Macro File: `PWMDRV_CLA_C.h`

Peripheral Initialization File: `PWM_1ch_UpCntDB_Compl_Cnf.c`

Description: This macro provides the interface between DP library variables and the ePWM modules on C28x. The macro first loads the Period value into the EPwm:n:Regs.TBPRD register (the PWM period register). The Duty value is scaled by the Period value and subtracted from the initial Period value. The difference is stored in the EPwm:n:Regs.CMPA.half.CMPA register. Subtracting the EPwm:n:Regs.DBFED register value from the PWM period register advances the falling edge shift. The change in the PWM period register is corrected for during each PWM synchronization event. No conflicts are generated since both the falling edge advancement amount and the PWM duty cycle should not exceed 50% of the period. The rising edge is delayed by the value specified in EPwm:n:Regs.DBRED using the dead-band module. This macro must be used in conjunction with the peripheral configuration file `PWM_1ch_UpCntDB_Compl_Cnf.c`. The file defines the function:

```
void PWM_1ch_UpCntDB_Compl_CNF(int16 n, int16 period, int16 mode,
    int16 phase)
```

Where:

n	PWM peripheral number; PWM to be configured in up-count mode
period	is the maximum value of the PWM counter

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 with the dead-band sub-module to output PWM channel A, with dead-band applied. The module outputs an active high duty on channel A of the PWM peripheral. This configuration provides a duty complementary to that configured in `PWM_1ch_UpCntDB_Cnf.c`.

The configuration function only configures the dead-band at initialization time. However, it may be necessary to change the dead-band during operation. This can be done by calling the function:

```
void PWM_1ch_UpCntDB_Compl_UpdateDB (int16 n, int16 DbRed, int16
    DbFed)
```

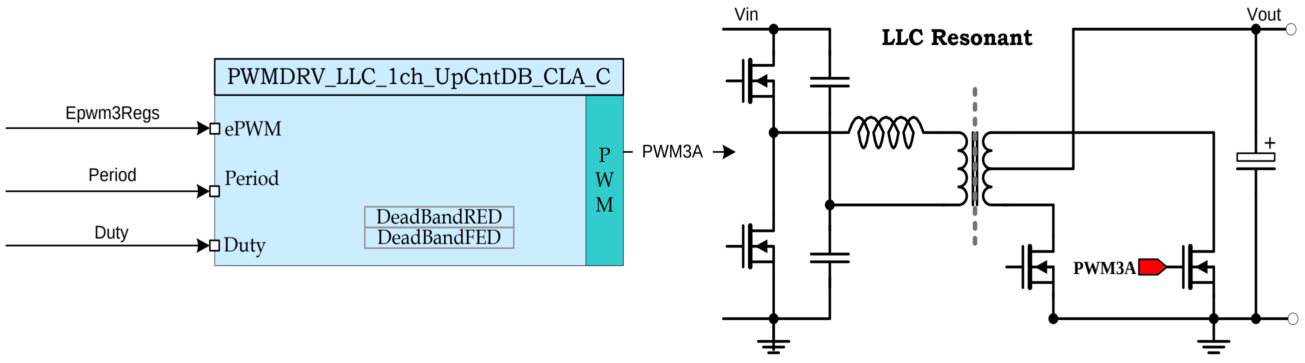
Where:

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

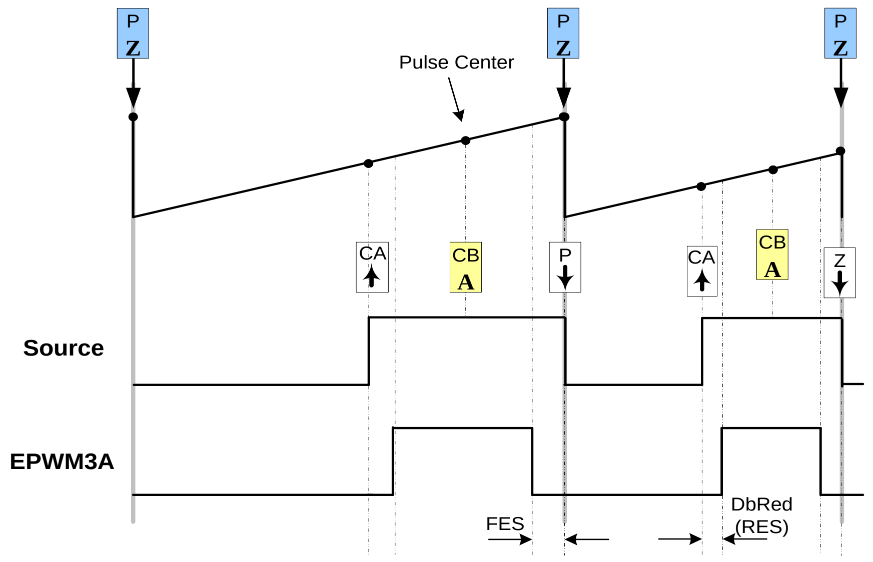
Alternatively, a macro is provided to update the dead-band if the update needs to be done at a faster rate; inside the ISR. The dead-band update macro is called: `PWMDRV_LLC_1ch_UpCntDB_Compl_CLA_C_UpdateDB`.

Detailed Description

The following section explains how this module can be used to excite one of the Synchronous Rectifier legs of an LLC Resonant power stage (Please note this module is specific to using NPN transistors only). The function configures the PWM peripheral in up-count mode. In order to configure a 100kHz switching frequency when CPU is operating at 60Mhz, the period value needed is (System Clock/Switching Frequency) = 600. The TBPRD is stored with a value of 600-1, to take the up-count mode into account; this is done by the CNF function.



LLC Resonant converter Synchronous Rectifier leg driven by PWMDRV_LLC_1ch_UpCntDB_Compl module



PWM generation for PWMDRV_LLC_1ch_UpCntDB_Compl Macro

Usage: This section explains how to use the PWMDRV_LLC_1ch_UpCntDB_Compl_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Connect the PWMDRV_LLC_1ch_UpCntDB_Compl_CLA_C macro within a CLA-Task. If the dead-band will be updated within the control loop, add calls to the dead-band update macro.

```
interrupt void Cla1Task4(void) {
        ...
        PWMDRV_LLC_1ch_UpCntDB_Compl_CLA_C(EPwm1Regs, period, duty);
        ...
        PWMDRV_LLC_1ch_UpCntDB_Compl_CLA_C_UpdateDB(Epwm1Regs, dbRed, dbFed);
        ...
}
```

Step 3 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
        ...
        PWM_1ch_UpCntDB_Compl_CNF(1, 600, 1, 0);
        ...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

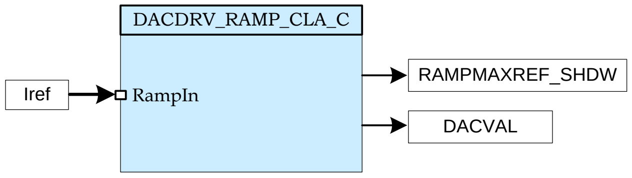
Module Parameters:

```
        PWMDRV_LLC_1ch_UpCntDB_Compl_CLA_C(EPwm:n:Regs, period, duty);

        PWMDRV_LLC_1ch_UpCntDB_Compl_CLA_C_UpdateDB(Epwm:n:Regs, dbRed, dbFed);
```

Parameter Name	Type	Description	Acceptable Range
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven.	Device dependent
period	Input	New period for the PWM module.	Uint16
duty	Input	Desired PWM module duty cycle for the given period.	float32 [0, 1)
dbRed	Input	Dead-band value for rising edge delay.	Uint16
dbFed	Input	Dead-band value for falling edge delay.	Uint16

DACDRV_RAMP_CLA_C — *DAC Ramp Driver Interface*

Description: This module controls DAC and ramp generator to control a power stage in peak current mode control (PCMC) using on-chip ramp generator for slope compensation.



Macro File: `DACDRV_RAMP_CLA_C.h`

Peripheral Initialization File: `DAC_Cnf.c`

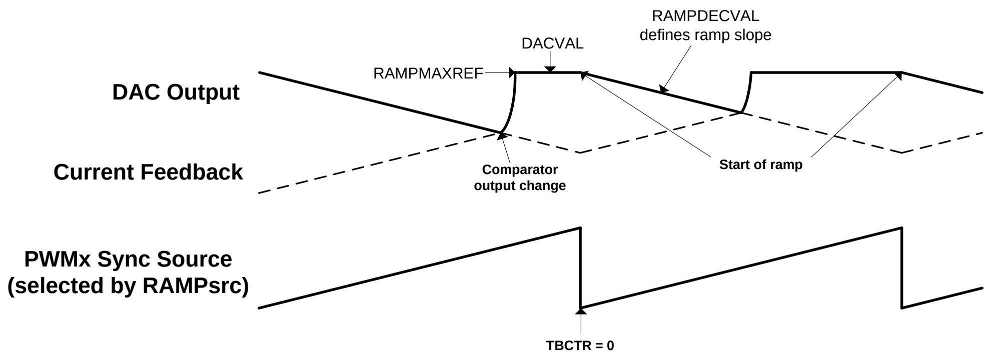
Description: This module forms the interface between the control software and on-chip DAC and ramp generator module. The macro converts the DACDRV_RAMP_In value to an appropriate 16-bit value (*RAMPMAXREF*), which is the starting value of the RAMP used for slope compensation. This module also drives an appropriate 10-bit value to the *DACVAL* register, which can be used if slope compensation is not needed or provided externally. This macro is used in conjunction with the peripheral configuration file `DAC_Cnf.c`. The file defines the function:

```
void DacDrvCnf(int16 n, int16 DACval, int16 DACsrc, int16
RAMPsrc, int16 Slope_initial)
```

Where:

n	Comparator target module number.
DACval	Provides the DAC value when internal ramp is not used.
DACsrc	Selects DACval or internal ramp as DAC source
RAMPsrc	Selects source to synchronize internal ramp used for slope compensation.
Slope_initial	Initial slope value for slope compensation. This value can be changed any time during execution.

Detailed Description As seen below the PWM sync signal can start the ramp at a known point in the switching cycle. Here this happens at TBCTR = 0. When feedback current equals reference current generated by the DAC and RAMP module it resets the comparator output, which then causes a reset of the ramp to its starting value. The slope of this ramp can be programmed using RAMPDECVAL register. When internal ramp generator is not used (by selecting DACsrc as DACval for DacDrvCnf function), DAC output is a constant reference set by the DACVAL register value.



DAC and ramp for PCMC control of a power stage

Usage: This section explains how to use the DACDRV_RAMP_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `DACDRV_RAMP_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Connect the DACDRV_RAMP_CLA_C macro within a CLA-Task. If the dead-band will be updated within the control loop, add calls to the dead-band update macro.

```
interrupt void Cla1Task4(void) {
	...
	DACDRV_RAMP_CLA_C(Comp1Regs, RampIn);
	...
}
```

Step 3 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
	...
	DacDrvCnf(1, 1280, 1, 2, slope)
	...
}
```

Step 7 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Module Parameters:

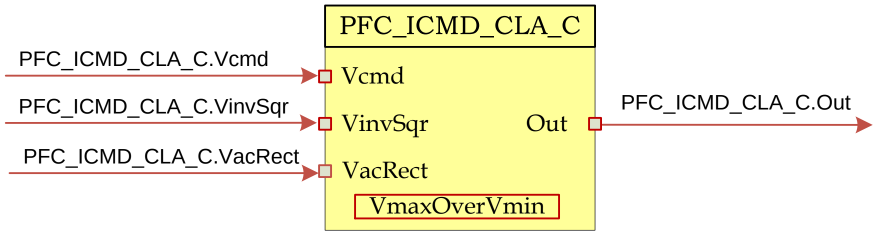
```
    DACDRV_RAMP_CLA_C(Comp:n:Regs, RampIn);
```

Parameter Name	**Type**	**Description**	**Acceptable Range**
Comp:n:Regs	Input	Comparator module registers. The :n: indicates which Comparator is driven.	Device dependent
RampIn	Input	Current reference value	Uint16

5.4. Application Specific

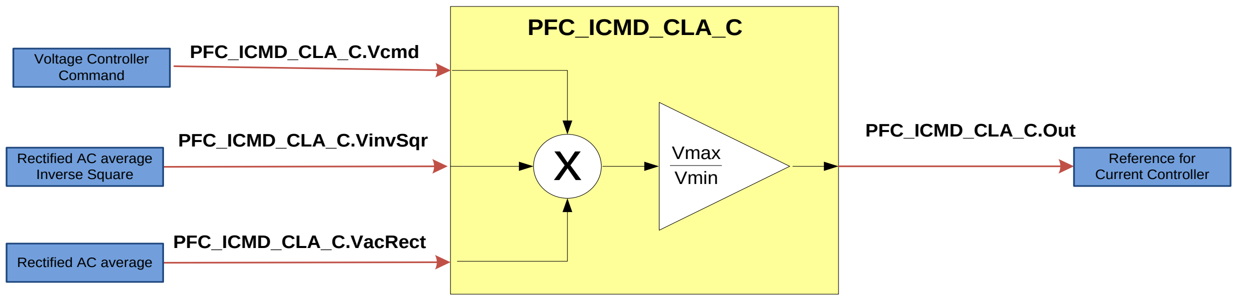
PFC_ICMD_CLA_C	*Current Command for Power Factor Correction using CLA*

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



Macro File: `PFC_ICMD_CLA_C.h`

Technical: This software module performs a computation of the current command for 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_C block then generates an output command profile that is half-sinusoidal, with the 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 variables (Vcmd, VinvSqr and VacRect) must be normalized float. The module multiplies these values together and then scales them by the VmaxOverVmin factor. The result is also in normalized float format. All input and output variables are stored in internal memory in a PFC_ICMD_CLA_C structure. A PFC stage is typically designed to work over a range of AC line conditions. VmaxOverVmin is the ratio of maximum over minimum voltage the PFC stage is designed for represented in float format. The following diagram illustrates the math functions in this block.



Usage: This section explains how to use the PFC_ICMD_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PFC_ICMD_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare PFC_ICMD_CLA_C structures in the `{ProjectName}-CLA_Tasks.cla` file. Specify structure variables and CLA memory locations.

```
/* DPlib variables & memory locations */
#pragma DATA_SECTION(pfc_icmd1, "Cla1ToCpuMsgRAM")
PFC_ICMD_CLA_C pfc_icmd1;
```

Step 3 Add shared declaration to `{ProjectName}-CLA_Shared.h` file if access is shared between CPU and CLA.

```
/* DPlib variables */
// Declare net variables shared between CPU and CLA here
extern volatile PFC_ICMD_CLA_C pfc_icmd1;
```

Step 4 Initialize macro in CLA Task 8. Edit the task found in the `{ProjectName}-CLA_Tasks.cla` file to initialize the macro coefficients.

```
interrupt void Cla1Task8(void) {
	...
	PFC_ICMD_CLA_C_INIT(pfc_icmd1);
	// The example assumes PFC Stage designed for 230VAC to 90VAC
	// Hence VmaxOverVmin = (float32) (230.0 / 90.0) = 2.5555
	pfc_icmd1.VmaxOverVmin = (float32) 2.555;
	...
}
```

Step 5 Connect the PFC_ICMD_CLA_C controller within a CLA-Task.

```
interrupt void Cla1Task4(void) {
	...
	pfc_icmd1.Vcmd = Command;
	pfc_icmd1.VinvSqr = VinvS;
	pfc_icmd1.VacRect = Vrect;
	PFC_ICMD_CLA_C(pfc_icmd1);
	Out = pfc_icmd1.Out;
	...
}
```

Step 4 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
	...
	PWM_1ch_CNF(1, 600, 1, 0);
	...
}
```

Step 5 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Object Definition:

```
/**
 * Current command PFC structure
 */
typedef struct {
      float32 Vcmd;
      float32 VinvSqr;
      float32 VacRect;
      float32 Out;
      float32 VmaxOverVmin;
} PFC_ICMD_CLA_C;
```

Special Constants and Data types

PFC_ICMD_CLA_C

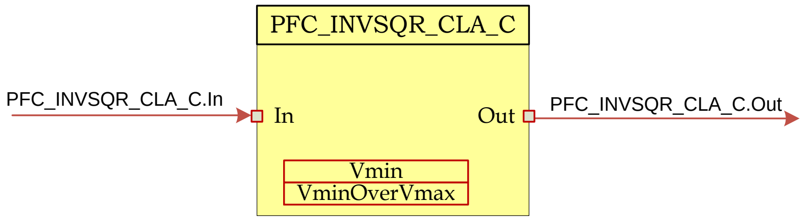
Structure defined to store module values. To create multiple instances of the module simply declare variables of type PFC_ICMD_CLA_C.

Module Structure Definition:

Parameter Name	Types	Description	Acceptable Range
Vcmd	Input	Voltage controller output.	float32 [0, 1)
VinvSqr	Input	PFC_INVSQR_CLA_C block output.	float32 [0, 1)
VacRect	Input	MATH_EMAVG_CLA_C block output.	float32 [0, 1)
Out	Output	Current loop reference.	float32 [0,1)
VmaxOverVmin	Input	Internal scaling factor	float32

PFC_INVSQR_CLA_C — *Inverse Square Math Block for Power Factor Correction*

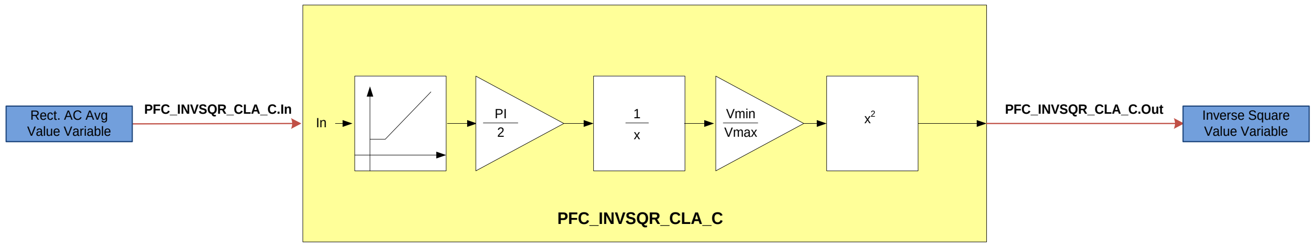
Description: This software module performs a reciprocal function on a scaled unipolar input signal and squares it.



Macro File: `PFC_INVSQR_CLA_C.h`

Technical: The input variable must be in normalized float format and copied into the PFC_INVSQR_CLA_C structure. The module scales and inverts this value. The result is also in normalized float format and stored in the Out variable of the structure. The module uses two internal data variables to specify the range and scaling, which are 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. VminOverVmax parameter is the ratio of minimum voltage over the maximum voltage the PFC stage is designed for represented in float. The Vmin parameter is equal or less than the minimum AC Line voltage that the PFC stage is designed for represented in a normalized float format. Note that Vmin value 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 the PFC_INVSQR_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PFC_INVSQR_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare PFC_INVSQR_CLA_C structures in the `{ProjectName}-CLA_Tasks.cla` file. Specify structure variables and CLA memory locations.

```
/* DPlib variables & memory locations */
#pragma DATA_SECTION(pfc_invsqr1, "Cla1ToCpuMsgRAM")
PFC_INVSQR_CLA_C pfc_invsqr1;
```

Step 3 Add shared declaration to `{ProjectName}-CLA_Shared.h` file if access is shared between CPU and CLA.

```
/* DPlib variables */
// Declare net variables shared between CPU and CLA here
extern volatile PFC_INVSQR_CLA_C pfc_invsqr1;
```

Step 4 Initialize macro in CLA Task 8. Edit the task found in the `{ProjectName}-CLA_Tasks.cla` file to initialize the macro coefficients.

```
interrupt void Cla1Task8(void) {
      ...
      PFC_INVSQR_CLA_C_INIT(pfc_icmd1);
      // The example assumes PFC Stage designed for 230VAC to 90VAC
      // The voltage feedback is designed for 400V
      // VmaxOverVmin = (float32) (90.0 / 230.0) = 2.5555
      pfc_invsqr1.VmaxOverVmin = (float32) 0.3913;
      // Vmin = (float32) (90.0 / 400.0) = 0.225
      pfc_invsqr1.Vmin = (float32) 0.225;
      ...
}
```

Step 5 Connect the PFC_INVSQR_CLA_C controller within a CLA-Task.

```
interrupt void Cla1Task4(void) {
      ...
      pfc_invsqr1.In = Command;
      PFC_INVSQR_CLA_C(pfc_invsqr1);
      Out = pfc_invsqr1.Out;
      ...
}
```

Step 4 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
	...
	PWM_1ch_CNF(1, 600, 1, 0);
	...
}
```

Step 5 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Object Definition:

```
/**
 * PFC_INVSQR_CLA_C structure
 */
typedef struct {
	float32 In;
	float32 Out;
	float32 Vmin;
	float32 VminOverVmax;
} PFC_INVSQR_CLA_C;
```

Special Constants and Data types

PFC_INVSQR_CLA_C

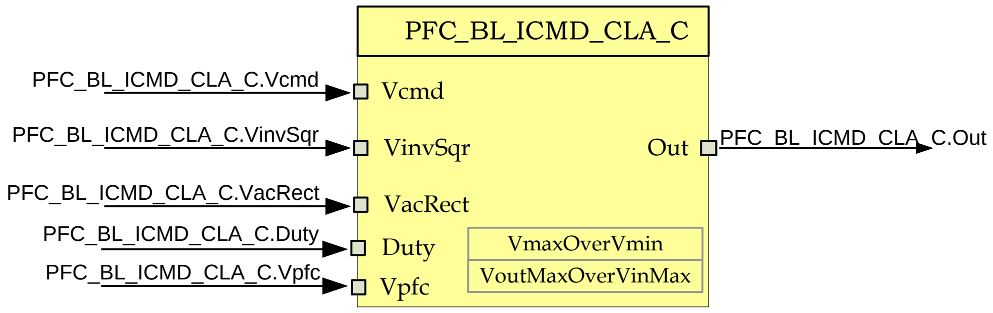
Structure defined to store module values. To create multiple instances of the module simply declare variables of type PFC_INVSQR_CLA_C.

Module Structure Definition:

Parameter Name	Types	Description	Acceptable Range
In	Input	Input voltage.	float32 [0, 1)
Out	Output	PFC_INVSQR_CLA_C output.	float32 [0, 1)
Vmin	Input	Ratio of minimum designed AC line the PFC stage over the max feedback voltage.	float32 [0, 1)
VminOverVmax	input	Ratio of minimum designed AC line voltage over maximum designed AC line voltage.	float32 [0,1)

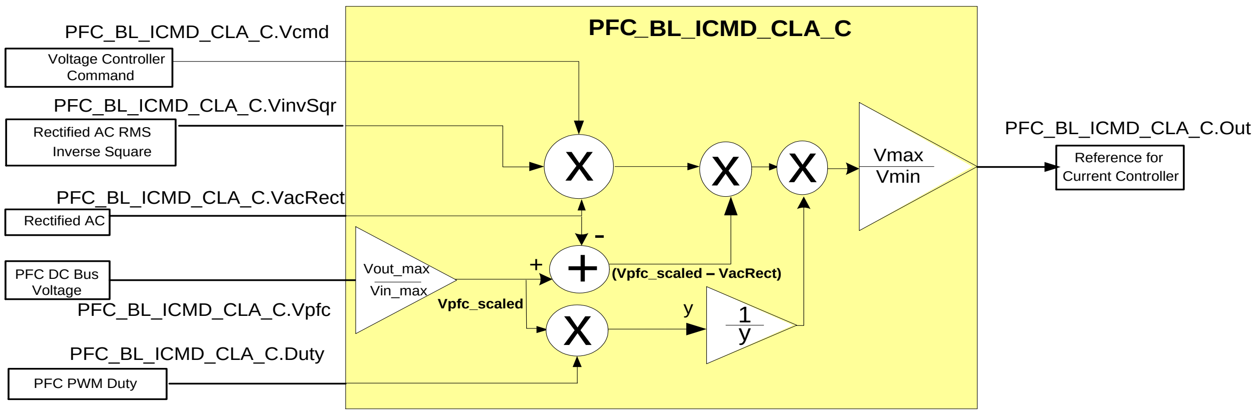
PFC_BL_ICMD_CLA_C — *Bridgeless PFC Current Command*

Description: This software module performs a computation of the current command for the Power Factor Correction (PFC) stage that uses PFC boost switch current sensing in order to control the PFC inductor current.



Macro File: `PFC_BL_ICMD_CLA_C.h`

Technical: This software module performs a computation of the current command for the power factor correction stage that uses boost PFC switch current sensing in order to implement PFC input current control. The inputs to the module are the inverse-squared-rms line voltage, the rectified line voltage, PFC PWM duty ratio, PFC dc bus voltage and the output of the voltage loop controller. The PFC_BL_ICMD_CLA_C block then generates an output command profile which is then connected to the current controller to produce the required inductor current. The inputs as well as the output are stored in a PFC_BL_ICMD_CLA_C structure. The module uses these inputs to implement a math function and then scales the result by multiplying it by a factor (which is also stored in the structure) VmaxOverVmin. A PFC stage is typically designed to work over a range of input and output voltages. VmaxOverVmin is the ratio of maximum over minimum AC input voltage the PFC stage is designed for, in float format. VoutMaxOverVinMax is the ratio of maximum output voltage over minimum AC input voltage in float format. The following diagram illustrates the math function operated on in this block.



Usage: This section explains how to use the PFC_BL_ICMD_CLA_C module.

Step 1 Add library header file and CLA shared header file to the {ProjectName}-Main.c file. Un-comment PFC_ICMD_CLA_C.h include found at the top of the DPlib.h file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare PFC_ICMD_CLA_C structures in the {ProjectName}-CLA_Tasks.cla file. Specify structure variables and CLA memory locations.

```
/* DPlib variables & memory locations */
#pragma DATA_SECTION(pfc_bl_icmd1, "Cla1ToCpuMsgRAM")
PFC_BL_ICMD_CLA_C pfc_bl_icmd1;
```

Step 3 Add shared declaration to {ProjectName}-CLA_Shared.h file if access is shared between CPU and CLA.

```
/* DPlib variables */
// Declare net variables shared between CPU and CLA here
extern volatile PFC_BL_ICMD_CLA_C pfc_bl_icmd1;
```

Step 4 Initialize macro in CLA Task 8. Edit the task found in the {ProjectName}-CLA_Tasks.cla file to initialize the macro coefficients.

```
interrupt void Cla1Task8(void) {
	...
	PFC_BL_ICMD_CLA_C_INIT(pfc_bl_icmd1);
	// The example assumes PFC Stage designed for 280VAC to 80VAC RMS
	// Max DC Bus voltage 450V
	// VmaxOverVmin = (float32) (280.0 / 80.0) = 3.5
	pfc_bl_icmd1.VmaxOverVmin = (float32) 3.5;
	// VoutMaxOverVinMin = (float32) (450 / (1.414 * 280) = 1.14
	pfc_bl_icmd1.VoutMaxOverVinMin = (float32) 1.14;
	...
}
```

Step 5 Connect the PFC_ICMD_CLA_C controller within a CLA-Task.

```
interrupt void Cla1Task4(void) {
	...
	pfc_bl_icmd1.Vcmd = Command;
	pfc_bl_icmd1.VinvSqr = VinvS;
	pfc_bl_icmd1.VacRect = Vrect;
	pfc_bl_icmd1.Duty = Duty;
	pfc_bl_icmd1.Vpfc = Vpfc;
	PFC_BL_ICMD_CLA_C(pfc_icmd1);
	Out = pfc_bl_icmd1.Out;
	...
}
```

Step 4 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
	...
	PWM_1ch_CNF(1, 600, 1, 0);
	...
}
```

Step 5 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** "Steps to use CLA DPlib" for details on how to modify this function to suit your application needs.

Object Definition:

```
/**
 * Current command PFC BL structure
 */
typedef struct {
	float32 Vcmd;
	float32 VinvSqr;
	float32 VacRect;
	float32 Duty;
	float32 Vpfc;
	float32 Out;
	float32 VmaxOverVmin;
	float32 VoutMaxOverVinMax;
} PFC_BL_ICMD_CLA_C;
```

Special Constants and Data types

PFC_BL_ICMD_CLA_C

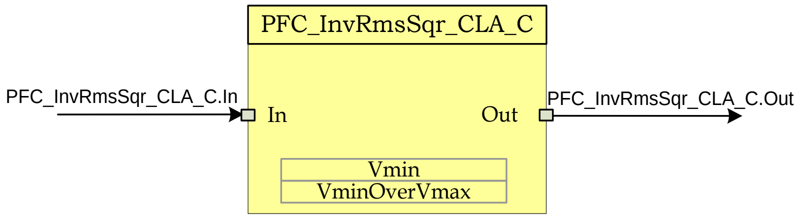
Structure defined to store module values. To create multiple instances of the module simply declare variables of type PFC_BL_ICMD_CLA_C.

Module Structure Definition:

Parameter Name	Types	Description	Acceptable Range
Vcmd	Input	Voltage controller output.	float32 [0, 1)
VinvSqr	Input	PFC_InvRmsSqr_CLA_C block output.	float32 [0, 1)
VacRect	Input	PFC rectified input voltage.	float32 [0, 1)
Duty	Input	PFC stage duty cycle	float32 [0, 1)
Vpfc	Input	PFC output voltage	float32 [0,1)
Out	Output	Current loop reference.	float32 [0,1)
VmaxOverVmin	Input	Internal scaling factor	float32
VoutMaxOverVinMax	Input	Internal scaling factor	float32

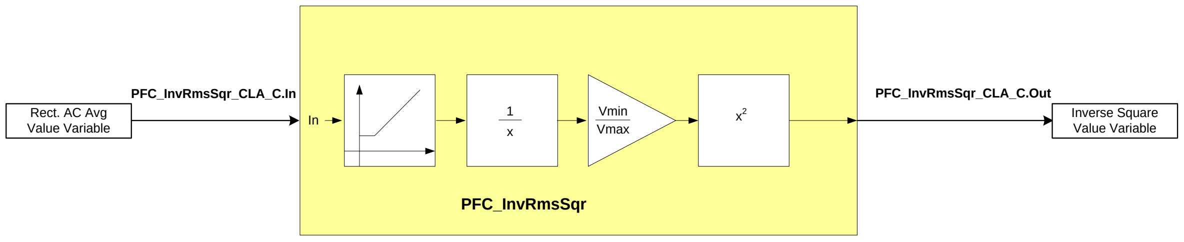
PFC_InvRmsSqr_CLA_C — *Inverse of the RMS Squared*

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



Macro File: `PFC_InvRmsSqr_CLA_C.h`

Technical: The module scales and inverts the float In value and stores the result in the Out variable. The module also uses two internal data variables to specify the range and scaling, which is dependent on the hardware design of the Power Factor Correction (PFC) stage. The four values are stored within a PFC_InvRmsSqr_CLA_C structure. A PFC stage is typically designed to operate over a range of AC line conditions. For example a normal operating range could be 85Vrms ~ 264Vrms. However, the range of AC line voltage used for signal scaling (in software implementation) must be wider than this operating range. For example, if the absolute maximum amplitude of 400V (= 283Vrms) for the AC voltage is scaled down (using resistor divider) to generate the full scale ADC input signal, the Vmax signal used for this module will be 400V. Also, the minimum rms voltage that is used to normalize the inverse rms signal output from this module must be lower than the selected minimum operating range of 85Vrms, i.e. about 80Vrms. This will allow the PFC to operate normally at 85Vrms and saturate the inverse signal outside this range, i.e., at 80Vrms. With these values identified, the parameter VminOverVmax structure parameter is the ratio of minimum over maximum voltage, which in this example is 80Vrms/283Vrms. The VminOverVmax and Vmin parameters are in float format. This is used for limiting the minimum input to the module, allowing proper scaling and saturation of the inverse signal. The module clamps the input signal to this minimum value to allow the PFC system to detect input under-voltage condition without causing overflows, which can cause undesirable 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 the PFC_InvRmsSqr_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PFC_InvRmsSqr_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare PFC_InvRmsSqr_CLA_C structures in the `{ProjectName}-CLA_Tasks.cla` file. Specify structure variables and CLA memory locations.

```
/* DPlib variables & memory locations */
#pragma DATA_SECTION(pfc_InvRmsSqr, "Cla1ToCpuMsgRAM")
PFC_InvRmsSqr_CLA_C pfc_InvRmsSqr;
```

Step 3 Add shared declaration to `{ProjectName}-CLA_Shared.h` file if access is shared between CPU and CLA.

```
/* DPlib variables */
// Declare net variables shared between CPU and CLA here
extern volatile PFC_InvRmsSqr_CLA_C pfc_InvRmsSqr;
```

Step 4 Initialize macro in CLA Task 8. Edit the task found in the `{ProjectName}-CLA_Tasks.cla` file to initialize the macro coefficients.

```
interrupt void Cla1Task8(void) {
      ...
      PFC_InvRmsSqr_CLA_C_INIT(pfc_InvRmsSqr);
      // The example assumes PFC Stage designed for 265VAC to 84VAC
      // voltage designed for 400V with a minimum voltage of 80VAC RMS
      // VminOverVmax = (float32) (80 / 283) = 0.283
      pfc_InvRmsSqr.VminOverVmax = (float32) 0.283;
      // Vmin <= 0.283;
      pfc_InvRmsSqr.Vmin = (float32) 0.225;
      ...
}
```

Step 5 Connect the PFC_InvRmsSqr_CLA_C controller within a CLA-Task.

```
interrupt void Cla1Task4(void) {
      ...
      pfc_InvRmsSqr.In = In;
      PFC_BL_InvRmsSqr_CLA_C(pfc_InvRmsSqr);
      Out = pfc_InvRmsSqr.Out;
      ...
}
```

Step 4 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
	...
	PWM_1ch_CNF(1, 600, 1, 0);
	...
}
```

Step 5 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** "Steps to use CLA DPlib" for details on how to modify this function to suit your application needs.

Object Definition:

```
/**
 * PFC_InvRmsSqr_CLA_C structure
 */
typedef struct {
	float32 In;
	float32 Out;
	float32 Vmin;
	float32 VminOverVmax;
} PFC_InvRmsSqr_CLA_C;
```

Special Constants and Data types

PFC_InvRmsSqr_CLA_C

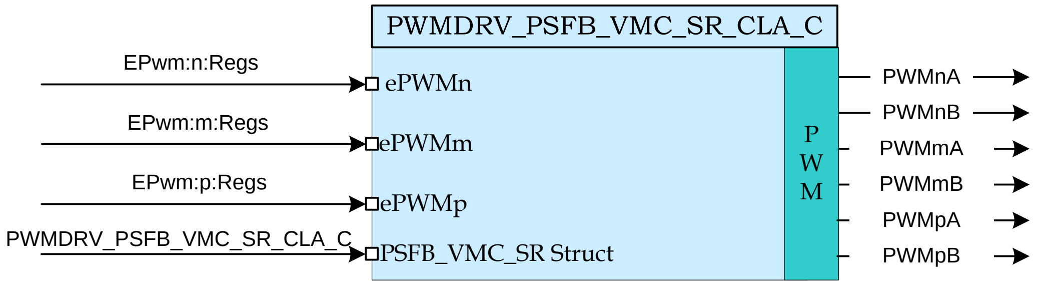
Structure defined to store module values. To create multiple instances of the module simply declare variables of type PFC_InvRmsSqr_CLA_C.

Module Structure Definition:

Parameter Name	Types	Description	Acceptable Range
In	Input	Input voltage.	float32 [0, 1)
Out	Input	PFC_InvRmsSqr_CLA_C block output.	float32 [0, 1)
Vmin	Input	Ratio of minimum designed AC line over maximum designed feedback voltage.	float32 [0, 1)
VminOverVmax	Input	Ratio of minimum over maximum voltage the PFC stage is designed for.	float32 [0, 1)

PWMDRV_PSFB_VMC_SR_CLA_C — *PWM Driver for VMC controlled PSFB Stage with SR*

Description: This module controls the PWM generators to control a phase shifted full bridge (PSFB) in voltage mode control (VMC) and also drives synchronous rectifiers (SR), if used. Additionally, this module offers control over dead-band amounts for switching signals in the two legs of the bridge.



Macro File: `PWMDRV_CLA_C.h`

Peripheral Initialization File: `PWM_PSFB_VMC_SR_Cnf.c`

Description: This module forms the interface between the control software and the device PWM pins. The macro scales the phase structure parameter by the PWM period value. The value is then stored in the EPwm:m:Regs.TBPHS and EPwm:p:Regs.TBPHS registers. The numbers :n: and :m: identify the PWM peripherals used to drive switches in the two legs while :p: identifies the PWM peripherals used to drive the SR switches. This macro is used in conjunction with the peripheral configuration file `PWM_PSFB_VMC_SR_CNF.c`. The file defines the function:

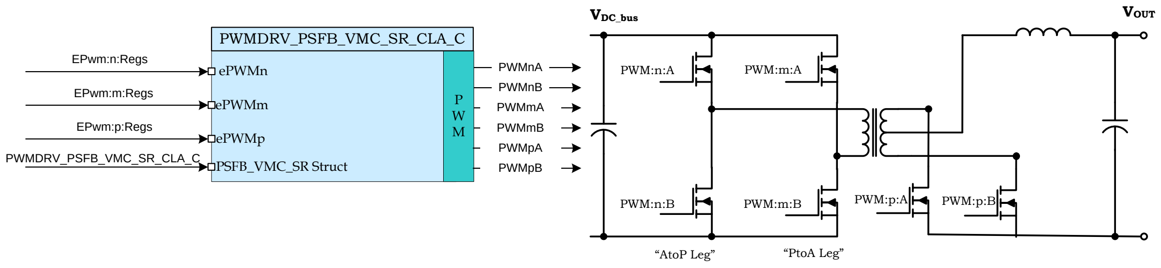
```
void  PWMDRV_PSFB_VMC_SR_CNF  (int16  n,  int16  period,  int16
SR_Enable, int16 Comp1_Prot)
```

Where:

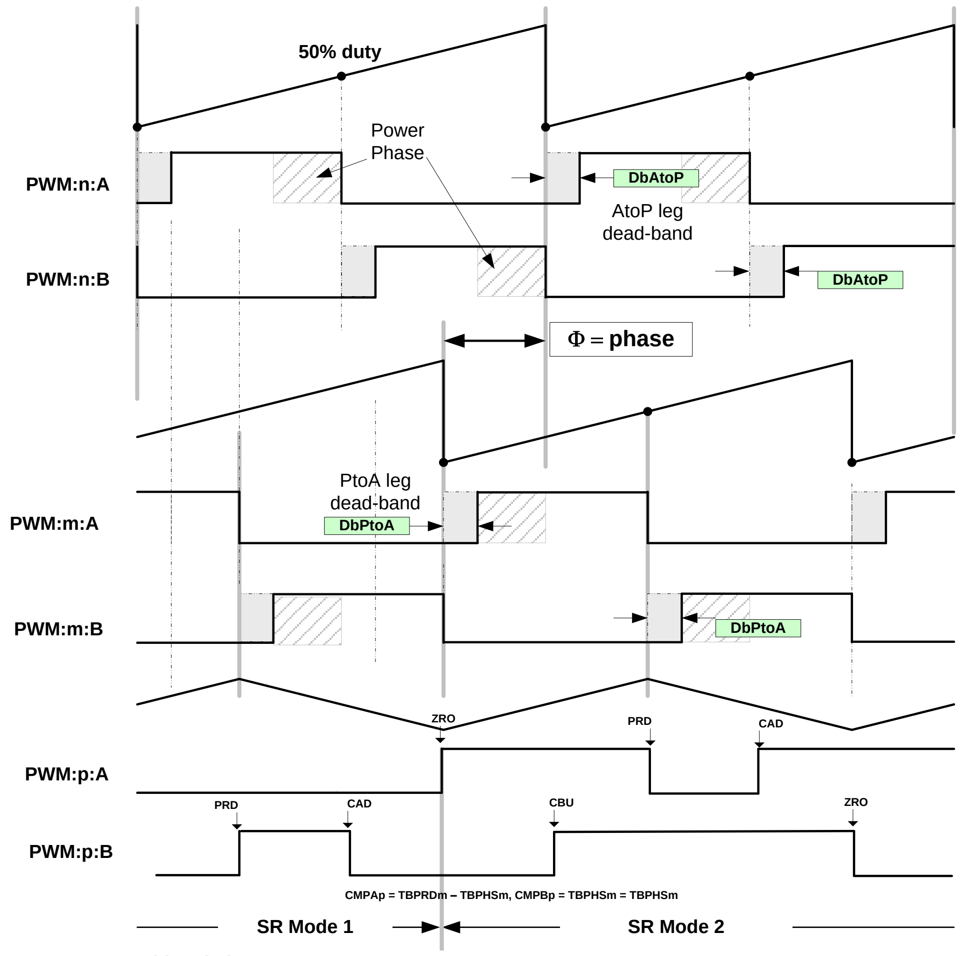
n	is the master PWM peripheral configured for driving switches in one leg of the full bridge. PWM m (n+1) is configured to work with synch pulses from PWM n module and drives switches in the other leg. PWM p (n + 3) drives SR switches if SR_Enable is 1.
period	is the maximum count value of the PWM timer
SR_Enable	This enables drive to SR switches using PWM p module. If a different PWM module is desired to be used for SR that module can be configured similar to PWM p module configuration in this file. Note that if a different PWM module is selected for SR it should be a module number greater than n.
Comp1_Prot	Enables catastrophic protection based on on-chip comparator1 and DAC.

Detailed Description

The following section explains with help of a timing diagram, how the PWM is configured to generate waveforms to drive a PSFB power 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



Abbreviations:
CMPA/Bx – Compare A/B register value for PWM module 'x'
CAU – Time base counter = Compare A event when the timer is counting UP
CAD – Time base counter = Compare A event when the timer is counting DOWN
PRD – Time base counter = Period event
ZRO – Time base counter = Zero event

Phase shifted PWM generation with the EPWM module.

Usage: This section explains how to use the PWMDRV_PSFB_VMC_SR_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `PWMDRV_PSFB_VMC_SR_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare PWMDRV_PSFB_VMC_SR_CLA_C structures in the `{ProjectName}-CLA_Tasks.cla` file. Specify structure variables and CLA memory locations.

```
/* DPlib variables & memory locations */
#pragma DATA_SECTION(psfbvmc, "Cla1ToCpuMsgRAM")
PWMDRV_PSFB_VMC_SR_CLA_C psfbvmc;
```

Step 3 Add shared declaration to `{ProjectName}-CLA_Shared.h` file if access is shared between CPU and CLA.

```
/* DPlib variables */
// Declare net variables shared between CPU and CLA here
extern volatile PWMDRV_PSFB_VMC_SR_CLA_C psfbvmc;
```

Step 4 Initialize macro in CLA Task 8. Edit the task found in the `{ProjectName}-CLA_Tasks.cla` file to initialize the macro coefficients.

```
interrupt void Cla1Task8(void) {
	...
	PWMDRV_PSFB_VMC_SR_CLA_C_INIT(psfbvmc);
	...
}
```

Step 5 Connect the PWMDRV_PSFB_VMC_SR_CLA_C controller within a CLA-Task.

```
interrupt void Cla1Task4(void) {
	...
	psfbvmc.phase = phase;
	psfbvmc.dbPtoA = deadbandPtoA;
	psfbvmc.dbAtoP = deadbandAtoP;
	PWMDRV_PSFB_VMC_SR_CLA_C(psfbvmc, EPwm1Regs, EPwm2Regs,EPwm4Regs);
	...
}
```

Step 4 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
	...
	PWM_1ch_CNF(1, 600, 1, 0);
	...
}
```

Step 5 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Object Definition:

```
/**
 * PWMDRV_PSFB_VMC_SR_CLA_C structure.
 */
typedef struct {
      // Internal values
      float32 temp;
      float32 temp2;

      // Inputs
      float32 phase;
      float32 dbPtoA;
      float32 dbAtoP;
} PWMDRV_PSFB_VMC_SR_CLA_C;
```

Special Constants and Data types

PWMDRV_PSFB_VMC_SR_CLA_C

Structure defined to store module values. To create multiple instances of the module simply declare variables of type PWMDRV_PSFB_VMC_SR_CLA_C.

Module Structure Definition:

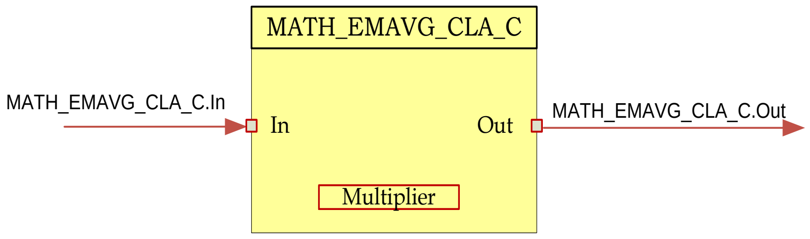
```
PWMDRV_PSFB_VMC_SR_CLA_C(PWMDRV_PSFB_VMC_SR_CLA_C, EPwm:n:Regs,
EPwm:m:Regs, EPwm:p:Regs);
```

Parameter Name	Types	Description	Acceptable Range
PWMDRV_PSFB_VMC_SR_CLA_C	Input	PFSB VMC SR structure	N/A
phase	Input	Desired phase shift.	float32 [0, 1)
dbPtoA	Input	Dead-band value for PtoA leg delay.	Uint16
dbAtoP	Input	Dead-band value for AtoP leg delay.	Uint16
EPwm:n:Regs	Input	PWM module registers. The :n: indicates which PWM is driven.	Device dependent
EPwm:m:Regs	Input	PWM module registers. The :m: indicates which PWM is driven.	Requirement: m = (n + 1)
EPwm:p:Regs	Input	PWM module registers. The :p: indicates which PWM is driven.	Requirement: m = (n + 3)

5.5 MATH Blocks

MATH_EMAVG_CLA_C ***Exponential Moving Average using CLA***

Description: This software module performs exponential moving average.



Macro File: `MATH_EMAVG_CLA_C.h`

Technical: This software module performs exponential moving average over float data provided via the MATH_EMAVG_CLA_C.In structure parameter. The result, also in float format, is stored in the structure and can be accessed by the MATH_EMAVG_CLA_C.Out parameter. 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'.

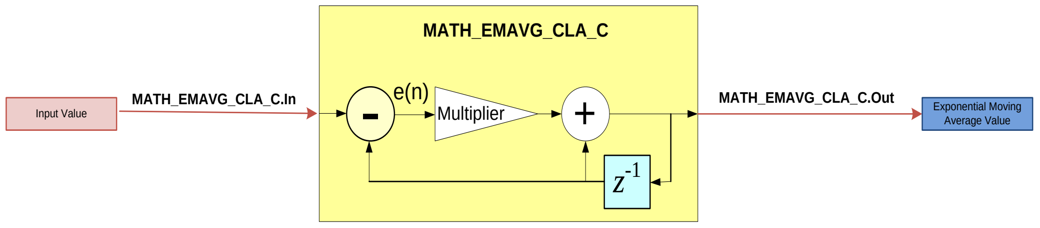
$Multiplier$ is the weighting factor used in exponential moving average

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, the lower the $Multiplier$ value, the larger the time constant and more sluggish filter response.

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



Usage:

The block is used in 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 a 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:

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

Usage: This section explains how to use the MATH_EMAVG_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `MATH_EMAVG_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare MATH_EMAVG_CLA_C structures in the `{ProjectName}-CLA_Tasks.cla` file. Specify structure variables and CLA memory locations.

```
/* DPlib variables & memory locations */
#pragma DATA_SECTION(math1, "Cla1ToCpuMsgRAM")
MATH_EMAVG_CLA_C math1;
```

Step 3 Add shared declaration to `{ProjectName}-CLA_Shared.h` file if access is shared between CPU and CLA.

```
/* DPlib variables */
// Declare net variables shared between CPU and CLA here
extern volatile MATH_EMAVG_CLA_C math1;
```

Step 4 Initialize macro in CLA Task 8. Edit the task found in the `{ProjectName}-CLA_Tasks.cla` file to initialize the macro coefficients.

```
interrupt void Cla1Task8(void) {
	...
	MATH_EMAVG_CLA_C_INIT(math1);
	math1.Multiplier = (float32) 0.3913;
	...
}
```

Step 5 Connect the MATH_EMAVG_CLA_C controller within a CLA-Task.

```
interrupt void Cla1Task4(void) {
	...
	math1.In = value;
	MATH_EMAVG_CLA_C(math1);
	Out = math1.Out;
	...
}
```

Step 4 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
	...
	PWM_1ch_CNF(5, 500, 1, 0);
	...
}
```

Step 5 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Object Definition:

```
/**
 * Exponential-moving average structure
 */
typedef struct {
	float32 In;
	float32 Out;
	float32 Multiplier;
} MATH_EMAVG_CLA_C;
```

Special Constants and Data types

MATH_EMAVG_CLA_C

Structure defined to store module values. To create multiple instances of the module simply declare variables of type MATH_EMAVG_CLA_C.

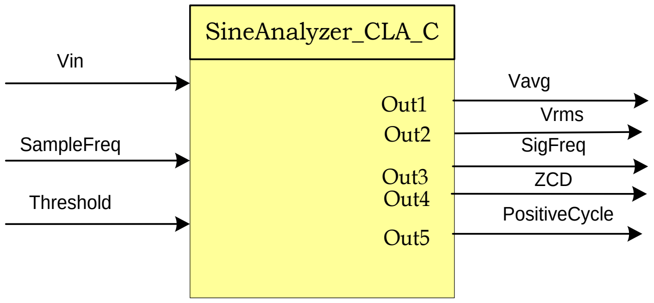
Module Structure Definition:

Parameter Name	Types	Description	Acceptable Range
In	Input	Data to be averaged.	float32 [0, 1)
Out	Output	Averaged result.	float32 [0, 1)
Multiplier	Input	Weighing factor for exponential average.	float32 [0, 1)

SineAnalyzer_CLA_C

Sine wave analyzer

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



Macro File: `SineAnalyzer_CLA_C.h`

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

1) Sine wave in (Vin): This is the signal sampled by ADC and ADC result converted to float format. This module expects a rectified sine wave as input without any offset.

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

3) Sampling Frequency (SampleFreq): This input should be set to the Frequency at which the input sine wave is sampled, in float format.

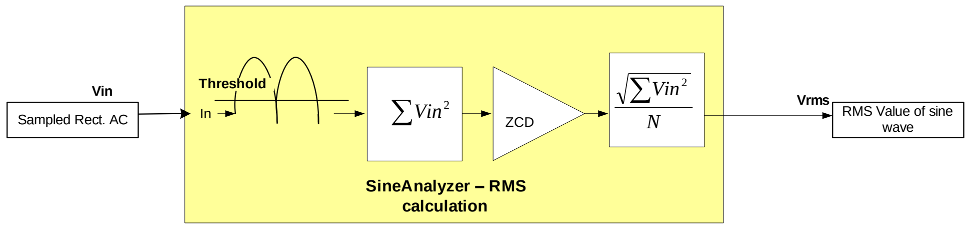
Upon Macro call – Input sine wave (Vin) is checked to see if the signal crossed over the threshold value. Once the cross over event occurs, successive Vin samples are accumulated until the occurrence of another threshold cross-over point. Accumulated values are used for calculation of Average, RMS values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

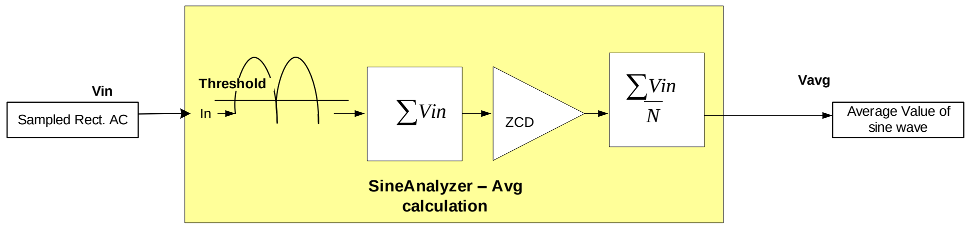
This module generates the following Outputs:

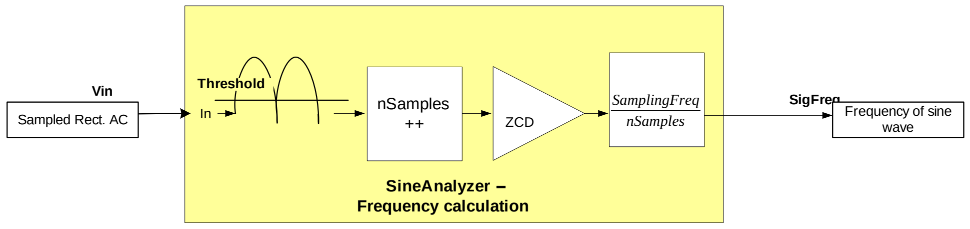
1) RMS value of sine wave (Vrms): Output reflects the RMS value of the sine wave input signal in float format. RMS value is calculated and updated at every threshold crossover point.

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

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







Usage: This section explains how to use the SineAnalyzer_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `SineAnalyzer_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare SineAnalyzer_CLA_C structures in the `{ProjectName}-CLA_Tasks.cla` file. Specify structure variables and CLA memory locations.

```
/* DPlib variables & memory locations */
#pragma DATA_SECTION(sineAnalyzer, "Cla1ToCpuMsgRAM")
SineAnalyzer_CLA_C sineAnalyzer;
```

Step 3 Add shared declaration to `{ProjectName}-CLA_Shared.h` file if access is shared between CPU and CLA.

```
/* DPlib variables */
// Declare net variables shared between CPU and CLA here
extern volatile SineAnalyzer_CLA_C sineAnalyzer;
```

Step 4 Initialize macro in CLA Task 8. Edit the task found in the `{ProjectName}-CLA_Tasks.cla` file to initialize the macro coefficients.

```
interrupt void Cla1Task8(void) {
        ...
        SineAnalyzer_CLA_C_INIT(sineAnalyzer);
        sineAnalyzer.Threshold = (float32) 1.39;
        ...
}
```

Step 5 Connect the SineAnalyzer_CLA_C controller within a CLA-Task.

```
interrupt void Cla1Task4(void) {
        ...
        sineAnalyzer.Vin = value;
        SineAnalyzer_CLA_C(sineAnalyzer);
        Out = sineAnalyzer.Vrms;
        ...
}
```

Step 4 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
        ...
        PWM_1ch_CNF(5, 500, 1, 0);
        ...
}
```

Step 5 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Object Definition:

```
/**
 * SineAnalyzer_CLA_C structure.
 */
typedef struct {
        float32 Vin;                // Sine signal
        float32 SampleFreq;         // Signal sampling frequency
        float32 Threshold;          // Voltage level corresponding to zero i/p
        float32 Vrms;               // RMS value
        float32 Vavg;               // Average value
        float32 SigFreq;            // Signal frequency
        float32 ZCD;                // Zero Cross detected
        float32 PositiveCycle;

        float32 Vacc_avg;
        float32 Vacc_rms;
        float32 curr_sample_norm;   // Normalized value of current sample
        float32 prev_sign;
        float32 curr_sign;
        float32 nsamples;           // Samples in half cycle mains
        float32 inv_nsamples;
        float32 inv_sqrt_nsamples;
} SineAnalyzer_CLA_C;
```

Special Constants and Data types

SineAnalyzer_CLA_C

Structure defined to store module values. To create multiple instances of the module simply declare variables of type SineAnalyzer_CLA_C.

Module Structure Definition:

Parameter Name	Types	Description	Acceptable Range
In	Input	Data to be averaged.	float32 [0, 1)
Out	Output	Averaged result.	float32 [0, 1)
Multiplier	Input	Weighing factor for exponential average.	float32 [0, 1)

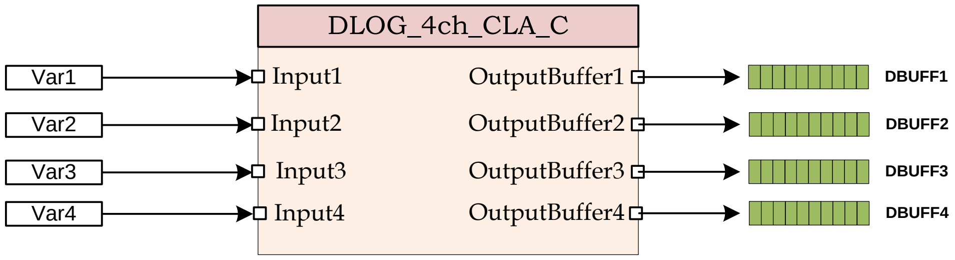
Module Structure Definition:

Net name	Type	Description	Acceptable Range
Vin	Input	Sampled Sine Wave input	float32 [0, 1)
Threshold	Input	Threshold to be used for cross over detection	float32
SampleFreq	Input	Frequency at which the Vin (input sine wave) is sampled, in Hz	float32
Vrms	Output	RMS value of the sine wave input (Vin) updated at cross over point	float32
Vavg	Output	Average value of the sine wave input (Vin) updated at cross over point	float32
SigFreq	Output	Frequency of the sine wave input (Vin) updated at cross over point	float32
ZCD	Output	When “1” - indicates that Cross over happened and stays high till the next call of the macro.	float32
Vacc_avg	Internal	Used for accumulation of samples for Average value calculation.	float32
Vacc_rms	Internal	Used for accumulation of squared samples for RMS value calculation	float32
Nsamples	Internal	Number of samples between two crossover points	float32
inv_nsamples	Internal	Inverse of nsamples	float32
inv_sqrt_nsamples	Internal	Inverse square root of nsamples	float32
Prev_sign, Curr_sign	Internal	Used for calculation of cross over detection	float32

5.6 Utilities

DLOG_4ch_CLA_C — ***Four Channel Data Logger***

Description: This software module performs data logging to emulate an oscilloscope in software to graphically observe system variables. The data is logged in the buffers and viewed as graphs in graph windows to observe the system variables as waveforms.



Macro File: `DLOG_4ch_CLA_C.h`

Technical: This software module performs data logging over data stored, in float format, within Input1-4 variables of a DLOG_4ch_CLA_C structure. The input values are stored within buffers pointed to by OutputBuffer1-4 variables also within the structure. The data logger is triggered at the positive edge of the value received in Input1. The trigger value is programmable by writing the desired trigger value to the structure TrigVal parameter. The size of the data logger has to be specified using the Size variable in the structure. The module can be configured to log data every n-th module call, by specifying a scalar value in the PreScalar variable. The following example illustrates how best to select these values using a PFC algorithm example.

Usage: When using the DLOG module for observing system variables in a PFC algorithm, it is desirable to observe the logged variable over a multiple line AC period to verify the algorithm is working. The PFC algorithm is typically run at 100kHz, the AC signal has a frequency of 60Hz. Taking memory constraints into account it is reasonable to expect 200 word arrays for each buffer.

If the DLOG module is called each time in the ISR i.e. at 100kHz, and the samples are logged every call the buffer size required to observe one sine period is 100KHz/60Hz ~= 1666. With memory constraints having four buffers like this is not feasible. 200 words array for each buffer would be a reasonable buffer size that would fit into the RAM of the device. Thus samples need be taken every alternate number or pre scalar number of times. Assuming two sine periods need to be observed in the watch window;

$$\mathrm{Pr}\,eScalar = \frac{100Khz}{(60Hz * BufferSize)} = 8.33$$

The trigger Value is used to trigger the logging of data at a positive edge around the trigger value of the data pointed to by Input1.

Usage: This section explains how to use the DLOG_4ch_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `DLOG_4ch_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare DLOG_4ch_CLA_C structures and buffer arrays in the `{ProjectName}-CLA_Tasks.cla` file. Specify structure variables and CLA memory locations.

```
/* DPlib variables & memory locations */
#pragma DATA_SECTION(buffer1, "Cla1DataRam0")
float32 buffer1[200];
#pragma DATA_SECTION(buffer2, "Cla1DataRam0")
float32 buffer2[200];
#pragma DATA_SECTION(buffer3, "Cla1DataRam0")
float32 buffer3[200];
#pragma DATA_SECTION(buffer4, "Cla1DataRam0")
float32 buffer4[200];

#pragma DATA_SECTION(dlog4ch, "Cla1ToCpuMsgRAM")
DLOG_4ch_CLA_C dlog4ch;
```

Step 3 Add shared declaration to `{ProjectName}-CLA_Shared.h` file if access is shared between CPU and CLA.

```
/* DPlib variables */
// Declare net variables shared between CPU and CLA here
extern volatile DLOG_4ch_CLA_C dlog4ch;
```

Step 4 Initialize macro in CLA Task 8. Edit the task found in the `{ProjectName}-CLA_Tasks.cla` file to initialize the macro coefficients.

```
interrupt void Cla1Task8(void) {
       ...
       DLOG_4ch_CLA_C_INIT(dlog4ch);
       dlog4ch.OutputBuffer1 = buffer1;
       dlog4ch.OutputBuffer2 = buffer2;
       dlog4ch.OutputBuffer3 = buffer3;
       dlog4ch.OutputBuffer4 = buffer4;
       dlog4ch.PreScalar = 8;
       dlog4ch.TrigVal = 0.23;
       ...
}
```

Step 5 Connect the DLOG_4ch_CLA_C controller within a CLA-Task.

```
interrupt void Cla1Task4(void) {
	...
	dlog4ch.Input1 = value1;
	dlog4ch.Input2 = value2;
	dlog4ch.Input3 = value3;
	dlog4ch.Input4 = value4;
	DLOG_4ch_CLA_C(dlog4ch);
	...
}
```

Step 4 Edit the `DPL_CLAInit()`, which should be declared in `{ProjectName}-Main.c` file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
	...
	PWM_1ch_CNF(5, 500, 1, 0);
	...
}
```

Step 5 Edit the `CLA_Init()` function within `{ProjectName}-DevInit.c`. See **Section 3.3** "Steps to use CLA DPlib" for details on how to modify this function to suit your application needs.

Object Definition:

```
/**
 * Data logger structure.
 */
typedef struct {
	float32 CurrTask;
	float32 Size;
	Uint16 Cnt;
	float32 PreScaler;
	float32 SkipCnt;
	float32 Input1;
	float32 Input2;
	float32 Input3;
	float32 Input4;
	float32 TrigVal;
	float32 *OutputBuffer1;
	float32 *OutputBuffer2;
	float32 *OutputBuffer3;
	float32 *OutputBuffer4;
} DLOG_4ch_CLA_C;
```

Special Constants and Data types

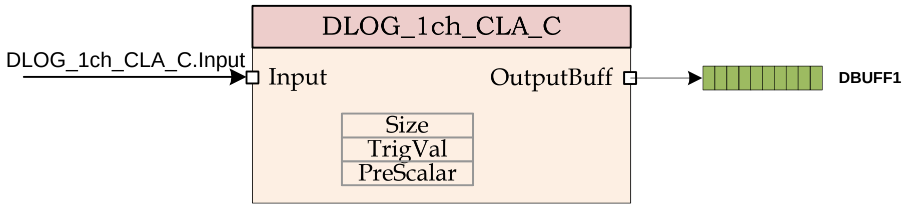
DLOG_4ch_CLA_C

Structure defined to store module values. To create multiple instances of the module simply declare variables of type DLOG_4ch_CLA_C.

Module Structure Definition:

Parameter Name	Types	Description	Acceptable Range
Input1-4	Input	Data that needs to logged.	float32
OutputBuffer1-4	Output Pointer	Pointer to output data buffer location where the logged data is stored.	float32
Size	Internal	Buffer size being used by the DLOG module.	float32
PreScalar	Internal Data	Data storing the pre-scalar value.	float32
TrigVal	Internal Data	Data variable storing the trigger value for the DLOG module.	float32

Description: This software module performs data logging to emulate an oscilloscope in software to graphically observe a system variable. The data is logged in a buffer that can be viewed as a graph to observe the system variables as waveforms.



Macro File: `DLOG_1ch_CLA_C.h`

Technical: This software module performs data logging over data stored, in float format, in the Input variable within a DLOG_1ch_CLA_C structure. The input variable value is then stored in the array pointed to by the OutputBuff structure variable. The data logger is triggered at the positive edge of the input. The trigger value is programmable by writing the desired trigger value to the TrigVal variable within the structure. The size of the data logger has to be specified using the Size structure variable. The module can be configured to log data every n-th module call, by specifying a scalar value in the PreScalar variable.

Usage: This section explains how to use the DLOG_1ch_CLA_C module.

Step 1 Add library header file and CLA shared header file to the `{ProjectName}-Main.c` file. Un-comment `DLOG_1ch_CLA_C.h` include found at the top of the `DPlib.h` file.

```
#include "DPlib.h"
#include "{ProjectName}-CLA_Shared.h"
```

Step 2 Declare DLOG_1ch_CLA_C structures and buffer arrays in the `{ProjectName}-CLA_Tasks.cla` file. Specify structure variables and CLA memory locations.

```
/* DPlib variables & memory locations */
#pragma DATA_SECTION(buffer1, "Cla1DataRam0")
float32 buffer1[200];

#pragma DATA_SECTION(dlog1ch, "Cla1ToCpuMsgRAM")
DLOG_1ch_CLA_C dlog1ch;
```

Step 3 Add shared declaration to `{ProjectName}-CLA_Shared.h` file if access is shared between CPU and CLA.

```
/* DPlib variables */
// Declare net variables shared between CPU and CLA here
extern volatile DLOG_1ch_CLA_C dlog1ch;
```

Step 4 Initialize macro in CLA Task 8. Edit the task found in the {ProjectName}-CLA_Tasks.cla file to initialize the macro coefficients.

```
interrupt void Cla1Task8(void) {
       ...
       DLOG_1ch_CLA_C_INIT(dlog1ch);
       dlog1ch.OutputBuff = buffer1;
       dlog1ch.PreScalar = 8;
       dlog1ch.TrigVal = 0.54;
       ...
}
```

Step 5 Connect the DLOG_1ch_CLA_C controller within a CLA-Task.

```
interrupt void Cla1Task4(void) {
       ...
       Dlog1ch.Input = value1;
       DLOG_1ch_CLA_C(dlog1ch);
       ...
}
```

Step 4 Edit the DPL_CLAInit(), which should be declared in {ProjectName}-Main.c file, to initialize hardware peripherals.

```
/* Digital Power (DP) CLA library peripheral initialization */
void DPL_CLAInit(){
       ...
       PWM_1ch_CNF(5, 500, 1, 0);
       ...
}
```

Step 5 Edit the CLA_Init() function within {ProjectName}-DevInit.c. See **Section 3.3** “Steps to use CLA DPlib” for details on how to modify this function to suit your application needs.

Object Definition:

```
/**
 * Data logger structure
 */
typedef struct {
       float32 CurrTask;
       float32 Size;
       Uint16 Cnt;
       float32 PreScaler;
       float32 SkipCnt;
       float32 Input;
       float32 TrigVal;
       volatile float32 *OutputBuff;
} DLOG_1ch_CLA_C;
```

Special Constants and Data types

DLOG_1ch_CLA_C

Structure defined to store module values. To create multiple instances of the module simply declare variables of type DLOG_1ch_CLA_C.

Module Structure Definition:

Parameter Name	Types	Description	Acceptable Range
Input1-4	Input	Data that needs to logged.	float32
OutputBuff	Output Pointer	Pointer to output data buffer location where the logged data is stored.	float32
Size	Internal	Buffer size being used by the DLOG module.	float32
PreScalar	Internal Data	Data storing the pre-scalar value.	float32
TrigVal	Internal Data	Data variable storing the trigger value for the DLOG module.	float32

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 Input Name changed from In to Duty for PWMDRV_1ch, PWMDRV_1chHiRes, PWMDRV_ComplPairDB
V3.1	December 2010	1. Corrected the documentation for PWMDRV_ComplPairDB, macro, removed reference od DbRed and DbFed from the CNF function 2. Added PWMDRV_BuckBoost Macro to the C28x library
V3.2	April 2011	No changes, versioning to match up with C28x update of the library
V3.3	November 2011	1. Corrected documentation for PWMDRV_BuckBoost Macro in C28x Lib 2. Added initialization for CMPA and CMPB values in Configuration files which were missing this 3. Added missing structure definition for CNTL2P2Z_CLA in the DPlib header file 4. Added PWMDRV_LLCComplpairDB_CLA macro 5. Added phase capability to PWMDRV_DualUpDwnCnt
V3.4	October 2012	Major Release: DPlib in CLA C language released.