

# **C Digital Power Library**

V1.0

Jun-15

## **Module User's Guide**

**C2000 Foundation Software**



## IMPORTANT NOTICE

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

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

Customers are responsible for their applications using TI components.

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

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

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

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

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

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

Copyright ©2015, Texas Instruments Incorporated

## 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

<b>Chapter 1. Introduction .....</b>	<b>6</b>
<b>1.1. Introduction.....</b>	<b>6</b>
<b>Installing the C DP Library.....</b>	<b>8</b>
<b>1.2. C DP Library Package Contents .....</b>	<b>8</b>
<b>1.3. How to Install the C Digital Power Library .....</b>	<b>8</b>
<b>1.4. Naming Convention .....</b>	<b>9</b>
<b>Chapter 2. Using Digital Power Library .....</b>	<b>10</b>
<b>2.1. Library Description and Overview .....</b>	<b>10</b>
<b>2.2. Memory Requirements in CLA based systems .....</b>	<b>10</b>
<b>2.3. Steps to use the DP library .....</b>	<b>11</b>
<b>Chapter 3. Module Summary.....</b>	<b>12</b>
<b>3.1. DP C Library Function Summary .....</b>	<b>12</b>
<b>Chapter 4. DP Module Descriptions.....</b>	<b>15</b>
<b>4.1. Controllers.....</b>	<b>15</b>
5.1.1 CNTL_2P2Z_F_C : Two Pole Two Zero Controller.....	15
5.1.2 CNTL_3P3Z_F_C : Three Pole Three Zero Controller .....	20
<b>4.2. Peripheral Configuration .....</b>	<b>25</b>
5.2.1 ADC_SOC_Cnf : ADC Configuration .....	25
5.2.2 PWM_PSFBS_PCMC_Cnf : PWM for PCMC controlled PSFB Stage.....	27
5.2.3 PWM_1ch_UpCntDB_ActivHIC_Cnf : Complementary PWM Pair .....	28
<b>4.3. Peripheral Drivers .....</b>	<b>29</b>
5.3.1 ADCDRV_1ch_F_C : ADC Driver Single Channel.....	29
5.3.4 PWMDRV_1ch_F_C : PWM Driver Single Channel .....	30
5.3.5 PWMDRV_1chHiRes_F_C : PWM Driver Single Channel Hi Res.....	34
5.3.6 PWMDRV_1chHiRes_UpDwnCnt_F_C : PWM Driver 1ch Hi Res Up Down Count Mode.....	38
5.3.7 PWMDRV_PFC2PHIL_F_C : PWM Driver for Two Phase Interleaved PFC Stage	42
5.3.8 PWMDRV_PSFBS_F_C : PWM Driver for Phase Shifted Full Bridge Stage.....	45
5.3.9 PWMDRV_ComplPairDB_F_C : PWM Driver with complementary chA & chB PWM .....	48
5.3.10 PWMDRV_DualUpDwnCnt_F_C : Dual PWM Driver , independent Duty on chA & ChB.....	52
5.3.11 PWMDRV_BuckBoost_F_C : Dual PWM Driver, independent Duty on chA & chB .....	55
5.3.12 PWMDRV_2ch_UpCnt_F_C : Dual PWM Driver, independent Duty on chA and chB.....	59
5.3.13 PWMDRV_1ch_UpDwnCnt_F_C : PWM Driver, Duty chA centered at period.....	62
5.3.14 PWMDRV_1ch_UpDwnCntCompl_F_C : PWM Driver, Duty chA centered at period.....	65
5.3.15 PWMDRV_PSFBS_VMC_SR_F_C : PWM Driver for VMC PSFB Stage with SR..	67
5.3.16 PWMDRV_LLC_ComplPairDB_F_C : PWMDriver for compl. PWM period modulation .....	71

5.3.17 PWMDRV_LLC_1ch_UpCntDB_F_C : PWM Driver for chA PWM edge shift, period mode.....	75
5.3.18 PWMDRV_LLC_1ch_UpCntDB_Compl_F_C : PWM Driver for chA PWM edge shift, period mode .....	79
5.3.18 DACDRV_RAMP_F_C : DAC Ramp Driver Interface .....	83
<b>4.4. Application Specific .....</b>	<b>85</b>
5.4.1 PFC_ICMD_F_C : Current Command for Power Factor Correction .....	85
5.4.2 PFC_INVSQR_F_C : Inverse Square Math Block for Power Factor Correction.....	88
5.4.3 PFC_BL_ICMD_F_C : Bridgeless PFC Current Command .....	91
5.4.4 PFC_InvRmsSqr_F_C : Inverse of the RMS Squared .....	94
<b>5.5 MATH Blocks .....</b>	<b>97</b>
5.5.1 MATH_EMAVG_F_C : Exponential Moving Average .....	97
5.5.2 SineAnalyzer_F_C : Sine Wave Analyzer .....	101
<b>5.6 Utilities.....</b>	<b>106</b>
5.6.1 DLOG_4ch_F_C : Four Channel Data Logger .....	106
5.6.2 DLOG_1ch_F_C : Single Channel Data Logger .....	110
<b>Chapter 5. Revision History.....</b>	<b>113</b>

# Chapter 1. Introduction

## 1.1. Introduction

Texas Instruments C Digital Power Library (DPlib) is designed to enable flexible and efficient coding of digital power supply application. 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 same C library functions can be run on the C28x FPU and the CLA, 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.

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 used together to build any desired control loop structure. The major difference between using the library for CLA and C28x FPU is memory allocation for different components of the modules in the power library. One may choose to put certain variables in CPU writable CLA readable space, some in CLA writable and CPU readable space.

An example of a simple block diagram showing the connection of three DPlib modules to form a closed loop system is shown below.

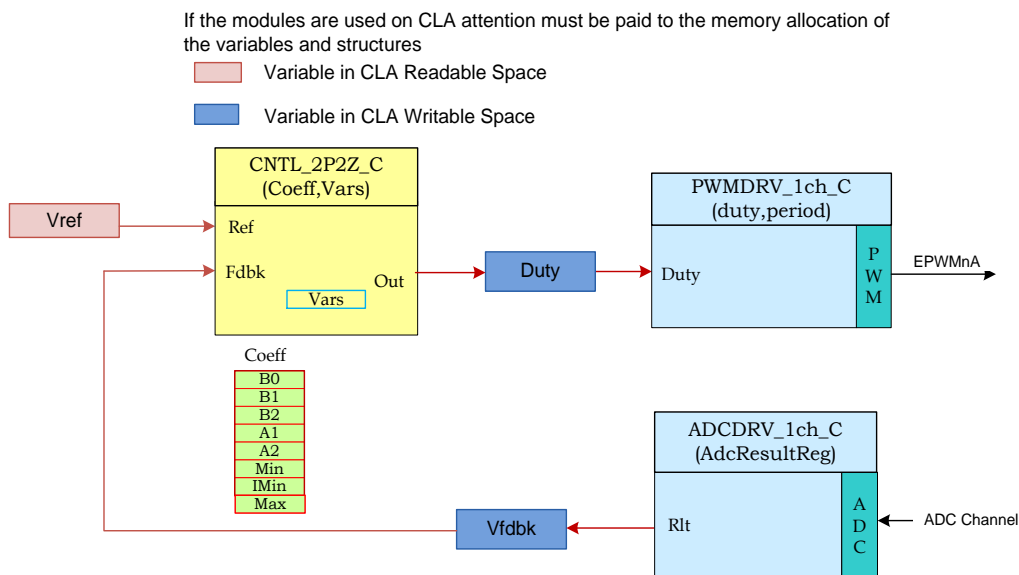


Figure 1: Closed loop system using C DPlib blocks

In the example above, three library blocks are connected: an ADC driver, a second order digital controller, and a PWM driver.

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, “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.

# Installing the C DP Library

## 1.2. C DP Library Package Contents

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

- C configuration files
- C header macro files
- Documentation
- Library header file

## 1.3. How to Install the C Digital Power Library

The 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:

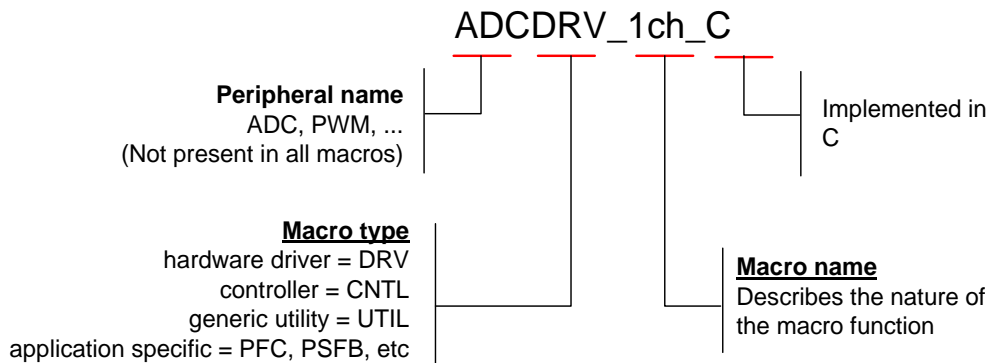
<base>\C_macros	Contains implementation of the DP Library for the C28x FPU and the Control Law Accelerator using the CLA C Compiler
<base>\Doc	Contains this file
<base>\CNF	C-language initialization/configuration files
<base>\include	contains DPlib header file

Note, as C28x FPU and CLA are floating point core's the same source code can be used on both.



## 1.4. Naming Convention

The initialization and execution code for each 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\_F\_C.h

Macro call:     ADCDRV\_1ch\_F\_C()

# Chapter 2. Using Digital Power Library

## 2.1. Library Description and Overview

Typical software will consist of a main framework file written in C and an ISR implemented in C, if CLA is used, a CLA Task file will be present. The C framework contains code to configure the device hardware, the CLA, peripheral interrupts to the C28x FPU and CLA-Task triggers from the PWMs or ADCs. When using the CLA, because it operates independently of the CPU core, main CPU can be freed to do other tasks.

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

1. **Includes** – All the function are prototyped in DPlib.h which is located inside the include folder. One must add both the include and the C\_macros folder to the include search path in the project.
2. **Initialization**- Hardware initialization or module initialization functions are called from the C environment using a C callable function such as `PWM_1CH_UpdDwnCnt_CNF() /CNTL_2P2Z_F_C_VAR_INIT`. This would also mean the corresponding source file under CNF folder needs to be added to the project for the peripheral configuration.
3. **Configuration** – Some DPlib modules require additional variables. When using CLA 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 which is included across C28x FPU and CLA Task file.
4. **Execution** – Macro-code is executed in ISR/CLA tasks.

## 2.2. 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.

### Memory Allocation on F28x7x

All local shared LSx memory on F38x7x can be assigned to CLA as program or data memory. User must ensure that the memory is assigned to the CLA as program and data memory and verify the CMD file to make sure the mapping is correct.

In addition to the LSx memory two dedicated memory section can be used with specific privileges assigned to each section.

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.

## 2.3. Steps to use the DP library

The first task before using the 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. when CLA is used, 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. In addition to this use must also add the following path to the include path list

```
controlSUITE\libs\app_libs\digital_power\{device_name_VerNo}\C_macros
```

**Step 2** *Declare array of registers address pointers for PWM and ADC registers*

```
volatile struct EPWM_REGS *ePWM[] = {
    &EPwm1Regs,           //intentional: (ePWM[0] not used)
    &EPwm1Regs, &EPwm2Regs, &EPwm3Regs, &EPwm4Regs, &EPwm5Regs,
    &EPwm6Regs,
    &EPwm7Regs, &EPwm8Regs, &EPwm9Regs, &EPwm10Regs, &EPwm11Regs,
    &EPwm12Regs};

volatile struct ADC_REGS *ADC[] = {
    &AdcaRegs,
    &AdcaRegs, &AdcbRegs, &AdccRegs, &AdcdRegs
};
```

**For rest of the steps refer to the individual module documentation**

# Chapter 3. Module Summary

## 3.1. DP C Library Function Summary

The Digital Power 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	Multiple Instance Support
ADCDRV_F_C	HW	ADC driver	Yes	Yes
CNTL_2P2Z_F_C*	CNTL	Second order control law	N/A	Yes
CNTL_3P3Z_F_C*	CNTL	Third order control law	N/A	Yes
DACDRV_RAMP_F_C	HW	DAC driver for slope compensation	Yes	Yes
DLOG_1ch_F_C*	UTIL	1 channel data logger module	N/A	Yes
DLOG_4ch_F_C*	UTIL	4 channel data logger module	N/A	Yes
MATH_EMAVG_F_C*	MATH	Exponential moving average module	N/A	Yes
PFC_BL_ICMD_F_C*	APPL	Power factor correction current command block	N/A	Yes
PFC_ICMD_F_C*	APPL	Power factor correction current command block	N/A	Yes
PFC_InvRmsSqr_F_C	APPL	Power Factor Correction Inverse Square Block	N/A	Yes
PFC_INVSQR_F_C*	APPL	Power factor correction inverse square block	N/A	Yes
PWMDRV_1ch_F_C	HW	Single channel PWM driver	Yes	Yes
PWMDRV_1ch_UpDwnCnt_F_C	HW	Single channel driver with up down modulation	Yes	Yes
PWMDRV_1ch_UpDwnCntCompl_F_C	HW	Single channel driver with up down modulation centered around period	Yes	Yes

PWMDRV_1chHiRes_F_C	HW	Single channel PWM driver with HiRes capability	Yes	Yes
PWMDRV_1chHiRes_UpDwnCnt_F_C	HW	Single channel PWM driver with HiRes capability	Yes	Yes
PWMDRV_2ch_UpCnt_F_C	HW	PWM driver with independent duty control on ch A and ch B, using up down count mode	Yes	Yes
PWMDRV_BuckBoost_F_C	HW	PWM driver for a four switch Buck Boost stage	Yes	Yes
PWMDRV_BuckBoost_F_C_UpdateDB	HW	Update dead band of Buck Boost stage	Yes	Yes
PWMDRV_ComplPairDB_F_C	HW	PWM driver for complimentary pair PWM	Yes	Yes
PWMDRV_ComplPairDB_F_C_UpdateDB	HW	Update complementary PWM dead band	Yes	Yes
PWMDRV_DualUpDwnCnt_F_C	HW	PWM driver with independent duty control on ch A and B	Yes	Yes
PWMDRV_LLC_1ch_UpCntDB_F_C	HW	Driver with frequency modulation & rising/falling-edge adjustment via DB registers	Yes	Yes
PWMDRV_LLC_1ch_UpCntDB_F_C_UpdateDB	HW	Updates 1ch LLC dead band	Yes	Yes
PWMDRV_LLC_1ch_UpCntDB_Compl_F_C	HW	Driver with frequency modulation and rising/falling-edge adjustment via DB registers	Yes	Yes
PWMDRV_LLC_1ch_UpCntDB_Compl_F_C_UpdateDB	HW	Dead band adjustment for LLC 1ch UpCntDB Compl	Yes	Yes
PWMDRV_LLC_ComplPairDB_F_C	HW	PWM driver for LLC	Yes	Yes
PWMDRV_LLC_ComplPairDB_F_C_UpdateDB	HW	Update PWM dead band for LLC ComplPair DB	Yes	Yes
PWMDRV_PFC2PhiL_F_C	HW	PWM driver for Two Phase Interleaved PFC stage	Yes	Yes

PWMDRV_PSFB_F_C	HW	PWM driver for Phase Shifted Full Bridge Power stage	Yes	Yes
PWMDRV_PSFB_VMC_SR_F_C*	HW	PWMDRV for PSFB PCMC with SR	Yes	Yes
SineAnalyzer_F_C*	UTIL	Calculates wave RMS, average and frequency.	N/A	Yes

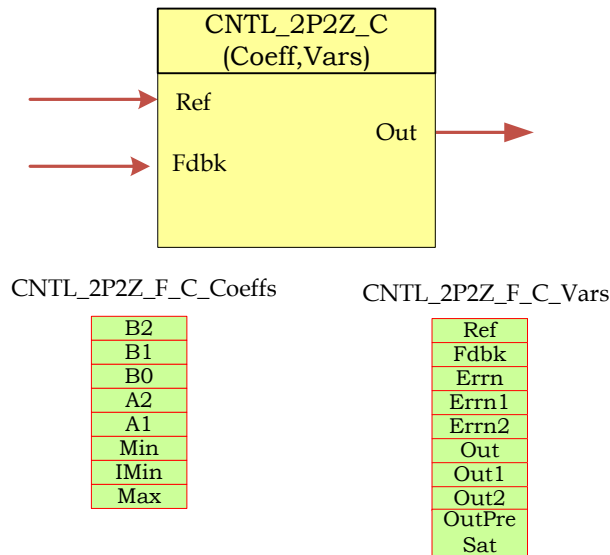
\* Requires the use of C structure to store internal variables. Each module has additional code to initialize respective structures.

# Chapter 4. DP Module Descriptions

## 4.1. Controllers

### 5.1.1 CNTL\_2P2Z\_F\_C : Two Pole Two Zero Controller

**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.



**Macro File:** CNTL\_2P2Z\_F\_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:

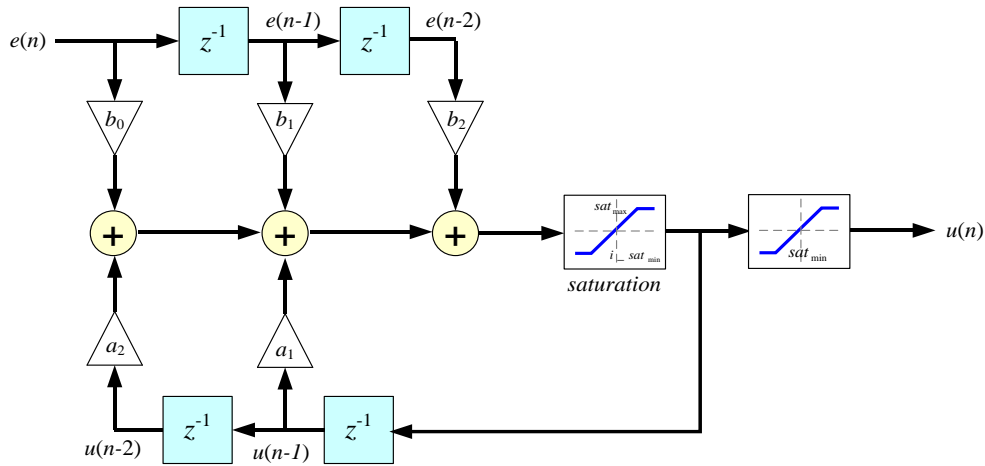
$u(n)$  = present controller output (after saturation)

$u(n-1)$  = controller output on previous cycle

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

$e(n)$  = present controller input  
 $e(n-1)$  = controller input on previous cycle  
 $e(n-2)$  = controller input two cycles previously

The 2P2Z control law may be represented graphically as shown below where  $sat_{max}$  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.



Input and output data is stored in memory within a CNTL\_2P2Z\_F\_C\_Vars structure. The controller coefficients and saturation settings are stored in memory within a CNTL\_2P2Z\_F\_C\_Coeffs structure, shown in an earlier diagram.

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

**Usage:** This section explains how to use the CNTL\_2P2Z\_F\_C module.

**Step 1 Add include for DPLib header file to the source file and add the DPLib include and C\_macros folder to the include path of the project.**

**Step 2 Declare structure** in the source file

```

/* DPLib variables */
CNTL_2P2Z_CLA_C_Coeffs coeffs;
CNTL_2P2Z_CLA_C_Vars vars;

```

In case of CLA specify memory locations, by default CLA data section will be used.

```

/* DPLib variables memory locations */
#pragma DATA_SECTION(coeffs, "CpuToCla1MsgRAM")
#pragma DATA_SECTION(vars, "Cla1ToCpuMsgRAM")

```

**Step 3 For CLA add shared declaration** to {ProjectName}-CLA\_Shared.h if variables are shared between CPU and CLA.\



```
/* DPlib variables */  
extern volatile CNTL_2P2Z_F_C_Coeffs coeffs;  
extern volatile CNTL_2P2Z_F_C_Vars vars;
```

**Step 4 Macro initialization** For CLA edit the task8 with the initialization routine shown below. This task found in the {ProjectName}-CLA\_Tasks.cla. For C28x call the init function in the main file before enabling interrupts

```
...  
CNTL_2P2Z_F_C_INIT(coeffs, vars);  
coeffs.Coeff_B2 = 5;  
coeffs.Min = 10;  
coeffs.IMin = 11;  
coeffs.Max = 12;  
...
```

**Step 5 Call** the macro within a CLA-Task of C28x ISR.

```
...  
vars.Ref = Ref;  
vars.Fdbk = Fbk;  
CNTL_2P2Z_F_C(coeffs, vars);  
Out = vars.Out;  
...
```

**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_F_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_F_C_Vars;
```

### Special Constants and Data types

#### **CNTL\_2P2Z\_F\_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\_F\_C\_Coeffs.

#### **CNTL\_2P2Z\_F\_C\_Vars**

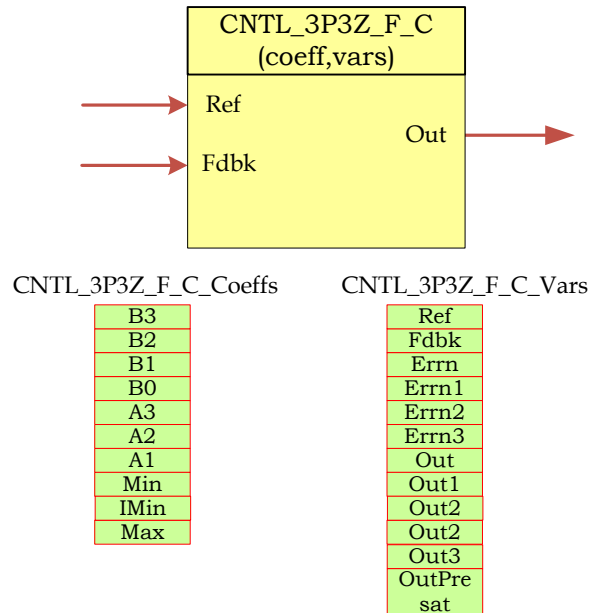
Structure defined to store the internal values and output. To create multiple instances of the module simply declare variables of type CNTL\_2P2Z\_F\_C\_Vars.

#### **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

## 5.1.2 CNTL\_3P3Z\_F\_C : Three Pole Three Zero Controller

**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.



**Macro File:** CNTL\_3P3Z\_F\_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)

$u(n-1)$  = controller output on previous cycle

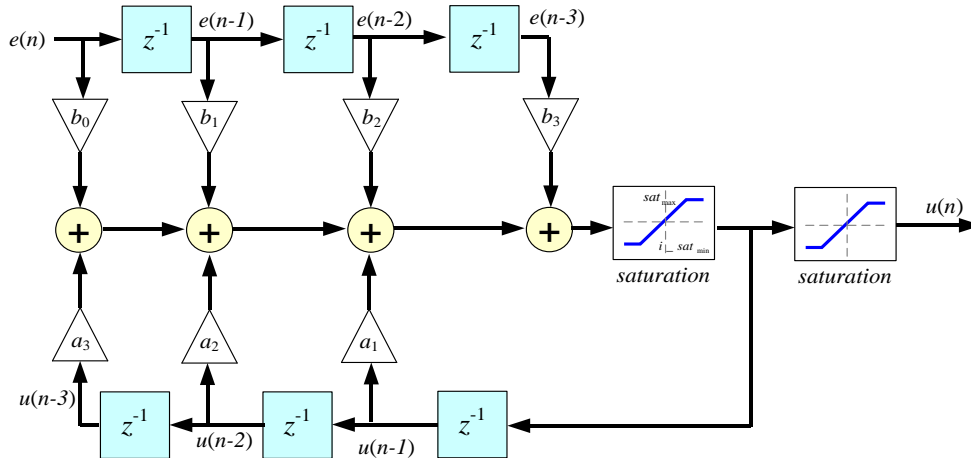
$u(n-2)$  = controller output two cycles previously

$e(n)$  = present controller input

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

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

The 3P3Z control law may be represented graphically as shown below. Where  $sat_{max}$  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.



Input and output data is stored in memory within a CNTL\_3P3Z\_F\_C\_Vars structure. The controller coefficients and saturation settings are stored in memory within a CNTL3P3Z\_F\_C\_Coeffs structure shown earlier.

**Usage:** This section explains how to use the CNTL\_3P3Z\_CLA\_C module.

**Step 1 Add include for DPLib header file to the source file and add the DPLib include and C\_macros folder to the include path of the project.**

**Step 2 Declare structure** in the source file

```
/* DPLib variables */
CNTL_3P3Z_CLA_C_Coeffs coeffs;
CNTL_3P3Z_CLA_C_Vars vars;
```

In case of CLA specify memory locations, by default CLA data section will be used.

```
/* DPLib variables memory locations */
#pragma DATA_SECTION(coeffs, "CpuToCla1MsgRAM")
#pragma DATA_SECTION(vars, "Cla1ToCpuMsgRAM")
```

**Step 3 For CLA add shared declaration** to {ProjectName}-CLA\_Shared.h if variables are shared between CPU and CLA.\

```
/* DPLib variables */
extern volatile CNTL_3P3Z_F_C_Coeffs coeffs;
extern volatile CNTL_3P3Z_F_C_Vars vars
```

**Step 4 Macro initialization** For CLA edit the task8 with the initialization routine shown below. This task found in the {ProjectName}-CLA\_Tasks.cla. For C28x call the init function in the main file before enabling interrupts

```
...
CNTL_3P3Z_F_C_INIT(coeffs, vars);
coeffs.Coeff_B2 = 5;
coeffs.Min = 10;
coeffs.IMin = 11;
coeffs.Max = 12;
...
```

**Step 5 Call** the macro within a CLA-Task of C28x ISR.

```
...
vars.Ref = Ref;
vars.Fdbk = Fbk;
CNTL_3P3Z_F_C(coeffs, vars);
Out = vars.Out;
...
```

### 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_F_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_F_C_Vars;
```

### Special Constants and Data types

#### **CNTL\_3P3Z\_F\_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\_F\_C\_Coeffs.

#### **CNTL\_3P3Z\_F\_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\_F\_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



## 4.2. Peripheral Configuration

### 5.2.1 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 powers up this on chip ADC, configures it to be in 12 bit mode and enables the user to configure the ADC as desired. The function is defined as:

```
void ADC_SOC_CNF(int AdcNo,int ChSel[], int Trigsel[], int
ACQPS[], int IntChSel, int mode, Uint16 resolution, Uint16
signalmode )
```

Where:

ADCNo     1-> ADC A  
           2-> ADC B  
           3-> ADC C  
           4-> ADC C

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.

resolution	0 for 12 bit mode 1 for 16 bit mode
signalmode	0 for single ended mode 1 for differential mode

Note the function configures the complete ADC module in a single function call for the device.

### Usage:

**Step 1 Add include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Declare Arrays** in the source file for the channel selection, acquisition window size, and trigger select

```
// Default ADC initialization
int ChSel[16]   = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
int TrigSel[16] = {5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5};
int ACQPS[16]   = {8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8};
```

**Call the peripheral configuration function** in {ProjectName}-Main.c, this function is defined in ADC\_SOC\_CNF.c. This file must be included manually into the project.

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

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

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

// Call the ADC Configuration Function
ADC_SOC_CNF(1,ChSel,TrigSel,ACQPS,1,0,
            ADC_RESOLUTION_12BIT,ADC_SIGNALMODE_SINGLE);
```

## 5.2.2 PWM\_PSFB\_PCMC\_Cnf : PWM 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);
```

### 5.2.3 PWM\_1ch\_UpCntDB\_ActivHIC\_Cnf : Complementary PWM Pair

**Description:** This module configures the PWM generators to control a synchronous buck power stage with complimentary PWM.

**Peripheral**

**Initialization File:** PWM\_1ch\_UpCntDB\_ActivHIC\_Cnf.c

**Description:** This module sets the initial configuration of a PWM peripheral to drive a synchronous buck power stage. This file is used in conjunction with the PWMDRV\_1ch\_CLA driver. Look at the driver documentation for more details.

```
void PWM_1ch_UpCntDB_ActivHIC_CNF(int16 n, int16 period,  
int16 mode, int16 phase)
```

where

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

mode

Period is the maximum count value of the PWM timer

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

Mode =1 PWM configured as a master

Mode = 0 PWM configured as slave

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

**Usage:**

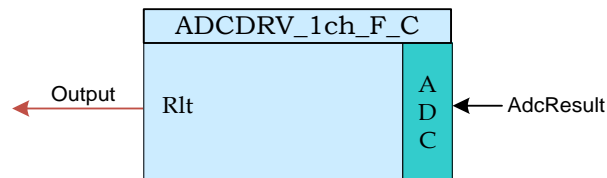
**Call the peripheral configuration function** PWM\_1ch\_UpCntDB\_ActivHIC\_CNF(int16 n, int16 period, int16 mode, int16 phase) in {ProjectName}-Main.c, this function is defined in PWM\_1ch\_UpCntDB\_ActivHIC\_Cnf.c. This file must be included manually into the project.

```
// Configure PWM1 for 100Khz, @60Mhz CPU Clock, in master mode  
PWM_1ch_UpCntDB_ActivHIC_CNF(1, 600, 1, 0);
```

## 4.3. Peripheral Drivers

### 5.3.1 ADCDRV\_1ch\_F\_C : ADC Driver Single Channel

**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\_F\_C.h

**Peripheral**

**Initialization File:** ADC\_SOC\_Cnf.c

**Description:** The ADCDRV\_1ch\_F\_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\_F\_C module.

**Step 1 Add include for DPLib header file to the source file and add the DPLib include and C\_macros folder to the include path of the project.**

**Step 2 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks.

```
...
ADC_SOC_CNF(1,ChSel,TrigSel,ACQPS,1,0,
             ADC_RESOLUTION_12BIT,ADC_SIGNALMODE_SINGLE);
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```
...
temp0 = ADCDRV_1ch_F_C(AdcResult.ADCRESULT0);
...
```

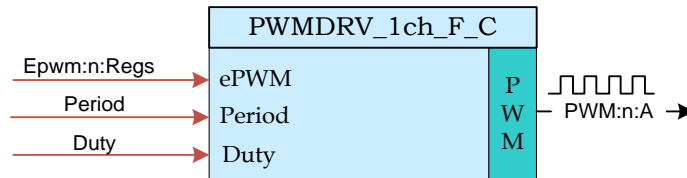
**Module Parameters:**

```
ADCDRV_1ch_F_C(EPwm:n:Regs, period, duty);
```

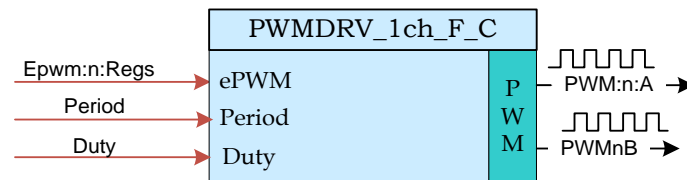
Parameter Name	Types	Description	Acceptable Range
AdcConvResult	Input	ADC Converter Result Register	Uint16

### 5.3.4 PWMDRV\_1ch\_F\_C : PWM Driver Single Channel

**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.



(When used with PWM\_1ch\_Cnf.c)



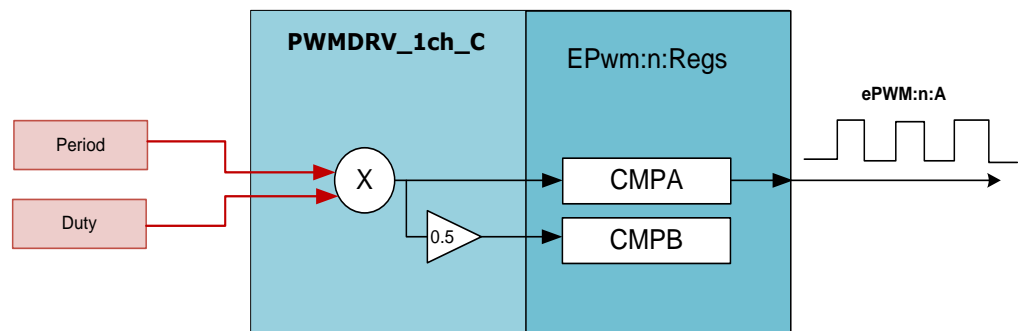
(When used with PWM\_1ch\_UpCntDB\_ActivHIC\_Cnf.c)

**Macro File:** PWMDRV\_F\_C.h

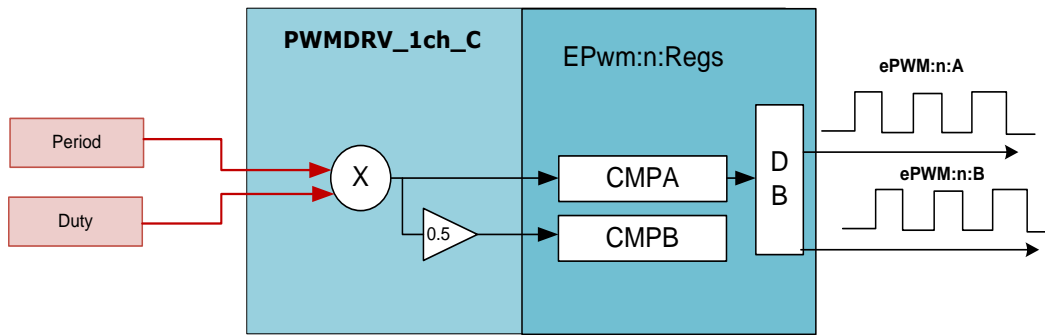
**Peripheral**

**Initialization File:** PWMDRV\_1ch\_Cnf.c or PWM\_1ch\_UpCntDB\_ActivHIC\_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:.



(When used with PWM\_1ch\_Cnf.c)



(When used with `PWM_1ch_UpCntDB_ActivHIC_Cnf.c`)

This macro is used in conjunction with the peripheral configuration file `PWMDRV_1ch_Cnf.c` or `PWM_1ch_UpCntDB_ActivHIC_Cnf.c`. The file defines the function

```
void PWM_1ch_CNF(int16 n, int16 period, int16 mode, int16 phase)
```

```
void PWM_1ch_UpCntDB_ActivHIC_CNF(int16 n, int16 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.

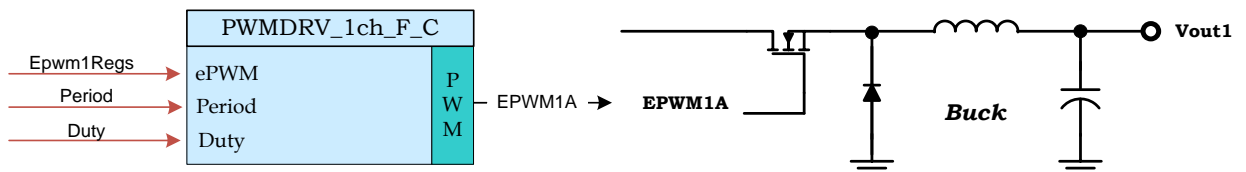
Mode = 1 PWM configured as master

Mode = 0 PWM configured as slave

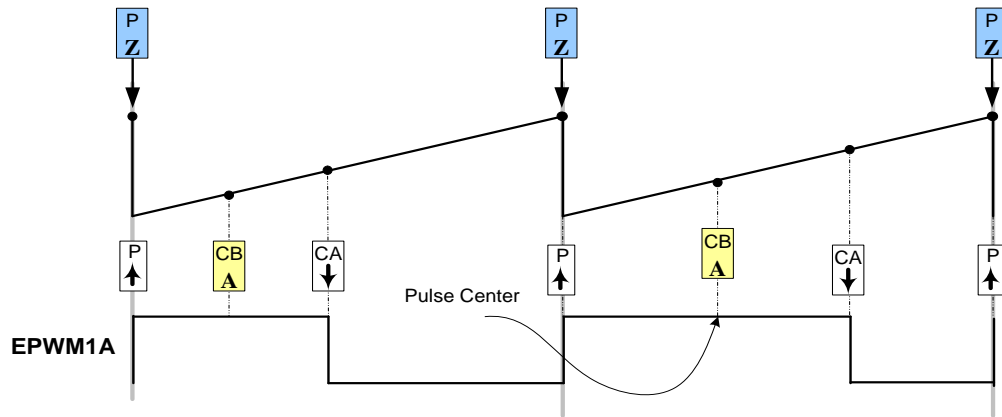
phase Specifies the phase offset that is used when the PWM module is synced, this value only has meaning when the PWM is configured as a slave.

### Detailed Description

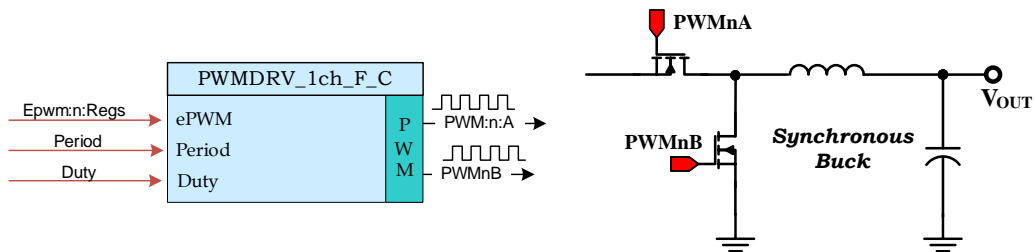
The following section explains how this module can be used to excite buck power stage. To configure 100 kHz switching frequency with CPU operating at 60 MHz, the period value needed is (System Clock/Switching Frequency) = 600.



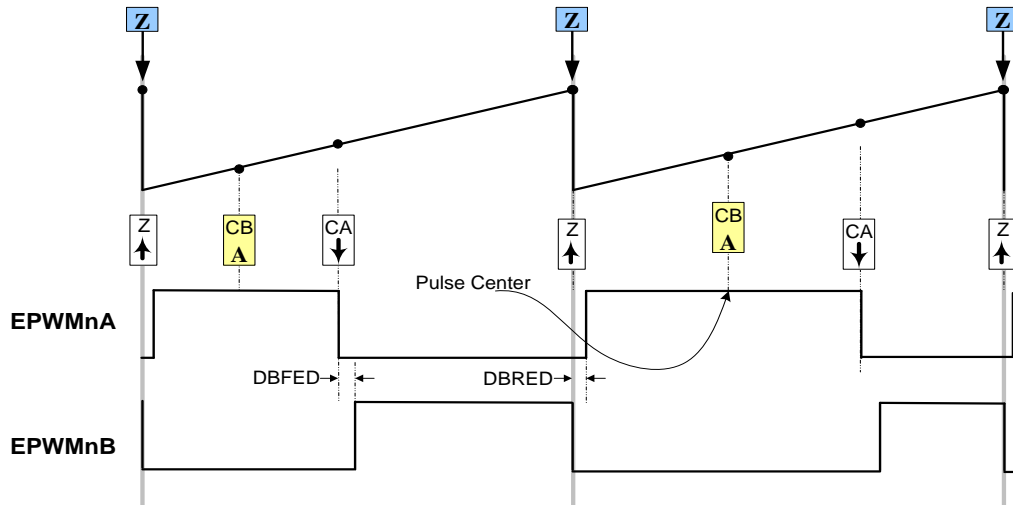
**Buck converter driven by `PWMDRV_1ch_F_C` module (when used with `PWM_1ch_Cnf.c`)**



**PWM generation with the EPWM module (when used with PWM\_1ch\_Cnf.c)**



**Buck converter driven by PWMDRV\_1ch module (when used with PWM\_1ch\_UpCntDB\_ActivHIC\_Cnf.c)**



**PWM generation with the EPWM module (When used with PWM\_1ch\_UpCntDB\_ActivHIC\_Cnf.c)**

**Usage:** This section explains how to use the PWMDRV\_1ch\_F\_C module.



**Step 1 Add include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. The CNF file needs to be manually added to the project as well.

```
...
PWM_1ch_CNF(1, 600, 1, 0);
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```
...
PWMDRV_1ch_F_C(EPwm1Regs, EPwm1Regs.TBPRD, duty);
...
```

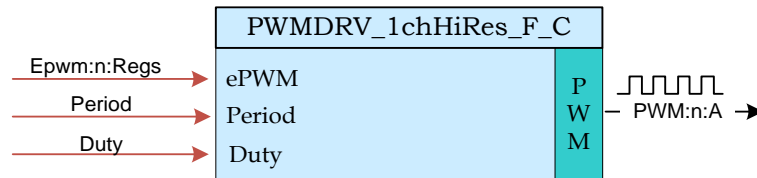
**Module Parameters:**

```
PWMDRV_1ch_F_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)

### 5.3.5 PWMDRV\_1chHiRes\_F\_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.

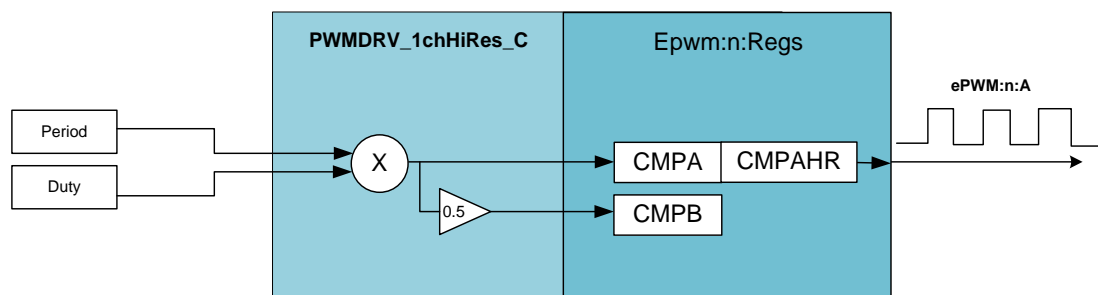


**Macro File:** `PWMDRV_F_C.h`

**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)
```

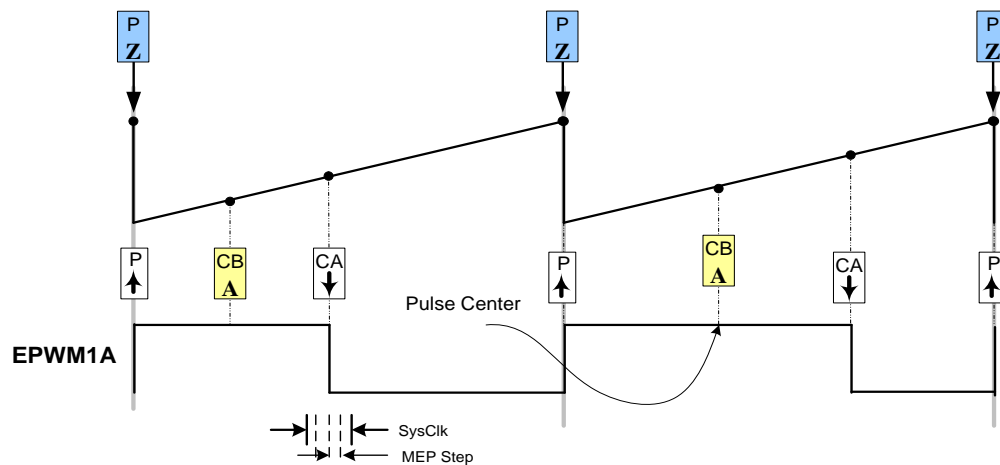
Where:

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

**period** is the maximum count value of the PWM timer

Mode = 1 PWM configured as master  
Mode = 0 PWM configured as 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** to the source file and add the DPLib include and C\_macros folder to the include path of the project. 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.

*control\SUITE\device\_support\<Device\_Name>\<Version>\<Device\_Name>\_common\lib*

*control\SUITE\device support\<Device Name>\<Version>\<Device Name> common\include*

**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 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. 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. The CNF file needs to be manually added to the project as well.

```
...
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;
...

```

**Step 4 Call the macro within a CLA-Task or C28x ISR.**

```
...
PWMDRV_1chHiRes_F_C(EPwm1Regs, EPwm1Regs.TBPRD, duty);
...

```

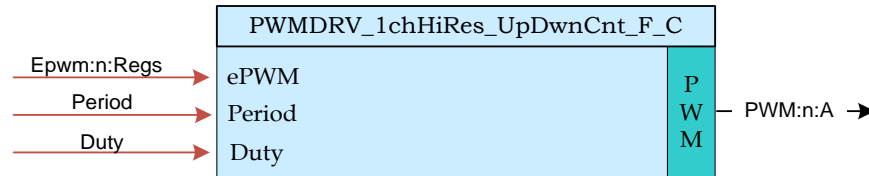
**Module Parameters:**

```
PWMDRV_1chHiRes_F_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)

### 5.3.6 PWMDRV\_1chHiRes\_UpDwnCnt\_F\_C : PWM Driver 1ch Hi Res Up Down Count Mode

**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.

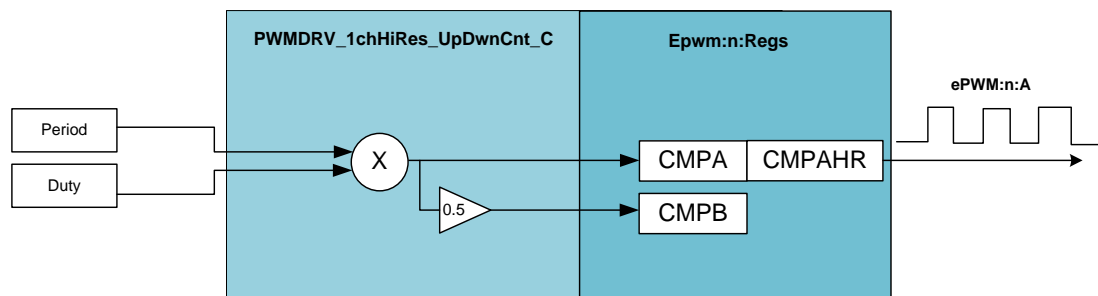


**Macro File:** PWMDRV\_F\_C.h

**Peripheral**

**Initialization File:** PWM\_1chHiRes\_UpDwnCnt\_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:

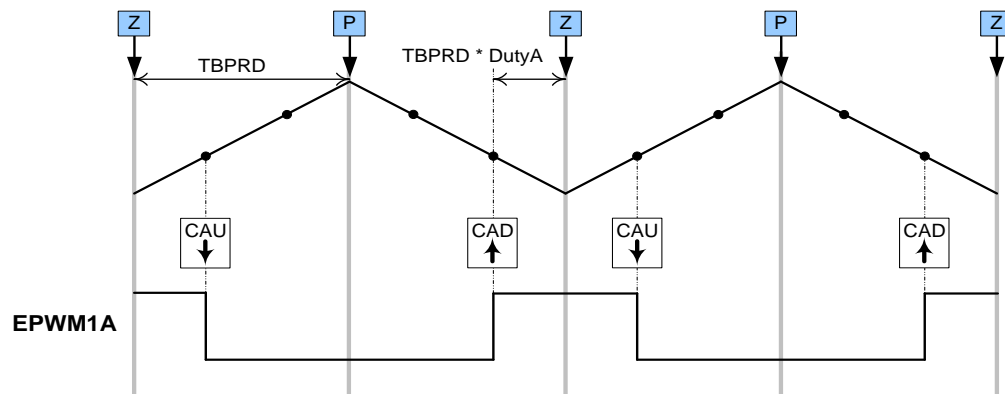
n PWM Peripheral number; PWM will be configured in up-down count mode

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\_F\_C module.

**Step 1 Add library header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project. 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*

**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 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. 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. The CNF file needs to be manually added to the project as well.

```
...
PWM_1chHiResUpDwnCnt_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;
...
}
```

**Step 4 Call the macro within a CLA-Task or C28x ISR.**

```
...
PWMDRV_1chHiResUpDwnCnt_F_C(EPwm1Regs, EPwm1Regs.TBPRD, duty);
...
```

#### Module Parameters:

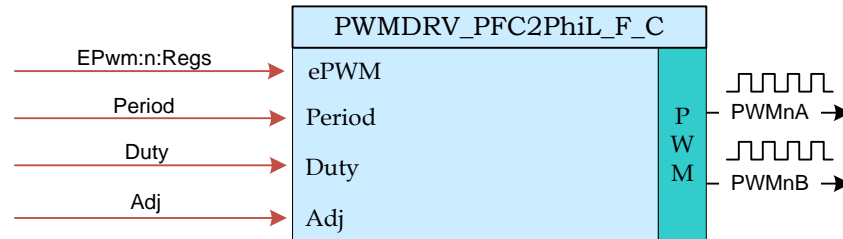
```
PWMDRV_1chHiRes_UpDwnCnt_F_C(EPwm:n:Regs, period, duty);
```



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)

### 5.3.7 PWMDRV\_PFC2PHIL\_F\_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\_F\_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:

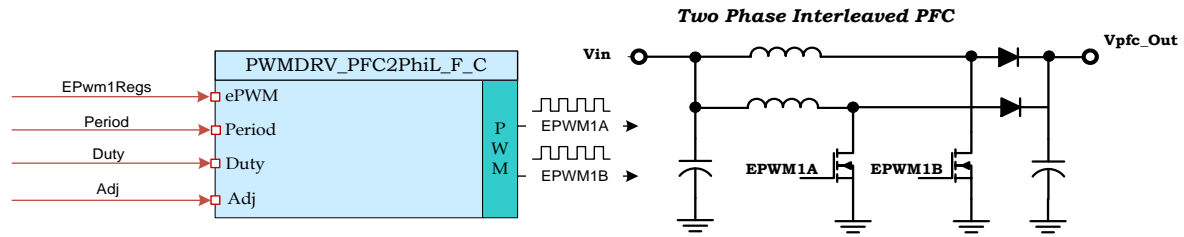
n PWM peripheral number, PWM will be configured in up-down count mode

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).

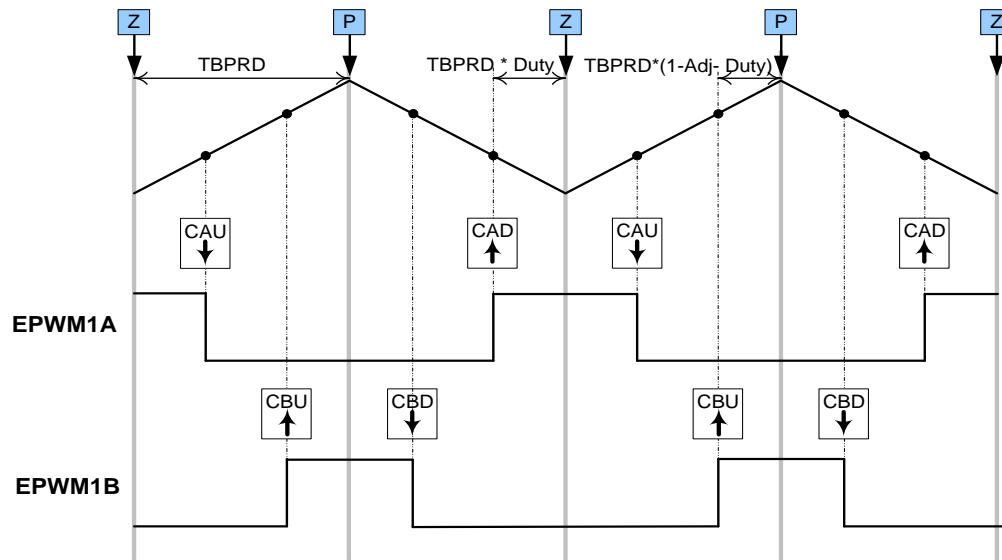
**Detailed**

**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\_F\_C module.

**Step 1 Add include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. The CNF file needs to be manually added to the project as well.

```
...
PWM_PFC2PHIL_CNF(1, 600);
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```
...
PWMDRV_PFC2PhiL_F_C(EPwm1Regs, EPwm1Regs.TBPRD, duty, adj);
...
```

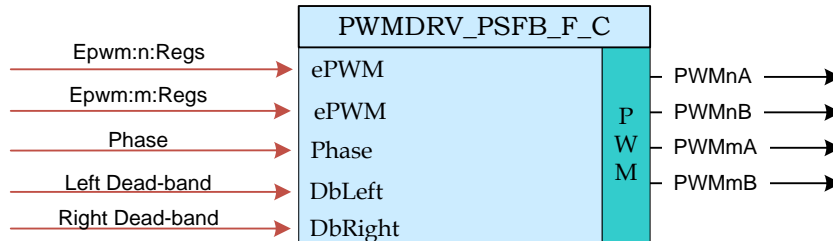
**Module Parameters:**

```
PWMDRV_PFC2PhiL_F_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)

### 5.3.8 PWMDRV\_PSFB\_F\_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\_F\_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:

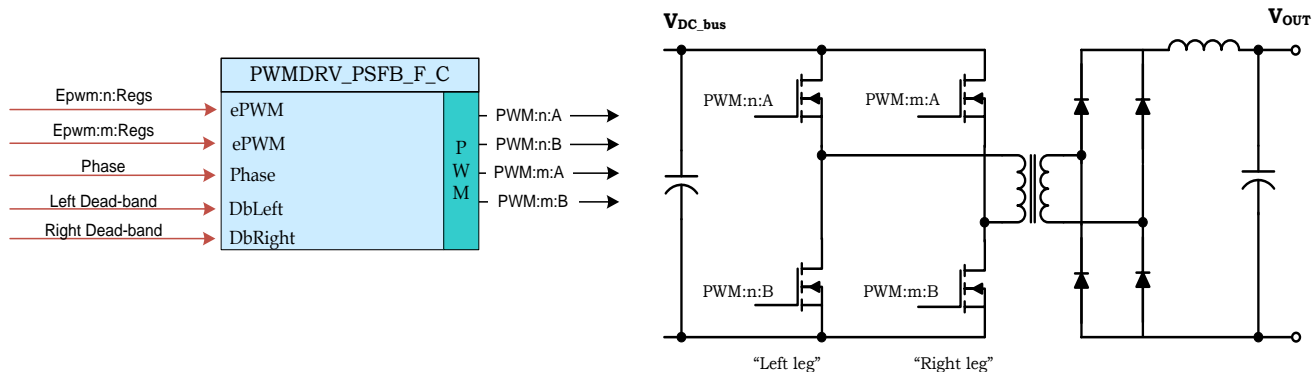
n PWM Peripheral number; PWM will be configured for PSFB topology, PWM m ( $m = n + 1$ ) is configured to work with synch pulses from PWM n module

period is the maximum count value of the PWM timer

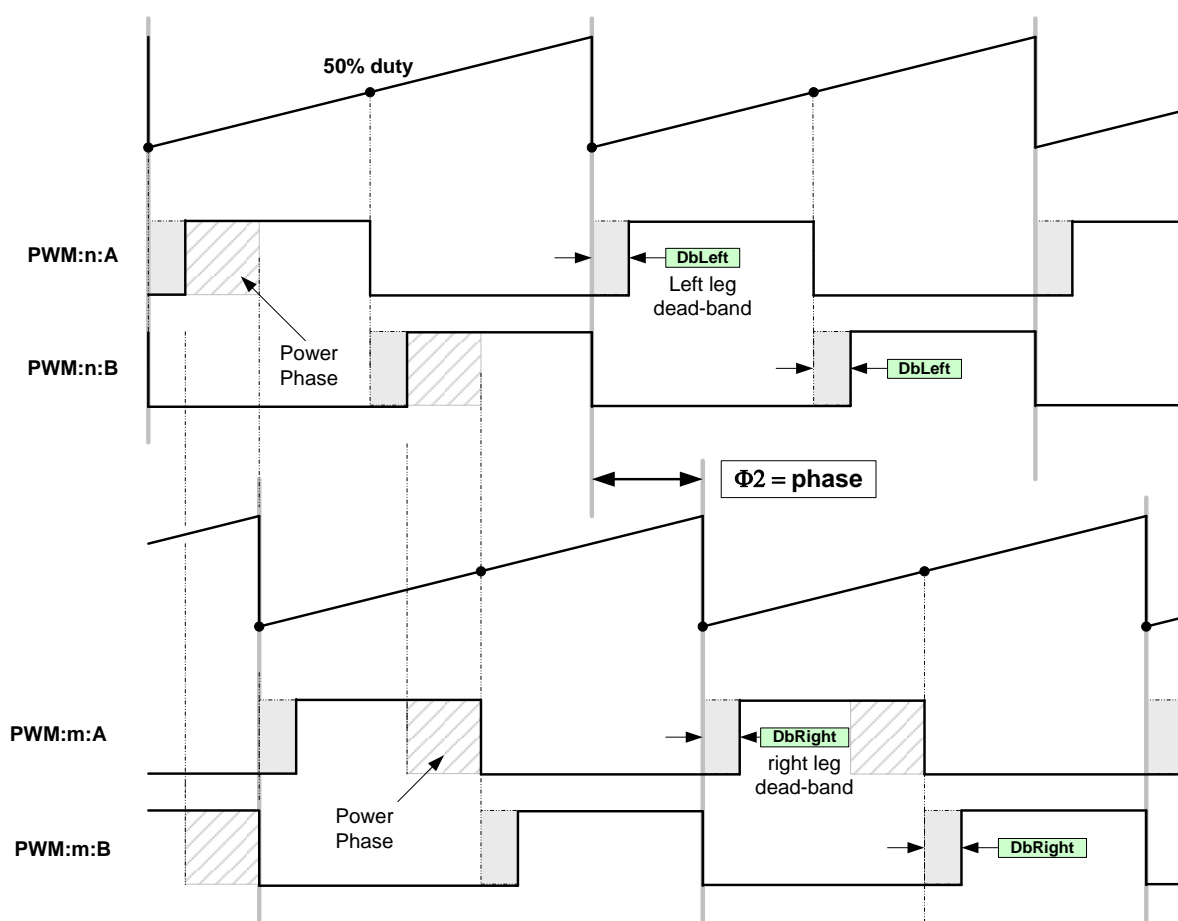
**Detailed**

**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\_F\_C module.

**Step 1 Add include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. The CNF file needs to be manually added to the project as well.

```
...
PWM_PSFB_CNF(1, 600);
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```
...
PWMDRV_PSFB_F_C(EPwm1Regs, EPwm2Regs, phase, dbLeft, dbRight);
...
```

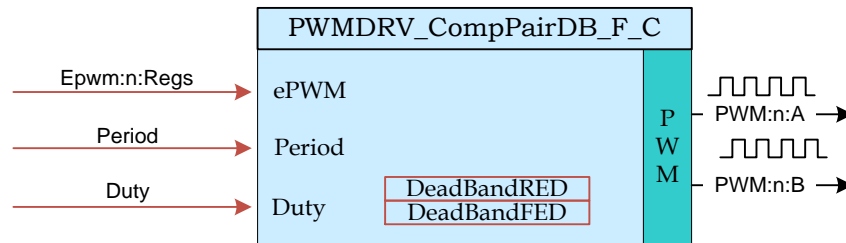
**Module Parameters:**

```
PWMDRV_PSFB_F_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

### 5.3.9 PWMDRV\_CompPairDB\_F\_C : PWM Driver with complementary chA & 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\_F\_C.h

**Peripheral**

**Initialization File:** PWM\_CompPairDB\_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\_CompPairDB\_Cnf.c. The file defines the function:

```
void PWM_CompPairDB_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

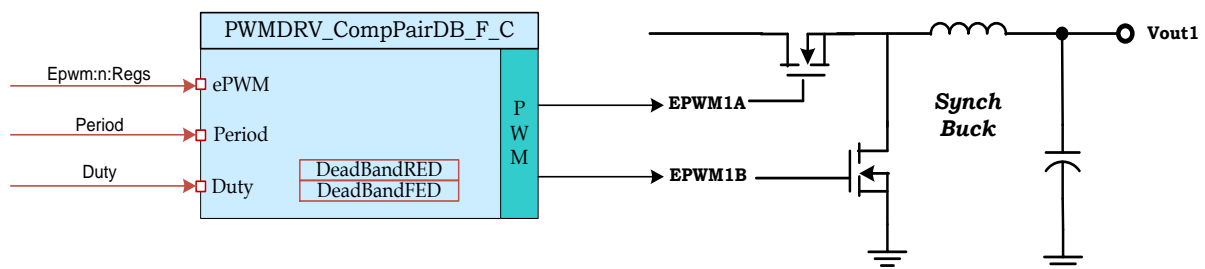
DbRed        is the new rising edge delay

DbFed        is the new falling edge delay

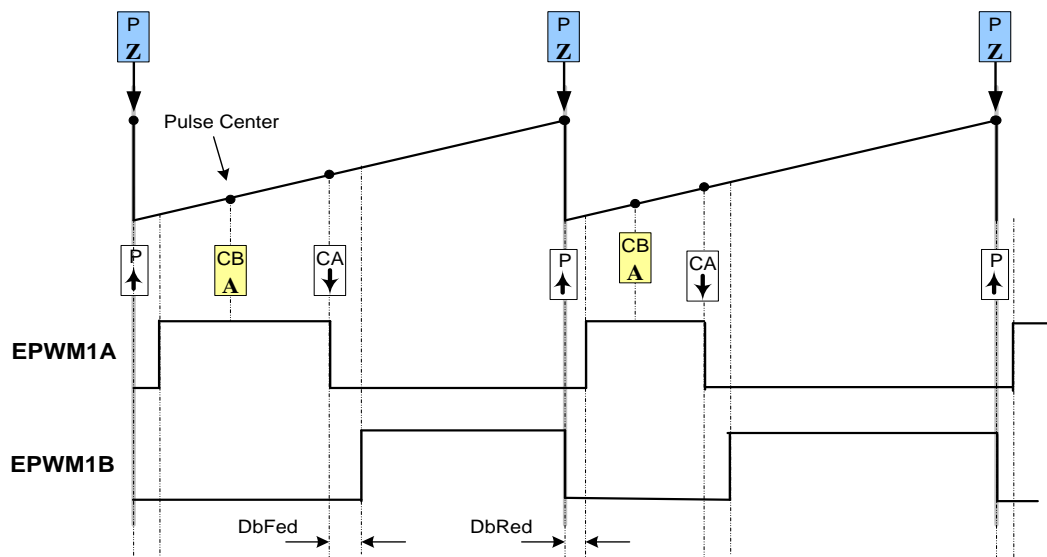
Alternatively a CLA macro is provided to update the dead-band; if this needs to be done at a faster rate inside the ISR. The dead-band update macro is called PWMDRV\_ComplPairDB\_F\_C\_UpdatedB.

### 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\_F\_C module.

**Step 1 Add include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. The CNF file needs to be manually added to the project as well.

```
...
PWM_ComplPairDB_CNF(1, 600, 1, 0);
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```
...
PWMDRV_ComplPairDB_F_C(EPwm1Regs, EPwm1Regs.TBPRD, duty);
...
PWMDRV_ComplPairDB_F_C_UpdateDB(Epwm1Regs, dbRed, dbFed);
...
```

#### Module Parameters:

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

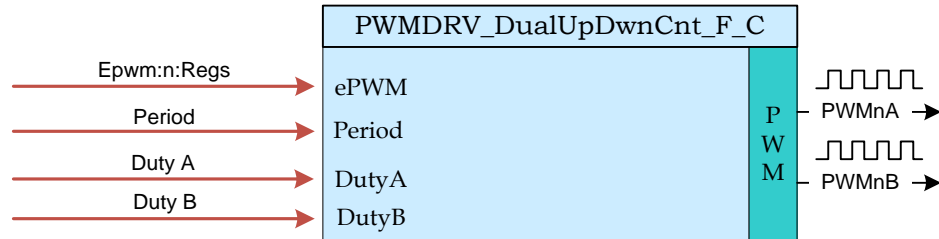
```
PWMDRV_ComplPairDB_F_C_UpdateDB(Epwm:n:Regs, dbRed, dbFed);
```

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

### 5.3.10 PWMDRV\_DualUpDwnCnt\_F\_C : Dual PWM Driver , independent 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.

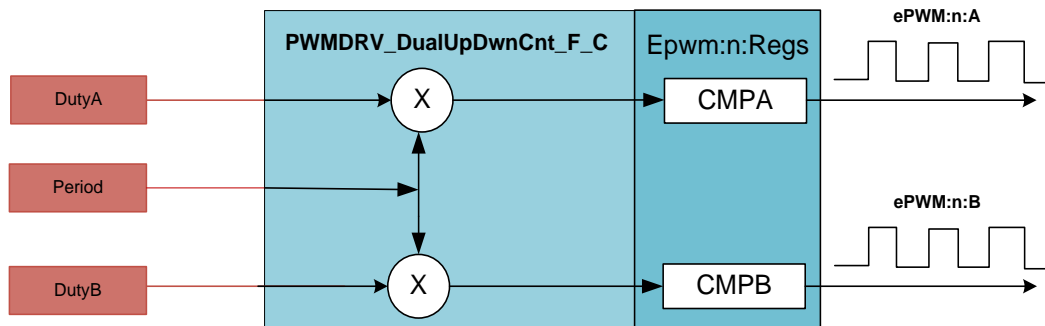


**Macro File:** PWMDRV\_F\_C.h

**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

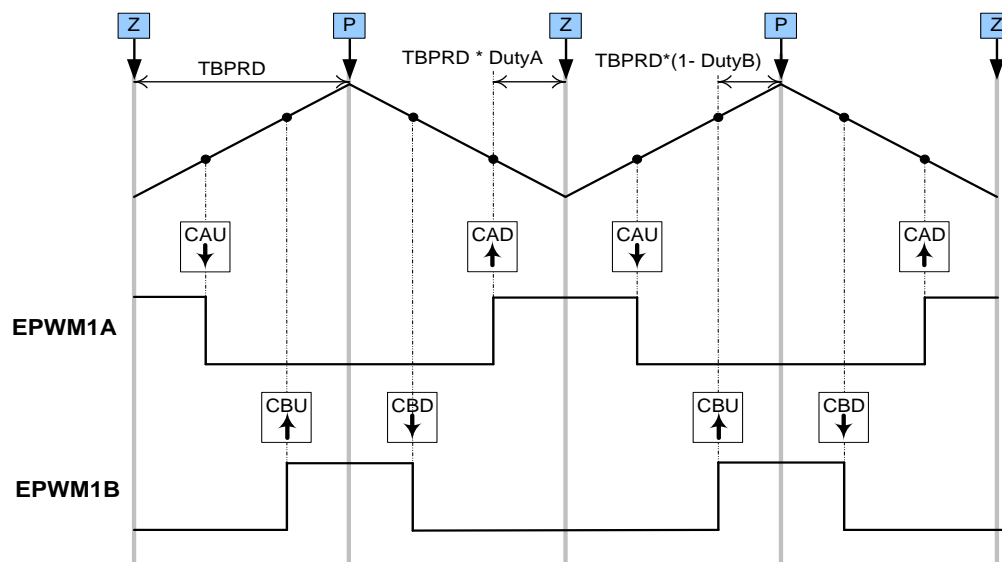
mode determines whether the PWM is to be configured as slave or master, when configured as the master the TBSYNC signal is ignored by the PWM module.

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  $(\text{System Clock}/\text{Switching Frequency})/2 = 300$  must be used in the CNF function.



**PWM generation with the EPWM module.**

**Usage:** This section explains how to use the PWMDRV\_DualUpDwnCnt\_F\_C module.

**Step 1 Add include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. The CNF file needs to be manually added to the project as well.

```
...
PWM_DualUpDwnCnt_CNF(1, 300, 1, 0);
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```
...
PWMDRV_DualUpDwnCnt_F_C(EPwm1Regs, EPwm1Regs.TBPRD, dutyA, dutyB);
...
```

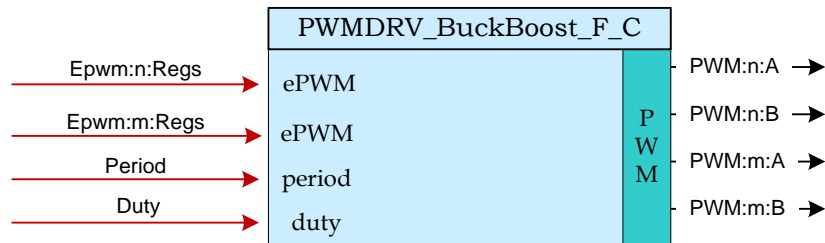
**Module Parameters:**

```
PWMDRV_DualUpDwnCnt_F_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)

### 5.3.11 PWMDRV\_BuckBoost\_F\_C : Dual PWM Driver, independent Duty on chA & chB

**Description:** This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a duty on PWM channel A, and 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_F_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_F_C_UpdateDB (int16 n, int16 DbRed, int16 DbFed)
```

Where:

n is the PWM Peripheral number

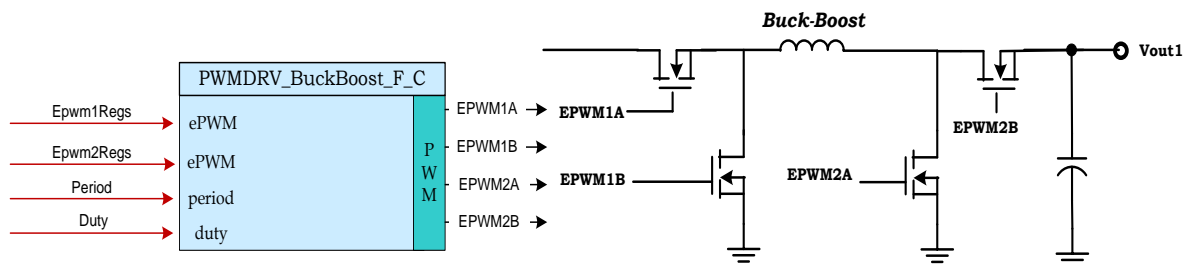
DbRed is the new rising edge delay

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_F_C_UpdateDB` macro can be used.

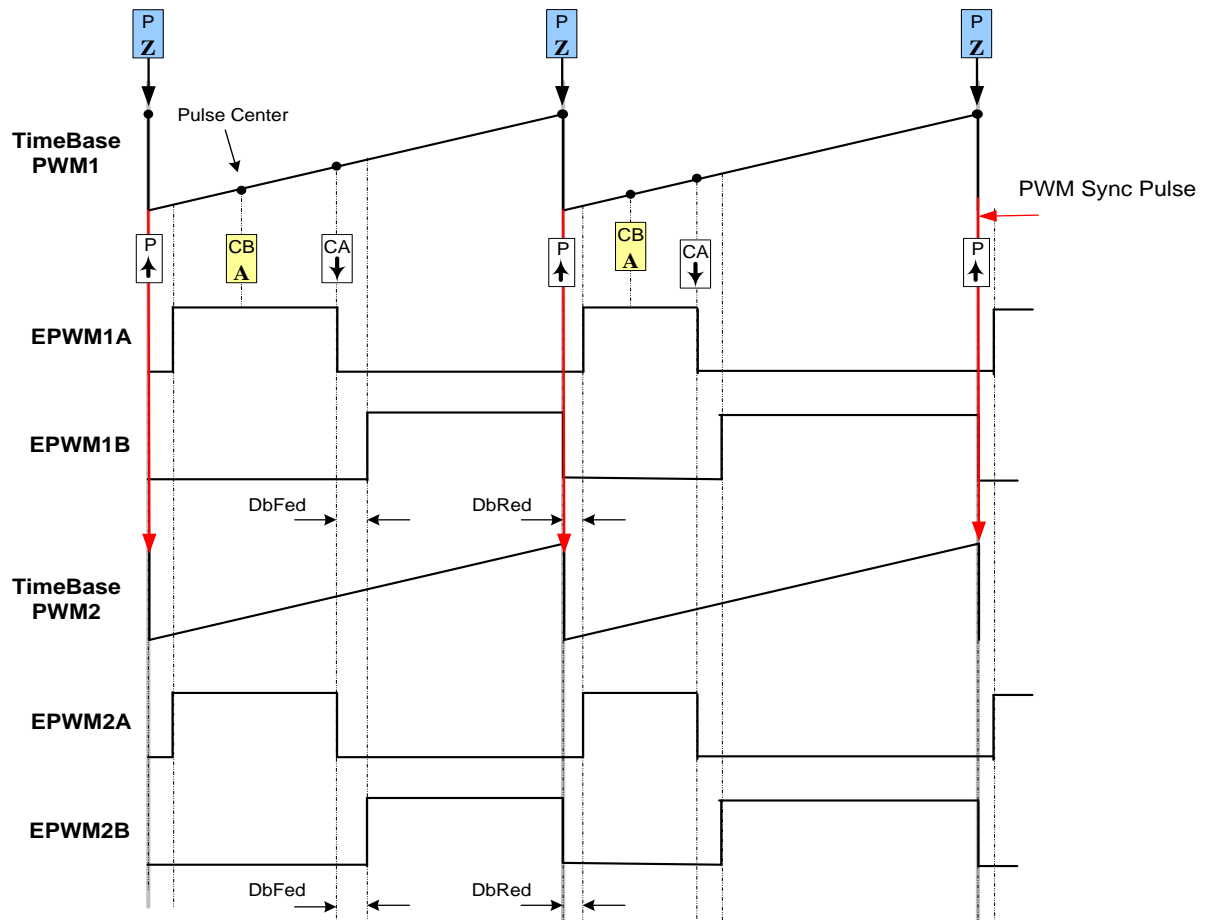
### Detailed Description

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\_F\_C module.

**Step 1 Add include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. The CNF file needs to be manually added to the project as well.

```
...
PWM_BuckBoost_CNF(1, 2, 600);
PWM_BuckBoost_F_C_UpdateDB(1, dbRED, dbFED);
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```

...
PWMDRV_BuckBoost_F_C(EPwm1Regs, EPwm2Regs, EPwm1Regs.TBPRD, duty);
...
PWMDRV_BuckBoost_F_C_UpdatedB(EPwm1Regs, dbRed, dbFed);...

```

#### Module Parameters:

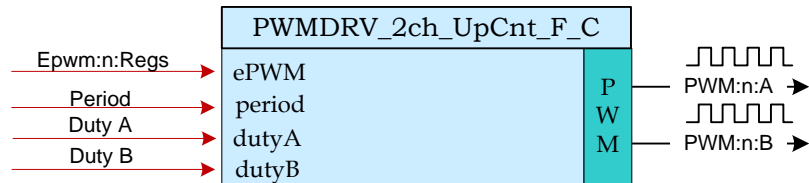
```
PWMDRV_BuckBoost_F_C(EPwm:n:Regs, EPwm:m:Regs, period, duty);
```

```
PWMDRV_BuckBoost_F_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 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

### 5.3.12 PWMDRV\_2ch\_UpCnt\_F\_C : Dual PWM Driver, independent Duty on chA and chB

**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.

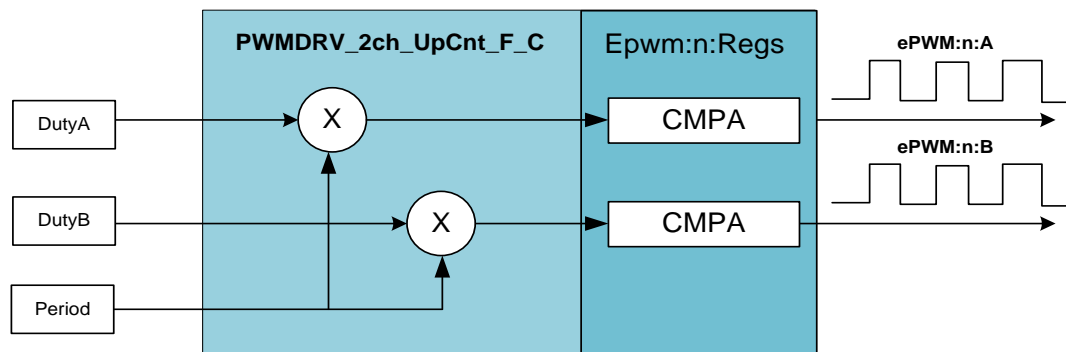


**Macro File:** PWMDRV\_F\_C.h

**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

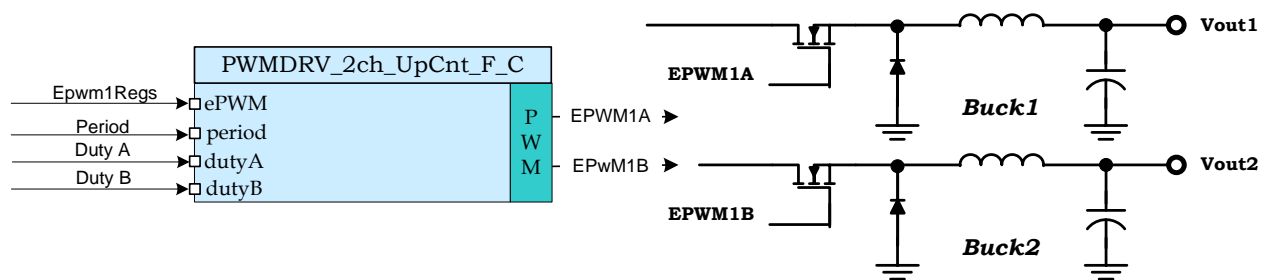
Mode = 0 PWM configured as slave

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.

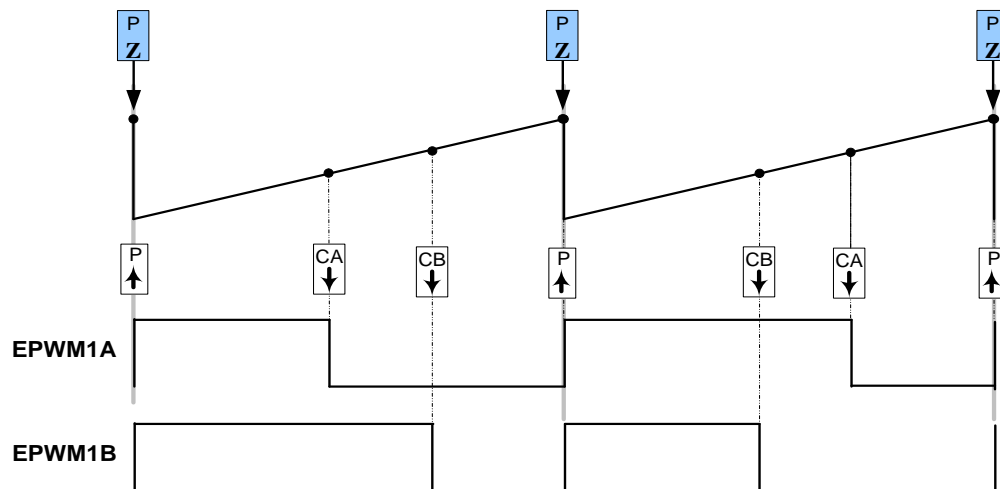
The function configures the PWM peripheral in up-count mode. The figure below shows, with help of a timing diagram, how the PWM is configured to generate the waveforms.

## 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  $(\text{System Clock}/\text{Switching Frequency}) = 600$ . 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\_F\_C module.

**Step 1 Add include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. The CNF file needs to be manually added to the project as well.

```
...  
PWM_2ch_UpCnt_CF(1, 600 , 1, 0);  
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```
...  
PWMDRV_2ch_UpCnt_F_C(EPwm1Regs, EPwm1Regs.TBPRD, dutyA, dutyB);  
...
```

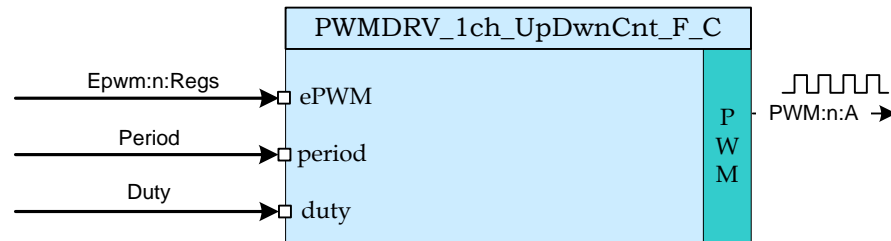
**Module Parameters:**

```
PWMDRV_2ch_UpCnt_F_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)

### 5.3.13 PWMDRV\_1ch\_UpDwnCnt\_F\_C : PWM Driver, Duty chA 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 input value DutyA. The waveform is centered at the zero value of the up-down count time base of the PWM.

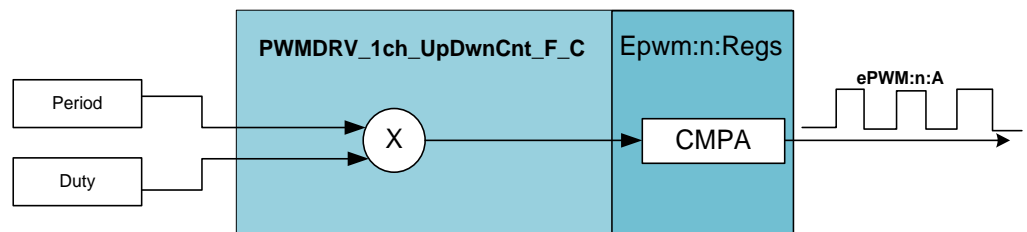


**Macro File:** PWMDRV\_F\_C.h

**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

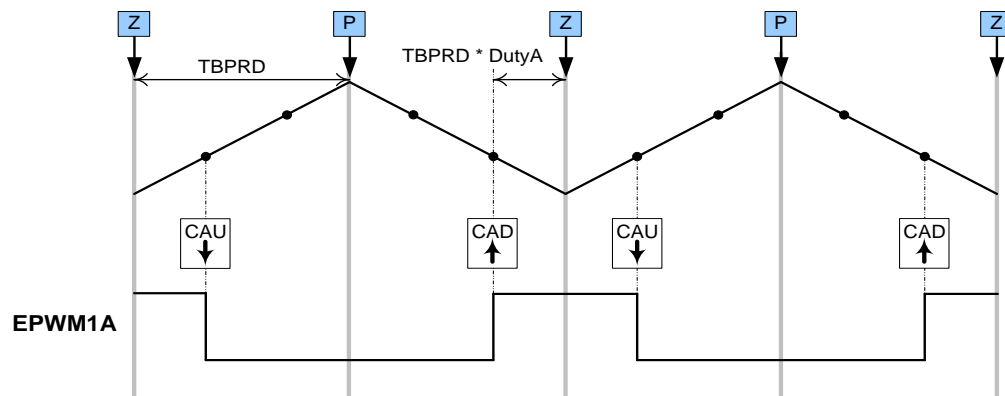
mode determines whether the PWM is to be configured as slave or master, when configured as the master the TBSYNC signal is ignored by the PWM module.

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  $(\text{System Clock}/\text{Switching Frequency}) = 600$  must be used for the CNF function.



**PWM generation with the EPWM module.**

**Usage:** This section explains how to use the PWMDRV\_1ch\_UpDwnCnt\_F\_C module.

**Step 1 Add include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. The CNF file needs to be manually added to the project as well.

```
...
PWM_1ch_UpDwnCnt_CNF(1, 600, 1, 0);
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```
...
PWMDRV_1ch_UpDwnCnt_F_C(EPwm1Regs, EPwm1Regs.TBPRD, duty);
...
```

#### Module Parameters:

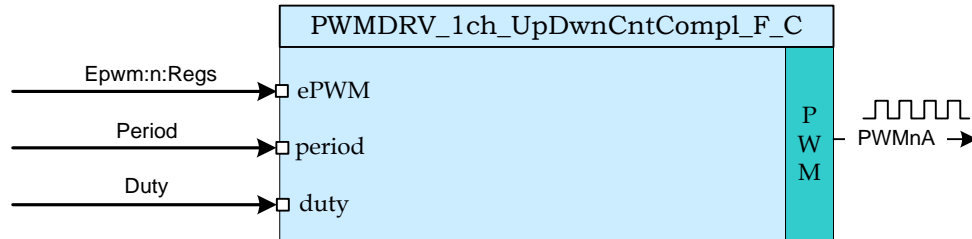
```
PWMDRV_1ch_UpDwnCnt_F_C(EPwm:n:Regs, period, duty);
```

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)



### 5.3.14 PWMDRV\_1ch\_UpDwnCntCompl\_F\_C : PWM Driver, Duty chA 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\_F\_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.

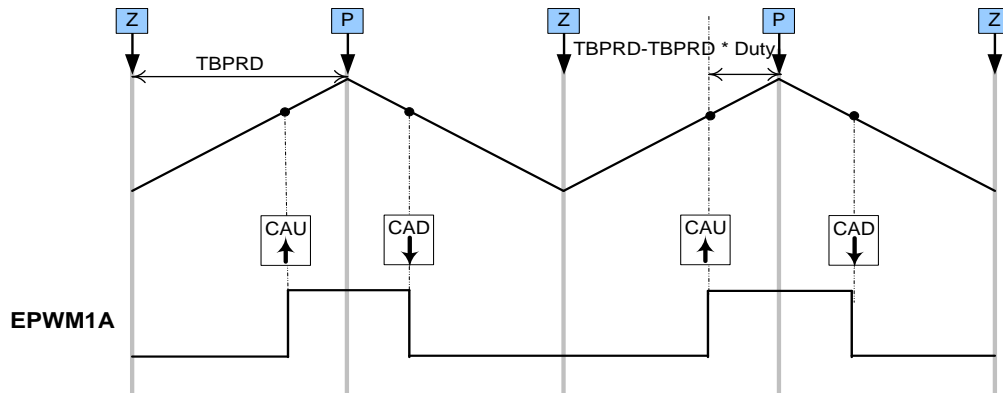
             Mode = 1 PWM configured as a master

             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\_F\_C module.

**Step 1 Add include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. The CNF file needs to be manually added to the project as well.

```
...
PWM_1ch_UpDwnCntCompl_CNF(1, 600 , 1, 0);
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```
...
PWMDRV_1ch_UpDwnCntCompl_F_C(EPwm1Regs, EPwm1Regs.TBPRD, duty);
...
```

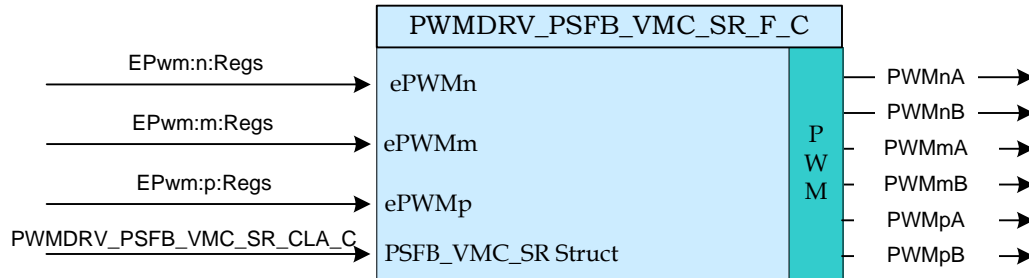
#### Module Parameters:

```
PWMDRV_1ch_UpDwnCntCompl_F_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)

### 5.3.15 PWMDRV\_PSFB\_VMC\_SR\_F\_C : PWM Driver for VMC 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\_F\_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.

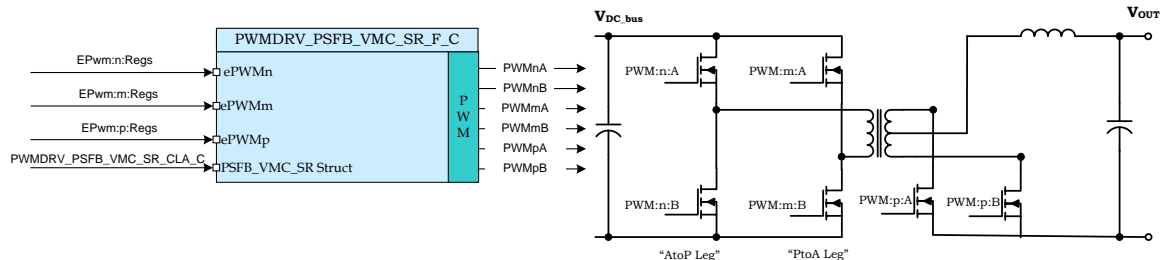
period is the maximum count value of the PWM timer

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.

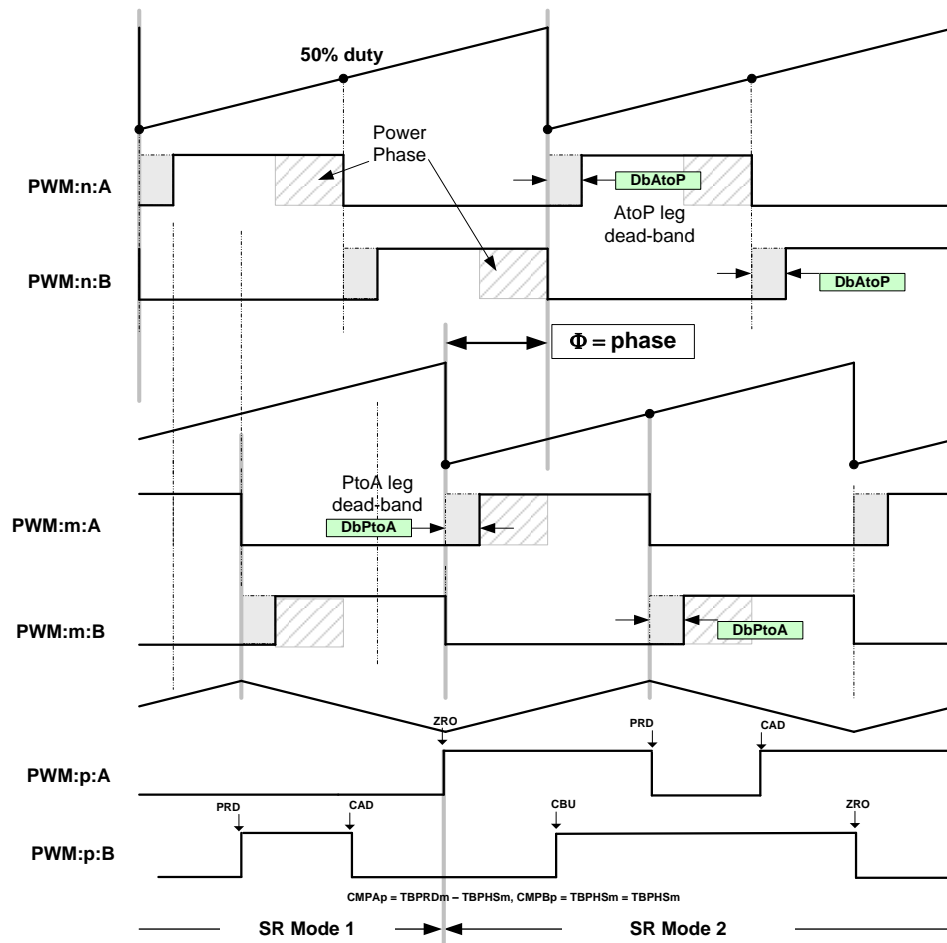
Comp1\_Prot Enables catastrophic protection based on on-chip comparator1 and DAC.

## 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\_PSFBC\_VMC\_SR\_F\_C module.

**Step 1 Add include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Declare structure** in the source file

```
/* DPLib variables */
PWMDRV_PSFBC_VMC_SR_F_C psfbvmc;
```

In case of CLA specify memory locations, by default CLA data section will be used.

```
/* DPLib variables memory locations */
#pragma DATA_SECTION(psfbvmc, "Cla1ToCpuMsgRAM")
```

**Step 3 For CLA add shared declaration** to {ProjectName}-CLA\_Shared.h if variables are shared between CPU and CLA.\

```
/* DPLib variables */
extern volatile PWMDRV_PSFBC_VMC_SR_F_C psfbvmc;
```

**Step 4 Macro initialization** For CLA edit the task8 with the initialization routine shown below. This task found in the {ProjectName}-CLA\_Tasks.cla. For C28x call the init function in the main file before enabling interrupts

```
...
PWMDRV_PSFBC_VMC_SR_F_C_INIT(psfbvmc);
...
```

**Step 5 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. The CNF file needs to be manually added to the project as well.

```
...
PWMDRV_PSFBC_VMC_SR_CNF(1,200,1,1)
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```
...
psfbvmc.phase = phase;
psfbvmc.dbPtoA = deadbandPtoA;
psfbvmc.dbAtoP = deadbandAtoP;
PWMDRV_PSFBC_VMC_SR_F_C(psfbvmc, EPwm1Regs, EPwm2Regs, EPwm4Regs);
...
```

**Object Definition:**

```

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

    // Inputs
    float32 phase;
    float32 dbPtoA;
    float32 dbAtoP;
} PWMDRV_PFSB_VMC_SR_F_C;

```

**Special Constants and Data types****PWMDRV\_PFSB\_VMC\_SR\_F\_C**

Structure defined to store module values. To create multiple instances of the module simply declare variables of type PWMDRV\_PFSB\_VMC\_SR\_F\_C.

**Module Structure Definition:**

```

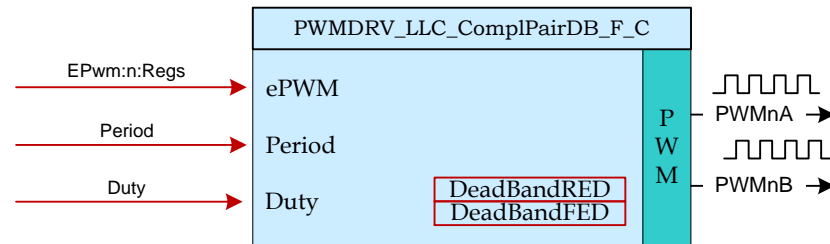
PWMDRV_PFSB_VMC_SR_F_C(PWMDRV_PFSB_VMC_SR_F_C, EPwm:n:Regs, EPwm:m:Regs,
EPwm:p:Regs);

```

Parameter Name	Types	Description	Acceptable Range
PWMDRV_PFSB_VMC_SR_F_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: $m = (n + 3)$

### 5.3.16 PWMDRV\_LLC\_ComplPairDB\_F\_C : PWMDriver 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\_F\_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

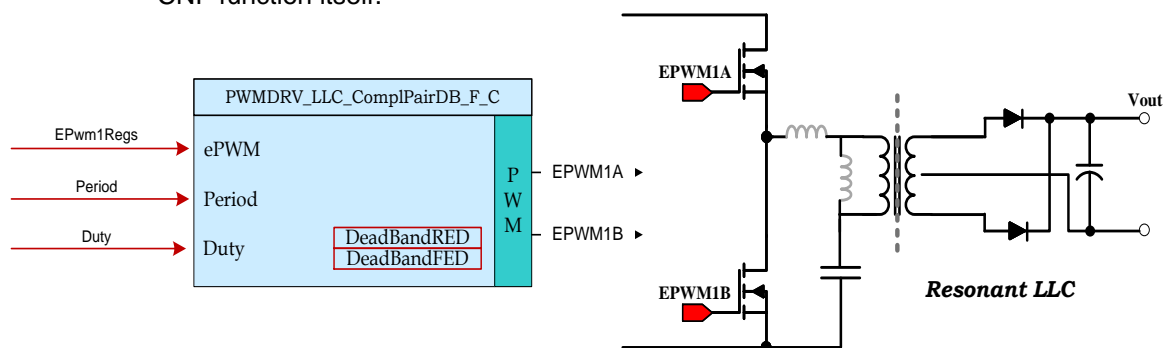
DbRed is the new rising edge delay

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\_F\_C\_UpdateDB.

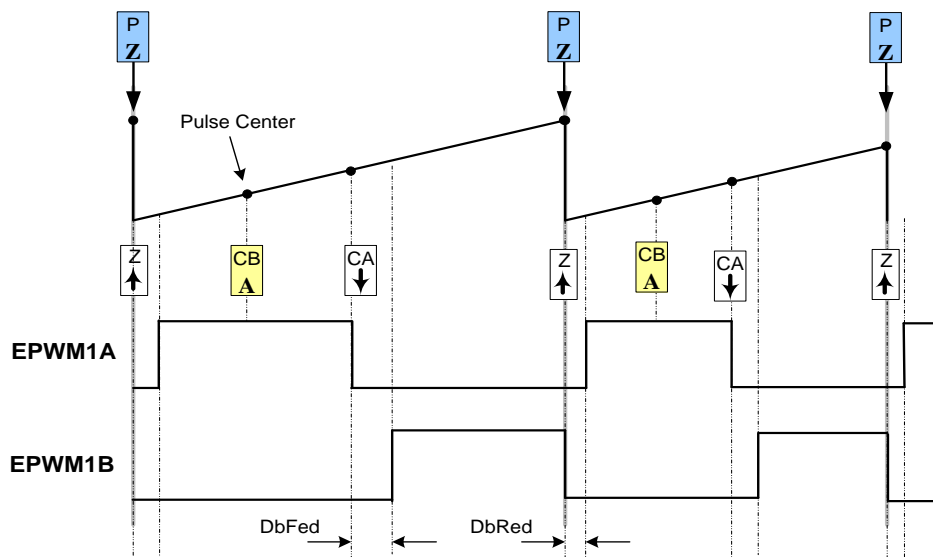
## Detailed Description

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\_F\_C module**





**PWM generation for PWMDRV\_LLC\_CompPairDB Macro**

**Usage:** This section explains how to use the PWMDRV\_LLC\_CompPairDB\_F\_C module.

**Step 1 Add include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. The CNF file needs to be manually added to the project as well.

```
...
PWM_CompPairDB_CNF(1, 600, 1, 0);
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```
...
PWMDRV_LLC_CompPairDB_F_C(EPwm1Regs, EPwm1Regs.TBPRD, duty);
...
```

#### Module Parameters:

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

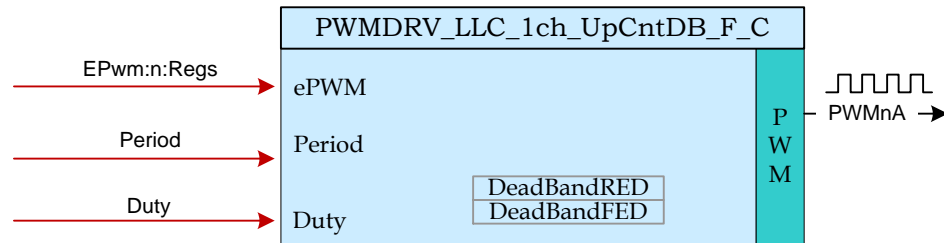
```
PWMDRV_LLC_CompPairDB_F_C_UpdateDB(Epwm:n:Regs, dbRed, dbFed);
```

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	Uint16

is driven on the PWM module.			
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

### 5.3.17 PWMDRV\_LLC\_1ch\_UpCntDB\_F\_C : PWM Driver for chA PWM edge shift, period mode

**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\_F\_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

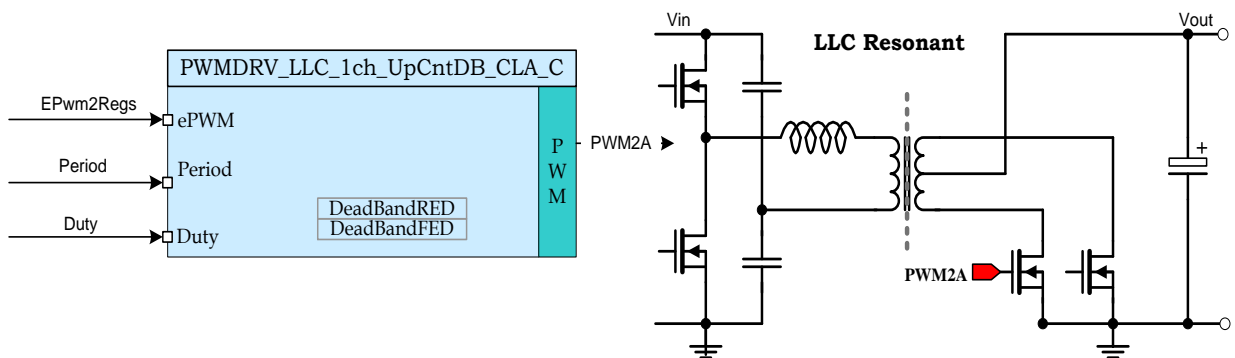
DbRed is the new rising edge delay

DbFed is the new falling edge delay

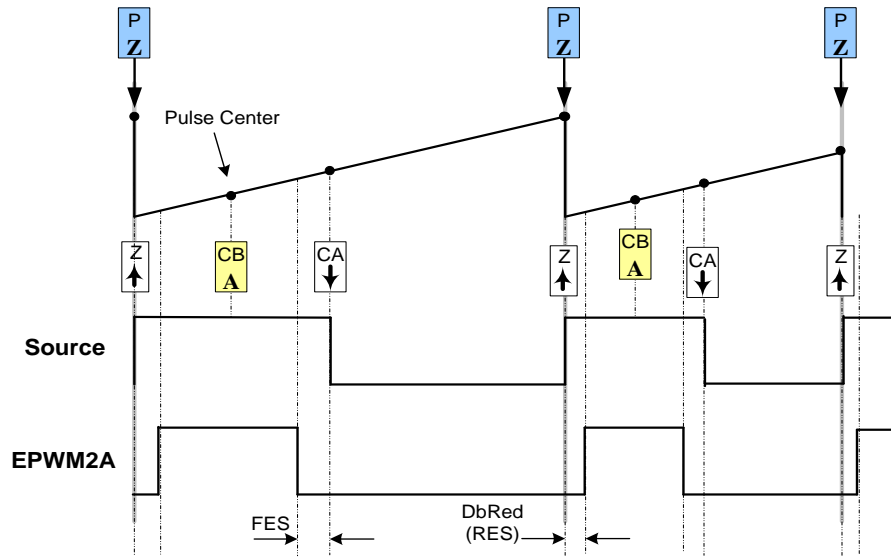
Alternatively, a macro is provided to update the dead-band if the update needs to be done at a faster rate; inside the ISR. The dead-band update macro is called `PWMDRV_LLC_1ch_UpCntDB_F_C_UpdatedB`.

#### 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  $(\text{System Clock}/\text{Switching Frequency}) = 600$ . 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 include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. The CNF file needs to be manually added to the project as well.

```
...
PWM_1ch_UpCntDB_CNF(1, 600, 1, 0);
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```
...
PWMDRV_LLC_1ch_UpCntDB_F_C(EPwm1Regs, period, duty);
...
PWMDRV_LLC_1ch_UpCntDB_F_C_UpdateDB(Epwm1Regs, dbRed, dbFed);
...
```

#### Module Parameters:

```
PWMDRV_LLC_1ch_UpCntDB_F_C(EPwm:n:Regs, period, duty);
```

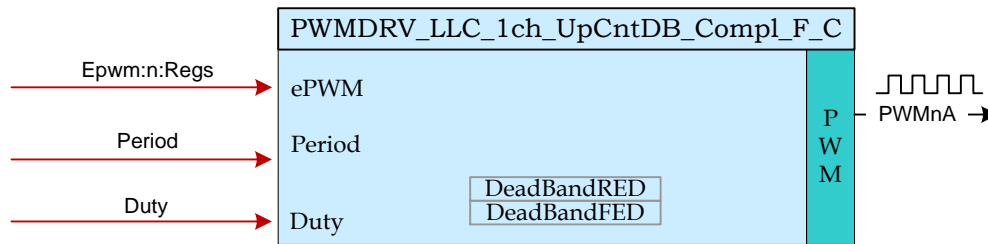
```
PWMDRV_LLC_1ch_UpCntDB_F_C_UpdateDB(Epwm:n:Regs, dbRed, dbFed);
```

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

### 5.3.18 PWMDRV\_LLC\_1ch\_UpCntDB\_Compl\_F\_C : PWM Driver for chA PWM edge shift, period mode

**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\_F\_C.h.



**Macro File:** PWMDRV\_F\_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

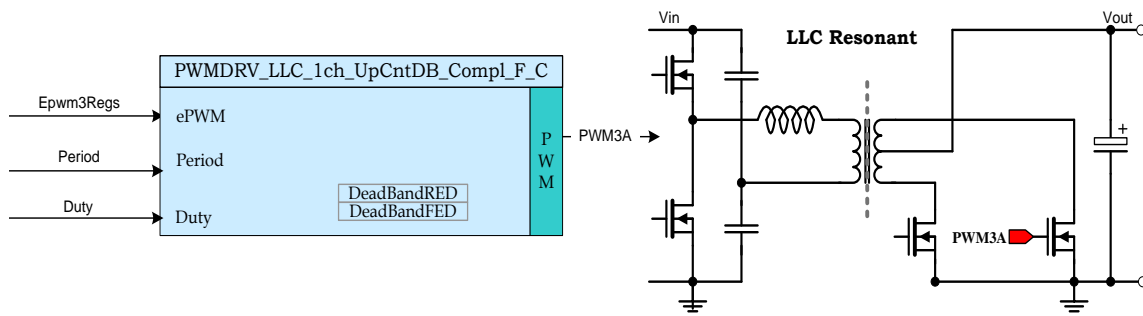
DbFed        is the new falling edge delay

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_F_C_UpdateDB`.

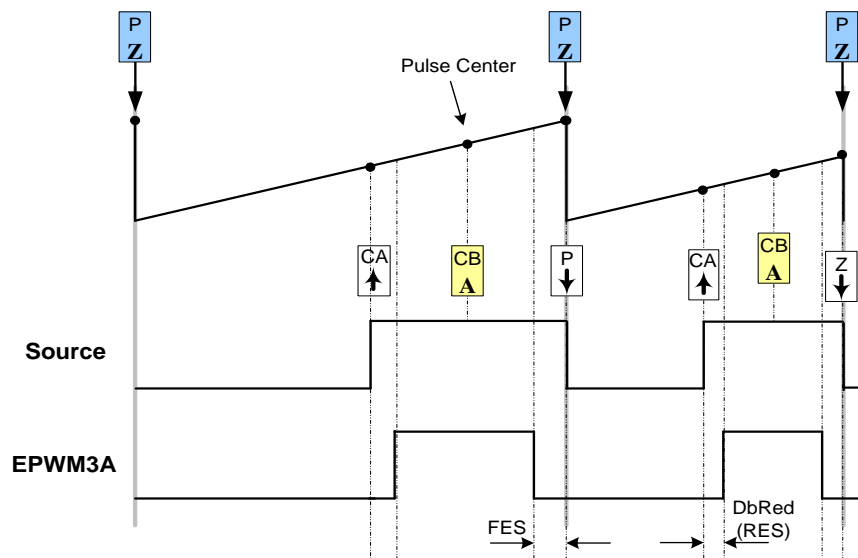
## Detailed Description

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\_Cmpl module**



**PWM generation for PWMDRV\_LLC\_1ch\_UpCntDB\_Cmpl Macro**

**Usage:** This section explains how to use the PWMDRV\_LLC\_1ch\_UpCntDB\_Cmpl\_CLA\_C module.

**Step 1 Add include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. The CNF file needs to be manually added to the project as well.

```
...
PWM_1ch_UpCntDB_Cmpl_CNf(1, 600, 1, 0);
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```
...  
PWMDRV_LLC_1ch_UpCntDB_Comp1_F_C(EPwm1Regs, period, duty);  
...  
PWMDRV_LLC_1ch_UpCntDB_Comp1_F_C_UpdatedDB(Epwm1Regs, dbRed, dbFed);  
...
```

**Module Parameters:**

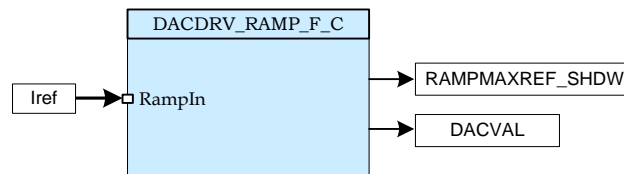
PWMDRV\_LLC\_1ch\_UpCntDB\_Comp1\_F\_C(EPwm:n:Regs, period, duty);

PWMDRV\_LLC\_1ch\_UpCntDB\_Comp1\_F\_C\_UpdatedDB(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

### 5.3.18 DACDRV\_RAMP\_F\_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\_F\_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.

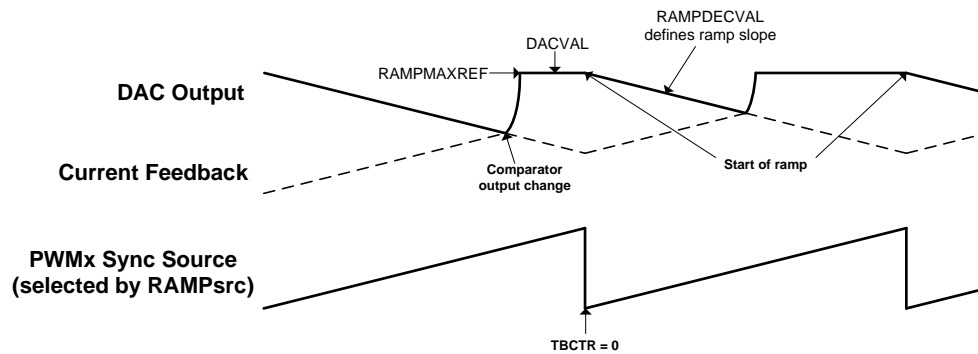
DACsrc                Selects DACval or internal ramp as DAC source

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.

**Usage:** This section explains how to use the PWMDRV\_LLC\_1ch\_UpCntDB\_CLA\_C module.

**Step 1 Add include for DPLib header file** to the source file and add the DPLib include and C\_macros folder to the include path of the project.

**Step 2 Peripheral initialization** Call the CNF function in the main file before enabling interrupts or CLA tasks. The CNF file needs to be manually added to the project as well.

```
...
DacDrvCnf(1, 1280, 1, 2, slope);
...
```

**Step 3 Call** the macro within a CLA-Task or C28x ISR.

```
...
DACDRV_RAMP_F_C(Comp1Regs, RampIn);
...
```

#### Module Parameters:

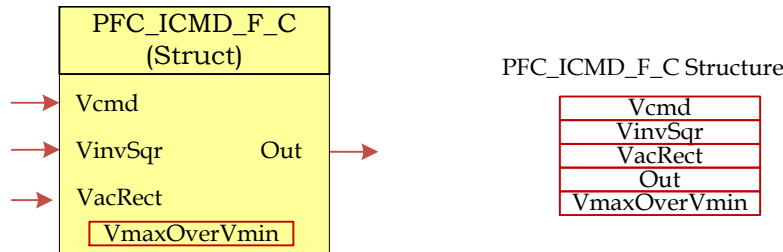
```
DACDRV_RAMP_F_C(Comp:n:Regs, RampIn);
```

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

## 4.4. Application Specific

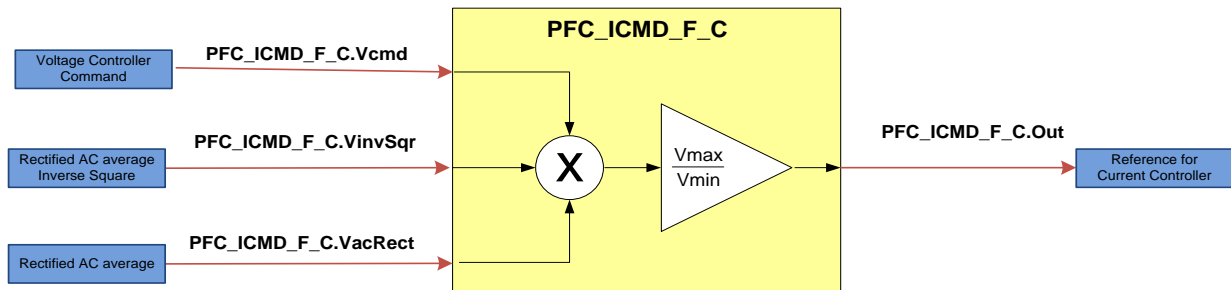
### 5.4.1 PFC\_ICMD\_F\_C : Current Command for Power Factor Correction

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



**Macro File:** PFC\_ICMD\_F\_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\_F\_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\_F\_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\_F\_C module.

**Step 1 Add include for DPLib header file to the source file and add the DPLib include and C\_macros folder to the include path of the project.**

**Step 2 Declare structure** in the source file

```
/* DPLib variables */
PFC_ICMD_F_C pfc_icmd1;
```

In case of CLA specify memory locations, by default CLA data section will be used.

```
/* DPLib variables memory locations */
#pragma DATA_SECTION(pfc_icmd1, "Cla1ToCpuMsgRAM")
```

**Step 3 For CLA add shared declaration** to {ProjectName}-CLA\_Shared.h if variables are shared between CPU and CLA.

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

**Step 4 Macro initialization** For CLA edit the task8 with the initialization routine shown below. This task found in the {ProjectName}-CLA\_Tasks.cla. For C28x call the init function in the main file before enabling interrupts

```
...
PFC_ICMD_F_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 Call** the macro within a CLA-Task of C28x ISR.

```
...
pfc_icmd1.Vcmd = Command;
pfc_icmd1.VinvSqr = VinvS;
pfc_icmd1.VacRect = Vrect;
PFC_ICMD_F_C(pfc_icmd1);
Out = pfc_icmd1.Out;
...
```

**Object Definition:**

```

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

```

**Special Constants and Data types****PFC\_ICMD\_F\_C**

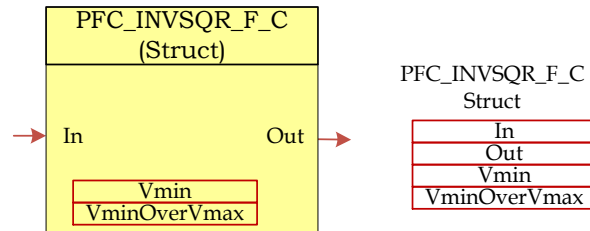
Structure defined to store module values. To create multiple instances of the module simply declare variables of type PFC\_ICMD\_F\_C.

**Module Structure Definition:**

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

## 5.4.2 PFC\_INVSQR\_F\_C : Inverse Square Math Block for Power Factor Correction

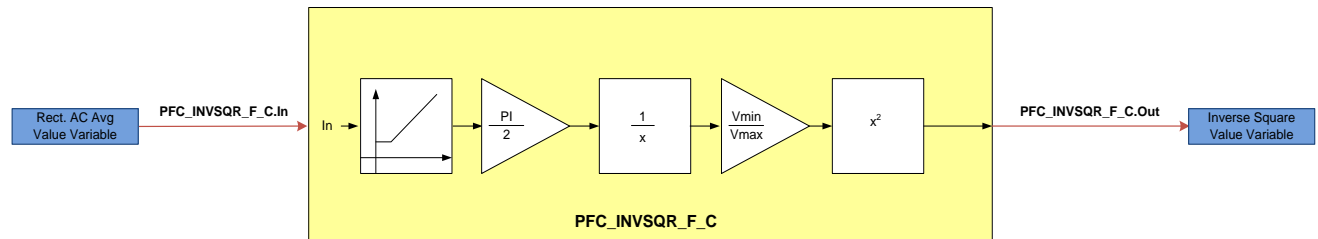
**Description:** This software module performs a reciprocal function on a scaled unipolar input signal and squares it.



**Macro File:** PFC\_INVSQR\_F\_C.h

**Technical:** The input variable must be in normalized float format and copied into the PFC\_INVSQR\_F\_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\_F\_C module.

**Step 1 Add include for DPLib header file to the source file and add the DPLib include and C\_macros folder to the include path of the project.**

**Step 2 Declare structure** in the source file

```
/* DPLib variables */
PFC_INVSQR_F_C pfc_invsqr1;
```

In case of CLA specify memory locations, by default CLA data section will be used.

```
/* DPLib variables memory locations */
#pragma DATA_SECTION(pfc_invsqr1, "Cla1ToCpuMsgRAM")
```

**Step 3 For CLA add shared declaration** to {ProjectName}-CLA\_Shared.h if variables are shared between CPU and CLA.

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

**Step 4 Macro initialization** For CLA edit the task8 with the initialization routine shown below. This task found in the {ProjectName}-CLA\_Tasks.cla. For C28x call the init function in the main file before enabling interrupts

```
...
PFC_INVSQR_F_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 Call** the macro within a CLA-Task of C28x ISR.

```
...
pfc_invsqr1.In = Command;
PFC_INVSQR_F_C(pfc_invsqr1);
Out = pfc_invsqr1.Out;
...
```

**Object Definition:**

```

/**
 * PFC_INVSQR structure
 */
typedef struct {
    float32 In;
    float32 Out;
    float32 Vmin;
    float32 VminOverVmax;
} PFC_INVSQR_F_C;

```

**Special Constants and Data types****PFC\_INVSQR\_F\_C**

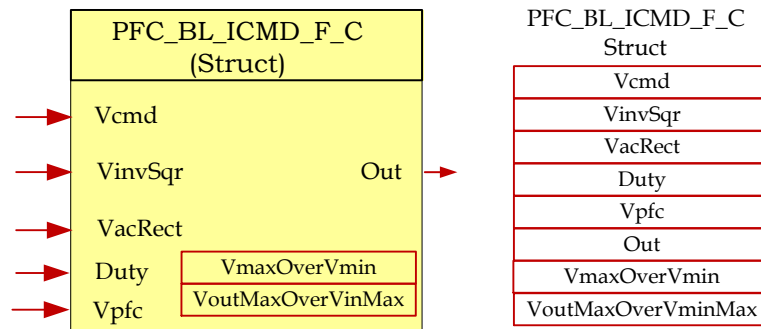
Structure defined to store module values. To create multiple instances of the module simply declare variables of type PFC\_INVSQR\_F\_C.

**Module Structure Definition:**

Parameter Name	Types	Description	Acceptable Range
In	Input	Input voltage.	float32 [0, 1)
Out	Output	PFC_INVSQR_F_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)

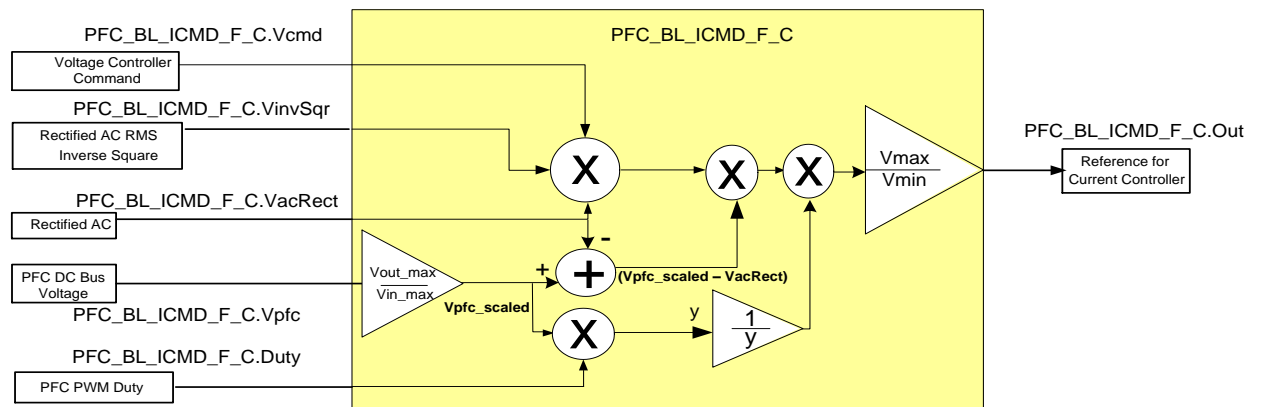
### 5.4.3 PFC\_BL\_ICMD\_F\_C : Bridgeless PFC Current Command

**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\_F\_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\_F\_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\_F\_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\_F\_C module.

**Step 1 Add include for DPLib header file to the source file and add the DPLib include and C\_macros folder to the include path of the project.**

**Step 2 Declare structure** in the source file

```
/* DPLib variables */
PFC_BL_ICMD_CLA_C pfc_bl_icmd1;
```

In case of CLA specify memory locations, by default CLA data section will be used.

```
/* DPLib variables memory locations */
#pragma DATA_SECTION(pfc_bl_icmd1, "Cla1ToCpuMsgRAM")
```

**Step 3 For CLA add shared declaration** to {ProjectName}-CLA\_Shared.h if variables are 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 Macro initialization** For CLA edit the task8 with the initialization routine shown below. This task found in the {ProjectName}-CLA\_Tasks.cla. For C28x call the init function in the main file before enabling interrupts

```
...
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 Call** the macro within a CLA-Task of C28x ISR.

```
...
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;
...
```

**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_F_C;

```

**Special Constants and Data types****PFC\_BL\_ICMD\_F\_C**

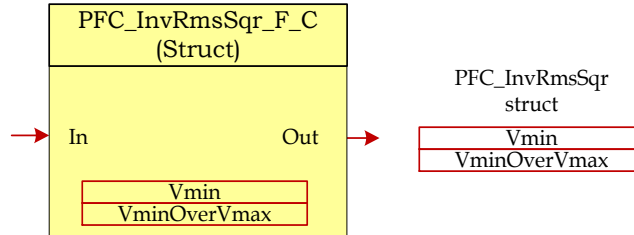
Structure defined to store module values. To create multiple instances of the module simply declare variables of type PFC\_BL\_ICMD\_F\_C.

**Module Structure Definition:**

Parameter Name	Types	Description	Acceptable Range
Vcmd	Input	Voltage controller output.	float32 [0, 1)
VinvSqr	Input	PFC_InvRmsSqr_F_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

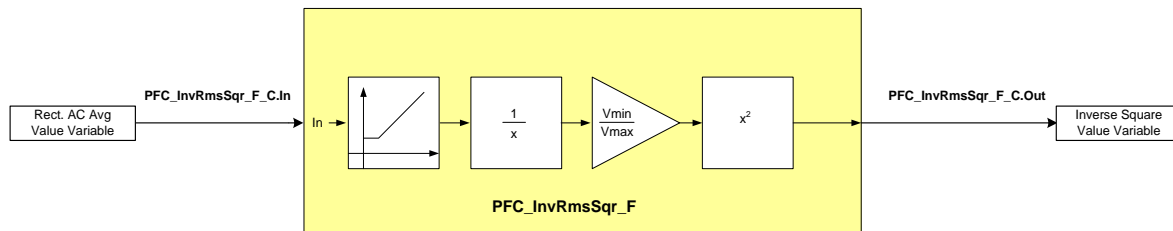
#### 5.4.4 PFC\_InvRmsSqr\_F\_C : Inverse of the RMS Squared

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



**Macro File:** PFC\_InvRmsSqr\_F\_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\_F\_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\_F\_C module.

**Step 1 Add include for DPLib header file to the source file and add the DPLib include and C\_macros folder to the include path of the project.**

**Step 2 Declare structure** in the source file

```
/* DPLib variables */
PFC_InvRmsSqr_CLA_C pfc_InvRmsSqr;
```

In case of CLA specify memory locations, by default CLA data section will be used.

```
/* DPLib variables memory locations */
#pragma DATA_SECTION(pfc_InvRmsSqr, "Cla1ToCpuMsgRAM")
```

**Step 3 For CLA add shared declaration** to {ProjectName}-CLA\_Shared.h if variables are 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 Macro initialization** For CLA edit the task8 with the initialization routine shown below. This task found in the {ProjectName}-CLA\_Tasks.cla. For C28x call the init function in the main file before enabling interrupts

```
...
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 Call** the macro within a CLA-Task of C28x ISR.

```
...
pfc_InvRmsSqr.In = In;
PFC_BL_InvRmsSqr_CLA_C(pfc_InvRmsSqr);
Out = pfc_InvRmsSqr.Out;
...
```

**Object Definition:**

```

/**
 * PFC_InvRmsSqr_C structure
 */
typedef struct {
    float32 In;
    float32 Out;
    float32 Vmin;
    float32 VminOverVmax;
} PFC_InvRmsSqr_F_C;

```

**Special Constants and Data types****PFC\_InvRmsSqr\_F\_C**

Structure defined to store module values. To create multiple instances of the module simply declare variables of type PFC\_InvRmsSqr\_F\_C.

**Module Structure Definition:**

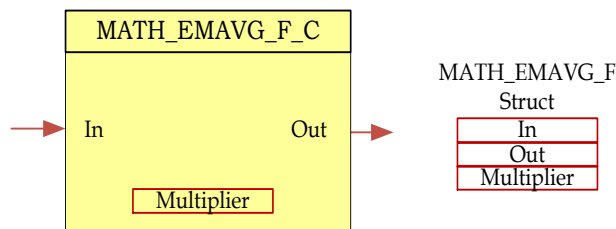
Parameter Name	Types	Description	Acceptable Range
In	Input	Input voltage.	float32 [0, 1)
Out	Input	PFC_InvRmsSqr_F_C block output.	float32 [0, 1)
Vmin	Input	Ratio of minimum designed AC line over maximum designed feedback voltage.	float32 [0, 1)
VminOverVmax	Input	Ratio of minimum over maximum voltage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



## 5.5 MATH Blocks

### 5.5.1 MATH\_EMAVG\_F\_C : Exponential Moving Average

**Description:** This software module performs exponential moving average.



**Macro File:** MATH\_EMAVG\_F\_C.h

**Technical:** This software module performs exponential moving average over float data provided via the MATH\_EMAVG\_F\_C.In structure parameter. The result, also in float format, is stored in the structure and can be accessed by the MATH\_EMAVG\_F\_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'.

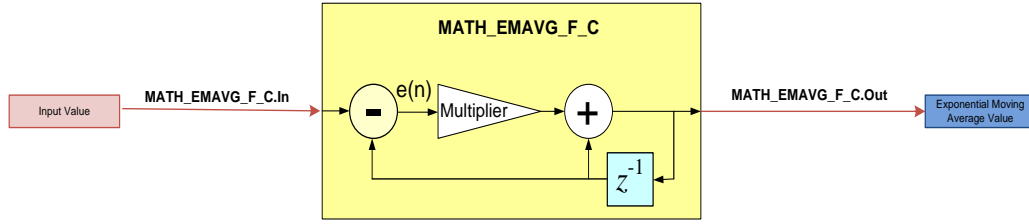
$Multiplier$  is the weighting factor used in exponential moving average

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  $\leq 1$ , 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  $(100\text{kHz}/120) = 833$  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:

$$\text{Multiplier} = 1 / \text{SAMPLE\_No} = 1 / 3332 = 0.0003$$

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:

$$\frac{\text{Out}(s)}{\text{Input}(s)} = \frac{1 + sT_s}{1 + s \frac{T_s}{\text{Mul}}}$$

Comparing with the analog domain low pass filter, the following equation can be written:

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

**Usage:** This section explains how to use the MATH\_EMAVG\_F\_C module.

**Step 1 Add include for DPLib header file to the source file and add the DPLib include and C\_macros folder to the include path of the project.**

**Step 2 Declare structure** in the source file

```
/* DPLib variables */  
MATH_EMAVG_F_C math1;
```

In case of CLA specify memory locations, by default CLA data section will be used.

```
/* DPLib variables memory locations */  
#pragma DATA_SECTION(math1, "Cla1ToCpuMsgRAM")
```

**Step 3 For CLA add shared declaration** to {ProjectName}-CLA\_Shared.h if variables are shared between CPU and CLA.

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

**Step 4 Macro initialization** For CLA edit the task8 with the initialization routine shown below. This task found in the {ProjectName}-CLA\_Tasks.cla. For C28x call the init function in the main file before enabling interrupts

```
...  
MATH_EMAVG_F_C_INIT(math1);  
math1.Multiplier = (float32) 0.3913;  
...
```

**Step 5 Call** the macro within a CLA-Task of C28x ISR.

```
...  
math1.In = value;  
MATH_EMAVG_F_C(math1);  
Out = math1.Out;  
...
```

**Object Definition:**

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

**Special Constants and Data types****MATH\_EMAVG\_F\_C**

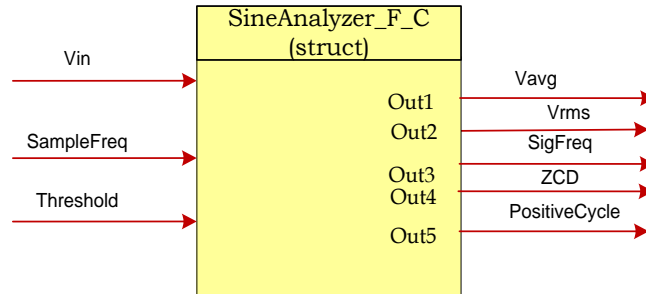
Structure defined to store module values. To create multiple instances of the module simply declare variables of type MATH\_EMAVG\_F\_C.

**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)

## 5.5.2 SineAnalyzer\_F\_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_F_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.

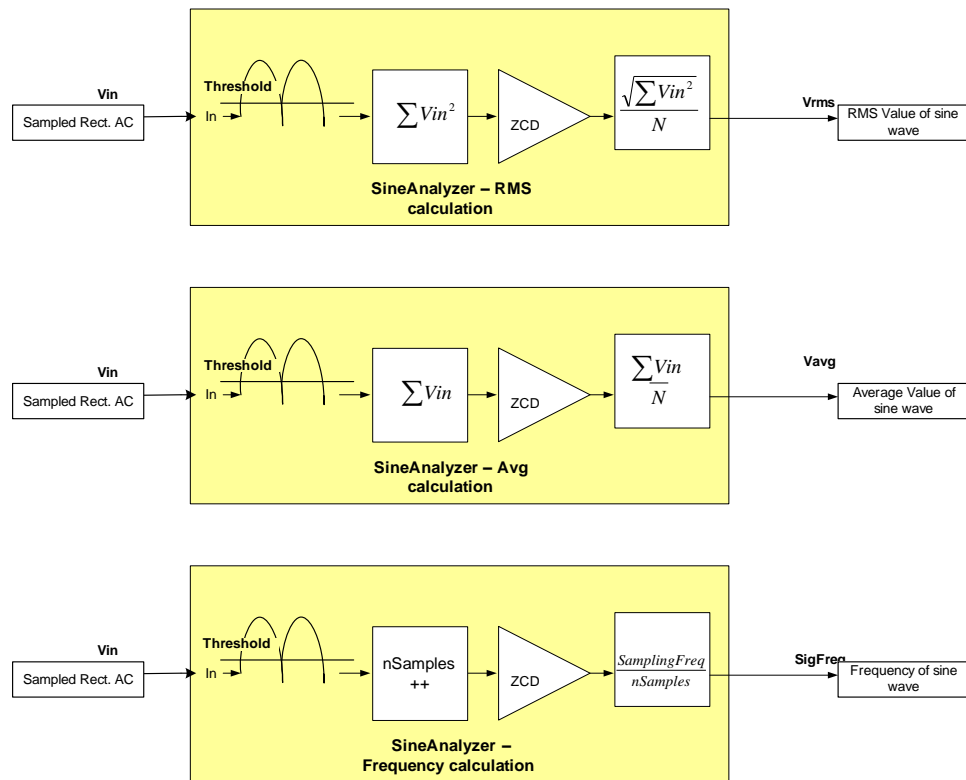
This module expects the following inputs:

- 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 include for DPLib header file to the source file and add the DPLib include and C\_macros folder to the include path of the project.**

**Step 2 Declare structure** in the source file

```
/* DPLib variables */
SineAnalyzer_F_C sineAnalyzer;
```

In case of CLA specify memory locations, by default CLA data section will be used.

```
/* DPLib variables memory locations */
#pragma DATA_SECTION(sineAnalyzer, "Cla1ToCpuMsgRAM")
```

**Step 3 For CLA add shared declaration** to {ProjectName}-CLA\_Shared.h if variables are shared between CPU and CLA.

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

**Step 4 Macro initialization** For CLA edit the task8 with the initialization routine shown below. This task found in the {ProjectName}-CLA\_Tasks.cla. For C28x call the init function in the main file before enabling interrupts

```
...  
SineAnalyzer_F_C_INIT(sineAnalyzer);  
sineAnalyzer.Threshold = (float32) 1.39;  
...
```

**Step 5 Call** the macro within a CLA-Task of C28x ISR.

```
...  
sineAnalyzer.Vin = value;  
SineAnalyzer_F_C(sineAnalyzer);  
Out = sineAnalyzer.Vrms;  
...
```

**Object Definition:**

```

/**
 * SineAnalyzer_F_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_F_C;

```

**Special Constants and Data types****SineAnalyzer\_F\_C**

Structure defined to store module values. To create multiple instances of the module simply declare variables of type SineAnalyzer\_F\_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

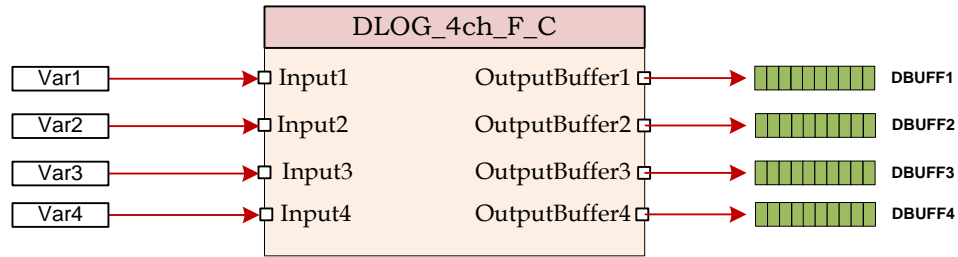


SigFreq	Output	Frequency of the sine wave input (Vin) updated at cross over point	float32
ZCD	Output	When "1" - indicates that Cross over happened and stays high till the next call of the macro.	float32
Vacc_avg	Internal	Used for accumulation of samples for Average value calculation.	float32
Vacc_rms	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

### 5.6.1 DLOG\_4ch\_F\_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\_F\_C.h

**Technical:** This software module performs data logging over data stored, in float format, within Input1-4 variables of a DLOG\_4ch\_F\_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  $100\text{KHz}/60\text{Hz} \approx 1666$ . 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;

$$\text{PreScalar} = \frac{100\text{KHz}}{(60\text{Hz} * \text{BufferSize})} = 8.33$$

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 module.

**Step 1 Add include for DPLib header file to the source file and add the DPLib include and C\_macros folder to the include path of the project.**

**Step 2 Declare structure** in the source file

```
/* DPLib variables */
// Buffers for storing data
float32 buffer1[200];
float32 buffer2[200];
float32 buffer3[200];
float32 buffer4[200];
// dlog instance
DLOG_4ch_F_C dlog4ch;
```

In case of CLA specify memory locations, by default CLA data section will be used.

```
/* DPLib variables memory locations */
#pragma DATA_SECTION(dlog4ch, "Cla1ToCpuMsgRAM")
#pragma DATA_SECTION(buffer1, "Cla1DataRam0")
#pragma DATA_SECTION(buffer2, "Cla1DataRam0")
#pragma DATA_SECTION(buffer3, "Cla1DataRam0")
#pragma DATA_SECTION(buffer4, "Cla1DataRam0")
```

**Step 3 For CLA add shared declaration** to {ProjectName}-CLA\_Shared.h if variables are shared between CPU and CLA.

```
/* DPLib variables */
// Declare net variables shared between CPU and CLA here
extern volatile DLOG_4ch_F_C dlog4ch;
```

**Step 4 Macro initialization** For CLA edit the task8 with the initialization routine shown below. This task found in the {ProjectName}-CLA\_Tasks.cla. For C28x call the init function in the main file before enabling interrupts

```
...
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 Call** the macro within a CLA-Task or C28x ISR.

```
...
dlog4ch.Input1 = value1;
dlog4ch.Input2 = value2;
dlog4ch.Input3 = value3;
dlog4ch.Input4 = value4;
DLOG_4ch_F_C(dlog4ch);

...
```

**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_F_C;
```

### Special Constants and Data types

#### DLOG\_4ch\_F\_C

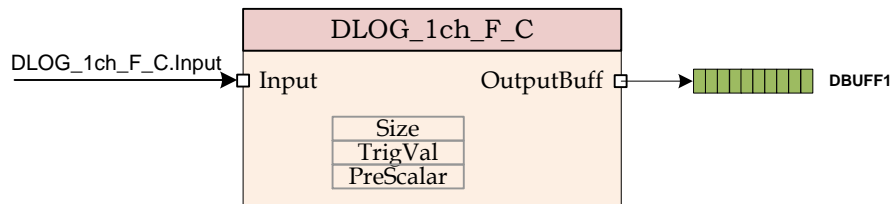
Structure defined to store module values. To create multiple instances of the module simply declare variables of type DLOG\_4ch\_F\_C.

#### 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

## 5.6.2 DLOG\_1ch\_F\_C : Single Channel Data Logger

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



**Macro File:** DLOG\_1ch\_F\_C.h

**Technical:** This software module performs data logging over data stored, in float format, in the Input variable within a DLOG\_1ch\_F\_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 module.

**Step 1 Add include for DPLib header file to the source file and add the DPLib include and C\_macros folder to the include path of the project.**

**Step 2 Declare structure** in the source file

```
/* DPLib variables */
// Buffers for storing data
float32 buffer1[200];
// dlog instance
DLOG_1ch_CLA_C dlog1ch;
```

In case of CLA specify memory locations, by default CLA data section will be used.

```
/* DPLib variables memory locations */
#pragma DATA_SECTION(dlog4ch, "Cla1ToCpuMsgRAM")
#pragma DATA_SECTION(buffer1, "Cla1DataRam0")
```

**Step 3 For CLA add shared declaration** to {ProjectName}-CLA\_Shared.h if variables are 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 Macro initialization** For CLA edit the task8 with the initialization routine shown below. This task found in the {ProjectName}-CLA\_Tasks.cla. For C28x call the init function in the main file before enabling interrupts

```
...
DLOG_1ch_F_C_INIT(dlog1ch);
dlog1ch.OutputBuff = buffer1;
dlog1ch.PreScalar = 8;
dlog1ch.TrigVal = 0.54;
...
```

**Step 5 Call** the macro within a CLA-Task or C28x ISR.

```
...
Dlog1ch.Input = value1;
DLOG_1ch_F_C(dlog1ch);
...
```

**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_F_C;
```

### Special Constants and Data types

#### DLOG\_1ch\_F\_C

Structure defined to store module values. To create multiple instances of the module simply declare variables of type DLOG\_1ch\_F\_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 5. Revision History

Version	Date	Notes
V2.0	May 28, 2015	First release of DPLib for F28x7x devices