C28x Digital Power Library

v2.0

July 2010

Module User's Guide

C28x Foundation Software

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 functions.

C28x: Refers to devices with the C28x CPU core.

IQmath: Fixed-point mathematical functions in C.

Q-math: Fixed point numeric format defining the binary resolution in bits.

Contents

Chapter 1. Introduction

1.1. Introduction

Texas Instruments Digital Power (DP) library is designed to enable flexible and efficient coding of digital power supply applications using the C28x processor. An important consideration of these 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. In many cases this makes the use of C code impossible, or at least inadvisable, for the ISR.

A further requirement in code development and test is for the software structure to be flexible and adaptable. This enables the designer to experiment with various control loop layouts, and to monitor software variables at various points in the code to confirm correct operation of the system. For this reason, the DP library is constructed in a modular form, with macro functions encapsulated in re-usable code blocks which can be connected together to build any desired software structure.

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

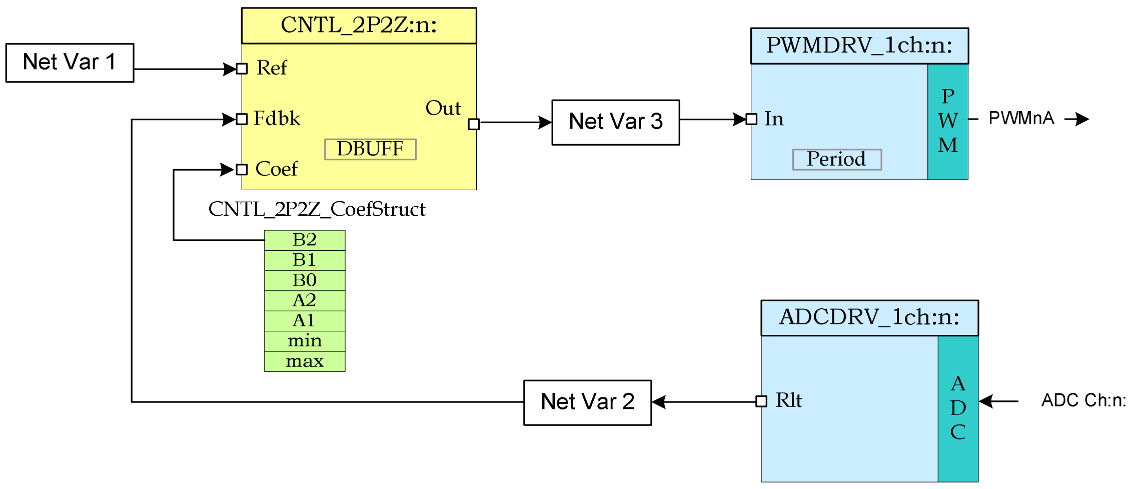


Figure 1 Close Loop System using DPLib

In this example, three library macro-blocks are connected: an ADC driver, a second order digital controller, and a PWM driver. The labels “Net Var 1”, Net Var 2” and “Net Var 3” correspond to software variables which form the connection points, or “nodes”, in the diagram. The input and output terminals of each block are connected to these nodes by a simple method of C pointer assignment in software. In this way, designs may be rapidly re-configured to experiment with different software configurations.

Library blocks have been color coded: “turquoise” blocks represent those which interface to physical device hardware, such as an A/D converter, while “yellow” blocks indicate macros 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 well give undesirable results!).

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

Once the blocks have been initialized and connected, they can be executed by calling the appropriate code from an assembly ISR. Macro blocks execute sequentially, each block performing a precisely defined computation and delivering its' result to the appropriate net list variables, before the next block begins execution.

Chapter 2. Installing the DP Library

2.1. DP Library Package Contents

The TI Digital Power library consists of the following components:

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

2.2. How to Install the Digital Power Library

The DP library is distributed through the controlSUITE installer. The user must select the Digital Power Library Checkbox to install the library in the controlSUITE directory. By default, the installation places the library components in the following directory structure:

<base> install directory is C:\ti\controlSUITE\libs\app_libs\digital_power\<device>

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

<base>\asm	Contains assembly macros
<base>\C	C initialization files
<base>\doc	Contains this file
<base>\include	Contains the library header file for the "`DPlib.h`"
<base>\template	Contains template files that can be modified by user

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

controlSUITE\development_kits\BlankTemplatesDPlib\BlankTemplate-<device>

These template projects can be quickly modified to start a new project using the DPlib. Refer to the BlankTemplatesDPlib\ directory on how to modify the project.

2.3. Naming Convention

Each macro of the digital power library has an assembly file that contains the initialization and the run time code for the block. In addition to the assembly block peripheral interface blocks use a peripheral configuration function as well.

An example of the naming convention used is shown below:

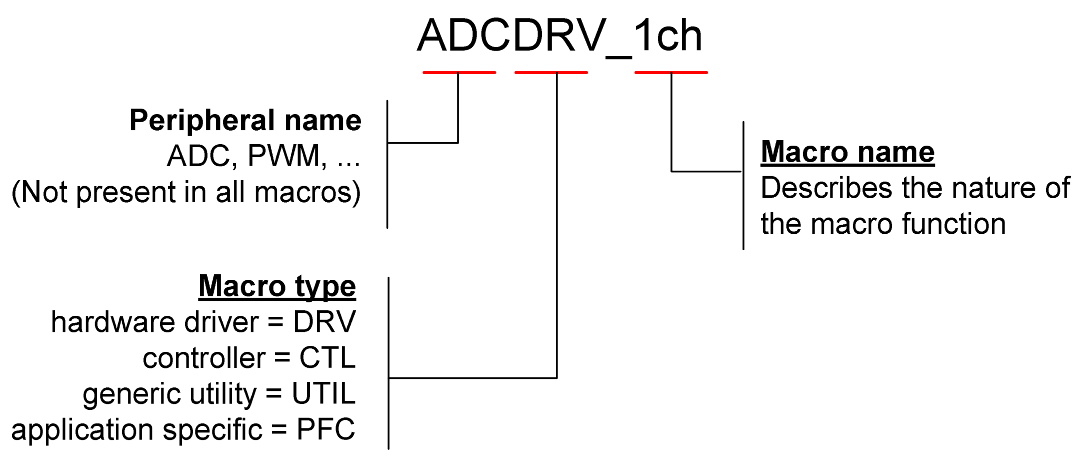


Figure 2 – Function Naming Convention

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

Include file:	`ADCDRV_1ch.asm`	
Init function:	`ADCDRV_1ch_INIT`	`n`
Execution Macro:	`ADCDRV_1ch`	`n`

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

Note: In the case of some Peripheral Drivers (e.g. ADC Drivers and PWM drivers) the instance number also implies the Peripheral Number the DP Library macro would drive or use on the device. For example

`ADCDRV_1ch 0`	normalizes AdcResult0 of the ADC Peripheral &
`PWMDRV_1ch 3`	drives the EPWM3 peripheral present on the device.

Chapter 3. Using the Digital Power Library

3.1. Library Description and Overview

Typical user software will consist of a main framework file written in C and a single Interrupt Service Routine (ISR) written in assembly. The C framework contains code to configure the device hardware and initialize the library macros. The ISR consists of a list of optimized macro modules which execute sequentially each time a hardware trigger event occurs.

This structure of setting up an interrupt based program is common in embedded real-time systems which do not use a scheduler. For examples of device initialization code, refer to the peripheral header file examples for the C28x device.

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

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

2 ***Configuration***. C pointers of the macro block terminals are assigned to net nodes to form the desired control structure. Net nodes are 32-bit integer variables declared in the C framework. Note names of these net nodes are no dependent on the macro block.

3 ***Execution***. Macro block code is executed in the assembly ISR ("`DPL_ISR`"). This function is defined in the "{`ProjectName}-DPL-ISR.asm`" .

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

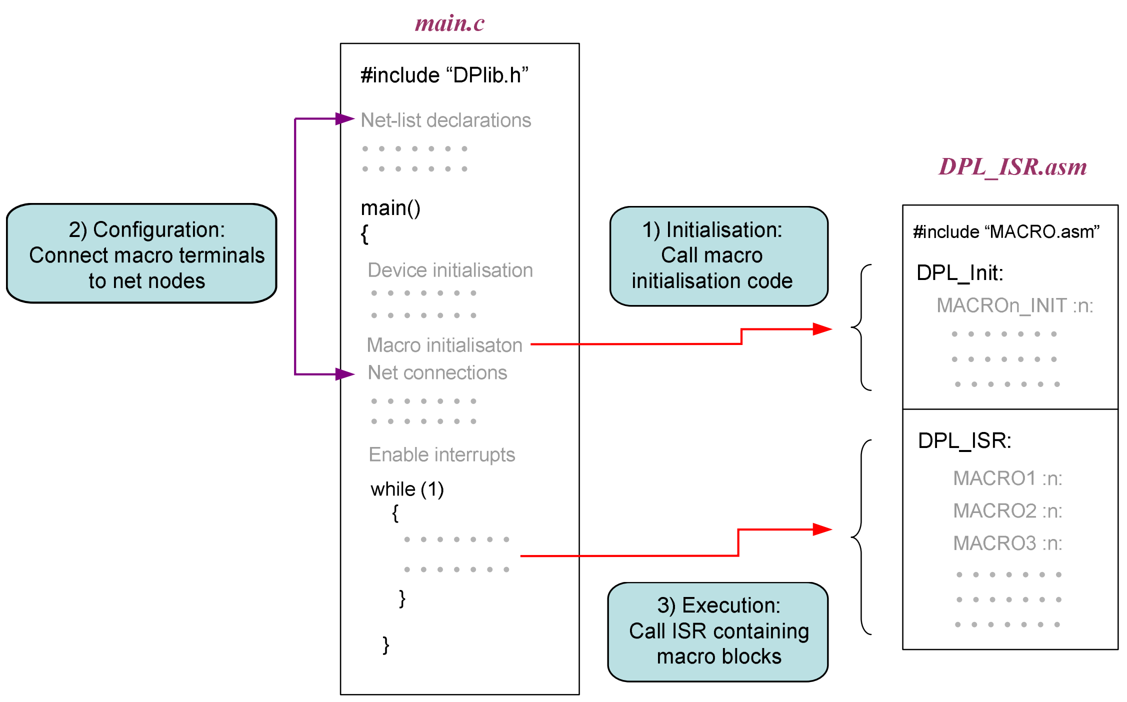


Figure 3 : Relation between Main.c & ISR.asm file

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

The assembly code in “`DPL_Init`” is C callable, and its’ prototype is in the library header file “`DPlib.h`” described above. To initialize the macros, edit the DPL_Init section of the file “`{ProjectName}-DPL-ISR.asm`” to add initialization calls for each macro required in the application. The order of the calls is not important, providing one call is made for each macro-block required in the application. The respective macro assembly file must be included at the starting of the “`{ProjectName}-DPL-ISR.asm`” file.

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

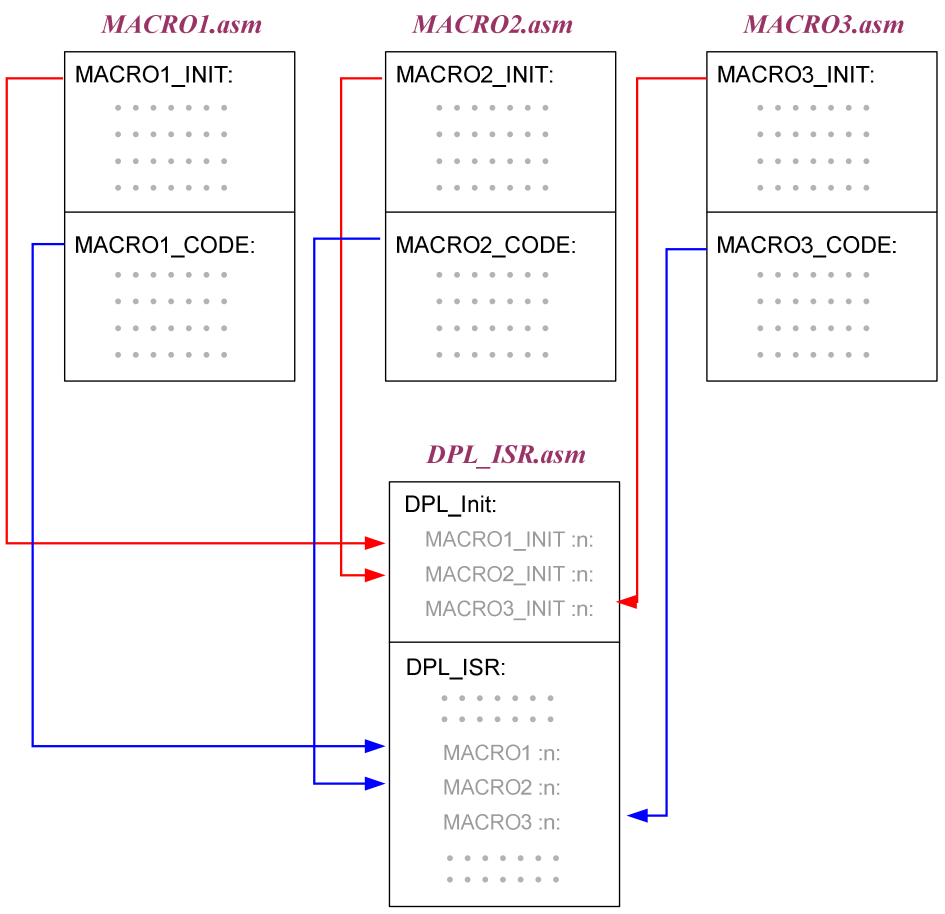


Figure 4 DP library assembly ISR and macro file

The ISR contains context save and context restore blocks to protect any registers used by the assembly modules. By default, the template performs a complete context save of all the main CPU registers. PUSH/POP instructions can be commented to save cycles if specific registers are known to be unused by any of the macros in the ISR. A list of registers used by each module is shown below.

3.2. Steps to use the DP library

The first task before using the DP library should be to sketch out in diagram form the modules and block topology required. The aim should be to produce a diagram similar to that in Figure 1.This will indicate which macro-blocks are required and how they interface with one another. Once this is known, the code can be configured as follows:

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

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

This file is located in the at,
`controlSUITE\libs\app_libs\digital_power\{device_name_VerNo}\include`

This path needs to be added to the include path in the build options for the project.

Step 2 ***Declare terminal pointers in C***. The "`{ProjectName}-Main.c`" file needs to be edited to add extern declarations to all the macro terminal pointers which will be needed in the application under the "`DPLIB Net Terminals`" section inside this file. In the example below, three pointers to an instance of the 2P2Z control block are referenced. Please note the use of volatile keyword for the net pointers, as they point to net variables which are volatile as the ISR computes these values.

```
// ---------------------------- DPLIB Net Pointers ---------------------
// Declare net pointers that are used to connect the DP Lib Macros  here
// CNTL_2P2Z #instance 1
extern volatile long    *CNTL_2P2Z_Ref1;
extern volatile long    *CNTL_2P2Z_Fdbk1;
extern volatile long    *CNTL_2P2Z_Out1;
extern volatile long    *CNTL_2P2Z_Coef1;
```

Step 3 ***Declare signal net nodes/net variables in C***. Edit the "`{ProjectName}-Main.c`" C file to define the net variables which will be needed in the application under the "`DPLIB Variables`" section . In the example below, three arbitrarily named variables are declared as global variables in C.

```
// ---------------------------- DPLIB Variables ------------------------
// Declare the net variables being used by the DP Lib Macro here
volatile long     Net1, Net2, Net3;
```

Step 4 ***Call the Peripheral configuration function***. Call the peripheral configuration functions that are needed to configure the peripherals being used by the library macros being used in the system.

```
Note as CNTL_2P2Z is a software block this step is not needed.
```

Step 5 ***Call the initialisation function from C***. Call the initialization function from the C framework using the syntax below.

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

Step 6 ***Assign macro block terminals to net nodes.*** This step connects macro blocks together via net nodes to form the desired control structure. The process is one of pointer assignment using the net node variables and terminal pointers declared in the previous two steps.

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

```
      // feedback node connections
      ADCDRV_1ch_Rlt0 = &Net2;
      CNTL_2P2Z_Fdbk1 = &Net2;
```

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

Step 7 ***Add the ISR file***. A single assembly file containing the ISR code and calls to the macro initialisation functions must exist in the project. The relationship between these elements is described in Chapter 3.1. A blank template "`ProjectName-DPL-ISR.asm`" is included with the DP library for this purpose in the template directory. To use this file, rename the file as "`{ProjectName}-DPL-ISR.asm`" and add it to the project.

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

```
            .include "CNTL_2P2Z.asm"
```

Step 9 ***Initialize required macro blocks***. Edit the function "`DPL_Init`" in the above ISR file to add calls to the initialization code in each macro file. Each call is invoked with a number to identify the unique instance of that macro block. For example, to create an instance of the 2P2Z control block with the identifier "`1`":

```
      CNTL_2P2Z_INIT     1
```

Step 10 ***Edit the assembly ISR to execute the macros in the required order***. Edit the function "`DPL_Run`" to add calls to the run time routine of each macro and instance being used in the system. In the example above the first instance of a 2P2Z control macro would be executed by:

```
      CNTL_2P2Z    1
```

Step 11 ***Add the DP library sections to the linker command file***. The linker places DP library code is a named sections as specified in the linker command file "`{DeviceName-RAM/FLASH-ProjectName}.CMD`". A sample CMD file is provided with the sections specified for the entire DPS library in the template folder. The files only provides a sample memory allocation and can be edited by the user to suit their application.

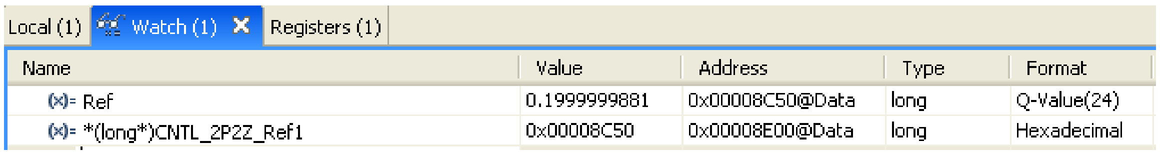
This DPLib Macros need to be placed in the data RAM. The sample linker file specifies where each memory section from each DP library module would be placed in internal memory. An example of section placement for the CNTL_2P2Z module is shown below.

```
  /* CNTL_2P2Z section */
 CNTL_2P2Z_Section       : > dataRAM                   PAGE = 1
 CNTL_2P2Z_InternalData  : > dataRAM                   PAGE = 1
 CNTL_2P2Z_Coef          : > dataRAM                   PAGE = 1
```

Where dataRAM is a location in the RAM on the device which is specified in the sample CMD file.

3.3. Viewing DP CLA library variables in watch window

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



Name	Value	Address	Type	Format
Ref	0.1999999881	0x00008C50@Data	long	Q-Value(24)
(long)CNTL_2P2Z_Ref1	0x00008C50	0x00008E00@Data	long	Hexadecimal

Chapter 4. Module Summary

4.1. DP Library Function Summary

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

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

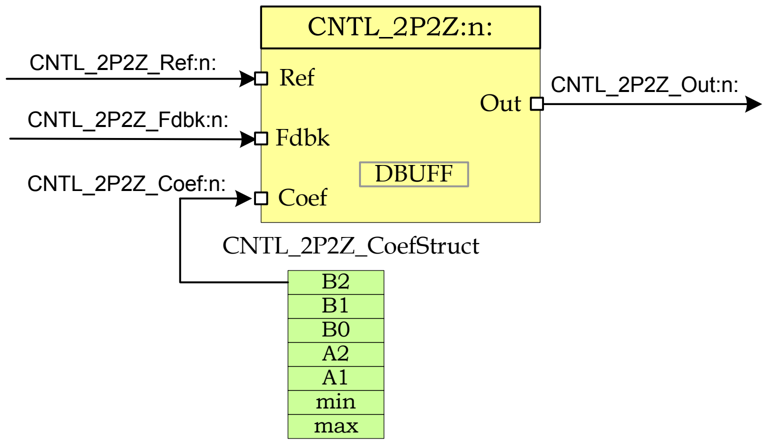
Module Name	**Module Type**	**Description**	**HW Config File**	**Cycles**	**Init Code Size (W)**	**Run Code Size (W)**	**Data Size (W)**	**Multiple Instance Support**
CNTL_2P2Z	CNTL	Second Order Control Law	NA	34	17	46	18	Yes
CNTL_3P3Z	CNTL	Third Order Control Law	NA	42	17	56	22	Yes
ADCDRV_1ch	HW	Single Channel ADC Driver	Yes	5	5	8	2	Yes
ADCDRV_4ch	HW	Four Channel ADC Driver	Yes	14	8	20	8	No
PWMDRV_1ch	HW	Single Channel PWM Driver	Yes	10	12	12	4	Yes
PWMDRV_1chHiRes	HW	Single Channel PWM Driver with Hi Res capability	Yes	10	12	12	4	Yes
PWMDRV_PFC2PhiL	HW	PWM driver for Two Phase Interleaved PFC stage	Yes	17	13	18	6	Yes
PWMDRV_PSFB	HW	PWM driver for Phase Shifted Full Bridge Power stage	Yes	21	13	21	6	Yes
PWMDRV_ComplPairDB	HW	PWM driver for complimentary pair PWMs	Yes	10	15	12	6	Yes
PWMDRV_DualUpDwnCnt	HW	PWM driver with independent duty control on ch A and ch B	Yes	14	13	16	6	Yes
PFC_ICMD	APPL	Power Factor Correction Current Command Block	NA	17	9	19	10	Yes
PFC_INVSQR	APPL	Power Factor Correction Inverse Square Block	NA	71	15	39	12	Yes
MATH_EMAVG	MATH	Exponential moving average	NA	16	6	16	6	Yes
DLOG_4ch	UTIL	4 channel Data Logger Module	NA	33 (Avg)	14	56	24	No
DLOG_1ch	UTIL	1 channel Data Logger Module	NA	20 (Avg)	17	41	12	Yes

Chapter 5. C28x Module Descriptions

5.1. Controllers

CNTL_2P2Z — *Two Pole Two Zero Controller*

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



Macro File: `CNTL_2P2Z.asm`

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

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

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

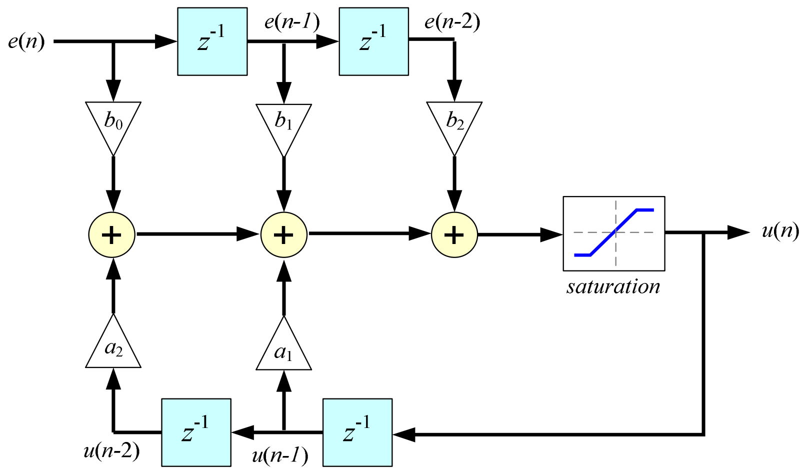
This may be expressed in difference equation form as:

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

Where…

$u(n)$	= present controller output (after saturation)
$u(n\text{-}1)$	= controller output on previous cycle
$u(n\text{-}2)$	= controller output two cycles previously
$e(n)$	= present controller input
$e(n\text{-}1)$	= controller input on previous cycle
$e(n\text{-}2)$	= controller input two cycles previously

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



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

CNTL_2P2Z_DBUFF

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

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

CNTL_2P2Z_CoefStruct

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

Where sat_{max} and sat_{min} are the upper and lower control effort bounds respectively. Note that to preserve maximum resolution the coefficients are saved in Q26 format and the saturation limits are stored in Q24 format to match the output format.

Controller coefficients must be initialized before the controller is used. A structure `CNTL_2P2Z_CoefStruct` is used to ensure that the coefficients are stored exactly as shown in the table as the `CNTL_2P2Z` accesses them relative to a base address pointer. The structure is defined in the library header file `DPlib.h` to allow easy access to the elements from C.

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

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

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

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

```
// ---------------------------- DPLIB Net Pointers ----------------------
// declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
// CONTROL_2P2Z - instance #1
extern volatile long    *CNTL_2P2Z_Ref1;
extern volatile long    *CNTL_2P2Z_Out1;
extern volatile long    *CNTL_2P2Z_Fdbk1;
extern volatile long    *CNTL_2P2Z_Coef1;
```

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

Note signal net mode names change from system to system, no dependency exist between these names and module.

```
// ---------------------------- DPLIB Variables ------------------------
// declare the net nodes/variables being used by the DP Lib Macro here
long Ref , Fdbk , Out;

#pragma DATA_SECTION(CNTL_2P2Z_CoefStruct1, "CNTL_2P2Z_Coef");
struct CNTL_2P2Z_CoefStruct CNTL_2P2Z_CoefStruct1;
```

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

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

// Connect the CNTL_2P2Z block to the variables
CNTL_2P2Z_Fdbk1 = &Fdbk;
CNTL_2P2Z_Out1  = &Out;
CNTL_2P2Z_Ref1  = &Ref;
CNTL_2P2Z_Coef1 = &CNTL_2P2Z_CoefStruct1.b2;
```

```
// Initialize the Controller Coefficients
CNTL_2P2Z_CoefStruct1.b2 = _IQ26(0.05);
CNTL_2P2Z_CoefStruct1.b1 = _IQ26(-0.20);
CNTL_2P2Z_CoefStruct1.b0 = _IQ26(0.20);
CNTL_2P2Z_CoefStruct1.a2 = _IQ26(0.0);
CNTL_2P2Z_CoefStruct1.a1 = _IQ26(1.0);
CNTL_2P2Z_CoefStruct1.max =_IQ24(0.7);
CNTL_2P2Z_CoefStruct1.min =_IQ24(0.0);

//Initialize the net Variables/nodes
Ref=_IQ24(0.0);
Fdbk=_IQ24(0.0)
Out=_IQ24(0.0);
```

Step 5 **Add** the ISR assembly file "`{ProjectName}-DPL-ISR.asm`" to the project

Step 6 **Include** the assembly macro file in the `{ProjectName}-DPL-ISR.asm`

```
;Include files for the Power Library Macro's being used by the system
.include "CNTL_2P2Z.asm"
```

Step 7 **Instantiate the INIT** macro in assembly (this is one-time pass through code) inside the C-callable DPL_Init() function which is defined in`{ProjectName}-DPL-ISR.asm`

```
;Macro Specific Initialization Functions
CNTL_2P2Z_INIT 1   ; CNTL_2P2Z Initialization
```

Step 8 **Call the run time macro** in assembly inside the C-callable function `DPL_ISR()` which is the looped or ISR code. The function is defined in`{ProjectName}-DPL-ISR.asm`

```
;"Call" the Run macro
CNTL_2P2Z 1         ; Run the CNTL_2P2Z Macro
```

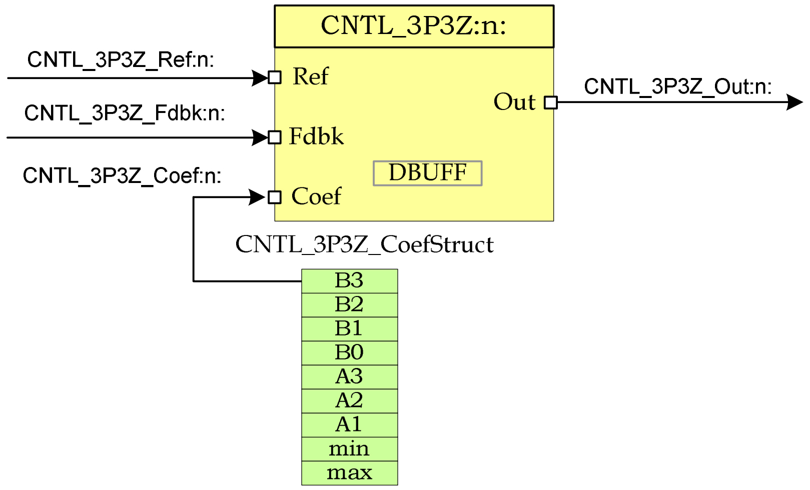
Step 9 **Include the memory sections** in `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD`. Note, for the CNTL_2P2Z module the net pointers and the internal data do not assume anything about allocation on a single data page.

```
/*CNTL_2P2Z sections*/
CNTL_2P2Z_Section        : > dataRAM PAGE = 1
CNTL_2P2Z_InternalData   : > dataRAM PAGE = 1
CNTL_2P2Z_Coef           : > dataRAM PAGE = 1
```

Module Net Definition:

Net Name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
CNTL_2P2Z_Ref:n:	Input Pointer	Pointer to 32 bit fixed point input data location storing the Reference value for the controller.	Q24: [0, 1)
CNTL_2P2Z_Fdbk:n:	Input Pointer	Pointer to 32 bit fixed point input data location storing the Feedback value for the controller.	Q24: [0, 1)
CNTL_2P2Z_Coef:n:	Input Pointer	Pointer to the location where coefficient structure is stored.	See Module Description
CNTL_2P2Z_Out:n:	Output Pointer	Pointer to 32 bit fixed point output location where the reference for the current loop is stored	Q24:[0,1)
CNTL_2P2Z_DBUFF:n:	Internal Data	Data Variable storing the scaling factor	See Module Description

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



Macro File: `CNTL_3P3Z.asm`

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 three delay lines: one for input data and one for output data, each consisting of three elements.

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

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

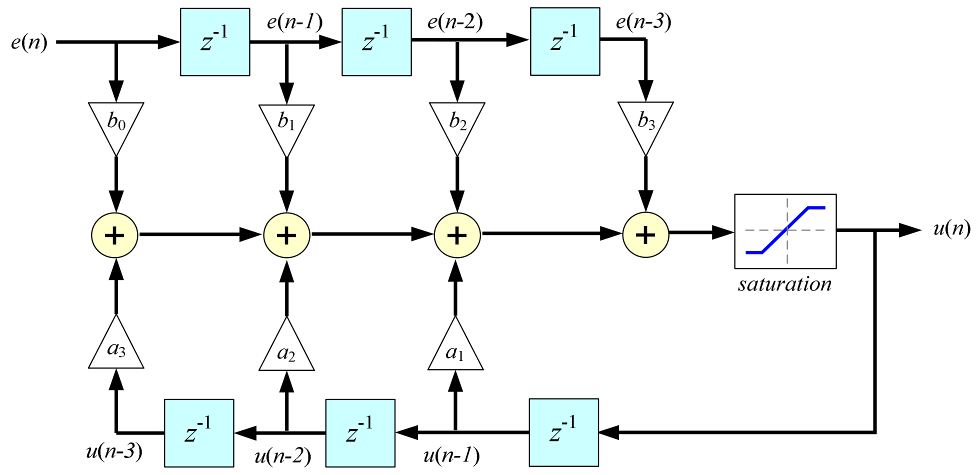
This may be expressed in difference equation form as:

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

Where…

$u(n)$	= present controller output (after saturation)
$u(n\text{-}1)$	= controller output on previous cycle
$u(n\text{-}2)$	= controller output two cycles previously
$u(n\text{-}3)$	= controller output three cycles previously
$e(n)$	= present controller input
$e(n\text{-}1)$	= controller input on previous cycle
$e(n\text{-}2)$	= controller input two cycles previously
$e(n\text{-}3)$	= controller input three cycles previously

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



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

CNTL_3P3Z_DBUFF

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

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

CNTL_3P3Z_CoefStruct

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

Where sat_{max} and sat_{min} are the upper and lower control effort bounds respectively. Note that to preserve maximum resolution the coefficients are saved in Q26 format and the saturation limits are stored in Q24 format to match the output format.

Controller coefficients must be initialized before the controller is used. A structure `CNTL_3P3Z_CoefStruct` is used to ensure that the coefficients are stored exactly as shown in the table as the `CNTL_3P3Z` accesses them relative to a base address pointer. The structure is defined in the library header file `DPlib.h` to allow easy access to the elements from C.

Usage: This section explains how to use this module.

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

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

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

```
// ---------------------------- DPLIB Net Pointers ----------------------
// declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
// CONTROL_3P3Z - instance #1
extern volatile long    *CNTL_3P3Z_Ref1;
extern volatile long    *CNTL_3P3Z_Out1;
extern volatile long    *CNTL_3P3Z_Fdbk1;
extern volatile long    *CNTL_3P3Z_Coef1;
```

Step 3 Declare signal net nodes/ variables in C in the file {ProjectName}-Main.c

Note signal net mode names change from system to system, no dependency exist between these names and module.

```
// ---------------------------- DPLIB Variables -------------------------
// declare the net nodes/variables being used by the DP Lib Macro here
volatile long Ref , Fdbk , Out;

#pragma DATA_SECTION(CNTL_3P3Z_CoefStruct1, "CNTL_3P3Z_Coef");
struct CNTL_3P3Z_CoefStruct CNTL_3P3Z_CoefStruct1;
```

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

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

// Connect the CNTL_2P2Z block to the variables
CNTL_3P3Z_Fdbk1 = &Fdbk;
CNTL_3P3Z_Out1  = &Out;
CNTL_3P3Z_Ref1  = &Ref;
CNTL_3P3Z_Coef1 = &CNTL_3P3Z_CoefStruct1.b2;
```

```
// Initialize the Controller Coefficients
CNTL_3P3Z_CoefStruct1.b2  = _IQ26(0.05);
CNTL_3P3Z_CoefStruct1.b1  = _IQ26(-0.20);
CNTL_3P3Z_CoefStruct1.b0  = _IQ26(0.20);
CNTL_3P3Z_CoefStruct1.a2  = _IQ26(0.0);
CNTL_3P3Z_CoefStruct1.a1  = _IQ26(1.0);
CNTL_3P3Z_CoefStruct1.max =_IQ24(0.7);
CNTL_3P3Z_CoefStruct1.min =_IQ24(0.0);

//Initialize the net Variables/nodes
Ref=_IQ24(0.0);
Fdbk=_IQ24(0.0)
Out=_IQ24(0.0);
```

Step 5 **Add** the ISR assembly file "`{ProjectName}-DPL-ISR.asm`" to the project

Step 6 **Include** the macro's assembly file in the `{ProjectName}-DPL-ISR.asm`

```
;Include files for the Power Library Macro's being used by the system
.include "CNTL_3P3Z.asm"
```

Step 7 **Instantiate the INIT** macro in assembly (this is one-time pass through code) inside the C-callable DPL_Init() function which is defined in `{ProjectName}-DPL-ISR.asm`

```
;Macro Specific Initialization Functions
CNTL_3P3Z_INIT 1  ; CNTL_3P3Z Initialization
```

`Step 8` **Call the run time macro** in assembly inside the C-callable function `DPL_ISR()` which is the looped or ISR code. The function is defined in `{ProjectName}-DPL-ISR.asm`

```
;"Call" the Run macro
CNTL_3P3Z 1        ; Run the CNTL_3P3Z Macro
```

`Step 9` **Include the memory section** in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD.` Note, for the CNTL_3P3Z module the net pointers and the internal data do not assume anything about allocation on a single data page.

```
/*CNTL_3P3Z sections*/
CNTL_3P3Z_Section        : > dataRAM PAGE = 1
CNTL_3P3Z_InternalData   : > dataRAM PAGE = 1
CNTL_3P3Z_Coef           : > dataRAM PAGE = 1
```

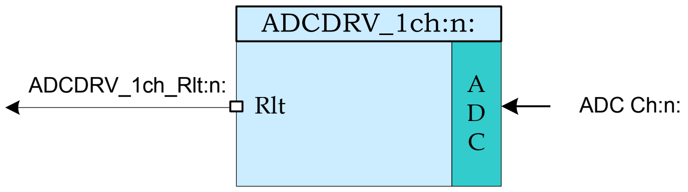
Module Net Definition:

Net Name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
CNTL_3P3Z_Ref:n:	Input Pointer	Pointer to 32 bit fixed point input data location storing the Reference value for the controller.	Q24: [0, 1)
CNTL_3P3Z_Fdbk:n:	Input Pointer	Pointer to 32 bit fixed point input data location storing the Feedback value for the controller.	Q24: [0, 1)
CNTL_3P3Z_Coef:n:	Input Pointer	Pointer to the location where coefficient structure is stored.	See Module Description
CNTL_3P3Z_Out:n:	Output Pointer	Pointer to 32 bit fixed point output location where the reference for the current loop is stored	Q24:[0,1)
CNTL_3P3Z_DBUFF:n:	Internal Data	Data Variable storing the scaling factor	See Module Description

5.2. Peripheral Drivers

ADCDRV_1ch ***ADC Driver Single Channel***

Description: This assembly macro reads a result from the internal ADC module Result Register:n: and delivers it in Q24 format to the output terminal, where :n: is the instance number. The output is normalized to 0-1.0 such that the minimum input voltage will generate nominally 0.0 at the driver output, and a maximum full scale input voltage read +1.0. The result is then stored in the memory location pointed to by the net terminal pointer.



Macro File: `ADCDRV_1ch.asm`

Peripheral Initialization File: `ADC_SOC_Cnf.c`

Description: The ADC module in the F2802x & F2803x devices includes a ratio-metric input which enables the user to determine the maximum and minimum input voltages. The ADC converts this input range with 12-bits of resolution. The ADCDRV macro reads one pre-defined result register (determined by the instance number of the macro i.e. instance 0 reads AdcResult.ADCRESULT0 and instance 5 reads AdcResult.ADCRESULT5) . The module then scales this to Q24 format and writes the result in unipolar Q24 format to the output net terminal.

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

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

where

ChSel[]	stores which ADC pin is used for conversion when a Start of Conversion(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 CLA Mode, configures ADC conversions to be started by the appropriate channel trigger. An ADC interrupt is triggered when conversion is complete and the ADC interrupt flag is automatically cleared. This mode is used for all of the CLA based projects.

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

Usage:

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

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

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

```
// ---------------------------- DPLIB Net Pointers ----------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//ADCDRV_1ch - instance #1
extern volatile long *ADCDRV_1ch_Rlt1;
```

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

Note signal net node names change from system to system, no dependency exist between these names and module.

```
// ---------------------------- DPLIB Variables -------------------------
// Declare the net variables being used by the DP Lib Macro here
volatile long Out;
```

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

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


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

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

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

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

```
//----------Connect the macros to build a system-------------------
// Digital Power (DP) library initialisation
DPL_Init();
// ADCDRV_1ch block connections
ADCDRV_1ch_Rlt1=&Out;
// Initialize the net variables
Out=_IQ24(0.0);
```

Step 6 **Add** the ISR assembly file "`{ProjectName}-DPL-ISR.asm`" to the project.

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

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

.include "ADCDRV_1ch.asm"
```

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

```
;Macro Specific Initialization Functions
ADCDRV_1ch_INIT 1 ; ADCDRV_1ch Initialization
```

Step 9 **Call the run time macro** in assembly inside the C-callable function DPL_ISR() which is the looped or ISR code. The function is defined in `{ProjectName}-DPL-ISR.asm`

```
;"Call" the Run macro
ADCDRV_1ch 1      ; Run ADCDRV_1ch
```

`Step 10` **Include the memory sections** in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD.` Note that the net pointers and the internal data are forced to be placed in a single data page by use of the `.usect` directive in the source file of the module.

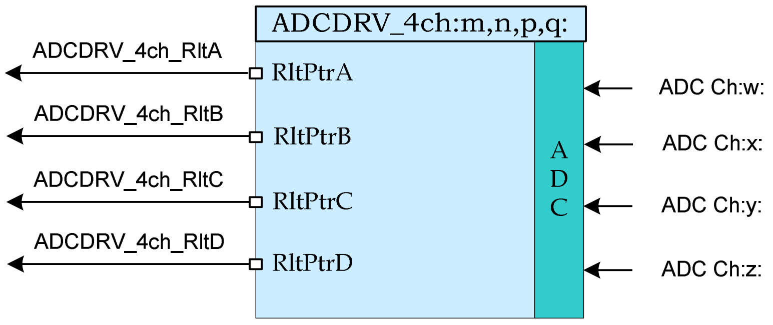
```
/*ADCDRV_1ch sections*/
ADCDRV_1ch_Section      : > dataRAM       PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
ADCDRV_1ch_Rlt:n:	Output Pointer	Pointer to 32 bit fixed point data location storing the result of the module.	Q24: [0, 1)

ADCDRV_4ch — *ADC Driver Single Channel*

Description: This assembly macro reads four results from the internal ADC module result registers m,n,p,q and delivers them in Q24 format to the output terminals. The output is normalized to 0-1.0 such that the minimum input voltage will generate nominally 0.0 at the driver output, and a maximum full scale input voltage read +1.0. The result are then stored in the memory location pointed to by the net terminal pointers.



Macro File: `ADCDRV_4ch.asm`

Peripheral Initialization File: `ADC_SOC_Cnf.c`

Description: The ADC module in the F2802x & F2803x devices includes a ratio-metric input which enables the user to determine the maximum and minimum input voltages. The ADC converts this input range with 12-bits of resolution. The ADCDRV macro reads the result register (determined by the numbers that are parsed to the run time macro m,n,p and q. The module then scales these to Q24 format and writes the result in unipolar Q24 format to the output net terminal.

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

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

where

ChSel[]	Array that stores which ADC pin is used for conversion when a start of conversion is received for the respective channel
TrigSel[]	stores what trigger Input starts the conversion of the respective channel
ACQPS[]	stores the acquisition window size used for the respective channel
IntChSel	is the channel number that triggers interrupt ADCINT 1. If the ADC interrupt is not being used enter a value of 0x10.

Mode determines what mode the ADC is configured in

Mode =0 Start/Stop mode, configures ADC conversions to be started by the appropriate channel trigger, an ADC interrupt is raised whenever conversion is complete for the IntChSel channel. The ADC interrupt flag needs to be cleared for the interrupt to be retriggered. This is the mode used for most C28x based projects.

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

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

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

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

Multiple instantiation of this macro is not supported.

Usage:

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

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

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

```
// ---------------------------- DPLIB Net Pointers ---------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//ADCDRV_4ch
extern volatile long *ADCDRV_4ch_RltA;
extern volatile long *ADCDRV_4ch_RltB;
extern volatile long *ADCDRV_4ch_RltC;
extern volatile long *ADCDRV_4ch_RltD;
```

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

Note signal net node names change from system to system, no dependency exist between these names and module.

```
// ---------------------------- DPLIB Variables ------------------------
// Declare the net variables being used by the DP Lib Macro here
volatile long RltA,RltB,RltC,RltD;
```

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

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


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

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

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

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

```
//----------Connect the macros to build a system-------------------
// Digital Power (DP) library initialisation
DPL_Init();
// ADCDRV_4ch block connections
ADCDRV_4ch_RltA=&RltA;
ADCDRV_4ch_RltB=&RltB;
ADCDRV_4ch_RltC=&RltC;
ADCDRV_4ch_RltD=&RltD;

// Initialize the net variables
RltA=_IQ24(0.0);
RltB=_IQ24(0.0);
RltC=_IQ24(0.0);
RltD=_IQ24(0.0);
```

Step 6 Add the ISR assembly file "`{ProjectName}-DPL-ISR.asm`" to the project.

Step 7 Include the macro's assembly file in the `{ProjectName}-DPL-ISR.asm`

```
;Include files for the Power Library Macro's being used by the system
.include "ADCDRV_4ch.asm"
```

Step 8 Instantiate the INIT macro in assembly (this is one-time pass through code) inside the C-callable DPL_Init() function which is defined in `{ProjectName}-DPL-ISR.asm`. Four numbers need to be specified to identify which result registers would be read and scaled results written to the respective pointers. The following code snippet would do the following

```
AdcResult.ADCRESULT0 -> (Scale) -> *(ADCDRV_4ch_RltA)

AdcResult.ADCRESULT5 -> (Scale) -> *(ADCDRV_4ch_RltB)

AdcResult.ADCRESULT2 -> (Scale) -> *(ADCDRV_4ch_RltC)

AdcResult.ADCSESULT1 -> (Scale) -> *(ADCDRV_4ch_RltD)
```

```
;Macro Specific Initialization Functions
ADCDRV_4ch_INIT 0,5,2,1 ; ADCDRV_4ch Initialization
```

Step 9 Call the run time macro in assembly inside the C-callable function DPL_ISR() which is the looped or ISR code. The function is defined in `{ProjectName}-DPL-ISR.asm`

```
;"Call" the Run macro
ADCDRV_4ch 0,5,2,1      ; Run ADCDRV_4ch
```

Step 10 Include the memory sections in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD`. Note that the net pointers and the internal data are forced to be placed in a single data page by use of the `.usect` directive in the source file of the module.

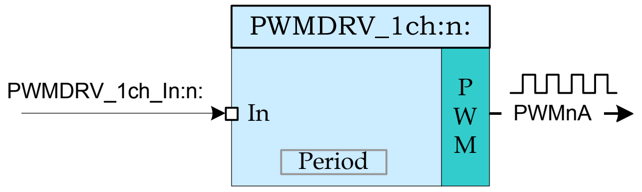
```
/*ADCDRV_4ch sections*/
ADCDRV_4ch_Section      : > dataRAM       PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
ADCDRV_4ch_RltA	Output Pointer	Pointer to 32 bit fixed point data location storing the result of the module.	Q24: [0, 1)
ADCDRV_4ch_RltB	Output Pointer	Pointer to 32 bit fixed point data location storing the result of the module.	Q24: [0, 1)
ADCDRV_4ch_RltC	Output Pointer	Pointer to 32 bit fixed point data location storing the result of the module.	Q24: [0, 1)
ADCDRV_4ch_RltD	Output Pointer	Pointer to 32 bit fixed point data location storing the result of the module.	Q24: [0, 1)

PWMDRV_1ch — *PWM Driver Single Channel*

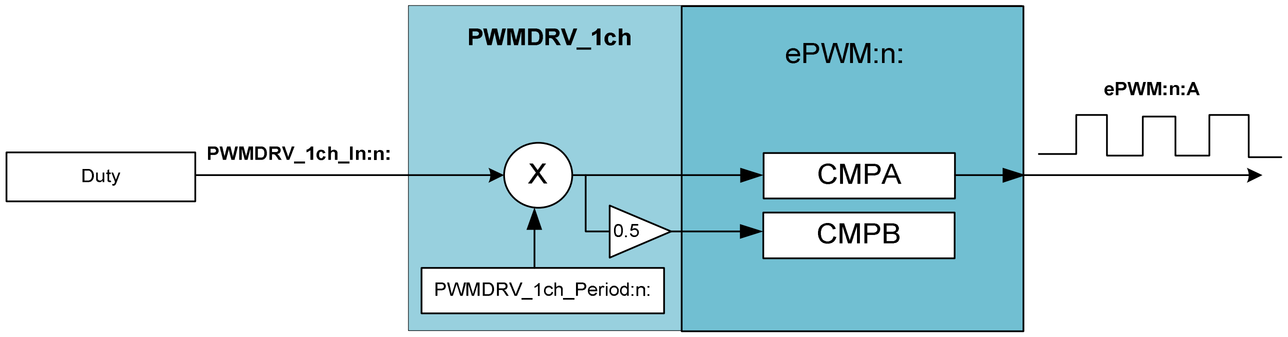
Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a duty on PWM channel A, dependent on the value of the input variable.



Macro File: `PWMDRV_1ch.asm`

Peripheral Initialization File: `PWM_1ch_Cnf.c`

Description: This assembly macro provides the interface between a DP library net variable and the ePWM module on C28x. The macro converts the unsigned Q24 input, pointed to by the net pointer PWMDRV_1ch_In:n: into an unsigned Q0 number scaled by the PWM period value, and stores this value in the EPwmRegs:n:.CMPA. The module also writes half the value of CMPA into CMPB register. This is done to enable ADC start of conversions to occur close to the mid point of the PWM waveform to avoid switching noise. Which PWM Macro is driven and what period value is used for scaling is determined by the instance number of the macro i.e. :n:.



This macro is used in conjunction with the peripheral configuration file PWM_1ch_Cnf.c. The file defines the function

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

where

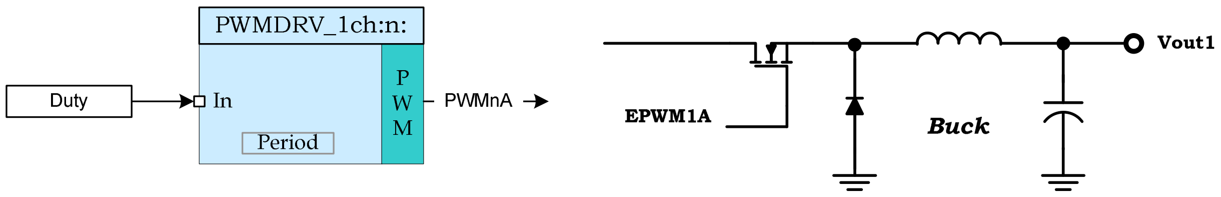
n	is the PWM Peripheral number which is configured in up count mode
Period	is the maximum count value of the PWM timer

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

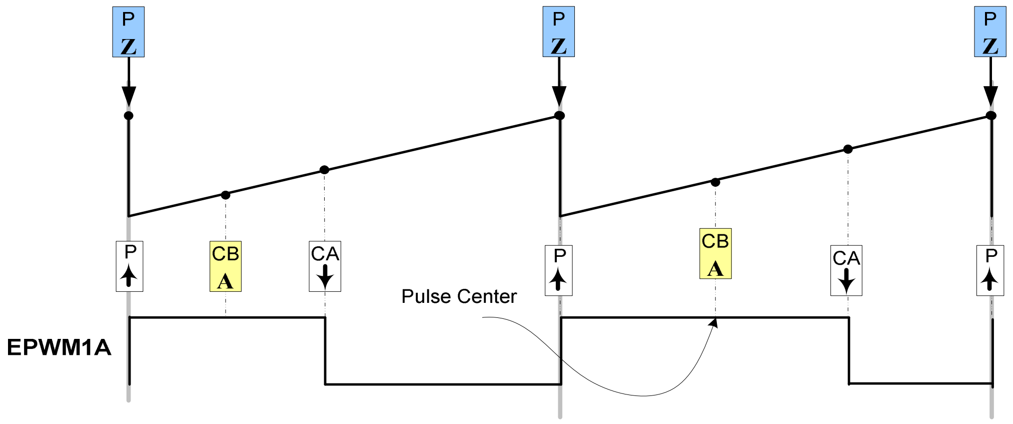
The function configures the PWM peripheral in up-count mode. The figure below, shows with help of a timing diagram how the PWM is configured to generate the waveform.

Detailed Description

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



Buck converter driven by PWMDRV_1ch module



PWM generation with the EPWM module.

Usage:

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

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

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

```
// ---------------------------- DPLIB Net Pointers ----------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PWMDRV_1ch - instance #1
extern volatile long *PWMDRV_1ch_In1;
extern volatile long PWMDRV_1ch_Period1;   // Optional
```

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

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

```
// ---------------------------- DPLIB Variables -----------------------
// Declare the net variables being used by the DP Lib Macro here
volatile long Duty;
```

Step 4 Call the peripheral configuration function `PWM_1ch_CNF(int16 n, int16 period, int16 mode, int16 phase)` in `{ProjectName}-Main.c`, this function is defined in `PWM_1ch_Cnf.c`. This file must be included manually into the project.

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

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

Step 5 "Call" the `DPL_Init()` to initialize the macros and **"connect"** the module terminals to the Signal nets in "C" in `{ProjectName}-Main.c`. Also note the `DPL_Init()` function calls the `PWMDRV_1ch_INIT` function. This function initializes the value of `PWMDRV_1ch_Period:n:` with the period value of PWM being used. Hence the value of PWM period must be stored before calling this function i.e. step 4 must be carried out before this step.

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

// Digital Power (DP) library initialisation
DPL_Init();
// PWMDRV_1ch block connections
PWMDRV_1ch_In1=&Duty;
// Initialize the net variables
Duty=_IQ24(0.0);
```

Step 6 Add the ISR assembly file "`{ProjectName}-DPL-ISR.asm`" to the project.

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

```
;Include files for the Power Library Macro's being used by the system
.include "PWMDRV_1ch.asm"
```

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

```
;Macro Specific Initialization Functions
PWMDRV_1ch_INIT 1 ; PWMDRV_1ch Initialization
```

`Step 9` **Call the run time macro** in assembly inside the C-callable function DPL_ISR() which is the looped or ISR code. The function is defined in `{ProjectName}-DPL-ISR.asm`

```
;"Call" the Run macro
PWMDRV_1ch 1      ; Run PWMDRV_1ch (Note EPWM1 is used for instance#1)
```

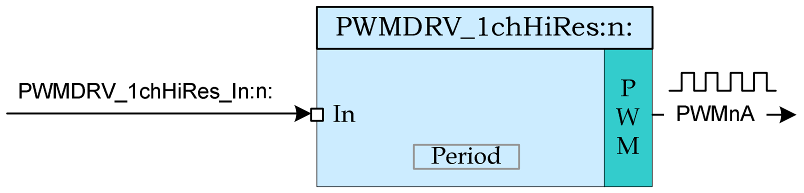
`Step 10` **Include the memory sections** in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD`. Note that the net pointers and the internal data are forced to be placed in a single data page by use of the `.usect` directive in the source file of the module.

```
/*PWMDRV_1ch sections*/
PWMDRV_1ch_Section      : > dataRAM       PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
PWMDRV_1ch_In:n:	Input Pointer	Pointer to 32 bit fixed point input data location storing Duty Value	Q24: [0, 1)
PWMDRV_1ch_Period:n:	Internal Data	Location storing the period value of the PWM being driven (This is done to save 1 cycles penalty in reading the TBPRDM register)	Q16: [0, 65536)

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

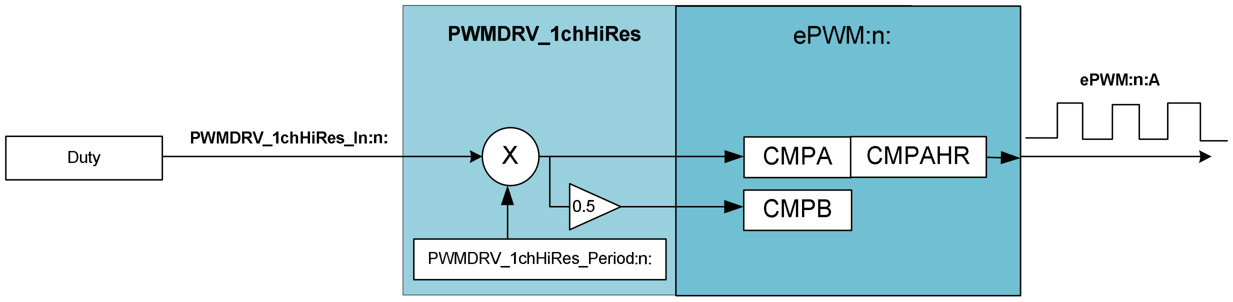


Macro File: `PWMDRV_1chHiRes.asm`

Peripheral Initialization File: `PWM_1chHiRes_Cnf.c`

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

The assembly macro provides the interface between a DP library net variable and the ePWM module on C28x. The macro converts the unsigned Q24 input, pointed to by the Net Pointer PWMDRV_1chHiRes_In:n: into an unsigned Q0 number scaled by the PWM period value, and stores this value in the EPwmRegs:n:.CMPA and EPwmRegs:n:.CMPAHR. The module also writes half the value of CMPA into CMPB register. This is done to enable ADC Start of conversions to occur close to the mid point of the PWM waveform to avoid switching noise. Which PWM Macro is driven and what period value is used for scaling is determined by the instance number of the macro i.e. :n:.



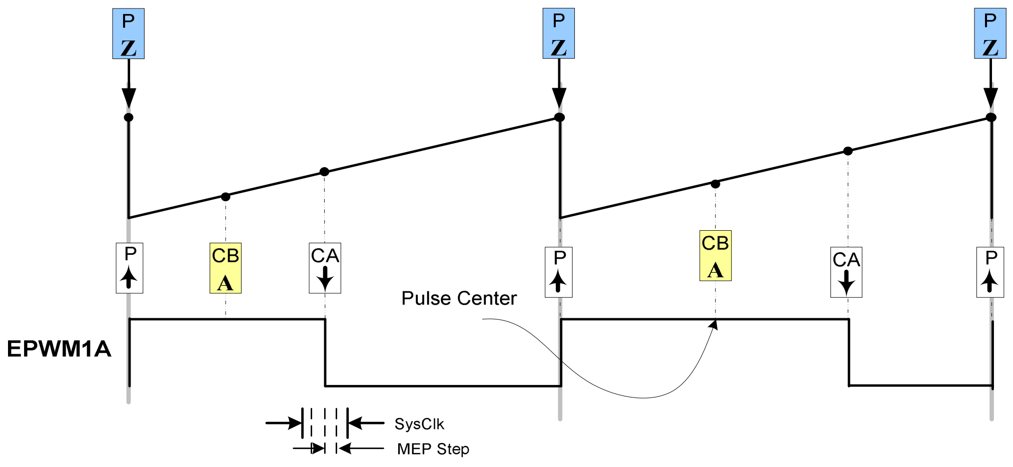
This macro is used in conjunction with the Peripheral configuration file PWM_1chHiRes_Cnf.c. The file defines the function

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

where

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

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



PWM generation with the EPWM module.

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

Usage:

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

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

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

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

```
// ---------------------------- DPLIB Net Pointers ----------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PWMDRV_1chHiRes - instance #1
extern volatile long *PWMDRV_1chHiRes_In1;
extern volatile long PWMDRV_1chHiRes_Period1;    // Optional
//=====================================================================
// The following declarations are required in order to use the SFO
// library functions:
//
int MEP_ScaleFactor; // Global variable used by the SFO library
                     // Result can be used for all HRPWM channels
                     // This variable is also copied to HRMSTEP
                     // register by SFO() function.

int status;
```

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

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

```
// ---------------------------- DPLIB Variables -----------------------
// Declare the net variables being used by the DP Lib Macro here
volatile long Duty;
```

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

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

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

Step 5 **"Call"** the `DPL_Init()` to initialize the macros and **"connect"** the module terminals to the signal nets in "C" in `{ProjectName}-Main.c`. Also note the `DPL_Init()` function calls the `PWMDRV_1ch_INIT` function. This function initializes the value of `PWMDRV_1chHiRes_Period:n:` with the period value of PWM being used. Hence the value of PWM period must be stored before calling this function i.e. step 4 must be carried out before this step. **"Call"** the SFO() function to calculate the HRMSTEP, and update the HRMSTEP register if calibration function returns without error. The User may want to call this function in a background task to account for changing operating conditions.

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

// Digital Power (DP) library initialisation
DPL_Init();
// PWMDRV_1ch block connections
PWMDRV_1chHiRes_In1=&Duty;
// Calling SFO() updates the HRMSTEP register with calibrated
MEP_ScaleFactor.
// MEP_ScaleFactor/HRMSTEP must be filled with calibrated value in order
// for the module to work
status = SFO_INCOMPLETE;

while  (status== SFO_INCOMPLETE){  // Call until complete
        status = SFO();
}
if(status!=SFO_ERROR) { // IF SFO() is complete with no errors
      EALLOW;
      EPwm1Regs.HRMSTEP=MEP_ScaleFactor;
      EDIS;
}
if (status == SFO_ERROR) {
            while(1); // SFO function returns 2 if an error occurs
                     // The code would loop here for infinity if it
                     // returns an error
}
// Initialize the net variables
Duty=_IQ24(0.0);
```

Step 6 **Add** the ISR assembly file "`{ProjectName}-DPL-ISR.asm`" to the project.

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

```
;Include files for the Power Library Macro's being used by the system
.include "PWMDRV_1chHiRes.asm"
```

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

```
;Macro Specific Initialization Functions
PWMDRV_1chHiRes_INIT 1  ; PWMDRV_1ch Initialization
```

Step 9 **Call the run time macro** in assembly inside the C-callable function DPL_ISR() which is the looped or ISR code. The function is defined in `{ProjectName}-DPL-ISR.asm`

```
;"Call" the Run macro
PWMDRV_1chHiRes 1 ; Run PWMDRV_1ch (Note EPWM1 is used for instance#1)
```

`Step 10` **Include the memory sections** in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD`. Note that the net pointers and the internal data are forced to be placed in a single data page by use of the `.usect` directive in the source file of the module.

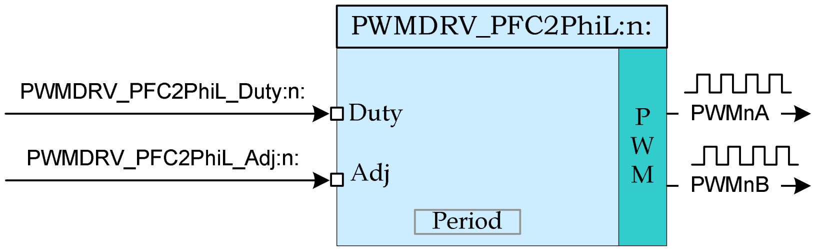
```
/*PWMDRV_1ch sections*/
PWMDRV_1chHiRes_Section : > dataRAM       PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
PWMDRV_1chHiRes_In:n:	Input Pointer	Pointer to 32 bit fixed point input data location storing Duty Value	Q24: [0, 1)
PWMDRV_1chHiRes_Period:n:	Internal Data	Location storing the period value of the PWM being driven (This is done to save 1 cycles penalty in reading the TBPRDM register)	Q16: [0, 65536)

PWMDRV_PFC2PhiL — *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 cane be used to drive a 2 phase interleaved PFC stage.



Macro File: `PWMDRV_PFC2PhiL.asm`

Peripheral Initialization File: `PWM_PFC2PhiL_Cnf.c`

Description: This module forms the interface between the control software and the device PWM pins. The macro converts the unsigned Q24 input pointed to by the Net Pointer PWMDRV_PFC2PhiL_Duty:n: into an unsigned Q0 number scaled by the PWM period value, and stores this value in the EPwmRegs:n:.CMPA. Which PWM module is written to is determined by the instance number of the macro i.e. :n:.

The value pointed by the PWMDRV_PFC2PhiL_Adj:n: stores a Q24 number that is scaled with the PWM period value to add an offset to the duty being driven on PWMnA and PWMnB. The value stored can be positive or negative depending on whether the duty driven on PWMnB needs to greater or smaller relative to PWMnA. In summary:

$$CMPA = Duty * PWMPeriod$$

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

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

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

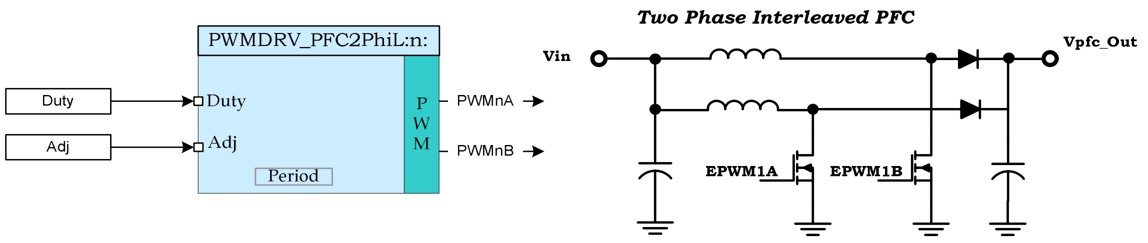
where

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

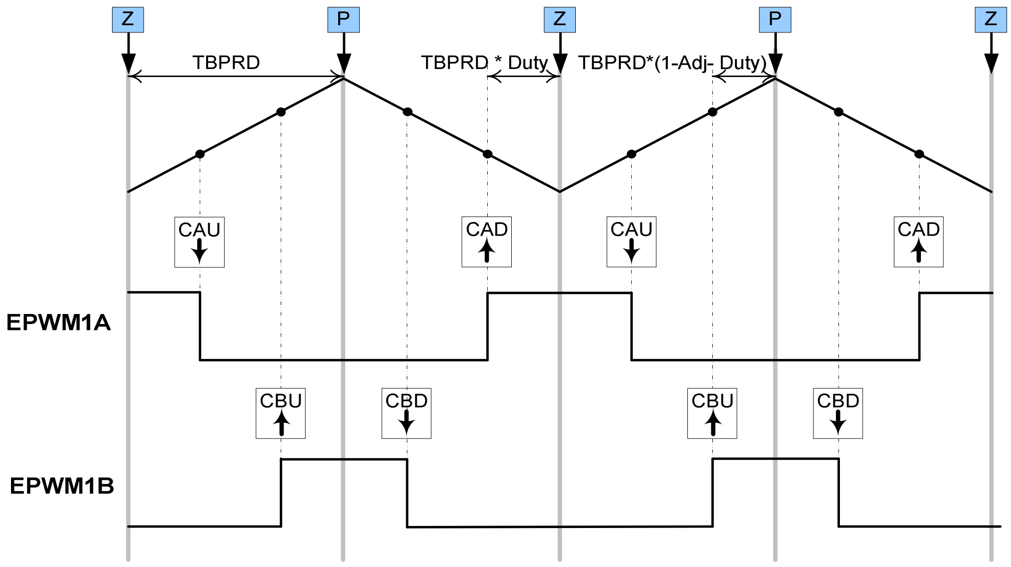
The function configures the PWM peripheral in up down count mode. The figure below, shows with help of a timing diagram, how the PWM is configured to generate the waveform. When in up down count mode the zero and period event can be used to trigger ADC conversions at mid point of the switching duty.

Detailed Description

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



PFC2PhiL driven by PWMDRV_PFC2PhiL module



PWM generation for PFC2PhiL stage with the F280x EPWM module.

Usage:

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

```
#include “DPLib.h”
```

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

```
// ---------------------------- DPLIB Net Pointers ----------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PWMDRV_PFC2PhiL - instance #1
extern volatile long *PWMDRV_PFC2PhiL_Duty1;
extern volatile long *PWMDRV_PFC2PhiL_Adj1;
extern volatile long PWMDRV_PFC2PhiL_Period1;   // Optional
```

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

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

```
// ---------------------------- DPLIB Variables -----------------------
// Declare the net variables being used by the DP Lib Macro here
volatile long Duty, Adj;
```

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

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

PWM_PFC2PHIL_CNF(1,300);
```

Step 5 "Call" the `DPL_Init()` function and then **"connect"** the module terminals to the Signal nets in "C" in `{ProjectName}-Main.c` The `DPL_Init()` function must be called before the signal nets are connected. Also note the `DPL_Init()` function calls the `PWMDRV_PFC2PhiL_INIT` function. This function also initialize the value of `PWMDRV_PFC2PhiL_Period:n:` with the period value of PWM being used. Hence the value of PWM period must be stored before calling this function i.e. step 4 must be carried out before this step.

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

// Digital Power (DP) library initialisation
DPL_Init();
// PWMDRV_PFC2PhiL block connections
PWMDRV_PFC2PhiL_Duty1=&Duty;
PWMDRV_PFC2PhiL_Adj1 =&Adj;
// Initialize the net variables
Duty=_IQ24(0.0);
Adj =_IQ24(0.0);
```

Step 6 Add the ISR assembly file "`{ProjectName}-DPL-ISR.asm`" to the project

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

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

.include "PWMDRV_PFC2PhiL.asm"
```

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

```
;Macro Specific Initialization Functions
PWMDRV_PFC2PHIL_INIT 1  ; PWMDRV_PFC2PHIL Initialization
```

Step 9 **Call the run time macro** in assembly inside the C-callable function DPL_ISR() which is the looped or ISR code. The function is defined in `{ProjectName}-DPL-ISR.asm`

```
;"Call" the Run macro
PWMDRV_PFC2PHIL 1 ; Run PWMDRV_PFC2PHIL (EPWM1 is used by instance#1)
```

Step 10 **Include the memory sections** in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD`. Note that the net pointers and the internal data are forced to be placed in a single data page by use of the `.usect` directive in the source file of the module.

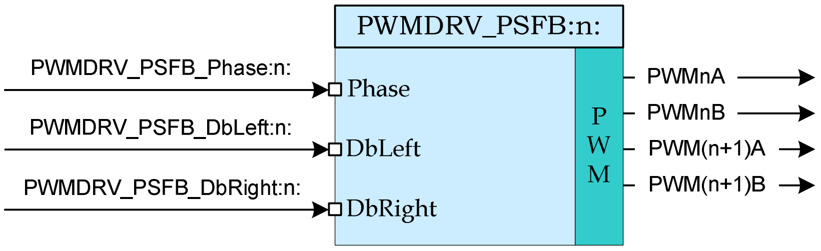
```
/*PWMDRV_PFC2PhiL sections*/
PWMDRV_PFC2PhiL_Section : > RAML2         PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
PWMDRV_PFC2PhiL_Duty:n:	Input Pointer	Pointer to 32 bit fixed point input data location to Duty value	Q24(0,1)
PWMDRV_PFC2PhiL_Adj:n:	Input Pointer	Pointer to 32 bit fixed point input data location to adjustment value	Q24(-1, 1)
PWMDRV_PFC2PhiL_Period:n:	Internal Data	Location storing the period value of the PWM being driven (This is done to save 1 cycles penalty in reading the TBPRDM register)	Q16 (0,65536)

PWMDRV_PSFB — *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, whereby providing the zero voltage switching capabilities. In addition to phase control, the module offers control over left and right leg dead-band amounts



Macro File: `PWMDRV_PSFB.asm`

Peripheral Initialization File: `PWM_PSFB_Cnf.c`

Description: This module forms the interface between the control software and the device PWM pins. The macro converts the unsigned Q24 input pointed to by the Net Pointer PWMDRV_PSFB_Phase:n: into an unsigned Q0 number scaled by the PWM period value, and stores this value in the EPwmRegs:n:.TBPHS.

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

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

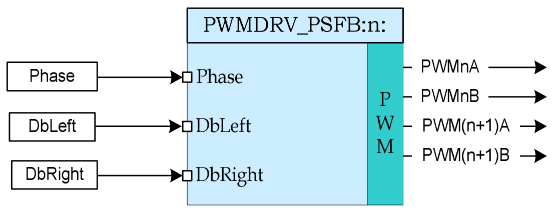
where

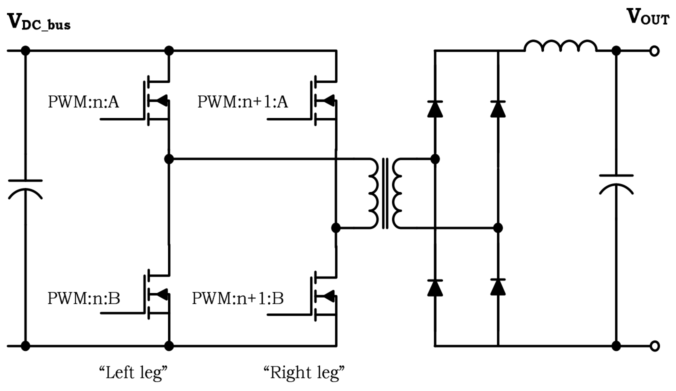
n	is the PWM Peripheral number configured for PSFB topology, PWM n+1 is configured to work with synch pulses from PWM n module
Period	is the maximum count value of the PWM timer

The figure below, shows with

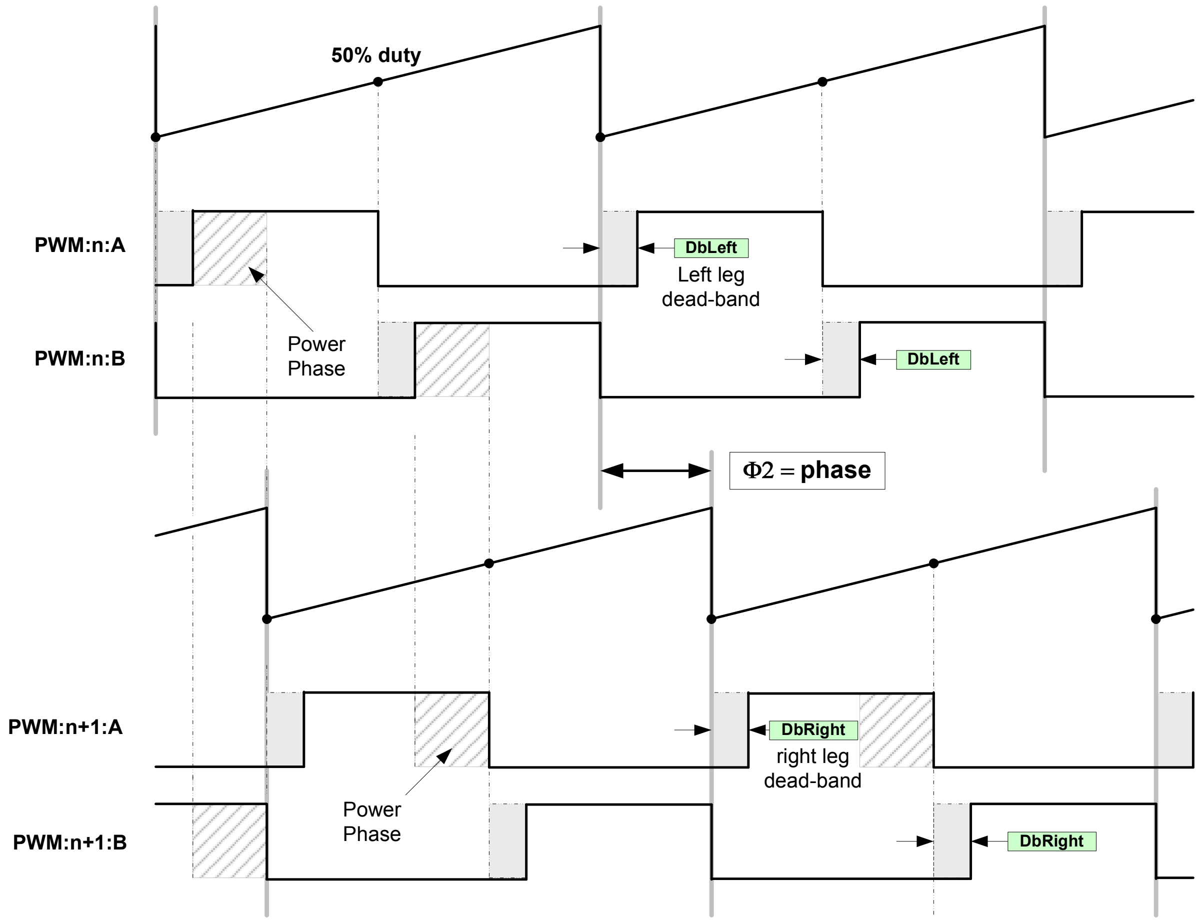
Detailed Description

The following section explains with help of a timing diagram, how the PWM is configured to generate the waveform for the 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



Phase shifted PWM generation with the EPWM module.

Usage:

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

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

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

```
// ---------------------------- DPLIB Net Pointers ---------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PWMDRV_PSFB - instance #1
extern volatile long *PWMDRV_PSFB_Phase1;
extern volatile long *PWMDRV_PSFB_DbLeft1;
extern volatile long *PWMDRV_PSFB_DbRight1;   // Optional
```

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

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

```
// ---------------------------- DPLIB Variables -------------------------
// Declare the net variables being used by the DP Lib Macro here
volatile long Phase, DbLeft, DbRight;
```

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

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

PWM_PSFB_CNF(1,600);
```

`Step 5` **"Call"** the `DPL_Init()` function and then **"connect"** the module terminals to the Signal nets in "C" in `{ProjectName}-Main.c` The `DPL_Init()` function must be called before the signal nets are connected.

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

// Digital Power (DP) library initialisation
DPL_Init();
// PWMDRV_PSFB block connections
PWMDRV_PSFB_Duty1   =   &Phase;
PWMDRV_PSFB_DbLeft1 =   &DbLeft;
PWMDRV_PSFB_DbRight1=   &DbRight;

// Initialize the net variables
Phase=_IQ24(0.0);
DbLeft=0;
DbRight=0;
```

`Step 6` **Add** the ISR assembly file "`{ProjectName}-DPL-ISR.asm`" to the project

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

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

.include "PWMDRV_PSFB.asm"
```

`Step 8` **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-callable DPL_Init() function which is defined in `{ProjectName}-DPL-ISR.asm.` Note when instantiating both n and n+1 needs to be passed as arguments. Any other number except n+1 would lead to unexpected behavior.

```
;Macro Specific Initialization Functions
PWMDRV_PSFB_INIT 1,2     ; PWMDRV_PSFB Initialization
```

`Step 9` **Call the run time macro** in assembly inside the C-callable function DPL_ISR() which is the looped or ISR code. The function is defined in `{ProjectName}-DPL-ISR.asm.` Note when calling the run time macro both n and n+1 needs to be passed as arguments. Any other number except n+1 would lead to unexpected behavior.

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

`Step 10` **Include the memory sections** in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD.` Note that the net pointers and the internal data are forced to be placed in a single data page by use of the `.usect` directive in the source file of the module.

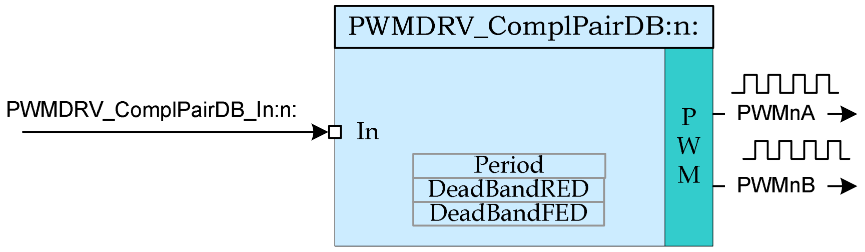
```
/*PWMDRV_PSFB sections*/
PWMDRV_PSFB_Section     : > dataRAM       PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
PWMDRV_PSFB_Phase:n:	Input Pointer	Pointer to 32 bit fixed point input data location to Phase value	Q24(0,1)
PWMDRV_PSFB_DbLeft:n:	Input Pointer	Pointer to 32 bit fixed point input data location to Dead Band Value for Left Leg	Q0
PWMDRV_PSFB_DbRight:n:	Input Pointer	Pointer to 32 bit fixed point input data location to Dead Band Value for Right Leg	Q0

PWMDRV_ComplPairDB — *PWM Driver to output complimentary chA and chB PWM*

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



Macro File: `PWMDRV_ComplPairDB.asm`

Peripheral Initialization File: `PWM_ComplPairDB_Cnf.c`

Description: This assembly macro provides the interface between a DP library net variable and the ePWM module on C28x. The macro converts the unsigned Q24 input, pointed to by the Net Pointer PWMDRV_ComplPairDB_In:n: into an unsigned Q0 number scaled by the PWM period value, and stores this value in the EPwmRegs:n:.CMPA. The corresponding configuration file has the deadband module configured to output complimentary waveform on chB with a dead band.

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

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

where

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

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

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

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

where

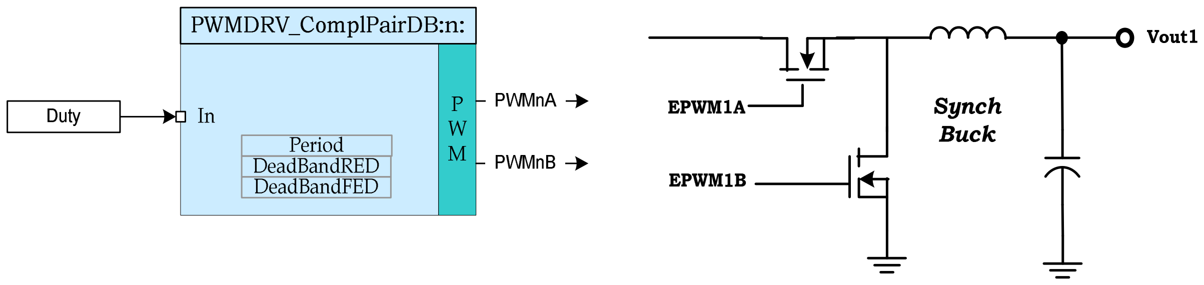
n	is the PWM Peripheral number
DbRed	is the new rising edge dealy
DbFed	is the new falling edge delay

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

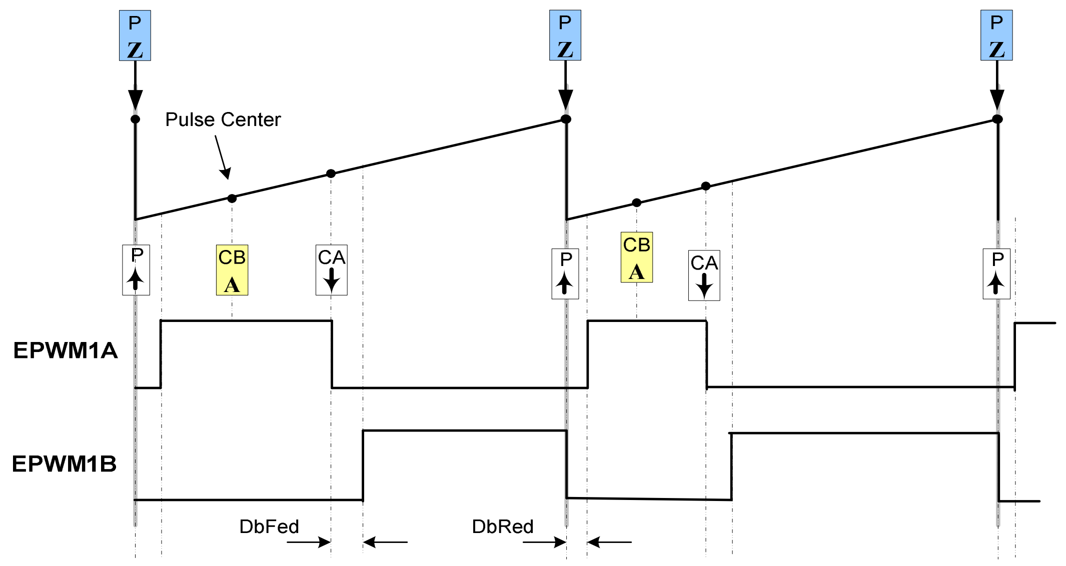
.

Detailed Description

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



Synchronous Buck converter driven by PWMDRV_ComplPairDB module



PWM generation for CompPairDB PWM DRV Maco

Usage:

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

```
#include “DPLib.h”
```

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

```
// ---------------------------- DPLIB Net Pointers ----------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PWMDRV_ComplPairDB - instance #1
extern volatile long *PWMDRV_ComplPairDB_In1;
extern volatile long PWMDRV_ComplPairDB_Period1;   // Optional
extern volatile int16 PWMDRV_ComplPairDB_DeadBandRED1;   // Optional
extern volatile int16 PWMDRV_ComplPairDB_DeadBandFED1;   // Optional
```

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

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

```
// ---------------------------- DPLIB Variables -----------------------
// Declare the net variables being used by the DP Lib Macro here
volatile long Duty;
```

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

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

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

Step 5 **"Call"** the `DPL_Init()` to initialize the macros and **"connect"** the module terminals to the Signal nets in "C" in `{ProjectName}-Main.c`. Also note the `DPL_Init()` function would call the `PWMDRV_ComplPairDB_INIT` function. This function initializes the value of `PWMDRV_ComplPairDB_Period:n:` with the period value of PWM being used. Hence the value of PWM period must be stored before calling this function i.e. step 4 must be carried out before this step.

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

// Digital Power (DP) library initialisation
DPL_Init();
// PWMDRV_ComplPairDB block connections
PWMDRV_ComplPairDB_In1=&Duty;
// Initialize the net variables
Duty=_IQ24(0.0);
```

Step 6 **Add** the ISR assembly file "`{ProjectName}-DPL-ISR.asm`" to the project.

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

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

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

```
;Macro Specific Initialization Functions
PWMDRV_ComplPairDB_INIT 1
```

Step 9 **Call the run time macro** in assembly inside the C-callable function DPL_ISR() which is the looped or ISR code. The function is defined in `{ProjectName}-DPL-ISR.asm`

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

Step 10 **Include the memory sections** in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD`. Note that the net pointers and the internal data are forced to be placed in a single data page by use of the `.usect` directive in the source file of the module.

```
/*PWMDRV_ComplPairDB sections*/
PWMDRV_ComplPairDB_Section    : > dataRAM       PAGE = 1
```

Step 11 Update Dead Band This can be done by calling the C function in the `{ProjectName}-Main.c file.`

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

If the dead band itself is part of the control loop the following assembly maco can be called.

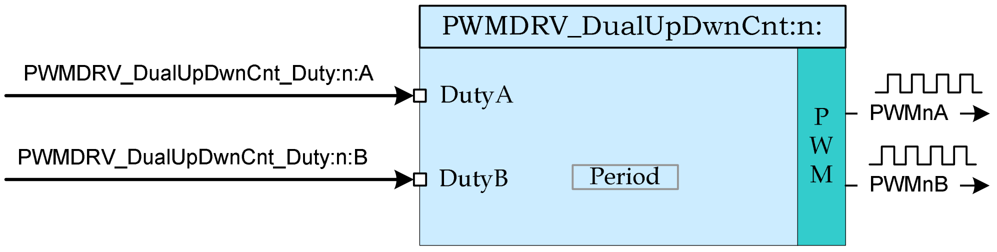
```
;Update dead band delays
PWMDRV_ComplPairDB_Update 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
PWMDRV_ComplPairDB_In:n:	Input Pointer	Pointer to 32 bit fixed point input data location storing Duty Value	Q24: [0, 1)
PWMDRV_ComplPairDB_DeadBandRED:n:	Input Variable	Value used by the assembly macro to update the PWM peripheral dead band registers.	Q0
PWMDRV_ComplPairDB_DeadBandFED:n:	Input Variable	Value used by the assembly macro to update the PWM peripheral dead band registers.	Q0
PWMDRV_ComplPairDB_Period:n:	Internal Data	Location storing the period value of the PWM being driven (This is done to save 1 cycles penalty in reading the TBPRDM register)	Q16: [0, 65536)

PWMDRV_DualUpDwnCnt *Dual PWM Driver, independent Duty on chA & chB*

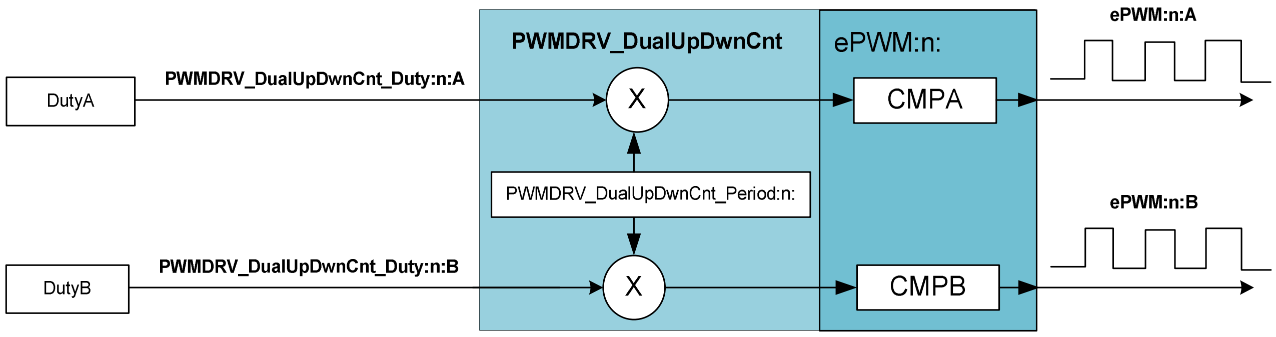
Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a duty on PWM channel A, and PWM channel B dependent on the value of the input variables DutyA and DutyB.



Macro File: `PWMDRV_DualUpDwnCnt.asm`

Peripheral Initialization File: `PWM_DualUpDwnCnt_Cnf.c`

Description: This assembly macro provides the interface between DP library net variables and the ePWM module on C28x. The macro converts the unsigned Q24 input, pointed to by the Net Pointers PWMDRV_DualUPDwnCnt_Duty:n:A and PWMDRV_DualUPDwnCnt_Duty:n:B into an unsigned Q0 number scaled by the PWM period value, and stores this value in the EPwmRegs:n:.CMPA and EPwmRegs:n:.CMPB such as to give independty duty cycle control on channel A and B of the PWM module.


PWMDRV_DualUpDwnCnt
ePWM:n:
ePWM:n:A
DutyA
PWMDRV_DualUpDwnCnt_Duty:n:A
X
CMPA
PWMDRV_DualUpDwnCnt_Period:n:
ePWM:n:B
DutyB
PWMDRV_DualUpDwnCnt_Duty:n:B
X
CMPB

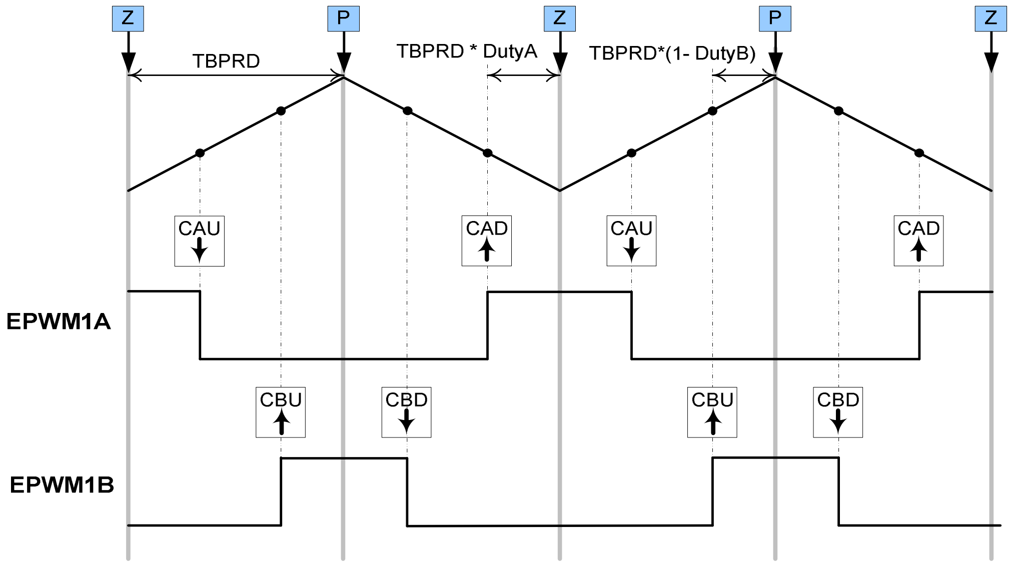
This macro is used in conjunction with the peripheral configuration file PWM_DualUpDwnCnt_Cnf.c. The file defines the function

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

where

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

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



PWM generation with the EPWM module.

Usage:

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

```
#include “DPLib.h”
```

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

```
// ----------------------------- DPLIB Net Pointers ----------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PWMDRV_DualUpDwnCnt - instance #1
extern volatile long *PWMDRV_DualUpDwnCnt_Duty1A;
extern volatile long *PWMDRV_DualUpDwnCnt_Duty1B;
extern volatile long PWMDRV_DualUpDwnCnt_Period1;    // Optional
```

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

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

```
// ---------------------------- DPLIB Variables -----------------------
// Declare the net variables being used by the DP Lib Macro here
volatile long DutyA,DutyB;
```

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

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

PWM_DualUpDwnCnt_CNF(1,300);
```

Step 5 **“Call”** the `DPL_Init()` to initialize the macros and **“connect”** the module terminals to the Signal nets in “C” in `{ProjectName}-Main.c.` Also note the `DPL_Init()` function calls the `PWMDRV_DualUpDwnCnt_INIT` function. This function initializes the value of `PWMDRV_DualUpDwnCnt_Period:n:` with the period value of PWM being used. Hence the value of PWM period must be stored before calling this function i.e. step 4 must be carried out before this step.

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

// Digital Power (DP) library initialisation
DPL_Init();
// PWMDRV_1ch block connections
PWMDRV_DualUpDwnCnt_Duty1A=&DutyA;
PWMDRV_DualUpDwnCnt_Duty1B=&DutyB;

// Initialize the net variables
DutyA=_IQ24(0.0);
DutyB=_IQ24(0.0);
```

Step 6 **Add** the ISR assembly file “`{ProjectName}-DPL-ISR.asm`” to the project.

Step 7 **Include** the macro’s assembly file in the `{ProjectName}-DPL-ISR.asm`

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

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

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

`Step 9` **Call the run time macro** in assembly inside the C-callable function DPL_ISR() which is the looped or ISR code. The function is defined in `{ProjectName}-DPL-ISR.asm`

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

`Step 10` **Include the memory sections** in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD`. Note that the net pointers and the internal data are forced to be placed in a single data page by use of the `.usect` directive in the source file of the module.

```
/*PWMDRV_DualUpDwnCnt sections*/
PWMDRV_DualUpDwnCnt_Section    : > dataRAM       PAGE = 1
```

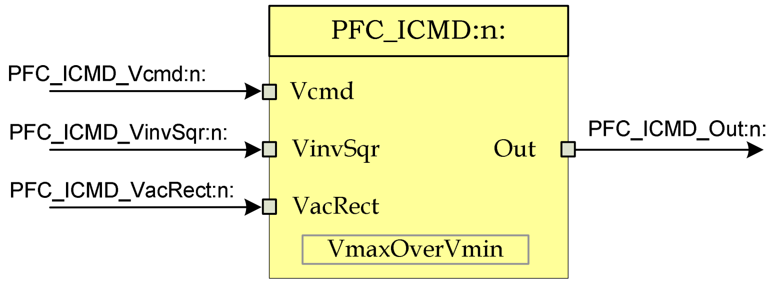
Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
PWMDRV_DualUpDwnCnt_Duty:n:A	Input Pointer	Pointer to 32 bit fixed point input data location storing DutyA Value	Q24: [0, 1)
PWMDRV_DualUpDwnCnt_Duty:n:B	Input Pointer	Pointer to 32 bit fixed point input data location storing DutyB Value	Q24: [0, 1)
PWMDRV_ DualUpDwnCnt _Period:n:	Internal Data	Location storing the period value of the PWM being driven (This is done to save 1 cycles penalty in reading the TBPRDM register)	Q16: [0, 65536)

5.3. Application Specific

PFC_ICMD *Current Command for Power Factor Correction*

Description: This software module performs a computation of the current command for the Power Factor Correction(PFC)



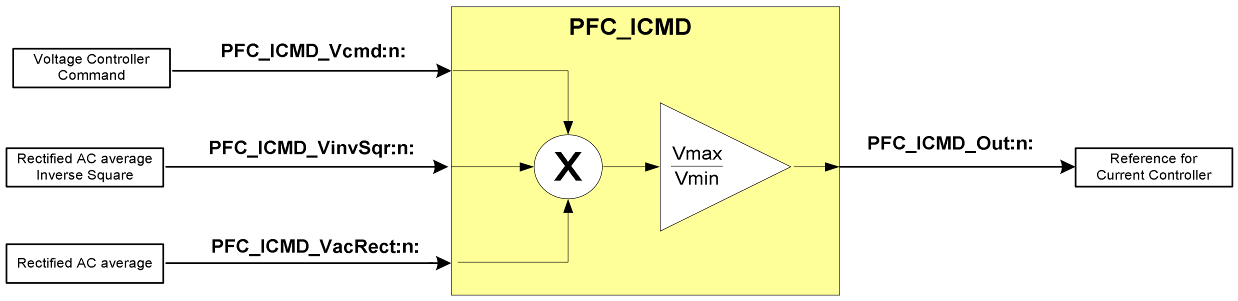
Macro File: `PFC_ICMD.asm`

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

The input pointers `PFC_ICMD_Vcmd:n:`, `PFC_ICMD_VinvSqr:n:` and `PFC_ICMD_Vcmd:n:` points to a variable represented in Q24 format. The module multiplies these values together and then scales them by multiplying with a factor which is stored in the internal data `PFC_ICMD_VmaxOverVmin:n:` The result in Q24 format is written to the variable pointed by the output pointer `PFC_ICMD_Out:n:`

A PFC stage is typically designed to work over a range of AC line conditions. `PFC_ICMD_VminOverVmax:n:` is the ratio of minimum over maximum voltage the PFC stage is designed for represented in the Q24 format.

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



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

```
PFC_ICMD_VmaxOverVmin:n:  = _IQ24(230/90)=_IQ24(2.5555)
```

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

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

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

```
// ---------------------------- DPLIB Net Pointers ----------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PFC_ICMD - instance #1
extern volatile long *PFC_ICMD_Vcmd1;
extern volatile long *PFC_ICMD_VacRect1;
extern volatile long *PFC_ICMD_VinvSqr1;
extern volatile long *PFC_ICMD_Out1;
extern volatile long PFC_ICMD_VmaxOverVmin1;
```

Step 3 Declare signal net nodes/ variables in C in the file {ProjectName}-Main.c

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

```
// ---------------------------- DPLIB Variables -----------------------
// Declare the net variables being used by the DP Lib Macro here
volatile long Vcmd1,VacRect,VinvSqr,CurrCmd;
```

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

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

// Digital Power (DP) library initialisation
DPL_Init();
// PFC_ICMD block connections
PFC_ICMD_Vcmd1=&Vcmd;
PFC_ICMD_VacRect1=&VacRect;
PFC_ICMD_VinvSqr1=&VinvSqr;
PFC_ICMD_Out1=&CurrCmd;
PFC_ICMD_VmaxOverVmin1=_IQ24(2.5555);

// Initialize the net variables
Vcmd=_IQ24(0.0);
VinvSqr=_IQ24(0.0)
VacRect=_IQ24(0.0);
CurrCmd=_IQ24(0.0);
```

Step 5 **Add** the ISR assembly file "`{ProjectName}-DPL-ISR.asm`" to the project

Step 6 Include the macro's assembly file in the `{ProjectName}-DPL-ISR.asm`

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

.include "PFC_ICMD.asm"
```

Step 7 Instantiate the INIT macro in assembly (this is one-time pass through code) inside the C-callable DPL_Init() function which is defined in `{ProjectName}-DPL-ISR.asm`

```
;Macro Specific Initialization Functions
PFC_ICMD_INIT 1    ; PFC_ICMD Initialization
```

Step 8 Call the run time macro in assembly inside the C-callable function DPL_ISR() which is the looped or ISR code. The function is defined in `{ProjectName}-DPL-ISR.asm`

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

Step 9 Include the memory sections in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD`. Note that the net pointers and the internal data are forced to be placed in a single data page by use of the `.usect` directive in the source file of the module.

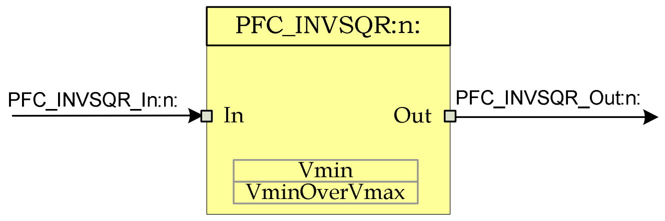
```
/*PFC_ICMD sections*/
PFC_ICMD_Section   : > RAML2          PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range**
PFC_ICMD_Vcmd:n:	Input Pointer	Pointer to 32 bit fixed point input data location storing the output of the voltage controller	Q24: [0, 1)
PFC_ICMD_VinvSqr:n:	Input Pointer	Pointer to 32 bit fixed point input data location storing the output of the PFC_INVSQR block	Q24: [0, 1)
PFC_ICMD_VacRect:n:	Input Pointer	Pointer to 32 bit fixed point input data location storing the output of the MATH_EMAVG block	Q30:[0,1)
PFC_ICMD_Out:n:	Output Pointer	Pointer to 32 bit fixed point output location where the reference for the current loop is stored	Q24:[0,1)
PFC_ICMD_VmaxOverVmin:n:	Internal Data	Data Variable storing the scaling factor	Q24:[0,8)

PFC_INVSQR *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.

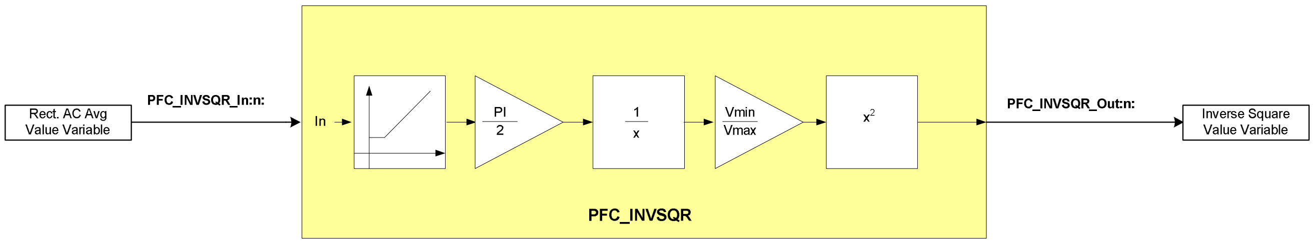


Macro File: `PFC_INVSQR.asm`

Technical: The input pointer `PFC_INSQR_In:n:` points to a variable represented in Q24 format. The module scales and inverts this value and writes the result in Q24 format to a variable pointed by the output pointer `PFC_INVSQR_Out:n:` The module uses two internal data variables to specify the range and scaling, which is dependent on the Power Factor Correction(PFC) stage the module is used for.

A PFC stage is typically designed to work over a range of AC line conditions. `PFC_INVSQR_VminOverVmax:n:` is the ratio of minimum over maximum voltage the PFC stage is designed for represented in the Q30 format. The `PFC_INVSQR_Vmin:n:` is equal or less than the minimum AC Line voltage that the PFC stage is designed to run for represented in the Q24 format. Note that `PFC_INVSQR_Vmin:n:` depends on what range the Voltage Feedback in the PFC system is designed for.

The module allows for the fact that the input value is the average of a half-sine (rectified AC), whereas what is desired for power factor correction is the representation of the peak of the sine. In addition the input signal is clamped to a minimum to allow the PFC system to work with very low line voltages without overflows, which can cause undesirable effects. The module also saturates the output for a maximum of 1.0 in Q24 format. The following diagram illustrates the math function operated on in this block.



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

```
PFC_INVSQR_VminOverVmax:n:  = _IQ30(90/230)=_IQ30(0.3913)

PFC_INVSQR_Vmin:n:          =<_IQ24(90/400)=_IQ24(0.225)
```

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

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

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

```
// ---------------------------- DPLIB Net Pointers ---------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//PFC_INVSQR - instance #1
extern volatile long *PFC_INVSQR_In1;
extern volatile long *PFC_INVSQR_Out1;
extern volatile long PFC_INVSQR_VminOverVmax1;
extern volatile long PFC_INVSQR_Vmin1;
```

Step 3 Declare signal net nodes/ variables in C in the file {ProjectName}-Main.c

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

```
// ---------------------------- DPLIB Variables -----------------------
// Declare the net variables being used by the DP Lib Macro here
volatile long In, Out;
```

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

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

// Digital Power (DP) library initialisation
DPL_Init();
// PFC_INVSQR block connections
PFC_INVSQR_In1=&In;
PFC_INVSQR_Out1=&Out;
PFC_INVSQR_VminOverVmax1=_IQ30(0.3913);

PFC_INVSQR_Vmin1=_IQ24(0.225);

// Initialize the net variables
In=_IQ24(0.0);

Out=_IQ24(0.0);
```

Step 5 **Add** the ISR assembly file "{ProjectName}-DPL-ISR.asm" to the project

Step 6 **Include** the macro's assembly file in the {ProjectName}-DPL-ISR.asm

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

.include "PFC_INVSQR.asm"
```

Step 7 **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-callable DPL_Init() function which is defined in{ProjectName}-DPL-ISR.asm

```
;Macro Specific Initialization Functions
PFC_INVSQR_INIT 1 ; PFC_INVSQR Initialization
```

Step 6 **Call** run time macro in assembly inside the C-callable function DPL_ISR() which is the looped or ISR code. The function is defined in{ProjectName}-DPL-ISR.asm

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

Step 7 **Include the memory sections** in the {DeviceName}-{RAM/FLASH}-{ProjectName}.CMD. Note that the net pointers and the internal data are forced to be placed in a single data page by use of the .usect directive in the source file of the module.

```
/*PFC_INVSQR sections*/
PFC_INVSQR_Section      : > RAML2         PAGE = 1
```

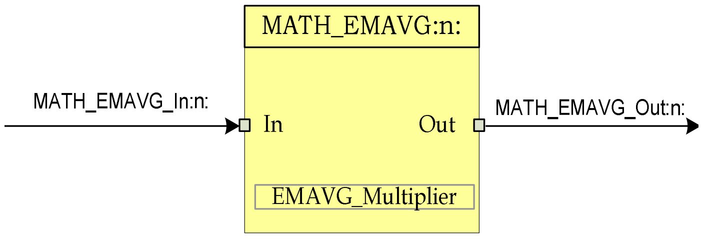
Module Net Definition:

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

5.4 Math Blocks

MATH_EMAVG ***Exponential Moving Average***

Description: This software module performs exponential moving average



Macro File: `MATH_EMAVG.asm`

Technical: This software module performs exponential moving average over data stored in Q24 format, pointed to by `MATH_EMAVG_In:n:` The result is stored in Q24 format at a 32 bit location pointed to by `MATH_EMAVG_Out:n:`

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

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

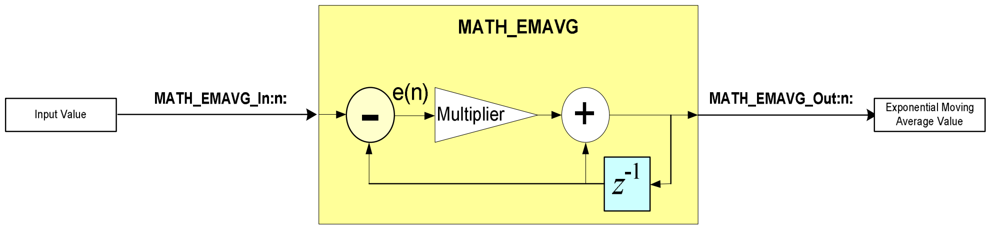
Where $Input(n)$ is the input data at sample instance 'n',
$EMA(n)$ is the exponential moving average at time instance 'n',
$EMA(n-1)$ is the exponential moving average at time instance 'n-1'.
$Multiplier$ is the weighting factor used in exponential moving average

In z-domain the equation can be interpreted as

$$\frac{Output}{Input} = \frac{Multiplier}{1-(1-Multiplier)z^{-1}}$$

This can be seen as a special case for a Low Pass Filter, where pass band gain is equal to $Multiplier$ and filter time constant is $(1-Multiplier)$. Note $Multiplier$ is always ≤ 1, hence $(1-Multiplier)$ is always a positive value. Also the lower the value of $Multiplier$, the larger is the time constant and more sluggish the response of the filter.

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



Usage: The block is used in the PFC software to get the average value of AC Line. The multiplier value for this can be estimated through two methods as follows:

Time Domain: The PFC stage runs at 100Khz and the input AC signal is 60Hz. As the average of the rectified sine signal is desired the effective frequency of the signal being averaged is 120Hz. This implies that (100Khz/120) = 833 samples in one half sine. For the average to be true representation the average needs to be taken over multiple sine halves (note taking average over integral number of sine halves is not necessary). The multiplier value distributes the error equally over the number of samples for which average is taken. Therefore

$$Multiplier = \_IQ30(1/SAMPLE\_No) = \_IQ30(1/3332) = \_IQ30(0.0003)$$

For AC line average a value of 4000 samples is chosen, as it averages roughly over 4 sine halves.

Frequency Domain: Alternatively the multiplier value can be estimated from the z-domain representation as well. The signal is sampled at 100Khz and the frequency content is at 60Hz. Only the DC value is desired, therefore assuming a cut-off frequency of 5Hz the value can be estimated as follows,

For a first order approximation, $z = e^{sT} = 1 + sT_s$

,
where T is the sampling period and solving the equation,

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

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

$$Multiplier = \_IQ30((2 * \pi * f_{cutt\_off}) / f_{sampling}) = \_IQ30(5 * 2 * 3.14 / 100K) = \_IQ30(0.000314)$$

The following steps explain how to include this module into your system

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

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

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

```
// ---------------------------- DPLIB Net Pointers ----------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//MATH_EMAVG - instance #1
extern volatile long *MATH_EMAVG_In1;
extern volatile long *MATH_EMAVG_Out1;
extern volatile long MATH_EMAVG_Multiplier1;
```

Step 3 **Signal net nodes/ variables in C** in the file {ProjectName}-Main.c

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

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

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

volatile long In,Out;
```

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

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

// Digital Power (DP) library initialization
DPL_Init();
// MATH_EMAVG block connections
MATH_EMAVG_In1=&In;
MATH_EMAVG_Out1=&Out;
MATH_EMAVG_Multilpier1=_IQ24(0.0025);

// Initialize the net variables
In=_IQ24(0.0);
Out=_IQ24(0.0)
```

Step 5 **Add** the ISR assembly file "{ProjectName}-DPL-ISR.asm" to the project

Step 6 **Include** the macro's assembly file in the {ProjectName}-DPL-ISR.asm

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

.include "MATH_EMAVG.asm"
```

Step 7 Instantiate the INIT macro in assembly (this is one-time pass through code) inside the C-callable DPL_Init() function which is defined in `{ProjectName}-DPL-ISR.asm`

```
;Macro Specific Initialization Functions
MATH_EMAVG_INIT 1 ; MATH_EMAVG Initialization
```

Step 8 Call the run time macro in assembly inside the C-callable function DPL_ISR() which is the looped or ISR code. The function is defined in `{ProjectName}-DPL-ISR.asm`

```
;"Call" the Run macro
MATH_EMAVG_INIT 1 ; Run the MATH_EMAVG Macro
```

Step 9 Include the memory sections in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD`. Note that the net pointers and the internal data are forced to be placed in a single data page by use of the `.usect` directive in the source file of the module.

```
/*MATH_EMAVG sections*/
MATH_EMAVG_Section      : > RAML2         PAGE = 1
```

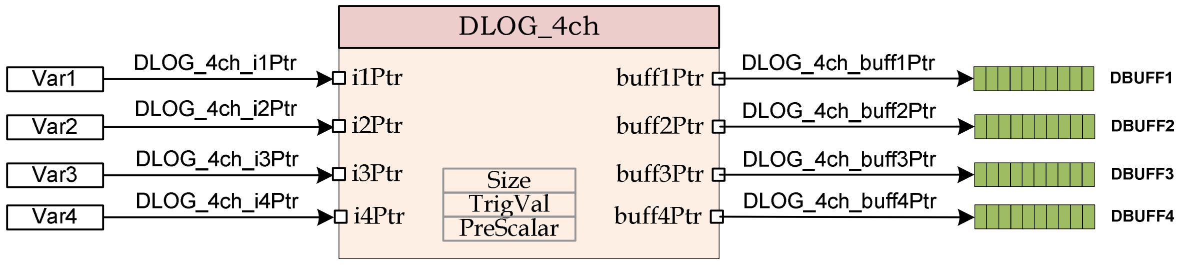
Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
MATH_EMAVG_In:n:	Input Pointer	Pointer to 32 bit fixed input data location storing the data that needs to averaged	Q24: [0, 1)
MATH_EMAVG_Out:n:	Output Pointer	Pointer to 32 bit fixed output data location where the computed average is stored	Q24: [0, 1)
MATH_EMAVG_Multiplier:n:	Internal Data	Data Variable storing the weighing factor for the exponential average.	Q30:[0,1)

5.5 Utilities

DLOG_4ch ***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.asm`

Technical: This software module performs data logging over data stored in Q24 format, pointed to by four pointers `DLOG_4ch_i1Ptr`, `DLOG_4ch_i2Ptr`, `DLOG_4ch_i3Ptr` and `DLOG_4ch_i4Ptr`. The input variable value is then scaled to Q15 format and stored in arrays pointed to by `DLOG_4ch_buff1Ptr`, `DLOG_4ch_buff2Ptr`, `DLOG_4ch_buff3Ptr` and `DLOG_4ch_buff4Ptr`.

The data logger is triggered at the positive edge of the value pointed by the pointer `DLOG_4ch_i1Ptr`. The trigger value is programmable by writing the Q24 trigger value to the module variable `DLOG_4ch_TrigVal`.

The size of the data logger has to be specified using the `DLOG_4ch_Size` module variable.

The module can be configured to log data every n number module call, by specifying a scalar value in variable `DLOG_4ch_PreScalar`. The following example illustrates how best these values be chose using a PFC algorithm example.

Usage: When using the DLOG module for observing system variables in the PFC algorithm it is desirable to observe the logged variable over a multiple line AC period to verify working of the algorithm. 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 words array for each buffer.

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

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

The trigger Value is used to trigger the logging of data at a positive edge around the trigger value of the data pointed to by `DLOG_4CH_iPtr`.

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

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

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

```
// ---------------------------- DPLIB Net Pointers ----------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//DLOG_4ch - instance #1
extern volatile long *DLOG_4ch_i1Ptr;
extern volatile long *DLOG_4ch_i2Ptr;
extern volatile long *DLOG_4ch_i3Ptr;
extern volatile long *DLOG_4ch_i4Ptr;
extern volatile int16 *DLOG_4ch_buff1Ptr;
extern volatile int16 *DLOG_4ch_buff2Ptr;
extern volatile int16 *DLOG_4ch_buff3Ptr;
extern volatile int16 *DLOG_4ch_buff4Ptr;
extern volatile long  DLOG_4ch_TrigVal;
extern volatile int16 DLOG_4ch_PreScalar;
extern volatile int16 DLOG_4ch_Size;
```

Step 3 Declare signal net nodes/ variables in C in the file {ProjectName}-Main.cc Note that the DLOG_SIZE is a #define in the {ProjectName}-Settings.h and can be modified if needed.

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

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

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

#pragma DATA_SECTION(DBUFF1,"DLOG_BUFF");
#pragma DATA_SECTION(DBUFF2,"DLOG_BUFF");
#pragma DATA_SECTION(DBUFF3,"DLOG_BUFF");
#pragma DATA_SECTION(DBUFF4,"DLOG_BUFF");
volatile int16 DBUFF1[DLOG_SIZE];
volatile int16 DBUFF2[DLOG_SIZE];
volatile int16 DBUFF3[DLOG_SIZE];
volatile int16 DBUFF4[DLOG_SIZE];
```

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

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

// Digital Power (DP) library initialization
DPL_Init();
// DLOG block connections
// Store address of the system variables that need to be logged
// In the example below PFC algorithm variables are logged
DLOG_4ch_i1Ptr =&VacLineRect;
DLOG_4ch_i2Ptr =&InvAvgSqr;
DLOG_4ch_i3Ptr =&PFCIcmd;
DLOG_4ch_i4Ptr =&VacLineAvg;
// Point the BuffPtr to the buffer location
DLOG_4ch_buff1Ptr =DBUFF1;
DLOG_4ch_buff2Ptr =DBUFF2;
DLOG_4ch_buff3Ptr =DBUFF3;
DLOG_4ch_buff4Ptr =DBUFF4;
// Setup Size, Trigger Value  and Pre Scalar
DLOG_4ch_TrigVal = _IQ(0.1);
DLOG_4ch_PreScalar = 25;
DLOG_4ch_Size=DLOG_SIZE;

// Zero the buffers
DLOG_4ch_BuffInit(DBUFF1, DLOG_SIZE);
DLOG_4ch_BuffInit(DBUFF2, DLOG_SIZE);
DLOG_4ch_BuffInit(DBUFF3, DLOG_SIZE);
DLOG_4ch_BuffInit(DBUFF4, DLOG_SIZE);
```

Step 5 **Add** the ISR assembly file "`{ProjectName}-DPL-ISR.asm`" to the project

Step 6 **Include** the macro's assembly file in the `{ProjectName}-DPL-ISR.asm`

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

.include "DLOG_4ch.asm"
```

Step 7 **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-callable DPL_Init() function which is defined in `{ProjectName}-DPL-ISR.asm`

```
;Macro Specific Initialization Functions
DLOG_4ch_INIT 1    ; DLOG_4CH Initialization
```

Step 8 **Call the** run time macro in assembly inside the C-callable function DPL_ISR() which is the looped or ISR code. The function is defined in `{ProjectName}-DPL-ISR.asm`

```
;"Call" the Run macro
DLOG_4ch 1   ; Run the DLOG_4CH Macro
```

Step 9 **Include the memory sections** in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD`. Note that the net pointers and the internal data are forced to be placed in a single data page by use of the `.usect` directive in the source file of the module.

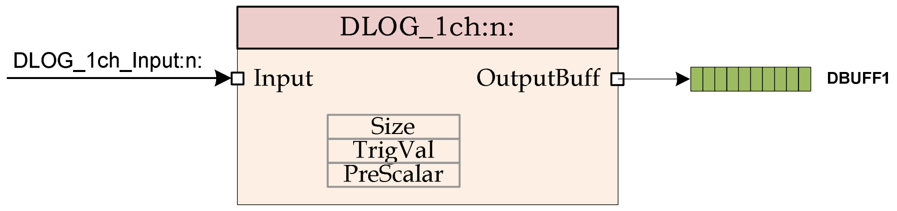
```
/*DLOG_4ch sections*/
DLOG_4ch_Section  : > dataRAM       PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
DLOG_4ch_i1Ptr, DLOG_4ch_i2Ptr DLOG_4ch_i3Ptr DLOG_4ch_i4Ptr	Input Pointers	Pointer to 32 bit input data location storing the data that needs to logged.	Q24: [0, 1)
DLOG_4ch_buff1Ptr, DLOG_4ch_buff2Ptr DLOG_4ch_buff3Ptr DLOG_4ch_buff4Ptr	Output Pointer	Pointer to 16 bit output data buffer location where the logged data is stored after scaling input data from Q24 to Q15.	Q15: [0, 1)
DLOG_4ch_Size	Internal Data	16 bit integer data storing the buffer size being used by the DLOG module	Q0
DLOG_4ch_PreScalar	Internal Data	16 bit integer data storing the pre scalar value	Q0
DLOG_4ch_TrigVal	Internal Data	Q24 data variable storing the trigger value for the DLOG module.	Q24:[0,1)

DLOG_1ch — *Single Channel Data Logger*

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



Macro File: `DLOG_1ch.asm`

Technical: This software module performs data logging over data stored in Q24 format, pointed to by pointer `DLOG_1ch_Input:n:` The input variable value is then scaled to Q15 format and stored in the array pointed to by `DLOG_1ch_OutputBuff.`

The data logger is triggered at the positive edge of the value pointed by the pointer `DLOG_1ch_Input:n:` The trigger value is programmable by writing the Q24 trigger value to the module variable `DLOG_1ch_TrigVal:n:`

The size of the data logger has to be specified using the `DLOG_1ch_Size:n:` module variable.

The module can be configured to log data every n number module call, by specifying a scalar value in variable `DLOG_1ch_PreScalar:n:`

`The value of the prescalar can be chosen`

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

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

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

```
// ---------------------------- DPLIB Net Pointers ----------------------
// Declare net pointers that are used to connect the DP Lib Macros here
// and the data variables being used by the macros internally
//DLOG_1ch - instance #1
extern volatile long  *DLOG_1ch_Input1;
extern volatile int16 *DLOG_1ch_OutputBuff1;
extern volatile long  DLOG_1ch_TrigVal1;
extern volatile int16 DLOG_1ch_PreScalar1;
extern volatile int16 DLOG_1ch_Size1;
```

`Step 3` **Declare signal net nodes/ variables in C** in the file `{ProjectName}-Main.c.` Note that the DLOG_SIZE is a #define in the {ProjectName}-Settings.h and can be modified if needed.

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

```
// ---------------------------- DPLIB Variables ------------------------
// Declare the net variables being used by the DP Lib Macro here
#pragma DATA_SECTION(DBUFF,"DLOG_BUFF");
volatile int16 DBUFF[DLOG_SIZE];
```

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

```
//----------Connect the macros to build a system-------------------
// Digital Power (DP) library initialization
DPL_Init();
// DLOG block connections
// Store address of the system variables that need to be logged
DLOG_1ch_Input1 =&VacLineRect;
// Point the BuffPtr to the buffer location
DLOG_1ch_OutputBuff1 =DBUFF;

// Setup Size, Trigger Value  and Pre Scalar
DLOG_1ch_TrigVal1 = _IQ(0.1);
DLOG_1ch_PreScalar1 = 25;
DLOG_1ch_Size1=DLOG_SIZE;

// Zero the buffers
DLOG_BuffInit(DBUFF, DLOG_SIZE);
```

`Step 5` **Add** the ISR assembly file "`{ProjectName}-DPL-ISR.asm`" to the project

`Step 6` **Include** the macro's assembly file in the `{ProjectName}-DPL-ISR.asm`

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

.include "DLOG_1ch.asm"
```

`Step 7` **Instantiate the INIT macro** in assembly (this is one-time pass through code) inside the C-callable DPL_Init() function which is defined in`{ProjectName}-DPL-ISR.asm`

```
;Macro Specific Initialization Functions
DLOG_1ch_INIT 1    ; DLOG_1CH Initialization
```

`Step 8` **Call** run time macro in assembly inside the C-callable function DPL_ISR() which is the looped or ISR code. The function is defined in`{ProjectName}-DPL-ISR.asm`

```
;"Call" the Run macro
DLOG_1ch 1
```

Step 9 Include the memory sections in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD`. Note that the net pointers and the internal data are forced to be placed in a single data page by use of the `.usect` directive in the source file of the module.

```
/*DLOG_1ch sections*/
DLOG_1ch_Section   : > dataRAM       PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
DLOG_1ch_Input:n:	Input Pointer	Pointer to 32 bit input data location storing the data that needs to logged.	Q24: [0, 1)
DLOG_1ch_OutputBuff:n:	Output Pointer	Pointer to 16 bit output data buffer location where the logged data is stored after scaling input data from Q24 to Q15.	Q15: [0, 1)
DLOG_1ch_Size:n:	Internal Data	16 bit integer data storing the buffer size being used by the DLOG module	Q0
DLOG_1ch_PreScalar:n:	Internal Data	16 bit integer data storing the pre scalar value	Q0
DLOG_1ch_TrigVal:n:	Internal Data	Q24 data variable storing the trigger value for the DLOG module.	Q24:[0,1)

Chapter 6. Revision History

Version	Date	Notes
V1.0	---	Original release of DP library modules for F280x platform.
V2.0	July 6, 2010	Major release of library to support Piccolo platform. Changed net node format to 32-bits.