C28x Digital Power Library

V1.0

Nov-11

Module User’s Guide

C28x Foundation Software

IMPORTANT NOTICE

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

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

Customers are responsible for their applications using TI components.

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

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

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

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

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

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

Copyright ©2011, Texas Instruments Incorporated

Trademarks

TMS320 is the trademark of Texas Instruments Incorporated.
All other trademarks mentioned herein are property of their respective companies

Acronyms

DPLib: Digital Power library functions.

C28x: Refers to devices with the C28x CPU core.

Float referrers to IEEE single precision floating point numbers.

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.

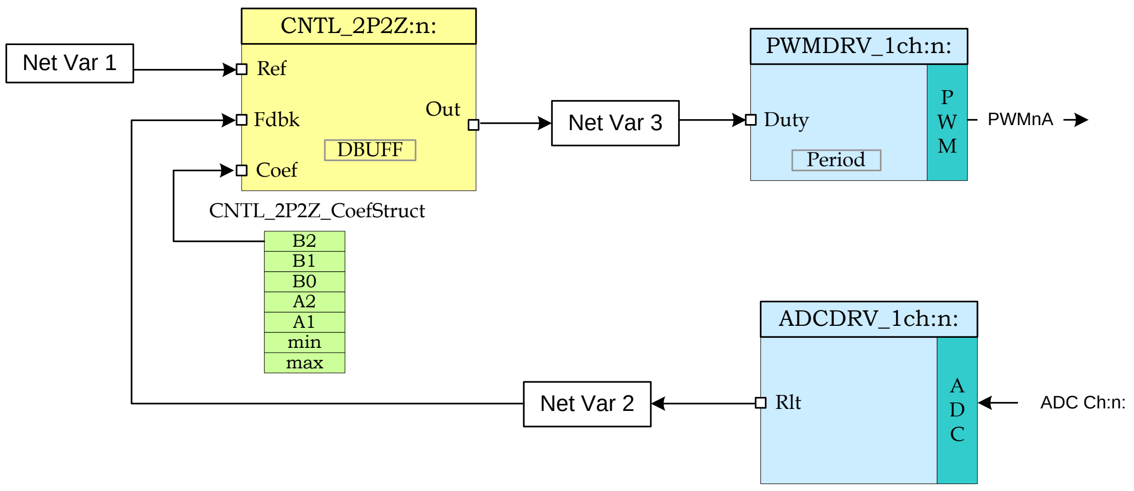


Figure 1 Close Loop System using DPLib

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`"

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

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:

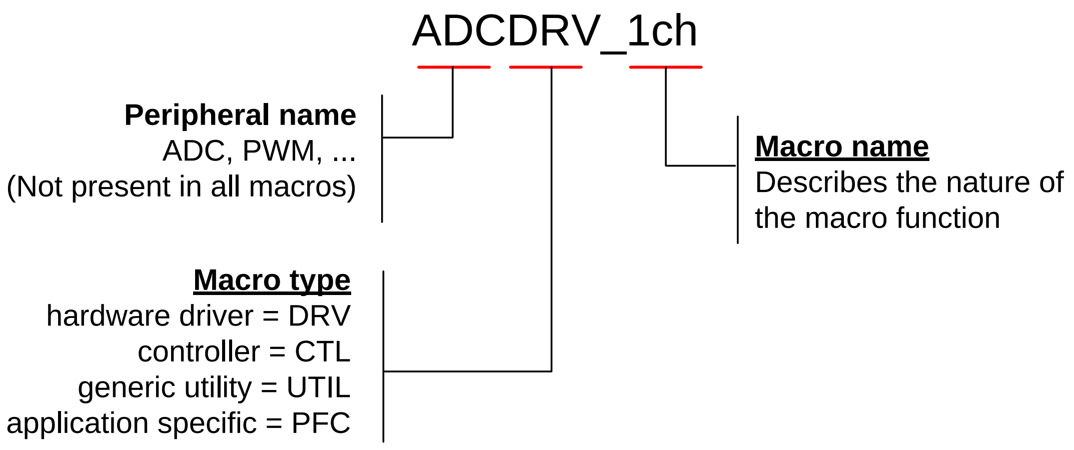


Figure 2 – Function Naming Convention

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.

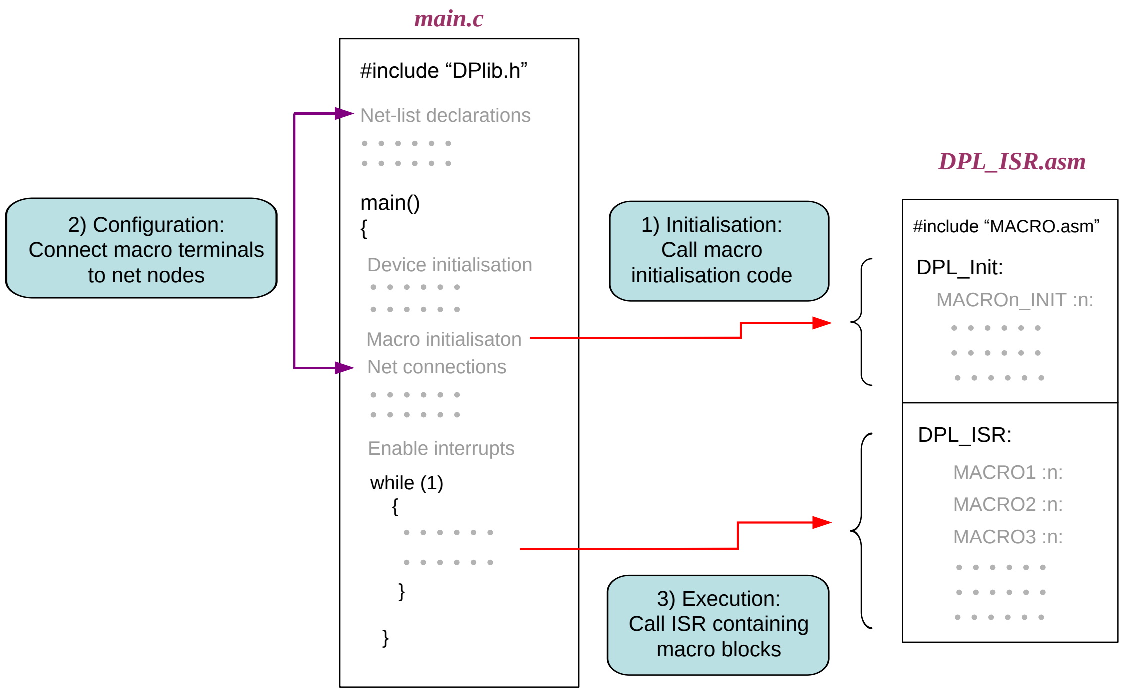


Figure 3 : Relation between Main.c & ISR.asm file

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.

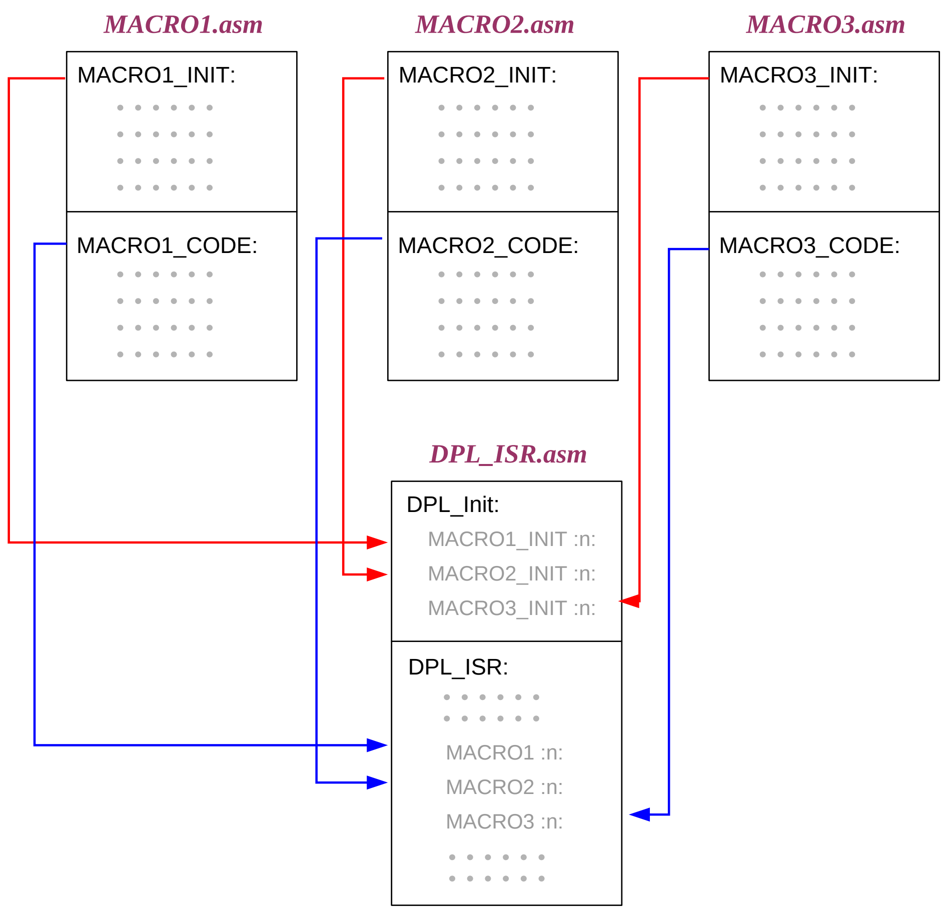


Figure 4 DP library assembly ISR and macro file

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 float   *CNTL_2P2Z_Ref1;
extern volatile float   *CNTL_2P2Z_Fdbk1;
extern volatile float   *CNTL_2P2Z_Out1;
extern volatile float   *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 float    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.

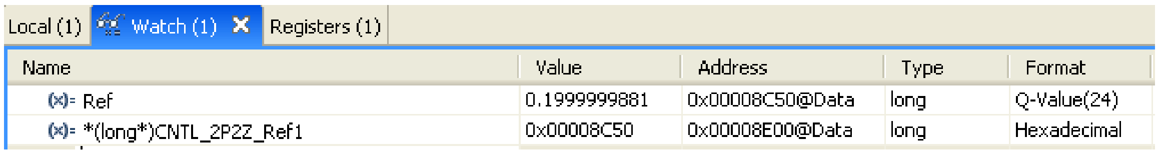
This DPLib Macros need to be placed in the data RAM. The sample linker file specifies where each memory section from each DP library module would be placed in internal memory. An example of section placement for the CNTL_2P2Z module is shown below.

```
  /* 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 library variables in watch window

If is desired to see the DP 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.


Local (1) | Watch (1) | Registers (1)

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**	**Multiple Instance Support**
CNTL_2P2Z	CNTL	Second Order Control Law	NA	35	Yes
ADCDRV_1ch	HW	Single Channel ADC Driver	Yes	6	Yes
ADCDRV_4ch	HW	Four Channel ADC Driver	Yes	18	No
PWMDRV_1ch	HW	Single Channel PWM Driver	Yes	10	Yes
PWMDRV_1ch_ UpDwnCnt	HW	Single channel driver with up down modulation	Yes	10	Yes
PWMDRV_1ch_ UpDwnCntCompl	HW	Single channel driver with up down modulation centered around period	Yes	12	Yes
MATH_EMAVG	MATH	Exponential moving average	NA	13	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…

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

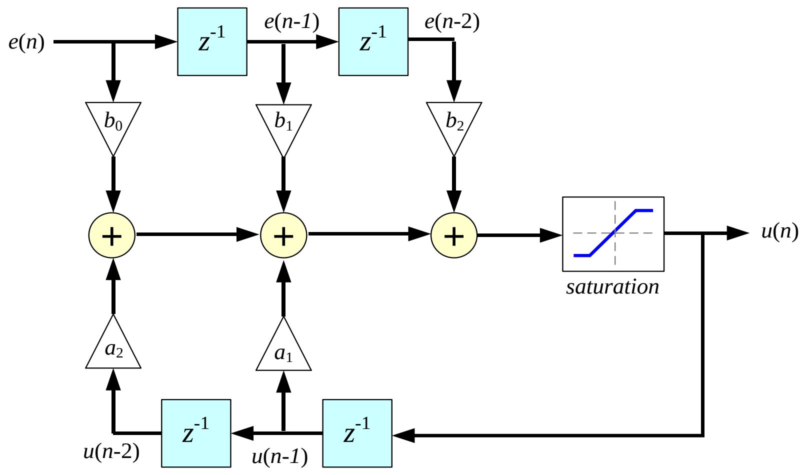
This may be expressed in difference equation form as:

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

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`.

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

b_2
b_1
b_0
a_2
a_1
sat_{max}
sat_{min}

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

Controller coefficients must be initialized before the controller is used. A structure `CNTL_2P2Z_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 float   *CNTL_2P2Z_Ref1;
extern volatile float   *CNTL_2P2Z_Out1;
extern volatile float   *CNTL_2P2Z_Fdbk1;
extern volatile float   *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

float 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 = 0.05;
CNTL_2P2Z_CoefStruct1.b1 = -0.20;
CNTL_2P2Z_CoefStruct1.b0 = 0.20;
CNTL_2P2Z_CoefStruct1.a2 = 0.0;
CNTL_2P2Z_CoefStruct1.a1 = 1.0;
CNTL_2P2Z_CoefStruct1.max =0.7;
CNTL_2P2Z_CoefStruct1.min =0.0;

//Initialize the net Variables/nodes
Ref=0.0;
Fdbk=0.0;
Out=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 floating point input data location storing the Reference value for the controller.	Float:[0,1)
CNTL_2P2Z_Fdbk:n:	Input Pointer	Pointer to floating point input data location storing the Feedback value for the controller.	Float:[0,1)
CNTL_2P2Z_Coef:n:	Input Pointer	Pointer to the location where coefficient structure is stored.	Float:[0,1)
CNTL_2P2Z_Out:n:	Output Pointer	Pointer to float output location where the output of the module is stored	Float:[0,1)
CNTL_2P2Z_DBUFF:n:	Internal Data	Data Variable storing the scaling factor	Float:[0,1)

5.2. Peripheral Configuration

ADC_Cnf — *ADC Configuration*

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

Peripheral Initialization File: `ADC_Cnf.c`

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

```
void ADC_Cnf(int AdcNum, int ChSel[], int Trigsel[], int
    ACQPS[])
```

where

AdcNum	Is the number of the ADC peripheral on the device (for devices that have multiple ADC's)
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

The file does not configure any interrupts on the ADC channel and these need to be done outside of this configuration routine.

Usage:

On F28M35x the C28x interacts with the on chip ADC through the ACIB (Refer to the F28M35x Technical Reference Manual SPRUH22). As the control peripherals are on the C28x these triggers need to be transferred across to the analog subsystem. Therefore the peripheral trigger from the control subsystem i.e. the C28x are muxed to generate a set of ACIB triggers which in turn can be configured to generate the SOC for the ADC channels. The selection of the peripheral trigger to CIB trigger is done using the following selection:

```
// Configure the CIB triggers (Note this needs to be done
// before the ADC SOC trigger selection)
EALLOW;
// EPWM3SOCA to TRIGGER 1 of the analog subsystem
```

```
AnalogSysctrlRegs.TRIG1SEL.all = ADCTRIG_EPWM3_SOCA;
// EPWM1SOCA to Trigger 2 of the analog subsystem
AnalogSysctrlRegs.TRIG2SEL.all = ADCTRIG_EPWM1_SOCA;
EDIS;
```

Now the appropriate channel selection can be made in the Chsel array and an ACIB trigger associated to that channel. The below example configures the ADC wrapper to start of conversion on pin A4 and store in channel 0 registers, which are triggered every ACIB trigger 1. After configuring these arrays the ADC_CNF function can be called to configure the appropriate ADC registers.

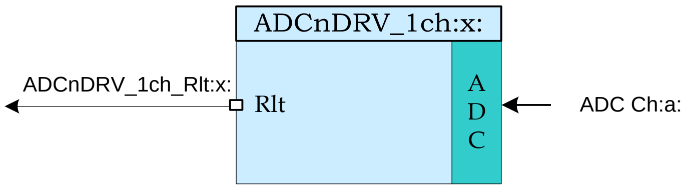
```
ChSel1[0] = 4;    //ADC1 SOC 0  -> ADC1-A4
TrigSel1[0]= 5;  //ACIB trigger 1 selection is number 5

ADC_CNF(1, ChSel1, TrigSel1, ACQPS1);
```

5.3. Peripheral Drivers

ADCnDRV_1ch *ADC Driver Single Channel*

Description: This assembly macro reads a result from the internal ADCn module Result Register:x: and delivers it in scaled float format to the output terminal, where :n: is the ADC number and :x: 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: `ADC1DRV_1ch.asm , ADC2DRV_1ch.asm`

Peripheral Initialization File: `ADC_CNF.c`

Description: The ADC module in the F28M35x device includes a ratio-metric ADC that converts the ADC value such that it generates a result value of 12-bits resolution. The ADCDRV macro reads one pre-defined result register (determined by the instance number of the macro i.e. instance 0 reads Adc:n:Result.ADCRESULT0 and instance 5 reads Adc:n:Result.ADCRESULT5) . The module then scales this to float format and writes the result to the output net terminal.

This macro is used in conjunction with the peripheral configuration file ADC_Cnf.c

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
//ADC1DRV_1ch - instance #1
extern volatile float *ADC1DRV_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 float Out;
```

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

```
// Configure the CIB triggers (Note this needs to be done // before the
ADC SOC trigger selection)
EALLOW;
// EPWM3SOCA to TRIGGER 1 of the analog subsystem
AnalogSysctrlRegs.TRIG1SEL.all = ADCTRIG_EPWM3_SOCA;
EDIS;
/*The below example configures the ADC wrapper to start of conversion
on pin A4 and store in channel 0 registers, which are triggered every
ACIB trigger 1. After configuring these arrays the ADC_CNF function can
be called to configure the appropriate ADC registers.*/
ChSel1[0] = 4;    //ADC1 SOC 0  -> ADC1-A4
TrigSel1[0]= 5;  //ACIB trigger 1 selection is number 5

ADC_CNF(1, ChSel1, TrigSel1, ACQPS1);
```

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
ADC1DRV_1ch_Rlt1=&Out;
// Initialize the net variables
Out=(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 "ADC1DRV_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
ADC1DRV_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
ADC1DRV_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.

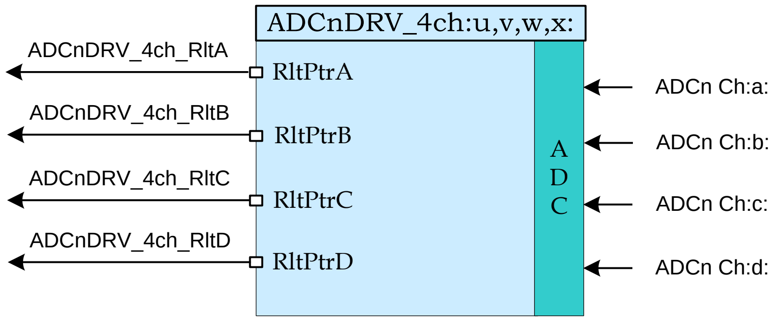
```
/*ADCDRV_1ch sections*/
ADCDRV_1ch_Section      : > dataRAM       PAGE = 1
```

Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
ADCnDRV_1ch_Rlt:n:	Output Pointer	Pointer to 32 bit floating point data location storing the result of the module.	Float:[0, 1)

ADCnDRV_4ch *ADCn Driver Four Channel*

Description: This assembly macro reads four results from the internal ADC module result registers u,v,w,x and delivers them in scaled floating point 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: `ADC1DRV_4ch.asm, ADC2DRV_4ch.asm`

Peripheral Initialization File: `ADC_Cnf.c`

Description: The ADC module in the F28M35x device includes a ratio-metric ADC that converts the ADC value such that it generates a result value of 12-bits resolution. The ADCDRV macro reads four pre-defined result register (determined by the parameters passed to the macro i.e. if 1,4,5,6 are passed to ADC1DRV_4ch macro then Adc:n:Result.ADCRESULT1,4,5,6 are read and scaled, with results stored in the output pointer.

This macro is used in conjunction with the peripheral configuration file ADC_Cnf.c

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
//ADC1DRV_4ch
extern volatile float *ADC1DRV_4ch_RltA;
extern volatile float *ADC1DRV_4ch_RltB;
extern volatile float *ADC1DRV_4ch_RltC;
extern volatile float *ADC1DRV_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 float RltA,RltB,RltC,RltD;
```

`Step 4` **Call the peripheral configuration function** `ADC_CNF(int AdcNum, int ChSel[],int TrigSel[],int ACQPS[])` in `{ProjectName}-Main.c`, this function is defined in `ADC_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. */

/* 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 */

// Configure the CIB triggers (Note this needs to be done // before the
ADC SOC trigger selection)
EALLOW;
// EPWM1SOCA to TRIGGER 1 of the analog subsystem
AnalogSysctrlRegs.TRIG1SEL.all = ADCTRIG_EPWM1_SOCA;
// EPWM5SOCA to TRIGGER 2 of the analog subsystem
AnalogSysctrlRegs.TRIG2SEL.all = ADCTRIG_EPWM5_SOCA;
EDIS;

// Specify ADC Channel – pin Selection for Configuring the ADC
ChSel1[0] = 13;          // ADC B5
ChSel1[1] = 3;           // ADC A3
ChSel1[2] = 7;           // ADC A7
ChSel1[5] = 10;          // ADC B2

// Specify the Conversion Trigger for each channel
TrigSel[0]= 5;
TrigSel[1]= 6;
TrigSel[2]= 5;
TrigSel[5]= 6;


ADC_CNF(1, ChSel1, TrigSel1, ACQPS1);
```

`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
ADC1DRV_4ch_RltA=&RltA;
ADC1DRV_4ch_RltB=&RltB;
ADC1DRV_4ch_RltC=&RltC;
ADC1DRV_4ch_RltD=&RltD;

// Initialize the net variables
RltA=(0.0);
RltB=(0.0);
RltC=(0.0);
RltD=(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 "ADC1DRV_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

```
Adc1Result.ADCRESULT0 -> (Scale) -> *(ADC1DRV_4ch_RltA)

Adc1Result.ADCRESULT5 -> (Scale) -> *(ADC1DRV_4ch_RltB)

Adc1Result.ADCRESULT2 -> (Scale) -> *(ADC1DRV_4ch_RltC)

Adc1Result.ADCSESULT1 -> (Scale) -> *(ADC1DRV_4ch_RltD)
```

```
;Macro Specific Initialization Functions
ADC1DRV_4ch_INIT 0,5,2,1      ; ADC1DRV_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
ADC1DRV_4ch 0,5,2,1     ; Run ADCDRV_4ch
```

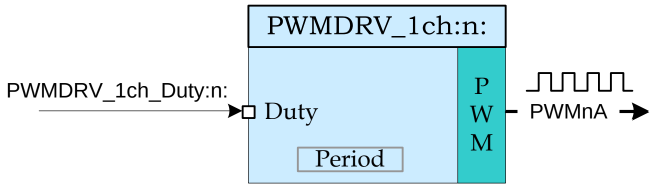
Step 10 **Include the memory sections** in the `{DeviceName}-{RAM/FLASH}-{ProjectName}.CMD`. Note that the net pointers and the internal data are forced to be placed in a single data page by use of the `.usect` directive in the source file of the module.

```
/*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**
ADCnDRV_4ch_RltA	Output Pointer	Pointer to 32 bit floating point data location storing the result of the module.	Float: [0, 1)
ADCnDRV_4ch_RltB	Output Pointer	Pointer to 32 bit floating point data location storing the result of the module.	Float: [0, 1)
ADCnDRV_4ch_RltC	Output Pointer	Pointer to 32 bit floating point data location storing the result of the module.	Float: [0, 1)
ADCnDRV_4ch_RltD	Output Pointer	Pointer to 32 bit floating point data location storing the result of the module.	Float: [0, 1)

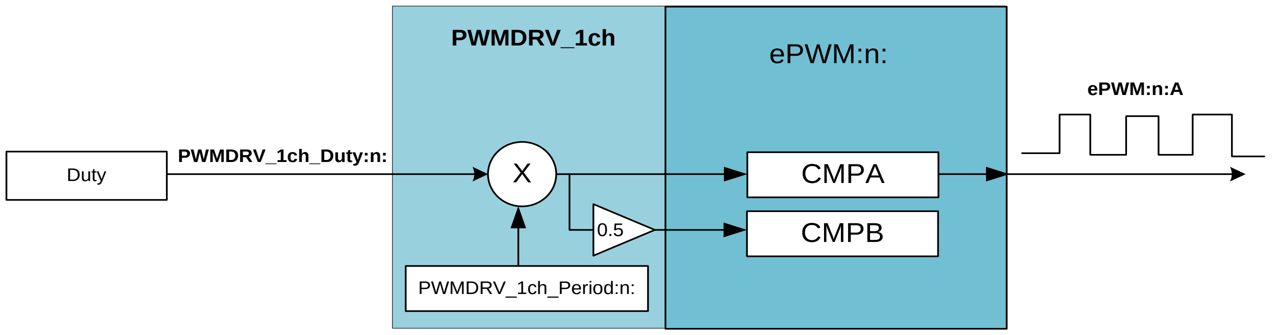
Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a duty on PWM channel A, dependent on the value of the input variable.



Macro File: `PWMDRV_1ch.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 floating duty input, pointed to by the net pointer PWMDRV_1ch_Duty:n: into an unsigned 16 bit integer 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.

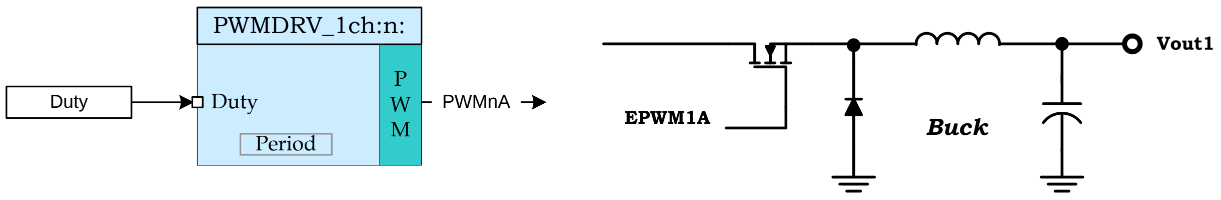
Mode =1 PWM configured as a master
Mode = 0 PWM configured as slave

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.

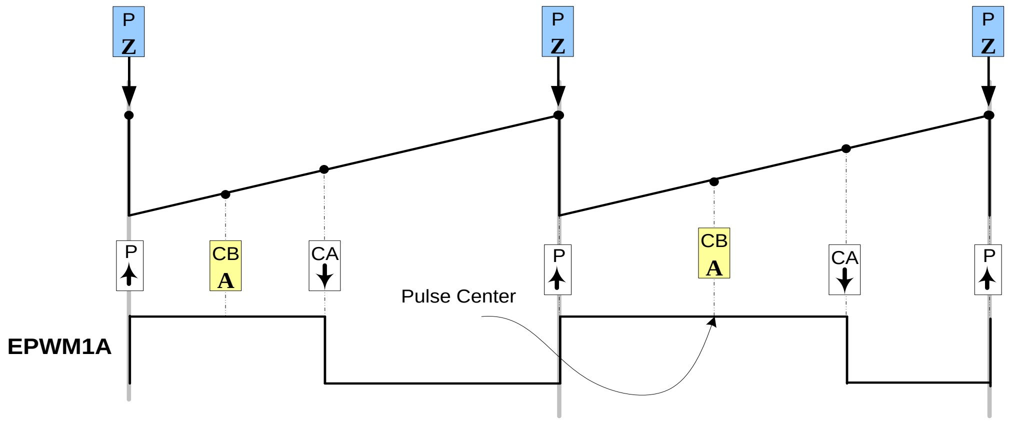
The function configures the PWM peripheral in up-count mode. The figure below, shows with help of a timing diagram how the PWM is configured to generate the waveform.

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. Note that the subtract 1 from the period value, to take into account the up count mode is done by the CNF function. Hence value of 600 needs to be supplied to the function.



Buck converter driven by PWMDRV_1ch module



PWM generation with the EPWM module.

Usage:

`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 float *PWMDRV_1ch_Duty1;
extern volatile float 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 float 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_Duty1=&Duty;
// Initialize the net variables
Duty=(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.

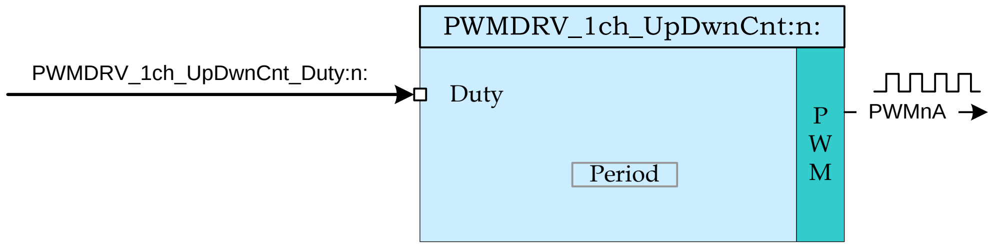
```
/*PWMDRV_1ch sections*/
PWMDRV_1ch_Section      : > dataRAM       PAGE = 1
```

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_Duty:n:	Input Pointer	Pointer to 32 bit floating point input data location storing Duty Value	Float: [0, 1)

PWMDRV_1ch_UpDwnCnt — *PWM Driver, Duty on chA centered at zero*

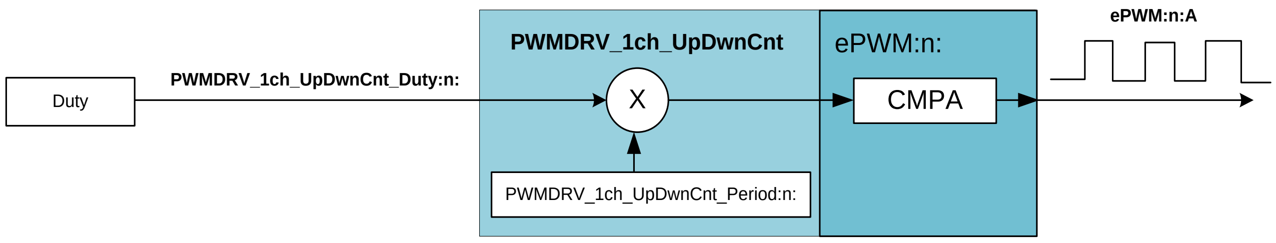
Description: This hardware driver module, when used in conjunction with the corresponding PWM configuration file, drives a duty on PWM channel A dependent on the value of the input variable DutyA. The waveform is centered at the zero value of the up down count time base of the PWM.



Macro File: `PWMDRV_1ch_UpDwnCnt.asm`

Peripheral Initialization File: `PWM_1ch_UpDwnCnt_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 floating point input, pointed to by the Net Pointer PWMDRV_1ch_UpDwnCnt_Duty:n:A into an unsigned Q0 number scaled by the PWM period value, and stores this value in the EPwmRegs:n:.CMPA such as to give duty cycle control on channel A of the PWM module.



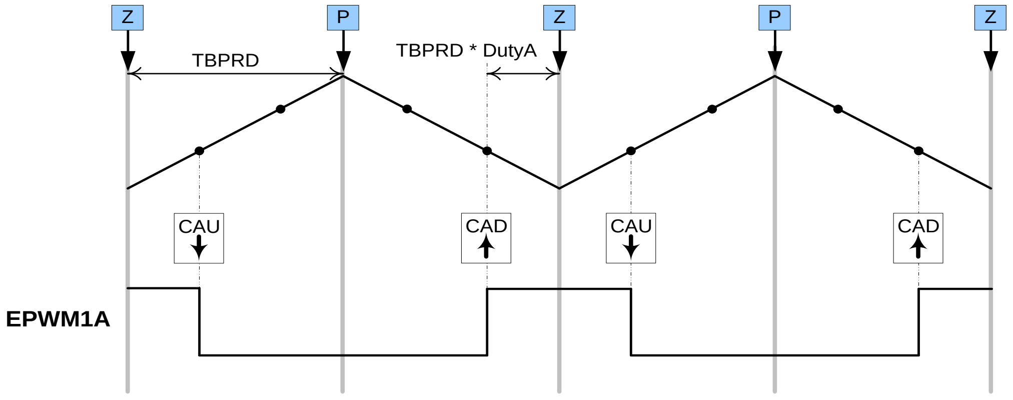
This macro is used in conjunction with the peripheral configuration file PWM_1ch_UpDwnCnt_Cnf.c. The file defines the function

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

where

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

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



PWM generation with the EPWM module.

Usage:

`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_UpDwnCnt - instance #1
extern volatile float *PWMDRV_1ch_UpDwnCnt_Duty1;
extern volatile float PWMDRV_1ch_UpDwnCnt_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 float Duty;
```

`Step 4` **Call the Peripheral configuration function** `PWM_1ch_UpDwnCnt_CNF(int16 n, int16 period, int16 mode, int16 phase)` in `{ProjectName}-Main.c`, this function is defined in `PWM_ 1ch_UpDwnCnt_Cnf.c`. This file must be included manually into the project.

```
// Configure PWM1 for switching frequency 100Khz, @60Mhz CPU clock =>
period = (60Mhz/100Khz) = 600

PWM_1ch_UpDwnCnt_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_UpDwnCnt_INIT` function. This function initializes the value of `PWMDRV_1ch_UpDwnCnt_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_UpDwnCnt block connections
PWMDRV_1ch_UpDwnCnt_Duty1=&Duty;

// Initialize the net variables
Duty=(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_UpDwnCnt.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_UpDwnCnt_INIT 1     ; PWMDRV_1ch_UpDwnCnt 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_UpDwnCnt 1    ; Run PWMDRV_1ch_UpDwnCnt
```

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.

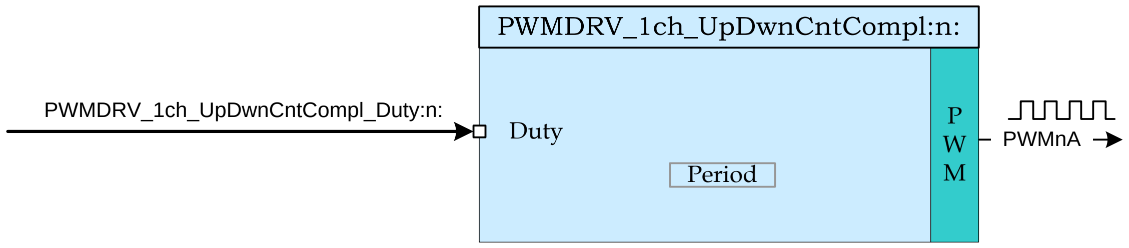
```
/*PWMDRV_1ch_UpDwnCnt sections*/
PWMDRV_1ch_UpDwnCnt_Section    : > dataRAM       PAGE = 1
```

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_UpDwnCnt_Duty:n:A	Input Pointer	Pointer to 32 bit floating point input data location storing DutyA Value	Float: [0, 1)

PWMDRV_1ch_UpDwnCntCompl — *PWM Driver, Duty on chA centered at period*

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



Macro File: `PWMDRV_1ch_UpDwnCntCompl.asm`

Peripheral Initialization File: `PWM_1ch_UpDwnCntCompl_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 float input, pointed to by the Net Pointer PWMDRV_1ch_UpDwnCntCompl_Duty:n: into an unsigned 16 bit integer number scaled by the PWM period value, and stores this value in the EPwmRegs:n:.CMPA such as to give duty cycle control on channel A of the PWM module.

```
EPwmRegs:n:.CMPA=Period-Period*Duty
```

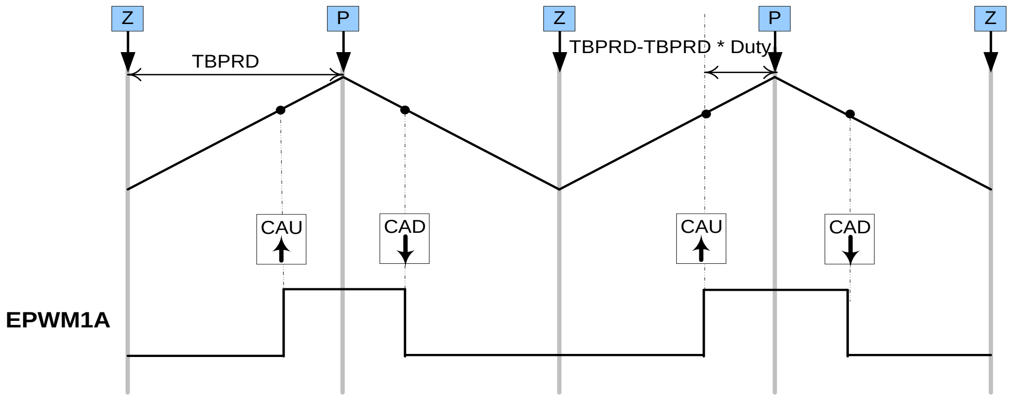
This macro is used in conjunction with the peripheral configuration file PWM_1ch_UpDwnCntCompl_Cnf.c. The file defines the function

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

where

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

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



PWM generation with the EPWM module.

Usage:

`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_UpDwnCntCompl - instance #1
extern volatile float *PWMDRV_1ch_UpDwnCntCompl_Duty1;
extern volatile float PWMDRV_1ch_UpDwnCntCompl_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 float Duty;
```

`Step 4` Call the Peripheral configuration function `PWM_1ch_UpDwnCntCompl_CNF(int16 n, int16 period, int16 mode, int16 phase)` in `{ProjectName}-Main.c`, this function is defined in `PWM_ 1ch_UpDwnCntCompl_Cnf.c`. This file must be included manually into the project.

```
// Configure PWM1 for switching frequency 100Khz, @60Mhz CPU clock =>
period = (60Mhz/100Khz) = 600

PWM_1ch_UpDwnCntCompl_CNF(1,600);
```

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_UpDwnCntCompl_INIT` function. This function initializes the value of `PWMDRV_1ch_UpDwnCntCompl_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_UpDwnCnt block connections
PWMDRV_1ch_UpDwnCntCompl_Duty1=&Duty;

// Initialize the net variables
Duty=(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_UpDwnCntCompl.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_UpDwnCntCompl_INIT 1 ;PWMDRV_1ch_UpDwnCntCompl
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_UpDwnCntCompl 1     ; Run PWMDRV_1ch_UpDwnCntCompl
```

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_UpDwnCntCompl sections*/
PWMDRV_1ch_UpDwnCntCompl_Section     : > dataRAM       PAGE = 1
```

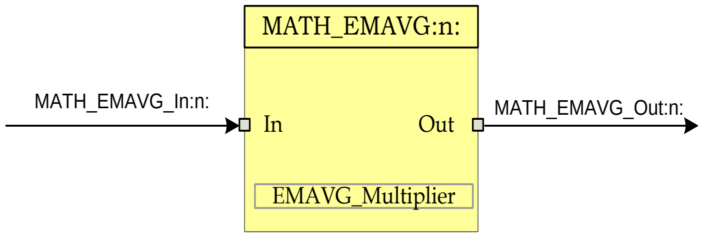
Module Net Definition:

Net name (:n: is the instance number)	**Description**	**Format**	**Acceptable Range of Variable or of the Variable being pointed to**
PWMDRV_1ch_UpDwnCntCompl_Duty:n:A	Input Pointer	Pointer to 32 bit floating point input data location storing DutyA Value	Float: [0, 1)

5.5 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 floating point format, pointed to by `MATH_EMAVG_In:n:` The result is stored in 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'.

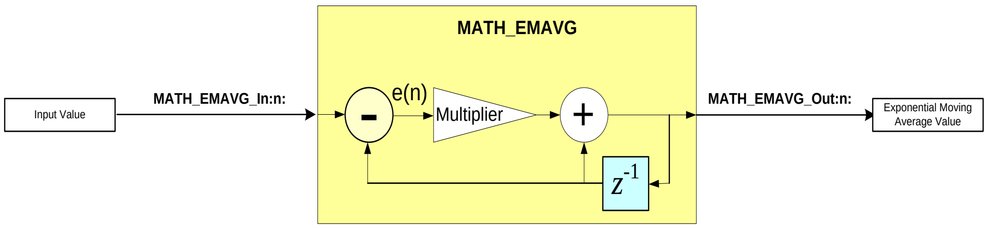
$Multiplier$ is the weighting factor used in exponential moving average

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

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

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

Frequency Domain: Alternatively the multiplier value can be estimated from the z-domain representation 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

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

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 float *MATH_EMAVG_In1;
extern volatile float *MATH_EMAVG_Out1;
extern volatile float 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 float 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=(0.0025);

// Initialize the net variables
In=(0.0);
Out=(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 floating point input data location storing the data that needs to averaged	Float: [0, 1)
MATH_EMAVG_Out:n:	Output Pointer	Pointer to 32 bit floating point output data location where the computed average is stored	Float: [0, 1)
MATH_EMAVG_Multiplier:n:	Internal Data	Data Variable storing the weighing factor for the exponential average.	Float:[0,1)

Chapter 6. Revision History

Version	Date	Notes
V1.0	November 12, 2011	Original release of DP library modules for F28M35x platform in floating point.