## DSP/BIOS PSC Device Driver

# Architecture/Design Document

*Revision History*

| Document Version | Author(s) | Date | Comments |
|---|---|---|---|
| 0.1 | Madhvapathi Sriram | September 18 2008 | Created the document |
| 0.2 | Imtiaz SMA | January 21, 2009 | Updated the document for the IOM driver and IDriver contrast sections. |

## IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

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. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments

Post Office Box 655303

Dallas, Texas 75265

Table of Contents

## List Of Figures

# 1 System Context

The purpose of this document is to explain the device driver design for PSC peripheral using DSP/BIOS operating system running on DSP OMAPL138

**Note:** The usage of structure names and field names used throughout this design document is only for indicative purpose. These names shall not necessarily be matched with the names used in source code.

## 1.1 Terms and Abbreviations

| Term | Description |
|---|---|
| API | Application Programmer's Interface |
| CSL | TI Chip Support Library – primitive h/w abstraction |
| IP | Intellectual Property |
| ISR | Interrupt Service Routine |
| OS | Operating System |
| PSC | Power Sleep Controller |

## 1.1 Disclaimer

This is a design document for the PSC driver for the DSP/BIOS operating system. Although the current design document explain the PSC driver in the context of the BIOS 6.x driver implementation, the driver design still holds good for the BIOS 5.x driver implementation as the BIOS 5.x driver is a direct port of BIOS 6.x driver. The BIOS 5.x drivers conform to the IOM driver model whereas the BIOS 6.x drivers confirm to the IDriver model. The subsequent section explains how this document can be used to understand and modify the IOM drivers found in this product. Please note that all the flowcharts, structures and functions described here in this document are equally applicable to the PSC driver 5.x.

## 1.2 IOM driver Vs IDriver

The following are the main difference between the BIOS 5.x and BIOS 6.x driver. Please refer to the reference documents for more details in the IOM driver model.

1. All the references to the stream module should be treated as references to a module that provides data streaming. In BIOS 5.x the equivalent modules are SIO and GIO.

2. This document refers to the IDriver model supported by the BIOS 6.x.All the references to the IDriver should be assumed to be equivalent to the IOM driver model.

3. The BIOS 6.x driver uses a module specifications file (*.xdc) for the declaration of the enumerations, structures and various constants required by the driver. The equivalent of this xdc file is the header file XXX.h and the XXXLocal.h.

   **Note**: The XXXLocal.h file contains all the declaration specified in the "internal" section of the corresponding xdc file.

4. In BIOS 6.x creation of static driver instances follow a different flow and cause functions in module script files to run during build time. In BIOS 5.x creation of driver (for both static and dynamic instances) result in the execution of the mdBindDev function at runtime. Therefore any references to module script files (*.xs files) can be ignored for IOM drivers.

5. The XXX_Module_startup function referenced in this document can be ignored for IOM drivers.

6. IOM drivers have an XXX_init function which needs to be called by the application once per driver. This XXX_init function initializes the driver data structures. This application needs to call this function in the application initialization functions which are usually supplied in the tci file.

7. The functionality and behavior of the functions is the same for both driver models. The mapping of IDriver functions to IOM driver functions are as follows:

| IDriver | IOM driver |
|---|---|
| XXX_Instance_init | mdBindDev |
| XXX_Instance_finalize | mdUnbindDev |
| XXX_open | mdCreateChan |
| XXX_close | mdDeleteChan |
| XXX_control | mdControlChan |
| XXX_submit | mdSubmitChan |

8. All the references to module wide config parameters in the IDriver model map to macro definitions and preprocessor directives (#define and #ifdef etc) for the IOM drivers. e.g.

   a. XXX_edmaEnable in IDriver maps to –D XXX_EDMA_ENABLE for IOM driver.

b. XXX_FIFO_SIZE in IDriver maps to #define FIFO_SIZE in IOM driver header file

9. In BIOS 6.x a cfg file is used for configuring the BIOS and driver options whereas in the BIOS 5.x the “tcf” and “tci” files are used for configuring the options.

10. In BIOS 5.x driver support for multiple devices is implemented as follows. A chip specific compiler define is required by the driver source files (-DCHIP_OMAPL138). Based on the include a chip specific header file (e.g soc_OMAPL138) which contains chip specific defines is used by the driver. In order to support a new chip, a new soc_XXX header file is required and driver sources files need to be changed in places where the chip specific define is used.

### 1.3 Related Documents

| | | |
|---|---|---|
| 1. | Yet to get SPRU Number | DSP/BIOS Driver Developer’s Guide |
| 3. | Yet to get SPRU Number | PSC Specs |

## 1.4 Hardware

The PSC device driver design is in the context of DSP/BIOS running on DSP OMAPL138 core. There are two PSC modules on this platform, name PSC0 and PSC1. Each of these modules cater to the power management requirements of a set of other modules.

The PSC module core used here has the following bocks:

YET TO GET THE BLOCK DIAGRAM

(Specification Diagram Should Not Be Copied)

Figure 1 PSC Block Diagram

## 1.5 Software

The PSC driver discussed here is running DSP/BIOS on the OMAPL138 DSP.

### *1.5.1 Operating Environment and dependencies*

Details about the tools and the BIOS version that the driver is compatible with can be found in the system Release Notes.

### *1.5.2 System Architechture*

The block diagram below shows the overall system architecture.

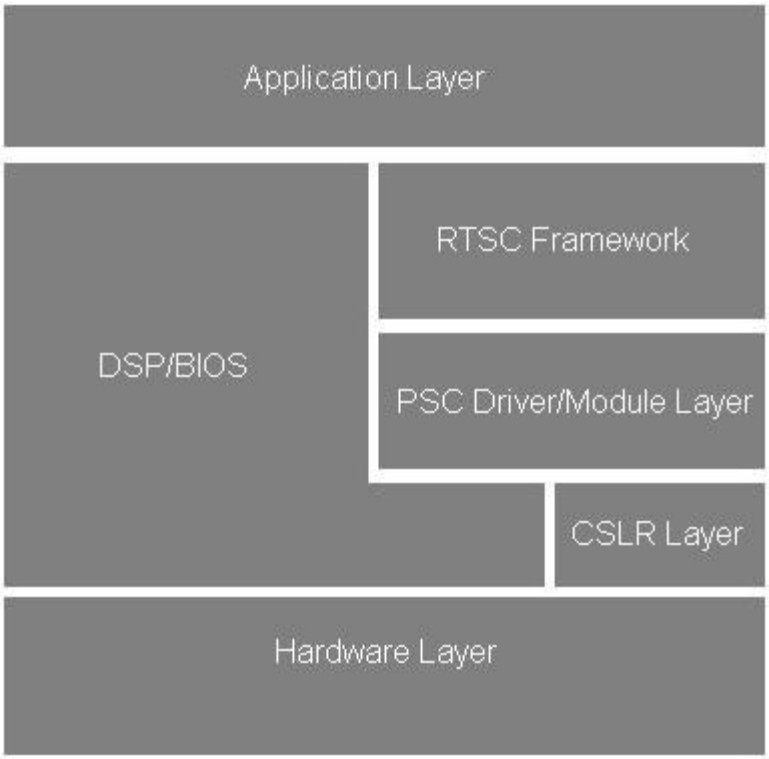


**Figure 2 System Architecture**

The PSC driver is designed as a standalone mode, unlike those inheriting/using IDriver framework. The rationale behind this strategy is that the PSC module exposes APIs which could be used by other driver modules also, apart from the application. In such a scenario, accessing the PSC APIs incase of a IDriver inherited framework would be cumbersome, viz passing through IDriver tables etc. Hence, the design adopts the

standalone module concept as used by other DSP/BIOS6 modules like Clock, Hwi etc, which expose functional APIs to access hardware.

Figure 2 shows the overall device driver architecture.

## 1.6 Component Interfaces

In the following subsections, the interfaces implemented by each of the sub-component are specified. Refer to PSC device driver API reference documentation for complete details on APIs.

### 1.6.1 Module Interface

The Module interface constitutes the Device Driver Manifest to Application, as a standalone component, the APIs of which can be directly called from an application or a driver.

The Module interface projects the PSC driver as a standalone module. Hence, driver and module could be used interchangeably in the current context.

**PSC module specification file (Psc.xdc)**

The XDC file defines the following in its public section: the data structures, enums, constants, IOCTLS, error codes and module wide config variables that shall be exposed for the user.

These definitions would include

ENUMS: Module clock states etc

STRUCTURES: Specific for instance specifc details

CONSTANTS: error ids and ioctls

Also this files specifies the list of configurable items which could be configured/specified by the application during instantiation (instance parameters)

In its private section it would contain the the data structures, enums, constants and module wide config variables. The Instance object (the driver object) would contain all the info related to that particular IO channel. This information might be irrelevant to the User. The instance object is the container for all driver variables, channel objects etc. In essence, it contains the present state of the instance being used by the application.

**PSC module script file ( Psc.xs )**

The script file is the place where static instantiations and references to module usage are handled. This script file is invoked when the application compiles and refers to the Psc module/driver. The Psc.xs file contains two parts

1. Handling the module use references

   When the module use is called in the application cfg for the PSC module, the module use function in the Psc.xs file is used to initialize the hardware instance specific details like base addresses, interrupt numbers, frequency etc. This data is stored for further use during instantiation. This gives the flexibility to design, to handle multiple SOC with single c-code base (as long as the IP does not deviate).

2. Handling static instantiation of the PSC instance

   When the PSC instance is instantiated statically in the application cfg file, (please note that dynamic instantiation is also possible from a C file) the instance static init function is called. If a particular instance is configured with set of instance parameters (fron CFG file) they are used here to configure the instance state. The instance state is populated based on the instance number, like the default state of the driver, channel objects etc.

The PSC module implements the following interfaces

| S.No | IOM Interfaces | Description |
|---|---|---|
| 1 | Psc_Instance_init() | Handle dynamic calls to module instantiation. This shall be a duplication of the tasks done in instance static init in the module script file. |
| 2 | Psc_Instance_finalize() | TBD |
| 3 | Psc_ModuleClkCtrl() | This is the function used by the application or other modules, if required, to control the m power state of the target module like, UART, I2C etc. In effect this controls the clocking of the target module |
| 5 | Psc_RegisterPwrmCallback() | This is currently a stub. This function shall be used, to provide the functionality of registering power management callbacks |

| | | |
|---|---|---|
| | | to the DSP/BIOS PWRM framework. |
| 6 | Psc_RegisterPwrmCallback() | This is currently a stub. This function shall be used, to provide the functionality of un-registering power management callbacks to the DSP/BIOS PWRM framework. |

### 1.6.2 CSLR Interface

The CSL register interface (CSLR) provides register level implementations. CSLR is used by the PSC module to configure PSC registers. CSLR is implemented as a header file that has CSLR macros and register overlay structure.

## 1.7 Design Philosophy

This device driver is written in conformance to the DSP/BIOS standalone model and handles access to and from the PSC hardware.

### 1.7.1 The Module and Instance Concept

The RTSC framework provides the concept of the *module and instance* for the realization of the device and its communication path as a part of the driver implementation.

The module word usage (PSC module) refers to the driver as one entity. Any detail, configuration parameter or setting which shall apply across the driver shall be a module variable. Default parameters shall be a module wide variable since, it should be available during module start up or use call, to initialize all the channels to their default values.

This instance word usage (PSC instance 1) refers to every instantiation of module due to a static or dynamic create call. Each instance shall represent the device directly by holding info like the individual PSC information like base address, device configuration settings, hardware configuration etc. Each hardware instance shall map to one PSC instance. This is represented by the Instance_State in the PSC module configuration file.

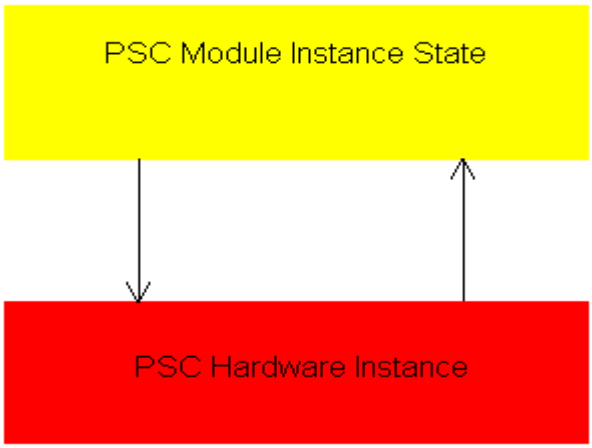


Figure 3 Instance Mapping

Hence every module shall only support the as many number of instantiations as the number of PSC hardware instances on the SOC

### *1.7.2* Design Constraints

PSC standalone driver module imposes the following constraint(s).

- None

# 2 PSC Driver Software Architecture

This section details the data structures used in the PSC module and the interface it presents to the application layer. A diagrammatic representation of the PSC module functions is presented and then the usage scenario is discussed in some more detail.

Following this, we'll discuss the deployed driver or the dynamic view of the driver where the driver operational scenarios are presented.

## 2.1 Static View

### 2.1.1 Functional Decomposition

The driver is designed keeping a device, also called instance, in mind. It is designed to be a standalone module.

This driver uses an internal data structure, HwInfo, to maintain the information of the PSC module, which for now is only the base address of the module register overlay . This information is populated to default values during module instantiation and later preserves the changes as done by the user. This hardware information is held inside the Instance State of the module, along with the instance number. However, this instance state is translated to the Psc_Object structure by the RTSC frame work. The data structures used to maintain this information are explained in greater detail in the following *Data Structures* sub-section. The following figure shows the static view of PSC driver.

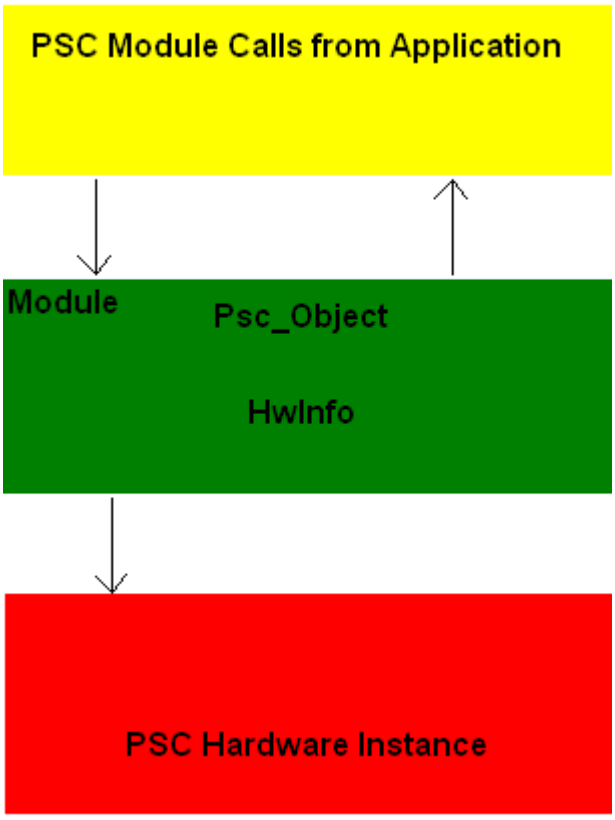


**Figure 4 PSC driver static view**

### 2.1.2 Data Structures

The module employs the Instance State (Psc_Object) and HwInfo structures to maintain information of the instance and banks respectively.

The following sections provide major data structures maintained by PSC module and the instance.

#### 2.1.2.1 The Instance State ( Psc_Object )

The instance state comprises of all data structures and variables that logically represent the actual hardware instance on the hardware. The handle to this is sent out to the application for access when the module instantiation is done via create statically (application CFG file) or dynamically (C file during run time). The parameters that are to be passed for this call is described in the section Device Parameters.

| S.No | Structure Elements *(PSC_Object)* | Description |
|---|---|---|
| 1 | *instNum* | Preserve the instance number of the PSC hardware instance |
| 2 | *HwInfo* | Preserve the hardware specific information of the PSC instance |

#### *2.1.2.2 The Hardware Information Structure (HwInfo)*

The information about PSC hardware module is preserved in this structure. Currently shall have only the base address.

| S.No | Structure Elements (Psc_HwInfo*)* | Description |
|---|---|---|
| 1 | *baseAddress* | The address of base of the register overlay for this PSC instance |

## 2.2 Dynamic View

### 2.2.1 The Execution Threads

The PSC module involves following execution threads:

**Application thread:** Instantiation of PSC module instance, PSC control operations will be under application thread.

### 2.2.2 Input / Output using PSC driver

The PSC module does not have any I/O operations.

### 2.2.3 Functional Decomposition

#### *2.2.3.1 Psc_Instance_init()*

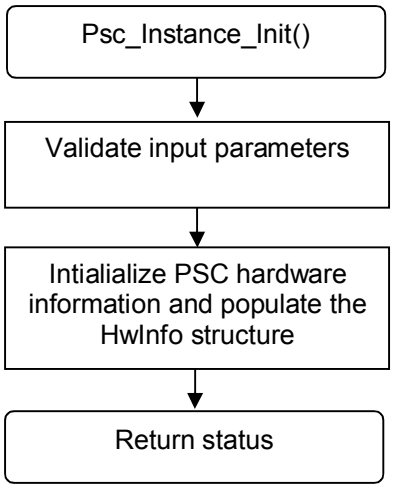


**Figure 5 PSC_Instance_Init() flow diagram**

The instance init function is called when the module is dynamically instantiated. This is the only context available for initialization per instance, when doing dynamic (application C file during run time) instantiation. Hence, this function should be including all the initialization done in the instance static init in the module script file and the module startup function. The return, value of this function represents the extent to which the instance (and hence its resources) were initialized. For example, we could use return value of 0 for complete (successful) initialization done, return value of 1 for failure at the stage of a resource allocation and so on. This return value is preserved by the RTSC/BIOS framework and passed to the Instance_finalize function, which does a clean up of the driver during instance removal accordingly

#### *2.2.3.2 Psc_Instance_finalize()*

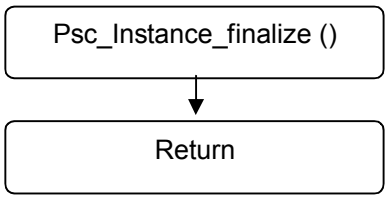


**Figure 6 PSC_Instance_finalize () flow diagram**

The Psc_Instance_init and Psc_Instance_finalize functions are called by the XDC/BIOS framework.

The instance init function does a start up initialization for the driver. This function is called when the module is instantiated dynamically by the Psc_create call by the application. At this module instantiation, any allocation of resource needed later by the driver or any initialization should be done here which form a pre-requisite before the actual functioning. However, for the PSC module no such task in required for now.

The instance finalize function does a final clean up before the driver could be relinquished of any use. Here, all the resources which were allocated during instance initialization shall be unallocated. After this the instance no more is valid and needs to be reinitialized. Please note that the input parameter for this function is the initialization status returned from the instance)init function. This helps in de-allocation of resources only that were actually allocated during instance_init.

#### *2.2.3.3 Psc_ModuleClkCtrl()*

This API shall allow the user to enable or disable clocking to a peripheral. The module on which the operation is required is specified by its LPSC number and additional argument is if the module is to be enabled. A value for enable is enumerated and any other value puts the target module in disabled state.

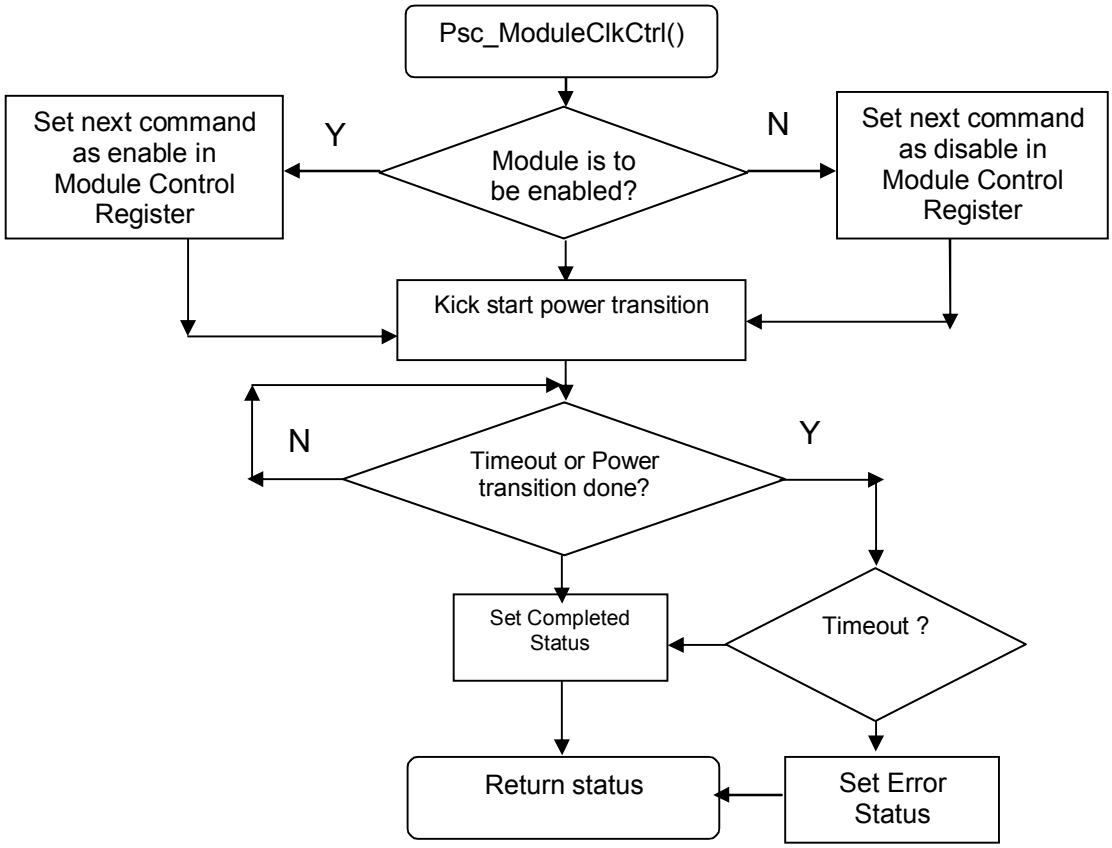


**Figure 7 Psc_ModuleClkCtrl () flow diagram**

#### *2.2.3.4 Psc_RegisterPwrmCallback()*

This API shall allow the user to register callbacks to PWRM module. This is a stub for now.

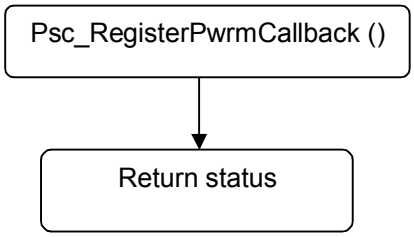


**Figure 8 Psc_RegisterPwrmCallback() flow diagram**

#### *2.2.3.5 Psc_UnregisterPwrmCallback()*

This API shall allow the user to unregister callbacks to PWRM module. This is a stub for now.

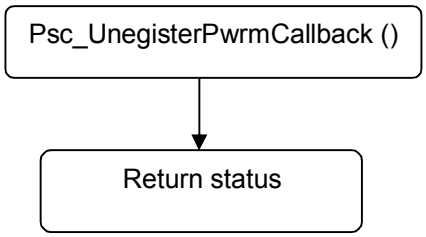


**Figure 9 Psc_UnregisterPwrmCallback() flow diagram**