DSP/BIOS Power Management for OMAP3430

Scott Gary, Ramsey Harris *Software Development Systems*

ABSTRACT

This document provides a summary of the configuration and application programming interfaces (APIs) for the C64x+ version of the DSP/BIOS Power Manager (PWRM module) included in the DSP/BIOS 5.31 product release. The OMAP3430 device is the only C64x+ device with PWRM support in this DSP/BIOS release.

Contents

1 Overview

This document summarizes the configuration and application programming interfaces (APIs) for the C64x+ version of the DSP/BIOS Power Manager (PWRM module) included in the DSP/BIOS 5.31 product release. The OMAP3430 device is the only C64x+ device with PWRM support in this DSP/BIOS release.

The DSP/BIOS PWRM module provides ways to use the following power-related features within an OMAP3430 application:

- **CPU load monitoring.** Your application can gather information about the percentage of time the CPU has been idle recently, and can take action to conserve power based on the results.
- **Dynamic resource tracking.** You can make runtime PWRM API calls to inform the Power Manager of the specific resources (for example, clock domains, peripherals, and clock pins) that your application is dependent upon. With this knowledge of required resources, PWRM can idle resources that have no declared dependencies.
- **Retention state.** You can put the DSP a low-power state where all memory and logic contents are retained.
- **Hibernation state.** You can completely power off the DSP, with context saved before going off, and restored upon power on.
- **Voltage and frequency scaling.** On the OMAP3430, dynamic changes to the operating voltage and frequency of the CPU are possible using ARM-side code. This is called V/F scaling. Since power usage is linearly proportional to the frequency and quadratically proportional to the voltage, using V/F scaling can result in significant power savings.

  On the OMAP3430 platform, setpoint changes occur only via ARM-side control code. These changes may occur either because DSP Bridge has requested a change to the setpoint or because an independent (ARM-side) decision has been made to change the setpoint. Whether the change request is made by DSP Bridge or by ARM-side code, the PWRM module does not control the actual V/F scaling. Instead, its role is to send out scaling event notifications to any DSP-side code that has registered with PWRM for scaling notifications.

  More specifically, when a setpoint is about to change, DSP Bridge calls PWRM_signalEvent to signal the PWRM_PENDINGSETPOINTCHANGE event to any PWRM clients registered for this event. When notifications have completed, DSP Bridge acknowledges this to the ARM-side control code so that scaling can proceed. Following the setpoint change, DSP Bridge calls PWRM_signalEvent to signal the PWRM_DONESETPOINTCHANGE event to any registered clients. As part of these notifications, PWRM's internal setpoint variable is updated and GBL_setFrequency is called to update the CPU frequency known to DSP/BIOS.

2 Configuration

2.1 Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the PWRM Manager Properties section.

Module Configuration Parameters

Name	Type	Default (Enum Options)
ENABLE	Bool	false
IDLECPU	Bool	false
IDLEFXN	Extern	prog.extern("_PWRM_F_idleStopClk")
LOADENABLE	Bool	false
NUMSLOTS	Numeric	1
USECLKPRD	Bool	false
CLKTICKSPERSLOT	Numeric	10
SLOTHOOKFXN	Extern	prog.extern(“FXN_F_nop”)
RESOURCETRACKING	Bool	false
USERRESOURCES	Numeric	0
SHAREDRESOURCEFXN	Extern	prog.extern(“FXN_F_nop”)
SCALING	Bool	false
WARMBOOTMEMSEG	Reference	prog.get(“IRAM”)
CTXBUFMEMSEG	Reference	prog.get(“IRAM”)
SCM_BASEADDR	Address	0x48002000
CM_BASEADDR	Address	0x48004000
PRM_BASEADDR	Address	0x48306000
IVAMMU_BASEADDR	Address	0x5D000000
USETIMER	Bool	false
TIMERID	Enumerated String	“Timer 6”
TIMERBASEADDR	Address	0
TIMERINPUTCLK	Unsigned	32
TIMERPERIOD	Numeric	10
TIMERINTR	Reference	HWI_UNUSED

2.2 PWRM Manager Properties

The following global properties can be set for the PWRM module in the PWRM Manager Properties dialog of Gconf or in a Tconf script:

Enable PWRM Manager. Check this box if you want to enable the power manager. If you do not plan to use the power manager, you should leave it disabled to reduce the size of your application. You must check this box if you intend to configure other PWRM properties.

Tconf Name: ENABLE Type: Bool
Example: `bios.PWRM.ENABLE = true;`

Idle CPU during BIOS idle loop. Check this box to have PWRM idle the CPU during IDL loop processing. This box must be checked if you plan to use PWRM's CPU load monitoring feature.

Tconf Name: IDLECPU Type: Bool
Example: `bios.PWRM.IDLECPU = true;`

Idle function. This property configures an idle function to be called each time the idle thread passes through the idle loop. The IDLECPU property must be true to allow the IDLEFXN property to be defined. You can assign your own idle function or one of the two functions provided by the PWRM module: _PWRM_F_idleStopClk and _PWRM_F_idleStandby. A third function is provided, _PWRM_F_idleGem, which may be called from your own idle function but should not be used as the idle function itself because it does not set the PDCCMD register. If you use this function, your idle function must first set the desired value in the PDCCMD register and then call the _PWRM_F_idleGem function.

These functions perform the following actions.

- **_PWRM_F_idleStopClk:** (default configuration) Sets PDCCMD to stop GEM clocks, performs steps to monitor the CPU load, and executes the idle instruction.
- **_PWRM_F_idleStandby:** Sets PDCCMD to place GEM into standby mode, performs steps to monitor the CPU load, and executes the idle instruction.
- **_PWRM_F_idleGem:** Performs steps to monitor the CPU load and executes the idle instruction.

The idle function signature is:

```
Void idleFxn(Void)
```

Tconf Name: IDLEFXN Type: Extern
Example: `bios.PWRM.IDLEFXN = prog.extern("_PWRM_F_idleStopClk");`

Enable CPU Load Monitoring. Check this box if you want PWRM to measure and accumulate CPU idle time. The "Idle CPU during BIOS idle loop" box must be checked for this box to be writeable.

Tconf Name: LOADENABLE Type: Bool
Example: `bios.PWRM.LOADENABLE = true;`

Number of history slots to buffer. This property allows you to specify the number of "slots" or intervals of CPU load history to be buffered within PWRM. The "Enable CPU Load Monitoring" box must be checked for this property to be writeable.

Tconf Name: NUMSLOTS Type: Numeric
Example: `bios.PWRM.NUMSLOTS = 5;`

Use CLK and PRD for finalizing slots. Check this box if you want to use the DSP/BIOS CLK and PRD modules to determine when a history slot should be finalized (completed). Currently this the only supported method for finalizing slots, and you should always check this box if you want to use PWRM's CPU load monitoring feature. The "Enable CPU Load Monitoring" box must be checked for this box to be writeable.

Tconf Name: USECLKPRD Type: Bool
Example: `bios.PWRM.USECLKPRD = true;`

Number of CLK ticks per slot. This property allows you to specify the duration of CPU load history slots, in terms of CLK clock ticks. For example, if the CLK module is configured for 1 tick per millisecond, then a value of 10 for this property means CPU history slot has a duration of 10 milliseconds. The "Use CLK and PRD for finalizing slots" box must be checked for this box to be writeable.

Tconf Name: CLKTICKSPERSLOT Type: Numeric
Example: `bios.PWRM.CLKTICKSPERSLOT = 10;`

Hook function to call upon slot finalization. This property allows you to configure a function to be called upon the finalization of each history slot. This function will be called as the last step of finalization, after the slot data has been written to PWRM's internal history buffer. The "Use CLK and PRD for finalizing slots" box must be checked for this property to be writeable.

The callout function signature is:

```
Void slotHookFxn(Uns arg0)
```

When this function is called, `arg0` indicates the slot finalization timestamp.

Tconf Name: SLOTHOOKFXN Type: Extern
Example: `bios.PWRM.SLOTHOOKFXN = prog.extern("myHookFunction");`

Enable Resource Tracking. Check this box if you want to enable PWRM support for dynamic resource tracking.

Tconf Name: RESOURCETRACKING Type: Bool
Example: `bios.PWRM.RESOURCETRACKING = true;`

Number of user-defined resources to support. This property allows you to define additional resources (beyond those device-specific resources supported by default), that can be tracked and reference counted by PWRM. For example, for OMAP3430, the header file pwrm3430.h defines a PWRM_Resource enumeration listing the resources PWRM supports by default. The last element of this enumeration is "PWRM_3430_USER_BASE", which can serve as a resource ID for the first user-defined resource. If, for example, you specify that there are two user-defined resources, PWRM allocates reference counting support for these resources at PWRM initialization time, and recognizes these resources as IDs PWRM_3430_USER_BASE and PWRM_3430_USER_BASE+1. The "Enable Resource Tracking" box must be checked for this property to be writeable.

Tconf Name: USERRESOURCES Type: Numeric
Example: `bios.PWRM.USERRESOURCES = 4;`

Callout function for shared resources. This property allows you to specify the function PWRM should call upon 0→1 and 1→0 reference count transitions.

For example, when a driver declares the first dependency upon a resource (via a call to PWRM_setDependency), this is a 0→1 reference count transition. When this occurs, PWRM makes a callout to the configured function to perform the necessary processing to enable the resource. Subsequent calls to set more dependencies on the same resource result in increments of the reference count, but no further calls to the callout function. Dependencies upon the resource can be released via calls to PWRM_releaseDependency, and when the last dependency upon the resource is released, this is 1→0 reference count transition, and PWRM will callout again to the configured function to do the necessary processing to disable the resource.

The callout function signature is:

```
Uns resourceCallout(Uns arg0, Uns arg1)
```

When this function is called: `arg0` indicates the resourceID of the corresponding call to PWRM_setDependency or PWRM_releaseDependency; and `arg1` is PWRM_SET if a 0→1 transition occurred, or PWRM_RELEASE if a 1→0 transition occurred. The callout function should return TRUE on success, and FALSE if a failure occurs.

The "Enable Resource Tracking" box must be checked for this property to be writeable.

Tconf Name: SHAREDRESOURCEFXN Type: Extern
Example: `bios.PWRM.SHAREDRESOURCEFXN = prog.extern("rsrcCallout");`

Enable V/F Scaling Support. Check this box to enable PWRM's support for V/F scaling.

Tconf Name: SCALING Type: Bool
Example: `bios.PWRM.SCALING = true;`

MEM section for hibernate resume code. This property allows you to specify the MEM section where PWRM's code for resuming from the PWRM_HIBERNATE sleep mode should be located. Since the C64x+ processor is powered off during hibernation, you must specify an external MEM section that remains powered during hibernation.

Tconf Name: WARMBOOTMEMSEG Type: Reference
Example: `bios.PWRM.WARMBOOTMEMSEG = prog.get("DDR");`

MEM section for hibernate context. This property allows you to specify the MEM section where the DSP context should be saved to prior to powering off the DSP for the PWRM_HIBERNATE sleep mode. You must specify an external MEM section that remains powered during hibernation.

Tconf Name: CTXBUFMEMSEG Type: Reference
Example: `bios.PWRM.CTXBUFMEMSEG = prog.get("DDR");`

Base address of System Control Module (SCM). When enabling the IVA MMU, use this configuration property to specify the virtual address mapped to the physical address of the System Control Module. The IVA MMU must map one page (4 KB) of virtual memory starting at the given address.

Tconf Name: SCM_BASEADDR Type: Address
Example: `bios.PWRM.SCM_BASEADDR = 0x12007000;`

Base address of Clock Manager (CM). When enabling the IVA MMU, use this configuration property to specify the virtual address mapped to the physical address of the Clock Manager. The IVA MMU must map one page (4 KB) of virtual memory starting at the given address.

Tconf Name: CM_BASEADDR Type: Address
Example: `bios.PWRM.CM_BASEADDR = 0x12004000;`

Base address of Power & Reset Manager (PRM). When enabling the IVA MMU, use this configuration property to specify the virtual address mapped to the physical address of the Power & Reset Manager. The IVA MMU must map one page (4 KB) of virtual memory starting at the given address.

Tconf Name: PRM_BASEADDR Type: Address
Example: `bios.PWRM.PRM_BASEADDR = 0x12006000;`

Base address of IVA MMU. When enabling the IVA MMU, use this configuration property to specify the virtual address mapped to the physical address of the IVA MMU. The IVA MMU must map one page (4 KB) of virtual memory starting at the given address.

Tconf Name: IVAMMU_BASEADDR Type: Address
Example: `bios.PWRM.IVAMMU_BASEADDR = 0x12005000;`

Use timer based CPU load monitoring. To use timer-based CPU load monitoring, set the USETIMER property to 1. Note, you must first set USECLKPRD to 0 in order to select the timer configuration (bios.PWRM.USECLKPRD = 0).

Tconf Name: USETIMER Type: Bool
Example: `bios.PWRM.USETIMER = 1;`

Select timer. Select which timer to use as the time base for CPU load monitoring. You may choose Timer 5, 6, 7, or 8. Note that you cannot use the same timer that is being used for the DSP/BIOS clock.

Tconf Name: TIMERID Type: Enumerated String
Example: `bios.PWRM.TIMERID = “Timer 7”;`

Base address of timer. When enabling the IVA MMU, use this configuration property to specify the virtual address mapped to the physical address of the timer. The IVA MMU must map one page (4 KB) of virtual memory starting at the given address. If the IVA MMU is disabled, the timer physical address must be specified.

Tconf Name: TIMERBASEADDR Type: Address
Example: `bios.PWRM.TIMERBASEADDR = 0x4903C000;`

Timer input clock speed. The timer input clock speed must be specified. It is specified in KHz.

Tconf Name: TIMERINPUTCLK Type: Unsigned
Example: `bios.PWRM.TIMERINPUTCLK = 19200;`

Timer period. Specify the period at which the timer should finalize the CPU load monitoring slots. The period is specified in milliseconds.

Tconf Name: TIMERPERIOD Type: Unsigned
Example: `bios.PWRM.TIMERPERIOD = 10;`

Timer interrupt. Specify which CPU interrupt the timer should be routed to.

Tconf Name: TIMERINTR Type: Reference
Example: `bios.PWRM.TIMERINTR = prog.get(“HWI_INT15”);`

2.3 Graphical View

The following dialog allows you to set PWRM properties in GConf:

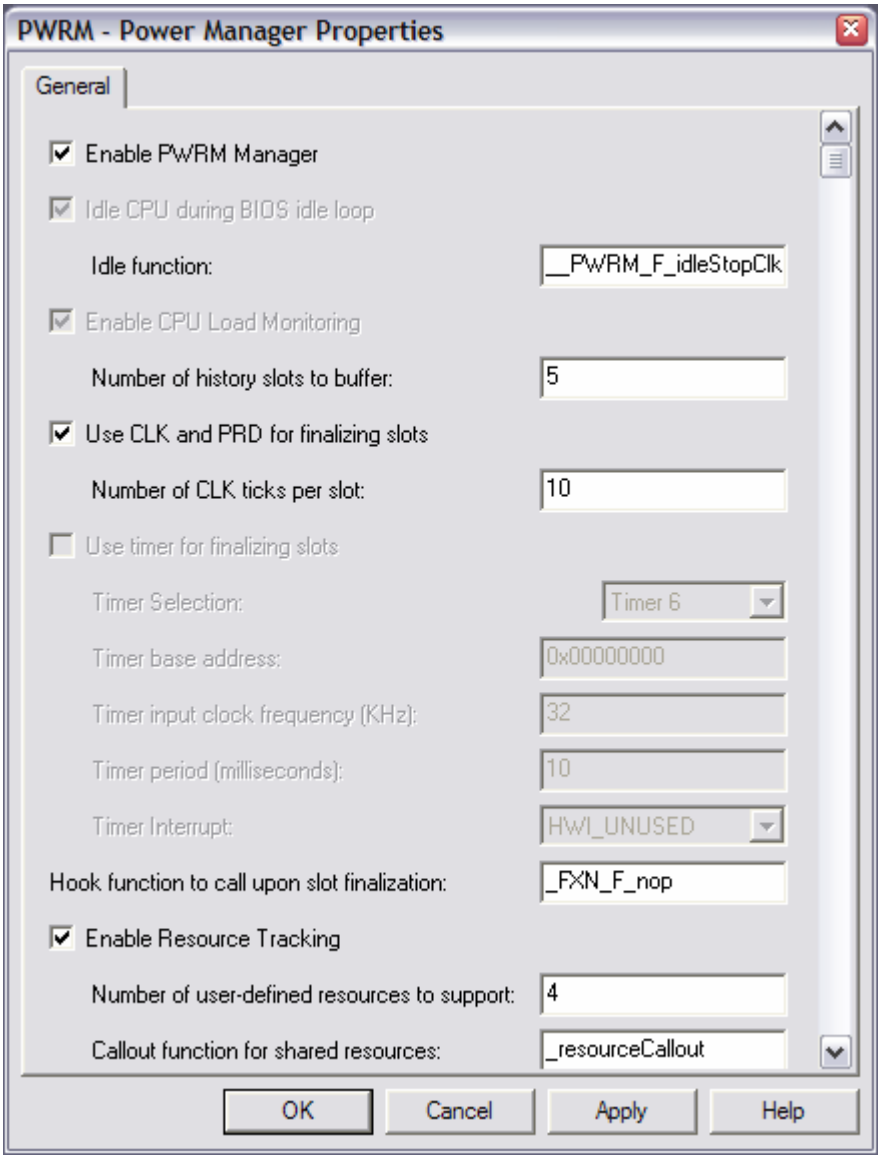

Figure 1. PWRM Properties Dialog (top portion)

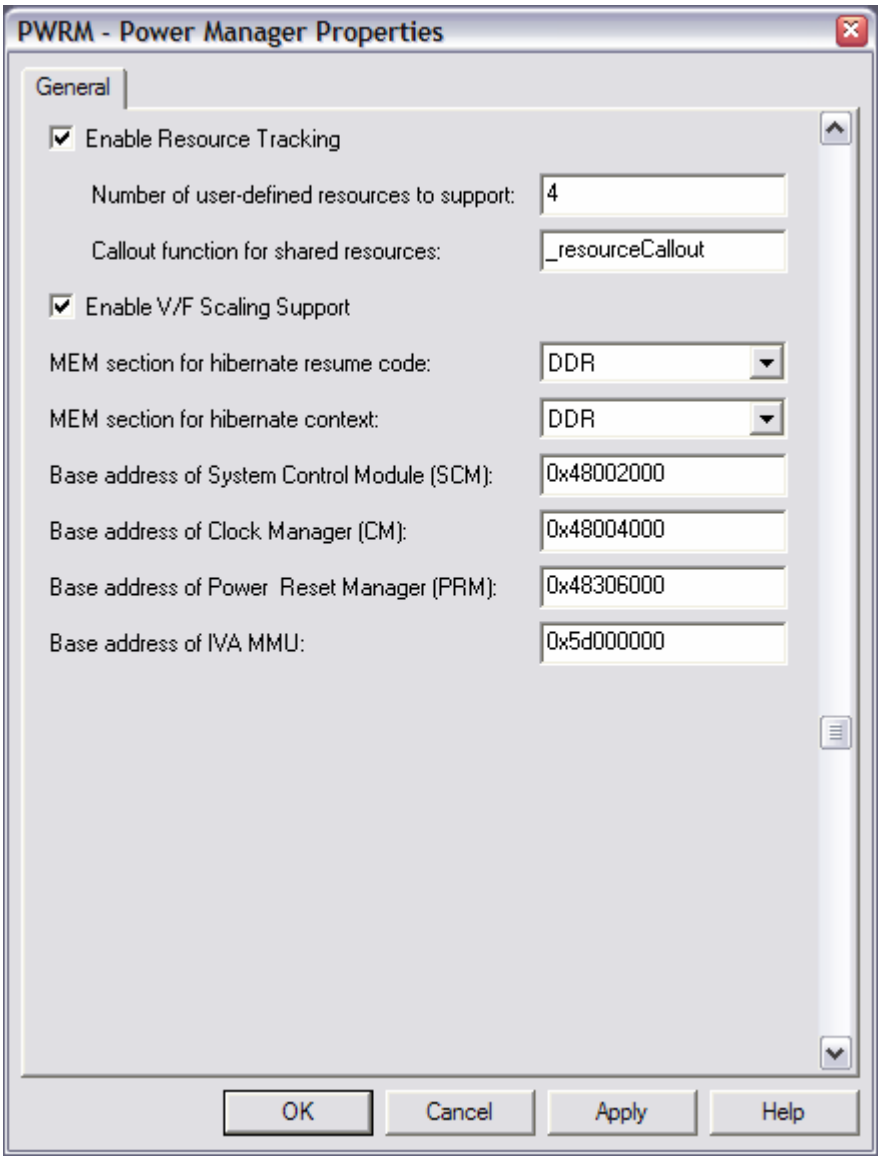

Figure 2. PWRM Properties Dialog (lower portion)

3 DSP/BIOS Power Manager API Reference

The PWRM APIs for C64x+ are summarized in the following table, and listed on the following pages in reference format.

Function	Purpose
PWRM_getCapabilities	Get information on PWRM's capabilities on the current platform
PWRM_getCPULoad	Get CPU load information as measured by PWRM.
PWRM_getCurrentSetpoint	Get the current setpoint in effect
PWRM_getDependencyCount	Get count of dependencies currently declared on a resource.
PWRM_getLoadMonitorInfo	Get PWRM load monitor configuration info.
PWRM_getNumSetpoints	Get the number of setpoints supported for the current platform
PWRM_getSetpointInfo	Get the corresponding frequency and voltage for a setpoint
PWRM_initSetpointInfo	Initialize PWRM's setpoint info.
PWRM_registerNotify	Register a function to be called on a specific power event
PWRM_releaseDependency	Release a dependency that has been previously declared.
PWRM_resetCPULoadHistory	Reset PWRM's CPU load history buffer.
PWRM_setDependency	Declare a dependency upon a resource.
PWRM_signalEvent	Signal a PWRM event to clients who've registered for the event.
PWRM_sleepDSP	Transition the DSP to a new sleep state
PWRM_startCPULoadMonitoring	Start CPU load monitoring.
PWRM_stopCPULoadMonitoring	Stop CPU load monitoring.
PWRM_unregisterNotify	Unregister for an event notification from PWRM.
PWRM_validateSetpoint	Check whether a setpoint conflicts with constraints of registered notification clients.

PWRM_getCapabilities	Get information on PWRM capabilities on the current platform

Syntax `status = PWRM_getCapabilities(capsMask);`

Parameters `Uns * capsMask /* ptr to location of capabilities mask */`

Return Value `PWRM_Status status; /* returned status */`

Reentrant Yes

Description PWRM_getCapabilities returns information about the PWRM module's capabilities on the current platform.

The capsMask parameter should point to the location where PWRM_getCapabilities should write a bitmask that defines the capabilities. You can use the following constants to check for capabilities in the bitmask:

Name	Usage
PWRM_CLOADMONITORING	The PWRM module supports CPU load monitoring.
PWRM_CRESOURCETRACKING	The PWRM module supports dynamic resource tracking.
PWRM_CVFSCALING	The PWRM module supports voltage and frequency scaling.

PWRM_getCapabilities returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded.
PWRM_EINVALIDPOINTER	The operation failed because the capsMask parameter was NULL.

Example

```
PWRM_Status status;
Uns capsMask;

/* Query PWRM capabilities on this platform */
status = PWRM_getCapabilities(&capsMask);
if (status == PWRM_SOK) {
    LOG_printf(TRACE, "Caps mask=0x%X", capsMask);
    if ((capsMask & PWRM_CVFSCALING) == 0) {
        LOG_printf(TRACE, "V/F scaling not supported!");
    }
}
else {
    LOG_printf(TRACE, "ERROR: status = %x", status);
}
```

PWRM_getCPULoad	Get CPU load information

Syntax

```
status = PWRM_getCPULoad(numSlots, loadInfo);
```

Parameters

```
Uns numSlots;  /* # of history slots of load info to retrieve */
PWRM_CPULoadInfo *loadInfo;
        /* array of load info structures to be filled by PWRM */
```

Return Value

```
PWRM_Status status;          /* returned status */
```

Reentrant Yes

Description PWRM_getCPULoad reports CPU load information accumulated by the PWRM module. Load history is accumulated in “slots” of a configured duration. (See PWRM Manager Properties on page 4 for details on configuring the number of slots to be buffered by PWRM and the duration of those slots.)

The PWRM_CPULoadInfo structure reports the total number of CPU cycles for the slot, the number of those cycles where the CPU was busy, and a timestamp indicating when the slot was finalized (completed).

```
typedef struct PWRM_CPULoadInfo {
    Uns busyCycles;              /* number of cycles CPU was busy */
    Uns totalCycles;             /* total number of CPU cycles in slot */
    Uns timeStamp;               /* time when slot finalized */
} PWRM_CPULoadInfo;
```

The numSlots parameter specifies how many history slots of the loadInfo array should be written by PWRM. History slots are reported in last-in, first-out (LIFO) order. In other words, the most recently finalized history slot is reported in the first element of the loadInfo array, the slot previous to that is reported in the second element, and so on.

PWRM maintains a ring buffer of history slots. Once all history slots have been filled, finalizing the next slot causes the oldest history slot to be overwritten. The number of slots buffered in PWRM is configured statically. If numSlots is greater than the number of slots maintained by PWRM, then a PWRM_EOUTOFRANGE error is reported, and no history data is copied to loadInfo.

If a history slot is “empty” then the slot data elements have a default (reset) value:

```
PWRM_CPULoadInfo _PWRM_resetLoad = {
    0,   /* busyCycles */
    0,   /* totalCycles */
    0    /* timestamp */
};
```

Slots are empty if not enough time has passed for PWRM’s internal slot buffer to fill, either since startup, or since a call to PWRM_resetCPULoadHistory.

PWRM only reports load information for finalized slots. In other words, PWRM does not report information on the currently filling but not yet completed slot. It is only when the current slot is finalized that the accumulated busy and total cycles will be stored in PWRM's ring buffer.

PWRM_getCPULoad returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded.
PWRM_EINITFAILURE	The operation failed because there was a failure during load monitor initialization.
PWRM_EINVALIDPOINTER	The operation failed because the loadInfo parameter was NULL.
PWRM_ENOTSUPPORTED	The operation failed because load monitoring is not enabled.
PWRM_EOUTOFRANGE	The operation failed because numSlots is greater than the number of history slots PWRM has been configured to maintain.

Example

```
#define NUMSLOTS          5

PWRM_CPULoadInfo history[NUMSLOTS];

status = PWRM_getCPULoad(NUMSLOTS, history);
if (status == PWRM_SOK) {
    displayCPULoad(history);
}
else {
    LOG_printf(TRACE, "Error: status = %x", status);
}
```

PWRM_getCurrentSetpoint	Get the current setpoint

Syntax status = PWRM_getCurrentSetpoint(setpoint);

Parameters
```
Uns *setpoint;          /* current V/F setpoint */
```

Return Value
```
PWRM_Status status;     /* returned status */
```

Reentrant No

Description PWRM_getCurrentSetpoint returns the V/F scaling setpoint currently in use. The setpoint parameter should point to the location where PWRM_getCurrentSetpoint should write the current setpoint.

PWRM_getCurrentSetpoint returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded.
PWRM_EINVALIDPOINTER	The operation failed because the setpoint parameter was NULL.
PWRM_EINITFAILURE	A failure occurred while initializing V/F scaling support; V/F scaling is unavailable.
PWRM_ENOTSUPPORTED	The operation failed because V/F scaling is not enabled.

Constraints and Calling Context

- Attempts to call PWRM_getCurrentSetpoint before PWRM_initSetpointInfo is called result in a return code of PWRM_EINITFAILURE. Once PWRM_initSetpointInfo is successfully called, PWRM_getCurrentSetpoint should become functional.
- If a call to PWRM_getCurrentSetpoint is made during a change to the current setpoint, the value PWRM_getCurrentSetpoint returns may be the old setpoint and not the new setpoint.

Example

```
PWRM_Status status;
Uns currSetpoint;
status = PWRM_getCurrentSetpoint(&currSetpoint);
if (status == PWRM_SOK) {
    LOG_printf(TRACE, "Setpoint: %d", currSetpoint);
}
else {
    LOG_printf(TRACE, "ERROR: status = %x", status);
}
```

PWRM_getDependencyCount	Get count of dependencies declared on resource

Syntax status = PWRM_getDependencyCount(resourceID, count);

Parameters

```
Uns resourceID;          /* resource ID  */
Uns *count;              /* pointer to where count is written */
```

Return Value

```
PWRM_Status status;      /* returned status */
```

Reentrant Yes

Description PWRM_getDependencyCount returns the number of dependencies that are currently declared on a resource. Normally this corresponds to the number of times PWRM_setDependency has been called for the resource, minus the number of times PWRM_releaseDependency has been called for the same resource.

Resource IDs are device-specific, and are defined in a PWRM_Resource enumeration in a device-specific header file. For example, see pwrm3430.h for OMAP3430. Additionally, users may declare “user-defined” resources, and reference these as an offset to the last enumerated pre-defined resource. For more information, see the "Number of user-defined resources to support" description in the PWRM Manager Properties section on page 4.

PWRM_getDependencyCount returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded, and the reference count was written to the location pointed to by count.
PWRM_EINITFAILURE	An error occurred during initialization for user-defined resources.
PWRM_EINVALIDPOINTER	The operation failed because the count parameter was NULL.
PWRM_ENOTSUPPORTED	The operation failed because resource tracking is not enabled.
PWRM_EOUTOFRANGE	The specified resourceID is outside the range of valid pre-defined or user-defined resource IDs.

Example

```
/* Display some dependency counts */
LOG_printf(TRACE, "Initial dependencies:");

PWRM_getDependencyCount(PWRM_3430_GPTIMER_5, &count);
LOG_printf(TRACE, "GPT5 count = %d", count);

PWRM_getDependencyCount(PWRM_3430_BIOS_CLK, &count);
LOG_printf(TRACE, "BIOS CLK count = %d", count);

PWRM_getDependencyCount(PWRM_3430_MCBSP_1,&count);
LOG_printf(TRACE, "MCBSP_1 count = %d", count);
```

PWRM_getLoadMonitorInfo — Get PWRM load monitor configuration

Syntax

```
status = PWRM_getLoadMonitorInfo(monitorInfo)
```

Parameters

```
PWRM_LoadMonitorInfo *monitorInfo;
                /* pointer to PWRM_LoadMonitorInfo structure */
```

Return Value

```
PWRM_Status status;     /* returned status */
```

Reentrant Yes

Description PWRM_getLoadMonitorInfo returns information about PWRM's load monitor configuration, in the structure pointed to by monitorInfo.

```
typedef struct PWRM_LoadMonitorInfo {
    Uns numSlots;                           /* number of history slots */
    Uns ticksPerSlot;                       /* number of CLK ticks per slot */
} PWRM_LoadMonitorInfo;
```

PWRM_getLoadMonitorInfo returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded.
PWRM_EINVALIDPOINTER	The operation failed because the monitorInfo parameter was NULL.
PWRM_ENOTSUPPORTED	The operation failed because CPU load monitoring is not enabled.

Example

```
PWRM_LoadMonitorInfo monitorInfo;
status = PWRM_getLoadMonitorInfo(&monitorInfo);
if (status == PWRM_SOK) {
    LOG_printf(TRACE, "numSlots=%d, ticksPerSlot=%d",
            monitorInfo.numSlots, monitorInfo.ticksPerSlot);
}
else {
    LOG_printf(TRACE, "Error: status = %x", status);
}
```

PWRM_getNumSetpoints	Get the number of setpoints supported by platform

Syntax `status = PWRM_getNumSetpoints(numberSetpoints);`

Parameters
```
Uns *numberSetpoints;   /* number of supported setpoints */
```

Return Value
```
PWRM_Status status;     /* returned status */
```

Reentrant Yes

Description PWRM_getNumSetpoints returns the number of setpoints supported by the platform. The numberSetpoints parameter should point to the location where PWRM_getNumSetpoints should write the number of setpoints.

PWRM_getNumSetpoints returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded.
PWRM_EINVALIDPOINTER	The operation failed because the numberSetpoints parameter was NULL.
PWRM_EINITFAILURE	V/F scaling support has not been initialized; V/F scaling is unavailable.
PWRM_ENOTSUPPORTED	The operation failed because V/F scaling is not enabled.

Constraints and Calling Context

- Attempts to call PWRM_getNumSetpoints before PWRM_initSetpointInfo is called result in a return code of PWRM_EINITFAILURE. Once PWRM_initSetpointInfo is successfully called, PWRM_getNumSetpoints should become functional.

Example

```
PWRM_Status status;
Uns numSetpoints;

status = PWRM_getNumSetpoints(&numSetpoints);
if (status == PWRM_SOK) {
    LOG_printf(TRACE, "NumSetpoints: %d", numSetpoints);
}
else {
    LOG_printf(TRACE, "Error: status = %x", status);
}
```

PWRM_getSetpointInfo	Get frequency and voltage for a setpoint

Syntax

```
status = PWRM_getSetpointInfo(setpoint, frequency, voltage);
```

Parameters

```
Uns setpoint;           /* the setpoint to query */
Uns *frequency;         /* CPU frequency (in kHz) */
Uns *voltage;           /* CPU core voltage (in millivolts) */
```

Return Value

```
PWRM_Status status;     /* returned status */
```

Reentrant Yes

Description This function returns the DSP CPU frequency and voltage for a given setpoint.

The setpoint parameter should specify the setpoint value for which you want to know the frequency and voltage on this platform. The frequency parameter should point to the location where PWRM_getSetpointInfo should write the CPU frequency for the specified setpoint. The voltage parameter should point to the location where PWRM_getSetpointInfo should write the CPU core voltage for the specified setpoint.

PWRM_getSetpointInfo returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded.
PWRM_EINVALIDPOINTER	The operation failed because the frequency or voltage parameter was NULL.
PWRM_EINITFAILURE	V/F scaling support has not been initialized; V/F scaling is unavailable.
PWRM_ENOTSUPPORTED	The operation failed because V/F scaling is not enabled.
PWRM_EOUTOFRANGE	The operation failed because the setpoint parameter is out of range of valid setpoints for the platform.

Constraints and Calling Context

- Attempts to call PWRM_getSetpointInfo before PWRM_initSetpointInfo is called result in a return code of PWRM_EINITFAILURE. Once PWRM_initSetpointInfo is successfully called, PWRM_getSetpointInfo should become functional.

Example

```
#define MAX_SETPOINTS    4
PWRM_Status status;

/* arrays for saving setpoint info */
Uns freq[MAX_SETPOINTS];
Uns volts[MAX_SETPOINTS];

status = PWRM_getSetpointInfo(i, &freq[i], &volts[i]);
if (status != PWRM_SOK) {
    LOG_printf(TRACE, "Error: status=%x", status);
}
```

PWRM_initSetpointInfo	Initialize V/F setpoint information for this platform

Syntax

```
status =  PWRM_initSetpointInfo(numSetpoints, currentSetpoint,
            setpointInfo);
```

Parameters

```
Uns numSetpoints ;              /* number of setpoints */
Uns currentSetpoint ;           /* the current setpoint */
PWRM_SetpointInfo * setpointInfo;   /* setpoint info array */
```

Return Value

```
PWRM_Status status;             /* returned status */
```

Reentrant No

Description PWRM_initSetpointInfo is called during program initialization to inform PWRM of the number of V/F setpoints available on the platform, the current setpoint, and the voltage and frequency information for each setpoint. Setpoint voltage and frequency details are specified via the setpointInfo array of PWRM_SetpointInfo structures.

```
typedef struct PWRM_SetpointInfo {
    Uns frequency;                          /* frequency in kHz */
    Uns voltage;                            /* voltage in millivolts */
} PWRM_SetpointInfo;
```

The first element of the array corresponds to setpoint “0”, the second element for setpoint “1”, etc.

PWRM_initSetpointInfo should be called only once during initialization. Attempts to call other setpoint related APIs (for example, PWRM_getNumSetpoints) before PWRM_initSetpointInfo is called will result in PWRM_EINITFAILURE return codes from those APIs. Once PWRM_initSetpointInfo is successfully called, the related setpoint APIs should become functional.

If PWRM_initSetpointInfo is called more than once, a PWRM_ETOOMANYCALLS error is returned.

PWRM_initSetpointInfo returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded.
PWRM_EFAIL	The operation failed due to a memory allocation failure.
PWRM_EINVALIDPOINTER	The operation failed because the setpointInfo parameter was NULL.
PWRM_ENOTSUPPORTED	The operation failed because V/F scaling is not enabled.
PWRM_ETOOMANYCALLS	The operation failed because setpoint info had already been initialized in a previous call.

Constraints and Calling Context

- PWRM_initSetpointInfo should only be called once, during program initialization.

Example

```
#define NUM_SETPOINTS  4

PWRM_SetpointInfo platformSPInfo[NUM_SETPOINTS];
/* initialize setpoint info array */

. . .

/* now populate PWRM's setpoint info */
status = PWRM_initSetpointInfo(NUM_SETPOINTS,2,platformSPInfo);
if (status != PWRM_SOK) {
    LOG_printf(TRACE, "Error: status = %x", status);
}
```

PWRM_registerNotify	Register a function to be called on a specific power event

Syntax

```
status = PWRM_registerNotify(eventType, eventMask, notifyFxn,
      clientArg, notifyHandle, delayedCompletionFxn);
```

Parameters

```
PWRM_Event eventType;          /* type of power event */
LgUns eventMask;               /* event-specific mask */
Fxn notifyFxn;                 /* function to call on event */
Arg clientArg;                 /* argument to pass to notifyFxn */
PWRM_NotifyHandle *notifyHandle;    /* handle for unregistering */
Fxn *delayedCompletionFxn;     /* funct. for delayed completion */
```

Return Value

```
PWRM_Status status;            /* returned status */
```

Reentrant Yes

Description PWRM_registerNotify registers a function to be called when a specific power event occurs. Registrations and the corresponding notifications are processed in FIFO order. The function registered must behave as described in the pwrmNotifyFxn section.

The eventType parameter identifies the type of power event for which the notify function being registered is to be called. This parameter has an enumerated type of PWRM_Event. This parameter may have one of the following values:

Name	Usage
PWRM_PENDINGSETPOINTCHANGE	V/F setpoint is about to change.
PWRM_DONESETPOINTCHANGE	The pending V/F setpoint change has now been made.
PWRM_GOINGTOSTANDBY	The DSP is going to standby state.
PWRM_AWAKEFROMSTANDBY	The DSP has awoken from standby.
PWRM_GOINGTORETENTION	The DSP is going to retention state.
PWRM_AWAKEFROMRETENTION	The DSP has awoken from retention.
PWRM_GOINGTORETENTIONALT	The DSP is going to retention state.
PWRM_AWAKEFROMRETENTIONALT	The DSP has awoken from retention.
PWRM_GOINGTOHIBERNATE	The DSP is going to hibernate (off) state.
PWRM_AWAKEFROMHIBERNATE	The DSP has awoken from hibernate.

The eventMask parameter is an event-specific mask. Currently eventMask is relevant only to setpoint changes, but it may be used in the future for other power events. For V/F setpoint registrations, this mask defines the setpoints the client supports. For example, if the client supports only one setpoint, it should set only the single corresponding bit in eventMask.

The notifyFxn parameter specifies the function to call when the specified power event occurs. The notifyFxn must behave as described in the pwrmNotifyFxn section.

The clientArg parameter is an arbitrary argument to be passed to the client upon notification. This argument may allow one notify function to be used by multiple instances of a driver (that is, the clientArg can be used to identify the instance of the driver that is being notified).

The notifyHandle parameter should point to the location where PWRM_registerNotify should write a notification handle. If the application later needs to unregister the notification function, the application should pass this handle to PWRM_unregisterNotify.

The delayedCompletionFxn is a pointer to a function provided by the PWRM module to the client at registration time. If a client cannot act immediately upon notification, its notify function should return PWRM_NOTIFYNOTDONE. Later, when the action is complete, the client should call the delayedCompletionFxn to signal PWRM that it has finished. The delayedCompletionFxn is a void function, taking no arguments, and having no return value. If a client can and does act immediately on the notification, it should return PWRM_NOTIFYDONE in response to notification, and should not call the delayedCompletionFxn.

For example, for a DMA driver to prepare for a setpoint change, it may need to wait for the current DMA transfer to complete. When the driver's DMA completes (for example, on the next hardware interrupt), it calls the delayedCompletionFxn function provided when it registered for notification. This completion function tells the PWRM module that the driver is finished. Meanwhile, the PWRM module was able to continue notifying other clients, and was waiting for all clients to signal completion.

This function returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded.
PWRM_EFAIL	The operation failed due to a memory allocation failure.
PWRM_EINVALIDPOINTER	The operation failed because the notifyFxn, notifyHandle or delayedCompletionFxn parameter was NULL.
PWRM_EINVALIDEVENT	Operation failed because eventType is invalid.

Constraints and Calling Context

- PWRM_registerNotify cannot be called from a SWI or HWI. This is because PWRM_registerNotify internally calls MEM_alloc, which may cause a context switch.

Example

```
/* client #1 allows all setpoints */
#define ALLSETPOINTSALLOWED 0xFFFFFFFF

/* client #2 doesn't allow lowest 4 setpoints */
#define SOMESETPOINTSALLOWED 0xFFFFFFF0

/* notification handles */
PWRM_NotifyHandle notifyHandle1;
PWRM_NotifyHandle notifyHandle2;

/* pointers to returned delayed completion fxns */
Fxn delayFxn1;
Fxn delayFxn2;

/* client #1 registers pre-setpoint notification */
PWRM_registerNotify(PWRM_PENDINGSETPOINTCHANGE,
    ALLSETPOINTSALLOWED, (Fxn)myNotifyFxn1,
    (Arg)0x1111, &notifyHandle1, (Fxn *) &delayFxn1);

/* client #2 registers post-setpoint notification */
PWRM_registerNotify(PWRM_DONESETPOINTCHANGE,
    SOMESETPOINTSALLOWED, (Fxn)myNotifyFxn2,
    (Arg)0x2222, &notifyHandle2, &delayFxn2);
```

pwrmNotifyFxn	Function to be called for power event notification

Syntax

```
status = notifyFxn(eventType, eventArg1, eventArg2, clientArg);
```

Parameters

```
PWRM_Event eventType;   /* type of power event */
Arg eventArg1;          /* event-specific argument */
Arg eventArg2;          /* event-specific argument */
Arg clientArg;          /* arbitrary argument */
```

Return Value

```
PWRM_NotifyResponse status;   /* returned status */
```

Description PWRM_registerNotify registers a function to be called when a specific power event occurs. Clients, which are typically drivers, register notification functions they need to run when a particular power event occurs.

This topic describes the required prototype and behavior of such notification functions. Your application must provide and register these functions. Registered functions are called internally by the PWRM module.

The eventType parameter identifies the type of power event for which the notify function is being called. This parameter has an enumerated type of PWRM_Event. The values for this parameter are listed in the PWRM_registerNotify topic.

The eventArg1 and eventArg2 parameters are event-specific arguments. Currently, eventArg1 and eventArg2 are used only for V/F scaling events:

- **PWRM_PENDINGSETPOINTCHANGE.** The eventArg1 holds the current setpoint, and eventArg2 holds the pending setpoint.
- **PWRM_DONESETPOINTCHANGE.** The eventArg1 holds the previous setpoint, and eventArg2 holds the new setpoint.

The clientArg parameter holds the arbitrary argument passed to PWRM_registerNotify when this function was registered. This argument may allow one notify function to be used by multiple instances of a driver (that is, the clientArg can be used to identify the instance of the driver that is being notified).

The notification function must return one of the following constants as a status value of type PWRM_NotifyResponse:

Name	Usage
PWRM_NOTIFYDONE	The client processed the notification function successfully.
PWRM_NOTIFYNOTDONE	The client must wait for interrupt processing to occur before it can proceed. The client must later call the delayedCompletionFxn specified when this function was registered with PWRM_registerNotify.
PWRM_NOTIFYERROR	Notification cannot be processed. Either an internal client error occurred or the client was notified of an event it could not process. (For V/F setpoint changes, the client registers setpoints it can accommodate to avoid this error.) When a client returns this error, the caller of the PWRM function that triggered the notification receives a PWRM_EFAIL return status.

Constraints and Calling Context

- The notification function should not call PWRM APIs that trigger a notification event (for example, PWRM_sleepDSP). If such an API is called, the PWRM_EBUSY status code is returned.

Example

```
/* notification function prototypes */
PWRM_NotifyResponse myNotifyFxn1(
      PWRM_Event eventType, Arg eventArg1, Arg eventArg2, Arg clientArg);
PWRM_NotifyResponse myNotifyFxn2(
      PWRM_Event eventType, Arg eventArg1, Arg eventArg2, Arg clientArg);

/* ======== myNotifyFxn1 ======== */
PWRM_NotifyResponse myNotifyFxn1(
      PWRM_Event eventType, Arg eventArg, Arg eventArg2, Arg clientArg)
{
#if VERBOSE
   LOG_printf(TRACE, "\nclient #1 notify, PENDINGSETPOINTCHANGE");
   LOG_printf(TRACE, "eventArg=%p, eventArg2=%p", eventArg, eventArg2);
   LOG_printf(TRACE, "clientArg=%p", clientArg);
   LOG_printf(TRACE, "signal notify complete");
#endif
   return(PWRM_NOTIFYDONE);   /* notify complete */
}
```

PWRM_releaseDependency	Release a dependency that was previously declared

Syntax `status = PWRM_releaseDependency(resourceID);`

Parameters
```
Uns resourceID;              /* resource ID  */
```

Return Value
```
PWRM_Status status;          /* returned status */
```

Reentrant Yes

Description This function is the companion to PWRM_setDependency. It releases a resource dependency that was previously set.

Resource IDs are device-specific, and are defined in a PWRM_Resource enumeration in a device-specific header file. For example, see pwrm3430.h for OMAP3430. Additionally, users may declare “user-defined” resources, and reference these as an offset to the last enumerated pre-defined resource. For more information, see the "Number of user-defined resources to support" description in the PWRM Manager Properties section on page 4.

PWRM_ETOOMANYCALLS is returned if you call PWRM_releaseDependency when there are no dependencies currently declared for the specified resource (either because all have been released or because none were set).

PWRM_releaseDependency returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded, and dependency has been released.
PWRM_EINITFAILURE	An error occurred during initialization for user-defined resources.
PWRM_EFAIL	The operation failed while attempting to release the resource.
PWRM_ENOTSUPPORTED	The operation failed because resource tracking is not enabled.
PWRM_EOUTOFRANGE	The specified resourceID is outside the range of valid pre-defined or user-defined resource IDs.
PWRM_ETOOMANYCALLS	A dependency was not previously set and was therefore not released.

Example

```
/* Release dependency upon MCBSP #1 */
PWRM_releaseDependency(PWRM_3430_MCBSP_1);
```

PWRM_resetCPULoadHistory	Clear the CPU load history buffered by PWRM

Syntax `status PWRM_resetCPULoadHistory(sync)`

Parameters
```
Bool sync              /* flag for action when slot finalizes */
```

Return Value
```
PWRM_Status status;    /* returned status */
```

Reentrant Yes

Description PWRM_resetCPULoadHistory is used to clear any accumulated CPU load history maintained by PWRM. All of the history slots maintained in PWRM's internal ring buffer are reset to their "empty" state. (See _PWRM_resetLoad in the PWRM_getCPULoad description.)

Calling PWRM_resetCPULoadHistory does not affect the periodic finalization of history slots by PWRM. The currently accumulating slot finalizes "on schedule" at the same time as if PWRM_resetCPULoadHistory was not called. The sync flag determines what is to be done when the currently accumulating slot completes:

- If sync is TRUE, then the new load accumulated from the time PWRM_resetCPULoadHistory is called until the current time is discarded, and no new history data is available from PWRM until the next slot finalizes.
- If sync is FALSE, then the new load accumulated from the time PWRM_resetCPULoadHistory is called until the current time is stored in the first history slot. This slot is shorter in duration than a "normal" slot (which has the duration statically configured for PWRM). This shorter duration is indicated in the totalCycles field of this slot's PWRM_CPULoadInfo structure.

PWRM_resetCPULoadHistory returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded.
PWRM_EINITFAILURE	The operation failed because there was a failure during load monitor initialization.
PWRM_ENOTSUPPORTED	The operation failed because load monitoring is not enabled.

Example

```
status = PWRM_resetCPULoadHistory(TRUE);
if (status != PWRM_SOK) {
    LOG_printf(TRACE, "Error: status = %x", status);
}
```

PWRM_setDependency	Declare a dependency upon a resource

Syntax `status = PWRM_setDependency(resourceID);`

Parameters `Uns resourceID;` `/* resource ID */`

Return Value `PWRM_Status status;` `/* returned status */`

Reentrant Yes

Description This function sets a dependency on a resource. It is the companion to PWRM_releaseDependency.

Resource IDs are device-specific, and are defined in a PWRM_Resource enumeration in a device-specific header file. For example, see pwrm3430.h for OMAP3430. Additionally, users may declare “user-defined” resources, and reference these as an offset to the last enumerated pre-defined resource. For more information, see the "Number of user-defined resources to support" description in the PWRM Manager Properties section on page 4.

PWRM_setDependency returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded, and dependency has been set.
PWRM_EINITFAILURE	An error occurred during initialization for user-defined resources.
PWRM_EFAIL	The operation failed while attempting to enable the resource.
PWRM_ENOTSUPPORTED	The operation failed because resource tracking is not enabled.
PWRM_EOUTOFRANGE	The specified resourceID is outside the range of valid pre-defined or user-defined resource IDs.

Example

```
/* Declare a driver dependency upon McBSP #1 */
PWRM_setDependency(PWRM_3430_MCBSP_1);
```

PWRM_signalEvent	Signal a power event to registered notification clients

Syntax

```
status = PWRM_signalEvent(eventType, eventArg1, eventArg2,
                notifyTimeout);
```

Parameters

```
PWRM_Event eventType;   /* the power event to be signaled */
Arg eventArg1;          /* event argument #1 */
Arg eventArg2;          /* event argument #2 */
Uns notifyTimeout;      /* max time to wait for notification */
```

Return Value

```
PWRM_Status status;     /* returned status */
```

Reentrant Yes

Description PWRM_signalEvent provides a mechanism for an application to signal power events. PWRM_signalEvent only signals the event occurrence; it does not implement the actual power event processing.

For example, on a platform where V/F scaling is accomplished on a separate processor, PWRM_signalEvent can be called by the DSP application before the scaling to notify registered clients on this processor of the PWRM_PENDINGSETPOINTCHANGE event. Similarly, after the scaling, PWRM_signalEvent can be called to signal the PWRM_DONESETPOINTCHANGE event. Here PWRM is not orchestrating the change to the V/F setpoint, but its registration, notification, and signaling capabilities allow synchronization between scaling by the other processor and the affected software on this processor.

PWRM_signalEvent is intended only for signaling power events that are outside the full control of the PWRM implementation on the current platform. It can be used for V/F scaling notification as described above when PWRM does not implement the actual scaling. For events where PWRM does orchestrate the processing, for example the PWRM_GOINGTORETENTION and PWRM_AWAKEFROMRETENTION events for PWRM_sleepDSP, PWRM handles the notifications automatically, and PWRM_signalEvent must not be used for these events. In other words, PWRM_signalEvent only validates that the indicated eventType is within the range of valid events for the platform; it does not validate whether it makes sense for the application to trigger the signaling of the indicated event.

PWRM_signalEvent returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded.
PWRM_EFAIL	A notification failure occurred.
PWRM_EINVALIDEVENT	The operation failed because eventType is invalid.
PWRM_ETIMEOUT	A registered notification function did not respond within the specified notifyTimeout.

Constraints and Calling Context

- PWRM_signalEvent can be called from a HWI or SWI only if notifyTimeout is 0.

Example

```
status = PWRM_signalEvent(PWRM_DONESETPOINTCHANGE, previousSetpoint, newSetpoint, 0);
if (status != PWRM_SOK) {
    LOG_printf(TRACE, "Error: status = %x", status);
}
```

PWRM_sleepDSP	Transition the DSP to a new sleep state

Syntax

```
status = PWRM_sleepDSP(sleepCode, sleepArg, notifyTimeout);
```

Parameters

```
Uns sleepCode;          /* new sleep state */
LgUns sleepArg;         /* a sleepCode-specific argument */
Uns notifyTimeout;      /* max time to wait for notification */
```

Return Value

```
PWRM_Status status;     /* returned status */
```

Reentrant Yes

Description PWRM_sleepDSP transitions the DSP to a new sleep state.

The sleepCode parameter indicates the new sleep state for the DSP. The following constants may be used on OMAP3430:

Name	Usage
PWRM_RETENTION	The DSP is put into a low-power state in which all memory and logic contents are retained.
PWRM_RETENTIONALT	The DSP is put into a low-power state in which all memory and logic contents are retained. This sleepCode allows the application to define two retention states. For example, one with clocks running and one with clocks stopped.
PWRM_HIBERNATE	The DSP is completely powered off, with context saved before going off, and restored upon power on.
PWRM_STANDBY	The GEM is put into a power-saving mode. Its clock is turned off. This mode has a minimal latency for wakeup.

A successful call to PWRM_sleepDSP returns when the DSP has awoken from the specified sleep state.

The sleepArg parameter may be zero or the address of an array that begins with a control word and is optionally followed by four words that indicate the wakeup conditions for returning from the sleep state to the CPU active state. The control word and wakeup conditions are specified via five masks with the following ordering:

```
Uns control;        /* control bits as defined below */
Uns WUGEN_MEVT0;    /* bits to be set in WUGEN_MEVT0 register */
Uns WUGEN_MEVT1;    /* bits to be set in WUGEN_MEVT1 register */
Uns WUGEN_MEVT2;    /* bits to be set in WUGEN_MEVT2 register */
Uns WUGEN_MEVT3;    /* bits to be set in WUGEN_MEVT3 register */
```

The control word should be initialized to zero. The bits in the control word may be set with the following constants:

- **PWRM_ARGSWUGENMASK.** When set, this bit indicates that the sleepArg array contains four additional words that define the wakeup conditions. If this bit is not set, only the first element of the array (the control word itself) is referenced.
- **PWRM_ARGSNOPLLCONFIG.** When set, this bit indicates that the IVA2 DPLL configuration will not be modified by PWRM_sleepDSP. It is assumed to be configured as needed to support the given sleep code. If this bit is not set, PWRM_sleepDSP will configure the IVA2 DPLL as needed to support the given sleep code.

Prior to activating the sleep state, PWRM_sleepDSP uses the specified masks to set the corresponding bits in the WUGEN_MEVTx registers. Bits that are cleared correspond to events that can wake the DSP from the sleep state. Bits that are set correspond to events that will be blocked as wakeup events by the WUGEN. After CPU wakeup from a sleep state, the WUGEN event mask registers are restored to their original state. Any pending interrupt that was not configured as a wakeup event but had been allowed to propagate prior to calling PWRM_sleepDSP is allowed to pass through once the original event masks are restored.

The notifyTimeout parameter is the maximum amount of time (in system clock ticks) to wait for registered notification functions (set by PWRM_registerNotify) to respond to a delayed completion, before declaring failure and returning PWRM_ETIMEOUT. If the notifyTimeout parameter is zero, then all notification functions must return PWRM_NOTIFYDONE—they cannot request a delayed completion. If a notification function does not return, the system will hang. The notifyTimeout is not used to abandon a notification function; rather it indicates the amount of time PWRM_sleepDSP waits for all delayed completion requests to complete. The wait-loop is entered after all notification functions have been invoked.

PWRM_sleepDSP returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	A successful sleep and wake occurred.
PWRM_EFAIL	A general failure occurred. Could not sleep the DSP.
PWRM_ENOTIMPLEMENTED	The requested sleep mode is not implemented on this platform.
PWRM_ETIMEOUT	A registered notification function did not respond within the specified notifyTimeout.
PWRM_EBUSY	The requested operation cannot be performed at this time; PWRM is busy processing a previous request.

Due to the critical system nature of sleep commands, clients that register for sleep notification should make every effort to respond immediately to the sleep event.

The application should treat return values of PWRM_ETIMEOUT or PWRM_EFAIL as critical system failures. These values indicate the notification client is unresponsive, and the system is in an unknown state.

Constraints and Calling Context

- PWRM_sleepDSP cannot be called from an HWI.
- This API cannot be called from a program's main() function.
- PWRM_sleepDSP can only be called from a SWI if notifyTimeout is 0.

Example

```
#define TIMEOUT   10   /* notification timeout after 10 ticks */
Uns sleepArgs[5];

/* set wakeup conditions */
sleepArgs[0] = PWRM_ARGSWUGENMASK | PWRM_ARGSNOPLLCONFIG;
sleepArgs[1] = 0xFFFFFFBF;       /* enable GP timer 5 as wakeup */
sleepArgs[2] = 0x0000FFFF;       /* no wakeups via WUGEN_MEVT1 */
sleepArgs[3] = 0x000FFFFF;       /* no wakeups via WUGEN_MEVT2 */
sleepArgs[4] = 0x00000001;       /* no wakeups via WUGEN_MEVT3 */

LOG_printf(TRACE, "Putting DSP to retention...");
status = PWRM_sleepDSP(PWRM_RETENTION, (LgUns)(&sleepArgs), TIMEOUT);
LOG_printf(TRACE, "DSP is awake from retention");
LOG_printf(TRACE, "Returned 0x%x", status);
```

PWRM_startCPULoadMonitoring	Start CPU load monitoring

Syntax `status = PWRM_startCPULoadMonitoring();`

Parameters `none`

Return Value `PWRM_Status status;` `/* returned status */`

Reentrant Yes

Description This function starts the collection of CPU load information for the purpose of monitoring the CPU load.

When CPU load monitoring is started, an internal call to PWRM_resetCPULoadHistory() is made. PWRM_setDependency() is also called to declare a dependency on the timer. If timer-based CPU load monitoring is being used, the timer is reloaded and started.

PWRM_startCPULoadMonitoring returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded, and the dependency has been released.
PWRM_EINITFAILURE	An error occurred during initialization for user-defined resources.
PWRM_EFAIL	The operation failed while attempting to release the resource.
PWRM_ENOTSUPPORTED	The operation failed because resource tracking is not enabled.
PWRM_EOUTOFRANGE	The specified resourceID is outside the range of valid pre-defined or user-defined resource IDs.

Example

```
PWRM_startCPULoadMonitoring();
```

PWRM_stopCPULoadMonitoring	Stop CPU load monitoring

Syntax `status = PWRM_stopCPULoadMonitoring();`

Parameters `none`

Return Value `PWRM_Status status;` `/* returned status */`

Reentrant Yes

Description This function stops the collection of CPU load information for the purpose of monitoring the CPU load. While stopped, no calls are made to the configured slot hook function.

If timer-based load monitoring is being used, the timer is stopped. In all cases, an internal call to PWRM_releaseDependency() is made to release the dependency on the timer.

PWRM_stopCPULoadMonitoring returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded, and the dependency has been released.
PWRM_EINITFAILURE	An error occurred during initialization for user-defined resources.
PWRM_EFAIL	The operation failed while attempting to release the resource.
PWRM_ENOTSUPPORTED	The operation failed because resource tracking is not enabled.
PWRM_EOUTOFRANGE	The specified resourceID is outside the range of valid pre-defined or user-defined resource IDs.
PWRM_ETOOMANYCALLS	A dependency was not previously set and was therefore not released.

Example

```
PWRM_stopCPULoadMonitoring();
```

PWRM_unregisterNotify	Unregister for an event notification from PWRM

Syntax `status = PWRM_unregisterNotify(notifyHandle);`

Parameters
```
PWRM_NotifyHandle notifyHandle;     /* handle to function */
```

Return Value
```
PWRM_Status status;                 /* returned status */
```

Reentrant Yes

Description PWRM_unregisterNotify unregisters an event notification that was registered by PWRM_registerNotify. For example, when an audio codec device is closed, it no longer needs to be notified, and must unregister for event notification.

The notifyHandle parameter is the parameter that was provided by PWRM_registerNotify when the function was registered.

PWRM_unregisterNotify returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The function was successfully unregistered.
PWRM_EFAIL	The operation failed due to a memory free failure.
PWRM_EINVALIDHANDLE	The operation failed because notifyHandle is invalid.

Constraints and Calling Context

- This API cannot be called from a program's main() function.

Example

```
PWRM_NotifyHandle notifyHandle1;
PWRM_registerNotify(PWRM_PENDINGSETPOINTCHANGE, ALLSETPOINTSALLOWED,
      (Fxn)myNotifyFxn1, (Arg)0x1111, &notifyHandle1, (Fxn *) &delayFxn1);

PWRM_unregisterNotify(notifyHandle1);
```

PWRM_validateSetpoint	Validate a setpoint versus constraints of registered clients

Syntax `status = PWRM_validateSetpoint(setpoint);`

Parameters
```
Uns  setpoint;       /* setpoint to validate */
```

Return Value
```
PWRM_Status status;  /* returned status */
```

Reentrant Yes

Description PWRM_validateSetpoint can be used to check whether a given setpoint has been declared as not supported during registrations for V/F scaling notifications. If the setpoint does not conflict with any registered constraints then PWRM_SOK is returned.

For example, a serial port driver may only be able to operate at a subset of available setpoints, and it indicates this via the eventMask parameter when it calls PWRM_registerNotify. Application code can quickly check to see if there is any such constraint on a setpoint by calling PWRM_validateSetpoint.

PWRM_validateSetpoint returns one of the following constants as a status value of type PWRM_Status:

Name	Usage
PWRM_SOK	The operation succeeded, no registered V/F scaling notification clients have indicated they cannot support the specified setpoint.
PWRM_EINITFAILURE	V/F scaling support has not been initialized; V/F scaling is unavailable.
PWRM_ENOTSUPPORTED	The operation failed because either: V/F scaling is not enabled, or a registered V/F scaling notification client has indicated it cannot support the specified setpoint.
PWRM_EOUTOFRANGE	The operation failed because the setpoint parameter is out of range of valid setpoints for the platform.

Constraints and Calling Context

- Attempts to call PWRM_validateSetpoint before PWRM_initSetpointInfo is called result in a return code of PWRM_EINITFAILURE. Once PWRM_initSetpointInfo is successfully called, PWRM_validateSetpoint should become functional.

Example

```
#define NUM_SETPOINTS  32
for (i = 0; i < NUM_SETPOINTS; i++) {
    status = PWRM_validateSetpoint(i);
    if (status == PWRM_SOK) {
        LOG_printf(TRACE, "SETPOINT %d is OK", i);
    }
}
```

4 Special Considerations

C64x+ developers must observe the following special considerations and cautions when using the DSP/BIOS Power Manager.

4.1 Sleep Mode Disruption of DSP/BIOS CLK Services

Invoking power-saving sleep modes on some platforms can have a significant impact upon the CLK module's accuracy and correctness. Two examples for OMAP3430 are:

- Putting the DSP into the PWRM_RETENTION state causes the C64x+ timestamp counter to freeze after some hardware transition delays. When the DSP wakes from retention, calls to CLK_gethtime do not account for the cycles that the timestamp counter was frozen. Also, if the CLK module's corresponding timer interrupt is not a wakeup condition, then clock ticks can be missed while the DSP is in retention, so the number of CLK ticks reported by CLK_getltime when the DSP eventually wakes can be in error.
- Putting the DSP into PWRM_HIBERNATE causes the C64x+ timestamp counter to be reset to zero when power to the DSP is turned back on. This means that even for a short hibernate time, the value reported by CLK_gethtime upon wakeup will be less than a value read immediately before going to hibernate. And similar to the retention case, if the CLK module's corresponding timer interrupt is not a wakeup condition for hibernate, then CLK ticks can be missed and not reported by subsequent calls to CLK_getltime.

4.2 Effect of Load Monitoring on IDL Loop Processing

To implement CPU load monitoring, PWRM measures and accumulates the number of CPU cycles spent in the CPU's "IDLE" instruction. This method allows for both energy savings while the CPU is idle waiting for an interrupt, and for a high accuracy measure of true CPU idle time.

The tradeoff for using this technique is that when PWRM idles the CPU, the DSP/BIOS idle (IDL) loop execution is "frozen" until the next CPU interrupt. This means that any other processing placed in the IDL loop does not run continuously, but runs only once per wakeup from the IDLE instruction, before IDLE is invoked again.

For debug environments, when the idle loop is configured to run the RTA data pump to pass data to the debug host, this means that HST channel data will not flow freely, dynamic updating of the real-time analysis data in CCStudio plugins will stall, and updates can appear "choppy" in CCStudio. If this presents a problem during debug, PWRM's CPU load monitoring feature should be disabled until the necessary debugging has been completed or until it is time to deploy the application. Idling by PWRM in the idle loop is ultimately intended for deployed systems, where there is no CCStudio debugger attached.

5 References

The following documents provide information about power management on the C55x platform. While the details of PWRM configuration and APIs are different for the OMAP3430, some of the background information provided in these documents is useful for OMAP3430 applications.

- *TMS320 DSP/BIOS User's Guide* (SPRU423)
- *TMS320C5000 DSP/BIOS API Reference* (SPRU404)

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 TI 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. Information of third parties may be subject to additional restrictions.

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.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
Low Power Wireless	www.ti.com/lpw	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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