# TI-RTOS 2.16 Power Management: MSP432, CC13xx/CC26xx, and CC3200 SimpleLink MCUs

# User's Guide

# Contents

# *Read This First*

## About This Manual

This manual describes the TI-RTOS Power Manager for CC13xx/CC26xx, CC3200, and MSP432 devices. It provides information for application developers and driver developers. The TI-RTOS version number as of the publication of this manual is v2.16.

## Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a special typeface. Examples use a bold version of the special typeface for emphasis.

  Here is a sample program listing:

  ```
  #include <xdc/runtime/System.h>
  int main(void){
      System_printf("Hello World!\n");
      return (0);
  }
  ```

- Square brackets ( [ and ] ) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in a **bold** typeface, do not enter the brackets themselves.

## Trademarks

Registered trademarks of Texas Instruments include Stellaris, and StellarisWare.

Trademarks of Texas Instruments include: the Texas Instruments logo, Texas Instruments, TI, TI.COM, BoosterPack, C2000, C5000, C6000, Code Composer, Code Composer Studio, Concerto, controlSUITE, DSP/BIOS, E2E, MSP430, MSP432, MSP430Ware, OMAP, SimpleLink, SPOX, Sitara, TI-RTOS, Tiva, TivaWare, TMS320, TMS320C5000, TMS320C6000, and TMS320C2000.

ARM is a registered trademark, and Cortex is a trademark of ARM Limited.

Windows is a registered trademark of Microsoft Corporation.

Linux is a registered trademark of Linus Torvalds.

IAR Systems and IAR Embedded Workbench are registered trademarks of IAR Systems AB:

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

February 9, 2016

# *Power Module API*

This chapter provides an overview of the TI-RTOS Power Manager. It starts with a definition of terms, and then summarizes the configuration interfaces and APIs that make up the Power Manager.

## 1.1 Overview

Power management offers significant extension of the time that batteries used to power an embedded application last. However, the application, operating system, and peripheral drivers can be adversely impacted if dynamic power-saving transitions occur when they are performing important operations. To manage such impacts, it is useful to provide power management capabilities for these components to coordinate and safely manage the transitions to and from power saving states.

TI-RTOS includes a Power Manager framework that supports the CC13xx/CC26xx, CC3200, and MSP432 devices. The same top-level APIs, concepts, and conventions are used for all three MCU families.

The same device-level implementation is shared by the CC13xx and CC26xx. File names, function names, and constants for this shared implementation use "CC26XX" as a prefix for both CC13xx and CC26xx devices.

This document provides a summary of the power management APIs, and their relevancy to the different components of the embedded application. It includes chapters with guidelines for developers of both power policies and device drivers.

## 1.2 Definitions / Terms

- **Constraint.** A constraint is a system-level declaration that prevents a specific action. For example, when initiating an I/O transfer, a driver can declare a constraint to temporarily prohibit a transition into a device sleep state. Without this communication to the Power Manager, a decision might be made to transition to a sleep state during the data transfer, which would cause the transfer to fail. After the transfer is complete, the driver releases the constraint it had declared. Constraints are declared with the Power_setConstraint() API, and released with the Power_releaseConstraint() API.

- **Dependency.** A dependency is a declaration by a driver that it depends upon the availability of a particular hardware resource. For example, a UART driver would declare a dependency upon the UART peripheral, which triggers the Power Manager to arbitrate and enable clocks (and power, as necessary) to the peripheral, if not already enabled. A dependency does not prevent specific actions by the Power Manager, for example, transition into a sleep state—constraints are used for that purpose. However, as the Power Manager transitions the device in and out of sleep states, upon wakeup it automatically restores dependencies that were established before the sleep state.

- **Notification.** A notification is a callback mechanism that allows a driver to be notified of specific power transitions or "events". To receive a notification the driver registers in advance, for the specific events it wants to be notified of, with the Power_registerNotify() API. For example, a driver may register to receive both the PowerCC26XX_ENTERING_STANDBY event (to be notified before the device transitions to standby), and the PowerCC26XX_AWAKE_STANDBY event (to be notified after the device has awoken from standby). Note that notifications are strictly that - there is no "voting" at the time the transition is being signaled. If a component is not able to accommodate a particular power transition, it needs to "vote in advance," by setting a constraint.

- **Policy Function.** A function that implements a Power Policy.

- **Power Manager.** The TI-RTOS Power management module (ti.drivers.Power).

- **Power Policy.** A function that makes power saving decisions and initiates those savings with calls to the Power Manager APIs.

- **Reference Policy.** A reference Power Policy provided with TI-RTOS, which aggressively activates power saving states when possible.

- **Sleep State.** A device state where the CPU is inactive and portions of the device are in reduced power-saving states. Sleep states are generally device-specific and may include: clock and clock domain gating, power domain gating, with and without state retention, as well as reduced operating frequencies and voltages.

## 1.3    Power Module API

The Power module API is used at a variety of development levels. In general, drivers are responsible for defining their specific requirements in relation to when power saving states can be used and what actions must be performed before and after use of a power saving state.

- **Application development:** Applications generally enable use of the Power module and otherwise do not use the Power module APIs to a significant extent. This chapter describes the minor changes needed to enable Power module use in Section 1.3.1 and Section 1.3.2.

- **Application Power Policy selection:** The Power Policy determines how aggressive the application will be about putting the device into a power saving state when the Idle thread runs. Chapter 2 describes the provided Power Policy options and how to customize a Power Policy to meet the needs of your application.

- **Driver development:** A device driver may need to take special actions in response to a notification from the Power Manager that the device is going into or coming out of a power saving state, or if the device performance level (MSP432 only) is going to change or has just changed. These actions may include saving registers or re-initializing the peripheral. Chapter 3 describes the process of adding Power module code to a driver, using a DMA-based SPI driver as an example.

### *1.3.1    Static Configuration*

Certain Power Manager features are statically configurable via a Power Manager configuration object defined in the TI-RTOS board file. The elements of the configuration object are device-family specific, and are defined in the relevant Power*.h device-specific header file.

For example, for CC3200, a configuration structure of type PowerCC3200_Config needs to be declared for the application. This structure and its elements are defined in `PowerCC3200.h`. The structure is typically declared in the device-specific file included by the TI-RTOS `Board.h` file, which in this case is `CC3200_LAUNCHXL.c`. If this structure is not included in the application, the application will fail to link.

The configuration object is defined and declared in the following locations:

| Target | Configuration Struct | Defined | Declared | Reference Policy Function |
|--------|----------------------|---------|----------|---------------------------|
| **CC13xx / CC26xx** | PowerCC26XX_Config | PowerCC26XX.h | CC1350STK.c<br>CC2650_LAUNCHXL.c<br>CC2650DK_4XS.c<br>CC2650DK_5XD.c<br>CC2650DK_7ID.c<br>CC2650STK.c<br>etc. | PowerCC26XX_standbyPolicy() |
| **CC3200** | PowerCC3200_Config | PowerCC3200.h | CC3200_LAUNCHXL.c | PowerCC3200_sleepPolicy() |
| **MSP432** | PowerMSP432_Config | PowerMSP432.h | MSP_EXP432P401RLP.c | PowerMSP432_sleepPolicy() |

**Definitions** of configuration objects are made in the header files for each target that are located in the `<tirtos_install_dir>\products\tidrivers_<version>\packages\ti\drivers\power` directory.

**Declarations** of configuration objects are made in board files for a particular target. These are located in the `<tirtos_install_dir>\products\tidrivers_<version>\packages\ti\boards` directory.

Reference policy functions are provided in the `Power<target>_tirtos.c` file located in the `<tirtos_install_dir>\products\tidrivers_<version>\packages\ti\drivers\power` directory.

For configuration details, see the Doxygen-generated documentation. In the top-level installation directory for TI-RTOS, open the Release Notes and follow the links to **Documentation** and then **Documentation Overview**. In the Documentation Overview, open the **TI-RTOS Drivers Runtime APIs (doxygen)** link. Select the **Power.h** link in the "Driver Interfaces" column, along with the device-family specific implementation, for example `PowerCC3200.h`.

This example shows the configuration object elements for CC13xx/CC26xx.

```
const PowerCC26XX_Config PowerCC26XX_config = {
    .policyInitFxn          = NULL,
    .policyFxn              = &PowerCC26XX_standbyPolicy,
    .calibrateFxn           = &PowerCC26XX_calibrate,
    .enablePolicy           = TRUE,
    .calibrateRCOSC_LF      = TRUE,
    .calibrateRCOSC_HF      = TRUE,
```

This example shows the configuration object elements for CC3200:

```
const PowerCC3200_Config PowerCC3200_config = {
    .policyInitFxn          = &PowerCC3200_initPolicy,
    .policyFxn              = &PowerCC3200_sleepPolicy,
    .enterLPDSHookFxn       = NULL,
    .resumeLPDSHookFxn      = NULL,
    .enablePolicy           = false,
    .enableGPIOWakeupLPDS    = true,
    .enableGPIOWakeupShutdown = false,
    .enableNetworkWakeupLPDS = false,
    .wakeupGPIOSourceLPDS    = PRCM_LPDS_GPIO13,
    .wakeupGPIOTypeLPDS      = PRCM_LPDS_FALL_EDGE,
    .wakeupGPIOSourceShutdown = 0,
    .wakeupGPIOTypeShutdown  = 0,
    .ramRetentionMaskLPDS    = PRCM_SRAM_COL_1 | PRCM_SRAM_COL_2 |
                               PRCM_SRAM_COL_3 | PRCM_SRAM_COL_4
};
```

This example shows the configuration object elements for MSP432:

```
const PowerMSP432_Config PowerMSP432_config = {
    .policyInitFxn          = &PowerMSP432_initPolicy,
    .policyFxn              = &PowerMSP432_sleepPolicy,
    .initialPerfLevel       = 2,
    .enablePolicy           = false,
    .enablePerf             = true
};
```

### 1.3.2 Runtime Configuration

There is one runtime configuration option for the Power Manager. For each target, one of the configuration elements of the Power configuration structure (that is, PowerCC3200_Config, PowerCC26XX_Config, or PowerMSP432_Config) is the "enablePolicy" flag. This Boolean determines whether the configured Power policy function is called on each pass through the Idle loop. This flag is typically set to "false" in the TI-RTOS examples. This allows the application to be initially run in a debugger without possible side-effects due to transitions into a low-power state. This is especially critical for CC3200 devices, because sleep transitions usually cause a debugger detach.

A runtime API called Power_enablePolicy() allows the application to explicitly enable the policy at runtime, overriding the setting in the static configuration structure. This allows a common board file to be used for several applications, because individual applications can individually enable the Power policy when appropriate.

### 1.3.3    Include Files

To use the Power API, the application should include both the `Power.h` header file and the appropriate device-specific Power header file. It can then call Power_enablePolicy() in main() or somewhere else in the program. For example, use these statements for CC13xx/CC26xx:

```
#include <ti/drivers/Power.h>
#include <ti/drivers/power/PowerCC26XX.h>

...

Power_enablePolicy();
```

The device-specific header file should be included as shown above, because applications typically use device-specific resource IDs, events, and sleep states.

### 1.3.4    API Functions

For API details, see the Doxygen-generated documentation. In the top-level installation directory for TI-RTOS, open the Release Notes and follow the links to **Documentation** and then **Documentation Overview**. In the Documentation Overview, open the **TI-RTOS Drivers Runtime APIs (doxygen)** link. Select the **Power.h** link in the "Driver Interfaces" column, along with the device-family specific implementation, for example `PowerCC3200.h`.

| **Note:** | If your code was developed using a previous version of the Power module for CC26xx, note that some changes have been made to the APIs. |
| --- | --- |

The following are the Power module APIs.

- **Power_enablePolicy()** enables the configured power policy function to run on each pass through the OS Idle loop. See Section 1.3.2 and Section 2.2.

- **Power_getConstraintMask()** gets a bitmask that identifies the current set of declared constraints. See Section 2.4 and Section 3.1.3.

- **Power_getDependencyCount()** gets the number of dependencies currently declared upon a resource. See Section 3.3.1.

- **Power_getPerformanceLevel()** gets the current performance level for the device. (MSP432 only)

- **Power_getTransitionLatency()** gets the minimal transition latency for a sleep state, in units of microseconds. See Section 2.4.

- **Power_getTransitionState()** gets the current Power module transition state.

- **Power_init()** is a function that needs to be called at startup to initialize the Power Manager state.

- **Power_registerNotify()** registers a function to be called upon a specific power event. See Section 3.1.2, Section 3.2.1, and Section 3.3.3.

- **Power_releaseConstraint()** releases a constraint that was previously set. See Section 3.1.3, Section 3.2.2, and Section 3.3.5.

- **Power_releaseDependency()** releases a dependency that was previously set. See Section 3.1.1, Section 3.2.4, and Section 3.3.6.

- **Power_setConstraint()** sets an operational constraint. See Section 3.1.3, Section 3.2.2, and Section 3.3.2.

- **Power_setDependency()** sets a dependency on a manageable resource. See Section 3.1.1, Section 3.2.1, and Section 3.1.1.

- **Power_setPerformanceLevel()** transitions the device to a new performance level. (MSP432 only)

- **Power_shutdown()** puts the device into a lowest-power shutdown state.

- **Power_sleep()** puts the device into a predefined sleep state. See Section 2.4 and Section 3.1.2.

- **Power_unregisterNotify()** unregisters a function from event notification. See Section 3.3.7.

### 1.3.5 *Instrumentation*

The Power Manager does not log any actions or provide information to the ROV tool.

The Power Manager provides an Assert if Power_releaseConstraint() or Power_releaseDependency() are called more times than the corresponding Power_setConstraint() or Power_setDependency() API. There are also asserts for: an invalid sleepState for Power_sleep(), an invalid shutdownState for Power_shutdown(), and invalid pointers for Power_registerNotify().

## 1.4 Target-Specific Power Conservation

Although the same Power module API can be used across supported targets, different targets support different power conservation states using the Power module.

### 1.4.1 *CC13xx/CC26xx Power Management*

CC13xx/CC26xx supports three sleep states: CPU wait for interrupt (WFI), IDLE_PD (WFI plus CPU domain power gating), and STANDBY (all device power domains powered off). In addition, the power manager implements low-frequency RCOSC (RCOSC_LF) calibration and high-frequency RCOSC (RCOSC_HF) calibration.

By default, RCOSC calibration is enabled. RCOSC calibration can be turned off by modifying the default configuration object as highlighted in bold below:

```
const PowerCC26XX_Config PowerCC26XX_config = {
    .policyInitFxn     = NULL,
    .policyFxn         = &PowerCC26XX_standbyPolicy,
    .calibrateFxn = &PowerCC26XX_noCalibrate, /* default is &PowerCC26XX_calibrate */
    .enablePolicy      = TRUE,
    .calibrateRCOSC_LF = FALSE,              /* default is TRUE */
    .calibrateRCOSC_HF = FALSE,              /* default is TRUE */
```

The following power events are supported:

- PowerCC26XX_ENTERING_STANDBY

- PowerCC26XX_ENTERING_SHUTDOWN

- PowerCC26XX_AWAKE_STANDBY

- PowerCC26XX_AWAKE_STANDBY_LATE

- PowerCC26XX_XOSC_HF_SWITCHED

Note that clients registered for PowerCC26XX_AWAKE_STANDBY are notified just after the domains are powered up. Clients registered for PowerCC26XX_AWAKE_STANDBY_LATE are notified after interrupts have been re-enabled.

The following constraints can be set or unset:

- PowerCC26XX_SB_VIMS_CACHE_RETAIN
- PowerCC26XX_SD_DISALLOW
- PowerCC26XX_SB_DISALLOW
- PowerCC26XX_IDLE_PD_DISALLOW
- PowerCC26XX_NEED_FLASH_IN_IDLE

Details regarding the implementation are provided in the `ti/drivers/power/PowerCC26XX.h` file.

## 1.4.2 CC3200 Power Management

CC3200 supports two sleep states: CPU wait for interrupt (WFI) and Low-Power Deep Sleep (LPDS). The reference power policy looks at the amount of idle time remaining to determine which sleep state it can transition to. It first checks to see if there is enough time to transition into LPDS. If it cannot transition into LPDS (for example, there is not enough time or a constraint has been set prohibiting LPDS), it goes to the CPU wait for interrupt state.

LPDS is the lowest power state that can be used while continuing to maintain the application context (through memory retention) and the networking context to retain any existing WiFi connection. This state has entry-exit latency overheads.

The following power events are supported:

- PowerCC3200_ENTERING_LPDS
- PowerCC3200_ENTERING_SHUTDOWN
- PowerCC3200_AWAKE_LPDS

The following constraints can be set or unset:

- PowerCC3200_DISALLOW_LPDS
- PowerCC3200_DISALLOW_SHUTDOWN

Details regarding the implementation are provided in the `ti/drivers/power/PowerCC3200.h` file.

## 1.4.3 MSP432 Power Management

MSP432 supports Sleep and Deep Sleep states. In addition, the performance level of the target can be set to one of three levels. Setting the performance level changes the target's clock speeds and core voltage levels that trade performance for power conservation.

The following power events are supported:

- PowerMSP432_ENTERING_SLEEP
- PowerMSP432_ENTERING_DEEPSLEEP
- PowerMSP432_ENTERING_SHUTDOWN
- PowerMSP432_AWAKE_SLEEP
- PowerMSP432_AWAKE_DEEPSLEEP
- PowerMSP432_START_CHANGE_PERF_LEVEL

- PowerMSP432_DONE_CHANGE_PERF_LEVEL

The following constraints can be set or unset:

- PowerMSP432_DISALLOW_SLEEP
- PowerMSP432_DISALLOW_DEEPSLEEP_0
- PowerMSP432_DISALLOW_DEEPSLEEP_1
- PowerMSP432_DISALLOW_SHUTDOWN_0
- PowerMSP432_DISALLOW_SHUTDOWN_1
- PowerMSP432_DISALLOW_PERFLEVEL_0
- PowerMSP432_DISALLOW_PERFLEVEL_1
- PowerMSP432_DISALLOW_PERFLEVEL_2
- PowerMSP432_DISALLOW_PERF_CHANGES

Details regarding the implementation are provided in the `ti/drivers/power/PowerMSP432.h` file.

# *Power Policies*

This chapter provides an overview of Power Policy concepts. It includes definitions of terms and the role of a Power Policy. It discusses how to enable and select a specific Power Policy. Reference policies are used to describe key concepts. It concludes with instructions for creating and enabling your own custom Power Policy.

## 2.1    Purpose of a Power Policy

The purpose of a Power Policy is to make a decision regarding power savings when the CPU is idle. The CPU is considered idle when the operating system's Idle loop is executed, when all application threads are blocked pending I/O, or blocked pending some other application event.

To make this decision, the Power Policy should consider factors such as:

- Constraints that have been declared to the Power module, which may disallow certain processor sleep states

- The time until the next OS-scheduled processing

- The transition latency in/out of allowed sleep states

To maximize power savings, the Power Policy should select the deepest power saving state that meets all the considered criteria. The selected power saving state can vary on each execution of the Idle loop, depending upon the changing values of the criteria that are being considered.

Once the Power Policy has decided upon the best allowed power savings, it will either: 1) make a function call to the Power Manager to enact the sleep state, or 2) for lighter saving, with minimal latency, invoke the savings directly (for example, by invoking the processor's native wait for interrupt instruction).

Upon the next interrupt that wakes the CPU, the corresponding interrupt service routine (ISR) will be run as part of wakeup processing, pre-empting execution of the Idle loop. The ISR may perform all the necessary processing, or it may ready an application thread that had been previously blocked. In either case, when all the processing triggered by the interrupt completes, the OS Idle loop runs again, and the Power Policy function resumes execution from the point where interrupts were re-enabled after device wakeup. The Power Policy function will then exit, and then be called again from the OS Idle loop, which will allow it to once again look at criteria and choose a power saving state.

## 2.2    How to Select and Enable a Power Policy

The Power Policy to be used, and whether it should be enabled to run at startup, is specified in the Power Manager configuration structure in the TI-RTOS board configuration file. For example, for CC3200, the relevant elements are highlighted below:

```
/* ======== PowerCC3200_config ======== */
const PowerCC3200_Config PowerCC3200_config = {
    &PowerCC3200_initPolicy,    /* policyInitFxn */
    &PowerCC3200_sleepPolicy,   /* policyFxn */
    NULL,                       /* enterLPDSHookFxn */
    NULL,                       /* resumeLPDSHookFxn */
    0,                          /* enablePolicy */
    1,                          /* enableGPIOWakeupLPDS */
    0,                          /* enableGPIOWakeupShutdown */
    0,                          /* enableNetworkWakeupLPDS */
    PRCM_LPDS_GPIO13,           /* wakeupGPIOSourceLPDS */
    PRCM_LPDS_FALL_EDGE,        /* wakeupGPIOTypeLPDS */
    0,                          /* wakeupGPIOSourceShutdown */
    0,                          /* wakeupGPIOTypeShutdown */
    PRCM_SRAM_COL_1|PRCM_SRAM_COL_2|PRCM_SRAM_COL_3|PRCM_SRAM_COL_4
                                /* ramRetentionMaskLPDS */
};
```

In this example, the Power Policy is the reference "PowerCC3200_sleepPolicy" provided with TI-RTOS. This policy determines the lowest allowed sleep state currently appropriate, and activates that sleep state by calling the Power Manager's Power_sleep() API. If you want to use a derivative of this policy or create your own, you can specify a new function name for policyFxn.

The reference policy performs some initialization at startup, so the "PowerCC3200_initPolicy" is specified for the policyInitFxn. Similar to the power policy function, you can substitute your own policy initialization function. If your policy does not need any initialization, you should specify "NULL" for the policyInitFxn.

Finally, the enablePolicy flag in the configuration structure indicates whether the Power Policy should be invoked on each pass through the OS Idle loop. When starting development of a new application, this element should normally be set to zero (that is, false) to allow easier application startup up and debugging (without the Power Manager opportunistically trying to save power during idle time). Once the application is working, this flag can be set to true to enable power savings by default. Or, as an alternative, the Power_enablePolicy() API can be called (once) at runtime to enable invocation of the policy function on each pass through the Idle loop.

## 2.3    CC13xx/CC26xx Reference Power Policy

For CC13xx/CC26xx, TI-RTOS includes a Power Policy that opportunistically puts the device into STANDBY state during periods of extended inactivity. If the STANDBY state is disallowed because of a constraint or because of inadequate time to transition in/out of STANDBY, the policy selects lighter power savings instead.

The CC13xx/CC26xx reference power policy—named PowerCC26XX_standbyPolicy()—is shown in the following sections to describe concepts and demonstrate a practical implementation of a Policy Function.

Note that this is an aggressive policy, which enacts STANDBY to power off portions of the device whenever possible. Depending upon the application, it may be best to begin application development using a lighter-weight power policy—for example, the Power_doWFI() policy—and then after basic application debugging is complete, enable the aggressive Standby policy.

The Standby policy is implemented in `PowerCC26XX_tirtos.c` in the TI-RTOS release (`<tirtos_install_dir>/products/tidrivers_<version>/packages/ti/drivers/power`). Code snippets are shown in this document for reference.

The first step of the policy is to disable interrupts (step 1) by calling CPUcpsid(). This prevents pre-emption during the decision making process.

The next step is to query the constraints (step 2) that have been declared to the Power module.

In this policy, if either STANDBY or IDLE_PD (power down) are disallowed, the light-weight idling option of simple Wait for Interrupt (WFI) is invoked, using the driverlib PRCMSleep() API (step 3). The goal of this early check is to decide if WFI is the only option as quickly as possible, and when appropriate to go to WFI immediately.

```
/* disable interrupts */
CPUcpsid();

/* query the declared constraints */
constraints = Power_getConstraintMask();

/* do quick check to see if only WFI allowed; if yes, do it now */
if ((constraints &
    ((1 << PowerCC26XX_SB_DISALLOW) | (1 << PowerCC26XX_IDLE_PD_DISALLOW))) ==
    ((1 << PowerCC26XX_SB_DISALLOW) | (1 << PowerCC26XX_IDLE_PD_DISALLOW))) {
    PRCMSleep();
}
```

If the WFI option was not chosen, the next step is to see if there is enough time to transition in/out of STANDBY. The Power_SB_DISALLOW constraint is checked (step 4). If STANDBY is not disallowed, the Clock_getTicksUntilInterrupt() API will be called, to query how many Clock Module tick periods will occur until the next scheduled processing (step 5).

If there is indeed sufficient time to transition in/out of STANDBY, then the policy has now made the decision to go into STANDBY (step 6). However, there will be some latency to wake up the device from STANDBY, to be ready to perform the processing that had been scheduled. To ensure the processor is ready in time to perform the scheduled processing, the policy will schedule an early wakeup event, by starting a Clock object that will cause an early device wakeup, prior to the application-scheduled work.

The Clock module schedules functions to run based upon Clock tick periods, so the number of ticks needed to wakeup early are subtracted from the expected ticks until wakeup, to determine the number of ticks until the early wakeup (step 7). Once this early wakeup time is determined, the policy uses Clock APIs to start a Clock object to trigger the early wakeup (step 8). Note that the Power module provides a pre-created, dedicated Clock object that a Power Policy can use for this purpose. The handle for that Clock object is used in step 8 below.

Now that the early wakeup has been scheduled in the Clock module, the policy calls to the Power module's Power_sleep() API to do the transition into STANDBY (step 9).

```
/* check if any sleep modes are allowed for automatic activation */
else {

    /* check if we are allowed to go to standby */
④  if ((constraints & (1 << PowerCC26XX_SB_DISALLOW)) == 0) {

        /*
         * Check how many ticks until the next scheduled wakeup. A value of
         * zero indicates a wakeup will occur as the current Clock tick
         * period expires; a very large value indicates a very large number
         * of Clock tick periods will occur before the next scheduled wakeup.
         */
⑤      ticks = Clock_getTicksUntilInterrupt();
        /* convert ticks to usec */
        time = ticks * Clock_tickPeriod;

        /* check if can go to STANDBY */
⑥      if (time > Power_getTransitionLatency(PowerCC26XX_STANDBY,
             Power_TOTAL)) {

            /* schedule the wakeup event */
⑦          ticks -= PowerCC26XX_WAKEDELAYSTANDBY / Clock_tickPeriod;
            Clock_setTimeout(Clock_handle(&PowerCC26XX_module.clockObj), ticks);
⑧          Clock_start(Clock_handle(&PowerCC26XX_module.clockObj));

            /* go to standby mode */
⑨          Power_sleep(PowerCC26XX_STANDBY);
⑩          Clock_stop(Clock_handle(&PowerCC26XX_module.clockObj));
⑪          justIdle = FALSE;
        }
    }
```

Once the device has awoken from STANDBY, and the wakeup processing which preempts the policy has completed, the CPU returns to the policy function. At this point (step 10) there is a call to stop the early wakeup Clock event, in case it was not the reason the device exited STANDBY (for example, if a GPIO interrupt awoke the device before the next scheduled processing). The next step is to set `justIdle` to FALSE (step 11), so that the policy function will unwind and return, to enable a fresh evaluation of the sleep criteria at the top of the policy function, the next time it is invoked in the Idle loop.

If the device was not transitioned into Standby, the `justIdle` flag will still be "TRUE", so the alternative code is invoked (below).

The next best option to STANDBY is IDLE_PD, and a check is made (step 12) to see if there is a constraint preventing this.

If IDLE_PD is *not* disallowed, there are a few steps the policy invokes before idling the CPU. Some of the steps performed are simplified here. The policy enables cache retention (step 13), enables the CPU domain to be powered down when deep sleep is activated (step 14), a sync operation is invoked to

ensure settings have propagated to the Always On (AON) domain (step 15), and then a driverlib call is made to invoke CPU deep sleep (step 16). Once the device wakes up, another sync of the AON domain is forced (step 17), and the policy function unwinds to return execution to the Idle loop.

If IDLE_PD *was* disallowed, the policy will simply invoke WFI (with driverlib's PRCMSleep()) (step 18).

```
            /* idle if allowed */
            if (justIdle) {

                /*
                 * power off the CPU domain; VIMS will power down if SYSBUS is
                 * powered down, and SYSBUS will power down if there are no
                 * dependencies
                 * NOTE: if radio driver is active it must force SYSBUS enable to
                 * allow access to the bus and SRAM
                 */
    (12)        if ((constraints & (1 << PowerCC26XX_IDLE_PD_DISALLOW)) == 0) {
    (13)            PRCMRetentionEnable();
    (14)            PRCMPowerDomainOff(PRCM_DOMAIN_CPU);
    (15)            SysCtrlAonSync();

    (16)            PRCMDeepSleep();

                    /* make sure MCU and AON is in sync */
    (17)            SysCtrlAonUpdate();
                }
                else {
    (18)            PRCMSleep();
                }
            }
        }

        /* re-enable interrupts */
(19)    CPUcpsie();
```

Finally, interrupts are re-enabled by the CPUcpsie() call (step 19). Note that if Power_sleep() was called to put the device into STANDBY (step 9), interrupts will be re-enabled within the Power_sleep() API, before late "awake" notifications are sent. So, the wakeup ISR will run at that point within Power_sleep() where interrupts are re-enabled. If lighter sleep is used with the driverlib APIs (step 16 and 18), interrupts will still be disabled when those functions return. So the wakeup ISR won't run until CPUcpsie() is called (step 19).

## 2.4    CC3200 Reference Power Policy

For CC3200, TI-RTOS includes a Power Policy that opportunistically puts the device into a sleep state during periods of extended inactivity. The policy favors the lowest power state that is appropriate. If the lowest state is not permitted (for example, because there is not enough anticipated idle time for that transition, or there is a constraint declared on that sleep state), it will next favor the next deepest power state, and so on. If none of the sleep states are appropriate, as a final option it will invoke the wait for interrupt (WFI) instruction to idle the CPU until the next interrupt.

The CC3200 Sleep policy is implemented in `PowerCC3200_tirtos.c` in the TI-RTOS release (in `<tirtos_install_dir>/products/tidrivers_<version>/packages/ti/drivers/power`). Code snippets are shown in this document for reference.

The  reference policy—named PowerCC3200_sleepPolicy()—is shown in the following sections to describe concepts and show practical implementation of a Policy Function.

Note that this is an aggressive policy; it enacts the lowest power state whenever possible. For the CC3200, Low-Power Deep Sleep (LPDS) is used to power off portions of the device whenever possible. It is best to begin application development with automatic power transitions disabled, and then after basic application debugging is complete, enable the policy with constraints set to permit the lightest sleep state only. Once that is found to be working, progressively release more constraints to allow transitions to deeper sleep states.

The first step of the policy is to disable interrupts (step 1) by calling CPUcpsid(). This prevents pre-emption during the decision making process.

The next step is to disable the TI-RTOS kernel schedulers, with calls to Swi_disable() and Task_disable() (step 2). These disables ensure that if a notification function readies a Swi or Task to run, that the scheduler will not immediately switch context to that new thread. Instead, the switch will be deferred until later, when appropriate, during wakeup and "unwinding" of the sleep state.

Next, Power_getConstraintMask() is called (step 3) to query the constraints that have been declared to the Power Manager.

```
/* disable interrupts */
CPUcpsid();

/* disable Swi and Task scheduling */
swiKey = Swi_disable();
taskKey = Task_disable();

/* query the declared constraints */
constraintMask = Power_getConstraintMask();
```

**1**
**2**
**3**

The next few steps analyze some of the current constraints on power savings. The returned constraintMask is checked (step 4) to see if the LPDS sleep state is allowed.

If LPDS is allowed, we still need to determine if there is enough time to transition into LPDS. The Clock_getTicksUntilTimeout() API is called (step 5) to find how many Clock module tick periods will occur before the next scheduled activity. This tick count is converted to microseconds (step 6), and compared with the device-specific constant for the total transition latency of LPDS (step 7).

```
     /* check if there is a constraint to disallow LPDS */
4    if ((constraintMask & LPDS_DISALLOWED) == 0) {

         /* query Clock for the next tick that has a scheduled timeout */
5       deltaTick = Clock_getTicksUntilTimeout();

         /* convert next tick to units of microseconds in the future */
6       deltaTime = deltaTick * Clock_tickPeriod;

         /* check if there is enough time to transition to/from LPDS */
7       if (deltaTime > PowerCC3200_TOTALTIMELPDS) {
```

If there is sufficient time to transition in and out of LPDS, then the policy has come to the decision to enter LPDS. However, there will be some latency to wake up the device from LPDS to be ready to perform the processing that is scheduled. To ensure that the processor is ready in time to perform the scheduled processing, the policy needs to compute an earlier wakeup time to accommodate the wake latency.

To determine the earlier wakeup time, first the current tick count is obtained with Clock_getTicks() (step 8). An earlier API call (step 5) got the number of future ticks when work was scheduled. However, since the actual tick count for that work is unknown, we got the current tick value and added the delta to that.

To avoid drift over time when computing elapsed ticks based on sampled RTC counts, the policy always references the tick count and RTC count from an initial reference time. These reference values are obtained once, the first time LPDS is activated (step 9). The tick count for the next scheduled work then combines the future tick count from step 5 and the current tick count from step 8 (step 10).

```
         /* decision is now made, going to transition to LPDS ... */
         /* get current tick count */
8       beforeTick = Clock_getTicks();

         /* if this is the first LPDS activation stash initial counts */
9       if (first) {
             firstTick = beforeTick;
             firstRTC = getCountsRTC();
             first = false;
         }

         /* set tick count for next scheduled work */
10      workTick = beforeTick + deltaTick;
```
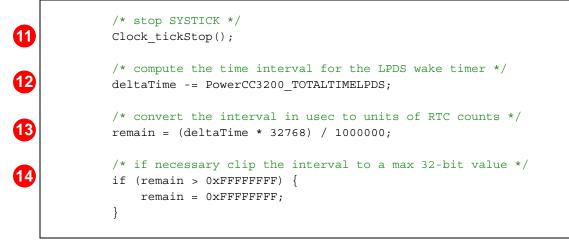
The SYSTICK timer is stopped, but will be started again later (step 11).

The policy then subtracts the latency to allow for LPDS from the future time (in units of microseconds) where there is work to do (step 12), and the interval is converted from microseconds to RTC counts, which is the units for the LPDS interval timer (step 13).

The LPDS interval timer is 32-bits wide, but the calculated RTC counts are 64-bit numbers. So the calculated RTC count is clipped to the maximum 32-bit value of 0xFFFFFFFF to set it to the maximum LPDS interval if the calculated RTC count is larger than the maximum interval (step 14). This is the maximum timed interval that can be specified.

```
        /* stop SYSTICK */
        Clock_tickStop();

        /* compute the time interval for the LPDS wake timer */
        deltaTime -= PowerCC3200_TOTALTIMELPDS;

        /* convert the interval in usec to units of RTC counts */
        remain = (deltaTime * 32768) / 1000000;

        /* if necessary clip the interval to a max 32-bit value */
        if (remain > 0xFFFFFFFF) {
            remain = 0xFFFFFFFF;
        }
```

The wakeup interval is set with the DriverLib API MAP_PRCMLPDSIntervalSet() (step 15). The LPDS interval timer is enabled as an LPDS wakeup source (step 16). Then the Policy calls Power_sleep() to transition the device to LPDS (step 17).

```
        /* set the interval for the LPDS wake timer */
        MAP_PRCMLPDSIntervalSet(remain);

        /* enable the LPDS interval timer as an LPDS wake source */
        MAP_PRCMLPDSWakeupSourceEnable(PRCM_LPDS_TIMER);

        /* now go to LPDS */
        Power_sleep(PowerCC3200_LPDS);
```

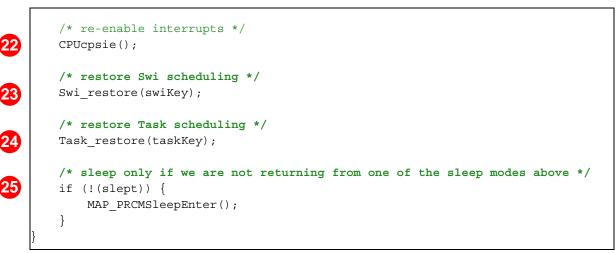Once the device has awoken from LPDS, and the Power_sleep() API returns, the RTC count is obtained (step 18) and a new Clock tick value is calculated (step 19). The Clock module's internal tick count is updated to reflect the elapsed time (step 20). Once the Clock ticks are updated, the Clock module is reconfigured and restarted (step 21).

```
            /* get the RTC count after wakeup */
 18         wakeRTC = getCountsRTC();

            /* calculate new Clock tick value based upon current RTC count:
             *  1. delta RTC = wakeRTC - firstRTC
             *  2. convert delta RTC to delta Clock tick periods
             *  3. new tick = firstTick + delta Clock tick periods
             */
 19       newTick = (((wakeRTC - firstRTC) * 1000000) /
                (32768 * Clock_tickPeriod)) + firstTick;

            /* make sure newTick doesn't exceed soonest scheduled timeout; if it
             * does, clip value used on wakeup to ensure timeout is not skipped */
            if (newTick >= workTick) {
                newTick = workTick - 1;
            }

            /* update Clock's internal tick counter to reflect elapsed time */
 20         Clock_setTicks(newTick);

            /* reconfigure and restart the SYSTICK timer */
 21         Clock_tickReconfig();
            Clock_tickStart();

            /* set a flag to indicate LPDS was invoked */
            slept = TRUE;
        }
    }
```

Next, interrupts are re-enabled (step 22). For LPDS, if the device is awoken by the LPDS interval timer, no ISR will run following wakeup. The Swi and Task schedulers are also restored to their previous states (steps 23 and 24). If the wakeup ISR or a notification function readied a thread to run (for example, if Semaphore_post() is called in the notification function to trigger a Task to run), the thread will run later, once the appropriate scheduler is restored (the Task scheduler for the case of a Semaphore).

When immediate work is completed, and all threads are again blocked, the Idle loop resumes, at the end of the policy function, which returns and allows another pass through the Idle loop (and another invocation of the policy function). If the policy did not attempt LPDS (as indicated by the returnFromSleep flag, step 25), as the lightest sleep option, the policy invokes wait for interrupt via the MAP_PRCMSleepEnter() API. This gates the CPU clock until the next interrupt occurs.

```
        /* re-enable interrupts */
 (22)   CPUcpsie();

        /* restore Swi scheduling */
 (23)   Swi_restore(swiKey);

        /* restore Task scheduling */
 (24)   Task_restore(taskKey);

        /* sleep only if we are not returning from one of the sleep modes above */
 (25)   if (!(slept)) {
            MAP_PRCMSleepEnter();
        }
}
```

## 2.5    MSP432 Reference Power Policy

For MSP432, TI-RTOS includes a Power Policy that opportunistically puts the device into a sleep state during periods of extended inactivity. The policy favors the lowest power state that is appropriate. If the lowest state is not permitted (because there is a constraint declared on that sleep state), it will favor the next deepest power state, and so on. If none of the sleep states are appropriate, as a final option it will invoke the wait for interrupt (WFI) instruction to idle the CPU until the next interrupt.

The default policyFxn for MSP432 is PowerMSP432_sleepPolicy(), in the `PowerMSP432_tirtos.c` file in the `<tirtos_install_dir>`/products/tidrivers_`<version>`/packages/ti/drivers/power directory. The MSP432 reference policyInitFxn is PowerMSP432_initPolicy(). This reference policy is shown in the following sections to describe concepts and show practical implementation of a Policy Function.

Note that this is an aggressive policy; it enacts the lowest power state whenever possible. It is often best to begin application development with automatic power transitions disabled, and then after basic application debugging is complete, enable the policy with constraints set to permit the lightest sleep state only. Once that is found to be working, progressively release more constraints to allow transitions to deeper sleep states.

The first step of the policy is to disable interrupts (step 1) by calling CPUcpsid(). This prevents pre-emption during the decision making process.

The next step is to disable the TI-RTOS kernel schedulers, with calls to Swi_disable() and Task_disable() (step 2). These disables ensure that if a notification function readies a Swi or Task to run, that the scheduler will not immediately switch context to that new thread. Instead, the switch will be deferred until later, when appropriate, during wakeup and "unwinding" of the sleep state.

Next, Power_getConstraintMask() is called (step 3) to query the constraints that have been declared to the Power Manager.

```
/* disable interrupts */
CPUcpsid();

/* disable Swi and Task scheduling */
swiKey = Swi_disable();
taskKey = Task_disable();

/* query the declared constraints */
constraintMask = Power_getConstraintMask();
```

The next step determines whether the constraints prevent the use of the Deep Sleep 1 state, the deepest level (step 4). If this state is allowed, Power_sleep() is called (step 5). When the device returns from Deep Sleep 1, a flag is set to indicate that the device slept (step 6).

```
 /* check if can go to DEEPSLEEP_1 */
if ((constraints & ((1 << PowerMSP432_DISALLOW_SLEEP) |
                    (1 << PowerMSP432_DISALLOW_DEEPSLEEP_0) |
                    (1 << PowerMSP432_DISALLOW_DEEPSLEEP_1))) == 0) {

    /* go to DEEPSLEEP_1 */
    Power_sleep(PowerMSP432_DEEPSLEEP_1);

    /* set 'slept' to true*/
    slept = true;
}
```

The policy next checks to see if the device did not go into Deep Sleep 1 but is allowed to go into Deep Sleep 0 (step 7). If so, it sleeps and sets the flag as with Deep Sleep 1 (step 8).

```
/* if didn't sleep yet, now check if can go to DEEPSLEEP_0 */
if (!slept && ((constraints & ((1 << PowerMSP432_DISALLOW_SLEEP) |
                    (1 << PowerMSP432_DISALLOW_DEEPSLEEP_0))) == 0)) {

    /* go to DEEPSLEEP_0 */
    Power_sleep(PowerMSP432_DEEPSLEEP_0);

    /* set 'slept' to true*/
    slept = true;
}
```

The policy checks to see if the device did not go into either Deep Sleep state but is allowed to go into Sleep state (step 9). If so, it sleeps and sets the flag as with the previous sleep states (step 10).

```
/* if didn't sleep yet, now check if can go to SLEEP */
if (!slept && ((constraints & (1 << PowerMSP432_DISALLOW_SLEEP)) == 0)) {

    /* go to SLEEP */
    Power_sleep(PowerMSP432_SLEEP);

    /* set 'slept' to true*/
    slept = true;
}
```

Next, interrupts are re-enabled (step 11). The Swi and Task schedulers are also restored to their previous states (steps 12 and 13). If the wakeup ISR or a notification function, readied a thread to run (for example, if Semaphore_post() is called in the ISR to trigger a Task to run), the thread will run later, once the appropriate scheduler is restored (the Task scheduler for the case of a Semaphore).

When immediate work is completed, and all threads are again blocked, the Idle loop resumes, at the end of the policy function, which returns and allows another pass through the Idle loop (and another invocation of the policy function). If the policy did not sleep (as indicated by the slept flag), as the lightest sleep option, the policy invokes the wait for interrupt via assembly.

```
/* re-enable interrupts */
CPUcpsie();

/* restore Swi scheduling */
Swi_restore(swiKey);

/* restore Task scheduling */
Task_restore(taskKey);

/* if didn't sleep yet, just do WFI */
if (!slept) {
    __asm(" wfi");
}
}
```

## 2.6 Creating a Custom Power Policy

You may want to write your own Power Policy, for example, to factor application-specific information into the decision process. The provided reference power policies are general policies; they do not consider non-Clock triggered wakeup events. If you want to factor other wakeup events into the policy or add other application-specific criteria, you can do so by creating a custom Power Policy.

You can start with the provided power policy function or start from scratch. Create a new Policy Function, and compile and link the new function into your application. Select your new policy by substituting its name for the "policyFxn" in the Power Manager configuration object, for example, the PowerCC3200_Config object in the CC3200 board file, `CC3200_LAUNCHXL.c`. For example:

```
const PowerCC3200_Config PowerCC3200_config = {
    .policyInitFxn          = &PowerCC3200_initPolicy,
    .policyFxn              = &PowerCC3200_sleepPolicy,
    .enterLPDSHookFxn       = NULL,
    .resumeLPDSHookFxn      = NULL,
    .enablePolicy           = false,
    .enableGPIOWakeupLPDS    = true,
    .enableGPIOWakeupShutdown = false,
    .enableNetworkWakeupLPDS  = false,
    .wakeupGPIOSourceLPDS    = PRCM_LPDS_GPIO13,
    .wakeupGPIOTypeLPDS      = PRCM_LPDS_FALL_EDGE,
    .wakeupGPIOSourceShutdown = 0,
    .wakeupGPIOTypeShutdown   = 0,
    .ramRetentionMaskLPDS    = PRCM_SRAM_COL_1 | PRCM_SRAM_COL_2 |
                               PRCM_SRAM_COL_3 | PRCM_SRAM_COL_4
};
```

By default, the Policy Function is invoked in the operating system's Idle loop, as this is the "natural" idle point for an application. Depending upon the application, the Idle loop may run frequently or infrequently, with a short or long duration before being preempted. So your policy must look at other criteria (besides the fact that the Idle loop is running) to make an appropriate decision.

When the Policy Function is enabled to run in the Idle loop, it will idle the CPU on each pass through the Idle loop—for CC3200, either for LPDS or simple WFI—with the result that any other work the application places in the Idle loop will be invoked only once per idling of the CPU. This may be fine for your application, or you may want to move that other work out of the Idle loop to a higher priority-thread context.

The Policy Function can, in theory, be run from another thread context (if you explicitly call your Policy Function from that thread). But lower-priority threads would be blocked from execution unless the Policy Function routinely decides to not invoke any idling of the CPU.

The Power_getTransitionLatency() API reports the minimum device transition latency to get into/out of a specific sleep state. It does not include any additional latency that may be incurred due to the latency of Power event notifications. So if your application has a significant number of notification clients, or notification latency, you'll want to factor that into the decision for activation of a sleep state.

# Power Management for Drivers

This chapter provides an overview of how a device driver should interact with the TI-RTOS Power Manager. It summarizes the different types of communication between a device driver and the Power Manager. A TI-RTOS SPI driver (for CC3200) is used as an example to illustrate the key function calls. The document concludes with a set of guidelines for the driver writer.

## 3.1 Types of Interaction

A device driver needs to read/write peripheral registers, and usually there is an initial step required to allow CPU access to the peripheral. For example, on a CC3200 device, a peripheral must have its clocks enabled first, otherwise an exception will occur when the CPU tries to access the peripheral. On a CC13xx/CC26xx device, a a peripheral must have its clocks and the relevant power domain enabled.

There are different ways to do this enabling. For example, the driver could write directly to clock and control registers, or it could use DriverLib APIs for this. However, if each driver does this independently, there will be inevitable problems when there are shared clock or power domain control registers. These problems can be avoided by using the Power Manager's APIs, which will properly synchronize and arbitrate the access to shared registers.

Similarly, the Power Manager APIs can be used to properly enable and disable multiple peripherals that share power domains. On the CC13xx/CC26xx, for example, one device driver may be using the GPIO module (which resides in the PERIPH domain) and another may be using the I2S module (also in the PERIPH domain). Suppose the I2S driver is closed and part of cleanup explicitly disables I2S clocks and turns off the PERIPH power domain. When this happens, the GPIO module will immediately cease to function because its power domain was just turned OFF.

### 3.1.1 Set/Release of Dependencies

The Power Manager provides two APIs for drivers to use to enable/disable access to peripherals (generally called "resources"): Power_setDependency() and Power_releaseDependency(). And the Power Manager uses a small device-specific database to represent the resource dependency tree for the device, so that it can arbitrate the enable/disable calls, and know when to properly enable/disable shared resources.

Drivers call Power_setDependency() to enable access to a specific peripheral. If the declaration is the first for the peripheral (that is, it is currently disabled), the Power Manager proceeds to activate the peripheral. The first step is to check to see if there is a "parent" resource. On the CC13xx/CC26xx, for example, the UART peripheral resides in the SERIAL domain, so the SERIAL domain is the "parent". If there is a parent resource, the Power Manager will next check to see if it is activated. If it is not, then the parent will be activated first. For example, for the UART, the SERIAL power domain will be switched ON. After the parent(s) are activated, the "child" resource (for example, the UART peripheral in this case) is activated. And then Power_setDependency() returns to the caller (the driver).

The enable/disable status of each resource is reference counted. So for example, if a dependency is set on a resource, if another active resource shares the same parent resource, that parent resource won't be turned ON again (because it is already ON), but the reference count for the parent resource is incremented.

There is a companion API for drivers to release a dependency and disable a resource: Power_releaseDependency(). This API will appropriately decrement the reference counts for resources, and when those counts reach zero, disable the resource (for both child and parent resources).

Reference counts allow the Power Manager to know precisely when a particular resource (child or parent) should actually be enabled/disabled.

Typically a driver declares its resource needs by calling Power_setDependency() in its "open" function, and releases those resources by calling Power_releaseDependency() in its "close" function. It is critical that the driver writer call these APIs in pairs, to maintain proper reference counts, and to enable the Power Manager to power down resources when they are no longer needed.

### 3.1.2    Registration and Notification

Some power transitions can adversely affect drivers. There is a constraint mechanism (described next) that allows a driver to prohibit certain transitions. For example, disallowing sleep during an active I/O transaction. But in addition to this, when transitions are allowed, there may be need for drivers to adapt to the transitions. For example, if a sleep state causes peripheral register context to be lost, the driver needs to restore that register context once the device is awake from the sleep state.

The Power Manager provides a callback mechanism for this purpose. Drivers register with the Power Manager for notifications of specific transitions they care about. These transitions are identified as power "events". For example, for CC3200, the PowerCC3200_ENTERING_LPDS event is used to signal that a transition to LPDS has been initiated. If a driver needs to do something when the event is signaled, for example, to save some state, or maybe externally signal that the driver will be suspended, it can do this in the callback. Once the Power Manager has notified all drivers that have registered for a particular power event, it will then proceed with the power transition.

The API drivers use to register for notifications is: Power_registerNotify(). With this call a driver specifies the event(s) that it wants to be notified of (one or more events), a callback function (provided by the driver) that the Power Manager should call when the event(s) occurs, an optional event-specific argument, and an arbitrary client argument that can be sent when the callback is invoked.

The callback function is called from the thread context where the power transition was initiated. For example, from the Idle task context, when a Power Policy has made a decision to go to sleep, and has invoked the Power_sleep() API. When the callback function is invoked the driver should take the necessary action, and return from the callback as quickly as possible. The callback function cannot block, or call any operating system blocking APIs. It must return with minimal latency, to allow the transition to proceed as quickly as possible.

Notifications are sent once a decision has been made and a transition is in progress. Drivers cannot "vote" at this point because the transition is in progress. They must take the necessary action, and return immediately.

Typically drivers registers for notifications in the driver's "open" function, and un-register for notifications in the "close" function.

### 3.1.3    Set/Release of Constraints

As described earlier, constraints can be used by drivers to temporarily prohibit certain power transitions, which would otherwise cause a driver to fail to function. The Power Manager provides the Power_setConstraint() API for declaring these constraints, and the Power_releaseConstraint() API to call when the constraint can be lifted.

Constraints are intended to be temporary and dynamic, and only declared when absolutely necessary. Once a constraint is no longer necessary, it should be released, to allow the Power Manager to aggressively reduce power consumption.

Similar to dependencies, constraints are reference counted. So to maintain proper reference counts, it is critical that a driver calls the Power_setConstraint() and Power_releaseConstraint() APIs in pairs.

Note that there is also a Power_getConstraintMask() API that allows a query of a bitmask that represents the currently active constraints. This API is used by a Power Policy when making a decision to go to a particular sleep state. Drivers might use the API to query active constraints, but they should not rely on the fact that a constraint is already raised, and not raise the constraint on their own. (Because another driver may release its constraints at any time.) If a driver has a constraint, it should declare it with Power_setConstraint(), and release it as soon as possible, with Power_releaseConstraint().

## 3.2 Example: CC3200 SPI Driver

This section uses the CC3200 SPI driver to illustrate the interaction between a driver and the Power Manager. The code shown in the following sections focuses on interactions with the Power Manager. Code that the SPI driver uses to perform its read and write action is not shown. See the `SPICC3200DMA.c` file for the full source code.

Section 3.3 then summarizes the concepts in a set of guidelines for the driver writer.

### 3.2.1 SPICC3200DMA_open()

When the SPI driver opens, it first declares a power dependency upon the SPI peripheral, with a call to Power_setDependency() (step 1). Since this driver is using DMA, it also declares a dependency upon DMA (step 2).

**1**
**2**
```
/* Register power dependency - i.e. power up and enable clock for SPI. */
Power_setDependency(object->powerMgrId);

Power_setDependency(PowerCC3200_PERIPH_UDMA);
```

After several other setup activities not related to power, the driver registers its Power notification function—SPICC3200DMA_postNotify()—to be called upon wakeup from LPDS (step 3)

**3**
```
Power_registerNotify(&(object->notifyObj), PowerCC3200_AWAKE_LPDS,
        SPICC3200DMA_postNotify, (uintptr_t)handle);
```
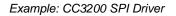
### 3.2.2 SPICC3200DMA_transfer()

When initiating a transfer, the driver declares a constraint to the Power Manager (step 4) by calling Power_setConstraint() to prevent a transition into LPDS during the transfer. Without this constraint, the Power Policy running in the Idle thread might decide to transition the device into a sleep state while a SPI transfer is in progress, which would cause the transfer to fail.

**4**
```
/* Set constraints to guarantee transaction */
Power_setConstraint(PowerCC3200_DISALLOW_LPDS);
```

When the transfer completes in SPICC3200DMA_hwiFxn(), the driver releases the constraint that it had raised previously (step 5). Now the SPI driver is no longer prohibiting sleep states, and the device can be transitioned to sleep if appropriate.

**5**
```
/* Release constraint since transaction is done */
Power_releaseConstraint(PowerCC3200_DISALLOW_LPDS);
```

### 3.2.3  Notification Callback

As shown in Section 3.2.1, in SPICC3200DMA_open() the driver registered for a notification when the device is awoken from LPDS. The notification callback that the driver registered is shown below.

SPI and DMA peripheral registers lose their context during LPDS, so the SPICC3200DMA_postNotify() function calls SPICC3200DMA_initHw() to restore the DMA state and the SPI peripheral state (step 6).

To signal successful completion back to the Power Manager, the notify function returns a status of Power_NOTIFYDONE (step 7):

```
/*
 *  ======== SPICC3200DMA_postNotify ========
 *  This function is called to notify the SPI driver of an ongoing transition
 *  out of LPDS mode. clientArg should be pointing to a hardware module which has
 *  already been opened.
 */
static int SPICC3200DMA_postNotify(unsigned int eventType, uintptr_t eventArg,
        uintptr_t clientArg)
{

    SPICC3200DMA_initHw((SPI_Handle)clientArg);

    return (Power_NOTIFYDONE);
}
```

**(6)** `SPICC3200DMA_initHw((SPI_Handle)clientArg);`

**(7)** `return (Power_NOTIFYDONE);`

### 3.2.4  SPICC3200DMA_close()

When the driver is being closed, it needs to release the dependencies it had declared upon the SPI (step 8) and DMA (step 9).

```
/* Release power dependency on SPI. */
Power_releaseDependency(object->powerMgrId);
Power_releaseDependency(PowerCC3200_PERIPH_UDMA);
```

**(8)** `Power_releaseDependency(object->powerMgrId);`

**(9)** `Power_releaseDependency(PowerCC3200_PERIPH_UDMA);`

It also needs to un-register for notification callbacks (step 10) by calling Power_unregisterNotify().

```
Power_unregisterNotify(&(object->notifyObj));
```

**(10)** `Power_unregisterNotify(&(object->notifyObj));`

## 3.3 Guidelines for Driver Writers

This section summarizes a set of guidelines and steps for enabling a driver to interact with the Power Manager. Notice in Section 3.2 that the amount of code required to enable a driver to use the Power Manager is small.

### 3.3.1 Use Power_setDependency() to enable peripheral access

Before accessing any peripheral registers, call Power_setDependency() specifying the peripheral's resource ID. For example, in the driver's UARTCC3200_open() function:

**Power_setDependency**(PowerCC3200_PERIPH_UARTA0);

This call enables peripheral clocks (for run and sleep states) and powers up the corresponding power domain if it is not already powered.

The Power Manager uses reference counting of all Power_setDependency() and Power_releaseDependency() calls for each resource. It arbitrates access to shared "parent" resources, enabling and disabling them only as needed. It is critical that your driver participate in this arbitration by calling these APIs; if it does not, there will likely be exceptions raised as your application runs.

It is also critical that your driver call Power_setDependency() and Power_releaseDependency() in matched pairs. For example, if Power_setDependency() is called twice for the resource, but Power_releaseDependency() is only called once, the resource remains in an enabled/powered state, when it could and should be disabled/powered down. You can use the Power_getDependencyCount() to get the current number of dependencies set on a resource.

### 3.3.2 Use Power_setConstraint() to disallow power transitions as necessary

If it needs to temporarily prevent a particular power transition, the driver should call Power_setConstraint(). For example, when initiating an un-interruptible I/O transaction, the driver can declare a constraint that the LPDS sleep state cannot be initiated:

**Power_setConstraint**(PowerCC3200_DISALLOW_LPDS);

As soon as the constraint can be lifted, the driver should release the constraint with a call to Power_releaseConstraint(), to enable aggressive power savings.

The Power Manager uses reference counting for constraints, so it is critical that your driver call Power_setConstraint() and Power_releaseConstraint() in matched pairs.

Note that the Power_setContraint() and Power_releaseConstraint() APIs do not "touch" the device clock and power control registers. They simply track and count the declaration and release of constraints. So these APIs can be called from any thread context.

### 3.3.3 Use Power_registerNotify() to register for appropriate power event notifications

If your device driver needs to know about certain power transitions, it should register for notification of the corresponding power events, using the Power_registerNotify() API.

For example, on CC3200 devices, during LPDS, the shared peripheral power domain is powered OFF. The domain is powered back ON upon wakeup. The content of peripheral registers is re-initialized to reset values when the domain is powered back ON. So your device driver may need to save some state

before the device goes into LPDS. If the driver registers for the PowerCC3200_ENTERING_LPDS event, it will receive advance notification of the transition, and can save the critical state data, as well as perform any other steps necessary for preparation for LPDS.

For example, the driver might de-assert an I/O line, which will hold off further communication from a peer on a communication bus, until the device wakes from LPDS, and re-asserts the I/O line. Similarly, the driver probably needs to take some specific action upon wakeup (for example, re-initializing peripheral registers), so it should register for notification for the PowerCC3200_AWAKE_LPDS event. And when that event is signaled, take the necessary action.

If there are multiple instances of a device driver (for example, three active instances of a UART driver), the "clientArg" passed with the Power_registerNotify() call can be used to distinguish different behavior when the notification callback functions are invoked. For example, the first instance of the driver specifies a clientArg of "1":

```
Power_registerNotify(&obj1, PowerCC3200_ENTERING_LPDS |
                     PowerCC3200_AWAKE_LPDS, notifyFxn, 1);
```

The second instance of the driver specifies a clientArg of "2":

```
Power_registerNotify(&obj2, PowerCC3200_ENTERING_LPDS |
                     PowerCC3200_AWAKE_LPDS, notifyFxn, 2);
```

When the PowerCC3200_ENTERING_LPDS event is signaled, the "notifyFxn()" callback will be called twice. For the first driver instance the call is:

```
notifyFxn(PowerCC3200_ENTERING_LPDS, 0, 1);
```

and for the second it is:

```
notifyFxn(PowerCC3200_ENTERING_LPDS, 0, 2);
```

Finally, the device driver should only register for those events that it needs to know about. In other words, there is no need to register for an event that is a "don't care" for the driver. For example, the driver may not need to do anything before a transition into LPDS. If this is the case, it should not register for the PowerCC3200_ENTERING_LPDS event.

### 3.3.4    *Minimize work done in notification callbacks*

Notification callback functions should be minimal functions, in which the driver performs just the necessary steps for a particular power transition, and then returns as quickly as possible.

Callback functions must not call any operating system blocking APIs—for example, Semaphore_pend().

The callback function is called from the context where the Power Manager API was invoked for initiating a particular power transition. So the callback function must be careful if it accesses shared data structures that may be used in different thread contexts.

### 3.3.5    *Release constraints when they are no longer necessary*

When a driver no longer needs to prohibit specific power transitions, it must release the corresponding constraints it declared with Power_setConstraint(). For example, when the driver no longer needs to inhibit LPDS, it calls:

```
Power_releaseConstraint(PowerCC3200_DISALLOW_LPDS);
```

It is critical that drivers use constraints only when necessary, and release the constraints as soon as possible.

The Power Manager uses reference counting for constraints, so it is critical that your driver call Power_setConstraint() and Power_releaseConstraint() in matched pairs.

### 3.3.6 Call Power_releaseDependency() when peripheral access is no longer needed

When a driver no longer requires access to a peripheral it should "release" the peripheral by calling Power_releaseDependency(), specifying the peripheral's resource ID. For example, in the driver's "close" function:

```
Power_releaseDependency(PowerCC3200_PERIPH_DTHE);
```

This call disables peripheral clocks, and if appropriate, powers OFF the corresponding power domain. It is critical that your driver release its dependencies dynamically, to allow the Power Manager to enact aggressive power savings.

The Power Manager uses reference counting of all Power_setDependency() and Power_releaseDependency() calls for each resource. It is critical that your driver call Power_setDependency() and Power_releaseDependency() in matched pairs.

### 3.3.7 Un-register for event notifications with Power_unregisterNotify()

If a driver is closing or otherwise no longer needs notifications, it must un-register its callback with the Power Manager using the Power_unregisterNotify() API. Otherwise, notifications may be sent to the closed driver. For example, to un-register for the events that were previously specified for notifyObj:

```
Power_unregisterNotify(&notifyObj);
```

## IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

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

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 relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have not been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

| Products | | Applications | |
|---|---|---|---|
| Audio | www.ti.com/audio | Automotive and Transportation | www.ti.com/automotive |
| Amplifiers | amplifier.ti.com | Communications and Telecom | www.ti.com/communications |
| Data Converters | dataconverter.ti.com | Computers and Peripherals | www.ti.com/computers |
| DLP® Products | www.dlp.com | Consumer Electronics | www.ti.com/consumer-apps |
| DSP | dsp.ti.com | Energy and Lighting | www.ti.com/energy |
| Clocks and Timers | www.ti.com/clocks | Industrial | www.ti.com/industrial |
| Interface | interface.ti.com | Medical | www.ti.com/medical |
| Logic | logic.ti.com | Security | www.ti.com/security |
| Power Mgmt | power.ti.com | Space, Avionics and Defense | www.ti.com/space-avionics-defense |
| Microcontrollers | microcontroller.ti.com | Video & Imaging | www.ti.com/video |
| RFID | www.ti-rfid.com | | |
| OMAP Mobile Processors | www.ti.com/omap | **TI E2E Community** | e2e.ti.com |
| Wireless Connectivity | www.ti.com/wirelessconnectivity | | |

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