TI-RTOS Power Management for CC3200 SimpleLink Wireless MCUs and MSP432

User's Guide

March 2015
SPRUI18

Contents

Read This First

About This Manual

This manual describes the TI-RTOS Power Manager for CC3200 and MSP432 devices. It provides information for application developers and driver developers.

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

March 20, 2015

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.

Topic **Page**

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 power saving states.

TI-RTOS includes a Power Manager framework that supports the CC3200 and MSP432 devices. The same top-level APIs, concepts, and conventions are used for both of these MCU families.

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 PowerCC3200_ENTERING_DEEPSLEEP event (to be notified before the device transitions to deep sleep), and the PowerCC3200_AWAKE_DEEPSLEEP event (to be notified after the device has awoken from deep sleep). 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 the device sleep 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 modes can be used and what actions must be performed before and after use of a power saving mode.

- **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 sleep 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 sleep state or shutdown state, or if the device performance level 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 TI-RTOS Board file. If this structure is not included in the application, the application will fail to link.

The device-specific configuration items are listed in the Doxygen documentation.

An example showing the configuration object elements for CC3200 is shown below:

```
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 */
};
```

For MSP432 the configuration object is of type PowerMSP432_Config, and there is a different set of elements. See the MSP_EXP432P401RLP.c board file and the PowerMSP432.h file.

1.3.2 Runtime Configuration

In addition to static configuration, there is one runtime configuration option for the Power Manager. For both CC3200 and MSP432, one of the configuration elements of the Power configuration structure (both in PowerCC3200_Config and PowerMSP432_Config) is the "enablePolicy" flag. This Boolean determines whether the configured Power policy function is called on each pass through the Idle loop. By default, this flag is 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.

To call this API, the application should include the Power.h header file, and then call Power_enablePolicy() in main() or somewhere else in the program:

```
#include <ti/drivers/Power.h>
...
Power_enablePolicy();
```

1.3.3 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.
- **Power_getConstraintMask()** gets a bitmask that identifies the current set of declared constraints. See Section 2.3 and Section 3.1.3 for examples.
- **Power_getDependencyCount()** gets the number of dependencies currently declared upon a resource.
- **Power_getPerformanceLevel()** gets the current performance level for the device.
- **Power_getTransitionLatency()** gets the minimal transition latency for a sleep state, in units of microseconds. See Section 2.3.
- **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.

- **Power_shutdown()** puts the device into a lowest-power shutdown state.
- **Power_sleep()** puts the device into a predefined sleep state. See Section 2.3 and Section 3.1.2.
- **Power_unregisterNotify()** unregisters a function from event notification. See Section 3.3.7.

For API details, see the Doxygen-generated documentation. In the top-level installation directory for TI-RTOS, click on the file named `tirtos_docs_overview.html`. Then click the "TI-RTOS Driver Runtime APIs (Doxygen)" link, and select the "Power" link in the "Driver Interfaces" column, along with the device-family specific implementation, for example `PowerCC3200.h`.

1.3.4 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().

1.3.5 Examples

See the *TI-RTOS Getting Started Guide* for your device family for a list of examples that use the Power Manager.

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. A reference policy is used to describe key concepts. It concludes with instructions for creating and enabling your own custom Power Policy.

Topic **Page**

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 at runtime to enable invocation of the policy function on each pass through the Idle loop.

2.3 Reference Power Policy

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

This section describes the CC3200 reference policy in detail, but the reference policy for MSP432 devices is similar. The reference policyFxn for MSP432 is PowerMSP432_policyFxn(), which is provided in the PowerMSP432_tirtos.c file. The MSP432 reference policyInitFxn is PowerMSP432_policyInitFxn()

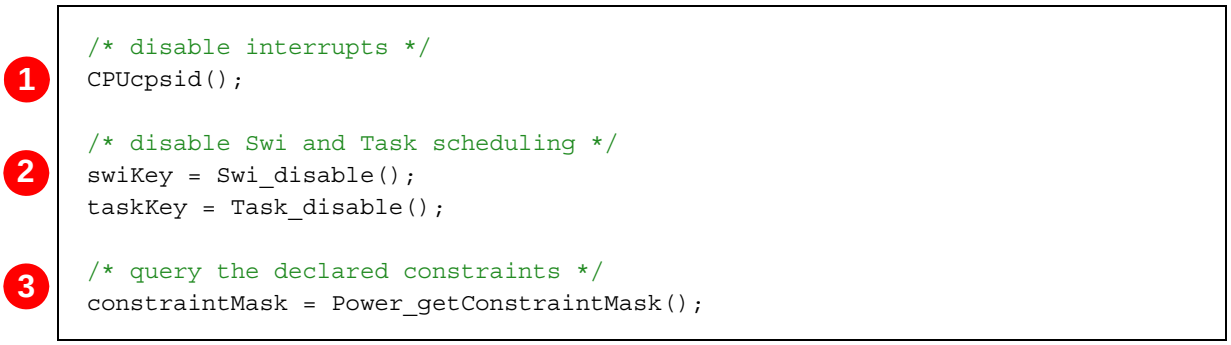
Note that these are aggressive policies; they enact the lowest power mode whenever possible. For the CC3200, low-power deep sleep (LPDS) is used to power off portions of the device whenever possible. As mentioned earlier, it is best to start application development with automatic power transitions disabled, 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 Sleep policy is implemented in `PowerCC3200_tirtos_policy.c` in the TI-RTOS release (`<tirtos_install_dir>/packages/ti/drivers/power/PowerCC3200_tirtos_policy.c`). 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 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 few steps analyze some of the current constraints on power savings. The returned constraintMask is checked (step 4) to see if either the LPDS sleep state or DEEPSLEEP are disallowed.

If neither is disallowed, we still need to determine if there is enough time to transition into LPDS. The Clock_getTicksUntilInterrupt() 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 total transition latency for LPDS as reported by Power_getTransitionLatency() (step 7).

```
/* check to see if either the LPDS sleep state or DEEPSLEEP are disallowed */
if ((constraintMask &
    ((1 << PowerCC3200_DISALLOW_LPDS) |
     (1 << PowerCC3200_DISALLOW_DEEEPLSEEP)))== 0) {

      /* Get the time remaining for the RTC timer to expire */
      ticks = Clock_getTicksUntilInterrupt();

      /* convert ticks to microseconds */
      time = ticks * Clock_tickPeriod;

      /* check if can go to LPDS */
      if (time > Power_getTransitionLatency(PowerCC3200_LPDS, Power_TOTAL)) {
```

If there is indeed 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 the processor is ready in time to perform the scheduled processing, the policy computes an earlier wakeup time, to accommodate the wake latency (step 8). The wakeup interval is set with the DriverLib API MAP_PRCMLPDSIntervalSet() (step 9), the RTC is enabled as an LPDS wakeup source (step 10), and then the Policy calls Power_sleep() to transition the device to LPDS (step 11).

```
          /* get the current and match values for RTC */
          match = MAP_PRCMSlowClkCtrMatchGet();
          curr = MAP_PRCMSlowClkCtrGet();
          remain = match - curr -
              (((uint64_t)PowerCC3200_TOTALTIMELPDS * 32768) / 1000000);

          /* set the LPDS wakeup time interval */
          MAP_PRCMLPDSIntervalSet(remain);

          /* enable the wake source to be timer */
          MAP_PRCMLPDSWakeupSourceEnable(PRCM_LPDS_TIMER);

          /* go to LPDS mode */
          Power_sleep(PowerCC3200_LPDS);

          /* set 'returnFromSleep' to TRUE*/
          returnFromSleep = TRUE;
      }
  }
```

After the checks and entry for LPDS, there is a similar block of code for entry into DEEPSLEEP. Here the constraintMask is checked to see if DEEPSLEEP is disallowed (step 12). The "returnFromSleep" flag is also checked to make sure the processor isn't just returning from LPDS.

The ticks until interrupt (step 13) are converted to microseconds (step 14) and then the transition latency to deep sleep is checked (step 15).

```
/* check if we are allowed to go to DEEPSLEEP */
if ((constraintMask & (1 << PowerCC3200_DISALLOW_DEEEPLSEEP) == 0) &&
    (!returnFromSleep)) {

    /* check how many ticks until the next scheduled wakeup. *?
    ticks = Clock_getTicksUntilInterrupt();

    /* convert ticks to microseconds */
    time = ticks * Clock_tickPeriod;

    /* check if can go to DEEPSLEEP */
    if (time > Power_getTransitionLatency(PowerCC3200_DEEPSLEEP,
        Power_TOTAL)) {
```

At this point, the policy determines there is enough time for DEEPSLEEP, and initiates the transition. Similar to the LPDS case, an early wakeup is setup by the policy to get the processor fully awake in time for the scheduled processing. In this case, since the RTC can directly wake the device from DEEPSLEEP, a Clock object is started to serve as the early wakeup (step 16).

Note that for LPDS, the RTC cannot directly wake the device; instead, a dedicated interval timer is set via the PRCMLPDSIntervalSet DriverLib API. For the case of DEEPSLEEP, the RTC wakeup can be configured directly with the TI-RTOS Kernel's Clock module.

Power_sleep() is called to put the device into DEEPSLEEP (step 17). When the Power_sleep() API has returned, the Clock object is stopped (in case the processor woke for some other reason first) (step 18), and then the returnFromSleep flag is set to indicate DEEPSLEEP was invoked (step 19).

```
        /* schedule the wakeup event */
        ticks -= PowerCC3200_RESUMETIMEDEEPSLEEP / Clock_tickPeriod;
        Clock_setTimeout(Clock_handle(&clockObj), ticks);
        Clock_start(Clock_handle(&clockObj));

        /* go to DEEPSLEEP mode */
        Power_sleep(PowerCC3200_DEEPSLEEP);
        Clock_stop(Clock_handle(&clockObj));

        /* set 'returnFromSleep' to TRUE so we don't go to sleep (below) */
        returnFromSleep = TRUE;
    }
}
```

Once the device has awoken from LPDS or DEEPSLEEP, and the Power_sleep() API returns, or when no sleep action was taken because the necessary criteria were not met, interrupts are re-enabled (step 20). For the case of DEEPSLEEP, it is at this point that the wakeup ISR runs. For LPDS, no ISR will run following wakeup.

Next the Swi and Task schedulers are restored to their previous states (steps 21 and 22). 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 attempt LPDS or DEEPSLEEP above (as indicated by the returnFromSleep flag, step 23), 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.

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

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

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

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

2.4 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 sleep policy is a general policy; it does 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 PowerCC3200_sleepPolicy() 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. For example:

```
const PowerCC3200_Config PowerCC3200_config = {
    &myPolicyInit,   /* policyInitFxn */
    &mySleepPolicy,  /* policyFxn */
...
```

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, DEEPSLEEP, 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.

Topic **Page**

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.

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.

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. 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.

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 earlier) 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 deep 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 CCC3200, 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, 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: 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).

```
/* Register power dependency - i.e. power up and enable clock for SPI. */
Power_setDependency(getPowerMgrId(hwAttrs->baseAddr));

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)

```
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 DEEPSLEEP (or any deeper sleep state) 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.

```
/* Set constraints to guarantee transaction */
Power_setConstraint(PowerCC3200_DISALLOW_DEEEPLSEEP);
```

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.

```
    /* Release constraint since transaction is done */
    Power_releaseConstraint(PowerCC3200_DISALLOW_DEEEPLSEEP);
```

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 restores the DMA state (step 6) and the SPI peripheral state (step 7). To signal successful completion back to the Power Manager, the notify function returns a status of Power_NOTIFYDONE (step 8):

```
/*
 *  ======== 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)
{
    /* Reconfigure the hardware when returning from LPDS */
6   MAP_uDMAEnable();
    MAP_uDMAControlBaseSet(uDMAControlBase);

7   SPICC3200DMA_initHw((SPI_Handle)clientArg);

8   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 9) and DMA (step 10).

```
   /* Release power dependency on SPI. */
9  Power_releaseDependency(getPowerMgrId(hwAttrs->baseAddr));
10 Power_releaseDependency(PowerCC3200_PERIPH_UDMA);
```

It also needs to un-register for notification callbacks (step 11) by calling Power_unregisterNotify().

```
11 Power_unregisterNotify(&object->postNotify);
```

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.

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, sleep, and deep 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 declares a constraint that the DEEPSLEEP sleep state cannot be initiated:

```
Power_setConstraint(PowerCC3200_DISALLOW_DEEEPLSEEP);
```

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, NULL);
```

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

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

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, 1);
```

and for the second it is:

```
notifyFxn(PowerCC3200_ENTERING_LPDS, 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 DEEPSLEEP, it calls:

```
Power_releaseConstraint(PowerCC3200_DISALLOW_DEEEPLSEEP);
```

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. 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