Power Management
for CC26xx SimpleLink Wireless MCUs

User's Guide

October 2014

Texas Instruments

Contents

Preface 3

1 Power Module API 4
- 1.1 Overview 4
- 1.2 Definitions / Terms 5
- 1.3 Power Module API 6
  - 1.3.1 Static Configuration 6
  - 1.3.2 Runtime Configuration 7
  - 1.3.3 APIs 7
  - 1.3.4 Instrumentation 8
  - 1.3.5 Examples 8

2 Power Policies 9
- 2.1 Purpose of a Power Policy 10
- 2.2 How to Select and Enable a Power Policy 10
- 2.3 Provided Standby Policy 11
- 2.4 Creating a Custom Power Policy 14

3 Power Management for Drivers 15
- 3.1 Types of Interaction 16
  - 3.1.1 Set/Release of Dependencies 16
  - 3.1.2 Registration and Notification 17
  - 3.1.3 Set/Release of Constraints 17
- 3.2 Example: UART Driver 18
  - 3.2.1 UART_open() 18
  - 3.2.2 UART_read() 19
  - 3.2.3 UART_write() 20
  - 3.2.4 Notification Callback 21
  - 3.2.5 UART_close() 22
- 3.3 Guidelines for Driver Writers 23
  - 3.3.1 Use Power_setDependency() to enable peripheral access 23
  - 3.3.2 Use Power_setConstraint() to disallow power transitions as necessary 23
  - 3.3.3 Use Power_registerNotify() to register for appropriate power event notifications 23
  - 3.3.4 Minimize work done in notification callbacks 24
  - 3.3.5 Release constraints when they are no longer necessary 24
  - 3.3.6 Call Power_releaseDependency() when peripheral access is no longer needed 25
  - 3.3.7 Un-register for event notifications with Power_unregisterNotify() 25

TEXAS INSTRUMENTS

Preface

October 2014

Read This First

About This Manual

This manual describes the TI-RTOS Power Manager for CC26xx 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

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, 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.

October 31, 2014

TEXAS INSTRUMENTS

Chapter 1

October 2014

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 and APIs that make up the Power module.

Topic **Page**

1.1 Overview **4**

1.2 Definitions / Terms **5**

1.3 Power Module API **6**

1.1 Overview

Power management offers significant extension of the time that batteries used to power an embedded application last. However, operating system and peripheral drivers can be adversely impacted if dynamic power transitions occur when they are performing important operations. Similar problems can occur if a driver does not save data in preparation for moving to a sleep state.

To manage such impacts, it is useful to provide power management capabilities within the operating system, so that the scheduler can manage the transition to a lower-power state.

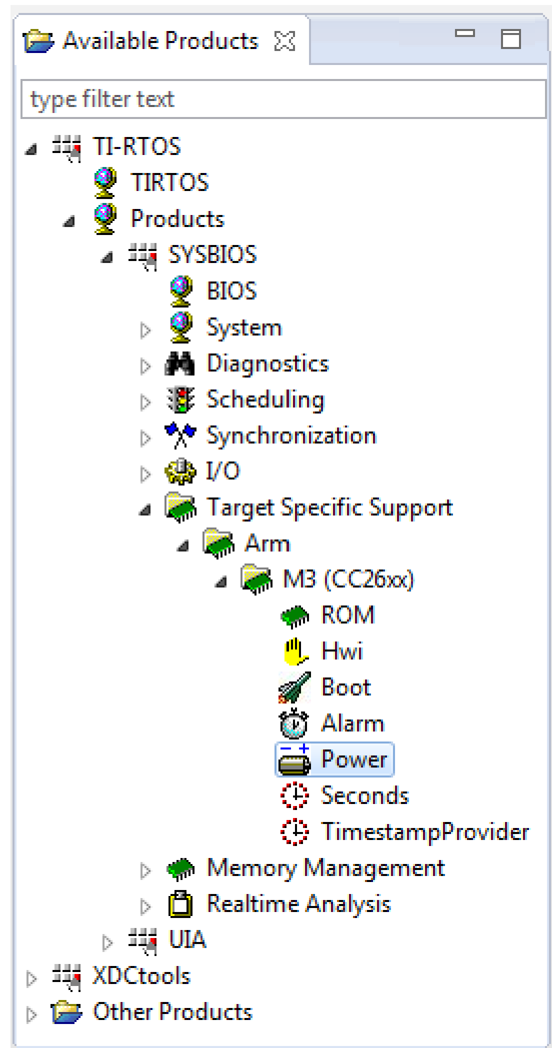
Because power management is very device-specific, the Power modules are provided in the ti.sysbios.family package. In the case of the CC26xx devices, the module is ti.sysbios.family.arm.cc26xx.Power.

The TI-RTOS Power module for CC26xx devices supports both putting the device in standby mode and powering down the CPU during idle time.

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 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) to the peripheral (and domain), 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 will automatically restore 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 Power_ENTERING_STANDBY event (to be notified before the device transitions to STANDBY), and the Power_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 Module (ti.sysbios.family.arm.cc26xx.Power).
- **Power Policy.** A function that makes power saving decisions and initiates those savings with calls to the Power Manager APIs.
- **Standby Policy.** A reference Power Policy provided with TI-RTOS, which aggressively activates the CC26xx STANDBY sleep state when possible.

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 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 CPU in standby or shutdown mode 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 standby or shutdown mode. 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 the UART driver as an example.

1.3.1 *Static Configuration*

In XGCONF, configure an application to use the Power module by expanding the list of Available Products as shown to the right. Select the **TI-RTOS > Products > SYSBIOS > Target Specific Support > Arm > M3 (CC26xx) > Power** module.

To enable messages about this module's activity that feed into the RTOS Object View (ROV) tool, add the following statement to your application's *.cfg file.

```
var Power = xdc.useModule('ti.sysbios.family.arm.cc26xx.Power');
```

You can additionally configure the Power.idle and Power.policyFunc parameters for the Power module to specify the Power Policy you want to use. See Section 2.2 for details about choosing and selecting a Power Policy.

1.3.2 Runtime Configuration

To use the Power module, the following Power header files should be included in an application:

```
#include <ti/sysbios/family/arm/cc26xx/Power.h>
#include <ti/sysbios/family/arm/cc26xx/PowerCC2650.h>
```

An application may then use the Power manager as determined by the drivers it uses, which use the Power module as described in Chapter 3. Or, an application may set constraints to be used by the Power policy. For example, the following constraints disallow powering down the CPU during idle time and switching to standby mode. The application may need to do this during system startup and allow the Power manager to later enable such power saving activities.

```
/* Set constraints for standby and idle power down */
Power_setConstraint(Power_IDLE_PD_DISALLOW);
Power_setConstraint(Power_SB_DISALLOW);
```

See Section 3.1.3, Section 3.2.2, and Section 3.3.2 for information about setting constraints.

1.3.3 APIs

The following are the Power module APIs:

- **Power_getConstraintInfo()** 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 on a resource. See Section 3.3.1.
- **Power_getTicksUntilWakeup()** gets the number of system ticks until the next scheduled wakeup event. See Section 2.3 for an example.
- **Power_getTransitionLatency()** gets the minimal transition latency for a sleep state, in system Clock tick units. See Section 2.3 and Section 2.4.
- **Power_getTransitionState()** gets the current Power module transition state.
- **Power_getXoscStartupTime()** gets the estimated crystal oscillator startup latency, in units of microseconds.
- **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 3.2.5, 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_shutdown()** puts the device in the SHUTDOWN state. See Section 2.3 and Section 3.1.2.
- **Power_sleep()** puts the device in a 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 details, see the reference help system. In CCS, choose **Help > Help Contents** and expand the TI-RTOS for SimpleLink Wireless MCUs item. Choose the Kernel Runtime APIs and Configuration (cdoc) item and expand the tree to find the `ti.sysbios.family.arm.cc26xx.Power` module.

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.

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.

TEXAS INSTRUMENTS

Chapter 2

October 2014

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 Standby 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 10

2.2 How to Select and Enable a Power Policy 10

2.3 Provided Standby Policy 11

2.4 Creating a Custom Power Policy 14

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 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

Power module configuration parameters are used to enable execution of a Power Policy in the Idle loop, and to select which Policy Function is to be invoked. For example, the following statements can be added to the application’s *.cfg file:

```
var Power = xdc.useModule('ti.sysbios.family.arm.cc26xx.Power');
Power.idle = true;
Power.policyFunc = Power.standbyPolicy;
```

When `Power.idle` is set to "true", the Power module will insert a function into the SYS/BIOS Idle loop to run a Power Policy. By default, `Power.idle` is "false", so the developer needs to explicitly enable usage of a Power Policy.

The `Power.policyFunc` parameter specifies the function that implements the Power Policy to be used.

The default `Power.policyFunc` is the Power_doWFI() function, which simply invokes the wait for interrupt (WFI) instruction. This default policy provides CPU power savings with negligible wakeup latency, and negligible impact on application execution.

Much better savings are available with a more aggressive policy, such as the Power_standbyPolicy() function provided with the Power module. This policy is enabled as shown in the configuration statements above.

2.3 Provided Standby Policy

The TI-RTOS release includes a Power Policy that opportunistically puts the device into the 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.

This policy is used in the following sections as a reference to describe concepts and show 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. It is best to start application development using a lighter-weight power policy—for example, the default Power_doWFI() policy—and then after basic application debugging is complete, enable the aggressive Standby policy.

The Standby policy is implemented in `Power_standbyPolicy.c` in the TI-RTOS release (`<`*`tirtos_install_dir`*`>/src/ti/sysbios/family/arm/cc26xx/Power_standbyPolicy.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 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();                                                          (1)

/* query the declared constraints */
constraints = Power_getConstraintInfo();                             (2)

/* do quick check to see if only WFI allowed; if yes, do it now */
if ((constraints & (Power_SB_DISALLOW | Power_IDLE_PD_DISALLOW)) ==
                   (Power_SB_DISALLOW | Power_IDLE_PD_DISALLOW)) {
    PRCMSleep();                                                     (3)
}
```

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 Power_getTicksUntilWakeup() 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 (referenced with Power_Module_State_clockObj()) 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 & Power_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 = Power_getTicksUntilWakeup();

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

            /* schedule the wakeup event */
(7)         ticks -= Power_wakeDelaySTANDBY / Clock_tickPeriod;
            Clock_setTimeout(Power_Module_State_clockObj(), ticks);
(8)         Clock_start(Power_Module_State_clockObj());

            /* go to standby mode */
(9)         Power_sleep(Power_STANDBY, NULL, NOTIFY_LATENCY);
(10)        Clock_stop(Power_Module_State_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 the IDLE_PD mode, 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: it enables the CPU domain to be powered down when deep sleep is activated (step 13), retention is enabled for the VIMS domain (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 & Power_IDLE_PD_DISALLOW) == 0) {

(13)            PRCMPowerDomainOff(PRCM_DOMAIN_CPU);

(14)            PRCMRetentionEnable(PRCM_DOMAIN_VIMS);
(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 "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 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 standbyPolicy 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 Power_standbyPolicy() function or start from scratch. Create a new Policy Function, and compile and link the new function into your application. Select your new policy with the "`Power.policyFunc`" configuration parameter. For example:

```
Power.policyFunc = &myPolicyFunc";
```

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 likely idle the CPU on each pass through the Idle loop, 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 the specified 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.

Texas Instruments

Chapter 3

October 2014

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 UART driver (for CC26xx) 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 16

3.2 Example: UART Driver 18

3.3 Guidelines for Driver Writers 23

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 CC26xx device, a peripheral must have its clocks and the relevant power domain enabled first, otherwise exceptions 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 power control registers, or it could use driverlib APIs for this. However, if each driver does this independently, there will be inevitable problems, because multiple peripherals reside in common power domains, and proper enable/disable of these domains requires synchronization and arbitration.

For example, say one device driver is using the GPIO module (which resides in the PERIPH domain), and another is using the I2S module (also in the PERIPH domain). Say the I2S driver is being closed, and as part of cleanup it 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 will proceed to activate the peripheral. The first step is to check to see if there is a "parent" resource, 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 another peripheral in the SERIAL domain is activated, the SERIAL domain won't be turned ON again (because it is already ON), but the reference count for the SERIAL domain 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 will declare its resource needs by calling Power_setDependency() in its "open" function, and release 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, the Power_ENTERING_STANDBY event is used to signal that a transition to the STANDBY sleep state 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 will be 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 will register 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_getConstraintInfo() 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: UART Driver

This section uses the CC26xx UART 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 UART driver uses to perform its read and write action is not shown. See the UARTCC26XX.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 UART_open()

When the UART driver opens, it first declares a power dependency upon the UART peripheral, with a call to Power_setDependency() (step 1). When Power_setDependency() returns, the UART is enabled for access, and the driver continues with UART driver initialization.

1

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

/* Initialize the UART hardware module */
UARTCC26XX_initHw(handle);
```

After getting a handle to the UART driver and creating internal objects needed to use the driver, the UART_open() function calls Power_registerNotify() (step 2) to register for Power_AWAKE_STANDBY event notification, specifying the uartPostNotify() driver callback function for that event. Now, when the device is awakened after exiting STANDBY, a "post" notification callback is made to uartPostNotify(), which is shown in Section 3.2.4.

2

```
/* Register notification function */
Power_registerNotify(&object->uartPostObj, Power_AWAKE_STANDBY,
                     (Fxn)uartPostNotify, (uint32_t)handle, NULL );

/* UART opened successfully */
Log_print1(Diags_USER1, "UART:(%p) opened", hwAttrs->baseAddr);

/* Return the handle */
return (handle);
```

3.2.2 UART_read()

When initiating a read transaction, the driver declares a constraint to the Power Manager (step 3) by calling Power_setConstraint() to prevent a transition into STANDBY during the transaction. Without this constraint, the Idle thread might decide to transition the device into STANDBY while a UART read is in progress, which would cause the read transaction to fail.

```
    /* Save the data to be read and restore interrupts. */
    object->readBuf = buffer;
    object->readCount = 0;

    Hwi_restore(key);

    /* Set constraint for sleep to guarantee transaction */
3   Power_setConstraint(Power_SB_DISALLOW);
```

When the read transaction completes (step 4), the driver releases the constraint that it had raised previously to prevent STANDBY. Now the UART driver is no longer prohibiting STANDBY, and the device can be transitioned to STANDBY if appropriate.

Note that setting the constraint is critical to allow the UART read transaction to complete uninterrupted. Likewise, the release of the constraint is critical to allow the Power Manager to once again activate aggressive power savings. In the UART driver, Power_releaseConstraint() is called in both the readFinishedDoCallback() callback function and the UARTCC26XX_readCancel() function.

```
static void readFinishedDoCallback(UART_Handle handle)
{
    UARTCC26XX_Object           *object;
    UARTCC26XX_HWAttrs const    *hwAttrs;

    /* Get the pointer to the object and hwAttrs */
    object = handle->object;
    hwAttrs = handle->hwAttrs;

    /* Release power constraint */
4   Power_releaseConstraint(Power_SB_DISALLOW);
```

3.2.3 UART_write()

Similar to the case for UART_read(), the driver asserts a constraint to prevent STANDBY (step 5) by calling Power_setConstraint() when initiating a new write transaction.

```
    /* Save the data to be written and restore interrupts. */
    object->writeBuf = buffer;
    object->writeCount = 0;

    Hwi_restore(key);

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

And it releases the constraint when either the write operation completes normally or the write transaction is canceled.

```
static void writeFinishedDoCallback(UART_Handle handle)
{
    UARTCC26XX_Object           *object;
    UARTCC26XX_HWAttrs const    *hwAttrs;

    /* Get the pointer to the object and hwAttrs */
    object = handle->object;
    hwAttrs = handle->hwAttrs;

    /* Stop the txFifoEmpty clock */
    Clock_stop((Clock_Handle) &(object->txFifoEmptyClk));

    /* Verifies that the FIFO is empty via BUSY flag */
    while(UARTBusy(hwAttrs->baseAddr));

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

3.2.4 Notification Callback

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

UART peripheral registers lose their context during STANDBY (because the SERIAL power domain is turned OFF during STANDBY, and back ON afterwards). So the uartPostNotify() callback reinitializes the UART peripheral by calling the UARTCC26XX_initHW() initialization function again:

```
/*
 *  ======== uartPostNotify ========
 *  This function is called to notify the UART driver of an ongoing transition
 *  out of sleep mode.
 *
 *  @pre    Function assumes that the UART handle (clientArg) is pointing to a
 *          hardware module which has already been opened.
 */
Power_NotifyResponse uartPostNotify(Power_Event eventType, uint32_t clientArg)
{
    /* Reconfigure the hardware if returning from sleep */
    if(eventType == Power_AWAKE_STANDBY) {
        UARTCC26XX_initHw((UART_Handle) clientArg);
    }
    else if(eventType == Power_AWAKE_POWERDOWN) {
        UARTCC26XX_initHw((UART_Handle) clientArg);
        /* If powerdown, IO configuration is lost as well */
        UARTCC26XX_initIO((UART_Handle) clientArg);
    }
    return Power_NOTIFYDONE;
}
```

7

If the device is powered down, the IO configuration is also lost, so UARTCC26XX_initIO() is called.

3.2.5 UART_close()

When the driver is being closed, it needs to release the dependency it had declared upon the UART (step 8) by calling Power_releaseDependency(), so that the UART clocks can be disabled and possibly the SERIAL power domain can be turned off to save power.

It also needs to un-register for notification callbacks (step 9) for STANDBY transitions by calling Power_unregsiterNotify().

```
/* Disable UART */
UARTDisable(hwAttrs->baseAddr);

/* Release power dependency - i.e. potentially power down serial domain. */
Power_releaseDependency(hwAttrs->powerMngrId);

/* Destruct the SYS/BIOS objects. */
Hwi_destruct(&(object->hwi));
Semaphore_destruct(&(object->writeSem));
Semaphore_destruct(&(object->readSem));
Clock_destruct(&(object->txFifoEmptyClk));

/* Mark the module as available */
key = Hwi_disable();
object->opened = false;
Hwi_restore(key);

/* Unregister power notification objects */
Power_unregisterNotify(&object->uartPostObj);

Log_print1(Diags_USER1, "UART:(%p) closed", hwAttrs->baseAddr);
```

8 — Power_releaseDependency

9 — Power_unregisterNotify

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 UART_open() function:

```
Power_setDependency(PERIPH_UART0);
```

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 STANDBY sleep state cannot be initiated:

```
Power_setConstraint(Power_SB_DISALLOW);
```

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 CC26xx devices, during STANDBY mode, most device power domains are powered OFF. The domains will be powered back ON upon wakeup. The content of peripheral registers will be 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 STANDBY. If the driver registers for the Power_ENTERING_STANDBY 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 STANDBY.

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 STANDBY, 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 Power_AWAKE_STANDBY 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, Power_ENTERING_STANDBY |
                            Power_AWAKE_STANDBY, notifyFxn, 1, NULL);
```

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

```
Power_registerNotify(&obj2, Power_ENTERING_STANDBY |
                            Power_AWAKE_STANDBY, notifyFxn, 2, NULL);
```

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

```
notifyFxn(Power_ENTERING_STANDBY, 1);
```

and for the second it is:

```
notifyFxn(Power_ENTERING_STANDBY, 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 STANDBY. If this is the case, it should not register for the Power_ENTERING_STANDBY 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 transitions. 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 STANDBY, it calls:

```
Power_releaseConstraint(Power_SB_DISALLOW);
```

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(PERIPH_CRYPTO);
```

This call will disable peripheral clocks, and if appropriate, power 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
Copyright © 2014, Texas Instruments Incorporated