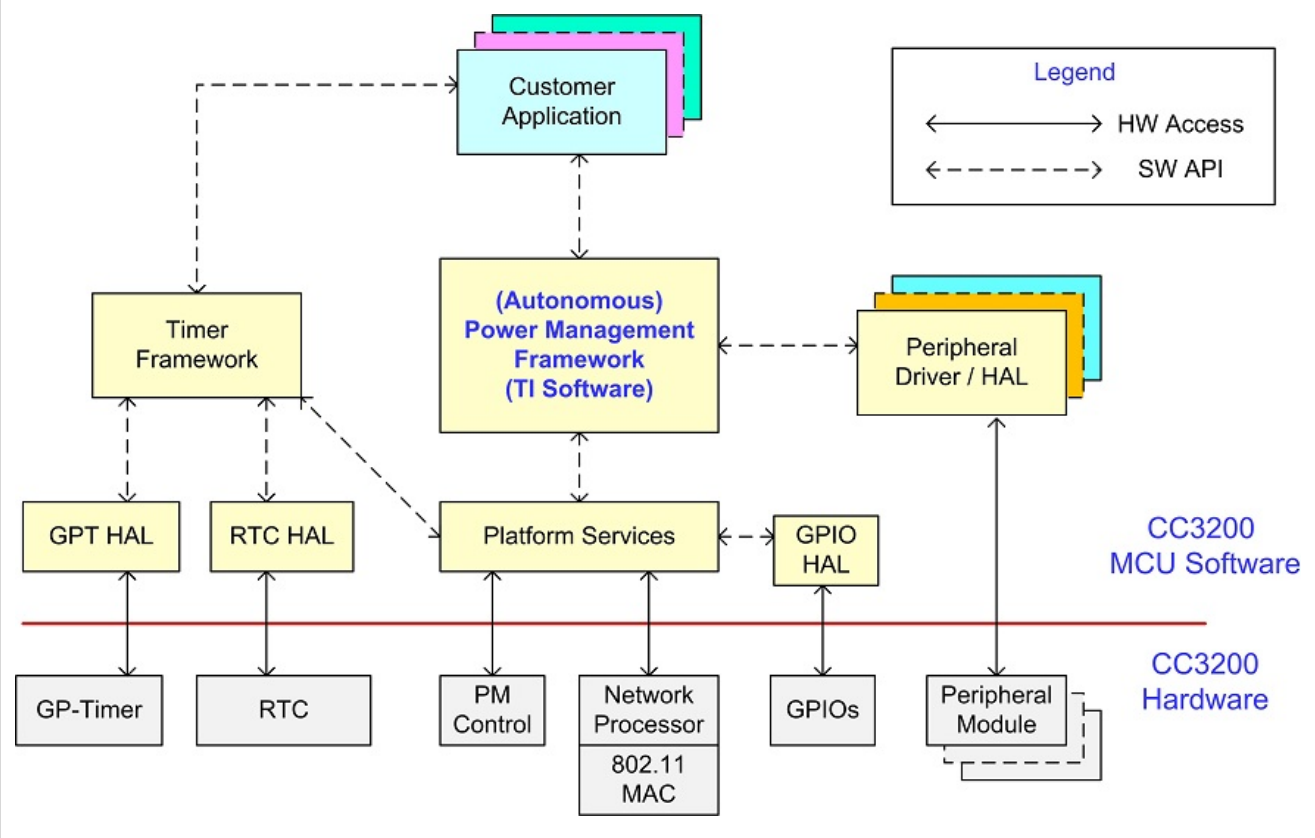
CC3200 Power Management Framework

Introduction

Return to CC31xx & CC32xx Home Page

Optimizing device power consumption is inherently complex and can be overwhelming to application developers. For the CC3200, TI offers a ready-made, deployment agnostic and easy-to-use software component called Power Management Framework (PMF). The PMF offering is intended to assist developers who are eagerly focused on the end-solution to take the system to production and deployment in a short amount of time. To facilitate the application developer's creation of a solution that is ready for end use, the PMF offers services in diverse aspects of the system. The following diagram captures the breadth of the feature scope of the PMF.



The concept of moving the system to the lowest possible configured power state, and then restoring the context of the system without any intervention of the application(s) is the central idea behind PMF.

The subsequent sections provide information about CC3200 power states and the services and API provided by the PMF and its supporting modules.

Device PM overview

The low power modes supported in the CC3200 subsystem are: Sleep, Deep Sleep, Low Power Deep Sleep (LPDS) and Hibernate (HIB).

The diagram below captures the key considerations that a developer needs to take in choosing the power mode suitable for a given use case.

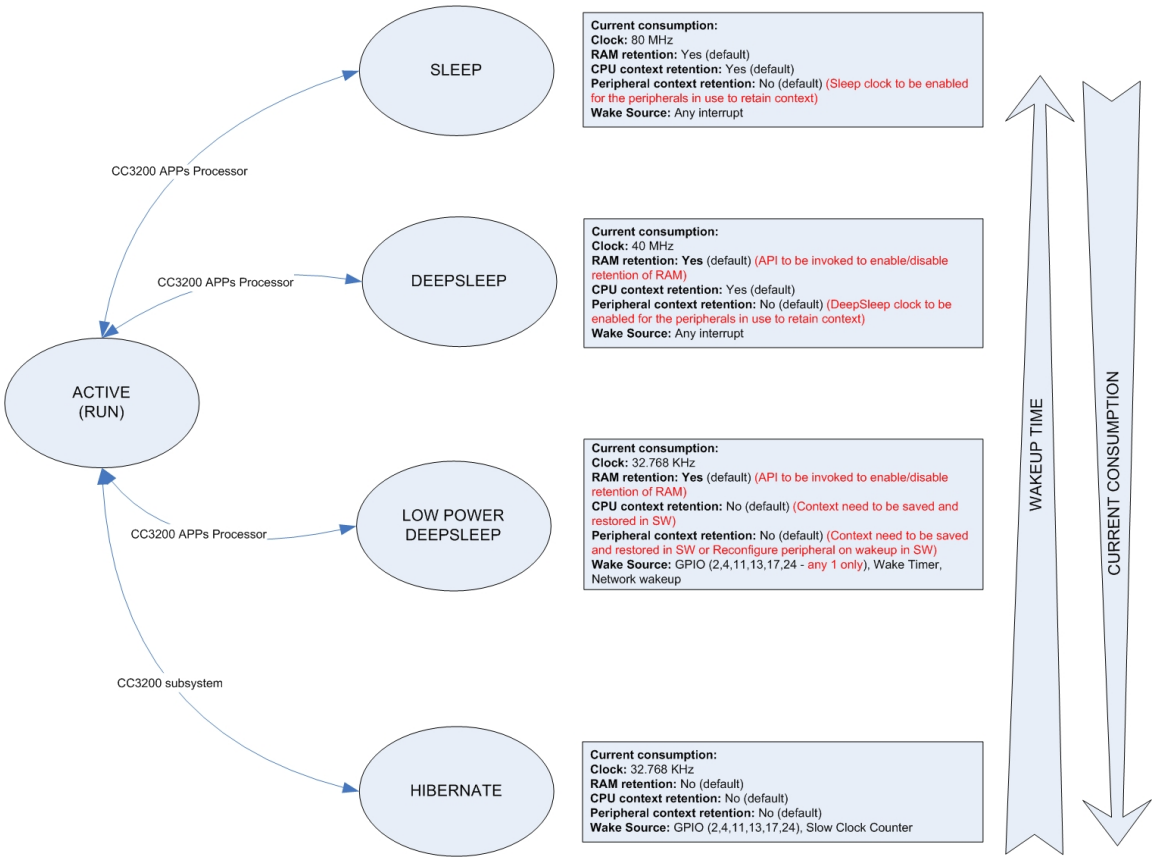
The key points to note are:

1. Sleep: By default, the sleep clock to the peripherals are disabled. If the application chooses to enter sleep anytime and requires certain peripherals to be active (as a source of wakeup), the sleep clock to the peripheral has to be enabled. This is an APPs processor control.

2. Deep Sleep: It is recommended to use Deep Sleep only when using networking services. Using peripherals in conjunction with Deep Sleep is not recommended. This is an APPs processor control.

3. Low Power Deep Sleep (LPDS): The CPU and peripheral context needs to be restored on exit from LPDS. Only 1 out of the 6 GPIOs can be used as a wake source from LPDS. This is an APPs processor control.

4. Hibernate (HIB): This is equivalent to a Power On Reset except for the fact that the SlowClockCOunter continues to tick and a minimal set of registers (two 32 bit registers) are retained across HIB cycles. This is a complete CC3200 subsystem control.



Overview - PMF Design

The objective of the power management framework is to provide an easy to use infrastructure for developers to create a power aware solution. The framework abstracts the inherently complex power management by providing simple services that can be invoked by application developers.

The salient features of the PM framework are:

- Ease of use through intuitive and handful API(s)
- Under the application control selects the lowest power state
- Restores the system context after wake-up
- Clear separation of concerns across apps, peripheral drivers & platform - Enables developers to focus on application and value addition

Software Interface

The programming interface provided by the power management framework is as described in the diagram below.

APIs can be divided into: Core APIs and User hook functions. Key points to note on the APIs are as follows:

1. PM framework initialization APIs - Mandatory

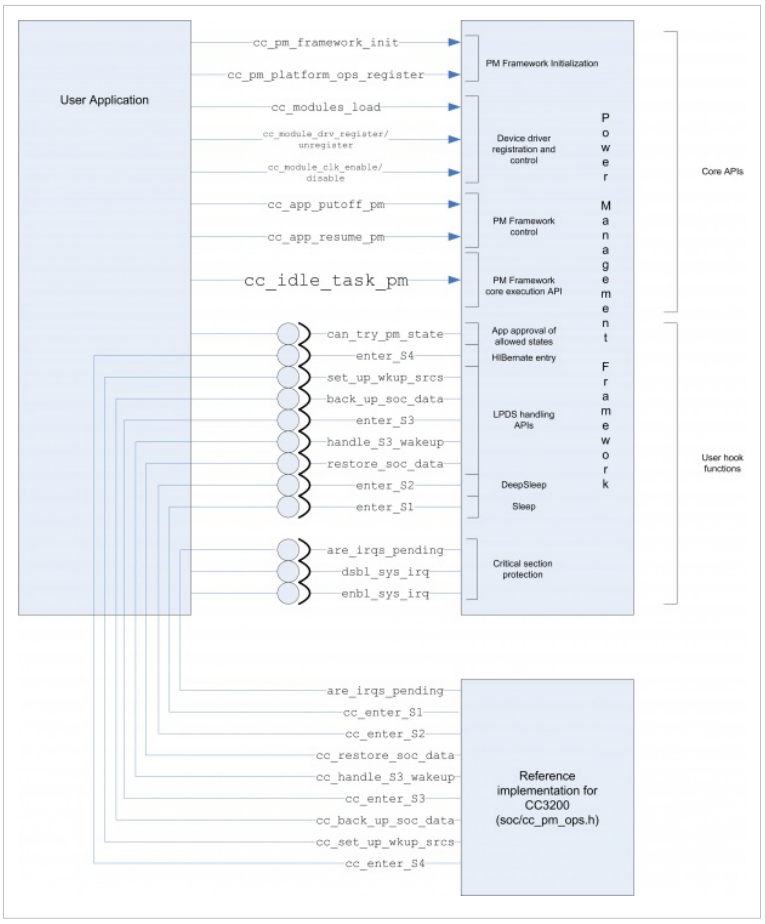
2. Device driver registration and control APIs - These APIs aid the peripheral power state transitions in sync with the system power state transitions.

3. PM framework control: User applications have a choice of controlling the PM framework kicking in (enable/disable) to better control the execution flow. The APIs are reference counted.

4. PM framework core execution API - The API has to be placed in the idle loop of the application. It attempts to enter the low power modes in the order HIB►LPDS►DeepSleep►Sleep. Before entry into HIB, LPDS or DeepSleep, the user callback function "can_try_pm_state" is invoked to get the approval to enter a particular state.

5. User hook functions - A reference implementation of the user hook functions are provided along with the package (middleware\soc\cc_pm_ops). These APIs can be overridden by the applications as deemed fit. The key user hook function to be implemented are:

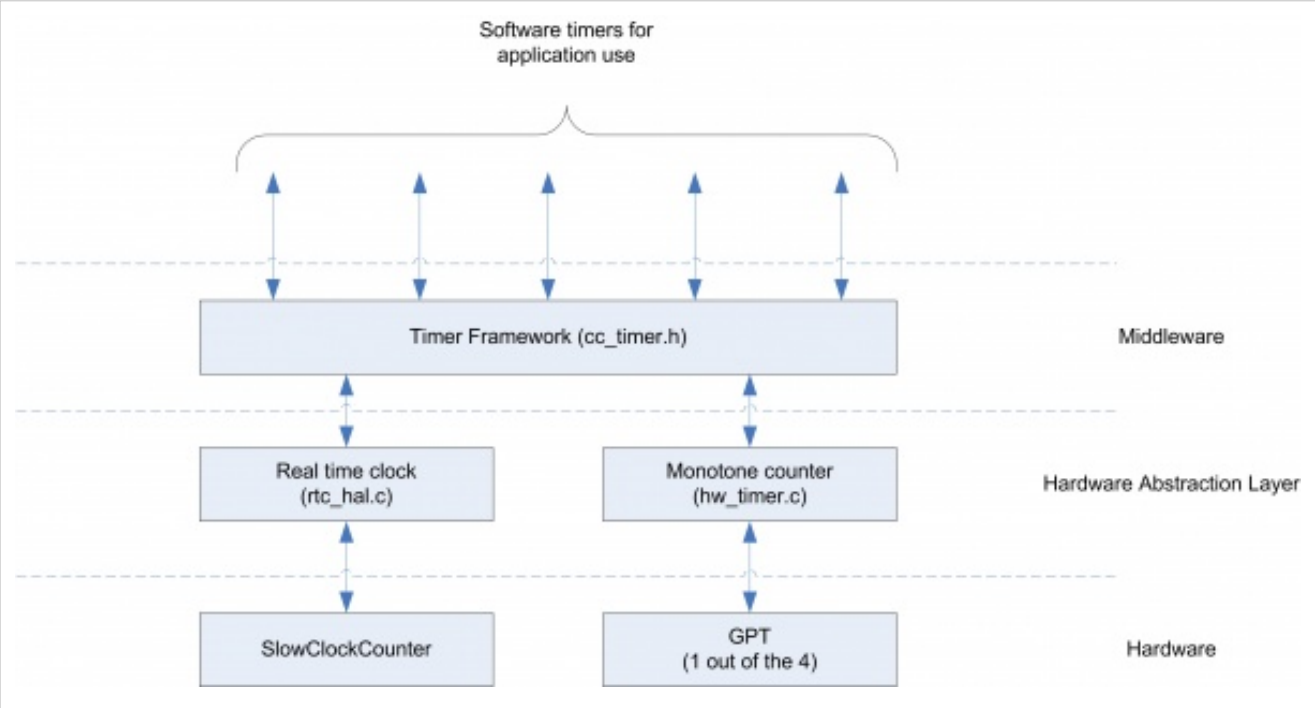
- "can_try_pm_state" - approves the PM framework to enter a particular low power mode.
- "back_up_soc_data" - saves any peripheral context before entering LPDS. Invokes the IO parking as desired (middleware\soc\cc_io_park).
- "restore_soc_data" - restores any peripheral context saved. PinMuxConfig() API has to be reinvoked - to enable clocks to the peripherals.


User Application
cc_pm_framework_init
cc_pm_platform_ops_register
cc_modules_load
cc_module_drv_register/ unregister
cc_module_clk_enable/ disable
cc_app_putoff_pm
cc_app_resume_pm
cc_idle_task_pm
can_try_pm_state
enter_S4
set_up_wkup_srcs
back_up_soc_data
enter_S3
handle_S3_wakeup
restore_soc_data
enter_S2
enter_S1
are_irqs_pending
dsbl_sys_irq
enbl_sys_irq
PM Framework Initialization
Device driver registration and control
PM Framework control
PM Framework core execution API
App approval of allowed states
HIBernate entry
LPDS handling APIs
DeepSleep
Sleep
Critical section protection
Power Management Framework
Core APIs
User hook functions
are_irqs_pending
cc_enter_S1
cc_enter_S2
cc_restore_soc_data
cc_handle_S3_wakeup
cc_enter_S3
cc_back_up_soc_data
cc_set_up_wkup_srcs
cc_enter_S4
Reference implementation for CC3200 (soc/cc_pm_ops.h)

Timer framework

Overview

Timer framework provides a software timer infrastructure built on top of 2 of the hardware resources (Slow clock counter and GeneralPurposeTimer (GPT)) available in the CC3200 device.



Real Time Clock (RTC) Infrastructure : Low precision, Aids power management

- Clock tick at 32.768Khz.
- Uses a countinuously ticking slow clock counter and configures the match values accordingly to realize multiple software timers.
- Ticks across all low power modes (HIB, LPDS, DeepSleep, Sleep, Active).
- Can be used for usecases that need to work with absolute time. This uses up 2 of the available user HIB registers that are saved across HIB cycles. If there is information that needs to be retained across HIB cycles by the application, the NVMEM (SFlash) can be used alternately.
- Mandatory to be used in usecases that need to excercise low power modes.

Monotone Clock infrastructure : High precision

- Clock tick at 80 Mhz.
- Uses a countinuously ticking GPT and configures the match values accordingly to realize multiple software timers.
- Ticks only in Active and Sleep (by enabling the sleep clock) modes.
- Cannot be used in conjunction with other low power modes (HIB, LPDS, DeepSleep) and hence is the applications resposibility to not enter any of these low power modes when the timer is running.

Exercising Low Power Modes

The low power modes discussed in this section are HIBernate and LowPowerDeepSleep.

Exercising low power modes involves

- **Backing up information**

HIB: There are two 32 bit registers that are retained (These are not available if the RTC is enabled). The NVMEM (SFlash) can also be used.

LPDS: CPU and peripheral context need to be saved (in RAM). Also the desired amount of RAM needs to be retained.

- **Setting up the wake sources**

HIB: The wake sources are: PRCM_HIB_SLOW_CLK_CTR, PRCM_HIB_GPIO2, PRCM_HIB_GPIO4, PRCM_HIB_GPIO11, PRCM_HIB_GPIO13, PRCM_

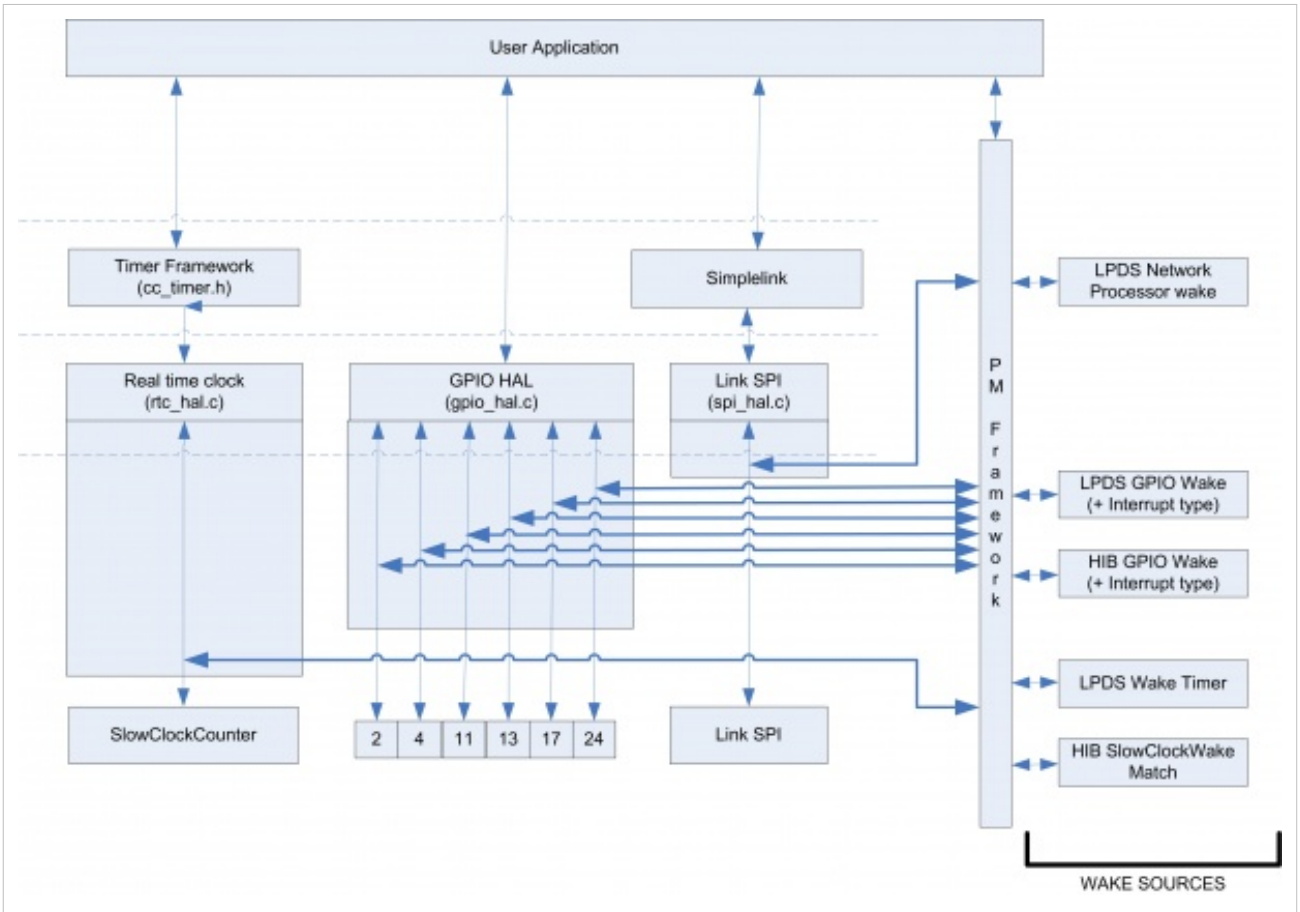
LPDS: The wake sources are: PRCM_HIB_SLOW_CLK_CTR, [PRCM_HIB_GPIO2 or PRCM_HIB_GPIO4 or PRCM_HIB_GPIO11 or PRCM_HIB_GPIO13 or PRCM_HIB_GPIO17 or PRCM_HIB_GPIO24] - any one GPIO at a time only, Network activity based wakeup

- **Entering the low power mode**
- **Restoring the information on wakeup**

HIB: The information can be read back from and restored. PRCMSysResetCauseGet API can be invoked to determine the wake cause.

LPDS: The CPU and peripheral context needs to be restored.

The PM framework abstracts out the information mentioned above and is indirectly derived by the choices the user application. The same are described in the diagram and subsection below.



Low Power Deep Sleep

GPIO as wake source

Applications have to use the gpio_hal.c APIs for the usage of GPIOs. Notifications have to be enabled for the GPIO to be used as a wake source.

```
//
// setting up GPIO as a wk up source and configuring other related
// parameters
//
tGPIOHndl = cc_gpio_open(GPIO_SRC_WKUP, GPIO_DIR_INPUT);
cc_gpio_enable_notification(tGPIOHndl, GPIO_SRC_WKUP, INT_FALLING_EDGE, (GPIO_TYPE_NORMAL | GPIO_TYPE_WAKE_SOURCE));
```

This selection by the application is used by the PM framework to configure the GPIO as a wake source anytime the system can enter LPDS. On wakeup from LPDS, the registered application callback is invoked.

RTC as wake source

Applications have to use the cc_timer.c, rtc_hal.c APIs for the usage of RTC as wake source.

The PM framework internally manages the setting up of LPDS waketimer value before entering LPDS.

```
//
// setting up Timer as a wk up source
//
/* Setup the RTC initial value - This can be obtained from NTP too */
init_time.secs = 0;
init_time.nsec = 0;
cc_rtc_set(&init_time);

/* Create a real time timer */
realtime_timer.source = HW_REALTIME_CLK;
realtime_timer.timeout_cb = timer_intr_hndlr;
realtime_timer.cb_param = NULL;

timer_hndl = cc_timer_create(&realtime_timer);

/* Configure the timer for a periodic wakeup */
interval_time.secs = LPDS_DUR_SEC;
interval_time.nsec = LPDS_DUR_NSEC;
cc_timer_start(timer_hndl, &interval_time, OPT_TIMER_PERIODIC);
```

In the above sequence, after the timer is setup, the PM framework internally manages the entry into LPDS for (interval_time - WAKEUP_TIME_LPDS). Then the timer fires and the application can perform its desired action. This loop continues periodically.

Network activity as wake source

Once the sl_start API is invoked by the application (and HostIRQ enabled), thereafter anytime the APPs processor enters LPDS, the wake on network activity is always enabled.

Hibernate

GPIO as wake source

Applications have to use the gpio_hal.c APIs for the usage of GPIOs. Notifications have to be enabled for the GPIO to be used as a wake source.

```
// setting up GPIO as a wk up source and configuring other related parameters
tGPIOHndl = cc_gpio_open(GPIO_SRC_WKUP, GPIO_DIR_INPUT);
cc_gpio_enable_notification(tGPIOHndl, GPIO_SRC_WKUP, INT_FALLING_EDGE, (GPIO_TYPE_NORMAL | GPIO_TYPE_WAKE_SOURCE));
```

This selection by the application is used by the PM framework to configure the GPIO as a wake source anytime the system can enter HIB. On wakeup from HIB, the application execution resumes from the reset vector.

RTC as wake source

Applications have to use the cc_timer.c, rtc_hal.c APIs for the usage of RTC as wake source.

The PM framework internally manages the setting up of HIB SlowClockMatch value before entering HIB.

```
//
// setting up Timer as a wk up source and other timer configurations
//
/* Setup the RTC initial value - This can be obtained from NTP too */'''
'''
sInitTime.secs = 0;
sInitTime.nsec = 0;
cc_rtc_set(&sInitTime);

/* Create a real time timer */
sRealTimeTimer.source = HW_REALTIME_CLK;
sRealTimeTimer.timeout_cb = NULL;
sRealTimeTimer.cb_param = NULL;
tTimerHndl = cc_timer_create(&sRealTimeTimer);

/* Configure the timer for a absolute wakeup */
sIntervalTimer.secs = HIB_DUR_SEC;
sIntervalTimer.nsec = HIB_DUR_NSEC;
cc_timer_start(tTimerHndl, &sIntervalTimer, OPT_TIME_ABS_VALUE);
```

In the above sequence, after the timer is setup, the PM framework internally manages the entry into HIB for (sIntervalTimer - WAKEUP_TIME_HIB). On wakeup from HIB, the application execution resumes from the reset vector.

IO Parking while entering LPDS

In order to save power (to avoid any leakage currents) when the system enters LPDS, it is recommended to park the IOs appropriately. The application is expected to specify the IO parking scheme. The various options available are:

- DONT_CARE - No parking of the pin
- NO_PULL_HIZ - No pulls, only HiZ the pin
- WEAK_PULL_UP_STD - Weak pullup as well as HiZ
- WEAK_PULL_DOWN_STD - Weak pulldown as well as HiZ
- WEAK_PULL_UP_OD - Weak pullup, Opendrain as well as HiZ
- WEAK_PULL_DOWN_OD - Weak pulldown, Opendrain as well as HiZ

An example of IO parking scheme table is as below:

```
struct soc_io_park cc32xx_io_park[] = {
{PIN_01, "GPIO_10", WEAK_PULL_DOWN_STD},
{PIN_02, "GPIO_11", WEAK_PULL_DOWN_STD},
{PIN_03, "GPIO_12", WEAK_PULL_DOWN_STD},
{PIN_04, "GPIO_13", WEAK_PULL_DOWN_STD},
{PIN_05, "GPIO_14", WEAK_PULL_DOWN_STD},
{PIN_06, "GPIO_15", WEAK_PULL_DOWN_STD},
{PIN_07, "GPIO_16", WEAK_PULL_DOWN_STD},
{PIN_08, "GPIO_17", WEAK_PULL_DOWN_STD},
{PIN_15, "GPIO_22", WEAK_PULL_DOWN_STD},
{PIN_16, "GPIO_23/JTAG_TDI", WEAK_PULL_DOWN_STD},
{PIN_17, "GPIO_24/JTAG_TDO", WEAK_PULL_DOWN_STD},
{PIN_18, "GPIO_28", WEAK_PULL_DOWN_STD},
{PIN_19, "GPIO_28//JTAG_TCK", WEAK_PULL_DOWN_STD},
{PIN_20, "GPIO_29/JTAG_TMS", WEAK_PULL_DOWN_STD},
{PIN_21, "GPIO_25/SOP2", WEAK_PULL_DOWN_STD},
{PIN_45, "DCDC_FLASH_SW_P", WEAK_PULL_DOWN_STD},
{PIN_50, "GPIO_00", WEAK_PULL_DOWN_STD},
{PIN_53, "GPIO_30", WEAK_PULL_DOWN_STD},
{PIN_55, "GPIO_01", WEAK_PULL_DOWN_STD},
{PIN_57, "GPIO_02", WEAK_PULL_DOWN_STD},
{PIN_58, "GPIO_03", WEAK_PULL_DOWN_STD},
{PIN_59, "GPIO_04", WEAK_PULL_DOWN_STD},
{PIN_60, "GPIO_05", WEAK_PULL_DOWN_STD},
{PIN_61, "GPIO_06", WEAK_PULL_DOWN_STD},
{PIN_62, "GPIO_07", WEAK_PULL_DOWN_STD},
{PIN_63, "GPIO_08", WEAK_PULL_DOWN_STD},
{PIN_64, "GPIO_09", WEAK_PULL_DOWN_STD}
};
```

Recommendations for use of PM framework

Choice of low power mode

Some of the key considerations for the choice of the low power mode can be the following (These are generic suggestions only):

- No activity duration

No activity duration - Order of minutes - Choose HIB

No activity duration - Order of seconds - Choose LPDS

No activity duration - Order of milliseconds - Choose Deepsleep/Sleep

- Acceptable lead time on wakeup

HIB - The application is reloaded from the SFLASH. Connection to the AP needs to be re-established. All desired actions must be performed again.

LPDS - On using the PM framework, the CPU context is restored. Reinitialization of peripherals is required. The connection to the AP is retained.

Sleep - Fastest response time with maximum power penalty.

- Retention of context

HIB - All context lost. All desired actions must be performed again. NVMEM or the two 32 bit registers can be used to store information across HIB cycles.

LPDS - Using the power management framework the CPU and registered peripheral context can be retained. All peripheral context need to be restored on exit from LPDS. Desired

amount of RAM can be retained (in multiples of 64 KB).

Choice of wake source

The choice of wake source is dependent on the application usecase. Summarizing the wake sources below:

1. Sleep - Any interrupt

2. Deepsleep - Any interrupt

3. LowPowerDeepSleep - Waketimer (specific counter that ticks only during LPDS), Any 1 GPIO (2,4,11,13,17,24), Network activity based wakeup - HostIRQ (This is in response to an API issued by the application before entering LPDS (for ex., sl_recv()))

4. HIBernate - SlowClockCounter (match value is set, the slow clock counter once started in PRCMCC3200MCUInit, continues to tick), Any combination of GPIOs (2,4,11,13,17,24)

Note: In case of HIBernate, the source of the wakeup cannot be determined.

Working with time of day (real time)

The time of day information can be obtained using NTP. This information can be passed to the timer framework and then can be used as the basis of having timed events based on time of day. Many usecases can be realized using this feature (alarms...).

Working with time of day requires setting up of the initial RTC time. Thereafter, the framework maintains the synchronization between the Real time and the free running SlowClockCounter.

The code could be like below:

```
//
// setting up Timer as a wk up source
```

```
//
/* Setup the RTC initial value obtained from NTP */
init_time.secs = NTP_TIME_SECS;
init_time.nsec = NTP_TIME_NSECS;
cc_rtc_set(&init_time);

/* Create a real time timer */
realtime_timer.source = HW_REALTIME_CLK;
realtime_timer.timeout_cb = timer_intr_hndlr;
realtime_timer.cb_param = NULL;
timer_hndl = cc_timer_create(&realtime_timer);

/* Configure the timer for a periodic wakeup */
interval_time.secs = WAKEUP_SEC_TIMEOFDAY;
interval_time.nsec = WAKEUP_NSEC_TIMEOFDAY;
cc_timer_start(timer_hndl, &interval_time, OPT_TIMER_PERIODIC);
```

There precision of the timer is only in the order of milliseconds. The input of nanoseconds is retained to be able to pass the values obtained over NTP to the API directly.

Note:

- The two 32bit HIB registers are used to maintain the mapping of initial timeofday with the slow clock counter ticks and hence are not available for application use.
- The APIs donot take care of the timezone correction (to be applied to the GMT time obtained over NTP). The timezone correction is expected to be handled by the application before initializing the RTC.

Disabling PM framework

The APIs to control the enabling and disabling of PM framework are: cc_app_resume_pm and cc_app_putoff_pm. These APIs are reference counted and the resumption of PM framework happens accordingly.

The intervention from the PM framework can be controlled by using these APIs. An example is as below:

```
SELECT_LPDS_LOW_POWER_MODE();


while(FOREVER){
      /* Resume the PM framwork operations before entering the Idle wait in the application */
        cc_app_resume_pm();
        /* Enter the Wait for registered event (for ex., sl_recv) */
        WAIT_HERE();  /* --------- Application will enter LPDS here on executing IDLE loop ----*/
        /* Put off the PM framework the process the event */
        cc_app_putoff_pm();

        /* --------- Application will remain ACTIVE here ----*/
        ..... /* Processing */
}
```

Extending/Customizing PM framework

A reference implementation of the user hook functions have been provided in the file cc3200-sdk\middleware\soc\cc_pm_ops.c. While this may suffice for most of the implementations, the applications can choose to override this implementation by hooking an alternate function.

The default user hook registration function will be as below:

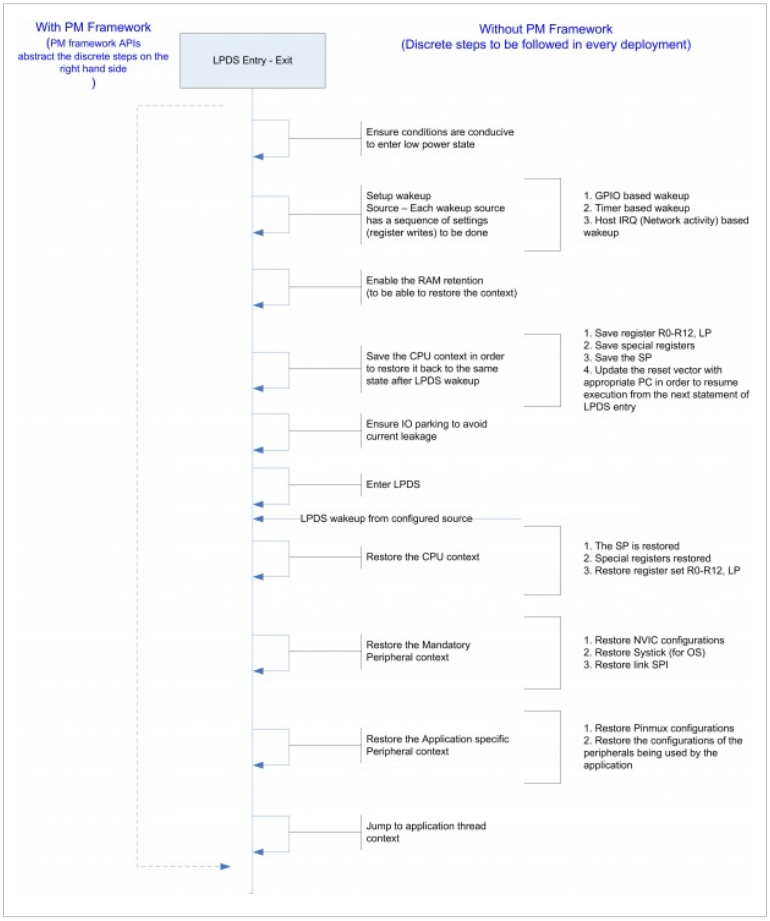
```
lp3p0_pm_ops->set_up_wkup_srcs = cc_set_up_wkup_srcs;
lp3p0_pm_ops->handle_S3_wakeup = cc_handle_S3_wakeup;
lp3p0_pm_ops->are_irqs_pending = cc_are_irqs_pending;
lp3p0_pm_ops->can_try_pm_state = lp3p0_can_try_pm_state;
lp3p0_pm_ops->enter_S4 = cc_enter_S4;
lp3p0_pm_ops->enter_S3 = cc_enter_S3;
lp3p0_pm_ops->enter_S2 = cc_enter_S2;
lp3p0_pm_ops->enter_S1 = cc_enter_S1;
lp3p0_pm_ops->back_up_soc_data = lp3p0_back_up_soc_data;
lp3p0_pm_ops->restore_soc_data = lp3p0_restore_soc_data;
lp3p0_pm_ops->dsbl_sys_irq = osi_EnterCritical;
lp3p0_pm_ops->enbl_sys_irq = osi_ExitCritical;
```

The application is expected to implement the following functions consciously:

- can_try_pm_state - approval of the entry into low power state
- back_up_soc_data - save any information (peripheral) if required before entering LPDS
- restore_soc_data - restore peripheral context (or reinitialize peripherals) on exiting LPDS

PM framework details

LPDS entry-exit sequence



Folder structure and API listing

Folder structure

The folder structure of the middleware component is as below:

+middleware

+ccs --- CCS project files

+ewarm --- IAR project files

+driver --- driver implementation (synchronouns drivers are spi_drv and uart_drv)

+hal --- Hardware abstraction layer for the peripherals

+framework

+pm --- Power management framework

+timer --- Timer framework

+soc --- Reference implementation of the PM framework user hook functions

Core PM framework APIs

API	Description
cc_pm_framework_init	Initialize the power management framework
cc_pm_platform_ops_register	Register the user hook functions with the PM framework
cc_modules_load	Load the registered modules
cc_module_drv_register	Register a new module driver with the PM framework
cc_module_drv_unregister	UnRegister a module driver from the PM framework
cc_module_clk_enable	TBD
cc_module_clk_disable	TBD
cc_app_putoff_pm	Put off the PM framework
cc_app_resume_pm	Resume the PM fra

Platform Service APIs

API	Description
cc_set_up_wkup_srcs	Sets up wake-up sources for indicated power mode
cc_handle_S3_wakeup	Process events that have woken up system from S3 (LPDS) by triggering a software interrupt
cc_are_irqs_pending	Check if any interrupts are pending in the system
cc_enter_S4	Enter HIBernate
cc_enter_S3	Enter LPDS
cc_enter_S2	Enter DeepSleep
cc_enter_S1	Enter Sleep
cc_back_up_soc_data	Enables the SRAM retention, Saves the NVIC registers
cc_restore_soc_data	Restores the NVIC registers, Acquires the I2C and GPIO semaphore
wake_interrupt_handler	Handles the Power and Reset Control Module (PRCM) wake events

Reference Implementation

Reference implementations using the power management framework is available in the SDK package. The example apps are the following:

1. Idle profile:

The CC3200 device is connected to an AP and is continuously waiting for a packet over the air (UDP Rx) to wakeup. For all the times when the application is idling, it enters LPDS. In order to showcase all wake sources, GPIO13 and timer based wakeup from LPDS is also supported. Please refer to the link for more details: CC32xx Idle Profile Application

2. Sensor profile:

The CC3200 device periodically wakes up from HIBernate, broadcasts a message and then enters HIBernate. The application also allows for wakeup from HIBernate based on GPIO13. Please refer to the link for more details: CC32xx Sensor Profile Application

Limitations and known issues

1. This is a feature complete version of the PM framework. System tests have been performed successfully. The production version of the framework would be released after system test cycles on ES 1.33 device.
2. ARM CM-4 Clock gating (WFI/WFE instructions) cannot be invoked if Low power deep sleep is invoked in the user application. Above restriction would be relaxed with production devices.
3. Usage of Deepsleep while using peripherals other than GPIO is not recommended due to the variation in clock frequency during the transition to the power state.
4. The drivers have not been tested for all possible input configurations.
5. Power Management framework would not work with ES 1.21 devices.

Article Sources and Contributors

CC3200 Power Management Framework *Source*: http://processors.wiki.ti.com/index.php?oldid=227434 *Contributors*: A0221015, Codycooke, Jitgupta, Made4engineering, SarahP

Image Sources, Licenses and Contributors

File:Cc31xx_cc32xx_return_home.png *Source*: http://processors.wiki.ti.com/index.php?title=File:Cc31xx_cc32xx_return_home.png *License*: unknown *Contributors*: A0221015
File:CC3200 PMFW Overview.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:CC3200_PMFW_Overview.jpg *License*: unknown *Contributors*: Jitgupta
Image:Low_power_modes.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Low_power_modes.jpg *License*: unknown *Contributors*: Jitgupta
Image:CC3200 PMF SoftwareInterface.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:CC3200_PMF_SoftwareInterface.jpg *License*: unknown *Contributors*: Codycooke
Image:CC3200 timer fmwk.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:CC3200_timer_fmwk.jpg *License*: unknown *Contributors*: Codycooke
Image:CC3200 low power wakesources.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:CC3200_low_power_wakesources.jpg *License*: unknown *Contributors*: Codycooke
Image:CC3200 LPDS entryexit.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:CC3200_LPDS_entryexit.jpg *License*: unknown *Contributors*: Codycooke