CC26x0, CC13x0 SimpleLink™ Wireless MCU Power Management Software Development

Reference Guide

Literature Number: SWRA486A
August 2015–Revised April 2017

Contents

List of Figures

List of Tables

CC26x0, CC13x0 SimpleLink™ Wireless MCU Power Management Software Development

The CC26x0 and CC13x0 family of devices are part of the SimpleLink™ microcontroller (MCU) platform, which consists of Wi-Fi®, Bluetooth® low energy, Sub-1 GHz, and host MCUs. All share a common, easy-to-use development environment with a single core software development kit (SDK) and rich tool set. A one-time integration of the SimpleLink platform lets you add any combination of devices from the portfolio into your design. The ultimate goal of the SimpleLink platform is to achieve 100 percent code reuse when your design requirements change. For more information, visit http://www.ti.com/simplelink.

Power management for the CC26x0 and CC13x0 SimpleLink ultra-low-power wireless MCU platforms is implemented as part of the TI-RTOS kernel. This document gives an overview of this implementation and what to consider when porting this implementation to another software framework.

This document addresses the parts specific to the CC26x0 and CC13x0 family of devices, but does not cover the generic TI-RTOS kernel implementation. This guide also describes tasks that the software must perform to safely operate this device. Tasks are in bold followed by their descriptions.

Trademarks

SimpleLink is a trademark of Texas Instruments.
Cortex is a registered trademark of ARM.
Bluetooth is a registered trademark of Bluetooth SIG.
Wi-Fi is a registered trademark of Wi-Fi Alliance.

Reference Guide

SWRA486A – August 2015 – Revised April 2017

CC26x0, CC13x0 SimpleLink™ Wireless MCU Power Management Software Development

1 Abbreviations

ADC	Analog-to-digital converter
AON	Always-on domain
CCFG	Customer configuration flash area
OSC	Oscillator
OSC_DIG	Digital part of oscillator module
PRCM	Power, reset, and clock management
RCOSC_HF	High-frequency RC oscillator
RCOSC_LF	Low-frequency RC oscillator
RTC	Real-time clock
RTOS	Real-time operating system
TDC	Time-to-digital converter
TI-RTOS	TI real-time operating system
TRM	Technical reference manual
WFI	Wait-for-interrupt
XOSC_HF	High-frequency crystal oscillator
XOSC_LF	Low-frequency crystal oscillator

2 Power Management Introduction

In the TI-RTOS implementation for CC26x0 and CC13x0 devices, power management implements within the RTOS kernel. Because of this implementation, you can write applications without handling power management. The application requires no changes when upgrading to new versions of the kernel with further optimized power management. Handling power management within TI-RTOS occurs by configuring the system to the lowest power mode while supporting the requested resources from the application. The TI-RTOS installation is available at the TI website. This installation includes the source code for handling power management.

TI-RTOS supports the following four power modes:

- Active
- Idle
- Standby
- Shutdown

Section 3 presents the modes in detail. In active and idle mode, all resources on the device are available. The standby and shutdown modes are optimized for power consumption and limited resources are available in these modes. In TI-RTOS, the application does not select which power mode to enter. The kernel selects the power mode based on the next wakeup and which resources the application requests (such as the radio or another peripherals). The application must call only shutdown mode because waking up from shutdown forces a reboot of the system.

The TI-RTOS implementation is explained in the *TI-RTOS Power Management User's Guide*, C:\ti\tirtos_cc13xx_cc26xx_#_##_##_##\docs is documented within the TI-RTOS installation. This document references the TI-RTOS implementation document, but focuses on how to configure the hardware when not using TI-RTOS. The sample code in this document is from the TI-RTOS implementation and the driverlib APIs.

3 TI-RTOS Power Modes

Table 1 lists the four different power modes TI-RTOS supports.

Table 1. Software Configurable Power Modes

Mode	Software Configurable Power Modes				Reset Pin Held
	Active	Idle	Standby	Shutdown	
System CPU	Active	Off	Off	Off	Off
System SRAM	On	On	Retained	Off	Off
Register Retention[1]	Full	Full	Partial	No	No
VIMS_PD (flash)	On	Available	Off	Off	Off
RFCORE_PD (radio)	Available	Available	Off	Off	Off
SERIAL_PD	Available	Available	Off	Off	Off
PERIPH_PD	Available	Available	Off	Off	Off
Sensor Controller	Available	Available	Available	Off	Off
Supply System	On	On	Duty-cycled	Off	Off
Current	Application dependent	Application Dependent	Approx. 1 µA	Approx. 0.1 µA	Approx. 0.1 µA
Time from CPU active to ready for Wakeup[2]	–	TBD	TBD	TBD	–
Wake-up time to CPU active[2]	–	25 µs	300 µs[1]	Approx. 1.5 ms	Approx. 1.5 ms
High-speed clock	XOSC_HF or RCOSC_HF	XOSC_HF or RCOSC_HF	Off	Off	Off
Low-speed clock	XOSC_LF or RCOSC_LF	XOSC_LF or RCOSC_LF	XOSC_LF or RCOSC_LF	Off	Off
Wakeup on RTC	Available	Available	Available	Off	Off
Wakeup on pin edge	Available	Available	Available	Available	Off
Wakeup on reset pin	Available	Available	Available	Available	Available

[1] When an emulator or debugger is attached to the device, the wake-up time is approximately 200 µs shorter because the system will not enter true standby.

[2] Numbers include TI-RTOS overhead.

This section describes how these power modes are implemented in TI-RTOS and how to port TI-RTOS to another software framework.

3.1 Active Mode

In active mode, the Cortex®-M3 either executes code or waits through a WFI instruction for interrupts. System resources are available in active mode. The CPU power domain, peripherals, and radio may be active. Because the resources on CC26x0 and CC13x0 devices are available to the application in this mode, the current consumption depends on which modules are used.

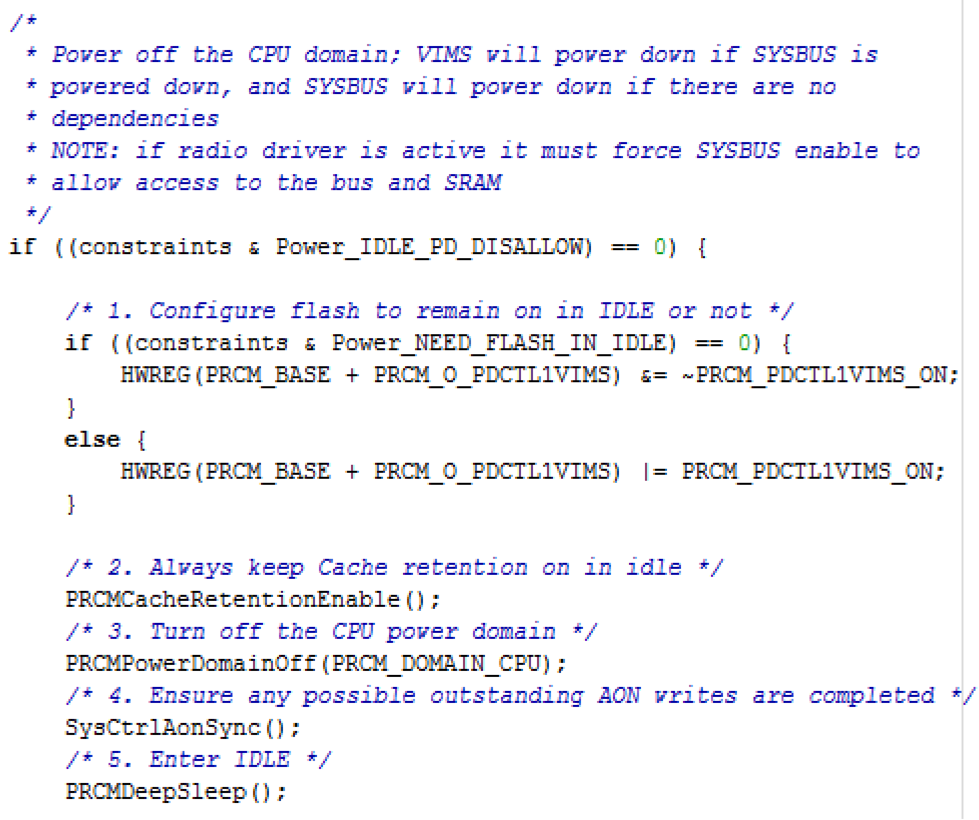
If the Cortex-M3 executed a WFI instruction, it can wake up from any enabled interrupt source. The difference between active and idle mode (see Section 3.2) is that in active mode the CPU power domain remains on, ensuring a quick wake up. The wake-up time from idle mode is longer because the CPU power domain must be turned on again.

3.2 Idle Mode

In idle mode, the Cortex-M3 is in deep-sleep after executing the WFI instruction. The Cortex-M3 waits for interrupts to wake up. The CPU power domain is off to reduce the current consumption. The CPU power domain increases the wake-up time of approximately 25 µs. Consider this increase if the application is timing sensitive. Although the CPU power domain is off, it retains its contents. When waking up from idle mode, the Cortex-M3 resumes executing code at the point it left off before entering idle. The state and register contents of the Cortex-M3 remain unchanged.

In idle mode, the application program can use system resources, including the high-speed crystal oscillator, the radio, and the peripherals. The Cortex-M3 can wake up from any enabled interrupt source. For details on entering idle mode, see Figure 1.

The code for entering idle mode is from the TI-RTOS implementation. Find it at the following path: *C:\ti\tirtos_cc13xx_cc26xx_#_##_##_##\products\tidrivers_cc13xx_cc26xx_#_##_##_##\packages\ti\drivers\power\PowerCC26XX_tirtos.c*.



```
/*
 * 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
 */
if ((constraints & Power_IDLE_PD_DISALLOW) == 0) {

    /* 1. Configure flash to remain on in IDLE or not */
    if ((constraints & Power_NEED_FLASH_IN_IDLE) == 0) {
        HWREG(PRCM_BASE + PRCM_O_PDCTL1VIMS) &= ~PRCM_PDCTL1VIMS_ON;
    }
    else {
        HWREG(PRCM_BASE + PRCM_O_PDCTL1VIMS) |= PRCM_PDCTL1VIMS_ON;
    }

    /* 2. Always keep Cache retention on in idle */
    PRCMCacheRetentionEnable();
    /* 3. Turn off the CPU power domain */
    PRCMPowerDomainOff(PRCM_DOMAIN_CPU);
    /* 4. Ensure any possible outstanding AON writes are completed */
    SysCtrlAonSync();
    /* 5. Enter IDLE */
    PRCMDeepSleep();
```

Figure 1. Idle Enter Sequence

1. **Configure flash to remain on in IDLE or not–** The flash can remain on to support DMA transactions from flash or if the RF core must access the flash. If the application requires the flash to remain available while in idle, set the PRCM:PDCTL1 bit (see *CC13x0, CC26x0 Technical Reference Manual*) before entering idle. If the application does not require flash access while in idle, clear this bit for the lowest power consumption.
2. **Always keep Cache retention on in idle–** Always enable the cache retention to ensure faster start-up time and easier software implementation. The current consumption caused by retaining the cache in idle is negligible compared to the total idle current consumption.
3. **Turn off the CPU power domain–** Ensure the CPU power domain is off when the Cortex-M3 enters deep-sleep. The CPU power domain is turned off after the Cortex-M3 executes the WFI instruction (with the NVIC SLEEP_DEEP bit set) and enters deep-sleep.

4. **Ensure any possible outstanding AON writes are completed–** If the AON domain has any outstanding writes, the driver must ensure they complete before entering idle. This function call stalls the Cortex-M3 execution for up to one SCLK_LF period because the AON domain runs on the SCLK_LF.
5. **Enter IDLE—** Put the Cortex-M3 in deep-sleep through the WFI instruction to enter idle. If the wake-up source is the RTC, the software must perform AON synchronization (in Step 4) before reading from the RTC module to ensure the RTC values are updated correctly.

3.3 Standby Mode

In standby mode, the Cortex-M3 waits in deep-sleep for interrupts to wake up after executing the WFI instruction. Before entering standby, power down system resources, including the high-frequency oscillator, radio, peripherals, and auxiliary domain. Many system modules have retention in standby including the Cortex-M3 (see the *CC13x0, CC26x0 Technical Reference Manual* for details). After waking up from standby, the Cortex-M3 resumes executing programs at the point before entering standby.

Because CC26x0 and CC13x0 devices run on a low-power LDO in standby, most system resources are unavailable in standby. Use the RTC, which runs in standby, to wake up the Cortex-M3. The Cortex-M3 can wake up on any interrupt-enabled, external input pin.

The application must consider the latency associated with entering and exiting when using standby mode.

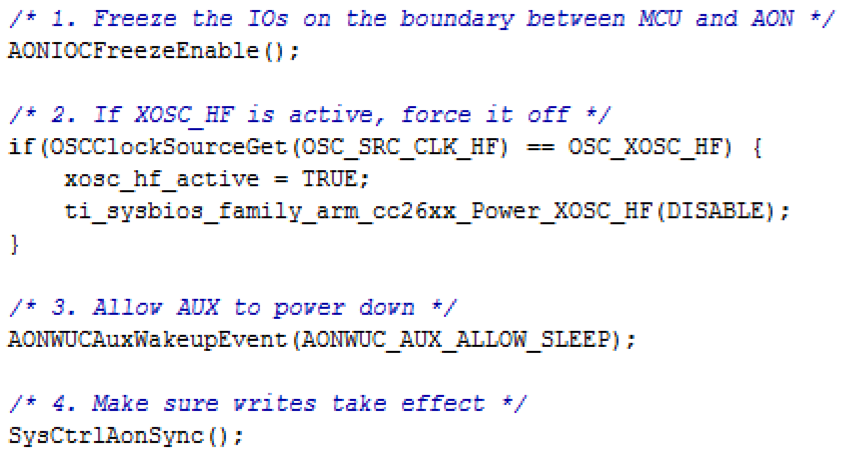
If the next wakeup is in less than 1 ms, the TI-RTOS implementation does not enter standby. If the application requires exact timing, consider the latency associated with entering and waking up from standby. The TI-RTOS implementation handles this latency by inserting wake-up event four RTC edges (64 µs) earlier than the event to ensure the system powers on in time to service the wake-up event scheduled by the application.

The TI-RTOS standby implementation allows a variation where the cache contents are retained. In standby, the system draws approximately 2 µA more than it does without retaining the cache. System start-up from standby is typically quicker. The main system RAM should always be retained in standby mode. The main system RAM retention is also the default configuration when the device boots up.

The following code for entering and exiting standby is from the TI-RTOS PowerCC26XX.c implementation: *C:\ti\tirtos_cc13xx_cc26xx_#_##_##_##\products\tidrivers_cc13xx_cc26xx_#_##_##_##\packages\ti\drivers\power\PowerCC26XX.c*.

3.3.1 Standby Enter Sequence

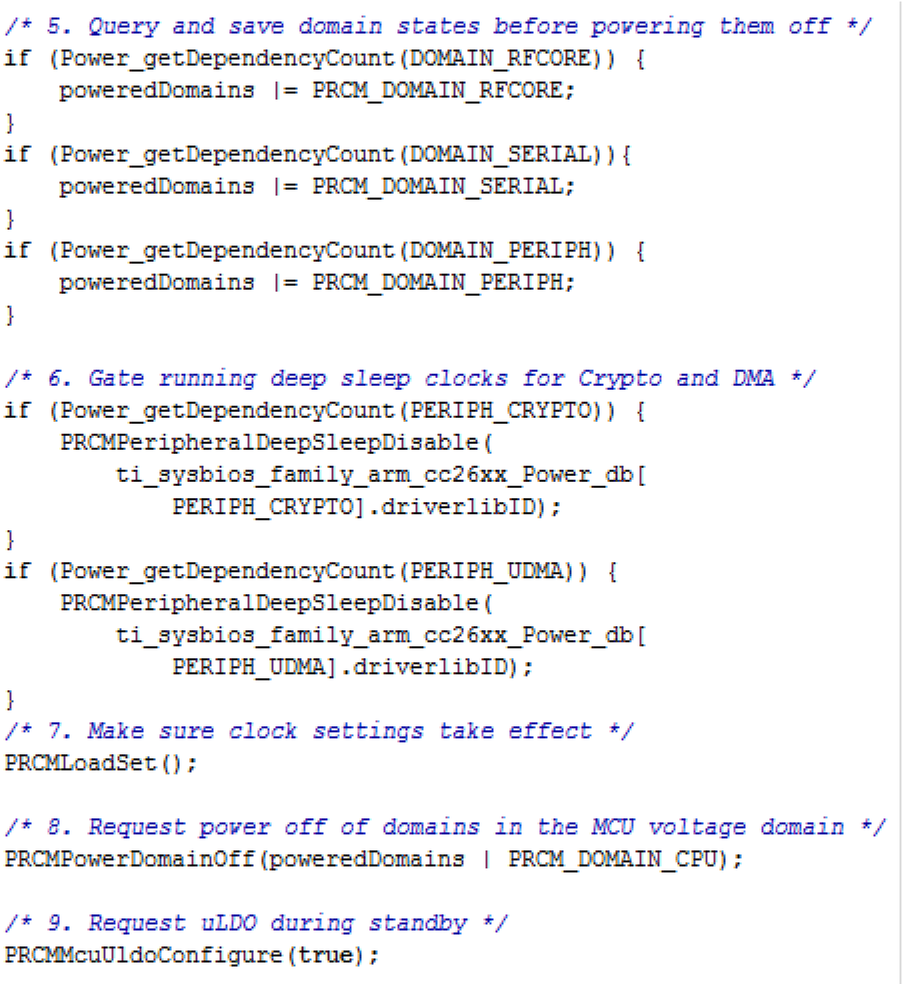
Figure 2, Figure 3, and Figure 4 show the code sequence for taking the device into standby mode. A brief explanation of each step is also provided.



```
/* 1. Freeze the IOs on the boundary between MCU and AON */
AONIOCFreezeEnable();

/* 2. If XOSC_HF is active, force it off */
if(OSCClockSourceGet(OSC_SRC_CLK_HF) == OSC_XOSC_HF) {
    xosc_hf_active = TRUE;
    ti_sysbios_family_arm_cc26xx_Power_XOSC_HF(DISABLE);
}

/* 3. Allow AUX to power down */
AONWUCAuxWakeupEvent(AONWUC_AUX_ALLOW_SLEEP);

/* 4. Make sure writes take effect */
SysCtrlAonSync();
```

Figure 2. Standby Enter Sequence, Part 1

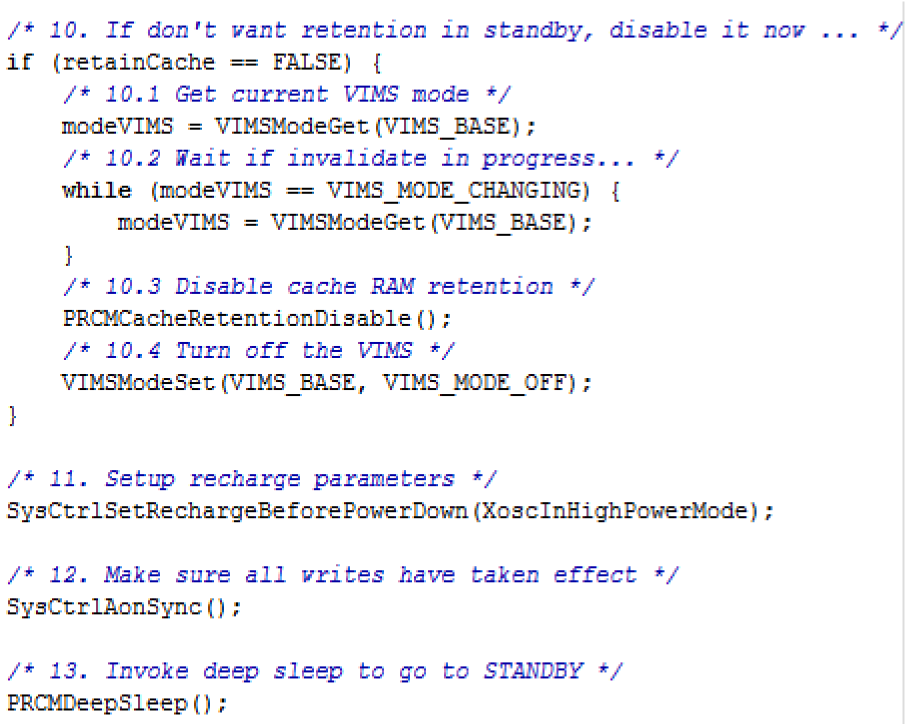
1. **Freeze the IOs on the boundary between MCU and AON–** Latch (freeze) the input and output pins to avoid glitches when in standby. When the inputs and outputs are frozen, the current state of the inputs and outputs is retained while available to trigger wakeup to the Cortex-M3. To configure the inputs for wakeup from standby, see Section 4.4.1.
2. **If XOSC_HF is active, force it off–** Turn off the high-speed crystal oscillator before entering standby. The hardware turns off the high-speed RC oscillator when entering standby. For details on how to configure the high-speed crystal oscillator, see Section 4.5.1.
3. **Allow AUX to power down–** Disconnect the auxiliary domain from the MCU in standby.
4. **Make sure writes take effect–** Ensure the writes to AON domain complete before continuing.



```
/* 5. Query and save domain states before powering them off */
if (Power_getDependencyCount(DOMAIN_RFCORE)) {
    poweredDomains |= PRCM_DOMAIN_RFCORE;
}
if (Power_getDependencyCount(DOMAIN_SERIAL)){
    poweredDomains |= PRCM_DOMAIN_SERIAL;
}
if (Power_getDependencyCount(DOMAIN_PERIPH)) {
    poweredDomains |= PRCM_DOMAIN_PERIPH;
}

/* 6. Gate running deep sleep clocks for Crypto and DMA */
if (Power_getDependencyCount(PERIPH_CRYPTO)) {
    PRCMPeripheralDeepSleepDisable(
        ti_sysbios_family_arm_cc26xx_Power_db[
            PERIPH_CRYPTO].driverlibID);
}
if (Power_getDependencyCount(PERIPH_UDMA)) {
    PRCMPeripheralDeepSleepDisable(
        ti_sysbios_family_arm_cc26xx_Power_db[
            PERIPH_UDMA].driverlibID);
}
/* 7. Make sure clock settings take effect */
PRCMLoadSet();

/* 8. Request power off of domains in the MCU voltage domain */
PRCMPowerDomainOff(poweredDomains | PRCM_DOMAIN_CPU);

/* 9. Request uLDO during standby */
PRCMMcuUldoConfigure(true);
```

Figure 3. Standby Enter Sequence, Part 2

5. **Query and save domain states before powering them off–** Turn off all power domains before entering standby.
6. **Gate running deep sleep clocks for Crypto and DMA–** Clear both the crypto and DMA deep-sleep clock enable bits before entering standby. For more information, see the PRCM:VDCTL register.
7. **Make sure clock settings take effect–** Load the clock settings from the PRCM module.
8. **Request power off of domains in the MCU voltage domain–**Turn off power domains inside the MCU voltage domain.
9. **Request uLDO during standby–** Use this call to request to use the micro-LDO during standby. The system runs off the micro-LDO in standby.



```
/* 10. If don't want retention in standby, disable it now ... */
if (retainCache == FALSE) {
    /* 10.1 Get current VIMS mode */
    modeVIMS = VIMSModeGet(VIMS_BASE);
    /* 10.2 Wait if invalidate in progress... */
    while (modeVIMS == VIMS_MODE_CHANGING) {
        modeVIMS = VIMSModeGet(VIMS_BASE);
    }
    /* 10.3 Disable cache RAM retention */
    PRCMCacheRetentionDisable();
    /* 10.4 Turn off the VIMS */
    VIMSModeSet(VIMS_BASE, VIMS_MODE_OFF);
}

/* 11. Setup recharge parameters */
SysCtrlSetRechargeBeforePowerDown(XoscInHighPowerMode);

/* 12. Make sure all writes have taken effect */
SysCtrlAonSync();

/* 13. Invoke deep sleep to go to STANDBY */
PRCMDeepSleep();
```

Figure 4. Standby Enter Sequence, Part 3

10. **If don't want retention in standby, disable it now...–** Disable cache retention and turn off the VIMS power domain to achieve lowest power consumption in standby.

    10.1 **Get current VIMS mode–** Read the current VIMS mode.

    10.2 **Wait if invalidate in progress...–** Ensure that the VIMS mode remains the same.

    10.3 **Disable cache RAM retention–** Disable cache RAM retention.

    10.4 **Turn off the VIMS–** Set the VIMS mode to off. The cache is off and the Cortex-M3 reads instructions from flash.

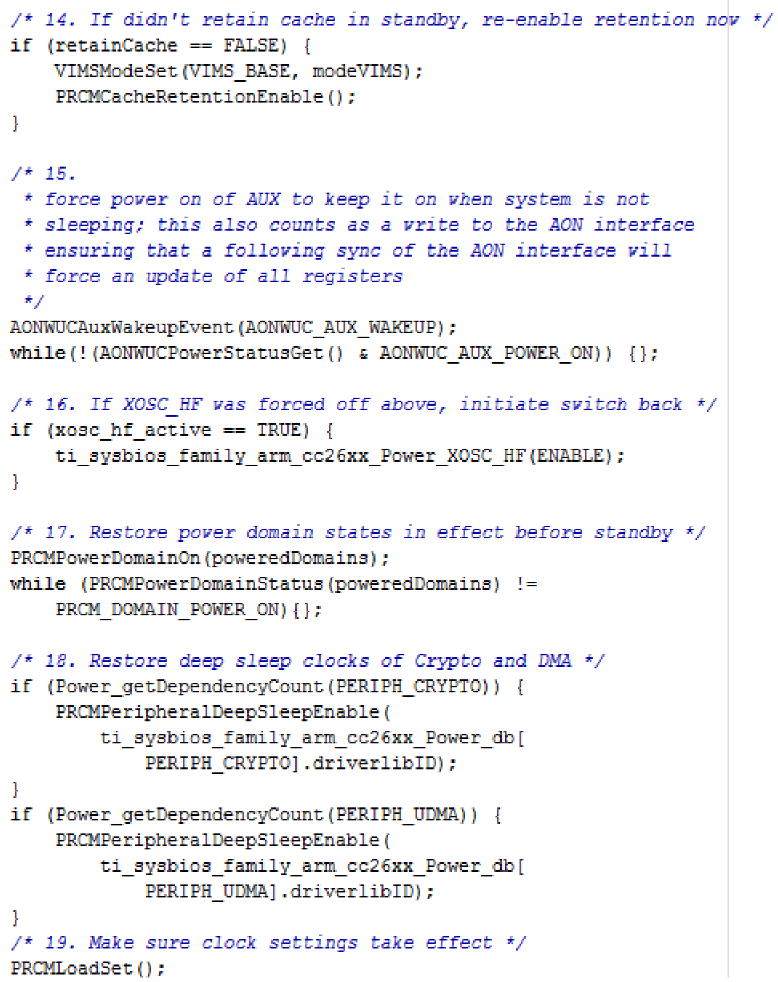
11. **Setup recharge parameters–** Configure safe recharge settings used when in standby. A XoscInHighPowerMode flag indicates that the high-frequency crystal oscillator is off in standby.

12. **Make sure all writes have taken effect—** Ensure all AON writes have taken effect.

13. **Invoke deep sleep to go to STANDBY—** Put the Cortex-M3 in deep-sleep through the WFI instruction to enter standby.

3.3.2 Standby Exit Sequence

Figure 5 and Figure 6 show the code sequence for taking the device out of standby mode. A brief explanation of each step is also provided.



```
/* 14. If didn't retain cache in standby, re-enable retention now */
if (retainCache == FALSE) {
    VIMSModeSet(VIMS_BASE, modeVIMS);
    PRCMCacheRetentionEnable();
}

/* 15.
 * force power on of AUX to keep it on when system is not
 * sleeping; this also counts as a write to the AON interface
 * ensuring that a following sync of the AON interface will
 * force an update of all registers
 */
AONWUCAuxWakeupEvent(AONWUC_AUX_WAKEUP);
while(!(AONWUCPowerStatusGet() & AONWUC_AUX_POWER_ON)) {};

/* 16. If XOSC_HF was forced off above, initiate switch back */
if (xosc_hf_active == TRUE) {
    ti_sysbios_family_arm_cc26xx_Power_XOSC_HF(ENABLE);
}

/* 17. Restore power domain states in effect before standby */
PRCMPowerDomainOn(poweredDomains);
while (PRCMPowerDomainStatus(poweredDomains) !=
    PRCM_DOMAIN_POWER_ON){};

/* 18. Restore deep sleep clocks of Crypto and DMA */
if (Power_getDependencyCount(PERIPH_CRYPTO)) {
    PRCMPeripheralDeepSleepEnable(
        ti_sysbios_family_arm_cc26xx_Power_db[
            PERIPH_CRYPTO].driverlibID);
}
if (Power_getDependencyCount(PERIPH_UDMA)) {
    PRCMPeripheralDeepSleepEnable(
        ti_sysbios_family_arm_cc26xx_Power_db[
            PERIPH_UDMA].driverlibID);
}
/* 19. Make sure clock settings take effect */
PRCMLoadSet();
```

Figure 5. Standby Exit Sequence, Part 1

14. **If didn't retain cache in standby, re-enable retention now**— Enable cache after waking up from standby if cache retention was disabled.

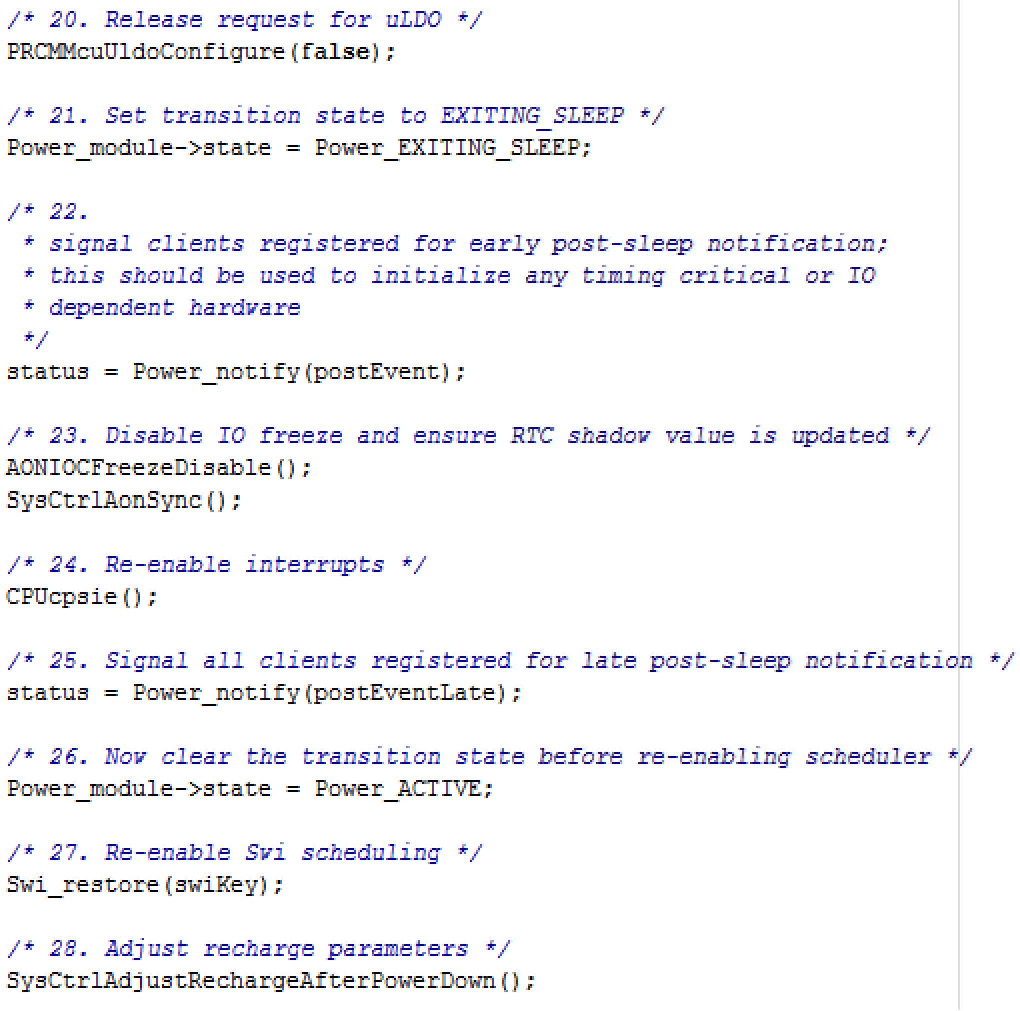
15. **Force power on of AUX to keep it on when system is not sleeping; this also counts as a write to the AON interface ensuring that a following sync of the AON interface will force an update of all registers—** To access the OSC_DIG registers to configure the oscillators, force the auxiliary domain on after waking up from standby.

16. **If XOSC_HF was forced off above, initiate switch back—** Turn on the high-speed crystal oscillator if it was enabled before entering standby. For details on how to configure the high-speed crystal oscillator, see Section 4.5.1.

17. **Restore power domain states in effect before standby—** Turn on all power domains that were on before entering standby.

18. **Restore deep sleep clocks of Crypto and DMA—** Restore deep-sleep clock settings for Crypto and DMA modules as they were before entering standby.

19. **Make sure clock settings take effect—** Load clock settings into PRCM for them to take effect.



```
/* 20. Release request for uLDO */
PRCMMcuUldoConfigure(false);

/* 21. Set transition state to EXITING_SLEEP */
Power_module->state = Power_EXITING_SLEEP;

/* 22.
 * signal clients registered for early post-sleep notification;
 * this should be used to initialize any timing critical or IO
 * dependent hardware
 */
status = Power_notify(postEvent);

/* 23. Disable IO freeze and ensure RTC shadow value is updated */
AONIOCFreezeDisable();
SysCtrlAonSync();

/* 24. Re-enable interrupts */
CPUcpsie();

/* 25. Signal all clients registered for late post-sleep notification */
status = Power_notify(postEventLate);

/* 26. Now clear the transition state before re-enabling scheduler */
Power_module->state = Power_ACTIVE;

/* 27. Re-enable Swi scheduling */
Swi_restore(swiKey);

/* 28. Adjust recharge parameters */
SysCtrlAdjustRechargeAfterPowerDown();
```

Figure 6. Standby Exit Sequence, Part 2

20. **Release request for uLDO—** Release the micro-LDO request to disable running on the micro-LDO when entering idle.

21. **Set transition state to EXITING_SLEEP—** TI-RTOS-specific code

22. **Signal clients registered for early post-sleep notification; this should be used to initialize any timing critical or IO dependent hardware—** Signal peripheral drivers to reinitialize their state because modules without retention lose their contents. TI-RTOS-specific code. For details on which modules have retention, see (*CC13x0, CC26x0 Technical Reference Manual*).

23. **Disable IO freeze and ensure RTC shadow value is updated—** Open the input and output latches. Because the peripheral drivers have been reinitialized, inputs and outputs have no glitches.

24. **Re-enable interrupts—** Re-enable interrupts. TI-RTOS-specific code

25. **Signal all clients registered for late post-sleep notification—** The peripheral drivers do not use this notification. (TI-RTOS-specific code)

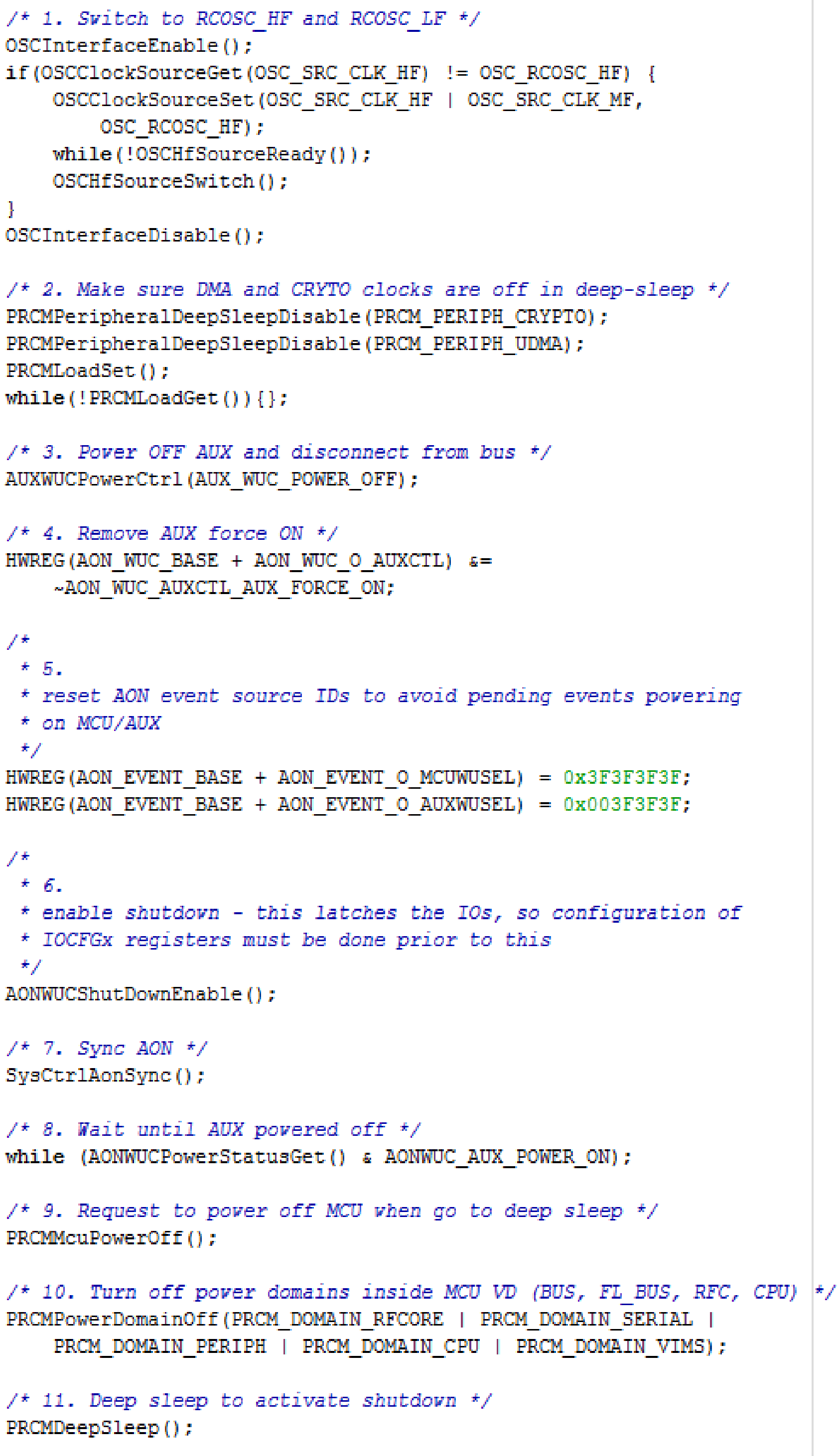
26. **Now clear the transition state before re-enabling scheduler—** TI-RTOS-specific code

27. **Re-enable Swi scheduling—** TI-RTOS-specific code

28. **Adjust recharge parameters—** Adjust the recharge settings to optimize power consumption the next time the application enters standby mode.

3.4 Shutdown Mode

In shutdown mode, the CC26x0 and CC13x0 devices shut off except for the input and output pins. The device can wake up on any input pin (configured through the IOCFGn registers). The system completely reboots when waking up from shutdown. You can call SysCtrlResetSourceGet() to detect that the device is booting from shutdown. For details on the TI-RTOS implementation of entering shutdown, see Figure 7.



```
/* 1. Switch to RCOSC_HF and RCOSC_LF */
OSCInterfaceEnable();
if(OSCClockSourceGet(OSC_SRC_CLK_HF) != OSC_RCOSC_HF) {
    OSCClockSourceSet(OSC_SRC_CLK_HF | OSC_SRC_CLK_MF,
        OSC_RCOSC_HF);
    while(!OSCHfSourceReady());
    OSCHfSourceSwitch();
}
OSCInterfaceDisable();

/* 2. Make sure DMA and CRYTO clocks are off in deep-sleep */
PRCMPeripheralDeepSleepDisable(PRCM_PERIPH_CRYPTO);
PRCMPeripheralDeepSleepDisable(PRCM_PERIPH_UDMA);
PRCMLoadSet();
while(!PRCMLoadGet()){};

/* 3. Power OFF AUX and disconnect from bus */
AUXWUCPowerCtrl(AUX_WUC_POWER_OFF);

/* 4. Remove AUX force ON */
HWREG(AON_WUC_BASE + AON_WUC_O_AUXCTL) &=
    ~AON_WUC_AUXCTL_AUX_FORCE_ON;

/*
 * 5.
 * reset AON event source IDs to avoid pending events powering
 * on MCU/AUX
 */
HWREG(AON_EVENT_BASE + AON_EVENT_O_MCUWUSEL) = 0x3F3F3F3F;
HWREG(AON_EVENT_BASE + AON_EVENT_O_AUXWUSEL) = 0x003F3F3F;

/*
 * 6.
 * enable shutdown - this latches the IOs, so configuration of
 * IOCFGx registers must be done prior to this
 */
AONWUCShutDownEnable();

/* 7. Sync AON */
SysCtrlAonSync();

/* 8. Wait until AUX powered off */
while (AONWUCPowerStatusGet() & AONWUC_AUX_POWER_ON);

/* 9. Request to power off MCU when go to deep sleep */
PRCMMcuPowerOff();

/* 10. Turn off power domains inside MCU VD (BUS, FL_BUS, RFC, CPU) */
PRCMPowerDomainOff(PRCM_DOMAIN_RFCORE | PRCM_DOMAIN_SERIAL |
    PRCM_DOMAIN_PERIPH | PRCM_DOMAIN_CPU | PRCM_DOMAIN_VIMS);

/* 11. Deep sleep to activate shutdown */
PRCMDeepSleep();
```

Figure 7. Shutdown Enter Sequence

1. **Switch to RCOSC_HF and RCOSC_LF—** high-speed RC oscillator.
2. **Make sure DMA and CRYTO clocks are off before entering shutdown**
3. **Power OFF AUX and disconnect from bus—** Configure the auxiliary domain.
4. **Remove AUX force ON—** Request to power off the auxiliary domain.
5. **Reset AON event source IDs to avoid pending events powering on MCU/AUX—** Ensure no events from AON prevent the device from entering shutdown.
6. **Enable shutdown – this latches the IOs, so configuration of IOCFGx registers must be done prior to this—** Configure shutdown mode. Because this function also latches the inputs and outputs, configure any input and output (such as configuring which inputs and outputs to wake up on) before executing this function.
7. **Sync AON—** Ensure AON configurations take effect.
8. **Wait until AUX powered off—** Ensure the auxiliary domain is off.
9. **Request to power off MCU when go to deep sleep—** Request to power off the MCU voltage domain when Cortex-M3 enters deep-sleep mode.
10. **Turn off power domains inside MCU VD (BUS, FL_BUS, RFC, CPU)—** Turn off power domains inside the MCU voltage domain.
11. **Deep sleep to activate shutdown—** Enter deep-sleep mode by executing WFI instruction. WFI instruction shuts down the system. Because the Cortex-M3 reboots when waking up, no instructions execute after the PRCMDeepSleep() function.

When waking up from a shutdown, the device reboots. The input and output state is retained through shutdown to avoid glitches. Software using the shutdown-power mode calls SysCtrlResetSourceGet() early in the application to check the wake-up source. If the device wakes up from a shutdown, the application restores the configuration of the input and output registers before opening to latches to avoid glitches. Write 1 to AON_SYSCTL:SLEEPCTL[0] to unlatch the inputs and outputs.

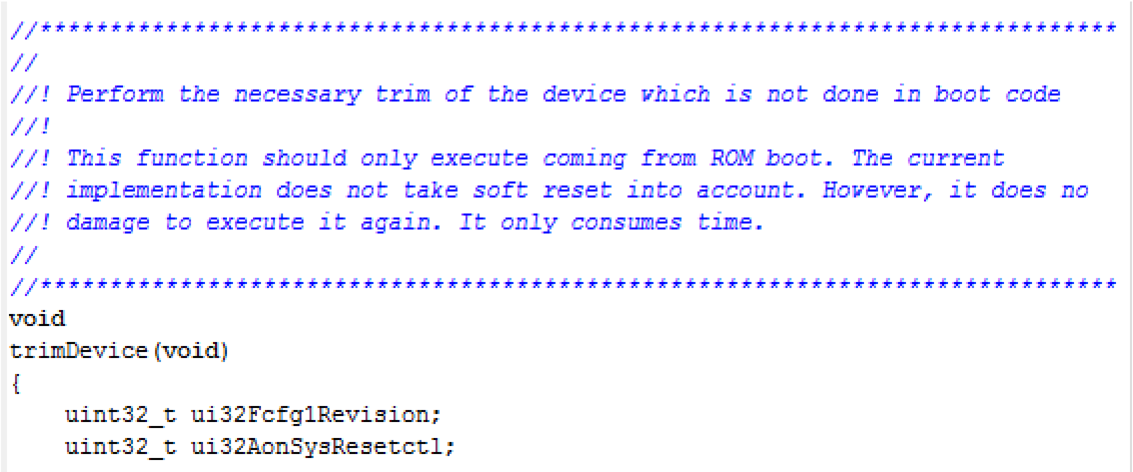
4 Implementation Considerations

This section describes important details to consider when writing software for the CC26x0 and CC13x0 families.

4.1 Device Initializing

4.1.1 Low-Level Initializing

The application must call trimDevice() in setup.c (available in the driverlib software package) after booting (see Figure 8). This function trims the different modules on the device not completed by ROM boot code. This trimming is required for the device to operate safely. This code runs again after coming out of shutdown because the Cortex-M3 reboots. The Cortex-M3 does not reboot when coming out of standby or idle.



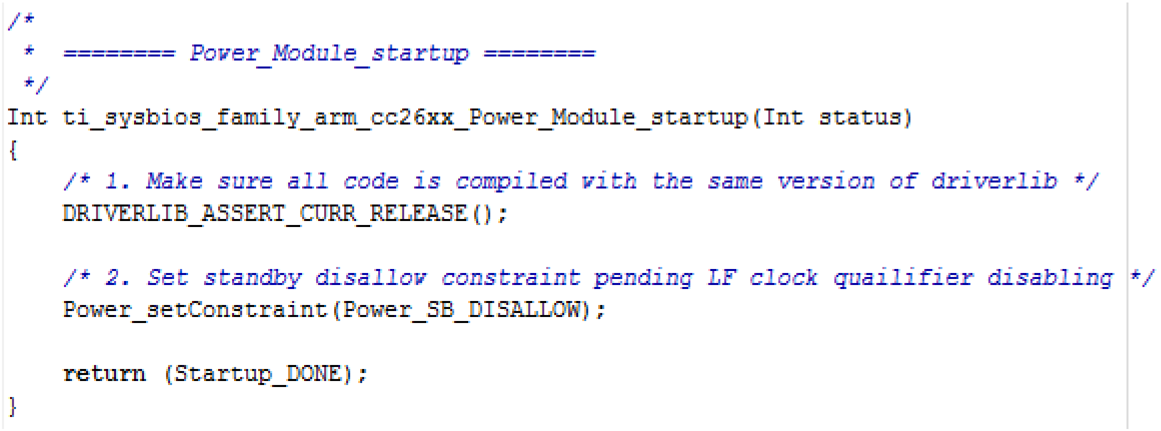
```
//*****************************************************************************
//
//! Perform the necessary trim of the device which is not done in boot code
//!
//! This function should only execute coming from ROM boot. The current
//! implementation does not take soft reset into account. However, it does no
//! damage to execute it again. It only consumes time.
//
//*****************************************************************************
void
trimDevice(void)
{
    uint32_t ui32Fcfg1Revision;
    uint32_t ui32AonSysResetctl;
```

Figure 8. trimDevice() Function in setup.c

NOTE: A check exists in the trimDevice() function to ensure the correct version of driverlib is on the device. If this check fails, the device spins in an infinite loop to indicate that the compiled software library is unsupported by the given device.

4.1.2 Initializating TI-RTOS

For more information on the device initialization by TI-RTOS, see Figure 9.



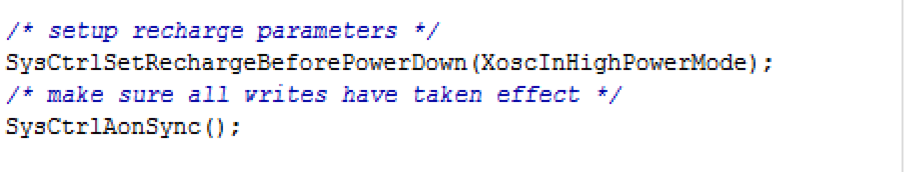
```
/*
 *  ======== Power_Module_startup ========
 */
Int ti_sysbios_family_arm_cc26xx_Power_Module_startup(Int status)
{
    /* 1. Make sure all code is compiled with the same version of driverlib */
    DRIVERLIB_ASSERT_CURR_RELEASE();

    /* 2. Set standby disallow constraint pending LF clock quailifier disabling */
    Power_setConstraint(Power_SB_DISALLOW);

    return (Startup_DONE);
}
```

Figure 9. TI-RTOS Initialization

1. **Make sure all code is compiled with the same version of driverlib—** Ensure that the TI-RTOS implementation is built with the same version of driverlib as the rest of the application.
2. **Set standby disallow constraint pending LF clock qualifier disabling—** TI-RTOS-specific code to ensure the application enters standby after the LF clock source selected in CCFG is selected and stable. Disable the LF clock qualifiers after selecting the LF clock source.

4.2 Recharging in Standby

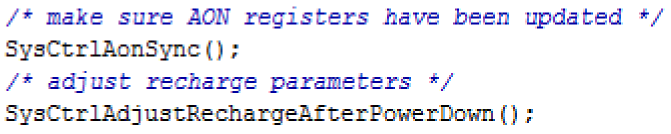
When the device is in the standby power mode, the hardware turns off the internal DC-DC converter and LDOs to save power. The device must wake up periodically to recharge the internal voltages to retain the chip state. Before entering standby, you must configure these recharge settings. For more details on these settings, see Figure 10. Because the recharge settings reside in the AON domain, the application must AON sync after writing the recharge settings to ensure they take effect before entering standby.



```
/* setup recharge parameters */
SysCtrlSetRechargeBeforePowerDown(XoscInHighPowerMode);
/* make sure all writes have taken effect */
SysCtrlAonSync();
```

Figure 10. Configure Recharge Settings Before Entering Standby

After the device wakes up from standby mode, set the recharge parameters through the call in Figure 11 to optimize the current consumption the next time the device enters standby. After waking up from standby, software must AON sync before adjusting the recharge settings to ensure the settings are correctly updated.

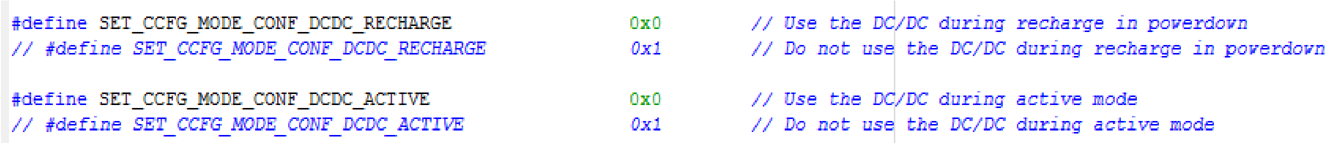


```
/* make sure AON registers have been updated */
SysCtrlAonSync();
/* adjust recharge parameters */
SysCtrlAdjustRechargeAfterPowerDown();
```

Figure 11. Configure Recharge Settings After Exiting Standby

4.3 Operating the DC-DC Converter

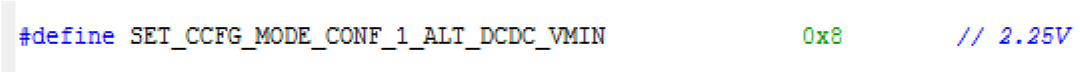
The CC26x0 and CC13x0 device families have an on-chip DC-DC converter which enables low-power applications. The DC-DC converter is enabled by trimDevice() based on the CCFG fields in Figure 12, and has a separate enable field for operating in active and standby modes (the hardware periodically recharges in standby mode). When the device operates in external regulator mode, disable the DC-DC converter during active and standby/recharge modes.



```
#define SET_CCFG_MODE_CONF_DCDC_RECHARGE             0x0        // Use the DC/DC during recharge in powerdown
// #define SET_CCFG_MODE_CONF_DCDC_RECHARGE          0x1        // Do not use the DC/DC during recharge in powerdown

#define SET_CCFG_MODE_CONF_DCDC_ACTIVE               0x0        // Use the DC/DC during active mode
// #define SET_CCFG_MODE_CONF_DCDC_ACTIVE            0x1        // Do not use the DC/DC during active mode
```

Figure 12. CCFG Fields For Enabling the DC-DC Converter During Active and Standby Mode

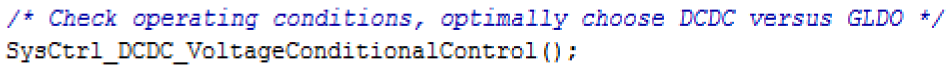
Because the DC-DC converter only operates with optimum efficiency at high VDDS voltages, turn off the DC-DC converter when VDDS decreases. To configure the voltage threshold to turn off the DC-DC converter, use the CCFG field in Figure 13. Do not modify this voltage threshold.



```
#define SET_CCFG_MODE_CONF_1_ALT_DCDC_VMIN            0x8        // 2.25V
```

Figure 13. The Voltage Threshold For Turning Off the DC-DC Converter

The application uses the function in Figure 14, the threshold configured in the CCFG field, to decide when to turn off the DC-DC converter. The application calls this function at periodic intervals. This function also turns on the DC-DC converter if the voltage exceeds the threshold (for example, when the battery is recharging).



```
/* Check operating conditions, optimally choose DCDC versus GLDO */
SysCtrl_DCDC_VoltageConditionalControl();
```

Figure 14. The Driverlib Function That Enables and Disables the DC-DC Converter at the Given CCFG Threshold

4.4 Configuring Device for External Input Interrupts and Wakeup

4.4.1 Interrupt and Wakeup from Active, Idle, and Standby

Configuring the device for external input interrupts and wakeups from active, idle, and standby power modes is a 3-step process (see PINCC26XX.c for reference).

1. Enable the INT_EDGE_DETECT interrupt to the Cortex-M3.
2. Enable wakeup from standby on an input by configuring one of the four available wakeup sources in AON_EVENT:MCUWUSEL to wake up on any PAD (0x20). (TI-RTOS uses AON_EVENT:MCUWUSEL.WU1_EV.)
3. Enable the individual DIO interrupts by configuring the IOCFGn[18:16] bits.

If multiple input and output pins are enabled for interrupt, software reads the GPIO:EVFLAGS31_0 to decode which input pin triggered the interrupt. Write 1 to the corresponding GPIO:EVFLAGS31_0 to clear the interrupt.

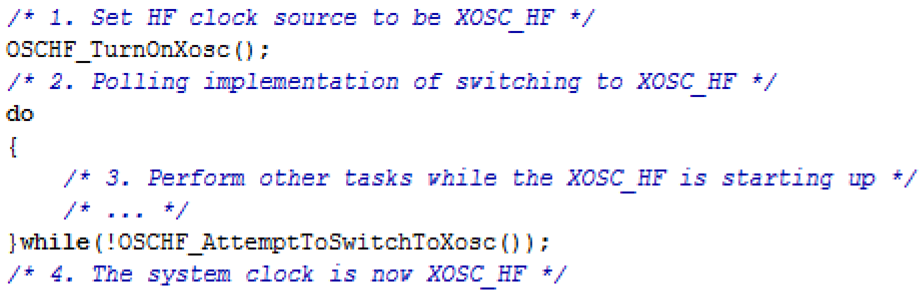
4.4.2 Wakeup from Shutdown Mode

Configure the IOCFGn[28:27] bits to configure the device to wake up from shutdown on an external input and output. The device reboots when waking up from shutdown.

4.5 Oscillators

4.5.1 High-Frequency Oscillators

The CC26x0 and CC13x0 devices boot with the 48-MHz high-speed RC oscillator that clocks the Cortex-M3. When using the radio, switch the main system clock to the high-speed crystal oscillator. The code in Figure 15 shows how to change to the high-speed crystal oscillator (see the functions in driverlib modules osc.h/osc.c).



```
/* 1. Set HF clock source to be XOSC_HF */
OSCHF_TurnOnXosc();
/* 2. Polling implementation of switching to XOSC_HF */
do
{
    /* 3. Perform other tasks while the XOSC_HF is starting up */
    /* ... */
}while(!OSCHF_AttemptToSwitchToXosc());
/* 4. The system clock is now XOSC_HF */
```

Figure 15. Switching from RCOSC_HF to XOSC_HF

NOTE: The high-speed crystal oscillator does not start immediately and the application can perform other tasks running on the RCOSC_HF while the XOSC_HF starts.

To optimize power consumption, another function gives the application an estimate of the length of the next XOSC_HF startup. The start-up time depends on the following three factors:

- How long the XOSC_HF has been off
- The operational parameters of the crystal
- The frequency of the RCOSC_HF

The OSCHF_GetStartupTime(uint32_t timeUntilWakeupInMs) returns an estimate of the expected XOSC_HF start-up time, depending on how long the XOSC_HF has been off. For example, the application can use this information to schedule wakeups and ensure that the XOSC_HF starts up before using the radio and transmitting a packet.

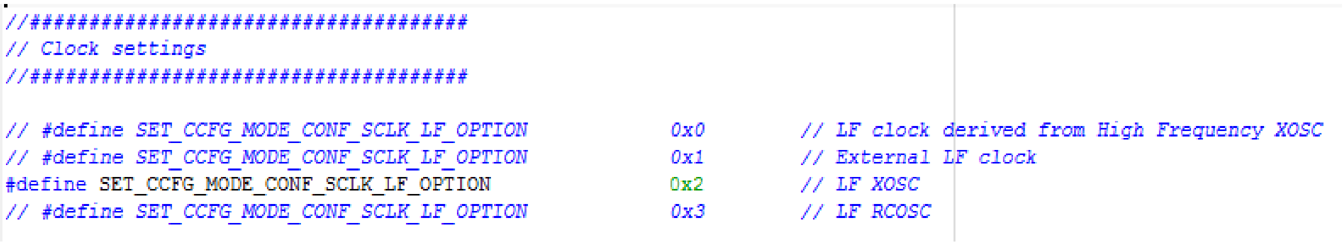
NOTE: You must enable the RTC for this functionality to work because the oscillator driver uses the RTC to calculate the start-up time.

You must turn off the XOSC_HF before entering standby or shutdown. To turn off the XOSC_HF, call OSCHF_SwitchToRcOscTurnOffXosc().

NOTE: You cannot switch to the RCOSC_HF before you have switched to the XOSC_HF. Aborting a switch to XOSC_HF is not supported.

4.5.2 Low-Frequency Oscillators

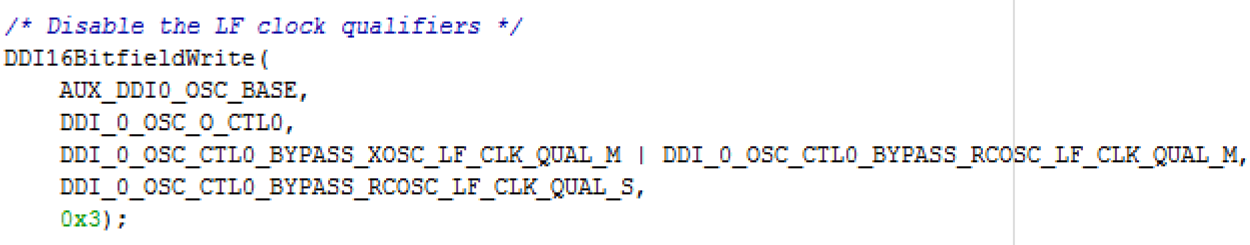
Modify the customer configuration area to select the low-frequency clock source. The application must ensure that the LF clock source selected in CCFG is selected and stable before entering standby mode. The LF clock source on which the device is running can be read through the driverlib function OSCClockSourceGet(…). For the configuration settings in CCFG, see Figure 16.



```
//#####################################
// Clock settings
//#####################################

// #define SET_CCFG_MODE_CONF_SCLK_LF_OPTION           0x0        // LF clock derived from High Frequency XOSC
// #define SET_CCFG_MODE_CONF_SCLK_LF_OPTION           0x1        // External LF clock
#define SET_CCFG_MODE_CONF_SCLK_LF_OPTION              0x2        // LF XOSC
// #define SET_CCFG_MODE_CONF_SCLK_LF_OPTION           0x3        // LF RCOSC
```

Figure 16. LF Configuration in ccfg.c

When the device running on the selected LF clock source as specified in CCFG, you must disable the LF clock qualifier circuitry before entering standby for the first time. Set bits OSC_DIG:CTL0[29:28] as in Figure 17.



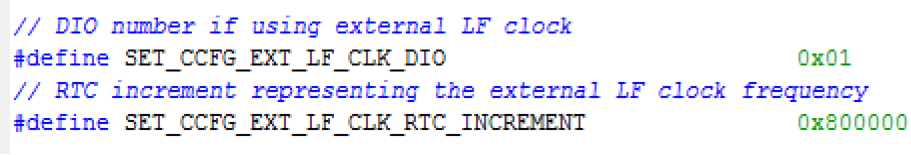
```
/* Disable the LF clock qualifiers */
DDI16BitfieldWrite(
    AUX_DDI0_OSC_BASE,
    DDI_0_OSC_O_CTL0,
    DDI_0_OSC_CTL0_BYPASS_XOSC_LF_CLK_QUAL_M | DDI_0_OSC_CTL0_BYPASS_RCOSC_LF_CLK_QUAL_M,
    DDI_0_OSC_CTL0_BYPASS_RCOSC_LF_CLK_QUAL_S,
    0x3);
```

Figure 17. Code for Disabling LF Clock Qualifiers

If the application wants to use an external LF clock source, another MCU can apply the LF clock to any input and output pin.

1. Configure the chosen input and output and the RTC increment value as in Figure 18.
2. Apply when trimDevice() runs.

An RTC increment value of 0x800000 corresponds to a frequency of 32.768 kHz. See AON_RTC:SUBSECINC for further details.



```
// DIO number if using external LF clock
#define SET_CCFG_EXT_LF_CLK_DIO                        0x01
// RTC increment representing the external LF clock frequency
#define SET_CCFG_EXT_LF_CLK_RTC_INCREMENT              0x800000
```

Figure 18. Configuration When Using an External LF Clock Source

4.5.3 RC Oscillator Calibration

Both the RCOSC_HF and RCOSC_LF frequency shift when the temperature changes and the oscillators age. Calibrate these oscillators periodically against the XOSC_HF.

Calibrating the RCOSC_HF gives more accurate timing when running the system off this clock and starts up the XOSC_HF faster. Calibrating the RCOSC_LF gives more accurate timing for the RTC when in standby. If the RCOSC_LF is unused by the application, calibrating this oscillator is unnecessary.

The auxiliary domain contains a TDC you can use to calibrate these oscillators against the XOSC_HF. TI-RTOS contains an implementation of calibrating both the RCOSC_HF and RCOSC_LF and can be a reference for those who must calibrate RC oscillators without TI-RTOS. The following implementation is in the TI-RTOS installation:

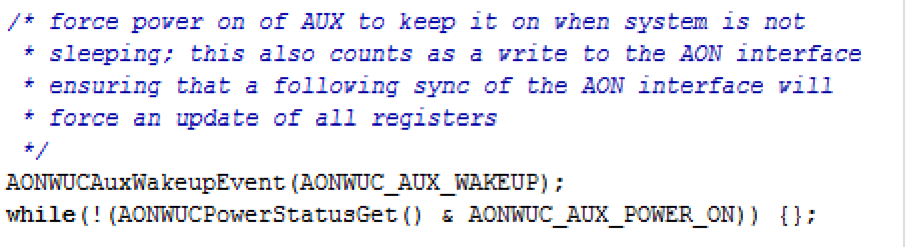
C:\ti\tirtos_cc13xx_cc26xx_#_##_##_##\products\tidrivers_cc13xx_cc26xx_#_##_##_##\packages\ti\drivers\power\Power_calibrateRCOSC.c.

4.6 Auxiliary Domain

Turn on the auxiliary domain for the Cortex-M3 to access the modules in the auxiliary domain, including the OSC_DIG registers to configure the oscillators. Power off the auxilliary domain before entering standby and shutdown modes. For further details on entering and exiting power modes, see Section 3.

4.6.1 Powering on the Auxiliary Domain

The auxiliary domain is forced on through the calls in Figure 19. Until you turn on the domain, the application should not access any modules inside the auxiliary domain or OSC_DIG. The application may complete other tasks while waiting for the auxiliary domain to power up, which takes approximately 50 µs.

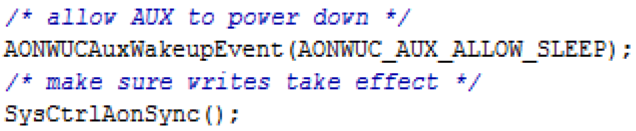


```
/* force power on of AUX to keep it on when system is not
 * sleeping; this also counts as a write to the AON interface
 * ensuring that a following sync of the AON interface will
 * force an update of all registers
 */
AONWUCAuxWakeupEvent(AONWUC_AUX_WAKEUP);
while(!(AONWUCPowerStatusGet() & AONWUC_AUX_POWER_ON)) {};
```

Figure 19. Powering Up the Auxiliary Power Domain

4.6.2 Powering Down the Auxiliary Domain

The Cortex-M3 must release the auxiliary force-on before entering standby and shutdown (not required when entering idle) as in Figure 20. If the sensor controller is running when AONWUCAuxWakeupEvent(...) function is called, the auxiliary domain remains on until the sensor controller idles. The system enters standby mode.



```
/* allow AUX to power down */
AONWUCAuxWakeupEvent(AONWUC_AUX_ALLOW_SLEEP);
/* make sure writes take effect */
SysCtrlAonSync();
```

Figure 20. Powering Down the Auxiliary Domain

4.6.3 Managing the Sensor Controller and the Auxiliary Domain Power

The sensor controller power management is independent of the system power modes in Section 3. The Cortex-M3 can force on the auxiliary domain (which contains the sensor controller) to access modules in the auxiliary domain or the oscillators.

The sensor controller is either active or standby where it executes code or waits for events. When the sensor controller is standing by, it can wake up and execute code and re-enter standby independently of the system state excluding shutdown. The sensor controller may take the rest of the system out of either idle or standby by configuring AON_EVENT:MCUWUSEL[29:24] to use the AUX_SWEV1. When the system state is shutdown, the sensor controller is unavailable. The sensor controller studio software uses AON_EVENT:AUXWUSEL[5:0] to wake up the sensor controller from standby using the RTC_CH2_DLY event.

NOTE: The sensor controller interface driver handles these events.

Table 2. Sensor Controller and Auxiliary Power Management

System State	Available Sensor Controller States
Active	Active or standby
Idle	Active or standby. Sensor controller can wake system from idle.
Standby	Active or standby. Sensor controller can wake system from standby
Shutdown	Not available

4.6.4 Sharing Resources Between the Sensor Controller and the Cortex®-M3

Users can access several resources like the OSC_DIG, TDC, and ADC on the CC26x0 and CC13x0 devices from both the sensor controller and the Cortex-M3. To ensure these accesses are correctly arbitrated on the bus, the Cortex-M3 should always access these modules through driverlib, which has this arbitration support built in.

If the Cortex-M3 and the sensor controller access the same resources, protect them using the auxiliary semaphore module. This module has eight semaphores reserved for protecting-shared resources in the auxiliary domain.

The convention is as follows:

- SMPH0 protects Adi/Ddi bus. This semaphore is built into CC26xxware and CC13xxware implementations.
- SMPH1 protects the TDC. The application enforces this.
- SMPH2 protects the ADC. The application enforces this.

For details on how to use the auxiliary semaphores, see the AUX_SMPH registers.

4.7 RTC

The RTC is clocked from the low-frequency oscillator. The RTC can generate timed events to the Cortex-M3 and wake the device from idle or standby.

The TI-RTOS implementation is available in the Timer.c source file in the TI-RTOS installation:

C:\ti\tirtos_cc13xx_cc26xx_#_##_##_##\products\bios_#_##_##_##\packages\ti\sysbios\family\arm\cc26xx\Timer.c

4.7.1 Initializing RTC

The Timer_start (Timer_Object *obj) TI-RTOS function initializes, resets the RTC, and configures channel 0 to generate interrupts to the Cortex-M3. The Cortex-M3 uses channel 0 because channel 1 is reserved for the RF core and channel 2 is reserved for the sensor controller.

4.7.2 Configuring RTC Compare Events

The Timer_periodicStub(UArg arg) function clears a pending RTC interrupt, sets a new compare value, and calls the appropriate clock tick function. Be careful when configuring a new compare value if the new value is near, because clearing an RTC event takes precedence over setting an event.

Implementation suggestion 1 (perform the AON sync after clearing the event):

1. Clear a pending event (AONRTCEventClear(AON_RTC_CH0)).
2. AON sync (SysCtrlAonSync()) to ensure the event is cleared and no longer active.
3. Set a new compare value. If the new compare value is now or up to 1 second in the past, the event triggers immediately.

Implementation suggestion 2 (TI-RTOS implementation):

1. Clear a pending event (AONRTCEventClear(AON_RTC_CH0)).
2. Set a new compare value adding a margin of four RTC edges (64 µs) to the compare value to ensure the event clears before the new compare value generates an interrupt.

4.8 Debugging Through Power Modes

When debugging software that enters and exits power modes with an emulator attached, the timing is slightly different than when running without an emulator attached. In active mode, the CPU power domain is always on, and the timing is the same. In idle, standby, and shutdown, the CPU power domain remains on when an emulator is attached to keep the debug session active. The wake-up time from these power modes is shorter than if an emulator is not attached to the target.

4.9 Using Peripherals

Peripheral drivers are part of the TI-RTOS installation. The CC26x0 and CC13x0 implementation is at the following path:
C:\ti\tirtos_cc13xx_cc26xx_#_##_##_##\products\tidrivers_cc13xx_cc26xx_#_##_##_##\packages\ti\drivers

For generic peripherals such as UART, SPI, and I^{2}C, TI-RTOS provides a common API for easy migration between different TI architectures.

The CC26x0 and CC13x0 specific drivers are documented in Doxygen format. This format explains how to use the drivers. The CC26x0 power-management document is distributed in the TI-RTOS installation. *TI-RTOS Power Management User's Guide* also explains how the TI-RTOS drivers interact with the power management part of the kernel.

The following rules apply when using any peripheral:

- Only use a peripheral in active or idle power modes. In standby and shutdown, you must turn off the peripheral and its power domain.
- Before accessing a peripheral, do the following:
  1. Turn on the power domain that contains the peripheral.
  2. Enable the run clock for the peripheral through the PRCM registers.
- Reconfigure a peripheral before opening the input and output latches when exiting standby to avoid input and output glitches.

5 References

1. Texas Instruments, *TI-RTOS*
2. Texas Instruments, *TI-RTOS Power Management*, User's Guide
3. Texas Instruments, *CC13x0, CC26x0* , Technical Reference Manual

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from August 4, 2015 to April 17, 2017	Page
• Changed app report title	4
• Updated abstract text	4
• Changed file paths in *Power Management Introduction*	6
• Changed file paths in *Idle Mode*	8
• Changed file paths in *Standby Mode*	9
• Changed file paths in *RC Oscillator Calibration*	22
• Changed file paths in *RTC*	23
• Changed file paths in *Using Peripherals*	24
• Updated document names in *References*	25

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