Application Report
ZHCA635 – May 2015

CC3200 Coretx-M4 核低功耗深度睡眠（LPDS）模式使用指南

Richard Ma *Shenzhen WCS and MCU FAE*

摘要

TI 推出的 Simplelink CC3200 Wi-Fi SoC，其内置的 Cortex-M4 内核支持低功耗深度睡眠（LPDS）模式，可以在保持网络连接的同时，提供最优的功耗。由于其特殊的工作方式，使用该功能时，除了需要用户完成基本的配置，还必须手动地保存和恢复其工作状态。本文主要介绍该模式的使用方法，以及 Cortex-M4 内核及周边主要外设的上下文保存、恢复的过程和方法，帮助用户更清楚地了解 CC3200 LPDS 的工作特性并更灵活地使用该模式。由于该过程比较繁杂，有很多细节需要考虑，本文主要介绍基本的功能及使用方法，具体的细节问题还需要读者在使用过程中逐步完善及优化。另外 CC3200 的网络处理器（NWP）也具有 LPDS 睡眠的能力，但是是由 NWP 自动控制的，故本文不作介绍。

目录

表格

1 CC3200 Cortex-M4 内核 LPDS 模式介绍

低功耗深度睡眠（LPDS）模式是 CC3200 中 Cortex-M4 核的一种重要的低功耗睡眠模式。在该模式下，CC3200 可以在功耗低于 800uA 的条件下保持网络连接。实现网络连接性能和功耗的平衡。

在该模式下，CC3200 的 Cortex-M4 核及绝大部分外设都会掉电，丢失状态及上下文信息，而 RAM 可以有选择地保留数据。在该模式下，CC3200 可以被特定的 GPIO 唤醒，也可以被内置的定时器唤醒；同时，也可以配置为接受网络处理器中断（NWP IRQ）唤醒。

在 LPDS 模式下，可以唤醒 CC3200 的 GPIO 有 6 个，如表格 1 所示。每次支持一路 GPIO 的唤醒。

表格 1. 支持 LPDS 唤醒的 GPIO

GPIO	引脚号
GPIO_02	57
GPIO_04	59
GPIO_11	2
GPIO_13	4
GPIO_17	8
GPIO_24	17

2 LPDS 使用步骤

由于在 LPDS 模式下，内核及外设的状态及上下文信息会丢失，故用户在使用 LPDS 时，除了调用指令进入睡眠，还需自行保留及恢复现场，以保证系统的正常工作。后文会介绍 LPDS 模式的配置方法及现场的保存及恢复步骤，这些步骤支持带操作系统的方式，也支持非操作系统方式。

2.1 实现方法伪代码示例

为便于理解，先使用如下伪代码展示使用 LPDS 模式的大概流程。后文会对具体细节进行介绍。

```
extern void my_enterLPDS(void)
extern void my_exitLPDS(void)


// 配置 LPDS 唤醒源及 RAM 数据维持
void my_setupLPDS()
{
    // 设置唤醒源
    LPDSWakeUpGPIOSelect(GPIO13, HIGH_LEVEL);         //使用 GPIO13 的高电平唤醒
    LPDSWakeupSourceEnable(GPIO);                     //启用 IO 唤醒

    // LPDS 器件维持 RAM 数据
    SRAMRetentionEnable();
}
```

```
// 唤醒后执行的动作
void my_exitLPDS()
{
    // 还原现场
    RESTORE_ARM_REGISTERS();            // 还原内核寄存器
    Restore_NVIC_register();            // 还原一般处理器寄存器

    // 重新启用外设
    PinMuxConfig();         // 重新配置管脚分配
    …
    InitTerm();             // 重新配置 UART
    spi_Open();             // 恢复和 NWP 通信（无需重新初始化 NWP）

}

void my_enterLPDS()
{
    // 备份现场
    Back_up_NVIC_register();            // 保存一般处理器寄存器
    BACK_UP_ARM_REGISTERS();            // 保存内核寄存器

    // 设置唤醒后执行的函数
    LPDSRestoreInfoSet(my_exitLPDS);

    // 进入 LPDS
    PRCMLPDSEnter();
}

// 在需要进入 LPDS 时，调用 my_enterLPDS，即可进入 LPDS
// 当被唤醒后从先执行 my_exitLPDS ，从 my_exitLPDS 退出后，从 my_enterLPDS 的下一行继续执行
void main()
{
    …
    // 设置 LPDS
    my_setupLPDS();

    // 进入 LPDS
    MasterIntDisable();
    my_enterLPDS();
    MasterIntEnable();

    // 退出 LPDS 后，程序从这里继续
    …
}
```

2.2 配置 LPDS 模式

对 LPDS 模式的配置主要包括激活及选择设置 LPDS 的唤醒源及设置内存数据维持范围。

- ***激活及选择设置 LPDS 唤醒源***

使用 PRCMLPDSWakeupSourceEnable 函数激活唤醒源。LPDS 模式的唤醒源包括：GPIO 唤醒（PRCM_LPDS_GPIO），定时器唤醒（PRCM_LPDS_TIMER）及 NWP 中断唤醒（PRCM_LPDS_HOST_IRQ）。示例代码如下：

```
// 使用 GPIO 及 TIMER 唤醒
MAP_PRCMLPDSWakeupSourceEnable(PRCM_LPDS_GPIO | PRCM_LPDS_TIMER);
```

使用不同唤醒源时也要进行相应设置。使用 GPIO 时需要指定唤醒时使用的引脚及触发的电平。

如下代码设置当高电平（PRCM_LPDS_HIGH_LEVEL）时唤醒。另外也支持上升沿（PRCM_LPDS_RISE_EDGE）、下降沿（PRCM_LPDS_FALL_EDGE）及低电平（PRCM_LPDS_LOW_LEVEL）唤醒。

```
// 当 GPIO13 为高电平时，唤醒芯片
MAP_PRCMLPDSWakeUpGPIOSelect(PRCM_LPDS_GPIO13, PRCM_LPDS_HIGH_LEVEL);
```

使用定时器唤醒时，需要设置定时时间。定时器使用 32.768K 时钟，PRCMLPDSIntervalSet 可用于指定唤醒延迟时间。如下例子指定 5 组 32768 个时钟周期（即 5 秒）后，唤醒芯片。

```
// 5s 后，唤醒芯片
MAP_PRCMLPDSIntervalSet(32768 * 5);
```

- ***设置内存数据维持范围***

使用 PRCMSRAMRetentionEnable 函数设置要保存的内存区块。不维持的区块的 RAM 在 LPDS 模式中会掉电丢失数据。由于 CC3200 的程序和数据在运行时都存储在 SRAM 中，故 SRAM 中的数据应小心保存。

示例代码如下，使用 PRCM_SRAM_COL_X 指定维持的区块，每个区块代表 64KB RAM。

```
// 设置在 LPDS 模式下维持全部内存数据
MAP_PRCMSRAMRetentionEnable(
             PRCM_SRAM_COL_1 | PRCM_SRAM_COL_2 | PRCM_SRAM_COL_3 | PRCM_SRAM_COL_4,
             PRCM_SRAM_LPDS_RET);
```

CC3200 最大有 256KB RAM，可指定维持的区块为 PRCM_SRAM_COL_1 到 PRCM_SRAM_COL_4，其分别对应 RAM 的如下地址：

- PRCM_SRAM_COL_1 - 0x2000.0000 to 0x2000.FFFF
- PRCM_SRAM_COL_2 - 0x2001.0000 to 0x2001.FFFF
- PRCM_SRAM_COL_3 - 0x2002.0000 to 0x2002.FFFF
- PRCM_SRAM_COL_4 - 0x2003.0000 to 0x2003.FFFF

在 CCS 中通过配置 cmd 文件可以将程序、数据指定到内存的不同位置。下面的 cmd 文件示例，将 CC3200 的 SRAM 划分为 SRAM_CODE (128K)、SRAM_DATA(64K)及 SRAM_BUF(64K)三部分。可以在程序设计时，把在 LPDS 时不需维持的 buffer 数据放在 SRAM_BUF 部分的数据，以进一步降低功耗。

```
#define RAM_BASE 0x20004000

/* System memory map */

MEMORY
{
    /* Application uses internal RAM for program and data */
    SRAM_CODE (RWX) : origin = 0x20000000, length = 0x20000
    SRAM_DATA (RWX) : origin = 0x20020000, length = 0x10000

    // Audio Buffer
    SRAM_BUF  (RWX) : origin = 0x20030000, length = 0x10000
}

/* Section allocation in memory */

SECTIONS
{
    .intvecs:   > RAM_BASE
    .init_array : > SRAM_CODE
    .vtable :   > SRAM_CODE
    .text   :   > SRAM_CODE
    .const  :   > SRAM_CODE
    .cinit  :   > SRAM_CODE
    .pinit  :   > SRAM_CODE
    .data   :   > SRAM_DATA
    .bss    :   > SRAM_DATA
    .sysmem :   > SRAM_DATA
    .stack  :   > SRAM_DATA(HIGH)

    // buffers for audio
    AUDIO_DATA_BUF : > SRAM_BUF
}
```

下面的代码展示了如何在 CCS 中将 AudioBuf 放进 AUDIO_DATA_BUF 段，上文中 cmd 文件已经将该段放进 SRAM_BUF 部分中。该部分内存完全可以在进入 LPDS 时不维持其内容以进一步降低功耗。

```
#pragma DATA_SECTION (AudioBuf,"AUDIO_DATA_BUF");
const unsigned char AudioBuf[1024];
```

2.3 进入 LPDS

- *关闭全局中断*

为避免在保存现场的过程中受到中断的影响，首先应关闭中断。直接的办法是关闭全局中断，但更加合理的设计是在逐一确认每个中断都被处理清空后，关闭中断。

- ***保存其它外设寄存器***

  如有需要，应保存外设的工作状态，以便在内核唤醒后重新恢复外设的工作。例如芯片的引脚功能分配（PinMux）、SPI 的波特率及其它配置等。另外需注意的是，为避免在 LPDS 下的漏电，应根据原理图逐一为每个引脚配置上下拉以确定电平；或者根据外围电路的需要将引脚置为标准输入作为高阻态。

- ***保存系统控制（System Control）寄存器、中断向量控制器（NVIC）寄存器及系统时钟（SysTick）寄存器***

  系统控制寄存器、中断向量控制器的配置对 MCU 的正常运行十分重要。所以在进入 LPDS 前应保存这些寄存器的状态以便在唤醒后恢复到之前相同的工作状态。但需要注意的是 Cortex-M4 内核的运行状态分为用户态和特权态，部分寄存器只能在特权态下访问，在保存寄存器状态时，应确保内核处于特权态。对于无操作系统的情况，内核会一直在特权态；部分操作系统的一般线程都是用户态，需要切回特权状态后再进行保存及睡眠。

  当使用的 RTOS 通过系统时钟（SysTick）提供定时功能时，需要恢复 SysTick 寄存器。若使用一般定时器（TIMER）作为定时器，则需恢复对应的 TIMER 的寄存器

  如下代码演示了如何在内存中申明一段空间保存对应的寄存器。寄存器都是 32 位的，故对应类型均为 unsigned long。

```
// 设置在 LPDS 模式下维持全部内存数据
typedef struct {
              unsigned long vector_table;   //中断向量表地址
              unsigned long aux_ctrl;       //辅助控制（Auxiliary Control）寄存器
              unsigned long int_ctrl_state; //中断控制及状态（Interrupt Control and State）寄存器
              unsigned long app_int;        //应用中断及复位请求（Application Interrupt Reset
                                            //control）寄存器
              unsigned long sys_ctrl;       //系统控制（System control）寄存器
              unsigned long config_ctrl;    //配置（Configuration control）寄存器
              unsigned long sys_pri_1;      //System Handler Priority 1
              unsigned long sys_pri_2;      //System Handler Priority 2
              unsigned long sys_pri_3;      //System Handler Priority 3
              unsigned long sys_hcrs;       //System Handler control and state register
              unsigned long systick_ctrl;   //系统时钟及状态（SysTick Control Status）寄存器
              unsigned long systick_reload; //系统时钟装载值（SysTick Reload）寄存器
              unsigned long systick_calib;  //系统时钟装校准（SysTick Calibration）寄存器
              unsigned long int_en[6];      //Interrupt set enable
              unsigned long int_priority[49]; //Interrupt priority
              }nvic_regs;

              nvic_regs g_nvic_reg_store;

              void BACK_UP_NVIC_REGISTERS(void)
              {
                      long indx = 0;                  //32-bits
                      unsigned long *base_reg_addr;   //32-bits
                      /* Save the NVIC control registers */
                      g_nvic_reg_store.vector_table = HWREG(NVIC_VTABLE);
                      g_nvic_reg_store.aux_ctrl = HWREG(NVIC_ACTLR);
                      g_nvic_reg_store.int_ctrl_state = HWREG(NVIC_INT_CTRL);
                      g_nvic_reg_store.app_int = HWREG(NVIC_APINT);
```

```
                    g_nvic_reg_store.sys_ctrl = HWREG(NVIC_SYS_CTRL);
                    g_nvic_reg_store.config_ctrl = HWREG(NVIC_CFG_CTRL);
                    g_nvic_reg_store.sys_pri_1 = HWREG(NVIC_SYS_PRI1);
                    g_nvic_reg_store.sys_pri_2 = HWREG(NVIC_SYS_PRI2);
                    g_nvic_reg_store.sys_pri_3 = HWREG(NVIC_SYS_PRI3);
                    g_nvic_reg_store.sys_hcrs = HWREG(NVIC_SYS_HND_CTRL);

                    /* Systick registers */
                    g_nvic_reg_store.systick_ctrl = HWREG(NVIC_ST_CTRL);
                    g_nvic_reg_store.systick_reload = HWREG(NVIC_ST_RELOAD);
                    g_nvic_reg_store.systick_calib = HWREG(NVIC_ST_CAL);

                    /* Save the interrupt enable registers */
                    base_reg_addr = (unsigned long *)NVIC_EN0;
                    for(indx = 0; indx < (sizeof(g_nvic_reg_store.int_en) / 4); indx++) {
                         g_nvic_reg_store.int_en[indx] = base_reg_addr[indx];
                    }

                    /* Save the interrupt priority registers */
                    base_reg_addr = (unsigned long *)NVIC_PRI0;
                    for(  indx = 0; indx < (sizeof(g_nvic_reg_store.int_priority) / 4);
                         indx++)
                    {
                         g_nvic_reg_store.int_priority[indx] = base_reg_addr[indx];
                    }

                    return;
               }
```

- ***保存 Cortex-M4 内核寄存器（Core Registers）***

  内核寄存器包含了内核当前的运行状态以及堆栈等最关键的信息，需要在睡眠前存入内存中。

  要访问到内核寄存器需要使用到汇编语言。如下代码演示了如何使用汇编语言保存内核寄存器，该代码支持 TI CCS 环境：

```
typedef struct  {
               unsigned long     msp;
               unsigned long     psp;
               unsigned long     psr;
               unsigned long     primask;
               unsigned long     faultmask;
               unsigned long     basepri;
               unsigned long     control;
}arm_CM4_core_regs;

arm_CM4_core_regs g_vault_arm_registers;

#ifdef ccs
#define BACK_UP_ARM_REGISTERS() {     \
        __asm(" push {r0-r12,LR} \n" \
          " movw r1, g_vault_arm_registers \n" \
          " movt r1, g_vault_arm_registers \n" \
          " mrs  r0,msp \n" \
          " str  r0,[r1] \n" \
          " mrs  r0,psp \n" \
          " str  r0,[r1, #4] \n" \
          " mrs  r0,primask \n" \
```

```
          " str  r0,[r1, #12] \n" \
          " mrs  r0,faultmask \n" \
          " str  r0,[r1, #16] \n" \
          " mrs  r0,basepri \n" \
          " str  r0,[r1, #20] \n" \
          " mrs  r0,control \n" \
          " str  r0,[r1, #24] \n"); \
          }
#endif
```

- ***设置唤醒时的程序计数器（PC）指针及堆栈（SP）指针***

  在进入 LPDS 前，CC3200 的 PRCM 会保存两个关键信息，即程序计数器（PC）指针及堆栈（SP）指针，在 Cortex-M4 被唤醒后，会首先设置堆栈寄存器并跳转去保存的程序地址。

  如下代码演示了如何把之前保存的堆栈地址并指定唤醒后执行的函数。须注意的是，如果函数是在用户空间进入 LPDS 的，返回时应使用 PSP 指针；若在特权空间进入 LPDS 的，应使用 MSP 指针。

  为程序计数器赋函数指针可以在唤醒后直接跳转至该函数，不同的是该过程只有跳转没有保存现场。故当从该函数返回时，相当于从调用 PRCMLPDSEnter()函数的函数中返回。

```
// 设置唤醒后执行 my_exitLPDS
MAP_PRCMLPDSRestoreInfoSet(g_vault_arm_registers.msp, (unsigned int)my_exitLPDS);
```

- ***进入 LPDS***

  调用 PRCMLPDSEnter()即可进入 LPDS。

2.4 从 LPDS 唤醒

当从 LPDS 唤醒后，Cortex-M4 核需要着手恢复之前的工作状态，可以分为如下的步骤。这些工作需要在 PRCMLPDSRestoreInfoSet 函数设置的唤醒后的函数中实现

- ***恢复 Cortex-M4 内核寄存器（Core Register）***

  恢复方法和保存方法类似。

```
#ifdef ccs
#define RESTORE_ARM_REGISTERS() {        \
    __asm(" movw r1, g_vault_arm_registers \n" \
          " movt r1, g_vault_arm_registers \n" \
          " ldr  r0,[r1, #24] \n" \
          " msr  control,r0 \n" \
          " ldr  r0,[r1] \n" \
          " msr  msp,r0 \n" \
          " ldr  r0,[r1,#4] \n" \
          " msr  psp,r0 \n" \
```

```
          " ldr  r0,[r1, #12] \n" \
          " msr  primask,r0 \n" \
          " ldr  r0,[r1, #16] \n" \
          " msr  faultmask,r0 \n" \
          " ldr  r0,[r1, #20] \n" \
          " msr  basepri,r0 \n" \
          " pop  {r0-r12,LR} \n"); \
         }
#endif
```

- ***恢复系统控制（System Control）寄存器、中断向量控制器（NVIC）寄存器及系统时钟（SysTick）寄存器***

恢复方法和保存方法类似。

```
void RESTORE_NVIC_REGISTERS(void)
{
        long indx = 0;                                                  //32-bits
        unsigned long *base_reg_addr;             //32-bits

        /* Restore the NVIC control registers */
        HWREG(NVIC_VTABLE) = g_nvic_reg_store.vector_table;
        HWREG(NVIC_ACTLR) = g_nvic_reg_store.aux_ctrl;
        HWREG(NVIC_APINT) = g_nvic_reg_store.app_int;
        HWREG(NVIC_SYS_CTRL) = g_nvic_reg_store.sys_ctrl;
        HWREG(NVIC_CFG_CTRL) = g_nvic_reg_store.config_ctrl;
        HWREG(NVIC_SYS_PRI1) = g_nvic_reg_store.sys_pri_1;
        HWREG(NVIC_SYS_PRI2) = g_nvic_reg_store.sys_pri_2;
        HWREG(NVIC_SYS_PRI3) = g_nvic_reg_store.sys_pri_3;
        HWREG(NVIC_SYS_HND_CTRL) = g_nvic_reg_store.sys_hcrs;

        /* Systick registers */
        HWREG(NVIC_ST_CTRL) = g_nvic_reg_store.systick_ctrl;
        HWREG(NVIC_ST_RELOAD) = g_nvic_reg_store.systick_reload;
        HWREG(NVIC_ST_CAL) = g_nvic_reg_store.systick_calib;

        /* Restore the interrupt priority registers */
        base_reg_addr = (unsigned long *)NVIC_PRI0;
        for(indx = 0; indx < (sizeof(g_nvic_reg_store.int_priority) / 4); indx++) {
                base_reg_addr[indx] = g_nvic_reg_store.int_priority[indx];
        }

        /* Restore the interrupt enable registers */
        base_reg_addr = (unsigned long *)NVIC_EN0;
        for(indx = 0; indx < (sizeof(g_nvic_reg_store.int_en) / 4); indx++) {
                base_reg_addr[indx] = g_nvic_reg_store.int_en[indx];
        }

        INTRODUCE_SYNC_BARRIER(); /* Data and instruction sync barriers */

        return;
}
```

- ***恢复和网络处理器通信的 SPI 接口寄存器***

  当 NWP 已经在正常工作时，进入 LPDS 不会影响其工作；从 LPDS 模式中唤醒时，无需重新初始化 NWP，只用重新初始化和网络处理器间的 SPI 工作即可。恢复方法参考如下代码：

```
Fd_t spi_Open(char *ifName, unsigned long flags)
{
    unsigned long ulBase;
    unsigned long ulSpiBitRate;
    tROMVersion* pRomVersion = (tROMVersion *)(ROM_VERSION_ADDR);


    //NWP master interface
    ulBase = LSPI_BASE;

    //Enable MCSPIA2
    MAP_PRCMPeripheralClkEnable(PRCM_LSPI,PRCM_RUN_MODE_CLK|PRCM_SLP_MODE_CLK);

    //Disable Chip Select
    MAP_SPICSDisable(ulBase);

    //Disable SPI Channel
    MAP_SPIDisable(ulBase);

    // Reset SPI
    MAP_SPIReset(ulBase);

    //
    // Configure SPI interface
    //

    if(pRomVersion->ucMinorVerNum == ROM_VER_PG1_21 )
    {
          ulSpiBitRate = SPI_RATE_13M;
    }
    else if(pRomVersion->ucMinorVerNum == ROM_VER_PG1_32)
    {
          ulSpiBitRate = SPI_RATE_13M;
    }
    else if(pRomVersion->ucMinorVerNum >= ROM_VER_PG1_33)
    {
          ulSpiBitRate = SPI_RATE_20M;
    }

    MAP_SPIConfigSetExpClk(ulBase,
          MAP_PRCMPeripheralClockGet(PRCM_LSPI),
          ulSpiBitRate,SPI_MODE_MASTER,SPI_SUB_MODE_0,
          ( SPI_SW_CTRL_CS | SPI_4PIN_MODE | SPI_TURBO_OFF
          | SPI_CS_ACTIVEHIGH | SPI_WL_32));

    if(MAP_PRCMPeripheralStatusGet(PRCM_UDMA))
    {
          g_ucDMAEnabled = (HWREG(UDMA_BASE + UDMA_O_CTLBASE) != 0x0) ? 1 : 0;
    }
    else
    {
          g_ucDMAEnabled = 0;
    }
#ifdef SL_CPU_MODE
    g_ucDMAEnabled = 0;
```

```
#endif
    if(g_ucDMAEnabled)
    {
          memset(g_ucDinDout,0xFF,sizeof(g_ucDinDout));

          // Set DMA channel
          cc_UDMAChannelSelect(UDMA_CH12_LSPI_RX);
          cc_UDMAChannelSelect(UDMA_CH13_LSPI_TX);

          MAP_SPIFIFOEnable(ulBase,SPI_RX_FIFO);
          MAP_SPIFIFOEnable(ulBase,SPI_TX_FIFO);
          MAP_SPIDmaEnable(ulBase,SPI_RX_DMA);
          MAP_SPIDmaEnable(ulBase,SPI_TX_DMA);

          MAP_SPIFIFOLevelSet(ulBase,1,1);

#if defined(SL_PLATFORM_MULTI_THREADED)
          osi_InterruptRegister(INT_LSPI,
                       (P_OSI_INTR_ENTRY)DmaSpiSwIntHandler,INT_PRIORITY_LVL_1);
                       MAP_SPIIntEnable(ulBase,SPI_INT_EOW);


          osi_MsgQCreate(&DMAMsgQ,"DMAQueue",sizeof(int),1);
#else
          MAP_IntRegister(INT_LSPI,(void(*)(void))DmaSpiSwIntHandler);
          MAP_IntPrioritySet(INT_LSPI, INT_PRIORITY_LVL_1);
          MAP_IntEnable(INT_LSPI);

          MAP_SPIIntEnable(ulBase,SPI_INT_EOW);


          g_cDummy = 0x0;
#endif

    }
    MAP_SPIEnable(ulBase);

    g_SpiFd = 1;
    return g_SpiFd;
}
```

- ***恢复其它外设寄存器***

  由于外设均已被复位，故需要逐一重新初始化外设并还原至之前的工作状态。需要还原的配置如 PinMux 分配、GPIO 设置等等。

  CC3200 在正常上电时，NWP 会初始化将引脚的全部控制权交给 Cortex-M4 核。在 LPDS 唤醒时，不会经过这个步骤，所以需要使用如下代码手工取得 GPIO 的全部控制权，否则会导致 GPIO 中断无响应。

```
// Take GPIO semaphore
ulRegVal = HWREG(0x400F703C);
ulRegVal = (ulRegVal & ~0x3FF) | 0x155;
HWREG(0x400F703C) = ulRegVal;
```

- ***激活全局中断***

  恢复全局中断，回到正常工作状态。

3 总结

CC3200 LPDS 模式的使用非常灵活且有非常多的细节需要考虑。所幸现在 CC3200 SDK 中已经提供了一个现成的电源管理框架中间件(Middleware of Power Management)来处理各种细节问题，需要使用 LPDS 模式的用户可以直接参考和使用 TI 提供的代码。针对使用 RTOS 的用户，在带操作系统的环境下，LPDS 模式的使用会有更多的细节需要推敲，建议在了解 LPDS 的基本使用原理的基础上，优先使用或者移植该中间件以减小工作量。

参考文档

1. *Cortex™-M4 Devices Generic User Guide*
2. *CC3200 MCU Technical Reference Manual*
3. *CC3200 Power Management Framework*
4. *CC3200 Peripheral Driver Library User's Guide*

重要声明

德州仪器(TI) 及其下属子公司有权根据 JESD46 最新标准, 对所提供的产品和服务进行更正、修改、增强、改进或其它更改， 并有权根据 JESD48 最新标准中止提供任何产品和服务。客户在下订单前应获取最新的相关信息, 并验证这些信息是否完整且是最新的。所有产品的销售都遵循在订单确认时所提供的TI 销售条款与条件。

TI 保证其所销售的组件的性能符合产品销售时 TI 半导体产品销售条件与条款的适用规范。仅在 TI 保证的范围内，且 TI 认为 有必要时才会使用测试或其它质量控制技术。除非适用法律做出了硬性规定，否则没有必要对每种组件的所有参数进行测试。

TI 对应用帮助或客户产品设计不承担任何义务。客户应对其使用 TI 组件的产品和应用自行负责。为尽量减小与客户产品和应 用相关的风险，客户应提供充分的设计与操作安全措施。

TI 不对任何 TI 专利权、版权、屏蔽作品权或其它与使用了 TI 组件或服务的组合设备、机器或流程相关的 TI 知识产权中授予 的直接或隐含权限作出任何保证或解释。TI 所发布的与第三方产品或服务有关的信息，不能构成从 TI 获得使用这些产品或服 务的许可、授权、或认可。使用此类信息可能需要获得第三方的专利权或其它知识产权方面的许可，或是 TI 的专利权或其它 知识产权方面的许可。

对于 TI 的产品手册或数据表中 TI 信息的重要部分，仅在没有对内容进行任何篡改且带有相关授权、条件、限制和声明的情况 下才允许进行复制。TI 对此类篡改过的文件不承担任何责任或义务。复制第三方的信息可能需要服从额外的限制条件。

在转售 TI 组件或服务时，如果对该组件或服务参数的陈述与 TI 标明的参数相比存在差异或虚假成分，则会失去相关 TI 组件 或服务的所有明示或暗示授权，且这是不正当的、欺诈性商业行为。TI 对任何此类虚假陈述均不承担任何责任或义务。

客户认可并同意，尽管任何应用相关信息或支持仍可能由 TI 提供，但他们将独力负责满足与其产品及在其应用中使用 TI 产品 相关的所有法律、法规和安全相关要求。客户声明并同意，他们具备制定与实施安全措施所需的全部专业技术和知识，可预见 故障的危险后果、监测故障及其后果、降低有可能造成人身伤害的故障的发生机率并采取适当的补救措施。客户将全额赔偿因 在此类安全关键应用中使用任何 TI 组件而对 TI 及其代理造成的任何损失。

在某些场合中，为了推进安全相关应用有可能对 TI 组件进行特别的促销。TI 的目标是利用此类组件帮助客户设计和创立其特 有的可满足适用的功能安全性标准和要求的终端产品解决方案。尽管如此，此类组件仍然服从这些条款。

TI 组件未获得用于 FDA Class III（或类似的生命攸关医疗设备）的授权许可，除非各方授权官员已经达成了专门管控此类使 用的特别协议。

只有那些 TI 特别注明属于军用等级或“增强型塑料”的 TI 组件才是设计或专门用于军事/航空应用或环境的。购买者认可并同 意，对并非指定面向军事或航空航天用途的 TI 组件进行军事或航空航天方面的应用，其风险由客户单独承担，并且由客户独 力负责满足与此类使用相关的所有法律和法规要求。

TI 已明确指定符合 ISO/TS16949 要求的产品，这些产品主要用于汽车。在任何情况下，因使用非指定产品而无法达到 ISO/TS16949 要求，TI不承担任何责任。

	产品		应用
数字音频	www.ti.com.cn/audio	通信与电信	www.ti.com.cn/telecom
放大器和线性器件	www.ti.com.cn/amplifiers	计算机及周边	www.ti.com.cn/computer
数据转换器	www.ti.com.cn/dataconverters	消费电子	www.ti.com/consumer-apps
DLP® 产品	www.dlp.com	能源	www.ti.com/energy
DSP - 数字信号处理器	www.ti.com.cn/dsp	工业应用	www.ti.com.cn/industrial
时钟和计时器	www.ti.com.cn/clockandtimers	医疗电子	www.ti.com.cn/medical
接口	www.ti.com.cn/interface	安防应用	www.ti.com.cn/security
逻辑	www.ti.com.cn/logic	汽车电子	www.ti.com.cn/automotive
电源管理	www.ti.com.cn/power	视频和影像	www.ti.com.cn/video
微控制器 (MCU)	www.ti.com.cn/microcontrollers		
RFID 系统	www.ti.com.cn/rfidsys		
OMAP应用处理器	www.ti.com/omap		
无线连通性	www.ti.com.cn/wirelessconnectivity	德州仪器在线技术支持社区	www.deyisupport.com

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