e Semiconductor
应用笔记

Document Number: AN4447
Rev 0, 01/2012

Freescale MQX 低功耗管理

作者: Derek Snell
现场应用工程师

内容

1 简介

飞思卡尔提供丰富的低功耗微控制器（MCU），包括 32 位 Kinetis 系列。Kinetis 具有多种不同的低功耗工作模式，提供超低待机和运行电流消耗。

飞思卡尔还提供 MQX™，一款全功能的免费实时操作系统（RTOS）。从版本 3.8 开始，MQX 集成了低功耗管理（LPM）驱动程序，可以在 MQX 应用程序中利用低功耗工作模式。本应用笔记说明了 MQX LPM，及其工作原理和使用方法。文中特别介绍了 Kinetis K60 系列和塔式 TWR-K60N512 开发板和板级支持包（BSP）。

LPM 还支持其他 MCU 和电路板；详细信息请参见最新的 MQX 版本记录。本文档还说明了 Kinetis 低功耗模式。关于这些功耗模式的更多详情，请参见 Kinetis 器件的参考手册和《Kinetis 快速参考用户指南》(搜索 KQRUG)。

2 LPM 概述

LPM 是一个 MQX 驱动程序，包含在特定 BSP 中。它可以使应用程序轻松改变工作模式，从而利用 MCU 的低功耗模式。LPM 由特定 BSP 的工作模式进行配置，这些工作模式会映射到 MCU 的 CPU 功耗模式。

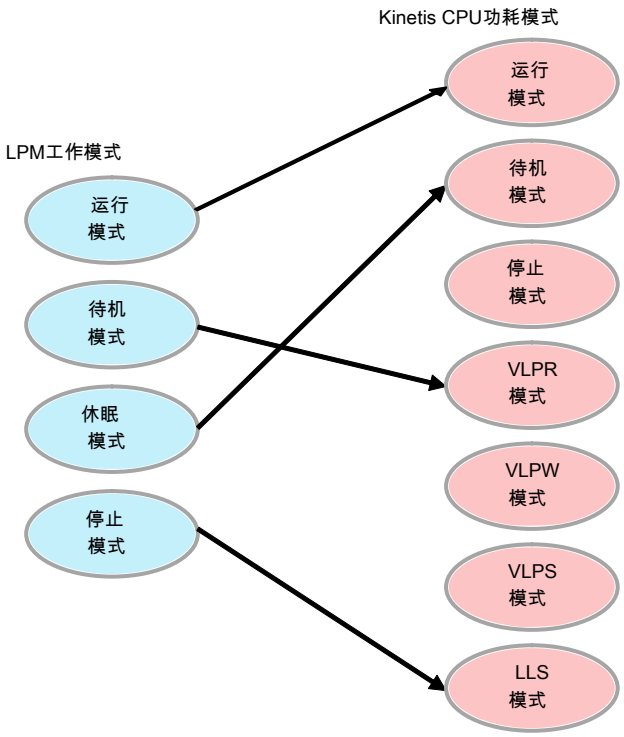

图 1. K60 BSP 的 LPM 工作模式映射到 CPU 功耗模式

LPM 还为 BSP 中的其他驱动程序提供了一种机制，用于注册 LPM。一旦注册成功，当应用程序改变工作模式或时钟配置时，这些驱动程序就会通过回调函数收到通知。这样，驱动程序可以关闭外设以降低功耗，或修改外设设定以适应变化。例如，如果应用程序降低了时钟频率以降低功耗，则串行 UART 驱动程序可以更新波特率分频器寄存器，从而在降低的时钟频率下保持相同的波特率。这种架构有助于将应用程序与底层硬件分离，并使应用程序更方便移植。

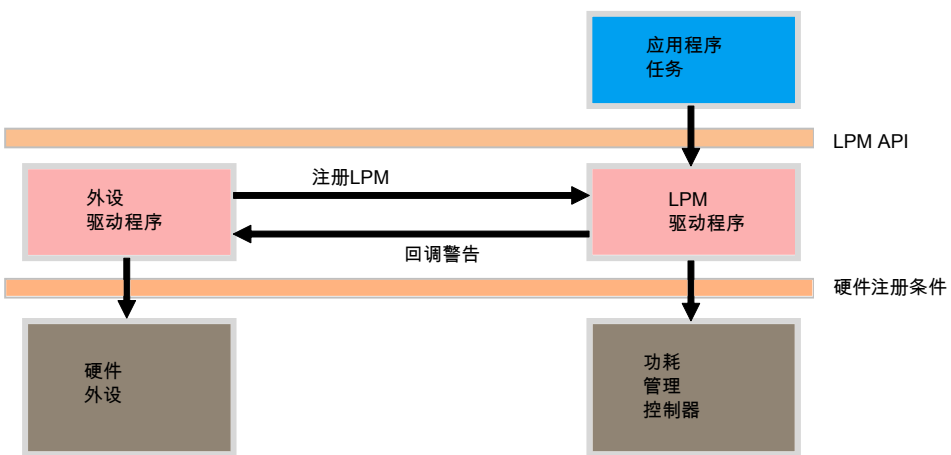

图 2. LPM 架构

3 LPM 示例

MQX 包含示例项目，用于评估 LPM 并帮助您了解使用方法。可从以下路径找到这些低功耗示例，其中包含适用于多种 BSP 的项目：<MQX 安装目录>\mqx\examples\lowpower。

示例按顺序执行不同的 BSP 工作模式。可以在开发板上测量各种低功耗模式下的电流消耗。

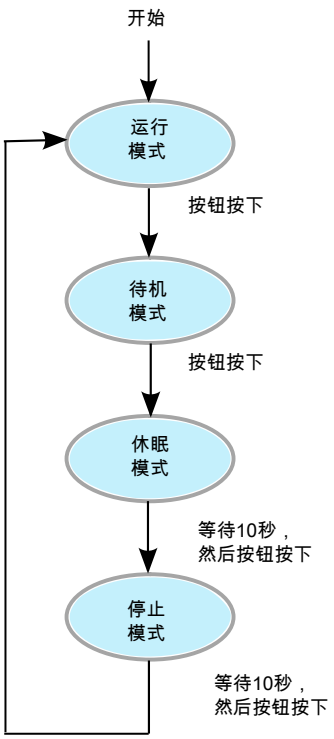

图 3. 低功耗示例流程图

例如，K60 示例从运行模式开始，BSP 在全速时钟状态下运行。通过调用 print 函数使用 BSP 的默认 UART（连接到终端）打印信息，为用户逐步说明示例内容。等待按钮按下后，示例会转为 LPM 待机模式，该模式映射到 Kinetis 的超低功耗运行（VLPR）模式。内核时钟频率降至 2 MHz 以降低功耗，但内核仍然处于运行状态。LPM 更新串行驱动程序以改变 UART 波特率分频器，从而使波特率保持不变。

在示例改变模式之前，终端显示 Kinetis 低漏电唤醒单元（LLWU）的唤醒设定。示例通过调用 API _lpm_set_clock_configuration()来改变模式，其传递的参数是 BSP 的时钟配置值。在这种情况下，该配置等于 2 MHz 时钟。一旦时钟频率改变且串行驱动程序更新，示例会调用_lpm_set_operation_mode()。这里传递的参数是代表 LPM 运行模式的值。此时将改变为 LPM 待机模式。

示例再次等待按钮按下，然后时钟变回默认的 BSP 频率。这一次，示例改变为 LPM 休眠工作模式，映射到 Kinetis 的待机模式。在该模式下，内核停止运行，并等待事件使其再次运行。此时，事件为实时时钟（RTC）或 UART 中断。示例使用 RTC 驱动程序设置一个 10 秒的闹钟，在 10 秒内仍处于休眠模式。10 秒后，示例唤醒，并输出一条消息，将等待下一次按钮按下。UART 唤醒中断也可将演示从休眠模式唤醒。要实现该中断，BSP 需要修改所使用的串行驱动程序的中断版本（ittyX）。

本示例的最后一种模式为 LPM 停止模式，映射到 Kinetis 的低漏电停止（LLS）模式。在该模式下禁止 UART，以助于降低功耗。LLWU 设置为允许使用 RTC 来唤醒 MCU。同样，示例通过 RTC 驱动程序设置 10 秒警报，然后进入停止模式。唤醒后，UART 再次启用；在按钮按下后，示例将再次从头开始执行序列。

4 LPM 驱动程序详细信息

LPM 驱动程序包含在 MQX BSP 中，允许应用程序方便地修改时钟设定和低功耗模式。应用程序只需调用 _lpm_set_operation_mode()或_lpm_set_clock_configuration()便能修改 LPM 状态。

在本应用笔记最后的附录中详细说明了 LPM API。LPM 还允许其他驱动程序注册 LPM，以便在 LPM 改变时钟设定或低功耗模式时获取通知。LPM 的大部分源代码可从<MQX 安装目录>\mqx\source\io\lpm 中获得。

4.1 CPU 功耗模式

LPM 需要可用 CPU 低功耗模式的列表。BSP 将 LPM 工作模式映射到 CPU 低功耗模式，这将在“BSP 中的 LPM 使用”章节中进一步讨论。对于 K60 BSP，CPU 功耗模式列表在下列 lpm_kinetis.c 文件中定义：

```
static const LPM_CPU_POWER_MODE LPM_CPU_POWER_MODES[LPM_CPU_POWER_MODES_KINETIS] =
{
    // Kinetis RUN
    {
        MC_PMCTRL_LPWUI_MASK,                       // voltage regulator ON after wakeup
        0,                                          // Mode flags == clear settings
    },
    // Kinetis WAIT
    {
        MC_PMCTRL_LPWUI_MASK,                       // voltage regulator ON after wakeup
        LPM_CPU_POWER_MODE_FLAG_USE_WFI,            // Mode flags == execute WFI
    },
    // Kinetis STOP
    {
        MC_PMCTRL_LPWUI_MASK,                       // voltage regulator ON after wakeup
        LPM_CPU_POWER_MODE_FLAG_DEEP_SLEEP
| LPM_CPU_POWER_MODE_FLAG_USE_WFI,                 // Mode flags == deepsleep, execute WFI
    },
    // Kinetis VLPR
    {
        MC_PMCTRL_RUNM(2),                          // VLPR
        0,                                          // Mode flags == clear settings
    },
    // Kinetis VLPW
    {
        MC_PMCTRL_RUNM(2),                          // VLPW
        LPM_CPU_POWER_MODE_FLAG_USE_WFI,            // Mode flags == execute WFI
    },
    // Kinetis VLPS
    {
        MC_PMCTRL_LPLLSM(2),                        // VLPS
        LPM_CPU_POWER_MODE_FLAG_DEEP_SLEEP
| LPM_CPU_POWER_MODE_FLAG_USE_WFI,                 // Mode flags == deepsleep, execute WFI
    },
    // Kinetis LLS
    {
        MC_PMCTRL_LPWUI_MASK | MC_PMCTRL_LPLLSM(3), // voltage regulator ON after wakeup, LLS
        LPM_CPU_POWER_MODE_FLAG_DEEP_SLEEP
| LPM_CPU_POWER_MODE_FLAG_USE_WFI,                 // Mode flags == deepsleep, execute WFI
    }
};
```

该数组中的每种功耗模式均为 LPM_CPU_POWER_MODE 类型，并在下列 lpm_kinetis.h 文件中定义。将 PMCTRL 中的值写入 Kinetis 寄存器 MC_PMCTRL，用于控制 MCU 的功耗模式。FLAGS 字段是 LPM 驱动程序在更改 CPU 功耗模式时使用的一组位。

```
typedef struct lpm_cpu_power_mode {
    /* Mode control register setup */
    uint_8      PMCTRL;

    /* Flags specifying low power mode behavior */
    uint_8      FLAGS;

} LPM_CPU_POWER_MODE, _PTR_ LPM_CPU_POWER_MODE_PTR;
```

4.2 lpm_state_struct

LPM 的状态由类型为 LPM_STATE_STRUCT 的全局结构 lpm_state_struct 维护。该结构在 lpm_prv.h 中声明，全局变量在 lpm.c 中声明。该结构持续跟踪 LPM 工作模式、可用 CPU 低功耗模式列表、注册 LPM 的其他驱动程序和 LPM 使用的信号量。该结构由_lpm_install()函数初始化，通过其他 LPM API 进行更新。结构定义如下：

```
typedef struct lpm_state_struct {
    /* CPU core operation mode behavior specification */
    const LPM_CPU_OPERATION_MODE _PTR_ CPU_OPERATION_MODES;

    /* Current system operation mode */
    LPM_OPERATION_MODE                 OPERATION_MODE;

    /* List of registered drivers */
    LPM_DRIVER_ENTRY_STRUCT_PTR        DRIVER_ENTRIES;

    /* Unique ID counter */
    _mqx_uint                          COUNTER;

    /* LPM functions synchronization */
    LWSEM_STRUCT                       SEMAPHORE;

} LPM_STATE_STRUCT, _PTR_ LPM_STATE_STRUCT_PTR;
```

4.3 经 LPM 注册的驱动程序

在 MQX BSP 中的其他驱动程序可以注册 LPM，以便在 LPM 改变时钟配置或低功耗模式时获取通知。驱动程序通过调用_lpm_register_driver()来注册 LPM。该 API 中的一个参数为 LPM_REGISTRATION_STRUCT 类型的结构，通过如下方式在 lpm.h 中定义：

```
typedef struct lpm_registration_struct {
    /* Callback called when system clock configuration changes */
    LPM_NOTIFICATION_CALLBACK CLOCK_CONFIGURATION_CALLBACK;

    /* Callback called when system operation mode changes */
    LPM_NOTIFICATION_CALLBACK OPERATION_MODE_CALLBACK;

    /* The order (priority) of notifications among other drivers */
    _mqx_uint                   DEPENDENCY_LEVEL;

} LPM_REGISTRATION_STRUCT, _PTR_ LPM_REGISTRATION_STRUCT_PTR;
```

4.3.1 用于已注册驱动程序的回调函数

LPM 可为注册驱动程序使用两种不同的回调函数。当 LPM 执行_lpm_set_clock_configuration() API 时，调用 CLOCK_CONFIGURATION_CALLBACK 函数。当 LPM 执行_lpm_set_operation_mode()来改变低功耗模式时，调用 OPERATION_MODE_CALLBACK 函数。LPM 在更改时钟配置的前后均会调用时钟配置回调。这样，已注册的驱动程序可以事先为更改做好准备，并在更改完成后进行更新。工作模式回调仅在模式更改前调用。当已注册的驱动程序获取其回调时，LPM 会向该回调传递两个参数：通知结构和驱动程序专用数据指针，如下所示。

```
(
        /* [IN] Low power notification */
        LPM_NOTIFICATION_STRUCT_PTR      notification,

        /* [IN/OUT] Driver specific data pointer */
        pointer                          device_specific_data
    )
```

传递给回调的通知结构为 LPM_NOTIFICATION_STRUCT 类型，并在以下 lpm.h 中定义。NOTIFICATION_TYPE 的值可为 LPM_NOTIFICATION_TYPE_PRE 或 LPM_NOTIFICATION_TYPE_POST，均在 lpm.h 中定义。根据传递给回调的通知类型，已注册的驱动程序即可了解回调是在 LPM 执行更改之前还是之后执行。此外还会传递 OPERATION_MODE 和 CLOCK_CONFIGURATION，以将新状态告知已注册的驱动程序。

```
typedef struct lpm_notification_struct {
    /* When the notification happens */
    LPM_NOTIFICATION_TYPE   NOTIFICATION_TYPE;

    /* Current system operation mode */
    LPM_OPERATION_MODE      OPERATION_MODE;

    /* Current system clock configuration */
    BSP_CLOCK_CONFIGURATION CLOCK_CONFIGURATION;

} LPM_NOTIFICATION_STRUCT, _PTR_ LPM_NOTIFICATION_STRUCT_PTR;
```

4.3.2 已注册驱动程序的依赖关系

当驱动程序通过_lpm_register_driver() API 注册 LPM 时，LPM_REGISTRATION_STRUCT 中的一个成员变量 DEPENDENCY_LEVEL 值将传递到 LPM。该变量值使 LPM 可以根据依赖性或优先级对已注册的驱动程序列表进行排序。依赖性级别较低的驱动程序早于级别较高的驱动程序获取 PRE 通知。当 LPM 改变状态时，已注册驱动程序的回调会根据该有序列表执行。

这里是驱动程序依赖性级别的一个示例。K60 具有 sdcard:驱动程序，为物理接口使用外设驱动程序 esdhc:。如果在 LPM 修改时钟配置的应用程序中均使用了这两个驱动程序，则它们需与 LPM 注册，且 esdhc:的依赖性级别高于 sdcard:。当应用程序使用 LPM API 来修改时钟配置时，回调函数会按照如下图 4 中的顺序发生。

1. 首先，sdcard:驱动程序接收 PRE 通知回调，因为其依赖性级别较低。此处的意图是在波特率变化时阻止通信，并防止使用低级驱动程序。
2. 接着，esdhc:驱动程序接收 PRE 通知回调，等待所有进行中的传输完成。
3. 然后，LPM 会修改 BSP 的时钟配置，并开始 POST 通知回调。POST 回调以与 PRE 回调相反的顺序发生。
4. esdhc:驱动程序接收到第一个 POST 回调，并用新的时钟频率更新 SDHC 外设。
5. 然后 sdcard:驱动程序获取 POST 回调来重新启用驱动程序，时钟配置更改至此完成。

如果时钟配置修改失败，也会使用该 POST 通知顺序。此时，LPM 利用 POST 回调顺序对发生失败前的时钟配置修改进行恢复。

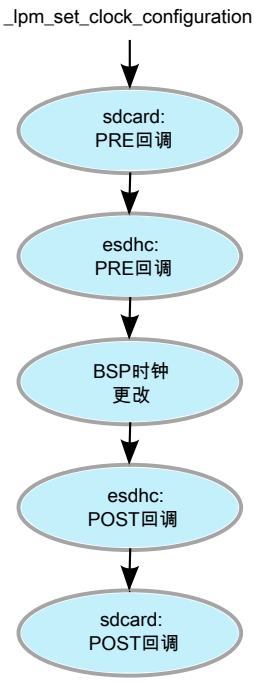

图 4. 已注册驱动程序依赖性的时钟修改示例

在 lpm_state_struct 中，LPM 通过 DRIVER_ENTRIES 指针对已注册驱动程序的链表进行维护。当驱动程序通过 _lpm_register_driver()注册 LPM 时，LPM 会根据依赖等级将驱动程序置于有序列表的适当位置。

5 BSP 中的 LPM 使用

MQX BSP 通常会根据最终应用加以修改。

本节讨论了 K60 BSP 如何使用 LPM，以及如何对其进行修改。

首先，要使用 LPM，需要在 user_config.h 中定义宏 MQX_ENABLE_LOW_POWER，并且需要重新编译 BSP 库。关于该流程的更多信息，请参见 MQX 文档。

5.1 LPM_CPU_OPERATION_MODES 数组

BSP 使用文件配置各驱动程序，对于 LPM，该文件为：<MQX 安装路径> \mqx\source\bsp\twrk60n512\init_lpm.c。

该文件包含 LPM_CPU_OPERATION_MODES 数组，用于定义各 LPM 工作模式。K60 文件如下所示：

```
const LPM_CPU_OPERATION_MODE LPM_CPU_OPERATION_MODES[LPM_OPERATION_MODES] =
{
    // LPM_OPERATION_MODE_RUN
    {
        LPM_CPU_POWER_MODE_KINETIS_RUN,         // Index of predefined mode
        0,                                      // Additional mode flags
        0,                                      // Mode wake up events from pins 0..3
        0,                                      // Mode wake up events from pins 4..7
        0,                                      // Mode wake up events from pins 8..11
        0,                                      // Mode wake up events from pins 12..15
        0                                       // Mode wake up events from internal sources
    },
    // LPM_OPERATION_MODE_WAIT
    {
```

```
        LPM_CPU_POWER_MODE_KINETIS_VLPR,        // Index of predefined mode
        0,                                      // Additional mode flags
        0,                                      // Mode wake up events from pins 0..3
        0,                                      // Mode wake up events from pins 4..7
        0,                                      // Mode wake up events from pins 8..11
        0,                                      // Mode wake up events from pins 12..15
        0                                       // Mode wake up events from internal sources
    },
    // LPM_OPERATION_MODE_SLEEP
    {
        LPM_CPU_POWER_MODE_KINETIS_WAIT,        // Index of predefined mode
        LPM_CPU_POWER_MODE_FLAG_SLEEP_ON_EXIT, // Additional mode flags
        0,                                      // Mode wake up events from pins 0..3
        0,                                      // Mode wake up events from pins 4..7
        0,                                      // Mode wake up events from pins 8..11
        0,                                      // Mode wake up events from pins 12..15
        0                                       // Mode wake up events from internal sources
    },
    // LPM_OPERATION_MODE_STOP
    {
        LPM_CPU_POWER_MODE_KINETIS_LLS,         // Index of predefined mode
        0,                                      // Additional mode flags
        0,                                      // Mode wake up events from pins 0..3
        0,                                      // Mode wake up events from pins 4..7
        LLWU_PE3_WUPE9(1),                      // Mode wake up events from pins - SW3 rising
        0,                                      // Mode wake up events from pins 12..15
        LLWU_ME_WUME5_MASK                      // Mode wake up events from sources - RTC
    }
};
```

该数组中的每个元素均为 LPM_CPU_OPERATION_MODE 类型，结构在 lpm_kinetis.h 中定义如下。MODE_INDEX 字段必须为 LPM_CPU_POWER_MODES 数组中定义的 CPU 功耗模式之一。这创建了用于将 LPM 工作模式映射到 CPU 功耗模式的链接。PEx 和 ME 字段为 Kinetis LLWU_PEx 和 LLWU_ME 寄存器中对应于这些工作模式的值。这些设定控制低功耗模式下的 LLWU 唤醒源。FLAGS 字段用于附加标志。可修改这些 BSP 设定来使用不同的 CPU 功耗模式和唤醒源，从而满足应用需要。

```
typedef struct lpm_cpu_operation_mode {
    /* Index into predefined cpu operation modes */
    LPM_CPU_POWER_MODE_KINETIS  MODE_INDEX;

    /* Additional modification flags */
    uint_8                      FLAGS;

    /* LLWU specific settings */
    uint_8                      PE1;
    uint_8                      PE2;
    uint_8                      PE3;
    uint_8                      PE4;
    uint_8                      ME;

} LPM_CPU_OPERATION_MODE, _PTR_ LPM_CPU_OPERATION_MODE_PTR;
```

5.2 串行驱动程序示例

K60 BSP 在 BSP 初始化时向 LPM 注册 UART 串行驱动程序，轮询和中断两种版本均如此。这允许 LPM 示例在 LPM 模式改变时利用串行驱动程序。此外还提供了如何在 BSP 中设置驱动程序注册和 LPM 回调的示例。

5.2.1 串行工作模式

串行驱动程序在文件：<MQX 安装路径>\mqx\source\bsp\twrk60n512\init_sci.c 中初始化。

该文件包含用于每个串行驱动程序的工作模式数组。在 K60 BSP 中，UART3 和 UART5 默认为启用。以下为用于 UART5 的_bsp_sci5_operation_modes 数组。数组中的每个元素对应于一种 LPM 工作模式。

```
const KUART_OPERATION_MODE_STRUCT _bsp_sci5_operation_modes[LPM_OPERATION_MODES] =
{
  /* LPM_OPERATION_MODE_RUN */
  {
     IO_PERIPHERAL_PIN_MUX_ENABLE | IO_PERIPHERAL_CLOCK_ENABLE | IO_PERIPHERAL_MODULE_ENABLE,
     0,
     0,
     0
  },

  /* LPM_OPERATION_MODE_WAIT */
  {
     IO_PERIPHERAL_PIN_MUX_ENABLE | IO_PERIPHERAL_CLOCK_ENABLE | IO_PERIPHERAL_MODULE_ENABLE,
     0,
     0,
     0
  },

  /* LPM_OPERATION_MODE_SLEEP */
  {
     IO_PERIPHERAL_PIN_MUX_ENABLE | IO_PERIPHERAL_CLOCK_ENABLE | IO_PERIPHERAL_MODULE_ENABLE
     | IO_PERIPHERAL_WAKEUP_ENABLE | IO_PERIPHERAL_WAKEUP_SLEEPONEXIT_DISABLE,
     0,
     0,
     0
  },

  /* LPM_OPERATION_MODE_STOP */
  {
     IO_PERIPHERAL_PIN_MUX_DISABLE | IO_PERIPHERAL_CLOCK_DISABLE,
     0,
     0,
     0
  }
};
```

此数组中的每个元素均为 KUART_OPERATION_MODE_STRUCT 类型，在 serl_kuart.h 中定义如下。这些元素包含当 LPM 修改模式时，串行驱动程序所使用的标志和设定。

```
typedef struct kuart_operation_mode_struct
{
    /* HW/wakeup enable/disable flags */
    uint_8      FLAGS;

    /* Wakeup register bits combination: UART_C2_RWU_MASK,
UART_C1_WAKE_MASK, UART_C1_ILT_MASK, UART_C4_MAEN1_MASK,
UART_C4_MAEN2_MASK */
    uint_8      WAKEUP_BITS;

    /* Wakeup settings of register UART_MA1 */
    uint_8      MA1;

    /* Wakeup settings of register UART_MA2 */
    uint_8      MA2;

} KUART_OPERATION_MODE_STRUCT, _PTR_ KUART_OPERATION_MODE_STRUCT_PTR;
```

5.2.2 串行驱动程序注册

安装时，K60 会轮询串行驱动程序以便向 LPM 注册。在代码中，这是在函数_io_serial_polled_install() in serl_pol.c 中完成的。以下代码用于注册驱动程序。串行驱动程序使用两个回调函数:
_io_serial_polled_clock_configuration_callback()和_io_serial_polled_operation_mode_callback()。这些回调使用 LPM 传递过来的通知结构和_bsp_scix_operation_modes 数组中的设定，针对模式更改来修改 UART 外设。这些回调函数位于 serl_pol_kuart.c 中。

```
#if MQX_ENABLE_LOW_POWER
   if (MQX_OK == result)
   {
      LPM_REGISTRATION_STRUCT registration;
      registration.CLOCK_CONFIGURATION_CALLBACK =
_io_serial_polled_clock_configuration_callback;
      registration.OPERATION_MODE_CALLBACK = _io_serial_polled_operation_mode_callback;
      registration.DEPENDENCY_LEVEL = BSP_LPM_DEPENDENCY_LEVEL_SERIAL_POLLED;
      result = _lpm_register_driver (&registration, dev_ptr,
       &(dev_ptr->LPM_INFO.REGISTRATION_HANDLE));
      if (MQX_OK == result)
      {
         _lwsem_create (&(dev_ptr->LPM_INFO.LOCK), 1);
         dev_ptr->LPM_INFO.FLAGS = 0;
      }
   }
#endif
```

6 结论

飞思卡尔 Kinetis MCU 系列拥有诸多极具吸引力的特性，业界领先的低功耗模式就是其中之一。LPM 驱动程序集成在 MQX 中，允许应用程序轻松利用这些低功耗模式。LPM 将这些低功耗模式的硬件相关设定从应用层代码中抽离，使得软件更易于移植。此外，LPM 集成了其他外设驱动程序，支持在不同模式之间顺畅切换。LPM 是采用飞思卡尔免费 MQX RTOS 的另一个原因。关于飞思卡尔 Kinetis MCU 和 MQX 的更多资源，请访问以下链接:

http://www.freescale.com/zh-Hans/webapp/sps/site/homepage.jsp?code=MQX_HOME

http://www.freescale.com/zh-Hans/webapp/sps/site/homepage.jsp?code=KINETIS

7 附录——LPM 驱动程序 API

_lpm_get_clock_configuration——返回当前时钟配置的标识符。

原型: `BSP_CLOCK_CONFIGURATION _lpm_get_clock_configuration()`
参数：无
返回: • 来自_cm_get_clock_configuration()的结果 • 错误：–1 = 等待信号量失败

_lpm_get_operation_mode——返回当前低功耗模式的标识符。

原型:

下一页继续介绍此表...

`LPM_OPERATION_MODE _lpm_get_operation_mode()`
参数：无
返回： • lpm_state_struct.OPERATION_MODE • 错误：–1 = 等待信号量失败

_lpm_install——将 LPM 安装到 MQX。设置 lpm_state_struct 的 CPU_OPERATION_MODES 和 OPERATION_MODE。

原型：
 	
`_mqx_uint _lpm_install (`
`  const LPM_CPU_OPERATION_MODE _PTR_ cpu_operation_modes,`
`  LPM_OPERATION_MODE default_mode)`

参数： • *cpu_operation_modes* [IN]——可用 CPU 内核低功耗工作模式的规格 • *default_mode* [IN]——默认的低功耗工作模式标识符
返回： • MQX_OK • 错误： MQX_INVALID_PARAMETER = 输入参数无效 MQX_COMPONENT_EXISTS = 已安装 LPM MQX_IO_OPERATION_NOT_AVAILABLE = 创建信号量时出错

_lpm_register_driver——将驱动程序注册到 LPM 中。向 lpm_state_struct.DRIVER_ENTRIES 中添加驱动程序，并根据依赖性级别排序列表。

原型：
 	
`_mqx_uint _lpm_register_driver (`
`  const LPM_REGISTRATION_STRUCT_PTR driver_registration_ptr,`
`  const pointer driver_specific_data_ptr,`
`  _mqx_uint_ptr registration_handle_ptr)`

参数： • *driver_registration_ptr* [IN]——驱动程序低功耗回调规格 • *driver_specific_data_ptr* [IN]——驱动程序特定数据 • *registration_handle_ptr* [OUT]——唯一的驱动程序注册句柄
返回： • MQX_OK • 错误： MQX_INVALID_PARAMETER = 无效指针参数 MQX_IO_OPERATION_NOT_AVAILABLE = 等待信号量失败 MQX_OUT_OF_MEMORY = 分配存储器失败

_lpm_set_clock_configuration——修改 MCU 时钟配置。在修改时钟配置前后向已注册的驱动程序发送通知。

原型：
 	
`_mqx_uint _lpm_set_clock_configuration (`
`  BSP_CLOCK_CONFIGURATION clock_configuration)`

参数：clock_configuration [IN]——时钟配置标识符

下一页继续介绍此表...

返回：
- 来自_cm_set_clock_configuration()的结果
- 错误：
  MQX_INVALID_PARAMETER = 无效时钟配置
  MQX_IO_OPERATION_NOT_AVAILABLE = 等待信号量失败，或驱动程序回调失败

_lpm_set_operation_mode——更改低功耗工作模式。在改变工作模式之前向已注册的驱动程序发送通知。

原型：

```
_mqx_uint _lpm_set_operation_mode (
     LPM_OPERATION_MODE operation_mode)
```

参数：工作模式 [IN]——低功耗工作模式标识符

返回：
- 来自_lpm_set_cpu_operation_mode()的结果
- 错误：
  MQX_INVALID_PARAMETER = 无效工作模式标识符
  MQX_IO_OPERATION_NOT_AVAILABLE = 等待信号量失败，或驱动程序回调失败

_lpm_uninstall——从 MQX 卸载 LPM。取消对 lpm_state_struct 的全局初始化。

原型：

```
_mqx_uint_lpm_uninstall()
```

参数：无

返回：
- MQX_OK
- 错误：MQX_IO_OPERATION_NOT_AVAILABLE = 等待信号量失败

_lpm_unregister_driver——从 LPM 中取消对驱动程序的注册。

原型：

```
_mqx_uint _lpm_unregister_driver (
     _mqx_uint registration_handle)
```

参数：*registration_handle* [IN]——由 LPM 注册函数返回的驱动程序注册句柄

返回：
- MQX_OK
- 错误：
  MQX_INVALID_PARAMETER = 无效注册句柄
  MQX_IO_OPERATION_NOT_AVAILABLE = 等待信号量失败
  MQX_INVALID_HANDLE = 未找到有效的注册记录

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

本文档中的信息仅供系统和软件实施方使用 Freescale 产品。 本文并未明示或者暗示授予利用本文档信息进行设计或者加工集成电路的版权许可。 Freescale 保留对此处任何产品进行更改的权利，恕不另行通知。

Freescale 对其产品在任何特定用途方面的适用性不做任何担保、表示或保证，也不承担因为应用程序或者使用产品或电路所产生的任何责任，明确拒绝承担包括但不局限于后果性的或附带性的损害在内的所有责任。 Freescale 的数据表和/或规格中所提供的“典型”参数在不同应用中可能并且确实不同，实际性能会随时间而有所变化。 所有运行参数，包括“经典值”在内，必须经由客户的技术专家对每个客户的应用程序进行验证。 Freescale 未转让与其专利权及其他权利相关的许可。 Freescale 销售产品时遵循以下网址中包含的标准销售条款和条件：freescale.com/SalesTermsandConditions.

Document Number
Revision 0, 01/2012