Freescale Semiconductor
Application Note

Document Number: AN4447
Rev. 0, 01/2012

Freescale MQX Low-Power Management

by: **Derek Snell**
Field Application Engineer

Contents

1 Introduction

Freescale offers many low-power microcontrollers (MCUs), including the 32-bit Kinetis family. Kinetis offers several different low-power modes of operation, providing very low standby and run current consumption. Freescale also offers MQX™, the full-featured and complimentary Real-Time Operating System (RTOS). Starting with version 3.8, MQX integrates a Low-Power Management (LPM) driver to take advantage of the low-power operating modes in MQX applications.

This application describes the MQX LPM, how it works, and how to use it. Specifically, it references the Kinetis K60 family and the Tower TWR-K60N512 development board and Board Support Package (BSP). However, other MCUs and boards are supported by the LPM; please refer to the latest MQX release notes to see what is supported. This document also refers to the Kinetis low-power modes. Please refer to the Kinetis device reference manual and the Kinetis Quick Reference User Guide (search KQRUG) for more details on these power modes.

2 LPM overview

The LPM is an MQX driver included with specific BSPs. It enables an application to easily change operation modes to take advantage of the MCU's low-power modes. The LPM is configured with operation modes of that BSP, which are mapped to the CPU power modes of the MCU.

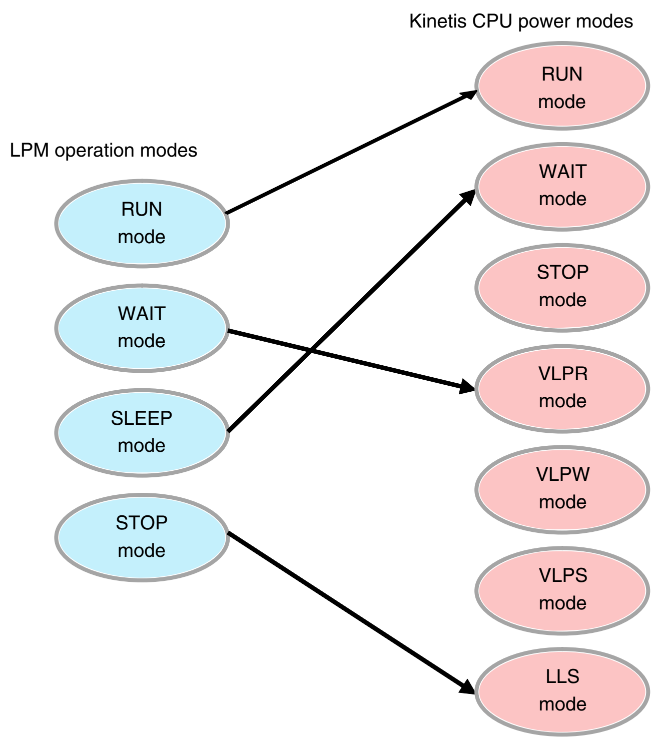

Figure 1. LPM operation modes of K60 BSP mapped to CPU power modes

The LPM also provides a mechanism for other drivers in the BSP to register with the LPM. Once registered, these drivers will be notified by callback functions when the application changes the operating mode or the clock configuration. This way, the drivers can shut down the peripherals to reduce power or modify the peripheral settings to adapt to the changes. For example, if the application reduces the clock frequency to reduce power consumption, the serial UART driver can update the baud rate divider registers to maintain the same baud rate at the reduced clock frequency. This architecture helps abstract the application from the hardware and allows the application to be more portable.

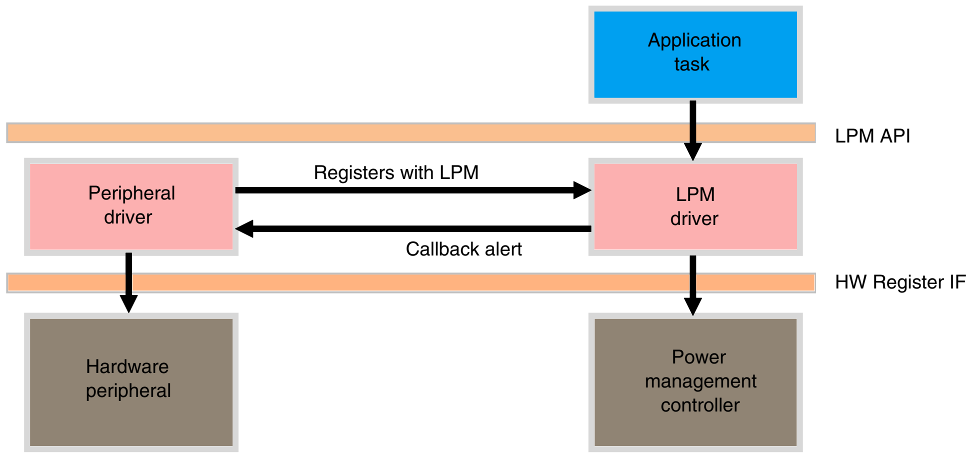

Figure 2. LPM architecture

3 LPM example

MQX includes example projects to evaluate the LPM and to help you understand how to use it. These low-power examples are found at the following path, and include projects for multiple BSPs: <MQX Installation Directory>\mqx\examples \lowpower.

The example sequences through different BSP operating modes. The current consumption can be measured in the different low-power modes on the development board.

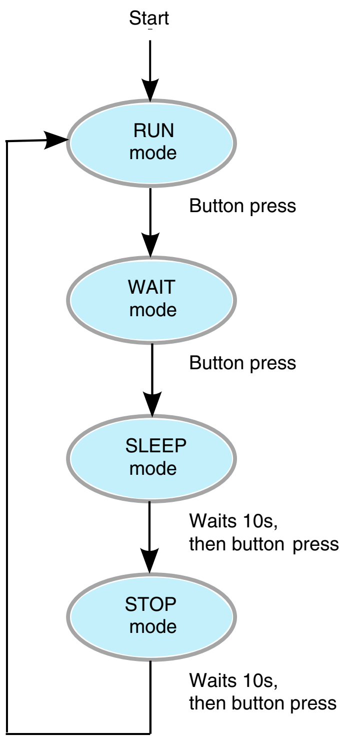

Figure 3. Low-power example flowchart

For example, the K60 example starts in Run mode, running at the full clock speed for the BSP. It uses print statements to the BSP's default UART connected to a terminal to walk the user through the example. It then waits for a button to be pressed, and then moves to the LPM Wait mode, which is mapped to the Kinetis Very Low-Power Run (VLPR) mode. The core clock frequency is reduced to 2MHz to reduce power consumption, but the core is still executing. The LPM updates the serial driver to change the UART baud rate dividers, allowing the baud rate to stay the same.

The terminal displays the wake-up settings of the Kinetis Low-Leakage Wake-Up Unit (LLWU) before the example changes modes. The example changes modes by calling the API _lpm_set_clock_configuration(), and the argument it passes is a number for the clock configuration of the BSP. In this case, that configuration equates to a 2MHz clock. Once the clock frequency is changed and the serial driver is updated, the example calls _lpm_set_operation_mode(). The argument passed here is a number for the operation mode of the LPM. In this case, it changes to the LPM Wait mode.

Again, the example waits for a button press, then it changes the clock back to the default BSP frequency. This time the example changes to the LPM Sleep operation mode, which is mapped to the Kinetis Wait mode. In this mode, the core stops executing and waits for an event to execute again. The event in this case is the Real-Time Clock (RTC) or a UART interrupt. The example uses the RTC driver to set an alarm for 10 seconds, and it remains in Sleep mode for that time. After 10 seconds, it wakes up and prints a message that it will wait for another button press. A UART wake interrupt can also wake the demo up from Sleep mode. To see this, the BSP needs to be changed to use the interrupt version of the serial driver, ittyX.

The final mode in this example is the LPM Stop mode, which is mapped to the Kinetis Low-Leakage Stop (LLS) mode. The UARTs are disabled in this mode to help reduce power consumption. The LLWU is set to allow the RTC to wake up the MCU. Again, the example uses the RTC driver to set a 10 second alarm, and then enters Stop mode. After waking, the UARTs are enabled again, and after a button press, the example starts this sequence over again.

4 LPM driver details

The LPM driver is included in the MQX BSP to allow applications to easily change clock settings and low-power modes. The application just calls _lpm_set_operation_mode() or _lpm_set_clock_configuration() to change the LPM state. The LPM APIs are documented in detail in the Appendix at the end of this appnote. The LPM also allows other drivers to register with the LPM and get notifications when the LPM changes the clock settings or low-power mode. Most of the source code for the LPM can be found in <MQX Installation Directory>\mqx\source\io\lpm.

4.1 CPU power modes

The LPM requires a list of the available low-power modes of the CPU. The BSP maps the LPM operation modes to the CPU low-power modes, which is discussed further in the section "LPM Usage in BSP". For the K60 BSP, the list of CPU power modes is defined below in lpm_kinetis.c:

```
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
    }
};
```

Each power mode in this array is of type LPM_CPU_POWER_MODE, defined below in lpm_kinetis.h. The value in PMCTRL is written to the Kinetis register MC_PMCTRL which controls the power mode of the MCU. The FLAGS field is a group of bits used by the LPM driver when changing CPU power modes.

```
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

The state of the LPM is maintained in a global structure called lpm_state_struct of type LPM_STATE_STRUCT. The structure is declared in lpm_prv.h, and the global variable is declared in lpm.c. This structure keeps track of the LPM operation mode, a table of the available CPU low-power modes, the other drivers that have registered with the LPM, and a semaphore that the LPM uses. This structure is initialized by the _lpm_install() function, and is updated by the other LPM APIs. Below is the structure definition:

```
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 LPM registered drivers

Other drivers in the MQX BSP can register with the LPM and get notified when the LPM changes the clock configuration or the low-power mode. The driver calls _lpm_register_driver() to register with the LPM. One of the arguments in this API is a structure of type LPM_REGISTRATION_STRUCT, which is defined below in lpm.h:

```
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 Callback functions to registered drivers

The LPM can use two different callback functions with the registering driver. The CLOCK_CONFIGURATION_CALLBACK function is called when the LPM executes the _lpm_set_clock_configuration() API. The OPERATION_MODE_CALLBACK function is called with the LPM executes _lpm_set_operation_mode() to change the low-power mode. The LPM calls the clock configuration callback both before and after it changes the clock configuration. The registered drivers can prepare for the change beforehand, and update after the change has been completed. The operation mode callback is only called before the mode is changed. When the registered driver gets its callback, the LPM passes two parameters to the callback: the notification structure and driver-specific data pointer, shown below.

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

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

The notification structure passed to the callbacks is of type LPM_NOTIFICATION_STRUCT and defined in lpm.h below. The NOTIFICATION_TYPE can be the value LPM_NOTIFICATION_TYPE_PRE or LPM_NOTIFICATION_TYPE_POST, both defined in lpm.h. With the notification type passed to the callback, the registered driver knows whether the callback is executing before or after the LPM makes the change. Also, the OPERATION_MODE and CLOCK_CONFIGURATION are passed to tell the registered driver what the new state will be.

```
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 Registered driver dependency

When a driver registers with the LPM using the _lpm_register_driver() API, there is a DEPENDENCY_LEVEL value in the LPM_REGISTRATION_STRUCT passed to the LPM. This level allows the LPM to sort the list of registered drivers by their dependency, or priority. Lower dependency level drivers get PRE notification callbacks before higher levels. When the LPM changes a state, the registered drivers' callbacks are executed in the order of this sorted list.

Here is an example of the driver dependency level. The K60 has an sdcard: driver which uses the peripheral driver esdhc: for the physical interface. If both of these are to be used in an application where the LPM changes the clock configuration, both drivers would register with the LPM, and the esdhc: dependency level would be higher than the sdcard: dependency level. When the application uses the LPM API to change the clock configuration, the callback functions occur in the order shown below in Figure 4.

1. First, the sdcard: driver receives the PRE notification callback because it has a lower dependency level. The intention here is to block communication while the baud rate is changing, and to prevent usage of the low-level driver.
2. Next, the esdhc: driver receives its PRE notification callback, which would wait for the completion of any ongoing transfers.
3. Then the LPM changes the clock configuration of the BSP and starts the POST notification callbacks. The POST callbacks occur in reverse order of the PRE callbacks.
4. The esdhc: driver receives the first POST callback and updates the SDHC peripheral with a new clock frequency.
5. Then the sdcard: driver gets the POST callback to re-enable the driver, and the clock configuration change is complete.

The POST notification order is also used if there is a failure during the clock configuration change. In that case, the LPM rolls back the change to the clock configuration before the failure, using the POST callback order.

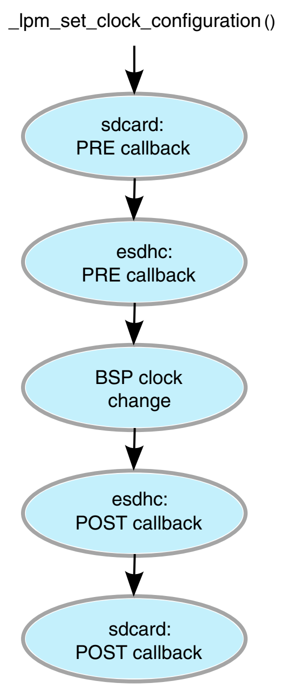

Figure 4. Registered driver dependency clock change example

The LPM maintains the linked list of registered drivers with the DRIVER_ENTRIES pointer in the lpm_state_struct. When a driver registers with the LPM using _lpm_register_driver(), the LPM adds that driver to the appropriate place in the sorted list based on this dependency level.

5 LPM usage in BSP

An MQX BSP is usually modified for the end application. This section discusses how the K60 BSP uses the LPM, and how it would be modified. First, to use the LPM, the macro MQX_ENABLE_LOW_POWER needs to be defined in user_config.h, and the BSP library needs to be rebuilt. Refer to the MQX documentation for further information on this process.

5.1 LPM_CPU_OPERATION_MODES array

The BSP uses a file to configure each driver, and for the LPM, this file is: <MQX Installation Path> \mqx\source\bsp \twrk60n512\init_lpm.c.

This file has an array LPM_CPU_OPERATION_MODES that defines each of the LPM operation modes. See below for the K60:

```
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
    }
};
```

Each entry in this array is of type LPM_CPU_OPERATION_MODE, and that structure is defined below in lpm_kinetis.h. The MODE_INDEX field must be one of the CPU power modes defined in the LPM_CPU_POWER_MODES array. This creates the link that maps the LPM operation mode to a CPU power mode. The PEx and ME fields are the values of the

Kinetis LLWU_PEx and LLWU_ME registers for these operation modes. These settings control the wake-up sources of the LLWU in the low-power mode. The FLAGS field is for additional flags. These BSP settings can be changed to use different CPU power modes and wake-up sources for the application's requirements.

```
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 Serial driver example

The K60 BSP has the UART Serial drivers register with the LPM in the BSP initialization, both the polled and interrupt versions. This allows the LPM example to take advantage of the serial drivers when the LPM modes change. It also provides an example of how to implement driver registration and LPM callbacks in the BSP.

5.2.1 Serial operation modes

The serial drivers are initialized in the file: <MQX Installation Path>\mqx\source\bsp\twrk60n512\init_sci.c

This file contains an array of operation modes for each serial driver. In the K60 BSP, UART3 and UART5 are enabled by default. Below is the _bsp_sci5_operation_modes array for UART5. Each entry in the array corresponds with an LPM operation mode.

```
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
  }
};
```

Each entry in this array is of type KUART_OPERATION_MODE_STRUCT, defined below in serl_kuart.h. These entries contain flags and settings used by the serial driver when the LPM changes modes.

```
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 Serial driver registration

The K60 polled serial driver registers with the LPM when it is installed. In the code, this happens in the function _io_serial_polled_install() in serl_pol.c. Below is the code used to register the driver. The serial driver uses two callback functions: _io_serial_polled_clock_configuration_callback() and _io_serial_polled_operation_mode_callback(). Those callbacks use the notification structure passed from the LPM and the settings in the _bsp_scix_operation_modes array to modify the UART peripheral for the mode change. These callback functions are located in serl_pol_kuart.c.

```
#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 Conclusion

The industry-leading low-power modes in Freescale's Kinetis MCUs are one of many attractive features of this family. The LPM driver integrated in MQX allows applications to easily take advantage of these low-power modes. The LPM abstracts the hardware-related settings of these low-power modes from the application, allowing more portable software. And the LPM integrates with other peripheral drivers allowing smooth transitions between modes. The LPM is another reason to take advantage of Freescale's complimentary MQX RTOS. For further resources on Freescale's Kinetis MCUs and MQX, please visit the links below:

http://www.freescale.com/mqx

http://www.freescale.com/kinetis

7 Appendix—LPM driver APIs

_lpm_get_clock_configuration—returns current clock configuration identifier.

Prototype: `BSP_CLOCK_CONFIGURATION _lpm_get_clock_configuration()`
Parameters: none
Returns: • Result from _cm_get_clock_configuration() • Errors: –1 = failed waiting for semaphore

_lpm_get_operation_mode—returns current low-power mode identifier.

Prototype: `LPM_OPERATION_MODE _lpm_get_operation_mode()`
Parameters: none
Returns: • lpm_state_struct.OPERATION_MODE • Errors: –1 = failed waiting for semaphore

_lpm_install—installs LPM into MQX. Sets CPU_OPERATION_MODES and OPERATION_MODE of lpm_state_struct.

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

Parameters: • *cpu_operation_modes* [IN]—specification of CPU core low-power operation modes available • *default_mode* [IN]—default low-power operation mode identifier

Table continues on the next page...

Returns:

- MQX_OK
- Errors:

    MQX_INVALID_PARAMETER = input parameters are invalid
    MQX_COMPONENT_EXISTS = LPM already installed
    MQX_IO_OPERATION_NOT_AVAILABLE = error in creating semaphore

_lpm_register_driver—registers driver into LPM. Adds driver to lpm_state_struct.DRIVER_ENTRIES, and sorts list by dependency level.

Prototype:

```
_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)
```

Parameters:

- *driver_registration_ptr* [IN]—driver low-power callback specification
- *driver_specific_data_ptr* [IN]—driver-specific data
- *registration_handle_ptr* [OUT]—unique driver registration handle

Returns:

- MQX_OK
- Errors:

    MQX_INVALID_PARAMETER = invalid pointer parameters
    MQX_IO_OPERATION_NOT_AVAILABLE = waiting for semaphore failed
    MQX_OUT_OF_MEMORY = failed to allocate memory

_lpm_set_clock_configuration—changes the MCU clock configuration. Sends notifications to registered drivers before and after changing clock configuration.

Prototype:

```
_mqx_uint _lpm_set_clock_configuration (
     BSP_CLOCK_CONFIGURATION clock_configuration)
```

Parameters: clock_configuration [IN]—clock configuration identifier

Returns:

- Result from _cm_set_clock_configuration()
- Errors:

    MQX_INVALID_PARAMETER = invalid clock configuration
    MQX_IO_OPERATION_NOT_AVAILABLE = failed waiting for semaphore, or driver callback failed

_lpm_set_operation_mode—changes low-power operation mode. Sends notifications to registered drivers before changing operation mode.

Prototype:

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

Parameters: *operation mode* [IN]—low-power operation mode identifier

Table continues on the next page...

Returns: • Results from _lpm_set_cpu_operation_mode() • Errors: MQX_INVALID_PARAMETER = invalid operation mode identifier MQX_IO_OPERATION_NOT_AVAILABLE = failed waiting for semaphore, or driver callback failed

_lpm_uninstall—uninstalls LPM from MQX. De-initializes global lpm_state_struct.

Prototype: `_mqx_uint_lpm_uninstall()`
Parameters: none
Returns: • MQX_OK • Errors: MQX_IO_OPERATION_NOT_AVAILABLE = waiting for semaphore failed

_lpm_unregister_driver—unregisters driver from LPM.

Prototype:
 	
`_mqx_uint _lpm_unregister_driver (`
`     _mqx_uint` *`registration_handle`*`)`

Parameters: *registration_handle* [IN]—driver registration handle returned by LPM registration function
Returns: • MQX_OK • Errors: MQX_INVALID_PARAMETER = invalid registration handle MQX_IO_OPERATION_NOT_AVAILABLE = failed waiting for semaphore MQX_INVALID_HANDLE = no valid registration record found

Document Number: AN4447
Rev. 0, 01/2012