Freescale Semiconductor

Clocks and Low Power modes with KSDK and Processor Expert

About this document

The Kinetis Software Development Kit (KSDK) provides a set of System Services including the next:

- Clock Manager
- Low Power Manager
- Hardware Timer
- Interrupt

This guide is an introduction to the Clock and Low Power Managers and a brief guide to leverage the flexible power and clock configurations in a Kinetis device by using Processor Expert.

Software versions

The example project and contents in this document were developed using the next software versions:

- KSDK v1.3.0
- KDS v3.0.0

The example projects are focused on the use of Processor Expert. In case PE is not desired, there are Low Power example projects in KSDK installation folder:

C:\Freescale\KSDK_1.3.0\examples\<BOARD>\demo_apps\ power_manager_hal_demo

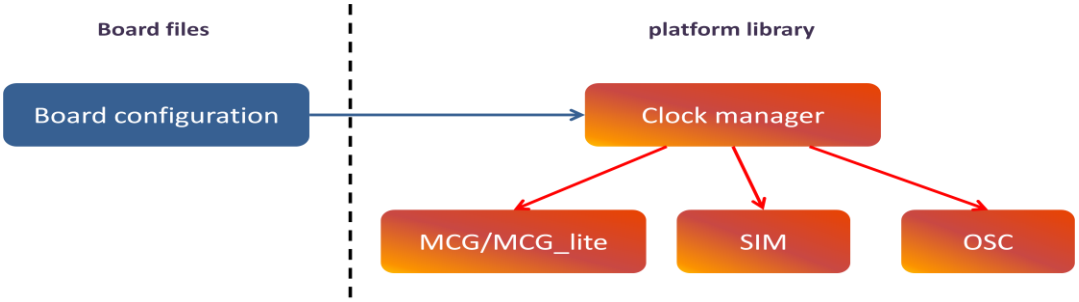
C:\Freescale\KSDK_1.3.0\examples\<BOARD>\demo_apps\ power_manager_rtos_demo

freescale semiconductor

CLOCK MANAGER

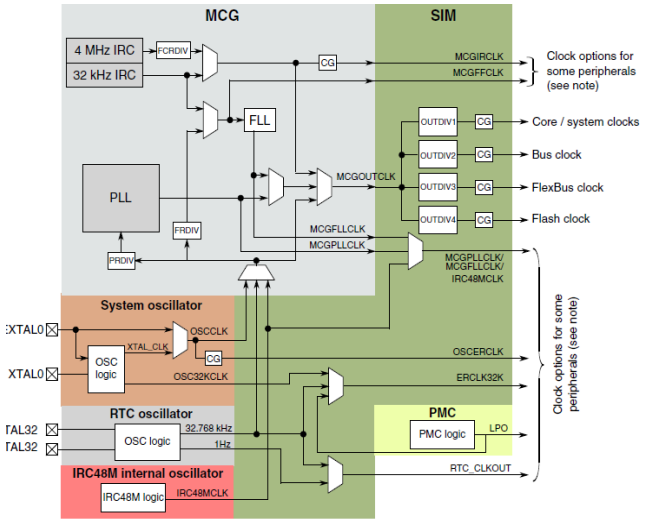
The clock manager in KSDK is designed for:

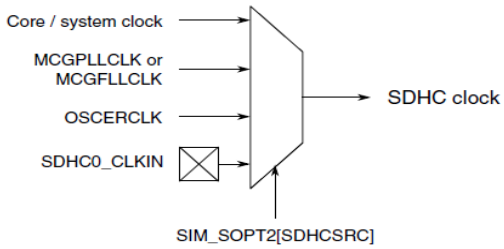
- System layer clock configuration, such as clock mode setting.
- Clock source and divider setting for peripherals.
- Clock frequency getting.
- Notification framework to inform modules when clock mode changes.



Clock source and divider setting

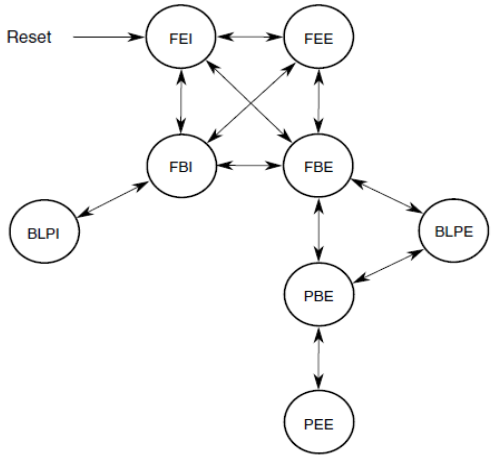
- The clock manager system provides APIs to configure the system clocks and peripheral clocks, as well as APIs to get clock frequency for every node.
- Clock manager does not implement clock tree auto configuration. For example, if SDHC selects OSCERCLK, application should enable OSCERCLK first manually.





Clock mode switch

- MCG mode switch is a critical and difficult part when developing a custom board, because of two main reasons:
  1) A proper path must be followed to reach a target mode (e.g. FEI -> FBE -> PBE -> PEE)
  2) Provide valid parameters (e.g. FLL reference clock frequency must be in the range of 31.25 kHz – 39.0625 kHz)



HAL:

- Provides APIs to set mode from available source mode
  For example ***CLOCK_HAL_SetFeiMode()*** sets MCG in FEI mode when the source mode is FBI/FBE/FEE.
- The APIs validate the parameters and ensure correct switching.

PD layer (Peripheral Driver) or Clock System:

- The function ***CLOCK_SYS_SetMcgMode()*** chooses a proper path to switch clock mode.
- ***CLOCK_SYS_SetConfiguration()*** changes clock configurations without notifications.
- ***CLOCK_SYS_UpdateConfiguration()*** changes clock configurations and uses the notification framework, sending BEFORE, AFTER and RECOVERY messages to the static callbacks.

Board files:

- Board files define the default base clock configurations on board design. These files are modified according to application needs or the custom board clocking options.

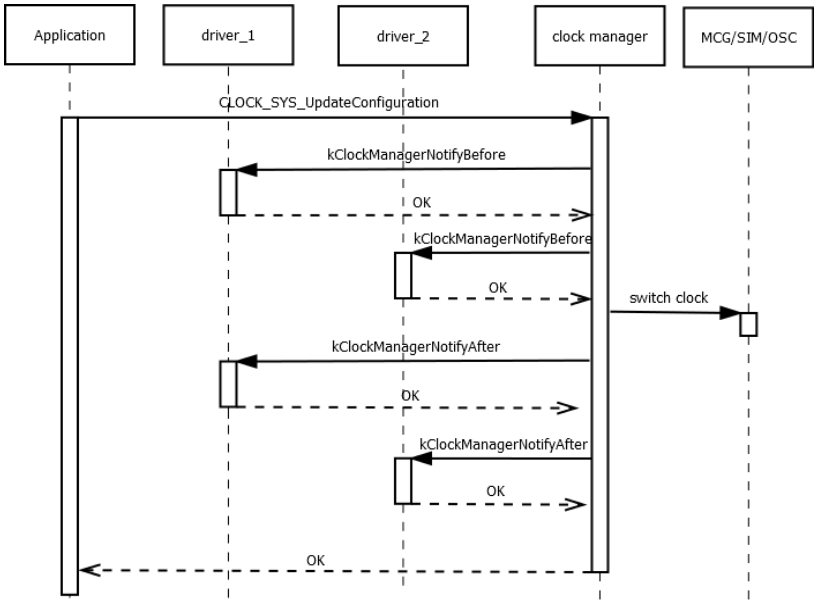
Clock manager notification framework

The notification framework in the KSDK Clock System has the next features:

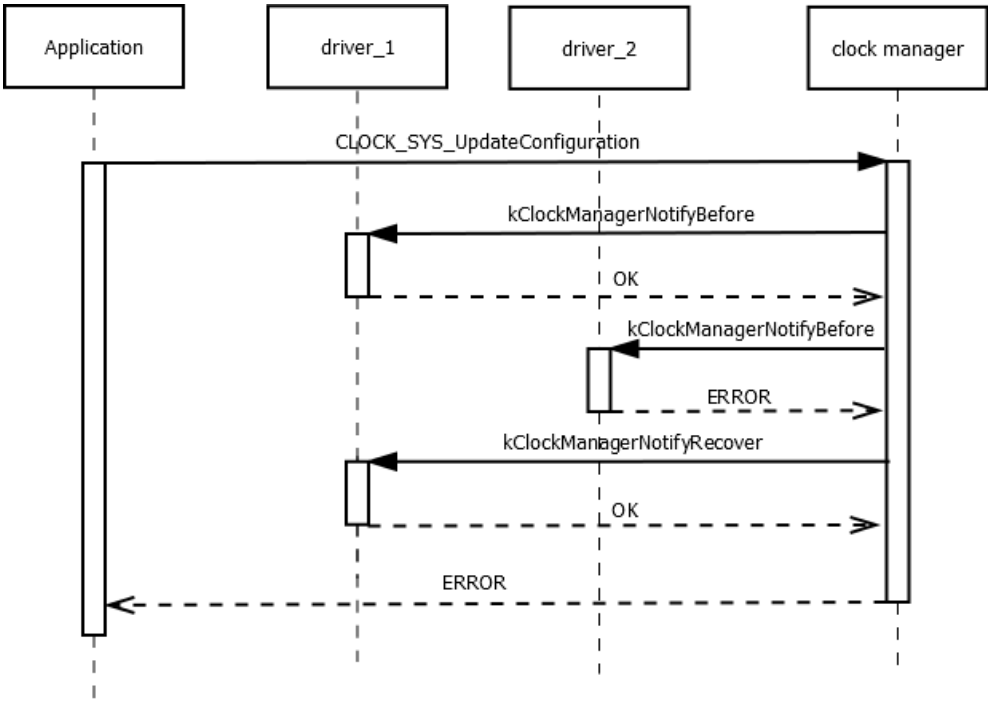
- Applications can register callbacks. When system clock is changed, clock manager sends notifications to the callbacks.
- Static clock configuration and callbacks registration.
- Support forceful clock change and graceful clock change.
- Support clock recovery when some module is not ready for clock change.

The basic actions taken by the clock system are:

- ➔ Before clock switching, clock manager sends the “BEFORE” message.
- ➔ After clock switching, clock manager sends the “AFTER” message.
- ➔ If some module is busy and could not change clock, the clock manager sends the “RECOVER” message to those modules which received the “BEFORE” message.
- ➔ The callback function should handle these messages.



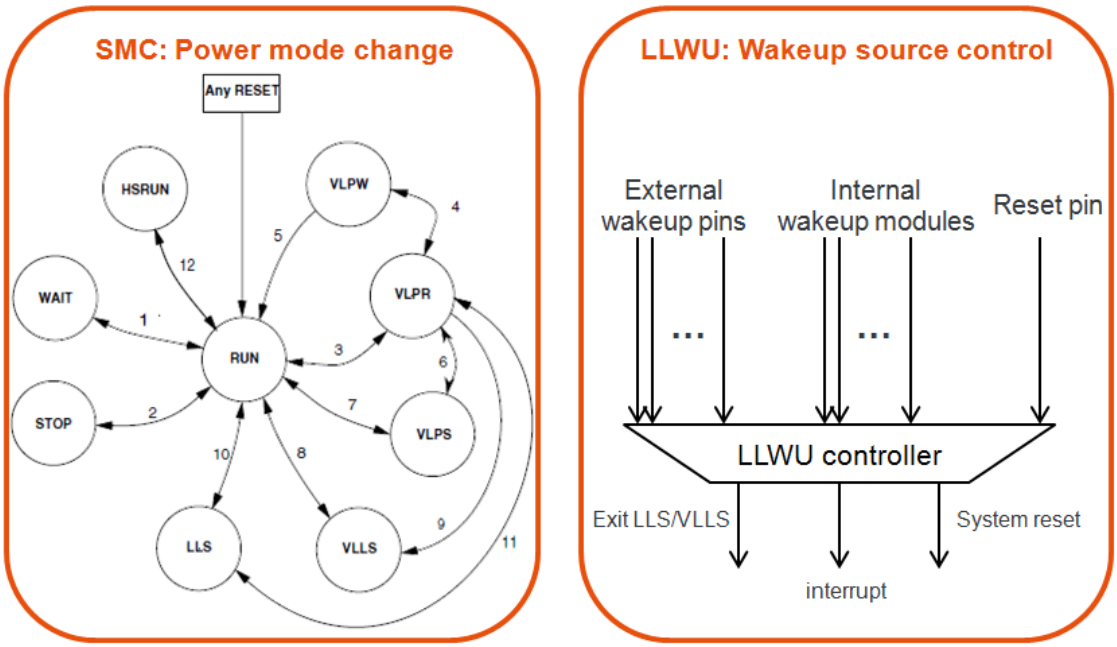
Successful clock switch (all modules return OK)



Clock switch unsuccessful (driver_2 not ready)

POWER MANAGER

- Power Manger system is based on Kinetis SMC (System Mode Controller) and LLWU (Low-Leakage Wake-up Unit)
- The power manager sets the system power mode and wakeup source.
- Provides a notification framework to inform drivers about power mode change.



Power Manager Notification framework

Similar to the Clock Manager the Power Manager has the next features:

- Static Power Mode registration
- Static callback registration
- Supports forceful or graceful power mode changes.
- Supports recovery when some module is not ready for power change.

CLOCKS AND POWER MODES WITH PROCESSOR EXPERT

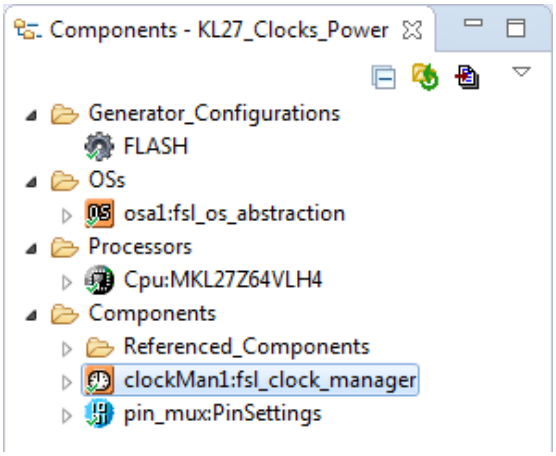
This chapter describes the use of different clock and power configurations with Processor Expert and KSDK and explains the source files requiring user code according to the application.

Clock configurations

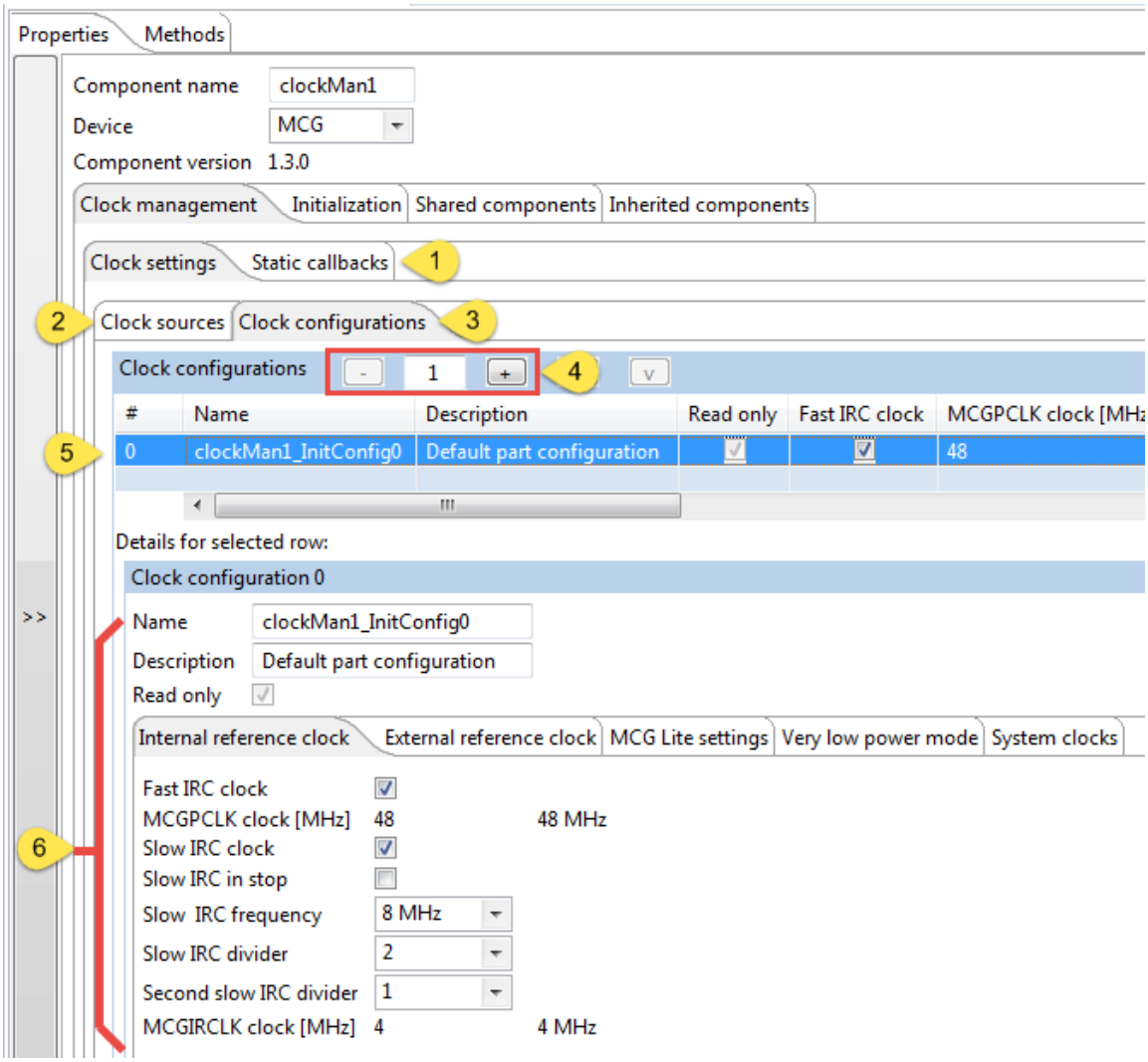
NOTE

> The New Project Wizard in KDS has the option to select between a development board and a target MCU. When a board is selected, some component settings are preconfigured (e.g. pin names, crystal, different clock configurations). For a custom board it is recommended to select the MCU and configure components accordingly.

By default a KSDK project in Kinetis Design Studio with Processor Expert enabled has the **fsl_clock_manager** component already included:



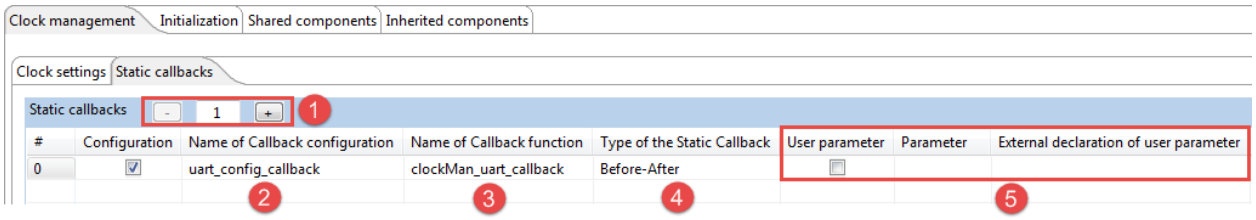
In the component inspector we will find the **Clock configurations** tab:



1- **Static callbacks:** This tab is used to create and configure as many callbacks as required. This is further explained in the next section of this document.
2- **Clock sources:** This tab is used to configure the external clock sources, crystals and/or reference clocks.
3- **Clock configurations:** This is the main tab. All the system clocking options are configured here. Depending on the application requirements, the user can have more than 1 configuration with different clock frequencies or enabling/disabling module clocks in low power modes as required.
4- **Configuration controls:** Used to increase or decrease the number of clock configurations.
5- **Configuration summary:** Shows an overview of the selected clock configuration settings.
6- **Configuration options:** The actual user configurable settings for the selected clock configuration (MCG settings, core/bus clock frequencies, enabling/disabling clocks in Low Power modes, etc).

Static clock system callbacks

The **Static callbacks** tab is used to manage all the required callbacks for the application. The example below shows a single callback used to reconfigure UART settings in the event of a clock configuration change.



1- **Callback controls:** Used to increase/decrease the number of callbacks.
2- **Name of Callback configuration:** A name used to identify different callback configurations.
3- **Name of Callback function:** This is the name of the callback function called by the clock system.
4- **Type of Static Callback:** User can select between having the function called BEFORE, AFTER or BEFORE & AFTER clock changes.
5- **User parameter settings:** The callback can optionally receive a pointer to user data as parameter.

For any use of the function ***CLOCK_SYS_UpdateConfiguration()***, the user callbacks will be executed according to the setting of **Type of Static Callback** (before clock change, after clock change or both cases). Each callback receives a parameter of type **clock_notify_struct_t**, defined as below:

```
/*! @brief Clock notification structure passed to clock callback function. */
typedef struct ClockNotifyStruct
{
    uint8_t targetClockConfigIndex;    /*!< Target clock configuration index. */
    clock_manager_policy_t policy;     /*!< Clock transition policy.          */
    clock_manager_notify_t notifyType; /*!< Clock notification type.          */
} clock_notify_struct_t;
```

targetClockConfigIndex: The index of the predefined clock configuration which is being set.

policy: Can be either **kClockManagerPolicyAgreement** or **kClockManagerPolicyForcible**.

notifyType: Indicates the callback what event is happening: **kClockManagerNotifyRecover**, **kClockManagerNotifyBefore** or **kClockManagerNotifyAfter**.

In the previous example the callback configuration has the name “uart_config_callback” for a simple application in which only the UART module would be affected by a clock change, but the flexible system allows the next use cases:

A) Updating all involved peripherals, saving application data, etc in a single callback function.
B) Having a separate callback for each peripheral or for different application stages (e.g. for a state-machine).
C) Combining (A) and (B).

Writing code in clock system callbacks

After generating code, the file **Events.c** will contain placeholders/skeletons for the callback functions. User has to fill up the callback functions according to the application:

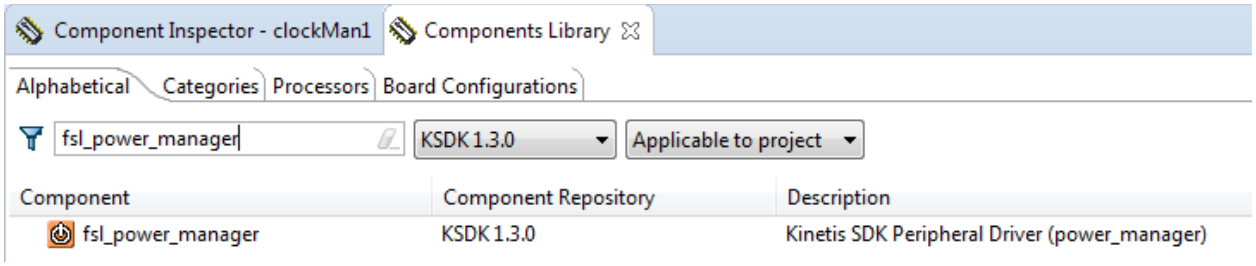
```
clock_manager_error_code_t clockMan_uart_callback(clock_notify_struct_t * notify, void * callbackData)
{
    clock_manager_error_code_t result = kClockManagerSuccess;

    /* Write user code here */

    return result;
}
```

Below an example of how a callback may look like:

```
clock_manager_error_code_t clockMan_uart_callback(clock_notify_struct_t * notify, void * callbackData)
{
    clock_manager_error_code_t result = kClockManagerSuccess;
    switch (notify->notifyType)
    {
        case kClockManagerNotifyBefore:     // Received "pre" message
            DbgConsole_DeInit();
            break;
        case kClockManagerNotifyRecover: // Received "recover" message
            break;
        case kClockManagerNotifyAfter:    // Received "post" message
            /*Initialize UART after clock configuration change*/
            if (CLOCK_VLPR == CLOCK_SYS_GetCurrentConfiguration())
            {
                CLOCK_SYS_SetLpuartSrc(BOARD_DEBUG_UART_INSTANCE, kClockLpuartSrcMcgIrClk);
            }
            else
            {
                CLOCK_SYS_SetLpuartSrc(BOARD_DEBUG_UART_INSTANCE, kClockLpuartSrcIrc48M);
            }
            DbgConsole_Init(BOARD_DEBUG_UART_INSTANCE, DEBUG_UART_BAUD, DEBUG_UART_TYPE);
             break;
        default:
            result = kClockManagerError;
            break;
    }
    return result;
}
```

Power configurations

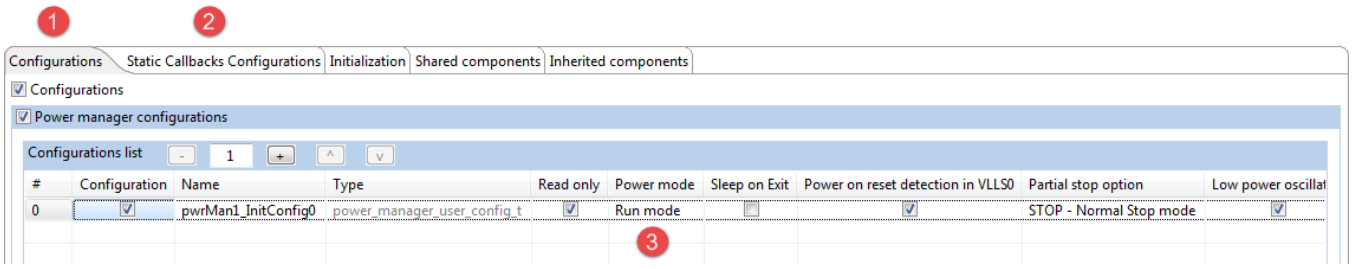
To manage different power configurations, it is required to add the **fsl_power_manager** component to the project:



NOTE

> When adding the Power Manager component, Processor Expert asks for a **fsl_gpio** component to be added as well. This is required for the case of configuring one or more pins as external Wake-up sources.

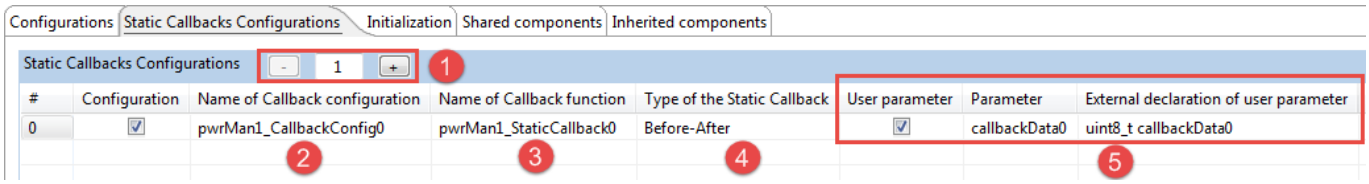
The **fsl_power_manager** component settings look as next:



1- **Configurations:** The main tab in the Power Manager component. Here the user creates the different power configuration options for the application.
2- **Static Callbacks Configurations:** This tab is used to manage callbacks for the Power Manager system. Further explained next.
3- **Power mode:** The most important setting. This is the actual Kinetis power mode to be entered when changing to the corresponding power configuration (e.g. RUN, VLPR, Wait, VLPW, etc).

Static power system callbacks

The **Static Callbacks Configurations** tab is used to manage Power System callbacks, creating all of the callbacks required by the application. Picture below shows a single callback configuration set to be called BEFORE and AFTER the power mode changes:



1- **Callback controls:** Buttons used to increase or decrease the number of callback configurations.
2- **Name of Callback configuration:** An identifier name for each callback configuration.
3- **Name of Callback function:** The name of the function that will be called from the Power Manager system.
4- **Type of the Static Callback:** This setting defines when each callback function is invoked (BEFORE, AFTER, or BEFORE & AFTER).
5- **User parameter settings:** The power manager callback functions can optionally receive some pointer to some application data.

In a similar way to the clock manager, when calling the function ***POWER_SYS_SetMode()*** the power manager notification framework calls the callbacks according to its type (BEFORE changing power mode, AFTER changing the mode or in both events). The callback function receives a parameter of type **power_manager_notify_struct_t**, conformed as next:

```
/*! @brief Power notification structure passed to the registered callback function. */
typedef struct _power_notify_struct
{
    uint8_t targetPowerConfigIndex;
    power_manager_user_config_t const *targetPowerConfigPtr;
    power_manager_policy_t policy;
    power_manager_notify_t notifyType;
} power_manager_notify_struct_t;
```

targetPowerConfigIndex: The number of one the predefined power configuration which is being set.

targetPowerConfigPtr: Pointer to the target configuration properties. This might be required for the callback to take action according to the new power configuration settings.

Policy: Either **kPowerManagerPolicyAgreement** or **kPowerManagerPolicyForcible**.

notifyType: Indicates the callback the stage of the power mode switch: **kPowerManagerNotifyRecover**, **kPowerManagerNotifyBefore** or **kPowerManagerNotifyAfter**.

The Power Manager callback configurations provide flexibility to implement 3 general use cases:

A) Have a single callback to adapt all peripherals or take required application preventive actions.
B) Having multiple callbacks.
C) Combining (A) and (B)

Writing code for power system callbacks

Processor Expert will generate the callback function's skeleton so the user can add custom code.

```
power_manager_error_code_t pwrMan1_StaticCallback0(power_manager_notify_struct_t * notify, power_manager_callback_data_t * dataPtr)
{
  power_manager_error_code_t status = kPowerManagerSuccess;
  /* Write your code here ... */

  return status;
}
```

Below picture shows an example of how the callback might look after adding custom application code:

```
power_manager_error_code_t pwrMan1_StaticCallback0(power_manager_notify_struct_t * notify, power_manager_callback_data_t * dataPtr)
{
  power_manager_error_code_t status = kPowerManagerSuccess;

  volatile bool isLastByteTranmistComplete = false;

  switch (notify->notifyType)
  {
      case kClockManagerNotifyBefore:     // Received "pre" message
          do
          {
              isLastByteTranmistComplete = LPUART_HAL_GetStatusFlag(LPUART0,kLpuartTxComplete);
          } while (!isLastByteTranmistComplete);

          if(*(uint8_t *)dataPtr == WAIT_N_STOP_MODES)
          { //Check if WAIT or STOP modes are requested to disable Systick interrupts
             // INT_SYS_DisableIRQ(SysTick_IRQn);
              SYSTICK_DISABLE();
          }
          if(*(uint8_t *)dataPtr == VLLSX_MODES)
          { //if the CPU is going to enter any VLLSx mode we start the LPTMR
                POWER_SYS_SetWakeupModule(kLlwuWakeupModule0,true);
                LPTMR_DRV_SetTimerPeriodUs(lpTmr1_IDX,SEC(5));  //Set time in seconds
                LPTMR_DRV_Start(lpTmr1_IDX);                    //Start the counter
          }
          break;
      case kClockManagerNotifyRecover: // Received "recover" message
      case kClockManagerNotifyAfter:    // Received "post" message
          if(*(uint8_t *)dataPtr == WAIT_N_STOP_MODES)
          { //Check if WAIT or STOP modes are requested to disable Systick interrupts
              SYSTICK_ENABLE();
          }
          break;
      default:
          status = kClockManagerError;
          break;
  }
  return status;
}
```

CONCLUSION

This document showed an introduction to KSDK clock and power management systems. It provided as well an explanation of how both systems are integrated within the Processor Expert framework, so it is easier for the user to have flexible clock and power configurations leveraging Kinetis MCUs features.

REFERENCES

1- KSDK webpage: www.nxp.com/ksdk
2- KSDK documents: C:\Freescale\KSDK_1.3.0\doc
3- KSDK demo and example projects: C:\Freescale\KSDK_1.3.0\examples

4- How to create a baremetal + PEx KSDK project in KDS:
https://community.freescale.com/docs/DOC-104318