NXP Semiconductors Document Number:
Application Note Rev. 0.1, 09/2018

JN516x to KW41Z ZigBee Migration Guide

1. Introduction

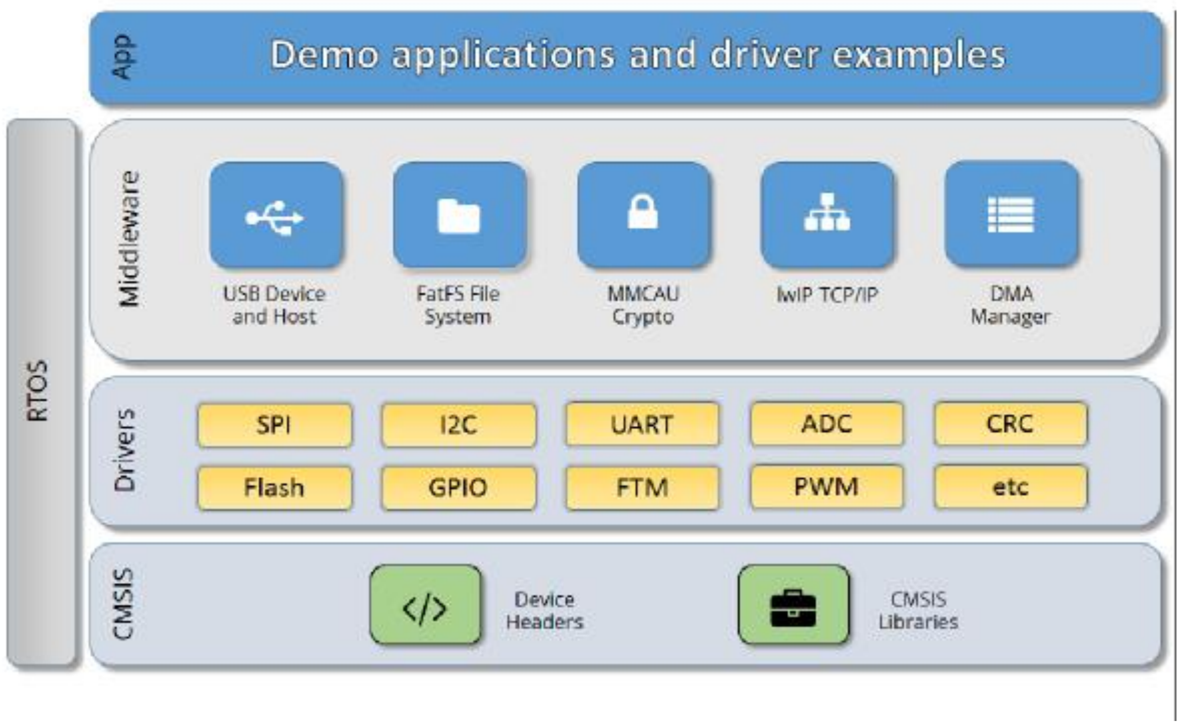
This Application Note provides guidance on migrating ZigBee 3.0 Base device application designed for the NXP JN516x wireless microcontrollers to the KW41Z. In particular, the KW41Z supports FreeRTOS and JN5169 is bare metal implementation. So, guidance is provided on migration from bare metal architecture to MCUXpresso SDK, which supports FreeRTOS as RTOS.

Contents

2. SDK Structure

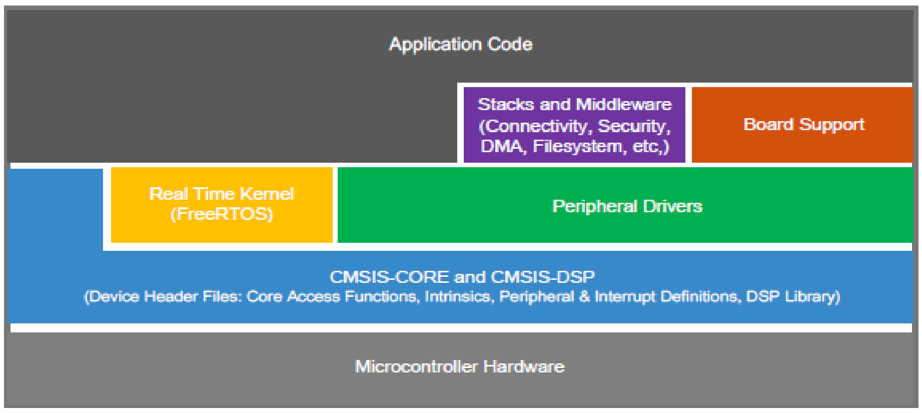
2.1. Overview

The MCUXpresso SDK architecture consists of five key components listed below:
1. The ARM Cortex Microcontroller Software Interface Standard (CMSIS) CORE compliance device specific header files, SOC Header, and CMSIS math/DSP libraries.
2. Peripheral Drivers
3. Real-time Operating Systems (RTOS)
4. Stacks and Middleware that integrate with the MCUXpresso SDK
5. Demo Applications based on the MCUXpresso SDK



2.2. High Level view of SDK

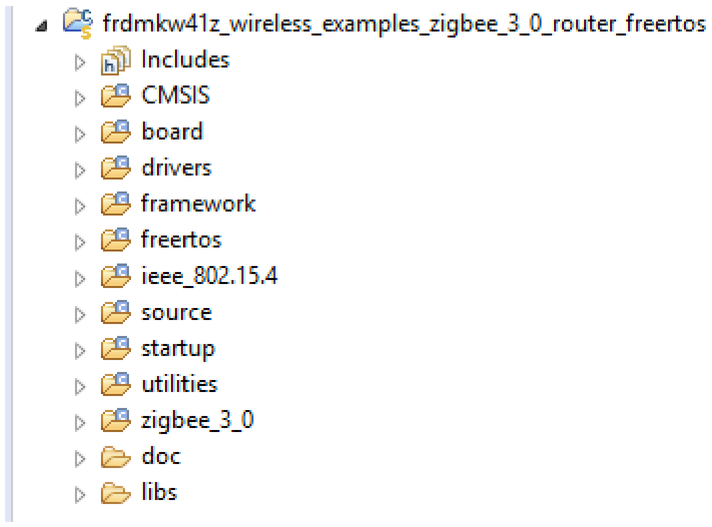
The MCUXpresso Software Development Kit (SDK) provides comprehensive software support for Kinetics and LPC Microcontrollers. The MCUXpresso SDK includes a flexible set of peripheral drivers designed to speed up and simplify development of embedded applications. Along with the peripheral drivers, the MCUXpresso SDK provides an extensive and rich set of example applications covering everything from basic peripheral use case examples to full demo applications. The MCUXpresso SDK contains FreeRTOS, a USB host and device stack, and various other


Application Code
Stacks and Middleware (Connectivity, Security, DMA, Filesystem, etc,)
Board Support
Real Time Kernel (FreeRTOS)
Peripheral Drivers
CMSIS-CORE and CMSIS-DSP
(Device Header Files: Core Access Functions, Intrinsics, Peripheral & Interrupt Definitions, DSP Library)
Microcontroller Hardware

middleware to support rapid development. For supported toolchain versions, see the MCUXpresso SDK Release Notes (document MCUXSDKRN). For the latest version of this and other MCUXpresso SDK documents, see the MCUXpresso SDK homepage, MCUXpresso-SDK: Software Development Kit.

2.3. Example folder structure

In MCUXpresso SDK based application, all files in the application folder are specific to that example, so it's very easy to start developing a custom application based on a project provided in the MCUXpresso SDK. The below figure illustrates the application folder structure.



```
frdmkw41z_wireless_examples_zigbee_3_0_router_freertos
    Includes
    CMSIS
    board
    drivers
    framework
    freertos
    ieee_802.15.4
    source
    startup
    utilities
    zigbee_3_0
    doc
    libs
```

The above application consists of below listed components:

MCU header files
Each supported MCU device in the MCUXpresso SDK has an overall System-on Chip (SoC) memory-mapped header file, which contains the memory map and register base address for each peripheral and the IRQ vector table with associated vector numbers.

CMSIS Support
The MCUXpresso SDK also includes common CMSIS header files for the ARM Cortex-M core and the math and DSP libraries.

MCUXpresso SDK Peripheral Drivers
The MCUXpresso SDK peripheral drivers mainly consist of low-level functional APIs for the MCU product family on-chip peripherals and also of high-level transactional APIs for some bus drivers to quickly enable the peripherals and perform transfers. Low-level functional APIs provide common peripheral functionality, abstracting the hardware peripheral register accesses into a set of stateless basic functional operations. Transactional APIs provide a quick method for customers to utilize higher-level functionality of the peripherals.

Framework Services
All framework services such as memory management, timer manager, serial manager, messaging interact with the operating system through the OS Abstraction Layer. The role of this layer is to offer an "OS-agnostic" separation between the operating system and the upper layers.

For detail understanding of MCUXPresso SDK, refer to the MCUXpresso SDK API Reference Manual_MKW41Z4.pdf & Getting Started with MCUXpresso SDK.pdf located inside the MCUXpresso SDK at <Path to SDK>\docs. You can download the latest MCUXpresso SDK from MCUXpresso SDK.

3. JN5169 & KW41Z devices default hardware & Framework changes

3.1. Overview

As KW41Z uses MCUXpresso based ZigBee3.0 SDK, many of the functionalities which were at application layer in JN5169 are now added inside the MCUXpresso framework. Also, the hardware abstraction layer for both would be different so application should consider the changes related to it in migration of the application.

3.2. Drivers

The JN5169 SDK includes “libAppApi_JN5169”, “libAppQueueApi_JN516x”, “libDBG_JN516x” etc. as the hardware abstraction layer for all functionalities such as watch dog timers, exceptions, clock configurations, serial communications etc.

The MCUXpresso architecture used by KW41Z is organized as described in SDK Structure. It provides drivers as hardware abstraction layer for all the modules. The drivers for all the interfaces are located at “<Application>\drivers” inside the MCUXpresso SDK. The below table provides the mapping list of the hardware APIs of JN516x development board that needs to be replaced with the FRDM-KW41Z APIs.

3.2.1. Hardware API Mapping

Module Name	JN5169 Hardware API	FRDM-KW41Z API	Remarks
Watch Dog	vAHI_WatchdogRestart bAHI_WatchdogResetEvent vAHI_WatchdogStop	COP_Refresh RCM_GetPreviousResetSources COP_Disable COP_GetDefaultConfig COP_Init	
Stack overflow	vAHI_SetStackOverflow	vApplicationStackOverflowHook	
Clock config	bAHI_GetClkSource bAHI_SetClockRate	CLOCK_EnableClock	- MCU clock is initialized inside hardware_init() in KW41Z SDK
DBG	DBG_vUartInit vAHI_UartSetBaudDivisor	Serial_InitInterface Serial_SetBaudRate	
Interrupts	TARGET_INITIALISE(); /* clear interrupt priority level */	-> From NVIC - NVIC_EnableIRQ	- Portmacros in assembly for

	SET_IPL(0);
	portENABLE_INTERRUPTS();
		- NVIC_DisableIRQ
		- NVIC_SetPriority
		- TSI_DisableInterrupts
			JN5169 to core_cm0.h in KW41Z

Software Reset	vAHI_SwReset	NVIC_SystemReset ResetMCU	
UART	DBG_vUartInit vAHI_UartSetBaudDivisor	Serial_InitInterface Serial_SetBaudRate	
GPIO/DIO Pin configurations	vAHI_DioSetDirection	GPIO_PinInit	Configure pin as Input/output
Configure pin interrupt	vAHI_DioInterruptEnable vAHI_DioInterruptEdge	PORT_SetPinInterruptConfig	Enables/disables interrupt on the pin, generates interrupt on a rising or falling edge

Table 1. Hardware API Mapping

For more information about GPIO configurations, please refer section 14.5 from MCUXpresso SDK API Reference Manual_MKW41Z4.pdf located inside the MCUXpresso SDK at <SDK>\docs.

3.2.2. Shell Abstraction Layer for serial communication on FRDM-KW41Z

The JN5169 SDK includes “libAppApi_JN5169” as the hardware abstraction layer for serial communication. The MCUXpresso architecture used by FRDM-KW41Z is designed as described in SDK Structure. It provides various interfaces for serial communication such as I2C, SPI and UART. The drivers for all the interfaces are located at **“<Application>\drivers”**. On top of the drivers, there is a new abstraction layer called as SerialManager located at **“<Application>\framework\SerialManager”**. This layer is common for all the hardware supported by MCUXpresso SDK Architecture. On top of the Serial Manager the framework also supports one more layer called Shell located at **“<Application>\framework\Shell”** which uses Serial Manager for communication. This layer provides a ready to use framework to add support of different commands on the shell required by the application. Application can use the shell for serial communication using commands such as form, steer, find, toggle, factory reset etc.

The application can take reference of “app_zb_shell.c” from **“frdmkw41z_wireless_examples_hybrid_ble_ZigBee_coordinator_freertos”** application.

3.2.3. Debug console functionality for printing debug messages

The JN5169 SDK includes “libDBG_JN516x” as the debug module for printing debug messages on serial console. The MCUXpresso SDK provides debug console module as hardware abstraction layer for printing debug messages located at “Application\utilities\fsl_debug_console.c”. The application needs to initialize debug console inside main_task function as below:

```
DbgConsole_Init(LPUART0_BASE, APP_SERIAL_INTERFACE_SPEED,
DEBUG_CONSOLE_DEVICE_TYPE_LPUART,
BOARD_GetLpuartClock(APP_SERIAL_INTERFACE_INSTANCE));
```

The debug messages are printed using debug console module which is defined inside “fsl_debug_console.h” file. So, application needs to include “fsl_debug_console.h" header file and modify DBG_vPrintf() function inside the file dbg.h as shown below:

```
#define DBG_vPrintf(STREAM, FORMAT, ARGS...)\

   do {                                    \

      if(STREAM)                                \

         PRINTF((FORMAT), ## ARGS);                 \

          }while(0)

#endif
```

3.3. Porting configurations to header files

ZigBee 3.0 device configuration:

The JN5169 SDK provides ZPSConfig file (ZPSConfig diagram) to the application for network and device configuration. This configuration file internally generates some of the source and header files (zps_gen.c, pdum_gen.c etc.). The MCUXpresso SDK architecture does not provide such configuration mechanism, so all these parameters can be configured using the header files (app_zcl_cfg.h, app_zps_cfg.h, app_zcl_globals.c, app_pdum_cfg.h etc.). For example, the type of security for the ZigBee network can be configured by the macro ZPS_ZDO_NWK_KEY_STATE inside the file app_zps_cfg.h. The application can change the security type to no network security as shown below:

```
#define ZPS_ZDO_NWK_KEY_STATE     ZPS_ZDO_NO_NETWORK_KEY
```

For more information about how to configure ZigBee 3.0 device on KW41Z, please refer to Chapter 12. Configuring ZPS inside ZigBee 3.0 Stack User Guide.pdf located inside the MCUXpresso SDK at <Path to SDK>\docs\wireless\Zigbee.

The PDUM module which is a part of pdum_gen.c is supported in MCUXpresso SDK and is located at “ZigBee_3_0\core\ZigBee_common\source\pdum_globals.c”.

The PDUM such as block size and number of blocks can be defined in app_pdum_cfg.h as shown below:

```
#define PdumsDetails_c \

    _pdum_handler_name_ (pdum_apduZDP)  _pdum_block_size_ (100) _pdum_queue_size_ (3)  _eol_ \

   _pdum_handler_name_ (pdum_apduZCL)  _pdum_block_size_ (100) _pdum_queue_size_ (10)  _eol_
```

ZigBee stack and application configuration:

In JN5169, configuration of ZigBee stack & heap are done using Makefile. These configurations can be done using header files on KW41Z. For example, the stack & heap size can be configured inside the header file as shown below:

```
#define  gTotalHeapSize_c           11000
#define gMainThreadStackSize_c  4500
```

MCUXpresso framework provides some modules such as MemManager, NVM, OSAbstraction, TimersManager etc., which can be reconfigured at the application layer as per the requirement. For example, NVM Table size can be configured inside the header file as shown below:

```
/* NVM Table Size */
#ifndef gNvTableEntriesCountMax_c
  #define gNvTableEntriesCountMax_c      20
#endif
```

For more examples please refer to app_config.h & app_framework_config.h of
"frdmkw41z_wireless_examples_ZigBee_3_0_dimmable_light_freertos" application.

FreeRTOS configuration:

FreeRTOS is configured through a header file called FreeRTOSConfig.h. The FreeRTOS configurations such as the scheduling algorithm to be used (non pre-emptive or pre-emptive), maximum priority of task, mutex & semaphores usage etc. are defined inside this header file. For example, the pre-emption policy can be set as shown below:

```
#define configUSE_PREEMPTION       1
```

For more FreeRTOS configurations please refer to "FreeRTOSConfig.h" file of "frdmkw41z_wireless_examples_ZigBee_3_0_dimmable_light_freertos" application.

3.4. API Changes for standard libraries

The MCUXpresso SDK provides the wrapper to use standard library APIs. The below table provides a mapping list of the standard library APIs.

API used on JN5169	API for MCUXpresso SDK
memset	FLib_MemSet
memcpy	FLib_MemCpy

Table 2. Standard Library API Mapping

3.5. Storing Data to NVM

The Persistent Data Manager (PDM) module handles the storage of stack context data and application data in Non-Volatile Memory (NVM). For the JN516x devices, this memory is internal EEPROM, while for KW41Z it is Flash.

In JN516x, the PDM module is a component of the SDK and consists of EEPROM read/write APIs. The KW41Z uses connectivity framework which provides Non-Volatile Storage Module that holds a pointer to a table where the upper layers register information about data that must be saved and restored by the storage system. The table contains one or more table entries that can be stored in RAM or in FLASH memory (default). A table entry contains a generic pointer to a contiguous RAM data structure, how many elements the structure contains, the size of a single element, a table entry ID, and an entry type. The stack context data entries are created by the PDM module added as a part of ZigBee3.0 SDK. The application needs to create data entry to the NVM Table for the application data to be stored in Flash. The NVM Table entry can be created by calling the below macro:

```
NVM_RegisterDataSet(pData, elementsCount, elementSize, dataEntryID, dataEntryType)
```

Where:

- **pData** is a pointer to the RAM memory location where the dataset elements are stored
- **elementsCount** represents how many elements the dataset has
- **elementSize** is the size of a single element
- **dataEntryID** is a 16-bit unique ID of the dataset
- **dataEntryType** is a 16-bit value representing the type of entry (mirrored/unmirrored/unmirrored auto restore)

For example, the start-up flow for the device is decided according to the state of the device. It can be saved in the non-volatile memory as below:

Register to PDM/NVM eNodeState:

```
NVM_RegisterDataSet(&sNodeState, 1, sizeof(tsNodeState),
PDM_ID_APP_COORD, gNVM_MirroredInRam_c);
```

Where:

- **sNodeState** is a variable of structure tsNodeState
- **tsNodeState** is a structure contains the enumerations used to specify a device state (E_STARTUP,E_RUNNING)

This call of the above macro will create a table entry that will be placed in the FLASH memory at link time.

When the data pointed by the table entry pointer (pData) has changed, the upper layers call the appropriate API function that requests the storage system to save the modified data. All the save operations, the page erase and page copy operations are performed on system idle task.
The application must create a task that calls NvIdle() (which processes the pending saves, erase or copy operations) in an infinite loop.

For more details about the NVM module working, please refer section 3.5 from Connectivity Framework Reference Manual.pdf located inside the MCUXpresso SDK at <Path to SDK>\docs\wireless\Common.

4. Programming Guidelines

This section describes the details to migrate JN5169 Base device applications developed using the bare metal implementation to FreeRTOS based KW41Z.

4.1. Initialization

As we are migrating application from bare metal to FreeRTOS based platform, the Startup of the application will be handled by the FreeRTOS. As per the SDK architecture of KW41Z, all the FreeRTOS related functions are performed using the abstraction layer which is called as OSAbstraction. The cold start of the application should be performed by the Startup task initialized by OS abstraction layer. The other application specific threads are invoked from the Startup task.
The other tasks running along with the Startup task as per the priority set by the FreeRTOS scheduler are as shown below:

1) Serial Manager
2) Timer
3) MAC

For more information about creating and defining tasks using fsl layer, please refer to Connectivity Framework Reference Manual.pdf located inside the MCUXpresso SDK at <Path to SDK>\docs\wireless\Common.

1. Remove all hardware related initializations and configurations from the vAppMain() and initialize the following as a part of Startup task:

    1) Timer (TMR_Init())
    2) Memory Manager (MEM_Init())
    3) RNG HW module (RNG_Init())
    4) Cryptographic Hardware Acceleration (SecLib_Init())
    5) Peripheral used for debug messages (DbgConsole_Init(uint32_t baseAddr, uint32_t baudRate, uint8_t device, uint32_t clkSrcFreq))
    6) LED (LED_Init())
    7) Keyboard (KBD_Init(KBDFunction_t pfCallBackAdr))
    8) Watchdog configuration (APP_vSetUpWdog())
    9) Shell (app_zb_shell_init())
2. In JN5169, the APP_vMainLoop contains the continuously running while loop which consists of all the event handling code. In FreeRTOS based platform, this functionality is added as a part of main task. So, remove the APP_vMainLoop and add its code inside while loop of the main task for FreeRTOS based application. This will lead to the below changes for Startup of the application:

KW41Z:

```
void main_task (uint32_t parameter)
{
            MEM_Init();
            RNG_Init();
// All other hardware related initialization
            vAppMain(); // APP related initialization

            while(1)
    {
            /* Place event handler code here… */
    }
}
```

3. PDM Initialization: The Persistent Data Manager (PDM) module handles the storage of stack context data and application data in Non-Volatile Memory. The PDM module should be initialized during both a warm start and a cold start using the function PDM_eInitialise(). This function builds a file system in RAM containing information about the segments that it manages in EEPROM.

JN5169:

```
PUBLIC PDM_teStatus PDM_eInitialise(
            uint8         u8NumberOfEEPROMsegments
#ifndef PDM_NO_RTOS
            ‘
            OS_thMutex      hPdmMutex
#endif);
```

Where:

- **u8NumberOfEEPROMsegments** is a number of contiguous EEPROM segments to be used by PDM (zero value indicates that the all segments should be used)
- **hPdmMutex** is an optional handle of the mutex in order to serialize PDM calls. The mutex is not available when using the PDM in applications developed with a ZigBee 3.0 SDK (JN-SW-4170 or JN-SW-4270) or the IEEE802.15.4 SDK (JN-SW-4163), in that case the flag PDM_NO_RTOS must be defined in the Makefile. The function parameter hPdmMutex is then disabled.

FRDM-KW41Z :

```
PUBLIC PDM_teStatus PDM_eInitialise(
            uint16        u16StartSegment,
            uint8         u8NumberOfEEPROMsegments
#ifndef PDM_NO_RTOS
            ‘
            OS_thMutex      hPdmMutex
#endif);
```

Where:

- **u16StartSegment** is a start segment of EEPROM
- **u8NumberOfEEPROMsegments** : Same as explained for JN5169
- **hPdmMutex** : Same as explained for JN5169

So, initialize the PDM module in the application as shown below:

<u>JN5169</u>:

```
PDM_eInitialise(63);
```

<u>FRDM-KW41Z</u>:

```
PDM_eInitialise(700, 63);
```

4. During the application initialization (inside the function APP_vInitialiseNode()), set the macaddress to specified location in SOC. This can be achieved using the function SOC_ZPS_vSetMacAddrLocation(). In JN5169, it is already handled inside the library.
5. Remove the function vDeletePDMOnButtonPress(). It is used to clear PDM context. This is covered in the factory reset functionality using switch.
6. Base device application has the following build configurations:
   - Coordinator
   - Router
   - End Device

In all the build configurations, initialization of the ZigBee Pro Stack, ZCL and BDB is performed using the functions ZPS_eAplAfInit(), APP_ZCL_vInitialise() and APP_vBdbInit() respectively.

ZigBee Pro Stack initialization:
The function ZPS_eAplAfInit() initializes the Application Framework. It will first request a reset of the Network layer of the ZigBee PRO stack and then initialize certain network parameters with values that have been pre-configured using the ZPS Configuration Editor.

ZCL initialization:
APP_ZCL_vInitialise() initializes the ZigBee Cluster Library and registers a base device with the ZCL layer by creating instances of different clusters.

BDB initialization:
APP_vBdbInit() initializes BDB attributes to their default values only if those are not already defined in the application (bdb_options.h file), registers Base Device message queue, sets the initial security keys based on device types, opens the timers for Base Device internal use.

Set PAN ID (For Coordinator and Router):
If application state is not E_RUNNING, then set PAN ID using ZPS_vNwkNibSetPanId() function. Use random number generator function RND_u32GetRand() by giving minimum value to 1 and maximum value to 0xfff0 to generate random PANID. In JN5169, PAN ID is set inside library.

Set TX buffers (For Router and End Device):
For router and End Device node, TX buffers are set by calling the function ZPS_u32MacSetTxBuffers().

Pre-sleep and Post-sleep Callback Functions Registration (For End Device):
In order to implement low power modes, we must provide user-defined callback functions to perform housekeeping tasks when the node enters and leaves low power mode. Registration functions (PWRM_vRegisterPreSleepCallback() and PWRM_vRegisterWakeupCallback()) are provided for these callback functions to ensure that callback is registered during a cold start. The registration functions must be called in the function vAppRegisterPWRMCallbacks( ).

```
void main_task (uint32_t parameter)
{
   MEM_Init();
   RNG_Init();
   vAppRegisterPWRMCallbacks(); // Registers PreSleep and PostSleep Callbacks
// Registers All other hardware related initialization
   vAppMain(); // Registers App related initializations

   while(1)
   {
          /* place event handler code here */
   }
}
```

ng the device into
device enters sleep
PreSleep)) saves the
MAC settings, disables tick timer, sets up wake up condition of DIO pins, disables specified UART. In KW41Z, Pre-sleep callback function should clear pending IRQs, disable clock, clear status flags. This can be achieved by replacing the code inside PWRM_CALLBACK(PreSleep) with the function vTimerServerDeinit(),which is the wrapper function for timer server manager.

FRDM-KW41Z:

```
PWRM_CALLBACK(PreSleep)
{
   vTimerServerDeinit ();
}
```

Similarly, Post-sleep callback function is called by the Power Manager just after the device leaves low-power mode (irrespective of how the device was woken from sleep) and should perform any housekeeping tasks that are necessary after the device wakes from sleep. For Example, In JN5169 based application, during Post-sleep callback, the tasks which are disabled before sleeping are re enabled. In FRDM-KW41Z, this can be achieved by using the function vTimerServerInit() and the polling can be started using APP_vStartUpHW().

FRDM-KW41Z:

```
PWRM_CALLBACK (Wakeup)
{
   VTimerServerInit ();
   APP_vStartUpHW ();
}
```

4.2. Message queues

1. A message queue can be created using the function ZQ_vQueueCreate(). This function allows the queue size, location in memory where the queue should start and the size of a message to be specified. A queue is given a unique handle, which is a pointer to a tszQueue structure containing information about the queue. The ZigBee queue structure tszQueue is shown below:

For JN5169 ZigBee 3.0 SDK (JN-SW-4170):

```
typedef struct
{
   uint32 u32Length;
   uint32 u32ItemSize;
   uint32 u32MessageWaiting;
   void  *pvHead;
   void  *pvWriteTo;
   void  *pvReadFrom;
}tszQueue;
```

Where:

- **u32Length** is the size of the queue in terms of the number of messages that it can hold
- **u32ItemSize** is the size of a message, in bytes
- **u32MessageWaiting** is the number of messages currently in the queue
- **pvHead** is a pointer to the beginning of the queue storage area
- **pvWriteTo** is a pointer to the next free place in the storage area where a new message can be written
- **pvReadFrom** is a pointer to the next message to be read from the storage area

For MCUXpresso based ZigBee 3.0 SDK:

```
typedef struct
{
   list_t list;
   uint32 u32ItemSize;
} tszQueue;
```

Where:

- **u32ItemSize** is the size of a message, in bytes
- **list** is an object of a structure list_t (list_t is a structure containing information about the queue),the structure list_t is shown below:

```
typedef struct
{
  struct listElement_tag *head;
  struct listElement_tag *tail;
  uint16_t size;
  uint16_t max;
}list_t;
```

Where:

- **head** is a pointer to the beginning of the queue storage area
- **tail** is a pointer to the end of the queue storage area
- **max** is a size of the queue in terms of the number of messages that it can hold

In JN5169, buffer of the specific queue size is defined and it is passed to the function ZQ_vQueueCreate() (as 4th argument) to initialize pvHead pointer with the address of buffer. pvHead specifies the location in memory where the queue should start. While in KW41Z, head (pointer to the beginning of the queue storage area) is initialized with the NULL. So, the argument pu8StartQueue of the function ZQ_vQueueCreate() should be NULL as shown below:

```
ZQ_vQueueCreate(&APP_msgAppEvents,    APP_QUEUE_SIZE,    sizeof(APP_tsEvent),  NULL);
```

2. For Coordinator device, remove APP_msgSerialTx and APP_msgSerialRx message queues. On JN5169, these message queues are used for transmitting and receiving serial data. On KW41Z, it is handled by shell framework module.

4.3. Timers

Application resource initialization:

1. Remove button scan timer as it is handled by keyboard framework module.

JN5169:

```
ZTIMER_eOpen(&u8TimerButtonScan,    APP_cbTimerButtonScan,  NULL,
ZTIMER_FLAG_PREVENT_SLEEP);
```

FRDM-KW41Z:

```
void KBD_Init( KBDFunction_t pfCallBackAdr )
{
        /* timer is used to determine short or long key press */
        mKeyScanTimerID = TMR_AllocateTimer();

#if gKBD_TsiElectdCount_c
        mTsiSwTriggerTimerID = TMR_AllocateTimer();
#endif
}
```

2. Replace the button scan timer callback by the keyboard callback function as per the switches on both the hardware. On FRDM-KW41Z the keyboard module will provide a key code to the callback function for all the switches and key press type (short and long key press). The application needs to define application events corresponding to all switches and keypress types.

FRDM-KW41Z:

```
typedef enum {
            APP_E_BUTTONS_BUTTON_1 = 0,
            APP_E_BUTTONS_BUTTON_SW2,
            APP_E_BUTTONS_BUTTON_SW3,
            APP_E_BUTTONS_BUTTON_SW4,
            APP_E_BUTTONS_BUTTON_SW5,
 } APP_teButtons;
```

Inside the keyboard callback, set the application event based on the key code, set the event type and post the event type to the application message queue.

FRDM-KW41Z:

```
PRIVATE void KBD_Callback( uint8_t events)
{
   APP_tsEvent sButtonEvent;
   switch(events)
   {
      case gKBD_EventPB1_c:
         /* SW5 on FRDM-KW41Z*/
         sButtonEvent.uEvent.sButton.u8Button =
APP_E_BUTTONS_BUTTON_SW5;
         break;
          /* similarly for all switches and events */
    }
        sButtonEvent.eType = APP_E_EVENT_BUTTON_DOWN;
       ZQ_bQueueSend(&APP_msgAppEvents, &sButtonEvent)
```

4.4. JN516x development board to FRDM-KW41Z hardware changes (LED and Keyboard)

LED and Keyboard functionality is handled by JN5169 hardware specific APIs. On FRDM-KW41Z, it is handled by framework module. So, all the hardware specific APIs of JN5169 based platform related to LED and keyboard are replaced by APIs exposed by MCUXpresso framework's LED and Keyboard modules. LED ON and toggle functionality is handled by LED module.

1. Remove the file app_leds.h.
2. Remove the LED and button initialization as they are already done at LED and Keyboard module initialization from main_task:

   JN5169:

```
APP_vLedInitialise();
APP_bButtonInitialise();
```

   FRDM-KW41Z :

```
void LED_Init(void);
void KBD_Init( KBDFunction_t pfCallBackAdr ){
kbdSwButtons[i].config_struct.pSwGpio = &switchPins[k++];
```

Coordinator:

On JN5169 based platform, we have only white LEDs while on FRDM-KW41Z we have LED3 and LED4 (RGB LED) which is used in the application for different functionalities. For LED functionality, application needs to add APP_vSetLedState() function inside app_zb_utils.c file.

Functionality	**LED State**	**MCUXpresso LED module API**
Factory default state	LED3 flashing	Led1Flashing()

Successfully formed network	LED3 solid red	StopLed1Flashing()
		Led1On()

During Identify request	LED3 solid red	Led1On()
After completing identify	LED3 OFF	Led1Off()

Table 3. MCUXpresso LED module APIs to set different LED states

Add below line inside APP_taskCoordinator() function to set LED state:

```
APP_vSetLedState (TRUE == sBDB.sAttrib.bbdbNodeIsOnANetwork);
```

NOTE: LED3 is named as LED1 in source code.

For base Coordinator application, DR1199 generic expansion board is used with DR1174 carrier board for JN5169. Different buttons are used for triggering different functionalities on JN5169 development board and FRDM-KW41Z. The mapping of functionality and switches for FRDM-KW41Z is as shown below:

Functionality	**Switch on JN5169**	**Switch on FRDM-KW41Z**
Forming a Network	DIO8 APP_E_BUTTONS_BUTTON_1	SW3 APP_E_BUTTONS_BUTTON_SW3
Allowing other devices to join	SW2	SW4 APP_E_BUTTONS_BUTTON_SW4
Binding Devices	SW4	SW2 APP_E_BUTTONS_BUTTON_SW2
Operating the Device	SW1	SW5 APP_E_BUTTONS_BUTTON_SW5
Re-joining the Network	Reset	Reset
Performing Factory Reset	DIO8 + RESET	SW2/SW3/SW4/SW5 Long press APP_E_BUTTONS_BUTTON_1

Table 4. Mapping of functionality and switches for FRDM-KW41Z

As per above mapping, application needs to add handling for the different switches inside APP_taskCoordinator() function.

```
case APP_E_BUTTONS_BUTTON_1:
/* Factory Reset */

if (ZPS_E_SUCCESS != ZPS_eAplZdoLeaveNetwork(0, FALSE,FALSE))
{
        /* Leave failed, so just reset everything */
        DBG_vPrintf(TRUE,"Deleting the PDM\n");
        APP_vFactoryResetRecords();
        ResetMCU();
}
break;
```

For factory reset, application should attempt to complete all the NVM related pending operations such as queued writes, page copy, and page erase and save on interval requests. For doing this, application needs to call NvCompletePendingOperations() function at the end of function APP_vFactoryResetRecords().

Similarly, Inside APP_taskCoordinator() function, application need to change buttons handling for different functionalities.

Router:

On JN516x development board, we have only white LEDs while on FRDM-KW41Z we have LED3

and LED4 (RGB LED) which is used in the application for different functionalities. The LED states can be set as per the functionalities using APP_vSetLedState(TRUE == sBDB.sAttrib.bbdbNodeIsOnANetwork) inside APP_taskRouter(). Refer to Table-3 to use MCUXpresso LED module API to set different LED states for different functionalities. The setting of LED states for binding and initial states should be handled in app_zcl_task.c as below:

1. Change the definition of APP_vSetLed as below and modify all the calls to this function by adding LEDtype(LED_RGB , LED1/Red LED) which defines LED1 or RGB_LED as per the functionality. For example, to use LED1 at initialization inside APP_ZCL_vInitialise call APP_vSetLed as below:
 APP_vSetLed(LED1,sBaseDevice.sOnOffServerCluster.bOnOff);

Similarly, set Ledtype for all as required:

2. APP_vHandleIdentify
2. vIdEffectTick - RGB LED
3. APP_vHandleClusterCustomCommands - RGB LED

```
PUBLIC void APP_vSetLed(uint8 u8Led,bool_t bOn){
        if(u8Led & LED_RGB)
        {
                if(bOn)
                {
                        Led2On();
                        Led3On();
                        Led4On();
                }
                else
                {
                        Led2Off();
                        Led3Off();
                        Led4Off();
                }
        }
        else
        {
                (bOn) ? Led1On() : Led1Off();

        }
}
```

NOTE: LED3 is named as LED1 and RGB LED is named as LED2, LED3, and LED4 for Red, Green, and Blue respectively in the code.

For the JN5169 Base device router application, the DR1175 is used along with DR1174 carrier board. This setup has limited switches so multiple functionalities are performed using a single switch. On FRDM-KW41Z board, we have more switches so we can use one switch per functionality for covering all base device functionalities. Please refer to section 4.3.8 from AN12061-MKW41Z-AN-Zigbee-3-0-Base-Device.pdf for mapping of router functionality and switches. As per this mapping, the application needs to add the handling of different switches inside APP_taskRouter() as shown in coordinator section.

End Device:

For base End Device application, DR1199 generic expansion board is used with DR1174 carrier board for JN5169. So, different buttons are used for performing different functionalities on JN5169 development board and FRDM-KW41Z. Please refer to section 4.4.8 from AN12061-MKW41Z-AN-Zigbee-3-0-Base-Device.pdf for mapping of end device functionality and switches. As per this mapping, the application needs to add the handling of different switches inside APP_taskEndDevice() as shown in coordinator section. The different functionalities are indicated by different LED states. For setting the LED states as per the functionalities, add below code inside APP_ taskEndDevice() function :

```
if(u8KeepAliveTime == 0) // If device enters low power, turn off LED3
{
            StopLed1Flashing();
            Led1Off();
 }
 else
 {
      if(TRUE == sBDB.sAttrib.bbdbNodeIsOnANetwork)  // Check if device has
joined network
           Led1On();
      else
      {
          Led1Flashing();  // Factory default state
       }
 }
```

4.5 Low-Power Modes (For End Device Only)

A number of low-power modes are available on the JN5169 device. In descending order of power consumption, the modes are:

- Doze mode (CPU clock is stopped, but all other parts of the device continue to run)
- Sleep modes :
  - Sleep with memory held (all power domains are powered down except those for the on-chip RAM and VDD supply)
  - Sleep without memory held (all power domains are powered down except the VDD supply)
- Deep Sleep mode (all switchable power domains are powered down and the 32-kHz oscillator is stopped)

On KW41Z, Power Management Control Unit (PMC) provides multiple power options to allow the users to optimize power consumption for the level of functionality needed. Depending on the stop requirements of the application, a variety of stop modes are available that provide state retention, partial power down or full power down of certain logic and/or memory. For each run mode there is a corresponding wait and stop mode. Wait modes are similar to ARM sleep modes. Stop modes are similar to ARM sleep deep mode. The three primary modes of operation are run, wait and stop. The various power modes are:

- Normal Run (all peripherals clock off)
- Normal Wait (Allows peripherals to function, while allowing CPU to go to sleep)
- VLPR (Very Low Power Run - all peripherals off)
- VLPW (Very Low Power Wait - Similar to VLPR, with CPU in sleep to further reduce power)
- VLPS (Very Low Power Stop - Lowest power mode with ADC and all pin interrupts functional)

For more information about power modes in KW41Z, please refer to the section 8.3 from MKW41Z/31Z/21Z Reference Manual.

MCUXpresso framework allows to enable/disable low power related code through the macro cPWR_UsePowerDownMode. Application can enable low power mode by setting value of cPWR_UsePowerDownMode to 1 (inside config.h). Low-power can be disabled at run-time using PWR_DisallowDeviceToSleep() from the application.

MCUXpresso framework provides low- power module which maintains a global variable (mLPMFlag) that enables/prevents the system to enter deep sleep. The system is allowed to enter deep sleep only when value of mLPMFlag is zero. Every software layer that needs to keep the system awake, calls PWR_DisallowDeviceToSleep(). The function PWR_DisallowDeviceToSleep() sets a critical section to prevent the system from entering low-power and increments the value of mLPMFlag. As software layers/entities decide that they can enter deep sleep, they call PWR_AllowDeviceToSleep(). The function PWR_AllowDeviceToSleep() clears the critical section set by PWR_DisallowDeviceToSleep() and decrements the value of mLPMFlag. There should be one call to PWR_AllowDeviceToSleep() for every call to PWR_DisallowDeviceToSleep().

When the node is inactive, the Power Manager will put the device into the lowest power mode possible. The function PWRM_vInit() is used to initialize the Power Manager and specify the low-power mode in which the device should be put when inactive. For example, to configure LLS3 low power mode, define it in app_framework_config.h as below:

```
/* low power mode - MCU in LLS3 mode */
#ifndef  cPWR_DeepSleepMode
  #define cPWR_DeepSleepMode       3
#endif
```

To set LLS3 low power mode, pass 3 (mode) as a parameter of PWRM_vInit() API.

The sleepy End Device enters in a low power mode when there is nothing to process and it wakes up by an interrupt. When device is woke up from keyboard interrupt, the function App_SedWakeUpFromKeyBoard() is called from the MCUXpresso based SDK. So, application should add the function App_SedWakeUpFromKeyBoard() at the application level, which sets the keep alive time and manages poll, sleep. This can be implemented as below:

```
PUBLIC void App_SedWakeUpFromKeyBoard(void)
{
    /* stop sleeping for KEEP_ALIVETIME */
    u8KeepAliveTime = KEEP_ALIVETIME;
    APP_cbTimerPoll(NULL);
}
```

4.6 Over-The-Air Upgrade (OTA)

The over the air upgrade cluster provides the facility to upgrade (or downgrade or re-install) application software on the nodes of a ZigBee PRO network. The software upgrade is performed from a node which acts as an OTA Upgrade cluster server, which can obtain the upgrade software from an external source. The nodes that receive the upgrade software act as OTA Upgrade cluster client. The upgrade application is downloaded into Flash memory associated with the device on the client node. This may be internal or external Flash memory. In the case of an external Flash device, the upgrade image must then be transferred to internal Flash memory. By default, OTA upgrade images are downloaded to a Flash memory device that is external to the device of the OTA Upgrade client.

The FRDM-KW41Z board provides external Flash memory for over the air programming (OTAP) support, while the JN5169 uses internal Flash memory for the same. In JN5169, hardware abstraction layer (AppHardwareApi_JN516x.h) provides APIs to access internal and external Flash memory. MCUXpresso based SDK provides OtaSupport module which contains APIs to access external Flash memory.

Description	JN5169	FRDM-KW41Z
Initializes memory	bAHI_FlashInit	OTA_InitExternalMemory
Erases the specified sector of memory	bAHI_FlashEraseSector	OTA_EraseBlock
Reads data from the memory	bAHI_FullFlashRead	OTA_ReadExternalMemory
Writes a data into the memory	bAHI_FullFlashProgram	OTA_WriteExternalMemory

Table 5. APIs to access Flash memory

For more information about how to implement OTA upgrade mechanism and handle OTA states using OTA state machine, please refer file “app_ota_client.c” of application “JN-AN-1218-Zigbee-3-0-Light-Bulb”.

5. Appendices

A. NFC & Installation code

The ZigBee Base Device allows devices to join a network using unique install codes. This install code is only used for the initial join and is replaced by the Trust Centre immediately after joining. The commissioning of a node into the network using installation code for generating the preconfigured link key can be considered as out of band commissioning. In out of band commissioning, the network parameters are passed using NFC or Serial commands. Adding support of NFC on JN516x development board as well as FRDM-KW41Z requires NXP NTAGI2C PLUS Kit. For hardware setup, please refer to section 3.1 from JN-AN-1217-ZigBee-3-0-Base-Device.pdf.

The NTAG connects with FRDM-KW41Z via I2C interface. The application needs to download the NTAG I2C Plus drivers from MCUXpresso website to add support of NTAG I2C Plus explorer kit on FRDM-KW41Z.

On JN5169 based platform, the installation code using NFC is covered at "<Application>/Common/Source/app_ntag_icode.c".

6. References

The following manuals will be useful in developing custom applications based on this Application Note:

- ZigBee 3.0 Stack User Guide [JN-UG-3113]
- ZigBee 3.0 Devices User Guide [JN-UG-3114]
- JN51xx Core Utilities User Guide [JN-UG-3116]
- JN516x Integrated Peripherals API User Guide [JN-UG-3087]
- ZigBee 3.0 Base Device Application Note [JN-AN-1217]
- MCUXpresso SDK API Reference Manual_MKW41Z4
- MKW41Z/31Z/21Z Reference Manual MKW41Z512RM
- Getting Started with MCUXpresso SDK
- Connectivity Framework Reference Manual