RELEASE NOTES
JN518x ZigBee 3.0 SDK

JN-SW-4470

Build v2042

NXP Semiconductors

For the contact details of your local NXP office or distributor, refer to:

www.nxp.com

CONTENTS

RELEASE SUMMARY (v2042)

The JN518x ZigBee 3.0 Software Developer's Kit (JN-SW-4470) contains software resources needed to develop ZigBee 3.0 applications for the NXP JN518x wireless microcontrollers on Windows and Linux platforms. This SDK must be installed on top of the MCUXpresso toolchain (**MCUXpressoIDE_10.2.1 [Build 795]**) (see Section 3).

This is a certified Zigbee PRO stack (Revision 22).

1. Software Components

This release of the JN-SW-4470 software includes the components and versions detailed in the **SW-Content-Register-JN-SW-4470.txt.**

2. Supported Hardware Products

This software release supports the following hardware products:

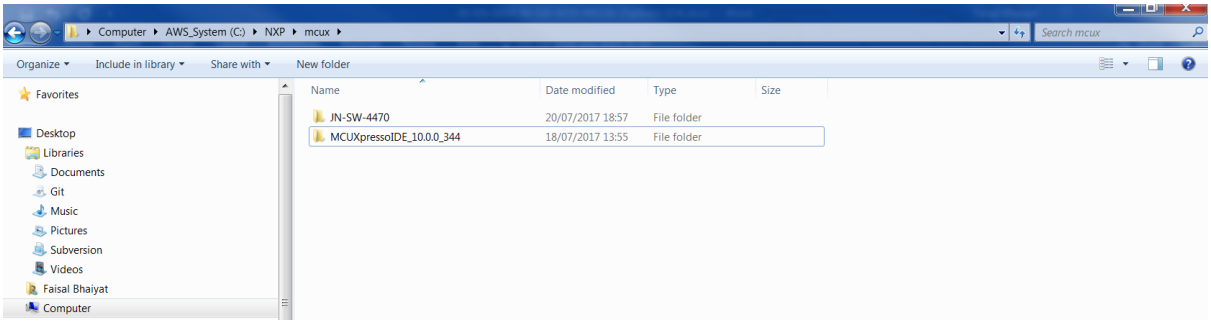
Chips	Modules	Evaluation Kits
JN5180-001 **(ES2 Only)**	JN5180-M10 (OM15059)	OM15076-3 Carrier Board OM15076-1 Carrier Board OM15081-1 Light/Sensor Expansion Board OM15082-2 Generic Expansion Board JN518x USB Dongle

3. Installation

This SDK (JN-SW-4470) in intended to be used with Eclipse-based MCUXpresso **(MCUXpressoIDE_10.2.1 [Build 795]).** The SDK is distributed as a self-extracting executable.

Installation steps:

1. Download the SDK installer (**JN-SW-4470 JN518x Zigbee 3.0 v2042.exe**)
2. Run the executable (**JN-SW-4470 JN518x Zigbee 3.0 v2042.exe**) in to a convenient location, such as C:\NXP\mcux. The SDK contains the device drivers, framework, connectivity stack and example projects.
3. Download **MCUXpresso** – see section 3.1. MCUXpresso toolchain must be installed at the same location as the SDK. After installing the SDK and MCUXpresso IDE you should have the directory structure shown below:



4. Launch the **MCUXpresso** installer and follow the on-screen instructions to install.
5. To add JN518x support to **MCUXpresso** it is necessary to add a number of files to the installation. These files are provided within the SDK which was installed during step 2. These can be found in the '**C:\NXP\mcux\JN-SW-4470\tools\lpcxpresso\bin**' folder.
6. Copy the '**C:\NXP\mcux\JN-SW-4470\tools\lpcxpresso\bin'** folder to '**C:\NXP\mcux\ MCUXpressoIDE_10.2.1 _795\ide**'. There should already be a 'bin' folder there and the copied version should be merged with it. If using

Windows Explorer to copy the folder across, Windows will complain that it is there already: this is correct, so click ‘Yes’ and/or ‘Copy and Replace’ as required.

Note that using the command line to copy the files may incorrectly set the access permissions whereas CTRL-C and CTRL-V in Windows Explorer work fine.

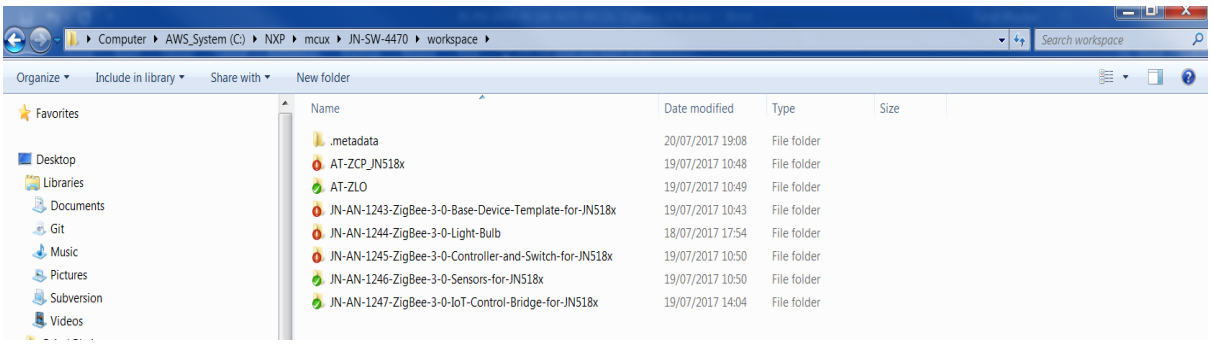
7. Run ‘mcuxpressoide.exe’ and select a folder to use as Workspace. This should be the folder where the SDK was installed ‘**C:\NXP\mcux\JN-SW-4470\workspace’**.
8. JN518x bootloader requires an image signature to verify the validity of the image. The Binary image generated is signed after the image is built in a two-stage process. The image signing tool is implemented in python. This requires an installation of python to exist. Python 2.7 is required. Python can be downloaded from the following web page:

   https://www.python.org/downloads/

   Once python is installed, it should be added to the windows path.
9. The python file is now using a crypto module, therefore you may have to install the module to compile successfully:
   C:\Python27\Scripts>pip install pycryptodome (python 2.7.13 (https://www.python.org/downloads/release/python-2713/) is required to have the pip tool)
10. The application notes are not provided as part of the Zigbee stack installer. They are provided separately as zips. The application notes should be extracted into the workspace created in step 7.

    C:\NXP\mcux\JN-SW-4470\workspace

    The directory structure may look like this:



3.1 MCUXpresso Installation

MCUXpresso can be obtained from the following NXP web page:

http://www.nxp.com/mcuxpresso/ide

To develop JN518x applications without limitation, we recommend that you purchase the Pro edition of MCUXpresso, however the free edition you can develop with applications up to 256KB.

The required version of MCUXpresso for this SDK release is:

MCUXpressoIDE_10.2.1 [Build 795] (MCUXpressoIDE_10.2.1_795)

Important: *This is the version with which the libraries within the SDK were compiled and verified. Other versions of* **MCUXpresso** *may not be compatible with the contents of the SDK and cannot be guaranteed to work or be supported with the JN51xx devices.*

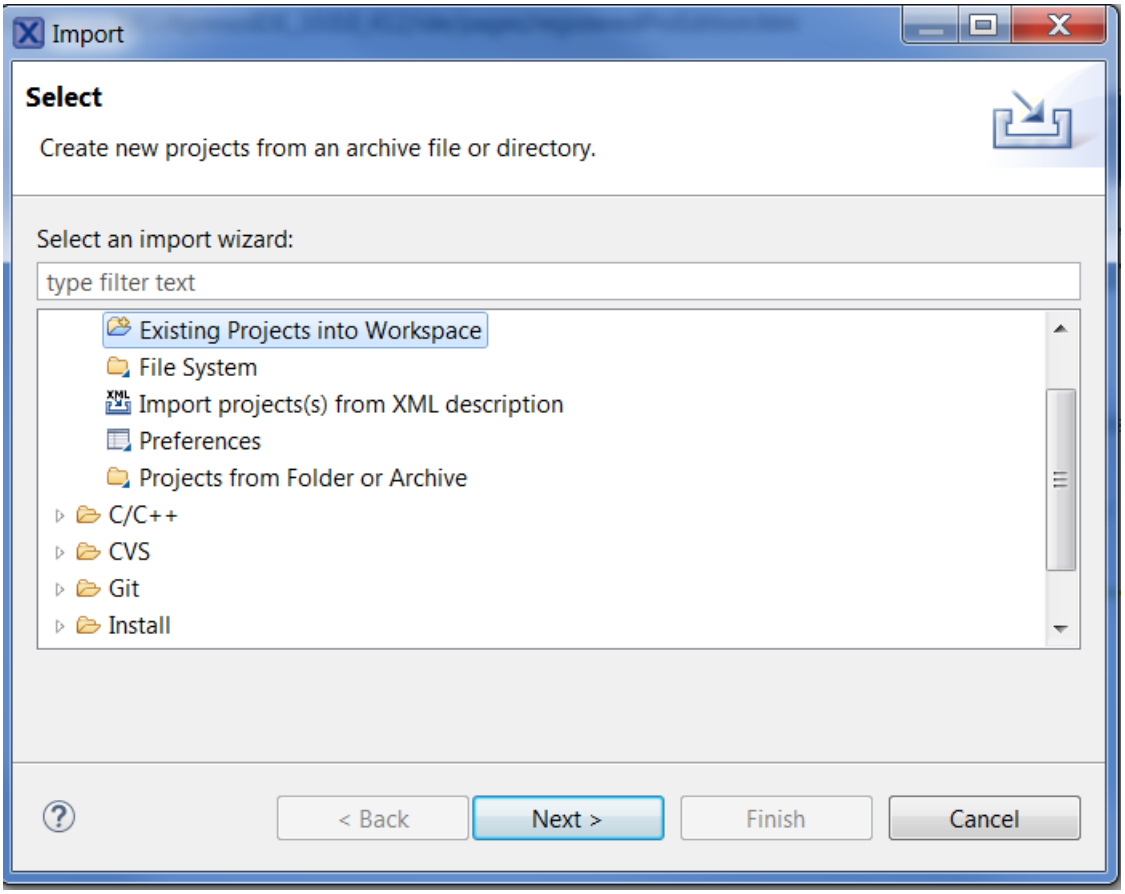
To obtain **MCUXpresso** and install it on your development machine:

1. If you do not already have a web account with NXP, navigate to www.nxp.com and create an account.
2. Sign in to your NXP web account.
3. Navigate to the page www.nxp.com/mcuxpresso/ide
4. Select the **Downloads** tab and then click the **Download** button.
5. Check whether the displayed version is the recommended version indicated above:
   - If it is the recommended version, download it.
   - If it is not the recommended version, click **Previous** and then select the recommended version and download it.

Full installation details are provided in the **MCUXpresso** *IDE Installation and Licensing Guide*, available on the **Documentation** tab of the above web page.

3.2 Importing Projects into MCUXpresso

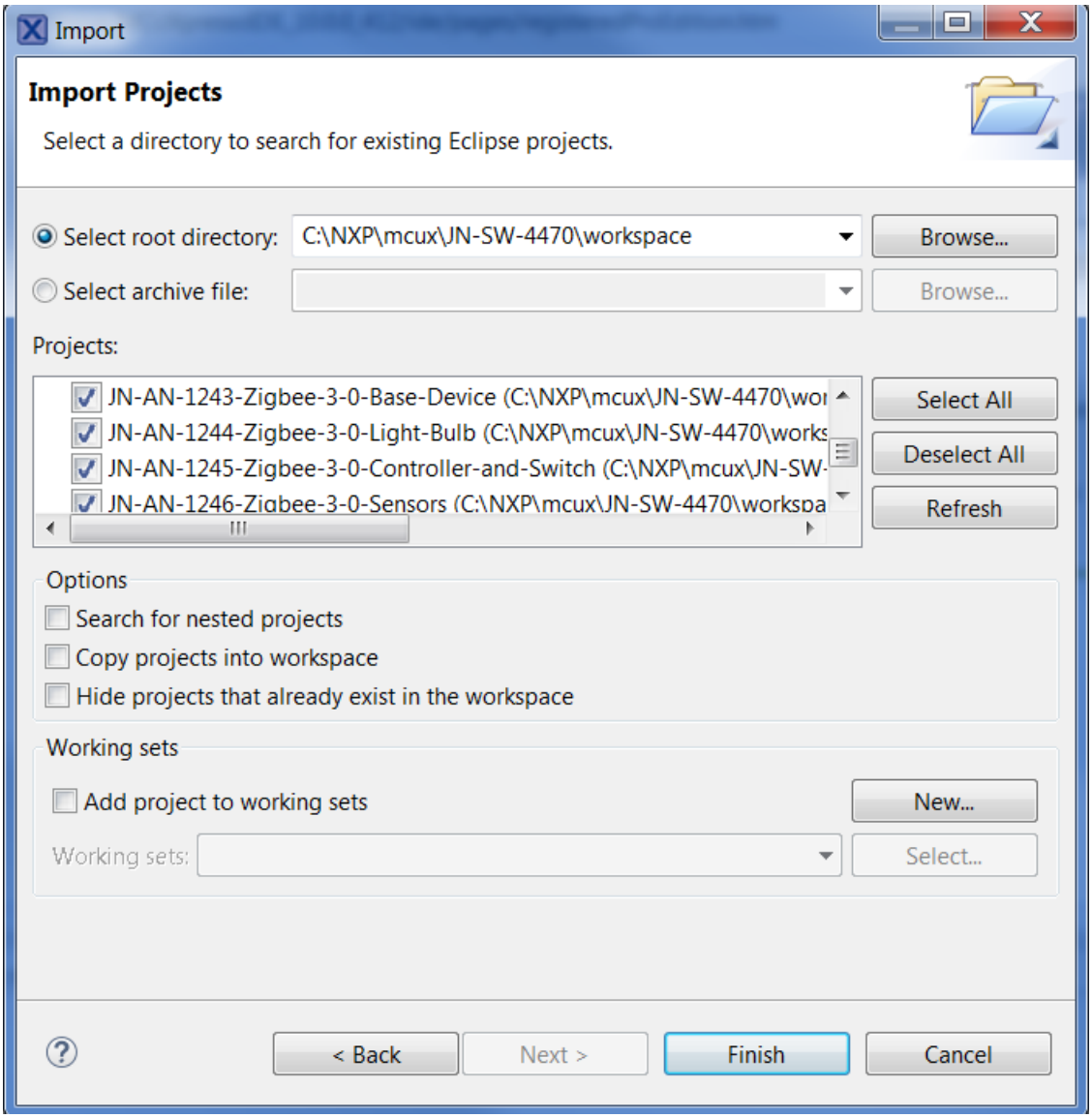
To import the example projects into MCUXpresso, click File -> Import… This will bring up a dialogue box. Select General -> Existing Projects into Workspace then press ‘Next’:



The next dialogue box allows you to select the location that the projects can be found at. ‘Select root directory’ will already be selected.

Click the ‘Browse…’ button to the right and then navigate to the folder where the SDK was installed. Navigate further into ‘**C:\NXP\mcux\JN-SW-4470\workspace’** then press ‘OK’.

Back in the original dialogue box ensure that ‘Copy projects into workspace’ is not selected, then press ‘Finish’:



The projects will now appear in the Project Explorer panel. To build one, select it then press the build button on the toolbar (looks like a hammer).

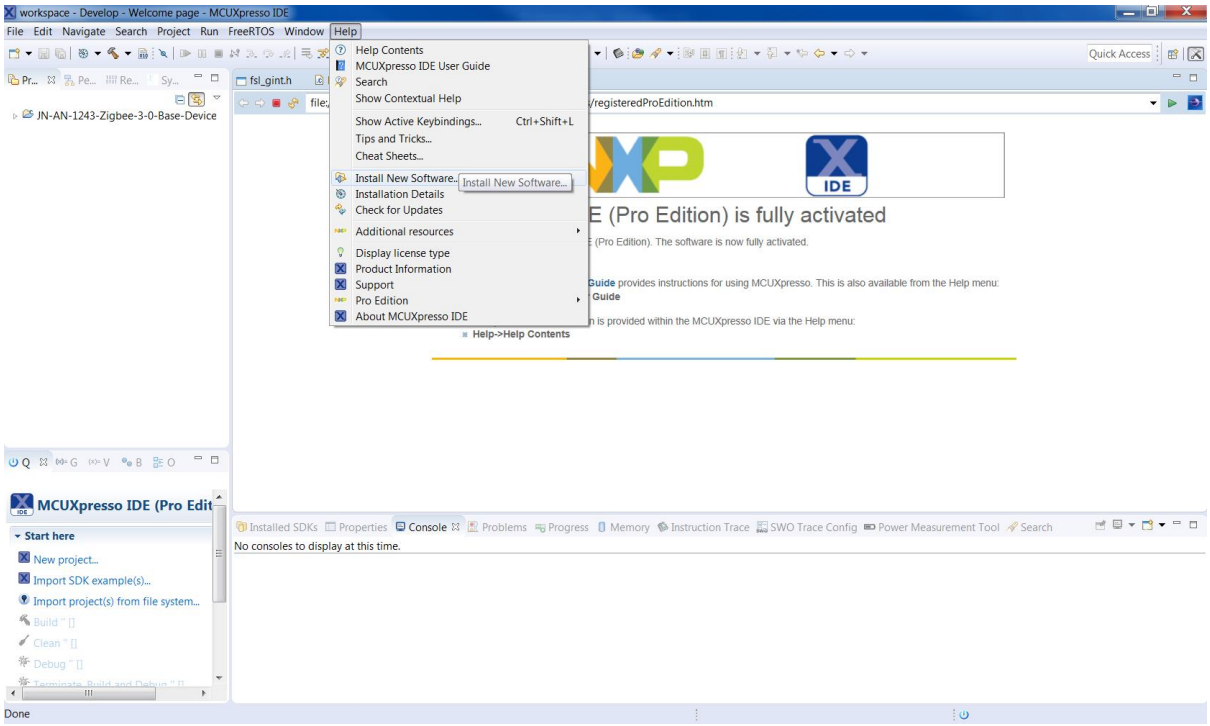
3.3 Installation of ZPSConfig plugins:

The Zigbee stack configuration plugins have been updated to be hosted as part of the MCUXpresso IDE.

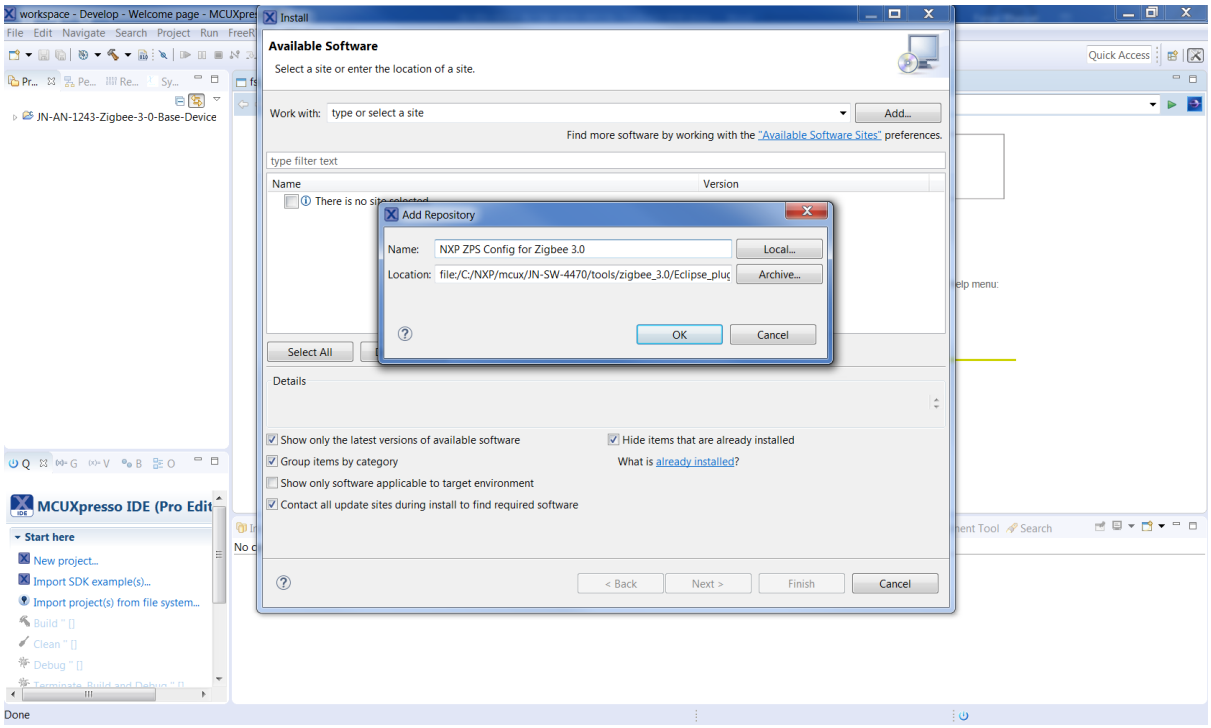
The plugins and features can be found located in the SDK folder structure as below:

C:\NXP\mcux\JN-SW-4470\tools\zigbee_3.0\Eclipse_plugins\com.nxp.sdk.update_site

1. To add the plugins. On the top menu pane of the MCUXPresso select the Help->Install New Software option



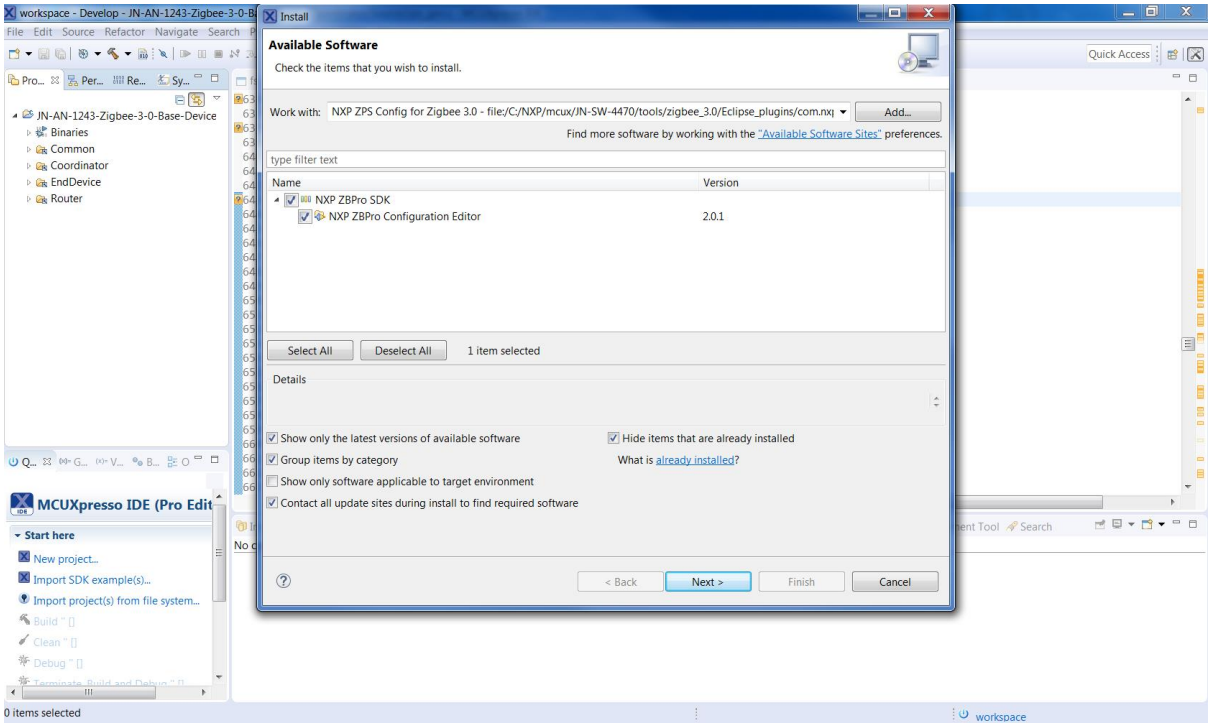
2. This should open a pop up menu selection as below:



Give the plugins a Name. This can be anything. Select the local button and browse to the location of the plugins which should be

C:/NXP/mcux/JN-SW-4470/tools/zigbee_3.0/Eclipse_plugins/com.nxp.sdk.update_site/

3. This should pull the available features to install;



4. Select the features listed and then press the “Next” button. Accept the subsequent Licence agreement and warning of Signature validation. Then press the “Finish” button and let MCUXpresso restart.
5. If the plugins have been successfully installed it will be listed in the “already installed” software list.

4. Release Details

4.1 Known Feature Limitations

The following features are not included in this release:

Feature	Description
Free RTOS	There is no RTOS support in this release. The application is designed to work as a bare metal implementation.
Flash programmer support in IDE	The flash programmer provided in this release is a command line variant. There is no MCU xPresso plugin for the JN518x. The command line flash programmer when used with multiple boards connected can fail to recognise COM ports. The MCU xPresso does support flashing of images through the SWD. This requires the setting up of the debug configuration detailed in section 4.2.1. There is contention of signal lines between the OM1580x expansion boards. The SWD image flashing mechanism will fail if the expansion board is connected.
Hardware limitation	SW4/DIO4 on OM15082 should not be used as mask for Wake.
Peripheral Drivers	All peripherals working except the DMIC
JN5189 ES1	This SDK is not compatible with ES1 devices and must not be used with ES1 device.
PSECT locations	The PSECT (protected sectors of flash) is not in the normal application flash use domain. It is used to store certain chip related configurations. It is possible to write to these pages through the flash programmer. It is **strongly advised not to write to these locations**. Writing to these sectors outside their intended use will cause the device to not function correctly.

4.2 Modifications Required

The JN518x SDK is different from previous JN516x and JN517x SDKs. The Peripherals are different and the layout of the SDK is also different.

The application notes must be placed at **C:\NXP\mcux\JN-SW-4470\workspace**.

The Zigbee stack components can be found at C:\NXP \mcux\JN-SW-4470\middleware\wireless\zigbee3.0.

The additional tools like the ZPSConfig parser and the Eclipse plugins for the ZPSconfig diagram can be found at C:\NXP \mcux\JN-SW-4470\tools\zigbee_3.0.

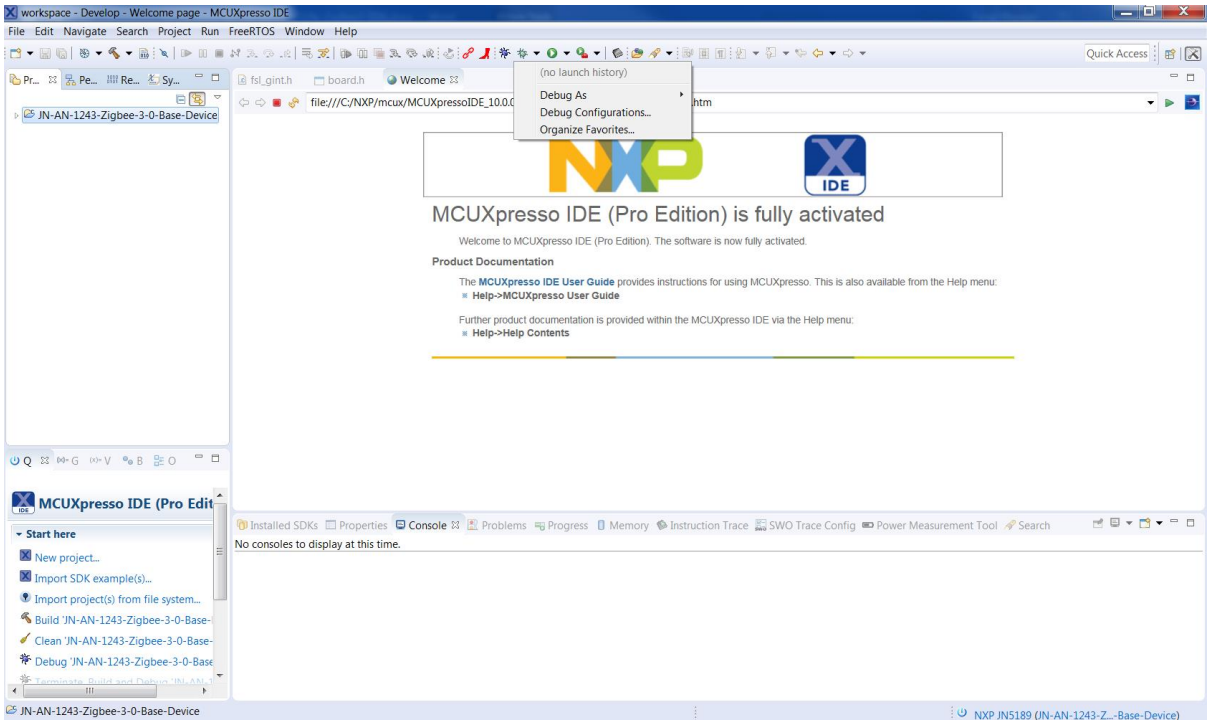
IT IS STRONGLY ADVISED THAT THESE INSTRUCTIONS ARE FOLLOWED CAREFULLY!

4.2.1 Debug Configuration

It is possible to flash the image for the JN518x by using the SWD (Single Wire Debug) functionality available in MCU xPresso.

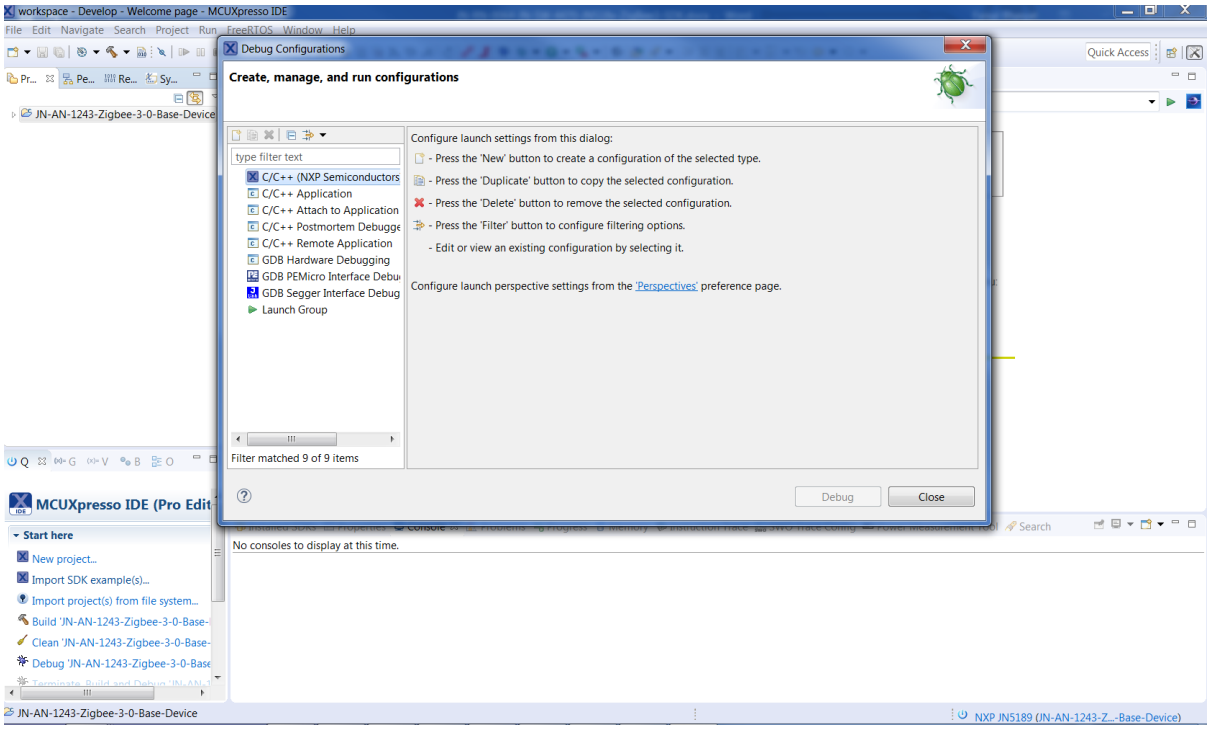
Select the dropdown menu next to the “bug” symbol on the selection panel of MCU xPresso IDE.

Select “Debug Configuration”.



The selection would bring up a pop up configuration box.

On the left-hand pane select the C/C++ NXP MCU Application and right click. That should provide drop down options, from the list select “New”.



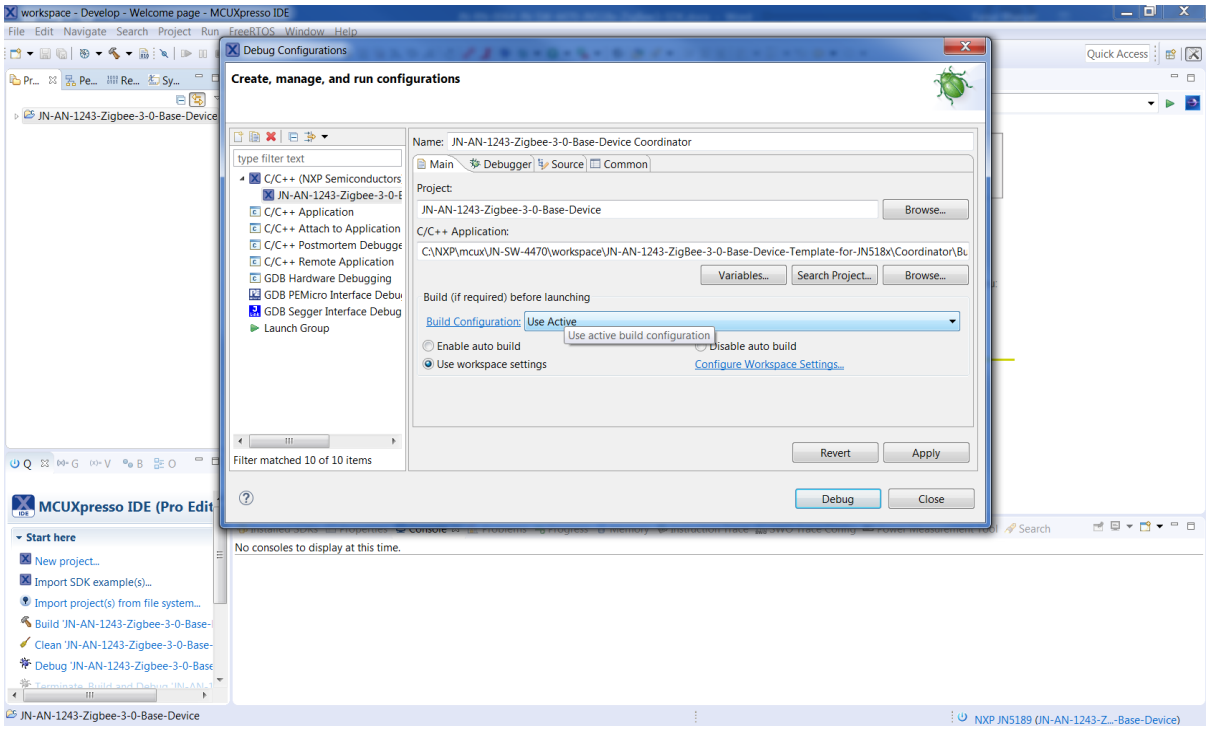
The subsequent pop up configuration box allows creating a debug configuration for the current build configuration.

In the Name box type a name of your choice for your current configuration. E.g. JN-AN-1243-Zigbee-3-0-Base-Device Coordinator

For the C/C++ Application box, use the browse button to find the ".axf" file for the image you want to debug/Flash.

e.g. C:\NXP\mcux\JN-SW-4470\workspace\JN-AN-1243-ZigBee-3-0-Base-Device-Template-for-JN518x\Coordinator\Build\jn518x_mcux\Coordinator_JN5180_DONGLE.axf

For the build configuration, use the dropdown menu to select "Use Active". Then save the changes by pressing the "Apply" button.



In the Debug Configuration pop up window, the Debugger needs to be configured.

Select the “Debugger” Tab.

This should update the existing Debug configuration pop up window with the possible configurations for the Debugger.

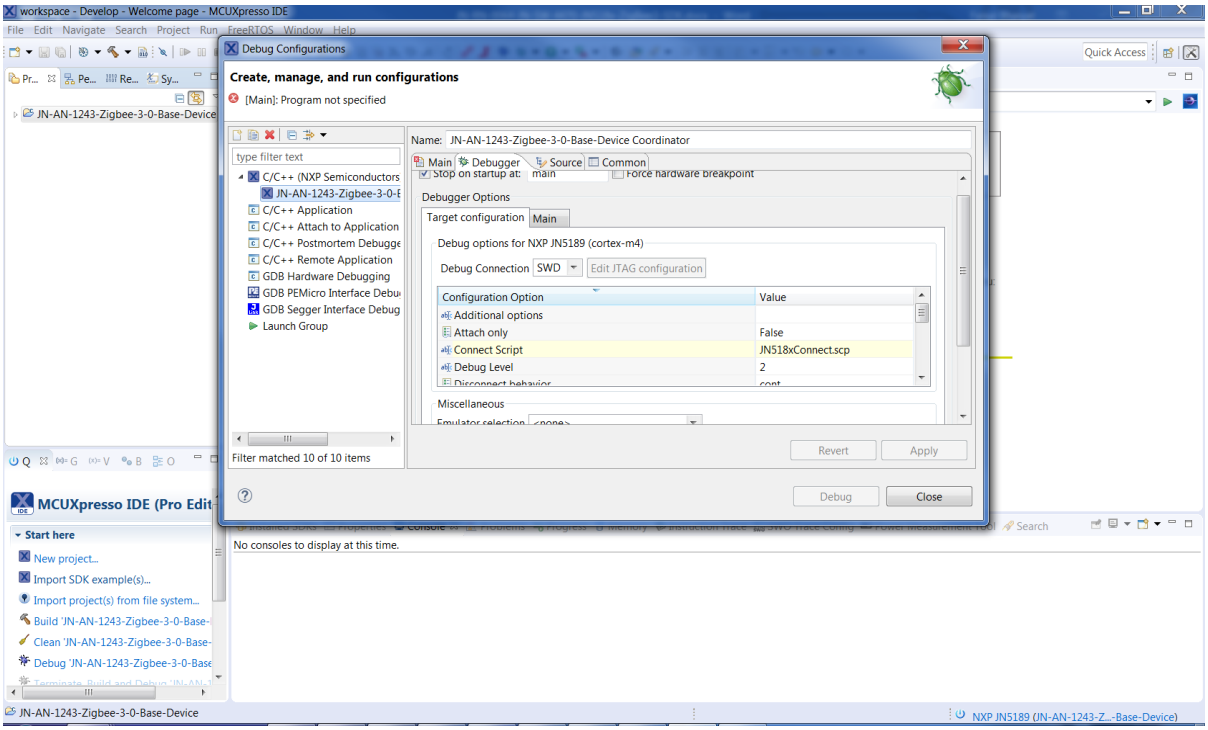
Tick the option for “Stop on startup at” and in the box type “main”.

Update the “Connect script” to be “JN518xConnect.scp”.

The “Emulator Selection” needs to be “LinkServer” and the “Debug options template” is “NXP JN5189 (*)”

To save these settings press “Apply”.

Now just press “Debug” and connect your board if not already connected. The board will be programmed with the image and this should allow a debug session to be started.



By default, there are no debug symbols present so it's possible that no source code would be presented when running a debug session.

To build with debug symbols, the following needs to be added to the Makefile "DEBUG=HW" and a clean build of the application is now required.

4.2.2 Programming firmware through the flash programmer

JN-SW-4407 JN51xx Production Flash Programmer is available as a standalone application which can be used to program the JN518x device.

To put the JN518x device into programming mode, The ISP button on the development board needs to be held pressed and then RESET is pressed once.

If programming different firmware or programming a new firmware for the first time the persistent data which is held in flash needs to be erased.

To erase all the flash:
./JN518xProgrammer.exe -V 2 -P 115200 -s COM56 -e FLASH: 646656@0

Or

./JN518xProgrammer.exe -V 2 -P 115200 -s COM56 -e FLASH

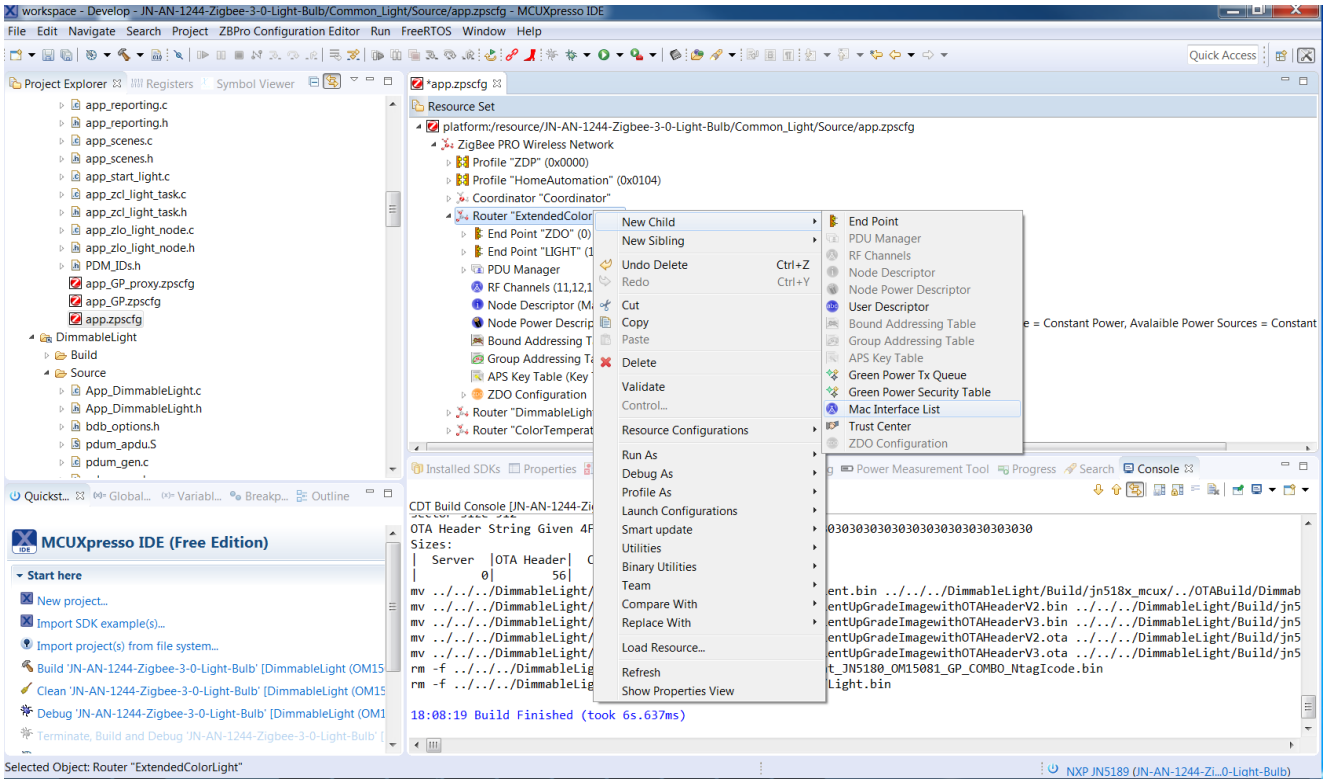
The sequence of parameters passed to the flash programmer are important. When programming the device care should be taken to have the -e option before the -p option.

Help on various options supported by the standalone command line flash programmer application is available through the -h or --help.

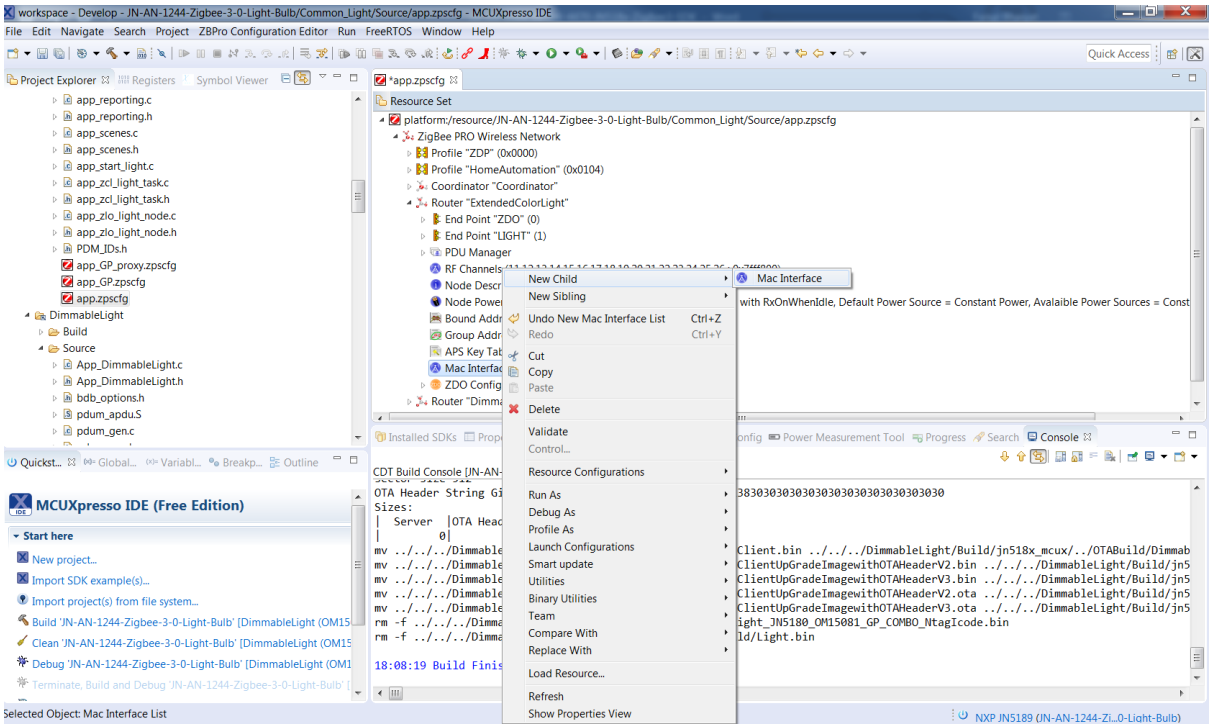
4.2.3 Porting to R22 stack:

The Zigbee PRO R22 version of the stack allows for multiple MAC interfaces to be present. This is to support both 2.4G and 868 MHz frequency bands using the single Zigbee stack. To address this a MAC interface table needs to be configured in the ZPS Config diagram.

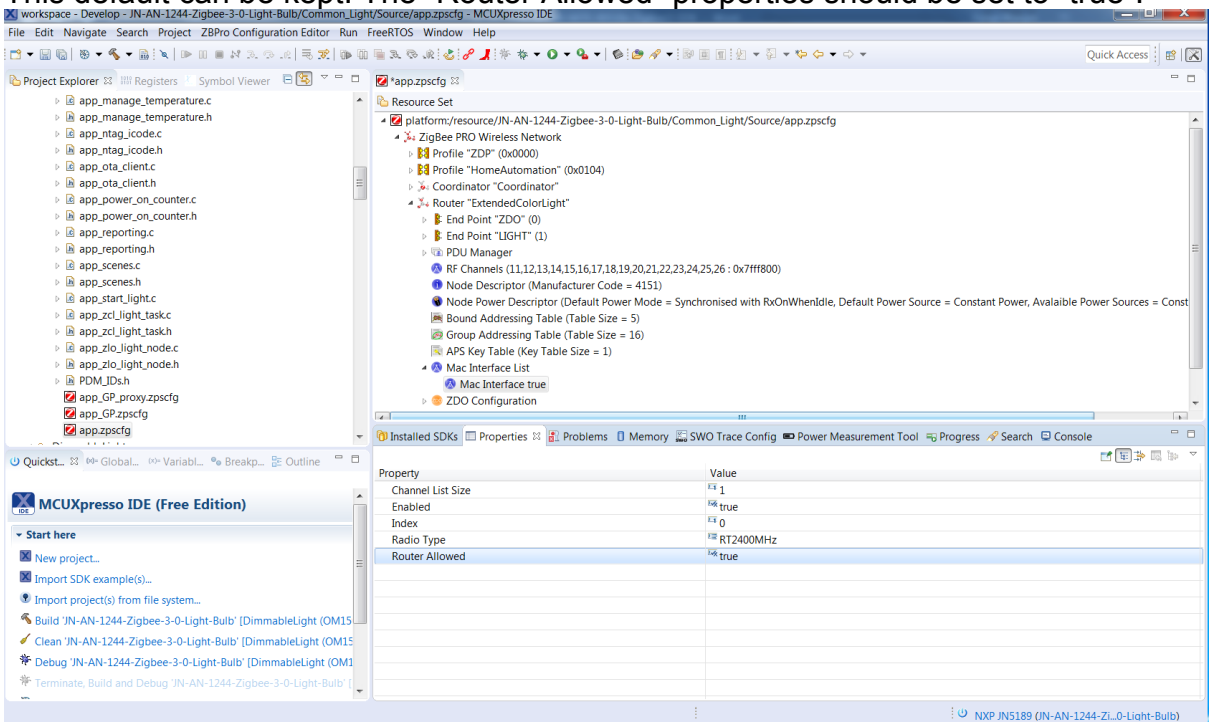
The Mac Interface list can be found as an option for the node. For e.g. if you have Zigbee network with a router node. You can select the router node and press the right mouse button to provide the options. The Mac Interface list can be found under New Child -> Mac Interface List.



After adding the Mac Interface List, select the Mac Interface list and press the right mouse button to provide the options. The Mac interface can be found under New Child -> Mac Interface.



After adding the Mac Interface, the properties can be updated. The default is 2.4G. This default can be kept. The “Router Allowed” properties should be set to “true”.



4.2.4 OTA configuration:

OTA cluster is enabled through the ZCL_options.h file. Please refer to the NXP Zigbee 3.0 clusters user guide (JN-UG-3115) for more details on this.

The OTA cluster requires initialisation of the location where the upgrade image can be stored. The application provides this through eOTA_AllocateEndpointOTASpace API.

Each Page on JN518x is 512 bytes, Usable flash size is 632K. Allowing for 32K for PDM (start page 1152) and 24K for customer data, leaves a usable flash size for image at 576K. If we Split it into 2 sections to support OTA. It means 288K becomes max image size. Each 288K section would be 576 flash pages. This could be represented as 32K sectors to keep in line with legacy devices.

So, for allocation to the OTA cluster:

uint8 u8MaxSectorPerImage = 0;

uint8 u8StartSector[1] = {9}; /* So next image starts at 9*32*1024 = 288K offset*/

u8MaxSectorPerImage = 9 ; /* 9 *32* 1024 = 288K is the maximum size of the image */

sNvmDefs.u32SectorSize = 512; /* Sector Size = 512 bytes*/

The OTA checks for the presence of the well-known Zigbee09 key at a fixed location within the image. This provides a convenient mechanism to test the decryption of an encrypted image and an additional sanity check to make sure the image is a valid image to progress downloading. Please note that this is not the mechanism for a full image validation. It is recommended that for OTA an encrypted image with the CRC check be used. That provides a better validation of an OTA image.

Each application note has an OTA_BUILD folder which holds the OTA compatible images.

LinkKey_3.txt is required for creation of the OTA image. That holds the Key which can be used for validation purpose as described above. There are configOTA_JN518x_Cer_Keys_HA_Light.txt and configOTA_JN518x_Cer_Keys_HA_Light_Generic.txt which provide the OTA image generator with the offset for the key.

Prior to this release the values were LinkKey_3.txt,02c0,16

This must now be LinkKey_3.txt,01b0,16

This release of the SDK provides the ability to change the default key used for encryption.

The upgrade image MUST be encrypted with the same key as that set in the currently running image. To change the keys for a build, the running image must be physically programmed using the flash programmer and then subsequent upgrade images can be built with the new key and OTA can start.

To set the encryption key the Makfile must be updated as follows:

Change:

```
ifeq ($(OTA_ENCRYPTED),1)
CFLAGS  += -DOTA_ENCRYPTED
Endif
```

To:

```
ifeq ($(OTA_ENCRYPTED),1)
CFLAGS  += -DOTA_ENCRYPTED
ENCRYPTION_KEY = 1234567890ABCDEFA1B2C3D4E5F6F1B4
CFLAGS += -DCLD_OTA_KEY=\"$(ENCRYPTION_KEY)\"
endif
```

The key used internally within OTA must also be passed to the JET tool.

The following option is added:

```
$(DEV_BLD_DIR)/../OTABuild/LightCreateOtaEncClient.bat
$(DEV_BLD_DIR)/../OTABuild $(JET_BASE) $(MANUFACTURER_CODE)
$(OTA_STRING) $(JET_VERSION) $(JENNIC_CHIP_FAMILY)
$(OTA_DEVICE_ID) $(ENCRYPTION_KEY)
```

In the *.bat which passes the command line options to the JET tool must also be updated, for e.g:

```
$2/JET.exe -m bin -f Client.bin -e Client_Enc.bin -k 0xffffffffffffffffffffffffffffffff -i
0x00000010111213141516171800000000 -v $5 -j $4 --sector_size=512
```

The -k option should be changed to:

```
$2/JET.exe -m bin -f Client.bin -e Client_Enc.bin -k $8 -i
0x00000010111213141516171800000000 -v $5 -j $4 --sector_size=512
```

4.2.5 Application specific updates

4.2.5.1 Porting from ES1 to ES2:

All reference to

```
PMC->CTRL |= PMC_CTRL_PWRUP_ZIGBLE(1);
   while((PMC->PWRSWACK & PMC_PWRSWACK_PDZIGBLE(1)) != 1);
```

should be removed.

```
FIREWALL->UPDATE_VALUE = 0x80000005;
FIREWALL->UPDATE_SETTING = 0x7 << 8;
```

And any other references to FIREWALL should also be removed.

4.2.5.2 General Application notes updates:

The watchdog clocks are not enabled by default. These need to be enabled by calling

```
/* WWDT clock config (32k oscillator, no division) */
CLOCK_AttachClk(kOSC32K_to_WDT_CLK);
CLOCK_SetClkDiv(kCLOCK_DivWdtClk, 1, true);
```

In *void BOARD_InitClocks(void)*

To debug the various exceptions *vDebugExceptionHandlersInitialise();*

Must be called in *void vAppMain(void)*

The SDK drivers do not set the priority of the various peripheral interrupts. These need to be set in the application.

These should be done after all the hardware initialisation. These can be added to

void APP_vSetUpHardware(void)

```
#define APP_BASE_INTERRUPT_PRIORITY  (5)
#define APP_WATCHDOG_PRIOIRTY        (1)

int   iAppInterrupt;
for (   iAppInterrupt = DMA_IRQn;
  iAppInterrupt < SHA_IRQn;
  iAppInterrupt ++ )
{
    NVIC_SetPriority( iAppInterrupt, APP_BASE_INTERRUPT_PRIORITY );
}
NVIC_SetPriority( WDT_BOD_IRQn, APP_WATCHDOG_PRIOIRTY );
```

The transition time parameter is now added for recall scene command. Please refer to the ZCL specification for details.

4.2.5.3 RAM saving on End Devices:

The application notes can use the following configuration to optimise the amount of RAM they use.

There are number of routing configurations which are present in the ZPS configuration editor. These are however not applicable for end devices.

The tool ignores the settings for end device automatically. It is not possible to set these configuration to 0, so they can be left at 1.

ActiveNeighbourTableSize="2"

RouteDiscoveryTableSize="1"

RoutingTableSize="1"

BroadcastTransactionTableSize="2"

RouteRecordTableSize="1"

MacTableSize="16" [Assuming Address map table of 14, This should be sized to Neighbour table + Address map table]

NumNPDUs="9"

apduZDP Instances="5" [Depends on application and may be reduced further]

apduZCL Instances="5" [Depends on application and may be reduced further]

BindingTable Size="16" [Depends on application and may be reduced further]

In addition to these there are number of queues which are configured for the stack.

```
#define MLME_QUEQUE_SIZE          4
#define MCPS_QUEUE_SIZE           2
#define MCPS_DCFM_QUEUE_SIZE      4
#define APP_QUEUE_SIZE            4
```

```
/* Stop clock to SRAM1; saves 10uA – Only do this if not using the second memory bank.*/
   SYSCON->AHBCLKCTRLCLR[0] = SYSCON_AHBCLKCTRLCLR0_SRAM_CTRL1_CLK_CLR_MASK;
```

4.2.5.4 Minimum RAM retention during sleep

WARNING:

It is possible to mark application data as being safe to discard during sleep with RAM held, by placing it into sections “.bss.discard.app” or “.data.discard.app” when using the minimum RAM retention feature. This data is also discarded when using the AppBuild16kEndDevice.ld linker script. The discarded data is also re-initialised on warm start: data in “ .bss.discard.app” is set to zero and data in “.data.discard.app” is reset to its initial value.

This SDK provides support for a feature to reduce the amount of RAM retention to minimum of 4Kb during sleep with RAM held.

The method of determining the RAM usage is to look at the _vStackTop variable in the map file.
e.g.

0x040151e8 PROVIDE (_vStackTop, DEFINED (__user_stack_top)?__user_stack_top:(((((__top_RAM0 - SIZEOF (.data)) - SIZEOF (.bss)) - SIZEOF (.heap)) - 0x20) & 0xfffffff8))

Top of RAM bank 0 is
__top_RAM0 = 0x4000400 + 0x15c00; /* 87K bytes */ = 0x4016000

If you subtract the two = 0x4016000 - 0x040151e8 = 0xE18 = 3608 bytes or ~3.5K. The processor stack grows and shrinks below _vStackTop as the application runs, growing as functions or interrupt handlers are called and shrinking when they are exited. At the point that the device goes to sleep, from the application's idle loop, the active part of the processor stack will be very small but it will not be 0. This active part of the processor stack is also retained through sleep, so it is important that the retained data + the retained processor stack is less than 4kB. With the JN-AN-1245 dimmer switch I measured the retained processor stack as 72 bytes, but customer applications may be different.

As the value varies from one application to another, it is easiest to read it back from a running device by looking at the PWRM's internal variable that it uses to decide which RAM banks to retain. To do this, after warm start:

```
extern volatile uint8 *s_pu8Stack;
DBG_vPrintf(TRUE, "Retained RAM low water mark:%x\n",
(uint32)s_pu8Stack);
```

You need the value to be at least 0x04015000 for 4kB retention or at least 0x04014000 for 8kB retention.

This release includes timing optimisations for the radio driver. To support these, the application should:

1. Provide bRadioCB_WriteNVM() and u16RadioCB_ReadNVM() functions, as provided in latest JN-AN-1245 Common_Switch\Source\app_main.c file. This allows the radio driver to store calibration settings in PDM so that it does not have to re-calculate them on cold starts. Without this, or if PDM has been erased, cold start will be 55.6ms longer because the radio driver will have to perform a full calibration
2. After calling (E_AHI_SLEEP_OSCON_RAMON), call PWRM_vForceRadioRetention(TRUE); This causes the radio retention registers to be powered through sleep. Without this, warm start will be 4.72ms longer because the radio driver will have to perform a partial recalibration

To build an application with minimum RAM retention, RAMOPT=1 build flag must be set.

The OTA cluster data is retained during sleep but none of the other ZCL cluster data is retained.

Before the device goes to sleep, in the presleep callback function, it is required that the OTA data is copied across from the cluster context structure into the persisted data structure. This functionality is provided through the vSetOTAPersistedDatForMinRetention and updated for the dimmer switch application note.

e.g. For the dimmer switch application

```
    #ifdef DimmerSwitch
       extern tsZLO_DimmerSwitchDevice sSwitch;
       memcpy( &sOTA_PersistedData,

&sSwitch.sCLD_OTA_CustomDataStruct.sOTACallBackMessage.sPersistedData,
            sizeof(tsOTA_PersistedData));
    #endif
```

When the device comes out of sleep and the *wakeup* callback function is called, it needs to re-initialise all the data not retained during sleep.

The following needs to be done:

1. All hardware needs to be re-initialised. This is normally implemented through the `APP_vSetUpHardware();` API
2. Base device functionality must be restarted using the `BDB_vRestart();`
3. Any application data which has not be retained should be initialised. By default all application data is retained and doesn't need re-initialising.
4. Following APIs must be called to re-initialise the stack components and buffers

   ```
   PDUM_vInit();
   ZPS_vExtendedStatusSetCallback(vfExtendedStatusCallBack);

   ZPS_eAplAfReInit();
   ```

5. The PDM and power manager should not be re-initialised.
6. All other functions like the ZCL and BDB initialisation should be done as it would be during a cold start.
7. When using OTA the following needs to be done:

```
      #if (defined SLEEP_MIN_RETENTION) && (defined CLD_OTA) &&
(defined OTA_CLIENT)
        vZCL_SetUTCTime(U32UTCTimeBeforeSleep+1);
         bInitialiseOTAClusterAndAttributes();
         /*ZCL time data is not retained so , OTA is. This make OTA
think ti's registered while
          * it isn't.
          */
         extern  void vOtaTimerClickCallback(
                 tsZCL_CallBackEvent         *psCallBackEvent);
         if(eZCL_TimerRegister(E_ZCL_TIMER_CLICK_MS, 0,
vOtaTimerClickCallback)!= E_ZCL_SUCCESS)
         {
             DBG_vPrintf(TRACE_SWITCH_TASK, "Failed to register the
timer\n");
         }

           #endif
```

8. The ZCL needs to be aligned with the amount of time the device has been asleep.

```
tsZCL_CallBackEvent sCallBackEvent;

sCallBackEvent.eEventType = E_ZCL_CBET_TIMER;
vZCL_EventHandler(&sCallBackEvent);
```



```
#ifdef CLD_OTA
    vRunAppOTAStateMachine(1000);
#endif
```

4.2.5.5 Selective OTA

The current OTA mechanism relies on transfer and saving of the entire image (application + stack) in the flash of the JN518x device. This requires the entire flash to be split up to accommodate two images, the running image and the new upgrade image.

Often these images are large and hence can take a long time to download. This has an impact on the power usage of a device. It is desirable to often update only the stack or just the application. Selective OTA is a NXP specific implementation to divide a running image into two self-contained parts. These are referred to as APP0 and APP1.

The design decision is to split the image on functional basis. APP0 is the application image, it consists of application specific code, BDB, ZCL, debug library, Zigbee common components like ZQueue, ZTimer, beacon filtering, power manager and peripheral driver code.

APP1 is the stack image, it consists of the ZPSNWK, ZPSAPL, ZPSMAC, MAC, PDM, PDUM, Radio components, ZPSTSV and random number generator.

For selective OTA APP0 and APP1 can be upgraded independent of each other. The memory is now split into 4 parts, APP0 active and upgrade image, APP1 active and upgrade image. The OTA cluster downloads these sequentially, so at any given time it’s only downloading either APP0 or APP1.

The maximum size for APP0 is 160K bytes and the maximum size for APP1 is 136K bytes.

APP1 is provided as a prebuilt un-encrypted binary for a routing device and end device.

There are two variants of APP1 image. There is an end device and routing device variant.

These can be found under:

C:\NXP\mcux\JN-SW-4470\middleware\wireless\zigbee3.0\ZigbeeCommon\SelectiveOtaApp1\JN518x_mcux\

There are the Release (this is the ZCR build) and ReleaseEndDevice (this is the end device build) folders.

Selective OTA feature assumes that the images, both application and stack image would be encrypted. Separate batch files (SelectiveOtaImageGen.bat) are provided to add specific OTA headers like strings, version number etc. to these. The batch scripts do not need any update, they assume some default OTA header components and can be used without change, care should be taken to make sure that the upgrade image has a higher version number than the current running image. The

batch files use the JET tool to create the programmable images and, also encrypts the upgraded image.

The default key in the SelectiveOtaImageGen.bat may be set to 0xffffffffffffffffffffffffffffffff so it will not accept any other key apart from the default. This can be changed by changing the -k 0xffffffffffffffffffffffffffffffff in the batch file and changing it to -k $8.

WARNING:

The APP0 and APP1 key MUST be kept in sync.

This should not be changed between the application and stack. Both application and stack must have the same encryption key. Currently this key cannot be changed. This will be available in subsequent releases.

The Makefiles can call the batch file and pass the OTA header information as follows:

4.2.5.5.1 Building the application APP0 image

The Makefile needs to be updated so that it uses the right linker command scirpts for selective OTA.

To build the APP0 image all references to “AppBuildNone.ld” should be changed to “AppBuildNone_APP0.ld”. To use the same Makefile to switch between different builds it would be advisable to do as suggested below.

```
ifeq ($(SELOTA),APP0)
SETIMAGESIZE = 163840
LNKCMD = AppBuildNone_APP0.ld
else
SETIMAGESIZE = 294912
LNKCMD = AppBuildNone.ld
endif
```

If it’s implemented as above the following change also needs to be done:

```
$(TOOLCHAIN_PATH)/$(CC)  -Wl,--gc-sections $(LDFLAGS) -T "AppBuildNone.ld" -T "jn5180_rom.ld"
```

To:

```
$(TOOLCHAIN_PATH)/$(CC)  -Wl,--gc-sections $(LDFLAGS) -T  $(LNKCMD) -T "jn5180_rom.ld"
```

When using the selective OTA the maximum application image size needs to be passed to the signing tool.

This is passed to the signing tool via -s option and needs to be done in addition to the above change of Makefile

```
jn518x_image_tool.py -s $(SETIMAGESIZE)
```

```
ifeq ($(OS),Windows_NT)
        $(DEV_BLD_DIR)/../OTABuild/LightCreateOtaEncClient.bat $(DEV_BLD_DIR)/../OTABuild $(JET_BASE) $(MANUFACTURER_CODE) $(OTA_STRING) $(JET_VERSION) $(JENNIC_CHIP_FAMILY) $(OTA_DEVICE_ID)
ifeq ($(SELOTA),APP0)
```

```
		$(NXP_OTA_APP)/SelectiveOtaImageGen.bat $(NXP_OTA_APP)
../../$(JET_BASE) $(NXP_MANUFACTURER_CODE) $(NXP_STRING)
$(JET_VERSION) $(JENNIC_CHIP_FAMILY) $(NXP_STACK_OTA_DEVICE_ID)
endif
else
```

This should only be done for ifeq ($(OTA_ENCRYPTED),1)

Selective OTA is supported only for encrypted OTA builds.

To build the selective OTA the following options should be added to the build configuration line

OTA_ENCRYPTED=1 SELOTA=APP0

4.2.5.5.1.1 Code changes in the application to build APP0

Changes to the zcl_options.h file:

The following OTA specific definitions must be provided for OTA cluster

```
#ifdef APP0
   #define OTA_APP1_ACTIVE_FLASH_OFFSET              (uint32)(456 * 1024) /*
320K + 136K Logical location for APP1 active image is present */
   #define OTA_APP1_SHADOW_FLASH_OFFSET              (uint32)(320 * 1024) /*
320K   Logical location for APP1 upgrade image to be stored*/
   #define OTA_APP1_MAX_SIZE                         (uint32)(136 * 1024)  /* Max size
in bytes must be in multiples of 512 bytes*/
#endif
```

Changes to app_main.c file:

Include the following headers:

```
#include "board.h"
#include "fsl_debug_console.h"
#include "app.h"
#include "Selective_OTA.h"
```

Provide reference to jump table for APP1 as below:

```
extern uint32 *pu32App1JumpTable;
```


In the void APP_vSetUpHardware(void) function after the BOARD_InitHardware(); add

```
    /* Look for App1 to run, populating the jump table pointer if found */
    if (E_SOTA_IMAGE_NOT_FOUND != eSOTA_FindImage(&pu32App1JumpTable))
    {
                    /* Call reset for App1. This allows App1 to initialise itself */
                    vSOTA_ImageReset();
    }
```

In the app_ota_client.c file , in the PUBLIC void vAppInitOTA(void)

Change following

```
#if (defined OTA_INTERNAL_STORAGE)

    sNvmDefs.u8FlashDeviceType = E_FL_CHIP_INTERNAL;
#if (defined JENNIC_CHIP_FAMILY_JN516x) || (defined JENNIC_CHIP_FAMILY_JN517x)
    uint8 u8StartSector[1] = {8};
    u8MaxSectorPerImage = 8 ;
    sNvmDefs.u32SectorSize = 32*1024; /* Sector Size = 32K*/
#else
    /* Each Page on JN518x is 512 bytes , Flash size is 632K. Taking into account 31.5K for PDM (start page 1153) and 24K for customer data
     * usable for image is 576K
     * Split it into 2 sections to support OTA, so 288K becomes Max image size
     * Each 288K section would be 576 pages. This could be represented as 32K sectors to keep in line with legacy devices.*/
    uint8 u8StartSector[1] = {9}; /* So next image starts at 9*32*1024 = 288K offset*/
    u8MaxSectorPerImage = 9 ;  /* 9 *32* 1024 = 288K is the maximum size of the image */
    sNvmDefs.u32SectorSize = 512; /* Sector Size = 512 bytes*/
#endif
```

To:

```
#if (defined OTA_INTERNAL_STORAGE)

    sNvmDefs.u8FlashDeviceType = E_FL_CHIP_INTERNAL;
```



```
#if (defined JENNIC_CHIP_FAMILY_JN516x) || (defined
JENNIC_CHIP_FAMILY_JN517x)
    uint8 u8StartSector[1] = {8};
    u8MaxSectorPerImage = 8 ;
    sNvmDefs.u32SectorSize = 32*1024; /* Sector Size = 32K*/
#else


#ifdef APP0


    /* Each Page on JN518x is 512 bytes , Flash size is 632K. Taking into account
31.5K for PDM (start page 1184 - move this by 16
     * to accommodate the customer data  ) and 8K for customer data
     * usable for image is 592K - APP0 + APP1  APP0 size is max of 160K , APP1 size
is max of 136K.*/
    uint8 u8StartSector[1] = {5}; /* So next image starts at 5*32*1024 = 160K offset*/
    u8MaxSectorPerImage = 5 ;  /* 5 *32* 1024 = 160K is the maximum size of the
image */
    sNvmDefs.u32SectorSize = 512; /* Sector Size = 512 bytes*/
#else
    /* Each Page on JN518x is 512 bytes , Flash size is 632K. Taking into account
31.5K for PDM (start page 1153) and 24K for customer data
     * usable for image is 576K
     * Split it into 2 sections to support OTA, so 288K becomes Max image size
     * Each 288K section would be 576 pages. This could be represented as 32K
sectors to keep in line with legacy devices.*/
    uint8 u8StartSector[1] = {9}; /* So next image starts at 9*32*1024 = 288K offset*/
    u8MaxSectorPerImage = 9 ;  /* 9 *32* 1024 = 288K is the maximum size of the
image */
    sNvmDefs.u32SectorSize = 512; /* Sector Size = 512 bytes*/
#endif
#endif
```

In the PUBLIC void vHandleAppOtaClient(tsOTA_CallBackMessage *psCallBackMessage)

Change:

```
else {
            /* down load about to start */
            vStartEffect(E_CLD_IDENTIFY_EFFECT_BREATHE);
```



```
            eOTA_State = OTA_DL_PROGRESS;
            u32OTAQueryTimeinSec = 0;
            u32TimeOut = 0;
            DBG_vPrintf(OTA_LNT, "Accept Image\n");
        }
```

To:

```
else {
            /* down load about to start */
            vStartEffect(E_CLD_IDENTIFY_EFFECT_BREATHE);
            eOTA_State = OTA_DL_PROGRESS;
            u32OTAQueryTimeinSec = 0;
            u32TimeOut = 0;
            DBG_vPrintf(OTA_LNT, "Accept Image\n");
            vOTAPersist();
        }
```

In the PRIVATE uint8 u8VerifyLinkKey(tsOTA_CallBackMessage *psCallBackMessage)

Change:

```
#ifdef OTA_INTERNAL_STORAGE
#if (defined JENNIC_CHIP_FAMILY_JN516x) || (defined JENNIC_CHIP_FAMILY_JN517x)
        u32LnkKeyLocation += psCallBackMessage->u8ImageStartSector[0] * sNvmDefsStruct.u32SectorSize;
#else
        /* For JN518x a segment has the same definition as a Page in this implementation
            1 Page = 32 FLASH words = 32 * 16 = 512 Bytes AHI uses 32K sector sizes
          (32*1024)/512 gives a factor of 64 */
        u32LnkKeyLocation += sNvmDefsStruct.u32SectorSize * psCallBackMessage->u8ImageStartSector[0]  * 64;
#endif
#endif
```

To:

```
#ifdef OTA_INTERNAL_STORAGE
```

```
#if (defined JENNIC_CHIP_FAMILY_JN516x) || (defined
JENNIC_CHIP_FAMILY_JN517x)
        u32LnkKeyLocation += psCallBackMessage->u8ImageStartSector[0] *
sNvmDefsStruct.u32SectorSize;
#else

#ifdef APP0
        if(psCallBackMessage->sPersistedData.bStackDownloadActive)
        {
            u32LnkKeyLocation += OTA_APP1_SHADOW_FLASH_OFFSET;
        }
        else
#endif
        {
            /* For JN518x a segment has the same definition as a Page in this
implementation
            1 Page = 32 FLASH words = 32 * 16 = 512 Bytes AHI uses 32K sector sizes
            (32*1024)/512 gives a factor of 64 */
            u32LnkKeyLocation += sNvmDefsStruct.u32SectorSize * psCallBackMessage-
>u8ImageStartSector[0]  * 64;
        }
#endif
#endif
```

In the PRIVATE void vManagaeOTAState(void)

Change:

```
uint8 u8SrcEp = app_u8GetDeviceEndpoint();
```

To:

```
uint8 u8SrcEp = app_u8GetDeviceEndpoint();
#ifdef APP0 /* Building with selective OTA */
    static bool_t bGetApp1ImageRequest = FALSE;
#endif
```

Change:

```
if(u32OTAQueryTimeinSec > OTA_IMAGE_QUERY_TIME_IN_SEC )
            {
                if(sZllState.bValid)
```



```
        {

                tsOTA_ImageHeader       sOTAHeader;
```

To:

```
if(u32OTAQueryTimeinSec > OTA_IMAGE_QUERY_TIME_IN_SEC )
        {
            if(sZllState.bValid)
            {

                tsOTA_ImageHeader       sOTAHeader;
#ifdef APP0 /* Building with selective OTA */
                if(bGetApp1ImageRequest)
                {
                    bGetApp1ImageRequest = FALSE;
                    vOTA_SetApp1OtaEnable(sZllState.u8OTAserverEP, TRUE);
                }
                else
                {
                    bGetApp1ImageRequest = TRUE;
                    vOTA_SetApp1OtaEnable(sZllState.u8OTAserverEP, FALSE);
                }
#endif
```

All references to `PDM_eInitialise(1152, 63, NULL);`

Should be changed to `PDM_eInitialise(1184, 63, NULL);`

The persistent data must be fully erased when using selective OTA image for the first time.

Once the OTA enabled image is built it should be programmed as normal using the flash programmer

./JN518xProgrammer.exe -V 2 -P 115200 -s COM56 -e FLASH -p ”C:\<image>”

There are two images available for APP1 depending on the build type.
These images are created when the application note is built. They can also be created by calling the batch file directly with the appropriate parameters.

```
# $1 : Path of files.
# $2 : Jet Base dir
```


$3 : the manufacturer code
$4 : 32 byte OTA header stack string
$5 : JET VERSION 6 JN518x
$6 : Jennic chip family like JN518x
$7 : NXP OTA Device Id
$8 : NXP encryption key

e.g.
Assuming the strings have been defined beforehand in the Makefile and the batch file has been updated to change the -k 0xffffffffffffffffffffffffffffffff to -k $8 then:

SelectiveOtaImageGen.bat $(NXP_OTA_APP) $(JET_BASE) $(NXP_MANUFACTURER_CODE) $(NXP_STRING) $(JET_VERSION) $(JENNIC_CHIP_FAMILY) $(NXP_STACK_OTA_DEVICE_ID) $(ENCRYPTION_KEY)

If the SelectiveOtaImageGen.bat has not been updated to accept a different key to what's hardcoded then the $(ENCRYPTION_KEY) shall be omitted.

If building for a routing device look at the location C:\NXP\mcux\JN-SW-4470\middleware\wireless\zigbee3.0\ZigbeeCommon\SelectiveOtaApp1\JN518x_mcux\Release

The file ZpsSelectiveOtaApp1.bin should be programmed.

If building for an end device look at the location C:\NXP\mcux\JN-SW-4470\middleware\wireless\zigbee3.0\ZigbeeCommon\SelectiveOtaApp1\JN518x_mcux\ReleaseEndDevice

The file ZpsSelectiveOtaApp1_ZED.bin should be programmed.

The production flash programmer can be used to program this image with the following option:

./JN518xProgrammer.exe -V 2 -P 115200 -s COM56 -p FLASH@0x50000=" C:\NXP\mcux\JN-SW-4470\middleware\wireless\zigbee3.0\ZigbeeCommon\SelectiveOtaApp1\JN518x_mcux\Release\ ZpsSelectiveOtaApp1.bin"

Care must be taken when programming the APP0 and APP1. If APP0 is programmed with the erase flash option, it will also remove any previously programmed APP1. So, it will require reprogramming the APP1.

The encrypted image which can then be hosted on an OTA server has the .ota extension at the location of APP1.

4.3 Changes from v1811 release

The following issues have been fixed in this release:

Internal ID	Description
artf630276	Stack doesn’t give the ZDP callback on received ZDP requests.
artf625125	PWRM has been updated to use the BOD wakeup and an interrupt protection guard is put for the increment and decrement of activity counter.
MCUZIGBEE-1618	Suppress BDB rejoin when restarting to improve startup time
MCUZIGBEE-1616	ZigBee initialisation to remove MLME-Reset.Req after MAC initialisation
MCUZIGBEE-1431	Allow MAC buffers to be OFF in sleep mode with/without RAM held
MCUZIGBEE-1553	Update coverity_scan script to use repo scripts
MCUZIGBEE-483	Add califro32k.bin
KPSDK-21266	Remove POWER_SLEEP mode
MCUZIGBEE-483	Fix calibraterfo32k generator -r issue
MCUZIGBEE-483	update frequency type from uint32 to uint64 add demo calibratefro32k to show fro32k calibration using 32m clock add new api to support reverse target and reference clock
MCUZIGBEE-1630	Minimize the delay time in ADC_Configuration function
MCUZIGBEE-1553	Update coverity APP_NOTES script
KPSDK-21267	Include pin_mux.h to fix build warning
KPSDK-21293	Add get INISTAT register driver api.
MCUZIGBEE-1553	Add NFC coverity scan
MCUZIGBEE-1446	sync for fro1m trim update
MCUZIGBEE-1609	GPP Functionality Attribute doesn't have the right bitmap
artf571504	NXP ZCL code has not implemented transitiontime parameter as part of Recall Scene command
MCUZIGBEE-1610	GP frame counter is wrong
MCUZIGBEE-1611	GP doesn't handle reserved and manufacturer specific commands
MCUZIGBEE-1612	Zigbee stack doesn't default back to not using install codes
MCUZIGBEE-1607	do not switch off APB bridge clock when disabling the Temperature sensor
MCUZIGBEE-1613	TSV timer context structures don’t get initialised

MCUZIGBEE-1234	**Clock divider glitch safe procedure 1.update the api named CLOCK_SetClkDiv in devices\JN5180\fsl_clock.c**
MCUZIGBEE-1587	**ADC SDK driver improvements/bugs follow-up**
MCUZIGBEE-1587	**[ADC] Data output not shifted when resolution below 12 bits**

4.4 Known Issues

None.

5. Related Documentation

The following user documentation supports this software release:

- ZigBee 3.0 Stack User Guide [JN-UG-3113]
- ZigBee 3.0 Devices User Guide [JN-UG-3114]
- ZigBee Cluster Library (for ZigBee 3.0) User Guide [JN-UG-3115]
- ZigBee Green Power (for ZigBee 3.0) User Guide [JN-UG-3119]
- JN51xx Core Utilities User Guide [JN-UG-3116]
- JN51xx Production Flash Programmer User Guide [JN-UG-3099]
- JN518x SDK API Reference Manual (replacing the JN51xx Integrated Peripheral User Guide) [MCUXSDKJN518xAPIRM]

All the above manuals are available on request from the NXP Applications Team.