NXP Semiconductors
User's Guide

Document Number: KTSADUG
Rev. 7, 08 April 2019

Kinetis Thread Stack

Application Developer’s Guide

Contents

Chapter 1
Introduction

This document explains how to start developing a Thread IPv6 mesh wireless protocol application.

The document also describes the configuration and initialization of the stack as performed by the application, followed by the presentation of API calls for specific use cases.

The following demo application modules related to performing stack operations are described in detail.

- Scanning for networks
- Joining initiation
- Commissioning parameters
- Selecting a Parent
- Inspecting IP address configuration
- Access the IP data plane via Sockets and CoAP sessions
- Low-power-specific configuration

Chapter 2
Example Applications

The Thread stack is provided along with IAR® Embedded Workbench and MCUXpresso example applications, which are useful starting points for application developers before they modify and enhance. The example application are templates for the main categories of Thread devices:

- **Router Eligible Device** – a standalone Thread wireless node which has mesh routing capabilities and also can act as a Parent, forwarding, buffering data and accomplishing network management functionality on behalf of attached child End Devices
- **End Device** – a standalone Thread wireless node which is always powered on with radio receiving, however may not have on-board resources to buffer and forward routed data in the wireless mesh or by acting as a Parent
- **Low Power End Device** – is an application template similar to the End Device with the distinguishing characteristic that the device is preconfigured to have both the microcontroller in low power state and the radio turned off for most of the device life cycle; the device “wakes up” periodically at a few seconds interval and actively polls its Parent Router for data addressed to it or optionally initiates sending data to the network by means of the Parent Router
- **Border Router** – when deployed, the IP stack is enabled to forward messages between the Thread IEEE® 802.15.4/6LoWPAN wireless mesh interface to the auxiliary interface, over the Thread network border; messages can thus reach upstream infrastructure, such as a standard home Local Area Network (LAN) or the Internet and cloud servers
- **Host Controlled Device** – an application implementing an expansive full featured, Router capable configuration of the stack in conjunction with activating a UART/SPI/USB peripheral which carries over the Thread Host Control Interface (THCI) protocol frames in order to facilitate scenarios where the Thread stack microcontroller is driven by a host application processor or a tool

The application share a similar high-level toolchain project structure being differentiated by the starting point application file and stack and system configuration.

For more details on the example applications, see the *Kinetis Thread Stack Demo Applications User’s Guide* (document KTSDAUG).

Chapter 3
Application Architecture

3.1 Overview

This section describes the common application structure and architecture which is implemented by the Kinetis Thread Stack example applications and is a recommended as a template/starting point for new applications.

The Router Eligible Device project is used as a base example and the other examples only when referring to their differentiating features.

Application developers are advised to choose an example template that matches closer to their application. However, when testing or deploying any device, a router capable device is required to bootstrap a new network entity and let other devices, including standalone end devices join the network.

3.2 IAR Embedded Workbench application project structure

The base example for a Router Eligible Device can be accessed and loaded into IAR Embedded Workbench version 7.80 or later from path \boards\frdmkw41z\wireless_examples\thread\router_eligible_device\freertos\iar. The frdmkw41z configuration corresponding to the FRDM-KW41Z development platform is used as a base example. In the downstream subfolders, projects are provided for FreeRTOS OS. The FreeRTOS OS project is used as the base for the examples below. The workspace example is:

\boards\frdmkw41z\wireless_examples\thread\router_eligible_device\freertos\iar\ router_eligible_device.eww

Launching the workspace file into the IAR EWARM tool shows the Project and individual file group and file organization in the Workspace panel on the left side of the window.

The Thread application workspace includes a standalone IAR project:

- **router_eligible_device** – the main application project, configured to output a binary S-Record (*.srec) file when built which can be loaded to run on device firmware

The following figure shows an expansion of the main application project file group structure along with the functionalities for the file in each group.

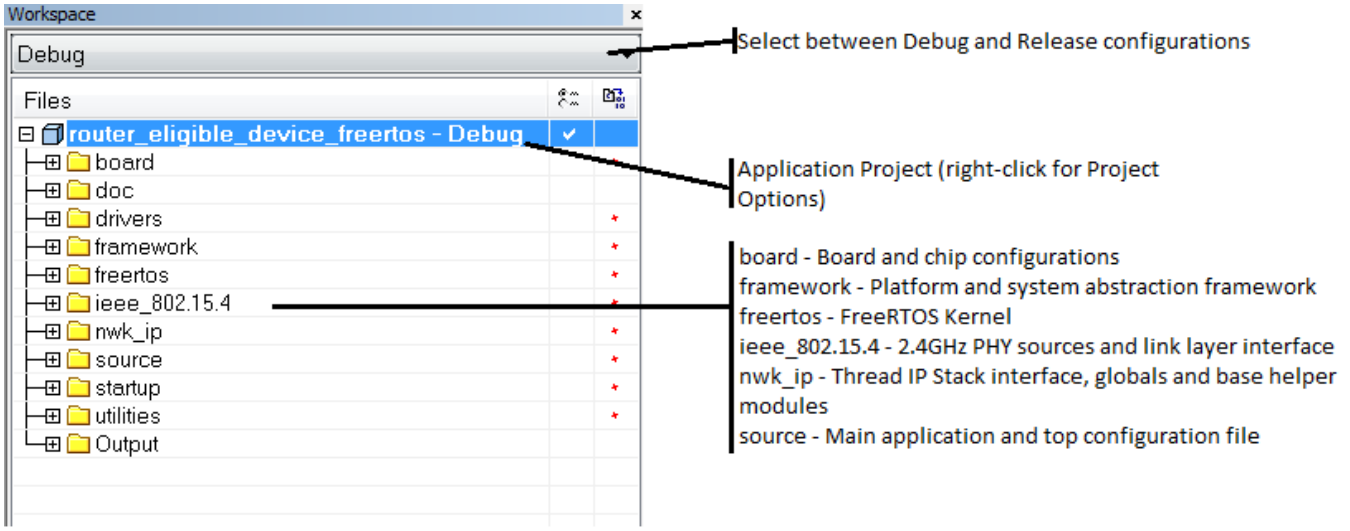

Figure 1. IAR Application Workspace

3.3 IAR Embedded Workbench compile time configuration

The compile time configuration for a Thread stack application is managed through a set of IAR EWARM and C module constructs, primarily preprocessor (macro) definitions.

3.3.1 IAR Embedded Workbench project options configuration

The configuration settings are primarily contained in the **IAR Project Options panel** and the *.h header files contained in **\middleware\wireless\nwk_ip_1.3.1\examples\common** and **\middleware\wireless\nwk_ip_1.3.1\examples\<application>\config** file groups within a project.

To inspect the configuration options set within the IAR Project Options panel right-click the application project as indicated in Figure 1.

The figures below shows the global Preprocessor options. These include:

- Setting the file at **\middleware\wireless\nwk_ip_1.2.8\examples\<application>\config** as a **Preinclude file** which is included by default whenever a source module compilation is invoked
- Preprocessor **Defined symbols** which consist in the main global settings for the system and stack for the application project

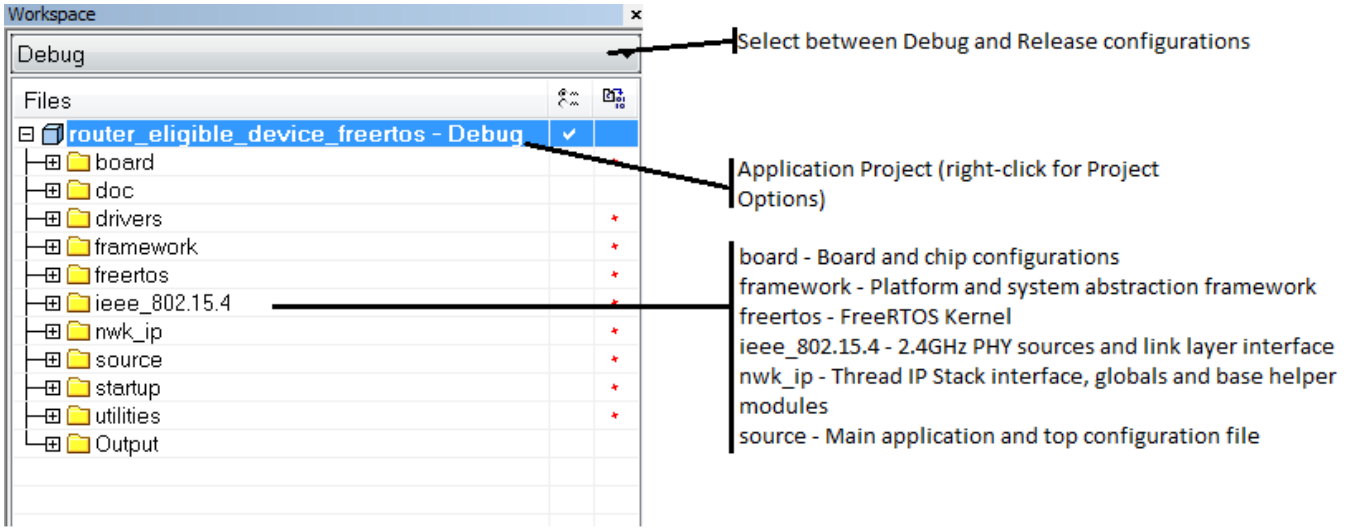

Figure 2. IAR Application Workspace

Similarly, the Linker sections contains application settings which are applied when the binary executable is created by the Linker. These include:

- Linker file to use for device memory layout of code and data
- Symbol definitions which affect whether an OTAP or Serial (FSCI) bootloader is included in the firmware memory map, if NVM section is reserved, enables placement of interrupts vector table in RAM memory, enables usage of internal flash for an OTA image, configures the number of flash sectors used for NVM and enables the erase of flash sectors used for NVM at first run.

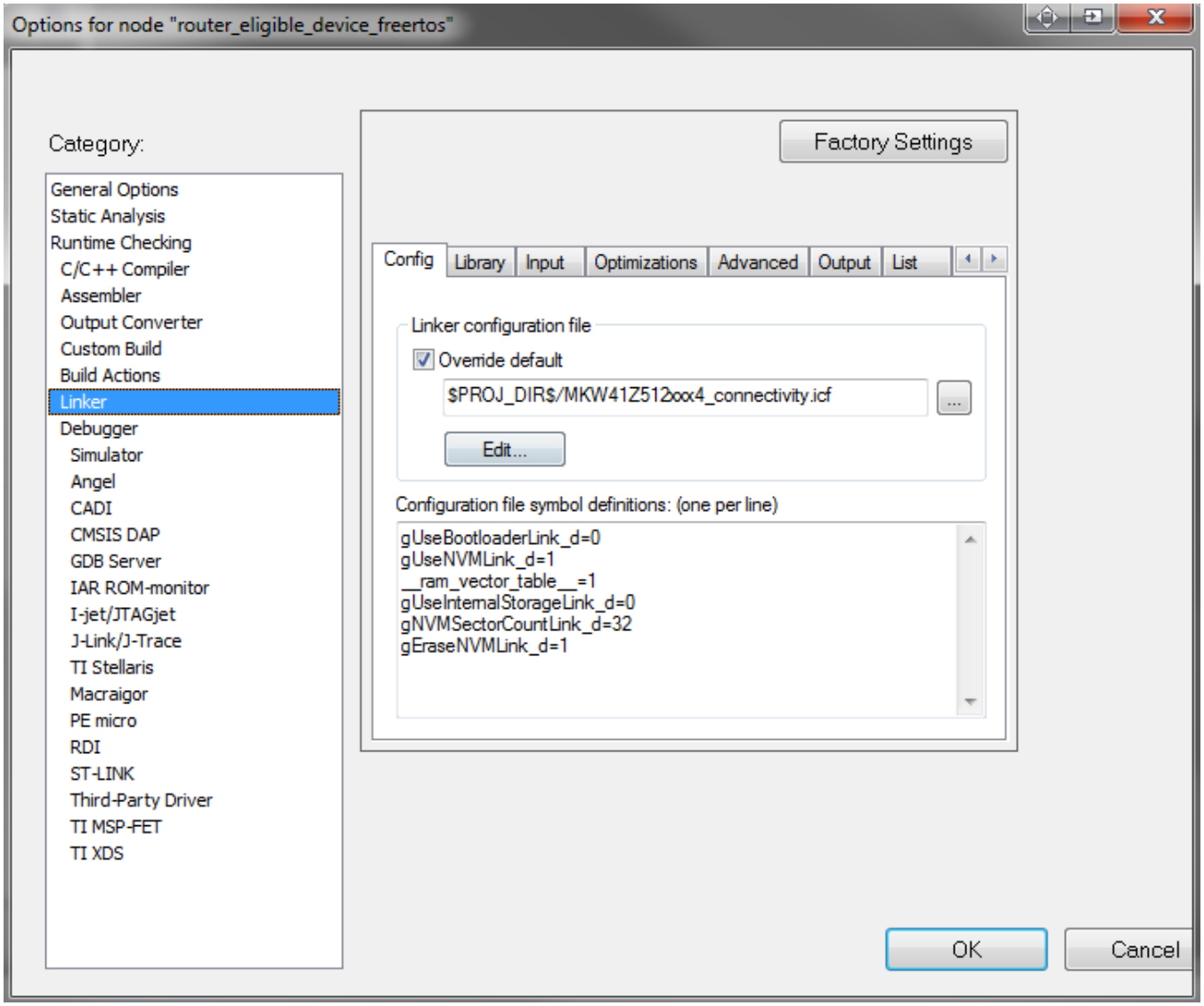

Figure 3. EWARM Project Options Configuration – linker settings

3.3.2 IAR Embedded Workbench header file configuration

The remainder of the stack and application compile time configuration is managed by means of header files in **\middleware\wireless\nwk_ip_1.2.8\examples\common** and Preinclude header file in **\middleware\wireless\nwk_ip_1.2.8\examples\<application>\config.** The settings in the headers in **\middleware\wireless\nwk_ip_1.2.8\examples\common** are default global settings applying to all applications projects. If an application must modify a set of specific global settings, it can override them either via the Preinclude header file in **\middleware\wireless\nwk_ip_1.2.8\examples\<application>\config .**. The Preinclude file has higher precedence as it is processed first when the compiler is invoked for a source module. The figure below shows the way Header file settings have precedence and can be overridden for application specific purposes:

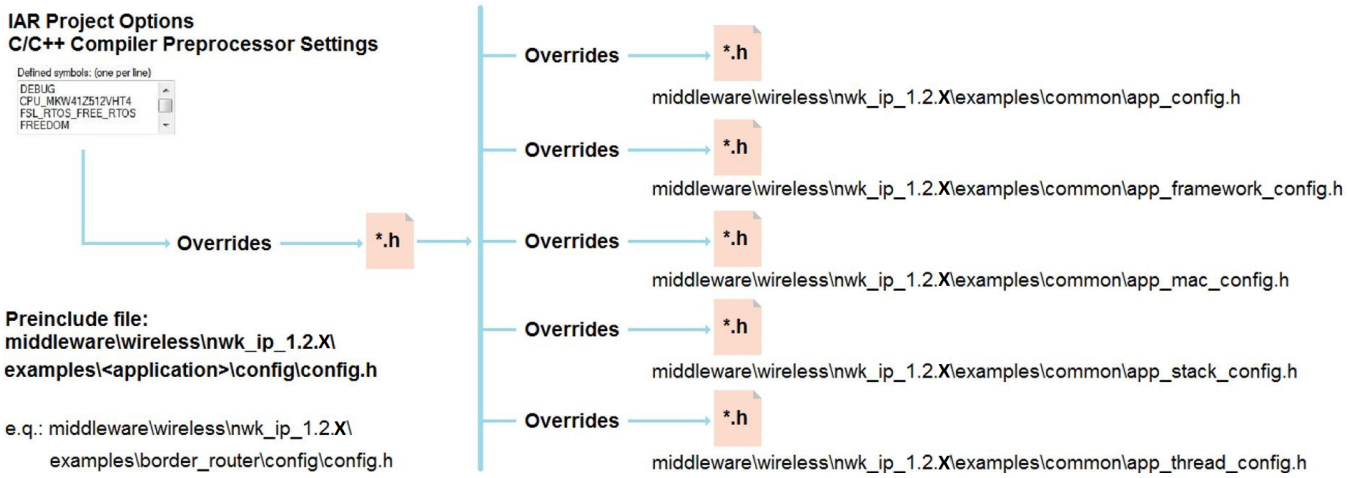

Figure 4. Compile time configuration setting header files structure and precedence

3.4 MCUXpresso IDE application project structure

The base example for a Router Eligible Device can be accessed and loaded into MCUXpresso IDE version 10.1.1 or later after successfully loading the sdk as shown in the Kinetis Thread Stack Demo Applications User's Guide section 7.

The projects can be loaded from the Quickstart Panel using the Import SDK example(s)… link.

In the downstream subfolders, projects are provided for FreeRTOS OS. The FreeRTOS OS project is used as the base for the examples below.

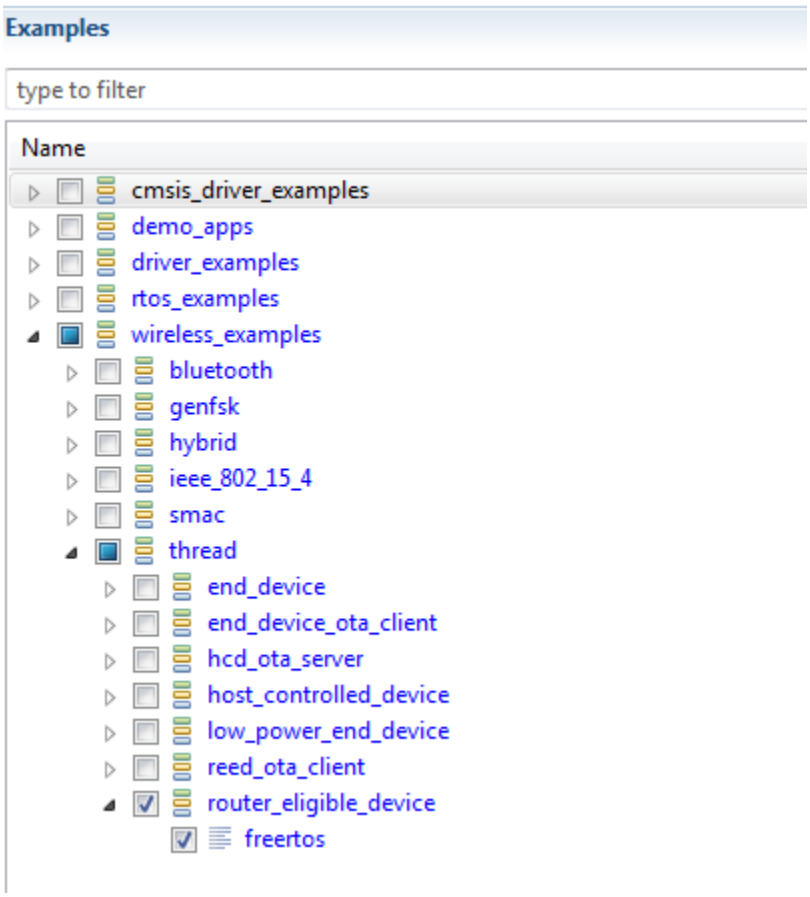

Figure 5. MCUXpresso IDE application project structure

Launching the workspace file into the MCUXpresso IDE tool shows the Project and individual file group and file organization in the C/C++ Projects panel on the left side of the window.

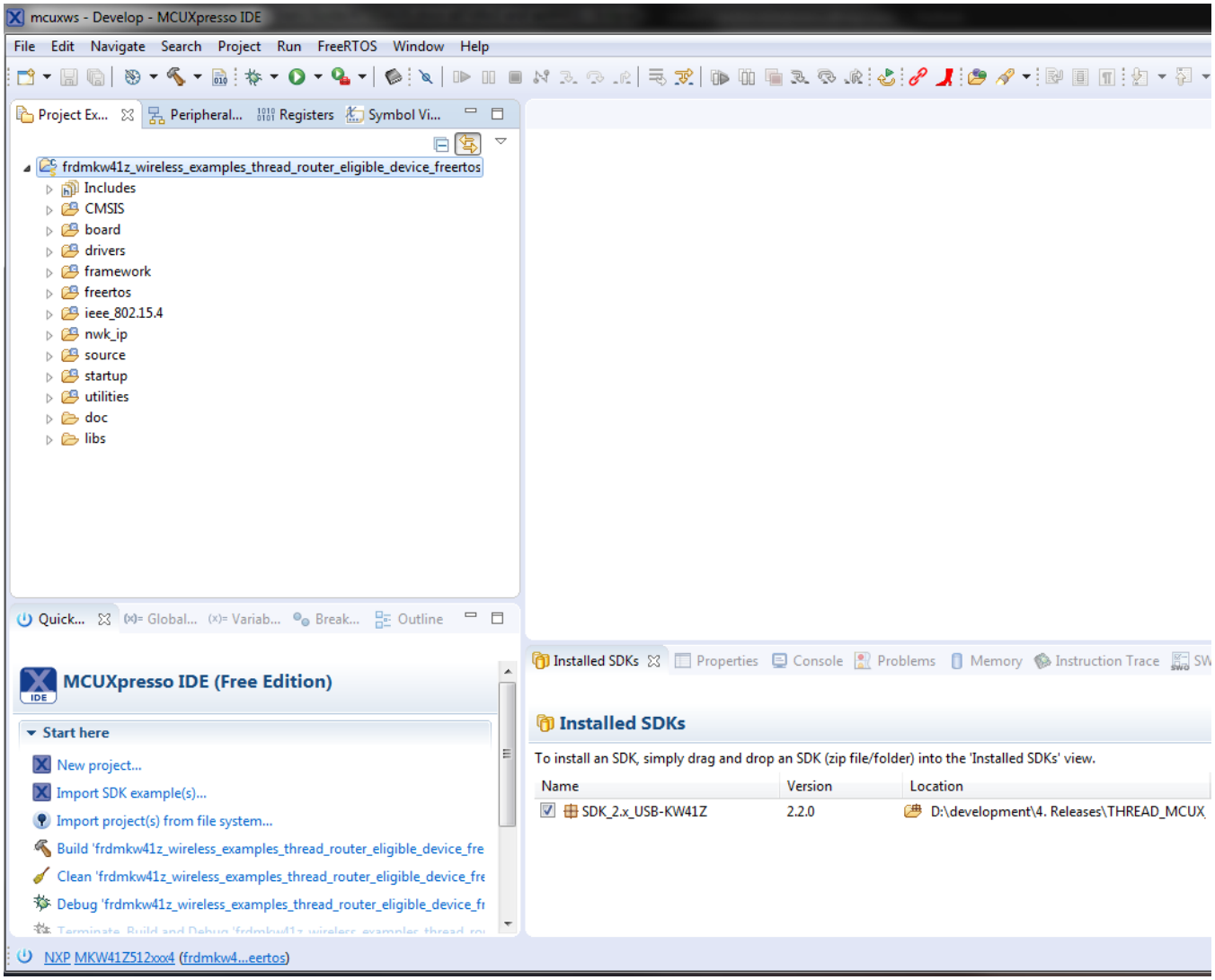

Figure 6. MCUXpresso IDE projects view

The Thread application workspace includes a standalone MCUXpresso project.

- router_eligible_device – the main application project, configured to output a binary S-Record (*.srec) file when built which can be loaded to run on the device firmware

The following figure shows an expansion of the main application project file group structure and the functionalities for the file in each group.

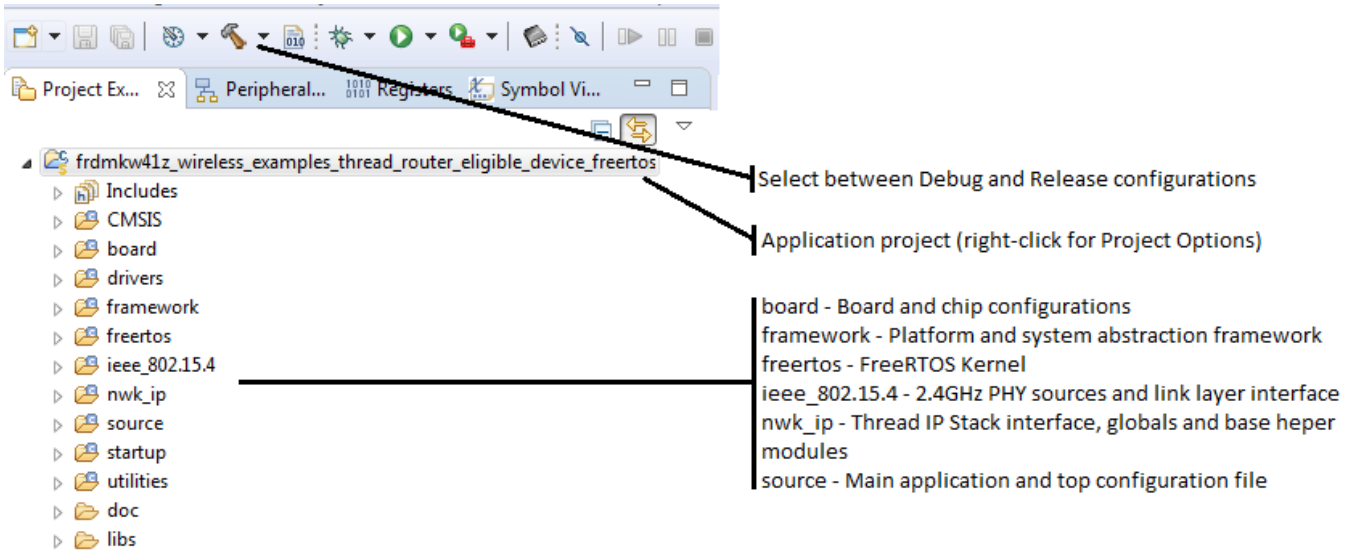

Figure 7. MCUXpresso IDE application workspace

3.5 MCUXpresso IDE compile time configuration

The compile time configuration for a Thread stack application is managed through a set of MCUXpresso and C module constructs, primarily preprocessor (macro) definitions.

3.5.1 MCUXpresso IDE project options configuration

The configuration settings are primarily contained in the **MCUXpresso Project Settings panel** and the *.h header files contained in **\middleware\wireless\nwk_ip_1.2.8\examples\common** and **\middleware\wireless\nwk_ip_1.2.8\examples\<application>\config**.

To see the configuration options set within the MCUXpresso Project Settings panel, right-click on the application project and select the Properties option.

The figure below shows the global Preprocessor options. These include the following:

- Setting the file at **\middleware\wireless\nwk_ip_1.2.8\examples\<application>\config\ config.h** as a **Preinclude file** which is included first by default whenever a source module compilation is invoked
- Preprocessor **Defined symbols** which consist in the main global settings for the system and stack for the application project

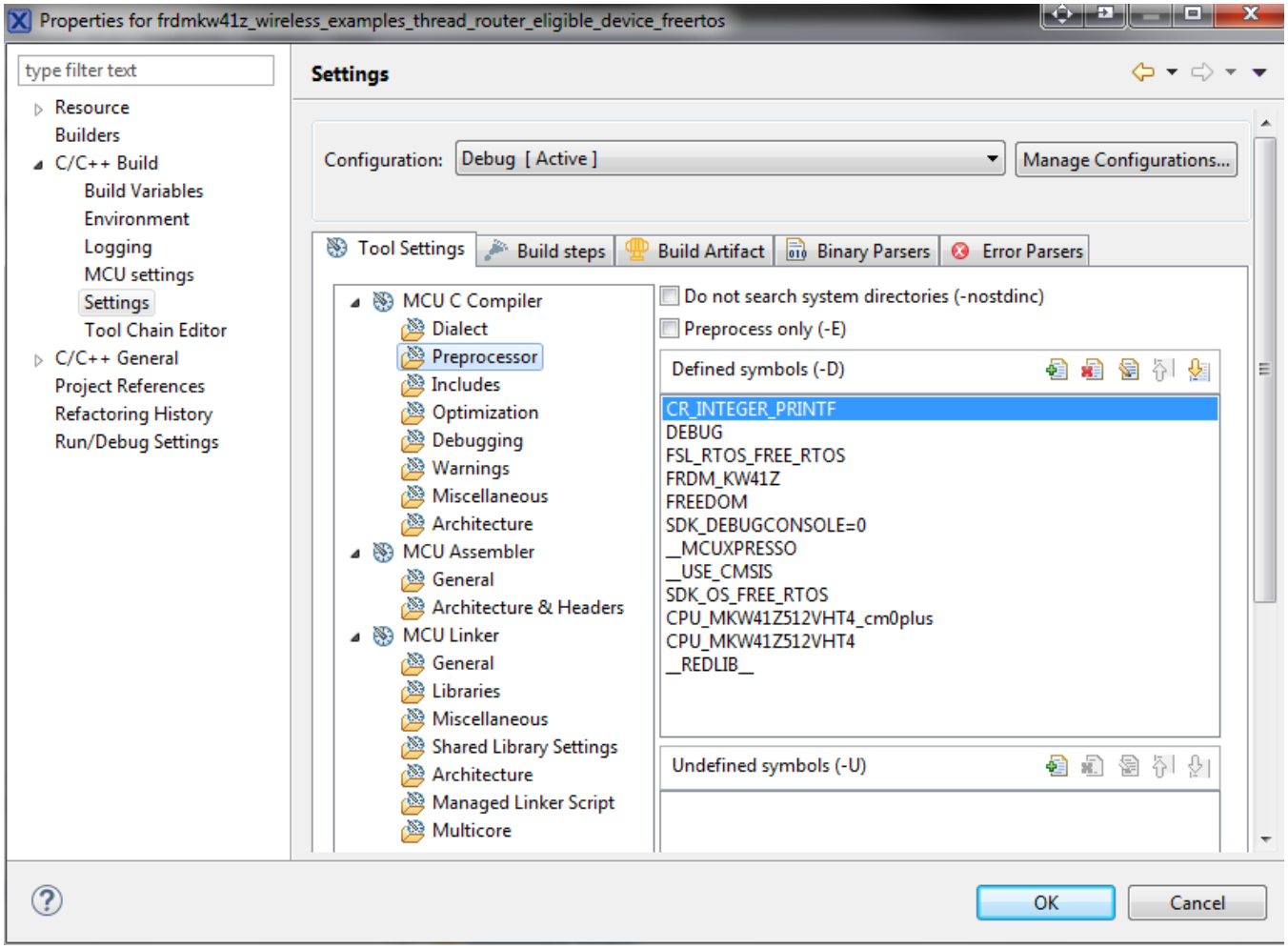

Figure 8. MCUXpresso IDE project options configuration – C compiler preprocessor settings

Compared to IAR the current MCUXpresso version does not support preprocessing the linker file. The application settings which are applied when the binary executable is created are contained inside the linker file which can be located from the Managed Linker Script section. These include the following:

- Symbol definitions which affect size of RTOS heap memory and whether an OTAP or Serial (FSCI) bootloader is included in the firmware memory map.

3.6 Run time bootstrapping and initialization

Once the Thread example application runs, FreeRTOS OS bootstraps the system and eventually runs the default **main_task** function defined in **\middleware\wireless\nwk_ip_1.2.8\examples\common\app_init.c** . The function represents the default FreeRTOS application-level task. This function initializes all other systems and IP stack modules by calling in turn their specific initializer functions. Notably, the call from main_task to Thread and IP stack initializer function THR_Init initializes the core Thread stack modules and creates a new FreeRTOS OS task which runs the stack functionality, state machine, and message processing. At Thread initialization, a unique device ID is read which is used by default as a vendor assigned suffix of the factory EUI64 used on the Thread interface. Function THR_Init is defined in application module

\middleware\wireless\nwk_ip_1.2.8\examples\common\app_thread_init.c

As part of their initialization other system modules such as the IEEE 802.15.4 MACPHY link layer or the blocking Serial Manager which handles serial interfaces communication (UART, USB, SPI and IIC) creates a new FreeRTOS OS task to allow non-blocking asynchronous processing. Each of the new system module RTOS tasks enter an infinite while(1) loop waiting on event signaling to the task via the OS Abstraction module EventWait API (OSA_EventWait). This is in order to be able to handle event and message processing for the respective module indefinitely while the system is running.

After it accomplishes all initializations, the main_task call enters its own while(1) loop where it repeatedly calls **APP_Handler** to serve high-level application events and functionality as well as allow entering low power state on idle for Low Power configurations, execute the pending operations of the NVM module and reset a watchdog.

3.7 Application task message queue and event handlers

APP_Handler must be defined in the main application file (in the base example: **\middleware\wireless\nwk_ip_1.2.8\examples \router_eligible_device\src\router_eligible_device_app.c**). It runs on the default application task, as called in the main_task loop. Its role is to extract any messages sent from other modules such as the core stack to the application by means of the application message queue **appThreadMsgQueue.** The call to **NWKU_MsgHandler**dequeues a message and calls the handler function callback specified in the function, thus ensuring the processing is done in the context of the RTOS task that was requested when creating the message.

To handle events within its message loop, the application registers the callback function using aStaticEventsTable from the file middleware\wireless\nwk_ip_1.2.8\base\stack_globals\event_manager_globals.c.

```
gThrEvSet_NwkJoin_c, Stack_to_APP_Handler, &mpAppThreadMsgQueue, gThrDefaultInstanceId_c, TRUE
```

Multiple incoming messages from other modules running their functionality in higher priority tasks are kept in the **appThreadMsgQueue** until the application task runs **APP_Handler** which dequeues them and calls the registered handler matching the event set identifier in the message.

This allows a high flexibility for asynchronous message passing and handling leveraging the RTOS provided task system.

3.8 Stack instance identifiers

Several Thread and IP stack APIs allow addressing different runtime entities by means of their API prototype. In the default configuration and for the majority of use cases, value 0 (the index for the first, default stack instance) must be used for the parameter when calling APIs which require the **instanceId_t thrInstanceID** parameter.

Chapter 4
Thread Network APIs

4.1 Factory default state

The first time an application runs after its firmware had been deployed to a device is in a factory default idle state where it is not connected to any network. For the examples deployed on standard development boards, this is indicated by RGB LED flashing blue and red LED flashing also.

From the Thread network perspective, when initially in Factory Default state or if reset to this state, the node radio is off and the device is off any Thread network.

In order to bring a Factory Default state device onto an active Thread network, applications can opt for one of the following:

- If the application has been configured at compile time to include routing capability, the application can opt to **create a new Thread network** entity and the current node can act as its initial Leader router. In order to create a new Thread network, the application must call **THR_NwkCreate** API. This API along with the rest of network state commands are defined in API interface header **\middleware\wireless\nwk_ip_1.2.8\core\interface\thread\ thread_network.h**.The application can also opt to establish the configuration setting for the network either at compile time, or dynamically.
- Otherwise, the application can choose to scan for networks to join created by other nodes using the **THR_NwkJoin** API. If joining a network instead of creating one, the application can set certain parameters, including whether the security credentials of the network are obtained by the DTLS in-band commissioning or whether the credentials are configured statically for testing or having been obtained out-of-band. **THR_NwkJoin** triggers commissioning or attached depending on the value of the iscommissioned attribute. It can be set to 0 (pre-commissioned, commissioned out-of-band, the node only attaches to a scanned network) or 1 (not pre-commissioned, the node perform in-band commissioning using DTLS).

Before joining, applications can also opt to set parameters dynamically by means of the THR_SetAttr API and do standalone scanning for networks using the THR_NwkScanWithBeacon or THR_NwkDiscoveryReq API.

4.2 Scanning for networks

To scan for networks in range, applications call THR_NwkScan as shown below.

First, install a handler for the scan events coming from the stack:

```
int32_t mThrInstanceId = 0;

static void APP_NwkScanHandler(uint8_t * param);

{gThrEvSet_NwkScan_c, APP_NwkScanHandler,&mpAppThreadMsgQueue, gThrDefaultInstanceId_c, TRUE}, from
aStaticEventsTable
```

Define the handler:

```
void APP_NwkScanHandler(uint8_t * param)
{
    thrEvmParams_t *pEventParams = (thrEvmParams_t *)param;
    thrNwkScanResults_t *pScanResults = &pEventParams->pEventData->nwkScanCnf;

    /* Handle the network scan result here */
    if(pScanResults)
    {
```

```
#if THREAD_USE_SHELL
        SHELL_NwkScanPrint(pScanResults);
#endif
        MEM_BufferFree(pScanResults);
    }
        /* Free Event Buffer */
    MEM_BufferFree(pEventParams);
}
```

With the handler declared, initiate the scan:

```
thrNwkScan_t thrNwkScan;
thrNwkScan.maxThrNwkToDisc = 10;          /* scan maximum 10 thread networks */
thrNwkScan.scanChannelsMask = 0x07FFF800; /* all channels*/
thrNwkScan.scanDuration = 2            /* exponential scale for duration as defined in IEEE 802.15.4 */;
thrNwkScan.scanType = gThrNwkScan_ActiveScan_c;
THR_NwkScan(mThrInstanceId, &thrNwkScan);
```

The scan results are received in APP_NwkScanHandler under the format thrNwkScanResults_t* shown below:

```
/*! The Network scan results*/

typedef struct thrNwkScanResults_tag
{
    thrNwkScan_t scanInfo;
    uint8_t      numOfEnergyDetectEntries;
    uint8_t     *pEnergyDetectList;     /*One byte for each channel. Only the channels from
scanInfo.scanChannelsMask should be handled; the rest of the channels are zeros */
    uint8_t      numOfNwkScanEntries;   /*Number of discovered network performing an active scan */
    thrNwkActiveScanEntry_t nwkScanList[]; /* where the network discovery list begins. Note that all
these buffres shall be freed */
}thrNwkScanResults_t;
```

4.3 Joining a network

For an application to initiate and join an existing Thread network the THR_NwkJoin API is called. The API supports initiating joining with in-band commissioning or with pre-commissioned settings matching an out-of-band commissioning configuration. The commissioning mode can be set at compile time using the THR_DEV_IS_OUT_OF_BAND_CONFIGURED parameter.

4.3.1 General network settings

The application settings and device attributes described below usually impact how a node joins or creates a network.

4.3.1.1 Network scan channel mask

The scan channel mask is a 32 bitmap value where bits 11 through 26 (LSB to MSB indexes) match a channel index in the 2.4GHz channel list allowed by IEEE 802.15.4.

The full channel mask with all valid channels allowed has the format: 0x07FFF800. The full channel mask should always be set as the default for production devices:

```
#define THR_SCANCHANNEL_MASK                        (0x07FFF800)
```

Alternatively, a channel mask allowing a single channel value can be set more simply using by shifting bit 1 to the left to the respective channel index. This setting can facilitate controlled joining testing during development by using channel separation for

guiding devices to networks and is the default value for the example application, with the selected channel being the one at index 25:

```
#define  THR_SCANCHANNEL_MASK                          (0x00000001 << 25)
```

To set the channel mask at runtime:

```
instanceId_t thrInstId = 0;
uint32_t aIndex = 0;

uint32_t scanMask = 0x07FFF800;
THR_SetAttr(thrInstId, gNwkAttrId_ScanChannelMask_c, aIndex, sizeof(uint32_t), &scanMask);
```

4.3.1.2 Short PAN identifier (16-bit PAN ID)

The short format for the PAN (Personal Area Network) identifier is a 16-bit value that is required by the IEEE 802.15.4 link layer in order to allow communicating nodes based on their short 16-bit address within a PAN. As such, nodes are sending messages to each other using 4 bytes: the 16-bit PANID followed by the 16-bit short address. In Thread, the network layer is managing both short PAN ID and short node addresses. Within the same Thread network entity regarded as a PAN, all nodes are using the same PAN ID and different PAN IDs in messages indicate different Thread networks.

It is recommended that the application does not change the default settings where the PAN ID is pre-set to 0xFFFF, indicating generation of a random PAN ID at network creation by the initial Leader, and otherwise for Joiners to not filter networks to join based on this field.

```
#define THR_PAN_ID                                    0xFFFF
```

For development and testing purposes the PAN ID can be set to a specific value, different from 0xFFFF. If the PAN ID is set to a specific value, the initial Leader specifically chooses this ID when creating a network. Nodes do not join or interact with existing network because they have different PAN IDs.

```
#define THR_PAN_ID                                    0xCAFE
```

To set the PAN ID at runtime (before creating or joining a network):

```
instanceId_t thrInstId = 0;
uint32_t aIndex = 0;

uint16_t panId = 0xCAFE;
THR_SetAttr(thrInstId, gNwkAttrId_PanId_c, aIndex, sizeof(uint16_t), panId);
```

4.3.1.3 Extended PAN identifier (64-bit XPAN ID)

The Extended PAN Identifier (XPAN ID) is a 64-bit value used by Thread to uniquely identify and discern between multiple Thread network entities. When generated randomly by the initial Leader (as done by Thread examples by default), the 64-bit values have much lower chances of duplicate conflicts than the 16-bit link layer PAN ID.

It is recommended that the application does not change the default settings where the XPAN ID is pre-set to all 0xFFs, indicating generation of a random XPAN ID at network creation by the initial Leader, and otherwise for Joiners to not filter networks to join based on this field.

```
#define THR_EXTENDED_PAN_ID           {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF}
```

The XPAN ID is advertised to the Joiner via beacon transmissions or discovery messages.

To set the XPAN ID at runtime (before creating or joining a network):

```
instanceId_t thrInstId = 0;
uint32_t aIndex = 0;

uint64_t xPanId = {0x00, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF};
THR_SetAttr(thrInstId, gNwkAttrId_ExtendedPanId_c, aIndex, sizeof(uint64_t), &xPanId);
```

4.3.1.4 Network name

The network name is a value which can be setup by the application to a default value, however it is recommended that applications which allow network creation via a rich User Interface request the value of the Network Name to be input by the user of the network. In that regard, the Network Name for Thread is an identifier similar to the SSID for Wi-Fi networks, allowing easy representation of the networks when scanning or selecting a network to join.

The example applications are setting the value of the network name to the default of Kinetis_Thread.

```
#define THR_NETWORK_NAME                          {14, "Kinetis_Thread"}
```

To set the Network Name at runtime (for example, before network creation, as indicated by the user).

```
instanceId_t thrInstId = 0;
uint32_t aIndex = 0;

thrOctet16_t newName = {15, “MyThreadNetwork”};
THR_SetAttr(thrInstId, gNwkAttrId_NwkName_c, aIndex, sizeof(newName), &newName);
```

The length of the Network Name is limited to 16 bytes.

4.3.2 Registering the network join event set handler

To monitor the asynchronous messages returned by the stack during the joining process, applications should register a callback for the network join event set. The code snippets below illustrates how the example application is registering the callback Stack_to_APP_Handler to also receive joining events and messages:

```
#define gThrDefaultInstanceId_c                0
static taskMsgQueue_t * mpAppThreadMsgQueue;

...
void APP_Init
(
  void
)
{
    /* Initialize pointer to application task message queue */
    mpAppThreadMsgQueue = &appThreadMsgQueue;

    /* Initialize main thread message queue */
    ListInit(&appThreadMsgQueue.msgQueue,APP_MSG_QUEUE_SIZE);
    ...
}

...

void Stack_to_APP_Handler
(
  uint8_t* param
)
```

```
{
    thrEvmParams_t *pEventParams = (thrEvmParams_t *)param;

    switch(pEventParams->code)
    {
    ...
        case gThrEv_NwkJoinCnf_Success_c:
        case gThrEv_NwkJoinCnf_Failed_c:
            APP_JoinEventsHandler(pEventParams->code);
            break;
    ...
        default:
            break;
    }

    /* Free event buffer */
    MEM_BufferFree(pEventParams->pEventData);
    MEM_BufferFree(pEventParams);
}

static void APP_JoinEventsHandler(thrEvCode_t evCode)
{
    if(evCode == gThrEv_NwkJoinCnf_Failed_c)
    {
    // JOINING HAS FAILED - could retry: (void)THR_NwkJoin(mThrInstanceId);
    ...
    }
    else if (evCode == gThrEv_NwkJoinCnf_Success_c)
    {
    // JOINING IS SUCCESFUL! - should indicate that to user
    ...
    }
}
```

4.3.3 Initiating joining

To initiate network joining, applications must simply call the THR_NwkJoin API:

```
instanceId_t thrInstId = 0;
thrJoinDiscoveryMethod_t discMethod = gUseThreadDiscovery_c;

 (void)THR_NwkJoin(thrInstId, discMethod);
```

Joining is initiated with either in-band or out-of-band commissioning depending on compile time flag THR_DEV_IS_OUT_OF_BAND_CONFIGURED and then at runtime by the value of the stack attribute having identifier gNwkAttrId_IsDevCommissisoned_c.

In all use cases (either for in-band or out-of-band commissioning), the result of the joining process is received by means of the callback system for Join events describe in the previous section.

Application can influence and make options during the joining state machine (for example, select which network to join by implementing specific callbacks with reference implementations shown in file **\middleware\wireless\nwk_ip_1.2.8\examples\common\app_thread_callbacks.c**.

4.3.4 Joining a network with in-band commissioning

4.3.4.1 Setting in-band configuration

If THR_DEV_IS_OUT_OF_BAND_CONFIGURED is set to FALSE (default), in-band commissioning is used and an active commissioner must be available in order to allow the device onto the network.

Alternatively setting a device to use In-Band commissioning can be done at runtime (while the device is in factory reset state) by updating the matching attribute as shown below:

```
instanceId_t thrInstId = 0;
uint32_t aIndex = 0;

bool_t isDeviceCommissioned = TRUE;

THR_SetAttr(thrInstId, gNwkAttrId_IsDevCommissioned_c, aIndex, sizeof(bool_t),
               &isDeviceCommissioned);
```

Furthermore, the IEEE EUI64 Address and PSKd device settings or attributes have an effect on network joining with commissioning.

4.3.4.2 The IEEE EUI64 address

The factory IEEE EUI64 is a unique per device address. The EUI64 should be provisioned as a universally unique link layer (MAC) address having the specific format defined by IEEE. To that point, the device manufacturer should obtain a vendor OUI prefix meant for its EUI set then use it in conjunction with a unique per-device identifier as a suffix to create and provision a universally unique EUI64 for each Thread interface.

In Thread, the EUI64 is not used in message headers sent to other nodes Thread instead uses privacy random extended addresses for the MAC frame headers. However the EUI64 is still used as an identifier by the commissioner in order to:

- steer (guide) devices having respective EUI64 to request joining specific networks by advertising a filter in the network beacons
- uniquely identify one or more devices as indicated by the user to the commissioner as expected joiners with which a DTLS session for network authentication is started

For the purposes above, the EUI64 is usually entered on the commissioner by having user either typing the address in or by reading the value from a QR code or NFC tag by the commissioner smart device application.

By default the example applications have address 0x146E0A0000000001 set:

```
#define THR_EUI64_ADDR                                0x146E0A0000000001
```

The address can be set at runtime (before joining) as shown below:

```
instanceId_t thrInstId = 0;
uint32_t aIndex = 0;

uint64_t ieeeEui64 = 0x146E0A0000000001;
THR_SetAttr(thrInstId, gNwkAttrId_IeeeAddr_c, aIndex, sizeof(uint64_t), &ieeeEui64);
```

4.3.4.3 The PSKd device password

In Thread, the PSKd is a per-device unique passphrase or serial number/identifier. It is used as a shared secret password in the DTLS EC-JPAKE based key-exchange which happens between a Joiner device and the commissioner. The PSKd is usually meant to be displayed on the device product label as an alphanumerical string or a QR code. The main role of the PSKd is to ensure that only the rightful user with physical access to the device can initiate commissioning it onto the network. As a unique per-device secret shared between the Joiner and the Commissioner, the PSKd also ensures the cryptographic seed for the EC-JPAKE key exchange.

Similar to the EUI64, the PSKd is also usually entered on the commissioner by having user either typing the string in or by reading the value from a QR code or NFC tag by the commissioner smart device application.

By default the example applications have the PSKd set to ‘THREAD’ as shown below. The PSKd must be encoded in base32-thread (0-9, A-Y excluding I, O, Q and Z). The numerical value in the definition indicates the length of the alphanumerical PSKd string which follows. The default value is to be used for development only. Production devices should always be provisioned with a per-device unique PSKd:

```
#define THR_PSK_D                                   {6, "THREAD"}
```

To set the PSKd to a different value at runtime, the application can set the matching attribute. Strings must have the minimum length 6 and the maximum length 32 bytes. For example, to set to string value A1B2C3.

```
instanceId_t thrInstId = 0;
uint32_t aIndex = 0;

thrOctet32_t newPskd = {6, “A1B2C3”};
THR_SetAttr(0, gNwkAttrId_PSKd_c, aIndex, sizeof(newPSKd), &newPSKd);
```

4.3.4.4 In-band commissioning network selection callback

Beyond setting specific attributes for a Joiner, the application may also need a granular control over how scanning for parent routers in existing networks is done for in-band commissioning by the Joiner node.

For that purpose, the Thread stack provides a callback function which must be implemented by the application to steer the stack during in-band commissioning network scanning. The callback is APP_JoinerDiscoveryRespCb and a reference implementation is showcased in file **\middleware\wireless\nwk_ip_1.2.8\examples\common\app_thread_callbacks.c**.

The callback receives three types of events as follows.

- Scan initialization: gThrDiscoveryStarted_c – at this point the application can perform further purging of the discovery table from previous scans, or allocating memory to handle incoming indication of networks and routers
- Indication for each Discovery Response packet received from a router which represents a candidate: gThrDiscoveryRespRcv_c – the Discovery Response attributes are found in pDiscoveryRespInfo argument
- Scan completion: gThrDiscoveryStopped_c – at this point the application can process the potential parent list and call API THR_NwkJoinWithCommissioning which expects the application to provide a filter list of the Discovery Responses from networks to join

The reference implementation for APP_JoinerDiscoveryRespCb is provided below:

```
void APP_JoinerDiscoveryRespCb
(
    instanceId_t thrInstId,
    thrDiscoveryEvent_t event,
    uint8_t lqi,
    thrDiscoveryRespInfo_t* pDiscoveryRespInfo,
    meshcopDiscoveryRespTlvs_t *pDiscoveryRespTlvs
)
{
    bool_t addToJoiningList = FALSE;

    if (event == gThrDiscoveryStarted_c)
    {
        const uint16_t maxSize  = THR_MAX_NWK_JOINING_ENTRIES * sizeof(thrNwkJoiningEntry_t);
        gNbOfNwkJoiningEntries = 0;
        if(gpNwkJoiningList == NULL)
        {
            gpNwkJoiningList = MEM_BufferAlloc(maxSize);
```

```
        }
        if(gpNwkJoiningList)
        {
            /* Reset the Network Joining list */
            FLib_MemSet(gpNwkJoiningList, 0, maxSize);
        }
    }
    else if ((event == gThrDiscoveryRespRcv_c) && gpNwkJoiningList)
    {
        thrNwkJoiningEntry_t joinerEntry = {0};
        joinerEntry.steeringMatch = gMeshcopSteeringMatchNA_c;
        joinerEntry.channel = pDiscoveryRespInfo->channel;
        joinerEntry.panId = pDiscoveryRespInfo->panId;
        FLib_MemCpy(joinerEntry.euiAddr, pDiscoveryRespInfo->aEui, 8);
        FLib_MemCpy(joinerEntry.aXpanId, pDiscoveryRespTlvs->pXpanIdTlv->xPanId,
            pDiscoveryRespTlvs->pXpanIdTlv->len);

        if(pDiscoveryRespTlvs->pCommissionerUdpPortTlv)
        {
            joinerEntry.commissionerUDPPort = ntohas(pDiscoveryRespTlvs->pCommissionerUdpPortTlv-
>aPort);
        }
        if(pDiscoveryRespTlvs->pJoinerUdpPortTlv)
        {
            joinerEntry.joinerUDPPort = ntohas(pDiscoveryRespTlvs->pJoinerUdpPortTlv->aPort);
        }

        /* Joiner case */
        if(gMeshcopCommissionerMode != gThrCommissionerModeNative_c)
        {
          if(THR_DISC_RSP_VER_GET(pDiscoveryRespTlvs->pDiscRespTlv->verFlags) ==  THR_PROTOCOL_VERSION)
            {
                if (THR_GetAttr_IsDevCommissioned(thrInstId) == TRUE)
                {
                    addToJoiningList = TRUE;
                }
                else
                {
                    joinerEntry.steeringMatch = MESHCOP_CheckSteeringData(pDiscoveryRespTlvs-
>pSteeringDataTlv);
                    if(joinerEntry.steeringMatch != gMeshcopSteeringMatchNone_c)
                    {
                        addToJoiningList = TRUE;
                    }
                }
            }
        }
        /* Native Commissioner case */
        else
        {
            /* Join as a Native Commissioner. Add filters here.*/
            if(
                (THR_DISC_RSP_VER_GET(pDiscoveryRespTlvs->pDiscRespTlv->verFlags) ==
THR_PROTOCOL_VERSION) &&
                (THR_DISC_RSP_N_GET(pDiscoveryRespTlvs->pDiscRespTlv->verFlags) == 1)
                )
            {
                addToJoiningList = TRUE;
            }
        }
```

```
        if(addToJoiningList)
        {
            addToJoiningList = APP_JoinerNwkListAdd(&joinerEntry, thrInstId, gpNwkJoiningList,
&gNbOfNwkJoiningEntries);
        }
    }
    else if ((event == gThrDiscoveryStopped_c) && gpNwkJoiningList)
    {
        /* Start the joining process. The list will be freed by the function. */
        APP_StartNwkJoin(thrInstId):
    }
}
```

4.3.5 Joining a network with out-of-band commissioning

4.3.5.1 Setting out-of-band configuration

If THR_DEV_IS_OUT_OF_BAND_CONFIGURED is set to TRUE, out-of-band commissioning is enabled and the network credentials, primarily the Master Key must be provided by the application.

Alternatively a device to use out-of-band commissioning can be done at runtime (while the device is in factory reset state) by updating the matching attribute as shown below:

```
instanceId_t thrInstId = 0;
uint32_t aIndex = 0;

bool_t isDeviceCommissioned = TRUE;

THR_SetAttr(thrInstId, gNwkAttrId_IsDevCommissioned_c, aIndex, sizeof(bool_t),
               &isDeviceCommissioned);
```

4.3.5.2 Thread master key

The Thread Master Key for a network consists in the base key information from which both link layer and MLE layer security as specified by Thread are derived for all nodes in the network. For in-band commissioning the Master Key is obtained by a Joiner node by means of the DTLS EC-JPAKE key exchange. For out-of-band commissioning, the application must provide the actual value of the key.

To set a specific value to the master key at compile time, set the THR_MASTER_KEY preprocessor define:

```
#define THR_MASTER_KEY           {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, \
                                  0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff}
The key can be set at runtime (before joining) as shown below:
instanceId_t thrInstId = 0;
uint32_t aIndex = 0;

uint8_t   masterKey[16] = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, \
                        0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff};
THR_SetAttr(thrInstId, gNwkAttrId_NwkMasterKey_c, aIndex, sizeof(masterKey), masterKey);
```

4.3.5.3 Out-of-band commissioning network selection callback

If THR_DEV_IS_OUT_OF_BAND_CONFIGURED is set to TRUE, then the device scans for Thread networks on the configured channel mask and attaches networks based on the application callback behavior as set by function

APP_OutOfBandSelectNwkWithBeaconCb. The default implementation of the callback can be found in file **\middleware\wireless\nwk_ip_1.2.8\examples\common\app_thread_callbacks.c**

This function receives beacon frames from existing networks which were discovered during scanning:

```
bool_t APP_OutOfBandSelectNwkWithBeaconCb( instanceId_t thrInstId, thrBeaconInfo_t *pThrBeacon)
{
    bool_t useThisNWK = FALSE;

    /* ADD YOUR FILTER HERE. */
    if(gaThrDeviceConfig[thrInstId].outOfBandChannel != 0)
    {
        /* If the channel is known, apply this filter */
        if(gaThrDeviceConfig[thrInstId].outOfBandChannel == pThrBeacon->channel)
        {
            useThisNWK = TRUE;
        }
    }
    else if(ntohall(gaThrDeviceConfig[thrInstId].xPanId) != THR_ALL_FFs64)
    {
        /* If the extended pan ID is known, apply this filter */
        if( FLib_MemCmp(pThrBeacon->payload.xpanId, gaThrDeviceConfig[thrInstId].xPanId,8) )
        {
            useThisNWK = TRUE;
        }
    }
#if ENABLE_MESHCOP_JOINER_FILTER_NWK_NAME
    else if(gaThrDeviceConfig[thrInstId].nwkName.length != 0)
    {
        /* If the network name is known, apply this filter */
        if(FLib_MemCmp(pThrBeacon->payload.nwkName, gaThrDeviceConfig[thrInstId].nwkName.aStr, 16))
        {
            useThisNWK = TRUE;
        }
    }
#endif
    else
    {
        useThisNWK = TRUE;
    }

    return useThisNWK;
}
```

4.4 Creating a new Thread network

As shown in Router Eligible Device, Border Router or Host Controlled Device files, an application can choose to create (bootstrap, form) a new Thread network entity. To create a new Thread network, the application must call:

```
THR_NwkCreate(thrInstanceId);
```

With a default stack configuration the behavior of the THR_NwkCreate is the following:

If more than one RF channel is specified in the RF channel mask, an energy scan is done to select an optimal channel for the options in the mask on which to create the network; this is done by processing in the callback APP_NwkCreateCb found in **\middleware\wireless\nwk_ip_1.2.8\examples\common\app_thread_callbacks.c**. After selecting the formation channel, the

callback calls **THR_FormNwkOnChannel**, a new PAN ID and extended PAN ID (XPAN ID, xpan) are created and a Leader Router starts on the respective channel

The result of the **THR_NwkCreate** API call is returned as a **gThrEv_NwkCreateCnf_Success_c** or **gThrEv_NwkCreateCnf_Failed_c** message events.

4.5 Starting and configuring the commissioner module

A Commissioner for a Thread network is a management network module and IP protocol end-point which is used to allow new devices to join Thread networks and optionally perform other management tasks. A Commissioner module does not necessarily have be deployed to a device having a Thread interface. The Mesh Commissioning protocol allows commissioners in the home Local Area Network (LAN) to interact with Thread networks through a Border Router in order to fulfill the commissioner role.

The Kinetis Thread Stack provides a full Mesh Commissioning Protocol API allowing devices to also start and operate an interior network Commissioner.

The example applications by default start a commissioner module on a Thread partition Leader with a default provisioning of security to allow all devices having PSKd 'THREAD' to join. This is done by calling the **MESHCOP_StartCommissioner** API when the device transitions to a Leader role, which can take place at the creation of a new Thread network or when a network is partitioned and the Leader role is taken by another Active Router of the partition.

To activate the commissioner module on any device, the application must call:

```
MESHCOP_StartCommissioner(thrInstanceId);
```

The effect of the Start Commissioner call is that the Thread device petitions the partition Leader to be accepted as a Commissioner. If the Leader role is active on the same device (Leader and Commissioner roles are collapsed on the same node), then the petitioning is auto-accepted locally, otherwise a petitioning message is sent through the Thread network between the Commissioner and the Leader.

The Leader responds to the petitioning by allowing or rejecting it. On the Commissioner device, this is indicated to the application using the **gThrEv_MeshCop_CommissionerPetitionAccepted_c** and **gThrEv_MeshCop_CommissionerPetitionRejected_c** event messages.

Before users initiate joining a new device to a network, the commissioner should be configured specifically to allow the specific device to join. The identifiers for the device are its Thread interface IEEE EUI64 and the device password (PSKd). In order for the Commissioner module to allow a specific device to join, the EUI64 and the PSKd must be added to the Commissioner allowed joiner List:

To add a new Joiner to the allowed joiner list, the application must call:

```
MESHCOP_AddExpectedJoiner(mThrInstanceId, aEui, aPSKd, sizeOfPSKdInBytes, TRUE)
```

The **MESHCOP_RemoveAllExpectedJoiners**, **MESHCOP_RemoveExpectedJoiner**, **MESHCOP_GetExpectedJoiner** APIs as defined in header file nwk_ip\core\interface\thread\thread_meshcop.h may be used to further manipulate the current expected joiner list which is stored by the Commissioner module.

The Expected Joiner list APIs are only processed internally and do not have network effects of steering new devices to join before the steering information is synchronized to the Active Routers. To disseminate and synchronize expected joiner information to the Thread network Active Routers (including the router entity on the local device of the Commissioner) the application must call:

```
MESHCOP_SyncSteeringData(mThrInstanceId, gMeshcopEuiMaskExpectedJoinerList_c)
```

The **MESHCOP_SyncSteeringData** API can also be called with the flags **gMeshcopEuiMaskAllFFs_c** to not filter the EUI64 and steer all devices for the respective network or alternatively to **gMeshcopEuiMaskAllZeroes_c** to deny all devices (the Commissioner "closes" the network for joining).

Active commissioner applications are strongly recommended to explicitly close the network for joining in most cases once the commissioning session is over by calling:

```
MESHCOP_SyncSteeringData(mThrInstanceId, gMeshcopEuiMaskAllZeroes_c)
```

To deactivate the commissioner, the application must call:

```
MESHCOP_StopCommissioner(mThrInstanceId)
```

This stops the keep alive messages sent from the Commissioner to the Leader, sends keep alive with reject and frees resources taken up by the commissioner.

Chapter 5
Reading IP Address Configuration

APIs defined in file **\middleware\wireless\nwk_ip_1.2.8\core\interface\thread\thread_utils.h** and data definitions in **\middleware\wireless\nwk_ip_1.2.8\core\interface\thread\thread_types.h** can be used to inspect the Thread IPv6 address configuration of a Thread node after the node is part of a Thread network.

The types of IPv6 addresses defined for a Thread are shown as part of the **nwkIPAddrType_t** definition below:

```
typedef enum nwkIPAddrType_tag
{
    gLL64Addr_c = 0x00,         /*!< Link-Local 64 address (the IID is MAC Extended address Which is
not the factory-assigned IEEE EUI-64,)*/
    gMLEIDAddr_c = 0x01,        /*!< Mesh-Local Endpoint Identifier address (the IID is randmon) */
   gRLOCAddr_c = 0x02,        /*!< Routing Locator address (the IID encodes the Router and Child IDs.)*/
    gGUAAddr_c  = 0x03,         /*!< Global Unicast Address*/
    gAnycastAddr_c = 0x04,      /*!< Anycast IPv6 addresses */
    gAnyIpv6_c = 0x05,          /*!< All IPv6 address */
    gAllThreadNodes_c = 0x06,   /*!< All Thread nodes address */
} nwkIPAddrType_t;
```

APIs defined in file **\middleware\wireless\nwk_ip_1.2.8\core\interface\thread\thread_utils.h** and data definitions in **\middleware\wireless\nwk_ip_1.2.8\core\interface\thread\thread_types.h** can be used to inspect the Thread IPv6 address configuration of a Thread node after the node is part of a Thread network.

The **THR_NumOfIP6Addr** function returns the number of bound IPv6 address of a specific type.

The **THR_GetIP6Addr** function returns the bound IPv6 addresses specified by IP address type parameter. At pDstIPAddr location should be allocated enough memory space to copy all the requested IPv6 addresses. Applications should use the **THR_NumOfIP6Addr** to find how much space is needed to get the requested IPv6 addresses.

For inspecting address configuration for the IP media interface used by the stack, applications may also use the advanced APIs in **\middleware\wireless\nwk_ip_1.2.8\core\interface\modules\ip_if_management.h**.

Chapter 6
Constrained Application Protocol (CoAP)

The Kinetis Thread stack provides a high-level CoAP module for using the Constrained Application Protocol over UDP. The CoAP module is used for both stack and application purposes.

The code examples below walk through how CoAP is used by the example applications.

Initialize a callback configuration structure for CoAP URI paths:

```
coapRegCbParams_t cbParams [] = { { APP_CoapCallback, (coapUriPath_t*)&gURI_OPTION } }
const coapUriPath_t gURI_OPTION = {SizeOfString("option"), "option"};
```

Register services to the CoAP modules by creating a CoAP instance (in this example, unsecured):

```
uint8_t mCoapInst;
sockaddrStorage_t coapStartUnsecParams = {0};
NWKU_SetSockAddrInfo(&coapStartUnsecParams, NULL, AF_INET6, port, 0, gIpIfSlp0_c);
mCoapInst = COAP_CreateInstance(NULL, &coapStartUnsecParams, &cbParams, 1);
```

When having data to send on the instance, open a session to the remote destination address

```
coapSession_t *pSession = NULL;
uint8_t buffer[3] = { 0x01, 0x02, 0x03};
uint8_t * pCoapPayload = &buffer[0]; /* This variable MUST be set to a valid location */
uint8_t payloadSize = sizeof(buffer); /* This variable MUST be set to a valid size */
pSession = COAP_OpenSession(mCoapInst);
if (NULL != pSession)
{
    /* initialize to Non confirmable by default for multicast */
    coapMsgTypesAndCodes_t coapMessageType = gCoapMsgTypeNonPost_c;

    /* do not use the callback for non-confirmable */
    pSession->pCallback = NULL;

    /* set remote IP address */
    FLib_MemCpy(&pSession->remoteAddr, &remoteAddress, sizeof(ipAddr_t));

    /* add options to message */
    COAP_SetUriPath(pTxSession, (coapUriPath_t*)&gURI_OPTION);

    /* application can change to Confirmable option if not unicast and set an ACK callback */
    if (!IP6_IsMulticastAddr(&gCoapDestAddress))
    {
        coapMessageType = gCoapMsgTypeConPost_c;
        pSession->pCallback = CoapCallback;
    }

    /* Send the COAP frame */
    COAP_Send(pSession, coapMessageType, pCoapPayload, payloadSize);
}
```

An example of callback called for an option follows:

```
static void APP_CoapLedCb
(
```

```
    coapSessionStatus_t sessionStatus,
    uint8_t *pData,
    coapSession_t *pSession,
    uint32_t dataLen
)
{
    /* Process the command only if it is a POST method */
    if((pData) && (sessionStatus == gCoapSuccess_c) && (pSession->code == gCoapPOST_c))
    {
        APP_ProcessLedCmd(pData, dataLen);
    }

    /* Send the reply if the status is Success or Duplicate */
    if((gCoapFailure_c != sessionStatus) && (gCoapConfirmable_c == pSession->msgType))
    {
        /* Send CoAP ACK */
        COAP_Send(pSession, gCoapMsgTypeAckSuccessChanged_c, NULL, 0);
    }
}
```

6.1 CoAP observe

CoAP Observe module is an extension of the CoAP core protocol. The source files are located at \middleware\wireless \nwk_ip_1.2.8\examples\common*app_coap_observe.c*.

A test application is delivered with the Observe module. The application source files are located at \middleware\wireless \nwk_ip_1.2.8\examples\common*app_observe_demo.c*. The application needs the Shell module to be enabled.

CoAP Observe works on a client and server model. The server is the authority for a specific resource/resources that changes its state/value. The client subscribes to a server to receive updates for a specific resource.

6.1.1 Enabling CoAP observe

CoAP Observe is by default disabled in all application projects. It can be enabled from all application configuration files, located at:

\middleware\wireless\nwk_ip_1.2.8\examples\<thread_application>\config\config.h

```
/* Enable CoAP Observe Client */
#define COAP_OBSERVE_CLIENT       1

/* Enable CoAP Observe Server */
#define COAP_OBSERVE_SERVER       1
```

The CoAP Observe Client and Server are started from the Shell command line. For projects that do not support Shell, call APP_ObserveStartClient() to initialize the Observe Client and/or APP_ObserveStartServer() to initialize the Observe Server. For more information about the interface functions, see the section below.

6.1.2 CoAP observe demo

This demo application shows how a resource that changes its state is reported from a server to a client. The server is the authority for the resource *"/number,* that is a random number generated once every twenty seconds. After each change of the resource, the server notifies its subscribers.

The client subscribes to the resource *"/number"* and receives a CoAP NON message after each state change of the resource.

On the server side, start the CoAP Observe server:

```
$ coap start observe server
 Success!
```

On the client side, start CoAP Observe Client, typing the server's IPv6 address and the resource.

```
$ coap start observe client fd01::3ead:4973:da79:49cd:521d /number

$ new value: 0xC01BFAB3 from fd01::3ead:4973:da79:49cd:521d
$ new value: 0xFAADB6CF from fd01::3ead:4973:da79:49cd:521d
$ new value: 0x1160E9EB from fd01::3ead:4973:da79:49cd:521d

$ coap stop observe client fd01::3ead:4973:da79:49cd:521d /number
$ new value: 0x5C261871 from fd01::3ead:4973:da79:49cd:521d
```

6.1.3 APIs for CoAP observe

To create an application that uses the CoAP Observe functionality, the following API is available.

6.1.3.1 CoAP observe server

```
uint8_t COAP_Server_InitObserve(ipIfUniqueId_t ipIfId, sockaddrStorage_t
                *pCoapStartUnsecParams, coapRegCbParams_t* pCallbacksStruct, uint32_t nbOfCallbacks)
```

This function starts a CoAP Observe server that runs on a given IP interface. It also creates the CoAP instance and registers a callback function for incoming messages with a specific resource.

The callback and the resource are defined as follows.

```
#define URI_OBSERVE_RESOURCE "/number"
        const coapUriPath_t gURI_OBSERVE_RESOURCE = {SizeOfString(URI_OBSERVE_RESOURCE),
            URI_OBSERVE_RESOURCE};
        #define COAP_OBSERVE_RESOURCE \
         {\
        {APP_ObserveServerRcvReqCb, (coapUriPath_t*)&gURI_OBSERVE_RESOURCE}\
        }
```

The server should keep a sequence number that must be incremented each time the resource is updated.

```
void COAP_Server_NotifyObservers(coapUriPath_t* pResource, coapMessageTypes_t
                coapMsgType, uint8_t sequenceId, uint8_t* pValue, uint8_t valueLen)
```

This function notifies all clients from the subscriber list for the specified resource. It sends a CoAP message (CON/NON) with the given payload, represented by the new value of the resource.

```
bool_t COAP_Server_AddObserver(coapSession_t* pSession, coapUriPath_t*
                pResource)
```

Adds an observer to the list of observers.

```
bool_t COAP_Server_RemoveObserver(coapSession_t* pSession, coapUriPath_t* pResource
                )
```

Removes an observer from the list of observers.

6.1.3.2 CoAP observe client

```
uint8_t COAP_Client_InitObserve(ipIfUniqueId_t ipIfId, sockaddrStorage_t*pCoapStartUnsecParams,
sockaddrStorage_t * pCallbacksStruct, uint32_t nbOfCallbacks)
```

This function starts a CoAP Observe client that runs on a given IP interface. It also creates the CoAP instance and registers a callback function for incoming messages with a specific resource.

```
void COAP_Client_StartObserving(ipAddr_t* pServerIpAddr, coapUriPath_t* pResource, coapCallback_t
rcvReplyCb)
```

Sends a CoAP GET message and asks for subscription.

```
void COAP_Client_StopObserving(ipAddr_t* pServerIpAddr, coapUriPath_t* pResource, coapCallback_t
rcvReplyCb)
```

Sends a CoAP GET message and removes the subscription.

Chapter 7
Socket Data APIs

The socket API allows creating generic BSD/POSIX-style IP sockets over the Thread IPv6 network and to exterior network destinations.

The following code block shows how to create a socket and send UDP data over the socket to an IPv6 destination address and port. Packets sent inside the Thread network need to be encrypted at the MAC layer. This is controlled at the socket layer by setting the flowinfo field with `BSDS_SET_TX_SEC_FLAGS(1,5)`, meaning key id mode 1 and security level 5. These are the standard MAC security settings for data packets inside a Thread network.

```
int32_t socketDescriptor = gBsdsSockInvalid_c;

socketDescriptor = socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDP);

if (socketDescriptor != gBsdsSockInvalid_c)
{
    sockaddrIn6_t socketAddrInfo;

    ipAddr_t IPV6_REMOTE_ADDRESS = { 0x20, 0x01, 0x00, 0x00, 0x00, 0x00, 0xD0, 0x00, \
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

    uint16_t remotePort = 1234;

    uint8_t dataToSend[] = { 0x41, 0x42, 0x43 };  // "ABC"

    uint8_t sendFlags = 0;

    socketAddrInfo.sin6_family = AF_INET6;
     IP_AddrCopy(&socketAddrInfo.sin6_addr, &REMOTE_ADDRESS);
     socketAddrInfo.sin6_port = remotePort;
    socketAddrInfo.sin6_scope_id = gIpIfSlp0_c;
    socketAddrInfo.sin6_flowinfo = BSDS_SET_TX_SEC_FLAGS(1, 5);

    sendto(socketDescriptor, dataToSend, sizeof(dataToSend), sendFlags, (sockaddrStorage_t *)
&socketAddrInfo, sizeof(sockaddrStorage_t));
}
```

The following code block shows how to create a UDP socket and bind it to any local IPv6 address and port:

```
/* application must track serverSocketDescriptor and serverSocketAddrInfo while the socket is needed
and release them after shutdown */

int32_t serverSocketDescriptor;

sockaddrIn6_t serverSocketAddrInfo;

...

/* for initialization and socket binding */

serverSocketDescriptor = socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDP);

if (serverSocketDescriptor != gBsdsSockInvalid_c)
{
    uint16_t localPort = 1234;
```

```
    serverSocketAddrInfo.sin6_family = AF_INET6;
     IP_AddrCopy(&serverSocketAddrInfo.sin6_addr, &in6addr_any);
     serverSocketAddrInfo.sin6_port = localPort;
    serverSocketAddrInfo.sin6_scope_id = gIpIfSlp0_c;
    serverSocketAddrInfo.sin6_flowinfo = BSDS_SET_TX_SEC_FLAGS(1, 5);

    bind(serverSocketDescriptor, (sockaddrStorage_t*) &serverSocketAddrInfo,
sizeof(sockaddrStorage_t));

    Session_RegisterCb(serverSocketDescriptor, ServerSessionCallback, &appThreadMsgQueue);
}

...

/* Data session callback */

void ServerSessionCallback(void *pPacket)
{
    sessionPacket_t *pSessionPacket = (sessionPacket_t*) pPacket;

    sockaddrIn6_t *pClientAddrInfo = (sockaddrIn6_t*) (&pSessionPacket->remAddr);

    /* Process data in pSessionPacket->pData */
    ...

    /* Must free the packets after processing */
    MEM_BufferFree(pSessionPacket->pData);

    MEM_BufferFree(pSessionPacket);
}

...

/* Close socket and free resources */
Session_UnRegisterCb(serverSocketDescriptor);
closesock(serverSocketDescriptor);
```

Further examples of using of the socket API are contained in the **\middleware\wireless\nwk_ip_1.2.8\examples\common \app_socket_utils.h and .c files**.

Chapter 8
Low Power End Device Provisioning

Low Power end devices examples are set to automatically enable the low power modes of the microcontrollers using the following settings (as shown in **\middleware\wireless\nwk_ip_1.2.8\examples\low_power_end_device\config\config.h**.

```
#define gLpmIncluded_d            1
#define cPWR_UsePowerDownMode     1
```

See the configuration file at **\middleware\wireless\framework_5.3.8\LowPower\Interface\<PLATFORM>,**

for example,

\middleware\wireless\nwk_ip_1.2.8\examples\common\app_framework_config.h for a configuration of the power down modes. The demo examples are using mode **3** as set by macro define: **cPWR_DeepSleepMode**.

Chapter 9
Revision History

Table 1. Revision history on page 35 summarizes revisions to this document.

Table 1. Revision history

Revision number	Date	Substantive changes
0	03/2016	Initial release
1	04/2016	Updates for the Thread KW41 Alpha Release
2	08/2016	Updates for the Thread KW41 Beta Release
3	09/2016	Updates for the Thread KW41 GA Release
4	03/2017	Updates for the Thread KW41 MCUX GA Release
5	01/2018	Updates for the Thread KW41 Maintenance Release
6	05/2018	Updates for Thread K3S Beta Release.
7	04/2019	Updates for KW41Z Maitenance Release 3.

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, All other product or service names are the property of their respective owners. Arm, AMBA, Arm Powered, are registered trademarks of Arm Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 08 April 2019
Document identifier: KTSADUG

arm