NXP Semiconductors
User's Guide

Document Number: BLEMESHUG
Rev. 1, 09/2016

Kinetis Bluetooth® Low Energy Mesh User's Guide

1 Introduction

This document describes the Kinetis Bluetooth® Low Energy (BLE) Mesh stack and provides the information required to develop a mesh application on the Kinetis Bluetooth Low Energy Host Stack.

The document introduces the mesh stack, the network functionality, packet format, security, and application layer philosophy. It also explains the main initialization APIs and the application lifecycle for a mesh device and a mesh commissioner.

The document briefly introduces the mesh profiles implemented by the stack, such as the Configuration Profile, the Light Profile, and the Temperature Profile. Finally, the document describes the mesh functionality and application level associations.

Contents

2 Mesh Overview

The Kinetis BLE Mesh stack defines protocols that enable the creation of a mesh network with Bluetooth-enabled devices. The Bluetooth 4.0+ specification needs to be supported to send and receive packets using this technology.

2.1 Mesh operation

The BLE Mesh stack allows devices that have joined the network, mesh nodes, to exchange data in a connectionless manner. Instead of creating a connection using the GAP Central-Peripheral interaction and exchanging data over GATT or L2CAP, a mesh node sends data by placing it in an advertising packet. Another mesh node receives the data by scanning for advertisement.

Furthermore, nodes that are not in range are able to communicate through packet forwarding or mesh relaying. A relay node that receives a packet, whose destination is not its own address, retransmits it. Through a series of retransmissions, a packet may reach a destination that may be multiple hops away from the source. Based on the relay capability, two topological roles are defined for a mesh network.

- Leaf node – a node that sends and receives application messages contained in mesh packets; it silently drops packets intended for other nodes (does not forward packets).
- Relay node – a node that, in addition to processing its own packets, relays or forwards packets intended for other nodes.

Because a mesh packet is always broadcast to all nodes in range and all relays are mandated to retransmit it towards a destination, the mesh protocol is flood-based. In other words, a packet reaches the destination on multiple paths. This gives the mesh network useful redundancy while keeping a low complexity of the network protocol.

2.2 Mesh addresses

Mesh addresses are 14-bit values that identify nodes (MSB is 0) and groups of nodes (MSB = 1). They are divided into three groups as follows.

1. Unicast addresses, with range 0x0001 – 0x1FFF.
2. Multicast addresses, with range 0x2000 – 0x3FFE.
3. The Broadcast Address, 0x3FFF.

2.3 Packet format

The packet format is shown in the figure below.

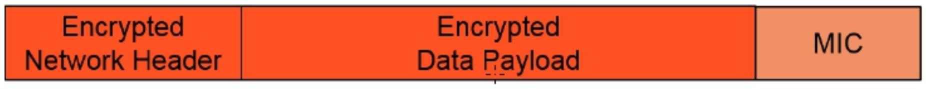


Figure 1. Packet format

All security is based on a shared, secret Network Key, which is a 128-bit key randomly generated by the commissioner and distributed to all nodes during commissioning. The Network Header is encrypted in AES128-ECB mode, using a Header Encryption Key, derived from the Network Key.

The Data Payload is encrypted in AES128-CCM with a 4-byte MIC, appended at the end of the packet. The CCM key is the Payload Encryption Key, which is also derived from the Network Key. The CCM nonce is constructed based on the unencrypted header and a network shared secret. The complete header format is illustrated in the figure below.

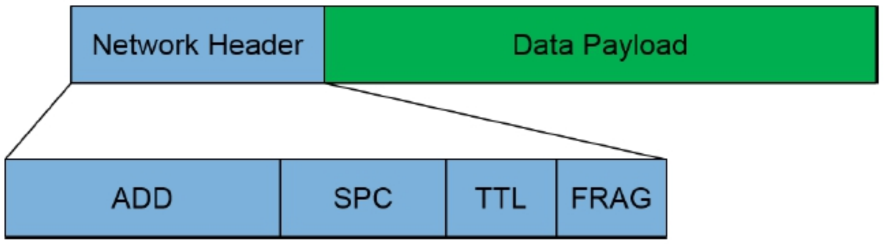


Figure 2. Header format

The ADD field contains addressing information, such as the network identifier, source and destination addresses. The SPC field is a Source Packet Counter, which is used to perform network layer duplicate filtering and protecting against replay attacks. The TTL field contains the Time-To-Live value of the packet, which controls the flood.

The FRAG field contains fragmentation information and is used to segment and reassemble large packets into the available payload size. The format of the data payload is illustrated in the figure below.

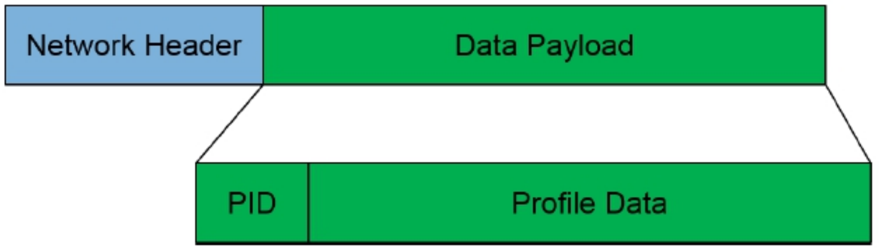


Figure 3. Payload format

The PID field is a one-byte Profile Identifier. The Profile Data is formatted according to the PID value.

2.4 Security

The figure below shows the encryption of a mesh packet, as performed by a sending node.

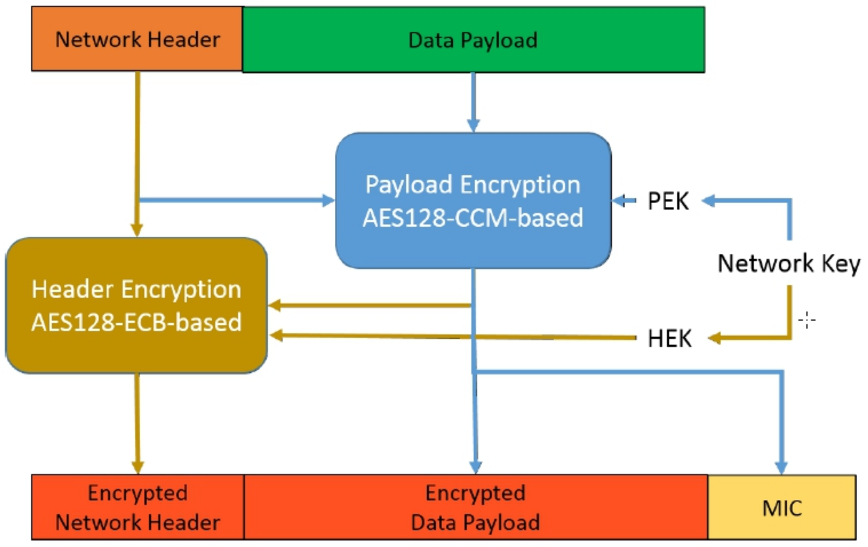


Figure 4. Packet encryption

The Network Header is used in the AES128-CCM nonce when encrypting the Data Payload. PEK is the Payload Encryption Key, which is derived from the Network Key. The Encrypted Data Payload and MIC are placed in the output packet and used as input in the header encryption block, where they are used together with HEK, the Header Encryption Key, to get the Encrypted Network Header.

The figure below shows the decryption of an encrypted mesh packet, as performed by a receiving node.

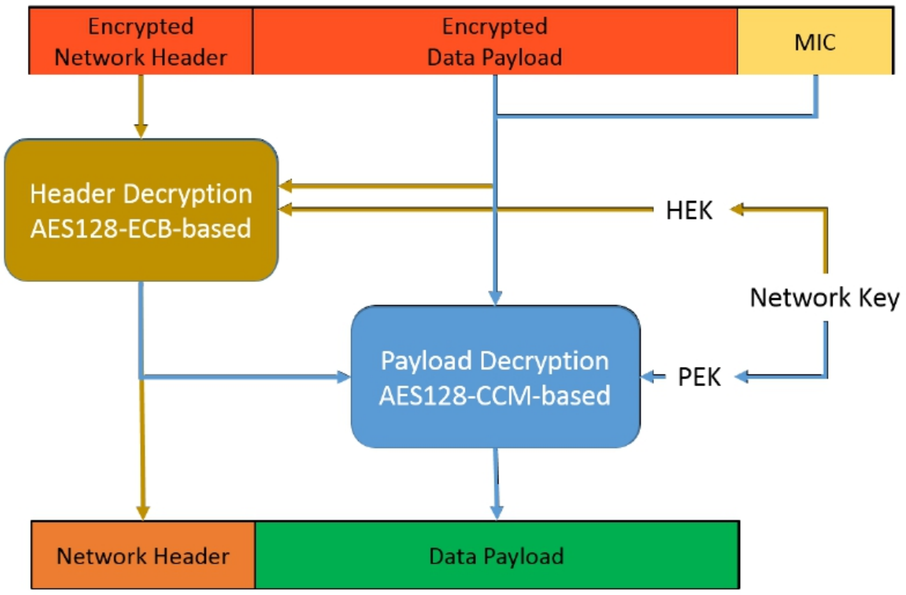


Figure 5. Packet decryption

The Encrypted Data Payload and MIC are fed into the header decryption block where they are used together with the HEK to get the original Network Header.

The Network Key needs to be commissioned to a new node in a secure manner. For a device supporting Bluetooth 4.2+, the LE Secure Connections pairing is recommended for commissioning. For older devices, the application must provide another secure mechanism, which can be OOB or in-band with strong encryption.

The LE Secure Connections pairing provides the most effective commissioning link security because it uses the Elliptic Curve Cryptography strength. The Diffie-Hellman Key resulting from the ECDH procedure is stored as a Node Key on the commissioner and can be used to encrypt one-to-one communication between the commissioner and the node. For example, this is useful when performing a full network key renewal procedure while blacklisting some of the nodes considered possible security holes.

2.5 Publish-subscribe

A mesh packet may have a destination that consists of a single node, a group of nodes, or all nodes. Respectively, this implies unicast, multicast, or broadcast communication.

To send a multicast message, the destination field is set to a multicast address. Receiving and processing these packets is defined under a publish-subscribe paradigm, which states that a node must be subscribed to a multicast address, or a publish address, to process the application message contained by the packet.

A node may have one publish address per profile, for example, a publish address for Light messages and another publish address for Temperature messages. Likewise, a node may have one subscription list per profile, which is a collection of publish addresses that the node listens to for that specific profile.

Setting a publish address or subscribing to one is performed either by the local application through generic configuration APIs, as described in Local node configuration, or by a remote Configuration Client, as defined by the Configuration Profile and described in Configuration client.

When a node needs to send a message, the application decides if the destination is a single node or a group of nodes. Example scenarios of the publish-subscribe mechanism can be found in Publish-subscribe on the Light Profile.

3 Main Interface

The main API functions are declared in the mesh_interface.h file. All macros and data types are defined in the mesh_types.h file. These functions are used for stack initialization, commissioning, local node configuration, and custom data transmission.

3.1 Mesh stack initialization

Two separate functions are provided to initialize either a mesh device, commissioned or uncommissioned, or a mesh commissioner.

3.1.1 Mesh device

The following function can be used to initialize a mesh device.

```
meshResult_t MeshNode_Init
(
  meshGenericCallback_t callback
);
```

The generic callback is used by the stack to deliver generic mesh events to the application and has the following prototype.

```
typedef meshResult_t (*meshGenericCallback_t)
(
  meshGenericEvent_t* pEvent
);
```

The following generic event types are defined.

```
// Generic mesh event types.
typedef enum meshGenericEventType_tag
{
  gMeshInitComplete_c, // Mesh stack has been initialized.
  gMeshCustomDataReceived_c, // Custom application data has been received.
} meshGenericEventType_t;
```

After the initialization is complete, a gMeshInitComplete_c generic event is generated. The following example implementation shows the information contained by this event.

```
static meshResult_t MeshGenericCallback
(
meshGenericEvent_t* pEvent
)
{
  switch (pEvent->eventType)
  {
    case gMeshInitComplete_c:
    {
      if (pEvent->eventData.initComplete.deviceIsCommissioned)
      {
        /* Perform mesh operations */
      }
      else
      {
        /* Prepare for commissioning */
      }
    }
      break;
  }
return gMeshSuccess_c;
}
```

On the very first initialization, the device is not commissioned. After commissioning, the data is saved in non-volatile memory. Therefore, on subsequent initializations, for example upon reset, the deviceIsCommissioned field is set to TRUE. The gMeshCustomDataReceived_c event is generated when the custom application data is received. See Section 3.4 for more information.

3.1.2 Mesh comissioner

Similarly, the following API initializes the mesh stack for a mesh commissioner.

```
meshResult_t MeshCommissioner_Init
(
  meshGenericCallback_t genericCallback
);
```

The callback has the same prototype and the same gMeshInitComplete_c event is triggered. The pEvent->eventData.initComplete.deviceIsCommissioned value is always TRUE because, on the first initialization, the entire commissioning data is internally generated and saved in non-volatile memory.

3.1.3 Non-volatile memory access

The mesh stack defines three functions as external references to be linked by the application with platform-specific NVM code. These functions are used by the stack to save and load commissioning and configuration data.

```
extern void App_NvmErase
(
  void
);
extern void App_NvmWrite
(
  void* pvRamSource,
  uint32_t cDataSize
);
extern void App_NvmRead
(
  void* pvRamDestination,
  uint32_t cDataSize
);
```

3.2 Comissioning

Commissioning consists of three phases as follows.

1. The commissioner application calls the following API to receive commissioning data from the stack.

```
meshResult_t MeshCommissioner_GetNextCommissioningData
(
  meshRawCommissioningData_t* pOutRawCommissioningData
);
```

2. The commissioner and the un-commissioned device share a secure link over which the commissioning data can be exchanged. The recommended method is to form a GATT connection between the devices, perform LE Secure Connections pairing, and have the commissioner send data over GATT.
3. The device application calls the following API to commission the node internally using the data it has received over the secure link.

```
meshResult_t MeshNode_Commission
(
  meshRawCommissioningData_t* pRawCommissioningData
);
```

The commissioning process with LE Secure Connections Pairing is shown in the figure below.

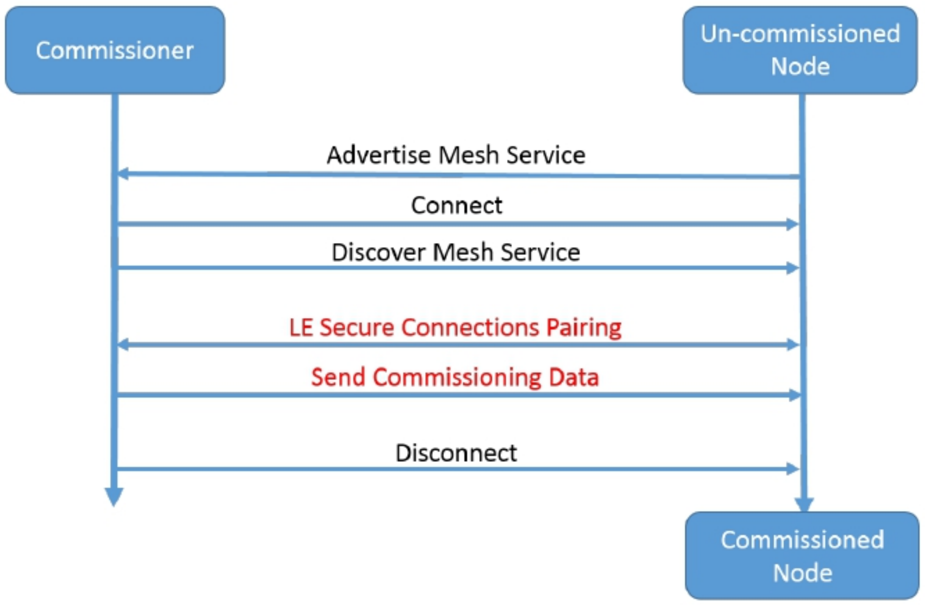


Figure 6. Commissioning over LE Secure Connections

3.3 Local node configuration

The following parameters can be configured for the local mesh node.

- Relay state
  - A Boolean that indicates whether or not the local node performs packet relaying.
- TTL value
  - The TTL value that the node uses whenever it sends a mesh packet.
- Publish address

    - A node can have a publish address for each supported mesh profile. See Section 2.5 for more information.
- Subscription list
    - A node can subscribe to one or more publish addresses for each supported profile. See Section 2.5 for more information.

Each parameter has a "Set" and a "Get" API in the mesh_interface.h file.

3.4 Custom data transmission

Although the stack offers mesh profile implementations for the most common profiles, an API is provided send custom application data to a peer mesh node.

```
meshResult_t Mesh_SendCustomData
(
  meshAddress_t destination,
  meshCustomData_t* pData
);
```

When the peer node receives the data, it signals the application through a generic mesh event of type gMeshCustomDataReceived_c, which contains the data and the source address.

4 Profiles

Mesh profiles organize data sent over the mesh and provide a user-friendly interface for a Client-Server paradigm. There are currently three profiles defined in the stack, the Configuration profile, the Light profile, and the Temperature profile.

4.1 Configuration profile

The Configuration profile APIs allow a Configuration Client to remotely control the node’s configuration through the interaction with a Configuration Server running on the node. They are declared in the mesh_config_client.h and mesh_config_server.h interface files.

4.1.1 Configuration client

To act as a Configuration Client, the application must install the configuration client callback using the following API.

```
meshResult_t MeshConfigClient_RegisterCallback
(
  meshConfigClientCallback_t callback
);
```

The following client event types are defined.

```
/*! Configuration client event types. */
typedef enum meshConfigClientEventType_tag
{
  gMeshConfigReceivedRelayState_c,
  /*! Relay state has been reported by a configuration server. */
  gMeshConfigReceivedTtl_c,
  /*! TTL value has been reported by a configuration server. */
  gMeshConfigReceivedPublishAddress_c,
  /*! Publish address has been reported by a configuration server. */
  gMeshConfigReceivedSubscriptionList_c,
  /*! Subscription list has been reported by a configuration server. */
} meshConfigClientEventType_t;
```

The Client APIs allow configuration of all node parameters described in Section 3.3. “Setting” and “getting” a parameter on the remote node triggers a response that generates an event. For example, the following API sets the relay state on a node.

```
meshResult_t MeshConfigClient_EnableRelay
(
  meshAddress_t destination,
  bool_t enable
);
```

After the peer receives the command and sets the requested relay state, it sends back a response with the new state, and the local application receives a gMeshConfigReceivedRelayState_c event, which contains the state and the peer node’s address. The same event is triggered if the application calls the “get” API.

```
meshResult_t MeshConfigClient_GetRelayState
(
  meshAddress_t destination
);
```

The same rules apply to all other Configuration Client APIs.

4.1.2 Configuration server

A Configuration Server instance is running by default on any node. Remote commands are handled internally and responses are sent automatically for this profile. To be informed of any parameter changes commanded by a remote client, the application may install the configuration server callback using the following API.

```
meshResult_t MeshConfigServer_RegisterCallback
(
  meshConfigServerCallback_t callback
);
```

The following informative server event types are defined.

```
/*! Configuration server event types. */
typedef enum meshConfigServerEventType_tag
{
  gMeshConfigRelayStateChanged_c,
  /*! Local relay state has been changed by the configuration client. */
  gMeshConfigTtlChanged_c,
  /*! Local TTL value has been changed by the configuration client. */
  gMeshConfigPublishAddressChanged_c,
  /*! Local publish address has been changed by the configuration client. */
  gMeshConfigSubscriptionListChanged_c,
  /*! Local subscription list has been changed by the configuration client. */
} meshConfigServerEventType_t;
```

The event data always contains the new parameter values.

4.2 Light profile

The Light profile APIs allow a Light Client to control and configure a remote Light Server running on a node with lighting capabilities, as well as to receive reports about the light state on the node hosting the Light Server. They are declared in the mesh_light_client.h and mesh_light_server.h interface files.

4.2.1 Light client

This role can be used by a light switch or a light monitor, for example. To act as a Light Client, the application must install the light client callback using the following API.

```
meshResult_t MeshLightClient_RegisterCallback
(
  meshLightClientCallback_t callback
);
```

The following client event types are defined.

```
/*! Light client event types. */
typedef enum meshLightClientEventType_tag
{
  gMeshLightReceivedLightState_c,
  /*! Light state has been reported by a light server. */
  gMeshLightReceivedReportState_c,
  /*! Light report state has been reported by a light server. */
} meshLightClientEventType_t;
Most of the Client APIs act on or read the peer node’s light state. For example:
meshResult_t MeshLightClient_ToggleLight
(
  meshAddress_t destination
);
```

With the publish-subscribe model, it is possible to act on multiple light nodes by publishing the toggle command.

```
meshResult_t MeshLightClient_PublishToggleLight();
```

This requires that the node sets a publish address for the light model through a local application configuration, as follows.

```
Mesh_SetPublishAddress
(
  gMeshProfileLighting_c,
  publishAddress
);
```

Alternatively, the node can set a publish address for the light model through a remote configuration from the Configuration Client.

4.2.2 Light server

This role can be used by a light bulb, for example. To act as a Light Server, the application must install the light server callback using the following API.

```
meshResult_t MeshLightServer_RegisterCallback
(
  meshLightServerCallback_t callback
);
```

The following server event types are defined.

```
/*! Light server event types. */
typedef enum meshLightServerEventType_tag
{
  gMeshLightSetCommand_c,
  /*! Received light set command from a light client. */
  gMeshLightGetCommand_c,
  /*! Received light get command from a light client. */
  gMeshLightToggleCommand_c,
  /*! Received light toggle command from a light client. */
  gMeshLightSetReportCommand_c,
  /*! Received light report set command from a light client. */
  gMeshLightGetReportCommand_c,
  /*! Received light report get command from a light client. */
} meshLightServerEventType_t;
```

To receive light commands that are published by Light Clients, the Server needs to be subscribed to the publish address with the local command as follows.

```
Mesh_Subscribe
(
  gMeshProfileLighting_c,
  publishAddress
);
```

The server can also be subscribed to through a remote configuration from a Configuration Client. The response can be sent either to the Client that sent the command or published to all nodes that have subscribed to its publish address. The latter can be achieved with the following function.

```
meshResult_t MeshLightServer_PublishState
(
  bool_t lightState
);
```

4.3 Temperature profile

The Temperature profile APIs allow a Temperature Client to control and configure a remote Temperature Server running on a node with temperature reading capabilities and to receive reports about the temperature measured by the node hosting the Temperature Server. They are declared in the mesh_temperature_client.h and mesh_temperature_server.h interface files.

4.3.1 Temperature client

This role can be used by, for example, a temperature monitor. To act as a Temperature Client, the application must install the temperature client callback using the following API.

```
meshResult_t MeshTemperatureClient_RegisterCallback
(
  meshTemperatureClientCallback_t callback
);
```

The following client event types are defined.

```
/*! Temperature client event types. */
typedef enum meshTemperatureClientEventType_tag
{
  gMeshTemperatureReceivedTemperature_c,
  /*! Temperature has been reported by a temperature server. */
  gMeshTemperatureReceivedReportState_c,
  /*! Temperature report state has been reported by a temperature server. */
} meshTemperatureClientEventType_t;
```

Similarly to the Light profile, these APIs allow reading temperature values from remote Temperature Servers.

4.3.2 Temperature server

This role can be used by, for example, a temperature sensor. To act as a Temperature Server, the application must install the temperature server callback using the following API.

```
meshResult_t MeshTemperatureServer_RegisterCallback
(
  meshTemperatureServerCallback_t callback
);
```

The following server event types are defined.

```
/*! Temperature server event types. */
typedef enum meshTemperatureServerEventType_tag
{
  gMeshTemperatureGetCommand_c,
  /*! Received temperature get command from a temperature client. */
  gMeshTemperatureSetReportCommand_c,
  /*! Received temperature set report command from a temperature client. */
  gMeshTemperatureGetReportCommand_c,
  /*! Received temperature get report command from a temperature client. */
} meshTemperatureServerEventType_t;
```

When a temperature get command is received, the Server may reply either to the Client requesting the value, or it can publish the temperature for the subscribers.

5 Demo Applications

The Kinetis BLE Mesh stack is shipped with two demo applications that can be used as a starting point for building binaries that can be uploaded on KW41Z boards to create one or more mesh networks

The mesh_commissioner application features a terminal-based setup that can be used to dynamically configure nodes within a network using terminal commands. The commissioner itself is also a node in the network and implements the Light Switch functionality.

The mesh_device application is used to create a regular mesh node that can implement one of the available profiles (Light and Temperature) and custom application profiles.

5.1 Mesh Commissioner application

The Mesh Commissioner application instantiates a node with the mesh address 0x0001. This address is fixed inside the library and cannot be changed from the application. The demo application is configurable with regards to the network key and the terminal functionality.

5.1.1 Network key

The network key is defined in the app_config.c file:

```
const uint8_t gMeshNetworkKey[gMeshKeySize_c] =
{
    0xa2, 0x35, 0xda, 0xaa, 0x56, 0xc0, 0xba, 0x21, 0x47, 0x88, 0x91, 0x9f, 0x2c, 0x77,
    0x02, 0x9e
};
```

The network key must be the same for all nodes within a single network. Change the key to build nodes for multiple networks.

5.1.2 Terminal commands

The app.c file contains the application implementation that deals with terminal commands. The commissioner application is configured as Client for all the available profiles:

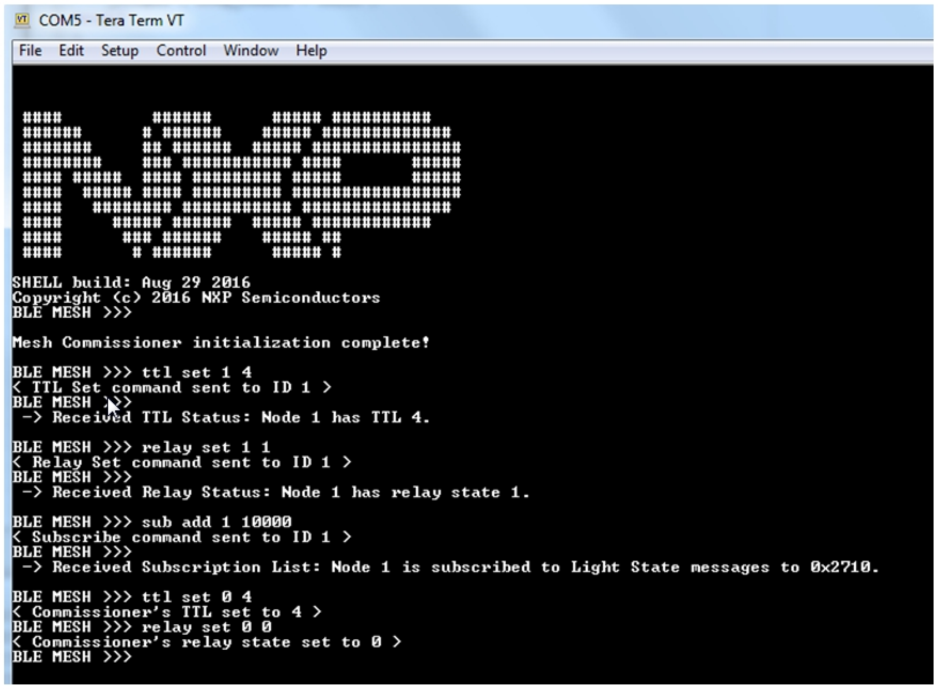
```
#include "mesh_interface.h"
#include "mesh_config_client.h"
#include "mesh_light_client.h"
#include "mesh_temperature_client.h"
```

The following terminal commands are used to send Configuration Client commands:

- *Pub* – get/set the Publish Address on a node.
- *Sub* – get/add/remove addresses from the Subscription List on a node.
- *Relay* – get/set the relay state on a node. When this is set to 1, the node becomes a relay and forwards incoming packets. When set to 0, the node acts as a leaf node and does not forward packets.
- *Ttl* – get/set the outgoing TTL value on a node. This value is set in the TTL field of all mesh packets sent by the node.

The *Light command* is used to send Light Client commands. All nodes are addressed with an ID that ranges from 1 to 255. The ID is converted to a mesh address by adding it to the the base 0x0100 address. For example, the node ID 1 corresponds to the mesh address 0x0101. For local configuration (for example, the local subscription list), the ID 0 is used.

For example, the command *ttl set 0 5* sets the value 5 for the TTL of the local node and the command *ttl set 10 5* send a Configuration Client Set TTL command to the node with an address 0x010A. The figure below shows commands entered in the terminal to configure the commissioner node and the node with ID 1.



```
SHELL build: Aug 29 2016
Copyright (c) 2016 NXP Semiconductors
BLE MESH >>>

Mesh Commissioner initialization complete!

BLE MESH >>> ttl set 1 4
< TTL Set command sent to ID 1 >
BLE MESH >>>
 -> Received TTL Status: Node 1 has TTL 4.

BLE MESH >>> relay set 1 1
< Relay Set command sent to ID 1 >
BLE MESH >>>
 -> Received Relay Status: Node 1 has relay state 1.

BLE MESH >>> sub add 1 10000
< Subscribe command sent to ID 1 >
BLE MESH >>>
 -> Received Subscription List: Node 1 is subscribed to Light State messages to 0x2710.

BLE MESH >>> ttl set 0 4
< Commissioner's TTL set to 4 >
BLE MESH >>> relay set 0 0
< Commissioner's relay state set to 0 >
BLE MESH >>>
```

Figure 7. Terminal-based configuration

5.2 Mesh Device application

The Mesh Device application instantiates a mesh node with a configurable mesh address, network key, and functionality.

5.2.1 Mesh address

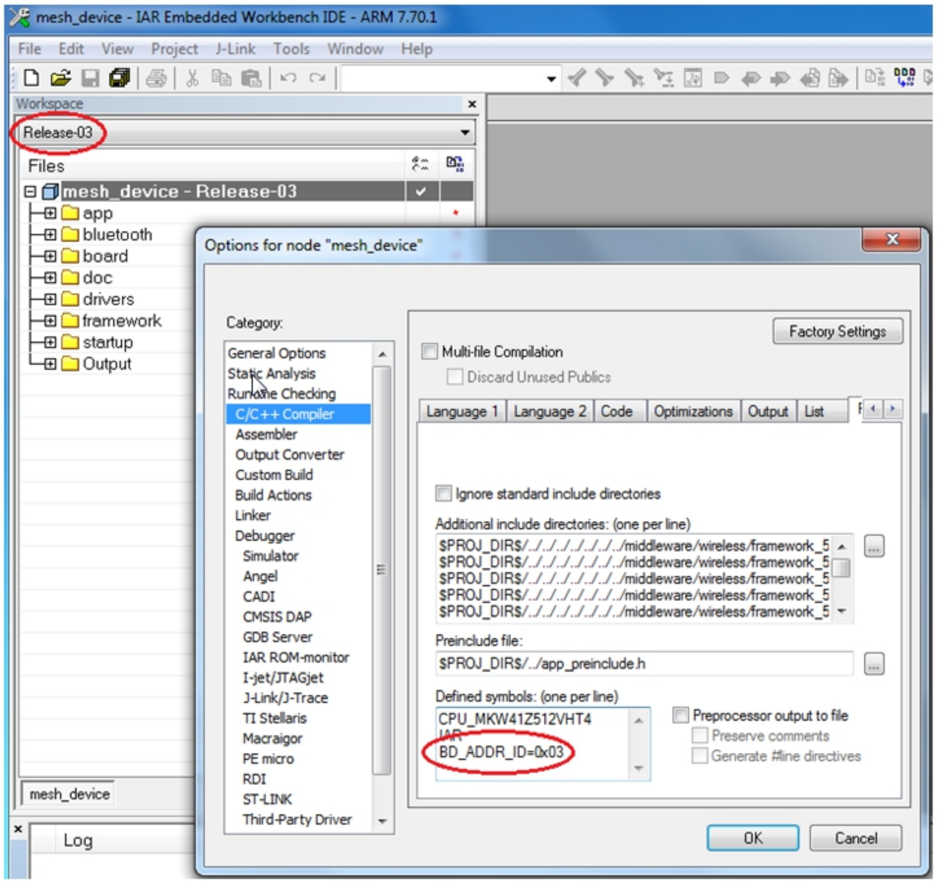
The node’s mesh address is computed by adding the base address 0x0100 with the BD_ADDR_ID value.

NOTE

The BD_ADDR_ID value must be unique for each new node added in the same network!
The BD_ADDR_ID value is, by default, set to 0x01in the app_preinclude.h file:

```
#ifndef BD_ADDR_ID
#define BD_ADDR_ID      0x01
#endif
```

When building multiple nodes, it is recommended not to modify this value directly, but rather overwrite it in Compiler -> Preprocessor -> Defined symbols. The valid range is 0x01 – 0xFF. Defining it in this manner allows creating multiple build configurations for the project and, in each configuration, the BD_ADDR_ID value can be defined in the preprocessor area. An example of such a build configuration is shown in the figure below.



Workspace: Release-03

Multi-file Compilation
Discard Unused Publics
Ignore standard include directories

Additional include directories: (one per line)

```
$PROJ_DIR$/../../../../../../../middleware/wireless/framework_5
$PROJ_DIR$/../../../../../../../middleware/wireless/framework_5
$PROJ_DIR$/../../../../../../../middleware/wireless/framework_5
$PROJ_DIR$/../../../../../../../middleware/wireless/framework_5
$PROJ_DIR$/../../../../../../../middleware/wireless/framework_5
```

Preinclude file: $PROJ_DIR$/../app_preinclude.h

Defined symbols: (one per line)

```
CPU_MKW41Z512VHT4
IAR
BD_ADDR_ID=0x03
```

Preprocessor output to file
Preserve comments
Generate #line directives

Figure 8. Creating a build configuration for a node with ID 3

The BD_ADDR_ID value is also included in the raw commissioning data defined in the app_config.c. This data is intended for demo purposes only and should not be modified.

By changing the BD_ADDR_ID, large networks can be built and not all nodes must be in radio range of each other given the packet forwarding performed by relay nodes.

5.2.2 Network key

The network key is defined in the app_config.c file:

```
const uint8_t gMeshNetworkKey[gMeshKeySize_c] =
{
    0xa2, 0x35, 0xda, 0xaa, 0x56, 0xc0, 0xba, 0x21, 0x47, 0x88, 0x91, 0x9f, 0x2c, 0x77,
    0x02, 0x9e
};
```

This network key must be the same for all nodes within a single network. Change the key to build nodes for multiple networks.

5.2.3 Supported profiles

In the app.h file, the following preprocessor defines are used to configure application functionality that is based on the Light Profile and the Temperature Profile:

```
/* App Configuration */
#define gAppLightSwitch_d       0
#define gAppLightBulb_d         1
#define gAppTempSensor_d        1


/* Consistency check */
#if gAppLightSwitch_d && gAppLightBulb_d
#error "Please define only one of the light roles!"
#endif
```

To build a light switch, set the gAppLightSwitch_d value to 1 and set the gAppLightBulb_d value to 0. To build a light bulb, these values are reversed. To build a temperature sensor that uses the Temperature Server functionality, the gAppTempSensor_d can be set to 1. Based on these definitions, the application code from the app.c source file includes the required header and performs different operations:

```
#include "mesh_interface.h"

#if gAppLightBulb_d
#include "mesh_light_server.h"
#elif gAppLightSwitch_d
#include "mesh_light_client.h"
#endif

#if gAppTempSensor_d
#include "mesh_temperature_server.h"
#endif
```

This code can be modified based on application needs. For example, the demo code refers the following function that is implemented as a stub:

```
static void UpdateLightUI(bool_t lightOn)
{
    /* Insert code here to update UI to indicate Light State ON/OFF. */
}

This function is called whenever a Light Client triggers a light state change or a local
button is pressed and should contain code to modify the user interface (e.g., turn a LED on
or off).
```

5.2.4 Pairing light switches and light bulbs

By default, a light switch application sends Light Toggle commands whenever a button is pressed, by calling the MeshLightClient_PublishToggleLight in the keypress handler function. In order to pair a light switch with a light bulb, the following two configuration steps must be taken:

1. Set a Publish Address on the switch. This can be done in two ways:
    - Statically (at compile time) by calling the Mesh_SetPublishAddress function declared in mesh_interface.h.
    - Dynamically (at runtime) by issuing a configuration command from the commissioner’s terminal (*pub set ID address*).
2. Susbcribe the light bulb to that publish address. This can also be done in two ways:
    - Statically (at compile time) by calling the Mesh_Subscribe function declared in mesh_interface.h.
    - Dynamically (at runtime) by issuing a configuration command from the commissioner’s terminal (*sub add ID address*).

Multiple light bulbs can be paired with the same light switch. Also, the light switch code can be modified to send Light On when pressing one button and Light Off when pressing the other button. The Publish Address must be a valid multicast address, in other words, in the range 0x2000 – 0x3FFE, which in the terminal corresponds to the decimal range *8192 – 16382*. See the examples in the next section of the document.

6 Mesh Packet Forwarding Example

The figure below shows a mesh network with four relay nodes and three leaf nodes associated with addresses 1 to 7. If node one needs to send a message to node seven, it builds the packet header with the following field values.

- Source (SRC) set to its own address, in this case 1
- Destination (DST) is set to, in this case 7
- The TTL value is set to 4

In direct radio range of node one are nodes two, three, and four.

Node four is another leaf node. After decryption, because the destination address does not match its own address (which is 4), it silently drops the packet and ignores all its future instances.

Nodes two and three are relays, which means that they forward the packet towards its destination after decreasing the TTL to 3. All other fields remain untouched and the packet is reencrypted because only one field has changed.

Node five is also a relay node. It forwards the packet received from node two, with the new TTL of 2. Similarly, node six decrements the TTL and resends the packet received from node three.

Finally, node seven receives the packet from nodes five and six. The first packet that arrives is processed and the second one is silently dropped.

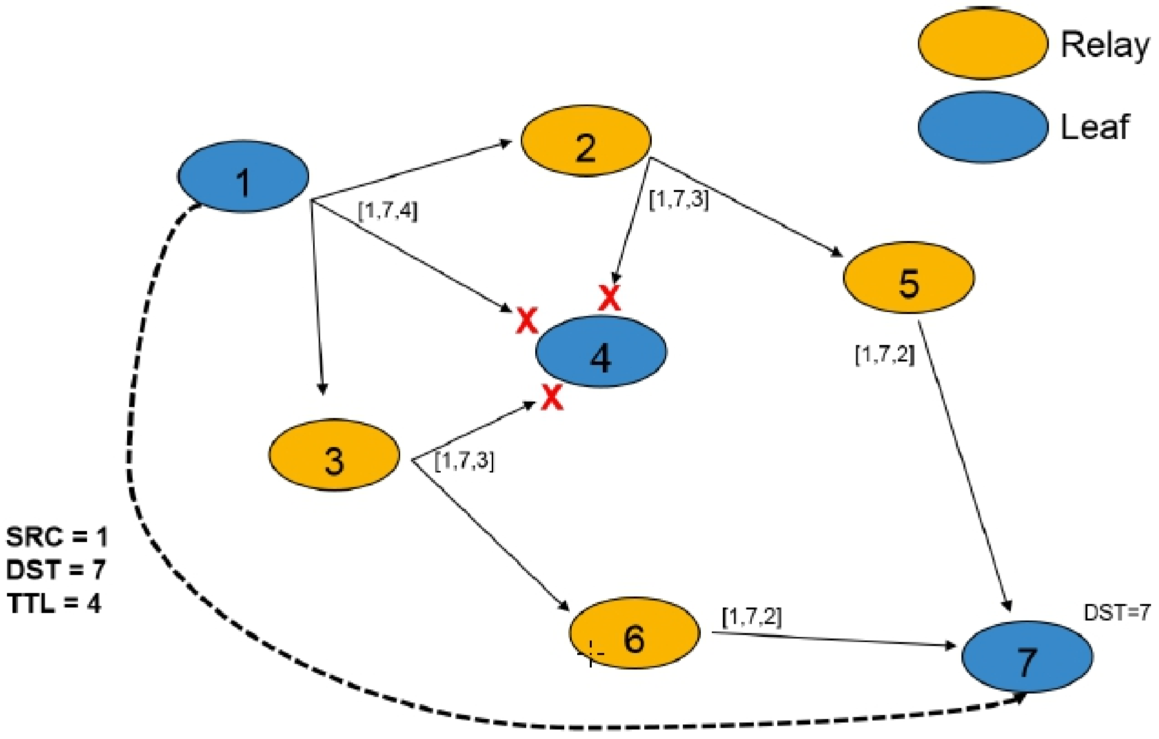


Figure 9. Mesh packet forwarding

6.1 Publish-subscribe on the Light Profile

The following examples contain five nodes, a light switch (address 0x0011), three light bulbs (addresses 0x0021, 0x0022 and 0x0023), and the commissioner.

6.1.1 Toggle all lights

The idea is to match the switch with the light bulbs such that, when the switch is acted upon, it toggles all light bulbs simultaneously. The application running on the light switch is configured to publish the Toggle Light message every time the switch is acted upon.

```
void SwitchHandler()
{
  MeshLightClient_PublishToggleLight();
}
```

The application running on the light bulbs is configured to act on the UI (the actual light circuit) when the Toggle Light message is received.

```
meshResult_t MeshLightServerCallback
(
meshLightServerEvent_t* pEvent
)
{
  switch (pEvent->eventType)
  {
    ...
    case gMeshLightToggleCommand_c:
    {
      mLightState = !mLightState;
      UpdateLightUI(mLightState);
    }
    break;
    ...
  }
}
```

To achieve the desired scenario, the commissioner acts as a Configuration Client and sends configuration messages to the Configuration Servers that are instantiated by default on the other four nodes.

- A “Set Publish Address” message is sent to the light switch with the multicast address 0x2001and the Light Profile identifier as parameters.
  - MeshConfigClient_SetPublishAddress (0x0011, gMeshProfileLighting_c, 0x2001);
- A “Subscribe” message is sent to the light bulbs with the same parameters.
  - MeshConfigClient_Subscribe (0x0021, gMeshProfileLighting_c, 0x2001);
  - MeshConfigClient_Subscribe (0x0022, gMeshProfileLighting_c, 0x2001);
  - MeshConfigClient_Subscribe (0x0023, gMeshProfileLighting_c, 0x2001);

The scenario is shown in the figure below.

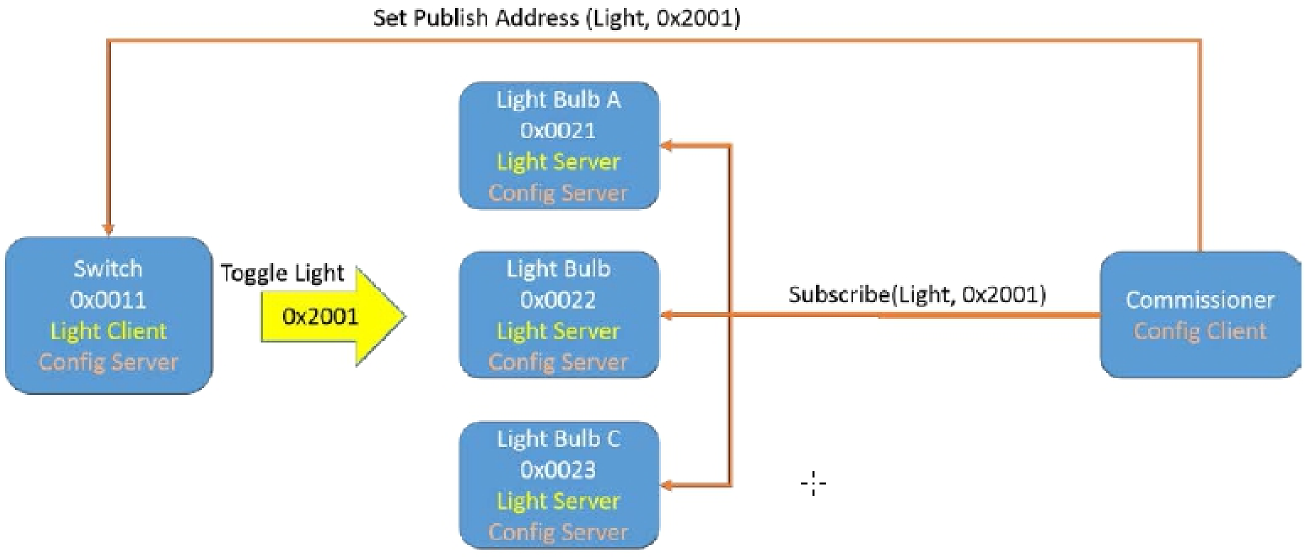


Figure 10. Light Toggle message published by a light sw

6.1.2 Monitor the lights

Building on the previous scenario, more functionality is needed. The commissioner needs to always be informed about all light state changes. For example, the commissioner may have a display indicating the light states.

To achieve this, the application running on the light bulbs is modified to publish the light state every time it changes (extra line added in red).

```
meshResult_t MeshLightServerCallback
(
  meshLightServerEvent_t* pEvent
)
{
  switch (pEvent->eventType)
  {
    ...
    case gMeshLightToggleCommand_c:
    {
      mLightState = !mLightState;
      UpdateLightUI(mLightState);
      MeshLightServer_PublishState(mLightState);
    }
    break;
    ...
  }
}
```

Then, the commissioner application sets the publish address 0x2002 on the light bulbs and subscribes itself to this address.

```
MeshConfigClient_SetPublishAddress ( 0x0021, gMeshProfileLighting_c, 0x2002 );
MeshConfigClient_SetPublishAddress ( 0x0022, gMeshProfileLighting_c, 0x2002 );
MeshConfigClient_SetPublishAddress ( 0x0023, gMeshProfileLighting_c, 0x2002 );
Mesh_Subscribe (gMeshProfileLighting_c, 0x2002 );
```

The resulting scenario is shown in the figure below.

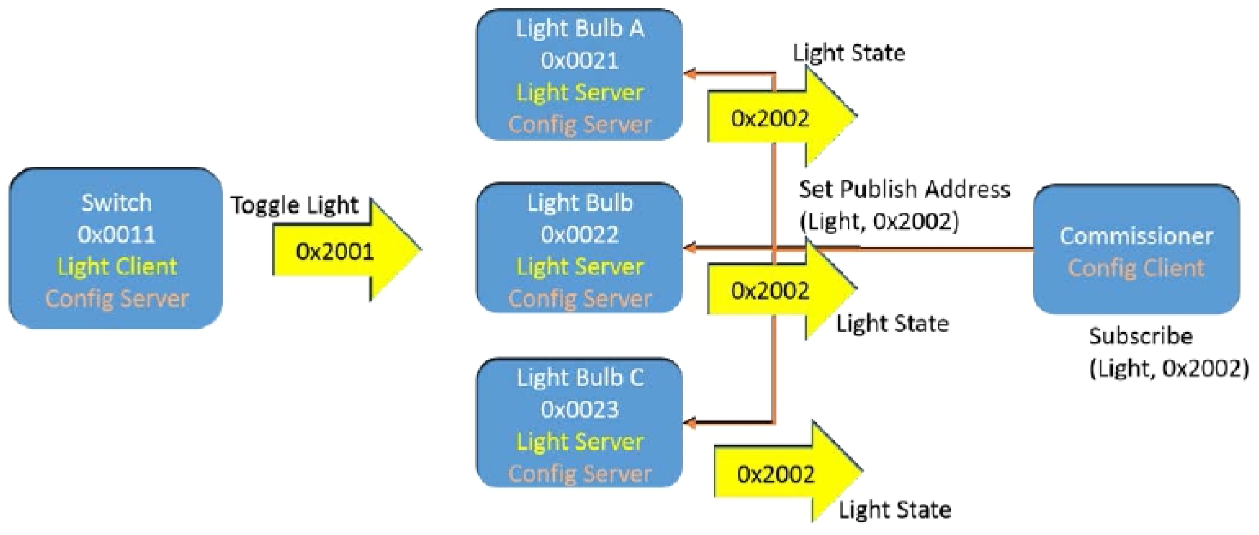


Figure 11. Light State message published by light bulbs

7 Revision History

This table summarizes revisions to this document.

Table 1. Revision history

Revision number	Date	Substantive changes
0	07/2016	Initial release
1	09/2016	Added new example and updated for KW41Z

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, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, C-5, CodeTest, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, and UMEMS are trademarks of NXP B.V. All other product or service names are the property of their respective owners. ARM, AMBA, ARM Powered, Artisan, Cortex, Jazelle, Keil, SecurCore, Thumb, TrustZone, and μVision are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. ARM7, ARM9, ARM11, big.LITTLE, CoreLink, CoreSight, DesignStart, Mali, mbed, NEON, POP, Sensinode, Socrates, ULINK and Versatile are trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2016 NXP B.V.

Document Number BLEMESHUG
Revision 1, 09/2016