RA4W1 Group

Bluetooth Mesh Development Guide

Introduction

Bluetooth Mesh Stack is the software library to build a mesh network that is compliant with Bluetooth Mesh Networking Specification and to perform many-to-many wireless communication. This document describes the overview of software architecture and its layers of the Bluetooth Mesh Stack and how to develop a Mesh Application. For more information on how to get started with Bluetooth Mesh Stack, refer to "*RA4W1 Group Bluetooth Mesh Startup Guide*" (R01AN5847).

Target Device

RA4W1 Group

Related Documents

The following documents are published on Renesas website.

Document Title	Document No.
RA4W1 Group User's Manual: Hardware	R01UH0883
Renesas Flexible Software Package User's Manual	-
RA4W1 Group Bluetooth Low Energy Application Developer's Guide	R01AN5653
RA4W1 Group Bluetooth Mesh Startup Guide	R01AN5847
RA4W1 Group Bluetooth Mesh sample application Application Note	R01AN5848
RA4W1 Group Bluetooth Mesh Development Guide	R01AN5849 This document

Contents

1. Bluetooth Mesh Overview 4
1.1 Node 4
1.2 Element 4
1.3 Address 5
1.4 State 5
1.5 Model 6
1.5.1 Client and server 6
1.5.2 Foundation models 6
1.5.3 Configuration model 7
1.5.4 Health model 7
1.5.5 Publication and subscription 7
1.6 Message 8
1.7 Mesh Bearer 9
1.8 Provisioning 10
1.9 Configuration 11
1.10 Optional Features 12
1.10.1 Relay 12
1.10.2 Proxy 13
1.10.3 Friendship 14

2. Mesh Application Overview 15
2.1 Software Structure 15
2.2 File Structure 17
2.3 Mesh Application 18
2.3.1 Mesh Core Module 19
2.3.2 Mesh Model Module 19
2.3.3 Mesh Model Composition 19
2.3.3.1 Configuration Model 21
2.3.3.2 Health Model 23
2.3.3.3 Generic OnOff Model 24
2.3.3.4 Vendor Model 24
2.4 Bluetooth Mesh Stack 25
2.5 Bluetooth Bearer 27
2.5.1 Bearer Functions for Message Transmission and Reception 27
2.5.2 Bearer Functions for Connection Control 28
2.5.3 Mesh GATT Services 29
2.5.4 ADV Bearer Operation 30
2.5.5 GATT Bearer Operation 31
2.6 MCU Peripheral Functions 32
2.7 Mesh Sample Program Configuration 35

2.8 Bluetooth Bearer Configuration 37

3. Application Development 38
3.1 Main Routine 39
3.2 Node Configuration 42
3.3 Provisioning 43
3.3.1 Provisioning Server 43
3.3.2 Provisioning Sequence 46
3.4 Proxy 50
3.4.1 Proxy Server 50
3.4.2 Proxy Client 52
3.4.3 Proxy Sequence 54
3.5 Friendship 56
3.5.1 Friend Node 56
3.5.2 Low Power Node 57
3.5.3 Low Power Node Sequence 60
3.5.4 Friend Node Sequence 62
3.6 Configuration 64
3.6.1 Configuration Server 64
3.6.2 Configuration Server Sequence 65
3.7 Health Model 66
3.7.1 Health Server 66
3.7.2 Health Server Sequence 69
3.8 Application Model 70
3.8.1 Server Model 70
3.8.2 Client Model 72
3.8.3 Generic OnOff Model Sequence 74
3.8.4 Vendor Model Sequence 74

4. Appendix 75
4.1 Command Line Interface Program 75
4.2 Program size 76

1. Bluetooth Mesh Overview

This chapter describes basic concepts of Bluetooth mesh defined by Bluetooth Mesh Networking Specifications. For more information, refer to Mesh Model and Mesh Profile specification on Specifications List. Figure 1-1 shows the typical topology of Bluetooth mesh network.

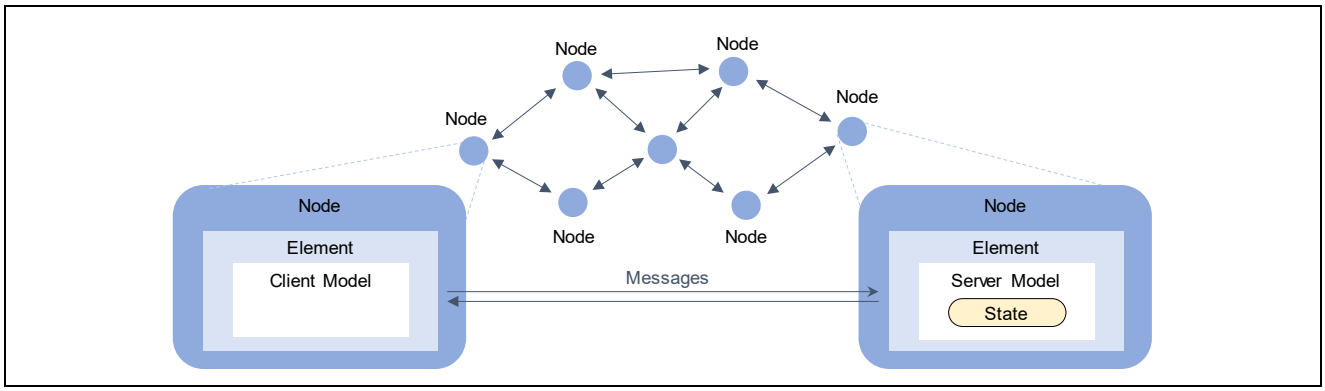


Figure 1-1 Basic Composition of Bluetooth Mesh Network

1.1 Node

A device joining a network is referred to as a Node. Network is a group of nodes sharing common address space and encryption keys. Communication among nodes is encrypted with Network Key. Each network is identified by Network ID generated from the Network Key. By default, one Network referred to as primary subnet is built. Multiple subnets can be also built to isolate communication scope.

1.2 Element

Element is a logical entity within a node. A node must have at least one element. And also the node can have multiple elements. First element is referred to as primary element. Each element is identified uniquely in a network by Unicast Address. The unicast address is assigned by provisioning.

1.3 Address

Address used in a mesh network has 16-bit length. Unassigned address, Unicast Address, Virtual Address, and Group address are defined as address types.

Table 1-1 Address Types

Address Type	Value	Value Range
Unassigned Address	0b0000000000000000	0x0000
Unicast Address	0b0xxxxxxxxxxxxxx (excluding 0b0000000000000000)	0x0001 to 0x7FFF
Virtual Address	0b10xxxxxxxxxxxxx	0x8000 to 0xBFFF
Group Address	0b11xxxxxxxxxxxxx	0xC000 to 0xFFFF

- **Unassigned address**

  Unassigned address is set to an element which has not been assigned unicast address yet. Unassigned address cannot be used as source address or destination address in a message.

- **Unicast address**

  Unicast address is an address to identify a single element. 32,767 of unicast address can be used in a mesh network. Unicast address can be used for source address and destination address in a message.

- **Virtual address**

  Virtual address is a multicast address generated by a Label UUID. Virtual address can be used for destination address in a message. Label UUID is a 128-bit value to categorize elements. This value can be generated randomly and shared by OOB (Out-Of-Band) among devices. Also, virtual address and Label UUID need not to be managed centrally.

- **Group address**

  Group address is a multicast address managed and assigned dynamically according to usage. Group address can be used for destination address in a message. Also, as shown Table 1-2, Fixed Group Addresses are defined for specific use case. (e.g. broadcasting to all-nodes.)

Table 1-2 Fixed Group Addresses

Fixed Group Address	Value
all-proxies	0xFFFC
all-friends	0xFFFD
all-relays	0xFFFE
all-nodes	0xFFFF

1.4 State

State is a value representing a condition of an element. States that are composed of two or more values are referred to as composite states. Moreover, State that change in conjunction with other states is referred to as bound states. The State can change instantaneously or can have transition time. Time from initial state to target state is referred to as transition time. Also, time from current state to target state is referred to as remaining time.

1.5 Model

Model is a standardized functionality so that nodes can perform typical operations in accordance with each scenario. Model defines states, messages that act upon a state, and associated behaviors.

1.5.1 Client and server

Model has server - client architecture. Server model have at least on state, but client model does not have state. Server model controls state of element by receiving messages the client model. Client model can get a state of server model with *GET* message and set a new state to server model with *SET* message or *SET Unacknowledged* message. Server model sends *STATUS* message as an acknowledge when state is changed, or an *GET* or *SET* message is received. Server model does not send *STATUS* message when *SET Unacknowledged* message is received.

Figure 1-2 shows node structure. A node can have multiple elements. An element can have multiple models, but not the same model in the element.

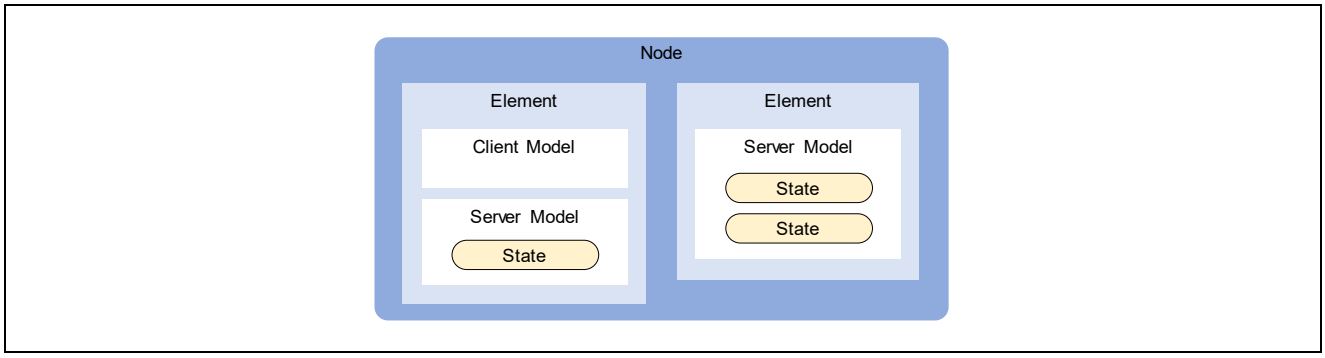


Figure 1-2 Node Composition

1.5.2 Foundation models

Foundation Models are models for configuring and managing behavior of elements. Primary element of each node must have configuration server model and health server model.

Table 1-3 Fixed Group Addresses

Model	SIG Model ID
Configuration Server	0x0000
Configuration Client	0x0001
Health Server	0x0002
Health Client	0x0003

1.5.3 Configuration model

Configuration model is a model for configuring behavior of node. Configuration values of a node and elements are defined as configuration states. Configuration server model has a configuration states. Configuration client model is a model for managing behavior of configuration server by configuration messages. Each configuration message is encrypted with a device key. Device keys are different from each node.

1.5.4 Health model

Health model is a model for monitoring physical condition of a node. Health server model is a model that has fault state for representing physical fault information. Health client model is a model for monitoring fault state of health server by health messages. Each health message is encrypted with an Application Key.

1.5.5 Publication and subscription

The model transmits messages is referred to as publication and receives messages is referred to as subscription. Model can publish messages to multiple elements by assigning multicast address as a destination address. The model can also selectively subscribe messages which have multicast address. Figure 1-3 shows how the model publishes and subscribes to messages. Each model sends messages in accordance with the publish address in model publication state. If the publish address is multicast address, each message is subscribed by multiple models in accordance with subscription addresses in subscription list state.

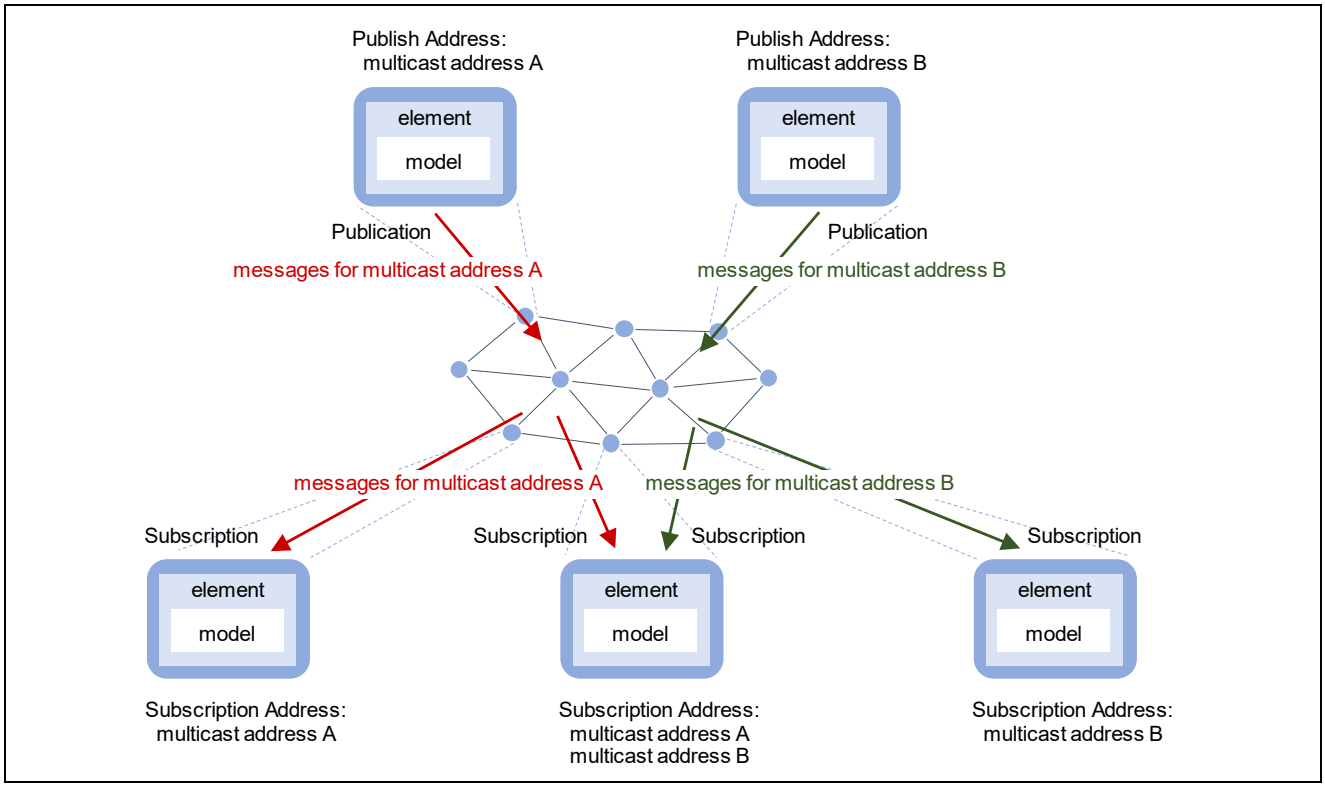


Figure 1-3 Message Publication and Subscription by models

1.6 Message

Data transmitted and received in a mesh network is referred to as message. Messages are categorized as follows whether the messages are segmented into multiple part or not.

- Unsegmented Message

  Unsegmented message is a message to transport unsegmented data. The message can transport Access PDU up to 11 bytes.

- Segmented Message

  Segmented Message is a message to transport each segmented data up to 32 segments. The message can transport Access PDU up to 380 bytes. When receiving all Segmented Messages, destination node reassembles data.

The access layer of Bluetooth mesh takes care of assemble and disassemble of access PDU. Figure 1-4 shows the assemble and disassemble of the PDU. Each node transmits and receives Network PDU as a Mesh Message.

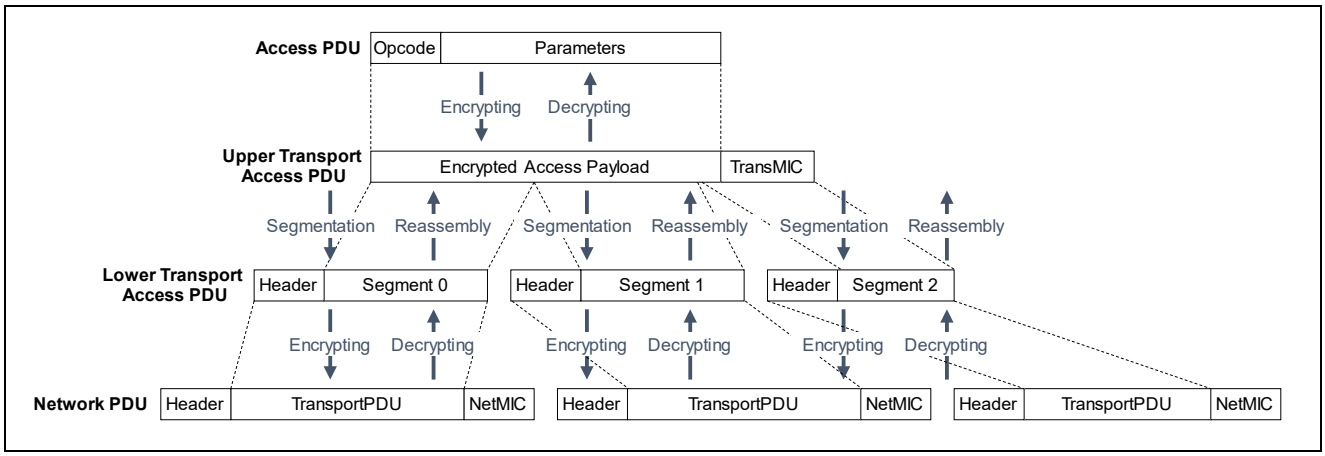


Figure 1-4 Segmentation and Reassembly of Access PDU

Header of network PDU includes fields such as source address (*SRC*), destination address (*DST*), and sequence number (SEQ). Network PDUs are encrypted with a Network Key, so only devices joining same mesh network can decrypt the PDUs. Also, *SRC* and *DST* of them are obfuscated, so other devices that does not have Network Key cannot trace the PDUs.

Header of Lower transport PDU includes *SEG* field to indicate whether unsegmented or segmented. And the lower transport PDU also includes *SeqZero*, *SegO*, and *SegN* field to use for reassemble segmented data.

Access PDU has two fields: application opcode and application parameters. The access PDU is encrypted with application key or device key, so data can be share among only nodes that share the keys. Applications keys are generated and are distributed by configuration client.

1.7 Mesh Bearer

Mesh Bearer is a method to transport messages in a mesh network. Two types of bearers that use Bluetooth Low Energy technology are defined as follows:

- **ADV bearer**

  ADV bearer sends messages by the Non-Connectable and Non-Scannable Undirected Advertising. Messages sent by ADV bearer can be received by many nodes simultaneously. Also, this bearer is referred to as PB-ADV when transmitting Provisioning PDUs on advertising channel during provisioning.

- **GATT bearer**

  GATT bearer sends messages over GATT service. A node of Client side sends messages by Write Without Response and a node of server side sends messages by Notification. Before communicating over the GATT service, establishing a connection is required. Messages sent by GATT bearer can be received by a connected peer node only. Also, this bearer is referred to as PB-GATT when transmitting Provisioning PDUs on data channel during provisioning.

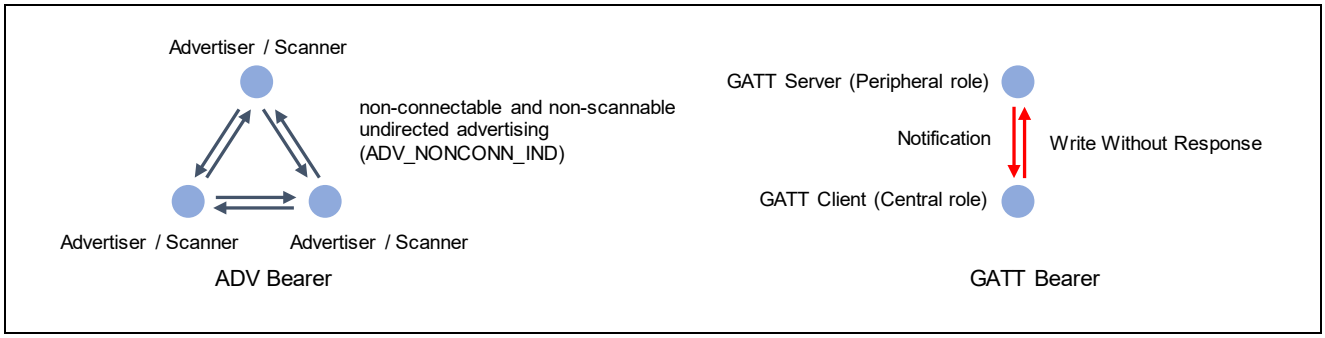


Figure 1-5 ADV Bearer and GATT Bearer

1.8 Provisioning

Provisioning is a process for joining a network. In provisioning, Provisioning Data that includes Network Key and Unicast Addresses of each element is distributed. Provisioning Data contains the following information.

- Network Key and Network Key Index
- Flags: Key Refresh Flag and IV Update Flag
- Current IV Index
- Unicast Address of the primary element

A device that is not joined mesh network yet is referred to as Unprovisioned Device. Each Unprovisioned Device is identified by 128-bit Device UUID.

A device that invites other devices to mesh network and distributes Provisioning Data is referred to as Provisioning Client or Provisioner. Generally, Provisioning Client is a smart phone or other mobile computing device.

A device that receives Provisioning Data and joins mesh network is referred to as Provisioning Server or Provisionee. The device that has joined mesh network is referred to as a Node.

1.9 Configuration

To communicate with other nodes by using Models, each node needs Configuration. By Configuration process, information required for Model operation such as Application Keys, Publish Address, Subscription Address is configured. Figure 1-6 shows a typical lifecycle of a node.

Newly introduced device is provisioned by Provisioner and joins mesh network. Furthermore, this device is configured by Configuration Client and becomes to be able to communicate with other nodes with Mesh Model. Generally, Configuration Client is a smart phone or other mobile computing device.

Configuration Client removes a node from a network by sending Config Node Reset message. Configuration Client will update encryption keys used in the mesh network, and the removed node becomes unable to communicate with other nodes.

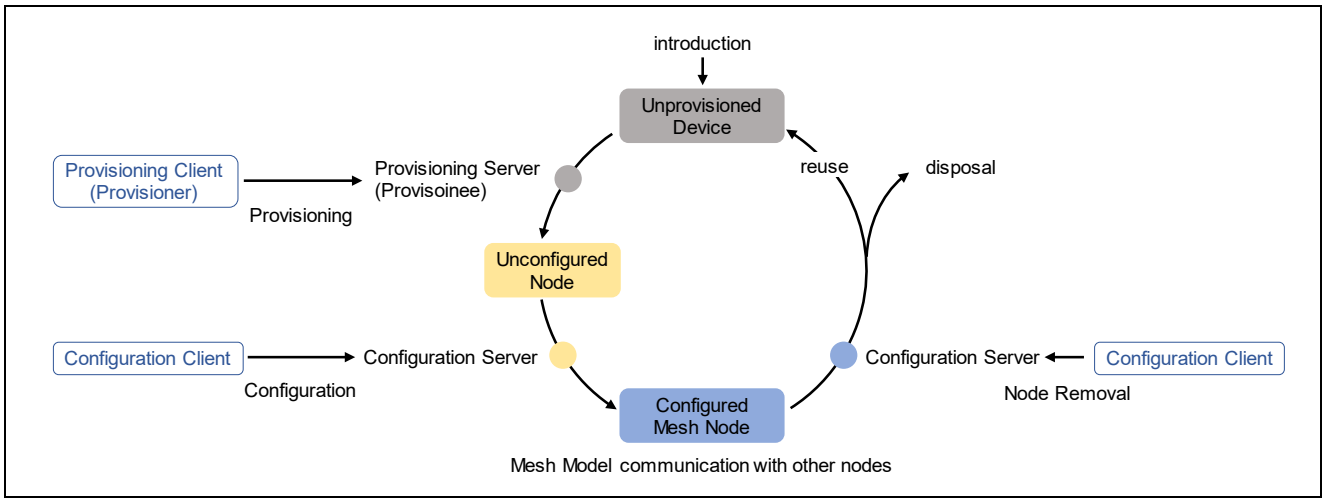


Figure 1-6 Lifecycle of a node

1.10 Optional Features

The following features are defined as optional features.

- Relay feature (refer to subsection 1.10.1)
- Proxy feature (refer to subsection1.10.2)
- Friend feature (refer to subsection 1.10.3)
- Low Power feature (refer to subsection 1.10.3)

It is possible to create various mesh network by enabling each optional features of nodes. Each optional features are described in the next section.

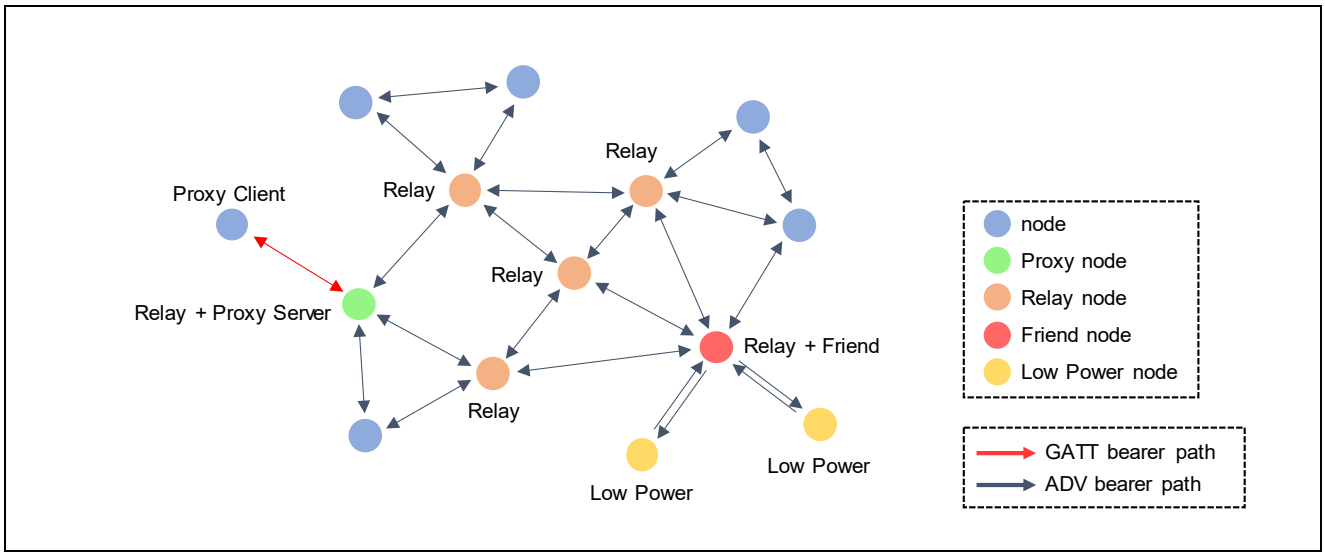


Figure 1-7 Mesh Network

1.10.1 Relay

The Relay feature is a feature that a node supporting ADV bearer relays received messages. Even if destination node is out direct radio range of a source node, messages are relayed by other nodes and spreads throughout a network, then the messages can reach the destination node. A node that relays message is referred to as a Relay node.

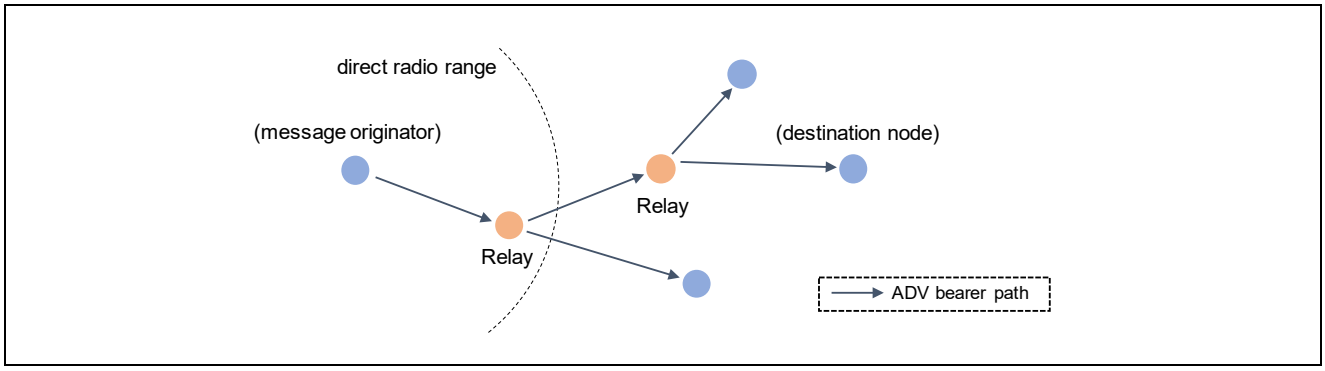


Figure 1-8 Relay

1.10.2 Proxy

Proxy feature is a feature that a node supporting both GATT bearer and ADV bearer forwards messages between both bearers.

A node supporting only GATT bearer communicates with a connected peer node only. When such a node sends and receives messages from / to mesh network, the node establishes a connection with a node that supports Proxy feature. Messages sent by this node can be forwarded by the proxy node to the ADV bearer and finally reach the destination node. And messages sent by other nodes are forwarded by GATT bearer of the Proxy node, and the messages can reach this node. A node that transmits messages between both GATT bearer and ADV bearer is referred to as a Proxy Server. A node that connects with Proxy Server and transmits / receives messages over GATT bearer is referred to as a Proxy Client.

Proxy Server has a list to manage Subscription Addresses of Proxy Client, and it is referred to as a Proxy Filter List. Either whitelist filter or blacklist filter can be set as a Proxy Filter Type. When Proxy Filter Type is whitelist filter, Proxy Server forwards only messages addressed to the address registered in the list. When Proxy Filter Type is blacklist filter, Proxy Server does not forward messages addressed to the address registered in the list.

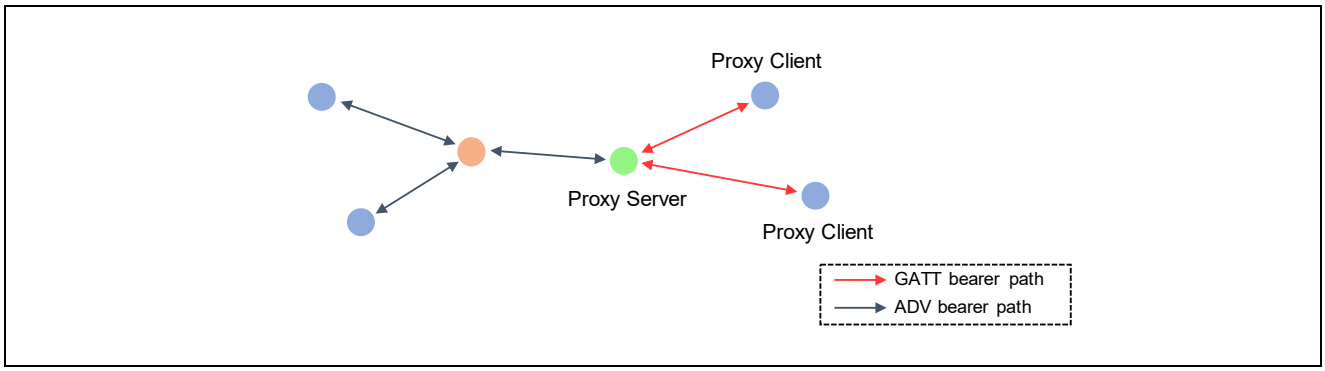


Figure 1-9 Proxy

1.10.3 Friendship

Friend feature is a feature that stores incoming messages needed by Low Power node and then forwards them when Low Power node requests. In general, a node supporting ADV bearer always perform Scan to receive Advertising packets including messages. Low Power feature is a feature to reduce Scan duty cycle. A node supporting Low Power feature can reduce power consumption by suspending Scan.

To perform Low Power feature, the node must establish a Friendship with one node supporting Friend feature. After establishing the friendship, Low Power node can suspend Scan, while Friend node must store received messages addressed to Low Power node.

Friend node has a list to manage Subscription Addresses of Low Power node, and it is referred to as a Friend Subscription List. After establishing a Friendship, Friend node stores messages addressed to Subscription Addresses registered in the list.

Low Power node polls Friend node intermittently if any messages are stored and resumes Scan only within a polling period. Friend node forwards the stored messages at this timing.

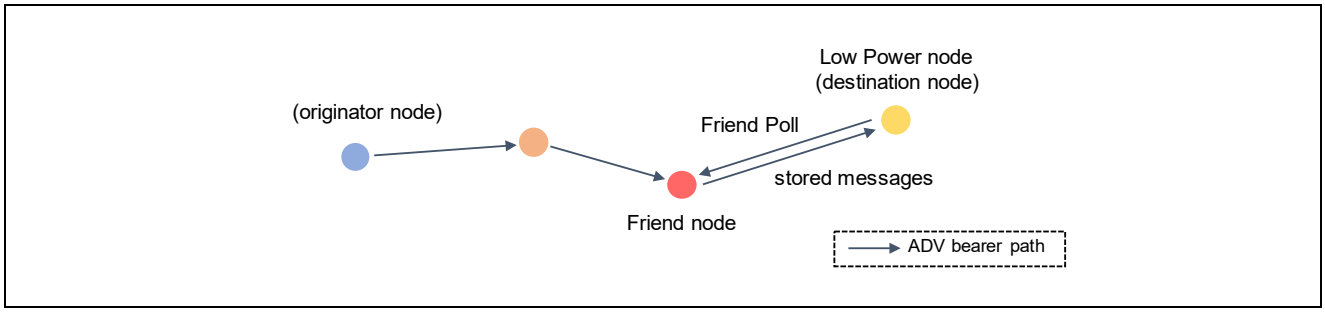


Figure 1-10 Friendship

2. Mesh Application Overview

This chapter describes the overview of Mesh Application.

2.1 Software Structure

Figure 2-1 shows the structure of software using Mesh Stack.

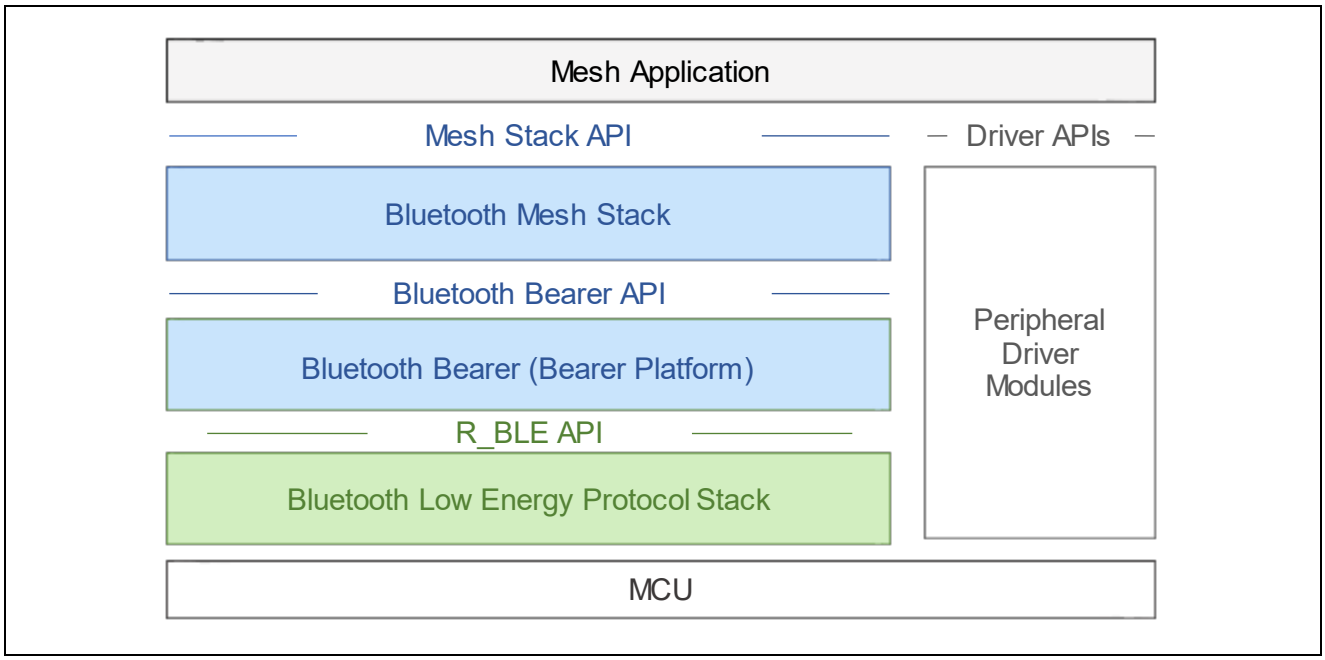


Figure 2-1 Software Architecture

The software using Mesh Stack is composed of the followings:

- **Mesh Application**

  The Mesh Application is an application program that performs Bluetooth mesh communication features. Users are required to understand specification of Mesh Stack API (*RM_BLE_MESH_XXXX, RM_MESH_XXXX*) and Bluetooth Bearer API (*RM_BLE_MESH_BEARER_XXXX*) to develop their own Mesh Applications. Also, sample program of Mesh Application is included in "*RA4W1 Group Bluetooth Mesh sample application*" (R01AN5848).

- **Bluetooth Mesh Stack**

  The Bluetooth Mesh Stack (hereinafter referred to as "Mesh Stack") is the software stack that provides applications with many-to-many wireless communication features which is compliant with the Bluetooth Mesh Networking specifications. This stack has Mesh Stack API to use mesh network communication features. Also, Mesh Stack is included in Mesh Module provided as Renesas Flexible Software Package (FSP).

- **Bluetooth Bearer (Bearer Platform)**

  The Bluetooth Bearer is the abstraction layer that provides the Bluetooth Mesh Stack and application with wrapper functions of Bluetooth Low Energy Protocol Stack. Also, Bluetooth Bearer is included in Bearer Platform Module provided as Renesas Flexible Software Package (FSP).

- **Bluetooth Low Energy Protocol Stack**

  The Bluetooth Low Energy Protocol Stack (hereinafter referred to as "Bluetooth LE Stack") is the software that provides upper layers with wireless communication features which is compliant with the Bluetooth Low Energy specifications. Bluetooth LE stack has *R_BLE* API to use Bluetooth Low Energy communication features. Also, Bluetooth LE Stack is included in BLE Module provided as Renesas Flexible Software Package (FSP).

- **Peripheral Driver Modules**

  Application, Mesh Stack, Bluetooth LE Stack use peripheral functions of microcontroller. Peripheral drivers that are provided as Renesas Flexible Software Package (FSP) can used for developing software for RA microcontrollers.

2.2 File Structure

"*RA4W1 Group Bluetooth Mesh sample application*" (R01AN5848) includes following sample programs of Mesh application.

- ekra4w1_mesh_client_baremetal: Project for EK-RA4W1 - Client Models
- ekra4w1_mesh_server_baremetal: Project for EK-RA4W1 - Server Models
- ekra4w1_mesh_cli_client_baremetal: Project for EK-RA4W1 - Command Line Interface Client Models
- ekra4w1_mesh_cli_server_baremetal: Project for EK-RA4W1 - Command Line Interface Server Models
- ekra4w1_mesh_client_freertos: Project for EK-RA4W1 - Client Models using FreeRTOS
- ekra4w1_mesh_server_freertos: Project for EK-RA4W1 - Server Models using FreeRTOS

Above programs also includes Mesh Stack, Bluetooth Bearer, Bluetooth LE Stack, and other Modules that are needed to build the sample program.

Structure of demo project is shown as below. This document describes software indicated in **bold**. For details of other Modules, refer to "*Renesas Flexible Software Package User's Manual*".

```
{project}\                                  :
    +---ra\fsp\lib\                         :  Mesh Stack Library
    |                                       :  Bluetooth LE Stack Library
    +---ra_cfg\                             :  Module Configuration
    |                                       :
    +---ra_gen\                             :  MCU Pin Configuration, Vector Table
    |                                       :
    +---src\                                :  Mesh Sample Program
        |   app_main.c                      :  Mesh Sample Header
        |   mesh_appl.h                     :  Mesh Core Module
        |   mesh_core.c                     :  Mesh Model Module
        |   mesh_model.c                    :
        |                                   :  Application Library
        +---app_lib\                        :
        |                                   :
        +---vendor_model\                   :  Vendor Model Header
        |       vendor_api.h                :  Vendor Client Module
        |       vendor_client.c                Vendor Server Module
        |       vendor_server.c
```

Regarding how to setup an environment for building sample program, refer to Chapter 2 in "*RA4W1 Group Bluetooth Mesh sample application*" (R01AN5848)

2.3 Mesh Application

Users is required to develop own Mesh Application for performing wireless communication capability with Bluetooth Mesh. "*RA4W1 Group Bluetooth Mesh sample application*" (R01AN5848) includes source code of sample program that can be used as a reference for developing Mesh Applications.

The sample program of Mesh Application (hereinafter referred to as "Mesh Sample Program") uses the API of Mesh Stack and performs Provisioning and basic operations as a mesh node. This section describes the detail of Mesh Sample Program. Supported features of Mesh Sample Program are shown as below:

- Unprovisioned Device operation: supports both PB-ADV bearer and PB-GATT bearer.
- Configuration Server operation: stores Configuration information in Data Flash memory.
- Generic OnOff Client operation: sends Generic OnOff Set message when on-board switch is pushed.
- Generic OnOff Server operation: controls on-board LED when Generic OnOff Set message is received.
- Vendor Client operation: sends Vendor Set message with character string input over UART.
- Vendor Server operation: outputs character string included in Vendor Set message received.
- Low Power operation: establishes a Friendship to Friend node and registers Subscription List with Friend Subscription List.
- Proxy Server operation: establish a connection to Proxy Client and forwards messages over GATT bearer.
- IV Update Initiation functionality: monitors sequence number of messages and initiates IV update procedure when the sequence number exceeds threshold value.

This sample program includes the following two modules. Those modules placed in *./src* folder of Mesh Sample Program.

- **Mesh Core Module**

  This module performs Provisioning as a Provisioning Server and enables GATT bearer as a Proxy Server after Provisioning. In addition, this module controls a Friendship as a Low Power Node. For more details, refer to Subsection 2.3.1.

- **Mesh Model Module**

  This module performs operations associated with Generic OnOff models and original Vendor models as well as Configuration Server model and Health Server model. For more details, refer to Subsection 2.3.2.

2.3.1 Mesh Core Module

Mesh Core Module included in Mesh Sample Program performs the following operations. This module is implemented in "*mesh_core.c*".

- Provisioning process (refer to Section 3.3)
- Proxy feature (refer to Section 3.4)
- Low Power feature (refer to Section 3.5)
- IV Update process

2.3.2 Mesh Model Module

Mesh Model module included in Mesh Sample Program performs the following operations. This module is implemented in "mesh_model.c".

- Mesh Model Composition (refer to Section 2.3.3)
- Configuration Model (refer to Section 2.3.3.1)
- Generic OnOff Model (refer to Section 2.3.3.3)
- Vendor Model (refer to Section 2.3.3.4)

2.3.3 Mesh Model Composition

This sample program uses the following model.

- Configuration Server model
- Health Server model
- Generic OnOff Server model (enabled when *ONOFF_SERVER_MODEL** macro is defined)
- Generic OnOff Client model (enabled when *ONOFF_CLIENT_MODEL** macro is defined)
- Vendor Server model (enabled when *VENDOR_SERVER_MODEL** macro is defined)
- Vendor Client model (enabled when *VENDOR_CLIENT_MODEL** macro is defined)

*These macros have defined as build option.

Figure 2-2 show the model structure of Mesh Sample Program. Generic OnOff Client model, Generic OnOff Server model, Vendor Client model, and Vendor Server model as well as Configuration Server model and Health Server model are located on the Primary element. The explanation of each model is given in the following sections.

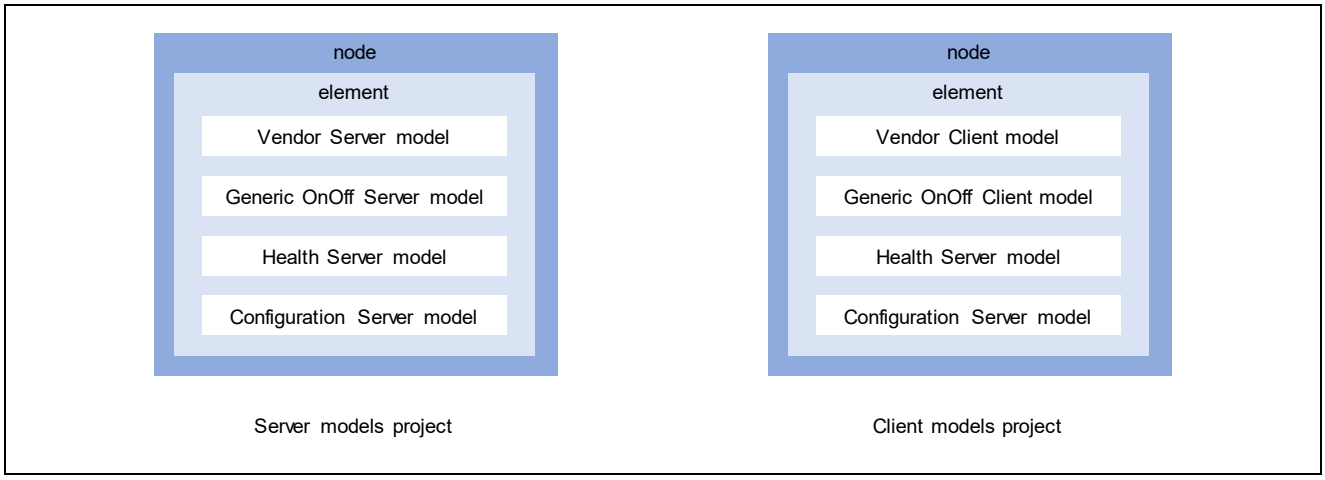


Figure 2-2 Model Composition of Mesh Sample Program

2.3.3.1 Configuration Model

Configuration model is the model to configure a node behavior. Configuration Server has multiple configuration states for storing configurations of node, element, and model behavior. These states are configured by messages from configuration client.

Table 2-1 States of Configuration Model

Model Name	SIG Model ID (16bits)	State
Configuration Server	0x0000	Secure Network Beacon Composition Data Default TTL GATT Proxy Friend Relay Model Publication Subscription List NetKey List AppKey List Model to AppKey List Node Identity Key Refresh Phase Heartbeat Publish Heartbeat Subscription Network Transmit Relay Retransmit PollTimeout List
Configuration Client	0x0001	-

Table 2-2 Configuration Messages

State	Message Name	Opcode	Direction
Secure Network Beacon	Config Beacon Get	0x8009	Client->Server
	Config Beacon Set	0x800A	Client->Server
	Config Beacon Status	0x800B	Server->Client
Composition Data	Config Composition Data Get	0x8008	Client->Server
	Config Composition Data Status	0x02	Server->Client
Default TTL	Config Default TTL Get	0x800C	Client->Server
	Config Default TTL Set	0x800D	Client->Server
	Config Default TTL Status	0x800E	Server->Client
GATT Proxy	Config GATT Proxy Get	0x8012	Client->Server
	Config GATT Proxy Set	0x8013	Client->Server
	Config GATT Proxy Status	0x8014	Server->Client
Friend	Config Friend Get	0x800F	Client->Server
	Config Friend Set	0x8010	Client->Server
	Config Friend Status	0x8011	Server->Client
Relay Relay Retransmit	Config Relay Get	0x8026	Client->Server
	Config Relay Set	0x8027	Client->Server
	Config Relay Status	0x8028	Server->Client
Model Publication	Config Model Publication Get	0x8018	Client->Server
	Config Model Publication Set	0x03	Client->Server
	Config Model Publication Virtual Address Set	0x801A	Client->Server
	Config Model Publication Status	0x8019	Server->Client
Subscription List	Config Model Subscription Add	0x801B	Client->Server
	Config Model Subscription Virtual Address Add	0x8020	Client->Server
	Config Model Subscription Delete	0x801C	Client->Server
	Config Model Subscription Virtual Address Delete	0x8021	Client->Server
	Config Model Subscription Virtual Address Overwrite	0x8022	Client->Server
	Config Model Subscription Overwrite	0x801E	Client->Server
	Config Model Subscription Delete All	0x801D	Client->Server
	Config Model Subscription Status	0x801F	Server->Client
	Config SIG Model Subscription Get	0x8029	Client->Server

State	Message Name	Opcode	Direction
	Config SIG Model Subscription List	0x802A	Server->Client
	Config Vendor Model Subscription Get	0x802B	Client->Server
	Config Vendor Model Subscription List	0x802C	Server->Client
NetKey List	Config NetKey Add	0x8040	Client->Server
	Config NetKey Update	0x8045	Client->Server
	Config NetKey Delete	0x8041	Client->Server
	Config NetKey Status	0x8044	Server->Client
	Config NetKey Get	0x8042	Client->Server
	Config NetKey List	0x8043	Server->Client
AppKey List	Config AppKey Add	0x00	Client->Server
	Config AppKey Update	0x01	Client->Server
	Config AppKey Delete	0x8000	Client->Server
	Config AppKey Status	0x8003	Server->Client
	Config AppKey Get	0x8001	Client->Server
	Config AppKey List	0x8002	Server->Client
Model to AppKey List	Config Model App Bind	0x803D	Client->Server
	Config Model App Unbind	0x803F	Client->Server
	Config Model App Status	0x803E	Server->Client
	Config SIG Model App Get	0x804B	Client->Server
	Config SIG Model App List	0x804C	Server->Client
	Config Vendor Model App Get	0x804D	Client->Server
	Config Vendor Model App List	0x804E	Server->Client
Node Identity	Config Node Identity Get	0x8046	Client->Server
	Config Node Identity Set	0x8047	Client->Server
	Config Node Identity Status	0x8048	Server->Client
-	Config Node Reset	0x8049	Client->Server
	Config Node Reset Status	0x804A	Server->Client
Key Refresh Phase	Config Key Refresh Phase Get	0x8015	Client->Server
	Config Key Refresh Phase Set	0x8016	Client->Server
	Config Key Refresh Phase Status	0x8017	Server->Client
Heartbeat Publication	Config Heartbeat Publication Get	0x8038	Client->Server
	Config Heartbeat Publication Set	0x8039	Client->Server
	Config Heartbeat Publication Status	0x06	Server->Client
Heartbeat Subscription	Config Heartbeat Subscription Get	0x803A	Client->Server
	Config Heartbeat Subscription Set	0x803B	Client->Server
	Config Heartbeat Subscription Status	0x803C	Server->Client
Network Transmit	Config Network Transmit Get	0x8023	Client->Server
	Config Network Transmit Set	0x8024	Client->Server
	Config Network Transmit Status	0x8025	Server->Client
PollTimeout List	Config Low Power Node PollTimeout Get	0x802D	Client->Server
	Config Low Power Node PollTimeout Status	0x802E	Server->Client

Memory region for storing configuration states are allocated in Mesh Stack. When receiving configuration message, Mesh Stack updates values of the configuration state automatically. Therefore, application does not need to take care of the configuration states. Also, application can access values of the Configuration states by using Mesh Stack API.

2.3.3.2 Health Model

Health model is the model to monitor the physical condition of a node. Health Server has Fault states for storing physical fault condition of node. These states are updated when fault occurs. In addition, self-testing of a node can be performed by messages from Health Client. Also, Health Server has Attention Timer state to activate a physical behavior (e.g., LED blinking, vibrating) for calling attention. This attention state may be used to indicate which device is performing provisioning procedure, etc.

Table 2-3 States of Health Model

Model Name	SIG Model ID (16bits)	State
Health Server	0x0002	Current Fault Registered Fault Health Period Attention Timer
Health Client	0x0003	-

Table 2-4 Health Messages

State	Message Name	Opcode	Direction
Current Fault	Health Current Status	0x04	Server->Client
Registered Fault	Health Fault Get	0x8031	Client->Server
	Health Fault Clear	0x802F	Client->Server
	Health Fault Clear Unacknowledged	0x8030	Client->Server
	Health Fault Status	0x05	Server->Client
	Health Fault Test	0x8032	Client->Server
	Health Fault Test Unacknowledged	0x8033	Client->Server
Health Period	Health Period Get	0x8034	Client->Server
	Health Period Set	0x8035	Client->Server
	Health Period Set Unacknowledged	0x8036	Client->Server
	Health Period Status	0x8037	Server->Client
Attention Timer	Health Attention Get	0x8004	Client->Server
	Health Attention Set	0x8005	Client->Server
	Health Attention Set Unacknowledged	0x8006	Client->Server
	Health Attention Status	0x8007	Server->Client

Memory region for storing Health states is allocated in Mesh Stack.

2.3.3.3 Generic OnOff Model

Generic OnOff Model is a model that is defined by Bluetooth SIG. Generic OnOff Server has a Generic OnOff state that stores value of either On or Off. This state is configured by messages from Generic OnOff Client.

Table 2-5 State of Generic OnOff Model

Model Name	SIG Model ID (16bits)	State
Generic OnOff Server	0x1000	Generic OnOff (0x00: Off, 0x01: On)
Generic OnOff Client	0x1001	-

Application must allocate memory for storing Generic OnOff state. Mesh Stack notifies received Generic OnOff message by callback function. Application must handle Generic OnOff state in accordance with Generic OnOff message notified by the callback function.

Table 2-6 Generic OnOff Messages

State	Message Name	Opcode	Direction
Generic OnOff	Generic OnOff Get	0x8201	Client->Server
	Generic OnOff Set	0x8202	Client->Server
	Generic OnOff Set Unacknowledged	0x8203	Client->Server
	Generic OnOff Status	0x8204	Server->Client

2.3.3.4 Vendor Model

User can define their own model as Vendor Model. This section describes Vendor Model implemented in Mesh Sample Program. Vendor Server implemented in Mesh sample program has a vendor state for storing any variable-length data. This state is configured by vendor client.

Table 2-7 State of Vendor Model

Model Name	Vendor Model ID (32bits)	State
Vendor Server	0x00010036 (default value)	Vendor state (any variable-length data)
Vendor Client	0x00020036 (default value)	-

Table 2-8 Vendor Messages

State	Message Name	Opcode	Direction
Vendor	Vendor Get	0xC10036 (default value)	Client->Server
	Vendor Set	0xC20036 (default value)	Client->Server
	Vendor Set Unacknowledged	0xC30036 (default value)	Client->Server
	Vendor OnOff Status	0xC40036 (default value)	Server->Client

2.4 Bluetooth Mesh Stack

Bluetooth Mesh Stack provides applications with many-to-many wireless communication features which is compliant with the Bluetooth Mesh Networking specifications. Flexible Software Package provides the Mesh stack as static library. And you can use the Mesh features via Bluetooth Mesh Stack API. Figure 2-3 shows the internal architecture of Bluetooth Mesh Stack.

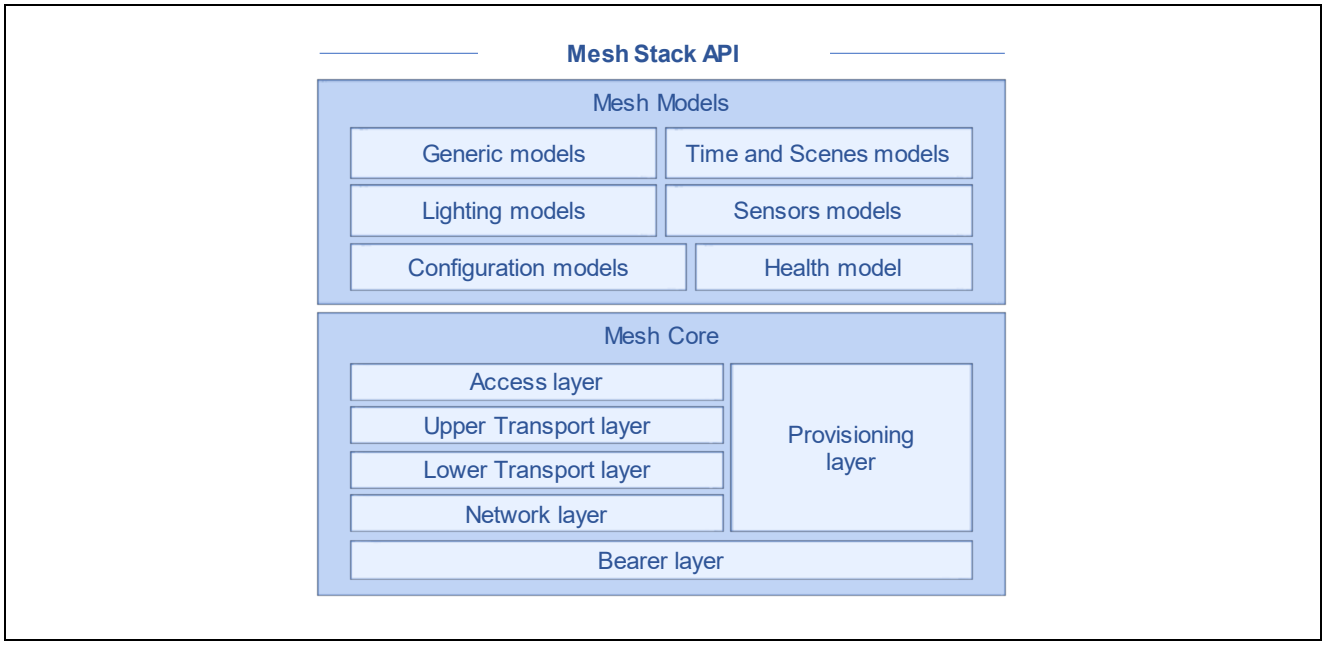


Figure 2-3 Internal Architecture of Bluetooth Mesh Stack

The Bluetooth Mesh Stack is composed of the following two blocks:

- **Mesh Core**

  Mesh Core block is composed of modules corresponding with each layer defined by Mesh Profile Specification and provides application with the features to perform Provisioning process and mesh networking operations. Regarding the Mesh Profile Specification, visit the Specifications List and refer to Mesh Profile Specification document.

- **Mesh Models**

  Mesh Models block is composed of modules corresponding with each model defined by Mesh Model Specification and provides application with the features to support Mesh models that defines basic operations on a mesh network. Regarding the Mesh Model Specification, visit the Specifications List and refer to Mesh Model Specification document.

Mesh Stack consists of modules to implement protocol defined by Bluetooth Mesh Networking Specifications. Mesh Stack API has the following function prefixes corresponding to each module. To make Mesh Application, it is necessary to use Mesh Stack APIs according to your use case. Regarding the specification of Mesh Stack API, refer to "*Renesas Flexible Software Package User's Manual*".

Table 2-9 Mesh stack APIs

Module		**Function Prefix**
Mesh Model		
	Generic OnOff	RM_MESH_GENERIC_ON_OFF_*()
	Generic Level	RM_MESH_GENERIC_LEVEL_*()
	Generic Default Transition Time	RM_MESH_GENERIC_DTT_*(),
	Generic Power OnOff	RM_MESH_GENERIC_POO_*()
	Generic Power Level	RM_MESH_GENERIC_PL_*()
	Generic Battery	RM_MESH_GENERIC_BATTERY_*()
	Generic Location	RM_MESH_GENERIC_LOC_*()
	Generic Property	RM_MESH_GENERIC_PROP_*()
	Sensor	RM_MESH_SENSOR_*()
	Time	RM_MESH_TIME_*()
	Scene	RM_MESH_SCENE_*()
	Scheduler	RM_MESH_SCHEDULER_*()
	Light Lightness	RM_MESH_LIGHT_LIGHTNESS_*()
	Light CTL	RM_MESH_LIGHT_CTL_*()
	Light HSL	RM_MESH_LIGHT_HSL_*()
	Light xyL	RM_MESH_LIGHT_XYL_*()
	Light LC	RM_MESH_LIGHT_LC_*()
	Configuration	RM_MESH_CONFIG_*()
	Health	RM_MESH_HEALTH_*()
Mesh Core		
	Access Layer	RM_BLE_MESH_ACCESS_*()
	Transport Layer	RM_BLE_MESH_UPPER_TRANS_*()
	Lower Transport Layer	RM_BLE_MESH_LOWER_TRANS_*()
	Network Layer	RM_BLE_MESH_NETWORK_*()
	Bearer Layer	RM_BLE_MESH_BEARER_*()
	Provisioning Layer	RM_BLE_MESH_PROVISION_*()

2.5 Bluetooth Bearer

Bluetooth Bearer provides Mesh Stack and applications with wrapper functions of Bluetooth LE Stack. Flexible Software Package provides Bluetooth bearer as static library. Bluetooth LE Stack provides upper layers with wireless communication features which is compliant with the Bluetooth Low Energy specifications. Flexible Software Package also provides Bluetooth LE stack as static library. Figure 2-4 shows the internal architecture of Bluetooth Bearer. Bearer functions for message transmission and reception are used by Mesh Stack. Bearer functions for connection control must be used by Mesh Application as necessary.

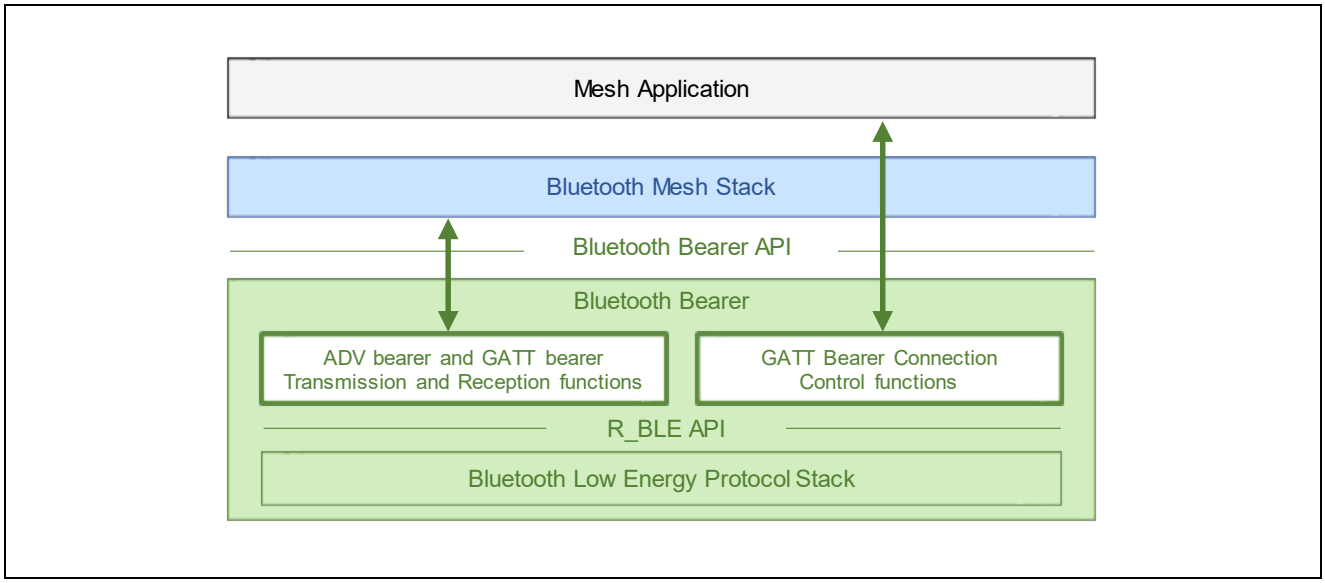


Figure 2-4 Bluetooth Bearer Operations

Regarding the specification of Bluetooth bearer API and R_BLE APIs, refer to "*Renesas Flexible Software Package User's Manual*".

2.5.1 Bearer Functions for Message Transmission and Reception

The Mesh stack provides message sending and receiving functions for each model with the prefixes shown in Table 2-9. You need to call *RM_MESH_BEARER_PLATFORM_Setup ()* API before using the message send / receive function. The Mesh stack uses these messages send and receive functions to send and receive provisioning PDUs and Mesh messages. For details on the message sending / receiving function, refer to the "*Renesas Flexible Software Package User 's Manual*".

2.5.2 Bearer Functions for Connection Control

Mesh Stack manages neither connection status nor GATT service. Therefore, to use GATT bearer, Mesh Application must control a connection and GATT services by using the bearer functions for connection control directly. Table 2-10 shows the bearer functions for connection control. Those functions provide the functionalities for service discovery and notification permission as well as connection establishment and disconnection.

Table 2-10 Bearer Functions for Connection Control

Function	Routine	GATT Server (Peripheral)	GATT Client (Central)
RM_MESH_BEARER_PLATFORM_CallbackSet()	Register GATT Interface Callback	used	used
RM_MESH_BEARER_PLATFORM_SetGattMode()	Set GATT Bearer Mode NOTE1	used	used
RM_MESH_BEARER_PLATFORM_GetGattMode()	Get GATT Bearer Mode NOTE1	used	used
RM_MESH_BEARER_PLATFORM_Disconnect()	Disconnect	used	used
RM_MESH_BEARER_PLATFORM_SetScanResponseData()	Set Scan Response Data	used	not used
RM_MESH_BEARER_PLATFORM_ScanGattBearer()	Scan Connectable Device	not used	used
RM_MESH_BEARER_PLATFORM_Connect()	Create Connection	not used	used
RM_MESH_BEARER_PLATFORM_DiscoverService()	Perform Service Discovery	not used	used
RM_MESH_BEARER_PLATFORM_ConfigureNotification()	Configure Mesh GATT Services Notification Permission NOTE2	not used	used

NOTE1: GATT Bearer Mode is either Provisioning Mode or Proxy Mode.

NOTE2: GATT Server configures MTU size to Mesh Stack when Notification is enabled. When changing MTU size, GATT Client has to perform MTU Exchange procedure before enabling Notification.

Regarding how to change MTU size, refer to Section 7.4 in "RA4W1 Group Bluetooth Low Energy Application Developer's Guide"(R01AN5653).

2.5.3 Mesh GATT Services

Mesh GATT Services are used for mesh message transmission and reception over GATT bearer. Composition of the Mesh GATT Services are listed in Table 2-11. Mesh Provisioning Service is used for Provisioning over PB-GATT bearer, and Mesh Proxy Service is used for Proxy connection after Provisioning. Which one of Mesh GATT Services are exposed is switched by *RM_MESH_BEARER_PLATFORM_SetGattMode*() API. GATT Database that defines Mesh GATT Service is held in Bluetooth Bearer.

Table 2-11 Composition of the Mesh GATT Services

Service (UUID)	Characteristic (UUID)	Property	Value
Mesh Provisioning Service (0x1827)	Mesh Provisioning Data In Characteristic (0x2ADB)	Write Without Response	Provisioning PDU from a Provisioning Client to a Provisioning Server
	Mesh Provisioning Data Out Characteristic (0x2ADC)	Notify	Provisioning PDU from a Provisioning Server to a Provisioning Client.
Mesh Proxy Service (0x1828)	Mesh Proxy Data In Characteristic (0x2ADD)	Write Without Response	Proxy PDU message containing Network PDU, mesh beacons, or proxy configuration from a Proxy Client to a Proxy Server
	Mesh Proxy Data Out Characteristic (0x2ADE)	Notify	Proxy PDU message containing Network PDU, mesh beacon, or proxy configuration from a Proxy Server to a Proxy Client.

2.5.4 ADV Bearer Operation

When Mesh Application calls *RM_MESH_BEARER_PLATFORM_Setup*(), Bluetooth Bearer registers message transmission and reception functions for ADV Bearer with Mesh Stack and starts Scan operation.

Advertising packets received by Bluetooth LE Stack are notified to Mesh Stack. Also, Bluetooth LE Stack transmits Advertising packets when Mesh Stack calls the message transmission function.

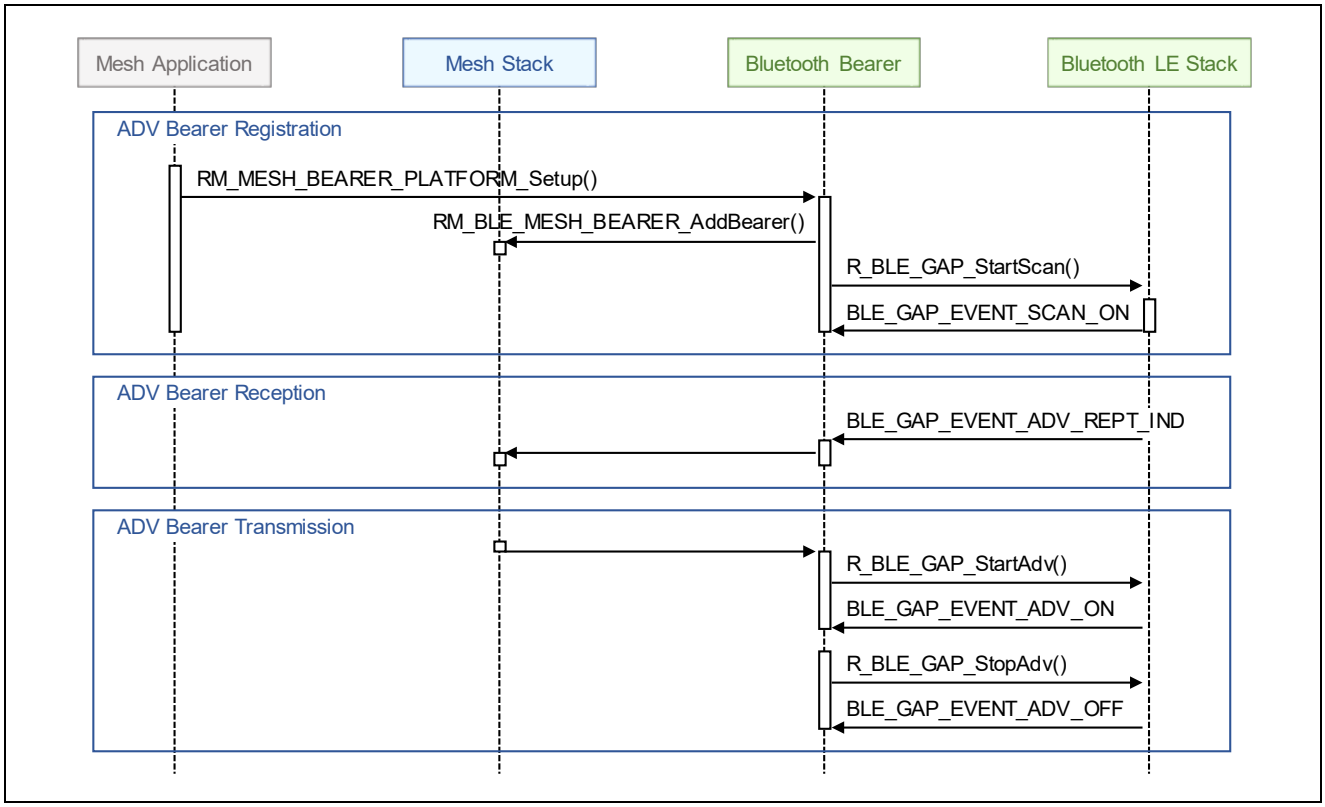


Figure 2-5 ADV Bearer Operation

2.5.5 GATT Bearer Operation

When a connection is established and enabling Notification completes, Bluetooth Bearer registers message transmission and reception functions for GATT Bearer with Mesh Stack.

In the case of node behaves as GATT Server, Bluetooth LE Stack transmits message by Notification when Mesh Stack calls the message transmission function. Also, message transmitted by Write Without Response is notified to Mesh Stack.

In the case of node behaves as GATT Client, Bluetooth LE Stack transmits message by Write Without Response when Mesh Stack calls the message transmission function. Also, message transmitted by Notification is notified to Mesh Stack.

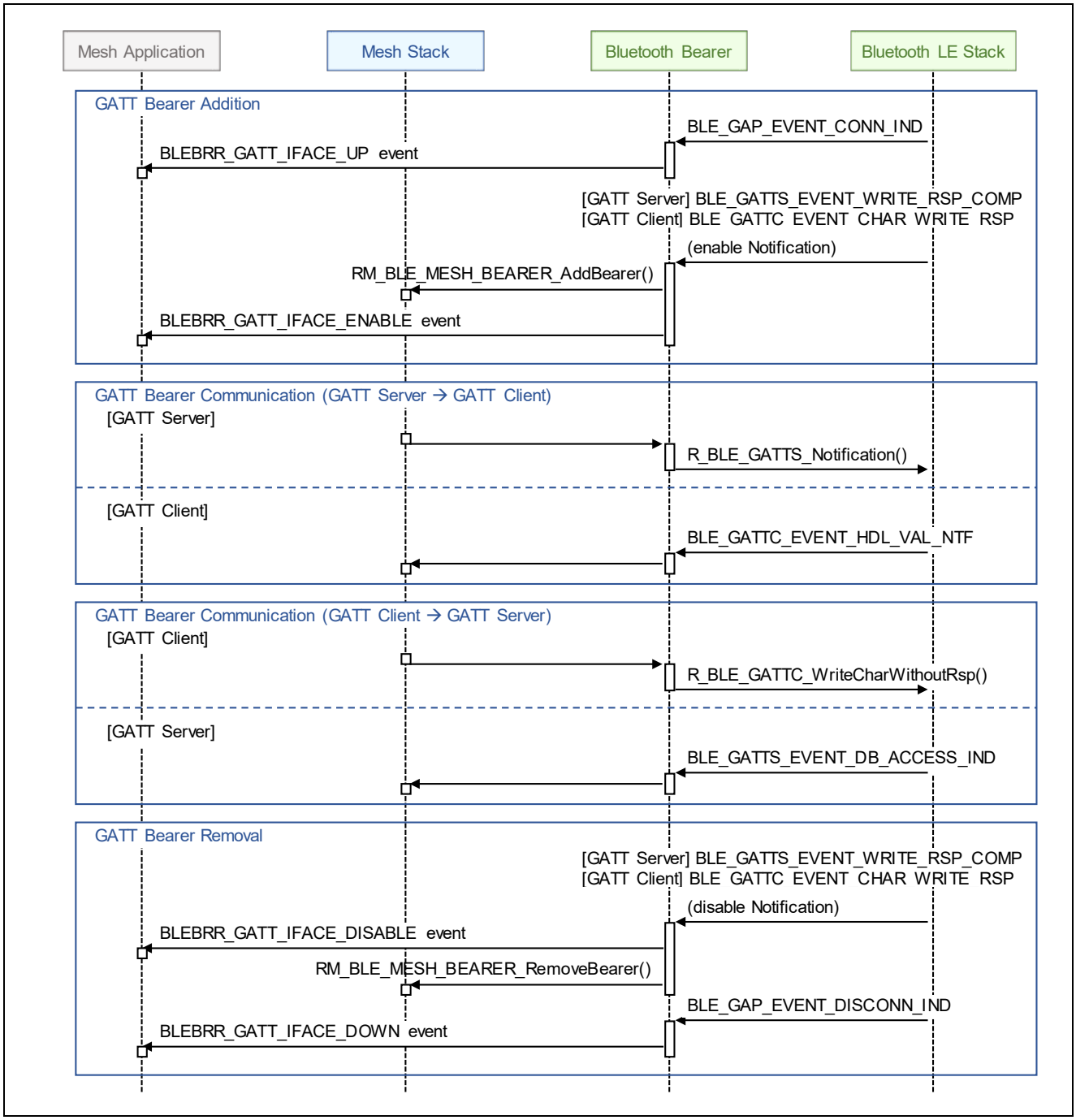


Figure 2-6 GATT Bearer Operation

2.6 MCU Peripheral Functions

Mesh Sample Program uses some RA4W1 peripheral functions listed in Table 2-12.

Table 2-12 RA4W1 Peripheral Functions used

RA4W1 Peripherals	FSP provided Peripheral Driver	Software using Peripherals
I/O Ports – P106, P402, and P404: when EK-RA4W1 is used Interrupt Controller Unit (ICU) – IRQ4	r_ioport Module r_icu Module	Mesh Sample Program
Interrupt Controller Unit (ICU) – BLEIRQ	r_icu Module	Bluetooth LE Stack
Serial Communication Interface (SCI) – SCI4	r_sci_uart Module	Mesh Sample Program
General PWM Timer (GPT) – GPT0: Bluetooth LE Stack use exclusively – GPT1: Mesh Stack and Bluetooth Bearer share – GPT2: Mesh Sample Program use	r_gpt Module	Mesh Sample Program Mesh Stack Bluetooth Bearer Bluetooth LE Stack
Low Power Modes (LPM)	r_lpm Module	Mesh Sample Program
Data Flash memory (FLASH) – Block 0 to 5	r_flash_lp Module	Mesh Stack

- **I/O Ports and Interrupt Controller Unit (ICU)**

  Mesh Sample Program uses *r_ioport* Module and *r_icu* Module to use I/O Ports for the following processing.

  - LED Control on development board
  - Switch Pushing Detection on development board

- **Interrupt Controller Unit (ICU)**

  Bluetooth LE Stack uses *r_icu* Module to detect RF interruption.

- **Serial Communication Interface (SCI)**

  Mesh Sample Program uses *r_sci_uart* Module to output and input console over UART.

- General PWM Timer (GPT)

  Bluetooth LE Stack of BLE Module uses GPT0 exclusively.

  Mesh Stack monitors 96 hours that is minimum duration of IV Update Procedure by using GPT1.

  Bluetooth Bearer uses GPT1 for the following processing.

  - Advertising Transmission Control for ADV Bearer

  Mesh Sample Program uses GPT2 for the following processing.

  - LED Blinking on development board
  - Avoiding Chattering of Switch on development board
  - MCU Reset Delay after receiving Config Node Reset
  - Completion of IV Update Procedure

- **Low Power Modes (LPM)**

  Mesh Sample Program uses *r_lpm* Module to enable Low Power Consumption function of MCU.

- Data Flash memory (FLASH)

  Data Flash driver to use Data Flash memory is registered by *RM_BLE_MESH_Open()* API to Mesh Stack. This driver accesses Data Flash memory by using *r_flash_lp* Module. The Mesh stack uses Data flash from Block 0 to *Storage/Block number* property which specified by *rm_ble_mesh* module *Storage/Block Number* property. Therefore, *rm_ble_mesh* module *Common/Data Flash Block for Security Data* property and *Common/Device Specific Data Flash Block* property should be configured avoiding the region used by *Storage/Block Number* property.

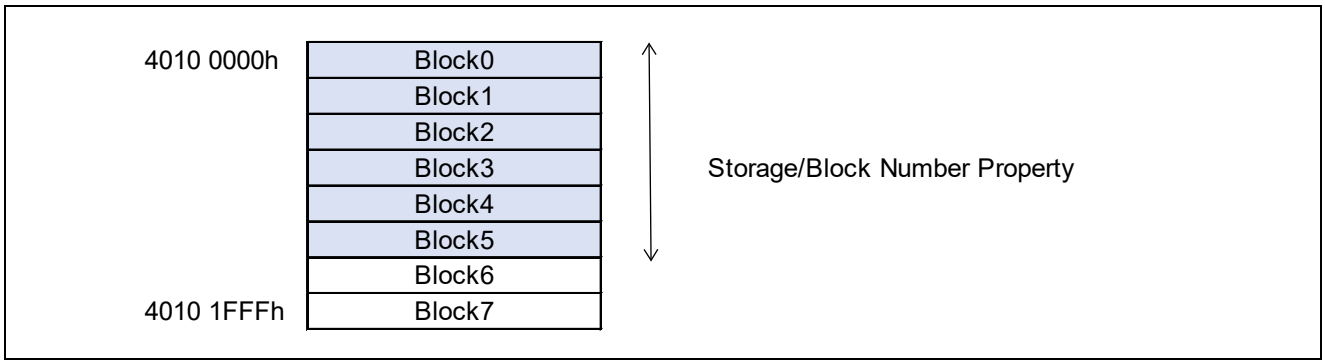


Figure 2-7 Data Flash memory region used

Mesh Stack stores the following information to Data Flash memory.

- Information exchanged during Provisioning
  - mesh addresses
  - encryption keys
- Information exchanged during Configuration
  - model composition
  - model configuration
- IV index and associated state
- Sequence Number

These information will be changed very rarely except for Sequence Number. The sequence number is incremented for each new network message transmission. If it is written for each increment, the flash memory reaches the write cycle limit in a short span of time. Specify the interval for writing the sequence number to the data flash by *Network Sequence Number Block Size* property of the *rm_ble_mesh* module to reduce the frequency of writing data flash.

2.7 Mesh Sample Program Configuration

Mesh Sample Program has multiple compilation switches to configure its operation. Compilation switches are implemented in "*mesh_appl.h*".

```
#define IV_UPDATE_INITIATION_EN         (1)
#define LOW_POWER_FEATURE_EN            (0)
#define CONSOLE_OUT_EN                  (1)
#define ANSI_CSI_EN                     (1)
#define SCI_RCV_STRING_EN               (1)
#define SCI_RCV_STRING_BUFFER_LEN       (0x100)
```

- **Enabling IV Update Initiation Processing**

  IV Update Initiation processing is enabled by setting the *IV_UPDATE_INITIATION_EN* macro to (1). This processing monitors sequence number of incoming and outgoing message and initiates IV Update procedure when the sequence number is greater than or equal to threshold. It prevents sequence number of own node or other nodes from exhausting.

Configuration Macro	Configuration Value	Description
IV_UPDATE_INITIATION_EN	0	Disable IV Update Initiation processing
	1	Enable IV Update Initiation processing

- **Enabling Low Power Feature**

  Low Power feature is enabled by setting the *LOW_POWER_FEATURE_EN* macro to (1). After Provisioning, Mesh Sample Program establishes a Friendship with Friend node and works as a Low Power node.

Configuration Macro	Configuration Value	Description
LOW_POWER_FEATURE_EN	0	Disable Transition to Low Power Node
	1	Enable Transition to Low Power Node

- **Console Output Configuration**

  Console Output is enabled by setting the *CONSOLE_OUT_EN* macro to (1). It is possible to trace API called by Mesh Sample Program and events returned by Mesh Stack on terminal emulator.

Configuration Macro	Configuration Value	Description
CONSOLE_OUT_EN	0	Disable Console Log Output
	1	Enable Console Log Output

- **ANSI CSI Console Output Configuration**

  Output ANSI CSI (*Control Sequence Introducer*) to console is enabled by setting the *ANSI_CSI_EN* macro to (1). Mesh Sample Program colors some log. In the case that serial terminal emulator you use does not support ANSI CSI, set the *ANSI_CSI_EN* macro to (0).

Configuration Macro	Configuration Value	Description
ANSI_CSI_EN	0	Disable ANSI CSI Output to Console Log
	1	Enable ANSI CSI Output to Console Log

- **Console String Reception Configuration**

  String reception from console is enabled by setting the SCI_RCV_STRING_EN macro to (1). String received is notified by a callback function.

Configuration Macro	Configuration Value	Description
SCI_RCV_STRING_EN	0	Disable String Reception from Console
	1	Enable String Reception from Console
SCI_RCV_STRING_BUFFER_LEN	0x0001 to 0xFFFF	String Reception Buffer Size

Mesh Sample Program has setting macros to configure Provisioning operation. Setting macros are implemented in "mesh_core.c".

```
#define CORE_PROV_BEACON_OOB_INFO           (0)
#define CORE_PROV_BEACON_URI_INFO           {.payload = "\x17//www.example.com", .length = 18}
```

- **OOB Information**

  OOB Information should be set to CORE_PROV_STATIC_OOBINFO macro. OOB Information is delivered by Unprovisioned Device Beacon. Multiple OOB Information can be set. When URI and Barcode are set, (PROV_OOB_TYPE_URI | PROV_OOB_TYPE_BARCODE) should be set.

Configuration Macro	Configuration Value	Description
CORE_PROV_BEACON_OOB_INFO	PROV_OOB_TYPE_OTHER	Other
	PROV_OOB_TYPE_URI	URI
	PROV_OOB_TYPE_2DMRC	2D machine-readable code
	PROV_OOB_TYPE_BARCODE	Bar code
	PROV_OOB_TYPE_NFC	Near Field Communication (NFC)
	PROV_OOB_TYPE_NUMBER	Number
	PROV_OOB_TYPE_STRING	String
	PROV_OOB_TYPE_ONBOX	On box
	PROV_OOB_TYPE_INSIDEBOX	Inside box
	PROV_OOB_TYPE_ONPIECEOFPAPER	On piece of paper
	PROV_OOB_TYPE_INSIDEMANUAL	Inside manual
	PROV_OOB_TYPE_ONDEVICE	On device

- **Encoded URI Information**

  When PROV_OOB_TYPE_URI is set to CORE_PROV_BEACON_OOB_INFO macro described above, Encoded URI Information must be set to CORE_PROV_BEACON_URI_INFO macro. Encoded URI Information will be delivered with <<URI>> of AD Type and Hash value of Encoded URI Information will be delivered with Unprovisioned Device Beacon.

Configuration Macro	Configuration Value	Description
CORE_PROV_BEACON_URI_INFO	max.29 octets	Encoded URI URI Scheme must be encoded with "URI Scheme Name String Mapping" defined in Assigned Numbers of Bluetooth SIG

2.8 Bluetooth Bearer Configuration

The following are advertising, scan, and connection parameters performed by Bluetooth bearer. These parameters cannot be changed by the user application except for the device address type.

- **Device Address Type Configuration**

  Device Address Type property used by Bluetooth Bearer can be configured by *rm_mesh_bearer_platform* module.

Property	Configuration Value	Description
BLEBRR_VS_ADDR_TYPE	0	Public Device Address
	1	Random Device Address

- **GATT Bearer Connectable Advertising Configuration**

Parameter	Value
Advertising Type	Connectable and Scannable Undirected Legacy Advertising
Advertising Interval	100msec
Channel Map	Ch37, 38, 39
Filter Policy	Process Scan Requests and Connection Requests from All Devices
Advertising Data length	31 bytes

- **ADV Bearer Scan Configuration**

Parameter	Value
Scan type	Passive
Scan interval	5msec
Filter Policy	None
Duplicate filter	None

- **GATT Bearer GATT Client Connection Configuration**

Parameter	Value
Connection Interval	80msec
Peripheral latency	0
Supervision timeout	9.5sec

3. Application Development

This chapter describes how to develop an application using Bluetooth Mesh Stack while referring to the implementation of "*Mesh Sample Program*" (R01AN5847). Figure 3-1 shows the sequence chart of Mesh Sample Program.

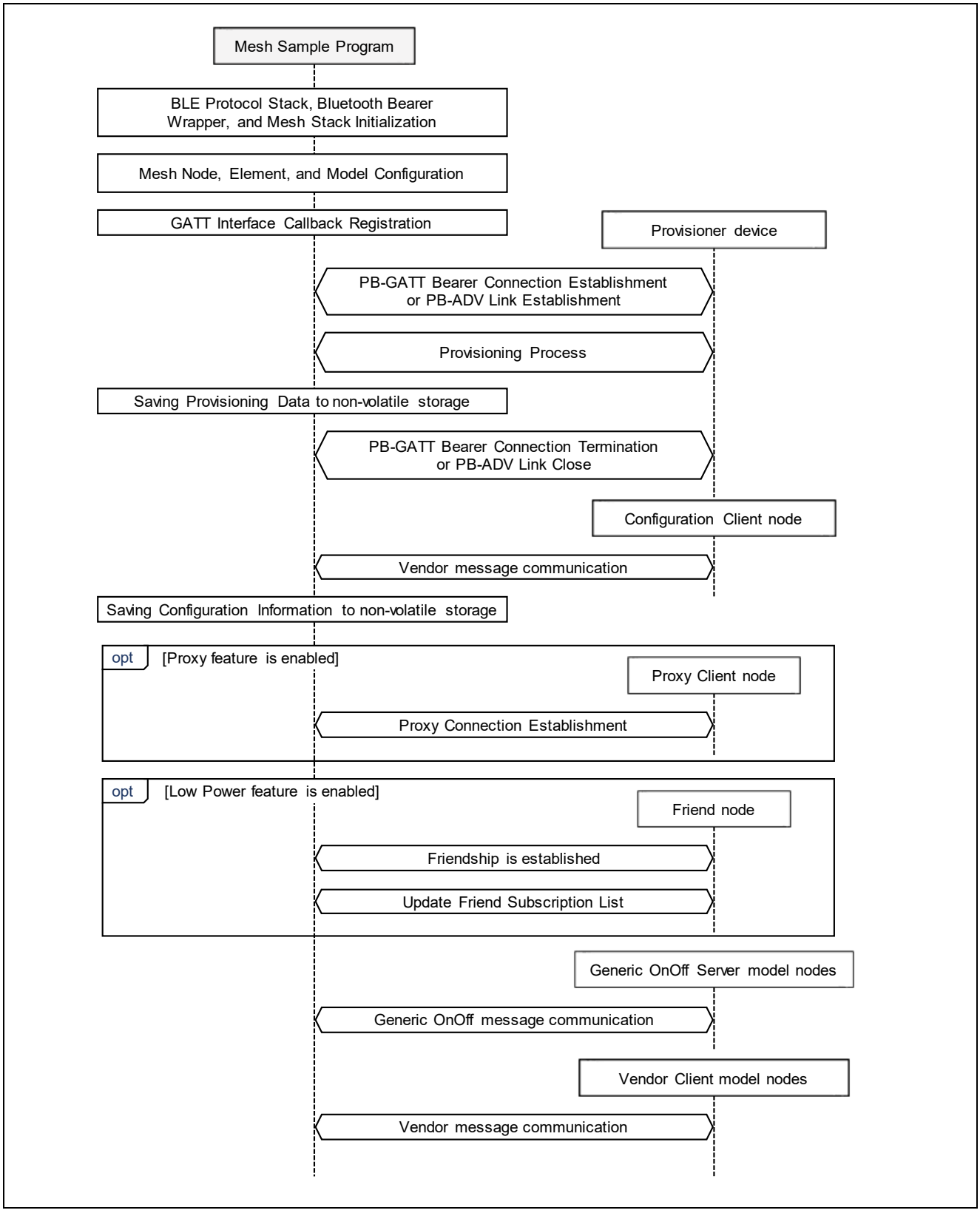


Figure 3-1 Sequence Chart of Mesh Sample Program

3.1 Main Routine

This section describes processing that should be implemented in main routine of user application. In the Mesh sample program given as an example, these processes are implemented in *app_main.c*.

- **Main Routine (app_main.c)**

The application must initialize the Bluetooth LE stack and Bluetooth bearer. These initialization processes are performed by the Bluetooth LE stack scheduler. In the mesh sample program in the bearmetal environment, the *R_BLE_Execute()* API of the scheduler API is repeatedly called in the while infinite loop. The mesh sample program in the FreeRTOS environment creates a task that repeatedly called the scheduler API. The completion of initialization is notified to the callback function described later in this section.

```
void app_main(void)
{
    API_RESULT retval;
.........
    /* Initialize BLE Protocol Stack */
    R_BLE_Open();

#if (BSP_CFG_RTOS == 2)
    /* Create Semaphore */
    g_semaphore = xSemaphoreCreateBinary();

    /* Get Current Task handle */
    g_ble_core_task = xTaskGetCurrentTaskHandle();

    /* Create Execute Task */
    xTaskCreate(execute_task_entry, "execute_task", 1280, &g_ble_core_task,
                                                                    4, &g_exe_task);
#endif /* (BSP_CFG_RTOS == 2) */

.........
    /* Initialize underlying BLE Protocol Stack to use as a Mesh Bearer */
    retval = RM_MESH_BEARER_PLATFORM_Open(&g_rm_mesh_bearer_platform0_ctrl,
                                          &g_rm_mesh_bearer_platform0_cfg);
.........

<Continue to next page>
```

FreeRTOS case.

Create task for periodically calling *R_BLE_Execute* API.

<Continue from previous page>

```
    while (1)
    {

/* When this BLE application works on the FreeRTOS */

#if (BSP_CFG_RTOS == 2)
        if(0 != R_BLE_IsTaskFree())
        {
            vTaskSuspend(NULL);
        }
        else
        {
            xSemaphoreGive(g_semaphore);
        }
#else /* (BSP_CFG_RTOS == 2) */
        /* Process BLE Event */
        R_BLE_Execute();
#endif /* (BSP_CFG_RTOS == 2) */

.........
}

.........

#if (BSP_CFG_RTOS == 2)
void execute_task_entry(void *pvParameters)
{
    FSP_PARAMETER_NOT_USED(pvParameters);
    while(1)
    {
        xSemaphoreTake(g_semaphore, portMAX_DELAY);

        while(0 == R_BLE_IsTaskFree())
            R_BLE_Execute();

        vTaskResume(g_ble_core_task);
    }
}
.........
#endif /* (BSP_CFG_RTOS == 2) */
```

FreeRTOS case.

If there is a task to be executed in the scheduler, transfer to task for periodically calling *R_BLE_Execute* API.

Baremetal case.

periodically calling *R_BLE_Execute* API.

FreeRTOS case.

periodically calling *R_BLE_Execute* API.

- **Callback Function of Bluetooth Bearer Initialization Completion**

The application must implement a callback function that receives Bluetooth bearer initialization complete notification mentioned in previous page. In the Mesh sample program, the callback function is implemented in the *blebrr_init_cb* function of *app_main.c*. In this callback function, the Mesh stack itself is initialized and the Bluetooth bearer is registered to the Mesh stack by calling *RM_BLE_Mesh_Open* API and *RM_MESH_BEARER_PLATFORM_Setup* API. After these initializations, the Mesh application can be started. In the Mesh sample program given as an example, these processes are implemented as following. The *mesh_model_config* function and *mesh_core_setup* functions are explained in the following sections.

```
static void blebrr_init_cb(st_ble_dev_addr_t * own_addr)
{
……
    /* Initialize Mesh Stack */
    RM_BLE_MESH_Open(&g_ble_mesh0_ctrl, &g_ble_mesh0_cfg);

……

    /* Registers ADV Bearer with Mesh Stack and Start Scan */
    RM_MESH_BEARER_PLATFORM_Setup(&g_ble_mesh_bearer_platform0_ctrl);

……

    /* Start Mesh Application */
    mesh_model_config(&gs_mesh_model_callbacks);

    mesh_core_setup();
}
```

Node configuration.

Start beaconing.

- **Mesh Stack Termination**

When Mesh Stack is no longer needed, Mesh Stack can be terminated by *RM_BLE_MESH_Close()*.
If Light LC Server Model is used, Light LC Server Model is terminated by *RM_MESH_LIGHT_LC_SRV_Close()*. Health Server Model is terminated by *RM_MESH_HEALTH_SERVER_Close()*.Mesh Stack is terminated by *RM_BLE_MESH_Close()*. Resources used by Bluetooth Bearer are freed by *RM_MESH_BEARER_PLATFORM_Close()*.
When Bluetooth LE Stack is no longer needed, Bluetooth LE Stack can be terminated by *R_BLE_Close()*.

```
    /* Deinitialize Light LC Server Model, if it was initialized */
    RM_MESH_LIGHT_LC_SRV_Close(&g_rm_mesh_light_lc_srv0_ctrl);

    /* Deinitialize Health Server Model */
    RM_MESH_HEALTH_SERVER_Close(&g_rm_mesh_health_srv0_ctrl);

    /* Terminate Mesh Stack */
    RM_BLE_MESH_Close(&g_rm_ble_mesh0_ctrl);

    /* Free the resources allocated by Bluetooth Bearer */
    RM_MESH_BEARER_PLATFORM_Close(&g_rm_mesh_bearer_platform0_ctrl);

    /* Terminate Bluetooth LE Protocol Stack */
    R_BLE_Close();
```

3.2 Node Configuration

The application needs to set up node configurations that include elements and models. This configuration depends on the user scenario. In the Mesh sample program, the node configuration is set by the *mesh_model_config* function implemented in *mesh_model.c* as follows.

```
API_RESULT mesh_model_config(const mesh_model_callbacks_t * callbacks)
{
    API_RESULT retval;

.........

    /* Create Node *//* Register Element */
    retval = g_rm_ble_mesh_access0.p_api->open(&g_rm_ble_mesh_access0_ctrl,
                                               &g_rm_ble_mesh_access0_cfg);
.........
    retval = mesh_foundation_model_register();

.........

    retval = mesh_application_model_register();


    retval = mesh_application_model_states_init();
.........
}
```

Register foundation model.

Register models used in application.

Initialize states referred in application.

3.3 Provisioning

After the node configuration is complete, the application should perform provisioning procedure as a provisioning server and receive provisioning data from provisioning client to join the network. The provisioning process of the Mesh sample program is implemented in mesh_core.c as follows.

3.3.1 Provisioning Server

(1) Registration of Provisioning Capabilities and Provisioning Callback Function (mesh_core.c)

In *mesh_core_setup* function of *mesh_core.c*, register a callback function for handling provisioning event and starting broadcast beacons when the device has not been provisioned. The callback function that implements provisioning event handling is specified in the *Provision Callback property* of the *rm_ble_mesh_provision* module in FSP configurator. *mesh_core_setup* function also implements the processing when the device has been provisioned. See section 3.4 for the implementation of this part.

```
API_RESULT mesh_core_setup(void)
{
    API_RESULT retval = API_SUCCESS;
……
    retval = RM_MESH_BEARER_PLATFORM_CallbackSet(&g_rm_mesh_bearer_platform0_ctrl,
                                                 mesh_core_gatt_iface_cb);
……

    /* Check if Provisioning is not complete */
    if (API_SUCCESS != mesh_core_get_primary_unicast_address(&addr))
    {

       /* Register Provisioning capabilities */
       retval = (API_RESULT) RM_BLE_MESH_PROVISION_Open(
                                                 &g_rm_ble_mesh_provision0_ctrl,
                                                 &g_rm_ble_mesh_provision0_cfg);

       if ((FSP_SUCCESS == retval) || (FSP_ERR_ALREADY_OPEN == retval))
       {

            /* Configure as Unprovisioned Device (Provisioning Server) */
            retval = mesh_core_prov_setup(RM_BLE_MESH_PROVISION_ROLE_DEVICE,
                                          RM_BLE_MESH_PROVISION_BEARER_TYPE_ADV |
                                          RM_BLE_MESH_PROVISION_BEARER_TYPE_GATT);
       }
    }
    else
    {
        /* Configure as a Proxy Server and Start Connectable Advertising */
        mesh_core_proxy_setup ();
        mesh_core_proxy_start(RM_BLE_MESH_NETWORK_GATT_PROXY_ADV_MODE_NET_ID);
……
    }
……
}
```

Register callback function for processing provisioning event.

Specify provisioning attribute and start broadcasting beacons.

See section 3.4.

(2) Start of Provisioning (mesh_core.c)

In *mesh_core_prov_setup* function of *mesh_core.c,* starts beacon transmission based on the attribute specified by the argument.

```
static API_RESULT mesh_core_prov_setup(rm_ble_mesh_provision_role_t role,
                                       rm_ble_mesh_provision_bearer_type_t brr)
{
    API_RESULT retval;
……
    if (RM_BLE_MESH_PROVISION_BEARER_TYPE_GATT & brr)
    {
        RM_MESH_BEARER_PLATFORM_SetGattMode(&g_rm_mesh_bearer_platform0_ctrl,
                                            RM_MESH_BEARER_PLATFORM_GATT_MODE_PROVISION);
    }

……
    retval = (API_RESULT)RM_BLE_MESH_PROVISION_Setup
                (
                    &g_rm_ble_mesh_provision0_ctrl,
                    role,
                    info,
                    CORE_PROV_SETUP_TIMEOUT_SECS
                );
……
    if (API_SUCCESS == retval)
    {
        if ((RM_BLE_MESH_PROVISION_ROLE_DEVICE == role) &&
              (RM_BLE_MESH_PROVISION_BEARER_TYPE_ADV & brr))
        {
            mesh_core_prov_bind(RM_BLE_MESH_PROVISION_BEARER_TYPE_ADV,
                                &gs_prov_device);
        }
    }
……
}
```

When PB-GATT use as provisioning bearer.

Configure provisioning parameters.

Start broadcasting beacon.

(3) Provisioning Callback Function (mesh_core.c)

Implement a callback function to receive Provisioning events. Provisioning Data provided from a Provisioning Client is required to be registered with Mesh Stack by *setProvisioningData* API of *rm_ble_mesh_access* module in *RM_BLE_MESH_PROVISION_EVENT_TYPE_PROVDATA_INFO* event.

```
void mesh_core_prov_cb(rm_ble_mesh_provision_callback_args_t * p_args)
{
……

    switch (p_args->event_type)
    {
……
        case RM_BLE_MESH_PROVISION_EVENT_TYPE_PROVDATA_INFO:
            rdata = (rm_ble_mesh_provision_data_t *)(p_args->event_data.payload);
            /* Provide Provisioning Data to Access Layer */
            retval = g_rm_ble_mesh_access0.p_api->
                     setProvisioningData(&g_rm_ble_mesh_access0_ctrl, rdata);
            break;
……
    }

    return;
}
```

(4) Cancellation of provisioning (app_main.c)

When provisioning is completed, 11 bytes of magic number indicating that provisioning has been completed is saved from top of the data flash (address 0x40100000). The magic number can be deleted at any time using *reset* API of the *rm_ble_mesh_access* module and the transmission of the un-provisioned device beacon can be resumed after rebooting a program. Sample program included with "*RA4W1 Group Bluetooth Mesh sample application*" (R01AN5848), the magic number will be erased when SW1 mounted on EK-RA4W1 is pressed and rebooted.

```
static void platform_reboot_timer_cb(void)
{
    API_RESULT retval;
    retval = g_rm_ble_mesh_access0.p_api->reset(&g_rm_ble_mesh_access0_ctrl);
……
}
```

3.3.2 Provisioning Sequence

(1) Provisioning Setup

This sample program supports both PB-ADV bearer and PB-GATT bearer and transmits Unprovisioned Device beacon for PB-ADV bearer and connectable advertising for PB-GATT bearer alternately.

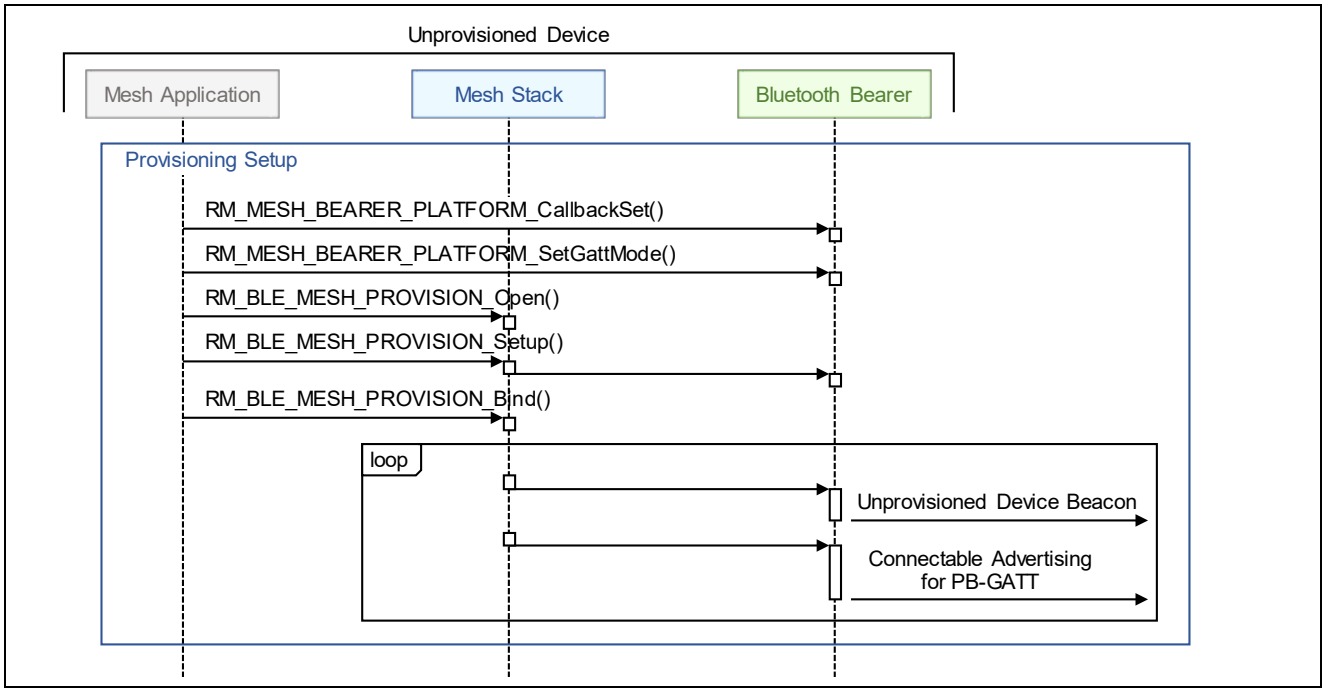


Figure 3-2 Provisioning Setup

(2) Session Establishment over PB-ADV

To perform Provisioning Process over PB-ADV, Provisioning Server establishes a session with Provisioning Client. Also, Provisioning Server closes a session after Provisioning Process.

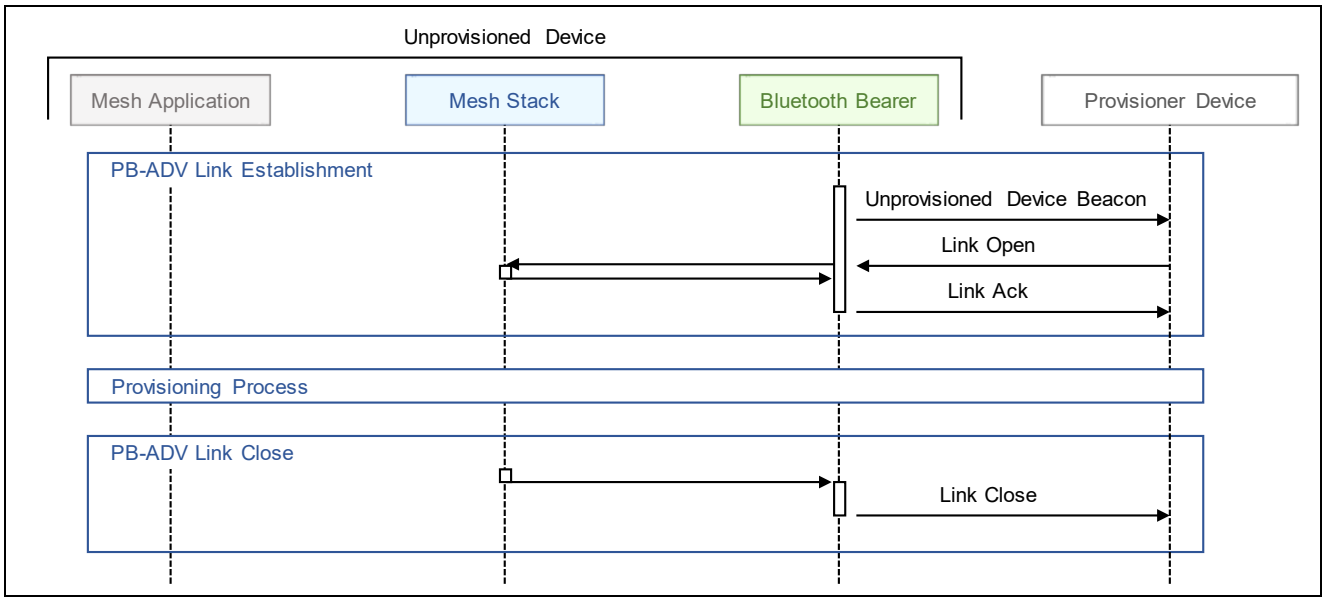


Figure 3-3 Session Establishment over PB-ADV

(3) Connection Establishment over PB-GATT

To perform Provisioning Process over PB-GATT, Provisioning Client establishes a connection with Provisioning Server and enables notification of the Mesh Provisioning Service. Also, Provisioning Client terminates the connection after Provisioning Process.

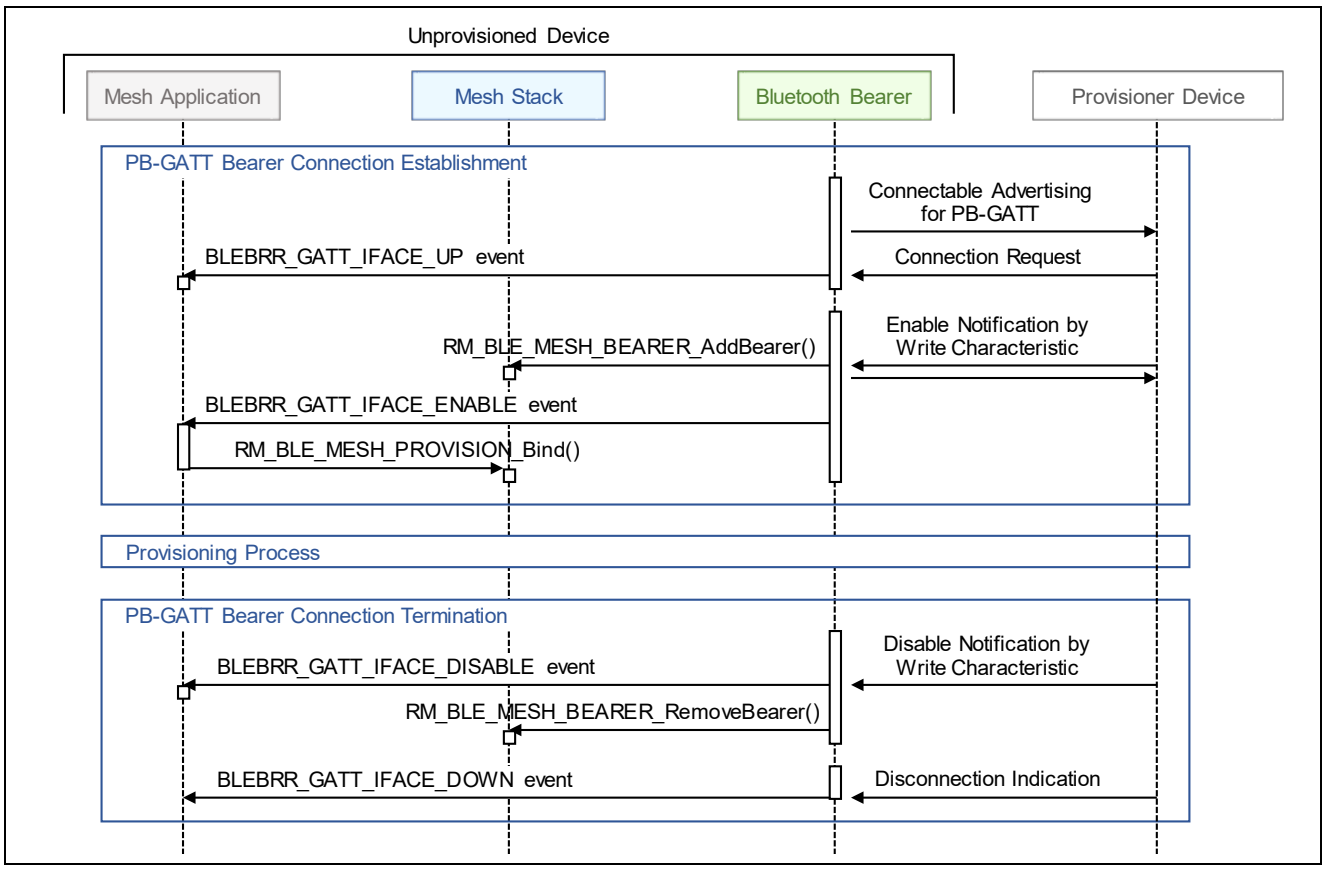


Figure 3-4 Connection Establishment over PB-GATT

(4) Provisioning Process

After establishing a session or a connection over Provisioning Bearer, Provisioning Process, from Invitation to Distribution of provisioning data, is performed and Provisioning PDUs are exchanged. The same process is performed over either PB-ADV or PB-GATT during Provisioning Process.

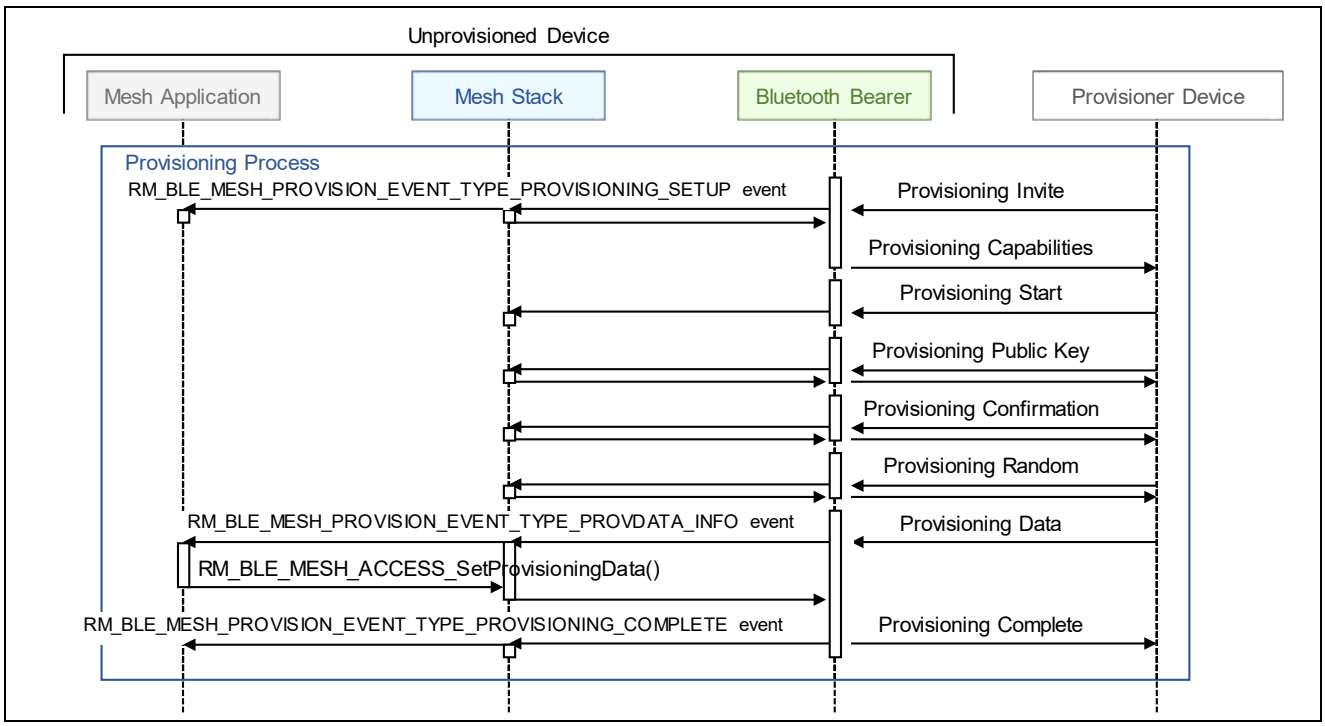


Figure 3-5 Provisioning Process

To reduce security risk of Provisioning Procedure, the followings are recommended:

- Using OOB (Out Of Band) in Public Key Exchange step.
  - To deliver Public Key via OOB, generate a Public and a Private Key by *RM_BLE_MESH_PROVISION_GenerateEcdhKey*() and set the Public Key to Mesh Stack by *RM_BLE_MESH_PROVISION_SetLocalPublicKey*(). Deliver the Public Key to peer Provisioner device via OOB.

- Selecting a cryptographically secure random value or a pseudorandom number having the maximum permitted 128bit entropy as AuthValue in Authentication step.
  - 128bit random number, that can be used as a AuthValue for OOB Authentication, can be generated by *RM_BLE_MESH_PROVISION_GenerateRandomizedNumber*(). When Static OOB Authentication is available. set the generated AuthValue to Mesh Stack with RM_BLE_MESH_PROVISION_SetOobAuthInfo(). When Output OOB Authentication is available. set the generated AuthValue to Mesh Stack with *RM_BLE_MESH_PROVISION_SetAuthVal*().

Figure 3-6 shows the provisioning process flow when using OOB.

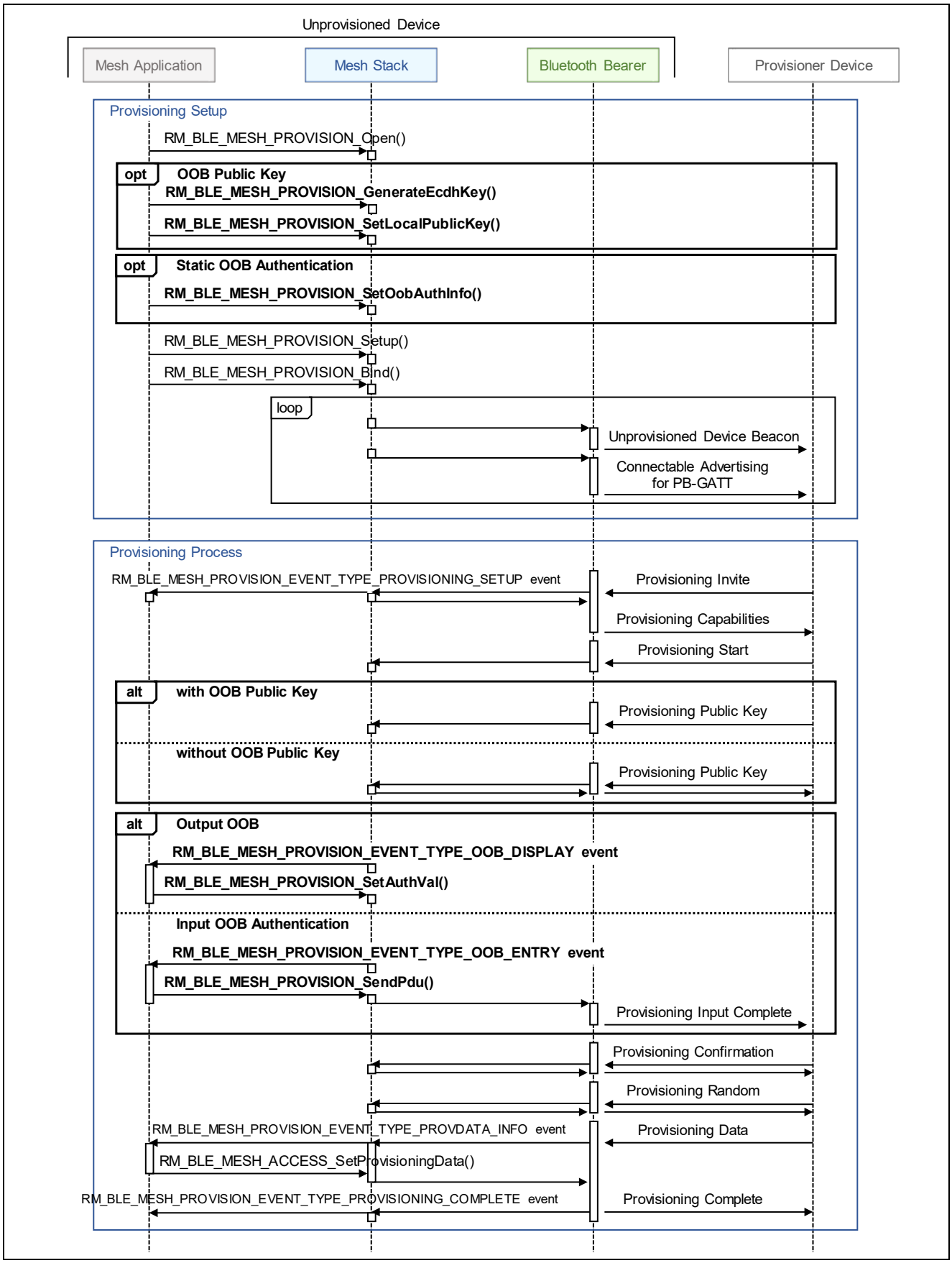


Figure 3-6 Provisioning Process with OOB

3.4 Proxy

Mesh sample program can work as a proxy server or proxy client. This section describes the implementation required to work as a proxy server or proxy client.

3.4.1 Proxy Server

Processing for working as a Proxy Server is shown as below.

(1) Registration of Proxy Callback Function (mesh_core.c)

Set Bluetooth Bearer Mode to *BLEBRR_GATT_PROXY_MODE* by using *RM_MESH_BEARER_PLATFORM_SetGattMode*() API. The callback function that implements proxy server related event handling is registered in the mesh stack by using *RM_BLE_MESH_NETWORK_Open*() API. The callback function can be specified *Callback* property of *rm_ble_mesh_network* module in FSP configuration. These processes are implemented in the *mesh_core_proxy_setup* function of *mesh_core.c* in the Mesh sample program.

```
static API_RESULT mesh_core_proxy_setup(void)
{
    API_RESULT retval;

    RM_MESH_BEARER_PLATFORM_SetGattMode(&g_rm_mesh_bearer_platform0_ctrl,
                                        RM_MESH_BEARER_PLATFORM_GATT_MODE_PROXY);

    /* Register Proxy Callback */
    retval = RM_BLE_MESH_NETWORK_Open(&g_rm_ble_mesh_network0_ctrl,
                                      &g_rm_ble_mesh_network0_cfg);
……
    return retval;
}
```

(2) Starting Connectable Advertising (mesh_core.c)

To establish a Proxy connection with Proxy Client, start Connection Advertising by *RM_BLE_MESH_NETWORK_StartProxyServerAdv*() API. These processes are implemented in the *mesh_core_proxy_start* function of *mesh_core.c* in the Mesh sample program.

```
API_RESULT mesh_core_proxy_start(uint8_t proxy_adv_mode)
{
……
    if (0 != proxy_adv_mode)
    {
       RM_BLE_MESH_NETWORK_StartProxyServerAdv(&g_rm_ble_mesh_network0_ctrl,
                                RM_BLE_MESH_NETWORK_PRIMARY_SUBNET, proxy_adv_mode);
        CONSOLE_STATUS("[PROXY] RM_BLE_MESH_NETWORK_StartProxyServerAdv()", retval);
……
    }
}
```

(3) Proxy Callback Function (mesh_core.c)

Implement a callback function to receive Proxy events. The callback function implemented in the *mesh_core_proxy_cb* function of *mesh_core.c* in the Mesh sample program. When a connection is established and GATT Proxy Service is enabled, *RM_BLE_MESH_NETWORK_EVENT_PROXY_UP* event will be notified to the callback function. To deliver Key Refresh Flag, IV Update Flag, and current IV Index to Proxy Client. send Secure Network Beacon by using *RM_BLE_MESH_NETWORK_BroadcastSecureBeacon*() API.

```
void mesh_core_proxy_cb(rm_ble_mesh_network_callback_args_t * p_args)
{
……
    switch (p_args->event)
    {
        case RM_BLE_MESH_NETWORK_EVENT_PROXY_UP:
……
                for (subnet_handle = 0; subnet_handle <
                      g_rm_ble_mesh0_cfg.maximum_subnets; subnet_handle++)
                {
……
                    retval =(API_RESULT)RM_BLE_MESH_NETWORK_BroadcastSecureBeacon
                                        (&g_rm_ble_mesh_network0_ctrl, subnet_handle);
……
                }
……
    }
}
```

(4) Terminating Proxy Connection (mesh_core.c)

To terminate a connection, call *RM_MESH_BEARER_PLATFORM_Disconnect*() API. In the Mesh sample program, it is implemented in the *mesh_core_proxy_disconnect* function of *mesh_core.c*.

```
API_RESULT mesh_core_proxy_disconnect(void)
{
    API_RESULT retval = API_SUCCESS;

    for (uint8_t idx = 0; idx < CORE_NUM_GATT_INTERFACES; idx++)
    {
        if (BLE_GAP_INVALID_CONN_HDL != gs_proxy_client_conn_hdl[idx])
        {
            retval = (API_SUCCESS ==
                        RM_MESH_BEARER_PLATFORM_Disconnect(&g_rm_mesh_bearer_platform
                        0_ctrl, gs_proxy_client_conn_hdl[idx])) ? retval :
                        API_FAILURE;
        }
    }

    return retval;
}
```

3.4.2 Proxy Client

Processing for working as a Proxy Client is shown as below.

(1) Registration of Proxy Callback Function (appl_proxy.c)

Refer to section “3.4.1(1) Registration of Proxy Callback Function (mesh_core.c)”.

(2) Establishing Proxy Connection (cli_brr.c)

Call *RM_MESH_BEARER_PLATFORM_Connect*() API to establish a proxy connection with the proxy server. In the Mesh sample program, the connection procedure is implemented as *connect* command by user. Refer to the *cli_create_gatt_conn* function in *./src/mesh/mesh_cli/cli_brr.c*.

```
API_RESULT cli_create_gatt_conn(uint32_t argc, uint8_t *argv[])
{
    st_ble_dev_addr_t peer_bd_addr;
    uint8_t service_mode;
    API_RESULT retval;

……

    peer_bd_addr.type = (uint8_t)CLI_strtoi(argv[0],
                                            (uint8_t)strlen((char*)argv[0]), 16);
    CLI_strtoarray_le
    (
        argv[1],
        (uint16_t)strlen((char*)argv[1]),
        &peer_bd_addr.addr[0],
        6
    );
    service_mode = (uint8_t)CLI_strtoi(argv[2], (uint8_t)strlen((char*)argv[2]), 16);

……
    retval =    RM_MESH_BEARER_PLATFORM_Connect(&g_rm_mesh_bearer_platform0_ctrl,
                    (uint8_t*)&(peer_bd_addr.addr), peer_bd_addr.type, service_mode);

……
}
```

The scan operation to search the proxy server is also implemented as *scan* command input by the user. As a result of the scan, the device address of the found proxy server is notified in the *BLEBRR_GATT_IFACE_SCAN* event to GATT callback function specified by *RM_MESH_BEARER_PLATFORM_CallbackSet* () API in section 3.3.1.

(3) Proxy Callback Function (mesh_core.c)

Implement a callback function to receive Proxy events. In the Mesh sample program, the callback function is implemented as *mesh_core_procy_cb* function in *mesh_core.c*. When a connection is established and GATT Proxy Service is enabled, *RM_BLE_MESH_NETWORK_EVENT_PROXY_UP* event will be notified. Configure Proxy Filter Type of Proxy Server with *RM_BLE_MESH_NETWORK_SetProxyFilter*() API, add Subscription Address to Proxy Filter List of Proxy Server by g*etAllModelSubscriptionList* () API of *rm_ble_mesh_access* module and *RM_BLE_MESH_NETWORK_ConfigProxyFilter*() API in the event handling part.

```
void mesh_core_proxy_cb(ble_mesh_network_callback_args_t * p_args)
{
    switch (event)
    {
        case RM_BLE_MESH_NETWORK_EVENT_PROXY_UP:
            retval = (API_RESULT)RM_BLE_MESH_NETWORK_SetProxyFilter(NULL, &route_info,
                                                BLE_MESH_NETWORK_PROXY_FILTER_TYPE_WHITELIST);
            gs_proxy_opcode = RM_BLE_MESH_NETWORK_PROXY_CONFIG_OPECODE_SET_FILTER;
            break;

        case RM_BLE_MESH_NETWORK_EVENT_PROXY_STATUS:
            switch (gs_proxy_opcode)
            {
                case RM_BLE_MESH_NETWORK_PROXY_CONFIG_OPECODE_SET_FILTER:
                    /* Add Subscription Addresses to Proxy filter list */
                    retval = mesh_core_proxy_add_addresses(handle,
                                                           BLE_MESH_NETWORK_PRIMARY_SUBNET);
                    gs_proxy_opcode = RM_BLE_MESH_NETWORK_PROXY_CONFIG_OPECODE_ADD_TO_FILTER;
                    break;
                }
            }
            break;
    }
}

static API_RESULT mesh_core_proxy_add_addresses(NETIF_HANDLE * netif_hdl, ble_mesh_network_subnet_handle_t
                                                subnet_hdl)
{
    PROXY_ADDR addr_list[g_rm_ble_mesh0_cfg.maximum_virtual_address +
g_rm_ble_mesh0_cfg.maximum_non_virtual_address];
    UINT16     addr_count = ARRAY_SIZE(addr_list);

    retval = g_ble_mesh_access0.p_api->getAllModelSubscriptionList(&g_ble_mesh_access0_ctrl,
                                                                   &addr_count, addr_list);
    if (0 != addr_count)
    {
        ble_mesh_network_route_info_t route_info;
        route_info.interface_handle = netif_hdl;
        route_info.subnet_handle = subnet_hdl;
        ble_mesh_network_proxy_address_list_t proxy_address_list;
        proxy_address_list.address = addr_list;
        proxy_address_list.count = addr_count;
        retval = (API_RESULT)RM_BLE_MESH_NETWORK_ConfigProxyFilter(NULL, &route_info,
                                           RM_BLE_MESH_NETWORK_PROXY_CONFIG_OPECODE_ADD_TO_FILTER,
                                           &proxy_address_list);
    }

    return retval;
}
```

(4) Terminating Proxy Connection (cli_brr.c)

Refer to section “3.4.1(4) Terminating Proxy Connection (mesh_core.c)”.

3.4.3 Proxy Sequence

(1) Proxy Setup

Mesh sample program supports Proxy feature so Configuration Client that supports only GATT bearer can configure the sample program over GATT bearer. Moreover, this sample program can forward messages between GATT bearer and ADV bearer for a node that supports only GATT bearer.

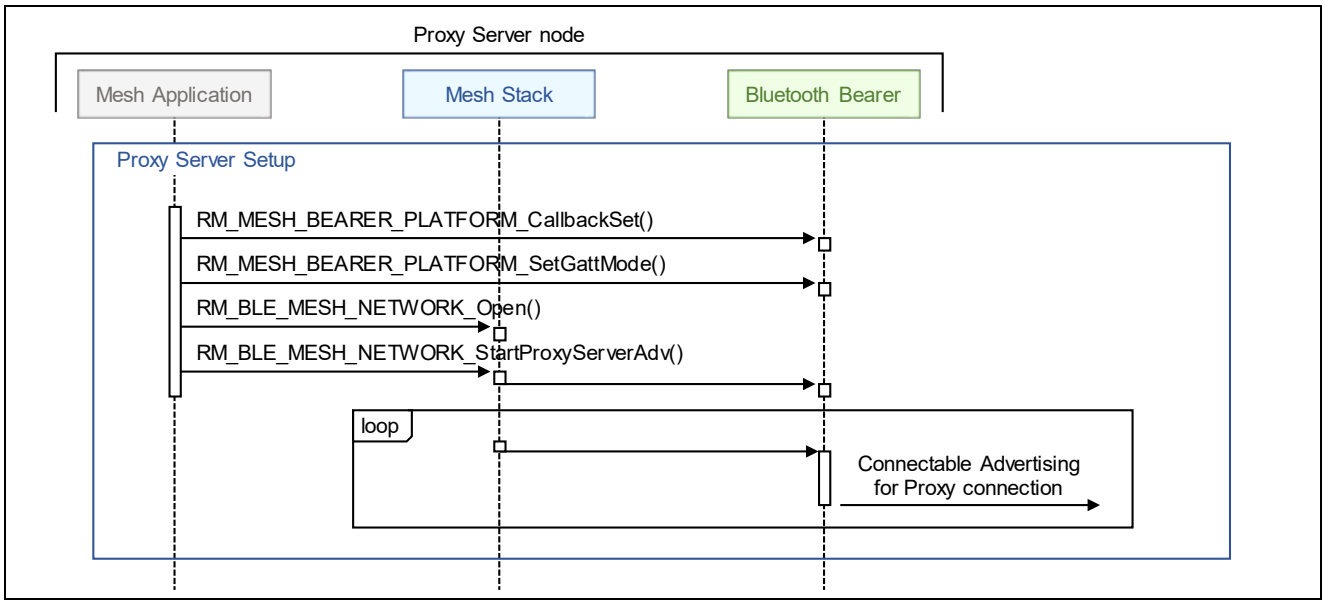


Figure 3-7 Proxy Setup

(2) Proxy Connection Establishment

Proxy Client establishes a connection with Proxy Server and enables notification of the Mesh Proxy Service. After enabling Notification, Proxy Client becomes able to perform Message Communication over GATT bearer.

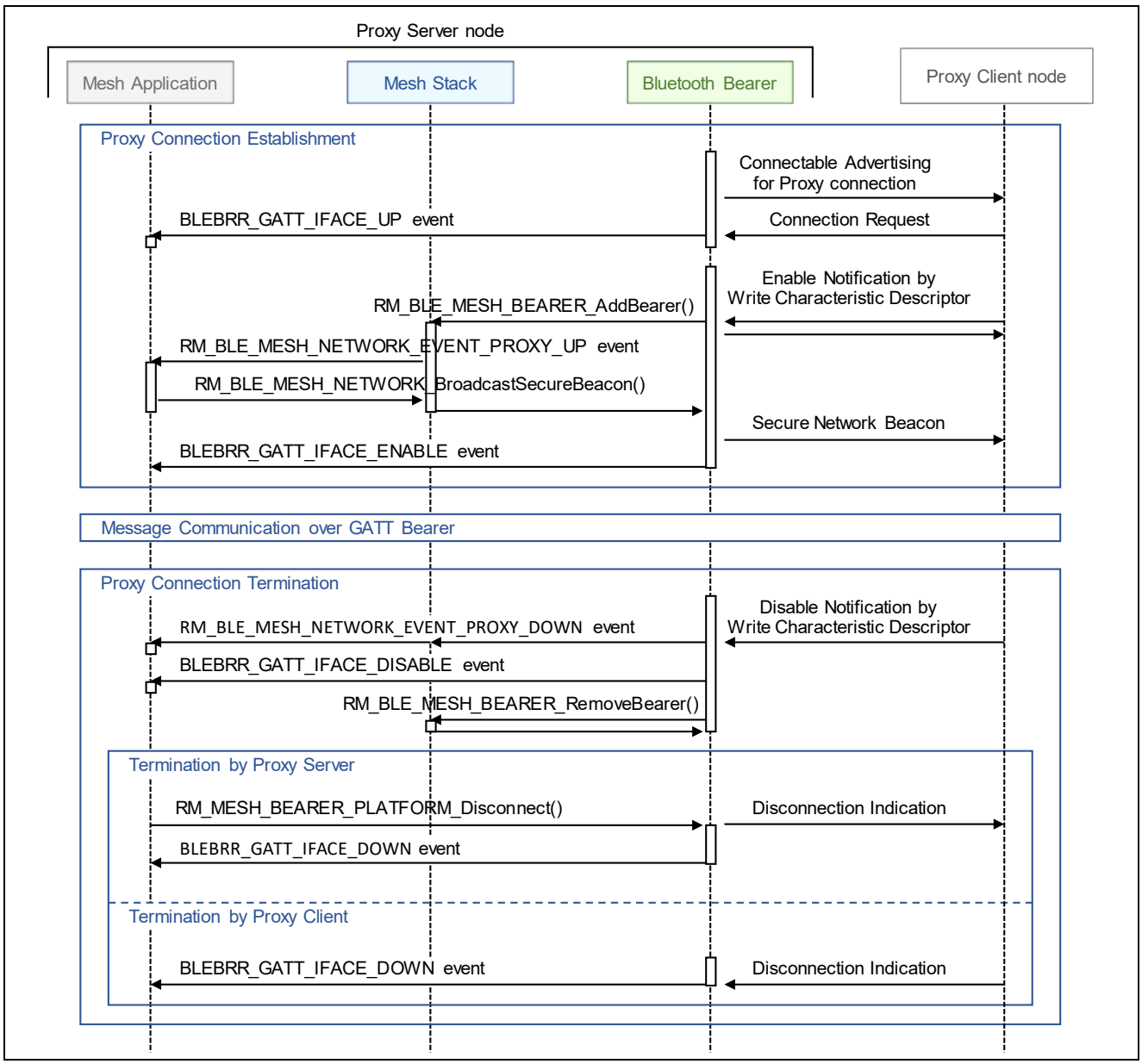


Figure 3-8 Proxy Connection Establishment and Termination

3.5 Friendship

This section describes how to implement for working as a Friend node or Low Power node.

3.5.1 Friend Node

To work as a Friend node, Friend feature must be enabled. Friend feature is enabled by the following way:

- Configuration Client sends Config Friend Set message.
- Application calls *setFeaturesField* () API of *rm_ble_mesh_access* module.

After enabling Friend feature, Friend-related-processing such as Friendship establishment, Friend Queue management, and response for Low Power node is handled automatically by Mesh Stack, so application does not have to handle it.

3.5.2 Low Power Node

To work as a Low Power node, application must enable Low Power feature and request a Friend node to establish a Friendship. After establishing a Friendship, Mesh Stack polls the Friend node if any messages are stored and suspends and resumes Scan automatically. Mesh Sample Program can work as a Low Power Node. Processing for establishing a Friendship as a Low Power node is shown as below.

NOTE: This feature is disabled by default in the Mesh sample application. To enable this feature, change the value of *LOW_POWER_FEATURE_EN* macro by referring to Section 2.7.

(1) Enabling Low Power feature (mesh_core.c)

Enable the low power feature by using *set_featureField* () API of *rm_ble_mesh_access* module when provisioning is completed. The API call is implemented in *mesh_core_prov_cb* function of *mesh_core.c* which explained in section 3.3.1.

```
void mesh_core_prov_cb(rm_ble_mesh_provision_callback_args_t * p_args)
{
……
    switch (p_args->event_type)
    {
……
        case RM_BLE_MESH_PROVISION_EVENT_TYPE_PROVISIONING_COMPLETE:
……

                #if LOW_POWER_FEATURE_EN
                g_rm_ble_mesh_access0.p_api->setFeaturesField
                          (&g_rm_ble_mesh_access0_ctrl, MS_ENABLE, MS_FEATURE_LPN);
                #endif /* LOW_POWER_FEATURE_EN */

                if (!gs_prov_is_gatt_iface)
                {
                  /* Configure as a Proxy Server and Start Connectable Advertising */
                    mesh_core_proxy_setup();

                    #if LOW_POWER_FEATURE_EN
                    /* Seek a Friend Node */
                    mesh_core_lpn_setup();
                    #endif /* LOW_POWER_FEATURE_EN */


……
            break;
……
    }
    return;
}
```

(2) Requirement for Friendship Establishment (mesh_core.c)

After enabling the low power feature, register the friendship callback function by using *RM_BLE_MESH_UPPER_TRANS_Open*() API. The callback function is specified by *Callback* property of *rm_ble_mesh_upper_trans* module in FSP configuration. After calling *RM_BLE_MESH_UPPER_TRANS_Open*() API, send *Friend Request* message by *RM_BLE_MESH_UPPER_TRANS_LpnSetupFriendship*() API to establish a friendship as a Low Power node. Set parameters related to timing of polling Friend node as arguments for this function.

```
#if LOW_POWER_FEATURE_EN
API_RESULT mesh_core_lpn_setup(void)
{
    API_RESULT retval;

    RM_BLE_MESH_UPPER_TRANS_Open(&g_ble_mesh_upper_trans0_ctrl, &g_ble_mesh_upper_trans0_cfg);

    ble_mesh_upper_trans_friendship_setting_t friendship_setting;
    friendship_setting.subnet_handle = 0x00;
    friendship_setting.criteria = CORE_FRIEND_CRITERIA;
    friendship_setting.rx_delay = CORE_FRIEND_RECEIVE_DELAY;
    friendship_setting.poll_timeout = CORE_FRIEND_POLLTIMEOUT;
    friendship_setting.setup_timeout = CORE_FRIEND_SETUPTIMEOUT;

    retval = RM_BLE_MESH_UPPER_TRANS_LpnSetupFriendship
                (
                    &g_ble_mesh_upper_trans0_ctrl,
                    &friendship_setting
                );

    return retval;
}
#endif /* LOW_POWER_FEATURE_EN */
```

(3) Friendship Callback Function (mesh_core.c)

Implement a callback function to receive Friendship events notified by Mesh Stack. In the mesh sample program, the callback function is implemented as *mesh_core_lpn_cb* of *mesh_core.c*. When a friendship is established, *RM_BLE_MESH_UPPER_TRANS_EVENT_FRIENDSHIP_SETUP* will be notified. Also, when a friendship is terminated, *RM_BLE_MESH_UPPER_TRANS_EVENT_FRIENDSHIP_TERMINATE* will be notified.

Low Power node can add and remove Subscription Addresses to/from Friend Subscription List of Friend node. After establishing a Friendship, Mesh Sample Program gets all Subscription Addresses from Subscription List by using *getAllModelSubscriptionListI* API of *rm_ble_mesh_access* module and edit the subscription list by using *RM_BLE_MESH_UPPER_TRANS_LpnManageSubscription*() API.

```
void mesh_core_lpn_cb(ble_mesh_upper_trans_callback_args_t * p_args)
{
    UCHAR  seek_req = MS_FALSE;
    UINT16 subscrn_list[g_rm_ble_mesh0_cfg.maximum_friend_subscription_list];
    UINT16 subscrn_count = g_rm_ble_mesh0_cfg.maximum_friend_subscription_list;

    switch(event_type)
    {
        case RM_BLE_MESH_UPPER_TRANS_EVENT_FRIENDSHIP_SETUP:
        {
            /* Friendship is established. */
            if (API_SUCCESS == status)
            {
                retval = g_ble_mesh_access0.p_api->getAllModelSubscriptionList
                                    (&g_ble_mesh_access0_ctrl, &subscrn_count, subscrn_list);

                if (0 != subscrn_count)
                {
                    retval = RM_BLE_MESH_UPPER_TRANS_LpnManageSubscription(NULL,
                                RM_BLE_MESH_UPPER_TRANS_CONTROL_OPCODE_FRIEND_SUBSCRN_LIST_ADD,
                                addr_list, count);
                }
            }
            /* Setup timeout is expired, and Friendship is not established. */
            else
            {
                seek_req = MS_TRUE;
            }
        }
        break;

        case RM_BLE_MESH_UPPER_TRANS_EVENT_FRIENDSHIP_TERMINATE:
        {
            /* Friendship is terminated. */
            seek_req = MS_TRUE;
        }
        break;
    }
……
}
```

3.5.3 Low Power Node Sequence

(1) Enabling Low Power Feature and Friendship Request

This sample program supports Low Power feature and transmits Friend Request to establish a Friendship.

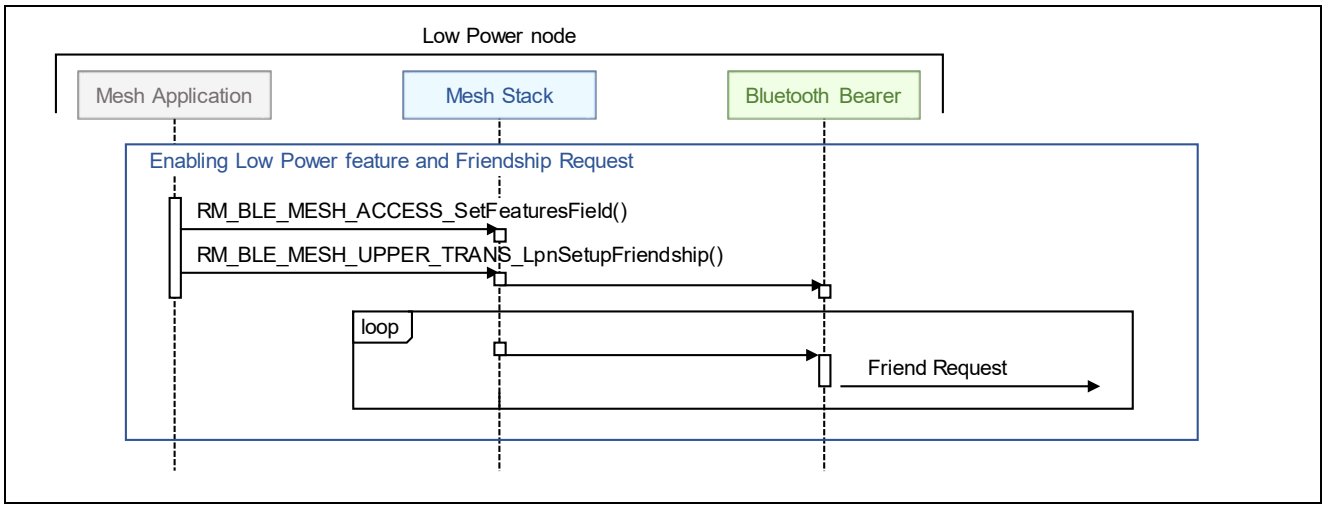


Figure 3-9 Enabling Low Power Feature and Friendship Request

(2) Friendship Establishment and Termination

Friendship is established by receiving Friend Offer. After Friendship establishment, this sample program registers all Subscription Addresses with Friend Subscription List of Friend node. After registration, the Friend node stores messages addressed to the Subscription Addresses. Also, Low Power node performs Message Polling periodically and receives messages from the Friend node.

When Message Polling fails continuously and then Friendship is terminated by arising Friend Poll Timeout, this sample program transmits Friend Request to establish a Friendship again.

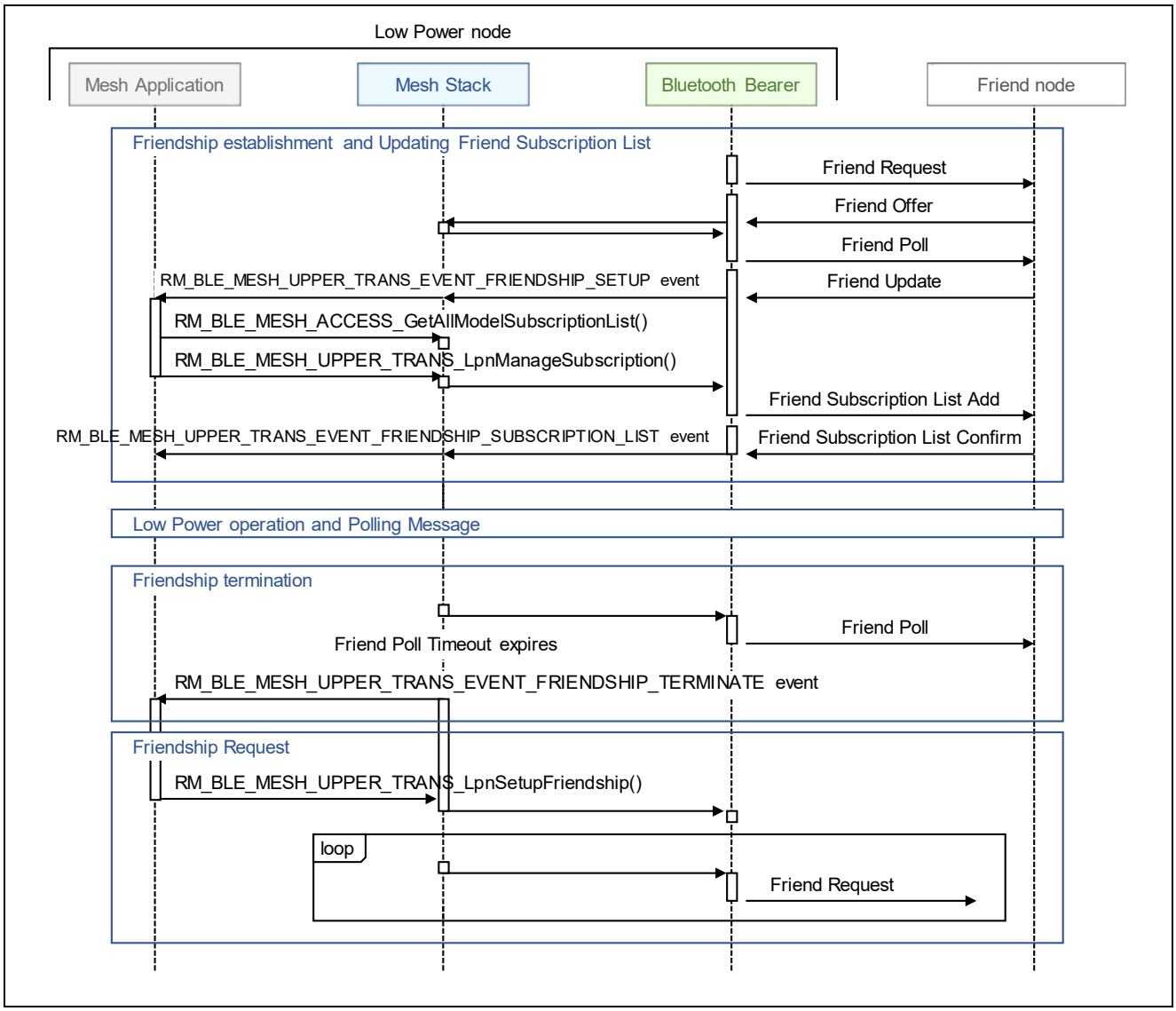


Figure 3-10 Friendship Establishment and Termination

If Low Power Node terminates a Friendship spontaneously, send Friend Clear message by *RM_BLE_MESH_UPPER_TRANS_LpnClearFriendship*() API. Completion of termination will be notified by *RM_BLE_MESH_UPPER_TRANS_EVENT_FRIENDSHIP_CLEAR* event.

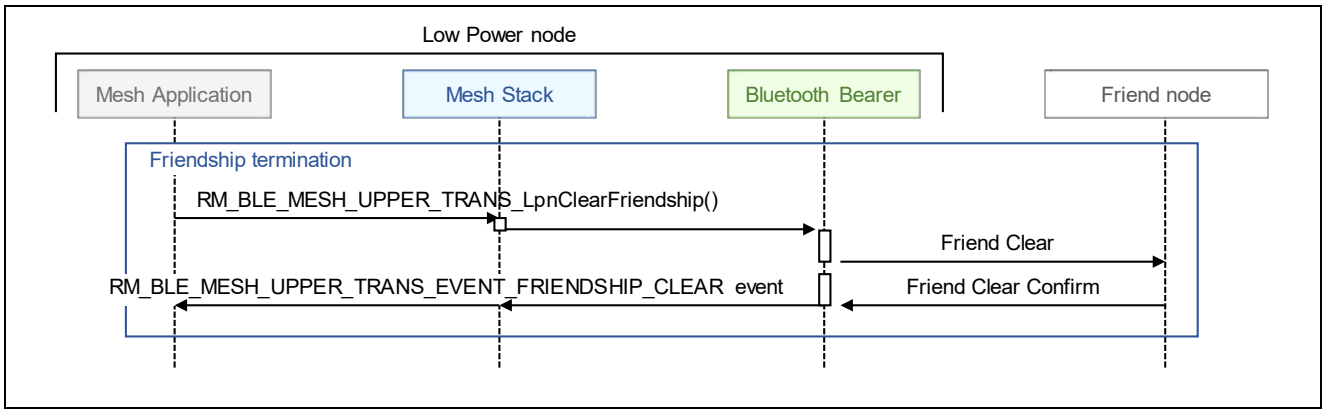


Figure 3-11 Friendship Termination

3.5.4 Friend Node Sequence

(1) Enabling Friend Feature

By being enabled Friend feature by Configuration Client, this sample program can work as a Friend Node. Mesh Stack does not notify any events regarding Friend Node operation such as friendship establishment and termination.

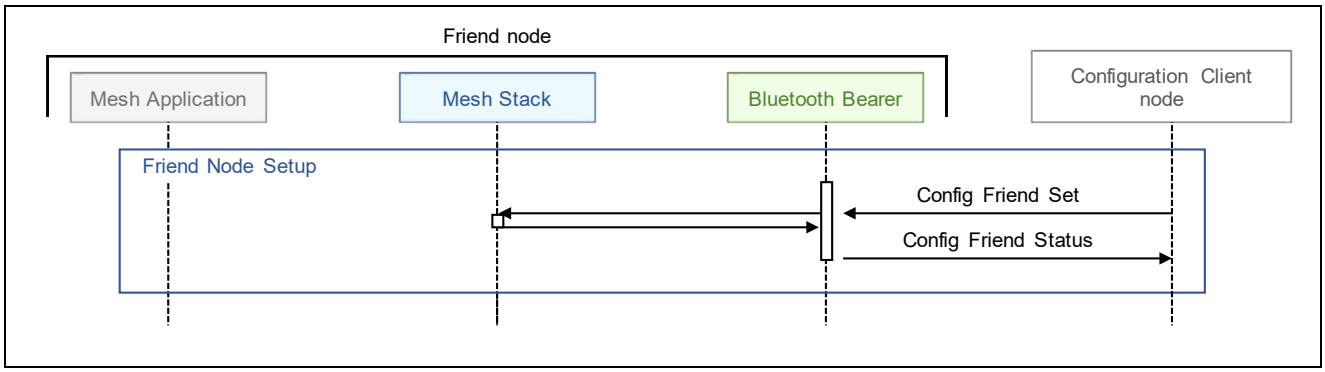


Figure 3-12 Enabling Friend Feature

(2) Friendship Establishment and Termination

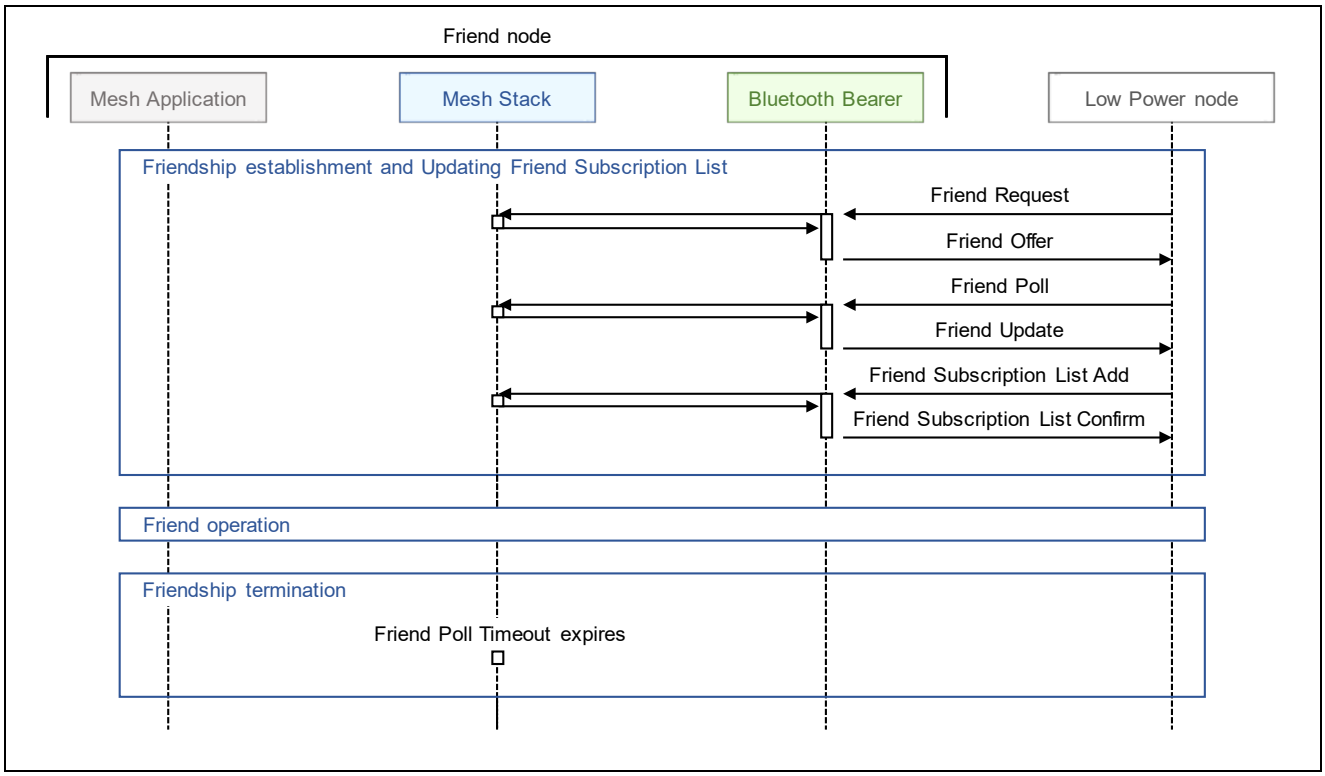


Figure 3-13 Friendship Establishment and Termination

3.6 Configuration

Mesh Sample Program works as a Configuration Server. Implementation for Configuration Server Model of Mesh Sample Program is shown as below.

3.6.1 Configuration Server

After Provisioning, a node must receive configuration information such as Application Key, Publish Address, and Subscription Address from Configuration Client so that a node communicates by each model. There configuration information is handled as Configuration States by Configuration model.

When application registers Configuration Server model, memory region for Configuration states are allocated in Mesh Stack. By receiving Configuration messages from Configuration Client, Mesh Stack updates the Configuration states automatically. Therefore, application does not have to manage the Configuration states.

Mesh Stack provides application with API to access local Configuration states. Application can access the Configuration states directly by using Mesh Stack API. Regarding API to access local Configuration states, refer to *RM_BLE_MESH_ACCESS_**() API in "*Renesas Flexible Software Package User's Manual*".

(1) Registration of Configuration Server Model (mesh_model.c)

Register Configuration Server model with element by *RM_MESH_CONFIG_SRV_Open*() API. In the mesh sample program, the registration process is implemented in *mesh_foundation_model_register* function called from *mesh_model_config* function which described in section 3.2.

```
static API_RESULT mesh_foundation_model_register(void)
{
    API_RESULT retval;

    retval = (API_RESULT)RM_MESH_CONFIG_SRV_Open(&g_rm_mesh_config_srv0_ctrl,
                                                 &g_rm_mesh_config_srv0_cfg);

    retval = RM_MESH_HEALTH_SERVER_Open(&g_rm_mesh_health_srv0_ctrl,
                                        &g_rm_mesh_health_srv0_cfg);

    return retval;
}
```

(2) Configuration Server Callback Function (mesh_model.c)

Implement a callback function to receive a message from Configuration Client. The callback function specified in *Callback* property of *rm_mesh_config_srv* module in FSP configuration. In this mesh sample application, the callback function is implemented as *mesh_model_config_server_cb*.

3.6.2 Configuration Server Sequence

When receiving *Config Node Reset* message, Mesh Stack delete all Configuration states. Also, this sample program resets MCU and performs Provisioning again.

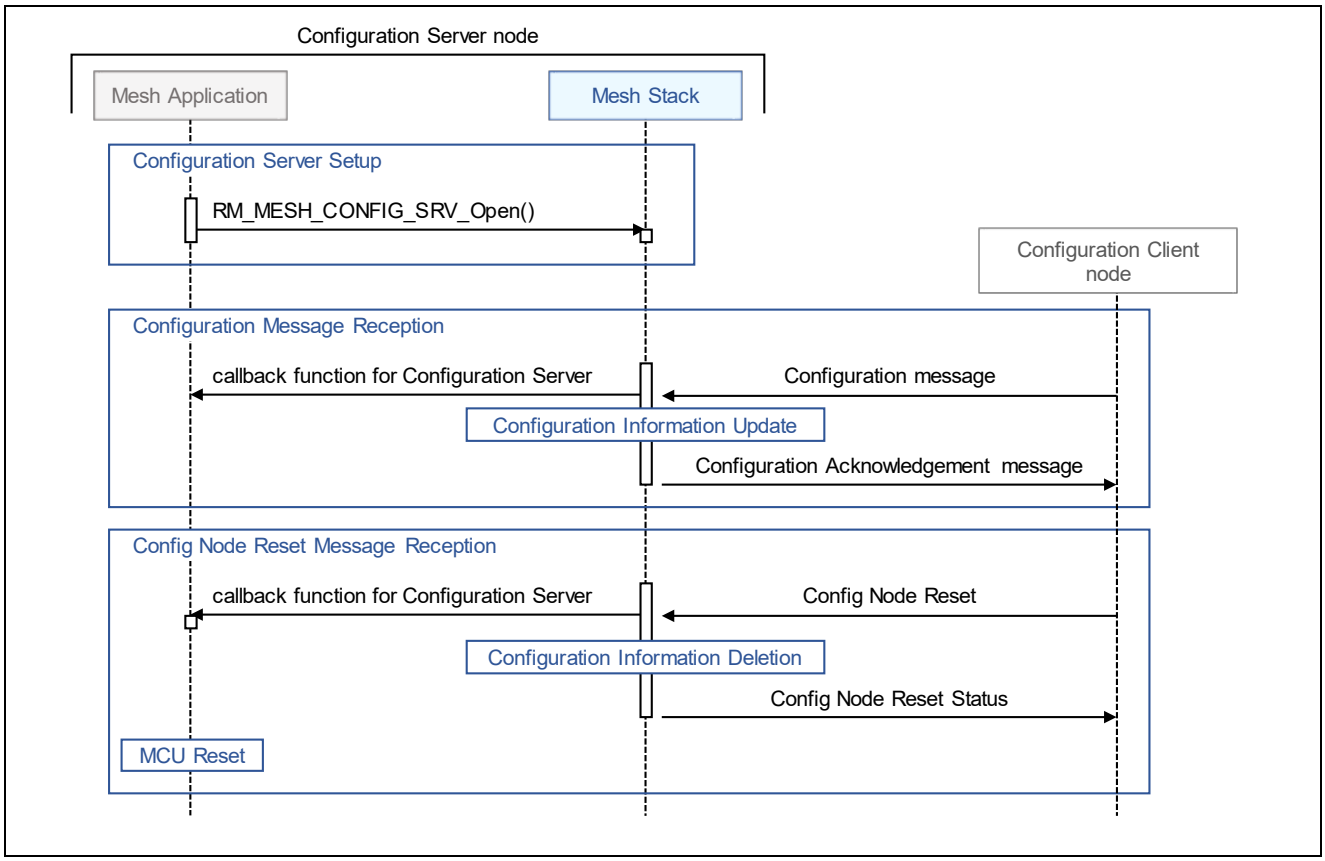


Figure 3-14 Configuration Server Model Operation of Mesh Sample Program

3.7 Health Model

This section describes how to implement for working as a Health Server.

3.7.1 Health Server

Health Server performs self-testing when receiving *Health Fault Test* message from Health Client. Also, Health Server starts Attention Timer when receiving *Health Attention Set* message from Health Client. In the mesh sample program, these procedures are implemented in *app_main.c*.

(1) Registration of Health Server Model (mesh_model.c)

Register Health Server model with element by *RM_MESH_HEALTH_SERVER_Open*() API. In the mesh sample program, the registration procedure is performed in the *mesh_founddation_model_register* function called from *mesh_model_config* function implemented in *mesh_model.c* which described in section 3.2.

```
static API_RESULT mesh_foundation_model_register (void)
{
    API_RESULT retval;

    retval = (API_RESULT)RM_MESH_CONFIG_SRV_Open(&g_rm_mesh_config_srv0_ctrl,
                                                 &g_rm_mesh_config_srv0_cfg);

    retval = RM_MESH_HEALTH_SERVER_Open(&g_rm_mesh_health_srv0_ctrl,
                                        &g_rm_mesh_health_srv0_cfg);

    return retval;
}
```

(2) Definition of Test ID (app_main.c)

Declare a structure in *app_main.c* that contains a function pointer to the function called by the *Health Fault Test* message, and specify it in the *Self Test* property of the *rm_mesh_health_srv* module in FSP configuration.

```
typedef enum
{
    MESH_HEALTH_TEST_ID_00 = 0x00,
    MESH_HEALTH_TEST_ID_01 = 0x01,
    MESH_HEALTH_TEST_ID_02 = 0x02,
} e_mesh_health_test_id_t;

……

rm_ble_mesh_health_server_self_test_t gs_health_server_self_tests[] =
{
    { MESH_HEALTH_TEST_ID_00, mesh_health_self_test_00 },
    { MESH_HEALTH_TEST_ID_01, mesh_health_self_test_01 },
    { MESH_HEALTH_TEST_ID_02, mesh_health_self_test_02 },
};
```

The function itself called when receiving *Health Fault Test* message as follows. In the Mesh sample program, Call *RM_MESH_HEALTH_SERVER_ReportFault* () API with health test result as argument of the API to the fault state and send *Health Fault Status* message.

```
static void mesh_health_self_test_00(UINT8 test_id, UINT16 company_id)
{
    if ((MESH_HEALTH_TEST_ID_00 == test_id) && (g_rm_ble_mesh0_cfg.default_company_id ==
company_id))
    {
        CONSOLE_OUT("[HEALTH] A Self-Test Procedure(TestID: 0x00)\n");
        mesh_model_health_server_fault_status(MESH_HEALTH_TEST_ID_00,
                                              RM_MESH_HEALTH_SERVER_FAULT_NO_FAULT);
    }
}

……

<mesh_model.c>

API_RESULT mesh_model_health_server_fault_status(UINT8 test_id, UINT8 fault_code)
{
    API_RESULT retval;

    retval = RM_MESH_HEALTH_SERVER_ReportFault
                (
                    &g_mesh_health_srv0_ctrl,
                    &(g_mesh_health_srv0_ctrl.model_handle),
                    test_id,
                    g_rm_ble_mesh0_cfg.default_company_id,
                    fault_code
                );

    return retval;
}
```

(3) Attention Timer Callback Function (app_main.c)

Implement an Attention Timer callback function performed when receiving *Health Attention Set* message. The callback function is specified in the *Callback* properties of *rm_mesh_health_srv* module in FSP configuration. In the Mesh sample program, it is implemented as *mesh_health_server_cb*.

RM_MESH_HEALTH_SERVER_SERVER_ATTENTION_START event will be notified when Attention Timer starts, and *RM_MESH_HEALTH_SERVER_SERVER_ATTENTION_RESTART* event will be notified when Attention Timer restarts. Start attention behavior such as LED blinking in the event handling.

RM_MESH_HEALTH_SERVER_SERVER_ATTENTION_STOP event will be notified when Attention Timer stops. Stop attention behavior by this event.

```
void mesh_health_server_cb(ble_mesh_model_health_callback_args_t * p_args)
{
    UINT8 attention_sec;

    switch (event_type)
    {
        case RM_MESH_HEALTH_SERVER_SERVER_ATTENTION_START:
        case RM_MESH_HEALTH_SERVER_SERVER_ATTENTION_RESTART:
            attention_sec = *event_param;
            if (0 != attention_sec)
            {
            }
            break;

        case RM_MESH_HEALTH_SERVER_SERVER_ATTENTION_STOP:
            break;
    }

    return API_SUCCESS;
}
```

3.7.2 Health Server Sequence

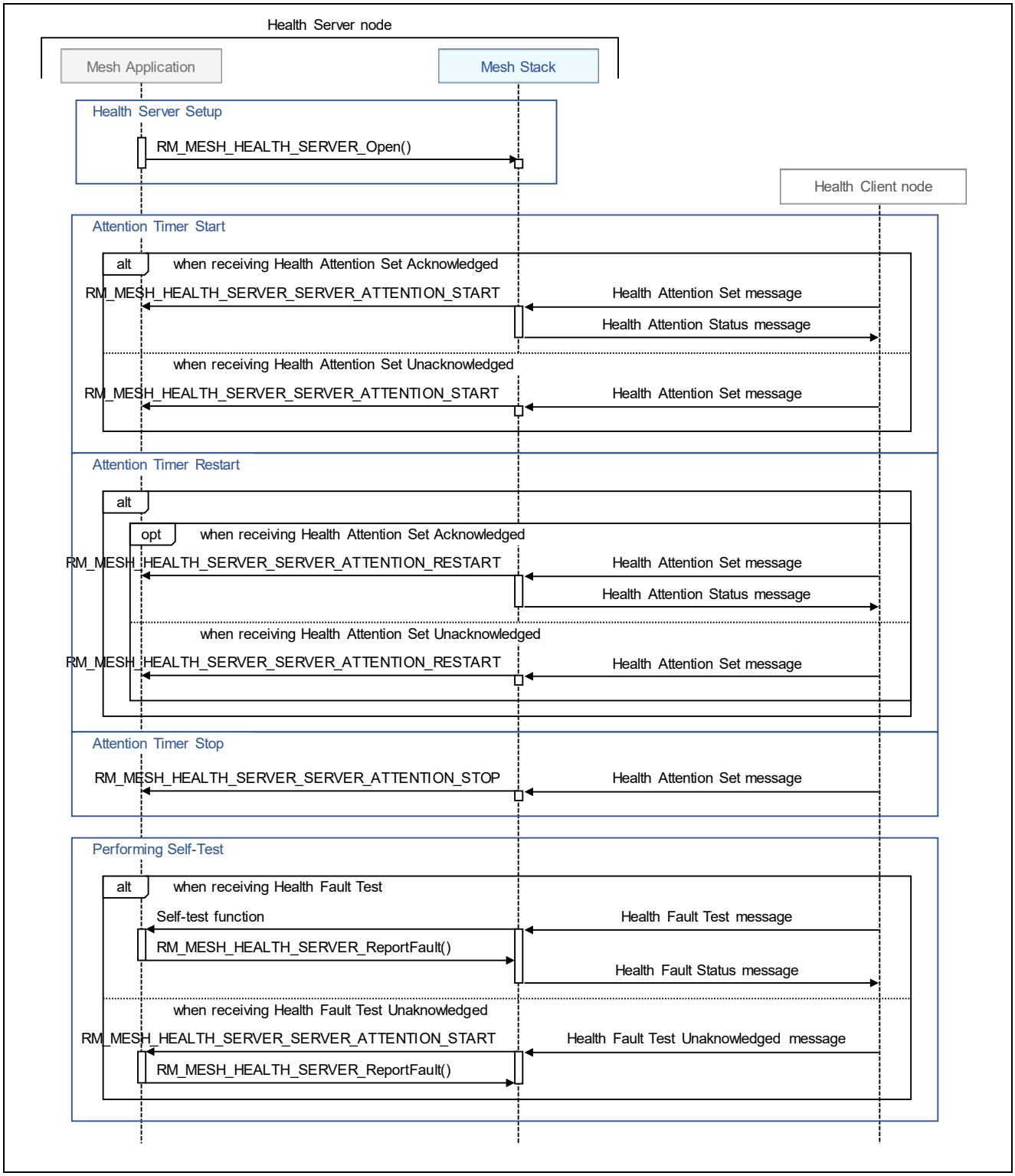


Figure 3-15 Health Server Model Operation of Mesh Sample Program

3.8 Application Model

Models that should be used by application differs depends on each application scenario. Application can use single model or multiple models. Mesh Stack provides application with API to use models defined by Bluetooth Mesh Model Specification.

This section shows how to implement Application Models while referring to the implementation of *Generic OnOff* model of Mesh Sample Program.

Mesh Sample Program works as a *Generic OnOff Client* or *Generic OnOff Server*. *Generic OnOff Client* model can change the *Generic OnOff* state of *Generic OnOff Server* model into either ON or OFF. Implementation for *Generic OnOff Server* Model of Mesh Sample Program is shown as below.

3.8.1 Server Model

(1) Instance of state (mesh_model.c)

Implement a global variable as instance of state by using the structure type for each state. In the Mesh sample program, the variable is declared in *mesh_modelc.c* as global variable. Refer to the "*Renesas Flexible Software Package User 's manual*" for structure types of states other model.

```
#ifdef ONOFF_SERVER_MODEL
static rm_mesh_generic_on_off_srv_info_t gs_onoff_state;
#endif /* ONOFF_SERVER_MODEL */
```

(2) Initialization of the state (mesh_model.c)

Initialize the state defined as the global variable in the Mesh sample program. The initialization is implemented by the *mesh_application_model_states_init* function called from the *mesh_config* function described in Section 3.2.

```
    #ifdef ONOFF_SERVER_MODEL
    memset(&gs_onoff_state, 0, sizeof(gs_onoff_state));
    #endif /* ONOFF_SERVER_MODEL */
```

(3) Registration of Server Model (mesh_model.c)

Register Server Model to register its element handle and the callback function. The callback function is specified by *Callback* property of *rm_mesh_generic_on_off_srv* module in FSP configuration. In the Mesh sample program, this initialization is implemented by the *mesh_application_model_register* function called from the *mesh_config* function described in Section 3.2.

```
    #ifdef ONOFF_SERVER_MODEL
    retval = RM_MESH_GENERIC_ON_OFF_SRV_Open(&g_ble_mesh_generic_on_off_srv0_ctrl,
                                             &g_ble_mesh_generic_on_off_srv0_cfg);
    #endif /* ONOFF_SERVER_MODEL */
```

(4) Server Model Callback function (mesh_model.c)

Implement a callback function to receive messages from Client and handle the state defined as the global variable. In the Mesh sample program, it is implemented in *mesh_model_onoff_server_cb* function.

```
void mesh_model_onoff_server_cb(ble_mesh_model_server_callback_args_t * p_args)
{
……

    retval = g_ble_mesh_access0.p_api->getElementHandleForModelHandle(&g_ble_mesh_access0_ctrl,
                                                    p_args->p_msg_context->handle, &elem_handle);

    if (API_SUCCESS == retval)
    {
        /* Check Message Type */
        switch (req_type->type)
        {
            case BLE_MESH_ACCESS_MODEL_REQ_MSG_TYPE_GET:
            {
                retval = mesh_model_onoff_server_state_get(state_params->state_type, &param);
……
            }
            break;

            case BLE_MESH_ACCESS_MODEL_REQ_MSG_TYPE_SET:
            {
                retval = mesh_model_onoff_server_state_set
                            (
                                state_params->state_type, state_params->state
                            );
……
            }
            break;

            default:
            break;
        }

        if (API_SUCCESS == retval)
        {
            retval = RM_MESH_GENERIC_ON_OFF_SRV_StateUpdate(&g_ble_mesh_generic_on_off_srv0_ctrl,
                                                                                    &state);
        }
    }

    return retval;
}
```

3.8.2 Client Model

(1) Registration of Client Model (mesh_model.c)

Register Client Model to register its element handle and the callback function. The callback function is specified by *Callback* property of *rm_mesh_generic_on_off_clt* module in FSP configuration. In the Mesh sample program, this initialization is implemented by the *mesh_application_model_register* function called from the *mesh_config* function described in Section 3.2.

```
#ifdef ONOFF_CLIENT_MODEL
    retval = RM_MESH_GENERIC_ON_OFF_CLT_Open(&g_ble_mesh_generic_on_off_clt0_ctrl,
                                                        &g_ble_mesh_generic_on_off_clt0_cfg);
#endif /* ONOFF_CLIENT_MODEL */
```

(2) Callback function to receive messages (mesh_model.c)

Implement a callback function to receive messages from Server. In the Mesh sample program, it is implemented in *mesh_model_onoff_client_cb* function.

```
#ifdef ONOFF_CLIENT_MODEL
void mesh_model_onoff_client_cb(ble_mesh_model_client_callback_args_t * p_args)
{
    API_RESULT retval = API_SUCCESS;
    mesh_generic_on_off_status_info_t status;

    switch (opcode)
    {
        case RM_BLE_MESH_ACCESS_MESSAGE_OPCODE_GENERIC_ONOFF_STATUS:
        {
            memcpy(&status, data_param, data_len);
            status.optional_fields_present =
                    (data_len > sizeof(status.present_onoff)) ? MS_TRUE : MS_FALSE;
        }
        break;
    }

    return retval;
}
#endif /* ONOFF_CLIENT_MODEL */
```

(3) Functions to transmit messages (mesh_model.c)

Implement functions to transmit message such as *GET* and *SET*. These methods are implemented as a *mesh_model_onoff_client_get*, *mesh_model_onoff_client_set*, *mesh_model_onoff_client_set_unack* function. These functions are called by pressing a switch mounted on the board or inputting a command from a terminal emulator.

```
#ifdef ONOFF_CLIENT_MODEL
API_RESULT mesh_model_onoff_client_get(void)
{
    API_RESULT retval;

    retval = RM_MESH_GENERIC_ON_OFF_CLT_Get(&g_ble_mesh_generic_on_off_clt0_ctrl);

    return retval;
}

API_RESULT mesh_model_onoff_client_set(UCHAR tid, UINT8 state)
{
    API_RESULT retval;
    mesh_generic_on_off_set_info_t  param;

    memset(&param, 0, sizeof(param));
    param.onoff = state;
    param.tid = tid;

    retval = RM_MESH_GENERIC_ON_OFF_CLT_Set(&g_ble_mesh_generic_on_off_clt0_ctrl, &param);

    return retval;
}

API_RESULT mesh_model_onoff_client_set_unack(UCHAR tid, UINT8 state)
{
    API_RESULT retval;
    mesh_generic_on_off_set_info_t  param;

    memset(&param, 0, sizeof(param));
    param.onoff = state;
    param.tid = tid;

    retval = RM_MESH_GENERIC_ON_OFF_CLT_SetUnacknowledged(&g_ble_mesh_generic_on_off_clt0_ctrl,
&param);

    return retval;
}
#endif /* ONOFF_CLIENT_MODEL */
```

3.8.3 Generic OnOff Model Sequence

Mesh Sample Program which works as a *Generic OnOff Client* node sends *Generic OnOff Set Unacknowledged* message when a switch on board is pushed. On the other hand, Mesh Sample Program which works as a *Generic OnOff Server* node turns LED on board either on or off when receiving *Generic OnOff Set* message or *Generic OnOff Set Unacknowledged* message.

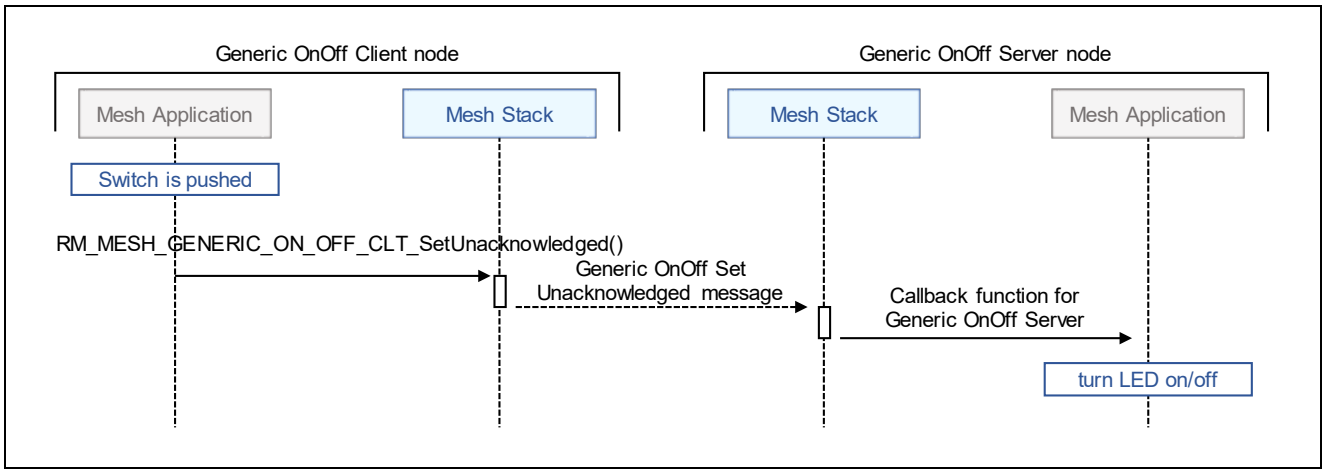


Figure 3-16 Generic OnOff Model Operation of Mesh Sample Program

3.8.4 Vendor Model Sequence

Mesh Sample Program which works as a *Vendor Server* node sends *Vendor Set Unacknowledged* message when character string is input from console. On the other hand, Mesh Sample Program which works as a *Vendor Server* output character string to console when receiving *Vendor Set* message or *Vendor Set Unacknowledged* message.

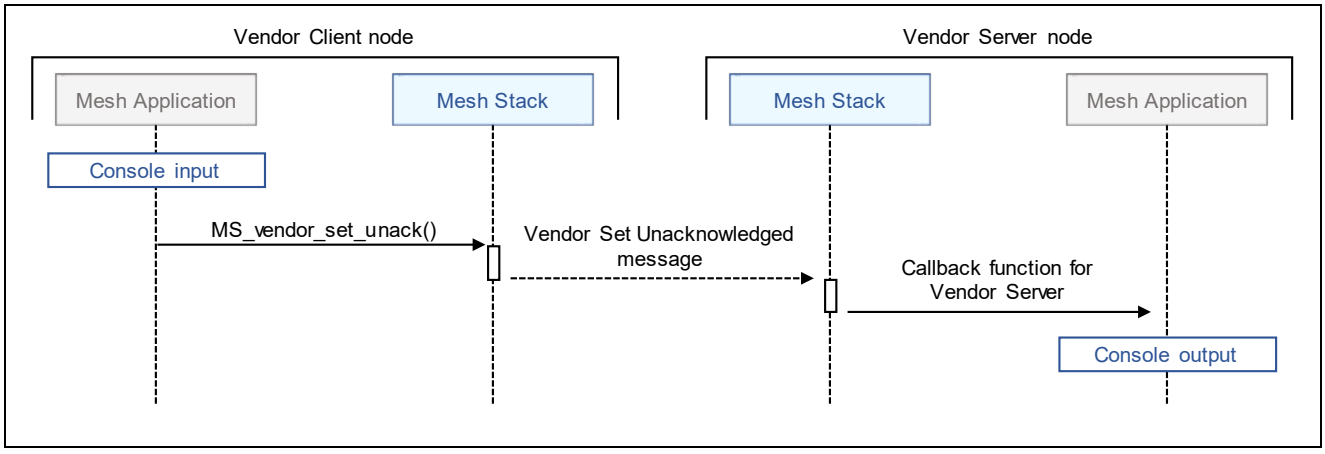


Figure 3-17 Vendor Model Operation of Mesh Sample Program

4. Appendix

4.1 Command Line Interface Program

Command Line Interface (CLI) is an interface to execute Mesh Stack API over serial interface from PC. Command Line Interface Program (mesh_cli) is included in "*RA4W1 Group Bluetooth Mesh sample application*" (R01AN5848). By using this program, you can check wireless communication operation of Mesh Stack. In addition, you can refer to the implementation of this program as an example for using Mesh Stack API. Figure 4-1 shows the example usage of Command Line Interface Program.

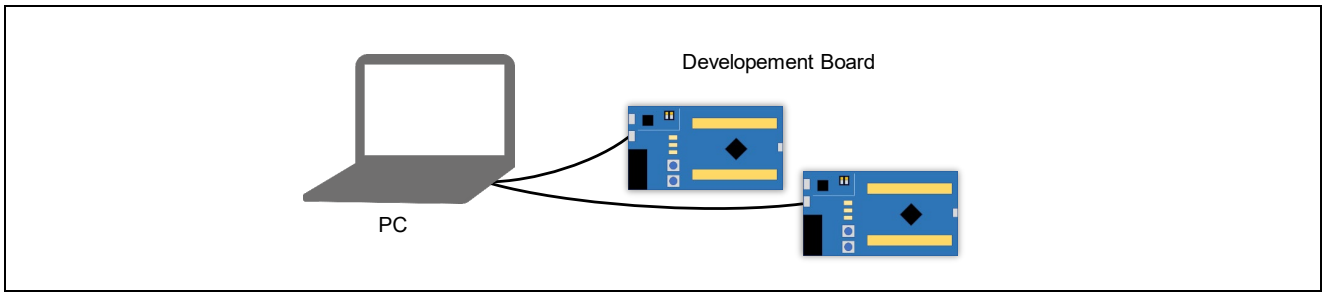


Figure 4-1 Example Usage of Command Line Interface Program

Figure 4-2 shows the example sequence of Command Line Interface. This program can work as both role such as Provisioning Client and Provisioning Server, Configuration Client and Configuration Server.

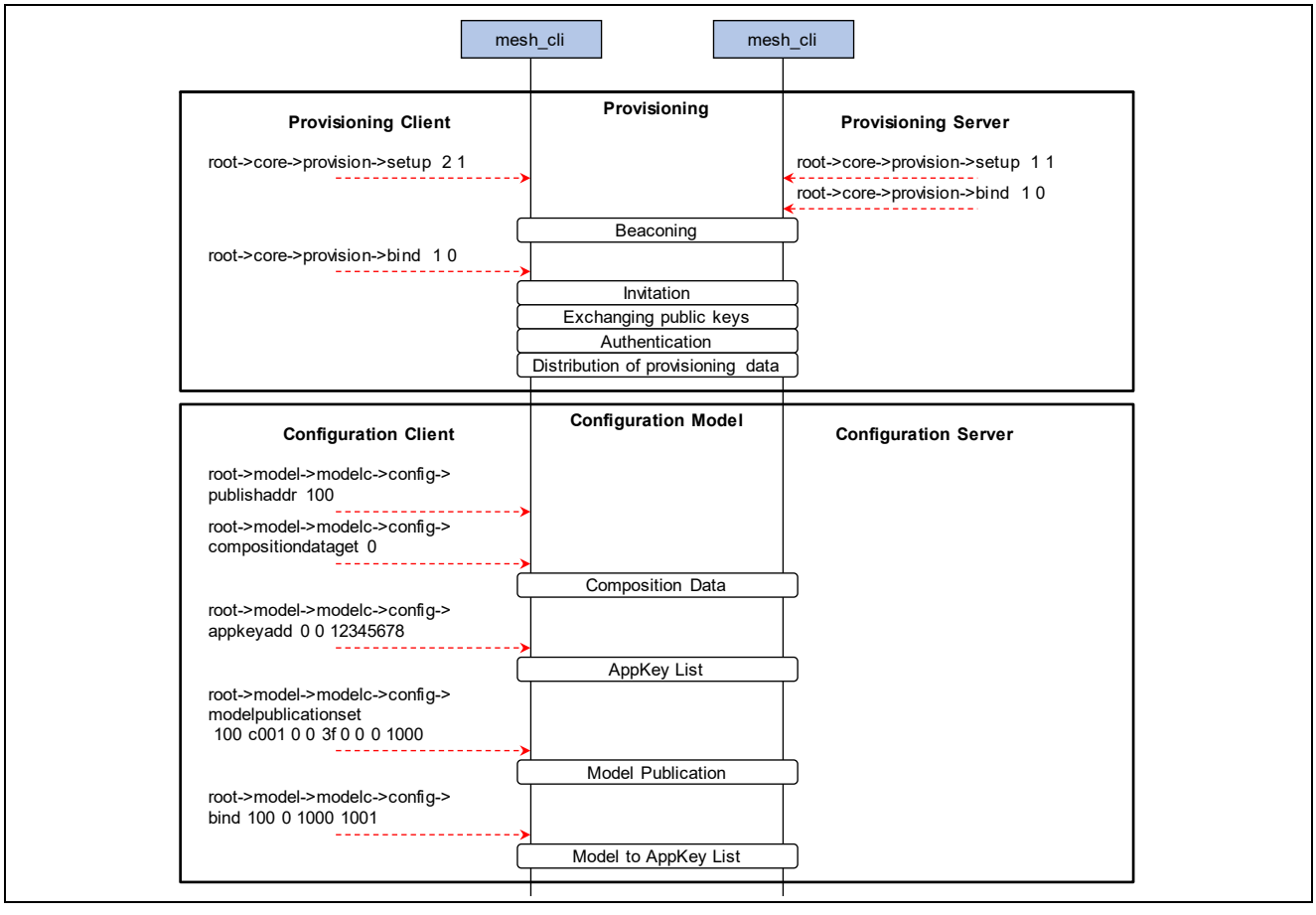


Figure 4-2 Example Sequence of Command Line Interface

Regarding the environment setup for building Command Line Interface, refer to Chapter 2 in "*RA4W1 Group Bluetooth Mesh sample application*" (R01AN5848) and use "*mesh_cli*" project generated in the workspace directory.

EK-RA4W1 has USB Serial Converter for communicating with PC. To operate Command Line Interface, use serial terminal tool on PC. (e.g. Tera Term)

Table 4-1 shows the serial port setting to communicate with Command Line Interface Program.

Table 4-1 Serial Port Setting

Item	Setting
Baud rate	115200 bps
Data	8 bits
Parity	none
Stop	1 bit
Flow Control	none

Regarding the specification of Command Line Interface, refer to "*mesh_cli_guide.pdf*" included in *"RA4W1 Group Bluetooth Mesh sample application* " (R01AN5848).

4.2 Program size

The program size required for,

- Mesh core and model module which describes in section 2.3.1 and 2.3.2.
- Mesh stack provided by Flexible software package as static library.
- Bluetooth LE stack provided by Flexible software package as static library.

Category	ROM	RAM
Mesh Core module	4.77KB	0.03KB
Mesh Model module	3.61KB	0.12KB
Bluetooth LE + Mesh stack*1 *2	304.81KB	46.56KB

*1 Includes Generic ON/OF, Configuration, Health model.

*2 Bluetooth LE stack configuration is extended.

Above table does NOT include user application, MCU peripheral which use user application, FreeRTOS kernel and BSP.

Trademark and Copyright

The *Bluetooth*® word mark and logos are registered trademarks owned by Bluetooth SIG, Inc. and any use of such marks by Renesas Electronics Corporation is under license. Other trademarks and registered trademarks are the property of their respective owners.

RA4W1 Group Bluetooth Mesh Stack uses the following open source software.

crackle; AES-CCM, AES-128bit functionality

BSD 2-Clause License

Copyright (c) 2013-2018, Mike Ryan
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Revision History

Rev.	Date	Description	
1.00	Feb. 25, 2022	-	First edition
1.01	Apr.27, 2022	-	Add section 3.3.1 item (4).
1.05	Dec. 27, 2022	P.35 P.36 P.37 P.41 P.48	Added the description of "Console String Reception Configuration". Added the description of configurations for Provisioning operation of Mesh Sample Program. Fixed GATT Bearer Connectable Advertising Interval. Added the description of "Mesh Stack Termination". Updated the description of using OOB Public Key and OOB Authentication.

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/.

© 2022 Renesas Electronics Corporation. All rights reserved.