RA4W1 Group

Bluetooth Low Energy Application Developer's Guide

Introduction

This document describes how to develop Bluetooth® Low Energy (Hereinafter referred to as "Bluetooth LE" or "BLE") applications and provides some guidance on developing Bluetooth Low Energy applications.

"BP:" in the text describes recommendations and risks based on the guideline (Bluetooth® Security and Privacy Best Practices Guide) published by the Bluetooth SIG so that implementers can select best practices for security and privacy. Please refer to it.

Target Device

RA4W1 Group

Related Documents

Bluetooth Core Specification (https://www.bluetooth.com)

Supplement of Bluetooth Core Specification (https://www.bluetooth.com)

Bluetooth® Security and Privacy Best Practices Guide (https://www.bluetooth.com)

RA4W1 Group User's Manual: Hardware (R01UH0883)

Renesas Flexible Software Package User's Manual

e^2 studio Getting Started Guide (R20UT4204)

RA4W1 Group Bluetooth LE Profile API Document User's Manual (R11UM0154)

Bluetooth Low Energy Profile Developer's Guide (R01AN6459)

Host Controller Interface Firmware(R01AN5429)

Public BD Address writing tool(R01AN5439)

Bluetooth Test Tool Suite operating instructions Application Note (R01AN4554)

RA4W1 Group Guidelines for 2.4 GHz Wireless Board Design (R01AN4886)

BLE sample application (R01AN5402)

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.

Contents

1. Overview 6
1.1 Development environment 6
1.1.1 Hardware requirements 6
1.1.2 Software requirements 6
1.2 Typical design flow 7
1.2.1 Flexible Software Package 9
1.2.2 QE for BLE 9
1.2.3 Related Tools 9
1.3 Usage of this document 10

2. BLE module 11
2.1 Supported features 11
2.2 How to add BLE module to project 13
2.3 Configuration Options 15
2.4 How to adjust configuration option 17
2.4.1 How to adjust RAM usage 17
2.4.2 How to configure BD address 19
2.4.3 How to use random address 21
2.4.4 How to configure for minimum current consumption 22

3. How to implement user code 26

4. Advertising 29
4.1 Advertising with abstraction API 30
4.1.1 Whitelist 31
4.1.2 Privacy 31
4.2 Advertising with GAP API 32
4.2.1 Set advertising parameter 32
4.2.2 Advertising Data / Scan Response Data 36
4.2.3 Start Advertising 36
4.2.4 Stop Advertising 36
4.3 Periodic Advertising with GAP API 37
4.3.1 Non-Connectable Advertising Parameter 38
4.3.2 Periodic Advertising Parameter 38
4.3.3 Periodic Advertising Data 38
4.3.4 Start Periodic Advertising 38
4.3.5 Stop Periodic Advertising 40
4.4 Advertising Data / Scan Response Data / Periodic Advertising Data 41
4.4.1 Data format 42
4.4.2 Advertising data update 44
4.4.3 Periodic Advertising Data Update 44

4.4.4 Total advertising data size 45
4.5 Typical use case for advertising 46
4.5.1 Connection with Smart Phone 46
4.5.2 Beacon 47

5. Scan 48
5.1 Scan with abstraction API 49
5.1.1 Scan filtering 49
5.1.2 Privacy 49
5.2 Scan with GAP API 50
5.2.1 Set scan parameters 50
5.2.2 Start scan 51
5.2.3 Stop scan 51
5.2.4 Received information by scan 52
5.3 Scan filtering 53
5.3.1 Whitelist 53
5.3.2 Duplicate advertising filtering 53
5.3.3 Discoverable mode filtering 54
5.3.4 Advertising data filtering 54
5.4 Periodic advertising synchronization with GAP API 55
5.4.1 Start scan 56
5.4.2 Detect periodic advertiser 56
5.4.3 Establish periodic advertising synchronization 56
5.4.4 Periodic advertiser list 57
5.4.5 Receive periodic advertising PDUs 57
5.4.6 Lost periodic advertising sync 57
5.4.7 Terminate periodic advertising sync 57

6. Connection 58
6.1 Requesting connection with abstraction API 59
6.1.1 Whitelist filtering 59
6.1.2 Privacy 59
6.2 Requesting connection with GAP API 60
6.3 Cancelling Connection Request 60
6.4 Whitelist filtering 60
6.5 Privacy 61
6.6 Multiple connection 62
6.6.1 Connecting to multiple peripheral devices 63
6.6.2 Connection to multiple central devices 68
6.6.3 Multi role connection 72
6.7 Disconnection 77

7. Communication 78
7.1 Changing PHY 78
7.2 Changing maximum transmission packet length 81
7.3 Updating connection parameter 83
7.4 Changing MTU 88
7.5 Flow control 90
7.6 High throughput communication 91

8. Security 92
8.1 Pairing 92
8.1.1 Pairing Parameters 94
8.1.2 Key generation and registration 98
8.1.3 OOB (Out of Band) data transmission and reception 99
8.1.4 Pairing request 99
8.1.5 Response to pairing request 99
8.1.6 Carrying out pairing method 100
8.1.7 Key exchange 102
8.1.8 Completion of pairing 102
8.2 Bonding 103
8.2.1 Store remote device keys 104
8.2.2 Store local device keys 108
8.2.3 Reset the stored keys 108
8.2.4 Delete the stored keys 108
8.2.5 Filtering remote devices after bonding 109
8.3 Encryption 110
8.3.1 Request Encryption 110
8.3.2 Respond to an encryption request 112
8.4 Privacy 116
8.4.1 Privacy with abstraction API 118
8.4.2 Privacy with GAP API 120
8.4.3 Resolve remote device RPA with abstraction API 124
8.4.4 Resolve remote device RPA with GAP API 127

9. Profile and service 130
9.1 Standard profile and Standard Service 131
9.2 APIs of GATT Procedure 137
9.2.1 Read operation 137
9.2.2 Write operation 138
9.2.3 WriteWithoutResponse operation 139
9.2.4 Notification operation 140
9.2.5 Indication operation 141
9.2.6 ReliableWrite operation 143

9.2.7 Broadcast Operation 145
9.3 Example of using GATT Procedure 146

A. Appendix : Sample applications 147
A.1 Beacon sample 150
A.1.1 Remote devices 150
A.1.2 Operations 150
A.1.3 Advertising Data 150
A.1.4 Configuration option 152
A.1.5 Configurable parameters 152
A.1.6 Command 152
A.2 Peripheral sample 153
A.2.1 Remote devices 153
A.2.2 Operations 153
A.2.3 Configuration option 155
A.2.4 Configurable parameters 155
A.3 Central sample 156
A.3.1 Remote devices 156
A.3.2 Operations 156
A.3.3 Configuration option 156
A.3.4 Configurable parameters 157
A.4 Multi-role sample 158
A.4.1 Topology 158
A.4.2 Remote devices 159
A.4.3 Operations 159
A.4.4 Configuration option 162
A.4.5 Configurable parameters 163

Revision History 164

1. Overview

This document describes how to develop Bluetooth Low Energy applications and provides some guidance on developing Bluetooth Low Energy applications.

1.1 Development environment

This section describes the environment for BLE application development.

1.1.1 Hardware requirements

Table 1 shows the hardware requirements for building and debugging the BLE application.

Table 1. Hardware requirements

Hardware	Description
Host PC	Windows® PC with USB interface.
MCU board	The board with RA4W1 or EK-RA4W1[RTK7EKA4W1S00000BJ] Note: This document uses EK-RA4W1 for explanation.
On-chip debugging emulators	The EK-RA4W1 has an on-board debugger (J-Link OB), therefore it is not necessary to prepare an emulator.
E2-Lite emulator	Needed if user want to write device-specific data in user's custom board by using Renesas Flash Programmer.
USB cables	Used to connect the PC to the MCU board. EK-RA4W1: 2 USB A ⇿ micro-B cable

1.1.2 Software requirements

Table 2 shows the software requirements for building and debugging the BLE application.

Table 2. Software requirements

Software		Version	Description
GCC environment	e² studio	2021-07 or later	Integrated development environment (IDE) for Renesas devices.
	GCC ARM Embedded	V9 or later	C/C++ Compiler. (Download from e² studio installer)
	Renesas Flexible Software Package (FSP)	V3.3.0 or later	Software package for making applications for the RA microcontroller series.
	QE for BLE[RA] QE for BLE[RA,RE,RX]	V1.2.0 or later V1.4.0 or later	Generates the source codes (BLE base skeleton program) as a base for the BLE Application and the BLE Profile. Refer to *R20UT4951EJ* about how to download QE for BLE. Note: QE for BLE[RA] has been integrated into QE for BLE[RA,RE,RX].
	QE utility [RA] QE utility [RA,RE,RX]	V1.2.1 or later V1.4.0 or later	
	SEGGER J-Flash	V6.86	Tool for programming the on-chip flash memory of microcontrollers.
Header files			All API calls and their supporting interface definitions located in r_ble_api.h and rm_ble_abs_api.h.
Integer types			It uses ANSI C99 "Exact width integer types". These types are defined in stdint.h.
Endian			Little endian

1.2 Typical design flow

Bluetooth SIG defines specifications of application profiles for typical use cases in BLE applications. Users can interconnect with existing BLE device by using such application profiles. On the other hand, it is necessary for newly designed application profiles as well as user applications where the user wants to perform a new BLE bidirectional communication. The user needs to design the following items when they make a new BLE bidirectional communication.

- The structure of application data exchanged between GATT server and clients.
- The method of accessing the GATT database.
- The setting of GAP communication parameters.
- The method of connecting devices.
- The setting of security.

BP: Support for authentication and encryption is recommended when there are modifiable GATT characteristics (e.g. a door lock mechanism where the remote device manipulates the lock state by changing the value of the attribute), including custom profile.

Renesas provides some tools for BLE application development. Users can design BLE applications by using these tools. Typical BLE application design flow and related Renesas provided tools in each step are shown in Figure 1.

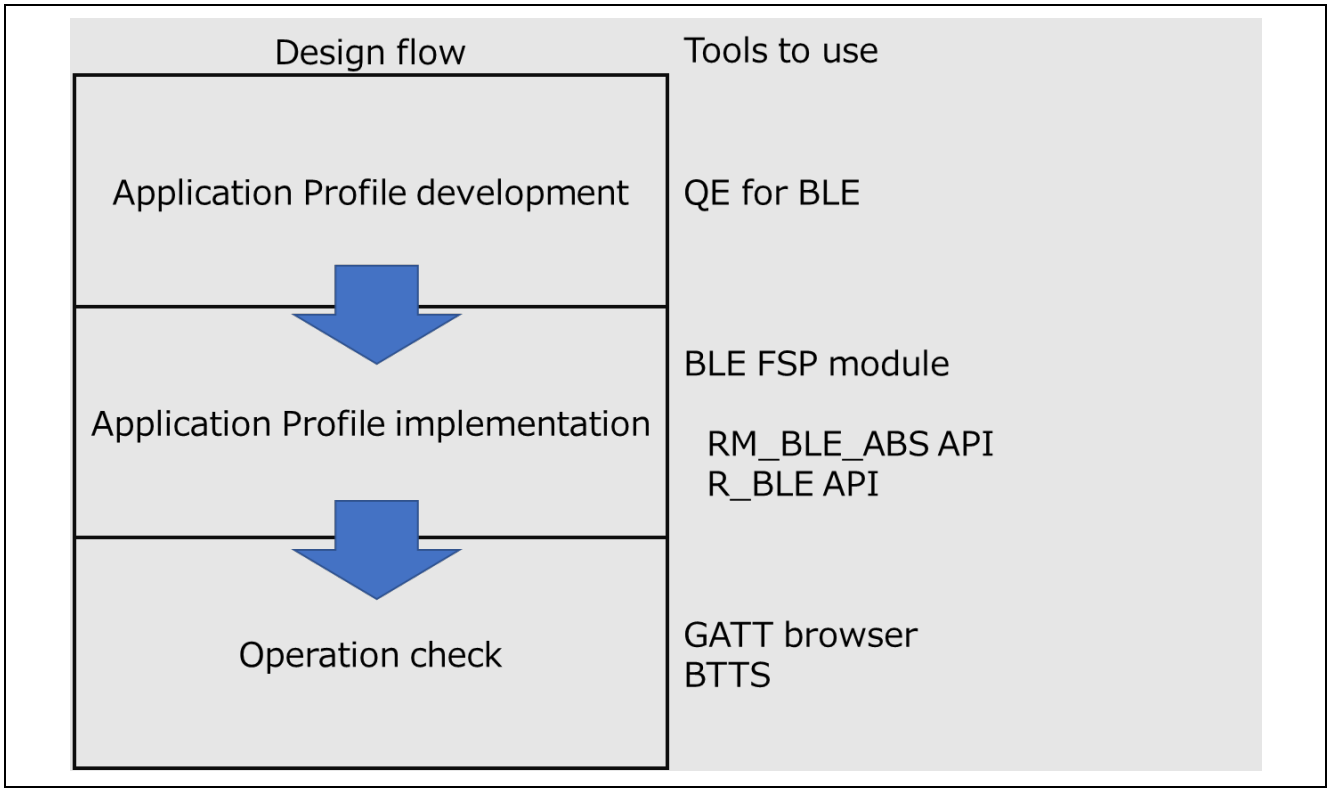


Figure 1. Bluetooth LE application development procedure and auxiliary tools

Figure 2 shows a typical software structure generated by the Renesas provided tools.

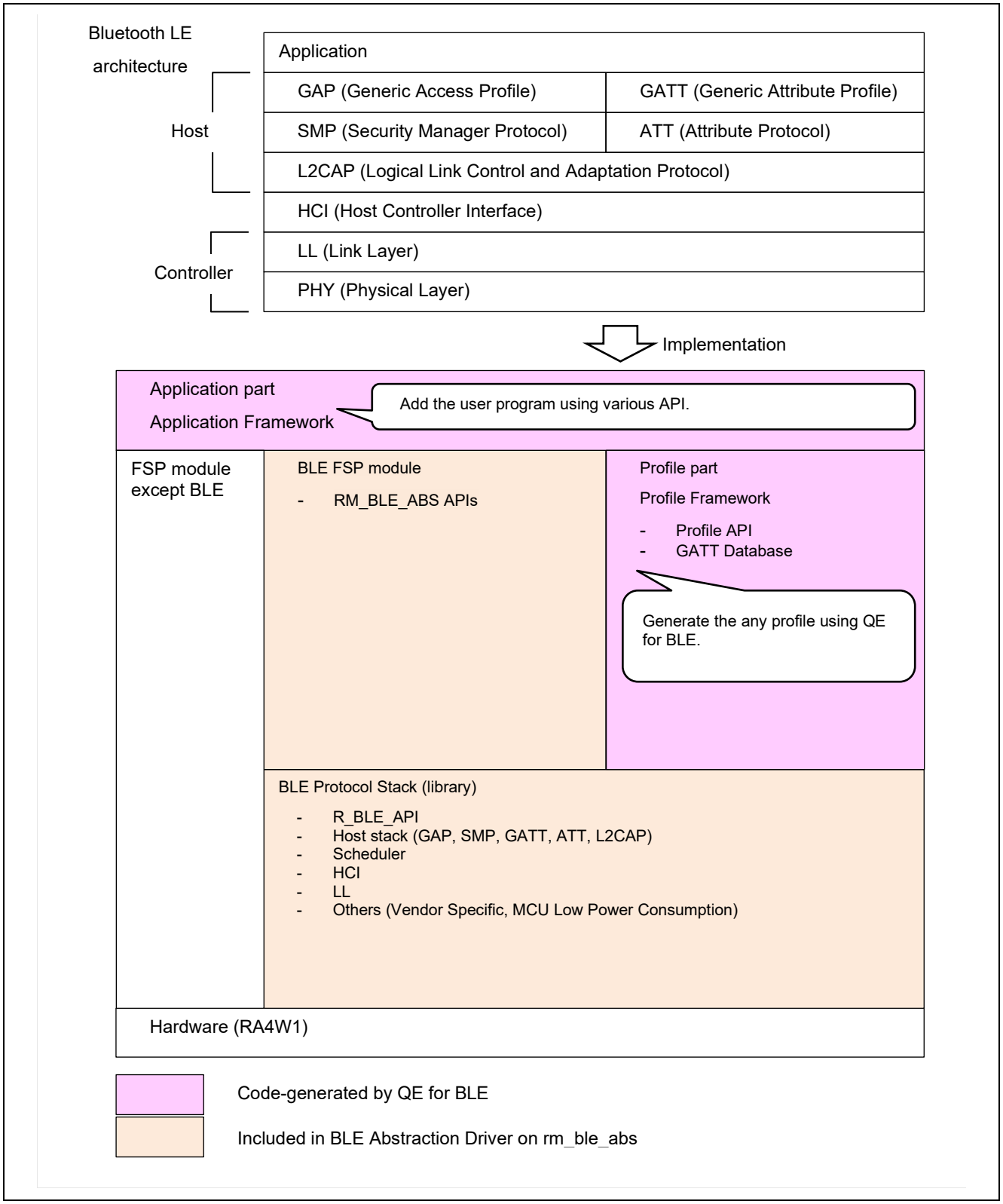


Figure 2. Software structure

1.2.1 Flexible Software Package

BLE Protocol Stack is included in *BLE Abstraction Driver on rm_ble_abs* (here after referred to as “BLE module”) which is a part of the Flexible Software Package. The driver provides the BLE features that comply with the Bluetooth Core Specification version 5.0 defined by Bluetooth SIG. Users can add the driver to their own project from the FSP configuration in e^2 studio and start BLE application development. The BLE features are provided in static library format as a BLE Protocol Stack. The BLE Protocol Stack controls the BLE procedures (e.g., Advertising, Scanning, Initiating, Connection) and manages execution of RF events. Refer to *BLE sample application (R01AN5402)* about adding BLE module to user’s project.

1.2.2 QE for BLE

QE for BLE is tool for designing BLE application profiles. The QE for BLE tool can generate profile and BLE application skeleton code. The user can modify QE for BLE generated codes according to the use case. Refer to *Bluetooth Low Energy Profile Developer's Guide(R01AN6459)* about usage of QE for BLE tool.

1.2.3 Related Tools

Renesas provides tools shown in Table 3 to assist BLE application development.

Table 3. Supporting tools for application development

Tool	Description
GATT Browser	Smartphone application used for accessing to GATT Server. Users can confirm the Bluetooth Low Energy primitive communication and GATT database structure on GATT server and so on from smartphone which installed this application. This application can be downloaded from, Android: https://play.google.com/store/apps/details?id=com.renesas.ble.gattbrowser iOS: https://itunes.apple.com/us/app/gattbrowser/id1163057977?mt=8
Bluetooth Test tool Suite (BTTS)	Tool suite to control RA4W1 connected with Windows PC via USB Serial and evaluate three functions of RF, Beacon and Data Communication in Bluetooth Core Specification 5.0. It can be also used when getting Radio Law certification for the device. Refer to *Bluetooth Test Tool Suite operating instructions Application Note (R01AN4554)*.

1.3 Usage of this document

One of the typical BLE applications on an RA4W1 is accepting connections from PCs, smartphones, etc., and operate as a GATT server or client. In each case, refer to the chapters shown in Table 4.

Table 4. Typical BLE application and referenced chapter

Application	Process	Description
GATT server	Advertising	Refer to Chapter 4.
	Connection	When receiving a connection request from Central, BLE Protocol Stack automatically establishes a connection and notifies *BLE_GAP_EVENT_CONN_IND* event to user application.
	Pairing	Refer to chapter 8.
	Data communication (Notification)	Refer to chapter 7.
GATT client	Scan	Refer to chapter 5.
	Connection	Refer to chapter 6.
	Pairing	Refer to chapter 8.
	Data Communication (Read, Write)	Refer to chapter 7.

Other examples of BLE applications on RA4W1 are shown below.

- GATT Server application that collects operation logs of industrial equipment and sensor data of healthcare equipment and uploads them to clients such as PCs and smartphones.
  - Refer to section 2.4.4, section 6.6 and chapter 8.

- GATT Server application that transfers the data downloaded from clients such as PCs and smartphones.
  - Refer to section 7.6 and chapter 8.

- GATT Server application that uploads data (e.g., image data, audio data, etc.) to clients such as PCs and smartphones.
  - Refer to section 7.6.

- GATT Server applications for electronic locks, OA devices, consumer devices, etc. that are operated from multiple clients such as smartphones.
  - Refer to section 6.6 and chapter 8.

- Beacon application that periodically broadcasts data such as sensor data.
  - Refer to section 4.5.2.

2. BLE module

BLE module is a part of the Flexible Software Package. The driver provides BLE features that comply with the Bluetooth Core Specification version 5.0 defined by Bluetooth SIG. BLE module provides the APIs (*RM_BLE_ABS_xxx*) to use these features.

2.1 Supported features

Table 5 shows BLE module supported features.

Table 5. BLE features

Bluetooth Version	LE features and description	Remark
5.0	LE 2M PHY (2 Msym/s PHY for LE) 2Mbps PHY data rate.	High data throughput. Low power consumption by short communication time.
5.0	LE Coded PHY (LE Long Range) 500kbps/125kbps PHY data rate.	Extend communication distance.
5.0	LE Advertising Extensions Enable Advertising by secondary channel. (Up to 4 independent Advertising can be executed simultaneously in RA4W1.) Expansion of Advertising Data/Scan Response Data size up from 31 bytes to 1650 bytes. Advertising by Long Range. Periodic Advertising is possible.	Wireless interference reduction. Beacon information expansion. Establishing connection in long-distance. Utilization of secondary channel.
5.0	LE Channel Selection Algorithm #2 Improving the channel hopping algorithm.	Wireless interference reduction.
5.0	High Duty Cycle Non-Connectable Advertising Shorten minimum Advertising Interval (100ms→20ms).	Shortening the time to connect. Higher frequency of beacon transmission.
4.2	LE Data Packet Length Extension Expand the data communication packet size (27 bytes→251 bytes).	High data throughput. Low power consumption by short communication time.
4.2	LE Secure Connections Support the pairing with the Elliptic curve Diffie-Hellman (ECDH) key exchange for passive eavesdropping protection.	Enhanced security.
4.2	Link Layer Privacy Link Layer supports address resolution of Privacy feature.	Faster address resolution.
4.2	Link Layer Extended Scanner Filter Policies	
4.1	Low Duty Cycle Directed Advertising Support Low Duty Cycle Advertising for reconnection with known devices.	
4.1	32-bit UUID Support in LE Support 32-bit UUID (extended to 128-bit when used by GATT).	
4.1	LE L2CAP Connection-Oriented Channel Support Support the communication using L2CAP credit based flow control channel.	
4.1	LE Privacy v1.1 Avoid the tracking from other LE devices by changing the BD Address periodically.	Enhanced security.
4.1	LE Link Layer Topology Support both Central and Peripheral roles and can operate as Central when connecting to one remote device and as Peripheral when connecting to another remote device.	Enhanced topology.
4.1	LE Ping Checks whether connection is maintained by a packet transmission request including MIC field after connection encryption.	

Bluetooth Version	LE features and description	Remark
Addendum 2	Appearance Data Type Appearance characteristic can be used in GAP service.	
4.0	Bluetooth Low Energy - Low Energy Controller Low Energy Physical Layer (PHY) Low Energy Link Layer (LL) - Low Energy Host Enhancements to L2CAP for Low Energy Security Manager (SM) - Enhancements to HCI for Low Energy - Low Energy Direct Test Mode - AES Encryption - Enhancements to GAP for Low Energy - Attribute Protocol (ATT) - Generic Attribute profile (GATT)	 Low Energy Controller is mandatory feature. Low Energy Host is mandatory feature. ATT is mandatory feature. GATT is mandatory feature.

Note: BR/EDR (Basic Rate/Enhanced Data Rate) is not supported.

Note: The feature except mandatory feature is optional (vendor dependent). Therefore, some device (e.g., smart phone may not support such an optional feature.

2.2 How to add BLE module to project

Refer to *BLE sample application (R01AN5402)* Chapter 3 and 4. Users can use the following categories of API to make their own application.

- Abstraction API (RM_BLE_ABS_XXX API)

  Users can use various Bluetooth LE procedures with a single API call. However, detailed parameter settings are not possible. To use this category of API, it is necessary to call *RM_BLE_ABS_Open* API for initializing BLE module.

- R_BLE API (R_BLE_XXX API)

  Users can use various Bluetooth LE procedure by combining several APIs. However, detailed parameter settings are possible. To use this category of API, it is necessary to call *R_BLE_Open* API for initializing BLE module.

QE for BLE generates initialization function (*ble_init()*) as the following. Modify the function as following when users want to make their own application without the abstraction API.

```
ble_status_t ble_init(void)
{
    ble_status_t status;
    fsp_err_t err;

    /* Initialize BLE */
#if 0
    err = RM_BLE_ABS_Open(&g_ble_abs0_ctrl, &g_ble_abs0_cfg);
    if (FSP_SUCCESS != err)
    {
        return err;
    }
#else
    err = R_BLE_Open();
    if (FSP_SUCCESS != err)
    {
        return err;
    }

    /* Initialize GAP layer */
    err = R_BLE_GAP_Init(gap_cb);
    if (FSP_SUCCESS != err)
    {
        return err;
    }

    /* Initialize GATT Server */
    err =  R_BLE_GATTS_Init(1);
    if (FSP_SUCCESS != err)
    {
        return err;
    }

    err = R_BLE_GATTS_RegisterCb(gatts_cb, 1);
    if (FSP_SUCCESS != err)
    {
        return err;
    }

    /* Initialize GATT Client */
    err =  R_BLE_GATTC_Init(1);
    if (FSP_SUCCESS != err)
    {
        return err;
    }
```

Omit RM_BLE_ABS_Open when users want to make their own application without abstraction API.

Add initialization by using R_BLE API.

```
    err = R_BLE_GATTC_RegisterCb(gattc_cb, 1);
    if (FSP_SUCCESS != err)
    {
        return err;
    }

    /* Initialize Vender specific */
    err = R_BLE_VS_Init(vs_cb);
    if (FSP_SUCCESS != err)
    {
        return err;
    }

#endif

/* some codes omitted*/
}
```

Code 1. Initialize function modification when do not use Abstraction API

2.3 Configuration Options

BLE module has some configuration options. These options can be modified in the properties of the FSP configuration, as shown in Figure 3.

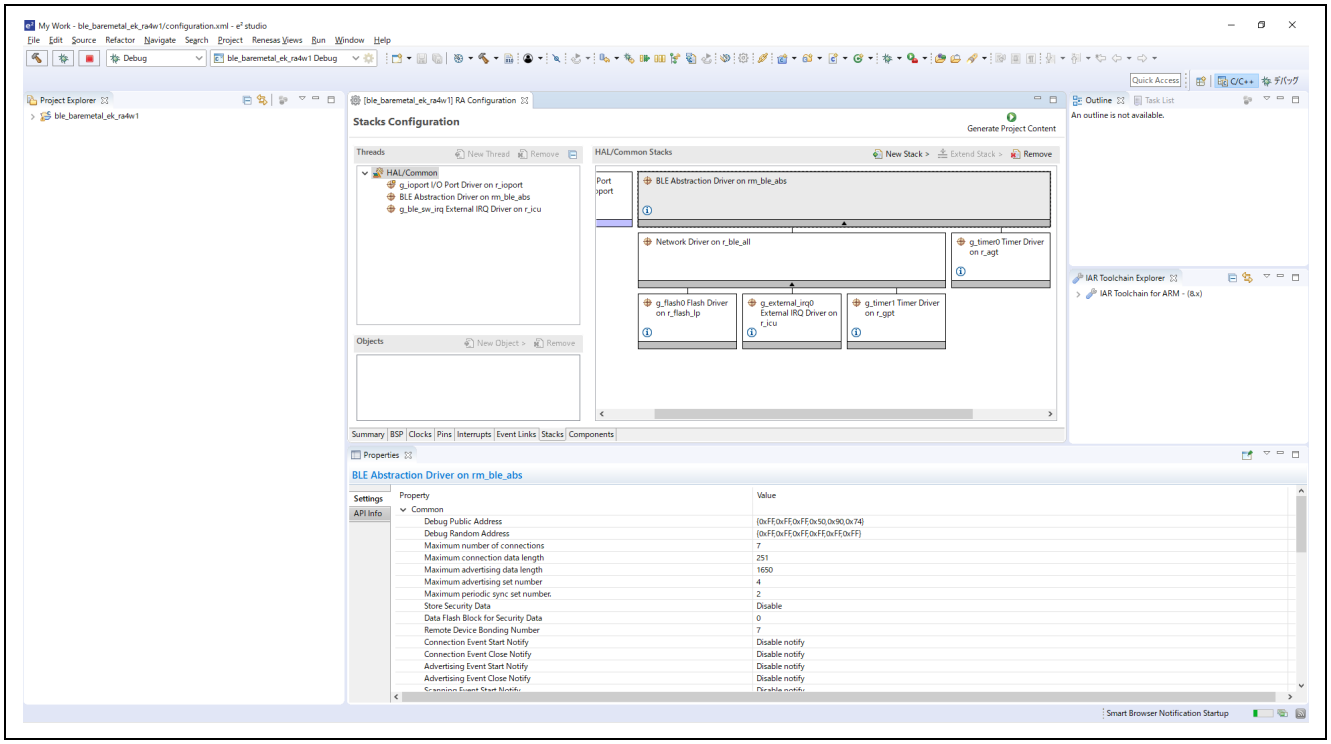


Figure 3. Common options

BLE module can select three configurations extended / balance / compact, which differ in the number of supported features. These configurations consume different ROM/RAM size. Refer to section 2.4 about consume ROM/RAM size in each configuration.

Table 6. Features supported by each type of BLE module

BLE Features	Library type		
	Extended	Balance	Compact
GAP role	Central, Peripheral, Observer, Broadcaster	Central, Peripheral, Observer, Broadcaster	Peripheral, Broadcaster
GATT role	Server, Client	Server, Client	Server, Client
LE 2M PHY	Yes	Yes	No
LE Coded PHY	Yes	Yes	No
LE Advertising Extensions	Yes	No	No
LE Channel Selection Algorithm #2	Yes	Yes	No
High Duty Cycle Non-Connectable Advertising	Yes	Yes	Yes
LE Secure Connections	Yes	Yes	Yes
Link Layer privacy	Yes	Yes	Yes

BLE Features	Library type		
	Extended	Balance	Compact
Link Layer Extended Scanner Filter policies	Yes	Yes	No
LE Data Packet Length Extension	Yes	Yes	Yes
LE L2CAP Connection Oriented Channel Support	Yes	No	No
Low Duty Cycle Directed Advertising	Yes	Yes	Yes
LE Link Layer Topology	Yes	Yes	No
LE Ping	Yes	Yes	Yes
32-bit UUID support in LE	Yes	Yes	Yes

Refer to *BLE sample application (R01AN5402)* Chapter 4 about other configuration option.

2.4 How to adjust configuration option

This section describes how to adjust configuration options in some scenarios.

2.4.1 How to adjust RAM usage

This section describes how to adjust RAM usage by changing the BLE module configuration options. BLE module (includes BLE Protocol Stack library, related peripherals, NOT include BLE application and profiles) consumes the ROM/RAM size shown in Table 7 according to extended / balance / compact configuration. Refer to *BLE sample application (R01AN5402) Chapter 1* about extended, balance and compact configuration.

Table 7. BLE module Driver ROM/RAM usage

Configuration	ROM [KB]	RAM[KB]
Extended	216	44
Balance	170	30
Compact	145	29

Consumption of RAM can be reduced by changing following configuration options. Table 8 shows the relationship between RAM consumption and related configuration options.

Table 8. Dependency of RAM size vs. configuration option

Configuration option	Setting range (default)	RAM size
Maximum number of connections	1 – 7 (7)	Require 1 [KB] per one connection.
Maximum connection data length	27 – 251 (251)	Require 0.5 [KB] per 64 [bytes] of connection data.
Maximum advertising data length	31 – 1650 (1650)	See Table 9.
Maximum advertising set number*1	1 - 4 (4)	See Table 9.
Maximum periodic sync set number*2	1 – 2 (2)	Require 64 [bytes] per one sync.

*1: Number of advertising sets that can be broadcasted simultaneously.

*2: Number of sets that can be synchronize with periodic advertising.

Table 9 shows relationship between RAM consumption, the *Maximum advertising data length* configuration option, and the *Maximum advertising set number* configuration option.

Table 9. Dependency of RAM size vs. Max. advertising data length and Max. advertising set number

Maximum advertising set number									
	1	Maximum advertising data length	0-252	253-504	505-756	757-1008	1009-1260	1261-1512	1513-1650
		Required additional RAM size [bytes] based on hatched cell	0	512	1024	1536	2048	2560	3072
	2	Maximum advertising data length	0-252	253-504	505-756	757-1008	1009-1260	1261-1512	1513-1650
		Required additional RAM size [bytes] based on hatched cell	0	1024	2048	3072	4096	5120	6144
	3	Maximum advertising data length	0-252	253-504	505-756	757-1008	1009-1260	1261-1650	
		Required additional RAM size [bytes] based on hatched cell	0	1536	3072	4608	6144	7680	
	4	Maximum advertising data length	0-252	253-504	505-756	757-1008	1009-1650		
		Required additional RAM size [bytes] based on hatched cell	0	2048	4096	6144	7168		

2.4.2 How to configure BD address

Bluetooth SIG defines Bluetooth Device address (BD address), as shown in Table 10.

Table 10. BD address types

BD address type			Description
Public device address			Unique 48bit Bluetooth device address. Note: Refer to Bluetooth Core Specification Vol 2, Part B, " 1.2 Bluetooth Device Addressing". It can be obtained in the same way as the MAC address.
Random device address	Static address		Random address where MSB starts with 11b, and the remaining bits can be set randomly to be used. **<Example>** Cx:xx:xx:xx:xx:xx, Dx:xx:xx:xx:xx:xx, Ex:xx:xx:xx:xx:xx, Fx:xx:xx:xx:xx:xx Note: Refer to *Bluetooth Core Specification Vol 6, Part B, "1.3.2 Random Device Address".* Note: BLE Protocol Stack does not check address format. Note: A static address consists of random numbers. The possibility of duplicate values with other devices is not zero.
	Private address	Non-resolvable private address	Random address where MSB starts with 00b, and the remaining bits can be dynamically regenerated. **<Example>** 0x:xx:xx:xx:xx:xx, 1x:xx:xx:xx:xx:xx, 2x:xx:xx:xx:xx:xx, 3x:xx:xx:xx:xx:xx
		Resolvable Private Address (RPA)	Random address where MSB starts with 01b, and the remaining bits can be dynamically regenerated and enhanced with privacy feature. **<Example>** 4x:xx:xx:xx:xx:xx, 5x:xx:xx:xx:xx:xx, 6x:xx:xx:xx:xx:xx, 7x:xx:xx:xx:xx:xx

BLE Protocol Stack adopts BD address from following area.

1. Data flash specified block
2. Code flash specified block
3. Firmware initial value

Related configurations are shown in Table 11. Refer to *BLE sample application (R01AN5402)* chapter 4 about details of these configurations.

Table 11. BD address configurations

Configuration option	Initial value
Debug Public Address	74:90:50:FF:FF:FF (Firmware initial value of Public address)
Debug Random Address	FF:FF:FF:FF:FF:FF (Firmware initial value of Random address)
Device Specific Data Flash Block	-1 (Data flash area is not used for BD address. Change the value from 0 to 7 when you want to store device specific data in data flash block.)
Code Flash (ROM) Device Data Block	255 (Code block 255 is used)

BLE module adopts BD address by R_BLE_Open at application startup according to the priority shown in Figure 4.

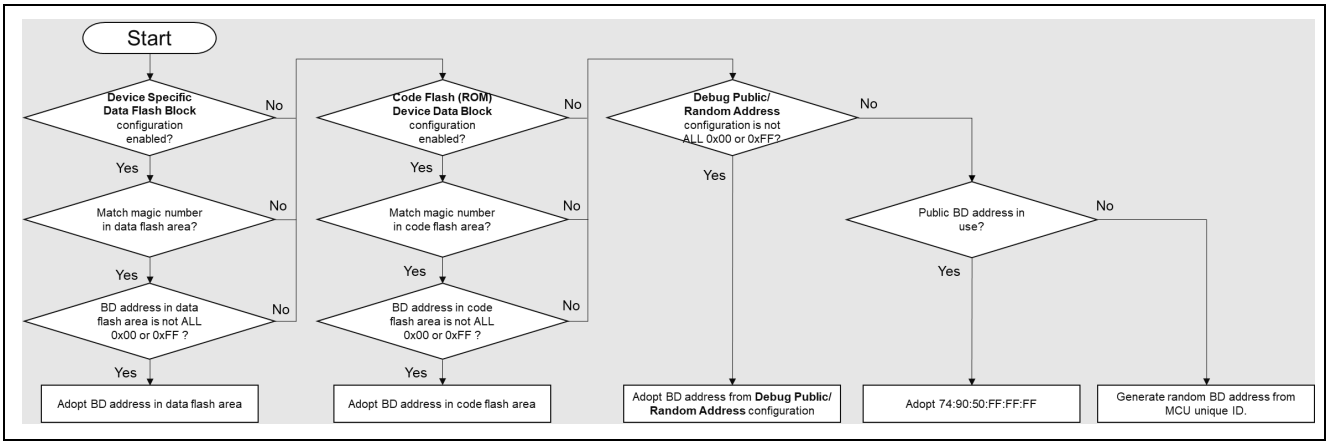


Figure 4. BD address adoption flow of BLE module

Following items describe how to modify BD addresses which are stored in data flash area, code flash area and RAM area.

(1) How to modify BD address which stored in data flash area

Use *R_BLE_VS_SetBdAddr()* API to write BD address to data flash area. After writing BD address, RA4W1 must be reboot at once to adopt the BD address. Refer to *Renesas Flexible Software Package User's Manual* for details of the API.

(2) How to modify BD address which stored in code flash area

To write BD address to code flash area, use the Renesas E2-Lite emulator and the Renesas Flash Programmer (RFP) unique code function. Refer to *BLE sample application (R01AN5402)* chapter 4 about detail procedures.

(3) Other method

When the user wants to dynamically change BD address, *R_BLE_VS_SetBdAddr* API can be used. Refer to *Renesas Flexible Software Package User's manual* for details of the API.

Note: Devices are identified using BD address, but you can additionally use Local Name of Advertising Data and Device Name of GAP service. Local Name is shown in "Table 22". Device Name is shown in "Table 48".

2.4.3 How to use random address

Code 2 is a sample code for advertising with a random address adopted by R_BLE_Open.

```
ble_abs_legacy_advertising_parameter_t g_ble_advertising_parameter =
{
……
.own_bluetooth_address_type = BLE_GAP_ADDR_RAND,
……
};

……

void gap_cb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
    switch(type)
    {
        case BLE_GAP_EVENT_STACK_ON:
        {
            /* Get BD address for Advertising */
            R_BLE_VS_GetBdAddr(BLE_VS_ADDR_AREA_REG, BLE_GAP_ADDR_RAND);
        } break;
……
}

……

void vs_cb(uint16_t type, ble_status_t result, st_ble_vs_evt_data_t *p_data)
{
……
    switch(type)
    {
        case BLE_VS_EVENT_GET_ADDR_COMP:
        {
            /* Start advertising when BD address is ready */
            st_ble_vs_get_bd_addr_comp_evt_t * get_address
             = (st_ble_vs_get_bd_addr_comp_evt_t *)p_data->p_param;

            memcpy(g_ble_advertising_parameter.own_bluetooth_address, get_address->addr.addr,
                   BLE_BD_ADDR_LEN);

            RM_BLE_ABS_StartLegacyAdvertising(&g_ble_abs0_ctrl, &g_ble_advertising_parameter);
        } break;
……
}
```

Set advertising parameter to use random address

Get random address

After this API call, *BLE_VS_EVENT_GET_ADDR_COMP* event will happen and *vs_cb()* will execute.

Copy random address to advertising parameter and start advertising.

Code 2. Sample of using random address

2.4.4 How to configure for minimum current consumption

The configurations shown in Table 12 make the current consumption minimize.

Table 12. Configurations for minimum current consumption

Configuration options		Comments
FSP configuration / Clocks	Clock source: HOCO HOCO clock frequency: 32MHz	Note: Disable non-used clocks or set minimum clock frequency to minimize current consumption. Note: R_BLE_Open optimizes the operation of R_BLE_Execute according to the clock frequency. If you change the clock frequency dynamically, call R_BLE_Open again after R_BLE_Close.
	ICLK: 32MHz	
	PCLKA and PCLKD: 32MHz	
	FCLK: 32MHz	
	CLKOUT: Disable	
BLE module properties	DC-DC converter: Enable DC-DC converter	Note: refer to *RA4W1 Group Guidelines for 2.4 GHz Wireless Board Design (R01AN4886)*.
	RF_DEEP_SLEEP Transition: Enable	Refer to *BLE sample application (R01AN5402)* chapter 4 about details of configuration options. Note: The transmit current can be reduced by lowering the RF transmit power, but the communication range will be shortened accordingly
	CLKOUT_RF Output: No Output	
	Transmission Power Maximum Value	
	Transmission Power Default Value	
MCU Low Power	Addition of *Low Power Modes Driver on r_lpm* required. Set Low Power Mode option on the driver to Software Standby Mode.	Refer to *BLE sample application (R01AN5402)* chapter 4 about adding *Low Power Modes Driver on r_lpm* to user application.

2.4.4.1 RF Sleep mode

The BLE Protocol Stack will transit to RF sleep mode to reduce current consumption of RF part when the following conditions are met.

- *RF_DEEP_SLEEP Transition* option (see section 2.3) is set to enable.
- There is no task to be executed by BLE Protocol Stack.
- There is a time of 80ms or more before the start of the next RF event time.
  - The “time” means the “RF idle time” between the completion of one RF event and the start of the next RF event. Therefore, it is necessary to set the RF event interval to 100ms or more in consideration of the processing time of each layer to shift the RF part to sleep mode. In Scanning operation, the time difference between the Scan interval and Scan window must also be set to 100ms or more.

The BLE Protocol Stack performs RF sleep processing and RF wake-up processing to transition the RF part to sleep mode. Figure 5 shows MCU/RF operation overview with RF sleep.

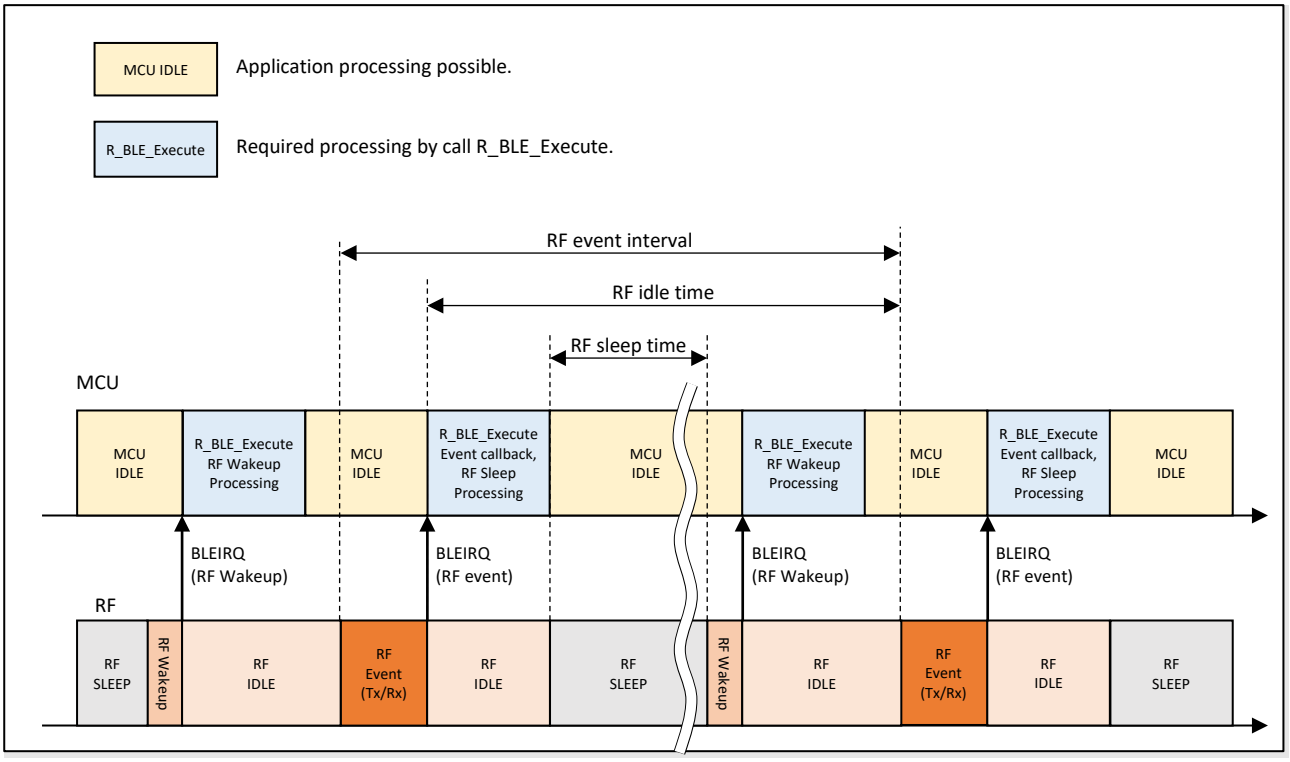


Figure 5. MCU/RF operation overview with RF sleep

While the MCU is idle, it is possible to transition the MCU to the low power consumption mode or execute processing of the other application. However, if the RF wakeup process is not performed before the RF event starts, the RF event cannot be executed because the other application process occupied. Therefore, application processing must be implemented so as not to interfere RF event execution.

The current consumption during RF idle time will increases when the condition to transition to RF sleep mode is not met. However, the MCU idle time can be used in user application processes since it is not necessary to consider the RF wake up process. Figure 6 shows MCU/RF operation without RF sleep.

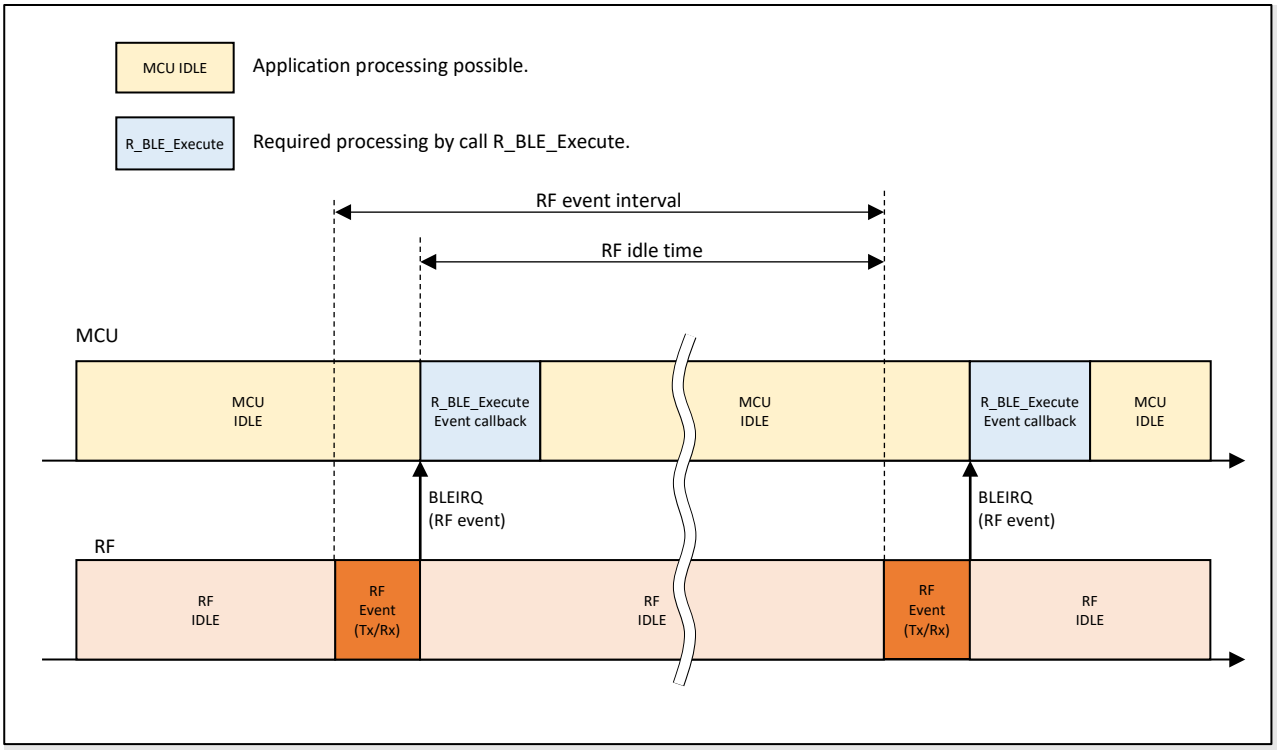


Figure 6. MCU/RF operation overview without RF sleep

If user application occupies the MCU and the RF event cannot be executed, then the BLE connection will be lost. It is recommended that the application is processed is short time to ensure RF event execution time.

Note: The Bluetooth LE Protocol Stack initializes the RF hardware state by R_BLE_Open. If the software is reset during RF communication operation, call R_BLE_Open to initialize the RF hardware state and stop RF operation.

2.4.4.2 MCU low power mode

(1) BareMetal environment

The MCU can be shifted to the low power consumption state even when using a BLE function. The basic policy of the transition to the low power consumption state is as below.

- BLE applications can use the MCU Low Power Mode between the completion of the RF event execution and the start of the next RF event execution.
- It is necessary to check whether all the used components (including the BLE function) can shift MCU to low power consumption state or not before entering MCU Low Power Mode.
- When BLE communication occurs, it resumes from MCU Low Power Mode by RF interrupt.
  However, since there is a possibility that RF interrupt may occur during the processing of disabling interrupts, check the status of BLE task once after disabling interrupts, If BLE task state is not free, skip the transition to low power consumption state of MCU.

Refer to *RA4W1 Group User's Manual: Hardware (R01UH0883)* regarding MCU low power mode.

Example for entering MCU low power mode is shown in Code 3.

```
void app_main(void)
{
……
    /* Initialize Low Power Module */
    g_lpm0.p_api->open(g_lpm0.p_ctrl, g_lpm0.p_cfg);

    /* Initialize BLE and profiles */
    ble_init();
……
    /* main loop */
    while (1)
    {
        /* Process BLE Event */
        R_BLE_Execute();
……
        /* Disable IRQ */
        __disable_irq();

        /* Check whether there are executable BLE task or not */
        if (0 != R_BLE_IsTaskFree())
        {
            /* There are no executable BLE task */
            /* Enter low power mode */
            g_lpm0.p_api->lowPowerModeEnter(g_lpm0.p_ctrl);

            /* Enable interrupt for processing interrupt handler after wake up */
            __enable_irq();
        }
        else
        {
            /* There is BLE related task */
            __enable_irq();
        }
……
}
```

Initialize MCU Low Power Driver on the top of application main.

Check whether it can be use MCU Low Power mode.

User can add further check here if needed.

(e.g. Check the operating status of peripheral which user application used)

Code 3. Transition to MCU Low power mode

(2) FreeRTOS environment

FreeRTOS kernel enters MCU Low Power Mode in *vApplicationIdleHook*.

3. How to implement user code

QE for BLE generates:

- Application profiles.
- Skeleton code for the user's BLE application.

These QE for BLE generated codes are ready to connect to the remote device shown in Figure 7. The BLE Protocol Stack automatically handles the dotted line responses and operations. Therefore, no code is required.

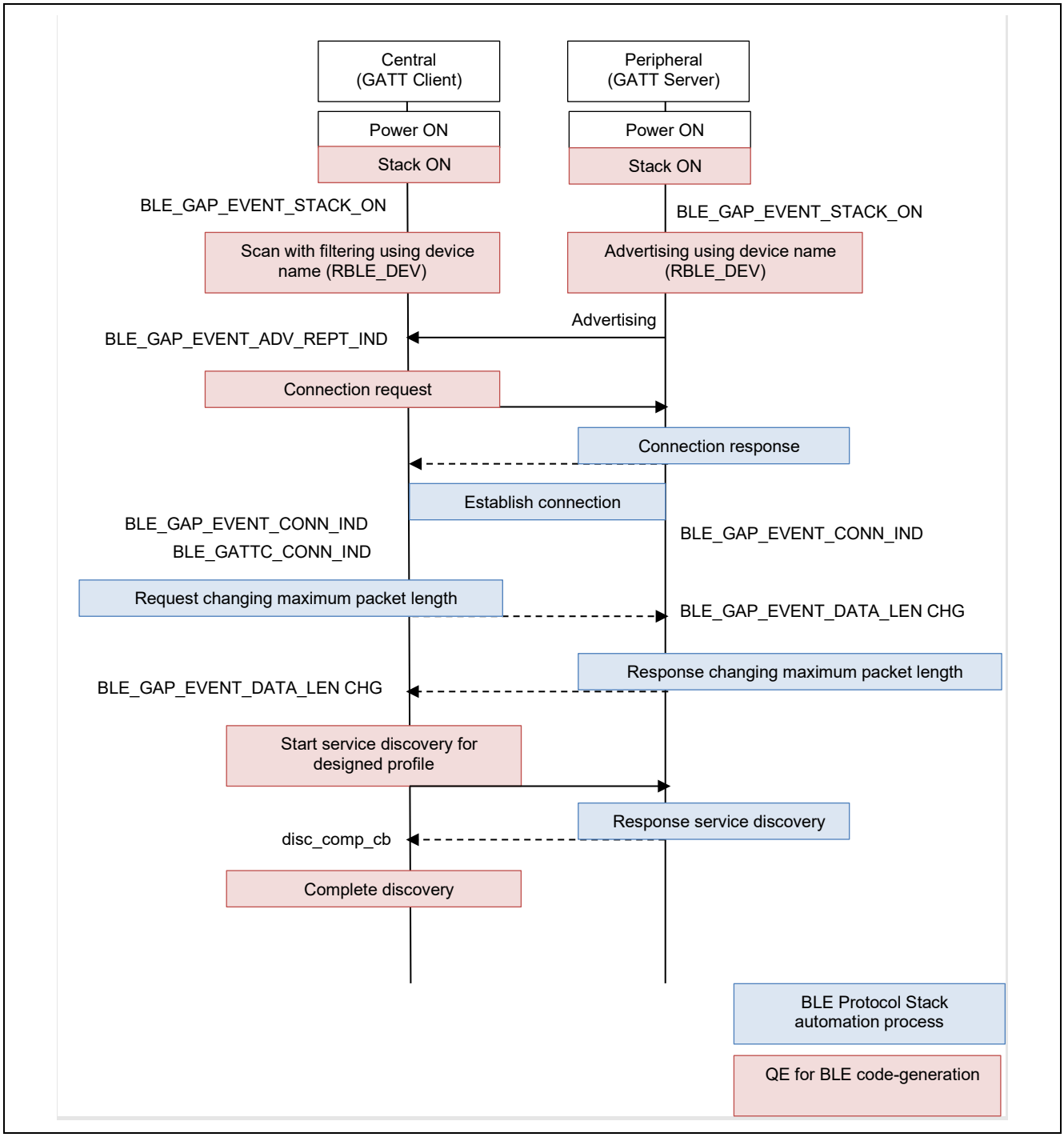


Figure 7. Behavior of codes generated by QE for BLE

The User can use APIs shown in Table 13 when making BLE application.

Table 13. APIs for making BLE application

Functions	API name	Usage
Bluetooth LE	R_BLE_XXX R_BLE_GAP_XXX R_BLE_GATT_GetMtu R_BLE_GATTS_XXX R_BLE_GATTC_XXX R_BLE_L2CAP_XXX	• R_BLE_GAP_XXX ➢ These are the APIs for performing GAP procedure such as advertising. ➢ API result can be received as BLE_GAP_EVENT_XXX event after registering callback function by using RM_BLE_ABS_Open or R_BLE_GAP_Init API. ➢ The Registered callback function is called when performing Advertising, Scanning, establishing connection, etc. • R_BLE_GATTS_XXX ➢ These are the APIs for performing GATT server procedures such as sending a notification, ➢ API result can be received as BLE_GATTS_EVENT_XXX event after registering callback function by using RM_BLE_ABS_Open or R_BLE_GATTS_RegisterCb API. ➢ The registered callback function is called when receiving a result of R_BLE_GATTS_XXX API and when access from the client has occurred. • R_BLE_GATTC_XXX ➢ These are the APIs for performing GATT client procedures such as sending a read / write. ➢ API result can be received as BLE_GATTC_EVENT_XXX event after registering callback function by using RM_BLE_ABS_Open or R_BLE_GATTC_RegisterCb API. ➢ The registered callback function is called when receiving a result of R_BLE_GATTC_XXX API and when access from the server has occurred. • R_BLE_L2CAP_XXX ➢ These are the APIs for performing L2CAP procedures such as sending a connection request for a L2CAP Credit Base Flow Control channel. ➢ API result can be received as BLE_L2CAP_EVENT_XXX event after registering callback function by using R_BLE_L2CAP_RegisterCfPsm API. ➢ The registered callback function is called when a response of L2CAP Credit-Base Flow Control request is returned, etc. • R_BLE_XXX and R_BLE_GATT_GetMtu ➢ No need to register callback function since the result of these API can be received immediately. Note : BLE_XXX_Init, R_BLE_XXX_RegisterCb, R_BLE_GAP_SetPairingParams API can also receive API result immediately.
Vendor Specific (VS)	R_BLE_VS_XXX	• These are the APIs for performing vender specific procedures such as getting a BD address. • API result can be received as BLE_VS_EVENT_XXX event once registering callback function by using RM_BLE_ABS_Open or R_BLE_VS_Init API. • The registered callback function is called when receiving a result of R_BLE_VS_XXX API.

Functions	API name	Usage
Abstraction API	RM_BLE_ABS_XXX	• These are the APIs that abstract various Bluetooth LE procedures and make them easy to use. • To use these abstraction APIs, it is necessary to call the RM_BLE_ABS_Open API.
Service discovery	R_BLE_DISC_Start	• This API is for performing the service discovery procedure. • Specify the callback function to be called when service discovery procedure is completed as an argument.
Profile API	R_BLE_ [service name] [S: Server, C: Client] _XXX	• These are the APIs for accessing the characteristics of each GATT profile. • These APIs are generated by QE for BLE. • Once registering callback function using R_BLE_[service name]_Init, an event can be received as BLE_[service name][S: Server, C: Client]_EVENT_XXX. • The registered callback function is called when receiving a Write, Read, Indication or Notification from remote device.
Profile common	R_BLE_SERV [S: Server, C: Client] _XXX	• These are common APIs for Profile API. • These APIs are generated by QE for BLE.

Example of user application implementation has provided in *BLE sample application (R01AN5402).*

Note: Since R_BLE_Close() stops RF H/W, when resetting software during RF communication, be sure to call R_BLE_Close() before resetting.

4. Advertising

Bluetooth LE devices broadcast data to nearby scanning devices by advertising. This chapter describes how to use the advertising feature by using related APIs. Figure 8 shows the flow chart of advertising procedure in an BLE application. Details of each step are explained in the following sections.

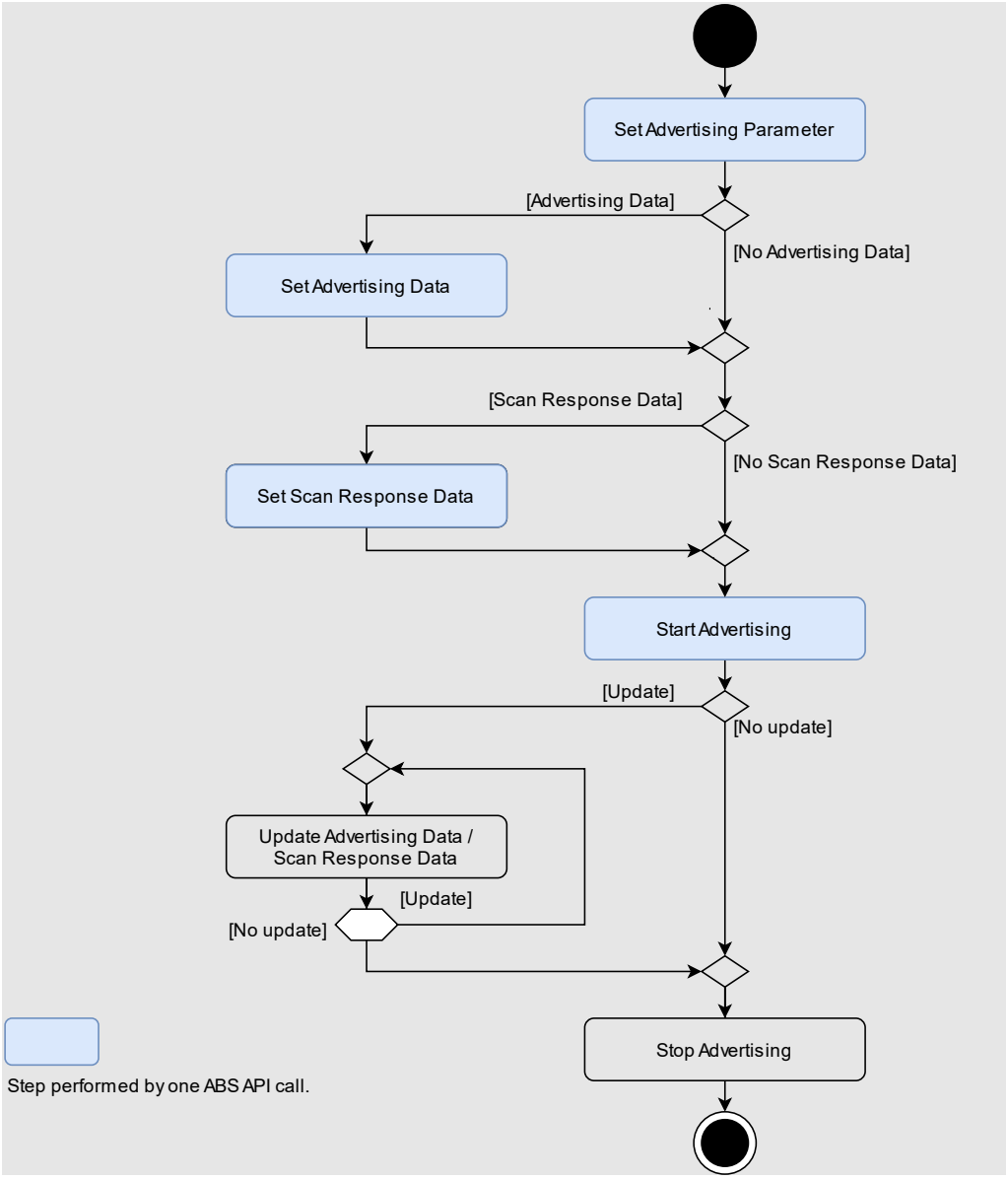


Figure 8. Advertising procedure

Users can use the following categories of APIs to perform the above procedure.

- **Abstraction API (*RM_BLE_ABS_XXX* API)**
  - Users can use the advertising feature with a single API call. However, detailed parameter settings are not possible.

- **GAP API (*R_BLE_GAP_XXX* API)**
  - Users can use the advertising feature by combining several APIs. Detailed parameter settings are possible.

4.1 Advertising with abstraction API

When users use the abstraction API, the procedure from setting advertising parameters to starting advertising is performed by a single abstraction API call. This kind of abstraction APIs are defined in Table 14. Refer to *Renesas Flexible Software Package User's manual* about the usage of these APIs. The samples of typical use cases are shown in section 4.5. Refer to section 4.2.4 about stop advertising.

Table 14. Advertising type supported by the Abstraction API

Abstraction API	Legacy or Extended	Advertising Type	Advertising PDU	Advertising handle	Maximum Advertising Data Size (Bytes)
RM_BLE_ABS_StartLegacy Advertising	Legacy	Connectable and Scannable Undirected	ADV_IND	0	31
RM_BLE_ABS_StartExtended Advertising	Extended	Connectable Undirected	ADV_EXT_IND	1	229
			AUX_ADV_IND		
		Connectable Directed	ADV_EXT_IND		229
			AUX_ADV_IND		
RM_BLE_ABS_StartNonConnectable Advertising	Legacy	Non-Connectable and Non-Scannable Undirected	ADV_NONCONN_IND	2	31
	Extended		ADV_EXT_IND		Maximum Advertising Data Length[*1]
			AUX_ADV_IND		
			AUX_CHAIN_IND		
RM_BLE_ABS_StartPeriodic Advertising	Extended	Periodic	ADV_EXT_IND	3	Maximum Advertising Data Length[*1]
			AUX_ADV_IND		
			AUX_SYNC_IND		
			AUX_CHAIN_IND		

[*1]: Configure in properties of BLE module. Refer to *BLE sample application (R01AN5402)* Chapter 4.

The supported advertising type depends on the BLE module configuration option. "Extended" configuration supports legacy and extended advertising. "Balance" and "Compact" configurations only support legacy advertising.

4.1.1 Whitelist

Whitelist is a feature that filters a specific BD address from the received wireless packet. *RM_BLE_ABS_StartLegacyAdvertising* and *RM_BLE_ABS_StartExtendedAdvertising* APIs can use the feature by applying following steps.

1. Register a known device BD address to the whitelist by calling *R_BLE_GAP_ConfWhiteList* API.

Note: The Whitelist cannot be added/deleted when the Whitelist filter enabled operation (advertising, scanning, connection request) is executed.

2. Set value listed in Table 18 to use whitelist feature for *advertising_filter_policy* field in:
   - *ble_abs_legacy_advertising_parameter_t* structure when use *RM_BLE_ABS_StartLegacyAdvertising* API.
   - *ble_abs_extended_advertising_parameter_t* structure when use *RM_BLE_ABS_StartExtenedAdvertising* API.

4.1.2 Privacy

Privacy is a feature that prevents other devices from tracking advertising packet by periodically changing BD address, which is a part of advertising packet. Advertising related abstraction APIs can use the privacy feature except *RM_BLE_ABS_StartPeriodicAdvertising* API. Privacy feature can be used after preparing local IRK for using privacy feature according to section 8.4 and set value of Table 15 to:

- *ble_abs_legacy_advertising_parameter_t* structure when using *RM_BLE_ABS_StartLegacyAdvertising* API.
- *ble_abs_extended_advertising_parameter_t* structure when using *RM_BLE_ABS_StartExtenedAdvertising* API.
- *ble_abs_non_connectable_advertising_parameter _t* structure when using *RM_BLE_ABS_StartNonConnectableAdvertising* API.

Table 15. Parameters used for the privacy feature

Field	Value	Description
own_bluetooth_address_type	BLE_GAP_ADDR_RPA_ID_PUBLIC(0x02)	If the IRK of the local device has not been registered in Resolving List, public address is used.
	BLE_GAP_ADDR_RPA_ID_RANDOM(0x03)	If the IRK of the local device has not been registered in Resolving List, the random address specified by the own_bluetooth_address field is used.
own_bluetooth_address	Specify the Static Address registered by *R_BLE_GAP_SetLocIdInfo*.API	Specify the value if the own_bluetooth_address_type is BLE_GAP_ADDR_RPA_ID_RANDOM(0x03).
p_peer_address	Specify the remote device identity address registered by *R_BLE_GAP_ConfRslvList* API.	---

4.2 Advertising with GAP API

When a user uses GAP API, the procedure from setting advertising parameters to starting or stopping advertising is performed by combining several API calls. This section describes each procedure. Procedures of this section do not need the use of abstraction API.

4.2.1 Set advertising parameter

It is necessary to configure advertising parameter to *st_ble_gap_adv_param_t* structure and call *R_BLE_GAP_SetAdvParam* API. Refer to *Renesas Flexible Software Package User's Manual* for more details about the structure. These parameters cannot be changed during advertising. If you use the Abstraction API, the procedure does not need. The following sections describe the parameter settings for some use cases.

4.2.1.1 Advertising Type

Advertising type is specified by a combination of following items.

- Response to a connection request from remote device (Connectable or Non-Connectable)
- Response to a scan request from remote device (Scannable or Non-Scannable)
- Designation of remote address (Direct or Undirect)
- Type of advertising that a remote device supports (legacy or extended advertising)
- Maximum size of the Advertising Data

The above combination is specified by *adv_prop_type* field in *st_ble_gap_adv_param_t* structure as shown in Table 16.

Table 16. Correspondence between Advertising type and adv_prop_type field

Advertising Type	Advertising PDU	The adv_prop_type field value	legacy or extended	Max Size(byte)
Connectable and Scannable Undirected [*5]	ADV_IND	BLE_GAP_LEGACY_PROP_ADV_IND	legacy	31
Connectable Undirected	ADV_EXT_IND AUX_ADV_IND	BLE_GAP_EXT_PROP_ADV_CONN_NOSCAN_UNDIRECT	extended	245[*1*4]
Connectable Directed	ADV_DIRECT_IND	BLE_GAP_LEGACY_PROP_ADV_DIRECT_IND or BLE_GAP_LEGACY_PROP_ADV_HDC_DIRECT_IND	legacy	0
	ADV_EXT_IND AUX_ADV_IND	BLE_GAP_EXT_PROP_ADV_CONN_NOSCAN_DIRECT or BLE_GAP_EXT_PROP_ADV_CONN_NOSCAN_HDC_DIRECT	extended	239[*1*4]
Non-Connectable and Non-Scannable Undirected	ADV_NONCONN_IND	BLE_GAP_LEGACY_PROP_ADV_NONCONN_IND	legacy	31
	ADV_EXT_IND AUX_ADV_IND AUX_CHAIN_IND[*2]	BLE_GAP_EXT_PROP_ADV_NOCONN_NOSCAN_UNDIRECT	extended	Maximum Advertising Data Length[*4 *6]
Non-Connectable and Non-Scannable Directed	ADV_EXT_IND AUX_ADV_IND AUX_CHAIN_IND[*3]	BLE_GAP_EXT_PROP_ADV_NOCONN_NOSCAN_DIRECT or BLE_GAP_EXT_PROP_ADV_NOCONN_NOSCAN_HDC_DIRECT	extended	Maximum Advertising Data Length[*4 *6]
Scannable Undirected [*5]	ADV_SCAN_IND	BLE_GAP_LEGACY_PROP_ADV_SCAN_IND	legacy	31
	ADV_EXT_IND AUX_ADV_IND	BLE_GAP_EXT_PROP_ADV_NOCONN_SCAN_UNDIRECT	extended	0
Scannable Directed [*5]	ADV_EXT_IND AUX_ADV_IND	BLE_GAP_EXT_PROP_ADV_NOCONN_SCAN_DIRECT or BLE_GAP_EXT_PROP_ADV_NOCONN_SCAN_HDC_DIRECT	extended	0

[*1]: If the *BLE_GAP_EXT_PROP_ADV_INCLUDE_TX_POWER* is added to *adv_prop_type* field, it's Max Size-1 byte.

[*2]: *AUX_CHAIN_IND* is not used if the size of Advertising Data is 245 bytes or less (It's reduced -18 bytes when using Periodic advertising. Furthermore, it is reduced by 1 byte when using *BLE_GAP_EXT_PROP_ADV_INCLUDE_TX_POWER*), since Advertising Data can be sent only with *AUX_ADV_IND*.

[*3]: *AUX_CHAIN_IND* is not used, if the size of Advertising Data is 239 bytes or less (It's reduced -18 bytes when using Periodic advertising. It's reduced -1 byte when using *BLE_GAP_EXT_PROP_ADV_INCLUDE_TX_POWER*), since Advertising Data can be sent only with *AUX_ADV_IND, AUX_CHAIN IND* is not used.

[*4]: If the size of Advertising Data is 230 bytes or more, since Advertising Data is divided according to Bluetooth specification, combine them on the receiver if necessary.

[*5]: The relationship between Scan Response Data and PDU and type is shown in Figure 10.

[*6]: Configure in properties of BLE module. Refer to *BLE sample application (R01AN5402)* Chapter 4.

The available advertising type depends on the BLE module configuration option. "Extended" configuration supports legacy and extended advertising. "Balance" and "Compact" configuration support only legacy advertising. If a scanner supports only the legacy advertising, it cannot receive extended advertising packets.

If the advertising type is extended and non-scannable, each PDU is sent in order shown in Figure 9. The *advDelay* is a random delay from 0 to 10ms.

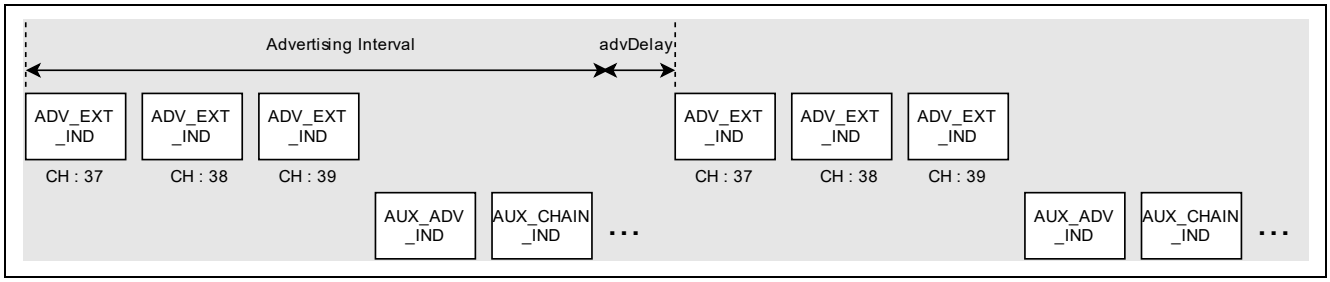


Figure 9. Extended Advertising PDU

If the advertising type is scannable and the Scan Response Data is set, the Scan Response Data is shown in Table 17. Scan response PDUs are sent in order shown in Figure 10.

Table 17. Scan Response Data PDU

Value set to the adv_prop_type field	Scan Response Data PDU	legacy or extended	Max Size (byte)
BLE_GAP_LEGACY_PROP_ADV_IND BLE_GAP_LEGACY_PROP_ADV_SCAN_IND	SCAN_RSP	legacy	31
BLE_GAP_EXT_PROP_ADV_NOCONN_SCAN_UNDIRECT BLE_GAP_EXT_PROP_ADV_NOCONN_SCAN_DIRECT BLE_GAP_EXT_PROP_ADV_NOCONN_SCAN_HDC_DIRECT	AUX_SCAN_RSP AUX_CHAIN_IND*1	extended	Maximum Advertising Data Length*2 *3 *4

*1 : AUX_CHAIN_IND is not used if the Scan Response Data is 253 bytes or less (It is reduced by 1 byte when using *BLE_GAP_EXT_PROP_ADV_INCLUDE_TX_POWER*), since Scan Response Data can be sent only with *AUX_SCAN_RSP*.

*2 : If the *BLE_GAP_EXT_PROP_ADV_INCLUDE_TX_POWER* is added to adv_prop_type, it's Max Size -1 byte.

*3 : If the size of Scan Response Data is 230 bytes or more, since Scan Response Data is divided according to Bluetooth specification, combine them on the receiver if necessary.

*4: Configure in properties of BLE module. Refer to *BLE sample application (R01AN5402)* Chapter 4.

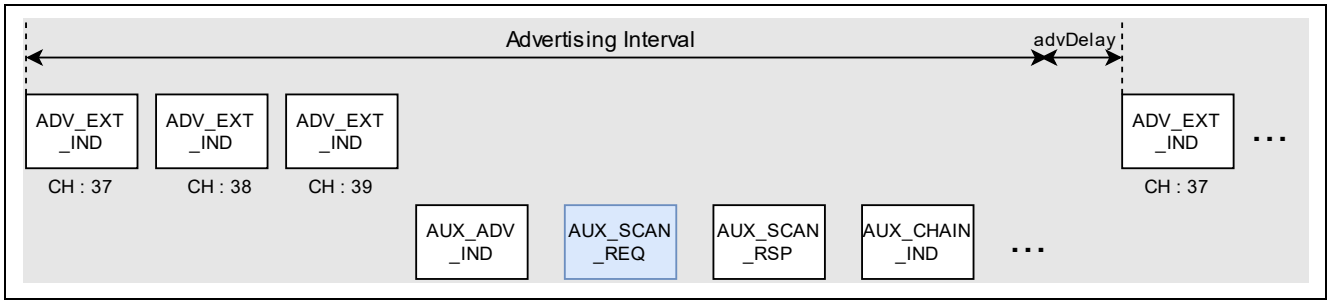


Figure 10. Scannable Advertising PDU

The blue box shows the PDU from a remote device. If the advertising type is Direct, set a remote device address to the *p_addr_type* and the *p_addr* field in the *st_ble_gap_adv_param_t* structure.

If the advertising type is Extended, set the PHY that sends Advertising to the adv_phy and the sec_adv_phy field in the st_ble_gap_adv_param_t structure. Specify the PHY (1M PHY or Coded PHY) of the primary channel (CH:37/38/39) for adv_phy. Specify the PHY (1M PHY, 2M PHY or Coded PHY) of the secondary channel (other than CH:37/38/39) for sec_adv_phy.

4.2.1.2 Whitelist

Whitelist is a feature that filters a specific BD address from the received wireless packet. If the advertising type is connectable or scannable, whitelist feature can be used by applying following steps.

1. Register a known device BD address to the whitelist by calling *R_BLE_GAP_ConfWhiteList* API.

Note: The Whitelist cannot be added/deleted when the Whitelist filter enabled operation (advertising, scanning, connection request) is executed.

2. Set to use whitelist feature for *filter_policy* field in *st_ble_gap_adv_param_t* structure as shown in Table 18.

Table 18. The value set to the filter_policy field

Value set to the filter_policy field	Description
BLE_GAP_SCAN_ALLOW_ ADV_ALL(0x00)	Respond to scan requests and connection requests from all devices.
BLE_GAP_ADV_ALLOW_ SCAN_WLST_CONN_ANY(0x01)	Respond to scan requests from whitelisted devices and respond to connection requests from all devices.
BLE_GAP_ADV_ALLOW_ SCAN_ANY_CONN_WLST(0x02)	Respond to scan requests from all devices and respond to connection requests from whitelisted devices.
BLE_GAP_ADV_ALLOW_ SCAN_WLST_CONN_WLST(0x03)	Respond to scan requests and connection requests from whitelisted devices.

4.2.1.3 Privacy

Privacy is a feature that prevents other devices from tracking advertising packet by periodically changing BD address, which is a part of the advertising packet. To use the privacy function, it is necessary to configure the field shown in Table 19 in *st_ble_gap_adv_param_t* structure and perform procedure described in section 8.4. The default update interval for BD address is 900 seconds. The interval can be changed with *R_BLE_GAP_SetRpaTo*().

Table 19. The parameters used for the privacy feature

Field	Value	Description
o_addr_type	BLE_GAP_ADDR_RPA_ID_PUBLIC(0x02)	If the IRK of local device has not been registered in Resolving List, public address is used.
	BLE_GAP_ADDR_RPA_ID_RANDOM(0x03)	If the IRK of local device has not been registered in Resolving List, the random address specified by the o_addr field is used.
o_addr	Specify the Static Address registered by R_BLE_GAP_SetLocIdInfo().	Specify the value if the o_addr_type is BLE_GAP_ADDR_RPA_ID_RANDOM(0x03).
p_addr_type	Specify the remote device identity address registered by R_BLE_GAP_ConfRslvList().	—
p_addr		

BP: Including advertising data that uniquely identifies a device may defeat the purpose of using private addresses to hide the device. Advertisement payloads obfuscation is recommended when using private addressing.

BP: The advertising data can be used to track the device even if the device address changes periodically to different addresses. It is therefore recommended to update the address and data at the same time.

4.2.1.4 Multiple advertising set

Multiple advertising set is a feature that broadcasts different parameters of Advertising in parallel. How many sets of advertising can be sent is configured by *Maximum advertising set number configuration* in properties of BLE module. Refer to *BLE sample application (R01AN5402)* in detail of configuration. Each advertising set is identified by *adv_hdl* field in the *st_ble_gap_adv_param_t* structure. However, when using multiple advertising set feature with abstraction APIs, the advertising handle is determined as Table 14 for each abstraction API.

4.2.2 Advertising Data / Scan Response Data

Refer to section 4.4.

4.2.3 Start Advertising

When starting advertising, call the *R_BLE_GAP_StartAdv* API. It is necessary to specify following arguments when calling the API.

- *adv_hdl*: advertising handle to start advertising.
- *duration*: advertising continuing period (duration x 10ms).
- *max_extd_adv_evt*: number of broadcasting advertising packets.

4.2.4 Stop Advertising

The API for stopping advertising, call *R_BLE_GAP_StopAdv* API. It is necessary to specify the advertising handle which want to stop advertising with argument *adv_hdl*. In case of connectable advertising, the advertising will stop automatically when established connection with a remote device. This API can be used both when Advertising starts with abstraction API and when it starts with GAP API.

4.3 Periodic Advertising with GAP API

Periodic advertising is a feature that broadcasts periodic advertising PDUs at predictable timing. When the scanner performs the synchronization with periodic advertising which is described in section 5.4, the scanner can get periodic advertising PDUs. The following sections describe the details of periodic advertising procedure shown in Figure 11.

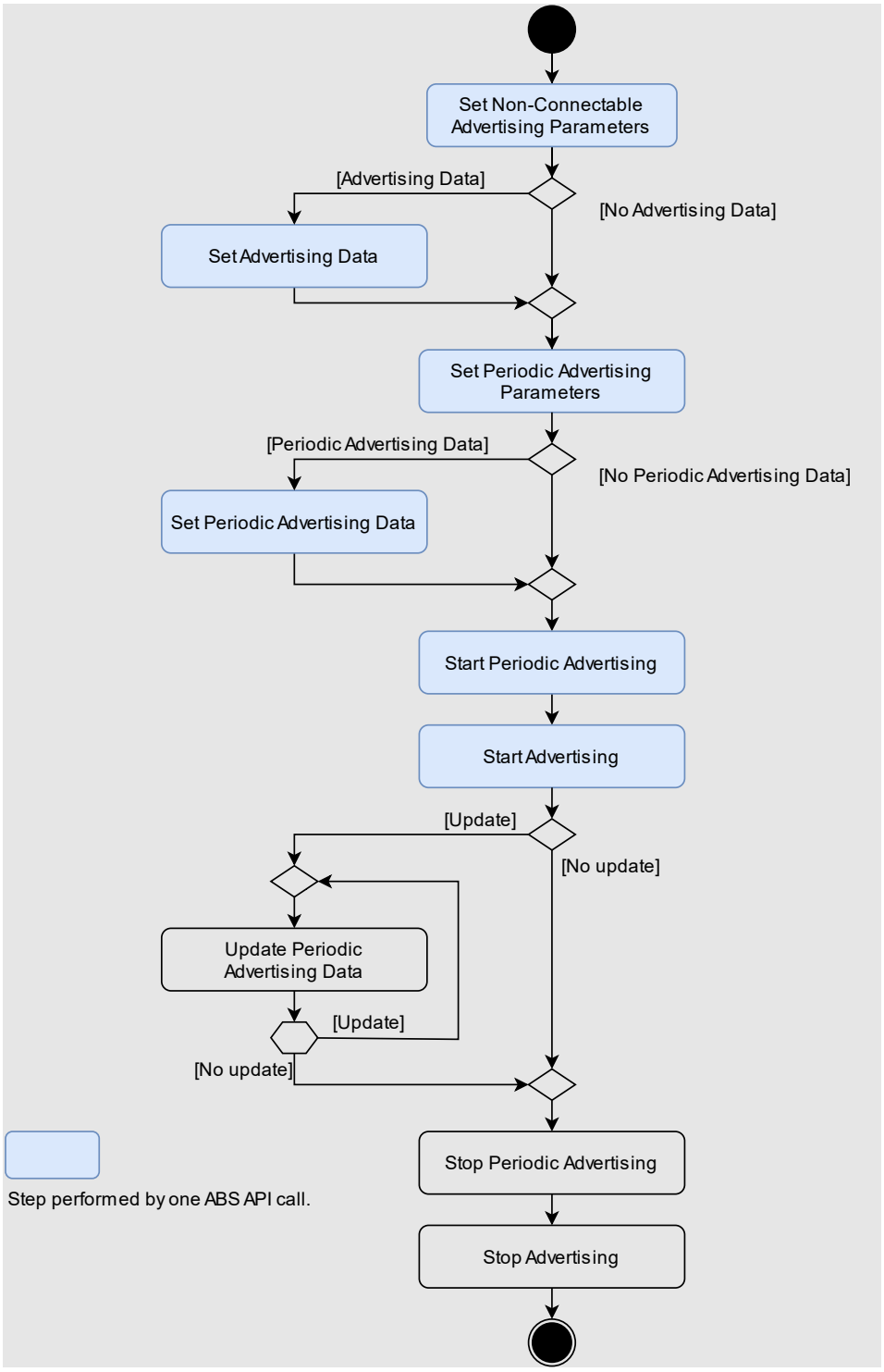


Figure 11. Periodic Advertising procedure

Periodic advertising feature is only available in “Extended” configuration, which is a BLE module configuration option.

4.3.1 Non-Connectable Advertising Parameter

To start periodic advertising, it is necessary to configure advertising parameter as a non-connectable advertising by using *R_BLE_GAP_SetAdvParam* API. Also, Refer to section 4.2.1.

4.3.2 Periodic Advertising Parameter

It is necessary to configure *st_ble_gap_perd_adv_param_t* structure and call *R_BLE_GAP_SetPerdAdvParam* API with the structure. Refer to *Renesas Flexible Software Package User's Manual* about the details of the structure.

4.3.3 Periodic Advertising Data

For details about setting the Periodic Advertising Data, refer to section 4.4.

4.3.4 Start Periodic Advertising

When starting periodic advertising, call *R_BLE_GAP_StartPerdAdv* API. The periodic advertising PDUs are shown in Table 20 and broadcast timing is shown in Figure 12.

Table 20. Periodic Advertising PDU

Advertising type	Periodic advertising PDU	Legacy or Extended	Maximum Size (Bytes)
Periodic Advertising	AUX_SYNC_IND	extended	Maximum advertising data length*2, *3, *4
	AUX_CHAIN_IND*1		

*1 : *AUX_CHAIN IND* is not used if the size of Periodic Advertising Data is 253 bytes or less (It's reduced -1 byte when setting *BLE_GAP_EXT_PROP_ADV_INCLUDE_TX_POWER* to *prop_type* field of *st_ble_gap_ext_adv_param_t*), since Periodic Advertising Data can be sent only with *AUX_SYNC_IND*.

*2 : If the *BLE_GAP_EXT_PROP_ADV_INCLUDE_TX_POWER* is added to adv_prop_type, it's Max Size -1 byte.

*3 : If the size of Periodic Advertising Data is 248 bytes or more, since Periodic Advertising Data is divided according to Bluetooth specification, combine them on the receiver if necessary.

*4: Configure in properties of BLE module. Refer to *BLE sample application (R01AN5402)* Chapter 4.

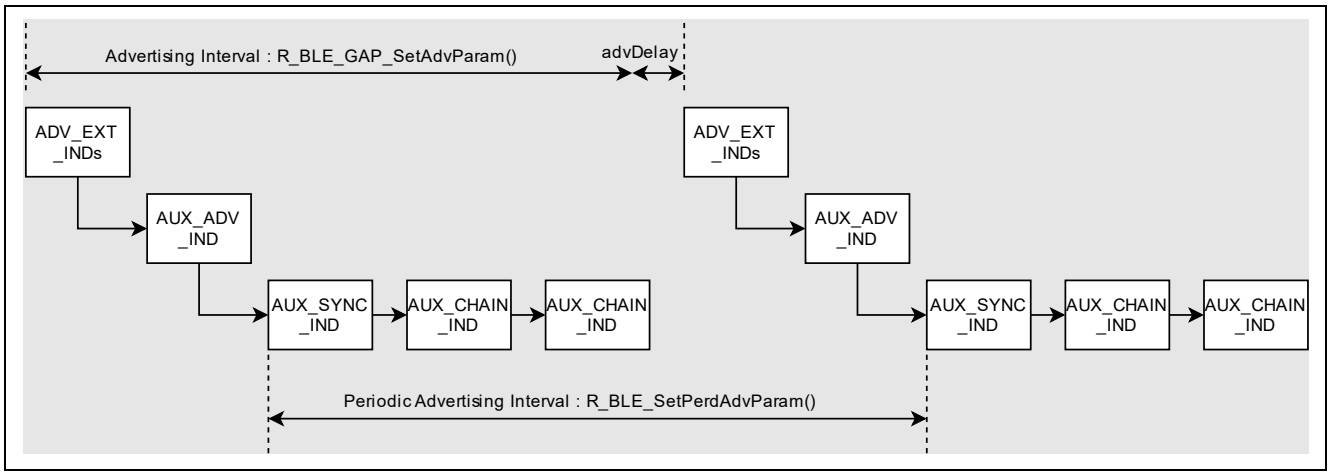


Figure 12. Periodic Advertising PDUs

An example of starting Periodic Advertising is shown in Code 4.

```
/* Advertising data */
static uint8_t gs_adv_data[] =
{

    /* Flag (mandatory) */
    2,          /* Data Size */
    0x01,       /* Data Type: Flag */
    (BLE_GAP_AD_FLAGS_LE_GEN_DISC_MODE |
     BLE_GAP_AD_FLAGS_BR_EDR_NOT_SUPPORTED),    /* Data */

    /* Complete Local Name */
    9,          /* Data Size */
    0x09,       /* Data Type: Complete Local Name */
    'R', 'B', 'L', 'E', '-', 'D', 'E', 'V', /* Data */
};

/* Periodic Advertising Data */
static uint8_t gs_perd_adv_data[] =
{

    /* Complete Local Name */
    9,         /* Data Size */
    0xFF,      /* Data Flag: Manufacturer Specific data type  */
    0x36, 0x00,/* Company ID: Renesas Electronics Corporation */
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05,  /* Data */

};

/* some code is omitted. */
static void gap_cb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
    st_ble_gap_adv_set_evt_t * p_adv_set_param;

    switch(type)
    {
        case BLE_GAP_EVENT_STACK_ON :
        {
                st_ble_gap_adv_param_t adv_param =
                {
                        .adv_hdl       = 0x02,
                        .adv_prop_type = BLE_GAP_EXT_PROP_ADV_NOCONN_NOSCAN_UNDIRECT,
                        .adv_intv_min  = 0x0200,
                        .adv_intv_max  = 0x0200,
                        .adv_ch_map    = BLE_GAP_ADV_CH_ALL,
                        .o_addr_type   = BLE_GAP_ADDR_PUBLIC,
                        .filter_policy = BLE_GAP_ADV_ALLOW_SCAN_ANY_CONN_ANY,
                        .adv_phy       = BLE_GAP_ADV_PHY_1M,
                        .sec_adv_phy   = BLE_GAP_ADV_PHY_1M,
                };
                /* Set Advertising parameter */
                R_BLE_GAP_SetAdvParam(&adv_param);
        } break;

        case BLE_GAP_EVENT_ADV_PARAM_SET_COMP :
        {
                p_adv_set_param = (st_ble_gap_adv_set_evt_t *)p_data->p_param;
                st_ble_gap_adv_data_t adv_data_param =
                {
                        .adv_hdl        = 0x02,
                        .data_type      = BLE_GAP_ADV_DATA_MODE,
                        .data_length    = ARRAY_SIZE(gs_adv_data),
                        .p_data         = gs_adv_data  ,
                };
                /* Set Advertising Data */
                R_BLE_GAP_SetAdvSresData(&adv_data_param);
        } break;
```

```
        case BLE_GAP_EVENT_PERD_ADV_PARAM_SET_COMP :
        {
                /* Periodic Advertising Data parameter */
                st_ble_gap_adv_data_t perd_adv_data_param = {
                        .adv_hdl        = 0x02,
                        .data_type      = BLE_GAP_PERD_ADV_DATA_MODE,
                        .data_length    = ARRAY_SIZE(gs_perd_adv_data),
                        .p_data         = gs_perd_adv_data  ,
                };

                /* Set Periodic Advertising Data */
                R_BLE_GAP_SetAdvSresData(&perd_adv_data_param);
        } break;

        case BLE_GAP_EVENT_PERD_ADV_ON :
        {
                p_adv_set_param = (st_ble_gap_adv_set_evt_t *)p_data->p_param;
                /* Start Advertising */
                R_BLE_GAP_StartAdv(0x02, 0, 0);
        }
        break;

        case BLE_GAP_EVENT_ADV_DATA_UPD_COMP :
        {
                st_ble_gap_adv_data_evt_t * p_adv_data_set_param;
                p_adv_data_set_param = (st_ble_gap_adv_data_evt_t *)p_data->p_param;
                if (BLE_GAP_ADV_DATA_MODE == p_adv_data_set_param->data_type)
                {
                        st_ble_gap_perd_adv_param_t perd_param =
                        {
                                .adv_hdl       = 0x02,
                                .prop_type     = 0x0000,
                                .perd_intv_min = 0x0100,
                                .perd_intv_max = 0x0100,
                        };
                        /* Set Periodic Advertising parameter */
                        R_BLE_GAP_SetPerdAdvParam(&perd_param);
                }
                else
                {
                        if(BLE_GAP_PERD_ADV_DATA_MODE == p_adv_data_set_param->data_type)
                        {
                                /* Start Periodic Advertising parameter */
                                R_BLE_GAP_StartPerdAdv(0x02);
                        }
                }
        } break;

        default:
        break;
    }
}
```

Code 4. Sample of starting Periodic Advertising

4.3.5 Stop Periodic Advertising

When stopping Periodic Advertising, call *R_BLE_GAP_StartPerdAdv* API.

4.4 Advertising Data / Scan Response Data / Periodic Advertising Data

Advertising PDU could include the following data to inform the auxiliary information to the scanner device.

- Advertising Data
- Scan Response Data
- Periodic Advertising Data

It is necessary to call *R_BLE_GAP_SetAdvSresData* API to configure / update above data. These APIs can be used even if user perform advertising by using abstraction API. The API has argument of *st_ble_gap_adv_data_t* structure. The *data_type* field in the structure is set, as shown in Table 21.

Table 21. Value set to the data_type field

Data type	Value set to the data_type field
Advertising Data	BLE_GAP_ADV_DATA_MODE(0x00)
Scan Response Data	BLE_GAP_SCAN_RSP_DATA_MODE(0x01)
Periodic Advertising Data	BLE_GAP_PERD_ADV_DATA_MODE(0x02)

When setting advertising data and scan response data continuously, it is necessary to perform following steps.

1. Set advertising data by calling *R_BLE_GAP_SetAdvSresData* API.
2. Confirm the completion of setting the advertising data by checking the *BLE_GAP_EVENT_ADV_DATA_UPD_COMP* event in the GAP callback function.
3. Set scan response data by calling *R_BLE_GAP_SetAdvSresData* API.
4. Confirm the completion of setting the scan response data by checking the *BLE_GAP_EVENT_ADV_DATA_UPD_COMP* event in the GAP callback function.

4.4.1 Data format

Figure 13 shows the format of Advertising Data / Scan Response Data / Periodic Advertising Data.

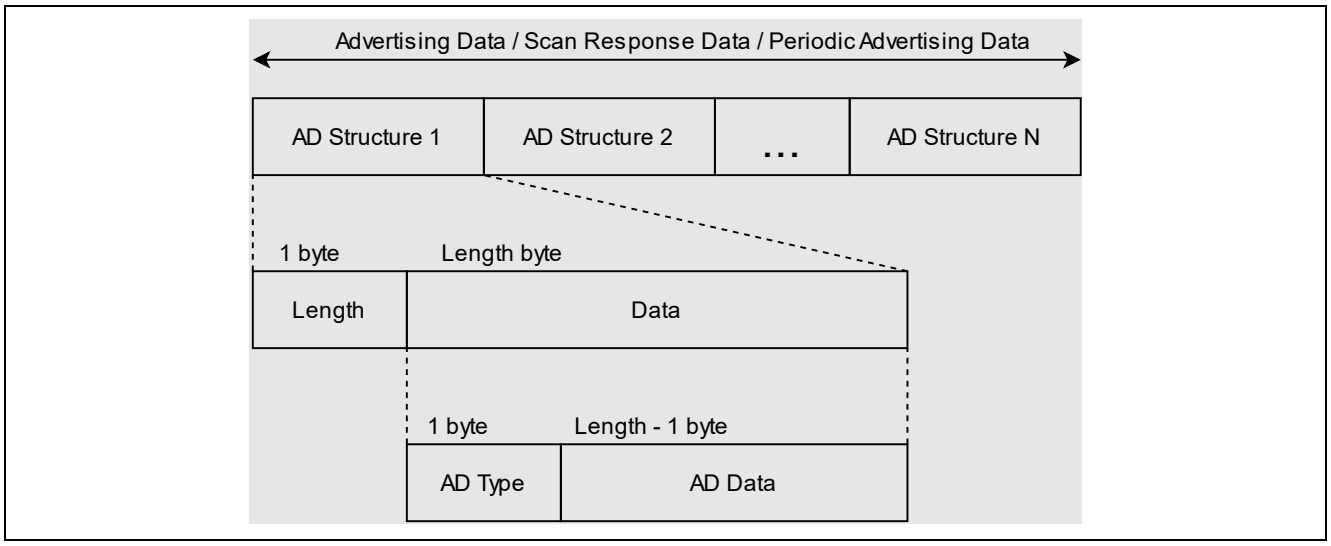


Figure 13. Advertising Data / Scan Response Data / Periodic Advertising Data format

Advertising Data / Scan Response Data / Periodic Advertising Data includes one or more AD Structures. Each AD Structure consists of Length, AD Type and AD Data. The Length is the sum of the size of AD Type (1 byte) and the size of the AD Data. Its unit is bytes. The value to be set in AD Type is defined by Bluetooth SIG in *Supplement to the Bluetooth Core Specification (CSS)*. Table 22 shows the AD Type that is often used.

Table 22. AD Type and AD Data

<table>
<tr><th colspan="2">Data Type</th><th>AD Type</th><th>Description</th></tr>
<tr><td colspan="2">Flags</td><td>0x01</td><td>Used for Connectable advertising.
The Flags value used for Bluetooth LE is as follows.
<table><tr><th>Octet</th><th>Bit</th><th>Description</th></tr><tr><td>0</td><td>0</td><td>LE Limited Discoverable Mode</td></tr><tr><td>0</td><td>1</td><td>LE General Discoverable Mode</td></tr><tr><td>0</td><td>2</td><td>BR/EDR Not Supported.</td></tr></table>A scanner is available Discoverable Mode for filtering by the mode.
If adding Discoverable Mode, select Limited or General.</td></tr>
<tr><td rowspan="6">Service UUID</td><td>Incomplete List of 16-bit Service UUIDs</td><td>0x02</td><td rowspan="6">UUID List.
The AD Type varies depending on the size.
If the AD Data includes all UUIDs, select Complete List.
If the AD Data include not all UUIDs, select Incomplete List.</td></tr>
<tr><td>Complete List of 16-bit Service UUIDs</td><td>0x03</td></tr>
<tr><td>Incomplete List of 32-bit Service UUIDs</td><td>0x04</td></tr>
<tr><td>Complete List of 32-bit Service UUIDs</td><td>0x05</td></tr>
<tr><td>Incomplete List of 128-bit Service UUIDs</td><td>0x06</td></tr>
<tr><td>Complete List of 128-bit Service UUIDs</td><td>0x07</td></tr>
<tr><td rowspan="2">Local Name</td><td>Shortened Local Name</td><td>0x08</td><td>Strings that show the first half of the device name.</td></tr>
<tr><td>Complete Local Name</td><td>0x09</td><td>Complete Device Name.</td></tr>
<tr><td colspan="2">Manufacturer Specific Data</td><td>0xFF</td><td>More than 2 bytes manufacturer specific data.
First 2 bytes shows the Company ID.
For details of the Company ID, refer to Assigned Number
(https://www.bluetooth.com/specifications/assigned-numbers/)</td></tr>
</table>

An example of setting the Advertising Data including Flags and Complete Local Name and the Scan Response Data including Complete Local Name is shown in Code 5.

```
/* Advertising Data */
uint8_t gs_adv_data[] =
{
    /* Flags */
    2,          /*  Data Size: 2byte */
    0x01,       /*  AD type: Flags */
    (BLE_GAP_AD_FLAGS_LE_GEN_DISC_MODE |
     BLE_GAP_AD_FLAGS_BR_EDR_NOT_SUPPORTED), /* Data */

    /* Complete Local Name */
    9,          /*  Data Size: 9byte */
    0x09,       /*  AD type: Complete Local Name */
    'R', 'B', 'L', 'E', '-', 'D', 'E', 'V',  /* Data */
};

/* Scan_Response Data */
uint8_t gs_sres_data[] =
{
    /* Complete Local Name */
    9,          /*  Data Size: 9byte */
    0x09,       /*  AD type: Complete Local Name */
    'R', 'B', 'L', 'E', '-', 'D', 'E', 'V', /* Data */

};
/* some code is omitted.  */

/* Advertising Data parameter */
st_ble_gap_adv_data_t adv_data_param = {
        .adv_hdl        = 0x00,
        .data_type      = BLE_GAP_ADV_DATA_MODE,
        .data_length    = ARRAY_SIZE(gs_adv_data),
        .p_data         = gs_adv_data  ,
};

/* Scan_Response Data parameter */
st_ble_gap_adv_data_t sres_data_param = {
        .adv_hdl        = 0x00,
        .data_type      = BLE_GAP_SCAN_RSP_DATA_MODE,
        .data_length    = ARRAY_SIZE(gs_sres_data),
        .p_data         = gs_sres_data,
};

/* some code is omitted. */

/* Set Advertising Data */
R_BLE_GAP_SetAdvSresData(&adv_data_param);

/* some code is omitted. */

/* GAP Callback  */
void gap_cb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
    switch(type)
    {
        /* some code is omitted.  */
        case BLE_GAP_EVENT_ADV_DATA_UPD_COMP :
        {
            st_ble_gap_adv_data_evt_t * p_adv_data_set_param;
            p_adv_data_set_param = (st_ble_gap_adv_data_evt_t *)p_data->p_param;
            if((0x00 == p_adv_data_set_param->adv_hdl) &&
               (BLE_GAP_ADV_DATA_MODE == p_adv_data_set_param->data_type))
            {
                R_BLE_GAP_SetAdvSresData(&sres_data_param);
            }
        } break;
        /* some code is omitted. */
    }
}
```

Code 5. Sample of setting Advertising Data and Scan Response Data

4.4.2 Advertising data update

Advertising data and scan response data can be dynamically updated while advertising if the conditions shown in Table 23 are met.

Table 23. Conditions for updating Advertising Data and Scan Response Data

Legacy or Extended	Condition
Legacy	Can be updated at any time
Extended advertising	Data to be updated is 251 bytes or less.

It is necessary to call *R_BLE_GAP_SetAdvSresData* API to update advertising data and scan response data. Example of the value for each field of *st_ble_gap_adv_data_t* structure for updating advertising data is shown in Code 6.

```
st_ble_gap_adv_data_t adv_data_param = {
    .adv_hdl        = “Advertising handle of the advertising data to be update”,
    .data_type      = BLE_GAP_ADV_DATA_MODE,
    .data_length    = “Size of the data to be updated”,
    .p_data         = “Pointer to the data to be updated”,
};
```

Code 6. Parameters for updating Advertising Data / Scan Response Data

If users want to update more than 252 bytes of data, stop advertising at once according to section 4.2.4 and use *R_BLE_GAP_SetAdvSresData* API to update the data.

4.4.3 Periodic Advertising Data Update

Periodic advertising data can be dynamically updated while periodic advertising is running if the conditions shown in Table 24 are met.

Table 24. Requirement for updating Periodic Advertising Data

Advertising type	Condition
Periodic Advertising	The data length is 252 bytes or less.

It is necessary to call *R_BLE_GAP_SetAdvSresData* API to update periodic advertising data. Example of the value for each field of *st_ble_gap_adv_data_t* structure for updating periodic advertising data is shown as following.

Set the parameters shown in Code 7 and call R_BLE_GAP_SetAdvSresData to update Periodic Advertising Data.

```
st_ble_gap_adv_data_t adv_data_param = {
    .adv_hdl        = “Advertising handle of the Periodic Advertising Data to be update”,
    .data_type      = BLE_GAP_PERD_ADV_DATA_MODE,
    .data_length    = “Size of the data to be updated”,
    .p_data         = “Pointer to the data to be updated”,
};
```

Code 7. Parameters for updating Periodic Advertising Data

If users want to update more than 253 bytes of data, stop periodic advertising at once according to section 4.3.5 and use *R_BLE_GAP_SetAdvSresData* API to update the data.

4.4.4 Total advertising data size

As shown in Table 16, extended advertising can be set Advertising Data or Scan Response Data up to the *Maximum advertising data length* configuration value. The size of buffer for Advertising Data and Scan Response Data in the BLE module is 4250 bytes. Therefore, the amount size of Advertising Data and Scan Response Data in all advertising sets must be 4250 bytes or less.

As shown in Table 20, periodic advertising can be set periodic advertising data up to the *Maximum advertising data length configuration* value. The size of buffer for Periodic Advertising Data in the BLE module is 4306 bytes. Therefore, the amount size of periodic advertising data in all advertising sets must be 4306 bytes or less.

Example of data size in each advertising set is shown in Figure 14 and Figure 15.

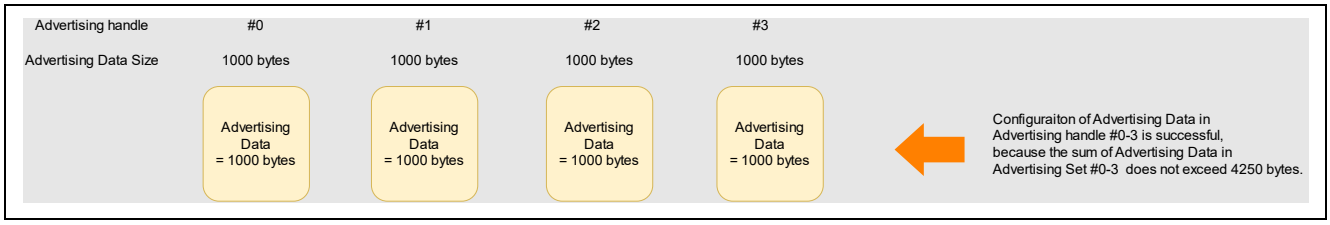


Figure 14. Example of setting advertising data (Successful case)

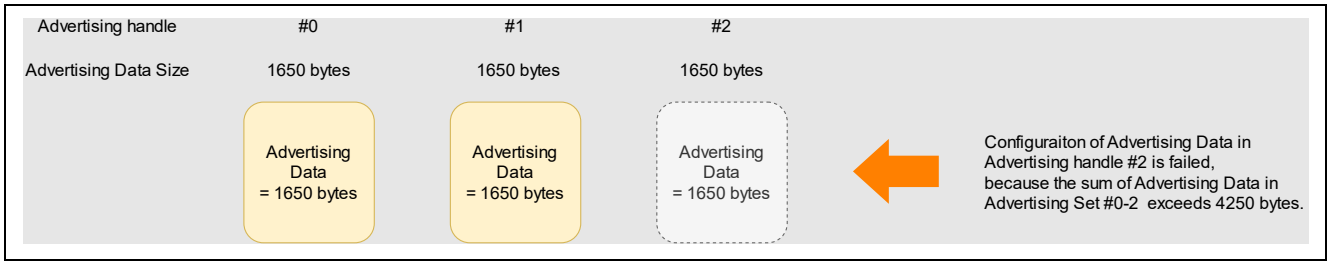


Figure 15. Example of setting advertising data (Failure case)

4.5 Typical use case for advertising

4.5.1 Connection with Smart Phone

An example of sending advertising packets to connect to a Smart Phone is shown in Code 8.

```
/* Advertising Data */
static uint8_t gs_adv_data[] =
{
    /* Flag (mandatory) */
    2,          /**< Data Size */
    0x01,       /**< Data Flag: Flag */
    (BLE_GAP_AD_FLAGS_LE_GEN_DISC_MODE | BLE_GAP_AD_FLAGS_BR_EDR_NOT_SUPPORTED),  /**< Data Value */

    /* Complete Local Name */
    9,         /**< Data Size */
    0x09,       /**< Data Flag: Complete Local Name */
    'R', 'B', 'L', 'E', '-', 'D', 'E', 'V', /**< Data Value */

};

/* Scan_Response Data */
static uint8_t gs_sres_data[] =
{
    /* Complete Local Name */
    9,         /**< Data Size */
    0x09,       /**< Data Flag: Complete Local Name */
    'R', 'B', 'L', 'E', '-', 'D', 'E', 'V', /**< Data Value */

};

/* Advertising parameters */
static ble_abs_legacy_adv_param_t gs_adv_param =
{
    .slow_adv_intv   = 0x00A0,
    .slow_period     = 0,
    .p_adv_data      = gs_adv_data,
    .adv_data_length = ARRAY_SIZE(gs_adv_data),
    .p_sres_data     = gs_sres_data,
    .sres_data_length = ARRAY_SIZE(gs_sres_data),
    .adv_ch_map      = BLE_GAP_ADV_CH_ALL,
    .filter          = BLE_ABS_ADV_ALLOW_CONN_ANY,
    .o_addr_type     = BLE_GAP_ADDR_PUBLIC,
    .o_addr          = {0},
};

/** some code is omitted  **/

/* Start Advertising */
RM_BLE_ABS_StartLegacyAdvertising(&g_ble_abs0_ctrl, &gs_adv_param);
```

Code 8. Sample of advertising for connecting with Smart Phone

When starting advertising, the user application will be notified in the *BLE_GAP_EVENT_ADV_ON* event. Smart Phones can detect the device to connect as “RBLE-DEV” after the event notification.

4.5.2 Beacon

When user want to broadcast iBeacon (Apple Inc) or Eddystone (Google), use non-connectable advertising.

An example of sending non-connectable advertising packets as beacon is shown in Code 9.

```
/* Advertising Data */
static uint8_t gs_adv_data[] =
{
    /* Flag */
    2,          /**< Data Size */
    0x01,       /**< Data Flag: Flag */
    BLE_GAP_AD_FLAGS_BR_EDR_NOT_SUPPORTED,    /**< Data Value */

    /* Complete Local Name */
    9,          /* Data Size */
    0x09,       /* Data Flag: Complete Local Name */
    'R', 'B', 'L', 'E', '-', 'D', 'E', 'V', /* Data */

};

/* Advertising parameters */
static ble_abs_non_conn_adv_param_t gs_non_conn_adv_param =
{
    .p_addr          = NULL,
    .p_adv_data      = gs_adv_data,
    .adv_intv        = 0x00A0,
    .duration        = 0,
    .adv_data_length = ARRAY_SIZE(gs_adv_data),
    .adv_ch_map      = BLE_GAP_ADV_CH_ALL,
    .o_addr_type     = BLE_GAP_ADDR_PUBLIC,
    .adv_phy         = BLE_GAP_ADV_PHY_1M,
    .sec_adv_phy     = BLE_GAP_ADV_PHY_1M,
    .o_addr          = {0},
};

/** some code is omitted  **/

/* Start Advertising */
RM_BLE_ABS_StartNonConnectableAdvertising(g_ble_abs0_ctrl, &gs_non_conn_adv_param);
```

Code 9. Sample of using RM_BLE_ABS_StartNonConnectableAdvertising

When starting advertising, user application will be notified in the *BLE_GAP_EVENT_ADV_ON* event. After the event notification, remote devices can detect the beacon as “RBLE-DEV” when performing scan

For more information about iBeacon and Eddystone, refer to the following.

iBeacon : https://developer.apple.com/ibeacon/

Eddystone : https://developers.google.com/beacons/eddystone

5. Scan

Bluetooth LE device receives advertising packets from other devices by scan. This chapter describes how to use the scan feature by using related APIs.

Users can use the following categories of API to perform the procedure shown in Figure 16.

- **Abstraction API (*RM_BLE_ABS_XXX* API)**
  - Users can use the scan feature with a single API call. However, detailed parameter settings are not possible.

- **GAP API (*R_BLE_GAP_XXX* API)**
  - Users use the scan feature by combining several APIs. However, detailed parameter settings are possible.

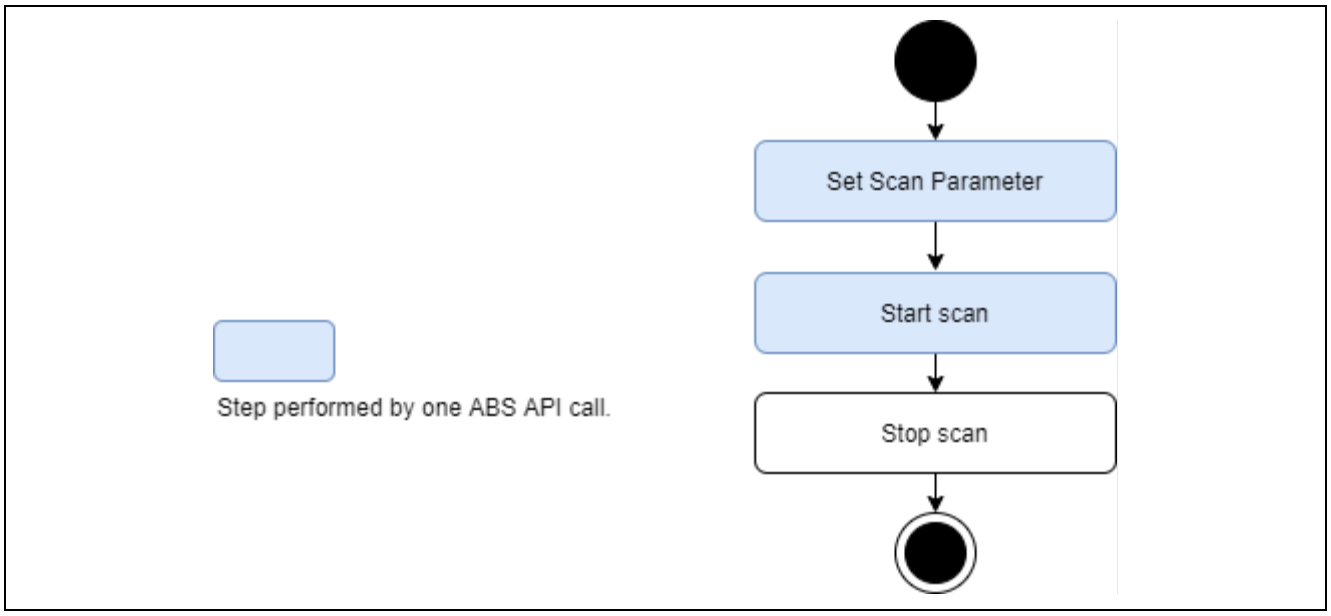


Figure 16. Scan procedure

5.1 Scan with abstraction API

When users use the abstraction API, the procedure from set scan parameter to start scan is performed by single abstraction API (*RM_BLE_ABS_Start_Scanning*) call. Refer to *Renesas Flexible Software Package User's Manual* about the usage of the API. The sample code for acquiring information obtained by scan is shown in section 5.2.4. Refer to section 5.2.3 about stop scan.

5.1.1 Scan filtering

Refer to section 5.3.

5.1.2 Privacy

Privacy is a feature that prevents other devices from tracking by periodically changing BD address. Scan abstraction API can use the privacy feature. Privacy feature can be used after preparing the local IRK for using privacy feature according to section 8.4 and set value of Table 25 to upper 4bits of *device_scan_filter_policy* field in the *ble_abs_scan_parameter_t* structure.

Table 25. Parameters used for the privacy feature

Field	Value	Description
device_scan_filter_policy (upper 4bits)	BLE_GAP_ADDR_RPA_ID_PUBLIC(0x02)	If the IRK of the local device has not been registered in Resolving List, the public address is used.
	BLE_GAP_ADDR_RPA_ID_RANDOM(0x03)	Resolvable Private Address. If the IRK of local device has not been registered in Resolving List, the Static Address set by *R_BLE_GAP_SetRandAddr()* or *R_BLE_VS_SetBdAddr()* is used.

BP: When supporting privacy, it is recommended to limit the use of scannable advertisements and active scanning to avoid the risk of devices being tracked.

5.2 Scan with GAP API

When the user uses the GAP API, the procedure from set scan parameters to start / stop scan is performed by combining several API calls. This section describes each procedure.

5.2.1 Set scan parameters

It is necessary to configure the *st_ble_gap_scan_param_t* and the *st_ble_gap_scan_on_t* structures and call *R_BLE_GAP_StartScan* API with these structures as arguments. Refer to *Renesas Flexible Software Package User's Manual* about the details of these structures and API. These structures include following fields which specify the interval and period of scan.

- *scan_intv*: Specify scan interval
- *scan_window*: Specify scan window
- *duration*: Specify scan duration
- *period*: Specify scan period
- *p_phy_param_1M / p_phy_param_coded*: Specify scan PHY

Figure 17 shows relationship of these parameters.

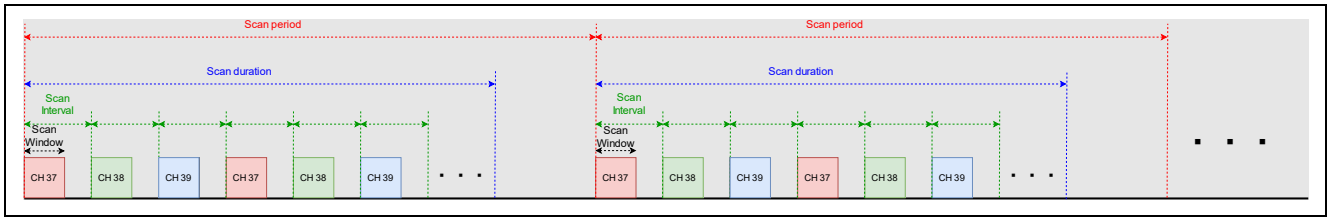


Figure 17. The relationship of scan interval, window, duration, period

These structures also include *fast_xxx* and *slow_xxx* fields. These fields specify the frequency of scan. Fast scan increases a detection probability of advertising PDUs from remote device and the slow scan decreases a detection probability of advertising PDUs from remote device. Figure 18 shows the relationship between the fast scan and slow scan.

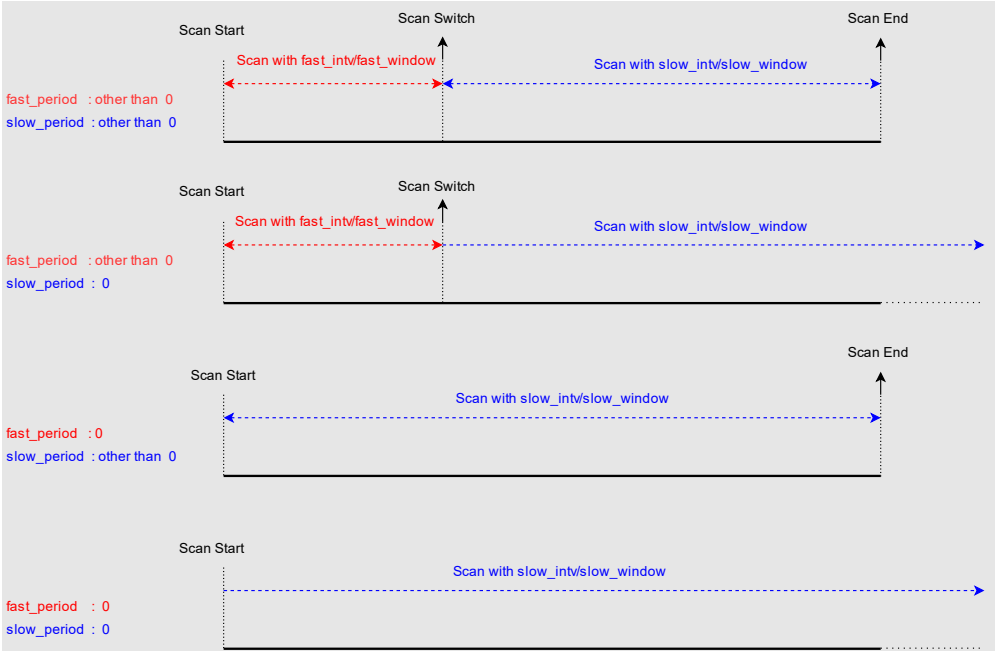


Figure 18. The relationship between the fast scan and slow scan

Table 26 shows the event regarding the fast scan and slow scan.

Table 26. The event regarding the fast scan and slow scan

Scan Start	Scan Fast/Slow switch	Scan End
BLE_GAP_EVENT_SCAN_ON	BLE_GAP_EVENT_SCAN_TO BLE_GAP_EVENT_SCAN_ON	BLE_GAP_EVENT_SCAN_TO
BLE_GAP_EVENT_SCAN_ON	BLE_GAP_EVENT_SCAN_OFF BLE_GAP_EVENT_SCAN_ON	BLE_GAP_EVENT_SCAN_OFF

The p_phy_param_1M and the p_phy_param_coded field specify the PHY of scan. Setting the p_phy_param_1M is required to receive Advertising that the primary channels (CH:37/38/39) use 1M PHY. Setting the p_phy_param_coded is required to receive Advertising that primary channels use Coded PHY.

5.2.1.1 Whitelist

Refer to section 5.3.

5.2.1.2 Privacy

Privacy is a feature that prevents other devices from tracking advertising packet by periodically changing BD address, which is a part of scan request packet. Privacy feature can use after preparing IRK for using privacy feature according to section 8.4 and set value shown in Table 27 and to *o_addr_type* field in *st_ble_gap_scan_param_t* structure. The default update interval for BD address is 900 seconds. The interval can be changed with *R_BLE_GAP_SetRpaTo*().

Table 27. The parameters used for the privacy feature

Field	Value	Description
o_addr_type	BLE_GAP_ADDR_RPA_ID_PUBLIC(0x02)	Resolvable Private Address. If the IRK of local device has not been registered in Resolving List, Public Address is used.
	BLE_GAP_ADDR_RPA_ID_RANDOM(0x03)	Resolvable Private Address. If the IRK of local device has not been registered in Resolving List, the Static Address set by *R_BLE_GAP_SetRandAddr()* or *R_BLE_VS_SetBdAddr()* is used.

5.2.2 Start scan

When starting a scan, call the *R_BLE_GAP_StartScan* API.

5.2.3 Stop scan

When stopping a scan, call the *R_BLE_GAP_StopScan* API. In addition to the API call, the conditions for stopping a scan are as follows.

- If the *period* field of *st_ble_gap_scan_on_t* structure which is argument of *R_BLE_GAP_StopScan* API is set to other than 0, the scan stops after the period is expired.

5.2.4 Received information by scan

After starting scan, the BLE Protocol Stack notifies the user application that an advertising packet is received from another device using the *BLE_GAP_EVENT_ADV_REPT_IND* event. If the advertiser uses AUX_CHAIN_IND, advertising data will be notified separately. Furthermore, since the size of Advertising Data that can be notified to upper layer according to Bluetooth specification is 229 byte or less, 230 byte or more Advertising Data will be notified separately. Combine them on the receiver if necessary.

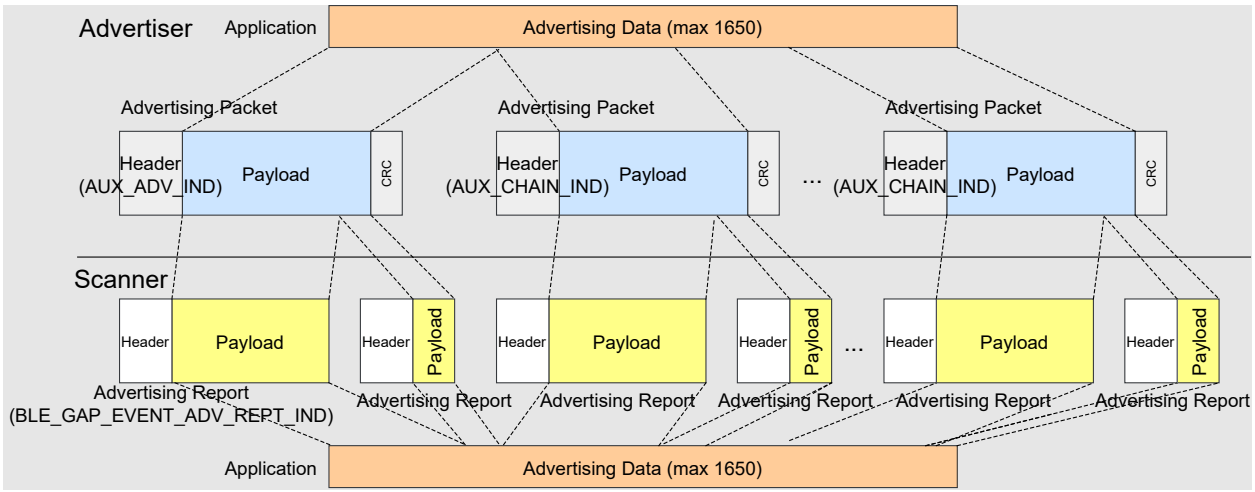


Figure 19. Dividing and combining Advertising Data

Received advertising packet is stored in a *st_ble_gap_adv_rept_evt_t* structure. Refer to *Renesas Flexible Software Package User's Manual* about the details of the structure. An example of displaying the RSSI of received advertising packet is shown in Code 10.

```
/* GAP callback function */
void gap_cb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
    int8_t rssi;
    switch (type)
    {
        /** some code is omitted  **/
        case BLE_GAP_EVENT_ADV_REPT_IND:
        {
            st_ble_gap_adv_rept_evt_t *adv_rept_evt_param =
                (st_ble_gap_adv_rept_evt_t *)data->p_param;

            switch (adv_rept_evt_param->adv_rpt_type)
            {
                /* receive legacy advertising PDU */
                case 0x00:
                {
                    st_ble_gap_adv_rept_t *adv_rept_param =
                        (st_ble_gap_adv_rept_t *)adv_rept_evt_param->param.p_adv_rpt;

                    /* Save RSSI */
                    Rssi = adv_rept_param->rssi;
                } break;

                /* receive extended advertising PDU */
                case 0x01:
                {
                    st_ble_gap_ext_adv_rept_t *ext_adv_rept_param =
                        (st_ble_gap_ext_adv_rept_t *)ext_adv_rept_param->
                                                    param.p_ext_adv_rpt;

                    /* Save RSSI */
                    rssi = ext_adv_rept_param->rssi;
                } break;
                /** some code is omitted  **/
```

Code 10. Sample of displaying the RSSI included in a received advertising packet

5.3 Scan filtering

Received advertising packets can be filtered by the following methods.

- Whitelist
- Duplicate advertising filtering
- Discoverable mode filtering
- Advertising data filtering

By using these methods, user can obtain essential advertising packets for their own application. Each method describes in following sections.

5.3.1 Whitelist

Whitelist is a feature that filters a specific BD address from the received wireless packet. Whitelist feature can be used by applying following steps.

1. Register a known device BD address to the whitelist by calling *R_BLE_GAP_ConfWhiteList* API.

Note: The Whitelist cannot be added/deleted when the Whitelist filter enabled operation (advertising, scanning, connection request) is executed.

2. Set to use whitelist feature for:
   - Set *BLE_GAP_SCAN_ALLOW_ADV_WLST* (0x01) to *filter_policy* field in,*st_ble_gap_scan_param_t* structure when performing scan with GAP API.
   - Set *BLE_GAP_SCAN_ALLOW_ADV_WLST* (0x01) to lower 4bits of *device_scan_filter_policy* field in *ble_abs_scan_parameter_t* structure when performing scan with abstraction API.

Refer to *Renesas Flexible Software Package User's Manual* about the other options of *filter_policy* and *device_scan_filter_policy* field.

5.3.2 Duplicate advertising filtering

Duplicate advertising filtering is a feature that avoid receiving duplicate advertising packet from same advertiser. Duplicate advertising filtering feature can be used by setting *BLE_GAP_SCAN_FILT_DUPLIC_ENABLE* (0x01) or *BLE_GAP_SCAN_FILT_DUPLIC_ENABLE_FOR_PERIOD* (0x02) to:

- *filter_duplicate* field in *ble_abs_scan_parameter_t* structure when performing scan with abstraction API.
- *filter_dups* field in *st_ble_gap_scan_on_t* structure when performing scan with GAP API.

Up to 8 types of advertising packets can be filtered with this feature. If there are more than 9 types of advertising packet around scan device, the 9th and subsequent advertising packet will not be filtered.

5.3.3 Discoverable mode filtering

Discoverable mode filtering is a feature to select advertising packet to be received according to *AD_TYPE* field contained in advertising data. Refer to section 4.4.1 about *AD_TYPE* field. This feature can be used by setting value shown in Table 28 to *proc_type* field in *st_ble_gap_scan_on_t* structure when performing scan with GAP API. Abstraction API does not support this feature.

Table 28. The value to be set for filtering with Discoverable Mode

Macro	Description
BLE_GAP_SC_PROC_OBS(0x00)	Observation Procedure. Notify all advertising PDUs.
BLE_GAP_SC_PROC_LIM(0x01)	Limited Discovery Procedure. Notify advertising PDUs from only devices in the limited discoverable mode.
BLE_GAP_SC_PROC_GEN(0x02)	General Discovery Procedure. Notify advertising PDUs from devices in the limited discoverable mode and the general discoverable mode.

5.3.4 Advertising data filtering

The Abstraction API can filter by referring advertising data. GAP API does not support this feature. Specify the data for filtering to the following parameters in the *ble_abs_scan_parameter_t* structure.

- *p_filter_data*: The filtered data.
- *filter_data_length*: The filtered data size.
- *filter_ad_type*: The AD_TYPE of the filtered data.

Example configuration to *ble_abs_scan_parameter_t* structure is shown in Code 11.

```
/* Scan filter data */
static uint8_t gs_filter_data[] =
{
    /* Complete Local Name */
    9,          /**< Data Size */
    0x09,        /**< Data Type: Complete Local Name */
    'R', 'B', 'L', 'E', '-', 'D', 'E', 'V', /**< Data Value */
};

/* Scan parameters */
static ble_abs_scan_parameter_t gs_scan_param =
{
    .phy_parameter_1M           = &gs_scan_phy_param,
    .p_filter_data              = gs_filter_data,
    .slow_scan_period           = 0,
    .filter_data_length         = ARRAY_SIZE(gs_filter_data),
    .device_scan_filter_policy  = BLE_GAP_SCAN_ALLOW_ADV_ALL,
    .filter_duplicate           = BLE_GAP_SCAN_FILT_DUPLIC_ENABLE,
};
```

Code 11. Sample of advertising data filtering

5.4 Periodic advertising synchronization with GAP API

A scanner can establish a Periodic Advertising Synchronization (Sync) with an advertiser which broadcasts periodic advertising packets. Figure 20 shows the procedure that a scanner establishes a Periodic Advertising Sync in application. The following sections describe the details of Periodic Advertising synchronization procedure.

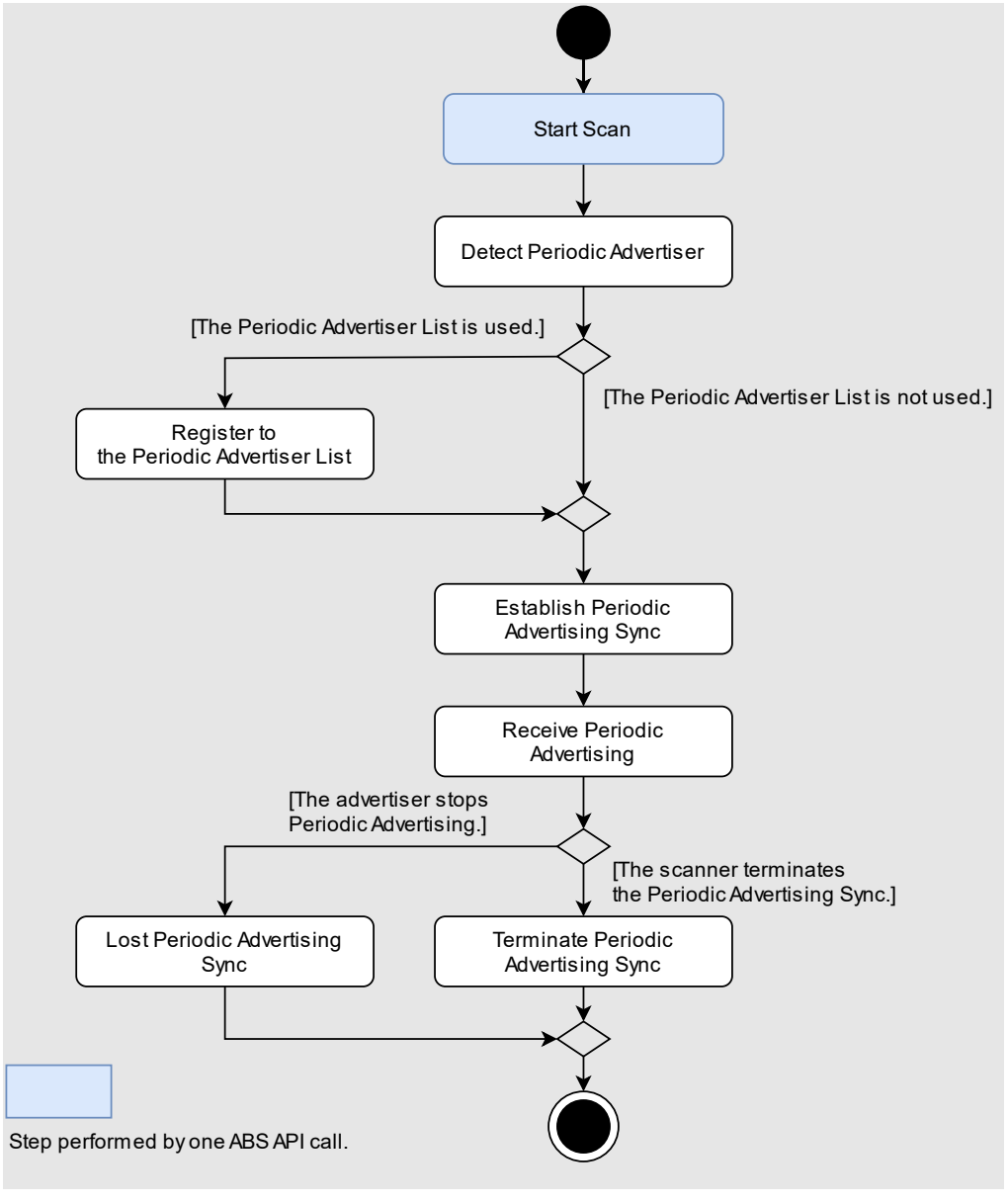


Figure 20. Periodic advertising sync procedure

User can use only GAP API for periodic advertising synchronization and this feature is only available in “Extended” configuration, which is BLE module configuration option.

5.4.1 Start scan

Refer to section 5.2.2.

5.4.2 Detect periodic advertiser

After calling the *R_BLE_GAP_StartScan* API, the BLE module notifies user application that an advertising packet is received from another device by *BLE_GAP_EVENT_ADV_REPT_IND* event. The scanner can establish a periodic advertising synchronization with the advertiser if *perd_adv_intv* field of *st_ble_gap_adv_rept_evt_t* structure included in a received advertising packet is not zero.

5.4.3 Establish periodic advertising synchronization

Call *R_BLE_GAP_CreateSync* API to establish a Periodic Advertising synchronization. Users can specify the advertiser to establish synchronization by setting argument of *R_BLE_GAP_CreateSync* API or by using periodic advertiser list feature which is described in section 5.4.4. The synchronization can be established up to the value specified in *Maximum periodic sync set number* option. Refer to *BLE sample application (R01AN5402)* Chapter 4 about the configuration option. To cancel establishing a Periodic Advertising Sync after calling *R_BLE_GAP_CreateSync* API, call *R_BLE_GAP_CancelCreateSync* API. When the cancellation has been completed, user application receives BLE *BLE_GAP_EVENT_SYNC_EST* event and the result is *BLE_ERR_NOT_YET_READY*(0x0012). An example of from starting scan to establishing a Periodic Advertising synchronization is shown in Code 12.

```
/** some code is omitted  **/

static st_ble_dev_addr_t gs_sync_advr;
static uint8_t gs_adv_sid;

static ble_abs_scan_phy_param_t gs_phy_param_1M =
{
    .fast_scan_interval  = 0x0200,
    .slow_scan_interval  = 0x0800,
    .fast_scan_window    = 0x0100,
    .slow_scan_window    = 0x0100,
    .scan_type           = BLE_GAP_SCAN_PASSIVE,
};

static ble_abs_scan_parameter_t gs_scan_param =
{
    .p_phy_parameter_1M          = &gs_phy_param_1M,
    .p_phy_parameter_coded       = NULL,
    .p_filter_data               = NULL,
    .fast_scan_period            = 0x0100,
    .slow_scan_period            = 0x0000,
    .filter_data_length          = 0,
    .device_scan_filter_policy   = BLE_GAP_SCAN_ALLOW_ADV_ALL,
    .filter_duplicate            = BLE_GAP_SCAN_FILT_DUPLIC_DISABLE,
};

static void gap_cb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
    /** some code is omitted  **/
    switch(type)
    {
        case BLE_GAP_EVENT_STACK_ON:
        {
            RM_BLE_ABS_StartScanning(&g_ble_abs0_ctrl, &gs_scan_param);
        } break;

        case BLE_GAP_EVENT_ADV_REPT_IND:
        {
            st_ble_gap_adv_rept_evt_t * p_adv_rept_evt_param =
                (st_ble_gap_adv_rept_evt_t *)p_data->p_param;

            switch (p_adv_rept_evt_param->adv_rpt_type)
            {
                case 0x01:
                {
                    st_ble_gap_ext_adv_rept_t * p_ext_adv_rept_param =
                        (st_ble_gap_ext_adv_rept_t *)p_adv_rept_evt_param->param.p_ext_adv_rpt;
```

```
                    if(0x0000 != p_ext_adv_rept_param->perd_adv_intv)
                    {
                        /* found */
                        memcpy(gs_sync_advr.addr, p_ext_adv_rept_param->p_addr,
                               BLE_BD_ADDR_LEN);
                        gs_sync_advr.type = p_ext_adv_rept_param->addr_type;
                        gs_adv_sid = p_ext_adv_rept_param->adv_sid;
                        R_BLE_GAP_ConfPerdAdvList(BLE_GAP_LIST_ADD_DEV,
                                                  &gs_sync_advr,
                                                  &gs_adv_sid,
                                                  1);
                    }

                } break;
                /** some code is omitted  **/
            }
        } break;

        case BLE_GAP_EVENT_PERD_LIST_CONF_COMP:
        {
            R_BLE_GAP_CreateSync(NULL, 0, 100, 100);
        } break;

        case BLE_GAP_EVENT_SYNC_EST:
        {
        } break;
        /** some code is omitted  **/
    }
}

/** some code is omitted  **/
```

Code 12. Sample of establishing a Periodic Advertising Sync

5.4.4 Periodic advertiser list

It is possible to register the BD address of a known advertiser to the Periodic Advertiser List by calling *R_BLE_GAP_ConfPerdAdvList* API. Code 12 contains an example of the API usage.

5.4.5 Receive periodic advertising PDUs

After the periodic advertising synchronization has been established with the advertiser, user application is notified by the *BLE_GAP_EVENT_ADV_REPT_IND* event that a periodic advertising packet is received. A received periodic advertising packet is stored in a *st_ble_gap_adv_rept_evt_t* structure. Refer to *Renesas Flexible Software Package User's Manual* about the details of the structure.

5.4.6 Lost periodic advertising sync

When the advertiser stops Periodic Advertising, user application is notified through the *BLE_GAP_EVENT_SYNC_LOST* event.

5.4.7 Terminate periodic advertising sync

By calling *BLE_GAP_TerminateSync* API, the scanner terminates the Periodic Advertising Sync. When the Periodic Advertising Sync has been terminated, user application is notified through the *BLE_GAP_EVENT_SYNC_TERM* event.

6. Connection

Bluetooth LE devices can communicate bi-directionally by establishing a connection between BLE devices. This chapter describes how to use the connection feature by using related APIs. Users can use following categories of API to perform above procedure.

Central device sends a connection request to Peripheral device running Connectable Advertising.by the below APIs. Peripheral device that receives a connection request terminates Connectable Advertising and responds to the connection request.

- **Abstraction API (*RM_BLE_ABS_XXX* API)**
  - Users can use the connection feature with a single API call. However, detailed parameter settings are not possible.

- **GAP API (*R_BLE_GAP_XXX* API)**
  - Users can use the connection feature by combining several APIs. However, detailed parameter settings are possible.

Setting the following is required to connect with the remote device that the primary channels (CH:37/38/39) of Advertising use 1M PHY.
The p_conn_param_1M field in st_ble_gap_create_conn_param_t structure used by R_BLE_GAP_CreateConn.
The p_conn_1M field in st_ble_abs_conn_param_t structure used by the R_BLE_ABS_CreateConn.

Setting the following is required to connect with the remote device that primary channels of Advertising use Coded PHY.
The p_conn_param_coded field in st_ble_gap_create_conn_param_t structure used by R_BLE_GAP_CreateConn.
The p_conn_coded field in st_ble_abs_conn_param_t structure used by the R_BLE_ABS_CreateConn.

The PHY after establishing a connection will be the PHY specified in the secondary channel (other than CH:37/38/39) of Advertising.

When the connection is established, the connection handle is notified in the BLE_GAP_EVENT_CONN_IND event. Since the connection handle is assigned a number that does not overlap with other connections in the range from 0 to BLE_ABS_CFG_RF_CONNECTION_MAXIMUM-1 in the lower 3 bits, it can be masked as shown in Code 13 below and used as the index of the management array of the connection handle.

```
#define BLE_APP_CONN_HDL_MASK               (0x0007)
uint16_t g_conn_hdl[BLE_ABS_CFG_RF_CONNECTION_MAXIMUM];
void gap_cb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
/** some code is omitted  **/
        case BLE_GAP_EVENT_CONN_IND:
        {
            if (BLE_SUCCESS == result)
            {
                /* Store connection handle */
                st_ble_gap_conn_evt_t *p_gap_conn_evt_param = (st_ble_gap_conn_evt_t *)p_data->p_param;
                uint16_t index = p_gap_conn_evt_param->conn_hdl & BLE_APP_CONN_HDL_MASK;
                g_conn_hdl[index] = p_gap_conn_evt_param->conn_hdl;
```

Code 13. Example of masking connection handle

6.1 Requesting connection with abstraction API

When user uses an abstraction API, call *RM_BLE_ABS_Create_Connection* API for requesting a connection with advertiser. Refer to *Renesas Flexible Software Package User's Manual* about usage of the API.

6.1.1 Whitelist filtering

Refer to section 6.4.

6.1.2 Privacy

Privacy is a feature that prevents other devices from tracking by periodically changing BD address. Connection abstraction API can use this privacy feature. Privacy feature can be used after preparing IRK for using privacy feature according to section 8.4 and set value of Table 29 to upper 4bits of *filter_parameter* field in *ble_abs_connection_parameter_t* structure.

Table 29. Parameters used for the privacy feature

Field	Value	Description
filter_parameter (upper 4bits)	BLE_GAP_ADDR_RPA_ID_PUBLIC(0x02)	If the IRK of local device has not been registered in Resolving List, public address is used.
	BLE_GAP_ADDR_RPA_ID_RANDOM(0x03)	Resolvable Private Address. If the IRK of local device has not been registered in Resolving List, the Static Address set by *R_BLE_GAP_SetRandAddr()* or *R_BLE_VS_SetBdAddr()* is used.

6.2 Requesting connection with GAP API

When a user uses the GAP API, call the *R_BLE_GAP_CreateConn* API for requesting connection with the advertiser. Refer to *Renesas Flexible Software Package User's Manual* about usage of the API.

6.3 Cancelling Connection Request

A connection request cannot be sent until the connection is established by a previous connection request or until the connection request is cancelled. After sending a connection request, if a user wants to send another connection request, it is possible to cancel the previous connection request by calling *BLE_GAP_CancelCreateConn* API. This API can be used in any case when requesting a connection using abstraction API and GAP API. After cancelling the connection request, user application is notified by *BLE_GAP_EVENT_CONN_IND* event that the result is *BLE_ERR_INVALID_HDL*(0x000E).

6.4 Whitelist filtering

When users want to reconnect to a known device, it is possible to use whitelist feature by applying the following procedures.

1. Register the BD address of the remote device to whitelist by calling *R_BLE_GAP_ConfWhiteList* API.

Note: The Whitelist cannot be added/deleted when the Whitelist filter enabled operation (advertising, scanning, connection request) is executed.

2. Set to use whitelist feature for:
   - Set *BLE_GAP_INIT_FILT_USE_WLST*(0x01) to lower 4bits of *filter_parameter* field in *ble_abs_connection_parameter_t* structure when use *RM_BLE_ABS_CreateConnection* API.
   - Set *BLE_GAP_INIT_FILT_USE_WLST*(0x01) to *init_filter_policy* filed in *st_ble_gap_create_conn_param_t* structure when use *R_BLE_GAP_CreateConn* API.

An example of connecting a remote device registered in the Whitelist is shown in Code 14.

```
/* remote device address */
dev.addr = {"Remote device BD_ADDR" };
dev.type = BLE_GAP_ADDR_PUBLIC;

/* register remote device to white list */
R_BLE_GAP_ConfWhiteList(BLE_GAP_LIST_ADD_DEV, &dev, 1);

/** some code is omitted  **/

/* reconnect */
st_ble_gap_conn_param_t conn_1M = {
    .conn_intv_min  = 0x0100,
    .conn_intv_max  = 0x0100,
    .conn_latency   = 0x0000,
    .sup_to         = 0x03BB,
    .min_ce_length  = 0xFFFF,
    .max_ce_length  = 0xFFFF,
};

st_ble_gap_create_conn_param_t conn_param;
conn_param.init_filter_policy = BLE_GAP_INIT_FILT_USE_WLST;
conn_param.own_addr_type = BLE_GAP_ADDR_PUBLIC;

/*  set connection parameters for 1M */
st_ble_gap_conn_phy_param_t conn_phy_1M = {
    .scan_intv   = 0x0300,
    .scan_window = 0x0300,
    p_conn_param = &conn_1M,
};
conn_param.p_conn_param_1M = &conn_phy_1M;
R_BLE_GAP_CreateConn(&conn_param);
```

```
/** some code is omitted  **/
```

Code 14. Connection Request using the Whitelist

6.5 Privacy

Privacy feature can use after preparing local IRK for using privacy feature according to section 8.4 and set value shown in Table 30 to *st_ble_gap_create_conn_param_t* structure.

Table 30. The parameters used for the privacy feature

Field	Value	Description
own_addr_type	BLE_GAP_ADDR_RPA_ID_PUBLIC(0x02)	If the IRK of local device has not been registered in Resolving List, public address is used.
	BLE_GAP_ADDR_RPA_ID_RANDOM(0x03)	Resolvable Private Address. If the IRK of local device has not been registered in Resolving List, the Static Address set by *R_BLE_GAP_SetRandAddr()* or *R_BLE_VS_SetBdAddr()* is used.
remote_bd_addr	Specify the remote device address registered by R_BLE_GAP_ConfRslvList.	—

6.6 Multiple connection

This section describes how to connect to multiple devices at the same time. With BLE module, up to 7 devices can be connected simultaneously. The connection procedure is the same as for one-to-one communication which describes in previous section. Following are the notes on creating a BLE application that performs multiple connection.

1. Connection handle

Connection handle specifies connection with remote device. User application is notified of the connection handle when establish connection. The connection handle allocated for connection, not device. Therefore, connection handle will change even when reconnecting same remote device.

2. Attribute handle

Attribute handle is used to access GATT database in remote device. It is necessary to keep the attribute handle for each remote device when BLE application perform GATT client role. By using Profile common which include QE for BLE, it can hold the attribute handles for each remote device up to 10.

3. Characteristics value

In the use case where the GATT server role accepts connections from multiple clients, there are some characteristic values that the server must be retained for each remote device, such as Client Configuration Characteristics Descriptor.

Implementation examples of application code that connects multiple devices for some typical use cases are explained in following sections.

6.6.1 Connecting to multiple peripheral devices

This section describes implementation example when local device performs GAP central and communicate with multiple GAP peripheral devices, as shown in Figure 21. This is a typical case when collecting data from multiple sensors which perform GAP peripheral.

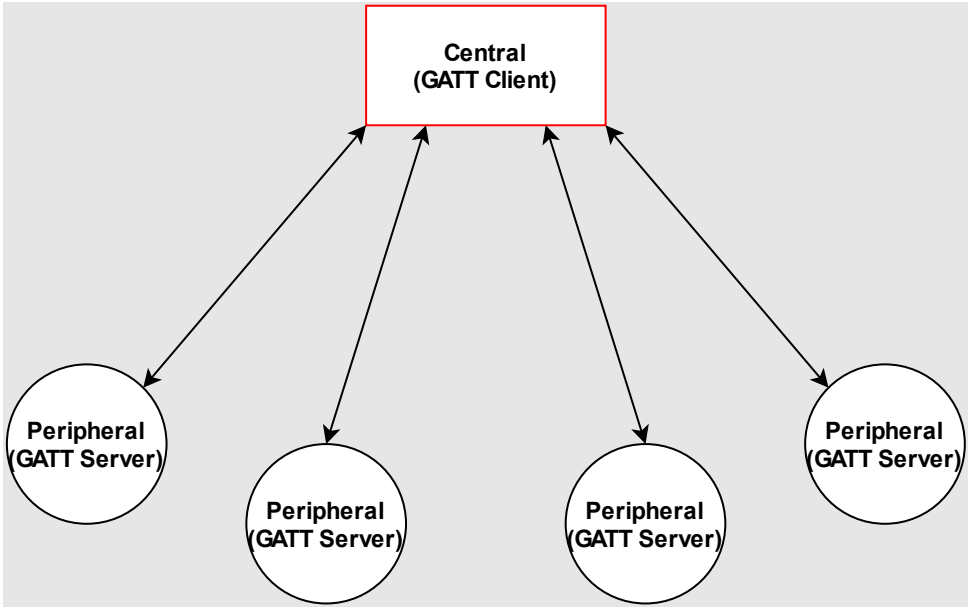


Figure 21. Connection with multiple peripheral devices

Sequence chart of implementation example is shown in Figure 22.

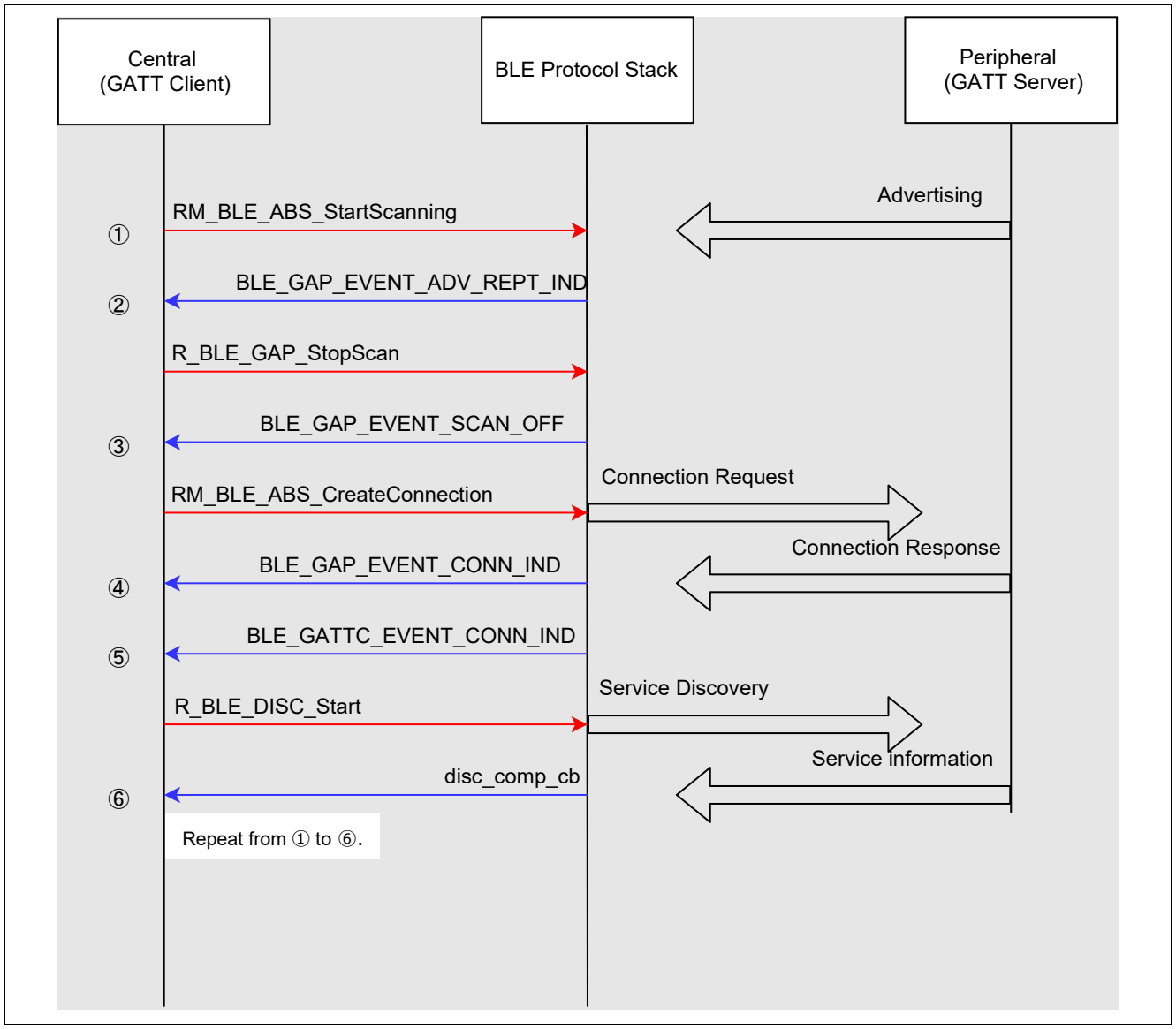


Figure 22. Sequence chart when connecting to a peripheral device

In this implementation, central device performs connection procedure after completing service discovery to ensure connection establishment one by one. The circled numbers in the sequence chart correspond to the numbers “(x)” in the example codes shown in Code 15, Code 16 and Code 17.

```
/* Scan phy parameters */
static ble_abs_scan_phy_paramter_t gs_scan_phy_param =
{
    /* TODO: Modify scan phy parameter. */
    .fast_scab_interval  = 0x200,
    .fast_scan_window    = 0x100,
    .slow_scan_interval  = 0x200,
    .slow_scan_window    = 0x100,
    .scan_type           = BLE_GAP_SCAN_PASSIVE,
};

/* Scan filter data */
static uint8_t gs_filter_data[] =
{
    /* TODO: Modify filter of advertise data. Value of Data Flag is defined in
       https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile */

    /* Complete Local Name */
    9,          /**< Data Size */
    0x09,       /**< Data Type: Complete Local Name */
    'R', 'B', 'L', 'E', '-', 'D', 'E', 'V', /**< Data Value */
};

/* Scan parameters */
static ble_abs_scan_parameter_t gs_scan_param =
{
    /* TODO: Modify scan parameter. */
    .p_phy_parameter_1M          = &gs_scan_phy_param,
    .p_filter_data               = gs_filter_data,
    .slow_scan_period            = 0,
    .filter_data_length          = ARRAY_SIZE(gs_filter_data),
    .device_scan_filter_policy   = BLE_GAP_SCAN_ALLOW_ADV_ALL,
    .filter_duplicate            = BLE_GAP_SCAN_FILT_DUPLIC_ENABLE,
};

/* Connection phy parameters */
static ble_abs_connection_phy_parameter_t gs_conn_phy_param =
{
    /* TODO: Modify connection phy parameter. */
    .connection_intverval        = 0x0130,
    .connection_slave_latency    = 0x0000,
    .supervision_timeout         = 0x03BB,
};

/*  Connection device address */
static st_ble_dev_addr_t gs_conn_bd_addr;

/* Connection parameters */
static ble_abs_conn_parameter_t gs_conn_param =
{
    .p_connection_phy_parameter_1M = &gs_conn_phy_param,
    .p_device_address    = &gs_conn_bd_addr,     /**< Set BD address of connecting device. */
    .filter_parameter    = BLE_GAP_INIT_FILT_USE_ADDR,
    .connection_timeout  = 5,
};
```

Code 15. Setting initial values for scan parameters and connection parameters

```
/* Connection handle */
uint16_t g_conn_hdl[BLE_ABS_CFG_RF_CONNECTION_MAXIMUM];
static void gap_cb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
    switch (type)
    {
        case BLE_GAP_EVENT_STACK_ON: /* (1) */
        {
            RM_BLE_ABS_StartScanning(&g_ble_abs0_ctrl, &gs_scan_param);
        } break;

        case BLE_GAP_EVENT_CONN_IND: /* (4) */
        {

            if (BLE_SUCCESS == result)
            {
                st_ble_gap_conn_evt_t *p_gap_conn_evt_param =
                    (st_ble_gap_conn_evt_t *)p_data->p_param;

                for(uint8_t i=0;i< BLE_ABS_CFG_RF_CONNECTION_MAXIMUM;i++)
                {
                 if(g_conn_hdl[i] == BLE_GAP_INVALID_CONN_HDL)
                 {
                         g_conn_hdl[i] = p_gap_conn_evt_param->conn_hdl;
                 }
                }

            }
        } break;

        case BLE_GAP_EVENT_DISCONN_IND:
        {
        st_ble_gap_disconn_evt_t *p_gap_disconn_evt_param =
                         (st_ble_gap_disconn_evt_t*)p_data->p_param;


        for(uint8_t i=0;i< BLE_ABS_CFG_RF_CONNECTION_MAXIMUM;i++)
        {
                 if(g_conn_hdl[i] == p_gap_disconn_evt_param->conn_hdl)
                 {
                         g_conn_hdl[i] = BLE_GAP_INVALID_CONN_HDL;
                 }
        }
        } break;

        case BLE_GAP_EVENT_ADV_REPT_IND: /* (2) */
        {
            st_ble_gap_adv_rept_evt_t *p_adv_rept_param = (st_ble_gap_adv_rept_evt_t *)p_data->p_param;
            st_ble_gap_ext_adv_rept_t *p_ext_adv_rept_param =
               (st_ble_gap_ext_adv_rept_t *)p_adv_rept_param->param.p_ext_adv_rpt;

            gs_conn_param.p_addr->type = p_ext_adv_rept_param->addr_type;
            memcpy(gs_conn_param.p_addr->addr, p_ext_adv_rept_param->p_addr, BLE_BD_ADDR_LEN)

            R_BLE_GAP_StopScan();
        } break;

        case BLE_GAP_EVENT_SCAN_OFF: /* (3) */
        {
            RM_BLE_ABS_CreateConnection(&g_ble_abs0_ctrl, &gs_conn_param);
        }
        default:
        {
            /* Do nothing. */
        } break;
    }
}
```

Code 16. Implementation example of GAP callback function when connecting multiple units

```
/* XXX Service UUID */
static uint8_t XXXC_UUID[] =
      { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

/* Service discovery parameters */
static st_ble_disc_entry_t gs_disc_entries[] = {

    {
        .p_uuid     = XXXC_UUID,
        .uuid_type  = BLE_GATT_128_BIT_UUID_FORMAT,
        .serv_cb    = R_BLE_XXXC_ServDiscCb,
    },
};

static void disc_comp_cb(uint16_t conn_hdl)
{
    /* TODO: Add function after discovery completed */
    RM_BLE_ABS_StartScanning(&g_ble_abs0_ctrl, &gs_scan_param); /* (6) */
    return;
}

static void gattc_cb(uint16_t type, ble_status_t result, st_ble_gattc_evt_data_t *p_data)
{
    R_BLE_SERVC_GattcCb(type, result, p_data);

    switch(type)
    {
        /* TODO: Set callback events of GATTC. Check BLE API reference for events. */

        case BLE_GATTC_EVENT_CONN_IND: /* (5) */
        {
            R_BLE_DISC_Start(p_data->conn_hdl, gs_disc_entries, ARRAY_SIZE(gs_disc_entries), disc_comp_cb);
        } break;

        default:
        {
            /* Do nothing. */
        } break;
    }
}
```

Code 17. Implementation example of service discovery using Profile Common Library

As shown in bold frame in Code 17, when user registers *R_BLE_XXX_ServDiscCb* which generated by QE for BLE, attribute handle of each peripheral device will be retained in the service API. The user application can access the GATT database of each peripheral device using the connection handle without managing the attribute handle of each peripheral device.

6.6.2 Connection to multiple central devices

This section describes implementation example when local device performs GAP peripheral and communicate with multiple GAP central devices, as shown in Figure 23. This is a one of typical case when home appliance equipment accepts control from multiple smart phones which perform GAP central.

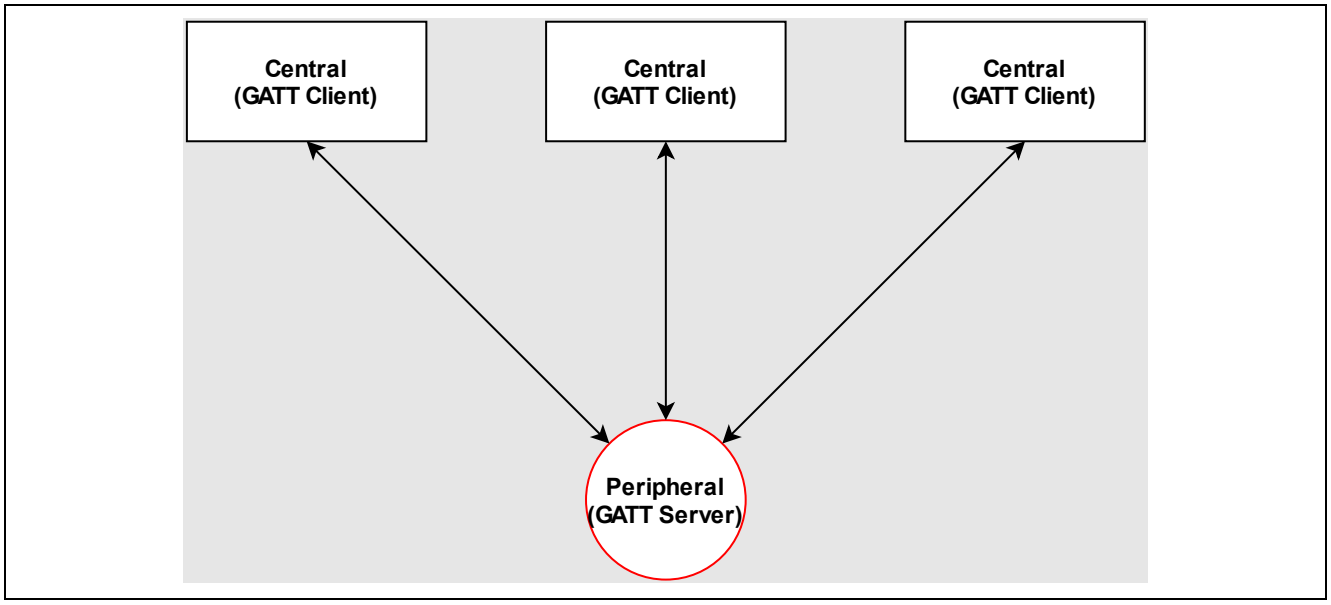


Figure 23. Connection with multiple central devices

Sequence chart of the implementation example is shown in Figure 24.

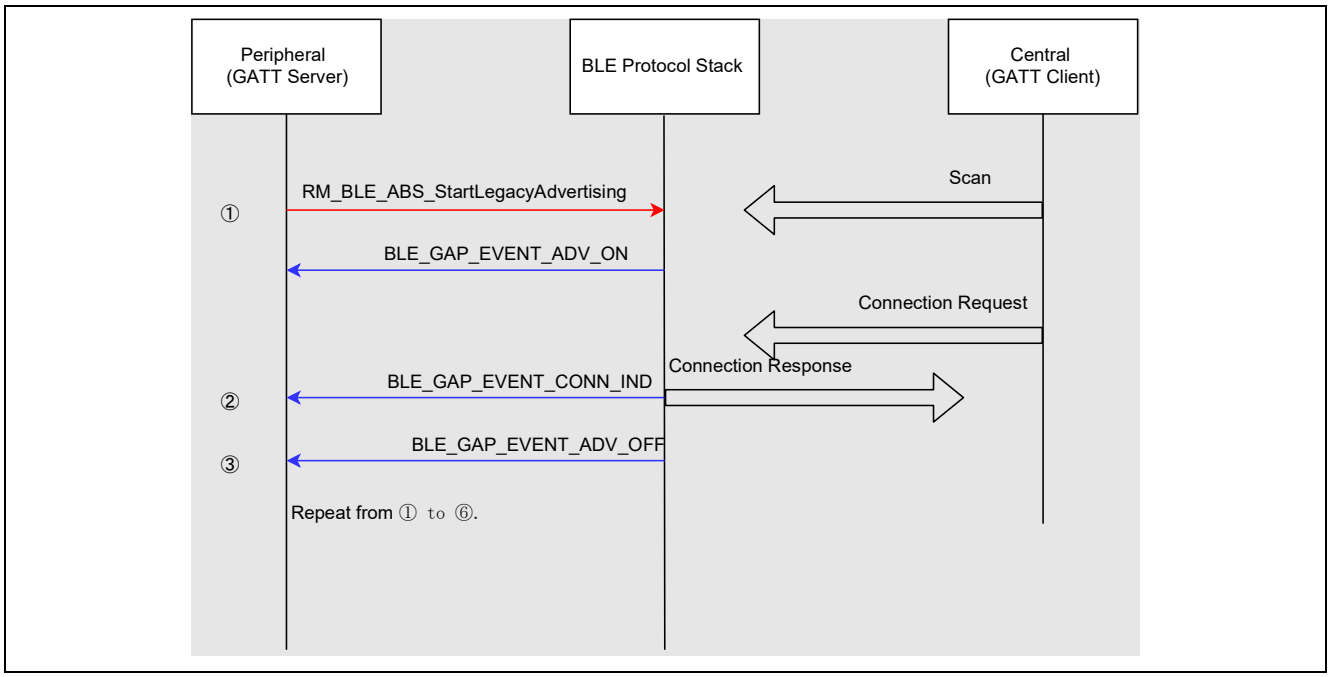


Figure 24. Sequence chart when connecting to a central device

Advertising will stop when establish connection with central device. Therefore, it is necessary to start advertising again to perform multiple connection with central devices. The circled numbers in the sequence chart correspond to the numbers “(x)” in the example codes shown in Code 18 and Code 19.

```
/* Advertising data */
static uint8_t gs_adv_data[] =
{
    /* TODO: Modify advertise data. Value of Data Flag is defined in
     https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile */

    /* Flag (mandatory) */
    2,          /**< Data Size */
    0x01,       /**< Data Type: Flag */
    (BLE_GAP_AD_FLAGS_LE_GEN_DISC_MODE | BLE_GAP_AD_FLAGS_BR_EDR_NOT_SUPPORTED),    /**< Data Value */

    /* Complete Local Name */
    9,         /**< Data Size */
    0x09,       /**< Data Type: Complete Local Name */
    'R', 'B', 'L', 'E', '-', 'D', 'E', 'V', /**< Data Value */
};

/* Scan response Data */
static uint8_t gs_sres_data[] =
{
    /* TODO: Modify scan response data. Value of Data Flag is defined in
       https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile */

    /* Complete Local Name */
    9,         /**< Data Size */
    0x09,       /**< Data Type: Complete Local Name */
    'R', 'B', 'L', 'E', '-', 'D', 'E', 'V', /**< Data Value */
};

/* Advertising parameters */
static ble_abs_legacy_advertising_parameter_t gs_adv_param =
{
    /* TODO: Modify advertise parameters. */
    .slow_advertising_interval   = 0x300,
    .slow_advertising_period     = 0,
    .p_advertising_data          = gs_adv_data,
    .advertising_data_length     = ARRAY_SIZE(gs_adv_data),
    .p_scan_response_data        = gs_sres_data,
    .scan_response_data_length   = ARRAY_SIZE(gs_sres_data),
    .advertising_channel_map     = BLE_GAP_ADV_CH_ALL,
    .advertising_filter_policy   = BLE_ABS_ADVERTISING_FILTER_ALLOW_ANY,
    .own_bluetooth_address_type  = BLE_GAP_ADDR_PUBLIC,
};
```

Code 18. Advertise packet and parameter settings

```
uint16_t g_conn_hdl[BLE_ABS_CFG_RF_CONNECTION_MAXIMUM];

static void gap_cb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
    switch (type)
    {
        case BLE_GAP_EVENT_STACK_ON:
        {
            RM_BLE_ABS_StartLegacyAdvertising(&g_ble_abs0_ctrl, &gs_adv_param);
        } break;

        case BLE_GAP_EVENT_CONN_IND:
        {

            if (BLE_SUCCESS == result)
            {
                st_ble_gap_conn_evt_t *p_gap_conn_evt_param =
                    (st_ble_gap_conn_evt_t *)p_data->p_param;
                RM_BLE_ABS_StartLegacyAdvertising(&g_ble_abs0_ctrl, &gs_adv_param);
                for(uint8_t i=0;i< BLE_ABS_CFG_RF_CONNECTION_MAXIMUM;i++)
                {
                if(g_conn_hdl[i] == BLE_GAP_INVALID_CONN_HDL)
                {
                        g_conn_hdl[i] = p_gap_conn_evt_param->conn_hdl;
                }

                }
            }
        } break;

        case BLE_GAP_EVENT_DISCONN_IND:
        {
        st_ble_gap_disconn_evt_t *p_gap_disconn_evt_param = (st_ble_gap_disconn_evt_t*)p_data->p_param;


        for(uint8_t i=0;i< BLE_ABS_CFG_RF_CONNECTION_MAXIMUM;i++)
        {
                if(g_conn_hdl[i] == p_gap_disconn_evt_param->conn_hdl)
                {
                        g_conn_hdl[i] = BLE_GAP_INVALID_CONN_HDL;
                }
        }
        } break;

        default:
        {
            /* Do nothing. */
        } break;
    }
}
```

Code 19. Implementation example of GAP callback function when accepting connections from multiple centrals

In Bluetooth Low Energy, the Central (central device) controls the communication timing. Therefore, a disconnection may happen when communication timing of each connection accidentally collides. To avoid such a disconnection, it is recommended to update the connection parameters so that there is a margin in peripheral latency and supervision timeout time. For updating the connection parameters, refer to section7.3.

The GATT server may expose a common characteristic value to all connected GATT clients or may expose a different value for each client. When exposing different values for each client such as Client Configuration Characteristic Descriptor, user can enable “Peer Specific” configuration on the characteristic screen of QE for BLE as shown in Figure 25.

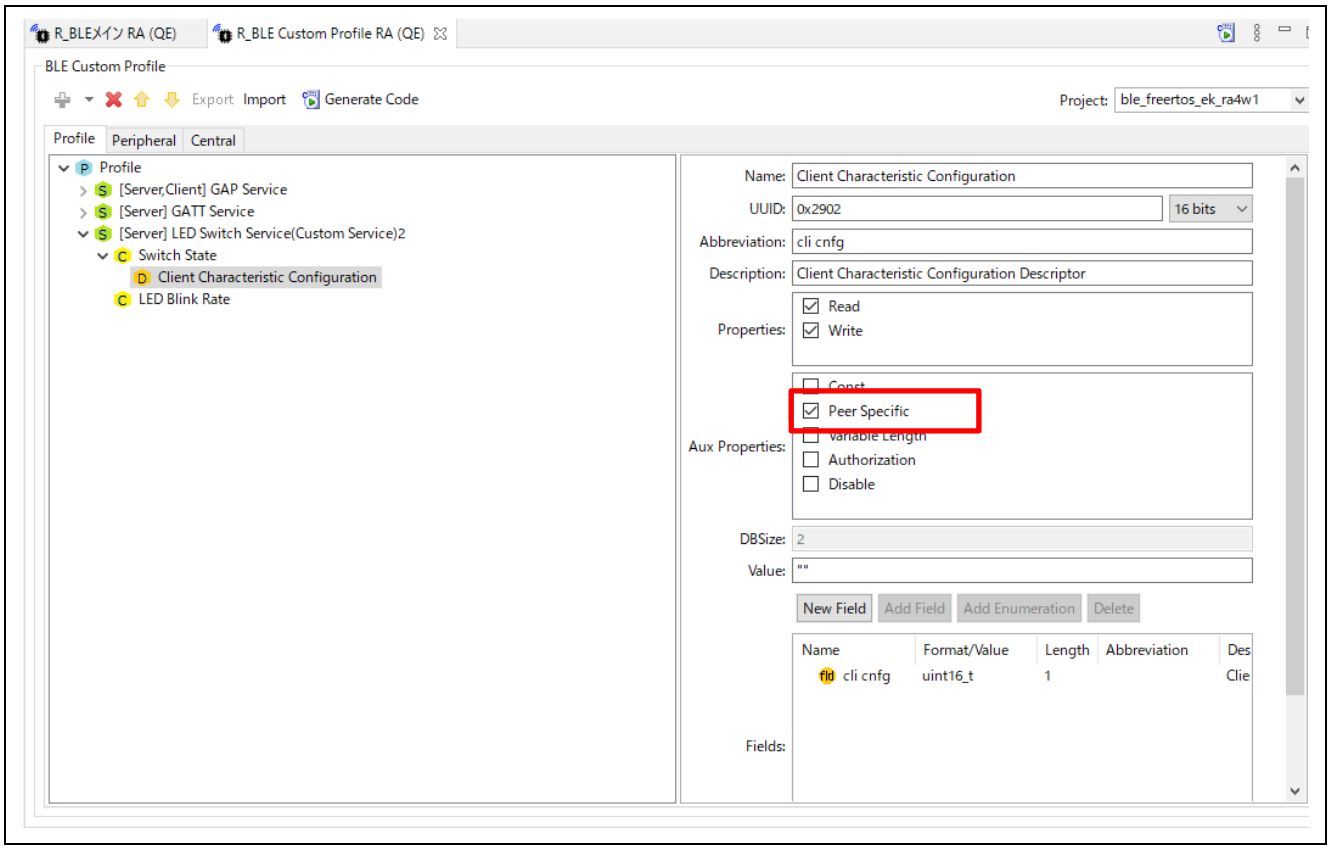


Figure 25. Setting to retain the value of characteristic for each device

A characteristic which has enabled “Peer Specific” configuration will be able to hold a separate value for up to 7 client devices and a GATT database value is returned for each client accessed.

6.6.3 Multi role connection

This section describes example of implementation when local device performs different GAP / GATT roles at the same time, as shown in Figure 26. In such a case, local device communicates as central to one remote device and as a peripheral to another remote device.

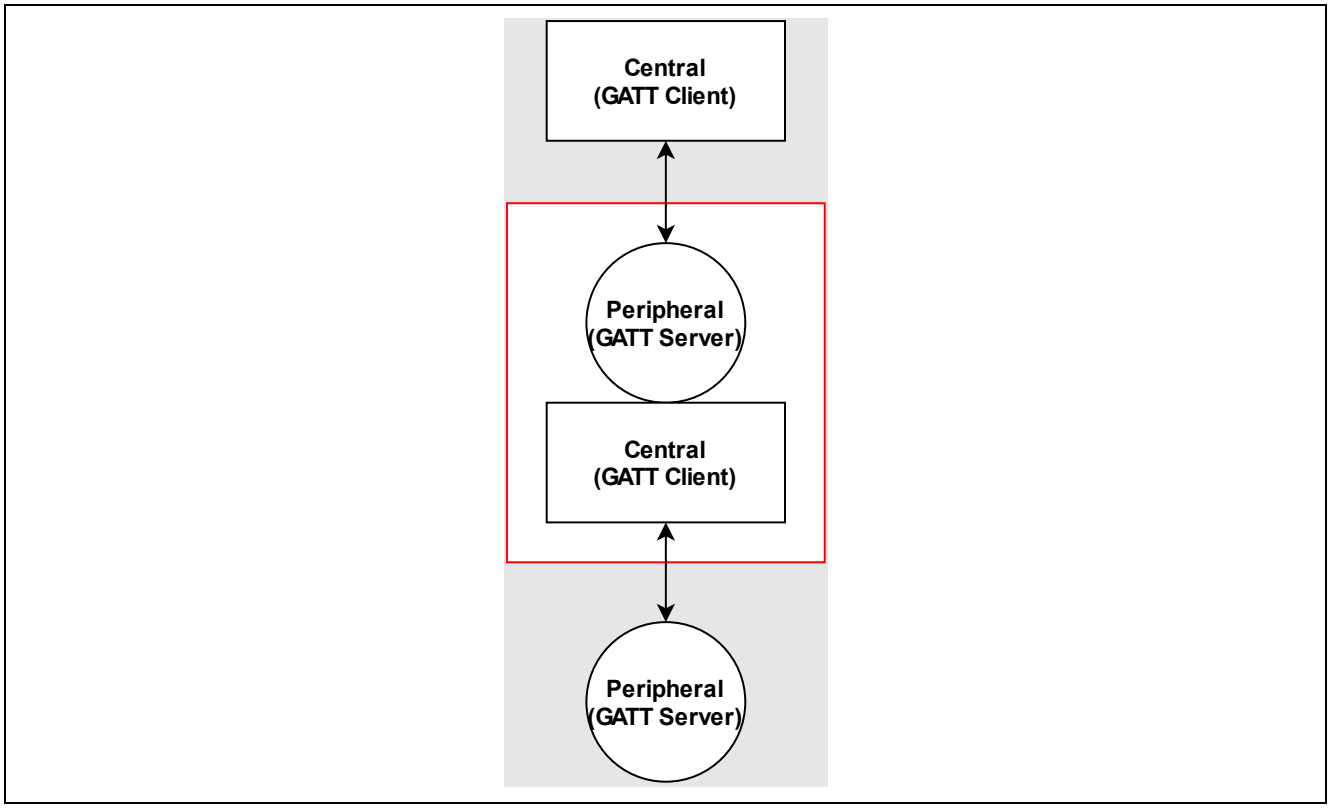


Figure 26. Multi roll connection example

1. GAP callback functions for each GAP role

QE for BLE cannot generate skeleton code for GAP callback function. Therefore, user needs to implement GAP callback function by themselves, as shown in Code 20. In this example, GAP callback function for peripheral and central is implemented separately.

```
static void gap_cb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
       ble_peripheral_gapcb(type, result, p_data);
       ble_central_gapcb(type, result, p_data);
}
```

Code 20. Call GAP callback function for each role

Each GAP role of example code is shown in Code 21 and Code 22.

```
/* Connection handle */
uint16_t g_central_conn_hdl;

static void ble_central_gapcb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
        switch (type)
        {
                case BLE_GAP_EVENT_STACK_ON:
                {
                        RM_BLE_ABS_StartScanning(&g_ble_abs0_ctrl, &gs_scan_param);
                } break;

                case BLE_GAP_EVENT_CONN_IND:
                {
                        if (BLE_SUCCESS == result)
                        {
                                st_ble_gap_conn_evt_t *p_gap_conn_evt_param =
                                        (st_ble_gap_conn_evt_t *)p_data->p_param;
                                if(0x00 == p_gap_conn_evt_param->role)
                                {
                                        g_central_conn_hdl = p_gap_conn_evt_param->conn_hdl;
                                }
                        }
                } break;

                case BLE_GAP_EVENT_DISCONN_IND:
                {
                        st_ble_gap_disconn_evt_t *p_gap_disconn_evt_param =
                                 (st_ble_gap_disconn_evt_t *)p_data->p_param;
                        if(p_gap_disconn_evt_param->conn_hdl == g_central_conn_hdl)
                        {
                                g_central_conn_hdl = BLE_GAP_INVALID_CONN_HDL;
                        }
                } break;

                case BLE_GAP_EVENT_CONN_PARAM_UPD_REQ:
                {
                        st_ble_gap_conn_upd_req_evt_t *p_conn_upd_req_evt_param =
                                 (st_ble_gap_conn_upd_req_evt_t *)p_data->p_param;
                        if(p_conn_upd_req_evt_param->conn_hdl == g_central_conn_hdl)
                        {
                                st_ble_gap_conn_param_t conn_updt_param = {
                                        .conn_intv_min = p_conn_upd_req_evt_param->conn_intv_min,
                                        .conn_intv_max = p_conn_upd_req_evt_param->conn_intv_max,
                                        .conn_latency  = p_conn_upd_req_evt_param->conn_latency,
                                        .sup_to        = p_conn_upd_req_evt_param->sup_to,
                                };

                                R_BLE_GAP_UpdConn(p_conn_upd_req_evt_param->conn_hdl,
                                                                        BLE_GAP_CONN_UPD_MODE_RSP,
                                                                        BLE_GAP_CONN_UPD_ACCEPT,
                                                                        &conn_updt_param);
                        }
                } break;
                case BLE_GAP_EVENT_ADV_REPT_IND:
                {
                        st_ble_gap_adv_rept_evt_t *p_adv_rept_param =
                                 (st_ble_gap_adv_rept_evt_t *)p_data->p_param;
                        st_ble_gap_ext_adv_rept_t *p_ext_adv_rept_param =
                                 (st_ble_gap_ext_adv_rept_t *)p_adv_rept_param->param.p_ext_adv_rpt;

                        gs_conn_param.p_addr->type = p_ext_adv_rept_param->addr_type;
                        memcpy(gs_conn_param.p_addr->addr, p_ext_adv_rept_param->p_addr, BLE_BD_ADDR_LEN);

                        R_BLE_GAP_StopScan();
                } break;

                case BLE_GAP_EVENT_SCAN_OFF:
                {
                        RM_BLE_ABS_CreateConnection(&g_ble_abs0_ctrl, &gs_conn_param);
                }break;
                 /** some code is omitted  **/
```

Code 21. Example of GAP callback function when connecting as a central role

```
/* Connection handle */
uint16_t g_peripheral_conn_hdl;

static void ble_peripheral_gapcb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
    switch (type)
    {
        case BLE_GAP_EVENT_STACK_ON:
        {
            RM_BLE_ABS_StartLegacyAdvertising(&g_ble_abs0_ctrl, &gs_adv_param);
        } break;

        case BLE_GAP_EVENT_CONN_IND:
        {
            if (BLE_SUCCESS == result)
            {
                st_ble_gap_conn_evt_t *p_gap_conn_evt_param = (st_ble_gap_conn_evt_t *)p_data->p_param;
                if(0x01 == p_gap_conn_evt_param->role)
                {
                g_peripheral_conn_hdl = p_gap_conn_evt_param->conn_hdl;
                }
            }
        } break;

        case BLE_GAP_EVENT_CONN_PARAM_UPD_REQ:
        {
            st_ble_gap_conn_upd_req_evt_t *p_conn_upd_req_evt_param =
                 (st_ble_gap_conn_upd_req_evt_t *)p_data->p_param;

            if(p_conn_upd_req_evt_param->conn_hdl == g_peripheral_conn_hdl)
            {
                st_ble_gap_conn_param_t conn_updt_param = {
                    .conn_intv_min = p_conn_upd_req_evt_param->conn_intv_min,
                    .conn_intv_max = p_conn_upd_req_evt_param->conn_intv_max,
                    .conn_latency  = p_conn_upd_req_evt_param->conn_latency,
                    .sup_to        = p_conn_upd_req_evt_param->sup_to,
                };

                R_BLE_GAP_UpdConn(p_conn_upd_req_evt_param->conn_hdl,
                                  BLE_GAP_CONN_UPD_MODE_RSP,
                                  BLE_GAP_CONN_UPD_ACCEPT,
                                  &conn_updt_param);
            }
        } break;

        case BLE_GAP_EVENT_DISCONN_IND:
        {
            st_ble_gap_disconn_evt_t *p_gap_disconn_evt_param =
                 (st_ble_gap_disconn_evt_t *)p_data->p_param;
            if(p_gap_disconn_evt_param->conn_hdl == g_peripheral_conn_hdl)
            {
                g_peripheral_conn_hdl = BLE_GAP_INVALID_CONN_HDL;
            }
        } break;


        default:
        {
        /* Do Nothing */
        }break;
}
```

Code 22. Example of GAP callback function when connected as a peripheral device

2. GATT callback functions for each GATT role

QE for BLE can generate the skeleton code for GATT in case of multi role. In the case of multi role, enable both the server and the client on QE for BLE as shown in Figure 27, and generate the service API for both the GATT client and the server.

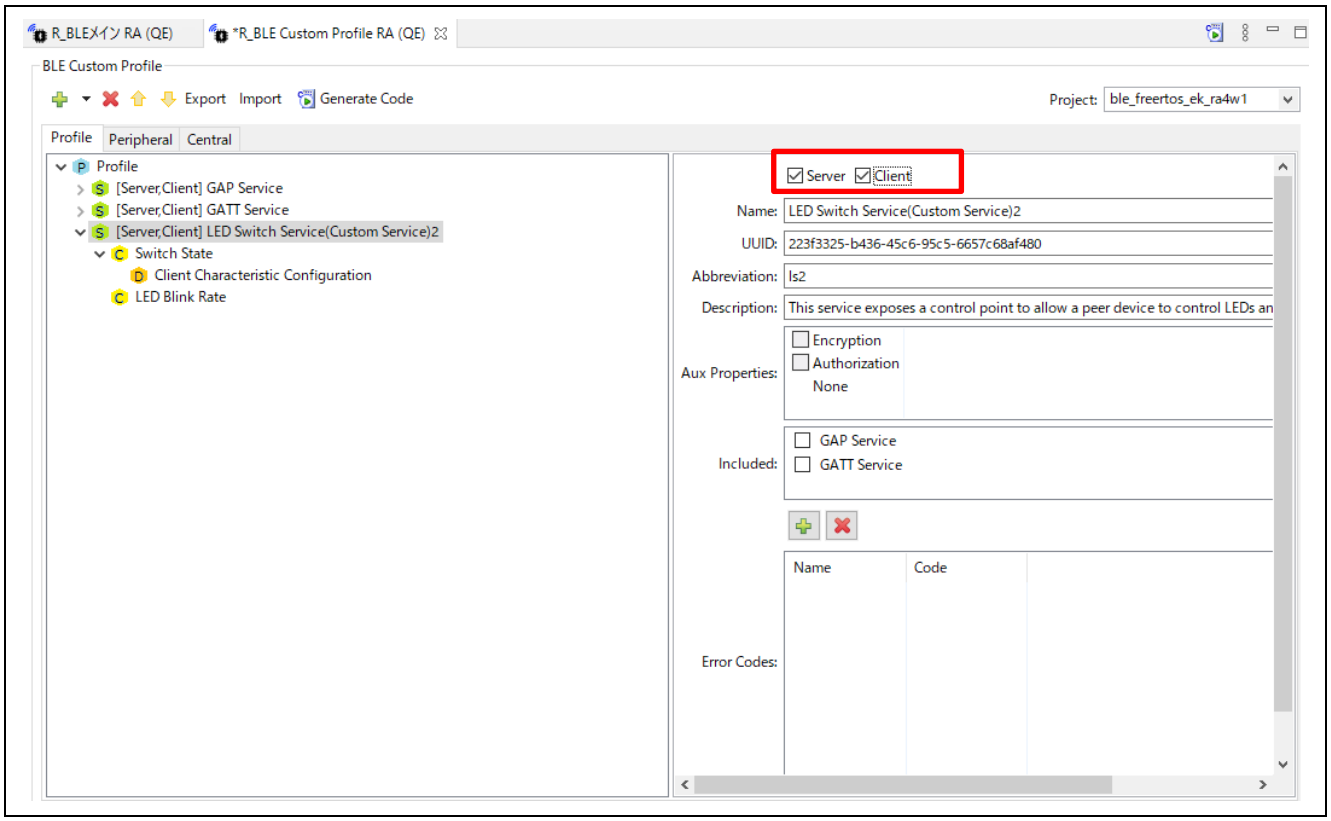


Figure 27. Select GATT Role on QE for BLE

In the example code shown in Code 23, local device performs as a GATT client when its own GAP role is central. Therefore, service discovery will be performed when connection is established with peripheral device.

```
/* XXX Service UUID */
static uint8_t XXXC_UUID[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00 };

/* Service discovery parameters */
static st_ble_disc_entry_t gs_disc_entries[] = {

    {
        .p_uuid     = XXXC_UUID,
        .uuid_type  = BLE_GATT_128_BIT_UUID_FORMAT,
        .serv_cb    = R_BLE_XXXC_ServDiscCb,
    },
};
static void disc_comp_cb(uint16_t conn_hdl)
{
    /* TODO: Add function after discovery completed */
    return;
}

static void gattc_cb(uint16_t type, ble_status_t result, st_ble_gattc_evt_data_t *p_data)
{
    R_BLE_SERVC_GattcCb(type, result, p_data);

    switch(type)
    {
        /* TODO: Set callback events of GATTC. Check BLE API reference for events. */

        case BLE_GATTC_EVENT_CONN_IND:
        {
                if(g_central_conn_hdl == p_data->conn_hdl)
                {
                        R_BLE_DISC_Start(p_data->conn_hdl,
                                         gs_disc_entries,
                                         ARRAY_SIZE(gs_disc_entries),
                                         disc_comp_cb);
                }
        } break;

        default:
        {
            /* Do nothing. */
        } break;
    }
}
```

Code 23. Implementation example of service discovery as a central device

As shown in bold frame of Code 23, when user registers *R_BLE_XXX_ServDiscCb* generated by QE for BLE, attribute handle of each peripheral device will be retained in the service API. The user application can access the GATT database of each peripheral device using the connection handle without managing the attribute handle of each peripheral device.

6.7 Disconnection

To disconnect an established connection, call the following *R_BLE_GAP_Disconnect* API. Specify the connection handle with *conn_hdl* and the disconnection reason with *reason* as argument of the API. Normally, 0x13 (*REMOTE USER TERMINATED CONNECTION*) is specified as the disconnection reason. For more information about the disconnection reason, refer to "*Bluetooth Core Specification Vol. 2 Part D, 2 Error Code Descriptions*".

Both of central and peripheral device can call this API. When the disconnection occurs, the user application is notified of the *BLE_GAP_EVENT_DISCONN_IND* event. The disconnection reason is notified in the reason field of the st_ble_gap_disconn_evt_t structure by the BLE_GAP_EVENT_DISCONN_IND event.

If the local device disconnects the link by *R_BLE_GAP_Disconnect* API,
reason: 0x16 (Connection Terminated by Local Host) will be notified.

If the remote device disconnects the link,
reason: 0x13 (Remote User Terminated Connection) will be notified in most cases. Otherwise, the disconnection reason that the remote device specifies will be notified.

If no packet is received within 6 connection intervals after starting a connection,
(for example, in an environment with many active scanning devices, peripheral device is busy responding to scan requests and cannot respond to connection request)
reason: 0x3E (Connection Failed to be Established) is notified.
As for connection interval, refer to "7.3 Updating connection parameter".

After establishing a connection, if no packet is received within the supervision timeout period,
reason: 0x08 (Connection Timeout) is notified.
As for supervision timeout, refer to "7.3 Updating connection parameter".

If the LTK of the local device and the remote device do not match when encryption is started,
reason: 0x3D (Connection Terminated due to MIC Failure) is notified.
Delete the bonding information and try pairing again, as the remote device cannot be trusted.
As for LTK, refer to "8.1 Pairing".

Note: When reconnecting to a disconnected remote device, the peripheral side needs to execute Connectable Advertising again.

7. Communication

Users can adjust the communication speed and power consumption to suit their own application by changing the communication parameters in Bluetooth Low Energy. This chapter describes how to configure communication parameters.

Table 31 shows the correspondence between the communication parameters described in this chapter and the libraries that support the functions.

Table 31. Communication parameters

Communication parameter	Feature name	Description	Library	Role
PHY	LE 2M PHY LE Coded PHY LE 1M PHY (optional other than 1M [*1])	Determined by Central's connection request and Peripheral's Advertising parameters when connecting. This can be changed after the connection.	Extended / Balance Extended / Balance All libraries	Central / Peripheral Central / Peripheral Central / Peripheral
Maximums transmission packet length	LE Data Length Extension (optional [*1])	Can extend Maximum number of transmitted bytes 27 → 251 bytes. The initial value is the value specified by BLE_CFG_RF_CONN_DATA_MAX. This can be changed after the connection.	All libraries	Central / Peripheral
Connection parameters	-	Determined by Central's connection request parameters when connecting. This can be changed after the connection.	All libraries	Central / Peripheral
MTU	-	The initial value is 23 bytes. This can be changed only once during the connection.	All libraries	Client

[*1]: The optional features may not be supported on remote device.

The following sections describe how to change the communication parameters by using various APIs. Refer to the *Renesas Flexible Software Package User's Manual* about details of these APIs.

7.1 Changing PHY

PHY is a parameter that indicates the physical layer modulation method and coding scheme. The user can expect improvement of throughput and reach of the radio wave. The modulation methods and coding schemes are shown below.

- **LE 1M PHY**
  - This is a modulation method that is compatible with all Bluetooth Low Energy devices.

- **LE 2M PHY**
  - This is a modulation method that doubles the symbol rate from LE 1M PHY and shortens the packet transmission time. This modulation method is used when performing high throughput communication. Users can also expect a reduction in power consumption since the packet transmission time is shortened.

- **LE Coded PHY**
  - This is a modulation method that applies a forward error correction code (coding scheme) of 1/2 or 1/8 to the header and payload of the packet. This modulation method increases certainty of data arrival to remote device and makes it possible to extend the communication distance compared to LE 1M and LE 2M PHY.

To change the PHY, use the *R_BLE_GAP_SetPhy* API. To use this API, it is necessary to specify the following arguments.

- **tx_phys**
  - The modulation method for transmission.
- **rx_phys**
  - The modulation method for reception.
- **phy_options**
  - The coding scheme for transmission. Note that the receiving coding scheme does not be changed.

Figure 28 shows the sequence chart when changing the PHY from the local device. PHY changes can be initiated from either role. The remote device can specify the PHY that allows changes with the R_BLE_GAP_SetDefPhy function. If not specified, all PHYs are accepted.

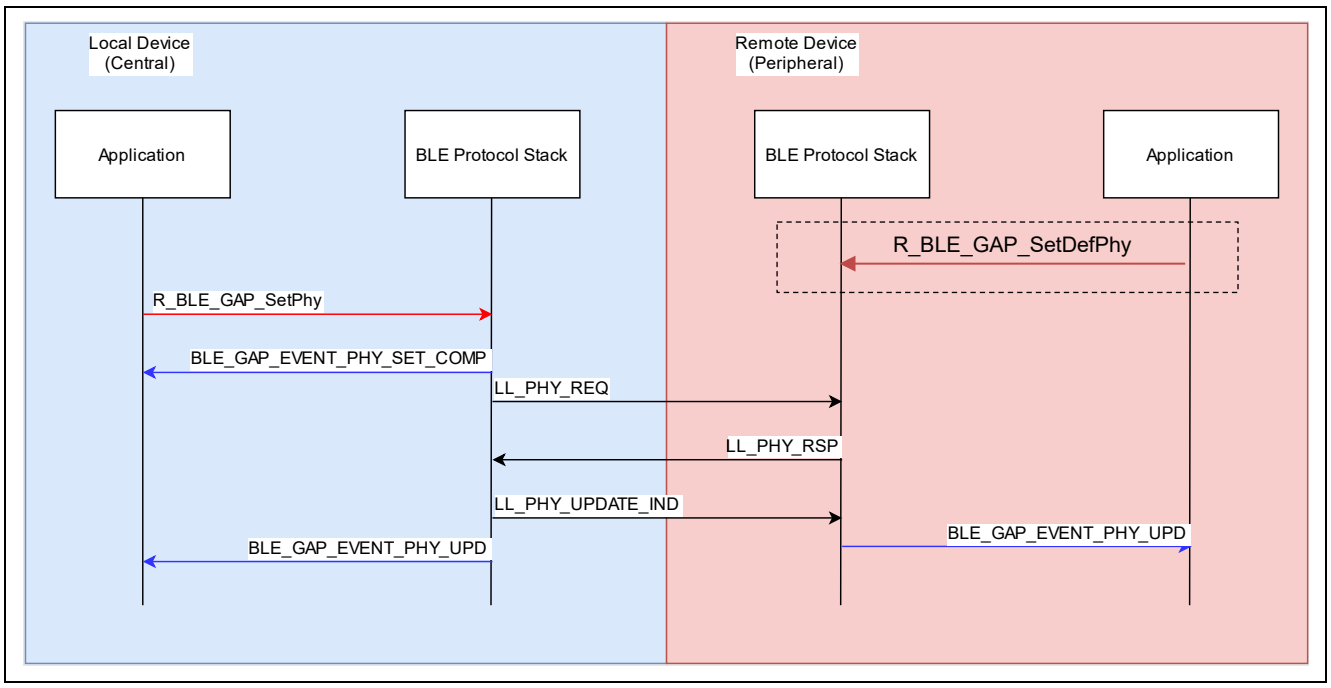


Figure 28. Sequence chart when changing PHY

The example for changing the PHY to LE Coded PHY (S=8) is shown in Code 24. Multiple PHYs can be also specified by bit sum. If you specify multiple PHYs, including the PHY in use, the PHY in use will not change. If you specify multiple PHYs without including the PHY in use, the PHY will change to the fastest PHY. The PHY you specify also applies to PHYs that allow modification from remote devices. The PHY in use can be obtained with the R_BLE_GAP_ReadPhy function.

```
st_ble_gap_set_phy_param_t set_phy = {
     .tx_phys = BLE_GAP_SET_PHYS_HOST_PREF_CD,
     .rx_phys = BLE_GAP_SET_PHYS_HOST_PREF_CD,
     .phy_options = BLE_GAP_SET_PHYS_OP_HOST_PREF_S_8
};

R_BLE_GAP_SetPhy(conn_hdl, &set_phy);
```

Code 24. Code to change PHY to LE Coded PHY (S=8)

The GAP callback function (*gap_cb*) will be notified of following two events when changing PHY, as shown in Code 25.

- **BLE _GAP_EVENT_PHY_SET_COMP**
  - This event will be issued when controller layer of the local device accepts the PHY change.

- **BLE_GAP_EVENT_PHY_UPD**
  - This event will be issued when the remote device accepts the PHY change. The issued event data, tx_phy and rx_phy, represent the actual PHY used when transmitting from the local device to the remote device and from the remote device to the local device respectively.

```
static void gap_cb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
    switch (type)
    {
      case BLE_GAP_EVENT_PHY_SET_COMP:
      {
        if(BLE_SUCCESS == result)
        {
          st_ble_gap_conn_hdl_evt_t *event_data =
            (st_ble_gap_conn_hdl_evt_t *)p_data->p_param;
          /*PHY parameter change in event_data->conn_hdl reaches Link Layer */
        }
        else if(BLE_ERR_INVALID_HDL == result)
        {
          st_ble_gap_conn_hdl_evt_t *event_data =
              (st_ble_gap_conn_hdl_evt_t *)p_data->p_param;
          /*The connection for event_data->conn_hdl was not found.*/
        }
        else
        {
          /* Do Nothing */
        }
      } break;

      case BLE_GAP_EVENT_PHY_UPD:
      {
        st_ble_gap_phy_upd_evt_t * event_data =
            (st_ble_gap_phy_upd_evt_t *)p_data->p_param;
      } break;
    }
}
```

Code 25. Event that occurs when PHY is changed

When the PHY is changed, the transmission time for the transmission packet length also changes. The BLE Protocol Stack will automatically change the maximum transmission packet length according to the applied PHY. When changed to LE Coded PHY, the maximum transmission packet length is set to 251 bytes and the transmission time is set to 2704µsec. If changing the maximum transmission packet length to 28 bytes or more, see section 7.2.

7.2 Changing maximum transmission packet length

This parameter sets the maximum packet length in the Link Layer. Users can perform efficient communication by extending the transmitting packet length when users want to transmit and receive application data that exceeds 23 bytes. Packet length extension requires remote device to support the LE Data Packet Length Extension feature in Bluetooth 4.2.

It is necessary for changing maximum transmission packet length to specify maximum number of bytes to be transmitted and the maximum transmission time. The packet transmission time is depended on PHY configuration which is described in the previous section. The maximum transmission packet length and maximum transmission time can be set depending on whether the LE Data Packet Length Extension and LE Coded PHY are supported. These relationships are shown in Table 32.

Table 32. Relationship between PHY and maximum transmission packet length and maximum transmission time

LE Data Packet Length Extension	LE Coded PHY feature supported	Parameters with names ending in “Octets”		Parameters with names ending in “Time"	
		Min	Max	Min	Max
No	No	27	27	328	328
Yes	No	27	251	328	2120
No	Yes	27	27	328	2704
Yes	Yes	27	251	328	17040

Bluetooth Core Specification V5.00 Vol 6, Part B

When connected to a remote device, BLE module request to change the maximum transmission packet length to the value specified by *“Maximum Connection data length*” configuration which is one of properties item of BLE module. Call *R_BLE_GAP_SetDataLen* API to change the maximum transmission packet length. It is necessary to specify connection handle whose configuration will be changed, maximum number of bytes to send and the maximum transmission time in microseconds as argument of the API. The BLE module adopts the smaller value of the time required to send the maximum number of bytes to be sent specified in the argument and the maximum transmission time specified in the argument. Figure 29 shows a sequence chart when maximum transmission packet length from the local device.

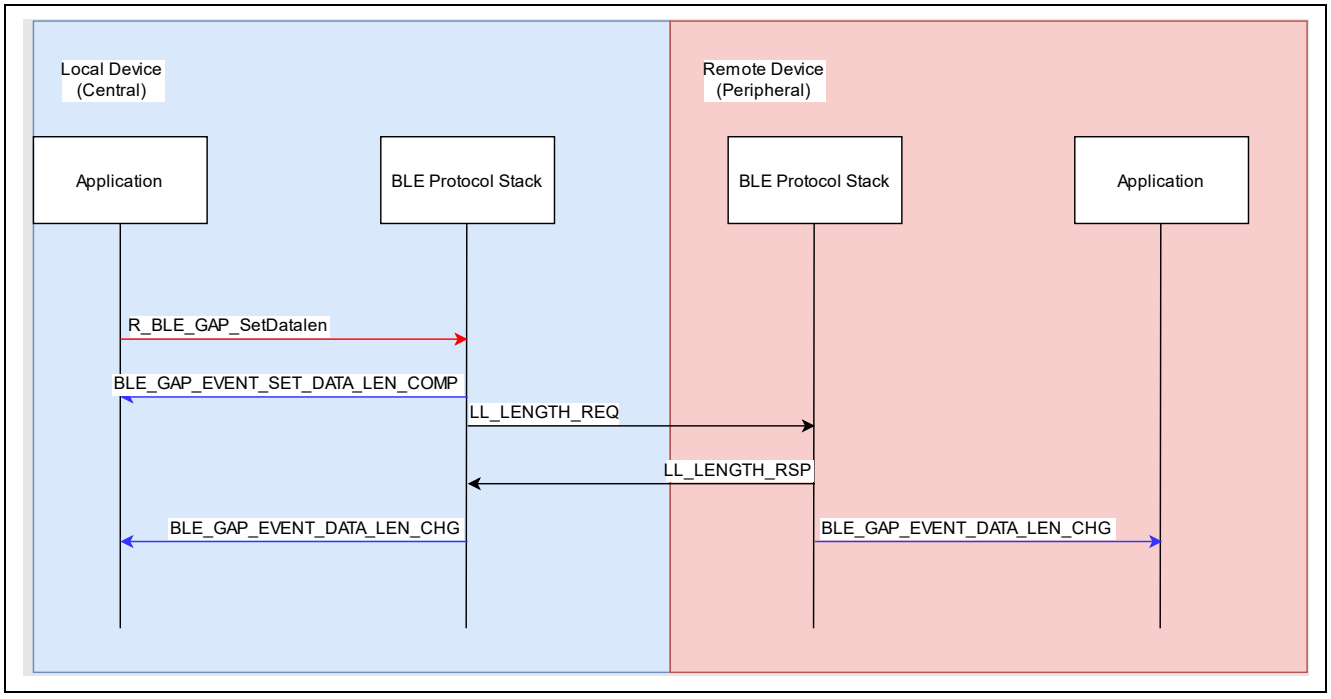


Figure 29. Sequence chart when changing the maximum transmission packet length

Code 26 is an example of expanding the packet length to 251 bytes when using the LE 1M PHY.

```
uint16_t tx_octets = 251;
uint16_t tx_time = 2120;

R_BLE_GAP_SetDataLen(conn_hdl, tx_octets, tx_time);
```

Code 26. Example of transmission packet length change request

GAP callback function (*gap_cb*) will be notified of following two events when changing the transmission packet length, as shown in Code 27.

- **BLE_GAP_EVENT_SET_DATA_LEN_COMP**
  - Occurs when the change in transmitted packet length is accepted by the controller layer.

- **BLE_GAP_EVENT_DATA_LEN_CHG**
  - Occurs when the transmission packet length changes with the remote device. This event will not occur if the remote device does not support LE Data Packet Length Extension.

```
static void gap_cb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
       switch(type)
       {
              case BLE_GAP_EVENT_SET_DATA_LEN_COMP:
              {
                     st_ble_gap_conn_hdl_evt_t * event_data =
                    (st_ble_gap_conn_hdl_evt_t *)p_data->p_param;
                     /* Do Nothing */
              } break;
              case BLE_GAP_EVENT_DATA_LEN_CHG:
              {
                     st_ble_gap_data_len_chg_evt_t * event_data =
                    (st_ble_gap_data_len_chg_evt_t *)p_data->p_param;
                     /* Do Nothing */
              } break;
       }
}
```

Code 27. Change packet length event

7.3 Updating connection parameter

Connection parameters are parameters related to communication frequency. Setting connection parameters is important for the efficient operation of user application. The connection parameters include the following items.

- **Connection Interval**
  - The interval between packet exchanges. When the connection interval is shortened, throughput will improve, but power consumption will increase. On the other hand, when the connection interval is lengthened, throughput will decrease, but power consumption can be reduced.

- **Peripheral Latency**
  - The number of times the Peripheral will ignore packets from the Central. When the Peripheral receives a packet from the Central, it returns a response. If there is no data to be transmitted from the Peripheral, the packet from the Central can be ignored for the number of times set for peripheral latency. The Peripheral does not have to return the response for the number of times, so the power consumption can be reduced. Figure 30 shows the relationship between peripheral latency and connection event.

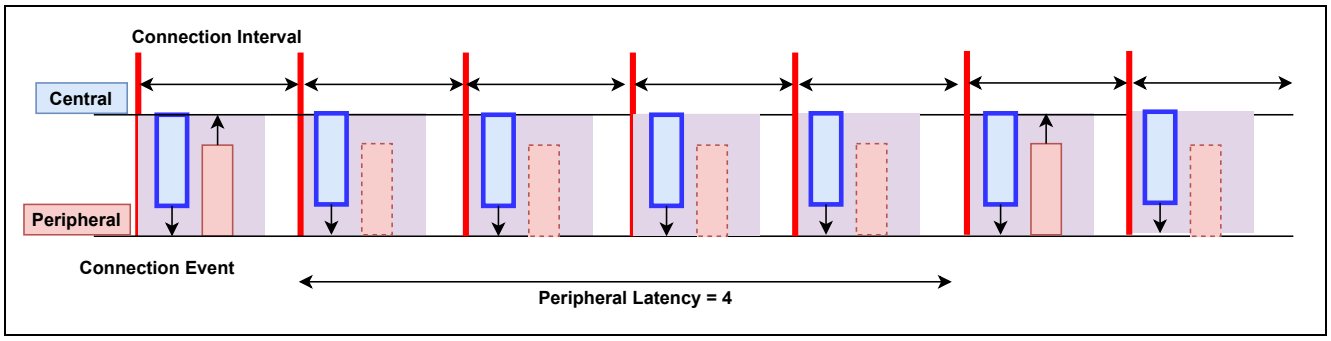


Figure 30. Peripheral latency and connection event

- **Supervision Timeout**
  - The time from the last packet reception to disconnection. If no packet is received within the time, the BLE connection will be disconnected. This time must be set to meet the following condition:

$$Supervision\ Timeout(msec) > \left(1 + Peripheral\ Latency(number)\right) * Connection\ Interval(msec) * 2$$

- **Connection Event Time**
  - Specify the connection event time that occurs at each connection interval. If zero is set, packets will be exchanged only once for each round trip per connection event, as shown in Figure 31. If 0xFFFF is specified, packets will be exchanged until the next connection event or until the More Data bit is not set, as shown in Figure 32.

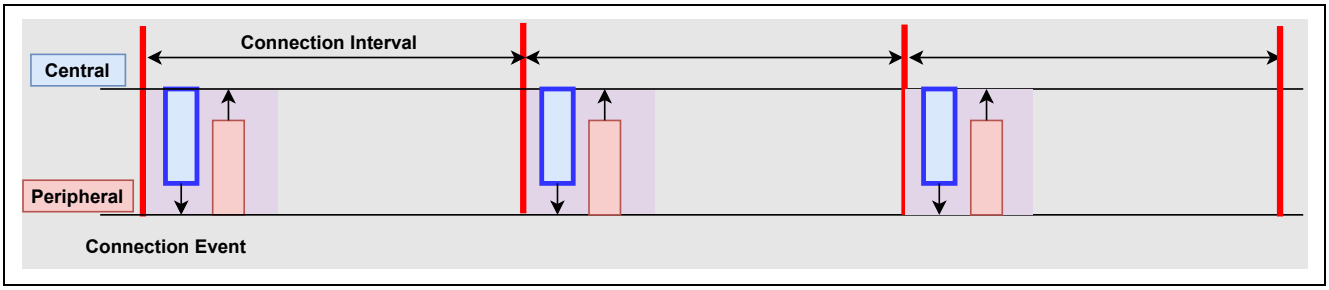


Figure 31. Connection event time and packet exchange (connection event time is set to 0)

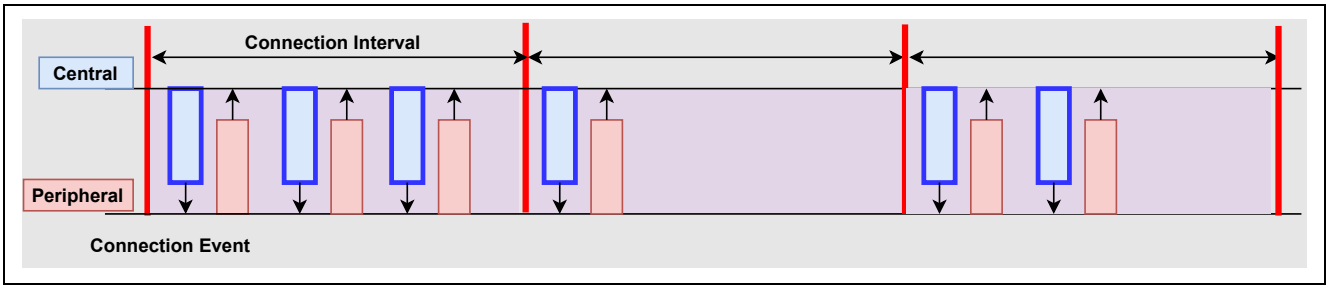


Figure 32. Connection event time and packet exchange (connection event time is set to 0xFFFF)

The Central determines and changes the connection parameters, but it is also possible to request connection parameters changes from Peripheral to Central. The connection parameters can be updated any number of times during the connection. The application flexibly updates the connection parameters to achieve efficient data communication. Following are typical scenarios.

- Since there is no data to send for a while, user wants to lengthen the connection interval to reduce power consumption.
- Since data communication is performed with multiple remote devices, user wants to lengthen the connection interval to ensure time for communication.
- User wants to shorten the connection interval to complete service discovery earlier.
- User wants to shorten the connection interval to send small data in short time.

Figure 33 shows the sequence chart for updating the connection parameters. The local device is the central and the remote device is the peripheral.

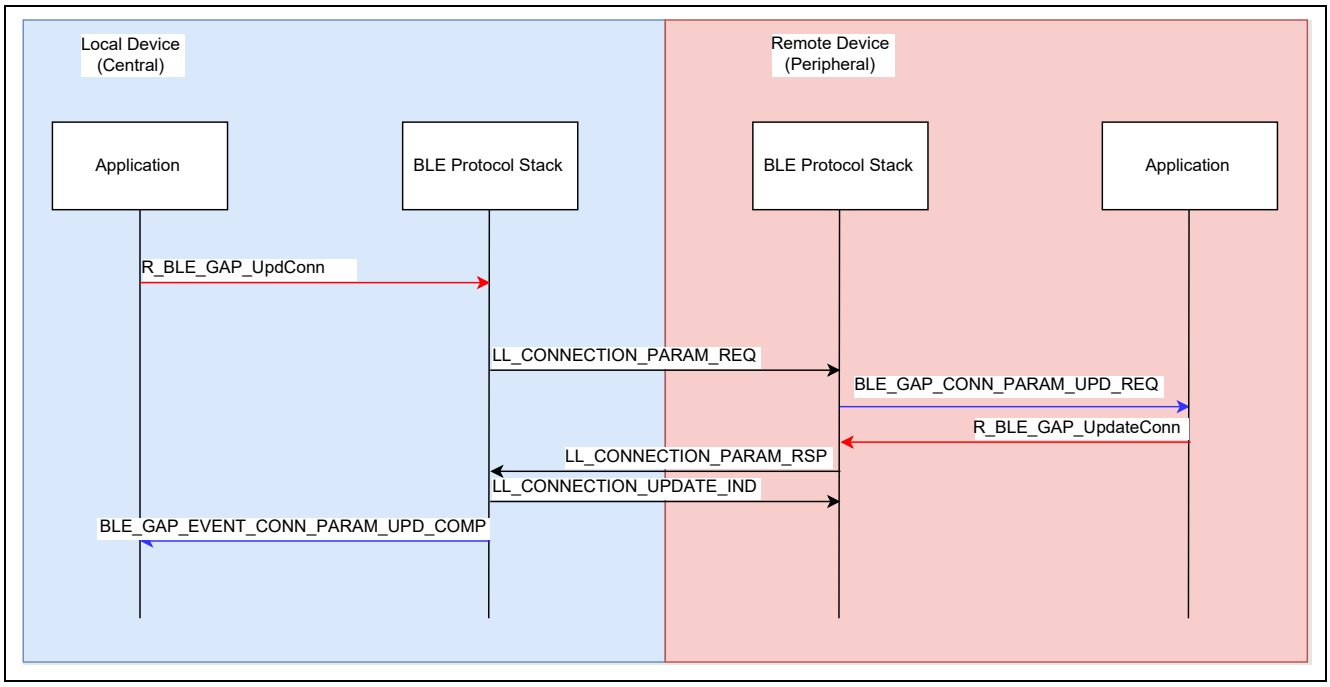


Figure 33. Sequence chart when updating connection parameters

Use *R_BLE_GAP_UpdConn* API for request/response of connection parameter update. Code 28 is an example of requesting to update the connection parameters from the local device.

```
st_ble_gap_conn_param_t conn_param = {
       .conn_intv_min = 0x0006,  //Connection Interval
       .conn_intv_max = 0x0006,
       .conn_latency  = 0x0000,  //Paripheral Latency
       .sup_to         = 0x0C80, //Supervision timeout
       .max_ce_length = 0xffff,  //Connection event time
       .min_ce_length = 0xffff
};

R_BLE_GAP_UpdConn(conn_hdl , BLE_GAP_CONN_UPD_MODE_REQ , 0 , &conn_param);
```

Code 28. Implementation example of connection parameter update request

GAP callback function (gap_cb) will be notified of following two events when updating the connection parameters.

- **BLE_GAP_EVENT_CONN_PARAM_UPD_REQ**
  - Issued when a request to update connection parameters is received from the remote device. The user needs to implement the process of whether to accept.

- **BLE_GAP_EVENT_CONN_PARAM_UPD_COMP**
  - Issued when the connection parameters have been updated. The argument *result* of *gap_cb* contains information whether the request to update the connection parameter was accepted or not. The argument *event* of *gap_cb* contains the connection parameters used in the actual connection.

Code 29 is an implementation example of the response to the connection parameter update request from the remote device. In this example, local device accepts all requests from remote devices. This process is implemented in *app_main.c*.

```
static void gap_cb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
    switch(type)
    {
      case BLE_GAP_EVENT_CONN_PARAM_UPD_REQ:
        {
            st_ble_gap_conn_upd_req_evt_t *p_conn_upd_req_evt_param =
                              (st_ble_gap_conn_upd_req_evt_t *)p_data->p_param;

            st_ble_gap_conn_param_t conn_updt_param = {
              .conn_intv_min = p_conn_upd_req_evt_param->conn_intv_min,
              .conn_intv_max = p_conn_upd_req_evt_param->conn_intv_max,
              .conn_latency  = p_conn_upd_req_evt_param->conn_latency,
              .sup_to    = p_conn_upd_req_evt_param->sup_to,
            };

            R_BLE_GAP_UpdConn(p_conn_upd_req_evt_param->conn_hdl,
                      BLE_GAP_CONN_UPD_MODE_RSP,
                      BLE_GAP_CONN_UPD_ACCEPT,
                      &conn_updt_param);
      } break;
    }
}
```

Code 29. Implementation example of response to connection parameter update request event

When connecting to a smartphone, update of connection parameters may not be accepted. For example, refer to the following document for more information about iOS. Accessories for Design Guidelines for Apple Devices (*https://developer.apple.com/accessories/Accessory-Design-Guidelines.pdf*)

If the remote device rejects to local device request, *BLE_ERR_INVALID_ARG*(0x0003) is stored in the *result* variable at the time of *BLE_GAP_EVENT_CONN_PARAM_UPD_COM*P event notification.

Code 30 is an implementation example in which the parameters are updated and requested again after being rejected by the remote device.

```
static void gap_cb(uint16_t type, ble_status_t result, st_ble_evt_data_t *p_data)
{
    switch(type)
    {
      case BLE_GAP_EVENT_CONN_PARAM_UPD_COMP:
        {
          if(BLE_ERR_INVALID_ARG == result)
          {
                st_ble_gap_conn_param_t conn_param = {
                 .conn_intv_min = 0x0028,      /* Connection Interval */
                 .conn_intv_max = 0x0028,
                 .conn_latency  = 0x0000,      /* Peripheral Latency */
                 .sup_to        = 0x0C80,      /* Supervision timeout */
                 .max_ce_length = 0xffff,      /* Connection event time */
                 .min_ce_length = 0xffff
                 };

              R_BLE_GAP_UpdConn(conn_hdl ,
                                BLE_GAP_CONN_UPD_MODE_REQ ,
                                0 ,
                                &conn_param);

          }
        } break;
    }
}
```

Code 30. Request to update connection parameters after being rejected by remote device

7.4 Changing MTU

MTU represents maximum packet length in GATT. Initial value of the MTU size is 23 bytes. This is called the default MTU. When users continue to use the default MTU as is:

- Client will use GATT Read Long Characteristic Value procedure to read data longer than 22 bytes from server. This means that multiple communications are required when reading data of 22 bytes or more from server.
- Client will use Write Long Characteristic Value procedure to write data longer than 20 bytes to server. This means that multiple communications are required when writing data of 20 bytes or more to server.
- Notification or Indication procedure cannot send more than 20 bytes of data from sever.

The MTU can be changed from the GATT client only once during the connection to avoid such a communication overhead. To minimize the overhead, user needs to adjust the relationship between MTU, and maximum transmission packet length described in section 7.2 as follows.

$$MTU(byte) = Maximum\ transmission\ packet\ length(byte) - 4(byte)$$

Figure 34 shows the sequence chart when changing the MTU.

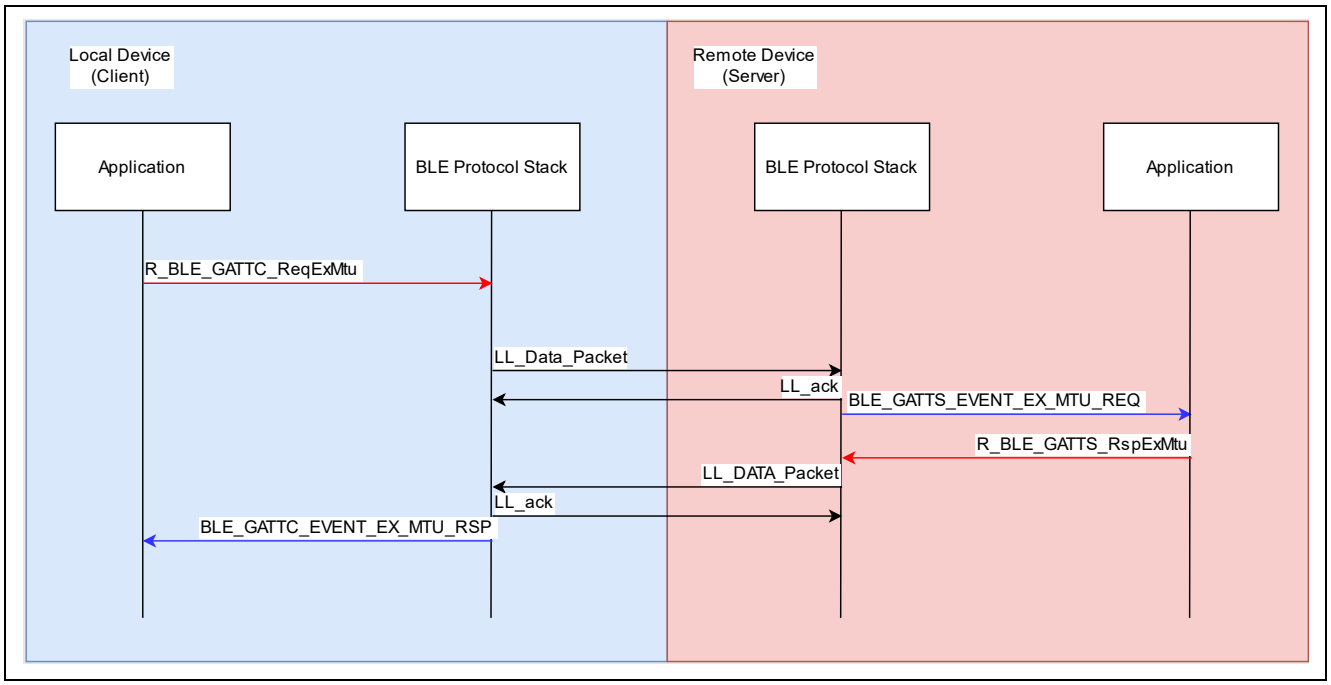


Figure 34. Sequence chart when changing MTU

Call *R_BLE_GATT_ReqExMtu* API to change the MTU, as shown in Code 31.

```
uint16_t mtu = 247
R_BLE_GATTC_ReqExMtu(conn_hdl, mtu);
```

Code 31. MTU change request example

GATT server / client callback function (*gatts_cb* / *gattc_cb*) will be notified of following two events when changing the MTU.

- **BLE_GATTS_EVENT_EX_MTU_REQ**
  - The server is notified when an MTU change request is received from a client device (*gatts_cb*). The server returns the MTU it supports in this event.

- **BLE_GATTC_EVENT_EX_MTU_RSP**
  - The client is notified when it receives an Exchange MTU Response from the server device (*gattc_cb*). The smaller of the MTU that client supports and the MTU included in the response is adopted as the MTU size.

Code 32 shows an implementation example of the GATT server response for the Exchange MTU Request from the GATT client. For the response, use *R_BLE_GATTS_RspExMtu* API. For the argument of the API, it is necessary to specify the MTU which supported in the local device. This process is implemented in *R_BLE_SERVS_GattsCb* function provided by QE for BLE. The size of the MTU returned by the GATT server is set in the *MTU Size Configured* configuration in properties of BLE module. When users generate the GATT server code from QE for BLE, user application does not need to implement MTU response.

```
static void gatts_cb(uint16_t type, ble_status_t result,
                     st_ble_gatts_evt_data_t *p_data)
{
    switch (type)
    {
        case BLE_GATTS_EVENT_EX_MTU_REQ:
        {
            R_BLE_GATTS_RspExMtu(p_data->conn_hdl, BLE_ABS_CFG_GATT_MTU_SIZE);
        } break;
    }
}
```

Code 32. Example of response to MTU change request

7.5 Flow control

BLE module has a flow control feature to send large application data in short time. BLE module has 10 send buffers for flow control feature. When flow control feature is enabled, an event will notify according to usage of the send buffer. Figure 35 shows the number of empty buffers and event notification timing. Number of remaining empty buffer will decrease as application repeatedly sends application data. The application will be notified of *BLE_VS_EVENT_TX_FLOW_STATE_CHG* event when number of remaining empty buffers reach *Low Water Mark*. Application should stop sending application data to prevent buffer overflow when receive the event.

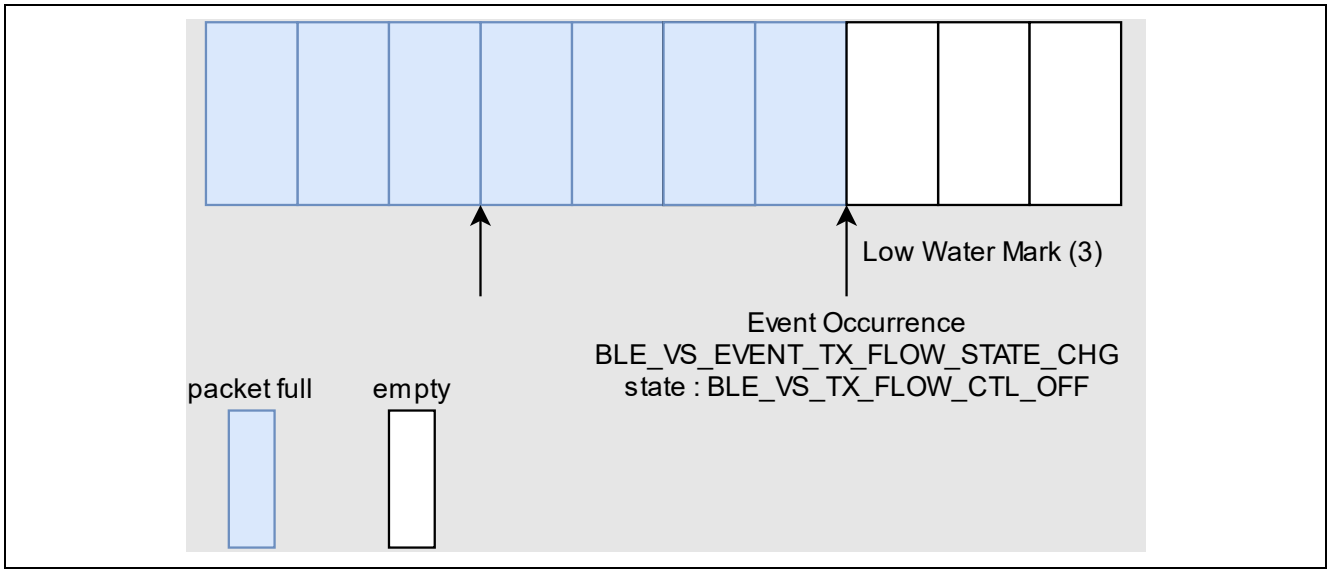


Figure 35. Number of empty buffers and events (Reach Low Water Mark)

Remaining empty buffer will increase as BLE module transmit application data to remote device. The application will be notified of *BLE_VS_EVENT_TX_FLOW_STATE_CHG* event when number of remaining empty buffer reached *High Water Mark,* as shown in Figure 36. Application should resume sending application data when receiving the event.

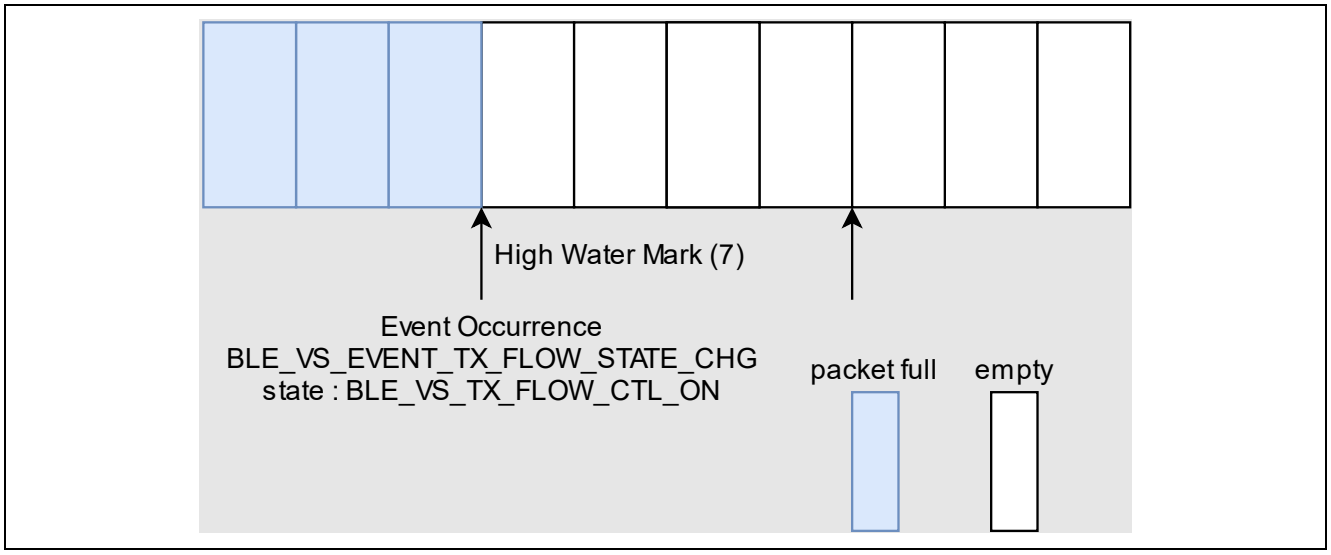


Figure 36. Number of empty buffers and events (Reach High Water Mark)

Application can transmit large data efficiently by repeating the above flow control.

The flow control feature is enabled by calling *R_BLE_VS_SetTxLimit* API and *R_BLE_VS_StartTxFlowEvtNtf* API, as shown in Code 33. *R_BLE_VS_SetTxLimit* API can configure *Low Water Mark* and *High Water Mark* of the send buffer where *BLE_VS_EVENT_TX_FLOW_STATE_CHG* event occurs.
R_BLE_VS_StartTxFlowEvtNtf API can enable event notification.

```
/* Enable Vender Specific Tx Flow Control */
#define LOW_WATER_MARK    (3)
#define HIGH_WATER_MARK   (7)

R_BLE_VS_SetTxLimit(LOW_WATER_MARK, HIGH_WATER_MARK);
R_BLE_VS_StartTxFlowEvtNtf();
```

Code 33. Start of flow control feature

The flow control feature notifies the application of *BLE_VS_EVENT_TX_FLOW_STATE_CHG* event. The event includes current buffer status. Example code is shown in Code 34. When number of empty buffers recovers to the High Water Mark, the notification API (*R_BLE_ServsCharNotification*) is called only (10-Low Water Mark) times continuously.

```
static void vs_cb(uint16_t type, ble_status_t result, st_ble_vs_evt_data_t *p_data)
{
      R_BLE_SERVS_VsCb(type, result, p_data);

      switch(type)
      {
          case BLE_VS_EVENT_TX_FLOW_STATE_CHG:
          {
              /* Apprize TxFlowState changed to txflow API */
            st_ble_vs_tx_flow_chg_evt_t * evt_data=
             (st_ble_vs_tx_flow_chg_evt_t *)p_data->p_param;
              if(BLE_VS_TX_FLOW_CTL_ON == evt_data->state)
              {
                for (int i=0; i<(10-LOW_WATER_MARK); i++)
               {
                   R_BLE_ServsCharNotification(conn_hdl, &app_data);
               }
              }
              else
              {
                /* Do Nothing */
              }
          } break;

}
```

Code 34. Implementation example of sending by flow control feature event

R_BLE_ServsCharNotification API is just an example. Therefore, it is necessary change the API according to service which using in your application.

7.6 High throughput communication

When performing high-throughput communication using Bluetooth Low Energy, it is important to set the communication parameters to optimal values and call the send API continuously using the flow control function.

8. Security

This section describes the security features provided by Bluetooth Low Energy.

In order to select the necessary security functions for the final product, it is important to determine the product use case and clarify the security requirements.

8.1 Pairing

Pairing procedure is necessary to use Bluetooth security features. Following shows typical scenarios which need the pairing process.

- There is GATT characteristics that can be changed.
- Requires address resolving for private addresses.
- Protect a device from malicious attacks such as data eavesdropping, falsification, and device tracking.

Pairing procedure exchanges following keys with a remote device. The remote device keys are notified by BLE_GAP_EVENT_PEER_KEY_INFO event. For more information about how to get the keys, see “8.1.7 Key exchange”.

- **LTK (Long Term Key), EDIV, Rand**
  - The LTK will be used as the encryption key (Key exchange is not enforced in LE Secure Connections).
- **IRK (Identity Resolving Key), Identity Address**
  - The IRK will be used as resolving private address of remote device.
- **CSRK (Connection Signature Resolving Key)**
  - Signed data send/receive will use CSRK.

Pairing procedure has LE Legacy pairing and LE Secure Connections. LE Secure Connections is supported from Bluetooth version 4.2. LE legacy pairing is the paring procedure is used by the device which does not support LE Secure Connections. If a remote device supports LE Secure Connections, the BLE Protocol Stack will perform LE Secure Connections. If a remote device does not support LE Secure Connections, the BLE Protocol Stack will perform LE Legacy Pairing. The pairing procedure in an application is shown in Figure 37. The following sections describe the details of pairing steps.

BP: LE Secure Connections is the most secure pairing and encryption mechanism where LTK is not sent over the air. LTK also has enough entropy at 16 octets to protect encrypted links from brute force attacks. Because of these things, for those designing more secure products, recommend supporting LE Secure Connections unless there are restrictions on the remote device side.
In order to avoid attacks from attackers who aim at the security down mechanism, when performing pairing only with LE Secure Connections (not accept LE legacy pairing), set *BLE_GAP_SC_STRICT*(0x01) in Table 41 to the secure_connection_only member of the parameter structure of *BLE_GAP_SetPairingParams* or *RM_BLE_ABS_Open* API.

BP: Signed data is provided for devices that want to avoid the overhead of encryption, and by ensuring data integrity, data falsification by attackers can be avoided. However, since it does not support protection against eavesdropping through encryption, it is possible for attackers to use replay attacks (unauthorized access by spoofing the user by using the data obtained by eavesdropping on communication). Therefore, it is recommended to avoid using signed data and support encryption.

Flow chart of pairing procedure is shown in Figure 37.

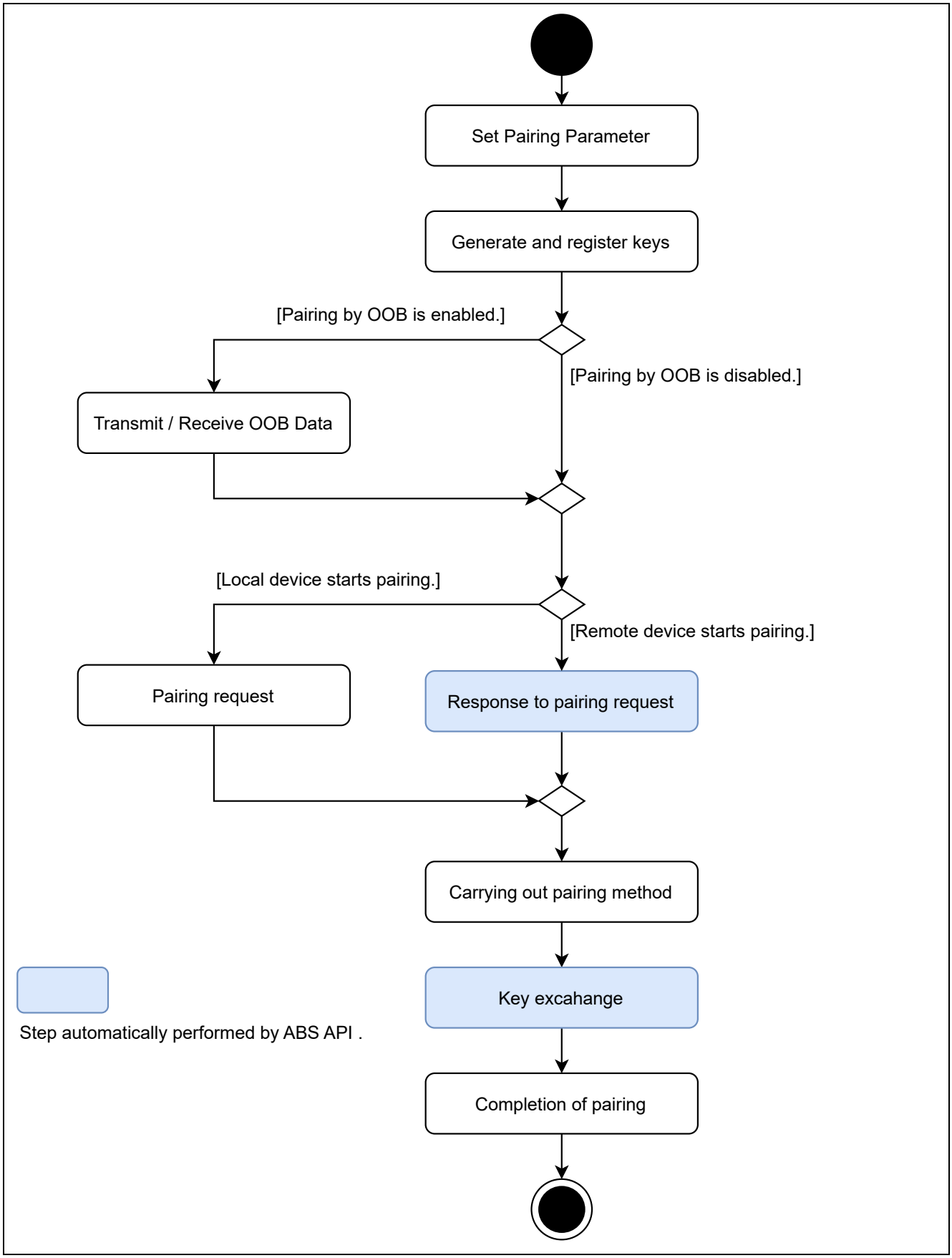


Figure 37. Pairing procedure in application

8.1.1 Pairing Parameters

It is necessary to configure pairing parameters before starting the pairing procedure. The pairing parameters are set by using *R_BLE_GAP_SetPairingParams* API or *RM_BLE_ABS_Open* API. Table 33 shows the pairing parameters. The following sections describe the detail of these parameters.

Table 33. Pairing Parameters

API		**RM_BLE_ABS_Open**	**R_BLE_GAP_SetPairing Params**	**Value Range**	**When generate application code by using QE for BLE, the application code is using RM_BLE_ABS_Open API and following pairing parameters.**
Parameter Structure		**ble_abs_pairing_parameter_t**	**st_ble_gap_pairing_param_t**		
1. Input Output capabilities		io_capabilitie_local_device	iocap	BLE_GAP_IOCAP_DISPLAY_ ONLY(0x00) BLE_GAP_IOCAP_DISPLAY_ YESNO(0x01) BLE_GAP_IOCAP_KEYBOARD_ ONLY(0x02) BLE_GAP_IOCAP_NOINPUT_ NOOUTPUT(0x03) BLE_GAP_IOCAP_KEYBOARD_ DISPLAY(0x04)	BLE_GAP_IOCAP_ NOINPUT_NOOUTPUT (0x03)
2. MITM Protection Request		mitm_protection_policy	mitm	BLE_GAP_SEC_MITM_BEST_ EFFORT(0x00) BLE_GAP_SEC_MITM_STRICT(0x01)	BLE_GAP_SEC_MITM_ BEST_EFFORT(0x00)
3. Bonding		No parameter Fixed to BLE_GAP_ BONDING(0x01)	bonding	BLE_GAP_BONDING_NONE(0x00) BLE_GAP_BONDING(0x01)	BLE_GAP_BONDING (0x01)
4. Encryption Key Size	Max Size	No parameter Fixed to 16	max_key_size	7 to 16	16
	Min Size	maximum_key_size	min_key_size	7 to 16	16
5. Exchange Key type	Keys that local device distributes	local_key_distribute	loc_key_dist	0(Keys are not distributed.) BLE_GAP_KEY_DIST_ENCKEY(0x01)	BLE_GAP_KEY_DIST_ ENCKEY(0x01)
	Keys that local device requests to distribute	remote_key_distribute	rem_key_dist	BLE_GAP_KEY_DIST_IDKEY(0x02) BLE_GAP_KEY_DIST_SIGNKEY(0x04)	0
6. Key Press Notification Support		No parameter Fixed to BLE_GAP_ SC_KEY_PRESS_ NTF_NOT_SPRT	key_notf	BLE_GAP_SC_KEY_PRESS_NTF_ NOT_SPRT(0x00) BLE_GAP_SC_KEY_PRESS_NTF_ SPRT(0x01)	BLE_GAP_SC_KEY_ PRESS_ NTF_NOT_SPRT(0x00)
7. LE Secure Connections Request		secure_connection_only	sec_conn_only	BLE_GAP_SC_BEST_EFFORT(0x00) BLE_GAP_SC_STRICT(0x01)	BLE_GAP_SC_BEST_ EFFORT(0x00)

1. Input Output capabilities

Table 34 and Table 35 show Input and output capabilities that the local device supports.

Table 34. Input capability

Input capability	Description
No Input	Device cannot indicate “Yes” and “No”.
Yes / No	Device can indicate “Yes” and “No”.
Keyboard	Device can indicate “Yes” and “No” and input numbers 0 through 9.

Table 35. Output capability

Output capability	Description
No Output	Device cannot display 6-digit number.
Numeric output	Device can display 6-digit number.

The values to be set in Input Output capabilities for each combination is shown in Table 36.

Table 36. Input Output capability

		Output	
		No output	Numeric output
Input	No input	NoInputNoOutput BLE_GAP_IOCAP_NOINPUT_NOOUTPUT(0x03)	DisplayOnly BLE_GAP_IOCAP_DISPLAY_ONLY(0x00)
	Yes / No	NoInputNoOutput BLE_GAP_IOCAP_NOINPUT_NOOUTPUT(0x03)	DisplayYesNo BLE_GAP_IOCAP_DISPLAY_YESNO(0x01)
	Keyboard	KeyboardOnly BLE_GAP_IOCAP_KEYBOARD_ONLY(0x02)	KeyboardDisplay BLE_GAP_IOCAP_KEYBOARD_DISPLAY(0x04)

2. MITM(Man-In-The-Middle) protection

The parameters in Table 37 specify whether to require protection against MITM.

Table 37. MITM Protection

MITM Protection	Settings
Depending on remote device	BLE_GAP_SEC_MITM_BEST_EFFORT(0x00)
Yes	BLE_GAP_SEC_MITM_STRICT(0x01)

According to section 8.1.6, completing pairing with any pairing method other than Just Works enables the MITM protection.

BP: For LE Secure Connections, it is recommended to design devices that support authenticated pairing using input, output or OOB mechanism to reduce the chances of a MITM obtaining a shared secret key during pairing.

3. Bonding

Table 38 shows the bonding parameter settings for whether the local device perform bonding or not. For more details about bonding, refer to section 8.2.

Table 38. Bonding

Bonding Type	Settings
No bonding	BLE_GAP_BONDING_NONE(0x00)
Bonding	BLE_GAP_BONDING(0x01)

If the application uses *RM_BLE_ABS_Open* API, the bonding type is fixed to “Bonding”.

4. Encryption Key Size

Select encryption key size between 7 to 16 bytes. It is recommended that the encryption key size is 16 bytes because a short encryption key size can cause an access rejection to the remote device.

BP: Recommend setting it to maximum entropy (16 octets) to protect encrypted links from brute force attacks.

5. Type of key exchanged by pairing

Table 39 shows the type of keys which local device distributes and requests to the remote device.

Type of key exchanged by pairing parameter can be specified by OR.

Table 39. Key Type

Key type	Settings
LTK, EDIV, Rand	BLE_GAP_KEY_DIST_ENCKEY(0x01)
IRK, Identity Address	BLE_GAP_KEY_DIST_IDKEY(0x02)
CSRK	BLE_GAP_KEY_DIST_SIGNKEY(0x04)

6. Key Press Notification support

Key Press Notification is used when Passkey Entry is selected according to section 8.1.6. If Key Press Notification is supported, the event is notified to the remote device when the local device key is pressed. Specify the feature support with the value in Table 40.

Table 40. Key Press Notification support

Key Press Notification Support	Value
Not Support	BLE_GAP_SC_KEY_PRESS_NTF_NOT_SPRT(0x00)
Support	BLE_GAP_SC_KEY_PRESS_NTF_SPRT(0x01)

Key Press Notification support is fixed to “Not Support” when the application uses *RM_BLE_ABS_Open* API.

7. LE Secure Connections Requirement

Table 41 shows the parameter determine whether pairing is permitted by only LE Secure Connections or not.

Table 41. Secure Connections Only Requirement

LE Secure Connections Only Requirement	Value
Depending on the remote device	BLE_GAP_SC_BEST_EFFORT(0x00)
Required	BLE_GAP_SC_STRICT(0x01)

When LE legacy pairing starts with *BLE_GAP_SC_STRICT* specified, *BLE_ERR_SMP_LE_AUTH_REQ_NOT_MET*(0x2003) is notified by result in BLE_GAP_EVENT_PAIRING_COMP event.

An example of setting the pairing parameters by using *R_BLE_GAP_SetPairingParams* API is shown in Code 35.

```
st_ble_gap_pairing_param_t pairing_param = {
    .iocap         = BLE_GAP_IOCAP_NOINPUT_NOOUTPUT,
    .mitm          = BLE_GAP_SEC_MITM_BEST_EFFORT,
    .bonding       = BLE_GAP_BONDING,
    .max_key_size  = 16,
    .min_key_size  = 16,
    .loc_key_dist  = BLE_GAP_KEY_DIST_ENCKEY | BLE_GAP_KEY_DIST_IDKEY,
    .rem_key_dist  = BLE_GAP_KEY_DIST_ENCKEY | BLE_GAP_KEY_DIST_IDKEY,
    .key_notf      = BLE_GAP_SC_KEY_PRESS_NTF_NOT_SPRT,
    .sec_conn_only = BLE_GAP_SC_BEST_EFFORT,
};

R_BLE_GAP_SetPairingParams(&pairing_param);
```

Code 35. Example of setting pairing parameter

The above code is not necessary when user uses *RM_BLE_ABS_Open* API to open BLE module.

8.1.2 Key generation and registration

This section describes how to generate and register local IRK and CSRK. These keys are used for key exchange, which is one of pairing procedures. Related APIs are shown in Table 42.

Table 42. The APIs used for key generation

Procedure	API
Local IRK generation	RM_BLE_ABS_SetLocalPrivacy *1 or R_BLE_VS_GetRand
Local IRK, Identity Address registration	RM_BLE_ABS_SetLocalPrivacy *1 or R_BLE_GAP_SetLocIdInfo
CSRK generation	R_BLE_VS_GetRand
CSRK registration	R_BLE_GAP_SetLocCsrk

*1 : *RM_BLE_ABS_SetLocalPrivacy* API performs both of generation and registration of the local device IRK and uses current Public Address or Static Address as Identity Address.

An example of local IRK generation and registration is shown in Code 36. In this example, 16-bytes of random number which obtained by *R_BLE_VS_GetRand* API is used to generate IRK and CSRK.

```
/** some code is omitted  **/
/* IRK generation */
R_BLE_VS_GetRand(0x10);
/** some code is omitted  **/

/* Vendor Specific Callback function */
void vs_cb(uint16_t event_type, ble_status_t result,
           st_ble_vs_evt_data_t * p_event_data)
{
    /** some code is omitted  **/
    case BLE_VS_EVENT_GET_RAND
    {
        st_ble_vs_get_rand_comp_evt_t * p_rand_param;
        p_rand_param = (st_ble_vs_get_rand_comp_evt_t *)p_event_data->p_param;
        /* register local IRK and identity address */
        R_BLE_GAP_SetLocIdInfo(&loc_bd_addr, p_rand_param);
    } break;
    /** some code is omitted  **/
}
```

Code 36. Example of key generation and registration

Refer to Code 41 about example of local IRK generation and registration. The example code shows local IRK generation and registration by using abstraction API.

Some notes about key generation and registration are shown below.

- It does not need to generate and register the local device IRK when the application does not use RPA (Resolvable Private Address).
- It does not need to generate and register the local device CSRK when the application does not communicate with the signed data.
- It does not need to generate the local device LTK (case of LE legacy pairing includes EDIV, Rand) before start pairing. Generated by the protocol stack as needed.

BP: EDIV is included in the data exchanged between devices paired with LE legacy pairing. EDIV is unique to a particular pair of devices, allowing tracking of paired devices using EDIV when using private addresses. It is recommended to periodically establish new pairings/bonds between devices to update the EDIV to prevent long-term tracking.

8.1.3 OOB (Out of Band) data transmission and reception

If local device and remote device have a common means of communications except Bluetooth (OOB) , the data for pairing can be transmitted and received through OOB. The data consists of confirm value (16 bytes) and random value (16 bytes). It needs to meet the condition in Table 43 to do pairing by OOB. If OOB is available, the data is transmitted and received before starting pairing.

Table 43. The conditions to do pairing by OOB

Pairing method	Condition
LE Secure Connections	The one device can transmit the data for pairing by OOB and the other can receive it.
LE legacy pairing	Both devices can transmit and receive the data for pairing by OOB.

Call *R_BLE_GAP_CreateScOobData* API on the local device to send data by OOB. The API will generate confirm value (16 bytes) and random value (16 bytes) as data for pairing according to the SMP specifications. When the data for pairing generation is complete, the *BLE_GAP_EVENT_SC_OOB_CREATE_COMP* event is issued. The local device should send the data for pairing to remote device by OOB after the event notified.

Call *R_BLE_GAP_SetRemOobData* API when the local device received data for pairing from remote device. The local device will notify remote device that OOB reception is success by calling the API.

BP: The TK (Temporary Key) exchanged in OOB has a maximum entropy of 128 bits and is the most resistant to MITM attacks. Support for the OOB mechanism is recommended if LE legacy pairing needs to be supported.

8.1.4 Pairing request

To request pairing from a local device, use one of the following APIs.

- *RM_BLE_ABS_StartAuthentication*
- *R_BLE_GAP_StartPairing*

These APIs can be called from both a central and a peripheral.

8.1.5 Response to pairing request

BLE_GAP_EVENT_PAIRING_REQ event will be issued when a pairing request is received from a remote device. It is necessary to respond to the event by using *R_BLE_GAP_ReplyPairing* API. An example of a response to a pairing request is shown in Code 37.

```
/* GAP Callback */
void gap_cb(uint16_t event_type, ble_status_t event_result, st_ble_evt_data_t * p_event_data)
{
    /** some code is omitted  **/
    case BLE_GAP_EVENT_PAIRING_REQ :
    {
        st_ble_gap_pairing_info_evt_t * p_param;
        p_param = (st_ble_gap_pairing_info_evt_t *)p_event_data->p_param;
        R_BLE_GAP_ReplyPairing(p_param->conn_hdl, BLE_GAP_PAIRING_ACCEPT);
    }
    break;
    /** some code is omitted  **/
```

Code 37. Response to a pairing request

Automatically respond a pairing request from remote device when *RM_BLE_ABS_Open* API is used to open BLE module.

8.1.6 Carrying out pairing method

By starting pairing or responding to pairing request, local device, and the remote device exchange pairing parameters. After exchanging the parameters, both devices select a pairing method from Table 44 and perform the pairing method.

Table 44. Pairing Method

Pairing Method	Description	MITM Protection
OOB	The application does not need to manage pairing, because the BLE Protocol Stack processes the OOB data previously received/transmitted.	Enable
Passkey Entry	The one device displays a 6-digit number, the other inputs the number.	Enable
Numeric Comparison	Both devices display a 6-digit number. Check if two numbers are same.	Enable
Just Works	The application does not need to manage pairing, because it is automatically performed.	Disable

The pairing method is determined according to following conditions.

1. If the OOB data is received/transmitted before pairing, the OOB pairing method will be selected.
2. If the OOB data is not received/transmitted and both devices do not require the MITM protection, the Just Works pairing method will be selected.
3. If the OOB data is not received/transmitted and which device requires the MITM protection, the pairing method is determined according to Table 45.

Table 45. Pairing Method Selection

Peripheral	Central				
	DisplayOnly	DisplayYesNo	KeyboardOnly	NoInputNoOutput	KeyboardDisplay
DisplayOnly	Just Works	Just Works	Passkey Entry	Just Works	Passkey Entry
DisplayYesNo	Just Works	Just Works (LE legacy pairing)	Passkey Entry	Just Works	Passkey Entry (LE legacy pairing)
		Numeric Comparison (LE Secure Connections)			Numeric Comparison (LE Secure Connections)
KeyboardOnly	Passkey Entry	Passkey Entry	Passkey Entry	Just Works	Passkey Entry
NoInputNoOutput	Just Works	Just Works	Just Works	Just Works	Just Works
KeyboardDisplay	Passkey Entry	Passkey Entry (LE legacy pairing)	Passkey Entry	Just Works	Passkey Entry (LE legacy pairing)
		Numeric Comparison (LE Secure Connections)			Numeric Comparison (LE Secure Connections)

The pairing events and the API used for the response, depend on selected pairing method.

- **Just Works, OOB**
  - Application is notified of no events.

- **Passkey Entry**
  - **[Input device]**
    - *BLE_GAP_EVENT_PASSKEY_ENTRY_REQ* event which requires to input 6-digit number is notified to an application. If the application receives the event and the remote device displays a 6-digit number, the application inputs the number by *R_BLE_GAP_ReplyPasskeyEntry* API.
  - **[Display device]**
    - It is necessary to display (e.g., on terminal emulator via UART) 6-digit number when *BLE_GAP_EVENT_PASSKEY_DISPLAY_REQ* event is received.

- **Numeric Comparison**
  - *BLE_GAP_EVENT_NUM_COMP_REQ* event which requires to check whether the number displayed on both devices are same. If the application receives the event, display the number (e.g., on terminal emulator via UART). After checking the number displayed on the remote device, send the result by *R_BLE_GAP_ReplyNumComp* API.

BP: When using except for OOB pairing methods, having the UX/UI inform the end-user that there are certain security or privacy risks is beneficial in mitigating security or privacy risks.

8.1.7 Key exchange

After the completion of the pairing method, both devices exchange keys. The link with the remote device is encrypted before key exchange and the completion is notified by *BLE_GAP_EVENT_ENC_CHG* event.

When the keys are distributed from the remote device, *BLE_GAP_EVENT_PEER_KEY_INFO* event is notified. Refer to section 8.2.1 for storing the keys received in the event.

When the local device is required to distribute the keys, user application is notified of *BLE_GAP_EVENT_EX_KEY_REQ* event. The local device responds to the request with *R_BLE_GAP_ReplyExKeyInfoReq* API. An example of the response to the key distribution request is shown in Code 38.

```
/* GAP Callback */
void gap_cb(uint16_t event_type, ble_status_t event_result, st_ble_evt_data_t * p_event_data)
{
    /** some code is omitted  **/
    case BLE_GAP_EVENT_EX_KEY_REQ :
        {
            st_ble_gap_conn_hdl_evt_t * p_param;
            p_param = (st_ble_gap_conn_hdl_evt_t *)p_event_data->p_param;
            R_BLE_GAP_ReplyExKeyInfoReq(p_param->conn_hdl);
        }
    break;
    /** some code is omitted  **/
```

Code 38. Sample of responding to a key distribute request

Automatically perform above procedure when *RM_BLE_ABS_Open* API used to open BLE module.

8.1.8 Completion of pairing

When the pairing has been completed, user application is notified of the *BLE_GAP_EVENT_PAIRING_COMP* event. If the pairing is successful, the event result is *BLE_SUCCESS*(0x0000). Any other value indicates a pairing failure.

If pairing is not completed within 30 seconds,
result: BLE_ERR_SMP_LE_TO(0x2011)
will be notified. Try pairing again.

If the bonding information about the remote device is lost, but the information remains in the Resolving List,
result: BLE_ERR_SMP_LE_DHKEY_CHECK_FAIL(0x200B)
will be notified. Delete the information about that device from the Resolving List as well using R_BLE_GAP_ConfRslvList().

If the bonding information was lost and the encryption could not be requested,
result: BLE_ERR_SMP_LE_LOC_KEY_MISSING(0x2014)
will be notified. Refer to "9.3.1 Request Encryption".

BP: If the pairing procedure fails, a waiting interval must elapse before initiating the next pairing with the same remote address. The wait interval increases exponentially with each repeated pairing procedure failure (maximum wait interval is implementation dependent). Introducing a wait interval reduces the ability of an attacker to repeatedly attempt the pairing procedure using different keys.

8.2 Bonding

Bonding is procedure which store the keys exchanged during pairing procedure to non-volatile area (e.g., Data Flash). When bonding process has done, pairing does not need to be done in reconnecting with a paired device. Figure 38 shows the bonding procedure.

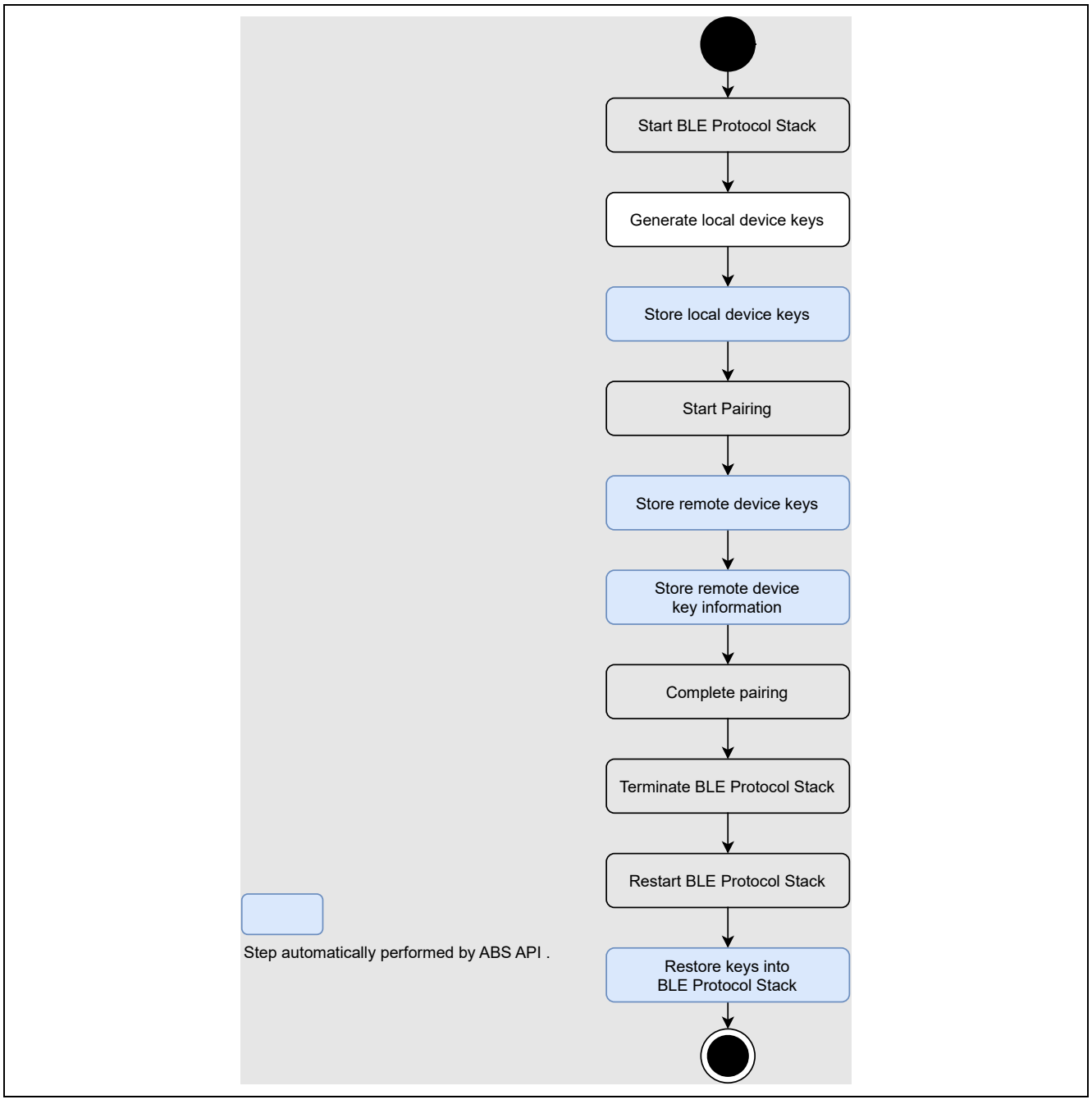


Figure 38. Boding procedure

8.2.1 Store remote device keys

Local device can store remote device keys and key information included in following events to Data Flash.

BLE_GAP_EVENT_PEER_KEY_INFO (key)
BLE_GAP_EVENT_PAIRING_COMP (key information)

If *RM_BLE_ABS_StartAuthentication* API is used to perform pairing and *Store security data* option on BLE module are enabled, the keys received by *BLE_GAP_EVENT_PEER_KEY_INFO* event and the key information received by *BLE_GAP_EVENT_PAIRING_COMP* event are automatically stored to Data Flash. The *Data Flash Block for Security Data* configuration on properties of BLE module specify which block of Data Flash used for storing key information. Refer to *BLE sample application (R01AN5402)* chapter 4 about structure of key information in the Data Flash.

If the Abstraction API is not used or *Store security data* option on BLE module are disabled, the keys received by *BLE_GAP_EVENT_PEER_KEY_INFO* event and the key information received by *BLE_GAP_EVENT_PAIRING_COMP* event are not stored. If the user wants to store the keys, it is necessary to implement the functionality by themselves as following.

```
case BLE_GAP_EVENT_PAIRING_COMP :
    {
        if(BLE_SUCCESS == event_result)
        {
            st_ble_gap_pairing_info_evt_t * p_param;
            p_param = (st_ble_gap_pairing_info_evt_t *)p_event_data->p_param;
            /* Add code storing p_param->auth_info into the Data Flash. */
        }
    }
    break;

case BLE_GAP_EVENT_PEER_KEY_INFO :
    {
        st_ble_gap_peer_key_info_evt_t * p_param;
        p_param = (st_ble_gap_peer_key_info_evt_t *)p_event_data->p_param;
        /* Add code storing p_param->key_ex_param into the Data Flash. */
    }
    break;
```

Code 39. Sample of storing received keys

If you want to resolve the bonded remote device address, you have to also register the remote device IRK and Identity Address to Resolving List.

Information (remote device keys) are stored in RAM and Data Flash. Bonding information can be stored in RAM up to BLE_ABS_CFG_RF_CONNECTION_MAXIMUM and stored in Data Flash up to BLE_ABS_CFG_NUMBER_BONDING. A bonding information is stored in order of bonding and if a new one is stored over the upper limit, the stack library deletes the oldest one and stores the new one according to the following policies.

- In RAM, the oldest bonding information of which the device is not connected is first automatically deleted.
- In Data Flash, the oldest bonding information is first automatically deleted regardless of the connection state.

If you do not want to automatically delete the bonding information, you have to check a bonding except for desired device and do not allow to bond beyond the desired bonding number in your application. It is recommended that BLE_ABS_CFG_RF_CONNECTION_MAXIMUM and BLE_ABS_CFG_NUMBER_BONDING are set to the same number and the desired bonding number + 1. An example that the desired bonding number is 2 (BLE_ABS_CFG_RF_CONNECTION_MAXIMUM=3, BLE_ABS_CFG_NUMBER_BONDING =3) is shown in **Figure 42** and **Figure 43**. The changes are shown in bold text. Figures 42 and 43 case is that application does not check about number of bonded devices. Therefore, bonding information of device A automatically deleted. To avoid such a scenario, you need to reject connection request (e.g., send disconnection) when number of bonded devices reached desired bonding number in your own application.

To allow connection request from bonded device, you can use Whitelist feature like following.
When number of bonded devices reached desired bonding number, Register identity address of bonded device to Whitelist and,

- Peripheral case
  Apply advertising filter policy as enable when peripheral perform adverting.
- Central case
  Apply initiator filter policy as enable when central perform initiating.

Refer to section 8.2.5 about Whitelist feature.

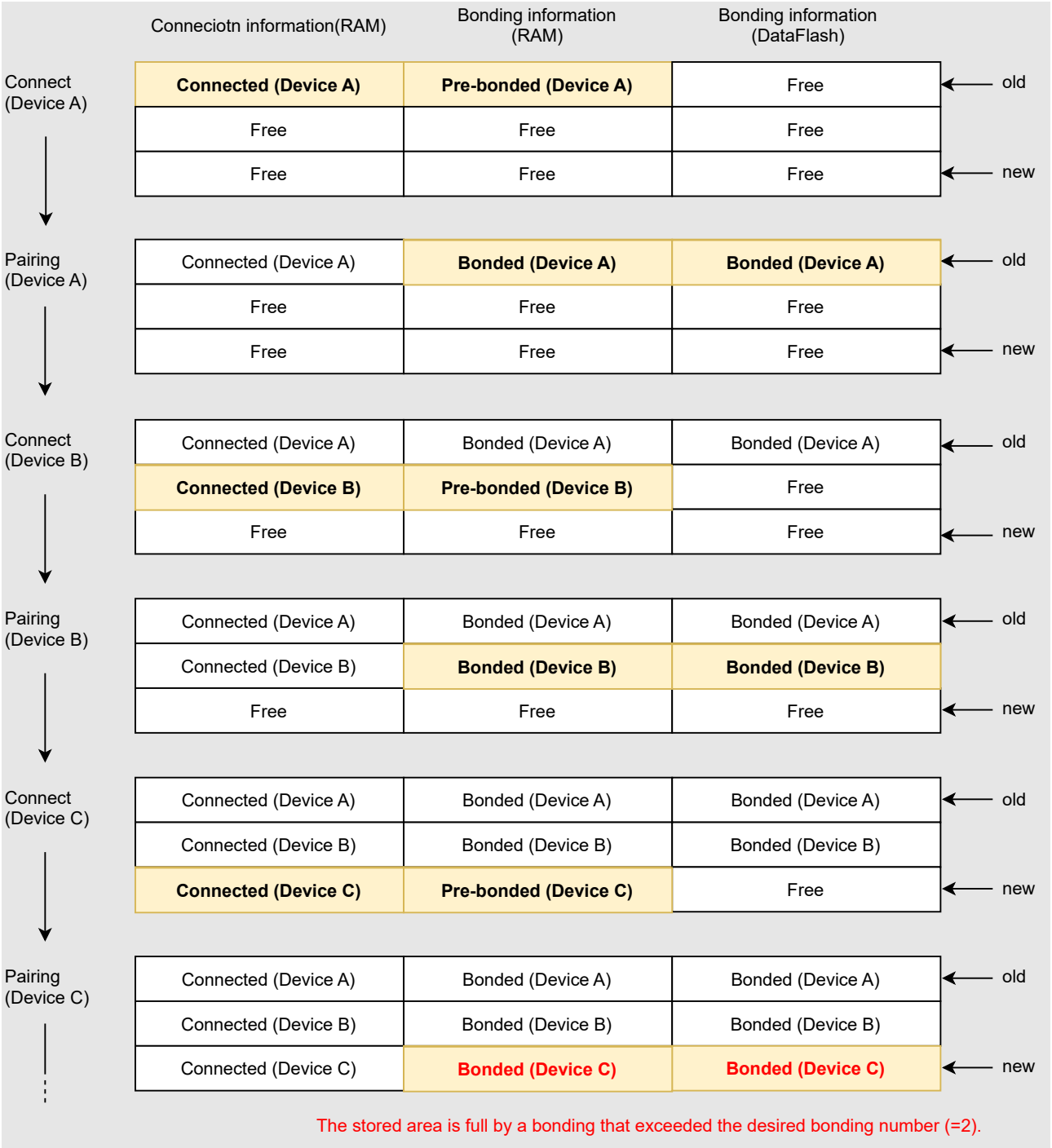


Figure 39. Boding information management in RAM, Data Flash (1)

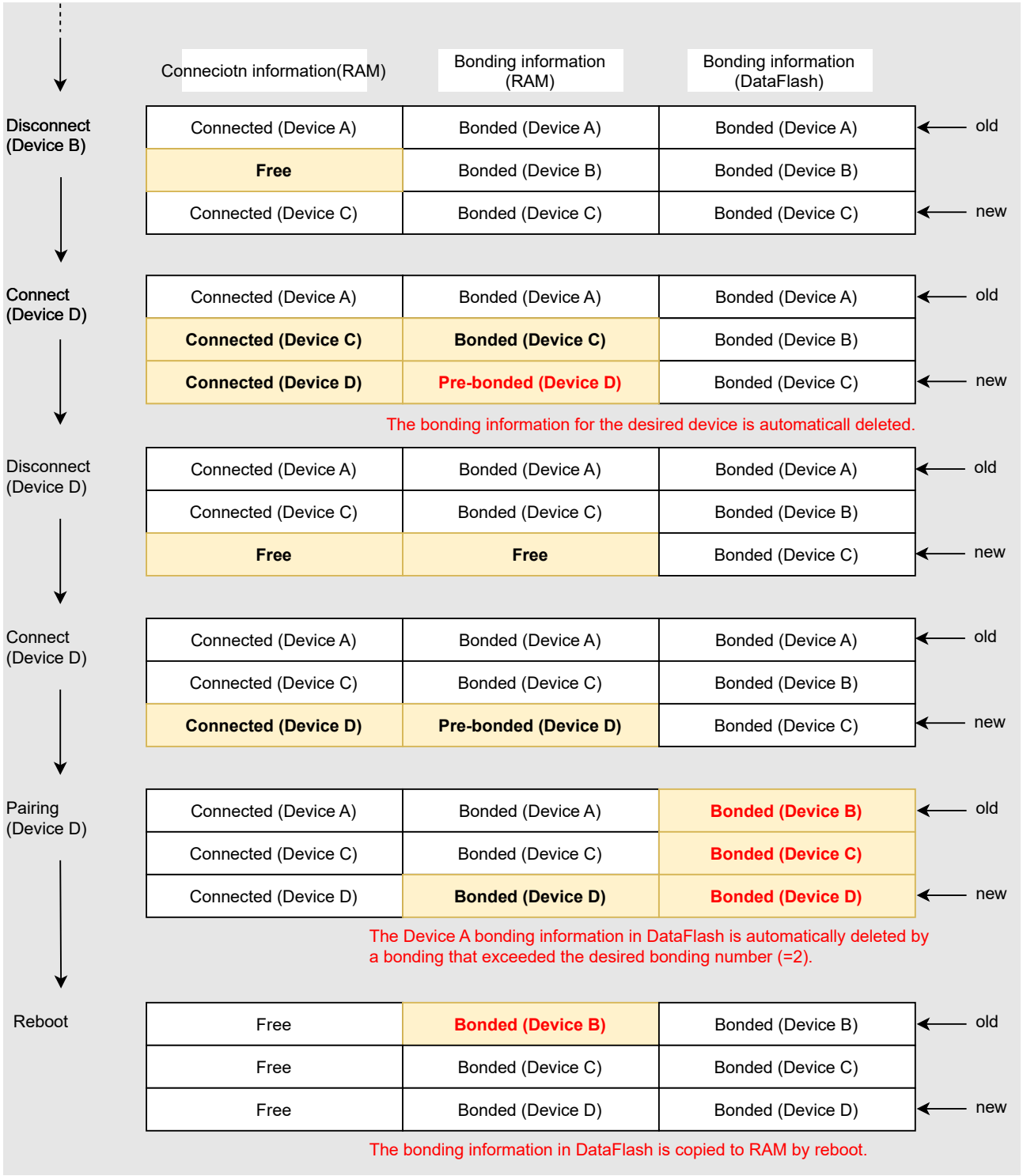


Figure 40. Boding information management in RAM, Data Flash (2)

8.2.1.1 Bonding information in RAM

Because bonding information in RAM is managed in the same area as the connection information, even if pairing or bonding is not done, the oldest bonding information of which the device is not connected is automatically deleted, and the area is released.

For more information about how to reconfigure the deleted bonding information from RAM, see section “8.2.3 Reset the stored keys”.

8.2.1.2 Bonding information in Data Flash

If a bonding information is stored in Data Flash exceeding BLE_ABS_CFG_NUMBER_BONDING, the oldest bonding information is automatically deleted regardless of the connection status and the new one is overwritten as shown in **Figure 39**, **Figure 40**.

If you do not want to automatically delete the bonding information in Data Flash, your application has to make sure that number of bonding does not exceed BLE_ABS_CFG_NUMBER_BONDING configuration. You have to implement procedure in your own application when additional connection request has happened (e.g., delete existing bonding information). For more information about how to delete the bonding information, see section “8.2.4 Delete the stored keys”.

8.2.2 Store local device keys

If the local device uses the privacy feature, the local IRK and the identity address are registered by *R_BLE_GAP_SetLocIdInfo* API or *RM_BLE_ABS_SetLocalPrivacy* API. When *RM_BLE_ABS_SetLocalPrivacy* API is used and *Store Security Data* configuration on properties of BLE module is enabled, the local device IRK generated by *RM_BLE_ABS_SetLocalPrivacy* API and the identity address are automatically stored in the Data Flash. The user can also use *RM_BLE_ABS_ImportKeyInformation* API to manually store the local device keys to the Data Flash.

8.2.3 Reset the stored keys

When the BLE Protocol Stack restarts, the stored keys in the device need to be restore to the stack by *R_BLE_GAP_SetBondInfo* API. If *RM_BLE_ABS_Open* API is used and *Store Security Data* configuration on properties of *BLE Abstraction Driver on rm_ble_abs* is enabled, the stored keys will be automatically restored to the BLE Protocol Stack in restarting. The user can also use the *RM_BLE_ABS_ExportKeyInformation* API to manually restore the keys.

When you want to resolve the bonded remote device address, you must also reconfigure the IRK and the Identity Address of the remote device to Resolving List. If *RM_BLE_ABS_Open* API is used and *Store Security Data* configuration on properties of *BLE Abstraction Driver on rm_ble_abs* is enabled, the IRK and the Identity Address of the remote device will be automatically restored to the Resolving List in restarting.

8.2.4 Delete the stored keys

When the bonding information in the remote device deleted, it is also necessary to delete the bonding information in the local device. You can use *RM_BLE_ABS_DeleteBondInformation* API or *R_BLE_GAP_DeleteBondInfo* API to delete the bonding information from local device.

Likewise, when the bonding information (the remote device security data) in the local device deleted, it is also necessary to delete the bonding information in the remote device.

If only one of the devices deleted the bonding information, the following issues will occur, and the security feature cannot be used.

- The device cannot access the GATT service that the encryption security requirement is configured to due to the loss of LTK.
- The device cannot resolve the remote device address and cannot connect to the remote device Identity Address due to the loss of IRK.

The bonding information that needs to be deleted is shown in **Figure 41** if ether a RA4W1 device or a remote device deletes the bonding information.

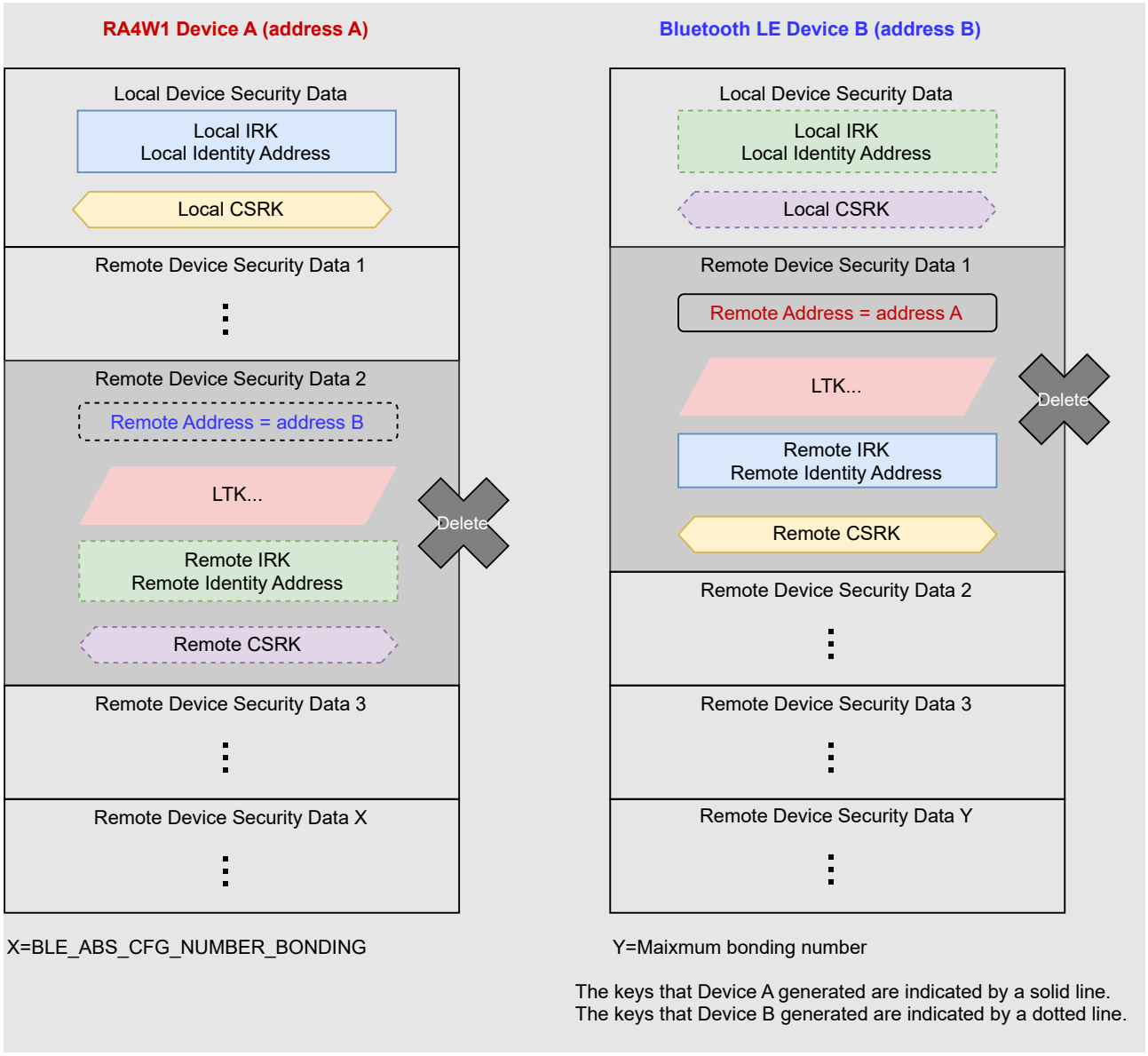


Figure 41. The bonding information that needs to be deleted if either device deletes it

If the bonded remote device address was resolved, you must also delete the device IRK and Identity Address from Resolving List.

8.2.5 Filtering remote devices after bonding

If you want to connect or communicate to the bonded device, you should register the remote device address to the Whitelist. Whitelist can register 4 devices in case of All features library and register 8 devices in case of Balance and Compact library. For more information about the use of White List, see 4.1.1 and 4.2.1.2 for Peripheral and 5.3.1 and 6.4 for Central.
When the RA4W1 device reboots, it is necessary to reconfigure the remote device addresses to Whitelist. If *RM_BLE_ABS_Open* API is used and *Store Security Data* configuration on properties of *BLE Abstraction Driver on rm_ble_abs* is enabled, the remote device addresses will be automatically reconfigured to the Whitelist using the bonding information of Data Flash while restarting.
Note that the number of bonding information in the data flash exceeding BLE_GAP_WHITE_LIST_MAX_ENTRY macro will not be reconstructed into the Whitelist. In that case, you must choose which bonding information is needed to register in your own application.
If the local device deletes the stored key, delete the remote device address from Whitelist.

8.3 Encryption

Bluetooth LE enables secure communication by encrypting data packets. The encryption in reconnection after pairing uses the key exchanged by pairing.

BP: If the encryption procedures fails, do not attempt to circumvent the failure or connect by other means. Switching to less secure options for convenience is the desired outcome for attackers. Failures at any stage should be aware of the potential for attackers to gain access through vulnerabilities or repeated attempts.

8.3.1 Request Encryption

When reconnecting with a paired or bonded remote device, the local device will request encryption using one of the following APIs.

- *RM_BLE_ABS_StartAuthentication*
- *R_BLE_GAP_StartEnc*

If the encryption has been completed successfully,
BLE_GAP_EVENT_ENC_CHG
Result: BLE_SUCCESS (0x0000)
event will be notified.

Depending on the remote device implementation, the remote device does not respond to an encryption request from a peripheral device. In this case, if the above API is called, pairing may start. The encryption request sequence is shown in Figure 42 and Figure 43.

1. Encryption request from local device(Central)

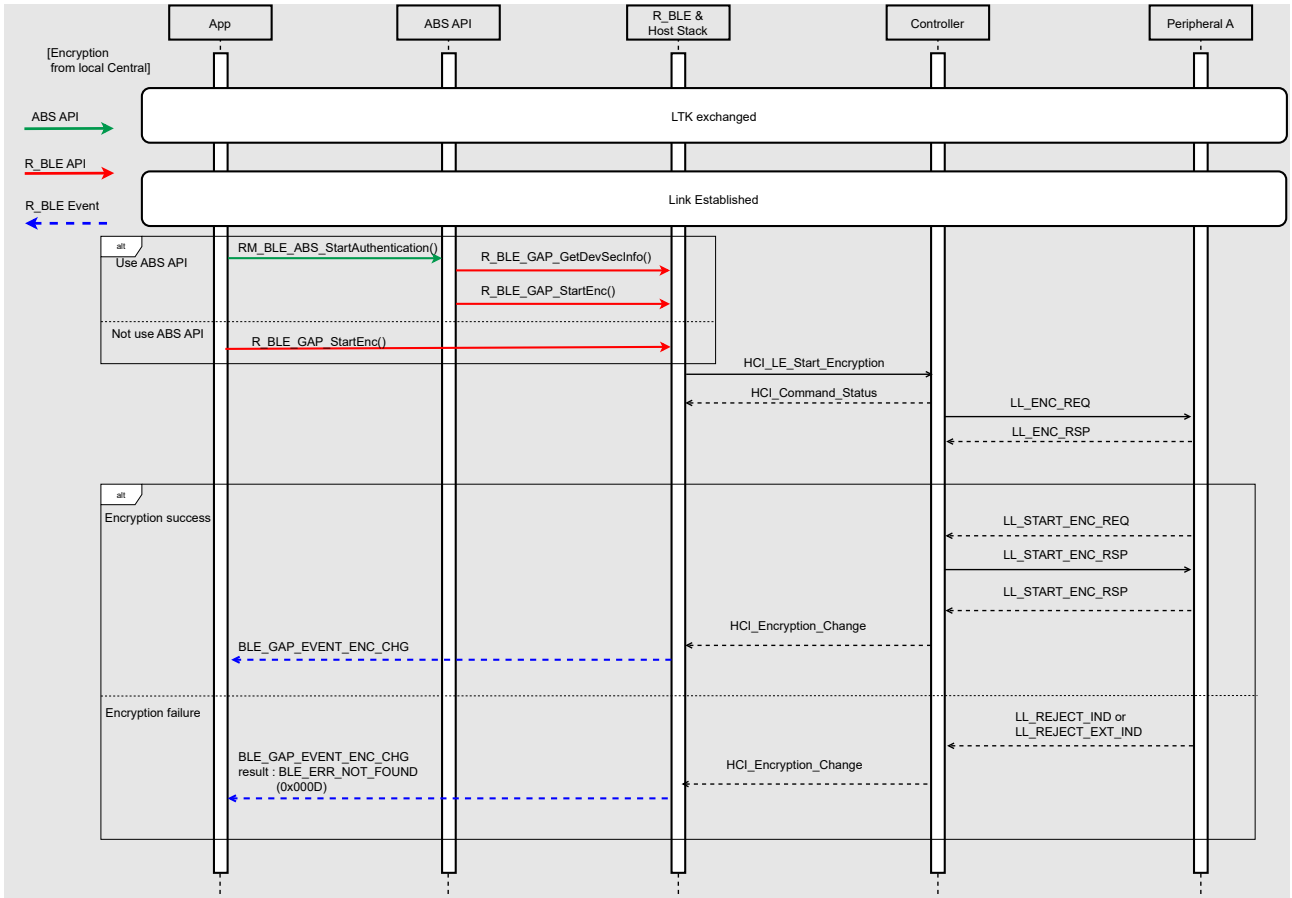


Figure 42. Sequence of encryption request from local device(Central)

If the remote device (Peripheral) lost the bonding information before the local device (Central) sends an encryption request,
BLE_GAP_EVENT_ENC_CHG
Result: BLE_SUCCESS (0x000D)
event will be notified, and the encryption will fail. In that case, although the link is still established, the device (Central) cannot access the service for which the encryption security requirement was configured.
The local device (Central) also needs to delete the bonding information and perform pairing procedures again to access the service.

If the local device (Central) lost the bonding information before sending an encryption request, the local device (Central) will send a pairing request and then start encryption.

2. Encryption request from local device(Peripheral)

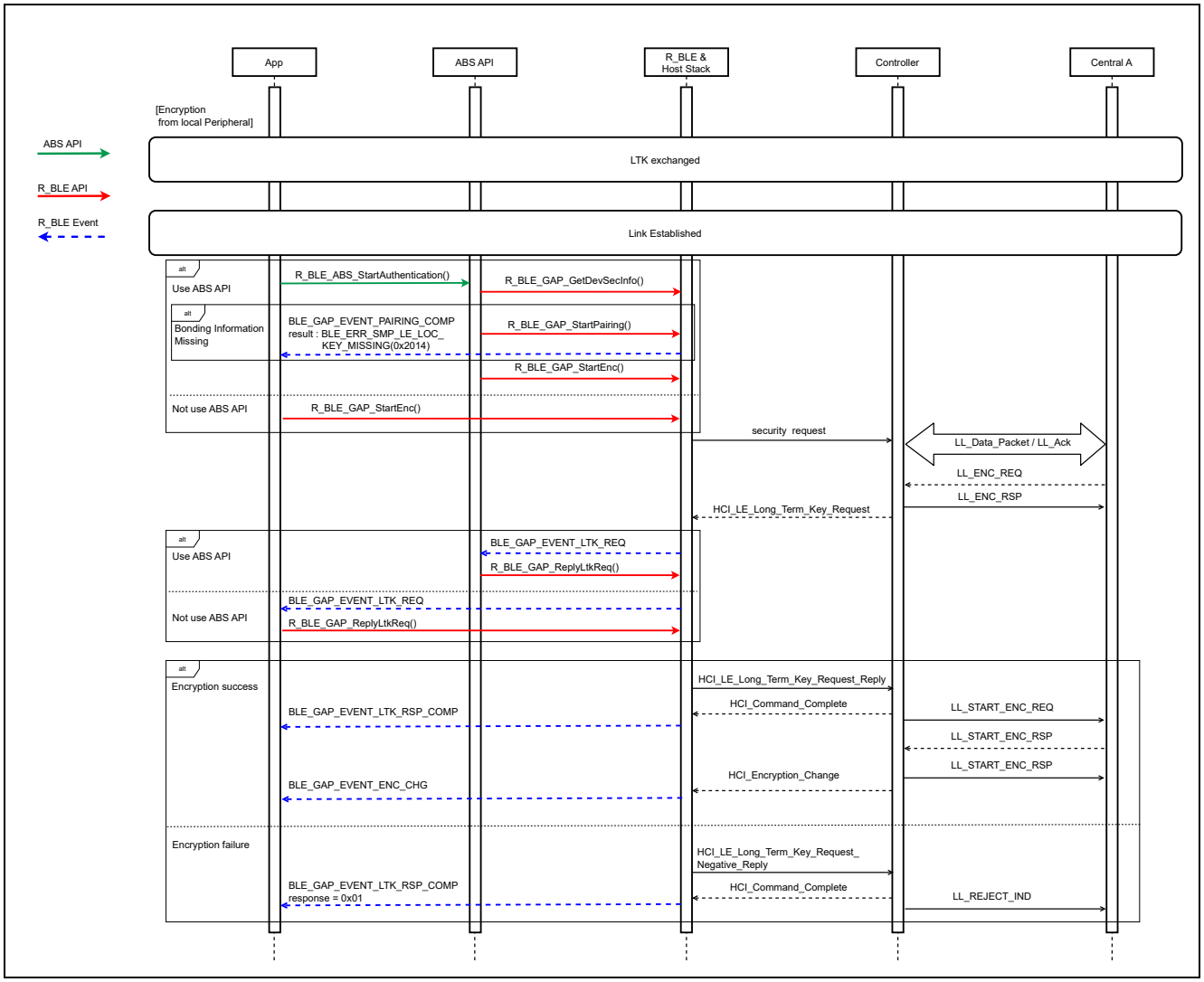


Figure 43. Sequence of encryption request from local device(Peripheral)

If the local device (Peripheral) lost the bonding information before sending an encryption request, BLE_GAP_EVENT_PAIRING_COMP
Result: BLE_ERR_SMP_LE_LOC_KEY_MISSING(0x2014)
event will be notified, and the encryption will fail. In that case, the device (Central) cannot access the service for which the encryption security requirement was configured.
The remote device (Central) also needs to delete the bonding information and perform pairing procedures again to access the service.

If the remote device (Central) lost the bonding information before the local device (Peripheral) sends an encryption request, the remote device (Central) will send a pairing request and then start encryption.

8.3.2 Respond to an encryption request

When receiving an encryption request from a remote device, user application will be notified of *BLE_GAP_EVENT_LTK_REQ* event. Call *R_BLE_GAP_ReplyLtkReq* API with the parameter received in the event for responding to the encryption request. When the LTK exchange is successfully completed, user application will be notified of *BLE_GAP_EVENT_LTK_RSP_COMP* event. If the encryption fails, remove the remote device LTK and perform pairing again.

Automatically perform above procedure when *RM_BLE_ABS_Open* API is used to open the BLE module.

When reconnecting with a paired remote device, the local device needs to respond to the encryption request. The sequence of response to an encryption request is shown in Figure 44 and Figure 45.

1. Response to an encryption request from remote device(Central)

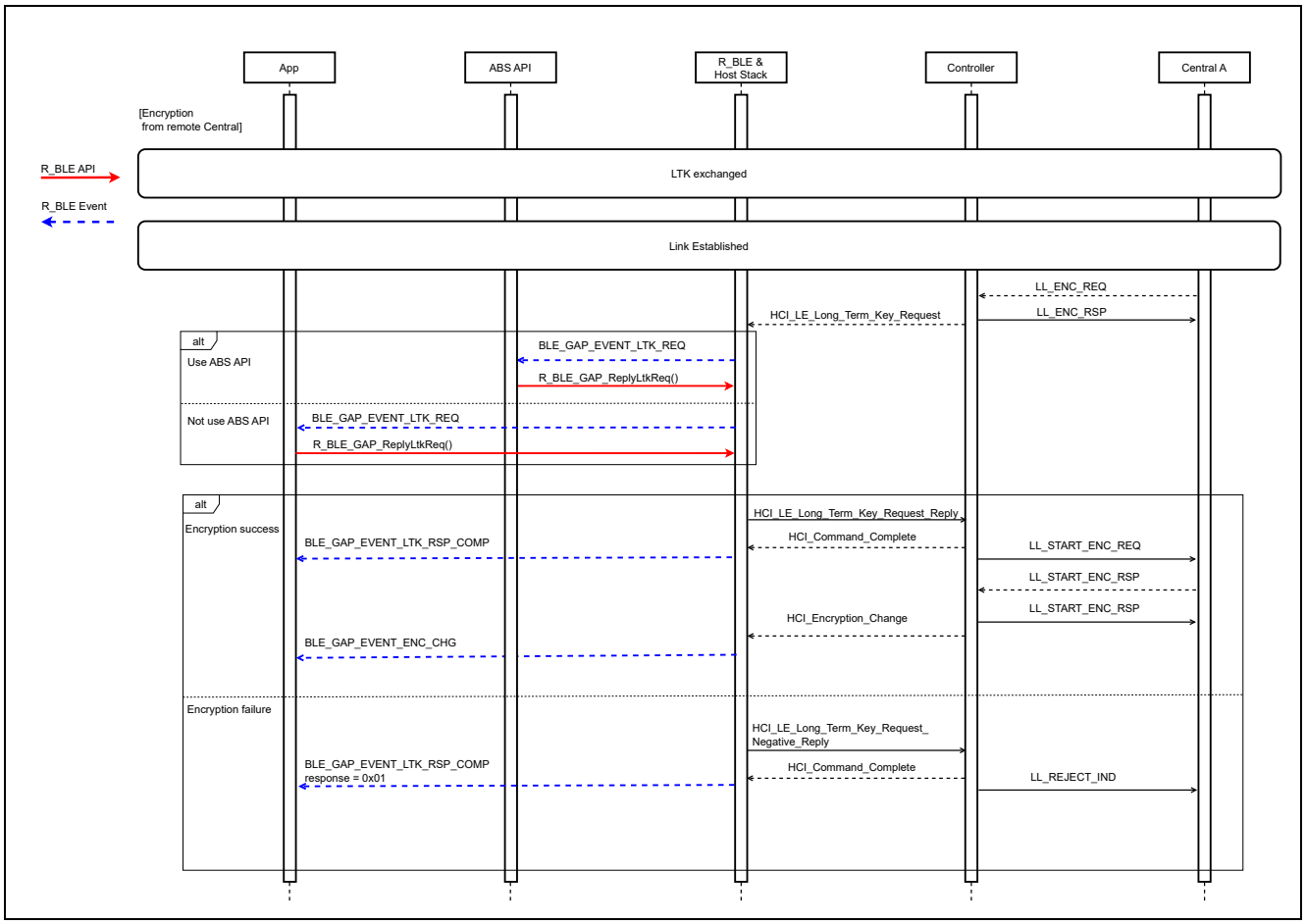


Figure 44. Sequence of response to an encryption request from remote device(Central)

When receiving an encryption request from a remote device, BLE_GAP_EVENT_LTK_REQ event will be notified. Local device (Peripheral) needs to reply to the encryption request by using R_BLE_GAP_ReplyLtkReq API with the parameter of BLE_GAP_EVENT_LTK_REQ event as an argument. When the LTK exchange is successfully completed, BLE_GAP_EVENT_LTK_RSP_COMP event will be notified.
If the encryption has been completed successfully,
BLE_GAP_EVENT_ENC_CHG
Result: BLE_SUCCESS (0x0000)
event will be notified.

If the local device (Peripheral) lost the bonding information before the remote device (Central) sends an encryption request and then the local device (Peripheral) calls R_BLE_GAP_ReplyLtkReq,
BLE_GAP_EVENT_LTK_RSP_COMP
response (Event data): 0x01
event will be notified, and the encryption will fail. In which case, the device (Central) cannot access the service that the encryption security requirement was configured.

The remote device (Central) also needs to delete the bonding information and perform pairing procedures again to access the service.

If the remote device (Central) lost the bonding information before sending an encryption request, the remote device (Central) will send a pairing request and then start encryption.

When the local device (Peripheral) connects to a smart phone (Central) for the first time, the local device is required to respond to a pairing request but is not required to respond to an encryption request. When the local device (Peripheral) connects to a paired smart phone, the local device (Peripheral) is required to respond to an encryption request.

An example of an encryption request event from the remote device (Central) and the response API is shown in Code 40.

```
/* GAP Callback */
void gap_cb(uint16_t event_type, ble_status_t event_result,
            st_ble_evt_data_t * p_event_data)
{
    /** some code is omitted **/
    /* Receive encryption request from a remote device */
    case BLE_GAP_EVENT_LTK_REQ :
        {
            st_ble_gap_ltk_req_evt_t * p_param;
            p_param = (st_ble_gap_ltk_req_evt_t *)p_event_data->p_param;
            R_BLE_GAP_ReplyLtkReq(p_param->conn_hdl, p_param->ediv,
                                  p_param->p_peer_rand, BLE_GAP_LTK_REQ_ACCEPT);
        }
        break;
        /** some code is omitted **/
```

Code 40. Sample of responding an encryption request in the event

2. Response to an encryption request from remote device(Peripheral)

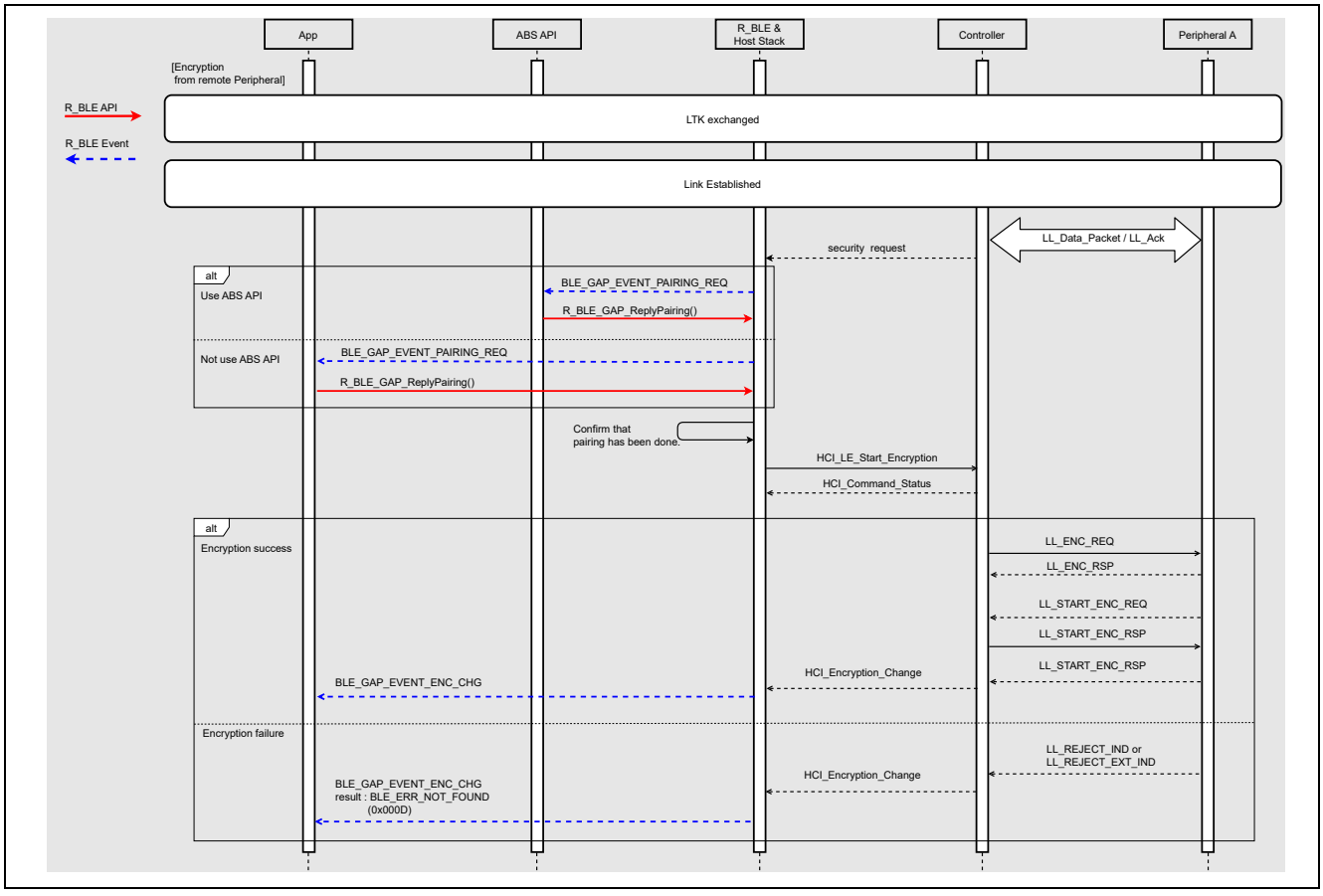


Figure 45. Sequence of response to an encryption request from remote device(Peripheral)

When receiving an encryption request from a remote device (Peripheral),
BLE_GAP_EVENT_PAIRING_REQ event will be notified. Local device (Central) needs to reply to the encryption request by using R_BLE_GAP_ReplyPairing API with the parameter of BLE_GAP_EVENT_PAIRING_REQ event as an argument.
If the encryption has been completed successfully,
BLE_GAP_EVENT_ENC_CHG
Result: BLE_SUCCESS (0x0000)
event will be notified.

If the remote device (Peripheral) lost the bonding information before sending an encryption request,
BLE_GAP_EVENT_ENC_CHG
Result: BLE_SUCCESS (0x000D)
event will be notified, and the encryption will fail. In that case, the device (Central) cannot access the service that the encryption security requirement has been configured.
The local device (Central) also needs to delete the bonding information and perform pairing procedures again to access the service.

If the local device (Central) lost the bonding information before a remote device (Peripheral) sends an encryption request, the local device (Central) sends a pairing request and then start encryption.

An example of an encryption request from a remote device (Peripheral) event and the response API is the same as Code 40.

8.4 Privacy

The privacy feature allows the local device to periodically change the address used by Advertising, Scan Request and Connection Request to another address to avoid being tracked by other devices. There are two privacy modes: Network Privacy Mode and Device Privacy Mode. In Device Privacy Mode, if the local device uses RPA (Resolvable Private Address), the local device will accept Advertising, Scan Request and Connection Request regardless of the remote address type. In Network Privacy Mode, if the local device uses RPA, the local device will not accept Advertising, Scan Request and Connection Request including identity address of the remote device. By default, the local device is Network Privacy Mode. RPA is generated and resolved by Resolving List in the local device.

Up to 8 sets of the IRK (Remote IRK) and Identity Address (ID) of the remote device and the IRK (Local IRK) of the local device can be registered in the Resolving List.

If the local device generates an RPA to initiate Advertising, scanning, or connection, the Local IRK and the ID of the remote device are registered to the Resolving List in advance, and the Resolving List is searched by the specified ID of the remote device. Details on how to generate RPA are given in 8.4.1 and 8.4.2.

When resolving RPA included in Advertising, Scan Request, and Connection Request from a remote device, you need to register the Remote IRK and ID obtained by pairing procedure with the remote device together with the Local IRK to the Resolving List. The local device will search a set which the received RPA match the RPA calculated from the IRK and ID of the Resolving List. Details on how to resolve RPA are given in 8.4.3 and 8.4.4.

Figure 46 shows an image of generating and resolving RPA using the Resolving List in Advertising.

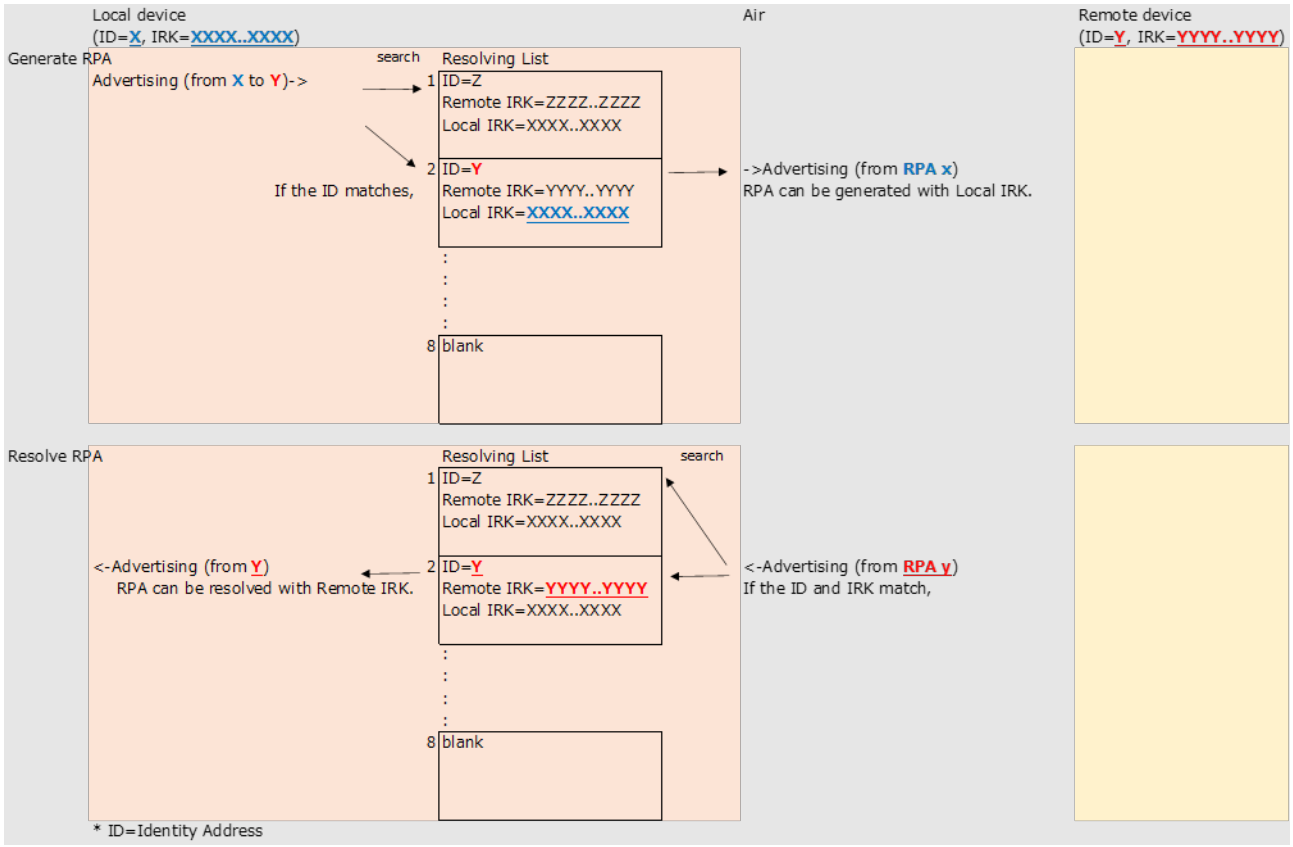


Figure 46. Image of Resolving List

The pairing procedure in an application is shown in Figure 47. The following sections describe the details of pairing steps.

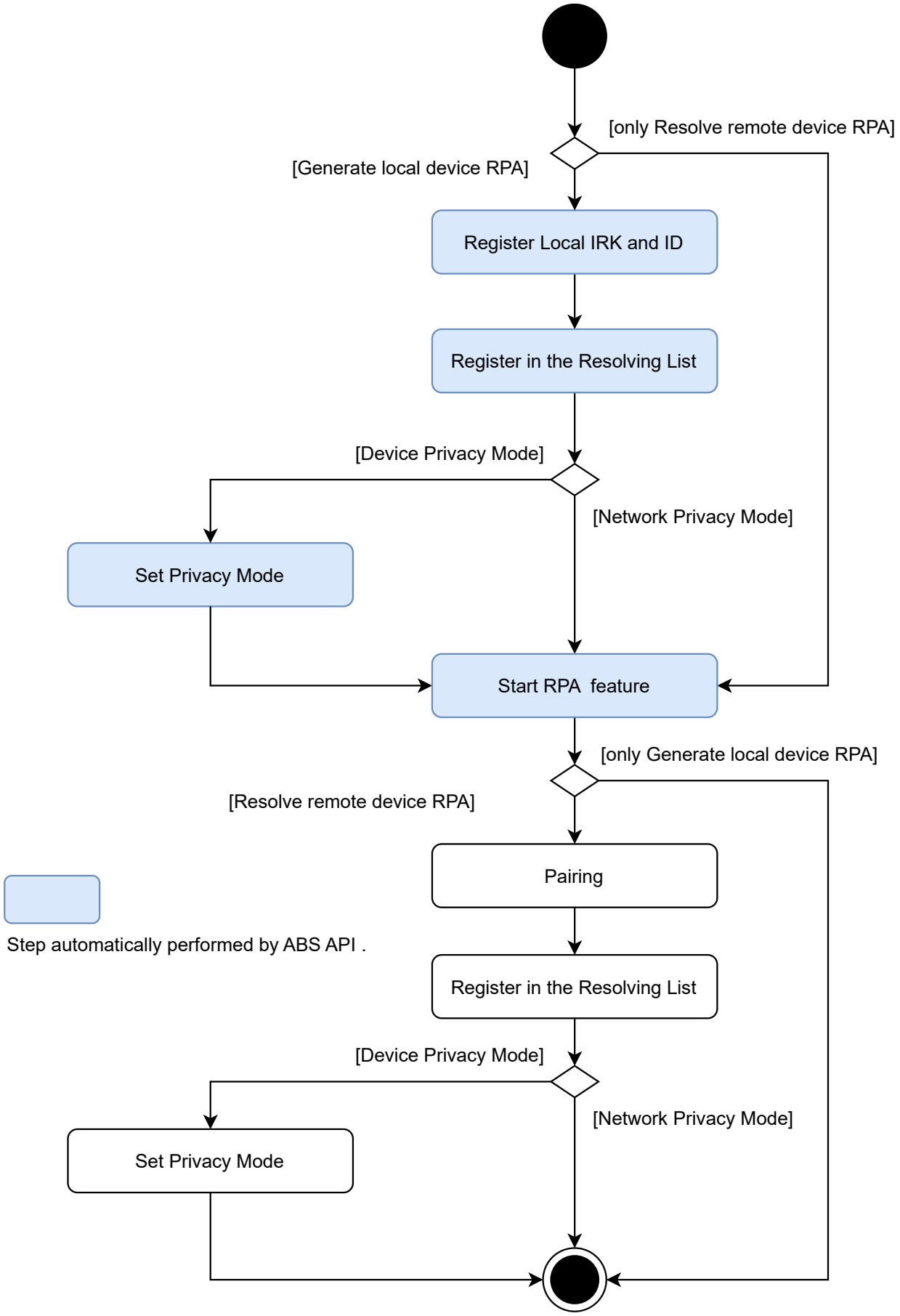


Figure 47. Privacy procedure in application

8.4.1 Privacy with abstraction API

This section describes the procedure to use privacy feature by using abstraction API. If the local device uses RPA by using abstraction API, perform the following steps.

1. Open BLE module by using *RM_BLE_ABS_Open* API. This API reads the bonding information (remote IRK and remote identity address, etc.) of the bonded devices from Data Flash and sets it to resolving list.

2. Check whether the local IRK has been generated by using *RM_BLE_ABS_ExportKeyInformation* API. If the local IRK is not generated, *FSP_ERR_NOT_INITIALIZED* is returned. In that case, specify *NULL* as the second argument and call *RM_BLE_ABS_SetLocalPrivacy* API. The *RM_BLE_ABS_SetLocalPrivacy* API automatically generates a local IRK, register that IRK to a resolving list and store the key in a Data flash when *Data Flash Block for Security Data* configuration on properties of BLE module is enabled. If the local IRK is generated, *FSP_SUCCESS* is returned. In that case, this step can be skipped.

3. The local IRK is stored in the memory specified in the third argument of *RM_BLE_ABS_ExportKeyInformation* API. In that case, specify the obtained local IRK as the second argument and call *RM_BLE_ABS_SetLocalPrivacy* API.

4. BLE_GAP_EVENT_RPA_EN_COMP event will be notified to the user application when the RPA configuration is completed.

An example of the procedures 1 - 4 is shown in Code 41. If the Local device generates RPA, destination address must match the Identity Address in the Resolving List.

```
/** some code is omitted  **/
st_ble_dev_addr_t gs_rem_bd_addr;

/* Advertising parameters */
static ble_abs_legacy_advertiding_parameter_t gs_adv_param =
{
    /* TODO: Modify advertise parameters. */
    .p_peer_address              = &gs_rem_bd_addr,
    .own_bluetooth_address_type  = BLE_GAP_ADDR_RPA_ID_PUBLIC,
    /** some code is omitted  **/
};
/** some code is omitted  **/

/* GAP Callback */
void gap_cb(uint16_t event_type, ble_status_t event_result,
            st_ble_evt_data_t * p_event_data)
{
    /** some code is omitted **/

        case BLE_GAP_EVENT_LOC_VER_INFO:
        {
            ble_device_address_t lc_id_addr;
            uint8_t lc_irk[BLE_GAP_IRK_SIZE];

            uint8_t irk_check[BLE_GAP_IRK_SIZE];
            memset(irk_check, 0x00, BLE_GAP_IRK_SIZE);

            fsp_err = RM_BLE_ABS_ExportKeyInformation(g_ble_abs0.p_ctrl, &lc_id_addr, lc_irk, NULL);

            if (FSP_ERR_NOT_INITIALIZED == fsp_err || 0 == memcmp(lc_irk, irk_check, BLE_GAP_IRK_SIZE))
            {
                /* If local IRK is not set */
                RM_BLE_ABS_SetLocalPrivacy(g_ble_abs0.p_ctrl, NULL,
                                           (BLE_GAP_NET_PRIV_MODE | (BLE_GAP_ADDR_PUBLIC << 4)));
            }
            else
```

```
            {
                /* If local IRK is set */
                RM_BLE_ABS_SetLocalPrivacy(g_ble_abs0.p_ctrl, lc_irk,
                      BLE_GAP_NET_PRIV_MODE | (BLE_GAP_ADDR_PUBLIC << 4)));
            }
        } break;

        case BLE_GAP_EVENT_RPA_EN_COMP:
        {
             /* Start advertising */
             RM_BLE_ABS_StartLegacyAdvertising(&g_ble_abs0_ctrl, &gs_adv_param);
        } break;
    /** some code is omitted **/
```

Code 41. Sample of load local IRK

RM_BLE_ABS_SetLocalPrivacy API can also specify privacy mode and identity address as third argument.

Table 46. Parameters used for the privacy feature

Value	Description
BLE_GAP_NET_PRIV_MODE | (BLE_GAP_ADDR_PUBLIC << 4)	Privacy mode : Network privacy mode. Identity address : Register public address to resolving list.
BLE_GAP_NET_PRIV_MODE | (BLE_GAP_ADDR_RAND << 4)	Privacy mode : Network privacy mode. Identity address : Register static address to resolving list.
BLE_GAP_DEV_PRIV_MODE | (BLE_GAP_ADDR_PUBLIC << 4)	Privacy mode : Device privacy mode. Identity address : Register public address to resolving list.
BLE_GAP_DEV_PRIV_MODE | (BLE_GAP_ADDR_RAND << 4)	Privacy mode : Device privacy mode. Identity address : Register static address to resolving list.

When the local device Advertising or Scan Request or Connection Request operation with specified the RPA as its own address, the packet includes the RPA. Refer to following about how to specify the RPA as its own address.

[Advertising]

It is necessary to configure *ble_abs_legacy_advertising_parameter_t* or *ble_abs_extended_advertising_parameter_t* according to section 4.1.2.

[Scan]

It is necessary to configure *ble_abs_scan_parameter_t* according to section 5.1.2.

[Connection]

It is necessary to configure *ble_abs_connection_parameter_t* according to section 6.1.2.

8.4.2 Privacy with GAP API

This section describes the procedure to use RPA by using GAP API. If the local device uses RPA by using GAP API, perform the following steps.

1. Check whether the local IRK has been generated by using *RM_BLE_ABS_ExportKeyInformation* API. If the local IRK is not generated, *FSP_ERR_NOT_INITIALIZED* is returned. In that case, it is necessary call *R_BLE_VS_GetRand* API to generate a random value (16 bytes). The random value will be notified in *BLE_VS_EVENT_GET_RAND* event. Register the random value to BLE module as local IRK by using *R_BLE_GAP_SetLocIdInfo* API. If local IRK is generated, *FSP_SUCCESS* is returned from *RM_BLE_ABS_ExportKeyInformation* API. In that case, it is necessary to register the obtained local IRK to BLE module by using *R_BLE_GAP_SetLocIdInfo* API.

2. Call *R_BLE_GAP_ConfRslvList* API to register the local IRK generated / obtained by step 1 to the resolving list. A set of identity address and IRK of a remote device needs to be registered to associate with the local device IRK. Set the Identity Address and IRK of the remote device to all 0x00 to associate with the local device IRK. The completion of this procedure is notified by *BLE_GAP_EVENT_RSLV_LIST_CONF_COMP* event.

3. Call *R_BLE_GAP_SetPrivMode* API to set the privacy mode. The completion of this procedure is notified by *BLE_GAP_EVENT_PRIV_MODE_SET_COMP* event.

4. Call *R_BLE_GAP_EnableRpa* API to enable the RPA generation and resolution. BLE_GAP_EVENT_RPA_EN_COMP event will be notified to the user application when the RPA configuration is completed.

An example of the procedures 1 - 4 is shown in Code 42. If the Local device generates RPA, destination address must match the Identity Address in the Resolving List.

```
/** some code is omitted  **/
st_ble_dev_addr_t gs_loc_bd_addr;
st_ble_dev_addr_t gs_rem_bd_addr;

/* Advertising parameters */
static ble_abs_legacy_advertiding_parameter_t gs_adv_param =
{
    /* TODO: Modify advertise parameters. */
    .p_peer_address              = &gs_rem_bd_addr,
    .own_bluetooth_address_type  = BLE_GAP_ADDR_RPA_ID_PUBLIC,
    /** some code is omitted  **/
};
/** some code is omitted  **/

/* Vendor Specific callback function */
void vs_cb(uint16_t event_type, ble_status_t event_result, st_ble_evt_data_t * p_data)
{
    switch(event_type)
    {
        /** some code is omitted  **/
        case BLE_VS_EVENT_GET_RAND :
        {
            st_ble_vs_get_rand_comp_evt_t * p_rand_param;
            p_rand_param = (st_ble_vs_get_rand_comp_evt_t *)p_data->p_param;
            R_BLE_GAP_SetLocIdInfo(&gs_loc_bd_addr, p_rand_param->p_rand);

            /* Set all zero remote address & remote IRK */
            st_ble_gap_rslv_list_key_set_t peer_irk;

            memset(peer_irk.remote_irk, 0x00, BLE_GAP_IRK_SIZE);
            peer_irk.local_irk_type = BLE_GAP_RL_LOC_KEY_REGISTERED;
            memset(gs_rem_bd_addr.addr, 0x00, BLE_BD_ADDR_LEN);
            gs_rem_bd_addr.type = BLE_GAP_ADDR_RPA_ID_PUBLIC;

            /* Add local IRK to resolving list */
            R_BLE_GAP_ConfRslvList(BLE_GAP_LIST_ADD_DEV, &gs_rem_bd_addr, &peer_irk, 1);
        }
        break;
        /** some code is omitted  **/
    }
}

/* GAP Callback */
void gap_cb(uint16_t event_type, ble_status_t event_result, st_ble_evt_data_t * p_data)
{
    switch(event_type)
    {
        /** some code is omitted  **/
        case BLE_GAP_EVENT_RSLV_LIST_CONF_COMP :
            {
                st_ble_gap_rslv_list_conf_evt_t * p_rslv_list_conf;
                p_rslv_list_conf = (st_ble_gap_rslv_list_conf_evt_t *)p_data->p_param;
                if(BLE_GAP_LIST_ADD_DEV == p_rslv_list_conf->op_code)
                {
                    uint8_t priv_mode;
                    priv_mode = BLE_GAP_NET_PRIV_MODE;

                    /* Set Network Privacy Mode. */
                    R_BLE_GAP_SetPrivMode(&gs_rem_bd_addr, &priv_mode, 1);
                }
            }
            break;

        case BLE_GAP_EVENT_PRIV_MODE_SET_COMP :
            {
                /* Enable RPA. */
                R_BLE_GAP_EnableRpa(BLE_GAP_RPA_ENABLED);
            }
            break;

        case BLE_GAP_EVENT_LOC_VER_INFO:
            {
```

```
                st_ble_gap_loc_dev_info_evt_t * ev_param;
                uint8_t lc_irk[BLE_GAP_IRK_SIZE];
                ev_param = (st_ble_gap_loc_dev_info_evt_t *)p_data->p_param;
                gs_loc_bd_addr = ev_param->l_dev_addr;

                gs_loc_bd_addr = ev_param->l_dev_addr;
                fsp_err = RM_BLE_ABS_ExportKeyInformation(g_ble_abs0.p_ctrl, & gs_loc_bd_addr,
                                                          lc_irk, NULL);

                /* If local IRK is not set */
                if (FSP_ERR_NOT_INITIALIZED == fsp_err
                        || 0 == memcmp(lc_irk, irk_check, BLE_GAP_IRK_SIZE))
                {
                        /* Generate IRK */
                        R_BLE_VS_GetRand(BLE_GAP_IRK_SIZE);
                }
                /* If local IRK is already generated */
                else
                {
                        /* Add local IRK to resolving list */
                        R_BLE_GAP_SetLocIdInfo(&gs_loc_bd_addr, lc_id_addr);

                        /* Dummy remote address & remote IRK */
                        st_ble_gap_rslv_list_key_set_t peer_irk;

                        memset(peer_irk.remote_irk, 0x00, BLE_GAP_IRK_SIZE);
                        peer_irk.local_irk_type = BLE_GAP_RL_LOC_KEY_REGISTERED;
                        memset(gs_rem_bd_addr.addr, 0x00, BLE_BD_ADDR_LEN);
                        gs_rem_bd_addr.type = BLE_GAP_ADDR_RPA_ID_PUBLIC;

                        R_BLE_GAP_ConfRslvList(BLE_GAP_LIST_ADD_DEV, &gs_rem_bd_addr, &peer_irk, 1);
                }
            } break;

        case BLE_GAP_EVENT_RPA_EN_COMP:
            {
                 /* Start advertising */
                 RM_BLE_ABS_StartLegacyAdvertising(&g_ble_abs0_ctrl, &gs_adv_param);
            } break;
        /** some code is omitted  **/
    }
}
```

Code 42. Prepare for using RPA in the local device with GAP API

When the local device Advertising, Scan Request, or Connection Request operation with specified the RPA as its own address, the packet includes the RPA.

[Advertising]

When setting the advertising parameters by *R_BLE_GAP_SetAdvParam* API, configure the parameters in Table 19.

[Scan]

When setting the scan parameters by *R_BLE_GAP_StartScan* API, configure the parameters in Table 27.

[Connection]

When create a connection by *R_BLE_GAP_CreateConn* API, configure the parameters in Table 30.

8.4.3 Resolve remote device RPA with abstraction API

RPA of Remote device is resolved according to the following procedures. If the local device resolves RPA by using abstraction API, perform the following steps.

1. Start RPA feature

Call *RM_BLE_ABS_SetLocalPrivacy* API to enable the RPA generation and resolution. The completion is notified by *BLE_GAP_EVENT_RPA_EN_COMP* event. If the local device does not use RPA, set a local IRK of the second parameter of *RM_BLE_ABS_SetLocalPrivacy* API to all zeros.

2. Pairing

Receive the remote device IRK and identity address by pairing. For more details about pairing, see "8.1 Pairing".

3. Register remote device key (IRK) and BD address

Call *R_BLE_GAP_ConfRslvList* API to register the remote device IRK and identity address in the Resolving List. The local device IRK is also registered at that time. If the local device does not use RPA, set the local device IRK to all 0x00 by setting the local_irk_type in st_ble_gap_rslv_list_key_set_t type array of third parameter to BLE_GAP_RL_LOC_KEY_ALL_ZERO.
BLE_GAP_EVENT_RSLV_LIST_CONF_COMP event notifies the user application that the registration is complete. If users want to store the remote device IRK and identity address to data flash, it is necessary implement the procedure by themselves.

4. Set Privacy Mode

If Network Privacy Mode which is the default is used, the procedure does not need to be done.
Call the *R_BLE_GAP_SetPrivMode* API to set the privacy mode.
BLE_GAP_EVENT_PRIV_MODE_SET_COMP event notifies user application of the completion.

5. Resolve RPA

After the procedures 1-4, the BLE module can resolve the remote device RPA included in the received packet and the remote device address included in the event that is notified to the application becomes identity address.

An example of the procedures 1-5 is shown below. If the local device resolves the RPA of the Remote device, the Identity Address and the IRK in the Resolving List must match Identity Address and IRK of the Remote device.

```
/** some code is omitted  **/
static ble_abs_scan_phy_parameter_t gs_scan_phy_parameter =
{
    .fast_scan_interval      = 0x0060, /* 60.0(ms) */
    .fast_scan_window        = 0x0030, /* 30.0(ms) */
    .slow_scan_interval      = 0x0800, /* 1,280.0(ms) */
    .slow_scan_window        = 0x0012, /* 11.25(ms) */
    .scan_type               = BLE_GAP_SCAN_PASSIVE
};
static ble_abs_scan_parameter_t gs_scan_parameter =
{
    .p_phy_parameter_1M        = &gs_scan_phy_parameter,
    .fast_scan_period          = 0x0000, /* 0(ms) */
    .slow_scan_period          = 0x0000,
    .device_scan_filter_policy = BLE_GAP_SCAN_ALLOW_ADV_ALL,
    .filter_duplicate          = BLE_GAP_SCAN_FILT_DUPLIC_ENABLE,
};
static ble_abs_connection_phy_parameter_t gs_connection_phy_parameter =
{
```

```
    .connection_interval      = 0x0028, /* 50.0(ms) */
    .supervision_timeout      = 0x0200, /* 5,120(ms) */
    .connection_slave_latency = 0x0000,
};
static ble_device_address_t gs_connection_device_address;
static ble_abs_connection_parameter_t gs_connection_parameter =
{
    .p_connection_phy_parameter_1M = &gs_connection_phy_parameter,
    .p_device_address              = &gs_connection_device_address,     /**< Set BD address of connecting
device. */
    .filter_parameter              = ((BLE_GAP_ADDR_RAND << 4) | BLE_GAP_INIT_FILT_USE_ADDR),
    .connection_timeout            = 0x05, /* 5(s) */
};
static ble_abs_pairing_parameter_t gs_abs_pairing_param =
{
    .io_capabilitie_local_device = BLE_GAP_IOCAP_NOINPUT_NOOUTPUT,
    .mitm_protection_policy      = BLE_GAP_SEC_MITM_BEST_EFFORT,
    .secure_connection_only      = BLE_GAP_SC_BEST_EFFORT,
    .local_key_distribute        = BLE_GAP_KEY_DIST_ENCKEY,
    .remote_key_distribute       = BLE_GAP_KEY_DIST_ENCKEY | BLE_GAP_KEY_DIST_IDKEY,
    .maximum_key_size            = 16,
};
/** some code is omitted  **/

void gap_cb(uint16_t event_type, ble_status_t event_result, st_ble_evt_data_t * p_data)
{
    /** some code is omitted  **/
    switch(event_type)
    {
        case BLE_GAP_EVENT_STACK_ON:
        {
        } break;
        case BLE_GAP_EVENT_LOC_VER_INFO:
        {
            uint8_t lc_irk[BLE_GAP_IRK_SIZE];
            memset(lc_irk, 0x00, BLE_GAP_IRK_SIZE);
            RM_BLE_ABS_SetLocalPrivacy(g_ble_abs0.p_ctrl, lc_irk,
                    (BLE_GAP_NET_PRIV_MODE | (BLE_GAP_ADDR_RAND << 4)));
        } break;
        case BLE_GAP_EVENT_RPA_EN_COMP:
        {
            RM_BLE_ABS_StartScanning(&g_ble_abs0_ctrl, &gs_scan_parameter);
        } break;
        case BLE_GAP_EVENT_ADV_REPT_IND:
        {
            st_ble_gap_adv_rept_evt_t *p_adv_rept_evt_param = (st_ble_gap_adv_rept_evt_t *)p_data->p_param;
            st_ble_gap_ext_adv_rept_t *p_adv_rept_param = (st_ble_gap_ext_adv_rept_t
*)p_adv_rept_evt_param->param.p_ext_adv_rpt;
            gs_connection_parameter.p_device_address->type = p_adv_rept_param->addr_type;
            memcpy(gs_connection_parameter.p_device_address->addr, p_adv_rept_param->p_addr,
BLE_BD_ADDR_LEN);
            R_BLE_GAP_StopScan();
        } break;
        case BLE_GAP_EVENT_SCAN_OFF:
        {
            gs_connection_parameter.p_device_address->type = gs_connection_parameter.p_device_address-
>type % 2;
            RM_BLE_ABS_CreateConnection(&g_ble_abs0_ctrl, &gs_connection_parameter);
            }
        } break;
        case BLE_GAP_EVENT_CONN_IND:
        {
            st_ble_gap_pairing_param_t  pair_param;
            /* update pairing parameters */
            pair_param.iocap = gs_abs_pairing_param.io_capabilitie_local_device;
            pair_param.mitm = gs_abs_pairing_param.mitm_protection_policy;
            pair_param.sec_conn_only = gs_abs_pairing_param.secure_connection_only;
            pair_param.loc_key_dist = gs_abs_pairing_param.local_key_distribute;
            pair_param.max_key_size = BLE_GAP_LTK_SIZE;
            pair_param.bonding = BLE_GAP_BONDING;
            pair_param.min_key_size = gs_abs_pairing_param.maximum_key_size;
            pair_param.key_notf = BLE_GAP_SC_KEY_PRESS_NTF_NOT_SPRT;
            pair_param.rem_key_dist = BLE_GAP_KEY_DIST_ENCKEY | BLE_GAP_KEY_DIST_IDKEY;
            R_BLE_GAP_SetPairingParams(&pair_param);
```

```
        } break;
        case BLE_GAP_EVENT_PEER_KEY_INFO:
        {
            st_ble_gap_peer_key_info_evt_t *p_peer_key_info_evt_param =
                (st_ble_gap_peer_key_info_evt_t *)p_data->p_param;
            st_ble_gap_key_dist_t * key_info;
            st_ble_gap_rslv_list_key_set_t  key_set;
            key_info = p_peer_key_info_evt_param->key_ex_param.p_keys_info;
            R_BLE_CLI_Printf("keys : 0x%02x\n", p_peer_key_info_evt_param->key_ex_param.keys);
            if(0 != (BLE_GAP_KEY_DIST_IDKEY & p_peer_key_info_evt_param->key_ex_param.keys))
            {
                /* Add remote address & irk to the resolving list. */
                st_ble_dev_addr_t r_id_addr;
                memcpy(key_set.remote_irk, key_info->id_info, BLE_GAP_IRK_SIZE);
                key_set.local_irk_type = BLE_GAP_RL_LOC_KEY_REGISTERED;
                memcpy(r_id_addr.addr, &key_info->id_addr_info[1], BLE_BD_ADDR_LEN);
                r_id_addr.type = key_info->id_addr_info[0];
                R_BLE_GAP_ConfRslvList(BLE_GAP_LIST_ADD_DEV, &r_id_addr, &key_set, 1);
            }
        } break;
        /** some code is omitted  **/
    }
}
static void disc_comp_cb(uint16_t conn_hdl)
{
    R_BLE_CLI_Printf("disc finished\n");
    RM_BLE_ABS_StartAuthentication(&g_ble_abs0_ctrl, conn_hdl);
    return;
}
/** some code is omitted  **/
```

Code 43. Sample for resolving RPA of remote device with abstraction API

After resolving the RPA, you will need to use the Identity Address to connect and to register whitelist.

If you restart the BLE protocol stack, you will need to reset the key stored on the device to the Resolving List by R_BLE_GAP_ConfRslvList.

Please refer to "8.2.3 Reset the stored keys".

8.4.4 Resolve remote device RPA with GAP API

RPA of Remote device is resolved according to the following procedures. If the local device resolves RPA by using GAP API, perform the following steps.

1. Start RPA feature

Call *R_BLE_GAP_EnableRpa* API to enable the RPA generation and resolution. The completion is notified by *BLE_GAP_EVENT_RPA_EN_COMP* event.

2. Pairing

Receive the remote device IRK and identity address by pairing. For more details about pairing, see "8.1 Pairing".

3. Register remote device key (IRK) and BD address

Call *R_BLE_GAP_ConfRslvList* API to register the remote device IRK and identity address in the Resolving List. The local device IRK is also registered at that time. If the local device does not use RPA, set the local device IRK to all 0x00 by setting the local_irk_type in st_ble_gap_rslv_list_key_set_t type array of third parameter to BLE_GAP_RL_LOC_KEY_ALL_ZERO.
BLE_GAP_EVENT_RSLV_LIST_CONF_COMP event notifies the user application that the registration is complete. If users want to store the remote device IRK and identity address to data flash, it is necessary implement the procedure by themselves.

4. Set Privacy Mode

If Network Privacy Mode which is the default is used, the procedure does not need to be done.
Call the *R_BLE_GAP_SetPrivMode* API to set the privacy mode.
BLE_GAP_EVENT_PRIV_MODE_SET_COMP event notifies user application of the completion.

5. Resolve RPA

After the procedures 1-4, the BLE module can resolve the remote device RPA included in the received packet and the remote device address included in the event that is notified to the application becomes identity address.

An example of the procedures 1-5 is shown below. If the Local device resolves the RPA of the Remote device, the Identity Address and the IRK in the Resolving List must match Identity Address and IRK of the Remote device.

```
/** some code is omitted  **/
static ble_abs_scan_phy_parameter_t gs_scan_phy_parameter =
{
    .fast_scan_interval      = 0x0060, /* 60.0(ms) */
    .fast_scan_window        = 0x0030, /* 30.0(ms) */
    .slow_scan_interval      = 0x0800, /* 1,280.0(ms) */
    .slow_scan_window        = 0x0012, /* 11.25(ms) */
    .scan_type               = BLE_GAP_SCAN_PASSIVE
};
static ble_abs_scan_parameter_t gs_scan_parameter =
{
    .p_phy_parameter_1M        = &gs_scan_phy_parameter,
    .fast_scan_period          = 0x0000, /* 0(ms) */
    .slow_scan_period          = 0x0000,
    .device_scan_filter_policy = BLE_GAP_SCAN_ALLOW_ADV_ALL,
    .filter_duplicate          = BLE_GAP_SCAN_FILT_DUPLIC_ENABLE,
};
static ble_abs_connection_phy_parameter_t gs_connection_phy_parameter =
{
    .connection_interval      = 0x0028, /* 50.0(ms) */
    .supervision_timeout      = 0x0200, /* 5,120(ms) */
```

```
    .connection_slave_latency = 0x0000,
};
static ble_device_address_t gs_connection_device_address;
static ble_abs_connection_parameter_t gs_connection_parameter =
{
    .p_connection_phy_parameter_1M = &gs_connection_phy_parameter,
    .p_device_address              = &gs_connection_device_address,     /**< Set BD address of connecting
device. */
    .filter_parameter              = ((BLE_GAP_ADDR_RAND << 4) | BLE_GAP_INIT_FILT_USE_ADDR),
    .connection_timeout            = 0x05, /* 5(s) */
};
static ble_abs_pairing_parameter_t gs_abs_pairing_param =
{
    .io_capabilitie_local_device = BLE_GAP_IOCAP_NOINPUT_NOOUTPUT,
    .mitm_protection_policy      = BLE_GAP_SEC_MITM_BEST_EFFORT,
    .secure_connection_only      = BLE_GAP_SC_BEST_EFFORT,
    .local_key_distribute        = BLE_GAP_KEY_DIST_ENCKEY,
    .remote_key_distribute       = BLE_GAP_KEY_DIST_ENCKEY | BLE_GAP_KEY_DIST_IDKEY,
    .maximum_key_size            = 16,
};
/** some code is omitted  **/

void gap_cb(uint16_t event_type, ble_status_t event_result, st_ble_evt_data_t * p_data)
{
    /** some code is omitted  **/
    switch(event_type)
    {
        case BLE_GAP_EVENT_STACK_ON:
        {
        } break;
        case BLE_GAP_EVENT_LOC_VER_INFO:
        {
            R_BLE_GAP_EnableRpa(BLE_GAP_RPA_ENABLED);
        } break;
        case BLE_GAP_EVENT_RPA_EN_COMP:
        {
            RM_BLE_ABS_StartScanning(&g_ble_abs0_ctrl, &gs_scan_parameter);
        } break;
        case BLE_GAP_EVENT_ADV_REPT_IND:
        {
            st_ble_gap_adv_rept_evt_t *p_adv_rept_evt_param = (st_ble_gap_adv_rept_evt_t *)p_data->p_param;
            st_ble_gap_ext_adv_rept_t *p_adv_rept_param = (st_ble_gap_ext_adv_rept_t
*)p_adv_rept_evt_param->param.p_ext_adv_rpt;
            gs_connection_parameter.p_device_address->type = p_adv_rept_param->addr_type;
            memcpy(gs_connection_parameter.p_device_address->addr, p_adv_rept_param->p_addr,
BLE_BD_ADDR_LEN);
            R_BLE_GAP_StopScan();
        } break;
        case BLE_GAP_EVENT_SCAN_OFF:
        {
            gs_connection_parameter.p_device_address->type = gs_connection_parameter.p_device_address-
>type % 2;
            RM_BLE_ABS_CreateConnection(&g_ble_abs0_ctrl, &gs_connection_parameter);
            }
        } break;
        case BLE_GAP_EVENT_CONN_IND:
        {
            st_ble_gap_pairing_param_t  pair_param;
            /* update pairing parameters */
            pair_param.iocap = gs_abs_pairing_param.io_capabilitie_local_device;
            pair_param.mitm = gs_abs_pairing_param.mitm_protection_policy;
            pair_param.sec_conn_only = gs_abs_pairing_param.secure_connection_only;
            pair_param.loc_key_dist = gs_abs_pairing_param.local_key_distribute;
            pair_param.max_key_size = BLE_GAP_LTK_SIZE;
            pair_param.bonding = BLE_GAP_BONDING;
            pair_param.min_key_size = gs_abs_pairing_param.maximum_key_size;
            pair_param.key_notf = BLE_GAP_SC_KEY_PRESS_NTF_NOT_SPRT;
            pair_param.rem_key_dist = BLE_GAP_KEY_DIST_ENCKEY | BLE_GAP_KEY_DIST_IDKEY;
            R_BLE_GAP_SetPairingParams(&pair_param);
        } break;
        case BLE_GAP_EVENT_PEER_KEY_INFO:
        {
            st_ble_gap_peer_key_info_evt_t *p_peer_key_info_evt_param =
                (st_ble_gap_peer_key_info_evt_t *)p_data->p_param;
```

```
            st_ble_gap_key_dist_t * key_info;
            st_ble_gap_rslv_list_key_set_t  key_set;
            key_info = p_peer_key_info_evt_param->key_ex_param.p_keys_info;
            R_BLE_CLI_Printf("keys : 0x%02x\n", p_peer_key_info_evt_param->key_ex_param.keys);
            if(0 != (BLE_GAP_KEY_DIST_IDKEY & p_peer_key_info_evt_param->key_ex_param.keys))
            {
                /* Add remote address & irk to the resolving list. */
                st_ble_dev_addr_t r_id_addr;
                memcpy(key_set.remote_irk, key_info->id_info, BLE_GAP_IRK_SIZE);
                key_set.local_irk_type = BLE_GAP_RL_LOC_KEY_REGISTERED;
                memcpy(r_id_addr.addr, &key_info->id_addr_info[1], BLE_BD_ADDR_LEN);
                r_id_addr.type = key_info->id_addr_info[0];
                R_BLE_GAP_ConfRslvList(BLE_GAP_LIST_ADD_DEV, &r_id_addr, &key_set, 1);
            }
        } break;
        /** some code is omitted  **/
    }
}
static void disc_comp_cb(uint16_t conn_hdl)
{
    R_BLE_CLI_Printf("disc finished\n");
    RM_BLE_ABS_StartAuthentication(&g_ble_abs0_ctrl, conn_hdl);
    return;
}
/** some code is omitted  **/
```

Code 44. Sample for resolving RPA of remote device with GAP API

After resolving the RPA, you will need to use the Identity Address to connect and to register the Whitelist.

If you restart the BLE protocol stack, you will need to reset the key stored on the device to the Resolving List by R_BLE_GAP_ConfRslvList.

Please refer to "8.2.3 Reset the stored keys".

9. Profile and service

Profiles in Bluetooth LE communication are mechanisms for ensuring interoperability between devices by defining the services and communication protocols that application share. Profile-based data communication is achieved by accessing a common data structure called GATT database. As shown in Figure 48, the GATT database consists of one or more services. Services consist of one or more characteristics that enable profile functionality, and characteristics define data structures and access procedures. The procedure for accessing characteristics is called GATT procedure, and this procedure defines how to send and receive data. The user profile can be designed using QE for BLE. For information on how to design profiles using QE for BLE, refer to "*Bluetooth Low Energy Profile Developer's Guide (R01AN6459)*". This chapter describes the profiles and services provided by Renesas and explains APIs for each GATT procedure including examples of how to use them.

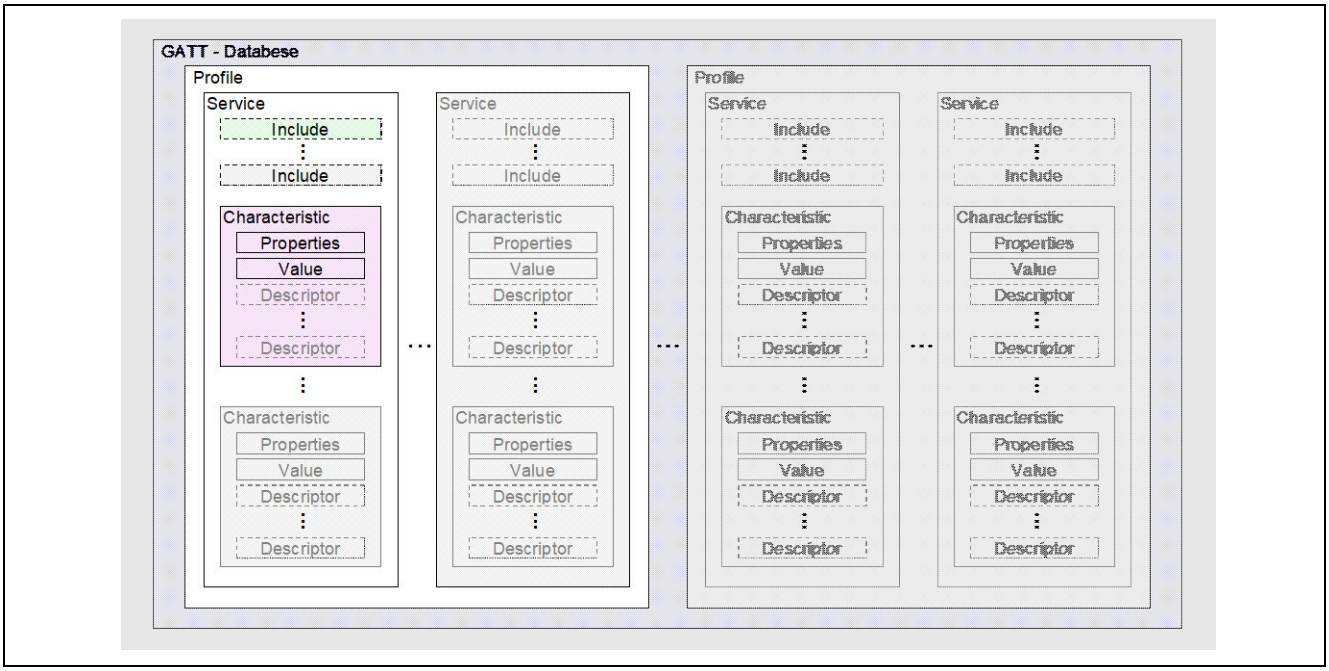


Figure 48. Data structure of GATT database

9.1 Standard profile and Standard Service

Standard profiles and services can be used in user applications using QE for BLE. RA4W1 supports the standard profiles and services listed in Table 47. Table 48 lists the characteristics that included in each standard service.

Table 47. Profile supported by RA4W1

Usage	Profile	Service			
Healthcare	Blood Pressure Profile	BLS	DIS		
	Health Thermometer Profile	HTS	DIS		
	Heart Rate Profile	HRS	DIS		
	Glucose Profile	GLS	DIS		
	Pulse Oximeter Profile	PLXS	DIS	BAS	CTS
		BMS			
	Continuous Glucose Monitoring Profile	CGMS	DIS	BMS	
	Reconnection Configuration Profile	RCS	BMS		
	Insulin Delivery Profile	IDS	DIS	BAS	CTS
		BMS	IAS		
Sports and Fitness	Cycling Power Profile	CPS	DIS	BAS	
	Cycling Speed and Cadence Profile	CSCS	DIS		
	Running Speed and Cadence Profile	RSCS	DIS		
	Location and Navigation Profile	LNS	DIS	BAS	
	Weight Scale Profile	WSS	BCS	DIS	BAS
		CTS	UDS		
	Fitness Machine Profile	FTMS	DIS	UDS	
	Environmental Sensing Profile	ESS	DIS	BAS	
Radio tag	Find Me Profile	IAS			
	Proximity Profile	IAS	LLS	TPS	
Smartphone	Alert Notification Profile	ANS			
	Phone Alert Status Profile	PASS			
	Time Profile	CTS	NDCS	RTUS	
HID (Human Interface Device)	HID over GATT Profile	HIDS	DIS	BAS	
	Scan Parameters Profile	SCPS			
Industrial equipment	Automation IO Profile	AIOS			

Table 48. Structure of standard service

Service	Characteristic	GATT Procedure
Alert Notification Service **ANS**	Supported New Alert Category	Read
	New Alert	Notify
	Supported Unread Alert Category	Read
	Unread Alert Status	Notify
Automation IO Service **AIOS**	Digital 0	Read, Write, WriteWithoutResponse, Notify
	Digital 1	Read, Write, WriteWithoutResponse, Notify
	Analog 0	Read, Write, WriteWithoutResponse, Notify
	Analog 1	Read, Write, WriteWithoutResponse, Notify
	Aggregate	Read, Notify
Battery Service **BAS**	Battery Level	Read, Notify
Blood Pressure Service **BLS**	Blood Pressure Measurement	Indicate
	Intermediate Cuff Pressure	Notify
	Blood Pressure Feature	Read, Indicate
Body Composition Service **BCS**	Body Composition Feature	Read
	Body Composition Measurement	Indicate
Bond Management Service **BMS**	Bond Management Control Point	Write, ReliableWrite
	Bond Management Feature	Read, Indicate
Continuous Glucose Monitoring Service **CGMS**	CGM Measurement	Notify
	CGM Feature	Read, Indicate
	CGM Status	Read
	CGM Session Start Time	Read, Write
	CGM Session Run Time	Read
	Record Access Control Point	Write, Indicate
	CGM Specific Ops Control Point	Write, Indicate
Current Time Service **CTS**	Current Time	Read, Write, Notify
	Local Time Information	Read, Write
	Reference Time Information	Read
Cycling Power Service **CPS**	Cycling Power Measurement	Notify, Broadcast
	Cycling Power Feature	Read
	Sensor Location	Read
	Cycling Power Vector	Notify
	Cycling Power Control Point	Write, Indicate
Cycling Speed and Cadence Service **CSCS**	CSC Measurement	Notify
	CSC Feature	Read
	Sensor Location	Read
	SC Control Point	Write, Indicate

Service	Characteristic	GATT Procedure
Device Information Service **DIS**	Manufacturer Name String	Read
	Model Number String	Read
	Serial Number String	Read
	Hardware Revision String	Read
	Firmware Revision String	Read
	Software Revision String	Read
	System ID	Read
	IEEE 11073-20601 Regulatory Certification Data List	Read
	PnP ID	Read
Environmental Sensing Service **ESS**	Descriptor Value Changed	Indicate
	Temperature 0	Read, Notify
	Temperature 1	Read, Notify
	Elevation 0	Read, Notify
	Elevation 1	Read, Notify
Fitness Machine Service **FTMS**	Fitness Machine Feature	Read
	Treadmill Data	Notify
	Cross Trainer Data	Notify
	Step Climber Data	Notify
	Stair Climber Data	Notify
	Rower Data	Notify
	Indoor Bike Data	Notify
	Training Status	Read, Notify
	Supported Speed Range	Read
	Supported Inclination Range	Read
	Supported Resistance Level Range	Read
	Supported Power Range	Read
	Supported Heart Rate Range	Read
	Fitness Machine Control Point	Write, Indicate
	Fitness Machine Status	Notify
GAP Service **GAP**	Device Name	Read, Write
	Appearance	Read
	Peripheral Preferred Connection Parameters	Read
	Central Address Resolution	Read
	Resolvable Private Address Only	Read
GATT Service **GATT**	Service Changed	Indicate
Glucose Service **GLS**	Glucose Measurement	Notify
	Glucose Measurement Context	Notify
	Glucose Feature	Read, Indicate
	Record Access Control Point	Write, Indicate

Service	Characteristic	GATT Procedure
Health Thermometer Service **HTS**	Temperature Measurement	Indicate
	Temperature Type	Read
	Intermediate Temperature	Notify
	Measurement Interval	Read, Write, Indicate
Heart Rate Service **HRS**	Heart Rate Measurement	Notify
	Body Sensor Location	Read
	Heart Rate Control Point	Write
Human Interface Device Service **HIDS**	Protocol Mode	Read, WriteWithoutResponse
	Report	Read, Write, WriteWithoutResponse, Notify
	Report Map	Read
	Boot Keyboard Input Report	Read, Write, Notify
	Boot Keyboard Output Report	Read, Write, WriteWithoutResponse
	Boot Mouse Input Report	Read, Write, Notify
	HID Information	Read
	HID Control Point	WriteWithoutResponse
Immediate Alert Service **IAS**	Alert Level	WriteWithoutResponse
Insulin Delivery Service **IDS**	IDD Status Changed	Read, Indicate
	IDD Status	Read, Indicate
	IDD Annunciation Status	Read, Indicate
	IDD Features	Read, Indicate
	IDD Status Reader Control Point	Write, Indicate
	IDD Command Control Point	Write, Indicate
	IDD Command Data	InformativeText, Notify
	IDD Record Access Control Point	Write, Indicate
	IDD History Data	InformativeText, Notify
Link Loss Service **LLS**	Alert Level	Read, Write
Location and Navigation Service **LNS**	LN Feature	Read
	Location and Speed	Notify
	Position Quality	Read
	LN Control Point	Write, Indicate
	Navigation	Notify
Next DST Change Service **NDCS**	Time with DST	Read
Object Transfer Service **OTS**	OTS Feature	Read
	Object Name	Read, Write
	Object Type	Read
	Object Size	Read
	Object First-Created	Read, Write

Service	Characteristic	GATT Procedure
	Object Last-Modified	Read, Write
	Object ID	Read
	Object Properties	Read, Write
	Object Action Control Point	Write, Indicate
	Object List Control Point	Write, Indicate
	Object List Filter 0	Read, Write
	Object List Filter 1	Read, Write
	Object List Filter 2	Read, Write
	Object Changed	Indicate
Phone Alert Status Service **PASS**	Alert Status	Read, Notify
	Ringer Setting	Read, Notify
	Ringer Control point	WriteWithoutResponse
Pulse Oximeter Service **PLXS**	PLX Spot-Check Measurement	Indicate
	PLX Continuous Measurement	Notify
	PLX Features	Read, Indicate
	Record Access Control Point	Write, Indicate
Reconnection Configuration Service **RCS**	RC Feature	Read, Indicate
	RC Settings	Read, Notify
	Reconnection Configuration Control Point	Write, Indicate
Reference Time Update Service **RTUS**	Time Update Control Point	WriteWithoutResponse
	Time Update State	Read
Running Speed and Cadence Service **RSCS**	RSC Measurement	Notify
	RSC Feature	Read
	Sensor Location	Read
	SC Control Point	Write, Indicate
Scan Parameters Service **SCPS**	Scan Interval Window	WriteWithoutResponse
	Scan Refresh	Notify
Tx Power Service **TPS**	Tx Power Level	Read
User Data Service **UDS**	First Name	Read, Write
	Last Name	Read, Write
	Email Address	Read, Write
	Age	Read, Write
	Date of Birth	Read, Write
	Gender	Read, Write
	Weight	Read, Write
	Height	Read, Write
	VO2 Max	Read, Write
	Heart Rate Max	Read, Write

Service	Characteristic	GATT Procedure
	Resting Heart Rate	Read, Write
	Maximum Recommended Heart Rate	Read, Write
	Aerobic Threshold	Read, Write
	Anaerobic Threshold	Read, Write
	Sport Type for Aerobic and Anaerobic Thresholds	Read, Write
	Date of Threshold Assessment	Read, Write
	Waist Circumference	Read, Write
	Hip Circumference	Read, Write
	Fat Burn Heart Rate Lower Limit	Read, Write
	Fat Burn Heart Rate Upper Limit	Read, Write
	Aerobic Heart Rate Lower Limit	Read, Write
	Aerobic Heart Rate Upper Limit	Read, Write
	Anaerobic Heart Rate Lower Limit	Read, Write
	Anaerobic Heart Rate Upper Limit	Read, Write
	Five Zone Heart Rate Limits	Read, Write
	Three Zone Heart Rate Limits	Read, Write
	Two Zone Heart Rate Limit	Read, Write
	Database Change Increment	Read, Write, Notify
	User Index	Read
	User Control Point	Write, Indicate
	Language	Read, Write
	Registered User	Read, Write
	Preferred Units	Read, Write
	High Resolution Height	Read, Write
	Middle Name	Read, Write
	Stride Length	Read, Write
	Handedness	Read, Write
	Device Wearing Position	Read, Write
	Four Zone Heart Rate Limits	Read, Write
	High Intensity Exercise Threshold	Read, Write
	Activity Goal	Read, Write
	Sedentary Interval Notification	Read, Write
	Caloric Intake	Read, Write
Weight Scale Service **WSS**	Weight Scale Feature	Read
	Weight Measurement	Indicate

9.2 APIs of GATT Procedure

QE for BLE generates APIs according to the GATT procedure set to the characteristic. This section describes how to implement each GATT procedure that can be configured from QE for BLE.
In the following description, we will use the function name and event name which will be generated from QE for BLE. Abbreviation of the service is set to “XXX” and abbreviation of characteristic is set to “YYY” in QE for BLE.

9.2.1 Read operation

Read operation is a procedure of the GATT client to check the data in the GATT database of the GATT server, as shown in Figure 49. Using this procedure when checking the configuration and status of the GATT server.

GATT server:
When GATT server receives “Read Request”, BLE Protocol Stack transmits “Read Response” with the value set in the GATT database. The event *BLE_XXX_EVENT_YYY_READ_REQ* occurs after receiving “Read Request” but before determining the data to be send in “Read Response”. If user want to change the data to be transmitted, use *R_BLE_XXX_SetYYY* API to change the value set in the GATT database. User can also send errors by using *R_BLE_GATTS_SetErrRsp* API.

GATT client:
“Read Request” can be transmitted by using *R_BLE_XXX_ReadYYY* API. BLE Protocol Stack notify the application of the event *BLE_XXX_EVENT_YYY_READ_RSP* indicating that “Read Response” has been received. The data received in this event is included in the structure which is defined in the *Fields* window of QE for BLE. The event *BLE_XXX_EVENT _YYY_READ_RSP* is received when read operation is completed. User can start another operation after received event.

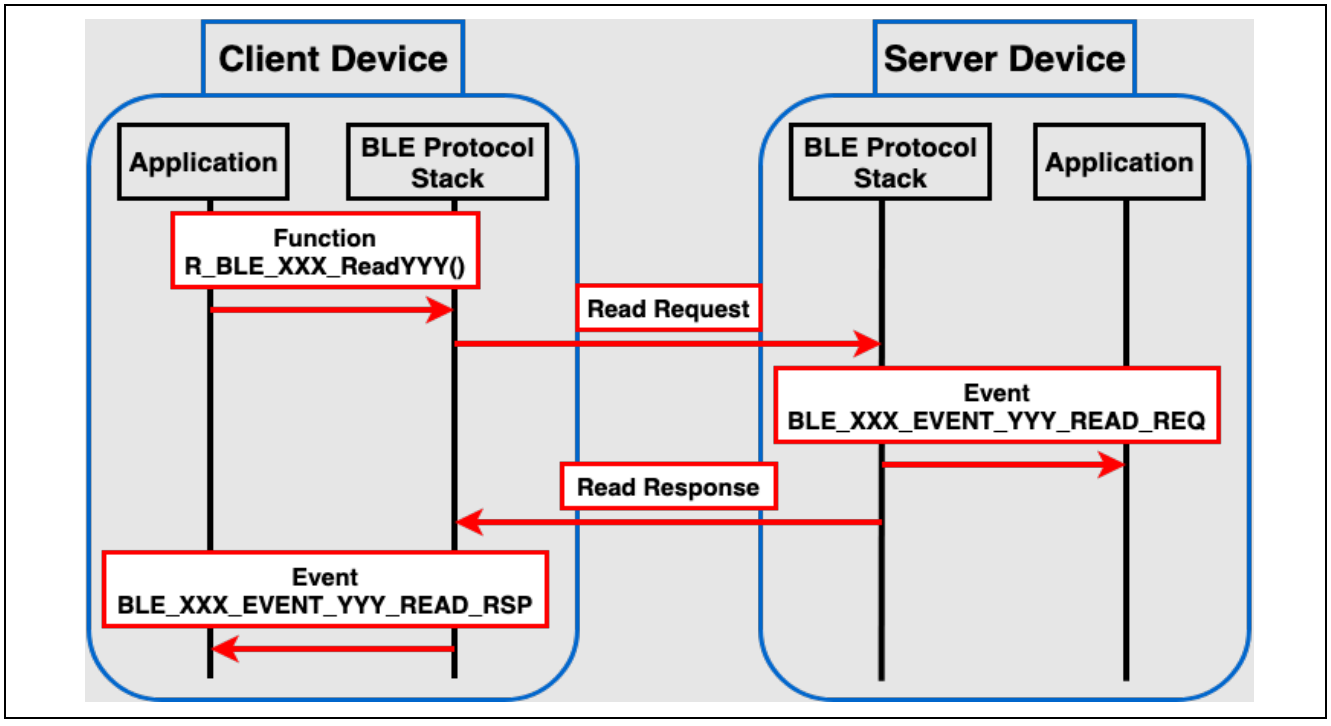


Figure 49. Flow of Read operation

9.2.2 Write operation

Write operation is procedure to change the GATT database of the GATT server by sending data from the GATT client, as shown in Figure 50. GATT client can check whether the submitted data is reflected in the GATT database in response from the GATT server. Using this procedure when user want to change the settings of the GATT server.

GATT server:
BLE Protocol Stack notifies the application of the event *BLE_XXX_EVENT_YYY_WRITE_REQ* and *BLE_XXX_EVENT_WRITE_COMP* indicating that “Write Request” has been received. The data received in this event is included in the structure which is defined in the *Fields* window of QE for BLE. Event *BLE_XXX_EVENT_WRITE_REQ* is an event to check the data received by “Write Request” before being written to the GATT database. If user receives invalid data, use *R_BLE_GATTS_SendErrRsp* API to send an error and the data would not be reflected in the GATT database. When using *R_BLE_GATTS_SendErrRsp* API, user can define unique error code. From 0x3080 to 0x309F can be used as unique error code. If user does not send an error, BLE Protocol Stack will send “Write Response”, so user does not need to add any process to respond in application. Event *BLE_XXX_EVENT_YYY_WRITE_COMP* is an event after the data received by “Write Request” is reflected in the GATT database and “Write Response” is sent. Process that refers to GATT database directly or corresponds to the data received by “Write Request” should be added after this event.

GATT client:
User can send “Write Request” by using *R_BLE_XXX_WriteYYY* API. Result of the Write operation can be checked by the event *BLE_XXX_EVENT_YYY_WRITE_RSP*. Write operation is completed when the event *BLE_XXX_EVENT _YYY_WRITE_RSP* is received. Users can start another operation after this event.

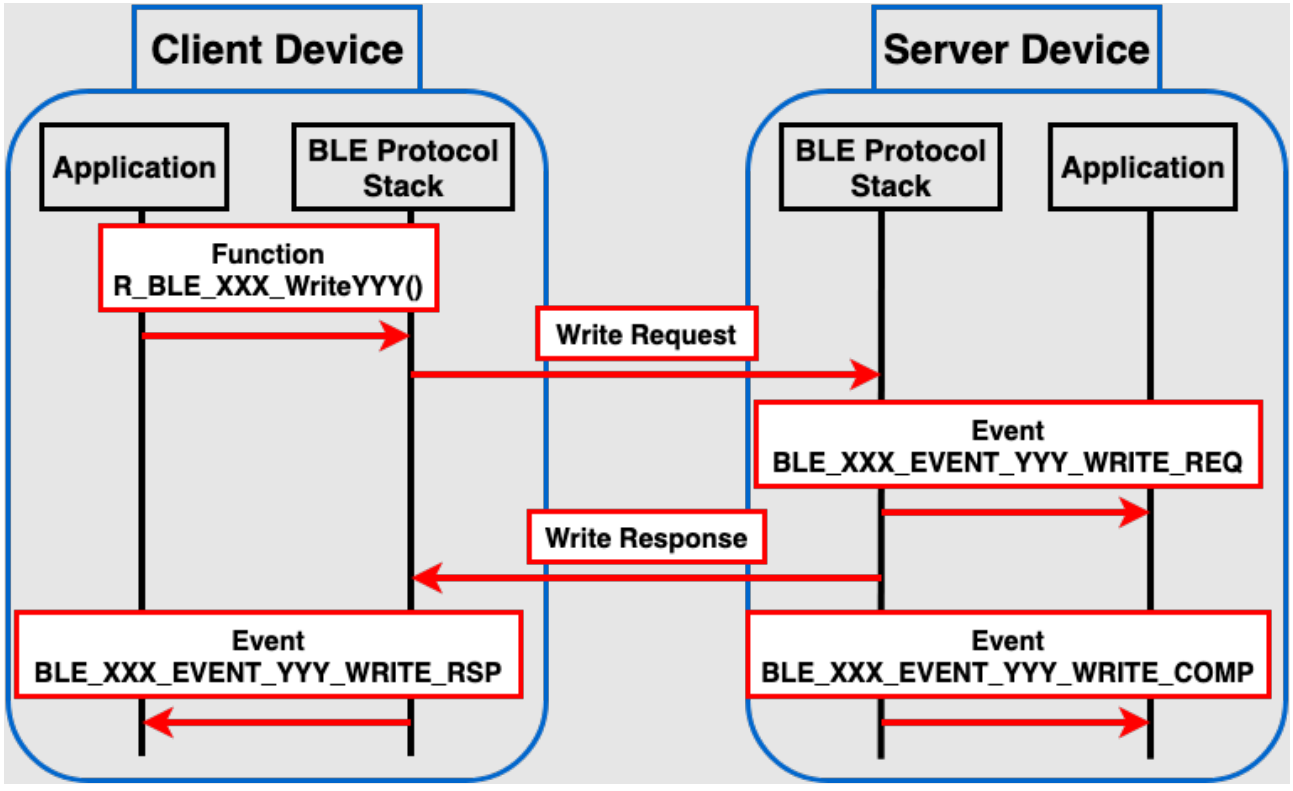


Figure 50.Flow of Write operation

9.2.3 WriteWithoutResponse operation

WriteWithoutResponse operation is a procedure to change the GATT database of the GATT server by sending data from the GATT client, as shown in Figure 51. Since there is no response from the GATT server, it is possible to continuously transmit data from the GATT client to GATT server and reduce the power consumption of the GATT server device. On the other hand, it is not possible to verify that the data sent by GATT client has been reflected in the GATT database. Using this procedure is recommended when the user wants to reduce power consumption on the user's device, or when the user wants to send data continuously from GATT client to GATT server.

GATT server:
BLE Protocol Stack notifies application of the event *BLE_XXX_EVENT_YYY_WRITE_CMD* indicating that "Write Command" has been received. The data received in this event is included in the structure which is defined in the *Fields* window of QE for BLE event. When the event *BLE_XXX_EVENT_YYY_WRITE_CMD* is received, changes to the GATT database are not reflected. Therefore, process that refers to the GATT database directly should not be added at the event.

GATT client:
Users can send "Write Command" by using the function *R_BLE_XXX_WriteWithoutResponseYYY* API. WriteWithoutResponse operation is completed when call *R_BLE_XXX_WriteWithoutResponseYYY* API. Users can start another operation after calling the API.

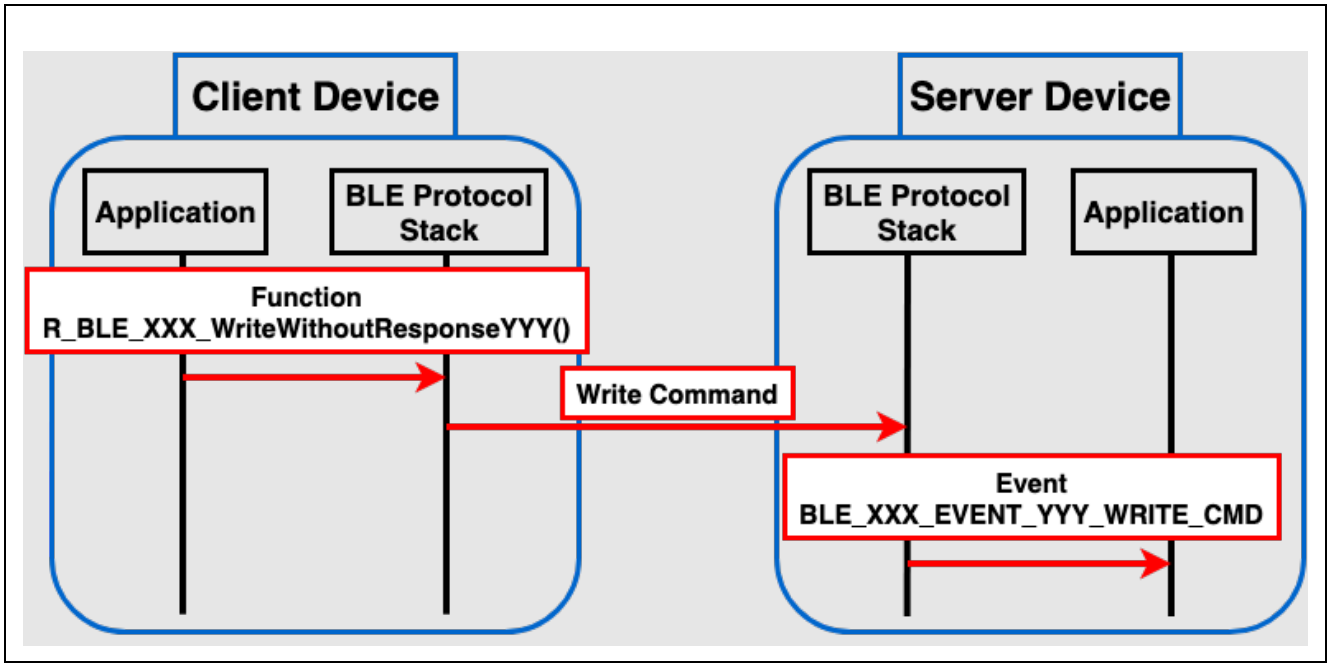


Figure 51. Flow of WriteWithoutResponse operation

9.2.4 Notification operation

Notification operation is a procedure to send data from GATT server to GATT client, as shown in Figure 52. For Notification operation, the CCCD must have been added as a descriptor. The GATT client must also set the CCCD to the appropriate value before the operation. Since there is no response from the GATT client, it is possible to send data continuously from the GATT server. On the other hand, it is not possible to confirm whether GATT client received the notification data. Using this procedure is recommended when user wants to send data continuously from the GATT server.

GATT server:
Before the operation, verify that the CCCD has been changed to an appropriate value. Make sure that *BLE_GATTS_CLI_CNFG_NOTIFICATION (0x0001)* is written in the event *BLE_XXX_EVENT_YYY_CLI_CNFG_WRITE_COMP* that occurs after the CCCD writing is completed. User can send “Handle Value Notification” by using *R_BLE_XXX_NotifyYYY* API. If the value of CCCD has not changed, the *R_BLE_XXX_NotifyYYY* API will return *BLE_ERR_INVALID_OPERATION* and does not send ”Handle Value Notification” from GATT server. Notification operation is completed when calling *R_BLE_XXX_NotifyYYY* API. Users can start another operation after calling the API.

GATT client:
Before the operation, it is necessary to change the value of CCCD to the appropriate value. Write *BLE_GATTS_CLI_CNFG_NOTIFICATION (0x0001)* to CCCD of characteristic which performs Notification operation. BLE Protocol Stack notifies the application of the event *BLE_XXX_EVENT_YYY_HDL_VAL_NTF* indicating that “Handle Value Notification” has been received. The data received in this event is included in the structure which is defined in the *Fields* window of QE for BLE.

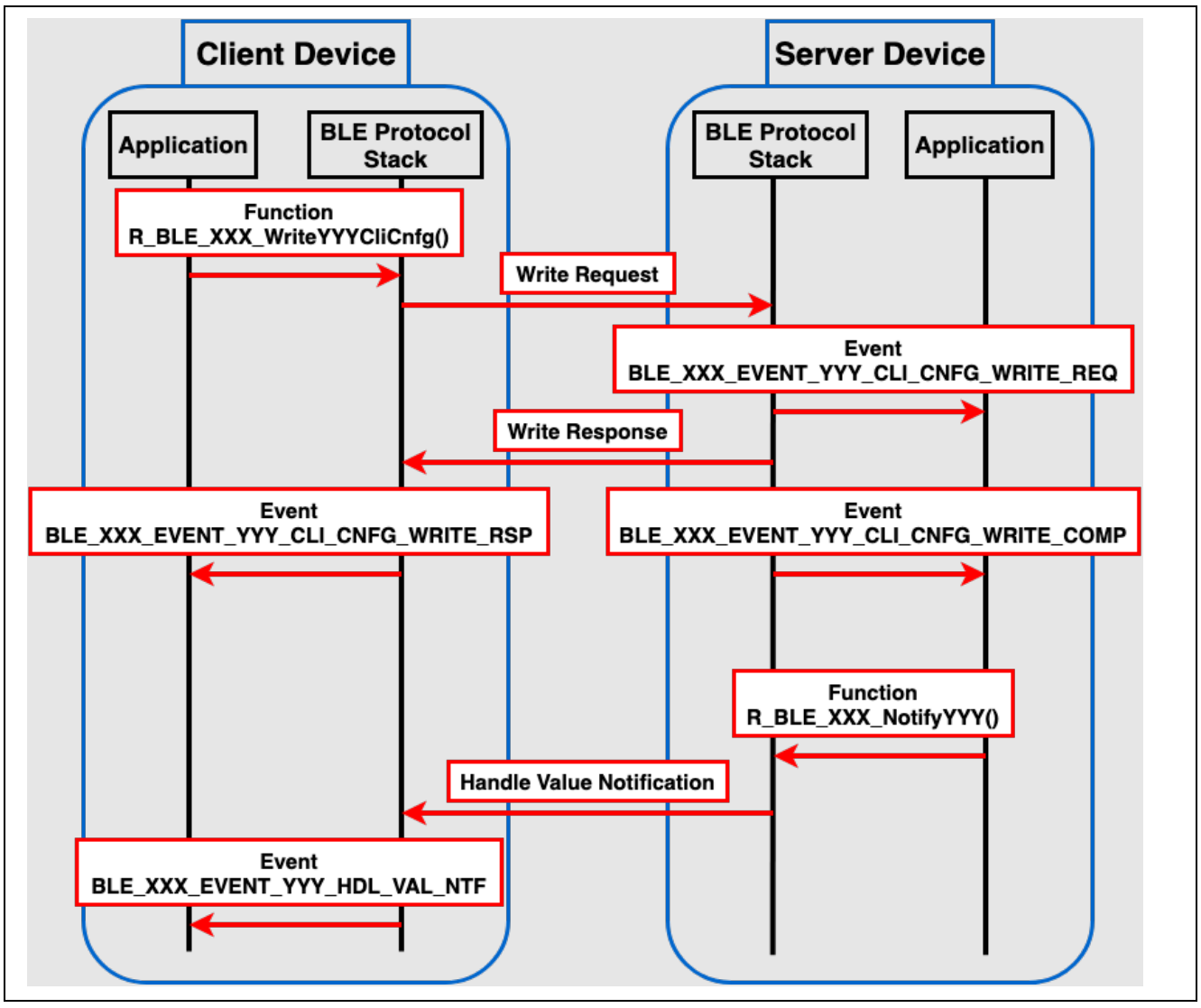


Figure 52. Flow of Notification operation

9.2.5 Indication operation

The Indication operation is a procedure to send data from GATT server to GATT client, as shown in Figure 53. For the Indication operation, the CCCD must have been added as a descriptor. The GATT client must also set the CCCD to the appropriate value before the operation. GATT server can verify that GATT client has received data sent from GATT server in a response from GATT client.

GATT server:
Before the operation, verify that the CCCD has been changed to appropriate value. Make sure that *BLE_GATTS_CLI_CNFG_INDICTION (0x0002)* is written in the event *BLE_XXX_EVENT_YYY_CLI_CNFG_WRITE_COMP* that occurs after the CCCD writing is completed. User can send “Handle Value Indication” by using *R_BLE_XXX_IndicateYYY* API. If the value of the CCCD has not changed, the function *R_BLE_XXX_IndicateYYY* API will return *BLE_ERR_INVALID_OPERATION* and does not send ”Handle Value Indication” from GATT server. Indication operation is completed when the event *BLE_XXX_EVENT_YYY_HDL_VAL_CNF* is received. User can start another operation after this event.

GATT client:
Before the operation, it is necessary to change the value of CCCD to the appropriate value. Write *BLE_GATTS_CLI_CNFG_INDICATION (0x0002)* to CCCD of characteristic which performs Indication operation. BLE Protocol Stack notifies the application of the event *BLE_XXX_EVENT_YYY_HDL_VAL_IND* indicating that “Handle Value Indication” has been received. The data received in this event is included in the structure which defined in the *Fields* window of QE for BLE. After the event *BLE_XXX_EVENT_YYY_HDL_VAL_IND*, BLE Protocol Stack automatically sends “Handle Value Confirmation”. Therefore, user does not need to add any process to send confirmation.

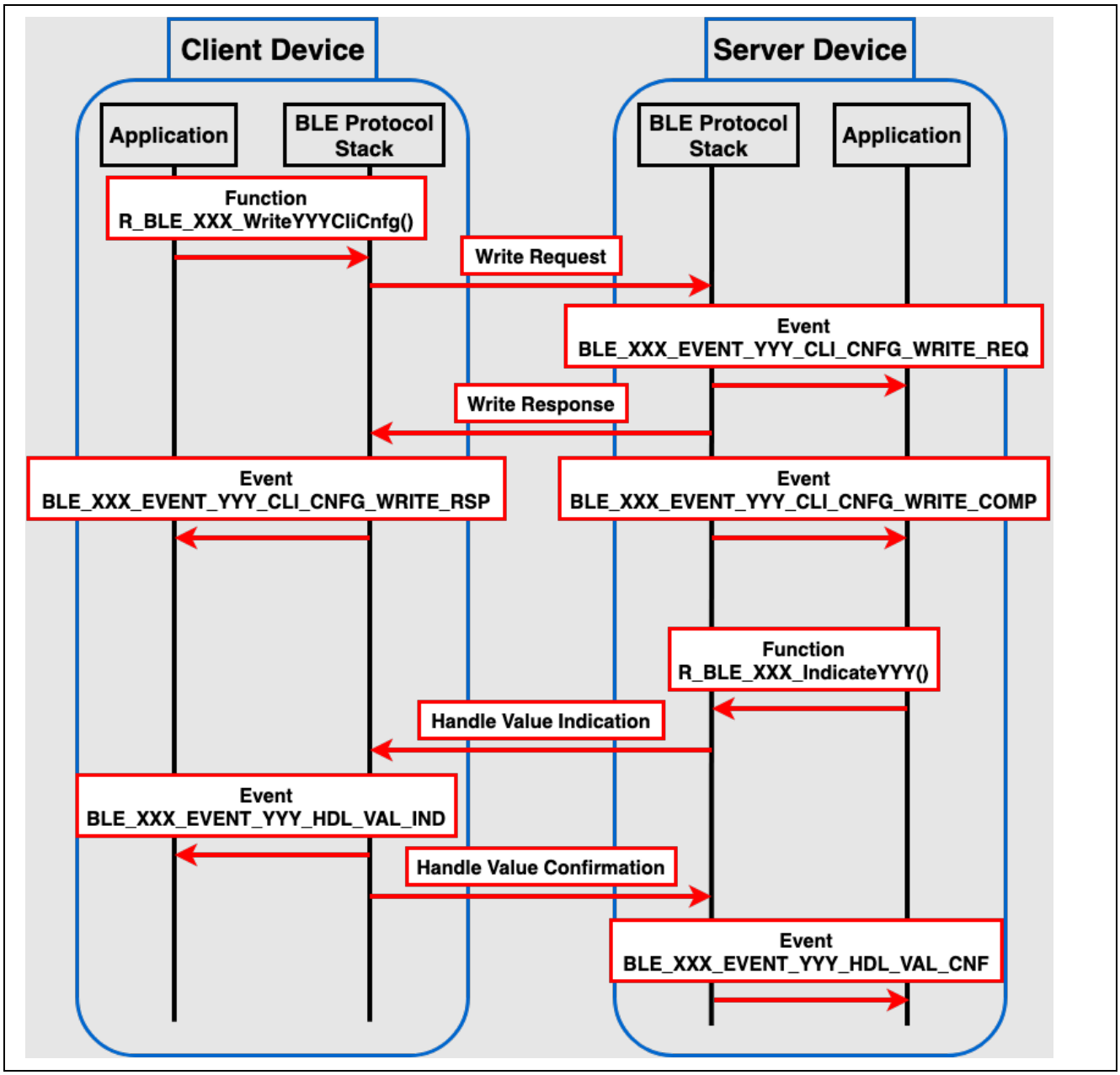


Figure 53. Flow of Indication operation

9.2.6 ReliableWrite operation

The ReliableWrite operation is a procedure to send data from GATT client to GATT server, ensure that the correct values are written, and then reflected in the GATT database, as shown in Figure 54. There are two steps for ReliableWrite operation.

1. GATT client sends data using “Prepare Write Request” and GATT server holds it in queue. GATT client can verify that the correct data is being written in “Prepare Write Response”.

2. GATT server reflects the data held in queue in GATT database when receives “Execute Write Request”.

Using this procedure is recommended when user wants highly reliable data communication. QE for BLE does not generate APIs of ReliableWrite operation. Therefore, user need to implement this procedure by using *R_BLE* APIs which provided BLE Protocol Stack. In addition, Characteristic Extended Properties Descriptor must have been added as a descriptor for ReliableWrite operation.

GATT server:
Before the operation, reserve a queue for receiving data using *R_BLE_GATTS_SetPrepareQueue* API. Size of the queue to be reserved should be greater than the total size of the characteristic which is able to ReliableWrite operation (e.g., If the total size is 6, specify value greater than or equal to 7). BLE Protocol Stack notifies the application of the event *BLE_XXX_EVENT_YYY_WRITE_REQ* indicating that “Prepare Write Request” has been received. BLE Protocol Stack notifies the application by the event *BLE_XXX_EVENT_YYY_WRITE_COMP* that GATT server received “Execute Write Request” and data held in the queue is reflected in GATT database.

GATT client:
Users can send “Prepare Write Request” using *R_BLE_GATTC_ReliableWrites* API. Users can receive “Prepare Write Response” for each data transmitted, and users can check the data in the event *BLE_GATTC_EVENT_RELIABLE_WRITE_TX_COMP*. After verifying whether GATT server is receiving the correct data, use *R_BLE_GATTC_ExecWrite* API with *BLE_GATTC_EXECUTE_WRITE_EXEC_FLAG* to send “Execute Write Request” for reflecting data in GATT database. If confirmed data is incorrect, use *R_BLE_GATTC_ExecWrite* API with *BLE_GATTC_EXECUTE_WRITE_CANCEL_FLAG* to send “Execute Write Request” to discard the data held by GATT server.

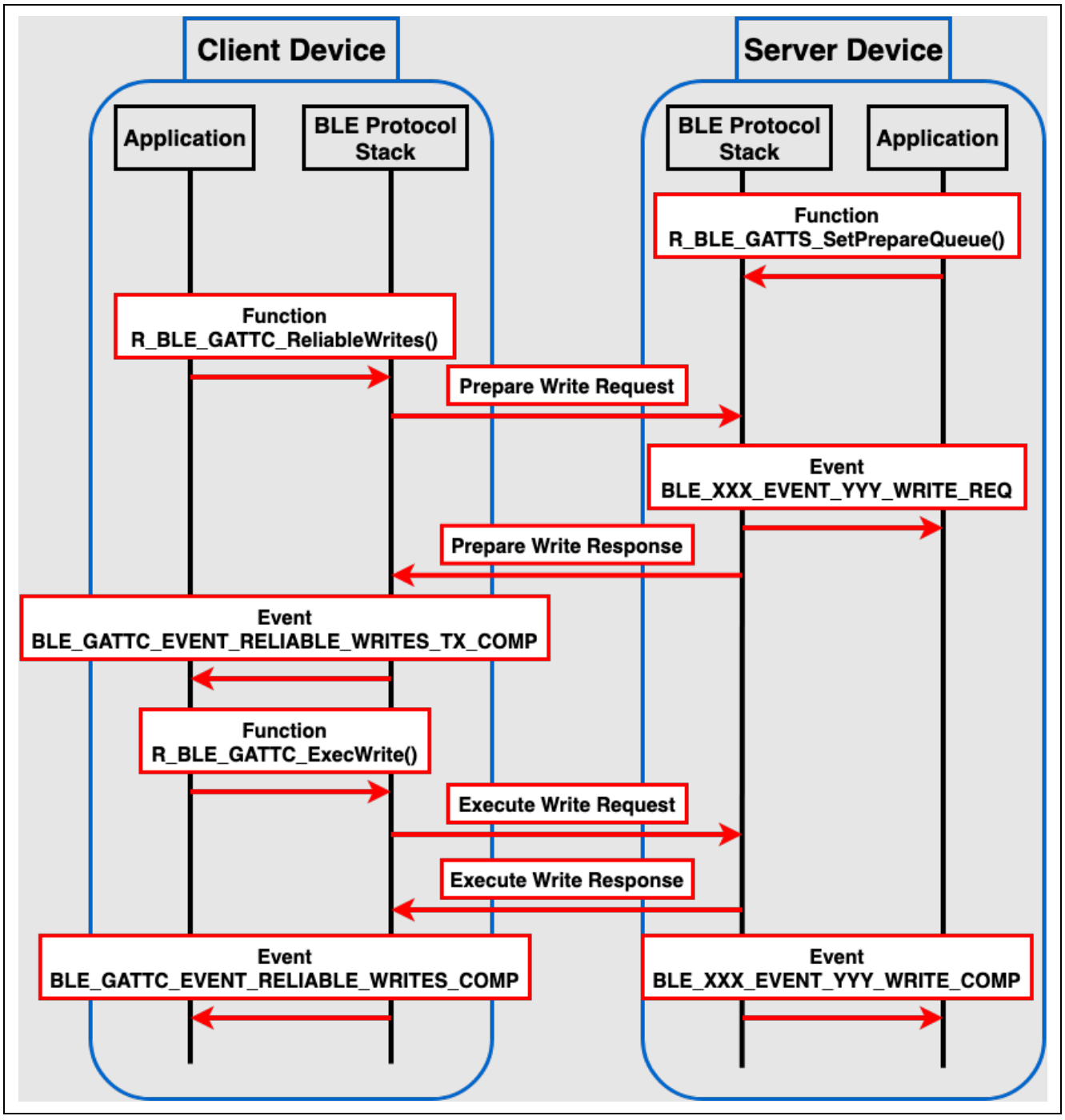


Figure 54. Flow of ReliableWrite operation

9.2.7 Broadcast Operation

Broadcast operation is procedure for transmitting data without a connection to an unspecified number of devices, as shown in Figure 55. The sender device is called Broadcaster and uses the Advertising operation. The receiver device is called Observer and uses the Scan operation. Because of the communication without a connection, there is no limit in number of devices that the Broadcaster can communicate at once, but it cannot be guaranteed that the receiver device is receiving data. QE for BLE does not generate APIs of Broadcast operation. Therefore, user needs to implement this procedure by using *R_BLE* APIs which provided BLE Protocol Stack. In addition, Server Characteristic Configuration Properties Descriptor must be added as a descriptor for Broadcast operation.

GATT server (Broadcaster):
Advertising operation is used for sending data. For an overview of advertising operation, refer to chapter 4. Note that when Advertising as Broadcast operation, there are following limitations:

- For the advertising type specification (section 4.2.1.1), set *adv_prop_type* field with value indicated in “Non-Connectable and Non-Scannable Undirected” or “Non-Connectable and Non-Scannable Directed” in Table 16.
- For Advertising Data configuration (section 4.4), users can broadcast service data by setting AD Structure which has “service Data (0x16 for 16-bit UUIDs, 0x21 for 128-bit UUIDs)” for AD Type and service UUIDs and data for AD Data. If the user wants to configure AD Structure with AD Type of ”Flags (0x01)”, do not set “LE Limited Discoverable Mode” or “LE General Discoverable Mode”.

GATT client (Observer):
Scan operation is used for receiving data. For an overview of scan operation, refer to chapter 5. There are no restrictions on the scan operation but set scan parameters so that user can receive the Advertising Event sent by Broadcaster.

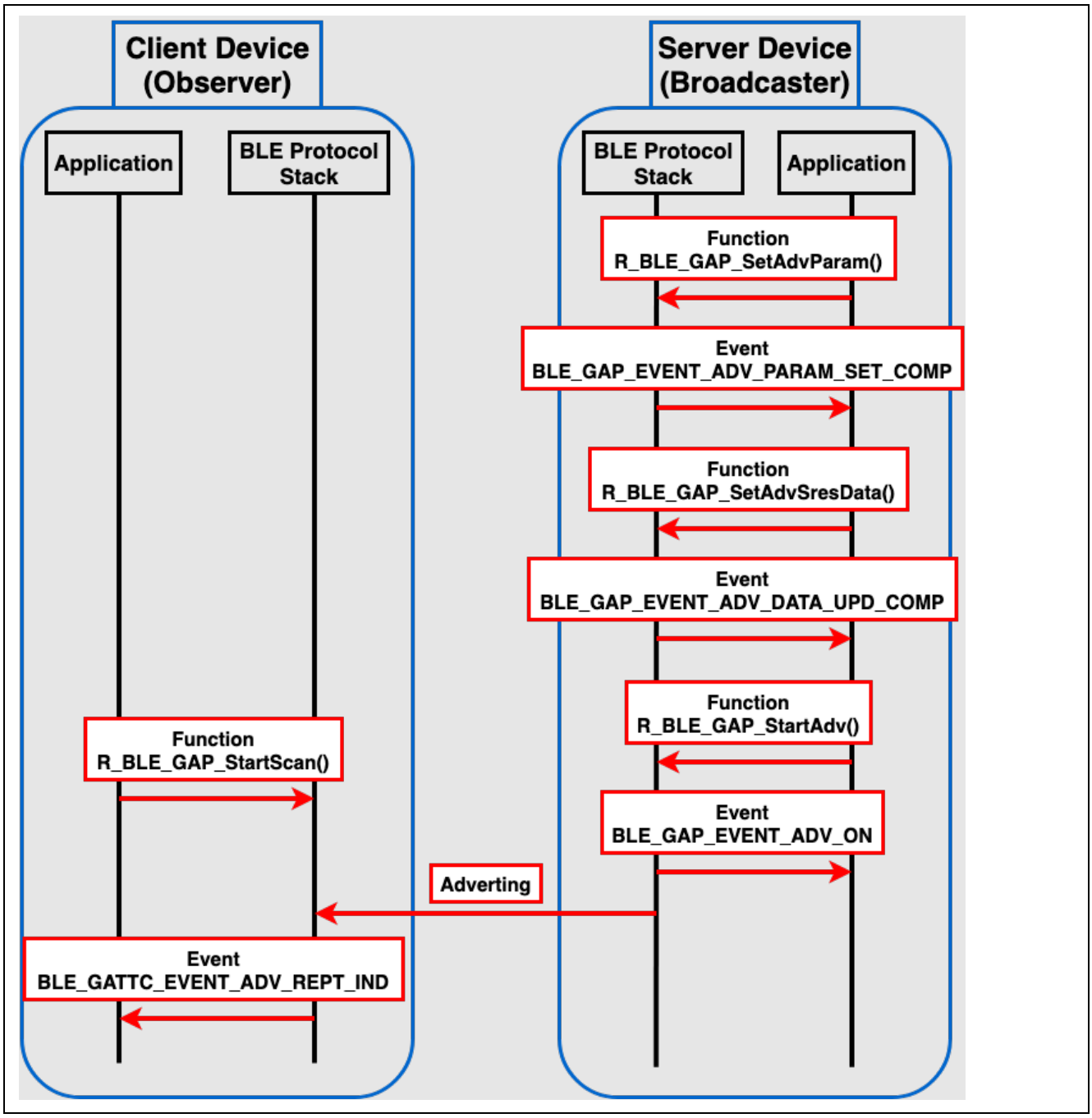


Figure 55. Flow of Broadcast operation

9.3 Example of using GATT Procedure

Refer to *BLE sample application (R01AN5402)*.

A. Appendix : Sample applications

Table A1 shows the sample applications for EK-RA4W1 attached to this APN.

Table A1. Sample applications

Application	Project	Reference
Beacon sample	ble_baremetal_ek_ra4w1_beacon	4.1 Advertising with abstraction API 4.5.2 Beacon
Peripheral sample	ble_baremetal_ek_ra4w1_peripheral	4.1 Advertising with abstraction API 4.5.1 Connection with Smart Phone 6.6.2 Connection to multiple central devices 7.4 Changing MTU88 8.1.1 Pairing Parameters 8.4 Privacy 9.2.4 Notification operation
Central sample	ble_baremetal_ek_ra4w1_central	5.1 Scan with abstraction API 5.2.4 Received information by scan 5.3.4 Advertising data filtering 6.1 Requesting connection with abstraction API 6.6.1 Connecting to multiple peripheral devices 7.1 Changing PHY 7.4 Changing MTU 8.1.1 Pairing Parameters 8.1.4 Pairing request 8.3.1 Request Encryption 8.4 Privacy 9.2.2 Write operation
Multi-role sample	ble_baremetal_ek_ra4w1_multirole	4.1 Advertising with abstraction API 4.5.1 Connection with Smart Phone 5.1 Scan with abstraction API 5.2.4 Received information by scan 5.3.4 Advertising data filtering 6.1 Requesting connection with abstraction API 6.6 Multiple connection 7.1 Changing PHY 7.4 Changing MTU 8.1.1 Pairing Parameters 8.1.4 Pairing request 8.3.1 Request Encryption 8.4 Privacy 9.2.2 Write operation 9.2.4 Notification operation

Table A2 shows the sample applications environment. The sample applications use FSP v3.7.0 (https://github.com/renesas/fsp/releases).

Table A2. Sample Applications environment

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio 2022-04
C compiler	GCC ARM Embedded V10
Board	EK-RA4W1
QE for BLE[RA] version	1.5.0
QE utility version	1.4.1
SEGGER J-Flash version	6.86

How to import the sample application in e^2 studio is described below.

(1) Right click the application developer's guide (Title: RA4W1 Group Bluetooth Low Energy Application Developer's Guide Application Note, Document No: R01AN5653EJYYYY (YYYY: version)) on Smart Browser and select "Sample Code (import projects)".

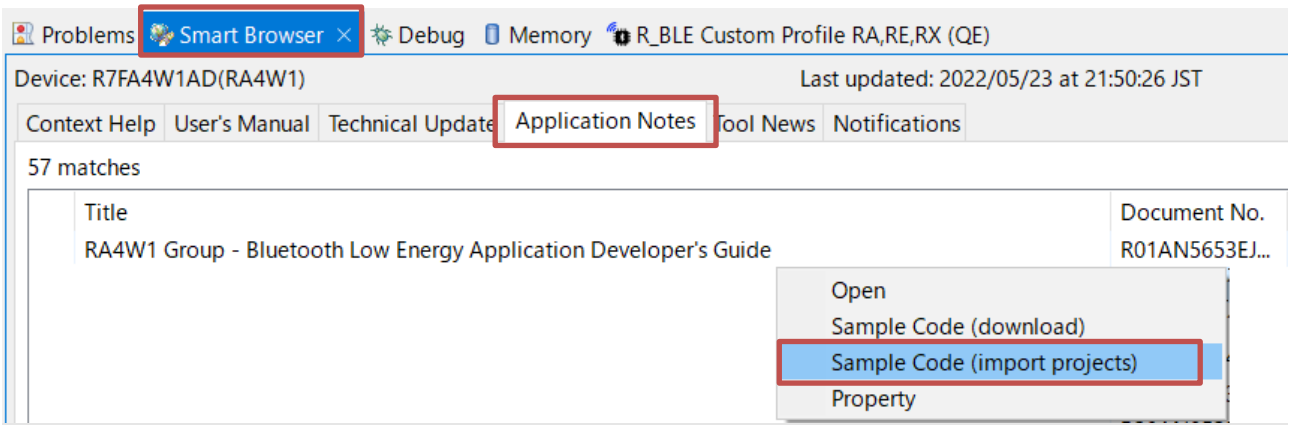


Figure A1. Sample application project import

(2) Download r01an5653xxYYYY-ra4w1-ble-adev.zip (YYYY: version) to the desired location. After the download has been completed, “Select import package” window is displayed. Select a sample application project.

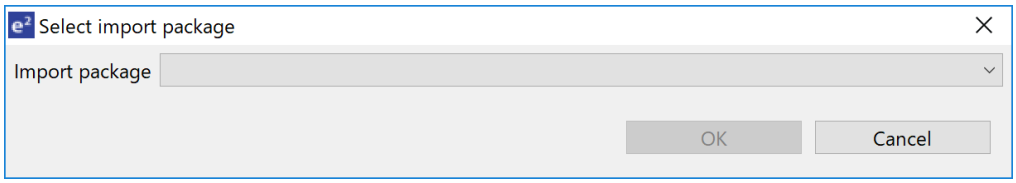


Figure A2. Import package selection

(3) When “Finish” button on “Import” window has been pressed, the sample application project is imported.

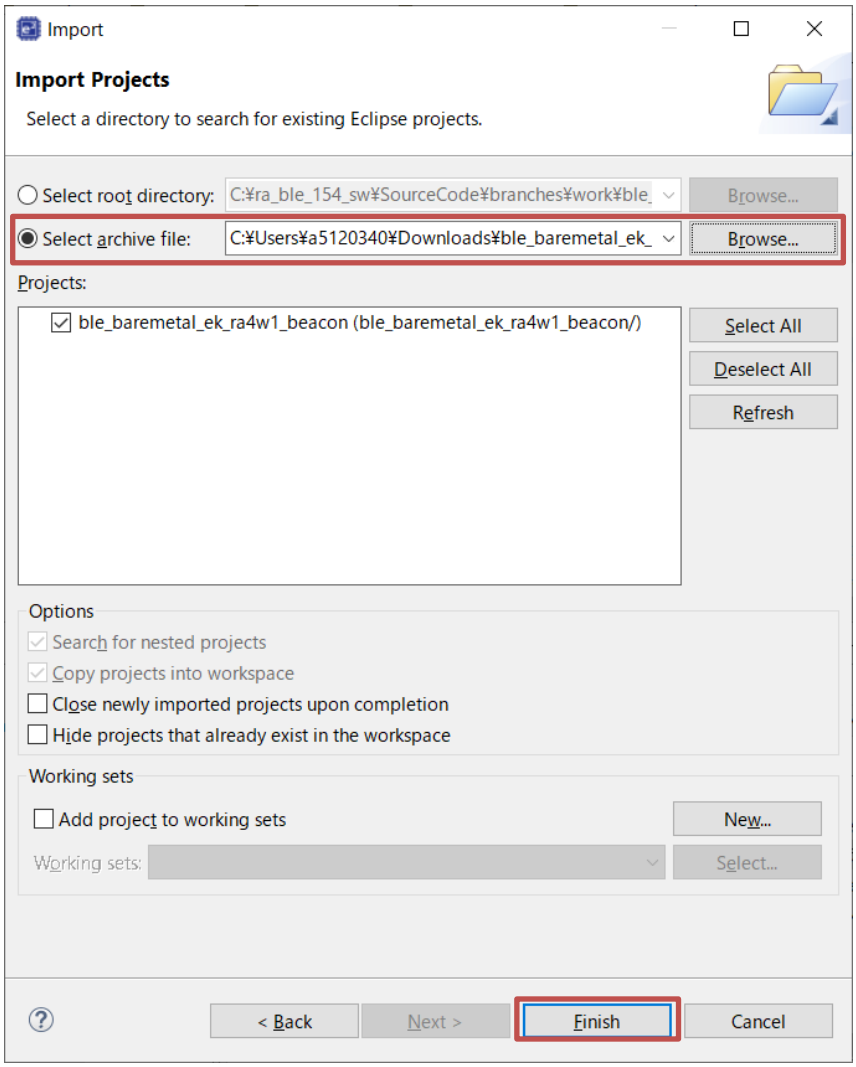


Figure A3. Project import

A.1 Beacon sample

A.1.1 Remote devices

Advertising packets from the beacon sample can be received with scan on the following remote devices.

- iOS device
- Android device

A.1.2 Operations

The beacon sample starts Non connectable undirected advertising (ADV_NONCONN_IND) after the boot. The beacon sample stops advertising by pressing SW1. After stopping advertising, the beacon sample restarts advertising by pressing SW1.

A.1.3 Advertising Data

Specify the following Advertising Data type with BLE_APP_BEACON_TYPE in app_main.c.

0: iBeacon (default)
1: Eddystone
2: broadcast mode

Each Advertising Data type is described below.

- iBeacon

  The iBeacon specification is published in https://developer.apple.com/ibeacon.

  The details of the Advertising Data are follows.

```
/* Advertising data */
static uint8_t gs_adv_data[] =
{
    /* TODO: Modify advertise data. Value of Data Flag is defined in
    https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile */

    /* Flag (mandatory) */
    2,          /**< Data Size */
    0x01,       /**< Data Type: Flag */
    (BLE_GAP_AD_FLAGS_LE_GEN_DISC_MODE | BLE_GAP_AD_FLAGS_BR_EDR_NOT_SUPPORTED),    /**< Data Value */

    /* Manufacturer data */
    0x1A,       /**< Data Size */
    0xFF,       /**< Data Type: Manufacturer data */
    0x4C, 0x00, /**< Company ID: Apple */
    0x02, 0x15, /**< Beacon Type:  */
    0x32, 0x46, 0x6a, 0x3a, 0x75, 0x52, 0xdb, 0xb4,
    0x97, 0x49, 0x19, 0x70, 0xd8, 0x56, 0x98, 0xaa,  /**< UUID: 32466a3a-7552-dbb4-9749-1970d85698aa */
    0x00, 0x01, /**< Major: 1 */
    0x00, 0x00, /**< Minor: 0 */
    0x00, /**< Measured Power:  */
};
```

Code A1. Default Advertising Data for iBeacon

You need to change the above UUID to your application specific value and the Major and Minor version to your application version. Measured Power are needed to change the power value measured according to iBeacon specification.

- Eddystone

  The Eddystone specification is published in https://github.com/google/eddystone.

  The beacon sample provides the Eddystone-URL type. This sample does not support the Eddystone Configuration GATT Service. The details of the Advertising Data are follows.

```
/* Advertising data */
static uint8_t gs_adv_data[] =
{
    /* TODO: Modify advertise data. Value of Data Flag is defined in
    https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile */

    /* Flag (mandatory) */
    2,          /**< Data Size */
    0x01,       /**< Data Type: Flag */
    (BLE_GAP_AD_FLAGS_LE_GEN_DISC_MODE | BLE_GAP_AD_FLAGS_BR_EDR_NOT_SUPPORTED),    /**< Data Value */

    /* Complete list of 16bit Service UUIDs */
    3,          /**< Data Size */
    0x03,       /**< Data Type: Complete list of 16bit Service UUIDs */
    0xAA, 0xFE, /**< 16bit Eddystone UUID */

    /* Service Data */
    14,         /**< Data Size */
    0x16,       /**< Data Type: Service Data */
    0xAA, 0xFE, /**< 16bit Eddystone UUID */
    0x10,       /**< Frame Type: URL */
    0x00,       /**< Tx power: 0 dBm */
    0x01,       /**< URL Scheme: https://www. */
    0x72, 0x65, 0x6E, 0x65, 0x73, 0x61, 0x73, 0x07 /**< Encoded URL: renesas.com */
};
```

Code A2. Default Advertising Data for Eddystone

You need to change the above Data Size, URL Scheme and Encoded URL in Service Data to suit to your application.

- Broadcast mode

  The broadcast mode Advertising Data does not include “LE General Discoverable Mode” and “LE Limited Discoverable Mode” in the Flags to meet the Broadcast Mode condition defined in Bluetooth Core Specification.

```
/* Advertising Data */
static uint8_t gs_adv_data[] =
{
    /* TODO: Modify advertise data. Value of Data Flag is defined in
https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile */

    /* Flag (mandatory) */
    2,          /**< Data Size */
    0x01,       /**< Data Flag: Flag */
    BLE_GAP_AD_FLAGS_BR_EDR_NOT_SUPPORTED,    /**< Data Value */

    /* Complete Local Name */
    9,         /**< Data Size */
    0x09,       /**< Data Flag: Complete Local Name */
    'R', 'B', 'L', 'E', '-', 'A', 'D', 'V', /**< Data Value */

};
```

Code A3. Default Advertising Data for broadcast mode

A.1.4 Configuration option

Table A3 shows the BLE configuration options changed from the default for the beacon sample.

Table A3. Changed configuration options

Macro (SC display name)	Value
BLE_CFG_LIBRARY_TYPE (BLE Driver (r_ble_compact))	2: Compact
BLE_ABS_CFG_RF_CONNECTION_MAXIMUM (Maximum number of connections)	1

A.1.5 Configurable parameters

(1) Address

Specify the following address type with BLE_BEACON_ADDR_TYPE in app_main.c.

BLE_GAP_ADDR_RAND : Static address (default)
BLE_GAP_ADDR_RPA_ID_RANDOM : RPA(static) address

(2) Advertising Data, Advertising parameters

The gs_adv_data (Advertising Data), gs_adv_param (Advertising parameters) variables in app_main.c are configurable according to each beacon specification in “A.1.3 Advertising Data”.

A.1.6 Command

If the beacon sample uses RPA, it supports the following command to display the current RPA.

```
beacon lrpa
```

A.2 Peripheral sample

A.2.1 Remote devices

The peripheral sample supports connections with the following remote devices.

- Central sample
- Multi-role sample
- iOS device
- Android device

A.2.2 Operations

The peripheral sample works as follows.

- The peripheral sample starts Connectable undirected advertising (ADV_IND) after the boot.
  It starts fast advertising (interval: 30ms) in the first 30s and changes to slow advertising (interval: 1000ms) in the next 30s.
- By scanning from a remote device, it is detected as the “RBLE-P-DEV” device name.

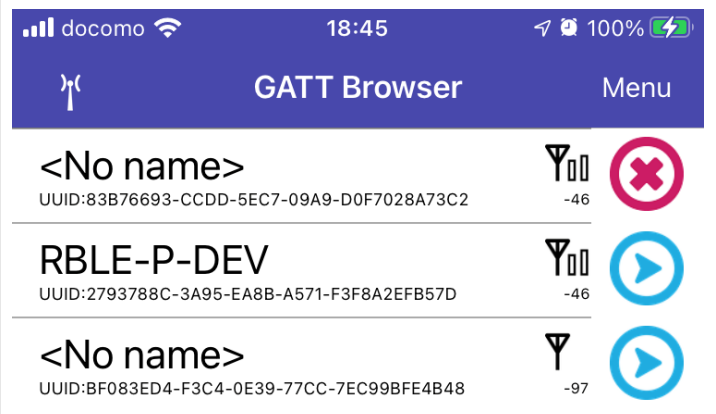


Figure A4. Scan result on central device

- After connection establishment, if the simultaneous multiple connections feature is not supported, the peripheral sample stops the advertising. Otherwise, it continues the advertising. For changing the simultaneous multiple connections feature, refer “(3)”.
- If a remote device searches GATT Services in the peripheral sample, the following service and characteristics are detected.

Table A4. Detected service and characteristics

Service, characteristic	UUID
LED Switch service	58831926-5F05-4267-AB01-B4968E8EFCE0
Switch State characteristic	58837F57-5F05-4267-AB01-B4968E8EFCE0
LED Blink Rate characteristic	5883C32F-5F05-4267-AB01-B4968E8EFCE0

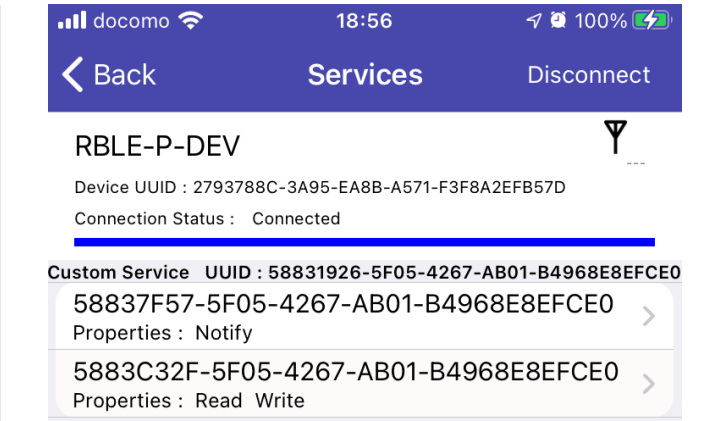


Figure A5. Detected GATT service and characteristics

- The peripheral sample sets BLE_GATT_DB_SER_SECURITY_UNAUTH | BLE_GATT_DB_SER_SECURITY_ENC to the second parameter which indicates LED Switch service security requirement in the gs_gatt_service variable in gatt_db.c. Therefore, if a remote device accesses a characteristic in LED Switch service, it requires pairing.

```
static const st_ble_gatts_db_serv_cfg_t gs_gatt_service[] =
{
    /* some code is omitted */
    /* LED Switch Service */
    {
        /* Num of Services */
        {
            1,
        },
        /* Description */
        BLE_GATT_DB_SER_SECURITY_UNAUTH | BLE_GATT_DB_SER_SECURITY_ENC,
        /* Service Start Handle */
        0x0010,
        /* Service End Handle */
        0x0015,
        /* Characteristic Start Index */
        6,
        /* Characteristic End Index */
        7,
    },

};
```

Code A4. LED Switch service security requirement

- After the remote device enables the Switch State characteristic Notification, the peripheral sample sends a Notification after SW1 is pressed on the board.
- If the remote device writes a value to the LED Blink Rate characteristic, the LED on the board blinks at the value x 100ms interval. When writing zero to the characteristic, the LED will turn off.
- If the link is disconnected, the peripheral sample restarts advertising.
- When press the reset button while pressing SW1, the bonding information is deleted.

A.2.3 Configuration option

Table A5 shows the BLE configuration options changed from the default for the peripheral sample.

Table A5. Changed configuration options

Macro (SC display name)	Value
BLE_CFG_LIBRARY_TYPE (BLE Driver (r_ble_balance))	1: Balance
BLE_ABS_CFG_RF_CONNECTION_MAXIMUM (Maximum number of connections)	3
BLE_ABS_CFG_NUMBER_BONDING (Remote Device Bonding Number)	3
BLE_ABS_CFG_ENABLE_SECURE_DATA (Store Security Data)	1: Enable

A.2.4 Configurable parameters

(1) Address

Specify the following address type with BLE_PERIPHERAL_ADDR_TYPE in app_main.c.

BLE_GAP_ADDR_RAND : Static address (default)
BLE_GAP_ADDR_RPA_ID_RANDOM : RPA(static) address

(2) Advertising Data, Scan Response Data, Advertising parameters

The gs_adv_data (Advertising Data), gs_sres_data (Scan Response Data) and gs_adv_param (Advertising parameters) variables in app_main.c are configurable. If you change the device name included in gs_adv_data or gs_sres_data and the peripheral sample connects with a Central sample, change the scan filter in the Central sample.

(3) Simultaneous multiple connections feature

If the BLE_PERIPHERAL_MULTI_CONNS macro in app_main.c is enabled, the peripheral sample supports the simultaneous multiple connections feature.

A.3 Central sample

A.3.1 Remote devices

The central sample supports connection with the following remote device.

- Peripheral sample

A.3.2 Operations

The central sample works as follows.

- The central sample starts scan to detect a peripheral sample after booting.
  It starts fast scan (interval: 60ms, window: 30ms) in a first 10s and then continues slow scan (interval: 1200ms, window: 11.25ms) for 10s. After stopping scan, the central sample restarts scan by pressing SW1.
- After detecting a peripheral sample, the central sample stops scan. It sends a connection request to the detected peripheral sample.
- After connection establishment, the packet length is updated.
- After packet length update, a MTU change request is sent to the remote device.
  If the PHY is changed, a PHY change request is sent to the remote device before sending a MTU change request.
- When receiving a MTU change response from the remote device, the central sample discovers LED Switch service in Table A4.
- After service discovery, the central sample writes 1 to the CCCD of the Switch State characteristic.
- If pairing is not completed, the peripheral sample returns an error. When the central sample receives the error, it starts pairing. If pairing is completed but encryption is not completed, the peripheral sample returns an error. When the central sample receives the error, it starts encryption.
- When pairing and encryption are completed, the central sample writes 1 to the CCCD again.
- Then after pressing SW1 on the peripheral sample board, the switch state characteristic is sent to the central sample as Notification.
- When press the reset button while pressing SW1, the bonding information is deleted.

A.3.3 Configuration option

Table A6 shows the BLE configuration options changed from the default for the central sample.

Table A6. Changed configuration options

Macro (SC display name)	Value
BLE_CFG_LIBRARY_TYPE (BLE Driver (r_ble_balance))	1: Balance
BLE_ABS_CFG_RF_CONNECTION_MAXIMUM (Maximum number of connections)	3
BLE_ABS_CFG_NUMBER_BONDING (Remote Device Bonding Number)	3
BLE_ABS_CFG_ENABLE_SECURE_DATA (Store Security Data)	1: Enable

A.3.4 Configurable parameters

(1) Address

Specify the following address type with BLE_CENTRAL_ADDR_TYPE in app_main.c.

BLE_GAP_ADDR_RAND : Static address (default)
BLE_GAP_ADDR_RPA_ID_RANDOM : RPA(static) address

(2) Scan parameters, connection parameters

The gs_scan_phy_param, gs_scan_param (scan parameters) and gs_conn_phy_param, gs_conn_param (connection parameters) variables in app_main.c are configurable. If you change the peripheral sample device name included in the advertising data or scan response data, change the gs_filter_data as scan filter.

(3) PHY

If you want to change PHY from 1M to 2M after connection establishment, set the BLE_APP_CHANGE_PHY_2M macro in app_main.c to 1. The default value is zero.

A.4 Multi-role sample

A.4.1 Topology

The multi-role sample connects with a central device and a peripheral sample and bridges the characteristic written from the central device or notified from the peripheral sample. Figure A6 shows the multi-role sample topology.

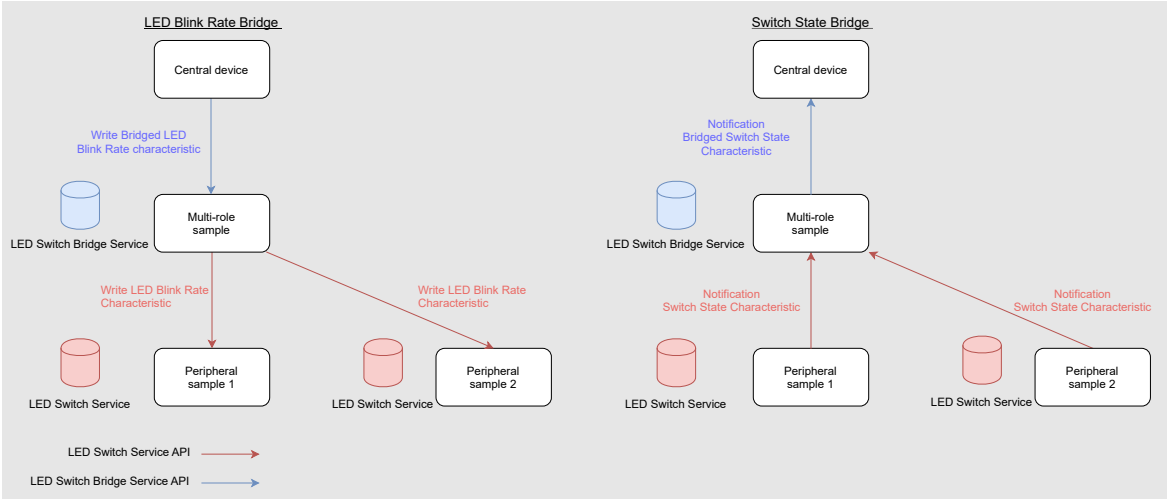


Figure A6. Multi-role sample topology

The multi-role sample adds LED Switch Bridge service which includes Bridged Switch State characteristic (Figure A7) and Bridged LED Blink Rate characteristic (Figure A8) to implement the bridge between the central device and the peripheral sample. The Bridged Switch State characteristic includes a Bluetooth device address to indicate which peripheral sample sends a notification. After conversion from the LED Switch Bridge service characteristic to the LED Switch service characteristic, the multi-role sample sends write request for the LED Switch service to the peripheral samples. Similarly, after conversion from the LED Switch service characteristic to the LED Switch Bridge service characteristic, the multi-role sample sends notification for the LED Switch Bridge service to the central device.

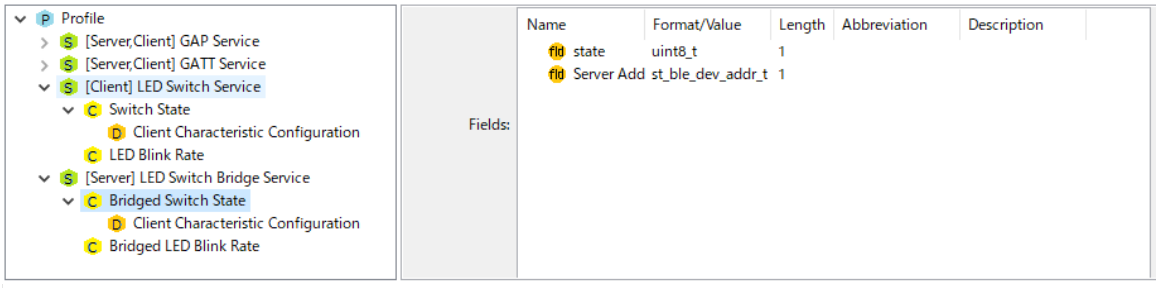


Figure A7. Bridged Switch State characteristic

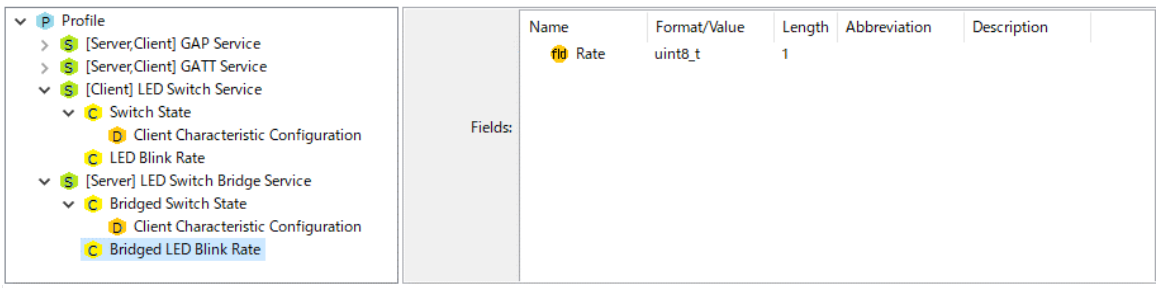


Figure A8. Bridged LED Blink Rate characteristic

A.4.2 Remote devices

The multi-role sample supports connection with the following remote device.

[Peripheral role] :

- iOS device
- Android device

[Central role] :

- Peripheral sample

A.4.3 Operations

The multi-role sample works as follows.

[Connection to the central device] :

- The multi-role sample starts Connectable undirected advertising (ADV_IND) after the boot.
  It starts fast advertising (interval: 30ms) in the first 30s and changes to slow advertising (interval: 1000ms) in the next 30s.
- By scanning from a remote device, it is detected as the “RBLE-MULTI-DEV” device name.



Figure A9. Scan result on central device

- After connection establishment, the multi-role sample stops advertising. The multi-role sample connects simultaneously to only one central device.
- If a remote device searches GATT Services in the multi-role sample, the following service and characteristics are detected.

Table A7. Detected service and characteristics

Service, characteristic	UUID
LED Switch Bridge service	908DCB17-7F42-44AC-AB9D-C36F63DCEBD8
Bridged Switch State characteristic	4CC8C6EC-3954-41D1-8CFF-3F2FE5EC0180
Bridged LED Blink Rate characteristic	458B6862-6D2C-4356-8B2E-B88BCE7F0C84

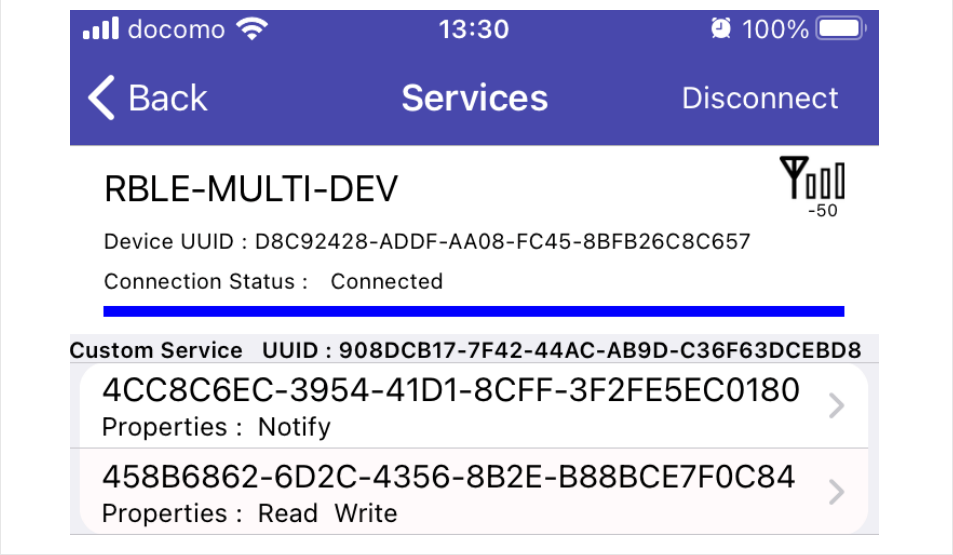


Figure A10. Detected GATT service and characteristics

- The multi-role sample sets BLE_GATT_DB_SER_SECURITY_UNAUTH | BLE_GATT_DB_SER_SECURITY_ENC to the second parameter which indicates LED Switch Bridge service security requirement in the gs_gatt_service variable in gatt_db.c. Therefore, if a remote device accesses a characteristic in LED Switch Bridge service, it requires pairing.

```
static const st_ble_gatts_db_serv_cfg_t gs_gatt_service[] =
{
    /* Some code is omitted */
    /* LED Switch Bridge Service */
    {
        /* Num of Services */
        {
            1,
        },
        /* Description */
        BLE_GATT_DB_SER_SECURITY_UNAUTH | BLE_GATT_DB_SER_SECURITY_ENC,
        /* Service Start Handle */
        0x0010,
        /* Service End Handle */
        0x0015,
        /* Characteristic Start Index */
        6,
        /* Characteristic End Index */
        7,
    },
};
```

Code A5. LED Switch Bridge service security requirement

- If the link with the central device is disconnected, the multi-role sample restarts advertising.

[Connection to the peripheral sample] :

- The multi-role sample starts scan to detect a peripheral sample after pressing SW1.
  It starts fast scan (interval: 60ms, window: 30ms) in a first 10s and then continues slow scan (interval: 1200ms, window: 11.25ms) for 10s. After stopping scan, the central sample restarts scan by pressing SW1.
- After detecting a peripheral sample, the multi-role sample stops scan. It sends a connection request to the detected peripheral sample.
- After connection establishment, the packet length is updated.

- After packet length update, a MTU change request is sent to the remote device.
  If the PHY is changed, a PHY change request is sent to the remote device before sending a MTU change request.
- When receiving a MTU change response from the remote device, the multi-role sample discovers LED Switch service.
- After service discovery, the multi-role sample writes 1 to the CCCD of the Switch State characteristic.
- If pairing is not completed, the peripheral sample returns an error. When the multi-role sample receives the error, it starts pairing. If pairing is completed but encryption is not completed, the peripheral sample returns an error. When the multi-role sample receives the error, it starts encryption.
- When pairing and encryption are completed, the multi-role sample writes 1 to the CCCD again.
- Then after pressing SW1 on the peripheral sample board, the switch state characteristic is sent to the multi-role sample as Notification.

[Delete the bonding information] :

- When press the reset button while pressing SW1, the bonding information is deleted.

[Switch State characteristic bridge] :

After the following configurations are complete, when pressing SW1, the multi-role sample converts the Switch State characteristic in the LED Switch State service to the Bridged Switch State characteristic in the LED Switch Bridged service and sends write request including the data. Therefore, the Bridged Switch State characteristic is notified to the central device as shown Figure A11. This characteristic includes the Switch State and the peripheral sample address as shown Figure A7.

- After connection with the central device, set 1 to the CCCD of the Bridged Switch State characteristic from the central device.
- After connection with the peripheral sample, the multi-role sample automatically sets 1 the CCCD of the Switch State characteristic.

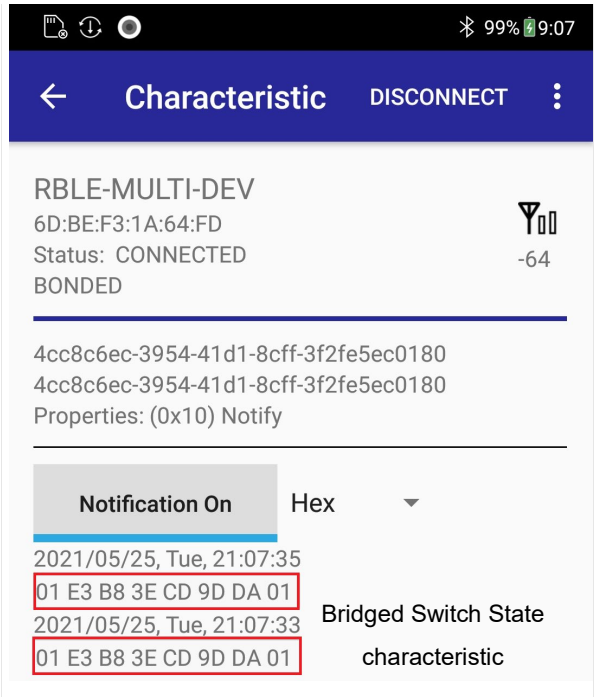


Figure A11. Bridged Switch State characteristic notification to central device

[LED Blink Rate characteristic bridge] :

After connection with the central device and the peripheral sample, when the central device writes a value to the Bridged LED Blink Rate characteristic as shown Figure A12, the LED blinks at the value x 100ms interval.

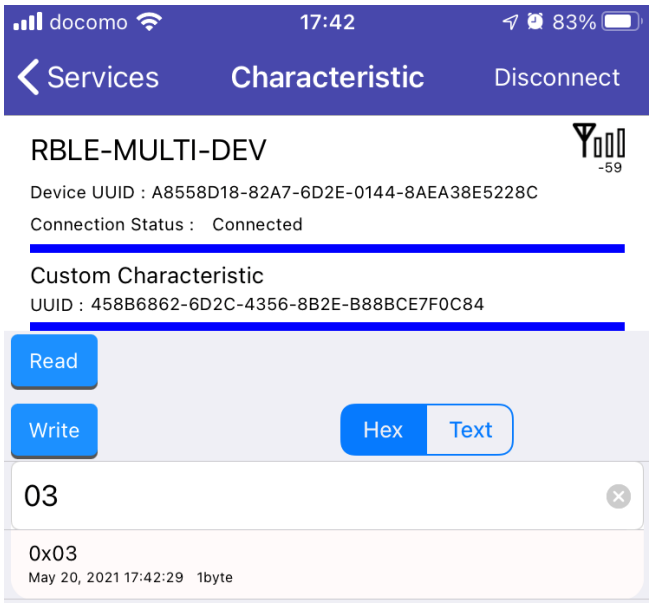


Figure A12. Bridged LED Blink Rate characteristic write from central device

A.4.4 Configuration option

Table A8 shows the BLE configuration options changed from the default for the multi-role sample.

Table A8. Changed configuration options

Macro (SC display name)	Value
BLE_CFG_LIBRARY_TYPE (BLE Driver (r_ble_balance))	1: Balance
BLE_ABS_CFG_RF_CONNECTION_MAXIMUM (Maximum number of connections)	3
BLE_ABS_CFG_NUMBER_BONDING (Remote Device Bonding Number)	3
BLE_ABS_CFG_ENABLE_SECURE_DATA (Store Security Data)	1: Enable

A.4.5 Configurable parameters

(1) Address

Specify the following address type with BLE_MULTIROLE_ADDR_TYPE in app_main.c.

BLE_GAP_ADDR_RAND : Static address (default)
BLE_GAP_ADDR_RPA_ID_RANDOM : RPA(static) address

(2) PHY

If you want to change PHY from 1M to 2M after connection establishment as central role, set the BLE_APP_CHANGE_PHY_2M macro in app_main.c to 1. The default value is zero.

(3) Advertising Data, Scan Response Data, Advertising parameters

The gs_adv_data (Advertising Data), gs_sres_data (Scan Response Data) and gs_adv_param (Advertising parameters) variables in app_main.c are configurable.

(4) Scan parameters, connection parameters

The gs_scan_phy_param, gs_scan_param (scan parameters) and gs_conn_phy_param, gs_conn_param (connection parameters) variables in app_main.c are configurable. If you change the peripheral sample device name included in the advertising data or scan response data, change the gs_filter_data as scan filter.

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Jan.13.2021	–	First edition issued.
1.01	Aug.30.2021	–	• Add ROM/RAM usage for extended/balance/compact configuration to section 2.4.
1.02	Oct.11.2021	-	• Update section 4.1.2, section 5.1.2, section 5.3.1, section 6.1.2, section 6.4, section 8.2.2, section 8.2.3, section 8.4 according to the abstraction APIs support for privacy and local key management in FSP 3.3.
1.20	Jun.30.2022	-	• Add “A Appendix : Sample applications”.
		Program	• Add sample applications.
		58	• Add how to retain the connection handles.
		104-107	• Add the relationship between the bonding number and the maximum simultaneous connection number.
		108-109	• Add the description about deleting the bonding information.
		109	• Add how to connect to only the bonded device.
		110-114	• Add the sequence if encryption was failed.
		116-129	• Add the supplementary description of RPA.
1.30	Dec.27.2022	1, 7, 35, 49, 92, 95, 96, 98, 99, 101, 102, 110	• Added recommendations and risk descriptions based on the "Bluetooth® Security and Privacy Best Practices Guide" published by the Bluetooth SIG so that implementers can select the best practices for security and privacy.
		35, 51	• Added about RPA default update interval and API to change update interval.
		96	• Added about notification information when LE legacy pairing starts with Secure Connection Only specified.
		120	• When the local device uses RPA, the setting value of Identity Address and IRK of the remote device to be associated is changed from "dummy" to "all 0x00", and removed registration conditions ("only the local device is uses RPA or the local device has not paired with remote device before").
		6	• Added the note about QE for BLE and QE Utility V1.40 or later.
		19	• Added the note about Public device address and Static address.
		20	• Added the note about the device identification.
		22	• Added the explanation about R_BLE_Open and clock frequency.
		24	• Added the explanation about initializing RF hardware state.
		28	• Added the explanation about R_BLE_Close and RF H/W stop.
		31, 35, 53, 60	• Added that whitelist cannot be set while it is used.
		31, 35	• Added explanation about o_addr for Privacy.
		32	• Added that parameters cannot be changed during Advertising.
		34, 51, 58	• Added the explanation about PHY of Advertising / Scan / Connection.
		77	• Added an explanation for the reason for disconnection.
		78	• Added libraries and roles that can change communication parameters.

		79	• Added the explanation about PHY that allows modification.
		102	• Added the explanation about result when pairing fails.
		117	• Added a figure for privacy procedures.

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.