RA4W1 Group

Bluetooth Low Energy Application Developer's Guide

Introduction

This document describes how to make Bluetooth Low Energy applications and provides some hints for making Bluetooth Low Energy applications.

Target Device

RA4W1 Group

Related Documents

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

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

RA4W1 Group User's Manual: Hardware (R01UH0883)

RA 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 (R01AN5428)

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)

Contents

1. Overview

This document describes how to make Bluetooth Low Energy applications and provides some hints for making Bluetooth Low Energy applications.

1.1 Development environment

This section describes 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® 10 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 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	e^2 studio	2021-04	Integrated development environment (IDE) for Renesas devices.
	GCC ARM Embedded	V9	C/C++ Compiler. (download from e^2 studio installer)
	Renesas Flexible Software Package (FSP)	V3.2.0	Software package for making applications for the RA microcontroller series.
	QE for BLE[RA]	V1.2.0	Generates the source codes (BLE base skeleton program) as a base for the BLE Application and the BLE Profile. https://www.renesas.com/qe-ble
	SEGGER J-Flash	V6.86	Tool for programming the on-chip flash memory of microcontrollers.
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 case in BLE. User can interconnect with existing BLE device by using such an application profile. On the other hand, it is necessary to newly design application profile as well as user application when user wants to perform new BLE bidirectional communication. User needs to design following items when user make the 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.
- Security level.

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

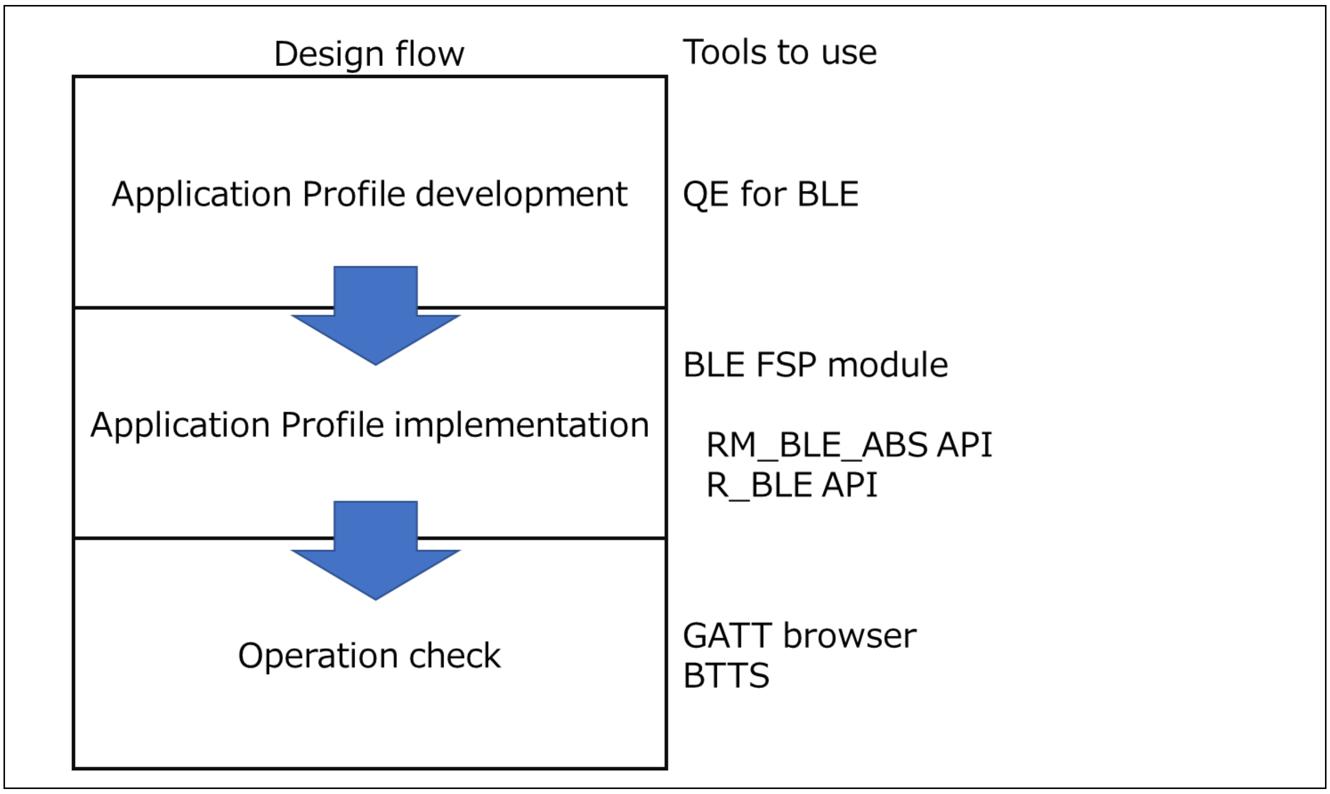


Figure 1. Bluetooth LE application development procedure and auxiliary tools

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

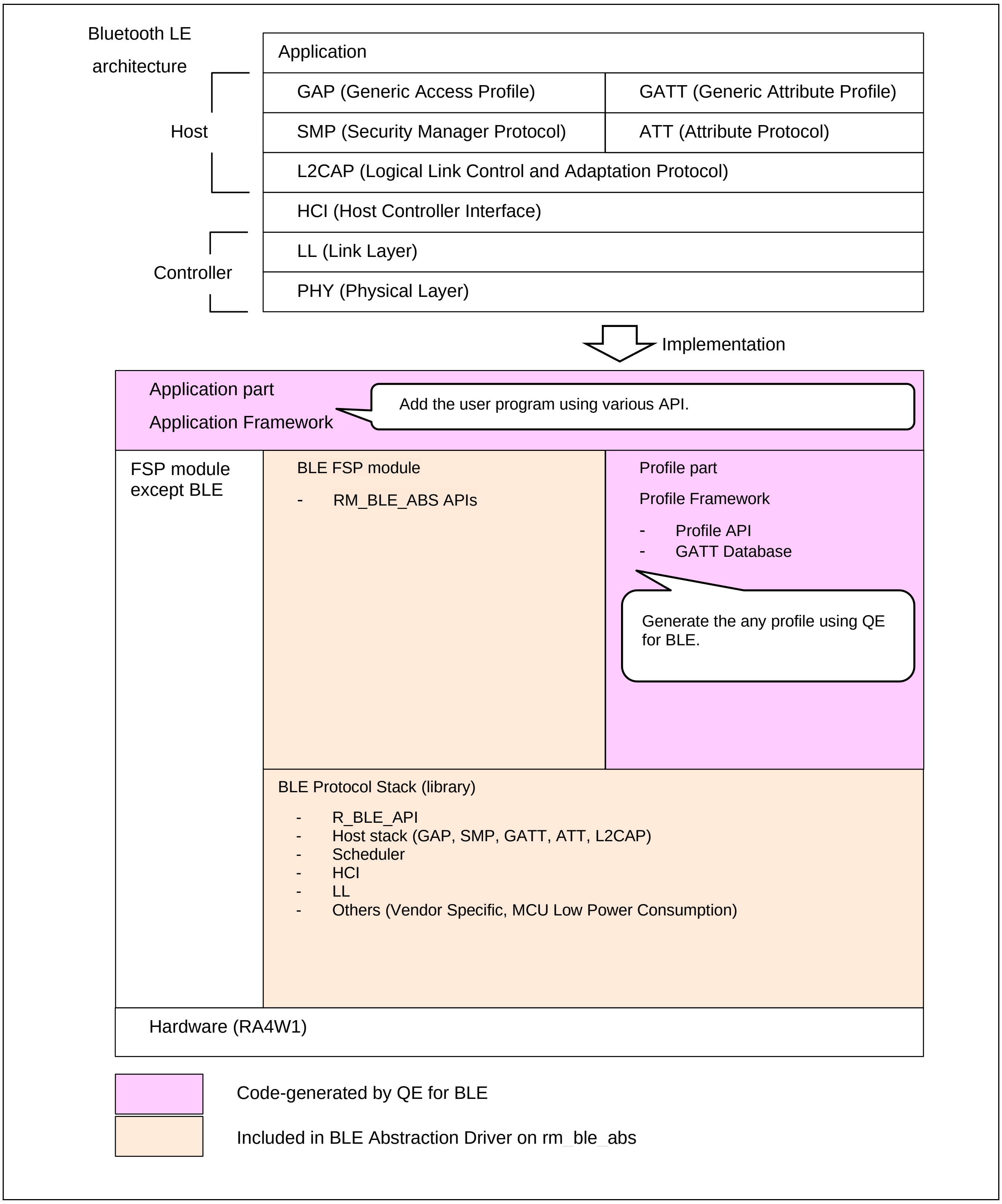

Bluetooth LE architecture
Host
Controller
Application
GAP (Generic Access Profile)
GATT (Generic Attribute Profile)
SMP (Security Manager Protocol)
ATT (Attribute Protocol)
L2CAP (Logical Link Control and Adaptation Protocol)
HCI (Host Controller Interface)
LL (Link Layer)
PHY (Physical Layer)
Implementation
Application part
Application Framework
Add the user program using various API.
FSP module except BLE
BLE FSP module
- RM_BLE_ABS APIs
Profile part
Profile Framework
- Profile API
- GATT Database
Generate the any profile using QE for BLE.
BLE Protocol Stack (library)
- R_BLE_API
- Host stack (GAP, SMP, GATT, ATT, L2CAP)
- Scheduler
- HCI
- LL
- Others (Vendor Specific, MCU Low Power Consumption)
Hardware (RA4W1)
Code-generated by QE for BLE
Included in BLE Abstraction Driver on rm_ble_abs

Figure 2. Software structure

1.2.1 Flexible Software Package

BLE Protocol Stack is included in *BLE Abstraction Driver on rm_ble_abs* which is a part of Flexible Software Package. The driver provides the BLE features that comply with the Bluetooth Core Specification version 5.0 defined by Bluetooth SIG. User can add the driver to their own project from 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 Abstraction Driver on rm_ble_abs* to user's project.

1.2.2 QE for BLE

QE for BLE is tool for designing BLE application profiles. QE for BLE tool can generate profile and BLE application skeleton code. And user can modify QE for BLE generated codes according to use case. Refer to *Bluetooth Low Energy Profile Developer's Guide(R01AN5428)* 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 for accessing to GATT Server. User can confirm about Bluetooth Low Energy primitive communication and GATT database structure on GATT server and so on from smartphone which installed this application. This application can download 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 typical BLE applications on RA4W1 is that accepts connection from PC, smartphones, etc. and operates 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 Master, BLE Protocol Stack automatically establishes a connection and notifies *BLE_GAP_EVENT_CONN_IND* event.
	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 Abstraction Driver

BLE Abstraction Driver is a part of Flexible Software Package. The driver provides BLE features that comply with the Bluetooth Core Specification version 5.0 defined by Bluetooth SIG.

2.1 Supported features

Table 5 shows BLE Abstraction Driver 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 Master and Slave roles, and can operate as Master when connecting to one remote device and as Slave 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.	
Addendum 2	Appearance Data Type Appearance characteristic can be used in GAP service.	

Bluetooth Version	LE features and description	Remark
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.

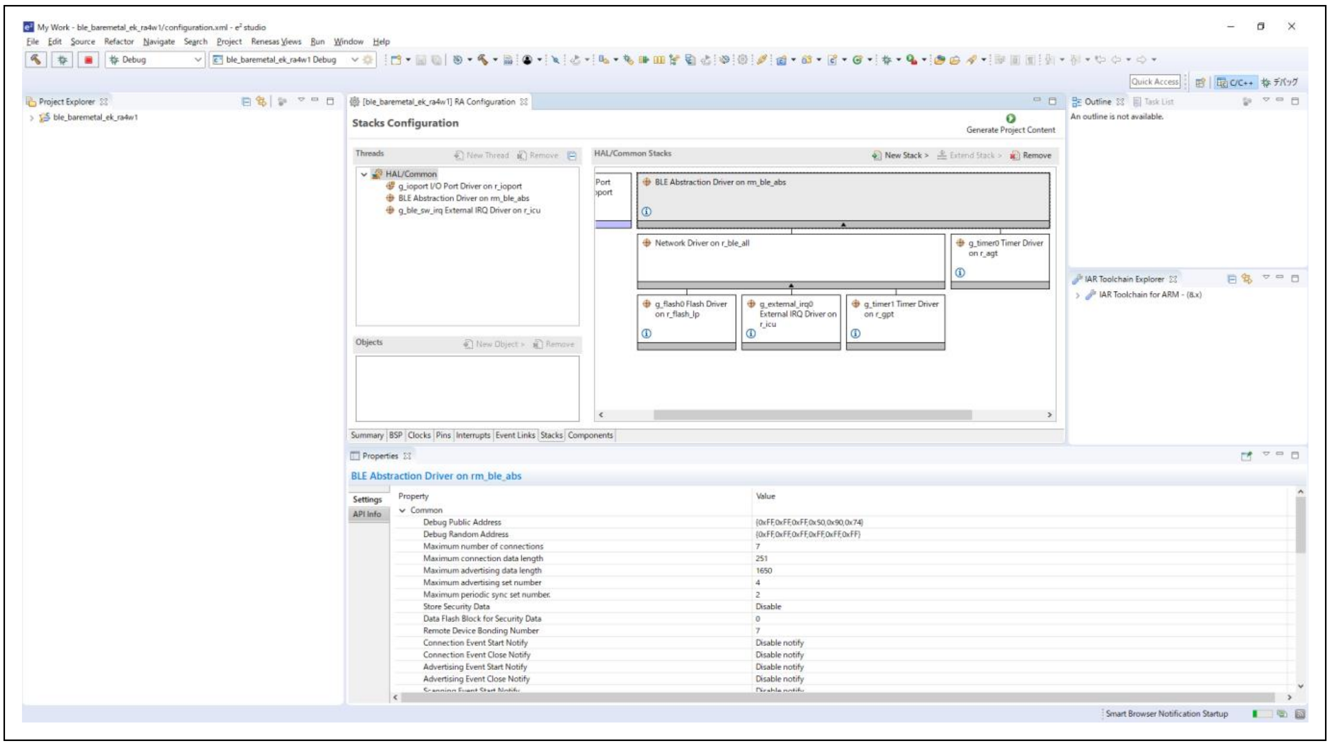
Note: The feature except mandatory feature is optional (vendor dependent). Therefore, some device (e.g. smart phone may not support such an optional feature.

2.2 How to add BLE Abstraction Driver to project

Refer to *BLE sample application (R01AN5402)* Chapter 3 and 4.

2.3 Configuration Options

BLE Abstraction Driver has some configuration options. These options can modify in properties of FSP configuration, as shown in Figure 3.


BLE Abstraction Driver on rm_ble_abs

Property	Value
Common	
Debug Public Address	{0xFF,0xFF,0xFF,0x50,0x90,0x74}
Debug Random Address	{0xFF,0xFF,0xFF,0xFF,0xFF,0xFF}
Maximum number of connections	7
Maximum connection data length	251
Maximum advertising data length	1650
Maximum advertising set number	4
Maximum periodic sync set number.	2
Store Security Data	Disable
Data Flash Block for Security Data	0
Remote Device Bonding Number	7
Connection Event Start Notify	Disable notify
Connection Event Close Notify	Disable notify
Advertising Event Start Notify	Disable notify
Advertising Event Close Notify	Disable notify
Scanning Event Start Notify	Disable notify

Figure 3. Common options

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

2.4 How to adjust configuration option

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

2.4.1 How to adjust RAM usage

This section describes how to adjust RAM usage by changing BLE Abstraction Driver configuration options. BLE Abstraction Driver (includes BLE Protocol Stack library, related peripherals, NOT include BLE application and profiles) consumes the ROM/RAM size shown in Table 6 according to extended / balance / compact configuration. Refer to *BLE sample application (R01AN5402) Chapter 1* about extended, balance and compact configuration.

Table 6. BLE Abstraction 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 7 shows relationship between RAM consumption and related configuration options.

Table 7. 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 8.
Maximum advertising set number*1	1 - 4 (4)	See Table 8.
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 8 shows relationship between RAM consumption and *Maximum advertising data length* configuration option and *Maximum advertising set number* configuration option.

Table 8. 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 9.

Table 9. BD address types

BD address type			Description
Public device address			Unique 48bit Bluetooth device 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, PartB, "1.3.2 Random Device Address".* Note: BLE Protocol Stack does not check address format.
	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 10. Refer to *BLE sample application (R01AN5402)* chapter 4 about details of these configurations.

Table 10. 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)
Code Flash (ROM) Device Data Block	255 (Code block 255 is used)

BLE Abstraction Driver adopts BD address according to the priority shown in Figure 4.

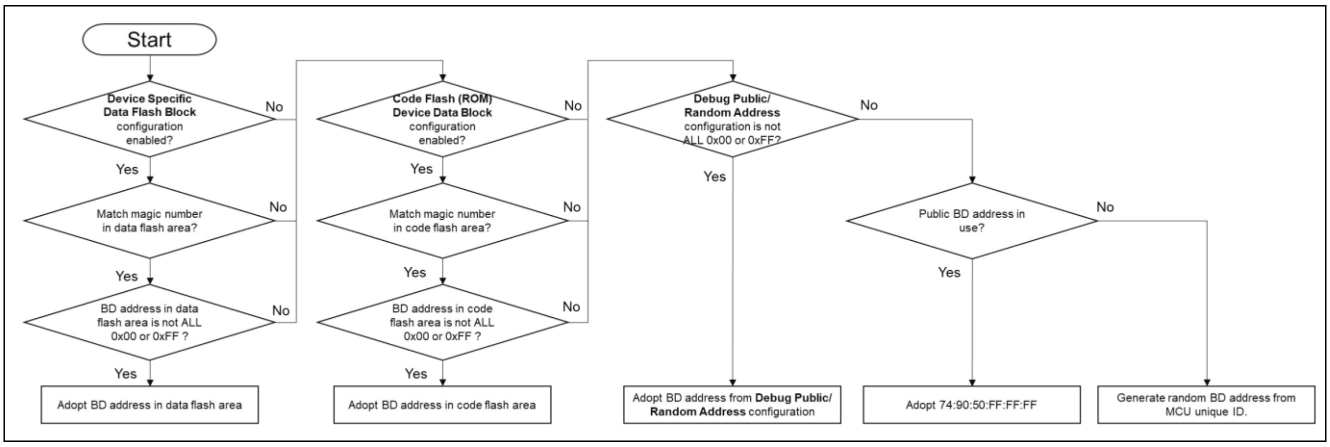

Start
Device Specific Data Flash Block configuration enabled?
No
Yes
Match magic number in data flash area?
No
Yes
BD address in data flash area is not ALL 0x00 or 0xFF ?
No
Yes
Adopt BD address in data flash area
Code Flash (ROM) Device Data Block configuration enabled?
No
Yes
Match magic number in code flash area?
No
Yes
BD address in code flash area is not ALL 0x00 or 0xFF ?
No
Yes
Adopt BD address in code flash area
Debug Public/ Random Address configuration is not ALL 0x00 or 0xFF?
No
Yes
Adopt BD address from Debug Public/ Random Address configuration
Public BD address in use?
No
Yes
Adopt 74:90:50:FF:FF:FF
Generate random BD address from MCU unique ID.

Figure 4. BD address adoption flow of BLE Abstraction Driver

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 *RA Flexible Software Package Documentation* 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 Renesas E2-Lite emulator and Renesas Flash Programmer (RFP) unique code function. Refer to *BLE sample application (R01AN5402)* chapter 4 about detail procedures.

3. Other method

When user wants to dynamically change BD address, *R_BLE_VS_SetBdAddr* API can be used. Refer to *RA Flexible Software Package Documentation* for details of the API.

2.4.3 How to use random address

Code 1 is a sample code for advertising with a random address.

```
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 1. Sample of using random address

2.4.4 How to configure for minimum current consumption

The configurations shown in Table 11 make the current consumption minimize.

Table 11. Configurations for minimum current consumption

Configuration options		Comments
FSP configuration / Clocks	Clock source: HOCO HOCO clock frequency: 32MHz	Note: Make non-used clocks disable or set minimum clock frequency to minimize current consumption.
	ICLK: 32MHz	
	PCLKA and PCLKD: 32MHz	
	FCLK: 32MHz	
	CLKOUT: Disable	
BLE Abstraction Driver 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 detail 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	Need to add *Low Power Modes Driver on r_lpm*. And 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” mean “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.

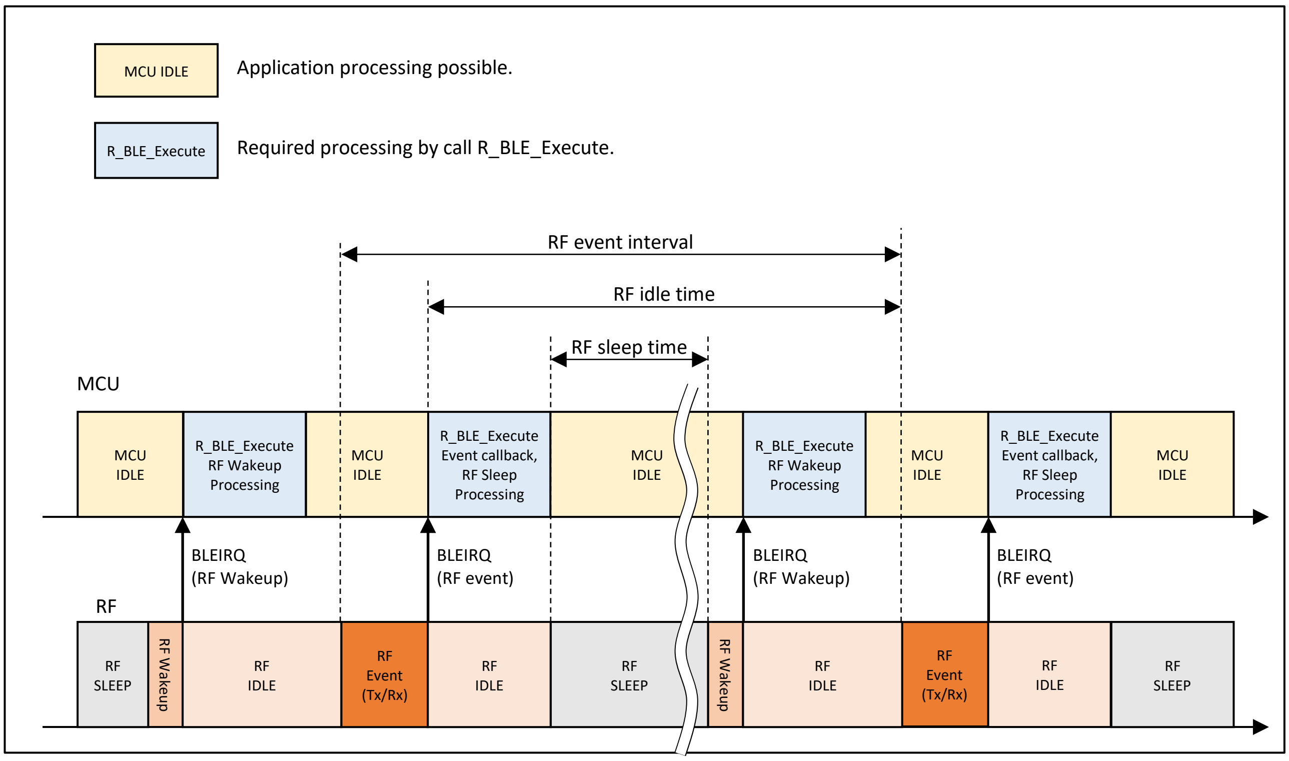


Figure 5. MCU/RF operation overview with RF sleep

While the MCU is idle, it is possible to transition the MCU to the low power consumption mode or execute the other application processing. 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 does not met. However, the MCU idle time can be used to user application process since it is not necessary to consider RF wake up process. Figure 6 shows MCU/RF operation without RF sleep.

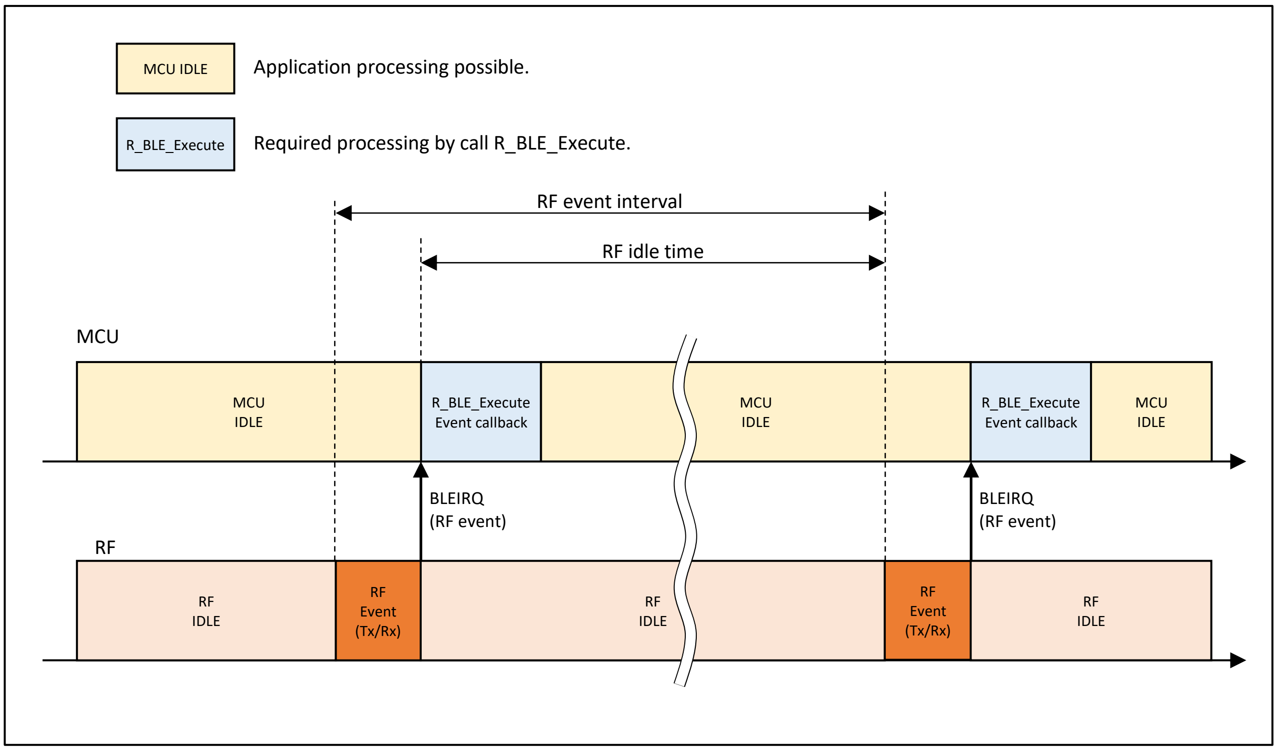


Figure 6. MCU/RF operation overview without RF sleep

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

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 the BLE function. The basic policy of the transition to Low power consumption state is as below.

- BLE application can use MCU Low Power Mode from completion RF event execution to start 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 processing for disabling interrupts, check the status of BLE task once after disabling interrupts, If BLE task state is not free, skip 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 2.

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

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

```
    /* 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())
        {
```

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)

```
            /* 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();
        }
………
}
```

Code 2. Transition to MCU Low power mode

(2) FreeRTOS environment

FreeRTOS kernel enter MCU low power mode in *vApplicationIdleHook*.

3. How to implement user code

QE for BLE generates:

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

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

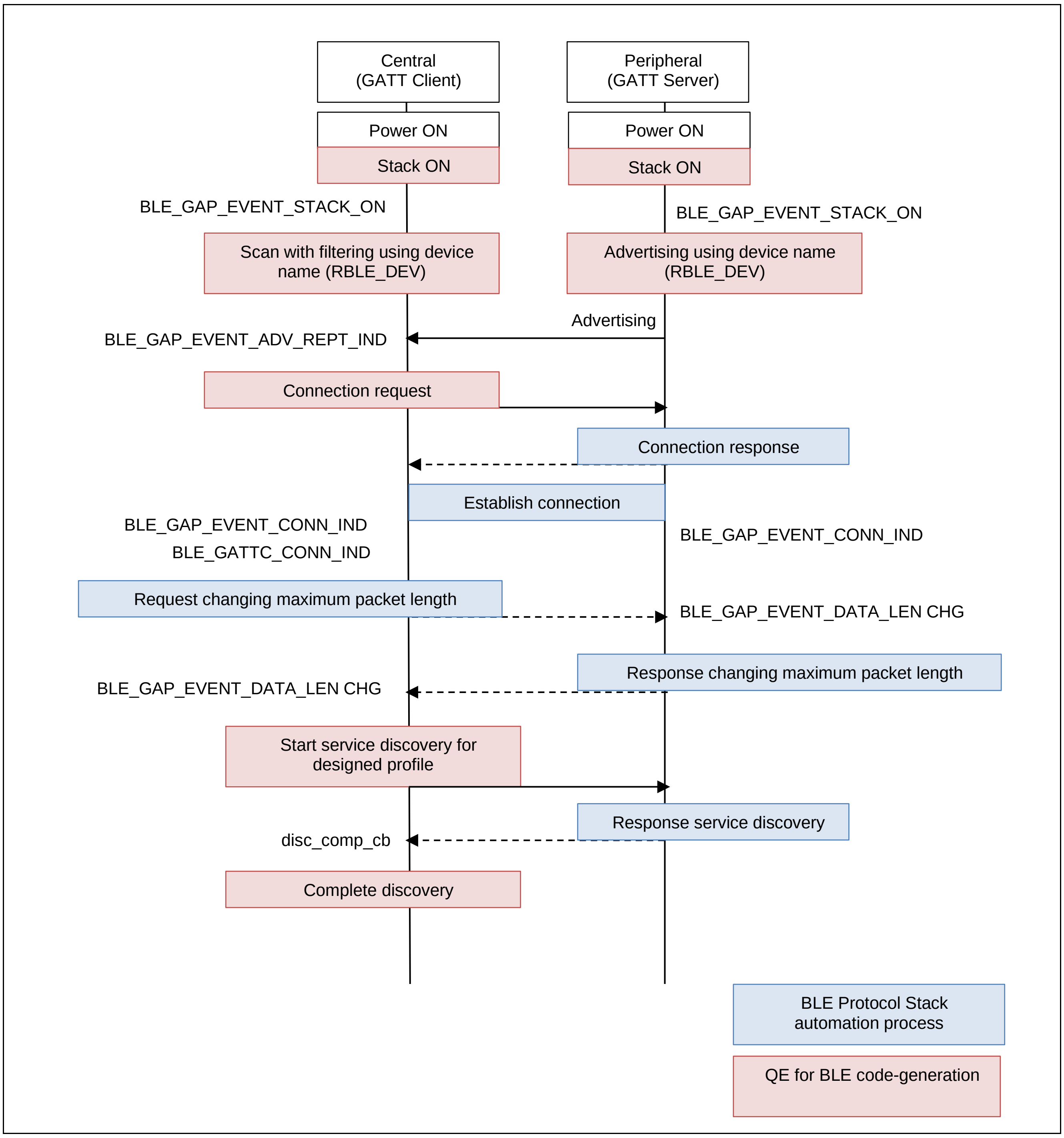

Central (GATT Client)
Peripheral (GATT Server)
Power ON
Stack ON
Power ON
Stack ON
BLE_GAP_EVENT_STACK_ON
BLE_GAP_EVENT_STACK_ON
Scan with filtering using device name (RBLE_DEV)
Advertising using device name (RBLE_DEV)
Advertising
BLE_GAP_EVENT_ADV_REPT_IND
Connection request
Connection response
Establish connection
BLE_GAP_EVENT_CONN_IND
BLE_GATTC_CONN_IND
BLE_GAP_EVENT_CONN_IND
Request changing maximum packet length
BLE_GAP_EVENT_DATA_LEN CHG
Response changing maximum packet length
BLE_GAP_EVENT_DATA_LEN CHG
Start service discovery for designed profile
Response service discovery
disc_comp_cb
Complete discovery
BLE Protocol Stack automation process
QE for BLE code-generation

Figure 7. Behavior of codes generated by QE for BLE

And User can use APIs shown in Table 12 when making BLE application.

Table 12. APIs for making BLE application

Functions	API/Macro name	Include header and 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	#include "r_ble_api.h" • R_BLE_GAP_XXX ➢ API result can be received as BLE_GAP_EVENT_XXX event once registering callback function by using RM_BLE_ABS_Open or R_BLE_GAP_Init API. • R_BLE_GATTS_XXX ➢ API result can be received as BLE_GATTS_EVENT_XXX event once registering callback function by using RM_BLE_ABS_Open or R_BLE_GATTS_RegisterCb API. • R_BLE_GATTC_XXX ➢ API result can be received as BLE_GATTC_EVENT_XXX event once registering callback function by using RM_BLE_ABS_Open or R_BLE_GATTC_RegisterCb API. • R_BLE_L2CAP_XXX ➢ API result can be received as BLE_L2CAP_EVENT_XXX event once registering callback function by using R_BLE_L2CAP_RegisterCfPsm API. No need to register callback function for R_BLE_XXX and R_BLE_GATT_GetMtu. Since result of these API can be received immediately. And R_BLE_XXX_Init, R_BLE_XXX_RegisterCb, R_BLE_GAP_SetPairingParams can also receive API result immediately.
Vendor Specific (VS)	R_BLE_VS_XXX	#include "r_ble_api.h" 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.
Abstraction API	RM_BLE_ABS_XXX	#include "rm_ble_abs.h" API result can be received as following events once registering callback function by using RM_BLE_ABS_Open. BLE_GAP_EVENT_XXX BLE_GATTS_EVENT_XXX BLE_GATTC_EVENT_XXX BLE_VS_EVENT_XXX event. event once registering callback function by using RM_BLE_ABS_Open API.

Functions	API/Macro name	Include header and Usage
Profile common	R_BLE_DISC_XXX R_BLE_SERVC_XXX R_BLE_SERVS_XXX	#include "discovery/r_ble_disc.h" #include "profile_cmn/r_ble_servc_if.h" #include "profile_cmn/r_ble_servs_if.h" These APIs are generated by QE for BLE. • R_BLE_DISC_XXX ➢ Once registering callback function by using R_BLE_DISC_Start, Service Discovery result can be received. • R_BLE_SERVC_XXX ➢ Once registering callback function using R_BLE_SERVC_GattcCb, API result can be received. • R_BLE_SERVS_XXX ➢ Once registering callback function using R_BLE_SERVS_GattsCb, API result can be received as event. • Function to receive VS event in SERVS ➢ It is necessary to passing the event data from callback function registered by R_BLE_VS_Init or RM_BLE_ABS_Init to R_BLE_SERVS_VsCb as it is.
Profile API	R_BLE_[service name]_XXX	#include "r_ble_[service name].h" Generated by QE for BLE. Once registering callback function using R_BLE_[service name]_Init, event can be received when receiving Write, Read, Indication, Notification from remote device.

3.1 Example of implementation

Refer to *BLE sample application (R01AN5402)* Chapter 3 and 4.

4. Advertising

Bluetooth LE device broadcasts data to nearby scanning devices by advertising. This chapter describes how to use 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.

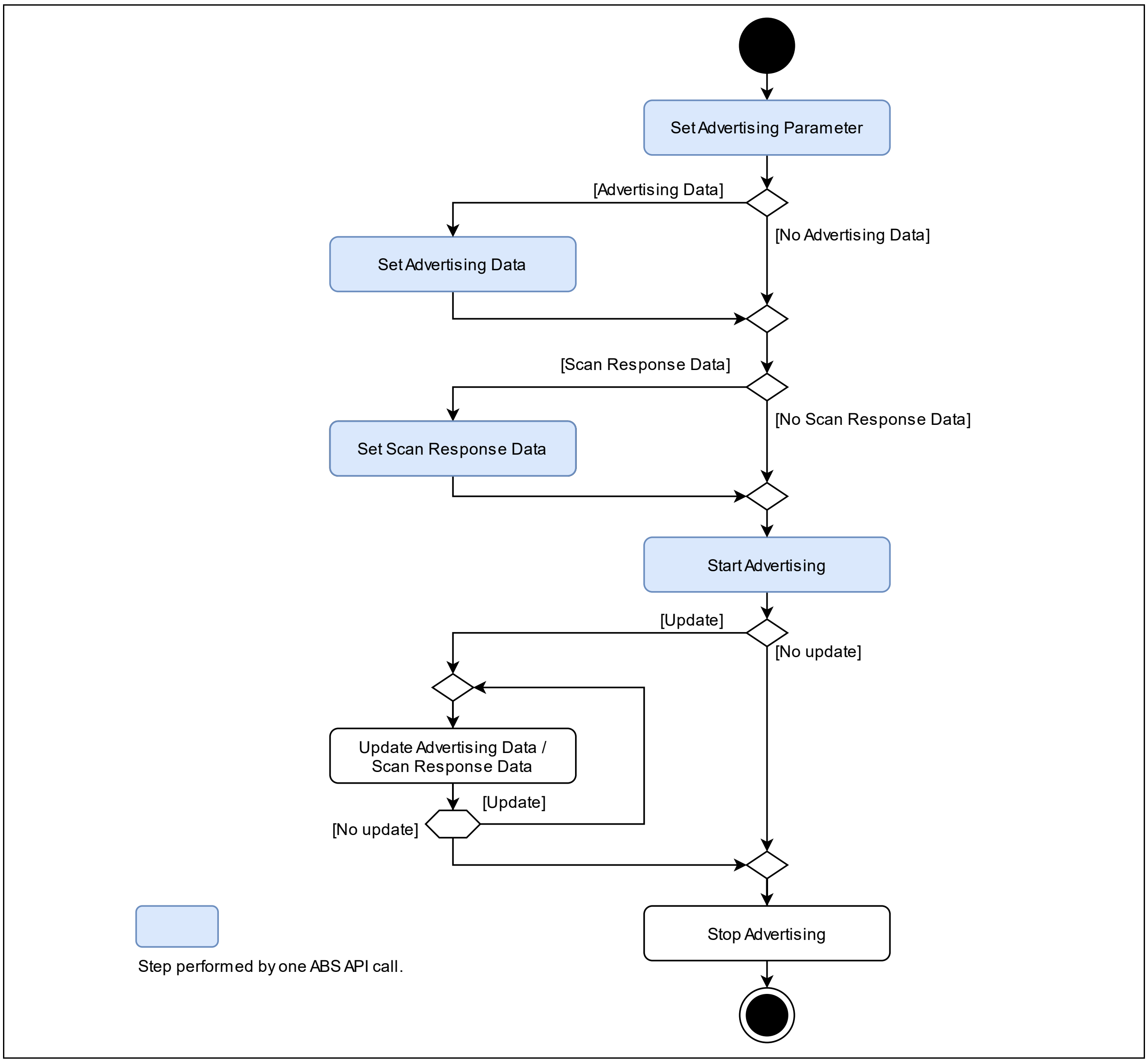


Figure 8. Advertising procedure

User can use following categories of API to perform above procedure.

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

- **GAP API (*R_BLE_GAP_XXX* API)**
  - User uses advertising feature by combining several APIs. However, detailed parameter settings are possible.

4.1 Advertising with abstraction API

When user uses abstraction API, the procedure from setting advertising parameters to starting advertising is performed by single abstraction API call. This kind of abstraction APIs are defined in Table 13. Refer to *RA Flexible Software Package Documentation* about usage of these APIs. And samples of typical use cases are shown in section 4.5.

Table 13. 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	245
			AUX_ADV_IND		
		Connectable Directed	ADV_EXT_IND		239
			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] option
			AUX_ADV_IND		
			AUX_CHAIN_IND		
	Extended	Non-Connectable and Non-Scannable Directed	ADV_EXT_IND		*Maximum advertising data length*[*1] option
			AUX_ADV_IND		
			AUX_CHAIN_IND		
RM_BLE_ABS_StartPeriodic Advertising	Extended	Periodic	ADV_EXT_IND	3	*Maximum advertising data length*[*1] option
			AUX_ADV_IND		
			AUX_SYNC_IND		
			AUX_CHAIN_IND		

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

4.1.1 White list

White list 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 white list by calling *R_BLE_GAP_ConfWhiteList* API.
2. Set value listed in Table 17 to use white list 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 IRK for using privacy feature according to section 8.4.1 and set value of Table 14 to:

- *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.
- *ble_abs_non_connectable_advertising_parameter _t* structure when use *RM_BLE_ABS_StartNonConnectableAdvertising* API.

Table 14. Parameters used for the privacy feature

Field	Value	Description
own_bluetooth_address_type	BLE_GAP_ADDR_RPA_ID_PUBLIC(0x02)	Specify the value if the Identity Address registered by *R_BLE_GAP_SetLocIdInfo* API is public address.
	BLE_GAP_ADDR_RPA_ID_RANDOM(0x03)	Specify the value if the Identity Address registered by *R_BLE_GAP_SetLocIdInfo* API is random address.
p_peer_address	Specify the remote device identity address registered by *R_BLE_GAP_ConfRslvList* API.	---

4.2 Advertising with GAP API

When 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.

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 *RA Flexible Software Package Documentation* about detail of the structure. 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 15.

Table 15. Correspondence between Advertising type and adv_prop_type field

Advertising type	Corresponding advertising PDU	The adv_prop_type field value	Legacy or extended	Max size of advertising data (byte)
Connectable and Scannable Undirected [*4]	ADV_IND	BLE_GAP_LEGACY_PROP_ADV_IND	legacy	31
Connectable Undirected	ADV_EXT_IND	BLE_GAP_EXT_PROP_ADV_CONN_NOSCAN_UNDIRECT	extended	245[*1]
	AUX_ADV_IND			
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	BLE_GAP_EXT_PROP_ADV_CONN_NOSCAN_DIRECT or BLE_GAP_EXT_PROP_ADV_CONN_NOSCAN_HDC_DIRECT	extended	239[*1]
	AUX_ADV_IND			
Non-Connectable and Non-Scannable Undirected	ADV_NONCONN_IND	BLE_GAP_LEGACY_PROP_ADV_NONCONN_IND	legacy	31
	ADV_EXT_IND	BLE_GAP_EXT_PROP_ADV_NOCONN_NOSCAN_UNDIRECT	extended	Maximum advertising data length[*5]
	AUX_ADV_IND			
	AUX_CHAIN_IND[*2]			
Non-Connectable and Non-Scannable Directed	ADV_EXT_IND	BLE_GAP_EXT_PROP_ADV_NOCONN_NOSCAN_DIRECT or BLE_GAP_EXT_PROP_ADV_NOCONN_NOSCAN_HDC_DIRECT	extended	Maximum advertising data length[*5]
	AUX_ADV_IND			
	AUX_CHAIN_IND[*3]			
Scannable Undirected [*4]	ADV_SCAN_IND	BLE_GAP_LEGACY_PROP_ADV_SCAN_IND	legacy	31
	ADV_EXT_IND	BLE_GAP_EXT_PROP_ADV_NOCONN_SCAN_UNDIRECT	extended	0
	AUX_ADV_IND			
Scannable Directed [*4]	ADV_EXT_IND	BLE_GAP_EXT_PROP_ADV_NOCONN_SCAN_DIRECT or BLE_GAP_EXT_PROP_ADV_NOCONN_SCAN_HDC_DIRECT	extended	0
	AUX_ADV_IND			

[*1]: Max size of advertising data is 1 byte less that the value listed in the table when *BLE_GAP_EXT_PROP_ADV_INCLUDE_TX_POWER* (0x0040) set to *adv_prop_type* field in st*_ble_gap_adv_param_t* structure.

[*2]: If the Advertising Data is 245 bytes or less, AUX_CHAIN ID is not used.

[*3]: If the Advertising Data is 239 bytes or less, AUX_CHAIN ID is not used.

[*4]: Figure 10 shows about scan response PDU and data length of the PDU.

[*5]: Configure in properties of *BLE Abstraction Driver on rm_ble_abs*. Refer to *BLE sample application (R01AN5402)* Chapter 4.

Advertising PDUs are sent as shown in Figure 9 when the advertising type is extended and non-scannable. The *advDelay* is a random delay from 0 to 10ms.

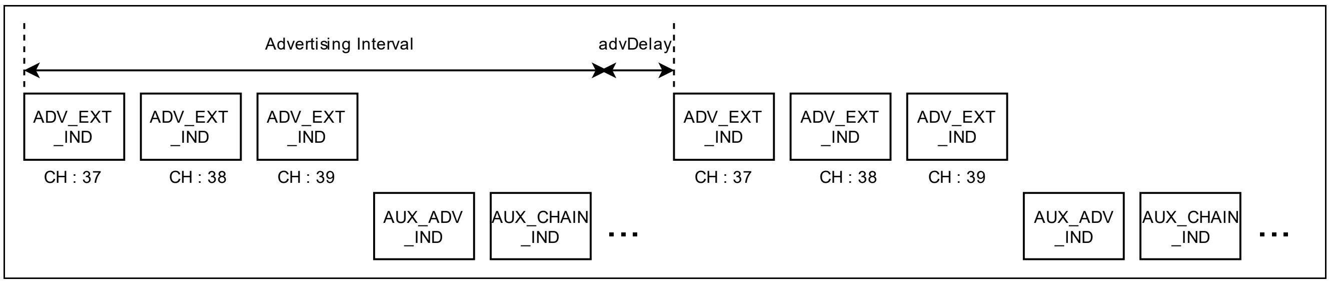


Figure 9. Extended Advertising PDU

Advertising PDUs are sent as shown in Figure 10 when the advertising type is extended and scannable.

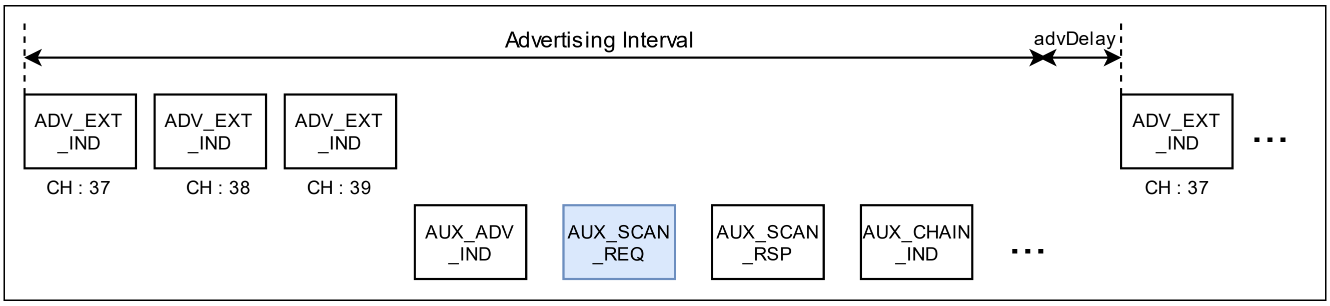


Figure 10. Scannable Advertising PDU

Remote device sent scan request PDU (*AUX_SCAN_REQ*) to advertising device, scan response PDUs (*AUX_SCAN_RSP* and *AUX_CHAIN_IND*) shown in Table 16 are sent according to configuration of *adv_prop_type* field in *st_ble_gap_adv_param_t* structure.

Table 16. 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	*BLE_CFG_RF_ADV_DATA_MAX**2 *3

*1: If the Scan Response Data is 253 bytes or less, the AUX_CHAIN_IND is not used.

*2: Max size of advertising data is 1 byte less that the value listed in the table when *BLE_GAP_EXT_PROP_ADV_INCLUDE_TX_POWER* (0x0040) set to *adv_prop_type* field in *st_ble_gap_adv_param_t* structure.

*3: Configure in properties of *BLE Abstraction Driver on rm_ble_abs*. Refer to *BLE sample application (R01AN5402)* Chapter 4.

4.2.1.2 White list

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

1. Register a known device BD address to the white list by calling *R_BLE_GAP_ConfWhiteList* API.
2. Set to use white list feature for *filter_policy* field in *st_ble_gap_adv_param_t* structure as shown in Table 17.

Table 17. The value set to the filter_policy field

Value set to the filter_policy field	Description
BLE_GAP_SCAN_ALLOW_ ADV_ALL(0x00)	Process scan and connection requests from all devices.
BLE_GAP_ADV_ALLOW_ SCAN_WLST_CONN_ANY(0x01)	Process connection requests from all devices and scan requests from only devices that are in the White List.
BLE_GAP_ADV_ALLOW_ SCAN_ANY_CONN_WLST(0x02)	Process scan requests from all devices and connection requests from only devices that are in the White List.
BLE_GAP_ADV_ALLOW_ SCAN_WLST_CONN_WLST(0x03)	Process scan and connection requests from only devices in the White List.

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 advertising packet. To use the privacy function, it is necessary to configure the field shown in Table 18 in *st_ble_gap_adv_param_t* structure and perform procedure described in section 8.4.1.

Table 18. The parameters used for the privacy feature

Field	Value	Description
o_addr_type	BLE_GAP_ADDR_RPA_ID_PUBLIC(0x02)	Specify the value if the Identity Address registered by *R_BLE_GAP_SetLocIdInfo* is public address.
	BLE_GAP_ADDR_RPA_ID_RANDOM(0x03)	Specify the value if the Identity Address registered by *R_BLE_GAP_SetLocIdInfo* is static address.
p_addr_type	Specify the remote device identity address registered by R_BLE_GAP_ConfRslvList().	—
p_addr		

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 Abstraction Driver on rm_ble_abs*. 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 13 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 advertising handle which want to stop advertising with argument *adv_hdl*. And in case of connectable advertising, the advertising will stop automatically when established connection with a remote device.

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 describe 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.

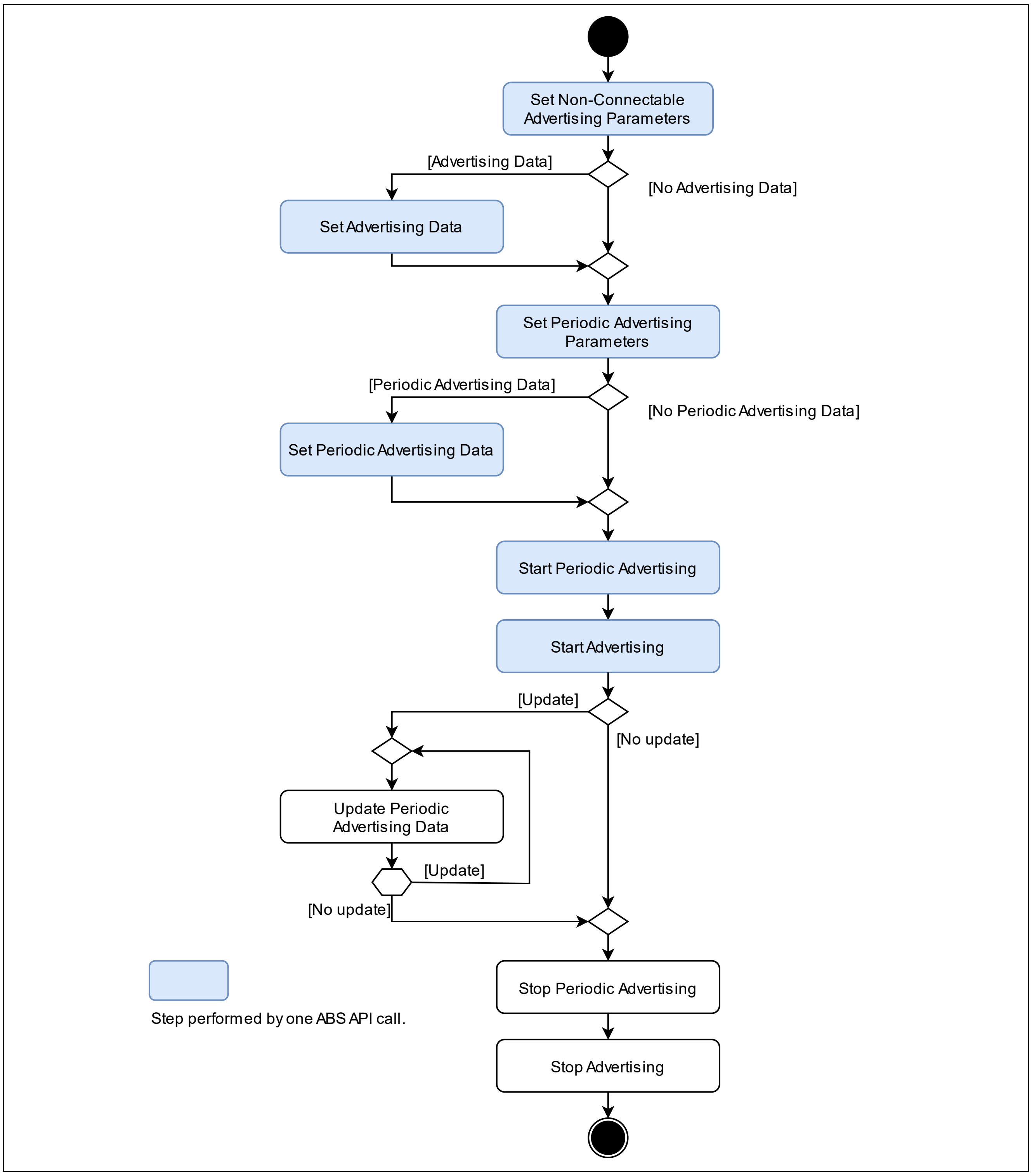

Set Non-Connectable Advertising Parameters
[Advertising Data]
[No Advertising Data]
Set Advertising Data
Set Periodic Advertising Parameters
[Periodic Advertising Data]
[No Periodic Advertising Data]
Set Periodic Advertising Data
Start Periodic Advertising
Start Advertising
[Update]
[No update]
Update Periodic Advertising Data
[Update]
[No update]
Stop Periodic Advertising
Stop Advertising
Step performed by one ABS API call.

Figure 11. Periodic Advertising procedure

User can use only GAP API for broadcasting periodic advertising PDUs.

4.3.1 Non-Connectable Advertising Parameter

To start periodic advertising, it is necessary to configure advertising parameter as 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 *RA Flexible Software Package Documentation* about detail of the structure.

4.3.3 Periodic Advertising Data

For details about setting 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 19 and broadcast timing is shown in Figure 12.

Table 19. 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]
	AUX_CHAIN_IND[*1]		

*1: If the periodic advertising Data is 253 bytes or less, AUX_CHAIN ID is not used.
*2: Max size of advertising data is 1 byte less that the value listed in the table when *BLE_GAP_EXT_PROP_ADV_INCLUDE_TX_POWER* (0x0040) set to *adv_prop_type* field in s*t_ble_gap_adv_param_t* structure.
*3: Configure in properties of *BLE Abstraction Driver on rm_ble_abs*. Refer to *BLE sample application (R01AN5402)* Chapter 4.

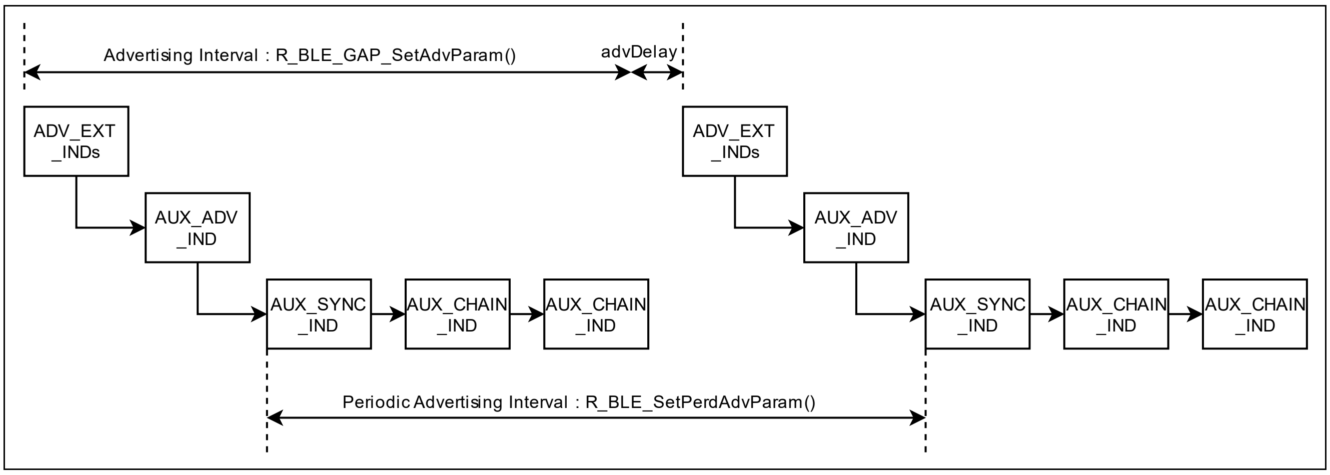


Figure 12. Periodic Advertising PDUs

An example of starting Periodic Advertising is shown in Code 3.

```
/* 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 3. 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 following data to inform auxiliary information to 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. And 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 20.

Table 20. 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 the advertising data setting in GAP callback function.
3. Set scan response data by calling *R_BLE_GAP_SetAdvSresData* API.
4. Confirm the completion of the scan response data setting in GAP callback function.

4.4.1 Data format

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

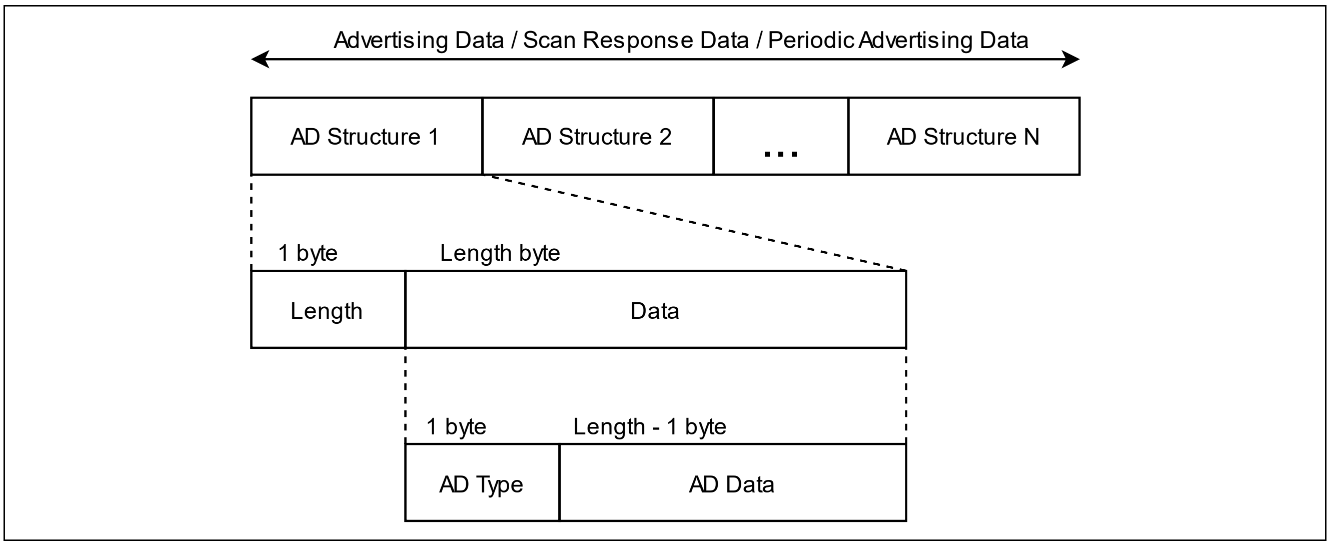


Figure 13. Advertising Data / Scan Response Data / Periodic Advertising Data format

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, and the 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 21 shows the AD Type that is often used.

Table 21. AD Type and AD Data

Data Type		AD Type	Description
Flags		0x01	Used for Connectable advertising. The Flags value used for Bluetooth LE is as follows. Octet | Bit | Description 0 | 0 | LE Limited Discoverable Mode 0 | 1 | LE General Discoverable Mode 0 | 2 | BR/EDR Not Supported. A scanner is available Discoverable Mode for filtering by the mode. If adding Discoverable Mode, select Limited or General.
Service UUID	Incomplete List of 16-bit Service UUIDs	0x02	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.
	Complete List of 16-bit Service UUIDs	0x03	
	Incomplete List of 32-bit Service UUIDs	0x04	
	Complete List of 32-bit Service UUIDs	0x05	
	Incomplete List of 128-bit Service UUIDs	0x06	
	Complete List of 128-bit Service UUIDs	0x07	
Local Name	Shortened Local Name	0x08	Strings that show the head of the device name to the middle.
	Complete Local Name	0x09	Complete Device Name.
Manufacturer Specific Data		0xFF	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/)

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 4.

```
/* 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 4. 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 is running if the conditions shown in Table 22 are met.

Table 22. 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 5.

```
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 5. Parameters for updating Advertising Data / Scan Response Data

If user 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 23 are met.

Table 23. 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 6 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 6. Parameters for updating Periodic Advertising Data

If user 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 15, extended advertising can be set Advertising Data or Scan Response Data up to the *Maximum advertising data length* configuration value. And the size of buffer for Advertising Data and Scan Response Data in the BLE Protocol Stack is 4250 bytes. Therefore, the amount size of Advertising Data and Scan Response Data in all sets must be 4250 bytes or less.

As shown in Table 19, periodic advertising can be set periodic advertising data up to the *Maximum advertising data length configuration* value. And the size of buffer for Periodic Advertising Data in the BLE Protocol Stack is 4306 bytes. Therefore, the amount size of periodic advertising data in all sets must be 4306 bytes or less.

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

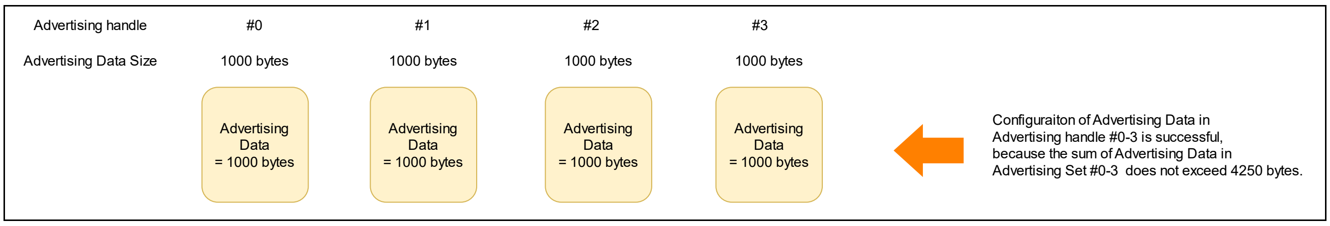

Advertising handle
#0
#1
#2
#3
Advertising Data Size
1000 bytes
1000 bytes
1000 bytes
1000 bytes
Advertising Data = 1000 bytes
Advertising Data = 1000 bytes
Advertising Data = 1000 bytes
Advertising Data = 1000 bytes
Configuraiton of Advertising Data in Advertising handle #0-3 is successful, because the sum of Advertising Data in Advertising Set #0-3 does not exceed 4250 bytes.

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

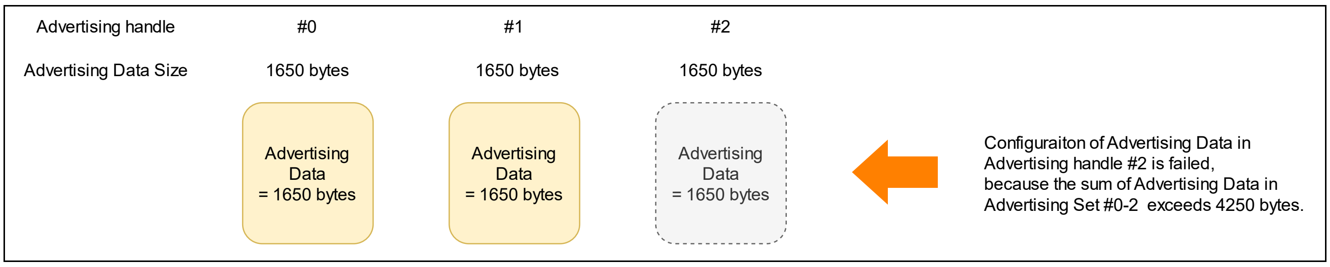

Advertising handle
#0
#1
#2
Advertising Data Size
1650 bytes
1650 bytes
1650 bytes
Advertising Data = 1650 bytes
Advertising Data = 1650 bytes
Advertising Data = 1650 bytes
Configuraiton of Advertising Data in Advertising handle #2 is failed, because the sum of Advertising Data in Advertising Set #0-2 exceeds 4250 bytes.

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 with Smart Phone is shown in Code 7.

```
/* 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 7. Sample of advertising for connecting with Smart Phone

When starting advertising, user application will be notified of the *BLE_GAP_EVENT_ADV_ON* event. Smart Phone 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 8.

```
/* 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 8. Sample of using RM_BLE_ABS_StartNonConnectableAdvertising

When starting advertising, user application will be notified of 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 scan feature by using related APIs.

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

- **Abstraction API (*RM_BLE_ABS_XXX* API)**
  - ➢ User can use scan feature with a single API call. However, detailed parameter settings are not possible.
- **GAP API (*R_BLE_GAP_XXX* API)**
  - ➢ User uses scan feature by combining several APIs. However, detailed parameter settings are possible.

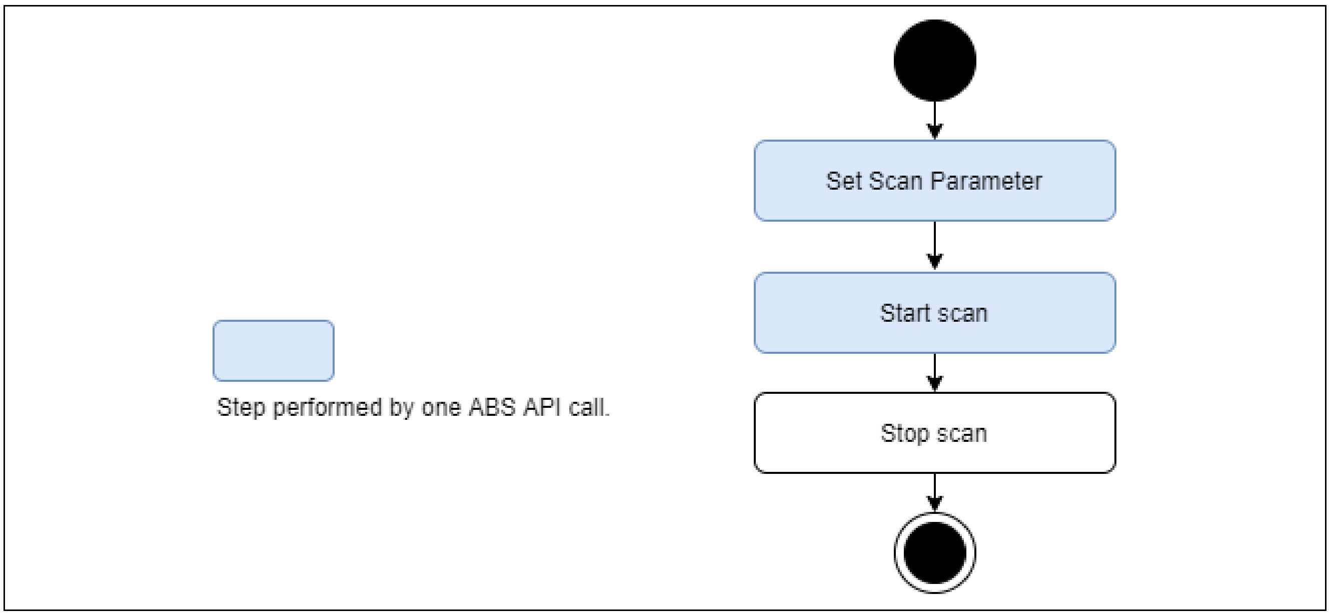


Figure 16. Scan procedure

5.1 Scan with abstraction API

When user uses 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 *RA Flexible Software Package Documentation* about usage of the API. And sample code of acquiring information obtained by scan is shown in section 5.2.4.

5.1.1 Scan filtering

Refer to section 5.3.

5.1.2 Privacy

Scan abstraction API cannot use the privacy feature. It is necessary to use GAP API when user want to use the privacy feature.

5.2 Scan with GAP API

When user uses 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 *st_ble_gap_scan_param_t* and *st_ble_gap_scan_on_t* structures and call *R_BLE_GAP_StartScan* API with these structures as arguments. Refer to *RA Flexible Software Package Documentation* about detail 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

Figure 17 shows relationship of these parameters.

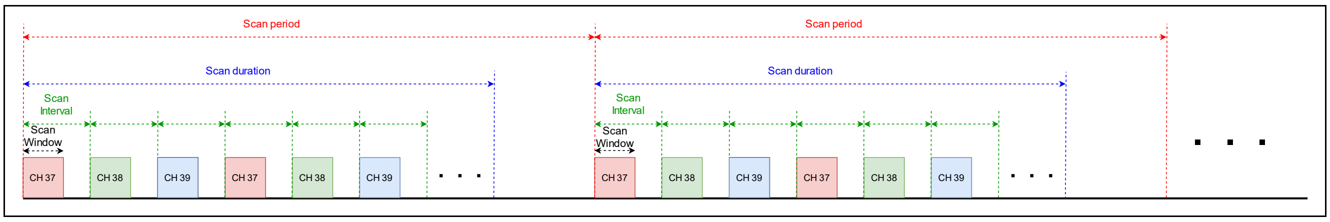


Figure 17. The relationship of scan interval, window, duration, period

These structures also include *fast_xxx* and *slow_xxx* fields. These fields specify 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.

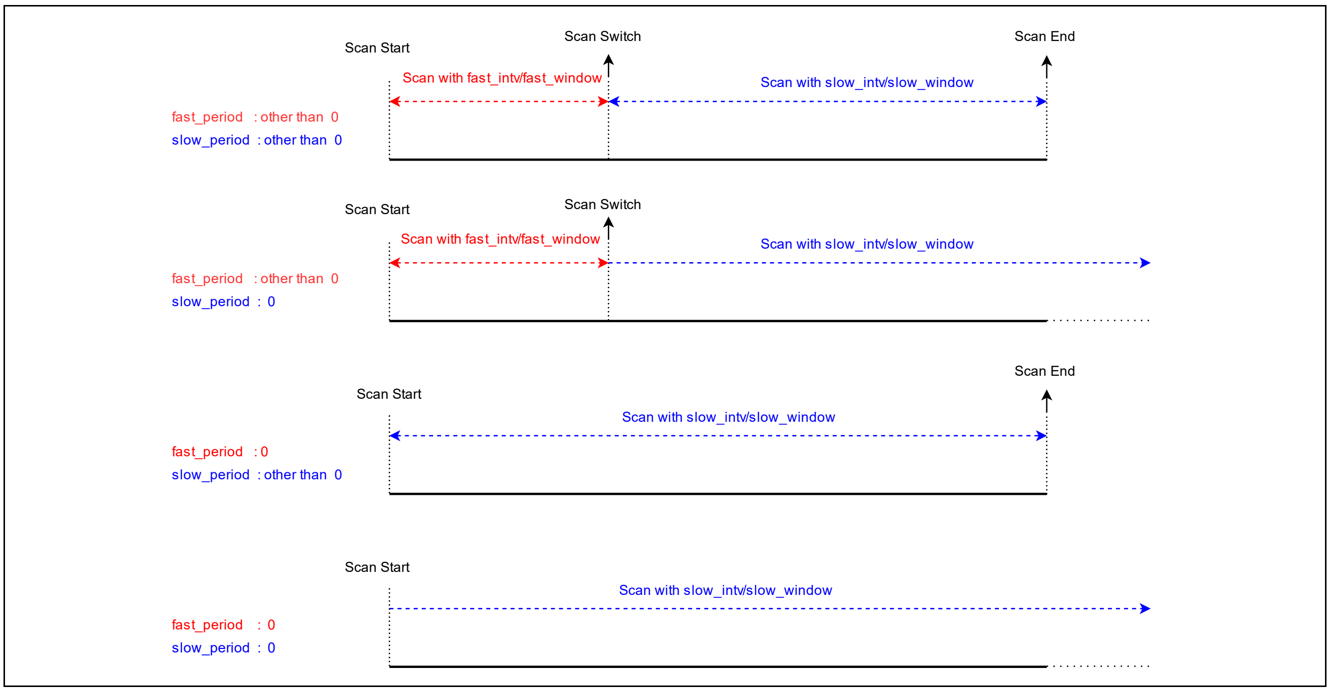

Scan Start
Scan Switch
Scan End
Scan with fast_intv/fast_window
Scan with slow_intv/slow_window
fast_period : other than 0
slow_period : other than 0
Scan Start
Scan Switch
Scan with fast_intv/fast_window
Scan with slow_intv/slow_window
fast_period : other than 0
slow_period : 0
Scan End
Scan Start
Scan with slow_intv/slow_window
fast_period : 0
slow_period : other than 0
Scan Start
Scan with slow_intv/slow_window
fast_period : 0
slow_period : 0

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

Table 24 shows the event regarding the fast scan and slow scan.

Table 24. 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

5.2.1.1 White list

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 8.4.1 and set value shown in Table 25 to *o_addr_type* field in *st_ble_gap_scan_param_t* structure.

Table 25. The parameters used for the privacy feature

Field	Value	Description
o_addr_type	BLE_GAP_ADDR_RPA_ID_PUBLIC(0x02)	Specify the value if the Identity Address registered by R_BLE_GAP_SetLocIdInfo is public address.
	BLE_GAP_ADDR_RPA_ID_RANDOM(0x03)	Specify the value if the Identity Address registered by R_BLE_GAP_SetLocIdInfo is static address.

5.2.2 Start scan

When starting scan, call *R_BLE_GAP_StartScan* API.

5.2.3 Stop scan

When stopping scan, call *R_BLE_GAP_StopScan* API. In addition to the API call, the conditions for stopping 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 user application that an advertising packet is received from another device by *BLE_GAP_EVENT_ADV_REPT_IND* event. Received advertising packet is stored in a *st_ble_gap_adv_rept_evt_t* structure. Refer to *RA Flexible Software Package Documentation* about detail of the structure. An example of displaying the RSSI of received advertising packet is shown in Code 9.

```
/* 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 9. 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.

- White list
- 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 White list

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

1. Register a known device BD address to the white list by calling *R_BLE_GAP_ConfWhiteList* API.
2. Set to use white list feature for:

- Set *BLE_GAP_SCAN_ALLOW_ADV_WLST* (0x01) to *device_scan_filter_policy* field in *ble_abs_scan_parameter_t* structure when performing scan with abstraction API.
- 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.

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 packet 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 26 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 26. 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 the data included in 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 10.

```
/* 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 10. 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 19 shows the procedure that a scanner establishes a Periodic Advertising Sync in application. The following sections describe the details of Periodic Advertising Sync procedure shown in Figure 19.

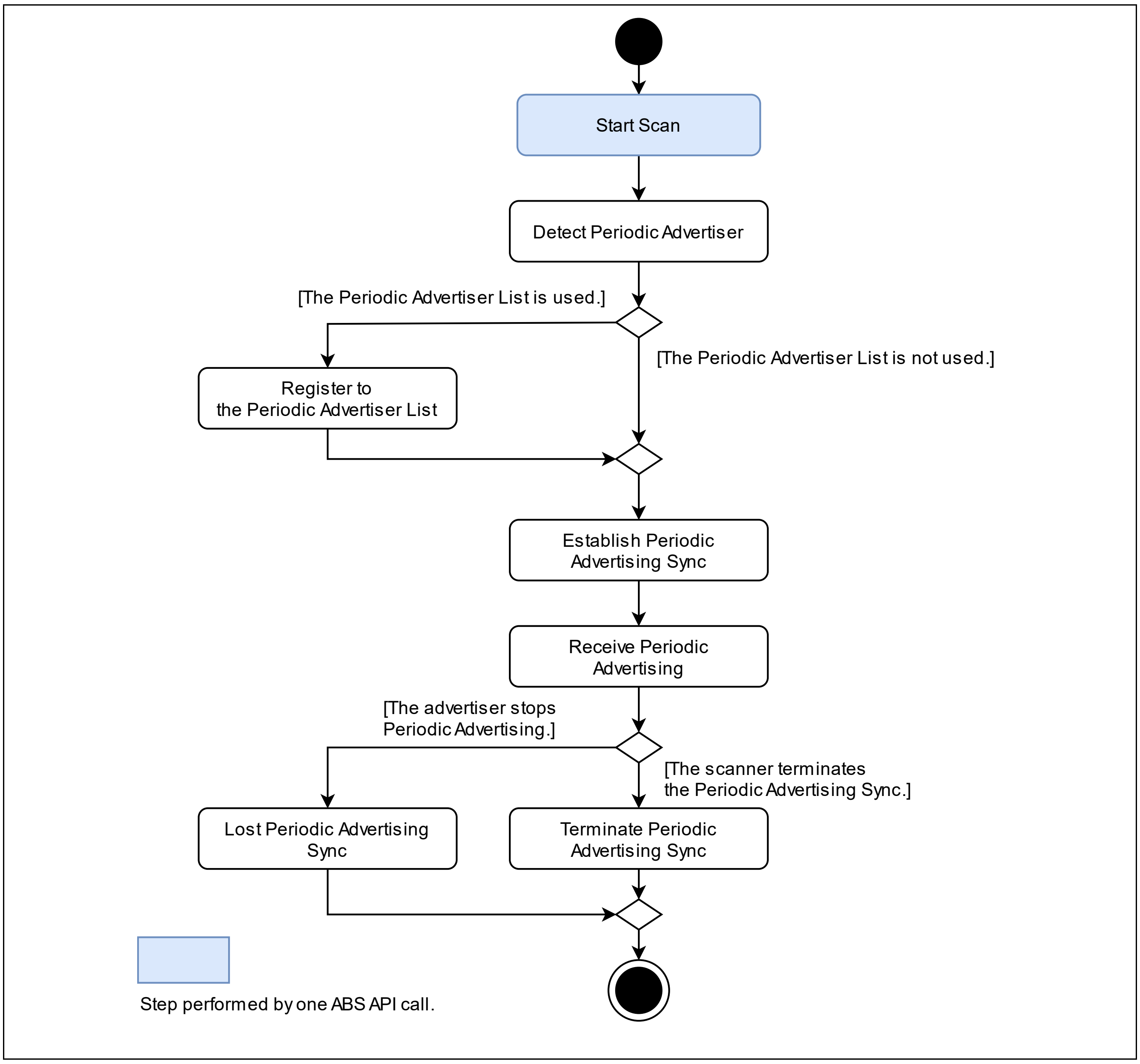

Start Scan
Detect Periodic Advertiser
[The Periodic Advertiser List is used.]
[The Periodic Advertiser List is not used.]
Register to the Periodic Advertiser List
Establish Periodic Advertising Sync
Receive Periodic Advertising
[The advertiser stops Periodic Advertising.]
[The scanner terminates the Periodic Advertising Sync.]
Lost Periodic Advertising Sync
Terminate Periodic Advertising Sync
Step performed by one ABS API call.

Figure 19. Periodic advertising sync procedure

User can use only GAP API for periodic advertising synchronization.

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 Protocol Stack 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 sync with the advertiser if *perd_adv_intv* field of *st_ble_gap_adv_rept_evt_t* structure includes in a received advertising packet is not zero. And 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.

5.4.3 Establish periodic advertising sync

Call *R_BLE_GAP_CreateSync* API to establish a Periodic Advertising Sync. User 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. 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 is notified of the BLE *BLE_GAP_EVENT_SYNC_EST* event, indicating that the result is *BLE_ERR_NOT_YET_READY*(0x0012). An example of from starting scan to establishing a Periodic Advertising Sync is shown in Code 11.

```
/** 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 11. 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 11 contains an example of the API usage.

5.4.5 Receive periodic advertising PDUs

After the periodic advertising sync 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. *RA Flexible Software Package Documentation* about detail of the structure.

5.4.6 Lost periodic advertising sync

When the advertiser stops Periodic Advertising, user application is notified by the *BLE_GAP_EVENT_SYNC_LOST* event that the Periodic Advertising Sync is loss.

5.4.7 Terminate periodic advertising sync

When the scanner terminates the Periodic Advertising Sync, call *BLE_GAP_TerminateSync* API. When the Periodic Advertising Sync has been terminated, user application is notified of 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 scan feature by using related APIs. User can use following categories of API to perform above procedure.

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

- **GAP API (*R_BLE_GAP_XXX* API)**
  - User uses connection feature by combining several APIs. However, detailed parameter settings are possible.

6.1 Requesting connection with abstraction API

When user uses abstraction API, call *RM_BLE_ABS_Create_Connection* API for requesting connection with advertiser. Refer to *RA Flexible Software Package Documentation* about usage of the API.

6.1.1 White list filtering

Refer to section 6.4.

6.1.2 Privacy

Abstraction API cannot use the privacy feature. It is necessary to use GAP API when user want to use the privacy feature.

6.2 Requesting connection with GAP API

When user uses GAP API, call *R_BLE_GAP_CreateConn* API for requesting connection with advertiser. Refer to *RA Flexible Software Package Documentation* about usage of the API.

6.3 Cancelling Connection Request

A connection request cannot be sent until the connection is established by previous connection request or the connection request is cancelled. After sending a connection request, if 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 use in any case where 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 White list filtering

When user want to reconnect with a known device, it is possible to use white list feature by applying following procedures.

1. Register the BD address of the remote device to white list by calling *R_BLE_GAP_ConfWhiteList* API.
2. Set to use white list feature for:
   - Set *BLE_GAP_INIT_FILT_USE_WLST*(0x01) to *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 White List is shown in Code 12.

```
/* 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 12. Connection Request using the White List

6.5 Privacy

Privacy feature can use after preparing IRK for using privacy feature according to section 8.4.1 and set value shown in Table 27 to *st_ble_gap_create_conn_param_t* structure.

Table 27. The parameters used for the privacy feature

Field	Value	Description
own_addr_type	BLE_GAP_ADDR_RPA_ID_PUBLIC(0x02)	Specify the value if the Identity Address registered by R_BLE_GAP_SetLocIdInfo is public address.
	BLE_GAP_ADDR_RPA_ID_RANDOM(0x03)	Specify the value if the Identity Address registered by R_BLE_GAP_SetLocIdInfo is static address.
remote_bd_addr_type	Specify the remote device address registered by R_BLE_GAP_ConfRslvList.	—

6.6 Multiple connection

This section describes how to connect with multiple devices at the same time. With the BLE Protocol Stack, 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 20. This is a one of typical case when collecting data from multiple sensors which perform GAP peripheral.

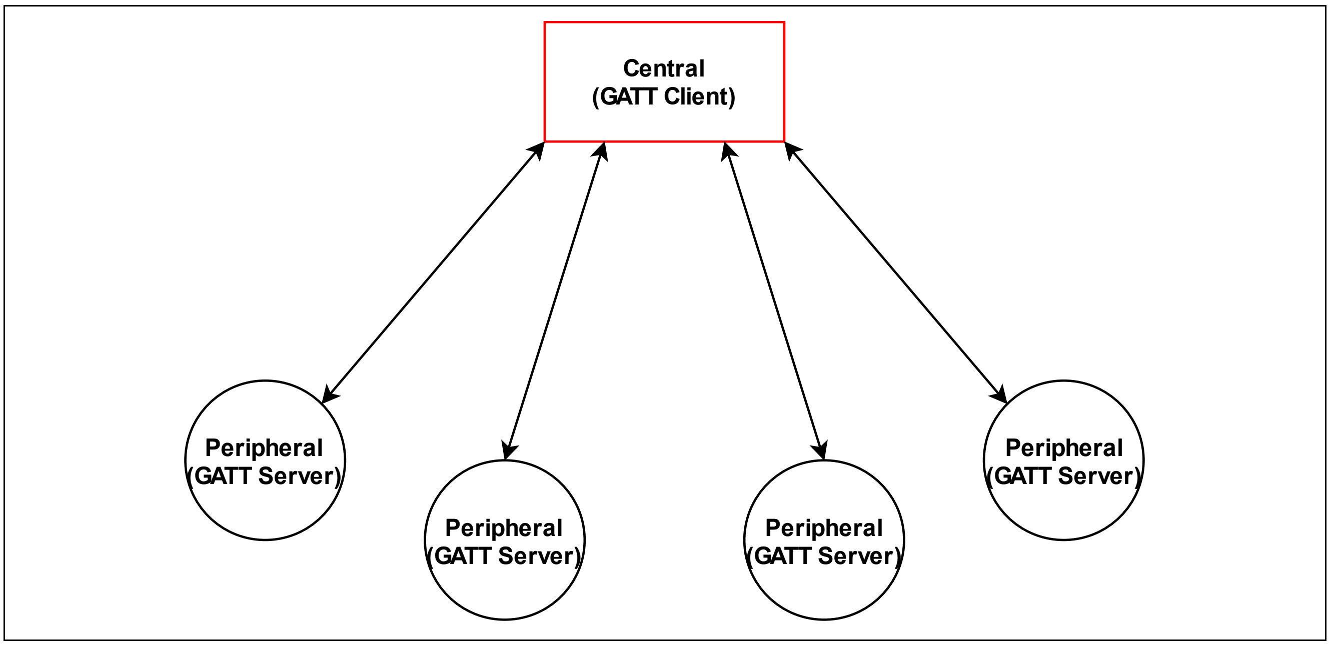


Figure 20. Connection with multiple peripheral devices

Sequence chart of implementation example is shown in Figure 21.

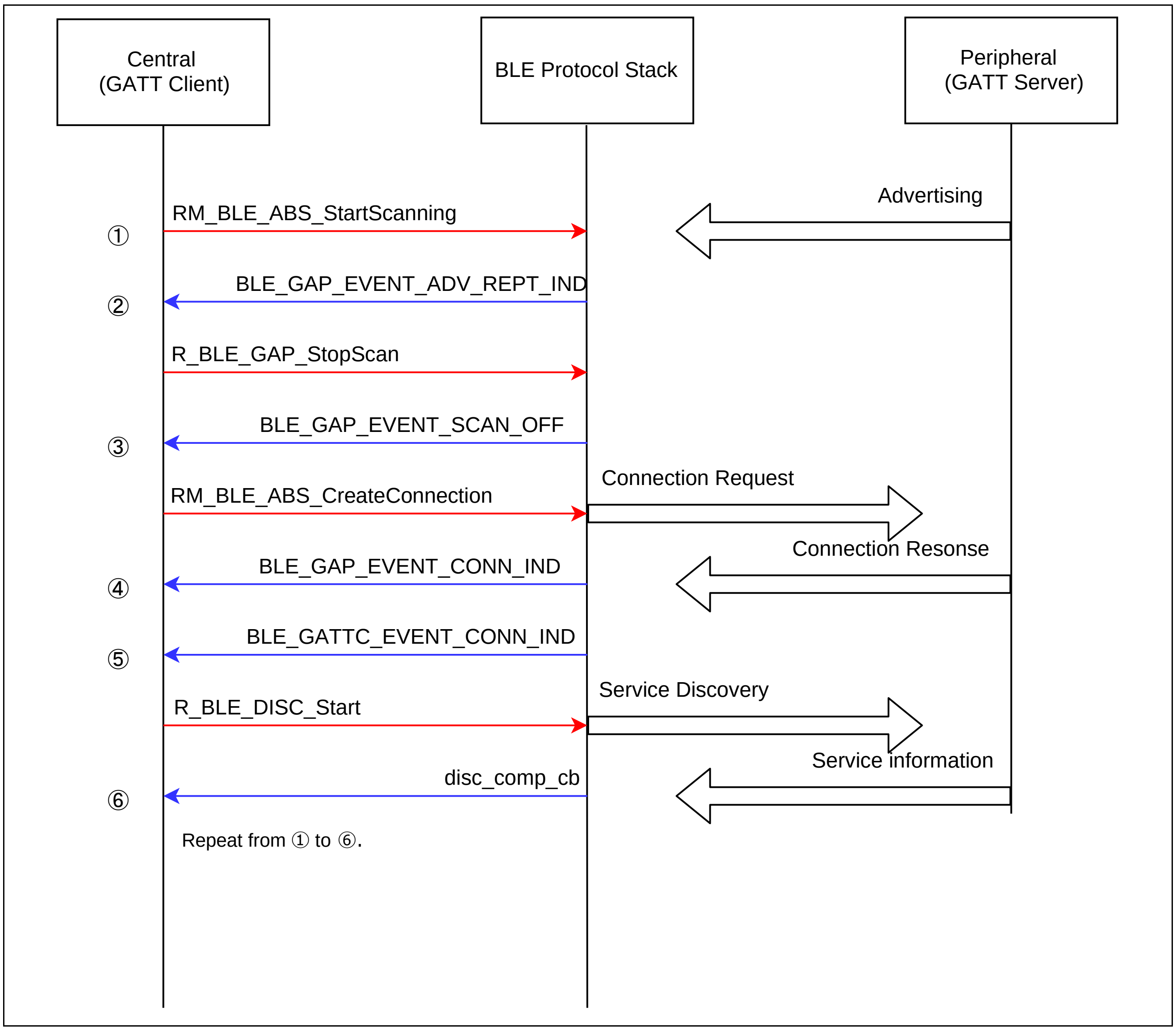


Figure 21. Sequence chart when connecting to a peripheral device

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 13, Code 14 and Code 15.

```
/* 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 13. Setting initial values for scan parameters and connection parameters

```
/* Connection handle */
uint16_t g_conn_hdl[BLE_CFG_RF_CONN_MAX];
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_CFG_RF_CONN_MAX;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_CFG_RF_CONN_MAX;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 14. 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 15. Implementation example of service discovery using Profile Common Library

As shown in bold frame of Code 15, 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 22. This is a one of typical case when home appliance equipment accepts control from multiple smart phones which perform GAP central.

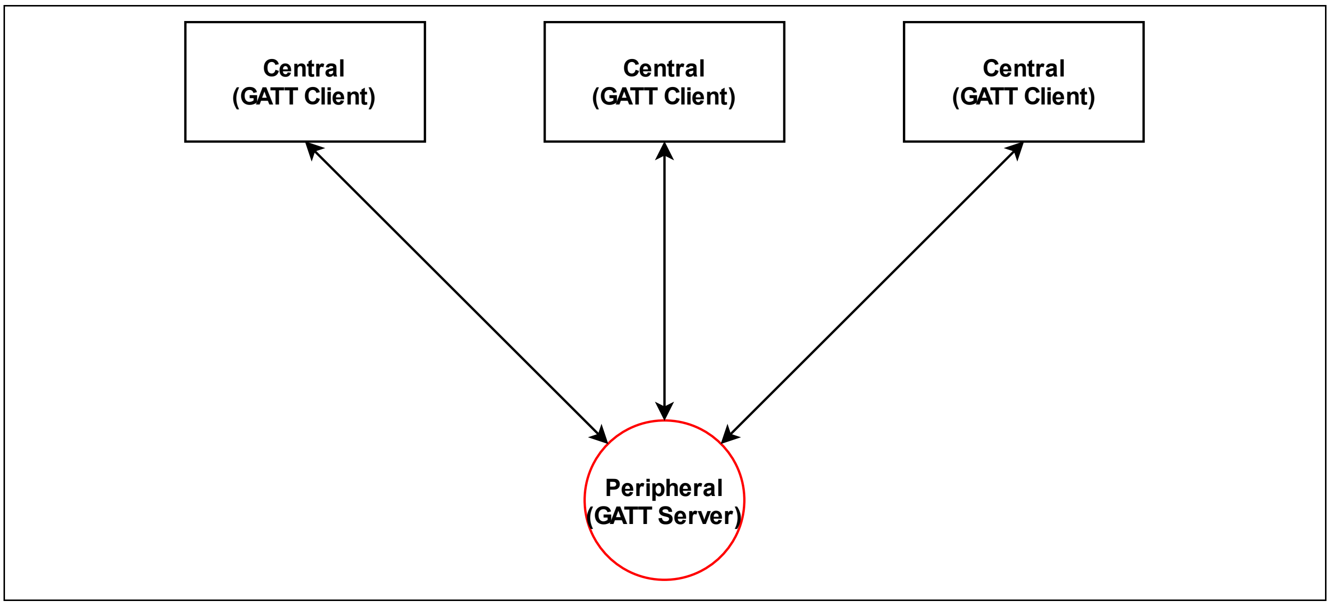


Figure 22. Connection with multiple central devices

Sequence chart of implementation example is shown in Figure 23.

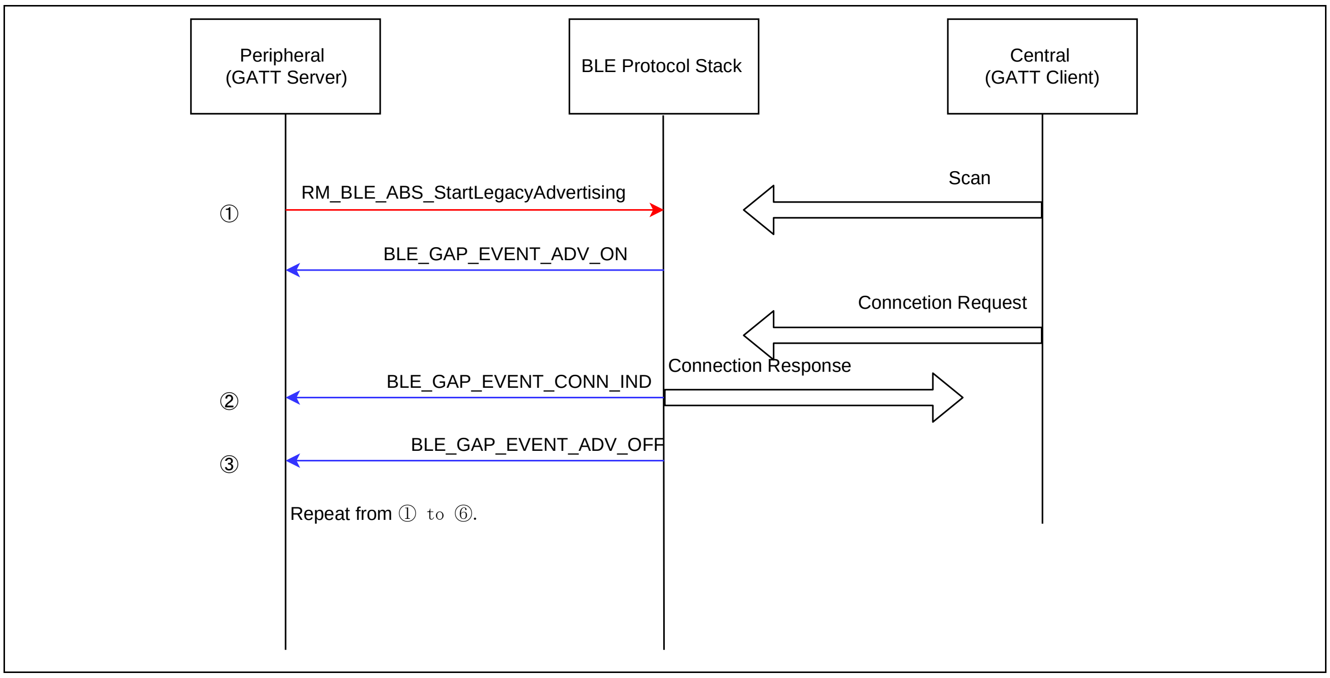


Figure 23. Sequence chart when connecting to a central device

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 16 and Code 17.

```
/* 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,
};
```

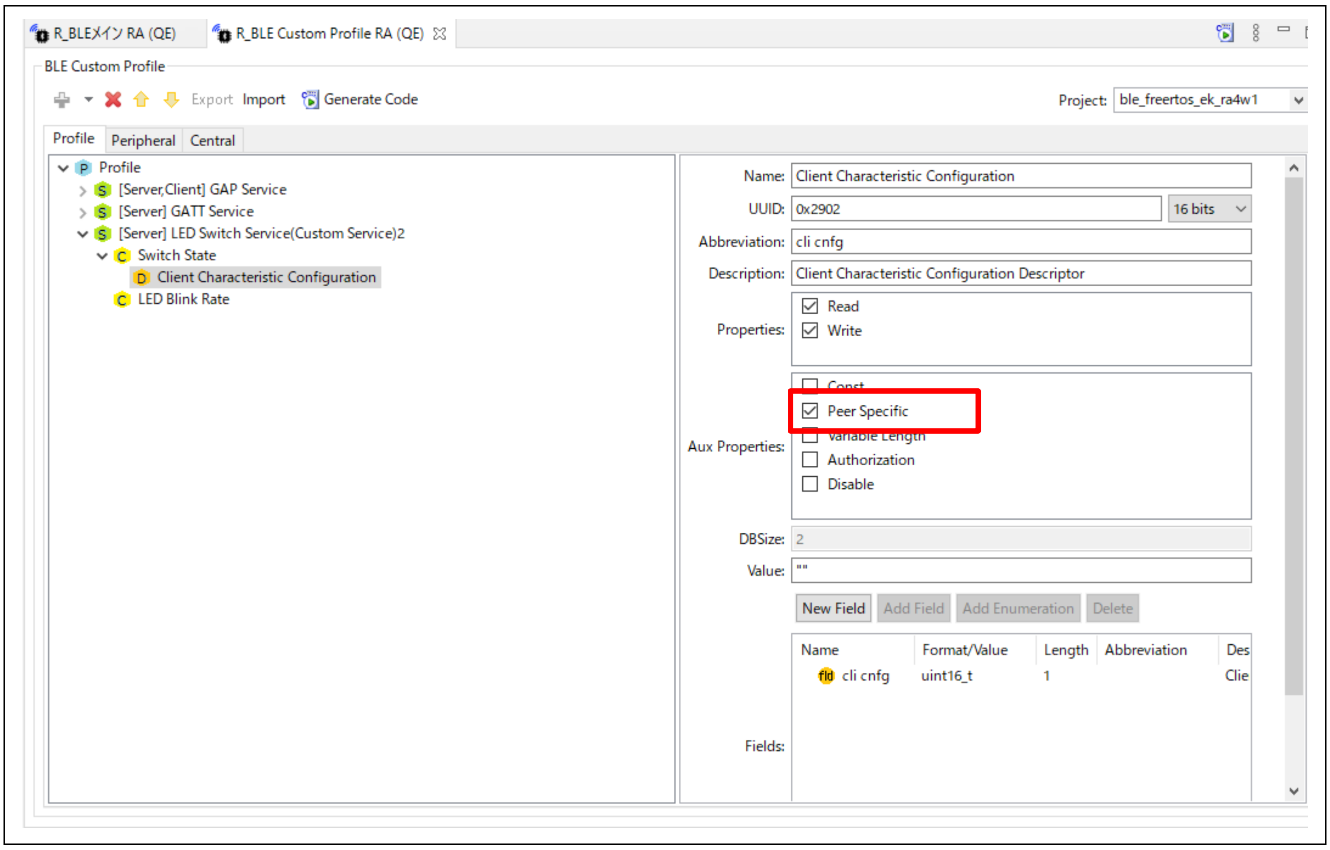
Code 16. Advertise packet and parameter settings

```
uint16_t g_conn_hdl[BLE_CFG_RF_CONN_MAX];

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_CFG_RF_CONN_MAX;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_CFG_RF_CONN_MAX;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 17. Implementation example of GAP callback function when accepting connections from multiple centrals

In Bluetooth Low Energy, the master (central device) controls the communication timing. Therefore, disconnection may happen when communication timing of each connection accidentally collided. To avoid such a disconnection, it is recommended to update the connection parameters so that there is a margin in slave latency and supervision timeout time. For updating 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 24.



Project: ble_freertos_ek_ra4w1

- Profile
  - [Server,Client] GAP Service
  - [Server] GATT Service
  - [Server] LED Switch Service(Custom Service)2
    - Switch State
      - Client Characteristic Configuration
    - LED Blink Rate

Name: Client Characteristic Configuration
UUID: 0x2902 16 bits
Abbreviation: cli cnfg
Description: Client Characteristic Configuration Descriptor
Properties: ☑ Read ☑ Write
Aux Properties: ☐ Const ☑ Peer Specific ☐ Variable Length ☐ Authorization ☐ Disable
DBSize: 2
Value: ""

Fields:

Name	Format/Value	Length	Abbreviation	Des
fld cli cnfg	uint16_t	1		Clie

Figure 24. 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 25. In such a case, local device communicates as central to one remote device and as a peripheral to another remote device.

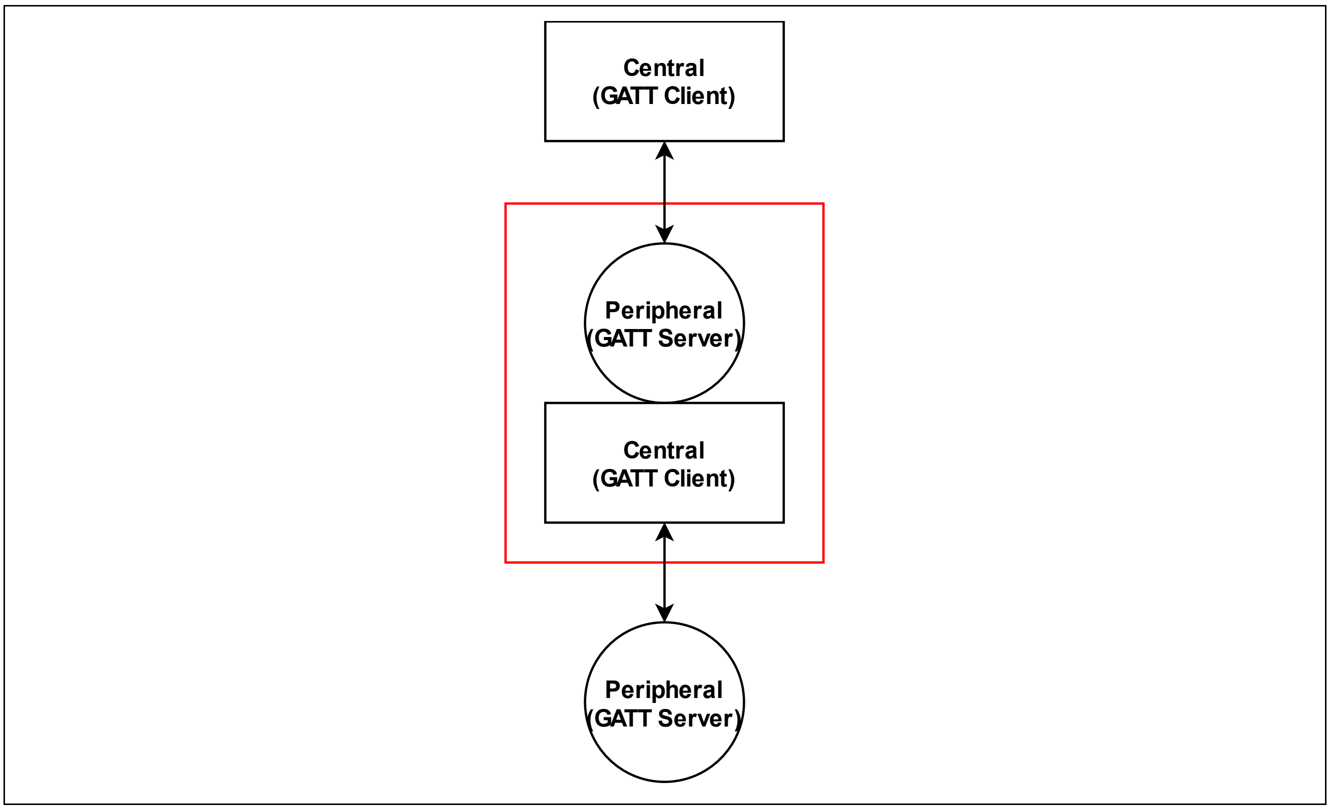


Figure 25. Multi roll connection example

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 18. 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 18. Call GAP callback function for each role

Each GAP role of example code is shown in Code 19 and Code 20.

```
/* 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  **/
```

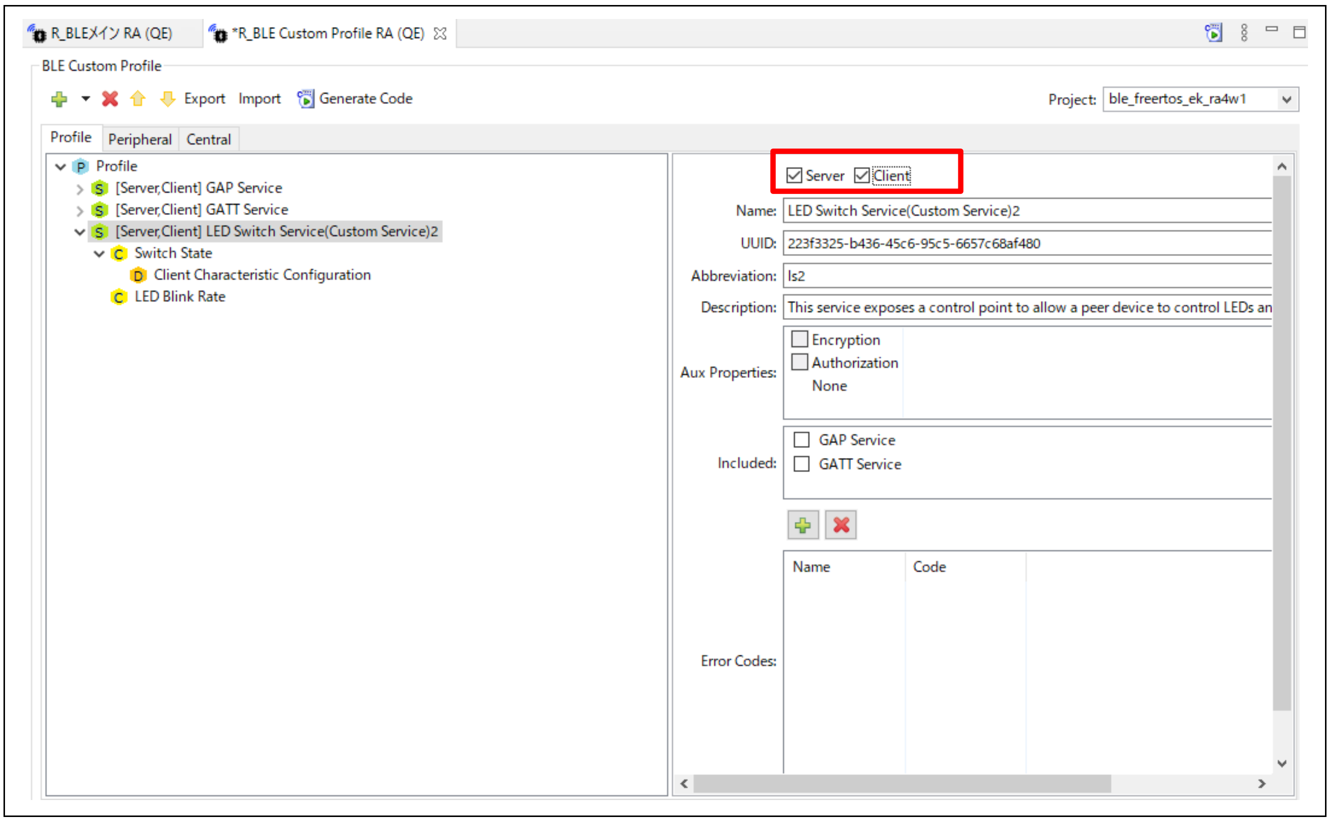
Code 19. Example of GAP callback function when connecting as a central role

```
/* 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 20. Example of GAP callback function when connected as a peripheral device

2. **GATT callback functions for each GATT role**

QE for BLE can generate skeleton code for GATT in case of multi roll. In the multi role case, enable both the server and the client on QE for BLE as shown in Figure 26, and generate the service API for both the GATT client and the server.



Project: ble_freertos_ek_ra4w1

- Profile
  - [Server,Client] GAP Service
  - [Server,Client] GATT Service
  - [Server,Client] LED Switch Service(Custom Service)2
    - Switch State
      - Client Characteristic Configuration
    - LED Blink Rate

☑ Server ☑ Client

Name: LED Switch Service(Custom Service)2

UUID: 223f3325-b436-45c6-95c5-6657c68af480

Abbreviation: ls2

Description: This service exposes a control point to allow a peer device to control LEDs an

Aux Properties: ☐ Encryption ☐ Authorization None

Included: ☐ GAP Service ☐ GATT Service

Error Codes:

Name	Code

Figure 26. Select GATT Role on QE for BLE

In the example code shown in Code 21, 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 21. Implementation example of service discovery as a central device

As shown in bold frame of Code 21, 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 the established connection, call the following *R_BLE_GAP_Disconnect* API. Need to 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, user application is notified of the *BLE_GAP_EVENT_DISCONN_IND* event. If the local device disconnects the link by *R_BLE_GAP_Disconnect* API, the *reason* field in the *st_ble_gap_disconn_evt_t* structure notified in the *BLE_GAP_EVENT_DISCONN_IND* event is 0x16 (Connection Terminated by Local Host). If the remote device disconnects the link, the *reason* field in the *st_ble_gap_disconn_evt_t* structure notified in the *BLE_GAP_EVENT_DISCONN_IND* event is specified as the reason why the remote device disconnects.

7. Communication

User 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 28 shows the correspondence between the communication parameters described in this chapter, supported features and the version of Bluetooth.

Table 28. Bluetooth version and supported features and parameters

Communication parameter	Feature name	Bluetooth Version	Description
PHY	LE 2M PHY LE Coded PHY LE 1M PHY	5.0 (optional [*1]) 5.0 (optional [*1]) 4.0	Double the symbol rate Forward error correction code added -
Maximums transmission packet length	LE Data Length Extension	4.2 (optional [*1])	Maximum number of transmitted bytes 27 → 251 bytes
Connection parameters	-	4.0	-
MTU	-	4.0	-

[*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 "*RA Flexible Software Package Documentation*" about details of these APIs.

7.1 Changing PHY

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

- **LE 1M PHY**
  - This is a modulation method that compatibles 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. User 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 LE1M and LE2M 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 27 shows the sequence chart when changing the PHY from the local device. In the figure, the local device performs as central, but PHY changes can be initiated from either role.

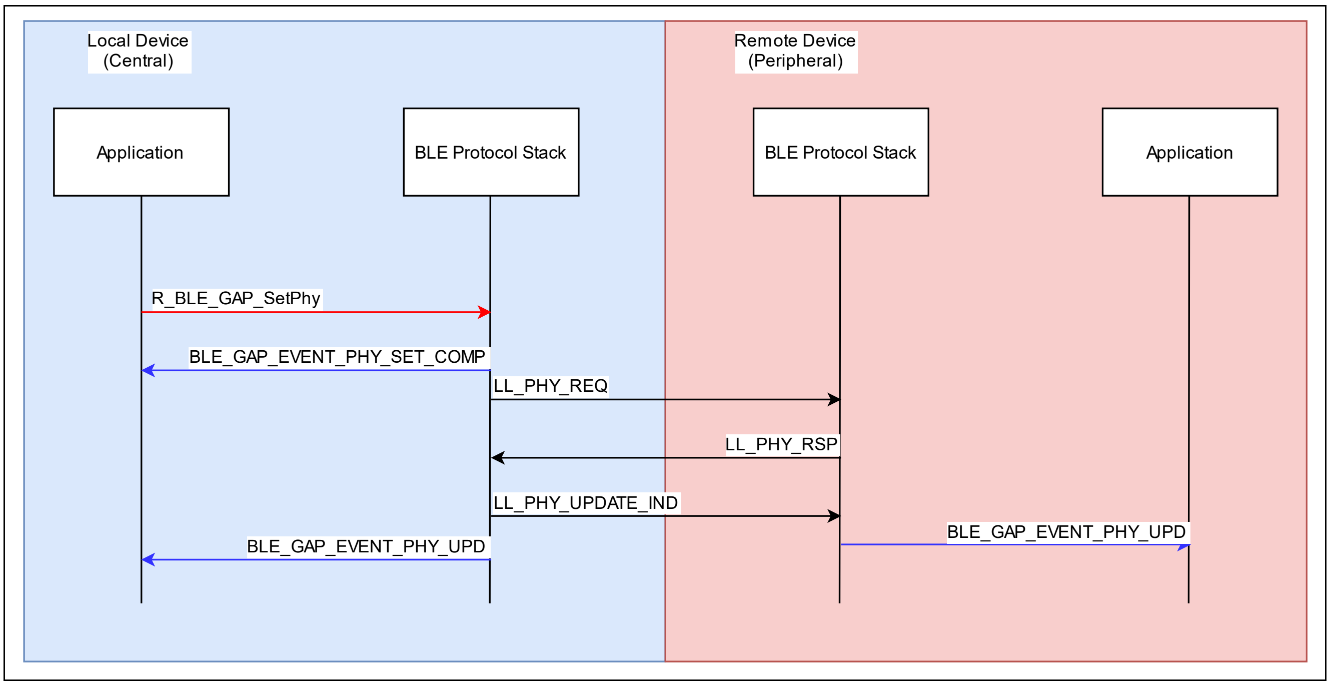


Figure 27. Sequence chart when changing PHY

The example for changing the PHY to LE Coded PHY (S=8) is shown in Code 22. Multiple PHYs can be also specified by bit sum.

```
st_ble_gap_set_phy_param_t set_phy = {
     .tx_phys = BLE_GAP_SET_PHYS_HOST_PREF_CD | BLE_GAP_SET_PHYS_HOST_PREF_1M,
     .rx_phys = BLE_GAP_SET_PHYS_HOST_PREF_CD | BLE_GAP_SET_PHYS_HOST_PREF_1M,
     .phy_options = BLE_GAP_SET_PHYS_OP_HOST_PREF_S_8
};

R_BLE_GAP_SetPhy(conn_hdl, &set_phy);
```

Code 22. 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 23.

- **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 23. 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 2704usec. 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. User can perform efficient communication by extending the transmitting packet length when user 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 maximum transmission time. The packet transmission time is depended on PHY configuration which describes in the previous section. And 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 29.

Table 29. 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 Protocol Stack 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 Abstraction Driver on rm_ble_abs*”. Call *R_BLE_GAP_SetDataLen* API to change maximum transmission packet length. It is necessary to specify connection handle whose configuration will be changed, maximum number of bytes to send and maximum transmission time in microsecond as argument of the API. The BLE Protocol Stack 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 28 shows a sequence chart when maximum transmission packet length from the local device.

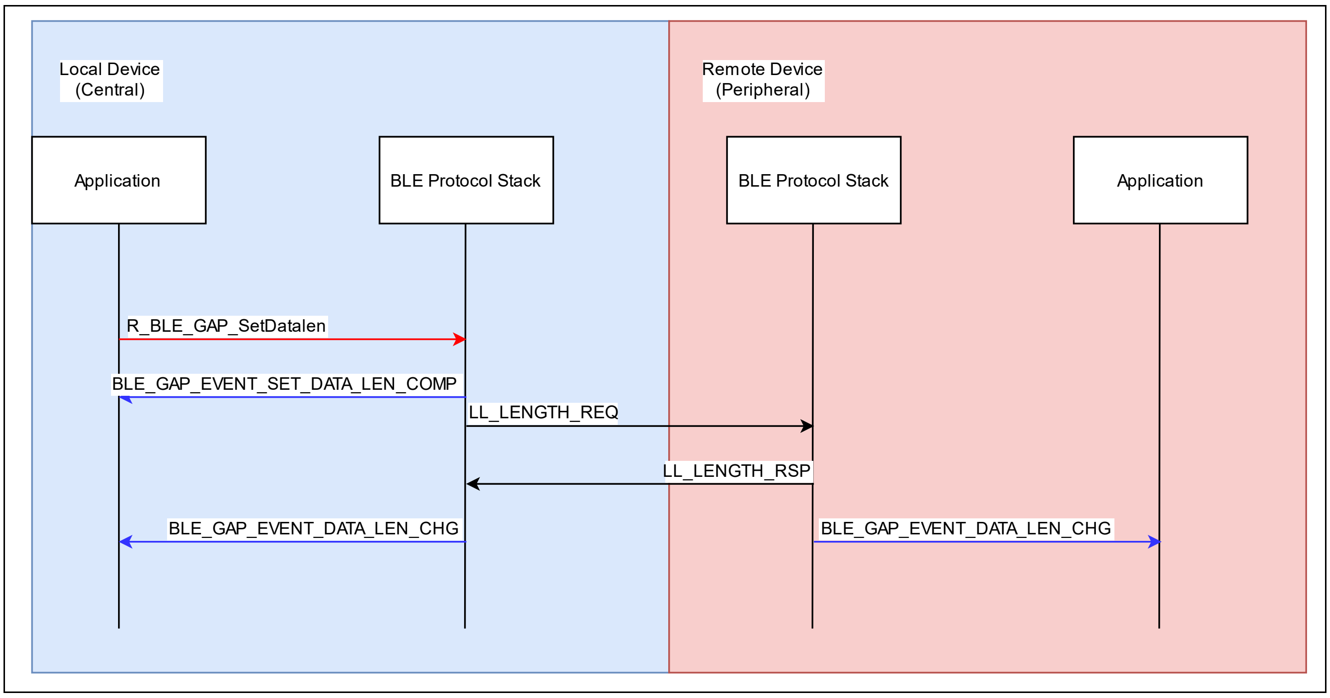


Figure 28. Sequence chart when changing the maximum transmission packet length

Code 24 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 24. 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 25.

- **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 25. 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.

- **Slave Latency**
  - The number of times the slave will ignore packets from the master. When the slave receives a packet from the master, it returns a response. If there is no data to be transmitted from the slave, the packet from the master can be ignored for the number of times set for slave latency. The slave does not have to return the response for the number of times, so the power consumption can be reduced. Figure 29 shows the relationship between Slave latency and connection event.

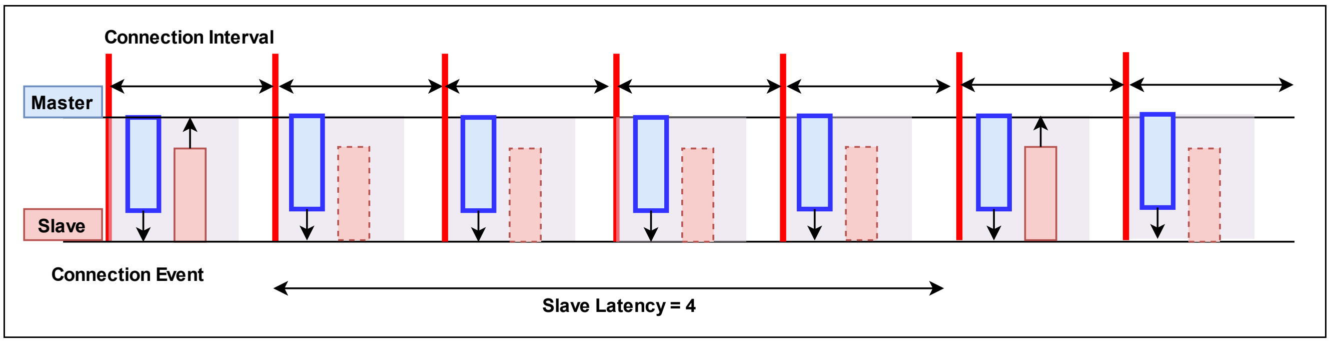


Figure 29. Slave latency and connection event

- **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:

$$Supervision\ Timeout(msec) > \big(1 + Slave\ Latency(number)\big) * Connection\ Interval(msec) * 2$$

- **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 30. 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 31.

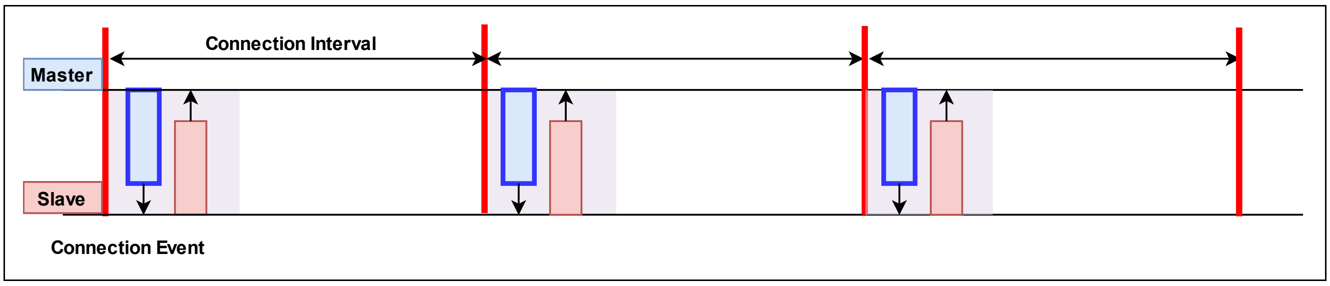


Figure 30. Connection event time and packet exchange (connection event time is set to 0)

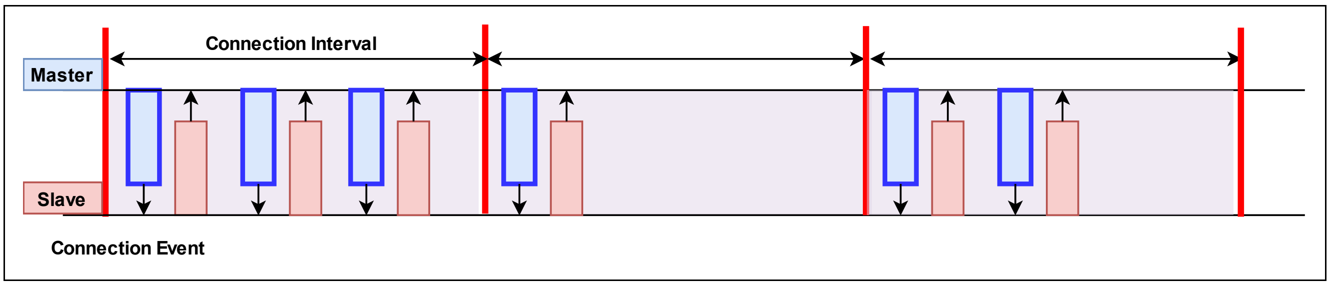


Figure 31. Connection event time and packet exchange (connection event time is set to 0xFFFF)

The master determines and changes the connection parameters, but it is also possible to request connection parameters changes from slave to master. 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. Followings 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 a short time.

Figure 32 shows the sequence chart for updating the connection parameters. The local device is the central and the remote device is the peripheral.

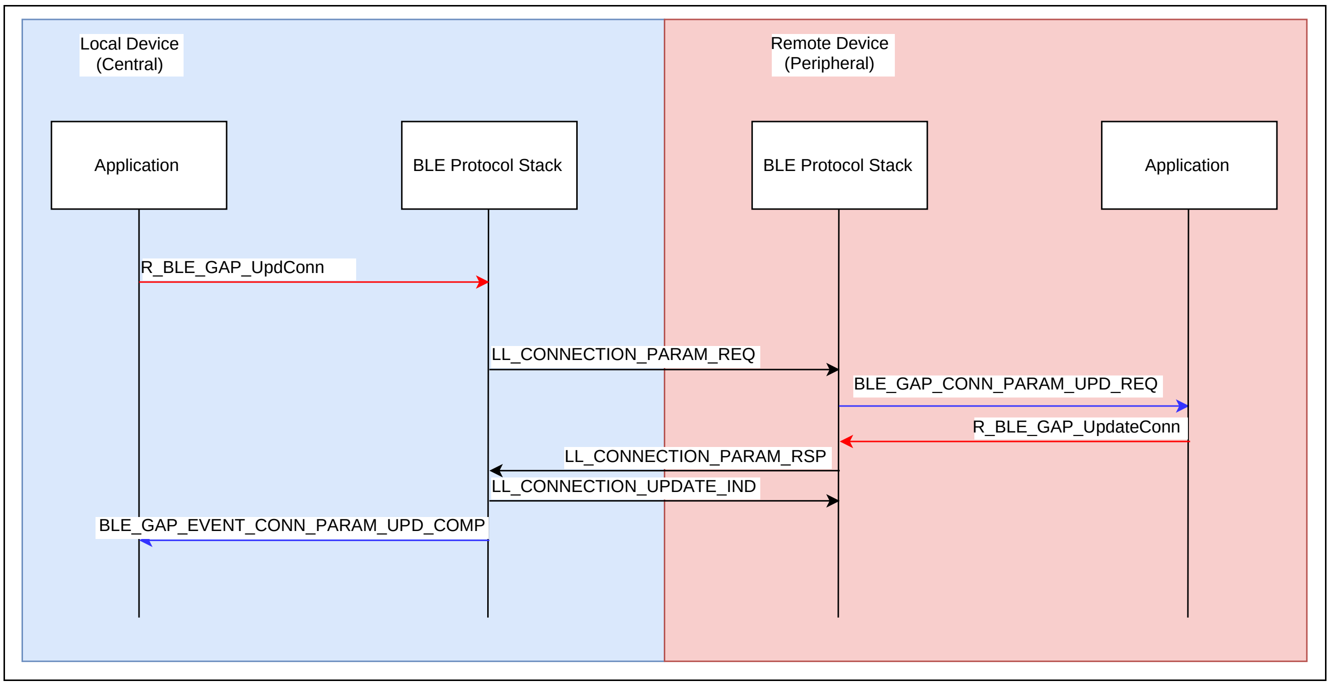


Figure 32. Sequence chart when updating connection parameters

Use *R_BLE_GAP_UpdConn* API for request/response of connection parameter update. Code 26 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,  //Slave 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 26. 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. 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. And the argument *event* of *gap_cb* contains the connection parameters used in the actual connection.

Code 27 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 27. 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 28 is an implementation example in which the parameters are updated and request 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,      /* Slave 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 28. 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 MTU size is 23 bytes. This is called the default MTU. When user 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 mean 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 mean 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 33 shows the sequence chart when changing the MTU.

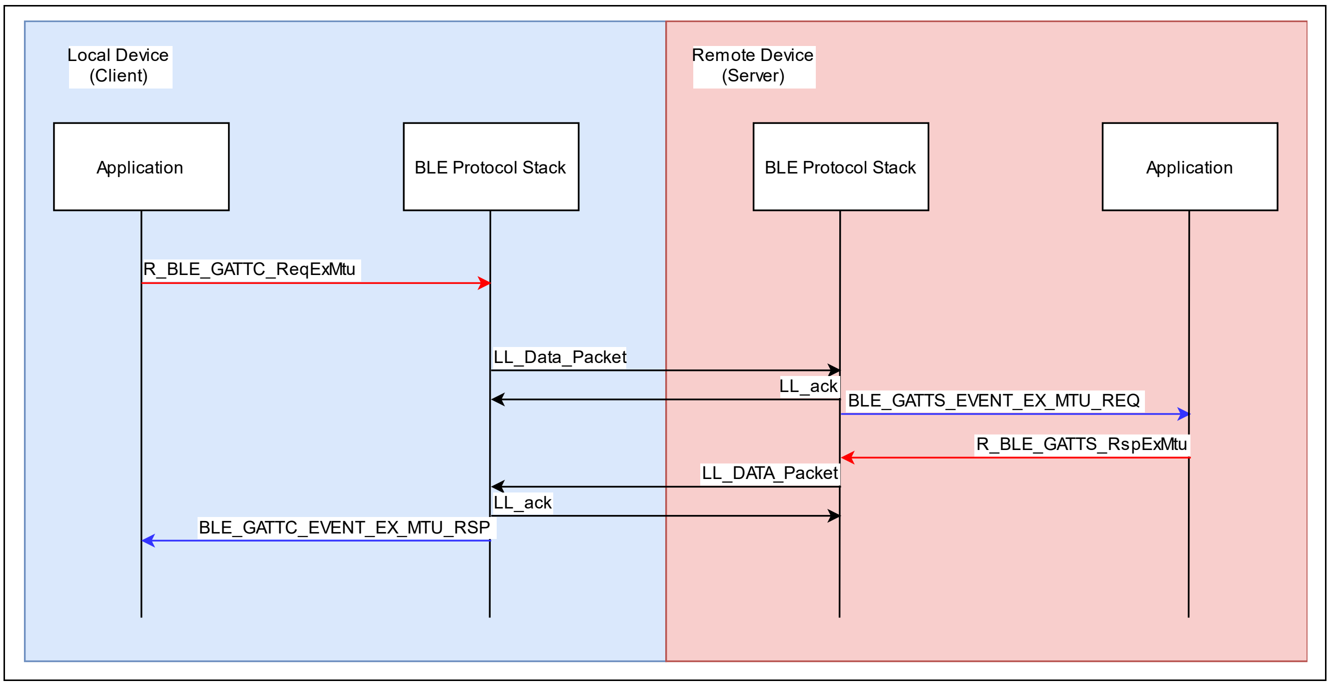


Figure 33. Sequence chart when changing MTU

Call *R_BLE_GATT_ReqExMtu* API to change the MTU, as shown in Code 29.

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

Code 29. 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 30 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. And the size of the MTU returned by the GATT server is set in the *MTU Size Configured* configuration in properties of *BLE Abstraction Driver on rm_ble_abs*. When user generated 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_CFG_GATT_MTU_SIZE);
        } break;
    }
}
```

Code 30. Example of response to MTU change request

7.5 Flow control

BLE Protocol Stack has a flow control feature to send large application data in a short time. BLE Protocol Stack 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 34 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 buffer reached *Low Water Mark*. Application should stop sending application data to prevent buffer overflow when receive the event.

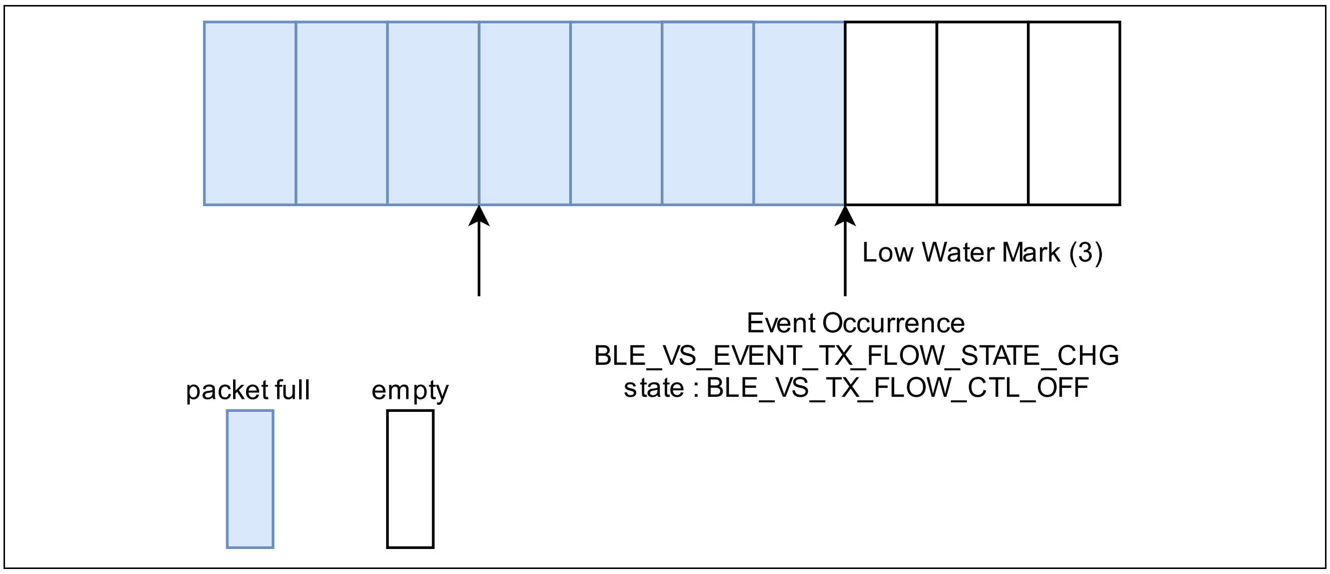

Low Water Mark (3)
Event Occurrence
BLE_VS_EVENT_TX_FLOW_STATE_CHG
state : BLE_VS_TX_FLOW_CTL_OFF
packet full
empty

Figure 34. Number of empty buffers and events (Reach Low Water Mark)

Remaining empty buffer will increase as BLE Protocol Stack 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 35. Application should resume sending application data when receiving the event.

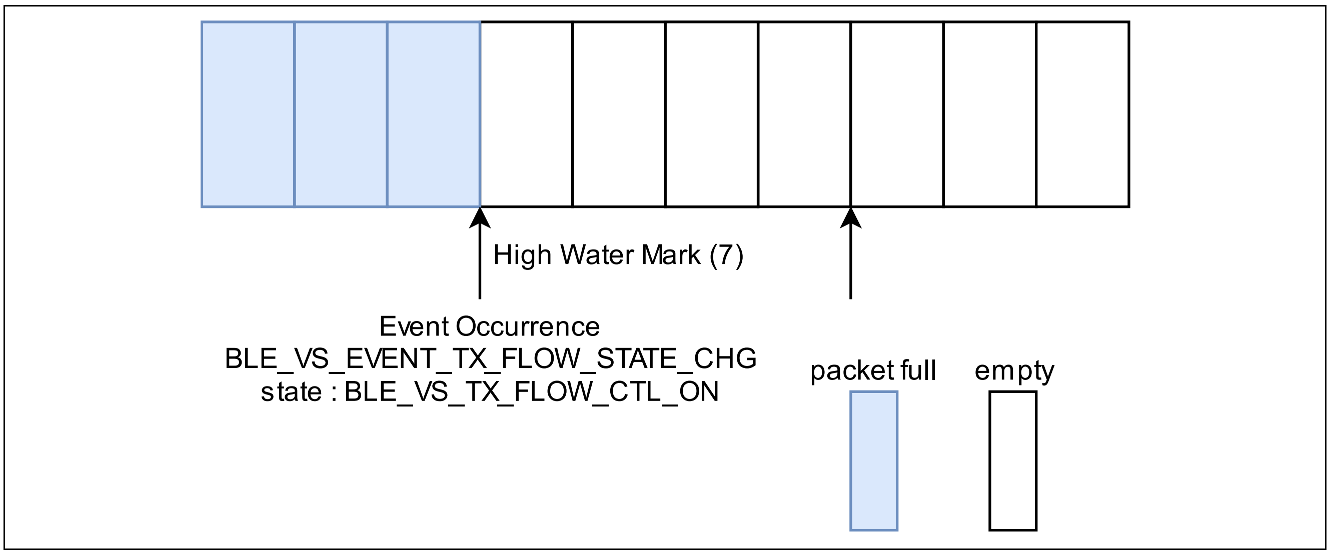


Figure 35. Number of empty buffers and events (Reach High Water Mark)

Application can be transmitted large data efficiently by repeating 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 31. *R_BLE_VS_SetTxLimit* API configure *Low Water Mark* and *High Water Mark* of the send buffer where *BLE_VS_EVENT_TX_FLOW_STATE_CHG* event occurs. And *R_BLE_VS_StartTxFlowEvtNtf* API to 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 31. 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 inCode 32. When number of empty buffer 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 32. Implementation example of sending by flow control feature event

R_BLE_ServsCharNotification API is just 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 the Bluetooth Low Energy.

8.1 Pairing

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

- The remote device sets security requirement for accessing the GATT service.
- The local device resolves the remote device address.

Pairing procedure exchanges following keys with a remote device.

- **LTK (Long Term Key)**
  - ➢ The LTK will be used as encryption key.

- **IRK (Identity Resolving Key)**
  - ➢ The IRK will be used as resolving private address of remote device.

- **CSRK (Connection Signature Resolving Key)**
  - ➢ Signed data transmission 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 shows Figure 36. The following sections describe the details of pairing steps.

Flow chart of pairing procedure is shown in Figure 36.

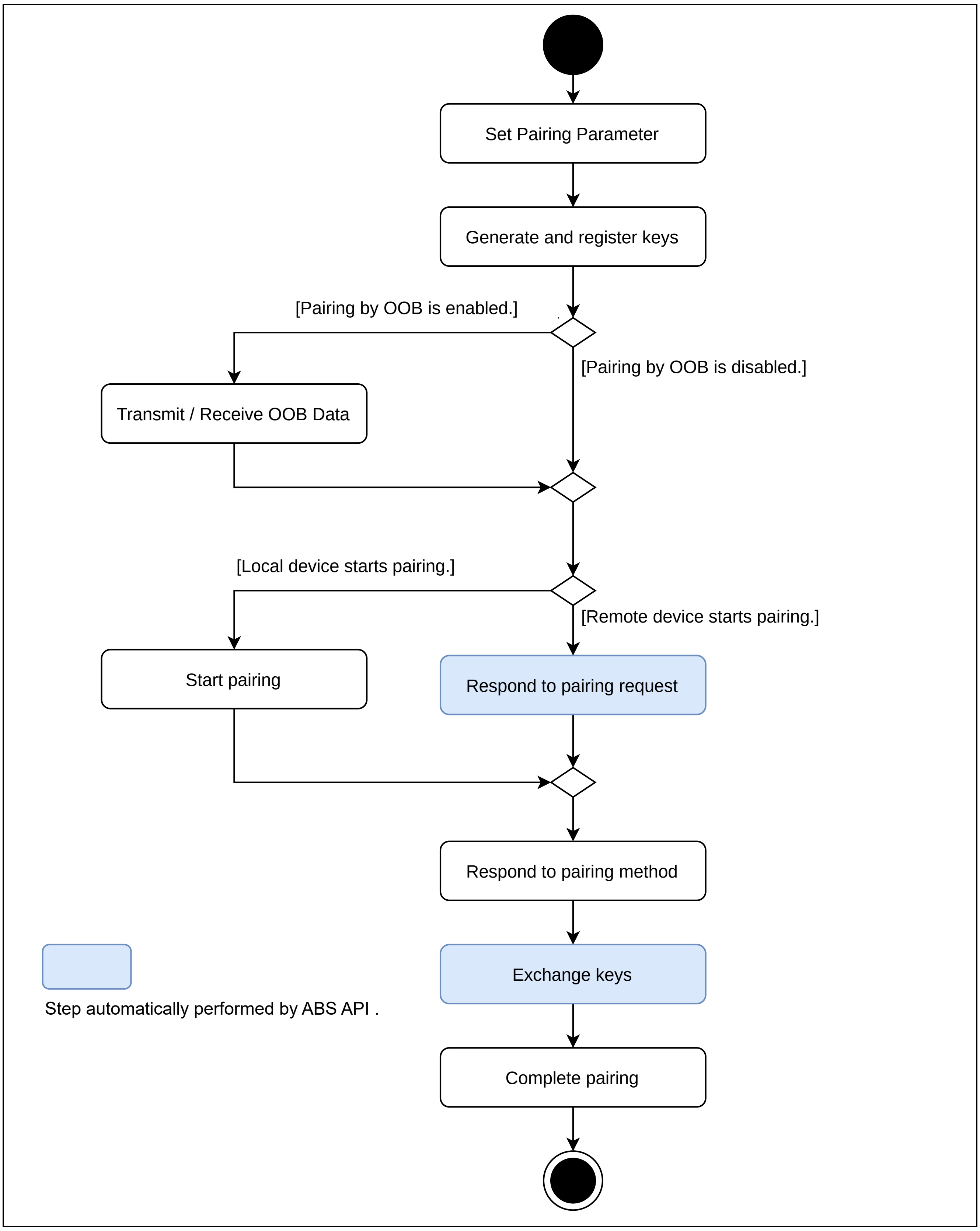

Set Pairing Parameter
Generate and register keys
[Pairing by OOB is enabled.]
[Pairing by OOB is disabled.]
Transmit / Receive OOB Data
[Local device starts pairing.]
[Remote device starts pairing.]
Start pairing
Respond to pairing request
Respond to pairing method
Exchange keys
Complete pairing
Step automatically performed by ABS API .

Figure 36. 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 30 shows the pairing parameters. And following sections describe the detail of these parameters.

Table 30. 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_ NOINPUT_NOOUTPUT (0x03)
				BLE_GAP_IOCAP_DISPLAY_ YESNO(0x01)	
				BLE_GAP_IOCAP_KEYBOARD_ ONLY(0x02)	
				BLE_GAP_IOCAP_NOINPUT_ NOOUTPUT(0x03)	
				BLE_GAP_IOCAP_KEYBOARD_ DISPLAY(0x04)	
2. MITM Protection Request		mitm_protection_policy	mitm	BLE_GAP_SEC_MITM_BEST_ EFFORT(0x00)	BLE_GAP_SEC_MITM_ BEST_EFFORT(0x00)
				BLE_GAP_SEC_MITM_STRICT(0x01)	
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 .. 16	16
	Min Size	maximum_key_size	min_key_size	7 .. 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)	0
				BLE_GAP_KEY_DIST_SIGNKEY(0x04)	
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_NOT_SPRT(0x00)
				BLE_GAP_SC_KEY_PRESS_NTF_ SPRT(0x01)	
7. LE Secure Connections Request		secure_connection_only	sec_conn_only	BLE_GAP_SC_BEST_EFFORT(0x00)	BLE_GAP_SC_BEST_ EFFORT(0x00)
				BLE_GAP_SC_STRICT(0x01)	

1. Input Output capabilities

Table 31 and Table 32 show Input and output capabilities that local device support.

Table 31. 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 32. 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 33.

Table 33. 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 34 specify whether to require protection against MITM.

Table 34. MITM Protection

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

Completing pairing with the pairing method except Just Works according to section 8.1.6 enables the MITM protection.

3. Bonding

Table 35 shows the bonding parameter settings whether the local device perform bonding or not. For more details about bonding, refer to section 8.2.

Table 35. 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 recommends that the encryption key size is 16 bytes because the short encryption key size causes to reject access to the remote device.

5. Type of key exchanged by pairing

Table 36 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 36. Key Type

Key type	Settings
LTK	BLE_GAP_KEY_DIST_ENCKEY(0x01)
IRK	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 37.

Table 37. 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 38 shows the parameter determine whether pairing is permitted by only LE Secure Connections or not.

Table 38. 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)

An example of setting the pairing parameters by using *R_BLE_GAP_SetPairingParams* API is shown in Code 33.

```
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 33. Example of setting pairing parameter

Above code does not need when the application uses *RM_BLE_ABS_Open* API.

8.1.2 Key generation and registration

This section describes how to generate and register IRK and CSRK. These keys are used for key exchange which is one of pairing procedures. Related APIs are shown in Table 39.

Table 39. The APIs used for key generation

Key	API for key generation
IRK	RM_BLE_ABS_SetLocalPrivacy [*1] or R_BLE_GAP_SetLocIdInfo
CSRK	R_BLE_GAP_SetLocCsrk

[*1] : *RM_BLE_ABS_SetLocalPrivacy* API performs both of generation and registration local device IRK.

An example of key generation and registration are shown in Code 34. 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 34. Example of key generation and registration

Some notes about key generation and registration are shown in following.

- 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 before start pairing.

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 by OOB. The data consists of confirm value (16 bytes) and random value (16 bytes). It needs to meet the condition in Table 40 to do pairing by OOB. If OOB is available, the data is transmitted and received before starting pairing.

Table 40. 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 when the local device send data by OOB. The API will generate confirm value (16 bytes) and random value (16 bytes) as data for pairing according to 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.

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 central and 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 with the event by using *R_BLE_GAP_ReplyPairing* API. An example of responding a pairing request is shown in Code 35.

```
/* 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 35. Response to a pairing request

If *RM_BLE_ABS_StartAuthentication* API is used, when receiving BLE_GAP_EVENT_PAIRING_REQ event, call *R_BLE_GAP_ReplyPairing* API to automatically respond to a pairing request.

8.1.6 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 in Table 41 and perform the pairing method.

Table 41. Pairing Method

Pairing Method	Description	MITM Protection
OOB	The application does not need to handle the 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 handle the 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 42.

Table 42. 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.
  - **[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.

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 36.

```
/* 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 36. Sample of responding to a key distribute request

If RM_BLE_ABS_StartAuthentication API is used, when BLE_GAP_EVENT_EX_KEY_REQ is notified, call *R_BLE_GAP_ReplyExKeyInfoReq* API to automatically respond to the key distribution request.

8.1.8 Completion of pairing

When 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*(0x00). Any other value indicates a pairing failure.

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 37 shows the bonding procedure.

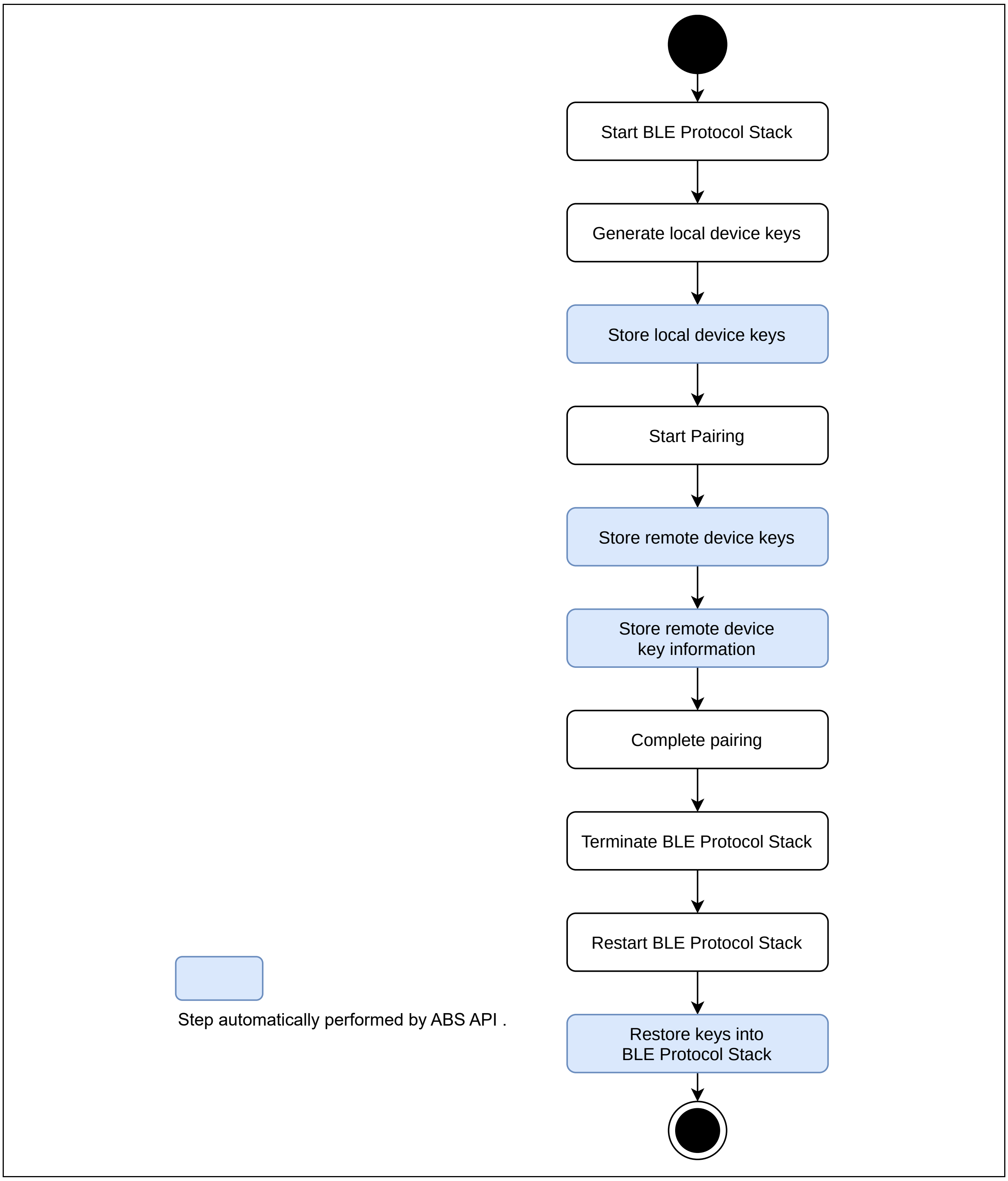

Start BLE Protocol Stack
Generate local device keys
Store local device keys
Start Pairing
Store remote device keys
Store remote device key information
Complete pairing
Terminate BLE Protocol Stack
Restart BLE Protocol Stack
Restore keys into BLE Protocol Stack
Step automatically performed by ABS API .

Figure 37. 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)

An example of storing remote device keys is shown in Code 37.

```
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 37. Sample of storing received keys

If *RM_BLE_ABS_StartAuthentication* API is used and *Store security data* option on *BLE Abstraction Driver on rm_ble_abs* 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. And *Data Flash Block for Security Data* configuration on properties of *BLE Abstraction Driver on rm_ble_abs* specifies which block of Data Flash used for storing key information.

If the Abstraction API is not used or *Store security data* option on *BLE Abstraction Driver on rm_ble_abs* 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 automatically.

8.2.2 Store local device keys

If the local device uses the privacy feature, the IRK and the identity address registered by *R_BLE_GAP_SetLocIdInfo* API or *RM_BLE_ABS_SetLocalPrivacy* API. When the Abstraction API is used and *Store Security Data* configuration on properties of *BLE Abstraction Driver on rm_ble_abs* is enabled, the local device IRK generated by *RM_BLE_ABS_SetLocalPrivacy* API and the identity address are automatically stored in 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 reset 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 are automatically reset to the BLE Protocol Stack in restarting.

8.3 Encryption

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

8.3.1 Request Encryption

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

- *RM_BLE_ABS_StartAuthentication*
- *R_BLE_GAP_StartEnc*

Depending on the remote device implementation, the remote device does not respond 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 38 and Figure 39.

1. Encryption request from local device(master)

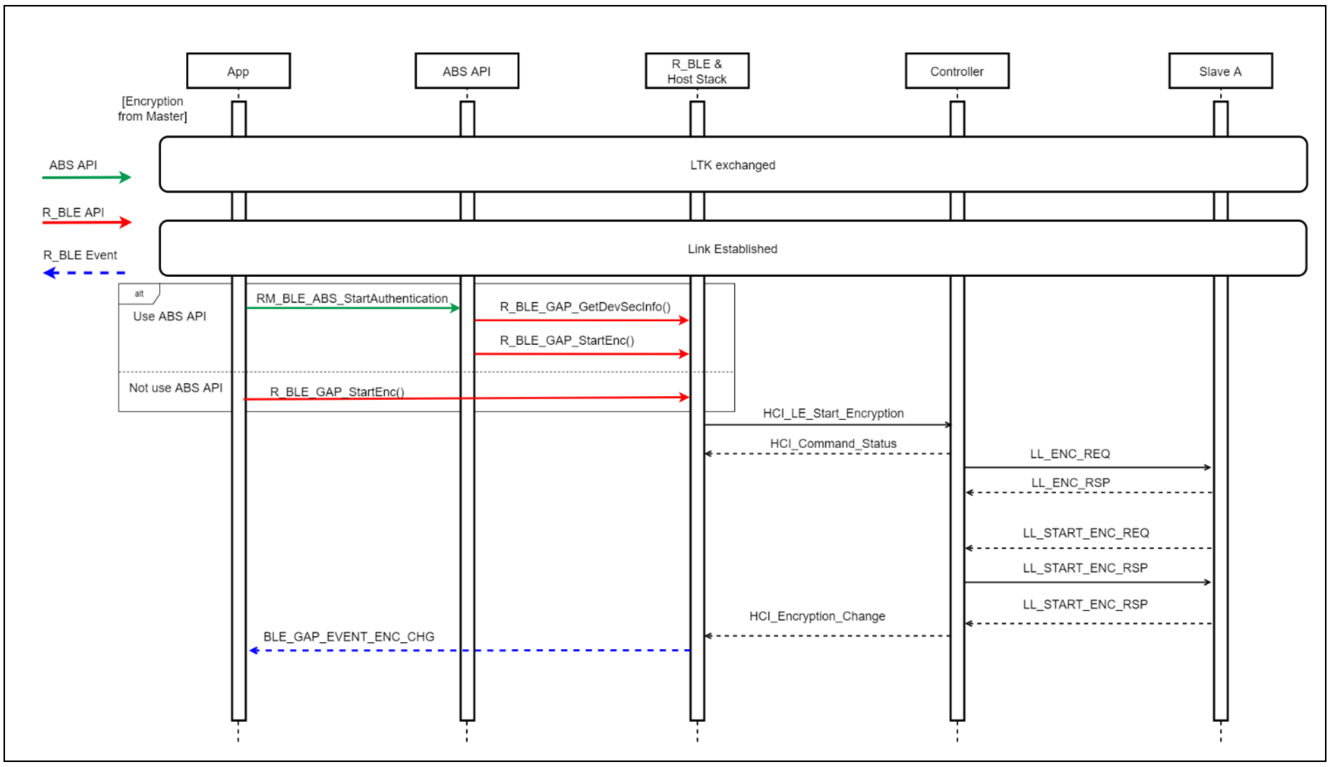


Figure 38. Sequence of encryption request from local device(master)

2. Encryption request from local device(slave)

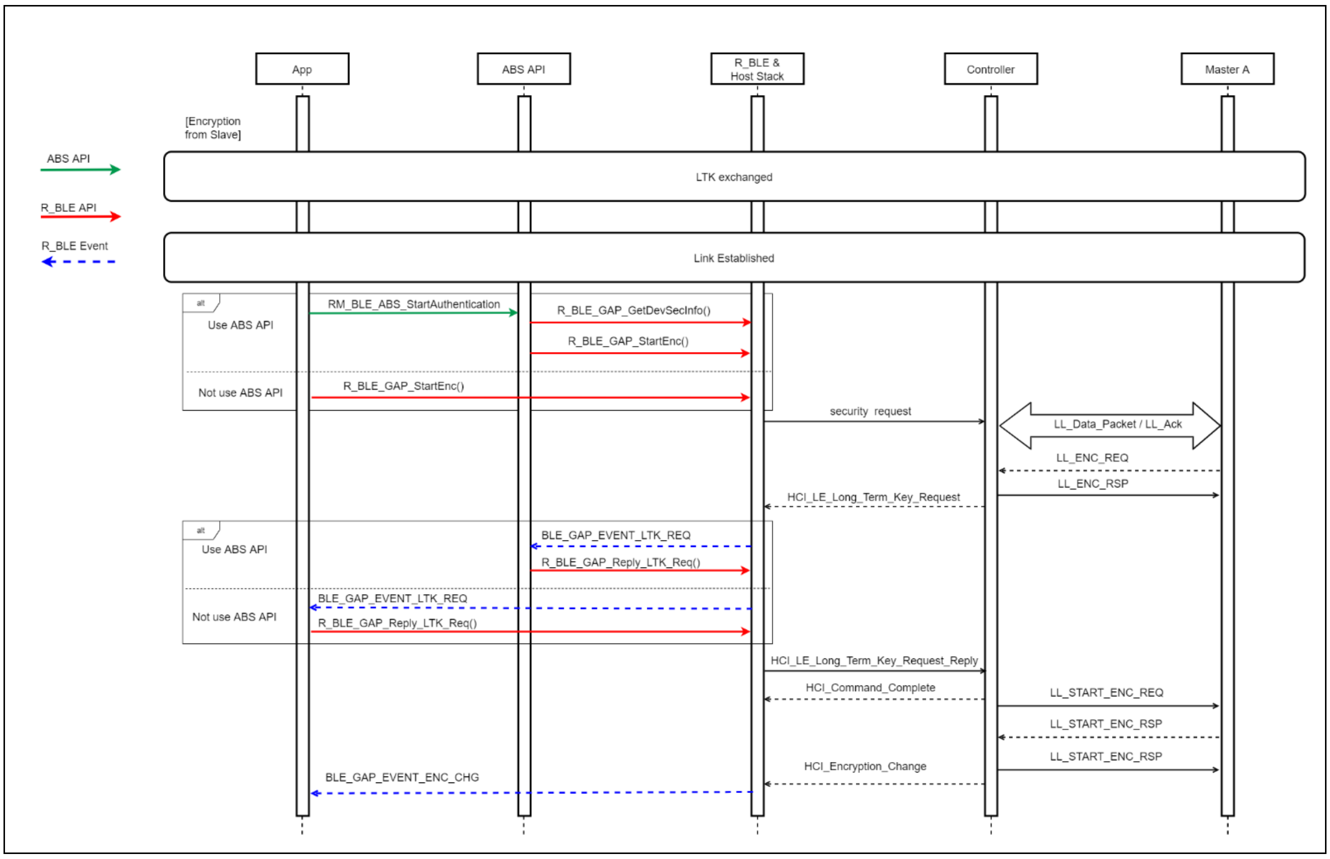


Figure 39. Sequence of encryption request from local device(slave)

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. If the encryption is complete successfully, 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.

An example of an encryption request event and respond API 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 **/
    /* 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 38. Sample of responding an encryption request in the event

If local device using Abstraction API to start encryption procedure, above response procedure is automatically perform.

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 40 and Figure 41.

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

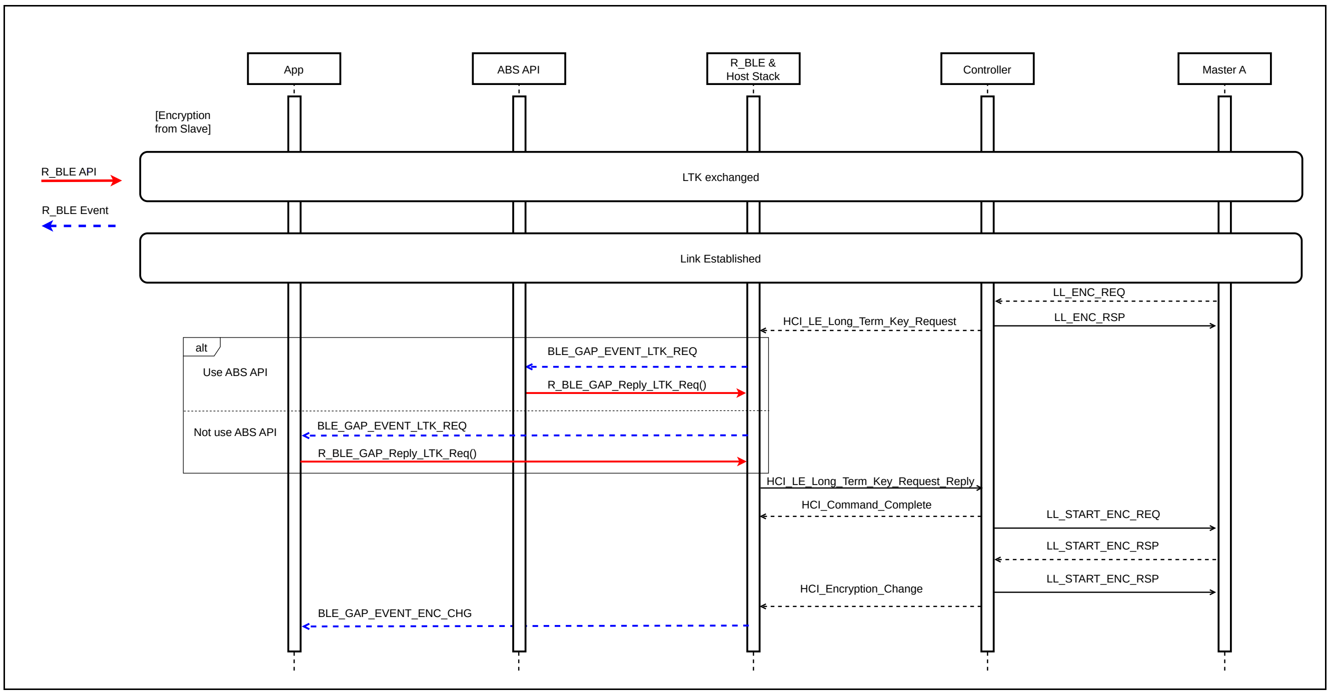


Figure 40. Sequence of response to an encryption request from remote device(master)

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

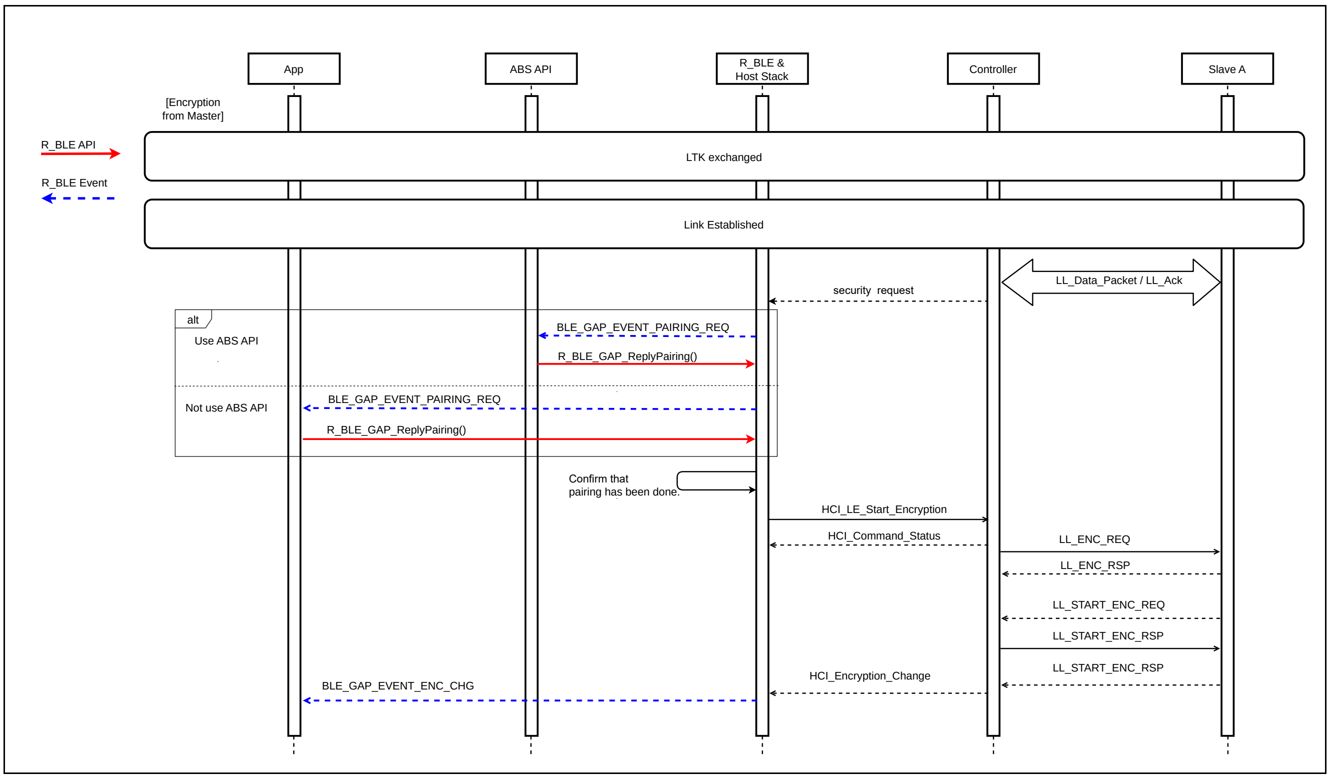


Figure 41. Sequence of response to an encryption request from remote device(slave)

8.3.3 Completion of encryption

If the encryption has been completed successfully, user application is notified of *BLE_GAP_EVENT_ENC_CHG* event. If the encryption has been failed because the remote device lost the LTK, user application is notified of *BLE_GAP_EVENT_PAIRING_COMP* event with result of *BLE_ERR_SMP_LE_LOC_KEY_MISSING*(0x2014). If the event is received, delete the local device LTK and do pairing again and encrypt.

8.4 Privacy

The privacy feature allows local device to change the address not to be identified from other devices. There are two privacy mode: Network Privacy Mode and Device Privacy Mode. In Network Privacy Mode, both local device and remote device use RPA. In Device Privacy Mode, only local device uses RPA. Default is Network Privacy Mode.

8.4.1 Generate and resolve local device RPA

Before local device uses RPA, perform the following step1-4. The API called in step 1-4 can replace *RM_BLE_ABS_SetLocalPrivacy* API.

1. **Register local device key (IRK) and BD address**

   Call *R_BLE_VS_GetRand* API to generate the random value (16 bytes) notified by BLE_VS_EVENT_GET_RAND event as IRK. The IRK and identity address are registered by R_BLE_GAP_SetLocIdInfo into the BLE Protocol Stack. The IRK is distributed to the remote device in pairing.

2. **Register the IRK to the Resolving List**

   Call *R_BLE_GAP_ConfRslvList* API to register the IRK generated by step 1 in 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. If only the local device is uses RPA or it is in unpaired state, register a dummy remote device identity address (e.g. All 0x55) and IRK (e.g. 0xAA) to associate with the local device IRK. The completion is notified by *BLE_GAP_EVENT_RSLV_LIST_CONF_COMP* event.

3. **Set Privacy Mode**

   If Network Privacy Mode which is used, the procedure does not need to be done.
   Call *R_BLE_GAP_SetPrivMode* API to set the privacy mode. The completion is notified by *BLE_GAP_EVENT_PRIV_MODE_SET_COMP* event.

4. **Start RPA feature**

   Call *R_BLE_GAP_EnableRpa* API to enable the RPA generation and resolution.
   BLE_GAP_EVENT_RPA_EN_COMP event notifies user application of the completion.

An example of the 1 - 4 procedure is shown in Code 39.

```
/** some code is omitted  **/
#include "sec_data/r_ble_sec_data.h"
/** 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);

            /*  store local id info to Data Flash */

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

            memset(peer_irk.remote_irk, 0xAA, BLE_GAP_IRK_SIZE);
            peer_irk.local_irk_type = BLE_GAP_RL_LOC_KEY_REGISTERED;
            memset(gs_rem_bd_addr.addr, 0x55, 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;
                ev_param = (st_ble_gap_loc_dev_info_evt_t *)p_data->p_param;
                gs_loc_bd_addr = ev_param->l_dev_addr;
                /* Generate IRK */
                R_BLE_VS_GetRand(BLE_GAP_IRK_SIZE);
            } 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 39. Prepare for using RPA in the local device (1)

An example when using *R_BLE_ABS_SetLocPrivacy* API is shown in Code 40.

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

/* Advertising parameters */
static ble_abs_legacy_advertising_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_data)
{
    switch(event_type)
    {
        case BLE_GAP_EVENT_LOC_VER_INFO:
            {
                R_BLE_ABS_SetLocPrivacy(&g_ble_abs0_ctrl, NULL, BLE_GAP_DEV_PRIV_MODE);
            } break;

        case BLE_GAP_EVENT_RPA_EN_COMP:
            {
                /* Start advertising */
                memset(gs_adv_param.p_addr->addr, 0x55, BLE_BD_ADDR_LEN);
                gs_adv_param.p_addr->type = BLE_GAP_ADDR_PUBLIC;
                RM_BLE_ABS_StartLegacyAdvertising(&g_ble_abs0_ctrl, &gs_adv_param);
            } break;
        /** some code is omitted */
    }
}
```

Code 40. Prepare for using RPA in the local device (2)

When the local device Advertising or Scan or Connection 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 18.

[Scan]

When setting the scan parameters by *R_BLE_GAP_StartScan* API, configure RPA as its own address type.

[Connection]

When create a connection by *R_BLE_GAP_CreateConn* API, configure RPA as its own address type.

8.4.2 Resolve remote device RPA

RPA of Remote device is resolved according to the following procedures.

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.

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, register a dummy IRK (e.g. All 0x55). *BLE_GAP_EVENT_RSLV_LIST_CONF_COMP* event notifies user application that the registration is complete.

4. **Set Privacy Mode**

   If Network Privacy Mode which is the default is used, the procedure does not need to be done.
   Call *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 1-3 procedures, the BLE Protocol Stack can resolve the remote device RPA included in the received packet and the remote device address included in the event that the application is notified of becomes identity address.

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 42, the GATT database consists of one or more multiple 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 (R01AN5428)*". This chapter describes the profiles and services provided by Renesas and explains APIs for each GATT procedure including examples of how to use them.

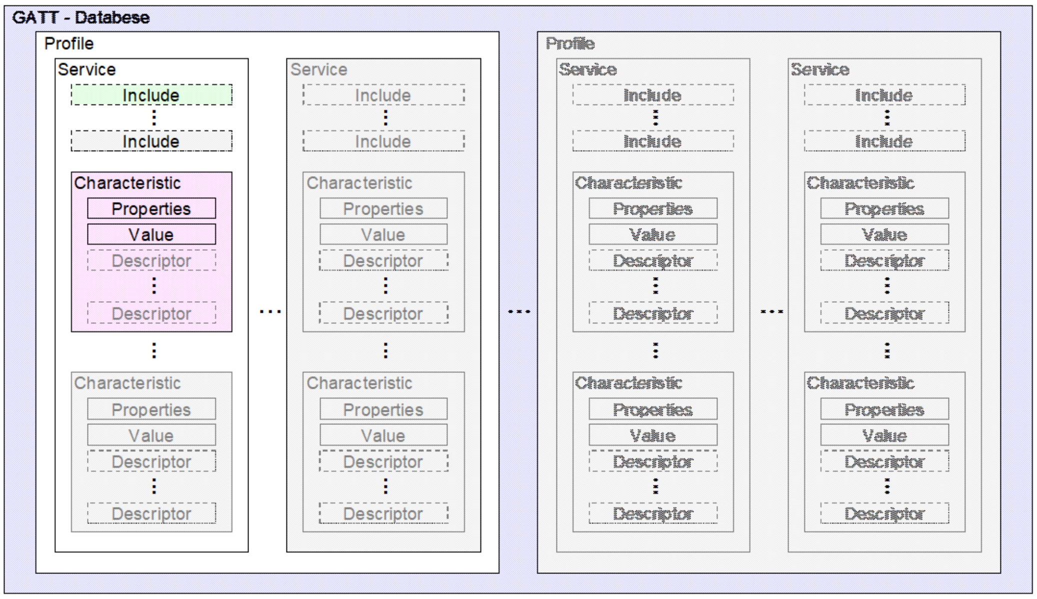


Figure 42. Data structure of GATT database

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 43. Table 44 lists the characteristics that included in each standard service.

Table 43. 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 44. 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
Body Composition Service **BCS**	Body Composition Feature	Read
	Body Composition Measurement	Indicate
Continuous Glucose Monitoring Service **CGMS**	CGM Measurement	Notify
	CGM Feature	Read
	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
Device Information Service	Manufacturer Name String	Read
	Model Number String	Read

Service	**Characteristic**	**GATT Procedure**
DIS	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
	Record Access Control Point	Write, Indicate
Health Thermometer	Temperature Measurement	Indicate

Service	**Characteristic**	**GATT Procedure**
Service **HTS**	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, 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
	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
	Object Last-Modified	Read, Write

Service	Characteristic	GATT Procedure
	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
	Record Access Control Point	Write, Indicate
Reconnection Configuration Service **RCS**	RC Feature	Read
	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
	Resting Heart Rate	Read, Write

Service	Characteristic	GATT Procedure
	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
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 following description, we will use 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 procedure of the GATT client to check the data in the GATT database of the GATT server, as shown in Figure 43. 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.

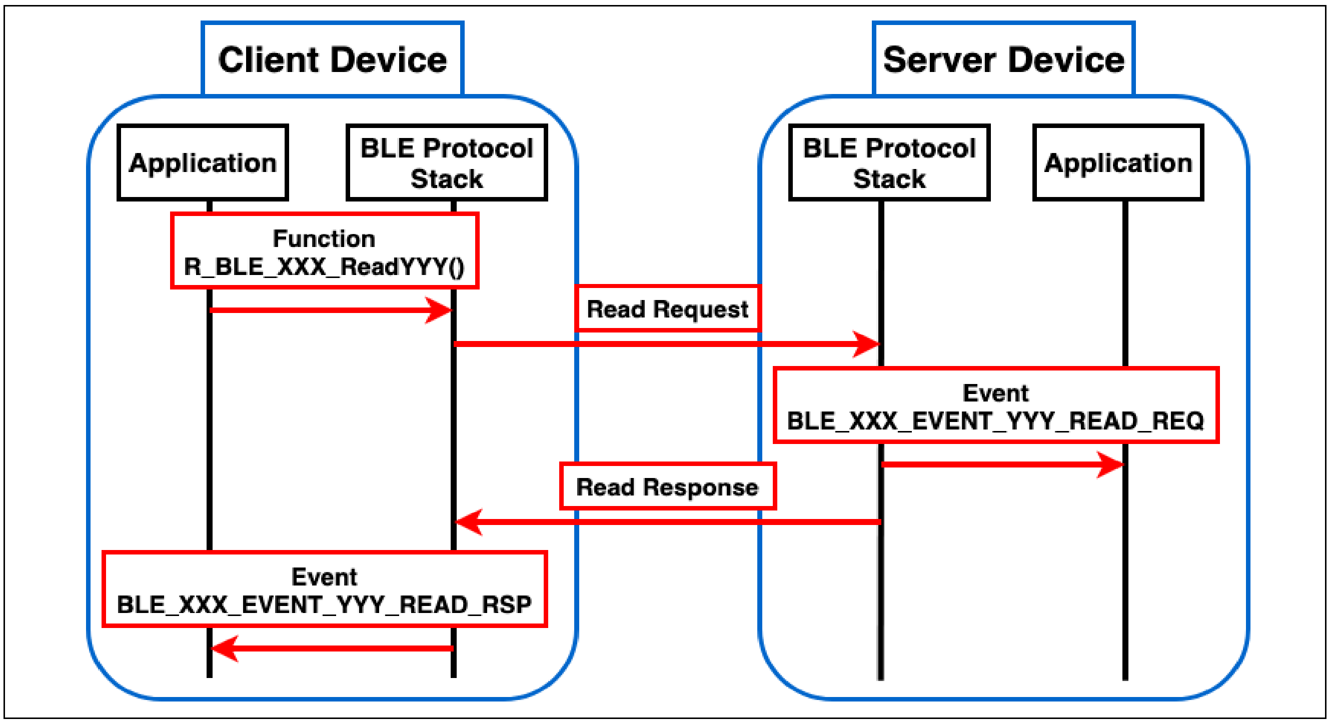


Figure 43. Flow of Read operation

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 44. 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. User can start another operation after this event.

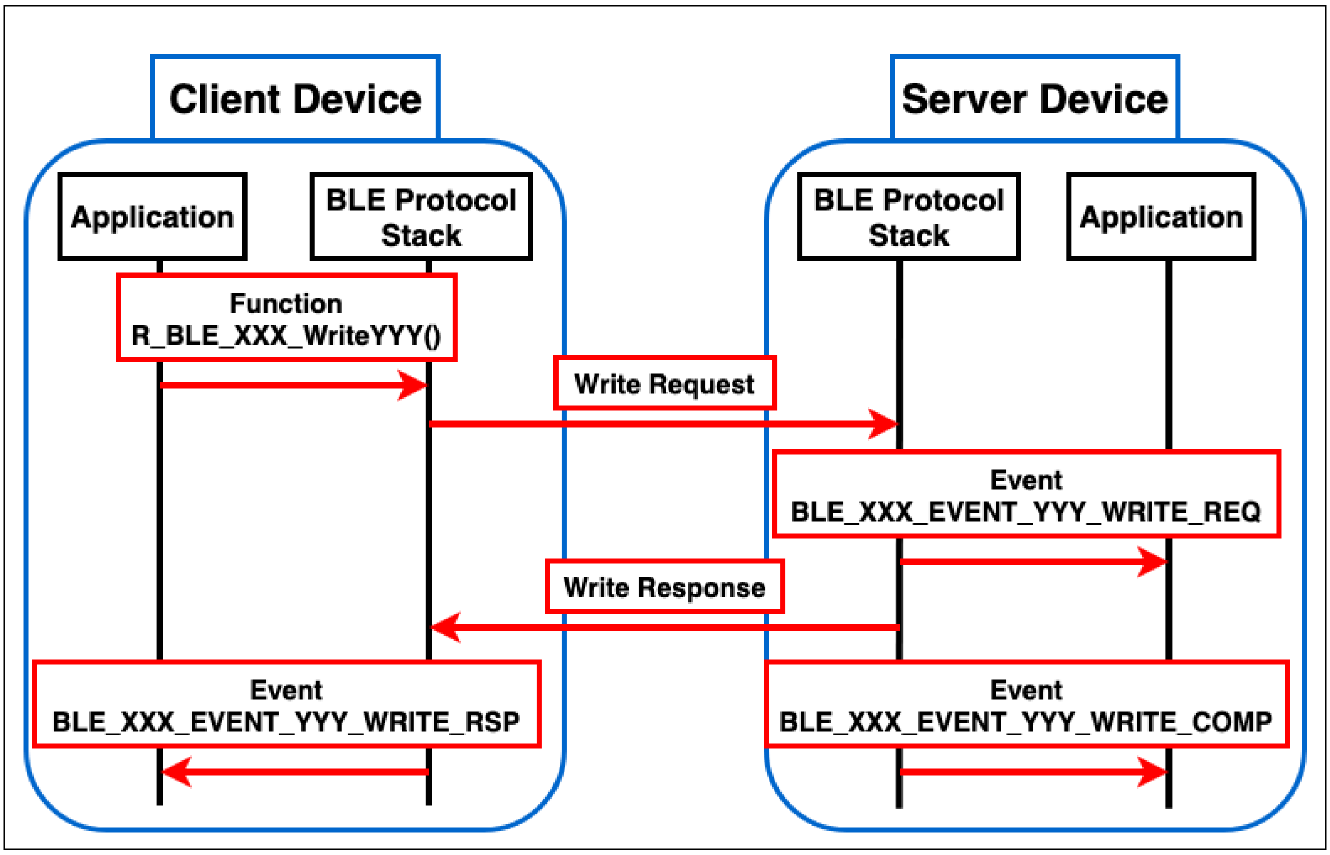

Client Device
Server Device
Application
BLE Protocol Stack
BLE Protocol Stack
Application
Function R_BLE_XXX_WriteYYY()
Write Request
Event BLE_XXX_EVENT_YYY_WRITE_REQ
Write Response
Event BLE_XXX_EVENT_YYY_WRITE_RSP
Event BLE_XXX_EVENT_YYY_WRITE_COMP

Figure 44. Flow of Write operation

9.2.3 WriteWithoutResponse operation

WriteWithoutResponse operation is procedure to change the GATT database of the GATT server by sending data from the GATT client, as shown in Figure 45. 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 user wants to reduce power consumption on user’s device, or when 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:
User can send “Write Command” by using the function *R_BLE_XXX_WriteWithoutResponseYYY* API. WriteWithoutResponse operation is completed when call *R_BLE_XXX_WriteWithoutResponseYYY* API. User can start another operation after calling the API.

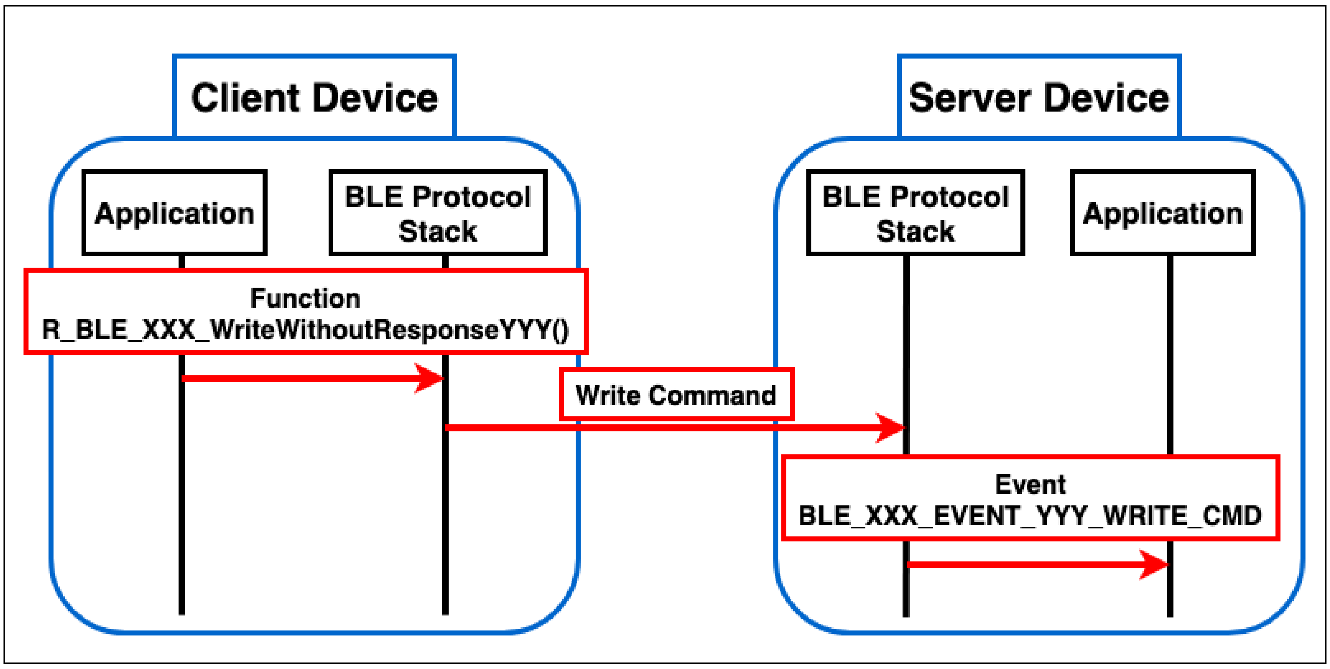


Figure 45. Flow of WriteWithoutResponse operation

9.2.4 Notification operation

Notification operation is procedure to send data from GATT server to GATT client, as shown in Figure 46. For Notification operation, the CCCD must have been added as 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 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. User 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.

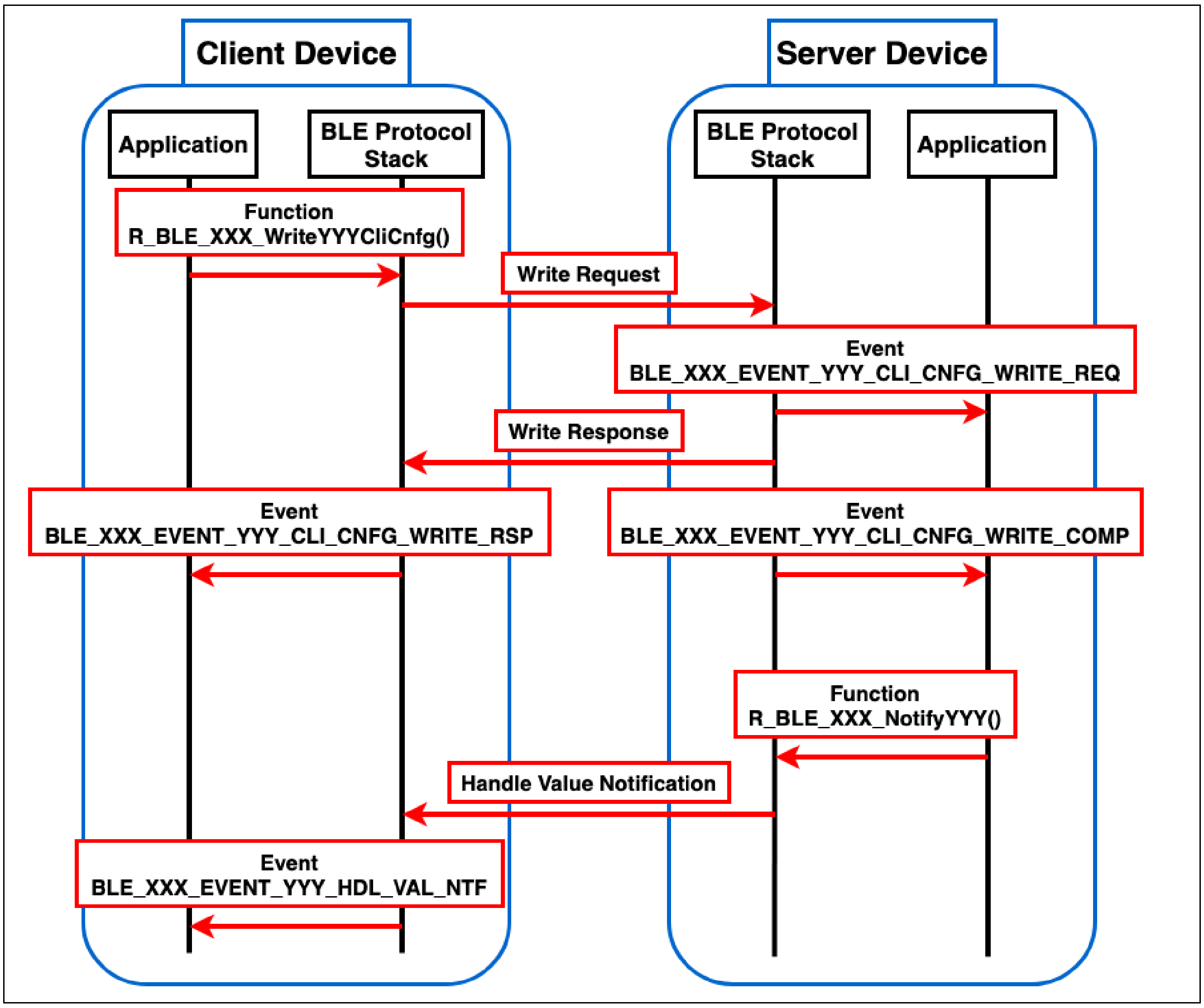

Client Device
Server Device
Application
BLE Protocol Stack
BLE Protocol Stack
Application
Function R_BLE_XXX_WriteYYYCliCnfg()
Write Request
Event BLE_XXX_EVENT_YYY_CLI_CNFG_WRITE_REQ
Write Response
Event BLE_XXX_EVENT_YYY_CLI_CNFG_WRITE_RSP
Event BLE_XXX_EVENT_YYY_CLI_CNFG_WRITE_COMP
Function R_BLE_XXX_NotifyYYY()
Handle Value Notification
Event BLE_XXX_EVENT_YYY_HDL_VAL_NTF

Figure 46. Flow of Notification operation

9.2.5 Indication operation

Indication operation is procedure to send data from GATT server to GATT client, as shown in Figure 47. For Indication operation, the CCCD must have been added as 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 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.

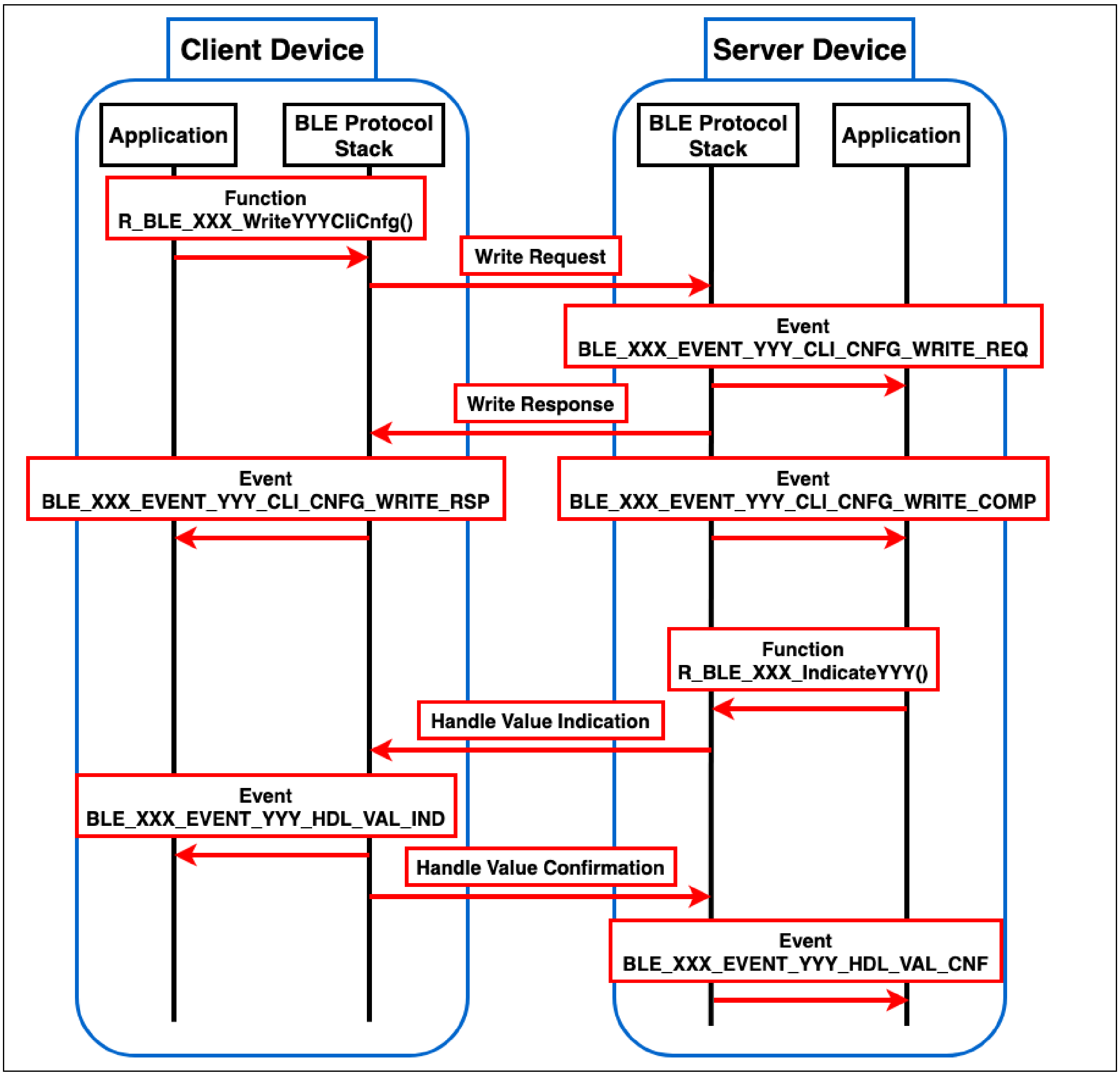

Client Device
Server Device
Application
BLE Protocol Stack
BLE Protocol Stack
Application
Function R_BLE_XXX_WriteYYYCliCnfg()
Write Request
Event BLE_XXX_EVENT_YYY_CLI_CNFG_WRITE_REQ
Write Response
Event BLE_XXX_EVENT_YYY_CLI_CNFG_WRITE_RSP
Event BLE_XXX_EVENT_YYY_CLI_CNFG_WRITE_COMP
Function R_BLE_XXX_IndicateYYY()
Handle Value Indication
Event BLE_XXX_EVENT_YYY_HDL_VAL_IND
Handle Value Confirmation
Event BLE_XXX_EVENT_YYY_HDL_VAL_CNF

Figure 47. Flow of Indication operation

9.2.6 ReliableWrite operation

The ReliableWrite operation is procedure to send data from GATT client to GATT server, ensure that the correct values are written, and then reflect it in the GATT database, as shown in Figure 48. 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 to 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:
User can send “Prepare Write Request” using *R_BLE_GATTC_ReliableWrites* API. User can receive “Prepare Write Response” for each data transmitted, and user 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.

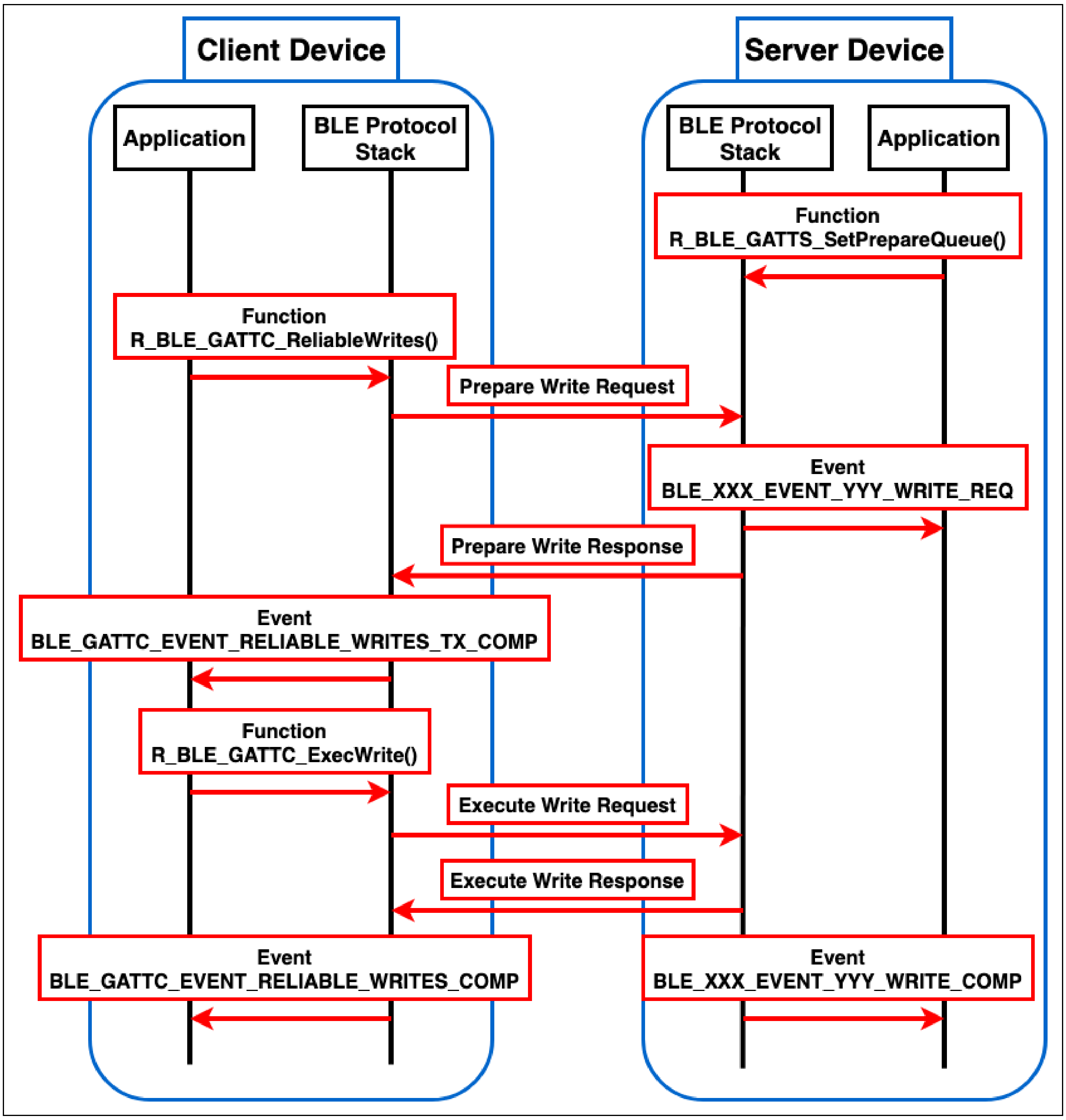

Client Device
Application
BLE Protocol Stack
Server Device
BLE Protocol Stack
Application
Function R_BLE_GATTS_SetPrepareQueue()
Function R_BLE_GATTC_ReliableWrites()
Prepare Write Request
Event BLE_XXX_EVENT_YYY_WRITE_REQ
Prepare Write Response
Event BLE_GATTC_EVENT_RELIABLE_WRITES_TX_COMP
Function R_BLE_GATTC_ExecWrite()
Execute Write Request
Execute Write Response
Event BLE_GATTC_EVENT_RELIABLE_WRITES_COMP
Event BLE_XXX_EVENT_YYY_WRITE_COMP

Figure 48. Flow of ReliableWrite operation

9.2.7 Broadcast Operation

Broadcast operation is procedure for transmitting data without connection to an unspecified number of devices, as shown in Figure 49. 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 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 have been 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 15.
- For Advertising Data configuration (section 4.4), user 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 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.

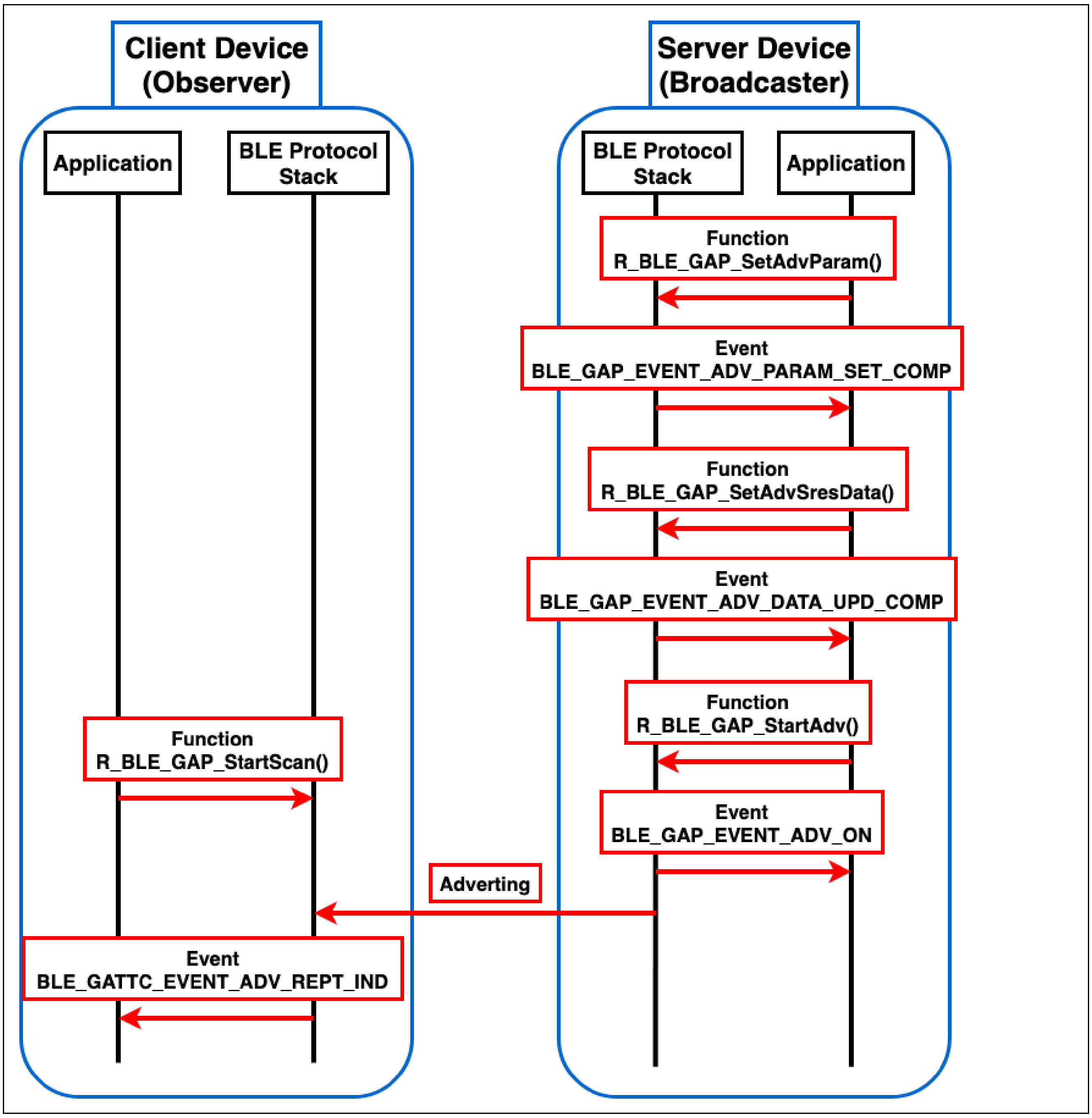

Client Device (Observer)
Application
BLE Protocol Stack
Server Device (Broadcaster)
BLE Protocol Stack
Application
Function R_BLE_GAP_SetAdvParam()
Event BLE_GAP_EVENT_ADV_PARAM_SET_COMP
Function R_BLE_GAP_SetAdvSresData()
Event BLE_GAP_EVENT_ADV_DATA_UPD_COMP
Function R_BLE_GAP_StartAdv()
Function R_BLE_GAP_StartScan()
Event BLE_GAP_EVENT_ADV_ON
Adverting
Event BLE_GATTC_EVENT_ADV_REPT_IND

Figure 49. Flow of Broadcast operation

9.3 Example of using GATT Procedure

Refer to *BLE sample application (R01AN5402)*.

Revision History

Rev.	Date	Description	
		Page	**Summary**
1.00	Jan.13.2021	–	First edition issued.
1.01	Aug.31.2021	–	• Add ROM/RAM usage for extended/balance/compact configuration to section 2.4.

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.

Corporate Headquarters

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

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/.

Trademarks