Bluetooth LE microprocessor / module

Bluetooth Low Energy Profile Developer's Guide

Introduction

This document guides developers of Bluetooth LE profiles for the following devices how to develop profiles using the Bluetooth LE development tool QE for BLE.

This document is a re-edited version of the following documents integrated.

RA4W1 Group Bluetooth Low Energy Profile Developer's Guide (R01AN5428)

RE01B Group Bluetooth Low Energy Profile Developer's Guide (R01AN5638)

RX23W Group Bluetooth Low Energy Profile Developer's Guide (R01AN4553)

Target Device

- RX23W Group
- RA4W1 Group
- RE01B Group

Related Documents

- Bluetooth Core Specification 【https://www.bluetooth.com】
- Core Specification Supplement 【https://www.bluetooth.com】
- Bluetooth® Security and Privacy Best Practices Guide 【https://www.bluetooth.com】

- RX23W Group User's Manual Hardware Edition (R01UH0823)
- RX23W Group BLE Module Firmware Integration Technology (R01AN4860)
- Bluetooth Low Energy Protocol Stack Basic Package User's Manual (R01UW0205)
- RX23W Group Bluetooth Low Energy Application Developer's Guide (R01AN5504)

- RA4W1 Group User’s Manual: Hardware (R01UH0883)
- RA Flexible Software Package Documentation
- RA4W1 Group BLE Sample Application (R01AN5402)
- RA4W1 Group Bluetooth Low Energy Application Developer’s Guide (R01AN5653)

- RE01B Group User's Manual Hardware Edition (R01UH0903)
- RE01B Group Development Startup Guide Using CMSIS Package (R01AN5310)
- RE01B Group Bluetooth Low Energy Sample Code (Using CMSIS Driver Package) (R01AN5606)
- RE01B Group Bluetooth Low Energy Application Developer's Guide (R01AN5643)

- e² studio User's Manual Getting Started Guide (R20UT4204)
- QE for BLE [RA, RE, RX] V1.5.0 Release Notes (R20UT5145)

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

Contents

1. Overview 4
1.1 Overview of Bluetooth LE Data Communication 4
1.2 Bluetooth LE program development environment 5
1.2.1 e² studio 5
1.2.2 QE for BLE 6
1.2.3 Bluetooth LE communication project 8
1.3 Software structure of the profile program 9

2. Building a development environment 11
2.1 Installing QE for BLE 11
2.1.1 How to add QE for BLE to an installed e² studio 11
2.1.2 How to add QE for BLE when installing e² studio 12
2.2 Getting of Bluetooth LE Communication Project 13
2.2.1 RX23W 13
2.2.2 RA4W1 13
2.2.3 RE01B 13

3. Profile development with QE for BLE 14
3.1 How to Use QE for BLE 14
3.2 Design of the profile 16
3.2.1 Application role settings 17
3.2.2 Adding and configuration service 18
3.2.3 Adding and configuration characteristic 23
3.2.4 Adding and configuration descriptor 26
3.3 Configuration of Peripheral 28
3.3.1 Advertising Data 28
3.3.2 Scan Response Data 30
3.3.3 Advertising Parameter 30
3.4 Configuration of Central 31
3.4.1 Scan Parameter 31
3.4.2 Scan Filter Data 32
3.4.3 Connection Parameter 33

4. Implementation of program 34
4.1 Service API Programs (r_ble_xxs.c / r_ble_xxc.c) 37
4.1.1 Description of encode/decode functions 40
4.1.2 Automatic generation of encode/decode functions 42
4.1.3 Implementing the encode-decode function 45
4.2 Application Framework (app_main.c) 47
4.2.1 Responding to security requirements 49
4.2.1.1 When set to Security Level 3 49
4.2.1.2 When set to Security Level 4 49
4.2.2 Exchange MTU 50
4.2.2.1 Implementation of Client 51
4.2.2.2 Implementation of Server 51

4.2.3 Write Operation 52
4.2.3.1 Implementation of Client 54
4.2.3.2 Implementation of Server 56
4.2.4 Write Without Response Operation 58
4.2.4.1 Implementation of Client 59
4.2.4.2 Implementation of Server 60
4.2.5 Read Operation 61
4.2.5.1 Implementation of Client 63
4.2.5.2 Implementation of Server 65
4.2.6 Notify Operation 66
4.2.6.1 Implementation of Client 67
4.2.6.2 Implementation of Server 68
4.2.7 Indicate Operation 69
4.2.7.1 Implementation of Client 70
4.2.7.2 Implementation of Server 71
4.3 GATT Database (gatt_db.c / gatt_db.h) 73

5. Build and Run program 75
5.1 RX23W 75
5.1.1 Migrating Profile Data due to Unifying a Plug-in 75
5.2 RA4W1 78
5.3 RE01B 79

6. Notice 80
6.1 Implementation of multiple services 80
6.2 Implementation of same service 80
6.3 Implementation of secondary service 82
6.4 Implementation of discovery operation about included service 86
6.5 Guide for Connection Update 88
6.6 Settings for connecting two MCUs for data communication 89
6.7 When using old version qualifications (QDID:134484) 94
6.7.1 QE for BLE generation code change setting 94
6.7.2 Get profile common library 96

Revision History 97

1. Overview

1.1 Overview of Bluetooth LE Data Communication

In Bluetooth LE communication, Generic Attribute Protocol (GATT) is primarily used. GATT communicates in a client-server architecture.

The communication protocol of an application that uses GATT Protocol is called a profile. Profiles are protocols developed for many applications. Profile communication is allowed between devices that supports same profile. A profile has one or more "Service" that represent the functionality of the application. A service consists of "Characteristic" that represent data structures and "Descriptor" that add information to the data in the characteristics.

The server has a GATT database that manages data for the service. Application data is held in the GATT database as a characteristic of service. The GATT database is accessed by specifying an attribute handle that indicates where the data is stored.

GATT defines Notify / Indicate operations for sending data from the server to the client and Read / Write operations for reading and writing the database by the client.

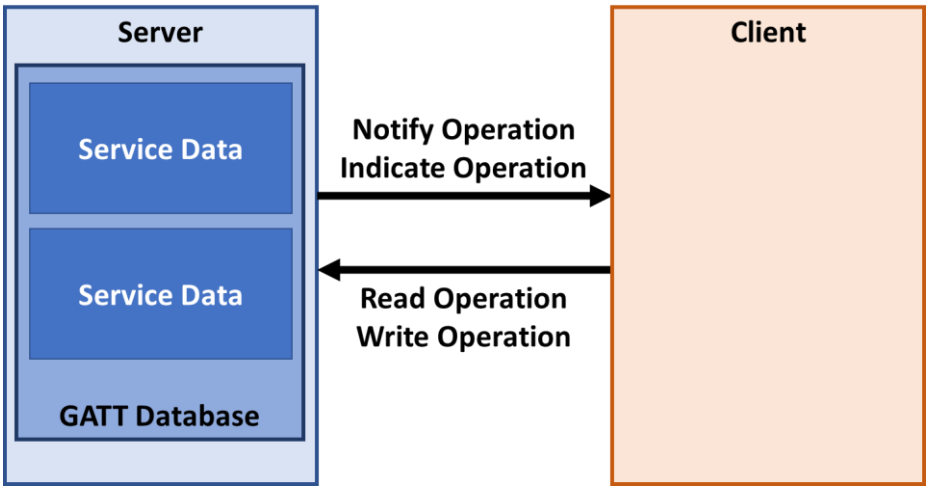


Figure 1.1 Overview of profile communication

Profile communication is established between peer-to-peer connected devices. A connection is established between a device (peripheral) that performs advertising operations and a device (central) that performs scan operations and connection operations.

1.2 Bluetooth LE program development environment

Renesas Electronics provides a Bluetooth LE profile development environment to support Bluetooth LE application development.

Figure 1.2, Figure 1.3 show the profile development environment. QE for BLE creates a program that realizes GATT profile communication. The generated program runs on the Bluetooth LE communication project.

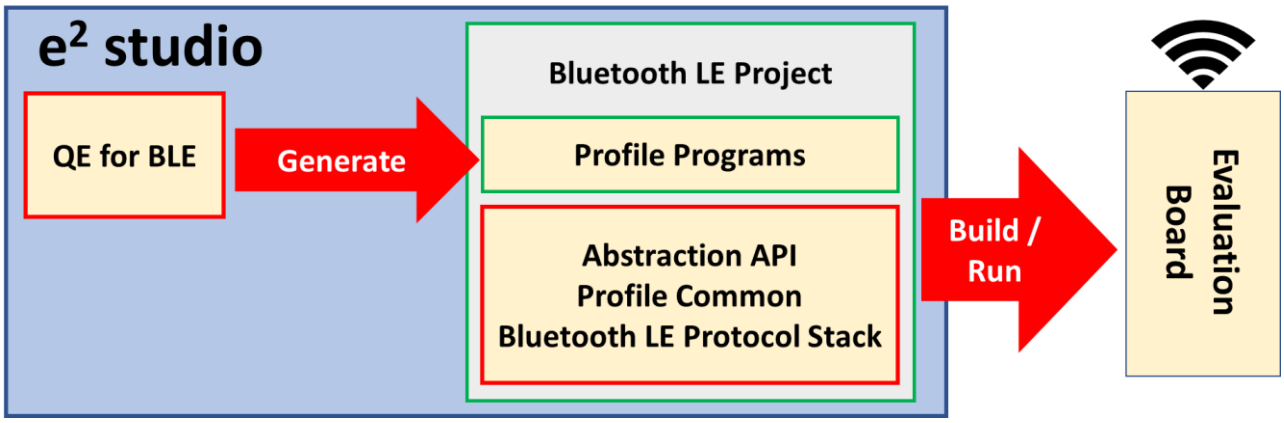


Figure 1.2 Profile Development Environment by QE for BLE (RE01B)

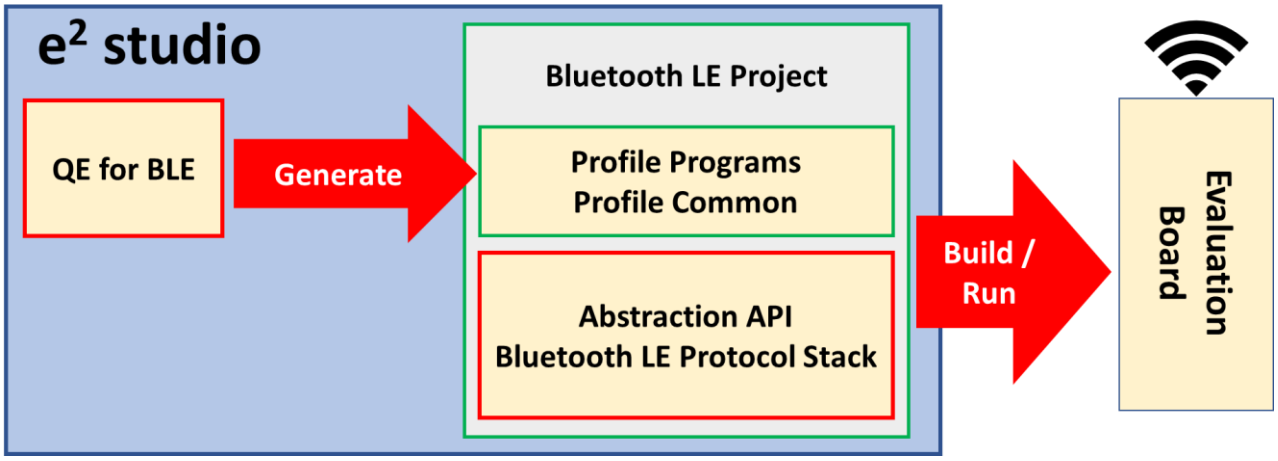


Figure 1.3 Profile Development Environment by QE for BLE (RX23W, RA4W1)

Profile development uses e² studio, QE for BLE, and Bluetooth LE communication project.

1.2.1 e² studio

The e² studio is an integrated development environment (IDE) for Renesas MCUs. In addition to code editor, the e² studio offers a rich range of extended functions. The e² studio covers all development processes, from the downloading of sample code to debugging.

e² studio | Renesas

1.2.2 QE for BLE

QE for BLE is a tool to design profiles on the GUI. The designed profiles are generated as source code. This product is provided as an extension to e^2 studio.

The Bluetooth specification defines several services by the Bluetooth SIG, which are referred to in this document as SIG adopted services. On the other hand, users can create their own services to achieve features that are not supported by the SIG adopted service. In this document, a service that you define yourself is called a custom service.

QE for BLE supports the SIG adopted services listed in Table 1.1. Many of these are certified. Table 1.2 also lists the profiles supported by QE for BLE. Specifications of each service is defined by Bluetooth SIG. Check Web page of Bluetooth SIG (https://www.bluetooth.com) for more information.

Table 1.1 SIG adopted service supported by QE for BLE

Service Name	Abbr	Version	Service Name	Abbr	Version
Alert Notification Service	ANS	1.0	Automation IO Service	AIOS	1.0
Battery Service	BAS	1.0	Blood Pressure Service	BLS	1.1.1
Body Composition Service	BCS	1.0	Bond Management Service	BMS	1.0.1
Continuous Glucose Monitoring Service	CGMS	1.0.2	Current Time Service	CTS	1.1
Cycling Power Service	CPS	1.1	Cycling Speed and Cadence Service	CSCS	1.0
Device Information Service	DIS	1.1	Environmental Sensing Service	ESS	1.0
Fitness Machine Service	FTMS	1.0	Glucose Service	GLS	1.0.1
Health Thermometer Service	HTS	1.0	Heart Rate Service	HRS	1.0
Human Interface Device Service	HIDS	1.0	Immediate Alert Service	IAS	1.0
Insulin Delivery Service	IDS	1.0.1	Link Loss Service	LLS	1.0.1
Location and Navigation Service	LNS	1.0	Next DST Change Service	NDCS	1.0
Object Transfer Service	OTS	1.0	Phone Alert Status Service	PASS	1.0
Pulse Oximeter Service	PLXS	1.0.1	Reconnection Configuration Service	RCS	1.0.1
Reference Time Update Service	RTUS	1.0	Running Speed and Cadence Service	RSCS	1.0
Scan Parameters Service	ScPS	1.0	Tx Power Service	TPS	1.0
User Data Service	UDS	1.1	Weight Scale Service	WSS	1.0
GATT Service	GATS	-	GAP Service	GAPS	-

Note: Object Transfer Service is not qualified by Bluetooth SIG.

Table 1.2 Profile supported by QE for BLE

Profile Name [Abbr]	Version	Services that configure profile			
Alert Notification Profile [ANP]	1.0	ANS			
Automation IO Profile [AIOP]	1.0	AIOS			
Blood Pressure Profile [BLP]	1.1.1	BLS	DIS		
Continuous Glucose Monitoring Profile [CGMP]	1.0.2	CGMS	DIS	(BMS)	
Cycling Power Profile [CPP]	1.1	CPS	(DIS)	(BAS)	
Cycling Speed and Cadence Profile [CSCP]	1.0	CSCS	(DIS)		
Environmental Sensing Profile [ESP]	1.0	ESS	(DIS)	(BAS)	
Find Me Profile [FMP]	1.0	IAS			
Fitness Machine Profile [FTMP]	1.0	FTMS	(DIS)	(UDS)	
Glucose Profile [GLP]	1.0.1	GLS	DIS		
Health Thermometer Profile [HTP]	1.0	HTS	DIS		
Heart Rate Profile [HRP]	1.0	HRS	DIS		
HID over GATT Profile [HOGP]	1.0	HIDS	DIS	BAS	(ScPS)
Insulin Delivery Profile [IDP]	1.0.1	IDS	DIS	(BAS)	(CTS)
		(BMS)	(IAS)		
Location and Navigation Profile [LNP]	1.0	LNS	(DIS)	(BAS)	
Phone Alert Status Profile [PASP]	1.0	PASS			
Proximity Profile [PXP]	1.0.1	IAS	(LLS)	(TPS)	
Pulse Oximeter Profile [PLXP]	1.0.1	PLXS	DIS	(BAS)	(CTS)
		(BMS)			
Reconnection Configuration Profile [RCP]	1.0.1	RCS	(BMS)		
Running Speed and Cadence Profile [RSCP]	1.0	RSCS	(DIS)		
Scan Parameters Profile [ScPP]	1.0	ScPS			
Time Profile [TIP]	1.0	CTS	(NDCS)	(RTUS)	
Weight Scale Profile [WSP]	1.0	WSS	DIS	(BCS)	(BAS)
		(CTS)	(UDS)		

Note: Services without () are mandatory services, and services with () are Optional services. If you add a profile in QE for BLE, only mandatory services are added to profile tree.

1.2.3 Bluetooth LE communication project

The source code generated by QE for BLE runs on the following program. For information on how to obtain each program, see "2.2 Obtaining Bluetooth LE Communications Projects".

1. Bluetooth LE Protocol Stack

   Bluetooth LE Protocol Stack is a program to realize the Bluetooth LE function. The Bluetooth LE Protocol Stack is provided as a static library. R_BLE_GATTC_XXX API and R_BLE_GATTS_XXX API are used from profile common library.

2. Profile Common Library

   The profile common library are programs that summarizes the common processing part of data communication by the profile. The common profile section runs on the Bluetooth LE Protocol Stack.

   R_BLE_DISC_XXX API, R_BLE_SERVC_XXX API, R_BLE_SERVS_XXX API are provided.

3. Abstraction API

4. The abstraction API is a program for simply implementing procedures related to connection and security in Bluetooth LE communication. It summarizes the functions often used in Bluetooth LE communication. R_BLE_ABS_XXX APIs are provided.

1.3 Software structure of the profile program

Figure 1.4, Figure 1.5 show the software configuration of the program that realizes the profile generated by the code generation function of QE for BLE.

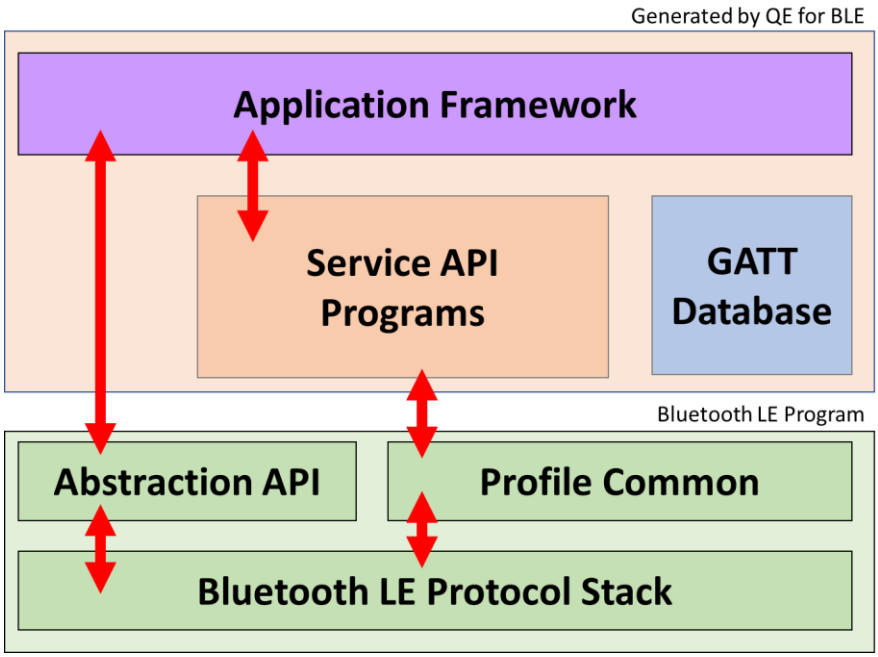


Figure 1.4 Programs generated by QE for BLE (RE01B)

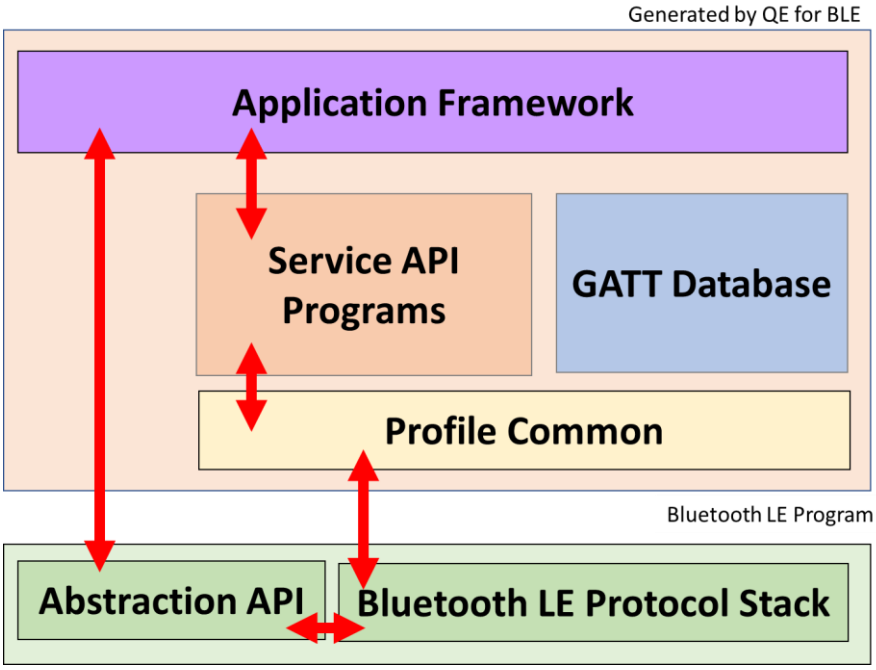


Figure 1.5 Programs generated by QE for BLE (RX23W, RA4W1)

The profile program generated from QE for BLE uses the profile common library to realize Bluetooth LE communication. The profile program consists of the following three programs. For details on how to generate profile programs, please refer to [3 Profile development with QE for BLE].

1. Application Framework

   This is a framework for using Bluetooth LE functions and profiles. User applications are implemented using the service API based on this framework. See "4.2 Application Framework (app_main.c)" for details.

2. Service API Programs

   This is an API program to access the data of services defined in the GATT database. See "4.1 Service API Programs (r_ble_xxs.c / r_ble_xxc.c)" for details.

3. GATT Database Program

   This is an implementation of the GATT database that reflects the data structure of the service. See "4.3 GATT Database (gatt_db.c / gatt_db.h)" for details.

2. Building a development environment

This chapter describes how to install QE for BLE and how to add a Bluetooth LE communication project to the e² studio workspace.

2.1 Installing QE for BLE

2.1.1 How to add QE for BLE to an installed e² studio

QE for BLE can be downloaded from the following pages.

- https://www.renesas.com/qe-ble

Install method is as follows:

1. Activate e² studio.
2. Select [Renesas Views] → [Renesas Software Installer] menu to open the [Renesas Software Installer] dialog.
3. Select [Renesas QE] and click the [Next>] button.
4. Check the [QE for BLE [RA, RE, RX]] check box and click the [Finish (F)] button.
5. In the [Install] dialog, make sure that the [Renesas QE for BLE [RA, RE, RX]] check box and the [Renesas QE for BLE [RA, RE, RX] Utility] check box are checked, and then click [Click the Next>] button.
6. Confirm that the installation targets are [Renesas QE for BLE [RA, RE, RX]] and [Renesas QE for BLE [RA, RE, RX] Utility], and then [Next (N)>]. Press the button.
7. After confirming the license, if you agree to the license, select the [I accept the terms of the terms of use (A)] radio button and click the [Exit (F)] button.
8. If the dialog for selecting a trusted certificate is displayed, check the displayed certificate, and then click the [OK] button to continue the installation.
9. Restart e²studio.

2.1.2 How to add QE for BLE when installing e² studio

QE for BLE can be installed with e² studio.

Check QE for BLE [RA, RE, RX] in the “Additional Software” selection screen of the e² studio installation wizard.

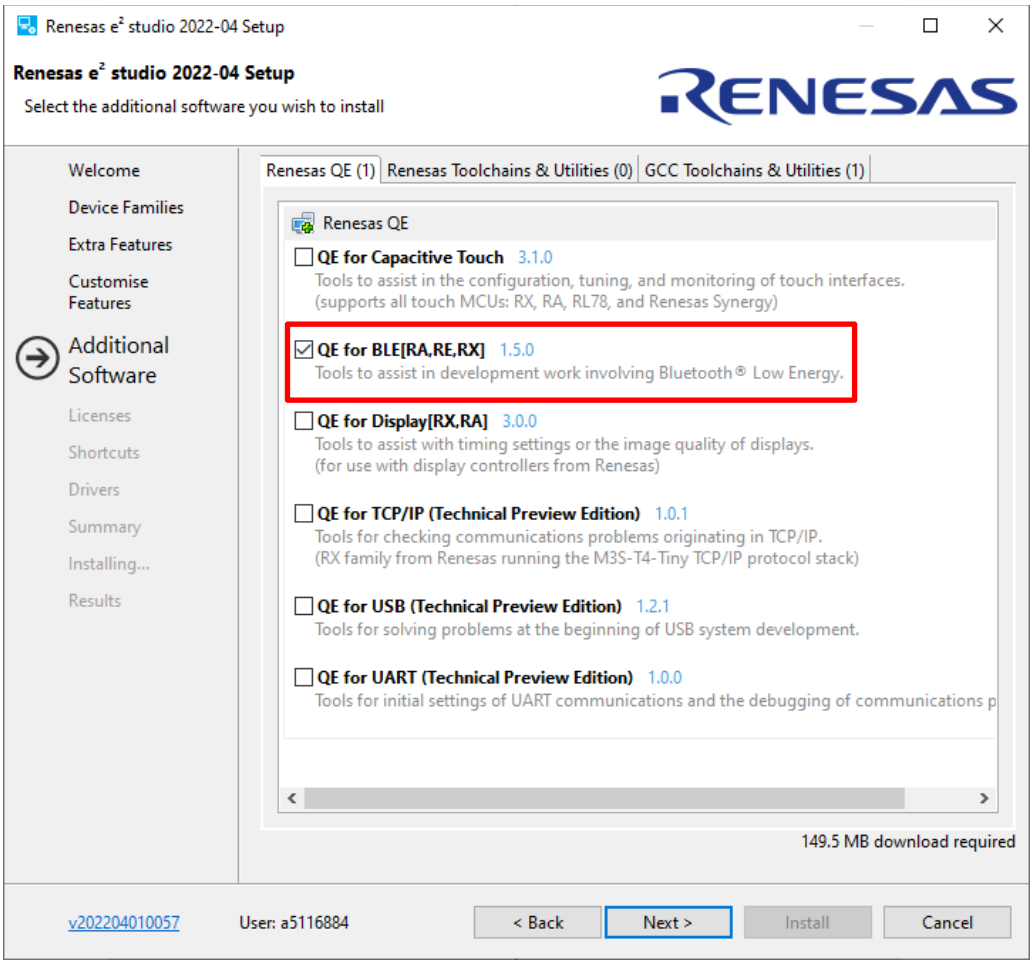


Figure 2.1 Installing QE for BLE

2.2 Getting of Bluetooth LE Communication Project

2.2.1 RX23W

For the RX23W, the Bluetooth LE Protocol Stack and abstraction API are provided in FIT format.

Refer to Chapter 4 "BLE FIT Module Project" in the following document to add the Bluetooth LE communication project to your e^2 studio workspace.

- RX23W Group BLE Module Firmware Integration Technology (R01AN4860)

Profile common library is generated from QE for BLE when using QE for BLE 1.60 or later and BLE FIT module 2.50 or later. BLE FIT module 2.40 or earlier includes the profile common library.

2.2.2 RA4W1

For RA4W1, the Bluetooth LE Protocol Stack and abstraction API are provided as modules of the FSP. The profile common library is generated from QE for BLE.

Refer to Chapter 2 "How to use demo project" in the following document to add a Bluetooth LE communication project to your e^2 studio workspace.

- RA4W1 Group BLE Sample Application (R01AN5402)

2.2.3 RE01B

For RE01B, the Bluetooth LE Protocol Stack, abstraction API and profile common library are provided as sample projects in the Application Notes.

Refer to Chapter 4, "Creating Projects" in the following document to add the Bluetooth LE communication project to your e^2 studio workspace.

- RE01B Group Bluetooth Low Energy Sample Code (Using CMSIS Driver Package) (R01AN5606)

3. Profile development with QE for BLE

This chapter describes how to design QE for BLE profiles. QE for BLE can be used to design GATT profiles as well as configure GAP roles and parameters for Bluetooth LE.

3.1 How to Use QE for BLE

Launch QE for BLE by selecting [Renesas view]→[Renesas QE]→[R_BLE Custom Profile RA, RE, RX (QE)] in menu of the e² studio.

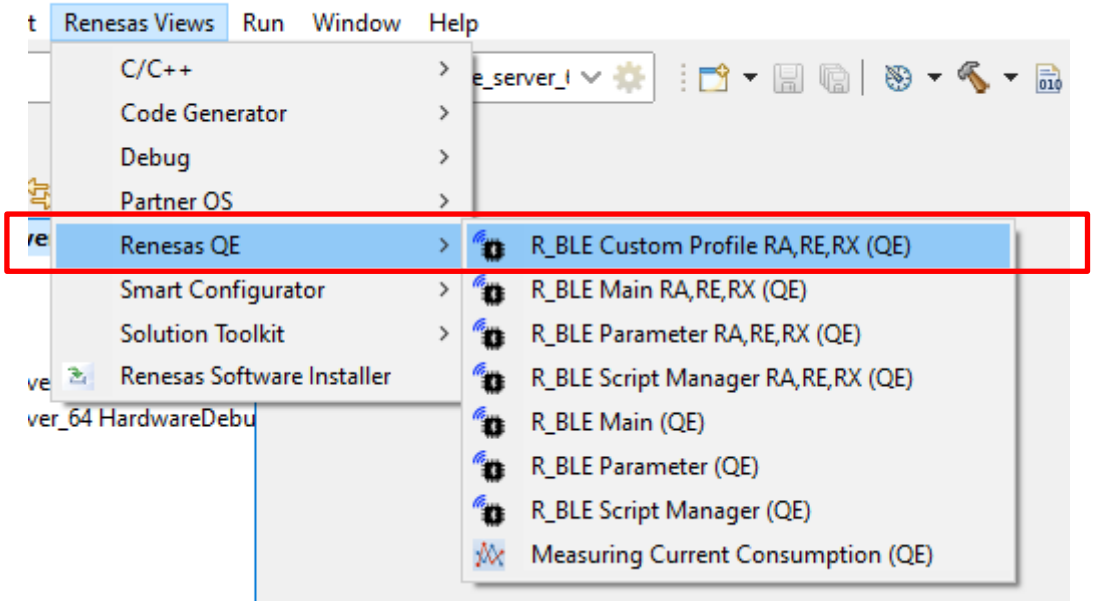


Figure 3.1 Open QE for BLE

Note: If your project contains an older version of QE for BLE, you will be prompted to migrate to the latest QE for BLE.

- QE for BLE [RA, RE, RX] V1.5.0 release notes

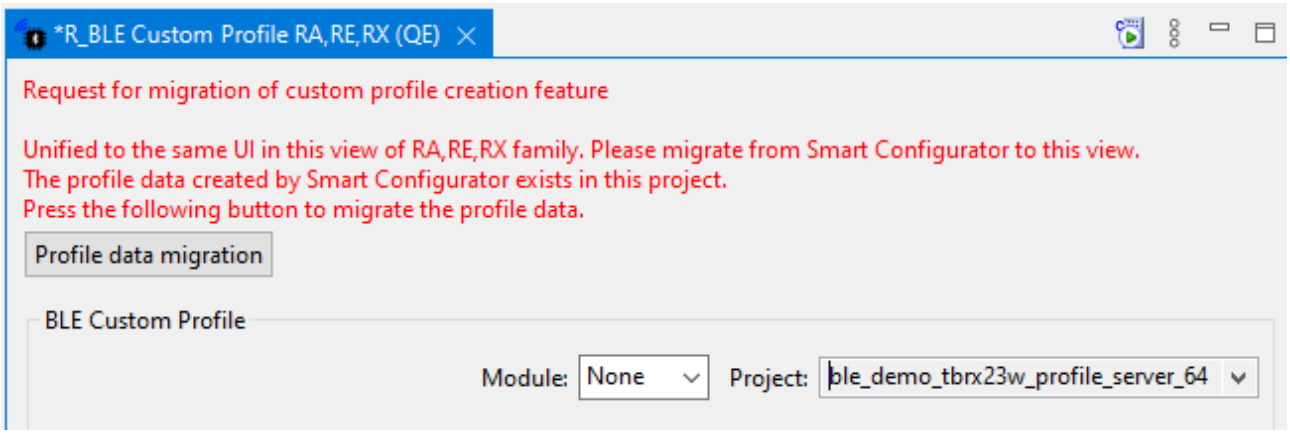


Figure 3.2 Profile updates when using older projects

From the project selection field in the upper right, select the project to which you want to add code.

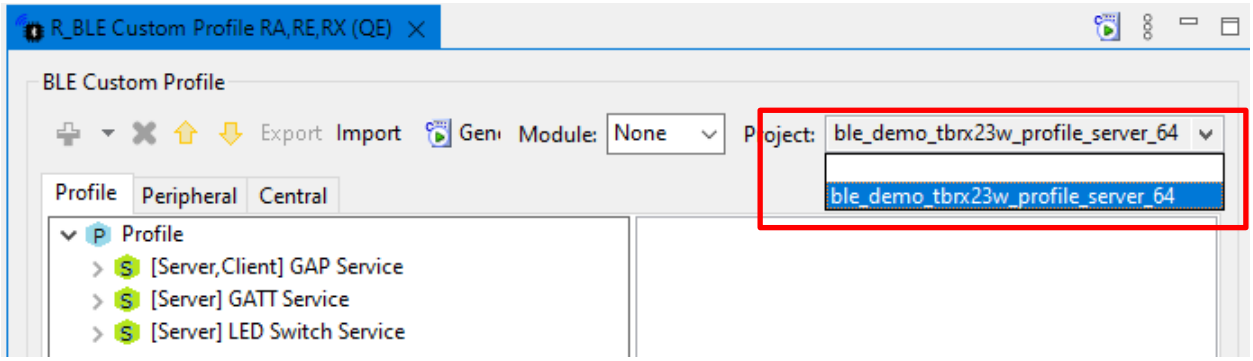


Figure 3.3 Select project

3.2 Design of the profile

This section describes how to design Bluetooth LE profiles used by applications.

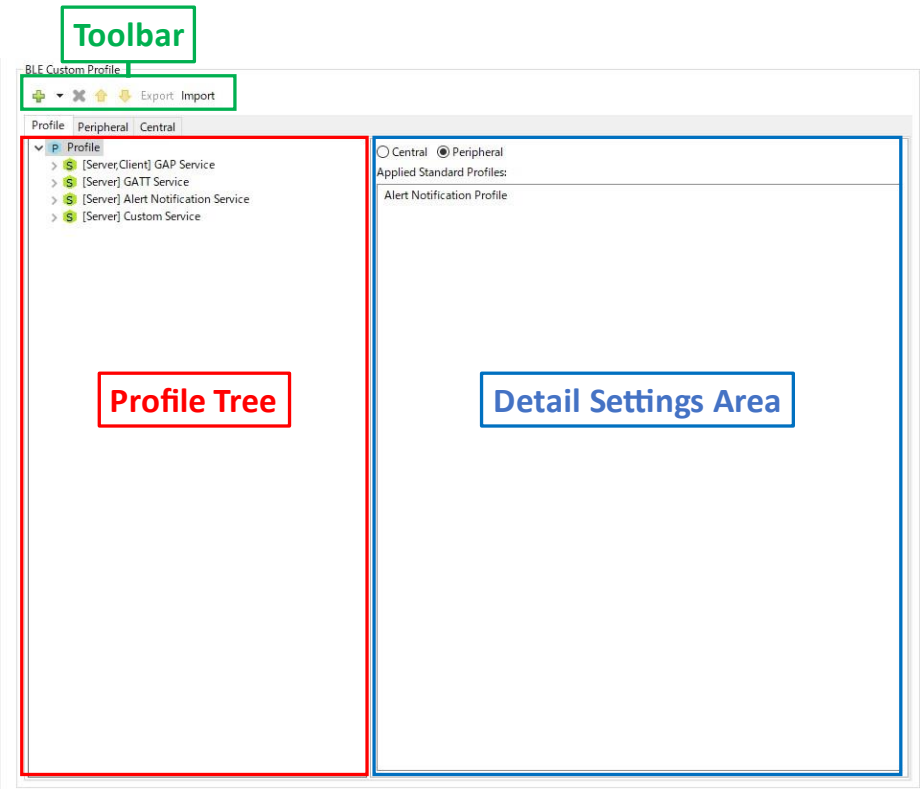


Figure 3.4 QE for BLE configuration screen

The data handled by the profile has a tree structure that consists of services, characteristic, and descriptors. This profile tree shows the data structure of the entire profile being designed.

When an element in the profile tree is selected, the set items of the selected element are displayed in the advanced settings area. In the Advanced Settings area, you can design the selected element.

Add/delete an element from the Profile tree from the toolbar. The icons and actions of the toolbars are shown below.

「」:Add the selected element.

「」:This feature deletes selected elements.

「」「」:Moves the selected element.

Export/Import button enable you to save/load the services selected in the Profile tree.

3.2.1 Application role settings

This section selects the GAP role to be used by the application. Select the profile [ ] in the profile tree and display the profile settings in the detail setting area.

Select the GAP role for the program to be generated. A skeleton program for the role selected here will be generated. In the case of Peripheral, the program to advertise is generated. Selecting "Central" generates a program that issues a scan and connection request.

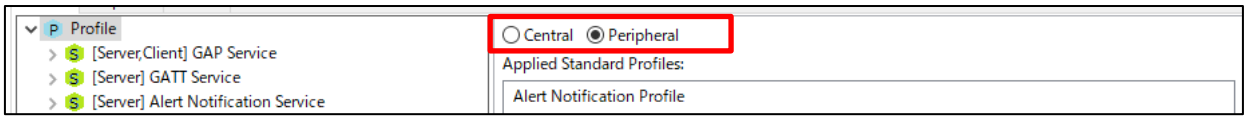


Figure 3.5 Selecting the GAP Role of an Application

The Profile name shown in Table 1.2 is displayed in the Detail Settings area depending on the combination of services to be added.

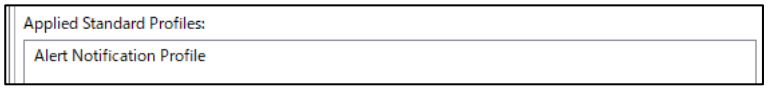


Figure 3.6 Confirmation of Profile screen

3.2.2 Adding and configuration service

With the profile [ ] selected, press the toolbar [ ] to add service to the profile.

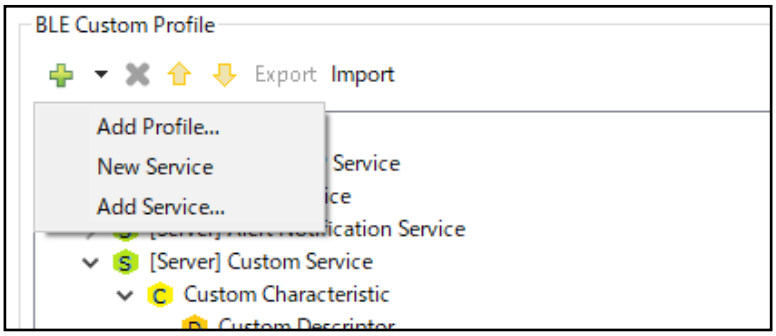


Figure 3.7 Adding service

Select [New Service] to add custom service, or [Add Service] to SIG adopted service. To create a profile defined by the Bluetooth SIG, select "Add Profile" and add the required SIG adopted services. If you need Optional SIG adopted services, add them individually from [Add Service].

Select Service [ ] in the Profile tree to display the service settings in the Detail Settings area. Figure 3.7 shows the setting items. Table 3.1 shows the descriptions of each setting item.

[Note] GAP Service and GATT Service are mandatory services. Do not delete it.

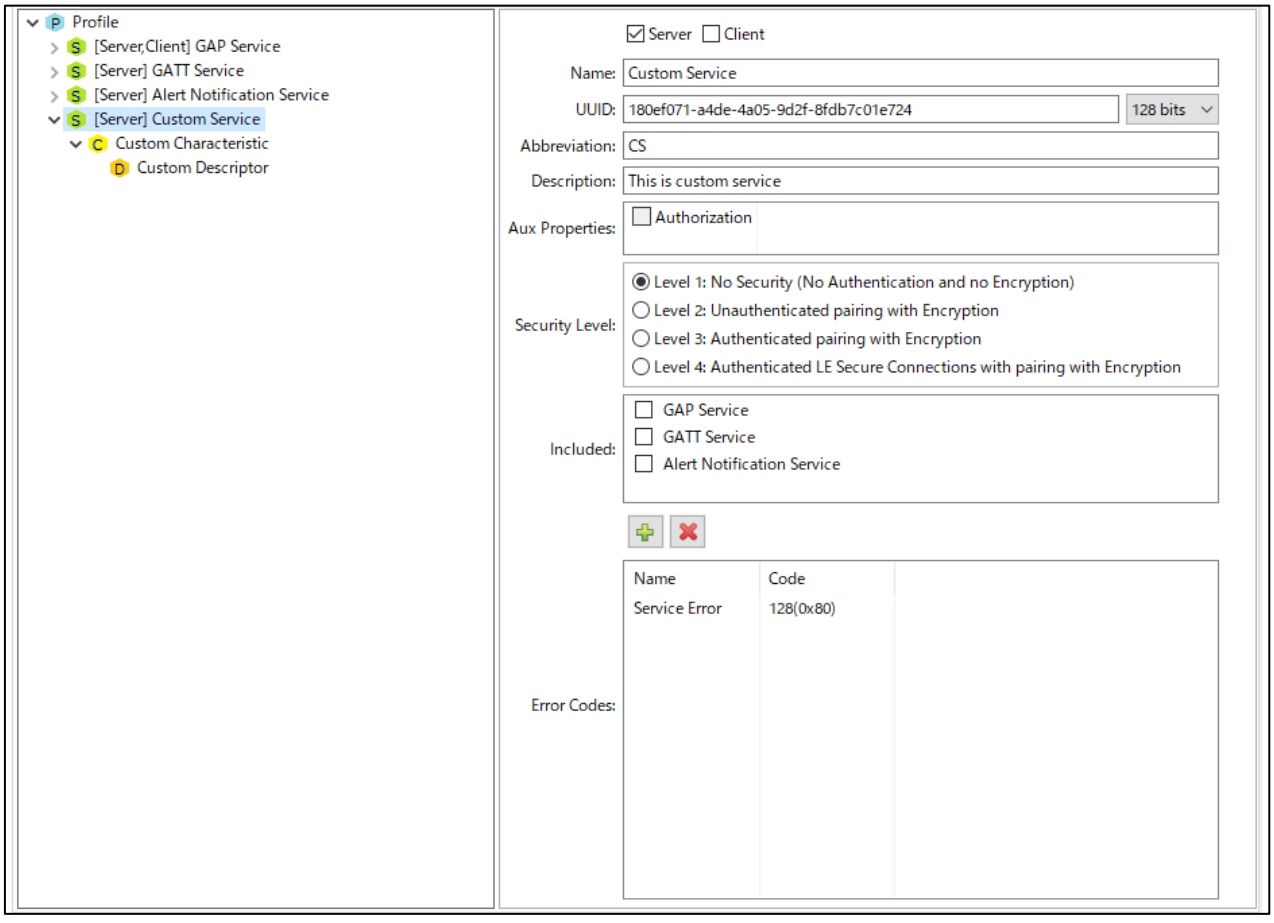


Figure 3.8 Service configuration items

Table 3.1 Service configuration

Item	Description	
Server [optional]	Set check on this item to generate service program as server. It also adds characteristic and descriptors to GATT database.	
Client [optional]	Set check on this item to generate service program as client.	
Name [mandatory]	Name of service. Example) Custom service	
UUID [mandatory]	UUID of service. Select 128bit if service is custom service. Initial value is entered randomly. Modify if needed. Example) 16bit : 0xe237 128bit : 96FE7990-2C76-89AB-DC49-AB7F123DEF9C Note: Lack of “0x” or “-” will not affect code generation.	
Abbreviation [mandatory]	Abbreviation of service. This value is used in file name, function name and variable name. Beware not to conflict with other services. Example) cs	
Description [optional]	Description of service. Explain usage if needed. This description will be used as comments in generated program. Example) This service used for sending sensor data.	
Aux properties [optional]	AUX properties of service. Items below can be configured.	
	Authorization	Enable authorization. The setting is invalid when “Client” is selected. Use function R_BLE_GAP_AuthorizeDev( ) to authorize.
Security Level [mandatory]	Security level required to access services from client devices. The setting is invalid when “Client” is selected. Select from below.	
	Level 1: No Security	Client can access services without Pairing and communication will not be encrypted.
	Level 2: Unauthenticated pairing with Encryption	Client can access services after Pairing without MITM protection using Just Works method. Communication will be encrypted.
	Level 3: Authenticated pairing with Encryption	Client can access services after Pairing with MITM protection using Passkey Entry or OOB mechanism. Communication will be encrypted.
	Level 4: Authenticated LE Secure Connections with pairing with Encryption	Client can access services after Pairing in LE Secure Connections method. Communication will be encrypted.
Included [optional]	Sets Included service. Select the service to be included from the list.	
Error Codes [optional]	Adds error code of service. Error code added can be used by function R_BLE_GATTS_SendErrRsp( ).	
	Name	Name of error code. Example) Value not Supported
	Code	Value of error code. Select from value list.

When a SIG adopted service is added, the changes are restricted. Figure 3.9 shows the service configuration screen for SIG adopted service. In this state, the items Server, Client, Aux Properties, Security Level, and Included can be configured.

Use [Customize] button in case creating a custom service based on SIG adopted service.

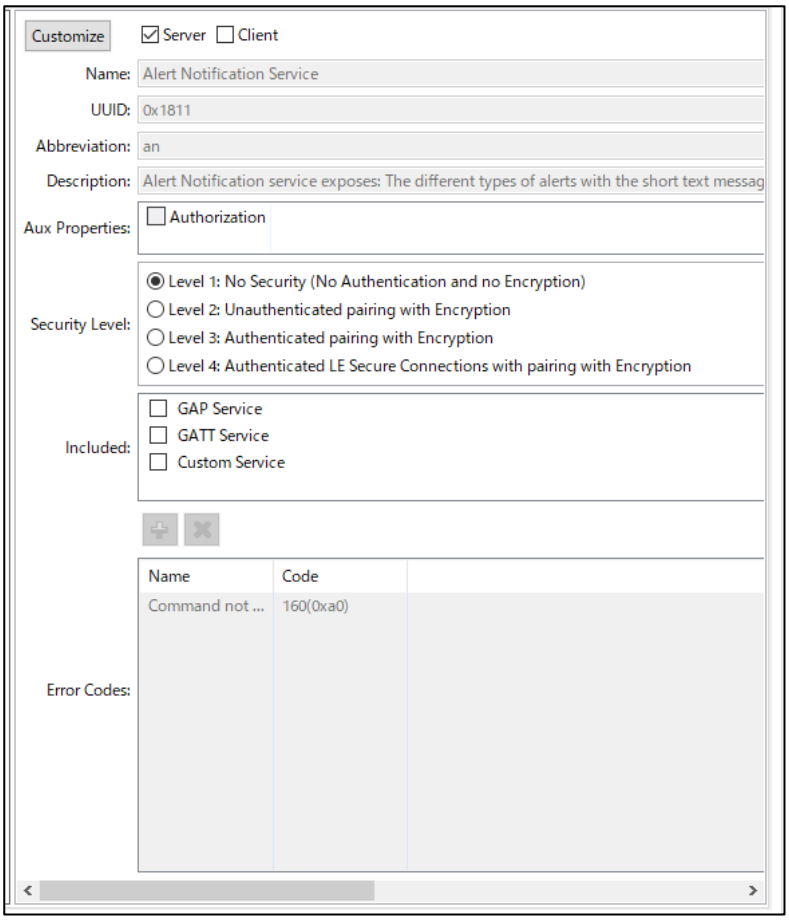


Figure 3.9 SIG adopted service configuration screen

Setting the "Security Level" is an important setting for protecting data in the GATT database.

- When adding a custom service

When adding a custom service, set the "Security Level" that matches the security level required by the product to be developed, referring to Table 3.2.

Note: The Bluetooth SIG's published guide (Bluetooth® Security and Privacy Best Practices Guide) recommends using Security Level 2 or higher for custom profiles.

Table 3.2 Security Level Overview

Security Mode 1	Encryption	User authentication operation during pairing	IO Capability or OOB for MITM Protection	Access via LE Legacy Pairing	Access via LE Secure Connections Pairing	Feature
Security Level 1	Optional	Optional	Optional	Acceptance	Acceptance	Pairing is optional. Settings that permit access to services without encryption.
Level 2	**Required**	Optional	Optional	Acceptance	Acceptance	Authentication operation is optional by end users when pairing. For products that cannot implement IO Capability.
Level 3	**Required**	**Required**	**Required**	Acceptance	Acceptance	Authentication operation is required by end users when pairing so that the pairing which is not involved the users can be suppressed. For products that can implement IO Capability.
Level 4	**Required**	**Required**	**Required**	**Rejection**	Acceptance	Authentication operation is required by end users when pairing so that the pairing whichthat is not involved the users can be suppressed. Since LE Legacy Pairing is rejected, it will strengthen measures against communication eavesdropping. For products that can implement IO Capability and accepts access from only more secure clients. (Access from devices that do not support LE Secure Connections is rejected.)

- When adding SIG standard service

Table 3.3 shows a list of Security Levels that require additional settings when using the SIG standard profiles provided by QE for BLE. Security Level is not set in the SIG standard service provided by QE for BLE, so it is required to set Security Level after adding the required services in the profile to be used.

Add the services required by the SIG standard profile by referring to Table 1.2, and set the Security Level of all added services to the level required by your product among the levels shown in Table 3.3. For example, if the Security Level of the IDP required by the product is determined to be '3', and set the Security Level of IDS and accompanying services (IAS, CTS, BAS, etc.) to '3'. The higher the Security Level value, the better the security. Refer to Table 3.2 for the security level required by the product.

Table 3.3 List of Security Levels that require additional setting when using the SIG standard profile

Profile Specification	Security Level	Remarks
Alert Notification Profile (ANP) 1.0	2 or 3	
Automation IO Profile (AIOP) 1.0	2 or higher	
Blood Pressure Profile (BLP) 1.1.1	2 or 3	
Continuous Glucose Monitoring Profile (CGMP) 1.0.2	2 or 3	Bond Management Service (BMS) must check "Authorization" in "Aux Properties".
Cycling Power Profile (CPP) 1.1	1, 2 or 3	
Cycling Speed and Cadence Profile (CSCP) 1.0	1, 2 or 3	
Environmental Sensing Profile (ESP) 1.0	2 or higher	Environmental Sensing Service (ESS) must check "Authorization" in "Aux Properties".
Find Me Profile (FMP) 1.0	2 or 3	
Fitness Machine Profile (FTMP) 1.0	2 or higher	
Glucose Profile (GLP) 1.0.1	2 or 3	
Health Thermometer Profile (HTP) 1.0	2 or 3	
Heart Rate Profile (HRP) 1.0	2 or 3	
HID over GATT Profile (HOGP) 1.0	2 or 3	
Insulin Delivery Profile (IDP) 1.0.1	3 or 4	Insulin Delivery Service (IDS) must check "Authorization" in "Aux Properties". Bond Management Service (BMS) must check "Authorization" in "Aux Properties".
Location and Navigation Profile (LNP) 1.0	1, 2 or 3	
Phone Alert Status Profile (PASP) 1.0	2 or 3	
Proximity Profile (PXP) 1.0.1	2 or 3	
Pulse Oximeter Profile (PLXP) 1.0.1	2 or higher	
Reconnection Configuration Profile (RCP) 1.0.1	1 or higher	Bond Management Service (BMS) must check “Authorization” in “Aux Properties”.
Running Speed and Cadence Profile (RSCP) 1.0	1, 2 or 3	
Time Profile (TIP) 1.0	2 or 3	
Weight Scale Profile (WSP) 1.0	2 or higher	

Note: The Security Levels listed in Table 3.3 include the requirements formulated prior to Bluetooth LE 4.1(LE Secure Connections function not supported). The Bluetooth SIG's published guide (Bluetooth® Security and Privacy Best Practices Guide) recommends support for Security Level 4 or LE Secure Connections Pairing with Security Levels 2,3 unless there are restrictions on the remote device side.

3.2.3 Adding and configuration characteristic

Press [ ] on the toolbar with service [ ] selected to add a character list to the service.

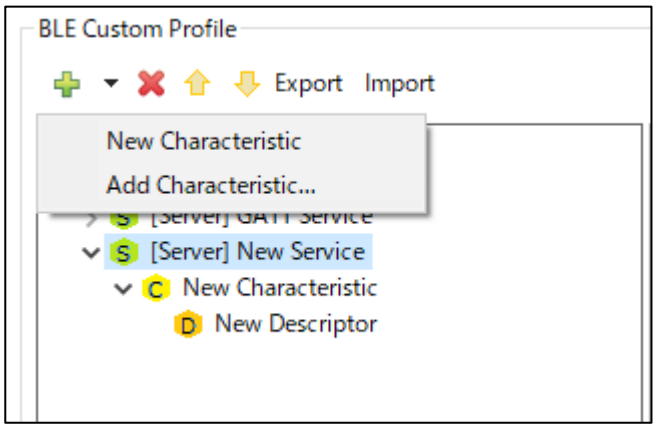


Figure 3.10 Add characteristic

To add a custom characteristic, select [New characteristic]. To add a SIG adopted characteristic, select [Add characteristic].

When you select characteristic [ ] in [Profile Tree], characteristic configuration screen (Figure 3.11) will be shown in [Detail Settings Screen]. Table 3.4 Characteristic configuration Table 3.4 and Table 3.5 describes each item on the configuration screen.

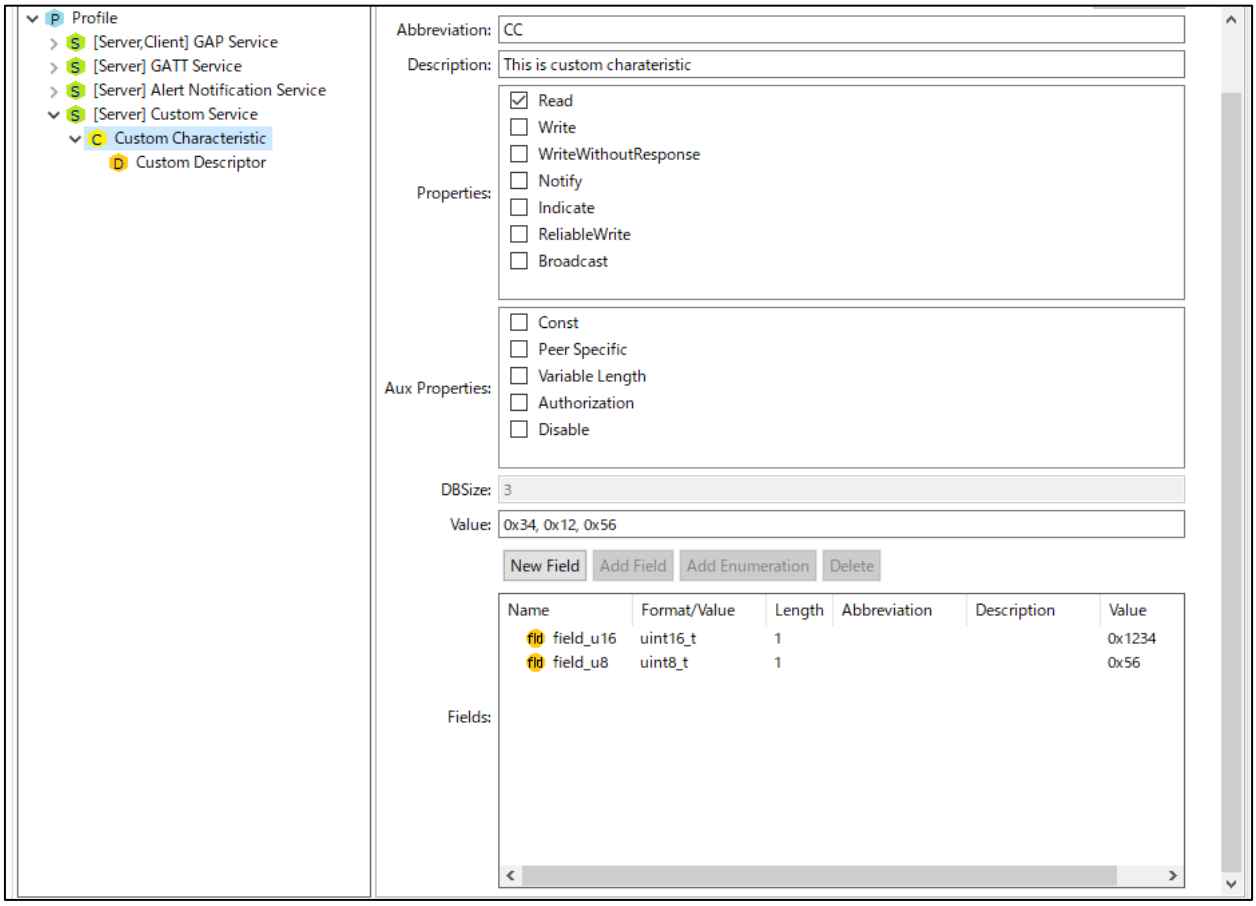


Figure 3.11 Characteristic configuration screen

Table 3.4 Characteristic configuration

Item	Description	
Name [mandatory]	Name of characteristic. Example) Custom Characteristic	
UUID [mandatory]	UUID of characteristic. Select 128bit if service is custom characteristic. Initial value is entered randomly. Modify if needed. Example) 16bit: 0xe237 128bit: 96FE7990-2C76-89AB-DC49-AB7F123DEF9C Note: Lack of “0x” or “-” will not affect code generation.	
Abbreviation [mandatory]	Abbreviation of characteristic. This value is used in function name and variable name. Beware not to conflict with other characteristics. Example) cc	
Description [optional]	Description of Characteristic. Explain usage if needed. This description will be used as comment of generated program. Example) This Characteristic is used for sending sensor data	
Properties [mandatory]	Properties of characteristic which defines operation on Bluetooth LE communication. API and events will be generated for each item checked. [Broadcast] and [ReliableWrite] won’t generate API and events due to its method. Client Characteristic Configuration Descriptor will be added if [Notify] or [Indicate] is selected. Items below can be configured.	
	Read	Enable Read operation.
	Write	Enable Write operation.
	WriteWithoutResponse	Enable Write Without Response operation.
	Notify	Enable Notify operation.
	Indicate	Enable Indicate operation.
	ReliableWrite	Enable Reliable Write operation.
	Broadcast	Enable Broadcast operation.
Aux Properties [optional]	AUX properties of characteristic. Items below can be configured.	
	Const	Value will not be able to change.
	Peer Specific	Value will be kept individually for each connection.
	Variable Length	Value length will be variable.
	Authorization	Enable authorization. Use function R_BLE_GAP_AuthorizeDev() to authorize.
	Disable	Disable attribute.
DBSize [mandatory]	Size of characteristic. Unit of value is byte. Size set in Field will be calculated automatically. If Field with [st_ble_seq_data_t] is set, put maximum size of data.	
Value [optional]	Initial value of characteristic. If you want to enter a number, enter it separated by 8bit digit. If you want to enter string, you can easily enter it by enclosing it in “”. Example) For numbers: 0x12, 0x34, 56,78 For string: “example”	
Field [mandatory]	Set value field used in application. Refer Table 3.5 for configuration.	

Table 3.5 Field configuration

Item	Description		
New Field	Add new field. Items below can be configured		
	Name [mandatory]	Name of field. Example) field_name	
	Format/Value [mandatory]	Format of field. Value can be selected from below.	
		bool	Boolean type
		char	char type
		uint8_t	unsigned 8bit data type
		uint16_t	unsigned 16bit data type
		uint32_t	unsigned 32bit data type
		int8_t	signed 8bit data type
		int16_t	signed 16bit data type
		int32_t	signed 32bit data type
		st_ble_ieee_11073_float_t	IEEE-11073 32bit FLOAT type
		st_ble_ieee_11073_sfloat_t	IEEE-11073 16bit SFLOAT type
		st_ble_date_time_t	Structure for setting date and time information.
		st_ble_dev_addr_t	Structure for setting Bluetooth LE address data.
		st_ble_seq_data_t	Structure for variable length data. Select this when only one field is set, and length is set more than 2.
		struct	Structure type. Select this when selecting [Add Field].
	Length [mandatory]	Data length of field.	
	Abbreviation [optional]	Abbreviation of field.	
	Description [optional]	Description of field. Explain usage if needed	
	Value	Initial value for each field. Value set here will apply to [Value] of descriptor.	
Add Field	Adds a new Field inside the selected Field. Use it if you configure data that has hierarchy. The Format/Value of the selected Field is set to [struct]. Added Field can be configured same items explained in [New Field].		
Add Enumeration	Defines enumeration usable for selected field. Items below can be configured.		
	Name [mandatory]	Name of enumeration. Example) enable	
	Format/Value [mandatory]	Value code of enumeration. Example) 0x01	
	Description [optional]	Description of enumeration.	
Delete	Delete selected field.		

3.2.4 Adding and configuration descriptor

Press [ ] on the toolbar with character list [ ] selected to add a descriptor to the characteristic.

To add a custom descriptor, select [Add New Descriptor]. To add a SIG adopted descriptors, select [Add Descriptor].

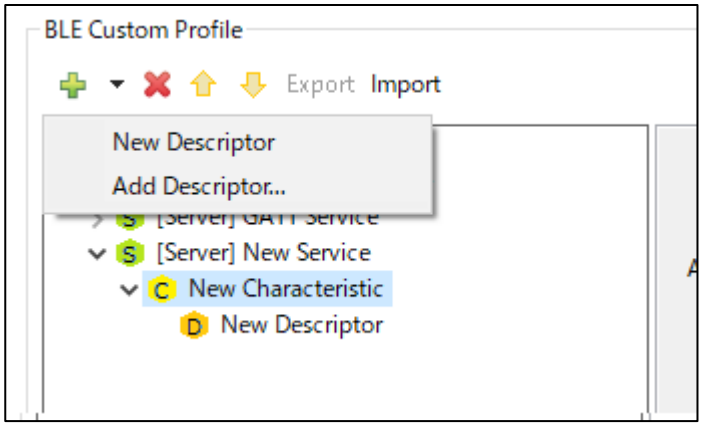


Figure 3.12 Adding descriptor

Selecting the Descriptor [ ] in the Profile tree displays the descriptor settings in the Detail Settings area. Figure 3.13 shows the setting items. Table 3.4 shows the explanation of each setting item.

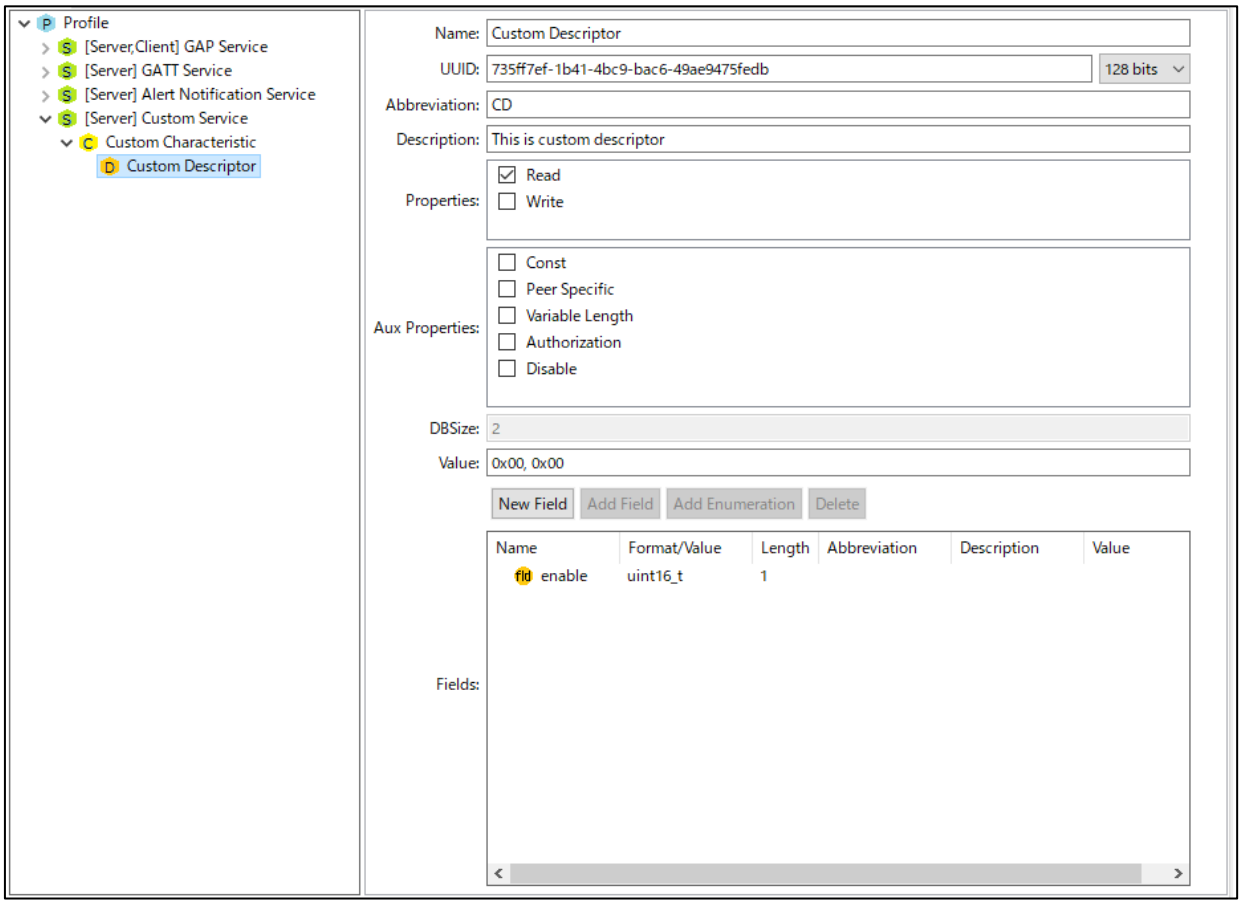


Figure 3.13 Descriptor configuration screen

Table 3.6 Descriptor configuration

Item	Description	
Name [mandatory]	Name of descriptor. Example) Custom Descriptor	
UUID [mandatory]	UUID of descriptor. Select 128bit if service is custom descriptor. Initial value is entered randomly. Modify if needed. Example) 16bit: 0xe237 128bit: 96FE7990-2C76-89AB-DC49-AB7F123DEF9C Note: Lack of “0x” or “-” will not affect code generation.	
Abbreviation [mandatory]	Abbreviation of descriptor. This value is used in function name and variable name. Beware not to conflict with other descriptors. Example) cd	
Description [optional]	Description of descriptor. Explain usage if needed. This description will be used as comment of generated program. Example) This descriptor is used for sending sensor data	
Properties [mandatory]	Properties of descriptor which defines operation on Bluetooth LE communication. API and events will be generated for each item checked. Items below can be configured	
	Read	Enable Read operation.
	Write	Enable Write operation.
Aux Properties [optional]	AUX properties of descriptor. Items below can be configured.	
	Const	Value will not be able to change.
	Peer Specific	Value will be kept individually for each connection.
	Variable Length	Value length will be variable.
	Authorization	Enable authorization. Use function R_BLE_GAP_AuthorizeDev() to authorize.
	Disable	Disable attribute.
DBSize [mandatory]	Size of descriptor. Unit of value is byte. Size set in Field will be calculated automatically. If Field with [st_ble_seq_data_t] is set, put maximum size of data.	
Value [optional]	Initial value of descriptor. If you want to enter a number, enter it separated by 8bit digit. If you want to enter string, you can easily enter it by enclosing it in “”. Example) For numbers: 0x12, 0x34, 56,78 For string: “example”	
Field [mandatory]	Set value field used in application. Refer Table 3.5 for configuration.	

3.3 Configuration of Peripheral

In [Peripheral] tab, you can configure parameters for GAP peripheral role. Parameters set in this tab are used in application framework when you select [Peripheral] in [Profile] tab.

In this tab, you can configure following settings.

Table 3.7 Configurable items in Peripheral

Item	Description
Advertising Data	You can configure Advertising data that will be sent in Advertising event.
Scan Response Data	You can configure Scan response data that will be sent in Advertising event.
Advertising Parameter	You can set parameters for Advertising operation.

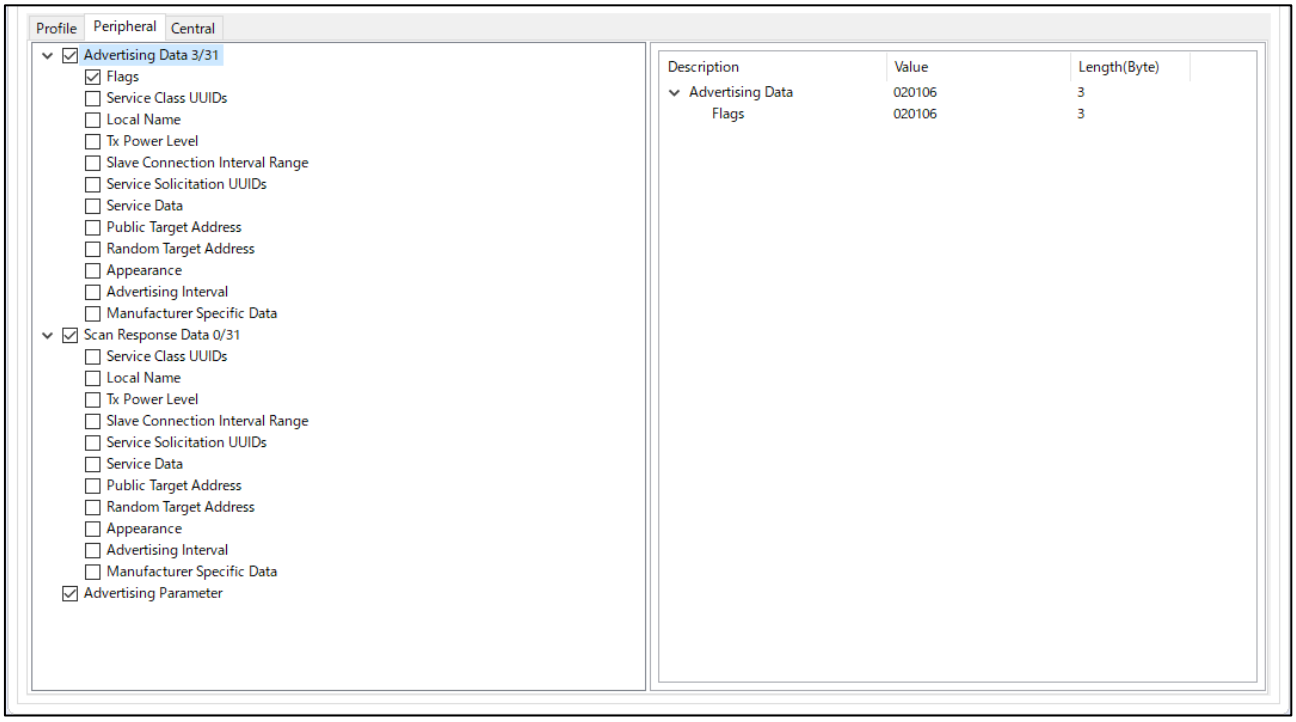


Figure3.14 Peripheral parameter configuration screen

3.3.1 Advertising Data

Advertising Data can be configured by this section. The Data type that are checked will be added as advertising data.

User can also input data value by selecting each data type. Data type that user can select is listed in Table 3.8. Maximum size of Advertising data is 31 bytes, so add data which will not exceed this size. Configure [3.3.2 Scan Response Data] for additional data. Refer [Core Specification Supplement <https://www.bluetooth.com>] for detail about Advertising data.

Table 3.8 List of Selectable Data Type

Data type name	Description	
Flags	This data describes flag of advertising data. Including this data type is necessary for connectable Advertising. This data type can't be selected for scan response data. Select discoverable mode and check for additional information.	
	LE Limited Discoverable Mode	Device will be discoverable for certain period.
	LE General Discoverable Mode	Device will be discoverable all the time.
	Non-Discoverable Mode	Device will not be discovered.
	BR/EDR Not Supported	Check if only Bluetooth LE function is supported.
	Simultaneous LE and BR/EDR to same Device Capable (Controller)	Check if function as Controller roll of Bluetooth LE and BR/EDR can be operated at same time.
	Simultaneous LE and BR/EDR to same Device Capable (Host)	Check if function as Host roll of Bluetooth LE and BR/EDR can be operated at same time.
Service Class UUIDs	This data shows the list of services device offers. You can select services that will be added to the list. Services those are added in Profile tab can be selected.	
Local Name	This data type describes name of advertising device. Select local name type and input the name. Local name can be selected from the below.	
	Short local name	This type describes shortened device name. Use this type when device name is long and extends the size advertising data
	Complete local name	This type describes complete device name.
TX Power Level	This data type describes TX power of advertising device.	
Slave Connection Interval Range	This data type describes connection interval that is recommended from advertising device. Input both MAX/MIN of connection interval.	
Service Solicitation UUIDs	This data type shows the list of service that advertising device requires. You can select services that will be added to the list. Services those are added in Profile tab can be selected.	
Service Data	This data type describes data of service. Value of this data type consists of service UUID and service Data. ex) Service UUID [0x1234] Service Data [0x56, 0x78, 0x9a, 0xbc] →Input data [123456789abc]	
Public Target Address	This data type describes Public BD Address of device that are target of advertising data. ex) Public BD Address [0x12:0x34:0x56:0x78:0x9a:0xbc] →Input data [12345678]	
Random Target Address	This data type describes Random BD Address of device that are target of advertising data. ex) Random BD Address [0x12:0x34:0x56:0x78:0x9a:0xbc] →Input data [12345678]	
Appearance	This data type describes appearance of Advertising device. The value of each appearance is listed in Assigned Numbers page in Bluetooth SIG. https://www.bluetooth.com	
Advertising Interval	This data type describes advertising interval of advertising event. The value input in this item will not be used as the advertising parameter.	
Manufacturer Specific Data	This data type describes data that manufacturer specifies by their own. Value of this data type consists of company ID and specific data. ex) Company ID [0x1234] Specific Data [0x56, 0x78, 0x9a, 0xbc] →Input data [341256789abc]	

3.3.2 Scan Response Data

Scan response data can be configured. The data type that are checked will be added as scan response data.

User can also input data value by selecting each data type. Data type that user can select is listed in Table 3.8.

3.3.3 Advertising Parameter

You can configure parameters used for Advertising operation. Table 3.9 lists the parameters that can be set.

Note: If you have difficulty connecting with the default settings, decrease the [Advertising Interval] parameter.

Table 3.9 Configurable items of Advertising operation

Item	Description	
Fast	You can configure timing information of advertising event. This parameter will be configurable if [Enable Fast Advertising] is checked. If not checked, parameter will be ignored. You can set following items.	
	Advertising Interval	Set Advertising Interval.
	Advertising period	Set Advertising Period. Parameters set in [Fast] will be used for this period.
Slow	You can configure timing information of advertising event. If [Enable Fast Advertising] is checked, this parameter will be used after Fast Advertising period. If not checked, this parameter will be used from the beginning of advertising operation. You can set following items.	
	Advertising Interval	Set Advertising Interval.
	Advertising period	Set Advertising Period. This parameter will be configurable if [Set Advertising Period] is checked. If you want to send Advertising only for certain period, set this parameter.
Advertising channel	You can select Advertising channel that will be used in Advertising. Advertising event will be sent in all channels that is selected.	
Address type	You can select address type that will be used in advertising. Address type can be selected from below.	
	Public address	Public address will be used in advertising event.
	Random Address	Random address will be used in advertising event. Device static address will be used as BD address.

3.4 Configuration of Central

In [Central] tab, you can configure parameters for GAP central role. Parameters set in this tab are used in application framework when you select [Central] in [Profile] tab.

In this tab, you can configure following settings.

Table 3.10 Configurable items of Central

Item	Description
Scan Parameter	You can set parameters for scan operation such as Scan Interval.
Scan Filter Data	You can configure Scan Filter Data that will be used during Scan operation.
Connection Parameter	You can set parameters for connection such as Advertising Interval. Parameter set here will be used in Connection Request.

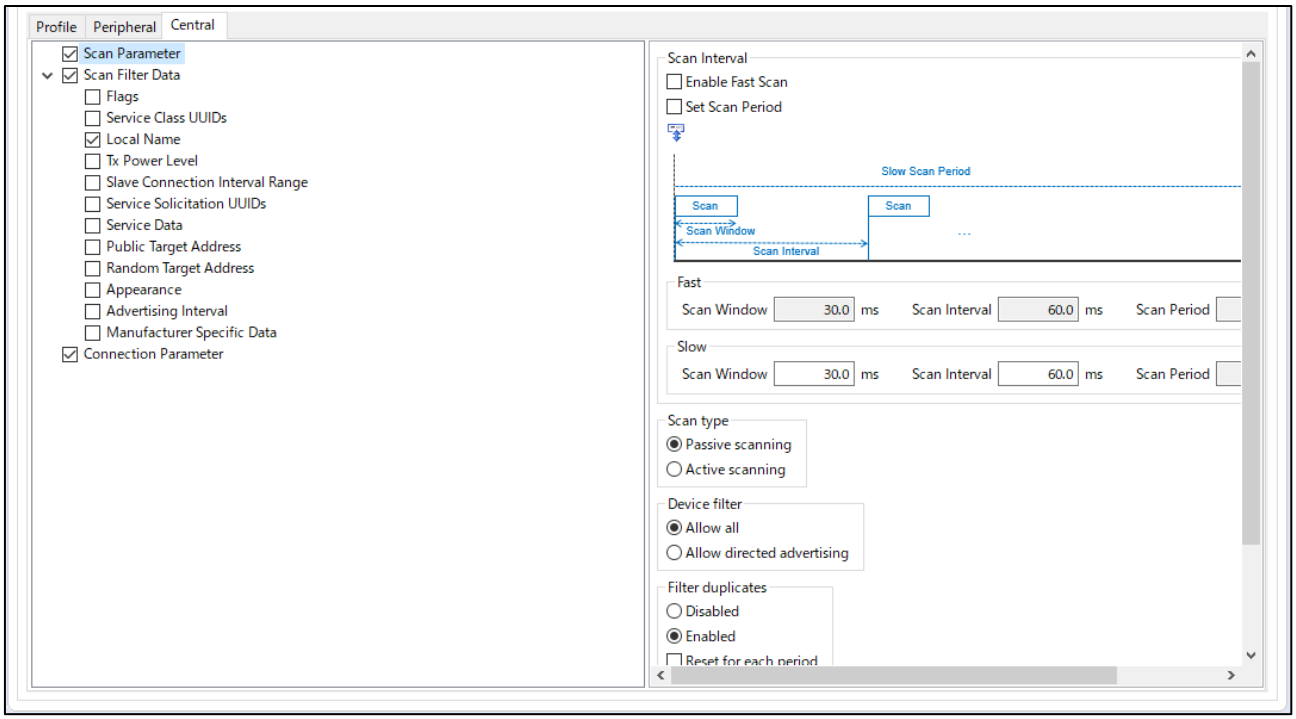


Figure 3.15 Central parameter configuration screen

3.4.1 Scan Parameter

You can configure parameters that will be used in scan operation. Table 3.10 lists the parameters that can be set.

Note: If you have difficulty connecting with the default settings, decrease the [Scan Interval] and increase the [Scan Window].

Table 3.11 Configurable items of Scan Parameter

Item	Description	
Fast	You can configure timing information of scan operation. This parameter will be configurable if [Enable Fast Scan] is checked. If not checked, parameter will be ignored. You can set following items.	
	Scan Window	Set Scan Window.
	Scan Interval	Set scan Interval.
	Scan Period	Set scan Period. Parameters set in [Fast] will be used for this period.
Slow	You can configure timing information of scan operation. If [Enable Fast Scan] is checked, this parameter will be used after Fast scan period. If not checked, this parameter will be used from the beginning. You can set following items.	
	Scan Window	Set Scan Window.
	Scan Interval	Set scan Interval.
	Scan Period	Set scan Period. This parameter will be configurable if [Set Scan Period] is checked. If you want to operate scan only for certain period, set this parameter.
Scan type	You can select scan type. Scan type can be selected from below.	
	Passive Scanning	Passive scan will operate as scan operation.
	Active Scanning	Active scan will operate as scan operation.
Device filter	You can select device filter that will be used in scan operation. Device filter can be selected from below.	
	Allow all	Scan operation well accept all advertising and scan response PDUs except directed advertising PDUs not addressed to local device.
	Allow directed advertising	Scan operation will accept all advertising and scan response PDUs except directed advertising PDUs whose target address is identity address but doesn't address local device. However, directed advertising PDUs whose target address is the local resolvable private address are accepted.
Filter duplicates	You can select filter duplicate parameter that will be used in scan operation. Filter duplicates can be selected from below.	
	Disable	Duplicate filter will be disabled.
	Enabled	Duplicate filter will be enabled. If you check [Reset for each period], duplicate filter will reset for each scan period.

3.4.2 Scan Filter Data

Filter data for scan operation can be configured by this section. Only advertising event which has data that matches filter data will be notified to the application framework.

The data type which is checked will be used as filter data. User can also input data value by selecting data type. Data type that user can select is listed in Table 3.8.

Note: Only one data type can be selected as Scan Filter Data

3.4.3 Connection Parameter

You can configure parameter used for connection event. This parameter will be used in connection request.

Table 3.12 Configurable item of connection

Item	Description	
Parameter	You can configure connection parameter. Parameter set here will be sent with connection request and used after connection established. You can set following items.	
	Connection Interval	Set connection interval.
	Connection Latency	Set peripheral latency.
	Connection Supervision Timeout	Set supervision timeout.
Connection cancel	You can configure connection cancel parameter. You can set following items.	
	Connection Timeout	Set connection timeout. If peripheral device doesn't respond to connection request for connection timeout, connection will be canceled.

4. Implementation of program

This chapter describes how to implement a user application based on source code generated from QE for BLE.

Table 4.1 shows the programs generated from QE for BLE.

Table 4.1 Programs generated by QE for BLE

Program	file name	description
Application Framework	app_main.c	A framework for user applications and profiles. Skeleton program that is the basis of application/profile development.
GATT Database Program	gatt_db.c gatt_db.h	GATT database program Data structure of service which is checked on [server] in QE for BLE is defined.
Service API Program	r_ble_[abbreviation][s or c].c r_ble_[abbreviation][s or c].h	Profile API program API program for accessing and notifying profile data. File is generated for each service that configure profile. Each file name is determined based on the [abbreviation], [server], and [client] set in QE for BLE. [abbreviation][s] is the server program, [abbreviation][c] is the client program. Example) [abbreviation]=[sig], [server]: r_ble_sigs.c, r_ble_sigs.h [abbreviation]=[cus], [client]: r_ble_cusc.c, r_ble_cusc.h
Profile Common Library	profile_cmn discovery	This is a program for the common part of the profile. It is generated for the RA4W1, RX23W environment.

This chapter provides an example implementation of a custom service with the following profile: XX service, YYY characteristic, and ZZZ descriptor are added.

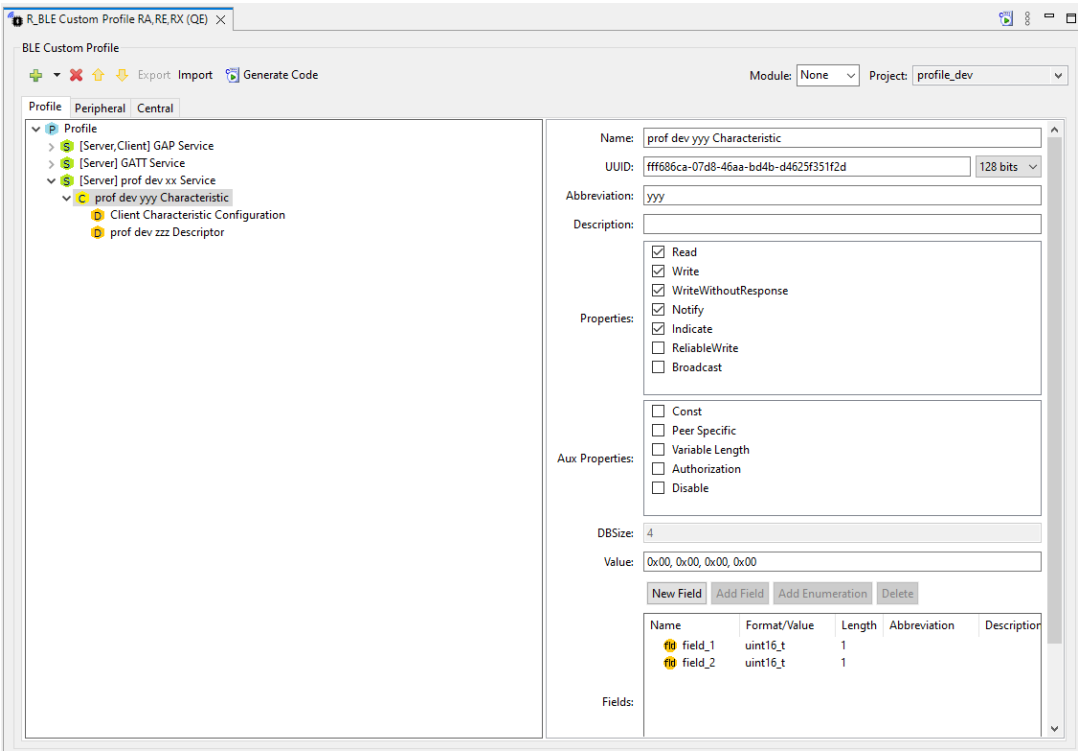


Figure 4.1 Characteristic used in description

Table 4.2 Profile used in description

Service	abbreviation	Role
GAP Service	gap	Server Client
GATT Service	gat	Server Client
prof dev XX Service	xx	Server Client
Characteristic name	**abbreviation**	**Property**
prof dev YYY Characteristic	yyy	Read Write WriteWithoutResponse Notify Indication
Descriptor name	**abbreviation**	**Property**
Client Configuration Characteristic Descriptor	cli cnfg	Read Write
prof dev ZZZ Descriptor	zzz	Read Write

Figure 4.2 shows an example of code generated by QE for BLE. Each program is generated in the following folder.

- [project name]/qe_gen/ble

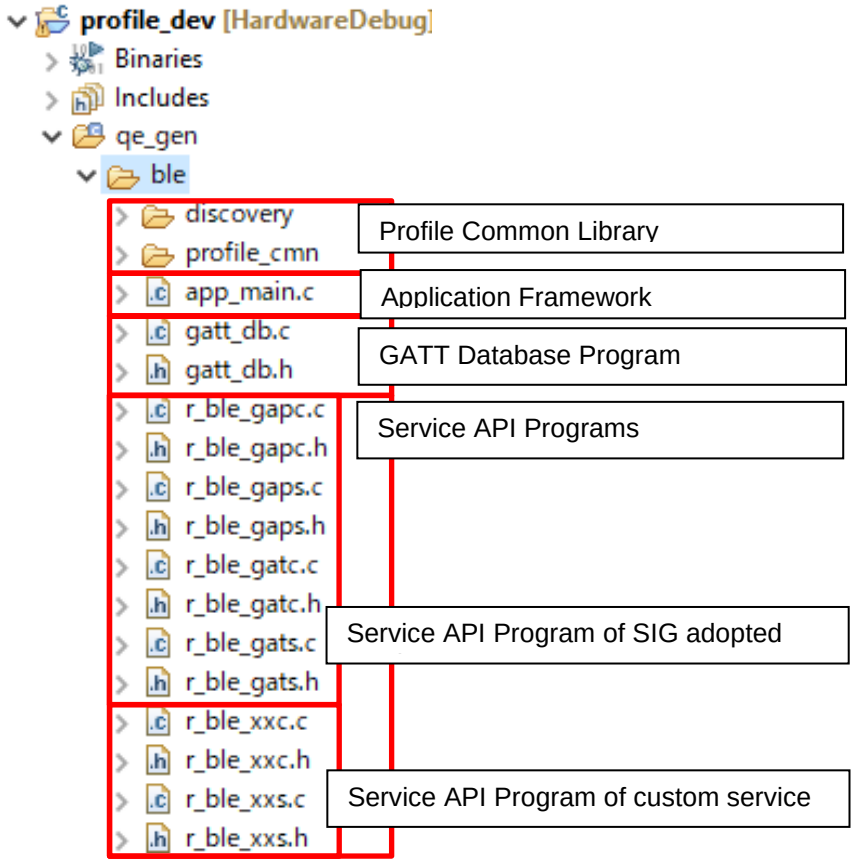


Figure 4.2 Source Code example generated by QE for BLE

QE for BLE provides user code protection during code generation. Code added within the code block shown below will be retained after code generation. When implementing it in each program, implement it between these comments.

```
/* Start user code for XXXX. Do not edit comment generated here */

    Implement user code here

/* End user code. Do not edit comment generated here */
```

Figure 4.3 User code blocking comment

4.1 Service API Programs (r_ble_xxs.c / r_ble_xxc.c)

The service API program is a program that simplifies data communication by profiles.

This section describes the details of the generated API and how to implement the encode/decode functions required to use the API.

Each service will generate the functions in Table 4.3, regardless of its role. In the table, [xx] is the string set to the [abbreviation] of the service in QE for BLE, and [S or C] is set to “S” if the service is a server or “C” if it is a client.

Table 4.3 API defined in each service API program

API	Description
R_BLE_[service][S or C]_Init	Initialization function for the service. Register the service in profile common library. Calling this function is necessary before using service API program. The result of the GATT operation is notified to the callback function registered with this function.

The Service API program provides an API for GATT operation. The API corresponding to the GATT operation is generated according to the [Property] selected in QE for BLE.

Table 4.4 GATT Operation API generated in client role

Function Name	Description
R_BLE_XXC_WriteYyy R_BLE_XXC_WriteYyyZzz	The GATT database is written from the client side. A response is returned from the server upon completion of the write. Generated when the Write property of QE for BLE is enabled.
R_BLE_XXC_ReadYyyZzz	Reads the server's GATT database. Generated when the Read property of QE for BLE is enabled.
R_BLE_XXC_WritewithoutRespYyy	The GATT database is written from the client side. No response is returned from the server upon completion of the write. Generated when the Write Without Response property of QE for BLE is enabled.
R_BLE_XXC_ServDiscCb	Used to receive and retain the results of service discovery by the discovery library in the profile common library.
R_BLE_XXC_GetServAttrHdl	Obtains a service-discovered attribute handle.

Table 4.5 GATT Operation API generated in server role

Function Name	Description
R_BLE_XXS_NotifyYyy	Sends data from the server to the client. This is generated when the Notify property of QE for BLE is enabled. The GATT database is not written with this operation.
R_BLE_XXS_IndicateYyy	Sends data from the server to the client. Receives acknowledgement of receipt from the client. This is generated when the Indicate property of QE for BLE is enabled. The GATT database is not written with this operation.
R_BLE_XXS_SetYyy R_BLE_XXS_SetYyyZzz	Sets a value in the GATT database. Generated when one of QE for BLE's Read, Write, or Write Without Response properties is enabled.
R_BLE_XXS_GetYyy R_BLE_XXS_GetYyyZzz	Retrieves a value into the GATT database. Generated when the Read, Write, or Write Without Response properties is enabled in QE for BLE.

Events during GATT operation are notified to the callback function registered in R_BLE_XX [S / C] _Init. The events notified to the server side are shown in Table 4.6, and the events notified to the client are shown in Table 4.7.

Table 4.6 Event that occurred server

Event	Description
BLE_XXS_EVENT_Yyy_WRITE_REQ	Occurs when a GATT database write request is received by the Write operation.
BLE_XXS_EVENT_Yyy_WRITE_COMP BLE_XXS_EVENT_Yyy_Zzz_WRITE_COMP	Occurs when writing to the GATT database by the Write operation is complete.
BLE_XXS_EVENT_WRITE_CMD	Occurs when a GATT database write by Write Without Response is received.
BLE_XXS_EVENT_Yyy_READ_REQ	Occurs when a GATT database read request is received by the Read operation.
BLE_XXS_EVENT_Yyy_HDL_VAL_CNF	Occurs when a confirmation packet for Indicate operation is received.

Table 4.7 Event that occurred client

Event	Description
BLE_XXS_EVENT_Yyy_WRITE_RSP BLE_XXS_EVENT_Yyy_Zzz_WRITE_RSP	Occurs when a response to a write request is received for a Write operation.
BLE_XXC_EVENT_Yyy_READ_RSP BLE_XXC_EVENT_Yyy_Zzz_READ_RSP	Occurs when the read result of the Read operation is received.
BLE_XXC_EVENT_Yyy_HDL_VAL_NTF	It occurs when data is received by the Notify operation.
BLE_XXC_EVENT_Yyy_HDL_VAL_IND	It occurs when data is received by the Indicate operation.

In the service API program, characters and descriptors are represented by data types that reflect the QE for BLE Field settings. The characteristic and descriptor data types can be seen in the characteristic definition structure as shown in Figure 4.4

```
/* prof dev yyy Characteristic characteristic definition */
static const st_ble_servc_char_info_t gs_yyy_char = {
    .uuid_128     = BLE_XXXC_YYY_UUID,
    .uuid_type    = BLE_GATT_128_BIT_UUID_FORMAT,
    .app_size     = sizeof(st_ble_xxc_yyy_t),
    .db_size      = BLE_XXC_YYY_LEN,
    .char_idx     = BLE_XXC_YYY_IDX,
    .p_attr_hdls  = gs_yyy_char_ranges,
    .decode       = (ble_servc_attr_decode_t)decode_st_ble_xxc_yyy_t,
    .encode       = (ble_servc_attr_encode_t)encode_st_ble_xxc_yyy_t,
    .num_of_descs = ARRAY_SIZE(gspp_yyy_descs),
    .pp_descs     = gspp_yyy_descs,
};
```

Figure 4.4 Application data type reference notified to the event (in r_ble_xx [c/s].c)

If there are multiple elements set [Fields] in the QE for BLE, the characteristic or descriptor structure is defined. It is defined in the header file as shown in Figure 4.5.

```
/*****************************************************************************//**
 * @brief prof dev yyy Characteristic value structure.
*****************************************************************************/
typedef struct {
    uint16_t field_1; /**< field_1 */
    uint16_t field_2; /**< field_2 */
} st_ble_xxc_yyy_t;
```

Figure 4.5 Characteristic application data structure (r_ble_xx[c/s].h)

Encode / decode functions convert the application data contained in the radio packet PDU and this structure to each other.

4.1.1 Description of encode/decode functions

The application layer handles characteristic and descriptor value in accordance with the format specified by the [Fields] of QE for BLE. On the other hand, in the GATT database and Bluetooth LE Protocol Stack, these formats are treated as an 8-bit data array.

The profile common library uses the encode / decode function for each characteristic to convert the data structure for the application and the 8-bit array data for the GATT database.

Figure 4.6 shows the feature of encode/decode function.

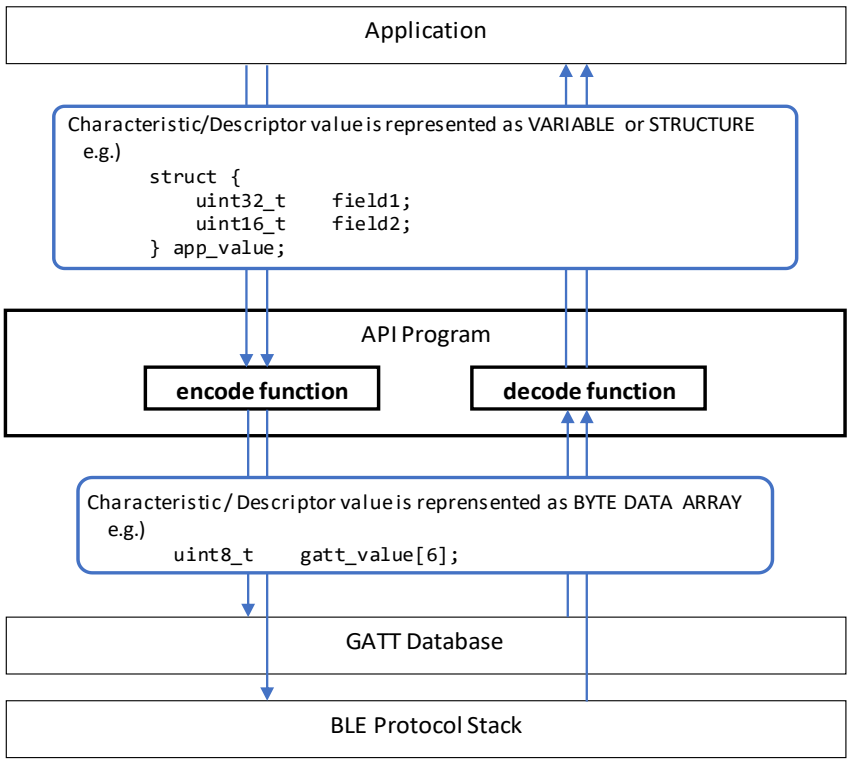


Figure 4.6 Feature of encode/decode function

The encode function is used by the profile common library when API to send characteristic or descriptor value or to change characteristic or descriptor value of own GATT Database is called. Also, the decode function is used by the profile common library before callback function to notify characteristic or descriptor value received.

Figure 4.7 shows a use-case of the encode/decode function that GATT Client writes new Characteristic value to peer GATT Server. The encode function is used by API Program of the client side and then the decode function is used by API Program of the server side.

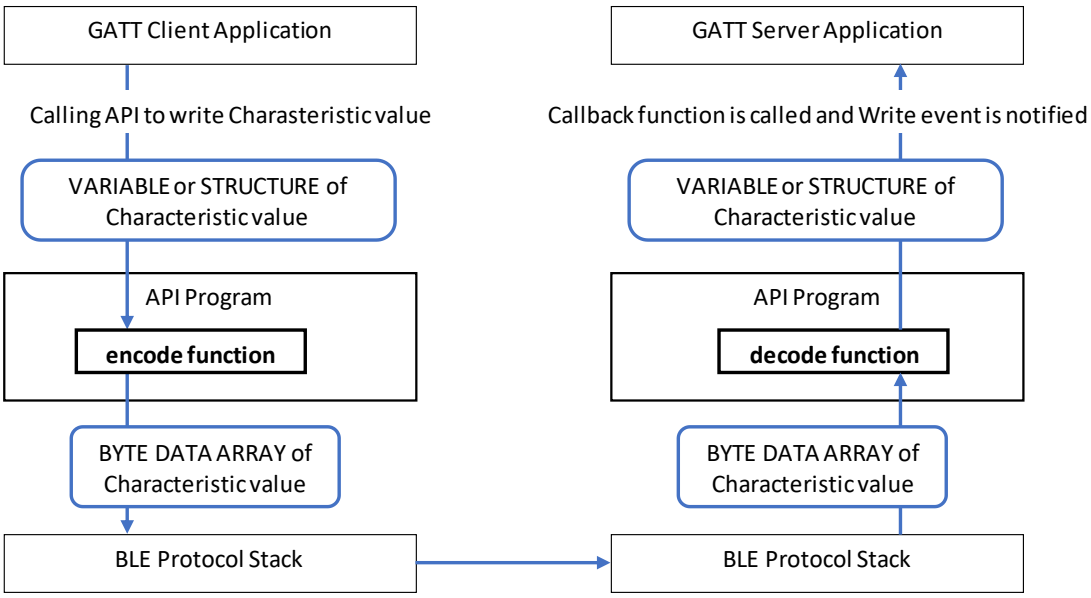


Figure 4.7 Use-Case of the encode/decode Function writing Characteristic value

Similarly, Figure 4.8 shows a use-case of the encode/decode function that GATT Server notifies new Characteristic value to peer GATT Client. The encode function is used by API Program of the server side and then the decode function is used by API Program of the client side.

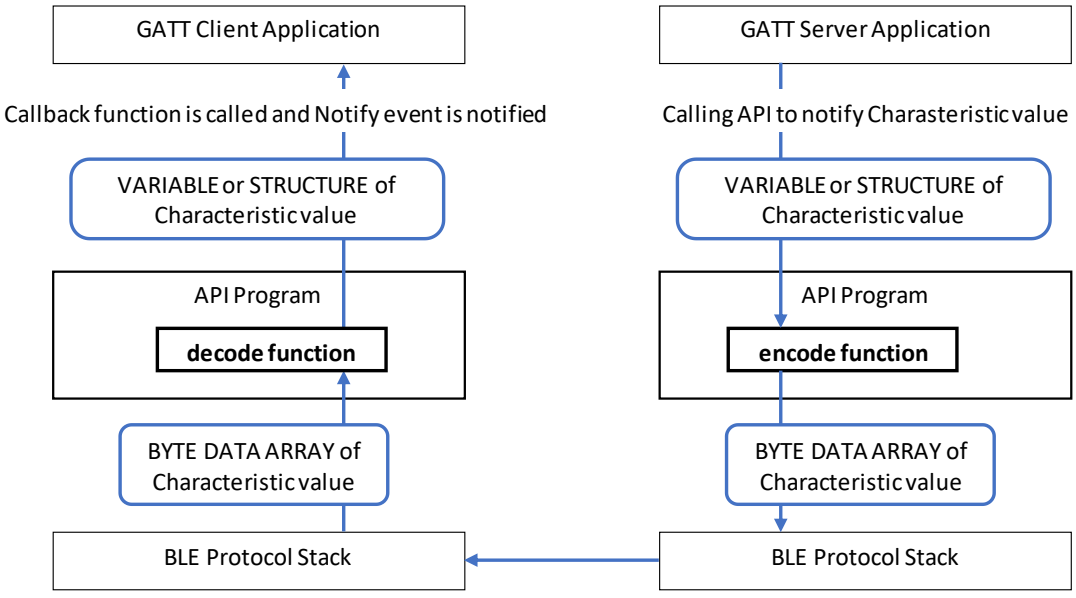


Figure 4.8 Use-Case of the encode/decode Function notifying Characteristic value

4.1.2 Automatic generation of encode/decode functions

Encode/decode functions are automatically generated by QE for BLE Utility 1.60 or later. This section describes the generated encode/decode functions. The encode/decode function encodes/decodes the data type set in the field so that it has the number of bytes shown in Table 4.8

Table 4.8 Structure and number of bytes that can be set in QE for BLE

Deta Type	Number of bytes
bool, char, uint8_t, int8_t	1
uint16_t, int16_t	2
uint32_t, int32_t	4
st_ble_ieee11073_sfloat_t	2
st_ble_ieee11073_float_t	3
st_ble_dev_addr_t	7
st_ble_date_time_t	7

If “st_ble_seq_data_t” is contained in a field with multiple data types, an empty encode/decode function is generated for that characteristic/descriptor.

Figure 4.9 shows examples of fields and byte sequences. Pack each field into the transmission data in order from the top.

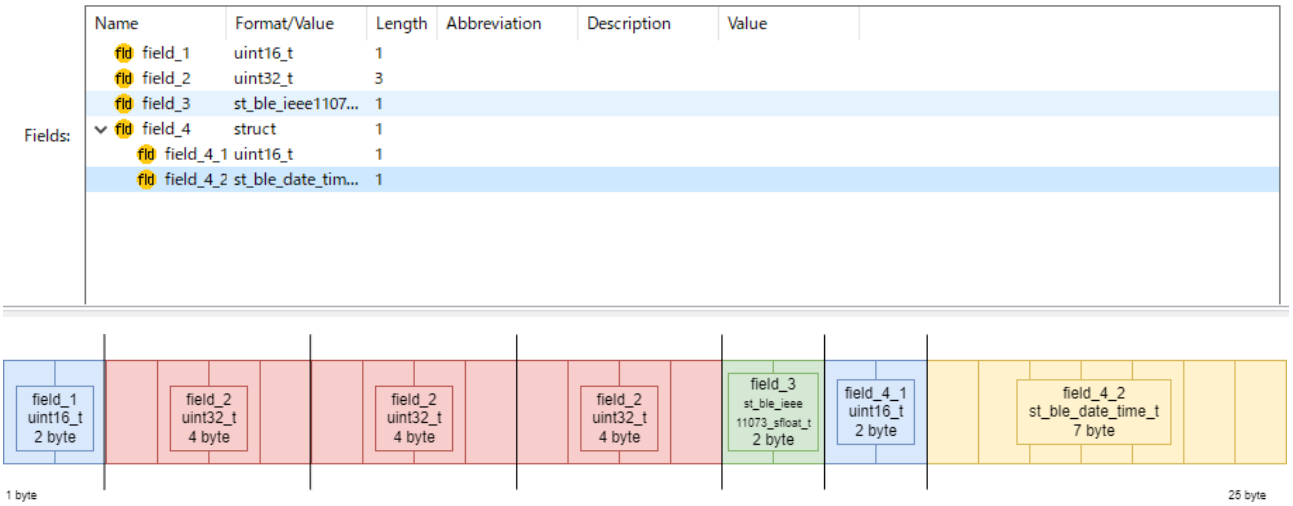


Figure 4.9 Descriptive fields and bytes of the generated encode/decode function

The generated encode functions are shown in Figure 4.10. If Length is set to a value greater than 1, the “for statement” will pack it into the transmission data.

```
static ble_status_t encode_st_ble_xxc_prof_t(
    const st_ble_xxc_prof_t *p_app_value,
    st_ble_gatt_value_t *p_gatt_value)
{
    /* Start user code for characteristic value encode function. */
    /* End user code. Do not edit comment generated here */
#ifndef BLE_XXC_DISABLE_ENCODE_DECODE

    uint32_t pos = 0;

    BT_PACK_LE_2_BYTE(&p_gatt_value->p_value[pos],&p_app_value->field_1);
    pos += 2;

    for (uint32_t i=0;i<3;i++)
    {
        BT_PACK_LE_4_BYTE(&p_gatt_value->p_value[pos],&p_app_value->field_2[i]);
        pos += 4;
    }

    pos += ble_pack_st_ble_ieee11073_sfloat_t(
                              &p_gatt_value->p_value[pos],
                              &p_app_value->field_3);

    pos += ble_pack_st_ble_xxc_prof_field_4_t(
                              &p_gatt_value->p_value[pos],
                              &p_app_value->field_4);

    p_gatt_value->value_len = (uint16_t)pos;
#endif /* BLE_XXC_DISABLE_ENCODE_DECODE */


    return BLE_SUCCESS;
  }
```

Figure 4.10 Example for generated encode function

Figure 4.11 shows the generated decode function.

```
static ble_status_t decode_st_ble_xxc_prof_t(
                                st_ble_xxc_prof_t *p_app_value,
                                const st_ble_gatt_value_t *p_gatt_value)
{
    /* Start user code for New Characteristic value decode function.   */
    /* End user code. Do not edit comment generated here */
#ifndef BLE_XXC_DISABLE_ENCODE_DECODE


    uint32_t pos = 0;

    BT_UNPACK_LE_2_BYTE(&p_app_value->field_1,&p_gatt_value->p_value[pos]);
    pos += 2;

    for (uint32_t i=0;i<3;i++)
    {
        BT_UNPACK_LE_4_BYTE(&p_app_value->field_2[i],&p_gatt_value->p_value[pos]);
        pos += 4;
    }

    pos += ble_unpack_st_ble_ieee11073_sfloat_t(
                            &p_app_value->field_3,
                            &p_gatt_value->p_value[pos]);

    pos += ble_unpack_st_ble_xxc_prof_field_4_t(
                            &p_app_value->field_4,
                            &p_gatt_value->p_value[pos]);

#endif /* BLE_XXC_DISABLE_ENCODE_DECODE */


    return BLE_SUCCESS;
  }
```

Figure 4.11 Example for generated decode function

These encode/decode functions can be disabled by defining the following macros in the user code block as shown in Figure 4.12.

If you disable it, refer to Chapter 4.1.3 and implement the encode/decode function.

```
/* Start user code for function prototype declarations and global variables. Do not
edit comment generated here */
#define BLE_XXC_DISABLE_ENCODE_DECODE
  /* End user code. Do not edit comment generated here */
```

Figure 4.12 Disabling generated encode/decode functions

4.1.3 Implementing the encode-decode function

Using for versions earlier than QE for BLE Utility 1.60 or when the automatically generated encode/decode function is disabled, implement the encode/decode function corresponding to each data structure. Therefore, implementation of the encode/decode function for each data structure is needed. For basic data structures such as uint8_t type and commonly used data structures such as ieee11073 SFLOAT type, you can implement encode/decode function by calling appropriate encode/decode macros and functions. Table 4.9 describes the list of provided encode/decode macros and functions.

Table 4.9 encode/decode macro or function

Type of Field	encode	decode
char uint8_t int8_t	BT_PACK_LE_1_BYTE(*dst, *src)	BT_UNPACK_LE_1_BYTE(*dst, *src)
uint16_t int16_t	BT_PACK_LE_2_BYTE(*dst, *src)	BT_UNPACK_LE_2_BYTE(*dst, *src)
uint32_t int32_t	BT_PACK_LE_4_BYTE(*dst, *src)	BT_UNPACK_LE_4_BYTE(*dst, *src)
st_ble_ieee11073_sfloat_t	pack_st_ble_ieee11073_sfloat_t(*p_dst, *p_src)	unpack_st_ble_ieee11073_sfloat_t(*p_dst, *p_src)
st_ble_date_time_t	pack_st_ble_date_time_t(*p_dst, *p_src)	unpack_st_ble_date_time_t(*p_dst, *p_src)

Figure 4.13 shows implementation of a encode function for characteristic which has field shown in Figure 4.14. In this encode function, encode macros and functions provided in Table 4.9 are used.

```
static ble_status_t decode_st_ble_xxxc_yyy_t(st_ble_xxxc_yyy_t *p_app_value, const
st_ble_gatt_value_t *p_gatt_value)
{
    /* Start user code for profile dev yyy Characteristic characteristic value decode
function. Do not edit comment generated here */
        uint32_t pos = 0;
        BT_UNPACK_LE_16_BYTE(&p_app_value->field_1, p_gatt_value->p_value[pos]);
        pos += 2;

        BT_UNPACK_LE_16_BYTE(&p_app_value->field_2, p_gatt_value->p_value[pos]);
        pos +=2;
    /* End user code. Do not edit comment generated here */
    return BLE_SUCCESS;
}

static ble_status_t encode_st_ble_xxc_yyy_t(const st_ble_xxc_yyy_t *p_app_value,
st_ble_gatt_value_t *p_gatt_value)
{
    /* Start user code for profile dev yyy Characteristic characteristic value encode
function. Do not edit comment generated here */
        uint32_t pos = 0;
        BT_PACK_LE_16_BYTE(&p_gatt_value->p_value[pos],p_app_value->field_1);
        pos += 2;

        BT_PACK_LE_16_BYTE(&p_gatt_value->p_value[pos],p_app_value->field_2);
        pos += 2;
        /* End user code. Do not edit comment generated here */
    return BLE_SUCCESS;
  }
```

Figure 4.13 Example of implementing encode function

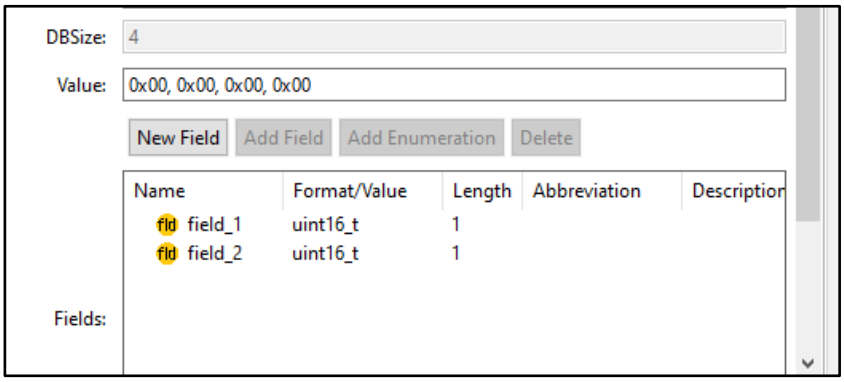


Figure 4.14 Example of field

For the SIG adopted service, the encode/decode functions are already implemented. Therefore, this step is unnecessary. Also, if [Fields] has only one of the following types, it is not necessary to implement the encode/decode function.

uint8_t, uint16_t, uint32_t, int8_t, int16_t, int32_t,

st_ble_seq_data_t, st_ble_ieee11073_sfloat, st_ble_date_time_t

4.2 Application Framework (app_main.c)

In app_main.c, a program that realizes Bluetooth LE communication according to the role of the application is pre-implemented.

Figure 4.15 shows the sequence chart when app_main.c generated by each role is executed. In this case, the central device is the GATT client, and the peripheral device is the GATT server.

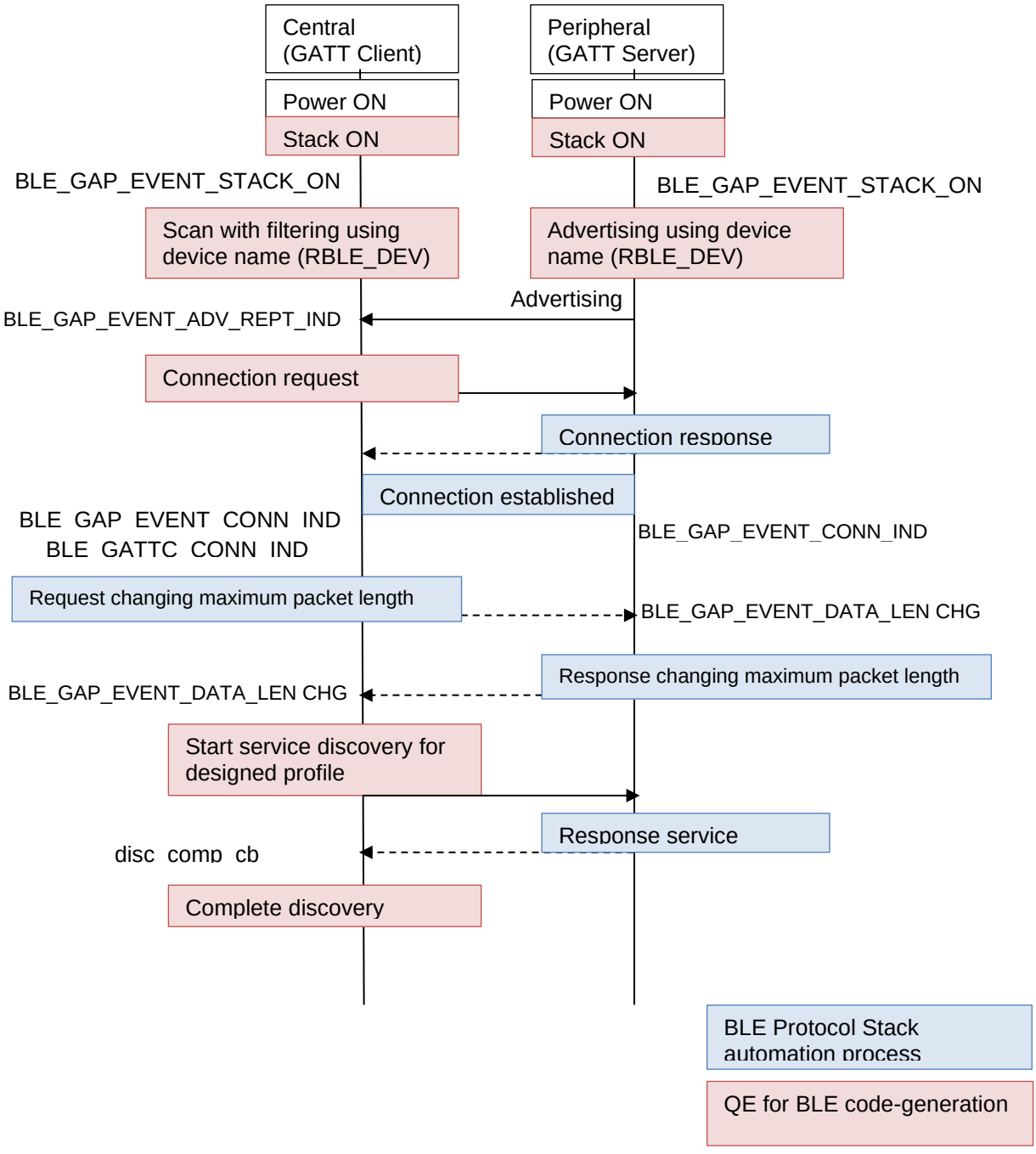


Figure 4.15 Behavior of the program implemented in app_main.c

When service discovery is complete, you will be notified of disc_comp_cb. The client can perform profile communication after calling this callback function.

app_main.c is an application skeleton program that contains a profile configured with QE for BLE. By adding processing to the function defined on the source code, data communication by profile is realized.

Here, we will explain how to implement profile data communication using a service API program, using a typical GATT operation as an example. If you want to implement other Bluetooth LE features, refer to the “Application Developer's Guide”.

4.2.1 Responding to security requirements

If you set the service Security Level 3 or higher in section 3.2.2, you need to change the pairing parameters to perform data communication.

4.2.1.1 When set to Security Level 3

Requires user interaction and MITM protection during pairing. To achieve these, the device must have input/output capabilities. Change the io capability to match your device's capabilities.

In the RX23W environment, change the pairing parameters in app_main.c.

```
/* Pairing parameters */
static st_ble_abs_pairing_param_t gs_abs_pairing_param =
{
    .iocap          = BLE_GAP_IOCAP_NOINPUT_NOOUTPUT,
    .mitm           = BLE_GAP_SEC_MITM_BEST_EFFORT,
    .sec_conn_only  = BLE_GAP_SC_BEST_EFFORT,
    .loc_key_dist   = BLE_GAP_KEY_DIST_ENCKEY,
    .rem_key_dist   = 0,
    .max_key_size   = 16,
  };
```

Figure 4.16 Change of pairing parameter in app_main.c (RX23W)

In RA4W1 environment, pairing parameters can be set from RA Configurator as shown in Figure 4.17

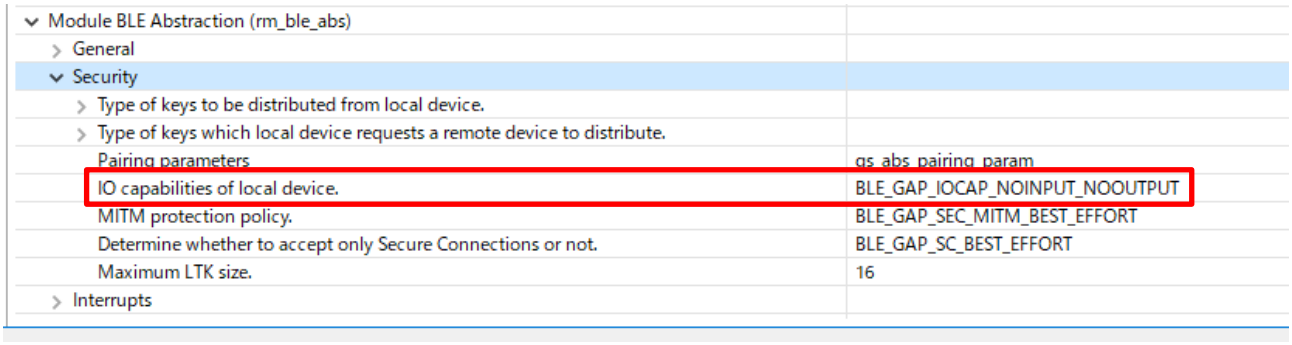


Figure 4.17 Change pairing parameters with RA Configurator (RA4W1)

Please see the following documents for details.

Table 4.10 Reference Documents

MCU	Documents	Chapter
RX23W	RX23W Group Bluetooth Low Energy Application Developer's Guide (R01AN5504)	9.1 Paring
RA4W1	RA4W1 Group Bluetooth Low Energy Application Developer's Guide (R01AN5653)	8.1 Pairing

4.2.1.2 When set to Security Level 4

After performing the settings in section 4.2.1.1, the remote device must support LE Secure Connection.

RX23W, RE01B, and RA4W1 all support LE Secure Connection, but if the remote device does not support LE Secure Connection, access to the GATT database will not be permitted.

4.2.2 Exchange MTU

MTU is the maximum data length that can be sent and received in one GATT operation. The MTU when connected is 23 bytes. The MTU can be changed from the client only once during the connection.

Notify and Write Without Response operations that do not require confirmation of receipt from the peer device can efficiently and continuously send data but cannot send data larger than the MTU-3 byte. One solution for efficient data transmission is to change the MTU so that the characteristic size is MTU-3 bytes or less.

The MTU size supported depends on the device. The Bluetooth LE Protocol Stack supports up to 247 bytes. The MTU is set to the smaller of the MTUs supported by each other's devices.

The sequence chart of MTU exchange is shown. The red text is the function call, and the blue text is the application notification event.

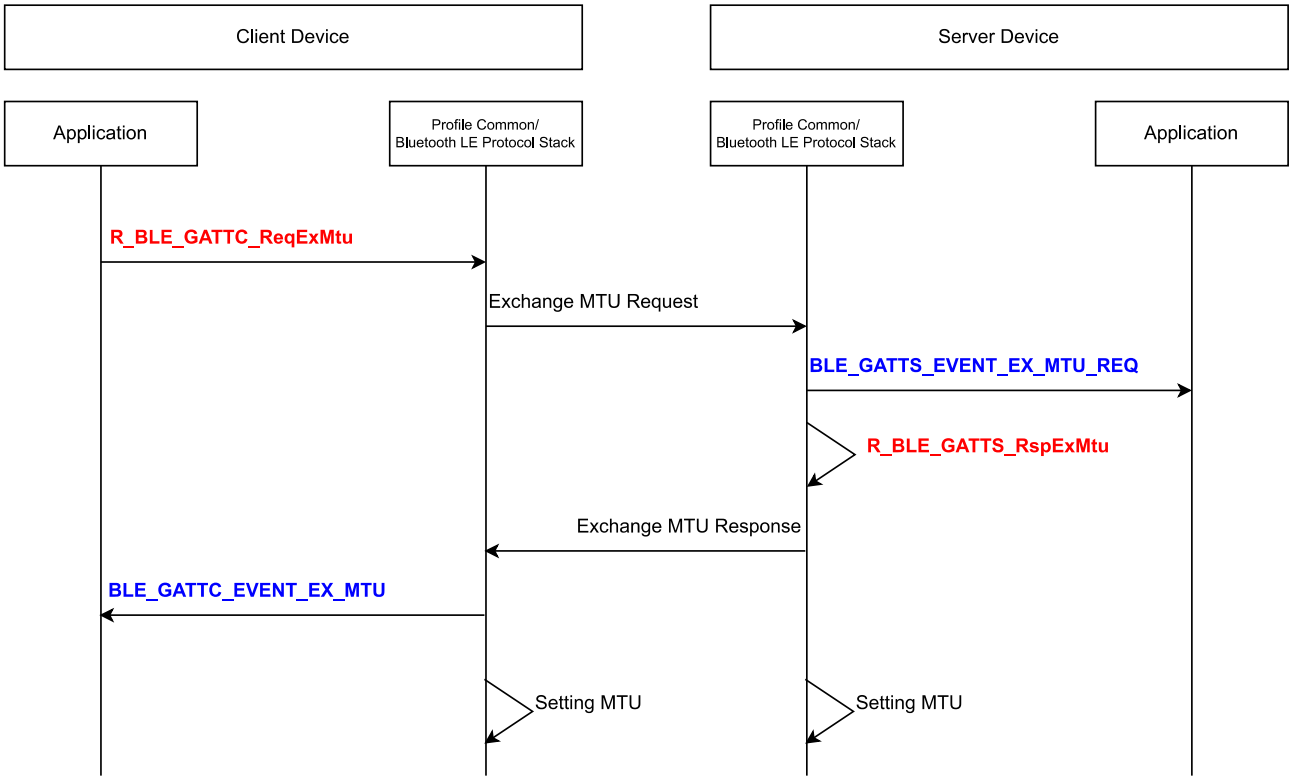


Figure 4.18 The sequence chart of MTU exchange

4.2.2.1 Implementation of Client

The MTU can be changed from the client. The API for exchanging MTUs is defined in the Bluetooth LE Protocol Stack.

```
ble_status_t R_BLE_GATTC_ReqExMtu(uint16_t conn_hdl, uint16_t mtu);
```

Figure 4.19 MTU exchange API

For example, you can implement service discovery efficiently by implementing it in the event callback gattc_cb of the GATT Client API as shown in Figure 4.14. The maximum MTU size supported by the device can be set on the Bluetooth LE Protocol Stack config screen.

```
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)
    {
        case BLE_GATTC_EVENT_CONN_IND:
        {
                uint16_t mtu = 247;
                R_BLE_GATTC_ReqExMtu(p_data->conn_hdl, mtu);
        } break;

        case BLE_GATTC_EVENT_EX_MTU_RSP:
        {
            /* Start discovery operation after mtu exchanged */
            R_BLE_DISC_Start(p_data->conn_hdl, gs_disc_entries,
                             ARRAY_SIZE(gs_disc_entries), disc_comp_cb);
        } break;
```

Figure 4.20 Implementation example of MTU exchange

4.2.2.2 Implementation of Server

The server sends the supported MTU to the client in the BLE_GATTS_EVENT_EX_MTU_REQ event.

This process is implemented in the R_BLE_SERVS_GattsCb function of the profile_cmn / r_ble_servs_if.c file in the profile common library.

No additional implementation is required.

```
void R_BLE_SERVS_GattsCb(uint16_t type, ble_status_t result, st_ble_gatts_evt_data_t *p_data)
{
    static uint16_t s_write_long_attr_hdl = BLE_GATT_INVALID_ATTR_HDL_VAL;

    switch (type)
    {
        case BLE_GATTS_EVENT_CONN_IND:
        case BLE_GATTS_EVENT_DISCONN_IND:
        break;

        case BLE_GATTS_EVENT_EX_MTU_REQ:
        {
            R_BLE_GATTS_RspExMtu(p_data->conn_hdl, BLE_PRF_MTU_SIZE);
        }
        break;
```

Figure 4.21 Implementation of MTU exchange processing in r_ble_servs_if.c file

4.2.3 Write Operation

The Write operation sends data from the client to the server and writes the value to the GATT database. The server responds to writes request. The client can send the data while confirming that the transmission to the server is complete.

The Write operation API performs Write Long operation if the data length to be sent is greater than MTU-3, and Write operation if it is MTU-3 or less.

Figure 4.16 shows the sequence chart for the Write operation. The red text is the function call, and the blue text is the application notification event.

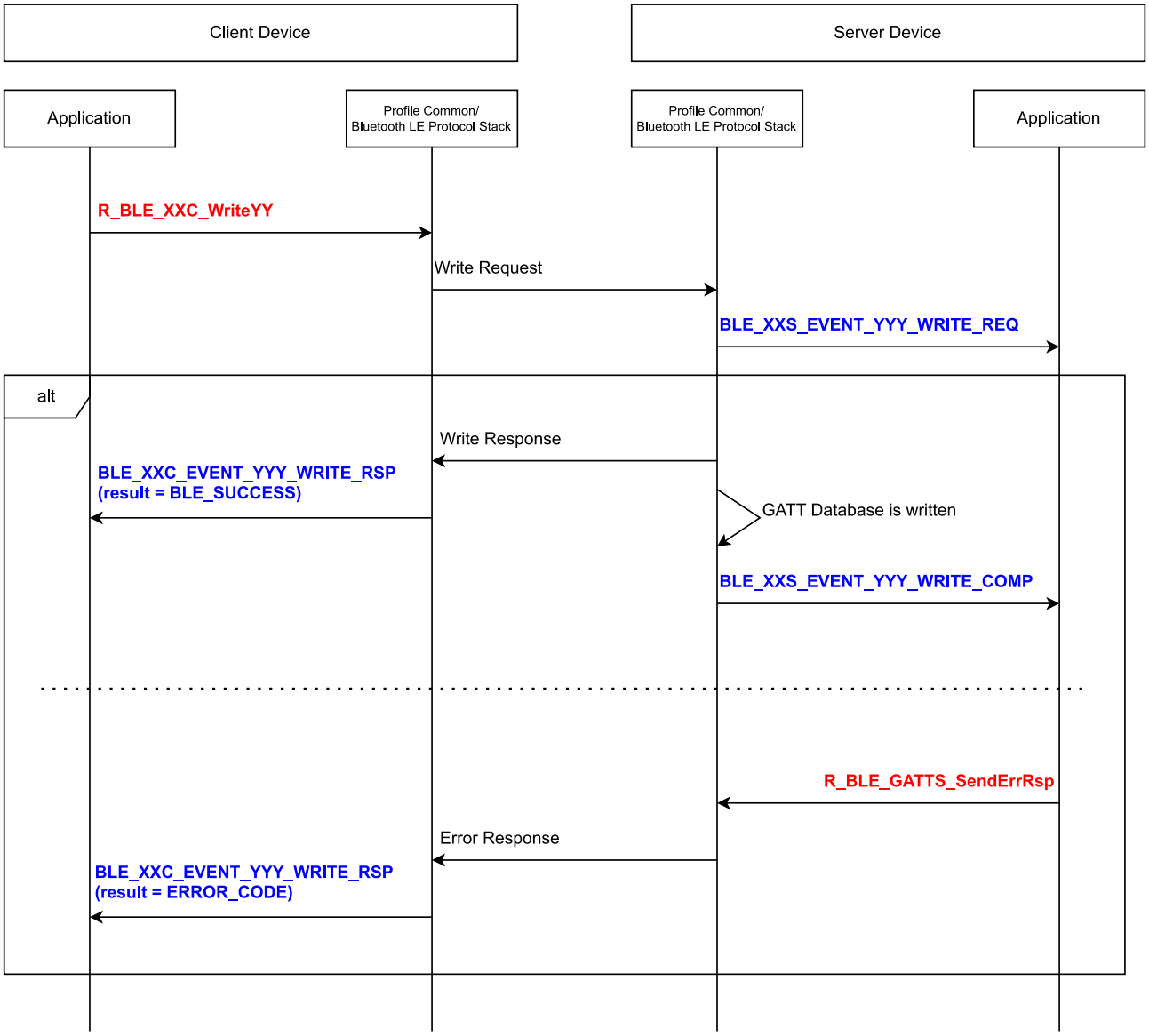


Figure 4.22 sequence chart for the Write operation

In the Write Long operation, the client divides the data to be sent to the server into pieces of a size that can be sent at once. After that, write all the data to the GATT database after transmission. The server stores the received data in the Prepare Write Queue. Figure 4.23 shows the sequence chart for Write Long operation. Functions are called in red, and events notified by the application are in blue.

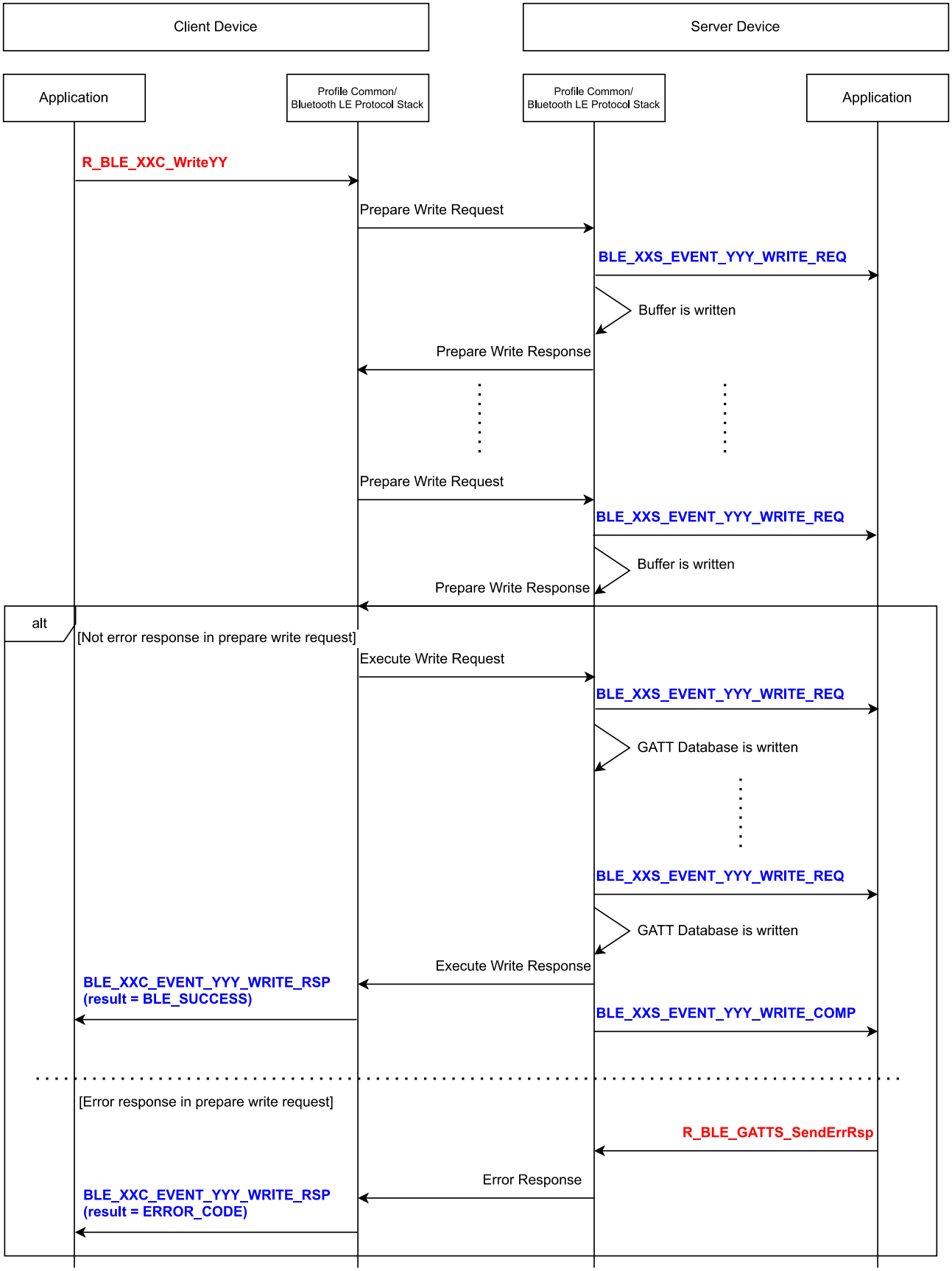


Figure 4.23 sequence chart for the Write Long operation

4.2.3.1 Implementation of Client

Write Operation API

The Write operation starts from the client. For the Write operation, use the Write Operation API implemented in the service API program. The arguments are the connection handle and the data sent to the target characteristic.

```
ble_status_t R_BLE_XXC_WriteYyy(uint16_t conn_hdl, const st_ble_xxc_yy_t *p_value);
ble_status_t R_BLE_XXC_WriteYyyZzz(uint16_t conn_hdl, const st_ble_xxc_yy_t *p_value);
```

Figure 4.24 Definition of Write operation API in service API program (r_ble_xxc.h)

Whether to perform Write operation or Write Long operation is determined by the data length set by the encode function of the characteristic passed as an argument.

Table 4.11 Relationship between transmission data length, MTU and operation performed

Relations between data length and MTU	Executing Operation
data_length <= MTU-3	Write Operation
data_length > MTU-3	Write Long Operation

The transmission data length of the write operation API is set by the p_gatt_value->value_len value (highlighted in yellow) of the characteristic encode function shown in Figure 4.25. When designing a characteristic that is not intended for Write Long operation, design so that this setting value does not exceed MTU-3.

```
static ble_status_t encode_st_ble_xxs_yyy_t
                        (const st_ble_xxs_yyy_t *p_app_value, st_ble_gatt_value_t *p_gatt_value)
{
    uint32_t pos = 0;

    BT_PACK_LE_16_BYTE(&p_gatt_value->p_value[pos], &p_app_value->field_1);
    pos += 2;

    BT_PACK_LE_16_BYTE(&p_gatt_value->p_value[pos], &p_app_value->field_2);
    pos += 2;

    p_gatt_value->value_len = pos;

    return BLE_SUCCESS;
}
```

Figure 4.25 Setting the transmission data length using the encode function

The Write operation API cannot be called consecutively. It can be called again after receiving the BLE_XXC_EVENT_YYY_WRITE_RSP event.

BLE_XXC_EVENT_YYY_WRITE_RSP Event

The client receives the result of the Write operation from the server.

When the GATT database is written, the result variable is notified with BLE_SUCCESS. If the server sends an Error Response, the result variable will be notified of the error code.

```
static void xxc_cb(uint16_t type, ble_status_t result, st_ble_servc_evt_data_t *p_data)
{
    switch(type)
    {
        case BLE_XXC_EVENT_YYY_WRITE_RSP:
        {
            if(result == BLE_SUCCESS)
            {
                /* GATT Database value in server is written. */
            }
            else
            {
                /* Error Response (0x30XX) or BLE_ERR_RSP_TIMEOUT (0x0011) */
            }

        } break;
    }
}
```

Figure 4.26 Implementation example of the event in Write operation

4.2.3.2 Implementation of Server

Setting the Prepare Write Queue

If you design a characteristic whose data length exceeds MTU min -3 bytes (20 bytes), it may be written to the GATT database by Write Long operation.

When accepting a Write Long operation from the client, prepare a temporary area (Prepare Write Queue) to hold the divided data. Write Long operation is supported by registering this Prepare Write Queue in Bluetooth LE Protocol Stack.

Prepare Write Queue processing is already added to app_main.c. QE for BLE code generation is set to hold 14 Prepare Write Requests for a 245-byte buffer when the number of simultaneous connections is 1.

Change the macro definition shown in Figure 4.27 according to your application.

```
/* Queue for Prepare Write Operation. Change if needed. */
#define BLE_GATTS_QUEUE_ELEMENTS_SIZE      (14)
#define BLE_GATTS_QUEUE_BUFFER_LEN         (245)
#define BLE_GATTS_QUEUE_NUM                (1)
```

Figure 4.27 Setting the Prepare Write Queue

If there is no possibility of using the Write Long operation, this processing can be disabled by defining the macro as shown in Figure 4.28.

If a Prepare Write Request is received without the Prepare Write Queue being registered, an error response is automatically sent from the Bluetooth LE Protocol Stack.

```
#define BLE_APP_PREPARE_WRITE_DISABLE (1)
```

Figure 4.28 Disabling the Prepare Write Queue

BLE_XXS_EVENT_YYY_WRITE_REQ Event

The write data sent by the client by the write operation is notified to the BLE_XXS_EVENT_YY_WRITE_REQ event on the server. You can see the value written by casting to a characteristic structure.

If the application evaluates the notified write data and does not accept the write to the GATT database, call the R_BLE_GATTS_SendErrRsp function. Calling this function sends an error response to the client and does not write it to the GATT database.

BLE_XXS_EVENT_YYY_WRITE_COMP Event

If the R_BLE_GATTS_SendErrRsp function is not called in the WRITE_REQ event, the Bluetooth LE Protocol Stack sends a write response to the client to write the value to the GATT database. The value of the GATT database after the write is completed is notified to the BLE_XXS_EVENT_YY_WRITE_COMP event. The server Write operation is complete with this event.

These events are notified to the xxs_cb function in app_main.c. Add application processing to this function. The implementation example is shown below. You can get the written value if the result variable is BLE_SUCCESS.

```
static void xxs_cb(uint16_t type, ble_status_t result, st_ble_servs_evt_data_t *p_data)
{
    switch(type)
    {
        case BLE_XXS_EVENT_YYY_WRITE_REQ:
        {
            if( BLE_SUCCESS == result)
            {
                /* Write Request Data*/
                st_ble_xxs_yyy_t *event_data = (st_ble_xxs_yyy_t*)p_data->p_param;

                if(event_data->field_1 != 0x01)
                {
                        /* Application can send Error Response */
                        uint16_t error_code = 0x3081;
                        R_BLE_GATTS_SendErrRsp(error_code);
                }
            }
        } break;

        case BLE_XXS_EVENT_YYY_WRITE_COMP:
        {
             if( BLE_SUCCESS == result)
             {
                /* Cast Application data*/
                st_ble_xxs_yyy_t *event_data = (st_ble_xxs_yyy_t *)p_data->p_param;

                /* Implement process in Write Complete */
                /* Application can execute next write operation */
            }
       } break;
    }
}
```

Figure 4.29 Implementation example of the event at the time of Write operation of the server

4.2.4 Write Without Response Operation

The Write Without Response operation writes a value from the client to the server to the GATT database. The server does not respond to writes to the GATT database. This is useful for sending data from the client to the server at high speed. Write Without Response cannot send data larger than MTU-3 bytes.

The sequence chart when the Write Without Response operation is performed is shown. The red text is the function call, and the blue text is the application notification event.

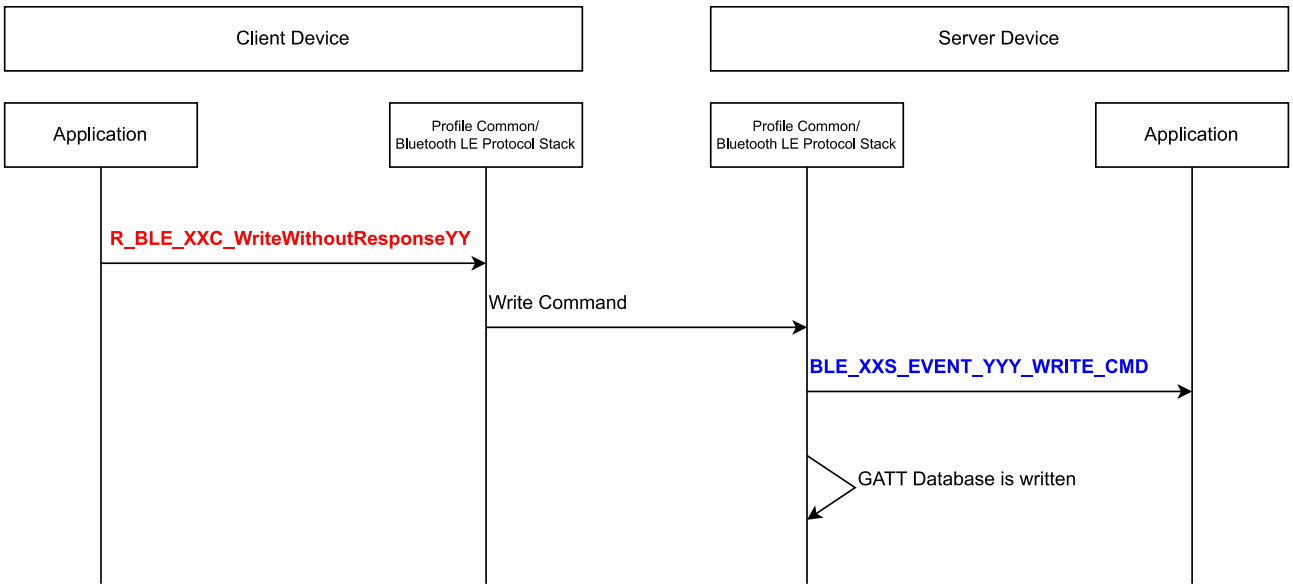


Figure 4.30 Sequence chart during Write Without Response operation

4.2.4.1 Implementation of Client

Write Without Response Operation API

The Write Without Response operation starts from the client. Use the Write Without Response operation API implemented in the service API program. Write Without Response operation is defined only characteristic.

The arguments are the connection handle and the characteristic value to write to.

```
ble_status_t R_BLE_XXC_WriteWithoutResponseYyy
                                (uint16_t conn_hdl, const st_ble_xxc_yy_t *p_value);
```

Figure 4.31 Definition of Write Without Response operation API in the service API program (r_ble_xxc.h)

No events are notified to the client since the API call.

4.2.4.2 Implementation of Server

BLE_XXS_EVENT_YYY_WRITE_CMD event

The data of the Write Without Response operation sent by the client is notified to the BLE_XXS_EVENT_YY_WRITE_CMD event of the server. You can see the value written by casting to a characteristic structure. You can get the written value if the result variable is BLE_SUCCESS.

```
static void xxs_cb(uint16_t type, ble_status_t result, st_ble_servs_evt_data_t *p_data)
{
    switch(type)
    {
        case BLE_XXS_EVENT_YYY_WRITE_CMD:
        {
            if(BLE_SUCCESS == result)
            {
                /* Cast Application data*/
                st_ble_xxs_yyy_t *event_data = (st_ble_xxs_yyy_t *)p_data->p_param;

                /* Implement process in Write Without Response */
                /* The GATT database value is not written when this event is notified. */
            }
        } break;
    }
}
```

Figure 4.32 Implementation example receiving event of Write Without Response of server

The value is written to the GATT database after this event processing is complete.

4.2.5 Read Operation

In the Read operation, the client reads the data in the GATT database. The server can also reject read requests.

The Read operation API performs Read operation when the data size of the GATT database is (MTU - 1) or less. If the data size of the GATT database is larger than MTU-1, read long operation is performed.

Figure 4.33 shows the sequence chart when the Read operation is performed. The red text is the function call, and the blue text is the application notification event.

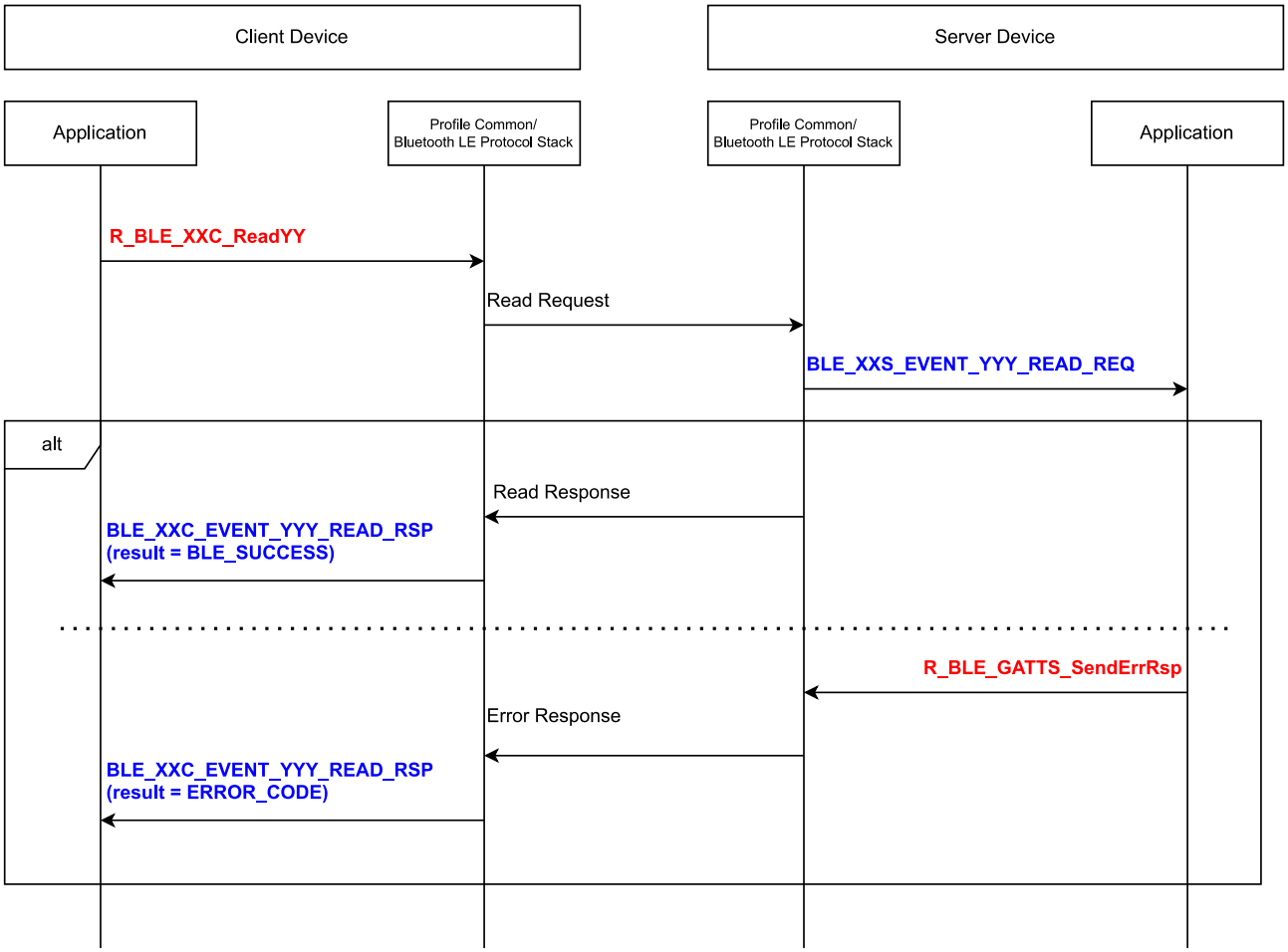


Figure 4.33 Sequence chart in Read operation

In the Read Long operation, the server sends to the client in units of the size of the characteristic data that can be sent at one time.

The server application generates read request events as many times as the size of the characteristic to be read / (MTU -1) (rounded up).

Figure 4.34 shows the sequence chart during Read Long operation. Functions are called in red, and events notified by the application are in blue.

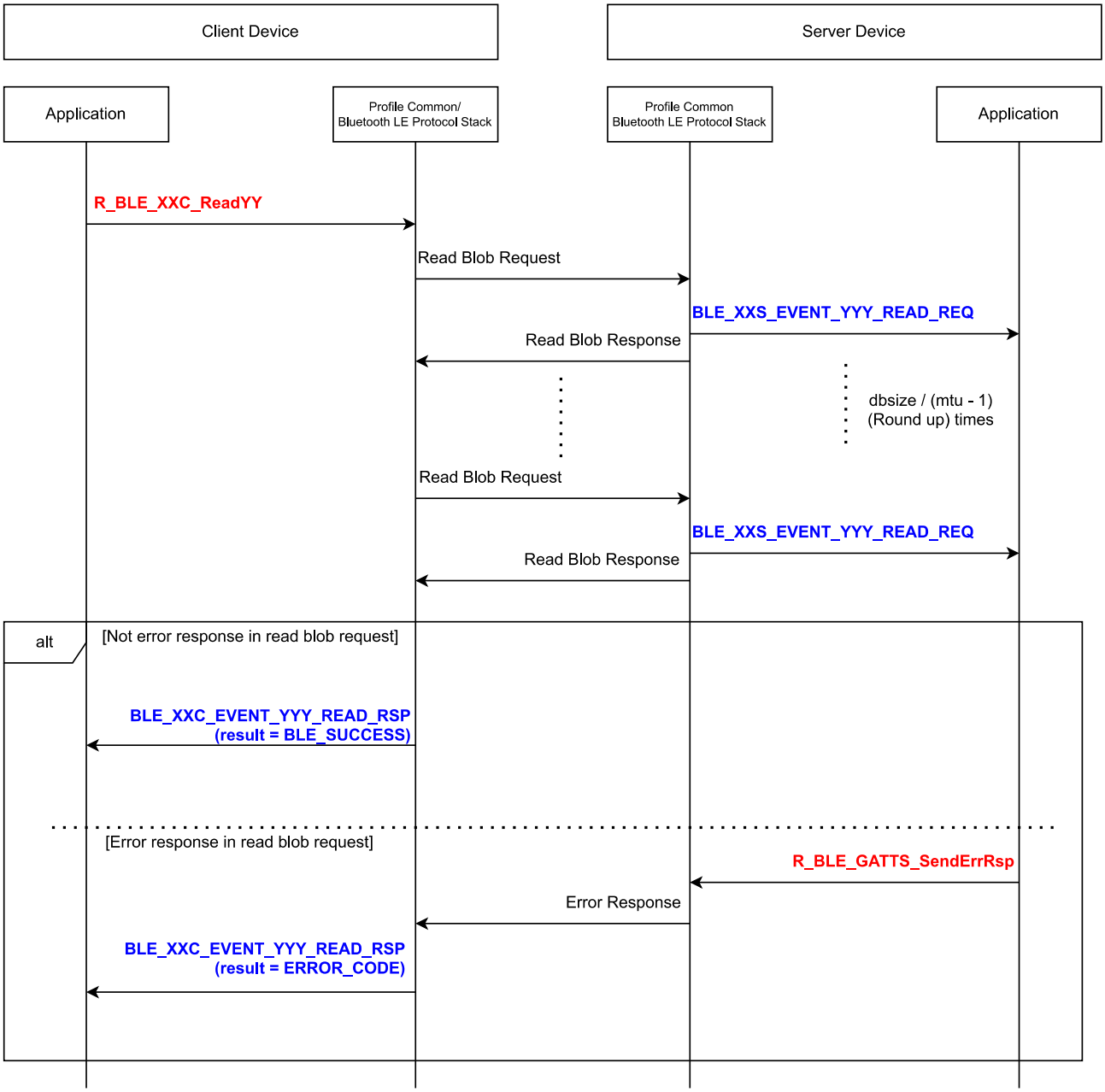


Figure 4.34 Sequence chart in Read Long operation

4.2.5.1 Implementation of Client

Read Operation API

The Read operation starts from the client. Use the Read behavior API implemented in the service API program. It is implemented for each characteristic and descriptor.

The only argument is the connection handle.

```
ble_status_t R_BLE_XXC_ReadYyy(uint16_t conn_hdl);
ble_status_t R_BLE_XXC_ReadYyyZzz(uint16_t conn_hdl);
```

Figure 4.35 Definition of Read operation API in service API program (r_ble_xxc.h)

Whether to perform Read operation or Read Long operation is determined by the characteristic db_size passed as an argument. Table 4.12 shows the relationship between the MTU and the operations performed.

Table 4.12 Relationship between db_size and MTU and actions performed

Relation between dbsize and MTU	Executing Operation
dbsize <= MTU-1	Read Operation
dbsize > MTU-1	Read Long Operation

For the db_size of the characteristic, refer to the db_size (highlighted in yellow) value registered in the characteristic definition structure in the r_ble_xxc.c file as shown in Figure 4.36.

```
static const st_ble_servc_char_info_t gs_yyy_char = {
    .uuid_128     = BLE_XXC_YYY_UUID,
    .uuid_type    = BLE_GATT_128_BIT_UUID_FORMAT,
    .app_size     = sizeof(st_ble_xxc_yyy_t),
    .db_size      = BLE_XXC_YYY_LEN,
    .char_idx     = BLE_XXC_YYY_IDX,
    .p_attr_hdls  = gs_yyy_char_ranges,
    .decode       = (ble_servc_attr_decode_t)decode_st_ble_xxc_yyy_t,
    .encode       = (ble_servc_attr_encode_t)encode_st_ble_xxc_yyy_t,
    .num_of_descs = ARRAY_SIZE(gspp_yyy_descs),
    .pp_descs     = gspp_yyy_descs,
};
```

Figure 4.36 Characteristic definition structure in r_ble_xxc.c file

BLE_XXC_EVENT_YYY_READ_RSP Event

The result of reading the data by the Read operation is notified to the BLE_XXC_YYY_EVENT_READ_RSP event. You can see the read value by casting it to a characteristic structure.

When an error response is received from the server, if the dynamic allocation of memory fails in the profile common library, or if the return value of the decode function is not BLE_SUCCESS, the error content is notified to the result variable.

```
static void xxc_cb(uint16_t type, ble_status_t result, st_ble_servc_evt_data_t *p_data)
{
    switch(type)
    {
        case BLE_XXC_EVENT_YYY_READ_RSP:
        {
            If (result == BLE_SUCCESS)
            {
                st_ble_xxc_yyy_t *event_data = (st_ble_xxc_yyy_t *)p_data->p_param;

                /*Implement application process. */

            }
        } break;
    }
}
```

Figure 4.37 Implementation example of the event at the time of Read operation of the client

4.2.5.2 Implementation of Server

BLE_XXS_EVENT_YYY_READ_REQ Event

Notified when a read is received from the client to the GATT database by Read operation. The value of the GATT database after event processing is sent to the client.

If you want to pass arbitrary data to the client read, update the GATT database with this event. Use the R_BLE_XXS_SetYyy function to update the GATT database.

```
static void xxs_cb(uint16_t type, ble_status_t result, st_ble_servs_evt_data_t *p_data)
{

    switch(type)
    {
        case BLE_XXS_EVENT_YYY_READ_REQ:
        {
            st_ble_xxs_yy_t new_value;

            R_BLE_XXS_SetYyy(&new_value);
        } break;
    }
}
```

Figure 4.38 How to update the GATT database

On the other hand, if it does not accept reads from the client, it calls the R_BLE_GATTS_SendErrRsp function and sends an Error Response.

```
static void xxs_cb(uint16_t type, ble_status_t result, st_ble_servs_evt_data_t *p_data)
{

    switch(type)
    {
        case BLE_XXS_EVENT_YYY_READ_REQ:
        {
            bool is_readable = false;

            /* Application does not read GATT Database */
            if ( !is_readable )
            {
                /* Application can send Error Response */
                uint16_t error_code = 0x3081;
                R_BLE_GATTS_SendErrRsp(error_code);
            }
        } break;
    }
}
```

Figure 4.39 Implementation example of the event in Read operation

4.2.6 Notify Operation

The Notify operation sends data from the server to the client.

To do the Notify operation, the client enables the Notify operation Client Configuration Characteristic Descriptor (CCCD).

The Notify operation cannot send data larger than MTU-3 bytes.

The sequence chart when performing the Notify operation is shown. The red text is the function call, and the blue text is the application notification event.



Figure 4.40 Sequence chart of Notify operation

4.2.6.1 Implementation of Client

Write to CCCD

First, write to CCCD. CCCD is represented by a 16-bit bit field. Write 0x0001 to CCCD to enable the Notify operation. It is macro-defined in the Bluetooth LE Protocol Stack with BLE_GATTS_CLI_CNFG_NOTIFICATION.

```
Function definition
ble_status_t R_BLE_XXXC_WriteYyyCliCnfg(uint16_t conn_hdl, const uint16_t *p_value)

Example of implementation
uint16_t cccd = BLE_GATTS_CLI_CNFG_NOTIFICATION;
R_BLE_XXXC_WriteYyyCliCnfg(conn_hdl, &cccd);
```

Figure 4.41 Write operation API to CCCD and implementation example in service API program (r_ble_xxs.h)

BLE_XXC_EVENT_YYY_HDL_VAL_NTF Event

When the client enables the Notify operation, the Notify operation sends data at any time on the server. The data is notified to BLE_XXC_EVENT_YYY_HDL_VAL_NTF.

You can see the received value by casting it to a characteristic structure.

```
static void xxc_cb(uint16_t type, ble_status_t result, st_ble_servc_evt_data_t *p_data)
{
    switch(type)
    {
        case BLE_XXC_EVENT_YYY_HDL_VAL_NTF:
        {
            if( BLE_SUCCESS == result )
            {
                st_ble_xxc_yyy_t *event_data = (st_ble_xxc_yyy_t)p_data->p_param;
                /*Implement application process. */
            }
        } break;
    }
}
```

Figure 4.42 Implementation example of the event when Notify of the client is operated

4.2.6.2 Implementation of Server

BLE_XXS_EVENT_YYY_CLI_CNFG_WRITE_COMP Event

This event occurs after the client has finished writing to CCCD. You will be notified of what was written to the CCCD.

In the following implementation example, the Notify setting of CCCD is confirmed and the data is sent to the client using the Notify operation API described later.

```
static void xxs_cb(uint16_t type, ble_status_t result, st_ble_servs_evt_data_t *p_data)
{

    switch(type)
    {
        case BLE_XXS_EVENT_YYY_CLI_CNFG_WRITE_COMP:
        {
            uint16_t cccd = *(uint16_t *)p_data->p_param;

            if( (cccd & BLE_GATTS_CLI_CNFG_NOTIFICATION) == BLE_GATTS_CLI_CNFG_NOTIFICATION)
            {
                st_ble_xxs_yyy_t notify_value;
                R_BLE_XXS_NotifyYyy(p_data->conn_hdl, &notify_value);
            }
        } break;
    }
}
```

Figure 4.43 Start of Notify in CCCD write completion event

Notify Operation API

Notify operation can be sent from the server whenever CCCD Notify is enabled. The API that operates Notify is as follows. Notify operation is defined only for characteristic.

The arguments are the connection handle and the value of the characteristic to be notified.

```
ble_status_t R_BLE_XXS_NotifyYyy(uint16_t conn_hdl, const st_ble_xxs_yyy_t *p_value);
```

Figure 4.44 Notify operation API definition in service API program (r_ble_xxs.h)

If Notify of CCCD is not enabled, the return value will be BLE_ERR_INVALID_OPERATION (0x0009) and data will not be sent by Notify.

No events are notified to the server since the API call.

4.2.7 Indicate Operation

The Indicate operation sends data from the server to the client. The Indicate operation sends a Handle Value Confirmation from the client to the server telling you that the data has been received. This is useful when you want to confirm the data transmission from the server to the client.

To perform the Indicate operation, the client enables the Indicate operation in the characteristic Client Configuration Characteristic Descriptor (CCCD).

The Indicate operation cannot send data larger than MTU-3 bytes.

The sequence chart in Indicate operation is shown. The red text is the function call, and the blue text is the application notification event.



Figure 4.45 Sequence char in indicate operation

4.2.7.1 Implementation of Client

Write to CCCD

First, write to CCCD. CCCD is represented by a 16-bit bit field. Write 0x0002 to CCCD to enable the Indicate operation. It is macro-defined in the Bluetooth LE Protocol Stack with BLE_GATTS_CLI_CNFG_INDICATION.

```
Function definition
ble_status_t R_BLE_XXXC_WriteYyyCliCnfg(uint16_t conn_hdl, const uint16_t *p_value)

implementation example
uint16_t cccd = BLE_GATTS_CLI_CNFG_INDICATION;
R_BLE_XXXC_WriteYyyCliCnfg(conn_hdl, &cccd);
```

Figure 4.46 Write operation API to CCCD and implementation example in service API program (r_ble_xxs.h)

BLE_XXS_EVENT_YYY_HDL_VAL_IND Event

When the client enables the Indicate operation, the data is sent by the Indicate operation at any time on the server. The data is notified to BLE_XXS_EVENT_YYY_HDL_VAL_IND.

You can see the received value by casting it to a characteristic structure.

```
static void xxc_cb(uint16_t type, ble_status_t result, st_ble_servc_evt_data_t *p_data)
{
    switch(type)
    {
        case BLE_XXC_EVENT_YYY_HDL_VAL_IND:
        {
            if( BLE_SUCCESS == result )
            {
                st_ble_xxc_yyy_t *event_data = (st_ble_xxc_yyy_t)p_data->p_param;
                /*Implement application process. */
            }
        } break;
    }
}
```

Figure 4.47 Implementation example of event in Indicate operation

The Bluetooth LE Protocol Stack automatically responds to the Handle Value Confirmation of the receipt confirmation.

4.2.7.2 Implementation of Server

BLE_XXS_EVENT_YYY_CLI_CNFG_WRITE_COMP Event

This event occurs after the client has finished writing to CCCD. You will be notified of what was written to the CCCD.

In the following implementation example, the Indicate setting of CCCD is confirmed and the data is sent to the client using the Indicate operation API described later.

```
static void xxs_cb(uint16_t type, ble_status_t result, st_ble_servs_evt_data_t *p_data)
{

    switch(type)
    {
        case BLE_XXS_EVENT_YYY_CLI_CNFG_WRITE_COMP:
        {
            uint16_t cccd = *(uint16_t *)p_data->p_param;

            if( (cccd & BLE_GATTS_CLI_CNFG_INDICATION) == BLE_GATTS_CLI_CNFG_ INDICATION)
            {
                st_ble_xxs_yyy_t notify_value;
                R_BLE_XXS_IndicateYyy(p_data->conn_hdl, &notify_value);
            }
        } break;
    }
}
```

Figure 4.48 Start of Indicate in CCCD write completion event

Indicate Operation API

Indicate operations can be sent from the server when CCCD Indication bit is enabled. The API for Indicate operation is as follows. Indicate operation is defined only for characteristic.

The arguments are the connection handle and the characteristic value of the Indicate target.

If CCCD Indicate is not enabled, the return value will be BLE_ERR_INVALID_OPERATION (0x0009) and data will not be sent by Indicate.

```
ble_status_t R_BLE_XXS_IndicateYyy(uint16_t conn_hdl, const st_ble_xxs_yyy_t
*p_value);
```

Figure 4.49 Definition of Indicate operation API in service API program (r_ble_xxc.h)

The Indicate operation API cannot be called consecutively. It can be called again after receiving the BLE_XXS_EVENT_YYY_HDL_VAL_CNF event.

BLE_XXS_EVENT_YYY_HDL_VAL_CNF Event

When the Indicate operation sends data from the server to the client, the client sends the confirmation packet to the server. This event is announced as a BLE_XXS_EVENT_YYY_HDL_VAL_CNF event.

This receipt allows the server to confirm that the data has arrived at the client.

```
static void xxc_cb(uint16_t type, ble_status_t result, st_ble_servc_evt_data_t *p_data)
{
    switch(type)
    {
        case BLE_XXC_EVENT_YYY_HDL_VAL_IND:
        {
            if( BLE_SUCCESS == result )
            {
                st_ble_xxc_yyy_t *event_data = (st_ble_xxc_yyy_t)p_data->p_param;
                /*Implement application process. */
            }
        } break;
    }
}
```

Figure 4.50Implementation example of the event in Indicate operation

4.3 GATT Database (gatt_db.c / gatt_db.h)

It implements a GATT database consisting of services used as [Server]. No need to change.

The gatt_db.h file expands the size of attribute value and the attribute handle of the characteristic and descriptor in macro format.

```
typedef enum
{
    BLE_INVALID_ATTR_HDL = 0x0000,
    BLE_GAPS_DECL_HDL = 0x0001,
    BLE_GAPS_DEV_NAME_DECL_HDL = 0x0002,
    BLE_GAPS_DEV_NAME_VAL_HDL = 0x0003,
    BLE_GAPS_APPEARANCE_DECL_HDL = 0x0004,
    BLE_GAPS_APPEARANCE_VAL_HDL = 0x0005,
    BLE_GAPS_PER_PREF_CONN_PARAM_DECL_HDL = 0x0006,
    BLE_GAPS_PER_PREF_CONN_PARAM_VAL_HDL = 0x0007,
    BLE_GAPS_CENT_ADDR_RSLV_DECL_HDL = 0x0008,
    BLE_GAPS_CENT_ADDR_RSLV_VAL_HDL = 0x0009,
    BLE_GAPS_RSLV_PRIV_ADDR_ONLY_DECL_HDL = 0x000A,
    BLE_GAPS_RSLV_PRIV_ADDR_ONLY_VAL_HDL = 0x000B,
    BLE_GATS_DECL_HDL = 0x000C,
    BLE_GATS_SERV_CHGED_DECL_HDL = 0x000D,
    BLE_GATS_SERV_CHGED_VAL_HDL = 0x000E,
    BLE_GATS_SERV_CHGED_CLI_CNFG_DESC_HDL = 0x000F,
    BLE_XXS_DECL_HDL = 0x0010,
    BLE_XXS_YYY_DECL_HDL = 0x0011,
    BLE_XXS_YYY_VAL_HDL = 0x0012,
    BLE_XXS_YYY_CLI_CNFG_DESC_HDL = 0x0013,
    BLE_XXS_YYY_ZZZ_DESC_HDL = 0x0014,
} e_ble_attr_hdl_t;

#define BLE_GAPS_DEV_NAME_LEN (128)
#define BLE_GAPS_APPEARANCE_LEN (2)
#define BLE_GAPS_PER_PREF_CONN_PARAM_LEN (8)
#define BLE_GAPS_CENT_ADDR_RSLV_LEN (1)
#define BLE_GAPS_RSLV_PRIV_ADDR_ONLY_LEN (1)
#define BLE_GATS_SERV_CHGED_LEN (4)
#define BLE_GATS_SERV_CHGED_CLI_CNFG_LEN (2)
#define BLE_XXS_YYY_LEN (4)
#define BLE_XXS_YYY_CLI_CNFG_LEN (2)
#define BLE_XXS_YYY_ZZZ_LEN (2)
```

Figure 4.51 GATT database macro definition

The gatt_db.c file implements the GATT database according to the specifications of the Bluetooth LE Protocol Stack. In addition, the service list of the profile designed by QE for BLE is displayed in comment format as a GATT database that implements it.

```
/**
 *  GATT DATABASE QUICK REFERENCE TABLE:
 *  Abbreviations used for PROPERTIES:
 *      BC = Broadcast
 *      RD = Read
 *      WW = Write Without Response
 *      WR = Write
 *      NT = Notification
 *      IN = Indication
 *      RW = Reliable Write
 *
 *  HANDLE | ATT_TYPE         | PROPERTIES  | ATT_VALUE                      | DEFINITION
 *  ===========================================================================================
 *  GAP Service
 *  ===========================================================================================
 *  0x0001 | 0x28,0x00        | RD          | 0x00,0x18                     | GAP Service Declaration
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x0002 | 0x28,0x03        | RD          | 0x0A,0x03,0x00,0x00,0x2A       | Device Name characteristic…
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x0003 | 0x00,0x2A        | RD,WR       | 0x00,0x00,0x00,0x00,0x00... | Device Name characteristic …
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x0004 | 0x28,0x03        | RD          | 0x02,0x05,0x00,0x01,0x2A       | Appearance characteristic …
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x0005 | 0x01,0x2A        | RD          | 0x00,0x00                     | Appearance characteristic …
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x0006 | 0x28,0x03        | RD          | 0x02,0x07,0x00,0x04,0x2A       | Peripheral Preferred …
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x0007 | 0x04,0x2A        | RD          | 0x00,0x00,0x00,0x00,0x00,0x00... | Peripheral Preferred …
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x0008 | 0x28,0x03        | RD          | 0x02,0x09,0x00,0xA6,0x2A       | Central Address Resolution …
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x0009 | 0xA6,0x2A        | RD          | 0x00                          | Central Address Resolution …
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x000A | 0x28,0x03        | RD          | 0x02,0x0B,0x00,0xC9,0x2A       | Resolvable Private Address …
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x000B | 0xC9,0x2A        | RD          | 0x00                          | Resolvable Private Address …
 *  ===========================================================================================
 *  GATT Service
 *  ===========================================================================================
 *  0x000C | 0x28,0x00        | RD          | 0x01,0x18                     | GATT Service Declaration
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x000D | 0x28,0x03        | RD          | 0x20,0x0E,0x00,0x05,0x2A       | Service Changed …
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x000E | 0x05,0x2A        | IN          | 0x00,0x00,0x00,0x00            | Service Changed …
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x000F | 0x02,0x29        | RD,WR       | 0x00,0x00                     | Client Characteristic …
 *  ===========================================================================================
 *  prof dev xx Service
 *  ===========================================================================================
 *  0x0010 | 0x28,0x00        | RD          | 0x50,0x44,0x02,0xb7,0xaf,0xf8... | prof dev xx Service …
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x0011 | 0x28,0x03        | RD          | 0x3E,0x12,0x00,0x2d,0x1f,0x35... | prof dev yyy char…
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x0012 | 0x2d,0x1f,0x35... | RD,WW,WR... | 0x00,0x00,0x00,0x00            | prof dev yyy char value…
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x0013 | 0x02,0x29        | RD,WR       | 0x00,0x00                     | Client Characteristic…
 *  -------+------------------+-------------+-------------------------------+---------------
 *  0x0014 | 0xb9,0xfd,0x08... | RD,WR       | 0x00,0x00                     | prof dev zzz Descriptor …
 *  ===========================================================================================

  */
```

Figure 4.52 GATT database structure comment example in gatt_db.c file

5. Build and Run program

This chapter describes the points to note when building the code generated from QE for BLE for each MCU.

To build and debug the project in e² studio, refer to "e² studio User's Manual Getting Started Guide (R20UT4204)".

5.1 RX23W

If you create a new project, the code generated from QE for BLE can be executed without changing the settings.

If you create a new project with a combination of BLE FIT module version 2.50 or later and QE for BLE version 1.50 or earlier, the profile common part will not be added to the project. Please refer to the following site and update QE for BLE to the latest environment.

https://www.renesas.com/qe-ble

File contention may occur when developing based on a sample project in the BLE FIT module version 2.30 or earlier.

If the following folders exist in the project, delete them.

- src/smc_gen/Config_BLE_PROFILE
- src/smc_gen/r_ble_qe_utility

5.1.1 Migrating Profile Data due to Unifying a Plug-in

If Figure 5.1 is displayed in the procedure in Section 3.1, perform data migration as described in this section.

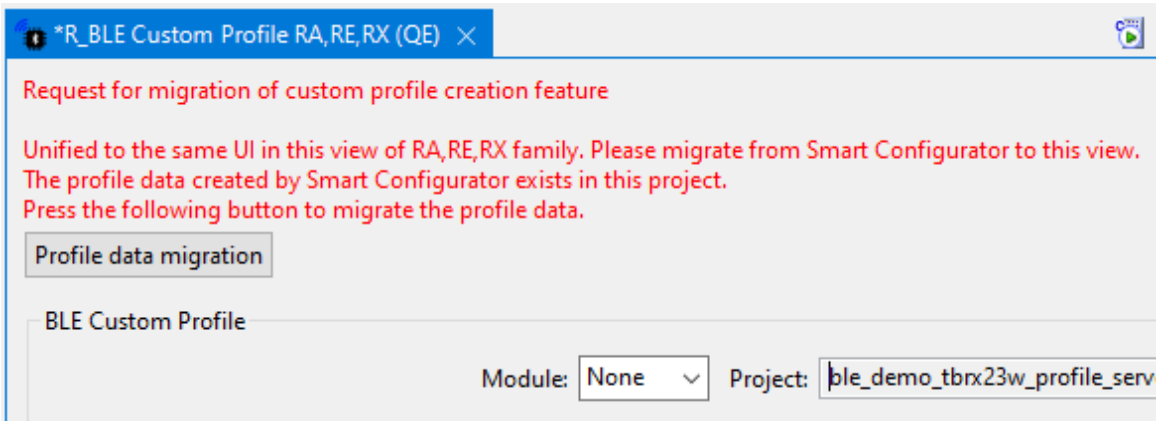


Figure 5.1 Request for migration of QE for BLE

Performs smart configurator data migration, component removal and code generation. Data migration is automatic. Follow the instructions in the profile migration dialog that pops up to remove the component.

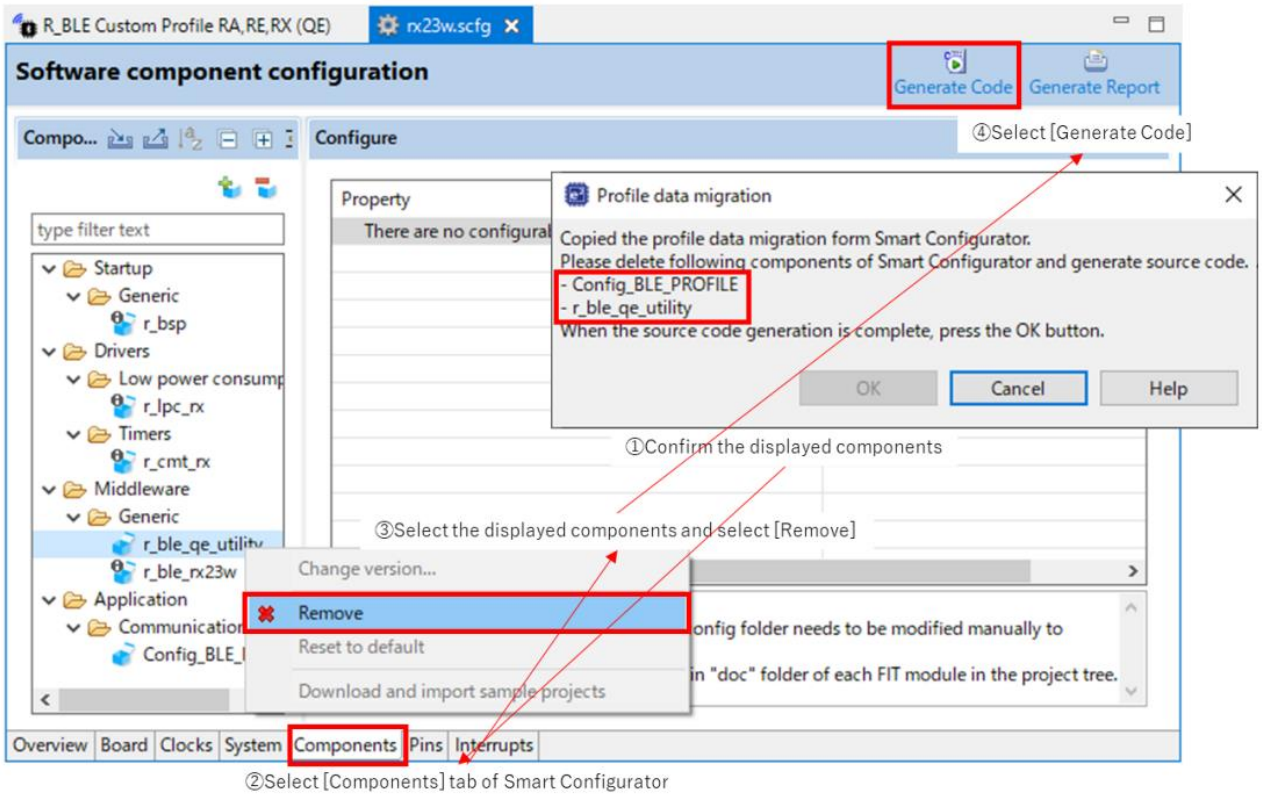


Figure 5.2 Component removal procedure

The migration is complete when the message below is displayed.

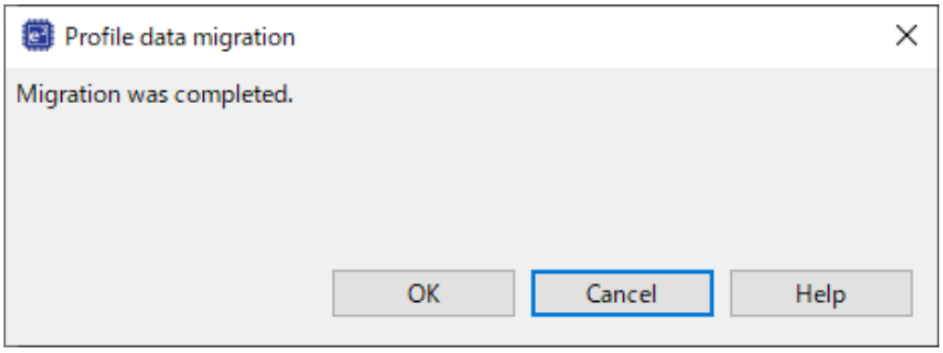


Figure 5.3 Migration complete message

Uninstall of plug-ins for RX family QE for BLE[RX] V1.0.0 or V1.1.0 is not used, so please uninstall it.

1. Select [Help -> About e^2 studio] to open the [About e^2 studio] dialog box.
2. Click the [Installation Details] button to open the [e^2 studio Installation Details] dialog box.
3. Select [Renesas QE for BLE[RX]] displayed on the [Installed Software] tabbed page and click the [Uninstall…] button to open the [Uninstall] dialog box.
4. Check the displayed information and click the [Finish] button. When prompted to restart e^2 studio, restart it.

Notes

- Do not add the component deleted by Smart Configurator again.
- If the following dialog is displayed when migrate the profile data, overwrite the profile data code-generated by [R_BLE Custom Profile RA, RE, RX (QE)] with the profile data code-generated by Smart Configurator.

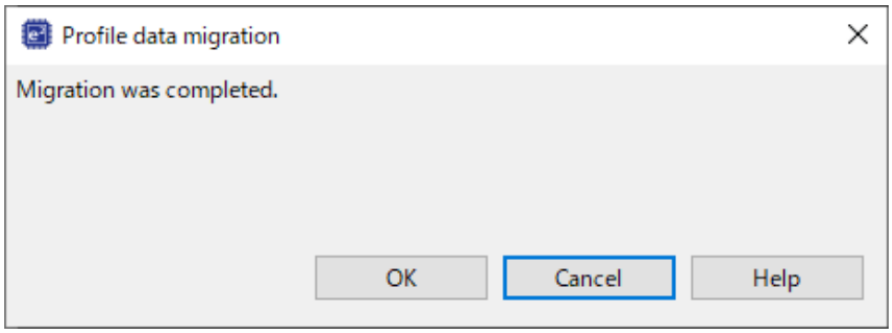


Figure 5.4 Overwrite confirmation dialog

5.2 RA4W1

If you want to run the code generated from QE for BLE, run app_main function in the hal_entry.c file. An implementation example is shown below.

```
extern void app_main(void);

void hal_entry(void) {
    /* TODO: add your own code here */
    app_main();
}
```

Figure 5.5 Calling app_main function in hal_entry.c

If you are developing a profile based on a project that uses FreeRTOS or AzureRTOS, the following build error may occur when building the code generated from QE for BLE.

- Error [Pe020]: identifier "g_ble_abs0_ctrl" is undefined.
- Error [Pe020]: identifier "g_ble_abs0_cfg" is undefined.

This is since the external reference declaration of the setting variable of the abstraction API module is not made.

These declarations are written in the task header files that the Bluetooth LE Protocol Stack contains. In the sample project it is “ble_core_task”.

Include this header file in the app_main.c file.

```
#include "ble_core_task.h"
```

Figure 5.6 Example of inclusion of header file for Bluetooth LE task (for sample project)

5.3 RE01B

If you use the code generated from QE for BLE based on the Bluetooth LE communication project, you can run it without changing the settings.

6. Notice

6.1 Implementation of multiple services

When implementing multiple services, take care of the characteristic and descriptor code sizes contained in the service. If the code size exceeds the RAM/ROM size of target device, it cannot be compiled.

6.2 Implementation of same service

If you add multiple same SIG adopted services to a profile, QE for BLE cannot correctly generate programs due to problem such as conflicts of file name. Therefore, if you want to implement multiple same services, you need to add only one service as SIG adopted service and add the others as custom service on QE for BLE. For example, assume that you want to implement 2 Human Interface Device Service (HIDS), which is SIG adopted service.

First, you need to add 2 HIDS as SIG adopted service in QE for BLE. Change 1 of these HIDS from SIG adopted service to custom service. To change from SIG adopted service to custom service, click the customize button on the service setting screen. You need to make the following changes to the service that you changed to the custom service:

1. Change [UUID] of service so that service UUID matches between the same service. If you want to treat the custom service as SIG adopted service, set [UUID] to 16bit and change the value.
2. Change [abbreviation] of service so that it does not conflict with other services. This is to prevent conflicts on file name, function name, and variable name because [abbreviation] is used for them. Similarly, set [abbreviation] of characteristic and descriptor to string which do not conflict with others.

Setting on QE for BLE is over. Figure6.1 shows how to configure multiple SIG adopted services on QE for BLE.

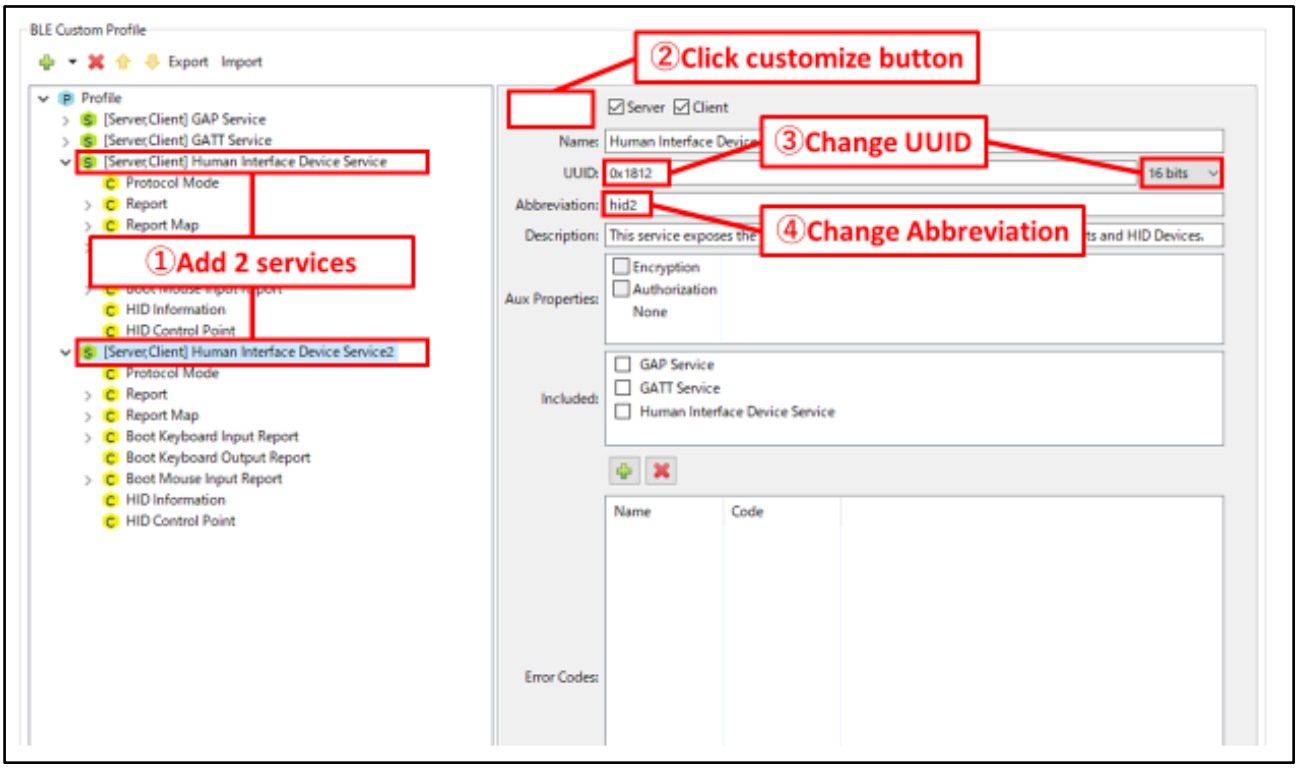


Figure6.1 Configure multiple service on QE for BLE

Because the program generated from custom services are skeleton program, it is necessary to implement the actual state of process. Program generated from SIG standard services has same mechanism and is implemented according to the defined specification, so refer this program to implement skeleton program of custom service. The parts that must be implemented vary from service to service, but in many cases, following implementation is needed:

1. Implements encode/decode function. Since the structure of the characteristic or descriptor remains the same, you can port many parts of implementation. Beware of differences in function name and variable name.
2. Implements callback function in service. This is used when you want to automatically return error for invalid value written or automatically return certain value for specific value written. Implementation is needed according to functionality of each service.

In addition, if the profile has at least one service selected as a [client] except the GAP service, discovery operation program using discovery library is implemented in file app_main.c. Among them, the array gs_disc_entries[] defines UUID and discovery callback function for each service included in profile. To discover services those have same service UUID, you need to add element idx which is index number for them. The following is example of implementing a program with 2 HIDS.

```
/* Human Interface Device Service UUID */
static uint8_t HIDC_UUID[] = { 0x12, 0x18 }; //HIDS specific service UUID
/* Human Interface Device Service2 UUID */
static uint8_t HID2C_UUID[] = { 0x12, 0x18 }; //Same service UUID

/* Service discovery parameters */
static st_ble_disc_entry_t gs_disc_entries[] = {
    {
            .p_uuid        = HIDC_UUID,
            .uuid_type     = BLE_GATT_16_BIT_UUID_FORMAT,
            .serv_cb       = R_BLE_HIDC_ServDiscCb,
            /* Add member [idx] */
            .idx           = 0, /* Set index number if service UUID is same */
    },
    {
      .p_uuid        = HID2C_UUID,
      .uuid_type     = BLE_GATT_16_BIT_UUID_FORMAT,
      .serv_cb       = R_BLE_HID2C_ServDiscCb,
      /* Add member [idx] */
      .idx           = 1, /* Set index number if service UUID is same */
    },
};
```

Figure6.2 Example of implementing 2 HIDSs

6.3 Implementation of secondary service

QE for BLE treats all services as primary services. Therefore, if you want to use secondary service, you need to modify the generated program. How to change program is different on the server side and client side.

Server Side

QE for BLE generates GATT database which stores information of services which have check in [server]. Since QE for BLE treats all services as primary service, generated GATT database defines all services as primary service. You need to modify service information defined in GATT database.

Change the array gs_gatt_type_table[ ] defined in file gatt_db.c. In this array, following 2 point needs to be changed:

1. Add definition for secondary service. Refer to the other elements of the array and create element that has [UUID_Offset] is 2 and correct attribute handles of secondary services.
2. Change element which defines [Primary Service Declaration]. Change it to specify the correct attribute handle.

The following is the example of implementation on array gs_gatt_type_table[ ].

```
static const st_ble_gatts_db_uuid_cfg_t gs_gatt_type_table[] =
{
    /* 0 : Primary Service Declaration */
    {
        /* UUID Offset */
        0,
        /* First Occurrence for type */
      /* Change this value to proper handle */
        0x000C,
        /* Last Occurrence for type */
      /* Change this value to proper handle */
        0x0026,
    },

  /* Add from here */
    /* 2 : Secondary Service Declaration */
    {
        /* UUID Offset */
       /* set 2 for this value */
        2,
        /* First Occurrence for type */
      /* Change this value to proper handle */
        0x0010,
        /* Last Occurrence for type */
      /* Change this value to proper handle */
        0x0000,
    },
  /* Add until here */

}
```

Figure6.3 GATT database of secondary service (1)

Also, change array gs_gatt_db_attr_table[ ]. n this array, following 2 point needs to be changed:

1. Change [UUID_Offset] section of service declaration which you want to change to secondary service. [UUID_offset] determines attribute type of data. In [UUID_Offset], 0 stands for primary service and 2 stands for secondary service. Set 2 for [UUID_Offset].
2. change element [Next Attribute Type Index] to indicate correct attribute handle. [Next Attribute Type Index] holds attribute handle of next data which has same attribute type. If modified data was the last data with same attribute type, enter 0x0000 for [Next Attribute Type Index].

The example of implementation on array gs_gatt_type_table[ ] is shown on Figure6.4.

Note: Make sure that the service which you changed to secondary service is included from at least one primary service.

```
static const st_ble_gatts_db_attr_cfg_t gs_gatt_db_attr_table[] =
{
  /* Handle: 0x000C */
    /* GATT Service: Primary Service Declaration */
    {
        /* Properties */
        BLE_GATT_DB_READ,
        /* Auxiliary Properties */
        BLE_GATT_DB_FIXED_LENGTH_PROPERTY,
        /* Value Size */
        2,

        /* Next Attribute Type Index */
        /* change this value to handle of next primary service declaration */
        0x0026,      /* 0x0010 → 0x0026 */

        /* UUID Offset */
        0,
        /* Value */
        (uint8_t *)(gs_gatt_const_uuid_arr + 20),
    },


  /* Example: Secondary Service Declaration */
  /* Handle: 0x0010 */
  /* Human Interface Device Service: Primary Service Declaration */
  {
       /* Properties */
       BLE_GATT_DB_READ,
       /* Auxiliary Properties */
       BLE_GATT_DB_FIXED_LENGTH_PROPERTY,
       /* Value Size */
       2,

      /* Next Attribute Type Index */
      /* Change this value to proper handle */
      /* Last secondary service declared: 0x0000 */
      /* Not last secondary service declared: handle of next secondary service declaration */
       0x0000,       /* 0x0026 → 0x0000 */

      /* UUID Offset */
      /* Change this value to proper Attribute type */
      /* Primary service declaration: 0 */
      /* Secondary service declaration: 2 */
       2,     /* 0 → 2 */

       /* Value */
       (uint8_t *)(gs_gatt_const_uuid_arr + 26),
    },


    /* Handle: 0x0026 */
    /* Human Interface Device Service2: Primary Service Declaration */

}
```

Figure6.4 GATT database of secondary service (2)

Client Side

If the profile has at least one service selected as a [client] except the GAP service, QE for BLE generate the code to perform the discovery operation. Generated program performs discovery operation only to primary service using Discovery Library provided by BLE Protocol Stack. When you need to discovery secondary service, perform discovery operation as the included service because secondary service is included from other primary service, refer to [6.4 Implementation of discovery operation about included service]. When you perform secondary service discovery operation to debug, call R_BLE_GATTC_DiscAllSecondServ() in GATT Client API provided by BLE Protocol Stack.

For more information about GATT Client API, refer the [R_BLE API document (r_ble_api_spec.chm)] that is included in BLE FIT module.

Table 6.1 Documentation for GATT Client API

MCU	Documents
RX23W	"R_BLE API document (r_ble_api_spec.chm)" attached to the BLE FIT module
RA4W1	“RA Flexible Software Package Documentation”.
RE01B	"R_BLE API document (r_ble_api_spec.chm)" attached to "Bluetooth Low Energy sample code (R01AN5606)"

6.4 Implementation of discovery operation about included service

Specifying included service

If the profile has at least one service selected as a [client] except the GAP service, QE for BLE generate the code to perform the discovery operation. Generated program performs discovery operation only to primary service using Discovery Library provided by BLE Protocol Stack.

If service has specific service as an included service, you need to confirm its structure to perform discovery operation to specific service. Discovery library provide feature to perform discovery operation confirming this structure. Discovery library perform discovery operation to attribute handle range that included service declaration has if included service entries are registered in discovery entry of parent service. Modify the variable gs_disc_entries in the app_main.c as the following, in order to register included service entries to discovery entry of parent service.

```
/*PRIMARY service entry */
static st_ble_disc_entry_t gs_disc_entries[] =
{
      {
              /*Weight Scale service disc entry */
              .p_uuid = (uint8_t *)BLE_WSC_UUID,
              .uuid_type = BLE_GATT_16_BIT_UUID_FORMAT,
              .serv_cb = R_BLE_WSC_ServDiscCb,
      },
      {       /*Body Composition service disc entry */
              .p_uuid = (uint8_t *)BLE_BCC_UUID,
              .uuid_type = BLE_GATT_16_BIT_UUID_FORMAT,
              .serv_cb = R_BLE_BCC_ServDiscCb,
      },
};
```

Figure 6.5 Code generated by QE for BLE

```
/*Add INCLUDE service entry*/
static st_ble_disc_entry_t gs_disc_wsc_inc_entries[] =
{
{/*Body Composition service disc entry AS A INCLUDE SERVICE IN WSS*/
              .p_uuid = (uint8_t *)BLE_BCC_UUID,
              .uuid_type = BLE_GATT_16_BIT_UUID_FORMAT,
               .serv_cb = R_BLE_BCC_ServDiscCb,
              .num_of_inc_servs = 0,
      },
};

/*PRIMARY service entry */
static st_ble_disc_entry_t gs_disc_entries[] =
{
/*Weight Scale service disc entry as a primary service*/
      {
              .p_uuid = (uint8_t *)BLE_WSC_UUID,
              .uuid_type = BLE_GATT_16_BIT_UUID_FORMAT,
              .serv_cb = R_BLE_WSC_ServDiscCb,

              /* Register include service entry*/
              .inc_servs = gs_disc_wsc_inc_entries,
              .num_of_inc_servs = 1

      },
};
```

Figure 6.6 code modified to discover included service

Store Attribute handle of included service

Discovered attribute handle of included service will be passed to parent service API program. But parent service API program don't store attribute handle of included service. Therefore, in case Service YYY is discovered as included service that Service XXX has, you can't get range of its attribute handle by calling service YYY's API R_BLE_YYY_GetServAttrhdl().

If service YYY's range of attribute handle is needed, modify service XXX's API program (r_ble_xxx.c) so that the notification that service YYY is discovered as a include service is delivered to service YYY's discovery callback function.

The following show example in case Service XXX have 16bit UUID and have service YYY as included service. Take care the data type is different in 128bit UUID and in 16bit UUID.

```
#include <string.h>
#include "r_ble_XXX.h"
#include "profile_cmn/r_ble_servc_if.h"

/* ADD : including discovery library and include service yyy */
#include "discovery/r_ble_disc.h"
#include "r_ble_yyy.h"

void R_BLE_XXX_ServDiscCb(uint16_t conn_hdl, uint8_t serv_idx, uint16_t type, void *p_param)
{
    /* ADD : */
    uint16_t YYY_UUID = 0x0000;
    if (type == BLE_DISC_INC_SERV_FOUND)
    {
        st_disc_inc_serv_param_t * evt_param =
           (st_disc_inc_serv_param_t *)p_param;

        if (evt_param->uuid_type == BLE_GATT_16_BIT_UUID_FORMAT)
        {
              if (YYY_UUID == evt_param->value.inc_serv_16.service.uuid_16)
              {
                  st_disc_serv_param_t serv_param = {
                      .uuid_type            = BLE_GATT_16_BIT_UUID_FORMAT,
                      .value.serv_16.range   = evt_param->value.inc_serv_16.service.range,
                      .value.serv_16.uuid_16 = evt_param->value.inc_serv_16.service.uuid_16,
                   };
                   R_BLE_YYY_ServDiscCb(
                               conn_hdl, /* Connection handle */
                               0, /* idx */
                               BLE_DISC_PRIM_SERV_FOUND, /* Notify as a primary service */
                               &serv_param); /* Service handle information */
              }
          }
      }
/* Generated code */
}
```

Figure6.7 Discovery of included service

6.5 Guide for Connection Update

In Bluetooth LE communication, you can change the communication frequency during communication by connection update.

Connection update can be performed by using function [R_BLE_GAP_UpdConn]. To change frequency of communication, change the following parameters.

1. Connection Interval
   - Sets frequency of communication. user can set maximum value and minimum value. Value is calculated by (set value) × 1.25ms.
   - variable: conn_intv_min, conn_intv_max
2. Peripheral latency
   - Ignores communications by the number of value set. If set to 5, communication until the 6th reception will be ignored after first reception.
   - variable: conn_latency
3. Supervision Timeout
   - Connection will be disconnected after the time set here. If user want to reduce the frequency of communication, this value needs to be changed accordingly. Value is calculated by (set value) × 10ms.
   - variable: sup_to

Figure 6.8 shows the example of implementing connection update function in function disc_comp_cb.

```
static void disc_comp_cb(uint16_t conn_hdl)
{
    st_ble_gap_conn_param_t conn_param = {
            .conn_intv_min    = 0x0100,
            .conn_intv_max    = 0x0100,
            .conn_latency     = 0x0010,
            .sup_to           = 0x0200,
            .min_ce_length    = 0xFFFF,
            .max_ce_length    = 0xFFFF,
        };
        R_BLE_GAP_UpdConn(conn_hdl, BLE_GAP_CONN_UPD_MODE_REQ, 0x00, conn_param);
/* End user code. Do not edit comment generated here */
    return;
}
```

Figure 6.8 Example of using Connection Update function

The connection parameter is changed by the connection update when the requested device accepts it. For information on changing connection parameters by connecting update, refer to "Update Connection Parameters" in the application developer guide.

6.6 Settings for connecting two MCUs for data communication

An example of profile design when connecting two projects for data communication using the QE for BLE application framework is shown below.

To connect and communicate between two projects, the following settings are required.

- The advertisement data must include scan filter data.
- Supporting the same profile

This chapter shows a design example when connecting two projects using the following projects.

- Central (GATT client): prof_dev_central
- Peripheral (GATT server): prof_dev_peripheral

Set the GAP roles on the Profile tab of QE for BLE to Central and Peripheral, respectively.

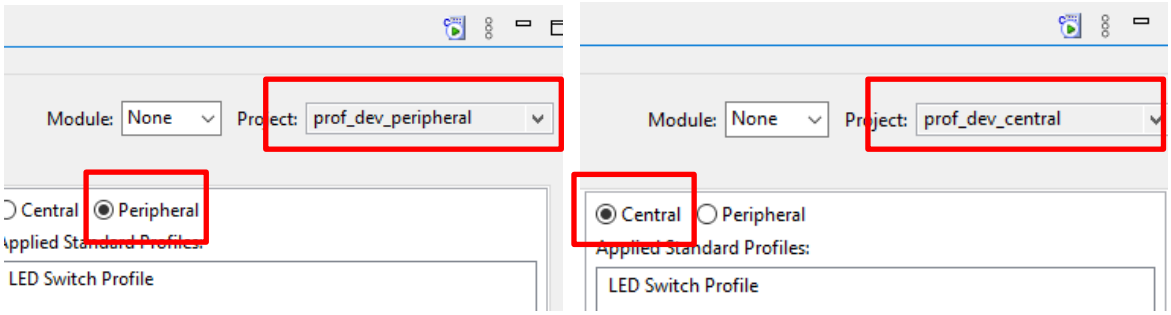


Figure 6.9 Example for setting GAP role

Next, set the profile for data communication. Add the same service using the service import / export function. This time, the central role will be the client and the peripheral role will be the server. Make sure the UUIDs of the services you use match.

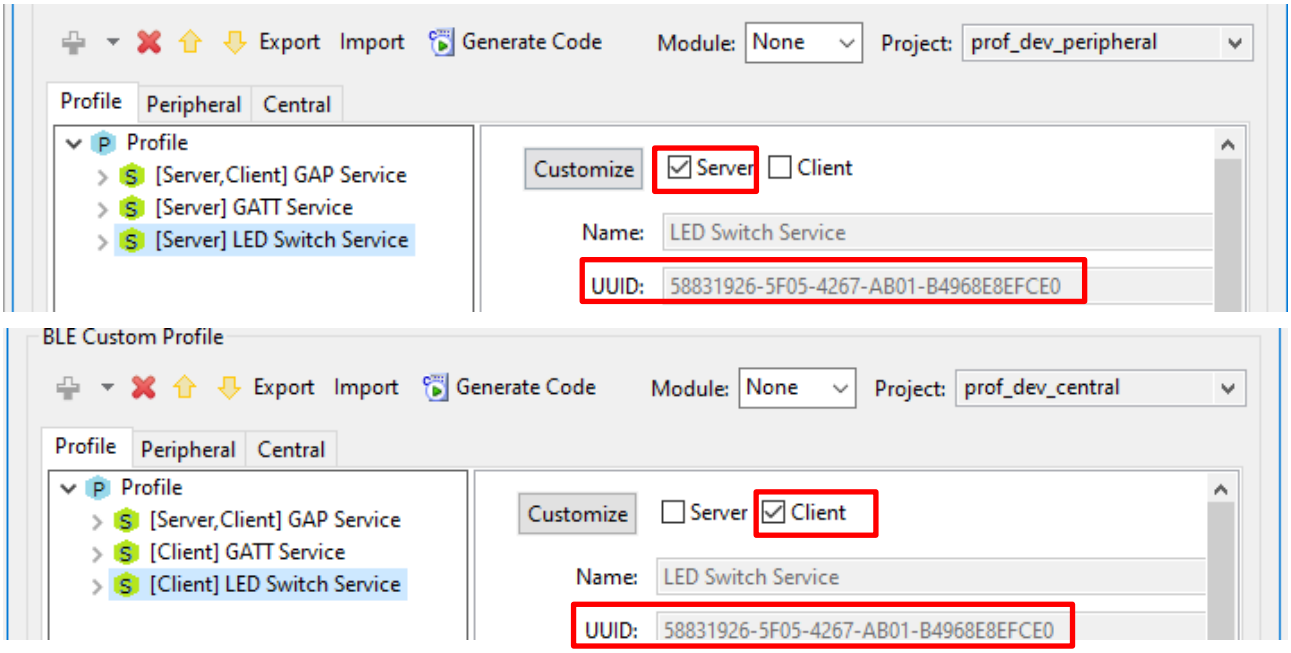


Figure 6.10 Example for setting profile

Finally, set the Advertise Data and Scan Filter Data respectively. Central makes a connection when it receives an advertisement with the data specified in Scan Filter Data. By matching this value, you can connect two projects. The setting example when "Local Name" is used for Advertise Data is shown.

For peripheral roles, set Advertise Data from the Peripheral tab.

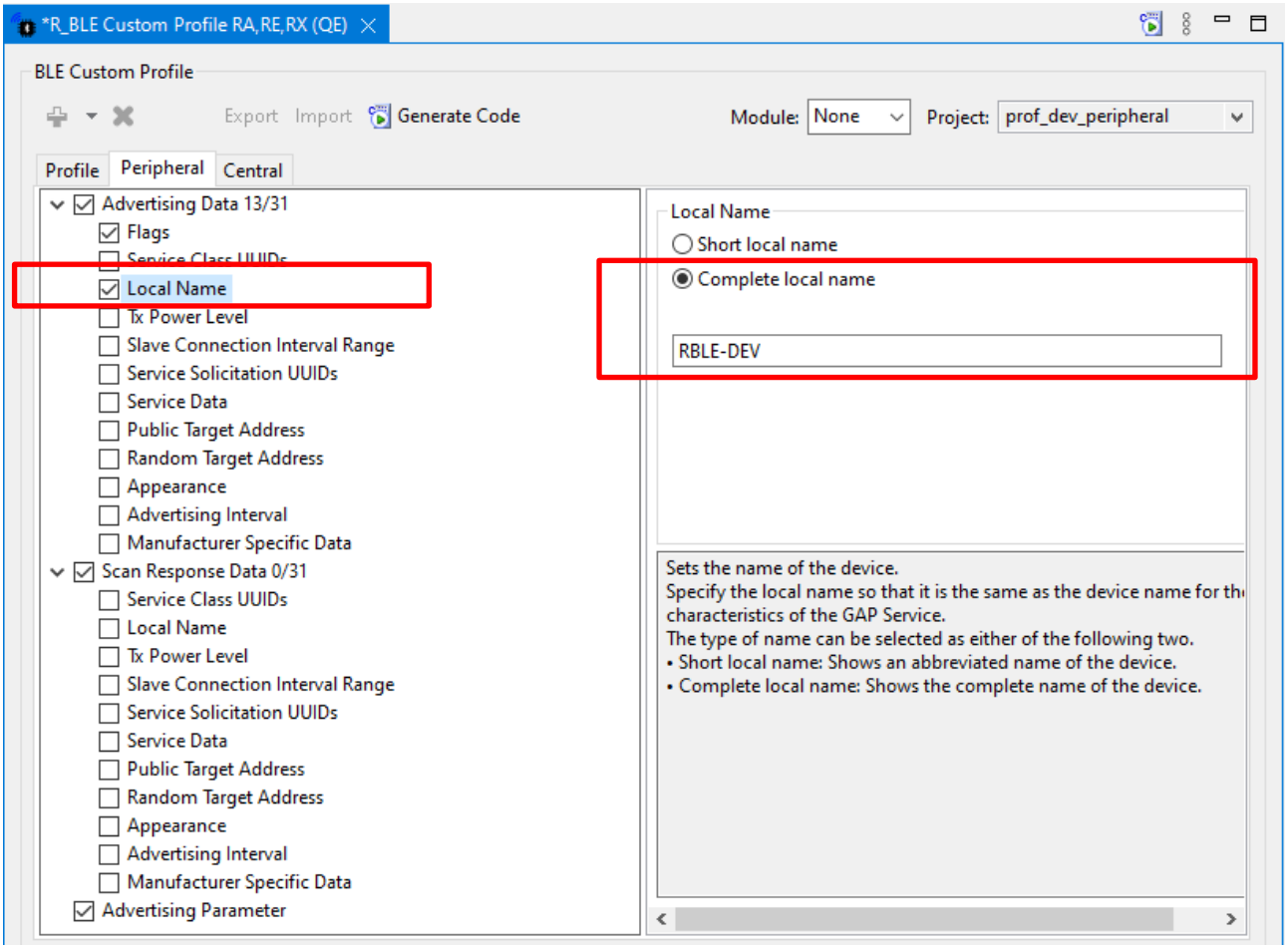


Figure 6.11 Example for setting advertising data

For central roles, set Scan Filter Data from the Central tab. Check the data same to the one set in Advertise Data of the peripheral and set the value.

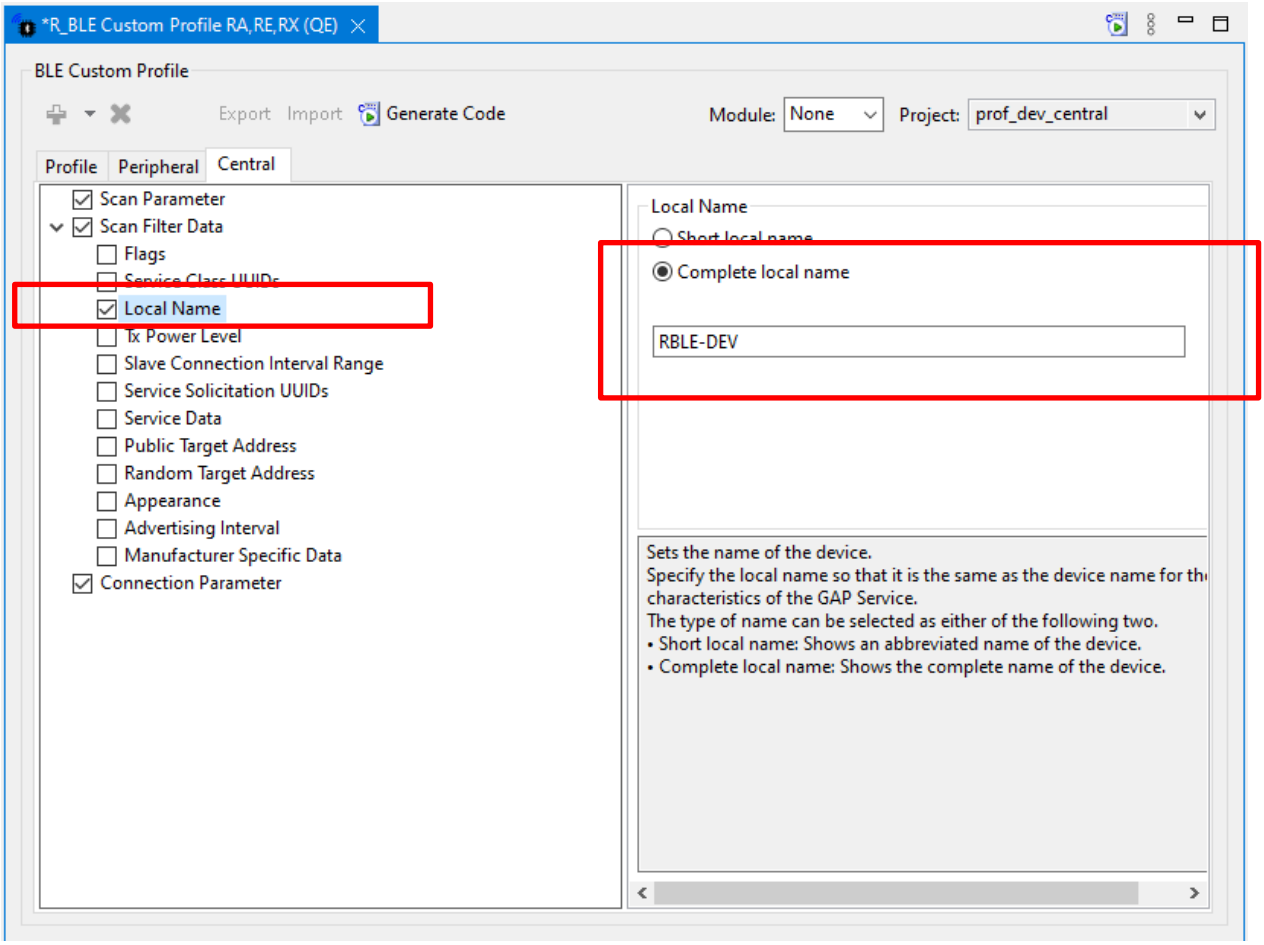


Figure 6.12 Example for setting Scan Filter Data

This completes the settings required to connect the two projects.

If you cannot connect after the above settings, review the advertisement interval and scan interval.

In the initial setting of QE for BLE, the Advertising operation and scan operation use the "Slow" setting. This setting consumes less power but makes device detection more difficult. If your application wants a quick connection, use the "Fast" setting.

The following is an example of setting "Fast" for the peripheral role. Set the "Advertising Parameter". If you want to use the "Fast" setting, check "Enable Fast Advertising". With this setting, advertisement packets are sent every 30 msec for 30 seconds from startup.

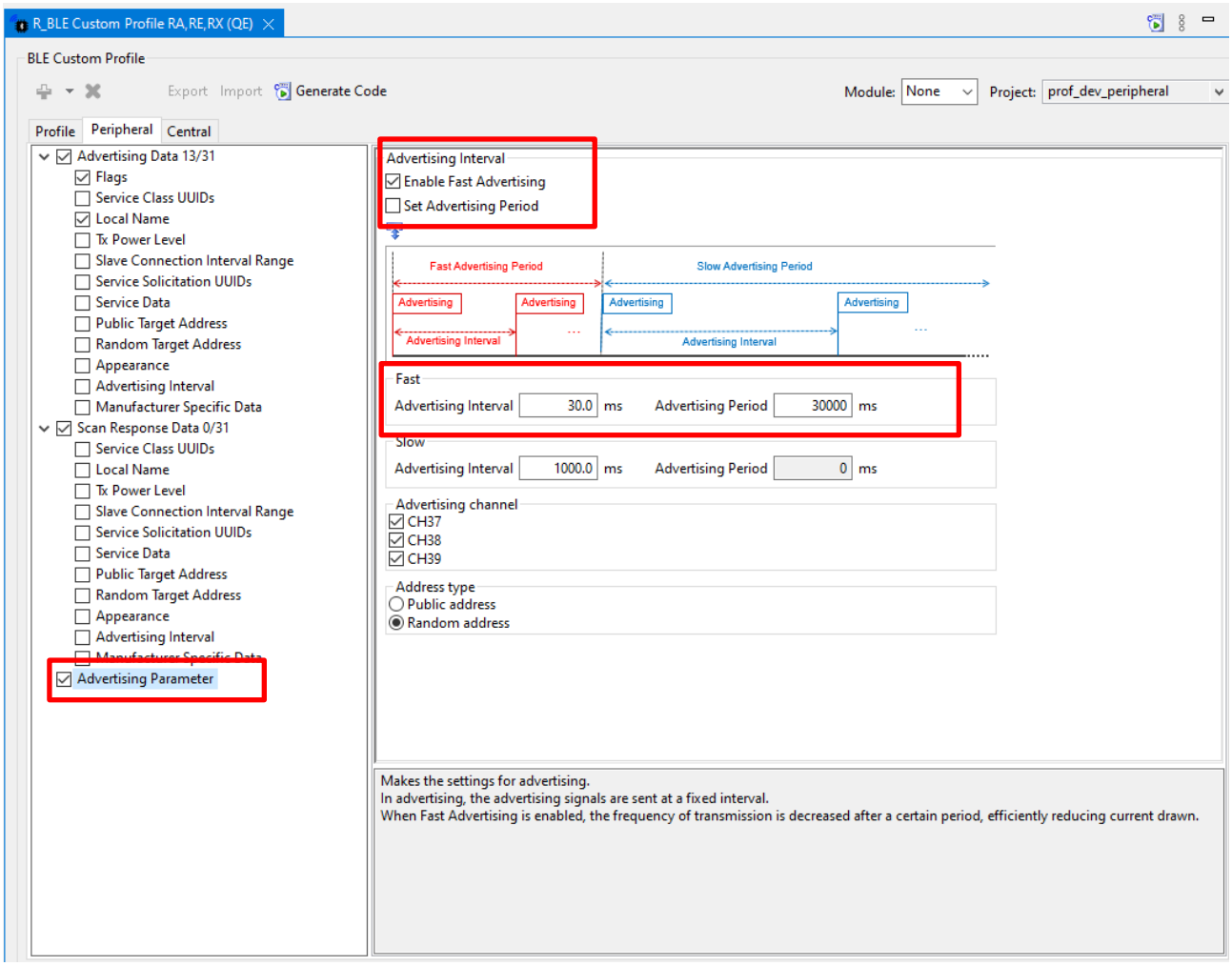


Figure 6.13 Setting example when using the "Fast" setting for Advertise operation

Here is an example of setting "Fast" for the central role. Set the "Scan Parameter". If you want to use the "Fast" setting, check "Enable Fast Scan". With this setting, the Scan Window for 30msec is opened every 60msec for 30 seconds from the start, and the Scan operation is executed.

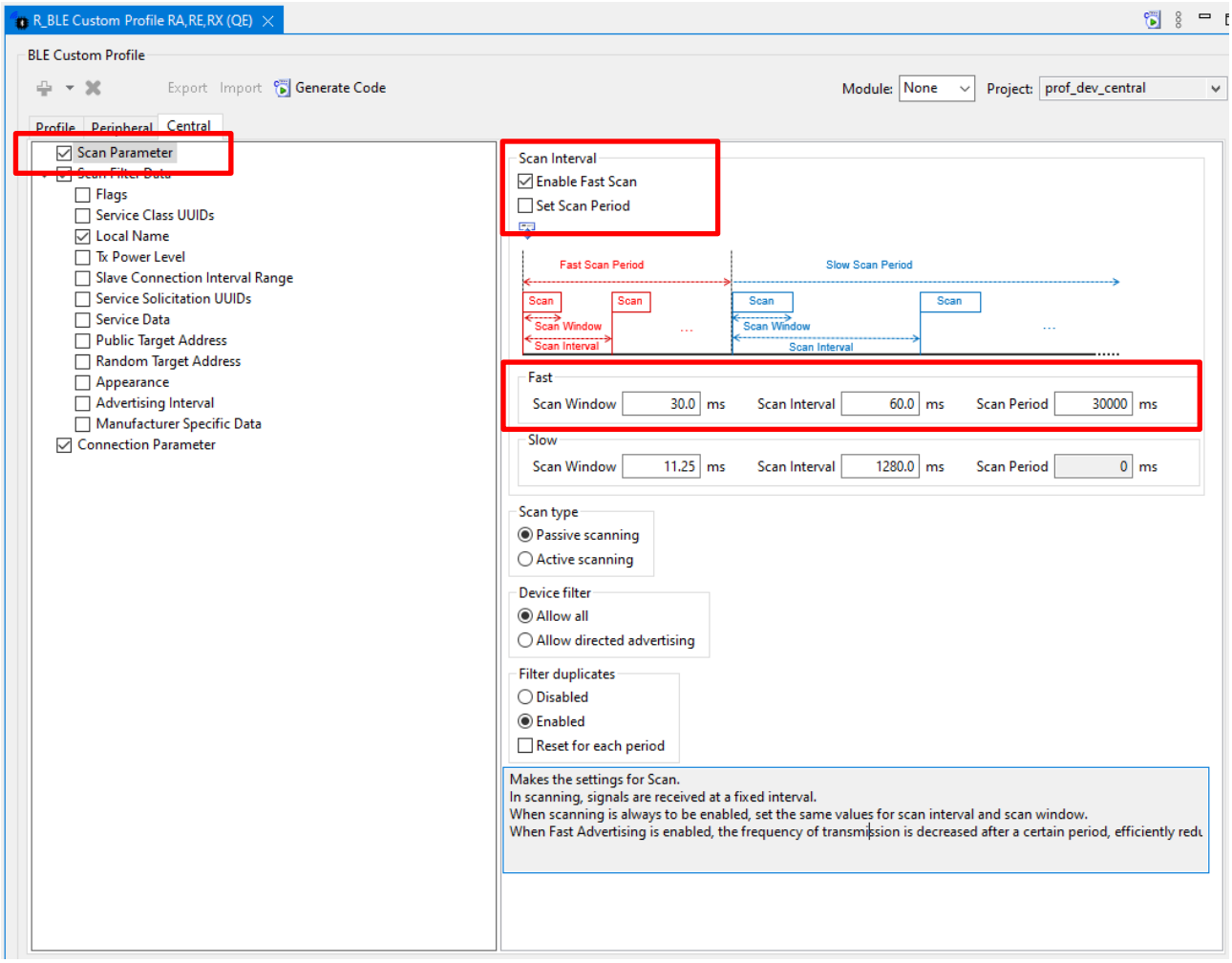


Figure 6.14 Setting example when using the "Fast" setting for scanning operation

6.7 When using old version qualifications (QDID:134484)

A service API program with "QDID: 1992482" qualifications is generated from QE for BLE Utility 1.60. If you want to continue developing using the previous qualification s(QDID:134484), follow the two steps below. *The qualifications (QDID:134484) cannot be newly registered as a Bluetooth qualified product after February 1, 2023. If the product under development is a derivative of an already qualified product, additional registration of the product is possible until January 31, 2024. After February 1, 2024, only existing certified products can be sold.

- QE for BLE generation code change setting
- Get profile common library (RX23W only)

6.7.1 QE for BLE generation code change setting

Change the QE for BLE settings when developing using the previous authentication information.

Open "Preferences" from "Window" in the e^2 studio menu bar.

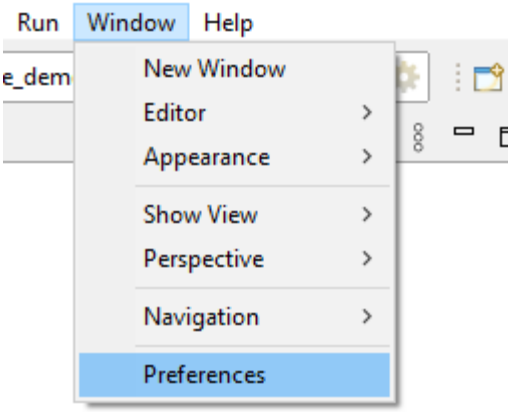


Figure 6.15 How to open QE for BLE options

Select "Renesas" → "Renesas QE" → "QE for BLE" from the list on the left, and check "Use the old QE for BLE [RA, RE, RX] Utility."

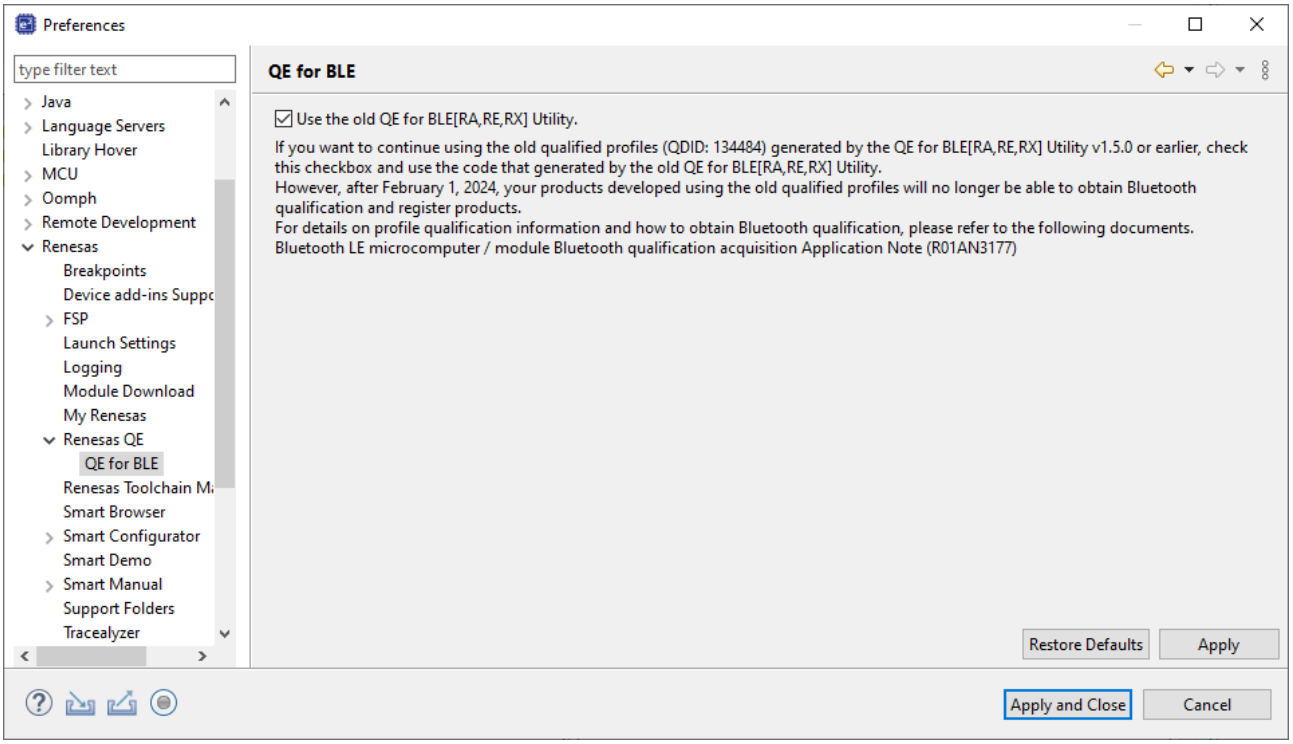


Figure 6.16 Settings screen that uses the old version of the service API program

If you generate code from QE for BLE after applying the settings, the old version of the service API program will be generated.

6.7.2 Get profile common library

This procedure is performed when using BLE FIT Module 2.50 or later in the RX23W environment. If you use BLE FIT 2.50 or later and use an older version, the profile common library is not generated from QE for BLE, so you need to add the profile common library to the project.

Please use one of the following methods.

- Add profile common library generated from QE for BLE.
- Restore the profile common library included in BLE FIT 2.40 from the trash folder.

Added profile common library generated from QE for BLE

Uncheck "Use the old QE for BLE[RA,RE,RX]" in section 6.7.1 and generate the code, and copy the generated folder below to the project path.

- qe_gen/discovery
- qe_gen/profile_cmn

After that, execute Section 6.7.1 again and set so that the old version of the service API program is generated.

Restore the profile common library included in BLE FIT 2.40 from the trash folder.

Copy the following folder of BLE FIT 2.40 in trash to the project path.

- src/smc_gen/r_ble_rx23w/src/discovery
- src/smc_gen/r_ble_rx23w/src/profile_cmn

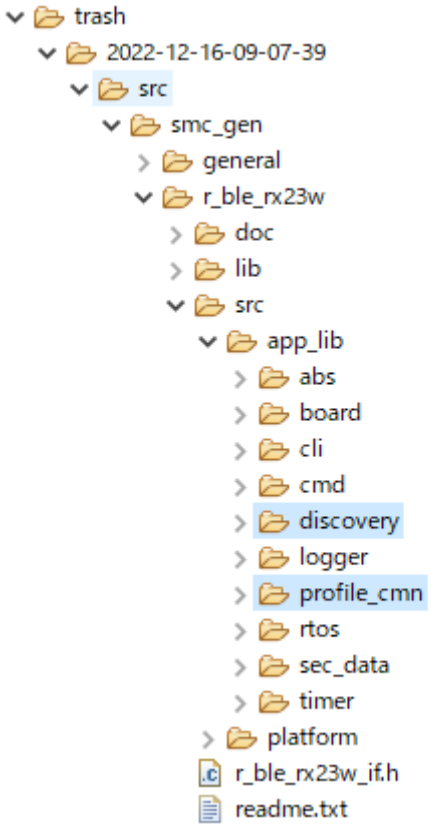


Figure 6.17 Restore from trash folder

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Jun.30.22	—	First edition issued.
1.10	Dec.27.22	5	Specified that profile common library is generated from QE for BLE in RX23W environment.
		6	Updated profiles/services version information supported by QE for BLE.
		20	Added security requirements for Custom and SIG standard profiles/services.
		42	Added explanation about automatic generation function of encode/decode function.
		49	Added implementation method when security requirements are set.
		52, 61	Added description of Write Long operation and Read Long operation.
		75	Added that there is no profile common library when combining BLE FIT 2.50 and QE for BLE Utility 1.50.
		94	Added 6.7 When using old version qualifications (QDID:134484).

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

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

1. Precaution against Electrostatic Discharge (ESD)

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

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

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

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

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

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

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

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

Notice

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

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

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

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

(Rev.5.0-1 October 2020)

Corporate Headquarters

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

Trademarks

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

Contact information

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

© 2022 Renesas Electronics Corporation. All rights reserved.