Bluetooth® low energy Protocol Stack

Application Development Guide

R01AN2768EJ0130
Rev.1.30
Dec 7, 2018

Introduction

This manual describes how to develop an application using the Bluetooth low energy software (hereafter called BLE software), and overview of RWKE (Renesas Wireless Kernel Extension) and BLE Protocol Stack.

If you will make an application in the Modem configuration, it is necessary to understand rBLE APIs to use BLE Protocol functions.

If you will make an application in the Embedded configuration, it is necessary to understand not only rBLE API but RWKE APIs to use RWKE functions.

Applicability

The descriptions in this guide apply to BLE software (RTM5F11A00NBLE0F10RZ) Version 1.20 and later.

Target Device

RL78/G1D

Contents

1. BLE software 4
2. RWKE 5
2.1 RWKE 5
2.2 Executing RWKE 7
2.3 RWKE API 8
2.3.1 Event Function 8
2.3.2 Message Function 10
2.3.3 Task State Function 13
2.3.4 Timer Function 15
2.3.5 Memory Function 17
2.3.6 Available RWKE API function in Interrupt Processing 18
2.3.7 Resources Related to RWKE 18
2.4 Use Case 20
2.4.1 Sequence to use data received by BLE communication 20
2.4.2 Sequence to execute processing after waiting specified time 20
2.4.3 Sequence driven by interrupt of RL78/G1D peripheral function 20
2.5 Implementing Application 21
2.6 Notes 24
3. BLE Protocol Stack 26
3.1 BLE Protocol Stack 26
3.2 rBLE API 27

3.3 How to Call rBLE Command 28
3.4 How to Receive rBLE Event 29
3.4.1 Preparation of Callback Function 29
3.4.2 Registration of Callback Function 30

4. Profile 31
4.1 Profile 31
4.2 GATT Database 33
4.3 How to make GATT Database 33
4.3.1 Adding Database Handle 34
4.3.2 Adding Database Index 34
4.3.3 Definition of UUID 35
4.3.4 Definition of Service 35
4.3.5 Definition of Characteristic 35
4.3.6 Adding to Database 36
4.4 How to make Custom Profile 37
4.4.1 Server Role 37
4.4.2 Behavior of Client Role 38
4.4.3 Data Access by Profile 39
4.4.4 Implementation of GATT Callback function 44

5. How application operates 45
5.1 Simple Sample Program 45
5.2 Start of BLE Software 47
5.3 Initializing BLE Protocol Stack 54
5.4 Starting Broadcast and Establishing Connection 55
5.5 Enabling Custom Profile 56
5.6 Data Communication of Custom Profile 57
5.6.1 Controlling LED Lighting Status 57
5.6.2 Periodical Switch Status Notification 59
5.7 Disabling Custom Profile and Restarting Broadcast 61

6. Development Tips 63
6.1 Sleep Function of BLE software 63
6.2 Bluetooth Device Address Supported by BLE software 68
6.3 Storing and Accessing Device Address 69
6.4 Broadcast Start Sequence 72
6.5 Usage of Bluetooth Device Name 74
6.5.1 Device Name of Local Device 74
6.5.2 Device Name of Remote Device 75
6.6 Update of Read Data 76

7. Appendix 77

7.1 How to Add Characteristic to Custom Profile 77
7.1.1 Definition of Sample Custom Service 78
7.1.2 Database Structure 79
7.1.3 Processing of Sample Custom Service Server Role 85
7.1.4 Adding Characteristic 89
7.1.5 Adding Server Profile API and Peripheral Application 93
7.1.6 Communication to Smartphone (Dipswitch State Characteristic) 99
7.1.7 Customize from Notification to Indication 101
7.1.8 Communication to Smartphone (Indication of Switch State Characteristic) 107

1. BLE software

BLE software includes BLE Protocol Stack and RWKE.

BLE Protocol Stack

BLE Protocol Stack is a software stack, which manages RF of RL78/G1D and provides an application with the functions to communicate by Bluetooth low energy.

BLE Protocol Stack provides **rBLE API**. By using the API, application in the Embedded and Modem configuration can access to Generic Access profile (GAP), Security Manager (SM), Vender Specific, and various profiles of BLE Protocol Stack

RWKE (Renesas Wireless Kernel Extension)

RWKE is a non-preemptive multitasking simple OS, to manage each processing of application and BLE Protocol Stack.

RWKE provides **RWKE API**. By using the API, application in the Embedded configuration can execute flexible processing sequence.

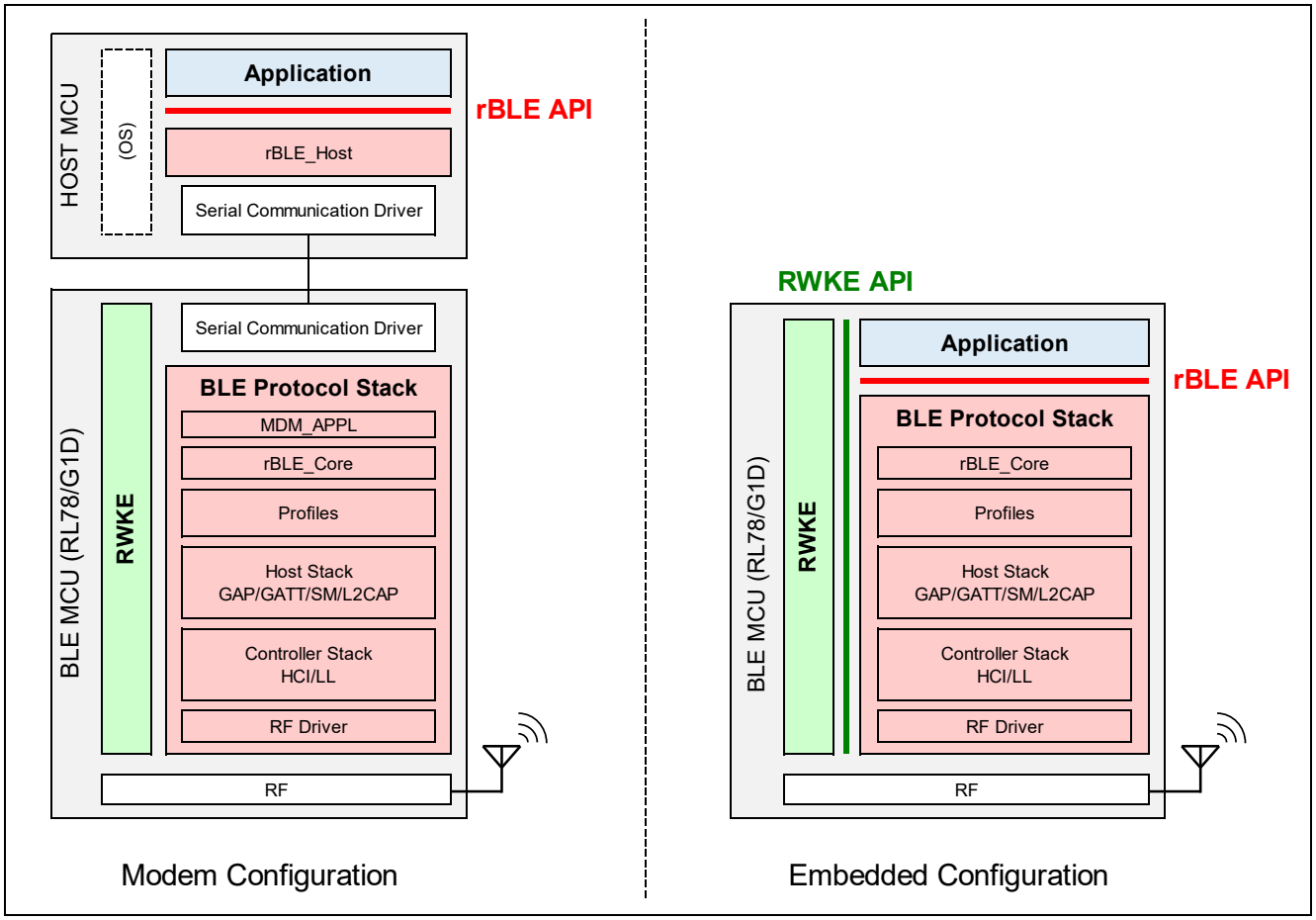


Figure 1-1 RWKE and BLE Protocol Stack

2. RWKE

2.1 RWKE

RWKE is a non-preemptive multitasking simple OS, to manage each processing of application and BLE Protocol Stack.

Application can use below RWKE functions. And RWKE provides RWKE API to use below functions.

- Event Function : When an event is set, RWKE executes event processing associated with the event in the order of event priorities.
- Message Function : When a message is sent, RWKE executes message processing associated with the message in the order of messages sent.
- Task State Function : RWKE changes message processing in accordance with task state.
- Timer Function : When timer is set, RWKE sends message after expiring the timer.
- Memory Function : RWKE allocates and releases memory area.

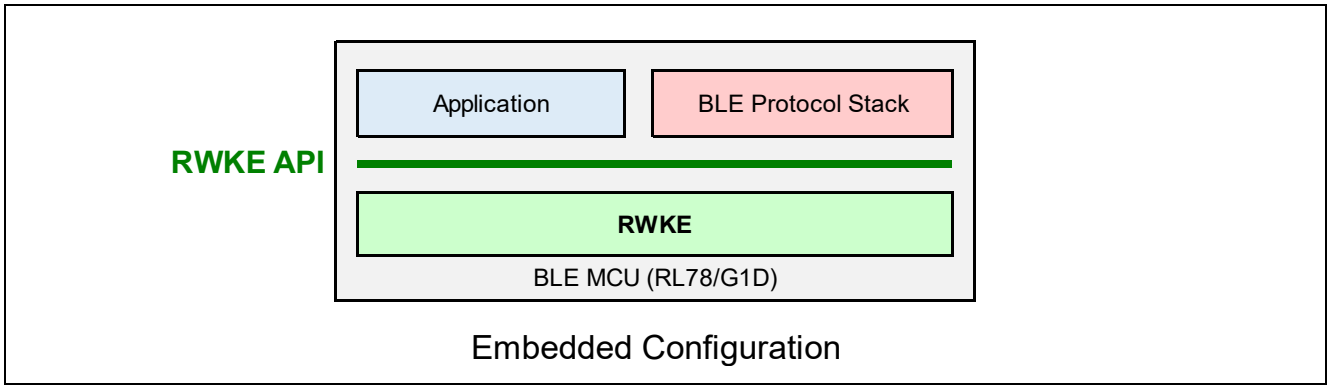


Figure 2-1 RWKE and RWKE API

The application can execute flexible processing sequence by using RWKE functions.

An example of application sequences driven by event function, message function, and timer function is as shown below.

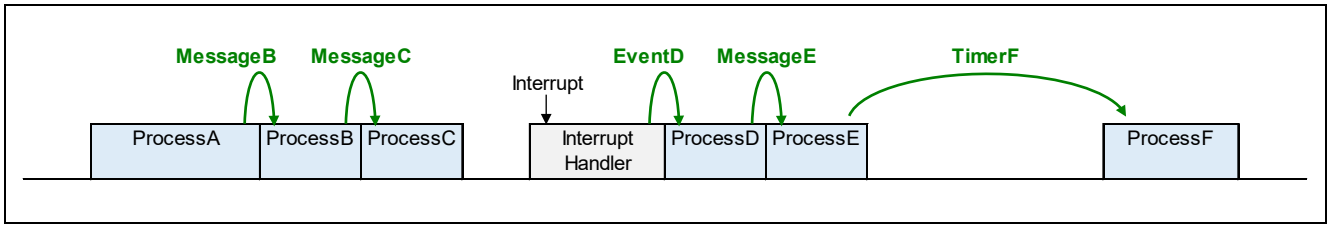


Figure 2-2 Example Sequence of Application

BLE Protocol Stack also executes processing to manage BLE communication by using RWKE functions.

An example of both application and BLE Protocol Stack sequence is as shown below.

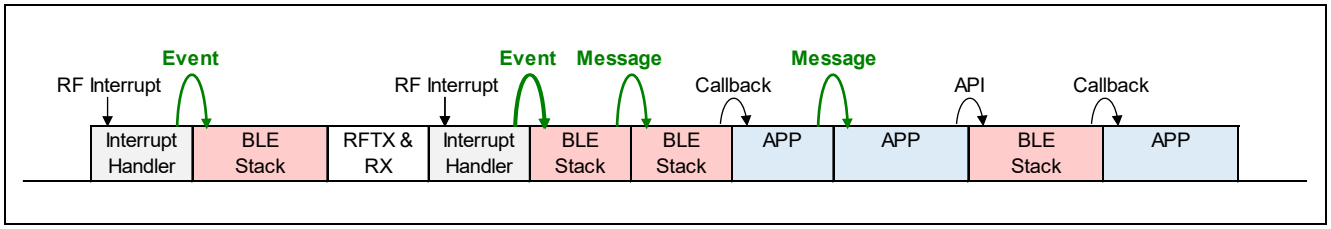


Figure 2-3 Example Sequence of Application and BLE Protocol Stack

Execution Priority of each event processing is shown below.

RWKE executes each event processing in accordance with execution priority. BLE Protocol Stack processing related to RF transmission and reception have higher priorities. And processing related to **message function** and **timer function** are managed as each one of event processing respectively.

Table 2-1 Execution Priority of RWKE Events

Event Priority	Event ID	Event Processing
0 (highest)	KE_EVT_EVENT_START	RF transmission and reception start
1	KE_EVT_RX	RF data reception
2	KE_EVT_EVENT_END	RF Transmission and Reception end
3	KE_EVT_HCI_TX_DONE	UART transmission end (only Modem configuration)
4	**KE_EVT_USR_0**	**Application event processing can be registered**
5	**KE_EVT_USR_1**	**Application event processing can be registered**
6	KE_EVT_KE_TIMER	**RWKE Timer check processing**
7	KE_EVT_KE_MESSAGE	**RWKE Message processing**
8	KE_EVT_CRYPT	RF encryption end
9	KE_EVT_HCI_RX_DONE	UART reception end (only Modem configuration)
10	**KE_EVT_USR_2**	**Application event processing can be registered**
11	**KE_EVT_USR_3**	**Application event processing can be registered**
12	reserved	reserved
:	:	:
31 (lowest)	reserved	reserved

RWKE executes each processing with non-preemptive. That means even if the higher priority event is set while RWKE executes an event processing of the lower priority event, RWKE doesn't suspend executing the event processing. After completion of executing the processing, RWKE start executing an event processing which has the higher priority event.

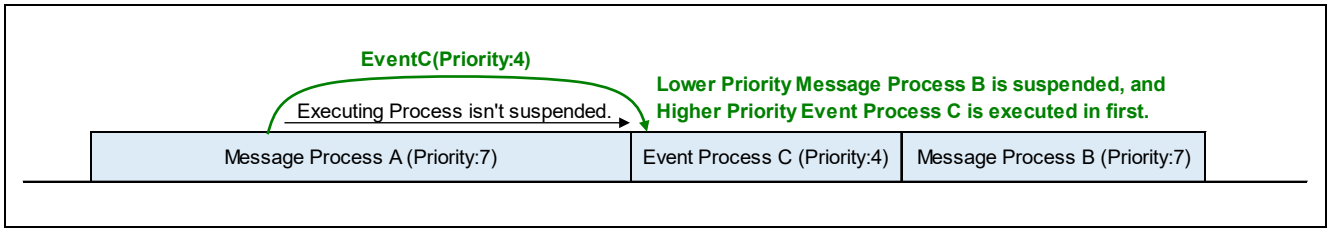


Figure 2-4 Execution Order of Message and Event

Note that RWKE is implemented to manage the schedule of software, so RWKE doesn't have below functions which typical other operating systems provide.

- Hardware Resource Management : It is necessary to implement each peripheral function driver.
- Interruption Management : It is necessary to implement each interrupt handler.
- Virtual Memory Space : It is necessary to consider how to use limited physical memory space.

2.2 Executing RWKE

RWKE is executed by **rwble_schedule function** in the main loop of BLE software.

This function executes RWKE functions consecutively. After that, if all of event or message are executed, this function finish processing. Again, if event or message to be executed is set, the main loop executes this function.

file: renesas/src/arch/rl78/arch_main.c

```
// And loop forever
for (;;)
{
    ...
    // schedule the BLE stack
    rwble_schedule();              executing RWKE

    // Checks for sleep have to be done with interrupt disabled
    GLOBAL_INT_DISABLE();
    // Check if the processor clock can be gated
    if ((uint16_t)rwble_sleep() != false)
    {
        // check CPU can sleep
        if ((uint16_t)sleep_check_enable() != false)
        {
                ...
                // Wait for interrupt
                WFI();
                ...
        }
    }
    // Checks for sleep have to be done with interrupt disabled
    GLOBAL_INT_RESTORE();

    sleep_load_data();
}
```

2.3 RWKE API

This section describes each RWKE function and RWKE API.

Regarding to the details of RWKE API, refer to below.

Bluetooth low energy Protocol Stack API Reference Manual: Basics (R01UW0088)
https://www.renesas.com/document/man/bluetooth-low-energy-protocol-stack-api-reference-manual-basics

- Chapter 9 "RWKE"

2.3.1 Event Function

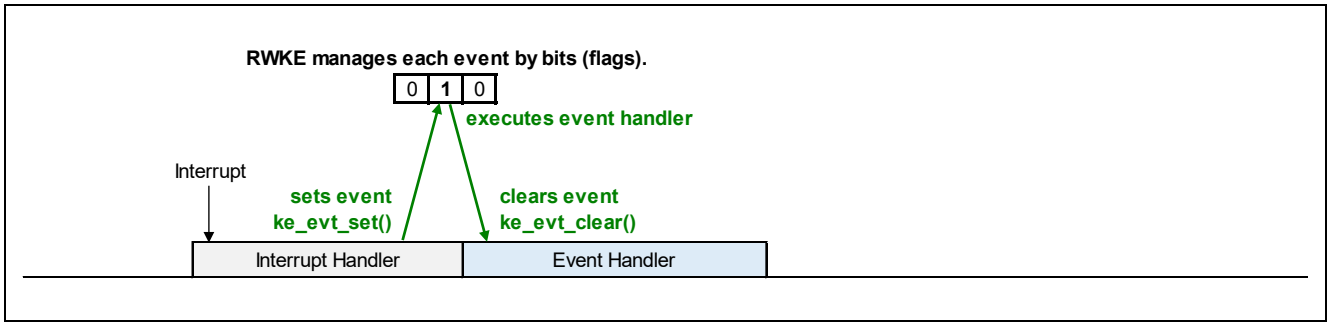
Overview of the Event Function is shown as below.

- The event function is a mechanism to execute event processing triggered by setting event.
- RWKE provides RWKE API to set and clear event.
- Each event is associated with each event processing (event handler) by event handler table.

Event Handler Table

Events	Event Handlers
Event A	Event Handler A
Event B	Event Handler B

- When event is set by RWKE API, RWKE executes event handler.
- The use-case is that an interrupt handler set event to execute succeeding application sequences.



The event function has some features compared to message function, which is described later.

- Events are defined by RWKE in advance, and events have different execution priority.
- Application use events which isn't used by other software.
- Application can set an event ID only. Other parameters can't be set.
- When an event is set, the RWKE doesn't allocate memory dynamically like as message function.
- Event are managed by bits. Even if same event is set consecutively at the same time, event handler is executed only once.
- If some different events are set, RWKE event handlers in the order of event execution priority.

RWKE API of Event function are shown as below.

Table 2-2 RWKE API of Event Function

RWKE API	Description
ke_evt_get	gets events status
ke_evt_set	sets an event
ke_evt_clear	clears an event

Example code of the event function on the simple sample program which is included in BLE software.

Regarding to the details of the simple sample program, refer to section 5.1.

1. Define the event bit (**KE_EVT_USR_0_BIT**) of the user0 event ID(KE_EVT_USR_0).

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.h

```
#define  KE_EVT_USR_0_BIT      CO_BIT(31 - KE_EVT_USR_0)
```

2. Declare new event handler (**app_evt_usr0**) for the user0 event.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.h

```
extern void codeptr app_evt_usr0(void);
```

3. Implement new event handler for the user0 event.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

```
void codeptr app_evt_usr_0(void)
{
    ke_evt_clear(KE_EVT_USR_0_BIT);

    ...
}
```

4. Register the event hander to the event handler table (**ke_evt_hdlr_ent**) of RWKE.

   file: renesas/src/arch/rl78/ke_conf_simple.c

```
/// Table of event handlers
_TSK_DESC const evt_ptr_t ke_evt_hdlr_ent[32] =
{
    ...
    DESGN(KE_EVT_USR_0  )  app_evt_usr0,
    DESGN(KE_EVT_USR_1  )  NULL,
    ...
    DESGN(KE_EVT_USR_2  )  NULL,
    DESGN(KE_EVT_USR_3  )  NULL,
};
```

5. Call **ke_evt_set function** to set event at any place of application.

```
{
    ...
    ke_evt_set(KE_EVT_USR_0_BIT);
}
```

If the user 0 event is set by procedure 5, RWKE executes the event handler implemented by procedure 3.

2.3.2 Message Function

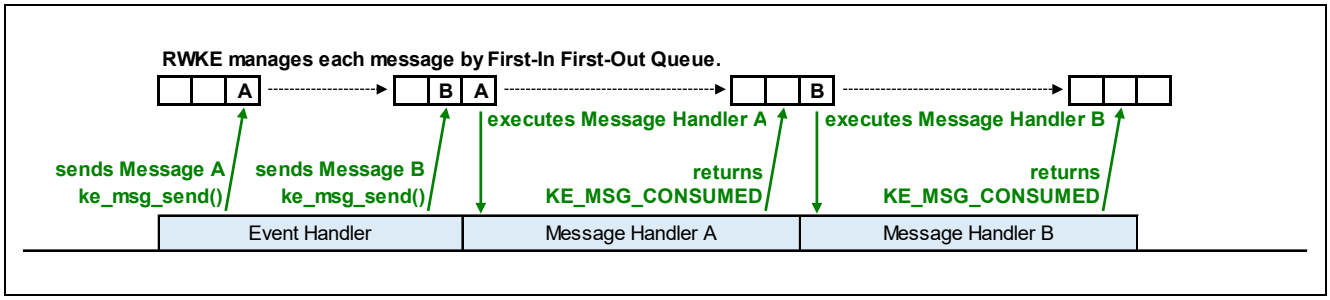
Overview of the Message Function is shown as below.

- The message function is a mechanism to execute message processing triggered by sending message.
- RWKE provides RWKE API to create and send event.
- Each message is associated with each message processing (message handler) by message handler table.

Message Handler Table

Messages	Message Handlers
Message A	Message Handler A
Message B	Message Handler B

- When message is sent by RWKE API, RWKE executes message handler.
- The use-case is that message handlers send message to execute continuous application sequence.



The message function has some features compared to the event function.

- Application can define various message, and event doesn't have execution priority.
- Message function can send not only a message ID but also parameters. And application can specify source and destination.
- When a message is sent, RWKE or application allocate memory dynamically to store the message.
- Message are managed by FIFO (First-In First-Out) queue of RWKE.
- If multiple messages are set, RWKE message handlers in the order of message sent.
- Even if same message is sent many time, same event handler is executed the number of times sent.
- When message handler returns KE_MSG_CONSUMED, RWKE releases memory to store message.

RWKE API of Message function are shown as below.

Table 2-3 RWKE API of Message Function

RWKE API	Description
ke_msg_alloc	allocates memory to store message
ke_msg_free	releases memory to store message
ke_msg_send	sends message which includes message header and parameter
ke_msg_send_basic	sends only message header which includes message ID, source task, and destination task
ke_msg_forward	forward message
ke_msg2param	gets parameters from message header
ke_param2msg	gets message header from parameters

Example code of the message function on the simple sample program which is included in BLE software.

Regarding to the details of the simple sample program, refer to section 5.1.

1. Define new message ID (**APP_MSG_MESSAGE_1**).

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.h

```
typedef enum {
    APP_MSG_BOOTUP = KE_FIRST_MSG(APP_TASK_ID) + 1,
    ...
    APP_MSG_MESSAGE_1,
} APP_MSG_ID;
```

2. Define message parameter structure (**app_param_1_t**), if application needs to send parameter.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.h

```
typedef struct {
    uint16_t member1;
    ...
} app_param_1_t;
```

3. Declare new message handler (**app_msg_message_1**) for the new message.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

```
static int_t app_msg_message_1(ke_msg_id_t const msgid, void const *param,
                        ke_task_id_t const dest_id, ke_task_id_t const src_id);
```

4. Implement the new message handler for the new message.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

```
static int_t app_msg_message_1(ke_msg_id_t const msgid, void const *param,
                         ke_task_id_t const dest_id, ke_task_id_t const src_id)
{
    ...
    return KE_MSG_CONSUMED;
}
```

   - If message which includes parameter is sent, application can get parameter by the argument **param** of the message handler.

```
static int_t app_msg_message_1(ke_msg_id_t const msgid, void const *param,
                         ke_task_id_t const dest_id, ke_task_id_t const src_id)
{
    app_param_1_t* app_param_1;

    app_param_1 = (app_param_1_t*)param;
    tmp = app_param_1->member1;
    ...

    return KE_MSG_CONSUMED;
}
```

5. Register the message ID and the message handler to the message handler table. For example, register them to the message handler table (app_connect_handler) for the task state APP_CONNECT_STATE.

   Note that message handler can be changed by using task state function, which is described later.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

```
const struct ke_msg_handler app_connect_handler[] = {
    ...
    { APP_MSG_MESSAGE_1, (ke_msg_func_t)app_msg_message_1 },
};
```

6. Call RWKE API to send message at any place of application.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

   - In the case of sending message which includes no parameter

     Call **ke_msg_send_basic function** to send message. This function allocates memory to store message. It isn't necessary to call ke_msg_alloc function.

```
{
    ...
    ke_msg_send_basic(APP_MSG_MESSAGE_1, APP_TASK_ID, APP_TASK_ID);
}
```

   - In the case of sending message which includes parameter

     Call **ke_msg_alloc function** to allocate memory to store message, and set parameter to the memory, and then call **ke_msg_send function** to send message.

```
{
    ...
    app_param_1_t* app_param_1;

    app_param_1 = (app_param_1_t*)ke_msg_alloc(APP_MSG_PARAM_1, APP_TASK_ID,
                                              APP_TASK_ID, sizeof(app_param_1_t));
    app_param_1->member1 = 0x0000;
    ...
    ke_msg_send(app_param_1);
}
```

If the message is sent by procedure 6, RWKE executes the message handler implemented by procedure 4.

2.3.3 Task State Function

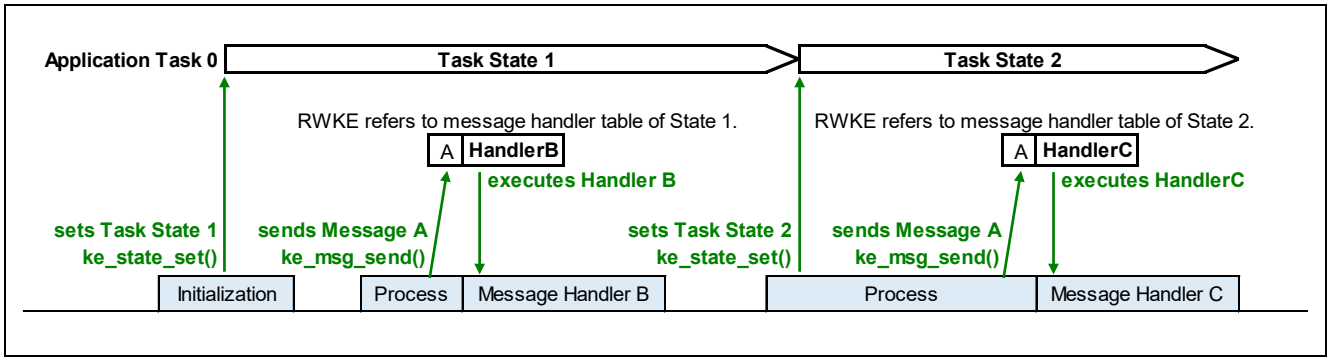
Overview of Task State Function is shown as below.

- The task State function is a mechanism to change the message sequences dynamically.
- RWKE provides RWKE API to get and set event.
- When task state is changed by RWKE API, RWKE changes message handler table.
- Each task state is associated with each message handler table by task state handler.

Task State Handler

Task States	Message Handler Tables
Task State 1	Message Handler Table 1
Task State 2	Message Handler Table 2

- When message is sent, RWKE refers to the message handler table of the current task state, and then execute the message handler.
- The use-case is that application change message sequence depends on each task state like as disconnected or connected. And if un-expected message was sent, application executes exception processing.



Message function has some features as shown below.

- Application can define various task states.
- Application can store multiple task sates simultaneously.

RWKE API of Task State function are shown as below.

Table 2-4 RWKE API of Task State Function

RWKE API	Description
ke_state_get	gets current task state
ke_state_set	sets new task state

Example code of the task state function on the simple sample program which is included in BLE software.

Regarding to the details of the simple sample program, refer to section 5.1.

1. Define new the task state (**APP_STATE1_STATE**).

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.h

```
typedef enum {
    APP_RESET_STATE = 0,
    APP_NONCONNECT_STATE,
    APP_CONNECT_STATE,
    APP_STATE1_STATE,
    APP_STATE_MAX
} APP_STATE;
```

2. Implement new message handler table (**app_state1_handler**) for the new task state.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

```
const struct ke_msg_handler app_state1_handler[] = {
    { APP_MSG_MESSAGE_1, (ke_msg_func_t)app_msg_message_1 },
    ...
};
```

3. Register the new message handler to task state handler (app_state_handler) of application.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

```
const struct ke_state_handler app_state_handler[APP_STATE_MAX] =
{
    KE_STATE_HANDLER(app_reset_handler),       //table for APP_RESET_STATE
    KE_STATE_HANDLER(app_nonconnect_handler),  //table for APP_NONCONNECT_STATE
    KE_STATE_HANDLER(app_connect_handler),     //table for APP_CONNECT_STATE
    KE_STATE_HANDLER(app_state1_handler),      //table for APP_STATE1_STATE
};
```

4. Call **ke_state_set function** to change task state at any place of application.

   ファイル：rBLE/src/sample_simple/rble_sample_app_peripheral.c

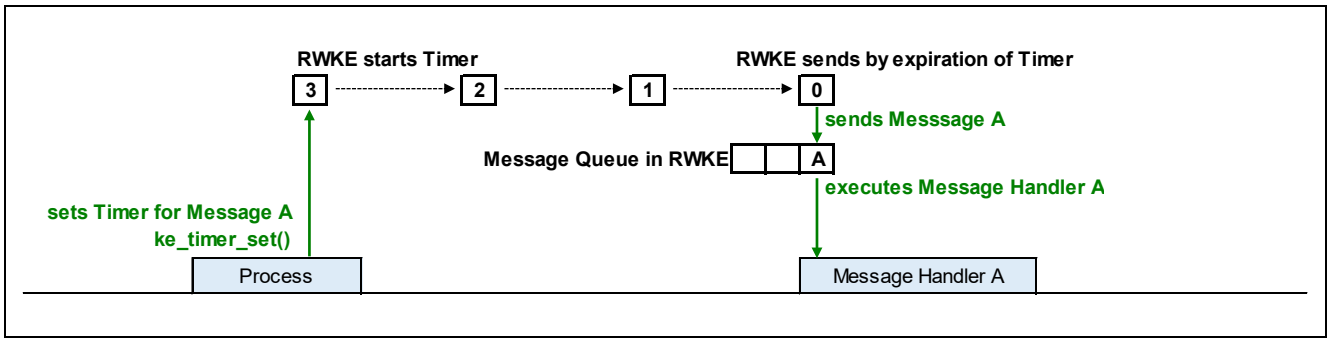
```
{
    ke_state_set(APP_TASK_ID, APP_STATE1_STATE);
    ...
};
```

If task state is changed by procedure 4, RWKE refers to the message handler table implemented by procedure 2 .

2.3.4 Timer Function

Overview of the Timer Function is shown as below.

- The timer function is a mechanism to send message after specified time expires.
- Temporal resolution of timer is 10msec.
- RWKE provides RWKE API to set and cancel timer.
- Timer is set by RWKE API. When timer expired, RWKE sends message.
- RWKE executes message handler corresponding to the message sent by the timer.
- The use-case is that application use the timer function to execute a processing periodically.



The timer function has some features compared to message function.

- Application can set a message ID and destination only. Parameter and source can't be set.
- When a timer is set, RWKE allocates memory dynamically to maintain the timer.
- If application sets same message ID and same destination again, RWKE doesn't allocate memory and update the time to send the message.
- The destination of message which is sent by timer, is empty (TASK_NONE).

RWKE API of Timer function are shown as below.

Table 2-5 RWKE API of Timer Function

RWKE API	Description
ke_time	gets current time
ke_timer_set	sets timer to send message
ke_timer_clear	cancels timer to send message

Example code of the timer function on the simple sample program which is included in BLE software.

Regarding to the details of the simple sample program, refer to section 5.1.

1. Define new message ID (**APP_MSG_TIMER_1**).

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.h

```
typedef enum {
    APP_MSG_BOOTUP = KE_FIRST_MSG(APP_TASK_ID) + 1,
    ...
    APP_MSG_TIMER_1,
} APP_MSG_ID;
```

2. Define new message handler (**app_msg_timer_1**) for the new message.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

```
static int_t app_msg_timer_1(ke_msg_id_t const msgid, void const *param,
                         ke_task_id_t const dest_id, ke_task_id_t const src_id);
```

3. Implement new message handler for the new message.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

```
static int_t app_msg_timer_1(ke_msg_id_t const msgid, void const *param,
                          ke_task_id_t const dest_id, ke_task_id_t const src_id)
{
    ...
    return KE_MSG_CONSUMED;
}
```

4. Register the message ID and the message handler to message handler table. For example, register them to the message handler table (app_connect_handler) for the task state APP_CONNECT_STATE.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

```
const struct ke_msg_handler app_connect_handler[] = {
    ...
    { APP_MSG_TIMER_1, (ke_msg_func_t)app_msg_timer_1 },
};
```

5. Call **ke_timer_set function** to set timer. For example, set 1sec (1msec * 100) to timer.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

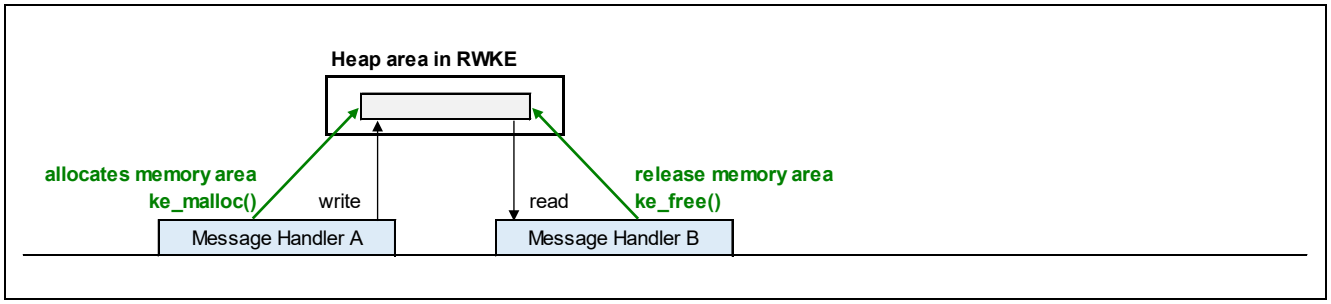
```
{
    ...
    ke_timer_set(APP_MSG_TIMER_1, APP_TASK_ID, 100);
}
```

If timer is set by procedure 5 and then the timer expired, RWKE executes the message handler implemented by procedure 3.

2.3.5 Memory Function

Overview of the Memory Function is shown as below.

- Memory function is a mechanism to use a specified size memory.
- RWKE provides RWKE API to allocate and release memory from heap memory.
- Application specifies size and allocates memory, and then write and read the memory.
- If application doesn't need the memory, application can release the memory.



RWKE API of Memory function are shown as below.

Table 2-6 RWKE API of Memory Function

RWKE API	Description
ke_malloc	allocates memory
ke_free	releases memory

Example code of the timer function on the simple sample program which is included in BLE software.

Regarding to the details of the simple sample program, refer to section 5.1.

1. Call **ke_malloc function** to allocate memory, and set data to the memory.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

```
app_param_1_t* app_param_1;

{
    app_param_1 = (app_param_1_t*)ke_malloc(sizeof(app_param_1_t));
    app_param_1->member1 = 0x0001;
    ...
}
```

2. Refer to data in the memory. If application doesn't need the memory, call **ke_free function** to release memory.

   file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

```
{
    tmp = app_param_1->member1;
    ...
    ke_free(app_param_1);
}
```

2.3.6 Available RWKE API function in Interrupt Processing

Available RWKE API functions in interrupt processing are shown below.

ke_evt_set : sets an event

ke_evt_clear : clears an event

ke_msg_alloc NOTE : allocates memory to store message

ke_msg_send NOTE : sends message which includes message header and parameter

ke_msg_send_basic NOTE : sends only message header which includes message ID, source task, and destination task

NOTE: These API need long processing time, so it isn't recommended to use these API in interrupt processing. In accordance with the use-case shown late, interrupt handler should set an event, and event handler sends message.

2.3.7 Resources Related to RWKE

Resources related to RWKE are shown below.

Regarding to how to implement resources of application, refer to section 2.5.

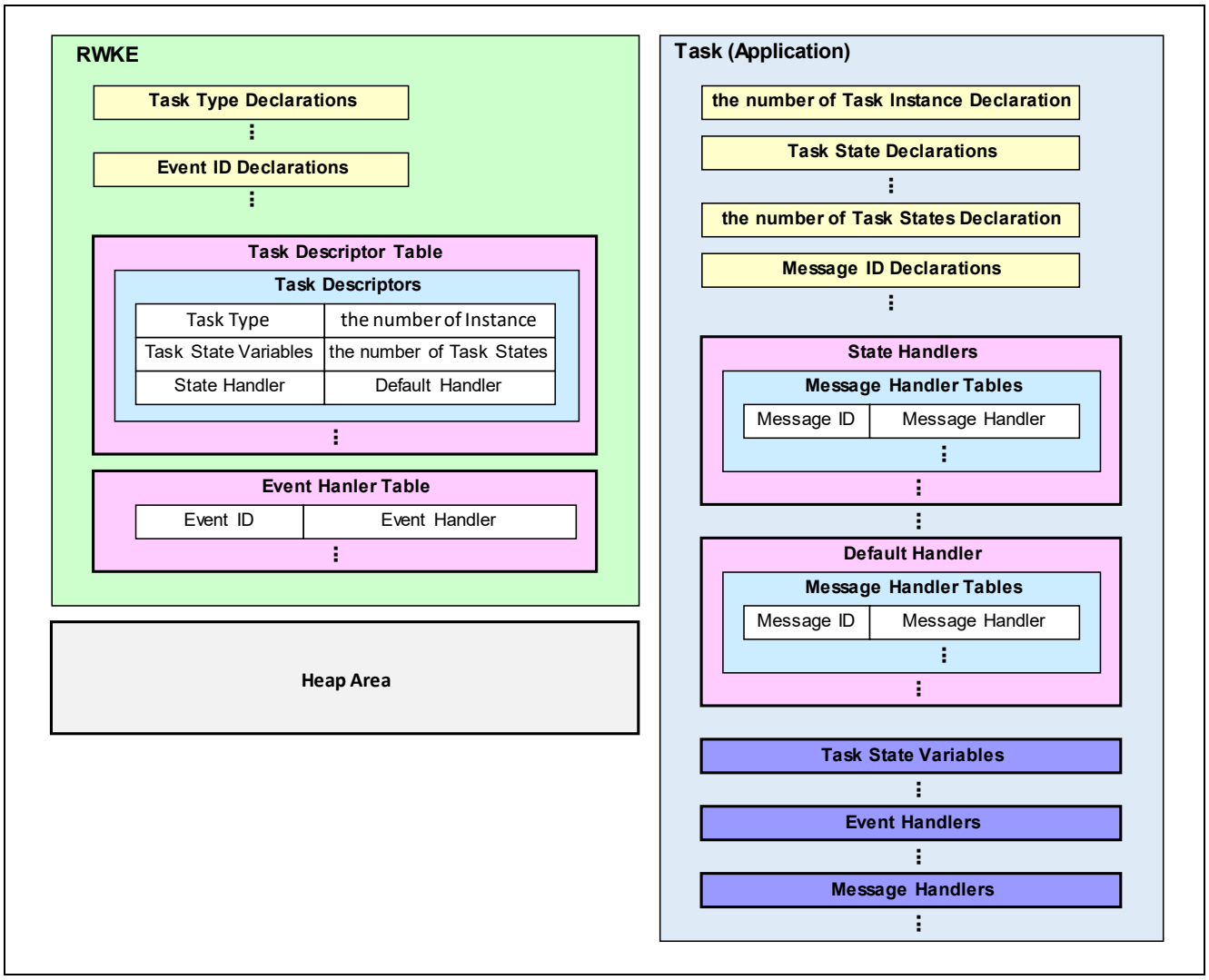


Figure 2-5 Resources Related to RWKE

Symbol names and implemented file names of resources related to RWKE are shown as below.

Note that this is in the case of the simple sample program which is included in BLE software. Regarding to the details of the simple sample program, refer to section 5.1.

Table 2-7 Resources Related to RWKE

Resource	Symbol Name ("*" means wild card)	Implemented File Name
Task Types	TASK_*	rwke_api.h
Event IDs	KE_EVT_*	
Task Descriptor Table	TASK_DESC_ent	ke_conf_simple.c
Event Handler Table	ke_evt_hdlr_ent	
Heap Area	ke_mem_heap_ent	arch_main.c
the number of Task Instance	APP_IDX_MAX	rble_sample_app_peripheral.h
Task States	APP_RESET_STATE APP_NONCONNECT_STATE APP_CONNECT_STATE	
the number of Task States	APP_STATE_MAX	
Message IDs	APP_MSG_BOOTUP APP_MSG_RESET_COMP APP_MSG_CONNECTED APP_MSG_DISCONNECTED APP_MSG_PROFILE_ENABLED APP_MSG_PROFILE_DISABLED APP_MSG_TIMER_EXPIRED	
State Handlers	app_state_handler	rble_sample_app_peripheral.c
Default Handler	app_default_handler	
Message Handler Tables	app_reset_handler app_nonconnect_handler app_connect_handler	
Task State Variable	app_state	
Message Handlers	app_reset app_advertise_start app_profile_enable app_profile_disable app_timer_expired	
Event Handlers	(not used)	(not used)

2.4 Use Case

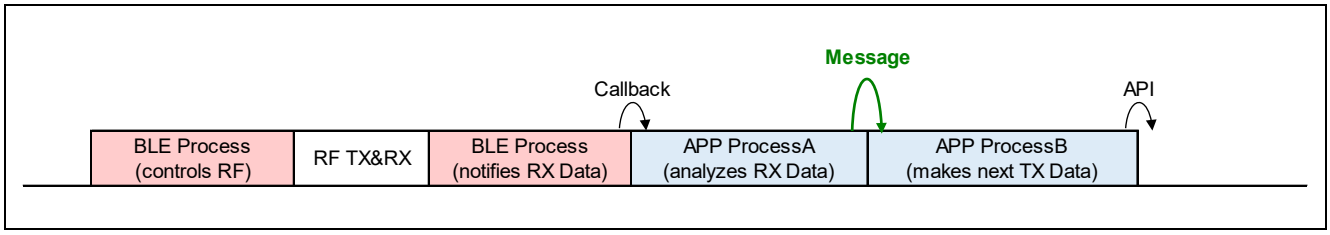
Use-cases of applications that use RWKE functions are shown below.

- Sequence to use data received by BLE communication
- Sequence to execute processing after waiting specified time
- Sequence driven by interrupt of RL78/G1D peripheral function

2.4.1 Sequence to use data received by BLE communication

Sequence to use data received by BLE communication is shown below.

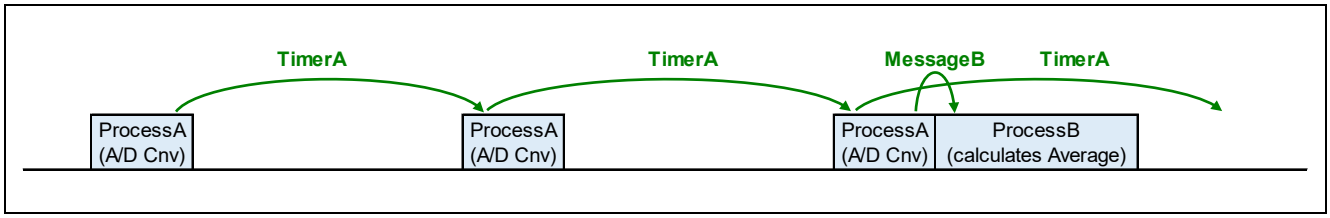
If BLE Protocol Stack receives data by executing RF transmission and reception, the stack notifies the received data to application by executing callback function. By analyzing the data and sending a variable message depends on the data, application can execute flexible processing sequence.



2.4.2 Sequence to execute processing after waiting specified time

Sequence to execute processing after waiting a specified time is shown below.

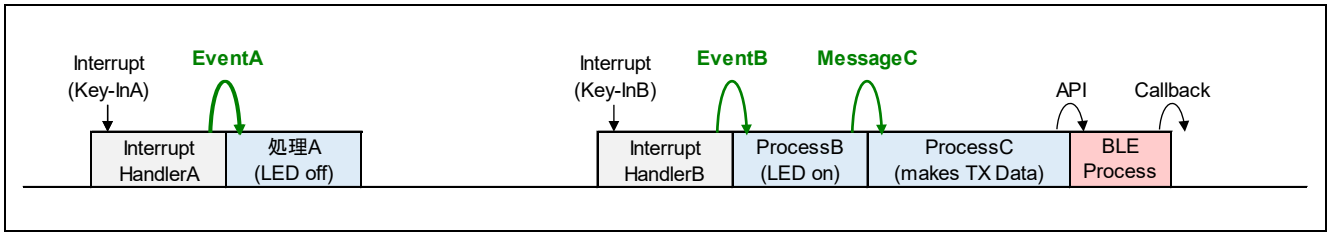
By using timer repeatedly, application can execute a processing periodically,



2.4.3 Sequence driven by interrupt of RL78/G1D peripheral function

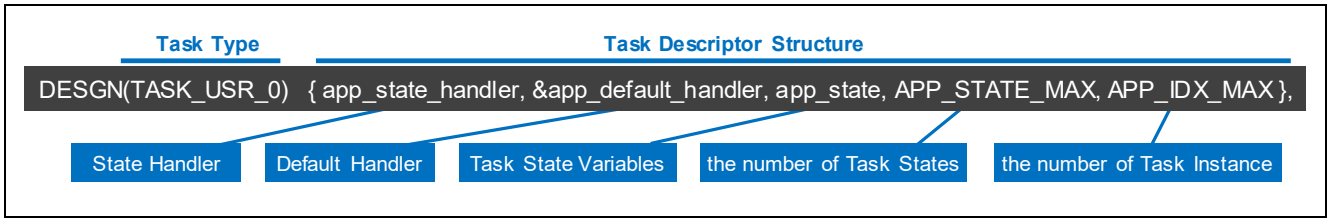
Sequence driven by interrupt of RL78/G1D peripheral function is shown below.

RL78/G1D peripheral function interrupt executes interrupt handler. Interrupt handler set an event and then event handler is executed. If it is necessary to execute additional processing, event handler sends message to execute message handler.



2.5 Implementing Application

To implement application, it is necessary to register application task to RWKE. Registering an application task needs below Task Type and Task Descriptor.



Task Type

Task type is defined as below. TASK_USR_0 and TASK_USR_1 are reserved for user application.

file: bleip/src/ke/ke_task.h

```
/// Tasks types.
enum
{
    ...
    // User Task (Embeded Portion)
    TASK_USR_0,
    TASK_USR_1,
    ...
};
```

Task Descriptor Structure

Task descriptor structure is defined by RWKE as below.

```
struct ke_task_desc
{
    const struct ke_state_handler *state_handler;     ..... State Handler
    const struct ke_state_handler *default_handler;   ..... Default Handler
    ke_state_t *state;                                ..... Task State Variable
    const uint16_t state_max;                         ..... the number of Task State
    const uint16_t idx_max;                           ..... the number of Task Instance
};
```

Task type and task descriptor are registered to task descriptor table. The task descriptor (**TASK_DESC_ent**) is implemented as shown below.

file: renesas/src/arch/rl78/ke_conf_simple.c in the case of simple sample program
file: renesas/src/arch/rl78/ke_conf.c not in the case of simple sample program

```
/// Table grouping the task descriptors
_TSK_DESC const struct ke_task_desc TASK_DESC_ent[] =
{
    ...

    DESGN(TASK_USR_0){ app_state_handler, &app_default_handler,
                                        app_state, APP_STATE_MAX, APP_IDX_MAX },
};
```

State Handler and the number of Task State

The State handler is a table that associates each task state with each message handler table. When a message is sent, RWKE refers to state handler, and refers to message handler table associated with current task state, and then execute message handler associated with the message.

And RWKE needs the number of task states defined by application to refer to state handler.

The state handler and the number of task states are registered to task descriptor.

```
DESGN(TASK_USR_0)  { app_state_handler, &app_default_handler, app_state, APP_STATE_MAX, APP_IDX_MAX },
```

Implementation in the case of the simple sample program is shown as below.

Application defines three task states, and the number of task state is **APP_STATE_MAX**.

file: rBLE/src/sample_simple/rble_sample_app_peripheral.c in the case of simple sample program

```
typedef enum {
    APP_RESET_STATE = 0,                ..... Reset State
    APP_NONCONNECT_STATE,               ..... Non-connected State
    APP_CONNECT_STATE,                  ..... Connected State
    APP_STATE_MAX                       ..... the number of Task State
} APP_STATE;
```

Each message handler tables associated with each task state are registered to state handler (**app_state_handler**). By referring to state handler, RWKE identifies the message handler table associated with current task state.

file: rBLE/src/sample_simple/rble_sample_app_peripheral.c in the case of simple sample program

```
const struct ke_state_handler app_state_handler[APP_STATE_MAX] =
{
    KE_STATE_HANDLER(app_reset_handler),
    KE_STATE_HANDLER(app_nonconnect_handler),
    KE_STATE_HANDLER(app_connect_handler),
};
```

In the above implementation, **app_nonconnect_handler** is registered as a message handler table of task state APP_NONCONNECT_STATE. If a message is sent when task state is APP_NONCONNECT_STATE, RWKE refers to **app_nonconect_handler** and executes message handler associated with the message.

file: rBLE/src/sample_simple/rble_sample_app_peripheral.c in the case of simple sample program

```
const struct ke_msg_handler app_nonconnect_handler[] = {
    {APP_MSG_RESET_COMP,          (ke_msg_func_t)app_advertise_start },
    {APP_MSG_DISCONNECTED,        (ke_msg_func_t)app_profile_disable },
    {APP_MSG_PROFILE_DISABLED,    (ke_msg_func_t)app_advertise_start },
};
```

Default Handler

If no message handler is associated with a message, RWKE executes message handler registered to default handler.

This default handler is registered to task descriptor.

```
DESGN(TASK_USR_0)  { app_state_handler, &app_default_handler, app_state, APP_STATE_MAX,  APP_IDX_MAX },
```

Implementation in the case of the simple sample program is shown as below.

If application doesn't need a default handler, default handler specifies **KE_STATE_HANDLER_NONE**.

file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

```
const struct ke_state_handler app_default_handler = KE_STATE_HANDLER_NONE;
```

Task State Variable and the number of Task Instance

Task state variable maintains task state of application. RWKE refer and change the task state variable.

By defining multiple task state variables, application can have multiple task states. The number of task state variables is called as the number of task instance.

These task state variables and the number of task instance are registered to task descriptor.

```
DESGN(TASK_USR_0)  { app_state_handler, &app_default_handler, app_state, APP_STATE_MAX,  APP_IDX_MAX },
```

Implementation in the case of the simple sample program is shown as below.

The number of task instance **APP_IDX_MAX** is defined as 1.

file: rBLE/src/sample_simple/rble_sample_app_peripheral.h

```
#define APP_IDX_MAX (1)
```

Task state variable **app_state** is defined as array.

file: rBLE/src/sample_simple/rble_sample_app_peripheral.h

```
ke_state_t app_state[APP_STATE_MAX];
```

Each task instance is identified by index in the range from 0 to (APP_IDX_MAX - 1). To use RWKE API, application make task ID by task type and macro **KE_BUILD_ID**.

Note that if index is 0, task ID is same as task type.

```
Task ID of application = KE_BUILD_ID(TASK_USR_0, idx);
```

2.6 Notes

This section describes notes to avoid un-intentional software behavior.

Application Implementation

- Each function of BLE Protocol Stack is managed by RWKE. To avoid un-intentional BLE communication behavior, the application also should be managed by RWKE.

  Basically, each processing of application should be managed by RWKE functions.

- RF transmission and reception processing is managed as high execution priority event by RWKE. To reduce an effect to processing scheduling of BLE Protocol Stack as much as possible, it is highly recommended that each processing time of application is shortened as much as possible, such as event handler, message handler, and callback function registered to BLE Protocol Stack. (Recommended processing time is within 30% of connection interval time.)

- If application processing consumes long time, it is necessary to divide processing into multiple message handler by using message function of RWKE.

- RWKE doesn't manage interrupts. If the application uses interrupts, to reduce an effect to processing scheduling of BLE Protocol Stack as much as possible, it is highly recommended that each interrupt processing time is shortened as much as possible. (Recommended processing time is within 1msec.)

  And if the application uses interrupts, in accordance with the use-case shown before, it is recommended that the interrupt handler set an event, and following application processing sequence is managed by RWKE.

RWKE Message Function

- Message function allocates a part of memory from heap area. If application continues to send many messages, heap area is exhausted eventually. To avoid exhausting heap area, application should be implemented so that many messages are not stored in the queue of RWKE.

Using rBLE API

- The rBLE API, which is described later, also use message function to execute BLE Protocol stack processing. To avoid exhausting heap area, basically, it is highly recommended that the application refrain from calling another rBLE API since it calls one rBLE API until BLE Protocol Stack notifies complete status.

- If application calls RBLE_GAP_Reset function which is one of rBLE API, RWKE also reset to avoid mismatch of BLE Protocol Stack processing scheduling.

  It is necessary to start application processing sequence after completion of resetting GAP by RBLE_GAP_Reset function.

Heap Area

- User can change the size of heap area managed by RWKE. If there is a possibility of exhausting heap area by application processing sequence, user needs to extend the heap area size.

  Note that when user extends heap area size, it is necessary to adjust the size to allocate stack memory area too.

  Implementation code of heap area **ke_mem_heap_ent** is shown as below.

  file: Renesas/src/arch/rl78/arch_main.c

```
uint8_t ke_mem_heap_ent[BLE_HEAP_SIZE];
```

Sleep Function

- BLE software has the function to change both MCU mode and RF mode of RL78/G1D into low power consumption mode. MCU mode and RF mode are changed into either one of below by this Sleep function.
  - Only RF mode is changed into standby (SLEEP mode or DEEP_SLEEP mode).
  - Not only RF mode is changed into standby (SLEEP mode or DEEP_SLEEP mode), but also MCU mode is changed into standby (STOP mode).

  Regarding to the details of Sleep function, refer to subsection 7.20.2 "Sleep" in Bluetooth low energy Protocol Stack User's Manual (R01UW0095).

MCU STOP Mode

- When there is no processing to be executed by RWKE such as event or message, Sleep function changes MCU mode into STOP mode.

  Implementation code to change MCU into STOP mode is shown as below.

  file: renesas/src/arch/rl78/arch_main.c

```
#if defined(_USE_CCRL_RL78)
 #define WFI()  __stop();
#else
 #define WFI()  __asm("stop");
#endif
```

  In STOP mode, almost peripheral functions stop such as serial array unit, timer array unit, A/D converter, and DMA controller. Regarding to the details, refer to chapter 19 "STANDBY FUNCTION" in RL78/G1D User's Manual: Hardware (R01UW0095).

  If the application needs to use a peripheral function, it is necessary to change not STOP mode but HALT mode temporary, or inhibit changing MCU mode into Sleep state temporary by using sleep_check_enable function.

3. BLE Protocol Stack

3.1 BLE Protocol Stack

BLE Protocol Stack is a software stack which manages RF unit of RL78/G1D and provides application with Bluetooth low energy communication functions.

Regarding to the details of functions provided by BLE Protocol Stack, refer to chapter 7 "Description of Features" in the user's manual (R01UW0095).

To execute BLE Protocol Stack functions, it is necessary to use rBLE API. Regarding to the details of rBLE API, refer to below.

Bluetooth low energy Protocol Stack API Reference Manual: Basics (R01UW0088)
https://www.renesas.com/document/man/bluetooth-low-energy-protocol-stack-api-reference-manual-basics

- Chapter 3 "Common Definitions"
- Chapter 4 "Initialization"
- Chapter 5 "Generic Access Profile"
- Chapter 6 "Security Manager"
- Chapter 7 "Generic Attribute Profile"
- Chapter 8 "Vendor Specific"

3.2 rBLE API

BLE Protocol Stack provides rBLE API. By using rBLE API, application can use functions of BLE Protocol Stack.

The rBLE API defines commands and events as follows.

Commands

Commands are requests to control each function of BLE Protocol Stack.
Commands are issued from application to BLE Protocol Stack.

Events

Events are notifications to indicate each executing results of BLE Protocol Stack.
Events are issued from BLE Protocol Stack to application.

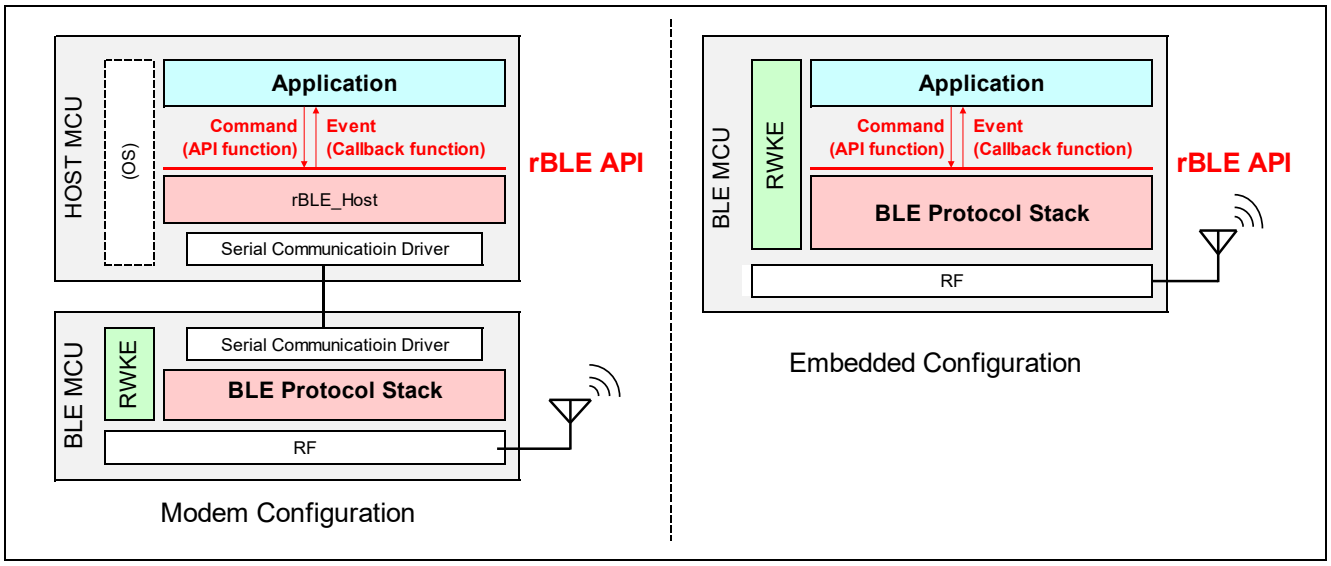


Figure 3-1 BLE Protocol Stack and rBLE API

rBLE API are implemented as non-blocking functions. This means that when application calls API function to issue a command, the API function finishes without waiting completion of executing command. After that BLE Protocol Stack executes the command and notifies an event by an argument of callback function.

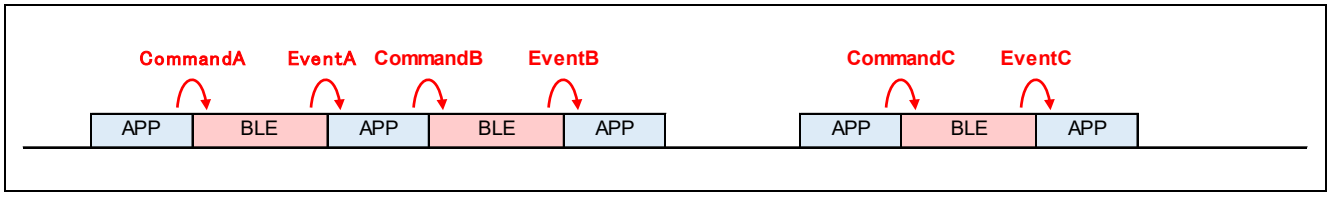


Figure 3-2 Example Sequence of Command and Event

Procedures to use rBLE API are shown as below. Regarding to the details, refer to following pages.

1. Register callback function to receive events.
2. Call API function to issue a command.
3. Receive an event from an argument of callback function.
4. Check the event and if application issues next command. → retry steps 2-4.

3.3 How to Call rBLE Command

To use a function of BLE Protocol Stack, application calls rBLE command API.

Include "rble_api.h" header file at first. As an example of using GATT function, call RBLE_GATT_Enable API to initialize GATT function. After completion of initializing GATT function, prepare parameters to set as argument of a command API, and then call command API.

An example of calling RBLE_GATT_Notify_Request API is shown as below.

file: rBLE/src/sample_simple/sam/sams.c

```
#include "rble_api.h"

static void sams_notify_request(void)
{
    RBLE_GATT_NOTIFY_REQ ntf;

    ntf.conhdl = sams_info.conhdl;              Connection Handle
    ntf.charhdl = sams_info.hdl;                Characteristic Handle

    (void)RBLE_GATT_Notify_Request(&ntf);
}
```

Figure 3-3 Example of calling rBLE Command

3.4 How to Receive rBLE Event

After calling rBLE command, application receives rBLE event by callback function. To receive rBLE events from BLE Protocol Stack, application needs to prepare and register the callback function.

3.4.1 Preparation of Callback Function

Callback function has rBLE event structure as an argument.

rBLE event structures are defined in "rble_api.h" header file and consists of **event type** and **union event parameter**.

As an example of rBLE event structure, the definition of EBLE_GAP_EVENT is shown as below.

file: rBLE/src/include/rble_api.h

```
typedef uint8_t     RBLE_GAP_EVENT_TYPE;

typedef struct RBLE_GAP_EVENT_t {
    RBLE_GAP_EVENT_TYPE     type;                          Event Type
    uint8_t                 reserved;
    union Event_Parameter_u {                              Event Parameters
        /* Generic Event */
        RBLE_STATUS          status;

        /* RBLE_EVT_GAP_Reset_Result */
        struct RBLE_GAP_Reset_Result_t {
            RBLE_STATUS status;
            uint8_t  rBLE_major_ver;       /* rBLE Major Version */
            uint8_t  rBLE_minor_ver;       /* rBLE Minor Version */
        }reset_result;
        ...
    } param;
} RBLE_GAP_EVENT;
```

Figure 3-4 Example of Event Structure

Event type in the event structure is set a value to identify each rBLE event.

file: rBLE/src/include/rble_api.h

```
typedef uint8_t     RBLE_GAP_EVENT_TYPE;

enum RBLE_GAP_EVENT_TYPE_enum {
RBLE_GAP_EVENT_RESET_RESULT = 1,         /* Reset result Complete */
RBLE_GAP_EVENT_SET_NAME_COMP,            /* Set name Complete */
RBLE_GAP_EVENT_OBSERVATION_ENABLE_COMP,  /* Observation enable Complete */
RBLE_GAP_EVENT_OBSERVATION_DISABLE_COMP, /* Observation disable Complete */
RBLE_GAP_EVENT_BROADCAST_ENABLE_COMP,    /* Broadcast enable Complete */
    ...
};
```

Figure 3-5 Example of Event Type Definition

Include "rble_api.h" header to refer to definition of event structure. Callback function checks event type and executes following application sequence.

An example of receiving rBLE event of GAP function is shown as below.

file: rBLE/src/sample_simple/rble_sample_app_peripheral.c

```
#include "rble_api.h"

void app_gap_callback(RBLE_GAP_EVENT *event)
{
    switch (event->type) {
    case RBLE_GAP_EVENT_RESET_RESULT:
        break;
    case RBLE_GAP_EVENT_BROADCAST_ENABLE_COMP:
        break;
    case RBLE_GAP_EVENT_BROADCAST_DISABLE_COMP:
        break;
    ...
    }
}
```

Figure 3-6 Example of Event Type Definition

3.4.2 Registration of Callback Function

To receive rBLE events from BLE Protocol Stack, application needs to register callback function. Application registers respective callback function for each BLE Protocol Stack function used.

```
static int_t app_reset(ke_msg_id_t const msgid, void const *param,
                       ke_task_id_t const dest_id, ke_task_id_t const src_id)
{
    (void)led_onoff_init(R_LED4);

    (void)RBLE_GAP_Reset(&app_gap_callback, &app_sm_callback);

    return KE_MSG_CONSUMED;
}
```

Figure 3-7 Example of Callback function registration

Below table shows API to register callback function for each BLE Protocol Stack function.

Table 3-1 List of Callback Registration API

Function	rBLE API	Callback Registration API	Event Identifier
INIT	RBLE_Init	**RBLE_Init**	RBLE_MODE_*
GAP	RBLE_GAP_*	**RBLE_GAP_Reset**	RBLE_GAP_EVENT_*
SM	RBLE_SM_*		RBLE_SM_EVENT_*
GATT	RBLE_GATT_*	**RBLE_GATT_Enable**	RBLE_GATT_EVENT_*
VS	RBLE_VS_*	**RBLE_VS_Enable**	RBLE_VS_EVENT_*
Profile	RBLE_XXX_YYY_*	**RBLE_XXX_YYY_Enable**	RBLE_XXX_EVENT_YYY_*

4. Profile

4.1 Profile

Profile is definitions of data structure and procedure on GATT to communicate data between connected devices.

Profile has below hierarchy. Profile consists of attributes such as Service, Included Service, and Characteristic.

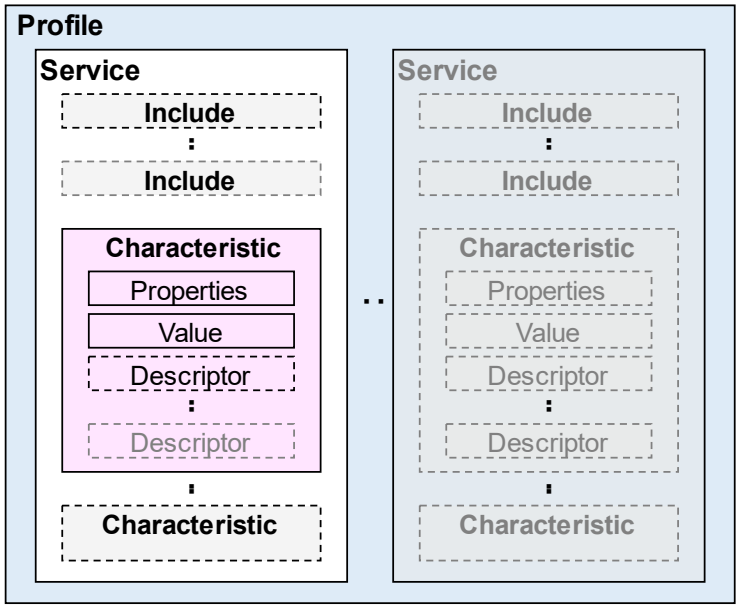


Figure 4-1 Profile Hierarchy

Service

Service is a collection of data and associated behaviors to accomplish functions or features.

Included Service

Included service is a method to incorporate another service definition on the server as part of the service.

Characteristic

Characteristic is the value used in a service along with properties and the configuration information which indicate that how to access the value and how the value is displayed or represented.

It may also contain below descriptors that describe the value or permit configuration of the server with respect to the characteristic value.

- Characteristic Extended Properties
- Characteristic User Description
- Client Characteristic Configuration
- Server Characteristic Configuration
- Characteristic Presentation Format
- Characteristic Aggregate Format

  Note: The above is part of them. Regarding to details, refer to below.
  https://www.bluetooth.com/specifications/assigned-numbers/

Client Characteristic Configuration Descriptor is important in characteristic descriptors. This descriptor is used for enabling Notification and Indication of Characteristic Value to the client.

The default value of this descriptor is set 0x0000 that means Notification and Indication are not permitted. The client sets 0x0001 and 0x0002 to permit Notification or Indication respectively. In addition, this set value of this descriptor shall be maintained between connected devices.

Table 4-1 Setting Value of Client Characteristic Configuration Descriptor

Configuration	Value	Description
No Notification / Indication	0x0000	The characteristic value is not allowed to notify / indicate.
Notification	0x0001	The characteristic value can be notified.
Indication	0x0002	The characteristic value can be indicated.

Each device behaves as Server or Client defined by Profile to communicates with each other. The server has a GATT database to define data structure of service.

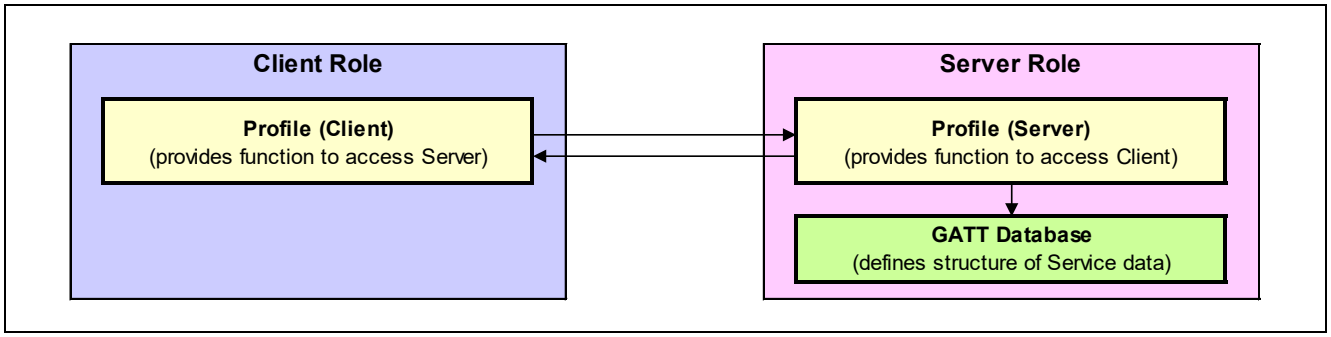


Figure 4-2 Profile Configuration

Multiple profiles are adopted by Bluetooth SIG. BLE Protocol Stack supports 15 adopted profiles, and application can use these adopted profiles by using rBLE API.

On the other hands, if application needs to use a profile that Bluetooth SIG adopted but BLE Protocol Stack doesn't support or original profile to realize original functionality, application can implement Custom Profile. Custom profile is implemented on application.

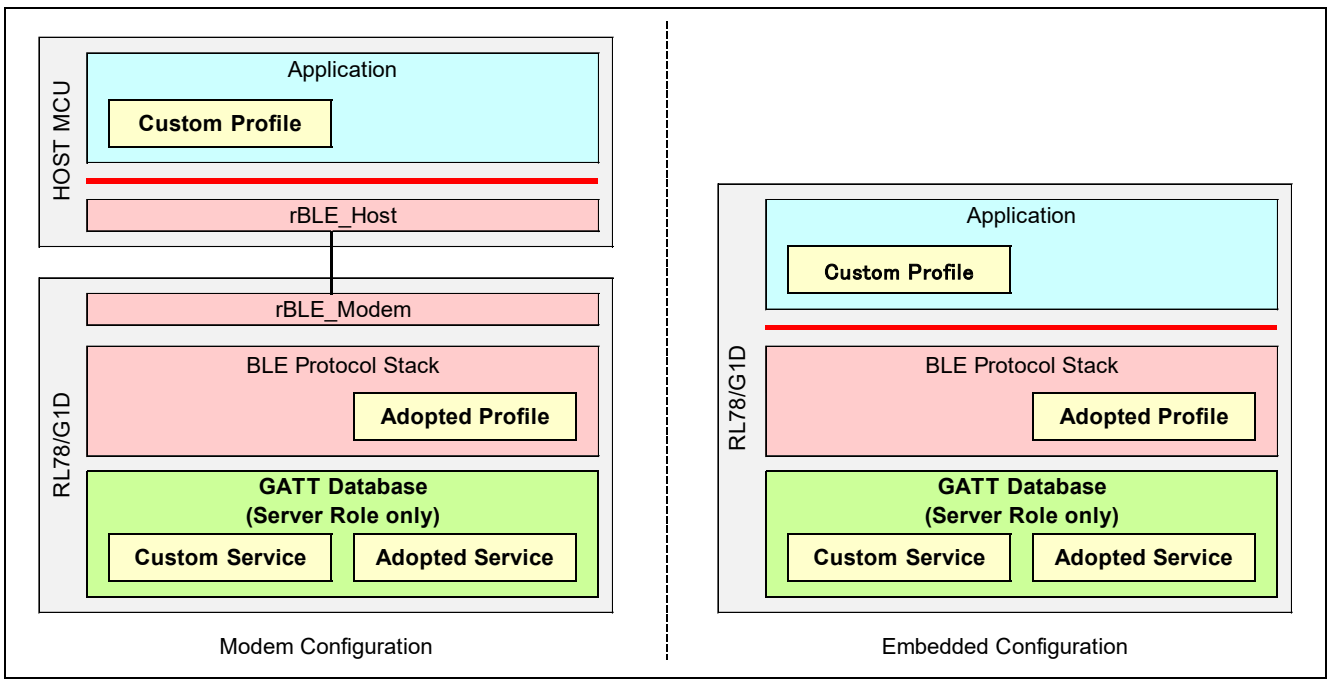


Figure 4-3 Implementation of Profiles and Services

4.2 GATT Database

GATT database is a collection of elements to define service data structure of all profiles. Application of the server role should create the database and set to BLE Protocol Stack.

Each element of the database is called "Attribute" and composes of the following four parameters.

Attribute Handle

Attribute Handle is an index to specify Attribute.
This is 16-bit value, and value is in the range of from 0x0001 to 0xFFFF.

Attribute Type

Attribute Type is a UUID to identify Attribute Value.
This is 128bit value, but the UUID which is adopted by Bluetooth SIG, is used with 16-bit.

Attribute Value

Attribute Value is the data of the Attribute.
The data structure is deferent from each Attribute Type.

Attribute Permission

Attribute Permission is configuration information to specify the access method to the Attribute Value.

GATT database is defined in two parts shown below.

- atts_desc_list_host[] in "prf_config_host.c" file : database for GAP and GATT
- atts_desc_list_prf[] in "prf_config.c" file : database for Profiles

If it is necessary to implement custom profile, user should add services and characteristics to atts_desc_list_prf[].

4.3 How to make GATT Database

To make custom profile, it is necessary to consider below items.

- What kind of functionality does custom profile perform? → Service
- What kind of data does the service handle? → Characteristic Composition
- What kind of structure does each characteristic has? → data size and structure elements of Characteristic Value
- How do the data be sent or received? → Permission of Characteristic

At first, user should design service data structure handled by custom profile, and then implement custom service to GATT database.

4.3.1 Adding Database Handle

Add the database handles for each service and each characteristic. The database handles are disclosed to the client, and used for accessing service or characteristic of server by client.

User should add new database handles before DB_HDL_MAX macro in "db_handle.h" header file.

file: renesas/src/arch/rl78/db_handle.h

```
/** Attribute database handles */
enum
{
    /* Generic Access Profile Service*/
    GAP_HDL_PRIM_SVC = 0x0001,
    GAP_HDL_CHAR_DEVNAME,
    ...

    /* Simple Sample Custom Service */
    SAMS_HDL_SVC,
    SAMS_HDL_SWITCH_STATE_CHAR,
    SAMS_HDL_SWITCH_STATE_VAL,
    SAMS_HDL_SWITCH_STATE_CCCD,
    SAMS_HDL_LED_CONTROL_CHAR,
    SAMS_HDL_LED_CONTROL_VAL,

    DB_HDL_MAX
};
```

Figure 4-4 Addition Example of Database Handle

4.3.2 Adding Database Index

Add the database indexes for each service and each characteristic. Database indexes are used for identifying elements of the database by BLE Protocol Stack.

User should add new database indexes to the last of the enumeration in "prf_config.h" header file.

file: renesas/src/arch/rl78/prf_config.h

```
/** Attribute database index */
enum
{
    /* Invalid index*/
    ATT_INVALID_IDX = 0x0000,

    /* Generic Access Profile Service */
    ...

    /* Simple Sample Custom Service */
    SAMS_IDX_SVC,
    SAMS_IDX_SWITCH_STATE_CHAR,
    SAMS_IDX_SWITCH_STATE_VAL,
    SAMS_IDX_SWITCH_STATE_CCCD,
    SAMS_IDX_LED_CONTROL_CHAR,
    SAMS_IDX_LED_CONTROL_VAL
};
```

Figure 4-5 Addition Example of Database Index

4.3.3 Definition of UUID

Define UUIDs for each service and each characteristic. The UUID for service and characteristic which have not been adopted by the Bluetooth SIG, should be defined with 128 bits random number.

User should define new UUDSs in where "prf_config.c" file can refer to.

file: rBLE/src/sample_simple/sam/sam.h

```
#define RBLE_SVC_SAMPLE_CUSTOM_SVC       {0x7A,0x8D,..., 0xF7,0xB9,0xC1,0x5B}
#define RBLE_CHAR_SAMS_SWITCH_STATE      {0x7A,0x8D,..., 0x80,0x8D,0xC1,0x5B}
#define RBLE_CHAR_SAMS_LED_CONTROL       {0x7A,0x8D,..., 0xEE,0x43,0xC1,0x5B}
```

Figure 4-6 Definition Example of UUID

4.3.4 Definition of Service

Define service. The UUID value defined in subsection 4.3.3 is assigned as an attribute value of service.

User should define new attribute value in "prf_config.c" file.

file: renesas/src/arch/rl78/prf_config.c

```
/* Service (sams) */
static const uint8_t sams_svc[RBLE_GATT_128BIT_UUID_OCTET] =
                                                    RBLE_SVC_SAMPLE_CUSTOM_SVC;
```

Figure 4-7 Definition Example of Service

4.3.5 Definition of Characteristic

Define characteristic. Characteristic has properties, attribute handle and attribute type.

User should define new characteristic in "prf_config.c" file.

file: renesas/src/arch/rl78/prf_config.c

```
/* Characteristic(sams:switch_state) */
static const struct atts_char128_desc switch_state_char = {
    RBLE_GATT_CHAR_PROP_NTF,                              Properties
    {
         (uint8_t)(SAMS_HDL_SWITCH_STATE_VAL & 0xff),     Attribute Handle
         (uint8_t)((SAMS_HDL_SWITCH_STATE_VAL >> 8) & 0xff)
    },
    RBLE_CHAR_SAMS_SWITCH_STATE                           Attribute Type(UUID)
};
```

Figure 4-8 Definition Example of Characteristics

Define characteristic value.

User should define new characteristic value in "prf_config.c".

file: renesas/src/arch/rl78/prf_config.c

```
uint8_t switch_state_char_val[RBLE_ATTM_MAX_VALUE] = {0};  Characteristic Value
struct atts_elmt_128 switch_state_char_val_elmt = {
        RBLE_CHAR_SAMS_SWITCH_STATE,                       Attribute Type(UUID)
        RBLE_GATT_128BIT_UUID_OCTET,                       UUID Length
        &switch_state_char_val[0] };                       Pointer to the Value
```

Figure 4-9 Definition Example of Characteristic Value

4.3.6 Adding to Database

Add the service and characteristic and characteristic value to GATT database.

User should add these attributes to **atts_desc_list_prf**[] in "prf_config.c" file.

file: renesas/src/arch/rl78/prf_config.c

```
const struct atts_desc atts_desc_list_prf[] =
{
    ...

    /*********************************
     * Simple Sample Service         *
     *********************************/
    { RBLE_DECL_PRIMARY_SERVICE,               Service
     sizeof(sams_svc),
     sizeof(sams_svc),
      TASK_ATTID(TASK_RBLE, SAMS_IDX_SVC),
      RBLE_GATT_PERM_RD,
     (void*)&sams_svc },
    /* Characteristic: switch_state */
    { RBLE_DECL_CHARACTERISTIC,                Characteristic
      sizeof(switch_state_char),
      sizeof(switch_state_char),
      TASK_ATTID(TASK_RBLE,SAMS_IDX_SWITCH_STATE_CHAR),
      RBLE_GATT_PERM_RD,
     (void*)&switch_state_char },
    { DB_TYPE_128BIT_UUID,                     Characteristic Value
      sizeof(switch_state_char_val),
      sizeof(switch_state_char_val),
      TASK_ATTID(TASK_RBLE,SAMS_IDX_SWITCH_STATE_VAL),
     (RBLE_GATT_PERM_NI),
     (void*)&switch_state_char_val_elmt },
    ...

    /* Reserved */
    {0,0,0,0,0,0}
};
```

Figure 4-10 Example of Adding Database

Here is end of making GATT database. Next, how to make custom profile will be explained following pages.

4.4 How to make Custom Profile

4.4.1 Server Role

As a server role, BLE software performs the following action.

- Notify the data of the Service to the Client. (Notification / Indication)
- Receive confirmation from the Client.
- Receive the writing data from the Client.
- Respond to the Write Request from the Client. (Write Response)
- Respond to the Read Request from the Client. (Read Response; but auto-answer)
- Set the data to the GATT database.

Table 4-2 shows the lists of the rBLE commands and events for GATT server role. To receive events from BLE Protocol Stack, call RBLE_GATT_Enable command at first.

Table 4-2 GATT APIs for Server Role

Commands
RBLE_GATT_Enable
RBLE_GATT_Notify_Request
RBLE_GATT_Indicate_Request
RBLE_GATT_Write_Response
RBLE_GATT_Set_Permission
RBLE_GATT_Set_Data

Events
RBLE_GATT_EVENT_HANDLE_VALUE_CFM
RBLE_GATT_EVENT_WRITE_CMD_IND
RBLE_GATT_EVENT_COMPLETE
RBLE_GATT_EVENT_RESP_TIMEOUT
RBLE_GATT_EVENT_SET_PERM_CMP
RBLE_GATT_EVENT_SET_DATA_CMP
RBLE_GATT_EVENT_NOTIFY_CMP

4.4.2 Behavior of Client Role

As a client role, BLE software performs the following action.

- Discover Service/Characteristic/Descriptor.
- Write the data to the Server. (Write Request)
- Read the data from the Server. (Read Request)
- Receive the data notification from the Server.
- Send the confirmation to the Server. (Confirmation)

Table 4-3 shows the list of rBLE commands and events for GATT client role. To receive events from BLE Protocol Stack, call RBLE_GATT_Enable command at first.

Table 4-3 GATT APIs for Client Role

Commands
RBLE_GATT_Enable
RBLE_GATT_Discovery_Service_Request
RBLE_GATT_Discovery_Char_Request
RBLE_GATT_Discovery_Char_Descriptor_Request
RBLE_GATT_Read_Char_Request
RBLE_GATT_Write_Char_Request
RBLE_GATT_Write_Rliable_Request
RBLE_GATT_Execute_Write_Char_Request

Events
RBLE_GATT_EVENT_DISC_SVC_ALL_CMP/128_CMP
RBLE_GATT_EVENT_DISC_SVC_BY_UUID_CMP
RBLE_GATT_EVENT_DISC_SVC_INCL_CMP
RBLE_GATT_EVENT_DISC_CHAR_ALL_CMP/128_CMP
RBLE_GATT_EVENT_DISC_CHAR_BY_UUID_CMP/128_CMP
RBLE_GATT_EVENT_DISC_CHAR_DESC_CMP/128_CMP
RBLE_GATT_EVENT_READ_CHAR_RESP
RBLE_GATT_EVENT_READ_CHAR_LONG_RESP
RBLE_GATT_EVENT_READ_CHAR_MULT_RESP
RBLE_GATT_EVENT_READ_CHAR_LONG_DESC_RESP
RBLE_GATT_EVENT_WRITE_CHAR_RESP
RBLE_GATT_EVENT_WRITE_CHAR_RELIABLE_RESP
RBLE_GATT_EVENT_CANCEL_WRITE_CHAR_RESP
RBLE_GATT_EVENT_HANDLE_VALUE_NOTIF
RBLE_GATT_EVENT_HANDLE_VALUE_IND
RBLE_GATT_EVENT_DISCOVERY_CMP
RBLE_GATT_EVENT_COMPLETE
RBLE_GATT_EVENT_RESP_TIMEOUT

4.4.3 Data Access by Profile

The basic communication protocol to handle the data is shown as below.

Table 4-4 Relationship of Basic Communication Protocol and Communication Direction

Command	Direction	Description
Read	Client → Server	Client requests to read a characteristic value of Server.
Write without Response	Client → Server	Client request to writes a characteristic value of Server, and needs no Response.
Write	Client → Server	Client requests to write a characteristic value of Server.
Response	Server → Client	Server responds to Write or Read request from Client.
Notification	Server → Client	Server notifies a characteristic value to Client, and needs no Confirmation.
Indication	Server → Client	Server indicates a characteristic value to Client.
Confirmation	Client → Server	Client responds to notify that Indication was confirmed.

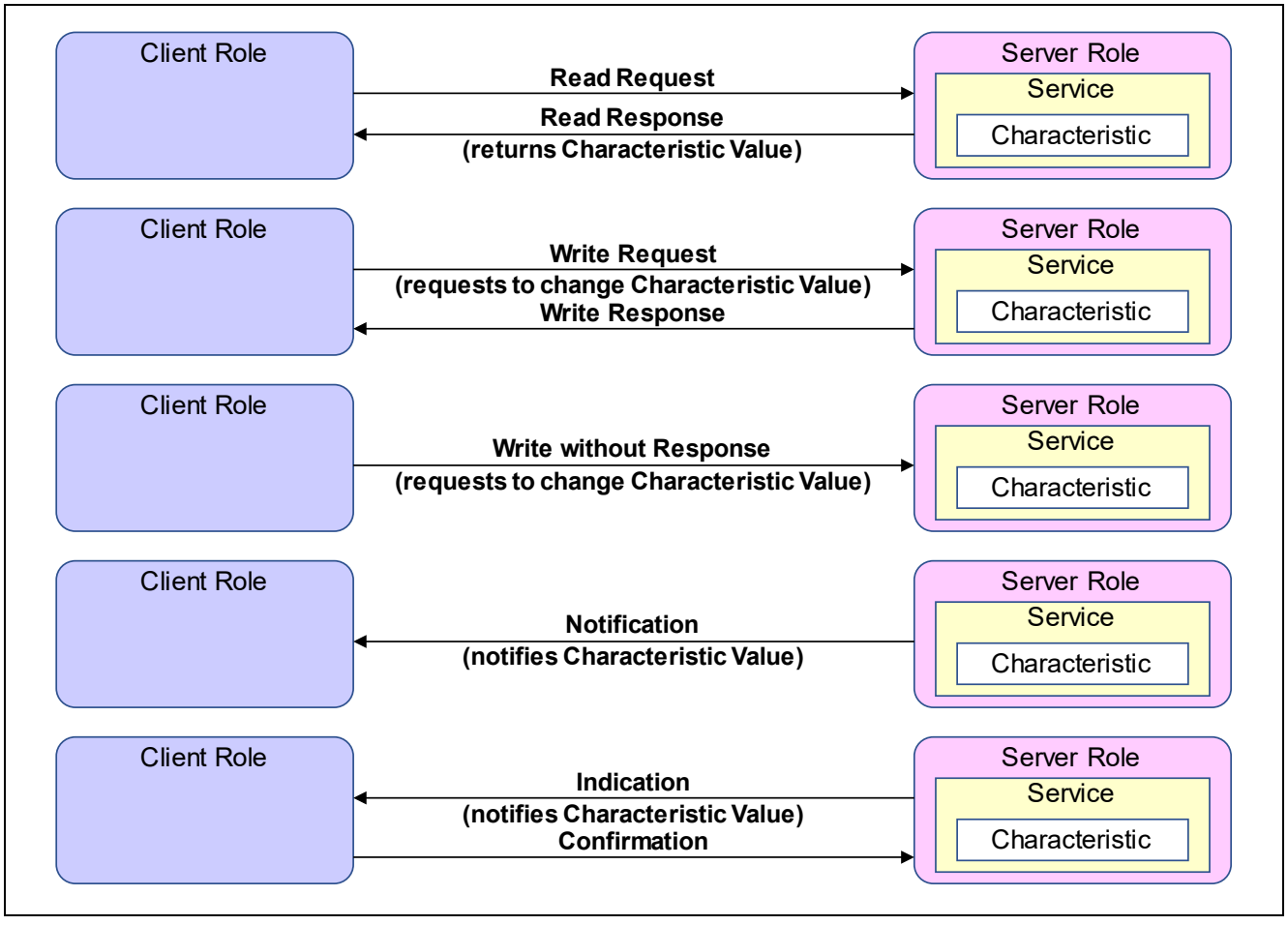


Figure 4-11 Profile Data Access

Write characteristic

Client can write data to characteristic that Server permits to write. When Client writes data to characteristic of Server, event is notified to application.

To write data to characteristic of Server, application of Client use below API.

1. RBLE_GATT_Write_Char_Request : Client writes data to characteristic value of Server.

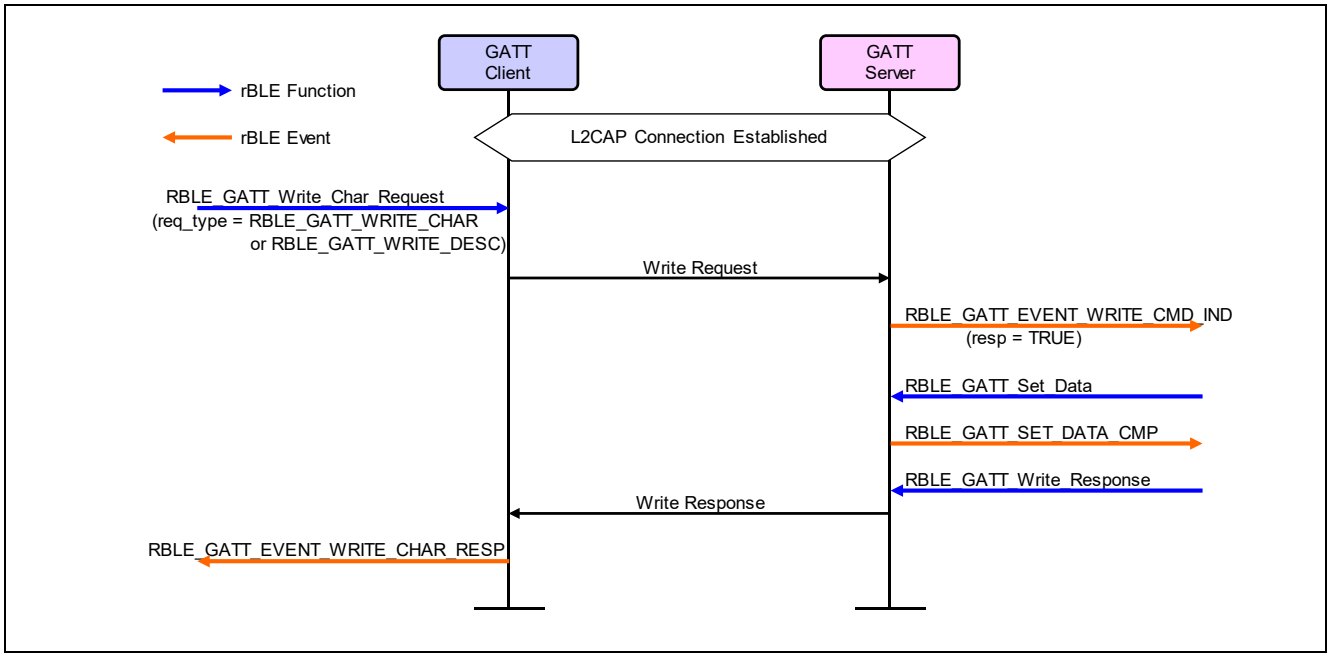


Figure 4-12 Write Characteristic

Note that above sequence is for custom profile which uses GATT function. If application uses adopted profiles provided by BLE Protocol Stack, refer to API reference manuals of each profile.

Read characteristic

Client can read data from characteristic that Server permits to read. When Client read data from characteristic of Server, Server responds automatically without notifying to application.

Application of Server can't respond to Read Request from Client. So, application of Server should update characteristic in advance. Regarding to the details, refer to section 6.6 "Update of Read Data".

To read data from characteristic of Server, application of Client use below API.

1. RBLE_GATT_Read_Char_Request : Client reads data from characteristic value of Server.

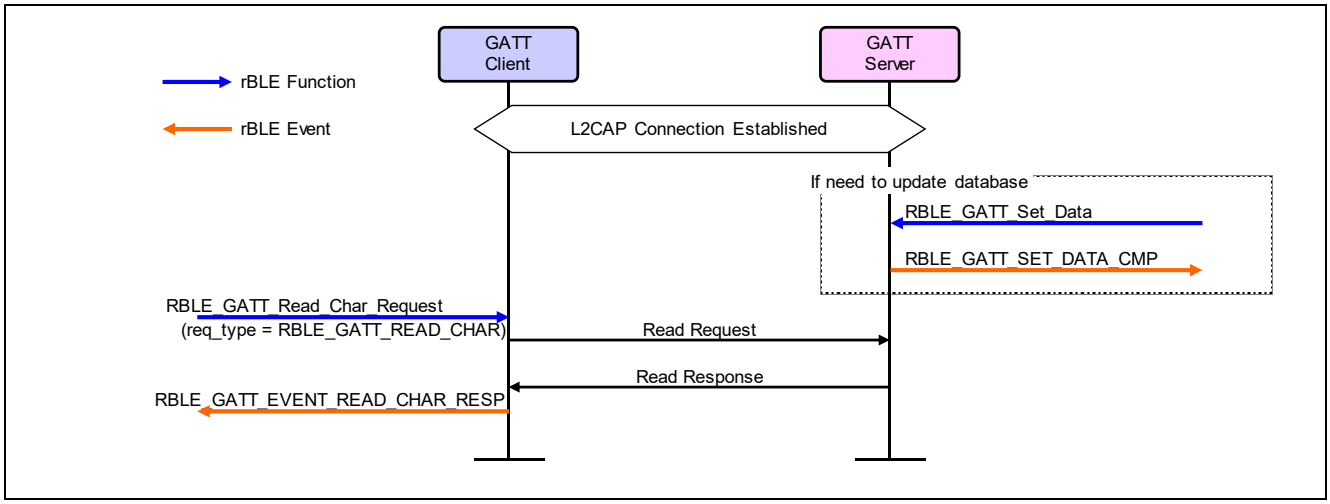


Figure 4-13 Read Characteristic

Note that above sequence is for custom profile which uses GATT function. If application uses adopted profiles provided by BLE Protocol Stack, refer to API reference manuals of each profile.

Notification characteristic

If Client permits Notification, Server can send data of characteristic to Client.

Note that Client doesn't notify receiving data. If Server needs to check if Client receives data, application of Server should use Indication.

To send Notification, application of Server use below APIs.

1. RBLE_GATT_Set_Data : Sever updates Characteristic Value which is sent by Notification.
2. RBLE_GATT_Notify_Request : Server sends Notification to Client.

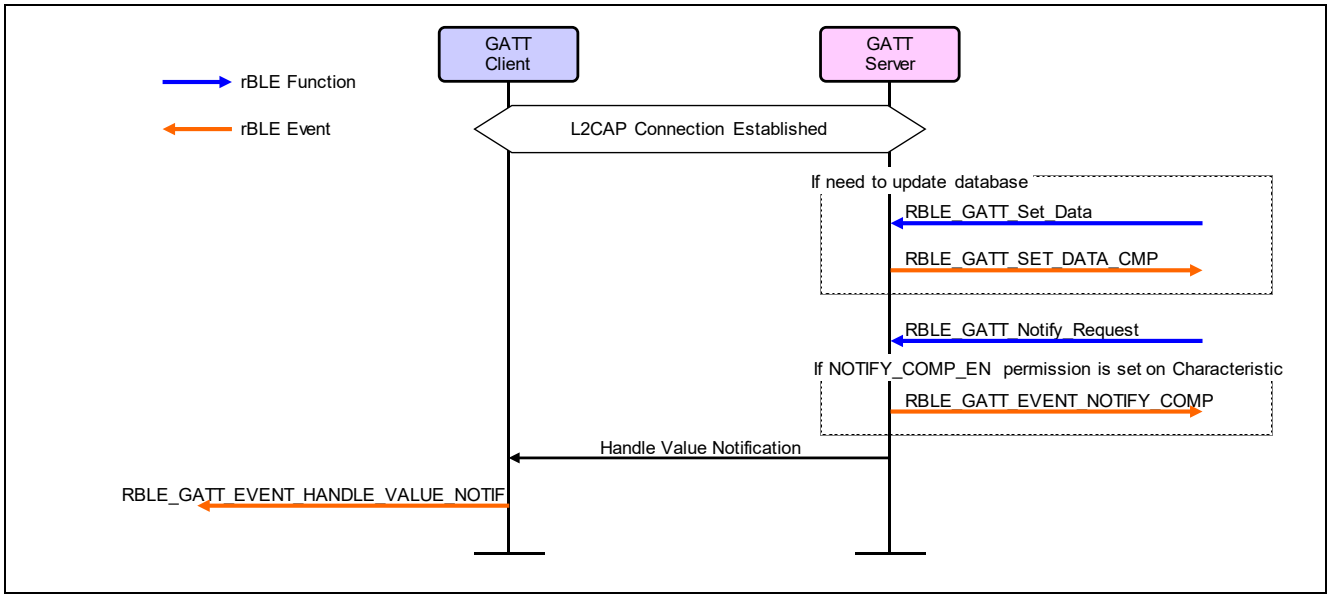


Figure 4-14 Notification Characteristic

Note that above sequence is for custom profile which uses GATT function. If application uses adopted profiles provided by BLE Protocol Stack, refer to API reference manuals of each profile.

Indication characteristic

If Client permits Indication, Server can send data of characteristic to Client.

Note that Client notifies receiving data after Indication, and data-transfer rate of Indication is lower than it of Notification. If higher data-transfer rate is needed, application of Server should use Notification.

To send Notification, application of Server use below APIs.

3. RBLE_GATT_Set_Data : Sever updates Characteristic Value which is sent by Indication.
4. RBLE_GATT_Inidicate_Request : Server sends Indication to Client.

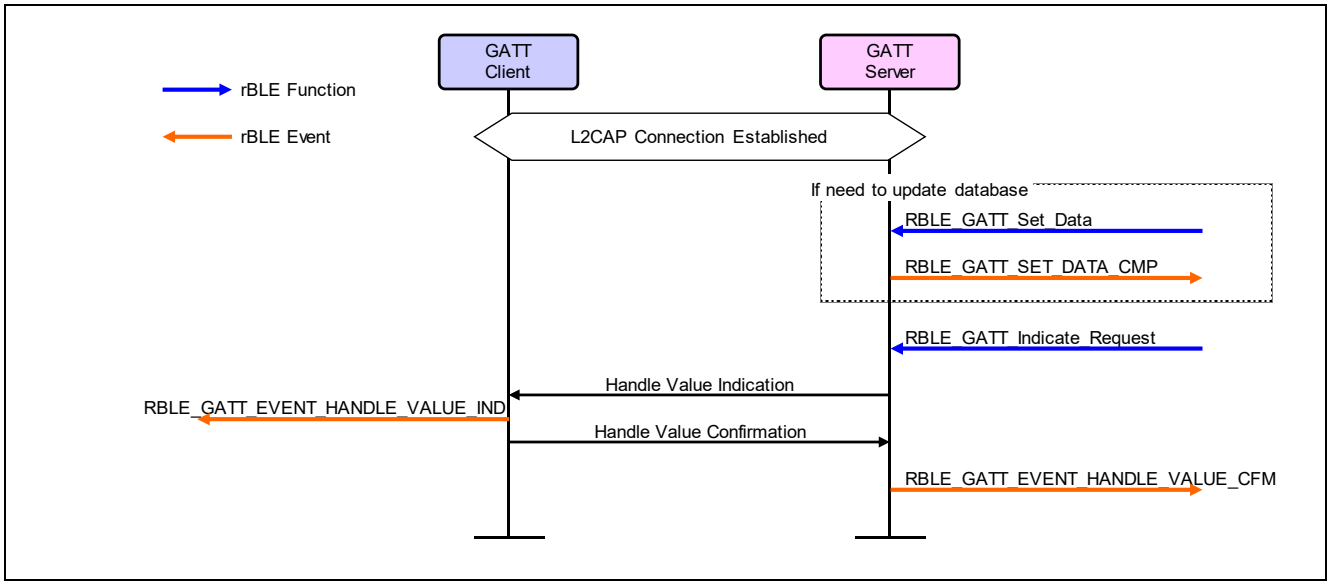


Figure 4-15 Indication Characteristic

Note that above sequence is for custom profile which uses GATT function. If application uses adopted profiles provided by BLE Protocol Stack, refer to API reference manuals of each profile.

4.4.4 Implementation of GATT Callback function

Define the GATT callback function to receive GATT event.

file: rBLE/src/sample_simple/sam/sams.c

```
static void sams_gatt_callback(RBLE_GATT_EVENT *event)
{
    switch (event->type) {
    case RBLE_GATT_EVENT_SET_DATA_CMP:
        sams_set_data_cmp_handler(event);
        break;

    case RBLE_GATT_EVENT_WRITE_CMD_IND:
        sams_write_cmd_ind_handler(event);
        break;

    default:
        Printf("unsupported event: 0x%x\n", event->type);
        break;
    }
}
```

Figure 4-16 Definition Example of GATT Callback Function

Register callback function by calling **RBLE_GATT_Enable function** before calling other GATT Command functions.

file: rBLE/src/sample_simple/sam/sams.c

```
    status = RBLE_GATT_Enable(&sams_gatt_callback);
    if (RBLE_OK != status) {
        return RBLE_STATUS_ERROR;
    }
```

Figure 4-17 Registration of GATT Callback Function

5. How application operates

This chapter describes how the simple sample program operates as an example of an application.

5.1 Simple Sample Program

The simple sample program is an application for Embedded configuration which is included in BLE software.

This application executes below items.

- After power-on, it starts the broadcast to establish a connection as a Slave role.
- After establishing a connection, it enables Slave role of Custom Profile.
- If LED control characteristic is updated, it controls lighting status of LED4 on the evaluation board.
- It notifies periodically whether switch SW4 on the evaluation board is pushed or not.
- If connection is disconnected, it starts the broadcast again.

Project files of the simple sample program is included in below place.

- BLE_Software_Ver_X_XX/RL78_G1D/Project_Source/renesas/tool/project_simple/

Regarding to usage of the simple sample program, refer to below.

- Bluetooth low energy Protocol Stack Sample Program Application Note (R01AN1375)
  https://www.renesas.com/document/apn/bluetooth-low-energy-protocol-stack-sample-program
  - chapter 6 "Usage of Simple Sample Program"

Entire processing sequence of the simple sample program is shown as below.

Regarding to each processing, refer to following pages.

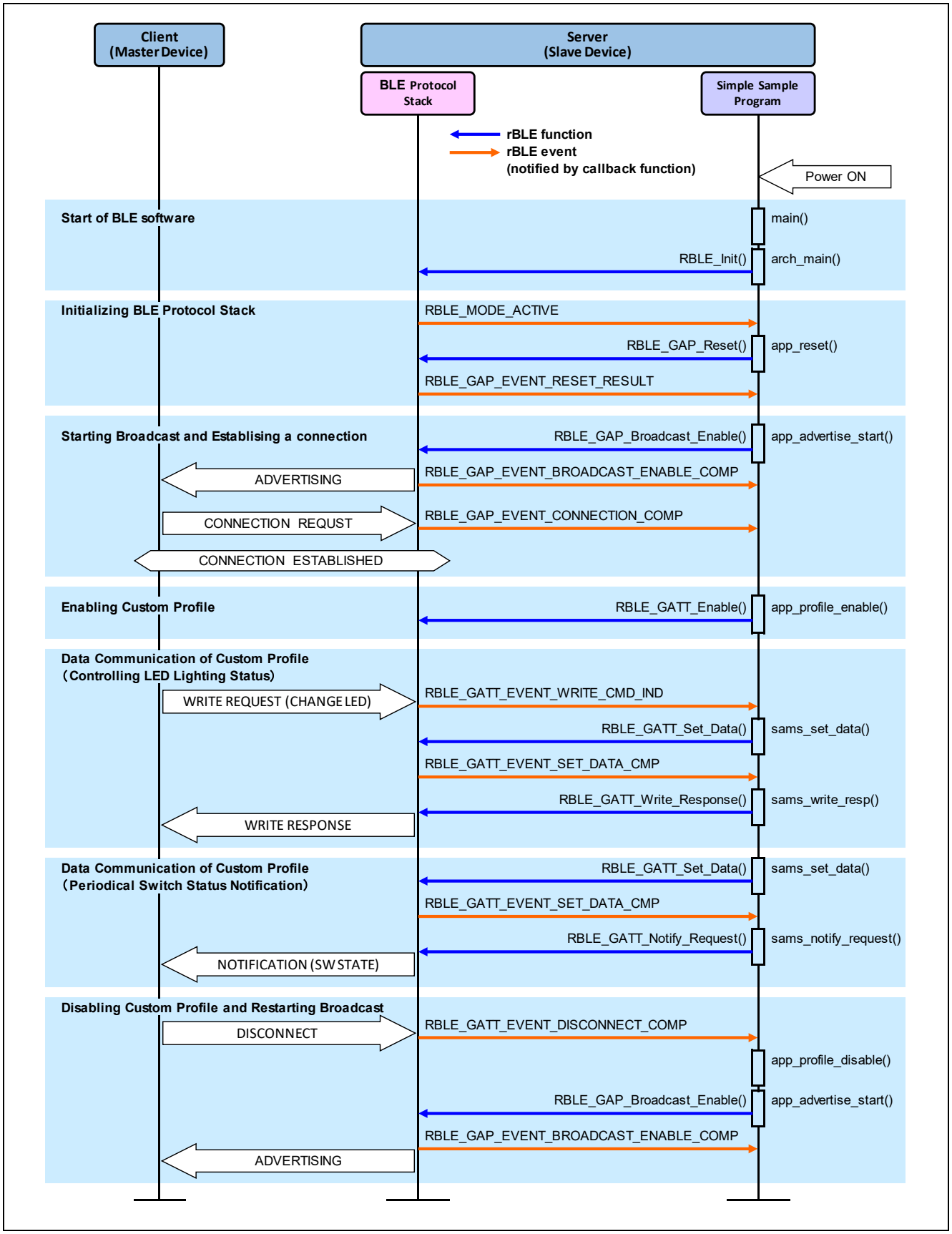


Figure 5-1 Entire Sequence of the simple sample program

5.2 Start of BLE Software

At first, BLE software initializes peripheral function drivers, RWKE, and BLE Protocol Stack. After finishing initialization, BLE Protocol Stack executes the main loop to execute its processing sequence.

As shown below, initialization processing of BLE Protocol Stack is different depends on whether it is necessary to execute FW update processing or not.

- **main function** : when FW update processing is executed
- **arch_main_ent function** : when not FW update processing but normal data communication processing is executed

After power on, **main function** is executed at first. The main function checks whether to execute FW update processing or not. If it is not necessary to execute FW update, this function calls **arch_main_ent function**.

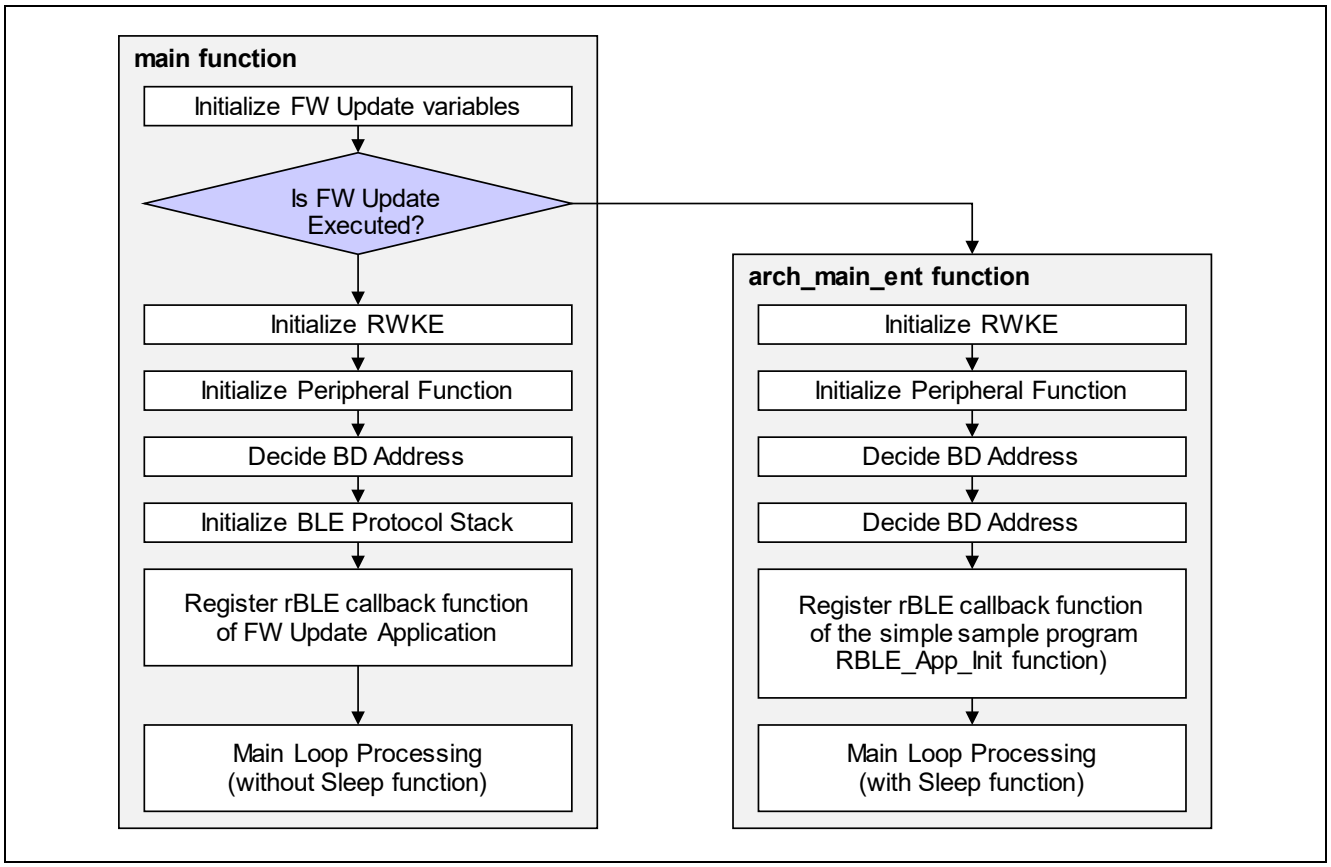


Figure 5-2 Start Flow Chart of BLE software

main function

After power on, the **main function** is executed at first. The main function checks whether to execute FW update processing or not.

If it is necessary to execute FW update, main function initializes peripheral function drivers, RWKE, and BLE Protocol Stack. After finishing initialization, this function executes FW update processing.

On the other hand, if it is not necessary to execute FW update, this function branches to call **arch_main_ent function**.

Implementation of main function is show as below.

```
_MAINCODE void main(void)
{
    ...                                          } initializing variables related to FW update

    /* during FW update? */
    if( true == check_fw_update() ) {            } checking whether it is necessary to execute FW update
        ...

        // And loop forever
        for (;;)
        {
            // schedule the BLE stack
            rwble_schedule();                    } FW update processing
        }
    }
    else
    {
        /* call arch main */
        arch_main();                             } normal data communication processing
    }
}
```

Figure 5-3 renesas/src/arch/rl78/main.c - main function

arch_main_ent function

The arch_main_ent function initializes peripheral function drivers, RWKE, and BLE Protocol Stack. And this function calls RBLE_App_Init function to initialize application of the simple sample program. After finishing initialization, this function executes the main loop.

Implementation for initializing RWKE and peripheral function drivers in arch_main_ent function is show as below.

```
void arch_main_ent(void)
{
    ...

    ble_connection_max    = BLE_CONN_MAX;   ····(1) Maximum number of connections

    // Initialize heap memory
    ke_init();                          ·····(2) RWKE initialization
    rwble_set_mem();                    ·····(3) BLE Protocol Stack memory allocation

    /*
     ********************************************************************
     * Platform initialization
     ********************************************************************
     */
    // init global variables
    variables_init();                   ·····(4) BLE Protocol stack variables initialization

    // init host database
    host_db_init();                     ·····(5) BLE Protocol stack DB initialization

    // init peak time
    peak_init( 0 );                     ·····(6) Peak current consumption notification initialization

    //init MCU clocks
    plf_init(CFG_PLF_INIT);             ·····(7) MCU unit initialization

    //init LED
    led_init();

    // Initialize the CSI21 module
    spi_init();                         ·····(8) RF unit control SPI interface initialization

    /* Initialize sleep driver */
    sleep_init();                       ·····(9) Sleep function initialization

    /* init dataflash driver */
    dataflash_init();                   ·····(10) Data Flash Library initialization

    /* get device address */
    flash_get_bda(&public_addr);        ·····(11) BD address decision
```

Figure 5-4 renesas/src/arch/rl78/arch_main.c - arch_main_ent function (1/3)

(1) Maximum number of connections

It registers the maximum number of connections which BLE software can establish to other slave devices concurrently when BLE software works as a master device.
If it is necessary to change the maximum number of connections, you should change the macro CFG_CON defined in compiler option. For example, set "CFG_CON=4" for connecting to 4 slave devices concurrently.
When BLE software works as a slave device, maximum number of connections is 1 regardless of this setting.

(2) RWKE initialization

It initializes the RWKE.

(3) BLE Protocol Stack memory allocation

It allocates memory from heap area to use BLE Protocol Stack.

(4) BLE Protocol Stack variables initialization

It initializes the global variables of BLE Protocol Stack.

(5) BLE Protocol Stack DB initialization

It initializes the GATT data-base of BLE Protocol Stack.

(6) Peak current consumption notification initialization

It initializes the peak current consumption notification.
Regarding to the details, refer to subsection 7.20.1 "Peak current consumption notification" in the user's manual (R01UW0095).

(7) MCU unit initialization

It initializes port functions and operation frequency.
If is necessary to change operation frequency, you should specify the macro in compiler option. For example, set the macro "CLK_HOCO_8MHZ" for using 8MHz operation frequency of internal high-speed oscillator. Regarding to the details, refer to subsection 6.1.3 "Changing the Operating Frequency" in the user's manual (R01UW0095).

(8) RF unit control SPI interface initialization

It initializes SPI interface to control RF unit.

(9) Sleep function initialization

It initializes Sleep function of BLE Protocol Stack.
Sleep function change MCU unit into low power consumption state, if RWKE doesn't have any message or any event and application permits BLE Protocol stack to change MCU unit into STOP mode. Regarding to the details, refer to section 6.1 "Sleep Function " in this document.

(10) Data Flash Library initialization

It initializes Data Flash Library.

(11) BD address decision

It decides BD address used by BLE Protocol Stack.
The BD address is defined in Data Flash, Code Flash or CFG_TEST_BDADDR macro, either one of BD address is used by BLE Protocol Stack. Regarding to the details refer to section 6.3 "Storing and Accessing Device Address" in this document.

Implementation for initializing BLE Protocol Stack and application in arch_main_ent function is show as below.

```
    /*
     ************************************************************************
     * BLE initializations
     ************************************************************************
     */
    // Disable the BLE core
    rwble_disable();                          ··········(12) disabling RF unit operation

    // Initialize RF
    rf_init(CFG_RF_INIT);                     ··········(13) RF unit initialization

    // input user random seed
    input_rand_value(0, userinfo_top);        ··········(14) Random seed initialization

    // Initialize BLE stack
    rwble_init(&public_addr, CFG_SCA);        ··········(15) BLE Protocol Stack initialization

    // Enable the BLE core
    rwble_enable();                           ··········(16) enabling RF unit operation
    ...

    // rBLE Initialize
    RBLE_App_Init();                          ··········(17) Application initialization
```

Figure 5-5 renesas/src/arch/rl78/arch_main.c - arch_main_ent function (2/3)

(12) disabling RF unit operation

It disables RF unit operation before executing RF unit initialization.

(13) RF unit initialization

It initializes RF unit. Following settings can be set in the argument.
- external power amplifier setting
- on-chip DC-DC converter setting
- RF slow clock setting
- high-speed clock output setting

Regarding to the details, refer to subsection 6.1.5 "Setting RF part initialization" in the user's manual (R01UW0095).

(14) Random seed initialization

It sets the initial seed value for generating pseudo-random number by rand function of standard library.
Regarding to the details, refer to subsection 6.1.5.1 "Setting seed value of the pseudo-random number" in the user's manual (R01UW0095).

(15) BLE Protocol Stack initialization

It initializes BLE protocol stack, and sets the BD address and Sleep Clock Accuracy (SCA) to BLE Protocol Stack.
Regarding to the details, refer to subsection 6.1.5 "Setting RF part initialization" in the user's manual (R01UW0095).

(16) enabling RF unit operation

It enables RF unit operation.

(17) Application initialization

It initializes application. And it initializes rBLE and registers callback functions to Generic Access Profile (GAP) and Security Manager (SM) respectively.

Implementation of the main loop in arch_main_ent function is show as below.

```
    for (;;)
    {
        ...

        // schedule the BLE stack
        rwble_schedule();                                   ····(18) RWKE scheduler

        // Checks for sleep have to be done with interrupt disabled
        GLOBAL_INT_DISABLE();                               ····(19) disabling Interrupt
        // Check if the processor clock can be gated
        if ((uint16_t)rwble_sleep() != false)               ····(20) suspending RF unit
        {
            // check CPU can sleep
            if ((uint16_t)sleep_check_enable() != false)    (21) checking Sleep permission
            {
                #ifndef CONFIG_EMBEDDED
                /* Before CPU enters stop mode, this function must be called
*/
                if ((uint16_t)wakeup_ready() != false)      (22) suspending Serial interface
                #endif // #ifndef CONFIG_EMBEDDED
                {
                    // Wait for interrupt
                    WFI();                                  ····(23) suspending MCU unit

                    #ifndef CONFIG_EMBEDDED
                    wakeup_finish();                        ····(24) resuming Serial interface
                    #endif // #ifndef CONFIG_EMBEDDED
                }
            }
        }
        // Checks for sleep have to be done with interrupt disabled
        GLOBAL_INT_RESTORE();                               ····(25) enabling Interrupt

        sleep_load_data();                                  ····(26) resuming RF unit
    }
}
```

Figure 5-6 renesas/src/arch/rl78/arch_main.c - arch_main_ent function (3/3)

(18) RWKE scheduler

It executes RWKE functions. It continues until all events and message to be executed finishes.

(19) disable interrupt

It disables interrupt before changing RF mode into low power consumption mode.

(20) suspending RF unit

It checks the operation status of RWKE and BLE Protocol Stack, and changes RF unit into low power consumption mode either SLEEP mode or DEEP_SLEEP mode. And, it notifies by return value whether it is possible to change MCU unit into STOP mode or not.

(21) checking Sleep permission

It checks if application permits to change MCU unit into STOP mode. Application can control STOP mode of MCU unit by changing the return value of sleep_check_enable function.

(22) suspending Serial interface

When BLE software is Modem configuration, it suspends serial interface used for communicating Host MCU.

(23) suspending MCU unit

It changes MCU unit into STOP mode. Stop mode is released by occurring unmasked interrupt.

(24) resuming Serial interface

When BLE software is Modem configuration, it resumes serial interface used for communicating Host MCU.

(25) enabling Interrupt

It enables interrupt.

(26) resuming RF unit

It resumes RF unit from low power consumption mode.

5.3 Initializing BLE Protocol Stack

Processing for initializing BLE Protocol Stack is shown as below.

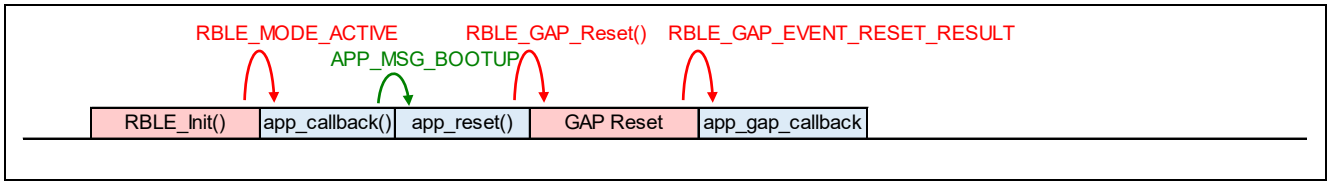


Figure 5-7 Initializing BLE Protocol Stack

RBLE_Init function

RBLE_App_Init function, which is called by arch_main_ent function, calls **RBLE_Init function** to initialize rBLE and register **app_callback function** as a rBLE callback. After completion of initializing rBLE, application can use BLE Protocol Stack functions through rBLE API.

After completion of initializing rBLE, rBLE calls the app_callback function to notify that rBLE finishes to change into **RBLE_MODE_ACTIVE** state.

```
BOOL RBLE_App_Init(void)
{
    status = RBLE_Init(&app_callback);
    ...
    ke_state_set(APP_TASK_ID, APP_RESET_STATE);
}
```

Figure 5-8 rBLE/src/sample_simple/rble_sample_app_peripheral.c - RBLE_App_Init function

When RBLE_MODE_ACTIVE is notified by rBLE, the app_callback function sends **APP_MSG_BOOTUP** message.

```
void app_callback(RBLE_MODE mode)
{
    switch (mode) {
    case RBLE_MODE_ACTIVE:
        app_msg_send(APP_MSG_BOOTUP);
        break;
    ...
    }
}
```

Figure 5-9 rBLE/src/sample_simple/rble_sample_app_peripheral.c - app_callback function

When APP_MSG_BOOTUP message is sent, **app_reset function** which is the message handler of this message, is executed.

The app_reset function calls **RBLE_GAP_Reset function** to initialize GAP function and SM function and register **app_gap_callback function** as a GAP callback and **app_sm_callback function** as a SM callback.

```
static int_t app_reset(ke_msg_id_t const msgid, void const *param,
                       ke_task_id_t const dest_id, ke_task_id_t const src_id)
{
    (void)RBLE_GAP_Reset(&app_gap_callback, &app_sm_callback);
    ...
}
```

Figure 5-10 rBLE/src/sample_simple/rble_sample_app_peripheral.c - app_reset function

5.4 Starting Broadcast and Establishing Connection

Processing for starting broadcast and establishing connection is shown as below.

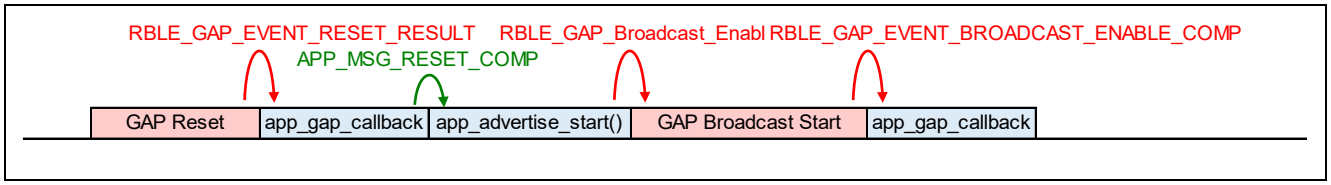


Figure 5-11 Starting Broadcast

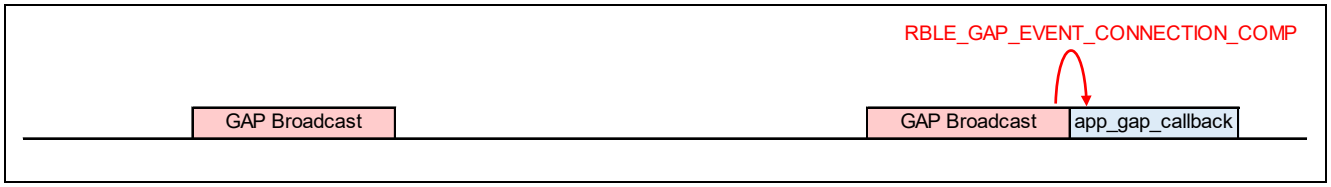


Figure 5-12 Establishing Connection

RBLE_GAP_Broadcast_Enable function

After completion of initializing GAP function of BLE Protocol Stack, rBLE calls the **app_gap_callback function** to notify **RBLE_GAP_EVENT_RESET_RESULT** event. When this event is notified by rBLE, the app_gap_callback function sends **APP_MSG_RESET_COMP** message.

By the way, if connection is established, rBLE notify **RBLE_GAP_EVENT_CONNECTION_COMP** event. When this event is notified by rBLE, the app_gap_callback function sends APP_MSG_CONNECTED message.

```
void app_gap_callback(RBLE_GAP_EVENT *event)
{
    switch (event->type) {
    case RBLE_GAP_EVENT_RESET_RESULT:
        ke_state_set(APP_TASK_ID, APP_NONCONNECT_STATE);
        app_msg_send(APP_MSG_RESET_COMP);
        break;

    case RBLE_GAP_EVENT_CONNECTION_COMP:
        ke_state_set(APP_TASK_ID, APP_CONNECT_STATE);
        app_msg_send(APP_MSG_CONNECTED);
        break;
    ...
}
```

Figure 5-13 rBLE/src/sample_simple/rble_sample_app_peripheral.c - app_gap_callback function

When APP_MSG_RESET_COMP message is sent, **app_advertise_start function** which is the message handler of this message, is executed.

The app_advertise_start function calls RBLE_GAP_Broadcast_Enable function to start broadcast for establishing a connection as a Slave.

```
static int_t app_advertise_start(ke_msg_id_t const msgid, void const *param,
                                 ke_task_id_t const dest_id, ke_task_id_t const src_id)
{
    (void)RBLE_GAP_Broadcast_Enable( ··· , &app_advertise_param);
    ...
}
```

Figure 5-14 rBLE/src/sample_simple/rble_sample_app_peripheral.c - app_advertise_start function

5.5 Enabling Custom Profile

Processing for Enabling Custom Profile is shown as below.

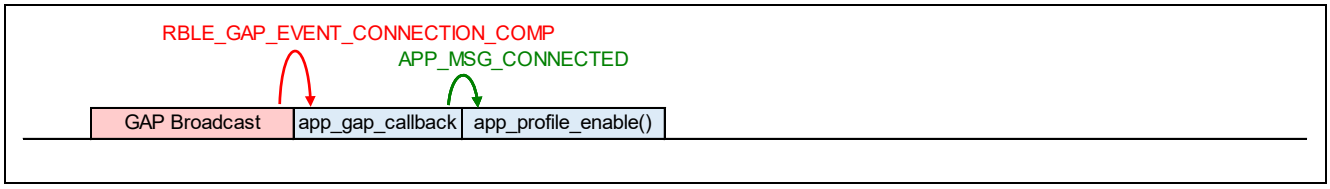


Figure 5-15 Enabling Custom Profile

RBLE_GATT_Enable function

When APP_MSG_CONNECTED message is sent, **app_profile_enable function** which is the message handler of this message, is executed.

The app_profile_enable function calls **SAMPLE_Server_Enable function** to enable the custom profile and register **app_sams_callback function** as a callback.

```
static int_t app_profile_enable(ke_msg_id_t const msgid, void const *param,
                                ke_task_id_t const dest_id, ke_task_id_t const src_id)
{
    (void)SAMPLE_Server_Enable(app_info.conhdl, RBLE_PRF_CON_DISCOVERY,
                               &samps_param, &app_sams_callback);
    ...
}
```

Figure 5-16 rBLE/src/sample_simple/rble_sample_app_peripheral.c - app_profile_enable function

The SAMPLE_Server_Enable function calls **RBLE_GATT_Enable function** to initialize GATT function of BLE Protocol Stack and register **sams_gatt_callback function** as a GATT callback.

```
RBLE_STATUS SAMPLE_Server_Enable(uint16_t conhdl, uint8_t con_type,
              SAMPLE_SERVER_PARAM *param, SAMPLE_SERVER_EVENT_HANDLER callback)
{
    sams_info.callback = callback;
    ...
    status = RBLE_GATT_Enable(&sams_gatt_callback);
    ...
}
```

Figure 5-17 rBLE/src/sample_simple/sam/sams.c - app_profile_enable function

5.6 Data Communication of Custom Profile

After establishing a connection, the simple sample program executes below operations.

- If LED control characteristic is updated, it controls lighting status of LED4 on the evaluation board.
- It notifies periodically whether switch SW4 on the evaluation board is pushed or not.

5.6.1 Controlling LED Lighting Status

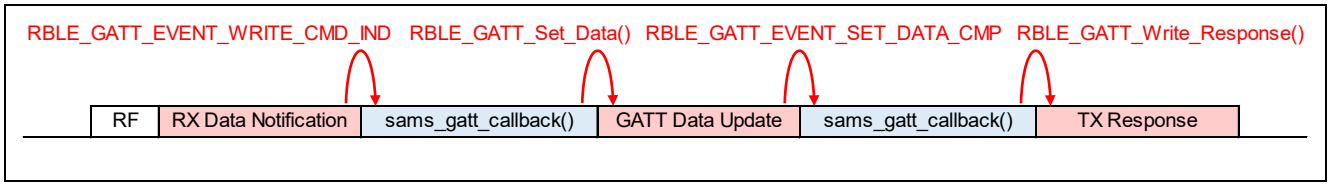


Figure 5-18 Controlling LED Lighting Status

The custom service which is implemented in the simple sample program, has a characteristic to control LED lighting status.

If this characteristic is written by client device that establishes a connection, rBLE calls **sams_gatt_callback function** to notify RBLE_GATT_EVENT_WRITE_CMD_IND event.

```
static void sams_gatt_callback(RBLE_GATT_EVENT *event)
{
    switch (event->type) {
    case RBLE_GATT_EVENT_SET_DATA_CMP:
    ...

    case RBLE_GATT_EVENT_WRITE_CMD_IND:
        ...
    }
}
```

Figure 5-19 rBLE/src/sample_simple/sam/sams.c - sams_gatt_callback function

RBLE_GATT_Set_Data function

If the custom profile is notified that any characteristic is written, it calls **RBLE_GATT_Set_Data function** to update characteristic value stored by BLE Protocol Stack with the value sent from client device.

```
static void sams_set_data(uint16_t hdl, uint16_t len, uint8_t *val)
{
    ...
    (void)RBLE_GATT_Set_Data(&data);
}
```

Figure 5-20 rBLE/src/sample_simple/sam/sams.c - sams_set_data function

RBLE_GATT_Write_Response function

After completion of updating characteristic, rBLE calls the sams_gatt_callback function to notify **RBLE_GATT_EVENT_SET_DATA_COMP** event.

When this event is notified by rBLE, the custom profile calls **RBLE_GATT_Write_Response function** to request BLE Protocol Stack to send Write Response.

```
static void sams_write_resp(void)
{
    ...
    (void)RBLE_GATT_Write_Response(&resp);
}
```

Figure 5-21 rBLE/src/sample_simple/sam/sams.c - sams_write_resp function

Furthermore, the custom profile calls app_sams_callback function to notify that a characteristic is updated. Finally, application calls **led_onoff_set function** to control LED lighting status.

```
void app_sams_callback(SAMPLE_SERVER_EVENT *event)
{
    switch (event->type) {
    case SAMPLE_SERVER_EVENT_CHG_LED_CONTROL_IND:
      if(event->param.change_led_control_ind.value == SAMPLE_LED_CONTROL_ON) {
          led_onoff_set(R_LED4, R_LED_STATE_ON);
      } else {
          led_onoff_set(R_LED4, R_LED_STATE_OFF);
      }
      break;

    ...
    }
}
```

Figure 5-22 rBLE/src/sample_simple/sam/sams.c - sams_write_resp function

5.6.2 Periodical Switch Status Notification

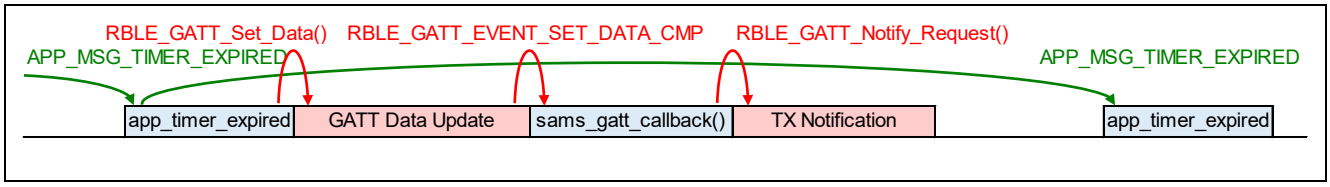


Figure 5-23 Periodical Switch Status Notification

The custom service that is implemented in the simple sample program has a characteristic to notify switch status. This characteristic is notified by sending Notification to client device.

At first, server device needs a permission by client device to send notification. If sending notification is permitted by client device, the custom profile calls **app_sams_callback function** to notify application. And then application starts timer function of RWKE.

```
void app_sams_callback(SAMPLE_SERVER_EVENT *event)
{
    switch (event->type) {
    case SAMPLE_SERVER_EVENT_WRITE_CHAR_RESPONSE:
        if (event->param.write_char_resp.value & RBLE_PRF_START_NTF) {
            ke_timer_set(APP_MSG_TIMER_EXPIRED, APP_TASK_ID,
                                                APP_SWITCH_STATE_CHECK_INTERVAL);
        }
        break;

    ...
    }
}
```

Figure 5-24 rBLE/src/sample_simple/rble_sample_app_peripheral.c - app_sams_callback function

When the timer expires, **APP_MSG_TIMER_EXPIRED** message is sent, and **app_timer_expired function** which is the message handler of this message, is executed.

The app_timer_expired function calls **SAMPLE_Server_Send_Switch_State function** to send notification. And, it starts timer again to send notification at next interval.

```
static int_t app_timer_expired(ke_msg_id_t const msgid, void const *param,
                               ke_task_id_t const dest_id, ke_task_id_t const src_id)
{
    ...
    (void)SAMPLE_Server_Send_Switch_State(app_info.conhdl, value);

    ke_timer_set(APP_MSG_TIMER_EXPIRED, APP_TASK_ID,
                                        APP_SWITCH_STATE_CHECK_INTERVAL);
}
```

Figure 5-25 rBLE/src/sample_simple/rble_sample_app_peripheral.c - app_timer_expired function

RBLE_GATT_Set_Data function

If SAMPLE_Server_Send_Switch_State function is called, the custom profile calls **RBLE_GATT_Set_Data function** to update characteristic value stored by BLE Protocol Stack with the value sent from client device.

```
static void sams_set_data(uint16_t hdl, uint16_t len, uint8_t *val)
{
    ...
    (void)RBLE_GATT_Set_Data(&data);
}
```

Figure 5-26 rBLE/src/sample_simple/sam/sams.c - sams_set_data function

RBLE_GATT_Notify_Request function

After completion of updating characteristic, rBLE calls the sams_gatt_callback function to notify **RBLE_GATT_EVENT_SET_DATA_COMP** event.

When this event is notified by rBLE, the custom profile calls **RBLE_GATT_Notify_Request function** to request BLE Protocol Stack to send Notification.

```
static void sams_notify_request(void)
{
    ...

    (void)RBLE_GATT_Notify_Request(&ntf);
}
```

Figure 5-27 rBLE/src/sample_simple/sam/sams.c - sams_notify_request function

5.7 Disabling Custom Profile and Restarting Broadcast

Processing for disabling custom profile and restarting broadcast is shown as below.

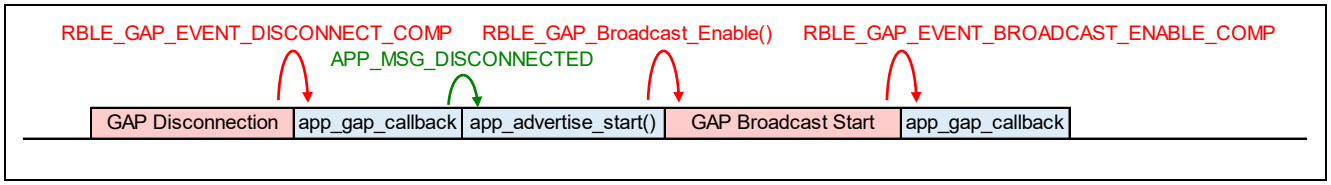


Figure 5-28 Disabling Custom Profile and Restarting Broadcast

If the connection is disconnected, GAP function of BLE Protocol Stack calls **app_gap_callback function** to notify RBLE_GAP_EVENT_DISCONNECT_COMP event.

When this event is notified by rBLE, the app_gap_callback function sends **APP_MSG_DISCONNECTED** message.

```
void app_gap_callback(RBLE_GAP_EVENT *event)
{
    switch (event->type) {
    case RBLE_GAP_EVENT_DISCONNECT_COMP:
        ke_state_set(APP_TASK_ID, APP_NONCONNECT_STATE);
        app_msg_send(APP_MSG_DISCONNECTED);
        break;
    ...
}
```

Figure 5-29 rBLE/src/sample_simple/rble_sample_app_peripheral.c - app_gap_callback function

When **APP_MSG_DISCONNECTED** message is sent, **app_timer_expired function** which is the message handler of this message, is executed.

The app_profile_disable function calls **SAMPLE_Server_Disable function** to disable the custom profile.

```
static int_t app_profile_disable(ke_msg_id_t const msgid, void const *param,
                                 ke_task_id_t const dest_id, ke_task_id_t const src_id)
{
    (void)SAMPLE_Server_Disable(app_info.conhdl);
}
```

Figure 5-30 rBLE/src/sample_simple/rble_sample_app_peripheral.c - app_profile_disable function

The SAMPLE_Server_Disable function disables the custom profile and calls **app_sams_callback function** to notify **SAMPLE_SERVER_EVENT_DISABLE_COMP** event.

When this event is notified, the app_sams_callback function sends **APP_MSG_PROFILE_DISABLED** event.

```
void app_sams_callback(SAMPLE_SERVER_EVENT *event)
{
    switch (event->type) {
    case SAMPLE_SERVER_EVENT_DISABLE_COMP:
        app_msg_send(APP_MSG_PROFILE_DISABLED);
        break;
    ...
    }
}
```

Figure 5-31 rBLE/src/sample_simple/rble_sample_app_peripheral.c - app_sams_callback function

RBLE_GAP_Broadcast_Enable function

When APP_MSG_PROFILE_DISABLED message is sent, **app_advertise_start function** which is the message handler of this message, is executed.

The app_advertise_start function calls RBLE_GAP_Broadcast_Enable function to restart broadcast.

```
static int_t app_advertise_start(ke_msg_id_t const msgid, void const *param,
                          ke_task_id_t const dest_id, ke_task_id_t const src_id)
{
    (void)RBLE_GAP_Broadcast_Enable( ··· , &app_advertise_param);
    ···
}
```

Figure 5-32 rBLE/src/sample_simple/rble_sample_app_peripheral.c - app_advertise_start function

6. Development Tips

6.1 Sleep Function of BLE software

BLE software provides Sleep function to reduce power consumption. When there isn't any event or message to be executed by RWKE and if changing into Sleep state is permitted by application, the sleep function changes MCU into low power consumption state. Regarding to the details, refer to subsection 7.20.2 "Sleep" in the user's manual (R01UW0095).

By the way, BLE software includes Console-based Sample Program to execute rBLE API commands confirm rBLE API events. Regarding to the usage of the sample program, refer to chapter 5 "Usage of Console-based Sample Program" in the Sample Program Application Note (R01AN1375).

Default setting of Console-based Sample Program disables Sleep function. How to enable Sleep function in Embedded configuration is shown as below.

Case A: Application doesn't use UART.

Change **sleep_cont_ent function** to return always TRUE.

file: renesas/src/arch/rl78/arch_main.c

```
/* sleep controll */
bool sleep_cont_ent(void)
{
    :
    (add the processing for always returning "TRUE")
}
```

Figure 6-1 Change of sleep_cont_ent Function

Case B: If application uses UART and Console.

Step 1) Change **console_can_sleep function** as shown below.

file: rBLE/src/sample_app/Console.c

Before changing:

```
/* sleep controll */
bool console_can_sleep(void)
{
    return( false );
}
```

After changing:

```
/* sleep controll */
bool console_can_sleep(void)
{
    return( !Send_Flg );
}
```

Figure 6-2 Change of console_can_sleep Function

Step 2) Change the baud rate that is set in **serial_init function** to 4800bps.

Note that if the baud rate higher than 4800bps is needed, it is necessary to use either 3-wire UART or 2-wire with branch UART communication to wake up from sleep mode. If not need to wake up, still use 2-wire UART. Regarding to the details of these Serial communication, refer to section 5.4 "Serial Communication in Modem Configuration" in the user's manual (R01UW0095).

file: renesas/src/driver/uart/uart.c

Before changing:

```
    /* MCK = fclk/n = 2MHz */
    write_sfr(SPS0L, (uint8_t)((read_sfr(SPS0L) | UART_VAL_SPS_2MHZ)));

    /* baudrate 250000bps(when MCK = 2MHz) */
    write_sfrp(UART_TXD_SDR, (uint16_t)0x0600U);
    write_sfrp(UART_RXD_SDR, (uint16_t)0x0600U);
    ------------------------------------------------------------------
```

After changing:

```
    /* MCK = fclk/n = 1MHz */
    write_sfr(SPS0L, (uint8_t)((read_sfr(SPS0L) | UART_VAL_SPS_1MHZ)));

    /* baudrate 4800bps(when MCK = 1MHz) */
    write_sfrp(UART_TXD_SDR, (uint16_t)0xCE00U);
    write_sfrp(UART_RXD_SDR, (uint16_t)0xCE00U);
```

Figure 6-3 Change of Baud Rate

Step 3) Change the stop flag that is set in **serial_init function**.

file: renesas/src/driver/uart/uart.c

Before changing:

```
    /* if baudrate is over than 4800bps, set disable */
    stop_flg = false;
```

After changing:

```
    /* if baudrate is 4800bps, set enable */
    stop_flg = true;
```

Figure 6-4 Change of Stop Flag

Step 4) Enable calling **wakeup_ready function** and **wakeup_finish function**.

file: renesas/src/arch/rl78/arch_main.c

Before changing:

```
#ifndef CONFIG_EMBEDDED
/* Before CPU enters stop mode, this function must be called */
if ((uint16_t)wakeup_ready() != false)
#endif // #ifndef CONFIG_EMBEDDED
{
    // Wait for interrupt
    WFI();

    #ifndef CONFIG_EMBEDDED
    /* After CPU is released stop mode, this function must be called */
    wakeup_finish();
    #endif // #ifndef CONFIG_EMBEDDED
}
```

After changing:

```
/* Before CPU enters stop mode, this function must be called */
if ((uint16_t)wakeup_ready() != false)
{
    // Wait for interrupt
    WFI();

    /* After CPU is released stop mode, this function must be called */
    wakeup_finish();
}
```

Figure 6-5 Enable calling wakeup Functions

Step 5) Enable registering **wakeup_init_ent function**.

file: renesas/src/arch/rl78/arch_main.c

Before changing:

```
_ACS_TBL const uint32_t access_table_ent[] =
{
    (uint32_t)arch_main_ent,
    (uint32_t)platform_reset_ent,
    (uint32_t)sleep_cont_ent,
#ifdef CONFIG_MODEM
    (uint32_t)wakeup_init_ent,
#else
    0,
#endif
    (uint32_t)RBLE_User_Set_Params,
    (uint32_t)&clk_table_ent[0],
    (uint32_t)&TASK_DESC_ent[0],
    (uint32_t)&ke_evt_hdlr_ent[0],
    (uint32_t)&ke_mem_heap_ent[0],
    (uint32_t)&ke_mem_heap_ent[BLE_HEAP_SIZE]
};
```

After changing:

```
_ACS_TBL const uint32_t access_table_ent[] =
{
    (uint32_t)arch_main_ent,
    (uint32_t)platform_reset_ent,
    (uint32_t)sleep_cont_ent,
    (uint32_t)wakeup_init_ent,
    (uint32_t)RBLE_User_Set_Params,
    (uint32_t)&clk_table_ent[0],
    (uint32_t)&TASK_DESC_ent[0],
    (uint32_t)&ke_evt_hdlr_ent[0],
    (uint32_t)&ke_mem_heap_ent[0],
    (uint32_t)&ke_mem_heap_ent[BLE_HEAP_SIZE]
};
```

Figure 6-6 Enable registering wakeup_init_ent Function

6) Enable the processing of wakep_init function.

file: renesas/src/arch/rl78/main.c

Before changing:

```
_MAINCODE void wakeup_init(void)
{
#ifdef CONFIG_MODEM
    uint32_t func_addr;

    #ifdef USE_FW_UPDATE_PROFILE
    if( false == check_fw_update() )
    #endif
    {
        func_addr = access_table[DMAIN_WAKEUP_INIT_IDX];
        ((DMAIN_WAKEUP_INIT)(func_addr))();
    }
#endif
}
```

After changing:

```
_MAINCODE void wakeup_init(void)
{
    uint32_t func_addr;

    #ifdef USE_FW_UPDATE_PROFILE
    if( false == check_fw_update() )
    #endif
    {
        func_addr = access_table[DMAIN_WAKEUP_INIT_IDX];
        ((DMAIN_WAKEUP_INIT)(func_addr))();
    }
}
```

Figure 6-7 Enable wakeup_init Function

6.2 Bluetooth Device Address Supported by BLE software

BLE software supports three BD address types. Regarding to the details of these types, refer to subsection 7.2.4 "Bluetooth Device Address" in the user's manual (R01UW005).

Public Device Address

This is Device unique address assigned by IEEE. It is the same as MAC address.

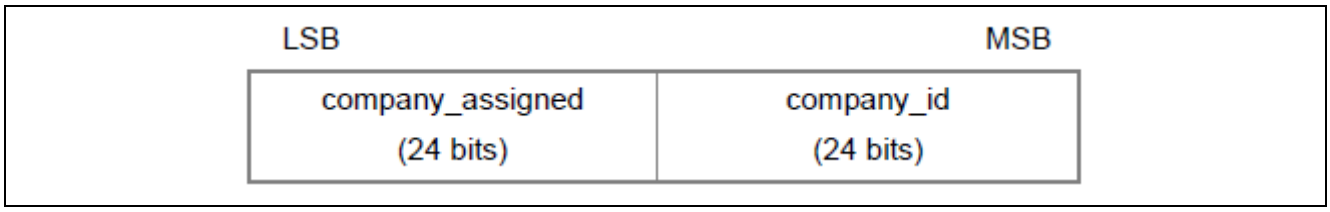


Figure 6-8 Format of public device address

Static Device Address

This is a Random fixed address. Handled in the same way as the Public Address to identify the device.

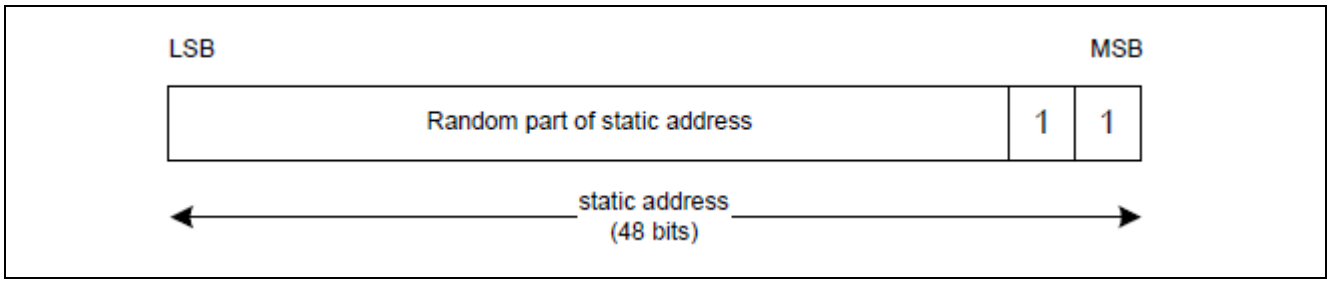


Figure 6-9 Format of static device address

Resolvable Private Address

This is Random address. Using the IRK for the identification of the device. It is recommended to change once in every 15 minutes.

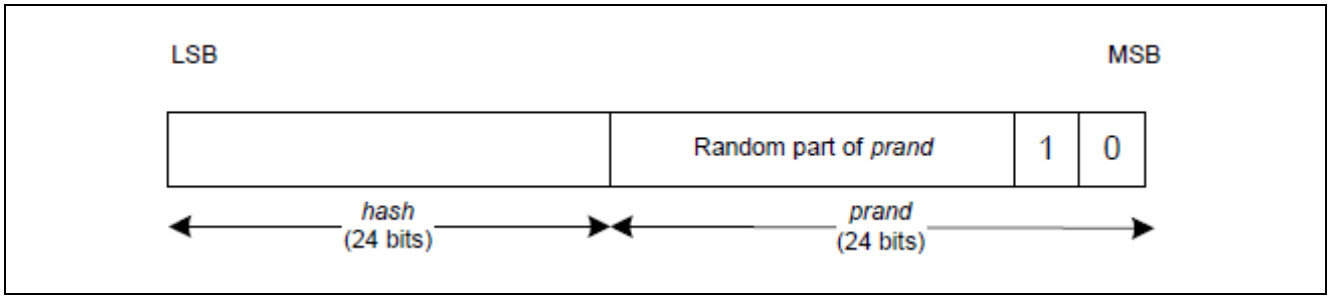


Figure 6-10 Format of resolvable private address

6.3 Storing and Accessing Device Address

Public Device Address

Public device address can be stored in three different ways: Data Flash area, Customer-specific information area and macro defined with `CFG_TEST_BDADDR`.

- Data Flash area : Device address is managed as "Data ID = 1" by Data flash library.
- Customer-specific Information area : Device address is stored in last block of Code Flash.
- CFG_TEST_BDADDR macro : Device address is defined in "config.h" header file.

It is possible that manufacturing process writes default device address to the customer-specific information area and then application writes new device address to data flash area to change Public Device Address. Note that device address defined by CFG_TEST_BDADDR macro should be used only in developing.

Figure 6-11 shows the processing flow to decide Public Device Address in initialization of BLE software.

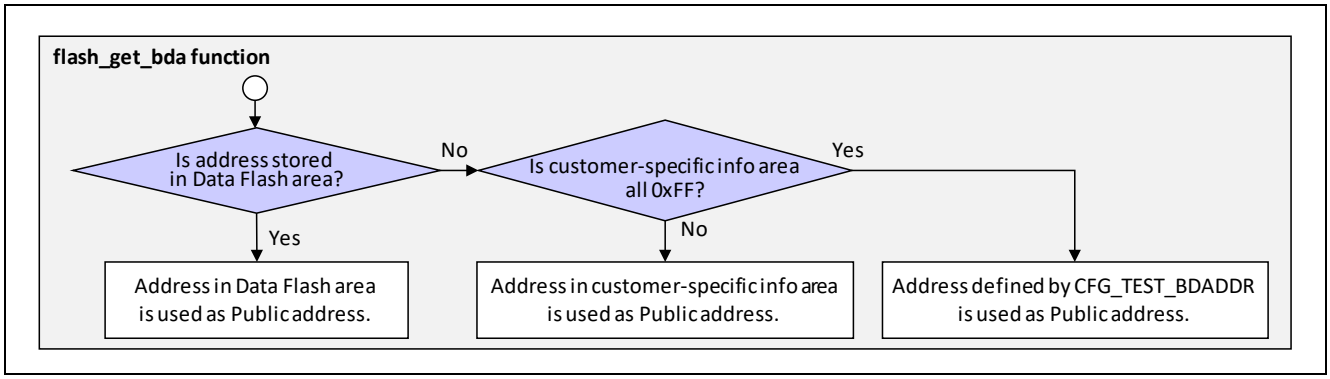


Figure 6-11 Deciding Flow of Public Device Address

Applications can write Public Device Address to Data Flash area through API. To write the Public Device Address to Data Flash area, call APIs in accordance with the sequence shown in Figure 6-12. After restarting RL78/G1D, the written device address is used by BLE software.

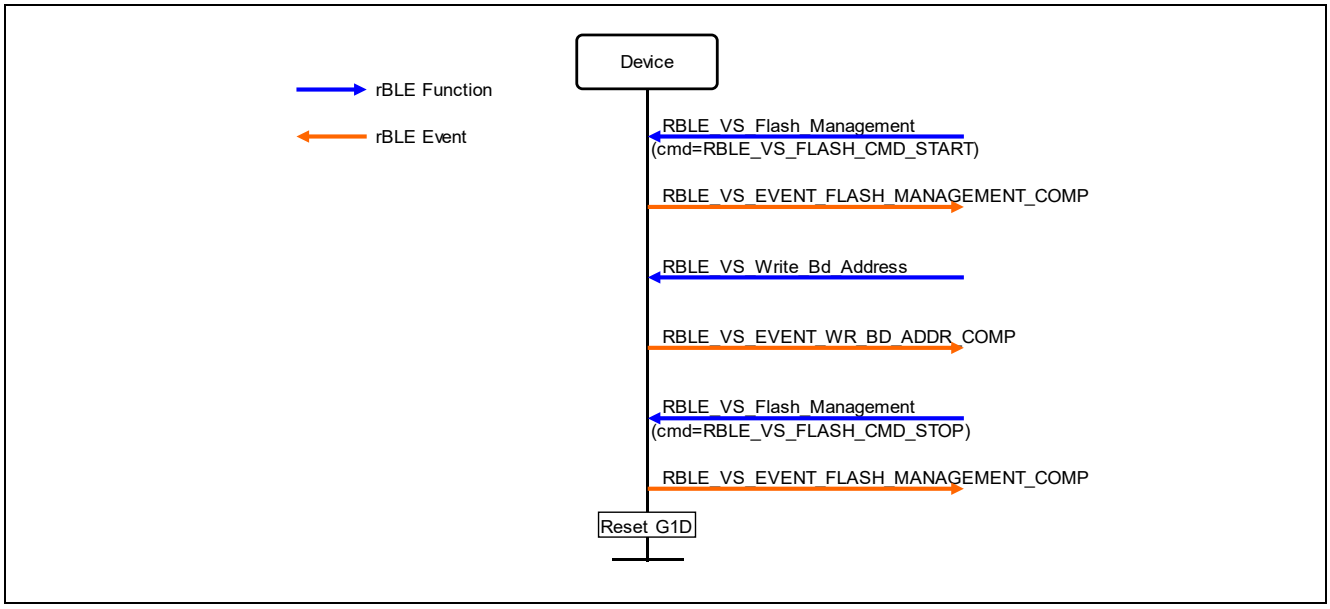


Figure 6-12 How to write Public Device Address

Static Device Address

By storing the random value to data flash area, it is possible to use the same value as Static Device Address.

To store Static Device Address to data flash area, define new data ID and data size in the descriptor of data flash library to store device address as shown in Figure **6-13** and Figure **6-14**.

file: renesas/src/driver/dataflash/eel_descriptor_t02.h

```
/* data id for descriptor */
enum
{
    EEL_ID_BDA = 0x1,
    EEL_ID_STATIC_BDA,
    EEL_ID_END
};
```

Figure 6-13 Definition Example of Data ID

file: renesas/src/driver/dataflash/eel_descriptor_t02.c

```
_EEL_CNST __far const eel_u08 eel_descriptor[EEL_VAR_NO+3] =
{
    (eel_u08)(EEL_VAR_NO),     /* variable count           */  \
    (eel_u08)(BD_ADDR_LEN),    /* id=1: EEL_ID_BDA         */  \
    (eel_u08)(BD_ADDR_LEN),    /* id=2: EEL_ID_STATIC_BDA  */  \
    (eel_u08)(0x00),           /* zero terminator          */  \
};
```

Figure 6-14 Definition Example of Descriptor

By defining to the descriptor, application can write and read Static Device Address in data flash area through APIs.

To write Static device address to data flash area, call API in accordance with the procedure as shown in Figure **6-15**.

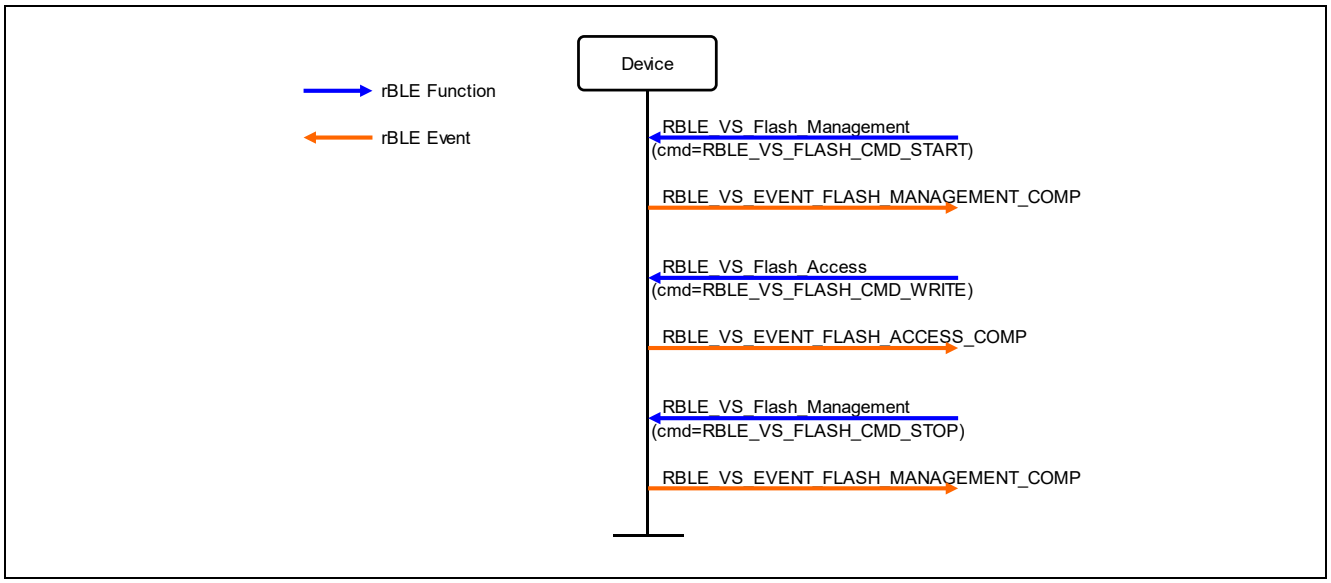


Figure 6-15 How to Write Static Device Address

To read Static device address from data flash area, call API in accordance with the procedure as shown in Figure **6-16**.

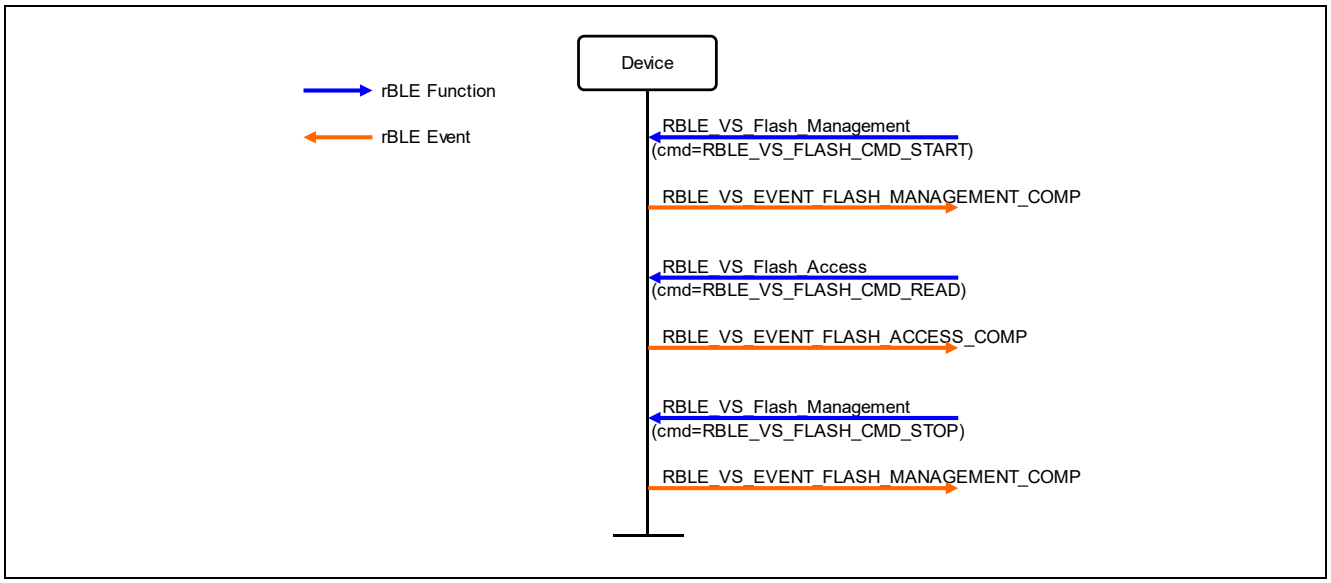


Figure 6-16 How to Read Static Device Address

6.4 Broadcast Start Sequence

The sequence to start broadcast is different from each device address type. Sequences to start broadcast for each device address type are shown as below.

Public Device Address

Figure **6-17** shows the sequence to advertise with Public Device Address.

To start broadcast, call BLE_GAP_Broadcast_Enable function with own_addr_type=RBLE_ADDR_PUBLIC.

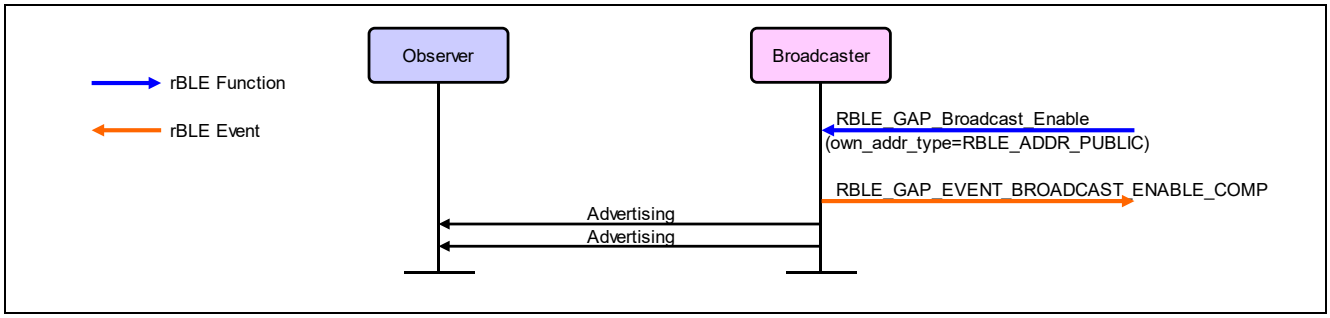


Figure 6-17 Broadcast with Public Device Address

Static Device Address

Figure **6-18** shows the sequence to advertise with Static Device Address.

To set Static Device Address which is generated by application, call RBLE_GAP_Set_Random_Address function.

To Start broadcast, call BLE_GAP_Broadcast_Enable function with own_addr_type=RBLE_ADDR_RAND.

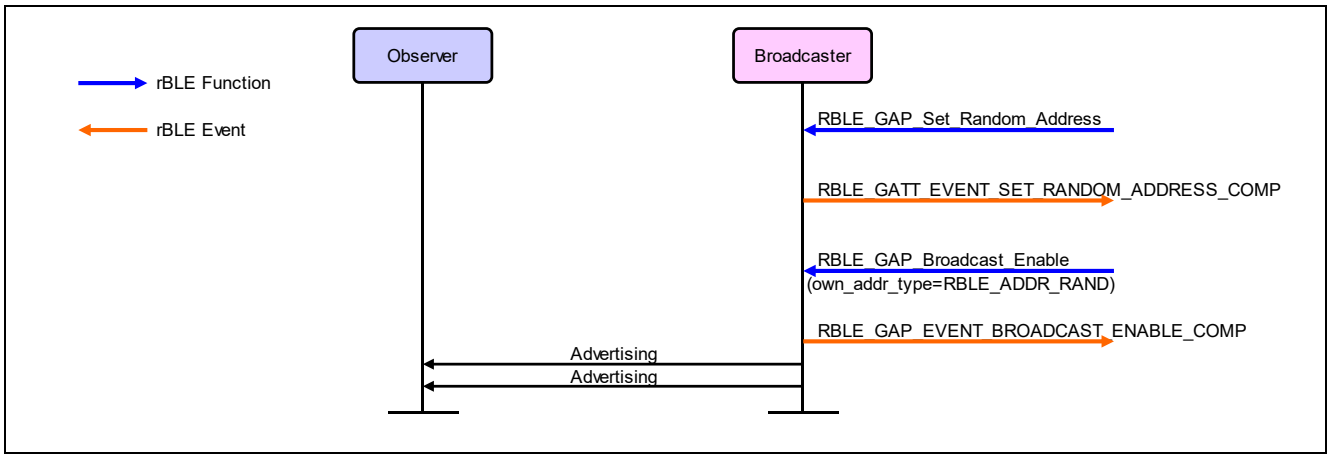


Figure 6-18 Broadcast with Static Device Address

Resolvable Private Address

Figure 6-19 shows the sequence to advertise with Resolvable Private Address.

To set IRK prepared by application, call RBLE_SM_Set_Key function with Key_code = RBLE_SMP_KDIST_IDKEY.

To generate Resolvable Private Address, call RBLE_GAP_Set_Privacy_Feature function with priv_flag = RBLE_PH_PRIV_ENABLE or RBLE_BCST_PRIV_ENABLE.

To start broadcast, call BLE_GAP_Broadcast_Enable function with own_addr_type = RBLE_ADDR_RAND.

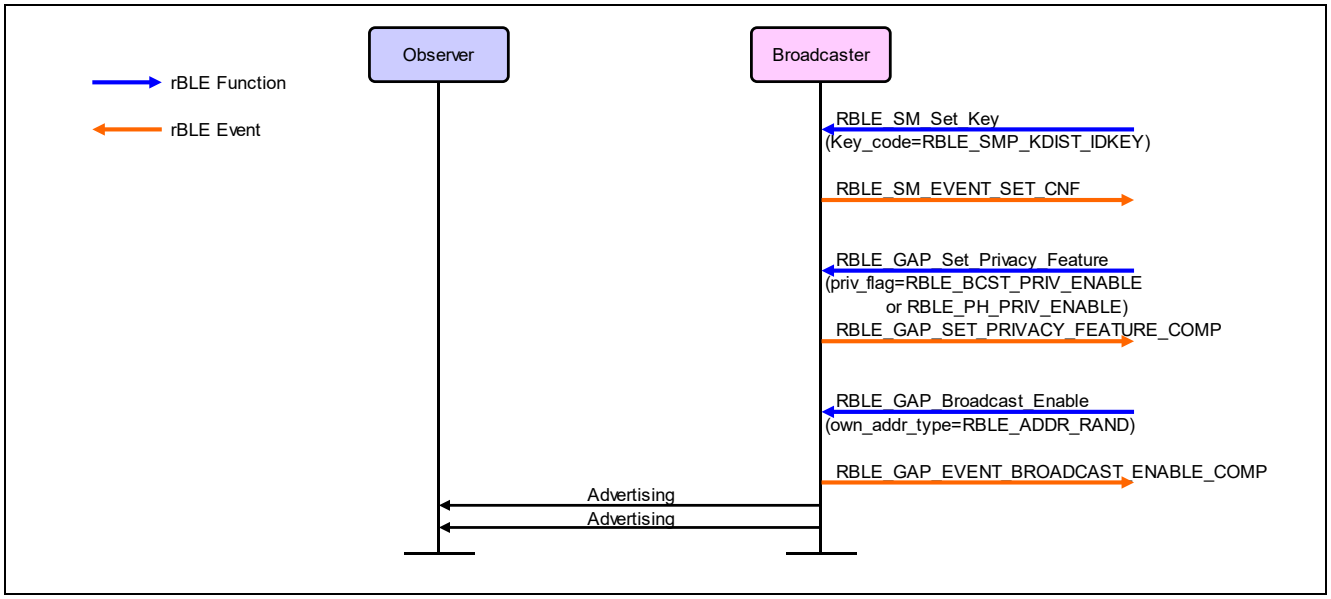


Figure 6-19 Broadcast with Resolvable Device Address

6.5 Usage of Bluetooth Device Name

Bluetooth Device Name is a user-friendly name that can be seen from remote devices. User can identify remote device by checking device name.

6.5.1 Device Name of Local Device

To inform own device name to remote device, set the device name to Device Name Characteristic in the database. Remote device can read the device name from the database after establishing the connection.

BLE software provides two ways to set device name to the database.

The first method is to use Device Name area of customer-specific information. The device name that has been stored into the customer-specific information can be used by accessing to the address 0x3FC06.

The second method is to use the GAP_DEV_NAME macro defined in "prf_config.h" header file. If device name isn't set in the customer-specific information, the macro definition is used as Device Name.

Table 6-1 Device Name Area

Information	Address	Size	Notes
Device Name	0x3FC06	66 bytes	Bluetooth Device Name User-friendly name for identifying the devices 0x3FC06: Length of Device Name (1 to 65) 0x3FC07 to 0x3FC48: Device Name (string of UTF-8)

file: renesas/src/arch/rl78/prf_config.h

```
#define GAP_DEV_NAME      "Renesas-BLE"
```

Figure 6-20 Definition Macro of Device Name

Advertising can inform own device name to remote devices before establishing a connection. To inform device name by advertising, set Local Name of AD type and device name to adv_data, and call RBLE_GAP_Broadcast_Enable function.

Note: The device name or unique data should not be contained into the advertising data on the privacy-enabled device to avoid that the device will be recognized from a malicious device.

6.5.2 Device Name of Remote Device

As shown in Figure 6-21 and Figure 6-22, by using RBLE_GAP_Get_Remote_Device_Name function, application can acquire device name of remote device regardless of whether connected or non-connected.

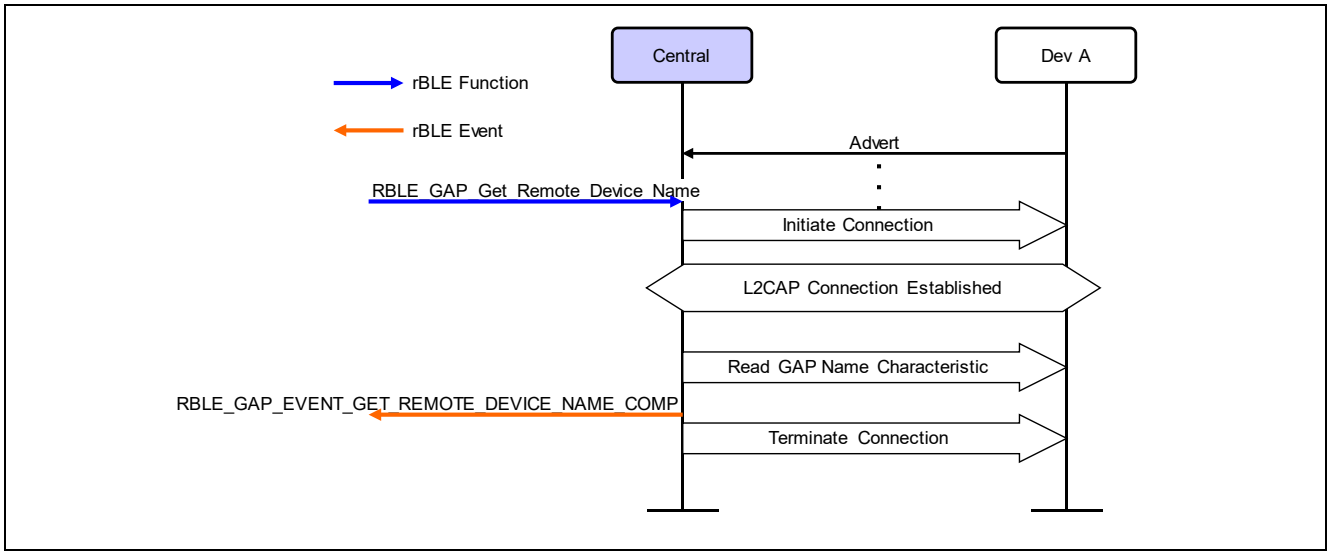


Figure 6-21 Acquiring Device Name when not connected

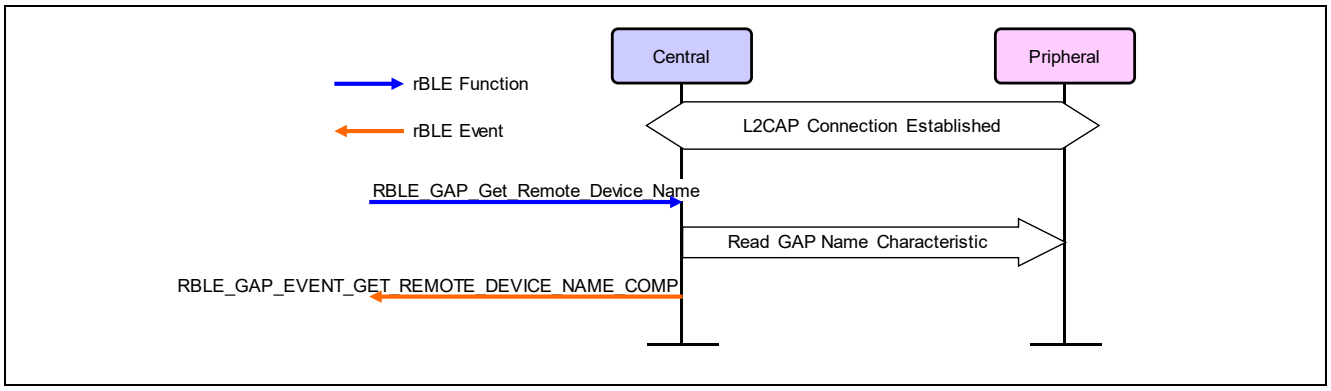


Figure 6-22 Acquiring Device Name when connected

6.6 Update of Read Data

When BLE Protocol receives Read Request from GATT client, the stack returns data required by Read Response automatically without notifying to user application. So, application can't update data notified by Read Response.

As shown in Figure 6-23, application must update the characteristic value by calling RBLE_GATT_Set_Data function in advance.

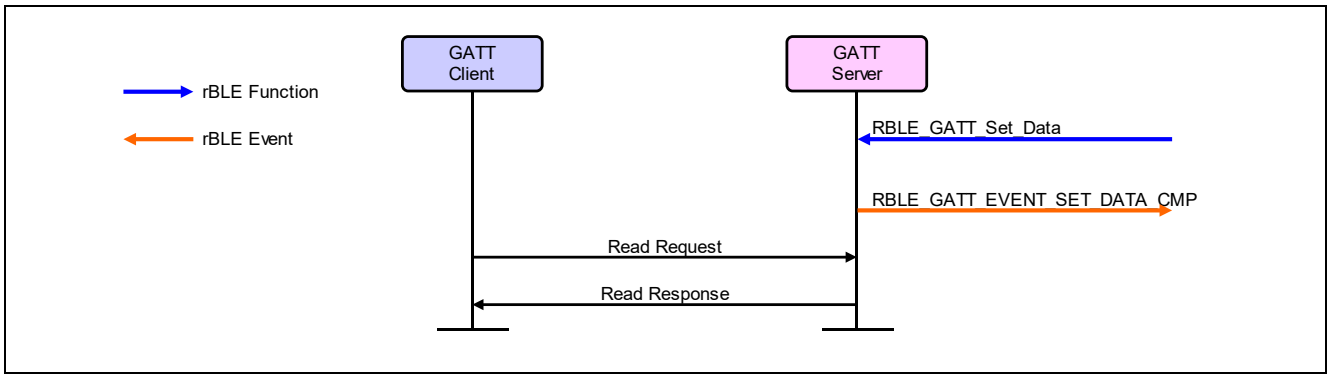


Figure 6-23 Read Timing of Updated Data

7. Appendix

7.1 How to Add Characteristic to Custom Profile

This chapter explains how rBLE API and rBLE Event are used in server processing using the Sample Custom Service used in the Simple Sample Program. It will also explain how to add a new characteristic to the Sample Custom Service. Also check the characteristic operation using the RL78/G1D evaluation board (RTK0EN0001D01001BZ) (hereinafter called "Evaluation board") and GATTBrowser of the smartphone application.

For the Simple Sample Program, refer to "5.1 Simple Sample Program" in this manual.

[Contents]

- 7.1.1 Explains the definition of the Sample Custom Service.
- 7.1.2 Explains the structure of the database.
- 7.1.3 Explains the server processing of the Sample Custom Service.
- 7.1.4 Explains how to add characteristic to the Sample Custom Service.
- 7.1.5 Explains how to add the server profile API and peripheral application processing so that the added characteristic can communicate with the client.
- 7.1.6 How to test added characteristic using a smartphone.
- 7.1.7 Explains how to make the Switch State Characteristic correspond from Notification to Indication.
- 7.1.8 How to test changed characteristic from Notification to Indication using a smartphone.

"7.1.4 Adding Characteristic", "7.1.5 Adding Server Profile API and Peripheral Application" and "7.1.7 Customize from Notification to Indication" requires knowledge of the profile configuration, also refer to "4 Profile". The listed source code uses the version of the BLE protocol stack below.

- BLE Protocol Stack
  "Bluetooth® low energy Protocol Stack (Ver.1.21)"
  https://www.renesas.com/document/lbr/bluetooth-low-energy-protocol-stack-ver121

[Related Documents]

- Embedded Configuration Sample Program
  "Bluetooth® low energy Protocol Stack Embedded Configuration Sample Program"(R01AN3319)
  https://www.renesas.com/document/scd/bluetooth-low-energy-protocol-stack-embedded-configuration-sample-program
- RL78/G1D Evaluation Board (RTK0EN0001D01001BZ)
  "RL78/G1D User's Manual: Evaluation Board"(R30UZ0048)
  https://www.renesas.com/document/man/rl78g1d-users-manual-evaluation-board
- GATTBrowser
  "GATTBrowser for Android Smartphone Application Instruction manual"(R01AN3802)
  https://www.renesas.com/document/apn/gattbrowser-android-smartphone-application-instruction-manual
  "GATTBrowser for iOS Smartphone Application Instruction manual"(R21AN0017)
  https://www.renesas.com/document/apn/gattbrowser-ios-smartphone-application-instruction-manual

7.1.1 Definition of Sample Custom Service

The definition of the Sample Custom Service used in the Simple Sample Program is shown below. For details of definition, refer to "Bluetooth® Low Energy Protocol Stack Embedded Configuration Sample Program - 5.4 Sample Custom Service Definition" (R01AN3319).

Table 7-1 Sample Custom Service definition of Simple Sample Program

Type	Value	Permission
Sample Custom Service		
Primary Service Declaration (0x2800)	UUID: 5BC1B9F7-A1F1-40AF-9043-C43692C18D7A	Read
Switch State Characteristic		
Characteristic Declaration (0x2803)	Property: Notification UUID: 5BC18D80-A1F1-40AF-9043-C43692C18D7A	Read
Characteristic Value	1 [octet]	Notification
Client Characteristic Configuration Descriptor (0x2902)	2 [octet]	Read, Write
LED Control Characteristic		
Characteristic Declaration (0x2803)	Property: Read, Write UUID: 5BC143EE-A1F1-40AF-9043-C43692C18D7A	Read
Characteristic Value	1 [octet]	Read, Write

(1) **Switch State Characteristic**

The Switch State Characteristic is a characteristic that sends switch status to the client with Notification. The client characteristic configuration descriptor informs the application that Notification enable/disable setting has been written from the client.

The application calls the API (SAMPLE_Server_Send_Switch_State) that transmits the pressed/released state of the switch on the evaluation board in response to Notification enable/disable notified in the event.

The SAMPLE_Server_Send_Switch_State() API stores switch press/release status in the Switch State Characteristic value and sends it to the client by Notification.

(2) **LED Control Characteristic**

The LED Control Characteristic is a characteristic that allows the client to control LED ON/OFF.

The LED Control Characteristic Value is written from the client LED ON/OFF status. Then it informs the application by event that it was written.

The application controls LED on the evaluation board according to the status of the LED notified in the event.

7.1.2 Database Structure

In order to implement a custom profile such as "Figure 7-1 Sample Custom Service GATT Database Structure" into a program, create a GATT database (hereinafter called "Database") consisting of services and characteristic.

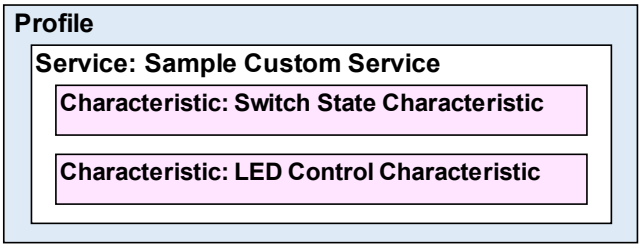


Figure 7-1 Sample Custom Service GATT Database Structure

The database of the BLE protocol stack is defined by a structure array, and it becomes a profile by defining services, characteristic, etc. for elements. A variable in the database structure array is declared in the following source file, and a GATT base profile is defined.

folder	\Renesas\BLE_Software_Ver_X_XX\RL78_G1D\Project_Source\renesas\src\arch\rl78
file	prf_config.c
value	struct atts_desc atts_desc_list_prf[]

(1) **Sample Custom Service Database Configuration**

Within the database structure array, the custom profile consists of a combination of several elements that set the attribute type UUID.

● Database

```
const struct atts_desc atts_desc_list_prf[] =
{
    { Attribute type UUID, Length of attribute value stored variable, Length of Attribute value stored variable,
                                Attribute task ID, Attribute permission, Pointer of attribute value stored variable },
    { Attribute type UUID, ... },
    { Attribute type UUID, ... },
            :
    /* Reserved */
    {0,0,0,0,0,0}
};
```

Attribute value stored variable (The variable used differs depending on the attribute type UUID)

● Primary service

uint8_t 16 octets array: Set 128bit UUID of service

● Characteristic

atts_char128_desc structure: Set property, attribute handle and 128bit UUID of characteristic

● 128bit Characteristic UUID

atts_elmt_128 structure: Set 128bit UUID of characteristic, length of 128bit UUID and pointer of communication buffer

● Client characteristic configuration descriptor

uint16_t value: Client sets whether to send notification or indication

Figure 7-2 Database Structure Array

The attribute type UUID used to configure a custom profile is shown below.

Table 7-2 Attribute Type UUID

Attribute Type UUID Name	Definition	UUID Value
Primary Service	RBLE_DECL_PRIMARY_SERVICE	0x2800
Characteristic	RBLE_DECL_CHARACTERISTIC	0x2803
128bit Characteristic UUID	DB_TYPE_128BIT_UUID	0xffff
Client Characteristic Configuration Descriptor	RBLE_DESC_CLIENT_CHAR_CONF	0x2902

* The above is the minimum definitions that configure a custom profile. Refer to "Bluetooth® Low Energy Protocol Stack User's Manual - 7.4.1.2 Attribute Type" (R01UW0095) for another attribute type UUID.

In "Table 7-1 Sample Custom Service definition of Simple Sample Program", set the service and characteristic attribute type UUID as follows.

Table 7-3 Sample Custom Service Attribute Type UUID

Attribute Type UUID		Definition	Attribute Type UUID Name
Sample Custom Service			
	Primary Service Declaration	RBLE_DECL_PRIMARY_SERVICE	Primary Service
Switch State Characteristic			
	Characteristic Declaration	RBLE_DECL_CHARACTERISTIC	Characteristic
	Characterristic Value	DB_TYPE_128BIT_UUID	128bit Characteristic UUID
	Client Characteristic Configuration Descriptor	RBLE_DESC_CLIENT_CHAR_CONF	Client Characteristic Configuration Descriptor
LED Control Characteristic			
	Characteristic Declaration	RBLE_DECL_CHARACTERISTIC	Characteristic
	Characterristic Value	DB_TYPE_128BIT_UUID	128bit Characteristic UUID

- Sample Custom Service Definition

Set RBLE_DECL_PRIMARY_SERVICE of the primary service as the definition of the service including the Switch State Characteristic and the LED control characteristic. By defining this, 128bit UUID (5BC1B9F7-A1F1-40AF-9043-C43692C18D7A) of Sample Custom Service can be assigned.

- Switch State Characteristic Definition

First, set RBLE_DECL_CHARACTERISTIC to indicate that it is characteristic. As additional information on this definition, set property, attribute handle and 128bit UUID of characteristic in the atts_char128_desc structure of the attribute value storage variable.

Second, set DB_TYPE_128BIT_UUID, which indicates the value to communicate with the client (buffer for communication with client). As additional information on this definition, set 128bit UUID of characteristic, length of 128bit UUID and pointer of communication in the atts_elmt_128 structure of the attribute value storage variable.

Third, set RBLE_DESC_CLIENT_CHAR_CONF, which indicates the client characteristic configuration descriptor (hereinafter called "CCCD"). The Switch State Characteristic is a characteristic that the server sends the state of the switch to the client with Notification. Refer to "4.1 Profile" for CCCD.

- LED Control Characteristic Definition

Like the Switch State Characteristic, do the first and second definitions. The LED Control Characteristic receives LED data sent from the client. CCCD is not defined because data will not be transmitted voluntarily from the server.

(2) **Database atts_desc structure**

The members of the atts_desc structure is shown below. A database is constructed by defining this structure as an array.

Table 7-4 atts_desc structure

atts_desc structure		
	uint16_t type;	Attribute type UUID
	uint8_t maxlength;	Length of attribute value stored variable (The variable used differs depending on the attribute type UUID)
	uint8_t length;	Length of attribute value stored variable Set the same size as maxlength. (The variable used differs depending on the attribute type UUID)
	ke_task_id_t taskid;	Upper 6bit: Profile task ID to which the attribute belongs Lower 10bit: Index to identify the attribute
	uint16_t perm;	Attribute permission
	void *value;	Pointer of attribute value stored variable (The variable used differs depending on the attribute type UUID)

- type

Sets the UUID of the attribute type. The following is the minimum definitions that configure a custom profile. Refer to "Bluetooth® Low Energy Protocol Stack User's Manual - 7.4.1.2 Attribute Type" (R01UW0095) for another attribute type UUID.

Table 7-5 Attribute UUID

Attribute type UUID	Description
RBLE_DECL_PRIMARY_SERVICE	Primary Service (0x2800)
RBLE_DECL_CHARACTERISTIC	Characteristic (0x2803)
DB_TYPE_128BIT_UUID	128bit Characteristic UUID (0xffff)
RBLE_DESC_CLIENT_CHAR_CONF	Client Characteristic Configuration Descriptor (0x2902)

- maxlength

Sets the length of attribute value stored variable. The variable used differs depending on the attribute type UUID. As a setting example, the variable declaration and setting to the member when RBLE_DECL_PRIMARY_SERVICE (primary service) is set to the attribute type UUID are shown below.

Declaration of 128bit UUID array: sams_svc[RBLE_GATT_128BIT_UUID_OCTET]

Setting example of member: sizeof(sams_svc)

- length

Set the same size as maxlength.

- taskid

Use the "TASK_ATTID" macro defined in prf_config.c to combine and set the task ID (TASK_RBLE) and the database index. For the database index, refer to "(7) Database Handle/Database Index" in this section. As a setting example, the following shows an example of setting to a member when RBLE_DECL_PRIMARY_SERVICE (primary service) is set as the attribute type UUID.

Setting example of member: TASK_ATTID(TASK_RBLE, SAMS_IDX_SVC)

- perm

Set the permissions of the attributes. Permissions are set to limit access from GATT clients. Permissions mainly used are shown below. If multiple settings are required, such as reading and writing, use the bitwise OR operator "|".

Table 7-6 Permission

Permission	Description
RBLE_GATT_PERM_RD	Readable from client
RBLE_GATT_PERM_WR	Writable from client
RBLE_GATT_PERM_NI	Able to be notified/indicated

* Refer to "Bluetooth® Low Energy Protocol Stack User's Manual - 7.4.1.2 Attribute Type" (R01UW0095) for another attribute type UUID.

- *value

Set the attribute value storage address (pointer of the attribute value storage variable). Attribute value storage variable differs depending on the attribute type UUID set in type. The following is an example of setting the attribute value storage destination address for each attribute type UUID.

Table 7-7 Pointer of attribute value stored variable

Attribute type UUID	Setting example: Pointer of attribute value stored variable
RBLE_DECL_PRIMARY_SERVICE	(void *)&sams_svc
RBLE_DECL_CHARACTERISTIC	(void *)&switch_state_char
DB_TYPE_128BIT_UUID	(void *)&switch_state_char_val_elmt
RBLE_DESC_CLIENT_CHAR_CONF	(void *)&switch_state_cccd

(3) Attribute Type UUID - Primary Service

If the attribute type UUID is RBLE_DECL_PRIMARY_SERVICE (primary service), set an array of 16 octets to store the 128bit UUID. The 128bit UUID in the program is set to array in little endian.

128bit UUID: 5BC1B9F7-A1F1-40AF-9043-C43692C18D7A

Definition: #define RBLE_SVC_SAMPLE_CUSTOM_SVC
{0x7A,0x8D,0xC1,0x92,0x36,0xC4,0x43,0x90,0xAF,0x40,0xF1,0xA1,0xF7,0xB9,0xC1,0x5B}

Setting to array:

uint8_t sams_svc[RBLE_GATT_128BIT_UUID_OCTET] = RBLE_SVC_SAMPLE_CUSTOM_SVC;

(4) Attribute Type UUID - Characteristic

If the attribute type UUID is RBLE_DECL_CHARACTERISTIC (characteristic), set the atts_char128_desc structure.

Table 7-8 atts_char128_desc structure

atts_char128_desc structure		
	uint8_t prop;	Property
	uint8_t attr_hdl[sizeof(uint16_t)];	Attribute Handle
	uint8_t attr_type[RBLE_GATT_128BIT_UUID_OCTET];	128bit UUID of characteristic

- prop

Define the properties of the characteristic (characteristics: Read, Write, Notify, Indicate etc). For the type of property, refer to "Bluetooth® Low Energy Protocol Stack User's Manual - 7.4 Generic Attribute Profile - Table 7-19 Properties of Characteristics" (R01UW0095). The main properties to use are shown below. If you need more than one setting, use the bitwise OR operator "|" to set it.

Table 7-9 Property

Permission	Description
RBLE_GATT_CHAR_PROP_RD	Permits reading of characteristic values from the client.
RBLE_GATT_CHAR_PROP_WR	Permits characteristic values can be written from the client.
RBLE_GATT_CHAR_PROP_NTF	Permits notification about characteristic values issued from the server to the client.
RBLE_GATT_CHAR_PROP_IND	Permits indication of characteristic values from the server to the client.

- attr_hdl[]

Sets the attribute handle of the element that defined RBLE_DECL_CHARACTERISTIC (characteristic) for the attribute type UUID. Below is an example of setting with the Switch State Characteristic.

Setting example of member: {(uint8_t)(SAMS_HDL_SWITCH_STATE_VAL & 0xff),
(uint8_t)((SAMS_HDL_SWITCH_STATE_VAL >> 8) & 0xff)},

- attr_type[]

Sets the 128bit UUID definition of characteristic. An example of setting with the Switch State Characteristic is shown below.

Setting example of member: RBLE_CHAR_SAMS_SWITCH_STATE

Definition file: sam.h

Definition: #define RBLE_CHAR_SAMS_SWITCH_STATE
{0x7A,0x8D,0xC1,0x92,0x36,0xC4,0x43,0x90,0xAF,0x40,0xF1,0xA1,0x80,0x8D,0xC1,0x5B}

(5) **Attribute Type UUID - 128bit Characteristic UUID**

If the attribute type UUID is DB_TYPE_128BIT_UUID (128bit characteristic UUID), set the atts_elmt_128 structure.

Table 7-10 atts_elmt_128 structure

atts_elmt_128 structure	
uint8_t attr_uuid[RBLE_GATT_128BIT_UUID_OCTET];	128bit UUID of characteristic
uint8_t uuid_len;	Length of 128bit UUID
void *value;	Pointer of communication buffer

- attr_uuid[]

Sets the definition of characteristic UUID. Here is an example of setting with the Switch State Characteristic.

Setting example of member: RBLE_CHAR_SAMS_SWITCH_STATE

128bit UUID definition file: sam.h

Definition: #define RBLE_CHAR_SAMS_SWITCH_STATE
{0x7A,0x8D,0xC1,0x92,0x36,0xC4,0x43,0x90,0xAF,0x40,0xF1,0xA1,0x80,0x8D,0xC1,0x5B}

- uuid_len

Sets the size of the 128bit UUID (16 octets).

- *value

Sets the pointer of communication buffer with client. Here is an example of setting with the Switch State Characteristic.

Setting example of member(array): &switch_state_char_val[0]

Setting example of member(variable): &switch_len_char_val

(6) **Attribute Type UUID - Client Characterristic Configuration Descriptor**

Set when the attribute type UUID is RBLE_DESC_CLIENT_CHAR_CONF (Client characteristic configuration descriptor). It is a variable of type uint16_t.

(7) **Database Handle/Database Index**

The database handle is exposed to the client and it is used by the client to access the service and characteristic of the server.

The database index is used by the BLE protocol stack to identify elements of the database.

Both definitions are defined to correspond one-to-one with the elements of the database. The following shows the database handle and database index corresponding to the Sample Custom Service database definition (only the attribute type UUID is shown). Refer to "4.3.1 Adding Database Handle" and "4.3.2 Adding Database Index" in this manual for a description method in the source code.

Table 7-11 Database handle/Database index

Attribute type UUID	Database handle	Database index
RBLE_DECL_PRIMARY_SERVICE	SAMS_HDL_SVC	SAMS_IDX_SVC
RBLE_DECL_CHARACTERISTIC	SAMS_HDL_SWITCH_STATE_CHAR	SAMS_IDX_SWITCH_STATE_CHAR
DB_TYPE_128BIT_UUID	SAMS_HDL_SWITCH_STATE_VAL	SAMS_IDX_SWITCH_STATE_VAL
RBLE_DESC_CLIENT_CHAR_CONF	SAMS_HDL_SWITCH_STATE_CCCD	SAMS_IDX_SWITCH_STATE_CCCD
RBLE_DECL_CHARACTERISTIC	SAMS_HDL_LED_CONTROL_CHAR	SAMS_IDX_LED_CONTROL_CHAR
DB_TYPE_128BIT_UUID	SAMS_HDL_LED_CONTROL_VAL	SAMS_IDX_LED_CONTROL_VAL

7.1.3 Processing of Sample Custom Service Server Role

This section explains using flow (Figure 7-3 Sample Custom Service Server Role Flow) how rBLE API, rBLE Event, and database handle (hereinafter called "Handle") are used in Sample Custom Service processing. The server process and related source code explained in this flow diagram are as follows.

(It does not explain enable/disable processing of the server.)

[Sample custom service process]

- (A) Flow chart for notifying the press/release state of the switch by Notification. (Figure 7-3: A-1, A-2, A-3)
- (B) Flow chart to turn on/off LED. (Figure 7-3: B-1, B-2)

[Related source code]

- rBLE\src\sample_simple\sam\sams.c
  Source code of the Sample Custom Service server role. It uses rBLE API and rBLE Event of GATT to process characteristic accessed by clients and send data.
- rBLE\src\sample_simple\sam\sams.h
  Define events of the Sample Custom Service and parameters of events.
- rBLE\src\sample_simple\sam\sam.h
  Define UUID of the Sample Custom Service, common macro used by server and client.

In the Sample Custom Service server process flow diagram, characteristic processing is initiated with the following triggers:

- Data was written to the server characteristic from the client, and a rBLE Event occurred in a callback function (sams_gatt_callback()) that notifies the GATT event.
- In the Sample Custom Service API (SAMPLE_Server_Send_Switch_State()) called from the application, the rBLE API of GATT is called and an rBLE Event is generated by a callback function (sams_gatt_callback()) notifying the event of GATT.

First, it will explain "(A) Flow chart for notifying the press/release state of the switch by Notification". Next, it will explain "(B) Flow chart to turn on/off LED".

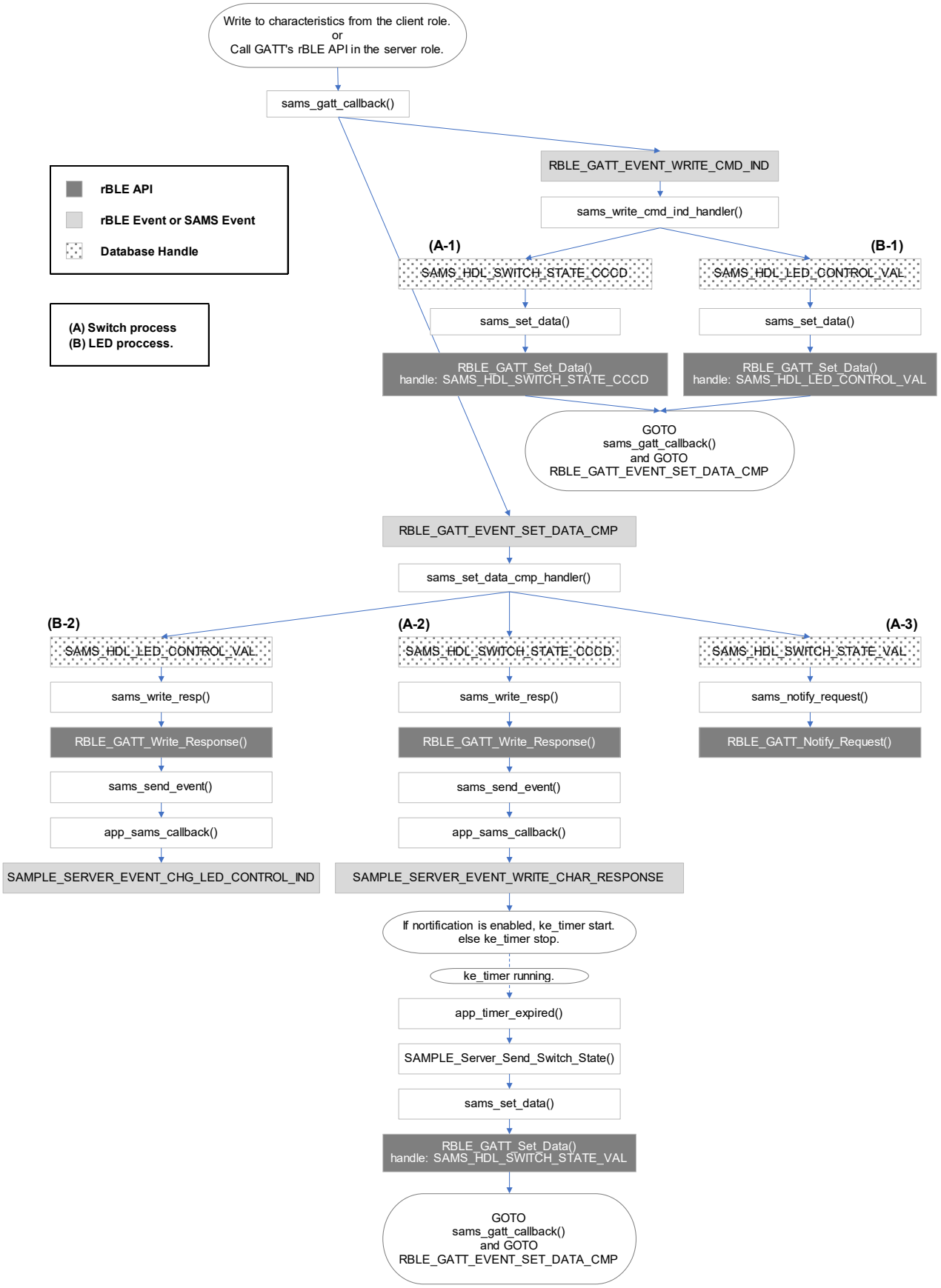


Figure 7-3 Sample Custom Service Server Role Flow

(1) **Flow chart for notifying the press/release state of the switch by Notification**

(A-1)

When Notification enable/disable is written from the client to the Switch State Characteristic - Client Characteristic Configuration Descriptor (hereinafter called "Switch State CCCD") of the server, the RBLE_GATT_EVENT_WRITE_CMD_IND event is occurred in the sams_gatt_callback().

The handle of the event parameter (*) is used to determine which characteristic data was written, and here it can see that it was written to SAMS_HDL_SWITCH_STATE_CCCD. Once written in this characteristic it is necessary to send a response to the client. Set the data to be sent to SAMS_HDL_SWITCH_STATE_CCCD by RBLE_GATT_Set_Data() of the rBLE API. RBLE_GATT_EVENT_SET_DATA_CMP event occur in sams_gatt_callback() due to rBLE API call.

(*) Handle is the number of service and characteristic that the server has. When connecting, the client searches for services and characteristic that the server has. When the client accesses the characteristic of the server, it transmits handle of characteristic and data in the communication packet to the server. The server identifies the handle of the received packet and processes it.

(A-2)

It identifies the currently processed handle SAMS_HDL_SWITCH_STATE_CCCD and executes Switch State CCCD processing. The data set in (A-1) is sent to the client by executing RBLE_GATT_Write_Response(). Notify the peripheral application by SAMPLE_SERVER_EVENT_WRITE_CHAR_RESPONSE event that the write to Switch State CCCD was executed and the response was sent.

In the peripheral application, the SAMPLE_SERVER_EVENT_WRITE_CHAR_RESPONSE event is occurred in app_sams_callback(). Judge enable/disable of Notification in event parameter. If Notification is enabled, call SAMPLE_Server_Send_Switch_State() which transmits the state of the switch at regular intervals using the timer function of the RWKE API. If Notification is disabled, the timer function is stopped and transmission of the switch status is stopped.

In case of Notification enabled, RBLE_GATT_Set_Data() is called from SAMPLE_Server_Send_Switch_State() and data is set in the handle SAMS_HDL_SWITCH_STATE_VAL (Switch State Characteristic - Characteristic Value).

RBLE_GATT_EVENT_SET_DATA_CMP event is occurred in sams_gatt_callback() due to rBLE API call.

(A-3)

It identifies the currently processed handle SAMS_HDL_SWITCH_STATE_VAL and RBLE_GATT_Notify_Request() is called due to send switch state to client by Notification.

Continue sending the switch state until the Notification disable is written from the client to the Switch State CCCD.

(2) **Flow chart to turn on/off LED**

(B-1)

When LED ON/OFF is written from the client to the LED Control Characteristic - Characteristic Value (hereinafter called "LED Control CV") of the server, the RBLE_GATT_EVENT_WRITE_CMD_IND event is occurred in the sams_gatt_callback().

The handle of the event parameter is used to determine which characteristic data was written, and here it can see that it was written to SAMS_HDL_LED_CONTROL_VAL. Once written in this characteristic it is necessary to send a response to the client. Set the data to be sent to SAMS_HDL_LED_CONTROL_VAL by RBLE_GATT_Set_Data() of the rBLE API. RBLE_GATT_EVENT_SET_DATA_CMP event occur in sams_gatt_callback() due to rBLE API call.

(B-2)

It identifies the currently processed handle SAMS_HDL_SWITCH_STATE_CCCD and executes LED Control CV processing. The data set in (B-1) is sent to the client by executing RBLE_GATT_Write_Response(). Notify the peripheral application by SAMPLE_SERVER_EVENT_CHG_LED_CONTROL_IND event that the write to LED Control CV was executed and the response was sent.

In the peripheral application, the SAMPLE_SERVER_EVENT_CHG_LED_CONTROL_IND event is occurred in app_sams_callback(). It controls the LED ON/OFF by the parameter of the event.

7.1.4 Adding Characteristic

Add a characteristic "Table 7-12 Dipswitch State Characteristic" to the Sample Custom Service that sends the dipswitch status on the evaluation board with a read request from the client. For the program code of the existing characteristic (Switch State Characteristic, LED Control Characteristic), see "4.3 How to make GATT Database" in this manual.

The contents of the characteristic and the source code are shown below.

(1) UUID

rBLE\src\sample_simple\sam\sam.h

(2) Database Handle

renesas\src\arch\rl78\db_handle.h

(3) Database Index

renesas\src\arch\rl78\prf_config.h

(4) Characteristic

renesas\src\arch\rl78\prf_config.c

(5) Database

renesas\src\arch\rl78\prf_config.c

Table 7-12 Dipswitch State Characteristic

Type		Value	Permission
Dipswitch State Characteristic			
	Characteristic Declaration (0x2803)	Property: Read UUID: 5BC11b83-A1F1-40AF-9043-C43692C18D7A	Read
	Characteristic Value	2 [octet] value[0]: SW6-1 value[1]: SW6-4	Read

(1) **UUID**

The custom service that Bluetooth SIG does not adopt and the UUID of that characteristic can be freely decided by the user. The UUID of the Dipswitch State Characteristic defines a common 128bit random value with the UUID of the Sample Custom Service. Then, in order to distinguish it from other characteristic, change the values of the 3rd byte and the 4th byte from the top. It is defined in source code with little endian.

UUID to add: 5BC11B83-A1F1-40AF-9043-C43692C18D7A

File: rBLE\src\sample_simple\sam\sam.h

```
        (*) Add bold part to source file

0007:   #define RBLE_SVC_SAMPLE_CUSTOM_SVC
        {0x7A,0x8D,0xC1,0x92,0x36,0xC4,0x43,0x90,0xAF,0x40,0xF1,0xA1,0xF7,0xB9,0xC1,0x5B}
0008:   #define RBLE_CHAR_SAMS_SWITCH_STATE
        {0x7A,0x8D,0xC1,0x92,0x36,0xC4,0x43,0x90,0xAF,0x40,0xF1,0xA1,0x80,0x8D,0xC1,0x5B}
0009:   #define RBLE_CHAR_SAMS_LED_CONTROL
        {0x7A,0x8D,0xC1,0x92,0x36,0xC4,0x43,0x90,0xAF,0x40,0xF1,0xA1,0xEE,0x43,0xC1,0x5B}
0010:   #define RBLE_CHAR_SAMS_DIPSW_STATE
        {0x7A,0x8D,0xC1,0x92,0x36,0xC4,0x43,0x90,0xAF,0x40,0xF1,0xA1,0x83,0x1B,0xC1,0x5B}
```

Figure 7-4 Dipswitch State Characteristic UUID

(2) **Database Handle**

Add the database handle of the Dipswitch State Characteristic. This is a number for the client to access the service and characteristic of the server, and it must be defined one-to-one with the elements of the database. Add the database handle before DB_HDL_MAX in the "db_handle.h" header file.

File: renesas\src\arch\rl78\db_handle.h

```
        (*) Add bold part to source file

        /* Attribute database handles */
        enum
        {
            ...
0404:       /* Simple Sample Custom Service */
0405:       SAMS_HDL_SVC,
0406:       SAMS_HDL_SWITCH_STATE_CHAR,
0407:       SAMS_HDL_SWITCH_STATE_VAL,
0408:       SAMS_HDL_SWITCH_STATE_CCCD,
0409:       SAMS_HDL_LED_CONTROL_CHAR,
0410:       SAMS_HDL_LED_CONTROL_VAL,
0411:       SAMS_HDL_DIPSW_STATE_CHAR,  /* Database handle of characteristic */
0412:       SAMS_HDL_DIPSW_STATE_VAL,   /* Database handle of value */
0413:       #endif /* #ifdef USE_SIMPLE_SAMPLE_PROFILE */

            DB_HDL_MAX
        };
```

Figure 7-5 Dipswitch State Characteristic Database Handle

(3) **Database Index**

Add the database index of the Dipswitch State Characteristic. This is a number for the BLE protocol stack to identify the database of server, and it must be defined one-to-one with the elements of the database. Add the database index in the "prf_config.h" header file.

File: renesas\src\arch\rl78\prf_config.h

```
        (*) Add bold part to source file

        /** Attribute database index */
        enum
        {
            ...
0496:       /* Simple Sample Custom Service */
0497:       SAMS_IDX_SVC,
0498:       SAMS_IDX_SWITCH_STATE_CHAR,
0499:       SAMS_IDX_SWITCH_STATE_VAL,
0500:       SAMS_IDX_SWITCH_STATE_CCCD,
0501:       SAMS_IDX_LED_CONTROL_CHAR,
0502:       SAMS_IDX_LED_CONTROL_VAL,   Append ',' at the end of SAMS_IDX_LED_CONTROL_VAL
0503:       SAMS_IDX_DIPSW_STATE_CHAR,  /* Database index of characteristic */
0504:       SAMS_IDX_DIPSW_STATE_VAL    /* Database index of value */
        };
```

Figure 7-6 Dipswitch State Characteristic Database Index

(4) **Characteristic**

Add the Dipswitch State Characteristic definition structure and value.

For the characteristic structure, set the property, attribute handle, and UUID of characteristic. Since read instructions are issued from the client, set the properties of the read.

Define value of characteristic. The dipswitch on the evaluation board has four slide switches, but since SW6-2 and SW6-3 can not be controlled, it is assumed to be 2bytes of SW6-1 and SW6-4.

Add the characteristic in the "prf_config.h" file.

File: renesas\src\arch\rl78\prf_config.c

```
        (*) Add bold part to source file

1236:   /* Characteristic(sams:dipswitch_state) */
1237:   static const struct atts_char128_desc dipsw_state_char = {
1238:       RBLE_GATT_CHAR_PROP_RD,                                 /* Property */
1239:       {
1240:           (uint8_t)(SAMS_HDL_DIPSW_STATE_VAL & 0xff),/* Attribute handle */
1241:           (uint8_t)((SAMS_HDL_DIPSW_STATE_VAL >> 8) & 0xff)
1242:       },
1243:       RBLE_CHAR_SAMS_DIPSW_STATE          /* 128bit UUID of characteristic */
1244:   };
1245:
1246:   uint8_t dipsw_state_char_val[2] = {0};  /* Value of characteristic */
1247:
1248:   struct atts_elmt_128 dipsw_state_char_val_elmt = {
1249:       RBLE_CHAR_SAMS_DIPSW_STATE,         /* 128bit UUID of characteristic */
1250:       RBLE_GATT_128BIT_UUID_OCTET,        /* Length of UUID */
1251:       &dipsw_state_char_val[0] };         /* Pointer of communication buffer */
1252:   #endif /* #ifdef USE_SIMPLE_SAMPLE_PROFILE */
```

Figure 7-7 Characteristic of Dipswitch State Characteristic

(5) **Database**

Finally, add characteristic and characteristic value to the GATT database. Add these definitions to atts_desc_list_prf[] in the "prf_config.c" file.

File: renesas\src\arch\rl78\prf_config.c

```
        (*) Add bold part to source file

        const struct atts_desc atts_desc_list_prf[] =
        {
            ...

            /*********************************
             * Simple Sample Service         *
             *********************************/
            ...
2151:       /* Characteristic */
2152:       { RBLE_DECL_CHARACTERISTIC,                       /* type */
2153:         sizeof(dipsw_state_char),                       /* maxlength */
2154:         sizeof(dipsw_state_char),                       /* length */
2155:         TASK_ATTID(TASK_RBLE,SAMS_IDX_DIPSW_STATE_CHAR),/* taskid */
2156:         RBLE_GATT_PERM_RD,                              /* perm */
2157:         (void*)&dipsw_state_char },                     /* *value */
2158:
2159:       /* 128bit Characteristic UUID */
2160:       { DB_TYPE_128BIT_UUID,                            /* type */
2161:         sizeof(dipsw_state_char_val),                   /* maxlength */
2162:         sizeof(dipsw_state_char_val),                   /* length */
2163:         TASK_ATTID(TASK_RBLE,SAMS_IDX_DIPSW_STATE_VAL), /* taskid */
2164:         (RBLE_GATT_PERM_RD),                            /* perm */
2165:         (void*)&dipsw_state_char_val_elmt },            /* *value */
2166:  #endif /* #ifdef USE_SIMPLE_SAMPLE_PROFILE */
```

Figure 7-8 Dipswitch State Characteristic Database

7.1.5 Adding Server Profile API and Peripheral Application

It add processing to peripheral application and the Sample Custom Service server role source code so that Dipswitch status can be transmitted by the Dipswitch State Characteristic read from client.

Add source code to the following file.

[Sample Custom Service Server Role Source Code]

- sams.c
- sams.h
- sam.h

[Peripheral Application Source Code]

- rble_sample_app_peripheral.c
- rble_sample_app_peripheral.h
- arch_main.c

The Sample Custom Service Server Role source code: Add server profile API that set state of dipswitch to the Dipswitch State Characteristic.

Peripheral Application source code: When the state of the dipswitch changes, add processing to call the server profile API.

(1) **Adding Server Profile API**

Add server profile API that set state of dipswitch to the Dipswitch State Characteristic. Dipswitch state is sent automatically at read response by read request from client. Since the server does not notify the application that reception of the read request has occurred, it is necessary to set the data to characteristic beforehand. Refer to "Figure 4-13 Read Characteristic" for sequence of read characteristics.

This API specification (Table 7-13) and the source code to be added are shown below.

- sams.c
- sams.h
- sam.h

Table 7-13 Dipswitch State Characteristic Server Role Profile API

RBLE_STATUS SAMPLE_Server_Set_Dipswitch_State (uint16_t conhdl, uint8_t *value)		
- This API set value to the Dipswitch State Characteristic of SAMS. - Value is sent automatically at read response by read request from client.		
Parameters:		
uint16_t	conhdl	Connection Handle
uint8_t	*value	Dipswitch State - Specify the start address of 2 octets array. - value[0]: state of SW6-1 - value[1]: state of SW6-4
Return:		
RBLE_OK	Success	
RBLE_STATUS_ERROR	Status Error	

RBLE_GATT_Set_Data() of the rBLE API is called in the sams_set_data() in SAMPLE_Server_Set_Dipswitch_State(). Set the dipswitch state to the Dipswitch State Characteristic by RBLE_GATT_Set_Data().

File: rBLE\src\sample_simple\sam\sams.c

```
        (*) Add bold part to source file

        static void sams_set_data_cmp_handler(RBLE_GATT_EVENT *event)
        {                                  This function is called from the RBLE_GATT_EVENT_SET_DATA_CMP
            ...                            event occurred by executing RBLE_GATT_Set_Data().

0213:       case SAMS_HDL_DIPSW_STATE_VAL:
0214:           /* do nothing */           The handle specified by the parameter of
0215:           break;                     RBLE_GATT_Set_Data() is added, but no
                                           processing is done.
            default:
```

Figure 7-9 Data Set Event

File: rBLE\src\sample_simple\sam\sams.c

```
        (*) Add the following to the source file

343:    RBLE_STATUS SAMPLE_Server_Set_Dipswitch_State(uint16_t conhdl, uint8_t *value)
344:    {
345:        if (sams_info.conhdl != conhdl) {      This function is a Server Role Profile API
346:            return RBLE_STATUS_ERROR;          called from an application. Set dipswitch
347:        }                                      state to characteristic.
348:
349:        if (SAMS_STATE_CONNECTED != sams_info.state) {
350:            return RBLE_STATUS_ERROR;
351:        }
352:
353:        sams_set_data(SAMS_HDL_DIPSW_STATE_VAL, SAMPLE_DIPSW_STATE_SIZE, value);
354:
355:        return RBLE_OK;
356:    }
```

Figure 7-10 Server Role Profile API

File: rBLE\src\sample_simple\sam\sams.h

```
        (*) Add the following to the source file

0081:   RBLE_STATUS SAMPLE_Server_Set_Dipswitch_State(uint16_t conhdl, uint8_t *value);
```

Figure 7-11 Prototype Definition of Server Role Profile API

File: rBLE\src\sample_simple\sam\sam.h

```
        (*) Add the following to the source file      Store state of SW6-1 and SW6-4 to
                                                      2 octeds array.
0017:   #define SAMPLE_DIPSW_STATE_SIZE   (2)
```

Figure 7-12 Length of Characteristic Value

(2) **Adding Peripheral Application Processing**

Add calling processing of server profile API that get Dipswitch state. Add source code to the following file.

- rble_sample_app_peripheral.c
- rble_sample_app_peripheral.h
- arch_main.c

Switch status change can not be detected because the dipswitch is assigned to a port for which no external interrupt of RL78/G1D occurs. Therefore, after connecting to the client, call the timer task periodically using the RWKE timer function. The timer task sets the state of the dipswitch to the Dipswitch State Characteristic.

The flow until setting the dipswitch state to the Dipswitch State Characteristic is shown below.

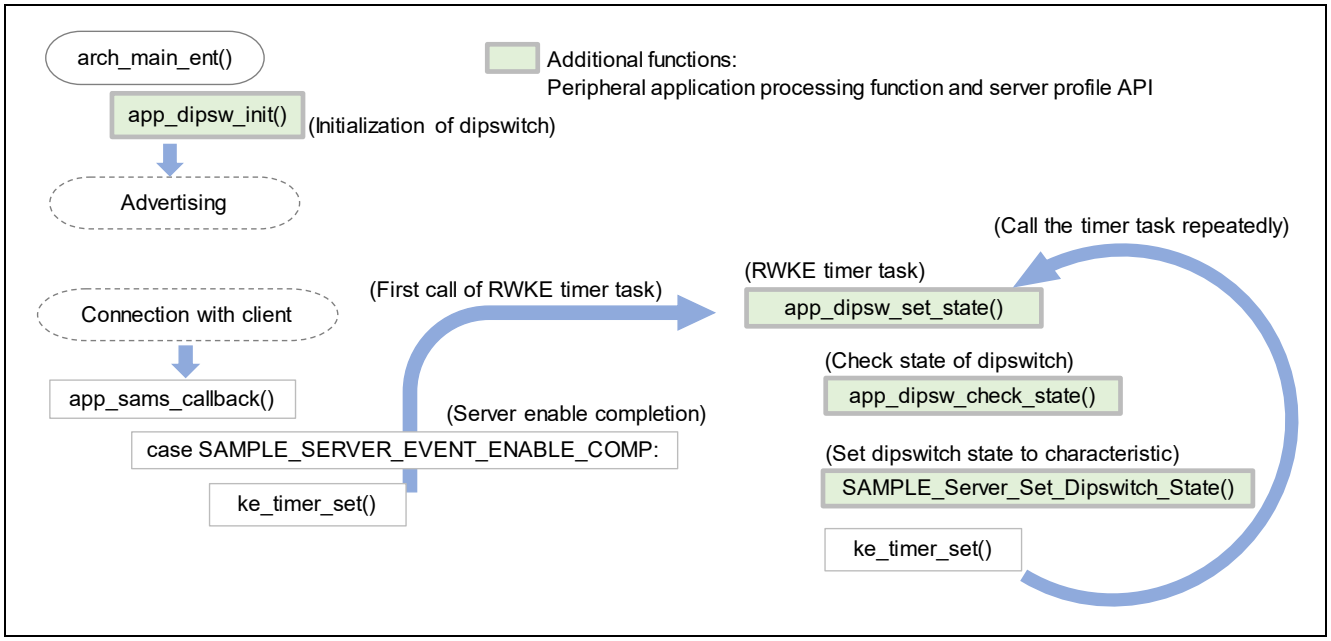


Figure 7-13 Setting Processing of the Dipswitch State

The source code of peripheral application processing is shown below.

File: rBLE\src\sample_simple\rble_sample_app_peripheral.c

```
      (*) Add the following to the source file

0066: void app_dipsw_init(void);
0067: void app_dipsw_check_state(void);
0068: static int_t app_dipsw_set_state(ke_msg_id_t const msgid, void const *param,
0069:                         ke_task_id_t const dest_id, ke_task_id_t const src_id);
0070:
0071: #define DIPSW_VALUE_SIZE    2
0072: static uint8_t dipsw_value[DIPSW_VALUE_SIZE];
```

Figure 7-14 Definition Function and Variable of Peripheral Application Processing

File: rBLE\src\sample_simple\rble_sample_app_peripheral.c

```
      (*) Add bold part to source file

      const struct ke_msg_handler app_connect_handler[] = {
          ...
          { APP_MSG_TIMER_EXPIRED,    (ke_msg_func_t)app_timer_expired },
0098:     { APP_MSG_DIPSW_CHECK,      (ke_msg_func_t)app_dipsw_set_state },
      };
                                      Register the task to be called with the RWKE timer function.
```

Figure 7-15 Message Handler Table of Peripheral Application Processing

File: rBLE\src\sample_simple\rble_sample_app_peripheral.c

(*) Add bold part to source file

```
    ....
0045: #define APP_DIPSW_STATE_CHECK_INTERVAL  (50)

      void app_gap_callback(RBLE_GAP_EVENT *event)
      {
          ....
          case RBLE_GAP_EVENT_DISCONNECT_COMP:
              ke_state_set(APP_TASK_ID, APP_NONCONNECT_STATE);
              app_msg_send(APP_MSG_DISCONNECTED);
0228:         ke_timer_clear(APP_MSG_DIPSW_CHECK, APP_TASK_ID);
              break;


      void app_sams_callback(SAMPLE_SERVER_EVENT *event)
      {
          ....
          case SAMPLE_SERVER_EVENT_ENABLE_COMP:
              app_msg_send(APP_MSG_PROFILE_ENABLED);
0249:         /* Start ke_timer for dipswitch state characteristic */
0250:         ke_timer_set(APP_MSG_DIPSW_CHECK, APP_TASK_ID,
                           APP_DIPSW_STATE_CHECK_INTERVAL);
0251:         break;


0318: void app_dipsw_init(void)
0319: {
0320:     write1_sfr(PU1, 0, 1);   /* SW6-1 */
0321:     write1_sfr(PM1, 0, 1);
0322:     write1_sfr(PU0, 2, 1);   /* SW6-4 */
0323:     write1_sfr(PM0, 2, 1);
0324: }
0325:
0326: void app_dipsw_check_state(void)
0327: {
0328:     /* Read dipswitch state. */
0329:     dipsw_value[0] = read1_sfr(P1, 0);  /* SW6-1 */
0330:     dipsw_value[1] = read1_sfr(P0, 2);  /* SW6-4 */
0331: }
0332:
0333: static int_t app_dipsw_set_state(ke_msg_id_t const msgid, void const *param,
0334:                       ke_task_id_t const dest_id, ke_task_id_t const src_id)
0335: {
0336:     app_dipsw_check_state();
0337:
0338:     /* Set dipswitch state to the dipswitch state characteristic. */
0339:     (void)SAMPLE_Server_Set_Dipswitch_State(app_info.conhdl, &dipsw_value[0]);
0340:     /* Restart ke_timer. */
0341:     ke_timer_set(APP_MSG_DIPSW_CHECK, APP_TASK_ID, APP_DIPSW_STATE_CHECK_INTERVAL);
0342:
0343:     return KE_MSG_CONSUMED;
0344: }
```

Define the interval for checking the dipswitch status with the RWKE timer function.

If disconnection occurs, the RWKE timer function is stopped.

After connecting, when the server enable is completed, the RWKE timer function is started.

After storing the dipswitch status in the array, call the server profile API and set the dipswitch status to characteristic. Then restart the RWKE timer.

Figure 7-16 Peripheral Application Processing Function

File: rBLE\src\sample_simple\rble_sample_app_peripheral.h

```
        (*) Add bold part to source file

        typedef enum {
            ...
0057:       APP_MSG_DIPSW_CHECK,
0058:   } APP_MSG_ID;
```

Define the message ID to be used with the RWKE timer function.

Figure 7-17 Message Handler Table of Peripheral Application Processing

The source code for calling peripheral application processing from the main function (arch_main_ent()) is shown below.

File: renesas\src\arch\rl78\arch_main.c

```
        (*) Add the following to the source file

0084:   extern void app_dipsw_init(void);
```

Figure 7-18 Prototype Definition of Peripheral Application Processing Function

File: renesas\src\arch\rl78\arch_main.c

```
        (*) Add bold part to source file

            // Enable the BLE core
            rwble_enable();

0310:       app_dipsw_init();

0312:       // finally start interrupt handling
0313:       GLOBAL_INT_START();
```

Figure 7-19 Calling Dipswitch Initialization Function

7.1.6 Communication to Smartphone (Dipswitch State Characteristic)

Connect with the GATTBrowser of the smartphone application and check the operation of the added characteristic. The explanation uses an Android smartphone, but it can also confirm with GATTBrowser of iOS with the same operation. The Simple Sample Program project file is stored in the following folder, please build and write it to the evaluation board.

- BLE_Software_Ver_X_XX/RL78_G1D/Project_Source/renesas/tool/project_simple/

The Simple Sample Program automatically starts advertising when it is executed and becomes connectable. Please connect to GATTBrowser according to the following procedure and check the operation of characteristic.

1. Turn on power to the evaluation board and execute the Simple Sample Program.
2. Execute the GATTBrowser on the android smartphone.
3. (Figure 1 - Arrow (1)) It can find the evaluation board on which the Simple Sample Program works. And, tap the connection icon to connect to the evaluation board.
4. (Figure 2 - Arrow (2)) Slide the connected screen upward to display the lowest characteristic.
5. (Figure 3 - Arrow (3)) Tap the added characteristic [UUID:5bc11b83-40af-9043-c43692c18d7a] to display the characteristic screen.
6. (Figure 4 - Arrow (4)) Tap the "Read" button to display the status of the dipswitch. Please operate SW6-1, SW6-4 on the evaluation board and confirm that the state of the dipswitch changes.

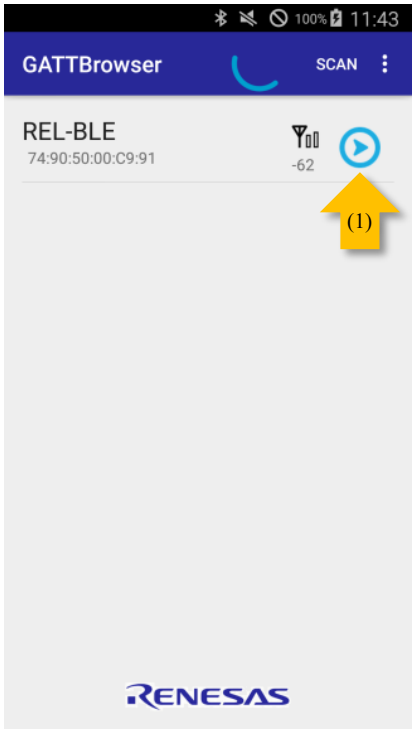


Figure 1

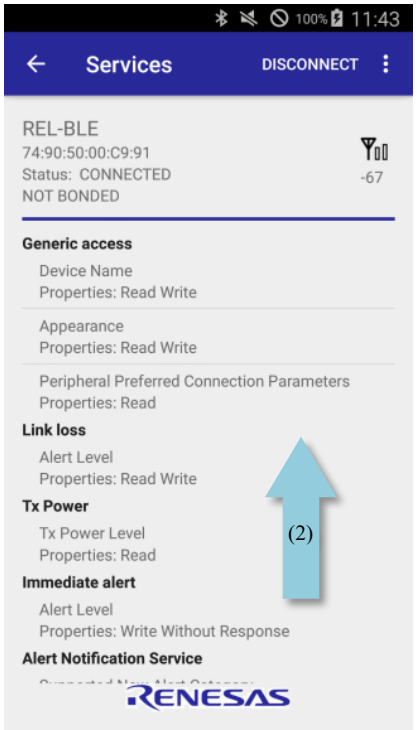


Figure 2

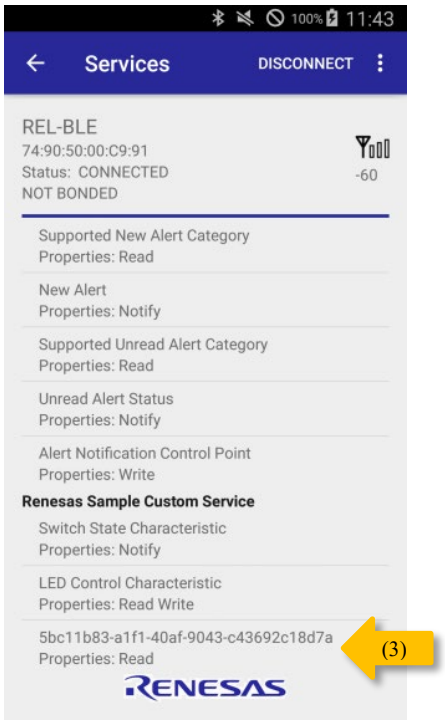


Figure 3

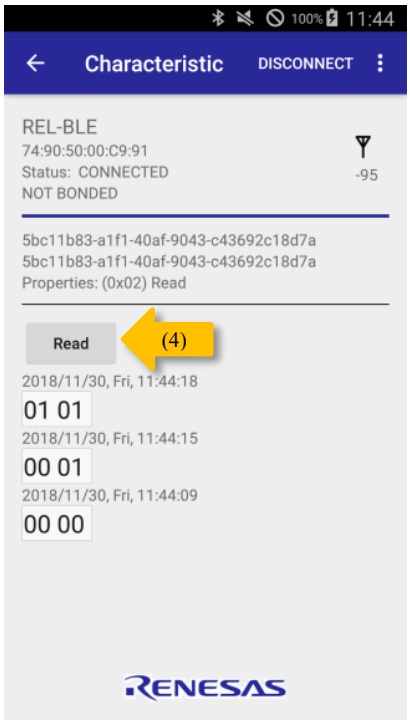


Figure 4

7.1.7 Customize from Notification to Indication

Notification does not respond from the client even if data is sent from the server to the client, but a response called Confirmation is returned from the client in Indication. By receiving the Confirmation, the server can know that the data has arrived at the client.

This section explains how to change Notification of the Switch State Characteristic to Indication. The changes are shown below.

(1) Modify of Switch State Characteristic Database

- Change database from Notification to Indication.

(2) Client Characteristic Configuration Descriptor Processing

- Change the judgment of the value set in CCCD from Notification to Indication.

(3) Data Transmission Processing

- Change the data transmission API from Notification to Indication.

(4) Addition of Confirmation Event and Notify Application

- Add event processing occurring in the Confirmation from the client and Notification processing to the application.

Refer to "Table 4-1 Setting Value of Client Characteristic Configuration Descriptor", "Figure 4-14 Notification Characteristic" and "Figure 4-15 Indication Characteristic" for setting value of CCCD and communication difference of Notification and Indication.

First, it will explain how the Switch State Characteristic is processed by changing to Indication, using "Figure 7-20 Indication Flow of Switch State Characteristic".

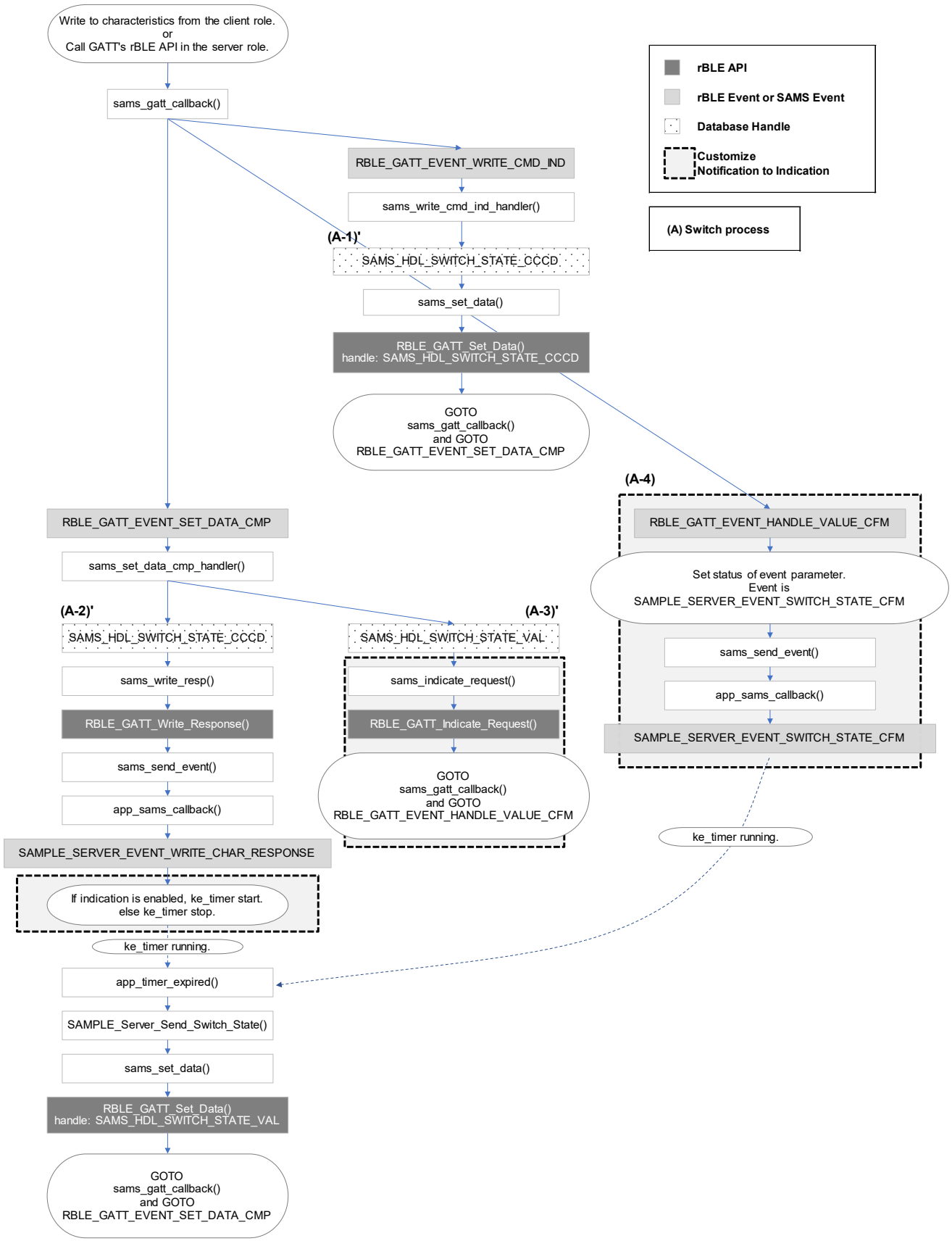


Figure 7-20 Indication Flow of Switch State Characteristic

(*) Underline part is processing to change from Notification to Indication.

(A-1)'

When Indication enable/disable is written from the client to the Switch State CCCD of the server, the RBLE_GATT_EVENT_WRITE_CMD_IND event is occurred in the sams_gatt_callback().

The handle of the event parameter is used to determine which characteristic data was written, and here it can see that it was written to SAMS_HDL_SWITCH_STATE_CCCD. Once written in this characteristic it is necessary to send a response to the client. Set the data to be sent to SAMS_HDL_SWITCH_STATE_CCCD by RBLE_GATT_Set_Data() of the rBLE API. RBLE_GATT_EVENT_SET_DATA_CMP event occur in sams_gatt_callback() due to rBLE API call.

(A-2)'

It identifies the currently processed handle SAMS_HDL_SWITCH_STATE_CCCD and executes Switch State CCCD processing. The data set in (A-1)' is sent to the client by executing RBLE_GATT_Write_Response(). Notify the peripheral application by SAMPLE_SERVER_EVENT_WRITE_CHAR_RESPONSE event that the write to Switch State CCCD was executed and the response was sent.

In the peripheral application, the SAMPLE_SERVER_EVENT_WRITE_CHAR_RESPONSE event is occurred in app_sams_callback(). Judge enable/disable of Indication in event parameter. If Indication is enabled, call SAMPLE_Server_Send_Switch_State() which transmits the state of the switch at regular intervals using the timer function of the RWKE API. If Indication is disabled, the timer function is stopped and transmission of the switch status is stopped.

In case of Indication enabled, RBLE_GATT_Set_Data() is called from SAMPLE_Server_Send_Switch_State() and data is set in the handle SAMS_HDL_SWITCH_STATE_VAL (Switch State Characteristic - Characteristic Value).

RBLE_GATT_EVENT_SET_DATA_CMP event is occurred in sams_gatt_callback() due to rBLE API call.

(A-3)'

It identifies the currently processed handle SAMS_HDL_SWITCH_STATE_VAL and RBLE_GATT_Indicate_Request() is called due to send switch state to client by Indication.

When client receives the switch status, it sends a Confirmation notifying that it received it. The server occurs RBLE_GATT_EVENT_HANDLE_VALUE_CFM event in sams_gatt_callback() due to receipt of Confirmation.

(A-4)

Notify the peripheral application that the Confirmation has been received from the client. Notification also passes the event parameter at the same time as the SAMPLE_SERVER_EVENT_SWITCH_STATE_CFM event. In the peripheral application, the RBLE_GATT_EVENT_HANDLE_VALUE_CFM event occurs in app_sams_callback(). Continue sending the switch state from the client until the Indication disable is written to the Switch State CCCD.

(1) **Modify of Switch State Characteristic Database**

Change property definition from Notification to Indication.

File: renesas\src\arch\rl78\prf_config.c

```
(*) Modify bold part

    ....

static const struct atts_char128_desc switch_state_char = {
1210:    RBLE_GATT_CHAR_PROP_IND,
    {(uint8_t)(SAMS_HDL_SWITCH_STATE_VAL & 0xff),
     (uint8_t)((SAMS_HDL_SWITCH_STATE_VAL >> 8) & 0xff)},
    RBLE_CHAR_SAMS_SWITCH_STATE};
```

Figure 7-21 Modify of Switch State Characteristic Database

(2) **Client Characteristic Configuration Descriptor Processing**

If enable/disable of Indication is set for CCCD from the client, SAMPLE_SERVER_EVENT_WRITE_CHAR_RESPONSE occurs in the peripheral application. Change the definition from Notification to Indication so that enable/disable of Indication can be determined.

File: rBLE\src\sample_simple\rble_sample_app_peripheral.c

```
(*) Modify bold part

void app_sams_callback(SAMPLE_SERVER_EVENT *event)
{
    ....
    case SAMPLE_SERVER_EVENT_WRITE_CHAR_RESPONSE:
        /* Start notification timer if switch_state characteristic cccd is
          set correctly. */
0255:   if (event->param.write_char_resp.value & RBLE_PRF_START_IND) {
            ke_timer_set(APP_MSG_TIMER_EXPIRED, APP_TASK_ID,
                         APP_SWITCH_STATE_CHECK_INTERVAL);
        } else {
            ke_timer_clear(APP_MSG_TIMER_EXPIRED, APP_TASK_ID);
        }
        break;
```

Figure 7-22 Client Characteristic Configuration Descriptor Processing

(3) **Data Transmission Processing**

Change it to use Indication API in data transmission.

File: rBLE\src\sample_simple\sam\sams.c

```
        (*) Add bold part to source file, or modify

            ....
0077:   static void sams_indicate_request(void);        Add


0163:   static void sams_indicate_request(void)         Add
0164:   {
0165:       RBLE_GATT_INDICATE_REQ ind;
0166:
0167:       ind.conhdl  = sams_info.conhdl;
0168:       ind.charhdl = sams_info.hdl;
0169:
0170:       (void)RBLE_GATT_Indicate_Request(&ind);
0171:   }


        static void sams_set_data_cmp_handler(RBLE_GATT_EVENT *event)
        {
            ....
            case SAMS_HDL_SWITCH_STATE_VAL:
0221:           //sams_notify_request();          Modify and Add
0222:           sams_indicate_request();
                break;


        RBLE_STATUS SAMPLE_Server_Send_Switch_State(uint16_t conhdl, uint8_t value)
        {                                          Modify
            ....
0342:       if ((sams_info.param.switch_state_cccd & RBLE_PRF_START_IND)
                != RBLE_PRF_START_IND) {
```

Figure 7-23 Data Transmission Processing

(4) **Addition of Confirmation Event and Notify Application**

Add processing of event (RBLE_GATT_EVENT_HANDLE_VALUE_CFM) which occurs by receiving Confirmation from client. Also add an event (SAMPLE_SERVER_EVENT_SWITCH_STATE_CFM) notifying the peripheral application that Confirmation has been received.

File: rBLE\src\sample_simple\sam\sams.h

```
        (*) Add bold part to source file
            ....
        typedef enum {
            SAMPLE_SERVER_EVENT_ENABLE_COMP = 0,
            SAMPLE_SERVER_EVENT_DISABLE_COMP,
            SAMPLE_SERVER_EVENT_CHG_LED_CONTROL_IND,
            SAMPLE_SERVER_EVENT_WRITE_CHAR_RESPONSE,
0049:       SAMPLE_SERVER_EVENT_SWITCH_STATE_CFM,
        } SAMPLE_SERVER_EVENT_TYPE;
```

Figure 7-24 Confirmation Reception Notification Event

File: rBLE\src\sample_simple\sam\sams.c

```
        (*) Add bold part to source file
            ....
        static void sams_send_event(SAMPLE_SERVER_EVENT_TYPE type)
        {
            ....
0114:       case SAMPLE_SERVER_EVENT_SWITCH_STATE_CFM:
0115:           event.status = sams_info.status;
0116:           break;

            default:

        static void sams_gatt_callback(RBLE_GATT_EVENT *event)
        {
            ....
0275:       case RBLE_GATT_EVENT_HANDLE_VALUE_CFM:
0276:           sams_info.status = event->param.handle_value_cfm.status;
0277:           sams_send_event(SAMPLE_SERVER_EVENT_SWITCH_STATE_CFM);
0278:           break;

            default:
```

Notify the peripheral application of Confirmation reception.

This event will occur on receipt of Confirmation

Figure 7-25 Confirmation Reception Processing

File: rBLE\src\sample_simple\rble_sample_app_peripheral.c

```
        (*) Add bold part to source file

        void app_sams_callback(SAMPLE_SERVER_EVENT *event)
        {
            ....
0262:       case SAMPLE_SERVER_EVENT_SWITCH_STATE_CFM:
0263:           /* do nothing */
0264:           break;
```

Peripheral application event notified of Confirmation reception.

Figure 7-26 Notify Confirmation Reception to Peripheral Application

7.1.8 Communication to Smartphone (Indication of Switch State Characteristic)

Use GATTBrowser to check the difference between the program in the initial state and the program changed with "7.1.7 Customize from Notification to Indication". As with "7.1.6 Communication to Smartphone (Dipswitch State Characteristic)", use GATTBrowser on Android smartphone.

The Simple Sample Program project file is stored in the following folder, please build and write it to the evaluation board.

- BLE_Software_Ver_X_XX\RL78_G1D\Project_Source\renesas\tool\project_simple\

The Simple Sample Program automatically starts advertising when it is executed and becomes connectable. Please connect to GATTBrowser according to the following procedure and check the operation of characteristic.

[Notification (Not customized Simple Sample Program)]

1. Connect GATTBrowser and the Simple Sample Program of Notification.
2. (Figure 1) Confirm that "Properties" of "Switch State Characteristic" is "Notify" on "Service" screen.
3. (Figure 2) Confirm that the button is "Notification" on "Characteristic" screen. When tap the button, the state of SW4 is transmitted at the Notification from the evaluation board. Confirm that "value" of "Descriptors" is "01 00" (Notification enable). (The value is little-endian, so the actual value is 0x0001)

[Indication (Customized Simple Sample Program)]

4. Connect GATTBrowser and the Simple Sample Program of Indication.
5. (Figure 3) Confirm that "Properties" of "Switch State Characteristic" is "Indicate" on "Service" screen.
6. (Figure 4) Confirm that the button is "Indication" on "Characteristic" screen. When tap the button, the state of SW4 is transmitted at the Indication from the evaluation board. Confirm that "value" of "Descriptors" is "02 00" (Indication enable). (The value is little-endian, so the actual value is 0x0002)

Notification	Indication

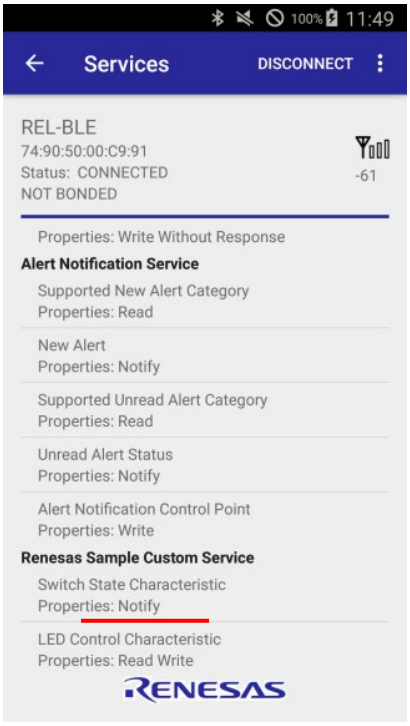


Figure 1

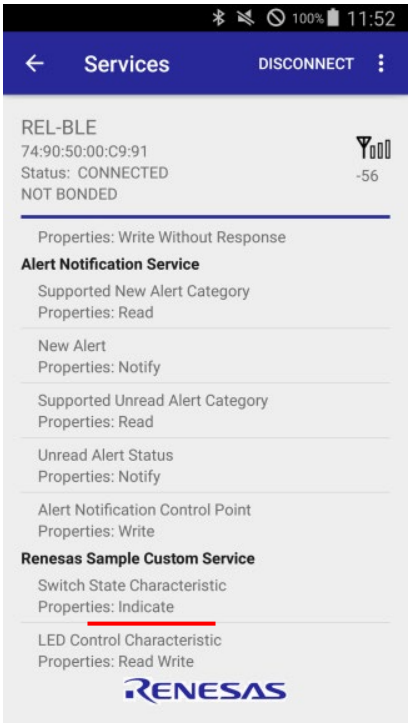


Figure 3

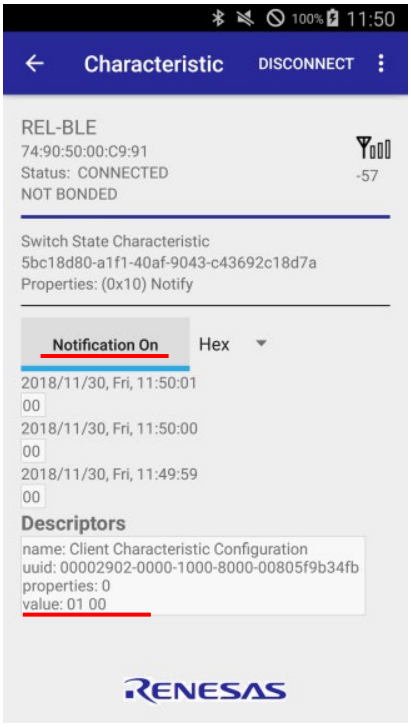


Figure 2

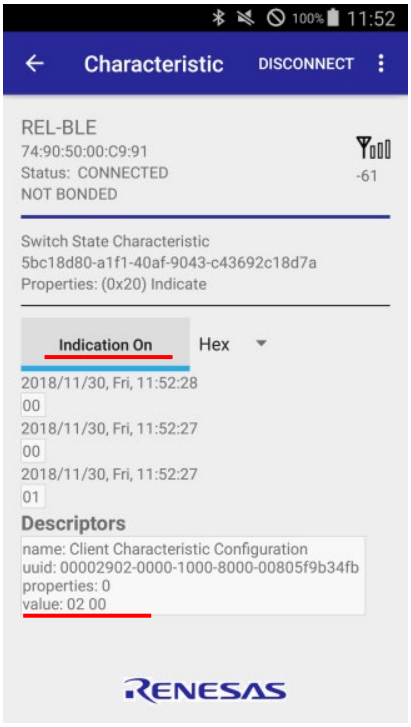


Figure 4

Website and Support

Renesas Electronics Website

http://www.renesas.com/

Inquiries

http://www.renesas.com/contact/

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Apr 28, 2015	-	First edition issued.
1.10	Sep 23, 2016	-	Apply to BLE protocol stack v1.20 Add Chapter 5,6,7, and 8
1.20	Nov 28, 2017	-	Chapter 2 and 5 are added, and composition is changed as shown below. Chapter 1 BLE software Chapter 2 RWKE Chapter 3 BLE Protocol Stack Chapter 4 Profile Chapter 5 How application operates Chapter 6 Development Tips
		P.5	Chapter 2 "RWKE" is added. Section 2.1 RWKE Section 2.2 Executing RWKE Section 2.3 RWKE API Section 2.4 Use Case Section 2.5 Implementing Application Section 2.6 Notes
		P.45	Chapter 5 "How application operates" is added Section 5.1 Simple Sample Program Section 5.2 Start of BLE Software Section 5.3 Initializing BLE Protocol Stack Section 5.4 Starting Broadcast and Establishing Connection Section 5.5 Enabling Custom Profile Section 5.6 Data Communication of Custom Profile Section 5.7 Disabling Custom Profile and Restarting Broadcast
1.30	Dec 7, 2018	P.77	Chapter 7 "Appendix" is added Section 7.1 How to Add Characteristic to Custom Profile

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. Handling of Unused Pins

   Handle unused pins in accordance with the directions given under Handling of Unused Pins in the manual.

   ¾ 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 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. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

   The state of the product is undefined at the moment when power is supplied.

   ¾ The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the moment 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 moment 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 moment when power is supplied until the power reaches the level at which resetting has been specified.

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

   ¾ The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.

   ¾ 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. Moreover, 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.

5. Differences between Products

   Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

   ¾ The characteristics of Microprocessing unit or Microcontroller unit products in the same group but having a different part number may differ in terms of the 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 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

RENESAS

SALES OFFICES

Renesas Electronics Corporation

http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
1001 Murphy Ranch Road, Milpitas, CA 95035, U.S.A.
Tel: +1-408-432-8888, Fax: +1-408-434-5351

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-651-700

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709 Quantum Plaza, No.27 ZhichunLu, Haidian District, Beijing, 100191 P. R. China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, 200333 P. R. China
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HAL 2nd Stage, Indiranagar, Bangalore 560 038, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
17F, KAMCO Yangjae Tower, 262, Gangnam-daero, Gangnam-gu, Seoul, 06265 Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5338

Colophon 7.1