Bluetooth® low energy プロトコルスタック アプリケーション作成ガイド

R01AN2768JJ0130
Rev.1.30
2018.11.19

要旨

このマニュアルは、Bluetooth low energy ソフトウェア（以降、BLE ソフトウェア）を使用したアプリケーションの作成方法や、RWKE（Renesas Wireless Kernel Extension）と BLE プロトコルスタックの概要について説明します。

アプリケーションを Modem 構成で作成する場合、BLE プロトコルスタックの機能を利用するための rBLE API について理解する必要があります。

アプリケーションを Embedded 構成で作成する場合、BLE プロトコルスタックの機能を利用するための rBLE API に加え、簡易 OS である RWKE の機能を利用するための RWKE API についても理解する必要があります。

適用

本ガイドの記載内容は、BLE ソフトウェア(RTM5F11A00NBLE0F10RZ)の Version1.20 以降に適用します。

動作確認デバイス

RL78/G1D

目次

1. BLE ソフトウェア

BLE ソフトウェアには、以下の BLE プロトコルスタックと RWKE が含まれます。

BLE プロトコルスタック

BLE プロトコルスタックは、RL78/G1D の RF 部を管理し、Bluetooth low energy 無線通信に必要な機能をアプリケーションに提供するソフトウェアスタックです。

BLE プロトコルスタックは **rBLE API** を提供します。Embedded 構成と Modem 構成のアプリケーションは rBLE API を使用することで、BLE プロトコルスタックの Generic Access Profile(GAP)、Security Manager(SM)、Vender Specific(VS)、そして各種プロファイルにアクセスすることができます。

RWKE (Renesas Wireless Kernel Extension)

RWKE は、アプリケーションと BLE プロトコルスタックの各動作をスケジューリングするための、ノンプリエンプティブルマルチタスク方式を採用した簡易 OS です。

RWKE は **RWKE API** を提供します。Embedded 構成のアプリケーションは RWKE API を使用することで、RWKE の機能を利用し、アプリケーションは自由度の高いシーケンスを実現することができます。

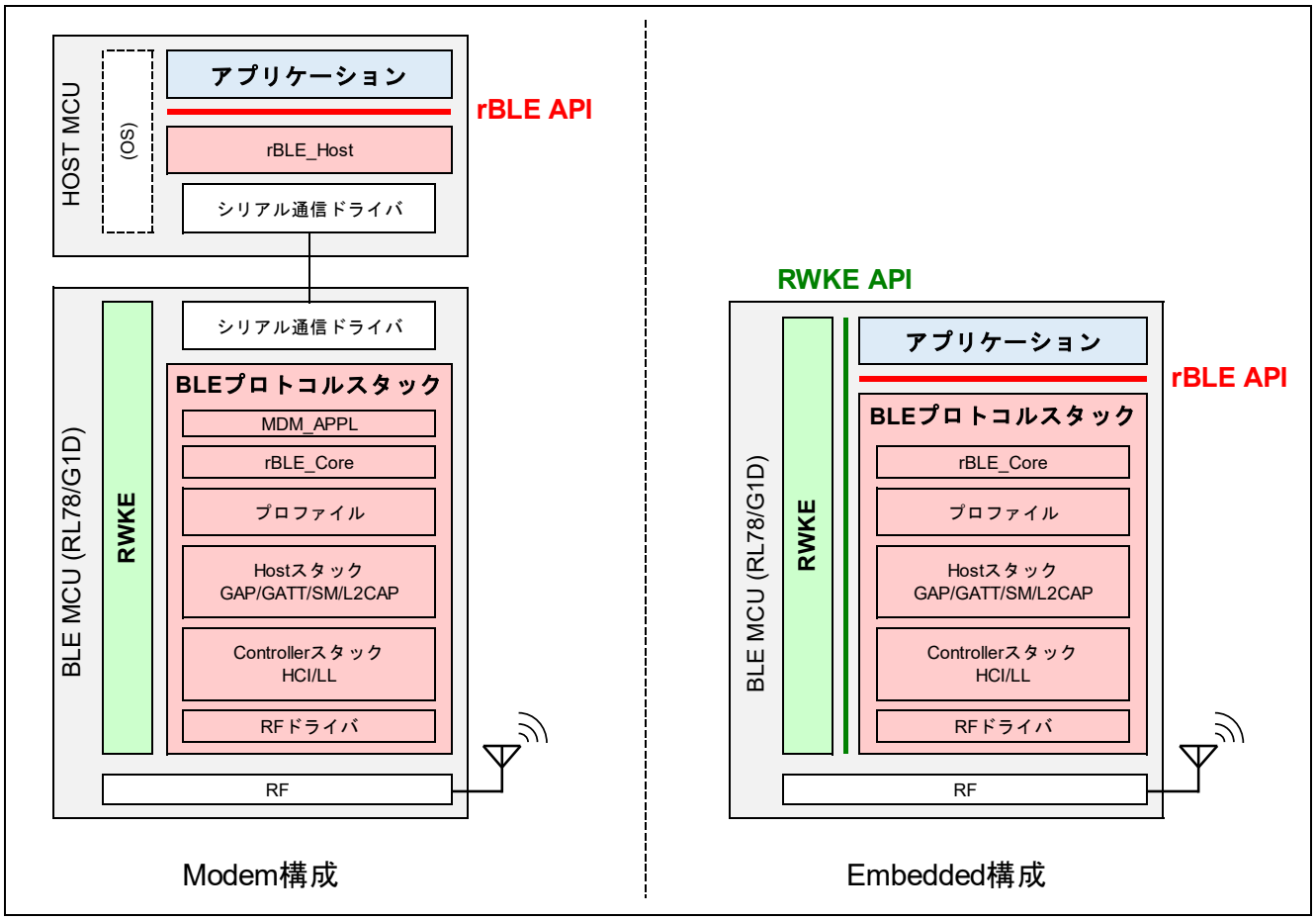


図 1-1　RWKE と BLE プロトコルスタック

2. RWKE

2.1 RWKE とは

RWKE は、アプリケーションと BLE プロトコルスタックの各動作をスケジューリングするための、疑似マルチタスク方式（ノンプリエンプティブルマルチタスク方式）の簡易オペレーティングシステムです。

アプリケーションは RWKE の下記の機能を利用できます。また RWKE は下記の機能を利用するための RWKE API を提供します。

- イベント機能　：セットされたイベントに対応するイベント処理を優先度順に実行
- メッセージ機能　：送信されたメッセージに対応するメッセージ処理を送信順に実行
- タスクステート機能　：各タスクのステートに応じて実行するメッセージ処理を切替
- タイマ機能　：指定時間の経過後にメッセージを送信
- メモリ機能　：指定されたサイズのメモリ領域の確保と解放

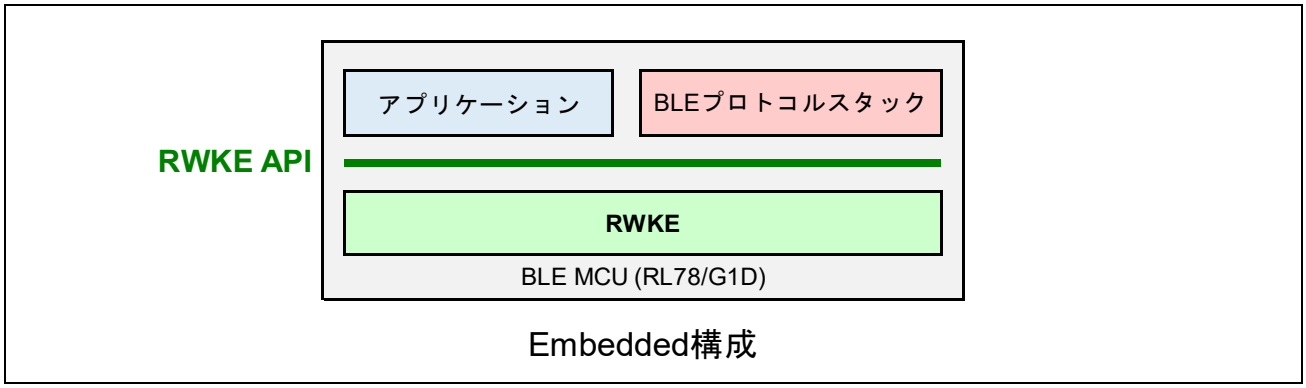


図 2-1　RWKE と RWKE API

各機能を組み合わせることにより、アプリケーションは自由度の高いシーケンスを実現できます。

下記はイベント機能、メッセージ機能、タイマ機能によるアプリケーションの動作シーケンス例です。

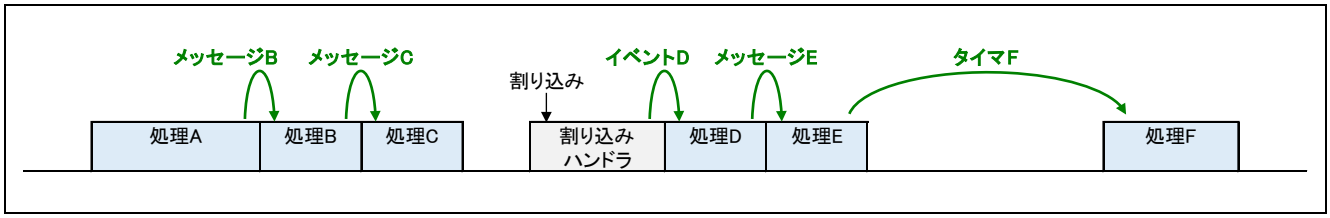


図 2-2　アプリケーションのシーケンス例

BLE プロトコルスタックもアプリケーションと同様に RWKE の各機能を利用することで、BLE 無線通信のための動作を実現します。

下記は BLE プロトコルスタックとアプリケーションそれぞれの動作シーケンス例です。

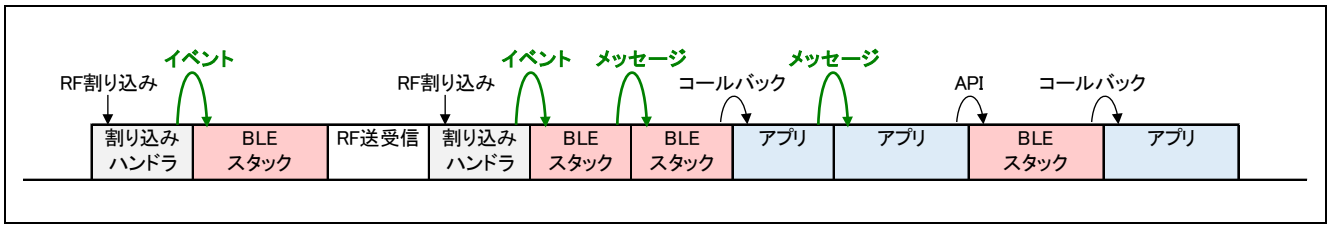


図 2-3　BLE プロトコルスタックとアプリケーションのシーケンス例

イベント処理の優先度を以下に示します。

RWKE はイベント処理を優先度に従って実行します。BLE プロトコルスタックの RF 送受信に関連する処理が最も高い優先度となります。また**メッセージ処理**、**タイマ処理**も 1 つのイベント処理として管理されます。

表 2-1　RWKE イベントの優先度

イベント優先度	イベント ID	イベント処理
0　(高)	KE_EVT_EVENT_START	RF 送受信開始処理
1	KE_EVT_RX	RF 受信処理
2	KE_EVT_EVENT_END	RF 送受信終了処理
3	KE_EVT_HCI_TX_DONE	UART 送信完了処理(Modem 構成のみ)
4	**KE_EVT_USR_0**	**アプリケーションのイベント処理を実装可能**
5	**KE_EVT_USR_1**	**アプリケーションのイベント処理を実装可能**
6	KE_EVT_KE_TIMER	**RWKE のタイマチェック処理**
7	KE_EVT_KE_MESSAGE	**RWKE のメッセージ処理**
8	KE_EVT_CRYPT	RF 暗号化完了処理
9	KE_EVT_HCI_RX_DONE	UART 受信完了処理(Modem 構成のみ)
10	**KE_EVT_USR_2**	**アプリケーションのイベント処理を実装可能**
11	**KE_EVT_USR_3**	**アプリケーションのイベント処理を実装可能**
12	リザーブ	リザーブ
:	:	:
31　(低)	リザーブ	リザーブ

ここで RWKE は、ノンプリエンプティブ方式で各機能の処理を実行します。つまり RWKE はアプリケーションもしくは BLE プロトコルスタックのある処理関数を実行すると、より優先度の高い機能が要求された場合においても、実行中の処理関数を中断しません。実行中の関数が完了後、より優先度の高い機能を実行します。

例として、メッセージ A とメッセージ B が発行され、メッセージ処理 A を実行中であるとします。メッセージ処理は優先度 7 です。実行中のメッセージ処理 A が優先度 4 のイベント C を通知した場合、メッセージ処理 A の完了後、優先度 4 のイベント処理 C が先に実行され、イベント処理 C の完了後に、次のメッセージ処理 B が実行されます。

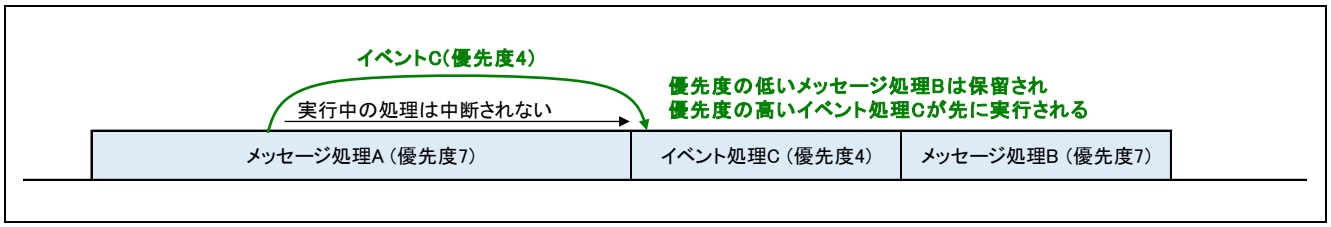


図 2-4　メッセージとイベントの実行順序

また RWKE はソフトウェアの各処理のスケジューリングを目的として実装されており、一般的なオペレーティングシステムが提供する下記のような機能はありません。

- ハードウェアリソースの管理　：周辺機能ドライバを別途実装頂く必要があります
- 割り込みの管理　：割り込みハンドラを別途実装頂く必要があります
- 仮想メモリ空間の提供　：物理メモリ空間の配置とサイズを十分考慮頂く必要があります

2.2 RWKE の実行

RWKE は、BLE ソフトウェアのメインループである以下の **rwble_schedule 関数**により実行されます。

本関数が RWKE の各機能を続けて実行し、実行すべきイベント、メッセージが空となると rwble_schedule 関数が終了します。また実行すべきイベント、メッセージがセットされるとメインループはまた rwble_schedule 関数を実行します。

ファイル：renesas/src/arch/rl78/arch_main.c

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

    // 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

RWKE が提供する機能と RWKE API の概要について示します。

RWKE API 仕様の詳細は、下記を参照してください。

Bluetooth low energy プロトコルスタック API リファレンスマニュアル 基本編 (R01UW088)
https://www.renesas.com/document/man/bluetooth-low-energy-protocol-stack-api-reference-manual-basics

- 9 章「RWKE」

2.3.1 イベント機能

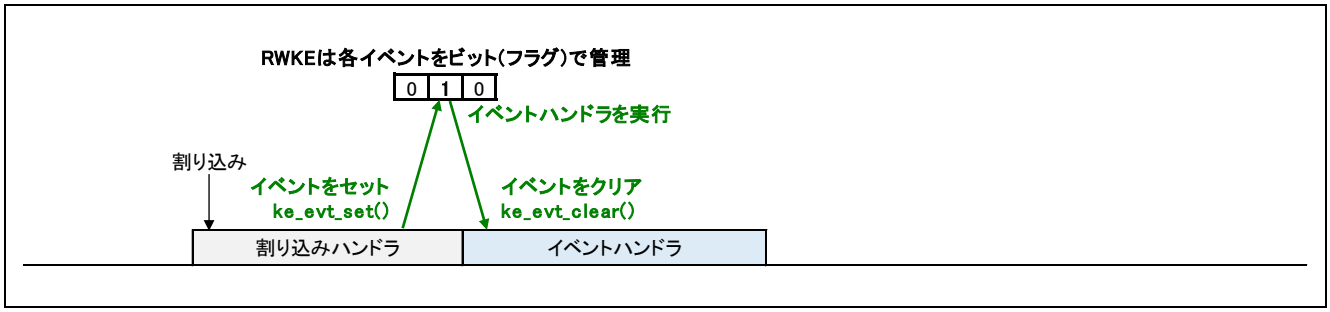
RWKE が提供するイベント機能の概要を以下に示します。

- イベント機能は、イベントのセットを契機としてイベント処理を実行するための仕組み
- RWKE はイベントをセット、クリアするための RWKE API を提供する
- 各イベントと各イベント処理関数（イベントハンドラ）は、イベントハンドラテーブルによって紐付けられる

イベントハンドラテーブル

イベント	イベントハンドラ
イベントA	イベントハンドラA
イベントB	イベントハンドラB

- RWKE API によってイベントがセットされると、RWKE はイベントハンドラを実行
- ユースケースは、割り込みハンドラからイベントをセットし、シーケンスの実行契機とする



後述するメッセージ機能と比較し、イベント機能は以下の主な特徴があります。

- イベントは RWKE によってあらかじめ定義されていて、各イベントは異なる優先度を持つ
- アプリケーションはいずれのソフトウェアからも使用されないイベントを独占して使用する
- 通知できるのはイベント ID のみ、通知元、通知先、パラメータは通知できない
- イベントがセットされる際、メッセージ機能のような動的なメモリ領域の確保はない
- イベントはビットで管理され、同時に同一イベントが何度セットされてもイベント処理の実行は 1 回
- 複数のイベントが設定された場合、RWKE は各イベントが持つ優先度順にイベント処理を実行

イベント機能の RWKE API を以下に示します。

表 2-2 イベント機能の RWKE API

RWKE API	概要
ke_evt_get	イベントの設定状態を取得
ke_evt_set	イベントをセット
ke_evt_clear	イベントをクリア

BLE ソフトウェアに含まれる簡易サンプルプログラムを例として、イベント機能を利用するコード例を以下に示します。簡易サンプルプログラムの詳細は、本書の5.1節を参照してください。

1. ユーザ 0 イベント ID(KE_EVT_USR_0)のイベント設定ビット(**KE_EVT_USR_0_BIT**)を定義します。

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

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

2. ユーザ 0 イベントのための新規イベントハンドラ(**app_evt_usr0**)を宣言します。

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

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

3. ユーザ 0 イベントのための新規イベントハンドラを実装します。

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

```
void codeptr app_evt_usr0(void)
{
    ke_evt_clear(KE_EVT_USR_0_BIT);

    ...
}
```

4. RWKE のイベントハンドラテーブル(**ke_evt_hdlr_ent**)に、イベントハンドラを登録します。

   ファイル：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. アプリケーションの任意の処理内で、イベントを設定する **ke_evt_set 関数**をコールします。

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

以上でコード例は完了です。手順5でイベントを設定すると、手順3で実装したイベントハンドラが実行されます。

2.3.2　メッセージ機能

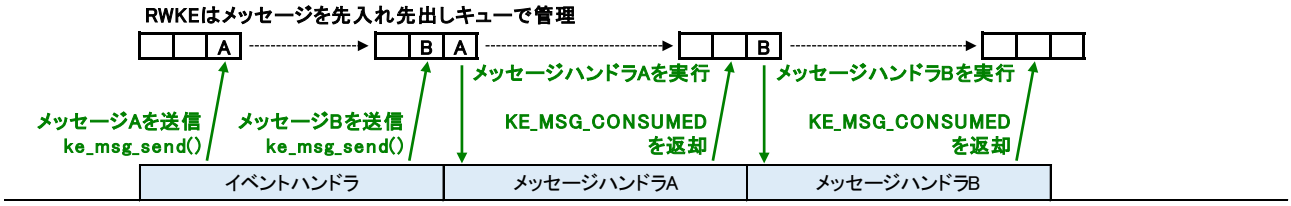
RWKE が提供するメッセージ機能の概要を以下に示します。

- メッセージ機能は、メッセージ送信を契機としてメッセージ処理を実行するための仕組み
- RWKE はメッセージを作成、送信するための RWKE API を提供する
- 各メッセージと各メッセージ処理関数（メッセージハンドラ）は、メッセージハンドラテーブルによって紐付けられる

メッセージハンドラテーブル

メッセージ	メッセージハンドラ
メッセージA	メッセージハンドラA
メッセージB	メッセージハンドラB

- RWKE API によってメッセージが送信されると、RWKE はメッセージハンドラを実行
- ユースケースは、アプリケーションの処理から処理へメッセージを送信して、シーケンスを形成する



前述のイベント機能と比較し、メッセージ機能は以下の主な特徴があります。

- アプリケーションはメッセージを自由に定義でき、各メッセージに優先度はない
- イベント ID とともにパラメータを送信でき、送信元タスク、送信先タスクを指定できる
- メッセージ送信の際、RWKE またはアプリケーションがメッセージを保持するためのメモリ領域を動的に確保する
- RWKE はメッセージを先入れ先出しのキューで管理する
- 複数のメッセージが送信された場合、RWKE はメッセージが送信された順にメッセージ処理を実行
- 同一のメッセージが複数回送信された場合も、RWKE は送信された回数のメッセージ処理を実行
- RWKE はメッセージ処理も 1 つのイベントとして扱い、メッセージ処理より優先度の高いイベントが発生した場合、メッセージ処理は保留される
- RWKE はメッセージハンドラから KE_MSG_CONSUMED が返却されると、メモリ領域を解放する

メッセージ機能の RWKE API を以下に示します。

表 2-3　メッセージ機能の RWKE API

RWKE API	概要
ke_msg_alloc	メッセージ用メモリ領域を確保
ke_msg_free	メッセージ用メモリ領域を解放
ke_msg_send	メッセージ（メッセージヘッダ＋パラメータ）を送信
ke_msg_send_basic	メッセージヘッダ（メッセージ ID、送信元タスク、送信先タスク）のみ送信
ke_msg_forward	メッセージを転送
ke_msg2param	メッセージヘッダからパラメータを取得
ke_param2msg	パラメータからメッセージヘッダを取得

BLE ソフトウェアに含まれる簡易サンプルプログラムを例として、メッセージ機能を利用するコード例を以下に示します。簡易サンプルプログラムの詳細は、本書の5.1節を参照してください。

1. 新規のメッセージ ID(**APP_MSG_MESSAGE_1**)を定義します。

   ファイル：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. メッセージでパラメータも渡すならば、メッセージパラメータ構造体(**app_param_1_t**)を定義します。

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

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

3. 追加するメッセージのための新規メッセージハンドラ(**app_msg_message_1**)を宣言します。

   ファイル：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. 追加するメッセージのためのメッセージハンドラを実装します。

   ファイル：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;
}
```

   - パラメータを含むメッセージが送信された場合、パラメータはメッセージハンドラの引数 **param** から渡されます。

```
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. メッセージハンドラテーブルに、メッセージ ID とメッセージハンドラを登録します。例として、タスクステートが APP_CONNECT_STATE の場合のメッセージハンドラテーブル(app_connect_handler)に登録します。

   なおメッセージハンドラテーブルは、後述するタスクステート機能で切り替えることができます。

   ファイル：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. アプリケーションの任意の処理内で、メッセージを送信する RWKE API をコールします。

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

   ■ パラメータを含まないメッセージを送信する場合

   ke_msg_send_basic 関数でメッセージを送信します。メッセージを保持するためのメモリ領域は本関数が確保するため、ke_msg_alloc 関数の実行は不要です。

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

   ■ パラメータを含むメッセージを送信する場合

   ke_msg_alloc 関数でメッセージを保持するためのメモリ領域を確保して、メッセージ領域にパラメータを設定後、**ke_msg_send 関数**でメッセージを送信します。

```
{
    ...
    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);
}
```

以上でコード例は完了です。手順6でメッセージを送信すると、手順4で実装したメッセージハンドラが実行されます。

2.3.3　タスクステート機能

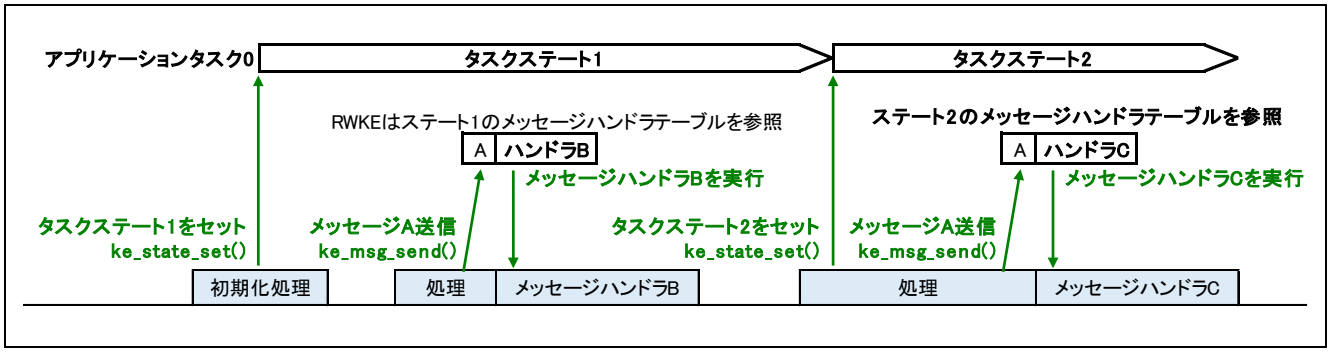
RWKE が提供するタスクステート機能の概要を以下に示します。

- タスクステートは、動的にメッセージシーケンスを変更するための仕組み
- RWKE はタスクステートを取得、設定するための RWKE API を提供する
- RWKE API によってタスクステートが変更されると、RWKE はメッセージハンドラテーブルを切り替える
- 各タスクステートと各メッセージハンドラテーブルは、タスクステートハンドラによって以下のように紐付けられる

タスクステートハンドラ

タスクステート	メッセージハンドラテーブル
タスクステート1	メッセージハンドラテーブル1
タスクステート2	メッセージハンドラテーブル2

- メッセージが送信されると、RWKE は現在のタスクステートのメッセージハンドラテーブルを参照し、メッセージハンドラを実行
- ユースケースは、未接続、接続などの各タスクステートによってメッセージシーケンスを変更し、意図しないメッセージが送信された場合は例外処理を実行する



タスクステート機能は以下の主な特徴があります。

- アプリケーションはタスクステートを自由に定義できる
- アプリケーションは複数のタスクステートを同時に管理できる

タスクステート機能の RWKE API を以下に示します。

表 2-4　タスクステート機能の RWKE API

RWKE API	概要
ke_state_get	現在のタスクステートを取得
ke_state_set	新しいタスクステートを設定

BLE ソフトウェアに含まれる簡易サンプルプログラムを例として、タスクステート機能を利用するコード例を以下に示します。簡易サンプルプログラムの詳細は、本書の5.1節を参照してください。

1. 新規のタスクステート(**APP_STATE1_STATE**)を定義します。

   ファイル：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. 追加したタスクステートのための新規メッセージハンドラテーブル(**app_state1_handler**)を実装します。

   ファイル：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. アプリケーションのタスクステートハンドラ(app_state_handler)に、メッセージハンドラテーブルを登録します。

   ファイル：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),        //APP_RESET_STATE 用テーブル
    KE_STATE_HANDLER(app_nonconnect_handler),   //APP_NONCONNECT_STATE 用テーブル
    KE_STATE_HANDLER(app_connect_handler),      //APP_CONNECT_STATE 用テーブル
    KE_STATE_HANDLER(app_state1_handler),       //APP_STATE1_STATE 用テーブル
};
```

4. アプリケーションの任意の処理内で、タスクステートを設定する **ke_state_set 関数**をコールします。

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

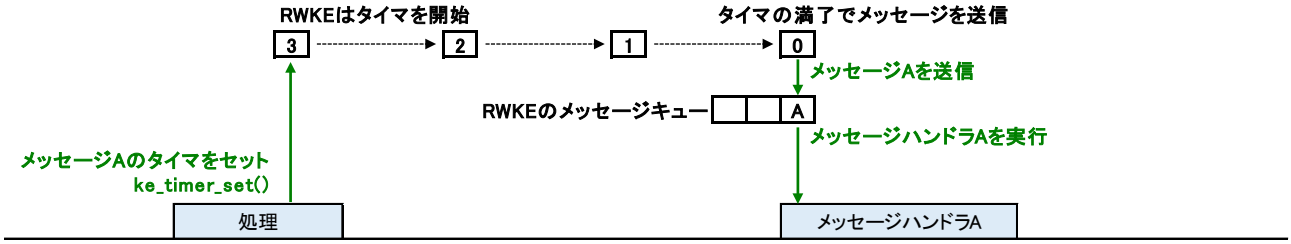
```
{
    ke_state_set(APP_TASK_ID, APP_STATE1_STATE);
    ...
};
```

以上でコード例は完了です。手順4でタスクステートを変更すると、手順2で実装したメッセージハンドラテーブルが適用されます。

2.3.4　タイマ機能

RWKE が提供するタイマ機能の概要を以下に示します。

- タイマは、指定したウェイト時間後にメッセージを送信するための仕組み
- タイマの時間分解能は 10 ミリ秒
- RWKE はタイマを設定、取消するための RWKE API を提供する
- RWKE API によってタイマにメッセージとウェイト時間を設定し、タイマが満了すると RWKE はメッセージを送信
- RWKE は送信されたメッセージに対応するメッセージ処理を実行
- ユースケースは、タイマを繰り返し使用して、アプリケーションの処理を周期的に実行



前述のメッセージ機能と比較し、タイマ機能は以下の主な特徴があります。

- タイマにはメッセージ ID と送信先タスクのみ設定でき、パラメータと送信元タスクは設定できない
- タイマ設定の際、設定情報を保持するためのメモリ領域を RWKE が動的に確保する
- タイマに同一メッセージ ID かつ同一送信先タスクを再度設定した場合、メモリ領域は確保されず、既存のウェイト時間が更新される
- タイマから送信されるメッセージの送信元タスクは空(TASK_NONE)となる

タイマ機能の RWKE API を以下に示します。

表 2-5　タイマ機能の RWKE API

RWKE API	概要
ke_time	現在のタイマ値を取得
ke_timer_set	メッセージ送信タイマを設定
ke_timer_clear	メッセージ送信タイマを取消

BLE ソフトウェアに含まれる簡易サンプルプログラムを例として、タイマ機能を利用するコード例を以下に示します。簡易サンプルプログラムの詳細は、本書の5.1節を参照してください。

1. 新規のメッセージ ID(**APP_MSG_TIMER_1**)を定義します。

   ファイル：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. 追加するメッセージのための新規メッセージハンドラ(**app_msg_timer_1**)を宣言します。

   ファイル：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. 追加するメッセージのための新規メッセージハンドラを実装します。

   ファイル：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. メッセージハンドラテーブルに、メッセージ ID とメッセージハンドラを登録します。例として、タスクステートが APP_CONNECT_STATE の場合のメッセージハンドラテーブル(app_connect_handler)に登録します。

   ファイル：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. アプリケーションの任意の処理内で、タイマをセットする **ke_timer_set 関数**をコールします。例として、1 秒（10 ミリ秒×100）のウェイトを設定します。

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

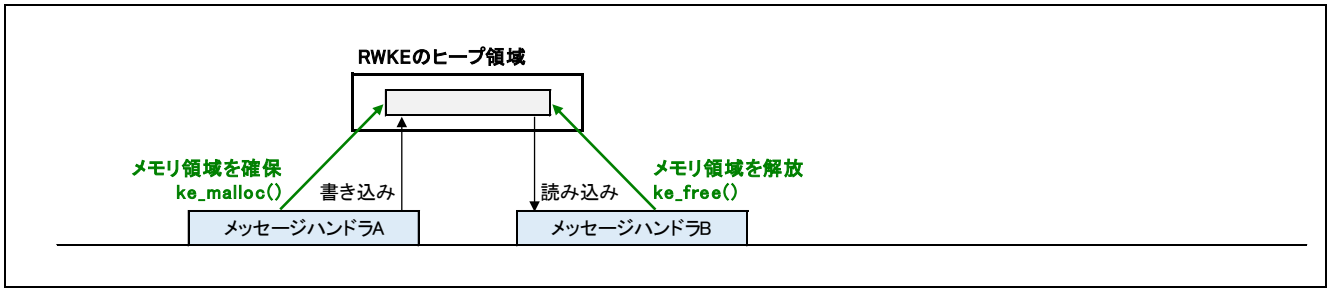
```
{
    ...
    ke_timer_set(APP_MSG_TIMER_1, APP_TASK_ID, 100);
}
```

以上でコード例は完了です。手順5でタイマを設定して 1 秒が経過すると、手順3で実装したメッセージハンドラが実行されます。

2.3.5　メモリ機能

RWKE が提供するメモリ機能の概要を以下に示します。

- メモリ機能は、任意のサイズのメモリ領域を使用するための仕組み
- RWKE はヒープ領域内のメモリ領域を確保、解放するための RWKE API を提供する
- アプリケーションはサイズを指定してメモリ領域を確保し、書き込み、読み込みを実行する
- アプリケーションでメモリ領域が不要となったならば、メモリ領域を解放する



メモリ機能の RWKE API を以下に示します。

表 2-6　メモリ機能の RWKE API

RWKE API	概要
ke_malloc	ヒープ領域から一部のメモリ領域を確保
ke_free	ヒープ領域から一部のメモリ領域を解放

BLE ソフトウェアに含まれる簡易サンプルプログラムを例として、メモリ機能を利用するコード例を以下に示します。簡易サンプルプログラムの詳細は、本書の5.1節を参照してください。

1. **ke_malloc 関数**をコールしてメモリ領域を確保し、データを設定します。

   ファイル：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. 格納されたデータを参照し、不要となったら **ke_free 関数**をコールしてメモリ領域を解放します。

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

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

以上でコード例は完了です。

2.3.6　割り込みハンドラから利用可能な RWKE API

割り込みハンドラから利用可能な RWKE API を以下に示します。

ke_evt_set	：イベントをセット
ke_evt_clear	：イベントをクリア
ke_msg_alloc※	：メッセージ用メモリ領域を確保
ke_msg_send※	：メッセージ（メッセージヘッダ＋パラメータ）を送信
ke_msg_send_basic※	：メッセージヘッダ（メッセージ ID、送信元タスク、送信先タスク）のみ送信

※これらの API は処理時間が長いため、割り込みハンドラからの利用は推奨されません。後述するユースケースの通り、割り込みハンドラからイベントを通知し、イベントハンドラからメッセージを送信します。

2.3.7　RWKE 関連の主なリソース

RWKE 関連の主なリソースを以下に示します。

アプリケーションのリソース実装については2.5節「アプリケーションの実装」を参照してください。

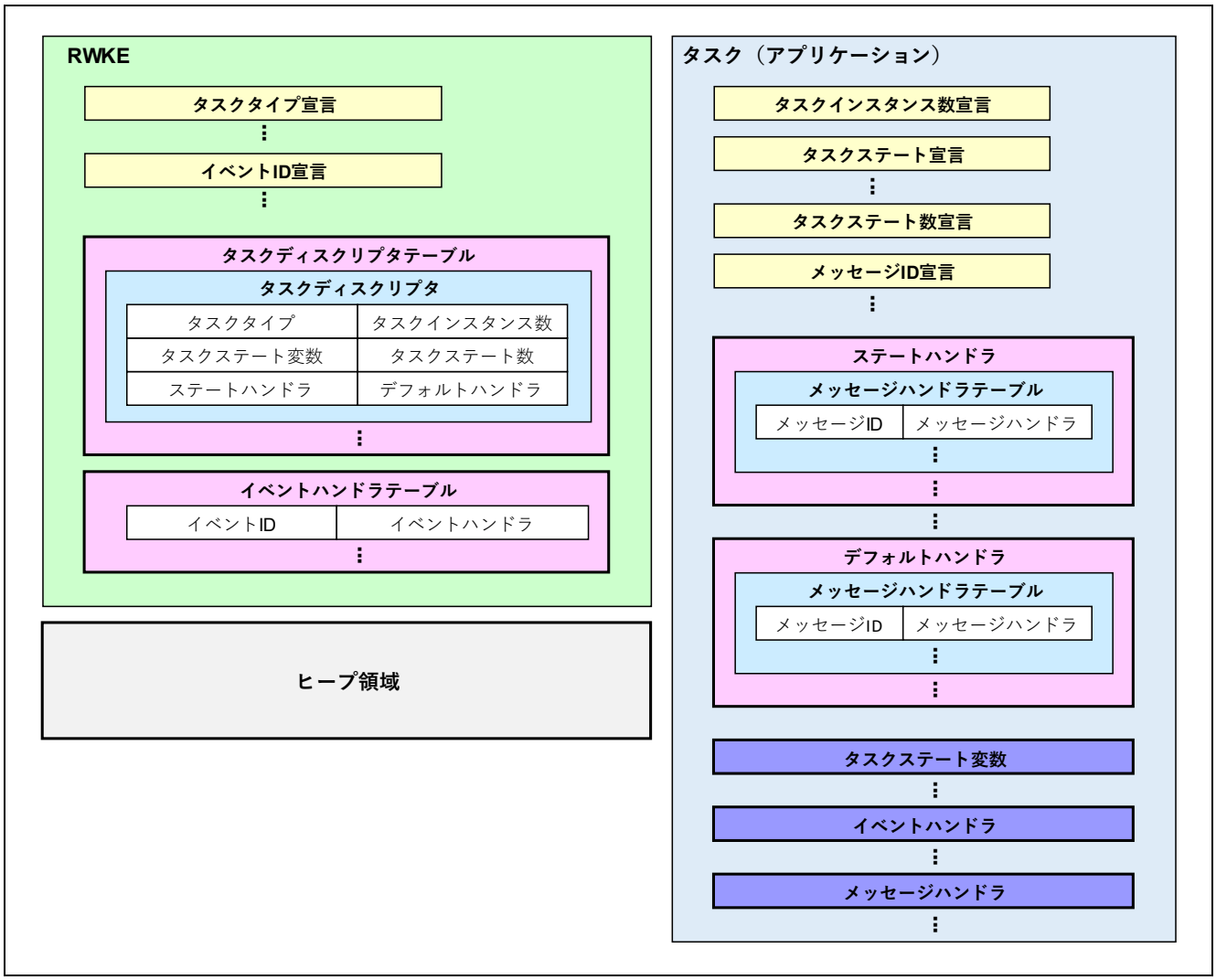


図 2-5　RWKE 関連の主なリソース

簡易サンプルプログラムの場合の、各リソースのシンボル名と実装ファイル名を以下に示します。簡易サンプルプログラムの詳細は、本書の5.1節を参照してください。

表 2-7　RWKE 関連の主なリソース

リソース	シンボル名 ("*"はワイルドカードを示す)	実装ファイル名
タスクタイプ	TASK_*	rwke_api.h
イベント ID	KE_EVT_*	
タスクディスクリプタテーブル	TASK_DESC_ent	ke_conf_simple.c
イベントハンドラテーブル	ke_evt_hdlr_ent	
ヒープ領域	ke_mem_heap_ent	arch_main.c
タスクインスタンス数	APP_IDX_MAX	rble_sample_app_peripheral.h
タスクステート	APP_RESET_STATE APP_NONCONNECT_STATE APP_CONNECT_STATE	
タスクステート数	APP_STATE_MAX	
メッセージ ID	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	
ステートハンドラ	app_state_handler	rble_sample_app_peripheral.c
デフォルトハンドラ	app_default_handler	
メッセージハンドラテーブル	app_reset_handler app_nonconnect_handler app_connect_handler	
タスクステート変数	app_state	
メッセージハンドラ	app_reset app_advertise_start app_profile_enable app_profile_disable app_timer_expired	
イベントハンドラ	(未使用)	(未使用)

2.4　ユースケース

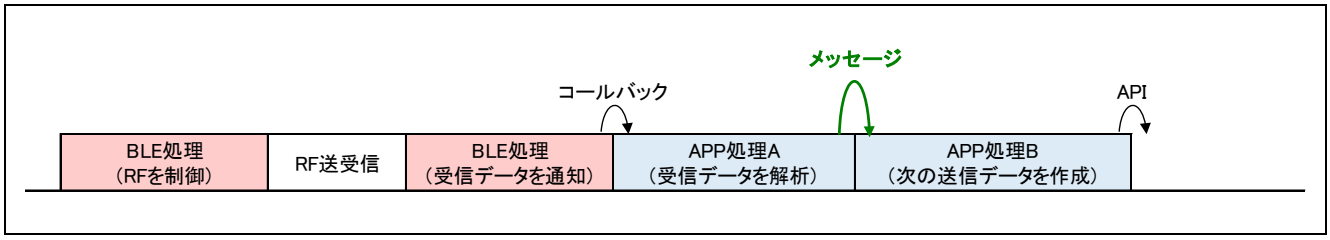
RWKE の機能を利用した、アプリケーションによる下記のユースケースを示します。

- BLE無線通信で受信したデータを利用するシーケンス
- 指定時間の経過後に処理を実行するシーケンス
- RL78/G1D周辺機能からの割り込みで処理を実行するシーケンス

2.4.1　BLE 無線通信で受信したデータを利用するシーケンス

BLE 無線通信で受信したデータを利用するシーケンスを示します。

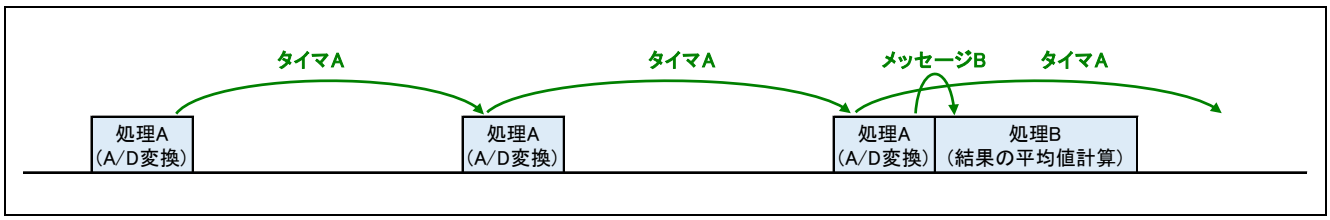
BLE プロトコルスタックは RF 送受信の実行によりデータを受信すると、コールバック関数により受信データをアプリケーションに通知します。アプリケーションは受信したデータを解析し、条件に応じて異なるメッセージを送信することで、柔軟なアプリケーション処理のシーケンスを実行できます。



2.4.2　指定時間の経過後に処理を実行するシーケンス

指定時間の経過後に処理を実行するシーケンスを示します。

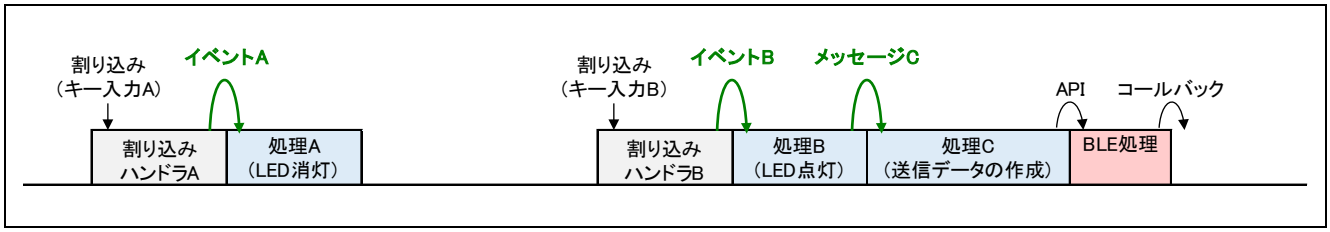
アプリケーションはタイマを繰り返し利用することにより、周期的に処理を実行することができます。



2.4.3　RL78/G1D 周辺機能からの割り込みで処理を実行するシーケンス

RL78/G1D 周辺機能からの割り込みで処理を実行するシーケンスを示します。

RL78/G1D 周辺機能からの割り込みで、まず割り込みハンドラを実行します。割り込みハンドラからイベントをセットして、イベントハンドラを実行します。さらにシーケンスを実行する場合、イベントハンドラからメッセージを送信し、メッセージハンドラを実行します。



2.5　アプリケーションの実装

アプリケーションを実装するには、RWKE にアプリケーションタスクを登録します。アプリケーションタスクの登録には、以下に示すタスクタイプとタスクディスクリプタが必要です。

タスクタイプ

タスクタイプは以下のように定義されます。アプリケーションタスクには TASK_USR_0 と TASK_USR_1 が予約されています。新しくアプリケーションタスクを追加する場合は TASK_APP より下に定義します。

ファイル：renesas/src/arch/rl78/rwke_api.h

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

    TASK_APP = TASK_RBLE,
    ...
};
```

タスクディスクリプタ構造体

タスクディスクリプタ構造体は、RWKE によって以下のように定義されます。

ファイル：renesas/src/arch/rl78/rwke_api.h

```
struct ke_task_desc
{
    const struct ke_state_handler *state_handler;     ····· ステートハンドラ
    const struct ke_state_handler *default_handler;   ····· デフォルトハンドラ
    ke_state_t *state;                                ····· タスクステート変数
    const uint16_t state_max;                         ····· タスクステート数
    const uint16_t idx_max;                           ····· タスクインスタンス数
};
```

タスクタイプとタスクディスクリプタは、タスクディスクリプタテーブルに登録します。タスクディスクリプタテーブル(**TASK_DESC_ent**)は、下記のように実装されています。

ファイル：renesas/src/arch/rl78/ke_conf_simple.c　※簡易サンプルプログラムの場合
ファイル：renesas/src/arch/rl78/ke_conf.c　※簡易サンプルプログラム以外の場合

```
/// 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 },
};
```

ステートハンドラ、タスクステート数

ステートハンドラは、各タスクステートでのメッセージハンドラテーブルを紐付けます。RWKE はメッセージが送信されると、ステートハンドラを参照します。そして現在のタスクステートとして登録されたメッセージハンドラテーブルを参照して、メッセージ ID と対応付けられたメッセージハンドラを実行します。

また RWKE は、ステートハンドラを参照する際、アプリケーションが定義するタスクステートの数を必要とします。

このステートハンドラとタスクステート数がタスクディスクリプタに登録されます。

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

簡易サンプルプログラムでの実装を以下に示します。

アプリケーションは３つのタスクステートを定義しており、タスクステート数は **APP_STATE_MAX** です。

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

```
typedef enum {
    APP_RESET_STATE = 0,                ..... リセット状態
    APP_NONCONNECT_STATE,               ..... 未接続状態
    APP_CONNECT_STATE,                  ..... 接続状態
    APP_STATE_MAX                       ..... タスクステート数
} APP_STATE;
```

ステートハンドラ(**app_state_handler**)には、各タスクステートでのメッセージハンドラテーブルが登録されます。RWKE はステートハンドラを参照することで、各タスクステートでのメッセージハンドラテーブルを識別します。

ファイル：rBLE/src/sample_simple/rble_sample_app_peripheral.c　※簡易サンプルプログラムの場合

```
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),
};
```

上記のステートハンドラでは、タスクステートが APP_NONCONNECT_STATE の場合のメッセージハンドラテーブルとして **app_nonconnect_handler** が登録されています。RWKE はアプリケーションのタスクステートが APP_NONCONNECT_STATE の場合にメッセージが送信されると、app_nonconect_handler を参照し、メッセージ ID と対応付けられたメッセージハンドラを実行します。

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

```
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 },
};
```

デフォルトハンドラ

RWKE は前述のメッセージハンドラテーブルを参照し、メッセージ ID に対応付けられたメッセージハンドラが登録されていない場合、デフォルトハンドラに登録されたメッセージハンドラを実行します。

このデフォルトハンドラがタスクディスクリプタに登録されます。

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

簡易サンプルプログラムでの実装を以下に示します。

デフォルトハンドラが必要ない場合、デフォルトハンドラで **KE_STATE_HANDLER_NONE** を指定します。

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

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

タスクステート変数、タスクインスタンス数

タスクステート変数は、アプリケーションのタスクステートを保持します。RWKE はこのタスクステート変数を参照または変更します。

また複数のタスクステート変数を登録することで、同時に複数のタスクステートを保持できます。このタスクステート変数の数は、タスクインスタンス数と呼ばれます。

このタスクステート数とタスクインスタンス数がタスクディスクリプタに登録されます。

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

簡易サンプルプログラムでの実装を以下に示します。

タスクインスタンス数 **APP_IDX_MAX** は 1 と定義されています。

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

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

タスクステート変数 **app_state** は配列として定義されています。

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

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

なお、各タスクインスタンスは 0 ～ (APP_IDX_MAX - 1) までのインデックスで識別します。RWKE API を使用する際には、前述のタスクタイプとマクロ KE_BUILD_ID を使用して、タスク ID を作成します。

※インデックスが 0 の場合、タスクタイプとタスク ID は同一の値となります。

```
アプリケーションのタスク ID  = KE_BUILD_ID(TASK_USR_0, idx);
```

2.6 注意点

RWKE を使用したアプリケーションを作成するにあたり、意図しない動作を避けるための注意点について説明します。

アプリケーションの実装について

- BLE プロトコルスタックの各機能は、RWKE によってスケジューリングされます。BLE 無線通信の意図しない動作を避けるため、アプリケーションもまた RWKE によって管理される必要があります。

  アプリケーションの処理は、原則的には RWKE の各機能によって管理するように実装してください。

- BLE プロトコルスタックの RF 制御処理は、優先度の高いイベントとして RWKE に管理されます。BLE プロトコルスタックのスケジューリングへの影響を低減するため、アプリケーションのイベントハンドラ、メッセージハンドラ、rBLE に登録するコールバック関数の各処理は短時間であることが推奨されます(推奨時間は接続インターバルの 30%以内)。

  長時間となる処理は、RWKE のメッセージ機能を使用して、複数のメッセージハンドラとして分割して実行してください。

- RWKE は、割り込みを管理しません。アプリケーションが周辺機能の割り込みを使用する場合、BLE プロトコルスタックのスケジューリングへの影響を低減するため、割り込みハンドラの実行時間が最小限(推奨時間は 1ms 以内)となるよう実装する必要があります。

  また、割り込みを使用する際には、前述のユースケースに従って RWKE のイベント機能を使用し、RWKE によって管理される処理シーケンスを実行することを推奨します。

メッセージ機能について

- メッセージの送信はヒープ領域の一部を確保するため、多数のメッセージが送信され続けるといずれはヒープ領域が枯渇します。アプリケーションは、同時に多数のメッセージがキューに設定されないようなシーケンスを実装する必要があります。

rBLE API の使用について

- 後述する rBLE API 関数もまた、BLE プロトコルスタックの機能を実行するためにメッセージ機能を使用します。ヒープ領域の枯渇を防止するため、原則的には 1 つの rBLE API 関数を実行後、完了ステータスが通知されるまで、次の API 関数を実行しないことが推奨されます。

- BLE プロトコルスタックの rBLE API である RBLE_GAP_Reset 関数を実行すると、BLE プロトコルスタックのスケジューリングの不整合を防止するため、RWKE の各機能も初期化します。

  アプリケーションは、RBLE_GAP_Reset 関数による GAP リセット処理の完了後に処理シーケンスを開始する必要があります。

ヒープ領域について

- RWKE が管理するヒープ領域のサイズは、ユーザが変更できます。アプリケーションの実行シーケンスにより、ヒープ領域が枯渇する可能性がある場合は、必要に応じてサイズを拡張してください。

  またヒープ領域を拡張する際には、スタックメモリ領域が不足しないようにする必要があります。

  ヒープ領域 **ke_mem_heap_ent** の実装コードを以下に示します。

  ファイル：Renesas/src/arch/rl78/arch_main.c

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

Sleep 機能について

- BLE ソフトウェアは RL78/G1D の MCU と RF を自動で低消費電力状態に遷移させる機能を持ちます。

  Sleep 機能により MCU と RF は以下のいずれかの低消費電力状態となります。

  - RF のみスタンバイ(SLEEP モードまたは DEEP_SLEEP モード)に遷移
  - RF もスタンバイ(SLEEP モードまたは DEEP_SLEEP モード)、MCU もスタンバイ(STOP モード)に遷移

  Sleep 機能の詳細は、Bluetooth low energy プロトコルスタック ユーザーズマニュアル(R01UW0095)の 7.20.2 項「Sleep 機能」を参照してください。

MCU の STOP モードについて

- RWKE が実行すべきイベント、メッセージが空となると、Sleep 機能は MCU を STOP モードに遷移させます。

  MCU を STOP モードに遷移させる処理の実装コードを以下に示します。

  ファイル：renesas/src/arch/rl78/arch_main.c

```
#if defined(__USE_CCRL_RL78)
 #define WFI()  __stop();
#else
 #define WFI()  __asm("stop");
#endif
```

  STOP モードでは一部を除き、シリアル・アレイ・ユニット、タイマ・アレイ・ユニット、A/D コンバータ、DMA コントローラといった周辺機能が停止状態となります。詳細は、RL78/G1D ユーザーズマニュアル ハードウェア編(R01UH0515)の 19 章「スタンバイ機能」を参照してください。

  周辺機能を使用する必要がある場合、条件に応じて MCU を STOP モードではなく一時的に HALT モードに遷移させる、または sleep_check_enable 関数を利用して、MCU の STOP モードへの遷移を一時的に禁止してください。

3. BLE プロトコルスタック

3.1 BLE プロトコルスタックとは

BLE プロトコルスタックとは、RL78/G1D の RF 部を管理し、Bluetooth low energy 無線通信に必要な機能をアプリケーションに提供するソフトウェアスタックです。

BLE プロトコルスタックが提供する機能の詳細は、下記を参照してください。

Bluetooth low energy プロトコルスタック ユーザーズマニュアル (R01UW0095)
https://www.renesas.com/document/man/bluetooth-low-energy-protocol-stack-users-manual

- 7 章「機能説明」

また BLE プロトコルスタックの機能を実行するには、rBLE API を使用します。rBLE API 仕様の詳細は、下記を参照してください。

Bluetooth low energy プロトコルスタック API リファレンスマニュアル 基本編 (R01UW0088)
https://www.renesas.com/document/man/bluetooth-low-energy-protocol-stack-api-reference-manual-basics

- 3 章「Common Definitions」
- 4 章「Initialization」
- 5 章「Generic Access Profile」
- 6 章「Security Manager」
- 7 章「Generic Attribute Profile」
- 8 章「Vendor Specific」

3.2 rBLE API

BLE プロトコルスタックは rBLE API を提供します。アプリケーションは rBLE API を使用することで、BLE プロトコルスタックの機能を利用することができます。

rBLE API は次のコマンドとイベントを定義します。

コマンド

BLE プロトコルスタックの各機能の動作を制御するための要求
アプリケーションから BLE プロトコルスタックに渡される

イベント

BLE プロトコルスタックの各機能の実行結果を示す通知
BLE プロトコルスタックからアプリケーションに渡される

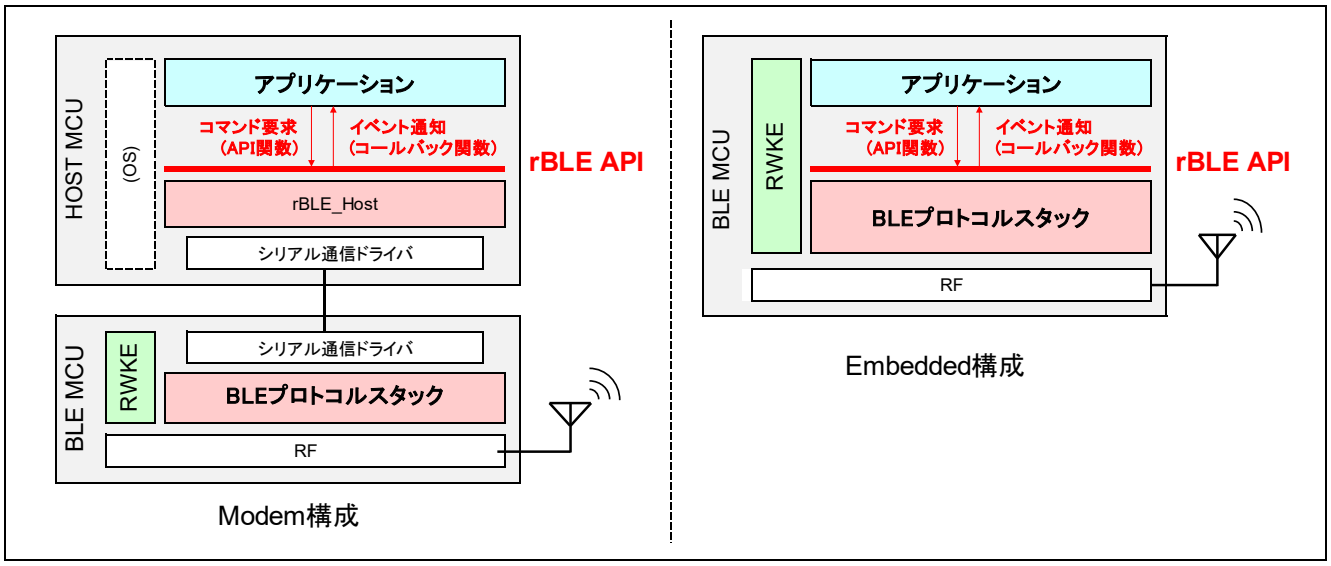


図 3-1　BLE プロトコルスタックと rBLE API

rBLE API は、ノンブロッキング関数として実装されています。つまりアプリケーションがコマンド要求の API 関数をコールすると、API 関数はコマンドの実行完了を待つことなく終了します。その後、BLE プロトコルスタックによってコマンドが実行され、その実行結果を示すイベントはコールバック関数の引数によって通知されます。

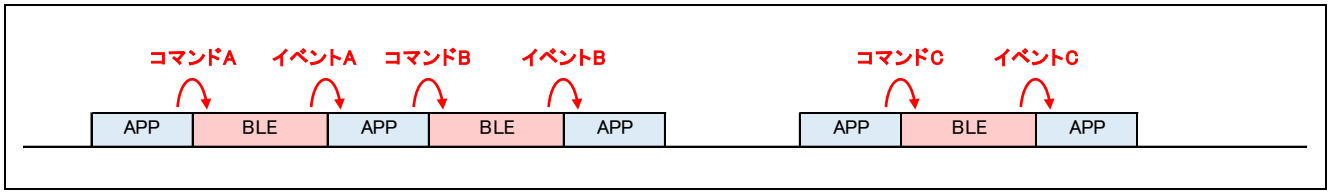


図 3-2　コマンドとイベントの実行例

アプリケーションが rBLE API を利用する手順は下記となります。詳細は次頁以降で示します。

1. イベントを受け取るためのコールバック関数を登録
2. コマンド要求の API 関数を実行
3. コールバック関数の引数からイベントを受け取る
4. イベントを参照し、次のコマンド要求を実行するかを判断する→手順2に戻る

3.3　rBLE コマンドの呼び出し方

アプリケーションが BLE プロトコルスタックの機能を利用するには、rBLE コマンド API を呼び出します。

まずは"rble_api.h"ヘッダファイルをインクルードしてください。例として GATT 機能を使用するには、RBLE_GATT_Enable 関数を最初に呼び出して GATT 機能を初期化します。初期化の完了後、コマンドに引数として渡すためのパラメータを用意し、コマンド API を呼び出します。

RBLE_GATT_Notify_Request API を呼び出すコード例を以下に示します。

ファイル：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;          コネクションハンドル
    ntf.charhdl = sams_info.hdl;            キャラクタリスティックハンドル

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

図 3-3　rBLE コマンドの呼び出し例

3.4 rBLE イベントの受け取り方

アプリケーションは rBLE コマンドの実行後、rBLE イベントをコールバック関数で受け取ります。BLE プロトコルスタックから rBLE イベントを受け取るためには、コールバック関数の準備と登録を行います。

3.4.1 コールバック関数の準備

コールバック関数は、引数として rBLE イベント構造体を持ちます。

イベント構造体は"rble_api.h"ヘッダファイルに定義されており、**イベントタイプ**と**イベントパラメータ共用体**で構成されます。

イベント構造体の例として RBLE_GAP_EVENT の定義を以下に示します。

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;                    イベントタイプ
    uint8_t                 reserved;
    union Event_Parameter_u {                        イベントパラメータ
        /* 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;
```

図 3-4　イベント構造体の例

イベント構造体のイベントタイプには、イベントの識別するための値が設定されます。

ファイル：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 */
    ...
};
```

図 3-5　イベントタイプの例

イベント構造体を参照するためにまずは"rble_api.h"をインクルードします。コールバック関数はイベントタイプを確認し、アプリケーションのシーケンスとして必要な処理を実行します。

GAP 機能の rBLE イベントを受け取るコード例を以下に示します。

ファイル：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;
    ...
    }
}
```

図 3-6　コールバック関数の例

3.4.2　コールバック関数の登録

イベントを受け取るために、rBLE にコールバック関数を登録します。アプリケーションは使用する BLE プロトコルスタックの機能毎にコールバック関数を登録します。

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

```
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;
}
```

図 3-7　コールバック関数の登録例

コールバック関数登録用 API を以下に示します。

表 3-1　コールバック関数登録用 API 一覧

機能	rBLE API	コールバック関数登録用 API	イベント識別子
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. プロファイル

4.1 プロファイルとは

プロファイルは、接続を確立したデバイス間でデータを通信するための、GATT 上のデータ構造や手続きを定義したものです。

プロファイルは下記のような階層を持ちます。サービス、インクルードサービス、キャラクタリスティックといったアトリビュートで構成されています。

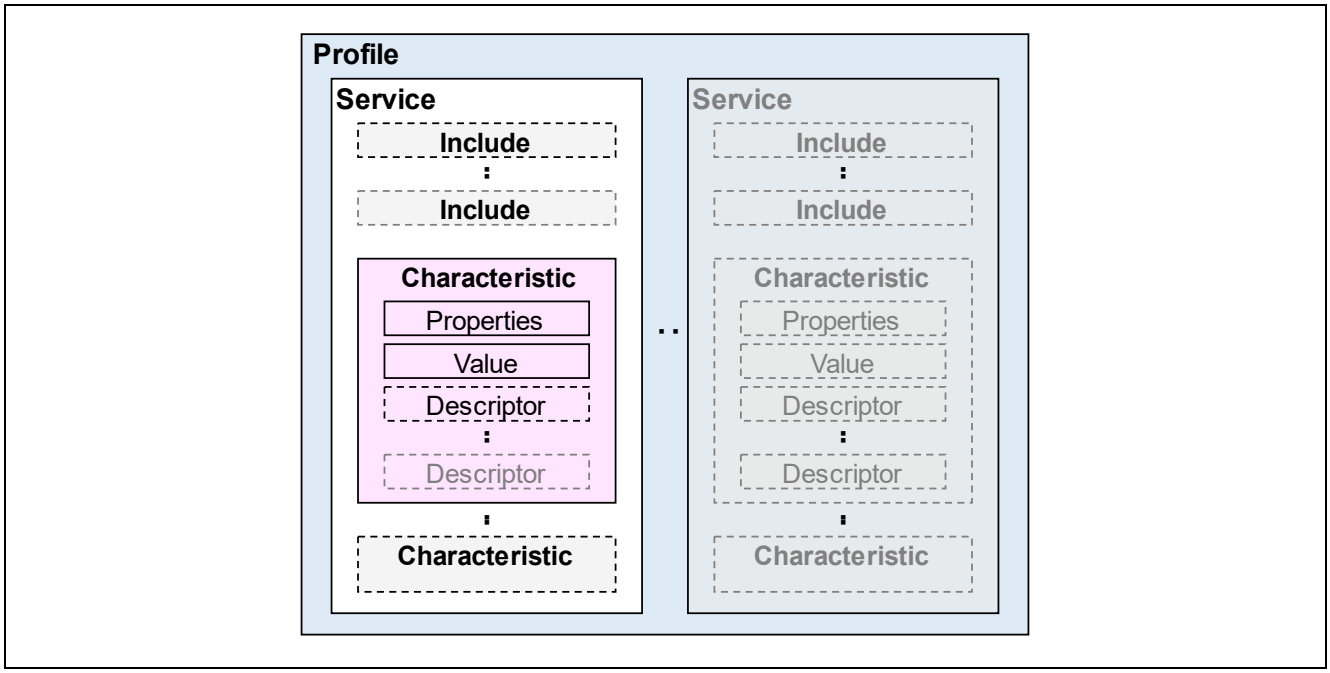


図 4-1　プロファイル階層

サービス

特定の機能や機構を実行するためのデータや関連する動作の集合

インクルードサービス

サービスの一部として、サーバ上の別のサービス定義を参照する仕組み

キャラクタリスティック

アクセス方法や表示方法に関するプロパティと、設定情報に従ってサービス内で利用される値
下記のような、値に関しての説明やサーバの設定を許可するディスクリプタを含む場合がある

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

注：上記は一例です。詳細は下記を参照してください。
https://www.bluetooth.com/specifications/assigned-numbers/

ディスクリプタの中で Client Characteristic Configuration Descriptor は重要です。本ディスクリプタはキャラクタリスティックの値をサーバからクライアントへ通知する動作（Notification または Indication）を許可するために使用します。デフォルト値は Notification と Indication が禁止であることを意味する 0x0000 が設定され、これらを許可するには、クライアントがそれぞれ 0x0001 と 0x0002 を設定します。また、本設定値は接続済みのデバイス間で維持する必要があります。

表 4-1　Client Characteristic Configuration Descriptor の設定値

設定	値	説明
Notification/Indication 禁止	0x0000	Notification もしくは Indication してはならない。
Notification 許可	0x0001	キャラクタリスティック・バリューを Notification することが可能
Indication 許可	0x0002	キャラクタリスティック・バリューを Indication することが可能

デバイスは、プロファイルで定義されたサーバまたはクライアントとして通信します。またサーバは、サービスのデータ構造を定義するための GATT データベースを持ちます。

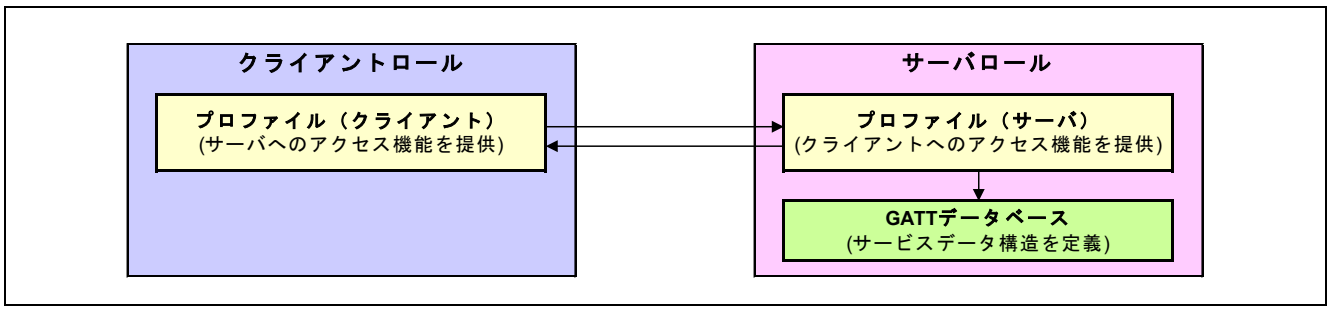


図 4-2　プロファイルの構成

複数のプロファイルが Bluetooth SIG によって採択されています。BLE ソフトウェアは、15 の採択済みプロファイルをサポートし、アプリケーションは rBLE API を通してそれらのプロファイルを利用できます。

一方、採択済みプロファイルのうち BLE プロトコルスタックがサポートしないプロファイルや、独自の機能を実現するためのプロファイルを利用する必要がある場合、アプリケーションはカスタムプロファイルとしてアプリケーション内に実装します。

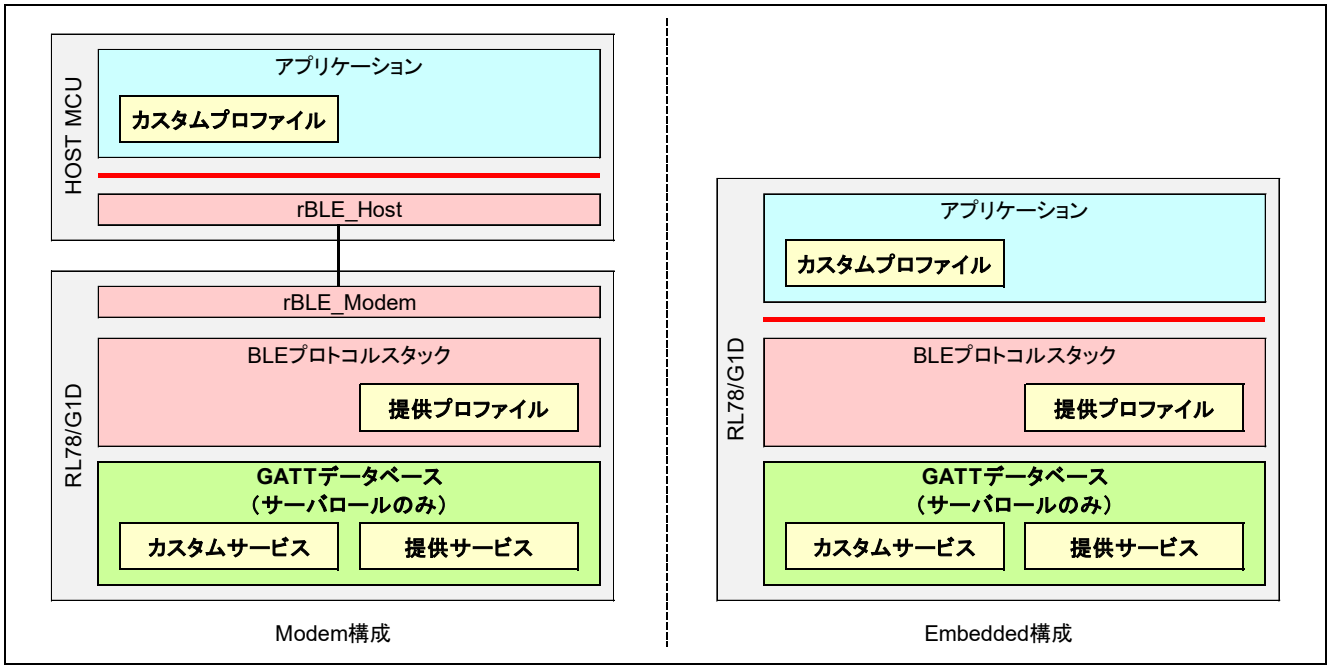


図 4-3　プロファイルとサービスの実装

4.2 GATT データベースとは

GATT データベースは、プロファイルのサービスデータ構造を定義する要素の集合で、サーバロールのアプリケーションが GATT データベースを実装し、BLE プロトコルスタックに設定する必要があります。

GATT データベースの各要素は"アトリビュート(属性)"と呼ばれ、次の 4 つパラメータを持っています。

アトリビュートハンドル

アトリビュートを特定するためのインデックス
16 ビットの値であり、0x0001 から 0xFFFF が割り当てられる

アトリビュートタイプ

アトリビュート値を識別するための UUID
128 ビットの値であるが、Bluetooth SIG によって採択された UUID は短縮した 16 ビットで利用される

アトリビュート値

アトリビュートのデータ
アトリビュートタイプによってデータ構造は異なる

アトリビュートパーミッション

アトリビュート値へのアクセス方法を指定するための設定

GATT データベースは、下記の 2 箇所で定義します。

- "prf_config_host.c"ファイルの atts_desc_list_host[] : GAP と GATT のデータベース
- "prf_config.c"ファイルの atts_desc_list_prf[] : Profile のデータベース

カスタムプロファイルを実装する場合は、 atts_desc_list_prf[]にサービス、キャラクタリスティックを追加します。

4.3 GATT データベースの作り方

カスタムプロファイルを作成するには、まず以下のような内容を検討する必要があります。

- どのような機能を実行する？(サービス)
- どのような種類のデータを扱う？(キャラクタリスティック構成)
- どのようなデータ構造？(各キャラクタリスティック・バリューのデータサイズ、構造体要素)
- どのようにデータが送受信される？(各キャラクタリスティックのパーミッション)

まずはカスタムプロファイルで扱うサービスデータ構造を設計し、GATT データベースにカスタムサービスを実装します。

4.3.1　データベースハンドルの追加

各サービスと各キャラクタリスティックのデータベースハンドルを追加します。データベースハンドルはクライアントに公開され、クライアントがサーバのサービスやキャラクタリスティックにアクセスするために使用されます。

データベースハンドルは、"db_handle.h"ヘッダファイルの DB_HDL_MAX より前に追加します。

ファイル：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
};
```

図 4-4　データベースハンドルの追加例

4.3.2　データベースインデックスの追加

サービスとキャラクタリスティックのデータベースインデックスを追加します。データベースインデックスは、BLE プロトコルスタックがデータベースの要素を識別するために使用します。

データベースの追加は、"prf_config.h"ヘッダファイルの列挙体の最後で定義します。

ファイル：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
};
```

図 4-5　データベースインデックスの追加例

4.3.3　UUID の定義

サービスとキャラクタリスティックの UUID を定義します。Bluetooth SIG によって採択されていないカスタムサービスとそのキャラクタリスティックの UUID は、128bit のランダム値で定義します。

UUID は"prf_config.c"ファイルから参照できるように定義します。

ファイル：rBLE/src/sample_simple/sam/sam.h

```
#define RBLE_SVC_SAMPLE_CUSTOM_SVC  {0x7A,0x8D,...,0xF1,0xA1,0xF7,0xB9,0xC1,0x5B}
#define RBLE_CHAR_SAMS_SWITCH_STATE {0x7A,0x8D,...,0xF1,0xA1,0x80,0x8D,0xC1,0x5B}
#define RBLE_CHAR_SAMS_LED_CONTROL  {0x7A,0x8D,...,0xF1,0xA1,0xEE,0x43,0xC1,0x5B}
```

図 4-6　UUID の定義例

4.3.4　サービスの定義

サービスを定義します。サービスのアトリビュート値として4.3.3項で定義した UUID を割り当てます。

サービスは"prf_config.c"ファイルに定義します。

ファイル：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;
```

図 4-7　サービスの定義例

4.3.5　キャラクタリスティックの定義

キャラクタリスティックを定義します。キャラクタリスティックにはプロパティ、アトリビュートハンドル、キャラクタリスティックの UUID を設定します。キャラクタリスティックは"prf_config.c"ファイルに定義します。

ファイル：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,                                  プロパティ
    {
        (uint8_t)(SAMS_HDL_SWITCH_STATE_VAL & 0xff),    アトリビュートハンドル
        (uint8_t)((SAMS_HDL_SWITCH_STATE_VAL >> 8) & 0xff)
    },
    RBLE_CHAR_SAMS_SWITCH_STATE                               キャラクタリスティックの UUID
};
```

図 4-8　キャラクタリスティックの定義例

キャラクタリスティック・バリューを定義します。

キャラクタリスティック・バリューは”prf_config.c”ファイルに定義します。

ファイル：renesas/src/arch/rl78/prf_config.c

```
uint8_t switch_state_char_val[RBLE_ATTM_MAX_VALUE] = {0};
                                                    キャラクタリスティック・バリュー
struct atts_elmt_128 switch_state_char_val_elmt = {
      RBLE_CHAR_SAMS_SWITCH_STATE,                  キャラクタリスティックの UUID
      RBLE_GATT_128BIT_UUID_OCTET,                  UUID 長
      &switch_state_char_val[0] };                  バリューへのポインタ
```

図 4-9　キャラクタリスティック・バリューの定義例

4.3.6　データベースへの追加

最後に、既に定義したサービス、キャラクタリスティック、キャラクタリスティック・バリューを GATT データベースに追加します。これらの定義は、”prf_config.c”ファイルの atts_desc_list_prf[]に追加します。

ファイル：renesas/src/arch/rl78/prf_config.c

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

    /*********************************
     * Simple Sample Service        *
     *********************************/
    { RBLE_DECL_PRIMARY_SERVICE,                    サービス
      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,                     キャラクタリスティック
      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,                          キャラクタリスティック・バリュー
      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}
};
```

図 4-10　データベースの追加例

以上で GATT データベースの作成は完了です。次に GATT データベースを利用したカスタムプロファイルの作り方を説明します。

4.4　カスタムプロファイルの作り方

4.4.1　サーバロール

サーバロールとして、BLE ソフトウェアは以下の動作を行います。

- クライアントにサービスのデータを通知 (Notification / Indication)
- クライアントから通知確認を受取
- クライアントから書き込みデータを受取
- クライアントからの書き込み要求に応答 (Write Response)
- クライアントからの読み出し要求に応答 (Read Response; 自動応答)
- GATT データベースにデータを設定

カスタムプロファイルのサーバが利用する GATT 機能のマンドとイベントを以下に示します。

イベントを受け取るためには、最初に RBLE_GATT_Enable をコールして、コールバック関数を登録する必要があります。

表 4-2　サーバロールで利用する GATT API

コマンド
RBLE_GATT_Enable
RBLE_GATT_Notify_Request
RBLE_GATT_Indicate_Request
RBLE_GATT_Write_Response
RBLE_GATT_Set_Permission
RBLE_GATT_Set_Data

イベント
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_COMP
RBLE_GATT_EVENT_SET_DATA_COMP
RBLE_GATT_EVENT_NOTIFY_COMP

4.4.2　クライアントロール

クライアントロールとして、BLE ソフトウェアは以下の動作を行います。

- サーバのサービス／キャラクタリスティック／ディスクリプタの発見 (Discovery)
- サーバへのデータの書き込み　(Write Request)
- サーバからデータの読み出し　(Read Request)
- サーバからのデータ通知を受取
- サーバに受信確認を送信　(Confirmation)

カスタムプロファイルのクライアントが利用する GATT 機能のコマンドとイベントを以下に示します。

イベントを受け取るためには、最初に RBLE_GATT_Enable をコールして、コールバック関数を登録する必要があります。

表 4-3　クライアントロールで利用する GATT API

コマンド
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

イベント
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　プロファイルによるデータアクセス

プロファイルがデータを扱うための基本的な通信プロトコルを以下に示します。

表 4-4　基本通信プロトコルと通信方向の関係

コマンド	方向	説明
Read	クライアント → サーバ	クライアントはサーバにキャラクタリスティック・バリュー読み出しを要求する
Write without Response	クライアント → サーバ	クライアントはサーバにキャラクタリスティック・バリュー書き込みを要求し、応答を必要しない
Write	クライアント → サーバ	クライアントはサーバにキャラクタリスティック・バリュー書き込みを要求する
Response	サーバ → クライアント	サーバはクライアントからの読み出しや書き込み要求に応答する
Notification	サーバ → クライアント	サーバはクライアントにキャラクタリスティック・バリューを通知し、応答を必要としない
Indication	サーバ → クライアント	サーバはクライアントにキャラクタリスティック・バリューを通知する
Confirmation	クライアント → サーバ	クライアントはサーバからの通知を確認して応答を返す

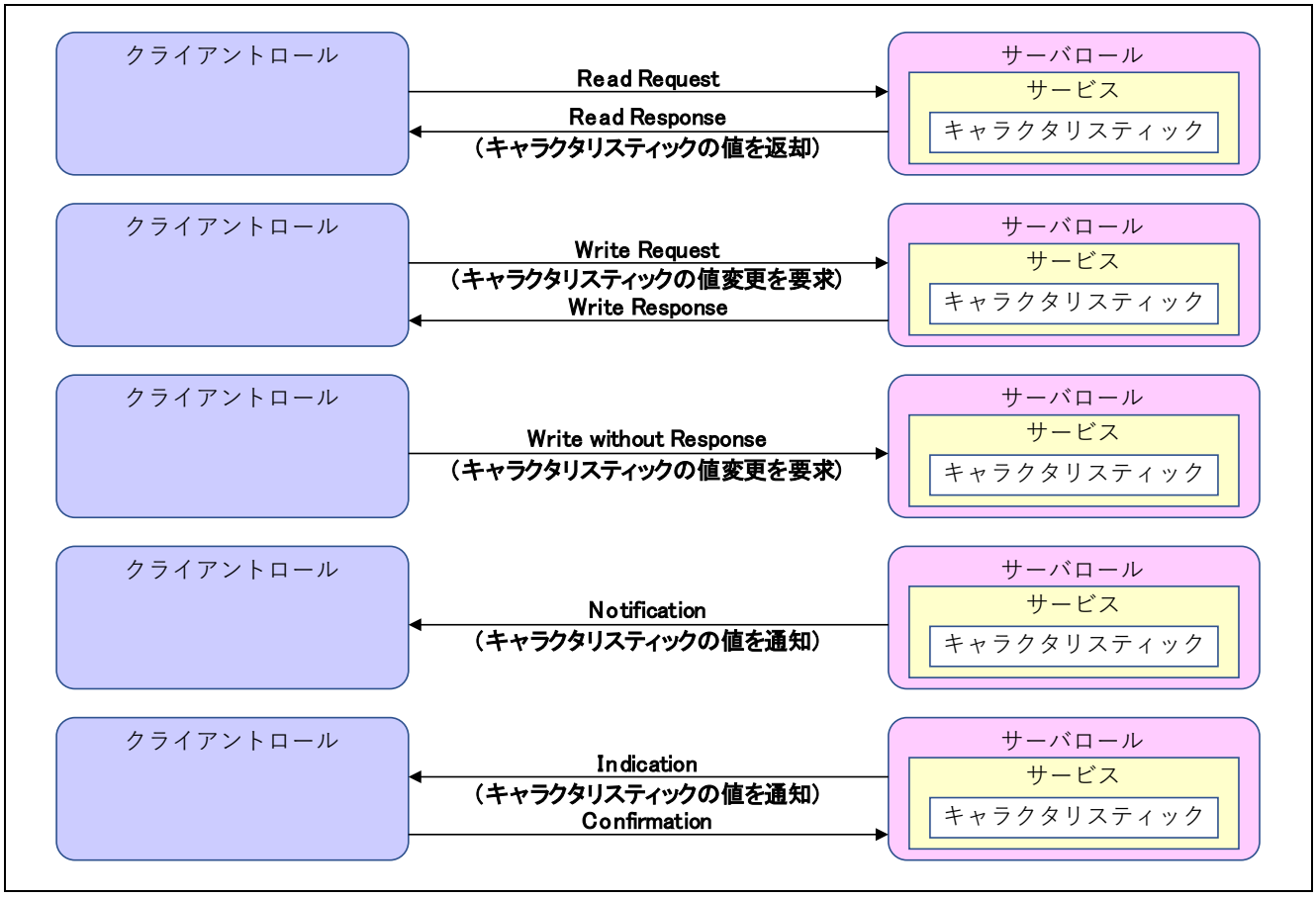


図 4-11　プロファイルによるデータアクセス

(a)　Write Characteristic

クライアントは、サーバが書き込みを許可しているキャラクタリスティックに対して、データを書き込むことができます。サーバはデータを書き込まれると、イベントによってアプリケーションに通知します。

クライアントがサーバにキャラクタリスティック・バリューを書き込むには、次の API を使用します。

1.　RBLE_GATT_Write_Char_Request ：サーバにキャラクタリスティック・バリューを書き込み

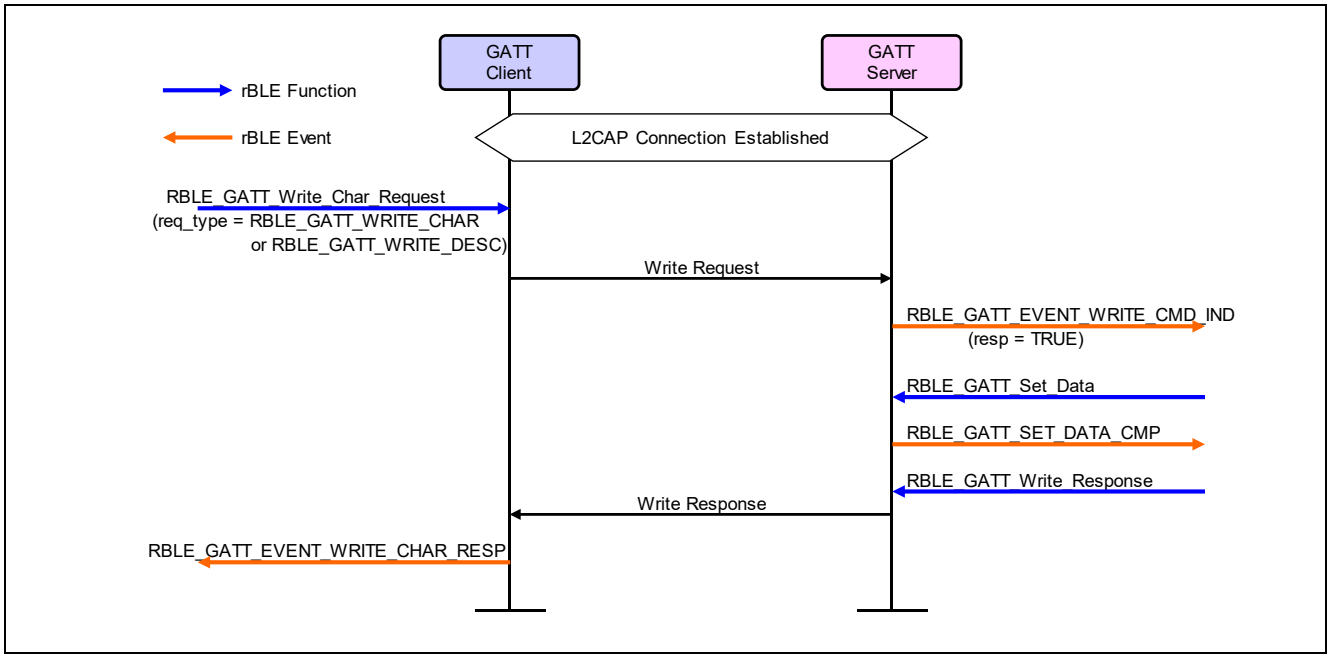


図 4-12　Write Characteristics

※上記は、GATT 機能を使用してカスタムプロファイルを実現するための動作となります。BLE プロトコルスタックが提供するプロファイルを使用する場合は、各プロファイルの API リファレンスマニュアルを参照してください。

(b) Read Characteristic

クライアントは、サーバが読み込みを許可しているキャラクタリスティックから、データを読み出すことができます。サーバはアプリケーションに通知することなく、データを自動的に返します。

サーバロールのアプリケーションはクライアントからの Read Request にリアルタイム応答にてデータを返せません。サーバは読み出しが始まる前に、あらかじめキャラクタリスティックの値を更新する必要があります。詳細は6.6節「読み出しデータの更新」を参照してください。

クライアントがサーバからキャラクタリスティック・バリューを読み込むには、次の API を使用します。

1. RBLE_GATT_Read_Char_Request ：サーバからキャラクタリスティック・バリューを読み込み

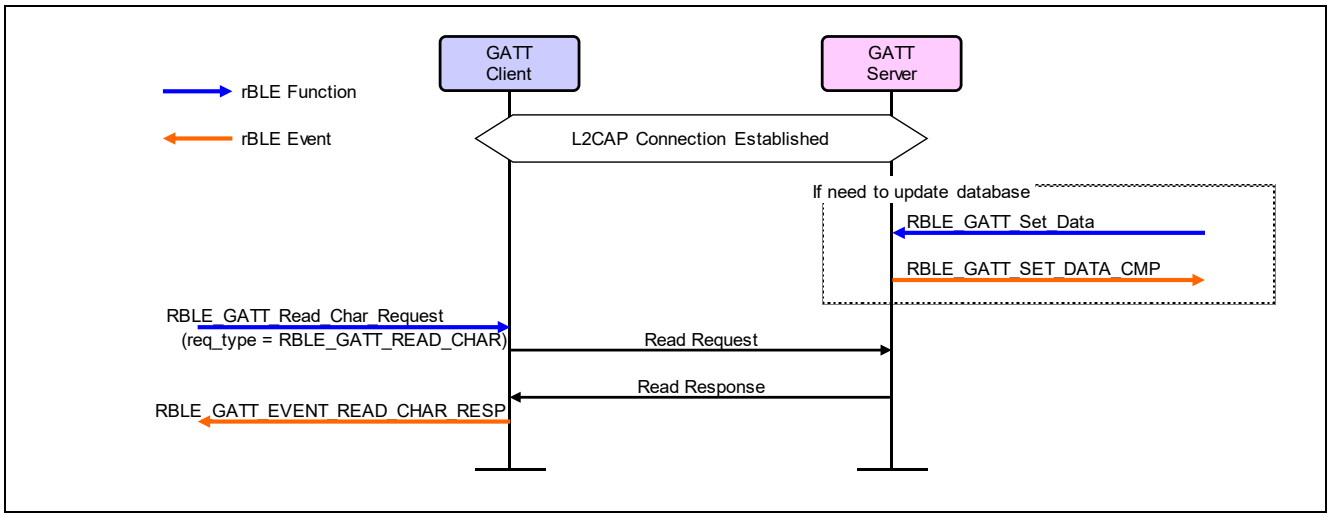


図 4-13 Read Characteristics

※上記は、GATT 機能を使用してカスタムプロファイルを実現するための動作となります。BLE プロトコルスタックが提供するプロファイルを使用する場合は、各プロファイルの API リファレンスマニュアルを参照してください。

(c) Notification Characteristic

サーバは、クライアントから Notification を許可されているならば、クライアントに対してデータを送信することができます。

※Notification の送信後、クライアントはデータを受け取ったことを通知しないため、データの到達は保証されません。データが確実にクライアントに到達したことを確認する必要がある場合は、Indication を使用してください。

サーバがクライアントにキャラクタリスティック・バリューを送信するには、次の API を使用します。

1. RBLE_GATT_Set_Data　：送信するキャラクタリスティック・バリューを更新
2. RBLE_GATT_Notify_Request　：クライアントに対して Notification を送信

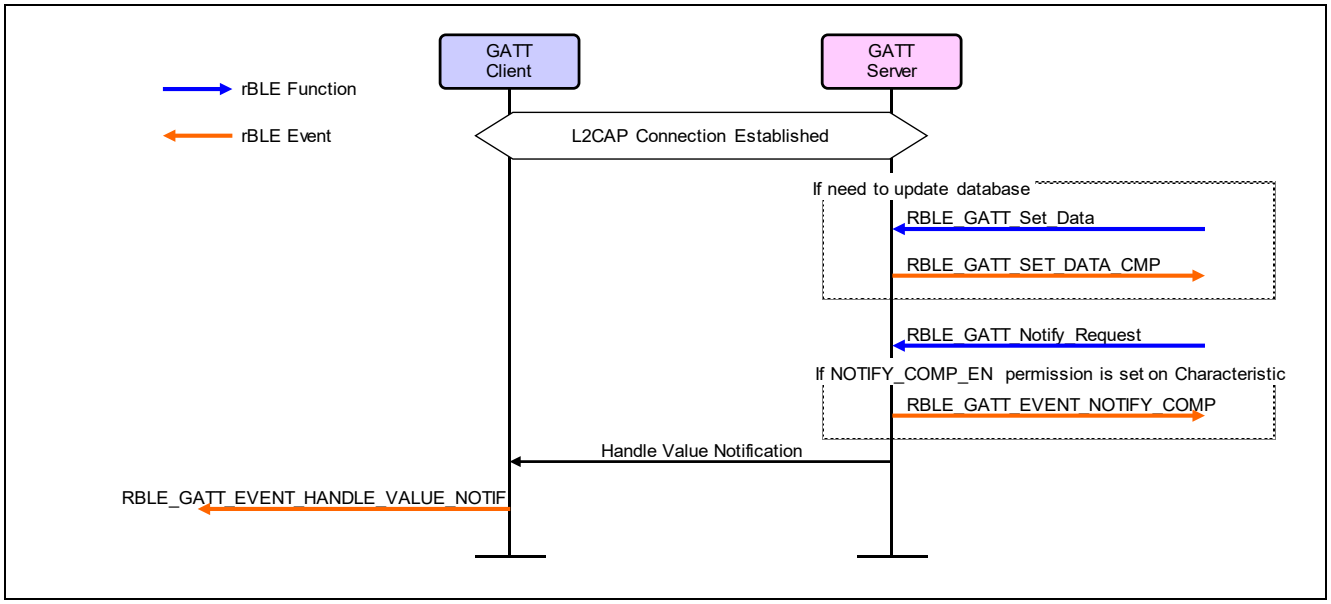


図 4-14　Notification Characteristics

※上記は、GATT 機能を使用してカスタムプロファイルを実現するための動作となります。BLE プロトコルスタックが提供するプロファイルを使用する場合は、各プロファイルの API リファレンスマニュアルを参照してください。

(d) Indication Characteristic

サーバは、クライアントから Indication を許可されているならば、クライアントに対してデータを送信することができます。

※Indication の送信後、クライアントはデータを受け取ったことを通知するため、Indication は Notification と比較してデータの転送レートが低くなります。データの転送レートを上げる必要がある場合は、Notification を使用してください。

サーバがクライアントにキャラクタリスティック・バリューを送信するには、次の API を使用します。

1. RBLE_GATT_Set_Data　：送信するキャラクタリスティック・バリューを更新
2. RBLE_GATT_Indicate_Request　：クライアントに対して Indication を送信

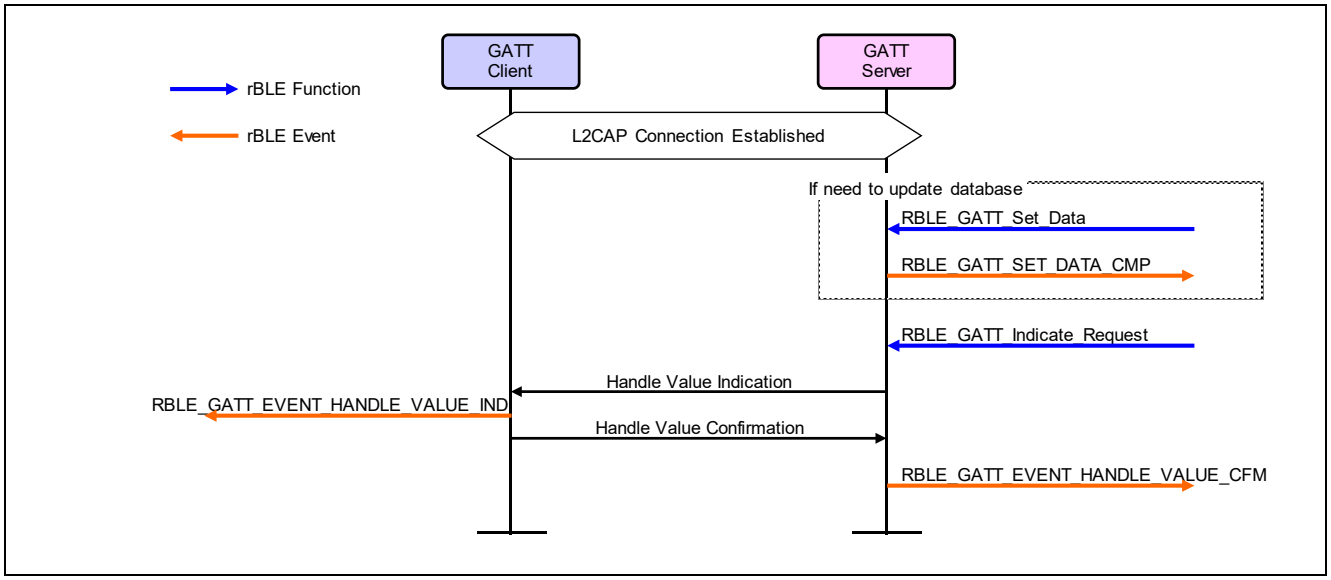


図 4-15　Indication Characteristics

※上記は、GATT 機能を使用してカスタムプロファイルを実現するための動作となります。BLE プロトコルスタックが提供するプロファイルを使用する場合は、各プロファイルの API リファレンスマニュアルを参照してください。

4.4.4　GATT コールバック関数の定義と登録

GATT イベントを受け取るために GATT コールバック関数を定義します。

ファイル：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;
    }
}
```

図 4-16　GATT コールバック関数の定義例

GATT コマンド API を実行する前に RBLE_GATT_Enable にコールバック関数を登録します。

ファイル：rBLE/src/sample_simple/sam/sams.c

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

図 4-17　GATT コールバック関数の登録

5. アプリケーションの動作例

本章ではアプリケーションの動作例として、BLE ソフトウェアに含まれる簡易サンプルプログラムを解説します。

5.1 簡易サンプルプログラムとは

簡易サンプルプログラムは、BLE ソフトウェアに含まれる Embedded 構成のアプリケーションです。

本アプリケーションは以下の動作を実行します。

- プログラムの起動後、Slave Role として接続を確立するためのブロードキャストを開始
- 接続が確立されると、Custom Profile の Server Role を有効化
- LED 制御キャラクタリスティックが更新されると、RL78/G1D 評価ボードの LED4 の点灯状態を制御
- RL78/G1D 評価ボードの SW4 の押下・開放状態を定期的に通知
- 接続が切断されると、ブロードキャストを再開

簡易サンプルプログラムのプロジェクトファイルは、下記の場所に格納されています。

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

簡易サンプルプログラムの使用方法については、下記を参照してください。

- Bluetooth low energy プロトコルスタック サンプルプログラムアプリケーションノート(R01AN1375)
  https://www.renesas.com/document/apn/bluetooth-low-energy-protocol-stack-sample-program
  - 6 章「簡易サンプルプログラムの使用方法」

ペアリングやデータ通信の暗号化、Server Role に加え Client Role を実行する Embedded 構成のサンプルプログラムについては、下記を参照してください。

- Bluetooth low energy プロトコルスタック Embedded 構成サンプルプログラム(R01AN3319)
  https://www.renesas.com/document/scd/bluetooth-low-energy-protocol-stack-embedded-configuration-sample-program

簡易サンプルプログラムの全体シーケンスを以下に示します。シーケンス内の各処理については、次頁以降に詳細を記載します。

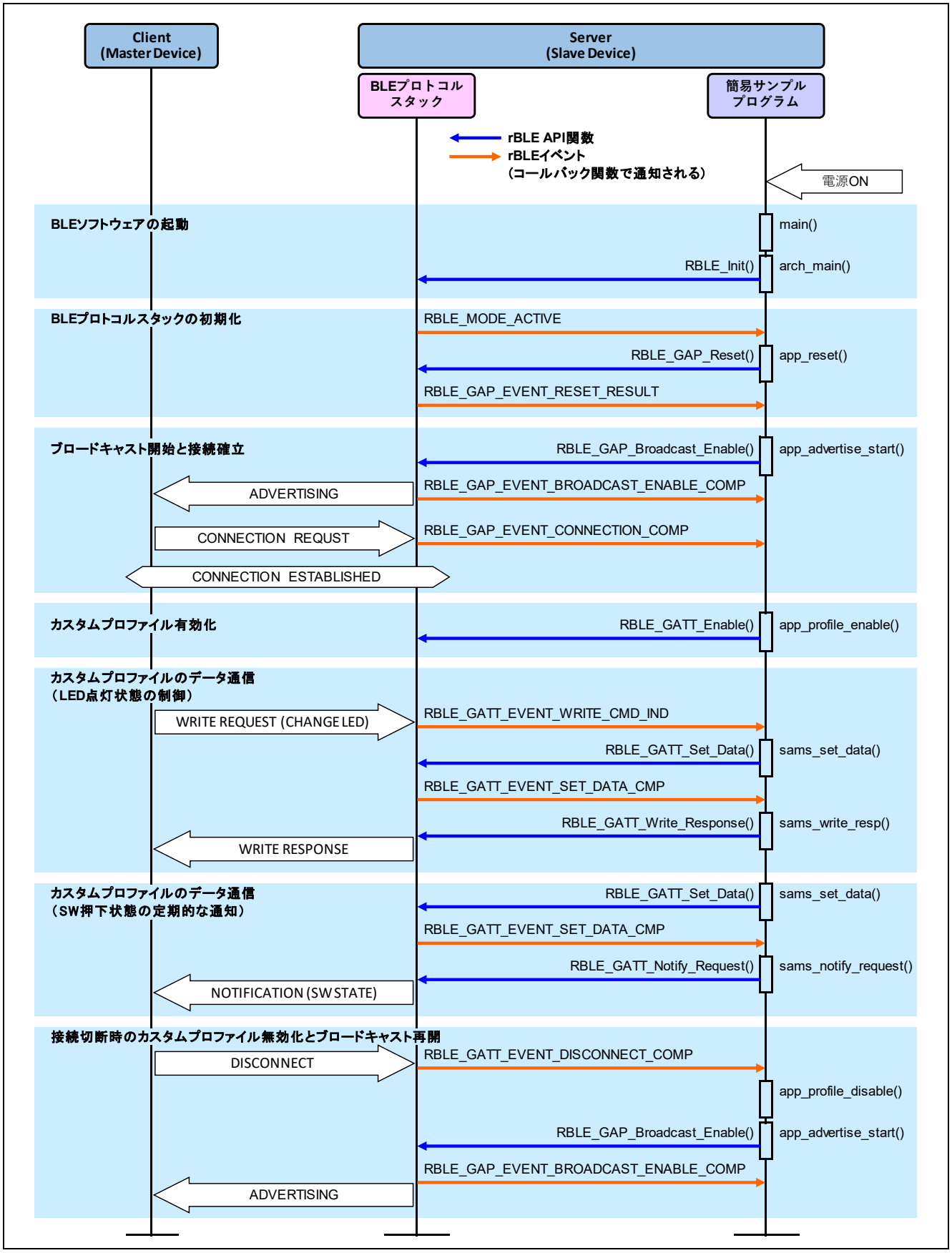


図 5-1　簡易サンプルプログラムの全体シーケンス

5.2 BLE ソフトウェアの起動

BLE ソフトウェアを起動すると、RL78/G1D 周辺機能ドライバ、RWKE、BLE プロトコルスタックの初期化処理を実行します。初期化処理が完了すると、BLE ソフトウェアのシーケンスを実行するためのメインループ処理を実行します。

BLE ソフトウェアの初期化処理とメインループ処理を実行する関数は、下記の通り FW アップデート処理の実行状態によって異なります。

- **main 関数**　：FW アップロード処理を実行状態
- **arch_main_ent 関数**　：FW アップロード処理は未実行状態、通常のデータ通信処理を実行

電源を投入すると、プログラムは **main 関数**から実行されます。main 関数では FW アップデート処理が実行状態であるかを判定し、FW アップデート処理は未実行状態であれば、**arch_main_ent 関数**を実行します。

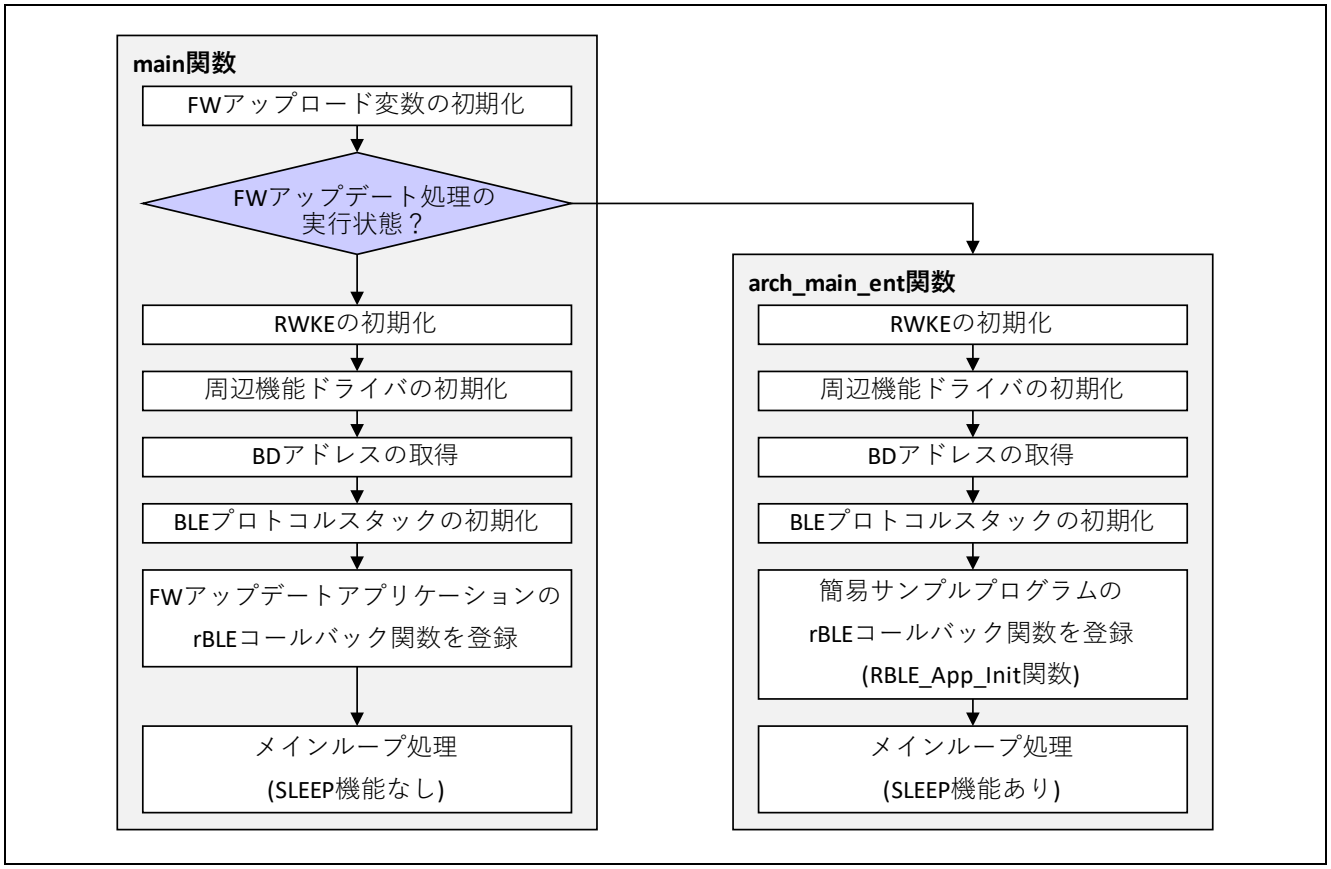


図 5-2　BLE ソフトウェアの起動フローチャート

main 関数

電源を投入すると、プログラムは main 関数から実行されます。main 関数では FW アップデート処理が実行状態であるかを判定します。

FW アップデート処理を実行状態であれば、main 関数で RL78/G1D 周辺機能のドライバ、RWKE、BLE プロトコルスタックの初期化処理と、ファームウェアのアップデート処理を実行します。FW アップデート処理を未実行状態であれば、arch_main 関数に分岐します。

main 関数の実装内容を以下に示します。

```
_MAINCODE void main(void)
{
    ...                                          } FW アップデート変数初期化

    /* during FW update? */
    if( true == check_fw_update() ) {            } FW アップデート処理の実行状態を判定

    ...

        // And loop forever
        for (;;)                                 }
        {                                        }
            // schedule the BLE stack            } FW アップデート処理を実行
            rwble_schedule();                    }
        }                                        }
    }
    else
    {
        /* call arch main */                     }
        arch_main();                             } 通常のデータ通信処理を実行
    }                                            }
}
```

図 5-3　renesas/src/arch/rl78/main.c ― main 関数

arch_main_ent 関数

arch_main_ent 関数を実行すると、RL78/G1D 周辺機能のドライバ、RWKE、BLE プロトコルスタックの初期化処理を実行します。さらに簡易サンプルプログラムのアプリケーションの処理化処理である RBLE_App_Init 関数を実行します。初期化が完了すると、RWKE のスケジューラを繰り返し実行するメインループを実行します。

RWKE と RL78/G1D 周辺機能ドライバの初期化に関する arch_main_ent 関数の内容を以下に示します。

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

    ble_connection_max    = BLE_CONN_MAX;   ・・・・・・・・・・・・(1)最大同時接続数登録

    // Initialize heap memory
    ke_init();                              ・・・・・・・・・・・・(2) RWKE 初期化
    rwble_set_mem();                        ・・・・・・・・・・・・(3)BLE プロトコルスタックメモリ確保

    /*
     ****************************************************************************
     * Platform initialization
     ****************************************************************************
     */
    // init global variables
    variables_init();                       ・・・・・・・・・・・・(4) BLE プロトコルスタック変数初期化

    // init host database
    host_db_init();                         ・・・・・・・・・・・・(5) BLE プロトコルスタック DB 初期化

    // init peak time
    peak_init( 0 );                         ・・・・・・・・・・・・(6)消費電流ピーク通知機能初期化

    //init MCU clocks
    plf_init(CFG_PLF_INIT);                 ・・・・・・・・・・・・(7) MCU 部初期化

    //init LED
    led_init();

    // Initialize the CSI21 module
    spi_init();                             ・・・・・・・・・・・・(8)RF 制御用 SPI インタフェース初期化

    /* Initialize sleep driver */
    sleep_init();                           ・・・・・・・・・・・・(9) Sleep 機能初期化

    /* init dataflash driver */
    dataflash_init();                       ・・・・・・・・・・・・(10)データフラッシュライブラリ初期化

    /* get device address */
    flash_get_bda(&public_addr);            ・・・・・・・・・・・・(11) BD アドレスの決定
```

図 5-4　renesas/src/arch/rl78/arch_main.c ー arch_main_ent 関数(1/3)

(1) 最大同時接続数登録

Master デバイスとして動作時、同時接続する Slave デバイスの最大数を BLE ソフトウェアに登録します。最大同時接続数を変更する場合は、コンパイラの定義マクロ(CFG_CON)で指定して下さい。例えば「CFG_CON=4」と設定すると、同時に 4 台の Slave デバイスと接続することができます。
なお Slave デバイスとして動作時は、本設定に関わらず最大同時接続数は 1 となります。

(2) RWKE 初期化

RWKE を初期化します。

(3) BLE プロトコルスタックメモリ確保

BLE プロトコルスタックが使用するメモリ領域をヒープ領域から確保します。

(4) BLE プロトコルスタック変数初期化

BLE プロトコルスタックで使用されるグローバル変数を初期化します。

(5) BLE プロトコルスタック DB 初期化

BLE プロトコルスタックで使用される GATT データベースを初期化します。

(6) 消費電流ピーク通知機能初期化

消費電流ピーク通知機能を初期化します。
消費電流ピーク通知機能は、RF の送受信動作により消費電流が増大することを通知する機能です。詳細は、ユーザーズマニュアル(R01UW0095)の 7.20.1 項「消費電流ピーク通知機能」を参照して下さい。

(7) MCU 部初期化

MCU のポート機能、動作周波数を初期化します。
動作周波数を変更する場合は、コンパイラオプションの定義マクロで指定して下さい。例えば、動作周波数として 8MHz の内蔵高速オンチップオシレータを使用する場合には、「CLK_HOCO_8MHZ」を設定します。詳細は、ユーザーズマニュアル(R01UW0095)の 6.1.3 項「動作周波数変更」を参照して下さい。

(8) RF 制御用 SPI インタフェース初期化

RF を制御するための SPI インタフェースを初期化します。

(9) Sleep 制御変数初期化

BLE ソフトウェアの Sleep 機能を初期化します。
Sleep 機能は、RWKE に実行すべきイベント、メッセージが無く、Sleep 状態への移行が許可されている場合、MCU を低消費電力状態に遷移させます。詳細は、6.1節「BLEソフトウェアのSleep機能」を参照してください。

(10) データフラッシュライブラリ初期化

データフラッシュライブラリを初期化します。

(11) BD アドレスの取得

この関数は、BLE プロトコルスタックが使用する BD アドレスを決定します。
BD アドレスは Data Flash 領域、Code Flash 顧客固有情報領域、BLE ソフトウェアの CFG_TEST_BDADDR で定義されており、いずれかの BD アドレスを使用します。詳細は、6.3節「デバイスアドレスの保存とアクセス」を参照してください。

BLEプロトコルスタックとアプリケーションの初期化に関するarch_main_ent関数の内容を以下に示します。

```
/*
 ****************************************************************************
 * BLE initializations
 ****************************************************************************
 */
// Disable the BLE core
rwble_disable();                                    ·····(12) RF 部動作禁止

// Initialize RF
rf_init(CFG_RF_INIT);                               ·····(13) RF 部初期化

// input user random seed
input_rand_value(0, userinfo_top);                  ·····(14) 乱数シード初期化

// Initialize BLE stack
rwble_init(&public_addr, CFG_SCA);                  ·····(15) BLE プロトコルスタック初期化

// Enable the BLE core
rwble_enable();                                     ·····(16) RF 部動作許可
...

// rBLE Initialize
RBLE_App_Init();                                    ·····(17) アプリケーション初期化
```

図 5-5　renesas/src/arch/rl78/arch_main.c ー arch_main_ent 関数(2/3)

(12) RF 部動作禁止

RF 部の初期化を実行する前に動作を禁止します。

(13) RF 部初期化

この関数は、RF 部を初期化します。以下の設定を引数で指定することが可能です。
- 外部パワーアンプの使用可否設定
- 内蔵 DC-DC コンバータの使用可否設定
- RF 用スロー・クロック部の設定
- 高速クロックの出力設定

RF 部の設定は、ユーザーズマニュアル(R01UW0095)の 6.1.5 項「RF 部の初期化設定」を参照して下さい。

(14) 乱数シード初期化

標準ライブラリの rand 関数が生成する疑似乱数の初期 seed を設定します。
詳細は、ユーザーズマニュアル(R01UW0095)の 6.1.5.1 項「擬似乱数のシード値設定」を参照して下さい。

(15) BLE プロトコルスタック初期化

BLE プロトコルスタックを初期化し、BD アドレスと SCA（Sleep Clock Accuracy）を設定します。
詳細は、ユーザーズマニュアル(R01UW0095)の 6.1.5 項「RF 部の初期化設定」を参照して下さい。

(16) RF 部動作許可

RF 部の動作を許可します。

(17) アプリケーション初期化

アプリケーションを初期化します。また rBLE の初期化と Generic Access Profile(GAP)、Security Manager(SM)のコールバック関数の登録を実行します。

メインループ処理に関する arch_main_ent 関数の内容を以下に示します。

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

        // schedule the BLE stack
        rwble_schedule();                                   ·····(18) RWKE スケジューラ

        // Checks for sleep have to be done with interrupt disabled
        GLOBAL_INT_DISABLE();                               ·····(19)割り込み禁止
        // Check if the processor clock can be gated
        if ((uint16_t)rwble_sleep() != false)               ·····(20) RF 部停止
        {
            // check CPU can sleep
            if ((uint16_t)sleep_check_enable() != false)    ·····(21) Sleep 許可判定
            {
                #ifndef CONFIG_EMBEDDED
                /* Before CPU enters stop mode, this function must be called */
                if ((uint16_t)wakeup_ready() != false)      ·····(22)シリアル通信停止
                #endif // #ifndef CONFIG_EMBEDDED
                {
                    // Wait for interrupt
                    WFI();                                  ·····(23) MCU 部停止

                    #ifndef CONFIG_EMBEDDED
                    wakeup_finish();                        ·····(24)シリアル通信復帰
                    #endif // #ifndef CONFIG_EMBEDDED
                }
            }
        }
        // Checks for sleep have to be done with interrupt disabled
        GLOBAL_INT_RESTORE();                               ·····(25)割り込み許可

        sleep_load_data();                                  ·····(26) RF 部復帰
    }
}
```

図 5-6　renesas/src/arch/rl78/arch_main.c ー arch_mein_ent 関数(3/3)

(18) RWKE スケジューラ

RWKE 機能を実行します。イベント機能、メッセージ機能が空となるまで実行され続けます。

(19) 割り込み禁止

RF 部を低消費電力状態に遷移させる前に割り込みを禁止します。

(20) RF 部停止

RWKE と BLE プロトコルスタックの動作状態を確認し、RF 部を低消費電力モードである SLEEP モードまたは DEEP_SLEEP モードに遷移させます。また MCU を STOP モードへ遷移させることが可能かどうかを返り値により通知します。

(21) Sleep 許可判定

アプリケーションが Sleep を許可しているかを確認します。アプリケーションは本関数の返り値を動的に変更することで、MCU 部の STOP モードへの遷移を制御できます。

(22) シリアル通信停止

BLE ソフトウェアが Modem 構成の場合、ホスト MCU とのシリアル通信を停止させます。

(23) MCU 部停止

MCU 部を STOP モードへ遷移させます。STOP モードはマスクされていない割り込みの発生で解除されます。

(24) シリアル通信復帰

BLE ソフトウェアが Modem 構成の場合、ホスト MCU とのシリアル通信を復帰させます。

(25) 割り込み許可

割り込みを許可します。

(26) RF 部復帰

RF 部を低消費電力モードモードから復帰させます。

5.3　BLE プロトコルスタックの初期化

BLE プロトコルスタックの初期化処理について示します。

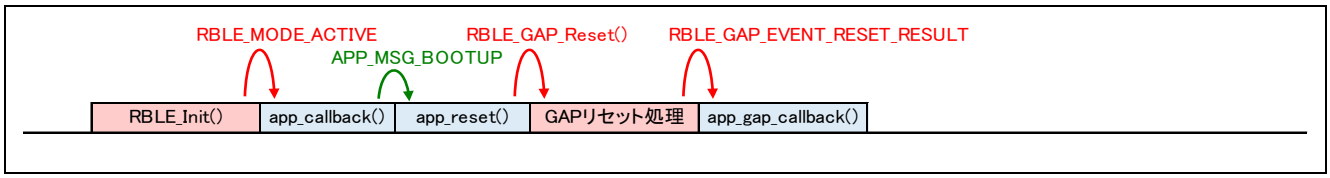


図 5-7　BLE プロトコルスタックの初期化

RBLE_Init 関数

arch_main_ent 関数から実行される RBLE_App_Init 関数は、**RBLE_Init 関数**を実行して rBLE を初期化し、rBLE コールバック関数として **app_callback 関数**を登録します。初期化が完了すると、rBLE API を通して BLE プロトコルスタック機能の利用が可能となります。

rBLE は初期化が完了すると、コールバック関数である app_callback 関数を実行して **RBLE_MODE_ACTIVE** 状態への遷移完了を通知します。

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

図 5-8　rBLE/src/sample_simple/rble_sample_app_peripheral.c ― RBLE_App_Init 関数

app_callback 関数は RBLE_MODE_ACTIVE が通知されると **APP_MSG_BOOTUP** メッセージを送信します。

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

図 5-9　rBLE/src/sample_simple/rble_sample_app_peripheral.c ― app_callback 関数

APP_MSG_BOOTUP メッセージが送信されると、APP_MSG_BOOTUP のメッセージハンドラである **app_reset 関数**が実行されます。

app_reset 関数は **RBLE_GAP_Reset 関数**を実行して GAP 機能と SM 機能を初期化します。また GAP コールバック関数として **app_gap_callback 関数**、SM コールバック関数として **app_sm_callback 関数**を登録します。

```
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);
    ...
}
```

図 5-10　rBLE/src/sample_simple/rble_sample_app_peripheral.c ― app_reset 関数

5.4　ブロードキャスト開始と接続確立

ブロードキャスト開始と接続確立の処理について示します。

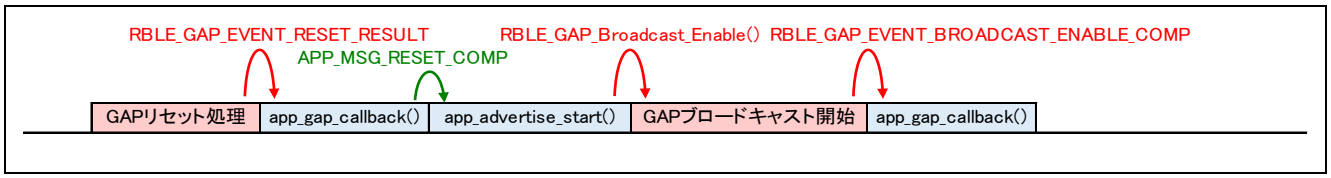


図 5-11　ブロードキャスト開始

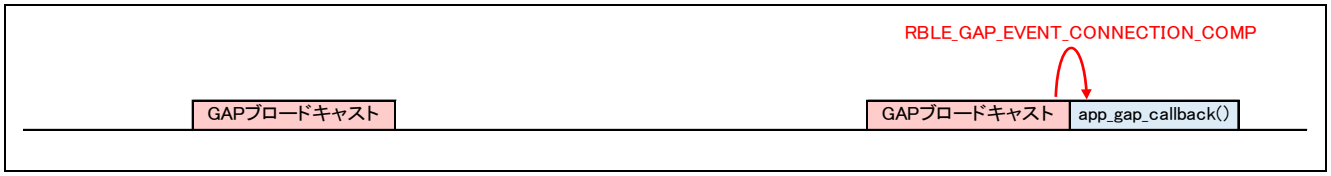


図 5-12　接続確立

RBLE_GAP_Broadcast_Enable 関数

GAP 機能は初期化が完了すると、GAP コールバック関数である **app_gap_callback 関数**を実行し、**RBLE_GAP_EVENT_RESET_RESULT** イベントを通知します。app_gap_callback 関数は本イベントが通知されると、**APP_MSG_RESET_COMP** メッセージを送信します。

なお GAP 機能は接続を確立すると、**RBLE_GAP_EVENT_CONNECTION_COMP** イベントを通知します。app_gap_callback 関数は本イベントが通知されると、**APP_MSG_CONNECTED** メッセージを送信します。

```
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;
    ...
}
```

図 5-13　rBLE/src/sample_simple/rble_sample_app_peripheral.c ― app_gap_callback 関数

APP_MSG_RESET_COMP メッセージが送信されると、APP_MSG_RESET_COMP のメッセージハンドラである **app_advertise_start 関数**が実行されます。

app_advertise_start 関数は **RBLE_GAP_Broadcast_Enable 関数**を実行して、Slave 接続のためのブロードキャストを開始します。

```
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);
    ...
}
```

図 5-14　rBLE/src/sample_simple/rble_sample_app_peripheral.c ― app_advertise_start 関数

5.5　カスタムプロファイル有効化

カスタムプロファイル有効化の処理について示します。

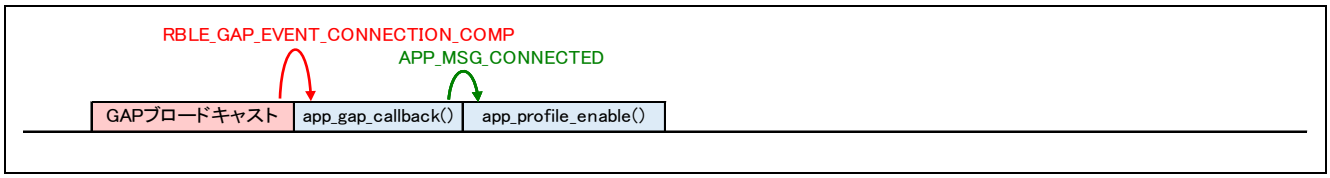


図 5-15　カスタムプロファイル有効化

RBLE_GATT_Enable 関数

APP_MSG_CONNECTED メッセージが送信されると、APP_MSG_CONNECTED のメッセージハンドラである **app_profile_enable 関数**が実行されます。

app_profile_enable 関数は **SAMPLE_Server_Enable 関数**を実行して、カスタムプロファイルの有効化します。またカスタムプロファイルからの通知を受け取るために、コールバック関数として **app_sams_callback 関数**を指定します。

```
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);
    ...
}
```

図 5-16　rBLE/src/sample_simple/rble_sample_app_peripheral.c ― app_profile_enable 関数

SAMPLE_Server_Enable 関数は **RBLE_GATT_Enable 関数**を実行して、GATT 機能の初期化し、GATT コールバック関数として **sams_gatt_callback 関数**を登録します。

```
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);
    ...
}
```

図 5-17　rBLE/src/sample_simple/sam/sams.c ― app_profile_enable 関数

5.6　カスタムプロファイルのデータ通信

接続の確立後、簡易サンプルプログラムは以下の動作を実現します。

- LED 制御キャラクタリスティックが更新されると、RL78/G1D 評価ボードの LED4 の点灯状態を制御
- RL78/G1D 評価ボードの SW4 の押下・開放状態を定期的に通知

5.6.1　LED 点灯状態の制御

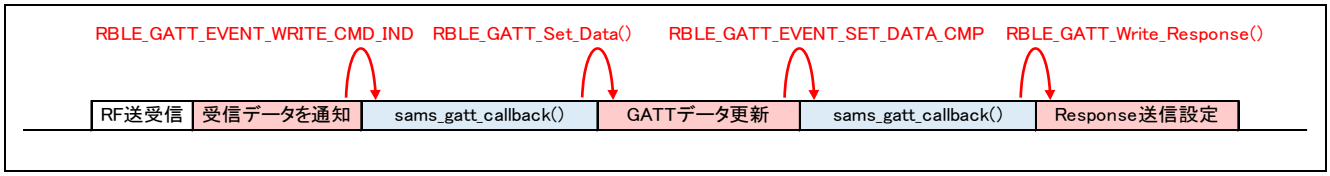


図 5-18　キャラクタリスティック書き込みへの Write Response 送信

簡易サンプルプログラムに実装されたカスタムサービスには、LED の点灯状態を制御するためのキャラクタリスティックがあります。

接続を確立したクライアントデバイスが本キャラクタリスティックに書き込みを行うと、rBLE は GATT コールバック関数である **sams_gatt_callback** 関数を実行して、**RBLE_GATT_EVENT_WRITE_CMD_IND** イベントを通知します。これにより、キャラクタリスティックが書き込まれたことを通知します。

```
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:
        ...
    }
}
```

図 5-19　rBLE/src/sample_simple/sam/sams.c ― sams_gatt_callback 関数

RBLE_GATT_Set_Data 関数

カスタムプロファイルはキャラクタリスティックの書き込みを通知されると、**RBLE_GATT_Set_Data 関数**を実行します。これにより、BLE プロトコルスタックが保持するキャラクタリスティックをクライアントデバイスから渡されたデータに更新します。

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

図 5-20　rBLE/src/sample_simple/sam/sams.c ― sams_set_data 関数

RBLE_GATT_Write_Response 関数

rBLE はキャラクタリスティックの更新が完了すると、sams_gatt_callback 関数を実行して **RBLE_GATT_EVENT_SET_DATA_CMP** イベントを通知します。

本イベントが通知されると、カスタムプロファイルは **RBLE_GATT_Write_Response 関数**を実行します。これにより、BLE プロトコルスタックはクライアントデバイスに Write Response を送信します。

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

図 5-21　rBLE/src/sample_simple/sam/sams.c ― sams_write_resp 関数

またカスタムプロファイルはコールバック関数である **app_sams_callback 関数**を実行し、キャラクタリスティックの更新を通知します。これによりアプリケーションは **led_onoff_set 関数**を実行して、LED の点灯・消灯を制御します。

```
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;

    ...
    }
}
```

図 5-22　rBLE/src/sample_simple/rble_sample_app_peripheral.c ― app_sams_callback 関数

5.6.2　SW 押下状態の定期的な通知

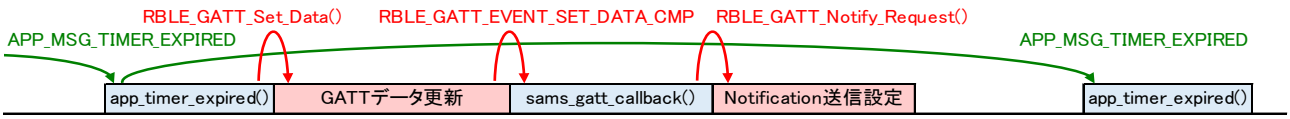


図 5-23　定期的な Notification 送信

簡易サンプルプログラムに実装されたカスタムサービスには、SW の押下状態を通知するためのキャラクタリスティックがあります。本キャラクタリスティックは、Notification 送信によってクライアントデバイスに通知されます。

Notification の送信開始のためには、まずクライアントデバイスからの Notification 送信許可が必要です。Notification 送信が許可されると、カスタムプロファイルは **app_sams_callback 関数**を実行し、アプリケーションに通知します。これによりアプリケーションは RWKE のタイマ機能を開始します。

```
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;

    ...
    }
}
```

図 5-24　rBLE/src/sample_simple/rble_sample_app_peripheral.c ― app_sams_callback 関数

アプリケーションが指定した時間が経過すると、**APP_MSG_TIMER_EXPIRED** イベントが送信され、APP_MSG_TIMER_EXPIRED のイベントハンドラである **app_timer_expired 関数**が実行されます。

app_timer_expired 関数は、カスタムプロファイルの **SAMPLE_Server_Send_Switch_State 関数**を実行して、Notification を送信します。また次の周期に Notification を送信するため、タイマ機能を再度開始します。

```
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);
}
```

図 5-25　rBLE/src/sample_simple/rble_sample_app_peripheral.c ― app_timer_expired 関数

RBLE_GATT_Set_Data 関数

カスタムプロファイルは SAMPLE_Server_Send_Switch_State 関数が実行されると、**RBLE_GATT_Set_Data 関数**を実行します。これにより BLE プロトコルスタックが保持するキャラクタリスティックを、アプリケーションから渡されたデータに更新します。

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

図 5-26　rBLE/src/sample_simple/sam/sams.c ー sams_set_data 関数

RBLE_GATT_Notify_Request 関数

rBLE はキャラクタリスティックの更新が完了すると、sams_gatt_callback 関数を実行して **RBLE_GATT_EVENT_SET_DATA_CMP** イベントを通知します。

本イベントが通知されると、カスタムプロファイルは **RBLE_GATT_Notify_Request 関数**を実行します。これにより、BLE プロトコルスタックはクライアントデバイスに Notification を送信します。

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

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

図 5-27　rBLE/src/sample_simple/sam/sams.c ー sams_notify_request 関数

5.7　カスタムプロファイル無効化とブロードキャスト再開

接続切断時のカスタムプロファイル無効化とブロードキャスト再開の処理を示します。

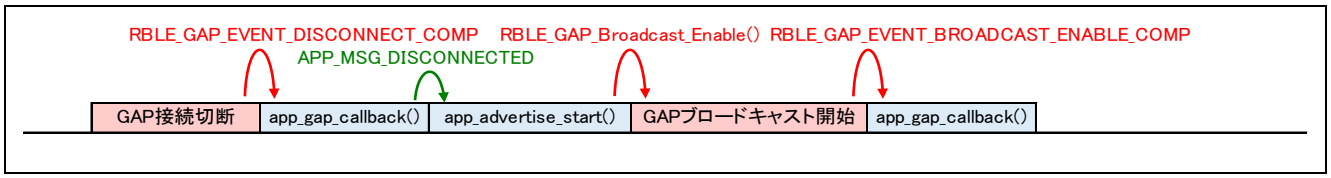


図 5-28　ブロードキャスト再開

GAP 機能は接続が切断されると、GAP コールバック関数である **app_gap_callback 関数**を実行して、**RBLE_GAP_EVENT_DISCONNECT_COMP** イベントを通知します。

app_gap_callback 関数は本イベントが通知されると **APP_MSG_DISCONNECTED** メッセージを送信します。

```
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;
    ...
}
```

図 5-29　rBLE/src/sample_simple/rble_sample_app_peripheral.c ー app_gap_callback 関数

APP_MSG_DISCONNECTED メッセージが送信されると、イベントハンドラである **app_profile_disable 関数**が実行されます。

app_profile_disable 関数は **SAMPLE_Server_Disable 関数**を実行し、カスタムプロファイルを無効化します。

```
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);
}
```

図 5-30　rBLE/src/sample_simple/rble_sample_app_peripheral.c ー app_profile_disable 関数

SAMPLE_Server_Disable 関数はカスタムプロファイルの無効化が完了すると、コールバック関数である **app_sams_callback 関数**を実行して、**SAMPLE_SERVER_EVENT_DISABLE_COMP** イベントを通知します。

これにより、app_sams_callback 関数は **APP_MSG_PROFILE_DISABLED** イベントを送信します。

```
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;
    ...
    }
}
```

図 5-31　rBLE/src/sample_simple/rble_sample_app_peripheral.c ー app_sams_callback 関数

RBLE_GAP_Broadcast_Enable 関数

APP_MSG_PROFILE_DISABLED メッセージが送信されると、APP_MSG_PROFILE_DISABLED のメッセージハンドラである **app_advertise_start 関数**が実行されます。

app_advertise_start 関数は **RBLE_GAP_Broadcast_Enable 関数**を実行し、ブロードキャストが再開されます。

```
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(
       RBLE_GAP_GEN_DISCOVERABLE, RBLE_GAP_UND_CONNECTABLE, &app_advertise_param);
    ...
}
```

図 5-32　rBLE/src/sample_simple/rble_sample_app_peripheral.c ー app_advertise_start 関数

6. 開発のヒント

6.1 BLE ソフトウェアの Sleep 機能

BLE ソフトウェアは、消費電力を低減するための Sleep 機能を提供します。Sleep 機能は、RWKE が実行すべきイベント、メッセージが無く、Sleep 状態への移行が許可されている場合、MCU を低消費電力状態に遷移させます。Sleep 機能の詳細は、ユーザーズマニュアル(R01UW0095)の 7.20.2 項「Sleep 機能」を参照してください。

また BLE ソフトウェアには、コンソール上で rBLE API のコマンドの実行とイベントを確認することができるコンソール入出力サンプルプログラムが含まれます。コンソール入出力サンプルプログラムの使用方法は、サンプルプログラムアプリケーションノート(R01AN1375)の 5 章「コンソール入出力サンプルプログラムの使用方法」を参照してください。

Embedded 構成のコンソール入出力サンプルプログラムは、デフォルトでは Sleep 機能を使用しない設定となっています。ここでは Embedded 構成において Sleep 機能を有効にする方法を説明します。

ケース A：UART を使用しない

sleep_cont_ent 関数が常に TRUE を返すように変更します。

ファイル：renesas/src/arch/rl78/arch_main.c

```
/* sleep control */
bool sleep_cont_ent(void)
{
    :
    (常に TRUE を返す処理を追加する)
}
```

図 6-1 sleep_cont_ent 関数の変更

ケース B：UART を使用し続ける

1) console_can_sleep 関数を下記のように変更します。

ファイル：rBLE/src/sample_app/Console.c

```
/* sleep control */
変更前：
bool console_can_sleep(void)
{
    return( false );
}

変更後：
bool console_can_sleep(void)
{
    return( !Send_Flg );
}
```

図 6-2 console_can_sleep 関数の変更

2) “uart.c”ファイルの serial_init 関数で設定されるボーレートを 4800bps に変更します。

4800bps よりも高いボーレートを使用する場合は、3 線式か 2 線分岐式の UART を選択して Sleep モードから起床する必要があります。これらのシリアル通信の詳細は、ユーザーズマニュアル(R01UW0095)の 5.4 節「Modem 構成時のシリアル通信」を参照して下さい。

ファイル：renesas/src/driver/uart/uart.c

変更前：

```
    /* 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);
```

変更後：

```
    /* 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);
```

図 6-3 ボーレートの変更

3)serial_init 関数で設定されるストップフラグを変更します。

ファイル：renesas/src/driver/uart/uart.c

変更前：

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

変更後：

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

図 6-4 ストップフラグの変更

4) wakeup_ready 関数と wakeup_finish 関数の実行を有効にします。

ファイル：renesas/src/arch/rl78/arch_main.c

変更前：

```
    #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
    }
```

変更後：

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

図 6-5 wakeup 関数実行の有効化

5) wakeup_init_ent 関数呼出しを有効にします。

ファイル：renesas/src/arch/rl78/arch_main.c

変更前：

```
_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]
};
```

変更後：

```
_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]
};
```

図 6-6 wakeup_init_ent 関数の有効化

6) wakeup_init 関数の処理を有効にします。

ファイル：renesas/src/arch/rl78/main.c

変更前：

```
_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
}
```

変更後：

```
_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))();
    }
}
```

図 6-7 wakeup_init_ent 関数の有効化

6.2 Bluetooth デバイスアドレス

ルネサス BLE ソフトウェアは 3 種類の Bluetooth デバイスアドレスをサポートしています。Bluetooth デバイスアドレスの特徴を理解し、製品ではどの Bluetooth デバイスアドレスを使用するかを決定します。

Bluetooth デバイスアドレスの詳細は、ユーザーズマニュアル(R01UW0095)の 7.2.4 項「Bluetooth デバイスアドレス」を参照してください。

Public Device Address

IEEE によって割り当てられるユニークなデバイスアドレスです。一般的な MAC アドレスと同じ構造です。

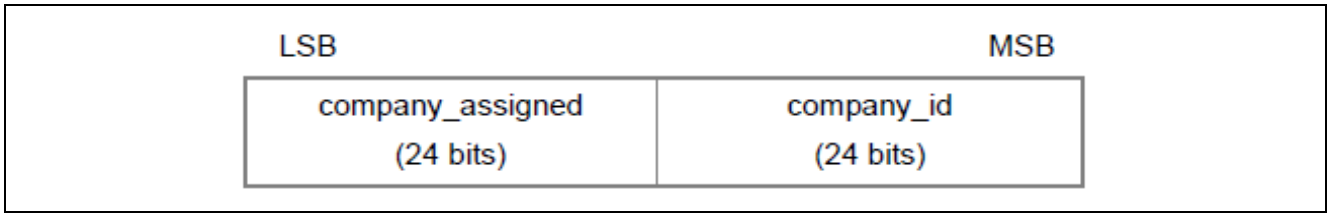


図 6-8　Public Device Address のフォーマット

Static Device Address

ランダムな固定のアドレスです。デバイスを識別するために Public Device Address と同等に扱われます。

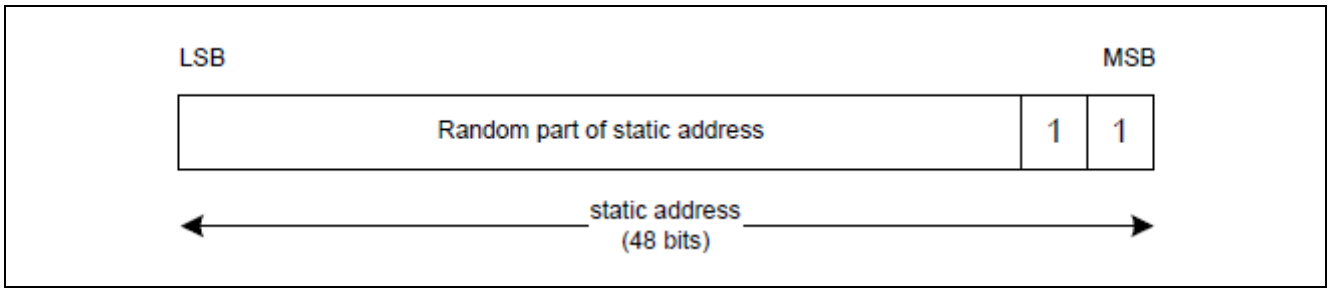


図 6-9　Static Device Address のフォーマット

Resolvable Private Address

ランダムなアドレスです。デバイスの識別のために IRK（Identity Resolving Key）を使用します。

15 分ごとに変更することが推奨されています。

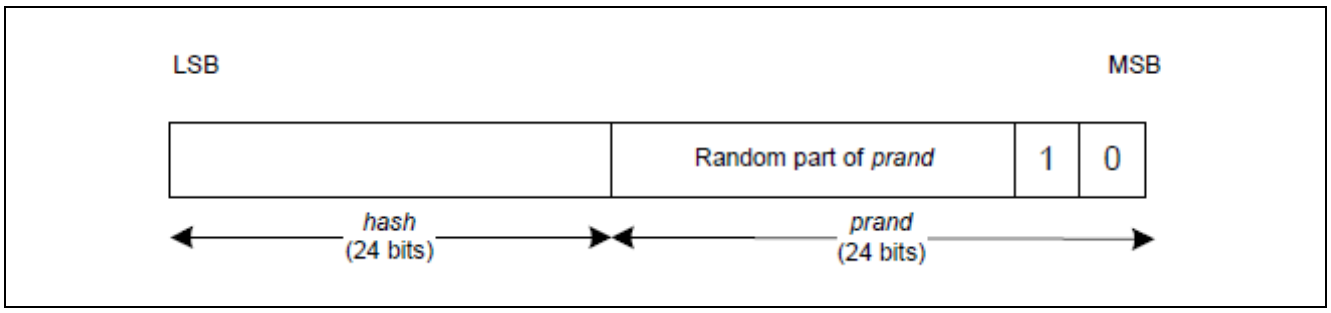


図 6-10　Resolvable Private Address のフォーマット

6.3　デバイスアドレスの保存とアクセス

Public Device Address

Public Device Address は、データフラッシュ領域、顧客固有情報領域、CFG_TEST_BDADDR 定義マクロの 3 つの場所に保存することが可能です。

- データフラッシュ領域　　: データフラッシュライブラリが「データ ID=1」として管理
- 顧客固有情報領域　　: RL78/G1D のコードフラッシュの最終ブロックに保存
- CFG_TEST_BDADDR マクロ　　: "config.h"ヘッダファイルで定義

製品出荷時には顧客固有情報領域にデフォルトのデバイスアドレスを書き込み、製品運用時にはデータフラッシュ領域に新しいデバイスアドレスを書き込むことで Public Device Address を変更することも可能です。なお CFG_TEST_BDADDR マクロで定義するデバイスアドレスは開発作業時のみご使用ください。

図 6-11は、BLE ソフトウェア初期化時における Public Device Address を決定するフローを示します。

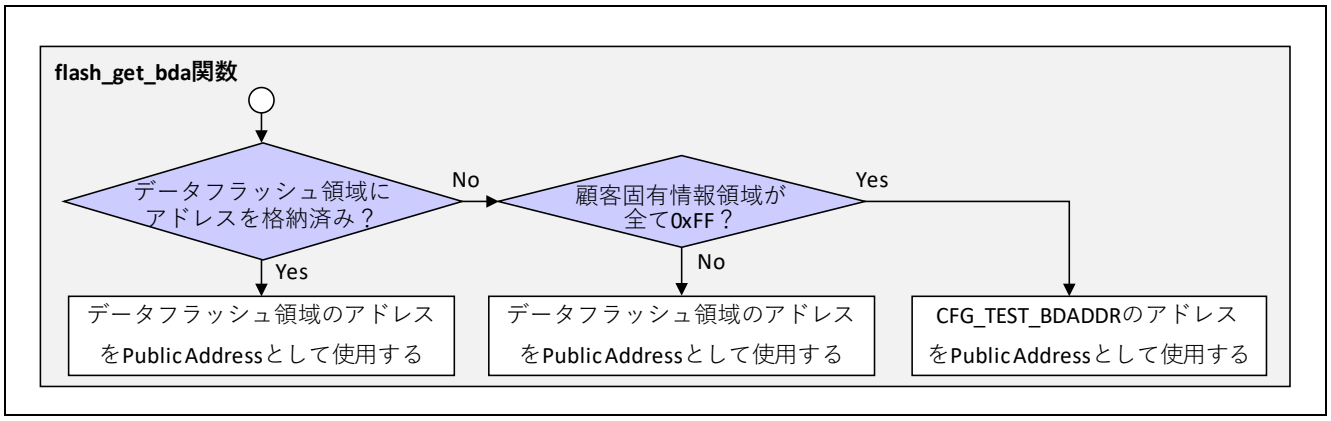


図 6-11　Public Device Address の決定フロー

アプリケーションは rBLE API を使用してデータフラッシュ領域に Public Device Address を書き込むことができます。Public Device Address をデータフラッシュ領域に書き込みは、図 6-12に示すシーケンスを実行してください。書き込んだデバイスアドレスは、RL78/G1D の再起動で BLE ソフトウェアに反映されます。

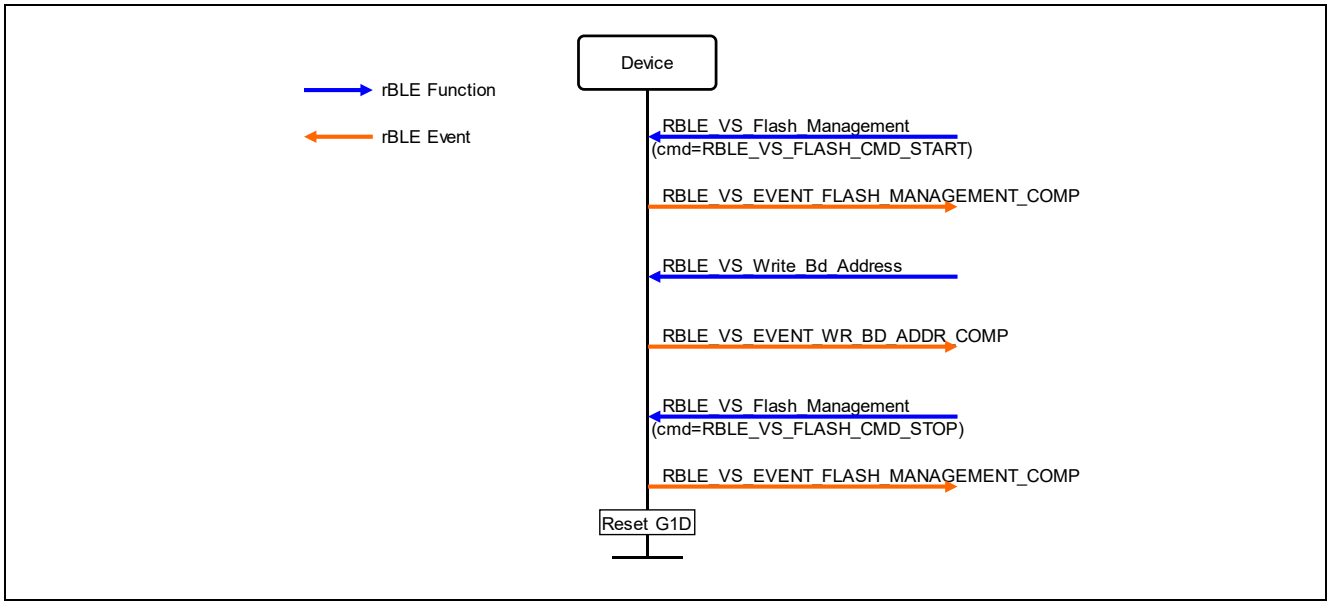


図 6-12　Public Device Address の書き込み方法

Static Device Address

Static Device Address として使用するデバイス固有のランダム値を、データフラッシュ領域に保存することで、常に同じ値を Static Device Address として利用することができます。

デバイス固有の Static Device Address をデータフラッシュ領域に保存するには、図 6-13と図 6-14のように、データフラッシュドライバのディスクリプタに Static Device Address を保存するためのデータ ID とデータサイズを定義します。

ファイル：renesas/src/driver/dataflash/eel_descriptor_t02.h

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

図 6-13　データ ID の定義例

ファイル：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         */ ¥
};
```

図 6-14　ディスクリプタの定義例

ディスクリプタを定義することで、アプリケーションは rBLE API を利用してデータフラッシュ領域への Static Device Address の書き込みと読み出しができます。

Static Device Address をデータフラッシュ領域に書き込むには、図 6-15に示すシーケンスを実行します。

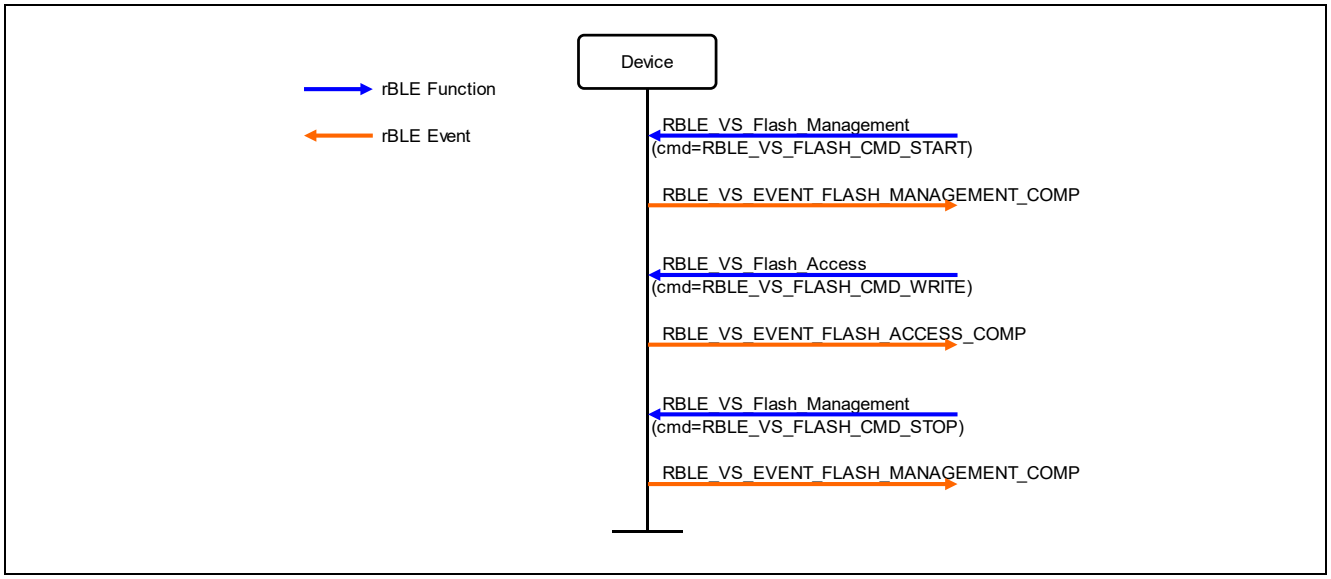


図 6-15　Static Device Address の書き込み方法

データフラッシュ領域に書き込まれた Static Device Address を読み込むには、図 6-16に示すシーケンスを実行します。

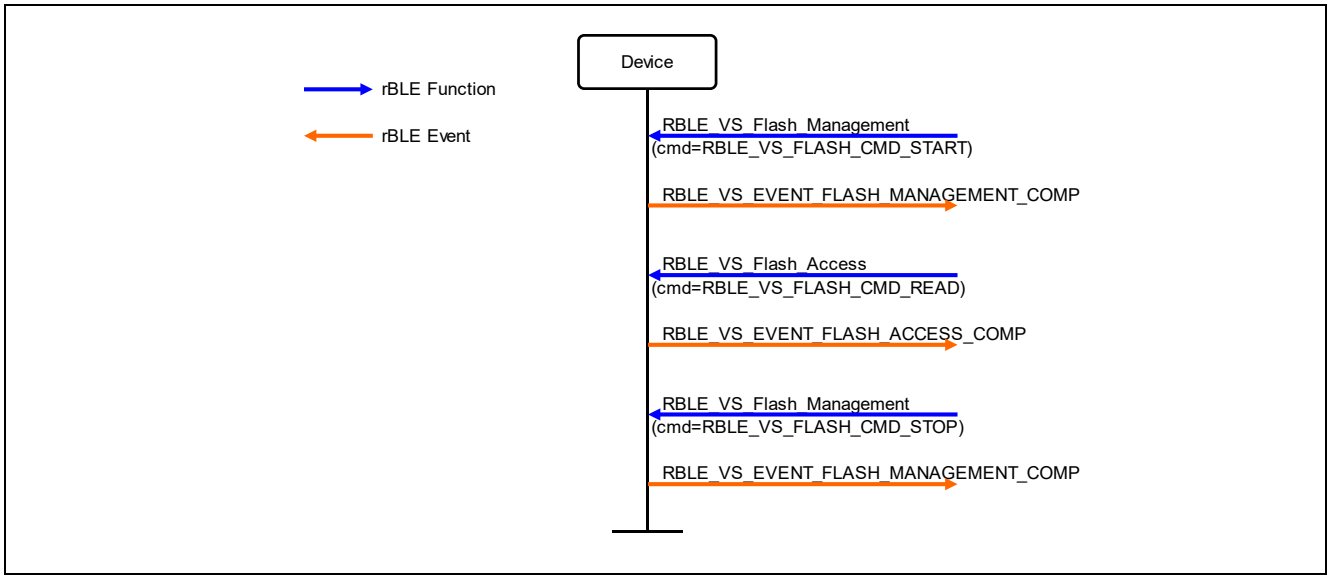


図 6-16　Static Device Address の読み出し方法

6.4　デバイスアドレス・タイプ別のブロードキャスト

デバイスアドレスの種類により、ブロードキャストを開始するためのシーケンスが異なります。デバイスアドレスの種類毎のブロードキャストを開始するためのシーケンスを示します。

Public Device Address

Public Device Address でブロードキャストする場合のシーケンスを図 6-17に示します。

ブロードキャストを開始するため、own_addr_type に RBLE_ADDR_PUBLIC を指定して RBLE_GAP_Broadcast_Enable 関数をコールします。

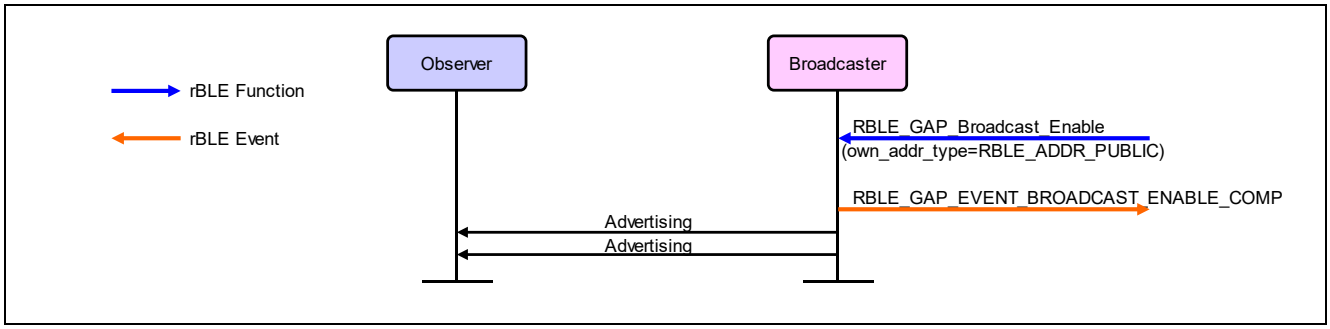


図 6-17　Public Device Address でのブロードキャスト

Static Device Address

Static Device Address でブロードキャストする場合のシーケンスを図 6-18に示します。

アプリケーションが準備した Static Device Address を設定するため、RBLE_GAP_Set_Random_Address 関数をコールしてします。

ブロードキャストを開始するため、own_addr_type に RBLE_ADDR_RAND を指定して RBLE_GAP_Broadcast_Enable 関数をコールします。

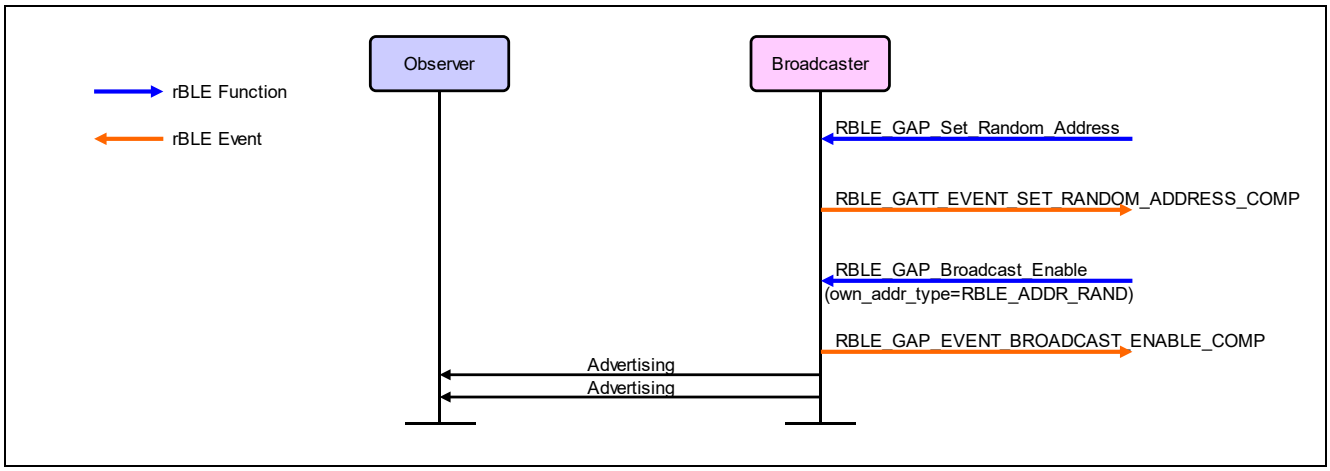


図 6-18　Static Device Address でのブロードキャスト

Resolvable Private Address

Resolvable Private Address でブロードキャストする場合のシーケンスを図 6-19に示します。

アプリケーションで準備した IRK を設定するため、Key_code に RBLE_SMP_KDIST_IDKEY を指定して RBLE_SM_Set_Key をコールします。

Resolvable Private Address を生成するため、priv_flag に RBLE_PH_PRIV_ENABLE または RBLE_BCST_PRIV_ENABLE を指定して RBLE_GAP_Set_Privacy_Feature 関数をコールします。

ブロードキャストを開始するため、own_addr_type に RBLE_ADDR_RAND を指定して RBLE_GAP_Broadcast_Enable 関数をコールします。

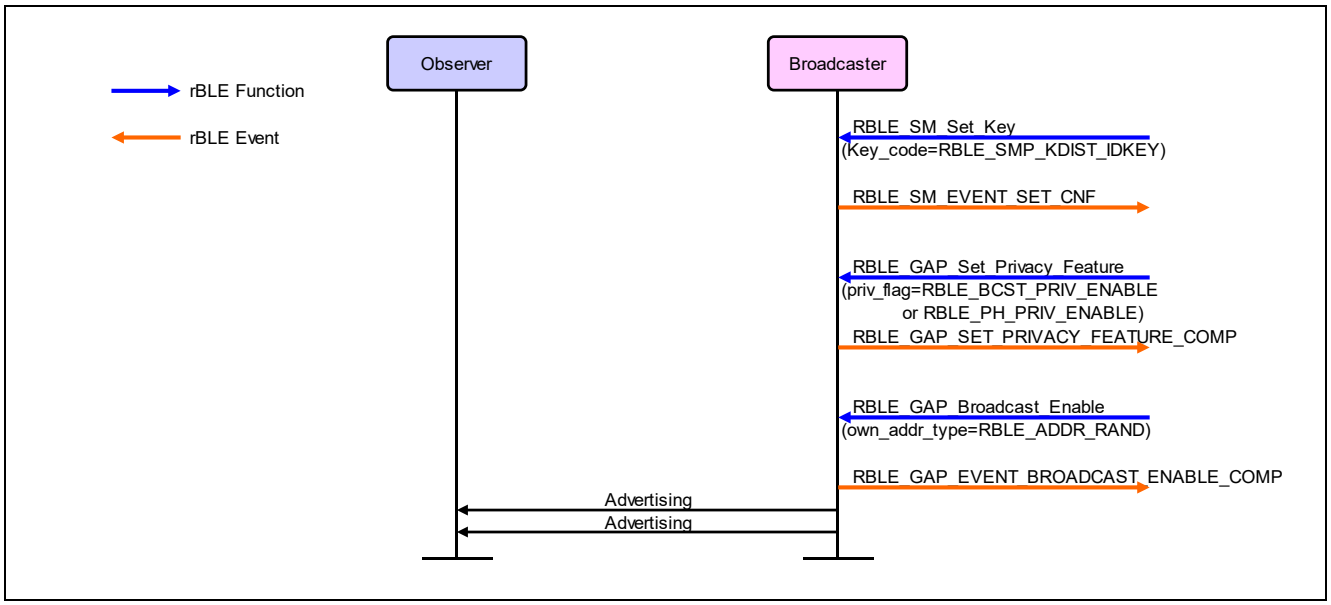


図 6-19　Resolvable Private Address でのブロードキャスト

6.5 Bluetooth デバイス名の利用方法

Bluetooth デバイス名はリモートデバイスに公開するユーザフレンドリな名前です。ユーザはデバイス名を確認することで、リモートデバイスを識別することができます。

6.5.1 ローカルデバイスのデバイス名

自分のデバイス名をリモートデバイスに公開するには、データベースの Device Name Characteristic にデバイス名を設定します。リモートデバイスは、接続後にデータベースからデバイス名を読み出すことが可能です。

BLE ソフトウェアは、データベースへ反映されるデバイス名を設定するために 2 通りの方法を提供します。

1 つは顧客固有情報のデバイス名領域を利用します。顧客固有情報に保存したデバイス名は、アドレス 0x3FC06 にアクセスすることで利用することが可能です。

2 つ目は"prf_config.h"に定義された GAP_DEV_NAME マクロです。顧客固有情報にデバイス名が設定されていない場合、このマクロが使用されます。

Table 6-1　デバイス名領域

情報	アドレス	サイズ	説明
デバイス名	0x3FC06	66 bytes	Bluetooth デバイス名 デバイス認識のためのユーザフレンドリな名前 0x3FC06: デバイス名長（1 から 65） 0x3FC07-0x3FC48: デバイス名（UTF-8）

ファイル：renesas/src/arch/rl78/prf_config.h

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

図 6-20　デバイス名のマクロ定義

また Advertising を実行することで、接続前にリモートデバイスへデバイス名を通知できます。デバイス名を通知するには、adv_data に Local Name AD タイプとデバイス名を設定して RBLE_GAP_Broadcast_Enable 関数を実行します。

注：　プライバシーが有効なデバイスは、デバイスが特定される恐れがあるため、デバイス名やユニークデータを Advertising データに含めるべきではありません。

6.5.2　リモートデバイスのデバイス名

RBLE_GAP_Get_Remote_Device_Name 関数を利用することで、接続の有無に関わらず図 6-21および図 6-22のようなシーケンスでリモートデバイスのデバイス名を取得することが可能です。

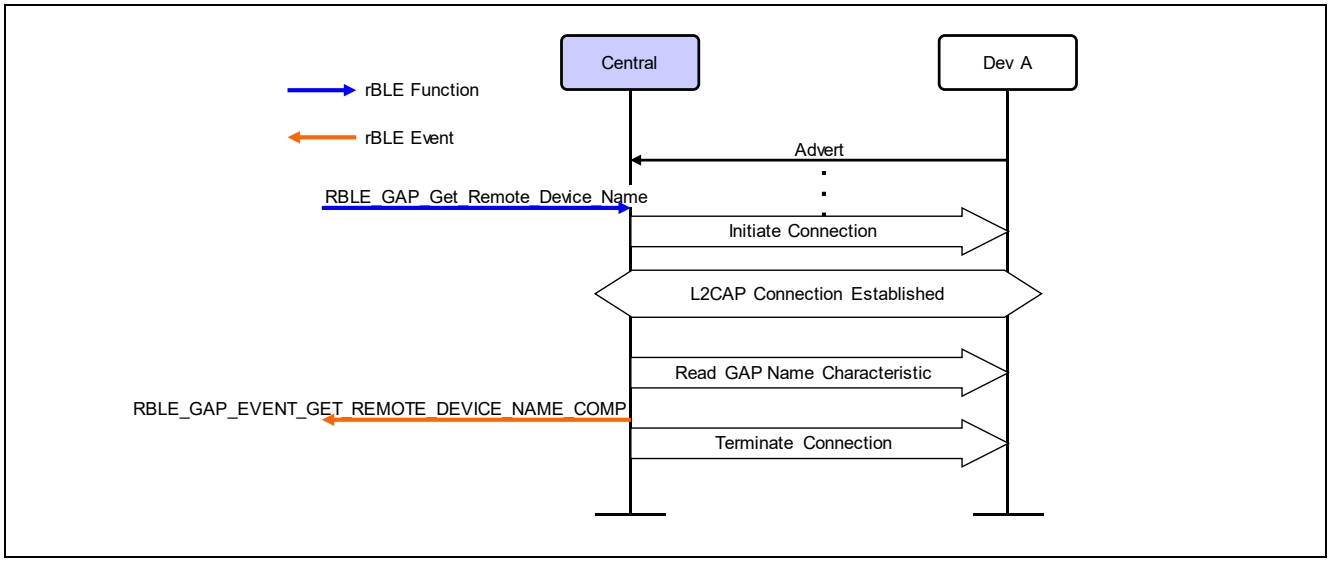


図 6-21　未接続時のデバイス名読み出し

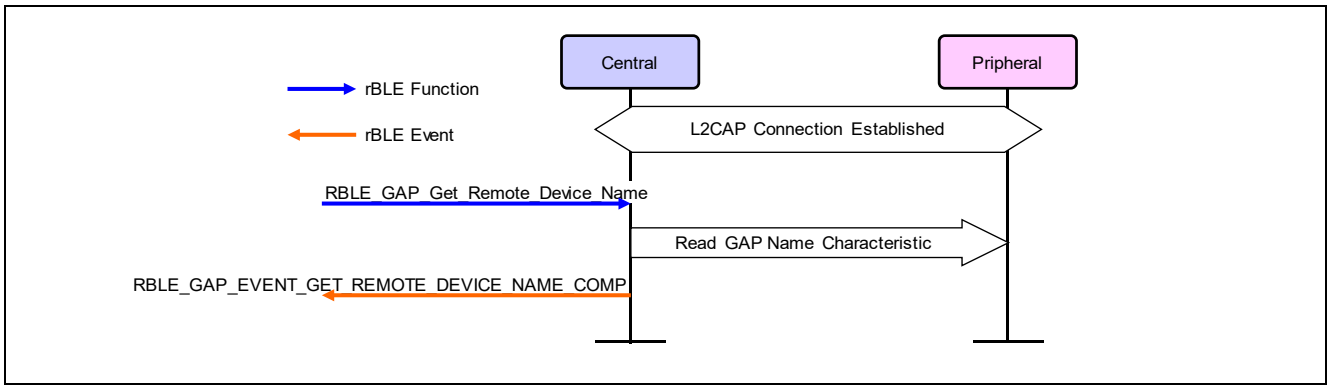


図 6-22　接続時のデバイス名読み出し

6.6　読み出しデータの更新

GATT クライアントから Read Request を受けた場合、BLE プロトコルスタックはアプリケーションに通知することなく、自動的に要求されたデータを Read Response で返します。したがって、アプリケーションは Read Response で返すデータを設定できません。

このため、図 6-23のように RBLE_GATT_Set_Data 関数によってあらかじめキャラクタリスティック・バリューを更新する必要があります。

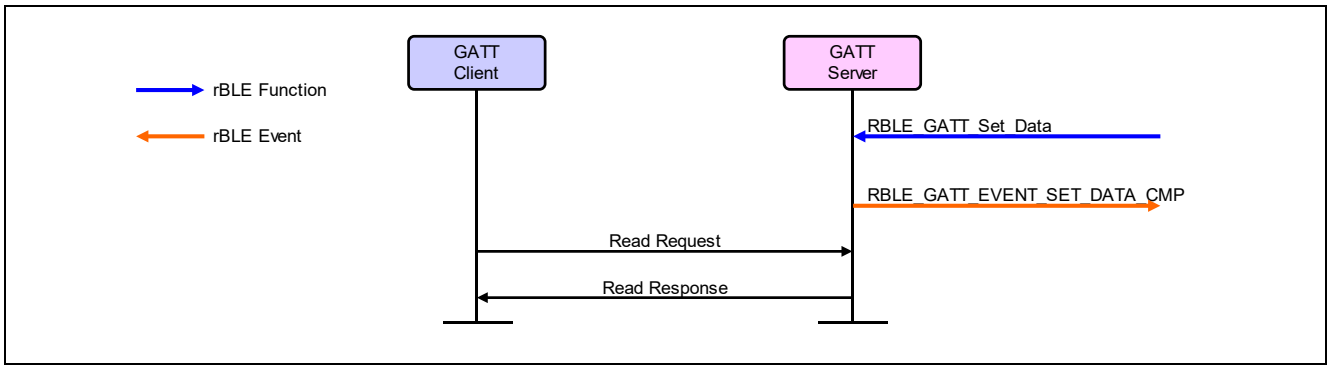


図 6-23　読み出しデータの更新と読み出しタイミング

7. Appendix

7.1 カスタムプロファイルへのキャラクタリスティック追加

簡易サンプルプログラムで使われている Sample Custom Service を使用して、サーバ処理の中で rBLE API や rBLE Event がどのように使用されるのか、さらに Sample Custom Service に新しくキャラクタリスティックを追加する方法を説明します。また、RL78/G1D 評価ボード(RTK0EN0001D01001BZ)(以下、評価ボード)とスマートフォンアプリケーションの GATTBrowser を使用してキャラクタリスティックの動作を確認します。

簡易サンプルプログラムについては、本書「5.1 簡易サンプルプログラムとは」を参照してください。

【項目】

- 7.1.1項 Sample Custom Service の定義を説明します
- 7.1.2項 データベースの構成を説明します
- 7.1.3項 Sample Custom Service サーバ処理を説明します
- 7.1.4項 Sample Custom Service にキャラクタリスティックを追加する方法を説明します
- 7.1.5項 追加したキャラクタリスティックがクライアントと通信できるようにサーバプロファイル API とペリフェラルアプリケーション処理の追加方法を説明します
- 7.1.6項 スマートフォンを使用して追加したキャラクタリスティックの動作を確認します
- 7.1.7項 Switch State Characteristic を Notification から Indication に対応させる方法を説明します。
- 7.1.8項 スマートフォンを使用して Indication に変更したキャラクタリスティックの動作を確認します

7.1.4項、7.1.5項、7.1.7項ではプロファイル構成の知識が必要になるため、併せて「4. プロファイル」も参照してください。また、記載されているソースコードは、以下バージョンの BLE プロトコルスタックを使用しています。

- BLE プロトコルスタック
「Bluetooth® low energy Protocol Stack (Ver.1.21)」
https://www.renesas.com/document/lbr/bluetooth-low-energy-protocol-stack-ver121

【参考資料】

- Embedded 構成サンプルプログラム
「Bluetooth low energy プロトコルスタック Embedded 構成サンプルプログラム」(R01AN3319)
https://www.renesas.com/document/scd/bluetooth-low-energy-protocol-stack-embedded-configuration-sample-program
- RL78/G1D 評価ボード(RTK0EN0001D01001BZ)
「RL78/G1D 評価ボード ユーザーズマニュアル」(R30UZ0048)
https://www.renesas.com/document/man/rl78g1d-users-manual-evaluation-board
- GATTBrowser
「GATTBrowser for Android スマートフォンアプリ取扱説明書」(R01AN3802)
https://www.renesas.com/document/apn/gattbrowser-android-smartphone-application-instruction-manual
「GATTBrowser for iOS スマートフォンアプリ取扱説明書」(R21AN0017)
https://www.renesas.com/document/apn/gattbrowser-ios-smartphone-application-instruction-manual

7.1.1 Sample Custom Service 定義

最初に、簡易サンプルプログラムで使用される Sample Custom Service の定義を以下に示します。定義の詳細は、「Bluetooth® Low Energy プロトコルスタック Embedded 構成サンプルプログラム - 5.4 Sample Custom Service」(R01AN3319)を参照してください。

表 7-1 簡易サンプルプログラムの Sample Custom Service 定義

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

Switch State Characteristic は Notification でスイッチ状態をクライアントへ送信するキャラクタリスティックです。Client Characteristic Configuration Descriptor は、クライアントから Notification 有効・無効の設定が書き込まれたことをアプリケーションへイベントで通知します。

アプリケーションはイベントで通知された Notification 有効・無効に応じて、評価ボード上のスイッチの押下・開放状態を送信する API(SAMPLE_Server_Send_Switch_State)を呼び出します。

SAMPLE_Server_Send_Switch_State API はスイッチの押下・開放状態を Switch State Characteristic Value に格納し、Notification でクライアントへ送信します。

(2) LED Control Characteristic

LED Control Characteristic は、クライアントが LED 点灯・消灯を制御できるキャラクタリスティックです。LED Control Characteristic Value は、クライアントから LED 点灯・消灯の状態が書き込まれ、アプリケーションへイベントで通知します。

アプリケーションはイベントで通知された LED の状態に応じて、評価ボード上の LED を制御します。

7.1.2　データベース構成

「表 7-1　簡易サンプルプログラムのSample Custom Service定義」のようなカスタムプロファイルをプログラムへ組み込むためには、サービスやキャラクタリスティックで構成される GATT データベース(以下、データベース)を作成します。

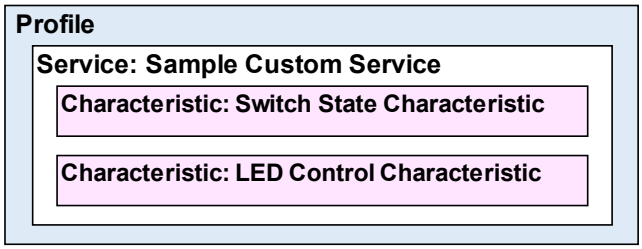


図 7-1　Sample Custom Service GATT データベース階層

BLE プロトコルスタックのデータベースは構造体配列で定義され、要素にサービスやキャラクタリスティック等を定義することでプロファイルとなります。以下のソースファイルでデータベース構造体配列の変数が宣言され、GATT ベースプロファイルが定義されています。

フォルダ	¥Renesas¥BLE_Software_Ver_X_XX¥RL78_G1D¥Project_Source¥renesas¥src¥arch¥rl78
ファイル名	prf_config.c
変数名	struct atts_desc atts_desc_list_prf[]

(1)　Sample Custom Service データベース構成

データベース構造体配列の中で、カスタムプロファイルはアトリビュートタイプ UUID を設定した幾つかの要素の組み合わせで構成されます。

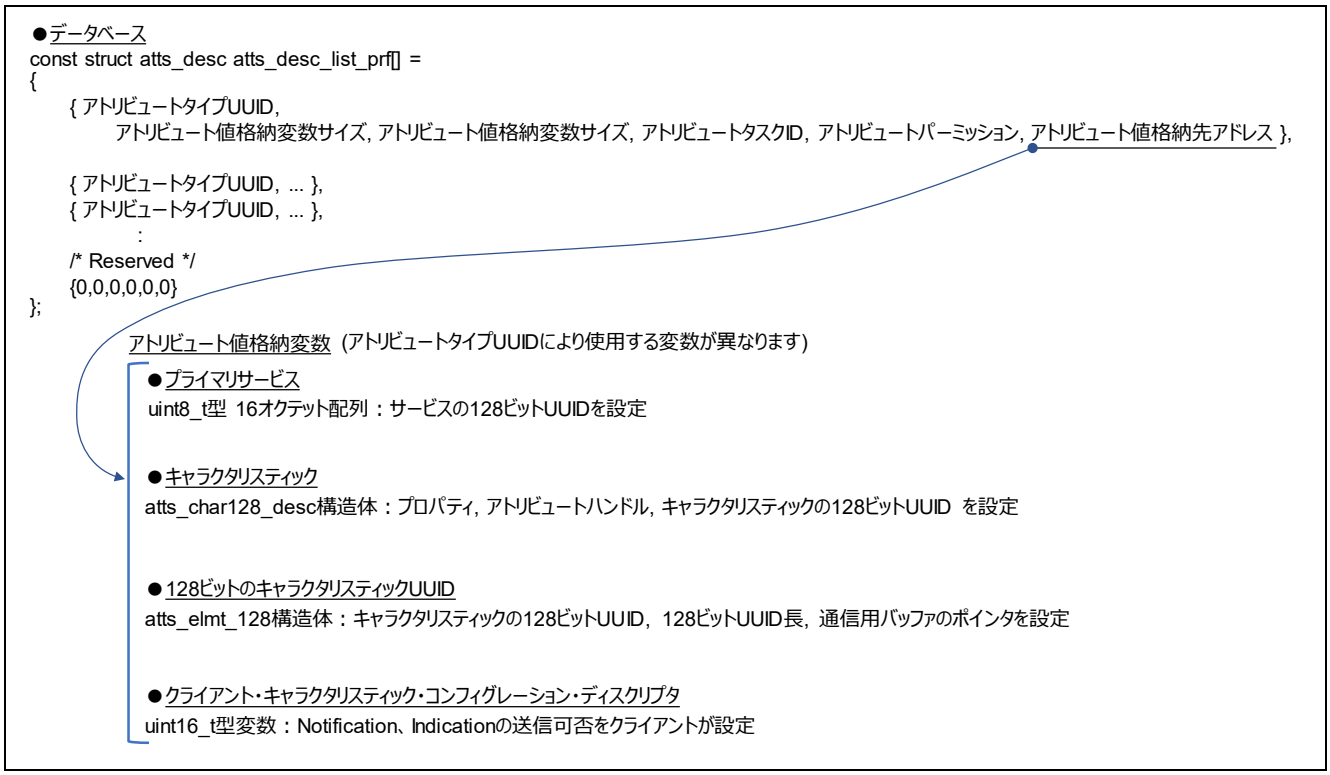


図 7-2　データベース構造体配列

カスタムプロファイルを構成するために使用されるアトリビュートタイプ UUID を以下に示します。

表 7-2　アトリビュートタイプ UUID

アトリビュートタイプ UUID 名	定義	UUID 値
プライマリサービス	RBLE_DECL_PRIMARY_SERVICE	0x2800
キャラクタリスティック	RBLE_DECL_CHARACTERISTIC	0x2803
128 ビットのキャラクタリスティック UUID	DB_TYPE_128BIT_UUID	0xffff
クライアント・キャラクタリスティック・コンフィグレーション・ディスクリプタ	RBLE_DESC_CLIENT_CHAR_CONF	0x2902

※上記はカスタムプロファイルを構成する必要最低限の定義です。他のアトリビュートタイプ UUID については「Bluetooth low energy プロトコルスタック ユーザーズマニュアル」(R01UW0095)の「7.4.1.2 アトリビュートタイプ」を参照してください。

「表 7-1　簡易サンプルプログラムのSample Custom Service定義」では、サービスとキャラクタリスティックのアトリビュートタイプ UUID を以下のように設定します。

表 7-3　Sample Custom Service アトリビュートタイプ UUID

Type		Value	Permission
Sample Custom Service			
	Primary Service Declaration	RBLE_DECL_PRIMARY_SERVICE	プライマリサービス
Switch State Characteristic			
	Characteristic Declaration	RBLE_DECL_CHARACTERISTIC	キャラクタリスティック
	Characterristic Value	DB_TYPE_128BIT_UUID	128 ビットのキャラクタリスティック UUID
	Client Characteristic Configuration Descriptor	RBLE_DESC_CLIENT_CHAR_CONF	クライアント・キャラクタリスティック・コンフィグレーション・ディスクリプタ
LED Control Characteristic			
	Characteristic Declaration	RBLE_DECL_CHARACTERISTIC	キャラクタリスティック
	Characterristic Value	DB_TYPE_128BIT_UUID	128 ビットのキャラクタリスティック UUID

- Sample Custom Service 定義

Switch State Characteristic と LED Control Characteristic を含むサービスの定義としてプライマリサービスの RBLE_DECL_PRIMARY_SERVICE を設定します。これにより Sample Custom Service の 128 ビット UUID (5BC1B9F7-A1F1-40AF-9043-C43692C18D7A)を割り当てることができます。

- Switch State Characteristic 定義

1 番目に、キャラクタリスティックであることを示す RBLE_DECL_CHARACTERISTIC を設定します。この定義の追加情報として、アトリビュート値格納変数の atts_char128_desc 構造体で、プロパティ定義、アトリビュートハンドル、キャラクタリスティックの UUID を設定します。

2 番目に、クライアントと通信するバリュー(クライアントとの通信用バッファ)の定義として DB_TYPE_128BIT_UUID を設定します。この定義の追加情報として、アトリビュート値格納変数の atts_elmt_128 構造体で、キャラクタリスティックの UUID 定義、128 ビット UUID サイズ、通信用バッファのポインタを設定します。

3 番目に、クライアント・キャラクタリスティック・コンフィグレーション・ディスクリプタ(Client Characteristic Configuration Descriptor 以下、CCCD)の定義として RBLE_DESC_CLIENT_CHAR_CONF を設定します。Switch State Characteristic は、サーバがスイッチの状態を Notificaiton で自発的にクライアントへ送信します。サーバが自発的にクライアントへデータを送信するためには、クライアントから送信許可を設定してもらう必要があり、CCCD は Notification の送信許可・禁止を設定する変数として用意します。CCCD については、本書「4.1プロファイルとは」を参照してください。

- LED Control Characteristic 定義

Switch State Characteristic と同様に 1 番目と 2 番目の定義を行います。LED Control Characteristic は、クライアントから送信された LED データを受信します。サーバから自発的にデータを送信しないため CCCD は定義しません。

(2) データベース atts_desc 構造体

atts_desk 構造体のメンバを以下に示します。この構造体を配列として定義することでデータベースが構成されます。

表 7-4　atts_desc 構造体

atts_desc 構造体		
	uint16_t type;	アトリビュートタイプ UUID
	uint8_t maxlength;	アトリビュート値格納変数サイズ ※アトリビュートタイプ UUID によってアトリビュート値格納変数が異なります。
	uint8_t length;	アトリビュート値格納変数サイズ maxlength と同じサイズを設定します ※アトリビュートタイプ UUID によってアトリビュート値格納変数が異なります。
	ke_task_id_t taskid;	上位 6bit：アトリビュートの属するプロファイルタスク ID 下位 10bit：アトリビュートを識別するためのインデックス
	uint16_t perm;	アトリビュートのパーミッション
	void *value;	アトリビュート値格納先へのポインタ ※アトリビュートタイプ UUID によってアトリビュート値格納変数が異なります。

- type

アトリビュートタイプの UUID を設定します。下記はカスタムプロファイルを構成する必要最低限の定義です。他のアトリビュートタイプ UUID については「Bluetooth low energy プロトコルスタック ユーザーズマニュアル」(R01UW0095)の「7.4.1.2 アトリビュートタイプ」を参照してください。

表 7-5　アトリビュートタイプ UUID

アトリビュートタイプ UUID	説明
RBLE_DECL_PRIMARY_SERVICE	プライマリサービス (0x2800)
RBLE_DECL_CHARACTERISTIC	キャラクタリスティック (0x2803)
DB_TYPE_128BIT_UUID	128 ビットのキャラクタリスティック UUID (0xffff)
RBLE_DESC_CLIENT_CHAR_CONF	クライアント･キャラクタリスティック･コンフィグレーション・ディスクリプタ (0x2902)

- maxlength

アトリビュート値格納変数のサイズを設定します。type で設定したアトリビュートタイプ UUID によってアトリビュート値格納変数が異なります。

設定例として、アトリビュートタイプ UUID に RBLE_DECL_PRIMARY_SERVICE(プライマリサービス)を設定した場合の変数宣言とメンバへの設定を以下に示します。

128 ビット UUID 配列宣言: sams_svc[RBLE_GATT_128BIT_UUID_OCTET]

メンバ設定例: sizeof(sams_svc)

- length

maxlength と同じ設定をします。

- taskid

prf_config.c で定義されている「TASK_ATTID」マクロを使用して、タスク ID(TASK_RBLE)と、データベースインデックスを結合し設定します。データベースインデックスについては本節「(7)データベースハンドル／データベースインデックス」を参照してください。

設定例として、アトリビュートタイプ UUID に RBLE_DECL_PRIMARY_SERVICE(プライマリサービス)を設定した場合のメンバへの設定例を以下に示します。

メンバ設定例: TASK_ATTID(TASK_RBLE, SAMS_IDX_SVC)

- perm

アトリビュートのパーミッションを設定します。パーミッションは GATT クライアントからのアクセスを制限するために設定します。主に使用するパーミッションを以下に示します。読み出しと書き込み等、複数の設定が必要な場合は、ビットごとの OR 演算子(|)を使用します。

表 7-6　パーミッション

パーミッション	説明
RBLE_GATT_PERM_RD	クライアントから読み出し可
RBLE_GATT_PERM_WR	クライアントから書き込み可
RBLE_GATT_PERM_NI	サーバから通知可(Notification, Indication)

※他の定義は「Bluetooth low energy プロトコルスタック ユーザーズマニュアル」(R01UW0095)の「7.4.1.2 アトリビュートタイプ - 表 7-21 アトリビュートパーミッション」を参照してください。

- *value

アトリビュート値格納先アドレス(アトリビュート値格納変数のポインタ)を設定します。type で設定したアトリビュートタイプ UUID によってアトリビュート値格納変数が異なります。アトリビュートタイプ UUID ごとのアトリビュート値格納先アドレスの設定例を以下に示します。

表 7-7　アトリビュート値格納先アドレス

アトリビュートタイプ UUID	アトリビュート値格納先アドレス設定例
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) アトリビュートタイプ UUID - プライマリサービス

アトリビュートタイプ UUID が RBLE_DECL_PRIMARY_SERVICE(プライマリサービス)の場合に 128 ビット UUID を格納する 16 オクテットの配列を設定します。プログラム上での 128 ビット UUID はリトルエンディアンで配列に設定します。

128 ビット UUID: 5BC1B9F7-A1F1-40AF-9043-C43692C18D7A

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

配列への設定:

uint8_t sams_svc[RBLE_GATT_128BIT_UUID_OCTET] = RBLE_SVC_SAMPLE_CUSTOM_SVC;

(4) アトリビュートタイプ UUID - キャラクタリスティック

アトリビュートタイプ UUID が RBLE_DECL_CHARACTERISTIC(キャラクタリスティック)の場合に atts_char128_desc 構造体を設定します。

表 7-8　atts_char128_desc 構造体

atts_char128_desc 構造体	
uint8_t prop;	プロパティ
uint8_t attr_hdl[sizeof(uint16_t)];	アトリビュートハンドル
uint8_t attr_type[RBLE_GATT_128BIT_UUID_OCTET];	キャラクタリスティックの UUID

- prop

キャラクタリスティックのプロパティ(特性: Read、Write、Notify、Indicate 等)を定義します。プロパティの種類については「Bluetooth low energy プロトコルスタック ユーザーズマニュアル」(R01UW0095)の「7.4 Generic Attribute Profile - 表 7-19 特性プロパティ」を参照してください。主に使用するプロパティを以下に示します。複数の設定が必要な場合は、ビットごとの OR 演算子(|)を使用し設定します。

表 7-9　プロパティ

パーミッション	説明
RBLE_GATT_CHAR_PROP_RD	クライアントから読み出し可
RBLE_GATT_CHAR_PROP_WR	クライアントから書き込み可
RBLE_GATT_CHAR_PROP_NTF	サーバから Notification 可能
RBLE_GATT_CHAR_PROP_IND	サーバから Indication 可能

- attr_hdl[]

アトリビュートタイプ UUID に RBLE_DECL_CHARACTERISTIC(キャラクタリスティック)を定義した要素のアトリビュートハンドルを設定します。以下に Switch State Characteristic での設定例を以下に示します。

メンバ設定例: {(uint8_t)(SAMS_HDL_SWITCH_STATE_VAL & 0xff),
(uint8_t)((SAMS_HDL_SWITCH_STATE_VAL >> 8) & 0xff)},

- attr_type[]

キャラクタリスティックの 128 ビット UUID 定義を設定します。Switch State Characteristic での設定例を以下に示します。

メンバ設定例: RBLE_CHAR_SAMS_SWITCH_STATE

定義ファイル: sam.h

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

(5) アトリビュートタイプ UUID - 128 ビットのキャラクタリスティック UUID

アトリビュートタイプ UUID が DB_TYPE_128BIT_UUID(128 ビットのキャラクタリスティック UUID)の場合に atts_elmt_128 構造体を設定します。

表 7-10　atts_elmt_128 構造体

atts_elmt_128 構造体	
uint8_t attr_uuid[RBLE_GATT_128BIT_UUID_OCTET];	キャラクタリスティックの UUID
uint8_t uuid_len;	128 ビット UUID 長
void *value;	通信用バッファのポインタ

- attr_uuid[]

キャラクタリスティックの UUID の定義を設定します。Switch State Characteristic での設定例を示します。

メンバ設定例: RBLE_CHAR_SAMS_SWITCH_STATE

128 ビット UUID 定義ファイル: sam.h

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

- uuid_len

128 ビット UUID のサイズ(16 オクテット)を設定します。

- *value

クライアントとの通信用バッファのポインタを設定します。Switch State Characteristic での設定例を示します。

メンバ設定例(配列): &switch_state_char_val[0]

メンバ設定例(変数): &switch_len_char_val

(6) アトリビュートタイプ UUID - クライアント・キャラクタリスティック・コンフィグレーション・ディスクリプタ

アトリビュートタイプ UUID が RBLE_DESC_CLIENT_CHAR_CONF(クライアント・キャラクタリスティック・コンフィグレーション・ディスクリプタ)の場合に設定します。uint16_t 型の変数です。

(7) データベースハンドル／データベースインデックス

データベースハンドルはクライアントに公開され、クライアントがサーバのサービスやキャラクタリスティックにアクセスするために使用されます。

データベースインデックスは、BLE プロトコルスタックがデータベースの要素を識別するために使用します。

両定義共にデータベースの要素と一対一で対応するように定義します。以下に Sample Custom Service データベース定義(アトリビュートタイプ UUID のみ示します)に対応するデータベースハンドル、データベースインデックスを示します。ソースコードへの記載方法は本書「4.3.1 データベースハンドルの追加」、「4.3.2 データベースインデックスの追加」を参照してください。

表 7-11　データベースハンドル、データベースインデックス

アトリビュートタイプ UUID	データベースハンドル	データベースインデックス
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 Sample Custom Service サーバ処理

Sample Custom Service 処理の中で rBLE API や rBLE Event、データベースハンドル(以下、ハンドル)がどのように使用されるのかフロー図(図 7-3 Sample custom serviceサーバ処理フロー図)で説明します。このフロー図で説明するサーバ処理と、関連するソースコードは以下になります。

(サーバのイネーブル・ディスエーブル処理の説明は省きます)

【Sample Custom Service サーバ処理】

- (A)スイッチの押下・解放状態を Notification で通知するフロー(図 7-3: A-1、A-2、A-3)
- (B) LED を点灯・消灯するフロー(図 7-3: B-1、B-2)

【関連ソースコード】

- rBLE/src/sample_simple/sam/sams.c
  Sample Custom Service のサーバソースコード。GATT の rBLE API や rBLE Event を使用して、クライアントからアクセスされたキャラクタリスティックの処理や、データ送信処理を行います。
- rBLE/src/sample_simple/sam/sams.h
  Sample Custom Service のイベントやイベントのパラメータを定義します。
- rBLE/src/sample_simple/sam/sam.h
  Sample Custom Service の UUID や、サーバ、クライアントで使用する共通のマクロを定義します。

Sample Custom Service サーバ処理フロー図について、キャラクタリスティックの処理は以下をトリガにして開始されます。

- クライアントからサーバのキャラクタリスティックに書き込まれ、GATT のイベントを通知するコールバック関数(sams_gatt_callback())で rBLE Event が発生した時
- アプリケーションから呼び出された Sample Custom Service API (SAMPLE_Server_Send_Switch_State())の中で GATT の rBLE API を呼び出し、GATT のイベントを通知するコールバック関数(sams_gatt_callback())で rBLE Event が発生した時

はじめに、「(A)スイッチの押下・解放状態を Notification で通知するフロー」を、次に「(B) LED の点灯・消灯を制御するフロー」を説明します。

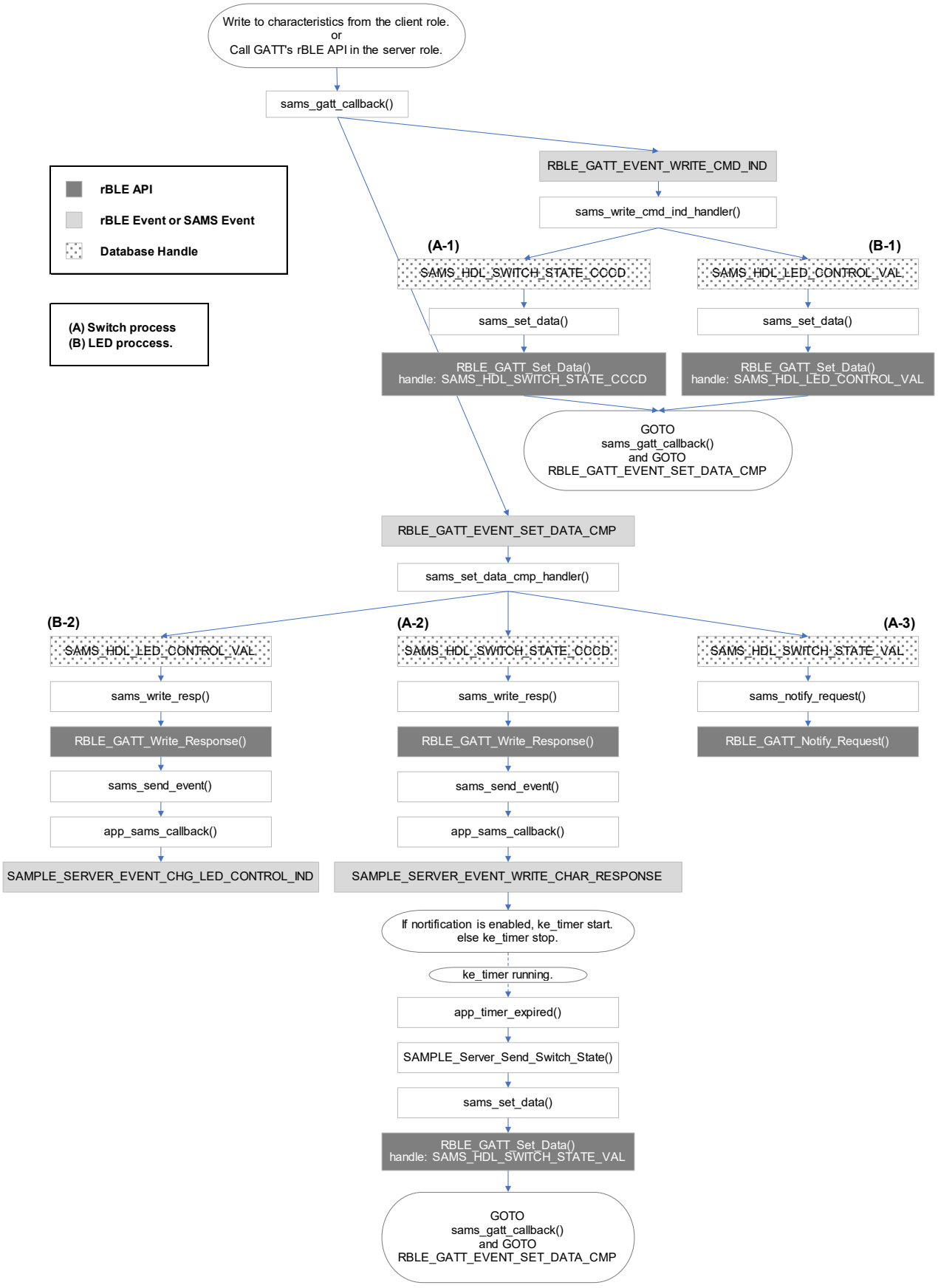


図 7-3 Sample custom service サーバ処理フロー図

(1) スイッチの押下・解放状態を Notification で通知するフロー

(A-1)

クライアントからサーバの Switch State Characteristic - Client Characteristic Configuration Descriptor(以下、Switch State CCCD)に Notification 許可・禁止が書き込まれると、sams_gatt_callback()で RBLE_GATT_EVENT_WRITE_CMD_IND イベントが発生します。

どのキャラクタリスティックに書き込こまれたかはイベントパラメータのハンドル(※)で判断され、ここでは SAMS_HDL_SWITCH_STATE_CCCD に書き込まれたことが分かります。このキャラクタリスティックへの書き込みはレスポンスを要求されているため、クライアントへ送信するデータを rBLE API の RBLE_GATT_Set_Data()で SAMS_HDL_SWITCH_STATE_CCCD にセットします。rBLE API の呼び出しにより sams_gatt_callback()で RBLE_GATT_EVENT_SET_DATA_CMP イベントが発生します。

(※) ハンドルとはサーバが持つサービスやキャラクタリスティックの通し番号です。クライアントは接続時にサーバが持つサービスやキャラクタリスティックを検索しています。クライアントがサーバのキャラクタリスティックにアクセスする時は、キャラクタリスティックのハンドルやデータを通信パケットの中に含めてサーバへ送信します。サーバは受信したパケットのハンドルを識別して処理を行います。

(A-2)

現在処理中のハンドル SAMS_HDL_SWITCH_STATE_CCCD を識別し Switch State CCCD の処理を実行します。(A-1)でセットしたデータは、RBLE_GATT_Write_Response()を呼び出しクライアントへ送信します。そしてペリフェラルアプリケーションへ Switch State CCCD に書き込みが発生しレスポンスを送信したことを SAMPLE_SERVER_EVENT_WRITE_CHAR_RESPONSE イベントで通知します。

ペリフェラルアプリケーションでは app_sams_callback()で SAMPLE_SERVER_EVENT_WRITE_CHAR_RESPONSE イベントが発生します。イベントのパラメータから Notification の許可・禁止を判定し、Nortification 許可であれば RWKE API のタイマ機能を使用して一定間隔でスイッチの状態を送信する SAMPLE_Server_Send_Switch_State()を呼び出します。Notification 禁止であればタイマ機能をストップし、スイッチ状態の送信を停止します。

Notification 許可の場合、SAMPLE_Server_Send_Switch_State() から RBLE_GATT_Set_Data()が呼び出されハンドル SAMS_HDL_SWITCH_STATE_VAL(Switch State Characteristic - Characteristic Value)にデータをセットします。

rBLE API の呼び出しにより sams_gatt_callback()で RBLE_GATT_EVENT_SET_DATA_CMP イベントが発生します。

(A-3)

現在処理中のハンドル SAMS_HDL_SWITCH_STATE_VAL を識別し、スイッチ状態をクライアントへ Notification で送信するために RBLE_GATT_Notify_Request()を呼び出します。

クライアントから Notificaiton 禁止を Switch State CCCD に書き込まれるまでスイッチ状態を送信し続けます。

(2) LED の点灯・消灯を制御するフロー

(B-1)

クライアントからサーバの LED Control Characteristic - Characteristic Value (以下、LED Control CV)に LED の点灯・消灯が書き込まれると、sams_gatt_callback()で RBLE_GATT_EVENT_WRITE_CMD_IND イベントが発生します。

どのキャラクタリスティックに書き込まれたかはイベントパラメータのハンドルで判断され、ここでは SAMS_HDL_LED_CONTROL_VAL に書き込まれたことが分かります。このキャラクタリスティックへの書き込みはレスポンスを要求されているため、クライアントへ送信するデータを rBLE API の RBLE_GATT_Set_Data()で SAMS_HDL_LED_CONTROL_VAL にセットします。rBLE API の呼び出しにより sams_gatt_callback()で RBLE_GATT_EVENT_SET_DATA_CMP イベントが発生します。

(B-2)

現在処理中のハンドル SAMS_HDL_LED_CONTROL_VAL を識別し LED Control CV の処理を実行します。(B-1)でセットしたデータは、RBLE_GATT_Write_Response()を呼び出しクライアントへ送信します。そしてペリフェラルアプリケーションへ、LED Control CV に書き込みが発生しレスポンスを送信したことを SAMPLE_SERVER_EVENT_CHG_LED_CONTROL_IND イベントで通知します。

ペリフェラルアプリケーションでは app_sams_callback()で SAMPLE_SERVER_EVENT_CHG_LED_CONTROL_IND イベントが発生します。イベントのパラメータから LED の点灯・消灯を制御します。

7.1.4　キャラクタリスティックの追加

クライアントからの読み出し要求で評価ボード上のディップスイッチ状態を送信するキャラクタリスティックの Dipswitch State Characteristic(表 7-12)を Sumple Custom Service に追加します。既存キャラクタリスティック(Switch State Characteristic、LED Control Characteristic)のプログラムコードについては本書「4.3 GATT データベースの作り方」を参照してください。

キャラクタリスティックを構成する項目とソースコードを以下に示します。

(1) UUID
　rBLE/src/sample_simple/sam/sam.h

(2) データベースハンドル
　renesas/src/arch/rl78/db_handle.h

(3) データベースインデックス
　renesas/src/arch/rl78/prf_config.h

(4) キャラクタリスティック
　renesas/src/arch/rl78/prf_config.c

(5) データベース
　renesas/src/arch/rl78/prf_config.c

表 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

Bluetooth SIG によって採択されていないカスタムサービスと、そのキャラクタリスティックの UUID はユーザが自由に決めることができます。Dipswitch State Characteristic の UUID は Samle Custom Service の UUID と共通の 128 ビットのランダム値を定義し、他のキャラクタリスティックと区別するため上位から 3 バイト目と 4 バイト目の値を変更します。ソースコードにはリトルエンディアンで定義します。

追加する UUID: 5BC11B83-A1F1-40AF-9043-C43692C18D7A

ファイル：rBLE/src/sample_simple/sam/sam.h

```
      ※ボールド部分をソースファイルへ追加

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}
```

図 7-4　Dipswitch State Characteristic UUID

(2) データベースハンドル

Dipswitch State Characteristic のデータベースハンドルを追加します。これはクライアントがサーバのサービスやキャラクタリスティックへアクセスするための番号で、データベースの要素と一対一に定義する必要があります。

データベースハンドルは、"db_handle.h"ヘッダファイルの DB_HDL_MAX より前に追加してください。

ファイル：renesas/src/arch/rl78/db_handle.h

※ボールド部分をソースファイルへ追加

```
/* Attribute database handles */
enum
{
    ...
    /* 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,
    SAMS_HDL_DIPSW_STATE_CHAR,  /* キャラクタリスティックのデータベースハンドル */
    SAMS_HDL_DIPSW_STATE_VAL,   /* バリューのデータベースハンドル */

    DB_HDL_MAX
};
```

図 7-5　Dipswitch State Characteristic データベースハンドル

(3) データベースインデックス

Dipswitch State Characteristic のデータベースインデックスを追加します。これは、BLE プロトコルスタックがサーバのデータベースを識別するために使用し、データベースの要素と一対一に定義する必要があります。

データベースインデックスは、"prf_config.h"ヘッダファイルに追加します。

ファイル：renesas/src/arch/rl78/prf_config.h

※ボールド部分をソースファイルへ追加

```
/** Attribute database index */
enum
{
    ...
    /* 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,
    SAMS_IDX_DIPSW_STATE_CHAR,  /* キャラクタリスティックのデータベースインデックス */
    SAMS_IDX_DIPSW_STATE_VAL    /* バリューのデータベースインデックス */
};
```

※SAMS_IDX_LED_CONTROL_VAL の行末に','を付加します。

図 7-6　Dipswitch State Characteristic データベースインデックス

(4) キャラクタリスティック

Dipswitch State Characteristic を定義する構造体と変数を追加します。

キャラクタリスティック構造体にはプロパティ、アトリビュートハンドル、キャラクタリスティックの UUID を設定します。クライアントから読み出しの指示を出すのでリードのプロパティとします。

キャラクタリスティック・バリューを定義します。評価ボードのディップスイッチは 4 つのスライドスイッチを持ちますが、SW6-2, SW6-3 は制御できないので SW6-1, SW6-4 の 2 バイトとします。

キャラクタリスティックは”prf_config.c”ファイルに定義します。

ファイル：renesas/src/arch/rl78/prf_config.c

```
      ※ボールド部分をソースファイルへ追加

1236: /* Characteristic(sams:dipswitch_state) */
1237: static const struct atts_char128_desc dipsw_state_char = {
1238:     RBLE_GATT_CHAR_PROP_RD,                          /* プロパティ */
1239:     {
1240:         (uint8_t)(SAMS_HDL_DIPSW_STATE_VAL & 0xff),/* アトリビュートハンドル */
1241:         (uint8_t)((SAMS_HDL_DIPSW_STATE_VAL >> 8) & 0xff)
1242:     },
1243:     RBLE_CHAR_SAMS_DIPSW_STATE          /* キャラクタリスティックの UUID  */
1244: };
1245:
1246: uint8_t dipsw_state_char_val[2] = {0};/* キャラクタリスティック・バリュー */
1247:
1248: struct atts_elmt_128 dipsw_state_char_val_elmt = {
1249:     RBLE_CHAR_SAMS_DIPSW_STATE,        /* キャラクタリスティックの UUID */
1250:     RBLE_GATT_128BIT_UUID_OCTET,       /* UUID 長 */
1251:     &dipsw_state_char_val[0] };        /*キャラクタリスティック・バリューへのポインタ */
1252: #endif /* #ifdef USE_SIMPLE_SAMPLE_PROFILE */
```

図 7-7　Dipswitch State Characteristic キャラクタリスティック

(5) データベース

最後に、キャラクタリスティック、キャラクタリスティック・バリューを GATT データベースに追加します。これらの定義を、”prf_config.c”ファイルの atts_desc_list_prf[]に追加します。

ファイル：renesas/src/arch/rl78/prf_config.c

```
      ※ボールド部分をソースファイルへ追加

      const struct atts_desc atts_desc_list_prf[] =
      {
          ...

          /*********************************
           * Simple Sample Service        *
           *********************************/
          ...
2151:     /* キャラクタリスティック */
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:     /* キャラクタリスティック・バリュー */
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 */
```

図 7-8　Dipswitch State Characteristic データベース

7.1.5 サーバプロファイル API とペリフェラルアプリケーション処理の追加

クライアントからの Dipswitch State Characteristic 読み出しにより、ディップスイッチの状態を送信できるようにペリフェラルアプリケーションと Sample Custom Service サーバのソースコードに処理を追加します。

ソースコードを追加するファイルを以下に示します。

【Sample Custom Service サーバソースコード】

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

【ペリフェラルアプリケーションソースコード】

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

Sample Custom Service サーバのソースコードには、Dipswitch State Characteristic にディップスイッチの状態を設定するサーバプロファイル API を追加します。

ペリフェラルアプリケーションのソースコードには、ディップスイッチの状態を監視して、変化があればサーバプロファイル API を呼び出す処理を追加します。

(1) サーバプロファイル API の追加

Dipswitch State Characteristic にディップスイッチの状態を設定するサーバプロファイル API を追加します。ディップスイッチの状態は、クライアントからの Read Request により、サーバが自動的に Read Response で送信します。サーバは Read Request の受信が発生したことをアプリケーションへ通知しないため、あらかじめデータをキャラクタリスティックへ設定しておく必要があります。キャラクタリスティックをリードするシーケンスは「図 4-13　Read Characteristics」を参照してください。

この API の仕様(表 7-13)と、追加するソースコードは以下になります。

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

表 7-13　Dipswitch State Characteristic サーバプロファイル API 仕様

RBLE_STATUS SAMPLE_Server_Set_Dipswitch_State (uint16_t conhdl, uint8_t *value)		
- 本 API は、SAMS の Dipswitch State Characteristic に value をセットします。 - value は、クライアントからの Read Request により、自動的に Read Response で送信されます。		
Parameters:		
uint16_t	conhdl	Connection Handle
uint8_t	*value	Dipswitch State - 2 オクテットの配列の先頭アドレスを指定します。 - value[0]:SW6-1 の状態 - value[1]:SW6-4 の状態
Return:		
RBLE_OK	Success	
RBLE_STATUS_ERROR	Status Error	

サーバプロファイル API のソースコードを以下に示します。SAMPLE_Server_Set_Dipswitch_State()内にある sams_set_data()の中で rBLE API の RBLE_GATT_Set_Data()が呼び出され、ディップスイッチ状態を Dipswitch State Characteristic に設定します。

ファイル：rBLE/src/sample_simple/sam/sams.c

```
      ※ボールド部分をソースファイルへ追加

      static void sams_set_data_cmp_handler(RBLE_GATT_EVENT *event)
      {
          ...

0213:     case SAMS_HDL_DIPSW_STATE_VAL:
0214:         /* do nothing */
0215:         break;

          default:
```

RBLE_GATT_Set_Data()の呼び出しにより発生する RBLE_GATT_EVENT_SET_DATA_CMP イベントから呼び出されるが、処理は行いません。

図 7-9　データセットイベント

ファイル：rBLE/src/sample_simple/sam/sams.c

```
      ※全体をソースファイルへ追加

343:  RBLE_STATUS SAMPLE_Server_Set_Dipswitch_State(uint16_t conhdl, uint8_t *value)
344:  {
345:      if (sams_info.conhdl != conhdl) {
346:          return RBLE_STATUS_ERROR;
347:      }
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;
356:35 }
```

アプリケーションから呼び出す API です。ディップスイッチ状態をキャラクタリスティックに設定します。

図 7-10　サーバプロファイル API

ファイル：rBLE/src/sample_simple/sam/sams.h

```
      ※全体をソースファイルへ追加

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

図 7-11　サーバプロファイル API のプロトタイプ宣言

ファイル：rBLE/src/sample_simple/sam/sam.h

```
      ※全体をソースファイルへ追加

0017: #define SAMPLE_DIPSW_STATE_SIZE  (2)
```

SW6-1, SW6-4 の状態を 2 バイトの配列に格納します。

図 7-12　キャラクタリスティックバリューサイズ

(2) ペリフェラルアプリケーション処理の追加

ディップスイッチの状態を取得し、サーバプロファイル API を呼び出す処理を追加します。追加するソースコードは以下になります。

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

ディップスイッチは RL78/G1D の外部割り込みが発生しないポートに割り当てられているため、ON から OFF への変化の検出ができません。そのためクライアントと接続した後に RWKE タイマ機能を用いて、定期的にタイマタスクを呼び出しディップスイッチの状態を Dipswitch State Characteristic へ設定します。

以下にディップスイッチ状態を Dipswitch State Characteristic へ設定するまでのフロー図を示します。

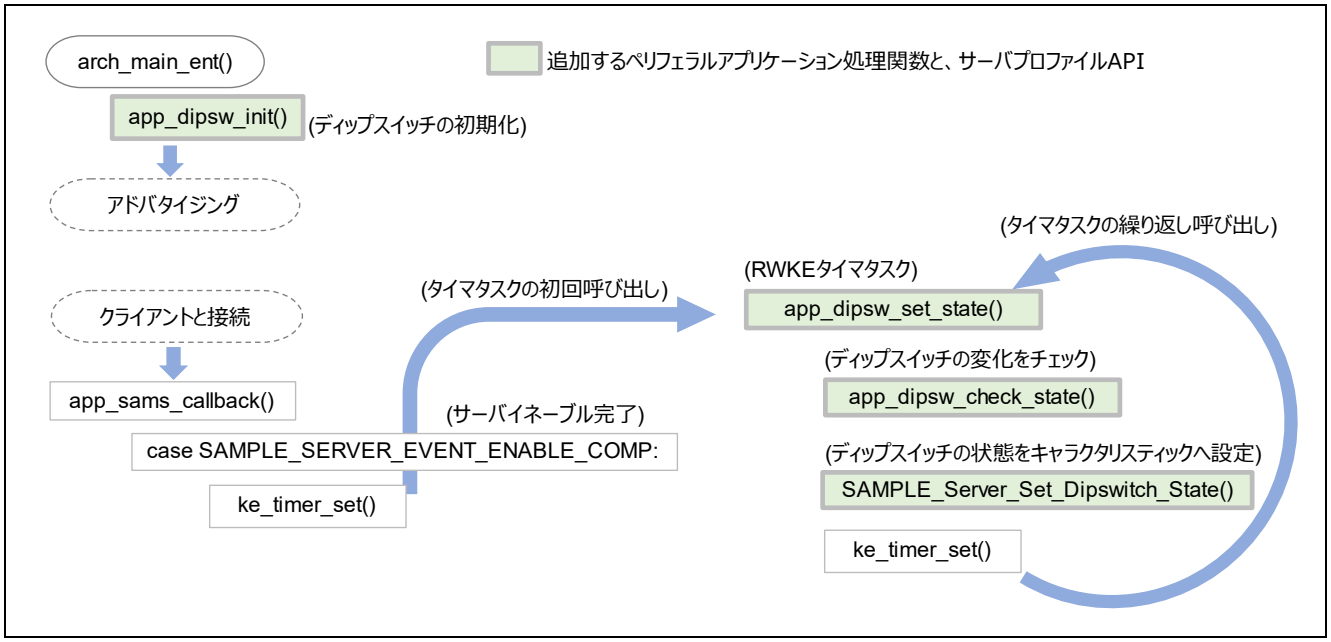


図 7-13　ディップスイッチ状態の設定処理

ペリフェラルアプリケーション処理のソースコードを以下に示します。

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

※全体をソースファイルへ追加

```
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];
```

図 7-14　ペリフェラルアプリケーション処理の関数と変数の定義

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

※ボールド部分をソースファイルへ追加

```
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 },
};
```

RWKE タイマ機能で呼び出すタスクを登録します。

図 7-15　ペリフェラルアプリケーション処理のメッセージハンドラテーブル

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

※ボールド部分をソースファイルへ追加

```
    ....
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: }
```

RWKE タイマ機能でディップスイッチ状態をチェックする間隔を定義します。

切断が発生した場合は、RWKE タイマ機能を停止します。

接続した後、サーバイネーブル完了で、RWKE タイマ機能を開始します。

ディップスイッチ状態を配列に格納した後、サーバプロファイル API を呼び出し、ディップスイッチ状態をキャラクタリスティックに設定します。そして RWKE タイマを再開します。

図 7-16　ペリフェラルアプリケーション処理の関数

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

```
      ※ボールド部分をソースファイルへ追加

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

RWKE タイマ機能で使用するメッセージ ID を定義します。

図 7-17　ペリフェラルアプリケーション処理のメッセージハンドラテーブル

メイン関数(arch_main_ent())からペリフェラルアプリケーション処理を呼び出すソースコードを以下に示します。

ファイル：renesas/src/arch/rl78/arch_main.c

```
      ※全体をソースファイルへ追加

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

図 7-18　ペリフェラルアプリケーション処理関数の extern 宣言

ファイル：renesas/src/arch/rl78/arch_main.c

```
      ※ボールド部分をソースファイルへ追加

          // Enable the BLE core
          rwble_enable();

0310:     app_dipsw_init();

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

図 7-19　ディップスイッチ初期化処理の呼び出し

7.1.6　スマートフォンを使用した動作確認 (Dipswitch State Characteristic)

スマートフォンアプリケーションの GATTBrowser と接続して、追加したキャラクタリスティックの動作確認を行ないます。ここでは Android スマートフォンを使用しますが、iOS の GATTBrowser も同じ操作で確認を行なうことができます。

簡易サンプルプログラムのプロジェクトファイルは、下記の場所に格納されていますのでビルドして評価ボードへ書き込んでください。

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

簡易サンプルプログラムは、実行すると自動的にアドバタイジングを開始し接続可能な状態になります。以下の手順にしたがって GATTBrowser と接続し、キャラクタリスティックの動作確認を行なってください。

1. 評価ボードに電源を入れ、簡易サンプルプログラムを実行します。
2. Android スマートフォンで GATTBrowser を実行します。
3. (図 1 の矢印(1)) 簡易サンプルプログラムが動作する評価ボードが見つかるので、接続アイコンをタップし評価ボードと接続を行います。
4. (図 2 の矢印(2)) 接続した画面を上にスライドさせ一番下のキャラクタリスティックを表示します。
5. (図 3 の矢印(3)) 追加したキャラクタリスティック[UUID:5bc11b83-40af-9043-c43692c18d7a]をタップしキャラクタリスティック画面を表示します。
6. (図 4 の矢印(4)) "Read"ボタンをタップするとディップスイッチの状態が表示されます。評価ボードの SW6-1, SW6-4 を操作しディップスイッチの状態が変わることを確認してください。

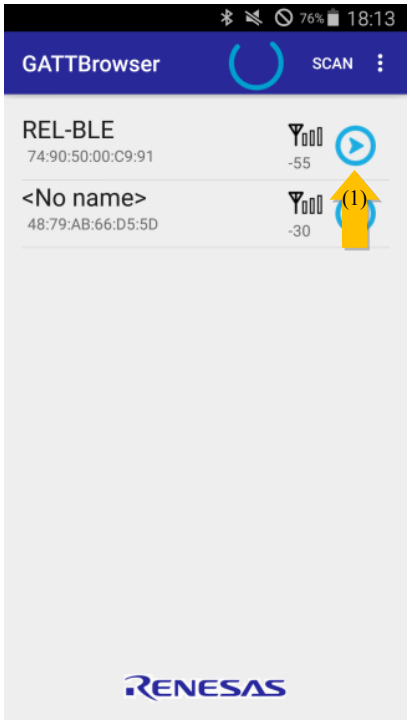


図 1

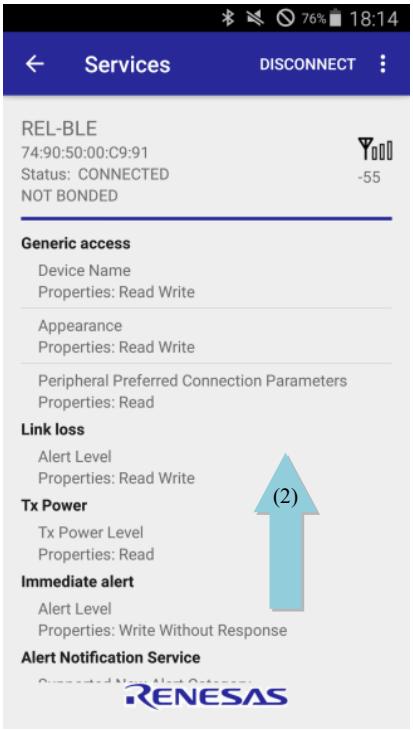


図 2

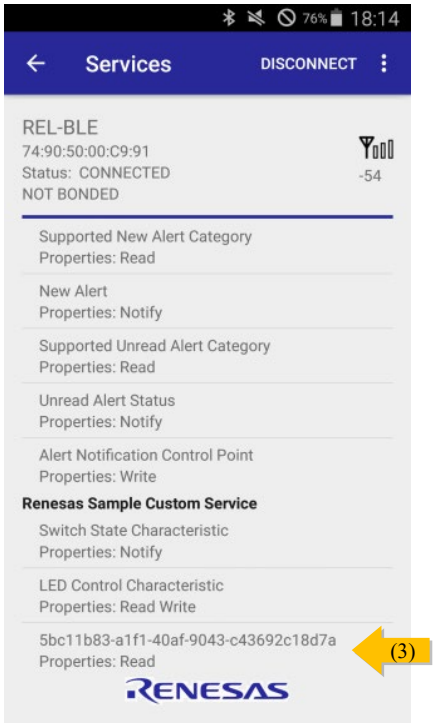

18:14
Services
DISCONNECT
REL-BLE
74:90:50:00:C9:91
Status: CONNECTED
NOT BONDED
-54
Supported New Alert Category
Properties: Read
New Alert
Properties: Notify
Supported Unread Alert Category
Properties: Read
Unread Alert Status
Properties: Notify
Alert Notification Control Point
Properties: Write
Renesas Sample Custom Service
Switch State Characteristic
Properties: Notify
LED Control Characteristic
Properties: Read Write
5bc11b83-a1f1-40af-9043-c43692c18d7a
Properties: Read
(3)
RENESAS

図 3

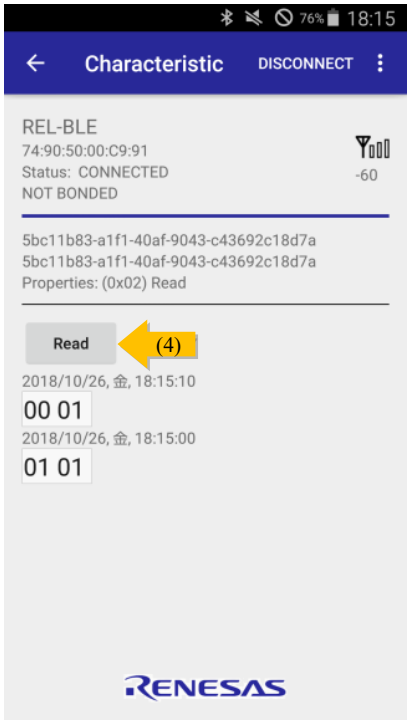

18:15
Characteristic
DISCONNECT
REL-BLE
74:90:50:00:C9:91
Status: CONNECTED
NOT BONDED
-60
5bc11b83-a1f1-40af-9043-c43692c18d7a
5bc11b83-a1f1-40af-9043-c43692c18d7a
Properties: (0x02) Read
Read
(4)
2018/10/26, 金, 18:15:10
00 01
2018/10/26, 金, 18:15:00
01 01
RENESAS

図 4

7.1.7 Notification から Indication への変更

Notification は、サーバからクライアントへデータ送信してもクライアントからの応答はありませんが、Indication ではクライアントから Confirmation という応答が返ってきます。サーバはその応答を受信することにより、クライアントにデータが届いたことを知ることができます。

ここでは、Switch State Characteristic の Notification を Indication に変更する方法を説明します。主な変更点を以下に示します。

(1) Switch State Characteristicデータベースの変更
- データベースを Notification から Indication に変更します

(2) Client Characteristic Configuration Descriptor処理
- クライアントから CCCD に設定された値の判定を、Indication に変更します

(3) データ送信処理
- データ送信を Indication の API を使用するように変更します

(4) Confirmationイベントの追加とアプリケーションへの通知
- クライアントからの Confirmation で発生するイベント処理とアプリケーションへの通知処理を追加追加します

CCCD の設定値や、Notification・Indication における通信方式の違いは、本書「表 4-1　Client Characteristic Configuration Descriptorの設定値」、「4.4.3(c) Notification Characteristic」、「4.4.3(d) Indication Characteristic」を参照してください。

最初に「図 7-3 Sample custom serviceサーバ処理フロー図」を用いて、Indication への変更によって Switch State Characteristic 処理がどのように実行されるか説明します。

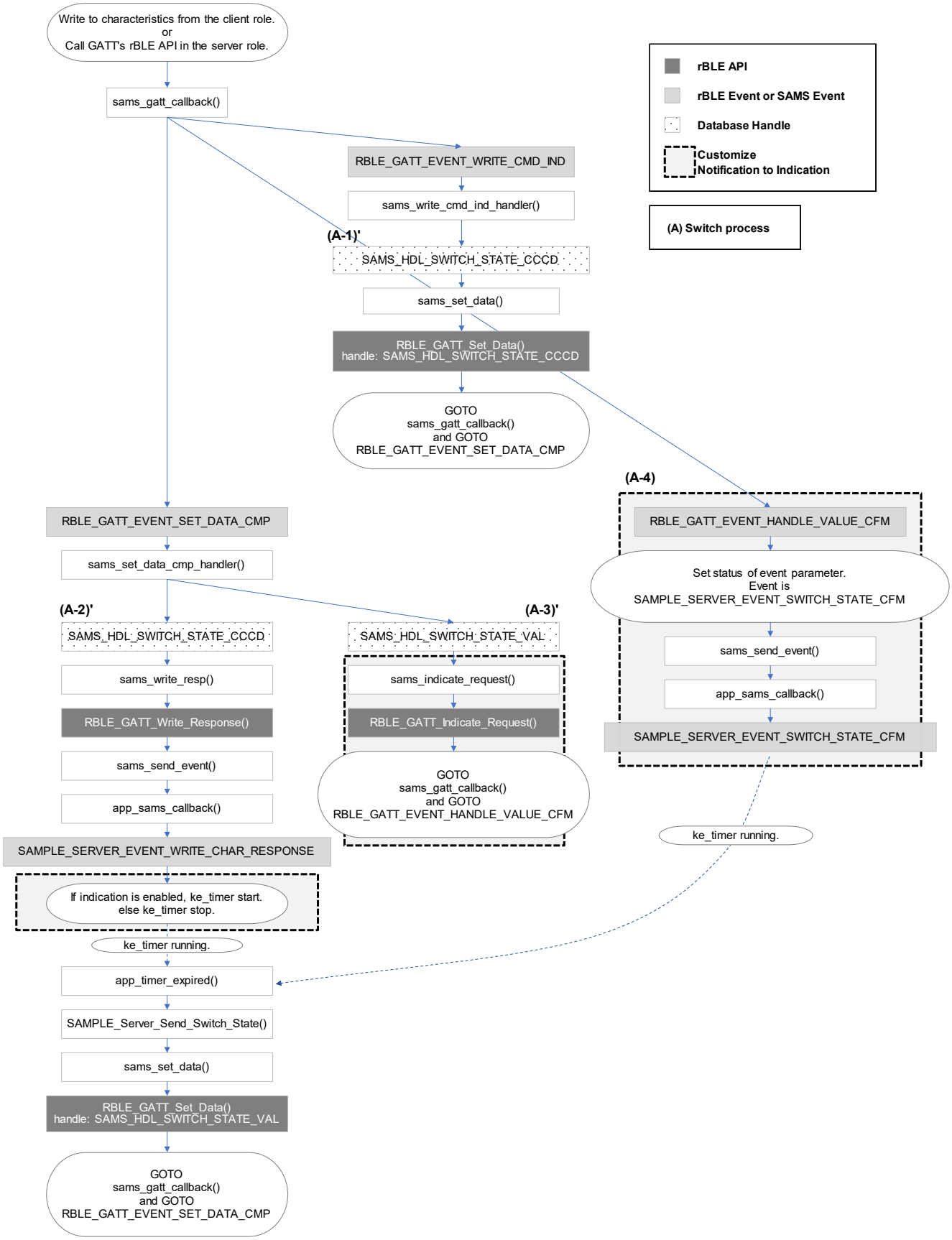


図 7-20 Switch State Characteristic の Indication 対応 フロー図

※アンダーラインの箇所が Notification から Indication に変更する処理です。

(A-1)'

クライアントからサーバの Switch State CCCD に Indication 許可・禁止が書き込まれると、sams_gatt_callback()で RBLE_GATT_EVENT_WRITE_CMD_IND イベントが発生します。

どのキャラクタリスティックに書き込こまれたかはイベントパラメータのハンドルで判断され、ここでは SAMS_HDL_SWITCH_STATE_CCCD に書き込まれたことが分かります。このキャラクタリスティックへの書き込みはレスポンスを要求されているため、クライアントへ送信するデータを rBLE API の RBLE_GATT_Set_Data()で SAMS_HDL_SWITCH_STATE_CCCD にセットします。rBLE API の呼び出しにより sams_gatt_callback()で RBLE_GATT_EVENT_SET_DATA_CMP イベントが発生します。

(A-2)'

現在処理中のハンドル SAMS_HDL_SWITCH_STATE_CCCD を識別し Switch State CCCD の処理を実行します。(A-1)でセットしたデータは、RBLE_GATT_Write_Response()を呼び出しクライアントへ送信します。そしてペリフェラルアプリケーションへ Switch State CCCD に書き込みが発生しレスポンスを送信したことを SAMPLE_SERVER_EVENT_WRITE_CHAR_RESPONSE イベントで通知します。

ペリフェラルアプリケーションでは app_sams_callback()で SAMPLE_SERVER_EVENT_WRITE_CHAR_RESPONSE イベントが発生します。イベントのパラメータから Indication の許可・禁止を判定し、Indication 許可であれば RWKE API のタイマ機能を使用して一定間隔でスイッチの状態を送信する SAMPLE_Server_Send_Switch_State()を呼び出します。Indication 禁止であればタイマ機能をストップし、スイッチ状態の送信を停止します。

Indication 許可の場合、SAMPLE_Server_Send_Switch_State() から RBLE_GATT_Set_Data()が呼び出されハンドル SAMS_HDL_SWITCH_STATE_VAL(Switch State Characteristic - Characteristic Value)にデータをセットします。

rBLE API の呼び出しにより sams_gatt_callback()で RBLE_GATT_EVENT_SET_DATA_CMP イベントが発生します。

(A-3)'

現在処理中のハンドル SAMS_HDL_SWITCH_STATE_VAL を識別し、スイッチ状態をクライアントへ Indication で送信するために RBLE_GATT_Indicate_Request()を呼び出します。

クライアントはスイッチ状態を受け取ると、受け取ったことを通知する Confirmation を送信します。サーバは Confirmation の受信により sams_gatt_callback()で RBLE_GATT_EVENT_HANDLE_VALUE_CFM イベントが発生します。

(A-4)

クライアントから Confirmation を受け取ったことをペリフェラルアプリケーションへ通知します。通知は SAMPLE_SERVER_EVENT_SWITCH_STATE_CFM イベントと同時にイベントパラメータも渡します。ペリフェラルアプリケーションでは app_sams_callback()で RBLE_GATT_EVENT_HANDLE_VALUE_CFM イベントが発生します。

クライアントから Indication 無効を Switch State CCCD に書き込まれるまでスイッチ状態を送信し続けます。

(1) Switch State Characteristic データベースの変更

プロパティ定義を Notification から Indication に変更します。

ファイル：renesas/src/arch/rl78/prf_config.c

※ボールド部分を変更

```
    ....

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};
```

図 7-21 Switch State Characteristic データベースの変更

(2) Client Characteristic Configuration Descriptor 処理

クライアントから CCCD に Indication の許可・禁止が設定されると、ペリフラルアプリケーションで SAMPLE_SERVER_EVENT_WRITE_CHAR_RESPONSE が発生します。Indication の許可・禁止が判定できるように定義を Notification から Indication に変更します。

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

※ボールド部分を変更

```
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;
```

図 7-22　ペリフェラルアプリケーション処理の関数と変数の定義

(3) データ送信処理

データ送信で Indication の API を使用するように変更します。

ファイル：rBLE/src/sample_simple/sam/sams.c

※ボールド部分をソースファイルへ追加、または変更

```
    ....
0077: static void sams_indicate_request(void);                    追加


0163: static void sams_indicate_request(void)                     追加
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();                              変更
0222:       sams_indicate_request();
            break;


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

図 7-23　ディップスイッチ状態変化検出関数の呼び出し処理

(4) Confirmation イベントの追加とアプリケーションへの通知

サーバからの Indication 送信によるクライアントからの Confirmation 応答で発生するイベント(RBLE_GATT_EVENT_HANDLE_VALUE_CFM)の処理を追加します。また、Confirmation を受信したことをペリフェラルアプリケーションへ通知するイベント(SAMPLE_SERVER_EVENT_SWITCH_STATE_CFM)を追加します。

ファイル：rBLE/src/sample_simple/sam/sams.h

```
      ※ボールド部分をソースファイルへ追加
          ....
      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;
```

図 7-24　Confirmation 受信を通知するイベント

ファイル：rBLE/src/sample_simple/sam/sams.c

```
      ※ボールド部分をソースファイルへ追加
          ....
      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:
```

ペリフェラルアプリケーションへ Confirmation の受信を通知します。

Confirmation を受信すると発生する GATT イベントです。

図 7-25　Confirmation 受信処理

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

```
      ※ボールド部分をソースファイルへ追加

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

ペリフェラルアプリケーションで Confirmation の受信が通知されるイベントです。

図 7-26　Confirmation 受信をペリフェラルアプリケーションへ通知

7.1.8　スマートフォンを使用した動作確認 (Indication of Switch State Characteristic)

初期状態の簡易サンプルプログラムと「7.1.7 NotificationからIndicationへの変更」で変更した簡易サンプルプログラムの違いを、GATTBrowser を使用して確認します。「7.1.6 スマートフォンを使用した動作確認 (Dipswitch State Characteristic)」と同様に Android スマートフォンの GATTBrowser を使用します。

簡易サンプルプログラムのプロジェクトファイルは、下記の場所に格納されていますのでビルドして評価ボードへ書き込んでください。

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

簡易サンプルプログラムは、実行すると自動的にアドバタイジングを開始し接続可能な状態になるので、GATTBrowser と接続してください。

1. 初期状態の簡易サンプルプログラム(Notification)と GATTBrowser を接続してください。
2. (図 1) 接続した"Service"の画面で"Switch State Characteristic"の"Properties"が"Notify"であることを確認してください。
3. (図 2) "Characteristic"の画面でボタンが"Notification"であることを確認してください。ボタンをタップすると評価ボードから Notification で SW4 の状態が送信されます。"Discriptors"の"value"が"01 00"(Notification 許可)であることを確認してください。(value の値はリトルエンディアンなので実際の値は 0x0001 です)
4. 変更した簡易サンプルプログラム(Indication)と GATTBrowser を接続してください。
5. (図 3) 接続した"Service"の画面で"Switch State Characteristic"の"Properties"が"Indicate"であることを確認してください。
6. (図 4) "Characteristic"の画面でボタンが"Indiation"であることを確認してください。ボタンをタップすると評価ボードから Notification で SW4 の状態が送信されます。"Discriptors"の"value"が"02 00"(Indication 許可)であることを確認してください。(value の値はリトルエンディアンなので実際の値は 0x0002 です)

Notification	Indication

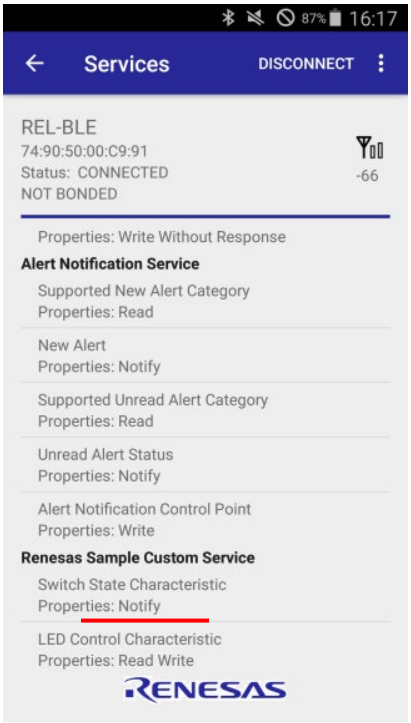


図 1

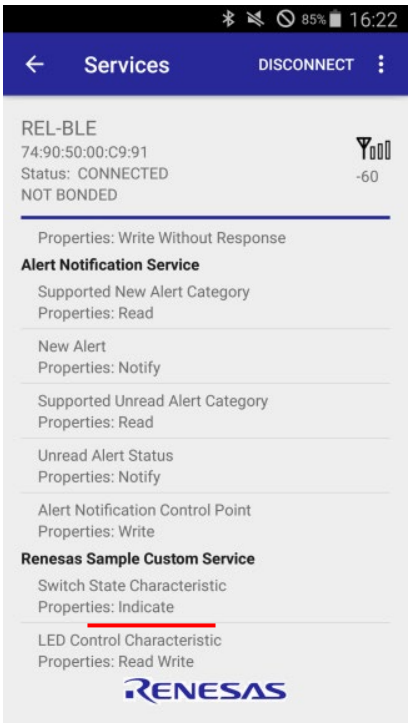


図 3

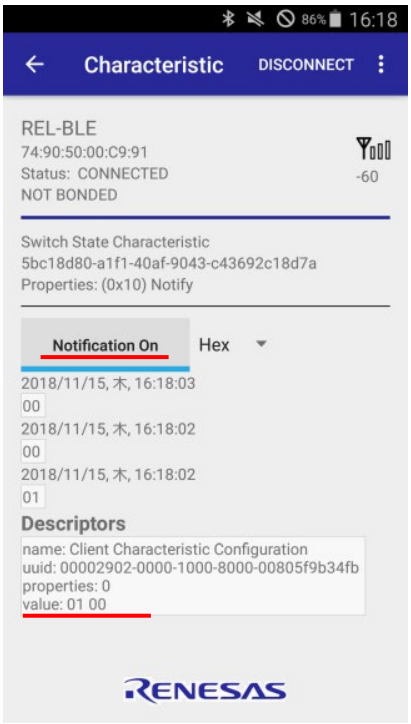


図 2

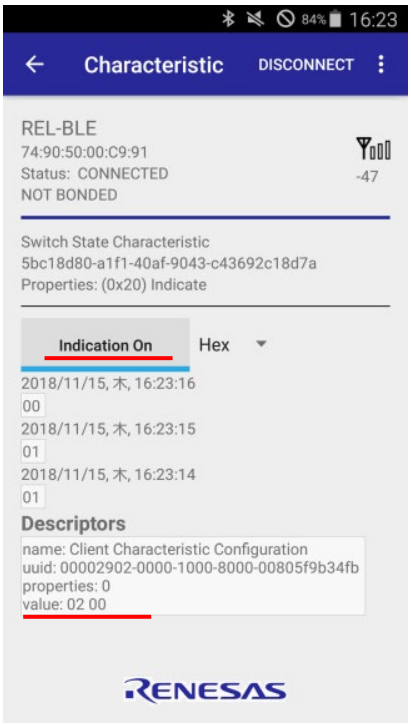


図 4

ホームページとサポート窓口

ルネサス エレクトロニクスホームページ
https://www.renesas.com/

お問合せ先
https://www.renesas.com/contact/

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2015.04.28	ー	初版発行
1.10	2016.09.23	ー	BLE プロトコルスタック V1.20 に適用 5 章、6 章、7 章、8 章を追加
1.20	2017.11.28	ー	2章、5章を追加し、内容を下記の通り再構成 1章 BLEソフトウェア 2章 RWKE 3章 BLEプロトコルスタック 4章 プロファイル 5章 アプリケーションの動作例 6章 開発のヒント
		P.5	2章 RWKEを追加、下記を含む 2.1節 RWKEとは 2.2節 RWKEの実行 2.3節 RWKE API 2.4節 ユースケース 2.5節 アプリケーションの実装 2.6節 注意点
		P.46	5章 アプリケーションの動作例を追加、下記を含む 5.1節 簡易サンプルプログラムとは 5.2節 BLEソフトウェアの起動 5.3節 BLEプロトコルスタックの初期化 5.4節 ブロードキャスト開始と接続確立 5.5節 カスタムプロファイル有効化 5.6節 カスタムプロファイルのデータ通信 5.7節 カスタムプロファイル無効化とブロードキャスト再開
1.30	2018.11.19	P.78	7章 Appendixを追加、下記を含む 7.1節 カスタムプロファイルへのキャラクタリスティック追加

製品ご使用上の注意事項

ここでは、マイコン製品全体に適用する「使用上の注意事項」について説明します。個別の使用上の注意事項については、本ドキュメントおよびテクニカルアップデートを参照してください。

1. 未使用端子の処理
   【注意】未使用端子は、本文の「未使用端子の処理」に従って処理してください。
   CMOS 製品の入力端子のインピーダンスは、一般に、ハイインピーダンスとなっています。未使用端子を開放状態で動作させると、誘導現象により、LSI 周辺のノイズが印加され、LSI 内部で貫通電流が流れたり、入力信号と認識されて誤動作を起こす恐れがあります。未使用端子は、本文「未使用端子の処理」で説明する指示に従い処理してください。
2. 電源投入時の処置
   【注意】電源投入時は，製品の状態は不定です。
   電源投入時には、LSI の内部回路の状態は不確定であり、レジスタの設定や各端子の状態は不定です。
   外部リセット端子でリセットする製品の場合、電源投入からリセットが有効になるまでの期間、端子の状態は保証できません。
   同様に、内蔵パワーオンリセット機能を使用してリセットする製品の場合、電源投入からリセットのかかる一定電圧に達するまでの期間、端子の状態は保証できません。
3. リザーブアドレス（予約領域）のアクセス禁止
   【注意】リザーブアドレス（予約領域）のアクセスを禁止します。
   アドレス領域には、将来の機能拡張用に割り付けられているリザーブアドレス（予約領域）があります。これらのアドレスをアクセスしたときの動作については、保証できませんので、アクセスしないようにしてください。
4. クロックについて
   【注意】リセット時は、クロックが安定した後、リセットを解除してください。
   プログラム実行中のクロック切り替え時は、切り替え先クロックが安定した後に切り替えてください。
   リセット時、外部発振子（または外部発振回路）を用いたクロックで動作を開始するシステムでは、クロックが十分安定した後、リセットを解除してください。また、プログラムの途中で外部発振子（または外部発振回路）を用いたクロックに切り替える場合は、切り替え先のクロックが十分安定してから切り替えてください。
5. 製品間の相違について
   【注意】型名の異なる製品に変更する場合は、製品型名ごとにシステム評価試験を実施してください。
   同じグループのマイコンでも型名が違うと、内部 ROM、レイアウトパターンの相違などにより、電気的特性の範囲で、特性値、動作マージン、ノイズ耐量、ノイズ輻射量などが異なる場合があります。型名が違う製品に変更する場合は、個々の製品ごとにシステム評価試験を実施してください。

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器・システムの設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因して生じた損害（お客様または第三者いずれに生じた損害も含みます。以下同じです。）に関し、当社は、一切その責任を負いません。
2. 当社製品、本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害またはこれらに関する紛争について、当社は、何らの保証を行うものではなく、また責任を負うものではありません。
3. 当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
4. 当社製品を、全部または一部を問わず、改造、改変、複製、リバースエンジニアリング、その他、不適切に使用しないでください。かかる改造、改変、複製、リバースエンジニアリング等により生じた損害に関し、当社は、一切その責任を負いません。
5. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、各品質水準は、以下に示す用途に製品が使用されることを意図しております。

   標準水準：　コンピュータ、OA機器、通信機器、計測機器、AV機器、
   家電、工作機械、パーソナル機器、産業用ロボット等

   高品質水準：　輸送機器（自動車、電車、船舶等）、交通制御（信号）、大規模通信機器、
   金融端末基幹システム、各種安全制御装置等

   当社製品は、データシート等により高信頼性、Harsh environment向け製品と定義しているものを除き、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（宇宙機器と、海底中継器、原子力制御システム、航空機制御システム、プラント基幹システム、軍事機器等）に使用されることを意図しておらず、これらの用途に使用することは想定していません。たとえ、当社が想定していない用途に当社製品を使用したことにより損害が生じても、当社は一切その責任を負いません。
6. 当社製品をご使用の際は、最新の製品情報（データシート、ユーザーズマニュアル、アプリケーションノート、信頼性ハンドブックに記載の「半導体デバイスの使用上の一般的な注意事項」等）をご確認の上、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他指定条件の範囲内でご使用ください。指定条件の範囲を超えて当社製品をご使用された場合の故障、誤動作の不具合および事故につきましては、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質および信頼性の向上に努めていますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は、データシート等において高信頼性、Harsh environment向け製品と定義しているものを除き、耐放射線設計を行っておりません。仮に当社製品の故障または誤動作が生じた場合であっても、人身事故、火災事故その他社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
8. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制するRoHS指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。かかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
9. 当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。当社製品および技術を輸出、販売または移転等する場合は、「外国為替及び外国貿易法」その他日本国および適用される外国の輸出管理関連法規を遵守し、それらの定めるところに従い必要な手続きを行ってください。
10. お客様が当社製品を第三者に転売等される場合には、事前に当該第三者に対して、本ご注意書き記載の諸条件を通知する責任を負うものといたします。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。
12. 本資料に記載されている内容または当社製品についてご不明な点がございましたら、当社の営業担当者までお問合せください。

注1.　本資料において使用されている「当社」とは、ルネサス エレクトロニクス株式会社およびルネサス エレクトロニクス株式会社が直接的、間接的に支配する会社をいいます。

注2.　本資料において使用されている「当社製品」とは、注1において定義された当社の開発、製造製品をいいます。

(Rev.4.0-1　2017.11)

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口　http://www.renesas.com

※営業お問合せ窓口の住所は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス株式会社　〒135-0061　東京都江東区豊洲3-2-24（豊洲フォレシア）

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：https://www.renesas.com/contact/