

# **BK3432 *Bluetooth*® Low Energy Software Developer's Guide**

## **v1.0**

Beken Corporation  
博通集成电路(上海)有限公司  
中国上海张江高科技园区  
张东路 1387 科技领袖之都 41 栋  
电话: (86)21 5108 6811  
传真: (86)21 6087 1277

文档含博通(BEKEN)公司保密信息, 非经书面许可, 不可外传



## 更改记录








版本号	日期	作者	注释
V1.0	2019-01-24	许海	文档建立

CONFIDENTIAL

## 目 录


1. SDK 架构介绍.....	4 -
1.1 BINARIES 目录.....	4 -
1.2 DOC 目录.....	4 -
1.3 LIBS 目录.....	4 -
1.4 PROJECTS 目录.....	4 -
1.6 UTILITIES 目录.....	5 -
2. 带你快速玩转 3432.....	5 -
2.1 用户配置.....	5 -
2.2 常用的 API 及回调函数.....	5 -
2.3 外设驱动使用举例.....	7 -
2.4 添加一个服务.....	10 -
2.5 添加一个自定义消息.....	12 -
2.6 配对方式修改.....	14 -
2.7 程序的烧录.....	15 -
2.8 OAD.....	17 -
2.9 RF 测试.....	17 -
3. SDK 例程介绍.....	17 -
3.1 数据透传.....	17 -
3.2 遥控器.....	18 -
3.3 苹果通知服务.....	18 -
3.4 微信接入.....	18 -
3.4 数据透传（128 BIT UUID）.....	19 -

## 1. sdk 架构介绍

	binaries	2018/2/5 11:03	文件夹	
	doc	2018/2/5 11:03	文件夹	
	libs	2018/2/5 11:03	文件夹	
	projects	2018/2/5 11:06	文件夹	
	sdk	2018/2/5 11:03	文件夹	
	utilities	2018/2/5 11:08	文件夹	
	readme.pdf	2018/2/5 10:48	Adobe Acrobat ...	90 KB

### 1.1 binaries 目录

该目录保存 sdk 中 demo 工程预编译完成的 bin 文件，可直接用于测试。

	bk3432_ble_demo_ancs.bin	2018/7/4 14:19	BIN 文件	137 KB
	bk3432_ble_demo_gatt.bin	2018/7/4 11:11	BIN 文件	131 KB
	bk3432_ble_demo_gatt_128.bin	2018/7/4 14:19	BIN 文件	132 KB
	bk3432_ble_demo_muti_chars.bin	2018/7/4 14:20	BIN 文件	132 KB
	bk3432_ble_demo_remote.bin	2018/7/4 10:32	BIN 文件	144 KB
	bk3432_ble_demo_wechat.bin	2018/7/4 14:24	BIN 文件	141 KB
	bk3432_fcc_test_v1.0.bin	2018/7/4 15:14	BIN 文件	123 KB

### 1.2 doc 目录


该目录保存开发过程中相关参考文档。

### 1.3 libs 目录

该目录用于存放与 sdk 相关的库文件。

### 1.4 projects 目录

该目录为 sdk 工程目录，存放芯片设计商提供的一些 demo 例程。

	ble_app_ancs	2019/1/7 18:37	文件夹	
	ble_app_gatt	2019/1/7 18:37	文件夹	
	ble_app_gatt_128	2019/1/7 18:37	文件夹	
	ble_app_gatt_muti_chars	2019/1/7 18:37	文件夹	
	ble_app_remote	2019/1/7 18:37	文件夹	
	ble_app_wechat	2019/1/7 18:37	文件夹	

ble\_app\_ancs: 支持苹果通知中心服务的最小工程;

ble\_app\_gatt: 数据传输最小工程;

ble\_app\_gatt\_128:128 bit uuid 使用 demo;  
ble\_app\_gatt\_muti\_chars:多个特征服务 demo;  
ble\_app\_remote: 语音遥控器最小工程;  
ble\_app\_wechat:支持微信接入功能的最小工程;

## 1.5 sdk 目录

 ble_stack	2018/2/5 11:03	文件夹
 plactform	2018/2/5 11:03	文件夹
 project_files	2018/2/5 11:03	文件夹

该目录存放 sdk 公共文件，其中 ble\_stack 保存蓝牙协议栈相关文件；plactform 保存平台相关文件，如 driver、rw 相关等；project\_files 保存工程通用性文件。

## 1.6 utilities 目录

该目录为工具类目录，用于保存 3432 开发过程中可能使用的相关工具，如烧录器估计、OTA 软件等。

## 2. 带你快速玩转 3432

### 2.1 用户配置

在软件开发包每个工程下面都有一个 user\_config.h 文件，这个文件主要是用来配置蓝牙的一些基本参数，如蓝牙名称、连接间隔、广播包数据、扫描响应包数据、驱动等，部分截图如下：

```
//设备名称
#define APP_DFLT_DEVICE_NAME          ("BK3432-GATT")

//广播包UUID配置
#define APP_FFF0_ADV_DATA_UUID        "\x03\x03\xF0\xFF"
#define APP_FFF0_ADV_DATA_UUID_LEN    (4)

//扫描响应包数据
#define APP_SCNRSP_DATA                "\x0c\x08\x42\x4B\x33\x34\x33\x32\x2D\x47\x41\x54\x54" //BK3432-GATT
#define APP_SCNRSP_DATA_LEN            (13)

//广播参数配置
/// Advertising channel map - 37, 38, 39
#define APP_ADV_CHMAP                  (0x07)
/// Advertising minimum interval - 100ms (160*0.625ms)
#define APP_ADV_INT_MIN                (160)
/// Advertising maximum interval - 100ms (160*0.625ms)
#define APP_ADV_INT_MAX                (160)
/// Fast advertising interval
#define APP_ADV_FAST_INT               (32)
```

### 2.2 常用的 API 及回调函数

### 2.2.1 启动一个软件定时器

```
/*  
 *  
 * timer_id : 定时器的 ID, 用来区分是哪个定时器  
 * task_id : 任务 ID  
 * Delay : 定时器时间, 单位 10ms  
 *  
 */
```

```
void ke_timer_set(ke_msg_id_t const timer_id, ke_task_id_t const task_id,  
                 uint32_t delay)
```

### 2.2.2 清除一个软件定时器

```
/*  
 *  
 * timer_id : 定时器 ID  
 *  
 * task_id : 任务 ID  
 *  
 */
```

```
void ke_timer_clear(ke_msg_id_t const timer_id, ke_task_id_t const  
                  task_id)
```

### 2.2.3 设备主动发起广播

```
void appm_start_advertising(void)
```

### 2.2.4 设备主动停止广播

```
void appm_stop_advertising(void)
```

### 2.2.5 设备主动断开连接

```
void appm_disconnect(void)
```

### 2.2.6 设备连接成功

```
static int gapc_connection_req_ind_handler(ke_msg_id_t const msgid,  
                                           struct gapc_connection_req_ind const *param,  
                                           ke_task_id_t const dest_id,  
                                           ke_task_id_t const src_id)
```

### 2.2.7 设备成功断开

```
static int gapc_disconnect_ind_handler(ke_msg_id_t const msgid,
                                      struct gapc_disconnect_ind const *param,
                                      ke_task_id_t const dest_id,
                                      ke_task_id_t const src_id)
```

### 2.2.8 数据发送状态

RW 提供 GATT\_CMP\_EVT 事件对 GATT 操作状态进行处理，当发送完数据之后，协议栈会上报 GATT\_CMP\_EVT 事件（在每个 profile 对应的 xxx\_task.c 中），可以在这个事件中判断数据发送的具体状态。

```
static int gattc_cmp_evt_handler(ke_msg_id_t const msgid,
                                 struct gattc_cmp_evt const *param,
                                 ke_task_id_t const dest_id,
                                 ke_task_id_t const src_id)

//Command complete event data structure
struct gattc_cmp_evt
{
    /// GATT request type
    uint8_t operation;
    /// Status of the request
    uint8_t status;
    /// operation sequence number - provided when operation is started
    uint16_t seq_num;
};
```

GAP\_ERR\_NO\_ERROR: 成功  
其他: 失败

## 2.3 外设驱动使用举例

以下通过在 ble\_app\_gatt 这个工程中添加 PWM 驱动来说明如何添加一个驱动到对应的工程项目中：

① 在工程对应的 ble\_3432\_sdk\_ext\_xx\_xxxx\projects\ble\_app\_gatt\config 目录下，打开 user\_config.h 文件，打开对应的宏定义，如下：

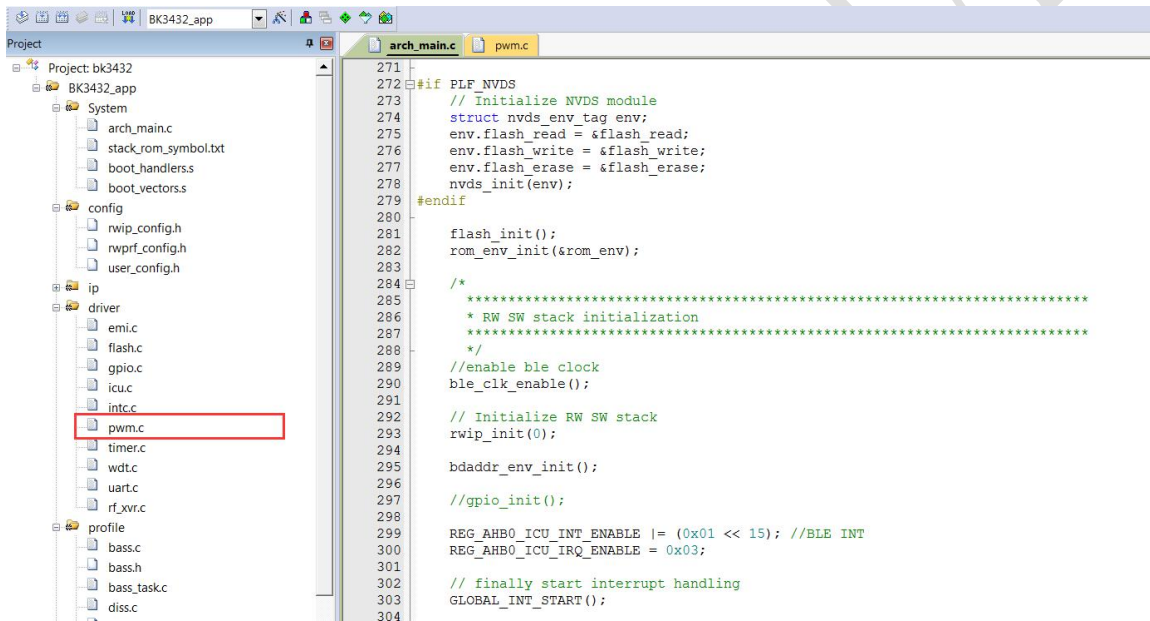
```

/*****
*****
* DRIVER MACRO CTRL *
*****
*****/

//DRIVER CONFIG
#define UART_DRIVER 1
#define GPIO_DRIVER 0
#define AUDIO_DRIVER 0
#define RTC_DRIVER 0
#define IR_DRIVER 0
#define ADC_DRIVER 0
#define I2C_DRIVER 0
#define PWM_DRIVER 1
#define SPI_DRIVER 0
#define UTC_DRIVER 0
#define WDT_DRIVER 0

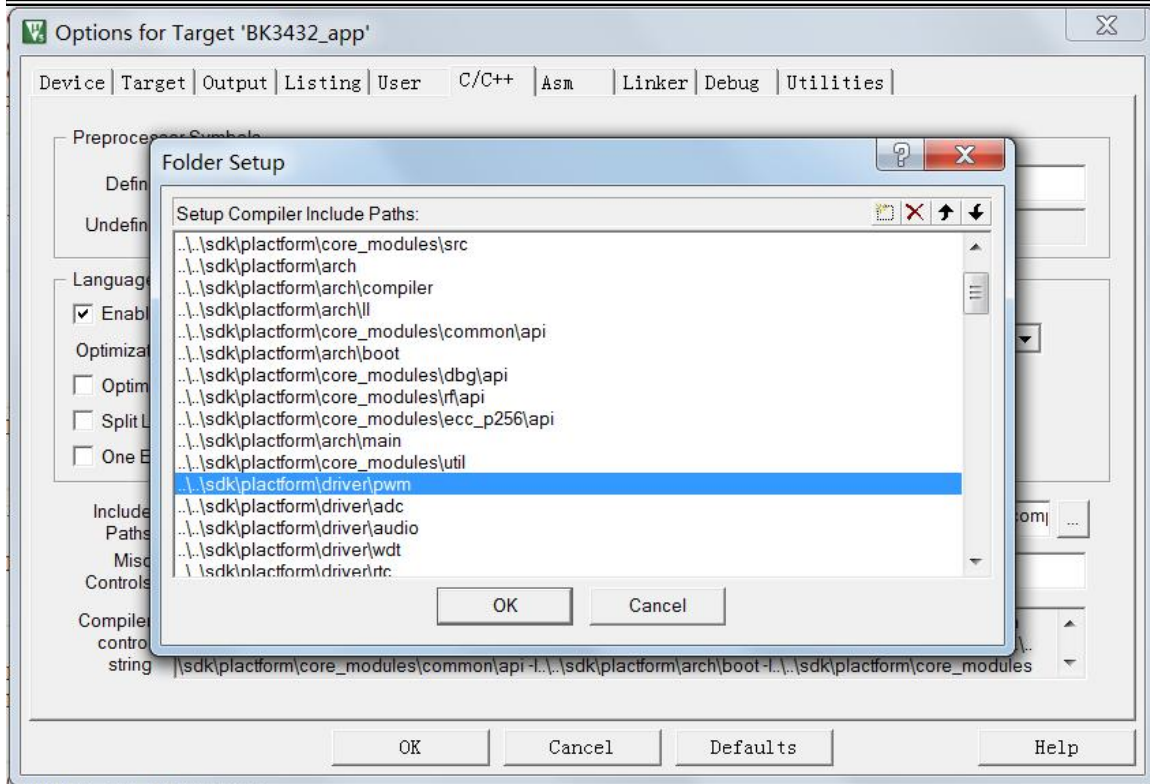
```

② 使用 keil 5.12 打开 ble\_app\_gatt 工程，在 keil 中添加 pwm.c 源文件如下：



③ 在 keil 工程配置中添加相关路径，如下：





④ 编写测试函数，对添加的驱动进行测试，如下：

//硬件timer初始化函数

```
void user_timer_init(void)
```

```
{
```

```
    rwip_prevent_sleep_set(BK_DRIVER_TIMER_ACTIVE); 必须设置
    PWM_DRV_DESC timer_desc;
```

```
    timer_desc.channel = 1; 设置为通道1
```

```
    timer_desc.mode = 1<<0 | 1<<1 | 1<<2 | 0<<4; 设置时钟为32.768K, 打开中断
```

```
    timer_desc.end_value = 600; 定时器时长
```

```
    timer_desc.duty_cycle = 0;
```

```
    timer_desc.p_Int_Handler = user_timer_cb; 定时器中断回调函数
```

```
    pwm_init(&timer_desc);
```

```
}
```

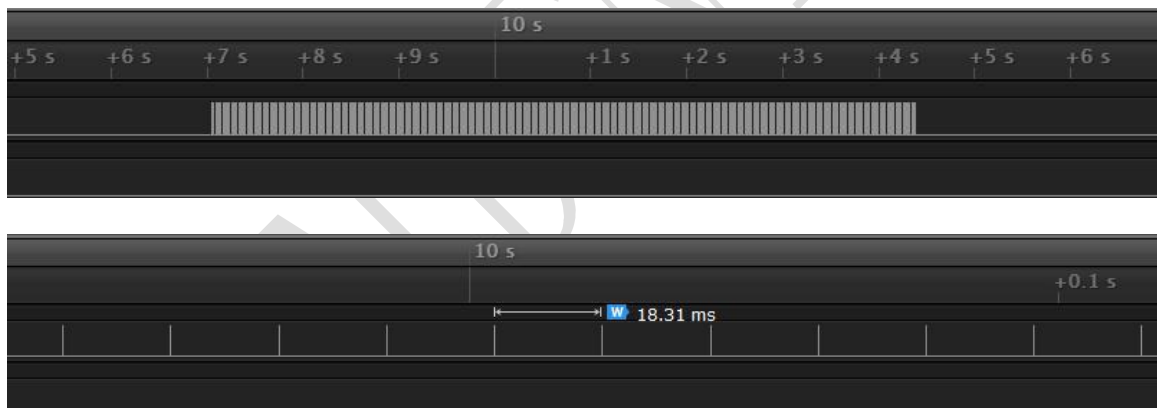
```
//timer中断回调函数
void user_timer_cb(unsigned char ucChannel)
{
    static uint32_t cnt = 0;
    cnt++;
    if(cnt > 1000)
    {
        pwm_disable(ucChannel); 关闭timer
        icu_set_sleep_mode(0); 允许系统进入降压休眠
        rwip_prevent_sleep_clear(BK_DRIVER_TIMER_ACTIVE); 清除标志
        uart_printf("app_timer_cb end %d\r\n", cnt);
    }
}
```

- ⑤ 在 main 函数中添加函数调用，编译工程测试驱动是否正常工作；

```
//这部分是添加的硬件timer测试函数
icu_set_sleep_mode(1); 设置当前系统工作在idle模式下，不允许降压休眠

user_timer_init(); 定时器初始化函数
```

- ⑥ 使用逻辑分析仪验证定时是否正确，如图：



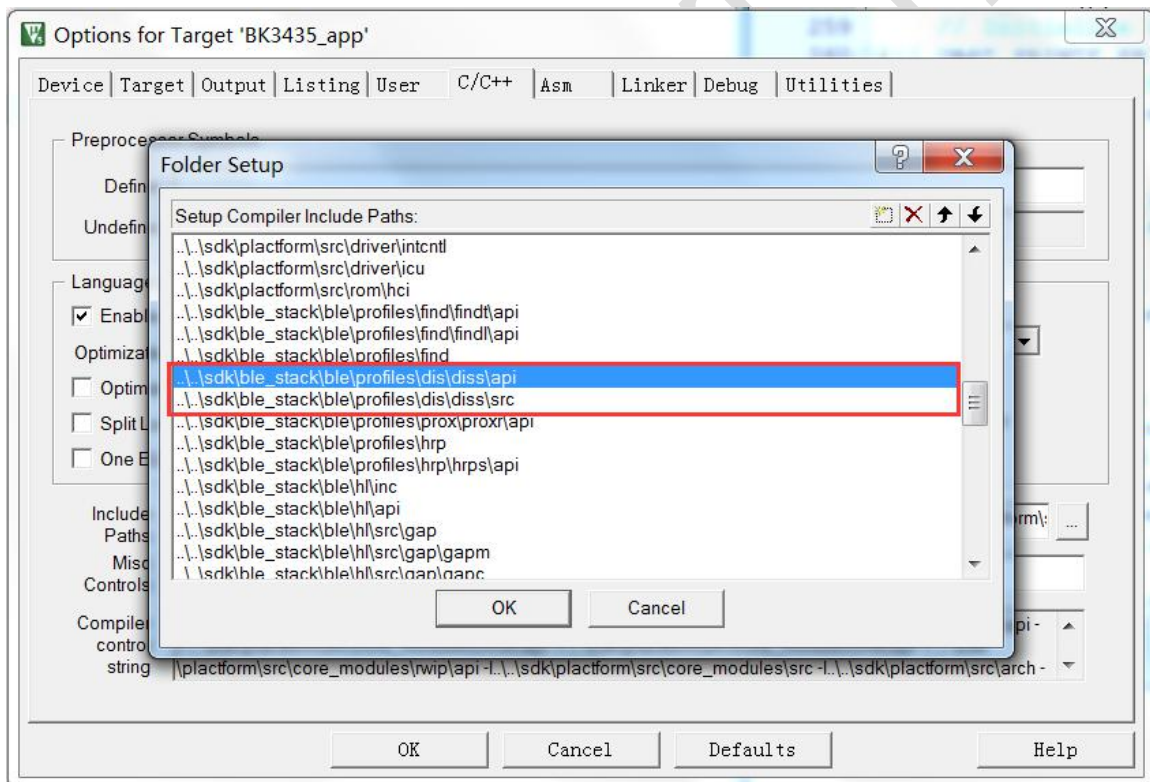
## 2.4 添加一个服务

本部分将通过在 ble\_app\_gatt 工程中添加一个 device information 服务来说明具体的添加步骤。

- ① 使用 keil 5.12 打开对应工程，在工程目录 profile 目录下添加指定 profile 相关源文件，本例中是 diss.c 和 diss\_task.c，如下：



② 在 keil C/C++选项卡下面添加对应 profile 的路径，如下：



③ 在 app\_task.c 文件中找到 appm\_msg\_handler 函数，在其中添加设备信息服务获取 handler 的函数，如下：

```
case (TASK_ID_BASS):
{
    // Call the Battery Module
    msg_pol = appm_get_handler(&app_batt_table_handler, msgid, param, src_id);
} break;
```

④ 在 app.c 文件中的服务列表中添加设备信息服务，如下：

```
/// List of service to add in the database
enum appm_svc_list
{
    APPM_SVC_FFF0,
    APPM_SVC_DIS,
    APPM_SVC_BATT,
    APPM_SVC_OADS,
    APPM_SVC_LIST_STOP ,
};
```

- ⑤ 在 app.c 文件中的函数列表中增加像 db 添加设备信息服务的函数，如下：

```
/// List of functions used to create the database
static const appm_add_svc_func_t appm_add_svc_func_list[APPM_SVC_LIST_STOP] =
{
    (appm_add_svc_func_t)app_fff0_add_fff0s,
    (appm_add_svc_func_t)app_dis_add_dis,
    (appm_add_svc_func_t)app_batt_add_bas,
    (appm_add_svc_func_t)app_oad_add_oads,
};
```

- ⑥ 在 app.c 文件中找到 appm\_init 函数，在其中添加 app\_dis\_init() 函数，如下：

```
// Device Information Module
app_dis_init();

// Battery Module
app_batt_init();

app_oads_init();
```

- ⑦ 打开 rwprf\_config.h 文件，将其中的 CFG\_PRF\_DISS 宏定义设置为 1，如下：

```
// <e> CFG_PRF_DISS
// <i> Battery Service Server Role
// </e>
#if ( 1 )
#define CFG_PRF_DISS
#endif
```

- ⑧ 重新编译工程，通过手机搜索连接，确认设备信息服务是否已经添加成功。

## 2.5 添加一个自定义消息

这部分通过在工程中添加一个连接参数更新的消息来描述在 RW 系统中如何添加一个自定义消息。

- ① 在 app\_task.h 中定义自定义消息 ID，如：APP\_PARAM\_UPDATE\_REQ\_IND。



```
enum appm_msg
{
    APPM_DUMMY_MSG = TASK_FIRST_MSG(TASK_ID_APP),
    APP_PARAM_UPDATE_REQ_IND,
};
```

② 在 const struct ke\_msg\_handler appm\_default\_state[] 中添加自定义消息 ID 及对应的 handler，如下，

```
{APP_PARAM_UPDATE_REQ_IND, (ke_msg_func_t)gapc_update_conn_param_req_ind_handler},
```

③ 实现 gapc\_update\_conn\_param\_req\_ind\_handler 函数

```
static int gapc_update_conn_param_req_ind_handler(ke_msg_id_t const msgid,
const struct gapc_param_update_req_ind *param,
ke_task_id_t const dest_id,
ke_task_id_t const src_id)
{
    UART_PRINTF("slave send param_update_req\r\n");
    struct gapc_conn_param up_param;

    up_param.intv_min = BLE_UAPDATA_MIN_INTVALUE; //最小连接间隔
    up_param.intv_max = BLE_UAPDATA_MAX_INTVALUE; //最大连接间隔
    up_param.latency = BLE_UAPDATA_LATENCY; //latency
    up_param.time_out = BLE_UAPDATA_TIMEOUT; //连接超时

    appm_update_param(&up_param);

    return KE_MSG_CONSUMED;
}
```

④ 在设备建立连接之后，设置一个定时器进行连接参数更新

```
ke_timer_set(APP_PARAM_UPDATE_REQ_IND, TASK_APP, 100);
```

⑤ 连接参数更新状态指示

```
static int gapc_cmp_evt_handler(ke_msg_id_t const msgid,
struct gapc_cmp_evt const *param,
ke_task_id_t const dest_id,
ke_task_id_t const src_id)
{
    UART_PRINTF("gapc_cmp_evt_handler operation = %x\r\n", param->operation);
    switch(param->operation)
    {
        case (GAPC_UPDATE_PARAMS): //0x09
        {
            if(param->status != GAP_ERR_NO_ERROR)
            {
                UART_PRINTF("gapc update params fail !\r\n");
            }
            else
            {
                UART_PRINTF("gapc update params ok !\r\n");
            }
        } break;
    }
```

⑥ 成功更新连接参数后，协议栈会上报 GAPC\_PARAM\_UPDATED\_IND 消息，并将最终更新的连接参数上报到应用层。

```
{GAPC_PARAM_UPDATED_IND, (ke_msg_func_t)gapc_param_updated_ind_handler},

static int gapc_param_updated_ind_handler (ke_msg_id_t const msgid,
                                           const struct gapc_param_updated_ind *param,
                                           ke_task_id_t const dest_id,
                                           ke_task_id_t const src_id)
{
    UART_PRINTF("%s \r\n", __func__);
    UART_PRINTF("con_interval = %d\r\n", param->con_interval);
    UART_PRINTF("con_latency = %d\r\n", param->con_latency);
    UART_PRINTF("sup_to = %d\r\n", param->sup_to);

    return KE_MSG_CONSUMED;
}
```

⑦ 如果是从机端不更新连接参数，主机端更新连接参数，我们可以设定从机端是否接受主机发过来的连接参数。当主机发起连接参数更新请求或，协议栈会像从设备应用层发送 GAPC\_PARAM\_UPDATE\_REQ\_IND 指示，如下，

```
{GAPC_PARAM_UPDATE_REQ_IND, (ke_msg_func_t)gapc_param_update_req_ind_handler},

static int gapc_param_update_req_ind_handler(ke_msg_id_t const msgid,
                                             struct gapc_param_update_req_ind const *param,
                                             ke_task_id_t const dest_id,
                                             ke_task_id_t const src_id)
{
    UART_PRINTF("%s \r\n", __func__);
    // Prepare the GAPC_PARAM_UPDATE_CFM message
    struct gapc_param_update_cfm *cfm = KE_MSG_ALLOC(GAPC_PARAM_UPDATE_CFM,
                                                    src_id, dest_id,
                                                    gapc_param_update_cfm);

    cfm->ce_len_max = 0xffff;
    cfm->ce_len_min = 0xffff;
    cfm->accept = true;

    // Send message
    ke_msg_send(cfm);

    return (KE_MSG_CONSUMED);
} ? end gapc_param_update_req_ind_handler ?
```

其中 cfm->accept = true 表示从设备接受主设备发起的连接参数更新。

## 2.6 配对方式修改

关于配对的相关操作都在 app\_sec.c 这个文件中。

```
static int gapc_bond_req_ind_handler( ke_msg_id_t const msgid,
                                      struct gapc_bond_req_ind const *param,
                                      ke_task_id_t const dest_id,
                                      ke_task_id_t const src_id)
```

在这个函数中对应如下几个参数

```
// Pairing Features
cfm->data.pairing_feat.auth      = GAP_AUTH_REQ_NO_MITM_BOND;
cfm->data.pairing_feat.iocap     = GAP_IO_CAP_NO_INPUT_NO_OUTPUT;
cfm->data.pairing_feat.key_size  = 16;
cfm->data.pairing_feat.oob       = GAP_OOB_AUTH_DATA_NOT_PRESENT;
cfm->data.pairing_feat.sec_req   = GAP_NO_SEC;
cfm->data.pairing_feat.ikey_dist = GAP_KDIST_ENCKEY | GAP_KDIST_IDKEY;
cfm->data.pairing_feat.rkey_dist = GAP_KDIST_ENCKEY | GAP_KDIST_IDKEY;
```

修改了 cfm->data.pairing\_feat.auth 这个参数之后对应在 void app\_sec\_send\_security\_req(uint8\_t conidx) 这个函数中的 cmd->auth 这个参数也要保持一致。

例如在 ble\_app\_ancs 这个程序默认是手机端确认的方式配对，如果需要改成 pincode 配对，那么需要修改如下几个参数：

```
cmd->auth      = GAP_AUTH_REQ_MITM_BOND;
cfm->data.pairing_feat.auth = GAP_AUTH_REQ_MITM_BOND;
cfm->data.pairing_feat.iocap = GAP_IO_CAP_DISPLAY_ONLY;
```

对应的配对 pincode 如下为 123456 就为配对时需要输入的密码，可自行修改：

```
case (GAPC_TK_EXCH):
{
    // Generate a PIN Code- (Between 100000 and 999999)
    uint32_t pin_code = 123456; // (100000 + (co_rand_word() % 900000));
```

## 2.7 程序的烧录

### 2.7.1 输出目录介绍

BK3435 工程编译后的 bin 文件保存在对应工程 output 目录下，如下：

app	2019/1/15 19:15	文件夹	
bim	2019/1/7 18:36	文件夹	
stack	2019/1/7 18:36	文件夹	
BinConvert_3432.exe	2018/4/25 17:14	应用程序	217 KB
encrypt_app.bat	2018/12/10 13:34	Windows 批处理...	1 KB

app: 工程编译后文件目录；

bk3432_ble_app.bin	2019/1/4 16:16	BIN 文件	23 KB
bk3432_ble_app_app.bin	2019/1/4 16:16	BIN 文件	23 KB
bk3432_ble_app_merge.bin	2019/1/4 16:16	BIN 文件	132 KB
bk3432_ble_app_stack.bin	2019/1/4 16:16	BIN 文件	123 KB

- ① bk3432\_ble\_app.bin: 编译生成的 app 部分原始 bin 文件
- ② bk3432\_ble\_app\_app.bin: 生成的升级 app 部分的 bin 文件
- ② bk3432\_ble\_app\_merge.bin: 生成的 bim、stack、app 合并之后的 bin 文件
- ③ bk3432\_ble\_app\_stack.bin: 生成的全部升级的 bin 文件

**bim 及 stack 文件夹中的固件芯片厂商已经编译好，放置在此只为程序链接合并使用，不可随意修改。**

## 2.7.2 程序烧录

打开 HID Download Tool V2.5.2，依次点击上位机左上方下载模式->接口选择->SPI 输出方式-->软件模式，如下：



按如下图设置进行烧录前相关配置，点击下载 flash 按钮即可启动烧录；



烧录过程如下：





## 2.8 OAD

### 2.8.1 空中升级

仅部分应用支持，最大可升级文件大小为 25K，可参考  
BK3432 RC OTA User's Guide V1.0.pdf

### 2.8.2 串口升级

具体参照 ble\_3432\_sdk\_ext\_3x\_XXXX/doc/BK3432 Download by UART User's Guide V2.0.pdf

## 2.9 RF 测试

具体参照 BK3432 RF User's Guide V1.0.pdf

## 3. sdk 例程介绍

### 3.1 数据透传

数据透传例程主要基于 fff0 服务，实现了数据在手机与 3432 端的互传功能。

服务 UUID:0xfff0

UUID	操作权限	功能定义
0xfff1	Read/Notify	Tx

0xff2	Write Without Response	Rx
-------	------------------------	----

数据发送接口: `void app_ffl_send_lvl(uint8_t* buf, uint8_t len);`

数据接收接口: `static int fff2_writer_req_handler(ke_msg_id_t const msgid,  
struct fff0s_fff2_writer_ind *param,  
ke_task_id_t const dest_id,  
ke_task_id_t const src_id)`

该例程主要实现的功能: 3435 与手机连接后, 手机端主动使能 `notify` 属性之后, 3435 会连续发送数据包到手机端, 数据包长度 20 bytes, 数据内容 0xcc。

## 3.2 遥控器

功能描述: 该工程主要实现了一个 HID 遥控器, 可对电视或者机顶盒进行控制。

应用层主要函数功能介绍:

`app_key.c` 及 `app_key.h` 主要实现了按键扫描功能;

`app_sec.c` 及 `app_sec.h` 主要是对配对相关的处理;

`app_hid.c` 及 `app_hid.h` 是 hid 相关接口实现及处理;

`app_task.c` 及 `app_task.h` 是应用层主任务处理函数;

重要的函数说明:

`void app_hid_send_report(uint8_t *data, uint8_t len):` 键值发送函数

`void app_hid_send_sensor_report(uint8_t *sensor_data):` sensor 数据发送函数

`void app_hid_send_mouse_report( struct mouse_msg report ):` 鼠标键值发送函数

## 3.3 苹果通知服务

本工程实现了 ANCS 的功能, 可以与苹果设备连接, 对苹果设备的通知信息进行实时的监听与接收。

## 3.4 微信接入

本工程实现了微信智能硬件接入协议, 实现了 3435 与微信公众号间数据的互相发送与接收。

### 3.4 数据透传（128 BIT UUID）

该数据透传例程主要基于 fcc0 服务，采用 128 bit uuid 实现了数据在手机与 3435 端的互传功能。

服务 UUID: FCC0-0000-0000-1000800000805F9B34FB

UUID	操作权限	功能定义
FCC2-84CFF7E355B46C4C9FD140100A16	Read/Notify	Tx
FCC1-84CFF7E355B46C4C9FD140100A16	Write Without Response	Rx

数据发送接口: void app\_fcc1\_send\_lvl(uint8\_t\* buf, uint8\_t len);

数据接收接口: static int fcc2\_writer\_req\_handler(ke\_msg\_id\_t const msgid,  
struct fcc0s\_fcc2\_writer\_ind \*param,  
ke\_task\_id\_t const dest\_id,  
ke\_task\_id\_t const src\_id)

该例程主要实现的功能: 3432 与手机连接后, 手机端主动使能 notify 属性之后, 3432 会连续发送数据包到手机端, 数据包长度 128 bytes, 数据内容 0xcc。

注意: BASE UUID 可以更改。