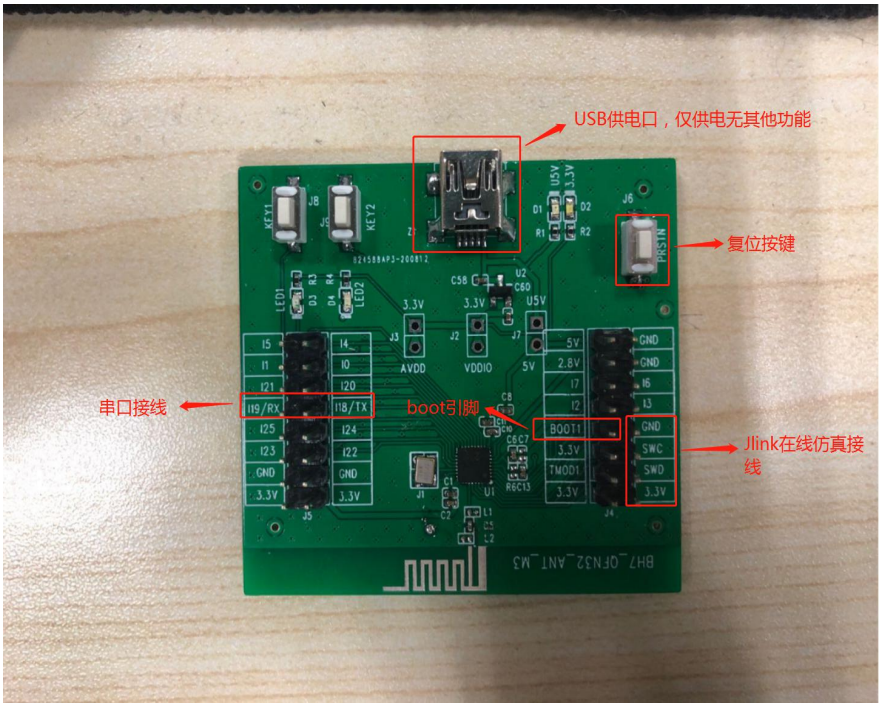
XC 蓝牙系列开发板使用说明

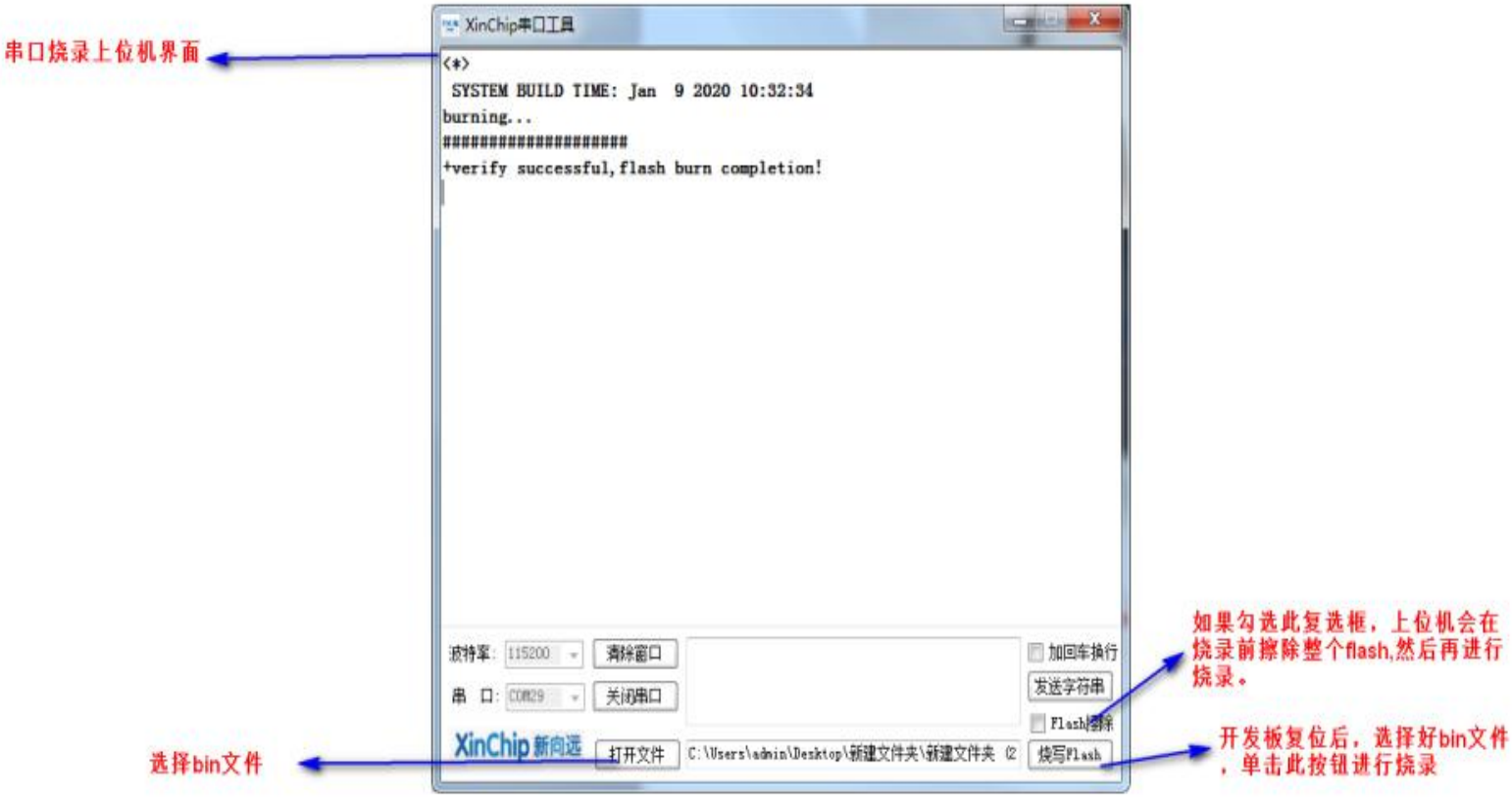
1. 接线说明



（注：所有开发板都可按照这样方式接线，有些开发板上的丝印位置不一样按实际丝印位置进行接线，16pin 开发板上没有复位键可重新上下电进行复位操作）

2. 程序烧写方式

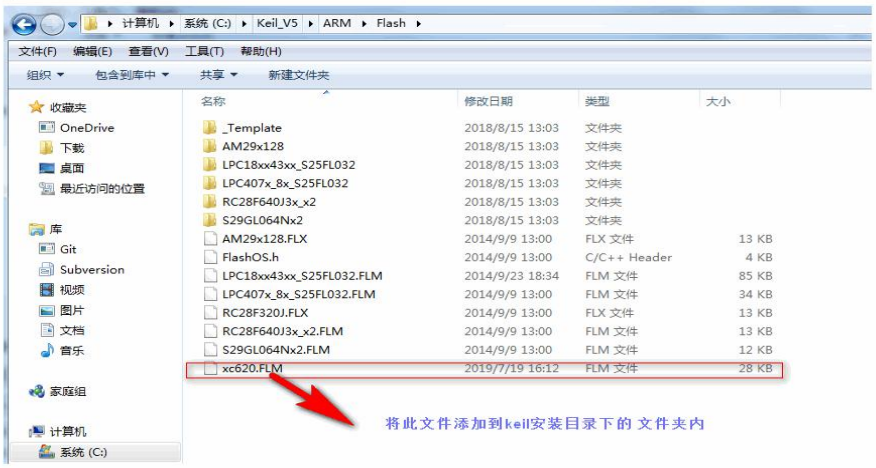
烧写固件前需要将 boot 脚拉高然后按一下复位键，即进入烧写模式（注：进行在线仿真的时候也需要将 boot 脚拉高然后按一下复位键，当重新进行在线仿真时需先重复上述操作）



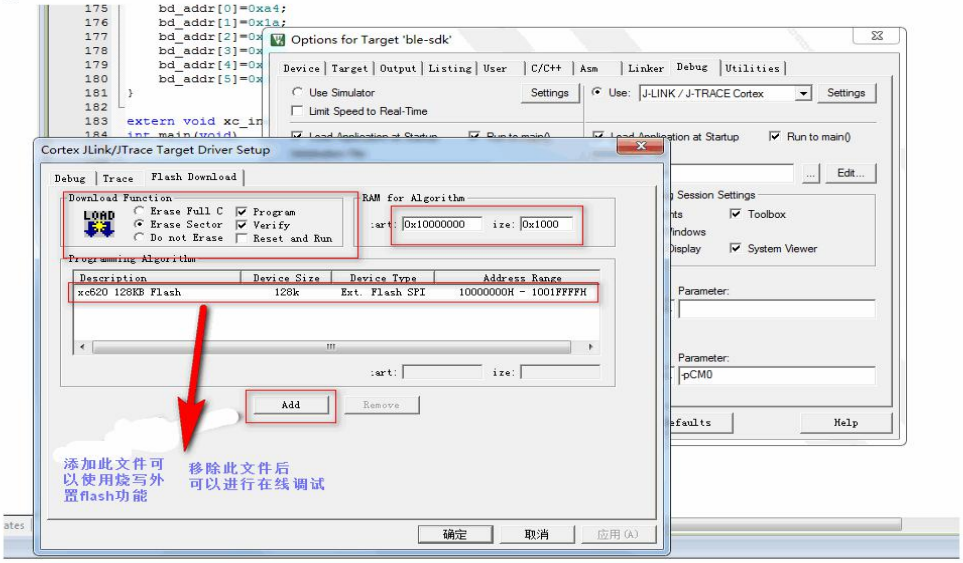
（相关上位机软件在tool 文件内）

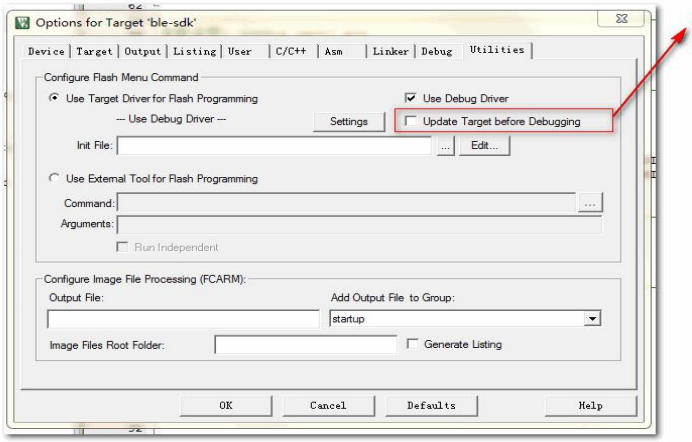
Jlink 烧写：

1.



2.





如果要烧写flash 此处需要勾选；
如果要进行在线仿真调试此处不能勾选

注意：keil工程如果配置成了在线仿真调试 那就不能在进行jlink flash烧录了。

如果配置成了flash烧录 那就不能进行在线仿真调试。

（注：若要进入在线仿真，需要将 FLM 文件 remove）

XC620610 BLE使用指南

一、修改 Mac 地址

在 mian.c 此数组是用来自定义蓝牙的 Mac 地址，需要在 main 函数的最开头调用生效

```
static void set_bd_addr()
{
    extern uint8_t bd_addr[6];

    bd_addr[0]=0xa4;
    bd_addr[1]=0x1a;
    bd_addr[2]=0x02;
    bd_addr[3]=0x20;
    bd_addr[4]=0x7a;
    bd_addr[5]=0x18;
}
```

```
int main(void)
{
    set_bd_addr();
    ble_init((void *)&blestack_init);
    gatt_client_init();
    // setup advertisements
    uint16_t adv_int_min = 0x0030;
    uint16_t adv_int_max = 0x0030;

    bd_addr_t null_addr;
    memset(null_addr, 0, 6);
    gap_advertisements_set_params(adv_in
    gap_advertisements_set_data(adv_data
    gap_scan_response_set_data(scanresp_
    gap_advertisements_enable(1);

    while(1) {
```

二、修改蓝牙名称

在 mian.c 此数组是用来自定义蓝牙的广播名称，0x0A 表示后面内容的字节大小

```
const uint8_t adv_data[] = {
    // Flags general discoverable, BR/EDR not supported
    0x02, 0x01, 0x06,
    // Name
    0x0A, 0x09, 'X', 'i', 'c', '_', 'B', 'L', 'E', 'v', '1',
};
```

三、修改蓝牙设备名称

1.需要先搭建 Python 环境（仅需搭建一个环境），在 XC_SDK\Xinc_ble_sdk_4.0\Ble 路径下的 Xin_ble.gatt 文件下修改红框中的内容后保存

```
PRIMARY_SERVICE, GAP_SERVICE
CHARACTERISTIC, GAP_DEVICE_NAME, READ, "Xinc_BLEv1"

// Test Service
PRIMARY_SERVICE, 0000FF10-0000-1000-8000-00805F9B34FB
// Test Characteristic A, notify
CHARACTERISTIC,  0000FF11-0000-1000-8000-00805F9B34FB, NOTIFY | DYNAMIC,
// Test Characteristic B, write and read
CHARACTERISTIC,  0000FF12-0000-1000-8000-00805F9B34FB, WRITE | READ | DYNAMIC,
```

2. 保存之后点击 profile.bat 脚本文件就会自动生成 profile.h 文件，完成上述操作之后直接编译工程即可

四、设置蓝牙广播间隔和连接间隔

1. 在 main.c 的 main 函数中修改这两个变量可设置蓝牙广播间隔

```
int main(void)
{
    set_bd_addr();
    ble_init((void *)&blestack_init);
    gatt_client_init();
    // setup advertisements
    uint16_t adv_int_min = 0x0030;
    uint16_t adv_int_max = 0x0030;
```

2. 此接口用作修改蓝牙的连接间隔，函数传参说明（连接 handle 值，min_interval，max_interval，latency，timeout_time）

```
gap_request_connection_parameter_update(connection_handle, 0x7, 0x7, 0,0x12);
```

五、蓝牙<->手机传输数据

1. 方向：蓝牙->手机

```
static void data_can_send_now(void * context){
    hci_con_handle_t con_handle = (hci_con_handle_t) (int) context;
    if(u_buff_length>20 && p_buff!=NULL)
    {
        att_server_notify(con_handle, ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE, p_buff, 20);
        p_buff+=20;
        u_buff_length -=20;
        if (user_data_client_configuration){
            user_data_callback.callback = &data_can_send_now;
            user_data_callback.context  = (void*) (int) user_data_client_configuration_connection;
            att_server_register_can_send_now_callback(&user_data_callback, user_data_client_configuration_connection);
        }
    }else{

        att_server_notify(con_handle, ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE, p_buff, u_buff_length);
        p_buff=NULL;
        u_buff_length =0;

    }
}

void data_sent(uint8_t *buffer,uint16_t length)//用户数据发送  ble->手机
{
    p_buff = buffer;
    u_buff_length = length;
    if (user_data_client_configuration){
        user_data_callback.callback = &data_can_send_now;
        user_data_callback.context  = (void*) (int) user_data_client_configuration_connection;
        att_server_register_can_send_now_callback(&user_data_callback, user_data_client_configuration_connection);
    }
}
```

PRIMARY_SERVICE 代表蓝牙服务，CHARACTERISTIC 代表蓝牙服务下的特征值，0000FF11-0000-1000-8000-00805F9B34FB 可以作为服务和特征值的 UUID，蓝牙联盟标准可将 UUID 定义成 128bit 格式和 16bit 格式，128bit 格式参考前面，16bit 可为 FF11.在特征值后可赋予 WRITE、READ、NOTIFY、INDICATE 属性，蓝牙向手机发送数据需要赋予 NOTIFY 或者 INDICATE 属性。生成的特征值 UUID 宏在 profile.h 文件中。

```
PRIMARY_SERVICE, GAP_SERVICE
CHARACTERISTIC, GAP_DEVICE_NAME, READ, "Xinc_BLEv1"

// Test Service
PRIMARY_SERVICE, 0000FF10-0000-1000-8000-00805F9B34FB
// Test Characteristic A, notify
CHARACTERISTIC,  0000FF11-0000-1000-8000-00805F9B34FB, NOTIFY | DYNAMIC,
// Test Characteristic B, write and read
CHARACTERISTIC,  0000FF12-0000-1000-8000-00805F9B34FB, WRITE | READ | DYNAMIC,
```

2. 方向：手机->蓝牙

当手机向蓝牙发送数据时会回调 main.c 中的 att_write_callback,在对应的特征值 case 下可接收到手机发送的数据

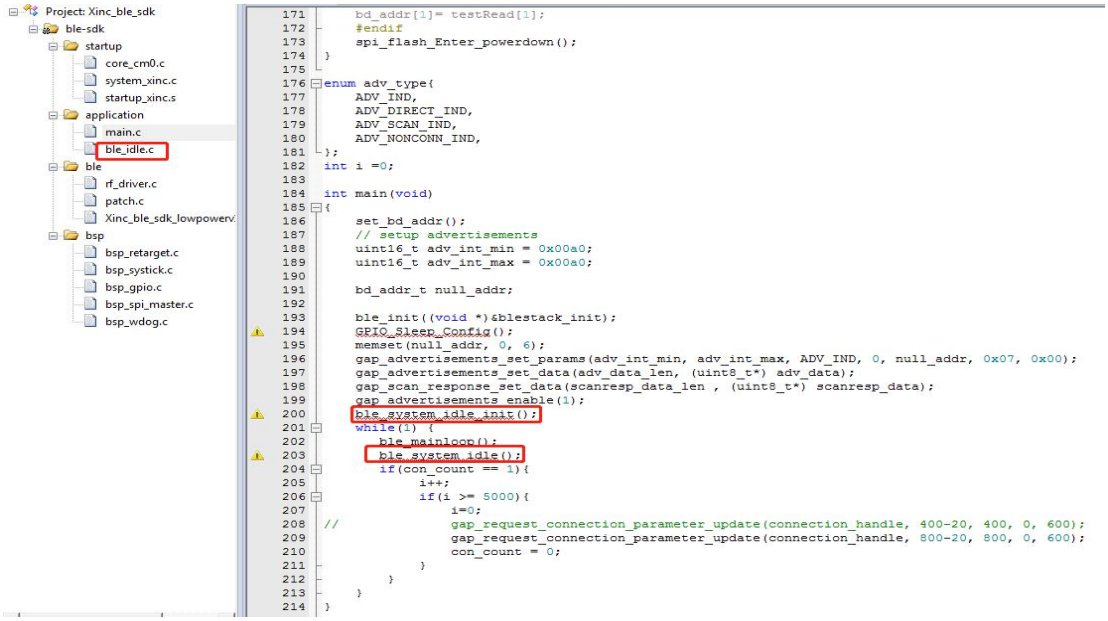
```
// write requests
static int att_write_callback(hci_con_handle_t con_handle, uint16_t att_handle, uint16_t transaction_mode, uint16_t offset, uint8_t *buffer, uint1

    uint32_t le_notification_enabled;
    printf("%s, con_handle=%x, att_handle=%x, offset=%x, buffer=%x, size=%x\n", __func__, con_handle, att_handle, offset, (uint32_t)buffer, buffe
    if (transaction_mode != ATT_TRANSACTION_MODE_NONE) return 0;
    switch(att_handle)
    {
    case    ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_CLIENT_CONFIGURATION_HANDLE:
        le_notification_enabled = little_endian_read_16(buffer, 0) == GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION;
        printf("Notifications enabled %u\n",le_notification_enabled);
        break;
    case    ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE:
    case    ATT_CHARACTERISTIC_0000FF12_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE:
        printf("att_handle=0x%x, offset=0x%x, length=0x%x\n", att_handle, offset, buffer_size);
        printf("att data: ");
        for(uint32_t i=0; i<buffer_size; i++) printf("0x%x ", buffer[i]);
        printf("\n");


        le_data_down(buffer,buffer_size);
        break;
    default:
        break;
    }
    return 0;
}
```

当手机读取蓝牙数据时回调 main.c 中的 att_read_callback,在对应的特征值 case 下可读取自定义数据

```
// read requests
static uint16_t att_read_callback(hci_con_handle_t con_handle, uint16_t att_handle, uint16_t offset, uint8_t *buffer, uint16_t buffer_size){

    printf("%s, con_handle=%x, att_handle=%x, offset=%x, buffer=%x, size=%x\n",__func__, con_handle, att_handle, offset,(uint32_t)buffer, buffer_size);

    if((att_handle != ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE) &&
       (att_handle != ATT_CHARACTERISTIC_0000FF12_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE)) return 0;

    static uint8_t read_test[10] = "012345678";
    return att_read_callback_handle_blob((const uint8_t *)read_test, sizeof(read_test), offset, buffer, buffer_size);
}
```

六、低功耗模式使用说明

1. 需要在 BTlowpower_Demo 工程上进行调试



2. 一般情况睡眠机制以广播间隔和连接间隔为基准进行睡眠和唤醒，如若需要随意切换睡眠和唤醒状态需进行以下操作：

打开唤醒屏蔽源

```
void    ble_system_idle_init(void)
{
      __write_hw_reg32(CPR_SYS_TIME, ((RST_READY_TIME<<12) | OSC32_STABLE_TIME));


      __write_hw_reg32(CPR_SLP_CTL, 0x00);
      __write_hw_reg32(CPR_SLPCTL_INT_MASK, 0xFFFFFfBF);
    //__write_hw_reg32(CPR_SLPCTL_INT_MASK, 0xFFFFFFFF);
    //__write_hw_reg32(CPR_SLP_PD_MASK, 0x80);
     __write_hw_reg32(CPR_SLP_PD_MASK,0x101);
      __write_hw_reg32(CPR_SLP_SRC_MASK, 0x60006);

      *((volatile unsigned *)(CPR_AO_BASE + 0x40)) &=0x0F;            //ROM断电
      *((volatile unsigned *)(CPR_AO_BASE + 0x44)) |=0x10;            //ROM断电隔离
      *((volatile unsigned *)(CPR_AO_BASE + 0x20))  =0x2C;            //auto switch core ldo voltage
      Timer_Register_Callback(system_wakeup, 1);

      /*gpio_direction_output(20);*/
}
```

2.1 M0 中断分配

M0 支持 20 个 IRQ 中断源，如表 2-1 所示。

表 2-1 M0 中断分配表

中断源	中断信号	中断序号分配	缺省中断优先级
IRQ			
BT	tab_intr_n	0	2
DMAS	dmas_int	1	2
CPR	cpr_intr	2	2
GPIO	gpio_intr	3	2
RTC	rtc_intr	4	2
TIMER0	timer0_intr	5	2
TIMER1	timer1_intr	6	2
TIMER2	timer2_intr	7	2
TIMER3	timer3_intr	8	2
WDT	wdt_intr	9	2
I2C	i2c_intr	10	2
UART0	uart0_intr	11	2
UART1	uart1_intr	12	2
SSI0	ssi0_intr	13	2
SSI1	ssi1_intr	14	2
KBS	kbs_intr	15	2
QDEC	qdec_intr	16	2
GPADC	gpadc_intr	17	2
SIM	sim_intr	18	2
AES	aes_intr	19	2

在代码库中有定义睡眠、唤醒标志位，广播态：adv_flag，连接态：con_flag，当标志位置 1 时蓝牙之后的广播间隔和连接间隔都将被唤醒不再睡眠，只有当标志位置 0 后的蓝牙的广播间隔和连接间隔都重新以两种状态下的间隔基准睡眠。

注：（不要以串口中断为唤醒源，晶振起振需要一定时间，否则可能会导致丢失数 据）

七、如何判断蓝牙是否连接及如何断开蓝牙

1.判断蓝牙连接

在 main 函数的 packe_handler 接口中可以看出，如下图红框中所示

```
static void packet_handler (uint8_t packet_type, uint16_t channel, uint8_t *packet, uint16_t size){
    if (packet_type != HCI_EVENT_PACKET) return;
    switch(hci_event_packet_get_type(packet))
    {
    case BTSTACK_EVENT_STATE:
        if (btstack_event_state_get_state(packet) != HCI_STATE_WORKING) return;
        printf("BTstack up and running.\n");
        break;

    case HCI_EVENT_LE_META:
        switch (hci_event_le_meta_get_subevent_code(packet))
        {
        case HCI_SUBEVENT_LE_CONNECTION_COMPLETE: {
                hci_con_handle_t connection_handle = hci_subevent_le_connection_complete_get_connection_handle(packet);
                printf("\n CONNECT RIGHT ! (HANDLE = 0x%x)\n", connection_handle);
                        g_conn_stat = 1;
//                          gap_request_connection_parameter_update(connection_handle, 0x7, 0x7, 0,0x12);
                printf("Connected, requesting conn param update for handle 0x%04x\n", connection_handle);
                gatt_client_read_value_of_characteristics_by_uuid16(packet_handler,connection_handle,0x01,0xff,0x2A00);
            }
            user_start();
            break;
        default:
            break;
        }
        break;
```

3. 断开蓝牙调用如下接口即可

```
void hci_disconnect_all(void);
```

八、设置蓝牙发射功率

在 rf_driver.c 中修改如下参数：

```
#ifdef QFN32
    wbit(0x0054,12,7,"101100");// txgm gc qfn32
#elif defined SSOP16
    wbit(0x0054,12,7,"100100");// txgm gc ssop16
#endif
```

发射功率配置：

用 05 寄存器的 RF_PWR ,<21:16>来控制发射功率，111111 最大，000001 最低。

RF_PWR	16 进制	发射功率（dBm）	说明
111111	0x3f	10.6	
111000	0x38	9.4	
110100	0x34	8.6	
110000	0x30	7.7	
101100	0x2c	6.8	250K 默认配置
101010	0x2a	6.3	
101000	0x28	5.7	
100100	0x24	4.5	
100000	0x20	3.2	
010100	0x14	2	
010000	0x10	0	
001100	0xC	-2.4	
001000	0x8	-6	
000100	0x4	-12	
000010	0x2	-18	
000001	0x1	-24	

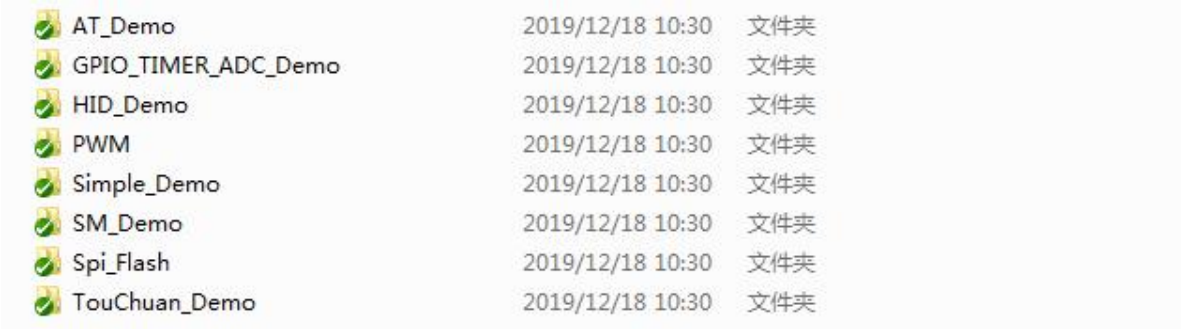
九、调整晶振电容阵列寄存器

位数	名称	方向	复位值	描述
31	bb_ldo_adda_bbvco_en	RW	0x0	
25:20	rg_dcxo_ctune	RW	0x3f	
17:12	rg_dcxo_gmtune	RW	0x3F	
11:10	rg_ism_lvshift_ldo_out	RW	0x2	
9:8	rg_clk_path_ldo_vout	RW	0x2	
6:4	rg_clk_path_vddres	RW	0x2	
2:0	rg_clk_bb_drive	RW	0x0	

((u_int32 volatile)0x40002430) = (u_int32)(0x3f3f820); 25:20 用来配置晶振电容阵列，111111 最大，000000 最小，码值加 1，频偏-3KHz（2.4GHz）。建议使用 32M 12pF 晶振（25:20=3f）。如果配成 3f 后还有比较大的+频偏(>20K)，可以在 板子上 qfn32 的 23 脚或者 ssop16 的 13 脚（靠近电源的那个脚）加一个小电容。（该寄存器在 patch.c 内修改）

XC620610 SDK介绍

工程 demo 说明



AT	AT 指令应用
BSP	gpio、timer、ADC 例程
HID	键盘应用例程
OTA	OTA 升级应用
PWM	PWM 例程
Simple	Ble 演示例程
SM	加密演示例程
Spi	SPI 通信例程
touchuan	蓝牙数据透传演示例程

XC620 为裸机系统主要是以中断、定时器方式来自定义任务的

蓝牙透传接口

```
int att_server_indicate(hci_con_handle_t con_handle, uint16_t
                        attribute_handle, const uint8_t *value,
                        uint16_t value_len);
```

相应的 demo 函数如下图所示：

```
#ifdef INDICATE_TEST
void indicate_test(void)
{
    static uint32_t loop = 0;
    loop++;
    if(loop == 0x10000)
    {
        loop = 0;
        att_server_indicate(0x10,
                            ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE,
                            (uint8_t *)"123",
                            3);
    }
}
```

模块中带有 INDICATE 属性的服务可以主动向 APP 端发送数据，其数据内容用户可自定义

调用方式：须在 main 函数里的循环体里调用相关接口，例子如下图

```
int main(void)
{

    ble_init((void *)&blestack_init);
  xc_init();

  // setup advertisements
    uint16_t adv_int_min = 0x0030;
    uint16_t adv_int_max = 0x0030;
    uint8_t adv_type = 0;
    bd_addr_t null_addr;
    memset(null_addr, 0, 6);
    gap_advertisements_set_params(adv_int_min, adv_int_max, adv_type, 0, null_addr, 0x07, 0x00);
    gap_advertisements_set_data(adv_data_len, (uint8_t*) adv_data);
    gap_scan_response_set_data(scanresp_data_len , (uint8_t*) scanresp_data);
    gap_advertisements_enable(1);

    while(1) {
        ble_mainloop();
#if (XC_UART1==1)
  if(queue_empty()==0) at_done(queue_read());
#endif
#ifdef INDICATE_TEST
        indicate_test();
#endif
    //xc_wdog_feed();//喂狗
    }
}
```

APP 端写数据回调函数接口：

```
static int att_write_callback(hci_con_handle_t con_handle, uint16_t
                              att_handle, uint16_t transaction_mode,
                              uint16_t offset, uint8_t *buffer,
                              uint16_t buffer_size)
```

通过此接口可以向模块发送数据

相应的 demo 函数如下图所示：

```
static int att_write_callback(hci_con_handle_t con_handle, uint16_t att_handle, uint16_t transaction_mode, uint16_t offset, uint8_t *buffer, uint16_t buffer_size){

    uint32_t le_notification_enabled;
    printf("%s, con_handle=%x, att_handle=%x, offset=%x, buffer=%x, size=%x\n", __func__, con_handle, att_handle, offset, (uint32_t)buffer, buffer_size);
    if (transaction_mode != ATT_TRANSACTION_MODE_NONE) return 0;
    switch(att_handle)
    {
    case    ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_CLIENT_CONFIGURATION_HANDLE:
        le_notification_enabled = little_endian_read_16(buffer, 0) == GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION;
        printf("Notifications enabled %u\n",le_notification_enabled);
        break;
    case    ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE:
    case    ATT_CHARACTERISTIC_0000FF12_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE:
        printf("att_handle=0x%x, offset=0x%x, length=0x%x\n", att_handle, offset, buffer_size);
        printf("att data: ");
        for(uint32_t i=0; i<buffer_size; i++) printf("0x%x ", buffer[i]);
        printf("\n");
      Uart_Send_Buf(1, "123",3);
#ifdef  INDICATE_TEST
   Uart_Send_Buf(0,buffer,buffer_size);
#endif
        break;
    default:
        break;
    }
    return 0;
}
```

Notify 函数接口：

int att_server_notify(hci_con_handle_t con_handle, uint16_t attribute_handle, const uint8_t *value, uint16_t value_len);

客户端可以通过修改 profile 属性来对模块进行写操作，通过此接口服务器可以主

动向客户端发起通知。

例如：权限是 GATT_PROP_NOTIFY | GATT_PROP_WRITE 即可通知，可写。（即客户端可以往特征值里写入数据，不能主动读取数据，服务器可以随时通知客户端此特征的特征值是什么内容。）

Profile 生成方式

XC620 低功耗蓝牙软件开发包中的 SCAN_RESPONSE、SERVICES 数据都是通过 Xinc_ble_sdk\ble\下的 profile.bat 工具自动生成的，用户只需要按一定规则 编辑 Xinc_ble.gatt 文件，工具将把该文本文件转成代码使用的 profile.h 头文件，并将该.h 头文件自动复制到 Xinc_ble_sdk\ble \目录下。Xinc_ble_sdk\ble \目录下如图 1 所示的 profile.h 文件都是由工具生成的。

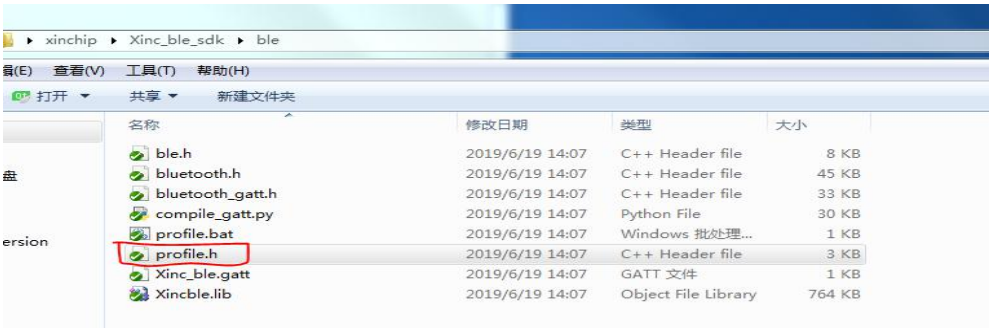


图 1 工具生成的.h 头文件

本文档主要对如何使用 $Xinc_ble_sdk\ble \ 下的工具生成 SCAN RESPONSE 及 SERVICES 的.h 头文件的方法，进行详细说明。此套应用层工具可运行于 WINDOWS XP 及更高版本环境下。$Project\Tools 里的工具由 Python 实现，因此请安装 Python3.6 及以上版本。

配置生成方法：

生成 SERVICES 数据方法

配置文件内容说明

根据 BLE 协议表述，一个 SERVICE 定义可由 SERVICE 和 CHARACTERISTIC 两种声明类型组成。在 XC620 低功耗蓝牙软件开发包中，提供了一套文本化的编辑工具，使用户通过直观的文本编辑就可以完成对 SERVICE 定义数据进行配置。

该配置文本内容由多行组成，每一行代表一个 SERVICE 声明或者一

个 CHARACTERISTIC 声明或者一个 DESCRIPTOR 声明。依据 BLE 协议定义，每一个 SERVICE 定义又是由一个 SERVICE 声明及至少一个 CHARACTERISTIC 声明组成。一个配置文本内可以有一个或多个 SERVICE 定义。DESCRIPTOR 声明是用来修饰 CHARACTERISTIC 声明的，故位于其要修饰的 CHARACTERISTIC 之后。

具体 SERVICE、CHARACTERISTIC 及 DESCRIPTOR 声明的编辑格式如下：

[SERVICE 类型关键字], [SERVICE UUID]

[CHARACTERISTIC 类型关键字], [CHARACTERISTIC UUID], [特性值], [数据值]

[DESCRIPTOR 类型关键字], [特性值] ,[数据值]

Profile 配置文件格式样例：

```
PRIMARY_SERVICE, GAP_SERVICE
CHARACTERISTIC, GAP_DEVICE_NAME, READ, "Xinc_BLEv1"

// Test Service
PRIMARY_SERVICE, 0000FF10-0000-1000-8000-00805F9B34FB
// Test Characteristic A, notify

CHARACTERISTIC,  0000FF11-0000-1000-8000-00805F9B34FB, NOTIFY | DYNAMIC

// Test Characteristic B, write and read
CHARACTERISTIC,  0000FF12-0000-1000-8000-00805F9B34FB, WRITE | READ |
 DYNAMIC
```

表 2：SERVICE 类型取值表

SERVICE 类型关键字	**协议定义 UUID**
PRIMARY_SERVICE	0x2800
SECONDARY_SERVICE	0x2801
INCLUDE_SERVICE	0x2802

表 3：SERVICE UUID 取值表

SERVICE UUID	**协议定义 UUID**
GAP_SERVICE	0x1800
GATT_SERVICE	0x1801
GATT_IA_SERVICE	0x1802
GATT_LL_SERVICE	0x1803
GATT_TP_SERVICE	0x1804
GATT_HT_SERVICE	0x1809
GATT_DI_SERVICE	0x180a
GATT_HR_SERVICE	0x180d
GATT_BAT_SERVICE	0x180f
GATT_HID_SERVICE	0x1812
GATT_SP_SERVICE	0x1813

自定义的 UUID-128 字符串	自定义 FF11 段，字符串格式如下： 0000**FF11**-0000-1000-8000-00805F9B34FB

表 4：CHARACTERISTIC 类型取值表

CHARACTERISTIC 类型关键字	**协议定义 UUID**
CHARACTERISTIC	0x2803

表 5：CHARACTERISTIC UUID 取值表

CHARACTERISTIC UUID	**协议定义 UUID**
GAP_DEVICE_NAME	0x2a00
GAP_APPEARANCE	0x2a01
GAP_PERIPHERAL_PRIVACY_FLAG	0x2A02
GAP_RECONNECTION_ADDRESS	0x2A03
GAP_PERIPHERAL_PREFERRED_CONNECTION_PARAMETERS	0x2A04
GATT_SERVICE_CHANGED	0x2a05
GATT_ALERT_LEVEL	0x2a06
GATT_TXPOWER_LEVEL	0x2a07
GATT_BATTERY_LEVEL	0x2a19
GATT_SYSTEM_ID_STRING	0x2a23
GATT_MODEL_NUM_STRING	0x2a24
GATT_SERIAL_NUM_STRING	0x2a25
GATT_HW_REV_STRING	0x2a27
GATT_FW_REV_STRING	0x2a26
GATT_SW_REV_STRING	0x2a28
GATT_MANU_NAME_STRING	0x2a29
GATT_IEEE_RCDL_STRING	0x2a2a
GATT_PNP_ID_STRING	0x2a50
GATT_PROTOCOL_MODE	0x2A4E
GATT_REPORT	0x2A4D
GATT_REPORT_MAP	0x2A4B
GATT_BOOT_KEYBOARD_INPUT_REPORT	0x2A22
GATT_BOOT_KEYBOARD_OUTPUT_REPORT	0x2A32
GATT_BOOT_MOUSE_INPUT_REPORT	0x2A33
GATT_HID_INFORMATION	0x2A4A
GATT_HID_CONTROL_POINT	0x2A4C
GATT_SCAN_INTERVAL_WINDOW	0x2A4F
GATT_SCAN_REFRESH	0x2A31
自定义的 UUID-128 字符串	自定义 FF11 段，字符串格式如下： 0000**FF11**-0000-1000-8000-00805F9B34FB
自定义 UUID-16 字符串	四位 16 进制，例如： FF11 或者 0xFF11

表 6：CHARACTERISTIC 特征值表

CHARACTERISTIC 特征值关键字	**协议定义 UUID**
BROADCAST	0x01
READ	0x02
WRITE_WITHOUT_RESPONSE	0x04
WRITE	0x08
NOTIFY	0x10
INDICATE	0x20
AUTHENTICATED_SIGNED_WRITE	0x40
EXTENDED_PROPERTIES	0x80
DYNAMIC	*CHARACTERISTIC VALUE 为动态值；当 Server 的该值由 Client 写入时，设置 此特征值，Value 为空；否则，Value 为静 态值；

表 7：DESCRIPTOR 类型取值表

DESCRIPTOR 类型关键字	**协议定义 UUID**
CHARACTERISTIC_USER_DESCRIPTION	0x2901
SERVER_CHARACTERISTIC_CONFIGURATION	0x2903
CHARACTERISTIC_FORMAT	0x2904
CHARACTERISTIC_AGGREGATE_FORMAT	0x2905
REPORT_REFERENCE	0x2908
EXTERNAL_REPORT_REFERENCE	0x2907

CHARACTERISTIC 和 DESCRIPTOR 的数据取值的填写可以是如下三种类型之一：

1) 字符串：用双引号包括的字符串； 如：
"this is a string"

2) 16 进制数字：以 0x 开头的 16 进制数字；
如：03 02 01 （means 0x010203）

3) 16进制队列：每个字节以空格分开的数字队列； 如：
0x010203

4.2 生成编译头文件

✧ **1) 新建 SERVICE 配置文件：**

进入 Xinc_ble_sdk\ble\目录，建立一个 新的 gatt后缀的文件，例如 new_gatt.gatt，打开 new_gatt.gatt，按照 表 2~表 7 的定义，编辑其内容。

✧ **2) 新建执行脚本文件：**

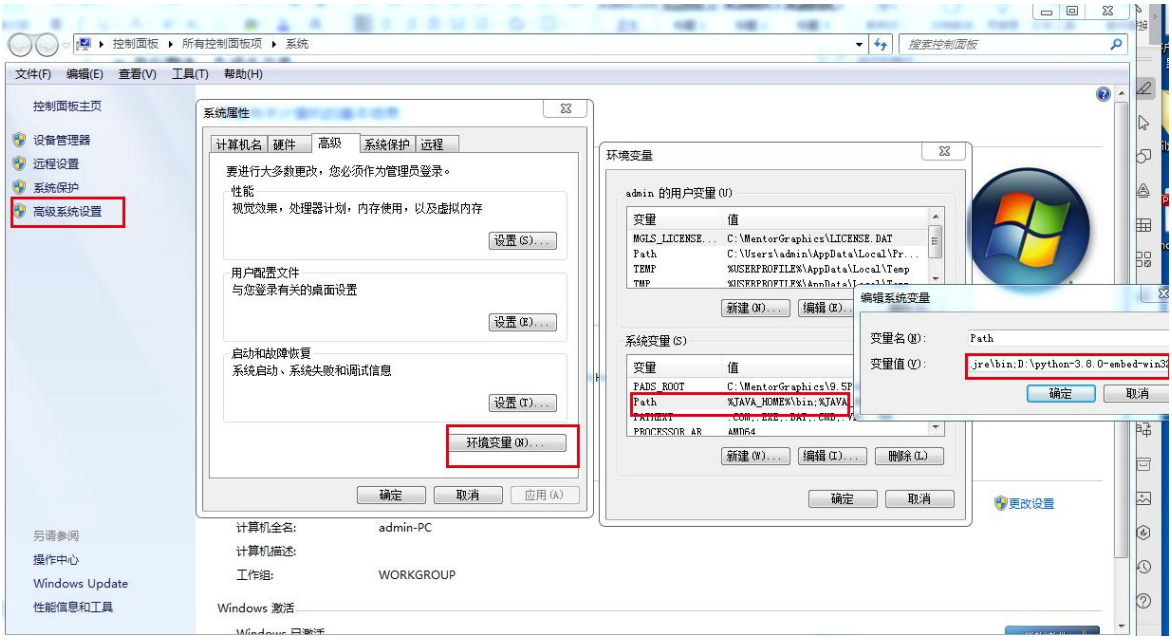
进入 Xinc_ble_sdk\ble\目录，建立一个新的 bat 后缀的文件，例如 new_profile.bat，打开 new_profile.bat，修改其中 内容为：

```
python compile_gatt.py -I .\ Xinc_ble.gatt profile.h
pause
```

✧ **3) 执行脚本，生成头文件：**

双击 profile.bat，即自动生成 SERVICE 数据头文件 profile.h

如何在 Windows 系统中搭建 Python 开发环境？



将 Python 文件夹所在路径添加到环境变量中后打开 Windows 命令行窗口输入 Python 查看是否有 Python 版本即可。

IO 配置使用说明

芯片的 IO 口都可自由配置复用功能，以下说明都是驱动文件（bsp_gpio.c）内默认配置的复用功能，具体配置需求可按实际情况而定。

一、复用功能选择

0: GPIO_Dx 普通 IO 口
1: UART0_TX　　2: UART0_RX
3: UART0_CTS　　4: UART0_RTS
5: I2C_SCL　　6: I2C_SDA
7: UART1_RX　　8: UART1_TX
9: SIM_IO　　10: SIM_RST　　11: SIM_CLK_OUT
12: PWM0　　13: PWM1
14: SSI1_CLK　　15: SSI1_SSN　　16: SSI1_RX　　17: SSI1_TX
18:PWM0_INV　　19:PWM1_INV

```
gpio_fun_sel_config_t  gpio_fun_sel_config = {

/*CPR_GPIO_FUN_SEL0*/.fun_sel0.bits.b0004=UART1_TX,/*GPIO 0*/.fun_sel0.bits.b0812=UART1_RX,/*GPIO 1*/.fun_sel0.bits.b1620= GPIO_Dx,/*GPIO 2*/.fun_sel0.bits.b2428= GPIO_Dx,/*GPIO 3*/
/*CPR_GPIO_FUN_SEL1*/.fun_sel1.bits.b0004= GPIO_Dx,/*GPIO 4*/.fun_sel1.bits.b0812= GPIO_Dx,/*GPIO 5*/.fun_sel1.bits.b1620= GPIO_Dx,/*GPIO 6*/.fun_sel1.bits.b2428= GPIO_Dx,/*GPIO 7*/
/*CPR_GPIO_FUN_SEL2*/.fun_sel2.bits.b0004= GPIO_Dx,/*GPIO 8*/.fun_sel2.bits.b0812= GPIO_Dx,/*GPIO 9*/.fun_sel2.bits.b1620= GPIO_Dx,/*GPIO10*/.fun_sel2.bits.b2428= GPIO_Dx,/*GPIO11*/
/*CPR_GPIO_FUN_SEL3*/.fun_sel3.bits.b0004= GPIO_Dx,/*GPIO12*/.fun_sel3.bits.b0812=SSI1_CLK,/*GPIO13*/.fun_sel3.bits.b1620=SSI1_SSN,/*GPIO14*/.fun_sel3.bits.b2428= SSI1_RX,/*GPIO15*/
/*CPR_GPIO_FUN_SEL4*/.fun_sel4.bits.b0004= SSI1_TX,/*GPIO16*/.fun_sel4.bits.b0812=    PWM0,/*GPIO17*/.fun_sel4.bits.b1620=UART0_TX,/*GPIO18*/.fun_sel4.bits.b2428=UART0_RX,/*GPIO19*/
/*CPR_GPIO_FUN_SEL5*/.fun_sel5.bits.b0004= GPIO_Dx,/*GPIO20*/.fun_sel5.bits.b0812= GPIO_Dx,/*GPIO21*/.fun_sel5.bits.b1620= GPIO_Dx,/*GPIO22*/.fun_sel5.bits.b2428= GPIO_Dx,/*GPIO23*/
/*CPR_GPIO_FUN_SEL6*/.fun_sel6.bits.b0004= GPIO_Dx,/*GPIO24*/.fun_sel6.bits.b0812=    PWM1,/*GPIO25*/.fun_sel6.bits.b1620= GPIO_Dx,/*GPIO26*/.fun_sel6.bits.b2428= GPIO_Dx,/*GPIO27*/
/*CPR_GPIO_FUN_SEL7*/.fun_sel7.bits.b0004= GPIO_Dx,/*GPIO28*/.fun_sel7.bits.b0812= GPIO_Dx,/*GPIO29*/.fun_sel7.bits.b1620= GPIO_Dx,/*GPIO30*/.fun_sel7.bits.b2428= GPIO_Dx /*GPIO31*/
};
```

在图示数组内选择对应的 IO 口写入复用功能配置选项（0-17）

二、IO 中断模式选择

0:NOT_INT　　-GPIO 关闭中断
5:RIS_EDGE_INT　- GPIO 上升沿中断
7:FAIL_EDGE_INT　-GPIO 下降沿中断

```
interrupt_config_t  interrupt_config = {
/*GPIO_INTR_CTRL0*/.intr_ctl0.pad.mode3=NOT_INT,/*GPIO 3*/.intr_ctl0.pad.mode2=NOT_INT,/*GPIO 2*/.intr_ctl0.pad.mode1=NOT_INT,/*GPIO 1*/.intr_ctl0.pad.mode0=NOT_INT,/*GPIO 0*/
/*GPIO_INTR_CTRL1*/.intr_ctl1.pad.mode3=NOT_INT,/*GPIO 7*/.intr_ctl1.pad.mode2=NOT_INT,/*GPIO 6*/.intr_ctl1.pad.mode1=NOT_INT,/*GPIO 5*/.intr_ctl1.pad.mode0=NOT_INT,/*GPIO 4*/
/*GPIO_INTR_CTRL2*/.intr_ctl2.pad.mode3=NOT_INT,/*GPIO11*/.intr_ctl2.pad.mode2=NOT_INT,/*GPIO10*/.intr_ctl2.pad.mode1=NOT_INT,/*GPIO 9*/.intr_ctl2.pad.mode0=NOT_INT,/*GPIO 8*/
/*GPIO_INTR_CTRL3*/.intr_ctl3.pad.mode3=NOT_INT,/*GPIO15*/.intr_ctl3.pad.mode2=NOT_INT,/*GPIO14*/.intr_ctl3.pad.mode1=NOT_INT,/*GPIO13*/.intr_ctl3.pad.mode0=NOT_INT,/*GPIO12*/
/*GPIO_INTR_CTRL4*/.intr_ctl4.pad.mode3=NOT_INT,/*GPIO19*/.intr_ctl4.pad.mode2=NOT_INT,/*GPIO18*/.intr_ctl4.pad.mode1=NOT_INT,/*GPIO17*/.intr_ctl4.pad.mode0=NOT_INT,/*GPIO16*/
/*GPIO_INTR_CTRL5*/.intr_ctl5.pad.mode3=NOT_INT,/*GPIO23*/.intr_ctl5.pad.mode2=NOT_INT,/*GPIO22*/.intr_ctl5.pad.mode1=NOT_INT,/*GPIO21*/.intr_ctl5.pad.mode0=NOT_INT,/*GPIO20*/
/*GPIO_INTR_CTRL6*/.intr_ctl6.pad.mode3=NOT_INT,/*GPIO27*/.intr_ctl6.pad.mode2=NOT_INT,/*GPIO26*/.intr_ctl6.pad.mode1=NOT_INT,/*GPIO25*/.intr_ctl6.pad.mode0=NOT_INT,/*GPIO24*/
/*GPIO_INTR_CTRL7*/.intr_ctl7.pad.mode3=NOT_INT,/*GPIO31*/.intr_ctl7.pad.mode2=NOT_INT,/*GPIO30*/.intr_ctl7.pad.mode1=NOT_INT,/*GPIO29*/.intr_ctl7.pad.mode0=NOT_INT,/*GPIO28*/
};
```

在相应的 IO 口写 RIS_EDGE_INT 为上升沿中断，写 FAIL_EDGE_INT 为下降沿中断，
写 NOT_INT 为关闭中断。

三、**IO 的使用说明**

(1)使能 GPIO_PCLK 和 GPIO_CLK 时钟。
(2)配置所有 GPIO 口的复用功能。
(3)配置所有 GPIO 口中断功能。
（注：以上三步在 gpio 初始化函数里执行（extern void　　Init_gpio(void)））

四、**IO 接口函数**

在工程 gpio.c 里提供 8 个 GPIO 接口函数，以便在程序的任意地方 使用这些接口配置 GPIO。

```
extern void      gpio_mux_ctl(uint8_t num,uint8_t mux);
extern void      gpio_fun_inter(uint8_t num,uint8_t inter);
extern void      gpio_fun_sel(uint8_t num,uint8_t sel);
extern void      gpio_output_high(uint8_t num);
extern void      gpio_output_low(uint8_t num);
extern uint8_t gpio_input_val(uint8_t num);
extern void      gpio_direction_output(uint8_t num);
extern   void    gpio_direction_input(uint8_t num, uint8_t pull_up_type);
```

1,函数 extern void　　gpio_mux_ctl(uint8_t num,uint8_t mux);

函数功能：GPIO 管脚连接控制
输入参数：num-gpio 管脚号，mux-　对应的 gpio 连接控制
mux 说明如下：

```
- 32/16 PIN脚
- GPIO0 : 【mux=0 连接到gpio_d[0] 】 【mux=1 NA】                  【mux=2 pwm2】          【mux=3 NA】
- GPIO1 : 【mux=0 连接到gpio_d[1] 】 【mux=1 NA】                  【mux=2 pwm3】          【mux=3 NA】
- GPIO2 : 【mux=0 连接到gpio_d[2] 】 【mux=1 NA】                  【mux=2 clk_12M_out】   【mux=3 NA】
- GPIO3 : 【mux=0 连接到gpio_d[3] 】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
- GPIO4 : 【mux=0 连接到gpio_d[4] 】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
- GPIO5 : 【mux=0 连接到gpio_d[5] 】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
- GPIO6 : 【mux=0 连接到gpio_d[6] 】 【mux=1 NA】                  【mux=2 txen】          【mux=3 NA】
- GPIO7 : 【mux=0 连接到gpio_d[7] 】 【mux=1 NA】                  【mux=2 rxen】          【mux=3 NA】
- GPIO8 : 【mux=0 连接到gpio_d[8] 】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
- GPIO9 : 【mux=0 连接到gpio_d[9] 】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
- GPIO10: 【mux=0 连接到gpio_d[10]】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
- GPIO11: 【mux=0 连接到BOOT_CTL  】 【mux=1 连接到gpio_d[11]】    【mux=2 NA】            【mux=3 NA】
- GPIO12: 【mux=0 连接到SWI的 SWCK】 【mux=1 连接到gpio_d[12]】    【mux=2 NA】            【mux=3 pwm4】
- GPIO13: 【mux=0 连接到SWI的 SWD 】 【mux=1 连接到gpio_d[13]】    【mux=2 NA】            【mux=3 pwm5】
- GPIO14: 【mux=0 连接到gpio_d[14]】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
- GPIO18: 【mux=0 连接到gpio_d[18]】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
- GPIO19: 【mux=0 连接到gpio_d[19]】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
- GPIO20: 【mux=0 连接到gpio_d[20]】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
- GPIO21: 【mux=0 连接到gpio_d[21]】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
- GPIO22: 【mux=0 连接到gpio_d[22]】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
- GPIO23: 【mux=0 连接到gpio_d[23]】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
- GPIO24: 【mux=0 连接到gpio_d[24]】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
- GPIO25: 【mux=0 连接到gpio_d[25]】 【mux=1 NA】                  【mux=2 NA】            【mux=3 NA】
```

2,函数 extern void gpio_fun_inter(uint8_t num,uint8_t inter);

函数功能：GPIO 中断控制

输入参数：num-gpio 管脚号,inter-对应的 gpio 中断模式

inter 说明如下:

```
-【inter=0  GPIO关闭中断:NOT_INT       】
-【inter=5  GPIO上升沿中断:RIS_EDGE_INT 】
-【inter=7  GPIO下降沿中断:FAIL_EDGE_INT】
```

3,函数 extern void gpio_fun_sel(uint8_t num,uint8_t sel);

函数功能：GPIO 复用控制

输入参数：num-gpio 管脚号,sel-对应的 gpio 复用功能设置

sel 说明如下:

```
-【sel=0   普通GPIO功能:GPIO_Dx                 】
-【sel=1   串口0发送:UART0_TX                   】
-【sel=2   串口0接收::UART0_RX                  】
-【sel=3   串口0流控:UART0_CTS                  】
-【sel=4   串口0流控:UART0_RTS                  】
-【sel=5   I2C时钟:I2C_SCL                      】
-【sel=6   I2C数据:I2C_SDA                      】
-【sel=7   串口1接收:UART1_RX                   】
-【sel=8   串口1发送:UART1_TX                   】
-【sel=9   SIM_IO                               】
-【sel=10  SIM_RST                              】
-【sel=11  SIM_CLK_OUT                          】
-【sel=12  PWM0输出:PWM0                        】
-【sel=13  PWM1输出:PWM1                        】
-【sel=14  SPI1时钟:SSI1_CLK                    】
-【sel=15  SPI1片选:SSI1_SSN                    】
-【sel=16  SPI1接收:SSI1_RX                     】
-【sel=17  SPI1发送:SSI1_TX                     】
-【sel=18  PWM0互补输出:PWM0_INV(32/16PIN新增)】
-【sel=19  PWM1互补输出:PWM1_INV(32/16PIN新增)】
```

4,函数 gpio_direction_input(uint8_t num, uint8_t pull_up_type);

函数功能：GPIO 输入控制

输入参数：num-gpio 管脚号,pull_up_type-对应的 gpio 输入上下拉设置

pull_up_type 说明如下:

```
-【pull_up_type=0  GPIO上拉:GPIO_Mode_Input_Up   】
-【pull_up_type=1  GPIO下拉:GPIO_Mode_Input_Down 】
-【pull_up_type=2  GPIO浮空:GPIO_Mode_Input_Float】
```

6,函数 gpio_input_val(uint8_t num);

函数功能：检测 GPIO 输入高低电平

输入参数：num-gpio 管脚号

返 回 值: 检测到输入为高电平返回 1、检测输入为低电平返回 0

6,函数 gpio_direction_output(uint8_t num);

函数功能：GPIO 设置为输出

输入参数：num-gpio 管脚号

7,函数 gpio_output_low(uint8_t num);

函数功能：GPIO 输出低电平

输入参数：num-gpio 管脚号

8,函数 gpio_output_high(uint8_t num);

函数功能：GPIO 输出高电平

输入参数：num-gpio 管脚号

四，GPADC使用注意事项

adc一共10个通道，其中0~7通道为外部输入接口，8通道用来测试内部AVDD*2/3电压，9通　道用来测试内部HVIN*0.4（用于5V或者锂电池供电场景）。16个周期出一个数，adc时钟最高支持8MHz，默认配置成1MHz，一般建议配置成1/2/4 MHz。

adc满量程电压范围可以配置成0~2.47V或者0~AVDD。默认配置成0~2.47V（电源电压3.3V）。常温下理论精度1.3%，实测精度<2%。

GADC_VREF_SEL（0x24<1>）默认是0，0~2.47V，改成1，0~AVDD。

Adc满量程电压和电源电压有关。在电源电压3.3V下为2.47V。电源电压每降低0.1V，Adc满量程电压降低0.0064V。(但是在3.6V下例外，为2.5749V)。