ESP32−C3 教程

尚硅谷

一　概述

ESP32-C3 SoC 芯片支持以下功能：

- 2.4 GHz Wi-Fi
- 低功耗蓝牙
- 高性能 32 位 RISC-V 单核处理器
- 多种外设
- 内置安全硬件

ESP32-C3 采用 40 nm 工艺制成，具有最佳的功耗性能、射频性能、稳定性、通用性和可靠性，适用于各种应用场景和不同功耗需求。

此芯片由乐鑫公司开发。

二　安装开发工具 ESP−IDF

ESP-IDF 需要安装一些必备工具，才能围绕 ESP32-C3 构建固件，包括 Python、Git、交叉编译器、CMake 和 Ninja 编译工具等。

在本入门指南中，我们通过 命令行 进行有关操作。

> ⚠ Warning
>
> 限定条件：
>
> - 请注意 ESP-IDF 和 ESP-IDF 工具的安装路径不能超过 90 个字符，安装路径过长可能会导致构建失败。
> - Python 或 ESP-IDF 的安装路径中一定不能包含空格或括号。
> - 除非操作系统配置为支持 Unicode UTF-8，否则 Python 或 ESP-IDF 的安装路径中也不能包括特殊字符（非 ASCII 码字符）
>
> 系统管理员可以通过如下方式将操作系统配置为支持 Unicode UTF-8：控制面板-更改日期、时间或数字格式-管理选项卡-更改系统地域-勾选选项 “Beta：使用 Unicode UTF-8 支持全球语言”-点击确定-重启电脑。

2.1　离线安装 ESP−IDF

点击链接下载离线安装包。

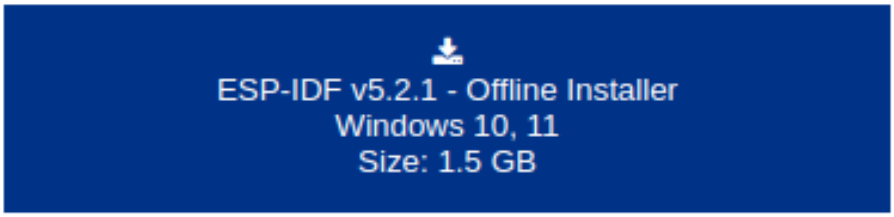

图 1　离线安装包示意图

2.2　安装内容

安装程序会安装以下组件：

- 内置的 Python
- 交叉编译器
- OpenOCD
- CMake 和 Ninja 编译工具
- ESP-IDF

安装程序允许将程序下载到现有的 ESP-IDF 目录。推荐将 ESP-IDF 下载到 `%userprofile%\Desktop\esp-idf` 目录下，其中 `%userprofile%` 代表家目录。

2.3　启动 ESP–IDF 环境

安装结束时，如果勾选了 `Run ESP-IDF PowerShell Environment` 或 `Run ESP-IDF Command Prompt (cmd.exe)`，安装程序会在选定的提示符窗口启动 `ESP-IDF`。

`Run ESP-IDF PowerShell Environment`：

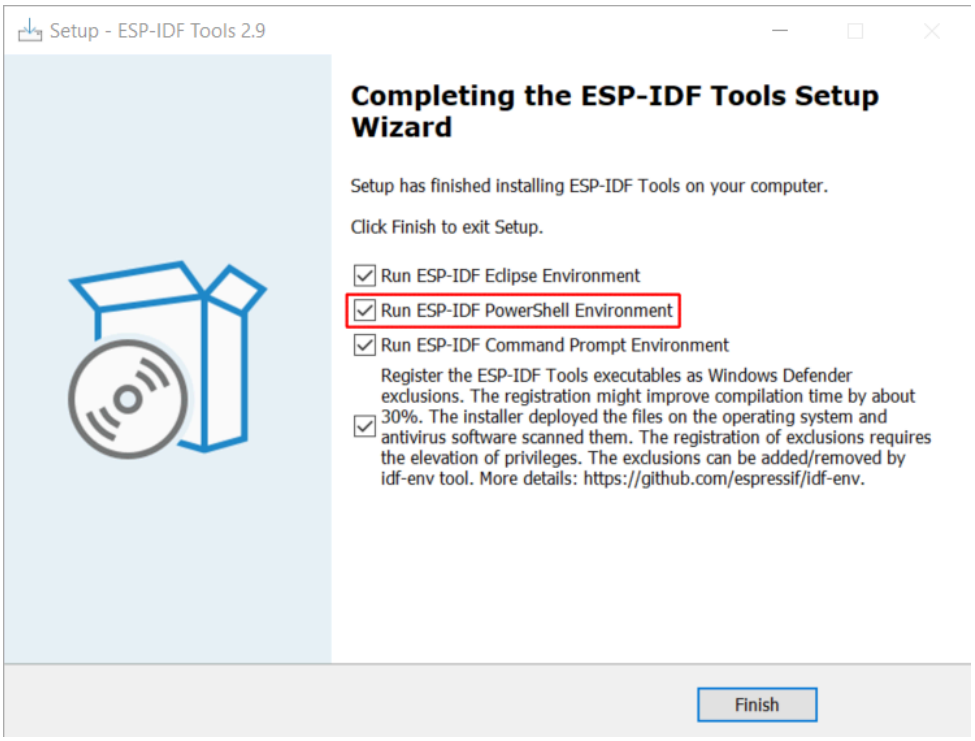

图 2　PowerShell

三　开始创建工程

现在，可以准备开发 ESP32 应用程序了。可以从 `ESP-IDF` 中 `examples` 目录下的 `get-started/hello_world` 工程开始。

> ⚠ Warning
>
> ESP-IDF 编译系统不支持 ESP-IDF 路径或其工程路径中带有空格。

将 `get-started/hello_world` 工程复制至本地的 `~/esp` 目录下：

复制工程

```
cd %userprofile%\esp
xcopy /e /i %IDF_PATH%\examples\get-started\hello_world hello_world
```

> **Info**
>
> ESP-IDF 的 examples 目录下有一系列示例工程，可以按照上述方法复制并运行其中的任何示例，也可以直接编译示例，无需进行复制。

3.1 连接设备

现在，请将 ESP32 开发板连接到 PC，并查看开发板使用的串口。

在 Windows 操作系统中，串口名称通常以 COM 开头。

3.2 配置工程

请进入 `hello_world` 目录，设置 ESP32-C3 为目标芯片，然后运行工程配置工具 `menuconfig` 。

配置代码

```
cd %userprofile%\esp\hello_world
idf.py set-target esp32c3
idf.py menuconfig
```

打开一个新工程后，应首先使用 `idf.py set-target esp32c3` 设置“目标”芯片。注意，此操作将清除并初始化项目之前的编译和配置（如有）。也可以直接将“目标”配置为环境变量（此时可跳过该步骤）。

正确操作上述步骤后，系统将显示以下菜单：

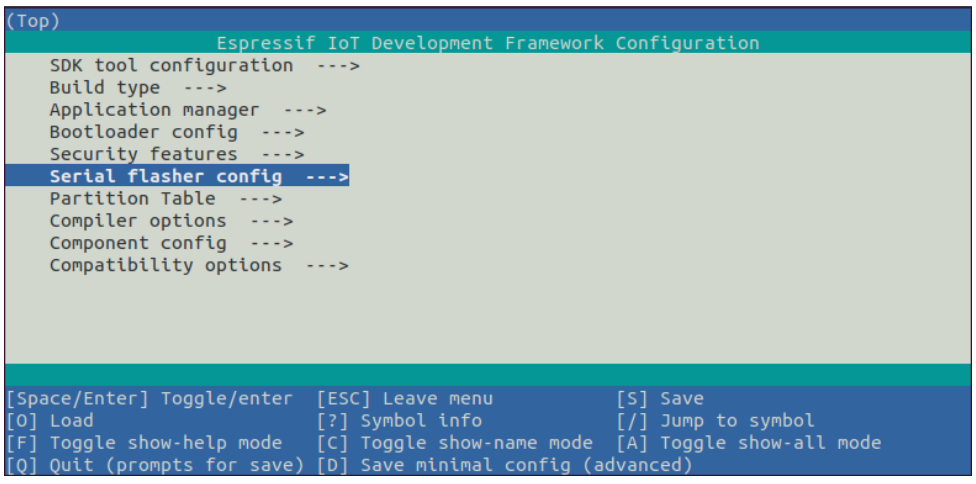

图 3　配置界面示意图

可以通过此菜单设置项目的具体变量，包括 Wi-Fi 网络名称、密码和处理器速度等。`hello_world` 示例项目会以默认配置运行，因此在这一项目中，可以跳过使用 `menuconfig` 进行项目配置这一步骤。

3.3 编译工程

请使用以下命令，编译烧录工程：

编译构建项目命令

```
idf.py build
```

运行以上命令可以编译应用程序和所有 ESP-IDF 组件，接着生成引导加载程序、分区表和应用程序二进制文件。

编译构建项目命令运行结果

```
$ idf.py build
Running cmake in directory /path/to/hello_world/build
Executing "cmake -G Ninja --warn-uninitialized /path/to/hello_world"...
Warn about uninitialized values.
-- Found Git: /usr/bin/git (found version "2.17.0")
-- Building empty aws_iot component due to configuration
-- Component names: ...
-- Component paths: ...

... (more lines of build system output)

[527/527] Generating hello_world.bin
esptool.py v2.3.1

Project build complete. To flash, run this command:
../../../components/esptool_py/esptool/esptool.py -p (PORT) -b 921600 write_flash --flash_mode dio
--flash_size detect --flash_freq 40m 0x10000 build/hello_world.bin  build 0x1000 build/bootloader/
bootloader.bin 0x8000 build/partition_table/partition-table.bin
or run 'idf.py -p PORT flash'
```

如果一切正常，编译完成后将生成 `.bin` 文件。

3.4 烧录到设备

请运行以下命令，将刚刚生成的二进制文件烧录至 ESP32 开发板：

编译+烧录

```
idf.py flash
```

> **Info**
>
> 勾选 `flash` 选项将自动编译并烧录工程，因此无需再运行 `idf.py build`。

3.5 常规操作

在烧录过程中，会看到类似如下的输出日志：

输出日志

```
...
esptool.py --chip esp32 -p /dev/ttyUSB0 -b 460800 --before=default_reset --after=hard_reset write_flash --flash_mode dio --flash_freq 40m --flash_size 2MB 0x8000 partition_table/partition-table.bin 0x1000 bootloader/bootloader.bin 0x10000 hello_world.bin
esptool.py v3.0-dev
Serial port /dev/ttyUSB0
Connecting.....
Chip is ESP32D0WDQ6 (revision 0)
Features: WiFi, BT, Dual Core, Coding Scheme None
Crystal is 40MHz
MAC: 24:0a:c4:05:b9:14
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 460800
Changed.
Configuring flash size...
Compressed 3072 bytes to 103...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (103 compressed) at 0x00008000 in 0.0 seconds (effective 5962.8 kbit/s)...
Hash of data verified.
Compressed 26096 bytes to 15408...
Writing at 0x00001000... (100 %)
Wrote 26096 bytes (15408 compressed) at 0x00001000 in 0.4 seconds (effective 546.7 kbit/s)...
Hash of data verified.
Compressed 147104 bytes to 77364...
Writing at 0x00010000... (20 %)
Writing at 0x00014000... (40 %)
Writing at 0x00018000... (60 %)
Writing at 0x0001c000... (80 %)
Writing at 0x00020000... (100 %)
Wrote 147104 bytes (77364 compressed) at 0x00010000 in 1.9 seconds (effective 615.5 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
Done
```

如果一切顺利，烧录完成后，开发板将会复位，应用程序 `hello_world` 开始运行。

3.6 监视输出

使用 串口助手 监视输出和调试。

> ⚠ Warning
>
> 当要进行烧写时，请关闭串口助手！

四　基本 GPIO 操作

4.1　GPIO 配置

普通配置

一般 GPIO 配置

```
gpio_config_t io_conf;
// 禁用中断
io_conf.intr_type = GPIO_INTR_DISABLE;
// 设置 GPIO 为输出模式
io_conf.mode = GPIO_MODE_OUTPUT;
// 设置 GPIO PIN 引脚为 GPIO1 和 GPIO2
io_conf.pin_bit_mask = ((1ULL << GPIO_NUM_1) | (1ULL << GPIO_NUM_2));
// 禁用下拉模式
io_conf.pull_down_en = 0;
// 开启上拉模式
io_conf.pull_up_en = 1;
// 使用以上配置来配置 GPIO
gpio_config(&io_conf);
```

有关中断的配置方法

GPIO 配置中断

```
// 上升沿触发中断
io_conf.intr_type = GPIO_INTR_POSEDGE;
// 设置为输入模式
io_conf.mode = GPIO_MODE_INPUT;
// 配置引脚
io_conf.pin_bit_mask = (1ULL << GPIO_NUM_0);
gpio_config(&io_conf);
```

操作 GPIO 的 API

操作 GPIO 引脚

```
// 将 GPIO 口设置为输入模式
gpio_set_direction(GPIO_NUM_2, GPIO_MODE_INPUT);
```

```
// 设置输出模式
gpio_set_direction(GPIO_NUM_2, GPIO_MODE_OUTPUT);
// 输出高低电平
gpio_set_level(GPIO_NUM_1, 1);
gpio_set_level(GPIO_NUM_1, 0);
// 获取 GPIO 的电平
gpio_get_level(GPIO_NUM_2);
```

有了这些 API，我们可以实现 I^2C 协议了。然后就可以实现按键功能了。键盘电路图如下：

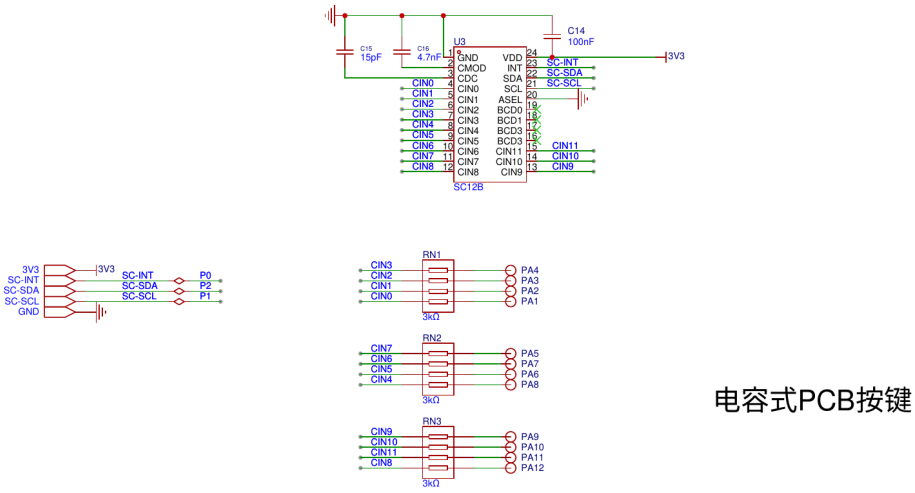

图 4　键盘模块电路图

为了方便操作，我们先来定义一组宏定义以及声明头文件。

先在 `main` 文件夹中创建 `drivers` 文件夹，然后创建文件 `keyboard_driver.h` 。文件内容如下：

操作 GPIO 引脚

```
#ifndef __KEYBOARD_DRIVER_H_
#define __KEYBOARD_DRIVER_H_

#include <inttypes.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"

#define SC12B_SCL GPIO_NUM_1
#define SC12B_SDA GPIO_NUM_2
#define SC12B_INT GPIO_NUM_0

#define I2C_SDA_IN gpio_set_direction(SC12B_SDA, GPIO_MODE_INPUT)
```

```
#define I2C_SDA_OUT gpio_set_direction(SC12B_SDA, GPIO_MODE_OUTPUT)

#define I2C_SCL_H gpio_set_level(SC12B_SCL, 1)
#define I2C_SCL_L gpio_set_level(SC12B_SCL, 0)

#define I2C_SDA_H gpio_set_level(SC12B_SDA, 1)
#define I2C_SDA_L gpio_set_level(SC12B_SDA, 0)

#define I2C_READ_SDA gpio_get_level(SC12B_SDA)

void Delay_ms(uint8_t time);
void I2C_Start(void);
void I2C_Stop(void);
void I2C_Ack(uint8_t x);
uint8_t I2C_Wait_Ack(void);
void I2C_Send_Byte(uint8_t d);
uint8_t I2C_Read_Byte(uint8_t ack);
uint8_t SendByteAndGetNACK(uint8_t data);
uint8_t I2C_Read_Key(void);
uint8_t KEYBOARD_read_key(void);
void KEYBORAD_init(void);

#endif
```

在 `drivers` 文件夹中创建 `keyboard_driver.c` 文件。内容如下：

操作 GPIO 引脚

```
#include "keyboard_driver.h"

/// 延时函数，使用 FreeRTOS 的 API 进行包装
void Delay_ms(uint8_t time)
{
    vTaskDelay(time / portTICK_PERIOD_MS);
}

/// 产生起始信号
void I2C_Start(void)
{
    I2C_SDA_OUT; // sda 线输出
    I2C_SDA_H;
    I2C_SCL_H;
    Delay_ms(1);
    I2C_SDA_L; // START:when CLK is high,DATA change form high to low
    Delay_ms(1);
    I2C_SCL_L; // 钳住 I2C 总线，准备发送或接收数据
    Delay_ms(1);
}

/// 产生停止信号
void I2C_Stop(void)
{
```

```
    I2C_SCL_L;
    I2C_SDA_OUT; // sda 线输出
    I2C_SDA_L;   // STOP:when CLK is high DATA change form low to high
    Delay_ms(1);
    I2C_SCL_H;
    Delay_ms(1);
    I2C_SDA_H; // 发送 I2C 总线结束信号
}

/// 下发应答
void I2C_Ack(uint8_t x)
{
    I2C_SCL_L;
    I2C_SDA_OUT;
    if (x)
    {
        I2C_SDA_H;
    }
    else
    {
        I2C_SDA_L;
    }
    Delay_ms(1);
    I2C_SCL_H;
    Delay_ms(1);
    I2C_SCL_L;
}

/// 等待应答信号到来，成功返回 0 。
uint8_t I2C_Wait_Ack(void)
{
    uint8_t ucErrTime = 0;
    I2C_SCL_L;
    I2C_SDA_IN; // SDA 设置为输入
    Delay_ms(1);
    I2C_SCL_H;
    Delay_ms(1);
    while (I2C_READ_SDA)
    {
        if (ucErrTime++ > 250)
        {
            // I2C_Stop();
            // printf("接受应答失败\n");
            return 1;
        }
    }
    I2C_SCL_L;
    // printf("接受应答成功\n");
    return 0;
}

/// 发送一个字节
void I2C_Send_Byte(uint8_t d)
```

```
{
    uint8_t t = 0;
    I2C_SDA_OUT;
    while (8 > t++)
    {
        I2C_SCL_L;
        Delay_ms(1);
        if (d & 0x80)
        {
            I2C_SDA_H;
        }
        else
        {
            I2C_SDA_L;
        }
        Delay_ms(1); // 对 TEA5767 这三个延时都是必须的
        I2C_SCL_H;
        Delay_ms(1);
        d <<= 1;
    }
}

/// 读 1 个字节
uint8_t I2C_Read_Byte(uint8_t ack)
{
    uint8_t i = 0;
    uint8_t receive = 0;
    I2C_SDA_IN; // SDA 设置为输入
    for (i = 0; i < 8; i++)
    {
        I2C_SCL_L;
        Delay_ms(1);
        I2C_SCL_H;
        receive <<= 1;
        if (I2C_READ_SDA)
        {
            receive++;
        }
        Delay_ms(1);
    }
    I2C_Ack(ack); // 发送 ACK
    return receive;
}

/// 发送数据并返回应答
uint8_t SendByteAndGetNACK(uint8_t data)
{
    I2C_Send_Byte(data);
    return I2C_Wait_Ack();
}

/// SC12B 简易读取按键值函数（默认直接读取）
```

```
130 /// 此函数只有初始化配置默认的情况下，直接调用，如果在操作前有写入或者其他读取不能调用默认
131 uint8_t I2C_Read_Key(void)
132 {
133     I2C_Start();
134     if (SendByteAndGetNACK((0x40 << 1) | 0x01))
135     {
136         I2C_Stop();
137         return 0;
138     }
139     uint8_t i = 0;
140     uint8_t k = 0;
141     I2C_SDA_IN; // SDA 设置为输入
142     while (8 > i)
143     {
144         i++;
145         I2C_SCL_L;
146         Delay_ms(1);
147         I2C_SCL_H;
148         if (!k && I2C_READ_SDA)
149         {
150             k = i;
151         }
152         Delay_ms(1);
153     }
154     if (k)
155     {
156         I2C_Ack(1);
157         I2C_Stop();
158         return k;
159     }
160     I2C_Ack(0);
161     I2C_SDA_IN; // SDA 设置为输入
162     while (16 > i)
163     {
164         i++;
165         I2C_SCL_L;
166         Delay_ms(1);
167         I2C_SCL_H;
168         if (!k && I2C_READ_SDA)
169         {
170             k = i;
171         }
172         Delay_ms(1);
173     }
174     I2C_Ack(1);
175     I2C_Stop();
176     return k;
177 }
178
179 uint8_t KEYBOARD_read_key(void)
180 {
181     uint16_t key = I2C_Read_Key();
```

```
182     if (key == 4)
183     {
184         return 1;
185     }
186     else if (key == 3)
187     {
188         return 2;
189     }
190     else if (key == 2)
191     {
192         return 3;
193     }
194     else if (key == 7)
195     {
196         return 4;
197     }
198     else if (key == 6)
199     {
200         return 5;
201     }
202     else if (key == 5)
203     {
204         return 6;
205     }
206     else if (key == 10)
207     {
208         return 7;
209     }
210     else if (key == 9)
211     {
212         return 8;
213     }
214     else if (key == 8)
215     {
216         return 9;
217     }
218     else if (key == 1)
219     {
220         return 0;
221     }
222     else if (key == 12)
223     {
224         return '#';
225     }
226     else if (key == 11)
227     {
228         return 'M';
229     }
230     return 255;
231 }
232 
233 /// GPIO 初始化
234 void KEYBORAD_init(void)
235 {
```

```
    gpio_config_t io_conf;
    // disable interrupt
    io_conf.intr_type = GPIO_INTR_DISABLE;
    // set as output mode
    io_conf.mode = GPIO_MODE_OUTPUT;
    // bit mask of the pins that you want to set,e.g.SDA
    io_conf.pin_bit_mask = ((1ULL << SC12B_SCL) | (1ULL << SC12B_SDA));
    // disable pull-down mode
    io_conf.pull_down_en = 0;
    // disable pull-up mode
    io_conf.pull_up_en = 1;
    // configure GPIO with the given settings
    gpio_config(&io_conf);

    // 中断
    io_conf.intr_type = GPIO_INTR_POSEDGE;
    io_conf.mode = GPIO_MODE_INPUT;
    io_conf.pin_bit_mask = (1ULL << SC12B_INT);
    gpio_config(&io_conf);
}
```

驱动编写好之后，我们可以在主函数中和电容键盘进行通信了。当按下按键，会产生中断，通过处理中断来识别我们的按键。

在 `smart-lock.c` 文件中，主函数是 line 44：`app_main`，`ESP-IDF` 在编译整个项目的时候，会将 `app_main` 注册为一个 RTOS 任务。无需我们自己编写 `main` 函数。

`smart-lock.c` 文件内容如下。

smart-lock.c

```
// 全局变量，用来存储来自 GPIO 的中断事件
static QueueHandle_t gpio_evt_queue = NULL;

static void IRAM_ATTR gpio_isr_handler(void *arg)
{
  uint32_t gpio_num = (uint32_t)arg;
  // 将产生中断的 GPIO 引脚号入队列。
  xQueueSendFromISR(gpio_evt_queue, &gpio_num, NULL);
}

// 轮训中断事件队列，然后挨个处理
static void process_isr(void *arg)
{
  uint32_t io_num;
  for (;;)
  {
    if (xQueueReceive(gpio_evt_queue, &io_num, portMAX_DELAY))
    {
      if (io_num == 0)
      {
        uint8_t key = KEYBOARD_read_key();
```

```
22          printf("按下的键：%d\r\n", key);
23        }
24      }
25    }
26  }
27
28  static void ISR_QUEUE_Init(void)
29  {
30    // 创建一个队列来处理来自 GPIO 的中断事件
31    gpio_evt_queue = xQueueCreate(10, sizeof(uint32_t));
32    // 开启 process_isr 任务。
33    // 这个任务的作用是轮训存储中断事件的队列，将队列中的事件
34    // 挨个出队列并进行处理。
35    xTaskCreate(process_isr, "process_isr", 2048, NULL, 10, NULL);
36
37    gpio_install_isr_service(0);
38    // 将 SC12B_INT 引脚产生的中断交由 gpio_isr_handler 处理。
39    // 也就是说一旦 SC12B_INT 产生中断，则调用 gpio_isr_handler 函数。
40    gio_isr_handler_add(SC12B_INT, gpio_isr_handler, (void *)SC12B_INT);
41  }
42
43  // 主程序
44  void app_main(void)
45  {
46    ISR_QUEUE_Init();
47  }
```

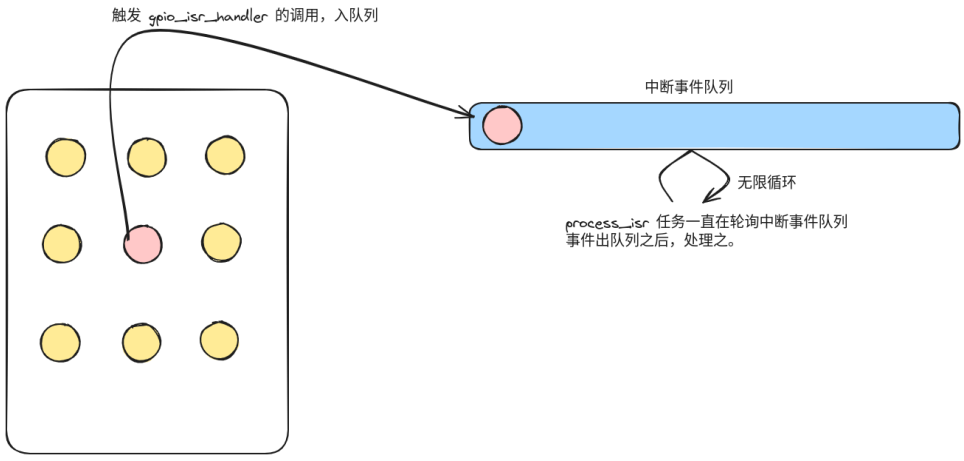

图 5　处理中断示意图

最后我们将编写好的代码添加到 `main` 文件夹下的 `CMakeLists.txt` 文件中。最终我们的项目的文件如下，`SRCS` 包含我们编写的 c 文件。`INCLUDE_DIRS` 包含我们编写的驱动的文件夹。

CMakeLists.txt

```
idf_component_register(
    SRCS
        "smart-lock.c"
        "drivers/keyboard_driver.c"
        "drivers/led_driver.c"
        "drivers/bluetooth_driver.c"
        "drivers/wifi_driver.c"
        "drivers/fingerprint_driver.c"
        "drivers/tcp_driver.c"
        "drivers/motor_driver.c"
        "drivers/audio_driver.c"
    INCLUDE_DIRS
        "."
        "./drivers"
)
```

五　红外遥控(RMT)

5.1　简介

红外遥控 (RMT) 外设是一个红外发射和接收控制器。其数据格式灵活，可进一步扩展为多功能的通用收发器，发送或接收多种类型的信号。就网络分层而言，RMT 硬件包含物理层和数据链路层。物理层定义通信介质和比特信号的表示方式，数据链路层定义 RMT 帧的格式。RMT 帧的最小数据单元称为 RMT 符号，在驱动程序中以 `rmt_symbol_word_t` 表示。

`ESP32-C3` 的 RMT 外设存在多个通道，每个通道都可以独立配置为发射器或接收器。

RMT 外设通常支持以下场景：

- 发送或接收红外信号，支持所有红外线协议，如 NEC 协议
- 生成通用序列
- 有限或无限次地在硬件控制的循环中发送信号
- 多通道同时发送
- 将载波调制到输出信号或从输入信号解调载波

5.2　RMT 符号的内存布局

RMT 硬件定义了自己的数据模式，称为 `RMT 符号` 。下图展示了一个 RMT 符号的位字段：每个符号由两对两个值组成，每对中的第一个值是一个 15 位的值，表示信号持续时间，以 RMT 滴答计。每对中的第二个值是一个 1 位的值，表示信号的逻辑电平，即高电平或低电平。

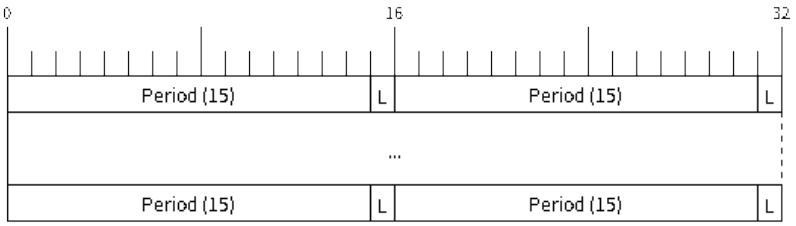

图 6　RMT 符号结构(L−信号电平)

5.3　RMT 发射器概述

RMT 发送通道 (TX Channel) 的数据路径和控制路径如下图所示：

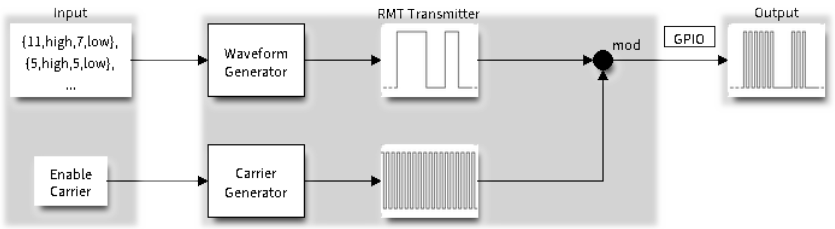

图 7　RMT 发射器概述

驱动程序将用户数据编码为 RMT 数据格式，随后由 RMT 发射器根据编码生成波形。在将波形发送到 GPIO 管脚前，还可以调制高频载波信号。

5.4　RMT 接收器概述

RMT 接收通道 (RX Channel) 的数据路径和控制路径如下图所示：

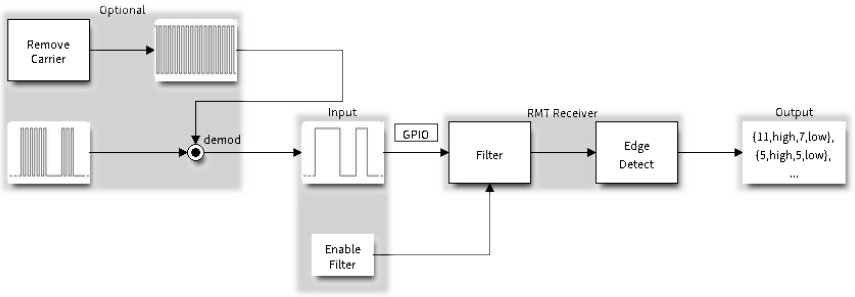

图 8　RMT 接收器概述

RMT 接收器可以对输入信号采样，将其转换为 RMT 数据格式，并将数据存储在内存中。还可以向接收器提供输入信号的基本特征，使其识别信号停止条件，并过滤掉信号干扰和噪声。RMT 外设还支持从基准信号中解调出高频载波信号。

5.5　补充知识：数字调制

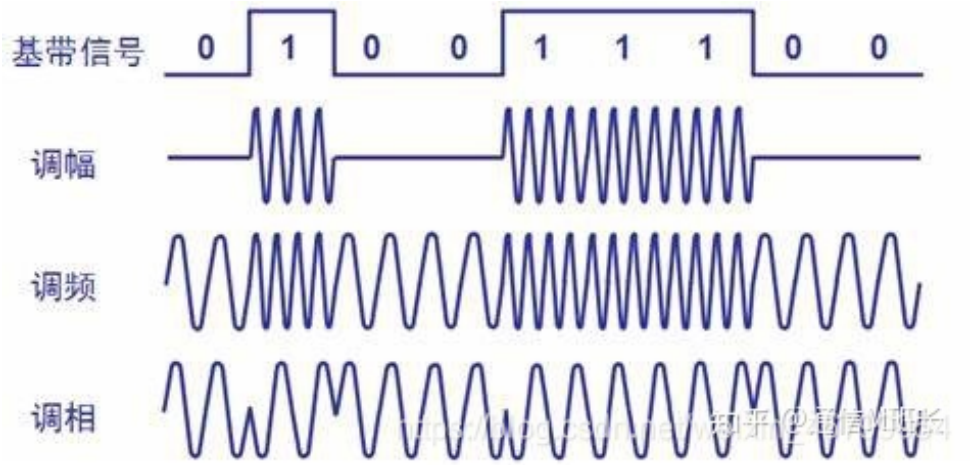

图 9　数字信号调制示意图

5.6　ws2812

文件夹 `esp-idf/examples/peripherals/rmt/led_strip` 是示例代码。修改 RMT 的 GPIO 引脚就可以直接部署运行。

我们的开发板的原理是 `esp32c3` 芯片使用 RMT 模块的功能通过 GPIO 引脚发送波形。而波形是经过编码的 RGB 值。

原理图如下：

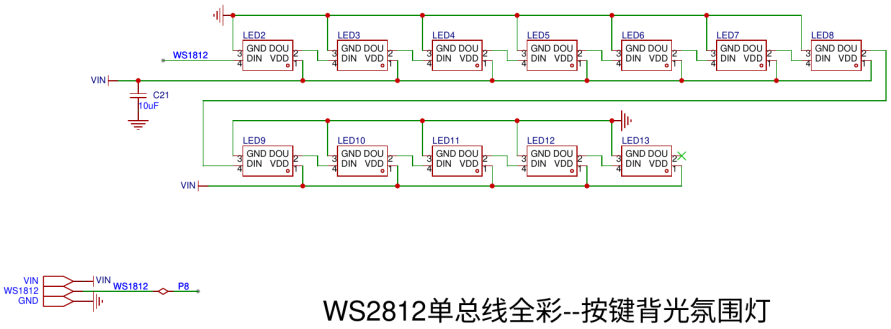

图 10　LED 灯原理图

驱动大部分外设来说，几乎是通过 GPIO 的高低电平来处理，而 ws2812 正是需要这样的电平；RMT（远程控制）模块驱动程序可用于发送和接收红外遥控信号。由于 RMT 灵活性，驱动程序还可用于生成或接收许多其他类型的信号。由一系列脉冲组成的信号由 RMT 的发射器根据值列表生成。这些值定义脉冲持续时间和二进制级别。发射器还可以提供载波并用提供的脉冲对其进行调制；总的来说它就是一个中间件，就是通过 RMT 模块可以生成解码成包含脉冲持续时间和二进制电平的值的高低电平，从而实现发送和接收我们想要的信号。

关于这个灯珠的资料网上多的是，我总的概述：

1. 每颗灯珠内置一个驱动芯片，我们只需要和这个驱动芯片通讯就可以达成调光的目的。所以，我们不需要用 PWM 调节。
2. 它的管脚引出有 4 个，2 个是供电用的。还有 2 个是通讯的，DIN 是输入，DOUT 是输出。以及其是 5V 电压供电。
3. 根据不同的厂商生产不同，驱动的方式有所不一样！下面发送数据顺序是：`GREEN -- BLUE -- RED` 。

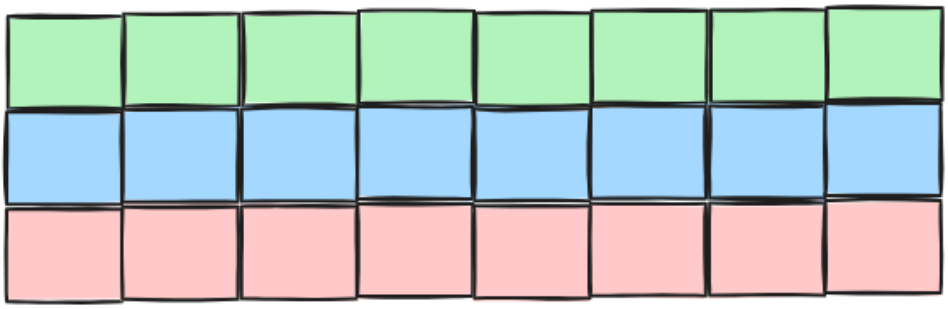

图 11　发送颜色的顺序

5.7 代码

由于大部分代码都是示例代码。这里只给出新添加的部分，也就是点亮某一个灯的代码。

点灯程序

```
// `led_num` 参数是要点亮的灯的索引。`LED_NUMBERS == 12`，因为我们有 12 个灯。
void light_led(uint8_t led_num)
{
    for (int i = 0; i < 3; i++)
    {
        // 构建 RGB 像素点
        hue = led_num * 360 / LED_NUMBERS;
        // 编码 RGB 值
        led_strip_hsv2rgb(hue, 30, 30, &red, &green, &blue);
        // 发送顺序 GREEN --> BLUE --> RED
        led_strip_pixels[led_num * 3 + 0] = green;
        led_strip_pixels[led_num * 3 + 1] = blue;
        led_strip_pixels[led_num * 3 + 2] = red;
    }

    // 将 RGB 值通过通道发送至 LED 灯。点亮灯。
    ESP_ERROR_CHECK(rmt_transmit(
        led_chan,
        led_encoder,
        led_strip_pixels,
        sizeof(led_strip_pixels),
        &tx_config));
    ESP_ERROR_CHECK(rmt_tx_wait_all_done(led_chan, portMAX_DELAY));

    // 延时 100 毫秒
    vTaskDelay(100 / portTICK_PERIOD_MS);

    // 清空像素矩阵
    memset(led_strip_pixels, 0, sizeof(led_strip_pixels));

    // 再次发送，将灯灭掉。
    ESP_ERROR_CHECK(rmt_transmit(
        led_chan,
        led_encoder,
        led_strip_pixels,
        sizeof(led_strip_pixels),
        &tx_config));
    ESP_ERROR_CHECK(rmt_tx_wait_all_done(led_chan, portMAX_DELAY));
}
```

尝试编写代码调用点灯方法，将灯点亮。

六　语音模块

我们使用 WTN6170 作为语音模块外设。可以使用一根 GPIO 线来控制 WTN6170 。

交互语音播放电路

图 12　语音模块电路图

我们来编写初始化 GPIO 引脚的代码。

`AUDIO_BUSY_PIN` 和 `AUDIO_SDA_PIN` 可查询电路图来进行配置。

语音模块 GPIO 引脚配置

```
{
    gpio_config_t io_conf = {};
    // 禁用中断
    io_conf.intr_type = GPIO_INTR_DISABLE;
    // 设置为输出模式
    io_conf.mode = GPIO_MODE_OUTPUT;
    // 引脚是数据线
    io_conf.pin_bit_mask = (1ULL << AUDIO_SDA_PIN);
    gpio_config(&io_conf);

    // 禁用中断
    io_conf.intr_type = GPIO_INTR_DISABLE;
    // 设置为输入模式
    io_conf.mode = GPIO_MODE_INPUT;
    // 引脚是忙线
    io_conf.pin_bit_mask = (1ULL << AUDIO_BUSY_PIN);
    gpio_config(&io_conf);
}
```

给语音模块发送数据并播报的代码，通过发送不同的 u8 数据，使语音模块播放不同的声音。具体参见语音模块文档。

播报语音代码

```c
void Line_1A_WT588F(uint8_t DDATA)
{
    uint8_t S_DATA, j;
    uint8_t B_DATA;
    S_DATA = DDATA;
    AUDIO_SDA_L;
    DELAY_MS(10); // 这里的延时比较重要
    B_DATA = S_DATA & 0X01;
    for (j = 0; j < 8; j++)
    {
        if (B_DATA == 1)
        {
            AUDIO_SDA_H;
            DELAY_US(600); // 延时 600us
            AUDIO_SDA_L;
            DELAY_US(200); // 延时 200us
        }
        else
        {
            AUDIO_SDA_H;
            DELAY_US(200); // 延时 200us
            AUDIO_SDA_L;
            DELAY_US(600); // 延时 600us
        }
        S_DATA = S_DATA >> 1;
        B_DATA = S_DATA & 0X01;
    }
    AUDIO_SDA_H;
    DELAY_MS(2);
}
```

七　电机驱动

电机用来开关锁。也就是通过驱动电机进行正转反转来开关锁。

当然我们还是通过 GPIO 的拉高拉低来驱动电机。比较简单。

电路图如下：

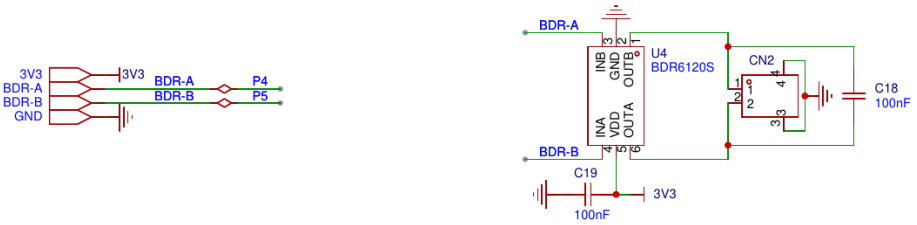

图 13　电机模块电路图

初始化 GPIO 引脚代码

电机 GPIO 引脚初始化

```
void MOTOR_Init(void)
{
    gpio_config_t io_conf;
    // 禁用中断
    io_conf.intr_type = GPIO_INTR_DISABLE;
    // 设置为输出模式
    io_conf.mode = GPIO_MODE_OUTPUT;
    // 设置要用的两个引脚
    io_conf.pin_bit_mask = ((1ULL << MOTOR_DRIVER_NUM_0) | (1ULL << MOTOR_DRIVER_NUM_1));
    gpio_config(&io_conf);

    // 最开始都输出低电平，这样就不转
    gpio_set_level(MOTOR_DRIVER_NUM_0, 0);
    gpio_set_level(MOTOR_DRIVER_NUM_1, 0);
}
```

开锁代码

控制电机转动来开关锁

```
void MOTOR_Open_lock(void)
{
    // 正转 1 秒
    gpio_set_level(MOTOR_DRIVER_NUM_0, 0);
    gpio_set_level(MOTOR_DRIVER_NUM_1, 1);
    vTaskDelay(1000 / portTICK_PERIOD_MS);

    // 停止 1 秒
    gpio_set_level(MOTOR_DRIVER_NUM_0, 0);
    gpio_set_level(MOTOR_DRIVER_NUM_1, 0);
    vTaskDelay(1000 / portTICK_PERIOD_MS);

    // 反转 1 秒
    gpio_set_level(MOTOR_DRIVER_NUM_0, 1);
    gpio_set_level(MOTOR_DRIVER_NUM_1, 0);
    vTaskDelay(1000 / portTICK_PERIOD_MS);

    // 停止转动并播报语音
    gpio_set_level(MOTOR_DRIVER_NUM_0, 0);
    gpio_set_level(MOTOR_DRIVER_NUM_1, 0);
    Line_1A_WT588F(25);
}
```

八　指纹模块

ESP32 使用串口和指纹模块进行通信。电路图如下：

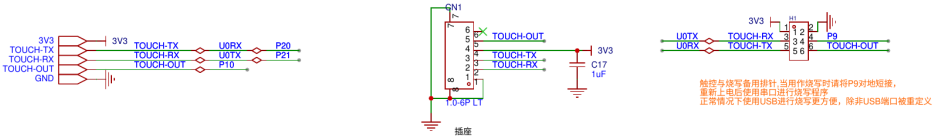

图 14　指纹模块电路图

我们先来写头文件

指纹模块头文件

```c
#ifndef __FINGERPRINT_DRIVER_H_
#define __FINGERPRINT_DRIVER_H_

#include "driver/uart.h"
#include "driver/gpio.h"

/// 下面的配置可以直接写死，也可以在 menuconfig 里面配置
#define ECHO_TEST_TXD (CONFIG_EXAMPLE_UART_TXD)
#define ECHO_TEST_RXD (CONFIG_EXAMPLE_UART_RXD)
#define ECHO_TEST_RTS (UART_PIN_NO_CHANGE)
#define ECHO_TEST_CTS (UART_PIN_NO_CHANGE)

#define ECHO_UART_PORT_NUM (CONFIG_EXAMPLE_UART_PORT_NUM)
#define ECHO_UART_BAUD_RATE (CONFIG_EXAMPLE_UART_BAUD_RATE)
#define ECHO_TASK_STACK_SIZE (CONFIG_EXAMPLE_TASK_STACK_SIZE)

#define BUF_SIZE (1024)

#define TOUCH_INT GPIO_NUM_8

/// 初始化指纹模块
void FINGERPRINT_Init(void);

/// 获取指纹芯片的序列号
void get_chip_sn(void);

/// 获取指纹图像
int get_image(void);

/// 获取指纹特征
int gen_char(void);

/// 搜索指纹
int search(void);

```

```
36  /// 读取指纹芯片配置参数
37  void read_sys_params(void);
38
39  #endif
```

然后编写头文件中接口的实现

指纹模块实现代码

```
1   #include "fingerprint_driver.h"
2
3   void FINGERPRINT_Init(void)
4   {
5       /* Configure parameters of an UART driver,
6        * communication pins and install the driver */
7       uart_config_t uart_config = {
8           .baud_rate = ECHO_UART_BAUD_RATE,
9           .data_bits = UART_DATA_8_BITS,
10          .parity = UART_PARITY_DISABLE,
11          .stop_bits = UART_STOP_BITS_1,
12          .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
13          .source_clk = UART_SCLK_DEFAULT,
14      };
15      int intr_alloc_flags = 0;
16
17      ESP_ERROR_CHECK(uart_driver_install(
18          ECHO_UART_PORT_NUM,
19          BUF_SIZE * 2, 0, 0, NULL,
20          intr_alloc_flags));
21      ESP_ERROR_CHECK(uart_param_config(
22          ECHO_UART_PORT_NUM, &uart_config));
23      ESP_ERROR_CHECK(uart_set_pin(
24          ECHO_UART_PORT_NUM,
25          ECHO_TEST_TXD,
26          ECHO_TEST_RXD,
27          ECHO_TEST_RTS,
28          ECHO_TEST_CTS));
29
30      // 中断
31      gpio_config_t io_conf;
32      io_conf.intr_type = GPIO_INTR_NEGEDGE;
33      io_conf.mode = GPIO_MODE_INPUT;
34      io_conf.pin_bit_mask = (1ULL << TOUCH_INT);
35      io_conf.pull_up_en = 1;
36      gpio_config(&io_conf);
37
38      printf("指纹模块初始化成功。\r\n");
39  }
40
41  void get_chip_sn(void)
42  {
43      vTaskDelay(200 / portTICK_PERIOD_MS);
44      uint8_t *data = (uint8_t *)malloc(BUF_SIZE);
```

```
45
46      // 获取芯片唯一序列号 0x34。确认码=00H 表示 OK；确认码=01H 表示收包有错。
47      uint8_t PS_GetChipSN[13] = {0xEF, 0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0x01, 0x00, 0x04, 0x34, 0x00, 0x00, 0x39};
48      uart_write_bytes(ECHO_UART_PORT_NUM, (const char *)PS_GetChipSN, 13);
49
50      // Read data from the UART
51          int len = uart_read_bytes(ECHO_UART_PORT_NUM, data, (BUF_SIZE - 1), 2000 / portTICK_PERIOD_MS);
52
53      if (len)
54      {
55          if (data[6] == 0x07 && data[9] == 0x00)
56          {
57              printf("chip sn: %.32s\r\n", &data[10]);
58          }
59      }
60
61      free(data);
62  }
63
64  // 检测是否有手指放在模组上面
65  int get_image(void)
66  {
67      uint8_t *data = (uint8_t *)malloc(BUF_SIZE);
68
69      // 验证用获取图像 0x01，验证指纹时，探测手指，探测到后录入指纹图像存于图像缓冲区。返回确认码表示：录入成功、无手指等。
70      uint8_t PS_GetImageBuffer[12] = {0xEF, 0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0x01, 0x00, 0x03, 0x01, 0x00, 0x05};
71
72      uart_write_bytes(ECHO_UART_PORT_NUM, (const char *)PS_GetImageBuffer, 12);
73
74          int len = uart_read_bytes(ECHO_UART_PORT_NUM, data, (BUF_SIZE - 1), 2000 / portTICK_PERIOD_MS);
75
76      int result = 0xFF;
77
78      if (len)
79      {
80          if (data[6] == 0x07)
81          {
82              if (data[9] == 0x00)
83              {
84                  result = 0;
85              }
86              else if (data[9] == 0x01)
87              {
88                  result = 1;
89              }
90              else if (data[9] == 0x02)
91              {
92                  result = 2;
93              }
```

```
        }
    }

    free(data);

    return result;
}

int gen_char(void)
{
    uint8_t *data = (uint8_t *)malloc(BUF_SIZE);

    // 生成特征 0x02，将图像缓冲区中的原始图像生成指纹特征文件存于模板缓冲区。
    uint8_t PS_GenCharBuffer[13] = {0xEF, 0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0x01, 0x00, 0x04, 0x02, 0x01, 0x00, 0x08};

    uart_write_bytes(ECHO_UART_PORT_NUM, (const char *)PS_GenCharBuffer, 13);

        int len = uart_read_bytes(ECHO_UART_PORT_NUM, data, (BUF_SIZE - 1), 2000 / portTICK_PERIOD_MS);

    int result = 0xFF;

    if (len)
    {
        if (data[6] == 0x07)
        {
            result = data[9];
        }
    }

    free(data);

    return result;
}

int search(void)
{
    uint8_t *data = (uint8_t *)malloc(BUF_SIZE);

    // 搜索指纹 0x04，以模板缓冲区中的特征文件搜索整个或部分指纹库。若搜索到，则返回页码。加密等级设置为 0 或 1 情况下支持此功能。
    uint8_t PS_SearchBuffer[17] = {0xEF, 0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0x01, 0x00, 0x08, 0x04, 0x01, 0x00, 0x00, 0xFF, 0xFF, 0x02, 0x0C};

    uart_write_bytes(ECHO_UART_PORT_NUM, (const char *)PS_SearchBuffer, 17);

        int len = uart_read_bytes(ECHO_UART_PORT_NUM, data, (BUF_SIZE - 1), 2000 / portTICK_PERIOD_MS);

    int result = 0xFF;

    if (len)
    {
```

```
        if (data[6] == 0x07)
        {
            result = data[9];
        }
    }

    free(data);

    return result;
}

void read_sys_params(void)
{
    uint8_t *data = (uint8_t *)malloc(BUF_SIZE);

    // 获取模组基本参数 0x0F，读取模组的基本参数（波特率，包大小等）。参数表前 16 个字节存放了模组的基本通讯和配置信息，称为模组的基本参数。
    uint8_t PS_ReadSysPara[12] = {0xEF, 0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0x01, 0x00, 0x03, 0x0F, 0x00, 0x13};

    uart_write_bytes(ECHO_UART_PORT_NUM, (const char *)PS_ReadSysPara, 12);

    int len = uart_read_bytes(ECHO_UART_PORT_NUM, data, (BUF_SIZE - 1), 2000 / portTICK_PERIOD_MS);

    if (len)
    {
        if (data[6] == 0x07)
        {
            if (data[9] == 0x00)
            {
                int register_count = (data[10] << 8) | data[11];
                printf("register count ==> %d\r\n", register_count);
                int fingerprint_template_size = (data[12] << 8) | data[13];
                printf("finger print template size ==> %d\r\n", fingerprint_template_size);
                int fingerprint_library_size = (data[14] << 8) | data[15];
                printf("finger print library size ==> %d\r\n", fingerprint_library_size);
                int score_level = (data[16] << 8) | data[17];
                printf("score level ==> %d\r\n", score_level);
                // device address
                printf("device address ==> 0x");
                for (int i = 0; i < 4; i++)
                {
                    printf("%02X ", data[18 + i]);
                }
                printf("\r\n");
                // data packet size
                int packet_size = (data[22] << 8) | data[23];
                if (packet_size == 0)
                {
                    printf("packet size ==> 32 bytes\r\n");
                }
                else if (packet_size == 1)
                {
```

```
194                 printf("packet size ==> 64 bytes\r\n");
195             }
196             else if (packet_size == 2)
197             {
198                 printf("packet size ==> 128 bytes\r\n");
199             }
200             else if (packet_size == 3)
201             {
202                 printf("packet size ==> 256 bytes\r\n");
203             }
204             // baud rate
205             int baud_rate = (data[24] << 8) | data[25];
206             printf("baud rate ==> %d\r\n", 9600 * baud_rate);
207         }
208         else if (data[9] == 0x01)
209         {
210             printf("send packet error\r\n");
211         }
212     }
213   }
214
215   free(data);
216 }
```

九　蓝牙功能

实现了蓝牙功能和我们后面的 WIFI 功能，其实就可以自己编写代码作为固件烧录到 ESP32C3 里面了。这样也可以作为 STM32 的外设来使用了。这是 ESP32 所具有的独特的功能。

蓝牙技术是一种无线通讯技术，广泛用于短距离内的数据交换。在蓝牙技术中，"Bluedroid"和"BLE"（Bluetooth Low Energy）是两个重要的概念，它们分别代表了蓝牙技术的不同方面。

Bluedroid

Bluedroid 是 Android 操作系统用于实现蓝牙功能的软件栈。在 Android 4.2 版本中引入，Bluedroid 取代了之前的 BlueZ 作为 Android 平台的蓝牙协议栈。Bluedroid 是由 Broadcom 公司开发并贡献给 Android 开源项目的（AOSP），它支持经典蓝牙以及蓝牙低功耗（BLE）。

Bluedroid 协议栈设计目的是为了提供一个更轻量级、更高效的蓝牙协议栈，以适应移动设备对资源的紧张需求。它包括了蓝牙核心协议、各种蓝牙配置文件（如 HSP、A2DP、AVRCP 等）和 BLE 相关的服务和特性。

BLE（Bluetooth Low Energy）

BLE，即蓝牙低功耗技术，是蓝牙 4.0 规范中引入的一项重要技术。与传统的蓝牙技术（现在通常称为经典蓝牙）相比，BLE 主要设计目标是实现极低的功耗，以延长设备的电池使用寿命，非常适合于需要长期运行但只需偶尔传输少量数据的应用场景，如健康和健身监测设备、智能家居设备等。

BLE 实现了一套与经典蓝牙不同的通信协议，包括低功耗的物理层、链路层协议以及应用层协议。BLE 设备可以以极低的能耗状态长时间待机，只有在需要通信时才唤醒，这使得使用小型电池的设备也能达到数月甚至数年的电池寿命。

总的来说，Bluedroid 是 Android 平台上用于实现蓝牙通信功能的软件栈，而 BLE 则是蓝牙技术中的一种用于实现低功耗通信的标准。两者共同为 Android 设备提供了广泛的蓝牙通信能力，满足了不同应用场景下的需求。

9.1 GATT SERVER 代码讲解

在本文档中，我们回顾了在 ESP32 上实现蓝牙低功耗（BLE）通用属性配置文件（GATT）服务器的 GATT SERVER 示例代码。这个示例围绕两个应用程序配置文件和一系列事件设计，这些事件被处理以执行一系列配置步骤，例如定义广告参数、更新连接参数以及创建服务和特性。此外，这个示例处理读写事件，包括一个写长特性请求，它将传入数据分割成块，以便数据能够适应属性协议（ATT）消息。本文档遵循程序工作流程，并分解代码以便理解每个部分和实现背后的原因。

9.1.1 头文件

头文件

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/event_groups.h"
#include "esp_system.h"
#include "esp_log.h"
#include "nvs_flash.h"
#include "esp_bt.h"
#include "esp_gap_ble_api.h"
#include "esp_gatts_api.h"
#include "esp_bt_defs.h"
#include "esp_bt_main.h"
#include "esp_gatt_common_api.h"
#include "sdkconfig.h"
```

这些头文件是运行 FreeRTOS 和底层系统组件所必需的，包括日志功能和一个用于在非易失性闪存中存储数据的库（也就是 flash）。我们对 `"esp_bt.h"`、`"esp_bt_main.h"`、`"esp_gap_ble_api.h"` 和 `"esp_gatts_api.h"` 特别感兴趣，这些文件暴露了实现此示例所需的 BLE API。

- `esp_bt.h`：从主机侧实现蓝牙控制器和 VHCI 配置程序。
- `esp_bt_main.h`：实现 Bluedroid 栈协议的初始化和启用。
- `esp_gap_ble_api.h`：实现 GAP 配置，如广告和连接参数。
- `esp_gatts_api.h`：实现 GATT 配置，如创建服务和特性。

Info

VHCI（Virtual Host Controller Interface）是一个虚拟的主机控制器接口，它通常用于软件或硬件模拟中，以模拟主机控制器的行为。在不同的上下文中，VHCI 可以指代不同的技术或应用，但基本概念相似，都是提供一个虚拟的接口来模拟实际的硬件或软件行为。

在蓝牙技术领域，VHCI 特别指向用于模拟蓝牙主机控制器（Host Controller）的接口。这可以用于蓝牙协议栈的开发和测试，允许开发者在没有实际蓝牙硬件的情况下模拟蓝牙设备的行为。通过 VHCI，软件可以模拟发送和接收蓝牙数据包，从而测试蓝牙应用程序和服务的实现。

在其他情况下，VHCI 也可以用于 USB（通用串行总线）技术，作为一个虚拟的 USB 主机控制器，来模拟 USB 设备的连接和通信。

总的来说，VHCI 是一个非常有用的工具，特别是在设备驱动和协议栈开发的早期阶段，它可以帮助开发者在没有实际硬件的情况下进行软件开发和测试。

9.1.2 入口函数

入口函数是 `app_main()` 函数。

复制工程

```
1  void app_main()
2  {
3      esp_err_t ret;
4      // Initialize NVS.
5      // 初始化 flash，很重要。
6      ret = nvs_flash_init();
7      if (ret == ESP_ERR_NVS_NO_FREE_PAGES
8          || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
9          ESP_ERROR_CHECK(nvs_flash_erase());
10         ret = nvs_flash_init();
11     }
12     ESP_ERROR_CHECK(ret);
13
14     esp_bt_controller_config_t bt_cfg = BT_CONTROLLER_INIT_CONFIG_DEFAULT();
15     esp_bt_controller_init(&bt_cfg);
16     esp_bt_controller_enable(ESP_BT_MODE_BLE);
17     esp_bluedroid_init();
18     esp_bluedroid_enable();
19
20     esp_ble_gatts_register_callback(gatts_event_handler);
21     esp_ble_gap_register_callback(gap_event_handler);
22     esp_ble_gatts_app_register(PROFILE_A_APP_ID);
23     esp_ble_gatts_app_register(PROFILE_B_APP_ID);
24     esp_ble_gatt_set_local_mtu(512);
25 }
```

主函数首先初始化非易失性存储库。这个库允许在 flash 中保存键值对，并被一些组件（如 Wi-Fi 库）用来保存 SSID 和密码：

初始化 flash

```
1  ret = nvs_flash_init();
```

9.1.3 蓝牙控制器和栈协议初始化(BT Controller and Stack Initialization)

主函数还通过首先创建一个名为 `esp_bt_controller_config_t` 的蓝牙控制器配置结构体来初始化蓝牙控制器，该结构体使用 `BT_CONTROLLER_INIT_CONFIG_DEFAULT()` 宏生成的默认设置。蓝牙控制器在控制器侧实现了主控制器接口（HCI）、链路层（LL）和物理层（PHY）。蓝牙控制器对用户应用程序是不可见的，它处理 BLE 栈协议的底层。控制器配置包括设置蓝牙控制器栈协议大小、优先级和 HCI 波特率。使用创建的设置，通过 `esp_bt_controller_init()` 函数初始化并启用蓝牙控制器：

初始化蓝牙控制器

```
1  esp_bt_controller_config_t bt_cfg = BT_CONTROLLER_INIT_CONFIG_DEFAULT();
2  ret = esp_bt_controller_init(&bt_cfg);
```

接下来，控制器使能为 BLE 模式。

使能为 BLE 模式

```
1  esp_bt_controller_enable(ESP_BT_MODE_BLE);
```

> **Info**
>
> 如果想要使用双模式（BLE + BT），控制器应该使能为 `ESP_BT_MODE_BTDM` 。

支持四种蓝牙模式：

1. `ESP_BT_MODE_IDLE`：蓝牙未运行
2. `ESP_BT_MODE_BLE`：BLE 模式
3. `ESP_BT_MODE_CLASSIC_BT`：经典蓝牙模式
4. `ESP_BT_MODE_BTDM`：双模式（BLE + 经典蓝牙）

在蓝牙控制器初始化之后，Bluedroid 栈协议（包括经典蓝牙和 BLE 的共同定义和 API）通过使用以下方式被初始化和启用：

初始化 Bluedroid 栈协议 API

```
1  esp_bluedroid_init();
2  esp_bluedroid_enable();
```

此时程序流程中的蓝牙栈协议已经启动并运行，但应用程序的功能尚未定义。功能是通过响应事件来定义的，例如当另一个设备尝试读取或写入参数并建立连接时会发生什么。两个主要的事件

管理器是 GAP 和 GATT 事件处理器。应用程序需要为每个事件处理器注册一个回调函数，以便让应用程序知道哪些函数将处理 GAP 和 GATT 事件：

注册事件处理的回调函数

```
1 esp_ble_gatts_register_callback(gatts_event_handler);
2 esp_ble_gap_register_callback(gap_event_handler);
```

函数 `gatts_event_handler()` 和 `gap_event_handler()` 处理所有从 BLE 栈协议推送给应用程序的事件。

9.1.4 应用程序配置文件(APPLICATION PROFILES)

如下图所示，GATT 服务器示例应用程序通过使用应用程序配置文件来组织。每个应用程序配置文件描述了一种分组功能的方式，这些功能是为一个客户端应用程序设计的，例如在智能手机或平板电脑上运行的移动应用。通过这种方式，单一设计，通过不同的应用程序配置文件启用，可以在被不同的智能手机应用使用时表现出不同的行为，允许服务器根据正在使用的客户端应用程序做出不同的反应。实际上，每个配置文件被客户端视为一个独立的 BLE 服务。客户端可以自行区分它感兴趣的服务。

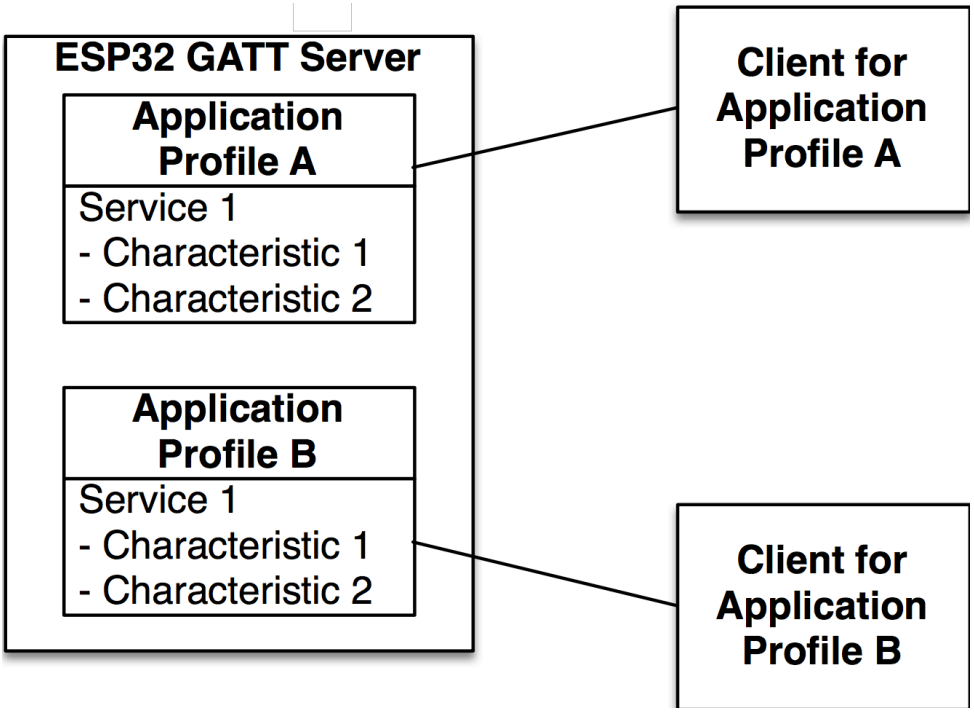

图 15　GATT 服务器

每个配置文件都定义为一个结构体，其中结构体成员取决于在该应用程序配置文件中实现的服务和特性。成员还包括一个 GATT 接口、应用程序 ID 、连接 ID 和一个回调函数来处理配置文件事件。在这个示例中，每个配置文件由以下组成：

- GATT 接口
- 应用程序 ID
- 连接 ID
- 服务句柄
- 服务 ID
- 特性句柄
- 特性 UUID
- 属性权限
- 特性属性
- 客户端特性配置描述符句柄
- 客户端特性配置描述符 UUID

从这个结构中可以看出，这个配置文件被设计为拥有一个服务和一个特性，并且该特性有一个描述符。服务有一个句柄和一个 ID，同样每个特性都有一个句柄、一个 UUID、属性权限和属性。此外，如果特性支持通知或指示，则必须实现一个客户端特性配置描述符（CCCD），这是一个额外的属性，描述通知或指示是否启用，并定义特性如何被特定客户端配置。这个描述符也有一个句柄和一个 UUID。

结构实现是：

结构体的定义

```
struct gatts_profile_inst {
    esp_gatts_cb_t gatts_cb;
    uint16_t gatts_if;
    uint16_t app_id;
    uint16_t conn_id;
    uint16_t service_handle;
    esp_gatt_srvc_id_t service_id;
    uint16_t char_handle;
    esp_bt_uuid_t char_uuid;
    esp_gatt_perm_t perm;
    esp_gatt_char_prop_t property;
    uint16_t descr_handle;
    esp_bt_uuid_t descr_uuid;
};
```

应用程序配置文件存储在一个数组中，并分配了相应的回调函数 `gatts_profile_a_event_handler()` 和 `gatts_profile_b_event_handler()`。GATT 客户端上的不同应用程序使用不同的接口，由 `gatts_if` 参数表示。对于初始化，此参数设置为 `ESP_GATT_IF_NONE`，意味着应用程序配置文件尚未链接到任何客户端。

为不同的应用注册回调函数

```
struct gatts_profile_inst heart_rate_profile_tab[PROFILE_NUM] = {
    [PROFILE_APP_IDX] = {
        .gatts_cb = gatts_profile_event_handler,
        .gatts_if = ESP_GATT_IF_NONE, /* Not get the gatt_if, so initial is ESP_GATT_IF_NONE */
    },
};
```

最后，使用应用程序 ID 注册应用程序配置文件，这是一个用户分配的数字，用于标识每个配置文件。通过这种方式，一个服务器可以运行多个应用程序配置文件。

使用应用 ID 注册配置文件

```
esp_ble_gatts_app_register(ESP_APP_ID);
```

9.1.5 gatts_event_handler 函数

gatts_event_handler

```
void gatts_event_handler(
    esp_gatts_cb_event_t event,
    esp_gatt_if_t gatts_if,
    esp_ble_gatts_cb_param_t *param)
{
    /* If event is register event, store the gatts_if for each profile */
    if (event == ESP_GATTS_REG_EVT)
    {
        if (param->reg.status == ESP_GATT_OK)
        {
            heart_rate_profile_tab[PROFILE_APP_IDX].gatts_if = gatts_if;
        }
        else
        {
            ESP_LOGE(GATTS_TABLE_TAG, "reg app failed, app_id %04x, status %d",
                    param->reg.app_id,
                    param->reg.status);
            return;
        }
    }
    do
    {
        int idx;
        for (idx = 0; idx < PROFILE_NUM; idx++)
        {
            /* ESP_GATT_IF_NONE, not specify a certain gatt_if, need to call every profile cb function */
            if (gatts_if == ESP_GATT_IF_NONE
                || gatts_if == heart_rate_profile_tab[idx].gatts_if)
            {
                if (heart_rate_profile_tab[idx].gatts_cb)
                {
                    heart_rate_profile_tab[idx].gatts_cb(event, gatts_if, param);
```

```
33              }
34            }
35          }
36      } while (0);
37  }
```

第 32 行是最关键的，就是执行回调函数。所以我们接下来实现这个回调函数。

9.1.6 gatts_profile_event_handler 函数

复制工程

```
1   void gatts_profile_event_handler(
2       esp_gatts_cb_event_t event,
3       esp_gatt_if_t gatts_if,
4       esp_ble_gatts_cb_param_t *param)
5   {
6       switch (event)
7       {
8       case ESP_GATTS_REG_EVT:
9       {
10          esp_ble_gap_set_device_name(SAMPLE_DEVICE_NAME);
11          esp_ble_gap_config_adv_data_raw(raw_adv_data, sizeof(raw_adv_data));
12          adv_config_done |= ADV_CONFIG_FLAG;
13          esp_ble_gap_config_scan_rsp_data_raw(raw_scan_rsp_data, sizeof(raw_scan_rsp_data));
14          adv_config_done |= SCAN_RSP_CONFIG_FLAG;
15          esp_ble_gatts_create_attr_tab(gatt_db, gatts_if, HRS_IDX_NB, SVC_INST_ID);
16      }
17      break;
18      case ESP_GATTS_READ_EVT:
19          break;
20      case ESP_GATTS_WRITE_EVT:
21          if (!param->write.is_prep)
22          {
23              esp_log_buffer_hex("接收到的数据：", param->write.value, param->write.len);
24              if (param->write.value[0] == 'a'
25                  && param->write.value[1] == 't'
26                  && param->write.value[2] == 'g'
27                  && param->write.value[3] == 'u'
28                  && param->write.value[4] == 'i'
29                  && param->write.value[5] == 'g'
30                  && param->write.value[6] == 'u')
31              {
32                  printf("通过蓝牙开锁成功\r\n");
33                  MOTOR_Open_lock();
34              }
35              if (heart_rate_handle_table[IDX_CHAR_CFG_A] == param->write.handle
36                  && param->write.len == 2)
37              {
38                  uint16_t descr_value = param->write.value[1] << 8 | param->write.value[0];
39                  if (descr_value == 0x0001)
40                  {
```

```
                uint8_t notify_data[] = "atguigu";
                // the size of notify_data[] need less than MTU size
                esp_ble_gatts_send_indicate(
                    gatts_if,
                    param->write.conn_id,
                    heart_rate_handle_table[IDX_CHAR_VAL_A],
                    sizeof(notify_data),
                    notify_data,
                    false);
            }
            else if (descr_value == 0x0002)
            {
                // ...
            }
            else if (descr_value == 0x0000)
            {
                // ...
            }
            else
            {
                // ...
            }
        }
        /* send response when param->write.need_rsp is true*/
        if (param->write.need_rsp)
        {
            esp_ble_gatts_send_response(
                gatts_if,
                param->write.conn_id,
                param->write.trans_id,
                ESP_GATT_OK,
                NULL);
        }
    }
    else
    {
        /* handle prepare write */
        example_prepare_write_event_env(gatts_if, &prepare_write_env, param);
    }
    break;
case ESP_GATTS_EXEC_WRITE_EVT:
    example_exec_write_event_env(&prepare_write_env, param);
    break;
case ESP_GATTS_MTU_EVT:
case ESP_GATTS_CONF_EVT:
case ESP_GATTS_START_EVT:
    break;
case ESP_GATTS_CONNECT_EVT:
    ESP_LOGI(GATTS_TABLE_TAG, "ESP_GATTS_CONNECT_EVT, conn_id = %d", param->connect.conn_id);
    esp_log_buffer_hex(GATTS_TABLE_TAG, param->connect.remote_bda, 6);
    esp_ble_conn_update_params_t conn_params = {0};
    memcpy(conn_params.bda, param->connect.remote_bda, sizeof(esp_bd_addr_t));
```

```
        /* For the iOS system, please refer to Apple official documents about the BLE connection parameters restrictions. */
        conn_params.latency = 0;
        conn_params.max_int = 0x20; // max_int = 0x20*1.25ms = 40ms
        conn_params.min_int = 0x10; // min_int = 0x10*1.25ms = 20ms
        conn_params.timeout = 400;  // timeout = 400*10ms = 4000ms
        // start sent the update connection parameters to the peer device.
        esp_ble_gap_update_conn_params(&conn_params);
        break;
    case ESP_GATTS_DISCONNECT_EVT:
        ESP_LOGI(GATTS_TABLE_TAG, "ESP_GATTS_DISCONNECT_EVT, reason = 0x%x", param->disconnect.reason);
        esp_ble_gap_start_advertising(&adv_params);
        break;
    case ESP_GATTS_CREAT_ATTR_TAB_EVT:
    {
        if (param->add_attr_tab.status != ESP_GATT_OK)
        {
            ESP_LOGE(GATTS_TABLE_TAG, "create attribute table failed, error code=0x%x", param->add_attr_tab.status);
        }
        else if (param->add_attr_tab.num_handle != HRS_IDX_NB)
        {
            ESP_LOGE(GATTS_TABLE_TAG, "create attribute table abnormally, num_handle (%d) \
                    doesn't equal to HRS_IDX_NB(%d)",
                    param->add_attr_tab.num_handle, HRS_IDX_NB);
        }
        else
        {
            ESP_LOGI(GATTS_TABLE_TAG, "create attribute table successfully, the number handle = %d\n", param->add_attr_tab.num_handle);
            memcpy(heart_rate_handle_table, param->add_attr_tab.handles, sizeof(heart_rate_handle_table));
            esp_ble_gatts_start_service(heart_rate_handle_table[IDX_SVC]);
        }
        break;
    }
    case ESP_GATTS_STOP_EVT:
    case ESP_GATTS_OPEN_EVT:
    case ESP_GATTS_CANCEL_OPEN_EVT:
    case ESP_GATTS_CLOSE_EVT:
    case ESP_GATTS_LISTEN_EVT:
    case ESP_GATTS_CONGEST_EVT:
    case ESP_GATTS_UNREG_EVT:
    case ESP_GATTS_DELETE_EVT:
    default:
        break;
    }
}
```