# ESP32简介

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。

我们使用的芯片是 ESP32-C3 。

# 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。

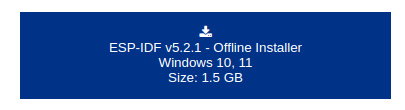
限定条件：

* 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）
* 各种路径中不要有中文！

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。

## 离线安装ESP-IDF

点击 https://dl.espressif.com/dl/esp-idf/?idf=4.4 下载离线安装包。



## 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

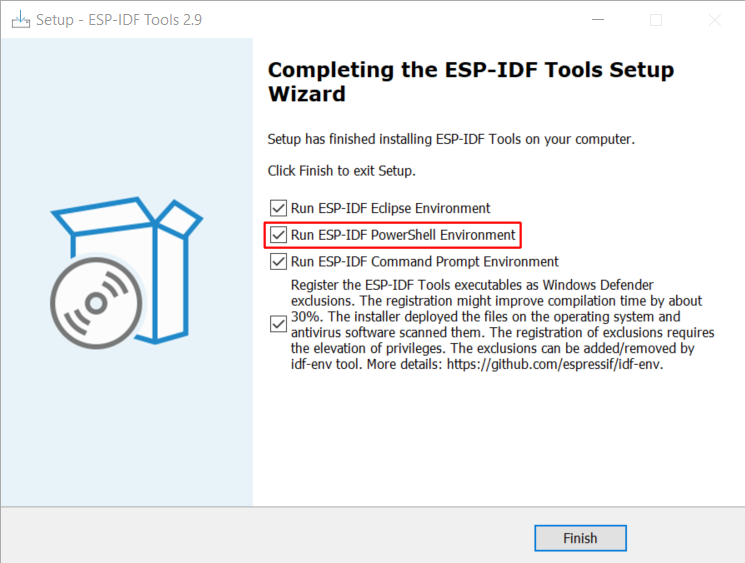
安装程序允许将程序下载到现有的ESP-IDF目录。

推荐将ESP-IDF下载到 %userprofile%\Desktop\esp-idf目录下，其中%userprofile%代表家目录。

## 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：



## 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。

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：

$ cd %userprofile%\esp  
$ xcopy /e /i %IDF\_PATH%\examples\get-started\hello\_world hello\_world

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。

## 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。

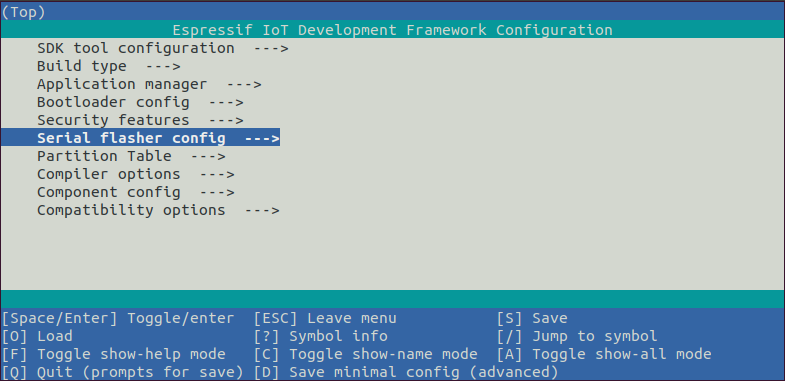
## 配置工程

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 。

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：



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。

## 编译工程

请使用以下命令，编译烧录工程：

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。

$ 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。

## 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：

idf.py flash

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。

## 常规操作

在烧录过程中，会看到类似如下的输出日志：

...  
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。

## 监视输出

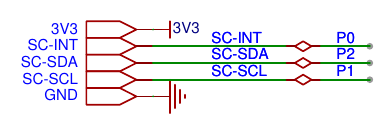
使用 串口助手 监视输出和调试。

当要进行烧写时，请关闭串口助手！

# 电容键盘模块

电容键盘模块的参考代码位于examples\peripherals\gpio\generic\_gpio文件夹。

电容键盘使用 SC12B 。电路图如下：



先引入这个模块的原因是为了让大家熟悉如何操作 ESP32 的 GPIO 引脚。和 STM32 的 HAL 库的使用是类似的。

在编写代码之前，让我们先来创建项目。

* 在 ~/esp 文件夹中创建新的文件夹 atguigu-gpio-example 。
* 在 atguigu-gpio-example 文件夹中创建文件 CMakeLists.txt 。
* 在 atguigu-gpio-example 文件夹中创建 main 文件夹。
* 在 main 文件夹中创建两个文件。gpio\_main.c 和 CMakeLists.txt 。

目录结构如下：

atguigu-gpio-example  
 - CMakeLists.txt  
 - main  
 - CMakeLists.txt  
 - gpio\_main.c

先来编写 atguigu-gpio-example/CMakeLists.txt 文件。内容如下：

# cmake版本管理工具的最小版本号  
cmake\_minimum\_required(VERSION 3.16)  
  
# 项目的cmake文件路径，包含在 ESP32 的 SDK 中  
include($ENV{IDF\_PATH}/tools/cmake/project.cmake)  
# 项目名称  
project(atguigu\_gpio\_example)

然后编写 atguigu-gpio-example/main/CMakeLists.txt 文件。内容如下：

idf\_component\_register(  
 # 要包含的源文件  
 SRCS "gpio\_main.c"  
 # 要包含的路径  
 INCLUDE\_DIRS ""  
)

接下来我们就可以来正式编写 atguigu-gpio-example/main/gpio\_main.c 文件的代码了。

为了方便上手，我们把代码写在一个文件中。但是分段讲解，所以大家只需要把代码从上往下写就可以了。

/// 整型定义  
#include <inttypes.h>  
/// RTOS通用API  
#include "freertos/FreeRTOS.h"  
/// RTOS任务相关API  
#include "freertos/task.h"  
/// RTOS队列相关API  
#include "freertos/queue.h"  
/// ESP32的GPIO接口  
#include "driver/gpio.h"

接下来，我们定义一些引脚操作的宏定义，增强代码可读性。

/// SCL时钟引脚  
#define SC12B\_SCL GPIO\_NUM\_1  
/// SDA数据引脚  
#define SC12B\_SDA GPIO\_NUM\_2  
/// INT中断引脚  
#define SC12B\_INT GPIO\_NUM\_0  
  
/// 设置SDA引脚为输入方向  
#define I2C\_SDA\_IN gpio\_set\_direction(SC12B\_SDA, GPIO\_MODE\_INPUT)  
/// 设置SDA引脚为输出方向  
#define I2C\_SDA\_OUT gpio\_set\_direction(SC12B\_SDA, GPIO\_MODE\_OUTPUT)  
  
/// 拉高SCL引脚  
#define I2C\_SCL\_H gpio\_set\_level(SC12B\_SCL, 1)  
/// 拉低SCL引脚  
#define I2C\_SCL\_L gpio\_set\_level(SC12B\_SCL, 0)  
  
/// 拉高SDA引脚  
#define I2C\_SDA\_H gpio\_set\_level(SC12B\_SDA, 1)  
/// 拉低SDA引脚  
#define I2C\_SDA\_L gpio\_set\_level(SC12B\_SDA, 0)  
  
/// 读取SDA引脚电平的值  
#define I2C\_READ\_SDA gpio\_get\_level(SC12B\_SDA)

定义一个单位为毫秒的延时函数。使用了 vTaskDelay 方法来实现这一点。

void Delay\_ms(uint8\_t time)  
{  
 vTaskDelay(time / portTICK\_PERIOD\_MS);  
}

接下来我们通过以上定义的方法，来实现软件 协议。因为 ESP32-C3 通过 协议和电容键盘进行通信。

由于 协议大家已经比较熟悉了，所以电平的拉高拉低以及延时不做过多讲解。

开始 通信。

void I2C\_Start(void)  
{  
 I2C\_SDA\_OUT;  
 I2C\_SDA\_H;  
 I2C\_SCL\_H;  
 Delay\_ms(1);  
 I2C\_SDA\_L;  
 Delay\_ms(1);  
 I2C\_SCL\_L;  
 Delay\_ms(1);  
}

停止 通信。

void I2C\_Stop(void)  
{  
 I2C\_SCL\_L;  
 I2C\_SDA\_OUT;  
 I2C\_SDA\_L;  
 Delay\_ms(1);  
 I2C\_SCL\_H;  
 Delay\_ms(1);  
 I2C\_SDA\_H;  
}

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。成功则返回 0 。

uint8\_t I2C\_Wait\_Ack(void)  
{  
 uint8\_t ucErrTime = 0;  
 I2C\_SCL\_L;  
 I2C\_SDA\_IN;  
 Delay\_ms(1);  
 I2C\_SCL\_H;  
 Delay\_ms(1);  
 /// 一直尝试读取数据线的低电平  
 while (I2C\_READ\_SDA)  
 {  
 if (ucErrTime++ > 250)  
 {  
 return 1;  
 }  
 }  
 I2C\_SCL\_L;  
 return 0;  
}

实现发送 1 个字节的功能。

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);  
 I2C\_SCL\_H;  
 Delay\_ms(1);  
 d <<= 1;  
 }  
}

实现读取 1 个字节的功能。

uint8\_t I2C\_Read\_Byte(uint8\_t ack)  
{  
 uint8\_t i = 0;  
 uint8\_t receive = 0;  
 I2C\_SDA\_IN;  
 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);  
 return receive;  
}

实现发送数据并返回应答的功能。

uint8\_t SendByteAndGetNACK(uint8\_t data)  
{  
 I2C\_Send\_Byte(data);  
 return I2C\_Wait\_Ack();  
}

实现 SC12B 电容键盘读取按键的值的方法。

uint8\_t I2C\_Read\_Key(void)  
{  
 I2C\_Start();  
 if (SendByteAndGetNACK((0x40 << 1) | 0x01))  
 {  
 I2C\_Stop();  
 return 0;  
 }  
 uint8\_t i = 0;  
 uint8\_t k = 0;  
 I2C\_SDA\_IN;  
 while (8 > i)  
 {  
 i++;  
 I2C\_SCL\_L;  
 Delay\_ms(1);  
 I2C\_SCL\_H;  
 if (!k && I2C\_READ\_SDA)  
 {  
 k = i;  
 }  
 Delay\_ms(1);  
 }  
 if (k)  
 {  
 I2C\_Ack(1);  
 I2C\_Stop();  
 return k;  
 }  
 I2C\_Ack(0);  
 I2C\_SDA\_IN;  
 while (16 > i)  
 {  
 i++;  
 I2C\_SCL\_L;  
 Delay\_ms(1);  
 I2C\_SCL\_H;  
 if (!k && I2C\_READ\_SDA)  
 {  
 k = i;  
 }  
 Delay\_ms(1);  
 }  
 I2C\_Ack(1);  
 I2C\_Stop();  
 return k;  
}

注意，按到的键和显示的按键值可能不一样。例如，按了 "1" 可能返回 "77"，所以需要写代码校正。校正代码如下：

uint8\_t KEYBOARD\_read\_key(void)  
{  
 uint16\_t key = I2C\_Read\_Key();  
 if (key == 4)  
 return 1;  
 else if (key == 3)  
 return 2;  
 else if (key == 2)  
 return 3;  
 else if (key == 7)  
 return 4;  
 else if (key == 6)  
 return 5;  
 else if (key == 5)  
 return 6;  
 else if (key == 10)  
 return 7;  
 else if (key == 9)  
 return 8;  
 else if (key == 8)  
 return 9;  
 else if (key == 1)  
 return 0;  
 else if (key == 12)  
 return '#';  
 else if (key == 11)  
 return 'M';  
 return 255;  
}

每个电容键盘的校正数值可能不一样。

好的。有关键盘的 驱动和读取按键值的代码已经写好了。

然后我们编写GPIO初始化的代码。

/// GPIO初始化  
void KEYBOARD\_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;  
 // 选择要使用的引脚  
 io\_conf.pin\_bit\_mask = ((1ULL << SC12B\_SCL) | (1ULL << SC12B\_SDA));  
 // 禁用下拉  
 io\_conf.pull\_down\_en = 0;  
 // 使能上拉  
 io\_conf.pull\_up\_en = 1;  
 // 是GPIO配置生效  
 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);  
}

接下来我们实现处理按键中断的代码。

首先初始化一个保存 GPIO 中断事件的队列。

static QueueHandle\_t gpio\_event\_queue = NULL;

然后实现响应来自 GPIO 中断的回调函数。也就是说，当我们按键时，产生的中断会触发回调函数的执行。GPIO 引脚号会作为参数传入函数。

/// 当回调函数执行时，参数 arg 是产生中断的 GPIO 引脚号。  
static void IRAM\_ATTR gpio\_isr\_handler(void \*arg)  
{  
 uint32\_t gpio\_num = (uint32\_t)arg;  
 /// 将 GPIO 引脚号添加到 gpio\_event\_queue 队列中。  
 xQueueSendFromISR(gpio\_event\_queue, &gpio\_num, NULL);  
}

接下来实现处理中断事件的逻辑。处理中断事件的方法是不断的轮询中断事件队列，从中取出事件并处理。

static void process\_isr(void \*arg)  
{  
 uint32\_t gpio\_num;  
 for (;;)  
 {  
 /// 如果队列中有 GPIO 中断事件，将 GPIO 引脚号存储到 gpio\_num 变量中。  
 if (xQueueReceive(gpio\_event\_queue, &gpio\_num, portMAX\_DELAY))  
 {  
 /// 如果产生中断的GPIO引脚号是 GPIO\_NUM\_0，也就是键盘中断引脚。  
 if (gpio\_num == 0)  
 {  
 /// 读取按键值。  
 uint8\_t key\_num = KEYBOARD\_read\_key();  
 /// 打印到上位机。通过串口助手查看。  
 printf("press key: %d\r\n", key\_num);  
 }  
 }  
 }  
}

很显然，process\_isr 方法需要注册为一个 RTOS 任务。

static void ISR\_QUEUE\_Init(void)  
{  
 /// 创建一个队列，用来保存中断事件。  
 gpio\_event\_queue = xQueueCreate(10, sizeof(uint32\_t));  
 /// 将 process\_isr 注册为一个 RTOS 任务。  
 xTaskCreate(process\_isr, "process\_isr", 2048, NULL, 10, NULL);  
 /// 监控来自 GPIO\_NUM\_0 的中断。  
 gpio\_install\_isr\_service(0);  
 /// 来自 GPIO\_NUM\_0 也就是 SC12B\_INT 的中断触发的回调函数是 gpio\_isr\_handler 。  
 gpio\_isr\_handler\_add(SC12B\_INT, gpio\_isr\_handler, (void \*)SC12B\_INT);  
}

接下来终于可以编写入口函数 app\_main 了。注意 ESP32 程序的入口函数是 app\_main 。实际上这个入口函数也是被注册成了一个 RTOS 任务。

void app\_main(void)  
{  
 ISR\_QUEUE\_Init();  
 KEYBOARD\_init();  
}

接下来，我们可以编译并烧写程序了。可以一条命令搞定。首先需要 cd 到项目的文件夹。例如，在我的电脑上命令如下：

cd ~/esp/atguigu-gpio-example  
idf.py set-target esp32c3  
idf.py flash

然后就可以测试程序了。

# 红外遥控（RMT）

LED模块的参考代码位于examples/peripherals/rmt/led\_strip文件夹。

## 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。

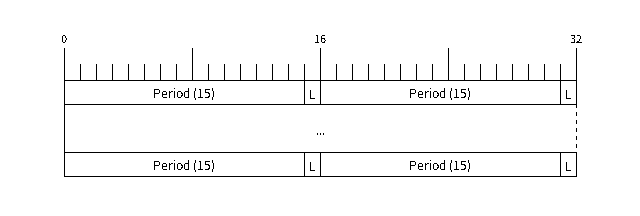
ESP32-C3 的 RMT 外设存在多个通道，每个通道都可以独立配置为发射器或接收器。

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

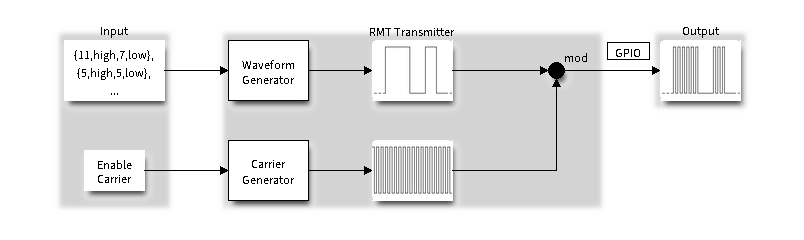
## RMT符号的内存布局

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。



## RMT发射器概述

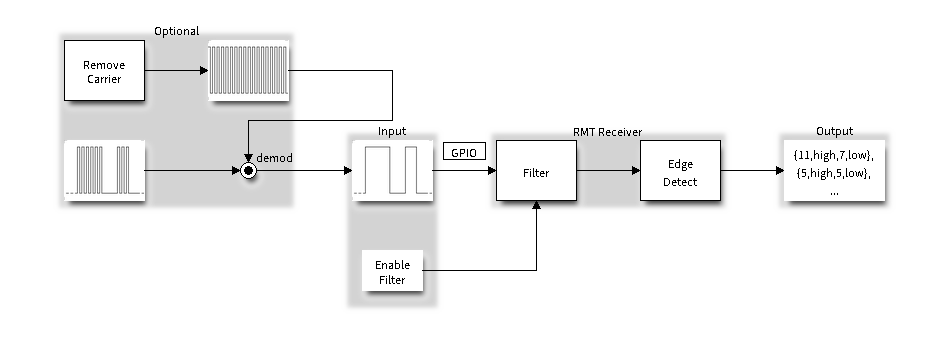
RMT 发送通道 (TX Channel) 的数据路径和控制路径如下图所示：



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。

## 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：



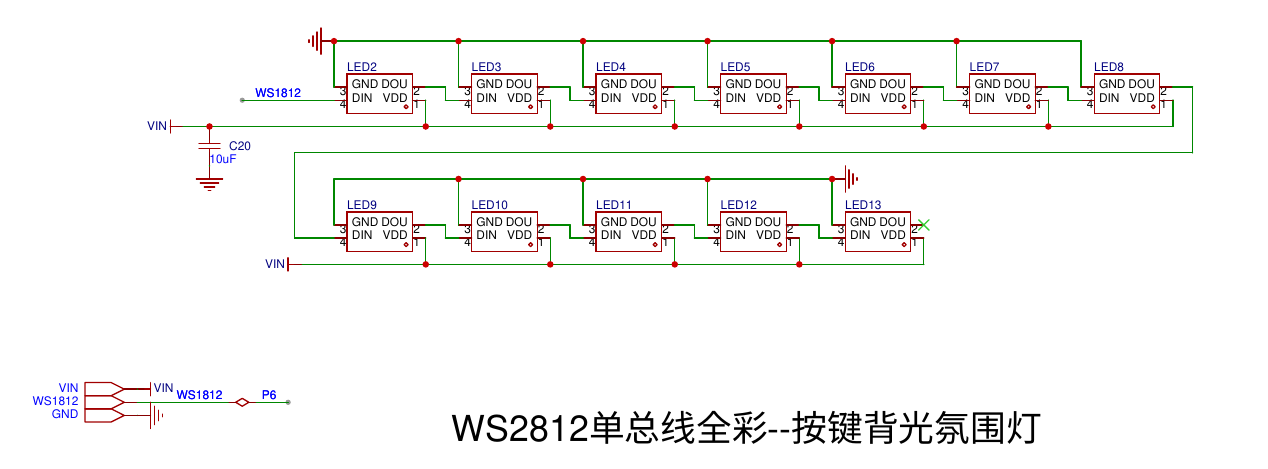
RMT 接收器可以对输入信号采样，将其转换为 RMT 数据格式，并将数据存储在内存中。还可以向接收器提供输入信号的基本特征，使其识别信号停止条件，并过滤掉信号干扰和噪声。RMT 外设还支持从基准信号中解调出高频载波信号。

## 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 ESP32-C3 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。

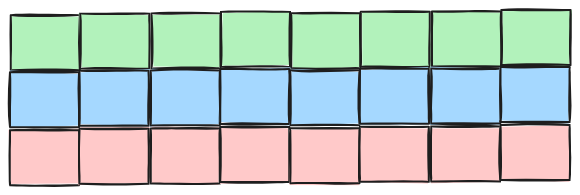
原理图如下：



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：

* 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。
* 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。
* 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 。



## 代码实现

先来编写 atguigu-led-example/CMakeLists.txt 文件。内容如下：

cmake\_minimum\_required(VERSION 3.16)  
  
include($ENV{IDF\_PATH}/tools/cmake/project.cmake)  
project(atguigu\_led\_example)

然后编写 atguigu-led-example/main/CMakeLists.txt 文件。内容如下：

idf\_component\_register(  
 SRCS "led\_main.c"  
 INCLUDE\_DIRS ""  
)

接下来我们就可以来正式编写 atguigu-led-example/main/led\_main.c 文件的代码了。

我们还是从上向下写代码。

首先引入一些头文件。

#include <inttypes.h>  
#include <string.h>  
#include "freertos/FreeRTOS.h"  
#include "freertos/task.h"  
/// ESP32 错误处理的API  
#include "esp\_check.h"  
/// RMT编码器相关API  
#include "driver/rmt\_encoder.h"  
/// RMT发送数据的API  
#include "driver/rmt\_tx.h"

我们的LED条带共有12个LED灯。而LED灯需要的波形的频率很高。我们定义两个宏。

// 10MHz的分辨率，1 tick = 0.1us，LED条带需要高分辨率。  
#define RMT\_LED\_STRIP\_RESOLUTION\_HZ 10000000  
// 只需要一个引脚，引脚号为6。  
#define RMT\_LED\_STRIP\_GPIO\_NUM GPIO\_NUM\_6  
  
// 共12个LED灯。  
#define LED\_NUMBERS 12

以下代码大部分来自示例代码。很多细节无需关心，只需要会修改可以实现我们的功能就可以了。

我们定义几个结构体。

// LED编码器配置结构体，主要配置编码器的频率。  
typedef struct  
{  
 uint32\_t resolution;  
} led\_strip\_encoder\_config\_t;  
  
// 编码器结构体  
typedef struct  
{  
 rmt\_encoder\_t base;  
 rmt\_encoder\_t \*bytes\_encoder;  
 rmt\_encoder\_t \*copy\_encoder;  
 int state;  
 rmt\_symbol\_word\_t reset\_code;  
} rmt\_led\_strip\_encoder\_t;

初始化几个全局变量。

// 日志前缀  
const char \*TAG = "LED ENCODER";  
  
// 每个灯需要发送RGB三个颜色，所以一共发送的数量要乘以3。  
// 这个数组用来存放待发送的RGB像素值。  
uint8\_t led\_strip\_pixels[LED\_NUMBERS \* 3];  
  
// 颜色的初始值。  
uint32\_t red = 0;  
uint32\_t green = 0;  
uint32\_t blue = 0;  
uint16\_t hue = 0;  
  
// LED通道  
rmt\_channel\_handle\_t led\_chan = NULL;  
// LED编码器  
rmt\_encoder\_handle\_t led\_encoder = NULL;  
// 发送配置，不进行循环发送  
rmt\_transmit\_config\_t tx\_config = {  
 .loop\_count = 0,  
};

然后编写RMT编码LED条带的代码。这部分代码来自示例程序。无需进行任何更改。

size\_t rmt\_encode\_led\_strip(rmt\_encoder\_t \*encoder, rmt\_channel\_handle\_t channel, const void \*primary\_data, size\_t data\_size, rmt\_encode\_state\_t \*ret\_state)  
{  
 rmt\_led\_strip\_encoder\_t \*led\_encoder = \_\_containerof(encoder, rmt\_led\_strip\_encoder\_t, base);  
 rmt\_encoder\_handle\_t bytes\_encoder = led\_encoder->bytes\_encoder;  
 rmt\_encoder\_handle\_t copy\_encoder = led\_encoder->copy\_encoder;  
 rmt\_encode\_state\_t session\_state = RMT\_ENCODING\_RESET;  
 rmt\_encode\_state\_t state = RMT\_ENCODING\_RESET;  
 size\_t encoded\_symbols = 0;  
 switch (led\_encoder->state)  
 {  
 case 0: // send RGB data  
 encoded\_symbols += bytes\_encoder->encode(bytes\_encoder, channel, primary\_data, data\_size, &session\_state);  
 if (session\_state & RMT\_ENCODING\_COMPLETE)  
 {  
 led\_encoder->state = 1; // switch to next state when current encoding session finished  
 }  
 if (session\_state & RMT\_ENCODING\_MEM\_FULL)  
 {  
 state |= RMT\_ENCODING\_MEM\_FULL;  
 goto out; // yield if there's no free space for encoding artifacts  
 }  
 // fall-through  
 case 1: // send reset code  
 encoded\_symbols += copy\_encoder->encode(copy\_encoder, channel, &led\_encoder->reset\_code,  
 sizeof(led\_encoder->reset\_code), &session\_state);  
 if (session\_state & RMT\_ENCODING\_COMPLETE)  
 {  
 led\_encoder->state = RMT\_ENCODING\_RESET; // back to the initial encoding session  
 state |= RMT\_ENCODING\_COMPLETE;  
 }  
 if (session\_state & RMT\_ENCODING\_MEM\_FULL)  
 {  
 state |= RMT\_ENCODING\_MEM\_FULL;  
 goto out; // yield if there's no free space for encoding artifacts  
 }  
 }  
out:  
 \*ret\_state = state;  
 return encoded\_symbols;  
}

删除编码器的代码。

esp\_err\_t rmt\_del\_led\_strip\_encoder(rmt\_encoder\_t \*encoder)  
{  
 rmt\_led\_strip\_encoder\_t \*led\_encoder = \_\_containerof(encoder, rmt\_led\_strip\_encoder\_t, base);  
 rmt\_del\_encoder(led\_encoder->bytes\_encoder);  
 rmt\_del\_encoder(led\_encoder->copy\_encoder);  
 free(led\_encoder);  
 return ESP\_OK;  
}

重置编码器的代码。

esp\_err\_t rmt\_led\_strip\_encoder\_reset(rmt\_encoder\_t \*encoder)  
{  
 rmt\_led\_strip\_encoder\_t \*led\_encoder = \_\_containerof(encoder, rmt\_led\_strip\_encoder\_t, base);  
 rmt\_encoder\_reset(led\_encoder->bytes\_encoder);  
 rmt\_encoder\_reset(led\_encoder->copy\_encoder);  
 led\_encoder->state = RMT\_ENCODING\_RESET;  
 return ESP\_OK;  
}

新建编码器的函数。

esp\_err\_t rmt\_new\_led\_strip\_encoder(const led\_strip\_encoder\_config\_t \*config, rmt\_encoder\_handle\_t \*ret\_encoder)  
{  
 esp\_err\_t ret = ESP\_OK;  
 rmt\_led\_strip\_encoder\_t \*led\_encoder = NULL;  
 ESP\_GOTO\_ON\_FALSE(config && ret\_encoder, ESP\_ERR\_INVALID\_ARG, err, TAG, "invalid argument");  
 led\_encoder = calloc(1, sizeof(rmt\_led\_strip\_encoder\_t));  
 ESP\_GOTO\_ON\_FALSE(led\_encoder, ESP\_ERR\_NO\_MEM, err, TAG, "no mem for led strip encoder");  
 led\_encoder->base.encode = rmt\_encode\_led\_strip;  
 led\_encoder->base.del = rmt\_del\_led\_strip\_encoder;  
 led\_encoder->base.reset = rmt\_led\_strip\_encoder\_reset;  
 // different led strip might have its own timing requirements, following parameter is for WS2812  
 rmt\_bytes\_encoder\_config\_t bytes\_encoder\_config = {  
 .bit0 = {  
 .level0 = 1,  
 .duration0 = 0.3 \* config->resolution / 1000000, // T0H=0.3us  
 .level1 = 0,  
 .duration1 = 0.9 \* config->resolution / 1000000, // T0L=0.9us  
 },  
 .bit1 = {  
 .level0 = 1,  
 .duration0 = 0.9 \* config->resolution / 1000000, // T1H=0.9us  
 .level1 = 0,  
 .duration1 = 0.3 \* config->resolution / 1000000, // T1L=0.3us  
 },  
 .flags.msb\_first = 1 // WS2812 transfer bit order: G7...G0R7...R0B7...B0  
 };  
 ESP\_GOTO\_ON\_ERROR(rmt\_new\_bytes\_encoder(&bytes\_encoder\_config, &led\_encoder->bytes\_encoder), err, TAG, "create bytes encoder failed");  
 rmt\_copy\_encoder\_config\_t copy\_encoder\_config = {};  
 ESP\_GOTO\_ON\_ERROR(rmt\_new\_copy\_encoder(&copy\_encoder\_config, &led\_encoder->copy\_encoder), err, TAG, "create copy encoder failed");  
  
 uint32\_t reset\_ticks = config->resolution / 1000000 \* 50 / 2; // reset code duration defaults to 50us  
 led\_encoder->reset\_code = (rmt\_symbol\_word\_t){  
 .level0 = 0,  
 .duration0 = reset\_ticks,  
 .level1 = 0,  
 .duration1 = reset\_ticks,  
 };  
 \*ret\_encoder = &led\_encoder->base;  
 return ESP\_OK;  
err:  
 if (led\_encoder)  
 {  
 if (led\_encoder->bytes\_encoder)  
 {  
 rmt\_del\_encoder(led\_encoder->bytes\_encoder);  
 }  
 if (led\_encoder->copy\_encoder)  
 {  
 rmt\_del\_encoder(led\_encoder->copy\_encoder);  
 }  
 free(led\_encoder);  
 }  
 return ret;  
}

然后编写将 HSV 颜色转换成 RGB 颜色的代码。

void led\_strip\_hsv2rgb(uint32\_t h, uint32\_t s, uint32\_t v, uint32\_t \*r, uint32\_t \*g, uint32\_t \*b)  
{  
 h %= 360; // h -> [0,360]  
 uint32\_t rgb\_max = v \* 2.55f;  
 uint32\_t rgb\_min = rgb\_max \* (100 - s) / 100.0f;  
  
 uint32\_t i = h / 60;  
 uint32\_t diff = h % 60;  
  
 // RGB adjustment amount by hue  
 uint32\_t rgb\_adj = (rgb\_max - rgb\_min) \* diff / 60;  
  
 switch (i)  
 {  
 case 0:  
 \*r = rgb\_max;  
 \*g = rgb\_min + rgb\_adj;  
 \*b = rgb\_min;  
 break;  
 case 1:  
 \*r = rgb\_max - rgb\_adj;  
 \*g = rgb\_max;  
 \*b = rgb\_min;  
 break;  
 case 2:  
 \*r = rgb\_min;  
 \*g = rgb\_max;  
 \*b = rgb\_min + rgb\_adj;  
 break;  
 case 3:  
 \*r = rgb\_min;  
 \*g = rgb\_max - rgb\_adj;  
 \*b = rgb\_max;  
 break;  
 case 4:  
 \*r = rgb\_min + rgb\_adj;  
 \*g = rgb\_min;  
 \*b = rgb\_max;  
 break;  
 default:  
 \*r = rgb\_max;  
 \*g = rgb\_min;  
 \*b = rgb\_max - rgb\_adj;  
 break;  
 }  
}

然后编写 RMT 初始化的代码。

void RMT\_Init(void)  
{  
 rmt\_tx\_channel\_config\_t tx\_chan\_config = {  
 .clk\_src = RMT\_CLK\_SRC\_DEFAULT, // 选择时钟源  
 .gpio\_num = RMT\_LED\_STRIP\_GPIO\_NUM, // GPIO引脚设置  
 .mem\_block\_symbols = 64, // increase the block size can make the LED less flickering  
 .resolution\_hz = RMT\_LED\_STRIP\_RESOLUTION\_HZ,  
 .trans\_queue\_depth = 4, // set the number of transactions that can be pending in the background  
 };  
 ESP\_ERROR\_CHECK(rmt\_new\_tx\_channel(&tx\_chan\_config, &led\_chan));  
  
 led\_strip\_encoder\_config\_t encoder\_config = {  
 .resolution = RMT\_LED\_STRIP\_RESOLUTION\_HZ,  
 };  
 ESP\_ERROR\_CHECK(rmt\_new\_led\_strip\_encoder(&encoder\_config, &led\_encoder));  
  
 ESP\_ERROR\_CHECK(rmt\_enable(led\_chan));  
}

以上代码基本都是来自示例代码。接下来我们自己写点亮某一个灯的代码。

void light\_led(uint8\_t led\_num)  
{  
 for (int i = 0; i < 3; i++)  
 {  
 // 构建 RGB 像素值。  
 hue = led\_num \* 360 / LED\_NUMBERS;  
 led\_strip\_hsv2rgb(hue, 30, 30, &red, &green, &blue);  
 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 像素值发送到 LED 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));  
  
 vTaskDelay(100 / portTICK\_PERIOD\_MS);  
  
 // 将像素数组置为0。  
 memset(led\_strip\_pixels, 0, sizeof(led\_strip\_pixels));  
  
 // 再次发送RGB像素值，熄灭LED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));  
}

然后编写入口函数。

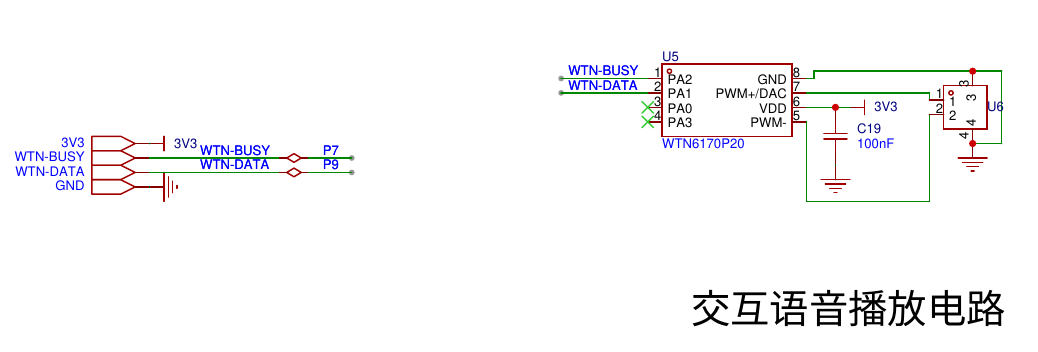
void Delay\_ms(uint32\_t time)  
{  
 vTaskDelay(time / portTICK\_PERIOD\_MS);  
}  
  
void app\_main(void)  
{  
 RMT\_Init();  
 for (int i = 0; i < LED\_NUMBERS; i++)  
 {  
 light\_led(i);  
 Delay\_ms(1000);  
 }  
}

然后编译并烧写程序。

cd ~/esp/atguigu-led-example  
idf.py set-target esp32c3  
idf.py flash

# 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 。



我们直接给出主程序代码，大家可以练习一下如何构建项目并编译烧写。参考前两个例子即可。

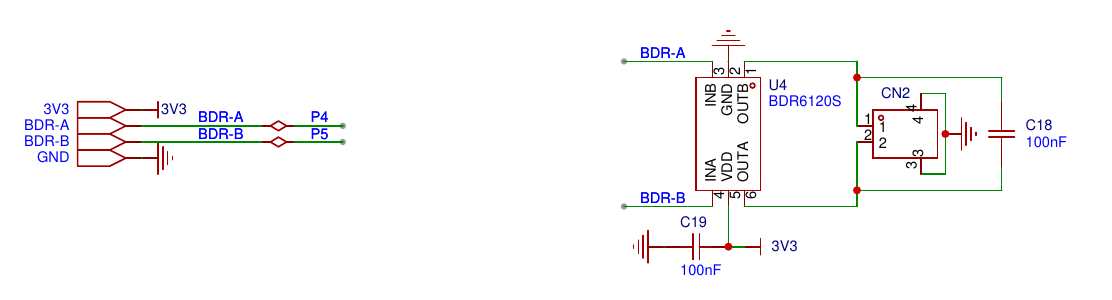
#include "driver/gpio.h"  
#include "sys/unistd.h"  
#include "esp\_log.h"  
#include "freertos/FreeRTOS.h"  
#include "freertos/task.h"  
  
#define DELAY\_US(i) usleep(i) // usleep(i)  
  
#define DELAY\_MS(i) vTaskDelay(i / portTICK\_PERIOD\_MS)  
  
#define AUDIO\_SDA\_PIN GPIO\_NUM\_9  
#define AUDIO\_BUSY\_PIN GPIO\_NUM\_7  
  
#define AUDIO\_SDA\_H gpio\_set\_level(AUDIO\_SDA\_PIN, 1)  
#define AUDIO\_SDA\_L gpio\_set\_level(AUDIO\_SDA\_PIN, 0)  
  
#define AUDIO\_READ\_BUSY gpio\_get\_level(AUDIO\_BUSY\_PIN)  
  
void Line\_1A\_WT588F(uint8\_t DDATA)  
{  
 uint8\_t S\_DATA, j;  
 uint8\_t B\_DATA;  
 S\_DATA = DDATA;  
 AUDIO\_SDA\_L;  
 // 重要延时  
 DELAY\_MS(10);  
 B\_DATA = S\_DATA & 0X01;  
 for (j = 0; j < 8; j++)  
 {  
 if (B\_DATA == 1)  
 {  
 AUDIO\_SDA\_H;  
 DELAY\_US(600);  
 AUDIO\_SDA\_L;  
 DELAY\_US(200);  
 }  
 else  
 {  
 AUDIO\_SDA\_H;  
 DELAY\_US(200);  
 AUDIO\_SDA\_L;  
 DELAY\_US(600);  
 }  
 S\_DATA = S\_DATA >> 1;  
 B\_DATA = S\_DATA & 0X01;  
 }  
 AUDIO\_SDA\_H;  
 DELAY\_MS(2);  
}  
  
void AUDIO\_Test(void)  
{  
 Line\_1A\_WT588F(85);  
 DELAY\_MS(2000);  
 Line\_1A\_WT588F(86);  
 DELAY\_MS(2000);  
 Line\_1A\_WT588F(69);  
 DELAY\_MS(2000);  
}  
  
void AUDIO\_Init(void)  
{  
 gpio\_config\_t io\_conf = {};  
 io\_conf.intr\_type = GPIO\_INTR\_DISABLE;  
 io\_conf.mode = GPIO\_MODE\_OUTPUT;  
 io\_conf.pin\_bit\_mask = (1ULL << AUDIO\_SDA\_PIN);  
 gpio\_config(&io\_conf);  
  
 io\_conf.intr\_type = GPIO\_INTR\_DISABLE;  
 io\_conf.mode = GPIO\_MODE\_INPUT;  
 io\_conf.pin\_bit\_mask = (1ULL << AUDIO\_BUSY\_PIN);  
 gpio\_config(&io\_conf);  
}  
  
void app\_main(void)  
{  
 AUDIO\_Init();  
 AUDIO\_Test();  
}

# 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：



由于驱动电机只涉及到GPIO的基本操作，所以这里我们只给出两个关键代码。需要大家补全剩下的代码。并驱动电机转起来。

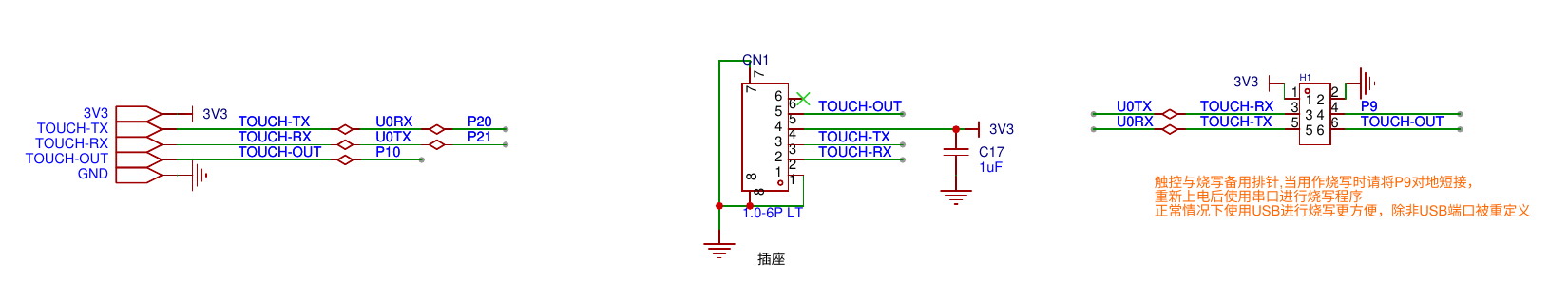
#define MOTOR\_DRIVER\_NUM\_0 GPIO\_NUM\_4  
#define MOTOR\_DRIVER\_NUM\_1 GPIO\_NUM\_5  
  
void MOTOR\_Init(void)  
{  
 gpio\_config\_t io\_conf;  
 io\_conf.intr\_type = GPIO\_INTR\_DISABLE;  
 io\_conf.mode = GPIO\_MODE\_OUTPUT;  
 io\_conf.pin\_bit\_mask = ((1ULL << MOTOR\_DRIVER\_NUM\_0) | (1ULL << MOTOR\_DRIVER\_NUM\_1));  
 gpio\_config(&io\_conf);  
  
 gpio\_set\_level(MOTOR\_DRIVER\_NUM\_0, 0);  
 gpio\_set\_level(MOTOR\_DRIVER\_NUM\_1, 0);  
}

开锁代码

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);  
}

# 串口通信

ESP32使用串口和指纹模块进行通信。电路图如下：



## 串口收发简单示例

我们先来写一个简单的串口示例程序。

#include "freertos/FreeRTOS.h"  
#include "freertos/task.h"  
#include "esp\_system.h"  
#include "esp\_log.h"  
#include "driver/uart.h"  
#include "string.h"  
#include "driver/gpio.h"  
  
static const int RX\_BUF\_SIZE = 1024;  
  
#define TXD\_PIN (GPIO\_NUM\_21)  
#define RXD\_PIN (GPIO\_NUM\_20)  
  
void init(void)  
{  
 const uart\_config\_t uart\_config = {  
 .baud\_rate = 115200,  
 .data\_bits = UART\_DATA\_8\_BITS,  
 .parity = UART\_PARITY\_DISABLE,  
 .stop\_bits = UART\_STOP\_BITS\_1,  
 .flow\_ctrl = UART\_HW\_FLOWCTRL\_DISABLE,  
 .source\_clk = UART\_SCLK\_DEFAULT,  
 };  
 // We won't use a buffer for sending data.  
 uart\_driver\_install(UART\_NUM\_1, RX\_BUF\_SIZE \* 2, 0, 0, NULL, 0);  
 uart\_param\_config(UART\_NUM\_1, &uart\_config);  
 uart\_set\_pin(UART\_NUM\_1, TXD\_PIN, RXD\_PIN, UART\_PIN\_NO\_CHANGE, UART\_PIN\_NO\_CHANGE);  
}  
  
int sendData(const char \*logName, const char \*data)  
{  
 const int len = strlen(data);  
 const int txBytes = uart\_write\_bytes(UART\_NUM\_1, data, len);  
 ESP\_LOGI(logName, "Wrote %d bytes", txBytes);  
 return txBytes;  
}  
  
static void tx\_task(void \*arg)  
{  
 static const char \*TX\_TASK\_TAG = "TX\_TASK";  
 esp\_log\_level\_set(TX\_TASK\_TAG, ESP\_LOG\_INFO);  
 while (1)  
 {  
 sendData(TX\_TASK\_TAG, "Hello world");  
 vTaskDelay(2000 / portTICK\_PERIOD\_MS);  
 }  
}  
  
static void rx\_task(void \*arg)  
{  
 static const char \*RX\_TASK\_TAG = "RX\_TASK";  
 esp\_log\_level\_set(RX\_TASK\_TAG, ESP\_LOG\_INFO);  
 uint8\_t \*data = (uint8\_t \*)malloc(RX\_BUF\_SIZE + 1);  
 while (1)  
 {  
 const int rxBytes = uart\_read\_bytes(UART\_NUM\_1, data, RX\_BUF\_SIZE, 1000 / portTICK\_PERIOD\_MS);  
 if (rxBytes > 0)  
 {  
 data[rxBytes] = 0;  
 ESP\_LOGI(RX\_TASK\_TAG, "Read %d bytes: '%s'", rxBytes, data);  
 ESP\_LOG\_BUFFER\_HEXDUMP(RX\_TASK\_TAG, data, rxBytes, ESP\_LOG\_INFO);  
 }  
 }  
 free(data);  
}  
  
void app\_main(void)  
{  
 init();  
 xTaskCreate(rx\_task, "uart\_rx\_task", 1024 \* 2, NULL, configMAX\_PRIORITIES, NULL);  
 xTaskCreate(tx\_task, "uart\_tx\_task", 1024 \* 2, NULL, configMAX\_PRIORITIES - 1, NULL);  
}

有了以上的基础，我们就可以来实现指纹模块的驱动程序了。

## 指纹模块

我们先来写头文件

#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 TXD\_PIN (GPIO\_NUM\_21)  
#define RXD\_PIN (GPIO\_NUM\_20)  
  
#define BUF\_SIZE (1024)  
  
#define TOUCH\_INT GPIO\_NUM\_10  
  
/// 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);  
  
/// 读取指纹芯片配置参数  
void read\_sys\_params(void);  
  
#endif

然后编写头文件中接口的实现

#include "fingerprint\_driver.h"  
  
void FINGERPRINT\_Init(void)  
{  
 /\* Configure parameters of an UART driver,  
 \* communication pins and install the driver \*/  
 uart\_config\_t uart\_config = {  
 .baud\_rate = 57600,  
 .data\_bits = UART\_DATA\_8\_BITS,  
 .parity = UART\_PARITY\_DISABLE,  
 .stop\_bits = UART\_STOP\_BITS\_1,  
 .flow\_ctrl = UART\_HW\_FLOWCTRL\_DISABLE,  
 .source\_clk = UART\_SCLK\_DEFAULT,  
 };  
 int intr\_alloc\_flags = 0;  
  
 ESP\_ERROR\_CHECK(uart\_driver\_install(  
 UART\_NUM\_1,  
 BUF\_SIZE \* 2, 0, 0, NULL,  
 intr\_alloc\_flags));  
 ESP\_ERROR\_CHECK(uart\_param\_config(  
 UART\_NUM\_1, &uart\_config));  
 ESP\_ERROR\_CHECK(uart\_set\_pin(  
 UART\_NUM\_1,  
 TXD\_PIN,  
 RXD\_PIN,  
 UART\_PIN\_NO\_CHANGE,  
 UART\_PIN\_NO\_CHANGE));  
  
 // 中断  
 gpio\_config\_t io\_conf;  
 io\_conf.intr\_type = GPIO\_INTR\_NEGEDGE;  
 io\_conf.mode = GPIO\_MODE\_INPUT;  
 io\_conf.pin\_bit\_mask = (1ULL << TOUCH\_INT);  
 io\_conf.pull\_up\_en = 1;  
 gpio\_config(&io\_conf);  
  
 printf("指纹模块初始化成功。\r\n");  
}  
  
void get\_chip\_sn(void)  
{  
 vTaskDelay(200 / portTICK\_PERIOD\_MS);  
 uint8\_t \*data = (uint8\_t \*)malloc(BUF\_SIZE);  
  
 // 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。  
 uint8\_t PS\_GetChipSN[13] = {0xEF, 0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0x01, 0x00, 0x04, 0x34, 0x00, 0x00, 0x39};  
 uart\_write\_bytes(UART\_NUM\_1, (const char \*)PS\_GetChipSN, 13);  
  
 // Read data from the UART  
 int len = uart\_read\_bytes(UART\_NUM\_1, data, (BUF\_SIZE - 1), 2000 / portTICK\_PERIOD\_MS);  
  
 if (len)  
 {  
 if (data[6] == 0x07 && data[9] == 0x00)  
 {  
 printf("chip sn: %.32s\r\n", &data[10]);  
 }  
 }  
  
 free(data);  
}  
  
// 检测是否有手指放在模组上面  
int get\_image(void)  
{  
 uint8\_t \*data = (uint8\_t \*)malloc(BUF\_SIZE);  
  
 // 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。  
 uint8\_t PS\_GetImageBuffer[12] = {0xEF, 0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0x01, 0x00, 0x03, 0x01, 0x00, 0x05};  
  
 uart\_write\_bytes(UART\_NUM\_1, (const char \*)PS\_GetImageBuffer, 12);  
  
 int len = uart\_read\_bytes(UART\_NUM\_1, data, (BUF\_SIZE - 1), 2000 / portTICK\_PERIOD\_MS);  
  
 int result = 0xFF;  
  
 if (len)  
 {  
 if (data[6] == 0x07)  
 {  
 if (data[9] == 0x00)  
 {  
 result = 0;  
 }  
 else if (data[9] == 0x01)  
 {  
 result = 1;  
 }  
 else if (data[9] == 0x02)  
 {  
 result = 2;  
 }  
 }  
 }  
  
 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(UART\_NUM\_1, (const char \*)PS\_GenCharBuffer, 13);  
  
 int len = uart\_read\_bytes(UART\_NUM\_1, 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(UART\_NUM\_1, (const char \*)PS\_SearchBuffer, 17);  
  
 int len = uart\_read\_bytes(UART\_NUM\_1, 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(UART\_NUM\_1, (const char \*)PS\_ReadSysPara, 12);  
  
 int len = uart\_read\_bytes(UART\_NUM\_1, 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)  
 {  
 printf("packet size ==> 64 bytes\r\n");  
 }  
 else if (packet\_size == 2)  
 {  
 printf("packet size ==> 128 bytes\r\n");  
 }  
 else if (packet\_size == 3)  
 {  
 printf("packet size ==> 256 bytes\r\n");  
 }  
 // baud rate  
 int baud\_rate = (data[24] << 8) | data[25];  
 printf("baud rate ==> %d\r\n", 9600 \* baud\_rate);  
 }  
 else if (data[9] == 0x01)  
 {  
 printf("send packet error\r\n");  
 }  
 }  
 }  
  
 free(data);  
}

# 蓝牙模块

蓝牙模块的参考代码位于examples/bluetooth/bluedroid/ble/gatt\_server文件夹。

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。

## 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。

### 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。

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。

### 入口函数

入口函数是 app\_main() 函数。

void app\_main(void)  
{  
 esp\_err\_t ret;  
  
 // Initialize NVS.  
 ret = nvs\_flash\_init();  
 if (ret == ESP\_ERR\_NVS\_NO\_FREE\_PAGES || ret == ESP\_ERR\_NVS\_NEW\_VERSION\_FOUND)  
 {  
 ESP\_ERROR\_CHECK(nvs\_flash\_erase());  
 ret = nvs\_flash\_init();  
 }  
 ESP\_ERROR\_CHECK(ret);  
  
 ESP\_ERROR\_CHECK(esp\_bt\_controller\_mem\_release(ESP\_BT\_MODE\_CLASSIC\_BT));  
  
 esp\_bt\_controller\_config\_t bt\_cfg = BT\_CONTROLLER\_INIT\_CONFIG\_DEFAULT();  
 ret = esp\_bt\_controller\_init(&bt\_cfg);  
 if (ret)  
 {  
 ESP\_LOGE(GATTS\_TAG, "%s initialize controller failed: %s\n", \_\_func\_\_, esp\_err\_to\_name(ret));  
 return;  
 }  
  
 ret = esp\_bt\_controller\_enable(ESP\_BT\_MODE\_BLE);  
 if (ret)  
 {  
 ESP\_LOGE(GATTS\_TAG, "%s enable controller failed: %s\n", \_\_func\_\_, esp\_err\_to\_name(ret));  
 return;  
 }  
 ret = esp\_bluedroid\_init();  
 if (ret)  
 {  
 ESP\_LOGE(GATTS\_TAG, "%s init bluetooth failed: %s\n", \_\_func\_\_, esp\_err\_to\_name(ret));  
 return;  
 }  
 ret = esp\_bluedroid\_enable();  
 if (ret)  
 {  
 ESP\_LOGE(GATTS\_TAG, "%s enable bluetooth failed: %s\n", \_\_func\_\_, esp\_err\_to\_name(ret));  
 return;  
 }  
  
 ret = esp\_ble\_gatts\_register\_callback(gatts\_event\_handler);  
 if (ret)  
 {  
 ESP\_LOGE(GATTS\_TAG, "gatts register error, error code = %x", ret);  
 return;  
 }  
 ret = esp\_ble\_gap\_register\_callback(gap\_event\_handler);  
 if (ret)  
 {  
 ESP\_LOGE(GATTS\_TAG, "gap register error, error code = %x", ret);  
 return;  
 }  
 ret = esp\_ble\_gatts\_app\_register(PROFILE\_A\_APP\_ID);  
 if (ret)  
 {  
 ESP\_LOGE(GATTS\_TAG, "gatts app register error, error code = %x", ret);  
 return;  
 }  
 ret = esp\_ble\_gatts\_app\_register(PROFILE\_B\_APP\_ID);  
 if (ret)  
 {  
 ESP\_LOGE(GATTS\_TAG, "gatts app register error, error code = %x", ret);  
 return;  
 }  
 esp\_err\_t local\_mtu\_ret = esp\_ble\_gatt\_set\_local\_mtu(500);  
 if (local\_mtu\_ret)  
 {  
 ESP\_LOGE(GATTS\_TAG, "set local MTU failed, error code = %x", local\_mtu\_ret);  
 }  
  
 return;  
}

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：

ret = nvs\_flash\_init();

### 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：

esp\_bt\_controller\_config\_t bt\_cfg = BT\_CONTROLLER\_INIT\_CONFIG\_DEFAULT();  
ret = esp\_bt\_controller\_init(&bt\_cfg);

接下来，控制器使能为 BLE 模式。

esp\_bt\_controller\_enable(ESP\_BT\_MODE\_BLE);

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：

* ESP\_BT\_MODE\_IDLE：蓝牙未运行
* ESP\_BT\_MODE\_BLE：BLE模式
* ESP\_BT\_MODE\_CLASSIC\_BT：经典蓝牙模式
* 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：

esp\_bluedroid\_init();  
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：

esp\_ble\_gatts\_register\_callback(gatts\_event\_handler);  
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。

在蓝牙协议栈中，GAP（Generic Access Profile）和GATT（Generic Attribute Profile）是两个非常重要的概念，它们各自承担着不同的职责。

GAP（Generic Access Profile）

GAP是蓝牙技术中的通用接入配置文件，它定义了蓝牙设备如何发现其他蓝牙设备以及如何建立连接和安全性的基本要求。简单来说，GAP负责蓝牙设备的连接模式和过程。它包括设备的广播、探索、连接和配对过程。GAP确保了不同厂商生产的蓝牙设备能够相互识别和连接。

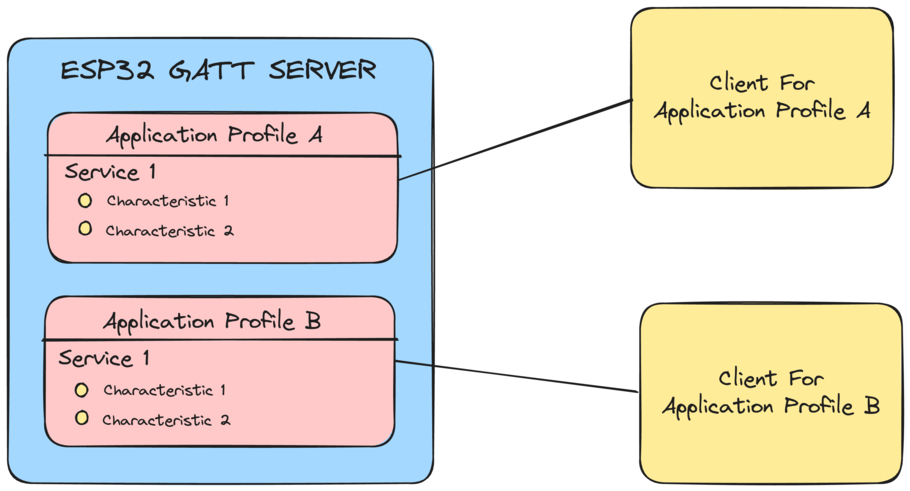
GATT（Generic Attribute Profile）

GATT是基于BLE（Bluetooth Low Energy，蓝牙低能耗）技术的一种协议规范，它定义了通过BLE连接进行数据交换的方式。GATT使用一个基于属性的数据模型，这些属性可以是数据、配置或者其他类型的信息，如设备名称或可测量的数据等。GATT协议规定了如何对这些属性进行格式化和传输，从而使得设备间能够交换具有结构的数据。GATT构建在ATT（Attribute Protocol）之上，主要用于定义设备如何使用一个通用的数据结构来交互。

简而言之，GAP负责定义和管理设备的连接，而GATT则负责定义设备间如何交换数据。这两者共同工作，使得蓝牙设备不仅能够连接，还能够有效地通信和交换数据。

### 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APP设计的，例如在智能手机或平板电脑上运行的APP。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。



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：

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。

static struct gatts\_profile\_inst gl\_profile\_tab[PROFILE\_NUM] = {  
 [PROFILE\_A\_APP\_ID] = {  
 .gatts\_cb = gatts\_profile\_a\_event\_handler,  
 .gatts\_if = ESP\_GATT\_IF\_NONE,  
 [PROFILE\_B\_APP\_ID] = {  
 .gatts\_cb = gatts\_profile\_b\_event\_handler,  
 .gatts\_if = 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。

esp\_ble\_gatts\_app\_register(PROFILE\_A\_APP\_ID);  
esp\_ble\_gatts\_app\_register(PROFILE\_B\_APP\_ID);

### 设置GAP参数

注册应用程序事件是程序生命周期中触发的第一个事件，此示例使用Profile A GATT事件句柄在注册时配置广告参数。

此示例提供了使用标准蓝牙核心规范广告参数或自定义原始缓冲区的选项。

可以通过CONFIG\_SET\_RAW\_ADV\_DATA定义来选择此选项。

原始广告数据可用于实现iBeacon、Eddystone或其他专有和自定义帧类型，如用于室内定位服务的帧类型，这些帧类型与标准规范不同。

用于配置标准蓝牙规范广告参数的函数是esp\_ble\_gap\_config\_adv\_data()。

它接受一个指向esp\_ble\_adv\_data\_t结构的指针。

广告数据的esp\_ble\_adv\_data\_t数据结构定义如下：

typedef struct {  
 bool set\_scan\_rsp; /\*!< Set this advertising data as scan response or not\*/  
 bool include\_name; /\*!< Advertising data include device name or not \*/  
 bool include\_txpower; /\*!< Advertising data include TX power \*/  
 int min\_interval; /\*!< Advertising data show slave preferred connection min interval \*/  
 int max\_interval; /\*!< Advertising data show slave preferred connection max interval \*/  
 int appearance; /\*!< External appearance of device \*/  
 uint16\_t manufacturer\_len; /\*!< Manufacturer data length \*/  
 uint8\_t \*p\_manufacturer\_data; /\*!< Manufacturer data point \*/  
 uint16\_t service\_data\_len; /\*!< Service data length \*/  
 uint8\_t \*p\_service\_data; /\*!< Service data point \*/  
 uint16\_t service\_uuid\_len; /\*!< Service uuid length \*/  
 uint8\_t \*p\_service\_uuid; /\*!< Service uuid array point \*/  
 uint8\_t flag; /\*!< Advertising flag of discovery mode, see BLE\_ADV\_DATA\_FLAG detail \*/  
} esp\_ble\_adv\_data\_t;

在这个例子里面，初始化以上结构体如下：

static esp\_ble\_adv\_data\_t adv\_data = {  
 .set\_scan\_rsp = false,  
 .include\_name = true,  
 .include\_txpower = true,  
 .min\_interval = 0x0006,  
 .max\_interval = 0x0010,  
 .appearance = 0x00,  
 .manufacturer\_len = 0, //TEST\_MANUFACTURER\_DATA\_LEN,  
 .p\_manufacturer\_data = NULL, //&test\_manufacturer[0],  
 .service\_data\_len = 0,  
 .p\_service\_data = NULL,  
 .service\_uuid\_len = 32,  
 .p\_service\_uuid = test\_service\_uuid128,  
 .flag = (ESP\_BLE\_ADV\_FLAG\_GEN\_DISC | ESP\_BLE\_ADV\_FLAG\_BREDR\_NOT\_SPT),  
};

最小和最大从设备首选连接间隔以1.25毫秒为单位设置。

在此示例中，最小从设备首选连接间隔定义为0x0006 \* 1.25毫秒 = 7.5毫秒，最大从设备首选连接间隔初始化为0x0010 \* 1.25毫秒 = 20毫秒。

广告负载最多可以包含31字节的数据。如果参数数据足够大以至于超过31字节的广告包限制，可能会导致栈切割广告包并且留下一些参数不包括在内。如果取消注释制造商长度和数据，这种行为可以在此示例中展示，这将导致重新编译和测试后服务不被广告。

也可以使用esp\_ble\_gap\_config\_adv\_data\_raw()和esp\_ble\_gap\_config\_scan\_rsp\_data\_raw()函数来广告自定义原始数据，这需要为广告数据和扫描响应数据创建并传递一个缓冲区。在此示例中，原始数据由raw\_adv\_data[]和raw\_scan\_rsp\_data[]数组表示。

最后，使用esp\_ble\_gap\_set\_device\_name()函数来设置设备名称。注册事件处理程序如下所示：

static void gatts\_profile\_a\_event\_handler(esp\_gatts\_cb\_event\_t event, esp\_gatt\_if\_t gatts\_if, esp\_ble\_gatts\_cb\_param\_t \*param) {  
 switch (event) {  
 case ESP\_GATTS\_REG\_EVT:  
 ESP\_LOGI(GATTS\_TAG, "REGISTER\_APP\_EVT, status %d, app\_id %d\n", param->reg.status, param->reg.app\_id);  
 gl\_profile\_tab[PROFILE\_A\_APP\_ID].service\_id.is\_primary = true;  
 gl\_profile\_tab[PROFILE\_A\_APP\_ID].service\_id.id.inst\_id = 0x00;  
 gl\_profile\_tab[PROFILE\_A\_APP\_ID].service\_id.id.uuid.len = ESP\_UUID\_LEN\_16;  
 gl\_profile\_tab[PROFILE\_A\_APP\_ID].service\_id.id.uuid.uuid.uuid16 = GATTS\_SERVICE\_UUID\_TEST\_A;  
  
 esp\_ble\_gap\_set\_device\_name(TEST\_DEVICE\_NAME);  
#ifdef CONFIG\_SET\_RAW\_ADV\_DATA  
 esp\_err\_t raw\_adv\_ret = esp\_ble\_gap\_config\_adv\_data\_raw(raw\_adv\_data, sizeof(raw\_adv\_data));  
 if (raw\_adv\_ret){  
 ESP\_LOGE(GATTS\_TAG, "config raw adv data failed, error code = %x ", raw\_adv\_ret);  
 }  
 adv\_config\_done |= adv\_config\_flag;  
 esp\_err\_t raw\_scan\_ret = esp\_ble\_gap\_config\_scan\_rsp\_data\_raw(raw\_scan\_rsp\_data, sizeof(raw\_scan\_rsp\_data));  
 if (raw\_scan\_ret){  
 ESP\_LOGE(GATTS\_TAG, "config raw scan rsp data failed, error code = %x", raw\_scan\_ret);  
 }  
 adv\_config\_done |= scan\_rsp\_config\_flag;  
#else  
 //config adv data  
 esp\_err\_t ret = esp\_ble\_gap\_config\_adv\_data(&adv\_data);  
 if (ret){  
 ESP\_LOGE(GATTS\_TAG, "config adv data failed, error code = %x", ret);  
 }  
 adv\_config\_done |= adv\_config\_flag;  
 //config scan response data  
 ret = esp\_ble\_gap\_config\_adv\_data(&scan\_rsp\_data);  
 if (ret){  
 ESP\_LOGE(GATTS\_TAG, "config scan response data failed, error code = %x", ret);  
 }  
 adv\_config\_done |= scan\_rsp\_config\_flag;  
#endif

### GAP事件句柄

一旦广告数据被设置，GAP事件ESP\_GAP\_BLE\_ADV\_DATA\_SET\_COMPLETE\_EVT就会被触发。对于原始广告数据集的情况，触发的事件是ESP\_GAP\_BLE\_ADV\_DATA\_RAW\_SET\_COMPLETE\_EVT。另外，当原始扫描响应数据被设置时，ESP\_GAP\_BLE\_SCAN\_RSP\_DATA\_RAW\_SET\_COMPLETE\_EVT事件将被触发。

static void gap\_event\_handler(esp\_gap\_ble\_cb\_event\_t event, esp\_ble\_gap\_cb\_param\_t \*param)  
{  
 switch (event) {  
#ifdef CONFIG\_SET\_RAW\_ADV\_DATA  
 case ESP\_GAP\_BLE\_ADV\_DATA\_RAW\_SET\_COMPLETE\_EVT:  
 adv\_config\_done &= (~adv\_config\_flag);  
 if (adv\_config\_done==0){  
 esp\_ble\_gap\_start\_advertising(&adv\_params);  
 }  
 break;  
 case ESP\_GAP\_BLE\_SCAN\_RSP\_DATA\_RAW\_SET\_COMPLETE\_EVT:  
 adv\_config\_done &= (~scan\_rsp\_config\_flag);  
 if (adv\_config\_done==0){  
 esp\_ble\_gap\_start\_advertising(&adv\_params);  
 }  
 break;  
#else  
 case ESP\_GAP\_BLE\_ADV\_DATA\_SET\_COMPLETE\_EVT:  
 adv\_config\_done &= (~adv\_config\_flag);  
 if (adv\_config\_done == 0){  
 esp\_ble\_gap\_start\_advertising(&adv\_params);  
 }  
 break;  
 case ESP\_GAP\_BLE\_SCAN\_RSP\_DATA\_SET\_COMPLETE\_EVT:  
 adv\_config\_done &= (~scan\_rsp\_config\_flag);  
 if (adv\_config\_done == 0){  
 esp\_ble\_gap\_start\_advertising(&adv\_params);  
 }  
 break;  
#endif  
...

无论哪种情况，服务器都可以使用esp\_ble\_gap\_start\_advertising()函数开始广告，该函数需要一个类型为esp\_ble\_adv\_params\_t的结构体，其中包含了栈操作所需的广告参数：

/// Advertising parameters  
typedef struct {  
 uint16\_t adv\_int\_min;  
 /\*!< Minimum advertising interval for undirected and low duty cycle directed advertising.  
 Range: 0x0020 to 0x4000  
 Default: N = 0x0800 (1.28 second)  
 Time = N \* 0.625 msec  
 Time Range: 20 ms to 10.24 sec \*/  
 uint16\_t adv\_int\_max;  
 /\*!< Maximum advertising interval for undirected and low duty cycle directed advertising.  
 Range: 0x0020 to 0x4000  
 Default: N = 0x0800 (1.28 second)  
 Time = N \* 0.625 msec  
 Time Range: 20 ms to 10.24 sec \*/  
 esp\_ble\_adv\_type\_t adv\_type; /\*!< Advertising type \*/  
 esp\_ble\_addr\_type\_t own\_addr\_type; /\*!< Owner bluetooth device address type \*/  
 esp\_bd\_addr\_t peer\_addr; /\*!< Peer device bluetooth device address \*/  
 esp\_ble\_addr\_type\_t peer\_addr\_type; /\*!< Peer device bluetooth device address type \*/  
 esp\_ble\_adv\_channel\_t channel\_map; /\*!< Advertising channel map \*/  
 esp\_ble\_adv\_filter\_t adv\_filter\_policy; /\*!< Advertising filter policy \*/  
}  
esp\_ble\_adv\_params\_t;

请注意，esp\_ble\_gap\_config\_adv\_data()配置将要广告给客户端的数据，并且需要一个esp\_ble\_adv\_data\_t结构体，而esp\_ble\_gap\_start\_advertising()使服务器实际开始广告，并且需要一个esp\_ble\_adv\_params\_t结构体。广告数据是展示给客户端的信息，而广告参数是GAP执行所需的配置。

对于这个示例，广告参数初始化如下：

static esp\_ble\_adv\_params\_t test\_adv\_params = {  
 .adv\_int\_min = 0x20,  
 .adv\_int\_max = 0x40,  
 .adv\_type = ADV\_TYPE\_IND,  
 .own\_addr\_type = BLE\_ADDR\_TYPE\_PUBLIC,  
 //.peer\_addr =  
 //.peer\_addr\_type =  
 .channel\_map = ADV\_CHNL\_ALL,  
 .adv\_filter\_policy = ADV\_FILTER\_ALLOW\_SCAN\_ANY\_CON\_ANY,  
};

这些参数将广告间隔配置在40毫秒到80毫秒之间。广告类型是ADV\_IND，这是一种通用的、不针对特定中央设备的可连接类型。地址类型是公开的，使用所有频道，并允许来自任何中央设备的扫描和连接请求。

如果广告成功开始，将生成一个ESP\_GAP\_BLE\_ADV\_START\_COMPLETE\_EVT事件，在本示例中用于检查广告状态是否确实为正在广告。否则，将打印一条错误消息。

...  
 case ESP\_GAP\_BLE\_ADV\_START\_COMPLETE\_EVT:  
 //advertising start complete event to indicate advertising start successfully or failed  
 if (param->adv\_start\_cmpl.status != ESP\_BT\_STATUS\_SUCCESS) {  
 ESP\_LOGE(GATTS\_TAG, "Advertising start failed\n");  
 }  
 break;  
...

### GATT事件句柄

当一个应用注册时，会触发ESP\_GATTS\_REG\_EVT事件。ESP\_GATTS\_REG\_EVT的参数如下所示：

esp\_gatt\_status\_t status; /\*!< Operation status \*/`  
uint16\_t app\_id; /\*!< Application id which input in register API \*/`

除了前面的参数之外，该事件还包含了由BLE栈分配的GATT接口。该事件被gatts\_event\_handler()捕获，该处理程序用于在配置文件表中存储生成的接口，然后将事件转发到相应的配置文件事件处理程序。

static 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) {  
 gl\_profile\_tab[param->reg.app\_id].gatts\_if = gatts\_if;  
 } else {  
 ESP\_LOGI(GATTS\_TAG, "Reg app failed, app\_id %04x, status %d\n",  
 param->reg.app\_id,  
 param->reg.status);  
 return;  
 }  
 }  
  
/\* If the gatts\_if equal to profile A, call profile A cb handler,  
 \* so here call each profile's callback \*/  
 do {  
 int idx;  
 for (idx = 0; idx < PROFILE\_NUM; idx++) {  
 if (gatts\_if == ESP\_GATT\_IF\_NONE||gatts\_if == gl\_profile\_tab[idx].gatts\_if) {  
 if (gl\_profile\_tab[idx].gatts\_cb) {  
 gl\_profile\_tab[idx].gatts\_cb(event, gatts\_if, param);  
 }  
 }  
 }  
 } while (0);  
}

### 创建服务

注册事件还用于使用esp\_ble\_gatts\_create\_service()创建服务。当服务创建完成时，会调用回调事件ESP\_GATTS\_CREATE\_EVT来向配置文件报告状态和服务ID。创建服务的方式是：

...  
esp\_ble\_gatts\_create\_service(gatts\_if, &gl\_profile\_tab[PROFILE\_A\_APP\_ID].service\_id, GATTS\_NUM\_HANDLE\_TEST\_A);  
break;  
...

句柄的数量定义为4：

#define GATTS\_NUM\_HANDLE\_TEST\_A 4

句柄包括：

* 服务句柄
* 特征句柄
* 特征值句柄
* 特征描述符句柄

服务被定义为一个具有16位UUID长度的主服务。

服务ID使用实例ID = 0和由GATTS\_SERVICE\_UUID\_TEST\_A定义的UUID进行初始化。

服务实例ID可以用来区分具有相同UUID的多个服务。在这个示例中，由于每个应用程序配置文件只有一个服务，并且服务具有不同的UUID，所以在配置文件A和B中服务实例ID都可以定义为0。然而，如果只有一个应用程序配置文件，且两个服务使用相同的UUID，则需要使用不同的实例ID来区分这两个服务。

应用程序配置文件B以与配置文件A相同的方式创建服务：

static void gatts\_profile\_b\_event\_handler(esp\_gatts\_cb\_event\_t event, esp\_gatt\_if\_t gatts\_if, esp\_ble\_gatts\_cb\_param\_t \*param) {  
 switch (event) {  
 case ESP\_GATTS\_REG\_EVT:  
 ESP\_LOGI(GATTS\_TAG, "REGISTER\_APP\_EVT, status %d, app\_id %d\n", param->reg.status, param->reg.app\_id);  
 gl\_profile\_tab[PROFILE\_B\_APP\_ID].service\_id.is\_primary = true;  
 gl\_profile\_tab[PROFILE\_B\_APP\_ID].service\_id.id.inst\_id = 0x00;  
 gl\_profile\_tab[PROFILE\_B\_APP\_ID].service\_id.id.uuid.len = ESP\_UUID\_LEN\_16;  
 gl\_profile\_tab[PROFILE\_B\_APP\_ID].service\_id.id.uuid.uuid.uuid16 = GATTS\_SERVICE\_UUID\_TEST\_B;  
  
 esp\_ble\_gatts\_create\_service(gatts\_if, &gl\_profile\_tab[PROFILE\_B\_APP\_ID].service\_id, GATTS\_NUM\_HANDLE\_TEST\_B);  
 break;  
...  
}

### 启动服务并创建特征

当服务成功创建时，由配置文件GATT处理程序管理的ESP\_GATTS\_CREATE\_EVT事件将被触发，可以用来启动服务并向服务中添加特征。对于配置文件A的情况，服务的启动和特征的添加如下所示：

case ESP\_GATTS\_CREATE\_EVT:  
 ESP\_LOGI(GATTS\_TAG, "CREATE\_SERVICE\_EVT, status %d, service\_handle %d\n", param->create.status, param->create.service\_handle);  
 gl\_profile\_tab[PROFILE\_A\_APP\_ID].service\_handle = param->create.service\_handle;  
 gl\_profile\_tab[PROFILE\_A\_APP\_ID].char\_uuid.len = ESP\_UUID\_LEN\_16;  
 gl\_profile\_tab[PROFILE\_A\_APP\_ID].char\_uuid.uuid.uuid16 = GATTS\_CHAR\_UUID\_TEST\_A;   
  
 esp\_ble\_gatts\_start\_service(gl\_profile\_tab[PROFILE\_A\_APP\_ID].service\_handle);  
 a\_property = ESP\_GATT\_CHAR\_PROP\_BIT\_READ | ESP\_GATT\_CHAR\_PROP\_BIT\_WRITE | ESP\_GATT\_CHAR\_PROP\_BIT\_NOTIFY;  
 esp\_err\_t add\_char\_ret =   
 esp\_ble\_gatts\_add\_char(gl\_profile\_tab[PROFILE\_A\_APP\_ID].service\_handle,   
 &gl\_profile\_tab[PROFILE\_A\_APP\_ID].char\_uuid,   
 ESP\_GATT\_PERM\_READ | ESP\_GATT\_PERM\_WRITE,   
 a\_property,   
 &gatts\_demo\_char1\_val,   
 NULL);  
 if (add\_char\_ret){  
 ESP\_LOGE(GATTS\_TAG, "add char failed, error code =%x",add\_char\_ret);  
 }  
 break;

首先，由BLE协议栈生成的服务句柄存储在配置文件表中，稍后应用层将使用它来引用此服务。

然后，设置特征的UUID及其UUID长度。特征UUID的长度再次为16位。使用先前生成的服务句柄，通过esp\_ble\_gatts\_start\_service()函数启动服务。触发了ESP\_GATTS\_START\_EVT事件，该事件用于打印信息。通过esp\_ble\_gatts\_add\_char()函数结合特征的权限和属性将特征添加到服务中。在这个示例中，两个配置文件中的特征如下设置：

权限：

* ESP\_GATT\_PERM\_READ：允许读取特征值
* ESP\_GATT\_PERM\_WRITE：允许写入特征值

属性：

* ESP\_GATT\_CHAR\_PROP\_BIT\_READ：可以读取特征
* ESP\_GATT\_CHAR\_PROP\_BIT\_WRITE：可以写入特征
* ESP\_GATT\_CHAR\_PROP\_BIT\_NOTIFY：特征可以通知值变化

对于读和写同时拥有权限和属性可能看起来有些冗余。然而，属性的读写属性是显示给客户端的信息，以便让客户端知道服务器是否接受读写请求。从这个意义上说，属性作为提示，以便客户端正确访问服务器资源。另一方面，权限是授权给客户端的，允许其读取或写入该属性。例如，如果客户端尝试写入一个没有写权限的属性，即使设置了写属性，服务器也会拒绝该请求。

此外，此示例给特征赋予了一个初始值，由gatts\_demo\_char1\_val表示。初始值定义如下：

esp\_attr\_value\_t gatts\_demo\_char1\_val =  
{  
 .attr\_max\_len = GATTS\_DEMO\_CHAR\_VAL\_LEN\_MAX,  
 .attr\_len = sizeof(char1\_str),  
 .attr\_value = char1\_str,  
};

Where char1\_str is dummy data:

uint8\_t char1\_str[] = {0x11,0x22,0x33};

特征长度定义为如下：

#define GATTS\_DEMO\_CHAR\_VAL\_LEN\_MAX 0x40

特征的初始值必须是一个非空对象，并且特征的长度必须始终大于零，否则栈将返回错误。

最后，特征被配置为每次读取或写入特征时都需要手动发送响应，而不是让栈自动响应。这是通过设置esp\_ble\_gatts\_add\_char()函数的最后一个参数，代表属性响应控制参数，为ESP\_GATT\_RSP\_BY\_APP或NULL来配置的。

### 创建特征描述符

向服务添加特征会触发一个ESP\_GATTS\_ADD\_CHAR\_EVT事件，该事件返回栈为刚添加的特征生成的句柄。事件包括以下参数：

esp\_gatt\_status\_t status; /\*!< Operation status \*/  
uint16\_t attr\_handle; /\*!< Characteristic attribute handle \*/  
uint16\_t service\_handle; /\*!< Service attribute handle \*/  
esp\_bt\_uuid\_t char\_uuid; /\*!< Characteristic uuid \*/

事件返回的属性句柄存储在配置文件表中，同时也设置了特征描述符的长度和UUID。

使用esp\_ble\_gatts\_get\_attr\_value()函数读取特征的长度和值，然后出于信息目的打印它们。最后，使用esp\_ble\_gatts\_add\_char\_descr()函数添加特征描述符。使用的参数包括服务句柄、描述符UUID、写和读权限、一个初始值以及自动响应设置。特征描述符的初始值可以是一个NULL指针，自动响应参数也设置为NULL，这意味着需要响应的请求必须手动回复。

case ESP\_GATTS\_ADD\_CHAR\_EVT: {  
 uint16\_t length = 0;  
 const uint8\_t \*prf\_char;  
  
 ESP\_LOGI(GATTS\_TAG, "ADD\_CHAR\_EVT, status %d, attr\_handle %d, service\_handle %d\n",  
 param->add\_char.status, param->add\_char.attr\_handle, param->add\_char.service\_handle);   
 gl\_profile\_tab[PROFILE\_A\_APP\_ID].char\_handle = param->add\_char.attr\_handle;  
 gl\_profile\_tab[PROFILE\_A\_APP\_ID].descr\_uuid.len = ESP\_UUID\_LEN\_16;   
 gl\_profile\_tab[PROFILE\_A\_APP\_ID].descr\_uuid.uuid.uuid16 = ESP\_GATT\_UUID\_CHAR\_CLIENT\_CONFIG;   
 esp\_err\_t get\_attr\_ret = esp\_ble\_gatts\_get\_attr\_value(param->add\_char.attr\_handle, &length, &prf\_char);   
 if (get\_attr\_ret == ESP\_FAIL){   
 ESP\_LOGE(GATTS\_TAG, "ILLEGAL HANDLE");  
 }  
 ESP\_LOGI(GATTS\_TAG, "the gatts demo char length = %x\n", length);  
 for(int i = 0; i < length; i++){  
 ESP\_LOGI(GATTS\_TAG, "prf\_char[%x] = %x\n",i,prf\_char[i]);  
 }   
 esp\_err\_t add\_descr\_ret = esp\_ble\_gatts\_add\_char\_descr(   
 gl\_profile\_tab[PROFILE\_A\_APP\_ID].service\_handle,   
 &gl\_profile\_tab[PROFILE\_A\_APP\_ID].descr\_uuid,   
 ESP\_GATT\_PERM\_READ | ESP\_GATT\_PERM\_WRITE,   
 NULL,NULL);  
 if (add\_descr\_ret){  
 ESP\_LOGE(GATTS\_TAG, "add char descr failed, error code = %x", add\_descr\_ret);  
 }  
 break;  
 }

一旦添加了描述符，就会触发ESP\_GATTS\_ADD\_CHAR\_DESCR\_EVT事件，在本示例中用于打印一条信息消息。

case ESP\_GATTS\_ADD\_CHAR\_DESCR\_EVT:  
 ESP\_LOGI(GATTS\_TAG, "ADD\_DESCR\_EVT, status %d, attr\_handle %d, service\_handle %d\n",  
 param->add\_char.status, param->add\_char.attr\_handle,   
 param->add\_char.service\_handle);  
 break;

这个程序在配置文件B的事件处理程序中重复，以便为该配置文件创建服务和特征。

### 连接事件

当客户端连接到GATT服务器时，将触发ESP\_GATTS\_CONNECT\_EVT事件。此事件用于更新连接参数，例如延迟、最小连接间隔、最大连接间隔和超时时间。连接参数被存储在一个esp\_ble\_conn\_update\_params\_t结构体中，然后传递给esp\_ble\_gap\_update\_conn\_params()函数。更新连接参数的程序只需要执行一次，因此配置文件B的连接事件处理程序不包括esp\_ble\_gap\_update\_conn\_params()函数。最后，事件返回的连接ID被存储在配置文件表中。

Profile A 连接事件：

case ESP\_GATTS\_CONNECT\_EVT: {   
 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 reference the apple official documents about the ble connection parameters restrictions. \*/  
 conn\_params.latency = 0;   
 conn\_params.max\_int = 0x30; // max\_int = 0x30\*1.25ms = 40ms   
 conn\_params.min\_int = 0x10; // min\_int = 0x10\*1.25ms = 20ms   
 conn\_params.timeout = 400; // timeout = 400\*10ms = 4000ms   
 ESP\_LOGI(GATTS\_TAG, "ESP\_GATTS\_CONNECT\_EVT, conn\_id %d, remote %02x:%02x:%02x:%02x:%02x:%02x:, is\_conn %d",   
 param->connect.conn\_id,   
 param->connect.remote\_bda[0],   
 param->connect.remote\_bda[1],   
 param->connect.remote\_bda[2],   
 param->connect.remote\_bda[3],   
 param->connect.remote\_bda[4],   
 param->connect.remote\_bda[5],   
 param->connect.is\_connected);  
 gl\_profile\_tab[PROFILE\_A\_APP\_ID].conn\_id = param->connect.conn\_id;  
 //start sent the update connection parameters to the peer device.  
 esp\_ble\_gap\_update\_conn\_params(&conn\_params);  
 break;  
 }

Profile B连接事件：

case ESP\_GATTS\_CONNECT\_EVT:   
 ESP\_LOGI(GATTS\_TAG, "CONNECT\_EVT, conn\_id %d, remote %02x:%02x:%02x:%02x:%02x:%02x:, is\_conn %d\n",   
 param->connect.conn\_id,   
 param->connect.remote\_bda[0],   
 param->connect.remote\_bda[1],   
 param->connect.remote\_bda[2],   
 param->connect.remote\_bda[3],   
 param->connect.remote\_bda[4],   
 param->connect.remote\_bda[5],   
 param->connect.is\_connected);  
 gl\_profile\_tab[PROFILE\_B\_APP\_ID].conn\_id = param->connect.conn\_id;  
 break;

esp\_ble\_gap\_update\_conn\_params() 函数触发了一个 GAP 事件 ESP\_GAP\_BLE\_UPDATE\_CONN\_PARAMS\_EVT，这个函数用来打印连接信息：

case ESP\_GAP\_BLE\_UPDATE\_CONN\_PARAMS\_EVT:  
 ESP\_LOGI(GATTS\_TAG, "update connection params status = %d, min\_int = %d, max\_int = %d,  
 conn\_int = %d,latency = %d, timeout = %d",  
 param->update\_conn\_params.status,  
 param->update\_conn\_params.min\_int,  
 param->update\_conn\_params.max\_int,  
 param->update\_conn\_params.conn\_int,  
 param->update\_conn\_params.latency,  
 param->update\_conn\_params.timeout);  
 break;

### 管理读事件

既然服务和特征都已经创建并启动了，那么程序需要读取和写入事件。

读取操作被表示为 ESP\_GATTS\_READ\_EVT 事件，这个事件有以下参数：

uint16\_t conn\_id; /\*!< Connection id \*/  
uint32\_t trans\_id; /\*!< Transfer id \*/  
esp\_bd\_addr\_t bda; /\*!< The bluetooth device address which been read \*/  
uint16\_t handle; /\*!< The attribute handle \*/  
uint16\_t offset; /\*!< Offset of the value, if the value is too long \*/  
bool is\_long; /\*!< The value is too long or not \*/  
bool need\_rsp; /\*!< The read operation need to do response \*/

在这个示例中，使用由事件给定的相同句柄，构造了一个包含虚拟数据的响应，并使用

esp\_ble\_gatts\_send\_response()

函数将其发送回主机。

除了响应外，GATT接口、连接ID和传输ID也作为参数包含在esp\_ble\_gatts\_send\_response()函数中。如果在创建特征或描述符时，自动响应字节被设置为NULL，则需要此函数。

case ESP\_GATTS\_READ\_EVT: {  
 ESP\_LOGI(GATTS\_TAG, "GATT\_READ\_EVT, conn\_id %d, trans\_id %d, handle %d\n",   
 param->read.conn\_id, param->read.trans\_id, param->read.handle);   
 esp\_gatt\_rsp\_t rsp;   
 memset(&rsp, 0, sizeof(esp\_gatt\_rsp\_t));   
 rsp.attr\_value.handle = param->read.handle;   
 rsp.attr\_value.len = 4;   
 rsp.attr\_value.value[0] = 0xde;   
 rsp.attr\_value.value[1] = 0xed;   
 rsp.attr\_value.value[2] = 0xbe;   
 rsp.attr\_value.value[3] = 0xef;   
 esp\_ble\_gatts\_send\_response(gatts\_if,   
 param->read.conn\_id,   
 param->read.trans\_id,   
 ESP\_GATT\_OK, &rsp);  
 break;  
 }

### 管理写事件

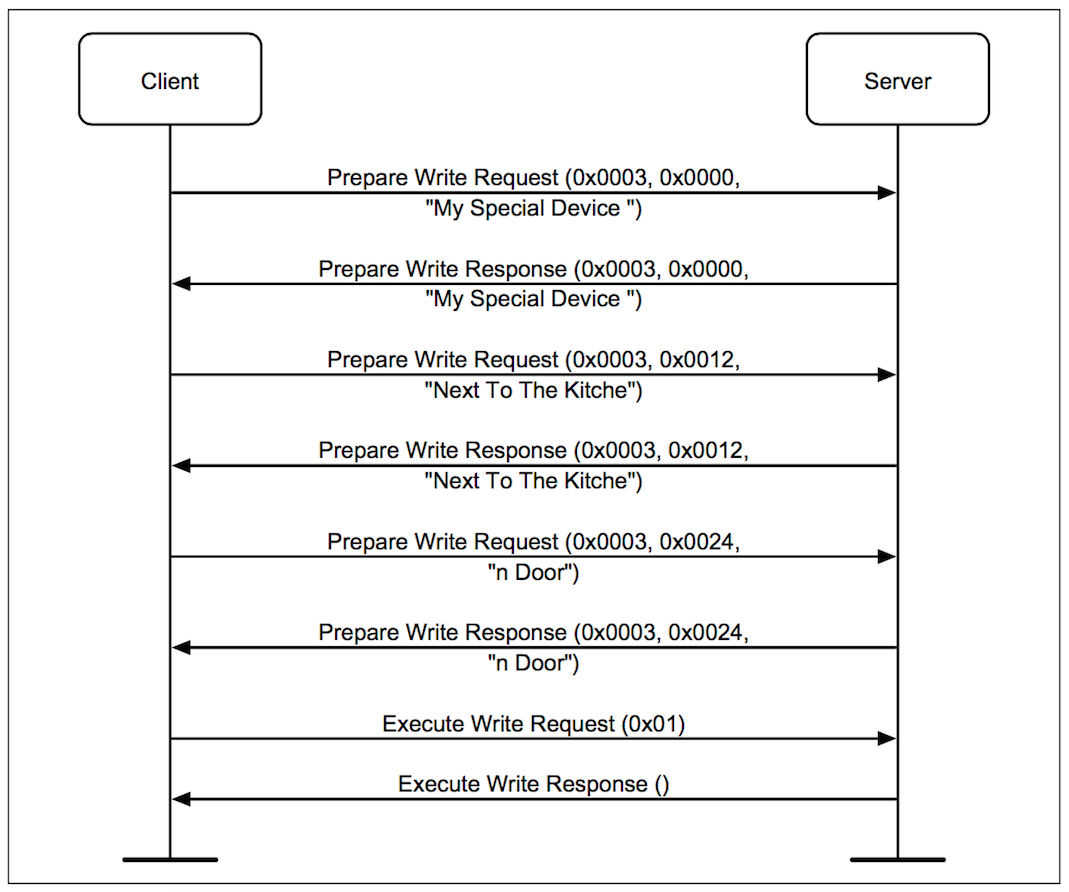
写事件被表示为 ESP\_GATTS\_WRITE\_EVT 事件，这个事件有以下参数：

uint16\_t conn\_id; /\*!< Connection id \*/  
uint32\_t trans\_id; /\*!< Transfer id \*/  
esp\_bd\_addr\_t bda; /\*!< The bluetooth device address which been written \*/  
uint16\_t handle; /\*!< The attribute handle \*/  
uint16\_t offset; /\*!< Offset of the value, if the value is too long \*/  
bool need\_rsp; /\*!< The write operation need to do response \*/  
bool is\_prep; /\*!< This write operation is prepare write \*/  
uint16\_t len; /\*!< The write attribute value length \*/  
uint8\_t \*value; /\*!< The write attribute value \*/

在这个示例中实现了两种类型的写入事件，写入特征值和写入长特征值。当特征值可以适应于一个属性协议最大传输单元（ATT MTU），通常为23字节时，使用第一种类型的写入。当要写入的属性长度超过一条ATT消息所能发送的长度时，使用第二种类型，通过使用准备写响应将数据分割成多个块，之后使用执行写请求来确认或取消完整的写请求。下图展示了写入长特征消息流程。

当写入事件被触发时，此示例会打印日志消息，然后执行example\_write\_event\_env()函数。

case ESP\_GATTS\_WRITE\_EVT: {   
 ESP\_LOGI(GATTS\_TAG, "GATT\_WRITE\_EVT, conn\_id %d, trans\_id %d, handle %d\n", param->write.conn\_id, param->write.trans\_id, param->write.handle);  
 if (!param->write.is\_prep){  
 ESP\_LOGI(GATTS\_TAG, "GATT\_WRITE\_EVT, value len %d, value :", param->write.len);  
 esp\_log\_buffer\_hex(GATTS\_TAG, param->write.value, param->write.len);  
 if (gl\_profile\_tab[PROFILE\_B\_APP\_ID].descr\_handle == param->write.handle && param->write.len == 2){  
 uint16\_t descr\_value= param->write.value[1]<<8 | param->write.value[0];  
 if (descr\_value == 0x0001){  
 if (b\_property & ESP\_GATT\_CHAR\_PROP\_BIT\_NOTIFY){  
 ESP\_LOGI(GATTS\_TAG, "notify enable");  
 uint8\_t notify\_data[15];  
 for (int i = 0; i < sizeof(notify\_data); ++i)  
 {  
 notify\_data[i] = i%0xff;   
 }  
 //the size of notify\_data[] need less than MTU size  
 esp\_ble\_gatts\_send\_indicate(gatts\_if, param->write.conn\_id,   
 gl\_profile\_tab[PROFILE\_B\_APP\_ID].char\_handle,   
 sizeof(notify\_data),   
 notify\_data, false);  
 }  
 }else if (descr\_value == 0x0002){  
 if (b\_property & ESP\_GATT\_CHAR\_PROP\_BIT\_INDICATE){  
 ESP\_LOGI(GATTS\_TAG, "indicate enable");  
 uint8\_t indicate\_data[15];  
 for (int i = 0; i < sizeof(indicate\_data); ++i)  
 {  
 indicate\_data[i] = i % 0xff;  
 }  
 //the size of indicate\_data[] need less than MTU size  
 esp\_ble\_gatts\_send\_indicate(gatts\_if, param->write.conn\_id,   
 gl\_profile\_tab[PROFILE\_B\_APP\_ID].char\_handle,   
 sizeof(indicate\_data),   
 indicate\_data, true);  
 }  
 }  
 else if (descr\_value == 0x0000){  
 ESP\_LOGI(GATTS\_TAG, "notify/indicate disable ");  
 }else{  
 ESP\_LOGE(GATTS\_TAG, "unknown value");  
 }  
 }  
 }  
 example\_write\_event\_env(gatts\_if, &a\_prepare\_write\_env, param);  
 break;  
}



example\_write\_event\_env()函数包含了写长特征过程的逻辑：

void example\_write\_event\_env(esp\_gatt\_if\_t gatts\_if, prepare\_type\_env\_t \*prepare\_write\_env, esp\_ble\_gatts\_cb\_param\_t \*param){  
 esp\_gatt\_status\_t status = ESP\_GATT\_OK;  
 if (param->write.need\_rsp){  
 if (param->write.is\_prep){  
 if (prepare\_write\_env->prepare\_buf == NULL){  
 prepare\_write\_env->prepare\_buf = (uint8\_t \*)malloc(PREPARE\_BUF\_MAX\_SIZE\*sizeof(uint8\_t));  
 prepare\_write\_env->prepare\_len = 0;  
 if (prepare\_write\_env->prepare\_buf == NULL) {  
 ESP\_LOGE(GATTS\_TAG, "Gatt\_server prep no mem\n");  
 status = ESP\_GATT\_NO\_RESOURCES;  
 }  
 } else {  
 if(param->write.offset > PREPARE\_BUF\_MAX\_SIZE) {  
 status = ESP\_GATT\_INVALID\_OFFSET;  
 }  
 else if ((param->write.offset + param->write.len) > PREPARE\_BUF\_MAX\_SIZE) {  
 status = ESP\_GATT\_INVALID\_ATTR\_LEN;  
 }  
 }  
  
 esp\_gatt\_rsp\_t \*gatt\_rsp = (esp\_gatt\_rsp\_t \*)malloc(sizeof(esp\_gatt\_rsp\_t));  
 gatt\_rsp->attr\_value.len = param->write.len;  
 gatt\_rsp->attr\_value.handle = param->write.handle;  
 gatt\_rsp->attr\_value.offset = param->write.offset;  
 gatt\_rsp->attr\_value.auth\_req = ESP\_GATT\_AUTH\_REQ\_NONE;  
 memcpy(gatt\_rsp->attr\_value.value, param->write.value, param->write.len);  
 esp\_err\_t response\_err = esp\_ble\_gatts\_send\_response(gatts\_if, param->write.conn\_id,   
 param->write.trans\_id, status, gatt\_rsp);  
 if (response\_err != ESP\_OK){  
 ESP\_LOGE(GATTS\_TAG, "Send response error\n");  
 }  
 free(gatt\_rsp);  
 if (status != ESP\_GATT\_OK){  
 return;  
 }  
 memcpy(prepare\_write\_env->prepare\_buf + param->write.offset,  
 param->write.value,  
 param->write.len);  
 prepare\_write\_env->prepare\_len += param->write.len;  
  
 }else{  
 esp\_ble\_gatts\_send\_response(gatts\_if, param->write.conn\_id, param->write.trans\_id, status, NULL);  
 }  
 }  
}

当客户端发送写请求或准备写请求时，服务器应当响应。然而，如果客户端发送一个无需响应的写命令，服务器不需要回复响应。通过检查write.need\_rsp参数的值来在写入过程中进行检查。如果需要响应，则继续执行响应准备；如果不存在，则客户端不需要响应，因此过程结束。

void example\_write\_event\_env(esp\_gatt\_if\_t gatts\_if, prepare\_type\_env\_t \*prepare\_write\_env,   
 esp\_ble\_gatts\_cb\_param\_t \*param){  
 esp\_gatt\_status\_t status = ESP\_GATT\_OK;  
 if (param->write.need\_rsp){  
...

然后，函数检查代表write.is\_prep的准备写请求参数是否被设置，这意味着客户端正在请求一个长特征写入。如果存在，程序将继续准备多个写响应；如果不存在，则服务器简单地发送回一个单一的写响应。

...  
if (param->write.is\_prep) {  
...  
} else {  
 esp\_ble\_gatts\_send\_response(gatts\_if, param->write.conn\_id, param->write.trans\_id, status, NULL);  
}  
...

为了处理长特征写事件，必须定义和初始化一个预备缓冲区：

typedef struct {  
 uint8\_t \*prepare\_buf;  
 int prepare\_len;  
} prepare\_type\_env\_t;  
  
static prepare\_type\_env\_t a\_prepare\_write\_env;  
static prepare\_type\_env\_t b\_prepare\_write\_env;

为了使用预备缓冲区，需要分配一些内存。为了预防因为内存不够而造成的内存分配失败，需要打印错误：

if (prepare\_write\_env->prepare\_buf == NULL) {  
 prepare\_write\_env->prepare\_buf =   
 (uint8\_t\*)malloc(PREPARE\_BUF\_MAX\_SIZE\*sizeof(uint8\_t));   
 prepare\_write\_env->prepare\_len = 0;  
 if (prepare\_write\_env->prepare\_buf == NULL) {   
 ESP\_LOGE(GATTS\_TAG, "Gatt\_server prep no mem\n");  
 status = ESP\_GATT\_NO\_RESOURCES;  
 }  
}

如果缓冲区不是NULL，这意味着初始化完成了。下面的代码检查了偏移量以及写入内容的消息长度是否能装进缓冲区。

else {  
 if(param->write.offset > PREPARE\_BUF\_MAX\_SIZE) {  
 status = ESP\_GATT\_INVALID\_OFFSET;  
 }  
 else if ((param->write.offset + param->write.len) > PREPARE\_BUF\_MAX\_SIZE) {  
 status = ESP\_GATT\_INVALID\_ATTR\_LEN;  
 }  
}

现在，程序准备了要发送回客户端的esp\_gatt\_rsp\_t类型的响应。响应是使用写请求的相同参数构造的，例如长度、句柄和偏移量。此外，设置了写入此特征所需的GATT认证类型为ESP\_GATT\_AUTH\_REQ\_NONE，这意味着客户端可以在不需要先进行认证的情况下写入此特征。一旦响应被发送，为其使用分配的内存将被释放。

esp\_gatt\_rsp\_t \*gatt\_rsp = (esp\_gatt\_rsp\_t \*)malloc(sizeof(esp\_gatt\_rsp\_t));  
gatt\_rsp->attr\_value.len = param->write.len;  
gatt\_rsp->attr\_value.handle = param->write.handle;  
gatt\_rsp->attr\_value.offset = param->write.offset;  
gatt\_rsp->attr\_value.auth\_req = ESP\_GATT\_AUTH\_REQ\_NONE;  
memcpy(gatt\_rsp->attr\_value.value, param->write.value, param->write.len);  
esp\_err\_t response\_err = esp\_ble\_gatts\_send\_response(gatts\_if, param->write.conn\_id,   
 param->write.trans\_id, status, gatt\_rsp);  
if (response\_err != ESP\_OK){  
 ESP\_LOGE(GATTS\_TAG, "Send response error\n");  
}  
free(gatt\_rsp);  
if (status != ESP\_GATT\_OK){  
 return;  
}

最终，输入数据被拷贝到创建的缓冲区中，缓冲区的长度需要加上偏移量：

memcpy(prepare\_write\_env->prepare\_buf + param->write.offset,  
 param->write.value,   
 param->write.len);  
prepare\_write\_env->prepare\_len += param->write.len;

客户端通过发送执行写请求来完成长写序列。这个命令触发一个ESP\_GATTS\_EXEC\_WRITE\_EVT事件。服务器通过发送响应并执行example\_exec\_write\_event\_env()函数来处理这个事件：

case ESP\_GATTS\_EXEC\_WRITE\_EVT:   
 ESP\_LOGI(GATTS\_TAG,"ESP\_GATTS\_EXEC\_WRITE\_EVT");   
 esp\_ble\_gatts\_send\_response(gatts\_if, param->write.conn\_id, param->write.trans\_id, ESP\_GATT\_OK, NULL);   
 example\_exec\_write\_event\_env(&a\_prepare\_write\_env, param);   
 break;

我们来看一下执行写函数：

void example\_exec\_write\_event\_env(prepare\_type\_env\_t \*prepare\_write\_env, esp\_ble\_gatts\_cb\_param\_t \*param){  
 if (param->exec\_write.exec\_write\_flag == ESP\_GATT\_PREP\_WRITE\_EXEC){  
 esp\_log\_buffer\_hex(GATTS\_TAG, prepare\_write\_env->prepare\_buf, prepare\_write\_env->prepare\_len);  
 }  
 else{  
 ESP\_LOGI(GATTS\_TAG,"ESP\_GATT\_PREP\_WRITE\_CANCEL");  
 }  
 if (prepare\_write\_env->prepare\_buf) {  
 free(prepare\_write\_env->prepare\_buf);  
 prepare\_write\_env->prepare\_buf = NULL;  
 }  
#### prepare\_write\_env->prepare\_len = 0;  
}

执行写操作用于通过长特征写入程序来确认或取消之前完成的写入过程。为了做到这一点，函数检查与事件一起接收的参数中的exec\_write\_flag。如果标志等于由exec\_write\_flag表示的执行标志，则写入被确认，并且缓冲区的内容将被打印在日志中；如果不是，则意味着写入被取消，所有已写入的数据将被删除。

if (param->exec\_write.exec\_write\_flag == ESP\_GATT\_PREP\_WRITE\_EXEC) {   
 esp\_log\_buffer\_hex(GATTS\_TAG,   
 prepare\_write\_env->prepare\_buf,   
 prepare\_write\_env->prepare\_len);  
} else {  
 ESP\_LOGI(GATTS\_TAG,"ESP\_GATT\_PREP\_WRITE\_CANCEL");  
}

最后，为了存储来自长写操作的数据块而创建的缓冲区结构被释放，其指针被设置为NULL，以便为下一个长写操作做好准备。

if (prepare\_write\_env->prepare\_buf) {  
 free(prepare\_write\_env->prepare\_buf);  
 prepare\_write\_env->prepare\_buf = NULL;  
}  
prepare\_write\_env->prepare\_len = 0;

### 总结

在本文档中，我们详细介绍了GATT服务器示例代码的每个部分。该应用程序是围绕应用程序配置文件的概念设计的。此外，还解释了此示例用于注册事件处理程序的程序。事件遵循一系列配置步骤，例如定义广告参数、更新连接参数以及创建服务和特征。最后，解释了如何处理读写事件，包括通过将写操作分割成可以适应属性协议消息的块来实现长特征的写入。

# WIFI模块

wifi模块相对蓝牙就简单很多了。

头文件如下：

#ifndef \_\_WIFI\_DRIVER\_H\_  
#define \_\_WIFI\_DRIVER\_H\_  
  
#include "freertos/FreeRTOS.h"  
#include "freertos/event\_groups.h"  
#include "esp\_log.h"  
#include "nvs\_flash.h"  
#include "esp\_mac.h"  
#include "esp\_wifi.h"  
#include "esp\_event.h"  
  
#define EXAMPLE\_ESP\_WIFI\_SSID "用户名"  
#define EXAMPLE\_ESP\_WIFI\_PASS "密码"  
#define EXAMPLE\_ESP\_MAXIMUM\_RETRY 5  
  
#define ESP\_WIFI\_SAE\_MODE WPA3\_SAE\_PWE\_BOTH  
#define EXAMPLE\_H2E\_IDENTIFIER CONFIG\_ESP\_WIFI\_PW\_ID  
#define ESP\_WIFI\_SCAN\_AUTH\_MODE\_THRESHOLD WIFI\_AUTH\_WPA2\_PSK  
  
#define WIFI\_CONNECTED\_BIT BIT0  
#define WIFI\_FAIL\_BIT BIT1  
  
void event\_handler(void \*arg, esp\_event\_base\_t event\_base,  
 int32\_t event\_id, void \*event\_data);  
void wifi\_init\_sta(void);  
void WIFI\_Init(void);  
  
#endif

对应的实现

#include "wifi\_driver.h"  
  
EventGroupHandle\_t s\_wifi\_event\_group;  
  
const char \*WIFI\_TAG = "wifi station";  
int s\_retry\_num = 0;  
  
void event\_handler(void \*arg, esp\_event\_base\_t event\_base,  
 int32\_t event\_id, void \*event\_data)  
{  
 if (event\_base == WIFI\_EVENT && event\_id == WIFI\_EVENT\_STA\_START)  
 {  
 esp\_wifi\_connect();  
 }  
 else if (event\_base == WIFI\_EVENT && event\_id == WIFI\_EVENT\_STA\_DISCONNECTED)  
 {  
 if (s\_retry\_num < EXAMPLE\_ESP\_MAXIMUM\_RETRY)  
 {  
 esp\_wifi\_connect();  
 s\_retry\_num++;  
 ESP\_LOGI(WIFI\_TAG, "retry to connect to the AP");  
 }  
 else  
 {  
 xEventGroupSetBits(s\_wifi\_event\_group, WIFI\_FAIL\_BIT);  
 }  
 ESP\_LOGI(WIFI\_TAG, "connect to the AP fail");  
 }  
 else if (event\_base == IP\_EVENT && event\_id == IP\_EVENT\_STA\_GOT\_IP)  
 {  
 ip\_event\_got\_ip\_t \*event = (ip\_event\_got\_ip\_t \*)event\_data;  
 ESP\_LOGI(WIFI\_TAG, "got ip:" IPSTR, IP2STR(&event->ip\_info.ip));  
 s\_retry\_num = 0;  
 xEventGroupSetBits(s\_wifi\_event\_group, WIFI\_CONNECTED\_BIT);  
 }  
}  
  
void wifi\_init\_sta(void)  
{  
 s\_wifi\_event\_group = xEventGroupCreate();  
  
 ESP\_ERROR\_CHECK(esp\_netif\_init());  
  
 ESP\_ERROR\_CHECK(esp\_event\_loop\_create\_default());  
 esp\_netif\_create\_default\_wifi\_sta();  
  
 wifi\_init\_config\_t cfg = WIFI\_INIT\_CONFIG\_DEFAULT();  
 ESP\_ERROR\_CHECK(esp\_wifi\_init(&cfg));  
  
 esp\_event\_handler\_instance\_t instance\_any\_id;  
 esp\_event\_handler\_instance\_t instance\_got\_ip;  
 ESP\_ERROR\_CHECK(esp\_event\_handler\_instance\_register(WIFI\_EVENT,  
 ESP\_EVENT\_ANY\_ID,  
 &event\_handler,  
 NULL,  
 &instance\_any\_id));  
 ESP\_ERROR\_CHECK(esp\_event\_handler\_instance\_register(IP\_EVENT,  
 IP\_EVENT\_STA\_GOT\_IP,  
 &event\_handler,  
 NULL,  
 &instance\_got\_ip));  
  
 wifi\_config\_t wifi\_config = {  
 .sta = {  
 .ssid = EXAMPLE\_ESP\_WIFI\_SSID,  
 .password = EXAMPLE\_ESP\_WIFI\_PASS,  
 /\* Authmode threshold resets to WPA2 as default if password matches WPA2 standards (pasword len => 8).  
 \* If you want to connect the device to deprecated WEP/WPA networks, Please set the threshold value  
 \* to WIFI\_AUTH\_WEP/WIFI\_AUTH\_WPA\_PSK and set the password with length and format matching to  
 \* WIFI\_AUTH\_WEP/WIFI\_AUTH\_WPA\_PSK standards.  
 \*/  
 .threshold.authmode = ESP\_WIFI\_SCAN\_AUTH\_MODE\_THRESHOLD,  
 .sae\_pwe\_h2e = ESP\_WIFI\_SAE\_MODE,  
 .sae\_h2e\_identifier = EXAMPLE\_H2E\_IDENTIFIER,  
 },  
 };  
 ESP\_ERROR\_CHECK(esp\_wifi\_set\_mode(WIFI\_MODE\_STA));  
 ESP\_ERROR\_CHECK(esp\_wifi\_set\_config(WIFI\_IF\_STA, &wifi\_config));  
 ESP\_ERROR\_CHECK(esp\_wifi\_start());  
  
 ESP\_LOGI(WIFI\_TAG, "wifi\_init\_sta finished.");  
  
 /\* Waiting until either the connection is established (WIFI\_CONNECTED\_BIT) or connection failed for the maximum  
 \* number of re-tries (WIFI\_FAIL\_BIT). The bits are set by event\_handler() (see above) \*/  
 EventBits\_t bits = xEventGroupWaitBits(s\_wifi\_event\_group,  
 WIFI\_CONNECTED\_BIT | WIFI\_FAIL\_BIT,  
 pdFALSE,  
 pdFALSE,  
 portMAX\_DELAY);  
  
 /\* xEventGroupWaitBits() returns the bits before the call returned, hence we can test which event actually  
 \* happened. \*/  
 if (bits & WIFI\_CONNECTED\_BIT)  
 {  
 ESP\_LOGI(WIFI\_TAG, "connected to ap SSID:%s password:%s",  
 EXAMPLE\_ESP\_WIFI\_SSID, EXAMPLE\_ESP\_WIFI\_PASS);  
 }  
 else if (bits & WIFI\_FAIL\_BIT)  
 {  
 ESP\_LOGI(WIFI\_TAG, "Failed to connect to SSID:%s, password:%s",  
 EXAMPLE\_ESP\_WIFI\_SSID, EXAMPLE\_ESP\_WIFI\_PASS);  
 }  
 else  
 {  
 ESP\_LOGE(WIFI\_TAG, "UNEXPECTED EVENT");  
 }  
}  
  
void WIFI\_Init(void)  
{  
 // Initialize NVS  
 esp\_err\_t ret = nvs\_flash\_init();  
 if (ret == ESP\_ERR\_NVS\_NO\_FREE\_PAGES || ret == ESP\_ERR\_NVS\_NEW\_VERSION\_FOUND)  
 {  
 ESP\_ERROR\_CHECK(nvs\_flash\_erase());  
 ret = nvs\_flash\_init();  
 }  
 ESP\_ERROR\_CHECK(ret);  
  
 ESP\_LOGI(WIFI\_TAG, "ESP\_WIFI\_MODE\_STA");  
 wifi\_init\_sta();  
}

# TCPIP服务

然后我们在wifi模块的基础上，构建一个tcpip服务器，用来接收tcpip消息。

头文件如下：

#ifndef \_\_TCP\_DRIVER\_H\_  
#define \_\_TCP\_DRIVER\_H\_  
  
#include "esp\_log.h"  
  
#include "lwip/err.h"  
#include "lwip/sockets.h"  
#include "lwip/sys.h"  
#include <lwip/netdb.h>  
  
#define PORT CONFIG\_EXAMPLE\_PORT  
#define KEEPALIVE\_IDLE CONFIG\_EXAMPLE\_KEEPALIVE\_IDLE  
#define KEEPALIVE\_INTERVAL CONFIG\_EXAMPLE\_KEEPALIVE\_INTERVAL  
#define KEEPALIVE\_COUNT CONFIG\_EXAMPLE\_KEEPALIVE\_COUNT  
  
void do\_retransmit(const int sock);  
void tcp\_server\_task(void \*pvParameters);  
  
#endif

对应的实现如下

#include "tcp\_driver.h"  
#include "motor\_driver.h"  
  
const char \*TCP\_TAG = "TCP服务器消息";  
  
void do\_retransmit(const int sock)  
{  
 int len;  
 char rx\_buffer[128];  
  
 do  
 {  
 len = recv(sock, rx\_buffer, sizeof(rx\_buffer) - 1, 0);  
 if (len < 0)  
 {  
 ESP\_LOGE(TCP\_TAG, "Error occurred during receiving: errno %d", errno);  
 }  
 else if (len == 0)  
 {  
 ESP\_LOGW(TCP\_TAG, "Connection closed");  
 }  
 else  
 {  
 rx\_buffer[len] = 0; // Null-terminate whatever is received and treat it like a string  
 ESP\_LOGI(TCP\_TAG, "Received %d bytes: %s", len, rx\_buffer);  
 if (rx\_buffer[0] == 'a' && rx\_buffer[1] == 't') {  
 MOTOR\_Open\_lock();  
 printf("通过wifi开锁成功\r\n");  
 }  
  
 // send() can return less bytes than supplied length.  
 // Walk-around for robust implementation.  
 int to\_write = len;  
 while (to\_write > 0)  
 {  
 int written = send(sock, rx\_buffer + (len - to\_write), to\_write, 0);  
 if (written < 0)  
 {  
 ESP\_LOGE(TCP\_TAG, "Error occurred during sending: errno %d", errno);  
 // Failed to retransmit, giving up  
 return;  
 }  
 to\_write -= written;  
 }  
 }  
 } while (len > 0);  
}  
  
void tcp\_server\_task(void \*pvParameters)  
{  
 char addr\_str[128];  
 int addr\_family = (int)pvParameters;  
 int ip\_protocol = 0;  
 int keepAlive = 1;  
 int keepIdle = KEEPALIVE\_IDLE;  
 int keepInterval = KEEPALIVE\_INTERVAL;  
 int keepCount = KEEPALIVE\_COUNT;  
 struct sockaddr\_storage dest\_addr;  
  
 if (addr\_family == AF\_INET)  
 {  
 struct sockaddr\_in \*dest\_addr\_ip4 = (struct sockaddr\_in \*)&dest\_addr;  
 dest\_addr\_ip4->sin\_addr.s\_addr = htonl(INADDR\_ANY);  
 dest\_addr\_ip4->sin\_family = AF\_INET;  
 dest\_addr\_ip4->sin\_port = htons(PORT);  
 ip\_protocol = IPPROTO\_IP;  
 }  
  
 int listen\_sock = socket(addr\_family, SOCK\_STREAM, ip\_protocol);  
 if (listen\_sock < 0)  
 {  
 ESP\_LOGE(TCP\_TAG, "Unable to create socket: errno %d", errno);  
 vTaskDelete(NULL);  
 return;  
 }  
 int opt = 1;  
 setsockopt(listen\_sock, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt));  
  
 ESP\_LOGI(TCP\_TAG, "Socket created");  
  
 int err = bind(listen\_sock, (struct sockaddr \*)&dest\_addr, sizeof(dest\_addr));  
 if (err != 0)  
 {  
 ESP\_LOGE(TCP\_TAG, "Socket unable to bind: errno %d", errno);  
 ESP\_LOGE(TCP\_TAG, "IPPROTO: %d", addr\_family);  
 goto CLEAN\_UP;  
 }  
 ESP\_LOGI(TCP\_TAG, "Socket bound, port %d", PORT);  
  
 err = listen(listen\_sock, 1);  
 if (err != 0)  
 {  
 ESP\_LOGE(TCP\_TAG, "Error occurred during listen: errno %d", errno);  
 goto CLEAN\_UP;  
 }  
  
 while (1)  
 {  
  
 ESP\_LOGI(TCP\_TAG, "Socket listening");  
  
 struct sockaddr\_storage source\_addr; // Large enough for both IPv4 or IPv6  
 socklen\_t addr\_len = sizeof(source\_addr);  
 int sock = accept(listen\_sock, (struct sockaddr \*)&source\_addr, &addr\_len);  
 if (sock < 0)  
 {  
 ESP\_LOGE(TCP\_TAG, "Unable to accept connection: errno %d", errno);  
 break;  
 }  
  
 // Set tcp keepalive option  
 setsockopt(sock, SOL\_SOCKET, SO\_KEEPALIVE, &keepAlive, sizeof(int));  
 setsockopt(sock, IPPROTO\_TCP, TCP\_KEEPIDLE, &keepIdle, sizeof(int));  
 setsockopt(sock, IPPROTO\_TCP, TCP\_KEEPINTVL, &keepInterval, sizeof(int));  
 setsockopt(sock, IPPROTO\_TCP, TCP\_KEEPCNT, &keepCount, sizeof(int));  
 // Convert ip address to string  
 if (source\_addr.ss\_family == PF\_INET)  
 {  
 inet\_ntoa\_r(((struct sockaddr\_in \*)&source\_addr)->sin\_addr, addr\_str, sizeof(addr\_str) - 1);  
 }  
 ESP\_LOGI(TCP\_TAG, "Socket accepted ip address: %s", addr\_str);  
  
 do\_retransmit(sock);  
  
 shutdown(sock, 0);  
 close(sock);  
 }  
  
CLEAN\_UP:  
 close(listen\_sock);  
 vTaskDelete(NULL);  
}

# OTA功能

OTA模块的参考代码位于examples/system/ota/simple\_ota\_example文件夹。

我们可以在线更新ESP32的固件，也就是说通过WIFI下载新的固件然后替换掉旧的固件，实现在线升级功能。

这部分代码也比较简单。首先我们要修改以下 Flash 的分区信息表。

分区信息表如下：

# Name, Type, SubType, Offset, Size, Flags  
# Note: if you have increased the bootloader size, make sure to update the offsets to avoid overlap  
  
nvs, data, nvs, , 0x4000,  
otadata, data, ota, , 0x2000,  
phy\_init, data, phy, , 0x1000,  
ota\_0, app, ota\_0, , 1800K,  
ota\_1, app, ota\_1, , 1800K,

然后执行进入菜单命令，将Flash大小修改为4MB。

idf.py menuconfig

然后编写头文件代码：

#ifndef \_\_OTA\_DRIVER\_H\_  
#define \_\_OTA\_DRIVER\_H\_  
  
#include "freertos/FreeRTOS.h"  
#include "freertos/task.h"  
#include "esp\_system.h"  
#include "esp\_event.h"  
#include "esp\_log.h"  
#include "esp\_ota\_ops.h"  
#include "esp\_http\_client.h"  
#include "esp\_https\_ota.h"  
#include "string.h"  
#include "esp\_crt\_bundle.h"  
#include "esp\_wifi.h"  
#include "esp\_netif.h"  
  
  
#define HASH\_LEN 32  
#define OTA\_URL\_SIZE 256  
#define EXAMPLE\_NETIF\_DESC\_STA "example\_netif\_sta"  
  
void get\_sha256\_of\_partitions(void);  
void ota\_task(void);  
  
#endif

然后编写对应的实现代码

#include "ota\_driver.h"  
  
static const char \*bind\_interface\_name = EXAMPLE\_NETIF\_DESC\_STA;  
  
static const char \*TAG = "OTA任务: ";  
extern const uint8\_t server\_cert\_pem\_start[] asm("\_binary\_ca\_cert\_pem\_start");  
extern const uint8\_t server\_cert\_pem\_end[] asm("\_binary\_ca\_cert\_pem\_end");  
  
esp\_netif\_t \*get\_example\_netif\_from\_desc(const char \*desc)  
{  
 esp\_netif\_t \*netif = NULL;  
 while ((netif = esp\_netif\_next(netif)) != NULL)  
 {  
 if (strcmp(esp\_netif\_get\_desc(netif), desc) == 0)  
 {  
 return netif;  
 }  
 }  
 return netif;  
}  
  
static void print\_sha256(const uint8\_t \*image\_hash, const char \*label)  
{  
 char hash\_print[HASH\_LEN \* 2 + 1];  
 hash\_print[HASH\_LEN \* 2] = 0;  
 for (int i = 0; i < HASH\_LEN; ++i)  
 {  
 sprintf(&hash\_print[i \* 2], "%02x", image\_hash[i]);  
 }  
 ESP\_LOGI(TAG, "%s %s", label, hash\_print);  
}  
  
void get\_sha256\_of\_partitions(void)  
{  
 uint8\_t sha\_256[HASH\_LEN] = {0};  
 esp\_partition\_t partition;  
  
 // get sha256 digest for bootloader  
 partition.address = ESP\_BOOTLOADER\_OFFSET;  
 partition.size = ESP\_PARTITION\_TABLE\_OFFSET;  
 partition.type = ESP\_PARTITION\_TYPE\_APP;  
 esp\_partition\_get\_sha256(&partition, sha\_256);  
 print\_sha256(sha\_256, "SHA-256 for bootloader: ");  
  
 // get sha256 digest for running partition  
 esp\_partition\_get\_sha256(esp\_ota\_get\_running\_partition(), sha\_256);  
 print\_sha256(sha\_256, "SHA-256 for current firmware: ");  
}  
  
esp\_err\_t \_http\_event\_handler(esp\_http\_client\_event\_t \*evt)  
{  
 switch (evt->event\_id)  
 {  
 case HTTP\_EVENT\_ERROR:  
 ESP\_LOGD(TAG, "HTTP\_EVENT\_ERROR");  
 break;  
 case HTTP\_EVENT\_ON\_CONNECTED:  
 ESP\_LOGD(TAG, "HTTP\_EVENT\_ON\_CONNECTED");  
 break;  
 case HTTP\_EVENT\_HEADER\_SENT:  
 ESP\_LOGD(TAG, "HTTP\_EVENT\_HEADER\_SENT");  
 break;  
 case HTTP\_EVENT\_ON\_HEADER:  
 ESP\_LOGD(TAG, "HTTP\_EVENT\_ON\_HEADER, key=%s, value=%s", evt->header\_key, evt->header\_value);  
 break;  
 case HTTP\_EVENT\_ON\_DATA:  
 ESP\_LOGD(TAG, "HTTP\_EVENT\_ON\_DATA, len=%d", evt->data\_len);  
 break;  
 case HTTP\_EVENT\_ON\_FINISH:  
 ESP\_LOGD(TAG, "HTTP\_EVENT\_ON\_FINISH");  
 break;  
 case HTTP\_EVENT\_DISCONNECTED:  
 ESP\_LOGD(TAG, "HTTP\_EVENT\_DISCONNECTED");  
 break;  
 case HTTP\_EVENT\_REDIRECT:  
 ESP\_LOGD(TAG, "HTTP\_EVENT\_REDIRECT");  
 break;  
 }  
 return ESP\_OK;  
}  
  
void ota\_task(void)  
{  
 ESP\_LOGI(TAG, "OTA任务开始了。\r\n");  
 esp\_http\_client\_config\_t config = {  
 .url = "http://192.168.232.162:8070/led\_strip.bin",  
 .crt\_bundle\_attach = esp\_crt\_bundle\_attach,  
 .event\_handler = \_http\_event\_handler,  
 .keep\_alive\_enable = true,  
 };  
  
 esp\_https\_ota\_config\_t ota\_config = {  
 .http\_config = &config,  
 };  
 ESP\_LOGI(TAG, "Attempting to download update from %s", config.url);  
 esp\_err\_t ret = esp\_https\_ota(&ota\_config);  
 if (ret == ESP\_OK)  
 {  
 ESP\_LOGI(TAG, "OTA Succeed, Rebooting...");  
 esp\_restart();  
 }  
 else  
 {  
 ESP\_LOGE(TAG, "Firmware upgrade failed");  
 }  
}

在Windows中打开Anaconda PowerShell Prompt。

然后在 二进制文件所在目录 执行以下命令：

python -m http.server 8070