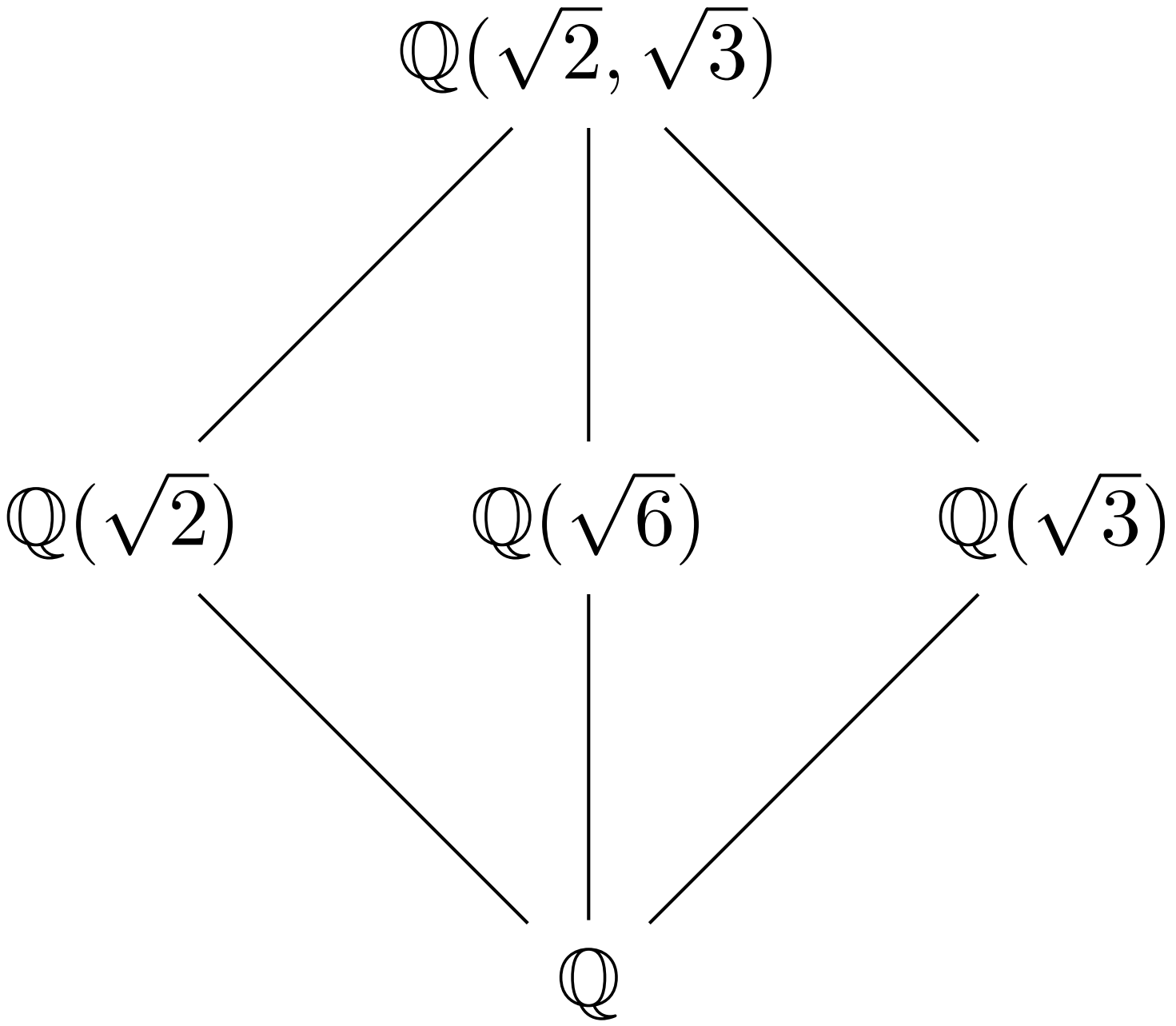

ESP32 **教程**

作者：左元

目录

第一章 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。

第二章 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。

2.1 离线安装 ESP-IDF

点击链接下载离线安装包。

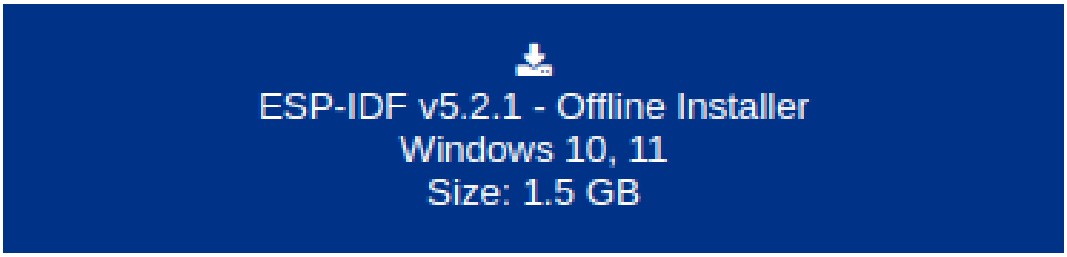

图 2.1: 离线安装包

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:

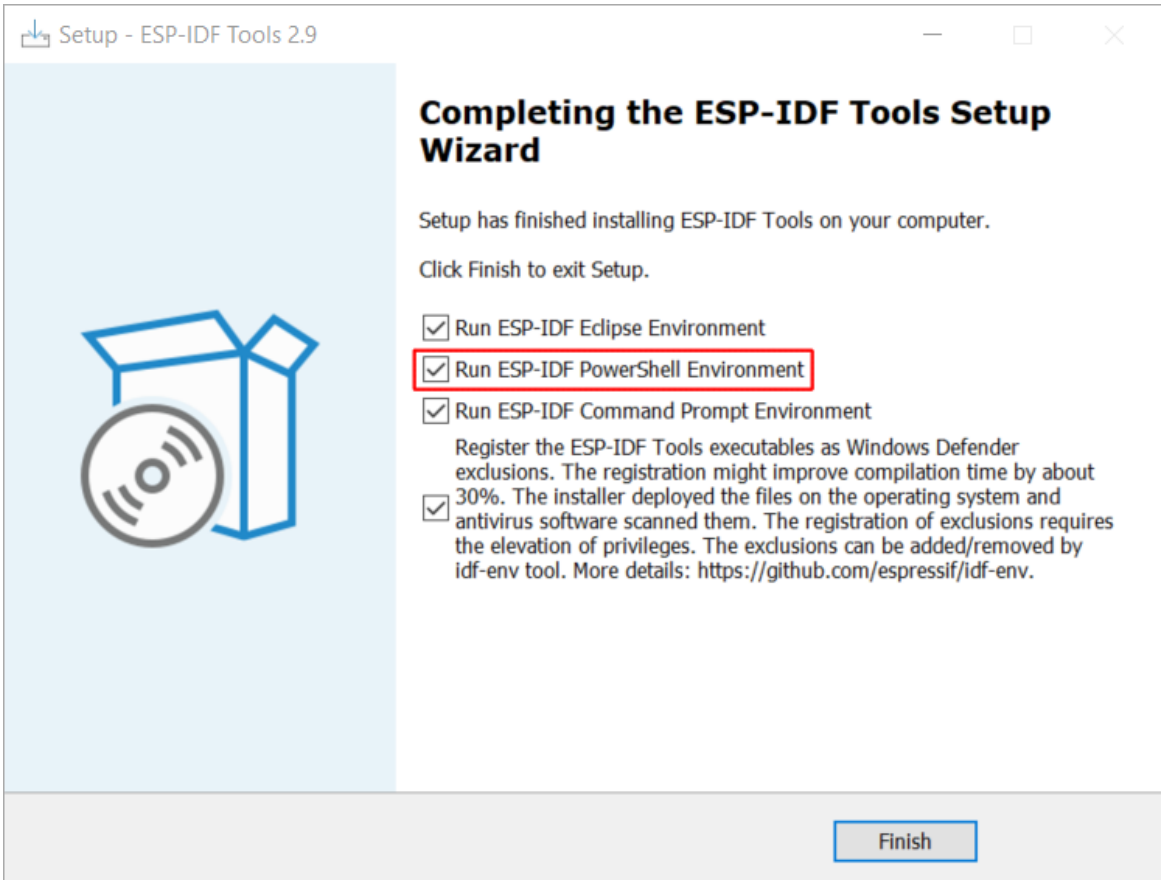

图 2.2: PowerShell

第三章 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：

复制工程命令

```
$ 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。

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 。

配置命令

```
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。

3.3 编译工程

请使用以下命令，编译烧录工程：

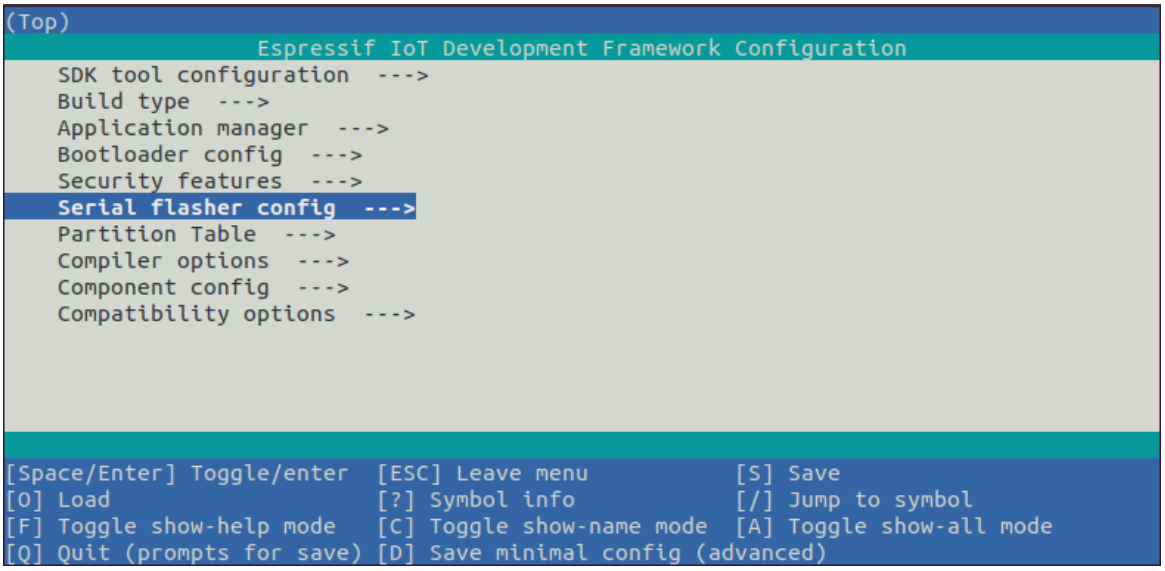

图 3.1: 配置界面示意图

编译工程的命令

```
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。

运行示意图

```
$ 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：

编译加烧录

```
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`。

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。

> ! 当要进行烧写时，请关闭串口助手！

第四章 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：

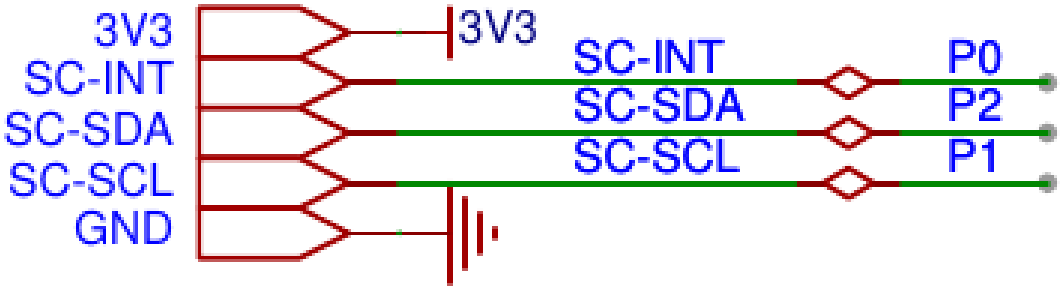

图 4.1: 电容键盘电路图

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。

1. 在 `~/esp` 文件夹中创建新的文件夹 `atguigu-gpio-example` 。
2. 在 `atguigu-gpio-example` 文件夹中创建文件 `CMakeLists.txt` 。
3. 在 `atguigu-gpio-example` 文件夹中创建 `main` 文件夹。
4. 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
├── main
│   ├── CMakeLists.txt
│   └── gpio_main.c
└── CMakeLists.txt
```

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

atguigu-gpio-example/CMakeLists.txt

```
1 # cmake版本管理工具的最小版本号
2 cmake_minimum_required(VERSION 3.16)
3
4 # 项目的cmake文件路径，包含在 ESP32 的 SDK 中
5 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：

atguigu-gpio-example/main/CMakeLists.txt

```
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。

atguigu-gpio-example/main/gpio_main.c

```
/// 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。

atguigu-gpio-example/main/gpio_main.c

```
/// 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。

atguigu-gpio-example/main/gpio_main.c

```
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 I^2C 协议。因为 ESP32-C3 通过 I^2C 协议和电容键盘进行通信。

由于 I^2C 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 I^2C 通信。

atguigu-gpio-example/main/gpio_main.c

```
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 I^2C 通信。

atguigu-gpio-example/main/gpio_main.c

```
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。

atguigu-gpio-example/main/gpio_main.c

```
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 。

atguigu-gpio-example/main/gpio_main.c

```
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。

atguigu-gpio-example/main/gpio_main.c

```
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。

atguigu-gpio-example/main/gpio_main.c

```
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。

atguigu-gpio-example/main/gpio_main.c

```
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。

atguigu-gpio-example/main/gpio_main.c

```
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：

atguigu-gpio-example/main/gpio_main.c

```
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 I^2C 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。

atguigu-gpio-example/main/gpio_main.c

```
/// 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。

atguigu-gpio-example/main/gpio_main.c

```
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。

atguigu-gpio-example/main/gpio_main.c

```
/// 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。

atguigu-gpio-example/main/gpio_main.c

```c
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。

atguigu-gpio-example/main/gpio_main.c

```c
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。

atguigu-gpio-example/main/gpio_main.c

```
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：

编译并烧写程序

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

然后就可以测试程序了。

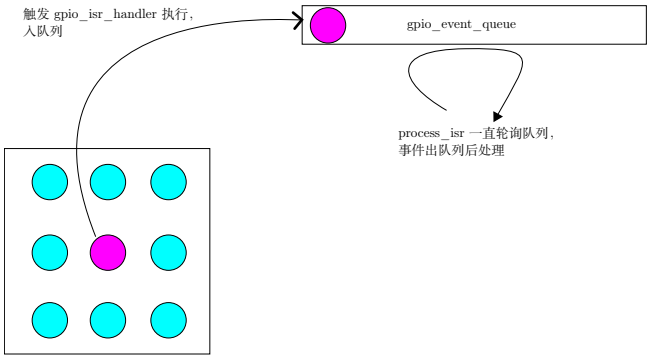

图 4.2: 处理中断

第五章 信号处理入门

5.1 正弦波定义

图 5.1 显示了正弦波的定义，包含幅度、相位和频率。

先不考虑相位，那么一个正弦波的公式为

$$s(t) = \sin(2\pi f t)$$

其中 f 是正弦波的频率。也就是 1 秒钟有多少个正弦波的周期。如果 $f = 2Hz$ 的话，也就是 1 秒钟之内有 2 个正弦波周期。那么当时间参数 t 从 $0 \rightarrow 1$ 的话，$2\pi f t$ 正好从 $0 \rightarrow 4\pi$，也就是转了 2 圈。说明 1 秒钟转了 2 圈。

所以 $2\pi f$ 又叫做角频率，或者圆频率，表示为 ω 。所以正弦波也可以写成

$$s(t) = \sin(\omega t)$$

如果一个正弦波的角频率是 4π 的话，那么说明 1 秒钟可以转两圈。

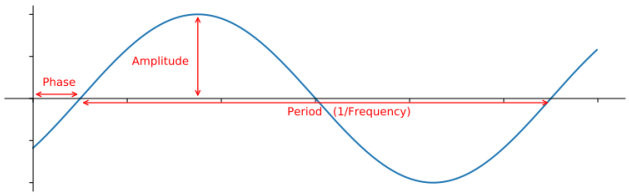

图 5.1: 正弦波信号

5.2 时域和频域

图 5.2 左侧是时域的图形，右侧是相同信号频域的图形。

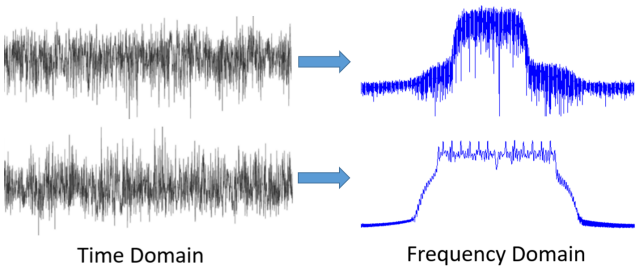

图 5.2: 时域和频域

5.3 周期信号是一系列正弦波的叠加

任意周期信号都可以分解成为不同频率的正弦波的叠加。那么为什么要将周期信号表示为正弦波的叠加呢？为什么不把周期信号表示为方波的叠加呢？

其实最主要的原因是物理世界中存在很多正弦波，例如电磁波，声波，水波等等。例如电容的充放电的过程也是正弦波。

图 5.3 显示了频率为 1 和频率为 2 的正弦波相加形成的波形。

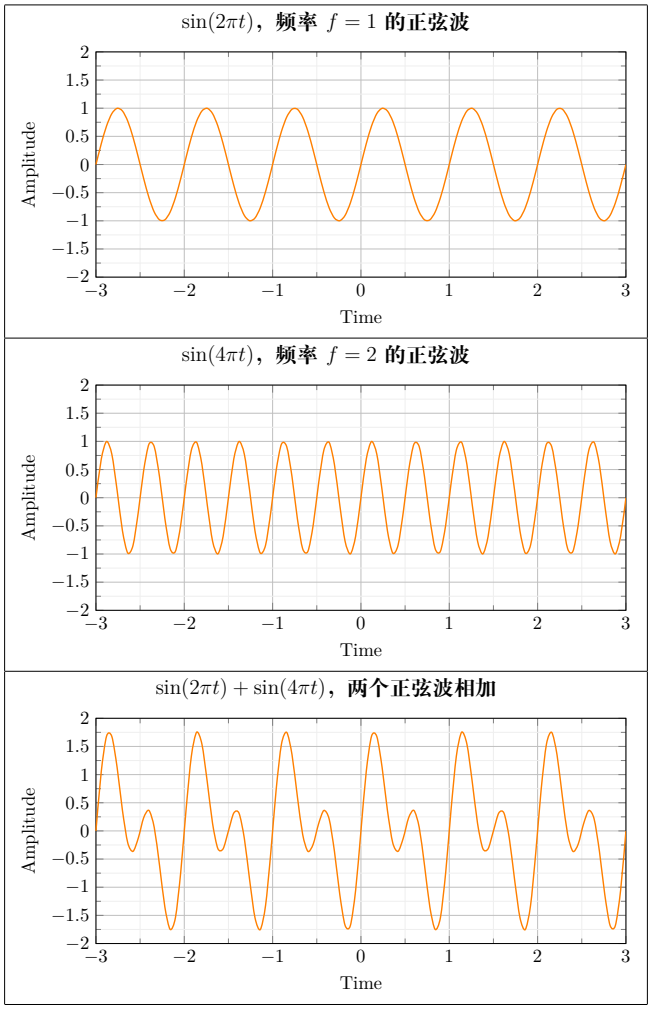

图 5.3: 频率为 1 和 2 的正弦波相加

图 5.4 显示了三角波是如何由不同频率的正弦波叠加而成的。

我们再来看一下方波信号的叠加，方波信号是由频率为 1 的正弦波到频率为无穷大的正弦波叠加而成的。图 5.5 显示了这一过程。

那么，知道了一个周期信号的函数是 $s(t) = s(t+T)$，如何知道这个信号是由哪些正弦波叠加而成的呢？

这就要引入大名鼎鼎的“傅里叶级数”了。

5.4 傅里叶级数

傅里叶级数得名于法国数学家约瑟夫 · 傅里叶（1768 年–1830 年），他提出任何函数都可以展开为三角级数。此前数学家欧拉、达朗贝尔和克莱罗，已发现在认定一个函数有三角级数展开后，通过积分方法计算其系数的公式，而拉格朗日等人已经找到了一些非周期函数的三角级数展开。将周期函数分解为简单振荡函数的总和的最早想法，可以追溯至公元前 3 世纪古代天文学家的均轮和本轮学说。

傅里叶的工作得到了丹尼尔 · 伯努利的赞助，傅里叶介入三角级数用来解热传导方程，其最初论文虽经西尔维斯特 · 拉克鲁瓦、加斯帕尔 · 蒙日同意，但在 1807 年经拉格朗日、拉普拉斯和勒让德评审后被拒绝出版，他的现在被称为傅里叶逆转定理的理论后来发表于 1822 年出版的《热的解析理论》。

周期函数 $s(t) = s(t+T)$ 的傅里叶级数为：

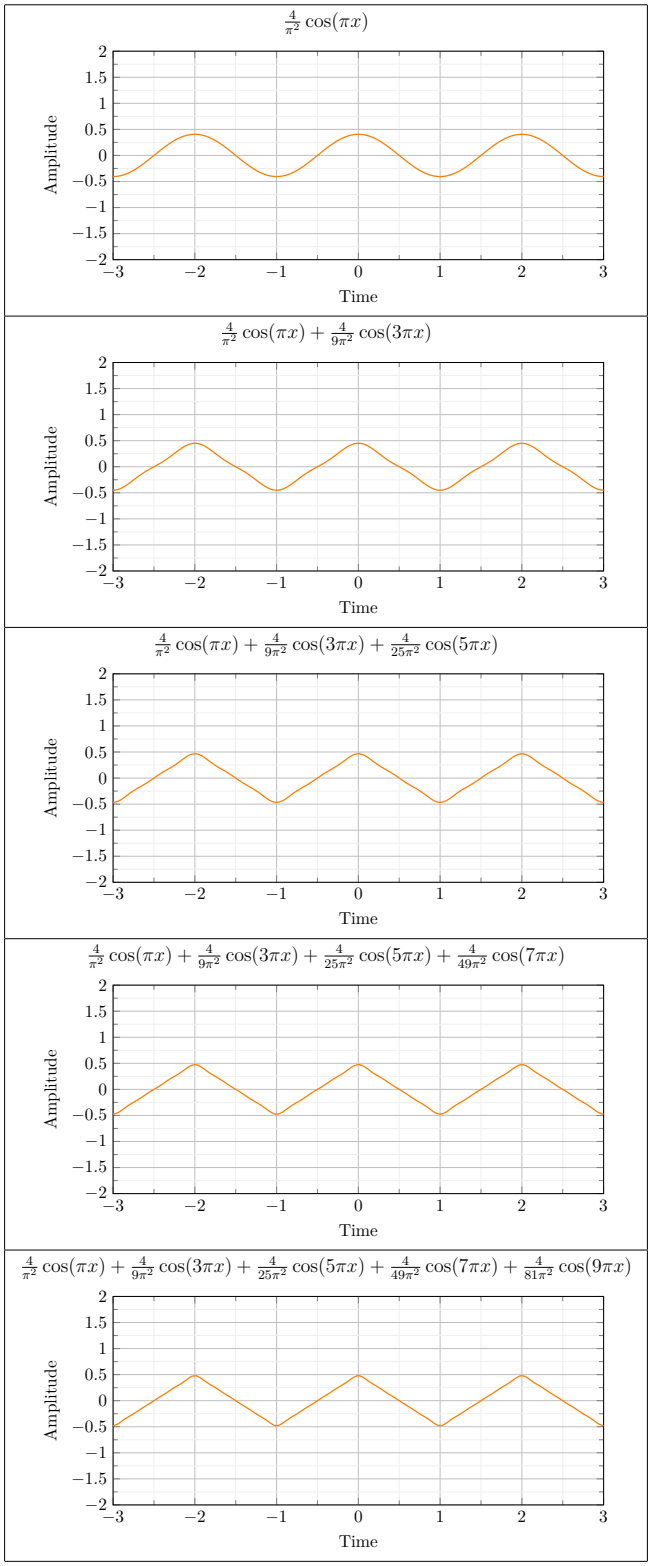

图 5.4: 三角波的叠加过程

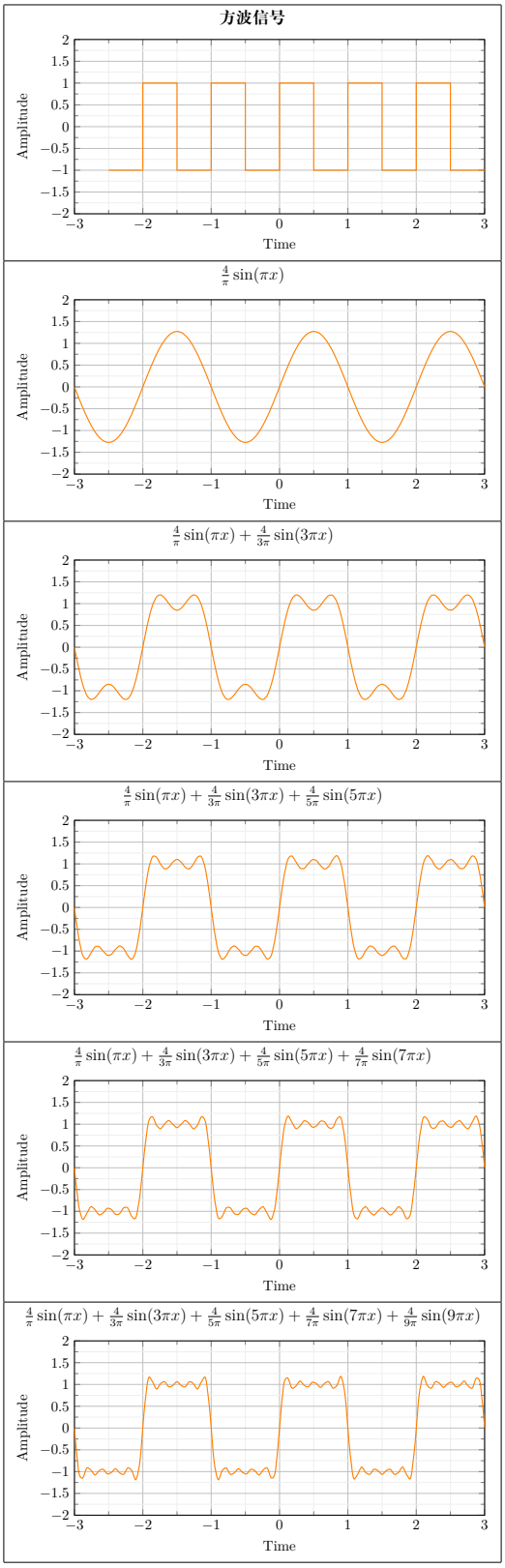
方波信号
Amplitude
Time
$\frac{4}{\pi}\sin(\pi x)$
$\frac{4}{\pi}\sin(\pi x)+\frac{4}{3\pi}\sin(3\pi x)$
$\frac{4}{\pi}\sin(\pi x)+\frac{4}{3\pi}\sin(3\pi x)+\frac{4}{5\pi}\sin(5\pi x)$
$\frac{4}{\pi}\sin(\pi x)+\frac{4}{3\pi}\sin(3\pi x)+\frac{4}{5\pi}\sin(5\pi x)+\frac{4}{7\pi}\sin(7\pi x)$
$\frac{4}{\pi}\sin(\pi x)+\frac{4}{3\pi}\sin(3\pi x)+\frac{4}{5\pi}\sin(5\pi x)+\frac{4}{7\pi}\sin(7\pi x)+\frac{4}{9\pi}\sin(9\pi x)$

图 5.5: 方波的叠加过程

$$s(t) = A_0 + \sum_{n=1}^{\infty}(A_n \cos(\frac{2\pi nx}{T}) + B_n \sin(\frac{2\pi nx}{T}))$$

其中，

$$A_0 = \frac{1}{T}\int_T s(t)dt$$
$$A_n = \frac{2}{T}\int_T s(t)\cos(\frac{2\pi nx}{T})dt$$
$$B_n = \frac{2}{T}\int_T s(t)\sin(\frac{2\pi nx}{T})dt$$

我们以方波信号为例子，方波信号的函数表达式如下：

$$s(t) = \sum_{n=-\infty}^{\infty} s_{one}(t - nT)$$

其中，

$$s_{one}(x) = \begin{cases} 1, & 0 < x < \frac{T}{2} \\ -1, & \frac{T}{2} < x < T \\ 0, & \text{其它情况} \end{cases}$$

那么计算出来的傅里叶级数是：

$$s(t) = \sum_{n=1,3,5,\ldots}^{\infty} \frac{4}{n\pi}\sin(\frac{2\pi nt}{T})$$

我们来可视化一下，如图 5.6 所示。

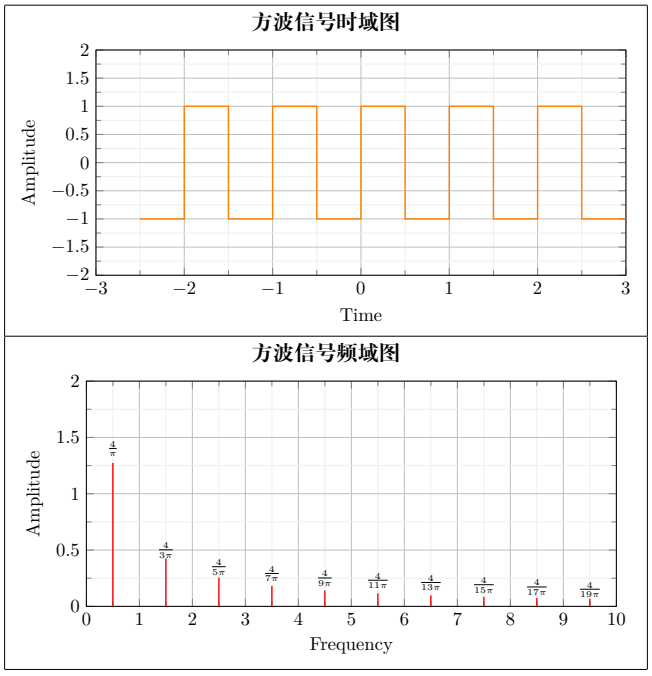

图 5.6: 方波的时域和频域

5.5 时域和频域举例

图 5.7 → 5.10 给出了几种常见信号的时域和频域的对比图。

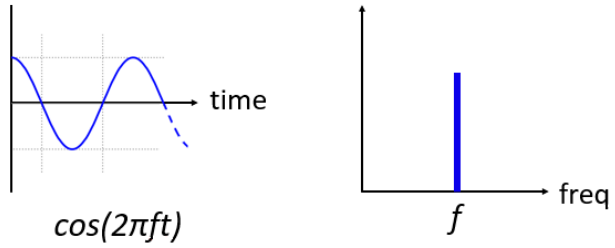

图 5.7: 正弦波

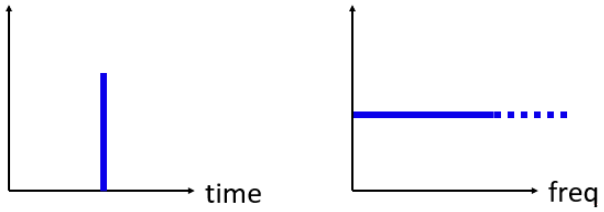

图 5.8: 脉冲信号

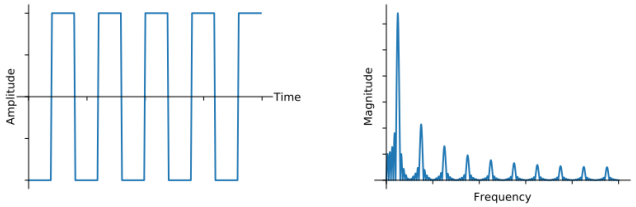

图 5.9: 方波信号的时域和频域

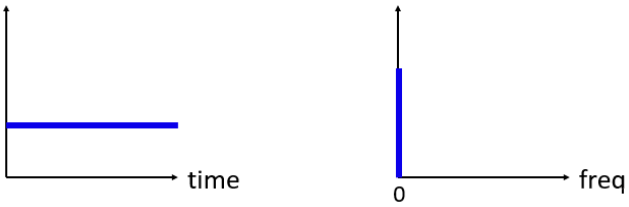

图 5.10: 常量信号

5.6 傅里叶变换

上面我们看了时域和频域的对比图，那么我们有了时域的信号，如何求出相同信号的频域呢？这就来到了傅里叶变换。

同一个信号的时域表达式是：$x(t)$，频域表达式为：$X(f)$。

傅里叶变换为：

$$X(f) = \int x(t)e^{-j2\pi ft}dt$$

逆变换为：

$$x(t) = \frac{1}{2\pi}\int X(f)e^{2\pi ft}df$$

信号的频率为 f ，例如 $f = 1$ ，表示正弦波的周期是 1 秒钟。也就是 1 秒钟一个正弦波周期。f 的单位是 Hz。

还有一个概念叫做角频率（也叫圆频率），定义为

$$\omega = 2\pi f$$

角频率描述的是每一秒钟信号转动的角度，例如 $f = 1$，那么表示角频率是 2π，也就是每秒钟转动一圈。

我们在公式中为了简便，经常使用 ω 代替 $2\pi f$ 。

以上是连续信号的傅里叶变换，但连续信号无法被计算机处理，所以我们可以转换成离散傅里叶变换的形式。

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{j2\pi}{N}kn}$$

k 的范围是 $0 \le k \le N-1$。

5.7 滤波

有了时域和频域的转换，我们就可以进行滤波了。也就是将指定的频率范围的正弦波信号过滤掉。可以看到将图 5.11 中的上下两个频域函数相乘，就得到了滤波以后的信号的 `频域函数` 。

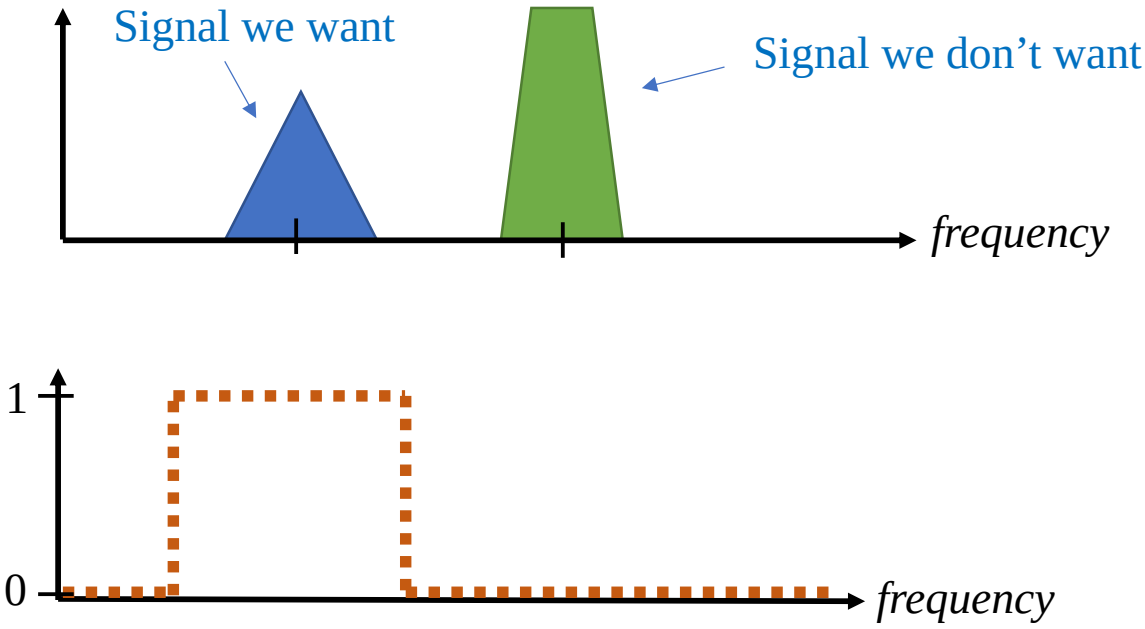

图 5.11: 频域相乘和时域相乘的对应关系

那么两个频域函数 $X(f)$ 和 $Y(f)$ 相乘，也就是 $X(f)Y(f)$ 该如何转换回时域呢？因为我们得针对时域函数进行滤波。

关系如图 5.12 。

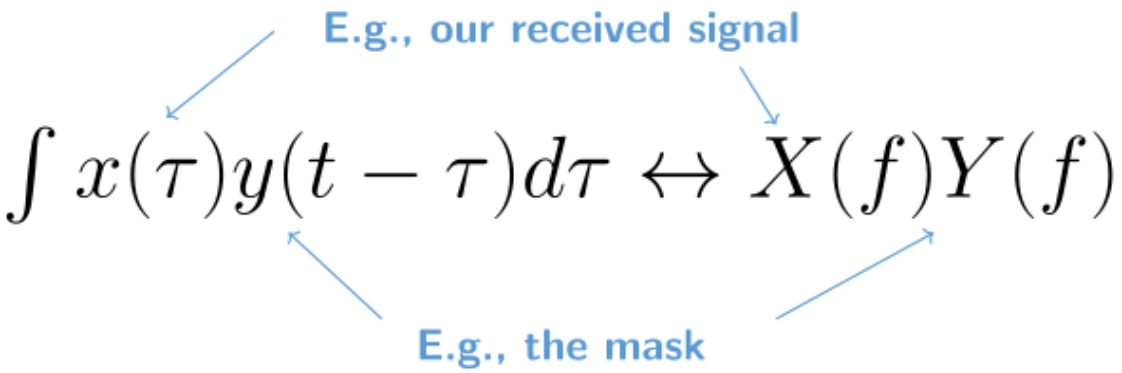

图 5.12: 频域相乘和时域相乘的对应关系

5.8 采样定理

定理 5.1 (采样定理)

想要完整的还原信号，采样频率 f_s 必须大于信号的成分中最高频率的 2 倍。

我们来举例说明，对于一个正弦波，当每个周期只采样一个点时，那么只能得到直线。如图 5.13 所示。

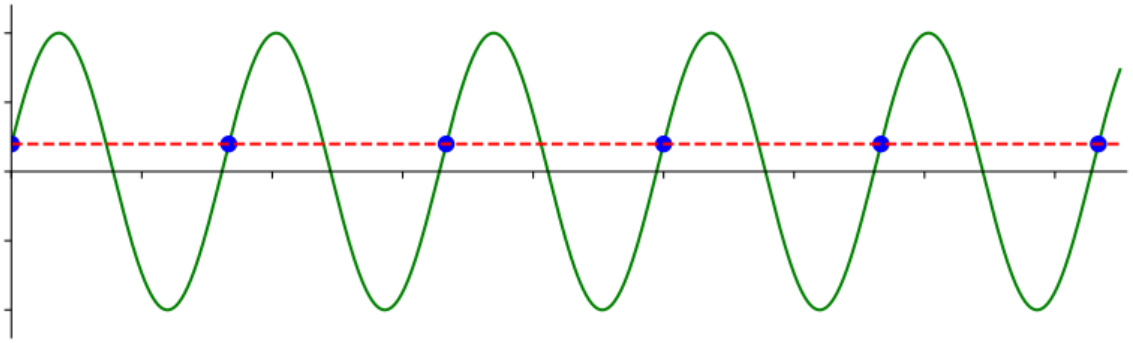

图 5.13: 每个周期采样 1 个点

每个周期采样 1.2 个点，还是无法还原原始信号。如图 5.14 所示。

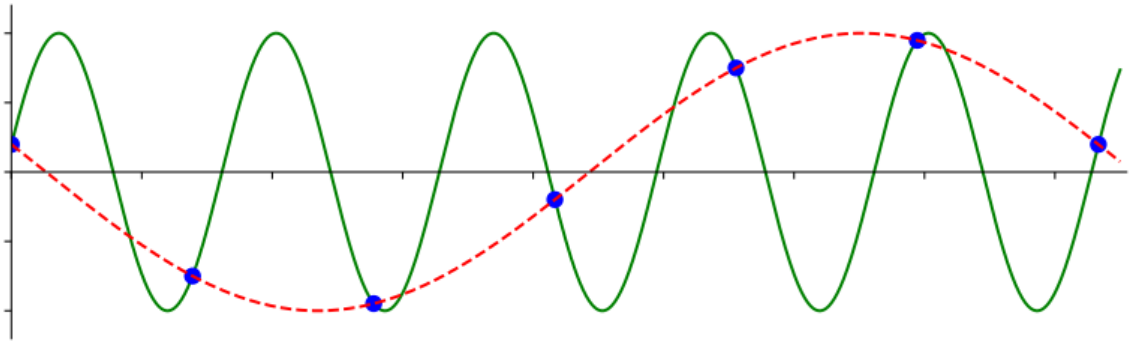

图 5.14: 每个周期采样 1.2 个点

继续加大采样率，每个周期采样 1.5 个点。也同样无法还原原始信号。如图所示 5.15 。
继续加大采样率，我们发现每个周期采样 2 个点。就可以还原原始信号了。如图所示 5.16 。
所以想要还原任意一个原始信号，采样率必须大于原始信号的最高频率成分的 2 倍才能做到这一点。

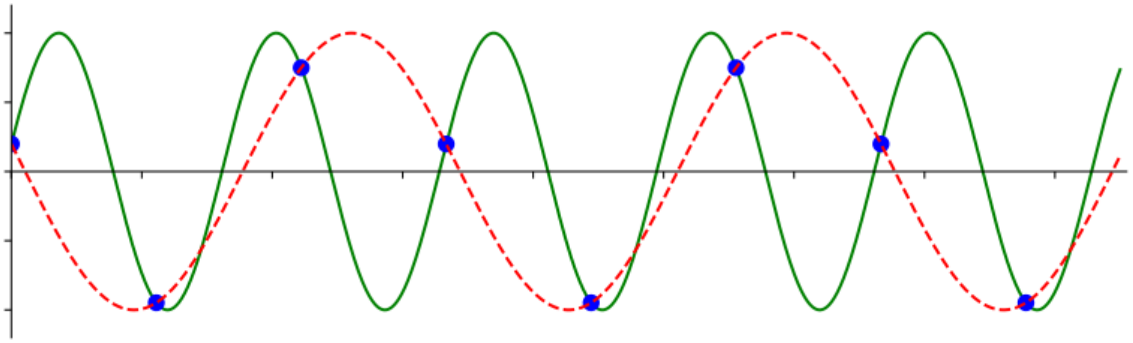

图 5.15: 每个周期采样 1.5 个点

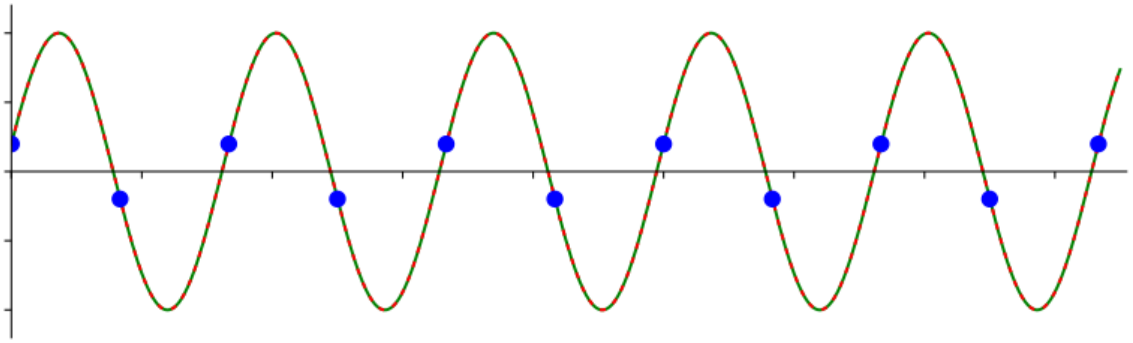

图 5.16: 每个周期采样 2 个点

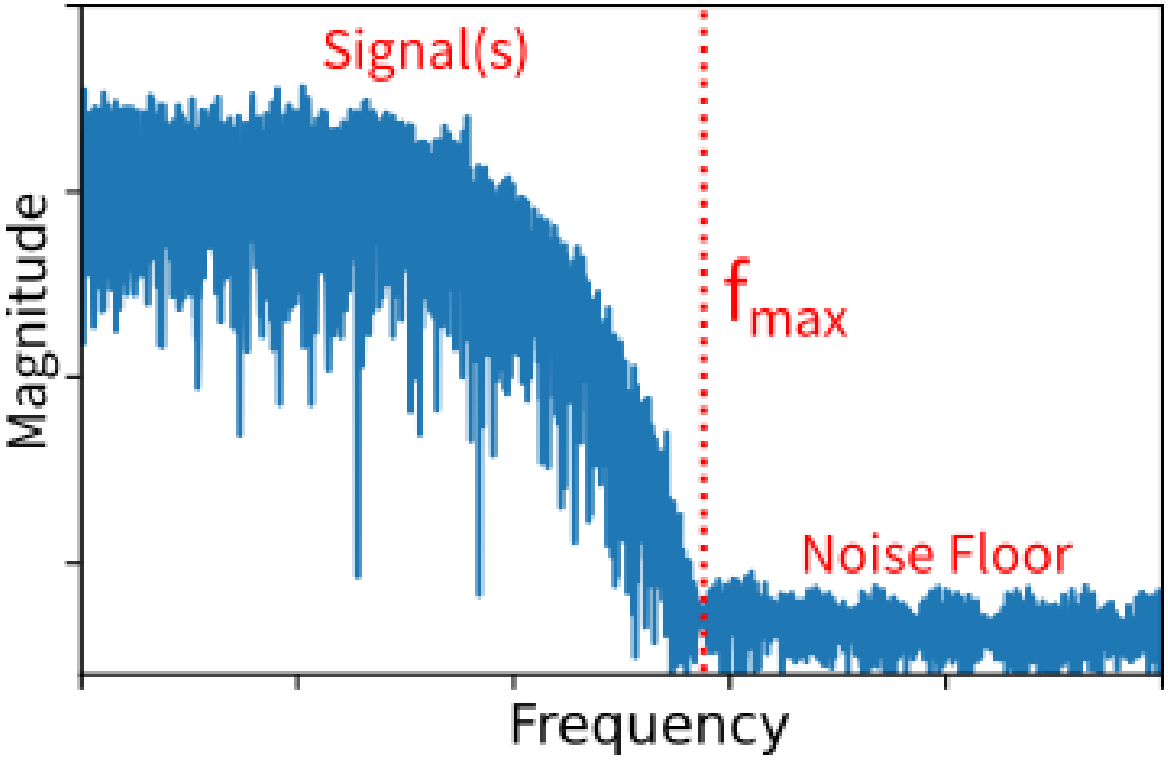

图 5.17: 采样定理

! 所以我们永远无法彻底还原一个方波信号，因为方波信号中存在无限大频率的成分。而我们的 ADC 是无法达到这个采样频率的。
我们在做心电信号采集时，由于人的心跳频率不会高于 $200Hz$ 。所以只要 ADC 的采样频率高于 $400Hz$ 就可以看出心跳的波形了。

5.9 数字调制

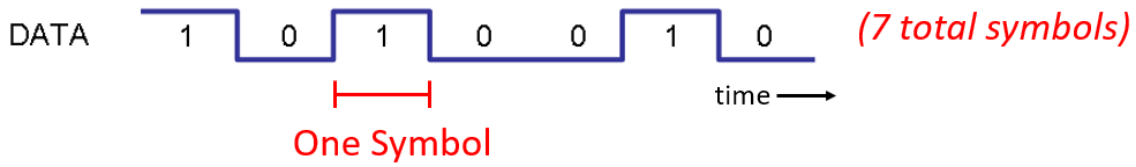

图 5.18: 数字信号

以太网传输的信号如图 5.19 。

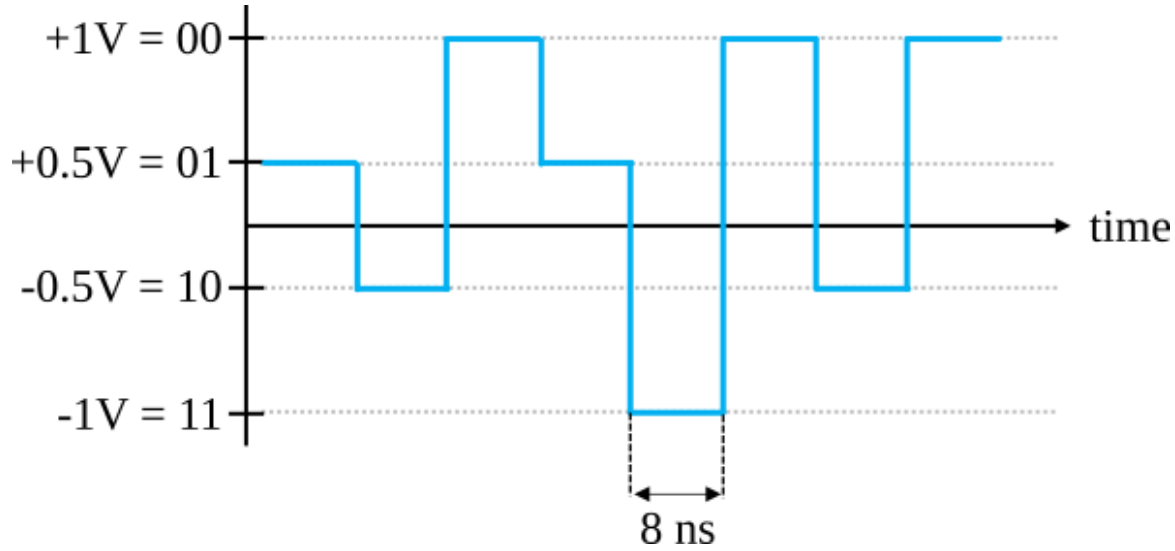

图 5.19: 以太网中的信号

1. 在图 5.19 中，每秒钟能够传输多少个 bit 呢？（bits per second, bps）
答：$\frac{1}{8\times10^{-9}}\times 2 = 250Mbps$。
2. 需要多少根线才能达到 1Gbps?
答：4 根线。这也是以太网线缆采用的方案。
3. 如果某个调制方案可以有 16 个不同的电平值，那么每个符号可以表示多少个 bit 呢?
答：4 个 bit。
4. 16 个不同的电平值，每个符号持续时间是 8ns，那么带宽是多少 bps?
答：$\frac{1}{8\times10^{-9}}\times 4 = 0.5Gbps$。

5.9.1 数字调幅

ASK，Amplitude Shift Keying。幅移键控。如果用 2 个电平调制信号，那么叫做 2-ASK，如图 5.20 。如果用 4 个不同的电平调制信号，那么叫做 4-ASK，如图 5.21 。对于 4-ASK，每个符号可以表示 2 个 bit。

5.9.2 数字相位调制

我们使用相位调制数字信号，相位没变化表示 1 ，相位偏移 180 度表示 0 。

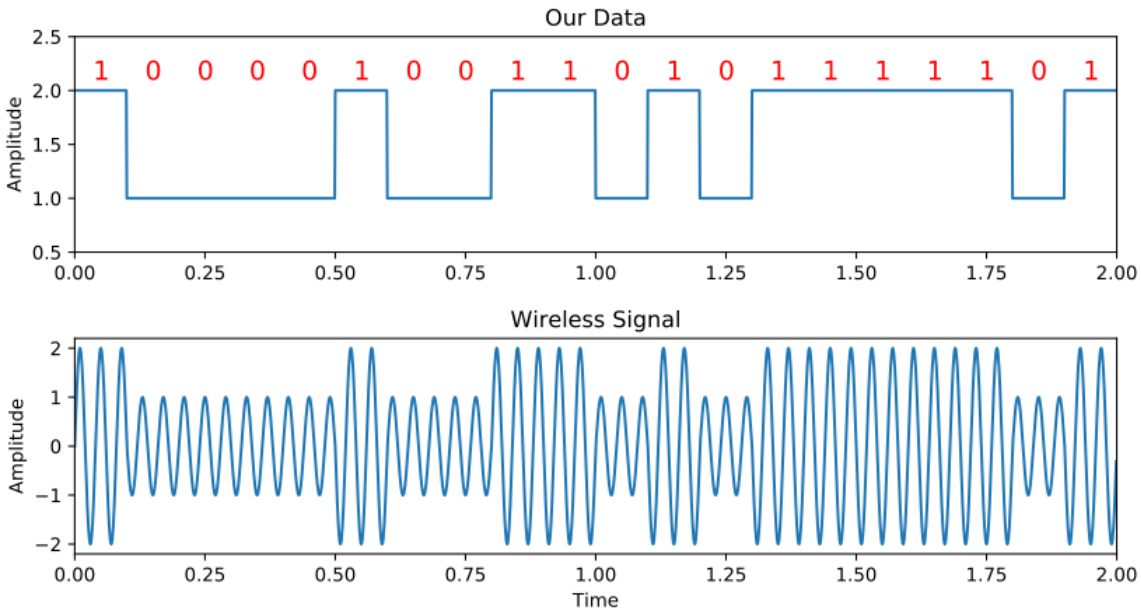

图 5.20: 2-ASK 调制

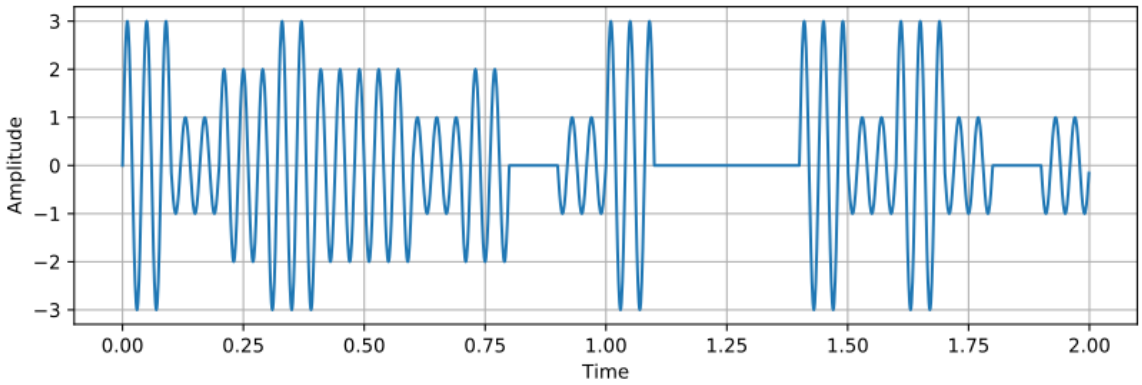

图 5.21: 4-ASK 调制

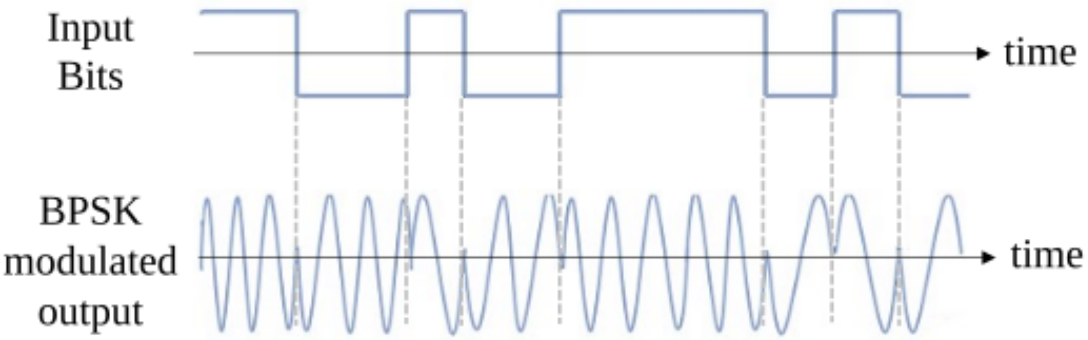

图 5.22: BPSK 调制

5.9.3 数字频率调制

5.10 模拟调制

5.10.1 模拟调幅

5.10.2 模拟调频

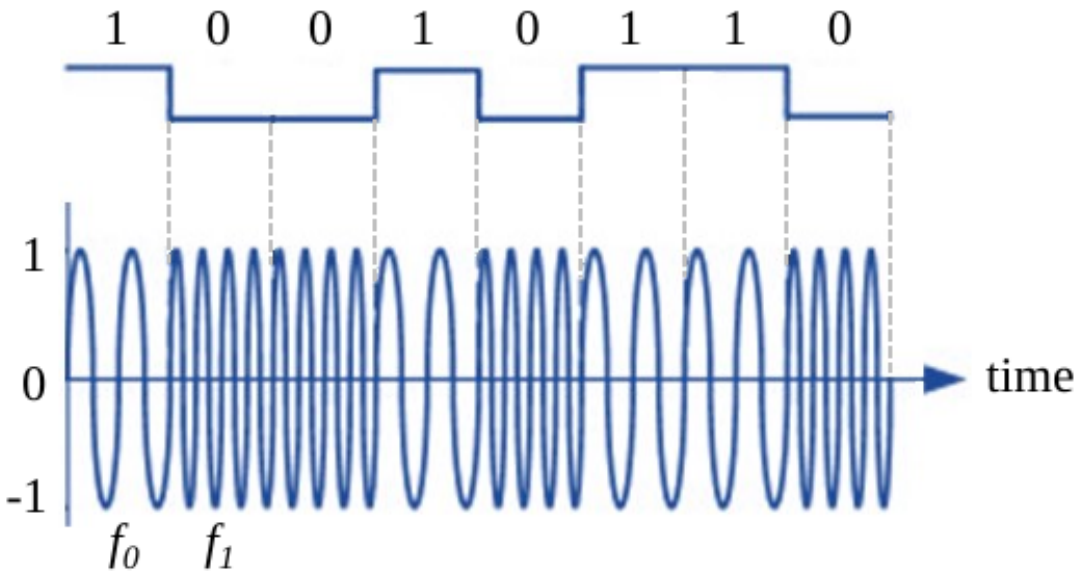

图 5.23: FSK 调制

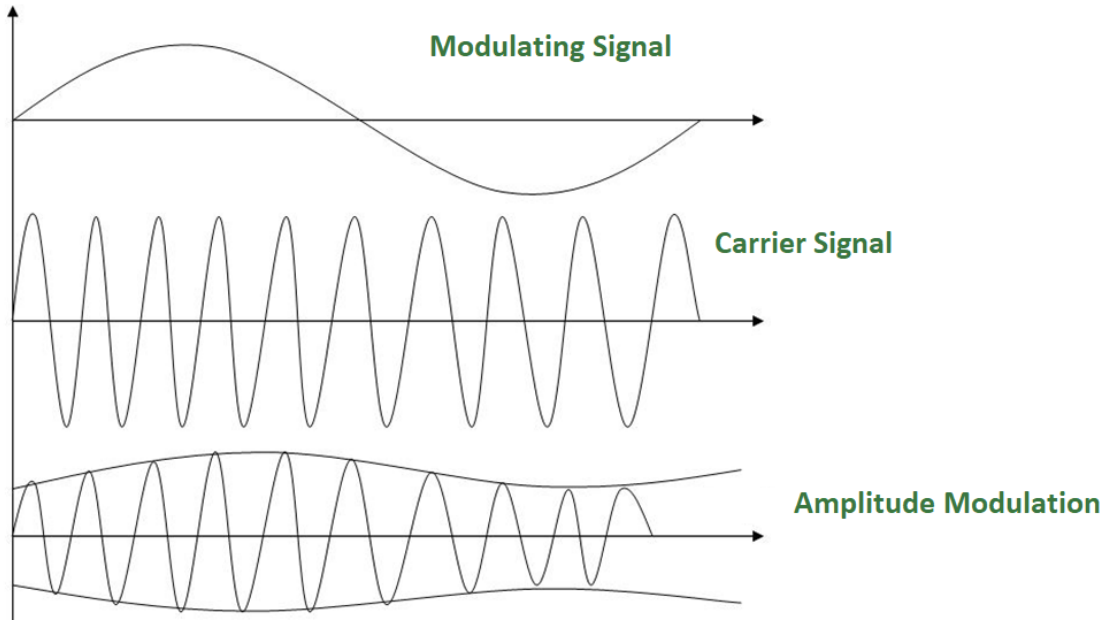

图 5.24: AM 调制

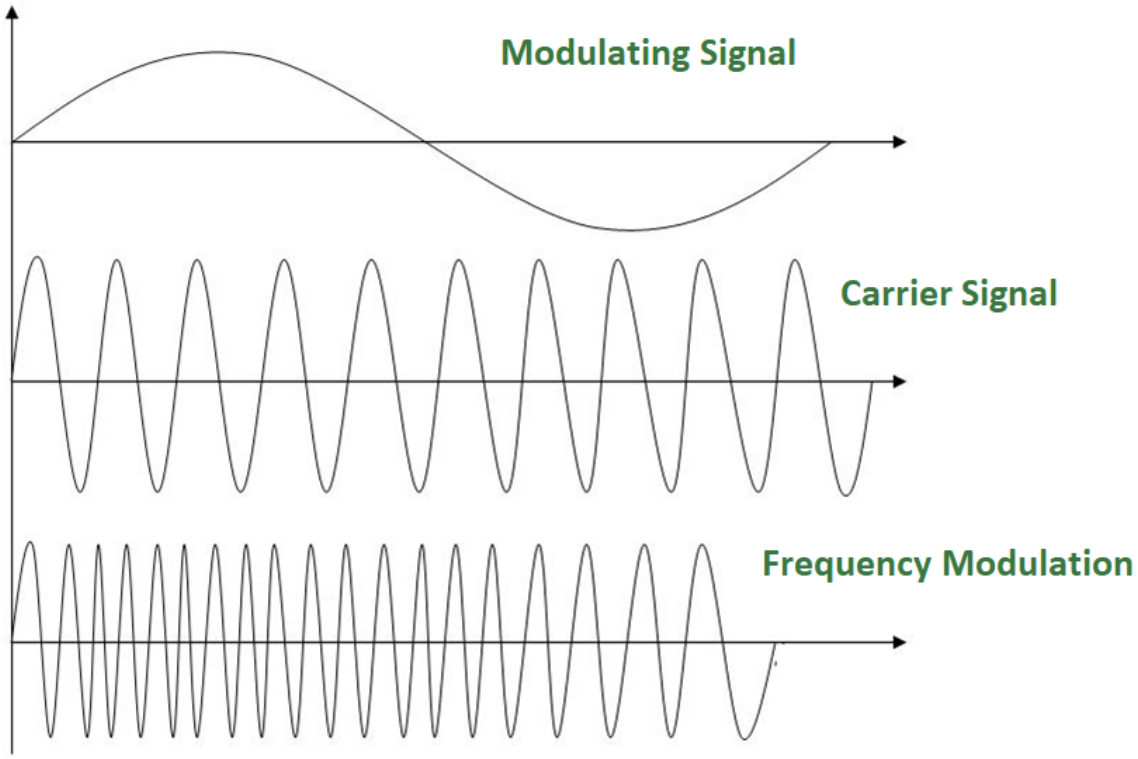

图 5.25: FM 调制

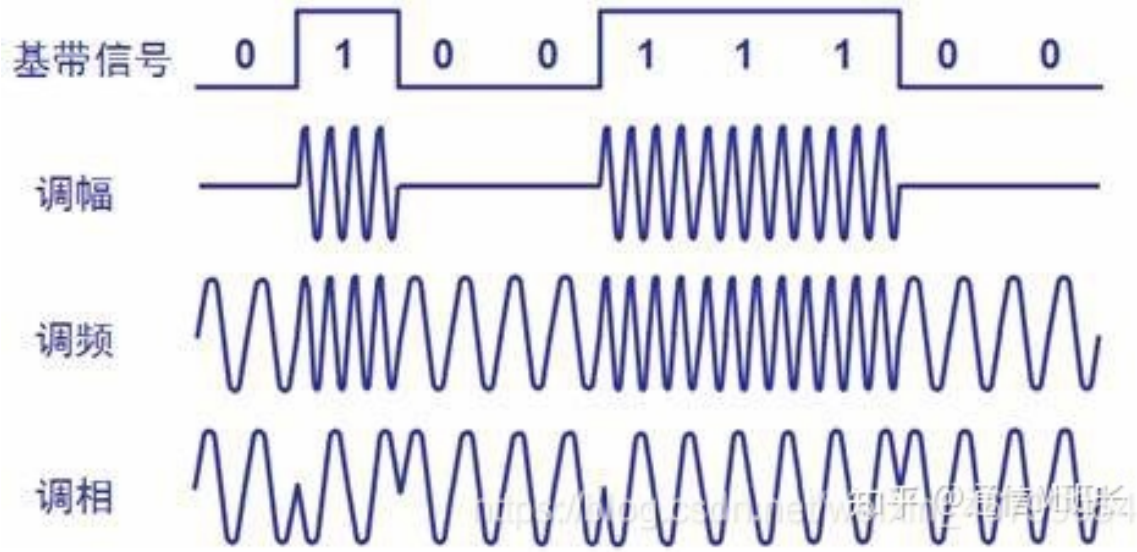

图 5.26: 数字调制

第六章 红外遥控（RMT）

LED 模块的参考代码位于 `examples/peripherals/rmt/led_strip` 文件夹。

6.1 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

6.2 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。

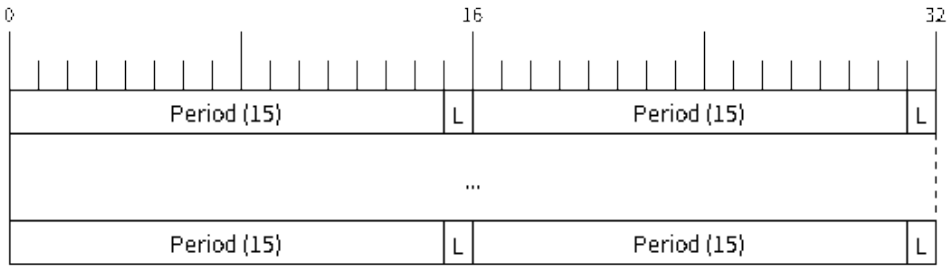

图 6.1: RMT 符号结构 (L-信号电平)

6.3 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：

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。

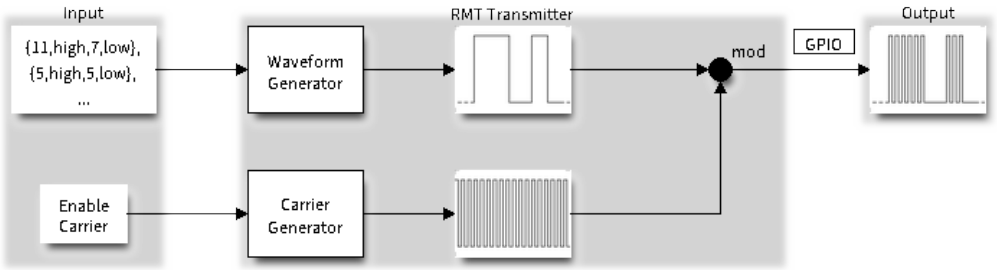

图 6.2: RMT 发射器概述

6.4 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：

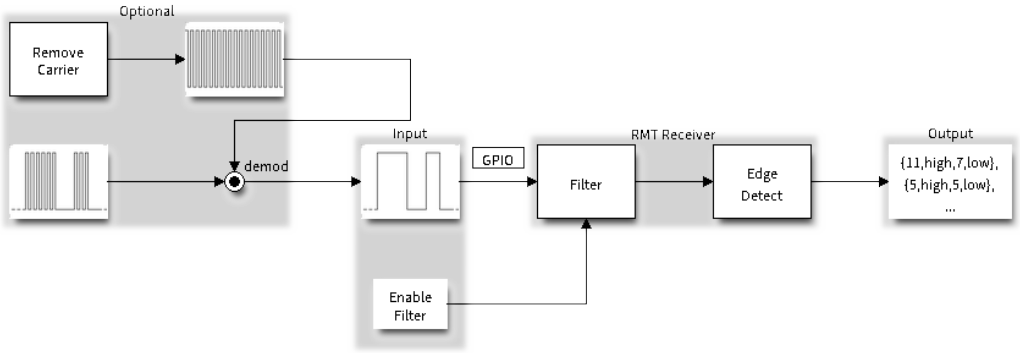

图 6.3: RMT 接收器概述

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。

6.5 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：

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。

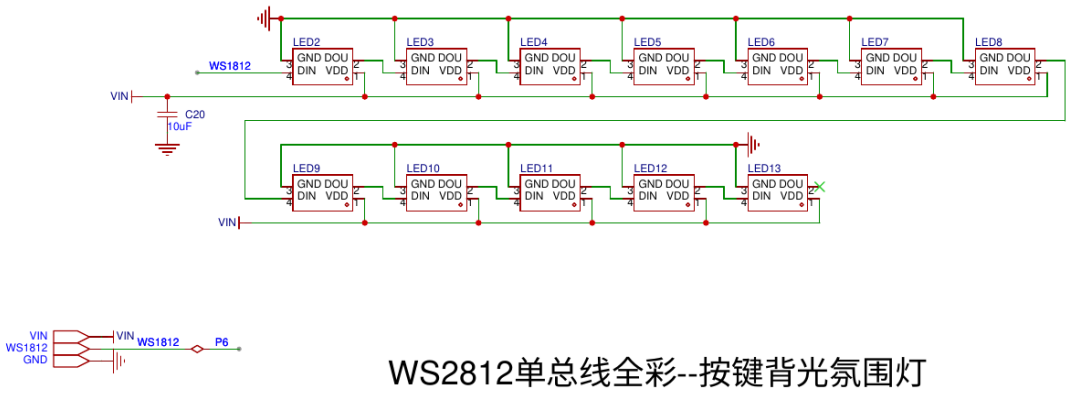

图 6.4: LED 灯原理图

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` 。

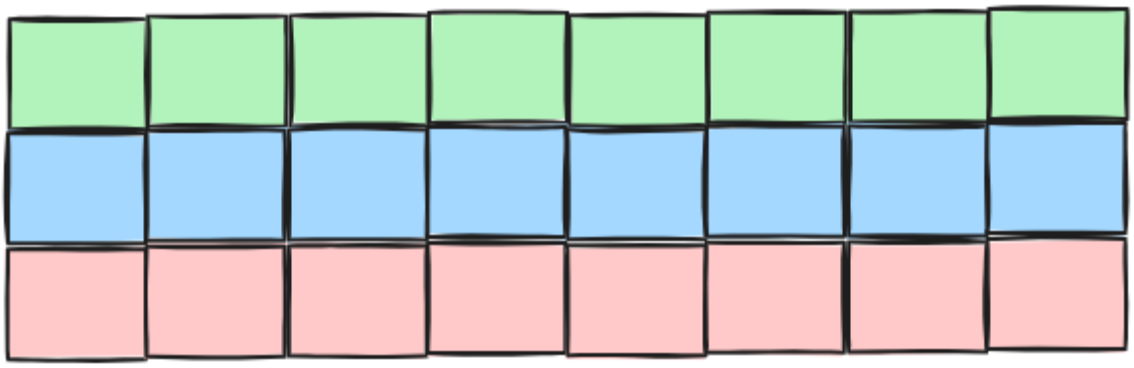

图 6.5: 发送颜色的顺序

6.6 代码实现

首先我们看一下文件夹的目录结构。

```
atguigu-led-example
├─ main
│   ├─ CMakeLists.txt
│   └─ led_main.c
└─ CMakeLists.txt
```

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

atguigu-led-example/CMakeLists.txt

```
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：

atguigu-led-example/main/CMakeLists.txt

```
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。

atguigu-led-example/main/led_main.c

```
#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。

atguigu-led-example/main/led_main.c

```
// 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。

atguigu-led-example/main/led_main.c

```
// 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。

atguigu-led-example/main/led_main.c

```
// 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。

atguigu-led-example/main/led_main.c

```
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。

atguigu-led-example/main/led_main.c

```
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。

atguigu-led-example/main/led_main.c

```
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。

atguigu-led-example/main/led_main.c

```
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。

atguigu-led-example/main/led_main.c

```
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:
```

```
41          *r = rgb_max;
42          *g = rgb_min;
43          *b = rgb_max - rgb_adj;
44          break;
45      }
46  }
```

然后编写 RMT 初始化的代码。

atguigu-led-example/main/led_main.c

```
1   void RMT_Init(void)
2   {
3       rmt_tx_channel_config_t tx_chan_config = {
4           .clk_src = RMT_CLK_SRC_DEFAULT, // 选择时钟源
5           .gpio_num = RMT_LED_STRIP_GPIO_NUM, // GPIO引脚设置
6           .mem_block_symbols = 64, // increase the block size can make the LED
                less flickering
7           .resolution_hz = RMT_LED_STRIP_RESOLUTION_HZ,
8           .trans_queue_depth = 4, // set the number of transactions that can be
                pending in the background
9       };
10      ESP_ERROR_CHECK(rmt_new_tx_channel(&tx_chan_config, &led_chan));
11
12      led_strip_encoder_config_t encoder_config = {
13          .resolution = RMT_LED_STRIP_RESOLUTION_HZ,
14      };
15      ESP_ERROR_CHECK(rmt_new_led_strip_encoder(&encoder_config, &led_encoder));
16
17      ESP_ERROR_CHECK(rmt_enable(led_chan));
18  }
```

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。

atguigu-led-example/main/led_main.c

```
1   void light_led(uint8_t led_num)
2   {
3       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。

atguigu-led-example/main/led_main.c

```
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);
```

```
12          Delay_ms(1000);
13      }
14  }
```

然后编译并烧写程序。

编译并烧写程序

```
1  cd ~/esp/atguigu-led-example
2  idf.py set-target esp32c3
3  idf.py flash
```

第七章 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 。

图 7.1: 语音模块电路图

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。

audio_main.c

```c
#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;
```

```
25      DELAY_MS(10);  ⟶ 这个延时比较重要
26      B_DATA = S_DATA & 0X01;
27      for (j = 0; j < 8; j++)
28      {
29          if (B_DATA == 1)
30          {
31              AUDIO_SDA_H;
32              DELAY_US(600);
33              AUDIO_SDA_L;
34              DELAY_US(200);
35          }
36          else
37          {
38              AUDIO_SDA_H;
39              DELAY_US(200);
40              AUDIO_SDA_L;
41              DELAY_US(600);
42          }
43          S_DATA = S_DATA >> 1;
44          B_DATA = S_DATA & 0X01;
45      }
46      AUDIO_SDA_H;
47      DELAY_MS(2);
48  }
49
50  void AUDIO_Test(void)
51  {
52      Line_1A_WT588F(85);
53      DELAY_MS(2000);
54      Line_1A_WT588F(86);
55      DELAY_MS(2000);
56      Line_1A_WT588F(69);
57      DELAY_MS(2000);
58  }
59
60  void AUDIO_Init(void)
61  {
62      gpio_config_t io_conf = {};
63      io_conf.intr_type = GPIO_INTR_DISABLE;
64      io_conf.mode = GPIO_MODE_OUTPUT;
65      io_conf.pin_bit_mask = (1ULL << AUDIO_SDA_PIN);
```

```
66      gpio_config(&io_conf);
67
68      io_conf.intr_type = GPIO_INTR_DISABLE;
69      io_conf.mode = GPIO_MODE_INPUT;
70      io_conf.pin_bit_mask = (1ULL << AUDIO_BUSY_PIN);
71      gpio_config(&io_conf);
72  }
73
74  void app_main(void)
75  {
76      AUDIO_Init();
77      AUDIO_Test();
78  }
```

第八章 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：

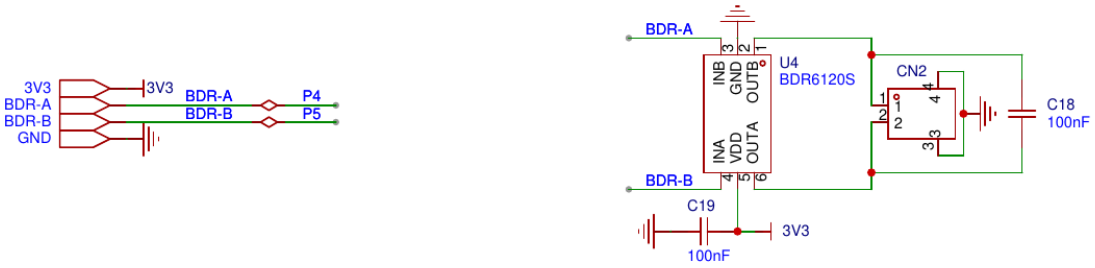

图 8.1: 电机模块电路图

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。

电机 *GPIO* 引脚初始化

```
#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

控制电机转动来开关锁

```c
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);
}
```

第九章 串口通信

ESP32 使用串口和指纹模块进行通信。电路图如下：

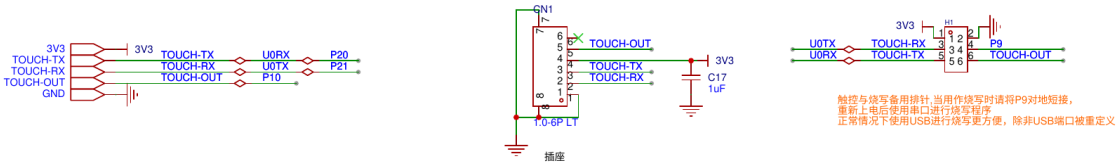

图 9.1: 指纹模块电路图

9.1 串口收发简单示例

我们先来写一个简单的串口示例程序。

串口收发程序

```c
#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);
```

```
26      uart_param_config(UART_NUM_1, &uart_config);
27      uart_set_pin(UART_NUM_1, TXD_PIN, RXD_PIN, UART_PIN_NO_CHANGE,
            UART_PIN_NO_CHANGE);
28  }
29
30  int sendData(const char *logName, const char *data)
31  {
32      const int len = strlen(data);
33      const int txBytes = uart_write_bytes(UART_NUM_1, data, len);
34      ESP_LOGI(logName, "Wrote %d bytes", txBytes);
35      return txBytes;
36  }
37
38  static void tx_task(void *arg)
39  {
40      static const char *TX_TASK_TAG = "TX_TASK";
41      esp_log_level_set(TX_TASK_TAG, ESP_LOG_INFO);
42      while (1)
43      {
44          sendData(TX_TASK_TAG, "Hello world");
45          vTaskDelay(2000 / portTICK_PERIOD_MS);
46      }
47  }
48
49  static void rx_task(void *arg)
50  {
51      static const char *RX_TASK_TAG = "RX_TASK";
52      esp_log_level_set(RX_TASK_TAG, ESP_LOG_INFO);
53      uint8_t *data = (uint8_t *)malloc(RX_BUF_SIZE + 1);
54      while (1)
55      {
56          const int rxBytes = uart_read_bytes(UART_NUM_1, data, RX_BUF_SIZE, 1000
                / portTICK_PERIOD_MS);
57          if (rxBytes > 0)
58          {
59              data[rxBytes] = 0;
60              ESP_LOGI(RX_TASK_TAG, "Read %d bytes: '%s'", rxBytes, data);
61              ESP_LOG_BUFFER_HEXDUMP(RX_TASK_TAG, data, rxBytes, ESP_LOG_INFO);
62          }
63      }
64      free(data);
```

```
65 }
66
67 void app_main(void)
68 {
69     init();
70     xTaskCreate(rx_task, "uart_rx_task", 1024 * 2, NULL, configMAX_PRIORITIES,
           NULL);
71     xTaskCreate(tx_task, "uart_tx_task", 1024 * 2, NULL, configMAX_PRIORITIES -
            1, NULL);
72 }
```

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。

9.2 指纹模块

我们先来写头文件

指纹模块头文件

```
1  #ifndef __FINGERPRINT_DRIVER_H_
2  #define __FINGERPRINT_DRIVER_H_
3
4  #include "driver/uart.h"
5  #include "driver/gpio.h"
6
7  /// 下面的配置可以直接写死，也可以在 menuconfig 里面配置
8  #define TXD_PIN (GPIO_NUM_21)
9  #define RXD_PIN (GPIO_NUM_20)
10
11 #define BUF_SIZE (1024)
12
13 #define TOUCH_INT GPIO_NUM_10
14
15 /// 初始化指纹模块
16 void FINGERPRINT_Init(void);
17
18 /// 获取指纹芯片的序列号
19 void get_chip_sn(void);
20
21 /// 获取指纹图像
```

```
22 int get_image(void);
23
24 /// 获取指纹特征
25 int gen_char(void);
26
27 /// 搜索指纹
28 int search(void);
29
30 /// 读取指纹芯片配置参数
31 void read_sys_params(void);
32
33 #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 = 57600,
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         UART_NUM_1,
19         BUF_SIZE * 2, 0, 0, NULL,
20         intr_alloc_flags));
21     ESP_ERROR_CHECK(uart_param_config(
22         UART_NUM_1, &uart_config));
23     ESP_ERROR_CHECK(uart_set_pin(
```

```
24          UART_NUM_1,
25          TXD_PIN,
26          RXD_PIN,
27          UART_PIN_NO_CHANGE,
28          UART_PIN_NO_CHANGE));
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(UART_NUM_1, (const char *)PS_GetChipSN, 13);
49
50      // Read data from the UART
51      int len = uart_read_bytes(UART_NUM_1, 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(UART_NUM_1, (const char *)PS_GetImageBuffer, 12);
73
74     int len = uart_read_bytes(UART_NUM_1, 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             }
94         }
95     }
96
97     free(data);
98
99     return result;
100 }
```

```
101
102 int gen_char(void)
103 {
104     uint8_t *data = (uint8_t *)malloc(BUF_SIZE);
105
106     // 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。
107     uint8_t PS_GenCharBuffer[13] = {0xEF, 0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0x01, 0
        x00, 0x04, 0x02, 0x01, 0x00, 0x08};
108
109     uart_write_bytes(UART_NUM_1, (const char *)PS_GenCharBuffer, 13);
110
111     int len = uart_read_bytes(UART_NUM_1, data, (BUF_SIZE - 1), 2000 /
        portTICK_PERIOD_MS);
112
113     int result = 0xFF;
114
115     if (len)
116     {
117         if (data[6] == 0x07)
118         {
119             result = data[9];
120         }
121     }
122
123     free(data);
124
125     return result;
126 }
127
128 int search(void)
129 {
130     uint8_t *data = (uint8_t *)malloc(BUF_SIZE);
131
132     // 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。
133     uint8_t PS_SearchBuffer[17] = {0xEF, 0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0x01, 0
        x00, 0x08, 0x04, 0x01, 0x00, 0x00, 0xFF, 0xFF, 0x02, 0x0C};
134
135     uart_write_bytes(UART_NUM_1, (const char *)PS_SearchBuffer, 17);
136
137     int len = uart_read_bytes(UART_NUM_1, data, (BUF_SIZE - 1), 2000 /
```

```
        portTICK_PERIOD_MS);
138
139     int result = 0xFF;
140
141     if (len)
142     {
143         if (data[6] == 0x07)
144         {
145             result = data[9];
146         }
147     }
148
149     free(data);
150
151     return result;
152 }
153
154 void read_sys_params(void)
155 {
156     uint8_t *data = (uint8_t *)malloc(BUF_SIZE);
157
158     // 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。
159     uint8_t PS_ReadSysPara[12] = {0xEF, 0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0x01, 0
        x00, 0x03, 0x0F, 0x00, 0x13};
160
161     uart_write_bytes(UART_NUM_1, (const char *)PS_ReadSysPara, 12);
162
163     int len = uart_read_bytes(UART_NUM_1, data, (BUF_SIZE - 1), 2000 /
        portTICK_PERIOD_MS);
164
165     if (len)
166     {
167         if (data[6] == 0x07)
168         {
169             if (data[9] == 0x00)
170             {
171                 int register_count = (data[10] << 8) | data[11];
172                 printf("register count ==> %d\r\n", register_count);
173                 int fingerprint_template_size = (data[12] << 8) | data[13];
174                 printf("finger print template size ==> %d\r\n",
```

```
                fingerprint_template_size);
175             int fingerprint_library_size = (data[14] << 8) | data[15];
176             printf("finger print library size ==> %d\r\n",
                fingerprint_library_size);
177             int score_level = (data[16] << 8) | data[17];
178             printf("score level ==> %d\r\n", score_level);
179             // device address
180             printf("device address ==> 0x");
181             for (int i = 0; i < 4; i++)
182             {
183                 printf("%02X ", data[18 + i]);
184             }
185             printf("\r\n");
186             // data packet size
187             int packet_size = (data[22] << 8) | data[23];
188             if (packet_size == 0)
189             {
190                 printf("packet size ==> 32 bytes\r\n");
191             }
192             else if (packet_size == 1)
193             {
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 }
```

第十章 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。

10.1 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。

10.1.1 头文件

蓝牙功能头文件

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
```

```
6  #include "freertos/event_groups.h"
7  #include "esp_system.h"
8  #include "esp_log.h"
9  #include "nvs_flash.h"
10 #include "esp_bt.h"
11 #include "esp_gap_ble_api.h"
12 #include "esp_gatts_api.h"
13 #include "esp_bt_defs.h"
14 #include "esp_bt_main.h"
15 #include "esp_gatt_common_api.h"
16 #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。

10.1.2 入口函数

入口函数是 `app_main()` 函数。

入口函数

```
1  void app_main(void)
2  {
3      esp_err_t ret;
4
```

```
5     // Initialize NVS.
6     ret = nvs_flash_init();
7     if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
          ESP_ERR_NVS_NEW_VERSION_FOUND)
8     {
9         ESP_ERROR_CHECK(nvs_flash_erase());
10        ret = nvs_flash_init();
11    }
12    ESP_ERROR_CHECK(ret);
13
14    ESP_ERROR_CHECK(esp_bt_controller_mem_release(ESP_BT_MODE_CLASSIC_BT));
15
16    esp_bt_controller_config_t bt_cfg = BT_CONTROLLER_INIT_CONFIG_DEFAULT();
17    ret = esp_bt_controller_init(&bt_cfg);
18    if (ret)
19    {
20        ESP_LOGE(GATTS_TAG, "%s initialize controller failed: %s\n", __func__,
              esp_err_to_name(ret));
21        return;
22    }
23
24    ret = esp_bt_controller_enable(ESP_BT_MODE_BLE);
25    if (ret)
26    {
27        ESP_LOGE(GATTS_TAG, "%s enable controller failed: %s\n", __func__,
              esp_err_to_name(ret));
28        return;
29    }
30    ret = esp_bluedroid_init();
31    if (ret)
32    {
33        ESP_LOGE(GATTS_TAG, "%s init bluetooth failed: %s\n", __func__,
              esp_err_to_name(ret));
34        return;
35    }
36    ret = esp_bluedroid_enable();
37    if (ret)
38    {
39        ESP_LOGE(GATTS_TAG, "%s enable bluetooth failed: %s\n", __func__,
              esp_err_to_name(ret));
40        return;
```

```
41      }
42
43      ret = esp_ble_gatts_register_callback(gatts_event_handler);
44      if (ret)
45      {
46          ESP_LOGE(GATTS_TAG, "gatts register error, error code = %x", ret);
47          return;
48      }
49      ret = esp_ble_gap_register_callback(gap_event_handler);
50      if (ret)
51      {
52          ESP_LOGE(GATTS_TAG, "gap register error, error code = %x", ret);
53          return;
54      }
55      ret = esp_ble_gatts_app_register(PROFILE_A_APP_ID);
56      if (ret)
57      {
58          ESP_LOGE(GATTS_TAG, "gatts app register error, error code = %x", ret);
59          return;
60      }
61      ret = esp_ble_gatts_app_register(PROFILE_B_APP_ID);
62      if (ret)
63      {
64          ESP_LOGE(GATTS_TAG, "gatts app register error, error code = %x", ret);
65          return;
66      }
67      esp_err_t local_mtu_ret = esp_ble_gatt_set_local_mtu(500);
68      if (local_mtu_ret)
69      {
70          ESP_LOGE(GATTS_TAG, "set local MTU failed, error code = %x",
                local_mtu_ret);
71      }
72
73      return;
74  }
```

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();
```

10.1.3 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);
```

> ! 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。

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。

10.1.4 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 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

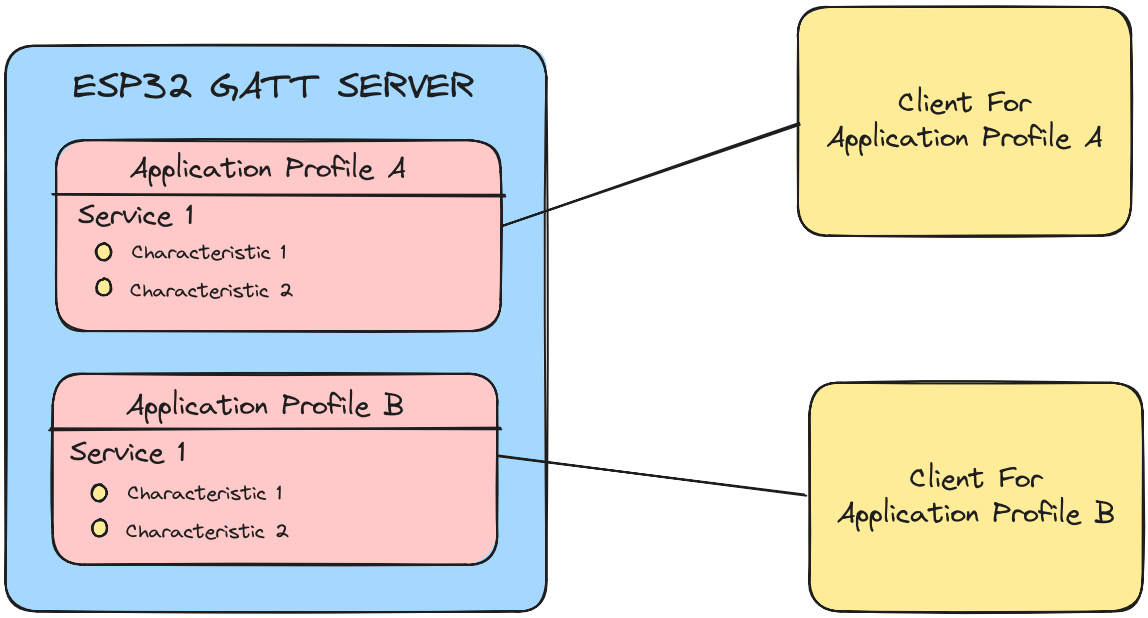

图 10.1: GATT 服务器

- 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

```
1 static struct gatts_profile_inst gl_profile_tab[PROFILE_NUM] = {
2     [PROFILE_A_APP_ID] = {
3         .gatts_cb = gatts_profile_a_event_handler,
4         .gatts_if = ESP_GATT_IF_NONE,
5     [PROFILE_B_APP_ID] = {
6         .gatts_cb = gatts_profile_b_event_handler,
7         .gatts_if = ESP_GATT_IF_NONE,
8     },
9 };
```

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

```
1 esp_ble_gatts_app_register(PROFILE_A_APP_ID);
2 esp_ble_gatts_app_register(PROFILE_B_APP_ID);
```

10.1.5 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：

esp_ble_adv_data_t

```
1 typedef struct {
2     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：

结构体初始化

```
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：

gatts_profile_a_event_handler

```
1  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) {
2      switch (event) {
3      case ESP_GATTS_REG_EVT:
4          ESP_LOGI(GATTS_TAG, "REGISTER_APP_EVT, status %d, app_id %d\n", param->
               reg.status, param->reg.app_id);
5          gl_profile_tab[PROFILE_A_APP_ID].service_id.is_primary = true;
6          gl_profile_tab[PROFILE_A_APP_ID].service_id.id.inst_id = 0x00;
7          gl_profile_tab[PROFILE_A_APP_ID].service_id.id.uuid.len =
               ESP_UUID_LEN_16;
8          gl_profile_tab[PROFILE_A_APP_ID].service_id.id.uuid.uuid.uuid16 =
               GATTS_SERVICE_UUID_TEST_A;
9
10         esp_ble_gap_set_device_name(TEST_DEVICE_NAME);
11 #ifdef CONFIG_SET_RAW_ADV_DATA
12         esp_err_t raw_adv_ret = esp_ble_gap_config_adv_data_raw(raw_adv_data,
               sizeof(raw_adv_data));
13         if (raw_adv_ret){
14             ESP_LOGE(GATTS_TAG, "config raw adv data failed, error code = %x ",
                   raw_adv_ret);
15         }
16         adv_config_done |= adv_config_flag;
17         esp_err_t raw_scan_ret = esp_ble_gap_config_scan_rsp_data_raw(
               raw_scan_rsp_data, sizeof(raw_scan_rsp_data));
18         if (raw_scan_ret){
19             ESP_LOGE(GATTS_TAG, "config raw scan rsp data failed, error code = %x
                   ", raw_scan_ret);
20         }
21         adv_config_done |= scan_rsp_config_flag;
22 #else
23         //config adv data
24         esp_err_t ret = esp_ble_gap_config_adv_data(&adv_data);
25         if (ret){
```

```
26            ESP_LOGE(GATTS_TAG, "config adv data failed, error code = %x", ret);
27        }
28        adv_config_done |= adv_config_flag;
29        //config scan response data
30        ret = esp_ble_gap_config_adv_data(&scan_rsp_data);
31        if (ret){
32            ESP_LOGE(GATTS_TAG, "config scan response data failed, error code = %
                  x", ret);
33        }
34        adv_config_done |= scan_rsp_config_flag;
35    #endif
```

10.1.6 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。

gap_event_handler

```
1   static void gap_event_handler(esp_gap_ble_cb_event_t event,
        esp_ble_gap_cb_param_t *param)
2   {
3       switch (event) {
4   #ifdef CONFIG_SET_RAW_ADV_DATA
5       case ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT:
6           adv_config_done &= (~adv_config_flag);
7           if (adv_config_done==0){
8               esp_ble_gap_start_advertising(&adv_params);
9           }
10          break;
11      case ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT:
12          adv_config_done &= (~scan_rsp_config_flag);
13          if (adv_config_done==0){
14              esp_ble_gap_start_advertising(&adv_params);
15          }
16          break;
17  #else
18      case ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT:
19          adv_config_done &= (~adv_config_flag);
```

```
20          if (adv_config_done == 0){
21              esp_ble_gap_start_advertising(&adv_params);
22          }
23          break;
24      case ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT:
25          adv_config_done &= (~scan_rsp_config_flag);
26          if (adv_config_done == 0){
27              esp_ble_gap_start_advertising(&adv_params);
28          }
29          break;
30  #endif
31  ...
```

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：

esp_ble_adv_params_t

```
1   /// Advertising parameters
2   typedef struct {
3       uint16_t adv_int_min;
4       /*!< Minimum advertising interval for undirected and low duty cycle
            directed advertising.
5                   Range: 0x0020 to 0x4000
6                   Default: N = 0x0800 (1.28 second)
7                   Time = N * 0.625 msec
8                   Time Range: 20 ms to 10.24 sec */
9       uint16_t adv_int_max;
10      /*!< Maximum advertising interval for undirected and low duty cycle
            directed advertising.
11                  Range: 0x0020 to 0x4000
12                  Default: N = 0x0800 (1.28 second)
13                  Time = N * 0.625 msec
14                  Time Range: 20 ms to 10.24 sec */
15      esp_ble_adv_type_t adv_type;      /*!< Advertising type */
16      esp_ble_addr_type_t own_addr_type; /*!< Owner bluetooth device address type
             */
17      esp_bd_addr_t peer_addr;          /*!< Peer device bluetooth device address
            */
18      esp_ble_addr_type_t peer_addr_type; /*!< Peer device bluetooth device
```

```
        address type */
19      esp_ble_adv_channel_t channel_map; /*!< Advertising channel map */
20      esp_ble_adv_filter_t adv_filter_policy; /*!< Advertising filter policy */
21  }
22  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：

广告参数

```
1   static esp_ble_adv_params_t test_adv_params = {
2       .adv_int_min     = 0x20,
3       .adv_int_max     = 0x40,
4       .adv_type        = ADV_TYPE_IND,
5       .own_addr_type   = BLE_ADDR_TYPE_PUBLIC,
6       //.peer_addr     =
7       //.peer_addr_type =
8       .channel_map     = ADV_CHNL_ALL,
9       .adv_filter_policy = ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY,
10  };
```

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。

打印错误消息

```
1   ...
2       case ESP_GAP_BLE_ADV_START_COMPLETE_EVT:
3       //advertising start complete event to indicate advertising start
            successfully or failed
4           if (param->adv_start_cmpl.status != ESP_BT_STATUS_SUCCESS) {
5               ESP_LOGE(GATTS_TAG, "Advertising start failed\n");
6           }
7           break;
```

```
8 ...
```

10.1.7 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：

参数

```
1 esp_gatt_status_t status; /*!< Operation status */‘
2 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。

gatts_event_handler

```
1  static void gatts_event_handler(esp_gatts_cb_event_t event, esp_gatt_if_t
       gatts_if, esp_ble_gatts_cb_param_t *param)
2  {
3     /* If event is register event, store the gatts_if for each profile */
4     if (event == ESP_GATTS_REG_EVT) {
5        if (param->reg.status == ESP_GATT_OK) {
6           gl_profile_tab[param->reg.app_id].gatts_if = gatts_if;
7        } else {
8           ESP_LOGI(GATTS_TAG, "Reg app failed, app_id %04x, status %d\n",
9                 param->reg.app_id,
10                param->reg.status);
11          return;
12       }
13    }
14
15 /* If the gatts_if equal to profile A, call profile A cb handler,
16  * so here call each profile's callback */
17    do {
18       int idx;
19       for (idx = 0; idx < PROFILE_NUM; idx++) {
20          if (gatts_if == ESP_GATT_IF_NONE||gatts_if == gl_profile_tab[idx].
              gatts_if) {
21             if (gl_profile_tab[idx].gatts_cb) {
```

```
                gl_profile_tab[idx].gatts_cb(event, gatts_if, param);
            }
        }
    }
    } while (0);
}
```

10.1.8 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

```
...
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：

句柄数量

```
#define GATTS_NUM_HANDLE_TEST_A 4
```

句柄包括：

1. 服务句柄
2. 特征句柄
3. 特征值句柄
4. 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：

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;
...
}
```

10.1.9 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：

ESP_GATTS_CREATE_EVT

```
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 =
```

```
10      esp_ble_gatts_add_char(gl_profile_tab[PROFILE_A_APP_ID].service_handle,
11                             &gl_profile_tab[PROFILE_A_APP_ID].char_uuid,
12                             ESP_GATT_PERM_READ | ESP_GATT_PERM_WRITE,
13                             a_property,
14                             &gatts_demo_char1_val,
15                             NULL);
16      if (add_char_ret){
17          ESP_LOGE(GATTS_TAG, "add char failed, error code =%x",add_char_ret);
18      }
19      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：

gatts_demo_char1_val

```
1  esp_attr_value_t gatts_demo_char1_val =
2  {
3      .attr_max_len = GATTS_DEMO_CHAR_VAL_LEN_MAX,
4      .attr_len     = sizeof(char1_str),
5      .attr_value   = char1_str,
6  };
```

Where `char1_str` is dummy data:

char1_str

```
uint8_t char1_str[] = {0x11,0x22,0x33};
```

特征长度定义为如下：

特征长度

```
#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_a` 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。

10.1.10 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：

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。

ESP_GATTS_ADD_CHAR_EVT

```
    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。

ESP_GATTS_ADD_CHAR_DESCR_EVT

```
    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。

10.1.11 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：

ESP_GATTS_CONNECT_EVT

```
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：

ESP_GATTS_CONNECT_EVT

```
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：

ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT

```
   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;
```

10.1.12 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：

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。

ESP_GATTS_READ_EVT

```
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;
   }
```

10.1.13 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：

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。

ESP_GATTS_WRITE_EVT

```
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,
19                                  sizeof(notify_data),
20                                  notify_data, false);
21              }
22          }else if (descr_value == 0x0002){
23              if (b_property & ESP_GATT_CHAR_PROP_BIT_INDICATE){
24                  ESP_LOGI(GATTS_TAG, "indicate enable");
25                  uint8_t indicate_data[15];
26                  for (int i = 0; i < sizeof(indicate_data); ++i)
27                  {
28                      indicate_data[i] = i % 0xff;
29                   }
30                   //the size of indicate_data[] need less than MTU size
31                  esp_ble_gatts_send_indicate(gatts_if, param->write.conn_id,
32                                  gl_profile_tab[PROFILE_B_APP_ID].
                                        char_handle,
33                                  sizeof(indicate_data),
34                                  indicate_data, true);
35              }
36          }
37          else if (descr_value == 0x0000){
38              ESP_LOGI(GATTS_TAG, "notify/indicate disable ");
39          }else{
40              ESP_LOGE(GATTS_TAG, "unknown value");
41          }
42      }
43  }
44  example_write_event_env(gatts_if, &a_prepare_write_env, param);
45  break;
46 }
```

`example_write_event_env()` 函数包含了写长特征过程的逻辑：

example_write_event_env

```
1 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){
2   esp_gatt_status_t status = ESP_GATT_OK;
3   if (param->write.need_rsp){
4     if (param->write.is_prep){
```

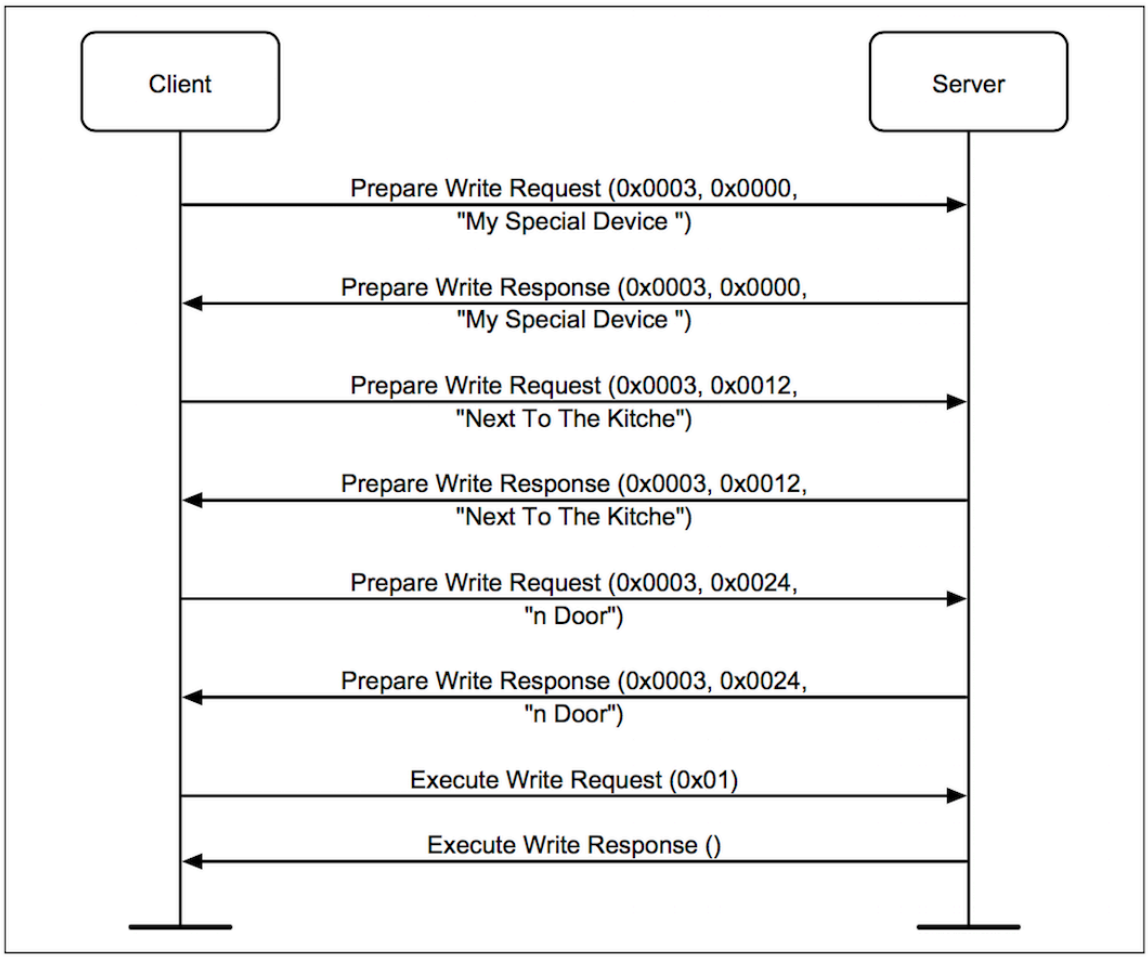

图 10.2: 写长特征的消息流

```
5           if (prepare_write_env->prepare_buf == NULL){
6               prepare_write_env->prepare_buf = (uint8_t *)malloc(
                    PREPARE_BUF_MAX_SIZE*sizeof(uint8_t));
7               prepare_write_env->prepare_len = 0;
8               if (prepare_write_env->prepare_buf == NULL) {
9                   ESP_LOGE(GATTS_TAG, "Gatt_server prep no mem\n");
10                  status = ESP_GATT_NO_RESOURCES;
11              }
12          } else {
13              if(param->write.offset > PREPARE_BUF_MAX_SIZE) {
14                  status = ESP_GATT_INVALID_OFFSET;
15              }
16              else if ((param->write.offset + param->write.len) >
                    PREPARE_BUF_MAX_SIZE) {
17                  status = ESP_GATT_INVALID_ATTR_LEN;
18              }
19          }
20
21          esp_gatt_rsp_t *gatt_rsp = (esp_gatt_rsp_t *)malloc(sizeof(
                esp_gatt_rsp_t));
22          gatt_rsp->attr_value.len = param->write.len;
23          gatt_rsp->attr_value.handle = param->write.handle;
24          gatt_rsp->attr_value.offset = param->write.offset;
25          gatt_rsp->attr_value.auth_req = ESP_GATT_AUTH_REQ_NONE;
26          memcpy(gatt_rsp->attr_value.value, param->write.value, param->write.
                len);
27          esp_err_t response_err = esp_ble_gatts_send_response(gatts_if, param
                ->write.conn_id,
28                                                               param->write.trans_id,
                                                                     status, gatt_rsp);
29          if (response_err != ESP_OK){
30             ESP_LOGE(GATTS_TAG, "Send response error\n");
31          }
32          free(gatt_rsp);
33          if (status != ESP_GATT_OK){
34              return;
35          }
36          memcpy(prepare_write_env->prepare_buf + param->write.offset,
37                param->write.value,
38                param->write.len);
39          prepare_write_env->prepare_len += param->write.len;
```

```
40
41          }else{
42              esp_ble_gatts_send_response(gatts_if, param->write.conn_id, param->
                    write.trans_id, status, NULL);
43          }
44      }
45  }
```

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。

example_write_event_env

```
1  void example_write_event_env(esp_gatt_if_t gatts_if, prepare_type_env_t *
       prepare_write_env,
2                               esp_ble_gatts_cb_param_t *param){
3      esp_gatt_status_t status = ESP_GATT_OK;
4      if (param->write.need_rsp){
5  ...
```

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。

写响应

```
1  ...
2  if (param->write.is_prep) {
3  ...
4  } else {
5      esp_ble_gatts_send_response(gatts_if, param->write.conn_id, param->write.
           trans_id, status, NULL);
6  }
7  ...
```

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：

prepare_type_evn_t

```
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:

分配内存

```
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。

检查消息是否能装进缓冲区

```
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。

发送响应

```
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：

拷贝输入数据

```
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：

ESP_GATTS_EXEC_WRITE_EVT

```
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：

执行写函数

```
1  void example_exec_write_event_env(prepare_type_env_t *prepare_write_env,
       esp_ble_gatts_cb_param_t *param){
2      if (param->exec_write.exec_write_flag == ESP_GATT_PREP_WRITE_EXEC){
3          esp_log_buffer_hex(GATTS_TAG, prepare_write_env->prepare_buf,
               prepare_write_env->prepare_len);
4      }
5      else{
6          ESP_LOGI(GATTS_TAG,"ESP_GATT_PREP_WRITE_CANCEL");
7      }
8      if (prepare_write_env->prepare_buf) {
9          free(prepare_write_env->prepare_buf);
10         prepare_write_env->prepare_buf = NULL;
11     }
12 ####    prepare_write_env->prepare_len = 0;
13 }
```

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。

打印一些日志

```
1  if (param->exec_write.exec_write_flag == ESP_GATT_PREP_WRITE_EXEC) {
2     esp_log_buffer_hex(GATTS_TAG,
3                        prepare_write_env->prepare_buf,
4                        prepare_write_env->prepare_len);
5  } else {
6      ESP_LOGI(GATTS_TAG,"ESP_GATT_PREP_WRITE_CANCEL");
7  }
```

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。

释放缓冲区

```
1  if (prepare_write_env->prepare_buf) {
2      free(prepare_write_env->prepare_buf);
```

```
3     prepare_write_env->prepare_buf = NULL;
4 }
5 prepare_write_env->prepare_len = 0;
```

10.1.14 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。

第十一章 WIFI 模块

wifi 模块相对蓝牙就简单很多了。
头文件如下：

wifi_driver.h

```c
#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

wifi_driver.c

```c
#include "wifi_driver.h"

```

```
3  EventGroupHandle_t s_wifi_event_group;
4
5  const char *WIFI_TAG = "wifi station";
6  int s_retry_num = 0;
7
8  void event_handler(void *arg, esp_event_base_t event_base,
9                     int32_t event_id, void *event_data)
10 {
11     if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_START)
12     {
13         esp_wifi_connect();
14     }
15     else if (event_base == WIFI_EVENT && event_id ==
           WIFI_EVENT_STA_DISCONNECTED)
16     {
17         if (s_retry_num < EXAMPLE_ESP_MAXIMUM_RETRY)
18         {
19             esp_wifi_connect();
20             s_retry_num++;
21             ESP_LOGI(WIFI_TAG, "retry to connect to the AP");
22         }
23         else
24         {
25             xEventGroupSetBits(s_wifi_event_group, WIFI_FAIL_BIT);
26         }
27         ESP_LOGI(WIFI_TAG, "connect to the AP fail");
28     }
29     else if (event_base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP)
30     {
31         ip_event_got_ip_t *event = (ip_event_got_ip_t *)event_data;
32         ESP_LOGI(WIFI_TAG, "got ip:" IPSTR, IP2STR(&event->ip_info.ip));
33         s_retry_num = 0;
34         xEventGroupSetBits(s_wifi_event_group, WIFI_CONNECTED_BIT);
35     }
36 }
37
38 void wifi_init_sta(void)
39 {
40     s_wifi_event_group = xEventGroupCreate();
41
42     ESP_ERROR_CHECK(esp_netif_init());
```

```
43
44      ESP_ERROR_CHECK(esp_event_loop_create_default());
45      esp_netif_create_default_wifi_sta();
46
47      wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
48      ESP_ERROR_CHECK(esp_wifi_init(&cfg));
49
50      esp_event_handler_instance_t instance_any_id;
51      esp_event_handler_instance_t instance_got_ip;
52      ESP_ERROR_CHECK(esp_event_handler_instance_register(WIFI_EVENT,
53                                                          ESP_EVENT_ANY_ID,
54                                                          &event_handler,
55                                                          NULL,
56                                                          &instance_any_id));
57      ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_EVENT,
58                                                          IP_EVENT_STA_GOT_IP,
59                                                          &event_handler,
60                                                          NULL,
61                                                          &instance_got_ip));
62
63      wifi_config_t wifi_config = {
64          .sta = {
65              .ssid = EXAMPLE_ESP_WIFI_SSID,
66              .password = EXAMPLE_ESP_WIFI_PASS,
67              /* Authmode threshold resets to WPA2 as default if password matches
                    WPA2 standards (pasword len => 8).
68               * If you want to connect the device to deprecated WEP/WPA networks,
                    Please set the threshold value
69               * to WIFI_AUTH_WEP/WIFI_AUTH_WPA_PSK and set the password with
                    length and format matching to
70               * WIFI_AUTH_WEP/WIFI_AUTH_WPA_PSK standards.
71               */
72              .threshold.authmode = ESP_WIFI_SCAN_AUTH_MODE_THRESHOLD,
73              .sae_pwe_h2e = ESP_WIFI_SAE_MODE,
74              .sae_h2e_identifier = EXAMPLE_H2E_IDENTIFIER,
75          },
76      };
77      ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
78      ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &wifi_config));
79      ESP_ERROR_CHECK(esp_wifi_start());
80
```

```
81      ESP_LOGI(WIFI_TAG, "wifi_init_sta finished.");
82
83      /* Waiting until either the connection is established (WIFI_CONNECTED_BIT)
            or connection failed for the maximum
84       * number of re-tries (WIFI_FAIL_BIT). The bits are set by event_handler()
            (see above) */
85      EventBits_t bits = xEventGroupWaitBits(s_wifi_event_group,
86                                             WIFI_CONNECTED_BIT | WIFI_FAIL_BIT,
87                                             pdFALSE,
88                                             pdFALSE,
89                                             portMAX_DELAY);
90
91      /* xEventGroupWaitBits() returns the bits before the call returned, hence
            we can test which event actually
92       * happened. */
93      if (bits & WIFI_CONNECTED_BIT)
94      {
95          ESP_LOGI(WIFI_TAG, "connected to ap SSID:%s password:%s",
96                   EXAMPLE_ESP_WIFI_SSID, EXAMPLE_ESP_WIFI_PASS);
97      }
98      else if (bits & WIFI_FAIL_BIT)
99      {
100         ESP_LOGI(WIFI_TAG, "Failed to connect to SSID:%s, password:%s",
101                  EXAMPLE_ESP_WIFI_SSID, EXAMPLE_ESP_WIFI_PASS);
102     }
103     else
104     {
105         ESP_LOGE(WIFI_TAG, "UNEXPECTED EVENT");
106     }
107 }
108
109 void WIFI_Init(void)
110 {
111     // Initialize NVS
112     esp_err_t ret = nvs_flash_init();
113     if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
            ESP_ERR_NVS_NEW_VERSION_FOUND)
114     {
115         ESP_ERROR_CHECK(nvs_flash_erase());
116         ret = nvs_flash_init();
117     }
```

```
118     ESP_ERROR_CHECK(ret);
119
120     ESP_LOGI(WIFI_TAG, "ESP_WIFI_MODE_STA");
121     wifi_init_sta();
122 }
```

第十二章 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：

tcp_driver.h

```
#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

tcp_driver.c

```
#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);
}
```

```
48
49  void tcp_server_task(void *pvParameters)
50  {
51      char addr_str[128];
52      int addr_family = (int)pvParameters;
53      int ip_protocol = 0;
54      int keepAlive = 1;
55      int keepIdle = KEEPALIVE_IDLE;
56      int keepInterval = KEEPALIVE_INTERVAL;
57      int keepCount = KEEPALIVE_COUNT;
58      struct sockaddr_storage dest_addr;
59
60      if (addr_family == AF_INET)
61      {
62          struct sockaddr_in *dest_addr_ip4 = (struct sockaddr_in *)&dest_addr;
63          dest_addr_ip4->sin_addr.s_addr = htonl(INADDR_ANY);
64          dest_addr_ip4->sin_family = AF_INET;
65          dest_addr_ip4->sin_port = htons(PORT);
66          ip_protocol = IPPROTO_IP;
67      }
68
69      int listen_sock = socket(addr_family, SOCK_STREAM, ip_protocol);
70      if (listen_sock < 0)
71      {
72          ESP_LOGE(TCP_TAG, "Unable to create socket: errno %d", errno);
73          vTaskDelete(NULL);
74          return;
75      }
76      int opt = 1;
77      setsockopt(listen_sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
78
79      ESP_LOGI(TCP_TAG, "Socket created");
80
81      int err = bind(listen_sock, (struct sockaddr *)&dest_addr, sizeof(dest_addr
            ));
82      if (err != 0)
83      {
84          ESP_LOGE(TCP_TAG, "Socket unable to bind: errno %d", errno);
85          ESP_LOGE(TCP_TAG, "IPPROTO: %d", addr_family);
86          goto CLEAN_UP;
87      }
```

```
88      ESP_LOGI(TCP_TAG, "Socket bound, port %d", PORT);
89
90      err = listen(listen_sock, 1);
91      if (err != 0)
92      {
93          ESP_LOGE(TCP_TAG, "Error occurred during listen: errno %d", errno);
94          goto CLEAN_UP;
95      }
96
97      while (1)
98      {
99
100         ESP_LOGI(TCP_TAG, "Socket listening");
101
102         struct sockaddr_storage source_addr; // Large enough for both IPv4 or
                IPv6
103         socklen_t addr_len = sizeof(source_addr);
104         int sock = accept(listen_sock, (struct sockaddr *)&source_addr, &
                addr_len);
105         if (sock < 0)
106         {
107             ESP_LOGE(TCP_TAG, "Unable to accept connection: errno %d", errno);
108             break;
109         }
110
111         // Set tcp keepalive option
112         setsockopt(sock, SOL_SOCKET, SO_KEEPALIVE, &keepAlive, sizeof(int));
113         setsockopt(sock, IPPROTO_TCP, TCP_KEEPIDLE, &keepIdle, sizeof(int));
114         setsockopt(sock, IPPROTO_TCP, TCP_KEEPINTVL, &keepInterval, sizeof(int))
                ;
115         setsockopt(sock, IPPROTO_TCP, TCP_KEEPCNT, &keepCount, sizeof(int));
116         // Convert ip address to string
117         if (source_addr.ss_family == PF_INET)
118         {
119             inet_ntoa_r(((struct sockaddr_in *)&source_addr)->sin_addr, addr_str,
                    sizeof(addr_str) - 1);
120         }
121         ESP_LOGI(TCP_TAG, "Socket accepted ip address: %s", addr_str);
122
123         do_retransmit(sock);
124
```

```
125         shutdown(sock, 0);
126         close(sock);
127     }
128 
129 CLEAN_UP:
130     close(listen_sock);
131     vTaskDelete(NULL);
132 }
```

第十三章 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：

partitions_ota.csv

```
# 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。

修改 Flash 大小

```
idf.py menuconfig
```

然后编写头文件代码：

ota_driver.h

```
#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"
```

```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

ota_driver.c

```c
#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：

启动服务器

```
python -m http.server 8070
```