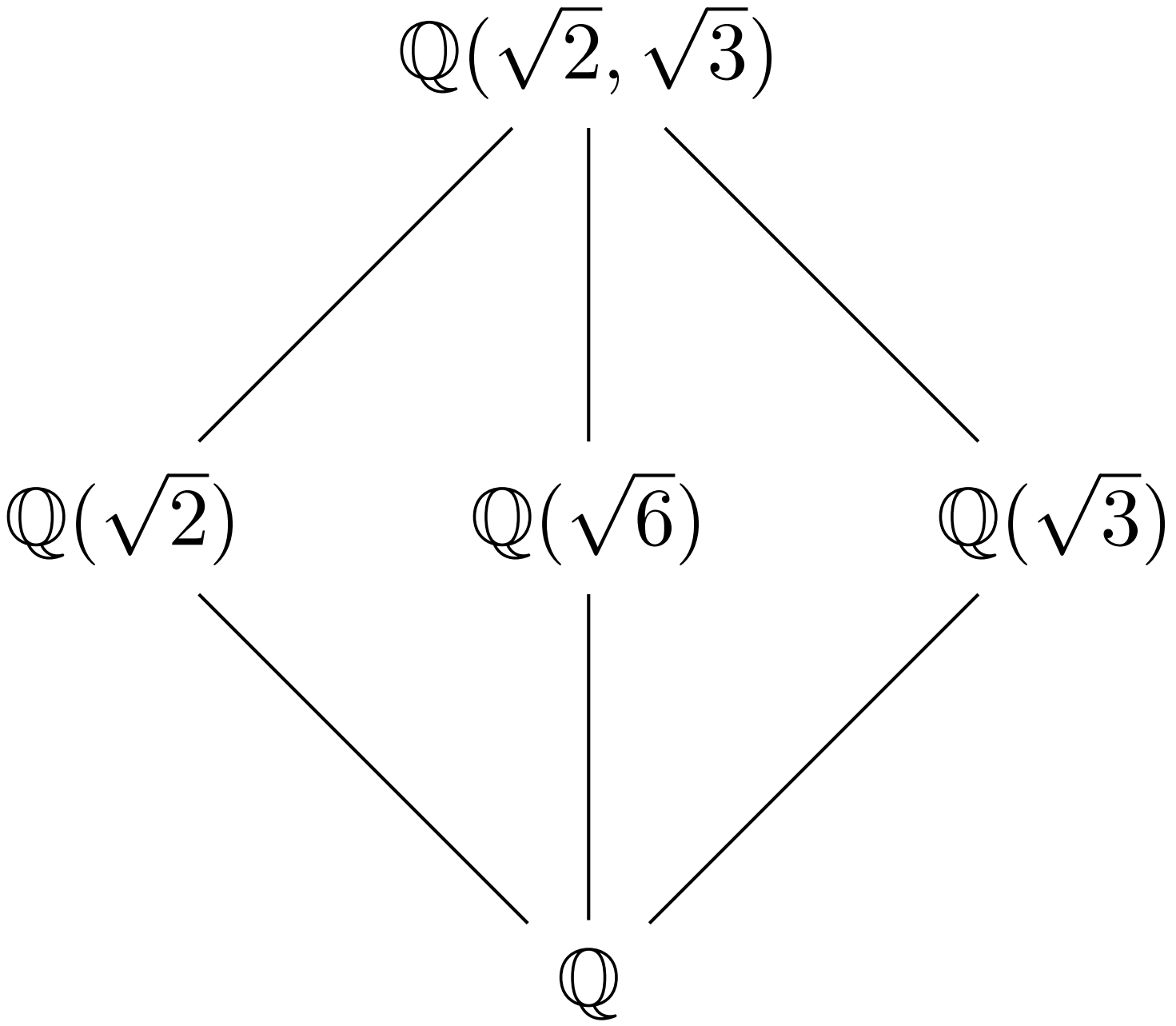


ESP32 **教程**

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。

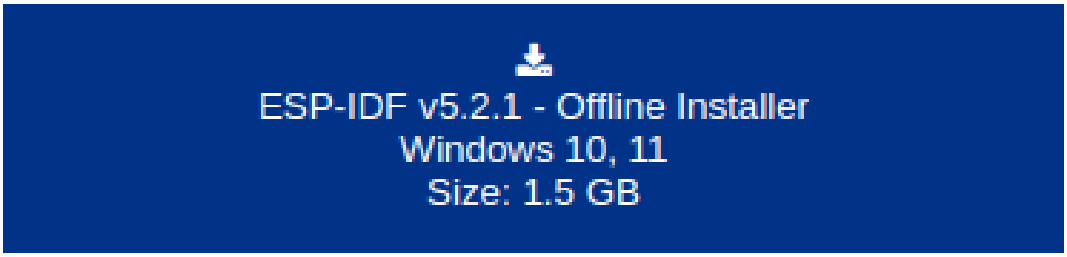


图 2.1: 离线安装包

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:

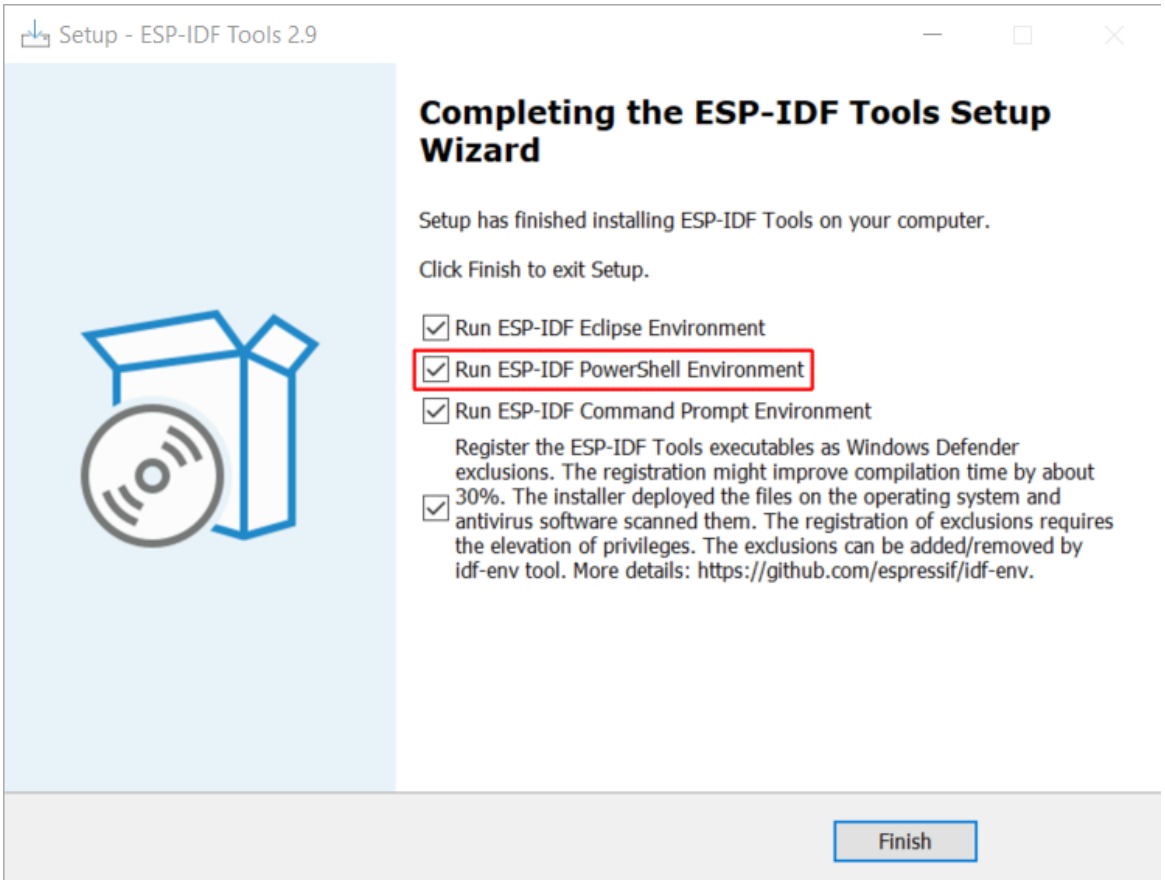


图 2.2: PowerShell

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：

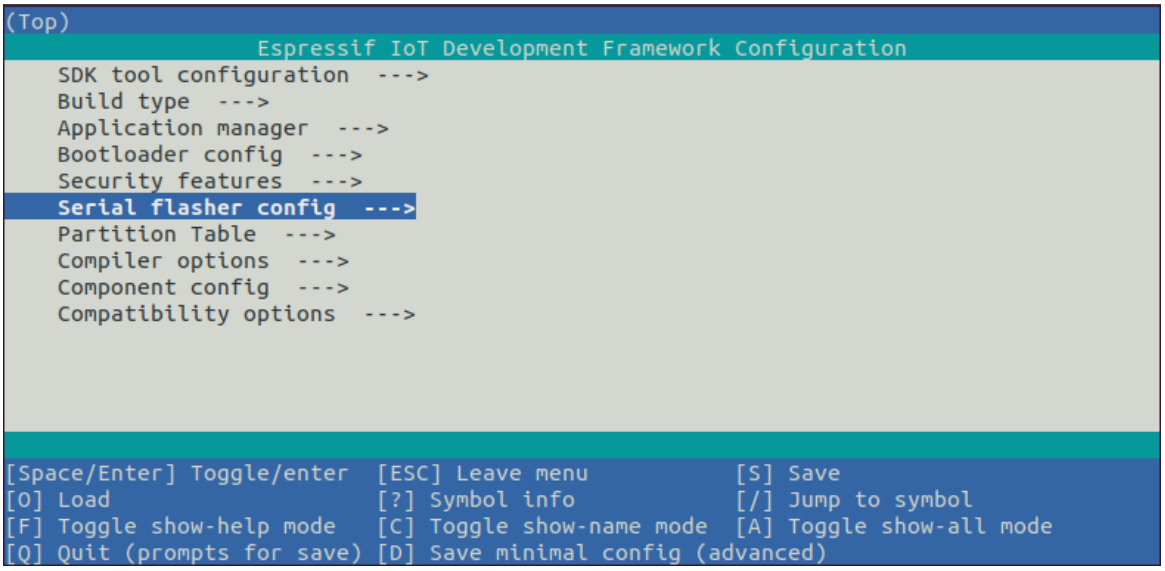


图 3.1: 配置界面示意图

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

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

```
# cmake版本管理工具的最小版本号
cmake_minimum_required(VERSION 3.16)

# 项目的cmake文件路径，包含在 ESP32 的 SDK 中
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
# 项目名称
project(atguigu_gpio_example)
```

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

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

```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

```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)

```

```
23 /// 读取SDA引脚电平的值
24 #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

```
1 void Delay_ms(uint8_t time)
2 {
3     vTaskDelay(time / portTICK_PERIOD_MS);
4 }
```

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

```
1  void I2C_Start(void)
2  {
3      I2C_SDA_OUT;
4      I2C_SDA_H;
5      I2C_SCL_H;
6      Delay_ms(1);
7      I2C_SDA_L;
8      Delay_ms(1);
9      I2C_SCL_L;
10     Delay_ms(1);
11 }
```

停止 I^2C 通信。

atguigu-gpio-example/main/gpio_main.c

```
1 void I2C_Stop(void)
2 {
3     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

```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;
        }
```

```
42          Delay_ms(1);
43      }
44      I2C_Ack(1);
45      I2C_Stop();
46      return k;
47  }
```

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

```
1   uint8_t KEYBOARD_read_key(void)
2   {
3       uint16_t key = I2C_Read_Key();
4       if (key == 4)
5           return 1;
6       else if (key == 3)
7           return 2;
8       else if (key == 2)
9           return 3;
10      else if (key == 7)
11          return 4;
12      else if (key == 6)
13          return 5;
14      else if (key == 5)
15          return 6;
16      else if (key == 10)
17          return 7;
18      else if (key == 9)
19          return 8;
20      else if (key == 8)
21          return 9;
22      else if (key == 1)
23          return 0;
24      else if (key == 12)
25          return '#';
26      else if (key == 11)
27          return 'M';
28      return 255;
```

```
29 }
```

> ! 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。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

```
1 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

```
1 /// 当回调函数执行时，参数 arg 是产生中断的 GPIO 引脚号。
2 static void IRAM_ATTR gpio_isr_handler(void *arg)
3 {
4     uint32_t gpio_num = (uint32_t)arg;
5     /// 将 GPIO 引脚号添加到 gpio_event_queue 队列中。
6     xQueueSendFromISR(gpio_event_queue, &gpio_num, NULL);
7 }
```

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

```
1  static void process_isr(void *arg)
2  {
3      uint32_t gpio_num;
4      for (;;)
5      {
6          /// 如果队列中有 GPIO 中断事件，将 GPIO 引脚号存储到 gpio_num 变量中。
7          if (xQueueReceive(gpio_event_queue, &gpio_num, portMAX_DELAY))
8          {
9              /// 如果产生中断的GPIO引脚号是 GPIO_NUM_0，也就是键盘中断引脚。
10             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

```
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();
    KEYBORAD_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 flash
```

然后就可以测试程序了。

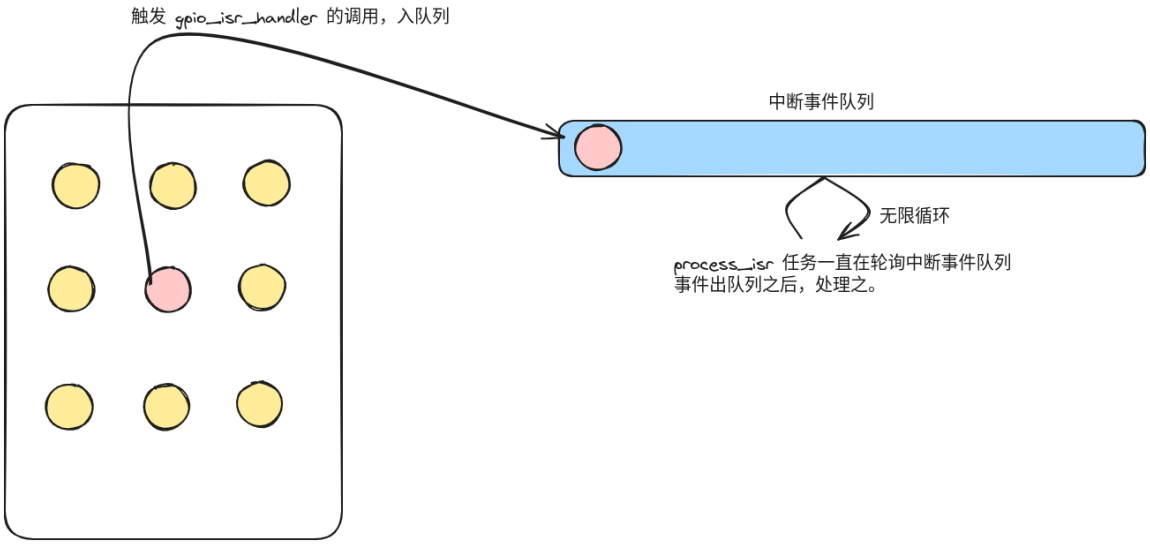


图 4.1: 处理中断的流程

第五章 红外遥控（RMT）

5.1 简介

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

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

RMT 外设通常支持以下场景：

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

5.2 RMT 符号的内存布局

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

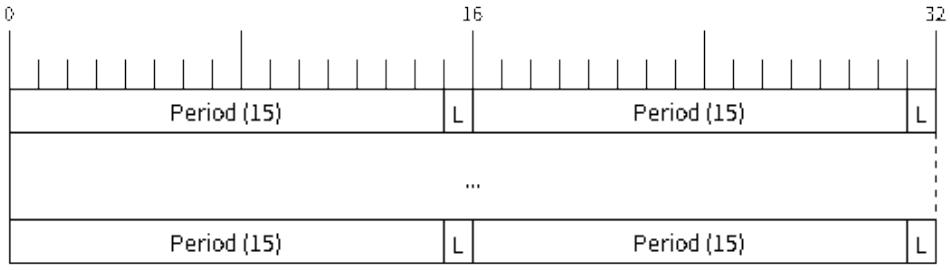


图 5.1: RMT 符号结构 (L-信号电平)

5.3 RMT 发射器概述

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

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

5.4 RMT 接收器概述

RMT 接收通道（RX Channel）的数据路径和控制路径如下图所示：

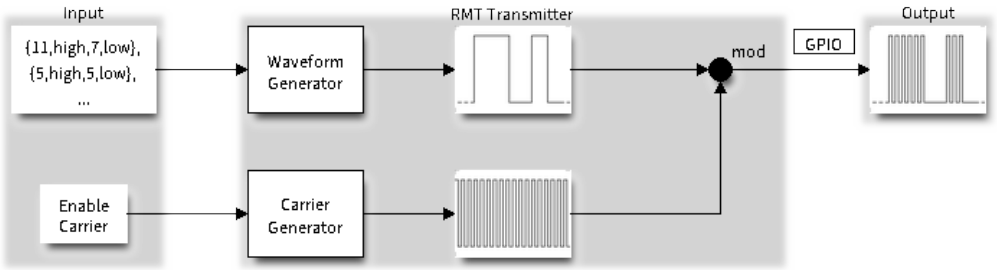


图 5.2: RMT 发射器概述

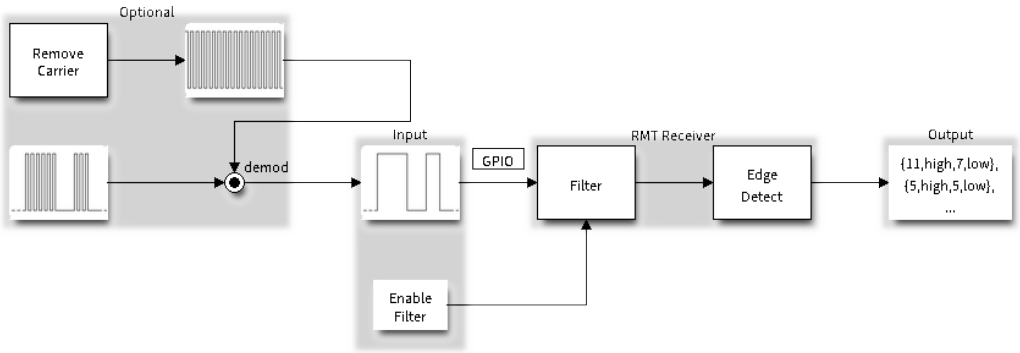


图 5.3: RMT 接收器概述

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。

5.5 补充知识：信号处理

5.5.1 正弦波定义

图 5.4 显示了正弦波的定义，包含幅度、相位和频率。

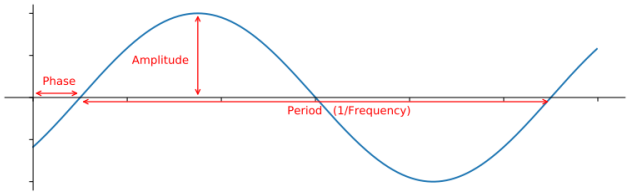


图 5.4: 正弦波信号

5.5.2 时域和频域

图 5.5 左侧是时域的图形，右侧是相同信号频域的图形。

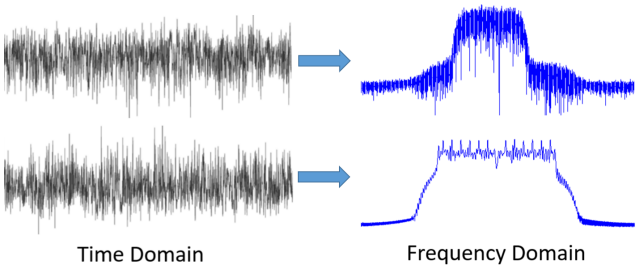


图 5.5: 时域和频域

5.5.3 周期信号是一系列正弦波的叠加

图 5.6 显示了频率为 1 和频率为 2 的正弦波相加形成的波形。

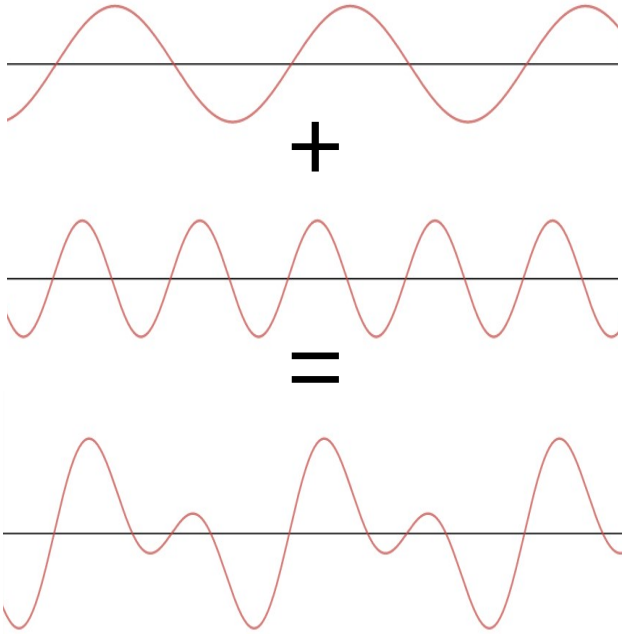


图 5.6: 频率为 1 和 2 的正弦波相加

图 5.7 → 5.16 显示了三角波是如何由不同频率的正弦波叠加而成的。

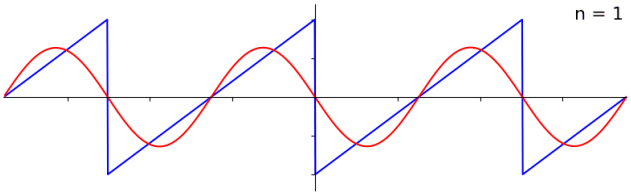


图 5.7: 三角波的叠加过程，n = 1

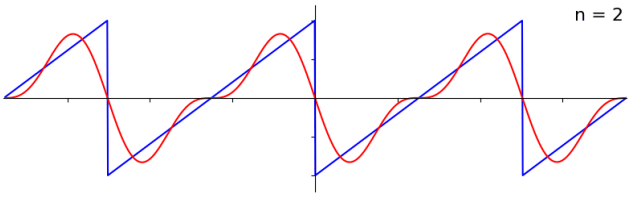


图 5.8: 三角波的叠加过程，n = 2

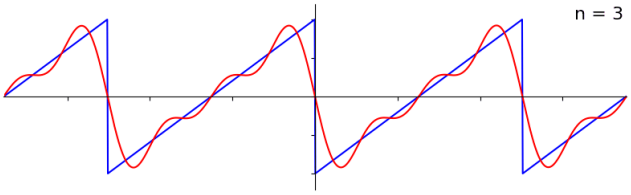


图 5.9: 三角波的叠加过程，n = 3

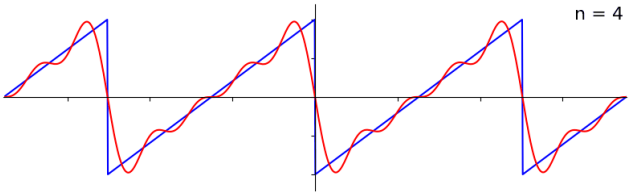


图 5.10: 三角波的叠加过程，n = 4

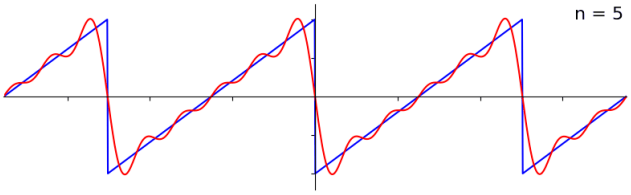


图 5.11: 三角波的叠加过程，n = 5

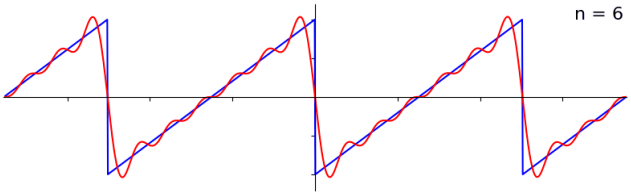


图 5.12: 三角波的叠加过程，n = 6

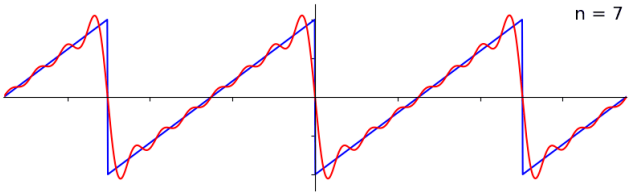


图 5.13: 三角波的叠加过程，n = 7

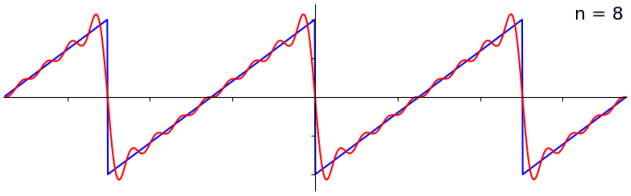


图 5.14: 三角波的叠加过程，n = 8

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 5.17 → 5.26 显示了这一过程。

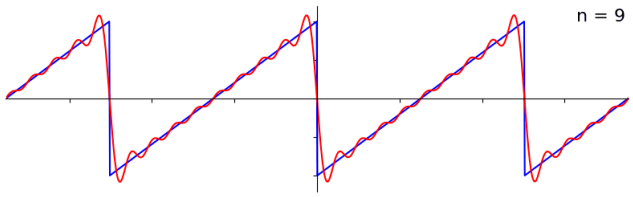


图 5.15: 三角波的叠加过程，n = 9

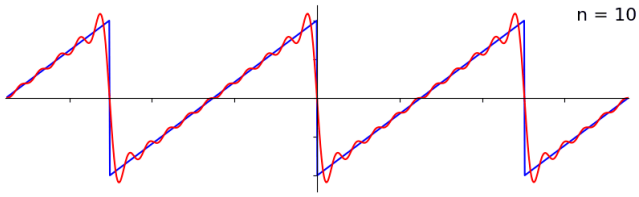


图 5.16: 三角波的叠加过程，n = 10

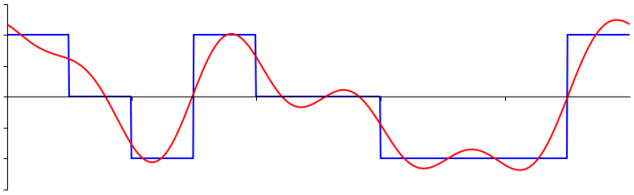


图 5.17: 方波的叠加过程-1

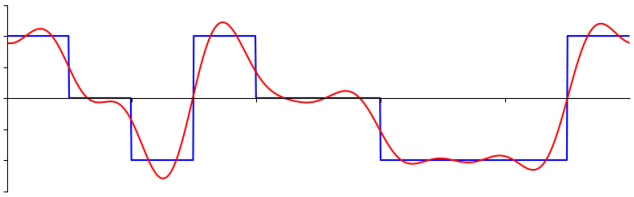


图 5.18: 方波的叠加过程-2

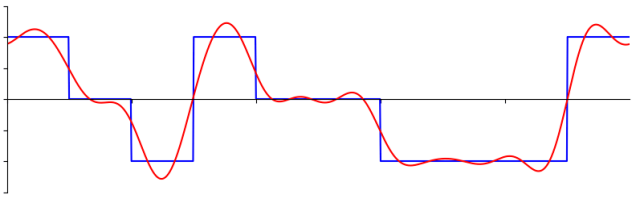


图 5.19: 方波的叠加过程-3

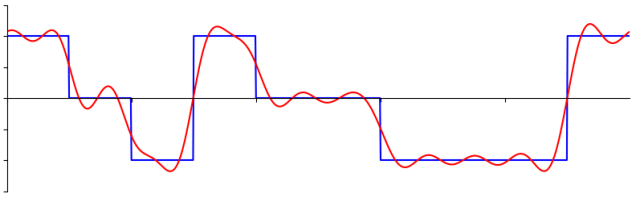


图 5.20: 方波的叠加过程-4

5.5.4 时域和频域举例

图 5.27 → 5.30 给出了几种常见信号的时域和频域的对比图。

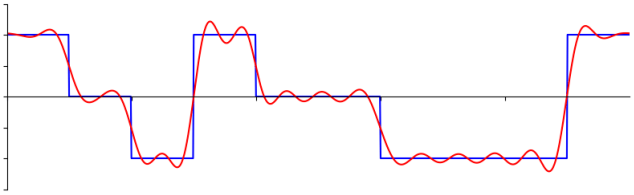


图 5.21: 方波的叠加过程-5

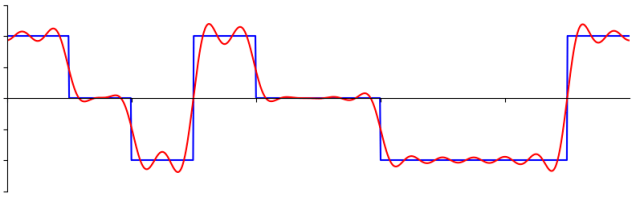


图 5.22: 方波的叠加过程-6

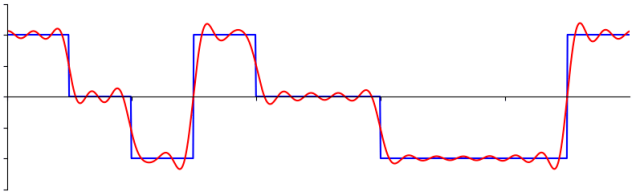


图 5.23: 方波的叠加过程-7

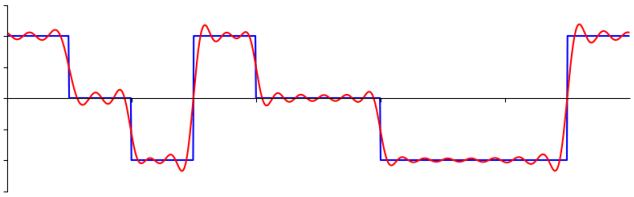


图 5.24: 方波的叠加过程-8

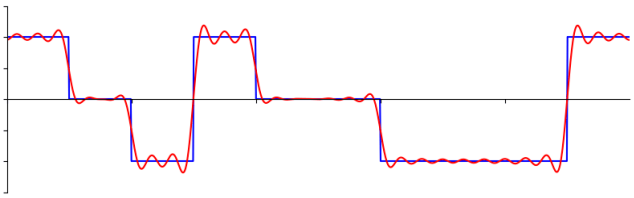


图 5.25: 方波的叠加过程-9

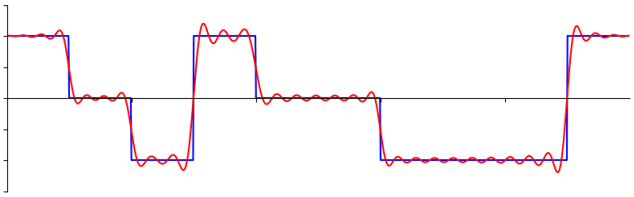


图 5.26: 方波的叠加过程-10

5.5.5 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。

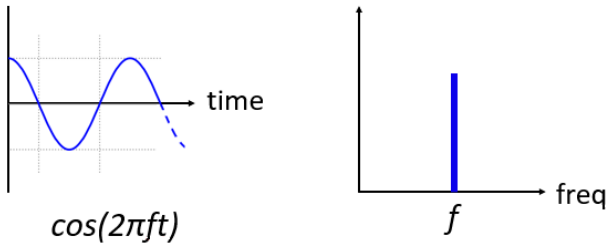


图 5.27: 正弦波

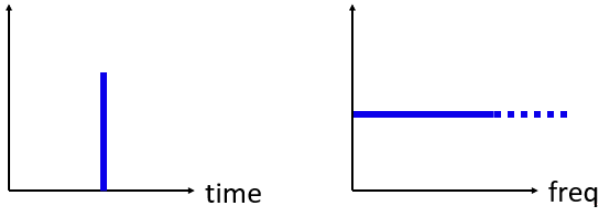


图 5.28: 脉冲信号

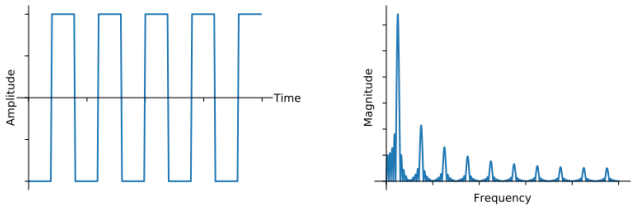


图 5.29: 方波信号的时域和频域

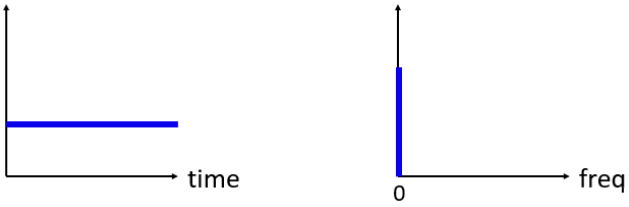


图 5.30: 常量信号

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 $0 \le k \le N-1$。

5.5.6 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 5.31 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` 。

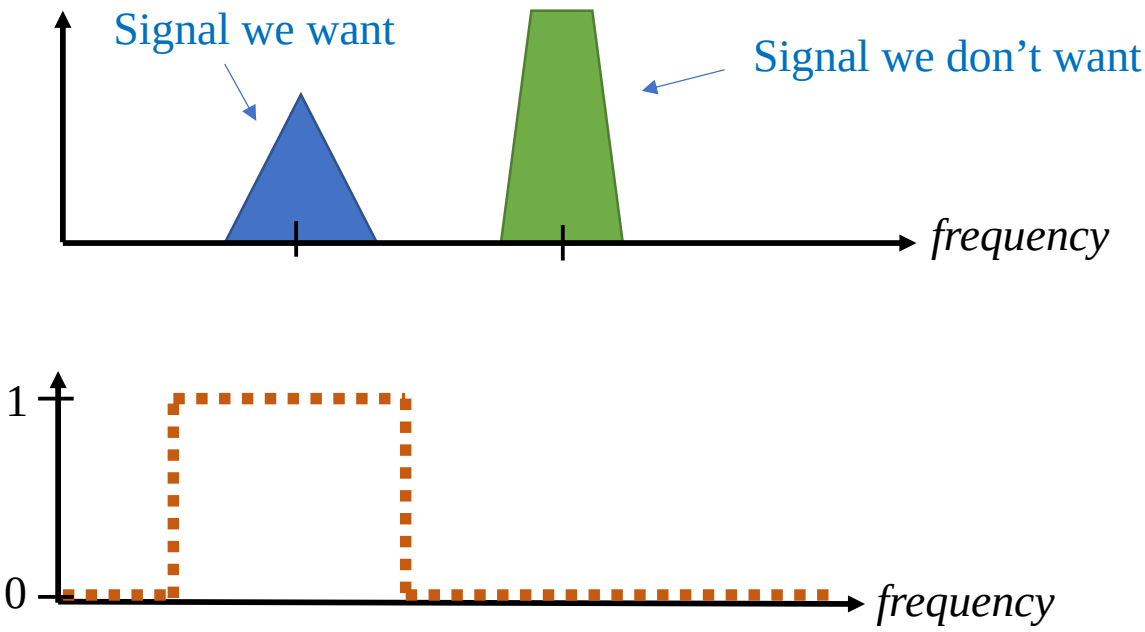


图 5.31: 频域相乘和时域相乘的对应关系

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 5.32 。

5.5.7 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 5.33 所示。

每个周期采样 1.2 个点，还是无法还原原始信号。如图 5.34 所示。

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 5.35 。

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 5.36 。

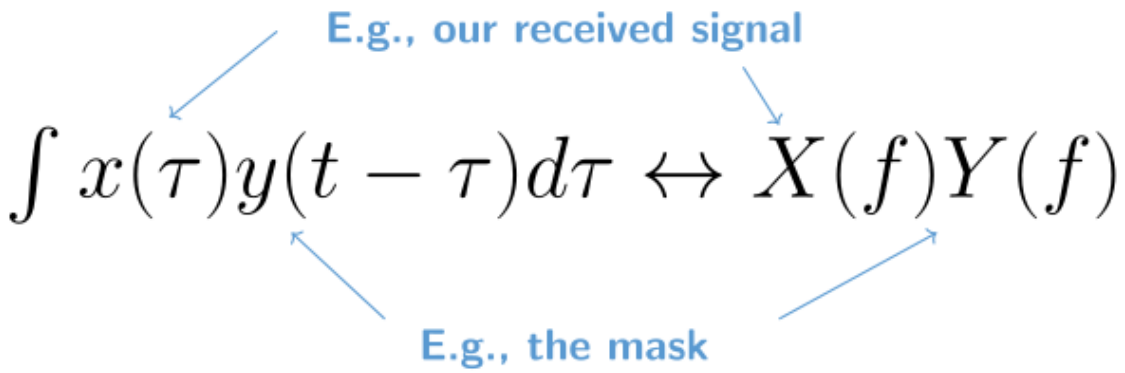


图 5.32: 频域相乘和时域相乘的对应关系

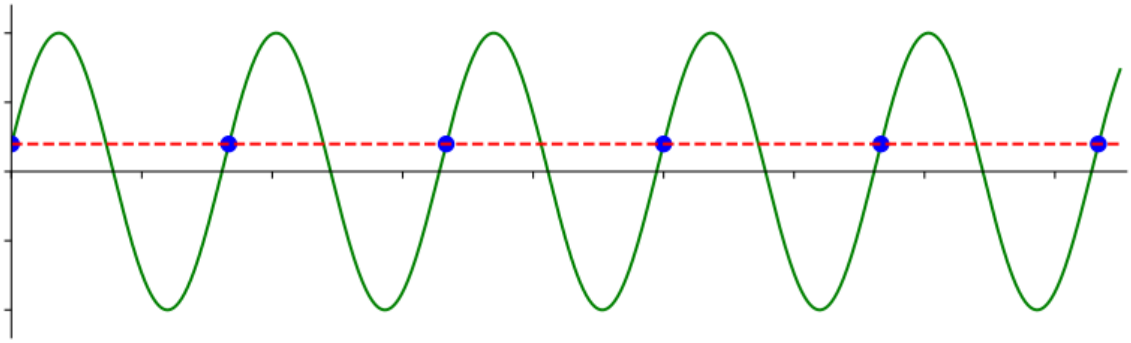


图 5.33: 每个周期采样 1 个点

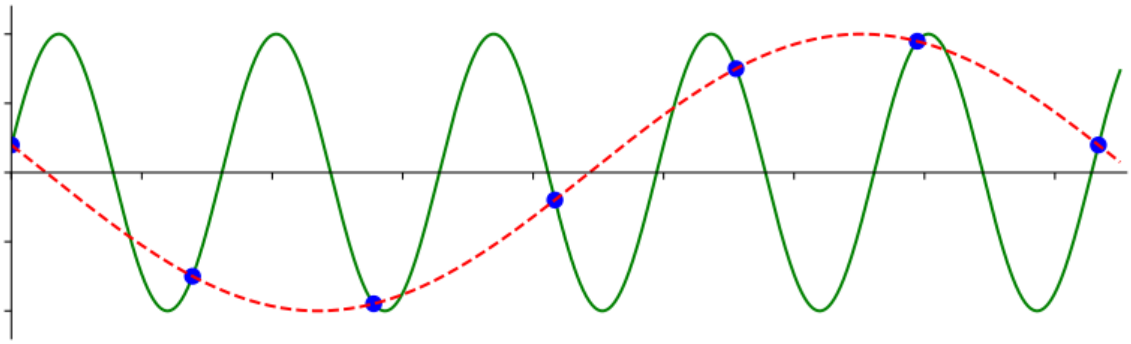


图 5.34: 每个周期采样 1.2 个点

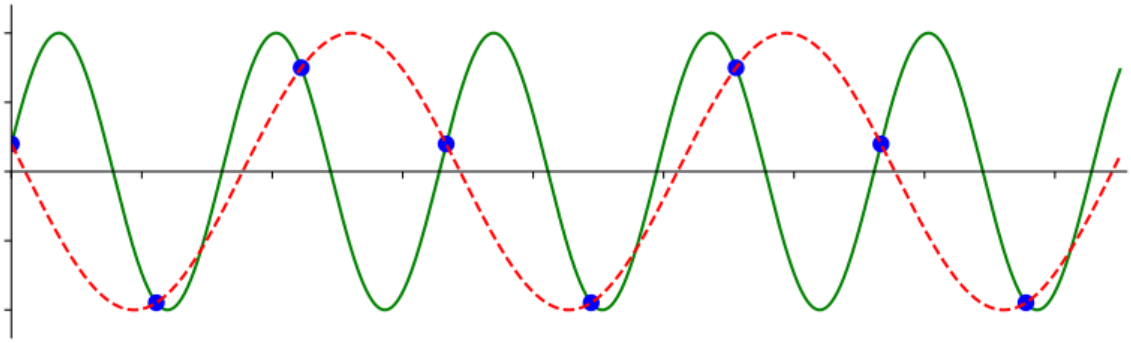


图 5.35: 每个周期采样 1.5 个点

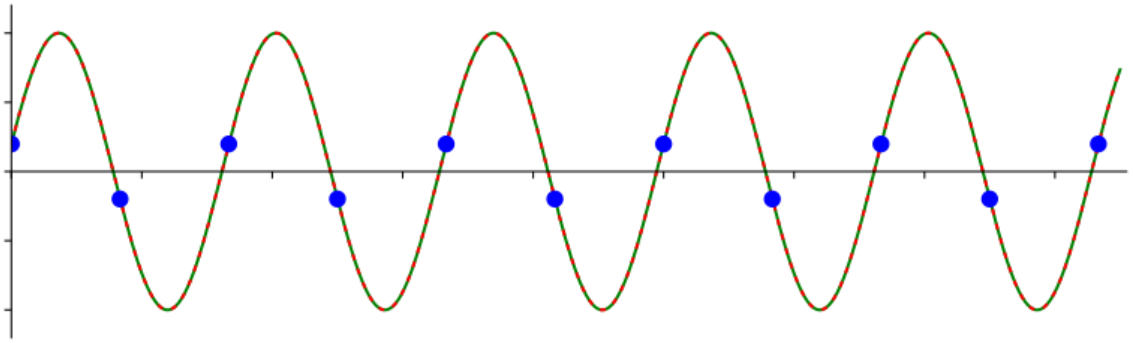


图 5.36: 每个周期采样 2 个点

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。

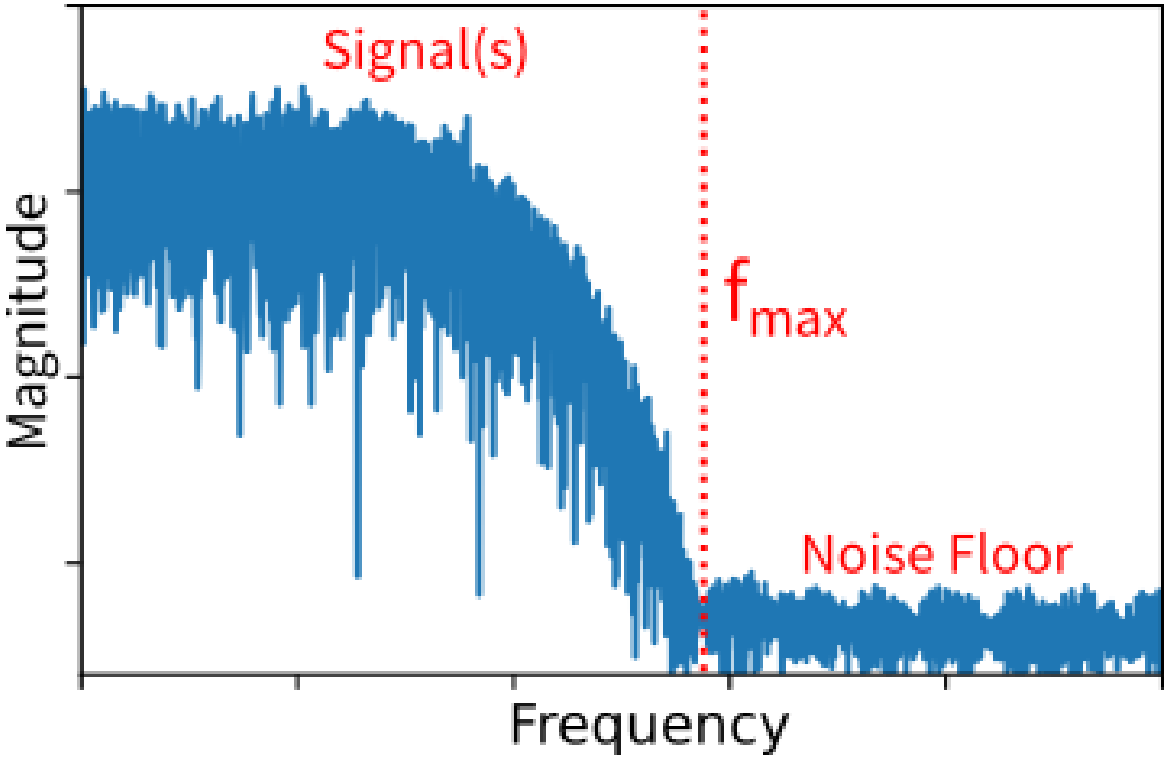


图 5.37: 采样定理

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。

5.5.8 数字调制

以太网传输的信号如图 5.39 。

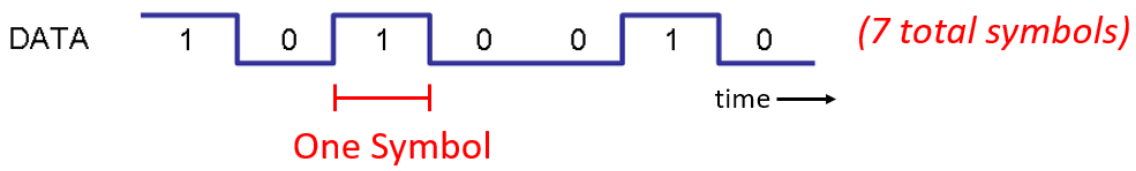


图 5.38: 数字信号

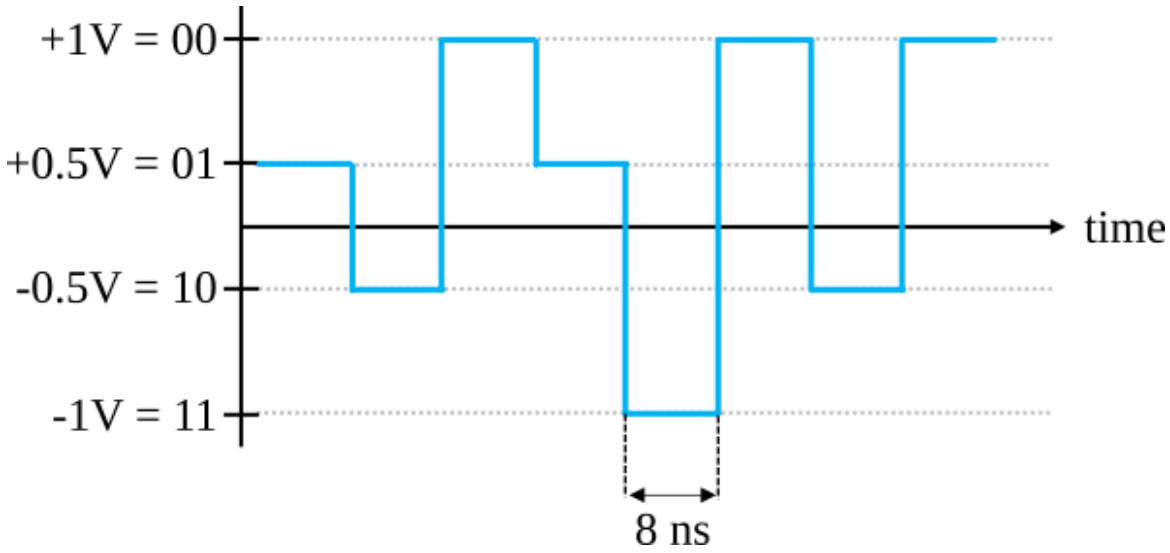


图 5.39: 以太网中的信号

1. 在图 5.39 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：$\frac{1}{8\times10^{-9}} \times 4 = 0.5Gbps$。

5.5.8.1 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 5.40 。如果用 4 个不同的电平调制信号，那么叫做 4-ASK，如图 5.41 。对于 4-ASK，每个符号可以表示 2 个 bit。

5.5.8.2 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 。

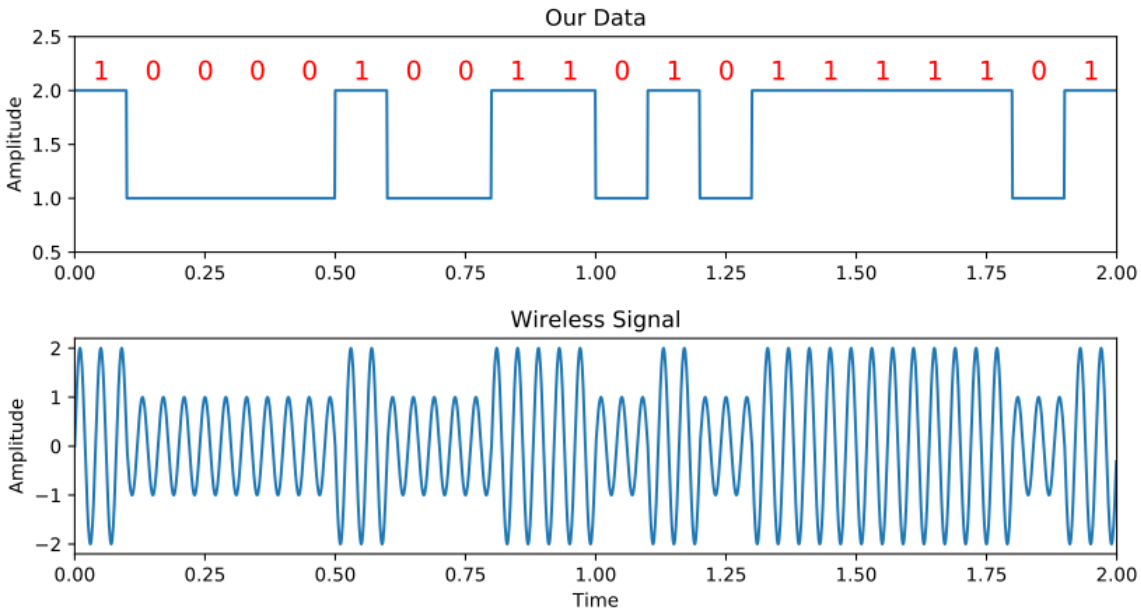


图 5.40: 2-ASK 调制

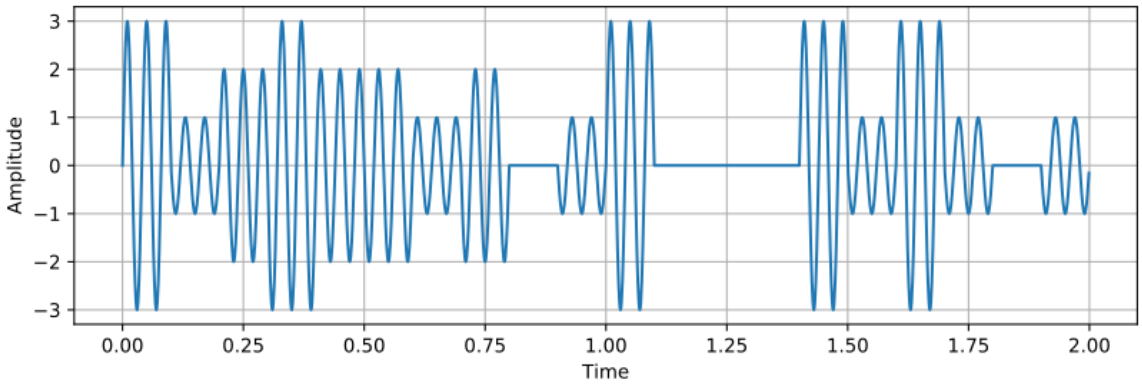


图 5.41: 4-ASK 调制

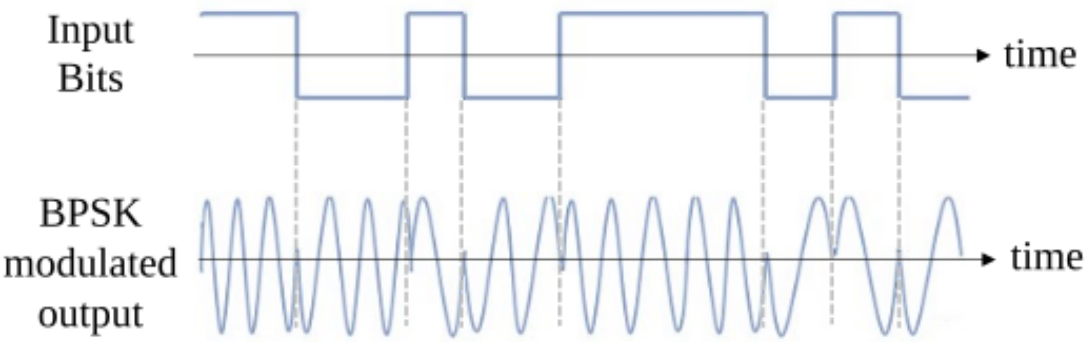


图 5.42: BPSK 调制

5.5.8.3 数字频率调制

5.5.9 模拟调制

5.5.9.1 模拟调幅

5.5.9.2 模拟调频

5.6 WS2812

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

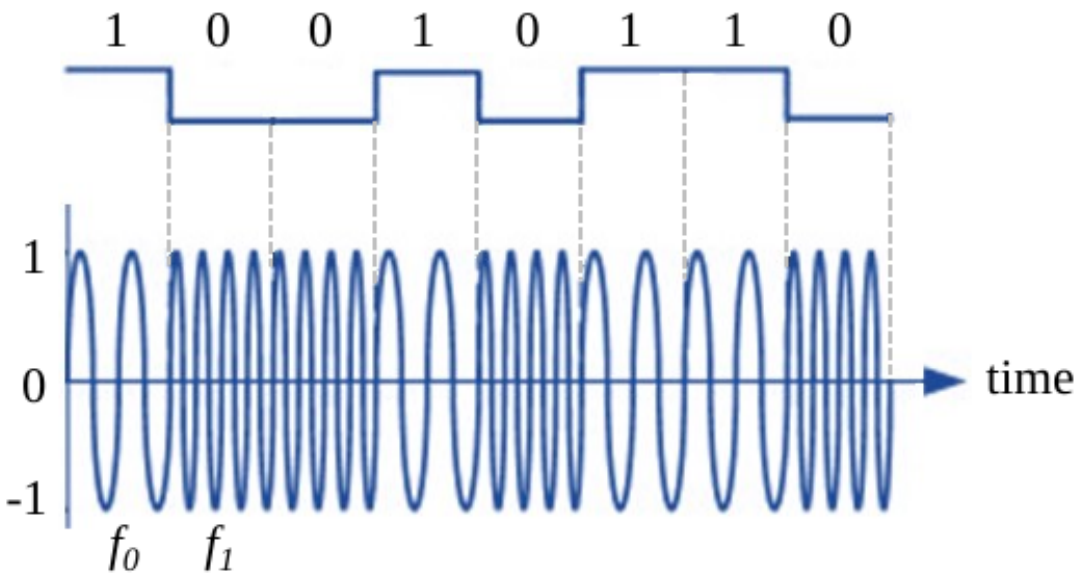


图 5.43: FSK 调制

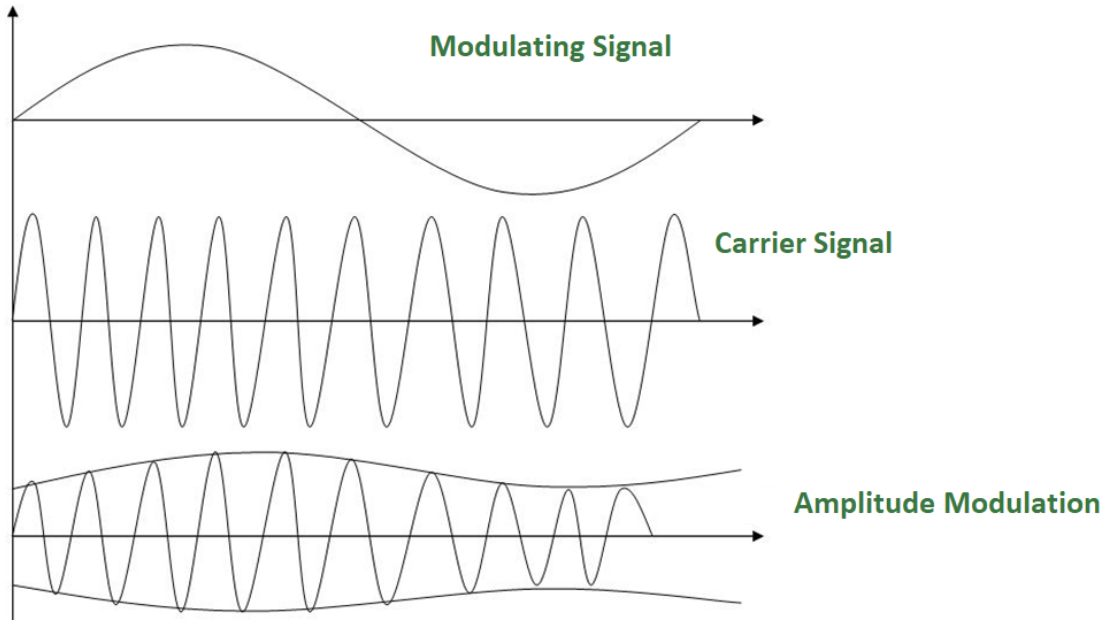


图 5.44: AM 调制

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。

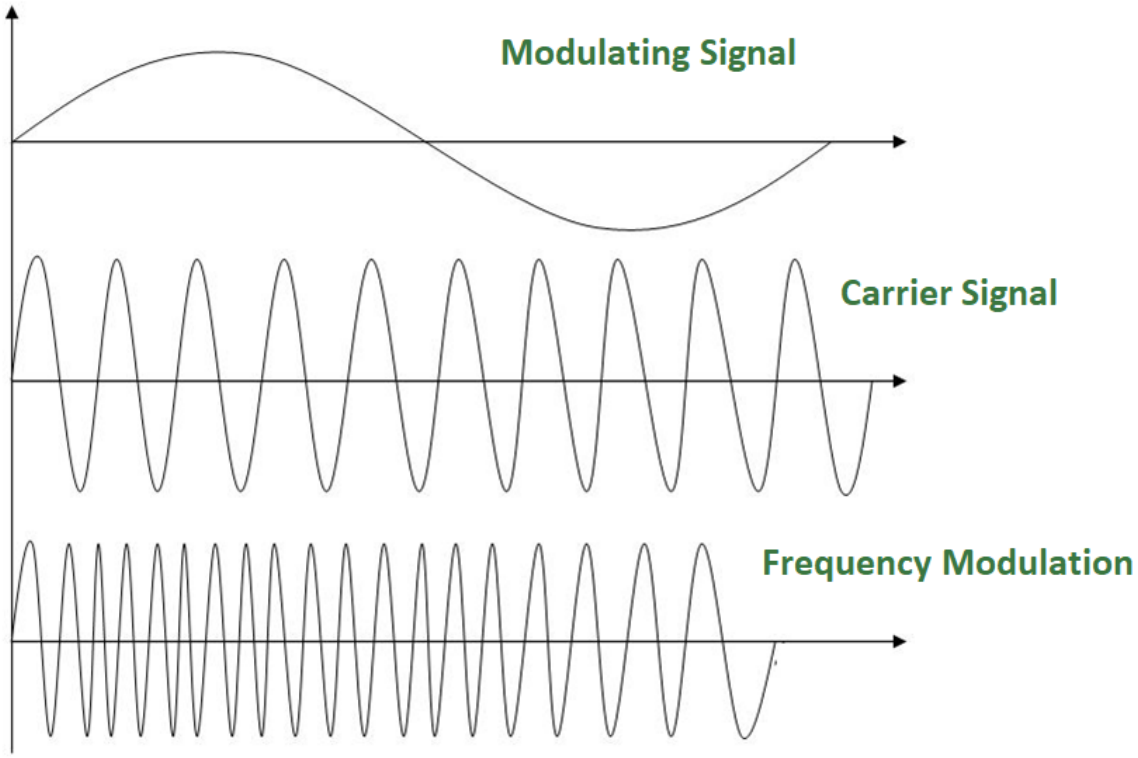


图 5.45: FM 调制

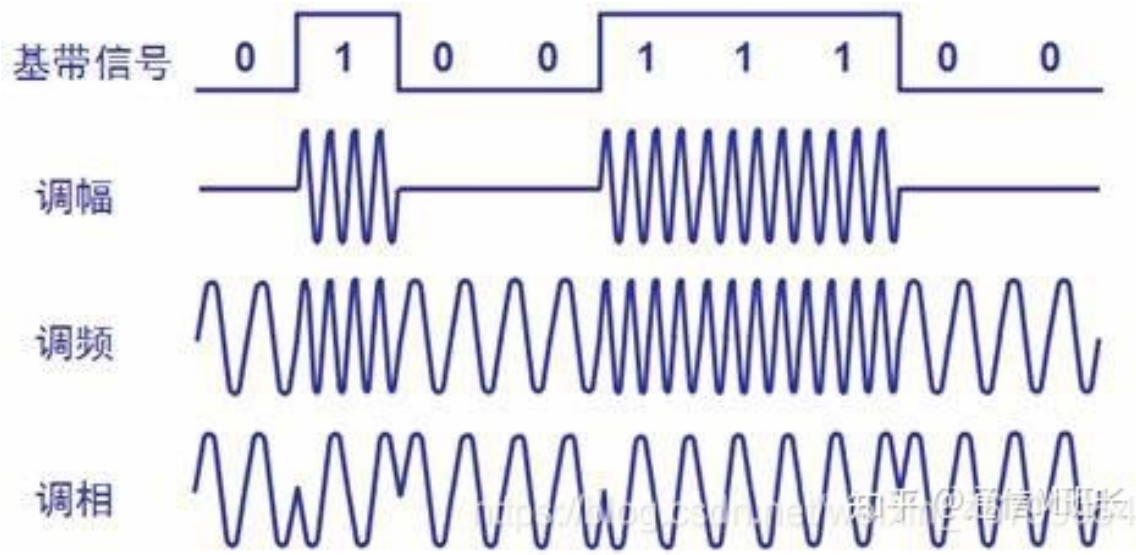
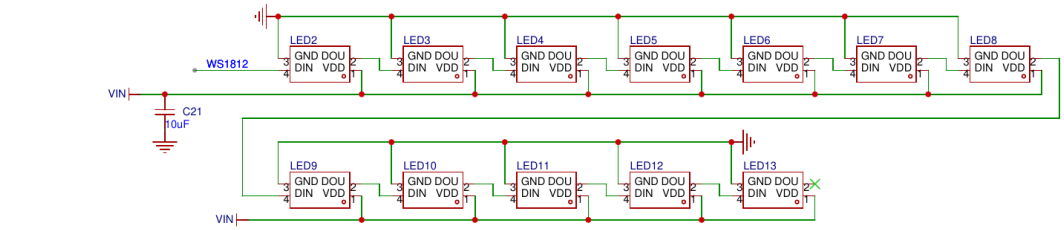


图 5.46: 数字调制

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` 。

5.7 代码实现

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



WS2812单总线全彩--按键背光氛围灯

图 5.47: LED 灯原理图

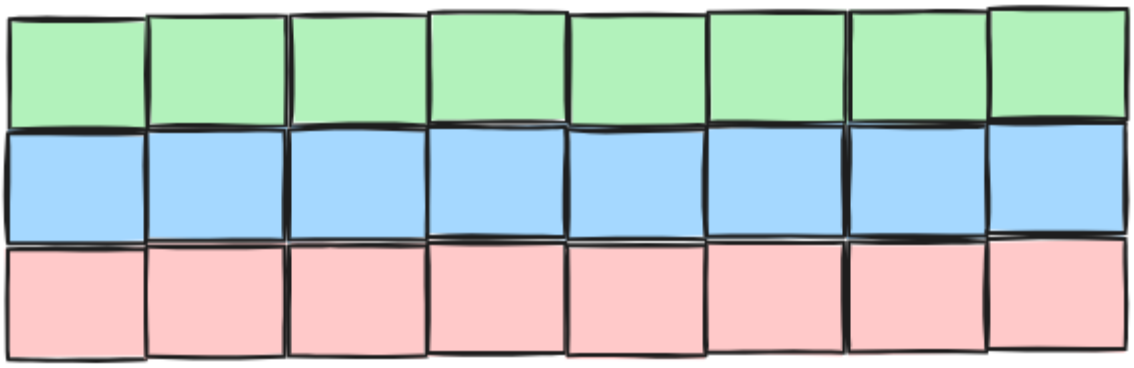


图 5.48: 发送颜色的顺序

点灯

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

    // 将 RGB 值通过通道发送至 LED 灯。点亮灯。
```

```
    ESP_ERROR_CHECK(rmt_transmit(
        led_chan,
        led_encoder,
        led_strip_pixels,
        sizeof(led_strip_pixels),
        &tx_config));
    ESP_ERROR_CHECK(rmt_tx_wait_all_done(led_chan, portMAX_DELAY));

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

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

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

尝试编写代码调用点灯方法，将灯点亮。

第六章 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 6.1: 语音模块电路图

我们来编写初始化 GPIO 引脚的代码。

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

语音模块 *GPIO* 引脚配置

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

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

给语音模块发送数据并播报的代码，通过发送不同的 `uint8_t` 数据，使语音模块播放不同的声音。具体参见语音模块文档。

播报语音代码

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

第七章 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：

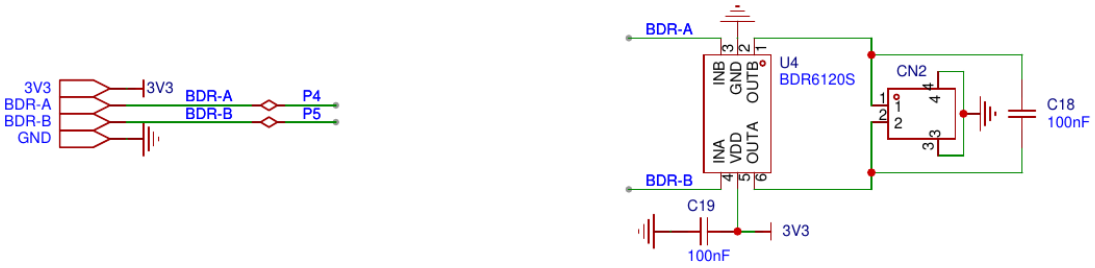


图 7.1: 电机模块电路图

初始化 GPIO 引脚代码

电机 *GPIO* 引脚初始化

```
1  void MOTOR_Init(void)
2  {
3      gpio_config_t io_conf;
4      // 禁用中断
5      io_conf.intr_type = GPIO_INTR_DISABLE;
6      // 设置为输出模式
7      io_conf.mode = GPIO_MODE_OUTPUT;
8      // 设置要用的两个引脚
9      io_conf.pin_bit_mask = ((1ULL << MOTOR_DRIVER_NUM_0) | (1ULL <<
           MOTOR_DRIVER_NUM_1));
10     gpio_config(&io_conf);
11
12     // 最开始都输出低电平，这样就不转
13     gpio_set_level(MOTOR_DRIVER_NUM_0, 0);
14     gpio_set_level(MOTOR_DRIVER_NUM_1, 0);
15 }
```

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);
}
```

第八章 指纹模块

ESP32 使用串口和指纹模块进行通信。电路图如下：

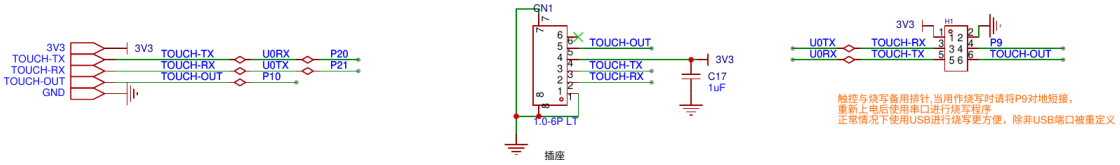


图 8.1: 指纹模块电路图

我们先来写头文件

指纹模块头文件

```h
#ifndef __FINGERPRINT_DRIVER_H_
#define __FINGERPRINT_DRIVER_H_

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

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

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

#define BUF_SIZE (1024)

#define TOUCH_INT GPIO_NUM_8

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

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

/// 获取指纹图像
int get_image(void);
```

```
29
30 /// 获取指纹特征
31 int gen_char(void);
32
33 /// 搜索指纹
34 int search(void);
35
36 /// 读取指纹芯片配置参数
37 void read_sys_params(void);
38
39 #endif
```

然后编写头文件中接口的实现

指纹模块实现代码

```
1  #include "fingerprint_driver.h"
2
3  void FINGERPRINT_Init(void)
4  {
5      /* Configure parameters of an UART driver,
6       * communication pins and install the driver */
7      uart_config_t uart_config = {
8          .baud_rate = ECHO_UART_BAUD_RATE,
9          .data_bits = UART_DATA_8_BITS,
10         .parity = UART_PARITY_DISABLE,
11         .stop_bits = UART_STOP_BITS_1,
12         .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
13         .source_clk = UART_SCLK_DEFAULT,
14     };
15     int intr_alloc_flags = 0;
16
17     ESP_ERROR_CHECK(uart_driver_install(
18         ECHO_UART_PORT_NUM,
19         BUF_SIZE * 2, 0, 0, NULL,
20         intr_alloc_flags));
21     ESP_ERROR_CHECK(uart_param_config(
22         ECHO_UART_PORT_NUM, &uart_config));
23     ESP_ERROR_CHECK(uart_set_pin(
24         ECHO_UART_PORT_NUM,
```

```
        ECHO_TEST_TXD,
        ECHO_TEST_RXD,
        ECHO_TEST_RTS,
        ECHO_TEST_CTS));

    // 中断
    gpio_config_t io_conf;
    io_conf.intr_type = GPIO_INTR_NEGEDGE;
    io_conf.mode = GPIO_MODE_INPUT;
    io_conf.pin_bit_mask = (1ULL << TOUCH_INT);
    io_conf.pull_up_en = 1;
    gpio_config(&io_conf);

    printf("指纹模块初始化成功。\r\n");
}

void get_chip_sn(void)
{
    vTaskDelay(200 / portTICK_PERIOD_MS);
    uint8_t *data = (uint8_t *)malloc(BUF_SIZE);

    // 获取芯片唯一序列号 0x34。确认码=00H 表示 OK；确认码=01H 表示收包有错。
    uint8_t PS_GetChipSN[13] = {0xEF, 0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0x01, 0x00,
         0x04, 0x34, 0x00, 0x00, 0x39};
    uart_write_bytes(ECHO_UART_PORT_NUM, (const char *)PS_GetChipSN, 13);

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

    if (len)
    {
        if (data[6] == 0x07 && data[9] == 0x00)
        {
            printf("chip sn: %.32s\r\n", &data[10]);
        }
    }

    free(data);
}

```

```
64  // 检测是否有手指放在模组上面
65  int get_image(void)
66  {
67      uint8_t *data = (uint8_t *)malloc(BUF_SIZE);
68
69      // 验证用获取图像 0x01，验证指纹时，探测手指，探测到后录入指纹图像存于图像缓冲
        区。返回确认码表示：录入成功、无手指等。
70      uint8_t PS_GetImageBuffer[12] = {0xEF, 0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0x01,
        0x00, 0x03, 0x01, 0x00, 0x05};
71
72      uart_write_bytes(ECHO_UART_PORT_NUM, (const char *)PS_GetImageBuffer, 12);
73
74      int len = uart_read_bytes(ECHO_UART_PORT_NUM, data, (BUF_SIZE - 1), 2000 /
        portTICK_PERIOD_MS);
75
76      int result = 0xFF;
77
78      if (len)
79      {
80          if (data[6] == 0x07)
81          {
82              if (data[9] == 0x00)
83              {
84                  result = 0;
85              }
86              else if (data[9] == 0x01)
87              {
88                  result = 1;
89              }
90              else if (data[9] == 0x02)
91              {
92                  result = 2;
93              }
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(ECHO_UART_PORT_NUM, (const char *)PS_GenCharBuffer, 13);
110 
111     int len = uart_read_bytes(ECHO_UART_PORT_NUM, 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(ECHO_UART_PORT_NUM, (const char *)PS_SearchBuffer, 17);
136 
137     int len = uart_read_bytes(ECHO_UART_PORT_NUM, 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(ECHO_UART_PORT_NUM, (const char *)PS_ReadSysPara, 12);
162
163     int len = uart_read_bytes(ECHO_UART_PORT_NUM, 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 }
```

第九章 蓝牙模块

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

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

Bluedroid

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

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

BLE（Bluetooth Low Energy）

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

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

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

9.1 GATT SERVER 代码讲解

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

9.1.1 头文件

蓝牙功能头文件

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "freertos/FreeRTOS.h"
5 #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。

9.1.2 入口函数

入口函数是 `app_main()` 函数。

入口函数

```
1  void app_main()
2  {
3      esp_err_t ret;
4      // Initialize NVS.
5      // 初始化flash，很重要。
6      ret = nvs_flash_init();
```

```
7     if (ret == ESP_ERR_NVS_NO_FREE_PAGES
8         || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
9         ESP_ERROR_CHECK(nvs_flash_erase());
10        ret = nvs_flash_init();
11    }
12    ESP_ERROR_CHECK(ret);
13
14    esp_bt_controller_config_t bt_cfg = BT_CONTROLLER_INIT_CONFIG_DEFAULT();
15    esp_bt_controller_init(&bt_cfg);
16    esp_bt_controller_enable(ESP_BT_MODE_BLE);
17    esp_bluedroid_init();
18    esp_bluedroid_enable();
19
20    esp_ble_gatts_register_callback(gatts_event_handler);
21    esp_ble_gap_register_callback(gap_event_handler);
22    esp_ble_gatts_app_register(PROFILE_A_APP_ID);
23    esp_ble_gatts_app_register(PROFILE_B_APP_ID);
24    esp_ble_gatt_set_local_mtu(512);
25 }
```

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

初始化 *Flash*

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

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

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

初始化蓝牙控制器

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

接下来，控制器使能为 BLE 模式。

使能为 *BLE* 模式

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

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。

9.1.4 应用程序配置文件 (APPLICATION PROFILES)

如下图所示，GATT 服务器示例应用程序通过使用应用程序配置文件来组织。每个应用程序配置文件描述了一种分组功能的方式，这些功能是为一个客户端 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。

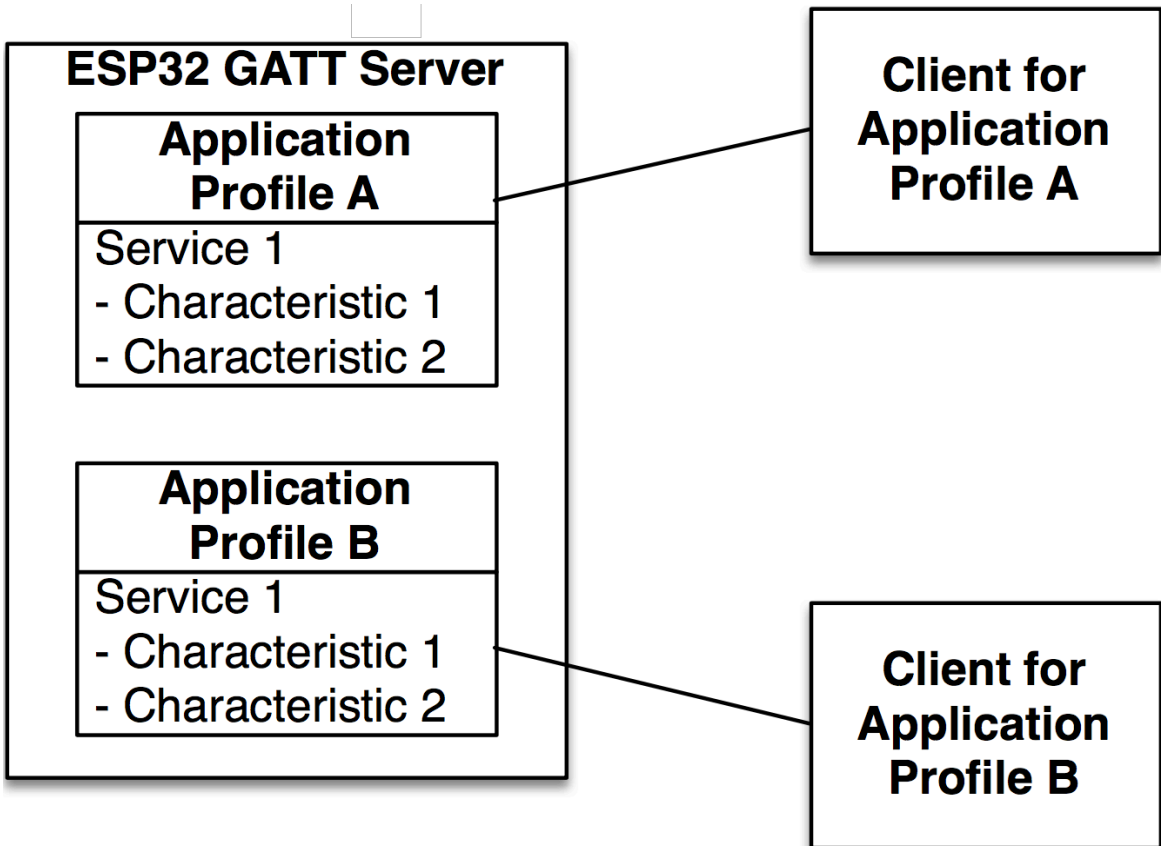


图 9.1: GATT 服务器

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

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

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

结构实现是：

结构体定义

```
1  struct gatts_profile_inst {
2      esp_gatts_cb_t gatts_cb;
3      uint16_t gatts_if;
4      uint16_t app_id;
5      uint16_t conn_id;
6      uint16_t service_handle;
7      esp_gatt_srvc_id_t service_id;
8      uint16_t char_handle;
9      esp_bt_uuid_t char_uuid;
10     esp_gatt_perm_t perm;
11     esp_gatt_char_prop_t property;
12     uint16_t descr_handle;
13     esp_bt_uuid_t descr_uuid;
14 };
```

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

为不同的应用注册回调函数

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

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

使用应用 *ID* 注册配置文件

```
esp_ble_gatts_app_register(ESP_APP_ID);
```

9.1.5 gatts_event_handler 函数

gatts_event_handler

```
void gatts_event_handler(
    esp_gatts_cb_event_t event,
    esp_gatt_if_t gatts_if,
    esp_ble_gatts_cb_param_t *param)
{
    /* If event is register event, store the gatts_if for each profile */
    if (event == ESP_GATTS_REG_EVT)
    {
        if (param->reg.status == ESP_GATT_OK)
        {
            heart_rate_profile_tab[PROFILE_APP_IDX].gatts_if = gatts_if;
        }
        else
        {
            ESP_LOGE(GATTS_TABLE_TAG, "reg app failed, app_id %04x, status %d",
                    param->reg.app_id,
                    param->reg.status);
            return;
        }
```

```
20      }
21      do
22      {
23          int idx;
24          for (idx = 0; idx < PROFILE_NUM; idx++)
25          {
26              /* ESP_GATT_IF_NONE, not specify a certain gatt_if, need to call
                    every profile cb function */
27              if (gatts_if == ESP_GATT_IF_NONE
28                 || gatts_if == heart_rate_profile_tab[idx].gatts_if)
29              {
30                  if (heart_rate_profile_tab[idx].gatts_cb)
31                  {
32                      heart_rate_profile_tab[idx].gatts_cb(event, gatts_if, param);
33                  }
34              }
35          }
36      } while (0);
37  }
```

第 32 行是最关键的，就是执行回调函数。所以我们接下来实现这个回调函数。

9.1.6 gatts_profile_event_handler 函数

gatts_profile_event_handler

```
1   void gatts_profile_event_handler(
2       esp_gatts_cb_event_t event,
3       esp_gatt_if_t gatts_if,
4       esp_ble_gatts_cb_param_t *param)
5   {
6       switch (event)
7       {
8       case ESP_GATTS_REG_EVT:
9       {
10          esp_ble_gap_set_device_name(SAMPLE_DEVICE_NAME);
11          esp_ble_gap_config_adv_data_raw(raw_adv_data, sizeof(raw_adv_data));
12          adv_config_done |= ADV_CONFIG_FLAG;
13          esp_ble_gap_config_scan_rsp_data_raw(raw_scan_rsp_data, sizeof(
                raw_scan_rsp_data));
```

```
14          adv_config_done |= SCAN_RSP_CONFIG_FLAG;
15          esp_ble_gatts_create_attr_tab(
16              gatt_db,
17              gatts_if,
18              HRS_IDX_NB,
19              SVC_INST_ID
20          );
21      }
22      break;
23      case ESP_GATTS_READ_EVT:
24          break;
25      case ESP_GATTS_WRITE_EVT:
26          if (!param->write.is_prep)
27          {
28              esp_log_buffer_hex("接收到的数据: ", param->write.value, param->write.
                    len);
29              if (param->write.value[0] == 'a'
30                &&  param->write.value[1] == 't'
31                &&  param->write.value[2] == 'g'
32                &&  param->write.value[3] == 'u'
33                &&  param->write.value[4] == 'i'
34                &&  param->write.value[5] == 'g'
35                &&  param->write.value[6] == 'u')
36              {
37                  printf("通过蓝牙开锁成功\r\n");
38                  MOTOR_Open_lock();
39              }
40              if (heart_rate_handle_table[IDX_CHAR_CFG_A] == param->write.handle
41                  && param->write.len == 2)
42              {
43                  uint16_t descr_value = param->write.value[1] << 8 | param->write.
                        value[0];
44                  if (descr_value == 0x0001)
45                  {
46                      uint8_t notify_data[] = "atguigu";
47                      // the size of notify_data[] need less than MTU size
48                      esp_ble_gatts_send_indicate(
49                          gatts_if,
50                          param->write.conn_id,
51                          heart_rate_handle_table[IDX_CHAR_VAL_A],
52                          sizeof(notify_data),
```

```
                    notify_data,
                    false);
            }
            else if (descr_value == 0x0002)
            {
                // ...
            }
            else if (descr_value == 0x0000)
            {
                // ...
            }
            else
            {
                // ...
            }
        }
        /* send response when param->write.need_rsp is true*/
        if (param->write.need_rsp)
        {
            esp_ble_gatts_send_response(
                gatts_if,
                param->write.conn_id,
                param->write.trans_id,
                ESP_GATT_OK,
                NULL);
        }
    }
    else
    {
        /* handle prepare write */
        example_prepare_write_event_env(gatts_if, &prepare_write_env, param);
    }
    break;
case ESP_GATTS_EXEC_WRITE_EVT:
    example_exec_write_event_env(&prepare_write_env, param);
    break;
case ESP_GATTS_MTU_EVT:
case ESP_GATTS_CONF_EVT:
case ESP_GATTS_START_EVT:
    break;
case ESP_GATTS_CONNECT_EVT:
```

```
94          ESP_LOGI(GATTS_TABLE_TAG, "ESP_GATTS_CONNECT_EVT, conn_id = %d", param->
                connect.conn_id);
95          esp_log_buffer_hex(GATTS_TABLE_TAG, param->connect.remote_bda, 6);
96          esp_ble_conn_update_params_t conn_params = {0};
97          memcpy(conn_params.bda, param->connect.remote_bda, sizeof(esp_bd_addr_t)
                );
98          /* For the iOS system, please refer to Apple official documents about
                the BLE connection parameters restrictions. */
99          conn_params.latency = 0;
100         conn_params.max_int = 0x20; // max_int = 0x20*1.25ms = 40ms
101         conn_params.min_int = 0x10; // min_int = 0x10*1.25ms = 20ms
102         conn_params.timeout = 400; // timeout = 400*10ms = 4000ms
103         // start sent the update connection parameters to the peer device.
104         esp_ble_gap_update_conn_params(&conn_params);
105         break;
106     case ESP_GATTS_DISCONNECT_EVT:
107         ESP_LOGI(GATTS_TABLE_TAG, "ESP_GATTS_DISCONNECT_EVT, reason = 0x%x",
                param->disconnect.reason);
108         esp_ble_gap_start_advertising(&adv_params);
109         break;
110     case ESP_GATTS_CREAT_ATTR_TAB_EVT:
111     {
112         if (param->add_attr_tab.status != ESP_GATT_OK)
113         {
114             ESP_LOGE(GATTS_TABLE_TAG, "create attribute table failed, error code
                    =0x%x", param->add_attr_tab.status);
115         }
116         else if (param->add_attr_tab.num_handle != HRS_IDX_NB)
117         {
118             ESP_LOGE(GATTS_TABLE_TAG, "create attribute table abnormally,
                    num_handle (%d) \
119                          doesn't equal to HRS_IDX_NB(%d)",
120                     param->add_attr_tab.num_handle, HRS_IDX_NB);
121         }
122         else
123         {
124             ESP_LOGI(GATTS_TABLE_TAG, "create attribute table successfully, the
                    number handle = %d\n", param->add_attr_tab.num_handle);
125             memcpy(heart_rate_handle_table, param->add_attr_tab.handles, sizeof(
                    heart_rate_handle_table));
126             esp_ble_gatts_start_service(heart_rate_handle_table[IDX_SVC]);
```

```
127         }
128         break;
129     }
130     case ESP_GATTS_STOP_EVT:
131     case ESP_GATTS_OPEN_EVT:
132     case ESP_GATTS_CANCEL_OPEN_EVT:
133     case ESP_GATTS_CLOSE_EVT:
134     case ESP_GATTS_LISTEN_EVT:
135     case ESP_GATTS_CONGEST_EVT:
136     case ESP_GATTS_UNREG_EVT:
137     case ESP_GATTS_DELETE_EVT:
138     default:
139         break;
140     }
141 }
```

第十章 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 }
```

第十一章 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 }
```

第十二章 OTA 功能

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.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"
#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
```

```
20 #define EXAMPLE_NETIF_DESC_STA "example_netif_sta"
21
22 void get_sha256_of_partitions(void);
23 void ota_task(void);
24
25 #endif
```

然后编写对应的实现代码

ota_driver.c

```
1  #include "ota_driver.h"
2
3  static const char *bind_interface_name = EXAMPLE_NETIF_DESC_STA;
4
5  static const char *TAG = "OTA任务: ";
6  extern const uint8_t server_cert_pem_start[] asm("_binary_ca_cert_pem_start");
7  extern const uint8_t server_cert_pem_end[] asm("_binary_ca_cert_pem_end");
8
9  esp_netif_t *get_example_netif_from_desc(const char *desc)
10 {
11     esp_netif_t *netif = NULL;
12     while ((netif = esp_netif_next(netif)) != NULL)
13     {
14         if (strcmp(esp_netif_get_desc(netif), desc) == 0)
15         {
16             return netif;
17         }
18     }
19     return netif;
20 }
21
22 static void print_sha256(const uint8_t *image_hash, const char *label)
23 {
24     char hash_print[HASH_LEN * 2 + 1];
25     hash_print[HASH_LEN * 2] = 0;
26     for (int i = 0; i < HASH_LEN; ++i)
27     {
28         sprintf(&hash_print[i * 2], "%02x", image_hash[i]);
29     }
```

```
    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:
```

```
70          ESP_LOGD(TAG, "HTTP_EVENT_ON_FINISH");
71          break;
72      case HTTP_EVENT_DISCONNECTED:
73          ESP_LOGD(TAG, "HTTP_EVENT_DISCONNECTED");
74          break;
75      case HTTP_EVENT_REDIRECT:
76          ESP_LOGD(TAG, "HTTP_EVENT_REDIRECT");
77          break;
78      }
79      return ESP_OK;
80  }
81
82  void ota_task(void)
83  {
84      ESP_LOGI(TAG, "OTA任务开始了。\r\n");
85      esp_http_client_config_t config = {
86          .url = "http://192.168.232.162:8070/led_strip.bin",
87          .crt_bundle_attach = esp_crt_bundle_attach,
88          .event_handler = _http_event_handler,
89          .keep_alive_enable = true,
90      };
91
92      esp_https_ota_config_t ota_config = {
93          .http_config = &config,
94      };
95      ESP_LOGI(TAG, "Attempting to download update from %s", config.url);
96      esp_err_t ret = esp_https_ota(&ota_config);
97      if (ret == ESP_OK)
98      {
99          ESP_LOGI(TAG, "OTA Succeed, Rebooting...");
100         esp_restart();
101     }
102     else
103     {
104         ESP_LOGE(TAG, "Firmware upgrade failed");
105     }
106 }
```